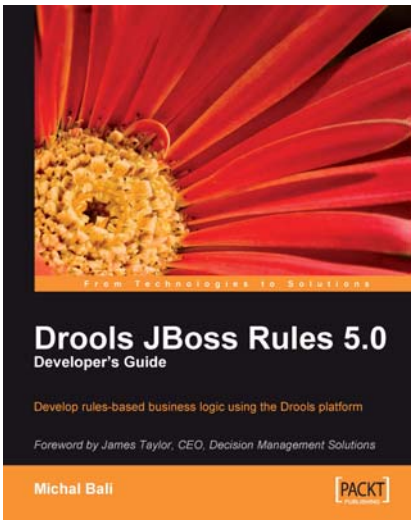
Drools JBoss Rules 5.0 Developer's Guide

Michal Bali



Chapter No. 5
"Human-readable Rules"

In this package, you will find:

A Biography of the author of the book

A preview chapter from the book, Chapter NO.5 "Human-readable Rules"

A synopsis of the book's content

Information on where to buy this book

About the Author

Michal Bali is a senior software developer at DeCare Systems, Ireland. He has four years experience working with Drools and has extensive knowledge of Java, JEE. Michal designed and implemented several systems for a major dental insurance company. Michal is an active member of the Drools community and can be contacted at `michalbali@gmail.com`.

I thank Drools lead Mark Proctor and his team that consists of Edson Tirelli, Michael Neale, Kris Verlaenen, Toni Rikkola, and other contributors for giving me something to write about. They were of great help while I was writing the book. Edson and Mark reviewed the book and helped me to correct various inaccuracies.

I'd like to thank all reviewers and the whole editorial team for their patience and help while I was writing this book. In particular, Sarah Cullington, Siddharth Mangarole, Aanchal Kumar, Conrad Sardinha, James Taylor, Sammy Larbi, Zainab Bagasrawala, Shilpa Dube, Lata Basantani, and other anonymous reviewers.

I thank James Taylor for writing the foreword and reviewing the book. I am honored that he chose to participate in this project.

Finally, I thank my fiancée Michala for supporting me and putting up with me while I wrote.

Drools JBoss Rules 5.0 Developer's Guide

Business rules and processes can help your business by providing a level of agility and flexibility. As a developer, you will be largely responsible for implementing these business rules and processes effectively, but implementing them systematically can often be difficult due to their complexity. Drools, or JBoss Rules, makes the process of implementing these rules and processes quicker and handles the complexity, making your life a lot easier!

This book guides you through various features of Drools, such as rules, ruleflows, decision tables, complex event processing, Drools Rete implementation with various optimizations, and others. It will help you to set up the Drools platform and start creating your own business. It's easy to start developing with Drools if you follow our real-world examples that are intended to make your life easier.

Starting with an introduction to the basic syntax that is essential for writing rules, the book will guide you through validation and human-readable rules that define, maintain, and support your business agility. As a developer, you will be expected to represent policies, procedures, and constraints regarding how an enterprise conducts its business; this book makes it easier by showing you the ways in which it can be done.

A real-life example of a banking domain allows you to see how the internal workings of the rules engine operate. A loan approval process example shows the use of the Drools Flow module. Parts of a banking fraud detection system are implemented with Drools Fusion module, which is the Complex Event Processing part of Drools. This in turn, will help developers to work on preventing fraudulent users from accessing systems in an illegal way.

Finally, more technical details are shown on the inner workings of Drools, the implementation of the ReteOO algorithm, indexing, node sharing, and partitioning.

What This Book Covers

Chapter 1: This chapter introduces the reader to the domain of business rules and business processes. It talks about why the standard solutions fail at implementing complex business logic. It shows a possible solution in the form of a declarative programming model. The chapter talks about advantages and disadvantages of Drools. A brief history of Drools is also mentioned.

Chapter 2: This chapter shows us the basics of working with the Drools rule engine—Drools Expert. It starts with a simple example that is explained step-by-step. It begins with the development environment setup, writing a simple rule, and then executing it. The chapter goes through some necessary keywords and concepts that are needed for more complex examples.

Chapter 3: This chapter introduces the reader to a banking domain that will be the basis for examples later in this book. The chapter then goes through an implementation of a decision service for validating this banking domain. A reporting model is designed that holds reports generated by this service.

Chapter 4: This chapter shows how Drools can be used for carrying out complex data transformation tasks. It starts with writing some rules to load the data, continues with the implementation of various transformation rules, and finally puts together the results of this transformation. The chapter shows how we can work with a generic data structure such as a map in Drools.

Chapter 5: The focus of this chapter is on rules that are easy to read and change. Starting with domain specific languages, the chapter shows how to create a data transformation specific language. Next, it focuses on decision tables as another more user-friendly way of representing business rules. An interest rate calculation example is shown. Finally, the chapter introduces the reader to Drools Flow module as a way of managing the rule execution order.

Chapter 6: This chapter talks about executing the validation decision service in a stateful manner. The validation results are accumulated between service calls. This shows another way of interacting with a rule engine. Logical assertions are used to keep the report up-to-date. Various ways of serializing a stateful session are discussed.

Chapter 7: This chapter talks about Drools Fusion—another cornerstone of the Drools platform is about writing rules that react to various events. The power of Drools Fusion is shown through a banking fraud detection system. The chapter goes through various features such as events, type declarations, temporal operators, sliding windows, and others.

Chapter 8: This chapter goes into more detail about the workflow aspect of the Drools platform. It is showed through a loan approval service that demonstrates the use of various nodes in a flow. Among other things, the chapter talks about implementing a custom work item, human task, or a sub-flow.

Chapter 9: The purpose of this chapter is to show you how to integrate Drools in a real web application. We'll go through design and implementation of persistence, business logic, and presentation layers. All of the examples written so far will be integrated into this application.

Chapter 10: The focus of this chapter is to give you an idea about the various ways of testing your business logic. Starting with unit testing, integration testing through acceptance testing that will be shown with the help of the Business Rules Management Server—Guvnor, this chapter provides useful advice on various troubleshooting techniques.

Chapter 11: This chapter shows integration with the Spring Framework. It describes how we can make changes to rules and processes while the application runs. It shows how to use an external build tool such as Ant to compile rules and processes. It talks about the rule execution server that allows us to execute rules remotely. It briefly mentions support of various standards.

Chapter 12: This chapter goes under the hood of the Drools rule engine. By understanding how the technology works, you'll be able to write more efficient rules and processes. It talks about the ReteOO algorithm, node sharing, node indexing, and rule partitioning for parallel execution.

Appendix A: It lists various steps required to get you up and running with Drools.

Appendix B: It shows an implementation of a custom operator that can be used to simplify our rules.

Appendix C: It lists various dependencies used by the sample web application.

5
Human-readable Rules

Business rules implementations presented so far were aimed mostly at developers. However, it is sometimes needed that these rules are readable and understandable by the business analysts. Ideally, they should be able to change the rules or even write new ones. An important aspect of business rules is their readability and user friendliness. Looking at a rule, you should immediately have an idea of what it is about. In this chapter, we'll look at **Domain Specific Languages** (**DSLs**), decision tables, and rule flows to create human-readable rules.

