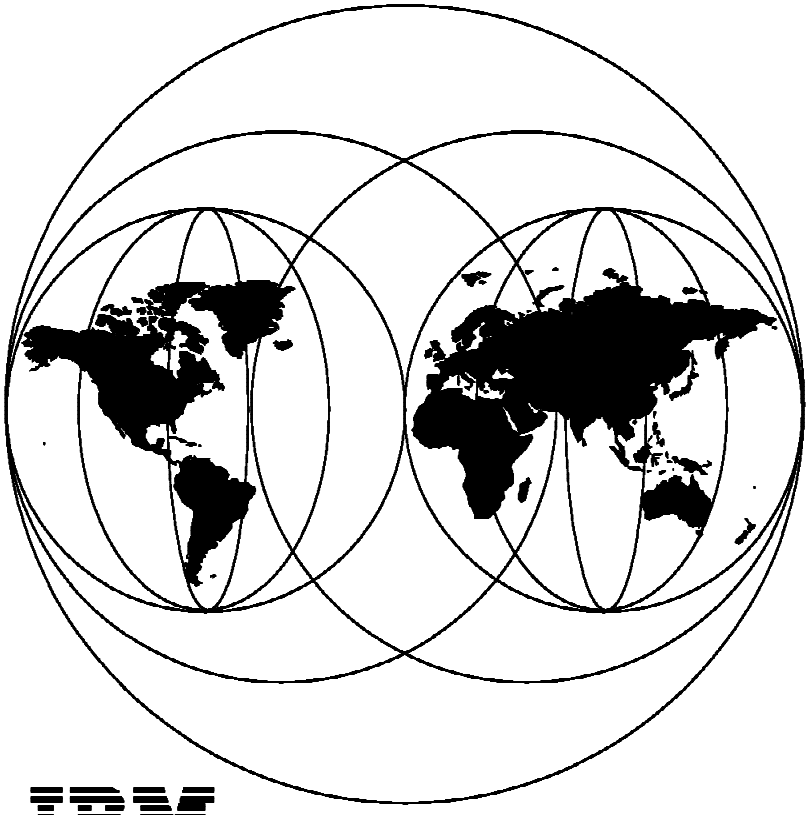
SG24-2157-00

San Francisco Concepts & Facilities

February 1998



IBM

International Technical Support Organization

Rochester Center

International Technical Support Organization

SG24-2157-00

San Francisco Concepts & Facilities

February 1998

Take Note!

Before using this information and the product it supports, be sure to read the general information in Appendix A, “Special Notices” on page 115.

First Edition (February 1998)

This edition applies to the IBM San Francisco Business Process Components Version 1 Release 1 Modification 0 (V1R1M0).

Comments may be addressed to:
IBM Corporation, International Technical Support Organization
Dept. JLU Building 107-2
3605 Highway 52N
Rochester, Minnesota 55901-7829

Contents

Figures

Preface

This book provides an overview of the IBM San Francisco Business Process Components and their benefits. It introduces San Francisco, its major components, key concepts, and considerations. It also provides an overview of application development using San Francisco.

The intended audience for this book is people with business knowledge as well as people with an information technology background. The book is not intended to be a technical reference; the San Francisco documentation provides the information you need to develop an application.

The book guides you through San Francisco in the following manner:

- Introduces the San Francisco approach to an application
- Explains the San Francisco value proposition
- Describes the layers of the San Francisco components and concepts
- Introduces the design patterns used for implementing San Francisco
- Discusses the utilities which are part of the San Francisco Base
- Gives an overview of application development with San Francisco
- Explains the methodology that is introduced by the San Francisco Roadmap
- Points out the issues to consider when planning to move to San Francisco
- Discusses the tools that are needed to develop with San Francisco

San Francisco requires knowledge of object-oriented technology and Java programming. This book also assumes familiarity with these base concepts. The bibliography points you to reference works that can help you acquire the necessary skills in these areas.

As an executive or manager, you should read the following sections:

- Chapter 1, “Introduction to San Francisco” on page 1
- Chapter 2, “San Francisco Architectural Approach” on page 9
- Introduction to each of the subsequent chapters

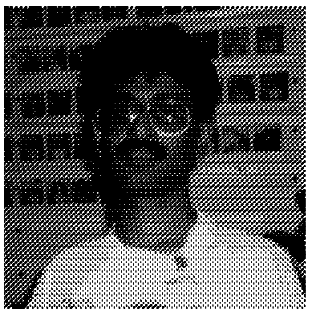
As a domain expert or business analyst, you should read the following sections:

- Chapter 1, “Introduction to San Francisco” on page 1
- Chapter 8, “San Francisco Application Development Methodology” on page 81.
- Chapter 5, “San Francisco Core Business Processes” on page 49
- Introduction to each of the other chapters

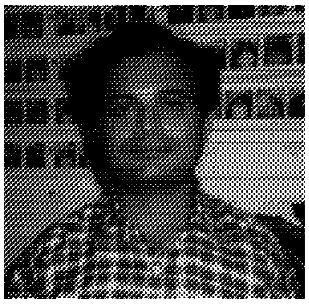
As a technical developer, you will find most of the book helpful.

The Team That Wrote This Redbook

This redbook was produced by a team of specialists from around the world working at the International Technical Support Organization Rochester Center.

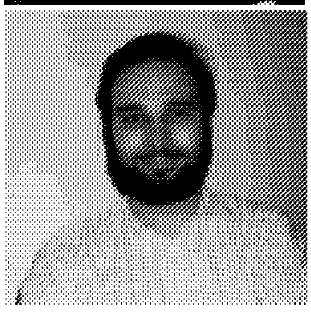


Srinath Abinavam is a Consultant working for Fast, an IBM Business Partner in New York. He has more than 4 years of experience in the I/T technology industry. His areas of expertise include object-oriented technology, mainly focusing on C + + and Obsydian programming. He holds a degree in Electronic Engineering from the Bharathiar University and a diploma of Systems Management from the NIIT, India.

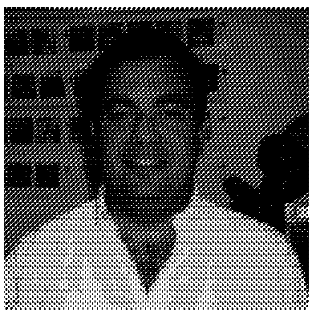


Amit Buch has worked as a Scientist in the Indian Space Research Organization (ISRO) in India for 10 years. Currently, he is a Consultant working for Fast, an IBM Business Partner in New York. His areas of interest include object-oriented analysis and design for complex systems, knowledge-based systems, and artificial neural networks. He holds a master's degree in Physics from the University of Saurashtra, India.

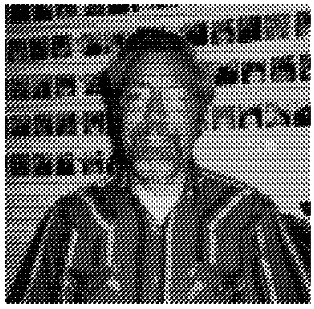
Eric Cattoir is a Support Engineer for San Francisco. He joined IBM Belgium in 1994, where his major task was to get IBM Business Partners up to speed modernizing their application suite. In this job, he got a lot of experience in C/S development, AS/400 connectivity, OO technology, and mentoring. In his current job, he is responsible for educating software developers on San Francisco. He holds an engineering degree in Electronics from the University of Ghent, Belgium.



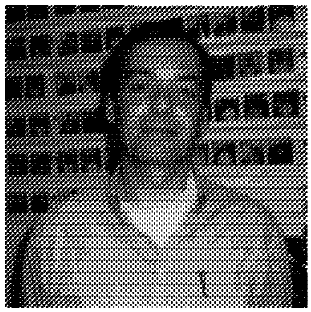
Michele Chilanti is a Support Engineer for San Francisco. He has 8 years of experience in the AS/400 field. He holds a degree in Electrical Engineering. His areas of expertise include application development, database, and object-oriented programming. In his current job, he is responsible for educating software developers on San Francisco.



Sherif El-Rafei is an Object-Oriented Technical Consultant and Mentor at IBM Middle East's Arabic Competence Center (ACC) in Cairo. He joined ACC in 1984 as an Application Designer. He has worked for 3 years in IBM Toronto Lab as an Architect, and has been involved with standards organizations (ISO, X/Open, and Unicode). His current interests are OO modeling, frameworks, and architectures. He also conducts OO mentoring and education at IBM's education center in La Hulpe. He holds a BSC in Electronics from Cairo University.



Alan Krause is an Application Consultant working for Management Technology Group, an IBM Business Partner in Denver, Colorado. He has 28 years of I/T experience and has specialized on the AS/400 system in the areas of application design and development, performance tuning, and CA/400. Most recently, he has designed and developed AS/400 client/server applications using Visual Basic, PowerBuilder, and Java. He has also developed and presented many classes and workshops on the AS/400 system and other PC programming subjects.



Werner Müller is an I/T Specialist in IBM Austria. He has more than 10 years of experience in application development on the IBM System/36 and the AS/400 system. He has 5 years of experience in developing object-oriented applications in a client/server environment as well as framework development. His special area of interest is GUI development. He holds a degree in Informatics from the Technical University of Vienna, Austria.

Fernando Zuliani is an AS/400 Certified I/T Specialist in the ITSO Rochester. He has more then 10 years of experience in the I/T field. He has worked at IBM for 9 years. His areas of expertise include OS/400, client/server programming, VisualAge Generator, application development, and performance. In his current job, he is the ITSO Project Leader for San Francisco. He holds a master's degree in Mathematics from the University of São Paulo, Brazil.

Thanks to the following people for their invaluable contributions to this project:

William F. Berg
Java Development, Rochester Lab

Curtis H. Brobst
San Francisco Base Development, Rochester Lab

James E. Carey
San Francisco CBOs and Core Business Process Development, Rochester Lab

Brent A. Carlson
San Francisco CBOs and Core Business Process Development, Stockholm Lab

Mary K. Dangler
San Francisco course developer/instructor, Partners in Development, Rochester Lab

Chuck S. Gauthier
San Francisco Tools Development, Rochester Lab

Scott A. Gerard
Persistent Object Management, Rochester Lab

Greg A. Hoffa
Object-Oriented Technology, Rochester Lab

Tim C. Hung
San Francisco Project Support, Rochester Lab

Katie A. Imming
San Francisco Project Support, Rochester Lab

Michael N. Jacobs
San Francisco Tools Development, Rochester Lab

Günther Kalod
San Francisco User Interface Development, APDC Vienna, Austria

Teresa C. Kan
Java Object Transaction Services and Schema Mapping, Rochester Lab

Thomas C. Lindner
I/T Specialist, ADCC Vienna, Austria

Victoria J. Mathews
San Francisco Consumability Team Leader, Rochester Lab

Mike D. McKeehan
San Francisco Base Development, Rochester Lab

Paul B. Monday
Java Beans for San Francisco Development, Rochester Lab

John J. Palof
San Francisco Tools Development, Rochester Lab

Mark A. Pasch
San Francisco Performance, Rochester Lab

LindaMay R. Patterson
San Francisco course developer/instructor, Partners in Development, Rochester Lab

Dianne E. Richards
San Francisco Base Development, Rochester Lab

Paula H. Richards
Java and San Francisco Application Development, Partners in Development, Rochester Lab

Brad S. Rubin, PhD.
San Francisco Lead Architect, Rochester Lab

Jeff M. Ryan
San Francisco Base Development, Rochester Lab

Clark A. Scholten
San Francisco course developer/instructor, Partners in Development, Rochester Lab

John R. Stoeckel
AS/400 Java Development, Rochester Lab

Toni Taliaferro
San Francisco Information Development, Rochester Lab

Jay D. Toogood
San Francisco Tools Development, Rochester Lab

Albert Unterfrauner
Product Development, APDC Vienna, Austria

Robert G. Waite
San Francisco Information Development, Rochester Lab

Comments Welcome

Your comments are important to us!

We want our redbooks to be as helpful as possible. Please send us your comments about this or other redbooks in one of the following ways:

- Fax the evaluation form found in “ITSO Redbook Evaluation” on page 125 to the fax number shown on the form.
- Use the electronic evaluation form found on the Redbooks Web sites:

  For Internet users `http://www.redbooks.ibm.com`
  For IBM Intranet users `http://w3.itso.ibm.com`

- Send us a note at the following address:

  `redbook@vnet.ibm.com`

Chapter 1. Introduction to San Francisco

This chapter provides a context for understanding the value of San Francisco. Section 1.1, “How This Book Is Organized” explains the structure of this book. Section 1.2, “The Evolution of Automated Business Management Systems” identifies the information technology trends that have influenced the San Francisco product. Then section 1.3, “The San Francisco Frameworks” on page 5 gives a high-level introduction to San Francisco and demonstrates the value of the product.

1.1 How This Book Is Organized

This book is an introduction to San Francisco and, therefore, is suitable for a broad audience. Information technology specialists, business domain experts, and technical leads of application development teams will all benefit from reading this book by becoming familiar with the aspects of San Francisco that are relevant to their activities. We have structured this book as follows:

- Chapter 1, “Introduction to San Francisco” positions San Francisco as an asset in your application development strategy.
- Chapter 2, “San Francisco Architectural Approach” on page 9 gives an overview of the architecture and content of San Francisco.
- Chapter 3, “San Francisco Foundation” on page 23, Chapter 4, “San Francisco Common Business Objects (CBOs)” on page 39, and Chapter 5, “San Francisco Core Business Processes” on page 49 deal with each of the layers of the architecture in more detail.
- Chapter 6, “San Francisco Patterns” on page 55 introduces the important concept of design patterns and shows how these are utilized and applied by San Francisco.
- Chapter 7, “San Francisco Utilities” on page 65 gives some guidelines on available utilities that you can use while developing applications based on San Francisco.
- Chapter 8, “San Francisco Application Development Methodology” on page 81 explains what methodologies are recommended for application development with San Francisco.
- Chapter 9, “San Francisco Open Tools Strategy” on page 93 establishes a San Francisco Open Tools Strategy, and highlights some of the tools already available.
- Chapter 10, “Developing Applications on Top of San Francisco” on page 99 discusses the considerations you need to address to develop an application based on San Francisco.

1.2 The Evolution of Automated Business Management Systems

Business management systems have been in use for nearly three decades. Many existing business management systems have problems in dealing with today's rapidly changing business environment, especially when it comes to integrating changes in the business processes and modernizing the development environment. In addition, traditional business management systems tend to be

highly platform specific. The following technology trends have influenced and complicated this evolution further.

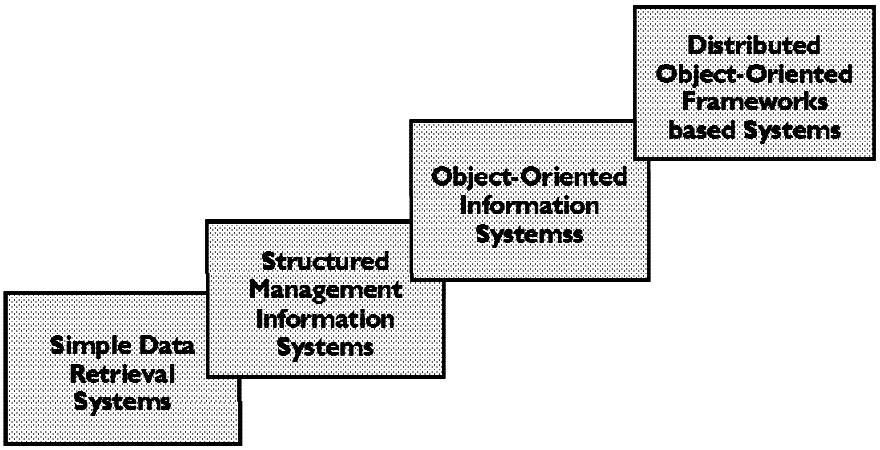


Figure 1. Evolution of Business Management Systems

1.2.1 Client/Server Architecture

Large enterprises employing an array of such systems felt a strong need to migrate to a more flexible computing architecture that grants broader autonomy to the end users. This led to the client/server (C/S) computing model, which enables multiple users (clients) to connect to a single server or cluster of servers through a communications network. This architecture leverages the ability of the servers to ensure data and transaction integrity and at the same time grants a high degree of autonomy to the clients. In exchange for their increased flexibility, these systems turned out to be tightly coupled to the underlying operating system and to its hardware resources. Recent versions of client/server systems employ object-oriented (OO) technology to encourage reusability and to lower the cost of maintenance.

1.2.2 Object-Oriented Programming

Object-oriented systems have gained tremendous industry focus, and consequently, investment. The main difference between traditional systems design and the object-oriented approach can be summarized as follows. Traditionally, systems are designed top-down (analysis, then design, then implementation). A rather rigid barrier separates data from processes. Business entities are represented by one or more relational database tables. Processes are implemented by more or less modular programs that reference the relational tables.

This approach presents some problems and can be unsuitable for the increased complexity and volatility of today's business environments. For example:

- Top-down methodology implies that little or no prototyping is possible. Users will experience the system when coding has been completed. Implementing changes for this environment is costly.
- Changes to business processes are hard to implement. Changing the data representation may impact applications in multiple places. Changing and testing the programs effected is often a complex task.

According to the object-oriented approach, the system models reality as closely as possible. The model is made up of a set of objects that contain data and *behaviors*. These behaviors are operations that can be performed on the data. Objects are meant to represent the business entities that are involved in a specific business domain. A business process is represented by a series of interactions among the various objects. The key concept of object-orientation is the concept of *class*. A class is a template for creating objects of the same kind, such as customers, bank accounts, or employees. When an application needs to create a new customer, it refers to the customer class and creates one by specifying the characteristics of the new customer.

The power of classes resides in the fact that they can be easily extended by using a mechanism called *inheritance*. New classes can be derived by extending existing ones. The new classes inherit all the characteristics of the old ones and are changed to include additional behaviors and data. If the business requires a more complex type of customer, you can capitalize on the simpler existing customer class and create a new, specialized class that incorporates new characteristics.

This approach encourages a cyclic approach to developing systems based on step-wise refinement. Simpler aspects are implemented first, and more complex features can be added later on. This approach is suitable for prototyping. Objects can be provided with a subset of their data and behavior and can still interact to demonstrate the validity of the underlying analysis and design. It is even possible to keep refining the draft model with the help of the end users until requirements are fully met.

As objects encapsulate behaviors and data in a single unit, changes to the business model translate into localized changes to the object model. In most cases, even the interaction among objects resides in specific objects, further enforcing the concept of isolation of changes. The dangerous "ripple effect" across the entire system is much less likely to happen.

Object-oriented technology has encouraged software providers to offer sets of reusable classes that address specific programming issues. Most readers are probably familiar with the concept of graphical user interface class libraries. Most vendors of application development tools integrate such a class library in their development environment. These libraries contain a set of components that can be directly reused, combined, or extended to create graphical interfaces.

1.2.3 Commercial Frameworks

After reviewing various business management systems, it became clear that there are a significant group of capabilities which are common within a particular application domain. These capabilities support the core business activities (processes) performed by any business within this domain.

For instance, you can expect that the degree of overlap among various implementations of General Ledger as they are available on the market can reach a significant amount of the business processes. Therefore, building an application from scratch means that a large percentage of the total investment goes into building components that implement standard procedures rather than into creating functions that represent added value.

These considerations led to the idea of *commercial application frameworks*, which are a set of classes representing business entities that cooperate to implement core business processes. The availability of object-oriented frameworks allows developers of business management systems to rely on a base of existing business processes and business objects. Developers can, therefore, concentrate their efforts on the aspects that are going to differentiate their final implementation from others in the marketplace.

Frameworks are profoundly different from class libraries.

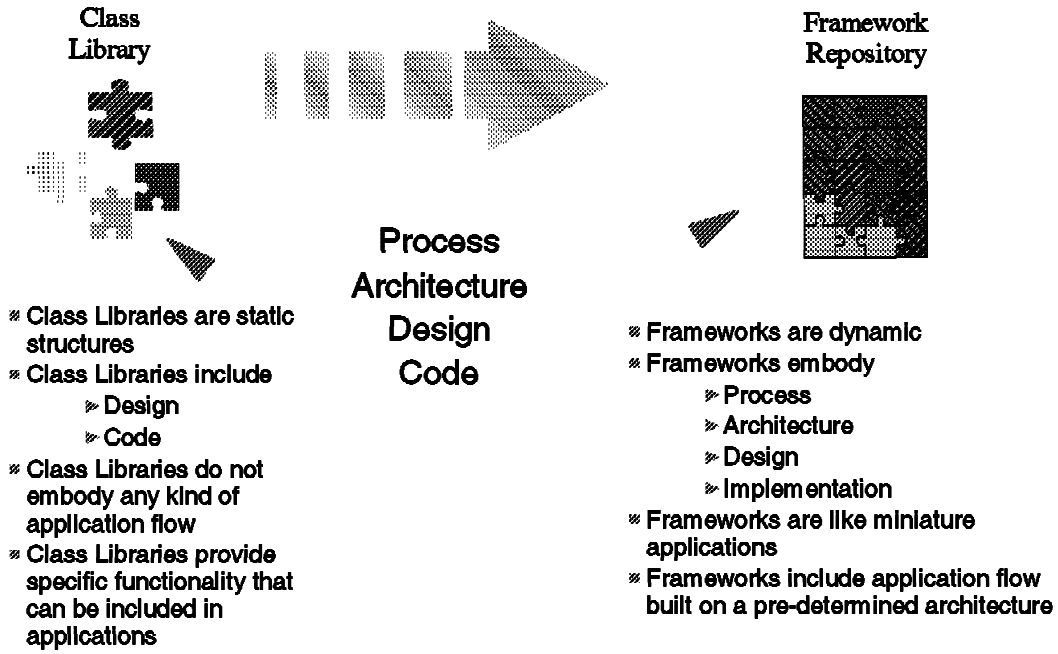


Figure 2. Difference Between Class Library and Framework Repository

As Figure 2 shows, frameworks includes a dynamic aspect that is totally absent in class libraries. In class libraries, the only relationships that you need to be aware of are *static* relationships. For example, when using a GUI class library, you need to know that a pull-down menu inherits from a more general menu class and that a window may contain a pull-down menu and multiple push buttons.

In a framework, classes are also tied together by static relationships just the same as a class library, but the classes are also connected so that they embody a process. A framework contains some application logic that ties the various classes together.

Here is a simple example. A General Ledger framework can provide you with a Bank Account class which you can use to represent bank accounts that your business deals with (internal or external). In this case, the framework already includes the logic that is needed to transform a bank account transaction into the appropriate entries of your General Ledger journals. One of the reusable aspects of such a framework is the Bank Account class, but another important reusable aspect of the framework is the process of creating a bank movement and the way it affects the General Ledger journals. Understanding the dynamic aspects of a framework is essential if you want to be able to take advantage of it.

1.3 The San Francisco Frameworks

San Francisco provides flexible application business frameworks that reduce the complexity, expense, and time-to-market for Independent Software Vendors (ISVs) to build customized multi-platform systems. Using San Francisco, ISVs can produce robust solutions built on top of high-level reusable business frameworks that are Internet-enabled, scalable, and deployable on multiple client/server platforms.

1.3.1 San Francisco Frameworks: Value Proposition

The brief historical perspective we have outlined in the previous sections illustrates the rationale that pushed IBM to invest in the San Francisco project. Software providers and the information technology industry in general have expressed the following needs:

- The ability to build applications starting from tested and robust components. Application development is moving toward using componentry, which plays a key role in lowering cost and improving quality.
- Platform independence
- Distributed and networked computing
- An established architecture for application development

IBM is convinced that commercial frameworks (and in particular, San Francisco Business Process Components) provide the answer to this demand.

The San Francisco value propositions are:

- A proven and tested OO technology base to enable transition to Java technology.
- Reusable common application elements to shorten development cycles
- Lower application development costs for software developers
- Faster development for multiple deployment platforms
- Redirected focus of software developers on business or domain problems instead of programming and technology infrastructure problems.

From an application provider's perspective, San Francisco provides the advantage of making available reusable common business processes that have been implemented in a robust and extensible way. Therefore, developers can concentrate their resources in creating those functions that represent a competitive advantage in the marketplace. San Francisco aims to save about 40% of the overall effort it takes to build an application from scratch.

Platform independence and portability are achieved by the fact that San Francisco is almost 100 percent pure Java. Therefore, it can be supported by a wide range of platforms.

San Francisco also allows business objects to be distributed across a network. The participating systems must be connected through TCP/IP and can run any operating system that San Francisco supports, even in a heterogeneous configuration of multiple platforms in the same network.

San Francisco is structured using a layered architecture that shields developers from the technical implementation of the system. Issues such as distributed

computing, platform independence, interaction among distributed objects, and transaction integrity are transparent to developers.

The book *Design Patterns, Gamma et al.* points out that frameworks form the architecture of your applications. This means that a framework tries not to minimize constraints as far as the functional content of an application built on top of it. The framework shows how software providers are supposed to customize it and complete it into a full-fledged application. However, a few rules lower the risk and the cost of integrating different applications. San Francisco provides more than just a set of architectural recommendations in this area. It also includes a series of classes that implement *design patterns*, which provide an architected way to resolve recurring programming issues. In addition, San Francisco clearly defines *extension points*, which are the classes that software vendors primarily modify, extend, or replace to plug their needed functionality into the frameworks.

From the standpoint of the end users, San Francisco presents a solid technological foundation and a sound design process. Frameworks are also structured in a way that makes it a straight forward process to integrate applications that come from different software providers. Software vendors, especially in the marketplace of small and medium business solutions, tend to excel in specific business domains. Integrating the best of breed of the various domains is a complex and failure-prone process. As we stated, San Francisco intends to lower the efforts and the risks involved in integrating different applications, which gives the customer the opportunity to choose from various software vendors.

1.3.2 The Future of San Francisco

San Francisco is not only about technology. It is also about commercial computing. San Francisco delivers server code that can run on a variety of platforms. San Francisco has delivered a beta version of GUI frameworks aimed at enabling vendors to create user interfaces with a consistent look and feel. Software developers are free to choose other GUI technology to interface with San Francisco business processes and objects.

San Francisco will be rolled out in stages and will keep growing as new frameworks are created by IBM or by Business Partners and ISVs.

Common Business Objects (CBOs) provide the most common business entities (Company, Currency, Business Partner, Natural and Fiscal Calendar, and so on) and some generalized mechanisms that apply to business needs.

San Francisco is available, together with the General Ledger Core Business Process on the Windows NT and AIX platforms. In the near future, we should see an Order Management Core Business Process, a Warehouse Management Core Business Process, and a Ledger Core Business Process (AP/AR). All these Core Business Processes will be available on current platforms and the AS/400.

Figure 3 on page 7 shows an outlook of the multi-layer architecture of San Francisco.

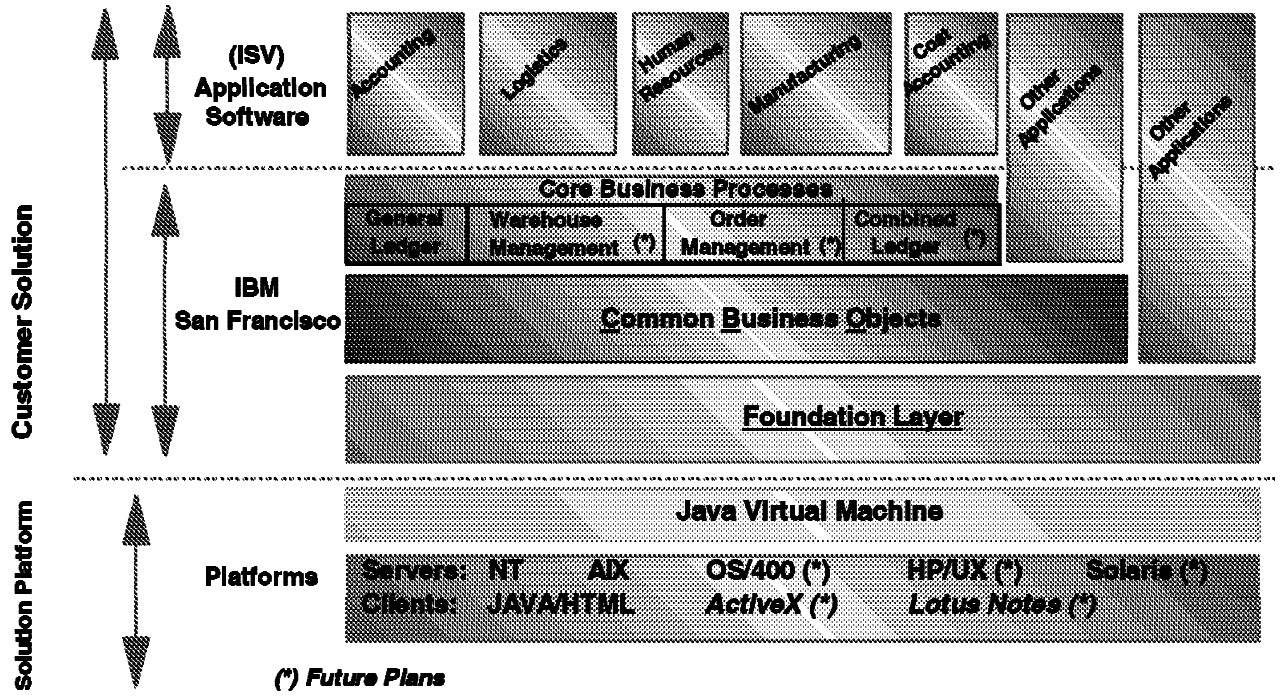


Figure 3. San Francisco: The Outlook

Like any programming language, Java has its idiosyncrasies that can translate to poor performance if you are not aware of them. San Francisco development is dedicated to ensuring that Java performance, and thus San Francisco performance is a top priority.

1.4 Conclusion

San Francisco gives you a set of advantages in moving toward a well-structured object-oriented application development environment. It provides you with assets at the business domain level, at the technological level, and at the level of physically implementing your applications. To benefit from San Francisco, you need to be aware of its architecture, of its contents and of the methodology to be applied when using it. The rest of the book explains each of these topics in more detail.

Chapter 2. San Francisco Architectural Approach

The purpose of this chapter is to explain the San Francisco architecture and how the frameworks provided by San Francisco let software vendors focus on their specific domain areas while taking care of the core concepts and technological issues. San Francisco is a layered architecture that shields developers from the details of the technical implementation of functions such as distributed computing or transaction integrity.

Using San Francisco, software providers can reallocate their resources to the aspects of their business that differentiate their product in the marketplace. San Francisco addresses not only the technological aspects of modern application development, it also provides the *foundation* for creating new applications for a specific industry segment.

It is important to become familiar with the San Francisco architecture to understand the advantages of using San Francisco. This chapter introduces the San Francisco architecture showing its layers and levels of abstraction. It contains the following sections:

- Section 2.1, “San Francisco Layers and Levels of Abstraction” describes, at a high level, the San Francisco layers. For each layer, there is a chapter in this book describing it in more detail.
- Section 2.2, “Platform Independence” on page 16 highlights some of the platform-independent aspects of San Francisco.

2.1 San Francisco Layers and Levels of Abstraction

San Francisco is a multi-layer, shareable, and flexible framework-based system for development of platform-independent, distributed, and object-oriented software solution using both published as well as specialized design patterns. The complete San Francisco system is comprised of three integrated layers:

- Foundation
- Common Business Objects (CBOs)
- Core Business Processes (also known as "Towers")

San Francisco provides a complete technology wrapper for a real-time platform-independent distributed business system. The conceptual San Francisco architecture is illustrated in Figure 4 on page 10.

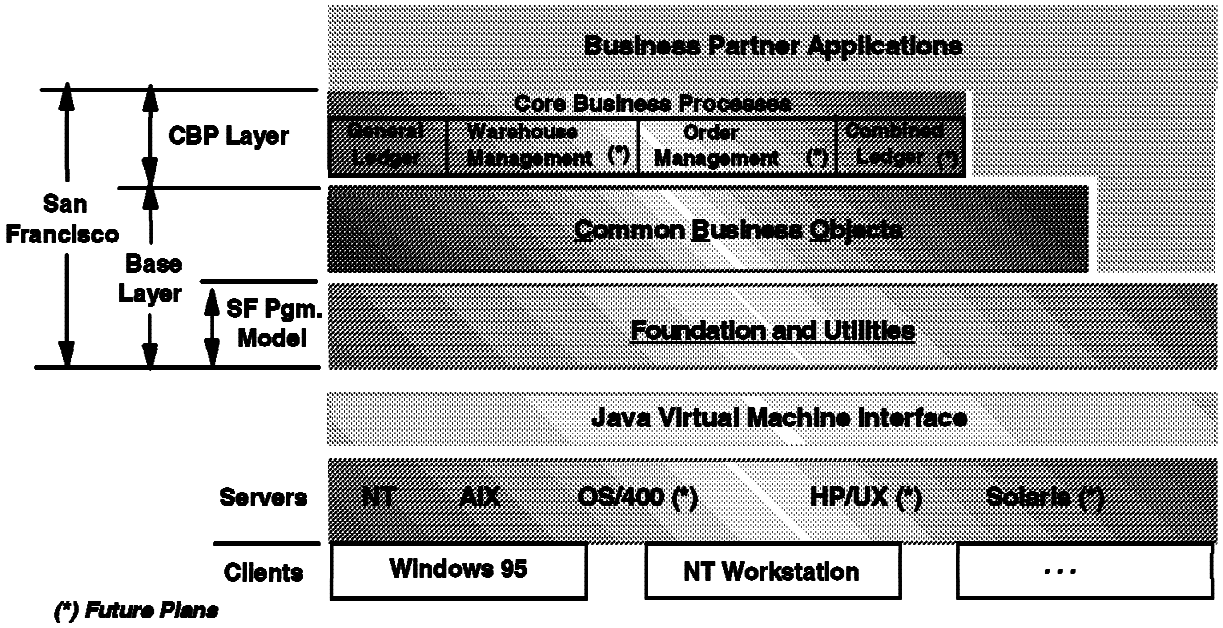


Figure 4. Overview of the San Francisco Architecture

2.1.1 Foundation Layer

Foundation Highlights...

- It is the distributed computing infrastructure for San Francisco underpinnings of San Francisco's middleware.
- It masks technologies from applications by providing common cross-platform, cross-application functions for application development and deployment.
- It includes APIs for distributed computing services such as security and transactions.
- It provides consistent behavior and common methods for administration of applications.
- It is the basis for managing distributable objects in San Francisco
- It promotes an architecture where a distinct object's role exists in the client/server paradigm:
  - Separation of user-interface from business logic
  - Isolation from the underlying mechanisms for persistence
  - Separation of command and selection logic from business data and user interface.
  - Distribution of business processing to allow for optimum performance

The Foundation layer is the core technological layer of San Francisco, and provides the fundamental services such as distributed object creation, synchronization, persistence, and a consistent application development model. It encapsulates the technological aspects of cross-platform distributed object management and provides an easy-to-use API (Application Programming Interface). It also includes core functionality to support security, distributed

transaction processing, and forms a middleware between a client application and server.

