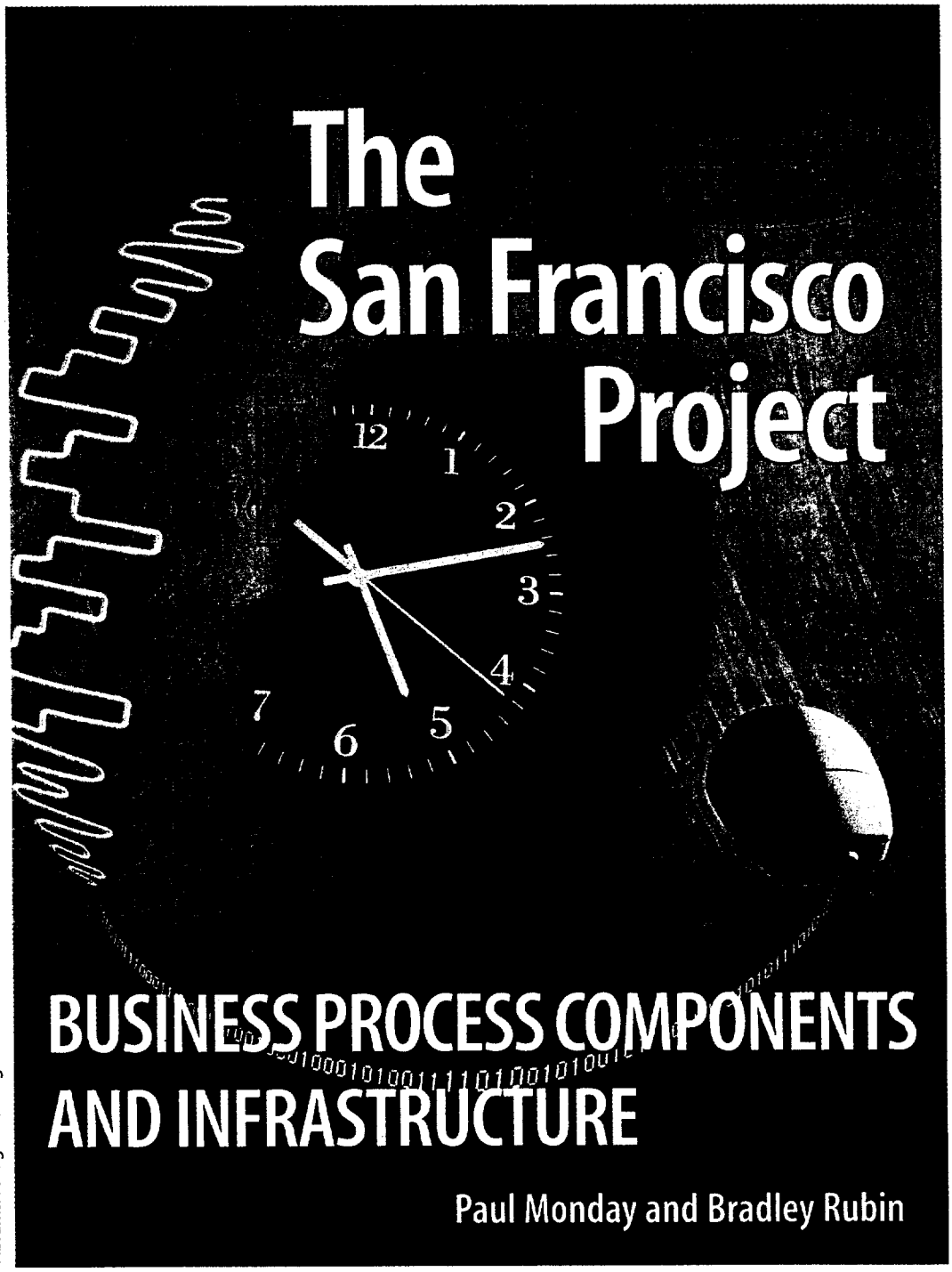

Abrams/Lacagnina/Image Bank

Using Java's strengths to build robust platforms

THE JAVA PROGRAMMING language is having a significant impact on the way applications are developed and deployed. Java makes distributed programming easier and platform neutrality a reality. The San Francisco Project from IBM uses the Java strengths and extends them to build a robust platform for developing object-oriented business applications. The goal of the San Francisco Project is to simplify the move to object-oriented business applications from more traditional procedural business languages and to provide a platform for a new generation of business applications using distributed object-oriented technology. The result of the project is an object-oriented infrastructure and a starter set of well-designed object framework code for ISVs to customize and reuse for building business applications. This article provides an overview of the project with an emphasis on the Java aspects.