Domain Specific Language

The **domain** in this sense represents the business area (for example, life insurance or billing). Rules are expressed with the terminology of the problem domain. This means that domain experts can understand, validate, and modify these rules more easily.

You can think of DSL as a translator. It defines how to translate sentences from the problem-specific terminology into rules. The translation process is defined in a `.dsl` file. The sentences themselves are stored in a `.dslr` file. The result of this process must be a valid `.drl` file.

Building a simple DSL might look like:

```
[condition][]There is a Customer with firstName
{name}=$customer : Customer(firstName == {name})
[consequence][]Greet Customer=System.out.println("Hello " +
$customer.getFirstName());
```

Code listing 1: Simple DSL file `simple.dsl`.

The code listing above contains only two lines (each begins with `[`). However, because the lines are too long, they are wrapped effectively creating four lines. This will be the case in most of the code listings.

When you are using the Drools Eclipse plugin to write this DSL, enter the text before the first equal sign into the field called **Language expression**, the text after equal sign into **Rule mapping**, leave the **object** field blank and select the correct scope.

The previous DSL defines two DSL mappings. They map a DSLR sentence to a DRL rule. The first one translates to a condition that matches a `Customer` object with the specified first name. The first name is captured into a variable called `name`. This variable is then used in the rule condition. The second line translates to a greeting message that is printed on the console. The following `.dslr` file can be written based on the previous DSL:

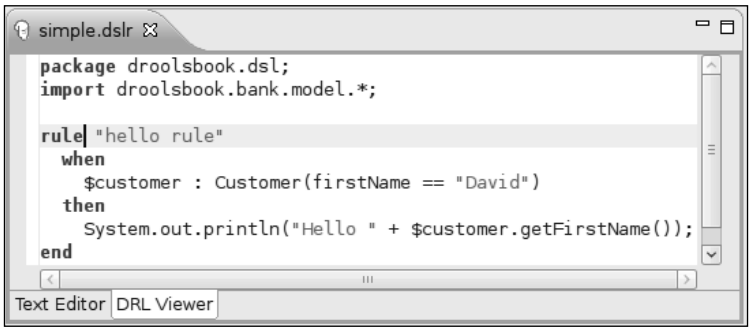
```
package droolsbook.dsl;
import droolsbook.bank.model.*;
expander simple.dsl
rule "hello rule"
  when
    There is a Customer with firstName "David"
  then
    Greet Customer
end
```

Code listing 2: Simple `.dslr` file (`simple.dslr`) with rule that greets a customer with name `David`.

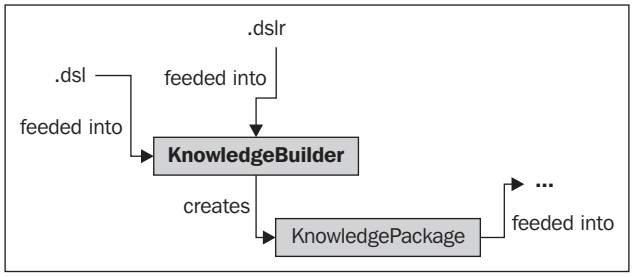
As can be seen, the structure of a `.dslr` file is the same as the structure of a `.drl` file. Only the rule conditions and consequences are different. Another thing to note is the line containing `expander simple.dsl`. It informs Drools how to translate sentences in this file into valid rules. Drools reads the `simple.dslr` file and *tries to translate/expand each line by applying all mappings* from the `simple.dsl` file (it does it in a single pass process, line-by-line from top to bottom). The *order of lines is important* in a `.dsl` file. Please note that one condition/consequence must be written on one line, otherwise the expansion won't work (for example, the condition after the `when` clause, from the rule above, must be on one line).

When you are writing `.dslr` files, consider using the Drools Eclipse plugin. It provides a special editor for `.dslr` files that has an editing mode and a read-only mode for viewing the resulting `.drl` file. A simple DSL editor is provided as well.

The result of the translation process will look like the following screenshot:



This translation process happens in memory and no `.drl` file is physically stored. We can now run this example. First of all, a knowledge base must be created from the `simple.dsl` and `simple.dslr` files. The process of creating a package using a DSL is as follows (only the package creation is shown, the rest is the same as we've seen in Chapter 2, *Basic Rules*):



`KnowledgeBuilder` acts as the translator. It takes the `.dslr` file, and based on the `.dsl` file, creates the DRL. This DRL is then used as normal (we don't see it; it's internal to `KnowledgeBuilder`). The implementation is as follows:

```
private KnowledgeBase createKnowledgeBaseFromDSL()
    throws Exception {
  KnowledgeBuilder builder =
      KnowledgeBuilderFactory.newKnowledgeBuilder();
  builder.add(ResourceFactory.newClassPathResource(
      "simple.dsl"), ResourceType.DSL);
  builder.add(ResourceFactory.newClassPathResource(
      "simple.dslr"), ResourceType.DSLR);
  if (builder.hasErrors()) {
    throw new RuntimeException(builder.getErrors()
        .toString());
  }
  KnowledgeBase knowledgeBase = KnowledgeBaseFactory
```

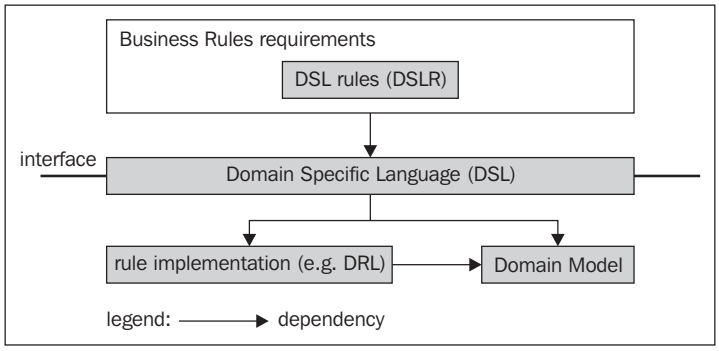
```
            .newKnowledgeBase();
        knowledgeBase.addKnowledgePackages(
            builder.getKnowledgePackages());
        return knowledgeBase;
    }
```

Code listing 3: Creating knowledge base from `.dsl` and `.dslr` files.

The `.dsl` and subsequently the `.dslr` files are passed into `KnowledgeBuilder`. The rest is similar to what we've seen before.

DSL as an interface

DSLs can be also looked at as another level of indirection between your `.drl` files and business requirements. It works as shown in the following figure:



The figure above shows DSL as an interface (dependency diagram). At the top are the business requirements as defined by the business analyst. These requirements are represented as DSL sentences (`.dslr` file). The DSL then represents the interface between DSL sentences and rule implementation (`.drl` file) and the domain model. For example, we can change the transformation to make the resulting rules more efficient without changing the language. Further, we can change the language, for example, to make it more user friendly, without changing the rules. All this can be done just by changing the `.dsl` file.

DSL for validation rules

The first three implemented object/field required rules from Chapter 2, *Basic Rules*, can be rewritten as:

- If the Customer does not have an address, then Display warning message
- If the Customer does not have a phone number or it is blank, then Display error message

- If the Account does not have an owner, then Display error message for Account

We can clearly see that all of them operate on some object (Customer/Account), test its property (address/phone/owner), and display a message (warning/error) possibly with some context (account). Our `validation.dslr` file might look like the following code:

```
expander validation.dsl

rule "address is required"
   when
      The Customer does not have address
   then
      Display warning
end

rule "phone number is required"
   when
      The Customer does not have phone number or it is blank
   then
      Display error
end

rule "account owner is required"
   when
      The Account does not have owner
   then
      Display error for Account
end
```

Code listing 4: First DSL approach at defining the required object/field rules (`validation.dslr` file).