The Foundation provides you with the distributed architecture through its set of Foundation classes. Using those Foundation classes provides a means to set up a distributed application. Because the Foundation masks the technology differences between different platforms, it supports reaching true portability and multi-platform environments. The Foundation also provides classes that form an interface to the different object services.

By using the Foundation and complying with the San Francisco programming model, you can implement several OO paradigms.

Note: Refer to reference 5 on page 117 in B.3, "Other Publications" on page 117 for detailed information about "Distributed OO paradigms."

The Foundation allows you to divide processing between several systems, which gives you a control of the performance and scalability of your application.

For further details on the Foundation layer, refer to Chapter 3, "San Francisco Foundation" on page 23.

2.1.1.1 Foundation: The Control Tower

Figure 5 shows the Foundation as the control tower of San Francisco; most of your requests are accepted and routed by the Foundation layer. The Foundation is your interface to the different services. It deals with the persistent data stores and makes it easy to switch between different methods of persistence without having to recode your application or modify your business objects. Changes are made through the configuration data, which can be updated at run time.

The Foundation layer is also the interface to the *Factory mechanism* that plays an important role in managing distributed objects. We discuss the Factory in more detail in Chapter 3, "San Francisco Foundation" on page 23, but remember that all object maintenance (create, delete, copy, and so on) must be done through the Factory.

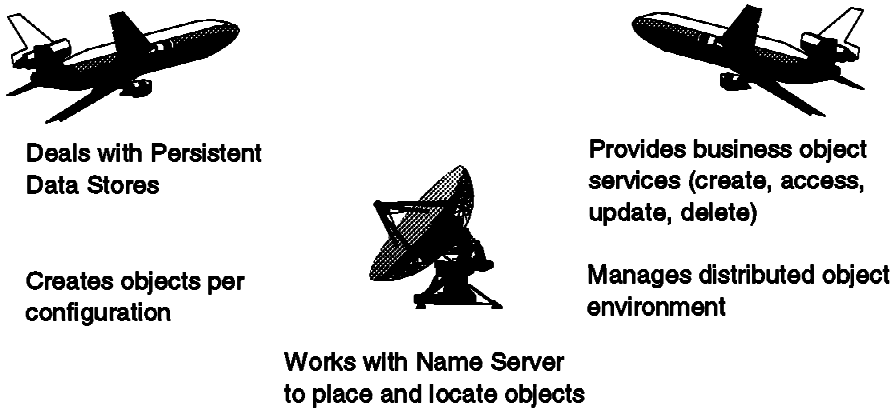


Figure 5. San Francisco Foundation Layer: The Control Tower

2.1.2 Common Business Objects (CBOs) Layer

CBO Highlights...

- Contains functions and objects commonly needed across business domains:
  - It represents a variety of constructs common to most business environments
  - The Core Business Processes use CBO classes extensively
  - It includes classes of various complexity
- Interfaces to the financial Core Business Processes:
  - Posts to General Ledger

Common Business Objects (CBOs) is built on top of the Foundation layer. This layer, together with the Foundation and Utilities, form the Base as shown in Figure 4 on page 10. The Common Business Objects layer consists of objects that perform functions commonly needed across business domains. They fall into one of the three categories described in Section 2.1.2.1, "CBO: A Categorization."

2.1.2.1 CBO: A Categorization

Figure 6 shows a categorization of Common Business Objects providing a means of thinking about them.

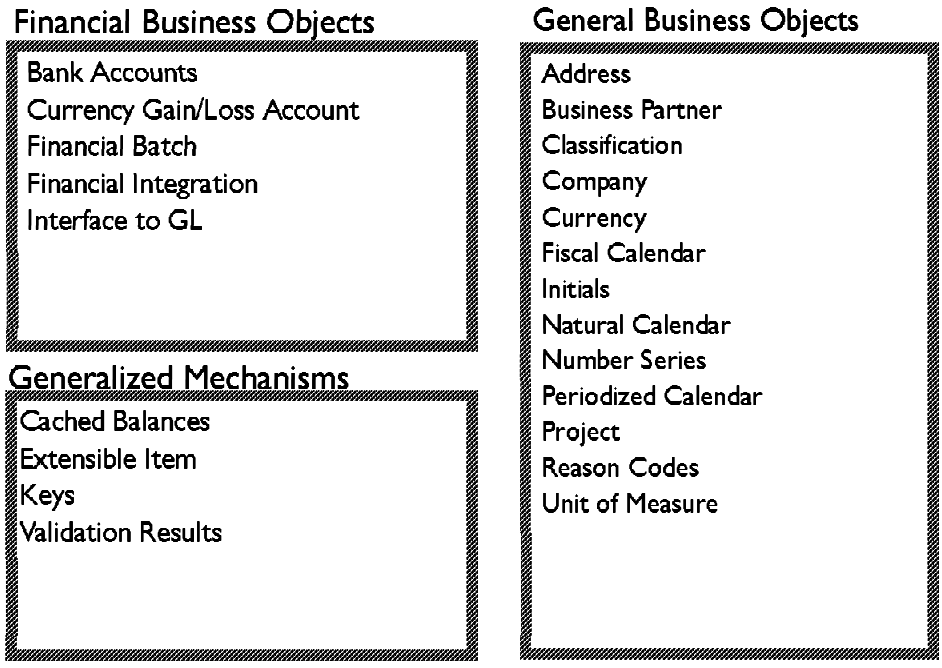


Figure 6. CBO: A Categorization

General Business Objects: These categories include classes that most commercial applications have to deal with such as a Business Partner (Customer), Address, Currency, and so on. You can also see that the Company class is reported here. A Company has special importance in San Francisco. Usually other business objects belong to a Company object.

Using the Common Business Objects ensures that San Francisco-based applications from different vendors can cooperate. Examples of CBOs include Company, Currency and Address.

Financial Business Objects: All businesses have to deal with money and virtually all businesses need to deal with Bank Accounts, Currency Gain/Loss Accounts, and so on.

Generalized Mechanisms: We have grouped here some mechanisms that are common to many aspects of business in general. Cached Balances, for instance, are totals that can be maintained over inventories or other summarizable containers of information.

2.1.2.2 An Example of CBO: Company

Example: Company

- A fundamental class:
  - Allows for a hierarchy of Companies
  - The Enterprise is the root of the organization
  - The Company Controller manages the hierarchy
- Most objects in an application will typically belong to the Company

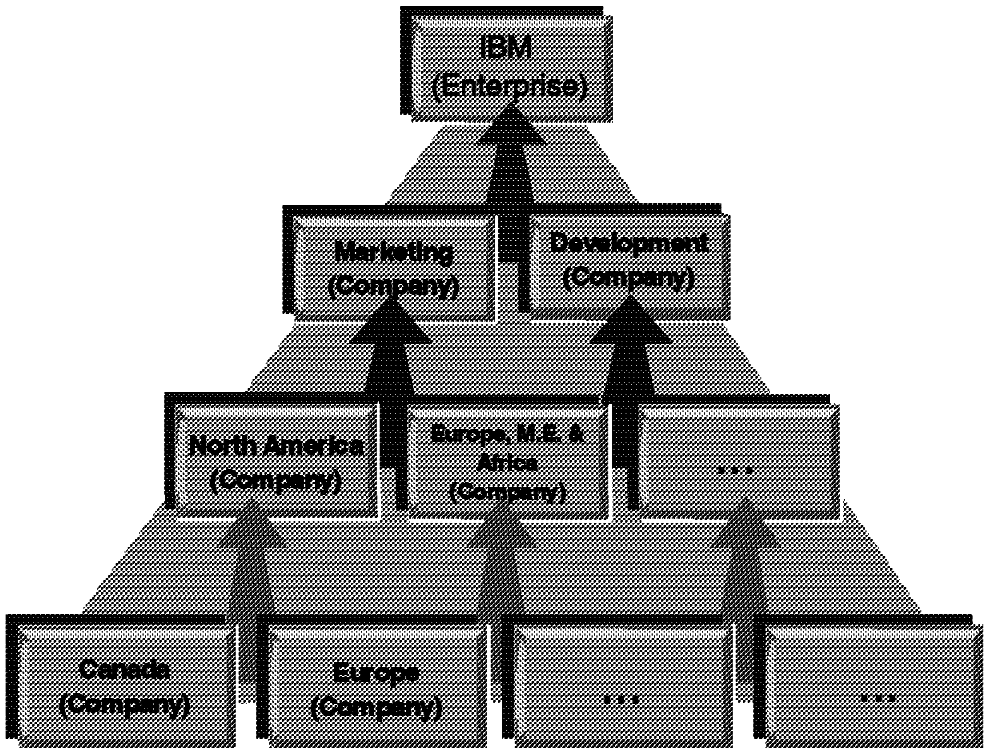


Figure 7. An Example of CBO: A Company Hierarchy

Figure 7 shows an example of a hierarchy of the Company category. The diagram shows that CBO includes an Enterprise class, which inherits from Company. Thus, when we say Company, this includes the Enterprise. The only difference is that the Enterprise is the root of a multi-company organization. It is also the only Company of a single-company organization. In a multi-company

organization, the Company Controller keeps track of all of the companies in the hierarchy and the actual hierarchy is managed by the relationships between the companies. The Company is meant to represent the organization of the business that runs the application. Customers and suppliers of this business can be represented by using the Business Partner class in CBO.

Despite of the simple appearance of this diagram, the Company is a complex object. The Company class inherits from Describable Dynamic Entity. A Dynamic Entity is a subclass of Entity that implements the design pattern of the Property Container. This design pattern is discussed in more detail later in Chapter 6, “San Francisco Patterns” on page 55. Here it is sufficient to mention that the Property Container allows applications to extend the attributes and the behavior of an object by dynamically adding properties to it. This design pattern allows you to change the characteristics of objects without creating new subclasses.

The Company allows you to define a wide variety of properties. Actually, the Company is supposed to contain a relatively large number of properties that San Francisco applications always refer to during execution. Most of these properties are Controllers that keep track of key characteristics of the business such as which currencies are used in the business, what kind of exchange rate policies are used, and which banks the business deals with. For this reason, we can affirm that most of the CBOs *need* the Company to be able to work. It is virtually impossible to create an application using San Francisco CBOs without creating an instance of the Company.

2.1.3 Core Business Processes Layer

CBP Highlights...

- Core Business Processes contain classes that are particular to a business domain.
- General Ledger (GL) is currently available
- In the near future, the Core Business Processes will include:
  - Warehouse Management
  - Order Management
  - Combined Ledger

Core Business Processes is the top layer of San Francisco and provides an extensible connection mechanism between the basic structural objects (including CBOs) and fundamental (or default) behavior. This layer is, in other words, a layer providing generic business processes. The rest of the application provides for domain or application specific processes (for example, special GL journals, lead time calculation policies, and so on) is integrated with this layer through the framework extension mechanisms. For example, San Francisco contains the General Ledger Core Business Process, which includes the architecture, design, and default logic to build a General Ledger application. The developer, instead of building it from scratch, needs only to enhance and extend this layer to build a customized General Ledger application.

The application domain frameworks contain the processes that are specific for a certain domain and cannot be used for other domains.

See Figure 8 on page 15 for an example of some functions provided by each Core Business Process. Also refer to Chapter 5, “San Francisco Core Business Processes” on page 49 for a complete description about the General Ledger (GL) Core Business Process.

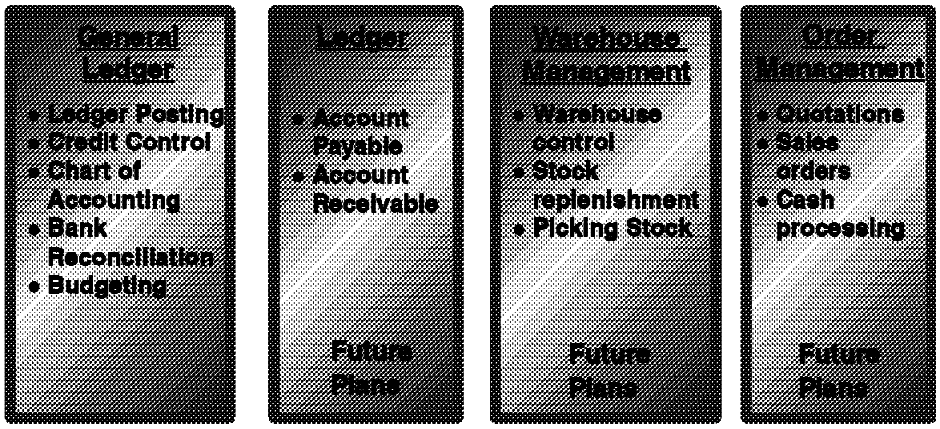


Figure 8. Core Business Processes Examples

In the San Francisco architecture, these Core Business Processes are almost applications on their own. If you build a user interface on top of the model included in the application domain business objects, you have a simple application. It is up to the application programmer to extend this framework where needed and replace default policies that implement the most common behavior. It will also be possible to adapt the framework to implement local regulations or customer requested features. The architects of the San Francisco project have designed special classes into the frameworks that encapsulate business logic that is likely to change. These classes are identified in the documentation as *extension points*. Extension points typically follow one of the San Francisco patterns so their use and behavior can be well understood. For further details on Core Business Processes, refer to Chapter 5, “San Francisco Core Business Processes” on page 49.

2.1.4 Commercial Applications (Business Domain)

The commercial applications (Business Partner application) is the actual, real-life application developed using San Francisco-supplied core functionality as well as user-provided customized functionality.

The software built over the San Francisco layers is created by application developers. They extend the San Francisco Core Business Processes. In all cases, commercial applications will be created using Java.

Since San Francisco is a new concept, it is not yet clear how much of the total effort to create an application will be provided. For now, IBM is suggesting a conservative 40%. In reality, it will never be possible to measure accurately since those that use this process to develop applications will do things differently than they would in a traditional programming environment. As developers create their own unique applications, they can choose how much of San Francisco they need or want to use within their solutions.

For further information on the design and development of commercial applications, refer to Chapter 8, “San Francisco Application Development Methodology” on page 81.

2.2 Platform Independence

This section covers the fact that the *Java Virtual Machine* will insulate San Francisco from the operating system. There are, however, other additional things such as the *name configuration* and the *containers*, which make it possible to change, for example, from one server to another or to switch between data stores. Additionally, we also describe the security model, which isolates the application from the operating system security.

Platform independence is one of the salient features of San Francisco. By platform independence, we mean not only portability across different operating systems, but also independence of used data storage management system and communication protocols and of physical configurations. This section highlights some of the platform-independent aspects of San Francisco.

2.2.1 Operating System Independence

Multiple operating system support is one of the primary targets of San Francisco. Several server and client platforms are supported, or planned to be supported; see Section 2.2.1.1, “Operating System Support.” This is achieved by providing a layer of indirection between San Francisco applications and the underlying operating system (for more details, see Section 2.2.1.2, “Java - Platform and Communications Independence” on page 17). Additionally, San Francisco provides some of the high-level services usually provided by the operating system to further shield the application writer and the administrator from operating system specifics (for more details, see Section 2.2.1.3, “High-Level Services in San Francisco” on page 19).

2.2.1.1 Operating System Support

San Francisco is planned to eventually run wherever an appropriate Java Virtual Machine is provided (see Section 2.2.1.2, “Java - Platform and Communications Independence” on page 17). Figure 9 illustrates the San Francisco vision.

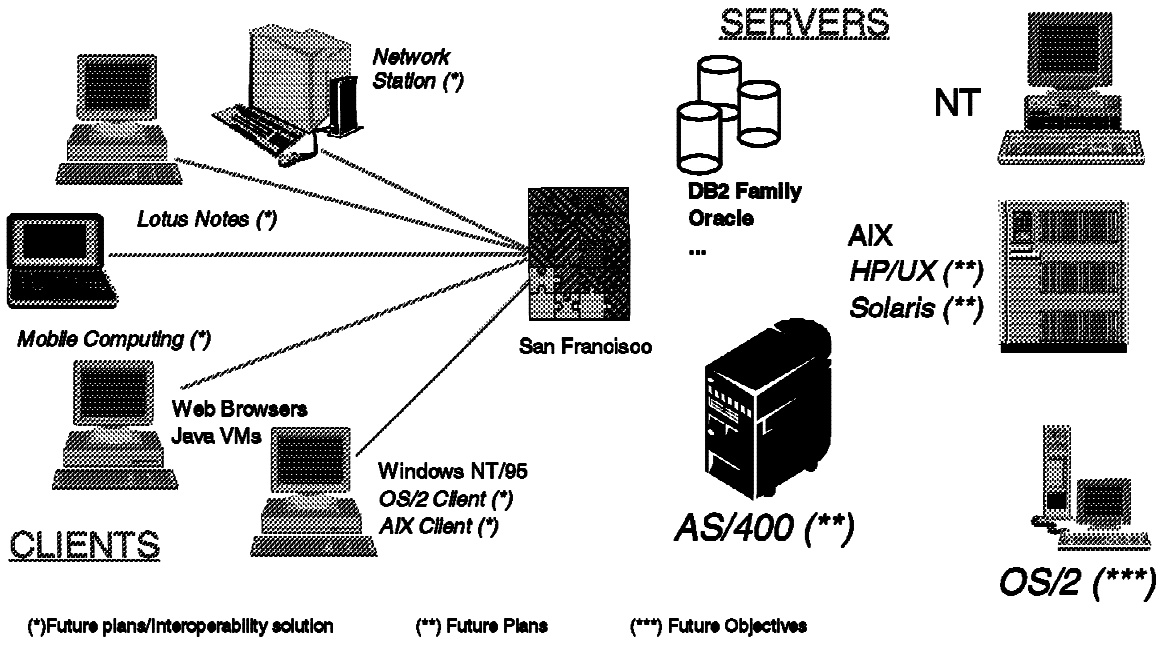


Figure 9. Operating System Support in San Francisco

Currently, Windows NT and AIX are the first *server* operating systems running San Francisco as of July 1997. Meanwhile, the AS/400 system is expected to play a major role as a San Francisco server in the near future.

As far as clients are concerned, the objective is to enable any client providing a Java run time environment to run San Francisco. The currently available clients are Windows 95, Windows NT, and Java-enabled Internet browsers.

2.2.1.2 Java - Platform and Communications Independence

The objective of operating system independence has been one of the compelling reasons for using Java to develop the frameworks. First, we briefly introduce Java, and then we describe what makes Java a good candidate for developing platform-independent applications.

Java is a programming environment similar to C++ but much easier to use. It has been created by Sun Microsystems and released through the Internet in late 1995 originally for sprucing-up Web pages through *applets* ("light weight applications"). Java has been met with an extremely fast adoption rate.

a Definition of Java

"A simple, object-oriented, distributed, interpreted, robust, secure, architecture neutral, portable, high-performance, multi-threaded, and dynamic language."

(Source: *The Java Language: A White Paper*, by Sun Microsystems, `http://www.javasoft.com/nav/read/whitepapers.html`).

Java is an interpreted language suitable for building Web page applets or full stand-alone applications. Figure 10 illustrates steps taken to run a Java application.

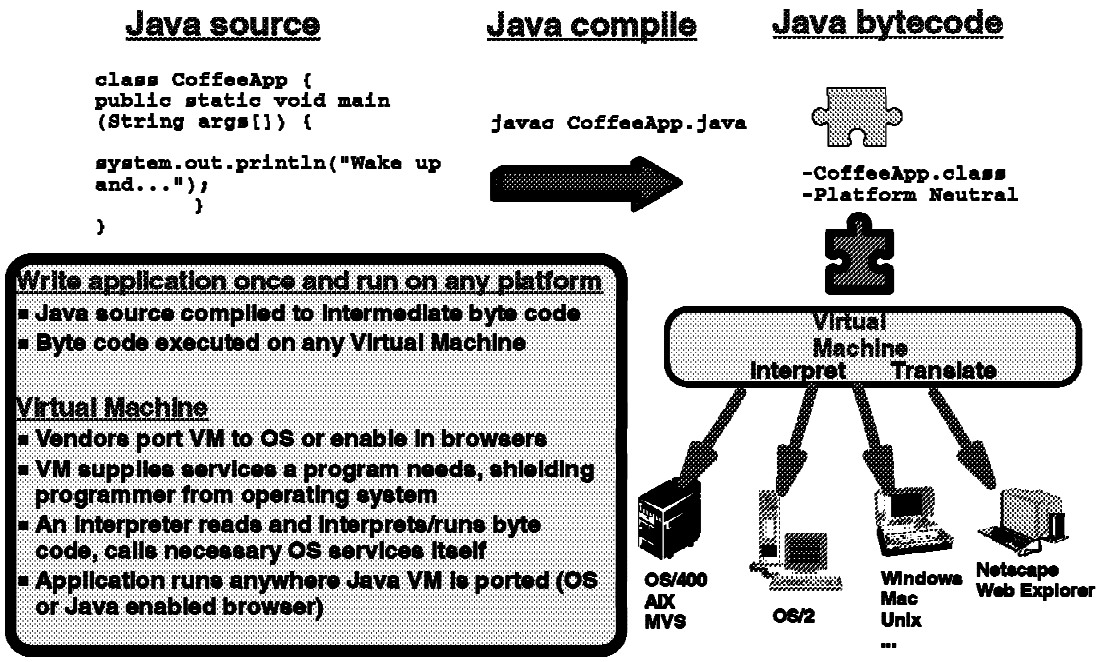


Figure 10. Running a Java Application

1. The Java source code is compiled to what is called the common intermediate code (also known as *byte-code*). Intermediate code is executed on a Java *Virtual Machine*.
2. A Java Virtual Machine (VM) interprets (that is, runs) the Java byte-code. It provides the services needed to run a program. While doing so, it calls the necessary operating system services itself, shielding the Java programmer from the operating system used.

A virtual machine is ported by vendors to operating systems or enabled in Internet browsers. Once a VM is ported to an operating system or a browser, Java byte-code can be interpreted there *without* change.

Java also includes a set of standard features that support *distributed computing*. For example, the following features are introduced by successive levels of Java Development Tool Kits (also known as JDKs):

JDK 1.0 Introduced classes implementing sockets and URLs as part of the `Java.net` package.

JDK 1.1 Introduced implementation of Java Database Connectivity (JDBC) and Remote Method Invocation (RMI).

JDK 1.2 Plans to incorporate a bridge between RMI and CORBA's IIOP (Internet Inter-Orb Protocol).

The RMI is of special interest for San Francisco as it is used for handling interactions between client and server objects. More importantly, users of RMI are shielded from the underlying *communication protocol*. Figure 11 illustrates the layered architecture of RMI and how it provides a layer of indirection (that is, the *Remote Reference Layer*) between its users and the transport layer implementing the communication protocol. This enables San Francisco to take advantage of future technologies *without* affecting any existing San Francisco applications.

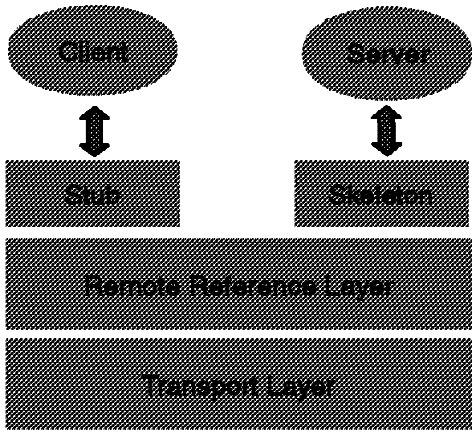


Figure 11. Remote Method Invocation

In addition to portability, Java also offers many attractive features, some of which are:

Robustness

- *Pointers* are not used in Java language; hence, Java eliminates problems related to pointer manipulation such as accidental access to another object in the same address space.
- *Automatic garbage collection* is a feature provided by Java to remove objects no longer in use, automatically freeing memory occupied by them.
- Compiler enforces that exceptions are properly handled in addition to run time support for exceptions provided by the VM.

Ease of Use

- The reduced complexity of Java (for example, simpler syntax, no pointers, no operator overloading, no templates, and so on) makes the language much easier to use.
- The standard features of the language (for example, automatic garbage collection) reduces program complexity, and frees programmers to focus on the solution.

Broad Market Acceptance

- Java has witnessed an extremely fast adoption rate since announcement by Sun in fourth quarter 1995.
- Java has gained acceptance as a Web development as well as non-Web application development language.
- Emphasizing this wide acceptance is the numerous Integrated Development Environments (IDEs) available (about 30 in the fall of 1997 and counting).

2.2.1.3 High-Level Services in San Francisco

San Francisco offers a set of high-level services to both programmers and users. Some of these services are San Francisco unique (for example, Configuration Utility), while some are the San Francisco version of operating system services (for example, Security). These services include:

Security: Allows definition of users and user groups, user profile creation and maintenance, authority and access rights assignment, and customization of password rules. It is managed through a graphical user interface, and it provides a set of APIs for applications to interact with the San Francisco security model. For more details, see Section 7.2, “Security Configuration” on page 72.

Printing Utility: Enables you to quickly create your report layouts using a WYSIWYG (What You See Is What You Get) editor. These layouts can be later used by applications to format business data for printing using a Print Formatter class in your application. Existing class definitions and actual objects can be referenced while creating the report layouts to facilitate formatting. This utility generates reports in PostScript; however, several different print output types will be supported (for example, Advanced Function Printing - AFP). For more details, see Section 7.4, “Print Utility” on page 75.

Additional services are also provided by San Francisco; these are covered in Chapter 7, “San Francisco Utilities” on page 65.

2.2.2 Data Store Independence

One of the key benefits of San Francisco architecture is data store independence. This means that San Francisco applications are not only uncoupled from the storage technique used, but also they are not aware of the physical location of the data store being used. Thus, the implementers of San Francisco applications are free to switch to the latest data storage technology, support multiple storage techniques in the *same* application, and apply their applications on a variety of system configurations *without* even recompiling their code. These concepts are highlighted in the following paragraphs.

Data in San Francisco is encapsulated within persistent business objects. San Francisco defines storage pools for its persistent objects called *containers*. Containers are named data stores that are configured at run time through the Configuration utility in the Foundation. There are different types of containers that are currently supported by San Francisco; these are:

Posix These containers are basically flat (stream) files in Posix directories such as a Windows directory or an AS/400 IFS directory. Every object is contained in a separate file. This option is suitable for testing purposes only and not recommended for production.

ODBC These containers store objects as rows in relational DB tables. DB2 family databases and Oracle are currently supported with an objective to support other popular RDBMS. There are two alternatives when defining these containers: either you direct San Francisco to automatically define the tables needed to store the persistent objects, or you manually control the definition of these tables (for example, to integrate with *existing* legacy databases). These alternatives are supported by a San Francisco utility called the *Schema Mapper*.

The power of San Francisco architecture arises from its layered style. All object creation, storage, retrieval, and deletion is done through the Foundation. A special object in the Foundation, called the Base Factory, offers architecture-neutral interfaces for these services. The Foundation services provides the definition of a default container for each San Francisco class as part of the run time configuration. An application need not know anything about containers. For example, when creating an object, an application may either request creation in the default container, or in the same container as that of another object passing in the other object reference. The Base Factory determines the default container, or the container of the passed-in reference using the Foundation's Naming services, and handles the specific interactions appropriate to the type of container used. Thus, the Base Factory encapsulates all needed functionality of locating and interacting with the different types of containers.

Once a storage technique is supported by San Francisco, it is readily available to all existing San Francisco applications. This gives rise to many advantages, for example:

- Existing San Francisco applications can immediately take advantage of new storage techniques *without* recompiling.
- Multiple storage techniques may be used in the same application enabling optimum usage of deployment platforms (for example, in a network of AS/400 systems and RS/6000 systems running AIX, an application may concurrently use DB2 containers on an AIX system and SLS containers, *when available*,

on the AS/400 system). Multiple storage types (containers) per system are also allowed.

- An administrator is free to distribute the containers with regards to the network configuration *without* affecting the applications. For example, an administrator may move a container to a different server for load balancing (however, special care *must* be taken for moving the contents of the container).

Chapter 3. San Francisco Foundation

This chapter describes the contents and the functions of the San Francisco Foundation layer. This chapter provides an overview, not technical details or specific programming issues. We recommend you refer to the *San Francisco Programmer's Guide* for a comprehensive description of those issues.

This chapter contains the following sections:

- Section 3.1, “Purpose of the Foundation Layer” explains the purpose of the Foundation layer as part of the San Francisco architecture.
- Section 3.2, “Foundation Classes” contains a detailed view of the fundamental Java classes included in the Foundation layer.
- Section 3.3, “Foundation Library Classes” on page 35 gives a quick overview of the Foundation Library Classes contents.
- Section 3.4, “The San Francisco Programming Model” on page 37 lists and briefly explains the San Francisco programming model, which defines a set of rules programmers should follow to be complaint with a certain standard or to correctly use certain services.

3.1 Purpose of the Foundation Layer

Object technology raises several formidable challenges when objects move into a distributed environment. To provide real value to application developers, platform independence is a primary need. Satisfying this need becomes mandatory in a distributed world. Once objects are distributed, a program must be able to refer to instances that do not reside in the same address space as the program itself. These instances may be created on the same system as the client program or on some remote system running a different operating system on a completely different hardware context. Nevertheless, we need to ensure that the client program can access those objects with total transparency as if they were created locally. Method calls, parameters, and return values must flow transparently between the client program and the distributed objects.

Persistence is also a major challenge. We need to create objects that survive beyond the life of a single application session. Distributed and persistent objects are then visible by multiple users at the same time. Therefore, the system must provide mechanisms to lock objects and transaction services to ensure data consistency. Objects also need to be uniquely identified within the entire network and users must be able to assign user-defined names to their objects.

Java and the JDK address only a subset of these aspects. The San Francisco Foundation layer provides the services and the facilities that are needed for programmers to work with distributed objects.

3.2 Foundation Classes

The Foundation layer includes a series of fundamental Java classes that represent the root of the inheritance tree for the San Francisco business objects. Programmers must inherit from these classes if they want to take advantage of the services that the Foundation provides.

Figure 12 on page 24 shows the inheritance hierarchy for the most commonly used Foundation classes.

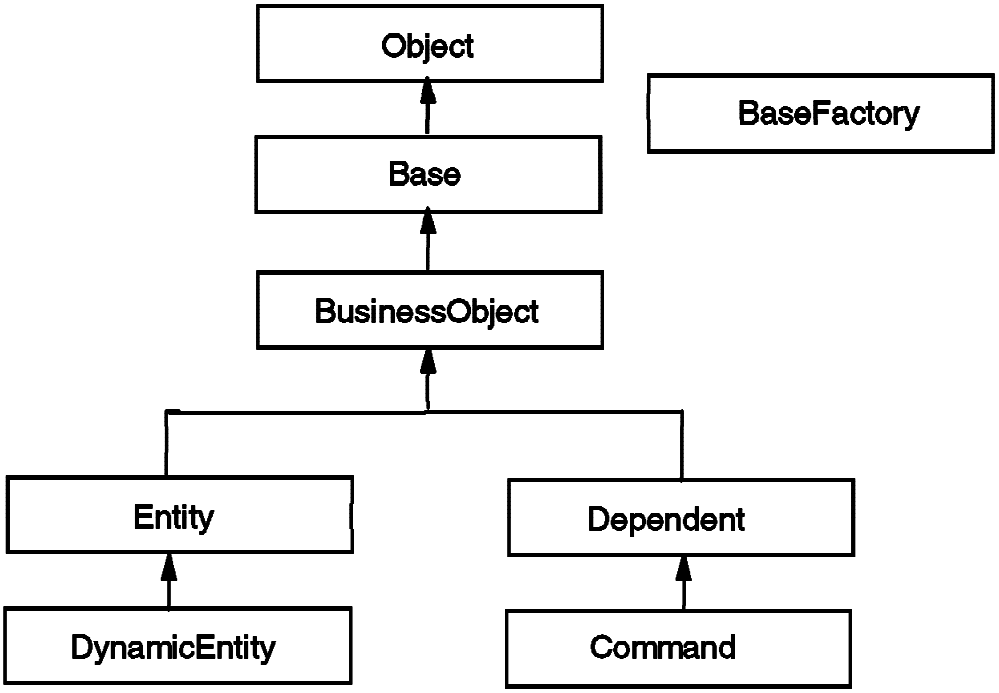


Figure 12. The Foundation Classes

3.2.1 Entity

Programmers can extend the Entity class whenever they need to create business objects that are inherently persistent, that can be shared among multiple processes, and that can participate in a transaction.

Although transient instance of Entity can also be created upon request, in the vast majority of the cases this class is used to create persistent instances. Programmers only need to request the creation of an instance to make it persistent. Programmers can also retrieve existing instances once they have been created and committed to permanent storage. No additional programming effort is required to make an instance persistent and, if the logic of the application requires it, programmers can be totally unaware of the physical data store that eventually contains the instances.

Once an instance of a persistent business object is instantiated (either created from scratch or retrieved from the data store), it behaves the same as any regular Java object. Client programs can call methods on those instances even if they live in separate address spaces, possibly on a remote system.

In accordance with the San Francisco programming model, you do not create instances of San Francisco business objects by using the new Java operator. In order to take advantage of the San Francisco Foundation, the instance you create does not benefit from the services of the Foundation layer. Whenever you need to create an instance, use the methods provided by a special San Francisco object called the Base Factory, which provides a level of indirection for run time choice between local/remote, server location, etc. The Base Factory is

the major mediator between programmers and the services provided by the Foundation. The role of the Base Factory is described in Section 3.2.7, “The Base Factory” on page 31. At this point, it is sufficient for you to understand that programmers need to ask the Base Factory to create a new instance or to retrieve an existing one.

The Base Factory returns a normal Java reference as a result of the programmer's request. Once the Java reference is obtained, you can use it as if you got it from the new operator. You only need to start a transaction before you request the creation or the retrieval of a persistent business object. You also need to keep in mind that as soon as the transaction is concluded by a commit or a rollback operation, the Java reference that you obtained is no longer usable. If you need to reference the same instance again, reissue the request to the Base Factory.

Another interesting consideration of the Base Factory is regarding objects deletion. A major advantage brought by Java to the programmers' community is automatic garbage collection. Unreferenced Java objects automatically cease to exist and the storage they occupied is made available for use. This mechanism in San Francisco only applies to transient instances of business objects. Persistent instances are not garbage collected. It is up to the applications to explicitly ask the Base Factory for the deletion of a persistent instance. Also see Section 3.2.4, “Ownership” on page 28 for more details on this aspect.

More details about this issue are in Section 3.2.8, “Command” on page 31.

3.2.1.1 Home Mode versus Local Mode

At the core of the San Francisco Foundation, we find the mechanisms that enable objects to be distributed across a network. As we mentioned in Section 3.1, “Purpose of the Foundation Layer” on page 23, one of the requirements of every distributed object architecture is to enable a client program to create instances of objects that may live in separate address spaces. These instances are then used as if they are local to the requester, in total transparency.

The San Francisco Foundation introduces an additional degree of freedom: you can request to access distributed objects in home mode or in local mode.

