REPOSITORY-BASED DATA VALIDATION: A BUSINESS RULES APPROACH

Dr. Paul Dorsey, Dulcian, Inc.

Introduction

Ensuring that data is valid is one of the core requirements of most information systems. Many data validation requirements are easily supported by the database or application through simple properties. Properties such as data type or length are so simple that it is tempting to say that these rules should not even be independently handled, but simply a part of the data structure. However, this is often not the case. For example, in the situation where customer information is periodically accepted from an external data source, the raw data should be entered into a table for manipulation and validation, even if it violates rules. The mechanism handled by SQL Loader which takes the rejected records and writes them to a separate file is neither convenient nor user-friendly. It is much easier to allow the data file to enter into the system with no validation rules whatsoever and then apply the validation rules to the records now comfortably stored in the database. At that point, it is possible to identify which business rules have been violated by the entire logical record.

Many business rules, especially complex rules involving multiple records, cannot be practically enforced by the database. Even if it is possible to enforce these rules in the database, it is often not desirable to do so. It is frequently better to allow the invalid data to be entered into the system so that it can be corrected and made valid at a later time. There is another reason for handling validation rules independently. The validation of information often does not take place until a particular point in a logical object's process flow. For example, a merchandise order needs to be valid before it is processed. Sometimes particular validation rules are contingent and may only apply to an object under specific circumstances. Numbers on a budget may be considered valid if revenues "are close to expenses" whereas in an accounting system, for each journal entry, debits must equal credits to the penny.

Validation rules are often very complex and span multiple database tables as demonstrated in the following examples:

Validating that debits equal credits may involve looking in the Journal Entry Detail table, summing all of the journal entries of type "Debit," and comparing them to a summing of the journal entries of type "Credit."

Ensuring that the dates where a person resided in a specific place are later than that person's date of birth involves comparing information that resides in multiple tables.

These are just a few examples demonstrating that validation business rules are not simply properties in the database and cannot be easily handled through database triggers.

What is the appropriate solution for handling validation?

Traditionally, validation rules are embedded in code scattered throughout data entry applications, creating a maintenance nightmare where even finding a specific rule can be difficult, let alone changing it. “What are all of the validation business rules being enforced?” is an almost impossible question to answer. One solution to this dilemma is to collect all of the business rules into a single code structure. Traditional Oracle shops may create a large PL/SQL package. Java shops might create a large collection of classes to reside in the middle tier. Users of JDeveloper may use the Application Development Framework Business Components (ADF BC) wizards to generate many of the simple rules and use Java hand coding for the more complex rules. All of these approaches require manual writing of thousands (if not tens of thousands) of lines of code because the number of validation business rules in a typical system is usually very large. It is not until a system attempts to bring all of the validation business rules into one place that it becomes obvious how many there actually are. It is not uncommon for hundreds (or even thousands) of validation business rules to be contained in a system.

Not only are there a large number of validation rules, but these rules are constantly changing. Validation rules are more volatile than most other kinds of system requirements. This is not only due to changes in the business, but also due to the fact that users do not always recognize that the specified rules have many exceptions. The conditions under which many of the rules are enforced need to be continually refined.

Why haven’t existing systems collapsed under the weight of all of these validation rules? The easy answer is that IT professionals largely ignore these rules. Reliance is placed on users to enter data correctly and only apply validation rules when required to do so. Validation business rules are rarely explicitly and methodically gathered and enforced. Once a development shop decides to capture and enforce all validation rules in a given system, it is quickly apparent that there must be some way of dealing with these rules other than writing thousands of lines of code.

The optimal solution is to develop some type of rule grammar and/or repository to support all of the validation logic. This paper proposes a solution involving both elements, namely, a simple grammar as well as a small repository.

Case Study: The Problem – Description of Project

The project that inspired the solution described in this paper was a recruiting system designed and built for the U.S Air Force Reserve. During the recruiting process, there is a great deal of information gathered about individuals being admitted into the Reserve. In addition to the standard information, for security reasons, an individual’s education, employment, residence and medical histories are all gathered. Security clearances are handled by a different agency. Therefore, all of the information collected must be bundled into an XML file and sent off for processing by a separate organization. There are hundreds of logical rules associated with the data in this XML file. Some of these rules are very complex. For example, there cannot be any gaps in an individual’s employment history. The document describing all of the rules is summarized in a 22-page Excel spreadsheet, where each row is a separate rule. The narrative description of the rules is contained in a 250-page monograph.

