# Advanced Drools Features

So far we've covered several big topics such as what a rule engine is, why we would want to use one, and where you can download the Drools Rule Engine. We wrote rules using the Guvnor web editor and the more sophisticated JBoss IDE, before testing those rules using a variety of tools. We used Excel to hold both rules and data, and wrote our own DSL and rule flows. Not only that, we also looked under the covers to see some rule engine internals and understand how we would go about deploying rules in real life.

This chapter is a bit different from the earlier chapters. It is a level up from the introductory chapters as it presents additional information about the topics already covered in those chapters. A lot of information comes from the latest version of Drools. Although these features are stable and can be used in production systems, they may undergo substantial enhancements by the time you read this.

We will cover the new features of Guvnor. They include a new more powerful

API for calling the rule engine from Java, the ability to load data, create Java Beans dynamically, **Complex** **Event** **Processing** (**CEP**), and Drools Solver. We also explain some more under-the-cover features such as backward/forward chaining, controlling conflict resolution, and Rule Engine Standards (JSR-94).

Let's start with Drools Fusion—Complex Events Processing.

## Pigeons, Drools, and Complex Event Processing

Pigeons (the birds that flock around city squares and parks) aren't known for being clever. So it may strike as strange that they appear in a book about business intelligence. However, during the Cold War, the Soviets (allegedly) trained pigeons to inspect ball-bearings on the production line. The pigeons would sit in comfortable little boxes while the shiny silver ball-bearings steamed past on a conveyor belt. If the pigeon spotted any that were defective, they would peck a button and the broken bearing would be gone. Since the fall of the Berlin Wall, all of the pigeons have been gainfully re-employed at Google (**http://www.google.com/technology/ pigeonrank.html**).

Thankfully, the pigeons didn't go to work at a bank in the city. (Have you ever seen anything with feathers drive a Ferrari?) Although the pigeons would be very good at responding to simple market events (if market is up then sell, and if market down then buy), more complex analysis escapes them. For example, consider a situation where the market is down for 30 minutes. The shares in Acme corp are down by more than 10% than the average. But if you see three buy orders for that share in the last 60 seconds, then you may think that the market is about to turn, and hence buy shares in Acme corp.

Never mind the pigeons. Even most humans would find that difficult—think about trying to read the stock ticker prices (the ones you see rolling across the screen at MSNBC) for all stocks, while trying to hold the buy and sell information for the last 30 minutes in your head. And do that not only for one, but for a hundred different types of shares in the market. You have to do this while keeping an eye on your own trading position so that you're not exposed to one sector of the market (for example, keeping enough cash, and not too many property or technology shares). No wonder most traders make their millions and burn out before they're 30—that sort of **Complex** **Event** **Processing** (**CEP**) will wear you out.

Most IT applications are like pigeons; they can only handle simple events. Press the button. Do something. The way to make millions is to design applications that can handle these complex events, and apply sophisticated business rules to the (evolving) situation. You have to do it quickly enough (milliseconds) to seize the opportunity before somebody else does, and keep on doing it as long as the market is open.

CEP is what Drools Fusion provides.

### Implementing Complex Event Processing using Fusion

There are a lot of 'under the cover' changes to the latest version of Drools. These allow it to handle the volume of events that happen (often all at once) during CEP. However, from the end user point of view using CEP is strangely familiar to:

1. Writing facts (Java Objects).
2. Writing rules—same as before.
3. Notifying Drools of events (in the same way as we asserted facts previously—facts and events are interchangeable in many ways).

The third step is similar to asserting the facts, which we've done before. The difference is that events will be in greater number (especially in a stock market trading system) than the facts. And there will (potentially) be no limit to the amount of events that can happen, while we will tend to have a good idea of the number of facts that we will encounter. Fortunately, the upgrades to Drools during the latest version are specifically intended to handle large volumes of multiple events.

Although we can manually assert the events into working memory, one optimization for complex event handling is to set the working memory to listen to a stream, and automatically pipe the event from one to the other. Even with this shortcut, our solution using CEP will involve the following three files:

1. The Java file describing the event object being used: It is a simple JavaBean, which is the same as the facts in the previous examples (that is, the facts could be used as events if desired).
2. The Rule file: It imports rules and other assets (as before), declares the events used in the rules, and connects them to any event streams that the rules listen to.
3. The Java code in the application: It loads the rules, either inserts the events or links it to the stream specified, fires the rules and then does something with the results.

We'll work through this based on a sample from the Drools code, **StreamsTest. java**. This is a unit test, but shows how CEP works in practice. We'll start with the JavaBean file.

**public class StockTick { private int tradeNumber; private String stockName; private int price; private long currentTimeMillis;**

**public StockTick(int tradeNumber, String stockName, int price, long currentTimeMillis) { this.tradeNumber=tradeNumber; this.stockName = stockName; this.price = price;**

**this.currentTimeMillis = currentTimeMillis;**

**}**

**public int getTradeNumber() { return tradeNumber; }**

**public String getStockName() { return stockName;}**

**public int getPrice() {**

**return price;}**

**public long getCurrentTimeMillis() {**

**return currentTimeMillis; }**

**}**

Note that this class is immutable—once it's created it cannot be changed. In real life, events once occurred can't be altered, so the reasoning behind this feature is understandable.

This 'immutability' is implanted in two parts in the code above: The first part is that values that can only be set when the JavaBean is created (the line starting with **public** **StockTick(int** **tradeNumber,** **String** **stockName,** **int** **price,** **long** **currentTimeMillis)**). The second is that there is no 'set' method on the JavaBean, but there are methods beginning with 'get' to read the values. There is no chance to change an event that has already happened.