When you choose home mode, the San Francisco Foundation dynamically and automatically creates a proxy object in the address space of the requester. The proxy *looks the same as* the real instance; that is, it has the same interface. The client transparently invokes methods on the proxy and the proxy routes the method calls to the real instance. This mechanism relies on the Java RMI (Remote Method Invocation) software layer.

If local mode is chosen, the Foundation returns a copy of the real instance to the requester program. Depending on the lock mode you selected, the Foundation may also automatically reflect all changes back onto the real object at commit time. See Section 3.2.5, “Locking and Commitment Control” on page 29 for more details on the lock modes.

The choice between home and local mode can be made by the application at run time or by using the Configuration utility at installation time. This choice does not impact the programming logic and is administrative. Whether an object is accessed in home or local mode is totally transparent to programmers. You can obtain a reference to the instance and then use it as you do with any regular Java object.

There is an important performance trade-off that should guide you in choosing between home and local access mode. Accessing an object through a proxy is convenient; on the performance side, you must remember that every method call translates into a remote call and has to penetrate the RMI and TCP/IP stacks. On the other hand, if an object is accessed in local mode, every method call is local and, therefore, more efficient. However, in this case, a copy is taken and if the object encapsulates a large amount of data, that may impact performance. Home mode is ideal when executing a limited number of method calls on a large object, whereas local mode works the best when executing a large number of method calls on a small instance. Refer to Section 3.2.8, “Command” on page 31 to understand how Commands can also play an important role in tuning San Francisco applications.

3.2.2 Handle

As we pointed in Section 3.1, “Purpose of the Foundation Layer” on page 23, persistent and shareable objects can be accessed from outside the process that has created them. In a distributed environment, persistent objects need to be accessible to processes that may run on different systems, possibly on different platforms. Persistent and distributed objects must, therefore, be uniquely identified in the distributed environment they belong to.

San Francisco automatically generates an object identifier for persistent objects at instance creation time. This identifier is guaranteed to be unique within the Logical San Francisco Network, which is the group of systems and processes encompassed by a certain San Francisco installation.

This unique identifier is then encapsulated into a special object called the Handle for that persistent object. Only persistent objects (Entities) have a Handle. Handles are extremely important in a San Francisco application because they allow objects to store a reference to other persistent objects and client programs to request the retrieval of existing instances.

If a San Francisco business object has a persistent object among its attributes (for instance, an invoice having a customer), it only needs to store the Handle to the contained object.

Handles can also be used to retrieve a persistent instance. The Base Factory can locate and *materialize* the instance based on the information contained in the Handle. This is the preferred way for retrieving instances in San Francisco. You also have the option of giving persistent objects a name (*user alias*) and to retrieve them based on this information. However, this technique should be only applied to a limited number of top-level objects. Assigning, maintaining, and using user aliases is not as efficient as using Handles.

3.2.3 Persistence

As we mentioned, San Francisco objects are made persistent automatically and transparently. Programmers do not have to worry about how and where objects are written to permanent storage.

San Francisco supports various ways for making persistence happen. You can store your objects in a relational database or you can choose to store objects in a Posix data store (stream files in a Windows or Unix directory). On the AS/400 system, San Francisco developers are planning to introduce a rather unique

form of persistence, often called single-level store persistence model. In this section, we provide a quick overview of the various forms of persistence.

It is important for you to understand that choosing the persistence model is a purely administrative task that can be performed without influencing the application coding. You can configure your persistent data stores with an administrative utility, which we briefly discuss in Chapter 7, “San Francisco Utilities” on page 65. If you later need to modify your choices in terms of persistent stores, you can do it without modifying a single line of code.

This extremely powerful feature allows you to configure the same application in completely different ways as far as persistence is concerned. You may install an application for a small customer who only uses a single database server, and later you may install the same application at a larger customer's site where multiple distributed databases are used. In the first case, you may choose to concentrate all the persistent data on a single database; in the latter case, you can distribute objects across multiple data stores. All this does not require a single change to the application code.

Figure 13 shows that persistent objects with San Francisco can be stored in several different ways, which are referred to as container types.

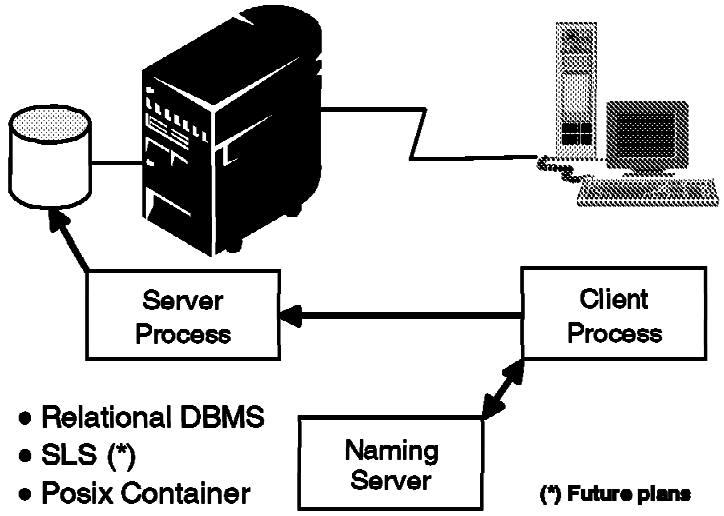


Figure 13. San Francisco Persistence: Several Options

3.2.3.1 Posix Persistence

If you choose this option, data contained in business objects is written by the Foundation into stream files. Every instance is contained in a separate stream file and for every class, a new directory is created.

All the services of the Foundation are supported also with this form of persistence. A limitation you must be aware of, though, is that only single-phase commitment control is supported.

However, even though Posix persistence works fine, consider using it only for demonstration or prototyping purposes. The level of performance, robustness, and of integration with legacy databases guaranteed by Posix persistence is probably unsatisfactory for the needs of most application environments.

3.2.3.2 RDB Persistence

San Francisco makes objects persistent in relational database tables through OBDC.

At present, San Francisco supports the following database management systems for persistence purposes:

- DB2 for Windows NT
- Oracle on Windows NT
- DB2 Common Server on AIX

There are two ways you can instruct San Francisco to make objects persistent in relational tables. You can just turn on ODBC persistence using the Configuration utility (see Section 7.1.3, “Container Configuration” on page 69) and let the Foundation create the relational tables for you using the Default Schema Mapper. In this way, the first time an instance of a certain class is created, San Francisco creates the corresponding table using a default mapping between the data types of the attributes and the relational database columns. The default mapping might not be totally satisfactory if integration with legacy and performance is essential. For example, every String attribute is mapped to a variable-length character field, which might not be efficient; in addition, for every new class, the Foundation generates a separate table.

A second option consists of using the Extended Schema Mapper tool. This tool allows you to perform a more refined mapping between objects and columns. The tool allows for instance to map an Invoice Number field to a 10-character, fixed-length data type in your database and to map the unique object identifier generated by San Francisco to a user-defined primary key. The Extended Schema Mapper is, therefore, an essential element for ensuring a good degree of coexistence between San Francisco and legacy data. We discuss the potential of the tool in Chapter 7, “San Francisco Utilities” on page 65.

Relational database persistence also allows San Francisco to exploit two-phase commitment control using the XA/Open standard, allowing objects to be scattered across multiple database servers.

3.2.4 Ownership

We have mentioned that when a business object contains another object, it stores the Handle of the contained object as a reference to it. An invoice may bear a customer Handle among its attributes as a reference to the customer for which it was issued. It may also store the Handle of a chase letter object if the customer is not solvent. What happens to the customer and to the chase letter if the invoice is deleted? You probably expect that the chase letter should be removed as well since it does not make much sense without the corresponding invoice. You also expect that the customer will not disappear as a result of deleting an invoice. This scenario is a good example to illustrate the concept of Ownership. If an Entity owns another Entity, the life cycle of the owned Entity is tied to the life cycle of the owner. Whenever the owning object is deleted, the owned objects are deleted as well. This is the case of the invoice and the chase letter. The invoice owns the chase letter.

Objects may also be associated by relationships that express pure containment. In this case, deleting the containing object does not trigger the deletion of the contained object such as in the invoice-customer relationship.

This concept and its consequences to the San Francisco application design are of extreme importance. The San Francisco programming model (discussed in Section 3.4, “The San Francisco Programming Model” on page 37) defines, among other things, the rules you need to follow to establish and manage these relationships among business objects. It is absolutely imperative that you follow these rules when you develop San Francisco applications to ensure data consistency.

3.2.5 Locking and Commitment Control

The discussion about Entities brings about the subject of transaction integrity and locking support. The San Francisco Foundation is responsible for granting these services to programmers.

You can obtain different types of locks on persistent objects. Once you obtain a Read or a Write lock, you can be sure that no one else can modify or, in the case of a Write lock, no one else can read the object you are working with. These two types of locks belong to the category of the Pessimistic locks.

Optimistic locks work on the following principle: a copy of the object is taken and the requesting program works on the copy. At commit time, the changes are copied back to the original object if no one else has changed it in the meantime. If the original has changed, the commit operation will fail. This type of locking technique is based upon a trade-off between maximizing concurrent access and the risk of experiencing a failure at commit time. While you hold an Optimistic lock, other users may acquire a Read, Write, or Optimistic lock. Many *traditional* applications also work according to this principle.

Figure 14 shows an Optimistic lock scenario.

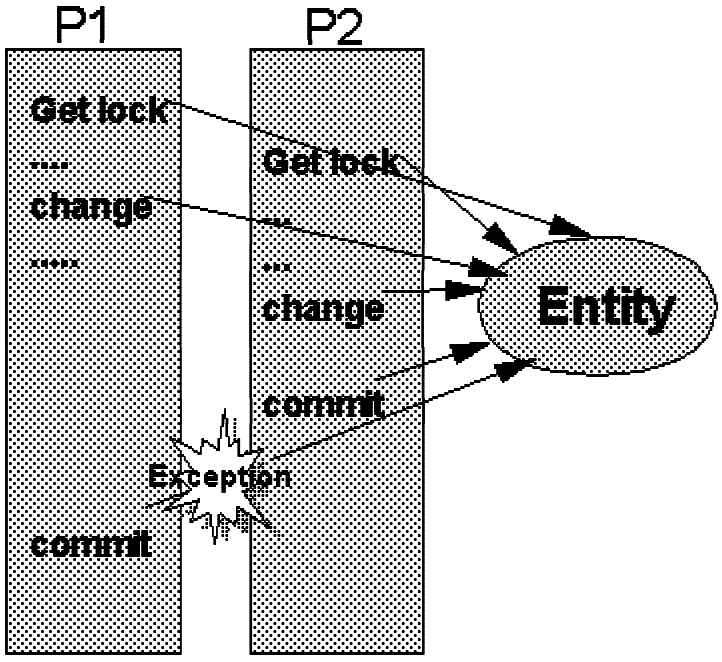


Figure 14. Optimistic Lock Scenario

When you acquire an Optimistic Clean lock, you want to access a business object in a read-only mode, but you also want to make sure that commit succeeds only if the object for which this lock was granted has not changed during the transaction. This is the case, for instance, of a stock exchange transaction. The decision to buy or sell a certain amount of shares depends on the value of the share. Your application can read the value and the operator can let the customer know. If the value changes during the transaction, the decision needs to be reconsidered. This case is a good candidate for the use of Optimistic Clean locking.

A special type of locking is represented by the NO_LOCK mode. When you request this level of locking, you receive a read-only copy of the object. The copy can even be taken outside of transaction boundaries and the Java reference you receive will survive across multiple transactions. This is the only case in which you are allowed to acquire a reference to a persistent object outside of transaction boundaries. NO_LOCK copies can be convenient. For instance, if you intend to show a list of objects in a GUI listbox from which a user can make selections, NO_LOCK copies offer an ideal solution since you can build the list outside of transaction boundaries and reuse it across multiple transactions.

3.2.6 Dependent

Whenever an object is logically part of something else and it is not meant to be shared by multiple objects, consider extending the San Francisco Dependent class in your code. Dependent objects are not inherently persistent. They can become persistent if they are contained in a business object that is persistent. They are private to the object that contains them. Dependent objects are not assigned a Handle and they can only be passed by copy.

Figure 15 shows an example of accessing Dependent objects where client programs are only allowed to work on transient copies.

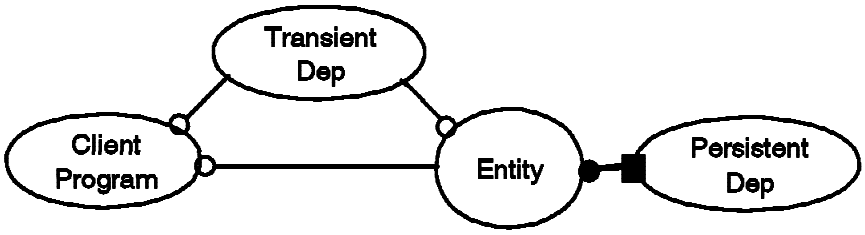


Figure 15. San Francisco Dependent Classes

Dependent objects are lighter weight than Entities and, in the right circumstances, can improve performance. Creating Dependent objects requires less effort on behalf of the Foundation and, therefore, performance can benefit. However, if a Dependent object is frequently accessed from outside of the object that contains it, a copy is taken for every access and performance may ultimately suffer. This is an indication that the Dependent object needs to be shared and that you should have preferred an Entity subclass to represent it.

3.2.7 The Base Factory

Most of the services offered by the San Francisco Foundation are accessible through the Base Factory object. You can obtain a reference to the Base Factory by invoking the `Global.factory()` method in your code.

The following list is a partial list of services that are accessible through the Base Factory:

- Starting a transaction (`Global.factory().begin()`)
- Committing/Rolling back (`Global.factory().commit()` or `rollback()`)
- Creating business objects (for instance, `Global.factory().createEntity(...)`, `Global.factory().createDependent(...)`)
- Deleting persistent business objects
- Requesting lock levels on business objects
- Accessing the notification mechanisms

3.2.8 Command

A special subclass of the Dependent class is the Command class. In San Francisco, Commands are meant to encapsulate series of actions that are part of a specific business process. In his book *Object-Oriented Software Engineering*, Ivar Jacobson established the need for Command objects to fulfill this role. There are two good reasons for isolating specific business logic in Command objects:

1. Reuse of Entity classes is improved by the fact that they are more generic when used in conjunction with Commands.
2. If business rules change, you can quickly identify which parts of your application need to be modified.

You can also use Commands to edit attributes of business objects. In this case, you can also provide `undo()` and `redo()` methods to your Commands. Query Command is a special subclass of Command that is meant to encapsulate a query request to be executed on a collection of objects. We expand on this subject in Section 3.2.9, “Collections.”

Commands can play an important role in tuning applications for performance. If a process involves dealing with multiple Entities and invoking a relatively large number of methods, both home mode and local mode might turn out to be unsatisfactory. In these cases, it might be worthwhile creating a Command that works in the remote address space and that execute the process, accessing the various Entities locally. San Francisco application performance may vary dramatically depending on this type of tuning.

3.2.9 Collections

Among the San Francisco Foundation classes, there are a series of Collection classes that are intended for organizing groups of business objects to fit specific needs. From a semantic point of view, San Francisco provides:

List Collections They remember the order of insertion and allow duplicate objects.

Set Collections No duplicates are allowed and no specific order is maintained.

Map Collections Allow objects to be retrieved by a specific key.

All Collections support the Iterator pattern. Iterators allow scanning the collection of objects from top to bottom in a way that does not depend on the internals of the Collection.

In Figure 16, you can see that there are two major families of Collections: those that own the elements they contain (Entity Owning...) and those that do not. The Owning Collections can only contain Entities. The other Collections can contain Entities, Dependents, and Strings. Keys of any type of Map collection can be Entities, Dependents, or Strings. When you delete an Owning Collection, all the contained elements are deleted as well.

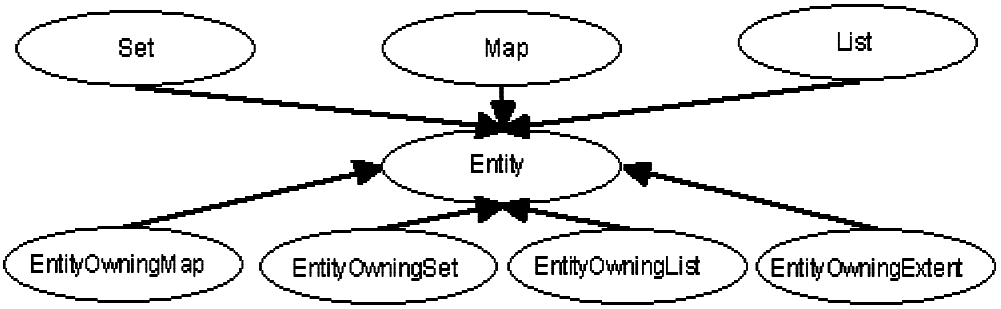


Figure 16. San Francisco Entity Collections

A special type of Collection is the Entity Owning Extent Collection. It supports keyed access in a slightly different way than a normal Map Collection. It has been specifically designed to be associated with a relational database table and, therefore, is well-positioned for containing a large number of objects.

When you access an object in an Entity Owning Extent, the San Francisco Foundation can exploit the keyed access mechanisms of the underlying relational database.

The Entity Owning Extent plays a key role in San Francisco applications when it comes to managing high volumes of data and to integrating new functions developed with San Francisco with legacy applications around a common relational database.

3.2.10 Query

Objects in San Francisco Collections can be queried using an object query language whose syntax resembles closely the SQL ANSI-92 standard. As a result of a query on a San Francisco collection, you receive a transient collection that stores the objects that meet the query criteria.

The query language provided by the Foundation offers some extensions to the standard SQL language and includes some limitations. You can query objects comparing the results of method calls with your query selection criteria such as in the following simple case:

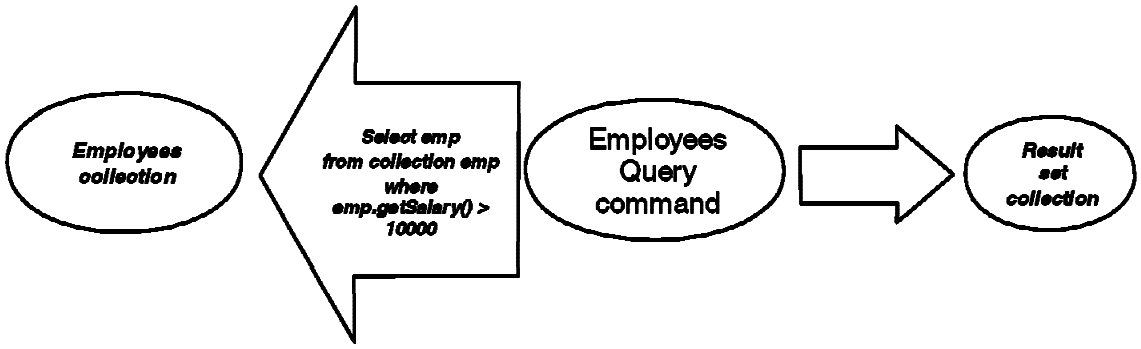


Figure 17. San Francisco Query Commands

Figure 17 shows that as a result of this query, the result of the method `getSalary()` is compared with 10 000 for all the objects in the collection. Those objects that satisfy the condition are selected and put into the resulting collection. It is syntactically allowed to call any type of method. You are not limited to getter methods. However, the syntax previously shown only allows methods that return a Java primitive data type such as a number or a string. If you need to perform more complex comparisons, you can use the Select Compare object. The Select Compare object is described in detail in the *San Francisco Programmer's Guide* and it represents a powerful extension to the normal SQL syntax.

3.2.11 Notification

The Foundation also offers a notification service. They can generate an event whenever their state changes or whenever certain conditions are met.

Any Java object can become an *observer* of these Entities and be notified when an event belonging to a certain category (Interest) occurs. Observers can be San Francisco Entities or normal Java objects such as user interface components. In fact, the notification mechanism is primarily applied to develop graphical interfaces to San Francisco server code.

Two types of notification approaches are supported in San Francisco:

Asynchronous When this approach is chosen, the notification service takes care of directly notifying the observers by invoking an `update()` method they need to implement.

On demand When this approach is chosen, the Foundation posts the event notifications in a mailbox from which the observer later retrieves them.

Figure 18 on page 34 illustrates the two different notification mechanisms that are available in San Francisco. Both mechanisms are based on the relationship in-between an observable and an observer. The observable is the source of the notification and needs to be a persistent Entity.

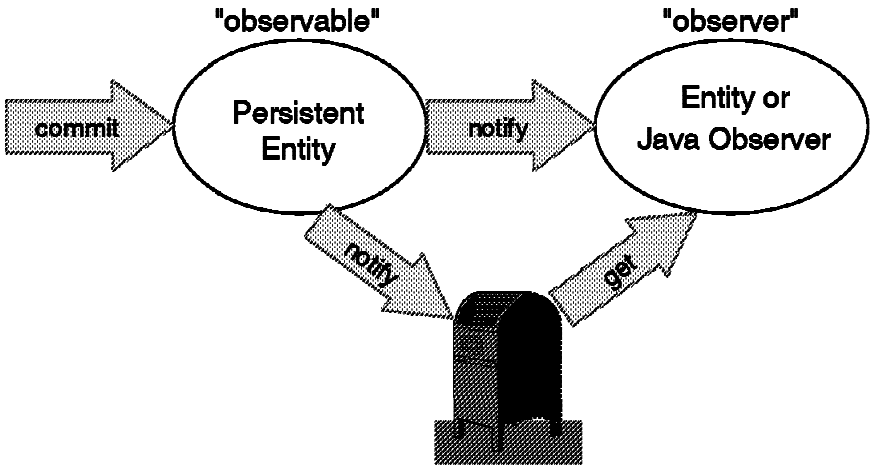


Figure 18. San Francisco Notification Mechanisms

Objects that want to become observers need to register with the Base Factory, specifying which observable objects they want to monitor and what their interest is.

3.2.12 Security

San Francisco allows the creation and the administration of users and passwords. You can also define your password policy, which establishes rules about the format and the expiration of passwords. Users and passwords can be administered with an appropriate utility provided by the Foundation (the User Administration utility). Once users have been defined, users have to identify themselves to San Francisco before they can access any application.

Another aspect of security is represented by the definition of secure tasks. You may want to make certain activities restricted to a subset of users. For instance, customer maintenance should only be accessed by some individuals in the marketing department. In this case, you can make customer maintenance a secure task and authorize only certain users to this task. San Francisco provides you with a series of APIs that can be used from within the applications to initiate secure tasks. The User Administration utility allows you to authorize individual users to the various tasks. You can further restrict the user's authorization to a secure task to be used only for the objects of one or more specified companies.

3.2.13 Naming

The Naming service is responsible for some key run time functions of the San Francisco architecture and insulates application coding from some underlying technical aspects such as persistent data stores.

There are three main areas where the Naming service plays an important role:

1. Assignment of user aliases and retrieval of objects based on user aliases.
2. Making objects persistent in the data store (container) configured for them.
3. Performing *class replacement*. When an application requests the creation of an instance, it uses a *class token* to indicate the San Francisco class the instance belongs to. The Naming service can dynamically replace this class

name with a different one, defined outside the application by using the configuration tools. This mechanism ensures that if you want to globally substitute a class with a different one (typically a subclass), you can do it without modifying the applications. Suppose you run an application that uses a Customer class. You might want to integrate with your own application that uses a subclass of Customer, say MyCustomer. Class replacement allows you to instruct the Naming to use MyCustomer whenever Customer is referenced.

The Naming service runs in the Global Name Server, which is discussed in Chapter 7, “San Francisco Utilities” on page 65.

3.3 Foundation Library Classes

Some commonly used classes are included in a category called the Foundation Library Classes. This section only gives a quick overview of the content.

3.3.1 The DDecimal Class

Decimal data types are widely used in commercial applications to represent numerical values. Decimal data types guarantee that the appropriate precision is preserved in calculations and that is why they are preferred to floating-point arithmetic.

The San Francisco Foundation includes a DDecimal class that allows you to represent decimal amounts in your applications. You can represent signed decimal numbers with an arbitrary precision. This class supports all the necessary arithmetic operations and, for each of them, it supports various forms of rounding results (raising, truncating, rounding, and so on).

3.3.2 The DTime Class

DTime objects store a point in time of a Gregorian calendar date (year, month, and day) and a time in that day (hours, minutes, and seconds). They represent an actual date and time. When you create a DTime instance, you specify a *precision* that allows you to ignore certain portion of the Dtime instance. For example, if you decide to ignore its minutes, seconds, and milliseconds, you specify an hour precision. The default is the highest possible precision (milliseconds). You can perform date and time arithmetic with DTime objects. For instance, you can determine how many days there are between two dates, or add a certain time to a date to obtain a new one. For more complex date-related calculations such as determining the number of working days between two dates, look at the Natural Calendar class in the Common Business Object layer.

3.3.3 Locale Support

San Francisco has been designed with great consideration for the issues that relate to internationalizing applications. For this purpose, Locale objects are used to define a series of parameters that depend on a specific country, language, or cultural environment. The date format, the time format, the decimal point format, and the character set are some of the parameters that are locale dependent.

This information is not kept by the Locale objects, but by the locale sensitive class. The Locale objects are only used as an identifier of the set of

culture-dependent parameters. The classes that are locale sensitive also have the responsibility for implementing the locale-dependent behavior.

San Francisco allows your applications to work with multiple locales. At application setup time (or through a maintenance procedure), you must create all the locales that are supported by your application.

Some classes in San Francisco are locale sensitive. Their behavior may change depending on the locale. In general, these classes implement the Translatable interface or the Describable interface.

When a class implements Translatable, it has to implement two versions of a `format()` method. The first version takes no parameters and returns a formatted version of the object according to the active locale. The second version takes a parameter that is a Locale object and returns the object formatted according to the passed locale. DTime, DDecimal, and DCurrencyValue (CBO layer) are translatable and can be formatted according to the supported locales.

If a class implements Describable, it has to support some methods that allow defining and retrieving descriptions in multiple languages. A Product, for instance, might have as many descriptions associated with it as many languages are supported by the application. In the San Francisco Foundation, there is already a subclass of Entity that implements Describable. It is called Describable Dynamic Entity and can be used to create persistent business objects that need to support multiple descriptions. Dependent objects can also be provided with multilingual descriptions by extending the Describable Dependent class. Most classes in the CBO layer extend either of these two describable classes. These classes make use of the Descriptive Information class, which in turns encapsulates and allows you to manage multiple descriptions.

The San Francisco Foundation also includes a set of classes that allow you to easily manage the multilingual resources in your applications. For example, the Locale Resource Controller holds and manages the set of repositories (message catalogs) where the various locale sensitive messages are stored.

In San Francisco, we have three categories of locale sensitive objects:

- Static messages, which are immutable and primarily used to describe error conditions. They are contained in Static Message Catalogs.
- Dynamic messages, which can change over time. They can be used as Describable Entities and Dependents and are contained in Dynamic Message Catalogs.
- Entities, Dependents and Strings are also locale sensitive

Refer to the *San Francisco Programmer's Guide* for more details on developing multilingual applications using these classes.

One of the methods that are defined by Describable is the `getDescription()` method. When this method is invoked with no parameters on a Describable Dynamic Entity, San Francisco will automatically try to retrieve the description that corresponds to the language indicated by the active locale. This mechanism enables an application to generically retrieve descriptions; at run time, the active local could be set appropriately and San Francisco would retrieve the correct description automatically.

3.4 The San Francisco Programming Model

A programming model defines a set of rules programmers should follow in order to be compliant with a certain standard or to correctly use certain services. San Francisco defines a programming model which specifies rules that depend on the role programmers are assigned to:

Developer role: It applies to programmers that need to develop new business objects or to extend existing classes to create new ones. This programming model addresses issues such as what methods need to be provided and how relationships among business objects need to be implemented.

Client role: It applies to programmers who use existing classes and deals with issues such as creating instances of business objects, starting and concluding transactions and so on.

The purpose of the San Francisco programming model is to ensure that programmers make a correct and safe use of the services offered by the Foundation. In addition, the programming model intends to guarantee a high degree of structural consistency across applications. For example, whenever you derive a new subclass of Entity, the programming model wants you to create a class factory that client programmers will have to use when a new instance needs to be created. If you are coding a Customer class, you will therefore have to provide a Customer Factory class with appropriate `createCustomer()` methods. These methods will in turn invoke the `createEntity()` method of the Base Factory. Client programmers will never invoke the `createEntity()` method; they will invoke `createCustomer()` on the class factory.

Thanks to these rules, the process of creating new instances is highly standardized across every San Francisco application. In addition, client programmers will be shielded from the complexity of using the Base Factory and are provided with a standard way for creating instances of any class.

Understanding the programming models is the first step you need to take before you start developing with San Francisco. The programming model needs to be strictly followed and some of the tasks that it entail are highly repetitive. For this reason, the San Francisco Code Generator (see 9.3.2, “San Francisco Code Generator” on page 97) is of extremely great help in developing San Francisco applications, as it automatically generates code that complies with the programming model.

Chapter 4. San Francisco Common Business Objects (CBOs)

The purpose of this chapter is to introduce the Common Business Objects layer of San Francisco and to explain the role that the Common Business Objects play in San Francisco application development. The last section of this chapter is an overview of the categories of Common Business Objects that are available in San Francisco.

This chapter contains the following sections:

- Section 4.1, “Introduction to Common Business Objects” describes the concept of a Common Business Object.
- Section 4.2, “Using Common Business Objects” on page 40 discusses using Common Business Objects to develop applications.
- Section 4.3, “Common Business Object Categories” on page 41 gives an overview of the existing Common Business Objects in San Francisco.

4.1 Introduction to Common Business Objects

Common Business Objects (CBOs) are a set of classes organized into groups, called categories, which perform functions commonly needed within business applications. These categories are derived from the need for the same function, or class, in more than one of the frameworks. By placing these categories in the San Francisco CBO layer and not in an individual framework, they can be used in several frameworks as well as in applications written directly to the CBO layer.

The CBOs can be divided into three groups:

- Those that contain business objects which are common across business domains.
- Those that contain function used to solve problems common across business domains.
- Those that provide a way for financial status to be affected from outside of the financial domain.

The first group of CBOs consists of business objects common across business domains. They are the objects that are needed in many applications. For example, the Company category contains the Company and Enterprise classes, which are used to represent the company and/or company hierarchy of the domain. And the Business Partner category contains the Business Partner class, which is used to represent those individuals, companies, and/or company hierarchies that our domain does business with.

The second group of the CBOs consists of function used to solve problems common across business domains. These functions are provided in a very flexible form so that they can be used to solve problems within the domain which are similar. This kind of function is called a pattern, for a more formal definition of patterns see Chapter 6, “San Francisco Patterns” on page 55. An example of this kind of pattern is contained in the Cached Balances category. This category provides function which is the basis for defining, maintaining and manipulating cached aggregations. A cached aggregation can be anything you add up. For example when trying to determine the balance of your bank account, you can

either add up all transactions you have ever had with the bank, or you can keep (cache) and update (maintain) the balance. In most domains, it is more complex than this, with each transaction having many criteria over which balances of single and/or multiple criteria are desired.

The final group of the CBOs provides a way for financial status to be affected from outside of the financial domain. This is provided as part of the CBOs, because most domains will have an affect on financial status. For example, if you sell something, in the simplest case, you must tell the financial domain to reduce the value of your inventory (by what it cost you) and increase your cash on hand (by the amount you received). An example of this kind of category is the Interface to General Ledger category. This category contains classes which allow you to give financial information to the General Ledger. It allows domains outside of the General Ledger to tell the General Ledger about financial status changes without having to know the details of the General Ledger or to know if a General Ledger is even present.

The Common Business Object layer, by its very nature, will continue to grow as more classes and function, as defined above, are identified by application and framework development.

4.2 Using Common Business Objects

CBOs can be used indirectly through the Core Business Processes and used directly. The way in which the CBOs are used depends on how the application scenarios map to the Core Business Processes. When an application scenario maps directly to a Core Business Processes scenario, CBO are used as part of using that scenario. When a Core Business Processes scenario is extended, it may require working with CBOs directly to implement the extension. When the application scenario does not map to any Core Business Processes scenario, the CBOs will be used directly. Regardless of how the CBOs are used, they provide an extensive set of classes and function that can be used.

For example, suppose we need to create the part of an application used in an exchange bureau. The clerk behind the counter needs to be able to enter an amount in a certain currency, select another currency and calculate the value of that amount in the new currency. At this time, this scenario is not provided by any of the Core Business Processes. However, if we look at the CBOs, we discover that within the Currency category, the functions and classes to represent the amounts and do the conversion are supplied. Thus the CBOs are very helpful in implementing the exchange rate calculator.

Common Business Objects, however, are not used like a class library where you can select a single class and use it in isolation. The CBOs are a framework, where using a class requires using the class within its collaborative relationships and understanding its interactions with other classes. In the example above, a number of collaborating classes from the Currency category are used to implement the exchange rate calculator.

One aspect of this collaboration is that in order to use a particular category in the CBOs, setup of another CBO category may be a prerequisite. For example, some CBOs work closely with the Company category, so the Company category must be setup prior to using those CBOs.

For example, the Currency category contains exchange rates which are associated with a particular company (contained in the Company category), thus the Company category must be setup first.

The company is normally one of the key things to setup when using the CBOs. There are two main reasons for this:

- The Company category provides the ability to represent our own organization and its hierarchy. It allows us to contain specific data about each of the companies, division and groups of our organization.
- Normally it is our organization that owns many of the business objects. Typically these are the business objects that are not naturally owned by another business object or are applicable across the entire organization. In the exchange calculator example above, the different currencies available in our organization would be owned by the organization, thus the Company category must be setup prior to defining the available currencies.

Another consideration when using the Common Business Objects is how different applications which use the CBOs will coexist and interact. For example, we need to be able to define the organizations we deal with, using the Business Partner category, and have them used by both our order entry application and our accounts receivable application. A potential problem could lie in the fact that each application may need to extend the classes in Business Partner to add information for their business processes. By using the San Francisco extension mechanisms with the associated rules to extend CBOs, the extended CBOs are able to coexist and interact. See the the *San Francisco Extension Guide* for these extension mechanisms and the associated rules.

4.3 Common Business Object Categories

The Common Business Objects categories can be divided into three groups:

General Business Objects: This group consists of business objects that are common across business domains (and the common functions associated with them).

Financial Business Objects: This group consists of business objects related to financials that are common across business domains (and the common functions associated with them), and categories that provide a way for financial status to be affected from outside the financial domain.

Generalized Mechanisms: This group consists of function used to solve problems common across many business domains. This function has been abstracted so that it can be used in many different ways. In many cases it is provided as a small framework, which is independent of the other CBOs.

Figure 19 on page 42 shows which group the current CBO categories are in.

