Deferring Design Decisions in an Application Framework

JAMES E. CAREY
BRENT A. CARLSON

IBM Corporation

For an application framework to be useful, many design decisions must be deferred to the application developer. The degree to which such decisions should be deferred must be addressed systematically throughout a framework in order for that framework to become successful. The IBM SanFrancisco™ application framework addresses this issue by implementing specific design patterns which exhibit varying degrees of design deferral.

Additional Keywords and Phrases: D.1.5 [Software]: Object-oriented Programming, D.2.2 [Software Engineering]: Design Tools and Techniques, Object-oriented design methods, D.3.3 [Programming Languages]: Language Constructs and Features, Frameworks, Patterns, Design, Application Frameworks

1.0 INTRODUCTION

To be useful to an application developer, a framework must be designed with an overall application architecture that provides structure to the framework and guides the application developer through the process of building an application that incorporates the framework. Once this framework architecture is in place, the framework designer must then explicitly incorporate flexibility via deferred design decisions into key points throughout the framework architecture in order to make the framework broadly applicable. The following sections discuss specific design pattern examples taken from the IBM SanFrancisco™[1] application frameworks which address the issue of design deferral. These examples range from a fully deferred design decision to a directed design which guides the application developer through performance/flexibility tradeoffs.

1.1 THE IBM SANFRANCISCO PROJECT

The IBM SanFrancisco project[i] is a set of object-oriented application frameworks developed in Java™[2] and built on a multi-platform client/server infrastructure. The application frameworks provided by SanFrancisco can easily be extended by application developers. The primary objective of SanFrancisco is to increase the productivity of application developers, both by easing their entry into the development of distributed object oriented applications, and by providing a significant portion of the core function required by an application.

The function provided by SanFrancisco is divided into three layers. The Base layer provides a platform independent client/server infrastructure. The Common Business Objects layer resides on top of the Base layer and provides a set of commonly used business objects (e.g., BusinessPartner, Company, Address) and a set of generic and reusable object-oriented

---

[1] IBM SanFrancisco is a trademark of the International Business Machines Corporation.
[2] Java is a trademark of Sun Microsystems, Inc.

mechanisms. These business objects and mechanisms are in turn used by various Core Business Processes, each of which provides an application framework oriented towards a particular business domain (e.g., general ledger, warehouse management, order management). The patterns described in the remainder of this paper are oriented towards and used by both the Common Business Objects and Core Business Process layers of SanFrancisco.

2.0 DEFERRED DESIGN DECISION PATTERNS

2.1 FULLY DEFERRED DESIGN DECISIONS (A.K.A. THE "PASS THE BUCK" PATTERN)

Sometimes it's necessary for the framework designer to "pass the buck" for a particular design decision. This involves fully deferring the decision to the application developer. Such fully deferred design decisions typically arise in areas where the application usage of the framework is highly unconstrained.

An example of a fully deferred design decision within the IBM SanFrancisco framework is the use of deletion policies. (The policy design pattern as used in the IBM SanFrancisco framework is equivalent to the strategy design pattern described in **Design Patterns**.[ii] We have found that the term policy is more descriptive than strategy to the domain experts we have been working with during framework design, and the term policy will be used in place of strategy throughout the remainder of this chapter.)

Because an application developer will typically design significant numbers of new classes which interact with the classes provided by the framework, it is impossible for a framework designer to determine what algorithm should be used when attempting to delete a persistent framework object. For example, an application class may have one or more references to a framework class which must remain valid throughout the life cycle of a framework class instance. The IBM SanFrancisco framework deals with this problem through a mechanism that easily allows the application developer to define and implement a separate deletion algorithm for each domain class provided by the framework if needed.