The second file is the rules file, an example of which is given as follows:

**package org.drools; import org.drools.StockTick; global java.util.List results; declare StockTick @role( event ) end rule "Test entry point" when**

**$st : StockTick( company == "ACME", price > 10 ) from entry-point**

**StockStream then**

**results.add( $st ); end**

This rules file has the usual package, import, and global statement. The three lines beginning with **declare** **StockTick** are new. They take the **StockTick** JavaBean we declared in the previous file, and let the rule engine know that we want to treat it as an event.

The rule matches all the events from the external stream where that event concerns the **ACME** stock, and where the price is greater than 10. When fired, the **then** part adds it to a **results** list—declared as a global variable at the top of the rules file.

The second new part in the rule file is in the **when** part of the rule beginning with **from** **entry-point** **StockStream**. We have met the **from** keyword before. It allows us to reach outside the rules' working memory (for example, to values in a database using the Hibernate framework) and allow those values to trigger rules. In this case, the source of those values is an event stream called **StockStream**. The code that calls the rule engine will supply this stream.

An extract of the application code with the complete file is available at

**http://anonsvn.labs.jboss.com/labs/jbossrules /trunk/drools-compiler/ src/test/java/org/drools/integrationtests/StreamsTest.java**. Most of what has been removed is package and import statements (most of the Drools classes referred to can be found in the **org.drools** package, or subpackages under it, from the Drools core library). The code is in the form of a unit test, that is, it has **assert** statements to check the results of the rules; these statements have also been removed in the interest of clarity.

The first method loads the rules base using the new Drools API (we will talk more about this API shortly). It searches the classpath for a given file name, loads it as a rule file, and then returns Drools **Knowledgebase** based on that rule file.

**private KnowledgeBase loadKnowledgeBase(final String fileName)**

**throws IOException, DroolsParserException**

**{**

**KnowledgeBuilder kbuilder =**

**KnowledgeBuilderFactory.newKnowledgeBuilder();**

**kbuilder.add( ResourceFactory. newClassPathResource(fileName,getClass()),ResourceType.DRL);**

**KnowledgeBase kbase = KnowledgeBaseFactory.newKnowledgeBase(); kbase.addKnowledgePackages( kbuilder.getKnowledgePackages() ); return kbase; }**

The second important method in the file is the one that uses these rules to 'do something'. It's called **testEventAssertion()**. You may remember from Chapter 7 that unit tests such as this will have every method starting with 'test', and will be called by the framework. In real life, this method would most likely be called something similar to 'call rules', but the contents of the method will be similar.

**public void testEventAssertion() throws Exception {**

**// read in the source using the method explained previously**

**// and the rules file (drl) we featured earlier**

**KnowledgeBase kbase = loadKnowledgeBase( "test\_EntryPoint.drl" );**

**KnowledgeSessionConfiguration conf = new SessionConfiguration();**

**((SessionConfiguration) conf)**

**.setClockType( ClockType.PSEUDO\_CLOCK ); StatefulKnowledgeSession session = kbase.newStatefulSession( conf ); final List results = new ArrayList(); session.setGlobal( "results", results );**

**StockTick tick1 = new StockTick(**

**1, "DROO", 50, System.currentTimeMillis() );**

**StockTick tick2 = new StockTick(**

**2, "ACME", 10, System.currentTimeMillis() );**

**StockTick tick3 = new StockTick(**

**3, "ACME", 10, System.currentTimeMillis() );**

**StockTick tick4 = new StockTick(**

**4, "DROO", 50, System.currentTimeMillis() );**

**……**

**WorkingMemoryEntryPoint entry =**

**session.getWorkingMemoryEntryPoint( "StockStream" ); entry.insert( tick1 );**

**……**

**session.fireAllRules(); }**

In this artificial sample we create the four events ourselves, and then stop them. In real life, there would be another application constantly passing us events over a longer period. These events can be passed by another Java thread, a web service, or passed over a network connection.

Several important things are going on in this method:

1. The rules (that we saw earlier) are loaded into a Drools **knowledgebase** object using the helper method which we have just reviewed.
2. We create configuration settings. Since this is a test, here we want to use a clock that we have more control over, rather than one that is tied to an internal system clock. This is important in situations (such as testing, where you have multiple rule servers, or you need to 'replay' events later) where you cannot guarantee that the time will be exactly as you expect. A pseudo clock gives us control over this situation.
3. Using this configuration, a **StatefulKnowledgeSession** is created—similar in concept to the stateful session that we met earlier. A global variable (results) is passed to this session.
4. Next, four events (of type **StockTick**) are created. Each time we create a **StockTick**, we must pass the stock tick number, the stock name, the price, and the current system time.
5. We get a handle to the **StockStream** from the recently created **StatefulKnowledgeSession**, using the **getWorkingMemoryEntryPoint()** method.
6. Using the handle to the **StockStream**, we pass in the **StockTick** events. Passing in the **StockTick** events is exactly the same as passing in facts to the working memory. The advantage of using the stream method is its speed, as only rules that listen to events will inspect the newly inserted objects.
7. When we've inserted all of the objects, we allow the rules to fire by calling **session.fireAllRules** again. Once the rules have finished firing, the objects (such as the global variable results that we passed in earlier) are updated and become available for use in the rest of the Java code.

If this unit test runs correctly, only one event will match the rule and cause one line to be added to the results list.

If you look at the test on the web site, you'll see that the actual line for inserting events contains a reference to an item called a **facthandle**. It is a simple handle to the object after we pass it to the rule engine. It's useful for testing (and other advanced situations), but does not change the meaning of the line from the one described on the previous page.

