Self Evaluating Rules

DAVID M. WEST
University of St. Thomas
dmwest@stthomas.edu

KEVIN JOHNSON
University of St. Thomas
kmj@acm.org

Abstract: Every organization establishes a myriad of rules that define, associate, constrain, and formulate. Every information system must accommodate the support and maintenance of those rules. This paper proposes a means to treat rules as independent objects capable of self resolution. Benefits from this approach include the ability to provide other objects with the rules they require to complete their assigned tasks, to modify the behavior of objects at run time by replacing the rules they use, and to modify the rules (and hence the nature of the object returned as a result of evaluation) at run time as well.

"A Dependent is eligible for Family Coverage if she or he is: under eighteen years of age; over eighteen but a Full-time Student and receiving a majority of their support from the Insured, but not more than twenty-four years of age; or if he or she is Dependent due to Disability or Illness."

"Your interest rate is the average prime rate (as published in the Wall Street Journal) for the thirty days preceeding the issue date of your statement plus 4.5 percent."

An Employee may have 0 or 1 Spouse

$$Py = (1/360 * FA) + (1/12 * (.075 * CB)$$

The preceding are examples of business rules. Businesses are rife with these kinds of statements, formulas, conditionals and mandated relationships.

A major task in software analysis involves the identification, modeling and implementation of business rules. This is also true of object-oriented analysis.

Despite the ubiquity and importance of business rules, they seldom merit detailed discussion in texts dealing with object analysis methods. The term does not merit inclusion in any of the major object method book indices. Most treatments of business rules in object methods are found in the discussion of static class relationships. (Traditional methods cover the topic in the same way, using entity

relationships in a data model.) These static relationships are then supposed to be established and enforced by appropriate code in object methods or by establishing various objects specializing in the maintenance of relationships. The most ubiquitous of these objects being a database management system (either relational or object).

A prime dictum of object orientation is the notion that, "everything is an object." Business rules should, if that is the case, be objects in their own right. Numerous suggestions have been made on how to best implement rules in an object context (see bibliography) but most of them still treat the rules themselves as passive entities manipulated by rule engines or their equivalent.

The approach we outline in this paper treats rules as independent behavioral objects. They have the ability to obtain any necessary information they require and to perform any computations required in order to evaluate themselves and return an expected value (as an object of course) that represents the resolution of the rule.

Structural Abstraction of a SelfEvaluatingRule

A useful heuristic for performing object decomposition is to take an example and separate its physical parts. We can do this with a simple example of a rule expressed as a formula:

$$\mathbf{X = 4q + (p*r)}$$

Analysis of this equation yields five distinct types of physical components:

X, q, p and **r** represent things which are currently unknown but knowable.
Following convention, we will call theses items **variables.**
4 represents a **constant** value.
+ and ***** are behavioral **operators.**
(and **)** in combination represent an aggregation and precendence **operator.**
= is an **operator** of symmetry - indicating that the thing on one side of the equation is in some sense symmetrical with, equal to, or to be assigned to the thing on the other side.

The order in which these components appear is important, as is the associate of the components with their immediate neighbors. Our initial, structural, abstraction of a rule then follows:

A rule is an ordered collection of variables, constants and operators.

Each of these components can be analyzed to discover its behavioral characteristics. Constants are perhaps the simplest, most often being instances of known classes like Integers, Float, Reals, Character, String, *et cetera*, although nothing prevents them from being any known object with a fixed or constant value (state).

Variables are more interesting. At first glance we seem to have two different types: those, like X, that equate (are assigned) to the resolution of the entire rule; and those, like **p**, **q** and **r**, that represent a discrete value which can be used to resolve the rule.

If we think of variables as a place where a value (an object) could be but currently is not, we can posit a structural abstraction for a variable. In the case of variables like **X**, that abstraction would consist of a "targetObject" and a "setterMessage"[1]. For the q, p and r variables, the abstraction is a "sourceObject" and a "getterMessage". Behaviorally, a variable is responsible for obtaining its value, for instantiating itself. It does so by sending the getterMessage to the indicated sourceObject.

