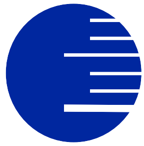
Strategic Technologies



Benchmarking Partners

Marketplace Modeling

The demand for plug-and-play business applications is strong, yet the available options are limited. A new discipline is required to achieve the vision of application interoperability. *Marketplace modeling* is a unique design approach that could finally allow users to combine best-in-class capabilities from multiple application vendors.

Volume 1, Issue 2 *June 1996*

Contents

Executive Summary 2
Introduction 3
The Demand for Plug-and-Play Software 3
Vendor-based Integration 3
Interoperability Initiatives 4
Technical Challenges 4
Best-in-Class Practices 5
Modeling Goals 6
Service, Contract, and Marketplace Modeling 7
Systems Integration and Interoperability 15
Summary of Modeling Techniques 25
Best-in-Class Systems 25
Implementation Technology 26
Technology Impact on Modeling 27
Conclusion 32
Glossary 33
Acronym Guide 34

Benchmarking Partners is an industry analysis and consulting firm. The firm is the premier provider of strategic advisory services that enable clients to achieve strategic and tangible ROI by applying best-in-class business practices and best-in-class information technology across the global value chain. The clients of Benchmarking Partners are leading multinational manufacturers, distributors, retailers, health care providers, financial service companies, and information systems suppliers. In each part of the supply chain, the Benchmarking Partners team of specialists sets the standard for in-depth, industry-specific expertise.

Benchmarking Partners, Inc. publishes Supply Chain Strategies for the sole use of Benchmarking Partners Annual Strategic Advisory Service members. It may be duplicated and distributed to employees of the member organization only. Any other duplication, reproduction, or distribution without prior written permission of Benchmarking Partners is prohibited. Copyright ©1996.

Reported information and opinions reflect the most accurate information available at the time of publication and are subject to change.

For more information, contact Benchmarking Partners, One Main Street, Cambridge, MA 02142 USA. Tel. (01)617/225-7800.
Internet: info@benchmarking.com

Executive Summary

Multinational corporations require business applications produced by multiple suppliers. To interface this software to their existing information systems, these organizations currently must pursue custom projects, resulting in increased costs, implementation time, and maintenance effort. As a result, many have demanded that future applications include built-in integration interfaces, allowing *plug-and-play* compatibility.

Some business application vendors have formed alliances to offer complementary products as an integrated suite. However, these fall short of true interoperability. Vendor consortium efforts focused on interoperability, such as the Open Applications Group (OAG), have made only limited progress.

To carry interoperability beyond the proof-of-concept stage, developers will need to employ interface design techniques that specifically address interoperability issues. One such approach is called *marketplace modeling*. This study describes how marketplace modeling differs from existing modeling approaches, and evaluates how it could be used with popular integration frameworks (such as OMG CORBA and Microsoft DCOM) to build interoperable networks of applications.

©1996 BY BENCHMARKING PARTNERS, INC.

Introduction

The Demand for Plug-and-Play Software

As organizations acquire and deploy business application software, most find that a single vendor is unable to meet all of their needs. Every company has special demands, which could be addressed by custom, best-of-breed, or industry-specific solutions.

With very few exceptions, however, business applications have not been designed to be integrated with each other. It is usually up to whoever is deploying the solution to custom configure and program interfaces that allow them to communicate. Custom integration carries many penalties: increased implementation time, higher project cost, and ongoing maintenance responsibilities.

It is only reasonable, then, that customers of major business applications have begun to demand pre-integrated software. They visualize a universe of plug-compatible modules, which would allow them to choose the best applications from each supplier.

Vendor-Based Integration

Some application vendors have responded by establishing alliances to create integrated packages. For example, PeopleSoft has embedded technology from Red Pepper Software in its application suite to perform advanced planning. SAP has defined standard application programming interfaces (APIs) for its R/3 package to work with imaging, plant automation, and warehouse communications. Datalogix and IMI System ESS interact with Oracle Financials.

Vendor API-based solutions solve many of the problems of integration, but they do not achieve the goal of plug-and-play software. All are proprietary,

©1996 BY BENCHMARKING PARTNERS, INC.

because customers have only a limited ability to substitute alternative applications. If a company selects a vendor who is not an alliance partner as one of the components of their application environment, it must pursue integration as a custom project.

Interoperability Initiatives

A variety of business issues have inhibited the development of a plug-compatible business application marketplace. Ironically, the applications that are most crucial to the process–those with a large installed base–are the least likely to be integrated. Integration capabilities bring the threat of rapid displacement by best-of-breed solutions. Historically, vendors have tried to add their own best-of-breed features instead of integrating with others.

However, there has been some recent progress. The Open Applications Group (OAG), a consortium of major software vendors, has begun to define standard transactions that link financial and manufacturing packages. The OAG Integration Specification (OAGIS) that the OAG has published to date is too narrow in scope to serve as the foundation for integration projects. Nonetheless, some vendors have committed to supporting OAG interfaces, and others continue to join the group.

Technical Challenges

Just as business barriers seem to be breaking, technical challenges rise in their place. There are no guidelines for creating interfaces for interoperable business applications. The OAG's approach defines interfaces around a few transactions (such as posting inventory receipts to the general ledger) that have widely agreed-upon semantics. Broader integration must address semantic variations, differences in application module boundaries, and the need to view a set of applications as a single, integrated system.

Object-oriented design, with its emphasis on modularity and data hiding has promise, but existing object modeling approaches used for building

applications or integrating them at a single site are inadequate. Object-based standards for integration from the OMG and Microsoft (CORBA and DCOM) also have limitations when applied to the problem of interoperability. A new modeling discipline is required, explicitly focused on achieving interoperability.

This paper assesses best-in-class practices for application interoperability, and introduces a business object modeling technique that addresses many of the problems with existing approaches. It also evaluates CORBA and DCOM as technologies for the deployment of interoperable environments.

What is a Business Object?

