Java and the IBM San Francisco project

by B. S. Rubin
A. R. Christ
K. A. Bohrer

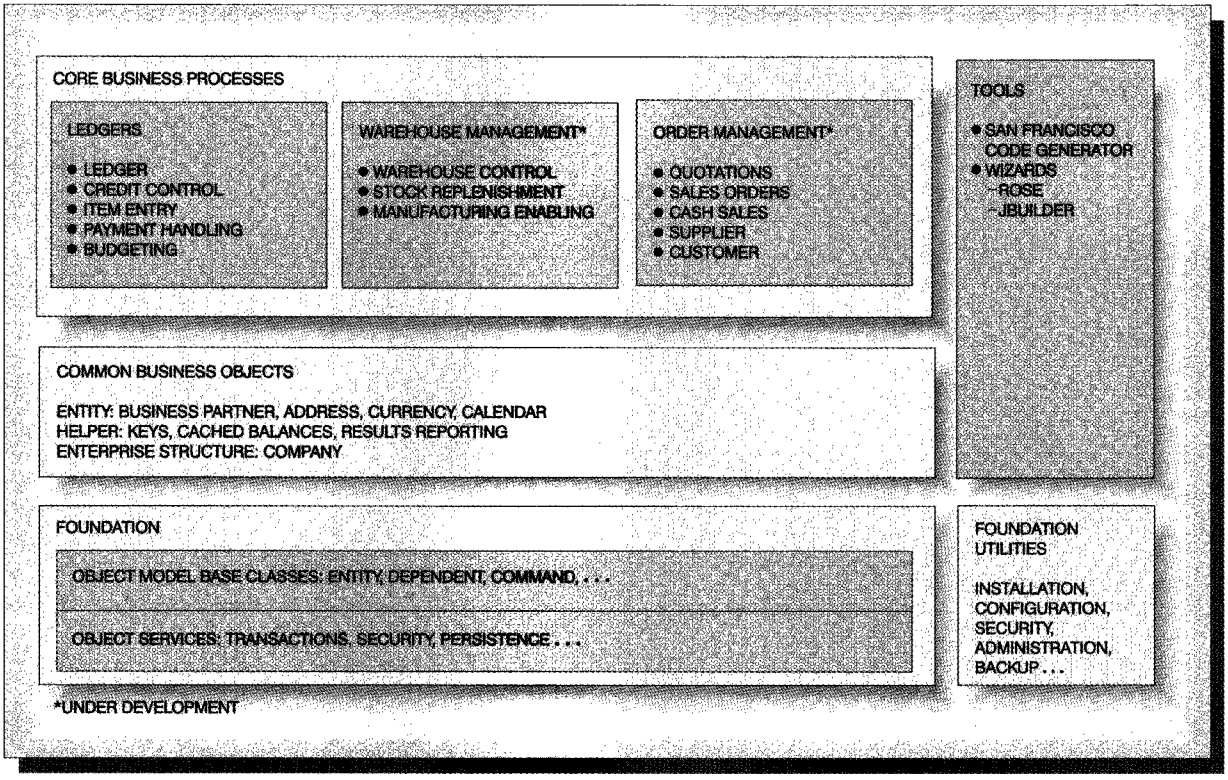
The San Francisco™ project establishes a new paradigm for building business applications. The product, targeted for independent software vendors (ISVs), provides a distributed object infrastructure (foundation), common business objects (CBOs), and business process components (BPCs). Together, they provide a platform-independent business application foundation, ready for extension by ISVs to produce end-customer, business-critical applications. The San Francisco project is written almost entirely in Java™ and to our knowledge is currently the largest Java development effort in the world. This paper provides an overview of the San Francisco project, with emphasis on the Java considerations of the product's development, the lessons learned, and our recommendations for future Java language maturity.

