

Extending Business Objects with Business Rules

Isabelle Rouvellou,
Lou Degenaro
IBM T.J. Watson Research Center
E-mail:isabelle@watson.ibm.com

Kevin Rasmus
Country Companies
Insurance

Dave Ehnebuske,
Barbara McKee
IBM Software Solutions

Abstract

It is common to imbed business rules within the code of distributed object systems. When business practices and/or policies change, as they often do, it is difficult if not impossible to correctly reflect those changes in the applications implementing them. This paper describes a framework that enables enterprises to develop distributed business applications that systematically externalize the time- and situation-variable parts of their business logic as externally applied entities called business rules. Decoupling business rules from the application can provide a number of advantages. Because business rules are external to the applications that depend upon them, the variable business logic contained in them can easily be changed. Because the management of the externalized business rules is done explicitly through a rule management facility, it is easy to understand what rules exist and to locate those that need to be changed. The Accessible Business Rule (ABR) framework is available as early test function in IBM Component Broker 2.0 and 3.0 (a part of WebSphere Enterprise).

1. Introduction

Often when people explain to “outsiders” how a business process works, they start by explaining the broad sweep of the process. Then they go back, filling in the details as a series of variations on the theme. For example, when explaining how a rate quote for automobile insurance is arrived at, an expert in the field will explain that, “It depends on where a particular vehicle is being driven, by whom, and on how much of what is being covered.” Then the details come into play. Exactly what class of vehicle are we talking about? Is the principal driver a young person? What sort of driving record does this person have? Is the vehicle used to drive to work or school? How far? Where is the vehicle garaged? Have we written other insurance for this insured? What coverages are we talking about? What deductibles? Does this meet state requirements? How much of this information are we allowed to take into consideration in this jurisdiction? And so on. Additionally, the details keep changing as new types of policies are offered and as the regulations in the various jurisdictions change.

The Accessible Business Rules Framework (ABR) for Component Broker [1] is designed to enable developers to build business applications that cope gracefully with requirements such as these. It allows applications to be structured so that the core behavior of the business, application, and user interface objects are built-in, while the variations are managed and applied externally. ABR found its basis in two earlier consulting engagement projects with IBM customers. A predecessor of ABR has been used since early 1998 by a production application at Country Companies Insurance (details can be found in [2]).

The ABR approach starts with extending the object oriented analysis and design to include identifying the rules and the points of variability in a business process. Where benefits (reuse,

reduced maintenance, consistency, etc.) can be achieved, the rule is externalized and the point of variability is, during development, translated into a *trigger point* (i.e., a piece of code in an object method that interfaces with the ABR runtime to attach and execute business rules dynamically during application execution). The trigger points can be generated using a code generation tool, they may be inherited, or they may be hand-coded.

The definition of business rule that ABR has adopted is: “A *business rule* is a statement that defines or constrains some aspect of the business. It is intended to assert business structure or to control the behavior of the business” [3]. ABR further refines the definition by distinguishing between *constraint*, *invariant*, *derivation*, and *classification*. Rules may equally well constrain or prescribe business behavior. For example, a given rule may check that an operation has met all of its obligations (and so be a postcondition *constraint*). Another might ensure that multiple changes made by an operation are properly related to one another (that is, ensure that some *invariant* is not violated), while still another may calculate a value (and so act as a *derivation* prescription). Finally, some rules may only apply to certain classification groups (for example, a person *classified* as a full-time employee might have different rules applied to them than someone *classified* as a part-time employee would).

Rules are pieces of business practice that tend to change more often than the low-level data structures and functions (more often than the base object model). Changes may come from within the company or may be mandated from outside; typically by regulatory agencies. The ABR approach encourages rules to be rendered explicit and made external to the objects that make up an application. Such an assertion naturally raises the question, “why is this a good idea?”.

Broadly speaking there are two reasons. First, making business rules explicit allows an enterprise to understand the rules it is currently operating under. When they are not explicit, business rules may be known in only a few people's heads and their implementation may be buried in application code. People forget, or sometimes leave the company, and rules buried in application code are often replicated and inconsistently applied. Second, externalizing business rules allows an enterprise to cut its cost of application maintenance. Finding and managing rules when they are not externalized becomes an increasingly expensive proposition as time passes and as the number of rules buried in the code grows. For example, the effort required to find all the places in a system in which an eligibility (such as a “senior citizen discount”) is determined can be substantial. Ensuring that the application system complies with new rules when, say, the State Commission on Aging makes a new definition for “senior citizen” is often nearly impossible.