Business objects are independent building blocks that mirror the features of an enterprise: its business documents (invoices, orders, and bills of lading), physical resources (materials, personnel, and facilities), and logical structures (departments, product lines, and operating concerns). Objects within a system (such as an order and its associated material) communicate with one another to execute business processes. Application functions and data are hidden (“encapsulated”) behind the objects’ interfaces, giving system designers more freedom to incorporate new technologies without changing the way users interact with the objects.

©1996 BY BENCHMARKING PARTNERS, INC.

Best-in-Class Practices

Modeling Goals

One lesson that can be learned from the vast literature on object modeling techniques is that no specific modeling technique applies to all situations. Not only should the object modeling technique vary depending on the project stage, it should also vary depending on the goal of the project. These goals may be classified into three categories of modeling:

- *Analysis:* understanding or explaining a business problem
- *Implementation:* designing and developing a system or a part of a system
- *Integration:* creating an interface between component applications

Much of the popular literature on object modeling discusses the second category: implementing a system design. There are also prominent studies of object-oriented requirements analysis (the first category). However, the third category, modeling to create interfaces, is critical to application-level interoperability. This topic has remained largely unexplored.

To develop a viable interface model, it is important to first explore how and why interface modeling differs from implementation object modeling.

©1996 BY BENCHMARKING PARTNERS, INC.

Object Terminology

Unfortunately, the popularization of object technology has resulted in a wealth of terms with incompatible definitions, including *client*, *interface*, and *implementation*. (See Glossary.)

In discussing modeling in this paper, the term *client* refers to program code that makes use of a function or group of functions. The client sends requests to an implementation of the functions, using the rules specified in the interface.

The term *implementation* refers to a program code that implements an interface. All of the program code together is considered the implementation.

An *interface* refers to a complete set of rules for accessing a function or group of functions.

Service, Contract, and Marketplace Modeling

Service Modeling

The goal of service modeling is to create a system or part of a system. Most conventional object modeling techniques fall into this category. In this context, objects are designed as bundles of functionality. Objects can be thought of as service providers, or perhaps miniature machines.

Attributes and Restrictions

Service modeling represents a *tools-based* approach. Its attributes and restrictions include the following.

- *Rich Functionality*

  In service modeling it is generally desirable to make each object as rich in functionality as possible to satisfy users. If, for example, a development group in charge of designing and implementing billing objects adds attributes and operations to support alternative payment terms, those who use the billing objects will be happier.

©1996 BY BENCHMARKING PARTNERS, INC.

- *Reuse of Difficult Implementations*

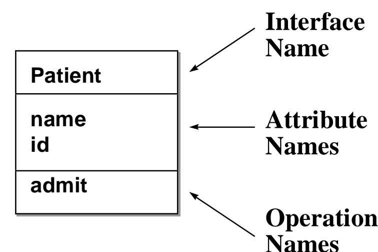
  One of the principal goals of service modeling is reusing code, which refers to implementing objects only once and reusing them whenever applicable.

- *Cooperative Use*

  In service modeling, designing frameworks that require the clients to behave correctly to guarantee the integrity of the objects is considered acceptable. For example, it is common in service models to specify operations such as *create <object>,* and *destroy <object>,* and require the client to destroy the *<object>*'s state when finished with it. An uncooperative client who goes into an infinite loop calling *create <object>* can crash the implementation.

Object Modeling Notation

Objects can be diagrammed in a variety of ways. In the Rumbaugh notation (used in this study), object types are represented by rectangles. The figure below shows a simple object type that uses this notation.



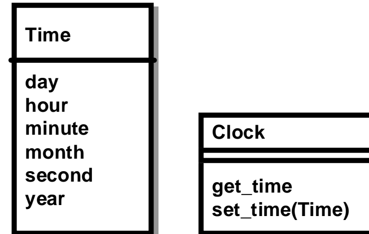
The uppermost section of the rectangle is reserved for the object type name. The middle of the rectangle lists the attributes (data properties) that the object contains. The bottom section lists operations: functions that can be called to manipulate or access the object's state in more complex ways.

The complete set of features illustrated for an object type is called its interface. Objects can only be accessed via the interface; all other details about the object are private.

An Example

In defining a service model, a simple clock service can be used as an example. At a minimum, this clock service should be able to set or get the time of day, so the first cut might resemble the model shown in Example 1.

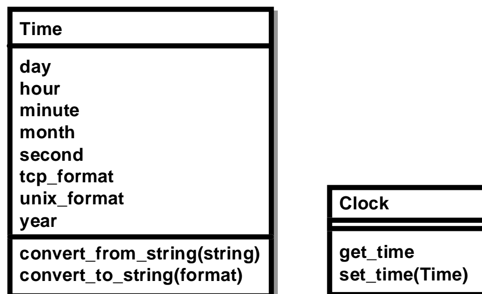
Example 1
Clock Service



Note that as a service model, the clock interface can be visualized as an actual clock or as a control panel wired up to an actual clock. The time can be read by calling the *get_time* operation or set using the *set_time* operation.

To be an effective implementation, the clock service should be more general and functional. The first consideration might be to allow a wider variety of output time formats. One way of achieving this would be to define a *time* object that could perform conversions. The refined interface is shown in Example 2.

Example 2
Clock Service with Formatting Capabilities



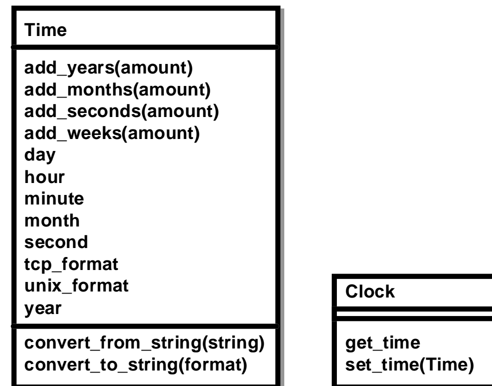
With this new interface, the time can be read as year/month/day/hour/minute/second, or as the number of seconds since January 1, 1900, or as a text string in a user-defined format, such as, “January 03,

©1996 BY BENCHMARKING PARTNERS, INC.

1996." The time can also be set any number of ways. This is clearly a more powerful service model than the first one.