This simple example only hints at the power of complex event processing. For example, instead of **fireAllRules** used in the example, we could have called **fireUntilHalt**. This is more suitable for situations (such as events) where we neither know the number of objects or events that may come our way, nor the timescale in which they will be made available to us. **FireUntilHalt** is more suitable for these situations. Typically, there would be two threads to your program (that is, the computer is doing two things at one time). One thread is similar to the code in the example above, and calls **fireUntilHalt**. The other is listening for events and adding them to the working memory. At a time of our choice (for example, we choose to stop trading on the stock market) we call **halt()** and the rules will no longer fire, even if the events continue to happen.

### More powerful events

The events syntax in the previous sample is fairly simple. Primitive events are those events in which there is no restriction of duration, no need for two events to occur in or out of sequence, or to occur within a particular time window. To add these kind of complex restrictions to the **when** part of the rule, you can use **coincides**, **before**, **after**, **meets**, **overlaps**, **during**, **starts**, and **finishes** as special Drools keywords. They can help in filtering your rules. The following is an example:

**when**

**$st : StockTick( company == "ACME", starts > '01/01/2009' ) from entry-point StockStream**

There is also the option for sliding time windows. For example, to match all events in the last 60 seconds, the syntax has to do the following:

**StockTick( ) over window:time( 60 )**

This will match all of the **StockTick** events that happened in the last 60 seconds.

A detailed list of the syntax is available in **test\_CEP\_TimeRelationalOperators. drl** file at **http://anonsvn.labs.jboss.com/labs/jbossrules/trunk/droolscompiler/src/test/resources/org/drools/integrationtests/test\_CEP\_ TimeRelationalOperators.drl**.

## Inline beans

For the above sample, as with every other sample in the book, we've had to create a JavaBean to hold the information going into and out of the rule engine. Although writing this JavaBean is relatively straightforward, it still involves writing code in Java. As we saw, this is nothing to be scared of, but it is something you hardly want to do while writing your rules.

The latest version of Drools can help you with this problem. It allows you to declare beans within your rule files, and use those beans exactly as we did in other samples. Declaring an inline bean is fairly straightforward.

**global … declare SomeJavaBeanName javaBeanId : long name : String quantity : Integer price : double end rule …**

In this extract we declare the **SomeJavaBeanName** bean with fields of **javaBeanId** (number, long), **name** (string or text), **quantity** (number, integer), **price** (number, double). The field types (String, double, and so on) are the same types used in the real JavaBeans.

Inline beans are ideal for event handling, especially where the event is coming from an external source. In this case, we change the above bean definition to include **@ role(** **event** **)** as follows:

**declare SomeJavaBeanName**

**@role( event ) javaBeanId : long …**

### Loading data when your beans don't exist—Smooks

Declaring the bean inline is great for editing, but how does the rest of the system know the structure of the bean? If no JavaBean exists in a **.java** file, then Enterprise Java has no way of knowing what information it can pass in or out, the names of these fields, and the type (for example, text or numbers) of these fields.

As we've already seen in Chapter 8, the solution for this is providing the data in a standard format (such as Excel). This means that we know the format the data will be provided in, so that we can concentrate on the data rather than the format. In real life, we need to deal with other formats such as XML (a stricter form of HTML that makes up web pages, useful for data transportation), **CSV** (**Comma** **Separated** **Values**)—the most basic Excel format, and other proprietary formats. That's where the Smooks framework comes in.

If Smooks can handle data from Excel, then why use the method suggested back in Chapter 8? The answer is that Smooks is better at handling data. But given that it uses CSV instead of Excel files directly, it's not so good at keeping the format of the Excel file.

Smooks is an open source Java framework that exists independently of Drools.

Smooks is a Java Framework/Engine for processing XML and non-XML data (CSV, EDI, Java, JSON, and so on). You'll need to add the Smooks library to your project in order to use the functionality it provides.

To put it simply, Smooks loads the Excel (CSV), XML, or any other file and generates the events declared as inline beans in the rule file. It maps everything from the source file to the inline bean using the configuration that you provide in **smooksconfig.xml**. Once the data is loaded as an inline bean, it is asserted into the working memory. Then the rules that we have written have a chance to fire. The Java code to do this is standard Drools, with one line telling Drools to use Smooks to load the incoming data.

**//Load the rules into a rulebase and get a session as before PackageBuilder packageBuild = new PackageBuilder(); packageBuild.addPackageFromDrl( new InputStreamReader( getClass(). getResourceAsStream( "myRuleFile.drl" )) ); RuleBase ruleBase = RuleBaseFactory.newRuleBase(); ruleBase.addPackage( packageBuild.getPackage() );**

**StatefulSession session = ruleBase.newStatefulSession();**

**// Load Smooks using the Config file**

**Smooks smooks = new Smooks( "smooks-config.xml" );**

**//Tell Drools-Smooks the point in our file that we want to start loading data.**

**DroolsSmooksConfiguration conf = new DroolsSmooksConfiguration( " someJavaBeanName ", null );**

**// Prepare a (Stateless) Session using the Smooks and Drools conf**

**DroolsSmooksStatelessSession smooksSession = new**

**DroolsSmooksStatelessSession( session, smooks, conf );**

**//Get a handle to the data file javax.xml.transform.Source xmlSource = new StreamSource( getClass(). getResourceAsStream( "name-of-data-file.xml") )**

**//Load the data (and allow the rules a chane to fire) smooksSession.executeFilter(xmlSource);**

Once the rules have fired (following **excecuteFilter**), the Java code has a chance to inspect global rule variables. Thus, they get a handle to the result.

Mostly what we want to do is map the incoming XML (or other format) directly to the inline bean that we created in the rule file. In the following XML extract most of the names correspond directly to the names in our inline bean:

**<someJavaBeanName>**

**<javaBeanId>123</javaBeanId> <name>Onions</name>**

**<quantity>72</quantity>**

**<price>1.2</price> </someJavaBeanName>**

The **smooks-config.xml** configuration file to do this mapping can be quite sophisticated. For more information on how to use all of the available power, visit the Smooks web site at **http://www.smooks.org/**.