Some of the benefits that we hope to see as ABR is applied to business systems are:

- Explicit documentation of business practice decisions. Externalizing rules gets them out of people's heads, out of the application code, and up on the table where they can be seen.
- Clearer understanding of application behavior. Externalization makes it possible to inspect the application to see which business rules are being applied, under what circumstances, and when.
- Increased consistency of business practices. Because externalized rules promote reuse and facilitate clear understanding of business practice decisions, they provide a basis for improving business practice consistency within and across applications.
- Decreased maintenance and testing costs. Externalized rules have a clearly defined scope and are not tightly coupled to the application code. This makes them easy to modify and quick to test; decreasing costs and improving maintenance cycle time.
- Improved administration of business practice decisions. Externalization, change history, and inspectability all promote clear ownership and thereby a better definition of who can change rules and under what circumstances.

- Increased confidence in predicting the business impact of proposed changes. Because rules are available for inspection, and have well-defined scope, they make it easy to understand the likely impact of changes and to predict whether contemplated changes or additions will have unwanted ripple effects.

Benefits rarely come without some cost. Those costs include:

- Performance. There is an overhead associated with externalizing rules. Through caching and other techniques the performance overhead is minimized, but there is still a small impact.
- Rule administration. When rules are externalized, people other than developers can be in charge of administering them. This new task requires new procedures and increased coordination.
- Testing and debugging. ABR places an additional layer of complexity within the application. Though the framework and tools make the environment simple for most, if problems arise, the person doing debugging will need to gain a detailed level of understanding.

The paper proceeds as follows. Section 2 explains how we extend the application analysis and design processes to include identifying rules. Section 3 describes the framework runtime and Section 4 the administration component. Section 5 presents some of the lessons we learned while building the framework. Section 6 compares ABR with other rules systems. Finally, in Section 7, we summarize and list some further research.

2. Extension to Object Oriented Analysis and Design

We have not formalized a methodology for rule analysis and design at this point. There were, however, a number of activities that we undertook. Additionally, there are a number of concepts that need to be taken into consideration during the analysis and design process. Some of them complement and/or confirm concepts used by a number of consulting companies specializing in business rule analysis and design [4][5].

2.1. Extracting Variability from Use Cases

Trigger points are found during analysis by inspecting the use cases or user interaction scenarios that are typically developed as statements of requirement as input to the analysis process. A fragment of a use case is shown below:

The vehicle is entered into the system or chosen. The customer service representative attempts to locate the named driver in the system. If the driver is not found, she/he is added to the system and then picked. Otherwise the found driver is just picked.

If the vehicle is an auto, anyone between the ages of 16 and 75 can be picked as a driver. If the vehicle is a truck, only drivers 16 to 70 years old can be picked. And if the vehicle is a motorcycle, drivers 14 to 65 can be picked.

After the driver has been picked, a rate quote can be performed ...

Keywords can indicate the points of possible rule externalization: statements such as “if X is in a special category Y” (e.g., “if the vehicle is a truck” above), “except when”, “unless”, “depends on”, etc. It is equally important to identify the places where externalization of rules would be beneficial (as gauged by the benefits described earlier). Potential rules can then be marked as input into the object interaction diagramming process

2.2. Annotating Object Interaction Diagrams (OIDs)

While creating OIDs, we added a symbol (a target with an arrow) to represent a trigger point. Associated with the symbol was some text indicating which rules should be fired at that trigger point. An example of an annotated OID is shown in Figure 1.

While laying out OIDs based on the use cases that we had already analyzed, we found the identification of trigger points fairly easy. A number of patterns were observed. Below are some of the rules to look for and where the trigger point might be placed:

1. Validation of edits on create and set methods.
2. Referential integrity of edits on methods that set references.
3. Cardinality checks at a consistency point (a point in time where all of the data is expected to be self consistent).
4. Required fields checks and cross field edits at a consistency point.
5. Constraints or derivations that have a high potential for reuse (especially if the algorithm is complex) at any appropriate point.
6. Constraints or derivations that a business desire to be consistent across applications (at any appropriate point).
7. Constraints or derivations where the business wants to decouple the maintenance cycle for a rule from the maintenance cycle for the code (at any appropriate point).

