Integrating Applications with Digital's Framework-based Environment

James R. Kirkley
William G. Nichols

Digital has developed the Framework-based Environment to address the integration and interoperability needs of manufacturing and other business systems. FBE consists of a method for integrating existing applications, frameworks of industry models, and tools that use Digital's CORBA-compliant ObjectBroker integration software to manage the exchange of information between cooperating servers on the network. Using these products, Digital Consulting and its partner systems integrators provide FBE application integration services to large organizations.

The increasing quality and cost-effectiveness of computer application software has revolutionized the way organizations share and manage their information. Rather than develop custom information systems with their internal programming staffs, many businesses now purchase software available in standard "off-the-shelf" packages. A well-chosen standard package can save development time and cost. Before it can be useful, however, it must be integrated with other new software and with the mature (legacy) applications that hold current business data and processes.

Application integration can be a substantial effort. If business changes are not anticipated during the planning phase, an integrated system can be inflexible. The existing applications, both legacy and new, rarely meet current requirements. An ad hoc integration that starts with the existing applications' interfaces will seldom be flexible in ways that accommodate future business changes without widespread program changes.

An integration derived from a clear model of current and expected business processes provides a basis for growth and flexible change. Digital has developed the Framework-based Environment (FBE), consisting of reference models, methodologies, and a toolkit. Together, these products provide flexible systems integration.

In this paper, we provide a brief overview of FBE and characterize the projects that can benefit from using it. We describe flexible application integration and the benefits of model-driven integration. Finally, we discuss our experience using FBE.

Overview of the Framework-based Environment

FBE consists of the following components.

- MethodF is an object-oriented methodology based on two systems integration methodologies recognized in the industry: Jacobson's use case analysis and Rumbaugh's Object Modeling Technique[1,2,3,4] These methodologies are explained in the section Model-driven Integration with FBE.

- ObjectPlus is a modeling tool from Protosoft, Inc. that has been tailored for MethodF with an FBE-specific code generator. In addition to the methodologies described above, the tool has extensions that provide the ability to create an implementation model. The implementation model describes how objects are distributed among the various applications.
- ObjectBroker, Digital's object-oriented integration software product, is compliant with the Common Object Request Broker Architecture (CORBA) specification from the Object Management Group (OMG).[5,6]
- A suite of supporting libraries and tools includes reference models and associated code libraries that have been abstracted from previous projects and made available for reuse. The reference models and associated code libraries are organized into frameworks of industry-oriented business objects, as given in Table 1.

  The tools include two important components: (1) The FBE Design Center is an extensible workbench architecture that supports the analysis, design, and implementation of CORBA-based distributed object systems. (2) The FBE Adapter Development System, which fits into the FBE Design Center, automatically generates CORBA- or ObjectBroker-compliant code and the necessary files to compile and link the code into platform-specific executables.

Integration Projects Appropriate for FBE

Any integration project automates previously manual processes involving existing applications. FBE and its flexible approach to systems integration allow a business to replace or add component applications efficiently as business conditions change.

FBE provides the most benefits when many different kinds of well-defined business transactions occur between a mixture of commercial and custom applications. Not all projects can benefit fromFBE or its style of development. For example, if the primary task is to integrate data sources for decision support, a database integrator or a data warehouse may solve the problem quickly. If a company is not trying to gain an advantage by automating accounting more cheaply or completely than its competition, an off-the-shelf accounting package may be the right choice. At the other extreme, if the task to be automated is completely new, there may be no appropriate packages available, even as components of an integrated solution. New development would also be preferable if high-performance or real-time operation were more important than the flexibility to plug in existing, unmodified applications.

As an example of an appropriate FBE integration, consider a manufacturing operation automating a manual procedure that collects orders from an order processing system, schedules production runs, and passes the schedule to the manufacturing floor. In this example, the company wants to obtain a competitive advantage by dynamically rescheduling production based on new customer orders, at once reducing inventory costs, and improving delivery performance. This is more than a decision support system: the integration requires that applications interact with each other. Although finding a turnkey package that can operate the entire factory is unlikely, factory scheduling applications are readily available. Buying one would be more cost-effective than writing one in-house. The project would then need to integrate the legacy order processing system with the newly purchased scheduling application. The order processing system is too important to the company to risk modifying it significantly at the same time as introducing new automation.