## From pigeons to biscuits—Drools Solver for your local supermarket

Stock market trading might appear to be a little exotic. How about another trading problem that is causing the manager of your local supermarket to scratch his or her head?

Supermarkets, no matter how big they are, have a limited amount of shelf space. Obviously, supermarkets want to make as much money as possible. So, they want to stock the most profitable products on their shelves. However, even if chocolate bars are the most profitable line, a supermarket full of nothing but chocolate bars isn't going to make many sales. Even the most ardent chocolate fans are going to need things such as milk, bread, and cheese. So what mix of products will make the supermarket the most money?

Supermarkets have detailed information on the amount of sales in each of their stores—those barcode scanners do more than just give you your bill. They track each and every item sold in the shop. The barcode information also includes the most basic list of must-have items of the customers. The supermarket knows how much shelf space it has in any particular store and also the profit margin on each product. But this information doesn't tell the store manager how much money the store could potentially make if they had a different mix of goods on the shelves.

Part of the problem in answering the question about the best mix of products is that there are many potential factors involved—there are 100,000 products in a typical supermarket, and there are about 1,000 different ways of displaying them (top shelf, end shelf, back of store, next to the milk) in different combinations. (Does beer really sell well if it is placed next to the baby nappies as the urban legend has it?)

In classic mathematical or computing terms, this is similar to the 'travelling salesman' problem, where it is technically possible to calculate the answer. Even with computers more powerful than they are today, it would take you about 100 years to get an answer; and another 100 years to calculate it all over again if another type of beer is released in the market.

Once you tell this to the supermarket manager, you realize that (s)he would be happy even if you found a slightly better way of placing the products than the current situation. It may not be the best solution, but it's good enough to get him his bonus. And good enough solutions are something that Drools Solver can help with.

### How Drools Solver works

At a very simplistic level, the way Drools Solver works is as follows:

1. One correct solution is supplied at the start—the current mix of products on the supermarket shelves.
2. Alternative solutions are generated either randomly or via an algorithm that is most likely to provide usable results.
3. Impossible solutions (for example, milk not being stored in refrigerators) are discarded. These hard constraints are expressed as rules.
4. The remaining solutions are scored using business rules. The scoring (or soft constraints) is also expressed as rules.
5. The best available solution is used when we loop back to step 1.

This process repeats until either a certain time period has elapsed, or until Drools Solver cannot find a better solution.

The Solver could have been written without using business rules, but the addition of Drools makes it so much better.

* Clear rules: What combinations are allowed, and how we score alternatives is clearly stated.
* Scalable rules: The Drools Rete Algorithm means that adding additional rules will only add milliseconds to the time required, and will not double the time needed (as per traditional solutions). This is especially important when the calculations are run repeatedly (as happens within the Solver).
* Optimized calculations: Drools detects the change from a previous to new scenario, and can carry out only those calculations required. That is, it does not need to recalculate everything from scratch.

#### Implementing a Solver

A full Solver solution is provided as part of the Drools examples on the download site (on which the solution below is based). Implementing the five steps given above is relatively straightforward.

Steps 1 and 5 are provided by implementing the Solution interface. This, like most of the files mentioned in this section, can be found in the **org.drools.solver.core** package (or its subfolders).

We did warn that this chapter was more advanced than the previous ones! Although we'll explain concepts to a non-technical level, you'll probably need to have some Java experience (or know somebody who does) to actually implement the examples.

Think of an interface as a template guiding you to fill in the gaps and complete the solution—in general, this 'complete template' is contained in a second Java file. In this case, **getFacts()** returns the JavaBeans representing the goods on the shelves of the supermarket as we begin our analysis. The **cloneSolution()** method allows step 5 to copy any better solution (that is, save a good copy while it tries to improve it even further).

**public interface Solution { Collection<? extends Object> getFacts();**

**Solution cloneSolution();**

**}**

Running the solver is easy. We create a configuration, load the configuration file, and then start the Solver (which may take some time).

**XmlSolverConfigurer configurer = new XmlSolverConfigurer(); configurer.configure("SolverConfig.xml"); Solver solver = configurer.buildSolver();**

**The configuration file ties the remaining parts of the solution together. The Comments explain what each part means.**

**<!—Name of the file containing the solver business rules used for rating the alternatives--> <scoreDrl>SolverRulesFile.drl</scoreDrl>**

**<!—How we compare the various solutions – Simple is one of the three built in options, or can build or own -->**

**<scoreCalculator>**

**<scoreCalculatorType>SIMPLE</scoreCalculatorType>**

**</scoreCalculator>**

**<!—Finish solving after 2 minutes, or after 100 attemps, or after we have achieved the perfect score-->**

**<finish>**

**<finishCompositionStyle>OR</finishCompositionStyle>**

**<maximumMinutesSpend>2</ maximumMinutesSpend>**

**<maximumStepCount>100</maximumStepCount>**

**<feasableScore>0.0</feasableScore>**

**</finish>**

**<!—Name of the Java File that generates alternative solutions-->**

**<selector>**

**<moveFactoryClass>NameOfMoveFactory</moveFactoryClass>**

**</selector>**

**<!—An accepter filters out the most crazy alternatives. This is expressed as a score – trial and error will give a value that works best --> <accepter>**

**<completeSolutionTabuSize>1000</completeSolutionTabuSize>**

**</accepter>**

**<!—which of the best alternatives do we use – in this case take the alternative with the highest score from the rules - ->**

**<forager>**

**<foragerType>MAX\_SCORE\_OF\_ALL</foragerType>**

**</forager>**

**</localSearchSolver>**

The **MoveFactory** is another Java class that implements another interface or template (although there are other files that help in the background as 'abstract' and 'super' classes). The most important method is **createMoveList**, which is a list of the items that we wish to swap on the supermarket shelves in search for a better solution. It takes the implementation of the **Solution** interface as a parameter that we created earlier.

