JBoss Enterprise BRMS Platform 5.0

JBoss Rules 5 Reference Guide

Your complete guide to using JBoss Rules 5 with the JBoss Enterprise BRMS Platform.

Mark Proctor

Michael Neale

Edson Tirelli

JBoss Enterprise BRMS Platform 5.0 JBoss Rules 5 Reference Guide
Your complete guide to using JBoss Rules 5 with the JBoss Enterprise BRMS Platform.
Edition 1

Author	Mark Proctor	
Author	Michael Neale	
Author	Edson Tirelli	
Editor	Darrin Mison	*dmison@redhat.com*

1801 Varsity Drive
Raleigh, NC 27606-2072 USA
Phone: +1 919 754 3700
Phone: 888 733 4281
Fax: +1 919 754 3701
PO Box 13588 Research Triangle Park, NC 27709 USA

This guide contains a complete overview and detailed reference for JBoss Rules 5. It is intended for use with the JBoss Enterprise BRMS Platform.

Preface

1. Document Conventions

This manual uses several conventions to highlight certain words and phrases and draw attention to specific pieces of information.

In PDF and paper editions, this manual uses typefaces drawn from the *Liberation Fonts*[1] set. The Liberation Fonts set is also used in HTML editions if the set is installed on your system. If not, alternative but equivalent typefaces are displayed. Note: Red Hat Enterprise Linux 5 and later includes the Liberation Fonts set by default.

1.1. Typographic Conventions

Four typographic conventions are used to call attention to specific words and phrases. These conventions, and the circumstances they apply to, are as follows.

`Mono-spaced Bold`

Used to highlight system input, including shell commands, file names and paths. Also used to highlight key caps and key-combinations. For example:

> To see the contents of the file **`my_next_bestselling_novel`** in your current working directory, enter the **`cat my_next_bestselling_novel`** command at the shell prompt and press **`Enter`** to execute the command.

The above includes a file name, a shell command and a key cap, all presented in Mono-spaced Bold and all distinguishable thanks to context.

Key-combinations can be distinguished from key caps by the hyphen connecting each part of a key-combination. For example:

> Press **`Enter`** to execute the command.
>
> Press **`Ctrl`**-**`Alt`**-**`F1`** to switch to the first virtual terminal. Press **`Ctrl`**-**`Alt`**-**`F7`** to return to your X-Windows session.

The first sentence highlights the particular key cap to press. The second highlights two sets of three key caps, each set pressed simultaneously.

If source code is discussed, class names, methods, functions, variable names and returned values mentioned within a paragraph will be presented as above, in **`Mono-spaced Bold`**. For example:

> File-related classes include **`filesystem`** for file systems, **`file`** for files, and **`dir`** for directories. Each class has its own associated set of permissions.

Proportional Bold

This denotes words or phrases encountered on a system, including application names; dialogue box text; labelled buttons; check-box and radio button labels; menu titles and sub-menu titles. For example:

[1] https://fedorahosted.org/liberation-fonts/

> Choose **System > Preferences > Mouse** from the main menu bar to launch **Mouse Preferences**. In the **Buttons** tab, click the **Left-handed mouse** check box and click **Close** to switch the primary mouse button from the left to the right (making the mouse suitable for use in the left hand).
>
> To insert a special character into a **gedit** file, choose **Applications > Accessories > Character Map** from the main menu bar. Next, choose **Search > Find…** from the **Character Map** menu bar, type the name of the character in the **Search** field and click **Next**. The character you sought will be highlighted in the **Character Table**. Double-click this highlighted character to place it in the **Text to copy** field and then click the **Copy** button. Now switch back to your document and choose **Edit > Paste** from the **gedit** menu bar.

The above text includes application names; system-wide menu names and items; application-specific menu names; and buttons and text found within a GUI interface, all presented in Proportional Bold and all distinguishable by context.

Note the **>** shorthand used to indicate traversal through a menu and its sub-menus. This is to avoid the difficult-to-follow 'Select **Mouse** from the **Preferences** sub-menu in the **System** menu of the main menu bar' approach.

`Mono-spaced Bold Italic` or ***`Proportional Bold Italic`***

Whether Mono-spaced Bold or Proportional Bold, the addition of Italics indicates replaceable or variable text. Italics denotes text you do not input literally or displayed text that changes depending on circumstance. For example:

> To connect to a remote machine using ssh, type **`ssh`** ***`username@domain.name`*** at a shell prompt. If the remote machine is **`example.com`** and your username on that machine is john, type **`ssh john@example.com`**.
>
> The **`mount -o remount`** ***`file-system`*** command remounts the named file system. For example, to remount the **`/home`** file system, the command is **`mount -o remount /home`**.
>
> To see the version of a currently installed package, use the **`rpm -q`** ***`package`*** command. It will return a result as follows: ***`package-version-release`***.

Note the words in bold italics above — username, domain.name, file-system, package, version and release. Each word is a placeholder, either for text you enter when issuing a command or for text displayed by the system.

Aside from standard usage for presenting the title of a work, italics denotes the first use of a new and important term. For example:

> When the Apache HTTP Server accepts requests, it dispatches child processes or threads to handle them. This group of child processes or threads is known as a *server-pool*. Under Apache HTTP Server 2.0, the responsibility for creating and maintaining these server-pools has been abstracted to a group of modules called *Multi-Processing Modules* (*MPMs*). Unlike other modules, only one module from the MPM group can be loaded by the Apache HTTP Server.

1.2. Pull-quote Conventions

Two, commonly multi-line, data types are set off visually from the surrounding text.

Output sent to a terminal is set in `Mono-spaced Roman` and presented thus:

```
books         Desktop   documentation  drafts  mss    photos   stuff  svn
books_tests   Desktop1  downloads      images  notes  scripts  svgs
```

Source-code listings are also set in `Mono-spaced Roman` but are presented and highlighted as follows:

```
package org.jboss.book.jca.ex1;

import javax.naming.InitialContext;

public class ExClient
{
   public static void main(String args[])
       throws Exception
   {
      InitialContext iniCtx = new InitialContext();
      Object         ref    = iniCtx.lookup("EchoBean");
      EchoHome       home   = (EchoHome) ref;
      Echo           echo   = home.create();

      System.out.println("Created Echo");

      System.out.println("Echo.echo('Hello') = " + echo.echo("Hello"));
   }

}
```

1.3. Notes and Warnings

Finally, we use three visual styles to draw attention to information that might otherwise be overlooked.

Note

A Note is a tip or shortcut or alternative approach to the task at hand. Ignoring a note should have no negative consequences, but you might miss out on a trick that makes your life easier.

Important

Important boxes detail things that are easily missed: configuration changes that only apply to the current session, or services that need restarting before an update will apply. Ignoring Important boxes won't cause data loss but may cause irritation and frustration.

Warning

A Warning should not be ignored. Ignoring warnings will most likely cause data loss.

2. We Need Feedback!

If you find a typographical error in this manual, or if you have thought of a way to make this manual better, we would love to hear from you! Please submit a report in Bugzilla: *http://bugzilla.redhat.com/bugzilla/* against the product **Documentation.**

When submitting a bug report, be sure to mention the manual's identifier: *JBoss_Rules_5_Reference_Guide*

If you have a suggestion for improving the documentation, try to be as specific as possible when describing it. If you have found an error, please include the section number and some of the surrounding text so we can find it easily.

The Rule Engine

1.1. What is a Rule Engine?

1.1.1. Introduction and Background

Artificial Intelligence (A.I.) is a very broad research area that focuses on "Making computers think like people" and includes disciplines such as Neural Networks, Genetic Algorithms, Decision Trees, Frame Systems and Expert Systems. Knowledge representation is the area of A.I. concerned with how knowledge is represented and manipulated. Expert Systems use Knowledge representation to facilitate the codification of knowledge into a knowledge base which can be used for reasoning, i.e. we can process data with this knowledge base to infer conclusions. Expert Systems are also known as Knowledge-based Systems and Knowledge-based Expert Systems and are considered to be "applied artificial intelligence". The process of developing with an Expert System is Knowledge Engineering. EMYCIN was one of the first "shells" for an Expert System, which was created from the MYCIN medical diagnosis Expert System. Whereas early Expert Systems had their logic hard-coded, "shells" separated the logic from the system, providing an easy to use environment for user input. Drools is a Rule Engine that uses the rule-based approach to implement an Expert System and is more correctly classified as a Production Rule System.

The term "Production Rule" originates from formal grammars where it is described as "an abstract structure that describes a formal language precisely, i.e., a set of rules that mathematically delineates a (usually infinite) set of finite-length strings over a (usually finite) alphabet" (*http://en.wikipedia.org/wiki/Formal_grammar*).

Business Rule Management Systems build additional value on top of a general purpose Rule Engine by providing business user focused systems for rule creation, management, deployment, collaboration, analysis and end user tools. Further adding to this value is the fast evolving and popular methodology "Business Rules Approach", which is a helping to formalize the role of Rule Engines in the enterprise.

The term Rule Engine is quite ambiguous in that it can be any system that uses rules, in any form, that can be applied to data to produce outcomes. This includes simple systems like form validation and dynamic expression engines. The book "How to Build a Business Rules Engine (2004)" by Malcolm Chisholm exemplifies this ambiguity. The book is actually about how to build and alter a database schema to hold validation rules. The book then shows how to generate VB code from those validation rules to validate data entry. While a very valid and useful topic for some, this caused quite a surprise to this author, unaware at the time in the subtleties of Rules Engines' differences, who was hoping to find some hidden secrets to help improve the Drools engine. JBoss jBPM uses expressions and delegates in its Decision nodes which control the transitions in a Workflow. At each node it evaluates, there is a rule set that dictates the transition to undertake, and so this is also a Rule Engine. While a Production Rule System is a kind of Rule Engine and also an Expert System, the validation and expression evaluation Rule Engines mentioned previously are not Expert Systems.

A Production Rule System is Turing complete, with a focus on knowledge representation to express propositional and first order logic in a concise, non-ambiguous and declarative manner. The brain of a Production Rules System is an Inference Engine that is able to scale to a large number of rules and facts. The Inference Engine matches facts and data against Production Rules - also called Productions or Rules - to infer conclusions which result in actions. A Production Rule is a two-part structure using First Order Logic for reasoning over knowledge representation.

```
when
 <conditions>
then
 <actions>
```

The process of matching the new or existing facts against Production Rules is called Pattern Matching, which is performed by the Inference Engine. There are a number of algorithms used for Pattern Matching by Inference Engines including:

- Linear
- Rete
- Treat
- Leaps

Drools now implements and extends the Rete algorithm, instead of the Leaps Algorithm. Leaps is not longer supported due to poor maintenance. The Drools Rete implementation is called **`ReteOO`**, an enhanced and optimized implementation of the Rete algorithm for Object Oriented systems. The most common enhancements to Rete based systems are covered in "Production Matching for Large Learning Systems (Rete/UL)"(1995) by Robert B. Doorenbos.

The Rules are stored in the Production Memory and the facts that the Inference Engine matches against the Working Memory. Facts are asserted into the Working Memory where they may then be modified or retracted. A system with a large number of rules and facts may result in many rules being true for the same fact assertion, these rules are said to be in conflict. The Agenda manages the execution order of these conflicting rules using a Conflict Resolution strategy.

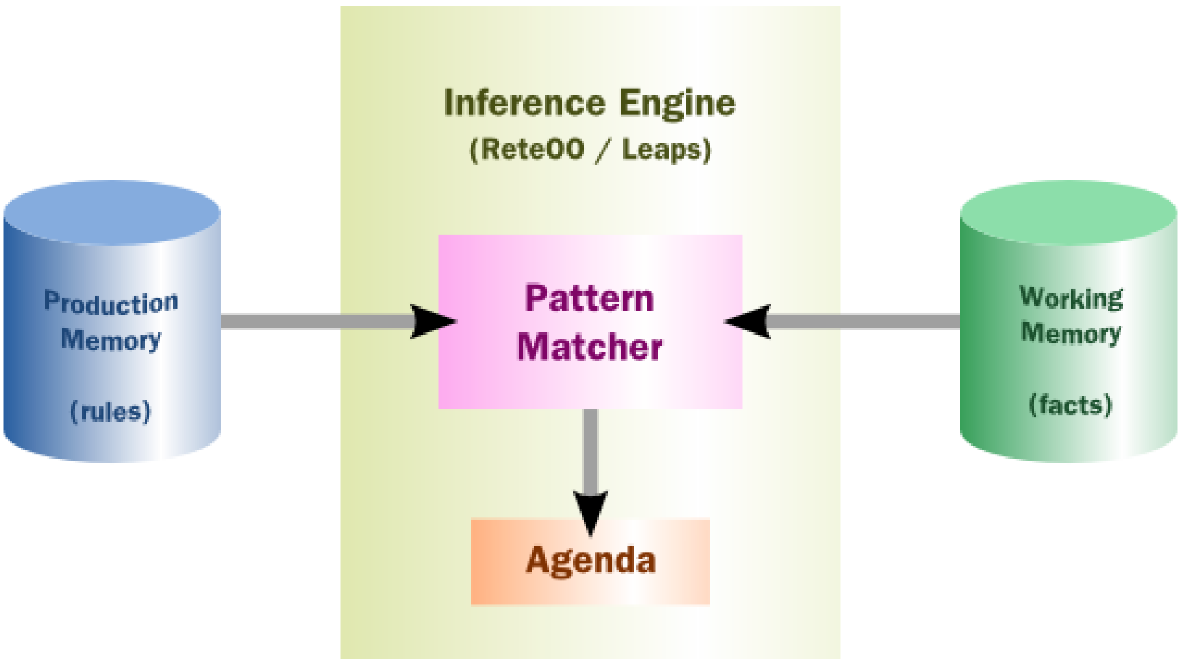


Figure 1.1. High-level View of a Rule Engine

There are two methods of execution for a Production Rule Systems, *Forward Chaining* and *Backward Chaining*. Systems that implement both methods are called *Hybrid Production Rule Systems*.

Understanding these two modes of operation are key to understanding why a Production Rule System is different and how to optimise them.

Forward Chaining

Forward chaining is 'data-driven', it reacts to data presented to it. Facts are inserted into the working memory which results in one or more rules being true and scheduled for execution by the *Agenda*.

Drools is a Forward Chaining engine.

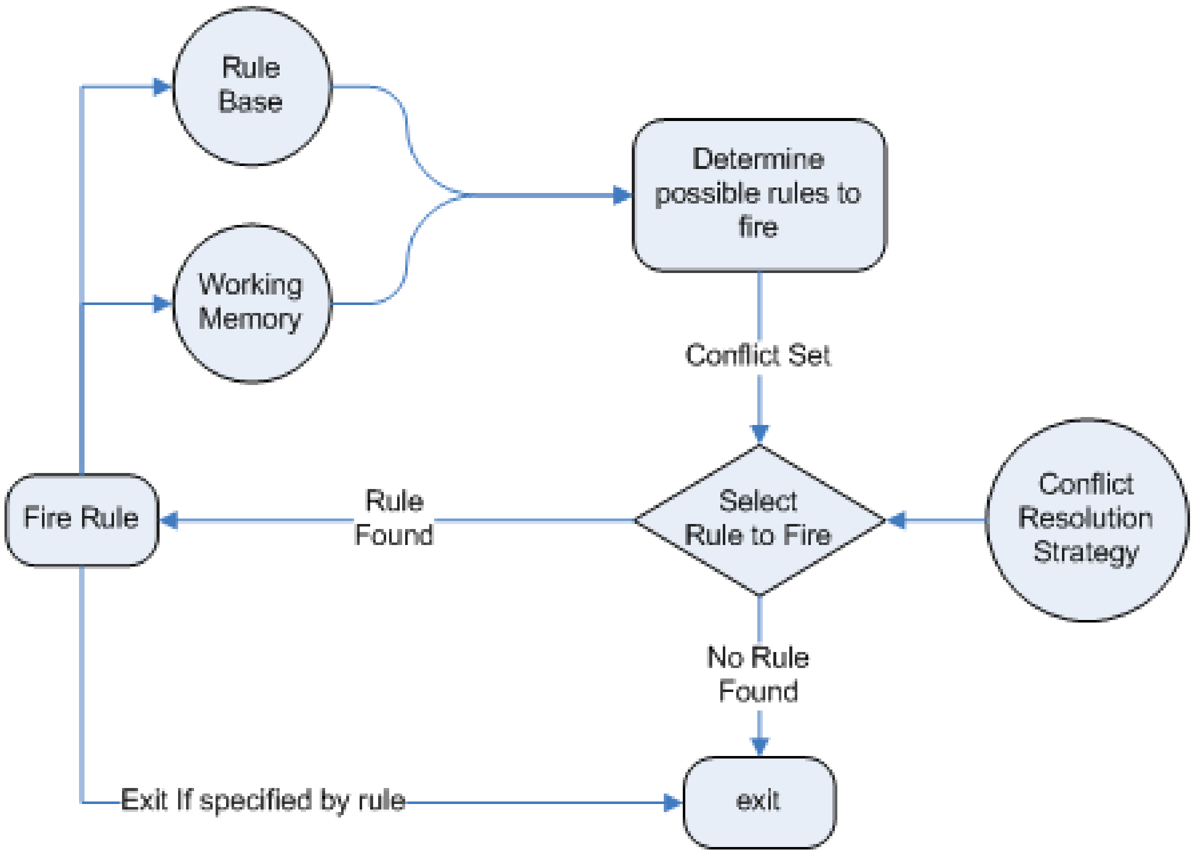


Figure 1.2. Forward Chaining

Backward Chaining

Backward chaining is 'goal-driven'. The system starts with a *conclusion* which the engine tries to satisfy. If this conclusion cannot be satisfied the engine searches for *sub goals*, conclusions that will help satisfy a part of the current goal. It continues this process until either the initial conclusion is satisfied or there are no more unsatisfied sub goals. Prolog is an example of a Backward Chaining engine.