After the integration project has been completed, though, the order processing system might be made more cost-effective by moving its function from a mainframe application developed in-house to a standard client-server product. Perhaps business conditions will have changed and the order processing system needs to be augmented so customers can submit orders directly by electronic data interchange (EDI). The project manager might decide to purchase an EDI processor to augment or replace the existing order processing system.

Later, after the manual processes have been automated on the factory floor, another project could extend the integration to send the schedule directly

Table 1
Frameworks of Industry-oriented Business Objects

Base Business Models	Manufacturing Business Models	Industry Business Models
Activity management	Order management	Semiconductor
Production management	Schedule management	Oil and gas
Resource management	Product management	Pharmaceutical
	Process management	Batch process
	Quality management	Banking and finance

to factory cell controllers. Then, if a more efficient scheduling package becomes available, it could be substituted for the older one. The modular design of FBE would minimize the programming changes required for this substitution and give the organization the flexibility to use the most cost-effective solutions.

Model-driven Integration with FBE

An integration project needs a clear process and a means to avoid being biased by the assumptions built into its component applications. We use object modeling to plan and document an integrated system in a uniform manner. The abstraction inherent in object modeling hides detail. This makes the model meaningful and allows modeler and client alike to ensure that the model matches the intended business processes. The abstraction also helps to separate the interface from the implementation. The interface describes *what* is to be done; the implementation describes *how.* The *what* of a business process changes comparatively little over time: a factory takes orders and schedules production runs, a stockbroker trades stock, a mail-order business ships packages. The *how* changes dramatically from year to year.

In the following sections, we trace the steps of a typical systems integration project as conducted by Digital Consulting or by Digital's partner systems integrators. We show how a modeler might use the FBE method, tools, and frameworks to provide application integration services.

Object Modeling

Before we start object modeling, we ensure that a business process model, or its equivalent, is completed. Sometimes a business process model results from a formal business process reengineering. More often it comes from a less formal understanding of existing processes and required changes. In both cases, the modeler will cooperate closely with someone who understands the process well. As always, the better we understand our goals, the more likely we are to achieve them.

With this knowledge, we can start FBE's object-oriented analysis and design process, known as MethodF. MethodF begins with Jacobson's use case analysis method. A use case traces a chain of events initiated by a single person (or other entity), acting in a single role, as he, she, or it works through some task. For example, we might trace what happens when a customer calls an order desk through the clerk's responses, catalog checks, inventory checks, order placement, picking list generation, and finally, package shipment. As we do this, we note all the objects and the uses that the actors make of them. Then we follow another use case. Perhaps this time the customer asks for a product that is out of stock. We follow the discussions about back-ordering and price guarantees that will make our business attractive to this customer. After analyzing many use cases, we have a list of *business analysis objects* (objects that describe requirements in business terms) and a list of the functions and attributes of each object.

We then compare the analysis objects with the *business design objects* in FBE's reference model library. Here, we may well find similar objects that use different names and detailed constructs to describe the same functions and attributes. The next step in MethodF is to merge these design objects into the model. By using objects from the reference library, we take advantage of previous modeling experience built into the reference models and prepare to reuse code associated with the reference models as well.

We use the ObjectPlus modeling tool to capture use cases in diagrams according to Jacobson's conventions. We prefer the Rumbaugh Object Modeling Technique (OMT) notation, however, for describing the business objects. OMT diagrams, with FBE extensions, define objects and the interfaces between them in enough detail that a tool can use them to generate interface definitions that can be compiled. The ObjectPlus tool also captures OMT diagrams.

A direct connection exists from the use case models, through the business models, to the design models, and to the code. We use the term *model-driven* to describe the FBE approach, because necessary changes are first made to the models and new code is then generated from the models.

Generating Interface Code

Once we have completed the design objects, we use FBE tools that work with the ObjectPlus modeling tool to generate CORBA Interface Definition Language (IDL) from the design object definitions.[6] We chose CORBA because it is an emerging industry standard designed to build distributed object-oriented systems that include existing non–object-oriented applications. A CORBA implementation, such as Digital's ObjectBroker product, generates interface stub routines that marshal data to be sent to an object, whether the object is on the same computer or across a network. For example, the stubs convert integers sent from big-endian to little-endian computers. A CORBA implementation also provides an object request broker: a run-time library that routes requests to objects in a distributed system. This allows applications running on different systems to communicate without the need for applications to know which systems will be involved.