From looking at these rules and performing initial testing, it was clear that the task of validating the XML file would have required thousands of lines of code. In addition, the intention was to reuse the solution for other military service branches, some of which might be interested in a Java-based solution or some other programming language. Therefore, having to create massive amounts of code that might have to be re-written in another language was not a viable option.

For these reasons, the decision was made to use a repository-based approach.

Case Study: The Solution – Validation Rules to Support Security Clearance Validation

Initially, a one-table repository to handle the validation rules was envisioned. For rules involving multiple tables, views would be created joining those tables, and rules would be written against these views. For example, in the Contract table, the “Start Date must be less than End Date” rule was written as follows:

```
:StartDate < :EndDate
```

For a rule stating: “Every department must have at least one employee,” a view would be created for the Department table that would include the number of employees in that department. The rule would look like this:

```
:NumberofEmployees > 0
```

This architecture would have worked, but would have meant building and maintaining a large number of views with a lot of hand-coded PL/SQL to generate the calculated columns. Therefore this approach was abandoned fairly early on in the project. The architecture was too limited. Managing the rules in this environment would not have been significantly easier than writing them all in code. Further, if forced to move into a Java/XML environment, potentially not even using a database, the entire concept of using views would fall apart.

It was therefore decided to extend the validation rule grammar to support, not only references to columns in the table, but also take advantage of the parent-child relationships between the tables in order to unambiguously reference data values in a different table when running a validation rule.

The syntax used to accomplish this is as follows:

`:_Child`	to refer to child table (the “many” table in a 1-many relationship)
`:_Parent`	to refer to parent table (the “1” table in a 1-many relationship)

In order to say that the employee hire date must be greater than the create date for the parent department, the rule would be associated with the Employee table and be written as:

```
:_Parent.Department.CreateDate < :StartDate
```

For referencing the child table, there are usually many child records, so some type of aggregator method is called. To indicate that each department must have at least one employee, the rule would be associated with the Department table and be written as:

```
:_Child.Employee.Employee_OID.Count > 0
```

The grammar was designed to support all standard aggregation functions (Sum, Count, Min, Max, etc.).

It was also necessary for this project to filter the child rows, so a WHERE clause was added to the method call. For example, to enforce the rule that the department must have at least one active (isActive = ‘Y’) employee, the rule would be associated with the Department table and be written as:

```
:_Child.Employee.Employee_OID.Count(where = :_Child.Employee.isActive = ‘Y’) > 0
```

If there were more than one relationship between the two tables, there had to be some way to identify which relationship was the correct one to use for the join. In this case, the foreign key column was specified in the code. In the above example, if the foreign key column in the Employee table were DepartmentEmployer_OID, then the rule would be associated with the Department table and be written as:

```
:_Child.Employee(DepartmentEmployer_OID).Employee_OID.Count(
where = :_Child.Employee.isActive = ‘Y’) > 0
```

It is apparent that these rules can eventually become quite complex to read so it is just as easy to write "real" code. There was one rule that was so complex that hand-coding was used, even though the grammar could support the description. However, out of the 22 pages of rules, there were only a few rules that required any hand coding. Even the "Debits must equal Credits" rule for a journal entry is not too difficult to enforce. This rule would be associated with the JournalEntry table and written as:

```
:_Child.JEDetail.Amount.SUM(where = :_Child.JEDetail.DrCrType = 'Dr')  =
:_Child.JEDetail.Amount.SUM(where = :_Child.JEDetail.DrCrType = 'Cr')
```

Contingent Rule Execution

The next problem encountered was that the rules were only contingently being executed. For example, if the "Debits must equal credits" rule is only for "Financial" transactions that are "ReadyToBeProcessed," then this rule would be associated with the JournalEntry table and be written as:

```
(:_Child.JEDetail.Amount.SUM(where = :_Child.JEDetail.DrCrType = 'Dr')  =
:_Child.JEDetail.Amount.SUM(where = :_Child.JEDetail.DrCrType = 'Cr') )
and :JEType = 'Financial'
and :Status = 'ReadyToBeProcessed'
```

Now the rule is starting to look a lot more complex. In this system, there might be many similar rules for "Financial - Ready to be processed" journal entries. It was decided to move the conditional part of the rule to its own property. Then the rule would be associated with the JournalEntry table and be written as:

```
Condition:      :JEType = 'Financial'
                and :Status = 'ReadyToBeProcessed'

Rule:    (:_Child.JEDetail.Amount.SUM(where = :_Child.JEDetail.DrCrType = 'Dr')  =
         :_Child.JEDetail.Amount.SUM(where = :_Child.JEDetail.DrCrType = 'Cr') )
```

This was also the point where the decision was made to use a multi-table repository. Since object reuse was the driving force in the repository design, every element was created to be reusable.