There are four problems that the San Francisco Project set out to solve, which were brought to IBM by their business partners:

- Aging applications (often written in COBOL and RPG) are in need of update and rewrite. The reasons vary from the *Year 2000* problem, to adding Internet considerations into a product, to dealing

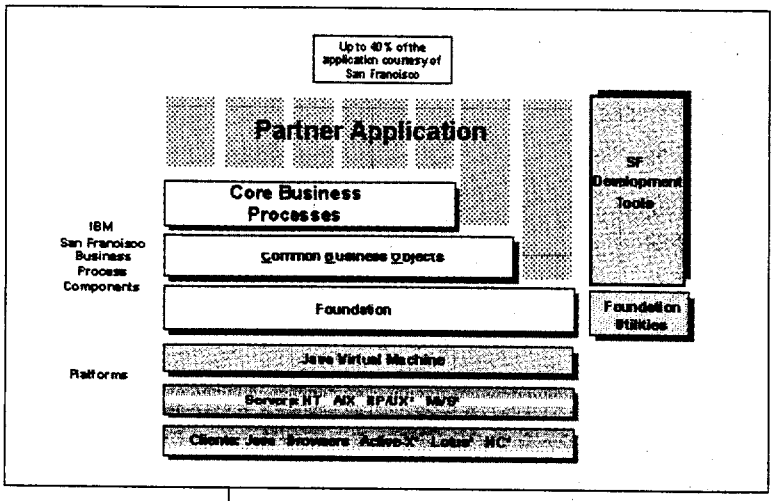

Figure 1.

Overview of the San Francisco architecture.

with the new Eurocurrency.

- Training new skills for an object-oriented solution can be difficult.
- The speed at which target markets move is always accelerating. A solution must be created that allows ISVs to respond to their customer's needs more quickly.
- Applications must be insulated from platform differences to speed delivery to a broader market.

Listing 1.

```
package COM.ibm.sf.samples.pgmmodel.JavaReport;

import COM.ibm.sf.gf.*;

public interface Person extends Entity


    /** Set Person name
     * @param name Person's name
     * @return void
     */
    public void setName( String name ) throws SFException;

    /** Get Person name
     * @return Person name
     */
    public String getName() throws SFException;

    /** Set the person's favorite programming language
     * @param  A string containing the name of a programming language
     */
    public void setFavoriteProgrammingLanguage( String programmingLanguage )
throws SFException;

    /** Gets the value containing the person's favorite programming language
     *  @return A string containing the name of a programming language
     */
    public String getFavoriteProgrammingLanguage( ) throws SFException;

/*-------------------------------------------------------------*/
/* Public Framework Use Only Methods                           */
/*-------------------------------------------------------------*/

    public void initialize(String name) throws SFException;

}
```

Providing object-oriented frameworks in Java addresses these problems. Object-oriented frameworks provide architecture and design and allow code reuse. In designing the San Francisco Frameworks, the goal is to provide approximately 40% of an application as a starting point for IBM's ISV community (see Figure 1). The Java language provides ISVs with a relatively easy-to-learn OO programming language, platform independence, and a handsome payoff when put into the new networked world.

San Francisco provides a programming model that defines a high-level interface for using distributed Java business objects. The implementation of the Programming Model is the Foundation Layer. (Using the Programming Model allows ISVs to concentrate on a higher-level abstraction for using distributed business objects rather than dealing with individual object services.) The programming model also insulates applications from changes in specific underlying object technologies and platform differences.

San Francisco was built with input from more than 200 companies' development talent from 14 countries. IBM released three major components in August '97:

- The object infrastructure, called Foundation.
- Common Business Objects (more than 170 total).
- General Ledger core business process.

Development of other core business processes is underway.

ARCHITECTURE OVERVIEW

The San Francisco Framework is built in 3 layers consisting of approximately 3,200 classes and 24,600 methods consisting of mostly pure Java code (see Figure 2). The layers are:

- Foundation Layer—Provides the Java distributed OO infrastructure for San Francisco and implements the San Francisco Programming Model.
- Common Business Objects—Provides communication between vertical domains, as well as reusable pieces common to each of the vertical domains, like Address.
- Core Business Processes—The top level of the framework, which provides business processes and tasks for ISVs in a vertical business domain, like General Ledger.

THE FOUNDATION LAYER

The Foundation Layer provides object services and a programming model for use in layers above the Foundation Layer. There are two sets of interfaces visible to application developers: a set of base classes and a set of application utilities.

