R-ARCH-104
R. Schulte

Strategic Analysis Report
28 April 1997

Architecture and Planning for Modern Application Styles

Management Summary

Most enterprises spend too little time thinking about the topology of their applications. This lack of attention results in application systems that take too long to deploy, cost too much, are difficult to maintain and expand, have needless redundancy in program logic and data, and suffer from low-quality data. In many enterprises, “architecture” means only a short list of standard products that have been approved for use, and even this form of architecture is unevenly implemented. Although it is often useful to minimize the diversity of software tools that are bought and deployed, short-list architectures do not offer application developers enough information to make successful design decisions.

A good architecture includes topology guidelines for partitioning and targeting both data and processes. Partitioning means dividing program logic and data into segments; targeting refers to the physical placement of those program and data segments onto one or more systems. Topology concerns a number of runtime deployment issues such as: What runs on the desktop client? What runs on the server? If something is on a server, which server?

The most common mistake in application topology is the failure to distinguish between local, intra-application architectural “blueprint” issues and macrocosmic, multiapplication “city planning” issues. For example, two- and three-tier architectures are useful blueprint concepts when designing individual parts of an application. However, the concepts do not apply to the relationships among whole application systems that are implemented by different divisions or departments. Such enterprisewide issues are addressed by macro-level topologies such as data warehouses, message brokers and organized batch transaction reconciliation architectures.

Successful modern IT architectures leverage the appropriate use of the following fundamental design principles:

- Modularity
- Encapsulation
- Reuse or sharing of functions
- Separation of presentation (user interface) logic from flow control, business rules and data access logic

- Use of server-centric processing to minimize software distribution problems and to maximize code reuse, and
- Incremental adoption of any desired changes in application design style or middleware.

These principles are applied on a fine-grained level within an application program using the concept of runtime component software. The same principles are also applied on a medium-grained level ("services") between related application modules in one application system and on a coarse-grained level between multiple whole, independently designed application systems.

The single most valuable thing that an enterprise can do to ensure a durable and extensible application portfolio is to use a service-oriented architecture for most new midsize or large applications. A service-oriented architecture maximizes code reuse and minimizes the redundancy of logic and data by organizing functions into shareable, encapsulated modules that can be accessed from multiple client (requesting) applications.

In this *Strategic Analysis Report,* we analyze the benefits and limitations of all of the major topological trends in modern application design, including two-, three- and multitier architectures; service-oriented architectures; organized and unorganized batch data reconciliation; shared data; real-time data integration; data warehouses; operational data stores (ODSs); and message brokers. We address the following SSA Key Issues:

- How will technology and business trends affect architectural trade-offs during the next five years?
- What is three-tier computing and why does it matter?
- What impact will objects and components have on data and process topology?
- How and when will mainstream enterprises derive practical benefits from message brokering?
- What are the trade-offs between database gateways and program-to-program middleware?
- How can new applications best be integrated with purchased applications and legacy systems?

Among the major Strategic Planning Assumptions that derive from our research into these issues are the following:

- Ninety percent of all new C/S applications will be multitier (three or more tiers) in 2001, up from 40 percent in 1996 (0.7 probability).
- By 1999, object request brokers (ORBs) and object transaction middleware (OTM) will be the dominant method of program-to-program communication for new applications (0.7 probability).
- Service-oriented topologies will account for more than one-third of new, mission-critical operational applications by 2001, up from less than 15 percent in 1997 (0.7 probability).
- Despite enterprise data modeling techniques and advances in database gateway middleware, the goal of sharing data directly among heterogeneous operational applications will remain unattainable during and beyond our five-year planning horizon, i.e., 2002 (0.9 probability).
- Until 2000, except for clearly focused applications, ODS deployment will be difficult and suitable only for the skillful and strategically minded who can justify the costs (0.8 probability).

- Through 2002, even in their most successful installations, ODSs and shared databases will never hold more than 25 percent of the data of record for any large enterprise (0.8 probability).
- By 2001, more than half of all large enterprises will have some form of message broker in production (0.7 probability).
- Message brokers, data warehouses, ODSs, multitier architectures and service-oriented architectures are complementary notions; none of them will replace another during our planning horizon (0.9 probability).
- No compelling architectural leadership will emerge from any individual vendor or consortium through 2002, compelling users to plan, assemble and manage their own architectures using pieces from many sources (0.9 probability).

CONTENTS

FIGURES

1.0 Introduction to IT Architecture

1.1 Challenges

The goal of this *Strategic Analysis Report* is to provide practical guidance on the subject of application topology in large and midsize enterprises. Topology is one of the most crucial aspects of architecture and refers the issues of data and process partitioning and targeting. Partitioning means dividing program logic and data into segments; targeting refers to the physical placement of those program and data segments onto one or more systems. Topology concerns a number of runtime deployment issues such as:

- What runs on the desktop client?
- What runs on the server? If something is on a server, which server?
- Within a system, how will the program logic modules relate to each other and to various sections of the data?

Topology issues may be addressed by two-tier or three-tier architectures, data warehouses, message brokers or any of several other design models. In this *Strategic Analysis Report*, we discuss these concepts in detail. There is more to architecture than just topology, however (although we focus here only on topology). There is no industry consensus on the nature or even the purpose of an IT "architecture." The term is applied to many different things, including:

- *A short list of products* to be used in IT projects, such as the approved hardware platforms, operating systems (OSs), DBMSs, development tools, middleware products and packaged applications. An enterprise may be said to base its architecture on RS/6000s, AIX, Oracle, Forte, CICS, SAP R/3 or some combination of these and other specific products. The goals of short-list architectures are the following:
  - To maximize application portability or interoperability through technology consistency across the enterprise
  - To reduce the number of redundant products that are used so that the technical staff does not have to maintain knowledge of many products
  - To save effort in the procurement process because, after the initial product selection is made, subsequent purchases will not require a reinvestigation of the competing alternatives
  - To achieve economies of scale by buying large quantities of a product from a single vendor (e.g., acquire an enterprisewide license)
- *A list of formal "open systems" standards,* such as XPG/4, Posix 1003.1, CORBA, ANSI SQL, ANSI C, ANSI COBOL or TCP/IP. This is an attempt to provide consistent technology across an enterprise without locking the enterprise into a particular vendor in any category. Open systems are based on the idealistic premise that portability and interoperability can be achieved across many different base products as long as they all conform to the same multivendor interface and protocol specifications. However, in many areas of IT, this premise has not been fulfilled in practice because the standards have been slow to develop and are incomplete, and vendors have continued to deviate from a strict implementation of the standards. Interest in open systems peaked about four years ago, and most user enterprises, even those that were previously the most enthusiastic in their pursuit of this strategy, have since reduced their efforts and their expectations in this direction. Most enterprises continue to pay attention to the importance of de facto standards, but many now place relatively little

reliance on official standards unless they also happen to be (or appear likely to become) de facto standards.

- *A specification of the functions and interfaces of the component parts of a system.* An architecture in this sense is essentially the high-level design of a particular thing, such as an application system, a program or a system software product. An “architected” system is one that has a formal, abstract specification of some type, in contrast to a nonarchitected system that is directly implemented without using an explicit model.
- *Any of a wide array of other general guidelines for analysis and design,* especially those that employ enterprise data, process or organization models. Some architectures specify the methodology of the design process. In other cases, the architecture is a template or specification for the data, process or organization model(s) that are produced by executing the design methodology. Such models can be aimed at the conceptual, logical or physical levels. It is difficult to generalize about the nature of such architectures because they vary so widely, but their general goals are usually to facilitate the development of more flexible and durable application systems by minimizing overlap and maximizing reuse of design, data and code. Such architectures are part of a formal approach to application design and often relate to concepts such as information engineering and computer-aided software engineering (CASE).

These “architectures” are not mutually exclusive; an enterprise may pursue one or all of them. However, the goal of providing good architecture for IT is as elusive as it is attractive. The following are some of the numerous obstacles that continuously thwart the efforts of users to implement architectures:

- Formal architecture projects are often characterized by their lack of clear deliverables, fixed deadlines and real management commitment. Team members who are responsible for developing an architecture may not even have an understanding of which type of architecture they are supposed to generate. Joint responsibility often translates into no responsibility.
- Architectural guidelines are difficult to enforce, particularly now that application development and management are usually dispersed across different groups and different geographical locations. IT is almost never concentrated within a single line of control in the organization hierarchy. It is difficult even to communicate the contents of an architecture across a large organization. It is especially difficult to get voluntary compliance with guidelines that maximize the overall, long-range corporate good but cause an increase in near-term expense or inconvenience to a local department or development team.

As a result, many architectures degenerate into shelfware, i.e., never-used documents that serve only to answer an objection when someone asks if there is a coherent plan behind some aspect of IS practice. Nevertheless, many other architectures succeed, particularly those that do the following: focus on the most critical aspects of IT practice; provide realistic, simple, understandable guidelines; can be implemented incrementally without having to throw away vast amounts of legacy code and databases; and provide tangible returns in a short time frame.

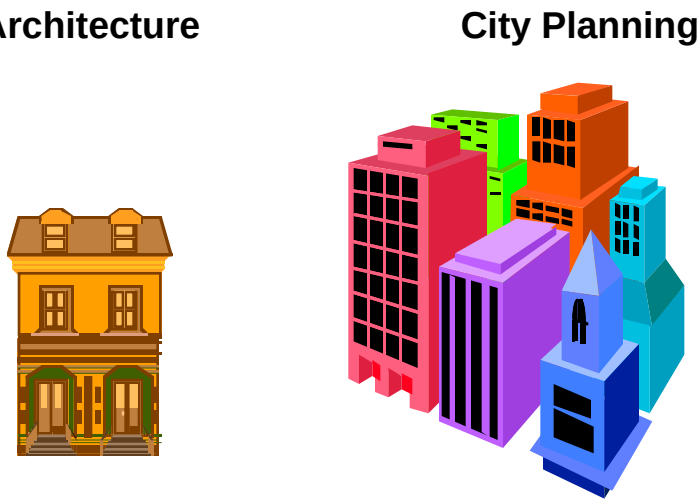
Here, we aim to improve the practice of IT architecture by clarifying topology issues that affect the design and deployment of many kinds of application systems. Topology is always relevant to application design because it deals with issues that are inescapable. All application systems have a topology, although some are determined implicitly and some are determined explicitly. We believe that a clear understanding of topology issues can improve the design of a single application program or one application system, even if

other applications and design teams do not adhere to the same guidelines. However, good topological decisions that affect multiple application systems and multiple groups can do even more – they can make a huge difference to the operation of the whole enterprise.

Most enterprises spend too little time thinking about topology. This lack of attention often results in application systems that cost too much to develop, are difficult to maintain and modify, have needless redundancy in program logic and data, have low-quality data and consume more hardware and network resources than are necessary. Many enterprises have only the short-list form of "architecture," which is usually valuable and worthwhile, but it does provide much help to developers in the design of the application topology. It is certainly a good idea to minimize the unnecessary diversity of technology, interfaces, protocols and products where practical. But many additional benefits can be derived by also using a thoughtful, planned blueprint for data and process partitioning and targeting. In the succeeding sections of this *Strategic Analysis Report,* we describe a variety of topologies and their roles in a successful IT infrastructure.

1.2 Architecture vs. City Planning

IT architectures fail more often than they succeed, and the single biggest source of trouble is the failure to distinguish between architectural "blueprint" level issues and macrocosmic "city planning" issues. The design of a building or an application system is an architectural issue; one set of blueprints can describe the structure in detail because there is one developer. However, it is not practical to enforce a blueprint for a whole city — or for an enterprise IS portfolio — because they are developed at different times by independent organizations (see Figure 1).



Source: Gartner Group

Figure 1. Architecture vs. City Planning

Cities and IS portfolios both evolve through a combination of circumstance and planning. City plans and building codes concentrate on the shared infrastructure, such as roads and pipes, and issues that affect outside parties, such as building height and parking spaces. A functioning city is the result of efforts on three levels: city agencies and utility companies provide the shared infrastructure; developers and contractors assemble the structures; and the occupants provide and arrange the furniture and decide the color of the walls. Similarly, three levels of work go into a functioning IT portfolio: the central IS

department provides infrastructure; application developers and outside vendors provide application systems; and end users may tailor them to their particular requirements. So the central IS department is most effective when it concentrates on the communication infrastructure, application interfaces and shared resources rather than trying to control design details. It is as impractical to mandate one DBMS, one language or one design style for all enterprise applications as it would be to mandate that all buildings in a city use the same layout and the same building materials.

Planning guidelines that affect the work of multiple development teams cannot be detailed and must focus on external interfaces and shared resources.

In the next section of this *Strategic Analysis Report,* we introduce some design principles that apply to all levels of application topology. In the succeeding sections, we review how these notions are applied to each level, beginning with simple applications (Section 3), then complex applications (Section 4) and then to collections of multiple applications (Section 5).

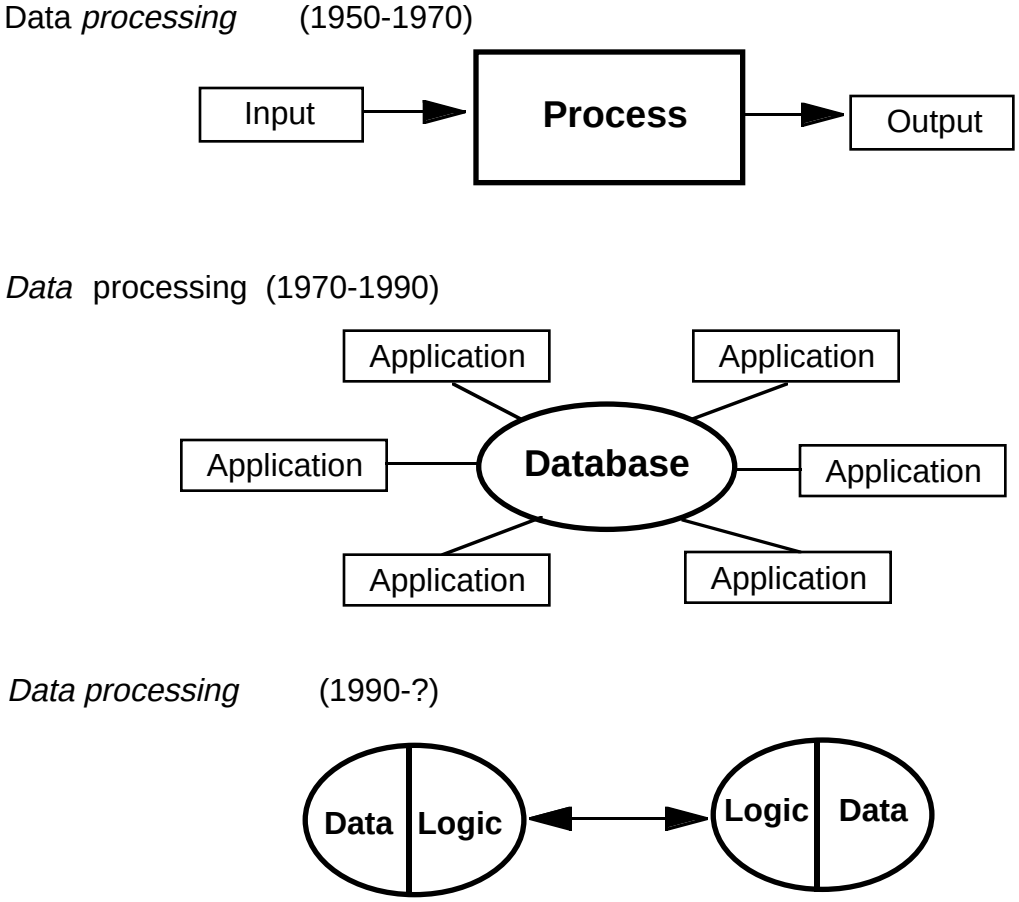
2.0 Trends in Application Design

2.1 Changing Relationships Between Data and Processing Logic

The way that application developers regard the relationship between data and logic has gradually evolved. The thinking has gone through three major steps since the beginning of the IT industry:

- The underlying insight of the early data *processing* era (approximately 1950 to 1970) was that computers can offload the drudgery of repetitive calculations from people. The focus was on logical algorithms; programs were considered durable, but data was something to be transformed. In business applications, most of the work was batch-oriented. The input-process-output diagram (see Figure 2, top section) represented the most common way of envisioning the operation of an application. The validity of this way of modeling computer work has never been repudiated. Input-process-output still works and it still is a useful way to model batch work and parts of some other application styles. However, new ideas on application design have relegated this approach to irrelevance for the majority of applications.
- The primary insight of the *data* processing or “database” era (approximately 1970 to 1990) was that data is a valuable asset in its own right. Data that is properly organized and maintained can be used by multiple application programs for many different purposes (see Figure 2, middle section). Processing logic was considered more ephemeral than the data model. Once the data is modeled and a database is implemented, application programs that use that data can be added and modified for a long time before the data model must change. During the database era, conventional wisdom considered the task of application integration to be a data integration issue and little attention was given to opportunities for reusing processing logic. Application programs related to each other through data, either by sharing the same database or by extracting data from one database to insert or update another database. Most of the critical software integration tools were aimed at shipping data from place to place, or they enabled remote programs to directly access foreign databases. Examples of such tools include file transfer utilities, database replication and propagation tools, database gateways and program generators for extract programs. The data-centric approach is still the best way to model many situations, including data warehousing and other types of decision support. However, for an increasing number of applications, another way of modeling computer work has become more useful (described next).

- We are now in the early stages of what may be termed the *data processing* era, which is driven by the insight that programming logic is often intrinsically related to the data upon which it acts (see Figure 2, lower section). It is often helpful to design data and logic together rather than in separate steps because the integrity constraints, access algorithms (navigational logic) and even business rules may be closely related to a particular set of data. Moreover, logic and data are sometimes even manipulated by the same software infrastructure products. For example, DBMS stored procedures, database triggers, OO programming environments and some ORBs combine data management and logic management into one product.