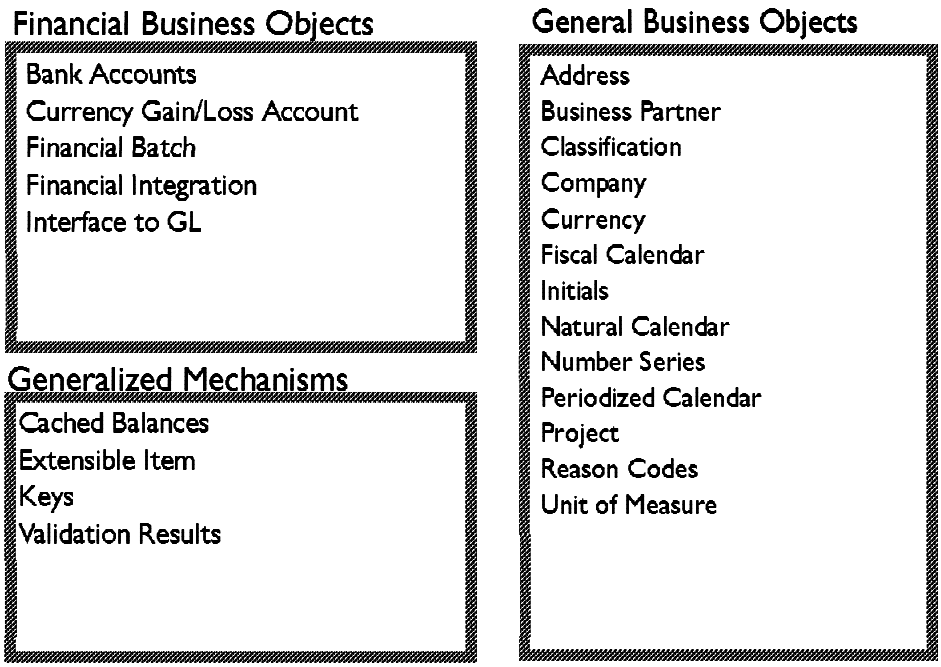


Figure 19. Common Business Object Categories

In this section we give an overview of the Common Business Object categories. However, because most of the categories in the Generalized Mechanisms group are patterns, they are described in Chapter 6, “San Francisco Patterns” on page 55. The descriptions are not duplicated in this chapter.

4.3.1 General Business Objects

These are the classes in the General Business Objects category currently available in the San Francisco Common Business Object layer.

4.3.1.1 Address

This category provides the classes and function for representing and storing addresses in the framework. The concept of address is used in a very broad sense and is conceived in a way that it can be extended for specific purposes. In particular the address represents and stores a specific physical location and the information for reaching (through many possible means) a business or individual. This category also covers the concepts of country and area. The area allows addresses to be classified. An example of area is dividing the USA into three regions: the East Coast, the West Coast and the Midwest.

4.3.1.2 Business Partner

The Business Partner category provides the classes and function for representing and working with Business Partners and their hierarchies. A Business Partner is any individual or organization with which we have a business relationship. For example, the customer for an order, or the supplier for a purchase order. This category supports the ability to spread the Business Partner information across the company hierarchy, having a portion of information, such as the Business Partner address, shared across all companies and other portions, such as the credit limit, applicable to only one company.

4.3.1.3 Classification

The Classification category supports grouping or classifying things in a dynamic manner. For example, a classification type can be defined, such as Color and then the values for that classification can be defined, such as Red, Green, and Blue. Then by associating a value, Green, with something, it is classified with respect to that type, Color. This classification is dynamic in the respect that new types can be defined and the values for a type can be modified. For example, a new type, Size or a new value, Purple, can be added at any time. In addition, the different types can be connected into a hierarchy.

Reason Code: Reason codes are used to record the rationale behind creating certain types of transactions (for example, write-offs). Since this is a form of classification, this category simply documents that the Classification category, above, is used for this function.

4.3.1.4 Company

This category provides the classes and function for two important roles:

- Organization representation
- Owner for unowned objects

The first role is to represent our own organization. It provides a means for containing information (data) about the different parts of an organization's hierarchy. The hierarchy consists of both legal and structural portions, which allows for representation of divisions or departments.

The second role is to own objects that do not have a natural owner. These are typically the business objects that are used almost everywhere, but for which another domain object which owns them cannot be found. In some applications these are the object that are already thought of as being owned by the company. For example, currencies are used in many business processes, but they do not have a natural owner, so they are owned by the company. However, addresses are always owned by another object, such as the business partner they are associated with, so in this case business partner is the natural owner.

4.3.1.5 Currency

This category supports the definition of currencies and exchange rates, and working with values associated with a particular currency.

The definition of currencies consists of defining things like the information necessary for formatting and displaying monetary values in a particular currency. For example, how many digits are significant.

Exchange rates, which define how to convert from one currency to another, are supported in a very flexible way in San Francisco. For example:

- Exchange rates can be based on dates or on the periods defined in the fiscal calendar.
- Multiple tables of exchange rates can be created and used. For example, having a special exchange rate table for tax purposes.
- The use of exchange rates can be customized with respect to how the tables are used. For example, to convert from Deutsche Marks to US Dollars, would the best course of action be to go from Deutsche Marks to French Francs and then from French Francs to US Dollars?

This category provides the support for working with a single value associated with a currency, called a currency value, and it supports working with two related currency values, called a transaction value. Both support the basic arithmetic operations, such as add. The transaction value allows an application to support multiple currencies. It allows the transaction to be recorded both in the base, system, currency and the prime, transaction, currency. For example, a French company would have a base currency of French Francs, but when billing a Germany company they would bill them in Deutsche Marks. This transaction value captures the amount on the actual bill, what we expect to get in Deutsche Marks and the equivalent amount in French Francs, and then allows us to manage our company in French Francs.

4.3.1.6 Fiscal Calendar

This category implements the basic behavior for the fiscal calendar. The fiscal calendar represents the accounting year and periods of the company. The fiscal calendar consists of fiscal years, which do not need to correspond to a normal year. Fiscal years covering parts of two calendar years are common when, for instance, the business of a company follows a yearly cycle that starts in the fall and ends in the summer. The fiscal year consists of fiscal periods, which do not necessarily correspond to months. For example, the fiscal periods may be four weeks, so that each period is the same length of time. The fiscal periods are used for recording transactions so that the costs and earnings are assigned the proper place in time. The fiscal periods are either dated or undated and have a particular type. The dated periods are used for normal accounting. The undated periods are opening or closing periods at the start or end of a fiscal year, and adjustment periods, which are used to make accounting adjustments.

This is similar to the support provided by the Periodized Calendar category (refer to Section 4.3.1.10, “Periodized Calendar” on page 45), however, the Periodized Calendar category only supports breaking the calendar into periods. It would be used in non-financial situations.

4.3.1.7 Initials

The Initials category supports the identification of certain roles of users of the system. An example is distinguishing supervisors from normal operators in an order entry system. Supervisors can approve high value orders and can allow a customer to, temporarily, go over the credit limit. A normal operator does not have these capabilities.

The Initials category is completely separate from San Francisco security.

4.3.1.8 Natural Calendar

The Natural Calendar category provides the support for defining and working with the calendar. As part of the definition, it allows information to be associated with particular days, such as working and non-working status and, for working status days, the working hours. This allows you to identify holidays as non-working and to identify a working day having only 3 hours. When working with the calendar, this information can be taken into account. As well as the normal calendar functions, it supports functions such as calculating the number of working days between two dates, and calculating the date which is a number of working hours from a given date.

4.3.1.9 Number Series

The Number Series category provides the support for generating numbers according to a specific numbering scheme. The support includes deciding where to start, end, increment, and how to build the generated number. For example, you could decide to start at 1, end at 203, increment by 3, and have a prefix of "MyNumber" with the number padded, with zeros, to 3 digits. Thus the number series would generate "MyNumber001", "MyNumber004", and so on.

The number series category also supports the concept of subseries, which allows a definition of a group of generated numbers that have the same characteristics. This is useful when the same characteristics are used from year to year, but each year we want to start over with the same initial value.

4.3.1.10 Periodized Calendar

This category provides a generic periodized calendar. The periodized calendar consists of periodized years. The periodized year can be defined with start and end dates different from a normal year. The periodized year consist of periods. Periods can be dated or undated and do not necessarily correspond to months. The periods cannot overlap and, within a periodized year, there cannot be any gaps between them.

This is similar to the support provided by the Fiscal Calendar category (refer to Section 4.3.1.6, “Fiscal Calendar” on page 44), however, the Fiscal Calendar category has the support necessary for financial use and should be used in those cases.

4.3.1.11 Project

This category supports the definition of project identifiers. These identifiers are associated with business objects, such as purchase orders, involved with a particular project so that those objects can then be grouped together to look at costs and revenue for a particular project.

4.3.1.12 Unit of Measure

This category supports definition of all the quantity related information relevant to a physical unit. It supports categorizing them by their physical properties and stores them by the category they are assigned to. In particular, it support includes:

- Defining units of measure. For example case, pallet, or 6-pack
- Working with values associated with a unit of measure. For example, 3 cases, 1 pallet, or 23 6-packs.
- Conversion between different units. For example, for one product we could have 1000 kg of sugar which corresponds to 1000 packs of 1 kg, and also to 4 pallets.
- Combining the quantity unit the transaction is carried out in with a base quantity unit. For example, our transaction would be carried out in kg of sugar, but internally we would track pallets of sugar.
- Defining the physical properties or dimensions of a concrete object. For example, the space 1 pack of sugar takes up in a warehouse or truck.

4.3.2 Financial Business Objects

These are the classes in the Financial Business Objects category currently available in the San Francisco Common Business Object layer.

4.3.2.1 Bank Accounts

This category provides the ability to define and work with banks and internal and external bank accounts. The banks are financial institutions you interact with. The internal bank accounts are your business' accounts at one or more banks, and the external bank accounts are the accounts of your business partners.

4.3.2.2 Currency Gain/Loss Accounts

This category provides the support for determining what General Ledger account to use for currency gains and losses. A currency gain, or loss, occurs when currency is revalued. An example of doing revaluation is when the exchange rate fluctuates and we have a bank account in a foreign currency. In other words, if we had a French Franc bank account with a balance of 200 French Francs, but our business runs in US Dollars, we would have used an earlier exchange rate to convert the French Francs to US Dollars. If the exchange rate was 5 French Francs to 1 US Dollar, the balance would be 40 US Dollars. If the exchange rate changes to 20 French Francs to 1 US Dollar, which translates to a new balance of 10 US Dollars, that new balance needs to reflect the fact that we now have 30 US Dollars less in the account. The currency gain/loss accounts are used to determine what account to use.

This category supports accounts for both realized and unrealized gains and losses. An unrealized gain or loss is when we do not know what the actual value is, but we know that the exchange rate has changed and we want to reflect that change. For example, if we are owed something in a currency other than our base currency and we want to know how much it is worth now. A realized gain or loss is when we know what the actual value is currency.

4.3.2.3 Financial Batches

This category provides the ability to process a group of financial items together. Which items get grouped together is up to the application. For example, all work by an individual, or all work for a particular session can be collected together. The use of Financial Batches is optional.

4.3.2.4 Financial Integration

This category provides the support for mapping from a particular domain to the General Ledger. In most businesses, whatever you do has an affect on the financial status of your business. It is critical that other applications be able to easily notify the General Ledger of changes to financial status. However, most applications do not deal with items that have a one to one mapping into the General Ledger. For example, within the warehouse we deal with products and stock types, but the General Ledger views them as accounts. Getting from products and stock types to the correct account is difficult to do.

What the Financial Integration category provides is a way of setting up the mapping in such a way that someone in the application (or domain) causing the financial status change can deal with the things they are familiar without having to know the details of the General Ledger. In addition, the knowledge of the mapping is centralized, so that the mapping expertise can be applied more effectively.

This category supports many mappings. Any number of domains can use it to map to the General Ledger. The application provider can determine to what extent the mapping are shared, if at all.

4.3.2.5 Interface to General Ledger

This category provides the clearly defined interfaces into the General Ledger. In most businesses, whatever you do has an affect on the financial status of your business. It is critical that other applications be able to easily notify the General Ledger of changes to financial status. The interfaces defined by this category are provided as part of CBOs, because they need to be available even when a General Ledger is not. By using these interfaces in a well defined way, an application providing information to the General Ledger does not need to know any details about the General Ledger or even if a General Ledger is present.

When complex mapping is required to determine the correct account to use in the General Ledger, the Financial Integration category (refer to Section 4.3.2.4, “Financial Integration” on page 46) should be used.

4.3.3 Generalized Mechanisms

These are the classes in the Generalized Mechanisms category currently available in the San Francisco Common Business Object layer.

4.3.3.1 Cached Balances

Refer to Section 6.3.7, “Cached Balances” on page 62.

4.3.3.2 Extensible Item

Refer to Section 6.3.8, “Extensible Item” on page 63.

4.3.3.3 Keys

Refer to Section 6.3.6, “Keys and Keyables” on page 61.

4.3.3.4 Validation Results

This category provides support for reporting back results of validation logic. It provides the ability to collect validation results together, allowing validation failures to be collected and returned together. This is used by all of the CBOs and frameworks.

Chapter 5. San Francisco Core Business Processes

The purpose of this chapter is to introduce the Core Business Process layer of San Francisco and to explain the role that the Core Business Processes play in San Francisco application development. We first introduce the general concept of Core Business Processes and how they are used. We then look at the General Ledger Core Business Processes and how they can be used. In looking at the General Ledger, we begin with a brief overview of what a General Ledger is, so that people unfamiliar with accounting are able to understand it as an example of a Core Business Process.

This chapter contains the following sections:

- Section 5.1, "What Are Core Business Processes?" explains the general concepts of Core Business Processes.
- Section 5.2, "Using Core Business Processes" on page 50 discusses using San Francisco Core Business Processes to build applications.
- Section 5.3, "Core Business Process: San Francisco General Ledger" on page 50 introduces the San Francisco General Ledger as an example of a Core Business Process.
- Section 5.4, "General Ledger Framework Categories" on page 53 gives an overview of the General Ledger categories.

5.1 What Are Core Business Processes?

Core Business Processes deliver a set of basic building blocks for an application in a specific business domain. This is done by providing the core processes for that domain. The core processes are identified by looking at various application implementations in the domain and identifying those processes that are needed by most of the applications within that domain. The San Francisco architecture supports these core processes and provides a flexible means to imbed them into ISV applications. An example of this flexibility lies in the volatile logic that is built into these core processes. This volatility is a result of countries having different legislation, different ways of doing things, and differing customer preferences.

In these volatile areas, the ability to change the logic or extend it is captured within extension points. For example, when using exchange rates to convert between currencies, how the entries in the exchange rate table can be used varies from application to application. In one case, only an exact match to an entry in the table is acceptable. In another case, multiple entries in the table can be used.

For example, if you need to convert from Deutsche Marks to US Dollars you can either find an entry from Deutsche Marks to US Dollars to do the conversion, or if the table entries are available, do the conversion by first converting from Deutsche Marks to French Francs and then converting from French Francs to US Dollars. So, in this particular case, how the exchange rate tables are used is captured in an extension point. For more information on extension points, see the *San Francisco Extension Guide*. Also, some extension points are discussed in Chapter 6, "San Francisco Patterns" on page 55 and in Section 8.4.2, "Extending San Francisco Framework Classes" on page 87.

San Francisco delivers the Core Business Processes as object-oriented frameworks. A framework is a set of interacting and related classes that provide the basic architecture, design, and functions of a business domain. These functions can be extended to accommodate the specific needs of an application built using it. Part of the Core Business Processes are the classes needed by the particular domain. These classes represent the business objects, which are uniquely part of this domain, which makes these objects familiar to experts in that domain. Classes that have broader applicability (found in various domains) are found in the Common Business Objects layer (refer to Chapter 4, “San Francisco Common Business Objects (CBOs)” on page 39). The Core Business Processes, and how the various business objects interact, also capture identifiable portions of the domain's business processes and how those processes interact.

5.2 Using Core Business Processes

The Core Business Processes are used to develop applications in a specific domain. The Core Business Processes provide the core processes, in the form of an object-oriented framework, for the domain. Using the Core Business Processes consists of determining what your application will do and then mapping this to what is provided by the Core Business Process framework and the CBOs. A methodology for developing an application using Core Business Processes along with a discussion of how to do this mapping, is described in Chapter 8, “San Francisco Application Development Methodology” on page 81.

In Core Business Process mapping, there are three possible situations:

- **The Core Business Process provides all of the function that is needed for your application.** In this case, you can use only what is implemented in the Core Business Process.
- **The Core Business Process provides what you want, but differs in the way certain algorithms are implemented with respect to what you want your application to do.** In this case, you need to look for extension points in the Core Business Process. Extensions points are places where the architecture for the Core Business Processes has been made easy to change. For example, separating a volatile algorithm into a separate object so that it is easy to replace.

  If an extension point is not found, a different means of extending the framework should be considered. See Chapter 8, “San Francisco Application Development Methodology” on page 81 for more information.
- **The Core Business Process does not have equivalent function to what your application needs.** In this case, your application will have to be implemented using the Core Business Processes classes, the Common Business Objects and the Foundation layer or by creating your own classes.

5.3 Core Business Process: San Francisco General Ledger

This section serves two purposes. The first is to provide an example of a Core Business Process. The second is to provide an overview of what is provided by the General Ledger Core Business Process. To serve both purposes, this section is divided into a brief introduction to the concepts of a General Ledger and an overview of what is provided by the San Francisco General Ledger Core Business Process. Even though the introduction is intended for someone

knowing little or nothing about a General Ledger, it also provides good base information for a General Ledger expert. As an expert, it will help you to gain an understanding of the terminology used in San Francisco's General Ledger.

At first the differences in terminology may be surprising, but many applications from many countries were looked at, each with their own terminology, causing us to settle on a general set of terms. Sometimes we implemented the abstract portions of the Core Business Process, which required us to use more abstract names for the business objects.

5.3.1 General Ledger Overview

The main purpose of the General Ledger is to report the flow of money for your company. This is required for legal reasons and for managing your business. A standard approach for this accounting process has been developed. This can be illustrated in the accounting cycle. This cycle consists of three main steps:

- Setting up the structure
- Day-to-day activities
- Periodic activities

5.3.1.1 Setting Up the Structure

The first step is to setup the structure for the information you want to keep in the ledger. To set up this structure, you identify your chart of accounts. The chart of accounts consist of all the accounts for which you want to keep information. The purpose of the account is to group related transactions together. The chart of accounts are customized depending on the kind of business you are running, the level of detail you want in the final report generation, and the type of information you want to keep in the accounts. The chart of accounts varies greatly between companies and is influenced by country legislations.

To support these differences, the Core Business Process has the concept of a Posting Combination. The Posting Combination represents ways to relate transactions. In some applications, this is called the account. In others, the account is one of many ways to relate transactions. For example, in a typical Anglo-Saxon chart of accounts, the Posting Combination will be equivalent to what is called an account. This account is broken down into many different pieces, such as department and project. In a typical Scandinavian application, one piece of the Posting Combination would be equivalent to the account, while the remainder would be different classifications for analysis, such as department and project.

The Posting Combination is divided into Analysis Groups. An Analysis Group names a particular piece of the Posting Combination and controls the valid values, called Analysis Codes, that can go into that piece. The validity of analysis codes differ from application to application. This is the place where the framework provides extension points to make it easy to define and enforce your application's requirements.

The San Francisco General Ledger framework provides a flexible way of identifying the type of account, such as balance accounts versus profit and loss accounts. Balance accounts keep track of assets and liabilities and are are part of the balance sheet. While, the profit and loss accounts keep track of revenue and cost and are part of the profit and loss statement.

Therefore, defining the chart of accounts consists of:

- Defining the Analysis Groups
- Defining the Analysis Codes
- Identifying what Posting Combinations are valid
- Identifying the types of accounts for your application

5.3.1.2 Day-to-Day Activities

The second part of the accounting cycle is the day-to-day activities. This consists of creating, working with and finalizing the transactions. A transaction consists of a number of pairings of an account, (in San Francisco terms, the Posting Combination), with an amount for a particular time period. In normal circumstances, the transaction will balance conceptually. For example, for each transaction, amounts will move between individual accounts. For example, when you pay rent, you would record a debit in your Rent account and a credit in your Bank account. In the San Francisco General Ledger, the pairing of a Posting Combination with an amount is called a *Dissection*. The transaction, or set of related Dissections, is called a Journal.

Within the Dissection, the amount is held as a Transaction Value, from the Currency category of Common Business Objects. The Transaction Value allows for the capture of both the prime and base values of the amount. The Prime value is the value in the currency used in the transaction. The Base value is the value in which the General Ledger is kept. For example, if a German company purchases something from a US company, the prime value would be the amount in US Dollars and the base value would be the amount in Deutsche Marks.

In some applications, quantities are included in the transactions. To support this, the San Francisco General Ledger allows quantities, called Quantity Dissections, to be associated with a Dissection. If a Dissection has a Quantity Dissection, then amount is optional.

What makes a Journal valid is something that varies from application to application and is an extension point in the framework. For example, an application could require that certain types of Dissections must have Quantity Dissections. When a Journal is classified final also varies between countries. In some countries, until the Journal information is printed for the government, the Journals can be changed. The framework provides an extension point for enabling this.

Therefore, to do the day-to-day activities, Journals are created, Dissections, and Quantity Dissections are added and removed until the Journal is valid to record, then the Journal is made final by posting it.

5.3.1.3 Periodic Activities

The last step in the accounting cycle coincides to activities that occur periodically, such as activities related to the fiscal periods and the generation of reports.

Activities related to the fiscal periods are processes that are run after a period is closed or at the end of the fiscal year. These processes include things such as currency revaluation, closing out certain accounts, reconciliation, and adjustments to correct mismatches. The way in which these tasks are performed is ruled by the accounting practices and legislation applicable to your application. The framework provides extension points to define these processes for your particular application. Also, these processes, such as currency

revaluation, are provided in such a way that they can either be done as part of the period processing or on an as needed basis.

The main purpose of the General Ledger is to manage the flow of money. To manage this flow, reports must be generated that allow the flow to be looked at in many different ways. There are two main types of reports:

- The balance sheet, which shows assets and liabilities
- The profit and loss statement (or income statement), which shows the revenue and costs.

For these main types, there are many different ways to report data. Because reporting varies dramatically from application to application and because this is a place where applications differentiate themselves from others, the San Francisco General Ledger does not provide a report generator.

However, the San Francisco General Ledger does provide a flexible way of getting the information necessary to produce any report. The framework has a set of criteria that can be used to produce account balances. The set of criteria could be something like: periods 1 through 4 and profit and loss accounts. The set of criteria is an extension point, so the application can add new criteria (or disable existing criteria). Also, the San Francisco General Ledger can identify certain sets of criteria as needing to have fast access. Fast access is provided by storing and maintaining the account balances associated with the set of criteria. The framework provides the mechanism for maintaining and retrieving the maintained balances, but leaves the definition of what balances should be maintained to the application.

5.4 General Ledger Framework Categories

As with other layers of San Francisco, the General Ledger is divided into categories. These categories group related processes, and their associated classes, together. This section provides an overview of each of the categories in the General Ledger.

Posting Combinations

This category provides the processes and business objects for defining and working with the chart of accounts. Posting combinations include the definition of the account, account validation criteria, and the type of account.

Journal

This category provides the processes and business objects for creating, maintaining, and finalizing transactions in the General Ledger. Journal includes support for both value and quantities, support for transactions in a currency other than your General Ledger's currency, and support to define what makes a valid transaction.

Balances

This category provides the basis for creating reports and balance inquires. It provides the processes and business objects to flexibly define criteria for which balances are needed. The criteria can include the fiscal period, the type of account, and the prime currency. Balances includes support to extend the criteria and to store and maintain balances for specific criteria for fast access.

Budgets

This category provides the ability to include budgets in the General Ledger. The San Francisco General Ledger keeps the budgets in the same ledger, which allows for all of the same functions to be applied. Although the two types of Journals are combined within the framework, the application could decide to make it appear as if two separate ledgers are used.

GL Fiscal Calendar

This category extends the Fiscal Calendar from the Common Business Objects to add information about the Fiscal Periods with respect to the General Ledger. In particular, whether the Fiscal Period is open or closed, and the current (or default) period.

Closing

This category provides the processes and business objects to support the closing of a fiscal year. The framework provides a simple example of closing because this process varies dramatically between businesses.

Bank Accounts

This category extends the Internal Bank Accounts in the Bank category of the Common Business Objects. It supports associating Posting Combinations with a particular Internal Bank Account. Also, when the Internal Bank Account is involved in a transaction, the correct Posting Combinations can be used for the bank account, bank charges and bank suspense.

Banks

This category supports the processes and business objects for dealing with the Bank. This includes tracking transactions within the bank, managing bank statements, and reconciling the bank statement with the bank transactions.

Revaluation

This category provides the processes and business objects to support the revaluation of currencies. Currency is revalued when you have a bank account in a currency other than your General Ledger's currency (base currency) and when the exchange rate changes over time. When this occurs, the original value in your base currency may no longer reflect the correct value, so a revaluation is done using a more current exchange rate. This category supports both realized and unrealized gain and losses.

These categories covering the accounting cycle make up the core processes and business objects needed for the General Ledger domain.

Chapter 6. San Francisco Patterns

The objective of San Francisco Business Process Components is to enable Independent Software Vendors (ISVs) to produce customized multi-platform business applications. To achieve this objective, San Francisco makes extensive use of design patterns. This chapter discusses design patterns and in particular those patterns used by San Francisco to facilitate the understanding and use of a framework. San Francisco uses both common and standard design patterns, and patterns developed to support particular problems found in the business domains.

To understand and extend a San Francisco framework, the application developers must understand general patterns and those patterns developed for San Francisco. This chapter introduces San Francisco design patterns and their use in the frameworks. It contains the following sections:

- Section 6.1, “What is a Pattern?” introduces design patterns.
- Section 6.2, “Why Use Patterns?” on page 56 covers general advantages of design patterns.
- Section 6.3, “San Francisco Patterns” on page 56 identifies specific patterns used by San Francisco and discusses their use.

6.1 What is a Pattern?

Design patterns are techniques aimed at solving recurring design problems. A good explanation of patterns was introduced by Christopher Alexander, who defined the concept of patterns in architecture, as follows:

A General Definition of Patterns

“Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice.”

(Source: *A Pattern Language: Town/Building/Constructions*).

Although Alexander's “environment” was architecture, the idea of patterns can be applied more generally. In the case of San Francisco, the idea of patterns has been applied to recurring problems encountered in the framework. In particular, a core set of patterns has been identified and used to introduce flexibility into the framework, which makes it easier to learn how to use and extend the framework.

The San Francisco development team used many of the patterns found in the book, *Design Patterns, Elements of Reusable Object-Oriented Software*; Erich Gamma, et al.; Addison-Wesley Publishing Company, 1994 (ISBN 0-201-63361-2). This book defines and shows examples of applying patterns to design object-oriented software. The following definition of design patterns in the object-oriented world is from that book.

A Definition of Patterns in Object-Oriented Design

"A design pattern names, abstracts, and identifies the key aspects of a common design structure that make it useful for creating a reusable object-oriented design."

(Source: *Design Patterns, Elements of Reusable Object-Oriented Software* Erich Gamma, et al.; Addison-Wesley Publishing Company, 1994).

The San Francisco development team used the patterns from this book whenever possible. However, certain problems in the domain either had not been solved before or had not been solved in a manner fit for a framework. When a problem was encountered more than once (or when it is known to recur), a new pattern was considered. Once a pattern was created, its definition and documentation was iterated on as other uses of it were discovered and as the understanding of the problem evolved.

6.2 Why Use Patterns?

The San Francisco patterns guarantee consistency throughout the framework. *Patterns systematically name, motivate, and explain a general solution that addresses a recurring problem*. Once you understand how the problem was solved using a particular pattern, the next time you encounter the problem or pattern you will understand the solution. Sometimes the solution is customized to fit the special conditions of a particular need, however, the core basis of the solution (pattern) is consistently used throughout the framework. Once developers are familiar with patterns and their use in solving problems, they articulate the solution to a particular problem through the use of one or more particular patterns, thereby creating a language to express this and other solutions.

6.3 San Francisco Patterns

Each of the patterns described in this chapter are structured as follows:

Intent States the intent of the pattern or the problem it is intended to solve.

Concept Gives a high-level description of how the pattern fulfills the intent.

Benefits Explains how the pattern serves the goals of extensibility and reuse.

It is important to note that patterns are often related to each other. We have elected not to include that information here. The information is included in the *San Francisco Extension Guide*, which discusses each pattern in detail.

6.3.1 Factory Class Replacement

6.3.1.1 Intent

Situations occur where a class needs to be modified to support a particular business need. Client programs which create, access or maintain instances of this class must be allowed to remain ignorant of the modifications to the said class unless they wish to use the new information.

6.3.1.2 Concept

Factory Class Replacement, similar to the Abstract Factory pattern from the *Design Patterns* book, allows for specifying a substitute class that is created instead of the one requested. When a client requests the creation of a new object, it issues a request to the desired class factory. The class factory requests the Base Factory (a San Francisco Foundation base class) to create a new object. The Base Factory determines the actual class instance to be created, which is returned to the client.

6.3.1.3 Benefits

The obvious advantage of using this pattern is the ability to replace a class without affecting any users of that class. This relieves client programs from the burden of needing to know about unrelated changes to the classes it uses.

6.3.2 Commands

6.3.2.1 Intent

The business environment changes over time and the application supporting that environment needs to be able to react to those changes, while minimizing the impacts to the overall application. An additional need is to provide a way to update a group of objects at once while ensuring that if the complete group is not updated successfully, the changes are all reversed. The business logic may span many business objects (classes), which makes the placement of this logic within one particular object impractical.

6.3.2.2 Concept

Commands (similar to the Command pattern in the *Design Patterns* book) are objects with the sole purpose to provide a separate location for a specific piece of business logic processing. By focusing this logic into a single object, the particular business logic changes can be modified without impacting various users, thereby isolating the users from the changes.

A Command often affects more than one business object, which may require that all the objects affected are updated or all changes reversed. Commands support a roll back capability that allows all these changes to be reversed.

Variations of the Command exist to support special requests. The Query Command is a special type of Command that encapsulates a particular query request, which then creates a subset of the elements of a collection. The Query Command encapsulates features of an object query *similar to SQL language*, including selection and sorting.

6.3.2.3 Benefits

The implementation of business logic affecting several distinct business objects through Commands clearly brings advantages to maintenance and also allows application designers to isolate activities. By encapsulating the business logic in a Command object, *using programs* are isolated from changes in that piece of logic. It then becomes much easier to replace, modify, or enhance a certain piece of logic without impacting its users. Command objects also localize business tasks, making it easier to locate them. This approach increases the flexibility of the framework.

6.3.3 Property Container

6.3.3.1 Intent

An ISV typically creates an application which supports a wide variety of customers. Sometimes a particular customer needs to add specific information to objects that are part of the application. This specific information may effect all instances of objects of a class, or only specific objects. For example, an application user may need to track the receiver of a particular invoice for those invoices over a certain dollar amount into the associated GL account.

6.3.3.2 Concept

The Property Container design pattern is a good solution for this problem. A property may be an additional piece of information (an attribute), or an association with another object (relationship in the object model). Properties do not effect the context (for example, the property does not effect any of the other relationships this object is involved in) of the object when added to an existing object. The property information is added, accessed and maintained as needed. An application is then able to extend the same object without impacting other users of the object. Furthermore, the pattern is used to allow selective extension of individual objects at run time, as opposed to subclassing, which extends *all* objects of the class at compile time.

Properties can be single attributes or more complex objects. Each property is added to a Property Container with an identifying string that is considered the property name. Properties are retrieved from a Property Container by specifying its property name.

6.3.3.3 Benefits

Loose coupling is a key benefit of this pattern. Loose coupling refers to the degree of “awareness” existing between coupled classes. By using Property Containers, there are no changes to the classes of affected (coupled) objects. Property Containers are implemented through generic interfaces inherited by all classes implementing this pattern. These interfaces allow Property Containers to manage properties, irrespective of their usage or type. Therefore, for any class implementing Property Container pattern, the addition, retrieval or update of properties does not require class changes. Classes are changed only if tighter coupling is needed.

An important benefit of the Property Container pattern is that it allows concurrent extensions of the same business object by multiple ISVs. Each ISV can add their own properties to the same business object without affecting the other. Property Containers are also useful where information does not need to be attached to every object of a particular class, but only to certain objects as discovered through business processing.

It is important to remember that Property Containers cannot directly cause behavior changes (that is, the addition of new methods in containing classes). To provide these capabilities, the developer should use other means, such as subclassing or class replacement (discussed in Section 8.4.2, “Extending San Francisco Framework Classes” on page 87).

6.3.4 Policy

6.3.4.1 Intent

Each business has a series of business policies that define exactly how that business participates in the marketplace. These policies are applicable for the employees of the business and the business' customer set. These policies include factors from the various layers of government, the business owners, how the company moves the product and the types products the business sells. Specific business policies (in the form of algorithms) are incorporated into the business applications developed to support a business environment. Businesses exist in ever changing environments, which can make these business policies volatile. Changes to business policy often impact the business application, which can require the application developers to make adjustments.

6.3.4.2 Concept

The Policy (similar to the Strategy Pattern in the *Design Patterns* book) pattern encapsulates the business policy (algorithm) as the primary motivation for the object. This allows these algorithms to be managed (created, replaced, extended, and so on) independent of objects relevant to the situation. This pattern also allows for the creation of families of related algorithms. For objects implementing this pattern, a user may select a suitable policy from the family of policies and dynamically associate it with a target object at run time. Thus, the policy that is used for an operation may vary among the different instances of the class.

San Francisco has identified three common scopes:

Object Specific
: Implies that a Policy is defined at object creation time, or possibly passed into the operation affected by it. Each instance of a used class may have a different Policy.

Company Wide
: Implies that a Policy (or family of Policies) is maintained at the company level (for details on company hierarchy, see Chapter 4, “San Francisco Common Business Objects (CBOs)” on page 39) and is used for all instances of this class within this company. Objects in different companies may have different Policies. These Policies are set up at company setup time.

Application Environment Wide
: Implies that a Policy is defined at the application level and is applicable for all instances used in all enterprises defined by this application. In this case, every object of this class in the entire application uses this Policy. The Policy is decided at application deployment through the configuration process.

A Policy may have *state* associated with it (that is, the Policy class may have internal attributes), or it may be a pure algorithm with no state. Most of the Policies in the framework are state-less (that is, they do not have internal attributes). This keeps the Policies as light-weight as possible, and ensures the exact algorithm that is executed for every instance of a specific Policy class remains the same.

6.3.4.3 Benefits

The Policy pattern allows for the isolation of volatile business logic making it easier to identify and make changes to that logic while minimizing impacts to the overall application. This pattern allows for the extension of default Policies provided by a San Francisco framework without impacting the classes using these Policies.

6.3.5 Controller

6.3.5.1 Intent

The business application needs to manage multiple instances of a business object within its organizational structure. This is normally achieved by using collections, however collections do not naturally cooperate with the organizational structure.

