Architecture of the San Francisco frameworks

by K. A. Bohrer

The San Francisco™ project is an IBM initiative, with over 130 independent software vendors, to provide business process components that can form the basis of rapidly developed distributed solutions for mission-critical business applications. This paper describes the original objectives of the San Francisco project and discusses the methodology, skills, and architecture that were used to achieve those objectives. The paper includes discussion of the importance of design patterns, extension points, and a well-defined programming model used in the San Francisco components. Most topics are touched on briefly to give an overview; some knowledge of object-oriented development techniques is assumed.

The San Francisco* project was conceived as a way to help independent software vendors (ISVs) deliver commercial, mission-critical business management systems that meet current and emerging customer demands. These demands are dominated by rapidly changing business requirements based on the re-engineering that is being done in many companies, and by the growing importance of distributed applications for intranet, Internet, and extranet.[1] Distributed object-oriented (OO) technology was recognized by many of these vendors as a necessary base for their future generation of applications. However, good object-oriented skills are difficult to find, and most of the design and programming skills available in these development shops were for server applications written in RPG and COBOL.

The idea to have IBM collaborate with vendors, to build a set of distributed object frameworks specifically for targeted application domains, was the beginning of the San Francisco project. These frameworks were to provide a substantial starting point for new application development.

This paper does not describe the specific content of the domain frameworks and only touches on a few of the infrastructure capabilities. Instead, it describes the specific objectives of the project and discusses the methodology, skills, and architecture used to achieve those objectives. Reader knowledge of object-oriented development techniques is assumed. Both successful and unsuccessful approaches are mentioned. This includes discussion of the importance of design patterns, extension points, and a well-defined programming model in the San Francisco components. Topics are touched on briefly to give an overview and to illustrate how they are important to the architecture and to achieving the product objectives. More detailed information is available in other papers in this issue and in other publications, including the San Francisco Toolkit documentation, available from the World Wide Web at http://www.ibm.com/java/sanfrancisco.

Project objectives

Three major product objectives were established at the start of product development.

©**Copyright** 1998 by International Business Machines Corporation. Copying in printed form for private use is permitted without payment of royalty provided that (1) each reproduction is done without alteration and (2) the *Journal* reference and IBM copyright notice are included on the first page. The title and abstract, but no other portions, of this paper may be copied or distributed royalty free without further permission by computer-based and other information-service systems. Permission to *republish* any other portion of this paper must be obtained from the Editor.

 0018-8670/98/$5.00 © 1998 IBM

Objective 1: Offer easy entry into OO technology. The first objective was to provide a product that offered an easy entry into the effective use of object-oriented technology to vendors that did not already have OO skills. More specifically, this objective was to be achieved through providing:

- Approximately 40 percent of the application in a reusable form (frameworks)
- Sample graphical user interfaces (GUIs) to exercise all domain framework features
- Support for evolutionary use of the frameworks, starting with existing applications and data
- Components that are easily extensible and modifiable
- Flexible persistent object storage
- An integrated set of tools for San Francisco developers

We decided that the frameworks would provide a reasonably complete model of each chosen application domain, but without the user interface. The frameworks would provide the business objects that encapsulated the persistent business data and business processing of the domain, but leave the user interface design and implementation to the ISV that produced the completed application on the framework base. However, in order to make the product easier to use and understand, we also decided to provide GUI samples that allowed the function of the frameworks to be executed and explored immediately after the product was installed. This was seen as a substantial advantage over requiring ISVs to become skilled enough to develop their own code before being able to execute the framework functionality.

Vendors knew they would not be able to replace their entire suite of applications with San Francisco-based applications at once, although complete replacement was considered to be the eventual goal. So, while the frameworks were designed for top-down new application development, it was necessary to support interoperability between new San Francisco applications and existing applications. From the beginning, both San Francisco-based and non-San Francisco-based applications were to share the same databases. This requirement has grown stronger over time, leading to more emphasis on interfaces that support direct interaction of applications, not just the sharing of data.

Since the frameworks were to provide a base for many different application solutions, they had to be designed to be easily extended with additional business processing and data, and the processes and data provided by the frameworks needed to be easily modifiable. The frameworks had to be reusable worldwide, allowing for country differentiation not only in languages but, more importantly, in business rules.

We felt that the persistent storage for the objects needed to allow for both relational databases and object-oriented databases. Most vendors wanted to use a relational database of their own choice. However, the San Francisco architects felt that there would be increasing value in using object-oriented databases, particularly the protected-mode object support being built into the Operating System/400* (OS/400*) system. Furthermore, the ability to use a simple file system for persistent object storage is very convenient for demonstrations, samples, and unit testing.

Easy use of object-oriented technology, and of frameworks in particular, is impossible without a good set of development tools. The traditional integrated development environment tools were needed; but we also decided that "wizards" would be very useful in guiding ISV developers to the right places in the framework to make modifications, and in providing automatic code generation where possible. A crucial decision in providing development tools was our choice of the Java** language. Java was very well accepted by the application programmers, while C++ was considered much too difficult and error-prone.

Objective 2: Enable applications that allow companies to be more competitive. The second objective was to make sure that the framework components provided a base that would produce applications with significant added value over those of the previous generation. The applications produced on San Francisco frameworks should allow companies that use them to be more competitive. This, after all, is the key reason for companies to buy new application software. This objective was to be realized in the frameworks through:

- True object-oriented design, easily modifiable and extendable
- Support for client/server and other distribution models
- Separation between the user interface and the business objects, allowing multiple and replaceable user interfaces

- Separation between business objects, commands, and selections
- Support for development of additional components

Object-oriented technology, and in particular, a good object-oriented design, was to be used so that the resulting applications would be easier to change in response to business process changes within a company, would be easier to maintain, and would hide the complexity of the variety of distributed environments supported.