We use the IDL interface definitions to guide programmers as they develop *adapters* between this object interface and the existing application's interface. For example, an existing program might take its

input as a formatted file and deliver its output in another type of file. Since the rest of the integration should not know about these files or their formats, we write an adapter that translates between these files and the methods and attributes of the objects defined in our model. Perhaps an alternative application uses a remote procedure call for I/O instead of the files our existing application uses. When we replace the existing application, we write new adapters using the same object interfaces. As a result, the rest of the integration needs no changes. Writing these adapters is not necessarily easy; application integration requires substantial effort, whether the integrator uses FBE or not. By restricting the changes to a single module, FBE minimizes the development and testing effort required to replace component applications.

We usually write the adapters in C, rather than C++ or a pure object-oriented language, because much of their interaction is with the applications being adapted. The existing applications were seldom built with object-oriented principles. In many cases, useful tools such as database translation programs and "screen scrapers" are available to communicate with applications that expect terminal I/O. These tools also were seldom built for object-oriented languages.

In some cases, an adapter needs to be so large that it is a small application in itself. In these cases, we might use an object-oriented language for the bulk of the code. A factory scheduler might generate production tasks based on a customer order, but the cell controllers in the factory might expect only a single task for each type of part produced. The adapter needs to combine the tasks for a given part type from several orders before it sends a message to the cell controller. As the cell controller reports progress on each task, the adapter allocates completed parts to the original customer orders. The cell controller simply makes parts, the factory scheduler simply fulfills orders, and the adapter bridges the gap between them.

Reference Models

As we gain experience working with integrators, we abstract and merge the models they build into reference models for the various application domains, such as discrete manufacturing, process manufacturing, and financial services. We collect and tailor the reference models to comply with accepted industry standards such as ISO STEP in the manufacturing domain and ISA SP88 in the process industry domain.[7,8] These reference models allow FBE modelers to build on previous experience. Even if they cannot use the reference model in its entirety, they can use it as a guide to save time and to check their own model for completeness. We also collect the adapters for frequently integrated applications into a library. Later, when we reuse a reference model, we will have corresponding adapters that can also be reused, usually after modification. It is important to note that anyone—Digital, the systems integrators (Digital's partners), and, most importantly, the customer—can build their own reference models.

From Applications to Objects: Experience Gained

Design always involves trade-offs between competing requirements. The trade-offs in an integration project are somewhat different from those in a new development project: an integration project must take existing applications into account while trying to implement a business model faithfully.

In this section, we discuss trade-offs due to the change from a functional view to an object view, then explore three familiar design topics from the point of view of an FBE integration project: top-down versus bottom-up design, improving reliability, and improving performance.

Overcoming the Legacy of Functional Decomposition

The challenge of object-oriented application integration is to make application programs, which are designed around individual business *functions,* support the unified business *object* model.

Figure 1 illustrates a sample mapping of business objects to application functions. It shows the logical objects of customer, product, and shipment, with their data structures and methods mapped to the several different application functions of transportation, warehousing, and billing. As the integration project maps business objects to application functions, it must

- Establish routings of requests for individual attributes or operations of an object to the applications that contain them
- Provide mechanisms to maintain consistency when multiple applications require the same data

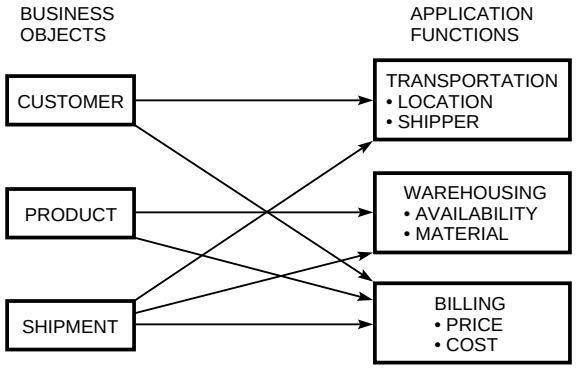


Figure 1
Sample Mapping of Business Objects to Application Functions

Split Instances When we develop the business object model, we may discover that a single logical object may be hosted (its underlying data structures and methods implemented) by more than one physical application. For example, a product object's *price* attribute is hosted by a billing application, and its *availability* attribute is hosted by a warehousing application. When we integrate these applications according to a business object model, we achieve a single logical object whose data and methods are stored in different physical applications and often in different locations. This is called a *split instance.*

When a client application requests the product's availability, the object request broker sends the request to the warehousing application and forwards a request for the price to the billing application. The requester neither knows nor cares where the information is held.

