If you thought that things like Artificial Intelligence or logical programming are all dead and buried in the 80s and have no relevance to our enterprise projects, think again. Drools is a Java framework that implements a form of AI called rule-based Expert System that it might not win you Jeopardy ([http://en.wikipedia.org/wiki/Watson\_%28computer%29http://en.wikipedia.org/wiki/Watson\_%28computer%29](http://en.wikipedia.org/wiki/Watson_(computer)http://en.wikipedia.org/wiki/Watson_(computer))) but it is an open-source project that can help you quickly process data according to large sets of business rules and it will allow you to define those rules in a readable, user-friendly way. When looking at a rule, it is pretty clear of what it is about to both a developer with little business domain knowledge and to a non technical business analyst:

**rule** "Valid for loan"

**when**

For an active loan type

There is a person that

- is at least 18 years old

- is not too old for the loan

- earns more than or equal to the minimum income for the loan

- has compatible citizenship with the loan

**then**

Tell that person they qualify for the loan

**end**

**1. Some theoretical background**

Expert Systems are computer systems that make decisions like a human expert would, based on a way of representing knowledge (which forms their so called *knowledge base*) they infer conclusions. This is different from conventional programming because it doesn't work by following a procedure but instead it tries to mimic human reasoning about knowledge. Drools is a Rule Engine that uses the rule-based approach to implement an Expert System. A Rule engine is any system that uses rules, in any form, that can be applied to data to produce outcomes.

To bring back some of the traumas of the August exams in college, the official documentation adds that Drools is more precisely classified as a Production Rule System, a concept in Formal Grammars (<http://en.wikipedia.org/wiki/Formal_grammar>).

In rulebased systems, knowledge is represented in the form of if-then rules. For example, the following rule could be part of such a system:

IF Person wants to buy a house

Person does not have enough money for a house

THEN Person goes to bank for a loan

To actually trigger this rule, we will need a Person object or *fact* matching the conditions of the rule. We need to provide our rules with a number of facts where they can work upon.

The process that decides weather each fact satisfies the Rules is called Pattern Matching, and is performed by the so called Inference Engine. There are several algorithms that can be used by an Inference Engine for pattern matching, like Rete or Leaps. Drools is based on a Java implementation of Rete which they called ReteOO. If a fact satisfies more than one rule, the matched rules are said to be in conflict and it becomes the job of a component called Agenda to decide the order in which those rules will be executed. The Rules are stored in the Production Memory and the facts that the Inference Engine matches against are kept in the Working Memory.

![A description...](data:None;base64,)

**2. Usages of a Rule Engine**

Rule Engines are a suitable solution for problems that don't have a satisfactory traditional programming approach. Here are some typical scenarios when this can happen:

* the problem may not be complex, but you can't see a non-fragile way of building a solution for it
* the problem is beyond any obvious algorithmic solution.
* the logic changes often
* domain experts (or business analysts) are readily available, but are nontechnical.

Obviously, rules are not any silver bullet, their usage is not suited for workflow or process execution. Rule engines work best when you are able to write declarative rules.

Some common use cases when Rule engines are recommended are: validations, calculations, routing and filtering, monitoring and diagnostics, logistics, etc.

**3. The Drools language**

To showcase some of the basic features of Drools I created a small Java project called LoanAdvisor that includes a simple expert system that can give feedback to a bank's clients about their eligibility for each available type of loan. We need some basic POJOs, first:

**public** **class** Person {

**private** String name;

**private** Integer age;

**private** Country citizenship;

**private** Double income;

...

}

**public** **class** LoanType {

**private** String name;

**private** Integer maxAge;

**private** Set<Country> compatibleCitizenships;

**private** Double minIncome;

**private** **boolean** active;

…

}

And a service to process the feedback for each person:

**public** **interface** IFeedbackService {

**void** addFeedback(Person person, String message);

String getFeedback();

}

Next comes the fun part, writing the rules. Drools allows you to write rules in more than one way. First step is to see the "native" rule language at work.