Description of the Rules Repository

The rules repository was implemented in an Oracle database. A Rule is grouped by Project. A Rule can belong to any number of Projects through the RuleUsage table. This was done so that the same rules could be used in various contexts. For example, there is one set of rules that validates whether or not a customer can be sent for a security clearance, but only a subset of those rules are relevant if the customer is being processed without a security clearance. Note that there is an active indicator in RuleUsage. This is so that a rule could be declared as active for debugging purposes.

A rule is optionally attached to a Condition. A Condition is a Boolean expression indicating whether or not the rule should be invoked. In practice, about half of all rules have conditions attached to them.

A Validation is the basic rule that is enforced (e.g. :StartDate < :EndDate). It has a many-many attachment to the Rule table through RuleDetail. A Validation can also be grouped using a ValidationGroup and then attached as a group to a rule.

Note that a Rule, Validation, and Condition are all attached to a specific table. Theoretically, these objects could be reused across tables if the column names were exactly the same. However, this idea seemed to unnecessarily complicate the model.

Error messages are built from the Condition and the Validation error_tx. The user enters a user-friendly error message for the Condition and the Validation. The system then uses that text to build the error text if the rule fails. For example, assume a rule on the Department table with the following repository values:

```
Condition
   Rule_tx = :Active_YN = 'N'
   Error_tx = 'the department is inactive'
Validation
   Rule_TX = :endDate is not null
   Error_tx = 'there must be a valid end date'
```

If the rule fails, the generated error message would be: "If the department is inactive then there must be a valid end date."

Since the system is created with reusable components, the error messages can be overridden for a specific usage. The Condition error_tx override is stored in the Rule class, and the Validation error_tx override is stored in the RuleDetail class.

A UML diagram of the rules repository is shown in Figure 1.

