# Domain Specific Language (DSL) and RuleFlow

People new to rule engines are often confused by two things: the syntax of the rules, and the fact that the rule engine (and not you) decides the order in which your rules are fired. This chapter will show some ways of making both of these things easy to understand.

In the previous chapter, we saw how to use Excel decision tables to make it easier to write rules. This chapter takes off from the 'easy to write rules' theme. Although writing rules in Excel is good, wouldn't it be much better to write them in a language that is closer to English, or whatever human language you prefer? **Domain Specific Language** (**DSL**) gives you this option.

The other point of confusion is the order in which rules are fired. Wouldn't it be great to draw a workflow diagram to see and control which (groups of) rules should fire, and when? Ruleflow gives us this control.

We'll come to Ruleflow shortly, but first we'll look at how to use DSLs.

**What is a Domain Specific Language (DSL)?**

Every profession has its own language, or what is called a jargon. Although, on the surface a jargon may appear to be related to English, it's often incomprehensible to outsiders. If you've watched ER, Grey's Anatomy, House, or any of the other medical dramas on TV, you know what I mean. You will understand this better with the help of the following example:

Doctor to nurse:

Give me sterilised scalpel number 4, spreaders—the patient's cardioangiogram is showing traces of acute defibrillation.

In plain English, this means:

*Give me a knife now, this guy's having a heart attack.*

One of the reasons why each profession, or domain, has its own specific language (or DSL) is that it conveys information much more precisely and concisely. A more cynical view would be that domain terminology allows professionals to baffle clients and charge more. But obviously, nobody in the IT industry would ever take advantage of that fact!

The fact is that business and other professionals speak one language. Our rules, even though they are 'plainer English', are written (so far) in another language. If we want to capture the business knowledge from the professionals, we need to speak the language of the business users. This is where DSLs provided by Drools come in.

Imagine that we could write our rules in a form similar to the extract that follows. This extract is pretty easy to understand without any technical knowledge. This would be a good thing, because more people would be able to review and maintain the rules.

**rule "Check Patient for Heart Attack"**

**when**

**There is a Patient**

* **appears not to be breathing**
* **has no pulse**
* **is white or blue in the face then**

**Call for Assistance**

**Start CPR**

**end**

You've been reading this book long enough now to know that the next line after 'imagine … ' is usually 'but of course you can!'. And **expanders** are a key part of how we do it.

## Expanders

Something needs to convert the near-English rules (like the previous sample) into the more formal or standard rule (DRL) format that we're used to seeing. That's the Drools expander. The concept is simple:

1. We start with the near-English business rules. They are saved in a text file with a **.dslr** extension, to make it clearer to us what the file contains.
2. The **rulefile** contains a statement similar to **expander** **chocolate-trading. dsl**. So Drools knows how to translate the near-English file into our standard rule file format.
3. Using the DSL file, Drools finds the English text and replaces it with a more technical rule language.
4. Once this find-and-replace process is complete, we will have a file (in memory, not on disk) containing our standard rule language. It is similar to the DRL rules that we saw in earlier chapters.
5. Drools can then execute the rules in this DRL as before.

The actual process is even more powerful than find and replace, given that we can use regular expressions (also known as regex) and wildcards. For example, **h\?t** would match (then replace) all three-letter words beginning with "h" and ending with "t"-for example, "hat", "het", "hit", "hot", "hut", as well as the more nonsensical words with endings such as "hbt", "hct", and so on.

How does Drools know which expander or DSL file to use? Our (near-English) rules file tells Drools which file is required, using a statement such as **expander** **my-dsl-file-name** near the top of the rules (**.dslr)** file.

The process of taking a more readable file, converting it to a rules file, and then applying those rules as appropriate, may seem familiar. It's the same sort of process used in the Excel-based decision tables from the previous chapter. Like decision tables, DSLs can automatically convert a human-friendly format into something more Drools-like, and then fire the generated business rules as appropriate.

If we had a medical DSL in place, this find-and-replace process would result in the following business rule (in Drools syntax):

**rule "Check Patient for Heart Attack"**

**when**

**$patient : Patient( breathing == false , pulse == false**

**(face == blue or face == white)**

**)**

**then**

**$hospital.callForAssistance();**

**$patient.startCPR();**

**end**

### The DSL format

What is the format of the DSL that will convert the 'medical' rule to a Drools rule, and vice versa? The format looks something like the following plain-text file, so that you can view it with almost any editor. The file will have a **.dsl** extension.

