How to deal with exceptions in software support

Now that our exceptional cases are described by ‘just some more rules’ as described in my previous column, it is easy to see that they can be supported by software systems. In this column we take a closer look at what options a software developer has to program such rules for automated software support.

On the use of else

The ‘if … then … else if ….‘ construct is widely used in all programming languages and can be used to handle a set of rules that describe different outcomes for related situations. When these constructs are nested it becomes difficult to understand the resulting run-time logic and it may be better to use a case statement.

Although the use of the *else* construct is acceptable in procedural code it is not needed in most production rule engines.

On the use of case statement

The ‘case’ construct is another popular programming construct that can be used for conditional branching. It chooses from a sequence of conditions, and executes one corresponding statement. The CASE statement evaluates a single expression and compares it against several potential values, or evaluates multiple Boolean expressions and chooses the first one that is TRUE.

The difference between using a case statement and using multiple ‘if … then … else if ….‘ constructs is that only one of the *when* statements is executed. The order of the conditions results in an implicit conflict resolution strategy.

On the use of rules and priorities

Many rule engines support the use of priorities as a conflict resolution strategy. When conflicts between rules are solved (as suggested in the previous column) and the structure of the business rules is taken as the basis for implementation, there is no need to use priorities. The results of the rule engine should be independent of the order in which the rule engine evaluates a set of consistent rules. The only reason that is acceptable to use rule priorities is to optimize the performance of the evaluation. Some rules may be ‘more expensive’ to evaluate than other rules. By giving the less expensive rules a higher priority one may optimize the performance of the system. Note that this is only useful if the rule engine stops right after reaching a conclusion.

Some rule engines support multiple conflict resolution strategies. The Drools engine supports many strategies, including strategies for *Lex Specialis* (as discussed in the previous column). However, the same reasoning applies to rules in inference engines as to business rules: it is better to solve a conflict because it makes knowledge explicit, improved understanding of the domain and it is easier to maintain in the long run.

On the use of default logic

The idea that everything is allowed unless prohibited by a rule, which is intuitive for most people, is not intuitive for a rule engine. It must be explicitly programmed; some program code that is processed after the rule engine has finished processing all the rules makes this default behavior explicit. The default behavior can be programmed with procedural code or with another rule. After all rules are processed there should be some logic that checks if some variable has been assigned a value by the rules. If the rules did not assign a value for the variable, there may be procedural logic that assigns the default value.

For our example of checking if someone may borrow a book from the library, we would first run the specific rules that authorize a person to borrow a book. Then we would have some logic that states that if it is still unknown if a person is authorized to borrow a book, he should not be allowed to borrow the book.

On the use of decision tables

If you have a rule engine supporting decision tables, you can implement logic that is described in a decision table very easily. Execution environments require transforming the design of the table such that all conclusions are in the last row(s) (or last column(s)) of the table. For the example introduced in the previous instalment this would result in the following table.

relation kind	elderly	elderly	student	student	child	child	none	none
family member of a family card holder	yes	no	yes	no	yes	no	yes	no
price	15	15	0	0	5	20	20	20

On the use of an inheritance hierarchy

For those of you who work in an object-oriented environment you may want to consider using the object hierarchy to deal with exceptions. For example, for the rules shown in decision table form above we would have:

```
Person.GetLibraryCardPrice                                  (return 20)
   Person.Adolescent.Student.GetLibraryCardPrice            (return 0)
   Person.Elderly.GetLibraryCardPrice                       (return 15)
   Person.Child.GetLibraryCardPrice                         (if this.is_member_of_a_famillycardholder then return 5 else return 20)
```

This is a very neat solution but imposes constraints on the way your hierarchy is constructed.