The business objects would be supported by an infrastructure that handled transactions, concurrency control, and persistence as invisibly as possible. "Distribution points" would be designed into the frameworks for optimum performance. The frameworks could then be customized to distribute processing and partition data across multiple servers and clients. A unique object accessing scheme would allow application developers to make decisions on optimistic vs pessimistic concurrency control, and local or remote object execution, on a transaction basis without any change to the business objects or the frameworks.

The frameworks would be designed to support a model-view-controller[2] application architecture. This is necessary to allow the business objects to be reused in different business processes with different user interface requirements. For example, an HTML (HyperText Markup Language) page might provide the user interface to a business process delivered over the Internet, while a traditional Microsoft Windows** application GUI might be provided for inhouse business processes. In addition, we intended to follow the more specific model-command-selection architecture that was used in Taligent's CommonPoint**[3] product. During development, this was modified somewhat, keeping the emphasis on commands but dropping any explicit support for selections. The frameworks have been designed to encapsulate business tasks in "command" objects that can be easily executed either as an independent transaction or as part of a larger transaction. Commands can be designated to run on specific servers. These commands offer a convenient target for invocation from workflow logic. They also facilitate partitioning of workload across servers.

The ideal solution for rapid application development may be to have a set of prebuilt components that hook together through a visual development environment with little or no coding required. The first release of San Francisco does not provide such components, but it does support development of these components by others. In particular, components can be built that use Visual Basic** controls or JavaBeans** as the front end to the server-side San Francisco framework function.

Objective 3: Provide an open solution, allowing trade-offs in cost, performance, and skill. The final objective was to ensure that the resulting frameworks allowed enough choices in operating system, hardware, and application architectures to allow ISVs a wide range of cost, performance, and skill trade-offs in determining the nature of their final solutions. Openness to a variety of application architectures has become increasingly important. We wanted to cover the spectrum from "thick" clients where both the user interface and much of the business logic runs on the client, to Internet applications that download HTML pages and applets, to "thin" clients where the business logic is on the server side, along with the data manipulation. These are some of the primary options the frameworks had to support:

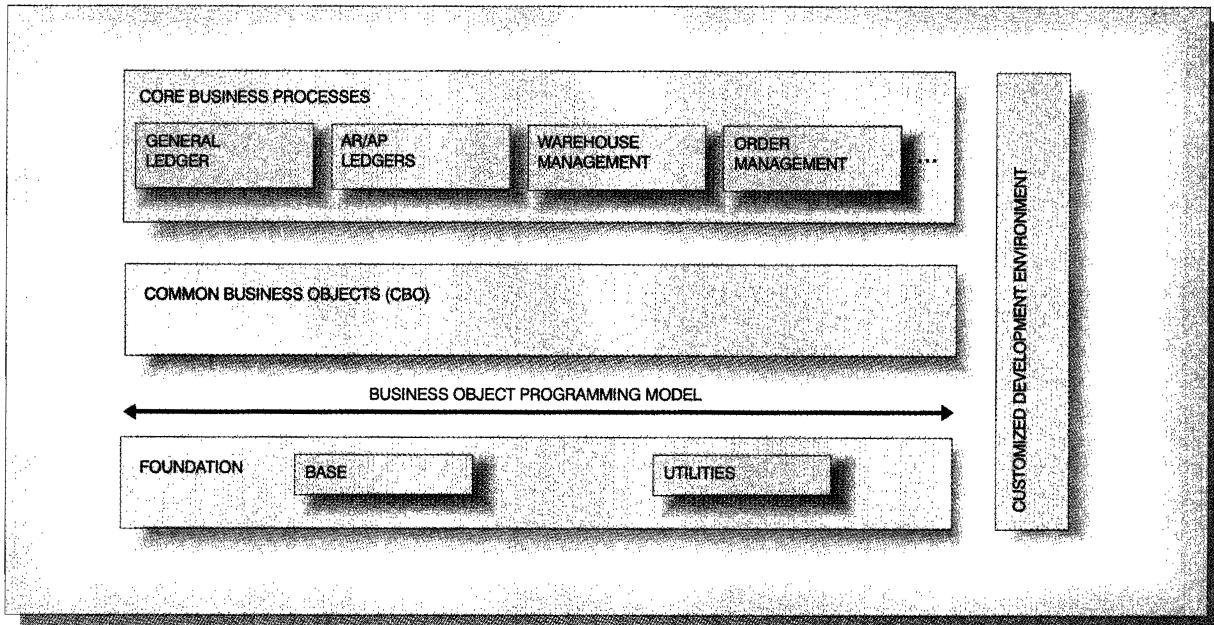
- Operating system options for the client
  - Windows 95**, Windows NT**
  - Any other Java client (in future releases)
- Operating system options for the server
  - Windows NT, AIX* (Advanced Interactive Executive)
  - OS/400, Solaris**, HP-UX** (in future releases)
- Application architecture options
  - HTML pages with applets
  - HTML pages with servlets[4]
  - Traditional applications and legacy applications
  - Visual Basic controls or JavaBean components as the front end to San Francisco objects

System structure

The San Francisco architecture and design were created from the top down (see Figure 1). Domain-specific frameworks, now called "core business processes," provide the highest level of reuse for applications in specific domains. The core business processes currently completed or in development are those for basic business management systems: general ledger, accounts receivable (AR) and accounts payable (AP) ledgers, warehouse management, and order management.

The next-lower level of the system, the common business objects, is provided to achieve the objectives

Figure 1   San Francisco structure



for reusability, understandability, maintainability, and interoperability of the core business processes. The common business tasks and common design solutions found during requirements modeling and analysis of the core business processes are put in the common business objects subsystem. The processes and objects in this layer are then reused across multiple core business processes. This includes domain object classes such as Company, Currency, BusinessPartner, etc. Also included are interfaces needed for different core business processes to interoperate, such as interfaces for posting entries to a general ledger application. There are also technology-level classes in the common business objects layer that implement design patterns that are reused in the design of the business processes. Design patterns are discussed in more detail in a later section.