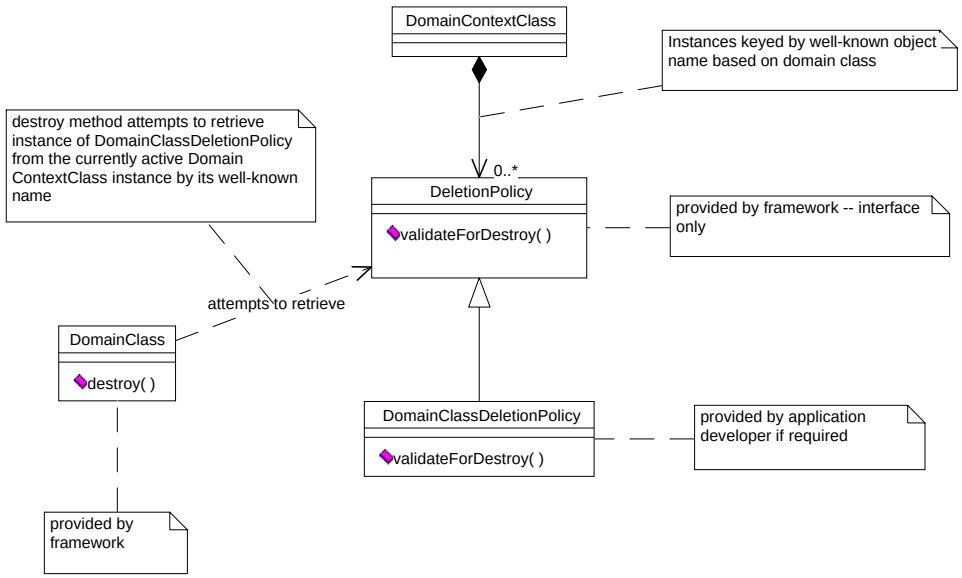

Figure 1 Deletion Policy Class Diagram

The mechanism works as follows: the framework defines a deletion policy base class (*DeletionPolicy* in the class diagram) with no concrete implementation classes. The application developer decides on a class by class basis if a deletion policy is required. Where deletion

policies are required (e.g., *DomainClassDeletionPolicy* in the class diagram), the application developer implements a subclass of the deletion policy base class and configures the application to use an instance of that class. This configuration typically occurs when the application is installed and consists of creating a deletion policy object instance for each class and associating that instance with an instance of the domain context class for the framework (*DomainContextClass* in the class diagram) using a well-known object naming convention.

Domain context objects may vary based on the application being supported by the framework. For most business-oriented applications, the domain context class is the *Company* class which represents the business entity using the application.

During object deletion, the deletion method of each framework class attempts to retrieve an instance of the deletion policy base class from the appropriate domain context object using the correct well-known object name. If an object associated with the object name does not exist, the framework object proceeds with the remainder of the deletion code. If a deletion policy object does exist, the framework object calls a validation method on that object which then has the opportunity to prevent the deletion of the object (e.g., because one or more persistent references to the object exist). Thus, the application designer can easily introduce deletion validation logic into the framework on a class by class basis, but is not required to make any changes to the framework where such logic is not required.

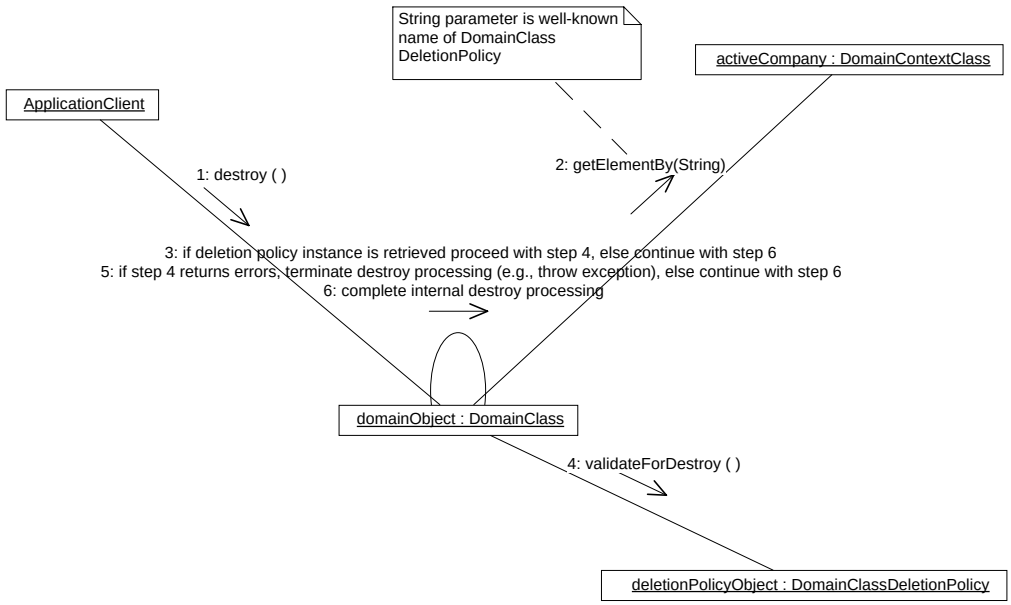

Figure 2 Object Deletion Object Interaction Diagram

2.2 COMPLETELY REPLACEABLE DESIGN DECISIONS

There are other situations that occur during framework design where the correct approach to use is a simple completely replaceable design approach. The standard strategy design pattern [ii] as shown below is often used to provide such a replaceable design within an application framework.