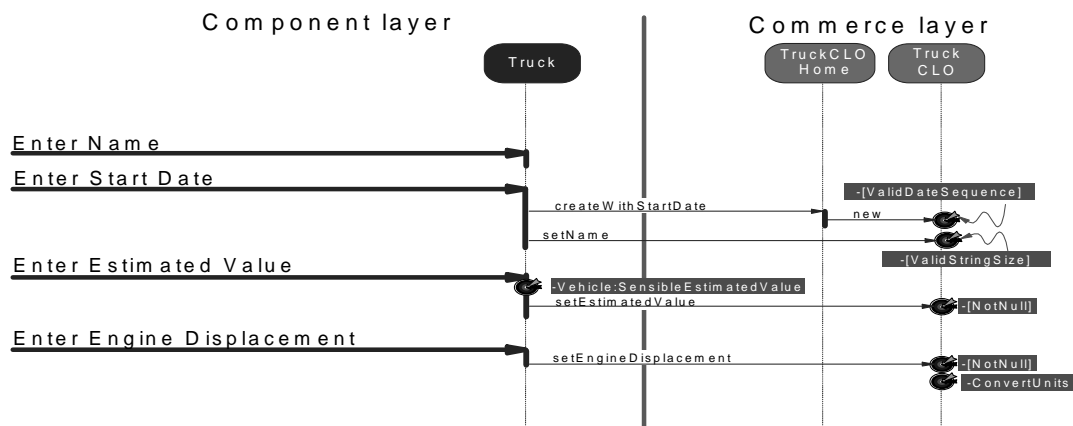


Figure 1. Object interaction diagram annotated with trigger points

2.3. Trigger Point Context Patterns

A trigger point is the specific place from which externalized rules will be *fired*. A trigger point's primary job is to select which rules will be fired, based on some criteria that we call *trigger point context*. In all cases, the rules to be selected can be changed over time using the administration system (discussed later). We identified several trigger point context patterns: they are directly supported by ABR (others can be hand coded). Four patterns are described below.

2.3.1. Fixed Business Context: A fixed business context represents a static business situation (i.e. it does not depend on run-time data). It is captured in ABR with either a simple "direct" name (e.g. ComputeRateQuote) or a compound name when the context is hierarchical (e.g. Vehicle::isEligible). The name usually provides information about the trigger point function and careful name-space management is necessary.

2.3.2. Situational Context: A situational context represents a dynamic business situation (e.g., in a "personalized web site", part of the content of a web page depends on which type of customer is currently accessing it). In situational context, the correct rules partly depend on business context computed at run time. ABR addresses such cases through the use of classification rules. Many business rules refer to (or apply to) the ways in which "things" are classified by the business (e.g., income level, trust, youthful drivers). Since the classification

process is itself rule based, we have in this case two levels of rules. A first phase occurs with the state of the application being assessed. The process of assessing application state is called classifying and is carried out by selecting and firing rules called classifier rules. The classifier rules selected for firing are those where the category context (a piece of the trigger point context) matches one or more desired classification categories (a grouping of classifiers which share a common purpose). After being found, classifier rules are fired to classify the current state of the system. If a classifier rule determines that the current state of the application is one that it recognizes, then the name(s) of the classification(s) is returned. If the application's state is not recognized by a classifier rule, nothing is returned. The relevant classifications are ordered by precedence and are used with the trigger point name to select the actual rules to fire. Depending on the type of trigger point, the rules can check some constraints or derive a value, such as a policy premium.

2.3.3. Jurisdiction: A special case of a situational/classification based pattern is what we refer to as jurisdictional pattern. This pattern adopts the view that rules are asserted by logical entities called jurisdictions that provide: 1) a way of telling whether the jurisdiction has authority over a given business object instance and 2) the rules it wishes to assert as well as the object class(es), and context(s) in which each applies.

The jurisdictions applicable to an application are collected together into a *jurisdiction group* which collaborates with the trigger point to select, order, and attach the appropriate rules. E.g., a U.S. insurance application could have rules issued by the state, by the federal government, and by the insurance company, each of which forms a jurisdiction. The collection of all of them is the jurisdiction group. Organizing rules into jurisdictions has two advantages. First, it makes it apparent in the rule model that most business rules have a specific source (owner) and that different sources might want to assert different things in the same business context. This provides a natural, organized way for business people to understand and manage the enterprise's business rules. Second, it provides a structured way to implement a common derivation pattern in which one jurisdiction specifically relies on another jurisdiction's computation result as input for its own computation (often, a more general context feeds its result to a more specific context). For example, in the computation of the minimum premium for a policy, the final premium computation could be done by the insurance company jurisdiction which sets the premium value to be the value imposed by state plus some computed additional dollar amount.

2.3.4. Structural Context: Generally the process of identifying and implementing the business rules involves transforming a high-level business requirement that may be vague and ambiguous into several lower level rules which apply to a precise context highly correlated to your object model. We refer to such contexts as structural contexts. They are usually method-based pre- or post-condition constraints, measurement unit conversions, UI edits, cardinality constraints, or assertions. Structural context names are usually formulated by concatenating the className with the methodName and a locationName (pre, post, body1, etc.) within the method. An example would be `Vehicle::setGrossWeight::pre`.