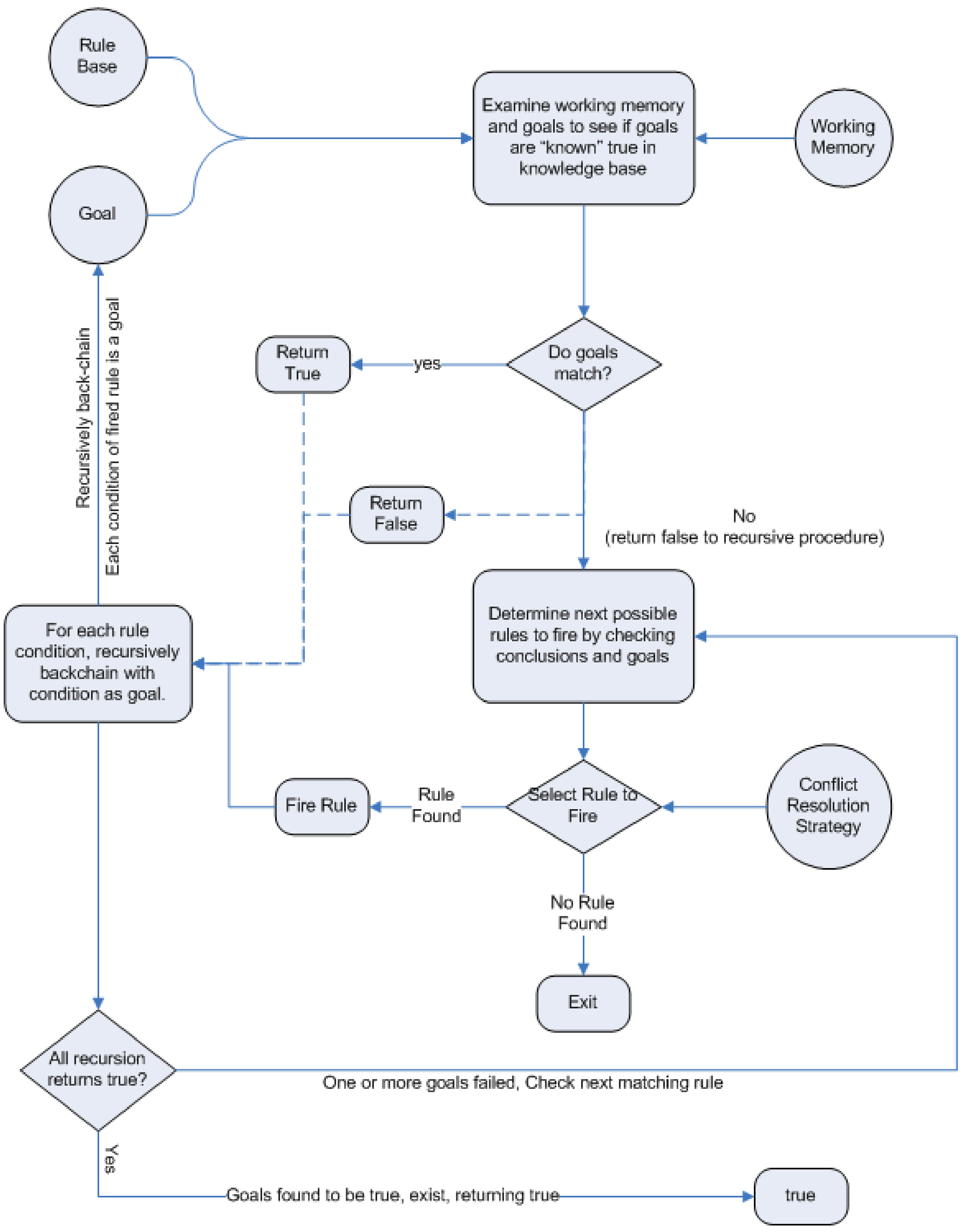

Rule Base
Goal
Examine working memory and goals to see if goals are "known" true in knowledge base
Working Memory
Recursively back-chain
Each condition of fired rule is a goal
Return True
yes
Do goals match?
Return False
No (return false to recursive procedure)
Determine next possible rules to fire by checking conclusions and goals
For each rule condition, recursively backchain with condition as goal.
Fire Rule
Rule Found
Select Rule to Fire
Conflict Resolution Strategy
No Rule Found
Exit
All recursion returns true?
One or more goals failed, Check next matching rule
Yes
Goals found to be true, exist, returning true
true

Figure 1.3. Backward Chaining

Support for Backward Chaining is planned for a future release of JBoss Rules.

1.2. Why use a Rule Engine?

The most frequently asked questions regarding Rules Engines are:

1. When should you use a rule engine?
2. What advantage does a rule engine have over hand coded "if...then" approaches?
3. Why should you use a rule engine instead of a scripting framework, like BeanShell?

We will attempt to address these questions below.

1.2.1. Advantages of a Rule Engine

- Declarative Programming

  Rule engines allow you to say "What to do", not "How to do it".

  Using rules can make it very easy to express solutions to difficult problems and consequently have those solutions verified. Declarative rules are much easier to read then imperative code.

  Rule systems are not only capable of solving very hard problems but also providing an explanation of how the solution was arrived at and why each decision along the way was made. This is not easy with other AI systems like neural networks.

- Logic and Data Separation

  Your data is in your domain objects, the logic is in the rules. This is a fundamental break from the object-orientated coupling of data and logic, which can be an advantage or a disadvantage depending on your point of view. The advantage is that the logic can be much easier to maintain when there are changes in the future, because it is all laid out in rules. This can be especially true if the logic is cross-domain or multi-domain logic. Instead of the logic being spread across many domain objects or controllers, it can all be organized in one or more very distinct rules files.

- Speed and Scalability

  The Rete algorithm,the Leaps algorithm, and their descendants such as Drools' ReteOO, provide very efficient ways of matching rule patterns to your domain object data. These are especially efficient when you have data sets that only change very slightly each time as the rule engine can remember past matches. These algorithms are battle proven.

- Centralization of Knowledge

  By using rules, you create a repository of knowledge (a knowledge base) which is executable. This means it's a single point of truth, for business policy for instance. Ideally rules are so readable that they can also serve as documentation.

- Tool Integration

  Tools such as Eclipse and Web based user interfaces such as the JBoss Enterprise BRMS Platform provide ways to edit and manage rules and get immediate feedback, validation and content assistance. Auditing and debugging tools are also available.

- Explanation Facility

  Rule systems can provide an "explanation facility" by logging the decisions made by the rule engine along with why the decisions were made.

- Understandable Rules

By creating object models and Domain Specific Languages that model your problem domain effectively you can write rules that look very close to natural language. These rules can be very understandable to non-technical domain experts.

1.2.2. When should you use a Rule Engine?

The shortest answer to this is "when there is no satisfactory traditional programming approach to solve the problem.". Given that short answer, some more explanation is required. The reason why there is no "traditional" approach is possibly one of the following:

- The problem is just too complex for traditional code.

  The problem may not be complex, but you can't see a robust way of building it.

- There are no obvious traditional solutions or the problem isn't fully understood.

- The logic changes often

  The logic itself may be simple but the rules change quite often. In many organizations software releases are rare and rules can help provide the "agility" that is needed and expected in a reasonably safe way.

- Domain experts and business analysts are readily available, but are nontechnical.

  Domain experts possess a wealth of knowledge about business rules and processes. They typically are nontechnical, but can be very logical. Rules can allow them to express the logic in their own terms. Of course, they still have to think critically and be capable of logical thinking. Many people in nontechnical positions do not have training in formal logic, so be careful and work with them, as by codifying business knowledge in rules, you will often expose problems with how the business rules and processes are currently understood.

If rules are a new technology for your project teams, the overhead in getting going must be factored in. It is not a trivial technology, but this document tries to make it easier to understand.

Typically you would use a rule engine to separate key parts of your business logic from your application. This is in opposition to the object-orientated (OO) concept of encapsulating all the logic inside your objects. This does not mean that you throw out OO practices away as business logic is only one part of your application. However you should consider a rule engine if your application code is becoming increasing complicated by conditionals (if, else, switch), excessive strategy patterns or other business logic that requires frequent change. If you are faced with tough problems of which there are no algorithms or patterns for, consider using rules.

Rules could be used embedded in your application or perhaps as a service. Often a rule engine works best as "stateful" component, being an integral part of an application. However, there have been successful cases of creating reusable rule services which are stateless.

For your organization it is important to decide about the process you will use for updating rules in systems that are in production. The options are many, but different organizations have different requirements. Often, rules maintenance is out of the control of the application vendors or project developers.

1.2.3. When not to use a Rule Engine

Rules engines are not designed to handle workflow or process executions. In the excitement of working with rules engines, that people sometimes forget that a rules engine is only one piece of a complex application or solution.

In some organizations rule engines are seen as a way of being able to update an application's behavior without the complications of having to formally re-deploy the application within their enterprise. In such circumstances it should be considered that rule engines work most effectively when the rules can be written in a declarative manner. If this cannot be done then you should consider alternative solutions such as data-driven designs, scripting engines or process engines.

Data-driven systems store meta-data that changes your applications behavior. These can work well when the control can remain relatively limited. However they often either grow to complex to maintain if extended too much or cause the application to stagnate as they are too inflexible.

Scripting engines separate your imperative business logic from your application. Your business logic is usually written in a simpler scripting language that does not need to be compiled. They are easy to implement and are a familiar environment for many imperative programmers. The downside of scripting engines is that you have created a tight coupling of your application to the scripts and such imperative scripts can easily grow in complexity and become difficult to maintain. When evaluating rule engines you may notice that some rule engines are really scripting engines.

Process and Workflow Engines such as jBPM allow you to graphically or programmatically describe steps in a process. Those steps can also involve decision points which can be considered simple rules. Process and rule engines complement each other very well, so they are not mutually exclusive.

1.2.4. Strong and Loose Coupling

No doubt you have heard terms like "tight coupling" and "loose coupling" in systems design. Generally people assert that "loose" or "weak" coupling is preferable in design terms, due to the added flexibility it affords. Similarly, you can have "strongly coupled" and "weakly coupled" rules. Strongly coupled in this sense means that one rule "firing" will clearly result in another rule firing, and so on; in other words, there is a clear (probably obvious) chain of logic. If your rules are all strongly coupled, the chances are that the will turn out to be inflexible, and, more significantly, that a rule engine is an overkill. A clear chain can be hard coded, or implemented using a Decision Tree. This is not to say that strong coupling is inherently bad, but it is a point to keep in mind when considering a rule engine and the way you capture the rules. "Loosely" coupled rules should result in a system that allows rules to be changed, removed and added without requiring changes to other, unrelated rules.

Quick Start

2.1. The Basics

For beginners JBoss Rules can be overwhelming because there is so much functionality provided to deal with the many use cases. The purpose of this chapter is to introduce the basics with some very simple examples.

2.1.1. Stateless Knowledge Session

A stateless session without inference is the simplest use case. A stateless session can be called like a function, passing it some data and then receiving some results back. Some common use cases for stateless sessions are, but not limited to:

- Validation, e.g. Is this person eligible for a mortgage?
- Calculation, e.g. Compute a mortgage premium.
- Routing and Filtering, e.g. Filtering incoming messages, such as emails, into folders or sending incoming messages to a destination.

So let's start with a simple example using a driving license application. First of all we need our data, the *fact* that will pass to our rule.

```
package com.company.license;

public class Applicant
{
    private String name;
    private int age;
    private boolean valid;
    // getter and setter methods here
}
```

Now that we have our data model we can write our first rule. This rule will perform a simple validation to disqualify any applicant younger than 18.

```
package com.company.license;

rule "Is of valid age"
when
    $a : Applicant( age < 18 )
then
    $a.setValid( false );
end
```

When the Applicant object is inserted into the rule engine it is evaluated against the constraints of each rule to see it it matches any of them. There is always an implied constraint of object type and then any number of explicit field constraints. The constraints are referred to as a *pattern* and this process is often referred to as *pattern matching*. When an inserted object satisfies all the constraints of a rule it is said to be *matched*.

In the "Is of valid age" rule we have two constraints:

1. the fact being matched against must be of type Applicant

2. the value of age must be less than 18, and

The **`$a`** is a binding variable which allows the matched object to be referenced in the rule's consequence where its properties can be updated. The dollar character ('$') is optional, but it helps to differentiate variable names from field names.

Let's assume that the rules are in the same folder as the classes, so we can use the classpath resource loader to build our first Knowledge Base. A Knowledge Base is what we call our collection of compiled rules, which are compiled using the **`KnowledgeBuilder`**.

```
KnowledgeBuilder kbuilder = KnowledgeBuilderFactory.newKnowledgeBuilder();
kbuilder.add( ResourceFactory.newClassPathResource(
 "licenseApplication.drl", getClass() ), ResourceType.DRL );
if ( kbuilder.hasErrors() ) {
    System.err.println( builder.getErrors().toString() );
}
```

The above code looks on the classpath for the **`licenseApplication.drl`** file, using the method `newClassPathResource()`. The **`ResourceType`** is DRL, for Drools Rule Language. Once the **`DRL`** file has been added we can check the **`KnowledgeBuilder`** for any errors. If there are no errors, we are now ready to build our session and execute against some data.

We then execute the data against the rules. Since the applicant is under the age of 18, the application is marked as invalid.

```
KnowledgeBase kbase = KnowledgeBaseFactory.newKnowledgeBase();
kbase.addKnowledgePackages( builder.getKnowledgePackages() );
StatelessKnowledgeSession ksession = kbase.newStatelessKnowledgeSession();
Applicant applicant = new Applicant( "Mr John Smith", 16 );
assertTrue( applicant.isValid() );
ksession.execute( applicant );
assertFalse( applicant.isValid() );
```

So far the data has only been a single object, but what if we want to use more than one? We can execute against any object implementing `Iterable`, such as a collection. Let's add another class called **`Application`**, which has the date of the application, and we'll also move the Boolean field valid to the **`Application`** class.

```
public class Applicant {
    private String name;
    private int age;
    // getter and setter methods here
}

public class Application {
    private Date dateApplied;
    private boolean valid;
    // getter and setter methods here
}
```

We can also add another rule to validate that the application was made within a period of time.

```
package com.company.license

rule "Is of valid age"
when
    Applicant( age < 18 )
    $a : Application()
then
    $a.setValid( false );
end

rule "Application was made this year"
when
    $a : Application( dateApplied > "01-jan-2009" )
then
    $a.setValid( false );
end
```

Because Java arrays do not implement the `Iterable` interface, so we have to use the Array method `asList()`. The code shown below executes against an **`Iterable`**, where all collection elements are inserted before any matched rules are fired.

```
StatelessKnowledgeSession ksession = kbase.newStatelessKnowledgeSession();
kbase.addKnowledgePackages( kbuilder.getKnowledgePackages() );
Applicant applicant = new Applicant( "Mr John Smith", 16 );
Application application = new Application();
assertTrue( application() );
ksession.execute( Arrays.asList( new Object[] {application, applicant} ));
assertFalse( application() );
```

The methods `execute(Object object)` and `execute(Iterable objects)` are actually wrapper methods for the `execute(Command command)` method from the interface `BatchExecutor`.

The **`CommandFactory`** is used to create commands, so that the following is equivalent to `execute( Iterable it )`:

```
ksession.execute(
  CommandFactory.newInsertIterable( new Object[] {application,applicant} )
);
```

`BatchExecutor` and **`CommandFactory`** are particularly useful when working with multiple Commands and output identifiers for results.

```
List<Command> cmds = new ArrayList<Command>();
cmds.add( CommandFactory.newInsert(new Person("Mr John Smith"), "mrSmith");
cmds.add( CommandFactory.newInsert( new Person( "Mr John Doe" ), "mrDoe" );
BatchExecutionResults results =
   ksession.execute( CommandFactory.newBatchExecution( cmds ) );
assertEquals(new Person( "Mr John Smith" ), results.getValue( "mrSmith" ));
```

`CommandFactory` supports many other Commands that can be used in the `BatchExecutor` like `StartProcess`, `Query`, and `SetGlobal`.