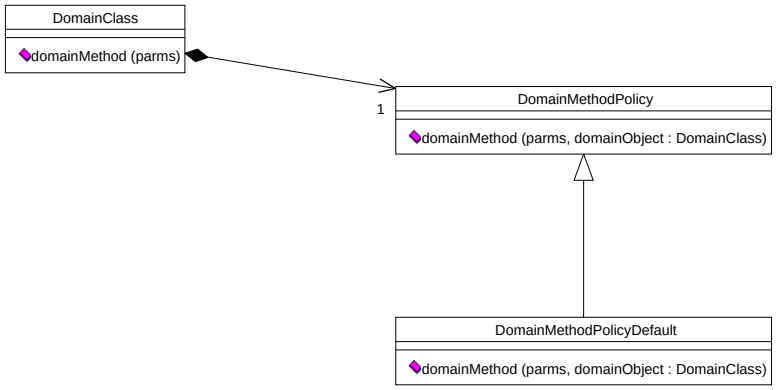

Figure 3 Strategy Design Pattern Class Diagram

An example of a replaceable design within the IBM SanFrancisco framework is the retrieval policy used for processing currency exchange rates. The purpose of this policy is to hold an algorithm for retrieval of a requested exchange rate across different combinations of possible exchange rate types (e.g., VAT, intercompany transfer). The IBM SanFrancisco framework provides both a policy base class and a default policy subclass which provides a standard implementation of exchange rate retrieval.

Many applications will find the default policy implementation provided by the framework acceptable and thus will not have to define an additional algorithm and associated policy subclass. However, application developers are free to define one or more additional policy subclasses which can be used to configure their application to support different exchange rate retrieval behavior. Depending upon the sophistication of the application, the choice of exchange rate retrieval policy may be made as part of the application installation process, or it may be exposed to the end user of the application, who can then configure their application as needed. In the extreme case, an application may choose to allow the end user to select different algorithms for each company configured within the application. For example, a multinational company may be legally required to use specific exchange rate conversion techniques which vary from one country to another.

2.3 DIRECTED DESIGN DECISIONS

We have found throughout the design of the IBM SanFrancisco framework that the standard policy pattern is often not sufficient to meet the flexibility needs of an application. In many cases, an application algorithm needs to vary not only based on the object doing the processing (a requirement which the standard policy pattern supports) but also based on the objects being processed.

As part of the IBM SanFrancisco framework, we have developed a design pattern which supports directed design decisions, providing the ability to:

- Configure a default behavior for a domain algorithm
- Override that default behavior on an object-by-object basis
- Make performance vs. flexibility tradeoff decisions on an object-by-object basis

This design pattern combines the standard policy (strategy) and chain of responsibility patterns described in **Design Patterns** [ii] into a class hierarchy which guides the application developer in making design decisions. The class hierarchy is composed of two separate policy class hierarchies which have been merged into a single composite class hierarchy, as shown below.

