Agile Business Rule Processing

A primer for implementing business rules

Haley

The Haley Enterprise, Inc.
Solutions that do what they're told

INTRODUCTION

The term “Business Rules” is a recent darling of Information Technology, but the devil is in the details. Business Rules are darlings because business is rife with them; they are food for IT professionals. Business Rules bedevil IT since the procedural or object-oriented languages and the database technologies used by IT do not directly support or implement them. By examining the requirements for Business Rule Processing, IT professionals readily comprehend why checking rules within member functions or triggers would be a viable solution for Business Rule Processing, but only if the implementation is done automatically and in a manner that results in acceptable performance.

After examining the challenges in either manual or automatically generating code for business rules, IT professionals find appreciation for the fact that manual codification of business rules using only procedural, object-oriented, or relational approaches is not a viable approach to Business Rule Processing. After establishing the basis for such understanding, we present an example of a viable approach that retains all the familiarity of established IT by bringing business rules to bear on relational databases seamlessly encapsulated within objects using inheritance and operator overloading to the extent that a programmer need not be aware that the objects can be persistent within a database that is being monitored by rules in real time.

PROCESS VERSUS KNOWLEDGE

Most Information Technology groups in corporate America have experience with Business Process Reengineering and Object-Oriented Programming. Nonetheless, many of these same IT organizations have come to understand that the principle knowledge assets of their corporations are not the work-flow-charts that can be expressed in procedural, process-oriented languages or the data structures and algorithms that can be encapsulated in software objects or components using object-oriented languages. Reluctantly perhaps, information technologists are admitting that the competitive advantages developed by their employers are based on knowledge about how to conduct business.

Many Information Technologists remain somewhat reluctant to admit that knowledge is the principle power. Knowledge about the business invariably arises from operational personnel and senior management. IT seeks to understand the business, model it as accurately as possible using procedural, object-oriented, and relational techniques, and automate those models using such technologies. IT's discomfort with business knowledge is that it is not *necessarily* procedural. Executives and managers dictate how business should be conducted. Engineers and operational personnel dictate constraints on how business can be conducted. A Harvard MBA does not necessarily conceive of how to operate a business by drawing a flowchart. Nor does an engineer express physical or electrical constraints in priority order, tables, or flowcharts. Nonetheless, IT must discern what these business authorities know or have decided and support the operational business with automation that brings such knowledge to bear or that supports the operation with decision support systems that encourage or constrain employees to operate within management or engineering guidelines or constraints.

To concede that knowledge - not flowcharts or data structures - is the principle asset supporting competitive vitality begs for further clarification. If the flowcharts are not intrinsic to the business but are conceived by IT in its endeavor to automate and support operations, is there - in fact - a business process? Was Business Process Reengineering about using new technology or about implementing systems that admitted more fundamentally that business processes were, in the first place, a figment of technologists' myopia? After all, if all you have is a hammer every task seems like driving a nail. If all you have is flow charts with data structures, everything looks like a process.

Business Process Reengineering has - for the most part - run its course. Perhaps the principle software contribution of BPR has been the emergence of Object-Oriented Programming. But OOP has not changed the fundamental problem facing IT: how can business knowledge be identified, codified, automated, and maintained through its life cycle. OOP is currently the best programming technology for procedural systems. Still OOP does not address the issue of implementing business knowledge that is not

- in and of itself - procedural. Like BPR, OOP does not help with the problems of managing and automating business knowledge.

Many Information Technologists have come to recognize the limitations of BPR and OOP for managing and automating business knowledge. As a result, the term “Knowledge Management” has become increasingly important in IT.

KNOWLEDGE MANAGEMENT

Knowledge Management, in its most general sense, involves managing corporate knowledge assets. However, in the context of IT, Knowledge Management is of little value if managing knowledge does not result in automation or decision support systems. Consequently, Information Technologists are not particularly compelled to adopt Knowledge Management as vended by document management companies. Although such products certainly have utility in that they record and provide access to corporate knowledge embodied in documents, such products cannot automatically bring such knowledge to bear in any operational sense.