**# Match against the when part of our medical rules**

**[when]There is a Patient = $patient : Patient()**

**[when]-has no pulse = pulse==false**

**[when]-appears not to be breathing = breathing== false**

**[when]-is {color1} or {color2} in the face = (face == {color1} or face**

**== {color2})**

**# Our possible medical actions in the 'when' part**

**[then]Call for Assistance= $hospital.callForAssitence();**

**[then]Start CPR=$patient.startCPR();**

The format allows a pretty simple find-and-replace process, using the following parts:

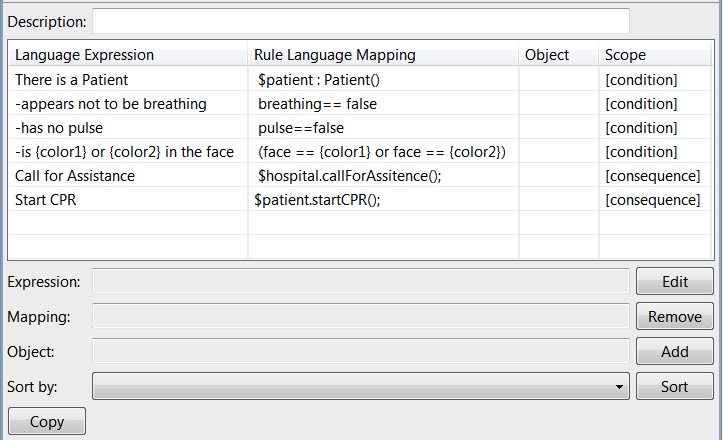
1. Comments (lines beginning with **#**) are ignored, although they are useful for explaining what is going on. Similarly, whitespace and blank lines are also ignored, but laying out the file cleanly using whitespace makes it a lot more readable.
2. The basic format of the file is **[when]** or **[then]** (something to find = something to replace). The **[when]** or **[then]** part means that we will only match against the left (when) or right (then) part of the rule, as appropriate.
3. For example, **[when]** **There** **is** **a** **Patient** **=** **$patient:** **Patient()** means, find the text **There** **is** **a** **patient** in the **when** part of our rule, and replace it with the more Drools-like language **$patent** **:** **Patient()** (that is, match against all patients in working memory).
4. If the next line starts with a '-' (for example, **-has** **no** **pulse** **=** **pulse==false**), then Drools is smart enough to add a filter to our rule in case of a match in the original file. This means that we end up with **$patient** **:** **Patient(pulse==false**), which will match against all patients with no pulse. This allows us to mix and match the conditions on our rules without having to specify every possible combination in our DRL translation file.
5. The **[when] find {value} = replace {value}** allows you to specify a value in your original rule, which will get passed to the final translation.

For example, **-is** **{color1}** **or** **{color2}** **in** **the** **face** means we can reuse this DRL translation for other colors (specified in the 'English' **.dslr** rules file) later.

1. The find-and-replace mechanism works in a similar way for translating the **[then]** part of the rules from 'near-English' to the more technical Drools language.

### Other DSL editing options

The DSL is just a text file, so we can edit it in Notepad. There is another option for editing a DSL file in the IDE—the guided editor. If you open a DSL file in the JBoss IDE, you'll see something similar to the following screenshot:



* There are three main sections to the editor: a place for the DRL **Description** at the top, a table showing existing entries in the DSL, and a space at the bottom to add new entries. The latter space also has buttons to **Edit**, **Remove**, **Add**, or **Copy** entries, as well as an option to **Sort** the existing ones.
* **Language Expression** is what we will **Find** in the original rules file. **Rule**

**Language Mapping** is something we will **Replace** it with to generate our Drools technical file. This find-and-replace mechanism is similar to the one we looked at in the text editor.

* We also have **Scope** (**[condition]** or **[consequence]**), which is just another name for **[when]** or **[then]**, which we looked at previously.
* What is new is the **Object** field. This object takes the full name of a JavaBean, including the package (for example **net.firstpartners.Patient**). This allows autocompletion (that is, pop-up suggestions) when we are writing the rules that use this DSL in Guvnor or the IDE.

The IDE editor saves the DRL file in the text format that we looked at earlier. So, we can switch back and forward between the two editors as required.

