Torry Harris

Torry Harris Business Solutions

Rule Engines save the day

Author - Manjunath Hanasi

www.thbs.com

The problem

- Have you developed/seen code with so many nested if statements that actually looked like a nest?
- Have you spent sleepless nights debugging code when you modified one of those 'if' statements and found that it distorted the next if statement?
- Ever wondered how others implement these?

Then read on; this article is exactly for you.

If our logic requires triggering different actions based on many conditions, then implementing such logic in the code becomes a hinderance. Understanding the code becomes even more difficult for people maintaining such code.

In addition, if the conditions are changing frequently, then incorporating these quick changes within the stiff deadlines becomes nearly impossible. We would ideally like to keep these changing conditions out of our code. Another reason for this separation is that Business people know the rules well and it would be better if they were allowed to write the rules rather than the developers. Rule engines allow us to do exactly the same.

What are rule engines and how do they solve my problem?

Rule engines are specialized applications that read a set of rules from external sources (known as rule database or knowledge base); identify the correct rule(s) based on input conditions and execute the corresponding action(s). Rule engines are a great way to collect complex decision-making logic and work with data sets too large for humans to effectively use.

Rule engines separate the source code from the rule database. Hence, the code does not have to change if there are any changes in the rules. The code looks much neater, simpler to understand and easier to maintain. Business people own the rules while developers own the code.

Case Study

After delving sufficiently with theory, lets look at some practical usage of rule engines and how they solve our problems.

For one of our telecom clients, we had to implement a component that determines whether a customer can register with the telecom operator. The decision was based on several factors and was expected to alter frequently.

Just to give you an idea, the decision was based on the following input conditions:

1. The front-end channel the customer is using, i.e. web-based or through CSRs
2. Customer's scheme, i.e. Monthly Rental (MR) or Pre-paid (PP)
3. Customer status, i.e. New or Existing
4. Response from External Credit History checking authorities (NCHC and ECHC)
5. Response from systems checking customer details against fraud profiles (PC)
6. Response from Address Verification systems against blacklisted addresses etc (AV)

To complicate the matters, some conditions could be ignored if some of the pre-conditions are met. E.g. If the response from External Credit check entities is to “DECLINE”, then customer must be declined irrespective of responses from other systems.

Lets see how we can implement the above logic in the code itself.

```
// request is object containing input data
if (request.getChannel().equals(“WEBSHOP”))
{
  if (request.contractType().equals(“PAYM”)
  {
    if (request.custStatus().equals(“NEW”)
    {
      if (request.getCreditCheckResp(“ACC”) &&
         request.getProfileCheckResp(“ACC”) &&
         request.getAddressCheckResp(“ACC”))
      {
        response.setDecision(“ACCEPT”);
      }
      else if (request.getCreditCheckResp(“DEC”))
      {
        response.setDecision(“DECLINE”);
      }
      else if (……)
      {
      }
    }
    else if (……)
    {
    }
  }
  else if (……)
  {
  }
}
else if (……)
{
}
```

Clearly the above code is unrealistic, but it shows the complexity.

The next thing we might think is, we will store all the possible values in a database table like the following:

Channel	Contract type	Cust status	NCC	FPC	FEC	HAC	Final Decision
WEBSHOP	PAYM	NEW	ACC	ACC		ACC	ACCEPT
WEBSHOP	PAYM	NEW	ACC	DEC		ACC	REFER
WEBSHOP	PAYM	NEW	DEC				DECLINE

The above might look like a perfect solution as we can get the final decision in one wonderful SQL query. But there is a problem.

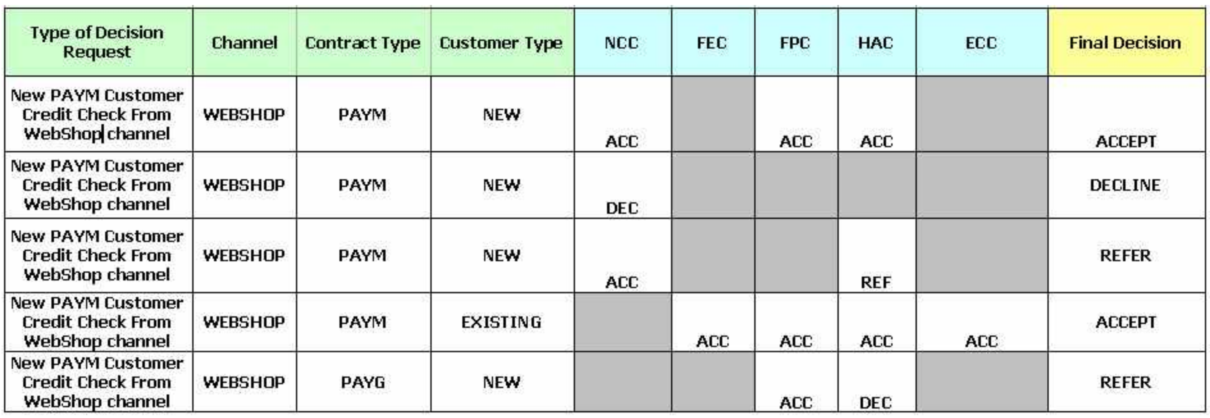
Consider that the input data contains the following responses.