Knowledge Management in IT requires the ability to express knowledge in a form akin to that in which business executives, managers, engineers, and other operational personnel express it and to allow for the body of such knowledge to evolve, allowing for the inconsistencies and incompleteness of the entire knowledge base that exists in every real business. Even if such Knowledge Management capabilities were obtained by IT, they would not be adequate to the task unless they further facilitated the production and maintenance of reliable and adequately performing automation and decision support systems that could be integrated with more algorithmic, straightforward procedural information systems such as legacy applications that implement back office processes.

IT has already taken the first step toward such Knowledge Management. IT has adopted the notion that business knowledge is commonly expressed by executives, managers, engineers, and other operational personnel in the form of Business Rules.

Managing knowledge expressed as Business Rules requires processes and methodologies for identifying the business rules. Such processes and methodologies are well established and already in place within IT. The requirements and functional specifications collected and documented by systems analysts are rife with business rules. Implementing what the business specifies as formalized by systems analysts is where the problems persist.

BUSINESS RULES

Business Rules identified by systems analysts are the principle content of front office automation and decision support system specifications. This has been the case since the late seventies when software development first stepped from almost algorithmic batch or transaction processing in the back office onto terminal screens in the highly conditional decision support environment of the front office: the dawn of the applications backlog.

The practice of automating business rules has traditionally consisted of applications programmers attempting to understand the business rules, conceive of how they might interact in operation, conceive of how they might be ordered, nested, and otherwise transformed from independent statements into a unified, all encompassing flowchart - never seen, but meticulously crafted in obscure but intricate programming syntax. The resulting implementation was, of course, The Business Process, which business authorities were too ill informed to conceive and from which we Information Technologists would eventually attempt to reverse engineer the business rules after convincing executive management to endorse BPR so that we could Reengineer the Business Process.

The principal failing of most procedural approaches to encoding business rules is the loss of identity that each business rule encounters as it is merged with other rules into an application flowchart. Merging rules into a single flowchart forces a total ordering upon them. Even though business policies may be unordered in practice, they will be completely ordered in a procedural implementation. Changing even a single rule can have a devastating impact on such an implementation. One more rule can expose all manner of latent assumptions and rip an entire implementation asunder.

PROCESSING KNOWLEDGE

Implementing business rules directly eliminates any assumption or tricky encoding that might subsequently become errant. Implementing business rules directly means to avoid any merging of the code that implements one business rule with code that implements another. By implementing a business rule we refer primarily to the code which checks whether a rule is applicable.

The principal issue in implementing business rules is checking their conditions. Checking rule conditions has two principal aspects: “how” and “when”. If each rule’s conditions are to be checked by code that stands alone (i.e., by an implementation that is not merged with, but remains independent of, all other rule implementations, as discussed above) we have a notion of “how”.

The obvious answer to “when” to check a rule is “whenever it may apply.” Easier said than done, however. Consider that a rule may involve multiple conditions. Checking a rule whenever it may apply requires checking the rule wherever any one of its conditions may be satisfied.

Implementing the logic to check a rule involves identifying all points within a flow chart or procedural code that constructs, destroys, or modifies data that is examined in the conditions of that rule. At each such point either a call to a procedure that checks the rule must be inserted or in-line code that checks the rule must be written. Failure to do so at any such point introduces the error prone - if not already incorrect - assumption that the rule is not relevant at that point.

Implementing the logic to apply a set of business rules involves identifying and inserting code at all such points for all conditions across all rules in the set. If two rules have similar (but not necessarily identical) conditions, two insertions should occur at all relevant points in the existing flow chart or procedural code.