3. Framework Runtime Overview

ABR runtime is implemented as a normal Component Broker (CB) server application. CB provides the base distributed object infrastructure (persistence, query, remote invocation, ...).

3.1. Externalized Business Rule

An externalized business rule is implemented as a trio of objects: a RuleUse, a Rule and a RuleImplementor. The RuleUse, is the kind of object the trigger point needs to find. Each

RuleUse links a trigger point to the rest of the business rule implementation. It has a reference to a Rule and a number of attributes indicating the business context in which the rule applies as well as the time period during which the rule is valid. A RuleUseHome is a factory for creating RuleUses. The RuleUseHome also maintains a persistent collection of all existing RuleUses and provides a number of *find* methods that trigger points use to locate the RuleUses they require. Each RuleUseHome find method takes various sorts of trigger point context (information about the trigger point and its environment) as input, builds an SQL query from them, and returns zero, one, or a sequence (array) of RuleUses matching the context.

Each RuleUse has a reference to a Rule. The Rules, like the RuleUses, are Component Broker-based persistent objects. The Rule contains a persistent set of values and a reference to an executable algorithm. The algorithm, which can be coded in C++ or Java or both, is called a RuleImplementor. A RuleImplementor is a transient object (not Component Broker managed) which the Rule instantiates and then uses to do the actual work. When the *fire()* method is called on the Rule object, the Rule object combines its persistent set of values with the parameters it received on invocation to create the parameter list for the RuleImplementor, then it invokes *fire()* on the RuleImplementor with this parameter list. The actual execution of the RuleImplementor algorithm can take place either remotely (within the CB server) or locally (within the native environment).

3.2. Rule Driven Object

A *rule driven object* is an object with trigger points. The code in a trigger point interacts with the ABR runtime to locate the required rule(s), gathers up the information required as input, fires rules locally or remotely in the desired language regime, and deals with the results of rule firing. There are multiple ways trigger points have for doing each of these steps. Combinations of them give rise to trigger point patterns. The framework directly supports the most common ones.

Trigger points may be inserted into code by hand-coding, through the use of inheritance, or through code generation. A trigger point framework is supplied that makes hand-coding practical. A trigger point generator is also included with the framework. The generator can analyze source code and insert trigger points based on method names, parameter types, or other information that can be extracted from source. Alternatively, the business object developer can add structured comments to the object methods corresponding to the type of trigger point desired. The generator interprets these structured comments and injects trigger point code into the object method body. The relationship between the rule driven object and the objects supporting externalized rules is shown in Figure 2.

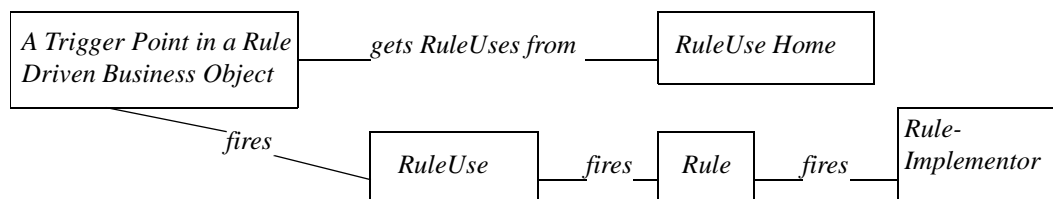


Figure 2. Relationship between a rule driven business object and the rule objects

3.3. Runtime Behavior

We discuss here a simple example of a trigger point selecting, executing, and then responding to the results of a business rule. The information a trigger point uses to select the required rules is called its *trigger point context*. There are several standard forms of trigger point context, but they all equate to some small set of data that is easily available to the trigger point. The first step is for the trigger point to assemble the context and then invoke a query method on the RuleUse-

Home. This will return zero or more RuleUses. If there is at least one RuleUse, the trigger point will assemble the data that will be sent as parameters (as an ordered sequence/array) to each RuleUse. The trigger point will then loop through the list of RuleUses invoking the *fire* method on each and passing the parameters. The results will be handled depending on the trigger point pattern being used.

In the simplest case, a RuleUse references (is bound directly to) a single Rule and that Rule references a RuleImplementor. When a trigger point invokes *fire* on a RuleUse, the RuleUse uses its reference (that is part of its persistent state) to delegate the request to the Rule it is bound to. The Rule in-turn instantiates the RuleImplementor and uses it to do the actual work (to execute the rule algorithm or test). Once it has arrived at a result, the RuleImplementor returns that result. For constraint rules (ones that arrive at a boolean true/false answer) the returned value is, by convention, a *ConstraintReturn*. A *ConstraintReturn* is a data structure indicating whether or not the constraint was satisfied, and if not, what went wrong. For derivation rules (ones that calculate a single generally non-boolean value) the return value may be of any type. In the simplest case, the return value from each RuleImplementor is returned back to the trigger point where it is analyzed to determine what action to take.