*3.1 Drools native rule format - DRL*

A rule file is simple a text file, typically with a .drl extension that is short for Drools Rule Language. In a DRL file you can have multiple rules, queries and functions, as well as some resource declarations like imports, globals and attributes that are assigned and used by your rules and queries. However, you are also able to spread your rules across multiple rule files (in that case, the extension .rule is suggested, but not required).

Let's dig into native language rules file src/main/rules/LoanAdvisor.drl:

1. **package** loan.advisor.drl
2. **import** loan.advisor.dto.Person
3. **import** loan.advisor.dto.LoanType
4. **import** loan.advisor.services.FeedbackService
5. **global** FeedbackService feedbackService

First is the package definition. A package represents a namespace, which ideally is kept unique for a given grouping of rules. The package name itself is the namespace, and is not related to files or folders in any way. Next come the imports, the Java types we reference. Then comes a so called global object that is an object that won't be part of our working memory but that is needed by our rules; these could be anything from services to loggers.

1. **rule** "Underage Persons"
2. **salience** 100 //filter out the kids first
3. **when**
4. $person: Person(age < 18)
5. **then**
6. feedbackService.addFeedback($person, "You have to grow up first!");
7. **retract**($person);
8. end
9. **rule** "Above maximum age limit"
10. **when**
11. LoanType(active==**true**, $loanName: name, $maxAge: maxAge)
12. $person: Person(age > $maxAge)
13. **then**
14. feedbackService.addFeedback($person,
15. String.format("You do NOT qualify for '%s': you are older than %d.", $loanName, $maxAge));
16. end

The first rule, called "Underage Persons", is supposed to identify the persons that are below the minimum legal age for asking a bank loan and remove them from the working memory since they won't be eligible for any kind of credit. So it makes sense that this should always be the very first rule to be executed. This is done by specifying a salience for the rule (line 10), the higher the salience, the higher the priority of the rule. 100 is a random value, since all the other rules have default saliences, any positive integer would have been the same.

The two most interesting parts of a rule are the when and the then part. The when part, which is called the left hand side(LHS) of the rule, contains the conditions that need to be fulfilled in order for the then part, called the right hand side(RHS) of the rule, to execute. If the LHS doesn't match anything in the working memory, the RHS is not executed. Writing a rule, the mindset is a little similar to writing SQL statements.

Line 12 contains a so called pattern that matches any instance of Person from the Working Memory that has the age property less than 18 and then it binds it to a variable called $person. The process of matching patterns against the inserted data is referred to as pattern matching. The prefixed dollar symbol ($) is just a convention but it can be useful in complex rules where it helps to easily differentiate between variables and fields. Also note the way the age property is being accessed. Drools follows the standard Java bean specification by using standard JDK Introspector class to map the properties. So all you need are public accessors that use the standard naming conventions. While something like Person(getAge() < 18) is also legal, by working directly with the fields we allow Drools to create field indexes that enhance the performance. Another thing to note is that Drools supports autoboxing and unboxing.

Line 14 sends add a feedback message for the matched person using our service. After that we remove the object from the working memory so it won't be processed by the other rules.

The second rule, "Above maximum age limit", compares the maximum age permited by each credit type with each person's age and tells the persons that are too old the reason they don't qualify for the loan. Here, at line 20, we are not binding a variable to a LoanType object but instead we are binding variables to the bean's properties: $loanName: name, $maxAge: maxAge

We can have more than one constraint in a single pattern. Let's see the last rule that gives positive feedback to the persons that satisfy all requirements for a type of loan:

1. **rule** "Valid for loan"
2. **when**
3. $loan: LoanType(active==**true**)
4. $person: Person(age >= 18, age <= $loan.maxAge,
5. income >= $loan.minIncome,
6. $loan.compatibleCitizenships **contains** citizenship)
7. **then**
8. feedbackService.addFeedback($person,
9. String.format("You qualify for '%s'! One of our agents will contact you shortly.", $loan.getName()));
10. **end**

Here, at lines 48 to 50, the 4 conditions are separated by commas which is the Drools AND logical operator. As you can see on line 50, DRL offers some convenience operators for working on Collections and Maps like contains, member of, in, etc.

