RulesEngine

7 January 2009

Martin Fowler

API DESIGN
DOMAIN SPECIFIC LANGUAGE

Should I use a Rules Engine?

A rules engine is all about providing an alternative computational model. Instead of the usual imperative model, which consists of commands in sequence with conditionals and loops, a rules engine is based on a Production Rule System. This is a set of production rules, each of which has a condition and an action - simplistically you can think of it as a bunch of if-then statements.

The subtlety is that rules can be written in any order, the engine decides when to evaluate them using whatever order makes sense for it. A good way of thinking of it is that the system runs through all the rules, picks the ones for which the condition is true, and then evaluates the corresponding actions. The nice thing about this is that many problems naturally fit this model:

```
if car.owner.hasCellPhone then premium += 100;
if car.model.theftRating > 4 then premium += 200;
if car.owner.livesInDodgyArea && car.model.theftRating > 2
   then premium += 300;
```

A rules engine is a tool that makes it easier to program using this computational model. It may be a complete development environment, or a

framework that can work with a traditional platform. Most of what I've seen in recent years are tools that are designed to fit in with an existing platform. At one time there was the notion of building an entire system using a tool like this, but now people (wisely) tend to use rule engines just for the sections of a system. The production rule computational model is best suited for only a subset of computational problems, so rules engines are better embedded into larger systems.

You can build a simple rules engine yourself. All you need is to create a bunch of objects with conditions and actions, store them in a collection, and run through them to evaluate the conditions and execute the actions. But mostly when people refer to "rules engine" they mean a product built specifically to help you build and run a rules engine. Techniques to specify rules can vary from an API for people to describe rules as Java objects, a DSL to express rules, or a GUI that allows people enter rules. More efficient execution engines help to quickly evaluate conditions on hundreds of rules using specialized algorithms (such as the Rete algorithm).

An important property of rule engines is **chaining** - where the action part of one rule changes the state of the system in such a way that it alters the value of the condition part of other rules. Chaining sounds appealing, since it supports more complex behaviors, but can easily end up being very hard to reason about and debug.

I've run into a few cases where people have made use of rules engine products, and each time things don't seem to have worked out well (disclaimer: I'm not a statistically valid sample). Often the central pitch for a rules engine is that it will allow the business people to specify the rules themselves, so they can build the rules without involving programmers. As so often, this can sound plausible but rarely works out in practice.

Even so, there's still value in a BusinessReadableDSL, and indeed this is an area where I do see value in this computational model. But here too lie dragons. The biggest one is that while it can make sense to cast your eyes down a list of rules and see that each one makes sense, the interaction of rules can often be quite complex - particularly with chaining. So I often hear that it was easy to

set up a rules system, but very hard to maintain it because nobody can understand this implicit program flow. This is the dark side of leaving the imperative computational model. For all the faults of imperative code, it's relatively easy to understand how it works. With a production rule system, it seems easy to get to a point where a simple change in one place causes lots unintended consequences, which rarely work out well.

I haven't spent enough time with these systems to get a sense of what heuristics we should follow to keep this implicit behavior under control.

- It does seem that it's important to limit the number of rules, indeed any system with enough rules to need sophisticated algorithms to get good performance probably has too many rules to be understood.
- Be very careful how you use chaining, often its best to organize your rules to limit or even eliminate chaining
- As in many places, testing is often undervalued here, but implicit behavior makes testing more important - and it needs to be done with production data.
- While building a rules system, I'd look to do things that would cause EarlyPain with modifications of the rule base.

All of these lead me to think that there's a lot to be said for avoiding rules engine products. The basic idea of production rules is very simple. In order to keep the implicit behavior under control you also need to limit the number of rules by keeping the rules within a narrow context. This would argue for a more domain specific approach to rules, where a team builds a limited rules engine that's only designed to work within that narrow context. Certainly if you're thinking of using a rules engine I'd suggest prototyping with both a product and a hand-rolled domain specific approach so you can get a good feel for how they would compare.

For more information on building your own simple rules engine, including a couple of toy examples, see the Production Rules System chapter of my DSL book.