Here is an example viewed from within the framework: 1) A single RuleUse is associated with the trigger point context “Truck::setGrossWeight::pre”. This is one of the standard forms of trigger point context. It means the trigger point occurs just before the first instruction of the setGrossWeight method of every Truck object. 2) A Rule, named maxTruckGrossWeight, is bound to the above RuleUse. The Rule contains an initialization parameter list consisting of a single value of 42000 (meaning a maximum gross weight of 42,000 lbs.). This Rule is bound to a RuleImplementor class called MaxRuleImpl. 3) MaxRuleImpl, when invoked, tests the parameter it is passed against the initialization list value and returns a *ConstraintReturn*. The *ConstraintReturn* will be set to true if the passed parameter is less than or equal to the initialization value. Otherwise, a *ConstraintReturn* is set to false and some information is added describing which values were compared and why the test failed.

Here is what it looks like from within the trigger point: 1) During the execution of the application, the setGrossWeight method is invoked on a truck. For a pre-condition trigger point, the trigger point code is the first executable code encountered in the method. 2) The trigger point invokes findPreConditionRuleUses on the RuleUseHome passing truck::setGrossWeight::pre as the trigger point context. It receives back a sequence (array) containing a single RuleUse reference (maxTruckGrossWeight). 3) Using the reference, the trigger point invokes *fire* on the RuleUse passing in as a parameter the proposed grossWeight to be set. 4) The trigger point receives back the *ConstraintReturn* as the result of the fire. 5) If the *ConstraintReturn*'s result is true, the trigger point does nothing and execution continues into the body of the method. If the *ConstraintReturn*'s result is false, the trigger point adds the *ConstraintReturn* to a sequence/array and then throws an exception containing the sequence/array.

This sounds (and actually is) quite complex. However, the tooling, the patterns, and the framework simplify things to a manageable level. The developer need not be overwhelmed by the complexity.

3.4. Exception Handling

ABR defines two runtime exceptions: RuleProcessingError and RuleViolation. Both define a *tracebackInfo* field (array) to allow information to be added at every level the exception is caught and re-thrown. They are added to most methods in a rule-driven object system.

A RuleProcessingError exception is thrown when something goes wrong during rule processing. For example, a *derivation* trigger point conventionally expects to find, at most, one RuleUse

associated with it. If it finds more, it throws a `RuleProcessingError`. A `RuleProcessingError` contains a number of pieces of data that allow the receiver to determine what type of error happened, where, and under what circumstances. The code that catches the exception should either handle it completely (i.e., recover from the error it reports) or re-throw it after adding to its `tracebackInfo`.

A `RuleViolation` exception is thrown by trigger points firing constraint-type rules. As mentioned previously, constraint rules return a standard structure, called `ConstraintReturn`. When violations occur (as indicated by a result of *false* in the `ConstraintReturn`), the trigger point accumulates a list of failures. After firing the last `RuleUse` (for a pattern that collects all violations), the trigger point creates a `RuleViolation` exception (containing its identification and the list of the accumulated `ConstraintReturn` structures) and throws it. If no constraint violations occur as a result of firing the `RuleUses`, the trigger point simply finishes silently. As with the `RuleProcessingError` exception, the code that catches the `RuleViolation` should either handle it completely or re-throw it after adding to its `tracebackInfo`.

4. Rule Administration Overview

The ABR Framework comes with an administration application that lets you manipulate Rules and RuleUses using a web-based GUI. It is implemented as a Java applet that runs in a Java-enabled web browser or an Appletviewer. An administration system is important in making externalized rules manageable and easy to understand. In our first projects, the administration of rules was done through the use of word processors and scripts. Manual administration and deployment of rules is difficult and error-prone.

In addition to the runtime find methods, the `RuleUseHome` and `RuleHome` provide a number of methods oriented towards management and administration. These allow special administrative clients to search the collection in various interesting ways, and to create, update, and delete instances. Rules and RuleUses contain a number of attributes of interest to the administration client: for example, the RuleUses contain dates that control when they become effective or are withdrawn, and they have a *businessIntent* and *originalBusinessAnalystRule* attributes which help tracing back the Rule/RuleUse to the original business statement that was elicited at analysis time from a business person (this statement is often ambiguous and non-rigorous and may have to be broken down in several more specific ABR rules).