A group of independent software vendors (ISVs) came to IBM a few years ago with several problems, seeking solutions. An increasingly large percentage of their development budget was being spent on multiple platform support, technology infrastructure, and business logic that was not unique to their industries. This left insufficient resources to react to changing customer requirements and to implement the key features that differentiate each ISV from its competitors. Many existing business solutions needed major updates, for issues like the Year 2000 problem and the change to a common European currency. Many ISVs also wanted to update their programming methodologies and skills from procedural to object-oriented development, in order to increase flexibility for quickly adapting to new requirements and to reduce maintenance costs. Many ISVs realized they could not afford the risk involved in proceeding on their own, and needed to work with other companies to deliver competitive solutions.

These requirements led to the San Francisco* project. The first customer shipments of the product, with five major pieces (see Figure 1), were made available in August 1997. The first layer, the foundation, provides a distributed object-oriented infrastructure that implements the San Francisco programming model, simplifying the interfaces with business objects. It also provides a degree of technology insulation from specific object services as well as hardware and operating system platforms. The second layer of San Francisco provides many "common business objects" (CBOs) that are common to multiple business domains and that implement common design patterns for business applications. The first of several planned business process components, general ledger, provides a set (a framework) of interconnected objects that implement the essence of this business domain and are designed for extension and customization by ISVs. ISVs can develop at any of the three layers (the foundation and CBO layers are packaged together and sold as the "base"). A set of development tools, a GUI (graphical user interface) framework, and several utilities, such as print, object save and restore, and configuration, round out the product. The product provides much of the functionality needed by a completed application, so ISVs can focus their resources on competitive differen-

0018-8670/98/$5.00 © 1998 IBM

Figure 1 San Francisco structure



tiators. The product is primarily aimed at applications for small- and medium-sized business markets.

The Version 1, Release 1 product released in August 1997 consisted of about 270 000 lines of Java code (3200 classes, 24 600 methods) and 15 000 lines of C++ code. It contains 51 San Francisco programming model objects, 143 common business objects, 111 general ledger business objects, and 3600 HTML (HyperText Markup Language) pages of documentation along with 156 static and over 300 dynamic object model diagrams represented in the Rational ROSE** tool.

Both Microsoft Windows NT** and IBM AIX* (Advanced Interactive Executive) servers are supported, along with any Java** client compliant with JDK** (Java Development Kit) 1.1.x. San Francisco has been developed with input from more than 200 software vendors who provided requirements and feedback. Some of these vendors also joined the development team, providing an impressive breadth and depth of industry domain knowledge.

As this paper goes to press, the Version 1, Release 2 product is being shipped. This release includes the additional domains of order management, warehouse management, accounts payable, accounts receivable, and AS/400* (Application System/400*) support, along with performance enhancements and improved legacy data support. Release 2 also supports the run-time environment for six national languages. Additional information on San Francisco can be found in two previous issues of the *IBM Systems Journal*,[1,2] including a special issue with a theme devoted to the project, and on our Web site at http://www.ibm.com/java/SanFrancisco.

The foundation layer

Many of the Java-specific aspects of San Francisco pertain to the foundation layer. A key architectural

decision in San Francisco was to develop a programming model, implemented by the foundation, that provided an easy way to create and manipulate business objects. Since most business objects require persistence, security, transactional capability, externalization ability, etc., we decided to simplify the decisions that a business object developer must make when building a distributed-object application. By providing an aggregation of all of these essential object services under high-level abstract interfaces, we provided simplification as well as insulation from specific object service technologies. The services provided by the foundation layer include: object request broker (ORB), factory, transactions, concurrency control, persistence, legacy data treated as business objects, containers for separating object location from application code, object query, collection classes, externalization, naming, notification, local/remote exception transparency, serviceability, server management, national language support (NLS) enabling, and security (authorization and authentication).

The selection of Java for San Francisco

There were many technical and business reasons why Java was selected for the programming language and environment for San Francisco. Java is a platform-neutral, easy-to-use object-oriented language environment allowing higher productivity than C++ and providing a sufficient level of security and robustness. This section explores these advantages in more detail.