The lowest level of San Francisco is the foundation infrastructure layer. The foundation includes the distributed object infrastructure and objects that are generally needed in implementing business object models. The foundation layer also includes utilities associated with installing, configuring, administering, and deploying applications built on the infrastructure.

The function in the base infrastructure layer was determined by the needs of the domain objects and resulting applications. Where function was the same as that standardized by Object Management Group's (OMG's) CORBA** and COS** specifications, the San Francisco infrastructure uses the same semantics and similar interfaces. However, we made no attempt to be OMG-compliant or to provide a complete set of OMG-specified object services. Our emphasis on providing the essentials included defining a programming model that hides as much of the infrastructure services and function as possible. Other infrastructures could support the common business objects and core business processes by implementing this programming model. IBM is doing this for its OMG-compliant Component Broker Connector[5] middleware.

Methodology

The methodology used for San Francisco development is based on a combination of Jacobson and Booch methodologies, with modifications specific to producing reusable business frameworks rather than specific application solutions.

Requirements. Development began with a team of domain experts for the first targeted core business process domains, with two experts in object technology as leaders. The domain experts were asked to write a requirements specification for a business application in the target domain, organized by a breakdown of business processes and tasks within those processes. This document became the basis for the "real" requirements document, called the framework requirements.

The domain experts were given education in object-oriented concepts and use-case modeling. Both Booch's[6] and Jacobson's[7] books were provided and relevant chapters were suggested reading. But to provide more specific guidance, since the domain experts generally had no previous knowledge in object-oriented development, the leaders produced a requirements process document and a template for the desired requirements documentation. The objective at this stage was not only to capture the domain function that would be provided, but to structure it in a way that would identify common business processes and identify where the requirements varied across application vendor or company solutions. This would provide a requirements document that would map well to the desired resulting framework structure.

The framework requirements included use-case modeling of the business processes in order to identify primary, secondary, extension, common, and abstract tasks. We modified Jacobson's use-case modeling methodology to add an "inherits" notion for abstract tasks and to emphasize decomposition of tasks within a process to the point where common and abstract tasks across processes were recognized. All user-interface detail was removed from the framework requirements. Instead, the scope of the system, for the purpose of the framework use cases, was constrained to input and output data used by the business tasks. Perhaps the most noticeable difference from traditional use-case-based requirements was our emphasis on classifying tasks as either high, medium, or low volatility with respect to different company or country requirements. This was to identify early which tasks needed to be designed to be easily extended by application developers. Examples of the variations in business rules, or other required flexibility, were explicitly provided in the framework requirements documents.

Object-oriented analysis. When the framework requirements documents were almost completed, additional developers with object-oriented analysis skills were added to the teams. The domain experts were given training in object-oriented analysis techniques similar to Jacobson's methodology. The object-oriented experts were given training in the application domains.

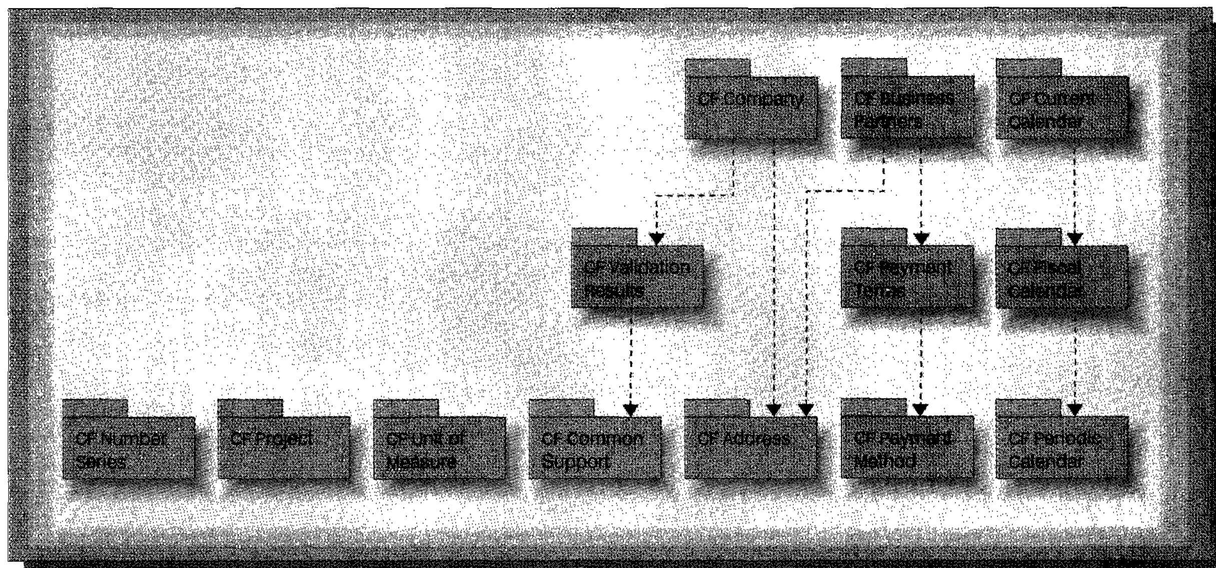
As was done in the requirements phase, the leaders produced an analysis process document explaining the steps to be followed and the expected analysis results. In this case, the analysis was to be documented using the Rational ROSE** tool, with all the analysis work captured in a ROSE model. The Booch notation was augmented to allow objects to be identified as either "entity" or "control" as recommended by Jacobson. Jacobson's notion of interface objects was expanded to "application" objects, which were used to model classes that would not be part of the frameworks but would represent an application's use of the frameworks.