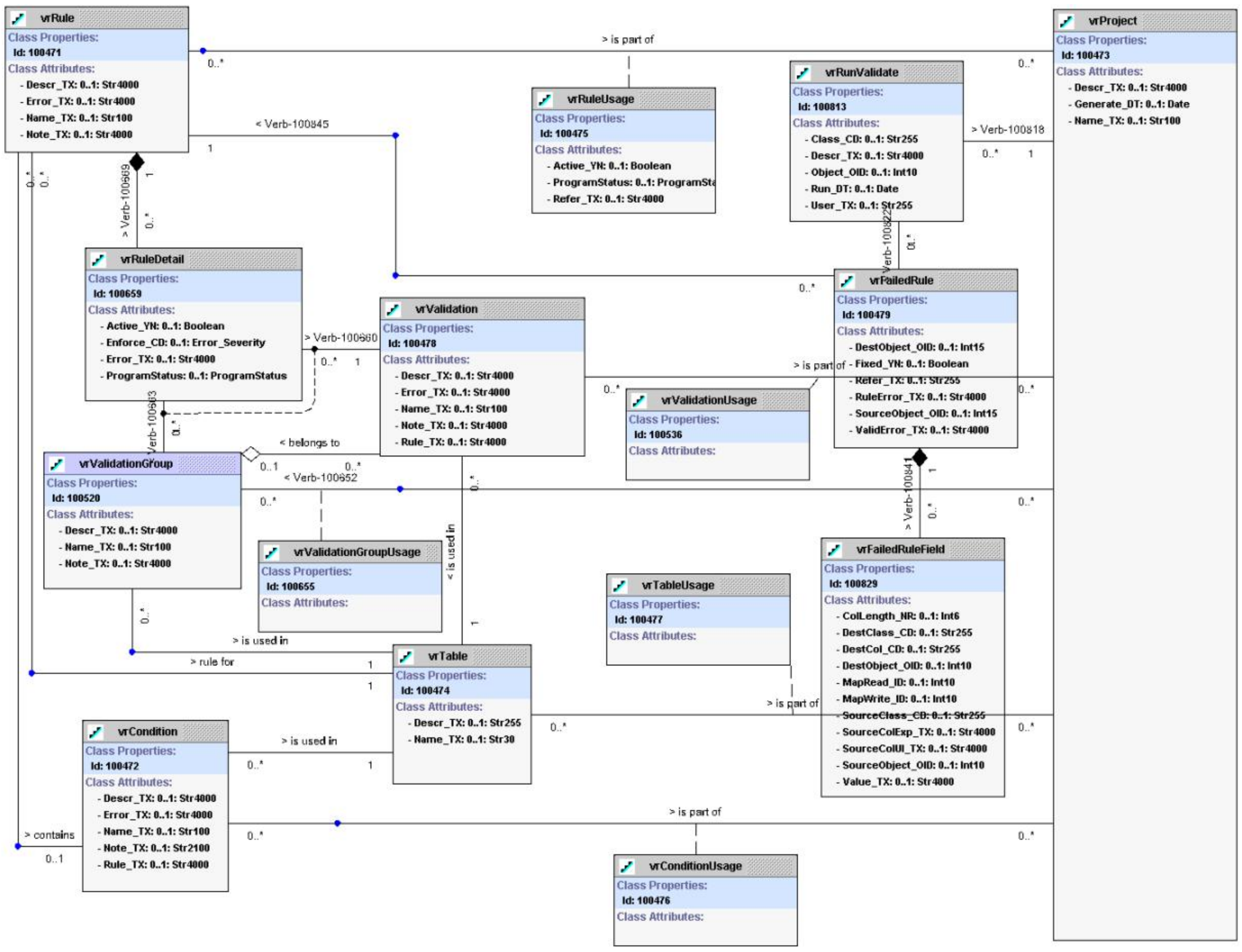


Figure 1: Rule repository data model

User-Friendly Rules

The rule syntax is reasonably friendly for an IT professional, but not very readable for most users. To increase readability, an English language translator for the rules was added. For example, in the previous example, the generated English language version of the rules would be:

“For each Department where the department is inactive (active indicator = ‘N’), there must be a valid end date (end date is not null).”

Note that the names of the columns in the generated rule include the user-friendly names from the repository rather than the actual column names.

The Validation Repository Manager

The level of object reuse in the system made it difficult to create a user interface for entering and maintaining rules. It is helpful to think of a simple master-detail relationship between the Rule table and individual validations. However, rules are reusable across different logical applications and individual validations are reusable across individual rules. The result was an application built with a simple master-detail relationship between rules and validations from a user perspective. The complexity of the many-to-many relationship is hidden from the person entering the rules. When a user specifies a new validation in a rule, a selection can be made from an existing validation, or a new one can be entered. When viewing validations or rules, users can see how many times each of these objects is used elsewhere in the system. This prevents those entering the rules from modifying an existing rule or individual validation and causing inadvertent side effects elsewhere in the system.

It was also challenging to communicate clearly to users what actual error message would be generated for a particular rule. The error message could be specified at the object or object usage level, both for conditions and individual validations. A button was added to the user interface to generate a sample error message that users could read to validate the appropriateness of the message in each context.

Screen real estate in such a complex system was also a problem. In addition to rules, error messages and descriptions in large text boxes, there was an English language translation of the rule displayed. This problem was solved by using a combination of on demand popup windows and scrolling text fields.

It became important to find and navigate to particular validations and rules. A separate Rule Finder was built where users could enter search criteria. Rules satisfying the entered criteria would appear in a list. Double clicking the rule navigated to the selected rule in the system.

Rule Enforcement

Once the rules have been specified, they must be enforced. For validation, this means that when objects fail a validation rule, some error message is generated. Up to this point, rule enforcement has not even been mentioned, nor has any specific environment been specified. This is the core idea of the business rules approach. The representation of the business rules is independent of the enforcement of the rules. It is possible to determine the appropriate logical grammar to describe the rules and in what context they will be enforced without making any assumptions about the physical representation of the data or the way in which the rules will be enforced.