2.1.2. Stateful Knowledge Session

Stateful sessions are long lived and allow iterative changes to facts over time. Some common use cases for stateful sessions are:

- Monitoring, e.g. Stock market monitoring and analysis for semi-automatic buying.
- Diagnostics, e.g. Fault finding, medical diagnostics
- Logistics, e.g. Parcel tracking and delivery provisioning
- Compliance, e.g. Validation of legality for market trades.

Unlike a stateless session the `dispose()` method must be called afterwards to ensure there are no memory leaks, as the **`KnowledgeBase`** contains references to **`StatefulKnowledgeSessions`** when they are created. **`StatefulKnowledgeSession`** also supports the `BatchExecutor` interface. Unlike **`StatelessKnowledgeSession`**, the `FireAllRules()` command is not automatically called at the end of a stateful session.

We will illustrate the monitoring use case with an example of raising a fire alarm. Our model represents rooms in a house, each of which has one sprinkler. A fire can start in any of the rooms.

```
public class Room
{
 private String name
  // getter and setter methods here
}

public classs Sprinkler
{
 private Room room;
 private boolean on;
 // getter and setter methods here
}

public class Fire
{
 private Room room;
 // getter and setter methods here
}

public class Alarm
{
}
```

In the previous section on stateless sessions the concepts of inserting and matching against data was introduced with a single object and literal constraints. Now that we have more than one piece of data the rules must express the relationships between those objects, such as a sprinkler being in a certain room. This is done by using a binding variable as a constraint in a pattern. This results in what is called cross products, which are discussed more fully in *Section 2.2.2, “Cross Products”*.

When a fire occurs an instance of the **`Fire`** class is created, for that room, and inserted into the session. The rule uses a binding on the room field of the **`Fire`** object to constrain matching to the Sprinkler for that room, which is currently off. When this rule fires and the consequence is executed the sprinkler is turned on.

The rule from the stateless session example used standard Java syntax to update a field. In this rule we we use the `modify` statement.

```
rule "When there is a fire turn on the sprinkler"
when
    Fire($room : room)
    $sprinkler : Sprinkler( room == $room, on == false )
then
    modify( $sprinkler ) { setOn( true ) };
    System.out.println("Turn on the sprinkler for room "+$room.getName());
end
```

The `modify` statement may contain a series of comma separated Java expressions, calls to methods of the object selected by the `modify` statement. This modifies the data, and makes the engine aware of those changes so it can be matched against the rules again. This process is called *inference* and is essential for the working of a stateful session.

Stateless sessions typically do not use inference. Inference can also be turned off explicitly by specifying "sequential mode". Refer to *Section 3.3.7.1, “Sequential Mode”* for additional information.

So far we have rules that tell us when matching data exists, but what about when it does *not* exist? How do we determine that a Fire has been extinguished? The `not` keyword matches when something does not exist.

The rule below turns the Sprinkler off when the Fire in that room has disappeared.

```
rule "When the fire is gone turn off the sprinkler"
when
    $room : Room( )
    $sprinkler : Sprinkler( room == $room, on == true )
    not Fire( room == $room )
then
    modify( $sprinkler ) { setOn( false ) };
    System.out.println("Turn off the sprinkler for room "+$room.getName());
end
```

There is just a single Alarm for the building. An Alarm is created when a Fire occurs, but only one Alarm is needed for the entire building regardless of the number of Fires. The keyword `exists` matches one or more instances of a fact.

```
rule "Raise the alarm when we have one or more fires"
when
    exists Fire()
```

```
then
    insert( new Alarm() );
    System.out.println( "Raise the alarm" );
end
```

When there are no Fires we want to remove the alarm.

```
rule "Cancel the alarm when all the fires have gone"
when
    not Fire()
    $alarm : Alarm()
then
    retract( $alarm );
    System.out.println( "Cancel the alarm" );
end
```

Finally there is a general health status message, that is printed when the application first starts and after the Alarm is removed and all Sprinklers have been turned off.

```
rule "Status output when things are ok"
when
    not Alarm()
    not Sprinkler( on === true )
then
    System.out.println( "Everything is ok" );
end
```

The above rules should be placed in a single file and saved in the classpath using the file name **`fireAlarm.drl`**. We can then build a KnowledgeBase using this new DRL file. This time however, we will create a stateful session.

```
KnowledgeBuilder kbuilder = KnowledgeBuilderFactory.newKnowledgeBuilder();
kbuilder.add( ResourceFactory.newClasspathResource( "fireAlarm.drl",
          getClass() ), ResourceType.DRL );
if ( kbuilder.hasErrors() )
 System.err.println( builder.getErrors().toString() );

StatefulKnowledgeSession ksession = kbase.newStatefulKnowledgeSession();
```

Once the session is created you can iteratively work with it over time. Four Rooms are created and inserted, a Sprinkler for each room is also inserted. At this point the engine has done all of its matching, but no rules have fired yet. Calling `fireAllRules()` on the **`ksession`** allows the matched rules to fire

```
String[] names = new String[]{"kitchen","bedroom","office","livingroom"};
Map<String,Room> name2room = new HashMap<String,Room>();

for( String name: names )
{
 Room room = new Room( name );
 name2room.put( name, room );
 ksession.insert( room );
 Sprinkler sprinkler = new Sprinkler( room );
 ksession.insert( sprinkler );
}

ksession.fireAllRules();
```

Of course without a fire this simply produces the health message.

```
> Everything is ok
```

We now create two fires and insert them, this time a reference is kept for the returned **`FactHandle`**. The **`FactHandle`** is an internal engine reference to the inserted object and allows objects to be retracted or modified at a later point in time. With the **`Fires`** now in the engine, once `fireAllRules()` is called, the Alarm is raised and the respective Sprinklers are turned on.

```
Fire kitchenFire = new Fire( name2room.get( "kitchen" ) );
Fire officeFire = new Fire( name2room.get( "office" ) );

FactHandle kitchenFireHandle = ksession.insert( kitchenFire );
FactHandle officeFireHandle = ksession.insert( officeFire );

ksession.fireAllRules();
```

```
> Raise the alarm
> Turn on the sprinkler for room kitchen
> Turn on the sprinkler for room office
```

After a while the fires will be put out and the Fire objects are retracted. This results in the Sprinklers being turned off, the Alarm being cancelled, and eventually the health message is printed again.

```
ksession.retract( kitchenFireHandle );
ksession.retract( officeFireHandle );

ksession.fireAllRules();
```

```
> Turn on the sprinkler for room office
> Turn on the sprinkler for room kitchen
> Cancel the alarm
> Everything is ok
```

2.2. A Little Theory

2.2.1. Methods versus Rules

New users often confuse methods and rules, and ask the question "How do I call a rule?" The previous section should have cleared up that confusion, but we summarize the differences below.

```
public void helloWorld(Person person)
{
 if ( person.getName().equals( "Chuck" ) )
 {
     System.out.println( "Hello Chuck" );
 }
}
```

- Methods are called directly.
- Specific instances are passed.
- One call results in a single execution.

```
rule "Hello World"
when
    Person( name == "Chuck" )
then
    System.out.println( "Hello Chuck" );
    end
```

- Rules execute by matching against any data as long it is inserted into the engine.
- Rules can never be called directly.
- Specific instances cannot be passed to a rule.
- Depending on the matches, a rule may fire once or several times, or not at all.

2.2.2. Cross Products

A cross product is the result of combining 2 or more sets of data. Consider the following rule for the fire alarm example.

```
rule "show sprinklers in rooms"
when
    $room : Room()
    $sprinkler : Sprinkler()
then
    System.out.println( "room:" + $room.getName() +
        " sprinkler:" + $sprinkler.getRoom().getName() );
end
```

The output from this rule will contain every combination of Room and Sprinkler, including those that make no sense.

```
room:office sprinker:office
room:office sprinkler:kitchen
room:office sprinkler:livingroom
room:office sprinkler:bedroom
room:kitchen sprinkler:office
```

```
room:kitchen sprinkler:kitchen
room:kitchen sprinkler:livingroom
room:kitchen sprinkler:bedroom
room:livingroom sprinkler:office
room:livingroom sprinkler:kitchen
room:livingroom sprinkler:livingroom
room:livingroom sprinkler:bedroom
room:bedroom sprinkler:office
room:bedroom sprinkler:kitchen
room:bedroom sprinkler:livingroom
room:bedroom sprinkler:bedroom
```

This cross products can obviously become huge and cause performance problems. To prevent this you can eliminate the nonsensical results by using variable constraints.

```
rule "show sprinklers in rooms"
when
    $room : Room()
    $sprinkler : Sprinkler( room == $room )
then
    System.out.println( "room:" + $room.getName() +
                        " sprinkler:" + $sprinkler.getRoom().getName() );
end
```

This results in just four rows of data, with the correct Sprinkler for each Room.

```
room:office sprinkler:office
room:kitchen sprinkler:kitchen
room:livingroom sprinkler:livingroom
room:bedroom sprinkler:bedroom
```

Note

This is the same technique that is used in SQL when performing "joins", e.g. `select * from Room, Sprinkler where Sprinkler.room_id = Room.id`

2.2.3. Activations, Agenda and Conflict Sets

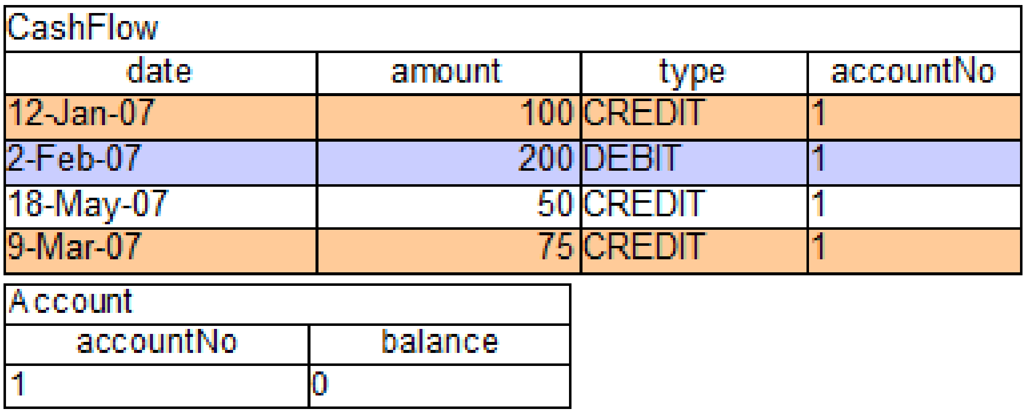
So far the data and the matching process has been simple and small. Once you have many rules and many facts being inserted over time the rule engine needs a way to manage the execution of the rule outcomes. JBoss Rules achieves this using *Activations*, *Agendas*, and a *conflict resolution strategy*.

This next example explores handling cashflow calculations over date periods. It is more complex than the previous examples. It is assumed that you are comfortable with the Java code for creating **`KnowledgeBases`** and populating a **`StatefulKnowledgeSession`** with facts so that code will not be repeated here. Diagrams are used to illustrate the state of the rule engine at key stages.

Three classes, **`Cashflow`**, **`Account`** and **`AccountPeriod`** are used as the data model.

```
public class Cashflow
{
 private Date   date;
 private double amount;
 private int    type;
 long           accountNo;
 // getter and setter methods here
}

public class Account
{
 private long   accountNo;
 private double balance;
 // getter and setter methods here
}

public AccountPeriod
{
 private Date start;
 private Date end;
 // getter and setter methods here
}
```

Figure 2.1, "CashFlows and Account" shows that a single Account fact was inserted along with four CashFlow facts.



CashFlow

date	amount	type	accountNo
12-Jan-07	100	CREDIT	1
2-Feb-07	200	DEBIT	1
18-May-07	50	CREDIT	1
9-Mar-07	75	CREDIT	1

Account

accountNo	balance
1	0

Figure 2.1. CashFlows and Account

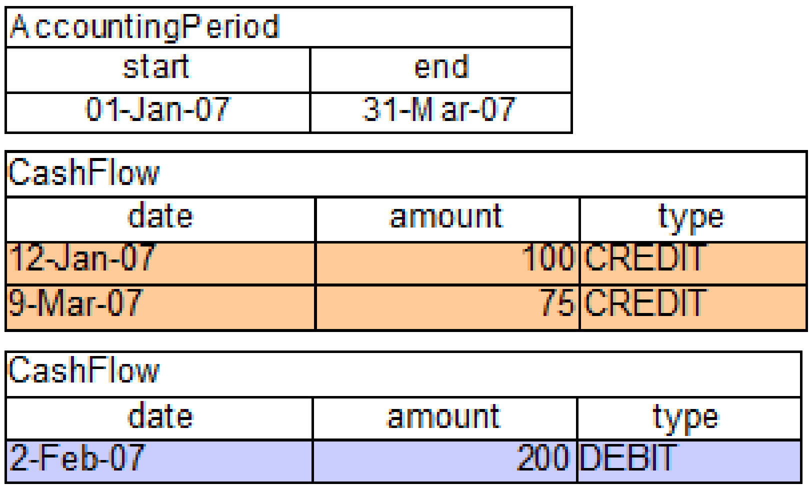
The following two rules are used to determine the debit and credit for the specified period and update the Account balance. Notice the && operator used to avoid repeating the field name.

```
rule "increase balance for credits"
when
  ap : AccountPeriod()
  acc : Account( $accountNo : accountNo )
  CashFlow( type == CREDIT,
    accountNo == $accountNo,
    date >= ap.start && <= ap.end,
    $amount : amount )
```

```
then
  acc.balance  += $amount;
end
```

```
rule "decrease balance for debits"
when
    ap : AccountPeriod()
    acc : Account( $accountNo : accountNo )
    CashFlow( type == DEBIT,
        accountNo == $accountNo,
        date &gt;= ap.start &amp;&amp; &lt;= ap.end,
        $amount : amount )
then
    acc.balance -= $amount;
end
```

As shown in *Figure 2.2, “CashFlows and Account”* the AccountPeriod start is set to the 1st of January and the end is set to the 31st of March. This constrains the data to two and one CashFlow objects for credit and debit respectively.



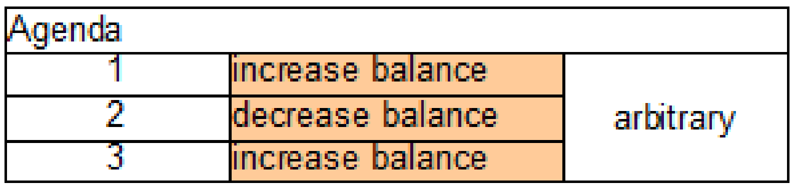
AccountingPeriod	
start	end
01-Jan-07	31-Mar-07

CashFlow		
date	amount	type
12-Jan-07	100	CREDIT
9-Mar-07	75	CREDIT

CashFlow		
date	amount	type
2-Feb-07	200	DEBIT

Figure 2.2. CashFlows and Account

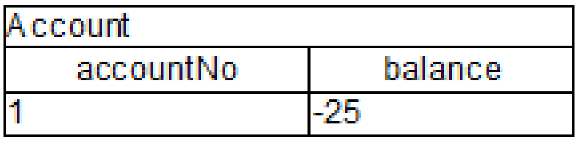
The data is matched during the insertion stage but because this is a stateful session the rules' consequences do not execute immediately. The matched rules and the corresponding data are referred to as Activations. Each Activation is added to a list called the Agenda. Each Activation on the Agenda is executed when `fireAllRules()` is called. Unless specified otherwise the Activations are executed one after another in an arbitrary order.



Agenda		
1	increase balance	arbitrary
2	decrease balance	
3	increase balance	

Figure 2.3. CashFlows and Account

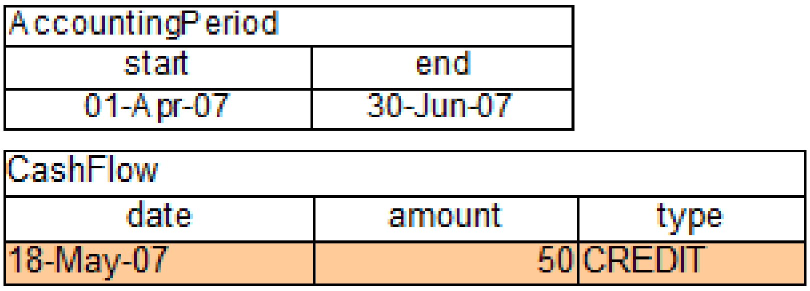
After all of the above activations are fired, the Account has a balance of -25.



Account	
accountNo	balance
1	-25

Figure 2.4. CashFlows and Account

If the AccountPeriod is updated to the second quarter, we have just a single matched row of data, and thus just a single Activation on the Agenda.