This analysis process was used to produce the static analysis object model, without any intermediate definition of a simpler domain object model.[8] This static model captured the relationships between classes and the cardinality of those relationships. Relationships included inheritance, containment without ownership, containment with ownership, and use. The analysis object model was done without any consideration of design constraints or infrastructure base classes.

Dynamic modeling of the run-time messages between objects was not done at the analysis level. This turned out to be a mistake. Initially it was thought that the domain experts were so knowledgeable that discussion of the dynamic business processing to be performed by the identified objects was sufficient to verify the static object model. We ended up redoing the analysis model later, during dynamic modeling at the design level. We now recommend doing both static and dynamic object modeling at the analysis level, before design and implementation constraints are ever considered.

The stability of the domain or analysis model throughout the product development life cycle is generally considered to be an indicator of the model's correctness. The San Francisco analysis models had quite a bit of change over time. One factor was the lack of dynamic modeling during the initial analysis work, as just mentioned. Another was the lack of object-oriented expertise in the initial analysis work and a significant turnover in domain experts for the fi-

Figure 2 Static dependencies (one-way) between class categories



nancial domains. Where there was continuity of personnel throughout requirements definition, analysis, and subsequent development, the models were more stable and development through design and coding was much more productive.

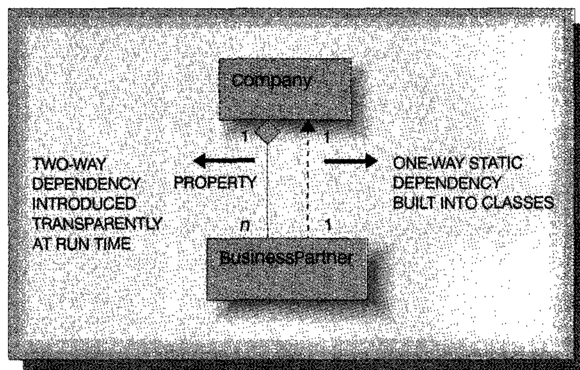
Subsystems and categories. Once analysis-level classes had been identified, they were divided into categories. Top-level categories, now called "package categories," were created to represent the major subsystems. The categories at this level map to Java packages that contain the framework implementation code. Within each top-level category a hierarchy of categories was defined, where the categories grouped together classes that had many dependencies on each other and were primarily involved in the same business tasks. Dependencies between categories were identified.

The overriding rule for assigning classes to categories was that categories could have only one-way dependencies on each other, either directly or indirectly. This ensured that the system could be built in a bottom-up fashion, or top-down where stubs could be used to temporarily satisfy dependencies. This structure of one-way dependency also results in a more modular system, with tight coupling within categories and looser coupling between categories. This makes it much easier to use part of the framework without being required to use, or pay for, the potentially large overhead of parts that are not needed. As an example, part of the category diagram for common business objects is shown in Figure 2, with the dependencies between categories. Notice that the CF Business Partners category depends on the CF Address category. (CF is the Java package name, and the ROSE category prefix, for interfaces and classes in the business object layer.) This is because the BusinessPartner class has an address. However, the Address class has no dependencies on any other class.

Breaking two-way dependencies between categories. In some cases the domain required two-way dependencies, making it difficult to find any reasonable partitioning of the classes. An example of this is the relationship between the Company category and the Business Partners category. In this domain it is often necessary to find all business partners associated with a company. Therefore the static model shows a 1–*n* relationship between the Company class and the BusinessPartner class. (See Figure 3.) There is also a 1–1 relationship from the business partner to the company with which it is doing business.

Situations like this one between the Company and BusinessPartner classes arose frequently in the analysis models. On closer examination, it was recognized

Figure 3 Supporting "loose" two-way dependencies dynamically



that the relationship in one direction was for finding or grouping objects of a class "A" (e.g., BusinessPartner) associated with objects of another class "B" (e.g., Company), where "B" (Company) did not really have to be designed with any particular business logic knowledge of "A" (BusinessPartner). That is, nothing in the behavior of a company object depends on business partner objects. Business partner objects on the other hand do modify their behavior and attributes based on the company object to which they are associated.

To be able to have one-way dependencies between class categories, we support a technique where relationships can be dynamically added between classes at run time through the use of *properties*. The property mechanism is described in more detail in the subsequent section on extension points.

In the case above, the association from Company to BusinessPartner would be dynamically created as a property on Company whenever an application used BusinessPartner. However, classes in the Company category would have no knowledge or code aware of Business Partners category classes.

Prototyping. Prototyping was done at several points in the project to discover or verify development decisions. The first prototype was actually done at the very beginning of the project. While the domain experts were working on the application requirements document, a team of programmers from IBM and from two of the ISV partners was formed. The objective of this team was to build a prototype of an object-oriented application to verify the usability of the development tools and to learn how to merge the development cultures of the three different companies that would be working on the project together.

Soon after this, infrastructure prototyping was initiated to determine whether the object services available within IBM at that time would be adequate and to learn how best to use these services. A third prototype was done by the object-oriented designers while analysis work was in progress. This prototype's objective was to discover and verify design patterns that would be applied when converting the analysis model into a design model. A second infrastructure prototype was done after the switch to a Java-based object infrastructure.

One more prototype was done about halfway through the first release development cycle. This occurred after what we hoped would be the last major iteration on the programming model. The objective was to verify our programming model by prototyping the application transaction models that it was designed to support. This included the applications' structuring of transactions with the user interface code. This turned out to be a difficult area because of the way the Java AWT (Abstract Windows Toolkit) GUI library uses threads.

The prototypes were useful, but in general were given inadequate time for completion. So, although the pattern prototype identified some initial patterns and helped set the programming model, pattern discovery continued during design. The C++ infrastructure prototype was never completed because of the switch to Java. The final application prototype was useful in confirming the programming model, but it should have been more widely publicized to the team. That would have helped more members of the team understand how the frameworks were to be used, saving misunderstandings and leading to sample code for more application structures.