Source: Gartner Group

Figure 2. Three IT Eras

Many types of applications in the new *data processing* era (post-1990) emphasize the use of program-to-program communication (function integration) more than the data transfer or direct data access mechanisms (data integration) that dominated in the previous era. Functionally integrated application systems interact via some form of message passing, such as RPCs, Sockets, message-oriented middleware, ORBs, TP monitors or OTM. Function integration applies to the architecture within a single application system, and it also applies to the global relationships between multiple application systems. Systems that are based on function integration are designed differently from traditional application systems, particularly in their frequent use of modularity and encapsulation.

2.2 The Growing Importance of the Black Box Metaphor

IT design practices are being reshaped by the increasing use of the black box metaphor and two related design principles:

- Modularity — organizing applications into smaller modules, and
- Encapsulation — hiding data and logic from uncontrolled external access.

Modularization divides big problems into smaller “bite sized” problems that are easier to solve. Encapsulation implies that all communication between modules is done only through controlled, documented interfaces (“contracts”). Encapsulation has the following two benefits:

- It helps protect internal data and code from careless or malicious misuse (i.e., integrity protection)
- It helps shield the external developers from the complexity and dynamism of the contents of a module

External people who want to use a module do not have to see or understand the logic or data model that is inside the module. The module is a “black box” that is accessed only through the defined interfaces. The reuse or sharing of a module implies reuse or sharing of both logic and data. By contrast, the direct reuse or sharing of data (as in data integration) means that one is *not* reusing any program logic.

The principles of modularity and encapsulation are applied on several levels:

- They are applied within a single application program via techniques such as component software (see Section 3), DBMS stored procedures, database triggers and OO programming. Each of these techniques has its own particular benefits and limitations (however, these are middleware issues that go beyond the scope of this *Strategic Analysis Report*).
- The same principles are applied between related application programs in the same application system or group of related application systems using a service-oriented architecture. Services are modules that contain a shareable set of business rules and data access logic that may be invoked by multiple requesting client application modules (see Section 4).
- The principles of modularity and encapsulation can also apply on a macrocosmic level, using program-to-program messages (function transfer) that cross the boundaries of independently designed application systems. In this manner, production databases are encapsulated on a coarser level. They are accessed only by their respective native application programs so that edits, integrity checks and business rules can be reused. Extract and update programs can be made unnecessary because data are indirectly kept in synch through message passing or other program-to-program communication. Section 5.7 explores message brokers which represent an organized approach to macrocosmic encapsulation.

In the remainder of this *Strategic Analysis Report,* we examine the practical implications and use of modularity and encapsulation in more detail.

3.0 Basic Intra-application Architecture

In this section, we examine architectural trends in the basic architecture of online applications (see Figure 3).

Source: Gartner Group

Figure 3. Architecture of Simple Structures

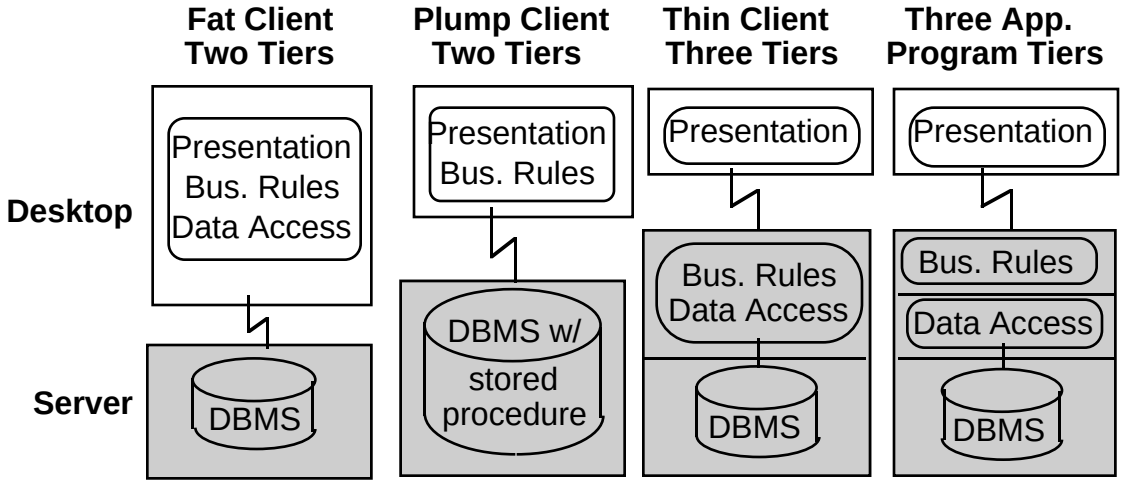
3.1 Overview of Two- and Three-Tier Architectures

The number of software tiers is an intra-application architectural issue. The design tiers of the software determine which functions run on the desktop and which functions run on the server. We define the various configurations of software tiers as follows:

- Two-tier fat-client topologies put all of the application program logic on the desktop and use the server simply to run the DBMS.
- Two-tier “plump client” applications put some of the application logic (usually presentation logic and business rules) on the desktop and rest of the application logic (usually the data access logic and, occasionally, some business rules) on the server in the form of DBMS stored procedures. “Data access logic” is implemented with a data manipulation language (DML) such as SQL.
- Three-tier thin-client applications are those that have some application logic (business rules or data access logic) running on the server in a tier that is separate from the DBMS. The desktop runs presentation logic and, occasionally, some business rules.
- Four-tier applications are those in which flow control functions are implemented in a software layer that is separate from the presentation, business rules and data access logic.

Figure 4 shows the two types of two-tier applications and two types of three-tier applications. In the interest of simplicity, we do not include a diagram of four-tier applications that have an explicit flow control tier. Flow control is the set of instructions that direct the sequence of operations to be performed for a particular task. In any application, logic flow must somehow be controlled within a program and between programs. Flow control between application programs was traditionally addressed by OS-related services such as IBM’s MVS Job Control Language (JCL), Digital Equipment’s VMS Digital Control Language (DCL) and Unix shell scripts. Flow control within an application program is usually left to the developer and programming tools, including the 3GL compiler (e.g., “do” loops, “perform” commands and if-then-else constructs). The nature of flow control has changed from fairly predictable, bounded sequences of steps to complex, unpredictable series of events. Flow control in a C/S application is sometimes implemented in the application program or it may be implemented in middleware or in a separate tier that may be implemented with a scripting language whose sole purpose is flow control. When flow control is in its own

tier, it may be more modular, flexible and easier to maintain than when it is embedded in the business rules.



Source: Gartner Group

Figure 4. Two- and Three-Tier Architectures

Most of the first user-written C/S applications used two-tier fat client topologies. As experience with C/S grew, more two-tier applications began to put some of the application logic on the server in the form of stored procedures. Three-tier “thin client” applications are rapidly becoming the dominant mode of C/S processing because of the growth of the World Wide Web (browsers usually are thin clients), the migration of vendor-written applications to three-tier topologies and improvements in multitier development tools and middleware (see Section 3.3). It is important to note that some of the processing paths within an application may be two-tier and others can be three-tier.

We do not recommend a rigid approach to partitioning the logic. It is likely that some business rules and flow control will find their way into presentation modules. It is also likely that some business rules will be embedded in a data access layer, or that some DML statements will be coded into a business logic layer. These exceptions are often good design. Purity can be bad design, although there are no simple rules as to when functions should be put into other layers.

A data access layer will be implemented differently on each platform. On MVS, it might be done as COBOL subroutines linked into an IMS/TM or CICS application program. In this case, the distinction between data logic and business rules may exist just at design time, because at runtime, the code may be linked into each transaction program. On PCs or Unix, the more common techniques are the following:

- User-written code, e.g., C++ classes. ISVs often do it this way so that they can readily port their products across several different DBMSs. However, this is generally not worthwhile for user enterprises because it is requires so much discipline on the part of the application developers.
- Data instantiation facilities that are built into a wide range of OO development/runtime tools, from Sybase’s PowerBuilder to Gemstone’s Gemstone, IBM’s Visual Age and Next Software’s Nextstep. Middleware such as Microsoft’s DCOM and Iona Technologies’ Orbix ORB also have services that load some kinds of data into components so that the component logic sometimes does not have to

execute traditional embedded DML statements to read or write data. This is a significant change in application design.

3.2 The Accelerating Use of Three-Tier Architectures

Recent trends in the software market have caused us to update our planning assumption regarding the spread of three-tier computing. Users are demanding the flexibility of three-tier topologies more broadly than we had anticipated, and vendors are lowering the cost and complexity of three-tier computing faster than we had expected. *Ninety percent of new C/S applications will be three or more tiers by 2001 (0.7 probability).*

We base our revised Strategic Planning Assumption on the following four general trends and events:

- *Web technology is spreading rapidly into many types of applications.* The Web is almost entirely a thin-client/three-tier phenomenon in which the browser executes only presentation logic. Even with the addition of applets, most of the business rules and data access logic will run on the server. Although browser applications technically can be two-tier, few or none are now, and only a minority will be in the future. Using a browser does not automatically make application design simple, but it acts as a catalyst for developers to make their server-side logic more efficient than it would be in a two-tier approach. To the extent that browser applications supplant traditional C/S GUIs, the number of two-tier applications will drop correspondingly. Furthermore, Web browsers are only one of many alternative front ends that cannot run traditional two-tier C/S applications. Other such front ends include hand-held appliances, interenterprise messaging and some point-of-sale devices and voice response units.
- *Packaged applications are moving rapidly to three-tier architectures.* Some of the major C/S application packages, including those from Baan and SAP, have been based on three-tier architectures for years. PeopleSoft, which had been primarily two-tier, announced a migration to three-tier applications in September 1996. It is reimplementing its applications in stages using a message-based infrastructure centered on BEA Systems' Tuxedo middleware product. Geac Computer (formerly Dun & Bradstreet Software Services) and Hyperion Software are also beginning to move from hybrid, mostly two-tier designs, to server-centric three-tier designs.
- *Oracle embraced three-tier topologies in its Network Computing Architecture (NCA) announcement* (Oct. 1, 1996). Oracle is the most powerful DBMS vendor and previously stressed two-tier computing. The two-tier architecture gives more control to the DBMS because DBMS middleware handles all client-to-server communication and the server application logic is locked into the proprietary stored procedure language. However, Oracle's upcoming NCA products will include full-blown three-tier application server middleware (the Web Application Server) and language-independent "data cartridges" (an enhancement to stored procedures). This will eventually also affect Oracle's packaged C/S applications. These were originally three-tier, but Oracle gradually rewrote them to rely more heavily on stored procedures and two-tier processing. We expect that Oracle will reverse this by rewriting its applications again, this time to exploit the power of NCA and three-tier topologies. Initial deliveries of the new generation of applications will not begin before 1999 (0.7 probability).
- *There has been a significant expansion in the number and quality of three-tier middleware and development tools.* Even Visual Basic (in version 4) and PowerBuilder (in v.5) were upgraded to support a limited form of three-tier computing. We expect that these tools will become more adept at

multitier applications as they exploit DCOM better and as Microsoft enhances DCOM with transactions and other advanced features. Other ORB vendors are also improving their middleware, reducing the complexity of client-to-server program-to-program communication which is the basis of three-tier computing.

The cumulative effect of all these trends is to make it easier and less costly for enterprises to buy or develop three-tier applications. In an increasing number of situations, particularly involving the Web, three-tier applications are less complex than two-tier architectures in terms of the design process, the software acquisition process and the ongoing tuning and administration tasks. At the same time, the inherent architectural power of multitier architectures is becoming increasingly important, even for decision support applications that were often considered to be naturally suited for two-tier topologies.

3.3 The Trade-offs Between Two- and Three-Tier Architectures

Application developers should be aware of the following trade-offs between two- and three-tier designs:

- *Advantages of two tiers:* Historically, the case for the two-tier architecture rested on its simplicity. The middleware and server programming tools (stored procedures), if any were used, were bought from one vendor (i.e., the DBMS provider). In contrast, three-tier architectures sometimes entailed buying middleware from one vendor, a (server) programming tool from another vendor and the DBMS from a third vendor. Furthermore, developers of fat-client applications do not have to decide how to partition the application: everything just goes on the desktop (client). Two-tier/plump-client applications and three-tier applications are more complex in this regard because the developer must decide what functions to put on the server and what functions to put on the desktop.
- *Disadvantages of two-tier fat clients:* Three-tier and plump-client/two-tier applications are more efficient than fat-client applications because they transmit fewer and smaller messages between client and server by executing the business logic processing closer to the location of the data. Such applications make it possible to confine all database I/O within the server and let server-based application logic preprocess the data so that only condensed messages go back to the client.
- *Advantages of three tiers:* The case for the three-tier architecture traditionally was based on the flexibility of general-purpose communication or middleware products (e.g., Sockets, CPI-C, RPCs, messaging systems and TP monitors) compared with stored procedures. Three-tier server applications run under the control of the OS or a middleware runtime environment and are thus more powerful than plump-client/two-tier applications that run in the DBMS and are subject to the limitations of stored procedures.

Stored procedures and the DBMS facilities that manage them at runtime have improved during the past three years, but they are still less flexible than some other forms of middleware in certain key respects. Depending on the particular middleware that is used, a three-tier architecture is superior to a two-tier, stored-procedure-based system in many ways. A three-tier solution:

- Can be dynamically load balanced for high numbers of users and high transaction rates (depending on the middleware);
- Can fail over to a backup copy, for uninterrupted availability;
- Can readily access data outside the local RDBMS, such as real-time data feeds, files or multiple heterogeneous DBMSs;

- Can easily invoke other heterogeneous server application programs, including those not embedded within the DBMS;
- Can readily call out across a network, e.g., invoke a transaction on CICS/MVS through a gateway;
- Is often visible to standard software distribution tools, system monitors, OS utilities and system management tools (depending on the middleware environment); and
- Helps enable DBMS independence (e.g., porting an application between Oracle and Sybase) albeit with some discipline and limitations.

The spread of Web technology and the delivery of better three-tier middleware and development tools are reducing the drawbacks to three-tier architectures. First, the partitioning decision for (three-tier) Web applications is as clear as the partitioning decision for two-tier applications, although the default design choice is reversed. Web applications put most business rules and data access logic on the server rather than on the client (presentation is on the client in both cases). It should be noted, however, that Web servers can be configured many different ways, so there may still be some complicated design issues. Second, the software to enable undemanding three-tier applications can now be bought from one vendor, e.g., Microsoft. As long as the server is NT, Microsoft offers the middleware (DCOM), server-side programming languages and the DBMS. Oracle's NCA product set will provide similar one-stop three-tier shopping by 1998. Finally, large or demanding applications have always been three-tier, and although one-stop shopping will not apply to these kinds of applications during the next five years, the enabling middleware (especially OTM) are becoming better and more available every year.

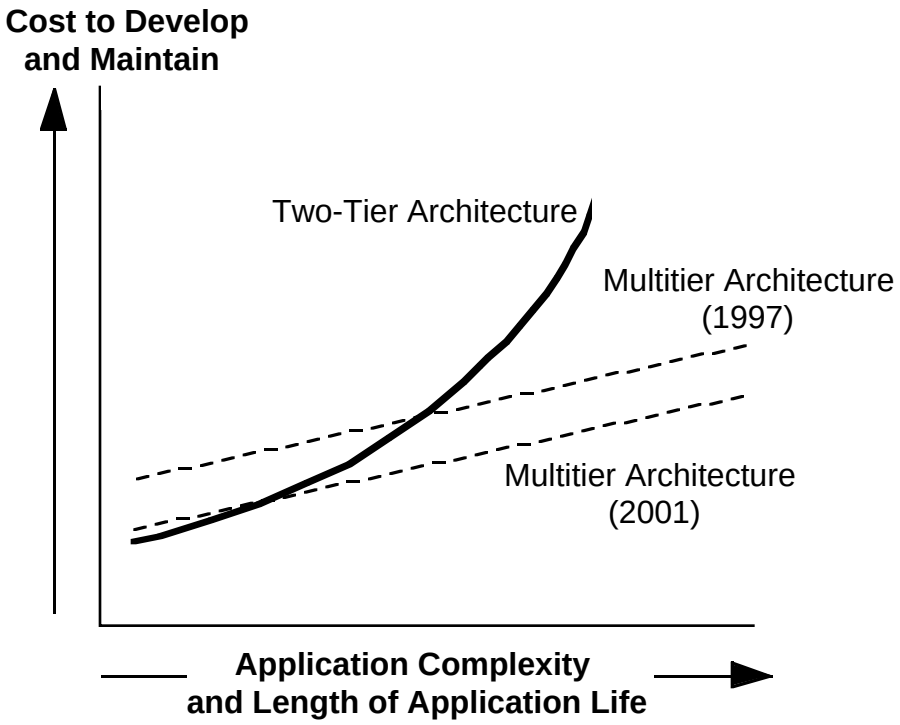
The net effect of these changes will be to push two-tier computing into a niche (see Figure 5). In 2001, two tier topologies will be appropriate only for applications that meet all of the following conditions:

- Use one DBMS, generally on the same LAN;
- Have a moderate-volume workload, generally fewer than 10,000 transactions per day;
- Have little or no interapplication communication, such as access to a mainframe application or interenterprise communication; and
- Always use a desktop PC front end.

Enhanced versions of stored procedures (e.g., data cartridges) will thrive, but their major role will be to supplement three-tier architectures rather than to run the whole server-side application logic.