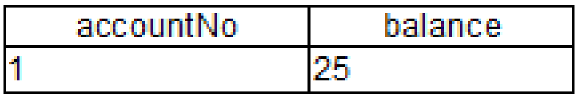


AccountingPeriod	
start	end
01-Apr-07	30-Jun-07

CashFlow		
date	amount	type
18-May-07	50	CREDIT

Figure 2.5. CashFlows and Account

The firing of that Activation results in a balance of 25.



accountNo	balance
1	25

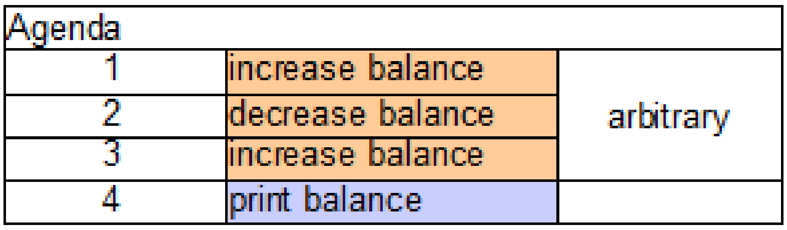
Figure 2.6. CashFlows and Account

When there is one or more Activations on the Agenda they are said to be in conflict, and a conflict resolver strategy is used to determine the order of execution. At the simplest level the default strategy uses *salience* to determine rule priority. Each rule has a default salience value of 0, the higher the value the higher the priority. Salience can also be negative. This lets you order the execution of rules relative to each other. The execution of rules with the same salience value is still arbitrary.

To illustrate this we add a rule to print the Account balance. This rule is to be executed after all the debits and credits have been applied for all accounts. It has a negative salience value so it will execute after the rules with the default salience value of 0.

```
rule "Print balance for AccountPeriod"
    salience -50
when
    ap : AccountPeriod()
    acc : Account()
then
    System.out.println( acc.accountNo + " : " + acc.balance );
end
```

The table below depicts the resulting Agenda. The three debit and credit rules are shown to be in arbitrary order, while the print rule is ranked last, to execute afterwards.



Agenda		
1	increase balance	arbitrary
2	decrease balance	
3	increase balance	
4	print balance	

Figure 2.7. CashFlows and Account

Important

JBoss Rules includes functionality for directing the flow of rules called RuleFlow. This functionality is not supported in the JBoss Enterprise BRMS Platform. Any workflow management is best delegated to a dedicated workflow engine such as JBoss jBPM.

2.3. More on building and deploying

2.3.1. Adding Rules using Changesets

The examples so far have used the JBoss Rules API to build each KnowledgeBase by manually adding each rule. JBoss Rules also provides a means to declare the rule resources to be added to a KnowledgeBase in XML. This feature is called *changesets*.

The changeset XML file contains a list of rule resources to be added to a KnowledgeBase. It may also point to another changeset XML file. Currently changesets only support the <add> element. Future versions will add support for <remove>and <modify>.

```
<change-set xmlns='http://drools.org/drools-5.0/change-set'
    xmlns:xs='http://www.w3.org/2001/XMLSchema-instance'
    xs:schemaLocation='http://drools.org/drools-5.0/change-set
    drools-change-set-5.0.xsd' >

    <add>
        <resource source='http://hostname/myrules.drl' type='DRL' />
    </add>

</change-set>
```

The source of each resource is specified using a URL. All the protocols provided by *java.net.URL* are supported. In addition the protocol classpath can be used, which refers to the current processes classpath for the resource. The type attribute must always be specified for a resource, it is not inferred from the file name extension.

To use the above XML the code is almost identical as before, except we change the ResourceType to **CHANGE_SET**.

```
KnowledgeBuilder kbuilder = KnowledgeBuilderFactory.newKnowledgeBuilder();
kbuilder.add( ResourceFactory.newClasspathResource( "myChangeSet.xml",
 getClass() ),
          ResourceType.CHANGE_SET );
if ( kbuilder.hasErrors() ) {
System.err.println( builder.getErrors().toString() );
}
```

Changesets can include any number of resources. They also support additional configuration information for decision tables. The example below loads rules from a http URL, and an Excel decision table using the classpath protocol.

```
<change-set xmlns='http://drools.org/drools-5.0/change-set'
    xmlns:xs='http://www.w3.org/2001/XMLSchema-instance'
    xs:schemaLocation='http://drools.org/drools-5.0/change-set.xsd' >
  <add>
    <resource source='http://hostname/myrules.drl' type='DRL' />
    <resource source='classpath:data/IntegrationTest.xls' type="DTABLE">
        <decisiontable-conf input-type="XLS" worksheet-name="Tables_2" />
    </resource>
  </add>
</change-set>
```

If a directory name is specified for the resource source instead of a single file all the files in the directory will be added. All the files must be of the specified type.

```
<change-set xmlns='http://drools.org/drools-5.0/change-set'
    xmlns:xs='http://www.w3.org/2001/XMLSchema-instance'
    xs:schemaLocation='http://drools.org/drools-5.0/change-set.xsd' >
    <add>
        <resource source='file://rules/' type='DRL' />
    </add>
</change-set>
```

2.3.2. Knowledge Agent

The **`KnowledgeAgent`** provides automatic loading, caching and reloading of rule resources. It's configuration is provided from a properties file. The **`KnowledgeAgent`** can update or rebuild a KnowlegeBase when the resources it uses are changed. The strategy for this is determined by the configuration given to the **`KnowledgeAgentFactory`**.

```
KnowledgeAgent kagent = KnowledgeAgentFactory.newKnowledgeAgent("MyAgent");
kagent.applyChangeSet( ResourceFactory.newUrlResource( url ) );
KnowledgeBase kbase = kagent.getKnowledgeBase();
```

The **`KnowledgeAgent`** will continosly scan all the added resources, using a default polling interval of 60 seconds. If the last modified date of the resources is changed it will rebuild the **`KnowledgeBase`** using the new resources. If a directory is specified as part of the resources then the entire contents of that directory will be scanned for changes.

Note that the previous **`KnowledgeBase`** reference will still exist and you'll have to call `getKnowledgeBase()` to access the newly built KnowledgeBase.

User Guide

3.1. Building

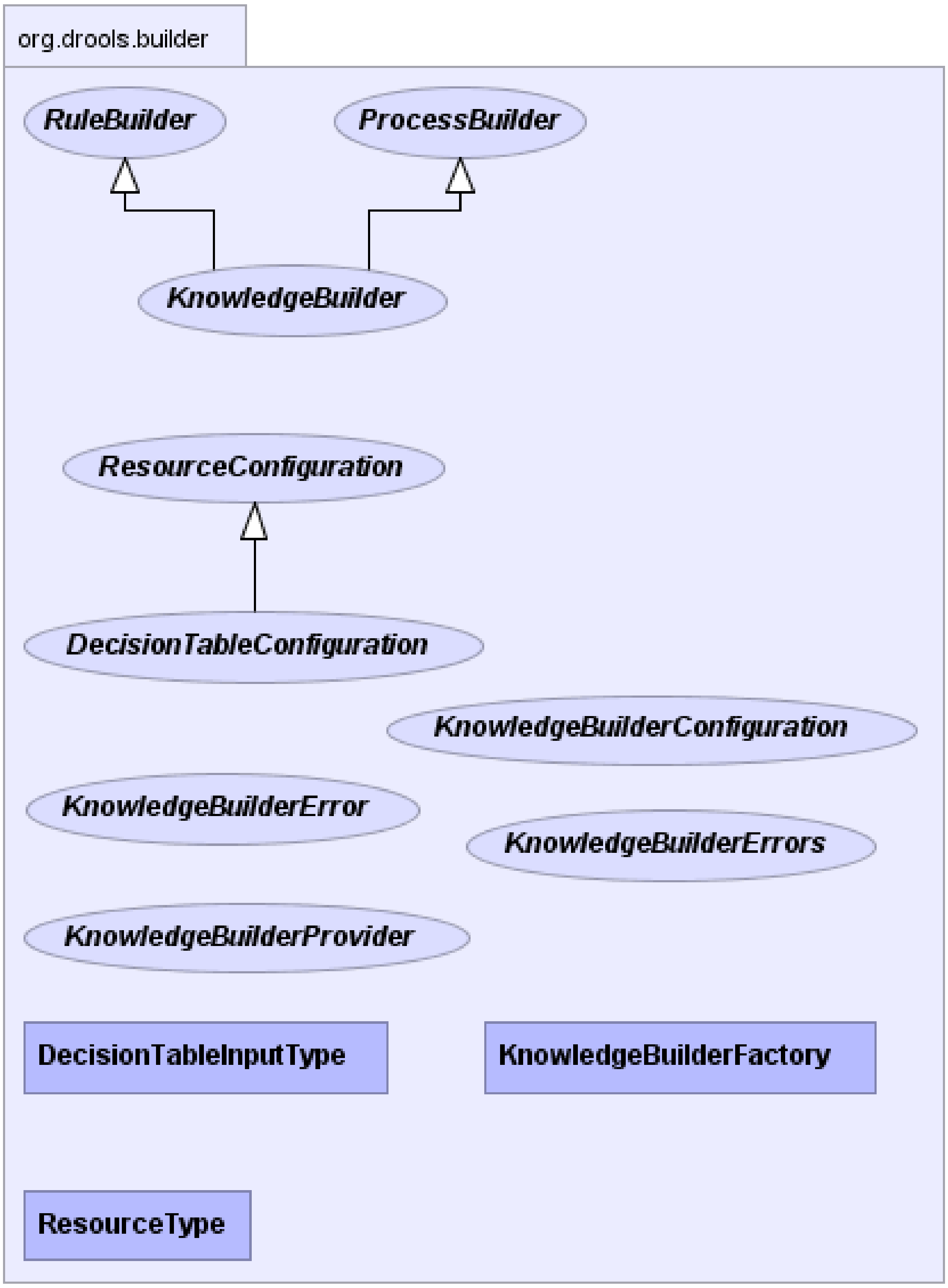


Figure 3.1. org.drools.builder

3.1.1. Building using Code

The **`KnowledgeBuilder`** is responsible for taking source files, such as a **`.drl`** file or an **`.xls`** file, and turning them into a **`KnowledgePackage`** of rule and process definitions which a **`KnowledgeBase`** can consume. An object of the class **`ResourceType`** indicates the type of resource it is being asked to build.

The ResourceFactory provides capabilities to load Resources from a number of sources, such as Reader, ClassPath, URL, File, or ByteArray. Binaries, such as .xls decision tables, should not use a Reader based Resource handler, which is only suitable for text based resources.

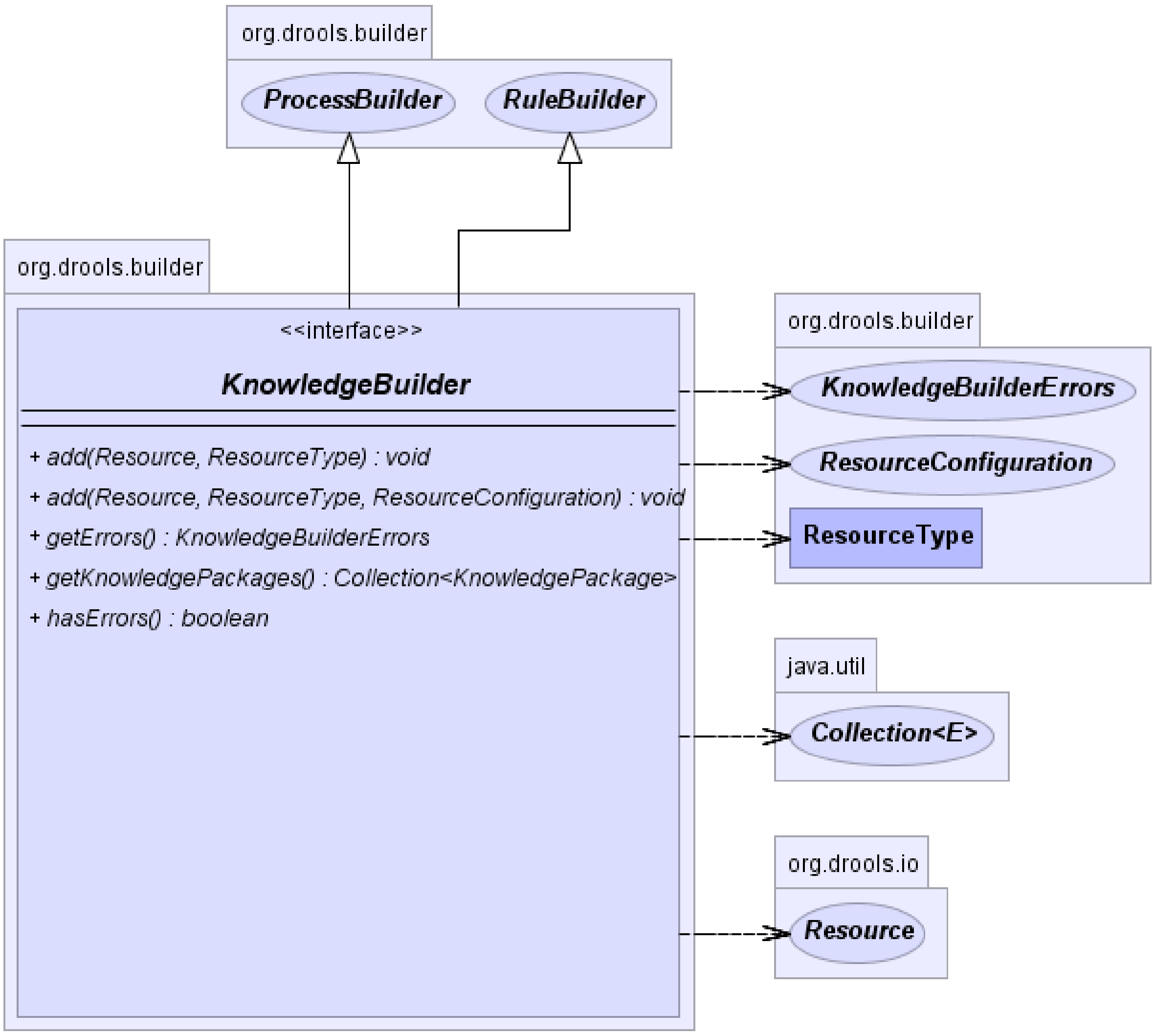


Figure 3.2. KnowledgeBuilder

The **`KnowlegeBuilder`** is created using the **`KnowledgeBuilderFactory`**.

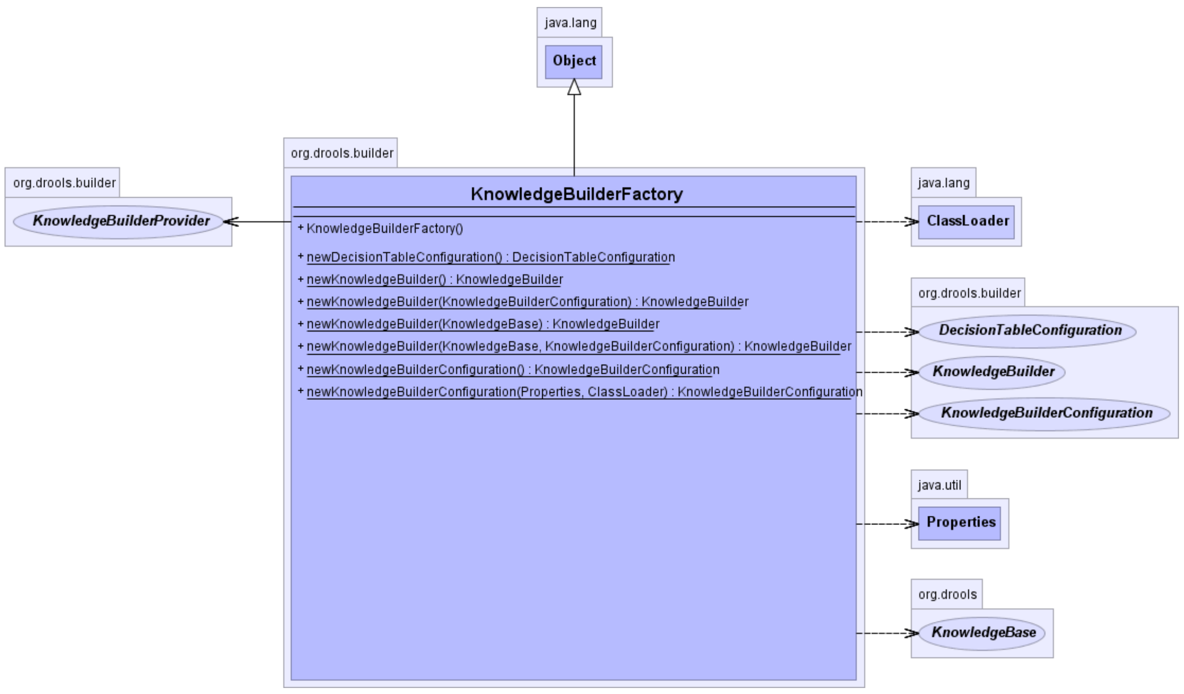

java.lang
Object
org.drools.builder
KnowledgeBuilderProvider
org.drools.builder
KnowledgeBuilderFactory
+ KnowledgeBuilderFactory()
+ newDecisionTableConfiguration() : DecisionTableConfiguration
+ newKnowledgeBuilder() : KnowledgeBuilder
+ newKnowledgeBuilder(KnowledgeBuilderConfiguration) : KnowledgeBuilder
+ newKnowledgeBuilder(KnowledgeBase) : KnowledgeBuilder
+ newKnowledgeBuilder(KnowledgeBase, KnowledgeBuilderConfiguration) : KnowledgeBuilder
+ newKnowledgeBuilderConfiguration() : KnowledgeBuilderConfiguration
+ newKnowledgeBuilderConfiguration(Properties, ClassLoader) : KnowledgeBuilderConfiguration
java.lang
ClassLoader
org.drools.builder
DecisionTableConfiguration
KnowledgeBuilder
KnowledgeBuilderConfiguration
java.util
Properties
org.drools
KnowledgeBase

Figure 3.3. KnowledgeBuilderFactory

A **`KnowledgeBuilder`** can be created using the default configuration.