Base classes encapsulate services that are available to all San Francisco business objects. Examples of base classes include:

- Entity—Provides support for independent, persistent, shareable, and distributed objects.
- Dependent—Lightweight classes that are designed to be a part of an Entity. Dependents cannot be separately shared, locked, or distributed.
- Command—A class that encapsulates a piece of business logic, which should be executed atomically. Command execution can be off-loaded to other servers, be a part of the client's transaction, or be executed as its own transaction.
- Factory—Lifecycle management for instances.

Examples of application utilities provide services that are needed by most applications:

- Configuration—Allows customization of the storage type used for object persistence and server location of objects.
- Security—Administration of security information (authorization and authentication).

Most systems available today require that the programmer has an intimate knowledge of the underlying distribution or persistence mechanism. With San Francisco, the programmer creates his/her instance with a factory. The factory uses the configuration data provided by the system administrator to determine where and how to persist the instance. The programmer has no knowledge of how the persistence is taking place, or even the physical location of the instance (whether it's in a posix file on a UNIX system, an ODBC table on an NT server, or a single level store object on an AS/400). The San Francisco Business Object is independent of machine architecture and persistence mechanism.

COMMON BUSINESS OBJECTS FRAMEWORK

The Common Business Object (CBOF) layer is built using the programming model. There are three primary reasons for the existence of this layer they provide:

- Classes that are used across multiple vertical domains (Address, Calendar, Business Partner, Currency, etc...).
- Interfaces that allow the vertical domains to communicate with each other and with legacy applications.
- Classes and design patterns that are useful to the vertical domains.

The classes provided within the CBO layer are more than simple classes providing a set of fields. The Address class provides format control, and relates the Address to the Locale and Language that the application is running under. The Currency class supports arithmetic and logical comparisons of currencies, exchange rate conversion, and more.

As core business processes are developed, and San Francisco developers attempt to interact with legacy applications, they will need to communicate. To communicate, an interface is developed for the desired business processing in the CBO layer. Each process accesses the other processes through the common interface. The first

application interface provided with San Francisco is the interface to General Ledger.

The third function provided in CBO is a finer grained set of objects that are useful to the vertical domains. Three examples of these are:

- *Keyables*—Used to create composite keys and support balances across different composite keys.
- *Classification Types*—Represent user defined types that modify business policies.
- *Result Messages*—Accumulate multiple warnings and errors from complex validation mechanisms.

CORE BUSINESS PROCESS

The top layer of San Francisco provides default business process behavior. The frameworks provide extension points for ISVs to add function and well-defined places where default behavior can be overridden. Several application domains are designed:

- *Business Financials*—Includes accounts payable, accounts receivable, and general ledger. General Ledger was shipped in August '97.
- *Order Management*—Sales and purchase orders.
- *Warehouse Management*—Receiving and shipping of materials.

Application designers will leverage the architecture and design of the supplied frameworks to jump start their own applications. It is expected that the frameworks will provide approximately 40% of an ISV's application.

THE SAN FRANCISCO PROGRAMMING MODEL: A SHORT SAMPLE

A short sample program can illustrate how the Foundation Layer aids building a client that uses Business Objects. The following sample code completes the following tasks:

- Starts a transaction.
- Retrieves an instance of "Person" named "Stella Big."
- Changes Stella's favorite programming language to "Java."
- Commits the transaction.

The Person class is a simple San Francisco Entity with some read/write methods on it. We will not go into the concrete implementation of Person (usually in a Java class named PersonImpl) here. The interface for Person is in Listing 1.

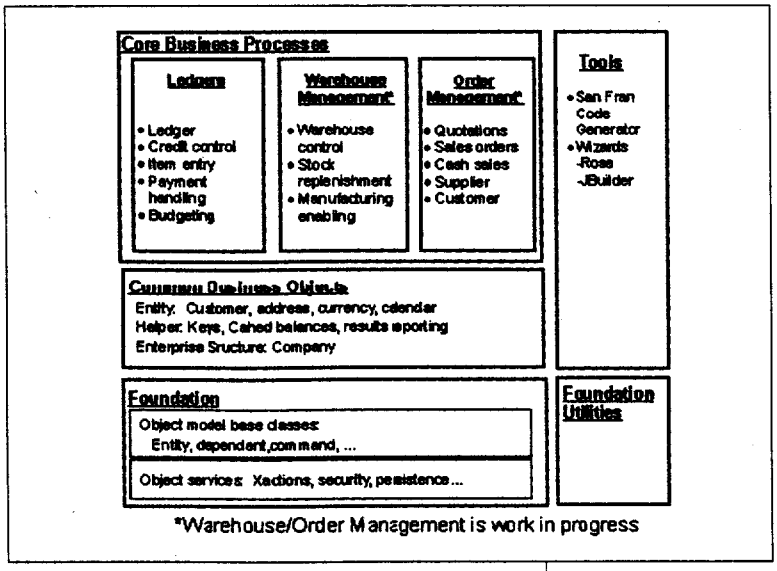

Figure 2.
Highlights of San Francisco layers.