Broadly speaking, the goal of rule administration is to alter the behavior of a rule-driven application by changing the set of business rules that will be in effect at the point in time that the new behavior is desired. A person doing rule administration, when presented with a change in rule requirements for a business system, might go through the following steps:

1. Understand the change in business behavior that is desired.
2. Inspect the application documentation, chiefly object interaction diagrams annotated with trigger point location information, to understand where in the system the changes will need to be made;
3. Query the corresponding set of existing business rules with the Administration Application to understand which need to change;
4. Use the Administration Application to construct one or more new RuleUses that will bind the correct new behavior to the appropriate trigger point(s);
5. On a test system, withdraw (by setting `endDate`s) the Rule Use(s) that are to be superseded;
6. Test the application to ensure that it behaves as expected;
7. Schedule the new Rule Use(s) to become effective at the correct point in time on the production system; and finally,

8. Schedule the Rule Use(s) which are to be withdrawn so they expire at the point in time at which the new Rule Uses come into effect.

The Administration Application provides facilities to carry out steps 3 to 8. Only occasionally for a running system, will you need to create a RuleImplementor to supplement the ones supplied with ABR or already built for your application (see Section 4.1 for more details). Except for creating a new RuleImplementor, all of the manipulations within the Administration Application are done without programming.

4.1. Building New Rules Using Rule Templates

Our first “real life” trial at designing a rule-driven application resulted in having to build a surprisingly small number of base template rules (around thirty). The other rules (about five hundreds) were derived from the base rules through simple parameters setting, mapping or combination. ABR comes with a base extensible “rule starter set” that allows an administrator to easily create new Rules from existing Rules using composite Rules or cloning existing Rules and modifying the values of their parameters.

ABR includes a number of basic RuleImplementors (e.g., in-range-check, less_than,...). Rules can persistently store within them initialization parameters. These persistent parameters are combined with the parameters passed to them at runtime when they are fired. The combined parameters are then passed to the RuleImplementor, supplying both the runtime value(s) to check and the static value(s) to compare against (see Section 3.3).

ABR also includes a number of composite Rules. An *AND* Rule calls multiple Rules and “ands” their results. An *OR* performs a similar function while “or-ing” the results. An *ifThenElse* Rule fires a Rule, tests its result, and then selects between two other Rules. We also have a *mapping* Rule that allows one to select, reorder, and add to, the parameter list provided by the trigger point in order to match the parameters required by the RuleImplementor. This is especially useful when the trigger point code follows a generic pattern and the parameters being passed to the rules are generically created: e.g., pre/post conditions trigger points build a parameter list containing a reference to the object where the trigger is located and all the parameters it is passed (i.e., the method parameters).

5. Lessons Learned

As mentioned in the introduction, ABR found its basis in two earlier projects. Much of what we learned in the course of these two efforts became the basic requirements for the ABR effort. We continue to learn as ABR progresses. The following sections describe some of our findings.

5.1. Performance

One of the early decisions that we had to make was which rules to externalize and which rules to leave embedded in code. This is a cost/benefit question. The costs, we found, came down to a single factor; performance. In our first “real-life trial”, performance was so bad that we would not be able to deploy applications utilizing externalized rules. But through some rework, tuning, and the implementation of caching, performance became acceptable.

A trigger point retrieves the applicable rules by querying the rule repository. Performance profiling clearly correlated the performance bottleneck with the overhead introduced by querying the persistent store (in our case DB2) where the rules are stored. We developed a mechanism which enabled us to cache the results of the queries (sets of applicable rules), using the query as a key in the cache for future fast retrieval, and to automatically keep the cache current with selective invalidation of the cache as updates are made to the rule repository (see [6] for details).

Another step that we took to improve performance was to give the flexibility to execute the rules locally when desired. This is further discussed in the following section.

5.2. Where to Trigger and Execute Rules

In ABR, rules can be triggered on the client or server side. We also included the flexibility to execute the rule either locally (to the trigger point) or on the rule server.

Often rules that run on the client will be different than those run on the server, because different types of business function are being implemented. Many rules need to be in the client so rule failures can be more clearly linked to objects that a user understands. Conversely, the same rule might be run on the client and the server, but the acceptable values might be more constrained on the client and more general on the server (so the server can accommodate use by multiple clients). In some cases, the same rule and values may need to be run on both sides because of timing, scoping, or trust issues. For example, if security is an issue, you may want to place rules so they are difficult to bypass (in the server). On the other hand, you may want to be notified of a rule failure at the earliest possible point (in the client). You may also want to receive only one or a few error notifications at a time rather than a big batch (at very granular points in the client). Or, you may want to review a large number of constraints (client, server, or both depending on where all of the needed data comes together). The key item here is that the place where you first recognized the rule may not be the place where you wish to execute the rule at runtime.