Object-oriented design. Following the completion of the initial analysis model and the pattern prototype, the size of the development team was dramatically increased as domain programmers and object-oriented designers and programmers were added. The composition of the frameworks team (not including the infrastructure team) became about 25 percent domain experts, 25 percent domain programmers, 40 percent object-oriented programmers, and 10 percent object-oriented designers.

Design involved refining both the use-case task descriptions and the analysis model. Each task was turned into one or more scenarios, which included details of the business logic to be implemented, with explicit references to classes used in that implementation. The scenarios were captured in HTML documents with a fairly rigid format. All scenarios shared the same terminology for common actions such as creation, deletion, validation, etc. Scenarios specified input (either mandatory or optional), output, and detailed processing. The domain experts wrote the scenarios.

During design, the analysis model became more detailed. At this point the programming model and its required base classes were introduced. Dynamic modeling was done using the object interaction diagrams in ROSE. The object-oriented designers created object interaction diagrams for the scenarios specified by the domain experts. Analysis-level classes were augmented, generally according to design patterns, to support easy modification of volatile business processes. Design-level classes that were specifically meant to facilitate extension were named with a special prefix that identified them as extension points. Each extension point identified in the OOD (object-oriented design) model follows a design pattern. (Design patterns are discussed in a later section.)

The result of design modeling was not a new ROSE model, but the augmentation of the original analysis-level ROSE model. To the original analysis-level class diagrams, new class diagrams were added, showing the design-level detail. Thus the ROSE model could be viewed at either the analysis or design level. State diagrams were used for objects that had an unusual or complex life cycle. Module diagrams were not used, as the top-level categories corresponded to Java packages, and each object had its own Java interface and class file in the package for its category.

Because extensive reuse of someone else's code is notoriously difficult, we developed and applied design patterns so that similar analysis models would map to designs and implementations that had a high degree of commonality. Two design pattern books[9,10] were used as references and as a starter set of already-established patterns. In addition, all designs were reviewed by the two object-oriented expert team leaders. They were responsible for ensuring that the agreed-upon patterns were being applied appropriately, and for developing new patterns as necessary. New design patterns were documented in a ROSE model, and sometimes in on-line HTML documentation as well. Many of the design patterns simplify modification or extension of the frameworks.

Coding and unit test. Coding was begun by the common business objects team. This was a small team of four, or sometimes five, programmers. Unlike the other domain frameworks teams, members of this team knew C++ and had some level of object-oriented design capability. This team influenced and verified the programming model that was concurrently being designed and documented. The model included the use of collections, the creation pattern for objects, and the accessing of objects.

The designers created object interaction diagrams for the scenarios created by the domain experts.

The programmers of the other teams were given education in C++ and in CORBA-based IDL (interface design language) and object services. The domain programmers had much trouble using C++ and IDL productively. It was not unusual to do 20 compilations before the C++ code compiled cleanly. In addition, the general education provided for object services left the programmers overwhelmed and wondering when and how to use the various services. Remember that object-oriented programming was new to many of these programmers, and the use of distributed objects was new to all of them. It was necessary to simplify the programming task dramatically in order to achieve the dates planned for product shipment.

Four major changes were made early in the coding cycle to increase productivity: (1) C++ was replaced by Java, a simpler language and a more productive development environment, (2) an explicit programming model was provided, covering all OOD relationships and hiding object services, (3) a code generator was provided that read the OOD model as input and adhered to the programming model when generating output, and (4) the programmers were re-

educated on Java and the programming model, as needed, for framework coding.

Java was found to be a much easier language and development environment than C++ and IDL. Java has garbage collection and does not have pointers. This eliminates a major source of errors in C++ programs. It also eliminates the confusion caused by pointer manipulation and dereferencing. The Java development environment we used, with RMI (remote method invocation) rather than an IDL-based distributed object infrastructure, is simpler because it is not necessary to deal in two "languages" (IDL and an implementation language such as C++ or Java).

A programming model was developed and documented that hid as much of the object service infrastructure as possible. It specified how OOD relationships should be implemented in the code. The programming model also specified how to create, find, and access objects within the framework code. Persistence, transactions, concurrency, security, and notification services were all implicitly supported, freeing framework programmers from these concerns.

A "wizard" was produced to convert the ROSE OOD model into Java source code that followed the programming model. This wizard implemented many of the required methods and added collection objects where needed for 1–*n* relationships. During its development, it was discovered that some additional information could be added to the ROSE model that would make the wizard even more useful; for example, directives for the type of collection class to be generated could be added. This led to another iteration on the design model, adding wizard directives and increasing the model's consistency.

Finally, the Java education was much more focused than the earlier education. We did not teach all of Java, its JDK (Java Development Kit) libraries, and RMI. Instead, we taught the Java basics needed for programming the business object frameworks, and we taught the San Francisco programming model. This was much more successful than the earlier approach with standard courses on specific technologies.

Design patterns

As mentioned earlier, design patterns were rigorously developed and applied whenever possible. Design patterns provide a "canned" solution to reoccurring problems, in this case, how to best organize a set of classes to perform a business process and still allow the required extensibility. This is analogous to using prefabricated hardware or preframed doors in house construction. Not everyone needs to understand the best way to design and create a lock, or the best way to frame and hinge a door. Design patterns facilitate the understanding of how a piece of software works, maximize reuse of both the code and design solutions, and therefore minimize maintenance.