Three-tier architectures will be appropriate for any application in which one or more of the following conditions exist:

- Most browser user interfaces, i.e., environments that require low administration and software distribution costs, such as network computers;
- Two or more heterogeneous data sources, e.g., two DBMSs or a DBMS and a file system;
- High-volume workload (more than 10,000 transactions per day);
- Need for very high availability or fault tolerance (failover);
- Significant interapplication communication, such as access to a mainframe application or interenterprise communication (message brokers will be appropriate for many of these needs); or
- Upsizing, i.e., the application may grow in time so that one of the previous conditions will apply.



Source: Gartner Group

Figure 5. Relative Merits of Two- and Three-Tier Architectures

In summary, developers must consider the nature of each application and choose between two-tier and three-tier designs on a case-by-case basis. For small, routine applications, a two-tier approach is still faster, simpler and less expensive. For complex or large applications, a three-tier design costs less to develop, maintain and administer. By 1999, however, most applications will be best implemented using three-tier topologies.

3.4 The Role of Component Software in Two- and Three-Tier Architectures

Software is becoming ever larger and more complicated because of the demands for greater functionality, friendlier user interfaces, tighter application integration and increasing distribution across multiple systems. It is therefore increasingly desirable to apply the principles of modularity and encapsulation so that applications can be assembled from multiple, finer-grained (smaller) programs. *Component* software has emerged as the most popular design technique for this purpose.

A runtime software *component* is a dynamically bindable package of one or more programs managed as a unit and accessed through documented interfaces that can be discovered at runtime. In other words, a component is a black box that is particularly friendly to the developer because it is implemented with a formal mechanism for defining and managing the parameters in the program-to-program messages. Components are generally implemented using ORB middleware. Other forms of middleware, such as RPCs, messaging systems and TP monitors, also support encapsulation and modularity (i.e., they can treat a program as a black box), but they do not manage the interfaces in a formal dynamic way. For this reason, we believe that the majority of future C/S applications will use an ORB such as Microsoft's

COM/DCOM or a CORBA-style ORB on either or both the client and server(s). *By 1999, ORBs and OTM will be the dominant method of program-to-program communication for new applications (0.7 probability).*

ORBs and OTM differ from other program-to-program middleware in two respects: formal, dynamic interface management, and explicit component uniqueness:

- *Formal interface management.* An ORB maintains or has access to directories and repositories that document various aspects of application programming interfaces (APIs). These directories and repositories make it possible to defer to runtime certain decisions about how the programs will connect to each other. They specify the name of each supported operation (a verb, or "method name") and its associated parameter list, including the data types of each input and output parameter. In COM/DCOM, there is a Registry for finding server components and a type library for specifying the message contents. In CORBA, an Implementation Repository helps locate server components and an Interface Repository specifies message contents. APIs are usually specified using an interface definition language (IDL), although they do not have to be.

  In theory, client programs could call a previously unfamiliar server component using information gleaned by programmatically inquiring into these runtime repositories to dynamically assemble the call. However, fully exploiting this possibility would require extremely sophisticated client software. In practice, runtime API inspection appears to be useful mostly for the more limited function of letting clients and servers of different development generations work together. For this reason, the clearest benefit of component software is to enable incremental maintenance and incremental enhancements of software (i.e., to make it easier to upgrade one module in a system without having to reinstall or modify other modules). Contrary to conventional wisdom, component reuse (recombining components in a new, unique way) is not the most important benefit of component software, although components do facilitate reuse in some circumstances (particularly for GUI desktop programs).

- *Explicit component uniqueness:* The second characteristic that distinguishes an ORB or OTM from other kinds of software runtime environments is the notion of explicitly managed component uniqueness. The ORB passes a unique identifier for a specific *instance* of a server component to a client program. The client uses this identifier to communicate with the server component and can also pass the identifier to other clients, enabling them to access the identical server component. In CORBA, the identifier is the "object reference"; in COM/DCOM, the interface pointer and optional moniker accomplish about the same thing. There are significant differences in how COM/DCOM and CORBA work, but a discussion of these differences is beyond our scope here. This *Strategic Analysis Report* addresses the general characteristics of components that are common between COM and CORBA.

A component contains logic and data — just as any other program does. But the ORB's unique identifier mechanism makes it easier to apply "object"-related concepts to components. We define an object as a representation of any thing, real or abstract, in terms of a set of operations and data values (which represent its state). Components can implement one or more of the following three different aspects of object technology (see Figure 6):

- *Programming objects.* All well-behaved components are objects in the sense that they encapsulate code and data. ORBs (like RPCs and some other middleware) by their nature hide code and data from external programs because messages into the component enter and exit only through formal, managed APIs. Of course, it is sometimes possible to bypass the ORB to expose data or entry

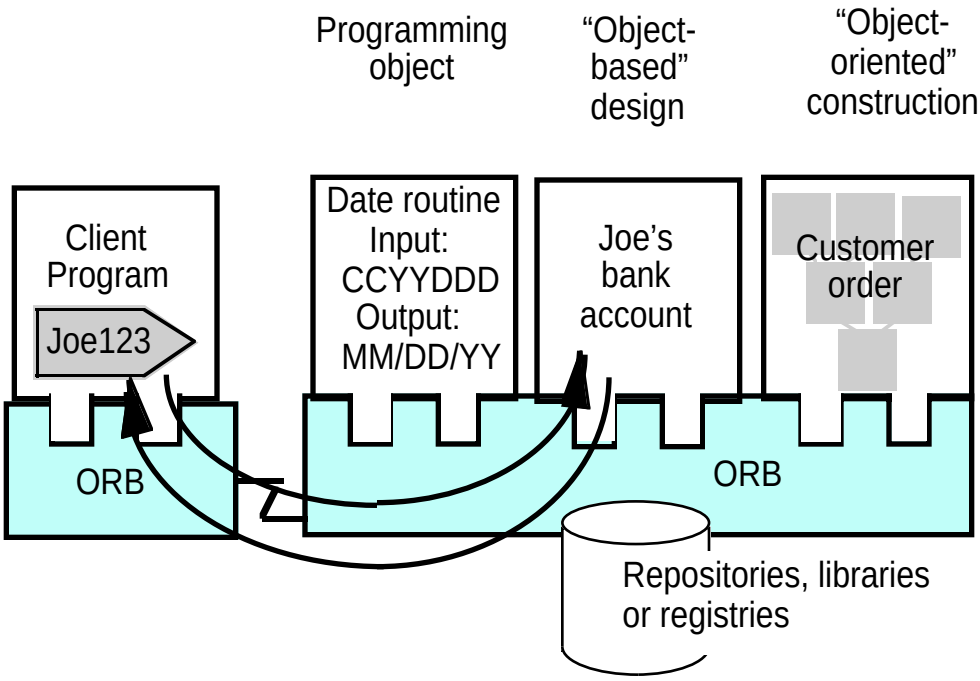
points to external programs, thereby creating a misbehaved component that is not exactly an object. Nevertheless, most components are well behaved in practice, and the words object and component are often used interchangeably to refer to ORB server programs (e.g., a COM/DCOM object means the same as a COM/DCOM component).

A programming object may not model any real world *thing*. For example, a currency conversion component for translating dollars to yen represents an action rather than a noun. Furthermore, a programming object may even not identify one particular target of the operation. For example, if the programmer is wrapping an entire legacy banking application as one "object" (implying that there is a single object identifier), the ORB is not aware of the difference between Joe's bank account and Fred's bank account. In this case, information regarding which account to access is passed as a parameter inside the message, just as it is in traditional middleware, rather than through the ORB's identifier mechanism. Finally, components are sometimes "process objects" that contain only processing logic and ephemeral data with no state (in a technical sense of the word). Strictly speaking, "state" implies that the component remembers changes to its data slots and will behave differently upon subsequent invocations, whereas stateless components do not remember changes between subsequent messages. A date routine may be an example of a ("stateless") process object because its remembered data values may just be static, unchangeable storage.

- *Object-based design.* An ORB is a good way to implement an object-based design metaphor because of its ability to generate or work with unique instance identifiers. "Object based" components operate on one coherent set of data that represents the state of a thing that has been modeled, e.g., Joe's bank account. This is a refinement on the notion of a programming object, and it introduces a different way of thinking about the server component. In a programming object, the identifier merely specifies a unique instance of a program. By contrast, in an object-based design, the identifier refers to a particular program instance and also explicitly specifies the exact subject of the operation. The subject is generally something that is understandable to users and system analysts, not just to the programmer (e.g., again, Joe's bank account). Each object operates on one and only one modeled thing. For example, the object-based component that represents Joe's bank account does not execute operations against Fred's bank account (it would send a message to Fred's bank account object if it needed to operate against it). Object-based objects are not necessarily object-oriented (OO). Applications built of object-based components are programmed differently than applications built of traditional programs or unspecific programming objects. In other words, the fact that ORBs and OTM support the unique instance identifier mechanism encourages (but does not force) developers to design components that are each dedicated to handling only one kind of data — a sometimes-helpful design discipline. Further discussion of object-based design issues goes beyond the scope of this *Strategic Analysis Report.*

- *Object-oriented construction.* Gartner Group defines object orientation (OO) as a paradigm in which objects have encapsulation, classes have inheritance and operations have polymorphism. We further define a class as a specification that defines the operations and data attributes for a set of (like) objects. (Implementation) inheritance is a relationship among classes in which a subclass shares, overrides or supplements operations or data values from one or more superclasses. A subclass is a specialization of one or more superclasses. Finally, polymorphism is the capability of an operation to accept arguments of different or unknown types. Parametric polymorphism executes the same operation on different types. Overloading polymorphism selects appropriate operations according to the type.

Programming objects and object-based components (as described previously) may or may not be constructed using OO technology. However, all OO components can be classified as either programming objects or object-based components. ORBs do not require nor do they directly enable OO-style polymorphism or implementation inheritance. ORBs support only the notions of encapsulation, *interface* inheritance and a restricted version of overloading polymorphism, which amounts to reusing the same verb name and signature in multiple programs (in other words, ORBs reuse only the API specification, not the underlying code). If a component happens to be an OO language object, that fact is not visible to external (client) programs that invoke the component through the ORB. Moreover, the ORB itself is not affected if the server component is written as a monolithic program or if it is written as an OO object with many levels of inheritance.



Source: Gartner Group

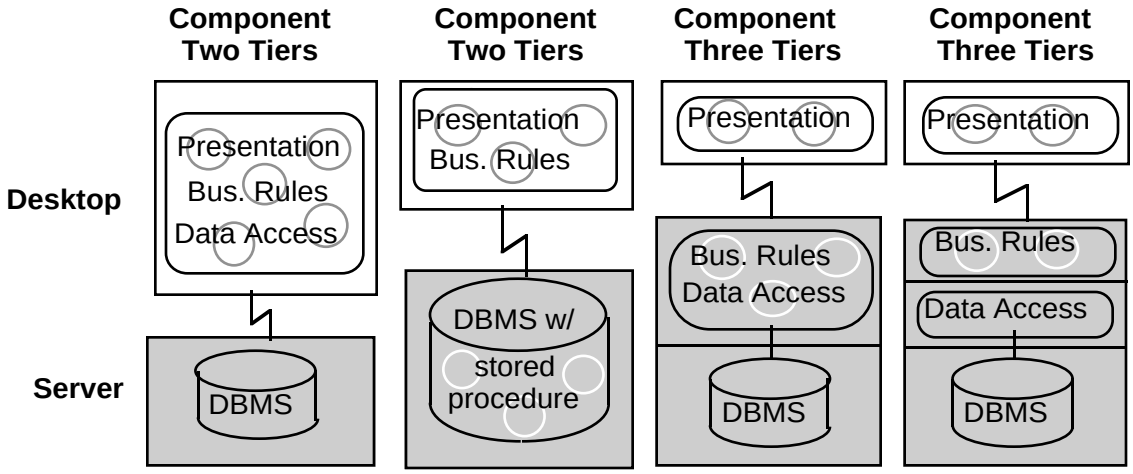
Figure 6. Three Different Notions of Objects

Programming objects, object-based components and even (in theory) OO components can be written in any programming language. In practice, more than 90 percent of all extant components (whether managed by COM/DCOM, CORBA or other ORBs) are written in C++, although Java is increasingly popular and there are also some components written in COBOL, C, Smalltalk, ADA and other languages. However, even components written in C++ or Java may not necessarily exercise a significant amount of OO construction characteristics.

Encapsulation is generally a feature of the runtime software environment (e.g., the ORB). Encapsulation simply means that the current rendering of an instance of data (the object state) is under the exclusive control of a particular component while that component is running. Of course, most object-based and many OO applications store their persistent data in a regular database (usually relational) where it may be used by other programs at other times. In other words, most object-based objects and OO objects do not exclusively own "their" data for all time. However, a few OO environments, particularly those implemented with Smalltalk or ODBMSs, do encapsulate their data on persistent storage as well as in memory.

Although component software will make it easier to expand and maintain applications incrementally, independently designed application modules will not really “plug and play.” Plug and play (or as some irreverently call it, “plug and pray”) means adding or changing part of the functionality of an application without disturbing the other functions that are already deployed. However, the full realization of this ideal is impeded by the intractable semantic and protocol incompatibilities between components that will remain unresolved during the five-year planning horizon. ORBs make it easier to interchange modules of different generations, including modules that are sometimes written by different groups, but ORBs still require that all of the development groups use the same interface definitions. In practical terms, this means that one development team will dictate the interface specifications and other development teams will follow those specifications to the letter if their modules are to work together. *Components may be independently developed, but they are not fully independently designed.*

Component software is a change in middleware technology; it is not a new application topology. Component applications will use the familiar two- or three-tier approaches for application and data partitioning and placement (see Figure 7). Component-based applications still require the same design decisions that apply to applications built with traditional middleware such as RPCs, messaging systems, Sockets or TP monitors. The component application developer must still decide whether to use two, three or more tiers, whether to use stored procedures, whether the desktop should be a fat or thin client, whether to use a Web browser and whether to separate data access logic from business rules. However, ORBs (including Microsoft’s DCOM) lower the barriers to multitier topologies by their ability to manage the interfaces between components in a formal, organized fashion. Like other good middleware, ORBs also use the same syntax and semantics between components on separate systems as is used between components that run on the same system. These characteristics of component software will accelerate the adoption of three-tier architectures somewhat because they lower the cost and complexity penalty of using a three-tier architecture (compared to a two-tier model).

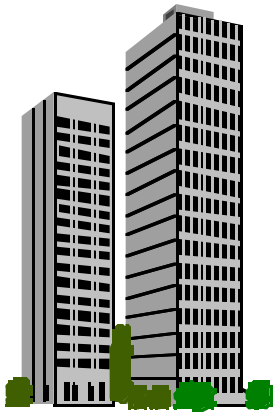


Source: Gartner Group

Figure 7. Component Software in Two- and Three-Tier Applications

4.0 The Architecture of Complex Structures

In the previous section, we concentrated on the processing paths for a single task. In this section, we look at the design of larger structures (see Figure 8), i.e., complex application systems that handle many related business tasks but are still designed by one development team.

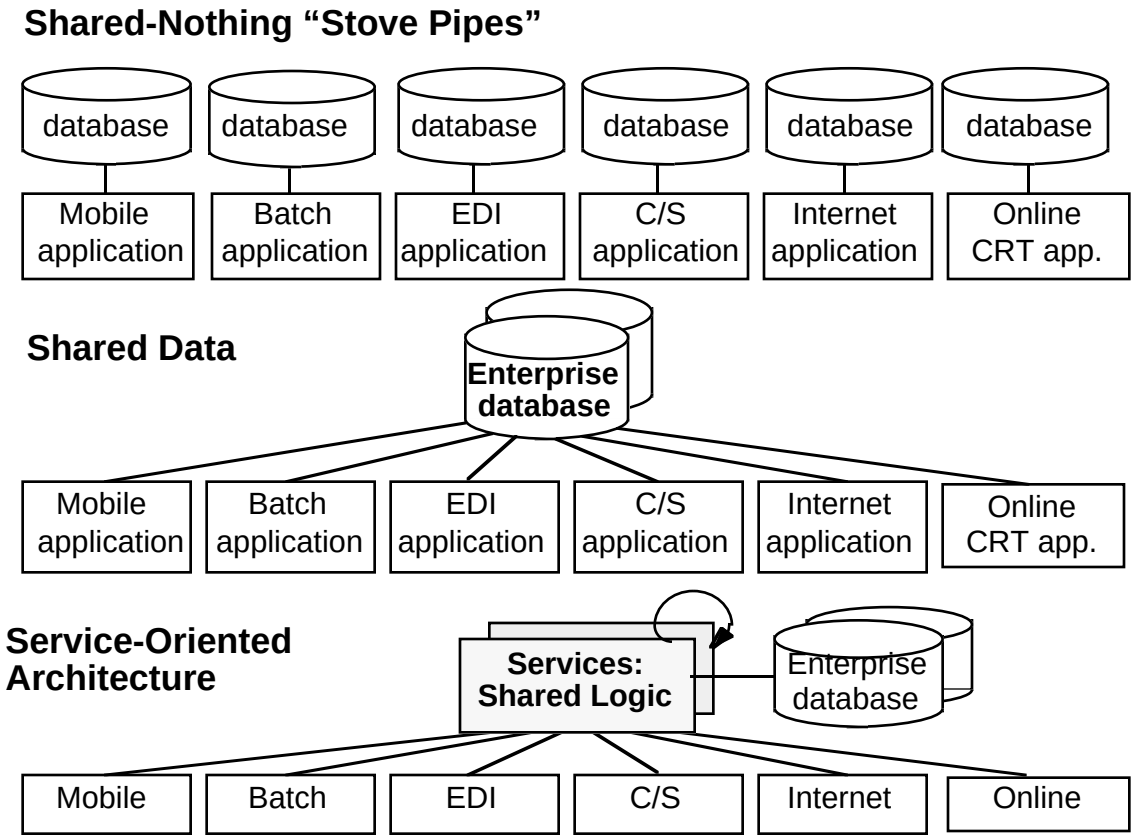


Source: Gartner Group

Figure 8. Complex Structures

4.1 Service-Oriented Architectures

Application systems are being broadened to support new channels of user access, notably, three-tier C/S, mobile computing, the Internet and interenterprise transactions. The very definition of an application "system" is evolving, as the logical overlap between related applications increases. There is a growing need to share data and code across multiple access channels (including batch programs) to streamline the development and management of complex application sets.

