Using JavaBeans components as accessors to Enterprise JavaBeans components

by A. Tost
V. M. Johnson

In this paper we describe how the use of JavaBeans™ components together with Enterprise JavaBeans™ (EJB) components can help to develop flexible, mission-critical applications. We show how an application can be structured into three conceptual tiers, and how the use of JavaBeans components on the middle tier can help develop applications that can run in different client/server environments (i.e., thin or thick clients). This eases the separation of server-side and client-side application development, each focusing on different problem domains. We also show that the additional level of indirection provided by using JavaBeans components as accessors to server-side business objects helps users of IBM SanFrancisco™ components to isolate their client-side application development from upcoming changes in the underlying back-end technology (i.e., the migration from today's SanFrancisco infrastructure toward future EJB-based releases of SanFrancisco).

Until fairly recently, the Java** programming language has been used primarily by client application programmers. It was used in "applets," little programs that can be downloaded from the Internet and run in a browser, or in stand-alone applications, for example, using the Swing class library for their graphical user interfaces. The appearance of the JavaBeans** standard helped to develop and promote the use of visual as well as nonvisual components written in the Java language.

Today more and more server-side use of the Java language can be found. Previously, Common Gateway Interface (CGI) scripts were usually used to let a browser make a call to a server-side program. CGI scripts are programs, called by the Web server, that provide dynamic information to be displayed in the browser. Now this approach is being replaced by the use of "servlets." Servlets are Java programs that are called by the Web server just like a CGI script, but they do not require a new process to be started by each new request. Instead, a new thread is started for each request, and each servlet instance can be shared by many end users.

In addition to its use for developing servlets, the Java language is being used for the development of entire suites of business applications. In the SanFrancisco* product, IBM delivers application business components written entirely in the Java language. These components allow the creation of business applications by building on existing Java classes.

The SanFrancisco components proved that the use of Java server-side components can considerably increase productivity in developing business applications. In the same timeframe, a standard was developed by Sun Microsystems, IBM, and other companies to formalize Java components on the server, as well as the environment for these components. This standard is called Enterprise JavaBeans** (EJB). The IBM SanFrancisco product will support the EJB standard in a future release.[1]

This will make it possible to develop all parts of an application in the Java language. The use of standard interfaces will further the reuse of existing business components. An Enterprise JavaBeans component developed on one platform, with one particular transaction and persistence environment, can be used without change on another platform, using a different transaction and persistence environment.

Application architectures: Three logical tiers

Most applications today are developed using the Model-View-Controller (MVC) pattern.[2] This pattern, developed by Trygve Reenskaug for the Smalltalk programming environment, separates application data (the model) from its presentation to the user (the view). Moreover, the application logic, which defines the flow of an application based on user interaction (the controller), is further separated. An application is built by associating all three—model, view, and controller—with one another.

SanFrancisco defines a set of object-oriented interfaces to an infrastructure that can be used to build e-business applications.

This allows an application to be conceptually split into three tiers. The first tier contains the view, i.e., the presentation information. In traditional applications, the first tier is built using a library of graphical user interface components. An example is the Swing class library that is provided as part of the Java Foundation classes. The library includes components such as frames, panels, buttons, and input fields that are used to build the presentation interfaces (screens) for a user of the application.

In thin-client environments, where no application software is installed and the client communicates with the server through the Internet or an intranet, the first tier is typically represented by a browser. The browser displays Web documents that provide the application interface. These documents are typically written using HyperText Markup Language (HTML) or Dynamic HyperText Markup Language (DHMTL). HTML and DHTML are standards that have been adopted to facilitate the electronic exchange and display of simple documents.

The second tier contains the controller. The controller handles the reaction of the application to user events and provides data to the view. In a traditional, thick-client application, this will be done by running the view and the controller in the same process on one machine. The view and the controller communicate using standard Java event mechanisms. In a thin-client environment, a "view server" might be required to handle the data traffic between the view and the controller, since they are running in different processes on different machines. For example, a servlet can play the role of a view server.

The third tier contains the model, i.e., the application data. The model knows nothing about how the data are presented to a user, and in many cases, it is designed and developed independently from any actual application. In a business application environment, the model describes the business domain model, which contains not only business data, but also the processes that control the data.