### Writing DSLs

Imagine that you had to write your own language, from the very beginning. Where would you start? You could try using an existing dictionary as your inspiration, but you're likely to map too many words that you don't really need. Or, you could make it up as you go along, but users would get frustrated with the many delays while you add yet another word to your language. Remember that the ideal is to have a more-or-less stable language that users can easily understand so that it is easier for them to write their rules. So, what are you going to do?

The answer is iteration. You (as a 'knowledge engineer' who understand rules and

DSLs) write the first 10, 20, or 100 rules, adding the elements that you need to the DSL, as you go. Over time, you'll find that your DSL becomes more and more stable, needing fewer additions to cope with each additional rule. At this point, you can hand over the DSL to your business users (with much mentoring and training) and the users can start writing their own rules based on the (near-stable) DSL.

Of course, picking a representative sample of rules will make writing the initial DSL much more effective. Staying around to pick up the occasional mapping that your rule writers need to add, will make them a lot happier. It also helps to have tests written against your rules (as described in Chapter 7) so that you know instantly if any of the changes that you are making breaks anything.

Let's look at this process, using a sample that we saw in the last chapter.

As there is a lot of crossover and code reuse, the samples for this chapter can be downloaded from the Chapter 8 sample at **http://code. google.com/p/red-piranha**.

### Meet the sample

Do you remember the rule in the DRL file that we had in the previous chapter—the one that logged the contents of Excel cells? At the start of **log-rules.dsl**, it looked something like the following:

**package net.firstpartners.chap8; import net.firstpartners.drools.log.ILogger import net.firstpartners.excel.Cell; import net.firstpartners.excel.Range; global ILogger log; rule "print cell initial values"**

**when**

**$cell : Cell(modified==false) then**

**//Logging message**

**log.info("initial cell value:"+$cell); end**

Our first step is to change the file name to **log-rules.dslr**, and create a blank DSL file called **cell-logging.dsl**. We link the two by adding an expander statement to the DSLR file (**expander cell-logging.dsl**).

Once we add the expander statement, we've got a problem: Drools expects to match everything between **when** and **end** against what it finds in the DSL. The problem is that the DSL is (for the moment) just a blank file. The lines are marked with '>' so that they are taken as they are. Our **.dslr** file will now look as follows:

**package net.firstpartners.chap8; import net.firstpartners.drools.log.ILogger import net.firstpartners.excel.Cell; import net.firstpartners.excel.Range; expander cell-logging.dsl global ILogger log; rule "print cell initial values"**

**when**

**> $cell : Cell(modified==false) then**

**//Logging message**

**> log.info("initial cell value:"+$cell); end**

For this iteration, we know that we need to put in the entries for the two lines marked with '>'. Starting with the 'then' part, we add the following lines to our DSL:

**#Cell Selection Rules**

**[when]There is a Cell = $cell: Cell()**

**[when]-unmodified = modified==false**

**[when]-modified = modified==true**

To cover the second-to-last line (**log.info**) we add some more lines to our DSL file:

**# Logging rules**

**[then]Log the cell contents = log.info("Cell value:"+$cell);**

Now that we have a DSL ready (at least for this iteration), we can update our rule file with more English-language rules. Because we've also changed the package declaration inline with this chapter's name, the sample now looks like this:

**package net.firstpartners.chap9; import net.firstpartners.drools.log.ILogger import net.firstpartners.excel.Cell; import net.firstpartners.excel.Range; global ILogger log;**

**# We must reference the dsl that we are using expander cell-logging.dsl rule "print cell initial values"**

**when**

**There is a Cell - unmodified**

**then**

**Log the cell contents**

**End**

In real life, once we were happy that this was working, we would carry out some more iterations. We would do this by covering other methods available on the Cell Object, and perhaps the Range Object as well.

## Running the DSL example

This example (with simple cell logging) is ready to run, as part of the example that you downloaded in the previous chapter. The start point is the **DslChocolateTradingExample** Java file found in the **net/firstpartners/chap9** folder. If we run it (right-click on the file, and then select **Run as | Java application** from the shortcut menu), we'll see all of the following values from the Excel input file being logged:

**ExcelLogger - Cell value:cellName:Z\_Broker\_4 value:n/a comment:null modified:false**

**ExcelLogger - Cell value:cellName:Z\_Broker\_3 value:500.0 comment:null modified:false**

**ExcelLogger - Cell value:cellName:Z\_Broker\_2 value:300.0 comment:null modified:false**

**ExcelLogger - Cell value:cellName:Z\_Broker\_1 value:false comment:null modified:false**

Note that in the **DslChocolateTradingExample**, we specify (again) the DSL file that we are using.