A key requirement for San Francisco was that it run on a variety of hardware and software platforms, both servers and clients, and allow an intermixing of servers with different architectures. While we could have built multiple foundation layers to mask these platform differences while presenting a single programming model to the layers above, our development expense was significantly reduced by Java's platform-neutralizing features, allowing for almost the same foundation code for all server platforms.

San Francisco supports both thick and thin clients. For thick-client support, traditional approaches would require a different dynamic link library (DLL) for each supported client. Java allowed us to use one set of code for *all* Java clients, which greatly simplifies the support and administration tasks.

Many of our ISVs were just beginning to use object-oriented development techniques. The learning curve for C++ was too great, yet they wanted some of the features of C++. Therefore, we needed a "business C++" language. Java fit the requirements nicely. Programmers are still faced with significant effort in learning object-oriented concepts, but can then easily tackle the programming syntax issues and become productive quickly. The number of available books on Java and the growing number of universities that have adopted Java as an instructional language, along with the rapid industry acceptance of Java in general, were also key factors in the decision to adopt Java.

Most of our foundation-layer San Francisco developers had previous experience in C++ programming. Many of our CBO and BPC developers had previous experience only in non-object-oriented languages like RPG and COBOL. Their initial attempt to learn C++, then use it with a CORBA** [3] IDL (interface definition language) environment resulted in extremely low productivity. The transition to Java was not only very fast (a matter of weeks), but we estimate that Java provided a productivity boost of 300 percent over C++. Factors for this gain included having fewer language-construct decisions to make, no explicit memory management, no null-pointer problems, no pointer-to-pointer arguments, no dereferencing, no IDL to produce or compile, and clear exception stacks for debugging code. The class library support and language support for threading were also key productivity enhancers. Note that most of our foundation layer development was done with the raw JDK; since it was very important for us to stay abreast of the latest level of Java, we chose not to wait for the integrated development environments (IDEs) to appear. Productivity would have further improved with an IDE, however.

Our development expense was significantly reduced by Java's platform-neutralizing features.

The security benefits of Java have been widely discussed,[4] but the aspect of robustness is especially important in a business-server environment. In a pointer-based language like C++, two business objects

in the same server process could accidentally or intentionally access each other's state. The resulting problems range from unauthorized access of sensitive data to corruption of object structures, leading to robustness problems. Other systems solve this problem by forcing objects into separate processes or introducing tag bits into the memory address architecture. Since a Java user cannot manipulate pointers, these problems are avoided. We have experienced excellent server process robustness using Java.

Careful technology selection

While Java addresses some of the necessary object service technologies today, it currently does not address them all. When we started development, we used what was available, and built what was not. Over time, some of the functions we have built have been addressed in Java. In some cases, we have migrated to the standard solution. Because the San Francisco programming model is at a higher level than many of these technologies, it provides both the upper layers of the product and the ISV applications insulation from these technology choices, allowing us to react to quickly changing technology without impacting the applications and business objects.

To provide a better understanding of how the foundation layer shields the business objects from technology change, here are two specific examples. First, distributed-object computing requires an object request broker (ORB) for handling requests from one object to invoke methods on another in a different process, whether that process is on the same or on a different machine. We wanted a pure Java solution and started with a third-party solution. Sun Microsystem's JavaSoft unit then released RMI (remote method invocation). We switched to RMI without impacting the program model. In response to increasing demand for CORBA IIOP (Internet Inter-Orb Protocol) support, JavaSoft announced that RMI will use the IIOP protocol at a future date. San Francisco intends to use this version of RMI, gaining CORBA IIOP capability. A second example is the code we wrote to parse class files to discover method and attribute names for object queries and persistence. Later, JavaSoft released a low-level reflection capability[5] that we were able to "snap in" without impact to the programming model and with a gain in performance.