As a last refinement, the ability to set the time as a change from the current time may be a desirable addition. For example, perhaps the clock could be turned back an hour for a seasonal time change. Given the latest interface, it makes the most sense to add these features to the *time* object. The final service model for a clock is illustrated in Example 3.

Example 3
Clock Service with Formatting and Relative Time Calculation



A clock service that conformed to this interface would be a useful, flexible tool. If a development group produced an implementation, clients would probably use it whenever they needed a clock service, which would provide the benefit of code reuse. For these reasons, this particular clock service is a good service model.

Contract Modeling

Contract modeling is critically important to the software marketplace. It is also the cornerstone of marketplace modeling, the interoperability architecture presented later in this study.

In contract modeling, the designer still defines sets of objects that work together to solve problems in a problem domain. However, the primary goal is to produce a contract, not a piece of functionality. This approach might also be thought of as a standards-based approach, in contrast to the

tools-based approach. Although the goal is superficially very similar to the goal of service modeling, many subtle differences exist.

To explain the difference between contract and service modeling, the example of an automobile design project follows. In this case, the automobile's design is almost complete, but the way to pour in fuel has not yet been designed. A service modeler would decide that there should be as many ways to fuel the car as possible, so that the driver could add fuel under any circumstances. In order to do the best possible job, the service modeler would redesign the engine so that it works with diesel, propane, hydrogen or conventional gasoline. The modeler would make a large hole in the gas tank to make sure that any size nozzle would fit. In addition, for extra flexibility, the modeler would add a nozzle with a built-in pump to the car, in case the driver needs to refuel directly from a gasoline truck that does not have its own nozzle.

The modeler could rest, having done a complete job. The department in charge of designing gas station pumps could interface to the car with a minimum of effort. The pumps designers would be very thankful that their job was simplified.

On the other hand, a contract modeler would have a completely different viewpoint and goal. The job would not be to produce any actual machines (except for proof-of-concept purposes). Instead, the contract modeler would assume that other organizations would produce the automobiles and gas pumps. The modeler's job would be to write a manual for the automobile producers stating the steps they must follow to ensure that the car can be refueled. The modeler then would produce a second manual for the gas pump producers stating the steps they must take to make their gas pumps work on all automobiles.

If the contract modeler decided that the automobile producers have to make cars with multiple fuel capability and that they had to provide a large fuel entry hole as well as a hose and nozzle, the automobile producers would not be thankful. The fuel pump producers would probably not be thankful either, since they would be unsure how to create good fuel pumps. The

©1996 BY BENCHMARKING PARTNERS, INC.

situation would likely result in wasted effort, because there would be no simple, clear, minimal contract between the auto makers and fuel pump makers.

On the other hand, if the auto manual said simply “cars must have a hole in their gas tank that is 2.25-2.50 inches in diameter,” and the fuel pump manual said “fuel pumps must have a nozzle 1.75-2.00 inches in diameter,” the modeler would have catalyzed the industry. The rules would be simple to follow for both automobile and fuel pump makers, and as long as everyone followed the rules, there would not be any compatibility problems.

Contract versus Service Modeling

- *Simple Contracts versus Rich Functionality*

  When creating contract models, it is not generally desirable to make each object as rich in functionality as possible. Instead, designers must make the contract between client and provider as simple as possible.

- *Choice of Implementations versus Reuse*

  Typically, each interface will have many implementations and many users, and these users will be in different enterprises. As a result, a marketplace evolves around the interface.

  The goal of contract modeling is not to reuse a given piece of code as much as possible, but rather to make sure that users can choose from a wide variety of implementations.

  If the interface is too difficult to implement, the marketplace will suffer because few will be able to create implementations. If the interface is easy to implement but not useful enough, the marketplace will suffer because few will be interested in using the implementation. Therefore, a careful balance must be struck. While service modelers must be careful to expose only the features that they can implement, contract modelers must specify only those features that would be reasonable in *any* implementation.

©1996 BY BENCHMARKING PARTNERS, INC.

- *Interface Design*

  In contract modeling, reliable interfaces must be designed. If clients of the interface operate incorrectly, they *must not* be able to inhibit the correct operation of the implementation. This is crucial since implementations may be shared by unrelated clients.

  The responsibility of the interface designer is to make it possible to create perfectly reliable implementations. Obviously, some providers may create bad implementations, but the designer must make sure that the interface design does not preclude the possibility of building good ones.

  In sum, designers must account in the design for adversarial clients who will try their best to crash the implementation. This is in contrast to service modeling, where designers can assume that the client is cooperative.

Contract Modeling in Perspective

Good contract models make it possible for users to write code that will be adaptable to future technology. For example, code written today to open database connectivity (ODBC) and Winsock interfaces will work with many of tomorrow’s databases and TCP/IP networks, because of wide agreement on a minimal contract between implementors and users.

Contract models also make it possible for implementers to write more general-purpose code. If an implementation team creates a new database and writes an ODBC driver for it, its database will automatically work with many pre-existing applications. In sum, contract models provide points of interoperability and help the software marketplace operate more efficiently.

Contract models often take on a life of their own. Once there are thriving marketplaces of implementers and users, the interfaces themselves resist change. New implementations of the interface cannot remove or change functionality without endangering compatibility with the many existing clients of the interface. Implementers also cannot add functionality because responsible users avoid new functions in order to make their code work with as many implementations as possible. This reaction reduces the

incentive to add extra functions. However, if a new, very compelling capability is added to an interface, people will be willing to test for the presence of the capability and have their code work both ways. When the Netscape Communications Corporation introduced nonstandard but compelling extensions to HTML, many implementations soon adopted them.

Contract Model Example

A contract model of a clock is a useful example of the contrast between contract and service modeling. For this clock, the goal is still to make a powerful, useful model, but the rules are different. A simple contract will make clocks straightforward to implement and to use. It will also define a marketplace in which people could compete to produce and sell clock implementations. Users can design their programs, and then choose a best-of-breed clock implementation later, knowing that it will be compatible.