**new RuleRunner().runStatelessRules(RULES\_FILES,DSL\_FILE, ranges.**

**getAllRangesAndCells(), globals, excelLogger);**

You'll notice that in the log file, both the rule file and the DSL file are loaded.

**RuleRunner - found rule file:log-rules.dslr**

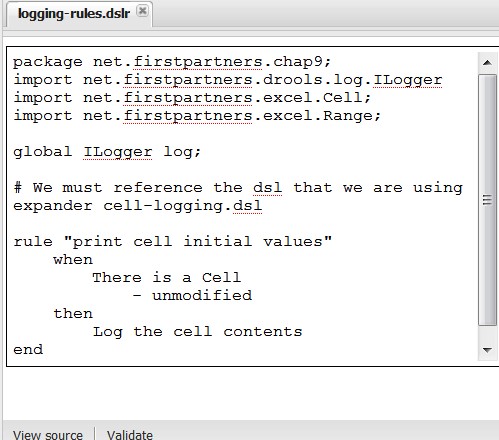
**RuleRunner - found dsl file:cell-logging.dsl**

### Guvnor and DSL-based rules

Domain-specific languages aim to make business users' life easier by allowing them to write rules in a format they're familiar with. On the other hand, Guvnor is a web editor intended for business users editing rules. What could be more perfect than combining the two?

From what we already know, it's pretty simple to start writing DSL-based rules in the Guvnor Web editor.

1. Open up the Guvnor web editor, as you did in Chapter 3.
2. Create a new package (**net.firstpartners.chap9**) to hold our information, like we did in Chapter 4.
3. Export our fact model that contains the Cell and Range JavaBeans from Eclipse (the JBoss IDE) and import them into Guvnor under the **net. firstpartners.chap9** package (as you did in Chapter 4).

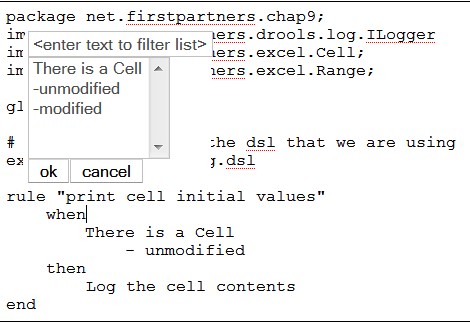


1. Create a new DSL and give it the same name as the one in Eclipse (**Cell. logging.dsl**). Under the **chap9** package, set the category to **anything**. Copy and paste the contents of our Eclipse-based DSL file into this package, and save the package. You can create a new DSL under the same menu as **Create a new rule**.
2. From the same menu (and in the same package), create a new DSL business rule (using the text editor) and name it **logging-rules.dslr**. Copy and paste the values from the Eclipse (the JBoss IDE) file of the same name.

The file in Guvnor will now look similar to the above screenshot. Clicking on the **Validate** button will show the following success message:



While editing your dslr rule, pressing the *Ctrl+Space* keys will enable the autosuggest feature, which is shown in the following screenshot:



So far, in this chapter, we've looked at DSL (Domain Specific Language) to make our rules easier to understand. Ruleflow also makes our rules more understandable. We'll take a look at that, now.

## Ruleflow

One of the key features of a rule engine is that we have no control exactly when our rules will fire. Our rules simply become available to fire (because the 'when' conditions have been filled), and the rule engine decides the best order in which to execute them. We may drop hints about the rules that have higher priority (using Salience), but the sort of 'fine-grained' controls that we have in other languages are (for good reason) not there. This is a good thing as it makes the individual rules simpler, reusable, and easier to understand. However, there are business situations where we may need to group our rules and control when they fire.

For example, for a mortgage application (which we'll call Homeloan) you may have several hundred business rules. These might be naturally grouped by the state of the loan application: rules that fire in the initial enquiry (for example, to provide a quick quote), rules that fire when the application is received (for example, to make sure that all of the paperwork is in order), rules that fire when the loan is drawn down (for example, to ensure that the money is sent out correctly), rules that fire every month (for example, to calculate interest and accept a loan repayment), and rules that fire if the loan goes into arrears or if there is a change in the interest rate.