*3.2 Domain Specific Language – DSL*

DSLs can serve as a layer of separation between rule authoring (and rule authors) and the

technical intricacies resulting from the modelling of domain object and the rule engine's native

language and methods. If your rules need to be read and validated by domain experts (such as

business analysts, for instance) who are not programmers, you should consider using a DSL; it

hides implementation details and focuses on the rule logic proper. DSL sentences can also act as

"templates" for conditional elements and consequence actions that are used repeatedly in your

rules, possibly with minor variations. You may define DSL sentences as being mapped to these

repeated phrases, with parameters providing a means for accomodating those variations.

Using DSL has no impact on performance because it works by changing the parsing at compile time.

Here is how the first two DRL rules we looked at look like when using a DLS file. This is from src/main/rules/LoanAdvisor.dslr:

1. **package** loan.advisor.dslr
2. **import** loan.advisor.dto.Person
3. **import** loan.advisor.dto.LoanType
4. **import** loan.advisor.services.FeedbackService
5. **global** FeedbackService feedbackService
6. **expander** dictionary.dsl
7. **rule** "Underage Persons"
8. **salience** 100
9. **when**
10. There **is** a person younger than 18
11. **then**
12. Tell that person "you have to grow up first!"
13. Exclude that person
14. end
15. **rule** "Above maximum age limit"
16. **when**
17. For an active loan type
18. A person **is** too old
19. **then**
20. Tell that person they **do** **not** qualify because of "maximum age limit"
21. end

The magic is done by creating another text file dictionary.dsl to describe our domain specific language and by declaring it as an expander for the rule file (line 9).

A dsl file contains a set of expressions and their synonym in DRL. For instance

[when]There is a person younger than {minAge}=$person:Person(age < {minAge})

[then]Exclude that person=retract($person);

The part enclosed inside the square parentheses specifies if the statement can be used in the LHS or in RHS. Next is the statement that can contain a place holder enclosed in accolades. The place holder {minAge} is passed from the rules file. What is left of the equals sign is then translated into the DRL expressions at compile-time.

Here is the rule that selects the persons valid for a loan type:

1. **rule** "Valid for loan"
2. **when**
3. For an active loan type
4. There **is** a person that
5. - **is** at least 18 years old
6. - **is** **not** too old **for** the loan
7. - earns more than **or** equal to the minimum income **for** the loan
8. - has compatible citizenship **with** the loan
9. **then**
10. Tell that person they qualify **for** the loan
11. **end**

And here is the LHS in the DLS:

1. [when]There is a person that=$person:Person()
2. [when]- is at least {minAge} years old=age >= {minAge}
3. [when]- is not too old for the loan=age <= $loan.maxAge
4. [when]- earns more than or equal to the minimum income for the loan=income >= $loan.minIncome
5. [when]- has compatible citizenship with the loan=$loan.compatibleCitizenships contains citizenship

The – after the [when] signals Drools that the condition will refer only to the fields of a selected object. It is a way to make the code more readable.

1. **Running the rules**

So far, we described how it all should work, but we didnt see the rule engine in action so far. The following code will change this:

1. // load up the knowledge base
2. KnowledgeBase kbase = *readDslKnowledgeBase*();
4. StatefulKnowledgeSession ksession = kbase
5. .newStatefulKnowledgeSession();
6. KnowledgeRuntimeLogger logger = KnowledgeRuntimeLoggerFactory
7. .*newFileLogger*(ksession, "loanAdvisor");
9. ksession.setGlobal("feedbackService", **new** FeedbackService());
10. *populateSession*(ksession);
11. // go !
12. ksession.fireAllRules();

It all starts from a KnowledgeBase object that contains our rules. From this we need a session which can be a stateless one if we don't need to use the inference engine or stateful for longer lived iterative processing. Notice line 9 where we link the global variable with a java object.