**public interface MoveFactory extends LocalSearchSolverAware, LocalSearchSolverLifecycleListener {**

**List<Move> createMoveList(Solution solution);**

**}**

The **LocalSearchSolverAware** and **LocalSearchSolverLifecycleListener**

files referenced are other interfaces. The methods (not listed here) that implement the **Solution** interface help the Solver avoid 'dead ends' that appear to be good solutions. We did not look at them in detail.

The following is an example of a rule file mentioned in the **config** file, which shows 'hard' constraints, such that beer and nappies should not be on the same shelf:

**rule "Stock items cannot be placed on the same shelf"**

**when**

**$item1 : StockItem($shelf : shelf, name =="beer");**

**$item2 : StockItem (shelf == $shelf, name =="nappies");**

**then**

**insertLogical(new UnweightedConstraintOccurrence(**

**"Incompatible objects on shelves", $item1, $item2)); end**

The match against the facts in the **when** part is the same as usual—remember all of the facts provided by the solution are inserted into the working memory, and updated as we cycle through the alternative solutions.

The **then** part shows how to communicate the fact that nappies and beer should not sit side by side on the same shelf. We insert a new

**UnweightedConstraintOccurrence** into the working memory. There are similar constraints (such as **IntConstraintOccurrence)** which allow us to communicate a score on how good the rules think the current solution is.

**InsertLogical** is similar to **insert**, except that the object will automatically be removed if the 'when' part is no longer true.

With all these pieces in place, Solver loads the initial solution, generates and evaluates alternative solutions, and then repeats the process until it decides to stop.

#### More information on Solver

Of course, the actual implementation of Drools Solver is much more sophisticated. More information is available in the Drools documentation. (See Chapter 2 for more details on where to obtain this.) When reading this documentation you may come across a line that might puzzle you.

It's recommended to use drools in forward-chaining mode […] This is a huge performance gain.

Forward and backward chaining are key (if advanced) rule engine concepts. So perhaps now is a good time to introduce them.

### Forward and backward chaining

You use forward and backward chaining in real life—except that you don't know it yet.

* Forward chaining is where you have the instructions, but you don't know the end goal of where the instructions are taking you. For example, it's like following the driving instructions on your SatNav or GPS where somebody else has entered the end coordinates.
* Backward chaining is where you have an end goal in mind, and need to work out how you get there. For example, if you need to go to the supermarket and work out the best way to drive there, then the way you work out the route is an example of backward chaining.

Drools is a hybrid rule engine because it allows both forward and backward chaining. Most of the examples in this book have used forward chaining. Backward chaining tends to either happen accidentally, or when we need to answer the question, 'Which is the set of facts that could lead us to a particular situation?' The Miss Manners example from Drools can be configured to show backward chaining behavior.

Forward and backward chaining is important because of its effect on performance. Backward chaining is very powerful, but much slower.

To explain this, consider the example of the tangle of wires behind your television set. There are probably power cables from your TV, DVD, satellite, games console, and Wi-Fi leading to the power sockets in the wall.

* If we want to confirm that the TV is plugged in (forward chaining), all we need to do is pull the power lead at the back of the TV, and confirm that the lead that is plugged in moves.
* If we want to confirm which gadgets are plugged in to where, we need to test each and every cord. This is why backward chaining is much slower.

Forward and backward chaining is implemented using the Rete algorithm, and can fill the entire contents of a book in itself. But if we did that, we'd miss the chance to touch on another important 'under the covers' topic of how Drools implements conflict resolution, and what you can do to configure it.

### Changing the conflict resolution methodology

In the previous chapter we met with a problem with rule engines. If there is more than one rule available to fire on the Agenda, how do we decide which one goes first? This is especially important as the first rule to fire might cause the subsequent rules to be removed from the agenda, and hence prevent them from firing. The solution was conflict resolution—an in-built method that comes with Drools. It uses this methodology to resolve which rule fires first. But what if we want to change this methodology?

The code sample below shows how we do this (this happens behind the scenes without us needing to change it). Normally, the current order of conflict revolvers is based on salience (number), last fired, rule simplicity, and then rule order. Adding the following code will allow you to tweak the conflict resolution strategy of your rule engine:

**ConflictResolver[] conflictResolvers = new ConflictResolver[] {**

**SimplicityConflictResolver.getInstance(),**

**SalienceConflictResolver.getInstance(),**

**RecencyConflictResolver.getInstance(),**

**LoadOrderConflictResolver.getInstance()**

**};**

**CompositeConflictResolver resolver = new CompositeConflictResolver( conflictResolvers);**

**businessRules = RuleBaseLoader.loadFromUrl(**

**BusinessLayer.class.getResource(**

**BUSINESS\_RULE\_FILE),resolver);**

In this case we've made simple rules (that is, fewer conditions in the **when** part) that are more likely to fire first. We've done that by building up an array of the conflict revolvers that Drools offers us in the *order of importance* that we want. Then we add this array (list) together in a **CompositeConflictResolver**. Finally, when we load the rules, we pass in the **CompositeResolver**. When the rules fire, conflicts (if any) are resolved using the strategies in the order that we specify.

Like forward and backward chaining, it's important to be aware of the conflict resolution strategy. Likewise, in everything but the most advanced situations, you shouldn't need to change it. So if you find yourself tweaking the conflict resolution, double-check that the problem isn't in your rules. Make sure you have a good set of unit tests around your rules. When fixing one rule, make sure that you don't inadvertently break others.