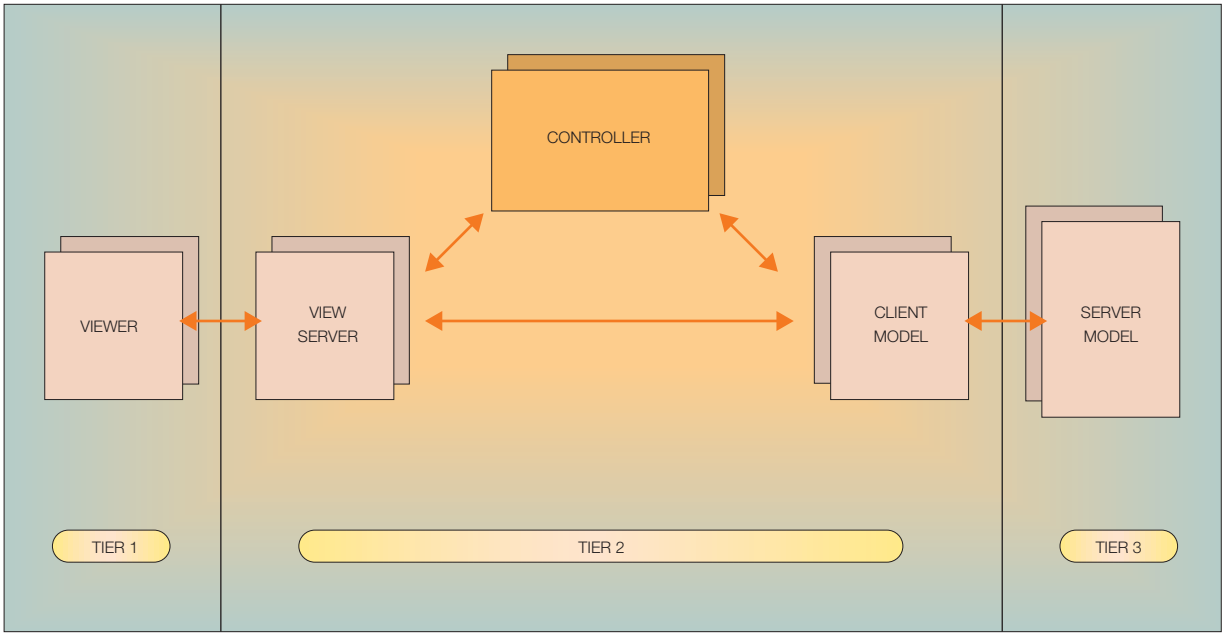
For example, the SanFrancisco components contain business objects as well as business processes that are modeled for specific business domains, like general ledger, or warehouse and order management. These domain-specific objects and processes are fairly stable, while the client application requirements change very often. Also, different client applications may share the same server-side business object without knowing about each other.

In order to simplify the development of an application, a client-side model may be introduced that represents the model to the view and the controller, and encapsulates the access to the server-side model. Figure 1 shows all three logical tiers of the application. Once the three tiers have been identified and designed, the actual application can be implemented for a particular target environment. In other words, all three tiers can run on the same machine, or they can be split up over several separate machines and processes.

The server model: Enterprise JavaBeans

The Enterprise JavaBeans (EJB) standard defines the server model. It specifies the interface that each business component must implement. Each EJB com-

Figure 1 Three logical tiers



ponent exists in a server environment called the "container." The container lets a client access an EJB component in an implementation-neutral way, and it handles persistence and transaction mechanisms, etc.

Business components with persistent state data are typically represented by "Entity" beans. Each Entity bean comes with a "home" interface, which allows a client to create, delete, and access the bean. An Entity bean's home is found by using a naming service. The transaction model that is used for an Entity bean is not defined in the bean's implementation, but in the form of a "deployment descriptor." The deployment descriptor is used by the bean container to determine, for example, if a method on a bean can be called only within an existing transaction context.