The notion of the split instance is a central principle of FBE. It allows us to model the business logically and independently of the way applications may implement business functions. The split instance is not without its problems: Many times the same information is stored in more than one application. In the above example, it is likely that both the manufacturing and the billing application maintain the product *name* attribute. Many other attributes are potentially duplicated as well. When an attribute of a type exists in two or more applications, the designer is faced with two questions:

1. When a *get attribute* operation is requested, to which application should it be delivered?
2. When a *set attribute* operation is requested, is it necessary to update only one or more than one application's data?

We cannot answer these questions in a general way, but we can highlight some points to keep in mind when addressing them.

- Get attribute. Can one application be considered the primary source for data about an object? Before any integration was in place, legacy systems provided a formal or informal process that updated secondary information sources from a primary source. The requirements statement is a good reference here. The designer should discuss this with the business domain experts to understand the way data is maintained and distributed. The primary application is the best source for such data. As a backup, secondary applications could serve as sources for the information. The designer should consider the effect of stale information on the operation of the business.
- Set attribute. When attributes are *set,* should all applications be updated simultaneously? Usually a category of infrequently changed "reference data" is accessible. The reference data is more often added to than changed. Changes to this kind of data essentially ripple through the company. Sometimes it is the slow communication of these changes throughout the organization that drives the requirements for integration (the push-pull phenomenon).

When we must guarantee simultaneous changes to data on multiple heterogeneous computing platforms or between applications that hide their data, we would prefer a two-phase commit transaction between dissimilar databases. Unfortunately, nothing is commercially available today (June 1995) that works on an arbitrary combination of databases and applications. Several products support a limited set of third-party databases and applications. If these products cannot address the need, and our applications require multiple application transactions, we may have to write the two-phase commit code.

As an alternative, we may be able to use a workflow to manage the update of several applications. An operation can be defined that is implemented as a workflow script. The workflow script can, in turn, perform the update (through additional method invocations) on the data stored in a number of different applications. This is probably closer to the customer's method and would be easily automated. A workflow capable of doing the update must have the capability of compensating for failure to update all applications. A workflow update is different from two-phase commit, because the data in the applications may be inconsistent for a brief time.

To our knowledge, Digital's ObjectBroker integration software is currently the only CORBA implementation that is able to route requests for a single object to multiple servers.

Bypassing Legacy Applications Sometimes it is tempting to bypass a legacy application and access its database directly from an adapter. The application may have a particularly difficult interface, or the required function and data may be a small part of a monolith. For simple applications, bypassing may be appropriate, but for most we must either use the application through its intended interface or replace it entirely.

The use of a legacy system to change data or perform a function can produce unwanted side effects that are not appropriate in the context of the integrated system. For example, most legacy applications maintain the referential integrity of their data through code. Invoking the database directly to add, update, or delete data risks violating this integrity.

Bypassing the application is also dangerous because changes may occur when the application is revised. Typically, application developers feel free to change the underlying data structures as long as the functionality at the user interface or formal program interface is maintained.

Top-down versus Bottom-up Design

Tension always exists between the goals of top-down and bottom-up designs. The FBE emphasizes top-down modeling; it starts with the analysis of use cases and then defines business objects independently of any existing applications. This keeps the design focused on the business problem and enhances the flexibility of our integration. We find that the most common modeling error is to accept an existing application's "myopic world view" without considering the overall system's needs. Usually, existing applications are a poor source for business object models, since many no longer represent desired business processes.

If we are not conscious of bottom-up demands on our design, however, we can design a system that requires needlessly large, complex, or slow adapters between the existing applications and our ideal model. Though we have no easy guidelines for balancing the top-down and bottom-up demands, some issues are encountered repeatedly.

The problem of partial implementations provides a simple example of this balancing requirement. Projects that use top-down modeling to derive their object models sometimes encounter a dilemma: attributes and operations appear in the model that no application in the network can implement. It is reasonable, for example, for the object model of a factory floor conveyor to define a stop operation, but the device control software installed in the factory may not provide an equivalent function.