The conditions could be mapped like this:

```
[condition][]The {object} does not have {field}=${object} : {object}(
{field} == null )
```

Code listing 5: `validation.dsl`.

This covers the address and account conditions completely. For the phone number rule, we have to add the following mapping at the beginning of the `validation.dsl` file:

```
[condition][] or it is blank =  == "" ||
```

Code listing 6: Mapping that checks for a blank phone number.

As it stands, the phone number condition will be expanded to:

```
$Customer : Customer( phone number  == "" || == null )
```

Code listing 7: Unfinished phone number condition.

To correct it, `phone number` has to be mapped to `phoneNumber`. This can be done by adding the following at the end of the `validation.dsl` file:

```
[condition][]phone number=phoneNumber
```

Code listing 8: Phone number mapping.

The conditions are working. Now, let's focus on the consequences. The following mapping will do the job:

```
[consequence][]Display {message_type} for {object}={message_type}(
 kcontext, ${object} );
[consequence][]Display {message_type}={message_type}( kcontext );
```

Code listing 9: Consequence mappings.

The three validation rules are now being expanded to the same `.drl` representation as we've seen in Chapter 2.

File formats

Before we go further, we'll examine each file format in more detail.

DSL file format

A line in a `.dsl` file has the following format:

```
[<scope>][<Type>]<language expression>=<rule mapping>
```

Code listing 10: The format of one line in a `.dsl` file.

As we've already seen, an example of a line in DSL file might look like this:

```
[condition][droolsbook.bank.model.Customer]The Customer does not have
address=Customer(address == null)
```

Code listing 11: Sample line from DSL file (note that it is just one line that has been wrapped).

The scope can have the following values:

- `condition`: Specifies that this mapping can be used in the condition part of a rule.
- `consequence`: Specifies that this mapping can be used in the consequence part of a rule.
- `*`: Specifies that this mapping can be used in both the condition and the consequence part of a rule.
- `keyword`: This mapping is applied to the whole file (not just the condition or the consequence part). Used mainly when writing DSLs in languages other than English or to hide the package/import/global statements at the beginning of the file behind a business friendly sentence.

Type can be used to further limit the scope of the mapping. Scope and Type are used by the Drools Eclipse plugin to provide auto-completion when writing `.dslr` files (when pressing *Ctrl* + *Space*, only relevant choices are offered). This is especially useful with the multiple constraints feature (refer to the section, *DSL for multiple constraints in a condition*).

DSL supports comments by starting the line with the hash character, `#`. For example:

```
#this is a comment in a .dsl file
```

DRL file format

As a side note, in a `.drl` file, it is valid to write the whole rule on a single line. This allows us to write more complex DSLs because one sentence in `.dslr` file can be translated into multiple conditions—even the whole rule. For example, these are valid rules on a single line:

```
rule "addressRequired" when Customer( address == null ) then
warning(kcontext); end
```

Code listing 12: `addressRequired` rule *on one line*.

Make sure that you add spaces between Drools keywords. Another more complex example of a rule on one line:

```
rule "studentAccountCustomerAgeLessThan" when Customer( eval (year
sPassedSince(dateOfBirth) >= 27) ) and $account : Account( type ==
Account.Type.STUDENT ) then error(kcontext, $account); System.out.
println("another statement"); end
```

Code listing 13: `studentAccountCustomerAgeLessThan` rule *on one line*.

The preceding rule contains two conditions and two Java statements in the consequence block. There is also an optional `and` keyword between the conditions to make it more readable.

DSLR file format

A `.dslr` file contains the sentences written using the DSL. The `.dslr` file is very similar to the `.drl` file. One thing to note is that by prepending a line with a '`>`', we can turn off the expander for the line. This allows us to write a hybrid `.dslr` file that contains traditional DRL rules and DSL rules. For example, if we are not yet sure how to map some complex rule, we can leave it in its original `.drl` file format.

DSL for multiple constraints in a condition

We'll go through more complex DSLs. Let's look at a standard condition for example:

```
Account( owner != null, balance > 100, currency == "EUR" )
```

Code listing 14: Condition that matches some account.

It is difficult to write DSL that will allow us to create conditions with any subset of constraints from the code listing above (without writing down all possible permutations). The '`-`' feature comes to the rescue:

```
[condition][]There is an Account that=$account : Account( )
[condition][]-has owner=owner != null
[condition][]-has balance greater than {amount}=balance > {amount}
[condition][]-has currency equal to {currency}=currency == {currency}
```

Code listing 15: DSL using the '`-`' feature. This can create seven combinations of the constraints.

When the DSL condition starts with '`-`', the DSL parser knows that this constraint should be added to the last condition (in a `.dslr` file). With the preceding DSL, the following condition can be created:

```
There is an Account that
-  has currency equal to "USD"
"has balance greater than 2000"
```

Code listing 16: Condition using the '`-`' feature (in a `.dslr` file).

The `'-'` feature increases the flexibility of the resulting language. It works just fine for simple cases involving only one pair of brackets. In case of multiple brackets in the condition, Drools always adds the constraint to the last pair of brackets. This may not always be what we want. We have to find a different way of specifying multiple constraints in a condition. We can also write our DSL in the following manner:

```
[condition][]There is an Account that {constraints} = Account(
{constraints} )
[condition][]has {field} equal to {value}={field} == {value}
[condition][]and has {field} equal to {value}=, {field} == {value}
```

Code listing 17: Flexible DSL that can be expanded to a condition with two field constraints.

With this DSL, the following DSLR can be written:

```
There is an Account that has owner equal to null and has balance equal
to 100
```

Code listing 18: DSLR that describes an account with two constraints.

If we want to have more conditions, we can simply duplicate the last line in the DSL. Remember? Translation is a single pass process.

Named capture groups

Sometimes, when a more complex DSL is needed, we need to be more precise at specifying what a valid match is. We can use named capture groups with regular expressions to give us the needed precision. For example:

```
{name:[a-zA-Z]+}
```

Code listing 19: Name that matches only characters.

Regular expressions (`java.util.regex.Pattern`) can be used not only for capturing variables but also within the DSL. For example, in order to carry out case insensitive matching. If we look at the DSL from code listing 15, the users should be allowed to type Account, account, ACCOUNT, or even aCcount in their `.dslr` files. This can be done by enabling the embedded case insensitive flag expression—`(?i)`: `[condition][]There is an (?i:account) that ....`

Another useful example is sentences that are sensitive to gender—`(s)?he` to support "he" and "she", and so on.