### Standard rule engine API—JSR 94

The standard rule engine API, **JSR-94** (**Java Specification Request**), like conflict resolution, will come up in conversation soon after you bring up the topic of rule engines with your colleagues. JSR-94 is the Java standard API for dealing with rule engines in almost the same way as the Java Database Connectivity (JDBC) is the Java standard way of connecting Java to databases. So why is this piece of information tucked away in the very last chapter?

* JSR-94 only specifies how we call the rule engine, and not the syntax of the rules themselves. There is no rules standard similar to ANSI SQL (for databases) to specify the format.
* Even when calling rule engines, JSR-94 gives you only a subset of the power of Drools (or most other rule engines).

Although there is no standard rule format, it is possible for Drools to mimic other commercial rule engine syntax (such as Blaze) using Domain Specific Language (DSL). At the time of writing you'll need to write this DSL yourself, but given the open source nature of Drools, expect much progress in this area. For example, Drools Clips emulation is already in an advanced stage.

If you choose to use the JSR-94 syntax, you'll need to add an additional library—**drools-jsr94.jar**—to your project. The code to load and fire the rules in the standard manner is very close in concept to the examples we saw earlier, even if the terminology used will look something similar to the extract below. This code sample is how the sample from Chapter 6 (remember those **ChocolateFactory** rules?) would look using JSR-94.

**//Specify the exact provider of the JSR-94 driver**

**Class.forName("org.drools.jsr94.rules.RuleServiceProviderImpl"); // Get the rule service provider from the provider manager.**

**RuleServiceProvider provider = RuleServiceProviderManager.getRuleServi ceProvider("http://drools.org/");**

**// Get a handle to the Administration API-**

**RuleAdministration ruleAdministrator = provider.**

**getRuleAdministrator();**

**LocalRuleExecutionSetProvider ruleExecSet = ruleAdmin. getLocalRuleExecutionSetProvider( null );**

**// Create a Reader for the drl**

**URL drl = new URL("shipping-rules.drl");**

**Reader drlReader = new InputStreamReader( drl.openStream() );**

**// Create the RuleExecutionSet for the drl RuleExecutionSet ruleExecutionSet = ruleExecSet. createRuleExecutionSet( drlReader, null );**

**// Register the RuleExecutionSet with the RuleAdministrator**

**String uri = ruleExectionSet.getName();**

**ruleAdministrator.registerRuleExecutionSet(uri, ruleExecutionSet, null);**

**//Get a stateless execution set from this set**

**RuleRuntime ruleRuntime = provider.getRuleRuntime();**

**(StatefulRuleSession) session = ruleRuntime.createRuleSession( uri, null,RuleRuntime.STATELESS\_SESSION\_TYPE );**

**//Insert the facts – assume that we created these earlier session.addObject (candyBarOrder); session.addObject (holiday1); session.addObject (holiday2);**

**//Fire the rules session.executeRules();**

In some ways the code is more complicated (as we need to specify at the start that we are using the Drools Rule Engine). Like before, once the rules have fired, the updated Java objects will be available for use by the Java program. The Drools documentation gives further information on JSR-94. If you feel you need to use this, there is an alternative API to call the rule engine.

#### Other rule engines

So, if somebody has gone through a lot of trouble to write the JSR-94 specification for portability (even if you choose not to use it), what other rule engines are available to use? There are literally hundreds of frameworks that claim to embody business rules. But if we filter the list according to those that have a moderate amount of features and traction, either commercially or in the open source community, we get the following alternatives: (Remember that this is not an exhaustive list.)

* BizTalk
* Blaze Advisor
* Jena
* Jess
* JRules
* OpenRules
* PegaRules
* RulesPower

Normally, you'd expect at this point a critical evaluation of the rule engines mentioned in this list—something like 'Jess is a mature Java rule engine with good tool support (for example, plug-ins from Eclipse). It's a commercial product, and the rules are written in a Prolog-style syntax, which can be confusing for many Java programmers.' All of this would be true. However, given my bias (of course, I'm going to recommend Drools on the basis that it's the most mature, open source Java-based rule engine), would you believe any of the analysis?

The best way to summarize is that you should do your own research, and Drools will inevitably be on the shortlist. Some of the frameworks mentioned (for example, Jena) offer basic rules functionality, but it's not their core purpose (for Jena, it's the semantic web). Other frameworks claim to be rule engines, but offer only a fraction of the functionality discussed in this book. Outside Java, remember that Drools.Net offers a rule engine on the .Net platform (although its features are not as mature as those in Drools Java), and Windows workflow (the workflow module that is part of .Net platform) also has a subset of rules functionality.

## New API

You may have noticed that the examples in this chapter are using a slightly different API from those in other chapters. The reason for this is that Drools has a new API. Don't worry, all of the previous examples will continue to work as before.

Why change something that isn't broken? The changes to the API make interacting with Drools more 'knowledge engine' based and less 'rule engine' based.

This may seem unusual for a book on rule engines. But a lot of the functionality that is being added is broadening the capabilities of the engine into areas such as complex event processing, solving (both of which we saw earlier in this chapter), and workflow. (We touched upon it in Chapter 9, and we will deal with the extensive improvements in the next section.) Chances are that if you need a rule engine, you'll need (at least some of) these capabilities. So it makes sense to enhance Drools and make the API more generic to accommodate this.