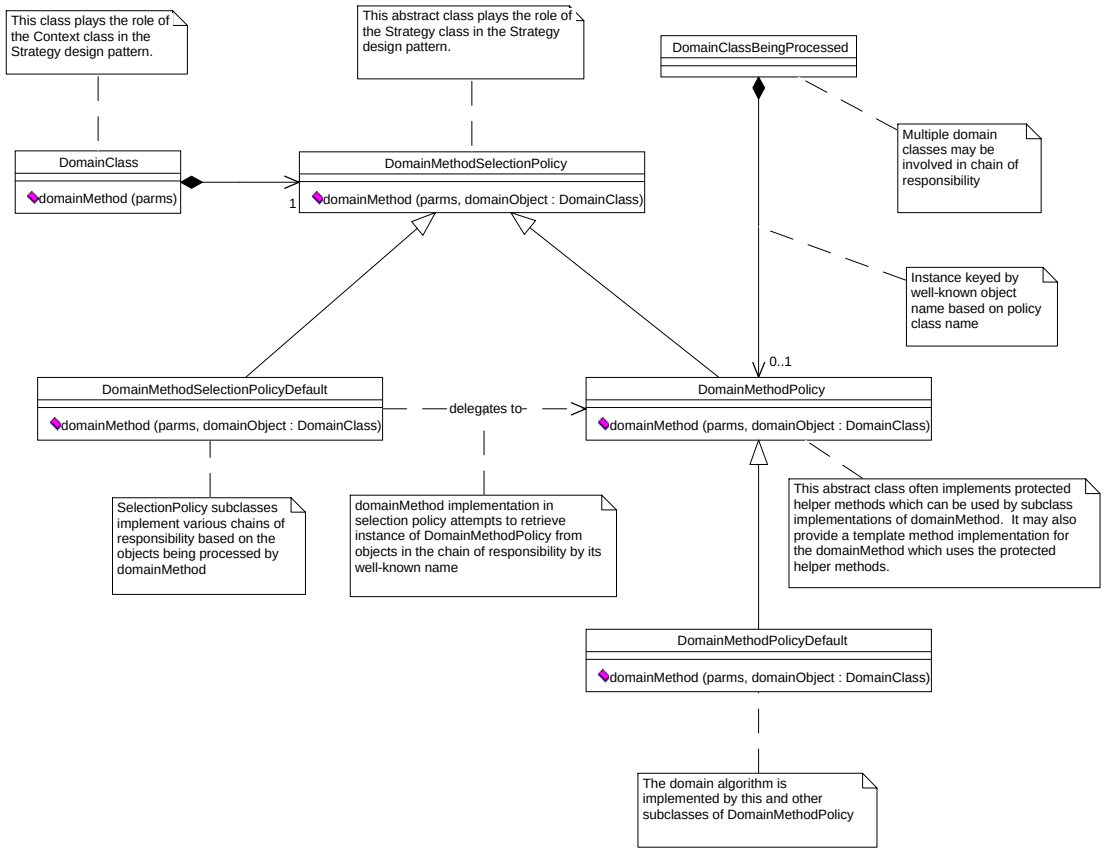

Figure 4 Chain Of Responsibility Driven Policy Pattern Class Diagram

The first class hierarchy is a standard policy class hierarchy, with a policy base class and one or more policy subclasses which provide different implementation algorithms. The classes *DomainMethodPolicy* and *DomainMethodPolicyDefault* represent this class hierarchy in the previous class diagram. (As described in the **Completely replaceable design decisions** section above, the framework provides one example policy subclass called a default policy which the application developer may or may not choose to use.)

The second class hierarchy is again a standard policy class hierarchy, but the purpose of this class hierarchy is to implement a selection policy whose implementation encapsulates a chain of responsibility. The classes *DomainMethodSelectionPolicy* and *DomainMethodSelectionPolicyDefault* represent this class hierarchy in the previous class diagram. The base class of the policy class hierarchy inherits from the base class of the selection policy class hierarchy, and both class hierarchies support a single public domain algorithm method signature which conforms to the standard policy pattern. Subclasses of the selection policy hierarchy implement various selection policies, each of which follows a chain of responsibility that searches through the objects being processed by the domain algorithm to determine which version of the domain algorithm (i.e., which subclass instance of the policy class hierarchy) is to be used during processing.

The policy and selection policy class hierarchies can be merged because each declares a common domain method. It may seem odd that the policy class hierarchy inherits from the selection policy class hierarchy, but policy classes are in fact also selection policy classes which implicitly select themselves rather than follow a chain of responsibility (or in other words, they implement a single object chain of responsibility).