When checking multiple rules at any one point within a flow chart or procedure the temptation to optimize the inserted logic may seem overwhelming. By overcoming this temptation, an IT professional addresses the principal “how” of business rule programming. That is, each rule is checked when (or where) needed and without latent, error prone assumptions that would make the application difficult to maintain as business rules are added, deleted, or modified.

There are some important details remaining about “how” and “when” to check business rules. How is the code that checks a rule actually to be written? And doesn’t “when” really correspond to “where” within a flow chart or procedure? The unfortunate answer is that “how” usually depends on “when” or “where”.

Consider, for example, a rule that applies if two fields of a record have specific values. Implementing such a business rule might involve identifying all points in a flow chart or procedure where a value is assigned to either of those fields in any record. At each point, code could be written to see if the value was the specific value required for that field and, if so, further check to see if the other required value existed in the second field. The “how”, i.e., the code written, depends on “where”.

The code to check a business rule could more reasonably, perhaps, be encapsulated within a predicate subroutine. This predicate would take a record as an argument and determine whether the rule applied. Although slightly less efficient, this approach would eliminate “how” depending on “where” for simple business rules that examined individual records. Even so, this “how” must be invoked every “where” in the procedural code.

Before dwelling upon how to maintain the invocation of “how” code every “where” it should be invoked from procedural code, it is important to realize that arbitrary rules cannot correspond to individual procedures. As described in the preceding paragraph, Business Rules whose conditions involve more than a single record cannot be encapsulated within a single predicate.

Consider, for example, a rule that applies when a record of one type has the same value for a field that occurs in another field of another record type. Implementing such a rule involves identifying all points at which a record of either type is constructed and all points at which either field is modified. The code to be inserted at points where a record of the first type is constructed or at which an assignment is performed on the field of the first record will differ from the code needed at points where a record of the second type is constructed or at which an assignment to the field of the second record is performed. Each of the code fragments could be encapsulated within procedures, but the argument of the first procedure would be a record of the first type and the argument of the second procedure would be a record of the second type.

IMPLEMENTING RULES

The preceding paragraphs lay out the only viable approach to implementing business rules in sustainable applications. They explain when and where business rules should be checked and how that checking should be implemented. The remaining issues are in:

- Implementation reliability,
- Programmer productivity, and
- Application performance and scalability.

The reliability of a business rule implementation is driven by the reliability with which:

1. Every "where" that each rule should be considered is identified and
2. The reliability with which every "how" that implements each rule is coded.

The productivity with which programmers implement business rules is a function of:

1. How much "how" code needs to be written
2. How difficult it is to identity every "where" each rule should be considered, and
3. How much code needs to be written at each "where".

Application performance can vary widely depending on the precise nature of the implementation. In most cases, performance degrades as rules are added. However, certain algorithms used in commercial rule engines have been shown to have performance that is "asymptotically independent of the number of rules."

RELATIONAL DATABASE TRIGGERS

In order to achieve high application reliability, the need to correctly identify every point at which a rule may be or become relevant within a flow chart or procedure must be alleviated. This is possible in IT applications where the business rules apply to rows in a relational database or that are implemented in an object-oriented programming language.

Modern relational databases provide for the attachment of procedures that can be "triggered" whenever a row in the table is added, removed, or changed. By attaching the appropriate "how" routine for each condition of a rule to the corresponding table business rules can be brought to bear without modifying application procedures that manipulate the database. Some IT professionals have had substantial success in implementing business rules in this manner and certain commercial rule engines support precisely this functionality.

OBJECT-ORIENTED PROGRAMS

Most IT applications store their objects in relational databases to which triggers could be attached. Object-oriented programming languages support the added flexibility of invoking the "how" routines that implement rules within the constructors or encapsulated assignment operators of an object class. In effect, these constructors and assignment operators correspond exactly with the triggered procedures provided by a relational database in which the objects might persist. By encapsulating every "where" that a "how" routine is invoked within the definition of object-oriented classes, the application code that uses those classes need not consider (i.e., need not be modified to check) rules.