In order to make the sentences space insensitive, Drools automatically replaces all spaces with `\s+`. Each `\s+` matches one or more spaces. For example, the following line in a `.dslr` file will be successfully expanded by the DSL from code listing 15:

```
There    is          an          Account that ....
```

DSL for data transformation rules

We'll now implement DSL for the data transformation rules from Chapter 4, *Data Transformation*. We'll reuse our rule unit tests to verify that we don't change the functionality of the rules but only their representation. The unit test class will be extended and the method for creating `KnowledgeBase` will be overridden to use the `.dsl` file and `.dslr` file as inputs. Rule names will stay the same. Let's start with the `twoEqualAddressesDifferentInstance` rule:

```
rule twoEqualAddressesDifferentInstance
   when
      There is legacy Address-1
      There is legacy Address-2
      - same as legacy Address-1
   then
      remove legacy Address-2
      Display WARNING for legacy Address-2
end
```

Code listing 20: Rule for removing redundant addresses (`dataTransformation.dslr` file).

The conditions can be implemented with the following DSL:

```
[condition][] legacy {object}-{id} = {object}-{id}
[condition][] There is {object}-{id} = ${object}{id} : Map( this["_
type_"] == "{object}" )
[condition][]- same as {object}-{id} = this == ${object}{id}, eval(
${object}1 != ${object}2 )
```

Code listing 21: DSL for conditions (`dataTransformation.dsl` file).

The first mapping is a simple translation rule, where we remove the word `legacy`. The next mapping captures a map with its type. The last mapping includes the equality test with the object identity test. Mapping for consequences is as follows:

```
[consequence][] legacy {object}-{id} = ${object}{id}
[consequence][]Display {message_type_enum} for {object}=validationRepo
rt.addMessage(reportFactory.createMessage(Message.Type.{message_type_
enum}, kcontext.getRule().getName(), {object}));
[consequence][]remove {object} = retract( {object} );
```

Code listing 22: DSL for consequences.

The first mapping just removes the word `legacy`. The second mapping adds a message to `validationReport`. Finally, the last mapping removes an object from the knowledge session. This is all we need for the `twoEqualAddressesDifferentInstance` rule.

As you can see, we started with the sentence in the domain specific language (code listing 1) and then we've written the transformation to reflect the rules (from Chapter 4). In reality, this is an iterative process. You'll modify the `.dslr` and `.dsl` files until you are happy with the results. It is also a good idea to write your rules in standard `.drl` first and only then try to write a DSL for them.

We'll move to the next rule, `addressNormalizationUSA`:

```
rule addressNormalizationUSA
   when
      There is legacy Address-1
      - country is one of "US", "U.S.", "USA", "U.S.A"
   then
      for legacy Address-1 set country to USA
end
```

Code listing 23: DSLR rule for normalizing address country field.

The rule just needs another constraint type:

```
[condition][]- country is one of {country_list} = this["country"] in
({country_list})
```

Code listing 24: Another condition mapping.

The consequence is defined with two mappings. The first one will translate the country to an `enum` and the second will then perform the assignment.

```
[consequence][]set country to {country}=set country to Address.
Country.{country}
[consequence][]for {object}set {field} to {value} = modify( {object} )
\{ put("{field}", {value} ) \}
```

Code listing 25: Consequence mapping for the country normalization rule.

Please note that the curly brackets are escaped. Moreover, the original rule used `mvel` dialect. It is a good idea to write your rules using the same dialect. It makes the DSL easier. Otherwise, the DSL will have to be "dialect aware".

The other country normalization rule can be written without modifying the DSL. We'll now continue with `unknownCountry` rule:

```
rule unknownCountry
Apply after address normalizations
   when
   There is legacy Address-1
   - country is not normalized
```

```
    then
       Display ERROR for legacy Address-1
end
```

Code listing 26: DSLR representation of the `unknownCountry` rule.

The whole sentence `Apply after address normalizations` is mapped as a keyword mapping:

```
[keyword][] Apply after address normalizations = salience -10
```

Code listing 27: `salience` keyword mapping.

Now, we can use the other rule attributes to achieve the same goal just by changing the DSL.

Additional mapping that is needed:

```
[condition][]- country is not normalized = eval(!($Address1.
get("country") instanceof Address.Country))
```

Code listing 28: Another condition mapping.

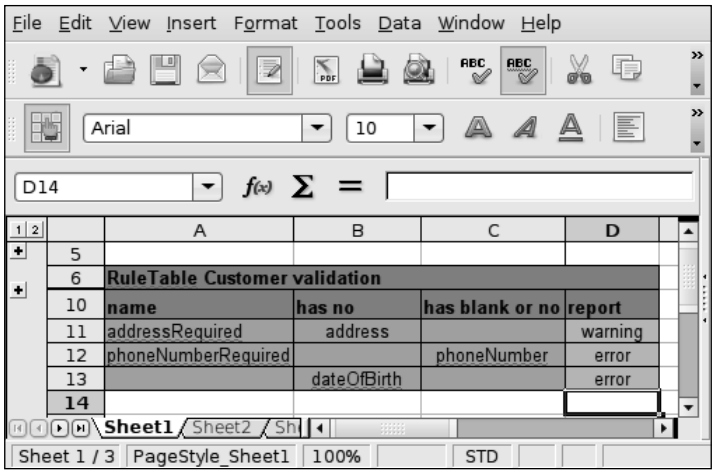
In the condition mapping, the `$Address1` is hard-coded. This is fine for the rules that we have.

As you can imagine, the rest of the rules follow similar principles.

What we have achieved by writing this DSL is better readability. A business analyst can verify the correctness of these rules more easily. We could push this further by defining a complete DSL that can represent any concept from the problem domain. The business analyst will then be able to express any business requirement just by editing the `.dslr` file.

Decision tables

Decision tables are another form of human-readable rules that are useful when there are lots of similar rules with different values. Rules that share the same conditions with different parameters can be captured in a decision table. Decision tables can be represented in an Excel spreadsheet (`.xls` file) or a comma separated values (`.csv` file) format. Starting from version 5.0, Drools supports web-based decision tables as well. They won't be discussed in this book; however, they are very similar. Let's have a look at a simple decision table in `.xls` format.



	A	B	C	D
5				
6	**RuleTable Customer validation**			
10	**name**	**has no**	**has blank or no**	**report**
11	addressRequired	address		warning
12	phoneNumberRequired		phoneNumber	error
13		dateOfBirth		error
14				

The preceding screenshot shows a decision table in `validation.xls` opened with OpenOffice Calc editor. It shows one decision table for validating a customer. Line **10** shows four columns. The first one defines rule name, the next two define conditions, and the last one is for defining actions/consequences. The next three lines (**11-13**) represent the individual rules—one line per rule. Each cell defines parameters for conditions/consequences. If a cell doesn't have a value, that condition/action is ignored. Some rows in the spreadsheet are grouped and hidden (see the two plus (**+**) signs in the left). This makes the decision tables more user-friendly, especially for business users. Please note that tables don't have to start on the first column. The full `validation.xls` file is as follows:

	A	B	C	D
1	**RuleSet**	droolsbook.decisiontables.validation		
2	**Import**	droolsbook.bank.model.*, droolsbook.bank.service.*, function droolsbook.bank.service.ValidationHelper.error, function droolsbook.bank.service.ValidationHelper.warning,		
3	**Variables**	ValidationReport validationReport, ReportFactory reportFactory, BankingInquiryService inquiryService		
4	**Notes**	Decision tables for customer validation		
5				
6	**RuleTable Customer validation**			
7	NAME	CONDITION	CONDITION	ACTION
8		$customer : Customer		
9		$param == null	$param == null || == ""	$param(kcontext);
10	**name**	**has no**	**has blank or no**	**report**
11	addressRequired	address		warning
12	phoneNumberRequired		phoneNumber	error
13		dateOfBirth		error