Example 1: domainObject configured with selection policy instance

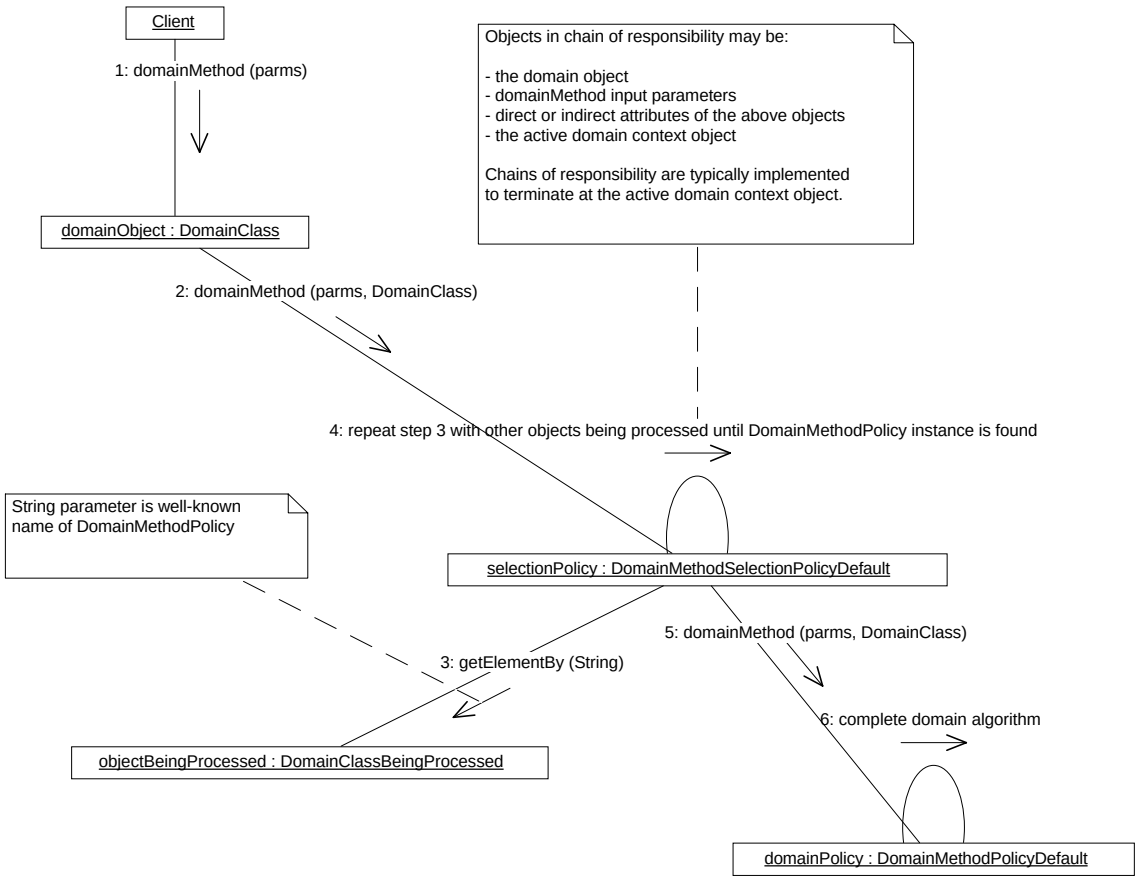

Figure 5 Selection Policy Configuration Object Interaction Diagram

As portrayed in the above object interaction diagram, the selection policy search is completed by using a well-known object naming convention to attempt to retrieve an object instance typed as the base class of the policy class hierarchy (e.g., *DomainMethodPolicy* in the example) from each of the objects included in the chain of responsibility. This search continues until either a policy object instance is found somewhere in the chain of responsibility or the default policy object instance (typically held by the currently active company object in the application) is reached. Thus, the policy used may be the default version configured for the application, or it may be an overriding version which is held by one of the objects being processed by the algorithm.

The application developer can choose to configure the domain object responsible for the algorithm being supported with a subclass instance of either the policy or of the selection policy class hierarchy. If the application developer chooses a subclass instance from the policy class hierarchy, the resulting behavior is that of the standard policy pattern, completely bypassing the chains of responsibility supported by subclasses of the selection policy class hierarchy. See the object interaction diagram below for a graphical portrayal of this behavior. The application developer would make this choice if performance is of greater importance than flexibility. If the application developer chooses a subclass instance from the selection policy class hierarchy, the chain of responsibility implemented by that instance will select the correct domain algorithm, thus increasing the flexibility of the application at the cost of some performance. If desired, the application developer can defer this decision to the end user of the application by providing a

creation/maintenance GUI that allows the end user to select from a set of policies, some of which may be selection policies. The application developer thus is given a directed approach to this portion of the application design.