We did not adopt every advance in Java for a variety of reasons, including functional immaturity, product schedules, and performance. One example is serialization, which is a Java function that produces a flattened, or "dehydrated," representation of an object for movement or storage. Before the appearance of serialization in Java, we wrote our own externalization and internalization routines in the business object class for this capability. When serialization became available in Java, it appeared to be a way to automatically accomplish the same result. Performance tests showed, however, that our explicit externalization performed much better. Also, Java serialization does not support reading serialized state into an existing object in memory. Lack of this support affects overall application performance, forcing unnecessary instantiations and garbage collection when refreshing an object's state from its persistent store, or when moving local copy state back into the server-side objects. The next two sections expand on other technology selection issues.

Local-remote transparency

To obtain programming-model simplicity, we needed to provide the same interface to local and remote objects. In addition, there are cases where it is more efficient to copy a server object to a client for method execution. We accomplished this by extending the rich functions already provided by RMI. These were implemented by changes to the RMI compiler (this special version is shipped only with San Francisco) and by subclassing RMI function, so that we use RMI run-time code as provided by JavaSoft. Our programming model allows objects to be accessed in one of two ways. *Home* mode means the object is resident on a server and accessed via proxy from the client. *Local* mode means the object is copied into the client and returned to the server at the end of a transaction. A factory call (see Gamma et al.,[6] Factory pattern) that requests an object has an "access mode" parameter that specifies both object location and lock type. This can be specified at either compile or run time. This factory call returns either the proxy or the actual object, and both inherit from a common interface. Applications program to this interface, so whether the object is local or remote is transparent to the application.

We also wanted this transparency to extend to exceptions, so exceptions "thrown" by a server object would appear as if they were thrown by a client object. Remote objects introduce the potential for communication failures, and RMI introduced an exception that must be "caught" by a method, called RemoteException, to cover these cases. We changed the RemoteException method to a run-time excep-

tion so the method calls look the same. A function in the outer transaction scope catches the exception, recovers, and can roll back the transaction. We also needed to be able to associate multiple requests from the same client transaction with the same server thread, so we added thread-switching support. Finally, we added some support to allow implicit flow of control between transaction and security contexts without these appearing in method signatures.

Object persistence

Object persistence was one of the biggest technical challenges, and successes, of San Francisco. We wanted a business object that was architected to be not only independent of specific client or server architecture, but also independent of the specific persistence technology. San Francisco supports plain files and relational databases, and our intention is to use other object storage technologies in the future. Externalization is used to flatten objects to plain files, and the Java Native Interface (JNI) features are used to move data between object attributes and relational database columns via ODBC (Open Database Connectivity, a standard protocol).

Two types of relational database persistence are supported. The first technology is designed for those who are creating new data and do not want to deal with the specific mapping between Java business objects and relational database tables. We provide a function that automatically creates tables with optimal layout for the objects. The second technology is used for existing data. We provide a schema-mapping tool that allows ISVs to map a San Francisco Java business object to a specific existing relational database-table layout. The programming model includes a collection (a grouping of business objects) that maps efficiently to a database table. We can thus treat existing relational database tables as a large collection of business object instances. Access to these objects (rows) leverages the existing database index and query capability, yet appears to the programming model as an object collection and supports object queries.

The ODBC interface and the installation utility are the only components of San Francisco not written in Java. The rest of the foundation layer, the CBO layer, and the BPC layer are written in Java. When JDBC** (Java Database Connectivity) becomes more pervasive and tuned for performance, supports a two-phase commit protocol for robust transactions across servers, and when we have the ability to access private instance data in a Java object via reflection, we will be able to provide a pure Java foundation. Platform-specific code is also needed for the installation utility.

Java object references are limited to only a single Java Virtual Machine (JVM). Since San Francisco needs business objects that can reside on any client or server, and these objects might not be resident in memory, a San Francisco business object references another object via a handle. A handle object contains a network-wide unique identity that includes a container identification, allowing a level of indirection for specifying object location and persistence form during administration time, instead of program development time. It also contains a class identification, supporting class replacement of objects to which the handle refers. The handle can optionally contain a database-table primary key for cases when legacy databases cannot be changed to add an identity column.