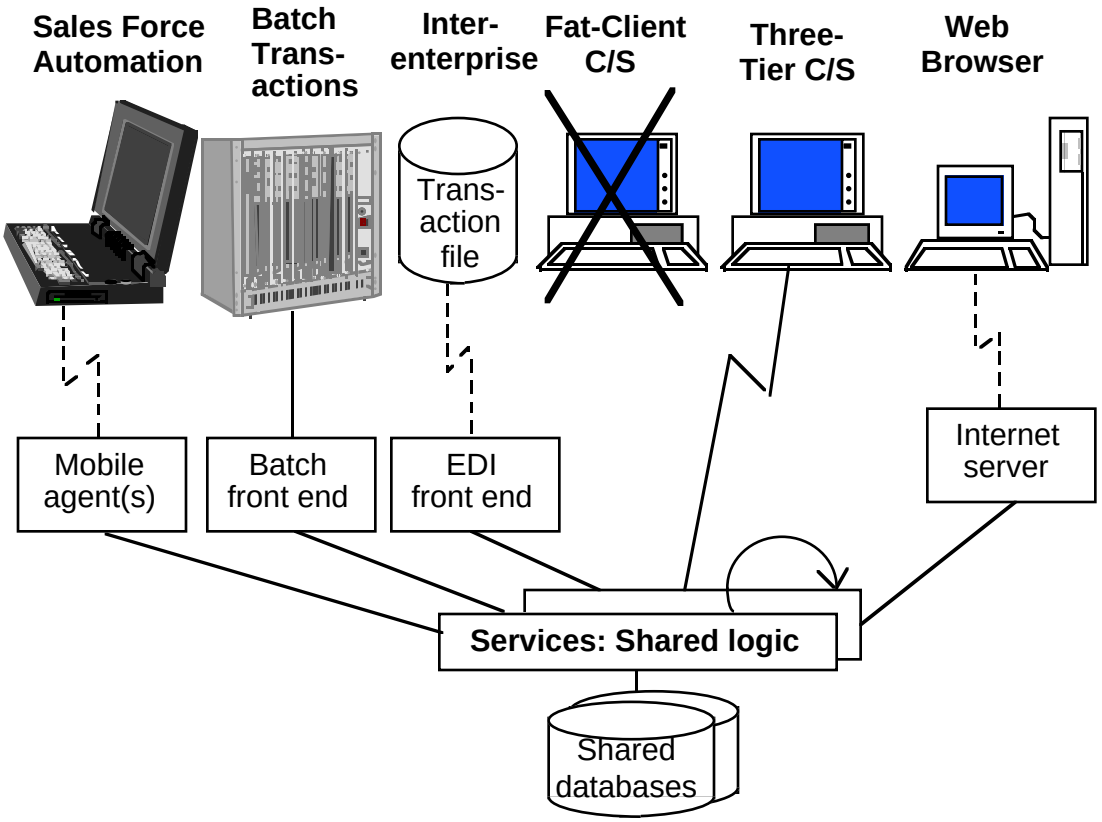


Source: Gartner Group

Figure 9. Designs That Reduce Data and Logic Redundancy

Traditional computing topologies rarely share data directly (see Figure 9, middle section) and virtually never share program logic (see Figure 9, lower section). Most enterprise application portfolios are a blend of shared-nothing stove pipes and data sharing (see Figure 9, top two options). Conventional application portfolios are poor at sharing logic mostly because of shortsighted design decisions regarding partitioning and targeting. The solution lies in service-oriented architectures that can share both logic and data among multiple applications.

A service-oriented architecture is a particular style of multitier computing that helps enterprises share logic and data. It assumes multiple software tiers and usually has thin clients and fat servers (i.e., little or no business logic on the client), but it is more than that. A service-oriented architecture leverages the principle that many aspects of processing logic are common to many users of some particular data set rather than being uniquely associated with one particular application. For example, the business rules, integrity checks and sequence of steps to enter an order may be common to all users (online or batch) of that order, billing and inventory data. The code associated with that order-entry function is therefore organized as a modular service that can be invoked by one or more “requesters” or software “client” programs using defined interfaces. The front-end logic, including presentation, business rules and flow control unique to each application and access mode (e.g., Internet, batch or C/S) is handled outside the service (see Figure 10). Common two-tier remote-data management C/S applications are particularly unhelpful; they cannot participate in a service-oriented architecture because the application logic is locked up in code that runs on the desktop and is thus unavailable for batch, Internet, mobile and interenterprise clients.



Source: Gartner Group

Figure 10. A Service-Oriented Architecture

A service is a black box that hides code and data from the developer of the client application. Services treat all transactions (online or batch) the same, except that online input sources may be given priority over batch sources in the order in which the work is executed. The difference between batch and online users is only on the requesting side. All online tasks, including C/S, Internet and dumb-terminal OLTP, must be executed immediately, because someone is waiting. Processing is deferred for batch tasks because no immediate feedback is expected. Online tasks can be conversational, returning up-to-the-minute information about inventory levels or account balances to a user who is deciding in real time how to proceed. Online applications can also return input errors and other feedback from integrity checks so that the user can re-enter the input immediately. All forms of batch processing must save feedback from failed transactions in files or queues for later delivery to a program or user that can correct and re-enter the transaction.

In practice, service-oriented architectures rarely span the range of application types shown in Figure 10. They often have been confined to online applications although more enterprises are now beginning to apply service-oriented topologies to some deferred processing (batch) jobs. The software in both a desktop computer and a server computer can be organized in a service-oriented fashion. Note that the kind of reuse supported by service-oriented architectures is much different from source-code copying or linking a separate copy of an executable module into an application program. Those other types of reuse are described in ADM *Strategic Analysis Report* R-480-129, Jan. 30, 1996.

Service-oriented topologies will account for more than one-third of new, mission-critical operational applications by 2001, up from less than 15 percent in 1997 (0.7 probability). However, service-oriented architectures are not a universal solution.

- They are not attractive for casual application development because they require some of the inevitable disciplines of reuse: design standards, quality assurance, an administrator or facilitator, incentives to encourage use and documentation for the inventory of prebuilt services.
- They will not replace two-tier C/S applications built with desktop-based tools, because legacy PC 4GLs are widely entrenched and end users often demand control of some of their software development. Small applications whose scope is limited to one person, workgroup or department may be developed faster and easier without the rigor of services.
- They provide no clear benefit for functions that are unique to a single application, although it is difficult to predict that a function will not become a candidate for sharing in the future.
- They are not efficient enough to replace all traditional batch applications where the task demands locking, sorting or sequential passes of many records.
- They are also insufficient for integrating applications that are designed by different development organizations. For this purpose, we must look at topologies that are broader and more flexible in their scope (see Section 5).

5.0 Organizing Multiple Applications of Heterogeneous Origin

We now turn to macrocosmic “city planning”-level design issues (see Figure 11) that deal with managing the relationships between multiple heterogeneous applications.

Source: Gartner Group

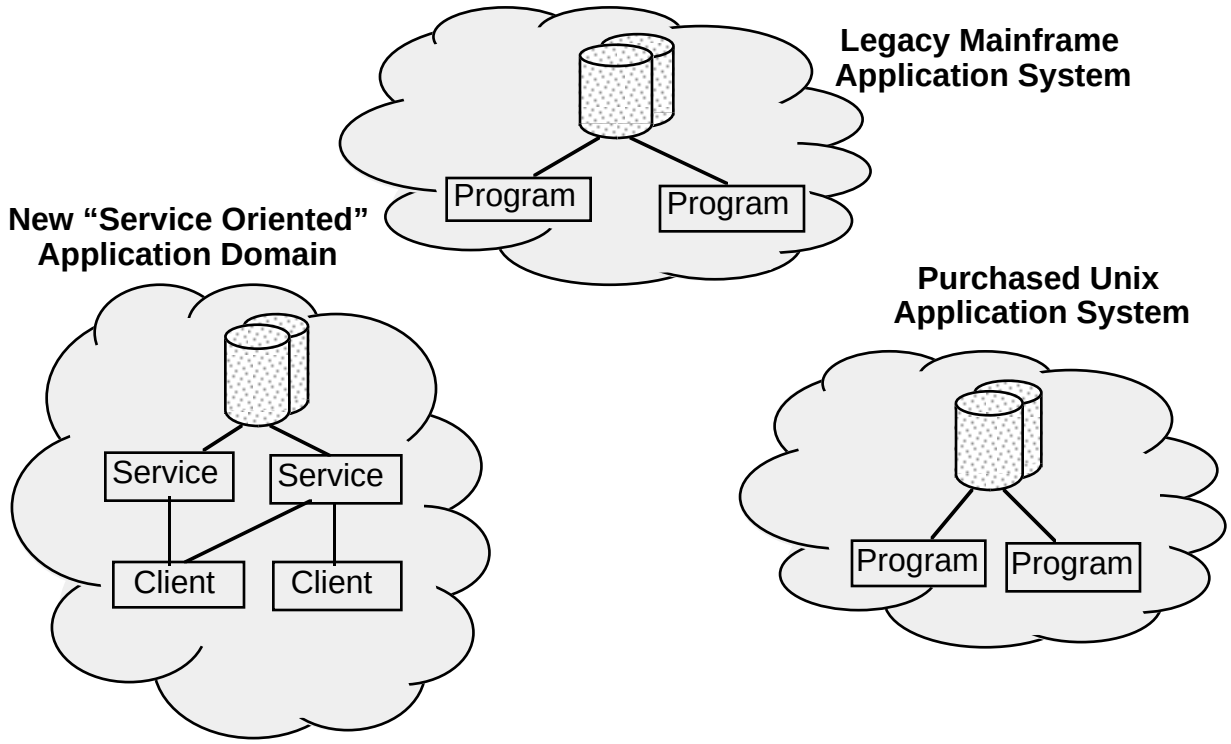
Figure 11. Macrocosmic "City Planning"-Level Design Issues

5.1 The Challenge of Coordinating Disparate Application Systems

Enterprise IT portfolios are, for the most part, still composed of many independently designed application domains (see Figure 12). Within a domain, the technology, data models and semantics are consistent because one development group designs all of the application programs and databases. Even if a domain encompasses multiple application "systems," multiple databases, several OSs (e.g., Windows 95 desktops and Unix servers), multiple development languages (one for the client and one for the server) and other complexities, it still can have a coherent architecture. It is possible to directly share some data and code, as in a service-oriented architecture, although there may be multiple copies of the same data and code in, for example, the data center, branch offices and departments.

Architectural concepts that apply to a single application domain are impractical for managing macrocosmic "city planning" issues. There is no simple way to impose a three-tier architecture or a service-oriented architecture on a collection of diverse applications that have been developed by separate groups at separate times. Purchased and legacy applications are notoriously troublesome because their data models and interfaces do not conform to enterprise standards (if any exist). Most enterprises suffer from cumbersome and inefficient integration schemes because the links between application systems are independently conceived and deployed, as additional applications are installed or modified. The problem is not in the technology of the individual interfaces — reasonable middleware tools exist, ranging from file-transfer utilities, database gateways, message-queuing products, RPC services, ORBs, TP monitors and such. The problem is that enterprises cannot effectively manage or maintain the overall morass of logical connections, and many cannot even document all of their interapplication data and message flows.

Despite the difficulty of the task, the need to integrate disparate application systems is becoming increasingly acute for a variety of reasons. In some cases, enterprises are re-engineering their business processes to accommodate customer-focused views that require close cooperation between previously "stove piped" applications. In other cases, the motivator is a corporate acquisition or divestiture. There is also movement toward "virtual enterprises" that must link application systems from different companies. Regardless of the motivation, however, integrating independently developed application systems is a challenge. Applications may be incompatible on one or many levels, such as the OS, DBMS, programming language, communication protocols, data models or data semantics.



Source: Gartner Group

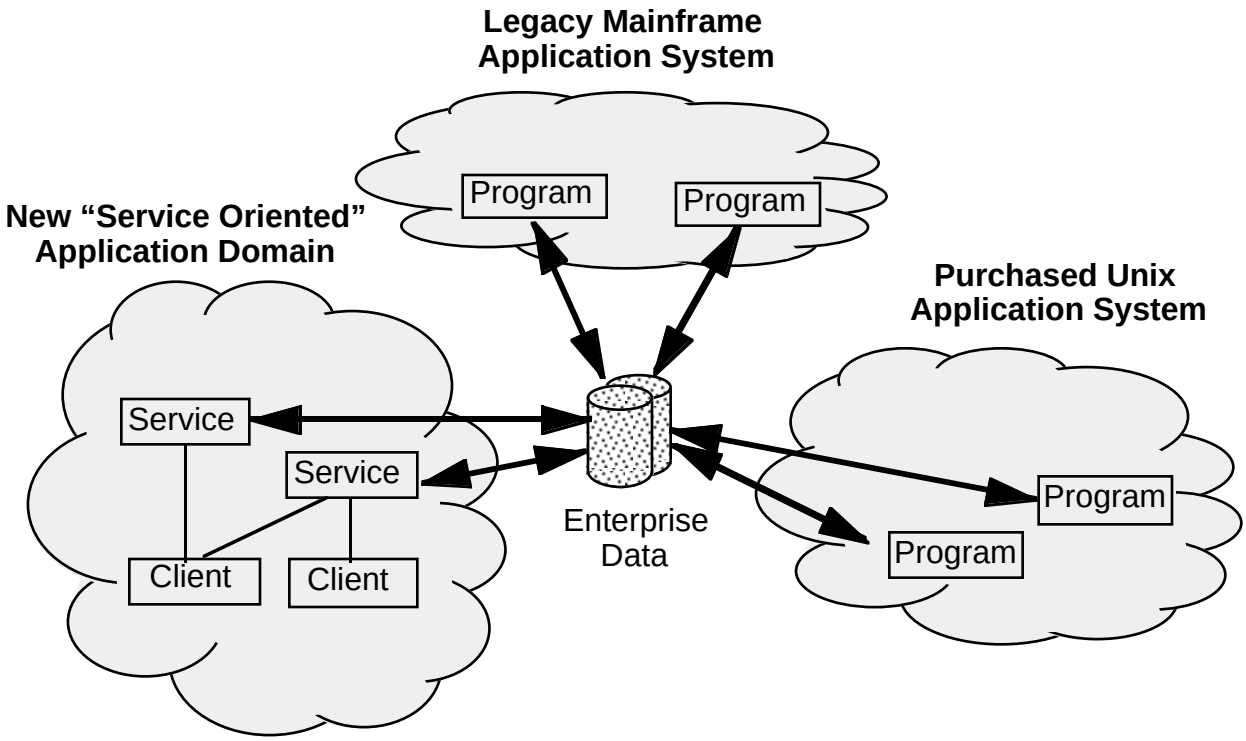
Figure 12. Unintegrated Applications

Enterprises attack this type of problem with macro-level techniques and architectures such as batch data transfer, real-time data integration, data warehouses, ODSs and message brokers. Some of these strategies are much more successful than others, and each has a particular set of conditions for which it is most appropriate. In the next sections of this *Strategic Analysis Report*, we review the available options in more detail.

5.2 Direct Data Sharing

It seems intuitively desirable to directly share one copy of all data across an entire enterprise (see Figure 13). If achieved, this configuration would eliminate data redundancy and all programs would have access to the most up-to-date version of the data. However, purchased and legacy applications are built around their own embedded data models and semantics, and there is no practical way to unhook their respective databases. Direct data sharing works only among programs within one application system or a set of related systems (one domain) that are built by cooperating development groups.

Despite enterprise data modeling techniques and advances in database gateway middleware, the goal of sharing data directly among heterogeneous operational applications will remain unattainable during our five-year planning horizon, i.e., 2002 (0.9 probability) and probably forever (0.8 probability).



Source: Gartner Group

Figure 13. The Myth of Enterprisewide Data Sharing

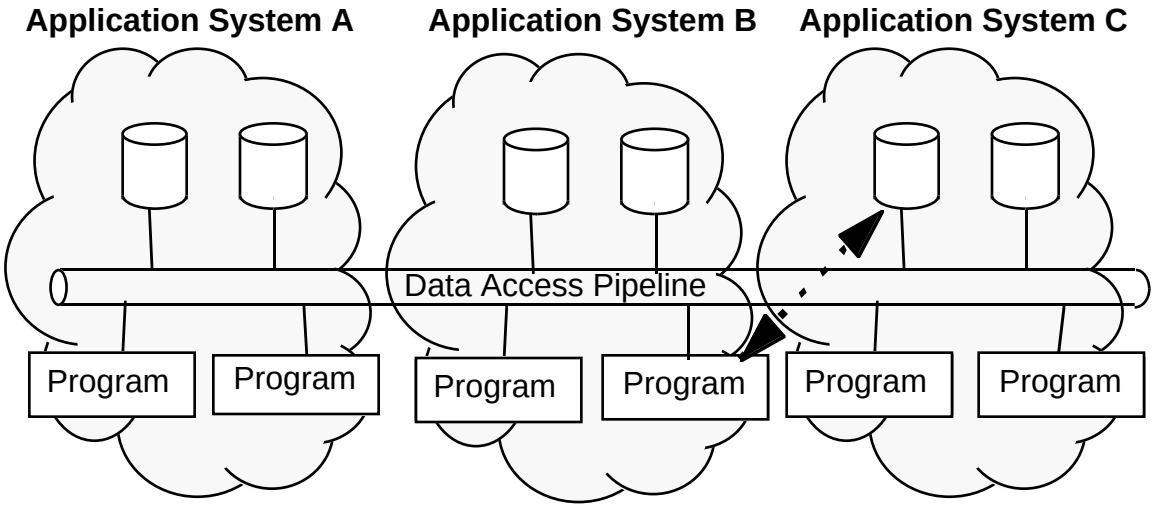
5.3 Data Integration

Periodically, we see enterprises attempting to install a data access middleware tier to create some version of a virtual shared database across multiple applications where a physically shared database is impossible to achieve for reasons described in the previous section. This strategy may be aimed at operational applications, read-only business intelligence (BI) applications or both. We will first examine the more ambitious, and less practical, of these alternatives, which is to use a data access tier for operational (updating) applications.