When implementers cannot support a model, they have two choices:

1. Modify the model to reflect the capabilities of the environment.
2. Implement only the part of the model that is feasible.

The first option appears to be the easier choice, but it limits the reusability of models and diminishes the effectiveness of the top-down approach. A top-down model of the conveyor should capture the business users' expectations; implementations may or may not meet these expectations. A partial implementation simply returns an error whenever a user accesses an attribute or invokes an operation that is not supported.

The partial implementation of a conveyor can still be substituted for a complete one, though the partial one always fails when a user sends a stop request. The system must be prepared to receive an error response from an operation invocation at any time; other errors could occur during the stop operation's processing, even if the implementation were complete.

A partial implementation opens the way for subsequent versions of the software to support the feature. It provides a placeholder for an attribute or an operation and preserves the integrity of the object's specification.

Improving Reliability

Finding bugs in an integrated system is often difficult. Even if we assume that the component applications work perfectly, bugs can arise from mismatches between the components. This commonly comes about because of inconsistent business rules between applications: what is allowed in one application may be illegal in another.

An adapter in an integrated system must be a firewall; that is, it must limit the spread of errors and misunderstandings from its application. We code pre- and post-condition checks around calls to component applications. This is helpful if we code for the right conditions and leave the checks in the production code. The use case analysis and business object descriptions sometimes suggest conditions to test, but this process is informal. We find that we need more run-time checks in adapter code than in individual applications.

We also need a way to isolate a suspect application from the integrated system so we can see how the integrated system behaves without it. FBE's Adapter Development System can generate simple stubs from an object's OMG IDL. The tool generates a client stub that makes appropriate requests and a server stub that echoes its input. The stubs are simple enough to be checked at a desktop device to ensure that they work as expected. The stubs are also useful as templates for starting new adapters.

Improving Performance

Without planning and careful monitoring, a large system of dissimilar applications can be slower than the performance of the component applications would suggest. We have used standard approaches to improve and monitor performance. It is worth noting here how these approaches influence FBE design and development.

Performance Requirements in Large Systems There is often a trade-off between performance and flexibility. Our integrated system would be ideally flexible if it made separate calls through an adapter to a component application for every datum in every different circumstance. We could change storage and behavior almost with abandon. On the other hand, if each adapter were an entire rewrite of its underlying application, we could, in principle, store and manipulate each datum in the most efficient way for all accesses.

Although FBE is designed for systems that require flexibility at the cost of some performance degradation, we must be careful to deliver satisfactory performance. In the following subsections, we discuss the trade-offs in caching and object granularity.

Caching Applications frequently generate large quantities of output in response to a command, rather than the fine-grained results that are appropriate to object-oriented requests. It is often appropriate for an adapter to return only a small part of the data it receives from an application interaction and cache the rest for future requests. Applications that produce data in batches typically do not modify their state for long intervals, so the cached values remain valid long enough to be useful. Of course, there must be a means to invalidate the cache. In some cases a timer will suffice; in other cases an event, such as a new batch run, must be extended to invalidate the cache.

Adapter caches greatly improve performance and can give the adapter developer the freedom to organize and present the data in a form appropriate to the object model.

Object Granularity Designing objects that work well in a distributed system is important to ensure flexibility. Parts of a distributed system frequently move from one computer to another. We should not expect our objects or their underlying component applications to remain in one particular place.

In a pure object-oriented system, for example the Smalltalk language, everything is an object. In distributed systems, operations on objects potentially involve interaction across a network and incur network overhead. Therefore, it is not practical for everything to be an object. Some business objects will be implemented as CORBA objects (those that have object references) and other business objects will be implemented as user-defined types (passed by value). This defines the *granularity* of the object model. The decision to implement a business object as a CORBA object or as a user-defined type involves balancing flexibility with system performance.

There are no hard and fast rules that determine the most appropriate granularity for an object model. Decisions need to be based on users' interactions with the system and on the way applications use the objects they share or exchange with each other. Several matters should be taken into account when determining the model's granularity.

As an illustration, let us consider a client application that needs to display a collection of customer names in a list box. The client sends a request for these names to an object instance called CustomerList; the client and object happen to be on different computers.