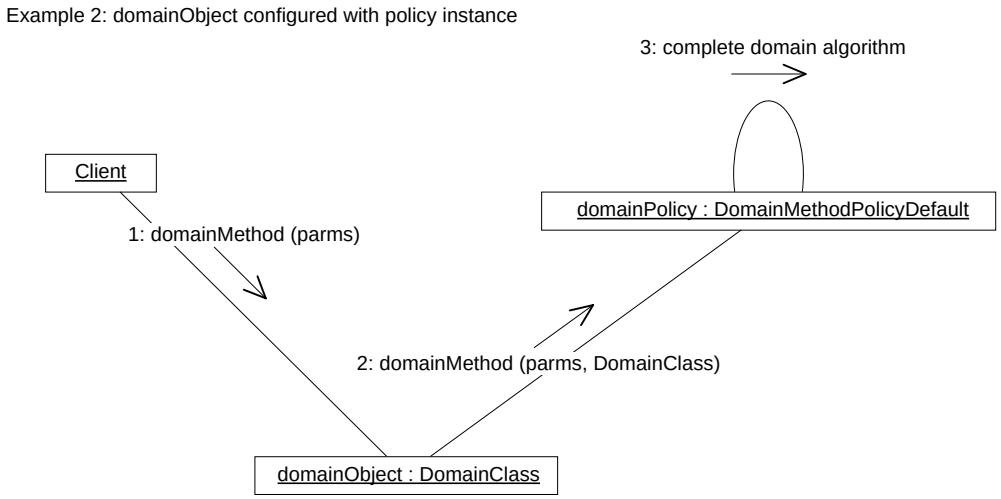

Figure 6 Policy Configuration Object Interaction Diagram

3.0 FINAL NOTES

We have found that one of the most difficult transitions a software developer has to make when switching from developing applications/operating systems/middleware to developing application frameworks[iii] is to resist the temptation to design highly focused, "bullet-proof" code. The typical application developer wants to build a "tank" which is capable of handling all sorts of unexpected conditions within the limited scope of the application. The framework developer instead needs to build an "erector set" which is capable of supporting many different applications, selectively exposing portions of the framework design to the application developer. This "erector set" can be used to build both good and poor applications. In order for a framework to be successful, the framework developer needs to guide the application developer through the necessary design decisions for these exposed portions in a consistent manner. Thus, a key part of the transition from application developer to framework developer is to begin thinking in terms of design patterns and abstractions, always stepping outside the specific problem to be solved and looking for related problems. If this is done consistently throughout the entire framework design, the framework has a good chance of being successful.

James E. Carey
IBM Corporation, 3605 Hwy. 52 N, Rochester, MN 55901
Tel: +1 (507) 253-0193
Fax: +1 (507) 253-3495
James Carey is a Senior Software Engineer at IBM Software Group in Rochester, Minnesota. He holds a M.S. in Computer Engineering from Syracuse University and has been working in object-oriented technology for over 8 years, beginning as part of the large team of developers who converted OS/400 licensed internal code to 64-bit processor technology using C++, one of the largest successful OO development projects in the software industry to date. He worked on IBM's SanFrancisco project from 1995 through 1999, initially as the lead designer for SanFrancisco's Common Business Objects (CBO) and General Ledger Core Business Process (CBP) and also as cross domain (CBO and all CBPs) architect for the product. From that experience, he coauthored an introductory book on SanFrancisco and a book on SanFrancisco's design patterns. He is currently component architect for IBM Software Group's WebSphere Business Components.

Brent A. Carlson
IBM Corporation, 3605 Hwy. 52 N, Rochester, MN 55901
Tel: +1 (507) 253-3032
Fax: +1 (507) 253-3495
Brent Carlson is a Senior Software Engineer at IBM Software Group in Rochester, Minnesota. He holds a B.S. in Computer Engineering from Iowa State University and has been working in object-oriented technology for over 8 years, beginning as part of the large team of developers who converted OS/400 licensed internal code to 64-bit processor technology using C++, one of the largest successful OO development projects in the software industry to date. He worked on IBM's SanFrancisco project from 1995 through 1999, initially as the lead designer for SanFrancisco's Warehouse Management and Order Management Core Business Processes and also as lead architect for the product. From that experience, he coauthored a book on SanFrancisco's design patterns. He is currently the lead architect for IBM Software Group's WebSphere Business Components, which is developing EJB-based business software components.

---

[i] JOHNSON V. and RUBIN B, *The SanFrancisco Project: Business Process Components and Infrastructure.* ACM Computing Surveys, 1998 (will appear)

[ii] GAMMA E., HELM R., JOHNSON R., and VLISSIDES J., *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley. 1994.

[iii] FAYAD M. and SCHMIDT D., *Object-Oriented Application Frameworks*. CACM, Vol. 40, No. 10, October 1997.