In using the rules approach to building systems, approximately 30% of the time is spent designing the repository and developing the rule grammar. Another 10% of the time is spent creating the appropriate user interface to work with the repository. Writing the code generator/enforcement mechanism requires approximately 10% of the project time. Entering the rules into the repository consumes the remaining 50% of the time. This breakdown indicates that even drastic changes to the way in which rules are enforced will only require a relatively small amount of effort (a few days to a few weeks of effort).

There are various alternatives, each of which is achievable and surprisingly simple to implement:

Interpreted mode: Leave the rules in the repository and generate the code using an EXECUTE IMMEDIATE mechanism. This is the simplest alternative and is frequently used for a proof-of-concept. The downside is performance. Every rule that is enforced requires an EXECUTE IMMEDIATE operation. If there are only a few rules, this mechanism works well; however, if the number of rules is in the hundreds or thousands, severe performance degradation will be experienced.

Generate PL/SQL into the database (Compiled mode): Using this alternative assumes that the data exists in Oracle tables and straightforward PL/SQL will be generated to execute the validation logic. The downside to this approach is that changes to the rules require regenerating the code. If everything is generated into a single package, this can require up to 30 seconds each time the code is regenerated. A 30-second recompile time might not seem long, but in a debugging cycle with numerous iterations of rule tweaking, waiting 30 seconds for each recompile is not desirable. The main benefit of this approach is its speed at runtime. By contrast, using the interpreted mode instead might require 30-40 seconds at runtime.

Generate Java to run against XML: In order to perform validation away from the Oracle database (for example, to run on an isolated laptop for later upload to the database), it is necessary to validate without access to Oracle. This can be done by generating Java code to perform the same validations against an XML file. PL/SQL code generators have been successfully migrated to an XML data source using the Sunopsis utility that allows the use of SQL to access an XML data source.

For the project described in this paper, the Compiled mode approach was used for version one. For the next version, the third approach will be used.

In addition to simple error messages, it was also important for the Validator to provide information allowing users to identify the reason(s) for the validation failure. The following information was provided to users:

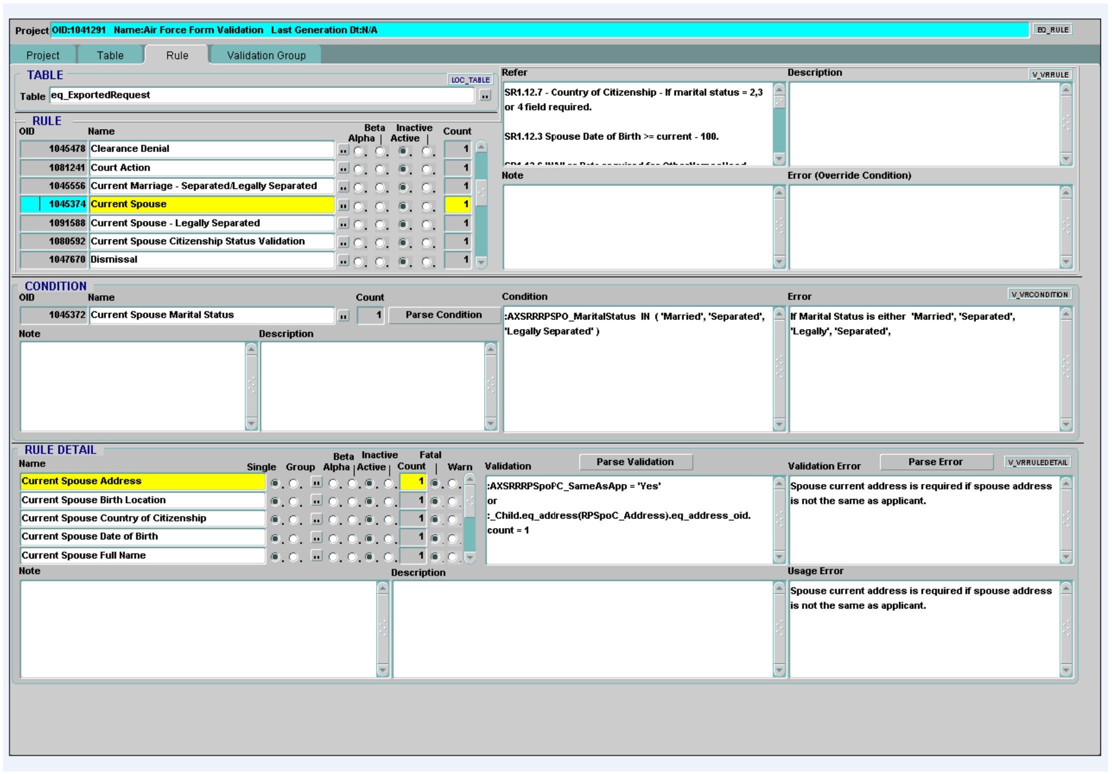
Object being validated when the rule failed

Value of each field referenced in the rule

Place in the user interface where the information could be corrected

The ultimate goal was to allow users to double click any field referenced in the error message and navigate to the appropriate portion of the application user interface.