In Case 1, the customer is a user-defined type represented as a C structure: it is passed by value and has no object reference. Customer attributes are stored in a CORBA-defined structure that the client code must access directly. In this case, the display of customer names may be accomplished in a single request, e.g., getCustomerNames(aCustomerList). All customer names would be passed by value. Figure 2 depicts this scenario.

In Case 2, the customer is a true object: it has an object reference and a set of attributes. The client calls the server separately for each attribute; thus the client is less dependent on the server's storage structure or any changes to that structure as it is modified in the future. In this case, a sequence of customer object references would be passed, e.g., getCustomers(aCustomerList). The client application then must request getName(aCustomer) for every customer object in the sequence. (See Figure 3.)

Clearly, the first case is more efficient in terms of network utilization; only one request is required. The second case requires $1 + n$ requests, where n is the number of customers. The first case is also more efficient at the server. Case 1 requires one database query to construct the name list, whereas Case 2 requires a separate database query for each customer.

At first glance, Case 1 would appear to be the easy winner in terms of efficiency and effective utilization of the server. This outcome, however, is not always true. Let us assume that the client application allows the user to choose from the list of customers and then displays attributes *address* and *accountStatus* for the selected customer. Here, we are faced with a choice between performance and flexibility:

1. The client could make another request that would return *all* information about a customer in a structure. Then the client application could sort through this information and display the required data. The performance is good: one request and database query provided all the data the client could want. Unless the volume of data is very large, sending the data in one message yields better

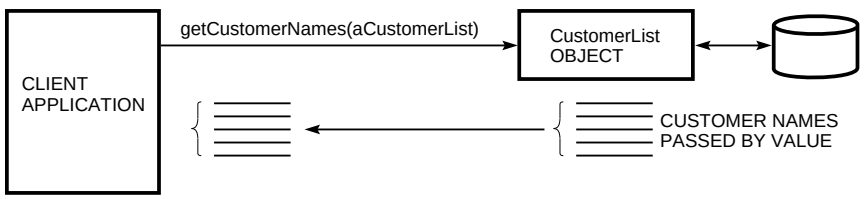


Figure 2
Case 1: User-defined Type

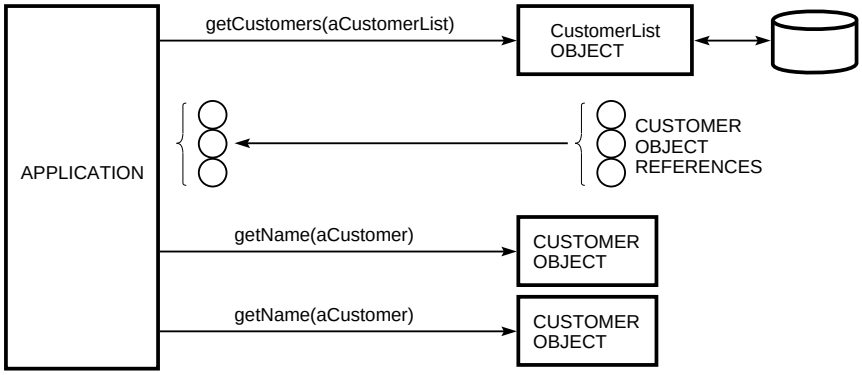


Figure 3
Case 2: True Object

performance than sending multiple messages for a subset of the data. On the other hand, this approach is inflexible: if the server changes the structure it uses to represent this data, all client software that reads the structure must change as well.

2. The client could make separate requests for each field. If the server returns an opaque object reference along with each customer's name, then the client can send a request asking for the specific fields it needs. The performance is worse than in Case 1, of course, because of the extra network traffic and message parsing. However, this approach is flexible. Since the client never looks in the object reference (it is opaque), we preserve the server's flexibility to use any data needed to retrieve the appropriate record. As long as the server continues to support the fields the client requires, the server finds them in its own database no matter how the storage structures have changed.

To ensure that the system provides the maximum flexibility, the designer should consider the following guidelines.

- Start with a fine-grained approach for modeling.
- Implement the approach using fine-grained methods.
- Change to a coarser grain if performance is an issue.

Summary and Future Directions

Developing integrated applications is not always a straightforward process. The applications being integrated are seldom an exact fit to their assigned roles in an integrated system. If they were, we would probably be able to purchase the integration from one or more of the vendors who had engineered the fit.