The main parts of the new API that you interact with are as follows—more or less one for each replacement for the Drools objects we used earlier:

* **org.drools.builder.KnowledgeBuilder**
* **org.drools.KnowledgeBase**
* **org.drools.agent.KnowledgeAgent**
* **org.drools.runtime.StatefulKnowledgeSession**
* **org.drools.runtime.StatelessKnowledgeSession**

The notes accompanying the new release have good examples. The main addition in the samples below is that in the new API we need to explicitly state that we're loading a rule file (**ResourceType.DRL**). This is because **KnowledgeBuilder** can also load other resources, such as workflow files. The following sample shows how we would load the chocolate factory from Chapter 6, but using the new Drools API:

**URL drl = new URL("shipping-rules.drl");**

**KnowledgeBuilder kbuilder = KnowledgeBuilderFactory. newKnowledgeBuilder();**

**kbuilder.add( ResourceFactory.newUrlResource( url ), ResourceType.DRL );**

**//check for errors**

**if ( kbuilder.hasErrors() ) {**

**System.err.println( builder.getErrors().toString() );**

**}**

**KnowledgeBase kbase = KnowledgeBaseFactory.newKnowledgeBase(); kbase.addKnowledgePackages( builder.getKnowledgePackages() ); StatefulKnowledgeSession ksession = knowledgeBase. newStatefulKnowledgeSession();**

**//Insert the facts – assume that we created these earlier ksession.addObject (candyBarOrder); ksession.addObject (holiday1); ksession.addObject (holiday2);**

**ksession.fireAllRules();**

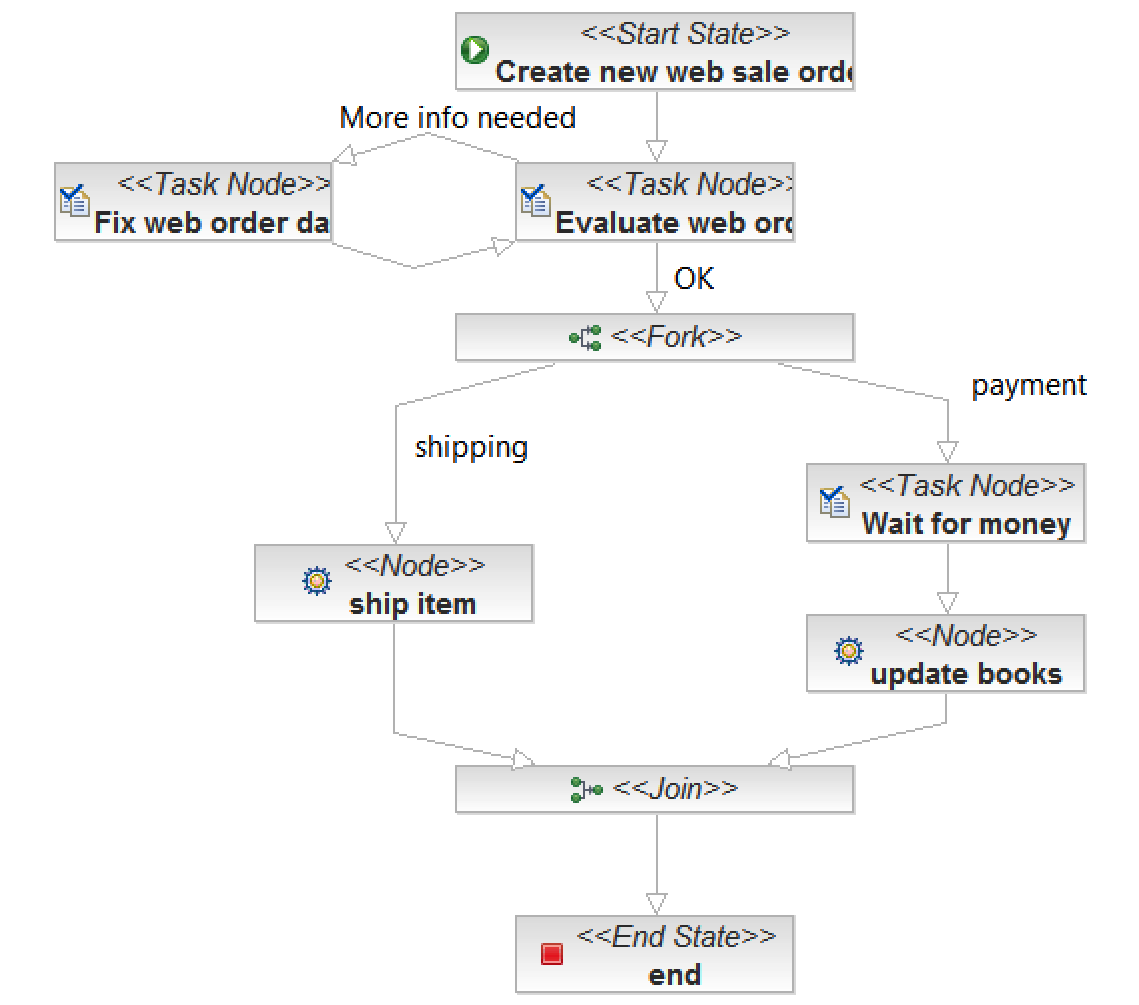
More information on these new objects can be found in the Javadoc at **https://hudson.jboss.org/hudson/job/drools/lastSuccessfulBuild/ artifact/trunk/target/javadocs/stable/drools-api/index.html**.

### Drools flow—a full workflow engine

Back in Chapter 9 we mentioned that Ruleflow is not a workflow. This holds true, and Ruleflow can still be used to control when the different sets of rules may fire. However, Drools 5 now has a first-class workflow engine integrated into it, plus all of the supporting tools that you'd expect. To remind ourselves of the differences, and what workflow is:

* Ruleflow says what might happen at each stage in the process. The actual rules that fire are selected by the rule engine.
* 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.

Typically, workflows model the business process. The following sample is from the **JBoss jBPM** (**Business Process Management**) documentation (the other JBoss workflow that Drools integrates well with). It describes the business steps needed to process an e-commerce order.