The purpose of a generalized data access tier (sometimes called a "pipeline") is to buffer application programs from the differences in the structure of the various application data models, the syntactic and semantic disparities of the many database data manipulation languages (DMLs) or the location of the databases in the enterprise (see Figure 14). The application program sees only a consistent API, usually SQL-based and generally based on the ODBC API. A data access tier theoretically would make it possible to change the data management software (e.g., from VSAM to Oracle), the server platform (e.g., move the database from MVS to Unix) and the application database design (e.g., normalize, denormalize, add columns or tables), all without changing the application program in any way. Application programs would read and write data only through an indirect, logical view.

A data access tier could be enabled by a database gateway, a user-written middleware layer or some combination of these. A user-written layer, built atop a general-purpose message-queuing product, for example, is the most flexible alternative, because it can be customized to read and update any data source (e.g., MVS VSAM or an obscure DBMS). Although it will have to include its own data catalogs, directories and possibly transaction management logic, a user-written pipeline could leverage purchased

gateways and replication capabilities for some of its functions to minimize the custom work required. A data access tier can be configured on two hardware tiers (e.g., a PC client and MVS server or an MVS client and Unix server, etc.), but it could also be deployed across three hardware tiers.



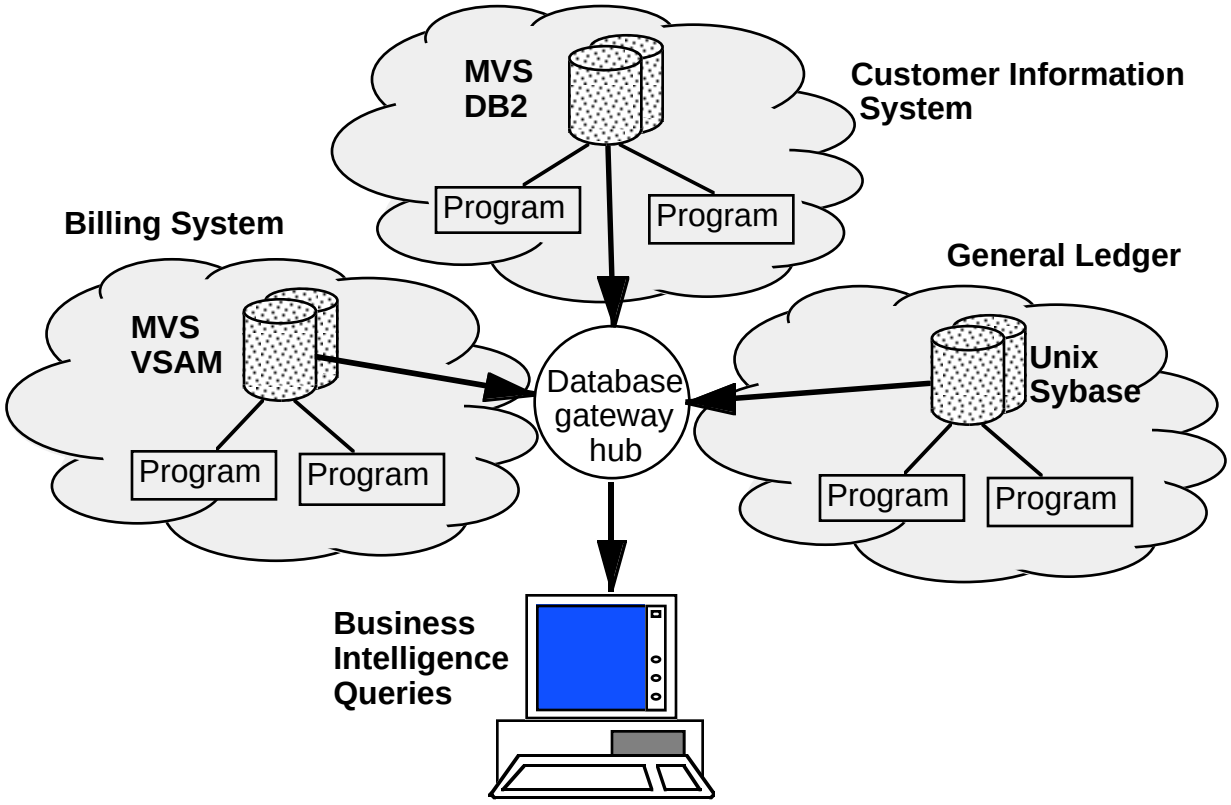
Source: Gartner Group

Figure 14. Universal Data Access Pipeline

We have not seen a successful deployment of a general data access tier for updating applications, however. User experience with database gateways suggests that real-time data integration is practical only for read-only applications and even then it is usually less practical than alternative solutions such as data warehouses. A practical solution for universal anything-to-anything data access with transparency and good performance does not seem to exist. It is difficult to map SQL into VSAM, IMS or other DMLs completely and efficiently. Moreover, it appears to be impossible with today's software and hardware technology to reformat every database request from each of the physical database schemas into the custom data structures that would be desired for reading and writing into application programs that were designed in isolation from the databases. The translation process would have to take place for every database read or write and would be one or two decimal orders of magnitude (10 to 100 times) less efficient than the normal execution of a DBMS DML request. Furthermore, a real-time data access pipeline could create huge holes in the integrity of application databases if it bypasses all of the integrity constraints, edits and business rules that are applied to normal database updates. The real-time data integration approach is even less likely to work if the server databases are nonrelational (e.g., VSAM, ISAM, IMS or Integrated Database Management System) or if the applications will run moderate or heavy production workloads. Historically, the primary reason that the database gateway vendors such as IBI and Sybase/MDI added RPC capabilities to their respective database gateways was because of the functional and performance limitations of this approach. A user-coded data access tier will encounter the same technical challenge.

The notion of real-time data integration is somewhat more successful when applied to read-only BI applications (see Figure 15), although there are some significant limitations that apply even for this mode of usage. The goal of universal data access has largely eluded large organizations, but it is not for lack of trying. On one level, it seems desirable to be able to present a comprehensive view of all the data in an enterprise to an end user who is armed only with a desktop BI tool. However, the intractable problems of complicated data models and undocumented data semantics continue to thwart this effort. End users

cannot be expected to understand where the data is, what it means and how it is stored across a broad range of independently designed production databases. The database gateway products, such as Information Builders' EDA/SQL, Sybase/MDI's OmniServer and Oracle's Transparent Gateway, do a creditable job of translating among SQL dialects, and some even provide access to nonrelational databases. However, database updates are slow where they are even supported, and the performance and security implications of allowing ad hoc access into production databases are significant. SQL access through database gateways will remain a niche solution for read-only decision support using preplanned queries into a subset of databases.



Source: Gartner Group

Figure 15. Data Integration for Decision Support

In summary, the siren song of universal data access has been muted as users have gradually recognized the limitations of direct access to heterogeneous data. Data integration does not work well for OLTP. It works a little for some forms of decision support, but even then it can be dangerous, costly and misleading, and we consider it to be generally inferior to alternative offline architectures such as data warehouses.

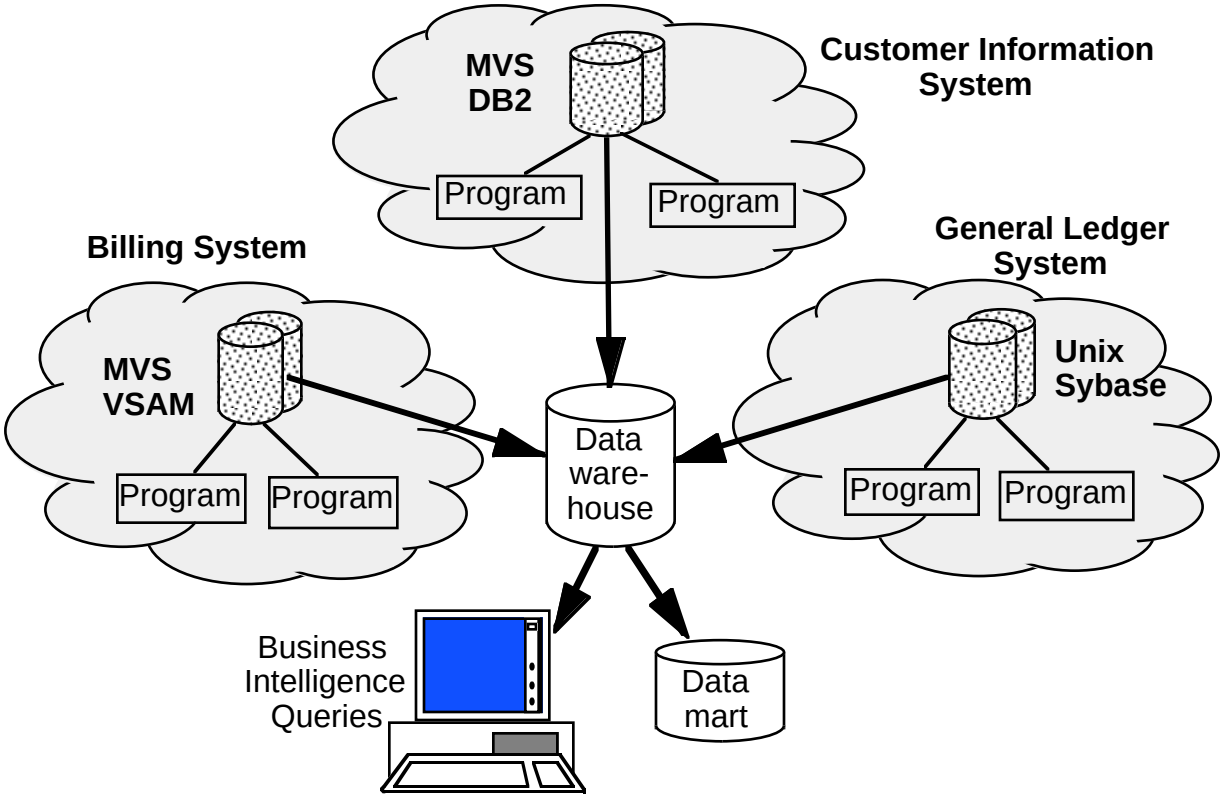
5.4 Data Warehouses

Data warehousing, simply stated, is a means for creating and managing a data architecture for user access and analysis**.** This involves the design and creation of physical and logical database structures intended specifically for this purpose, and the extraction, transformation, consolidation and quality improvement of data to form an information resource rather than raw data. This is the critical foundation for both BI and data mining.

BI, a term coined by Gartner Group in the late 1980s, describes an enterprise's ability to access and explore information (contained in a warehouse), analyze that information and develop insights and understanding, all of which lead to improved and informed decision making. BI products include decision support systems (DSSs), executive information systems (EISs), and query and report-writing tools.

Data mining, unlike BI tools, is far less user-directed and instead relies upon specialized algorithms (e.g., fuzzy logic, neural networks, genetic algorithms and induction) that correlate information (possibly from a data warehouse) and assist in discerning important (and perhaps obscured) trends, unguided by user bias and assumptions. Data mining also refers to a process rather than a technology, with the goal of that process to explore large amounts of data to discover new trends, relationships and categories in that data. Data mining is also referred to as knowledge discovery.

A data warehouse is an example of interapplication or "extra-application" data sharing. Data is extracted from multiple sources, pruned, "cleansed," reconciled and transformed into a more usable format and then loaded into the data warehouse for subsequent access by multiple interested consumers (see Figure 16).



Source: Gartner Group

Figure 16. Data Warehouse Architecture

A data warehouse architecture should include extracts of operational data that are "frozen views of information" trapped in time capsules, which in some cases have some level of summation and history associated with the view of information. The extracts are created either by handcrafted programs that take time and expense to maintain or through tools that help automate the generation of the extract applications or processes. These applications should provide the capabilities to perform the complex task of integrating data from multiple sources to create a consolidated view of the data, as well as the

transformation of data for use by BI applications. In addition, a data warehouse project can create the opportunity to perform procedures to ensure the quality of the data through data "unduplication" (e.g., householding) and data element validation. Once the extracts are created, their use should be audited automatically by system software that senses the use of files. This tells data administration who is using the data so that business changes that result in data changes can be reflected in extract programs. If not, enterprises may, at best, have their programs malfunction or, at worst, produce erroneous results for management.

In most circumstances, a data warehouse is more effective than a database gateway for decision support and other BI applications because its data model is designed for ad hoc query access. Operational databases are usually optimized for production purposes, including OLTP updates. And, of course, a data warehouse is a persistent data store, whereas a database gateway is an online process.

Data warehouses satisfy most decision support requirements better than database gateways into operational databases because the data models, data semantics and management practices of data warehouses are designed specifically for decision support.

DSSs and data warehouses are not the same. Although decision support is a major benefit that is provided by a data warehouse, the DSS activity can be independent of a data warehouse and exists outside a data warehouse architecture. Also, a stand-alone (i.e., stovepipe) DSS does not include the architecture, administration, infrastructure and auditability that a data warehouse does. Furthermore, because of the erroneous equivalency awarded to DSSs and data warehousing, and the similar terminology used by these disciplines, many enterprises believe they are planning to implement a data warehouse when they are simply implementing a DSS database specific to a particular business requirement. Out of approximately 2,000 companies that say they are planning to implement a data warehouse during the next two years, fewer than 350 of those actually are. Gartner Group identifies the retail, banking, insurance and telecommunications industries to be the leading industry segments that are putting together data warehousing architectures, with evidence that the pharmaceutical and healthcare industries are beginning to adopt this technology.

Unlike a specific DSS, a genuine data warehouse implementation is usually done in stages with an enterprise strategy in mind. The characteristics of the first stage include one or two BI applications, ad hoc query users and one or two subject areas of data. As enterprises transition to the latter stages, the main difference in characteristics are found in an increased number of BI applications and subject areas contained in the database. Adding subject areas enables an enterprise to perform cross-functional data analysis, which provides a more accurate look at business profiles. Here, the role of an enterprise strategy cannot be understated. After completing the first stage, enterprises without an enterprise strategy will experience difficulty in moving to subsequent stages.

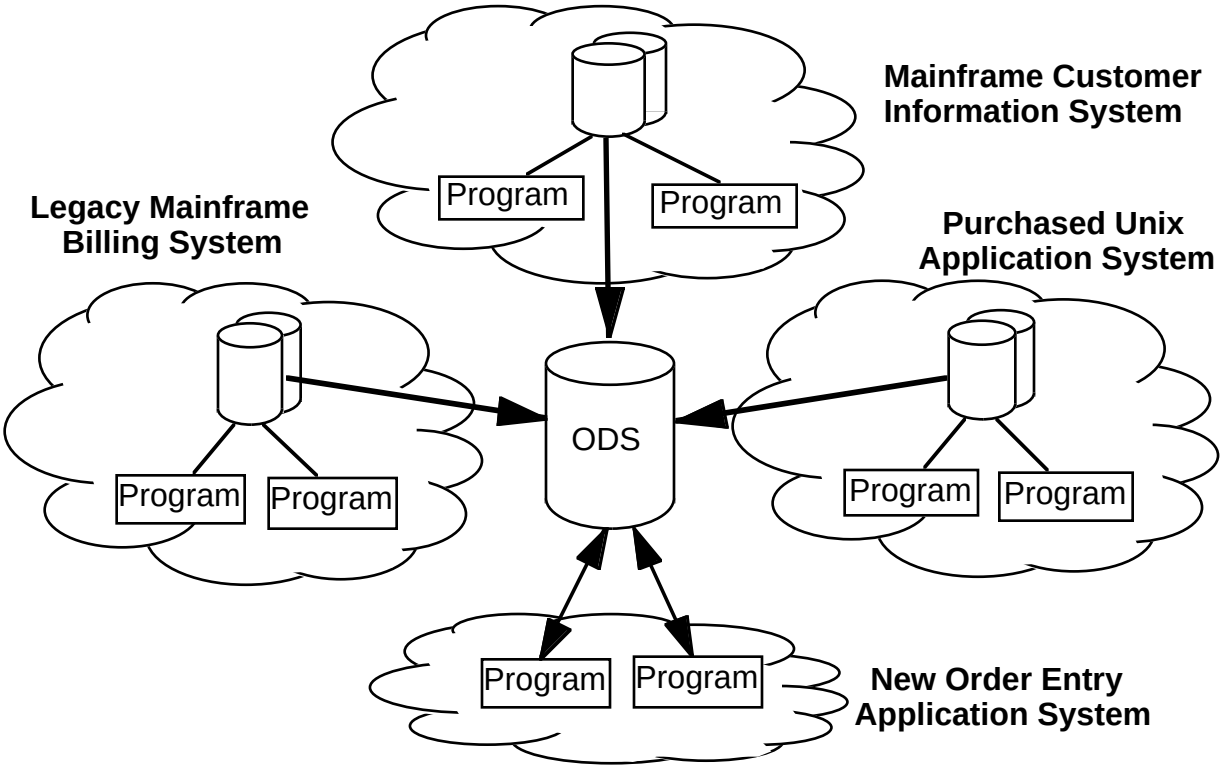
In summary, data warehouses reflect the growing need to provide a coordinated view of data across the enterprise. They are proving effective despite their cost and complexity. Although sometimes mishandled, they will continue to grow in popularity. Note, however, that data warehouses are read-only and do not address the needs of transaction processing application systems.