Integrated systems built with FBE are clearly documented with Jacobson use case diagrams, Rumbaugh OMT object diagrams, and OMG IDL. The existing applications are used indirectly through object interfaces and adapters, so the rest of the system can address them as if they were the ideal business objects modeled in the OMT diagrams. We call them business objects to emphasize their distinction from objects defined or implied by the existing applications.

The adapters are constrained by the interfaces that FBE generates automatically from the business object representations, so they do not stray from the models that document their behavior. Adapters are not always easy to write; they can be quite difficult, depending on the existing application's fit with its intended use. By restricting this awkward code to object adapters, we keep the overall integration modular. Thus we give an organization the flexibility to use the most cost-effective systems as business conditions change. We build on our experience by collecting reference models that help us to reuse the best models and adapters.

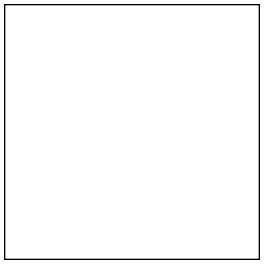
FBE continues to evolve rapidly, with improvements in the reference models, the tools, and the support for adapter writers. For example, developers have asked for better integration between the Jacobson and Rumbaugh models, between the modeling tools and the code generation tools, and for reliable queuing and workflow as well as CORBA communication between objects. In response to these requests, we now provide better integration between the analysis, design, and implementation portions of the FBE life cycle as well as code generation for trace messages and support for management and debugging of the runtime system. We would like to organize the reference libraries into pairs of object models and corresponding modules (applications and adapters) that can be assembled to build integrated applications, thus creating truly reusable business components.

We will be pursuing these and other improvements as our experience grows with integrated, distributed applications.

References

1. *The Framework Based Environment: MethodF, Version 3.0, FBE Engineering* (Maynard, Mass.: Digital Equipment Corporation, Order No. AA-QC50A-TH, 1994).
2. I. Jacobson et al., *Object-Oriented Software Engineering: A Use Case Driven Approach, 4th ed.* (Wokingham, England: Addison-Wesley Inc., 1992).
3. I. Jacobson et al., *The Object Advantage, Business Process Reengineering with Object Technology, 1st ed.* (Wokingham, England: Addison-Wesley Inc., 1995).
4. J. Rumbaugh et al., *Object-Oriented Modeling and Design* (Englewood Cliffs, N.J.: Prentice-Hall, Inc., 1991).
5. *ObjectBroker: Overview and Glossary, Version 1.0* (Maynard, Mass.: Digital Equipment Corporation, Order No. AA-Q9KJA-TK, 1994).
6. *The Common Object Request Broker: Architecture and Specification, Revision 1.2* (Framingham, Mass.: Object Management Group, Order No. 93.12.43, 1993).
7. *Industrial Automation Systems and Integration—Product Data Representation and Exchange—Part 44: Integrated Resources: Product Structure Configuration, ISO 10303-44, WG3 N127* (Geneva: International Organization for Standardization, 1992).
8. *Batch Control Part 1: Models and Terminology, Draft 12: 1994* (Research Triangle Park, N.C.: Instrument Society for Measurement and Control, Order No. ISA-dS88.01, 1994).

Biographies



James R. Kirkley III
Jim Kirkley has been with Digital for 16 years. For the last six years, he has been involved in the development of object-oriented architectures for business application integration. A software consulting engineer, Jim is the technical director for the Applied Objects Group, which is currently focused on the development of tools and methodologies for the integration of business systems. He is the principal author of the methodology used by Digital Consulting to deliver consulting and practice systems integration using CORBA-compliant middleware. He received a B.S. in electrical engineering from Colorado State University in 1971 and an M.S. in computer science from Colorado University in 1974.

William G. Nichols
As a consultant engineer with Digital, Wick Nichols was part of a team that reviewed the Framework-based Environment and provided a report suggesting several improvements. His familiarity with related networking products, particularly DCE, enabled Wick to participate in the delivery of several FBE projects to customers. During his 15 years with Digital, Wick contributed to several projects, including the development of distributed file services. He also served as project leader of a group that developed the DECnet-10 system and as project leader and supervisor for the DECnet-20 product. He received an A.B. from Harvard University in 1973.