CONDITION VERSUS QUERY ISSUES

The object-oriented approach suffers from a limitation addressed in the relational approach. Within a triggered procedure the relational approach can execute an arbitrary query over the entire database. Such a query in conjunction with the triggering row corresponds to the potentially complex conditions of a rule from the perspective of one of its conditions.

The conditions of a business rule correspond to arbitrary query against a model.

In an object-oriented program, each routine that implements a rule from the perspective of one of its conditions must explicitly traverse pointers compiled into the data model and iterate over lists of related objects explicitly maintained by all the classes involved. The infrastructure required to support and maintain these pointers and lists is a tremendous undertaking in its own right. In the relational approach, arbitrary joins can be executed within the triggered procedures without recompiling any application code and without maintaining any pointers or lists.

The flexibility of the relational approach to business rules usually tips the scales in favor of triggered implementation over object-oriented methods. The reasons are similar to those that motivated IT to abandon network databases in favor of relational databases. These are the same reasons that limit the ability of object-oriented databases to supplant relational databases.

PROGRAMMER PRODUCTIVITY

Although the relational approach requires substantially less implementation effort and offers much greater flexibility to accommodate unanticipated business rule changes than the object-oriented approach, both approaches involve writing at least one procedure per condition per rule and attaching each such procedure as a trigger or invoking it from within every relevant encapsulated assignment operator supported by a class. In an object-oriented approach, it is typically also necessary to code and attach each such procedure in the constructor of its triggering class and, if any rule involves the class in a negative condition, some procedure will need to be invoked from its destructor.

Although either the relational or object-oriented approach requires an equivalent number of procedures, the object-oriented approach requires code to be attached in many more places. In either case the number of such procedures is proportional to the number of conditions which is invariable greater than the number of rules.

As discussed above, coding business rules independently eliminates the extensive and error prone analysis and latent assumptions concerning rule interactions that manifest themselves in complex, unreliable, and unsustainable applications. Although coding business rules requires the coding of a procedure per condition, each procedure is more straightforwardly implemented from the expression of a rule than with any approach that merges them into a unified flow chart or procedure. Consequently, this code is generated more productively and is more sustainable than would otherwise be the case. Moreover, most if not all of this generated code would appear in a unifying flow chart, if one could be obtained. The only difference would be the effort to determine how to merge the implementation of a rule with that of others and the difficulty of understanding or changing the logic that implements a rule after it is spread throughout a flow chart or procedural program.

AUTOMATIC CODE GENERATION

Certain commercial rule engines provide syntax for expressing rules and a compiler of sorts that automatically generates the procedures that can be embedded within object models or attached to relational database tables to check rules whenever they may become applicable. In effect, such tools automate the code that would otherwise be coded by programmers to check business rules. Needless to say, such products are tremendously valuable. If a business rule corresponds to even a dozen lines of code and an application consists of merely hundreds of rules, the generation alone of such code may be worth in excess of $100,000. And the result, presumably, would be assuredly correct where hand written code would certainly not be.

Unfortunately, whether handwritten or automatically generated procedures are used to implement rules and whether they are attached as triggers or invoked from encapsulating methods, the performance of the application will degrade as rules are added. The only solution to the problem is to generate the code that checks rules and attach it in the manner of the Rete Algorithm. A detailed review of the Rete Algorithm is outside the scope of this paper, but the pertinent essence is that only the Rete Algorithm is known to provide performance that is both fast and scalable. Many independent benchmarks have shown the Rete Algorithm to outperform any other approach by many times, even for just a few dozen rules operating only a few dozen rows of data or objects. As the number of rules increases into the hundreds or thousands, adding rules is known to have no effect on the performance of the Rete Algorithm while every other approach slows down.