We provide a tool that maps a business object to an existing relational database table layout.

In San Francisco, business object creation is not done using a "new" method, but with a *create* call to a factory. The actual object returned might be either a proxy to a remote object or the object itself, as already discussed. The factory might also return a subclass of the requested class. This class replacement technique allows ISVs to replace San Francisco-provided classes with their own subclasses, without change to business process component code. The factory can also override configured default container information to control more specifically where an object is created. An ISV can replace the factory to establish a partitioning of objects across multiple servers and data stores.

Java lessons learned

San Francisco has pushed Java technology as perhaps no other development project has. While we

are very pleased with the results, there are some problems that need continued focus to enhance the role of Java on servers running large business applications.

Performance has been the biggest technical challenge for the development of San Francisco. While we currently support subsecond transaction rates, and continue to make significant advances, we have not yet reached our high goal of matching the performance of traditional procedural applications.

Some of the tuning of San Francisco involved changes to the architecture and programming model, but many of the gains came from changing coding styles, using compilation technology, and changes to the Java Virtual Machine.

In Java, almost everything is an object. This provides usability for programmers, but results in many object instantiations and garbage collections. These are time-consuming operations that are generally not optimized by compilation technologies. A prime example is the Java String class. Manipulation of strings looks simple in source code, but can result in several temporary objects, compounding the creation and destruction times. We made special efforts to reduce the string manipulations in the product and used the Flyweight pattern,[6] which allows a single instance to be used in multiple situations, to reduce object instantiations in general.

The Java class libraries added productivity in our development of the foundation layer and the GUI framework, but were generally not applicable to implementing the persistent business objects. The Java libraries are written for general-purpose needs. We gained performance by rewriting some of these for our needs. For example, the Java HashTable class assumes multithreaded access and uses the synchronization capability of Java. This is a useful, but time-intensive, operation. For cases in the foundation where we knew access was done only by single threading, our own unsynchronized hash table yielded significant savings.

Many benchmarks are available for small or numerically intensive Java applications, and these show great results when just-in-time (JIT) technologies are used. San Francisco has a workload that does not match these benchmarks. We wrote a Java benchmark, inspired by the Transaction Processing Performance Council's benchmark TPC-C,[7] that used the foundation layer. Using this benchmark, we found only a 30 to 40 percent gain from JIT technology, which is far short of the results obtained with other benchmarks. We are exploring the benefits of static compilation technology at this time.

Performance tools are critical for serious applications, and ours have not been adequate up to this point. However, existing tools did allow us to find bottlenecks in San Francisco code and to track memory allocation. The current JVMs have a performance-trace feature that shows an aggregation of method calls, which does not provide sufficient detail for all of our needs. We are working with the University of Minnesota on tools to show call-graph information, traces that show a merger of client and server profiles, and some interesting performance visualization techniques.

Garbage collection is an area in Java that can benefit from more sophisticated algorithms. We see the performance impact of threads being stopped while garbage collection is performed, and we could also benefit from decreased collection times. Also, garbage collection is not a guarantee against memory leaks. We had many cases where we no longer needed objects, but their references were still held by another object so the garbage collector did not free them. Tools that track object creation and destruction would be useful for Java.

Until recently, Java has been used for small applets and programs. The JVMs have not been optimized for server applications with large numbers of threads and classes, high object creation and destruction rates, etc. For example, the Java 1.1 release shows a good deal of heap contention as the number of threads increases, so when a thread tries to perform memory allocation or the garbage collector needs access to the heap, other heap-requesting threads are held off. We are working with others in IBM and in the industry to ensure that the next generation of Java works well for this emerging class of applications.