A screen shot of the repository manager is shown in Figure 2:



Project OID:1041291 Name:Air Force Form Validation Last Generation Dt:N/A

TABLE

Table eq_ExportedRequest

RULE

OID	Name	Alpha	Beta	Active	Inactive	Count
1045478	Clearance Denial	○	○	●	○	1
1081241	Court Action	○	○	●	○	1
1045556	Current Marriage - Separated/Legally Separated	○	○	●	○	1
1045374	Current Spouse	○	○	●	○	1
1091588	Current Spouse - Legally Separated	○	○	●	○	1
1080592	Current Spouse Citizenship Status Validation	○	○	●	○	1
1047670	Dismissal	○	○	●	○	1

Refer: SR1.12.7 - Country of Citizenship - If marital status = 2,3 or 4 field required.

SR1.12.3 Spouse Date of Birth >= current - 100.

Description:

Note:

Error (Override Condition):

CONDITION

OID	Name	Count
1045372	Current Spouse Marital Status	1

Condition: :AXSRRRPSPO_MaritalStatus IN ('Married', 'Separated', 'Legally Separated')

Error: If Marital Status is either 'Married', 'Separated', 'Legally', 'Separated',

Note:

Description:

RULE DETAIL

Name	Single	Group	Alpha	Beta	Active	Inactive	Count	Fatal	Warn
Current Spouse Address	●	○	○	○	●	○	1	●	○
Current Spouse Birth Location	●	○	○	○	●	○	1	●	○
Current Spouse Country of Citizenship	●	○	○	○	●	○	1	●	○
Current Spouse Date of Birth	●	○	○	○	●	○	1	●	○
Current Spouse Full Name	●	○	○	○	●	○	1	●	○

Validation:

```
:AXSRRRPSpoPC_SameAsApp = 'Yes'
or
:_Child.eq_address(RPSpoC_Address).eq_address_oid.
count = 1
```

Validation Error: Spouse current address is required if spouse address is not the same as applicant.

Note:

Description:

Usage Error: Spouse current address is required if spouse address is not the same as applicant.

Figure 2: Repository Manager User Interface

Lessons Learned: The Benefits of Repository-Based Development

This project was a great success for the repository-based development approach. The time spent building the architecture and generators was more than offset by the speed at which hundreds of rules could be supported. The validation engine built makes further rule specification extremely rapid.

As more and more business rules are moved into repositories and out of code, the benefits of repository-based coding have become more and more apparent. It is easier to make objects reusable and they are far easier to manage in a repository than as code objects. It is also easer to make changes to the repository and perform an impact analysis of potential changes.

At some level, it feels like a “better” way to do programming. With very little development time required to create a repository manager, a more efficient IDE for application development is created than can be built using tools with millions of dollars of development effort such as TOAD for PL/SQL, JDeveloper or Eclipse for Java. One can only wonder what could be achieved in repository-based development if it attracted the attention of one of the major software development vendors.

About the Author

Dr. Paul Dorsey is the founder and president of Dulcian, Inc. an Oracle consulting firm specializing in business rules and web based application development. He is the chief architect of Dulcian's Business Rules Information Manager (BRIM®) tool. Paul is the co-author of seven Oracle Press books on Designer, Database Design, Developer, and JDeveloper, which have been translated into nine languages. He is President of the New York Oracle Users Group and a Contributing Editor of IOUG's SELECT Journal. In 2003, Dr. Dorsey was honored by ODTUG as volunteer of the year, in 2001 by IOUG as volunteer of the year and by Oracle as one of the six initial honorary Oracle 9i Certified Masters. Paul is also the founder and Chairperson of the ODTUG Business Rules Symposium, (now called Best Practices Symposium), currently in its sixth year and the J2EE SIG (www.odtug.com/2005_J2EE.htm).