Every file for defining decision tables start with a global configuration section. The configuration consists of name-value pairs. As can be seen from the screenshot above:

- `RuleSet` defines the package
- `Import` specifies the classes used, including static imported functions
- `Variables` is used for global variables
- `Notes` can be any text

Further:

- `Functions` can be used to write local functions as in `.drl` format
- `Worksheet` specifies the sheet to be used; by default only the first sheet is checked for rules

The `RuleTable` then denotes the start of the decision table. It has no specific purpose. It is used only to group rules that operate on the same objects and share conditions. The next line defines column types. The following column types are available:

- `CONDITION`—defines a single rule condition or constraint, the following row can contain type for this condition, if it doesn't, then the next row must define full condition (with a type, not just a constraint as in the preceding case).
- `ACTION`—rule action. Similar to condition, the next line can contain any global or bound variable. Drools will then assume that the next line is a method that should be called on this global or bound variable.
- `PRIORITY`—for defining rule salience.
- `NAME`—by default rule names are auto generated, `NAME` can be used to explicitly specify the name.
- `No-loop` or `Unloop`—specifies the rule no-loop attribute.
- `XOR-GROUP`—specifies rule activation-group (this will be discussed in the upcoming section, *Drools Flow*).

For full configuration options, please consult the Drools manual (`http://www.jboss.org/drools/documentation.html`).

The next line from the preceding screenshot looks similar to what we see in a `.drl` file. It is a simple condition that matches any `Customer` object and exposes this object as the `$customer` variable. The only difference is that there are no brackets. They will be added automatically by Drools at parsing time. Please note that this line contains only two columns. The first two columns are **merged** into one column. This is because they operate on the same type (`Customer`). If we don't merge the two columns, they'll match two separate objects (which may or may not be the same instance).

The next line then defines individual constraints (in case of conditions) or code blocks (in case of actions). Special parameters can be used as `$param` or `$1`, `$2`, `$3`, and so on. The first one is used if our constraint needs only one parameter; otherwise, the `$n` format should be used.

The following line (corresponds to line 10 in the preceding screenshot showing a decision table in `validation.xls` file) is for pure informational purposes. It should contain some meaningful description of the column/action so that we don't have to always look at how it is implemented (by expanding/collapsing rows).

Finally, the actual values follow in subsequent rows. Each line represents one rule. For example, the first line gets translated behind the scenes to the following `.drl` rule:

```
#From row number: 11
rule "addressRequired"
   when
      $customer : Customer(address == null)
   then
      warning(kcontext);
end
```

Code listing 29: Generated rule from a decision table.

The `.drl` rule is exactly the same as we've implemented in Chapter 3, *Validation*. We can even reuse the same unit test to test this rule.

Advantages of a decision table

Here are the advantages of a decision table:

- It is easy to read and understand.
- Refactoring is quicker because we just have to change a column definition to change a group of related rules (that is, it is easy to change conditions across group of rules).
- Isolation is similar to DSL; decision tables can hide the rule implementation details.
- Provides separation between the rules and data (they are still in one file but separated).
- Any formatting available in a spreadsheet editor can be applied to present these data in a more readable manner (for example, using a drop-down for a list of values. It can reduce errors from mistyping a value into a cell by allowing only valid values).
- Eclipse Drools plugin can also validate a spreadsheet. This is very useful when writing rules. The **Problems** view in Eclipse shows what exactly is wrong with the generated `.drl` file.

Disadvantages of a decision table

- It can be awkward to debug/write these rules. Sometimes it helps to convert the spreadsheet to a `.drl` file, save this file, and fix it, as we're used to.
- Decision tables shouldn't be used if the rules don't share many conditions. Further, the order of conditions is important. In a decision table, the order of a condition is given by the order of a column. Care should be taken if you want to convert the existing DRL rules into decision tables, as the order of conditions may change (to take advantage of the reuse).
- XLS is a binary format which makes version management more difficult.

Calculating the interest rate

As an example, we'll calculate the interest rate based on the account `balance`, `currency`, `duration`, and `type`. This calculation is ideal for a decision table because we have a lot of constraints that are reused across rules with different data. The decision table looks as follows:

	A	B	C	D	E
1	**RuleSet**	droolsbook.decisiontables			
2	**Import**	droolsbook.decisiontables.bank.model.*, droolsbook.bank.model.Customer, droolsbook.bank.model.Account.Type, java.math.*,			
3	**Notes**	Decision tables for calculating interest rates.			
4					
5	RuleTable Interest Calculation				
6	CONDITION	CONDITION	CONDITION	CONDITION	ACTION
7	$a:Account				
8	type == Account.Type. $param	currency	balance >= $1 && < $2	monthsBetweenStartAndEndDate >= $1 && < $2	$a.setInterestRate(new BigDecimal($param));
9	**type**	**currency**	**balance <min, max)**	**months**	**set interest rate**
10	TRANSACTIONAL	EUR			"0.01"
11	STUDENT	EUR	0, 2000		"1.00"
12	SAVINGS	EUR	0, 100	0, 1	"0.00"
13	SAVINGS	EUR	0, 100	1, 3	"0.10"
14	SAVINGS	EUR	0, 100	3, 12	"2.00"
15	SAVINGS	EUR	100, 1000	0, 1	"0.10"
16	SAVINGS	EUR	100, 1000	1, 3	"3.00"
17	SAVINGS	EUR	100, 1000	3, 12	"3.25"
18	SAVINGS	EUR	1000, 5000	0, 1	"0.10"
19	SAVINGS	EUR	1000, 5000	1, 3	"3.25"
20	SAVINGS	EUR	1000, 5000	3, 12	"3.50"
21	SAVINGS	EUR	5000, 10000	0, 1	"0.10"
22	SAVINGS	EUR	5000, 10000	1, 3	"3.50"
23	SAVINGS	EUR	5000, 10000	3, 12	"3.75"
24	SAVINGS	USD	0, 100	0, 1	"0.00"

Please note that the `Account` object is used in every condition so the CONDITION columns are merged. We can see the use of parameters `$1` and `$2`. The first line can be read as: *For every transactional account with currency EUR, set its interest rate to 0.01 percent (regardless of the balance)*. Another line can be read as: *For every savings account whose balance is between 100 EUR and 1000 EUR that is opened for one to three months, set its interest rate to 3 percent*. The following rule will be generated:

```
#From row number: 16
rule "Interest Calculation_16"
   when
      $a:Account(type == Account.Type.SAVINGS,
         currency == "EUR", balance >= 100 && < 1000,
         monthsBetweenStartAndEndDate >= 1 && < 3)
   then
      $a.setInterestRate(new BigDecimal("3.00"));
end
```

Code listing 30: Generated rule for calculating the interest rate.

If we had not used a decision table, we would have to write such rules by hand. Please note that the second condition column in the decision table above doesn't have any operator or operand. It simply says `currency`. It is a special feature and this is automatically translated to `currency == $param`.