For example, complex business environments need to support ways to provide information at different levels within the overall business. For example, a company can be made up of divisions which are include many departments. To make the company run smoothly, certain information needs to be shared at different levels.

6.3.5.2 Concept

Controllers are objects used to track and manage primary business objects within a business domain. Controllers can be attached to the organizational hierarchy. Controllers for specific business objects can be attached at different levels in this hierarchy. The business objects held by the Controller and association with a particular level of the organizational hierarchy may be shared with, or isolated from, other levels. This is accomplished by the type of Controller that is selected to hold the business objects. San Francisco provides three types of Controllers:

Root This Controller manipulates only its immediately contained objects; hence, it assumes that there is *no* higher level in the organizational hierarchy. It does not have access to the objects held at a higher level in the organizational hierarchy.

Aggregating This Controller allows sharing objects used by higher levels in the organizational hierarchy. If a Controller of this type does not find a requested object in its immediately contained set of objects, it passes the request to the "parent" Controller (a Controller above it in the organizational hierarchy). If the parent Controller does not find the requested object, it passes the request to its parent Controller. This process continues up the hierarchy until the object is found or a root Controller is encountered.

Hiding This is a special type of aggregating Controller that maintains a collection of object IDs for objects held by the parent Controller which are hidden (that is, treated as nonexistent). This effectively restricts the use of these objects to their defined level in the organizational hierarchy.

6.3.5.3 Benefits

Controllers participate in the dynamic (that is, at run time) configuration of an organizational hierarchy for sets of business objects, each pertaining to a business area of interest (for example, business partners, banks, currencies). They enable sharing, overriding and restricting the use of these business objects at each level of the hierarchy. They also hide the complexity of the organizational hierarchy by encapsulating calls to higher levels of the hierarchy.

A key benefit of this pattern is that Controllers may be added, removed, or changed independently at any level of the organizational hierarchy independently without impacting other Controllers. Controllers implement *indirect* links through the organizational hierarchy. This means that changing a Controller does not necessitate that all children Controllers have to update their links to the new parent Controller (as the links are fixed by the organizational hierarchy).

6.3.6 Keys and Keyables

6.3.6.1 Intent

Applications often need to work with information held in objects based on specific characteristics. These characteristics can be directly identifiable with the object, like attributes, or indirectly identifiable with the object based on the particular use of the information (for example, using the purchasing Business Partner to identify invoices).

The specific characteristics being used need to be separated from *how* the characteristic is used. This separation is needed because the use of the characteristics is usually stable, but the particular characteristics are adjusted by the application and the end user to fit the current business need. For example, wanting to identify invoices is stable, but the particular characteristics (the invoice date or the currency) of an invoice can be volatile.

6.3.6.2 Concept

This pattern provides a way to associate characteristics with objects for a particular use. This pattern separates the definition from the usage of the characteristics by using the abstraction of a Key. A Key encapsulates a particular set of characteristics, and allows the Key to be used without knowledge of the actual characteristics. The individual characteristics in the Key are encapsulated in Keyables, allowing the Key to work with the characteristics in an abstract way. Two types of Keys are provided as part of this pattern. One, called an access key, is used to encapsulate a set of specific characteristics. The other, called a specification key, is used to encapsulate a set of grouped characteristics. The specification key is very powerful because it can be used to specify what characteristics are valid or invalid for use in access keys.

For example, in a library, we want to be able to find items, such as books, videos or magazines, based on certain characteristics. We would create, and maintain, a mapping from an access key to the item, or items, with the given characteristics. The access key could include characteristics such as the type of item, the author, the subject, and the title by encapsulating them into Keyables. We could then map from this access key to the item, or items, with those particular characteristics. This would allow for the use of a specification key to limit our access keys to only one particular item type, such as "book".

The use of the Keys pattern supports the ability for the application provider and end user to customize the characteristics used. By using the specification key, a characteristic can be treated as if it does not exist, which allows for the possibility for the set of characteristics to be customized. In the library example, we could customize the specification key so that the author characteristic is not used in the access keys.

6.3.6.3 Benefits

This pattern, by separating the usage and definition of characteristics, allows the code that uses the Key to be reused, without change, when the Key is extended. By combining specification keys with access keys, the application and end user can customize the characteristics to fit their particular needs.

6.3.7 Cached Balances

6.3.7.1 Intent

Users often need to repeatedly calculate values that involve collecting data from multiple objects. These calculations are usually needed on a continual basis and are needed rapidly. These calculations could be done by collecting the data from all of the objects each time it is needed. In many cases it is more valuable to store and maintain the results of the calculation to get them more rapidly.

6.3.7.2 Concept

This pattern provides a mechanism for defining what calculation results need to be stored and maintained for rapid access. The user of this mechanism can decide what level of control should be exposed, such as providing a predefined set of calculations, or allowing the user to specify what calculations they are interested in. This pattern uses the Keys/Keyables pattern to define the characteristics associated with a particular result of the calculation.

For example, using the library example in the Section 6.3.6, “Keys and Keyables” on page 61, we need to know the number of books by a specific item type and author. In addition, this pattern provides the ability to use the stored results to rapidly obtain derived results. In the library example, if we stored the number of items by item type and author, we could have 3 books by Eric, 1 video by Eric and 2 books by Mack. Then we can ask for the number of items by author and, instead of having to recalculate, this can be derived from the existing stored results. We would then have 4 items by Eric and 2 items by Mack.

6.3.7.3 Benefits

The Cached Balances pattern provides a mechanism that can be used to provide caching of dynamic information in a flexible way. It allows the classical trade-off of speed versus storage to be deferred to the appropriate level, application or end user. It also provides a means for leveraging the stored results to achieve further derived results.

6.3.8 Extensible Item

6.3.8.1 Intent

In certain circumstances, the behavior and/or attributes of an object need to change dynamically, such as when its state changes. For example, an employee's responsibilities can change from clerk to supervisor status. This change in responsibility affects behavior of the employee. For example, as a clerk, the employee could not terminate another employee, but as a supervisor, they could.

6.3.8.2 Concept

The Extensible Item pattern defines a way to dynamically add, remove, or modify methods and/or attributes of a particular object. The pattern also supports temporarily overriding a method. This is done by keeping a stack of the different version of the method (with the same name), allowing only the top one to be used. For example, if we have a method `print()` and we add an extension with a new version of `print()`, the original version of `print()` will be pushed down and the new version put on the top of the stack. When the `print()` method is invoked, the new version of the `print()` method will be used. If the new version of `print()` is removed, then the previous version is made available for use.

6.3.8.3 Benefits

The Extensible Item pattern enables the definition of objects that undergo extensive changes in their behavior. Because this pattern partitions the behavior into extensions, the implementation of such objects is simplified and their extensibility and clarity are enhanced.

Chapter 7. San Francisco Utilities

San Francisco's utilities are common services needed by most applications. For users, they present a graphical interface to set up San Francisco's functionality or to connect to the user's legacy database. For programmers, they define a set of APIs to interact with the framework setup.

This chapter introduces the utilities provided with San Francisco. It contains the following sections:

- Section 7.1, “Configuration and Server Management Configuration” introduces the Server Management Configuration mechanisms provided by San Francisco and how a logical San Francisco network is defined.
- Section 7.2, “Security Configuration” on page 72 introduces the Security Configuration mechanisms provided by San Francisco.
- Section 7.3, “Conflict Control” on page 75 introduces the Conflict Control service provided by San Francisco.
- Section 7.4, “Print Utility” on page 75 introduces the Print utility provided by San Francisco.
- Section 7.5, “Schema Mapping Tool” on page 77 introduces a San Francisco tool that allows you to use a relational database as the persistent store for San Francisco business objects.

7.1 Configuration and Server Management Configuration

San Francisco, as a distributed framework, allows multiple server processes to manage different tasks. Server processes may run on the same computer or be distributed across several computers while working together to implement San Francisco's functionality.

7.1.1 The Logical San Francisco Network

A Logical San Francisco Network (LSFN) is a group of client and business object processes that share the same instance of the Global Server Manager (GSM) process. Processes in an LSFN can be physically located on separate computers or reside on the same computer. TCP/IP is used to communicate between the various San Francisco processes. In the same TCP/IP network, multiple LSFNs can coexist (although they do not interoperate).

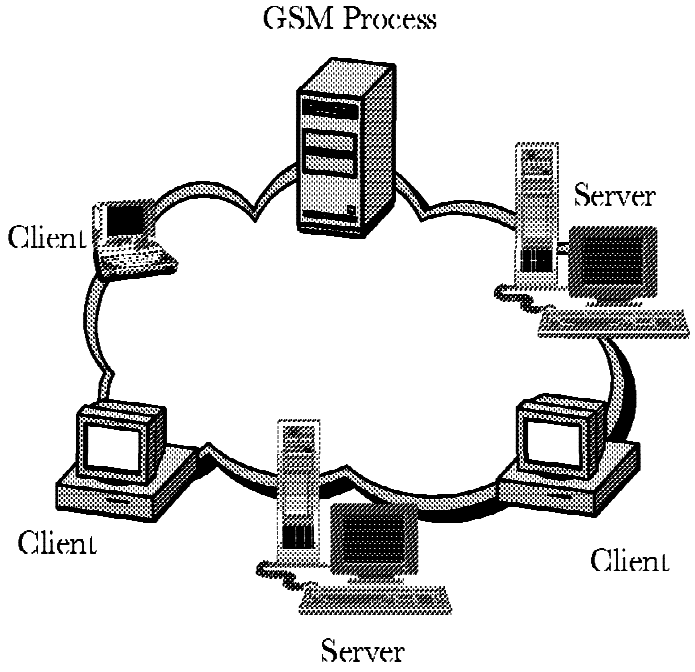


Figure 20. A Logical San Francisco Network

The following roles are found in one LSFN:

- Exactly one Global Server Manager (GSM) Process
- Zero or more Business Object Processes (BOPs)
- Zero or more client processes (although a network without clients makes no sense).

The GSM Process contains several service objects that are critical to LSFN functioning. These service objects are:

Global Server Manager (GSM) Service
This service is in charge of all the Business Object Processes (BOP). It holds all of the BOP configuration information, tracks all active and inactive BOPs, and is responsible for connecting clients with specific BOPs.

Global Distributed Process Manager (GDPM) Service
This service tracks all active clients in its LSFN. It must be running before clients can enter the LSFN.

Global Name Server (GNS)
This service is responsible for managing the naming space of the LSFN. The component knows:

- How to resolve the user aliases
- Which classes ultimately have to be used to create instances of business objects.
- Where these instances must be created
- How containers are configured (see Section 7.1.3, “Container Configuration” on page 69).

- How security is configured (see Section 7.2, “Security Configuration” on page 72).
- What commit protocol is in use (see Section 10.2.1, “Transaction Model” on page 102).

Other services that exist in the various BOPs are:

Local Server Manager Service
It is responsible for starting BOPs on first request. One instance of this service must be on each computer. It exists in one of the following processes:

- GSM Process of the Global Server computer
- SFSM Process for any other computer having BOPs

Security management is handled by security-controlling services:

Master Security Controller
This service manages authentication for the LSFN. It exists in the GSM Process.

Server Security Controller
This service authenticates client requests. It exists in each BOP as well as in the GSM Process.

Business object instances are managed by the following services:

Factory Manager
It is responsible for creating and holding instances of San Francisco business objects.

Transaction Service
It monitors transaction processing and coordinates recovery. One Transaction Service exists in each BOP that uses transactions (usually the process that holds a Factory Manager).

Additional administrative functions are implemented in the last two services:

Problem Service
It's duty is problem logging for internal services. There is no problem-recording interface for application programs.

Conflict Control Service
It manages conflict-controlled actions (see also Section 7.3, “Conflict Control” on page 75).

Each of the last two services may run in any BOP, but only once in an LSFN.

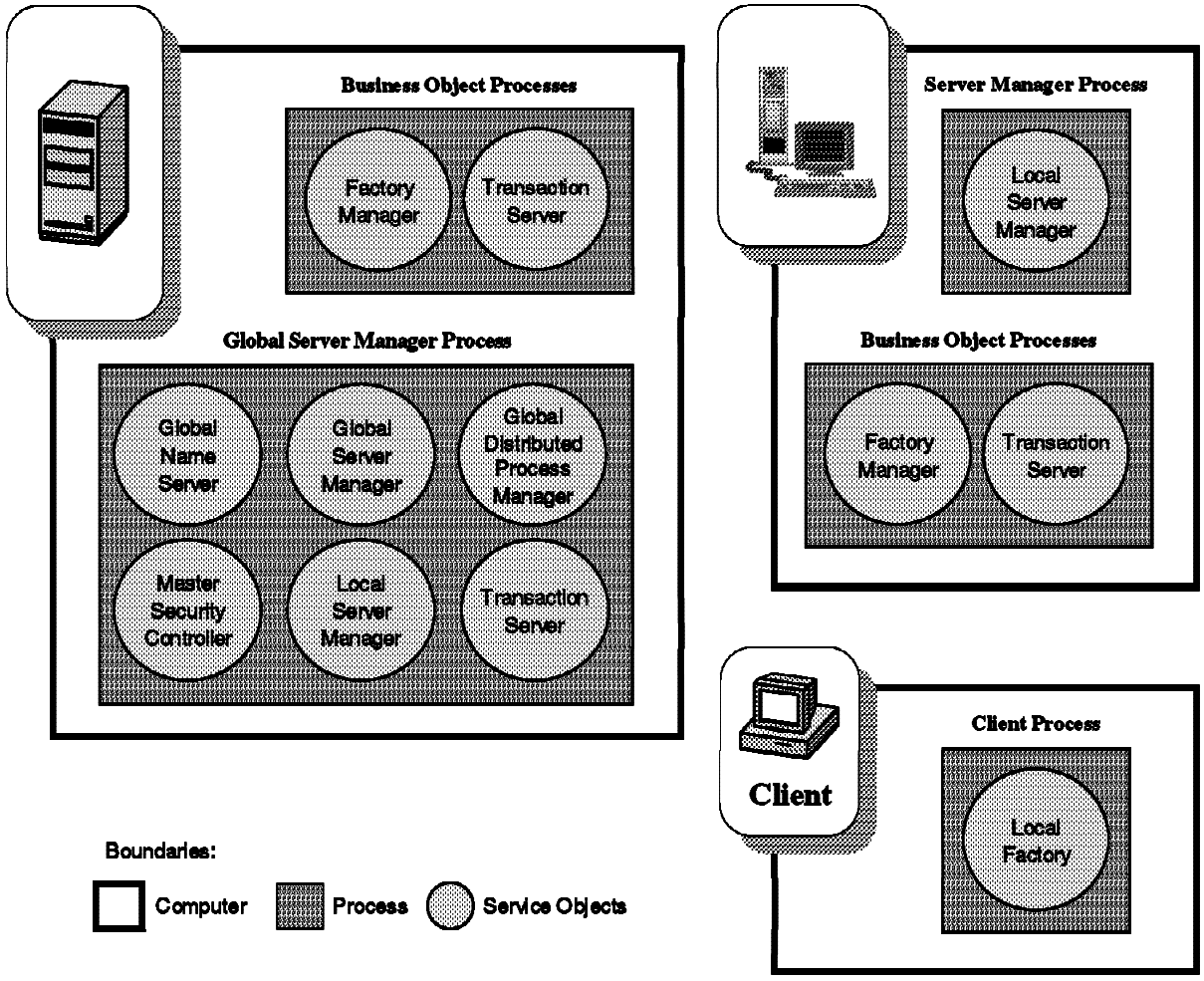


Figure 21. Processes and Services in an LSFN

7.1.2 The Server Management Configuration Console

The Server Management Configuration Console is used to configure servers in the San Francisco logical network. The administrator defines:

- Which hosts the BOPs will run on
- Which BOPs the services will exist in
- Which specific settings should be used when a BOP or service is invoked

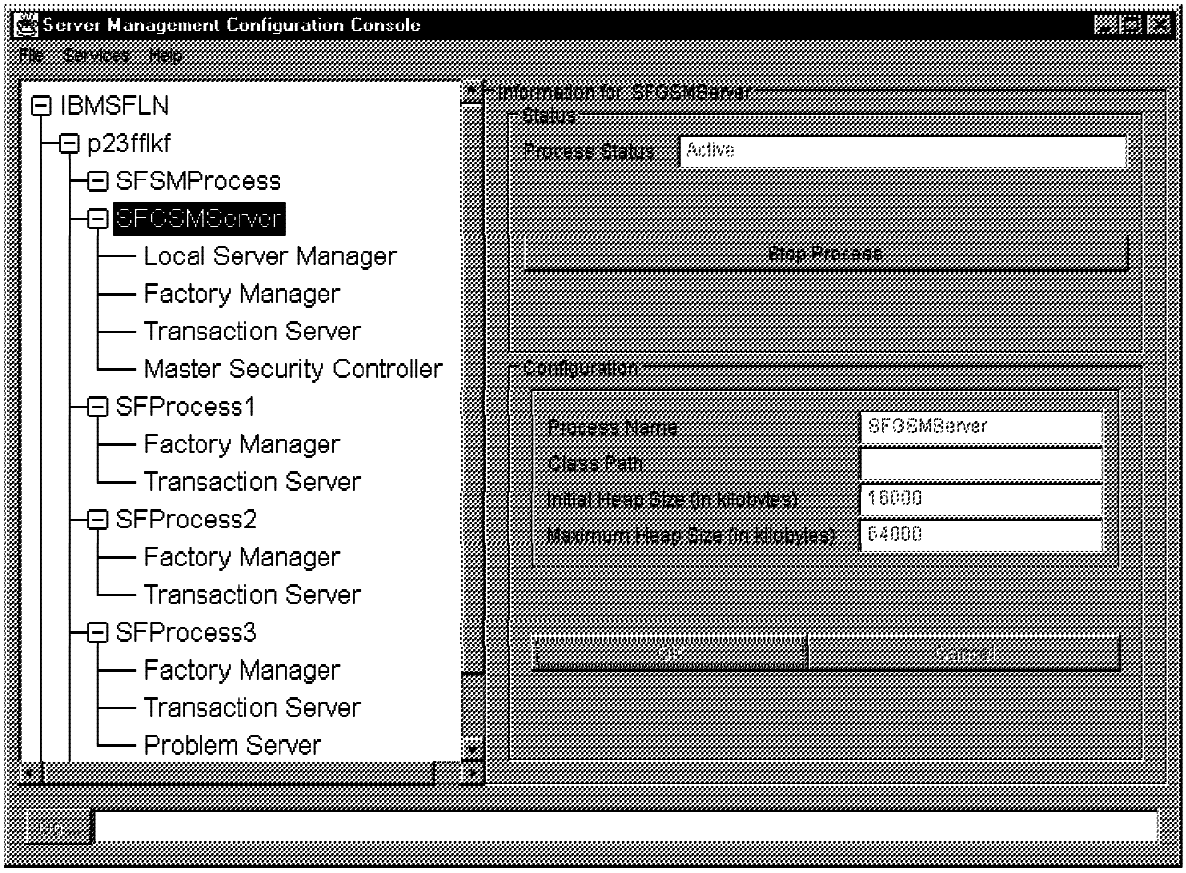


Figure 22. The Server Management Configuration Console

7.1.3 Container Configuration

San Francisco makes persistent object location transparent to business object and application developers. The location of persistent business objects is configured by placing them in business object containers. Developers are not required to know where business objects reside. Instead, this task is left to an administrator who has the knowledge about the underlying data store and overall network configuration.

A business object container is a collection of business objects used by San Francisco internally. Each Entity is held in exactly one container. The administrator defines the following for a container:

- The Factory Manager in which the container resides
- The data store where the container's objects are to be stored

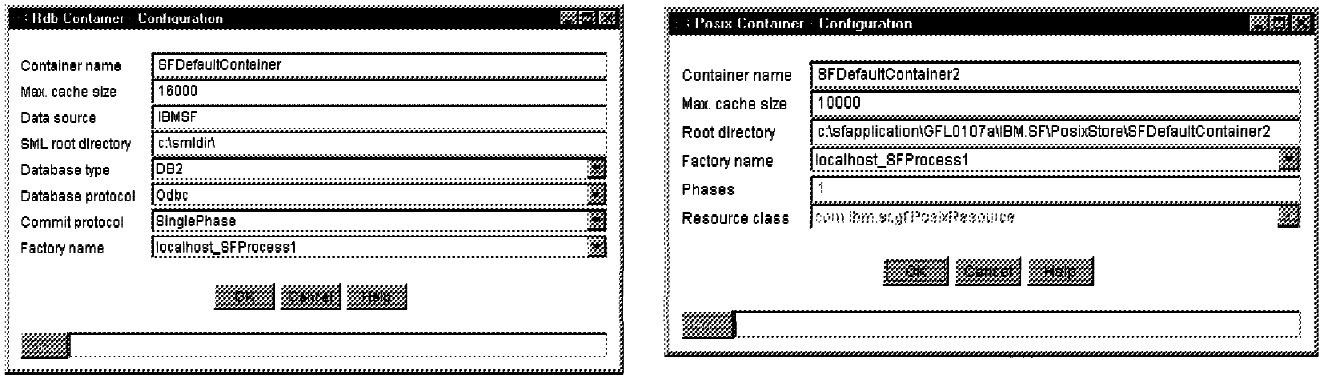


Figure 23. Container Configuration

Currently, two types of containers are supplied:

RDB Containers

They store Entities in a relational database using either ODBC or native database.

Posix Containers

They store Entities in a Posix file. Each Entity instance is a separate file. These containers are not recoverable; nor is any claim made to their scalability. It is presumed that these containers will be used in a testing environment *only*, or as a boot-strap to get San Francisco up and running.

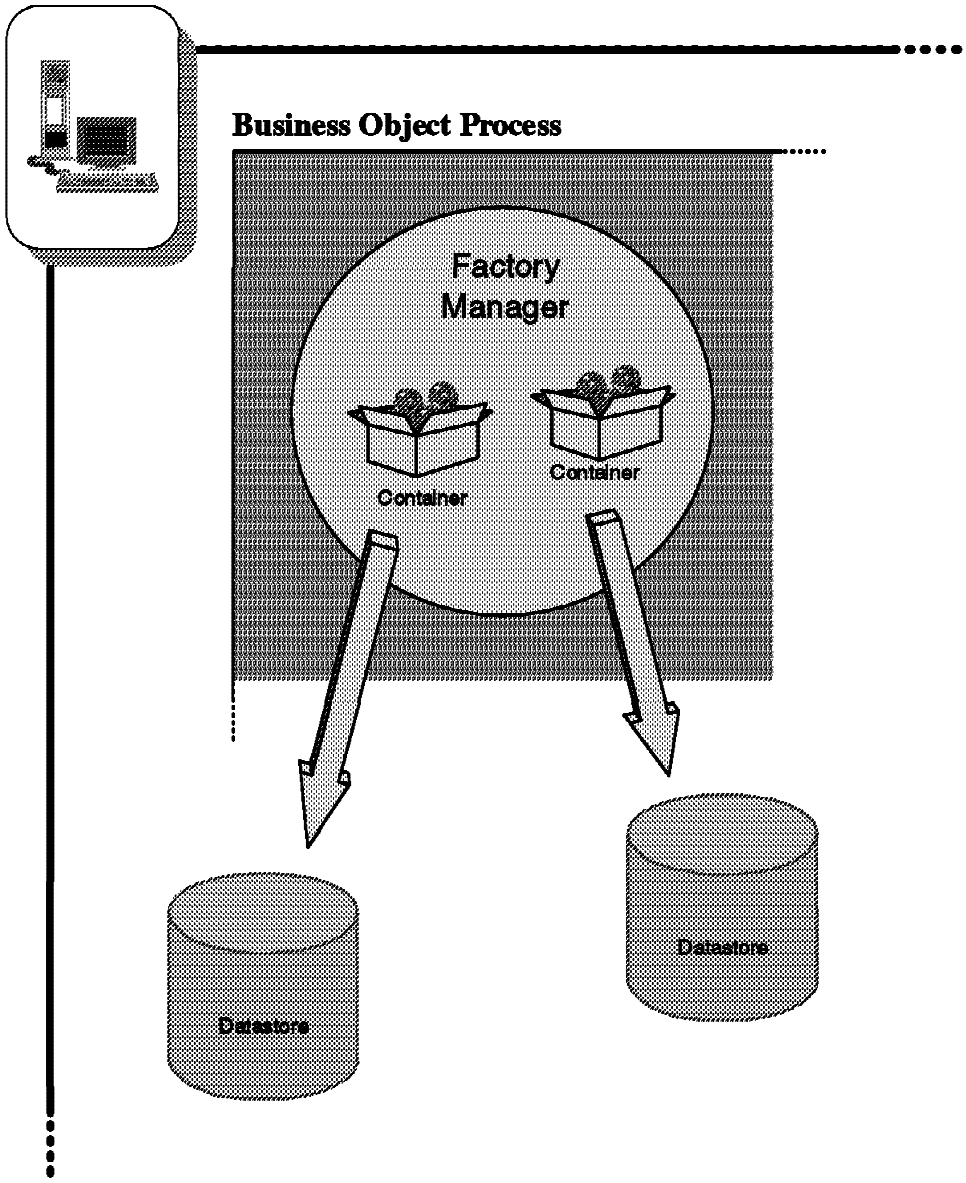


Figure 24. San Francisco Containers

7.1.4 Configuring Entities to Containers

Every container must be configured with the Entity class that the container is capable of persisting. The configuration data that is required is based on the type of container.

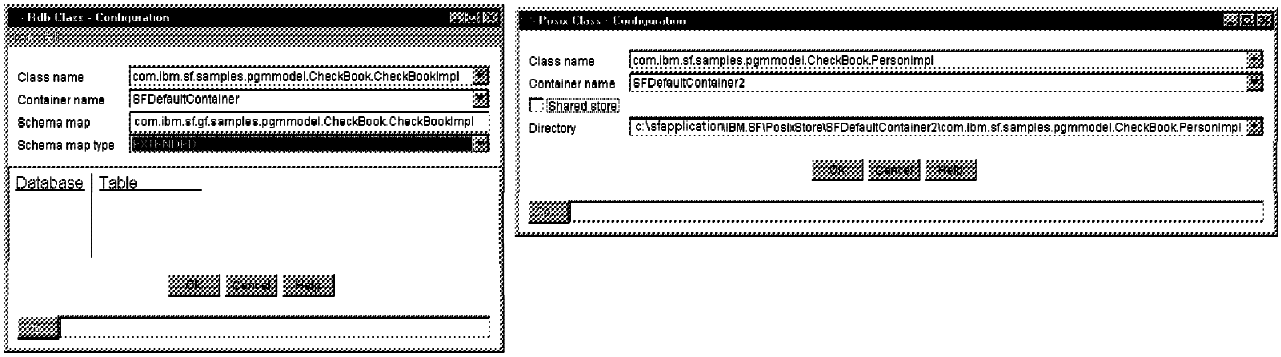


Figure 25. Container Class Configuration

For the Posix Containers class, each Entity class is configured with the name of the directory in which instances of the Entity class will be persisted. Each Entity instance will reside as a separate file in this directory.

For the RDB Containers class, each Entity class must be configured with the type of schema mapping that will be used to persist instances of the Entity class. Default schema mapping can be specified if you want San Francisco to completely manage the persistence, including creation of the tables used to persist the Entity and the definition of the columns of the table. Extended schema mapping can be specified if you want to use the Schema Mapping tool to perform your own mapping of the Entities state to columns of a table.

7.2 Security Configuration

In general, security may be implemented on the following levels:

- Application level
- System level

San Francisco implements security on an application level for the reasons that security administration:

- Is done centrally
- Uses a common interface on all platforms
- Is consistent across all application servers

San Francisco provides the programmer with:

- An easy way to use authentication API to present a fully featured logon window. This is the recommended way of establishing a user's identity for interactive applications. It handles erroneous input as well as expired passwords.
- A set of low-level APIs to implement authentication verification. These APIs are used by the logon window itself, as well as by client programs with specific needs for user authentication.
- An API to define secure tasks consisting of a security-sensitive sequence of actions. This API is typically used by the application's installation routine. The administrator again authorizes individual users or groups of users to execute various secure tasks.

- A pair of APIs to activate and deactivate a secure task around a block of code.

All programmer interaction happens within the *client role* defined by the San Francisco programming model. There is no security functionality implemented in the business object layer.

The administrator defines the security for an LSFN by maintaining:

- Security Policy, consisting of:
  - User ID and password checking (one of the following):
    - None
    - User ID only
    - User ID and password
  - Enabling secure task checking
- Users and user groups
- Access rights, consisting of:

  Actor A San Francisco user ID.

  Resource An identifiable Entity (for example, the Company object set).

  Action What the user is trying to do with the resource (the secure task).

San Francisco provides the administrator with graphical user interfaces to maintain those definitions.

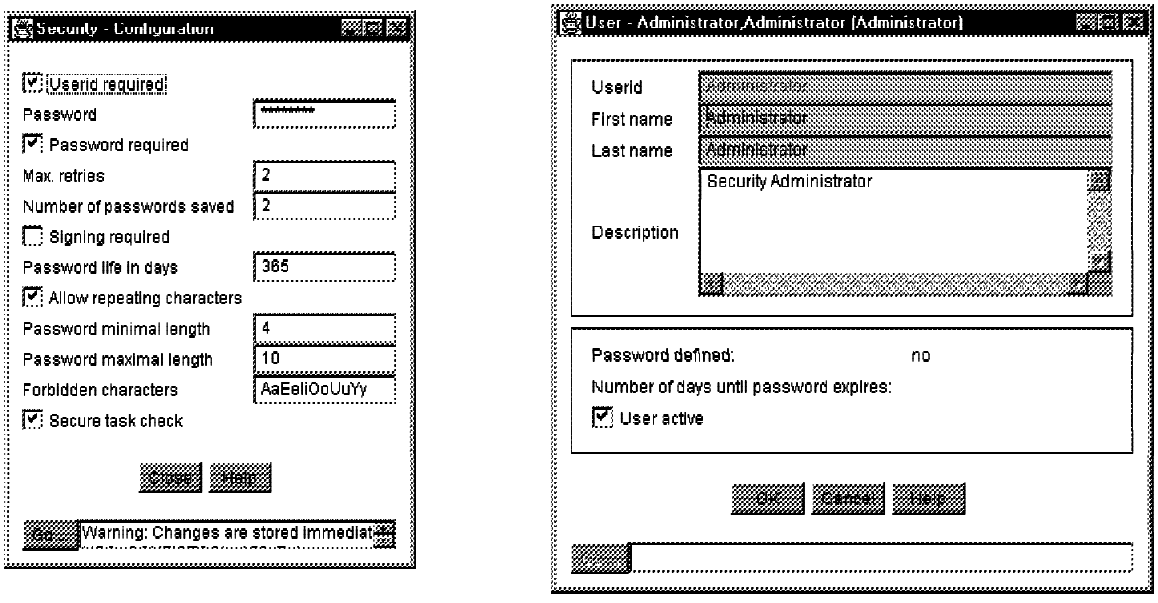


Figure 26. The Security and User Configuration Windows

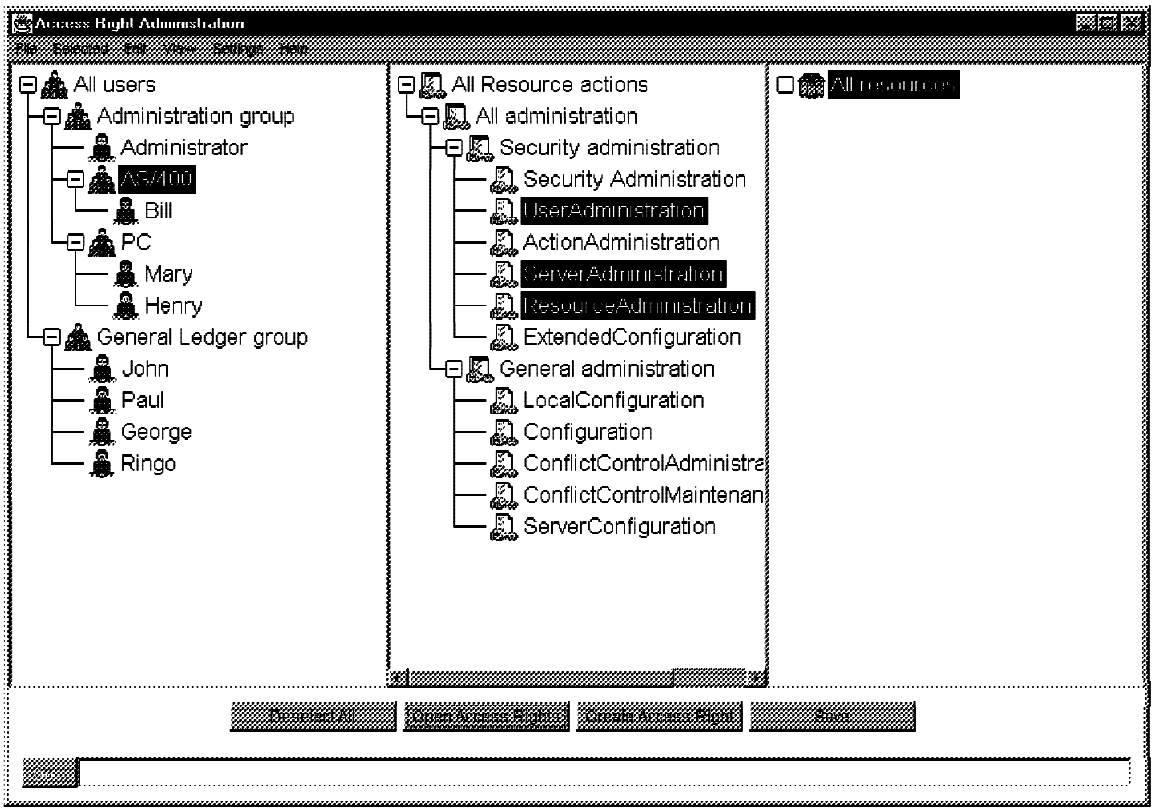


Figure 27. The Access Right Administration Window

7.3 Conflict Control

Conflicts are actions or groups of actions that must not be executed at the same time. The Conflict Control service manages the execution of conflicting actions.

An action can conflict not only with another action or group of actions; it can also conflict with itself. An example is the end-of-day closing that must not be started twice. Parameters can qualify actions (for example, the end-of-day closing for company A may be started while the end-of-day closing for company B is running).

San Francisco provides two ways of defining conflicts:

- Interactive using the Conflict Control Administration utility
- Batch using conflict definition files

The interactive Conflict Control Administration utility is used at the application installation site to review the shipped conflicts and to refine them. At a development site, this utility is also used to define conflicting actions.