In the realm of objects we are accustomed to treat operators as messages. In our simple example we see three nuances of operator: first, those representing standard messages sent to objects to invoke behavior (+ and * being examples); second, those that instruct the Compiler (an object not available at run time) to establish precedence (the parend operators); and three, assignment operators (the colon-equals).

For simplicity we will exclude assignment operators from consideration. We are giving the client of the rule the responsibility to make the assignment using the result provided by the rule (everything to the right of the assignment operator) whose responsibility is to determine that result.

Because the Compiler is not always available at run time, it is necessary to find an alternative means to "parse" and execute a rule with complex structure. This is accomplished by defining a new type of object, a Term. Following the lead of Smalltalk, we initially define three types of terms (reflecting the three types of messages): Unary, Binary, and Keyword Terms. A UnaryTerm provides a structure for associating an object (receiver) with an operator (message); a BinaryTerm extends this structure to include an argument; and a KeyWordTerm adds structure for a collection of arguments.

Constants and Variables are also defined as Terms and our Rule becomes a hierarchy of nested Terms. We can re-write our example rule as a hierarchy of terms and we can show the structure of our variables using the abbreviations: **sO** for sourceObject and **gM** for getterMessage.

---

[1] Assignment variables will not be a part of our ultimate abstraction nor will the assignment operator be included as part of a rule's structure. The rationale for this action is noted in the discussion of operators.

Two final structural points. First, it should be noted that the way we have defined variables allows for each part of the variable's structure to be itself a variable. We could, for example, replace any of the sO or gM notations in the preceding example with another, nested, variable. This would be useful in instances where either the source (whichever Window currently has the focus, perhaps) or the message to be sent would best be determined at run time.

Second, and a bit less obvious, we can replace any variable (or portion of the variable's structure) with an arbitrarily complex rule of the type being defined. We are not advocating this, too much use of this feature could result in writing entire programs as a complex rule - not necessarily a good idea.

Our attention can now turn to behavior and the dynamics of the Rule and Term structures we have defined.

Behavioral Abstraction

Beginning with the SelfEvaluatingRule object, what kind of behavior might we expect? To begin with, because it uses an ordered collection to store its Terms, we would expect it to be able to modify itself by adding and deleting Terms. This means it would have the ability to respond to messages like:

at: anInteger **put:** aTerm

at: anInteger

delete: anInteger

add: aTerm **at:** anInteger

We would not expect it to reorder, sort or otherwise rearrange itself.[2]

Our rules might involve the use of arguments and must be aware of the first, or root, term in the hierarchy of terms. To accommodate these needs we provide a rule with two instance variables (arguments and rootTerm) and the necessary protocol to set and get the objects occupying these variable locations.

The primary purpose of a rule is to evaluate itself. For these reason its protocol must include the message, **evaluate**[3]. Because the evaluation may depend on the presence of arguments and these

[2] Although, if we were really ambitious, and if our rules conformed to the expectations of mathematical expressions, we might expect them to perform transformations (by factoring for example) into alternative but equivalent forms for the purpose of optimizing their resolution and hence operational performance.

[3] Smalltalk code for most of the methods we are proposing is provided at the end of the paper.

arguments might be provided at the time the evaluate request is made we also need to add the message, **evaluateWithArguments: aCollection**. And a final behavioral requirement, arising from the fact that a Rule may be used as a Term within another Rule requires it to respond to the **resolve** message. This message simply causes the rule to **evaluate** itself.

All Terms are responsible for **resolve** 'ing themselves. Additionally they have responsibility for returning and setting their variables. (Unary terms have instance variables for a receiver and an operator; BinaryTerms, for receiver, operator, and argument; KeywordTerms, for receiver, operator, and arguments.)

In order for a Variable to resolve itself it must obtain a value from from a source. This is accomplished by giving it the behavior to **instantiate** itself.