Some of the design patterns used in the frameworks come from published works, particularly Gamma.[9] These include:

- Abstract Factory (supports substitution of derived classes with extended subclasses)
- Strategy (separates volatile business rules from the class of the decision maker, allowing multiple strategies to be used)
- Chain of Responsibility (allows any object in a chain to provide the necessary behavior)
- Adapter (maps from one interface to another)
- Composite (supports hierarchies of objects viewed as a single object)

Several new design patterns were developed specifically for our business object frameworks. They appear to be generally useful in commercial business processes. These are:

- Keyables (supports construction of a key from multiple attributes)
- Atomic Entity Update (supports updates whose validity depends on multiple loosely coupled objects)
- Encapsulated Chain of Responsibility (hides the chain of responsibility [COR] from the client in a simple method call)
- Extensible Item (supports the dynamic addition of behavior [methods] to an object)
- Aggregating Entity Controller (collects objects at different levels of the company hierarchy, allowing lower-level companies to see a composite view of parents)
- Shared Entity Attributes (allows some object attributes to be different for lower-level companies, and some attributes to be shared by all companies)
- COR-Driven Strategy (uses COR to find the appropriate business policy [strategy])
- Dynamic Identifier (supports dynamically extending valid values for domain data [types])

- Identifier-Driven Strategy (chooses the correct strategy based on an identifier argument)
- Life Cycle (allows the life cycle defined for an object to be changed statically or dynamically, including the permitted state transitions, and allows additional behavior to be associated with various states)

Extension mechanisms and extension points

The frameworks are designed to be extended. The extension mechanisms are limited to a small number that can be learned and applied over and over in building a completed application from the frameworks. Some of the design model classes were added to provide a clear and easy way to extend the business logic in the framework. Such classes are designated as extension points, and are identified in the OOD model by an "E<xy>_" prefix on their class name. Extension points clearly identify where the frameworks are meant to be extended. In addition to the extension points, some extension mechanisms are applicable to all business object classes, or to all business object classes that inherit from PropertyContainer.

All business object classes, unless otherwise noted, can be extended by building a subclass that adds new attributes (as instance variables or computed values), adds new methods, or overrides existing methods to modify the business logic in those methods. Care must be taken to maintain the "contract" of any overridden method and not to require initialization values for any new instance variables on creation. The new subclass can then be configured into the system as the implementation to be used instead of the original framework class. This is an application of the Abstract Factory pattern.

Any business object class that inherits from PropertyContainer can be extended with new attributes or relationships at run time, without modifying the class source or creating a new subclass. This is because any PropertyContainer can hold an arbitrary number of objects, each with an associated name. Properties are objects that can be added to the PropertyContainer with a name and retrieved by that name. Properties can be owned (or not) by their PropertyContainer. As an example of using a property to establish a relationship, consider a ChaseLetter class. An object of this class could add itself to an object of the DebtItem class in order to establish the fact that the debt item had been "chased." The DebtItem class does not need to be rewritten to know about "chase letter" processing.

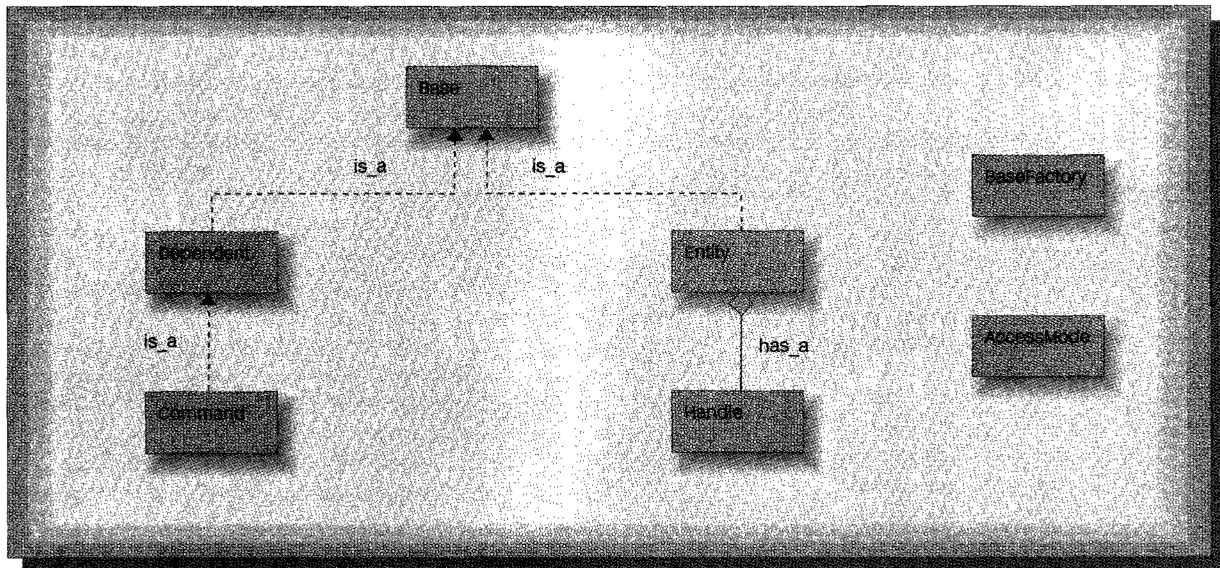
Business objects are all created with static creation methods on a corresponding factory class. For example, an Account class would be paired with an AccountFactory class. The factory class may be subclassed. This subclass would then be configured into the system. The static creation methods on the original factory class will "delegate" to an instance of the subclass. This allows the subclass to change, for example, the primary key used with the new object, or to change the persistent server location of the new object. In this way, substitute factories aid in partitioning objects across servers based on instance data, or mapping objects to legacy databases. If there is no substitute factory class, then the provided factory class uses the default configuration information to decide, on a class basis, where to create the new object, and assigns a unique identifier to be its primary key.