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

Example 3.1. Creating a new KnowledgeBuilder

A configuration can be created using the **`KnowledgeBuilderFactory`**. This allows the behavior of the **`KnowledgeBuilder`** to modified. The most common usage is to provide a custom **`ClassLoader`** so that the **`KnowledgeBuilder`** can resolve classes that are not in the default classpath. The first parameter is for Properties and is optional and may be left null for the default options will be used. The options parameter can be used for things like changing the dialect or registering new accumulator functions.

```
KnowledgeBuilderConfiguration kbuilderConf =
 KnowledgeBuilderFactory.newKnowledgeBuilderConfiguration(null,
                classLoader );
KnowledgeBuilder kbuilder =
 KnowledgeBuilderFactory.newKnowledgeBuilder(kbuilderConf);
```

Example 3.2. Creating a new KnowledgeBuilder with a custom ClassLoader

Resources of any type can be added iteratively. Unlike **`PackageBuilder`** from earlier versions, you can use **`KnowledgeBuilder`** to add resources from multiple namespaces.

```
kbuilder.add( ResourceFactory.newFileResource( "/project/myrules.drl" ),
 ResourceType.DRL);
```

Example 3.3. Adding DRL Resources

It is best practice to always check the `hasErrors()` method after an addition. You should not add more resources or retrieve the **`KnowledgePackage`**s if there are errors. `getKnowledgePackages()` returns an empty list if there are errors.

```
if( kbuilder.hasErrors() )
{
 System.out.println( kbuilder.getErrors() );
 return;
}
```

Example 3.4. Validating

When all the resources have been added and there are no errors the collection of **`KnowledgePackage`**s can be retrieved. It is a **`Collection`** because there is one **`KnowledgePackage`** per package namespace. These **`KnowledgePackage`**s are serializable and often used as a unit of deployment.

```
Collection<KnowledgePackage> kpkgs = kbuilder.getKnowledgePackages();
```

Example 3.5. Getting the KnowledgePackages

The final example puts it all together.

```
KnowledgeBuilder kbuilder = KnowledgeBuilderFactory.newKnowledgeBuilder();
if( kbuilder.hasErrors() )
{
 System.out.println( kbuilder.getErrors() );
 return;
}

KnowledgeBuilder kbuilder = KnowledgeBuilderFactory.newKnowledgeBuilder();
kbuilder.add( ResourceFactory.newFileResource( "/project/myrules1.drl" ),
 ResourceType.DRL);
kbuilder.add( ResourceFactory.newFileResource( "/project/myrules2.drl" ),
 ResourceType.DRL);

if( kbuilder.hasErrors() )
{
 System.out.println( kbuilder.getErrors() );
 return;
}

Collection<KnowledgePackage> kpkgs = kbuilder.getKnowledgePackages();
```

Example 3.6. Putting it all together

3.1.2. Building using Configuration and the ChangeSet XML

Instead of adding the resources to create definitions programmatically, JBoss Rules also provides a means to declare the rule resources to be added to a **`KnowledgeBase`** in XML. This feature is called *changesets*.

The changeset XML file contains a list of rule resources to be added to a KnowledgeBase. It may also point to another changeset XML file. Currently changesets only support the <add> element. Future versions will add support for <remove>and <modify>.

The following example loads a single DRL file.

```
<change-set xmlns='http://drools.org/drools-5.0/change-set'
    xmlns:xs='http://www.w3.org/2001/XMLSchema-instance'
    xs:schemaLocation='http://drools.org/drools-5.0/change-set.xsd' >
    <add>
        <resource source='file:/project/myrules.drl' type='DRL' />
    </add>
</change-set>
```

Example 3.7. Simple ChangeSet XML

The source of each resource is specified using a URL. All the protocols provided by *java.net.URL* are supported. In addition the protocol classpath can be used, which refers to the current processes classpath for the resource. The type attribute must always be specified for a resource, it is not inferred from the file name extension.

Using the ClassPath resource loader in Java allows you to specify the **`ClassLoader`** to be used to locate the resource but this is not possible for resources loaded using changesets. The **`ClassLoader`** will default to the same one being used by the **`KnowledgeBuilder`** or if the ChangeSet XML is itself loaded by the ClassPath resource, in which case it will use the **`ClassLoader`** specified for that resource.

Currently you still need to use the JBoss Rules API to load that ChangeSet. Support for containers such as Spring is planned for a future release.

```
kbuilder.add(ResourceFactory.newUrlResource(url),ResourceType.CHANGE_SET);
```

Example 3.8. Loading the ChangeSet XML

Changesets can include any number of resources. They even support additional configuration information although this is currently only used for decision tables. *Example 3.9, "ChangeSet XML with resource configuration"* loads rules from a http URL location, and an Excel decision table from the classpath.

```
<change-set xmlns='http://drools.org/drools-5.0/change-set'
    xmlns:xs='http://www.w3.org/2001/XMLSchema-instance'
    xs:schemaLocation='http://drools.org/drools-5.0/change-set.xsd' >
    <add>
        <resource source='http:org/domain/myrules.drl' type='DRL' />
        <resource source='classpath:data/IntegrationExampleTest.xls'
          type="DTABLE">
          <decisiontable-conf input-type="XLS" worksheet-name="Tables_2" />
        </resource>
    </add>
</change-set>
```

Example 3.9. ChangeSet XML with resource configuration

The ChangeSet is especially useful when working with **KnowledgeAgent**, as it allows for change notification and automatic rebuilding of the **KnowledgeBase**, which is covered in more detail in the section on the **KnowledgeAgent**, under Deploying.

If a directory name is specified for the resource source instead of a single file all the files in the directory will be added. All the files must be of the specified type.

```
<change-set xmlns='http://drools.org/drools-5.0/change-set'
    xmlns:xs='http://www.w3.org/2001/XMLSchema-instance'
    xs:schemaLocation='http://drools.org/drools-5.0/change-set.xsd' >
    <add>
        <resource source='file:/projects/myproject/myrules' type='DRL' />
    </add>
</change-set>
```

Example 3.10. ChangeSet XML which adds a directories contents

Note

ChangeSets can also be used in conjunction with the **KnowledgeAgent**. Refer to *Section 3.2.6, "KnowledgeAgent"* for more information.

3.2. Deploying

3.2.1. KnowledgePackage and Knowledge Definitions

A **KnowledgePackage** is a collection of Knowledge Definitions, e.g. rules and processes. It is created by the **KnowledgeBuilder**, as described in *Section 3.1, "Building"*. **KnowledgePackage**s are self-contained and serializable. They currently form the basic deployment unit.

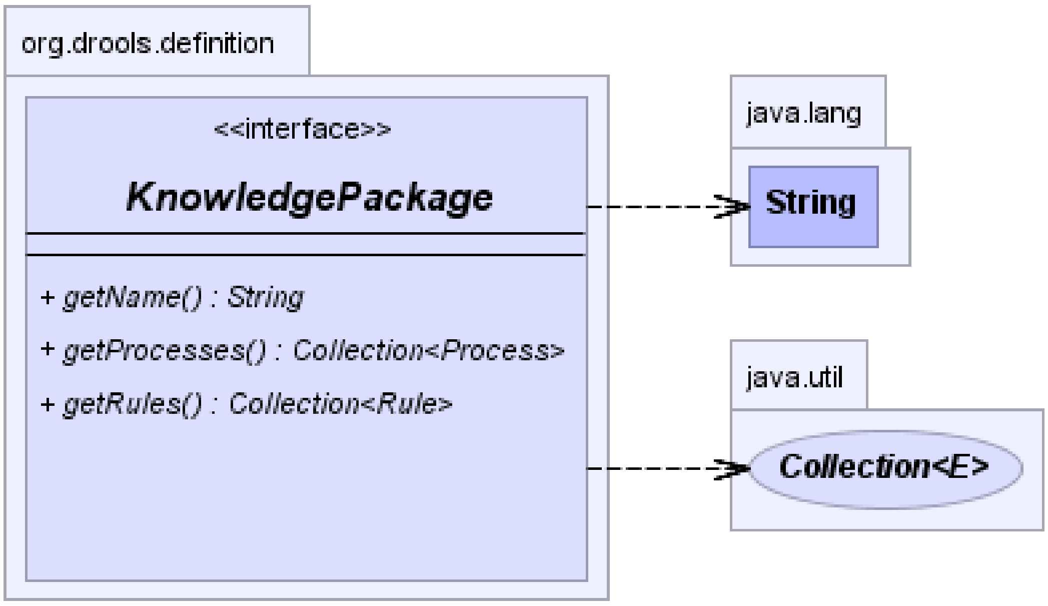


Figure 3.4. KnowledgePackage

`KnowledgePackages` are added to the **`KnowledgeBase`**. However, a **`KnowledgePackage`** instance cannot be reused once it's added to the **`KnowledgeBase`**. If you need to add it to another **`KnowledgeBase`**, try serializing it first and using the "cloned" result. This limitation will be removed in a future version of JBoss Rules.

3.2.2. KnowledgeBase

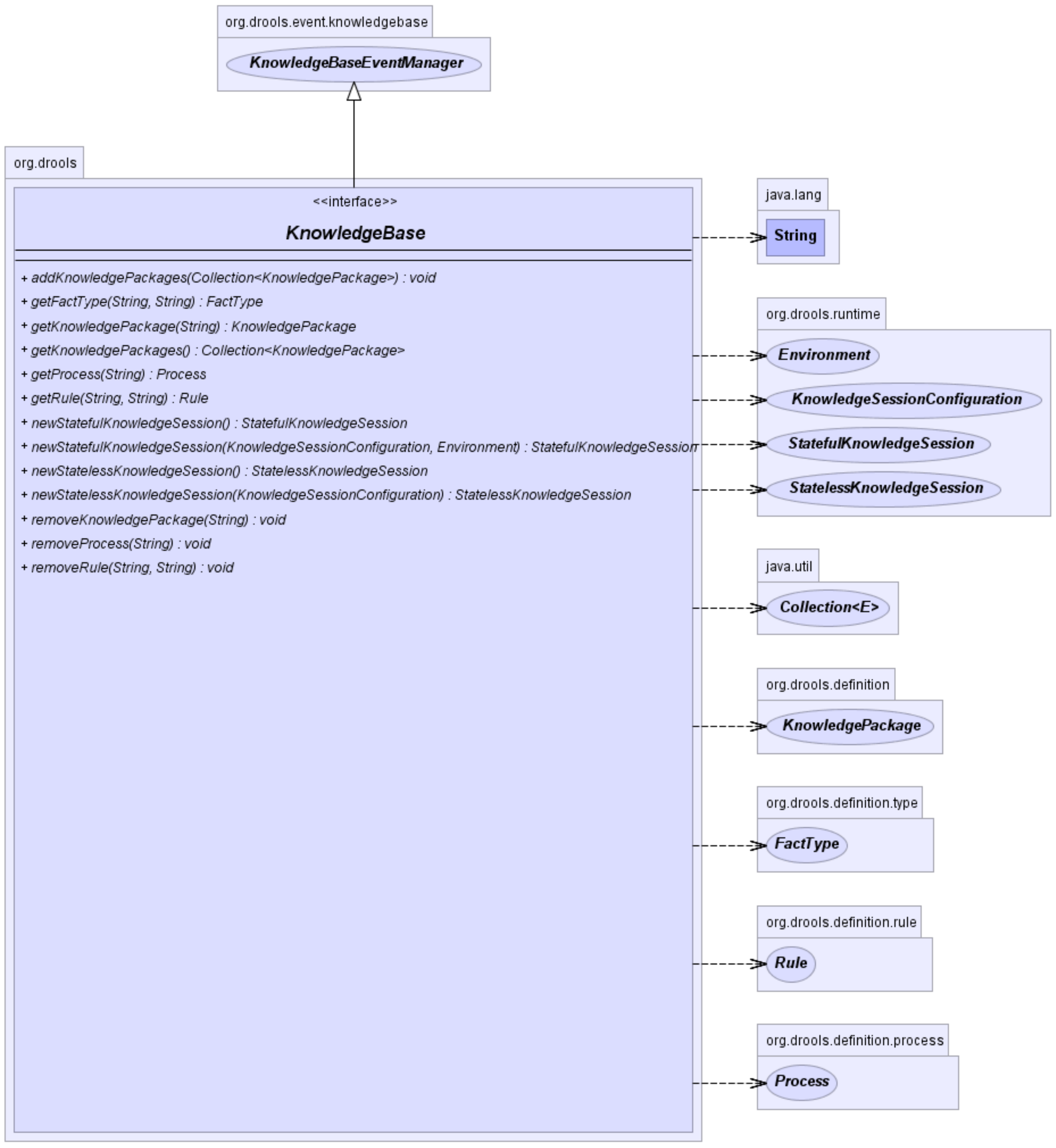

org.drools.event.knowledgebase
KnowledgeBaseEventManager
org.drools
<<interface>>
KnowledgeBase
+ addKnowledgePackages(Collection<KnowledgePackage>) : void
+ getFactType(String, String) : FactType
+ getKnowledgePackage(String) : KnowledgePackage
+ getKnowledgePackages() : Collection<KnowledgePackage>
+ getProcess(String) : Process
+ getRule(String, String) : Rule
+ newStatefulKnowledgeSession() : StatefulKnowledgeSession
+ newStatefulKnowledgeSession(KnowledgeSessionConfiguration, Environment) : StatefulKnowledgeSession
+ newStatelessKnowledgeSession() : StatelessKnowledgeSession
+ newStatelessKnowledgeSession(KnowledgeSessionConfiguration) : StatelessKnowledgeSession
+ removeKnowledgePackage(String) : void
+ removeProcess(String) : void
+ removeRule(String, String) : void
java.lang
String
org.drools.runtime
Environment
KnowledgeSessionConfiguration
StatefulKnowledgeSession
StatelessKnowledgeSession
java.util
Collection<E>
org.drools.definition
KnowledgePackage
org.drools.definition.type
FactType
org.drools.definition.rule
Rule
org.drools.definition.process
Process

Figure 3.5. KnowledgeBase

The **`KnowlegeBase`** is a repository of all the application's knowledge definitions. It may contain rules, processes, functions, and type models. The **`KnowledgeBase`** itself does not contain instance data, known as facts. Instead sessions are created from the **`KnowledgeBase`** into which data (facts) can be inserted and where process instances may be started. **`KnowlegeBase`** creation is a fairly intensive process, whereas session creation is not. It is recommended that **KnowledgeBase**s be cached where possible to allow for repeated session creation.

KnowledgeBase is serializable and it is possible to build and then store the serialized **KnowledgeBase**, treating it also as a unit of deployment, instead of the **KnowledgePackage**s.

The **KnowlegeBase** is created using the **KnowledgeBaseFactory**.