The client model: JavaBeans

As explained earlier, access to the server model should be encapsulated in form of a client model. "Client" in this context stands for the second logical tier, not necessarily the end-user client. This allows greater flexibility in two ways: the server model can be developed without a particular client environment in mind, and the client (say, first and second tier) can be developed without being dependent on a specific server infrastructure.

Moreover, the client model can hide many of the complexities of the server programming model, making it easier to develop an application that accesses, for example, Enterprise JavaBeans components. The EJB programming model requires the use of naming interfaces to find the right objects, transaction interfaces to handle transactional contexts, "home" interfaces to create, delete, and retrieve EJB objects, and many more interfaces. A client model can use all of these while providing a much simpler interface to its clients.

Since many of the interactions with the server model are done through the use of standard (EJB) interfaces, a client model can easily be generated. Each server-side EJB component can be handled by a client-side object that redirects client calls to the right server object, retrieves the EJB components' state, and so forth.

Thus, access to the server model is delegated to the client model. Given the logical three-tier architecture outlined above, the client model will always be accessed locally; no remote call support is needed.

In order to reach maximum flexibility for accessing the client model from different variations of view-controller pairs, the client model should support a standard for its access. The JavaBeans specification defines such a standard. So, for maximum flexibility the client model should always be designed and implemented as a JavaBeans component. This allows it to be used in a component-based development environment like IBM VisualAge* for Java, where JavaBeans components can be visually programmed. It can also be used in environments where the actual implementation of the client model is not known at compile time. A JavaBeans component can be accessed at run time by the "introspection" mechanism, which retrieves information about the interfaces supported by a JavaBeans component.

Commands

Most commercial applications not only access and change properties of business objects; they also run business processes that span multiple business objects. In the IBM SanFrancisco product, these processes are typically encapsulated in "commands." A command is a class that represents a business process. A SanFrancisco command object is instantiated locally by the client that uses it. It then gets a business object as its target. Upon execution, the command will be transferred to its target's address space (i.e., the server), executed there, and returned to the client with the result data.

This behavior fits very well into the architecture we have described. The client model will instantiate the command, send it to the server for execution, and provide the resulting data to the client application in a standardized way.

The SanFrancisco beans

The SanFrancisco product provides "SanFrancisco beans" that act as accessors to the server-side business objects. The most commonly used business objects are shipped as ready-to-use JavaBeans components that can be imported into an interactive development environment like VisualAge for Java. Moreover, a "bean wizard" is provided that lets a developer generate a JavaBeans component for any existing business class.

Access to server-side business object data can be optimized by caching some of the data in the JavaBeans component. If and how this is done is transparent to the client of the SanFrancisco bean, since there is no impact on the client interface that the bean provides.