Most of the explicit extension points in the OOD model are policy classes. These classes encapsulate some particular business rule. There are different types of policy classes. In some cases the framework is designed to work with one particular policy, and although this policy can be replaced, a default is provided in the framework. In other cases, the framework is designed to work with multiple policy classes that can be applied either company-wide or system-wide, or found through a chain of objects or through a separately supplied strategy. The last case is really a policy that is used to find policies. In the case of multiple policies, policy subclasses would be created to serve as additional classes to be used at run time, rather than as a replacement subclass for the framework-supplied policy. The policy extension variations are summarized below:

- Replace a default policy
- Add additional policy choices to be used in selected processes
- Change the object chain by which a policy is selected for a process

Another common extension mechanism in the frameworks is the use of the Dynamic Identifier pattern to represent basic application data types. There is often a need in business applications to define a set of values that are valid for a particular attribute. The set of values often depends on the final application, or even on the company in which the application will be deployed. Sometimes an application

Figure 4 Client programming interfaces



should allow run-time additions of new values. Examples might be valid transaction types in a financial system, valid routing destinations of documents, supplier categorizations, etc. Because the valid values are not known at compile time, an enumerated type, or fixed static values, cannot be used. The frameworks use a common set of classes that make up the Dynamic Identifier pattern. These classes are provided in the common business objects layer of the system.

Other extension mechanisms include changing the behavior of an object based on its life-cycle state, defining new summaries of data to be calculated and cached if desired, and adding new keys for quickly accessing objects from within a collection.

Business object programming model

A programming-model document was begun as a way to educate the teams that were to implement the business objects in the ROSE design model. The programming model also became the specification for the function and public interfaces of the infrastructure. This was very important in giving the infrastructure developers a clear vision of what they were building and how it would be used. The programming-model interfaces were divided into three broad categories, based on programming requirements. The client programming model describes interfaces needed when using business objects. The business object developer programming model describes interfaces needed to implement business objects, and the administration interfaces are needed for tuning, configuring, and administering the San Francisco frameworks and the applications built upon them.

Client programming model. The client programming model interfaces specify how a program or application uses business object classes. This includes creating, deleting, copying, and accessing objects. The client programming model also covers working with groups of objects, querying a group or a subgroup, or iterating through a group of objects. In order to support robust, commercial applications, all persistent business objects in San Francisco are accessed and modified from within a transaction. The client programming model supports defining the transaction scope and choosing various access options, such as locking mode and execution location.

The basic business object types are introduced relative to their use (see Figure 4). Two base classes, Entity and Dependent, provide support for business objects that primarily encapsulate data and the associated business logic for those data. One of these would be the base class for any object in the OOD model that is an entity, in Jacobson terminology. The

Dependent class was introduced for lightweight objects, generally representing abstract data types, that would normally be implemented as nondistributed Java objects. However, they need to be able to hold persistent data of other, particularly entity, objects and they need to be able to be passed on remote method calls to entity objects. An Entity subclass is used for persistent business objects that may need to be concurrently shared, independently recoverable in a transaction, globally shared, or remotely invoked.

The Command base class is a specialization of Dependent and intended for control logic that performs business tasks not belonging in any particular entity. Any control object in the OOD model would be implemented as a command object. When a command object is created it can be targeted to execute "near" any persistent business object. This causes the command to be shipped to the server process of that business object, and executed in that process. A command can be run as a single transaction or as part of a larger transaction. When a command is run as a single transaction, the Command class establishes the transaction scope and handles any rollback needed for "uncaught" exceptions.

The client programming model provides clear rules for dealing with business objects derived from each of the base types. For example, any instance of a subclass of Dependent is passed by value with "copy" semantics. Any instance of a subclass of Entity is passed by reference.

The concepts of transaction scope, transient vs persistent objects, owned vs unowned objects, and shared vs nonshared objects are explained by the model documentation. The programming model hides most of the distributed object services. However, some service interfaces are needed when using business objects. In particular, copying, deleting, and accessing an object and controlling transaction scope are part of the client programming model. These interfaces are made as easy to use as possible with methods, either on the object itself, or on a base factory class designed as a singleton[11] for each process. This singleton is accessed through a static method, and it delegates to the appropriate local or remote service objects as necessary.

Business object developer programming model. Additional interfaces were defined for the programmer developing a new business object class or extending an existing one. These make up the business object developer programming model. This model includes the methods that must be provided in the subclasses of the three major base classes: Dependent, Entity, and Command. Documentation for this model explains which methods must be overridden, which may be optionally overridden, and what new methods must be defined. Wherever possible, methods are implemented in the base class so they do not have to be overridden in subclasses.

The business object developer programming model also specifies how attributes and relationships shown in the OOD model should be implemented. The implementation of relationships varies with the cardinality of the relationship: 0–1, 1–1, 0–*n*, and 1–*n*. This also varies depending on whether the relationship involves ownership of the related object(s). Relationships with cardinality of 0–1 or 1–1 are implemented in the same way as attributes. In order to improve performance and decrease resource usage, entity objects are not declared directly within containing or related objects. Instead, a Handle class is used (see Figure 4).

Every entity object has a handle object that can be retrieved, stored, and later used to retrieve its entity. The handle stores the entity's unique identity, hiding its details from the programmer. Since persistent references to other business objects are implemented by storing them in a handle, the infrastructure can delay activating referenced entity objects until they are needed, rather than when a related object is activated. For example, a business partner object might hold a list of addresses. An application might access this object to get the name of this business partner. It would be unfortunate if all the addresses held by this business partner were also activated. Having persistent references to entity objects encapsulated in a handle also makes it possible for the infrastructure to implement reference counts to prevent deletion of entities still in use. This "safe" deletion capability is not in the current product, but we plan to add it in the future.

Although relationships to entity objects are implemented with a handle, client programmers access the entity directly. The get⟨name⟩ method in the containing object uses the stored handle to retrieve its associated entity, which is then returned.