Listing 2 contains a short San Francisco program. As you may have guessed, the Foundation Layer is doing much work "underneath the covers." Each line of source

code that interacts with San Francisco is discussed here in more depth.

```
Global.factory().begin();
```

Start a transaction by retrieving this process' factory from the Global singleton instance. The factory contains the object lifecycle methods, which is where transaction management methods are placed. From this point, all instances which are made "dirty" will be written back to persistent storage upon a commit. If an error occurs during a transaction, all changes can be rolled back.

```
AccessMode accessMode = AccessMode.
  createNormal();
```

AccessModes are a way to "encapsulate" all of the necessary information for gaining access to an instance. The Foundation Layer provides "default" AccessModes, which can be created by calling static methods. For a "Normal" AccessModes, the following attributes take hold:

Location = HOME (The instance is accessed on the server and client method calls take place via a proxy to the instance. LOCAL would copy the instance to the client.)

Lock = READ (The instance is initially retrieved with a READ lock, but will be upgraded to a WRITE lock if a method attempts to make data in the instance "dirty.")

```
Person person = (Person)Global.factory().
  getEntity( personsName, accessMode);
```

The factory locates the Global Name Server in the San Francisco Logical Network. The Global Name Server contains information such as:

- What server physically owns instances of a class type.
- If a class substitution takes place (San Francisco allows a class to be replaced by a subclass).
- What data store is used to store an instance.
- What UserAlias (name) relates to which instance.

After the Global Name Server tells the factory which server owns the instance, the factory obtains a lock on the instance. In this case, because the AccessLocation is HOME, a proxy to the retrieved instance is returned to the client.

```
person.setFavoriteProgrammingLanguage(
  favoriteLanguage);
```

A simple method call. Because the factory returned a proxy to us, the actual data change will take place on the server that holds the instance. This method, as programmed by the developer who created the concrete Person implementation, also calls a method "setDirty()" upon changing of the favorite programming language. The call to setDirty() upgrades the lock to a WRITE lock.

```
Global.factory().commit();
```

All instances with changes made will be returned to the server and stored in persistent storage. If classes have registered "Interest" with changes occurring to this instance, Notifications will be sent to those objects. All locks are released.

```
Global.factory().rollback();
```

Listing 2.

```
package COM.ibm.sf.samples.pgmmodel.JavaReport;

import  COM.ibm.sf.gf.*;
import java.io.*;
import java.util.*;

public class Client {

    /** Main method */
    public static void main( String argv[] ) throws SFException {
        // Arguments to the program
        // arg[0] = name
        // arg[1] = language
        if(argv.length < 2){
            System.err.println("java Client name language");
            System.exit(1);
            }

    String personsName = argv[0];
    String favoriteLanguage = argv[1];

    // start a transaction
    Global.factory().begin();

    // now create an access mode for the person
    // an accessmode gives information like 'what is the initial
    // lock for an instance,' 'what is the default timeout if
    // the lock cannot be granted,' 'where should the instance
    // be accessed (server or client).'
    // AccessLocation = HOME
    // Lock = Read (will automatically be upgraded if necessary)
    AccessMode accessMode = AccessMode.createNormal();

    try {
            // Get Person object from factory
            Person person = (Person)Global.factory().getEntity( personsName,
              accessMode);
person.setFavoriteProgrammingLanguage(favoriteLanguage);

        Global.factory().commit();
    } catch(Exception e) {
        System.out.println("Commit failed..rolling back transaction");
        Global.factory().rollback();
     }
     System.exit(0);
    } // main

} // end of Client
```

If an exception occurred during any of our method calls, it is possible to rollback all changes that were made.

PERFORMANCE

The development of San Francisco required special attention to performance, and many Java performance lessons were learned along the way. In this article, only a few specific items are mentioned.

JAVA CODING LESSONS

Coding Java is easy, coding well-performing Java requires some thought. Two items, that had high impact on the project, are highlighted here.