- DEC from NCC
- ACC from FPC
- ACC from HAC

The SQL query will not fetch any results with the above data in the tables. For the query to work, we will have to add all the combinations of rules even if we know that “DEC” from NCC system is sufficient to arrive at a final decision (As discussed previously, if response from Credit Check is 'DEC', the customer should be declined irrespective of responses from other systems). Though the solution is better, adding all the useless data into the above rule table seems pointless. Is there some way we can avoid this?

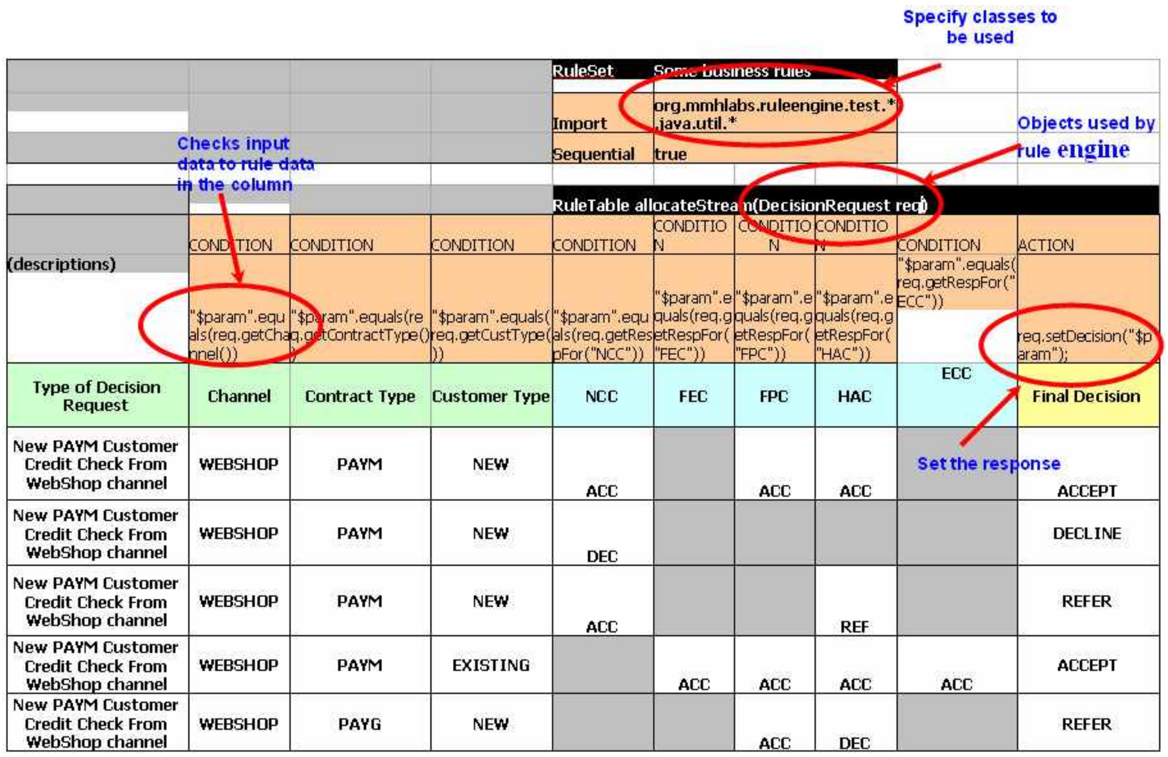
The answer is yes; rule engine comes to our help. There are many open source Java rule engines. We have chosen 'Drools' as our rule engine because it supports rule specification in an excel sheet.

The following is the rule specification spreadsheet:



Type of Decision Request	Channel	Contract Type	Customer Type	NCC	FEC	FPC	HAC	ECC	Final Decision
New PAYM Customer Credit Check From WebShop channel	WEBSHOP	PAYM	NEW	ACC		ACC	ACC		ACCEPT
New PAYM Customer Credit Check From WebShop channel	WEBSHOP	PAYM	NEW	DEC					DECLINE
New PAYM Customer Credit Check From WebShop channel	WEBSHOP	PAYM	NEW	ACC			REF		REFER
New PAYM Customer Credit Check From WebShop channel	WEBSHOP	PAYM	EXISTING		ACC	ACC	ACC	ACC	ACCEPT
New PAYM Customer Credit Check From WebShop channel	WEBSHOP	PAYG	NEW			ACC	DEC		REFER

As can be seen, Business people could easily add the rules to this spreadsheet document. This excel sheet will contain additional information that is relevant to developer and is hidden to the business user.