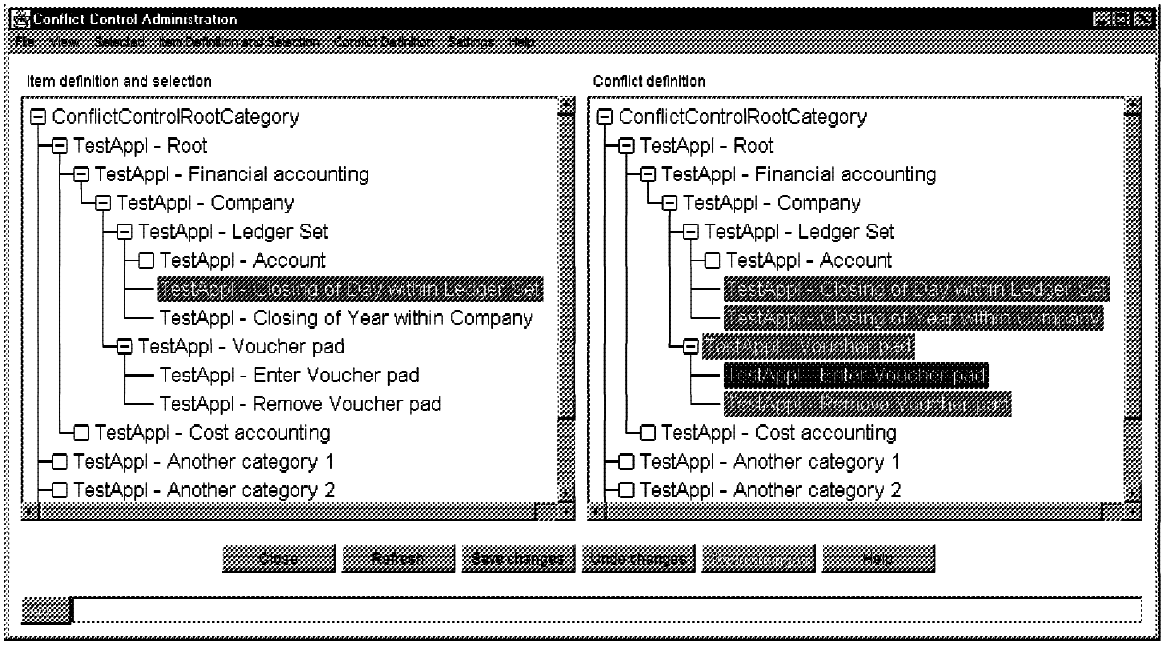


Figure 28. The Conflict Control Administration Utility

The Batch utility is used during application installation to define actions, action groups, and conflicts. It reads the definition files provided by the application and creates the appropriate action and conflict objects.

7.4 Print Utility

The Print utility allows creating documents that contain business data such as orders, invoices, or form letters. Data and presentation are separated by having static data and design information kept in the layout description, while variable data is stored in business objects only.

The Document Designer is a graphical WYSIWYG (What You See Is What You Get) layout editor to create print forms. It presents application object information to the user and allows the user to arrange it on the page using drag-and-drop. To collect information about the business object structure, the Document Designer calls methods described by the San Francisco `PrintableInfo` interface. This interface can be implemented by the business classes themselves, or by a supposing print class, which in turn accesses the business class for the correct data.

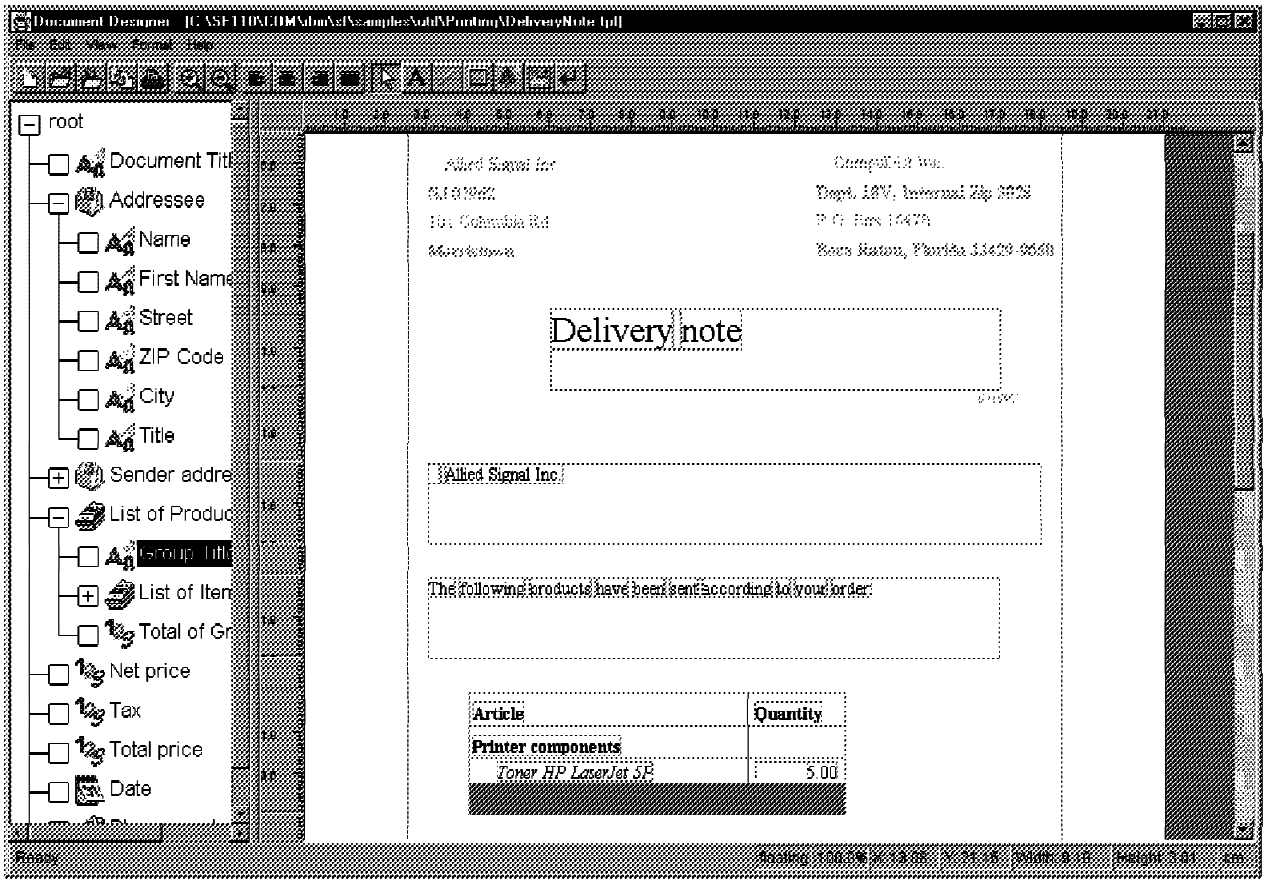


Figure 29. The Document Designer

The Print Formatter merges application data with a layout description and produces a PostScript file. Java Development Kit (JDK) based print support is planned. For printer management, an external tool has to be used.

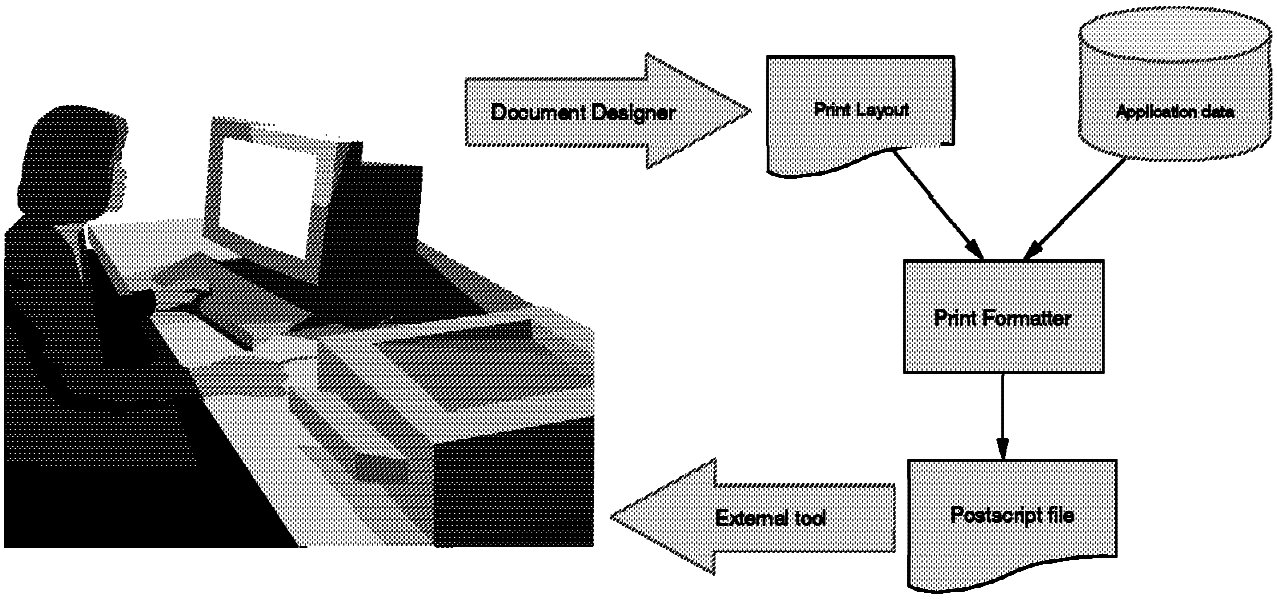


Figure 30. The Print Formatter

Documment Designer and Print Utility

Document Designer and Print Formatter are not stand-alone tools. They are San Francisco utility classes that can be used by applications to provide layout and print support. Thus, Design and Print menu items are not provided by San Francisco itself. If available, they are provided by individual applications that use those utility classes.

7.5 Schema Mapping Tool

To use a relational database as the persistent store for San Francisco business objects, a mapping between the object model and the relational model is needed. Schema Mapping means defining the mapping between an object schema and a relational schema; in other words, classes, objects, and attributes are mapped to tables, rows, and columns.

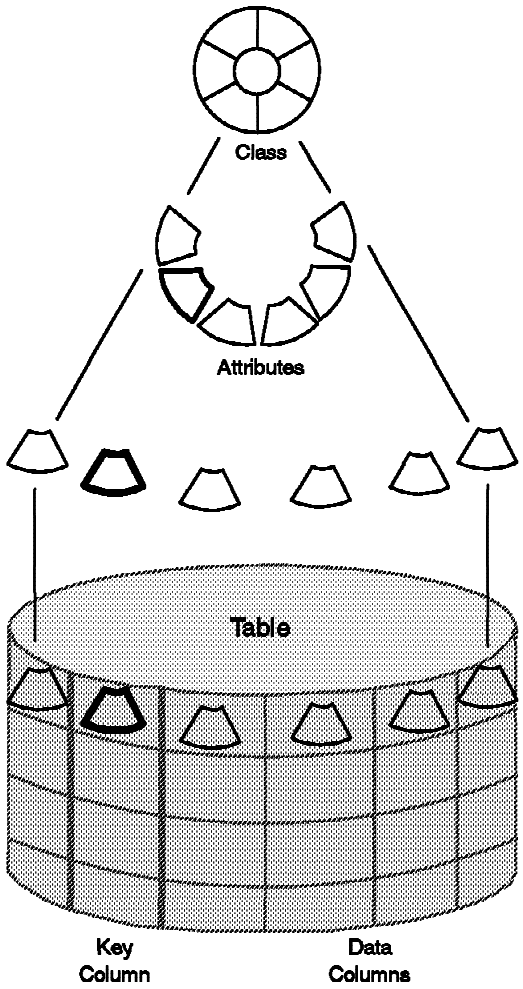


Figure 31. Schema Mapping

San Francisco provides the following varieties of Schema Mapper:

Default Schema Mapper (DSM)

The DSM is intended primarily for use in prototyping. It automatically maps a class to a table without input from the programmer. The table is defined at run time using a general set of rules. The class layout is obtained by parsing the class definition at run time. The DSM is supported only by the ODBC container. The advantage of using the DSM is that no schema mapping has to be configured.

Extended Default Schema Mapper (Extended DSM)

The Extended DSM obtains its information from a Schema Mapper Language (SML) file. The SML file is created using the Schema Mapping Tool (SMT), a graphical user interface (GUI) that allows the programmer and the administrator to name the table, columns, and specify column data types. The Extended DSM is only supported by the ODBC container. Using the Extended DSM, existing objects can be mapped to existing tables. The advantages of the Extended DSM are:

- Existing data stores, naming conventions, and conversions can be specified.

- The disk capacity for storing objects may be smaller since the size of data columns may be customized.

A variety that will be available in the near future is the:

Custom Schema Mapper (CSM)

The CSM provides no run time code. Instead, code that implements the mapping between classes and tables has to be written by the programmer. The CSM is supported by the ODBC container for which C code has to be implemented. This code has to be compiled and published with the application after each change to the mapping configuration. The advantages of the CSM are:

- It is as flexible as the Extended DSM
- It is faster since no SML file parsing has to be done at run time

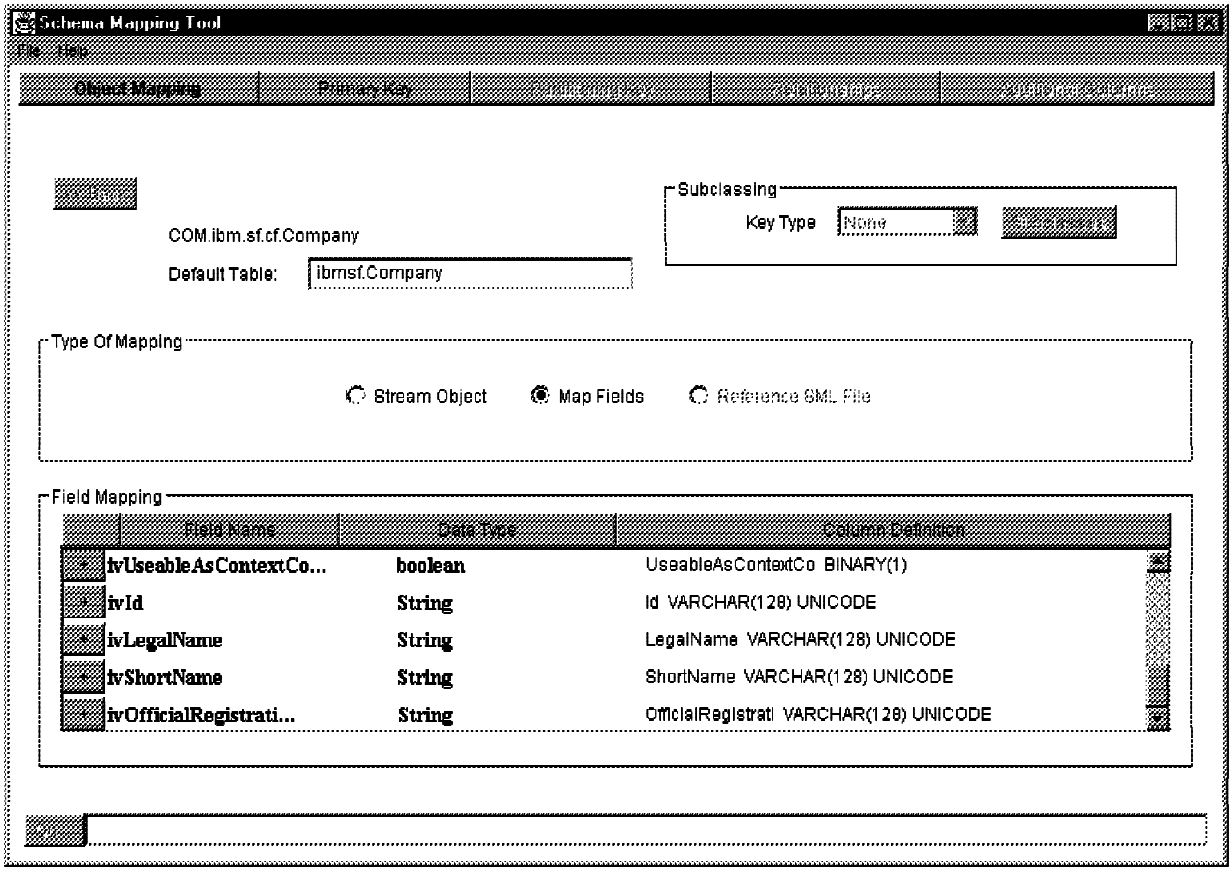


Figure 32. The Schema Mapping Tool

Chapter 8. San Francisco Application Development Methodology

This chapter provides insight into developing an application using San Francisco. Included is a description of how San Francisco has chosen to represent and structure the business processes provided for a domain. By reading this chapter, you will understand the general approach to be followed by developers using a San Francisco framework to implement their application.

The chapter covers the following topics:

- Section 8.1, “Representing Business Domains in San Francisco” explains how business domains are represented in San Francisco.
- Section 8.2, “San Francisco Development Approach” on page 82 explains the main approach used to define the capabilities provided by a San Francisco framework.
- Section 8.3, “The San Francisco Roadmap” on page 84 introduces the *San Francisco Roadmap* document, which guides you through the process of building solutions based on San Francisco.
- Section 8.4, “Building the San Francisco Framework Based Application” on page 87 refers to the *San Francisco Extension Guide*, which introduces the basic set of approaches for extending the frameworks.
- Section 8.5, “Integration with Legacy Applications” on page 90 covers the crucial point of integration with existing systems. It has been demonstrated that the success of a new technology is related to its ability to integrate preceding technologies. This point has not been lost on San Francisco, and more than one alternative is available.
- Section 8.6, “Plans of Transition to San Francisco” on page 92 suggests a plan for moving an organization to San Francisco.

8.1 Representing Business Domains in San Francisco

A San Francisco framework captures the core capabilities required to support a particular domain. San Francisco represents these core capabilities as business processes, embodying the core activities to support a particular domain. A San Francisco Business Process can be defined as a main business event occurring within a particular business domain. San Francisco Business Processes are composed of a set of related business tasks, each having the objective of producing an intended outcome. A business task is an elementary business activity having a clear triggering event and result.

An example business process within the San Francisco General Ledger Framework is Posting to the General Ledger. The Posting to the General Ledger process provides support for making various types of journal entries (composed of dissections) into the General Ledger. It also supports the updating of General Ledger balances as specified. The Posting to General Ledger Process is composed of various tasks. Two of those tasks are Suspend GL Journal (which supports suspending GL Journals if necessary) and Post GL Journal (which supports the validation of the journal and the updating of the GL balances if appropriate).

San Francisco decomposes these business processes and their tasks into application scenarios which are a set of specific uses or executions of a particular business task. The scenarios are refined through the development process to represent the implementation required to complete a specific execution of its related task.

Once the domain information (in the form of processes and tasks with scenarios) is collected, formalized, and reviewed, an object model is created. First, an analysis object model is created which evolves iteratively during the course of development to a design object model. The object model is composed of class categories. Some categories are generalized capabilities which span various processes, however most of the categories represent a specific business process.

The San Francisco analysis model category usually reflects the business objects involved in a particular business process. The design model category reflects the business objects central to a process as well as the objects necessary to support the specific process.

8.2 San Francisco Development Approach

San Francisco supports an iterative development cycle, as is customary with object-oriented technology development. There are a variety of entries into the San Francisco development process. These entries depend on what level of investigation has been done by the ISV in defining the application from an OO perspective. The entries are discussed later in this chapter. To reap maximum benefits with San Francisco, it is important to use as much as possible of the San Francisco methodology. It is important to stress that San Francisco did not invent a completely new OO development methodology, but rather San Francisco extends the standard OO approach where necessary.

San Francisco documents its development approach via the San Francisco Roadmap. This document comes with San Francisco and guides an ISV through the development cycle. It provides a set of standard templates used by San Francisco to document a domains' business processes, tasks and scenarios. By using these templates an ISV can readily adhere to the San Francisco approach for documenting their application design. The San Francisco Roadmap also has links to specific examples of creating portions of a GL application, documented according to the San Francisco Roadmap.

It is important to note that there are various ways of developing applications with San Francisco. These ways are:

- Build applications with the San Francisco Foundation
- Build applications with the San Francisco Foundation and the Common Business Objects (CBOs).
- Build applications using a San Francisco framework for a specific domain
- Build an additional San Francisco framework

The San Francisco Roadmap focuses on building an application using a San Francisco framework. There is work being done to document the other ways and there are courses/education available which facilitate the other types of development.

8.2.1 San Francisco Application Development Team

The application development team requires a mixture of skills and capabilities. The main team roles are:

- Domain Experts with knowledge of the specific domain
- OO Designer(s) with sound OO modeling and design skills
- Programmers with OO knowledge and skills in Java

With some effort on the various team members part, they can become a single working unit. They can use the activity of understanding a San Francisco framework as a base point for allowing the various members to bring their own particular knowledge to the team.

8.2.2 San Francisco Development Cycle

The San Francisco development cycle consists of four standard activities, which are:

- Collect and Document the Requirements
- Perform Analysis
- Perform Design
- Generate Code and Test

These activities are standard to any OO methodology. However, as we stated earlier San Francisco has documented their requirements as processes and tasks. If the ISV documents their requirements as processes, the incorporation of San Francisco function will be easier.

For ISVs developing applications using either the CBOs or a framework, an additional activity is necessary. This activity, known as mapping, provides the development team a means to relate their application requirements to the function provided by San Francisco. The mapping process should occur as part of the first three activities listed above.

The results of the mapping activity performed subsequent to a requirements definition will be rather high level. As development continues and the application becomes more completely defined, in the analysis and design stages, the mapping to San Francisco will reflect a more detailed utilization of San Francisco function. It is recommended that at the end of both analysis and design the mapping activity is performed.

Mapping is defined as the act of reviewing your application requirements (processes and tasks), scenarios and model with San Francisco. Mapping to the San Francisco application scenarios, which detail the activities performed for a particular task within a process, is the suggested approach. The results of the mapping process should be recorded. San Francisco provides a Mapping template, which allows the ISV to capture the results of the mapping exercise. The ISV can use this information aid in defining the amount of coverage San Francisco will provide as part of their application and can use this information to help in the project management activity for the project.

8.3 The San Francisco Roadmap

The San Francisco application development Roadmap is a guide for developers who intend to use a San Francisco framework as the basis for application development. The Roadmap guides you through the development cycle as shown in Figure 33. It guides you through a structured set of activities and provides templates for collecting your application information. The templates assume you are using the process, task, scenario approach which was discussed earlier. The Roadmap provides links to sample applications implemented using the Roadmap templates.

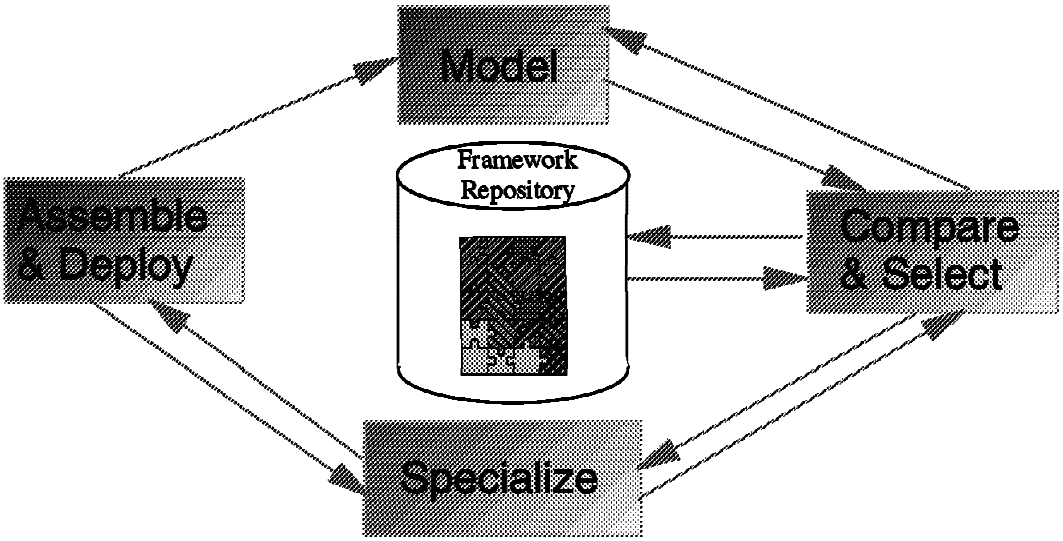


Figure 33. Development Cycle of San Francisco

Each activity in the development cycle has its own set of recommended deliverables. The development cycle steps are:

- Collect and Document the Requirements
- Perform Analysis
- Perform Design
- Generate Code and Test

You should add an additional mapping activity as part of the first three major cycle steps.

8.3.1 Collect and Document Requirements

The San Francisco Roadmap has identifies three general types of requirements to collect.

1. **Application Requirements**

   Application requirements state the portions of the business domain which are encapsulated in the application. The application requirements can be collected various ways. The Roadmap suggests using either:

   Process Modeling
   Defining your business processes.

Use-Case Modeling

Defining the key actors in the domain and how they do their work. For more details, refer to reference 4 on page 117 in B.3, “Other Publications” on page 117.

Once the initial requirements collection activity is completed, the Roadmap suggests the requirements be formalized as processes with tasks. The use-cases can be grouped and restructured into processes with tasks, while the process modeling approach needs to be formalized as processes with tasks.

2. **Technical and Tooling Requirements**

   The technical and tooling requirements include both the development tools and the deployment technology required. This information aids in the planning, budgeting and education plans for the development team. This information is also used as part of the application design considerations.

3. **Implementation Requirements**

   The implementation requirements view the development of the application from the customer/end user perspective. Things included here range from deployment considerations to end user education/documentation and application roll out.

These various requirements, if rolled together and used appropriately throughout the development process, will increase the likelihood of the project's ultimate success.

The exit criteria stated in the Roadmap for this cycle step is an agreed upon set of requirements (preferably written as processes with tasks) and a list of application scenarios for each task within a process. An application scenario states the specific executions of a task. The application scenarios list all the executions or usages for each process task within the application.

8.3.1.1 Mapping Requirements

The mapping activity allows the application development team the opportunity to map their requirements to San Francisco. This is a first pass at the mapping process, which should occur again as part of analysis and design. The Roadmap suggests using the application scenarios as the primary vehicle for the mapping process. This initial mapping exercise will yield a high level match which can be used to indicate the amount of coverage San Francisco provides for your application. As development continues and more detail is provided about the application, the mapping will be able to explicitly state the detail function within San Francisco that applies to the application.

The Roadmap provides a Mapping template which allows the development team a means in which to record the results of the mapping process. As development continues, this document should be revisited and refined to reflect the details of the application and the function provided by San Francisco.

8.3.2 Perform Analysis

The analysis step refines the information about the application by supplying more details about the users activity and the business logic required to support the domain. An analysis object model is created which identifies the Domain objects and their static relationships.

The major emphasis of this step is to provide the details of the application scenarios. The Roadmap provides a scenario template which allows the developers to record each step required to complete the execution of a scenario. Where the application is using San Francisco function, the application scenario will reference the San Francisco application scenario that provides the needed capability.

The business application is responsible to provide all the user interaction with the application. This user interaction takes various forms, which includes User Interfaces, Reports, and addition processes not provided by the San Francisco framework. The user interactions are defined during this step. User Interface prototypes and report layouts are considered part of this step, even though all the details may not be completed.

The analysis object model and the analysis level application scenarios are the primary exit criteria for this step. Analysis level object interaction diagrams would be of benefit, even though they are not required.

8.3.2.1 Mapping Analysis

While detailing the application scenarios, the development team may want to look at the function provided by the San Francisco framework. The suggested approach is to map the scenarios to the San Francisco framework after the scenarios have completed. The key point is to perform the mapping process and continually review the San Francisco framework to determine how it can aid in the development process.

8.3.3 Perform Design

The goal of the design process is to add the implementation specifics to the analysis results. The implementation specifics include:

- Extending the Analysis Model with design details
- Refining the Application Scenarios with all necessary implementation details
- Refining the User Interactions (UI and Reports)

A design model, based upon the analysis model, is extended to include various design specific classes, relationships and methods. San Francisco specific classes are incorporated which facilitate both the utilization of San Francisco capabilities and the subsequent code generation process. This activity can require the restructuring of the model to meet the technical architectural needs.

Each analysis level application scenario is enhanced to handle various aspects of the actual implementation. These aspects include details on validation, exception handling, and method calls on various objects. Along with application scenario detailing activity, object interaction diagrams should be created for every significant scenario. This ensures the completeness and verifies the design.

The design step defines both how the San Francisco framework will be extended to meet the application needs and how the new application will interact with the legacy application, if one exists. These two topics are discussed later in this chapter. It is important to recognize these activities as part of the design process.

The exit criteria for this step is a detailed description of each application scenario, well defined user interfaces and reports, and a complete design model

including object IDs as needed. This information must be mapped to the San Francisco framework to ensure the framework has been used to its fullest extent.

8.3.4 Coding and Testing

The last step documented in the Roadmap is code and test. In this step, the design model is used as input to the code generator, which generates San Francisco Java code. The code generator uses the static model as input to create the various San Francisco objects. The developer must add the business logic behavior to the generated code. The developer is also responsible for creating the user interface code. During this step, you must develop a test plan and carry out the plan against the completed code.

You must understand the San Francisco execution environment to complete the testing process. This is covered in Chapter 7, “San Francisco Utilities” on page 65.

8.4 Building the San Francisco Framework Based Application

A San Francisco framework provides the core processes for a particular domain. To provide a complete and robust business application, the ISV must both extend the function with the framework and add new function. Each application builder must extend the San Francisco framework with the application specific needs (for example, country, company or industry specific needs). San Francisco has defined an approach to accomplishing this activity in the *San Francisco Extensions Guide*. There are two major types of extensions to a San Francisco framework:

- Including New Function
- Extending San Francisco Framework Classes

8.4.1 Including New Function

San Francisco uses class categories as a means to subdivide a framework into smaller, manageable units. Because a San Francisco framework provides only the core processes within a particular application domain, it is expected that the ISV will create new class categories. The Java code generator for these class categories is shipped as Java packages along with the San Francisco packages. In the *San Francisco Extension Guide*, guidelines and rules are defined which detail the steps to identifying and integrating new class categories and packages with a framework to create the complete business application.

8.4.2 Extending San Francisco Framework Classes

San Francisco has provided various mechanisms to extend the function provided by a framework. These mechanisms (called extension points) are usually represented as implementations of San Francisco patterns (covered in Chapter 6, “San Francisco Patterns” on page 55). These extension points isolate the areas within the framework that need to be customized. Extension points are specific classes or groups of classes where the behavior of the framework can be easily modified through well-defined approaches. By taking advantage of these extension points, you will ensure:

- Consistency within your applications and with other San Francisco applications.
- Interoperability with other San Francisco applications

- Isolation of framework changes made by ISVs to a limited number of classes to make maintenance easier and allow upward compatibility.

Each extension point is identified in the San Francisco design model with a prefix to the class names which identifies the particular type of extension point.

The *San Francisco Extension Guide* supports extending framework classes using these basic approaches:

- Subclassing to Add a New Class
- Subclassing to Replace an Existing Framework Class
- Extending a Framework Class through Aggregation
- Extending a Framework Class through the Use of Properties

8.4.2.1 Subclassing to Add a New Business Object Class

When the application developer is unable to find an existing framework class to suit the application needs, *subclassing to add a new business object* is one way to satisfy that need. This entails either deriving a new class from a Foundation layer base class or from a domain class defined within the framework. These application defined subclasses fall into two primary categories:

- Classes introducing new function (methods and attributes) not related to classes provided by the framework.
- Classes adding function (methods or attributes) to or modifying methods provided by the framework.

Classes introducing new function not provided by a framework class must be derived from one of the Foundation layer base classes. The characteristics of the new class must be mapped to the appropriate Foundation layer base class and the specific implementation details applied.

Classes adding function to or modifying methods provided by the framework require the developer to determine the origin of the class (to be subclassed). The class could be either a framework class or a coexisting application class. In either case, the developer must evaluate the situation and follow the approach defined in the *San Francisco Extension Guide*.

8.4.2.2 Subclassing to Replace an Existing Business Object Class

When building an application using a San Francisco framework, it is natural to have situations where the developer wants to change the behavior of a framework provided class. This is accomplished by using the Abstract Factory pattern, which is implemented as part of the Foundation's Base Factory. The Abstract Factory pattern is discussed in Chapter 6, “San Francisco Patterns” on page 55. It is important to note that San Francisco restricts class replacement on the CBOs. This is because the CBOs can be used by multiple application providers a given customer might use. This would cause problems because of different implementations of the same business object.

Class Replacement: Allows the framework code and the application code to make calls specifying a class known at compile time, while allowing a derived subclass to be substituted at run time. This is based on the configuration information set through the Base Factory. It is important to keep the interface of the replacement class the same as the original class. Remember that the framework and possibly other applications (unless the developer implements it) have no knowledge of any additional methods that have been added. The

developer must not subclass those classes San Francisco specifies as not available for subclassing.

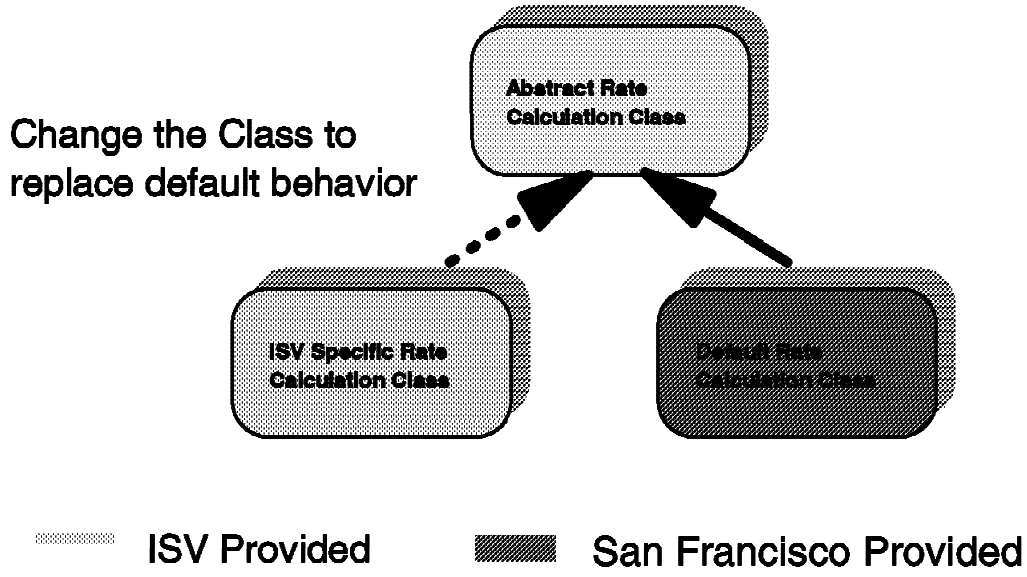


Figure 34. Framework Extension using Class Replacement

8.4.2.3 Extending Framework Classes through Aggregation

A well accepted way to extend the framework is through aggregation, which is containment of one or more framework class instances as attributes of an application class. A common reason for extending the framework through aggregation is to ensure type safety in your application, which applies:

- When using or passing the class
- When working with a framework class with generic interfaces

By using aggregation, the developer introduces a new class into the application. There are various decisions that the developer must make to ensure both the new class and the framework class are implemented correctly for the situation. The decisions are detailed in the *San Francisco Extension Guide*.