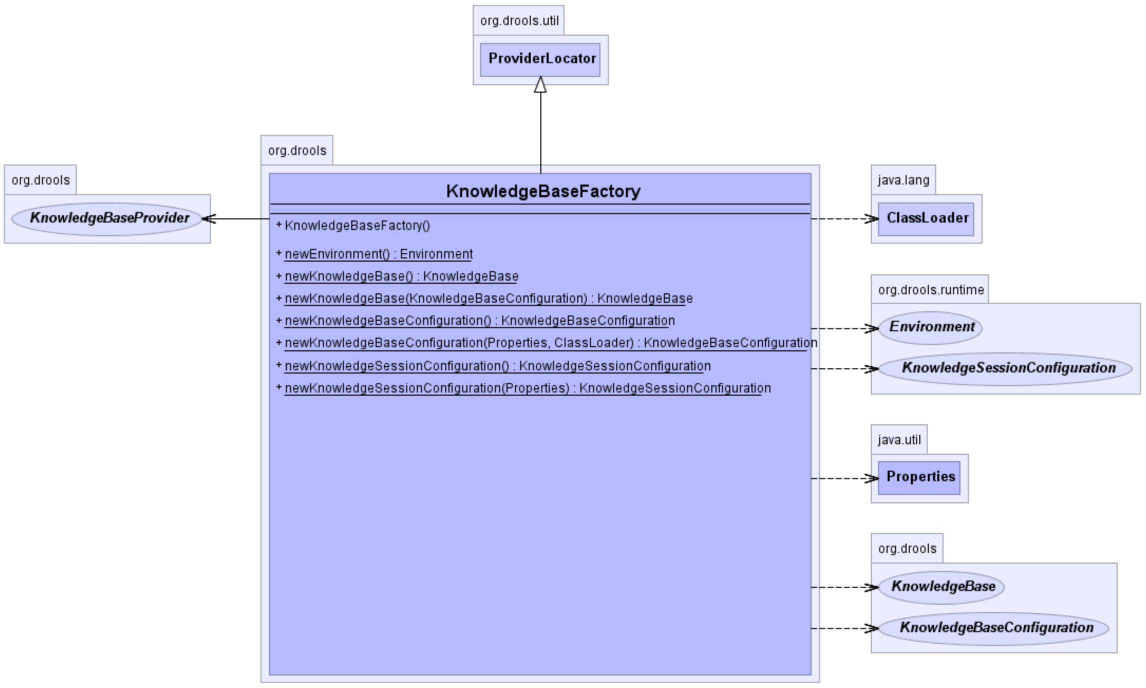

org.drools.util
ProviderLocator
org.drools
KnowledgeBaseFactory
+KnowledgeBaseFactory()
+newEnvironment() : Environment
+newKnowledgeBase() : KnowledgeBase
+newKnowledgeBase(KnowledgeBaseConfiguration) : KnowledgeBase
+newKnowledgeBaseConfiguration() : KnowledgeBaseConfiguration
+newKnowledgeBaseConfiguration(Properties, ClassLoader) : KnowledgeBaseConfiguration
+newKnowledgeSessionConfiguration() : KnowledgeSessionConfiguration
+newKnowledgeSessionConfiguration(Properties) : KnowledgeSessionConfiguration
org.drools
KnowledgeBaseProvider
java.lang
ClassLoader
org.drools.runtime
Environment
KnowledgeSessionConfiguration
java.util
Properties
org.drools
KnowledgeBase
KnowledgeBaseConfiguration

Figure 3.6. KnowledgeBaseFactory

A **KnowledgeBase** can be created using the default configuration.

```
KnowledgeBase kbase = KnowledgeBaseFactory.newKnowledgeBase();
```

Example 3.11. Creating a new KnowledgeBase

If a custom **ClassLoader** was used with the **KnowledgeBuilder** to resolve types not in the default **ClassLoader**, then that must also be set on the **KnowledgeBuilder**. The technique for this is the same as with the **KnowledgeBuilder**.

```
KnowledgeBaseConfiguration kbaseConf =
 KnowledgeBaseFactory.createKnowledgeBaseConfiguration( null, cl );
KnowledgeBase kbase = KnowledgeBaseFactory.newKnowledgeBase( kbaseConf );
```

Example 3.12. Creating a new KnowledgeBase with a custom ClassLoader

3.2.3. In-Process Building and Deployment

This is the simplest form of deployment. It compiles the knowledge definitions and adds them to the **KnowledgeBase** in the same JVM. This approach requires **drools-core.jar** and **drools-compiler.jar** to be on the classpath.

```
Collection<KnowledgePackage> kpkgs = kbuilder.getKnowledgePackages();
KnowledgeBase kbase = KnowledgeBaseFactory.newKnowledgeBase();
kbase.addKnowledgePackages( kpkgs );
```

Example 3.13. Add KnowledgePackages to a KnowledgeBase

Note that the `addKnowledgePackages(kpkgs)` method can be called iteratively to add additional knowledge.

3.2.4. Building and Deployment in Separate Processes

Both the **`KnowledgeBase`** and the **`KnowledgePackage`** are units of deployment and able to be serialized. This means you can have one machine do any necessary building, requiring **`drools-compiler.jar`**, and have another machine deploy and execute everything, needing only **`drools-core.jar`**.

Although serialization is standard Java, below is an example of how one machine might write out the deployment unit and how another machine might read in and use that deployment unit.

```
ObjectOutputStream out =
 new ObjectOutputStream( new FileOutputStream( fileName ) );
out.writeObject( kpkgs );
out.close();
```

Example 3.14. Writing the KnowledgePackage to an OutputStream

```
ObjectInputStream in =
 new ObjectInputStream( new FileInputStream( fileName ) );
// The input stream might contain an individual
// package or a collection.
Collection&lt;KnowledgePackages&gt; kpkgs = in.readObject( );
in.close();

KnowledgeBase kbase = KnowledgeBaseFactory.newKnowledgeBase();
kbase.addKnowledgePackages( kpkgs );
```

Example 3.15. Reading the KnowledgePackage from an InputStream

The **`KnowledgeBase`** is also serializable and some people may prefer to build and then store the **`KnowledgeBase`** itself, instead of the **`KnowledgePackage`**s.

The JBoss Enterprise BRMS Platform uses this deployment approach. After the BRMS Platform has compiled and published serialized **`KnowledgePackage`**s on a URL, JBoss Rules can use the URL resource type to load them.

3.2.5. StatefulknowledgeSessions and KnowledgeBase Modifications

`StatefulKnowledgeSession`s are discussed in more detail in *Section 3.3.2, “StatefulKnowledgeSession”*. The **`KnowledgeBase`** creates and returns

StatefulKnowledgeSessions and it may optionally keep references to those. When **KnowledgeBase** modifications occur those modifications are applied against the data in the sessions. This reference is a weak reference and it is also optional, which is controlled by a Boolean flag.

3.2.6. KnowledgeAgent

The **KnowlegeAgent** provides automatic loading, caching and re-loading of resources and is configured from a properties files. The **KnowledgeAgent** can update or rebuild this **KnowlegeBase** as the resources it uses are changed. The strategy for this is determined by the configuration given to the factory, but it is typically pull-based using regular polling. Push-based updates and rebuilds will be added in a future version. The **KnowledgeAgent** will continuously scan all the added resources, using a default polling interval of 60 seconds. If their date of the last modification is updated it will rebuild the cached **KnowledgeBase** using the new resources.

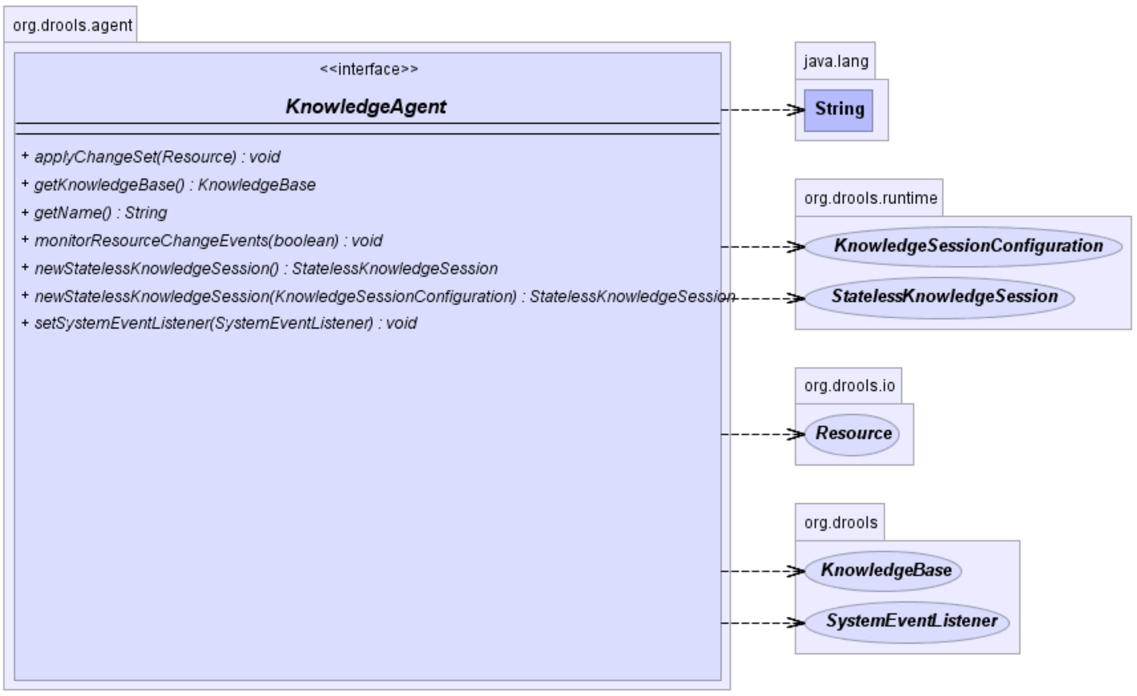

org.drools.agent
<<interface>>
KnowledgeAgent
+ applyChangeSet(Resource) : void
+ getKnowledgeBase() : KnowledgeBase
+ getName() : String
+ monitorResourceChangeEvents(boolean) : void
+ newStatelessKnowledgeSession() : StatelessKnowledgeSession
+ newStatelessKnowledgeSession(KnowledgeSessionConfiguration) : StatelessKnowledgeSession
+ setSystemEventListener(SystemEventListener) : void
java.lang
String
org.drools.runtime
KnowledgeSessionConfiguration
StatelessKnowledgeSession
org.drools.io
Resource
org.drools
KnowledgeBase
SystemEventListener

Figure 3.7. KnowledgeAgent

The **KnowlegeBuilder** is created using the **KnowledgeBuilderFactory**. The agent must specify a name, which is used in the log files to associate a log entry with the corresponding agent.

```
KnowledgeAgent kagent =
         KnowledgeAgentFactory.newKnowledgeAgent( "MyAgent" );
```

Example 3.16. Creating the KnowledgeAgent

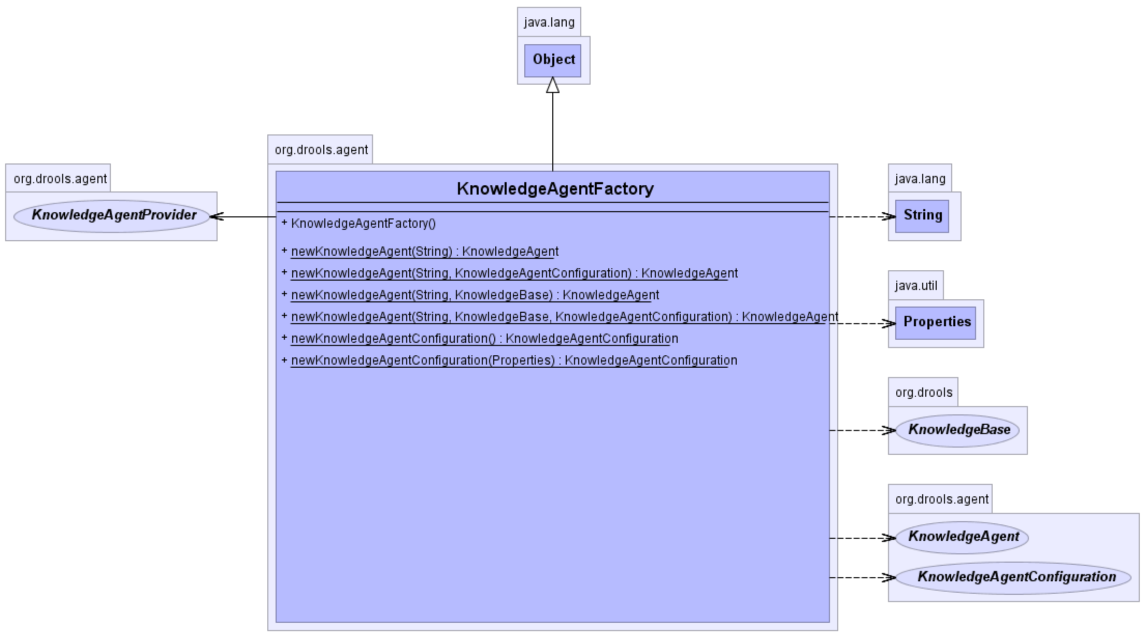

java.lang
Object
org.drools.agent
KnowledgeAgentFactory
+ KnowledgeAgentFactory()
+ newKnowledgeAgent(String) : KnowledgeAgent
+ newKnowledgeAgent(String, KnowledgeAgentConfiguration) : KnowledgeAgent
+ newKnowledgeAgent(String, KnowledgeBase) : KnowledgeAgent
+ newKnowledgeAgent(String, KnowledgeBase, KnowledgeAgentConfiguration) : KnowledgeAgent
+ newKnowledgeAgentConfiguration() : KnowledgeAgentConfiguration
+ newKnowledgeAgentConfiguration(Properties) : KnowledgeAgentConfiguration
org.drools.agent
KnowledgeAgentProvider
java.lang
String
java.util
Properties
org.drools
KnowledgeBase
org.drools.agent
KnowledgeAgent
KnowledgeAgentConfiguration

Figure 3.8. KnowledgeAgentFactory

The following example constructs an agent that will build a new **`KnowledgeBase`** from the specified ChangeSet. Refer to *Section 3.1.2, “Building using Configuration and the ChangeSet XML”* for additional details on ChangeSets. Note that the method can be called iteratively to add new resources over time. The **`KnowledgeAgent`** polls the resources added from the ChangeSet every 60 seconds, the default interval, to see if they are updated. Whenever changes are found it will construct a new **`KnowledgeBase`**. If the change set specifies a resource that is a directory its contents will be scanned for changes, too.

```
KnowledgeAgent kagent =
         KnowledgeAgentFactory.newKnowledgeAgent( "MyAgent" );
kagent.applyChangeSet( ResourceFactory.newUrlResource( url ) );
KnowledgeBase kbase = kagent.getKnowledgeBase();
```

Example 3.17. Writing the KnowledgePackage to an OutputStream

Resource scanning is not on by default. It is a service and must be specifically started. This is also true for for notification. Both can be done via the **`ResourceFactory`**.

```
ResourceFactory.getResourceChangeNotifierService().start();
ResourceFactory.getResourceChangeScannerService().start();
```

Example 3.18. Starting the Scanning and Notification Services

The default resource scanning period may be changed via the **`ResourceChangeScannerService`**. A suitably updated **`ResourceChangeScannerConfiguration`** object is passed to the service's `configure()` method, which allows for the service to be reconfigured on demand.

```
ResourceChangeScannerConfiguration sconf =
    ResourceFactory.getResourceChangeScannerService().
        newResourceChangeScannerConfiguration();
// Set the disk scanning interval to 30s, default is 60s.
sconf.setProperty( "drools.resource.scanner.interval", "30" );
ResourceFactory.getResourceChangeScannerService().configure( sconf );
```

Example 3.19. Changing the Scanning Intervals

KnowledgeAgents can take an empty **KnowledgeBase** or a populated one. If a populated **KnowledgeBase** is provided, the **KnowledgeAgent** will run an iterator from **KnowledgeBase** and subscribe to the resources that it finds. While it is possible for the **KnowledgeBuilder** to build all resources found in a directory, that information is lost by the **KnowledgeBuilder** so that those directories will not be continuously scanned. Only directories specified as part of the `applyChangeSet(Resource)` method are monitored.

One of the advantages of providing **KnowledgeBase** as the starting point is that you can provide it with a **KnowledgeBaseConfiguration**. When resource changes are detected and a new **KnowledgeBase** is instantiated, it will use the **KnowledgeBaseConfiguration** of the previous **KnowledgeBase**.

```
KnowledgeBaseConfiguration kbaseConf =
    KnowledgeBaseFactory.createKnowledgeBaseConfiguration( null, cl );
KnowledgeBase kbase KnowledgeBaseFactory.newKnowledgeBase( kbaseConf );
// Populate kbase with resources here.

KnowledgeAgent kagent =
    KnowledgeAgentFactory.newKnowledgeAgent( "MyAgent", kbase );
KnowledgeBase kbase = kagent.getKnowledgeBase();
```

Example 3.20. Using an existing KnowledgeBase

In the above example `getKnowledgeBase()` will return the same provided kbase instance until resource changes are detected and a new **KnowledgeBase** is built. When the new **KnowledgeBase** is built, it will be done with the **KnowledgeBaseConfiguration** that was provided to the previous **KnowledgeBase**.