To design the clock service, the modeler should first investigate whether there is a representation of time that is standard or very popular. If there is a single popular standard that would satisfy most users, there is no need to add anything to the model, since that would just make it harder for implementers and not add significant value to users.

For this example, one might consider the TCP/IP standard (seconds since 1/1/1900) or the UNIX standard (seconds since 1/1/1970), which are both in wide use. Alternatively, one could use the relevant ISO time/date representation standard (ISO 8601) or extract standards from time-related protocols such as NTP, which is used for synchronizing clocks across the Internet. Formal standards such as ISO 8601 have the advantage that many people have invested a great deal of time thinking about issues that may not be obvious at first. However, ISO 8601 in particular allows too many ambiguities and partially specified dates for this clock service example.

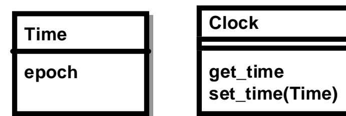
For this clock service, the TCP/IP standard is adequate.

The final representation may be:

©1996 BY BENCHMARKING PARTNERS, INC.

Example 4

Minimal Clock Interface



This is now a very simple implementation. It may not meet all requirements for all users, but it is very much in the spirit of the auto refueling solution discussed earlier. Although it does not seem to be a particularly powerful interface, a programmer writing a real production application would have no trouble using this as a clock service. Furthermore, companies in the business of writing clock-related software would find this interface simple to support.

This example may appear simplistic, because a clock is a relatively simple device. However, when modeling complex applications with tens of thousands of functions, keeping the interface minimal is even more critical.

Common Ground between Service and Contract Modeling

Although the differences between contract and service modeling are significant, they have much in common; in fact, much of the current literature about object modeling applies to both types. Both attempt to manage complexity by using techniques such as modular decomposition, abstraction, and hierarchy. Both modeling techniques stress the separation between interface and implementation, thereby isolating the client from irrelevant implementation details.

The differences lie only in the subtle decisions about which functions belong in each interface. By foregoing non-essential features, the modeler arrives at a simple, widely implementable interface.

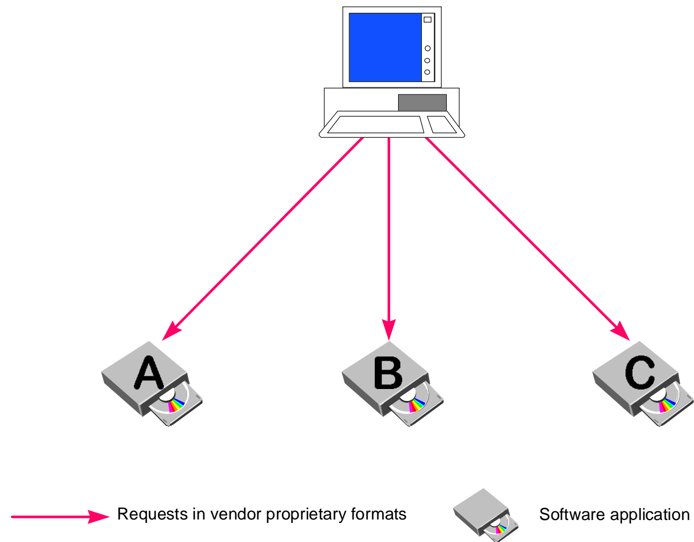
Systems Integration and Interoperability

While contracts must be made at the component object level, users experience integration at the application level. The goal of systems integration is to make a set of software applications work together to provide more value than the individual applications could separately (as

©1996 BY BENCHMARKING PARTNERS, INC.

shown in Figure 1). This value can be added by creating new functions or reconfiguring the applications.

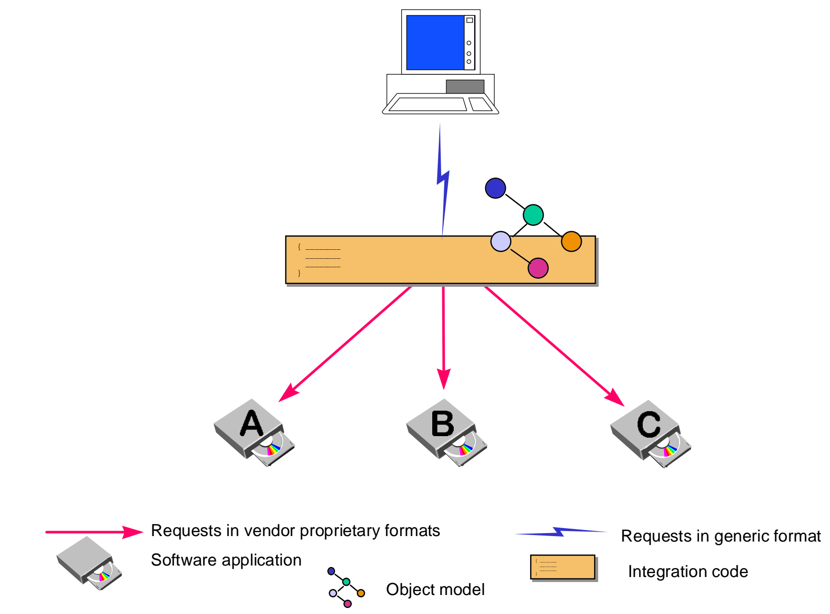
Figure 1
Application Integration



Source: Benchmarking Partners

The result of creating new functions that are not provided by any of the individual systems is that functions that require processing by more than one of the applications are handled by the integration code, and not by the applications themselves. These *delta* functions result in a superset system. Figure 2 illustrates an object-oriented integration environment with delta functions.

Figure 2
Model-Based Integration



Source: Benchmarking Partners

The second strategy for systems integration is to configure the applications so that they automatically remain synchronized. An example is a set of applications: one handles production scheduling, while another performs maintenance. The integration code could be designed so that if a task is scheduled in one application, the other application is automatically notified and marks the appropriate resources as unavailable. In such a case, no new functions are created, but the integration code adds value.