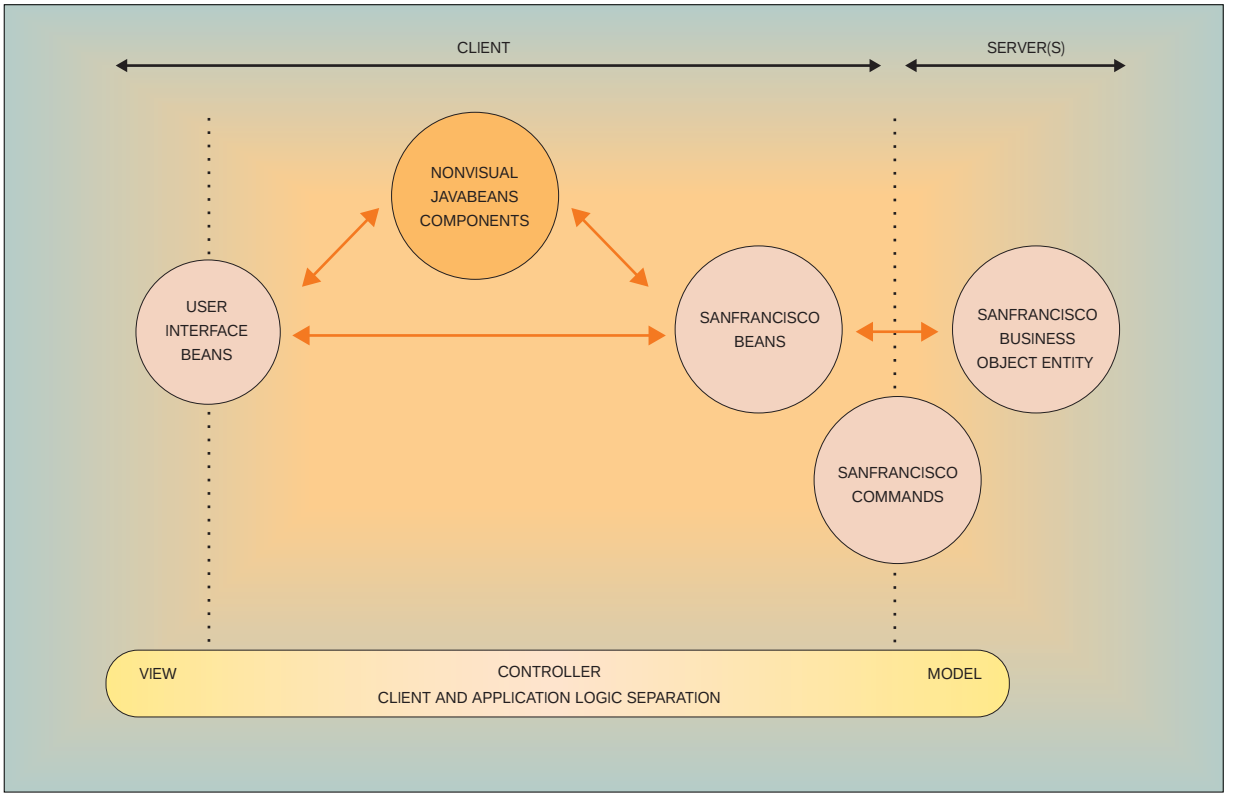
Most business objects are not stand-alone objects; they are part of a collection of business objects. Clients gain access to a target business object either by running a query or by accessing the entire collection and iterating over its elements until the target object is found. SanFrancisco business objects are often accessed through a "controller." A controller is associated with a "company," which represents the organizational hierarchy of the enterprise. Each controller handles those business objects that are part of the same company. One controller is responsible for one particular type of business object. The controller's interface lets a client retrieve, add, or delete an individual business object or an entire collection of business objects. It also provides a query interface.

Business objects that are not owned by a controller may be stand-alone objects, or they may be owned by another business object, or they may be contained in yet another collection of business objects. Thus objects can be accessed in different ways. A SanFrancisco bean hides this complexity from the client by providing different "modes" under which a bean can operate. Thus, a business object can be accessed through a controller or directly from its owning business object without any impact on the client interface of the bean.

Moreover, each bean can represent an individual business object as well as an entire collection of business objects. This is especially useful in the following common scenario: A client retrieves a collection of business objects and displays them on the screen. The user selects the business object to work with, and this object is displayed in a new window, where the user can make changes to it. This entire scenario can be handled by one SanFrancisco bean. The bean wizard even generates visual beans to display the collection and an individual object. Thus, the scenario described here can be implemented very quickly.

SanFrancisco beans also encapsulate the transaction handling that is necessary to access a server-side business object. Transactions are started and committed when necessary, based on a transaction policy that can be set on each bean. For example, each change to a property of the business object can be sent to the server object inside its own transaction. Or, all changes to an object may be collected in the

Figure 2 A thick-client approach



bean until the user's work is completed, and the business object then updated in one transaction.

Since the SanFrancisco beans hide the access to the server-side business object, the client is not dependent on how this is actually implemented. In other words, the EJB-based release of the SanFrancisco product will use EJB interfaces to access the server object instead of SanFrancisco-specific interfaces. All of the characteristics of the server-side business object, e.g., the fact that it may be contained in a controller, will remain, regardless of the underlying infrastructure.