As mentioned previously, a ChangeSet XML can specify a directory and all of its contents will be added. If this ChangeSet XML is used with the `applyChangeSet()` method it will also add any directories to the scanning process. When the directory scan detects an additional file it will be added to the **KnowledgeBase**. Any removed file is removed from the **KnowledgeBase**, and modified files will, as usual, force the build of a new **KnowledgeBase** using the latest version.

```
<change-set xmlns='http://drools.org/drools-5.0/change-set'
    xmlns:xs='http://www.w3.org/2001/XMLSchema-instance'
    xs:schemaLocation='http://drools.org/drools-5.0/change-set.xsd' >
   <add>
      <resource source='file:/projects/myproject/myrules' type='PKG' />
   </add>
</change-set>
```

Example 3.21. ChangeSet XML which adds a directories contents

Note that for the resource type PKG the drools-compiler dependency is not needed as the **`KnowledgeAgent`** is able to handle those with just drools-core.

The **`KnowledgeAgentConfiguration`** can be used to modify a **`KnowledgeAgent`**'s default behavior. You could use this to load the resources from a directory, while inhibiting the continuous scan for changes of that directory.

```
KnowledgeBase kbase = KnowledgeBaseFactory.newKnowledgeBase();

KnowledgeAgentConfiguration kaconf =
    KnowledgeAgentFactory.newKnowledgeAgentConfiguation();
// Do not scan directories, just files.
kaconf.setProperty( "drools.agent.scanDirectories", "false" );
KnowledgeAgent kagent =
    KnowledgeAgentFactory.newKnowledgeAgent( "test agent", kaconf );
```

Example 3.22. Change the Scanning Behavior

Previously we mentioned the JBoss Enterprise BRMS Platform and how it can build and publish serialized **`KnowledgePackage`**s on a URL, and that the ChangeSet XML can handle URLs and Packages. Taken together, this forms an important deployment scenario for the **`KnowledgeAgent`**.

3.3. Running

3.3.1. KnowledgeBase

The **`KnowlegeBase`** is a repository of all the application's knowledge definitions. It may contain rules, processes, functions, and type models. The **`KnowledgeBase`** itself does not contain instance data, known as facts. Instead sessions are created from the **`KnowledgeBase`** into which data (facts) can be inserted and where process instances may be started. **`KnowlegeBase`** creation is a fairly intensive process, whereas session creation is not. It is recommended that **`KnowledgeBase`**s be cached where possible to allow for repeated session creation.

```
KnowledgeBase kbase = KnowledgeBaseFactory.newKnowledgeBase();
```

Example 3.23. Creating a new KnowledgeBuilder

3.3.2. StatefulKnowledgeSession

The **`StatefulKnowledgeSession`** stores and executes on the runtime data and is created from the **`KnowledgeBase`**.

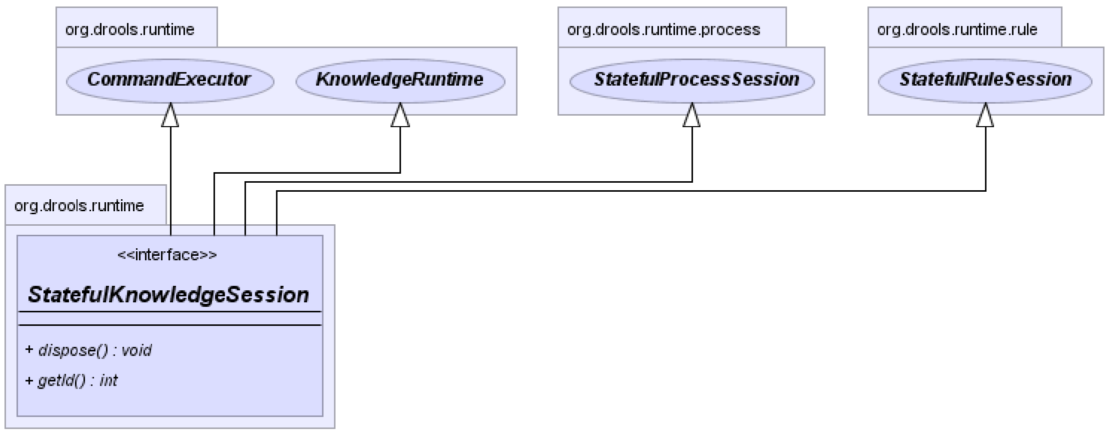


Figure 3.9. StatefulKnowledgeSession

```
StatefulKnowledgeSession ksession = kbase.newStatefulKnowledgeSession();
```

Example 3.24. Add KnowledgePackages to a KnowledgeBase

3.3.3. KnowledgeRuntime

3.3.3.1. WorkingMemoryEntryPoint

The **`WorkingMemoryEntry`** provides the methods around inserting, updating and retrieving facts. The term **`EntryPoint`** is related to the fact that we have multiple partitions in a **`WorkingMemory`** and you can choose which one you are inserting into. However this use case is aimed at event processing and most rule based applications will only make use of the default entry point.

The **`KnowledgeRuntime`** interface provides the main interaction with the engine and is available in rule consequences and process actions. In this manual the focus is on the methods and interfaces related to rules. But you'll notice that the **`KnowledgeRuntime`** inherits methods from both the **`WorkingMemory`** and the **`ProcessRuntime`**, this provides a unified API to work with process and rules. When working with rules three interfaces form the **`KnowledgeRuntime`**: **`WorkingMemoryEntryPoint`**, **`WorkingMemory`**, and the **`KnowledgeRuntime`** itself.

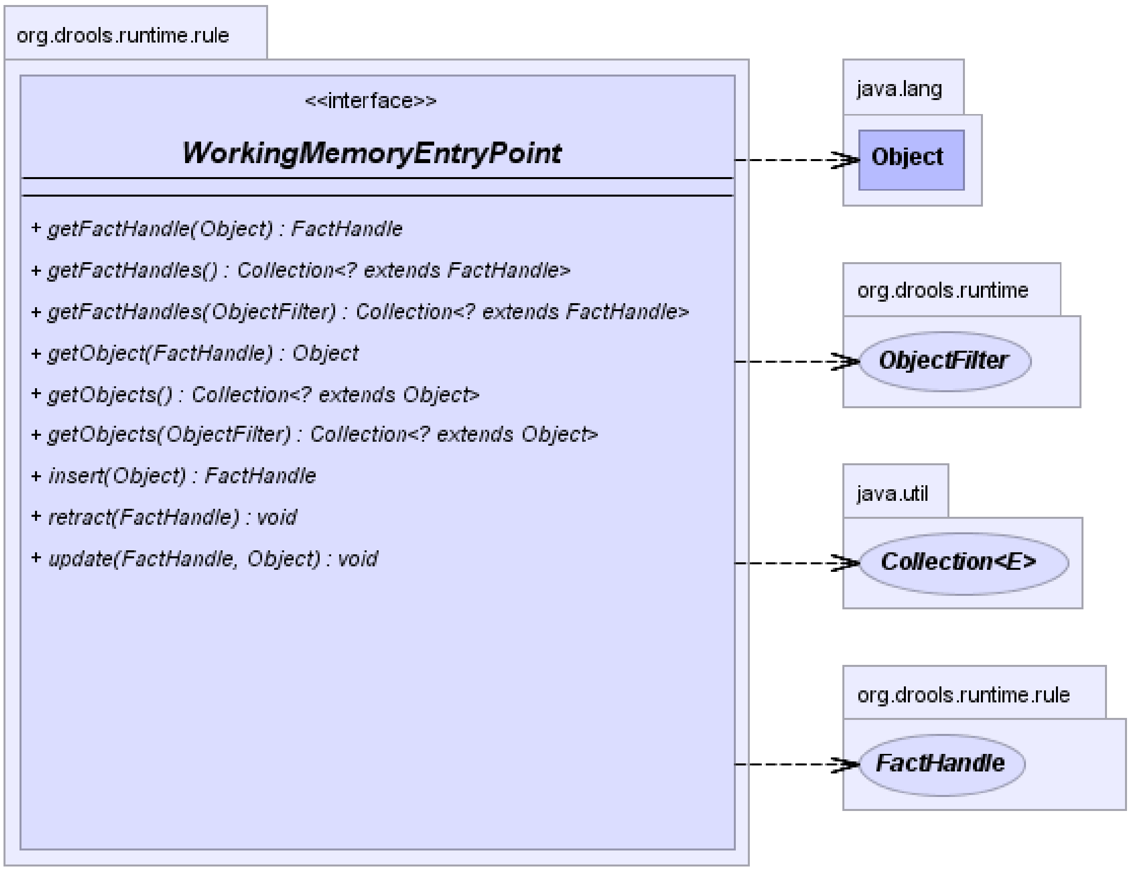

org.drools.runtime.rule
<<interface>>
WorkingMemoryEntryPoint
+ getFactHandle(Object) : FactHandle
+ getFactHandles() : Collection<? extends FactHandle>
+ getFactHandles(ObjectFilter) : Collection<? extends FactHandle>
+ getObject(FactHandle) : Object
+ getObjects() : Collection<? extends Object>
+ getObjects(ObjectFilter) : Collection<? extends Object>
+ insert(Object) : FactHandle
+ retract(FactHandle) : void
+ update(FactHandle, Object) : void
java.lang
Object
org.drools.runtime
ObjectFilter
java.util
Collection<E>
org.drools.runtime.rule
FactHandle

Figure 3.10. WorkingMemoryEntryPoint

3.3.3.1.1. Insertion

Insertion is the act of telling the **`WorkingMemory`** about a fact, e.g. `ksession.insert(yourObject)`. When you insert a fact, it is examined for matches against the rules. This means *all* of the work for deciding about firing or not firing a rule is done during insertion. However no rule is executed until you call the `fireAllRules()` method which you call after you have inserted all your facts.

Expert systems typically use the term *assert* or *assertion* to refer to facts made available to the system. However due to the `assert` being a keyword in most languages we have moved to use the `insert` keyword. It is common to hear the two terms used interchangably; so expect to hear the two used interchangeably.

When an object is inserted it returns a **`FactHandle`**. This **`FactHandle`** is the token used to represent your inserted object within the **`WorkingMemory`**. It is also used for interactions with the **`WorkingMemory`** when you wish to retract or modify an object.

```
Cheese stilton = new Cheese("stilton");
FactHandle stiltonHandle = ksession.insert( stilton );
```

As mentioned in the **`KnowledgeBase`** section a **`WorkingMemory`** may operate in one of two assertion modes: equality or identity. Identity is the default.

Identity means that the Working Memory uses an **`IdentityHashMap`** to store all asserted objects. New instance assertions always result in the return of a new **`FactHandle`** and repeated insertions of the same instance will simply return the original fact handle.

Equality means that the Working Memory uses a **`HashMap`** to store all asserted objects. New instance assertions will only return a new **`FactHandle`** if no equal objects have been asserted.

3.3.3.1.2. Retraction

Retraction is the removal of a fact from the Working Memory. The fact will no longer be tracked or matched to rules, and any rules that are activated and dependent on that fact will be cancelled. Note that it is possible to have rules that depend on the nonexistence of a fact, in which case retracting a fact may cause a rule to activate (see the `not` and `exist` keywords). Retraction is done using the **`FactHandle`** that was returned during the assert.

```
Cheese stilton = new Cheese("stilton");
FactHandle stiltonHandle = ksession.insert( stilton );

ksession.retract( stiltonHandle );
```

3.3.3.1.3. Update

The Rule Engine must be notified of modified Facts, so that they can be reprocessed. A fact which is identified as updated is actually automatically retracted from the **`WorkingMemory`** and inserted again.

If an modified object is not able to notify the **`WorkingMemory`** itself you must use the `update` method to notify the **`WorkingMemory`**. The `update` method always takes the modified object as a second parameter, which allows you to specify new instances for immutable objects. The `update` method can only be used with objects that have shadow proxies turned on.

The `update` method is only for use from Java code.Within a rule the `modify` keyword is supported and provides block setters.

```
Cheese stilton = new Cheese("stilton");
FactHandle stiltonHandle = workingMemory.insert( stilton );
....
stilton.setPrice( 100 );
workingMemory.update( stiltonHandle, stilton );
```

3.3.3.2. WorkingMemory

The **`WorkingMemory`** provides access to the Agenda, query executions as well getting access to named **`EntryPoints`**.

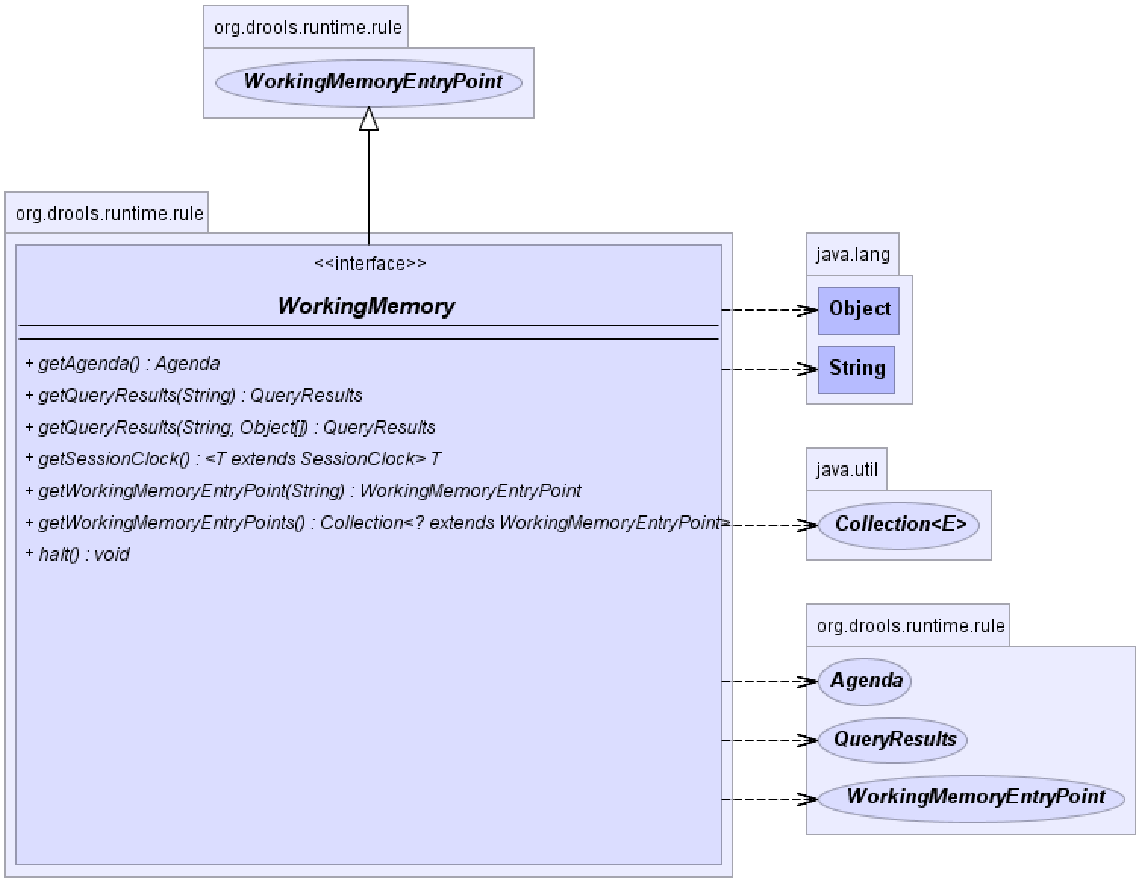

org.drools.runtime.rule
WorkingMemoryEntryPoint
org.drools.runtime.rule
<<interface>>
WorkingMemory
+ getAgenda() : Agenda
+ getQueryResults(String) : QueryResults
+ getQueryResults(String, Object[]) : QueryResults
+ getSessionClock() : <T extends SessionClock> T
+ getWorkingMemoryEntryPoint(String) : WorkingMemoryEntryPoint
+ getWorkingMemoryEntryPoints() : Collection<? extends WorkingMemoryEntryPoint>
+ halt() : void
java.lang
Object
String
java.util
Collection<E>
org.drools.runtime.rule
Agenda
QueryResults
WorkingMemoryEntryPoint

Figure 3.11. WorkingMemory

3.3.3.2.1. Query

Queries can be defined in the **`KnowlegeBase`** which can be called to return the matching results. Any bound identifier in the query can be accessed using the `get(String identifier)` method.

```
QueryResults results =
    ksession.getQueryResults( "my query", new Object[] { "string" } );
for ( QueryResultsRow row : results ) {
    System.out.println( row.get( "varName" ) );
}
```