Since the integrated system will have capabilities (i.e., delta functionality) that were unavailable in any of the original applications, additional interfaces for the integrated system must first be defined. Contract modeling can be used to design these interfaces. The integration can then be performed by creating an implementation of the new contract models.

For example, the scheduling system could be integrated with a cost accounting system to determine detailed costs for specific tasks. Calculating the costs incurred for a resource during a specific time period would require a new delta function.

Once the implementation is complete, programmers can write integrated applications. If they are using a delta function, they use the rules specified

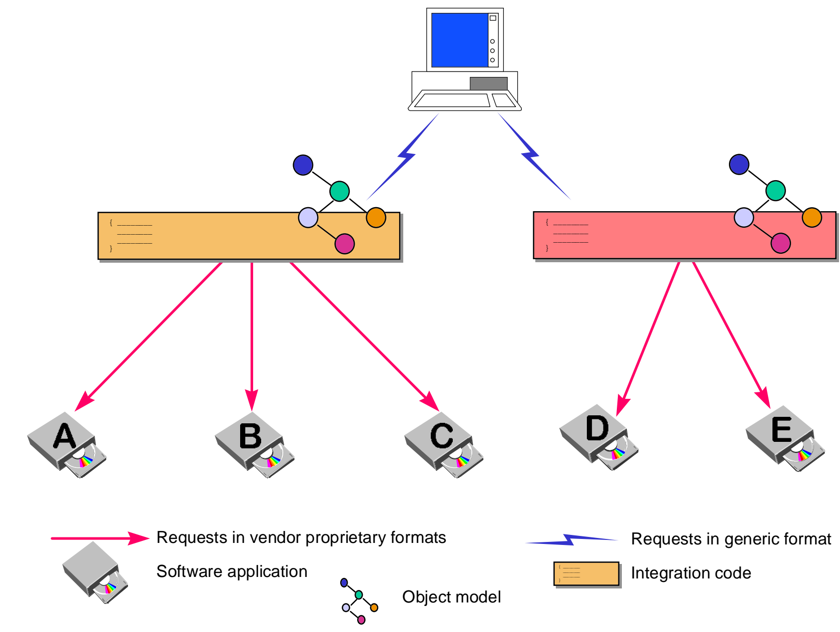
in the contract model. If they are using a nondelta function (calculating the standard cost for a resource, or scheduling a task), they use the original proprietary API provided by the vendor. The provision of contract models for nondelta functions would certainly be beneficial as well, but development of these has a much lower priority. It is the delta functions that the client currently cannot perform at all.

When the contract model for the integration is complete, it is possible for a marketplace of integrations to emerge. Another vendor can replicate the integration (conforming to the same contract model and using the same proprietary APIs), add higher performance or reliability, and then charge more money for it. Furthermore, vendors can integrate different but equivalent applications using the same contract model but with different proprietary APIs. In such a case, the integrated applications can in theory work without change on a different set of underlying applications. However, to the extent that the integrated applications use nondelta functions, they will not be able to switch easily to a different set of applications.

If the contract models are made powerful enough that useful integrated applications can be written using only the contract models, the contract models will work with *any* set of underlying applications. It will be necessary only to create new implementations of the integration for each set of applications. Each integration must conform to the contract model and must use whichever proprietary APIs are appropriate for the underlying applications. Figure 3 shows two alternative implementations of the same contract model, one supported by three applications, the other by two applications.

©1996 BY BENCHMARKING PARTNERS, INC.

Figure 3
Alternative Implementations of a Single Contract



Source: Benchmarking Partners

From Integration to Interoperability

Any code that uses contract models instead of proprietary APIs becomes more widely applicable and longer lasting. For example, an integrated application can be written once and work with a wide variety of underlying applications if it makes calls that conform to the contract model only.

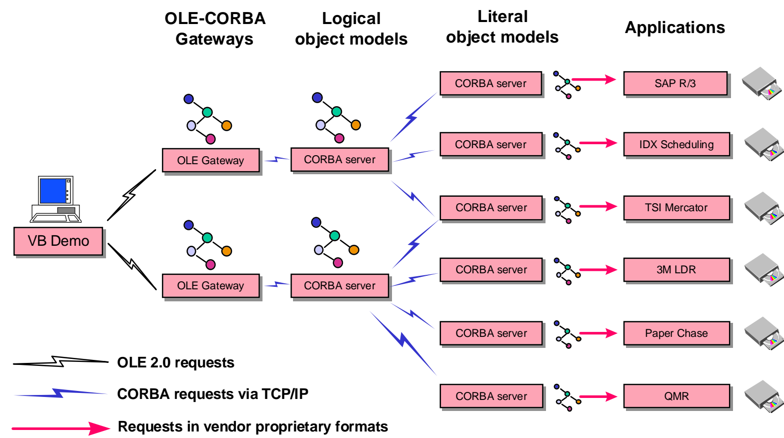
The difficulty is that in order to change the set of underlying applications (for example to another set that provides equivalent functionality), a new implementation of the integration has to be created. This is typically a large development effort.

The idea of interoperability is that an integration can be performed once, then can be used with a variety of combinations of underlying applications. This makes sense in theory. After all, if one integrates a scheduler and a costing system using specially designed logic, it ought to be possible to substitute a different scheduler without having to rewrite the integration logic substantially.

Contract models allow clients (that is, anything or anyone consuming a standard) to be widely applicable and adaptable to future technology.

Furthermore, contract models allow for the existence of a marketplace of substitutable implementations. The solution to interoperability is therefore to introduce another layer of contract models, and make the integration logic use only contract models to access the underlying applications. For example, the delta function discussed earlier, which calculates costs for resources used in an appointment would use the underlying contract models for costing and scheduling, respectively. The integration logic becomes a client of the new contract models, and the underlying applications become the marketplace of substitutable implementations. Figure 4 illustrates a multilayer integration environment that uses contract models at two levels to achieve interoperability.

Figure 4
Interoperability Architecture



Source: Benchmarking Partners

Marketplace Modeling

Creating layered object models that promote interoperable, integrated systems is called *marketplace modeling.* A service that can be implemented several times by different vendors, and offered at different levels of quality and price, is called a *marketplace*. Users can first design systems that use this service, then pick a vendor. As long as all users and implementers follow the relevant contract models, all systems will be compatible.