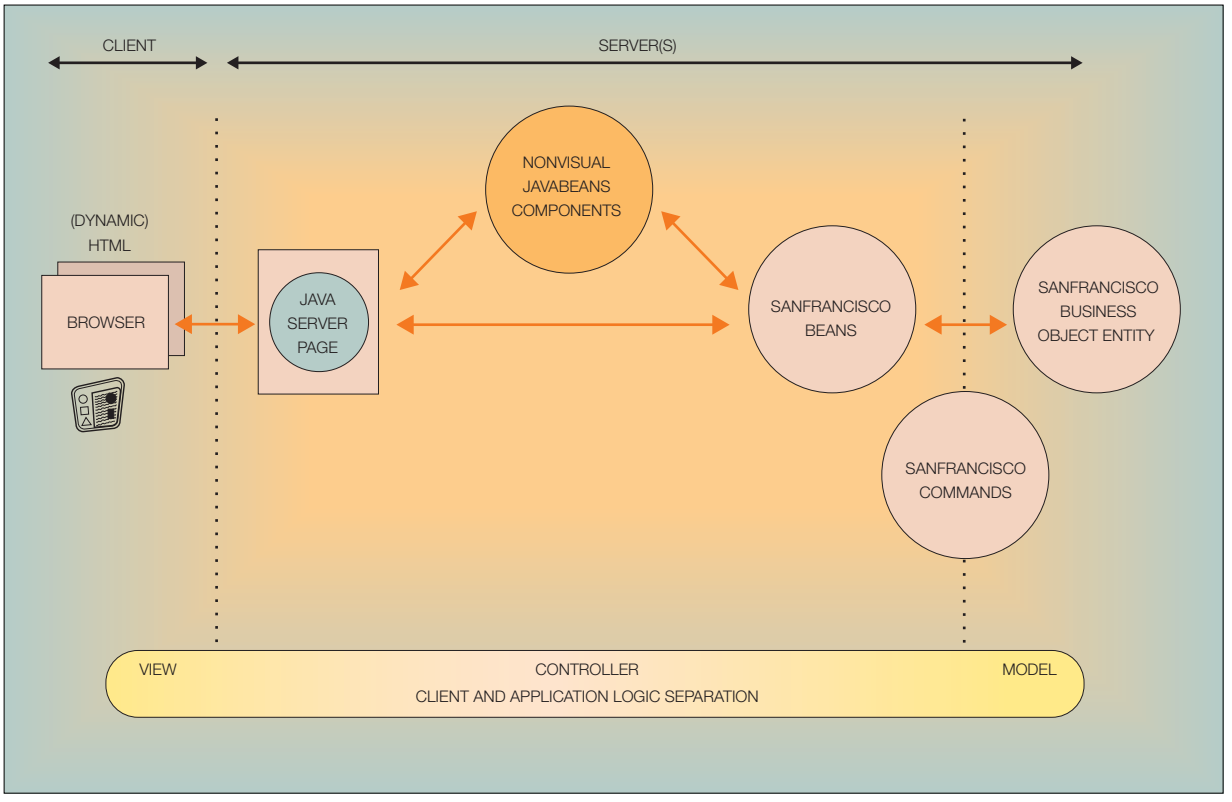
Thin or thick client?

More and more business applications are developed to run in a thin-client environment. This means the actual user uses a browser (or an applet) as the presentation interface, and the actual application logic runs on a middle tier, i.e., the Web server or application server. On the other hand, many applications still run as installed clients on a personal computer. These applications typically use the Swing class library for the graphical user interface.

Business objects and business processes should be developed to be independent from the client environment in which they will be used. JavaBeans components, if used as accessors to server-side business objects as described above, are useful in both environments.

IBM VisualAge for Java allows an application to be created visually by "wiring" together nonvisual and visual beans. Since SanFrancisco beans follow the JavaBeans standard, they can be used in this environment. The client application can now be created without having to know about the server-side programming interface. This is all handled by the SanFrancisco bean. Thus, no manual programming is required, making real component-based application development a reality. (See Figure 2.) Moreover, business processes can be modeled visually by wir-

Figure 3 A thin-client approach



ing nonvisual SanFrancisco beans together. The result of this is yet another bean, which now, on a higher level, represents some business logic. However, it is recommended that such business logic be encapsulated in a "command," which can be executed on the server, leading to better performance. SanFrancisco's predefined business processes are implemented as commands.