8.4.2.4 Extending Framework Classes by Using Properties

Business objects often need to be extended with information that is needed by the application, but which is not included in the pre-built San Francisco class for that business object. Another case is when particular instances of a business object need to have an additional specific piece of information. This information may be an attribute or a relationship in the object model. In San Francisco, either a normal attribute or a property can be used to extend a business object with new information. Attributes are added by creating a subclass of the business object's class. Properties are added to an object at run time and normally do not require modification to the San Francisco domain framework classes. Properties support loose coupling between classes or categories and also multiple application providers extending the same objects.

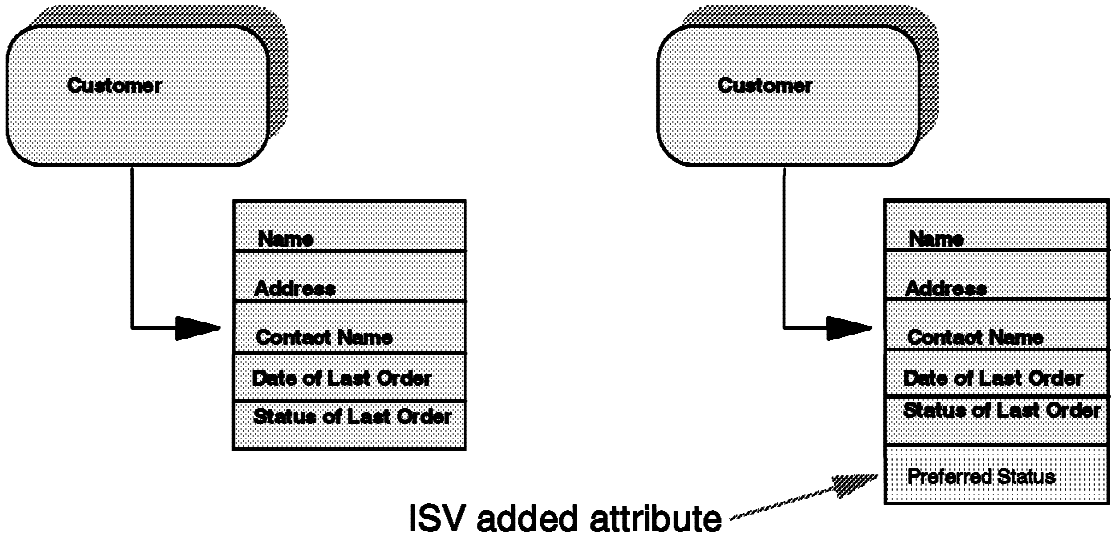


Figure 35. Extending Frameworks through Properties

Any class (that supports the Property Container interface) can have properties added to instances of that class at run time. Using properties can be done without any modification to the interface or class code of the business object. Most business objects in San Francisco domain frameworks support the Property Container interface.

8.5 Integration with Legacy Applications

Most ISVs have a suite of legacy applications that need to cooperate with the new San Francisco based application. It is impossible to replace all of this code at once, so a plan for migration needs to be put in place. The big advantage of the legacy code is that it is robust and tested. There are various options for integrating legacy systems with San Francisco technology. One of the first steps to take is to ensure your existing application is modularized to make the replacement of portions of the existing application doable. Mapping Java objects for interfacing between the "new" application code and the existing modules is an important activity.

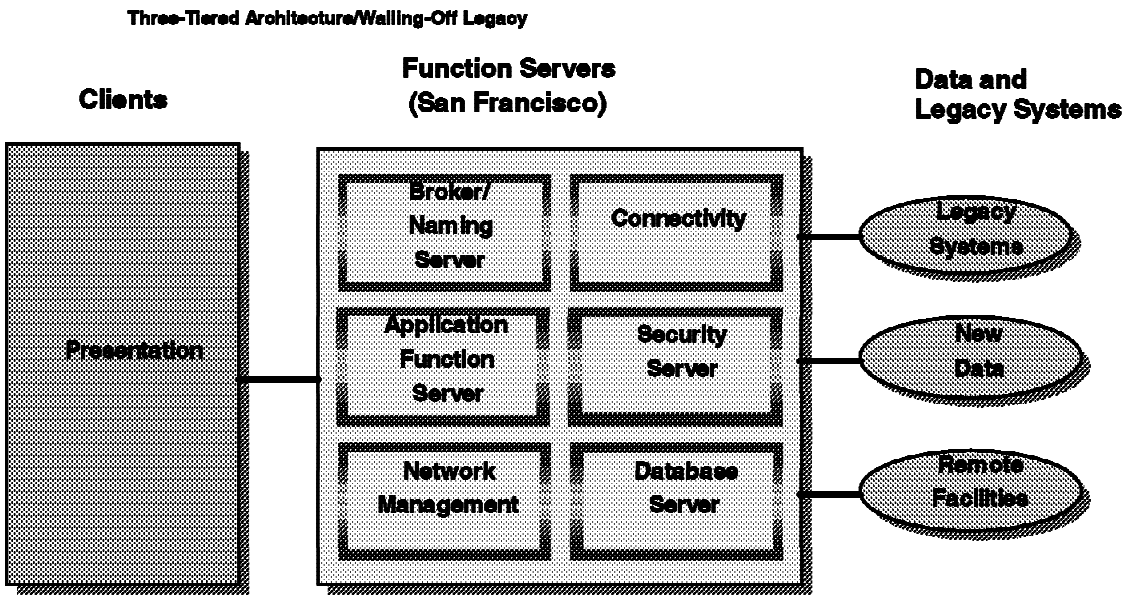


Figure 36. Techniques to Integrate Legacy Applications

It's most important to decide what components must be retained and integrated and what components must be replaced. Make a clear analysis of which modules can be retained and what needs to be replaced. This can help you understand how much time it will take to deploy new applications on top of San Francisco.

Another approach to integrating a San Francisco based application with a legacy application is through the use of a common database. This can achieved by using the Schema Mapping tool to map San Francisco business objects to existing database tables.

8.5.1 Schema Mapping

Schema mapping is the process of mapping business objects to relational database tables. The objectives of schema mapping include *transparency* and *flexibility*. Programmers need not be involved in determining how the objects will be made persistent in a relational table. The mapping is primarily an administrative activity and can be accomplished with a manageable amount of programming. In order to coexist with different legacy environments, the application must be able to persist business objects to different database layouts.

These objectives are fulfilled by the schema mapping support provided by San Francisco. At present, there are two ways to make San Francisco business objects persistent in a relational database:

1. Using the *default schema mapper*
2. Using the *extended schema mapper*

Using the *default schema mapper* only requires you to configure your business object classes to be persisted to a relational database. The San Francisco Foundation will do the rest: it will create the relational tables to contain business objects and will automatically map objects' attributes to relational columns. This

option is only suitable for new application environments, because the default schema mapper cannot adapt its choices to a pre-existing relational database layout.

The *extended schema mapper* allows a much more sophisticated type of mapping. Using the Schema Mapping tool, the developers are able to map each attribute of the business objects to an appropriate database column.

There are advantages in using the extended schema mapper including, being able to map to the legacy table and improving performance considerations.

For more information on the Schema Mapper Tool, see 7.5, “Schema Mapping Tool” on page 77.

8.6 Plans of Transition to San Francisco

When the decision is made to start a new development project using a San Francisco framework, the team must prepare themselves for the task at hand. After acquiring the necessary skills, a first step should be to build a prototype application. Building a prototype is the perfect preparation for building larger applications. The pilot project should have the following characteristics:

- Have a limited scope (about one or two core business processes)
- Incorporate user interfaces
- Implement some application specific business logic

When developing the pilot project, use the Roadmap guidelines and the *San Francisco Extension Guide*, as needed. Using these documents means your team profits from the experience which the San Francisco team acquired during the San Francisco framework development process. Also, your team becomes familiar with the architecture, the naming standards and the San Francisco documentation templates, and much more.

Developing a transition plan to San Francisco involves determining the overall strategy to ensure a smooth migration. The San Francisco Roadmap provides suggestions and guidelines for putting a project plan together.

After a prototype has been completed, the development team has obtained the necessary skills to tackle a complete application. Note that typically the prototype is used for learning and exploration but in most cases it is not used beyond that experience.

Chapter 9. San Francisco Open Tools Strategy

The San Francisco tools strategy is one which is founded on the slogan of "Optimized Openness." The thrust behind the optimization element is to focus our efforts on providing the best development technology specifically in support of the San Francisco project. The drive behind the Openness aspect is to make the statement that all tool providers are viable candidates for inclusion in the San Francisco project.

This chapter contains the following sections:

- Section 9.1, "Foundation" communicates the fundamental working assumptions, or principles, that collectively form the basis of the San Francisco Tools Strategy.
- Section 9.2, "Vision of San Francisco Tools End Game" on page 94 establishes long term plan directions for San Francisco Tools.
- Section 9.3, "San Francisco V1R1 Available Tools" on page 96 highlights some of the tools already available today to reduce the complexity of developing applications with San Francisco.

9.1 Foundation

The purpose of this section is to communicate the fundamental working assumptions, or principles, that collectively form the basis of the San Francisco Tools Strategy.

Those principles include:

1. **Tools are an important part of San Francisco**. Building complex applications requires industrial strength tools. Tools help simplify the process of building applications and allow you to use San Francisco more efficiently.
2. **The entire application life cycle should be supported.** The process of developing an application goes through requirements gathering, analysis, construction, testing and deployment. Our aim is to maximize the seamlessness, traceability and completeness of tools required for the entire life cycle.
3. **Multiple entry points into the life cycle should be supported.** While we intend to support the entire life cycle, it is acknowledged that you may want to use tools only in certain portions of the life cycle. This means that San Francisco tools need to provide multiple entry points into the life cycle by providing wizards or smart guides to assist you.
4. **An evolutionary approach to cross tool and tool provider integration will be taken**. As is the case with any major project, levels of sophistication and synergy between the various components of the San Francisco tools delivery will improve over time. This improvement will come in the future through two different channels. One is through the results of ongoing cross tool vendor integration efforts, and the other is through the addition of some number of more "full life cycle coverage" single tool offerings. Key elements of this integration activity include the architecture definition and instrumentation of a meta model, tool-to-tool control flow, debug and wizard services.

5. **Multiple deployment platforms must be enabled**. Support must be provided for the suite of target San Francisco fat/thin client and server combinations.
6. **Multiple tool suppliers will be leveraged**. The approach to providing tooling for San Francisco is one based on openness. Any tool provider that is interested in providing some element of support for the project should contact the IBM San Francisco Development Organization.
7. **Multiple tools per life cycle step will be offered**. The above statement on tool provider openness logically cascades to this principle. Direct feedback has been received from the application development community (San Francisco Independent Software Vendors) that they desire a set of tools to pick from for each of the major tools categories comprising the overall application life cycle.
8. **Multiple development audiences must be supported**. There are many different user audiences that need to be enabled for San Francisco. The user audience includes domain experts, business analysts, Java programmers, and application administrators.
9. **Multiple OO methodologies need to be enabled**. The San Francisco product is comprised of a distributed object-oriented infrastructure, a class library of reusable business objects, and a series of industry domain specific frameworks. Our intent is not to dictate the use of any specific OO methodology to understand and extend the San Francisco frameworks into application solutions.

9.2 Vision of San Francisco Tools End Game

One of the major purposes of this section is to establish long term plan directions for San Francisco tools. Figure 37 on page 95 shows the “San Francisco Tools Vision.”

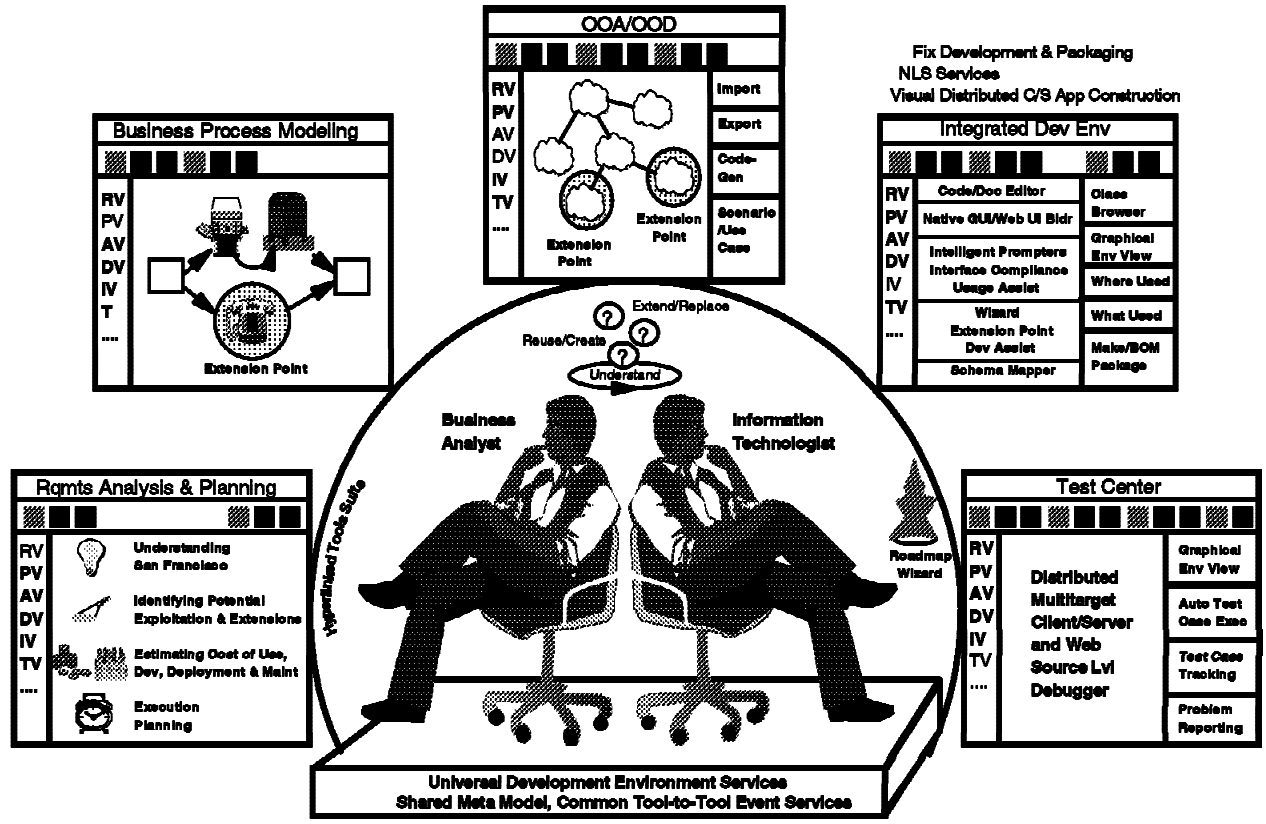


Figure 37. San Francisco Tools Vision

In order for the San Francisco product content to be usable to the masses of application developers, and to a smaller degree the emerging set of framework developers, a synergistic tools suite must be delivered. Simplification, leading to enhanced overall ease of use is paramount to a successful tools offering.

Based on direct Independent Software Vendor input, it is very important that the multiple ways of viewing a framework's content, semantic and syntax be supported. This includes the provision of a business process view, an object-oriented analysis and design view, and an implementation class hierarchy view. Also required is the view of how objects collaborate, which is many times orthogonal to the class hierarchy. Cutting across all of these views is the need for the identification of the framework extension points. For each extension point, it needs to be clear whether it is optional, meaning that some level of default behavior is provided as part of the framework product offering, or required, and what the required, or preferred approach is to completing the extension. Examples of completing an extension include subclassing or class replacement. The level of cross view synchronization desired is akin to what is provided today between a source level debugger and its supporting source code editor.

Due to the complexity of the frameworks, a special application solution requirements analysis and planning facility should be provided.

There are a number of different alternatives for how the specification and execution of business processes can be supported within San Francisco. In the San Francisco V1R1 time frame the business processes are implemented as an integral part of the industry domain frameworks. Extension points are used as a

way to specify where required, or specialize where optional, the business process flow of control and related data. The notion of a business role is not currently formalized in the San Francisco framework implementations. The potential use of an external workflow service, or better yet, the provision of a San Francisco embedded workflow service is under investigation for a future release.

Providing support for object-oriented analysis and design tools is a very important part of the San Francisco tooling effort. Examples in this arena include the likes of the Rational Rose category, class description and object interaction diagrams that are being delivered as part of San Francisco V1R1. It is anticipated that a number of different OOA/OOD tools will be enabled over time.

Partial code generation, when driven from either an OOA/OOD, IDE, or a stand alone tool is an excellent example of what is being pursued to simplify the development of San Francisco applications. Due to the criticality of code generation, the delivery of code generation services will begin in San Francisco V1R1 and grow, both in breadth and level of sophistication with each subsequent release.

There are a number of additional application development capabilities which are fundamental to the overall San Francisco tools package. They include the following:

- Java Integrated Development Environment
- Distributed Client/Server Application Development
- Schema Mapping Services (Relational and file system based -> OODBMS)
- National Language Support Tools
- Fix Development and Maintenance Services
- Testing Support (Unit, Component, System, Extended Framework, Enterprise, Performance, Usability).
- Deployment Support

9.3 San Francisco V1R1 Available Tools

The primary objective of the San Francisco tools is to reduce the complexity of developing applications using the framework. Another aspect that needs to be addressed is how to navigate through San Francisco. If you look at the sheer size of San Francisco, there are hundreds of business processes that involve thousands of different classes. It is literally impossible to look at San Francisco without a tool or a combination of tools and documentation that guides the business designer through the processes from a high-level perspective down to the individual task details, and that allows programmers to pick up the requirements and map them to the right extension points. Finally, once you become familiar with the San Francisco programming model, you realize that several programming tasks are rather repetitive and can be greatly automated. The tools that are being selected to help you with developing applications based on San Francisco address all of these aspects, from providing assistance in browsing through the numerous processes, object diagrams, classes, and extension points to generating class skeletons that pertain to the programming model.

The following sections in this chapter highlights the tools that have been already integrated with San Francisco V1R1. Additionally, the San Francisco team is looking at a set of more traditional tools to assist with the day-to-day activities of developing applications such as using an integrated development environment, team programming, versioning, and so on.

9.3.1 Rational Rose

Rose diagrams provide support for object-oriented analysis and design. They give a vivid picture of how the system will look during analysis and design phases. During these phases, you can depict the system in terms of relations, classes, categories, and so on. The appropriate diagrams can be drawn to show how the system is portrayed and the necessary enhancements and changes can be made. The latest version of Rational Rose supports several design notations such as Booch, UML, and OMT. Until now, the San Francisco development team has always used Booch notation, and the various diagrams in the documentation are provided in Booch. In the future, a conversion to UML might happen because this seems to be becoming the de facto standard notation.

Rose provides two main categories of diagrams: static and dynamic diagrams. Static diagrams are class diagrams, category diagrams, and use-case diagrams. Dynamic diagrams are object interaction diagrams and state diagrams. The San Francisco development process mainly uses the class diagrams and object interaction diagrams. In some cases, for complex classes, state diagrams are also used.

Rational Rose is a good choice for also doing your own application development because the code generator provided with San Francisco depends on Rose "mdl" and "cat" files. After the design phase of the development cycle, you end up with a set of class and object interaction diagrams. To prepare these for the code generator, you have to add some additional non-standard notations into the diagrams. By looking at an example, you can easily understand that certain aspects in the Rose notation do not map uniquely to a code implementation in Java using San Francisco components. If we have a 1-to-n relation between two classes, this might be implemented in several ways. San Francisco provides different types of collection classes depending on the type of elements in the collection and the size. This information needs to be passed into the code generator so that it can generate the appropriate code. This is done through a set of keywords (called #directives) that direct the code generator. These keywords need to be added into the documentation of the Rose diagrams.

To help you enter these #directives in the diagrams, San Francisco provides a wizard that can be installed into the Rational Rose tool menu bar. This wizard helps you enter the directives so that you do not have to memorize them. Once the model has been enhanced with these directives, it can be saved and used by the San Francisco Code Generator. In the future, San Francisco may support other object-oriented analysis and design tools.

9.3.2 San Francisco Code Generator

The San Francisco Code Generator produces Java code starting from your Rose design diagrams. Its main purpose is to eliminate a set of repetitive tasks and hide complexity that you must implement for every business object so that it complies with the San Francisco programming model. The Code Generator starts from your static class diagrams and generates the necessary code to implement the different business objects that you have defined together with

their attributes and relationships. At this moment, there are no provisions for generating business logic out of the object interaction diagrams or for generating client code using the various business objects.

As already discussed in the section 9.3.1, “Rational Rose” on page 97, you need to add the necessary #directives to your diagrams before loading them into the Code Generator. The Code Generator converts the diagram files to an internal database. From that database, you can select certain business classes and generate the necessary Java files for them.

Currently, there is no support for round-trip engineering. This means that changes you manually make to the Java files are not imported into the models, so if you regenerate a certain business object, you have to reimport any manual changes that were made. In future versions, this support might be provided so that managing code becomes easier.

The Code Generator also helps with the implementation of certain design patterns. Special #directives are defined to indicate that you want to implement a design pattern. The Code Generator then generates some or all of the classes needed for implementing this pattern.

9.3.3 IDE and Version Control in San Francisco

IDE Tools Strategy

Based on San Francisco's “Open Tools Strategy,” a number of different IDEs will be enabled over time. The rate at which a particular IDE is proven to work well with San Francisco is primarily up to the provider of the IDE. The extent to which an individual IDE is enhanced to provide San Francisco “Optimized Wizards or Intelligent Assists” is also heavily influenced by the company developing the IDE.

After the code generation step, you need to add additional code to the Java files. You also find out that it becomes difficult to keep track of different versions of your classes and the accompanying documentation. Therefore, you probably need a version control environment, which also gives you the capability to work with a team on the same set of source files. San Francisco does not recommend anything in particular to be used in this area, none of the IDEs have integrated support for team development and version control.

Besides the version control at development time, you also need to keep track of the versions of classes at the deployment of your application. One of the advantages of Java is that it is easy to replace one or more classes by a new version without having to recompile other parts of the application. This has great benefits. For example, you can start to profit from a new version of San Francisco without having to recompile any parts of your application. The drawback of this is that it becomes easy to create many different versions of classes and you can lose control of which level of code exists in a certain implementation of the application. It is necessary to develop a structured way of releasing applications and fixes to avoid this kind of problem.

Chapter 10. Developing Applications on Top of San Francisco

According to the rules defined by the San Francisco programming model, a *client role* (see 3.4, “The San Francisco Programming Model” on page 37) applies when the programmer can use business related classes and frameworks without changes or extensions. *Client*, therefore, refers to any code that uses existing object operations. Client code can be running in a batch-style process, in a user-interface program, or even in other business objects.

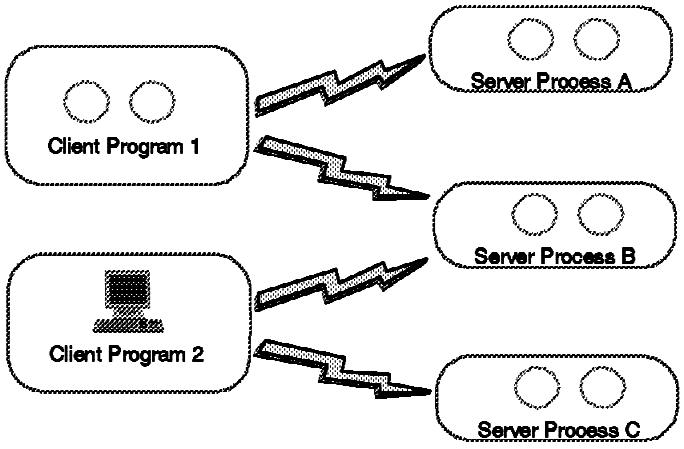


Figure 38. The Client Role Defined by the Programming Model

San Francisco provides base classes to develop a user interface (UI). These classes enable Independent Software Vendors (ISVs) to quickly build solutions. The UI classes are structured so that they provide easy extensibility and reuse as the business objects are extended. They extend the UI controls in Java's Abstract Window Toolkit (AWT) package to provide features such as an entry field with keystroke validation and frame with status message.

San Francisco can also be used with UI tools that support communication with Java Virtual Machines (JVMs). This currently includes Java based products. However, this is changing as non-Java based products such as Visual Basic include features that allow communication with JVMs.

This chapter contains the following sections:

- Section 10.1, “Applications and Applets” on page 100 explains the two models in which Java client programs may be executed.
- Section 10.2, “Client Programming with San Francisco” on page 102 explains the basic application architecture proposed and supported by San Francisco.
- Section 10.3, “The San Francisco User Interface Style Guide” on page 106 introduces the San Francisco User Interface Style Guide which defines an application-independent way of viewing and presenting business in a GUI.
- Section 10.4, “The San Francisco User Interface Framework” on page 108 introduces the San Francisco User Interface Framework which has been developed to make it easier to develop an UI to work with San Francisco business objects.

10.1 Applications and Applets

There are two modes in which a Java client program may be executed:

- Application
- Applet

An application is started the same as any other program from the client's disk, no matter if the disk is physically in the client machine or attached as a remote disk. As long as you have access to the executable file (and the program was written to be run stand-alone), it is called an application.

An applet is embedded in an HTML page and loaded with that page, usually over the network. You have no access to the executable file, since it is embedded in the page. If the applet shows a user interface, it is shown in the window of the HTML browser. For security reasons, applets loaded over the network have several restrictions:

- An applet cannot access files on the computer where it is executing
- An applet cannot make network connections except to the host from where it was loaded.

The following statement describes the way to access San Francisco business objects from a client computer.

- The client computer runs a Java applet that does not use San Francisco classes. This applet is connected to a Java server that, on the other hand, is a client to a San Francisco server. Using this configuration, no San Francisco classes need to be transferred over the network, thus reducing the amount of network traffic.

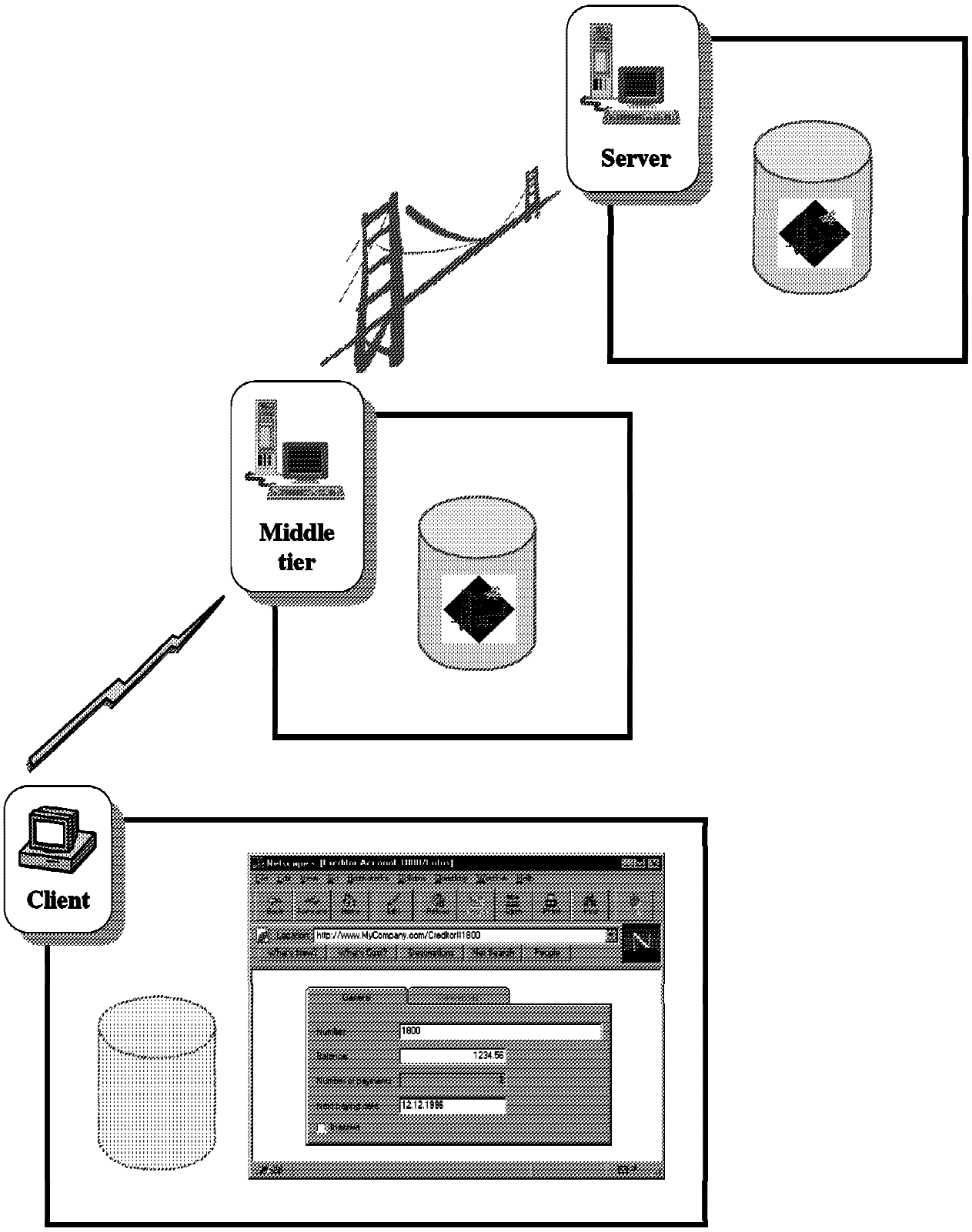


Figure 39. San Francisco Client using Middle Tier

10.2 Client Programming with San Francisco

The basic application architecture proposed and supported by San Francisco is shown in the following picture:

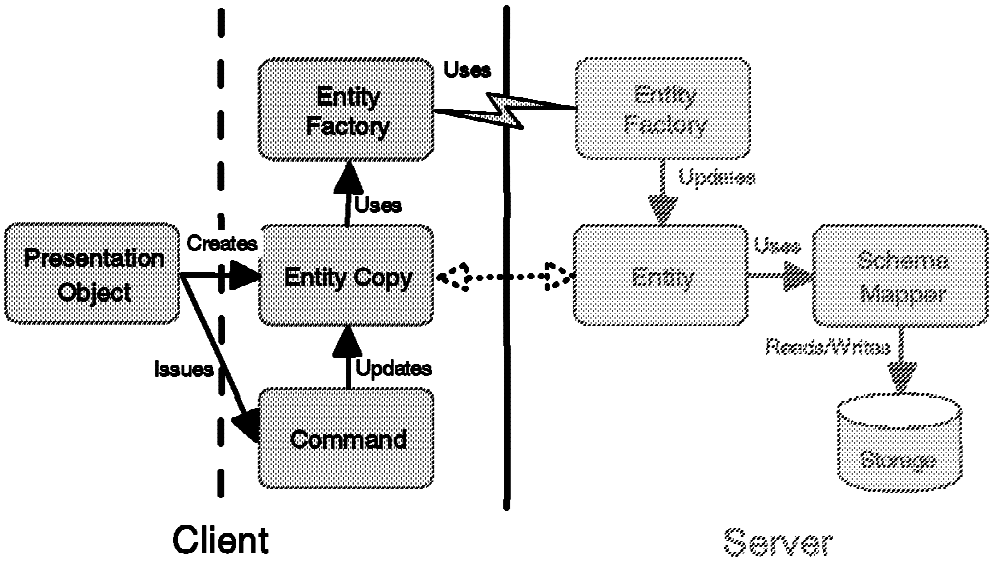


Figure 40. Application Architecture

The client issues its requests to a local copy of a business object. It uses a Command object to update the object. Object management actions are routed to the local Factory that, in turn, communicates with its corresponding server Factory.

10.2.1 Transaction Model

The Java implementation itself does not provide a transaction service. San Francisco's Java Object Transaction Service (JOTS) defines the architecture that allows multiple, distributed objects to cooperate to provide atomicity. The architecture enables the objects to either commit all changes together or to roll back all changes in the presence of failure, regardless of their actual location in the LSFN. This architecture is implemented in the Transaction Framework.

Every operation using persistent objects needs to be included in a transaction. The only exception to this rule is represented by NO_LOCK copies of objects. These transactions have support for commit and rollback behavior. The management of these transactions is done through the Base Factory. Changes to an object are made persistent by committing the transaction.

The scope of the transaction is determined by the client program. This can be done in one of the following two ways:

- The actions to be executed in the transaction are implemented in a Command. The Command can be executed as a transaction on the client side or the server side.
- The transactional actions are enclosed between the calls to begin and commit a transaction directly in the client code.

The Transaction Framework supports two kinds of transactions:

Two-phase This transaction type is used when all data stores used for San Francisco objects offer a two-phase commit interface such as DB2. Under two-phase commit protocol, the changes made in one unit of work are either all committed or all rolled back. It guarantees the data integrity and provides recovery in case of system/server failure.

Mixed-phase This transaction type is used when objects are stored on a data store that does not offer two-phase commit such as Posix files. It allows recoverable resources and non-recoverable resources to participate in the same transaction. In case the system fails, there is no recovery for the non-recoverable resources.

The choice of the appropriate transaction type is made by the administrator of a San Francisco installation. The application programmer does not deal with different types of transactions.

Note: Nested transactions are not supported.

10.2.2 Processes and Threads

San Francisco implements a distributed process (DP). It is based on the concept of a normal process in that it is an anchor point for one or more threads that are actively working under it. The threads of a distributed process can be spread across one or more Java Virtual Machines (JVMs). Like a normal process, the DP has a context, or shared information space, that each thread in the distributed process can access and modify. This context is called the distributed process context (DPC). Changes made to the DPC by one thread are immediately available to the other threads in the distributed process.

A work area is a thread-scoped data area that contains both transactional and non-transactional data. A thread can be working in only one work area at a time, but is allowed to switch work areas if required. The initial client thread has a default work area. You can transfer a work area to another client thread by suspending the work area in the first thread and resuming the work area in the second thread.

A thread must be associated with a work area to access the distributed process context. If the client creates a new Java thread, this thread can either create a new work area or resume work on a work area that was suspended in a different thread. San Francisco provides a subclass of the Java Thread class that, on creation, automatically creates a work area.

Note: User interface event handlers run in a separate Java thread. Before any calls to San Francisco are made, this thread has to be associated to a work area. After execution of the client code, the thread should release the work area again.

Server threads are associated with the work area of the client thread requesting the service.

There may be multiple independent transactions concurrently running in different threads, but there always is only one thread associated with the same transaction.

10.2.3 Creating and Deleting Entities

All Entities are created through a Factory class. Client programs are not allowed to use standard Java methods to create instances of objects; they need to pass through the provided Factories. These methods allow the creation of transient and persistent objects by specifying the appropriate parameters.