SCALABLE PERFORMANCE

The Rete Algorithm is essentially relational. It performs joins and selects in the same manner as relational databases. Rules written using the Rete Algorithm do not require an underlying pointer and list infrastructure that object-oriented models require in order to implement rules. The lack of such infrastructure requirements allows the Rete Algorithm to bring rules to bear even when the data model does not anticipate their conditions in advance.

In effect, the Rete Algorithm automatically generates all the “how” procedures that implement each rule and attaches them every “where” that a rule may be or become applicable and ensures that a rule will be determined to be applicable “when” ever it might be applicable.

SEAMLESS ENCAPSULATION

In order to bring the Rete Algorithm to bear within object-oriented programs it is necessary to leverage its relational capabilities within object-oriented models. Doing so allows business rules to apply to a object model without requiring all of its joins to be reflected in pointers between objects or list maintenance code in various member functions throughout a class taxonomy.

> *The agility required to accommodate new or changing business rules in object-oriented programs requires dynamic rule conditions in the same manner that relational databases support dynamic queries. To provide this functionality an object-oriented model must encapsulate a corresponding relational database. Such an encapsulation provides not only persistence but the ability to bring business rules to bear without inserting any code into applications that use the encapsulating data model.*

The essential requirements for encapsulating the capabilities of a relational database within an object model are that constructing an object must correspond to adding a row, destroying an object must correspond to deleting a row, and performing an assignment to an attribute of an object must correspond to changing a row. The challenge in implementing such an encapsulation is that the constructor must "know" to which table a row is to be appended, the destructor must "know" which row is being deleted, and the attribute or member datum to which a value is assigned must "know" which cell is being changed.

ABSTRACTION CONSTRAINTS

Encapsulating the Rete Algorithm within an object-model that frees application programmers from any need to be concerned with checking dynamic business rules adds constraints on the implementation. For example, if assignment of a value to an attribute of an object is encapsulated within an object model by a function that takes four arguments (i.e., the table, the row, the column, and the new value) it is possible to update the corresponding cell in a relational database where the object persists. Even though the number of arguments may be reduced to three if the object is provided and its class supports determination of the row and table in which the object persists, application code would be required to pass the object with every assignment. In C++, for example, this would preclude the use of assignment and auto-arithmetic operators (e.g., "=", "++", "--", "+=", "-=", "*=", etc.) Moreover, such an approach is incompatible with the passing of "reference" arguments that are part of the standard practice in object-oriented programming. A reference argument is provided to a function without the object that includes it. Consequently, if the receiving function modifies its reference argument it is not possible to determine which object (i.e., which row) has been modified.

AGILE BUSINESS RULES

One of our rules engines, Rete++ encapsulates the Rete Algorithm in C++ without compromising the use of C++ assignment or auto-arithmetic operators or reference arguments. Furthermore, Rete++ classes encapsulate the application of the Rete Algorithm in a manner that allows rules to be changed without recompiling any C++ code.   Consequently, Rete++ is an example of how business can remain agile by accommodating new or changing business rules within advanced applications that use object-oriented programming with or without a relational data store

THE HALEY ENTERPRISE

The Haley Enterprise is a global leader in the commercialization of Artificial Intelligence technology. We develop commercial software based on our expertise in rules based and natural language processing. Haley is privately held and has been consistently profitable throughout our existence. Among our customers are Fortune 100 companies such as Merrill Lynch, Cigna, AT&T and Ford and government entities such as the Department of Labor and the US Army. We are poised to lead the wave into ubiquitous computing based on our commercial offerings and extensive experience using the four cornerstones of artificial intelligence – reasoning, memory, language and learning.

Our most comprehensive, embeddable, efficient, scalable and integrated AI technologies coupled with decades of experience solving the most demanding knowledge management and automation challenges allow us to confidently assert that we can put your knowledge to work, too.

For more information, contact:

The Haley Enterprise, Inc.
1108 Ohio River Blvd.
Sewickley, PA 15143
Tel: 412-741-6420
Fax: 412-741-6457
Web: www.haley.com