What both Drools Flow and JBoss jBPM provide are Eclipse-based tools to 'draw' the workflow—the image above is an example on one using the Eclipse tools. Using the tool, you then state what Java code you wish to fire at each point in the workflow. (It is easy to write because the Java code doesn't have to worry about what is happening around it, only that it will be told to run in the correct circumstances.) Drools and jBPM also provide a runtime that executes the workflow as the various steps are met.

Drools hooks into the decision-making node for both Drools flow and jBPM, allowing you to make (rules-based) decisions on what path you want the process to proceed.

### New features in Guvnor

Guvnor gains a lot of enhancements in the latest Drools release. These include:

* Fine-grained security: You can restrict what groups of users can and cannot do.
* Web-based decision tables: They have a similar grid-like format to that which the Excel decision tables have (as seen in Chapter 8), but are accessible through the Guvnor web interface.
* More powerful scenario-based testing tools: The ability to load files using Windows Explorer, that is, drag-and-drop uploads to Guvnor.
* An editor to take advantage to the declarative modelling/inline beans: There is no need to create JavaBeans in Eclipse. It can be done in Guvnor instead.

There are also substantial under-the-cover improvements to the core rule engine, the net effect of which is to increase stability and performance.

**Does this still sound like where you work?**

Back in Chapter 1, you may have taken the 10-question pop quiz with the heading 'Does this sound like where you work?' The aim was to show you the problems that Drools can solve. Lets take a look at the solutions.

1. Is Bob in the corner the only person who knows how the system really works? Can the business scale only if we have an expert? Is critical knowledge lost when people like Bob leave?

Solution: Getting the knowledge out of Bob's head and into a business rule-based system is a scalable, durable, solution.

1. If you're Bob (owning the knowledge), are you sick of people asking you stupid questions? Do you think: don't these people know that you've got a job to do?

Solution: Write your knowledge as business rules. That way, at the very least it's documented; and even better, it can be run in the rule engine.

1. Are your customers getting a different answer every time they call your company (and getting more than slightly irate about it)? Are you at the risk of receiving a slap on the wrist (or worse) from a regulator or other standards body?

Solution: Having the company's knowledge in one place (the business rules), and referring to that each time means that clients will get consistent answers.

1. Do you find yourself working around, rather than with, your computer systems? Have you ever thought of pouring coffee into your computer keyboard in frustration?

Solution: Drools won't stop you from pouring coffee into your computer. But since rules-based systems are more flexible, you should be less tempted to try pouring coffee into your computer in the first place.

1. Are things always done with books, or is there a lot of informal knowledge that is just in people's heads?

Solution: There will always be an element of informal knowledge in every business. But since rules are much easier to update than code, the balance will be shifted towards documented business rules (a good thing).

1. Did you prepare for a quality (ISO 9001) audit and then leave the process documentation unused on a shelf? Is there anybody around who knows or wants to change this process?

Solution: If you move your knowledge to rules and workflow, the next time you have a quality audit you just print the rules and send the auditors their way.

1. Is your business knowledge in some format that gives payback? (For example, Electronic instead of that dusty paper copy, and not locked away in unreadable machine-code.) Is this format easy to update? Can everybody use it from one central location (so that copies do not get 'out of sync')? Can you track changes and roll back if you get it wrong?

Solution: Drools gives a repository to securely store your knowledge and track changes. Even better, it can 'execute' that knowledge (as rules) directly from the repository.

1. Do the right people (and only the right people) have access (both reading and updating) to this information? Does this access need to change depending on the context of what the user is doing at the time?

Solution: The rules repository that comes with Drools Guvnor allows you to do just this.

1. Do people in your organization work on projects? Do they come together to form goal-driven teams, and then go back when the objectives have been achieved? Do you know how to document the outcome of these projects as rules so that they can be reused both over time and over the organization?

Solution: The project team (among other things) can generate new or updated business rules. Captured by Drools, these can be shared with the rest of the organization.

1. No task is done in isolation. How do we ensure that tasks and team members collaborate effectively?

Solution: Rules encourage sharing so that everybody can understand what is going on. Even better, the Guvnor web editor allows many people to contribute at the same time, without needing to install special software.

OK, so our pop quiz was slightly biased (this is a Drools book, after all). But the fundamental problems that Drools solves, and the change that JBoss Drools can make to where you work, should still be clear.

## Summary

We've come a long way since the start of this book. Chapter 1 introduced the problem of capturing business logic in a way that is readable and updateable by non-technical users. It introduced Drools and rule engines as a solution to the problem. Chapter 2 showed where to obtain Drools and other open source software, and how to install it on your computer. Chapter 3 introduced the Guvnor web editor, available to help non-technical users write business rules.

In Chapter 4 we began to write our first business rules using the Guvnor editor, and wrote our first JavaBean to support these rules using Eclipse/the JBoss IDE. In Chapter 5 we wrote more advanced rules using Guvnor, rather than the JBoss IDE, to write the business rules due to the increased power that it offers. Chapter 6 continued this thread to give a comprehensive guide to the rules syntax.

Testing the rules written in the previous chapters is the focus of Chapter 7, while Chapter 8 showed how to use Excel to hold both the business rules and the data for those rules. Chapter 9 showed other ways of expressing your rules using Domain Specific Language and Ruleflow.

How to deploy all of the business rules created to date was the focus for Chapter 10. In Chapter 11 we took a view inside the rule engine to help us write better business rules. Finally, in this chapter, we looked at advanced rules concepts, many of which have become available in the latest Drools release.

I hope this book inspires you to use Drools business rules in your project. Good luck and remember to check for updates to Drools at **www.jboss.org/drools** and code samples at **http://code.google.com/p/red-piranha/**.