©1996 BY BENCHMARKING PARTNERS, INC.

Each contract model makes a miniature marketplace possible. Consequently, an objective of marketplace modeling is to use contract models as often as possible to promote miniature marketplaces. This is good for everyone: large vendors can create implementations for many marketplaces, small vendors can focus on a single marketplace, and programmers can choose best-of-breed technology for each miniature marketplace.

For interoperable systems, a marketplace of integrations and a marketplace for each underlying application type is desirable. Such integration would allow a company to implement a costing workstation in-house, buy a scheduler from one vendor, a costing system from another vendor, and an integration from a third vendor—all of which would work together. If a better integration were developed, the company could purchase the new integration, which would automatically work with the existing costing workstation, scheduler and costing system. Similarly, the scheduler would be replaceable without impacting the rest of the system.

It follows that interoperability requires layered contract models. The implementation of high-level contract models works by combining program logic with calls to lower-level contract models. In the company case scenario above, for instance, the following would be required: a contract model for the integration functionality, a contract model for the scheduler, and a contract model for the costing system.

In sum, marketplace modeling means using layered contract models to promote an efficient software marketplace built around individual business objects.

Literal and Logical Models for Interoperability

Although the general principles of contract modeling always apply, different refinements are appropriate at each level of contract-model layering. Based on the nature of these refinements, contract models can be classified as follows.

©1996 BY BENCHMARKING PARTNERS, INC.

Literal Models

Low-level contract models that are designed to promote marketplaces around individual applications (schedulers, clinical data systems, etc.) are called *literal models* because they are not very abstract. Marketplaces at this level are available already; for instance, a hospital can shop for a "scheduling system." However, these marketplaces do not work smoothly, since one scheduler cannot be easily substituted for another.

Low-level contract models are usually created by examining multiple applications in an existing marketplace and modeling an archetypal version of the application. Powerful new abstractions are not desirable for this level; rather, the goal is to standardize an abstraction that is already crudely formed. Consequently, the fairly literal expression of the underlying applications is termed *literal model*.

Logical Models

Higher-level contract models specify complete integrated systems and provide abstractions that cross the boundaries of the underlying applications. These are called *logical models,* since they are abstractions over the lower-level models. In order to be widely applicable and compatible with future technology, implementations of logical models must never use proprietary APIs of specific applications. Rather, the implementations of logical models should be a combination of logic (calculations, decisions, and sequencing) with calls to implementations of literal models. Business objects, such as orders, customers, or products play a role in logical models.

Traditionally, business applications have been decomposed along functional lines. They perform billing or scheduling, rather than taking a global business object view of the scheduled and billed for item. Literal models require object models that respect the functional boundaries of the applications they represent. Only limited concepts of each business object are visible. For example, the literal model for scheduling could include a resource object, whose meaning could be "important item that must be scheduled." The literal model for billing could contain a shipment object

©1996 BY BENCHMARKING PARTNERS, INC.

whose semantics could be "important item on whose behalf bills must be processed." Similarly, the idea of a shipment business object exists in the logical (integration) contract model. The business object implements all business concepts related to the idea of a shipment, and the implementation combines integration logic with calls to the literal objects related to shipments.

In sum, literal and logical models work together to construct marketplace models in two levels, promoting marketplaces of interoperable applications and integration frameworks.

Responsibilities and Benefits of Marketplace Modeling

Once literal models are accepted in the marketplace, individual application vendors should become responsible for the creation of implementations of the literal models relevant to their applications.

Once that occurs, learning the APIs of specific applications will be unnecessary. If one needed to interface to a scheduler, one could call functions using the rules in the scheduling literal model. Any scheduling application could then be used, as long as the vendor provided an implementation of the scheduling literal model.

The implementation by vendors of relevant literal models would, therefore, become the requirement for conformance to the framework standard.

Literal Models as Application Subsets

Since applications can be very complex and different from one another, it may seem impossible to create good literal models. However, the task of creating literal models is manageable because it is not necessary to model whole applications. It is only necessary to model those parts of the application that must be accessed electronically (by another application or by a logical model).

For example, it may be necessary in the model for a patient scheduling system to include functions for creating and looking up patient appointments. However, including functions for entering a new patient into the system may be unnecessary. Instead, policy can ensure that new

©1996 BY BENCHMARKING PARTNERS, INC.

patients are always entered using the original client interface. Or perhaps an electronic interface can enter new patients while the original interface backs up and restores the database. The literal models need to deal only with selected subsets of the application functionality.

Fine-Grained Literal Models

The wide variety of application capabilities provides another method for simplifying literal models. In some cases, it may be impossible to come up with sensible literal models that fit the applications. The solution is to choose finer-grained literal models allowing the underlying applications each to implement more than one model.

For example, if application A implements patient scheduling and clinical data functions, and application B implements clinical data and financial functions, the solution is to define three literal models (patient scheduling, clinical data, and financial). Then application A implements the first two models and application B implements the last two.

Modeling Paradoxes

The main difficulty in creating literal models is the wide variation in application capability. A model with fewer features enables conformity with a greater number of applications but will be a less powerful model. A model with more features will be more useful but will enable conformity with fewer applications. This paradox is typical of contract modeling.

In the long run, it may be necessary to introduce levels of conformance. However, if there are too many different levels of conformance, the marketplace can easily be damaged, since few vendors will implement the highest conformance level, and users will lose confidence in the model. On the other hand, if there are too few levels of conformance the resulting model will be the least common denominator.

The simplest approach is to keep things simple by having only one conformance level for each literal model, and go with a least common denominator approach. If simple literal models suffice for doing reasonable

©1996 BY BENCHMARKING PARTNERS, INC.

integration projects, the concept will be proved, and others can develop more elaborate literal models.

Summary of Modeling Techniques

Most object modelers today are involved in service modeling. Contract modeling emphasizes different priorities. Contract modeling is the best approach for creating interfaces, rather than systems.