A thin-client environment is usually implemented by using servlets. A servlet is a piece of Java code that resides on the server and is called by the Web server to process a request that came from a browser. IBM VisualAge for Java provides a "Visual Servlet Builder" that lets a developer visually create the user interface that a servlet will return. These visual servlets are JavaBeans components, and they can be programmed to interact with other beans, for example, with SanFrancisco beans. Here again, the fact that business objects are "wrapped" by JavaBeans components allows a servlet to be easily created.

Recently, the Java Server Pages standard emerged. A Java Server Page (JSP) can be described as a "reverse" servlet. While a servlet is a Java class that contains HTML or DHMTL pieces, a JSP is an HTML or DHTML page that contains Java code. Ideally, a combination of the two is used, with a servlet handling the nonvisual side of a user request, and the JSP defining the output. (See Figure 3.)

A JSP can retrieve properties from a JavaBeans component through the "⟨bean⟩" tag, using introspection. In a SanFrancisco environment, the servlet would set up the SanFrancisco bean and execute the required calls on it, and then pass it to an associated JSP to display its result state.

An application example

Examining an application that has been built and deployed on the IBM SanFrancisco product will help to illustrate the need for an architecture that pro-

vides flexible access to business objects while maintaining good overall performance. Provider Solutions' Provider*FIRST*** Outcomes Manager[3] application is a health care solution that allows medical personnel to collect, transmit, and view data about patients that are recovering or being treated in nonhospital settings. This includes care given at locations such as nursing homes, or even in-home care. Caregivers enter data about the status of the patients and the treatments that they are being given. Data may be entered into the application using scanning or faxing combined with optical character recognition. This approach eliminates the potential for keying errors, although data may also be entered using a manual keyboard. The application is designed to allow data to be entered or accessed from remote work locations, often where the patient is located. This also allows users in different locations to access a patient's data and collaborate with other caregivers on possible courses of treatment.

Data are stored locally for each branch of the health care agency. This makes the data available for immediate use and analysis. Internet access to the data allows users from different locations to collaborate when reviewing and analyzing the data. Data are also managed centrally, so that information can be shared or combined for analysis of larger areas. Data may be transmitted to external locations, such as government agencies, when necessary. The assessment data are used in a variety of ways, such as determining treatments that are needed, measuring the effectiveness of those treatments, and reporting required information to government agencies.

This diverse environment places several requirements on the application. Many of the users must access the data via the Internet. They must be able to do this without installing software, other than a browser, on their portable workstations. Fax and scanner input of data must also be supported. Security is critical in the application because of the personal nature of the data. It must be possible to limit different users to accessing different portions of the data, and to limit users to performing only authorized actions on the data they can access. Some of the analysis processing that is performed can be computationally intensive. It must be possible to perform that processing on adequate server systems. Several types of data must be maintained by the application. Personal information about patients and information about their treatments, as well as data about the results of those treatments, must be integrated to provide an overall history for each patient.

Architecturally these requirements have been met by distributing both data and processing. An integrated object design supports patient information as well as information on treatments and the effectiveness of those treatments. This same object design is used on each branch's server. A patient's data are stored on the server for the branch that is responsible for the care of the patient. These data may be accessed remotely by others in the health care network by instantiating objects remotely and executing commands local to the entity objects. Then only the results of the commands are moved across the network.

The data from all of the branches are also managed centrally to allow broader analysis. Outcomes form the basis of process improvement within the health care community, allowing better care for a particular diagnosis by monitoring and comparing objectively the results of various protocols of care for that diagnosis.

The data may also be reformatted and transmitted to required government agencies using the mandated government format—generally fixed-format record structures, one record per assessment. Objects are instantiated and the data are extracted and translated into the fixed format. The file is then transferred via FTP (File Transfer Protocol) over a TCP/IP (Transmission Control Protocol/Internet Protocol) connection. Nonvisual SanFrancisco client programs are used to format the file. From an application development standpoint, the file extraction and formatting is treated as just another user interface.

A thin-client architecture is used to provide remote access to the application over the Internet. A second-tier Web server acts as a SanFrancisco client and accesses the SanFrancisco business objects residing on a third-tier data server. A command framework is used to access all entities. Longer-lived objects use commands to execute business logic and return data to the user interface. The Web server code uses the results of the data server processing to dynamically build the pages returned for display in the user's browser.