In order to avoid a proliferation of collection classes, collections are not subclassed to define 0–*n* or 1–*n* relationships or collection-element types. Instead, a concrete collection, supplied by the infrastructure,

is used as an implementation detail of a business object that has a $0-n$ or $1-n$ relationship to other objects. The aggregating business object provides type-safe methods for manipulating the n objects. These methods delegate to corresponding non-type-safe methods on the collections. Set, list, and map collections are provided. Elements in collections are identified either by iteration, by query, or by a key value if the collection type is a map. Naming conventions for methods are explained in the documentation, such as get<name> and set<name> methods for attributes of an object, or add<name>, replace<name>, etc., for $0-n$ or $1-n$ relationships.

Early in our development cycle, the infrastructure developers identified scalability and performance problems with the initial collection interfaces. These interfaces could not be easily mapped onto relational database tables to speed up queries. A solution was needed that allowed optimal use of relational database tables for object storage, but did not make the persistence choice explicit to the application or business object developer. That is, the framework and business objects needed to be persistent-storage neutral and continue to present the domain model with minimal exposure of the new constraints being imposed for efficiency.

The result was a new type of concrete collection class, EntityOwningExtent, that from a domain model perspective: gains new elements by having them created in the collection, cannot have already existing objects added to it, and always owns its elements, so removal is always deletion. In addition, an element's key must be composed from attributes of the element.

These semantic changes to the already defined Map collection class were sufficient to allow a collection of this new type to be associated with a table in a database where all elements could be assumed to be in that table. Also, all rows with an optionally specified partition-key value could be assumed to be part of the collection. Whether an extent collection is actually mapped to a database table and partition key can be specified in configuration information and does not have to be coded into the business object classes, collection class, framework, or application software.

The business object programming model is designed to maximize the number of environments and applications in which the business objects can be reused. In particular, business objects are implemented without any knowledge of how their persistent data are stored. Separate schema-mapping information is provided through configuration tools when a relational database is used for persistent storage. It is also possible to use the file system for persistent storage, or, in the future, an object-oriented database. Entities are implemented without determination of transaction scope. So, they can be reused in whatever transaction scope is appropriate to each particular business task or process as defined by commands or by the application.

San Francisco uses "standard" access-mode objects within business objects that must access other objects, in order to allow locking and execution choices to be made by the application or command object. The application or command can configure the "standard" access modes to be the desired lock modes and execution locations. For example, one task can set the "normal" access mode to optimistic lock mode and local location. This would cause objects to be locked optimistically and a working copy instantiated in the local process of the caller. Another task might set the "normal" access mode to be read lock mode and home location. This would cause pessimistic read/write locks to be used and a proxy to the remote object would be instantiated in the local process of the caller.

Complete information on the San Francisco programming model is in the product documentation programming guide. A good introduction can be found in *Object Magazine*.[12]

Conclusion

This paper presented the overall objectives, development methodology, architecture, and some of the more interesting design points of the San Francisco project. Of course, there is much more to know about San Francisco. Information on obtaining the San Francisco Toolkit or an evaluation version is available on the Web site at http://www.ibm.com/java/sanfrancisco.

The architecture and development of application domain-specific frameworks that can be easily extended and modified for use in many different applications presented many challenges. The continual focus on the end result, driving design from the top down, enlisting domain experts who would eventually build applications using the system, were key to meeting the objectives in a timely fashion.

*Trademark or registered trademark of International Business Machines Corporation.

**Trademark or registered trademark of Sun Microsystems, Inc., Microsoft Corporation, Taligent, Inc., Hewlett-Packard Company, Object Management Group, or Rational Software Corporation.

Cited references and notes

1. An extranet links intranets to each other and to the Internet.
2. The model-view-controller (MVC) architecture separates the management of information (model) from its visual representation (view). The controller provides the means by which changes are triggered in either the model or the view, and is separate from both.
3. A discussion of the model-command-selection architecture is available at http://www.rs6000.ibm.com/aix_resource/Pubs/redbooks/htmlbooks/sg244474.00/44740211.html.
4. Like an applet, a servlet is a small Java application. Unlike an applet, which is designed to run within a Web page in a browser, a servlet is designed to run on a Web server.
5. See http://www.internet.ibm.com/news/25c2.html.
6. G. Booch, *Object-Oriented Design with Applications*, The Benjamin/Cummings Publishing Co., Redwood City, CA (1991).
7. I. Jacobson, M. Christerson, P. Jonsson, and G. Overgaard, *Object-Oriented Software Engineering: A Use Case Driven Approach*, ACM Press, Addison-Wesley Publishing Co., Reading, MA (1992).
8. In Jacobson's methodology, the domain model consists of objects that have real-world counterparts in the problem domain. The analysis model adds objects for interacting with the domain objects from outside the system, and for sequencing these interactions.
9. E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley Publishing Co., Reading, MA (1995).
10. J. Coplien and D. Schmidt, *Pattern Languages of Program Design*, Addison-Wesley Publishing Co., Reading, MA (1995).
11. A class that produces a singleton allows only one such object to be created.
12. K. Bohrer, "Middleware Isolates Business Logic," *Object Magazine* **7**, No. 9 (November 1997); also available at http://www.sigs.com/publications/objm/9711/bohrer.html.

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: San Francisco Project Technical Overview," *IBM Systems Journal* **36**, No. 3, 437–445 (1997).

R. C. Martin, *Designing Object-Oriented C++ Applications: Using the Booch Method*, Prentice Hall, Englewood Cliffs, NJ (1995).

O. Sims, *Business Objects: Delivering Cooperative Objects for Client-Server*, McGraw-Hill Book Company Europe, Maidenhead, Berkshire, England (1994).

Accepted for publication December 11, 1997.

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. in electrical engineering from Rice University. Ms. Bohrer was chief architect for the current, first release of the San Francisco product. She currently divides her time between San Francisco technical strategy and providing consulting services to ISVs.

Reprint Order No. G321-5669.