It would be possible to write our rules to check the status of the application, for example:

**When**

**Mortgage application is in 'Monthly Loan Repayment State'**

**And (some other conditions unique to this business rule)**

**……**

**Then**

**Carry out monthly interest calculations**

This goes on for each and every one of our several hundred rules. The problem with this is that if our business flow changes (for example, if our bank decides that there is an exciting new opportunity in the subprime lending market), then we have to change each and every one of the business rules to reflect this. Next, we test all of the rules to make sure that none of them inadvertently fire at the wrong time.

The alternative is to map our business flow graphically, like the following diagram. At a glance, we can now see what the sequence of rule firing is:

* We start at **Initial Enquiry** and allow those rules to fire.
* If the **Initial Enquiry** is followed up, then we loan the money.
* After passing through a join point, we make a decision based on the question: Are there any loan repayments outstanding? If yes, we give our rules that calculate the loan interest the chance to fire.
* There is a (non) rule action to collect repayment before rejoining the previous point.
* This time, if we don't have any repayments outstanding, we allow the **Complete Loan** rules to fire, before ending the Ruleflow.

InitialEnquiry

LoanMoney

RepaymentsOutstanding?

Calculate Interest

CompleteLoan

Join

End

Start

Collect Repayment

Understanding the Ruleflow in a diagram is far easier than deciphering the information buried in individual rules.

### Ruleflow is not workflow

Notice that in the previous section we didn't say 'and at stage X the rules fire'. What we said was 'at stage X we give the rules a chance to fire'. It's still up to the rule engine to decide which rules are the most appropriate to fire. Your rules are still rules, except that effectively another condition has been added to the 'when' part—checking that the Ruleflow is pointing to the group that our rule is in.

It's worth repeating this statement again. Ruleflow is not workflow. It might look similar (for example, the JBoss jBPM workflow product has similar diagrams), but it is subtly different. Let's look at the differences between the two:

* Workflow says exactly what will happen at each stage in the process. As soon as the workflow reaches a step, we will fire the actions associated with it.
* Ruleflow says what *might happen* at each stage in the process. The rule engine selects the rules that actually fire.

It is possible to set individual rules to fire at each step in the flow (just like traditional workflow). But if you find yourself wanting to do this, then you're really not using a rule engine in the best way. It is better to mark a group of rules (and not a single rule) to fire, and then let Drools do its rule-engine stuff.

If you do need traditional workflow, you've got plenty of choices. From JBoss alone you have **jBPM** (**java** **Business** **Process** **Management**) and Drools (with enhanced workflow features in Drools 5, which we will introduce in Chapter 12).

Both JBoss workflow products integrate well with the rule engine. For example, when we have a decision node (to choose what the next step is) we can use the rule engine to make this decision.

Whatever you do, don't be tempted to write your own workflow—there's no need to do so with hundreds of open source and commercial workflow engines to choose from.

### That Homeloan example again

To create a sample like the Homeloan, right-click on a project in the JBoss IDE and select the **New | other | Ruleflow** menu option. You'll be shown a blank drawing page (where you can draw your Ruleflow) with the following icons available on the lefthand side of the screen. To compose your Ruleflow, select the icons that you want to use, and then use **Connection Creation** to tie them together.

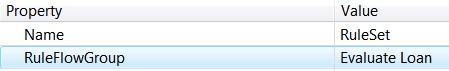
The available steps in our Ruleflow are:

* **Start**: Every flow should have only one start. Naturally enough, this is where our Ruleflow begins.

|  |  |
| --- | --- |
| |  | | --- | | Start | |

* **RuleFlowGroup**: Our **RuleFlowGroup** has a name. When our Ruleflow

reaches this point in the flow, rules belonging to this Ruleflow group (and only those rules belonging to this group) will be given the chance to fire.



* **RuleSet:** If you look at the **Property** tab at the bottom of the screen, you see additional information about each icon. In this case we can see that the display name of the icon is **RuleSet**, and that **RuleFlowGroup** (which we will use shortly to tie our Ruleflow to our rules) is **Evaluate Loan**.
* Remember that all icons, and not just the **RuleSet**, have properties that you can view and change.