The last condition column uses `getMonthsBetweenStartAndEndDate` method of the `Account` class.

```
private DateMidnight startDate;
private DateMidnight endDate;

/**
 * @return number of months between start and end date
 */
public int getMonthsBetweenStartAndEndDate() {
  if (startDate == null || endDate == null) {
    return 0;
  }
  return Months.monthsBetween(startDate, endDate)
      .getMonths();
}
```

Code listing 31: Implementation of `getMonthsBetweenStartAndEndDate` method of `Account`.

The implementation uses the Joda-Time library to do the calculation.

Project setup

The following libraries are needed on the classpath:

- `drools-decisiontables-5.0.1.jar`: used for compiling spreadsheets into `.drl` file format; it knows how to handle `.xls` and `.csv` formats.
- `jxl-2.4.2.jar` XLS API: used for parsing `.xls` spreadsheets.

Testing

For testing the interest calculation rules, we'll use a stateless knowledge session, an account, and a date object. All tests will reuse the stateless session. The test can be set up as follows:

```
static StatelessKnowledgeSession session;
Account account;
static DateMidnight DATE;

@BeforeClass
public static void setUpClass() throws Exception {
  KnowledgeBase knowledgeBase =
    createKnowledgeBaseFromSpreadsheet();
  session = knowledgeBase.newStatelessKnowledgeSession();
  DATE = new DateMidnight(2008, 1, 1);
}

@Before
public void setUp() throws Exception {
  account = new Account();
}
```

Code listing 32: Setup of the decision table test.

The date will be used to set deposit durations. An account is created for every test method. The `createKnowledgeBaseFromSpreadsheet` method is implemented as follows:

```
private static KnowledgeBase createKnowledgeBaseFromSpreadsheet()
    throws Exception {
  DecisionTableConfiguration dtconf =KnowledgeBuilderFactory
    .newDecisionTableConfiguration();
  dtconf.setInputType( DecisionTableInputType.XLS );
  //dtconf.setInputType( DecisionTableInputType.CSV );
  KnowledgeBuilder knowledgeBuilder =
    KnowledgeBuilderFactory.newKnowledgeBuilder();
  knowledgeBuilder.add(ResourceFactory.newClassPathResource(
```

```
        "interest calculation.xls"), ResourceType.DTABLE,
        dtconf);
    //knowledgeBuilder.add(ResourceFactory
    //  .newClassPathResource("interest calculation.csv"),
    //  ResourceType.DTABLE, dtconf);
    if (knowledgeBuilder.hasErrors()) {
      throw new RuntimeException(knowledgeBuilder.getErrors()
          .toString());
    }
    KnowledgeBase knowledgeBase = KnowledgeBaseFactory
      .newKnowledgeBase();
    knowledgeBase.addKnowledgePackages(
        knowledgeBuilder.getKnowledgePackages());
    return knowledgeBase;
  }
```

Code listing 33: Creating a `knowledgeBase` from a spreadsheet.

As opposed to the other knowledge definitions, the decision table needs a special configuration that is encapsulated in `DecisionTableConfiguration` class. This configuration specifies the type of decision table and it is then passed on to the knowledge builder. The commented lines show how to create a knowledge base from a `.csv` format. The rest should be familiar.

Note if you want to see the generated `.drl` source, you can get it like this:

```
    String drlString = DecisionTableFactory
        .loadFromInputStream(ResourceFactory
            .newClassPathResource("interest calculation.xls")
            .getInputStream(), dtconf);
```

Code listing 34: Getting the `.drl` representation of the decision table.

It is stored in a `drlString` string variable; it can be printed to the console and used for debugging purposes.

We'll now write a test for depositing 125 EUR for 40 days:

```
@Test
public void deposit125EURfor40Days() throws Exception {
  account.setType(Account.Type.SAVINGS);
  account.setBalance(new BigDecimal("125.00"));
  account.setCurrency("EUR");
  account.setStartDate(DATE.minusDays(40));
  account.setEndDate(DATE);
```

```
    session.execute(account);
    assertEquals(new BigDecimal("3.00"), account
        .getInterestRate());
}
```

Code listing 35: Test for depositing `125` EUR for `40` days.

The preceding test verifies that the correct interest rate is set on the `account`.

And one test for default transactional account rate:

```
@Test
public void defaultTransactionalRate() throws Exception {
  account.setType(Account.Type.TRANSACTIONAL);
  account.setCurrency("EUR");
  session.execute(account);
  assertEquals(new BigDecimal("0.01"), account
      .getInterestRate());
}
```

Code listing 36: Test for the default transactional account rate.

The test above, again, verifies that the correct interest rate is set.

Comma Separated values

The XLS spreadsheet can be easily converted into CSV format. Just select **Save as CSV** in your spreadsheet editor. However, there is one caveat—CSV format doesn't support merging of columns by default. For overcoming this, Drools has the following workaround: if we add three dots at the end of type declarations, they will be merged into one. It can be seen in the last line of the following CSV excerpt:

```
"RuleTable Interest Calculation",,,,
"CONDITION","CONDITION","CONDITION","CONDITION","ACTION"
"$a:Account...","$a:Account...","$a:Account...","$a:Account...",
```

Code listing 37: Excerpt from the `interest calculation.csv` file.

It is the only change that needs to be done. Tests should pass for CSV format as well.

CSV is a text format as opposed to XLS, which is a binary format. Binary format makes version management harder. For example, it is very difficult to merge changes between two binary files. CSV doesn't have these problems. On the other hand, the presentation suffers.

Rule Templates

If you like the concept of decision tables, you may want to look at **Drools Rule Templates**. They are similar to the decision tables but more powerful. With Rule Templates, the data is fully separated from the rule (for example, it can come from a database and have different templates over the same data). You have more power in defining the resulting rule. The data can define any part of rule (for example, condition operator, class, or property name). For more information, refer to the *Rule Templates* section of the *Drools Experts User Guide* (available at `http://www.jboss.org/drools/documentation.html`).

Drools Flow

Drools Flow (or **ruleflow** in short) is another way we can have human readable rules. It is not a substitute to rules as was the case with DSLs and decision tables. It is a way of defining the execution flow between complex rules. The rules are then easier to understand.

Drools Flow can *externalize the execution order from the rules*. The execution order can be then managed externally. Potentially, you may define more execution orders for one `KnowledgeBase`.

Drools Flow can be even used as a workflow engine replacement. It can execute arbitrary actions or user-defined work items at specific points within the flow. It can be even persisted as we'll see in Chapter 8, *Drools Flow*, which shows a bigger example of using ruleflows.

Drools Agenda

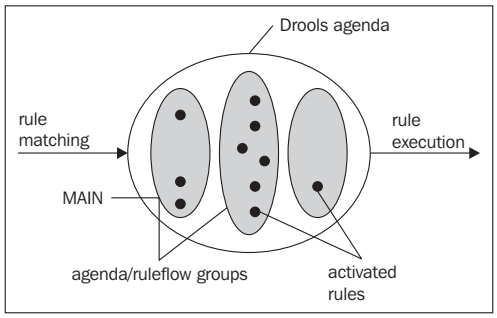
Before we talk about how to manage rule execution order, we have to understand Drools Agenda. When an object is inserted into the knowledge session, Drools tries to match this object with all of the possible rules. If a rule has all of its conditions met, its consequence can be executed. We say that a rule is **activated**. Drools records this event by placing this rule onto its agenda (it is a collection of activated rules). As you may imagine, many rules can be activated, and also deactivated, depending on what objects are in the rule session. After the `fireAllRules` method call, Drools *picks* one rule from the agenda and executes its consequence. It may or may not cause further activations or deactivations. This continues until the Drools Agenda is empty.