The knowledge base instantiation is pretty straight forward, it adds up to using a KnowledgeBuilder load and parse the rules files into so called knowledge packages:

1. **private** **static** KnowledgeBase readDslKnowledgeBase() **throws** Exception {
2. KnowledgeBuilder kbuilder = KnowledgeBuilderFactory
3. .*newKnowledgeBuilder*();
4. kbuilder.add(ResourceFactory.*newClassPathResource*("dictionary.dsl"),
5. ResourceType.*DSL*);
6. kbuilder.add(ResourceFactory.*newClassPathResource*("LoanAdvisor.dslr"),
7. ResourceType.*DSLR*);
8. KnowledgeBuilderErrors errors = kbuilder.getErrors();
9. **if** (errors.size() > 0) {
10. **for** (KnowledgeBuilderError error : errors) {
11. System.*err*.println(error);
12. }
13. **throw** **new** IllegalArgumentException("Could not parse knowledge.");
14. }
15. KnowledgeBase kbase = KnowledgeBaseFactory.*newKnowledgeBase*();
16. kbase.addKnowledgePackages(kbuilder.getKnowledgePackages());
17. **return** kbase;
18. }

The *populateSession* method populates the session object with the objects that we want in our working memory. For this test we'll insert 6 persons and 4 loan types:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Name | Maximum age | Compatible citizenships | Minimum income | Is loan active? |
| The Standard Credit | 65 | UK, French | 90,000 | true |
| The Platinum Credit | 80 | UK, French | 190,000 | true |
| The Student Loan | 23 | UK, French | 25,000 | true |
| The Poor Student Loan | 20 | UK, French | 12,000 | false |

Here is a basic output of the FeedbacService after running all the rules:

TOM HOPPER, aged 17, citizen of UK, yearly income of 0.00 €:

You have to grow up first!

MIHAI IONESCU, aged 33, citizen of Romania, yearly income of 80,000.00 €:

You do NOT qualify for 'The Student Loan': this loan is available only in [France, UK] .

You do NOT qualify for 'The Student Loan': you are older than 23.

You do NOT qualify for 'The Platinum Credit': this loan is available only in [France, UK] .

You do NOT qualify for 'The Platinum Credit': you earn less than 190,000.00 €.

You do NOT qualify for 'The Standard Credit': this loan is available only in [France, UK] .

You do NOT qualify for 'The Standard Credit': you earn less than 90,000.00 €.

JANE MILLER, aged 35, citizen of UK, yearly income of 95,000.00 €:

You do NOT qualify for 'The Student Loan': you are older than 23.

You do NOT qualify for 'The Platinum Credit': you earn less than 190,000.00 €.

You qualify for 'The Standard Credit'! One of our agents will contact you shortly.

JHON ADAMS, aged 21, citizen of UK, yearly income of 24,000.00 €:

You do NOT qualify for 'The Student Loan': you earn less than 25,000.00 €.

You do NOT qualify for 'The Platinum Credit': you earn less than 190,000.00 €.

You do NOT qualify for 'The Standard Credit': you earn less than 90,000.00 €.

MIKE MORRISON, aged 71, citizen of UK, yearly income of 190,000.00 €:

You do NOT qualify for 'The Student Loan': you are older than 23.

You qualify for 'The Platinum Credit'! One of our agents will contact you shortly.

You do NOT qualify for 'The Standard Credit': you are older than 65.

JEAN ALESSI, aged 25, citizen of France, yearly income of 86,000.00 €:

You do NOT qualify for 'The Student Loan': you are older than 23.

You do NOT qualify for 'The Platinum Credit': you earn less than 190,000.00 €.

You do NOT qualify for 'The Standard Credit': you earn less than 90,000.00 €.