5.3. Management of Rule Failures

Rule failures are thrown as exceptions. The content and handling of these exceptions is very important to a system where the user must correct something in order for the rules to fire successfully. ABR defines two types of exceptions: the runtime and the administration exceptions.

During general runtime, only one exception type is thrown from within the framework. One additional type may be thrown from the trigger point. We avoided to define a larger number of more specific exceptions because of the expectation of where exceptions will be caught. A `RuleProcessingError` can be thrown anywhere within the framework and must generally bubble up (through layers of delegation) to the trigger point or even higher in the call path before being caught. A `RuleViolation` is thrown by the trigger point and is often bubbled up all the way to just below the user interface before being translated to a user informational pop-up. If there were lots of exception types, the management of the throws clauses on methods would become quite difficult. With only two exceptions, we can follow the rule of thumb that says put both exceptions on any method you think might encounter the exception now or in the future.

ABR has a number of methods that were created for use by the administration system only rather than the general rule runtime system. These methods implement a more granular set of exceptions as the administration system takes a more traditional view of exception handling.

6. Comparison with Other Rules Systems

In a previous paper [7], we contrasted ABR with workflow rules systems. We will not repeat that here, but the essence is that workflow systems act on a higher level of business process than ABR. They are generally focused on controlling how work flows from one person to the next, who should get the work next, and how long work can remain in a queue before it should get reassigned. ABR is focused at factoring out the time- and situation-variable behavior in the business objects that automate business processes.

ABR can also be distinguished from rules systems such as R++ [8]. R++ is implemented as an R++ to C++ translator rather than a framework. R++ imbeds the rules directly within classes, rather than externalizing them as does ABR. R++ observes instance data as it changes and fires rules based on the changes. R++ has some restrictions on the locations of the data that it can observe (restricted to data within the class or data that can be gathered via following relationships between objects). An ABR `RuleImplementor` can use whatever data is sent to it (from the

trigger point) and whatever data can be acquired by following paths from the objects that it was sent. R++ implements inheritance and allows sub-rules to override super-rules. In ABR, trigger points can be inherited and that inheritance can change the trigger point context if you wish, which will then select a different set of rules for firing. We see ABR as an alternative to R++.

Section 6.1 and Section 6.2 compare ABR with respectively database and knowledge based approaches.

6.1. Database Triggers and Stored Procedures

Active databases and the concept of event-driven information systems have become popular as commercial databases such as Ingres, Sybase, Oracle or DB2 started to offer trigger mechanisms. Business rules in active databases are composed of three components: event, condition and action (ECA). A number of papers have been published describing work in this area [9][10].

Like ABR, database triggers and stored procedures offer the advantage of modularity: business rules are isolated from the rest of the application logic. However, triggers are harder to develop, maintain and/or extend. They are often expressed in SQL like languages that are proprietary and complex. More importantly, because they apply to the elements and values of the database, they tend to be more technically oriented than business oriented. Typical OO applications define different “object layers”-- each of which builds on what went before. The lowest level objects directly reflect the database data forming the relatively finely grained object model that experience shows is natural for fully capturing the required detail. As you move up the object model toward the user, the level of abstraction increases and business components are defined that are more easily understood by a business person; but their state data is only indirectly a reflection of data stored in the database. Business rules defined at this level (as allowed by ABR) are more naturally stated and can apply to procedural logic (i.e., methods on objects) as opposed to the triggers and stored procedures that are mostly limited to SQL operations, relationships and constraints on the data.

Some vendors such as USoft Corp[11] and Versata Software (previously Vision Software) [12] provide data-oriented business rules products that add modeling support, automatic generation, visual development, and database independence over conventional SQL coding and triggers. They implement their rules as a set of mid-tier services in a distributed application or database trigger. Their main disadvantage is that as they are still more focused on the data model rather than the business model. But given the number of database-oriented applications in existence today, the data-oriented approaches will have a long future.

6.2. Expert System and Artificial Intelligence (AI)

Rules have been used as a way to represent knowledge since the late 1960's in the AI community. In the 1970's, expert systems and logic programming emerged and became popular as a way to derive or confirm knowledge based on a complex set of rules. Rule-based systems typically include a set of rules, an inference engine, and a set of data (i.e., facts, which can be viewed as a special simple case of rules), along with perhaps other components (e.g., authoring or database interfaces). Rules are typically “if then” (perhaps “if then else”) statements, e.g., *if shopper purchases more than \$500 in last 6 months then offer discount of 20% on Specials*. Two common and important kinds of inference engine control strategies for processing these rules are: 1) forward chaining -- drawing new conclusions from existing data; and 2) backward chaining -- answering queries. More recently, a number of companies have developed knowledge-based rules systems that are object-oriented, such as Computer Associates' Aion [13] and Blaze Software (formerly called Neuron Data), Inc.'s Advisor Suite [14].