The purpose of the agenda is to manage the execution order of rules.

Methods for managing rule execution order

The following are the methods for managing the rule execution order (from the user's perspective). They can be viewed as alternatives to ruleflow. All of them are defined as rule attributes.

- `salience`: This is the most basic one. Every rule has a salience value. By default it is set to `0`. Rules with higher salience value will fire first. The problem with this approach is that it is hard to maintain. If we want to add new rule with some priority, we may have to shift the priorities of existing rules. It is often hard to figure out why a rule has certain salience, so we have to comment every salience value. It creates an invisible dependency on other rules.
- `activation-group`: This used to be called **xor-group**. When two or more rules with the same activation group are on the agenda, Drools will fire just one of them.
- `agenda-group`: Every rule has an agenda group. By default it is `MAIN`. However, it can be overridden. This allows us to partition Drools Agenda into multiple groups that can be executed separately.



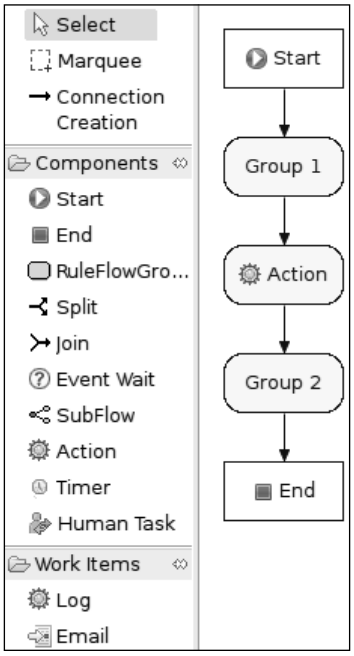
The figure above shows partitioned Agenda with activated rules. The matched rules are coming from left and going into Agenda. One rule is chosen from the Agenda at a time and then executed/fired.

At runtime we can programmatically set the active Agenda group (through the `getAgenda().getAgendaGroup(String agendaGroup).setFocus()` method of `KnowledgeRuntime`), or declaratively, by setting the rule attribute `auto-focus` to `true`. When a rule is activated and has this attribute set to `true`, the active agenda group is automatically changed to rule's agenda group. Drools maintains a **stack** of agenda groups. Whenever the focus is set to a different agenda group, Drools adds this group onto this stack. When there are no rules to fire in the current agenda group, Drools pops from the stack and sets the agenda group to the next one. Agenda groups are similar to ruleflow groups with the exception that ruleflow groups are not stacked.

Note that only one instance of each of these attributes is allowed per rule (for example, a rule can only be in one `ruleflow-group`; however, it can also define a `salience` within that group).

Ruleflow

As we've already said, ruleflow can externalize the execution order from the rule definitions. Rules just define a `ruleflow-group` attribute, which is similar to `agenda-group`. It is then used to define the execution order. A simple ruleflow (in the `example.rf` file) is shown in the following screenshot:



The preceding screenshot shows a ruleflow opened with the Drools Eclipse plugin. On the lefthand side are the components that can be used when building a ruleflow. On the righthand side is the ruleflow itself. It has a **Start** node which goes to ruleflow group called **Group 1**. After it finishes execution, an **Action** is executed, then the flow continues to another ruleflow group called **Group 2**, and finally it finishes at an **End** node.

Ruleflow definitions are stored in a file with the `.rf` extension. This file has an XML format and defines the structure and layout for presentational purposes.

Another useful rule attribute for managing which rules can be activated is `lock-on-active`. It is a special form of the `no-loop` attribute. It can be used in combination with `ruleflow-group` or `agenda-group`. If it is set to `true`, and an agenda/ruleflow group becomes active/focused, it discards any further activations for the rule until a different group becomes active. Please note that activations that are already on the agenda will be fired.

A ruleflow consists of various nodes. Each node has a name, type, and other specific attributes. You can see and change these attributes by opening the standard **Properties** view in Eclipse while editing the ruleflow file. The basic node types are as follows:

- **Start**
- **End**
- **Action**
- **RuleFlowGroup**
- **Split**
- **Join**

They are discussed in the following sections.

Start

It is the initial node. The flow begins here. Each ruleflow needs one start node. This node has no incoming connection—just one outgoing connection.

End

It is a terminal node. When execution reaches this node, the whole ruleflow is terminated (all of the active nodes are canceled). This node has one incoming connection and no outgoing connections.

Action

Used to execute some arbitrary block of code. It is similar to the rule consequence—it can reference global variables and can specify dialect.

RuleFlowGroup

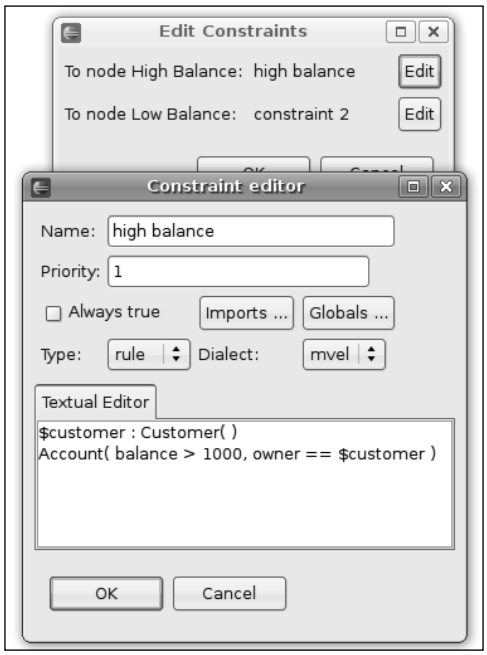
This node will activate a ruleflow-group, as specified by its `RuleFlowGroup` attribute. It should match the value in `ruleflow-group` rule attribute.

Split

This node splits the execution flow into one or many branches. It has two properties—name and type. Name is just for display purposes. Type can have three values: **AND**, **OR**, and **XOR**:

- **AND**: The execution continues through all of the branches.
- **OR**: Each branch has a condition. The condition is basically same as a rule condition. If the condition is true, the ruleflow continues through this branch. There must be at least one condition that is true; otherwise, an exception will be thrown.
- **XOR**: Similar to OR type, each branch has a condition, but in this case, with a priority. The ruleflow continues through just **one** branch, whose condition is true and it has the lowest value in the priority field. There must be at least one condition that is true; otherwise, an exception will be thrown.

The dialog for defining OR and XOR split types looks like the following screenshot:



The screenshot above shows Drools Eclipse plugin ruleflow constraint editor. It is accessible from the standard Eclipse **Properties** view.