5.5 ODSs

ODSs are a new articulation of the perennial concept of shared production data. Different from a data warehouse, an ODS supports day-to-day operational decision support (e.g., customer service) and

contains current value data propagated from operational applications. This causes the data maintained in the ODS to be subjected to frequent changes as the corresponding data in the operational system changes.

Like data warehouses, ODSs are generally populated by regular, periodic extracts from production databases (see Figure 17). Some are updated in near real time by replication or message-queuing mechanisms, others by a nightly batch run. ODSs are generally read-only. However, for new applications, the ODS may serve as the shared ("updatable") transaction processing database for new applications.



Source: Gartner Group

Figure 17. Operational Data Store

An ODS is an alternative to having operational query applications access data directly from the database that supports transaction processing. In some cases, this is helpful in eliminating the potential performance problems that transactional applications may otherwise experience when queries contend for the same data resources. However, many applications have no problems supporting queries into the production databases, either via direct lookups or, where there are multiple application systems involved, by going through a message broker (see Section 5.7).

ODSs and shared databases work best when all the applications that are involved are designed by one group or by cooperating development groups. However, ODSs are forever redundant with the databases in legacy and purchased applications, which sharply limits their benefits. Because of the complexity in obtaining the current data of record from operational systems in a timely manner and the effort required to maintain redundant copies of data, organizations will be reluctant to employ an ODS until it can coexist and be managed in concert with the data warehouse. With the data fragmentation commonly found in large enterprises, it is difficult to combine data from multiple systems. Most ODS systems will be targeted

to a particular application requirement, giving the database administrator and data administrator the opportunity to design the ODS to perform optimally for that specific application (e.g., by minimizing the number of indexes). However, this means that the ODS is less suitable for new, unforeseen application needs such as ad hoc queries.

Until 2000, except for clearly focused applications, ODS deployment will be difficult and suitable only for the skillful and strategically minded who can justify the costs (0.8 probability). Through 2001, even in their most successful installations, ODSs and shared databases will never hold more than 25 percent of the data of record for any large enterprise (0.8 probability).

In one example of which we are aware, a financial enterprise has successfully created a combined ODS and data warehouse using a single database. The ODS-oriented data is maintained in a near-current state by using several types of data input mechanisms, including:

- Message broker technology to input transaction data from local online trading application systems
- File transfer of flat-file spreadsheet extracts from remote locations

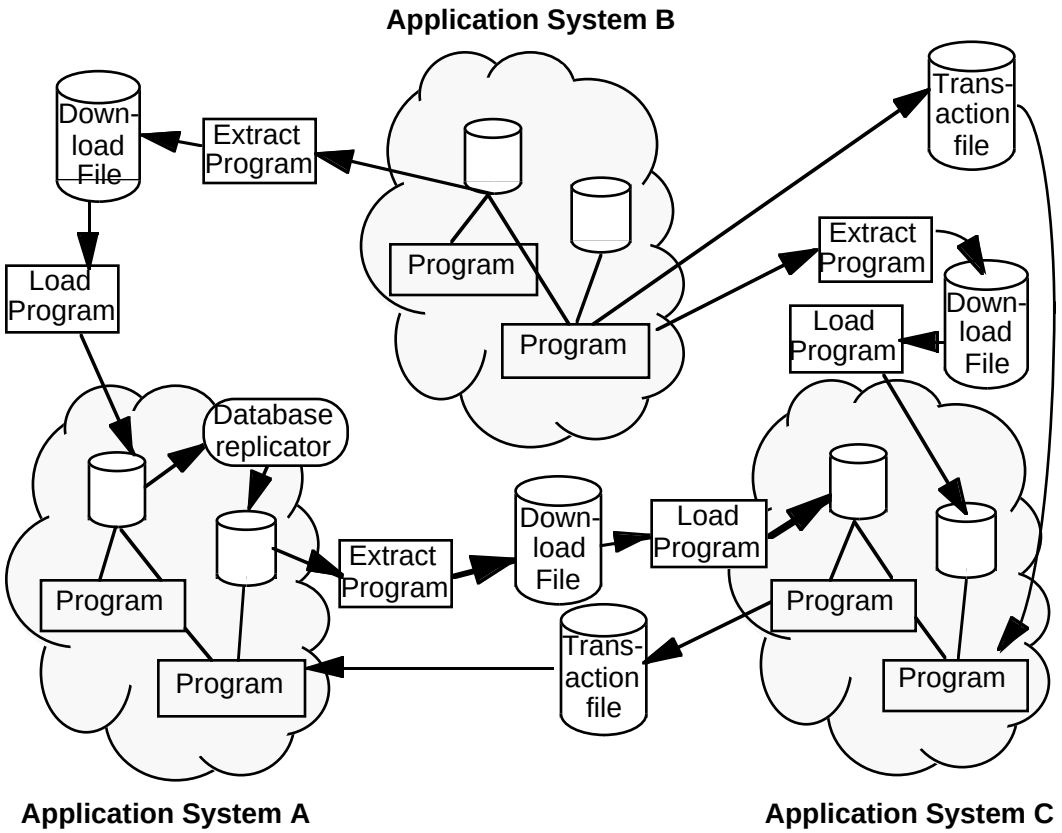
This data is maintained in a specially designed, ODS-oriented schema to support the performance requirements of a real-time “trader’s dashboard” application. On a nightly basis, data from these ODS-related tables are selected and transformed into another set of normalized tables (within the same database) that support the data warehouse requirements. We believe that this integration of ODS and data warehouse represents a likely direction of evolution for many variations of the ODS.

5.6 Batch Data Reconciliation

The most common technique for “integrating” disparate application systems is still batch file transfer (uploads and downloads of databases or database extracts). File transfer is a simple way to reconcile data that is held by different application systems. In a data synchronization approach, data is extracted from the source application database(s) with an add-on program or some data extraction utility. These updates are temporarily stored in a transfer file, queue or database (see Figure 18). Alternatively, an application program in the source application (system B) could be modified to write directly into the transfer file. At periodic intervals, usually nightly or weekly, a user-written update program, or possibly an incremental load utility, reads the updates from the transfer file and inserts them into one or more target operational databases.

Standard DBMS replication facilities are usually not flexible enough by themselves to translate and apply the database changes between disparate application domains. Replication utilities operate mostly within the domain of a homogeneously designed application system, and are primarily used to maintain a hot backup of data for availability purposes or to download data from a central site to branch or regional offices. Between independently designed systems, the data models and semantics in the source and recipient databases are incompatible. Therefore, user-written programs are generally involved, especially on the update side, in most enterprises employing this architecture.

This method works, and every application designer and programmer understands how to do it. However, it burdens the enterprise with a perpetual need to reconcile data between multiple stove-piped application domains because some data are inevitably redundant. Most enterprises store dozens of copies of their customers’ names and addresses, and these copies often disagree with each other. This solution is inherently complex and subject to the following drawbacks:



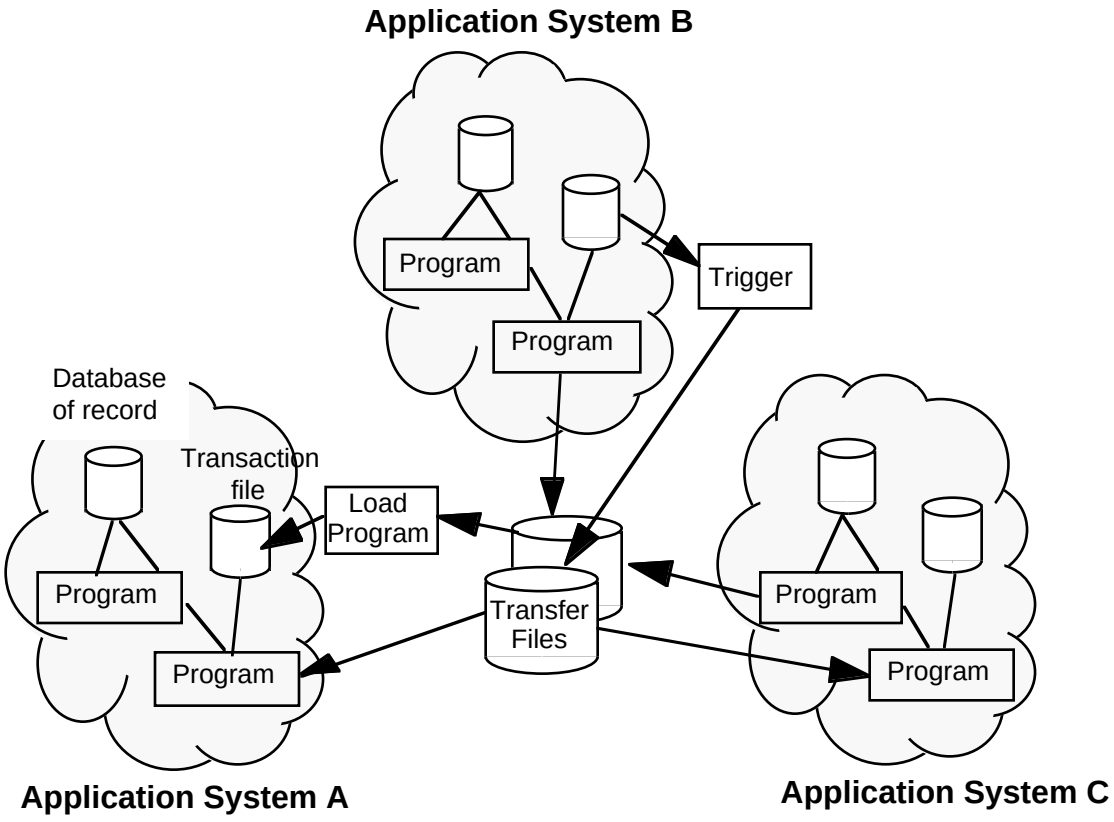
Source: Gartner Group

Figure 18. Batch Data Reconciliation

- Redundant copies of data are difficult to manage and potentially expensive if the databases are large.
- The data is not fully up-to-date, introducing a high probability of inconsistencies.
- The application design is relatively inflexible in time, because changes in one database require changes in other databases and other applications.

Thirty-five percent to 40 percent of a typical IS maintenance budget is spent on the extract and update programs that support this style of integration. However, there are ways to improve on this situation by applying planning and forethought to the data transfer process. A growing number of enterprises are moving to organize their batch data transfer operations around shareable transfer files or transfer databases (see Figure 19).

This is essentially a type of multisystem batch updates. It funnels all updates of each type through one or more shared transfer files that are available to all sending or receiving applications that need to deal with that class of data. The transfer may take place once a day or every few minutes, depending on the application requirements and the choice of middleware.



Source: Gartner Group

Figure 19. Organized Transaction Reconciliation

Organized transaction reconciliation is superior to other forms of batch data transfer in two ways:

- It can execute one-to-many transfers (one source to one or more recipients) or even many-to-many transfers. The source application system does not have to resend the data separately to each recipient system, so this technique is more efficient than traditional interapplication batch transfer "spaghetti" designs.
- It is more reliable and flexible than designs that enable file transfer programs to directly update the database of record in the receiving application systems. Updates are always processed through application programs that are native to the receiving application systems, so there are no add-on programs or utilities that duplicate application logic in the receiving application systems. The update program in the receiving application system may directly read from a shared transfer file or a load program may extract data from the transfer file, transform the data and create a transaction file in the receiving application system. By contrast, in direct batch data reconciliation schemes (see Figure 18) where the database of record is directly updated, someone must remember to change the add-on programs or utilities that apply the updates when the database in receiving application C is modified to add a new field, or when changes are made to some kinds of business rules in a program in receiving application A. This effort is redundant and prone to error. Organized transaction reconciliation is inherently simpler to maintain, because most database changes and business-rule changes will require no additional maintenance effort beyond the work required within the native application system.

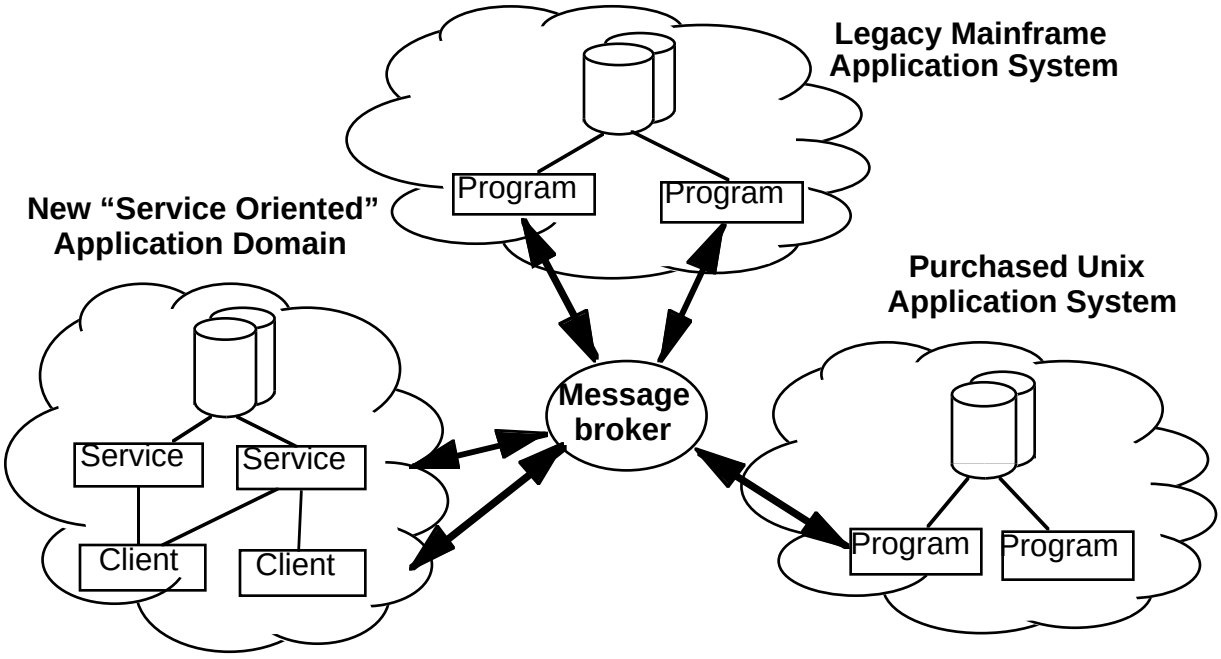
Organized transaction reconciliation is a batch version of message brokering (see Section 5.7). Both approaches can be reasonably reliable (i.e., not lose data) because they can rely on the backup and transaction integrity mechanisms of a DBMS or message-queuing system (respectively). Moreover, both approaches can leverage off-the-shelf transformation products that will reformat update records to match the receiving application without having to do all of this in a custom application program. However, batch transaction transfer models are still batch-oriented and still transfer information only in one direction. They are inferior to full-blown message brokers for addressing a number of requirements, such as:

- Up-to-the-minute consistency of data across multiple different applications.
- Multistep processes where data is processed and then forwarded to other applications in a complex workflow where flow control decisions are based on the value of individual data fields in a record.
- On-demand, request/reply communication models where the receiving application system asks for the data at a time of its own choosing (batch transfer is only a one-way process).

Therefore, organized batch transaction reconciliation can be used to complement real-time record-at-a-time message brokers, but is not sufficient as a replacement for message brokers.

5.7 Message Brokers

A message broker is a logical middleware-based hub that copies and resends messages to one or more destinations (see Figure 20). It is an intelligent third party (hence “broker”) between information sources and information consumers that makes communication an independent, sharable function.



Source: Gartner Group

Figure 20. Message Broker Architecture

Unlike the previous forms of integration across multiple applications, message brokers are based on function integration, i.e., they use program-to-program communication rather than direct data access. Message brokers add value to the communication process in the following ways:

- By transforming messages from the incoming message format to different output formats;
- By temporarily storing messages in a message warehouse to be retransmitted later; and
- By organizing and executing complex, multistep business procedures through flow-control (workflow) services.

Message brokers can be easily combined with service-oriented application systems because such systems already have clean interface contracts into the client and service application programs. Message brokers can also be fitted to work with standard (nonservice-oriented) systems by wrapping the application programs with some type of interface layer. However, this often involves modifying the application programs or using a screen scraper if there are no native interfaces available. Message broker architectures often leverage development tools, EDI interfaces, screen scrapers and other mechanisms that make it easier to build connections into legacy or purchased applications.