When creating persistent Entities, the client does not care about where new objects are stored. Defining the storage location is done by the application administrator at the installation site (see Section 7.1.3, “Container Configuration” on page 69).

To delete Entities from the persistent store, the `delete()` methods on the Base Factory have to be used. Transient objects do not need to be explicitly deleted because they are removed by the garbage collection.

10.2.4 User Aliases

Giving user aliases to persistent objects is an easy way to retrieve objects during the development of client programs. Generally, this is not a good way of using user aliases. In a production application, only a few top-level objects should get a user alias. Using containing business objects is the recommended way to retrieve Entities contained in them.

For manipulating user aliases, the Base Factory again provides the appropriate methods.

10.2.5 Accessing Entities

To update an existing Entity, the client program first needs to get a valid reference to it. This can be done through the Base Factory using the Entity's user alias, or the containing object, which may be a Collection or any other object having the target Entity as an attribute value.

To control concurrent access to objects, there are three locking strategies used with San Francisco:

No Locking No restriction on other processes to read or change the object.

Optimistic Locking
No restriction on other processes to read or change the object, but the commit fails if the object was changed by other processes while it was locked.

Pessimistic Locking
Other processes have restricted access to the object.

The latter two can be specified for read or write access.

An interesting strategy is optimistic read access (OPTIMISTIC_CLEAN). It means that the state of the object is checked at commit time even if it is only used for read access. Take, for example, a stock exchange transaction. You can read the current value of a company's stock and let a customer know. Depending on the value, the customer might tell you to go ahead and sell. You must make sure, though, that during your conversation with the customer, the share did not change, or otherwise your transaction must be rejected (the commit operation will fail).

The location where the accessed Entity should run is specified by one of two access locations:

Local

- The Entity is copied to the client process
- All method calls are performed on the copy
- The copy is written back home at commit time

Home

- All methods are executed in the Entity's server process
- Method calls are performed through a local proxy
- The Entity must be locked using a pessimistic strategy

The access mode (which includes lock mode and access location) is specified explicitly when requesting an object through the Base Factory using user aliases. When requesting an object as a component from a containing Entity through its `get...()` method, the obtained object inherits the access mode of its containing Entity. (There is no access mode parameter on a `get...()` method).

References to an object are valid only within a transaction. After commit, a new reference to the Entity must be obtained. The only exception is when using NO_LOCK copies. These are valid across transaction borders.

10.2.6 Collection Element Access

When accessing a Collection, two lock modes are specified. One applies to the Collection and the other to the elements of that Collection. The lock mode and location specified for the elements are automatically applied as elements are subsequently accessed using either iterator or non-iterator operations on the Collections. Nevertheless, the lock type of an element may be upgraded at any time.

10.2.7 Updating Entities

Changes to the Entity's attributes are made by using the `get...()` and `set...()` methods. These changes are written to the data store when committing the transaction. Changes made outside of a transaction or to a NO_LOCK copy of an object are not stored.

10.2.8 Notification Service

To be aware of changes to the currently accessed objects done by other processes, client programmers may want to make use of San Francisco's built-in notification services.

San Francisco defines two notification mechanisms:

Synchronous This mechanism is a direct one; at commit time of the changes on the source object, the observer, which can be an Entity or a Java class that implements Java observer, is notified of these changes. It can react to the changes properly as soon as the source object is committed.

Asynchronous In this mechanism the changes are stored in a notification mailbox; it is up to the observer to pick up outstanding notification messages. This mechanism is used when the observer wants to decide when to react to changes committed in the observed object.

Notification is generated when changes are committed on a persistent Entity.

Figure 41 illustrates the two different notification mechanisms that are available in the framework. Both mechanisms are based on the relationship between an *observable* and an *observer*. The observable is the source of the notification and needs to be a persistent Entity.

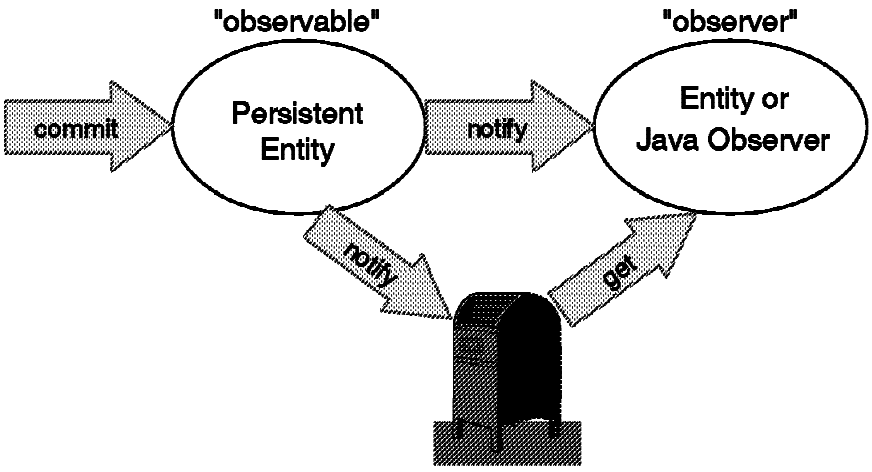


Figure 41. Notification Mechanisms

Using either of these two mechanisms, the action to be taken has to be implemented in the observer. There is no default action provided by the framework.

10.3 The San Francisco User Interface Style Guide

This guide defines an application-independent way of viewing and presenting business in a graphical user interface. It defines the concept of frames and controls.

Note: The User Interface Framework currently does not yet implement all definitions in the style guide. Open items are listed in the style guide.

Business objects and their attributes are displayed in standard user interface controls. These controls are arranged and used in standard frames. The interaction among business objects, controls, and frames is defined by the behavior concept.

10.3.1 The Standard Frame

The Standard frame describes the layout of controls common to all frames. It consists of the controls as shown in Figure 42 on page 107.

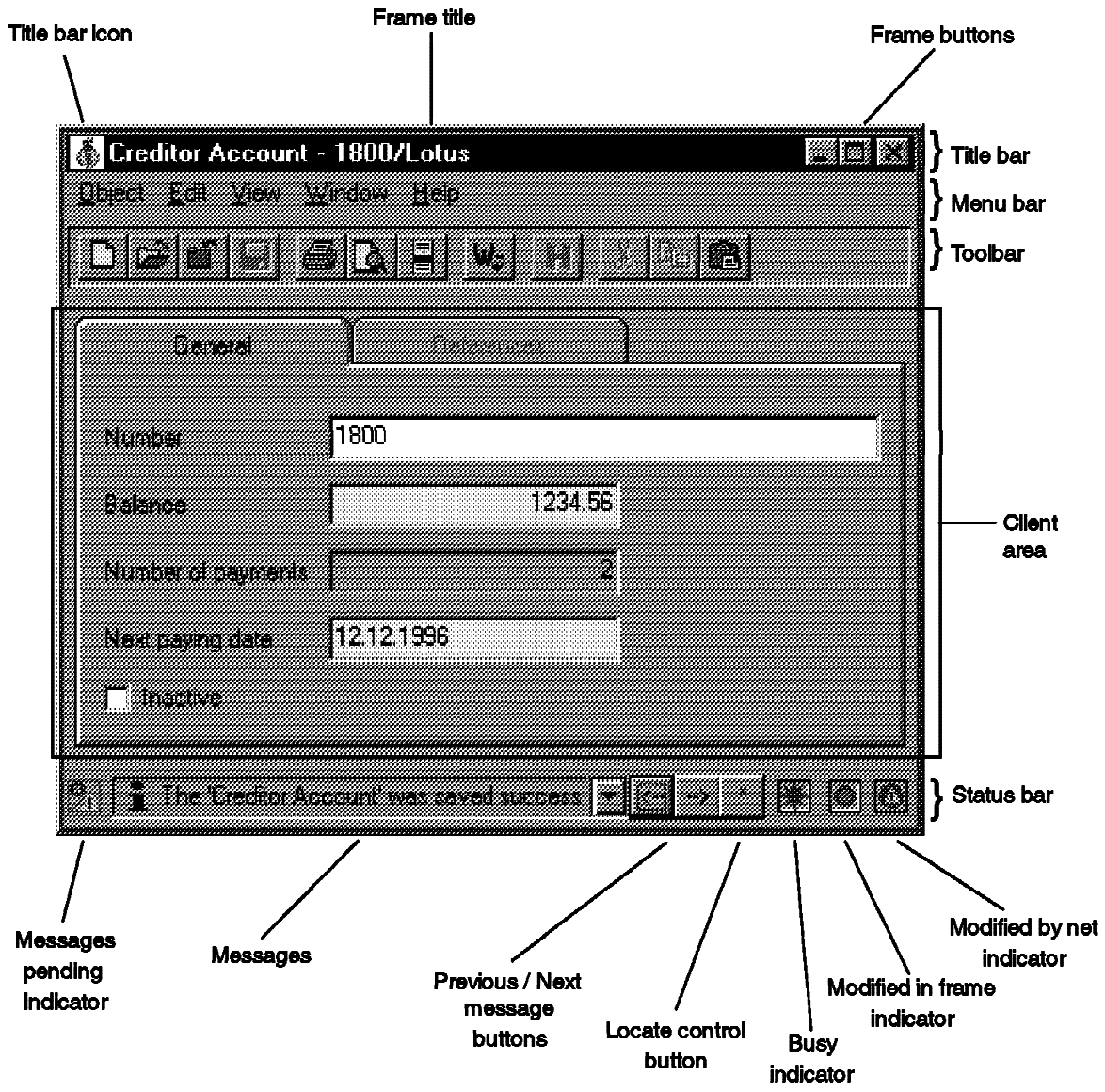


Figure 42. An Example of the Standard Frame and its Controls

Title bar icon Represents the type of information presented in the frame such as object type, frame type, and application.

Frame title States details about the information being presented and processed.

Frame buttons Are used to minimize, maximize, or close the window.

Menu bar Consists of menu items that represent actions available for the object, the frame, and the application.

Toolbar Consists of push buttons that represent actions available for the object, the frame, and the application.

Client area Holds the actual information.

Messages pending indicator
Consists of two parts indicating if there are messages or warnings pending.

Messages Holds the list of all messages.

Previous message button
Scrolls the message list to the message preceding the currently displayed message.

Next message button
Scrolls the message list to the message following the currently displayed message.

Locate control button
Sets the focus to the control that caused the currently displayed message.

Busy indicator Is shown during a user-initiated processing cycle that is going to last longer than one second (for example, accessing a database, opening a frame, or filling a collection). When the frame is busy, the indicator shows the busy icon; otherwise, the indicator is empty.

Modified in frame indicator
Is shown if the frame has unsaved changes. When there are unsaved changes, the indicator shows the modified icon; otherwise, the indicator is empty.

Modified by net indicator
Shows the state of the frame's contents. If the contents reflect the state on the database, the indicator is displayed empty. Different icons are displayed for the following states:

- Objects are currently loading
- Loading has been interrupted
- The frame is showing an old state because some objects have changed on the database.

All other frames described in the style guide are based on the definition of the Standard Frame. The most important frames are:

Main frame The starting point for an application.

Object frame The frame to maintain a business object.

Collection frame The frame to work with a collection of business objects.

10.4 The San Francisco User Interface Framework

The User Interface Framework has been developed to make it easier to develop a user interface to work with San Francisco business objects. The framework considers the standards and patterns used while developing San Francisco. When using the San Francisco user interface framework, you can easily build user interfaces that conform to the style guide.

San Francisco V1R1 and the UIF

This document was written based on San Francisco Version 1 Release 1 Modification 0 (V1R1M0). For that release, the User Interface Framework is still in beta status. Therefore, in future releases of San Francisco, there may be changes and additions.

The San Francisco User Interface Framework is built in Java Development Kit (JDK) 1.1. Most classes in Java's Abstract Windowing Toolkit (AWT) have been extended to add functionality. The added functionality makes it easier to use the components and to support the view-maintainer concept implemented in the framework. Some extra components such as a notebook and a collection container have also been added.

Like all other San Francisco components, the User Interface Framework currently lacks the beans support.

10.4.1 Basic Concepts

The architecture of the UI Framework consists of two main parts:

Views They are the interface to the user. The maintainer is the interface to the business object. The view is static. Once it is created, it is just a tool to show information. All of the controlling code is in the maintainer that is attached to the view.

Maintainers They perform the actions triggered by the user and the interaction with the business objects. It also controls the information flow between the view and business object.

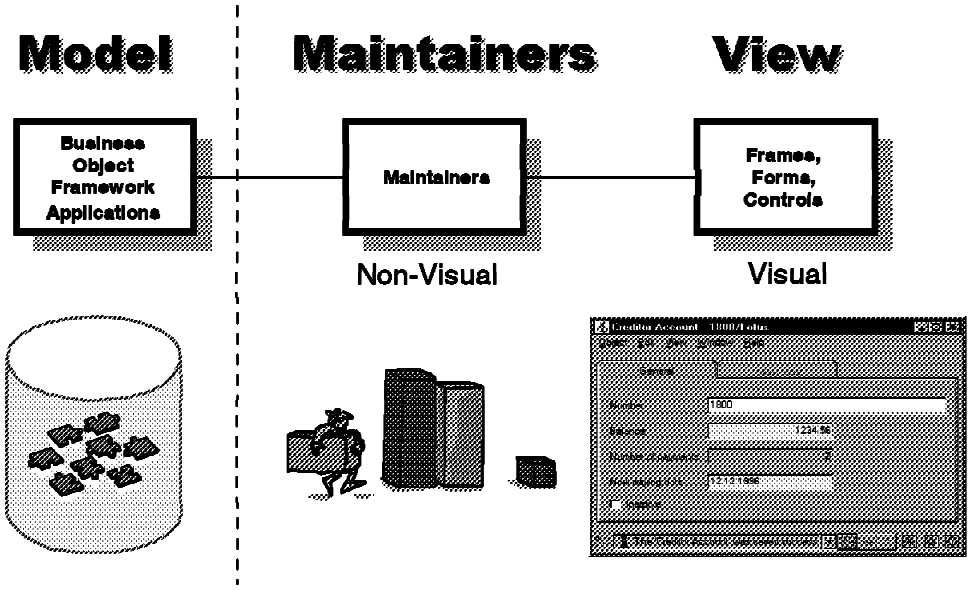


Figure 43. The User Interface Architecture

10.4.2 Views

The framework supports two major types of views:

- The collection view, which is used to display a collection of objects
- The object view, which is used to display detailed information for an object and its dependents.

To conform to the San Francisco User Interface Style Guide, the frame of each view is extended by a menu bar and a message area.

A view consists of:

- A frame
- Forms
- Controls

10.4.3 Frames

A frame in the San Francisco User Interface Framework is a window that conforms to the the style guide. This means it can display all frame components as defined in Section 10.3.1, “The Standard Frame” on page 106.

According to the style guide, different classes of frames are defined in the framework. These classes can be used as is without the need for subclassing.

10.4.4 Forms

A form is a reusable panel. Once a form is implemented, it can be used in either of the following ways:

- As the main form in the client area of a frame
- Embedded in other forms displaying a compound attribute of the outer form's object.

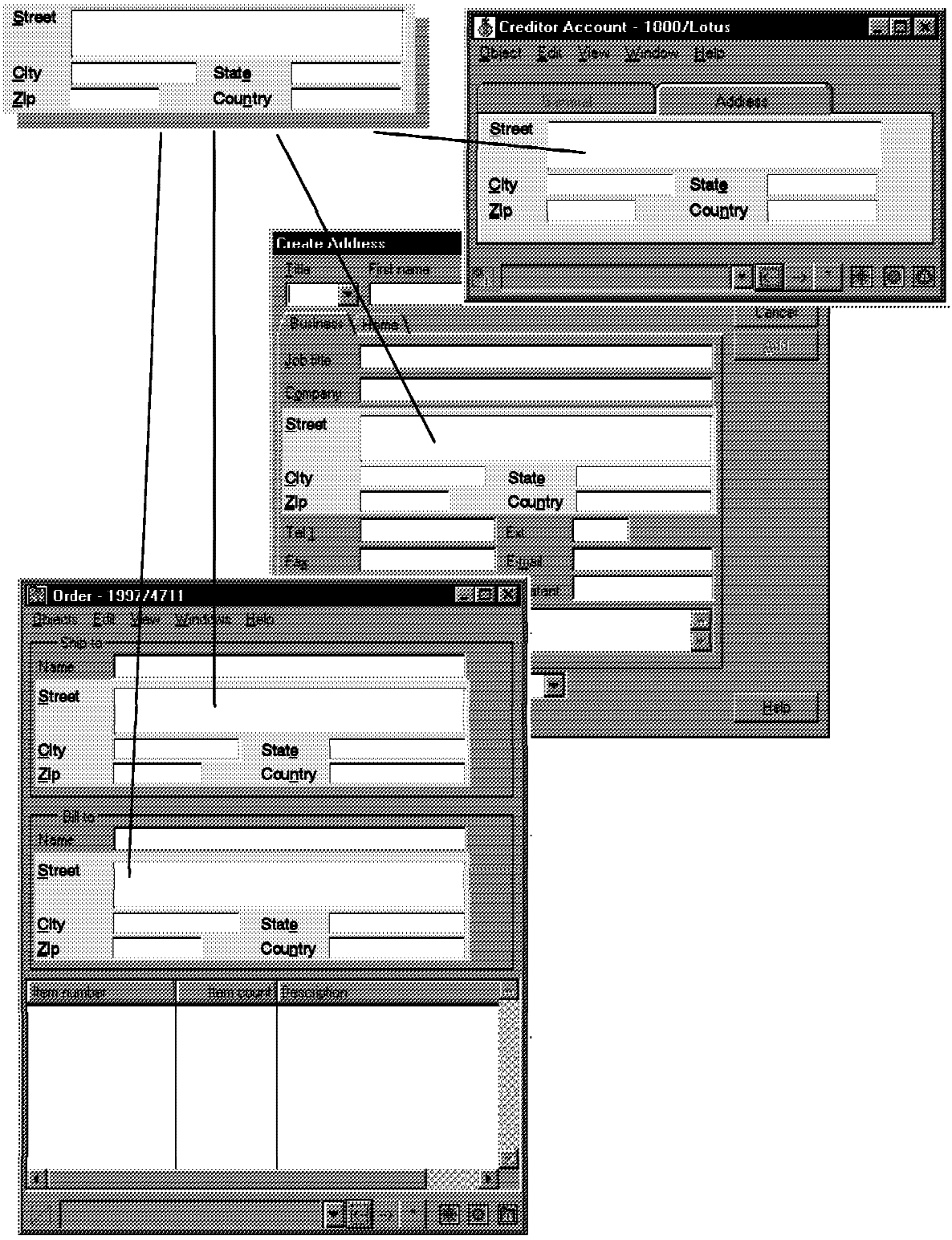


Figure 44. Reusing a Form

Major forms are:

Object Form

It creates and holds simple controls (such as text fields, or radio buttons) as well as other forms. Usually it is subclassed to set up the visual controls.

Collection Form

It creates and holds the collection control and may have pop-up menus and a button list as defined in the style guide. Usually it is not subclassed.

10.4.5 Maintainers

The maintainer is the active part in the UI. It holds the business object and reacts to actions triggered by the user. It is responsible for the manipulation of the view and the business object.

Major maintainers are:

Object Maintainer

It holds the business object displayed, maintains the view (the frame title and whether the object is enabled), and listens to action events. Object maintainers are always subclassed to support the individual attributes displayed in the view.

Collection Maintainer

It defines static user interface data for the view (such as selection type or column properties) fills the collection view, and listens to action events. Collection maintainers are always subclassed. The minimum to be overridden on a collection maintainer is how the collection is to be retrieved. Depending on the displayed information and the functions available to the user, other methods also have to be implemented in the subclass. The collection maintainer uses a collection item maintainer to handle each of the collection's element.

Collection Item Maintainer

It creates the (visible) collection item, fills the collection item with data, and, if the item is a node in a tree view, feeds its child items into the collection view. Collection item maintainers are usually subclassed only in case of a tree or multi-column collection view to implement retrieval of children or column data.

Attribute Maintainer

It transfers values between the view and the business object's attributes, routes warnings and messages to the message dispatcher, and converts data between view type and business type. Attribute maintainers are usually generic and need not be subclassed.

10.4.6 Client Area Controls

The User Interface Framework provides a set of controls to be used in the client area of a frame. These controls are designed to implement the style guide definitions and to cooperate with San Francisco business objects.

10.4.6.1 Data Validation Controls

Data validation controls can be linked to a simple attribute of a business object. They are grouped as follows:

- Text Controls
- Boolean Controls
- Enumeration Controls

Data Validation Controls are able to validate their content and to generate appropriate error messages. They implement the following styles:

- Mandatory
- Read-only
- Error

Data Validation Controls are associated with an attribute of the business object displayed in their containing form. They update the associated attribute according to user input. Text and enumeration controls are able to convert the attribute value to a formatted string and vice-versa. They validate user input and generate error messages. They support national languages.

10.4.6.2 Referenced Object

This is a powerful control to display an attribute that holds a reference to an Entity. Its behavior is derived from a drop-down list. The object referenced in the attribute is represented by its identifier and, optionally, by its description.

Besides the list drop-down button, the control holds an Edit button that allows the user to edit the currently selected object or to create a new object if no selection is currently active.

The client programmer, who uses this control, can specify:

- Which objects are displayed in the drop-down list
- Which attributes are used as identifier and as description
- How the attributes are formatted
- Which icon resource is to be used for the Edit button

10.4.6.3 Extended Controls

Extended controls consist of several children. They have a standardized layout that conforms to the style guide. They can be used without subclassing. The User Interface Framework currently supports:

Tabbed Control (Notebook)
: A panel that holds several pages. Each page is identified by a tab and only one page is visible at a time. The pages are object forms.

Collection Control
: A control to display collections of objects. Together with its container, it implements the different views and user actions defined in the style guide.

Status Area
: Implements the bottom part of a frame as defined in the style guide. In the current version of the framework, only the message area is displayed.

Button List
: Displays a list of predefined buttons, such as OK, CANCEL, and so on.

10.4.6.4 Message Handling

The User Interface Framework provides a message dispatcher that is the target for sending messages generated by a client program. The message dispatcher routes the messages to the corresponding frame, where they are displayed in the message list.

The client program that generates the message has to define which control a message belongs to. By pressing the Locate button, the user can set the input focus to the control that is specified in the message.

The framework takes care of removing messages when they are not needed anymore.

Appendix A. Special Notices

This publication is intended to help those people who are responsible for the task of recommending and implementing an object-oriented application development environment based on Java and Framework technologies. The information in this publication is not intended as the specification of any programming interfaces that are provided by the IBM San Francisco Business Process Components product. See the PUBLICATIONS section of the IBM Programming Announcement for for more information about what publications are considered to be product documentation.

References in this publication to IBM products, programs or services do not imply that IBM intends to make these available in all countries in which IBM operates. Any reference to an IBM product, program, or service is not intended to state or imply that only IBM's product, program, or service may be used. Any functionally equivalent program that does not infringe any of IBM's intellectual property rights may be used instead of the IBM product, program or service.

Information in this book was developed in conjunction with use of the equipment specified, and is limited in application to those specific hardware and software products and levels.

IBM may have patents or pending patent applications covering subject matter in this document. The furnishing of this document does not give you any license to these patents. You can send license inquiries, in writing, to the IBM Director of Licensing, IBM Corporation, 500 Columbus Avenue, Thornwood, NY 10594 USA.

Licensees of this program who wish to have information about it for the purpose of enabling: (i) the exchange of information between independently created programs and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact IBM Corporation, Dept. 600A, Mail Drop 1329, Somers, NY 10589 USA.

Such information may be available, subject to appropriate terms and conditions, including in some cases, payment of a fee.

The information contained in this document has not been submitted to any formal IBM test and is distributed AS IS. The information about non-IBM ("vendor") products in this manual has been supplied by the vendor and IBM assumes no responsibility for its accuracy or completeness. The use of this information or the implementation of any of these techniques is a customer responsibility and depends on the customer's ability to evaluate and integrate them into the customer's operational environment. While each item may have been reviewed by IBM for accuracy in a specific situation, there is no guarantee that the same or similar results will be obtained elsewhere. Customers attempting to adapt these techniques to their own environments do so at their own risk.

Any performance data contained in this document was determined in a controlled environment, and therefore, the results that may be obtained in other operating environments may vary significantly. Users of this document should verify the applicable data for their specific environment.

Reference to PTF numbers that have not been released through the normal distribution process does not imply general availability. The purpose of including these reference numbers is to alert IBM customers to specific information relative to the implementation of the PTF when it becomes available to each customer according to the normal IBM PTF distribution process.

The following terms are trademarks of the International Business Machines Corporation in the United States and/or other countries:

Advanced Function Printing	AFP
AIX®	AS/400®
DB2®	IBM®
OS/400®	RS/6000
System/36	System/390®
VisualAge®	400®

The following terms are trademarks of other companies:

C-bus is a trademark of Corollary, Inc.

Java and HotJava are trademarks of Sun Microsystems, Incorporated.

Microsoft, Windows, Windows NT, and the Windows 95 logo are trademarks or registered trademarks of Microsoft Corporation.

PC Direct is a trademark of Ziff Communications Company and is used by IBM Corporation under license.

Pentium, MMX, ProShare, LANDesk, and ActionMedia are trademarks or registered trademarks of Intel Corporation in the U.S. and other countries.

UNIX is a registered trademark in the United States and other countries licensed exclusively through X/Open Company Limited.

Other company, product, and service names may be trademarks or service marks of others.

Appendix B. Related Publications

The publications listed in this section are considered particularly suitable for a more detailed discussion of the topics covered in this redbook.

B.1 International Technical Support Organization Publications

For information on ordering these ITSO publications see “How to Get ITSO Redbooks” on page 119.

- *Accessing the AS/400 System with JAVA*, SG24-2152-00

B.2 Redbooks on CD-ROMs

Redbooks are also available on CD-ROMs. **Order a subscription** and receive updates 2-4 times a year at significant savings.

CD-ROM Title	Subscription Number	Collection Kit Number
System/390 Redbooks Collection	SBOF-7201	SK2T-2177
Networking and Systems Management Redbooks Collection	SBOF-7370	SK2T-6022
Transaction Processing and Data Management Redbook	SBOF-7240	SK2T-8038
AS/400 Redbooks Collection	SBOF-7270	SK2T-2849
RS/6000 Redbooks Collection (HTML, BkMgr)	SBOF-7230	SK2T-8040
RS/6000 Redbooks Collection (PostScript)	SBOF-7205	SK2T-8041
Application Development Redbooks Collection	SBOF-7290	SK2T-8037
Personal Systems Redbooks Collection	SBOF-7250	SK2T-8042

B.3 Other Publications

These publications are also relevant as further information sources:

1. *Object-Oriented Technology: A Manager's Guide*; David A. Taylor; Addison-Wesley Publishing Company, 1990 (ISBN 0-201-56358-4).
2. *Business Engineering with Object Technology*; David A. Taylor; John Wiley & Sons, Inc, 1995 (ISBN 0-471-04521-7).
3. *Design Patterns, Elements of Reusable Object-Oriented Software*; Erich Gamma, et al.; Addison-Wesley Publishing Company, 1994 (ISBN 0-201-63361-2).
4. *Object-Oriented Software Engineering: A Use-Case Driven Approach*; Ivar Jacobson; Addison-Wesley Publishing Company, 1992 (ISBN 0-201-40347-1).
5. *The Essential Distributed Objects*; Orfali, Harkey, and Edwards; John Wiley & Sons, Inc, 1996 (ISBN 0-471-12993-3).
6. *Rational Rose Essentials: Using the Booch Method*; Iseult White; The Benjamin/Cummings Publishing Company, 1994 (ISBN 0-8053-0616-1).
7. *Advanced Java -- Idioms, Pitfalls, Styles, and Programming Tips*; Chris Laffra; Prentice Hall, 1997 (ISBN 0-13-534348-8).
8. Object-Oriented System Development; Dennis de Champeaux, Douglas Lea, and Penelope Faure; Addison-Wesley Publishing Company, 1993 (ISBN 0-201-56355-X).

9. The following San Francisco documentation are available on-line at: `http://www.ibm.com/java/Sanfrancisco`
   - *San Francisco Roadmap*
   - *San Francisco Programmer's Guide*
   - *San Francisco Extension Guide*
   - *San Francisco User Interface Programmers's Guide*

How to Get ITSO Redbooks

This section explains how both customers and IBM employees can find out about ITSO redbooks, CD-ROMs, workshops, and residencies. A form for ordering books and CD-ROMs is also provided.

This information was current at the time of publication, but is continually subject to change. The latest information may be found at http://www.redbooks.ibm.com.

How IBM Employees Can Get ITSO Redbooks

Employees may request ITSO deliverables (redbooks, BookManager BOOKs, and CD-ROMs) and information about redbooks, workshops, and residencies in the following ways:

- **PUBORDER** —to order hardcopies in United States
- **GOPHER link to the Internet** - type GOPHER.WTSCPOK.ITSO.IBM.COM
- **Tools disks**

  To get LIST3820s of redbooks, type one of the following commands:

  ```
  TOOLS SENDTO EHONE4 TOOLS2 REDPRINT GET SG242157 PACKAGE
  TOOLS SENDTO CANVM2 TOOLS REDPRINT GET SG242157 PACKAGE (Canadian users only)
  ```

  To get BookManager BOOKs of redbooks, type the following command:

  ```
  TOOLCAT REDBOOKS
  ```

  To get lists of redbooks, type one of the following commands:

  ```
  TOOLS SENDTO USDIST MKTTOOLS MKTTOOLS GET ITSOCAT TXT
  TOOLS SENDTO USDIST MKTTOOLS MKTTOOLS GET LISTSERV PACKAGE
  ```

  To register for information on workshops, residencies, and redbooks, type the following command:

  ```
  TOOLS SENDTO WTSCPOK TOOLS ZDISK GET ITSOREGI 1998
  ```

  For a list of product area specialists in the ITSO: type the following command:

  ```
  TOOLS SENDTO WTSCPOK TOOLS ZDISK GET ORGCARD PACKAGE
  ```

- **Redbooks Web Site on the World Wide Web**

  http://w3.itso.ibm.com/redbooks

- **IBM Direct Publications Catalog on the World Wide Web**

  http://www.elink.ibmlink.ibm.com/pbl/pbl

  IBM employees may obtain LIST3820s of redbooks from this page.

- **REDBOOKS category on INEWS**
- **Online** —send orders to: USIB6FPL at IBMMAIL or DKIBMBSH at IBMMAIL
- **Internet Listserver**

  With an Internet e-mail address, anyone can subscribe to an IBM Announcement Listserver. To initiate the service, send an e-mail note to announce@webster.ibmlink.ibm.com with the keyword subscribe in the body of the note (leave the subject line blank). A category form and detailed instructions will be sent to you.

Redpieces

For information so current it is still in the process of being written, look at "Redpieces" on the Redbooks Web Site (http://www.redbooks.ibm.com/redpieces.htm). Redpieces are redbooks in progress; not all redbooks become redpieces, and sometimes just a few chapters will be published this way. The intent is to get the information out much quicker than the formal publishing process allows.

How Customers Can Get ITSO Redbooks

Customers may request ITSO deliverables (redbooks, BookManager BOOKs, and CD-ROMs) and information about redbooks, workshops, and residencies in the following ways:

- **Online Orders** —send orders to:

	IBMMAIL	**Internet**
In United States:	usib6fpl at ibmmail	usib6fpl@ibmmail.com
In Canada:	caibmbkz at ibmmail	lmannix@vnet.ibm.com
Outside North America:	dkibmbsh at ibmmail	bookshop@dk.ibm.com

- **Telephone orders**

United States (toll free)	1-800-879-2755
Canada (toll free)	1-800-IBM-4YOU
Outside North America	(long distance charges apply)
(+45) 4810-1320 - Danish	(+45) 4810-1020 - German
(+45) 4810-1420 - Dutch	(+45) 4810-1620 - Italian
(+45) 4810-1540 - English	(+45) 4810-1270 - Norwegian
(+45) 4810-1670 - Finnish	(+45) 4810-1120 - Spanish
(+45) 4810-1220 - French	(+45) 4810-1170 - Swedish

- **Mail Orders** —send orders to:

IBM Publications Publications Customer Support P.O. Box 29570 Raleigh, NC 27626-0570 USA	IBM Publications 144-4th Avenue, S.W. Calgary, Alberta T2P 3N5 Canada	IBM Direct Services Sortemosevej 21 DK-3450 Allerød Denmark

- **Fax** —send orders to:

United States (toll free)	1-800-445-9269
Canada	1-403-267-4455
Outside North America	(+45) 48 14 2207 (long distance charge)

- **1-800-IBM-4FAX (United States)** or **(+1)001-408-256-5422 (Outside USA)** —ask for:

  Index # 4421 Abstracts of new redbooks
  Index # 4422 IBM redbooks
  Index # 4420 Redbooks for last six months

- **Direct Services** - send note to softwareshop@vnet.ibm.com
- **On the World Wide Web**

Redbooks Web Site	http://www.redbooks.ibm.com
IBM Direct Publications Catalog	http://www.elink.ibmlink.ibm.com/pbl/pbl

- **Internet Listserver**

  With an Internet e-mail address, anyone can subscribe to an IBM Announcement Listserver. To initiate the service, send an e-mail note to announce@webster.ibmlink.ibm.com with the keyword subscribe in the body of the note (leave the subject line blank).

Redpieces

For information so current it is still in the process of being written, look at "Redpieces" on the Redbooks Web Site (http://www.redbooks.ibm.com/redpieces.htm). Redpieces are redbooks in progress; not all redbooks become redpieces, and sometimes just a few chapters will be published this way. The intent is to get the information out much quicker than the formal publishing process allows.

IBM Redbook Order Form

Please send me the following:

Title	Order Number	Quantity

First name Last name

Company

Address

City Postal code Country

Telephone number Telefax number VAT number

□ Invoice to customer number

□ Credit card number

Credit card expiration date Card issued to Signature

We accept American Express, Diners, Eurocard, Master Card, and Visa. Payment by credit card not available in all countries. Signature mandatory for credit card payment.

Index

Special Characters

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

V

W

ITSO Redbook Evaluation

San Francisco Concepts & Facilities
SG24-2157-00

Your feedback is very important to help us maintain the quality of ITSO redbooks. **Please complete this questionnaire and return it using one of the following methods:**

- Use the online evaluation form found at http://www.redbooks.com
- Fax this form to: USA International Access Code + 1 914 432 8264
- Send your comments in an Internet note to redbook@vnet.ibm.com

Please rate your overall satisfaction with this book using the scale:
(1 = very good, 2 = good, 3 = average, 4 = poor, 5 = very poor)

Overall Satisfaction _____

Please answer the following questions:

Was this redbook published in time for your needs? Yes____ No____

If no, please explain:

_____

_____

_____

_____

What other redbooks would you like to see published?

_____

_____

_____

Comments/Suggestions: **(THANK YOU FOR YOUR FEEDBACK!)**

_____

_____

_____

_____

_____

Printed in U.S.A.