RuleSet

* **Action**: In our Homeloan flow we have a step that was more suited to the traditional programming task (**Collect Repayment**). This could be printing a letter, or sending an XML message to another bank, requesting payment. Although this can be done via our rules (as it's calling the standard Java code), it's better to be able to state it as a clear step in our flow.

Action

* **Split**: Only the simplest flows run in a straight line from start to finish. Splits and joins allow the flow to branch and reconnect. As we saw in the

Homeloan sample diagram, splits and joins allow us to create loops in the Ruleflow. Splits can allow the flow to go down one path or another, or even down both paths. The decision about the path is rule-based.

Split

* **Join:** This action allows multiple branches to come back together. We can specify to wait for all branches to come back the **Join**, or just wait for one of them one.

Join

* **Milestone**: This is a wait state—waiting until specified conditions become true. The expressions used to specify the wait state are the same as those on the lefthand side of a rule. The flow will be held until this condition is fulfilled.

Milestone

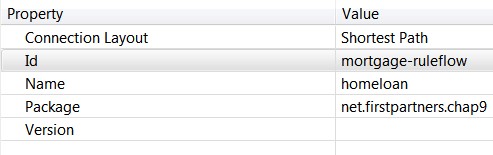
* **SubFlow**: This executes the flow in another Ruleflow file. The name of that other Ruleflow file is specified via the **Properties** tab.

SubFlow

* **End**: Every flow should have only one end point. Naturally enough, this is where our Ruleflow terminates.

|  |  |
| --- | --- |
| |  | | --- | | End | |

It's not just the icons that have properties. If you select the entire Ruleflow (by clicking on a blank area of the diagram), we can set properties for it as well.



In this case, we see that the ID of Ruleflow is **mortgage-ruleflow** and it is a part of the **net.firstpartners.chap9** package. There are also fields to set a version number and to give the Ruleflow a more human-readable name (in this case, **homeloan**).

After drawing your diagram and making connections between the Ruleflow steps (select two workflow steps, right click, then choose create connection from the context menu that appears). It's always useful to validate the diagram (so that you catch any errors now, rather than when you try to execute the flow for the first time). To validate the diagram, right-click anywhere in the whitespace of the Ruleflow diagram and select **Validate** from the context menu. Drools will notify you of any errors that it finds in the Ruleflow.

Just because your Ruleflow validates correctly doesn't necessarily mean that it will run without any errors. There are many things that can still go wrong at runtime. Validation is still a good way of reducing the chances of having errors.

### Linking rules to Ruleflow

So far, we drawn a few Ruleflow diagrams and stepped through the different actions that we can add to the Ruleflow. But how does Drools know which rules are available to fire at each step in the Ruleflow?

The answer is simple. We tag our rules with the name of the Ruleflow group that they belong to. (Remember that we set the Ruleflow group name as a part of the

Ruleflow diagram.) By doing this, we end up with a rule looking something like this:

**rule 'YourRule' ruleflow-group 'evaluate loan'**

**when ... then ... end**

This way, when our flow gets to the **evaluate** **loan** Ruleflow group, we know that we have at least one rule that is available for firing. Whether the rule fires or not will depend on the conditions attached to the **when** part of the rule.

There are other Ruleflow-related attributes that we can add to our rules. These are as follows:

* **agenda-group** is just another term for Ruleflow group.
* **auto-focus** **true** allows the rule to capture the focus if no Ruleflow-group is selected.
* **activation-group** **"some-name"** is an activation group from which only one rule in can fire. You can think of it as a more sophisticated form of **no-loop** **true**, which acts on groups instead of single rules.

There is a final part to tying rules and Ruleflow together, but first we need a quick lesson in stateful applications.

### A quick introduction to stateful applications

Most of the rules examples that we've seen are stateless. This means they run, fire the rules, print the values, and then terminate. When we run the example again, they 'forget' that they have been run before and produce exactly the same output.

Most real-life applications are stateful. How angry would you be if you'd logged into your web based online banking, only to find the application had forgotten about the money that you'd lodged in your account the last time you logged on?

In a normal application, such as an online banking web site, we need to remember what users did last (Are they logged in? What account are they looking at? Are they in the middle of making a payment? and so on). If we do not remember this data, users would get annoyed about having to repeat themselves at every step. It would also lead to some pretty complicated screens, to allow users to enter all of the information at once. Instead, we allow users to enter information in several steps, and remember where they are each time.