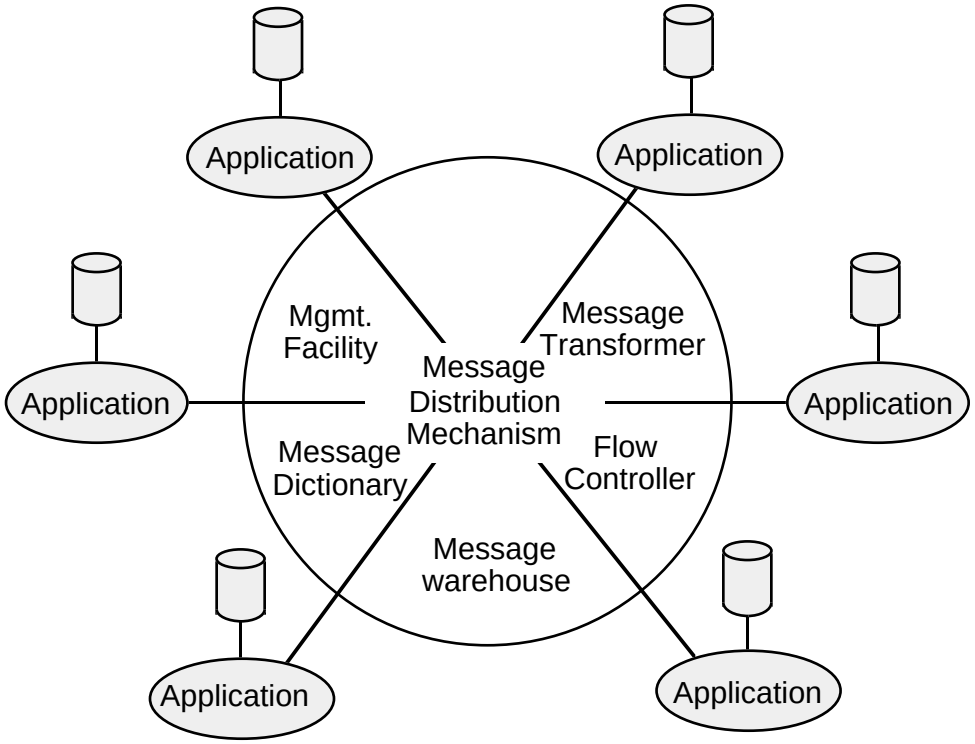
A message broker treats the entire application system domain as a black box. Users and developers from other application domains can interact with the code and data inside the black box only by sending a message through the message broker, using a documented formal message interface.

The ability to reuse messages, and the processing logic that sends and receives messages to and from each application, is a major part of the financial justification of message brokers. Message brokers have the potential to reduce software development and maintenance costs for situations that require connecting multiple applications. The other main benefit of a message broker its ability to encapsulate and reuse legacy and purchased applications. Through a message broker, organizations can impose a new workflow process, thus enabling business process re-engineering without discarding application programs or changing their databases.

Message brokers assume that the interfaces of the participating applications are inconsistent and therefore provide specific, tailorable integration services. However, message brokers are overkill for use within one application system where everything is new and programs and databases are inherently compatible. A service-oriented architecture and shared database are more appropriate for systems where there is commonality of technology and design decisions.

A message broker performs some or all of the following functions (see Figure 21):

- Message distribution. Copies and resends messages to multiple destinations.
- Transformation. Transforms messages from the incoming message format to different output formats.
- Message warehouse. Temporarily stores messages to be retransmitted at a later time based on logical selection criteria.
- Flow control (workflow). Organizes complex, multistep business procedures.
- Message dictionary. Holds metadata description of message formats for development purposes.
- Administration and monitoring. Manages the operation of the broker configuration.
- Adapters. Provides tools for connecting to or encapsulating participating applications.



Source: Gartner Group

Figure 21. Message Broker Functions

Message brokers have the potential to improve significantly the way heterogeneous applications are integrated. However, off-the-shelf message broker software is still fairly immature. For high-volume, high-integrity workloads, user enterprises often must write some of their own middleware code to augment ISV products that provide parts of the solution. A number of leading-edge enterprises have achieved significant benefits from message brokers. However, they are not a mainstream phenomenon yet, except in the banking and healthcare industries. Organizations with a high need to integrate heterogeneous systems and a willingness to tolerate immature technology should investigate this approach to message handling. As the relevant middleware tools mature and mainstream enterprises become familiar with this style of architecture, we expect message brokers to become more pervasive. *By 2001, more than half of all large enterprises will have some form of message broker in production (0.7 probability).*

6.0 Implementing Modern Architectures

6.1 Applying the Black Box Metaphor at Different Levels

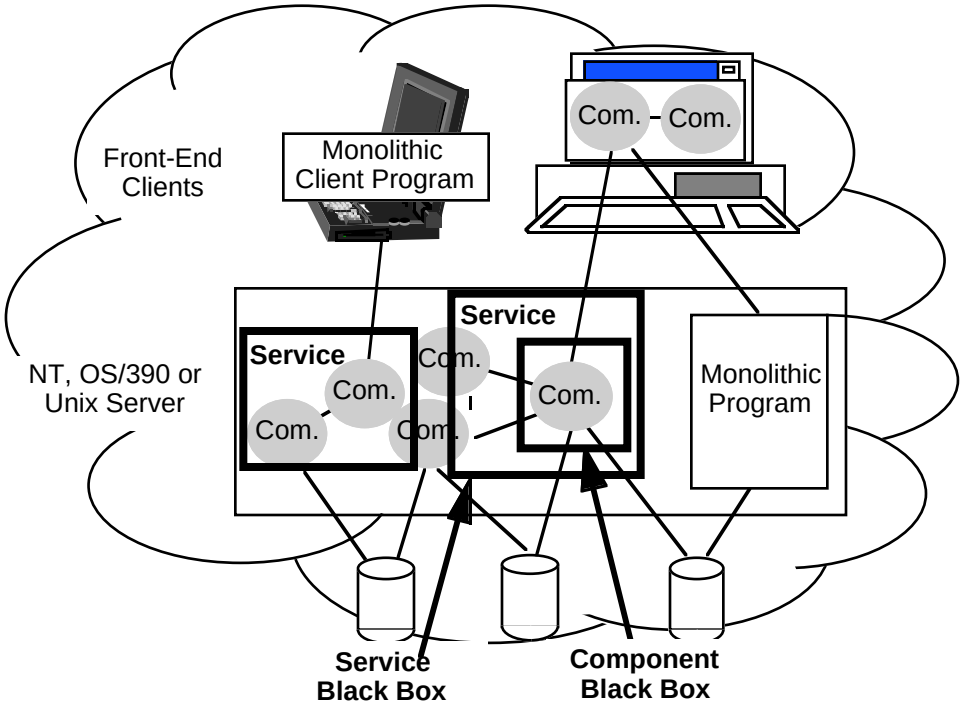
Now that we have reviewed the major modern architectures, it may be helpful to put them into perspective by examining how each of the architectures leverages the black box metaphor that was described in Section 2.2.

To repeat our earlier point, the major benefits of encapsulation are to protect data and code from careless or malicious misuse (i.e., to increase system integrity) and to shield the external developers from the complexity and dynamism of the contents of a module (thereby shortening the time it takes to deliver a working application and lowering the cost of application development and maintenance). Our use of the

term encapsulation implies that external users can interact with the logical black box only through interface contracts that specify the input and output parameters of the messages.

The three major message-based techniques described in this *Strategic Analysis Report* (component software, service-oriented architectures and message brokers) exploit the black box metaphor in the following three different ways:

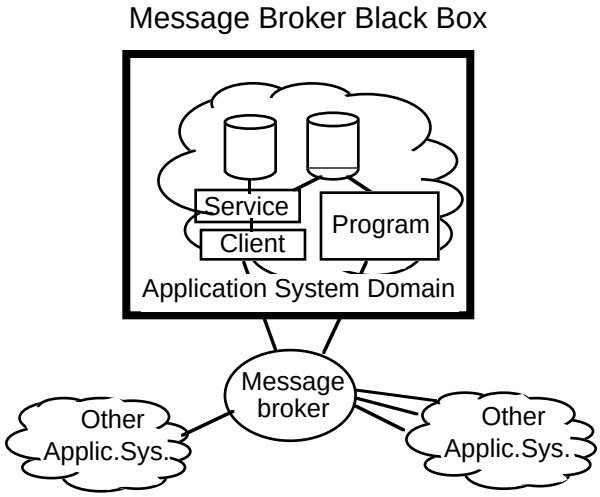
- A component is a program or subroutine that acts as if it is in a fine-grained software black box. A component can be as little as a few statements, although some components are many thousands of lines of code.
- Services use the black box metaphor on a larger and more abstract, medium-grained level. Contrary to the current conventional wisdom, one service usually does not map directly to one component. Some services can indeed be implemented as one entry point in one component, but most services would fail miserably if they were implemented in this fashion. This is an important point. For example, a service such as “enter an order,” must be resolved into a series of method calls into several different components (see Figure 22). The incoming “enter order” message may first be sent to a component that validates the customer name and credit history and then sends another message to a second component that validates the order line items and availability. Next, a third component may create a shipping transaction, a fourth component may update inventory databases and a fifth may update the billing system. Service-oriented architectures use the black-box metaphor as a design-level abstraction. Unlike components, services do not exist at runtime. There is no directory or repository of services except, possibly, at software development time for use by programmers. At runtime, only the executable code of the components or programs that implement the service actually exist.



Source: Gartner Group

Figure 22. Components and Services as Black Boxes

- A message broker can encapsulate an entire application system, including hundreds of programs and databases, in a black box. The message broker is the most macrocosmic implementation of encapsulation (see Figure 23).



Source: Gartner Group

Figure 23. Message Broker Encapsulation

Note that encapsulation is in the mind of the developer of the external, requesting (client) application module that sends requests into the black box or receives information out of a black box. Each of the three design styles hides the code and data from a different audience (type of developer). In the case of a message broker, the code and data are hidden from anyone outside the team that develops the encapsulated application system (see Figure 24). In the case of a service-oriented architecture, the service's code and data are hidden from client program developers working within the same development team. In the case of components, no one except the author of the component itself needs to know the contents of the component.

The benefits of components, service-oriented architectures and message brokers do not require that they have exclusive ownership of their respective data sources for all time. All three approaches work even if the data "inside" the black box is accessed by other components, services or application systems that are outside the black box (as in Figure 21 and Figure 22). However, there is clearly an advantage to having few or no other external modules that access the same data. Having fewer modules will increase code reuse, security and data integrity and make it easier to locate the source of any bugs. Ten developers writing 10 components/services or application systems that directly access certain data will have more problems than 10 developers that funnel through one, two or three components, services or systems to read and write the same data.

Note that in any of these approaches, there may be multiple interfaces (message types) in and out of a single black box. Moreover, all three can be implemented with an ORB style of interface, although it is not necessary to use an ORB-like interface for any of them. Again, we use the term ORB in a general sense, one that is applicable to CORBA-style products, Microsoft's DCOM and other middleware services that have roughly equivalent features.

	Message broker architecture	Service-oriented architecture	Component software
Rogue hackers			
Programmers outside this one component			X
Programmers of client applications in this domain		X	X
Developers from other domains	X	X	X

Source: Gartner Group

Figure 24. From Whom Data and Code Is Hidden

Examples of message brokers that use an ORB-style interface include SAP R/3's Business Object Broker, IBM's Business Object Server (BOS) and the Republic National Bank of New York's Pipeline project (the latter two use the CORBA IDL).

Examples of ORB-style service-oriented architectures include many current CORBA applications. We also expect that most applications implemented with Microsoft's new Transaction Server will be configured as services that consist of one or many DCOM components.

It is possible and potentially useful to use all three forms of encapsulation simultaneously. The whole system may be encapsulated through a message broker; within the system there may be encapsulated services; these encapsulated services may be made up of encapsulated components (like a Matroshka doll).

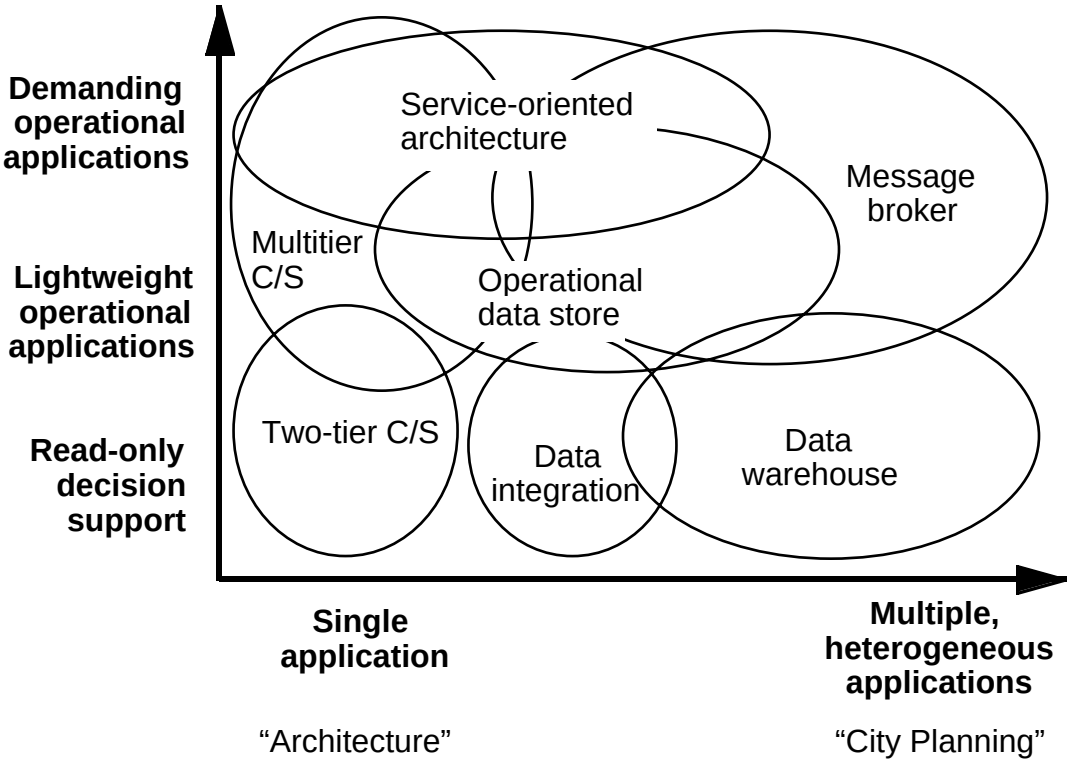
6.2 Selecting the Appropriate Topology

The architectures reviewed in this *Strategic Analysis Report* are complementary notions — most large enterprises can benefit from most of them, as long as each is used where it fits (see Figure 25). Some application systems will use several of the design styles simultaneously.

Multitier and service-oriented architectures apply within an application domain that has been designed as a coherent whole by one development team or closely cooperating development teams:

- Two- and three-tier concepts apply to the arrangement of individual parts of the application system. Some application systems use two-tier topologies for some work and three-tier topologies for other work, although this generally is to be discouraged because it can restrict the flexibility of the three-tier applications.

- Service-oriented architectures apply to large systems with many related programs and (usually) multiple databases. Service-oriented architectures are used in durable application systems that will grow and be maintained for a period of years. They are usually not relevant for casual, ad hoc small applications. Many modern, component-based applications will be both multitier and service-oriented.



Source: Gartner Group

Figure 25. Roles Played by the Major Architectures

Organized batch transaction synchronization, message brokers, database gateways and data warehouses are relevant to macrocosmic planning issues that cross the boundaries of heterogeneous application systems:

- Batch data transfer will remain a common and useful basic mechanism for transmitting data between heterogeneous application systems, both within and between enterprises. However, planned, consolidated, organized approaches to batch transaction transfer (i.e., those that send all updates of the same type of information through a common logical hub) are more efficient, flexible and manageable than the customary, unplanned morass of individual file transfers that dominate most enterprises today.
- ODSs generally contain information pulled from multiple sources but are occasionally relevant within a single domain, as a shared database. ODSs address the needs of applications that require predictable, individual queries into up-to-date data.
- Data warehouses address ad hoc, read-only BI requirements and can present a unified view of data derived from many different applications systems from across the enterprise.

- Message brokers are not ad hoc. They are for operational applications that may need to update as well as read the databases that are maintained in multiple independently designed application systems.

Message brokers, data warehouses, ODSs, multitier architectures and service-oriented architectures are complementary notions; none of them will replace another during our planning horizon (0.9 probability).

Some enterprises are now planning to use a message broker to keep a data warehouse up-to-the-minute by posting changes as they occur. Such a move, however, distorts the intent of a data warehouse. A warehouse is a static store of cleansed and reconciled data. It rarely needs to be up-to-the-minute and always needs to enforce strict quality standards concerning the data entered into it. A message broker is more likely to be helpful for updating an ODS, where the data is more current and the need for elaborate cleansing and reconciliation is lower. Just as a data warehouse is capable of meeting the needs of many ad hoc queries, a message broker can often accommodate new, unforeseen information sources or consumers without having to be restructured.

6.3 Case History: Combining Service-Oriented and Message Broker Architectures

Service-oriented architectures and message brokers work well together because they both are based on the notion of interface “contracts.” The primary difference between them is that service-oriented architectures assume a consistent middleware infrastructure and do not have message transformation, flow control and adapter toolkits (for building wrappers) that make it easier to coordinate the work of heterogeneous applications. However, it is possible to merge the two concepts within a single architecture, just as an ODS and data warehouse were merged in a previous example.

MCI is one company that achieved dramatic benefits by applying a service-oriented C/S architecture to a hybrid network that includes mainframes, Unix systems and PCs. It then enhanced its configuration by adding message-broker services to the service-oriented infrastructure so that it could support new application requirements using existing application code and data. So, MCI’s deployment is incremental in the following two ways:

- It is adding new applications and new message types gradually
- It is expanding the functional capabilities of the infrastructure gradually

MCI operates in a highly competitive industry where new products must be brought to market quickly. There is no time to develop new application programs every time the marketing department devises a new way to package or price services. MCI has a large, heterogeneous computing infrastructure that includes mainframes, PCs, Unix platforms and other systems.

MCI’s application portfolio had considerable redundancy of logic embedded in applications that had been developed incrementally. For example, there were nine separate application programs that could establish a new account, which made maintenance difficult and complicated such tasks as migrating some of the customer data from VSAM and Adabas to DB2. A second problem was the connection from client PCs to the mainframe. Most C/S applications relied on 3270 data stream screen scraping, resulting in high network overhead and slow response times.

MCI's initial objective was to reduce the time needed to implement a new product by reusing mainframe program logic more often and more efficiently. Its secondary objectives were to reduce the application maintenance effort, improve response times and reduce system and network overhead.