Using contract models in a layered fashion promotes interoperability and a fine-grained, efficient marketplace. The concept of layering contract models is termed marketplace modeling.

A simple use of marketplace modeling is a two-tiered structure of contract models. Low-level contract models (that is, literal models) are used to promote application interoperability. High-level contract models, (that is, logical models) are used to promote interoperability of system integration software.

Logical models contain business objects. Literal models contain limited versions of business objects, as they pertain to particular application functions.

©1996 BY BENCHMARKING PARTNERS, INC.

Best-in-Class Systems

Implementation Technology

Designing “universal” business objects—such as schedules and resources—that work across multiple implementations is the single greatest interoperability challenge. Technological details rarely come to play in the selection of the features that will appear in an object’s interface.

However, technology eventually comes into the picture. Once the standard schedule is defined, the rules need to be specified. For instance, how does a client request a schedule? How does a scheduling application receive the request and return a schedule?

Since this technology selection will become part of each interface, the designer must be careful not to accidentally create platform limitations. Engineers should be able to create conforming scheduler implementations in any programming language or operating system. Furthermore, they should be able to send requests from any programming language or operating system. Steering away from such limitations would provide the maximum amount of choice to users and the largest possible market for implementers.

Although it is reasonable to design ad hoc TCP/IP based interfaces that can operate almost anywhere (many popular protocols have been designed this way), higher-level tools that can read simple interface specifications and generate the appropriate networking code are available. High level tools are a good approach if the interface specification is powerful enough, published, and widely accepted, and if the TCP/IP message formats are also published and widely accepted.

©1996 BY BENCHMARKING PARTNERS, INC.

Candidates for specification standards include:

- OSF DCE: A system based on remote procedure calls
- OMG CORBA: An object-oriented system in which object references can be sent from system to system and method invocations can take place across application boundaries using TCP/IP
- ISO OSI Abstract Syntax Notation One (ASN.1) over an asynchronous message passing technology based on TCP/IP: ASN.1 is a generic notation standardized by the ISO for the specification of data types and values; frequently used in telecommunication protocols and applications
- XDR over an asynchronous message passing technology based on TCP/IP: XDR is another standard for the description and encoding of data, similar to ASN.1 but differing in the use of implicit versus explicit typing

Undoubtedly, many other possibilities exist. CORBA is the most attractive specification standard because it is object-oriented, has wide industry support and many commercial vendors, and has recently come out with a standardized TCP/IP encoding scheme. The object-oriented nature of CORBA makes it reasonably expressive, and also makes it work well with currently popular object-oriented design tools.

Technology Impact on Modeling

Ultimately, technology has a big impact on object modeling. It is naive to claim that all objects are essentially the same concept. A cursory examination of several object-oriented languages (for example C++, Smalltalk, CLOS and Self) will reveal vastly different concepts about what objects really are. When designing a large interoperable system, the middleware technology must be chosen before the detailed contract models are designed. The middleware technology will determine the set of idioms that can be used safely in the contract models.

©1996 BY BENCHMARKING PARTNERS, INC.

Impact of CORBA on Modeling

Resource Management

Models must make it possible to design perfectly reliable implementations. Resource management is problematic in CORBA because it does not provide any mechanisms that facilitate the task of distributed resource management.[1] If a CORBA server creates an object that exists only to service a particular client during a particular session, the server is dependent on the client to notify it when the object can be destroyed and its resources reclaimed.

This is an extremely serious problem. If the client violates its side of the contract by crashing, for example, the server will have an object allocated that will never be freed. An adversarial client can crash the server simply by going into a loop allocating objects and not freeing them. The server cannot free the object after a time-out period either, because the client may have saved the object reference in a database and will depend on it in the future.

If a very large, complex distributed system is built using many layered interfaces, the consequences of an accidental deletion of an important object can be very serious. Furthermore, if one increases safety by increasing the "time-out deadlines," the amount of wasted storage can be very high. For example, if a single incorrectly-written client creates a garbage object every five minutes, and the time-out is one month, over the long run there will be a constant overhead of 8,640 garbage objects per running client. Reference counting is not a good solution to this problem either, since it depends on a correctly programmed client.

This rules out the popular object-oriented idiom that uses iterators to traverse complex structures. Any CORBA server that uses the iterator idiom can be crashed by an adversarial client program. Many idioms that use analogous principles or "temporary" objects are also ruled out. For example, the notion of a database query object—which is created by a

[1] The best CORBA vendors have provided some very crude tools for distributed resource management, which are not part of the CORBA standard.

client, manipulated to represent a query, and submitted to a database—suffers from this same problem.

Stateless idioms must be substituted in these situations. For example, traversing an array can be done using a count and a fetch-by-index operation, rather than an iterator.

Structured Polymorphic Values

Another failing of CORBA is that it does not support polymorphism in values. Objects are polymorphic, meaning that an object reference to a *Car* can be manipulated uniformly, whether it refers to a *Mercedes* or a *Honda*. However, there is no equivalent for values. For instance, one might want to send an e-mail message from one system to another and want to define a *Message* value.[2] In CORBA, one possible solution is to define a structured data type (*struct*). However, if one later wanted to define a *DatedMessage* as a type of message with an extra date field, it would be desirable to use some sort of inheritance mechanism for this. Then if one sent a *DatedMessage* to a server expecting a *Message*, the date information would simply be ignored. However, inheritance does not work on structs in CORBA, so this type of polymorphism cannot be achieved.[3]

Another possible solution is to use the CORBA *any* construct, which does provide value polymorphism. However, the use of the *any* type hides structure from the CORBA interface definition language. The interpretation of *any* objects is up to the runtime code in the client and server. This means that the interface must be documented in written language, and eliminates the convenience and safety of the CORBA code generation facilities.

---

[2] Not an object, since in CORBA objects are not mobile and have an identity other than their value.