Specify classes to be used
Objects used by rule engine
Checks input data to rule data in the column
Set the response

				RuleSet	Some business rules				
				Import	org.mmhlabs.ruleengine.test.*,java.util.*				
				Sequential	true				
				RuleTable allocateStream(DecisionRequest req)					
	CONDITION	CONDITION	CONDITION	CONDITION	CONDITION	CONDITION	CONDITION	CONDITION	ACTION
(descriptions)	"$param".equals(req.getChannel())	"$param".equals(req.getContractType())	"$param".equals(req.getCustType())	"$param".equals(req.getRespFor("NCC"))	"$param".equals(req.getRespFor("FEC"))	"$param".equals(req.getRespFor("FPC"))	"$param".equals(req.getRespFor("HAC"))	"$param".equals(req.getRespFor("ECC"))	req.setDecision("$param");
Type of Decision Request	Channel	Contract Type	Customer Type	NCC	FEC	FPC	HAC	ECC	Final Decision
New PAYM Customer Credit Check From WebShop channel	WEBSHOP	PAYM	NEW	ACC		ACC	ACC		ACCEPT
New PAYM Customer Credit Check From WebShop channel	WEBSHOP	PAYM	NEW	DEC					DECLINE
New PAYM Customer Credit Check From WebShop channel	WEBSHOP	PAYM	NEW	ACC			REF		REFER
New PAYM Customer Credit Check From WebShop channel	WEBSHOP	PAYM	EXISTING		ACC	ACC	ACC	ACC	ACCEPT
New PAYM Customer Credit Check From WebShop channel	WEBSHOP	PAYG	NEW			ACC	DEC		REFER

Note: When Drools reads this excel sheet, '“$param”' is replaced by the value in the column

The code using the rule engine APIs is as follows:

```
private RuleBase _ruleBase;

private void loadRuleBase()
        throws SAXException, IOException,
        IntegrationException {

        InputStream stream =
                this.getClass().
                getResourceAsStream("RPD-Fraud-Decision-Table.xls");

        _ruleBase = DecisionTableLoader.loadFromInputStream(stream);
}

private void executeRules(DecisionRequest req)
               throws SAXException, IOException,
               IntegrationException, FactException {

        WorkingMemory engine = _ruleBase.newWorkingMemory();

       engine.assertObject(req);

       engine.fireAllRules();
}

public static void main(String[] args) throws Exception {

       loadRuleBase();

       // Populate DecisionRequest req object containing input data
       executeRules(req);

       // Print the decision stored in the request object by Rule engine
       System.out.println("Final decision: " + req.getDecision());
}
```

Given the above excel sheet, Drools reads the rules and executes them according to input conditions (data). The above solution is much neater, simple to understand and implement. It clearly separates the roles of Business analyst and developer, hence reduces considerable amount of time in development and testing of the applications.

Pros and Cons of Drools

Advantages:

- Drools is completely open-source and freely downloadable from the Internet. It uses very liberal ASL/BSD/MIT-esque license.
- Drools conforms to JSR-94 (Rule Engine API) specification.
- Drools rule APIs are easier to use.
- Drools provides very good documentation.
- Drools rule specification language is fairly easy and flexible compared to other rule engines.
- Drools supports rule specification in Excel Spreadsheets.
- Drools's performance is excellent.
- Commercial support available.
- Supports multiple semantic modules Java, Python, Groovy and C#.

Disadvantages:

-
- Dependencies on lot of JAR files.
- Initial load time for rulebase is high due to on-the-fly compilation of rule database (Janino embedded Java compiler).

Glossary

Term	Definition
MR	Monthly Rental Also known as Pay Monthly/Post-paid customers
PP	Pre-paid customers - Also known as Pay As You Go
NCHC	Credit History Check (verify the credit of new customer)
ECHC	Credit History Check (verify the credit of existing customer)
PC	Profile Check for customers
EC	Eligibility Check for customers
AC	Address Check for customers
CSR	Customer Service Representatives

Article sources:

- Why Use Rules
  http://drools.codehaus.org/Why+Use+Rules

- Legacy Drools site
  http://drools.codehaus.org/

- Drools site (v3.0)
  http://labs.jboss.com/portal/jbossrules

- Drools Decision Tables
  http://drools.codehaus.org/Decision+Tables

- Java Rule Engine API
  http://www.jcp.org/en/jsr/detail?id=94

- Some Guidelines For Deciding Whether To Use A Rules Engine
  http://www.jessrules.com/jess/guidelines.shtml