Conclusion

The San Francisco project promises to establish a new paradigm for developing business applications, providing the platform-independent infrastructure and business components that enable ISVs to build business applications more efficiently than in the past. This paper has focused on the Java aspects of the project, including why Java was an excellent match for the project requirements and some of the Java

technical issues encountered and their solutions, along with areas for future Java maturity. As well as demonstrating that large-scale development of Java business applications can be achieved, the project has influenced Java function, quality, and performance.

Acknowledgments

San Francisco was developed by a team of almost 150 persons: from IBM in Rochester, Minnesota, Böblingen, Germany, and Hannover, Germany, from International Business Systems (IBS) in Solna, Sweden, from JBA International, Warwickshire, United Kingdom, and from several other software vendors. We would like to thank this team for making the San Francisco project a reality and for pushing the limits on new technology in the process.

*Trademark or registered trademark of International Business Machines Corporation.

**Trademark or registered trademark of Microsoft Corporation, Sun Microsystems, Inc., Rational Software Corporation, the Object Management Group, or the Transaction Processing Performance Council.

JBuilder is a trademark of Borland International, Inc.

Cited references and notes

1. 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: San Francisco Project Technical Overview," *IBM Systems Journal* **36**, No. 3, 437–445 (1997).
2. San Francisco theme in *IBM Systems Journal* **37**, No. 2 (1998).
3. Standards have been published by the Object Management Group (OMG) for the Common Object Request Broker Architecture (CORBA). More information about the OMG can be found at http://www.omg.org.
4. JavaSoft Java Security white paper at http://www.javasoft.com/docs/white/index.html.
5. The reflection capability allows Java code to discover information about the fields, methods, and constructors of Java classes.
6. E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley Publishing Co., Reading, MA (1995).
7. Although our Java benchmark does not rigorously conform to TPC-C, it had inspiration from it. More information on TPC-C can be found at http://www.tpc.org/home.page.html.

Accepted for publication March 3, 1998.

Bradley S. Rubin *IBM AS/400 Division, 3605 Highway 52 North, Rochester, Minnesota 55901 (electronic mail: bsr@us.ibm.com).* Dr. Rubin joined IBM in 1985 and has held a variety of positions in hardware and software development, in both technical and management roles, on the AS/400 product. He received his B.S. degree in computer engineering and an M.S. degree in electrical engineering from the University of Illinois, Urbana, and his Ph.D. degree in 1996 from the University of Wisconsin, Madison, specializing in object-oriented databases and information retrieval systems. He was the team leader for the foundation layer in San Francisco and responsible for the proposal and prototyping for moving the San Francisco implementation to Java. He is a senior software engineer and currently the lead architect for the San Francisco project.

A. Ralph Christ *IBM AS/400 Division, 3605 Highway 52 North, Rochester, Minnesota 55901 (electronic mail: christ@us.ibm.com).* Mr. Christ joined IBM in 1980. He has held a variety of positions in System/38™ and AS/400 operating system development, as well as object-oriented application development. He received a B.S. degree in computer science at the University of Wisconsin. Mr. Christ was the initial development manager on the San Francisco project, and recommended the move to Java. He is currently a senior development manager responsible for San Francisco architecture and performance.

Kathy A. Bohrer *IBM AS/400 Division, 11400 Burnet Road, Austin, Texas 78758 (electronic mail: bohrer@us.ibm.com).* Ms. Bohrer joined IBM in 1974 and is an IBM Distinguished Engineer. She has held lead architectural positions in AIX (Advanced Interactive Executive) operating system development and object-oriented development related to OMG (Object Management Group) services and Taligent frameworks. She received a B.S. degree in electrical engineering from Rice University. Ms. Bohrer was chief architect for Version 1, Release 1 of the San Francisco project. She first encountered Java in early 1995 and recommended its use for business object and application programming. She currently divides her time between San Francisco technical strategy and providing consulting to ISVs.

Reprint Order No. G321-5682.