[3] Interestingly, C++ object-oriented language almost permits this type of thing, although not in a distributed system. Objects can be passed by value, except that a *slicing* problem occurs preventing polymorphism from working correctly unless some special programming idioms are brought into play. To be fair, most object-oriented languages do not support *value objects* at all. It is not so important in a monolithic system, since passing by value or reference does not have the dramatic performance impact it has in a distributed system. Also, global resource management policies (such as garbage collection) are often supported, so that reference-passing disciplines do not make servers vulnerable to incorrect clients.

Finally there are some creative solutions for creating a CORBA *union* of various structs, much like the C or C++ concept of union. This is polymorphic, but loses some of the benefits of object orientation.

The Object Management Group is currently collecting proposals for extending CORBA to include polymorphic values, but it will be some time before standard implementations become available.

Impact of Microsoft COM on Modeling

Another way to design contract models and implement them is using the Microsoft Component Object Model (COM) technology found in its OLE and DCOM protocols. With this technology, one can build *components*, which are chunks of functionality that can be distributed like application software. Microsoft advertises that component technology is superior to object technology because object technology does not provide standardized, reusable chunks of functionality. Today, for example, there is a thriving marketplace of widget components for graphical applications based on OLE (ActiveX controls). One can purchase a diskette full of these and use them in a Visual Basic program.

COM achieves plug and play interoperability, albeit in limited problem domains. However, Microsoft's conclusion that components are better than objects is an oversimplification. Components are really a combination of:

- An implied contract model based on Microsoft's binary standard for OLE components
- An implementation of that implied contract model

These contract models can be identified in studying a particular marketplace of OLE components. For example, there is a marketplace of graphical listboxes. If one wants to create a new graphical listbox component and sell it, one will find that there is already an implied contract model defined (originating at Microsoft, where the first listbox component was developed). Therefore, third-party listboxes must be developed using this contract model, which implies using Microsoft's OLE technology and implementing certain specific methods and properties.

By developing and selling many OLE components along with powerful tools that use those components (for example Excel and Visual Basic), Microsoft is creating large numbers of contract models, which immediately become de facto standards. The real value of the COM technology is in these contract models. The fact that popular applications use the models validates the models and makes creating and marketing alternative implementations attractive propositions.

Microsoft is currently working on defining COM objects (contract models) for application-level functionality (e.g., healthcare components, accounting components). If they succeed in creating a viable component marketplace at this level, the application software industry will benefit greatly.

Relationship to CORBA

Technically, COM has much in common with CORBA, yet CORBA is making slower progress for several reasons. First, a well-defined point of interoperability between CORBA implementations did not exist for a long time. OMG has now defined an interoperability specification that standardizes the TCP/IP messages sent between CORBA clients and servers. Before this development, CORBA clients could only communicate with CORBA servers written by the same vendor. (In fact, that is still the case, since most CORBA vendors have not conformed to the interoperability specification yet.) COM, on the other hand, had the advantage of platform uniformity. Since most COM users are on Microsoft platforms and Microsoft is the only COM development kit vendor, no need existed for achieving consensus on the encoding schemes.

Second, perhaps because of the fragmented CORBA marketplace, no software vendor has taken the initiative to develop a powerful, universally compelling suite of UNIX applications organized as a network of CORBA components. Because no vendors have invested in this “starter set” of CORBA-based contract models, organizations have been uncertain about CORBA’s viability and have been hesitant to invest in CORBA technology.

©1996 BY BENCHMARKING PARTNERS, INC.

Conclusion

Marketplace modeling is not a radical departure from current techniques. Rather, it is a set of subtle improvements to current object modeling approaches that takes into account the functional orientation and varying capabilities of application packages.

Plug-and-play interoperability is difficult to achieve, and despite advances in methods and technology, it is likely to stay that way. Few developers are aware of the principles of constructing models as simple contracts between implementors and users. Fewer still have considered the impact of selecting a standard such as CORBA or DCOM on the inherent reliability of their distributed designs. The hope for interoperability hinges upon the exploding demands of the Internet, and a new generation of tools, which use dynamic specification, interpreted execution, and distributed garbage collection to create flexible, reliable systems.

©1996 BY BENCHMARKING PARTNERS, INC.

Glossary

Service A useful function or group of functions.

Interface A complete set of rules for accessing a service. Interfaces can be specified in many ways: Interface Definition Language code of some kind, pictures in an object modeling tool, or lists of function headers. All of these techniques must be supplemented by written explanations, since they are usually incomplete by themselves. Interfaces are usually optimized either for use by humans (e.g., graphical user interfaces) or by programs (e.g., application programming interfaces).

Model Synonym for interface. The term model suggests a graphical representation of the interface, but this is not always the case.

Implementation A program that implements an interface. All of the program code together is considered the implementation. For example, if the implementation is written in C++, the header files, source files, make files and any other related files, taken together, constitute the implementation. (Header files are sometimes considered an “interface”; that definition is not used in this paper.)

Client Program code that makes use of a service. The client sends requests to an implementation of the service, using the rules specified in the interface. If the client and the implementation both follow the interface rules in the interface, they will work together properly. If a better implementation comes along, the client can presumably switch to the new implementation since the new implementation must conform to the same interface.

User A person operating a client application.

Class A term whose many inconsistent definitions render it useless. This paper only refers to the very specific term C++ class.

Modeling The act of creating a model (interface).

Service Modeling Creating a model (interface) with the goal of implementing a system or part of a system. This term is intended to refer to traditional object modeling as it is described in the literature. The name service modeling is derived from the assumption that an object is an abstraction that encapsulates a set of services.

Contract Modeling Creating a model (interface) with the goal of specifying an interface. The name is derived from the idea that an object definition is a contract that both users and implementers must honor.

Marketplace Modeling A way of layering contract models to achieve application interoperability.

©1996 BY BENCHMARKING PARTNERS, INC.

Acronym Guide

API	application programming interface
COM	Component Object Model
CORBA	Common Object Request Broker Architecture
DCOM	Distributed Common Object Model
IDL	interface definition language
OAG	Open Applications Group
OAGIS	Open Applications Group Integration Specification
ODBC	open database connectivity
OMG	Object Management Group

©1996 BY BENCHMARKING PARTNERS, INC.