MCI reworked its application architecture by defining certain business functions as logical services (e.g., "Establish new customer account") rather than embedding them in separate "stove piped" applications. All interactions from requesting application programs into the shared services (programs) are done using predefined, fully documented message types. The messages flow over a common, general-purpose middleware infrastructure called "Registry" that runs on 12 different OSs. MCI implemented Registry in layers. The lower layer uses a choice of message-queuing mechanisms, including a custom MVS subsystem, a custom Unix subsystem and IBM's MQSeries. The upper layer manages message status, recovery, exception reporting, application server triggering and other dialog management functions. It also does data-sensitive routing because customer data is geographically partitioned across several databases in three data centers.

Registry facts and figures

- Development started in the fall of 1993, with large-scale production beginning in 1995.
- Grew from seven product-related application system participants in January 1995 to 65 applications by December 1995, with continuing expansion in 1996.
- Runs on AIX, HP/UX, MVS, Nextstep, OS/2, OSF/1, Sun OS, Sun Solaris, Unisys, VMS, Windows 3.1 and Windows NT OSs and a total of 28 different combinations of languages, middleware and OSs.
- Currently supports more than 200 message types (services) and it is still expanding.
- Transfers 30 million online and batch transactions originating from hundreds of MCI locations throughout the United States per week.
- Largest single application is its SystemOne customer service system, with 5,000 online C/S end-users in 11 locations generating about 14 million transactions per week.

MCI achieved a dramatic reduction in the number of its application programs: 60 percent of the batch programs and 50 percent of the online programs are expected to be decommissioned in the affected application areas. For example, one service program, executing one message type, establishes a new customer account on behalf of nine requesting applications. Jobs that had previously run in batch mode were made more timely by redefining them as a series of periodic, asynchronous minibatches that call the same services that support online requests. PC-to-host communication has been converted from screen scraping to Registry message-passing for half of all the C/S workload so far (the remainder will migrate to Registry later). PC-based application tasks that required scraping as many as 19 separate screens of data have been modified to use a single pair of messages (a request and its associated reply). Furthermore, the turnaround time of an individual mainframe transaction was improved by 0.8 seconds, reducing network overhead, the mainframe processing load and the overall duration of an average customer center phone call.

MCI needed middleware that would be accessible from many different OSs and languages and from asynchronous (minibatch) and online requesters. It determined that a platform-independent message-queuing infrastructure was the best solution for its needs. When MCI first adopted a service-oriented architecture, it had to develop its own message-queuing middleware because the vendor products were

not strong enough. MCI has since decided to adopt IBM's MQSeries for most of its messaging work and is about halfway through a transition from its proprietary in-house middleware to MQSeries. This transition has been transparent to the applications and relatively simple overall because the customized Registry upper layer acts as a "super-API" layer of insulation whose "Standard Verb Set" has not changed. MCI has the option to swap in other middleware without affecting applications if it so chooses.

MCI's Registry project initially was aimed at getting the considerable benefits of a service-oriented architecture and the advantages of messaging. It found that a service-oriented architecture and messaging can be implemented separately, but the benefits of both are increased by combining them. However, MCI did not stop there. It then enhanced its infrastructure by adding message broker services. The message broker phase of the Registry is aimed at providing a layer of value-added coordination to improve customer service and implement customer-focused business processes by reusing previously developed application services in new ways.

Like most enterprises, MCI historically developed its applications as separate, distinct systems. Each application system did the account administration and other functions for one "product" (an MCI product is a particular package of telecommunication services). However, the telecommunication market is rapidly evolving to require "customer focused" processing that involves integrating data from multiple products that are related to a single customer. It would have cost too much and taken too long to write new customer-focused applications from scratch. MCI uses a diverse range of OSs, DBMSs, languages and other technologies, and its applications are written by independent development groups at five locations in the United States.

The initial implementation of the Registry provides a common API and a message-passing bus that connects a range of related applications, thus enabling the sharing of certain reusable program "services." The Registry does not yet directly provide value-added message broker services, but it provides a strong foundation for such features. In the near term, some message broker functions are implemented in MCI's Integrated Services Management (ISM) system, a separate software module. ISM is a hub that is connected to the Registry bus like any other Registry application. Its function is to provide directory services and to call a series of other Registry applications in a specific sequence to execute compound business functions that cross application boundaries.

ISM facts

- Written in C and runs on IBM SP/2 under AIX
- Its internal control tables are stored in an Oracle database on a separate HP/UX server
- Communicates with internal Long Distance, Internet, Wireless, Billing and other applications, which run on CICS/MVS and other platforms
- Is being extended to external applications in other business partners
- Includes basic message transformation to send and receive messages from product-related applications

MCI began the ISM project in January 1996 and put its first message broker application, MCI One, into production in May 1996. MCI One is a new packaging of multiple MCI services, such as paging, voice mail, Internet, mobile cellular and regular telephone numbers. It offers service using one telephone number through one telephone vendor with one bill.

MCI believes that its quick success was enabled by Registry, which provided a pre-existing middleware infrastructure and set of application connections. However, Registry is still a work in progress. MCI is exploring ways to add flow control to Registry by incorporating IBM's AIF product to decompose complex requests into separate processing steps. MCI is also investigating ISV transformation packages. Such services would expand Registry into a full-blown message broker and potentially decrease the amount of work that applications and add-on hubs, such as ISM, need to do.

The ISM project was able to show immediate value by delivering a running application quickly, even though its ultimate functionality was not completed. The highly modular, layered approach to the Registry makes it possible to add additional services without disrupting the current application base. MCI is planning to integrate the Registry/ISM infrastructure with its corporate X.500 directory, external security mechanisms and system management facilities (these functions are mostly performed outside the Registry/ISM infrastructure today). ISM imposed very little extra work on the development groups that own the participating applications because of the insulating effect of the Registry message "contracts." The interfaces (inputs and outputs) to all application programs are documented and managed by a central Registry team of 46 people (including infrastructure developers and developers who assist the application groups). However, application development and Registry usage is decentralized to autonomous IS groups.

In summary, MCI is gradually assembling a message broker by extending a well-structured messaging infrastructure. Like most successful message brokers, MCI started small and grew its broker configuration incrementally, application by application and message by message, rather than taking a "big bang" approach.

7.0 Summary of Recommendations

7.1 Architecture and Topology Planning Processes

We believe that central IS organizations have a valuable role to play in developing and disseminating an understanding of topological issues throughout the enterprise. These organizations can provide background guidance on topology trade-offs as part of their work in providing leadership in other aspects of IT architecture, such as maintaining enterprise IT standards and product short lists.

Central IS organizations are also the only ones that are in a position to actually implement the shared aspects of an IT infrastructure. These shared aspects of infrastructure are typically thought to be the central data center (with its mainframes and other systems) and the enterprise network. However, we believe that this view of "shared infrastructure" is too narrow. Central IS organizations must take a larger role in implementing some of the cross-system integration architectures described in this *Strategic Analysis Report,* specifically, data warehouses, organized transaction reconciliation configurations and message brokers. Such topologies have an impact across multiple departments within the same enterprise or across multiple enterprises. We believe that central IS organizations must even take charge of the overall management of some of the larger, cross-functional service-oriented architectures, specifically those whose scope spans multiple departments.

It is often difficult for individual divisional or departmental IS organizations to implement the macro-level architectures, because they do not control what happens in other groups and in many cases, do not even know much about what other IS groups are doing. Even when departmental or divisional IS groups are

aware of each others' projects, they may be unable or unwilling to coordinate their work because of the need to meet local targets for low cost and high (locally optimized) function.

Even central IS departments can sometimes be lead to implement suboptimal designs by the influence of disparate local development groups. For example, a large insurance company reported that it had developed a set of stovepiped applications. They had been created in response to clear and urgent needs from managers in different business units, with whom the central IS department had excellent relationships. However, IS managers were concerned about the lack of integration in these separate systems; they believed enterprises were missing an opportunity for cross-selling and defining new products. The central IS managers, however, were nervous about suggesting any additional work that might harm the excellent reputation of the central IS organization.

Their response was to appeal to the corporate strategy committee. Unfortunately, this committee was staffed with the same business-unit managers who had promoted the parochial systems in the first place. The managers felt threatened by any challenge to their responsibilities (e.g., control over sales force and data ownership issues). So the IS department continued to meet divisional needs through short-term tactical solutions, but not the long-term strategic needs of the enterprise, because no one was promoting the overall long-term interests of the IT infrastructure as a whole.

Users should expect little help on issues of macrocosmic application topology from most system vendors and ISVs. Product vendors of platforms (hardware and OSs), DBMSs, middleware and even packaged applications spend most of their attention on the local topology of their particular products and other issues of local architecture. Unless they are selling a tool that specifically targets data warehousing or message brokering, they rarely will bring up the subject of macro-level topologies. Even system integrators are often more interested in rewriting whole application systems than they are in figuring out how certain needs could be met by extending and leveraging legacy systems or other purchased applications with a modest amount of new code. The net effect of vendor pressure is therefore to divert much of the focus of the IS departments and end-user management away from macro-level topology planning issues.

Business unit managers are deluged with requests from users who have a thirst for data and applications and want the business units to fund an application implementation. Because of this demand, and with the encouragement of vendors, business unit managers often approve the purchase of hardware and software products without considering either local or macro-level design issues. This is commonly done in one of two ways: By shutting the IS department out of the process or by forcing it to create new systems that are relevant to the business unit but ignore the enterprise's vision.

When the IS department is shut out of the process, the created systems do not share data or logic with other applications. After the systems have been implemented, the business unit often discovers that it lacks the staff and expertise to support and maintain them. When this happens, one of three situations occur:

- The business unit outsources product support.
- The business unit builds its own modest IS department.
- The IS organization has to pick up the pieces and, by default, support these systems.

All three scenarios cost much more and are less flexible and durable than applications that are developed (whether within the central or dispersed IS departments) with a clear picture of how they will fit within the macro-level enterprise IT topologies.

The second method is to force the IS department to write or purchase stovepiped systems that are not easily integrated into other application systems. When this happens, it is difficult to build a data warehouse or a service-oriented architecture. However, with a modest amount of planning and work, a message broker can still be used to tie the new, stovepiped application into other application systems as long as central IS department is capable of implementing a message broker. However, it is easier and more effective to implement a message broker when it is part of the initial plan for deploying a new application system than it is to retrofit it into an application after it has gone into production.

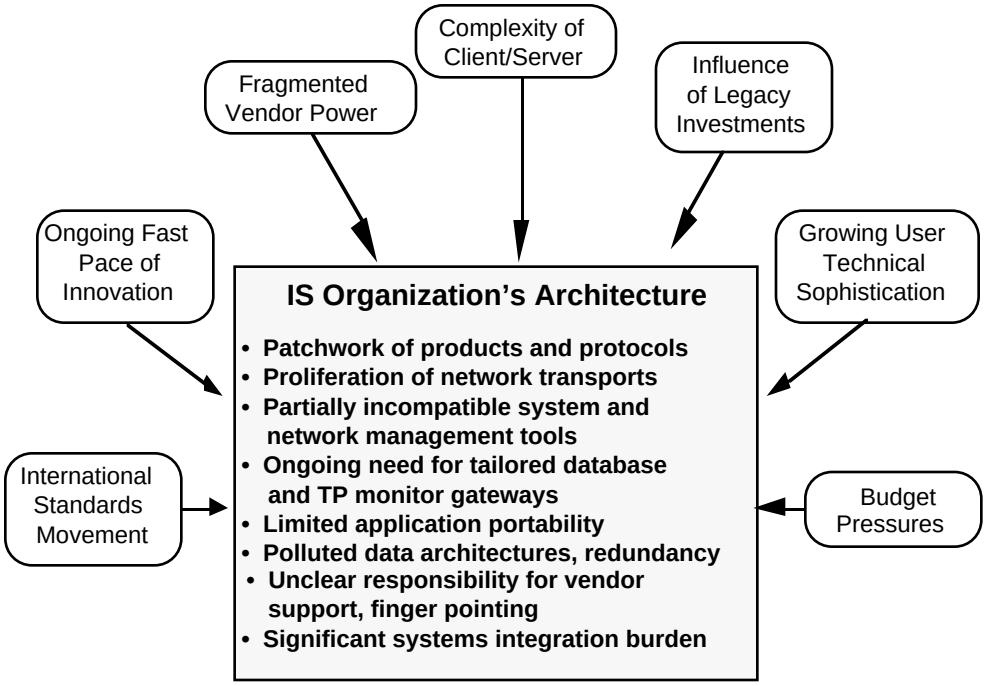
No standards organization or single vendor is sufficiently smart or well-financed to promulgate a comprehensive set of recommendations or de facto standards that can address all the architectural needs of modern enterprises. IBM tried to do this (with SAA) and failed, and it is trying again with its Open Blueprint (with about the same result). Microsoft would like to dominate enterprise computing and is putting all its bets on component software. Its ActiveX will be important as middleware, but, like SAA and the Open Blueprint, Microsoft addresses only the software technology aspects of architecture and, in particular, the relationship of modules within a single application domain. Packaged application suppliers, notably SAP, offer their own proprietary infrastructures but will be unable to impose order across applications that they do not write themselves. *Enterprises will have to deal with multiple infrastructure standards through 2001 (0.8 probability).*

None of these vendor frameworks directly addresses the major issues regarding application topology, particularly regarding the macrocosmic “city planning” issues that span multiple heterogeneous applications. *No compelling architectural leadership will emerge from any individual vendor or consortium through 2002, forcing users to plan, assemble and manage their own architectures using pieces from many sources (0.9 probability).* In a world of competing vendors and limited open systems standards, the burden for planning, organizing and managing the computing infrastructure falls to the consumer (see Figure 26). IS departments working with end-user departments must cooperate to find anchors and common threads. IT has never before held so much promise nor been so complex to handle.

7.2 Guidelines

Successful modern architectures leverage the appropriate use of the following fundamental design principles:

- Modularity
- Encapsulation
- Reuse or sharing of functions (services)
- Separation of presentation (user interface) logic from business rules, flow control and data access logic
- Server-centric processing to minimize software distribution problems and to maximize code reuse
- Incremental adoption of any desired changes in application design style or middleware



Source: Gartner Group

Figure 26. Forces Affecting User Architectures

The following general guidelines help apply the concepts developed in this *Strategic Analysis Report* to particular application deployment choices:

- Within a single application and for individual code paths in complex applications, the default choice now is to use a three-tier or multitier architecture for all but a few types of applications. Two-tier topologies should be used only for small, simple applications that use one DBMS, have a moderate-volume workload (generally fewer than 10,000 transactions per day), have little or no interapplication communication, no access to a mainframe application or interenterprise communication, have no need for a browser user interface and always use a desktop PC front end.
- For midsize or large new application systems that consist of many programs and involve multiple channels of access and overlapping logic and data, a service-oriented architecture is ideal. This is probably the single most valuable thing that an enterprise can do to build a durable and extensible application portfolio.
- A service-oriented architecture can be retroactively applied to applications, but this requires modifying applications to support the appropriate middleware interfaces and message types. Legacy and purchased applications are more readily integrated by means of a message broker because the message broker supports message transformation and flow control and has adapters that can conform to the peculiarities of foreign application systems.
- Batch file transfer is still a fundamental part of the processing in most large enterprises. It remains appropriate for data distribution within the bounds of a single application system when data does not have to be up-to-the-minute. There are times when it also makes sense between heterogeneous application systems, but it is overused in this role because of its familiarity. In some such cases, a message broker would be more appropriate because the message broker can buffer the application programs and databases from changes that may occur in other application systems.

- Real-time data integration using a data access tier or database gateway is never practical for operational (updating) applications and only occasionally appropriate for decision support. Except for a few, simple, undemanding, predictable low-volume situations, a data warehouse is a superior design for BI applications.
- An ODS is rarely practical except for predictable lookups of current data. ODSs will be more useful when they are integrated with data warehouses.
- A message broker is a general solution for integrating heterogeneous operational (read and write) production applications, although the relevant tools, techniques and middleware are immature. Leading-edge enterprises and aggressive mainstream users with the appropriate application profile can derive value from them now.
- Enterprises with successful message brokers will treat the management of interapplication interfaces as an independent discipline rather than leave this function to the individual application development groups.

We are aware of numerous enterprises that are doing projects that implement a blend of one or several of these concepts. Regardless of which new design approach(s) are being adopted, the project is much more likely to be successful if it is implemented in steps rather than with a big bang approach. All of these architectures can be approached incrementally, showing practical results from a partial implementation. The risk associated with any of them depends on exactly what choices are made in implementing the vision. In other words, none of the general topologies is inherently risky, but projects will fail if they use the wrong tools or poor application design.

Appendix: Acronym Key

3GL	Third-generation language
4GL	Fourth-generation language
ANSI	American National Standards Institute
API	Application programming interface
C/S	Client/server
CICS	Customer Information Control System
CORBA	Common Object Request Broker Architecture
CPI-C	Common Programming Interface for Communications
DBMS	Database management system
DCE	Distributed Computing Environment
DCOM	Distributed Component Object Model
EDA	Enterprise Data Access
EDI	Electronic data interchange
GUI	Graphical user interface
I/O	Input/output
IDL	Interface definition language
IS	Information systems
ISV	Independent software vendor
IT	Information technology
NCA	Network Computing Architecture
ODBC	Open Database Connectivity
OLTP	Online transaction processing
OO	Object-oriented
ORB	Object request broker
OS	Operating system
OSF	Open Software Foundation
OTM	Object transaction middleware
PC	Personal computer
RDBMS	Relational DBMS

RPC	Remote procedure call
SQL	Structured Query Language
TCP/IP	Transmission Control Protocol/Internet Protocol
TP	Transaction processing
VSAM	Virtual Storage Access Method