Join

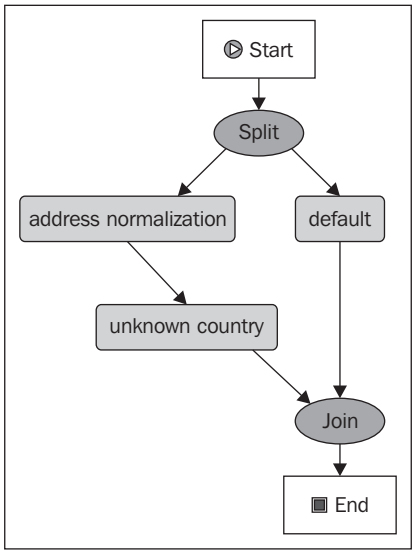
It joins multiple branches into one. It has two properties—name and a type. Name is for display purposes. Type decides when the execution will continue. It can have following values:

- **AND**: Join waits for all the incoming branches. The execution then continues.
- **XOR**: Join node waits for one incoming branch.

Please consult the Drools manual for further node types.

Example

If you look at data transformation rule in Chapter 4, *Data Transformation*, you'll see that in some rules we've used `salience` to define rule execution order. For example, all of the addresses needed to be normalized (that is, converted to `enum`) before we could report the unknown countries. The unknown country rule used salience of `-10`, which meant that it would fire only after all address normalization rules. We'll now extract this execution order logic into a ruleflow to demonstrate how it works. The ruleflow might look like the following screenshot:



When the execution starts, it goes through the **Start** node straight into the **Split** node. In this case, it is an *and* type split node. It basically creates two parallel branches that will be executed concurrently (note that this doesn't mean multiple threads). We can see that the flow is explicitly specified. **address normalization** happens before **unknown country** reporting. Parallel to this branch is a **default** ruleflow group. It contains the other rules. Finally, a join node of type *and* is used to block until all branches complete and then the flow continues to the terminal node. We had to use the **Join** node (instead of going straight to the **End** node), because as soon as some branch in a ruleflow reaches the **End** node, it terminates the whole ruleflow (that is, our branches may be canceled before competition, which is not what we want).

The process ID is set to `dataTransformation`. Click on the canvas in the ruleflow editor and then in the **Properties** view (Eclipse Properties plugin), set the ID to this value.

Rules

Next we create copy of `dataTransformation.drl` file from Chapter 4 and we'll name it `dataTransformation-ruleflow.drl`. We'll make the following changes:

- Each rule gets new attribute: `ruleflow-group "default"`
- Except the address normalization rules for example:

```
rule addressNormalizationUSA
ruleflow-group "address normalization"
```

  Code listing 37: Top part of the USA address normalization rule.

- Unknown country rule gets the "`unknown country`" ruleflow group.

KnowledgeBase setup

We can now create a knowledge base out of the ruleflow file and the `.drl` file.

```
static KnowledgeBase createKnowledgeBaseFromRuleFlow()
    throws Exception {
  KnowledgeBuilder builder = KnowledgeBuilderFactory
    .newKnowledgeBuilder();
  builder.add(ResourceFactory.newClassPathResource(
      "dataTransformation-ruleflow.drl"), ResourceType.DRL);
  builder.add(ResourceFactory.newClassPathResource(
      "dataTransformation.rf"), ResourceType.DRF);
  if (builder.hasErrors()) {
    throw new RuntimeException(builder.getErrors()
        .toString());
  }
```

```
    KnowledgeBase knowledgeBase = KnowledgeBaseFactory
        .newKnowledgeBase();
    knowledgeBase.addKnowledgePackages(builder
        .getKnowledgePackages());
    return knowledgeBase;
}
```

Code listing 38: Method that creates `KnoweldgeBase` with a ruleflow.

A knowledge base is created from both files `.drl` and `.rf`. To achieve true isolation of unit tests, consider constructing the knowledge base only from the `.drl` file or `.rf` file. That way, the unit tests can focus only on the relevant part.

Tests

The test setup needs to be changed as well. Ruleflows are fully supported only for stateful sessions. Stateful sessions can't be shared across tests because they maintain state. We need to create a new stateful session for each test. We'll move the session initialization logic from the `setUpClass` method that is called once per test class into the `intialize` method that will be called once per test method:

```
static KnowledgeBase knowledgeBase;
StatefulKnowledgeSession session;

@BeforeClass
public static void setUpClass() throws Exception {
  knowledgeBase = createKnowledgeBaseFromRuleFlow();
}

@Before
public void initialize() throws Exception {
  session = knowledgeBase.newStatefulKnowledgeSession();
```

Code listing 39: Excerpt from the unit test initialization.

Once the stateful session is initialized, we can use it.

We'll write a test that will create a new address map with an unknown country. This address map will be inserted into the session. We'll start the ruleflow and execute all of the rules. The test will verify that the `unknownCountry` rule has been fired:

```
@Test
public void unknownCountryUnknown() throws Exception {
  Map addressMap = new HashMap();
  addressMap.put("_type_", "Address");
```

```
    addressMap.put("country", "no country");
    session.insert(addressMap);
    session.startProcess("dataTransformation");
    session.fireAllRules();
    assertTrue(validationReport.contains("unknownCountry"));
}
```

Code listing 40: Test for the unknown country rule with an unknown country.

Note that the order of the three `session` methods is important. All of the facts need to be in the session before the ruleflow can be started and rules can be executed.

Please note that in order to test this scenario, we didn't use any agenda filter. This test is more like an integration test where we need to test more rules cooperating together.

Another test verifies that the ruleflow works with a known country:

```
@Test
public void unknownCountryKnown() throws Exception {
    Map addressMap = new HashMap();
    addressMap.put("_type_", "Address");
    addressMap.put("country", "Ireland");
    session.startProcess("dataTransformation");
    session.insert(addressMap);
    session.fireAllRules();
    assertFalse(validationReport.contains("unknownCountry"));
}
```

Code listing 41: Test for the unknown country rule with a known country.

As a stateful session is being used, every test should call the `dispose` method on the session after it finishes. It can be done in the following manner:

```
@After
public void terminate() {
    session.dispose();
}
```

Code listing 42: Calling the `session.dispose` method after every test.

Summary

In this chapter we've learned about writing more user-friendly rules using DSLs, decision tables, and ruleflows. You can mix and match these approaches. It makes sense to write some rules using DSL, some using decision tables, and more complex rules using pure `.drl` file format. `KnowledgeBase` can be created from multiple sources.

DSLs are very useful if there is a need for the business analyst to read and understand the existing rules and even write new rules. The resulting language uses businesses terminologies making it more natural for the business analyst. DSL provides an abstraction layer that hides complicated rule implementations. The Eclipse editor brings auto competition so that the rules are easier to write.

Decision tables, on the other hand, are useful when we have a lot of similar rules that use different values as was the case in the interest rate calculation example. It makes it easy to change such rules because the rule implementation is decoupled from the values they use. Spreadsheet format is also more concise. We can fit more rules into one screen, which makes it easier to understand the overall picture.

In the last section, we've learned about Drools Flow, Agenda, and various ways of managing rule execution order. Drools Flow managed the execution order in a nice human-readable graphical representation.

Where to buy this book

You can buy Drools JBoss Rules 5.0 Developer's Guide from the Packt Publishing website: `http://www.packtpub.com/drools-jboss-rules-5-0-developers-guide/book`

Free shipping to the US, UK, Europe and selected Asian countries. For more information, please read our shipping policy.

Alternatively, you can buy the book from Amazon, BN.com, Computer Manuals and most internet book retailers.

www.PacktPub.com

For More Information:
www.packtpub.com/drools-jboss-rules-5-0-developers-guide/book