In an application designed to be used by computers, we don't have to worry about this. We can force the computer to give us all of the information required in one go, for example username, password, bank account to take money from, bank account to give money to, the date on which to execute the transaction, and so on. This is actually easy for a computer, because we make one call to our banking service and we are told whether our transaction has succeeded or failed. It's also easier for us to build our service.

Each service (transferring money, booking flights, or executing share trade) does only one thing.

Because each service 'forgets' everything after each call, we don't need to worry about trying to remember what we were doing before.

Because we have no memory, stateless applications (and services) are very scalable. We can make several copies of the same service and put them in a pool. Any client can talk to any service—no waiting for a particular server to become available.

In summary, stateless applications are simpler and scale better. Stateful applications, although harder to design and build, are more user-friendly. There are scores of books dedicated to explaining which design to choose and how to build them. But for the time being, there is one important reason why we've introduced stateful applications.

#### Stateful rules and Ruleflow

In all of our previous examples, we've fired our rules in a stateless manner. That is, we pass all of the information that we need to our rules, allow the rules to fire, and then get the final result back. There is no 'state', and nothing to remember, as everything is fired in one go.

You may recall the following line from previous examples:

**runStatelessRules(RULES\_FILES,DSL\_FILE, facts, globals, logger);**

Notice that no value is returned by this piece of code, nor do we need it. We fire our rules in one go, and our facts (for example, cells in the Excel spreadsheet) are automatically updated.

Stateless rule sessions are fine for our simple examples, where everything is completed in a few seconds. But what about our mortgage application Ruleflow, where the process can last for days, weeks, or even months? We will have to call our rules in a stateful way to remember the way we left things the last time we invoked the rule engine.

Stateful and stateless rules are almost the same. The only difference is the manner in which we call the rule engine.

The code extract (from **RuleFlowExample.java** in the **net** **firstpartners** **chap9** package or folder) shows us how. It also shows that while we gain the power of having stateful rules, it takes a bit more work on our part to call the rule engine.

**StatefulSession session = new RuleRunner()**

**.getStatefulSession(RULES\_FILES, null,**

**RULEFLOW\_FILE, ranges.getAllRangesAndCells(), globals,**

**excelLogger);**

**session.startProcess(RULEFLOW\_ID); session.fireAllRules();**

This code does a few things, such as:

* It uses our **RuleRunner** to load the rule file.
* Like before, it passes in our facts (the Cell JavaBeans), globals, and a handle to the logger. It also passes a handle to the Ruleflow file (so that Drools knows what process we wish to use).
* Unlike before, rules are not fired at this point. Instead, we get a **session** back—that is, a handle to the rule engine, with everything loaded and ready to go.
* Our second line of code starts the Ruleflow process by calling the **session.startprocess** method. This simply puts the token at the start of the Ruleflow because no rules are fired as yet. The **RULEFLOW\_ID** that we pass in is the one that we set in the properties screen of our Ruleflow (for example, **mortgage Ruleflow**).
* The final line of our code (**session.fireAllRules**) starts our rules. The difference between the rules firing in this and the previous samples is that the Ruleflow is guiding which groups of rules become available to fire, and when. As before, when all of the possible rules have fired, our fact objects (the Cell JavaBeans) are updated, and control returns to our program.

The rest of the code example (for example, the code that converts the values from Excel to JavaBeans and back again, the logging, and so on) remains the same as the code we've used in other examples.

There are a lot of other things that we can do with the Drools session, and not just Ruleflow. For example, as we step through the web pages for a Homeloan application, we can have Drools working in the background to ensure that all rules remain true, by updating the facts. We'll cover this truth-maintenance capability in more detail in the next chapter.

## Summary

This chapter aimed to make our rules both easier to use, and more powerful.

We started with a DSL, or domain specific language. By using DSLs, business rule editors can write near-English rules in a language suited to their profession. Drools then does the hard work of finding and replacing these rules, using an expander, so that we end up with a more technical rule that we can execute in the normal way.

Ruleflow also makes our rules easier to understand. It allows us to extract the flow that might otherwise be hidden in our rules. We saw how to create the Ruleflow diagram, and how to run it as part of a stateful rules session.

In our next chapter, we'll see how to take these new rules capabilities and deploy them as a part of a real-world application.