All basic behavior is now available to our objects. A rule is asked to **evaluate** itself. It in turn asks its root term to **resolve** itself. If it is complex (contains terms) each of them is asked to **resolve** itself. This proceeds recursively until all variables have **instantiate**'d themselves and all constants have returned their value. Each resolution is then passed back up the recursion tree until the rule itself is able to return a single object representing the result of its self-evaluation.

Extending the SelfEvaluatingRule

Some additional structure and behavior we might want to provide our rules would include:

1. Using an instance variable to store a unique identifier.
2. Using an instance variable to store a colloquial name.
3. Using an instance variable to contain an operating scope. (An object that would provide information about when and under what circumstances it is appropriate for this rule to function.
4. Using an instance variable containing a list of valid users of this Rule.
5. Using an instance variable to give each Term a list of valid Operators.

Adding the third and fourth new structural items would enable us to create a collection of rules and allow objects needing to use a rule to seek one from the collection. (By asking the collection if it contains any rules which, by virtue of their scope and valid user information, deem themselves appropriate for the requesting object to use.)

Of course rules can be provided to any object having an instance variable containing "myRules" or its equivalent. The choice of centralizing or distributing the storage of, and access to, rules is a design

option.

Adding the fifth structural item would allow a Rule to validate its structure as that structure is being created. This also provides a way of distinguishing rule "types" from each other on the basis of the kinds of operators allowed in their Terms. For example, if we wanted to create a ProductionRule as an alternative to the kind of formula or equation type of rule we discussed in the preceding paragraphs. We could do so by extending its set of valid operators to include the Booleans (IF, AND, OR, NOR, CONCLUDE, *et cetera*). This ability avoids the need for extensive sub-classing of SelfEvaluatingRule. (There are circumstances in which sub-classing may be an appropriate and preferred alternative design. Nothing in this definition of rules prevents making and implementing such a decision.)

Benefits

A direct representation of any business rule as an instance of the SelfEvaluatingRule class.

Rules join blocks and methods as a general purpose mechanism to implement an algorithm, with the additional advantage - for rules - of not requiring access to a compiler-interpreter in order to be executed.

The ability to encapsulate the rules that constrain behavior in the same objects expected to exhibit that behavior.

The ability to simplify other aspects of the class hierarchy by replacing a sub-hierarchy with a single class whose instances are "typed" or "sub-classed" based on the rules they contain instead of their structure (both instance variables and methods).

Providing a new way to explore the modeling of complex knowledge systems (expert systems, for example, without the need to build passive rules and complex inference engines) and adaptive agent-based software systems.

Providing a foundation for the direct construction of rules by non-software experts (using an editor that is still under development) and the assignment of those rules to objects. The assignment of rules can be accomplished at run time.

The ability to construct complex rules using a directly accessible recursion hierarchy tree instead of layers of Meta-rules.

Implementation

The following hierarchy was created to implement the ideas in this paper.

Object

SelfEvaluatingRule

Term

Unary Term

BinaryTerm

KeywordTerm

Constant

Variable

Important particulars of each class are noted below. Instance variables are noted to the right of the class name and getters and setters for these instance variables are assumed and not noted.

SelfEvaluatingRule (arguments rootTerm)

evaluate

^rootTerm resolve

evaluateWithArguments

self arguments: aCollection

^self evaluate

resolve

^self evaluate *[Note: enables a SelfEvaluatingRule to be used as a Term.]*

Term *[Note: abstract class that defines protocol for sub-classes.]*

UnaryTerm (receiver operator)

resolve

^receiver perform: operator

BinaryTerm (receiver operator argument)

resolve

^receiver resolve perform: operator with: argument resolve

KeyWordTerm (receiver operator arguments)

resolve

receiver resolve

perform: operator withArguments:

[self arguments collect: [:each | each resolve]].

Variable (source getterMessage)

resolve

^self instantiate

instantiate

^source instantiate perform: getterMessage instantiate.

Constant (value)

resolve

^self value