Different users may access and update different parts of the persistent data. This is controlled by SanFrancisco object security combined with business logic that limits certain users to certain views of the data through the user interface.

Client programs are also used to support data entry via fax and scanners. These programs use the results of optical character recognition to initialize and make persistent new objects associated with a patient's care.

Summary

In order to access a server-side Enterprise JavaBeans component, it is useful to create a client-side JavaBeans component that handles all communication with the server. The client-side component can be migrated when server interfaces change (for example, when SanFrancisco migrates toward EJB support), and it provides a simple interface to the client.

A JavaBeans component can be examined through introspection. It can be used in environments that support visual application construction, like IBM VisualAge for Java. It can be used in both thick- and thin-client environments.

This allows the business object to be created without having a specific client environment in mind. At the same time, the client application can be created without much knowledge about the server-side programming model.

*Trademark or registered trademark of International Business Machines Corporation.

**Trademark or registered trademark of Sun Microsystems, Inc. or Provider Solutions Corporation.

Cited references

1. IBM SanFrancisco Migration to Enterprise JavaBeans, see http: //www.software.ibm.com/ad/sanfrancisco/pdf/sfmigrejb_430a.pdf.
2. T. Lewis, G. Andert, P. Calder, E. Gamma, W. Pree, L. Rosenstein, K. Schmucker, A. Weigand, and J. Vlissides, *Object-Oriented Application Frameworks*, Manning Publications Company, Greenwich, CT (1995), pp. 38–43.
3. E. J. Jaufmann, Jr. and D. C. Logan, "The Use of IBM SanFrancisco Core Business Processes in Human Resources Scheduling," *IBM Systems Journal* **39,** No. 2, 285–292 (2000, this issue).

General references

V. D. Arnold, R. J. Bosch, E. F. Dumstorff, P. J. Helfrich, T. C. Hung, V. M. Johnson, R. F. Persik, and P. D. Whidden, "IBM Business Frameworks: SanFrancisco Project Technical Overview," Technical Forum, *IBM Systems Journal* **36,** No. 3, 437–445 (1997).

K. Bohrer, "Middleware Isolates Business Logic," *Object Magazine* **7,** No. 9, 40–46 (November 1997).

K. Bohrer, V. Johnson, A. Nilsson, and B. Rubin, "Business Process Components for Distributed Object Applications," *Communications of the ACM* **41,** No. 6, 43–48 (June 1998).

C. F. Codella, D. N. Dillenberger, D. F. Ferguson, R. D. Jackson, T. A. Michalsen, and I. Silva-Lepe, "Support for Enterprise JavaBeans in Component Broker," *IBM Systems Journal* **37,** No. 4, 502–538 (1998).

Implementing Application Frameworks: Object-Oriented Frameworks at Work, M. Fayad, D. Schmidt, and R. Johnson, Editors, John Wiley & Sons, Inc., New York (1999).

P. Monday and A. Nilsson, "Real World Java for Business: IBM SanFrancisco Application Business Components," *Cutter IT Journal* **11,** No. 11, 17–23 (November 1998).

SanFrancisco papers in the *IBM Systems Journal* **37,** No. 2, 156–225 (1998).

Accepted for publication November 24, 1999.

André Tost *IBM Software Group, 3605 Highway 52 North, Rochester, Minnesota 55901 (electronic mail: atost@us.ibm.com).* Mr. Tost works as an advisory software engineer for IBM's Software Group. He holds a degree in electrical engineering from Berufsakademie Stuttgart, Germany, and has been working on various object-oriented development projects over the last six years. He currently works as a client architect for the IBM SanFrancisco project.

Verlyn M. Johnson *IBM Software Group, 3605 Highway 52 North, Rochester, Minnesota 55901 (electronic mail: verlyn@us.ibm.com).* Dr. Johnson joined IBM in 1979. His background includes building and maintaining financial and manufacturing applications, designing and administering databases, and building application development tools. Currently he is working with the SanFrancisco group to help customers understand how to best use SanFrancisco, and to build relationships between the development team and customers to ensure that requirements are understood and met in future releases of the product.