Aion (as an example of a knowledge based system) is a full environment, where ABR is a framework. Aion utilizes an “inference engine” which examines data as it changes and fires

rules based on those changes. ABR fires rules as trigger points are encountered during the normal processing of methods. Aion's rules are expected to execute within its environment. If you wish to couple Aion with existing objects, you will generally call to Aion through an API. Aion will then gather together the information that it needs to analyze, and will then trigger a large number of often complex rules.

ABR is targeted at firing generally one or a few rules at any trigger point. The amount of data examined is usually small. The performance overhead is minimized. The parameters passed to ABR rules rarely need any transformation. We see ABR positioned for execution of simple rules where there is benefit of externalization and the ability to do rule maintenance (modify which rules fire) without doing coding. ABR does not displace knowledge-based systems. Knowledge-based systems are appropriate when the rules become complex or when complex data relationships need to be analyzed. In fact, an ABR derivation rule would be a very good way of triggering a knowledge-base system. See [15] for a review of knowledge-based rule systems.

7. Conclusion and Future Work

The ABR framework defines a structuring mechanism for business applications and enables enterprises to develop distributed business applications that systematically externalize the *time-and situation-variable* parts of their business logic as externally applied entities called business rules. ABR is targeted at firing small sets of rules as you proceed through the normal processing of an application. This seems to be beneficial and differentiates it from other rule systems. Given the rise of interest in distributed object systems, it is hoped that ABR has a place in the future.

We are currently working on a number of extensions such as the incorporation of the externalization of rules within analysis and design (e.g., augmenting UML [16] to include trigger points and coupling with analysis and design tools), the exploration of natural language/data mining search and generation techniques to enrich our query and description capabilities, and the study of the trigger points patterns and rule templates specific to a given industry and/or application (e.g., personalization for eCommerce).

[1] IBM, Component Broker, information available at <http://www.software.ibm.com/ad/cb/>

[2] I. Rouvellou, L. Degenaro, D. Ehnebuske, B. McKee, K. Rasmus: "Externalizing Business Rules from Enterprise Applications: An Experience Report" in the Oopsla'99 Companion

[3] "Defining Business Rules - What are they really?", the final report of the "GUIDE Business Rules Project", not currently available, see further information at <http://www.guide.org/>

[4] EBG Consulting, Inc, information available at <http://www.ebgconsulting.com>

[5] Knowledge Partner's Inc, information available at <http://www.kpiusa.com>

[6] L. Degenaro, A. Iyengar, I. Lipkind, I. Rouvellou: "A Middleware System which Intelligently Caches Query Results" in the Proceedings of Middleware 2000.

[7] I. Rouvellou, I. Simmonds, D. Ehnebuske, B. McKee, K. Rasmus: "Business Objects and Business Rules" in "Business Object Design and Implementation: Oopsla '96, Oopsla '97, and Oopsla '98 Workshop Proceedings" Springer, ISBN: 1-85233-108-9.

[8] J. Crawford, D. Dvorak, D. Litman, A. Mishra, and P. Patel-Schneider: "Path-Based Rules in Object-Oriented Programming" in the "Proceedings of the Thirteenth National Conference on Artificial Intelligence (AAAI-96)", August 1996

[9] S. Chakravarthy, V. Krishnaprasad, E. Anwar, S. Kim: "Composite Events for Active Databases: Semantic Context and detection", in the Proceeding of the VLDB 1994, pp. 606-617

[10] H. Herbst: "Business Rules in System Analysis: A Meta-Model and Repository System" in Information System, 21(2), p147-166.

[11] "Business Rules Automation: A Proven Approach and Software Technology to Implement Computing Applications" (white paper), available at <http://www.usoft.com/whitepapers/>

[12] Versata, information available at <http://www.versata.com/default.htm>

[13] Computer Associates' Aion, information available at http://www.cai.com/products/platinum/appdev/aion_ps.htm

[14] Blaze Software, information available at <http://www.blazesoft.com/>

[15] Benjamin N. Groszof: "Business Rules for Electronic Commerce: Interoperability and Conflict Handling", available at <http://www.research.ibm.com/rules/>

[16] "Unified Modeling Language v1.1", Rational Software Corporation, available at <http://www.rational.com/uml/>