Strings are so easy to manipulate—they are dangerous. There are several examples that will show vast improvement if slightly rewritten to either avoid concatenation, use a StringBuffer to store intermediate values, or convert to a character array for direct access. This example revolves around having a static class that writes data to a tracelog. The first iteration came of the form:

```
public class B_Trace {
    private static boolean traceEnabled = false;
    public static void traceLine(String theString)
        if(traceEnabled)
            System.out.println(theString);
    }
    public setTraceEnabled(boolean
    traceEnabled){this.traceEnabled=
    traceEnabled;}
}
```

Again, keep in mind this is a simple example. Within the code the following call is made:

```
B_Trace.traceLine(this.getClass().
  getName() + tracePoint + "ouch that
  hurts" );
```

Even when B_Trace.traceEnabled is set to false, Java is forced into a minimum of two concatenations and an expensive call to get the name of the class we are in. Not only does the concatenation take time, but several objects are built during the assembling of the final String in the process. What most people forget is that in addition to the temporary objects, garbage collection is impacted by objects that only exist for a short time. Overhead occurs even when trace is disabled, because the conversion of the parameters occurs before the method call. By simply adding an isTraceEnabled method to the class, the code can be changed to:

```
if(B_Trace.isTraceEnabled())
    B_Trace.traceLine(this.getClass().getName() +
tracePoint + "ouch that doesn't hurt as much");
```

With this simple change, visible leaps in performance take place.

Another hard-earned lesson is in synchronization. A method call to a synchronized method is considerably more expensive than a simple method call without synchronization. There are several classes that contain Hashtables and Vectors but are already synchronized, thus the synchronized collections are overkill. By creating unsynchronized collections, one can eliminate a substantial amount of overhead when using these Java supplied collection types.

The Java class libraries and collection classes provide flexible and easy-to-use options, but occasionally great benefits can be found in writing your own libraries and collection classes for special situations.

NETWORK TRAFFIC

From the ground up, San Francisco embraces network technology and object distribution.

To control the network variable, minimizing traffic is critical. The Foundation Layer allows objects to be accessed in two different locations, HOME and LOCAL.

- AccessLocation.HOME—Methods are processed by a client via proxy to a server resident object.
- AccessLocation.LOCAL—A copy of the object instance is brought to the client, where the methods are processed.

The two ways to access an instance can be leveraged once the runtime attributes of a client are determined. For instance, a long running process that exists simply to create instances and stores them for use by other clients—this type of process fits well with AccessLocation.HOME, because the client itself is doing no changing of data beyond instance creation. A second type of process would be one that retrieves an instance from persistent storage and runs hundreds of get methods to obtain information about an instance. This type of access may be better suited to AccessLocation.LOCAL so that this information only has to flow across the network once.

FUTURE DIRECTIONS FOR SAN FRANCISCO

On the agenda for future releases are:

- Additional towers for ISVs (including Warehouse Management and Order Management).

- First-class support for more platforms (including AS/400).
- Support and integration of evolving Java standards (including JavaBeans and JFC).
- Tools to integrate the Programming Model with development tools (including IDEs and Beans).

CONCLUSION

San Francisco originated from customer requirements and continues to have customer involvement during the development phase. The goal of the framework is to deliver a platform that ISVs can use to launch their development efforts quickly, efficiently, and with minimal risk. San Francisco achieves these goals using layered object-oriented framework technology using Java. The framework encapsulates architecture, design, code, and business processes that are necessary for consistent flow and implementation of business applications. The Foundation Layer insulates the developers and architects from platform considerations and removes the responsibility for lifecycle considerations and object distribution. San Francisco allows ISVs to focus on business process rather than on the fine details of services and architecture.

AVAILABILITY AND PRICING

San Francisco Toolkit 1 is available. Toolkit 1 includes the Foundation Layer, the Common Business Object Layer, and the General Ledger Framework (which is in the Core Business Process Layer). More tools and frameworks will be available in 1998. A free evaluation edition can be downloaded from the San Francisco Project Web site. To obtain further information, contact IBM through addresses available on the Web site, or hop directly to the marketing page. ■

Paul Monday is a Staff Software Engineer at IBM AS/400 Division in Rochester, MN, and is currently working on the San Francisco Project, a distributed Object-Oriented Business Framework. He can be contacted at pmonday@vnet.ibm.com. Dr. Bradley Rubin is a Senior Software Engineer and Lead Architect for the San Francisco Project at the IBM AS/400 Division. He can be contacted at bsr@us.ibm.com.

URLs

IBM's San Francisco Project
http://www.ibm.com/Java/Sanfrancisco

IBM's San Francisco Marketing Page
http://www.ibm.com/Java/Sanfrancisco/marketing.html

Pierre-Yves Goavec/Image Bank