Example 3.25. Simple Query Example

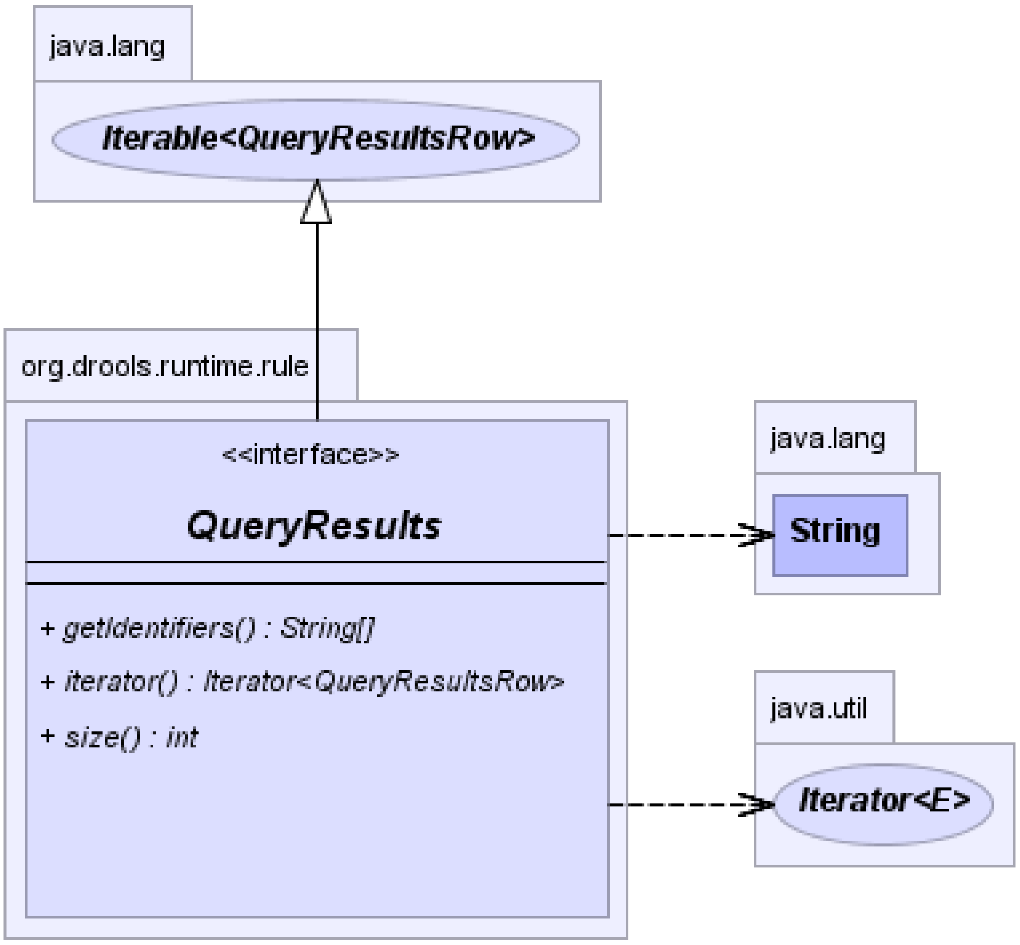


Figure 3.12. QueryResults

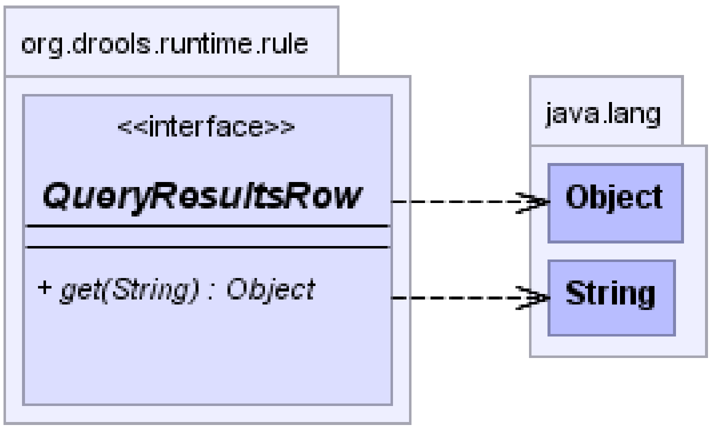


Figure 3.13. QueryResultsRow

3.3.3.3. KnowledgeRuntime

The **`KnowledgeRuntime`** provides further methods that are applicableto both rules and processes. Such as setting globals and registering **`ExitPoints`**.

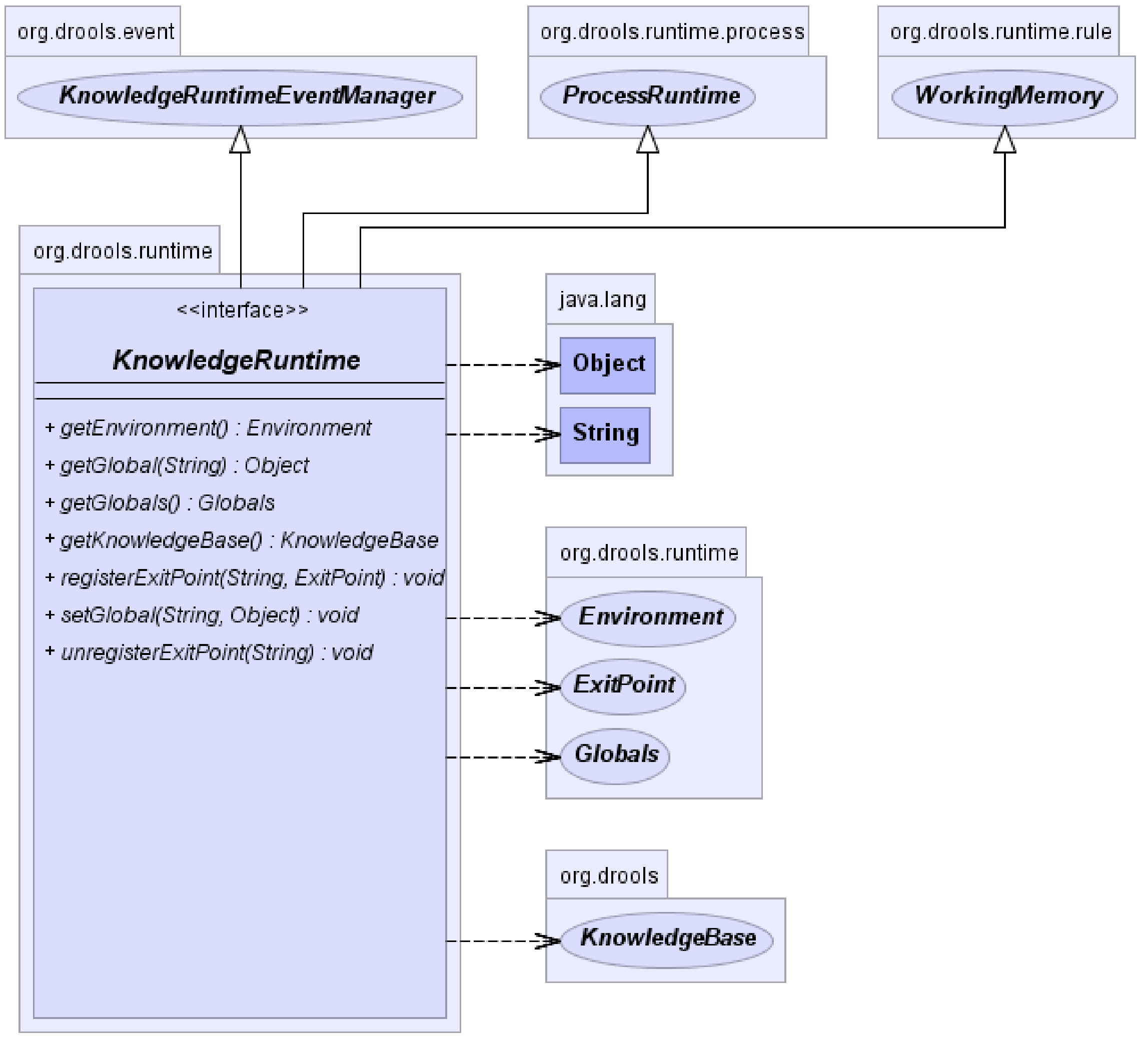

org.drools.event
KnowledgeRuntimeEventManager
org.drools.runtime.process
ProcessRuntime
org.drools.runtime.rule
WorkingMemory
org.drools.runtime
<<interface>>
KnowledgeRuntime
+ getEnvironment() : Environment
+ getGlobal(String) : Object
+ getGlobals() : Globals
+ getKnowledgeBase() : KnowledgeBase
+ registerExitPoint(String, ExitPoint) : void
+ setGlobal(String, Object) : void
+ unregisterExitPoint(String) : void
java.lang
Object
String
org.drools.runtime
Environment
ExitPoint
Globals
org.drools
KnowledgeBase

Figure 3.14. KnowledgeRuntime

3.3.3.3.1. Globals

Globals are named objects that can be passed in to the rule engine. They are not considered to be inserted nor are they matched against rules.

Globals are often used for static information, or for services that are used in the *right-hand side* (RHS) of a rule. They can also be used as a means to return objects from the rule engine. If you use a global on the *left-hand side* (LHS) of a rule, first ensure it is immutable. A global must first be declared in a rules file before it can be set on the session.

```
global java.util.List list
```

With the **`KnowledgeBase`** now aware of the global identifier and its type, it is now possible to call `ksession.setGlobal` for any session. Failure to declare the global type and identifier first will result in an exception being thrown. To set the global on the session use `ksession.setGlobal(identifier, value)`.

```
List list = new ArrayList();
```

```
ksession.setGlobal("list", list);
```

If a rule evaluates on a global before you set it you will get a `NullPointerException`.

3.3.3.4. StatefulRuleSession

The **`StatefulRuleSession`** is inherited by the **`StatefulKnowledgeSession`** and provides the rule related methods that are relevant from outside of the engine.

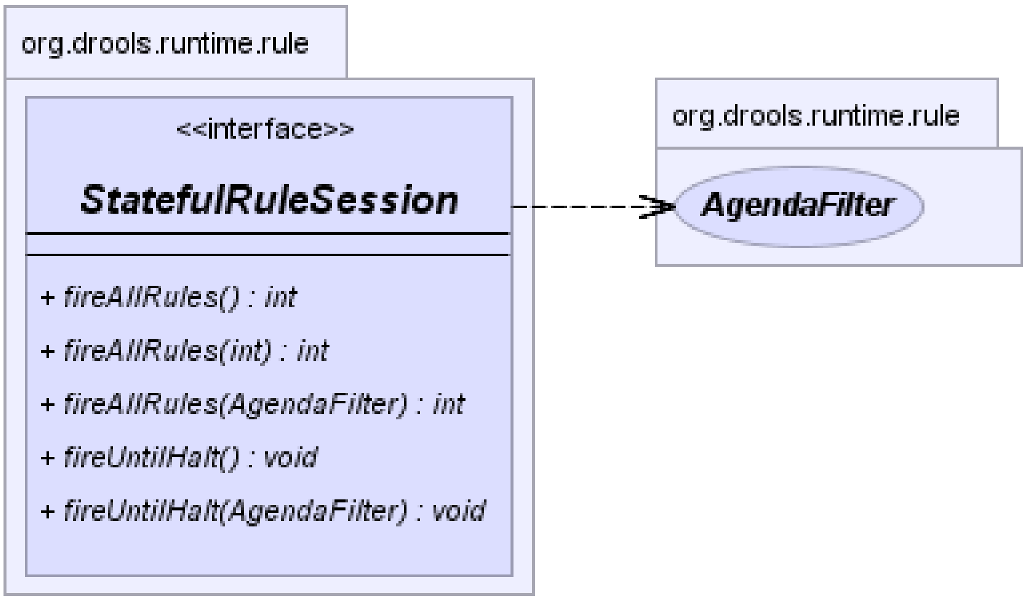


Figure 3.15. StatefulRuleSession

3.3.3.4.1. Agenda Filters

Agenda filters are optional implementations of the `filter` interface which are used to allow or deny the firing of an activation. What you filter on is entirely up to the implementation.

Earlier versions of JBoss Rules supplied several filters which are not provided in the version 5.0 drools-api. They are simple to implement and the JBoss Rules 4 code base can be referred to.

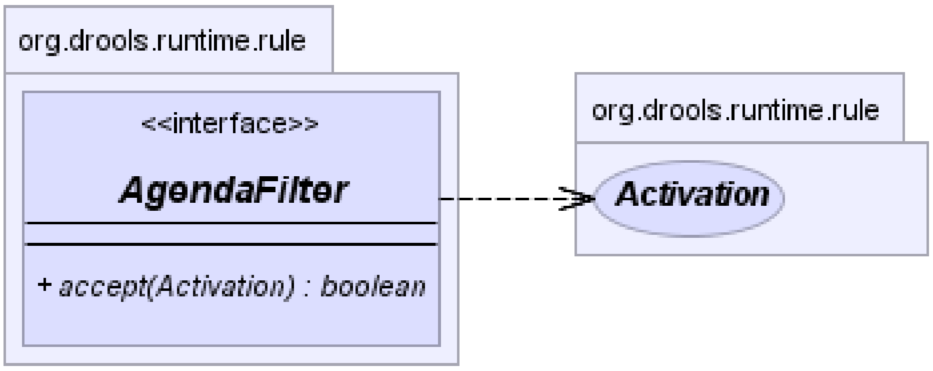


Figure 3.16. AgendaFilters

To use a filter specify it when calling `FireAllRules()`. The following example permits only rules ending in the string *Test*. All others will be filtered out.

```
ksession.fireAllRules( new RuleNameEndsWithAgendaFilter( "Test" ) );
```

3.3.4. Agenda

The *Agenda* is a RETE feature. During actions on the **`WorkingMemory`**, rules may become fully matched and eligible for execution. A single Working Memory Action can result in multiple eligible rules. When a rule is fully matched an Activation is created, referencing the rule and the matched facts, and placed onto the Agenda. The Agenda controls the execution order of these Activations using a Conflict Resolution strategy.

The engine cycles repeatedly through two phases:

1. Working Memory Actions. This is where most of the work takes place, either in the `Consequence` (the RHS itself) or the main Java application process. Once the `Consequence` has finished or the main Java application process calls `fireAllRules()` the engine switches to the Agenda Evaluation phase.

2. Agenda Evaluation. This attempts to select a rule to fire. If no rule is found it exits, otherwise it fires the found rule, switching the phase back to Working Memory Actions.

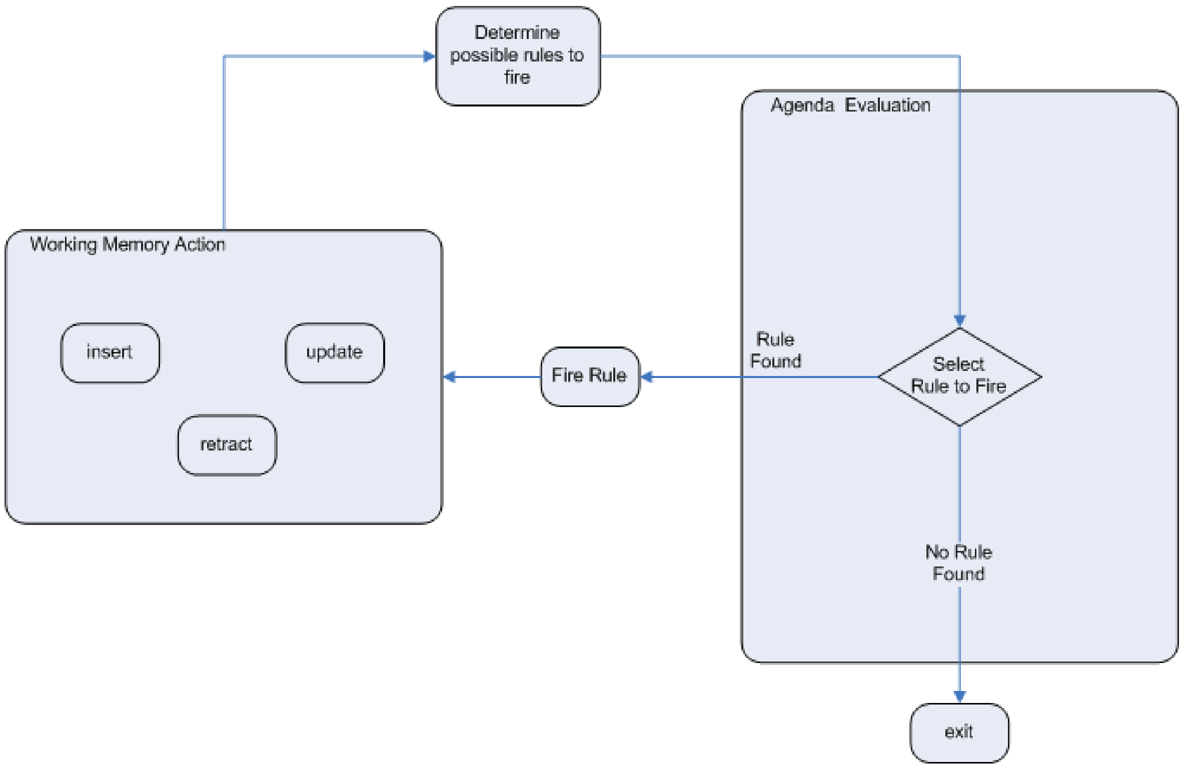


Figure 3.17. Two Phase Execution

The process repeats until the agenda is clear, in which case control returns to the calling application. When Working Memory Actions are taking place, no rules are being fired.

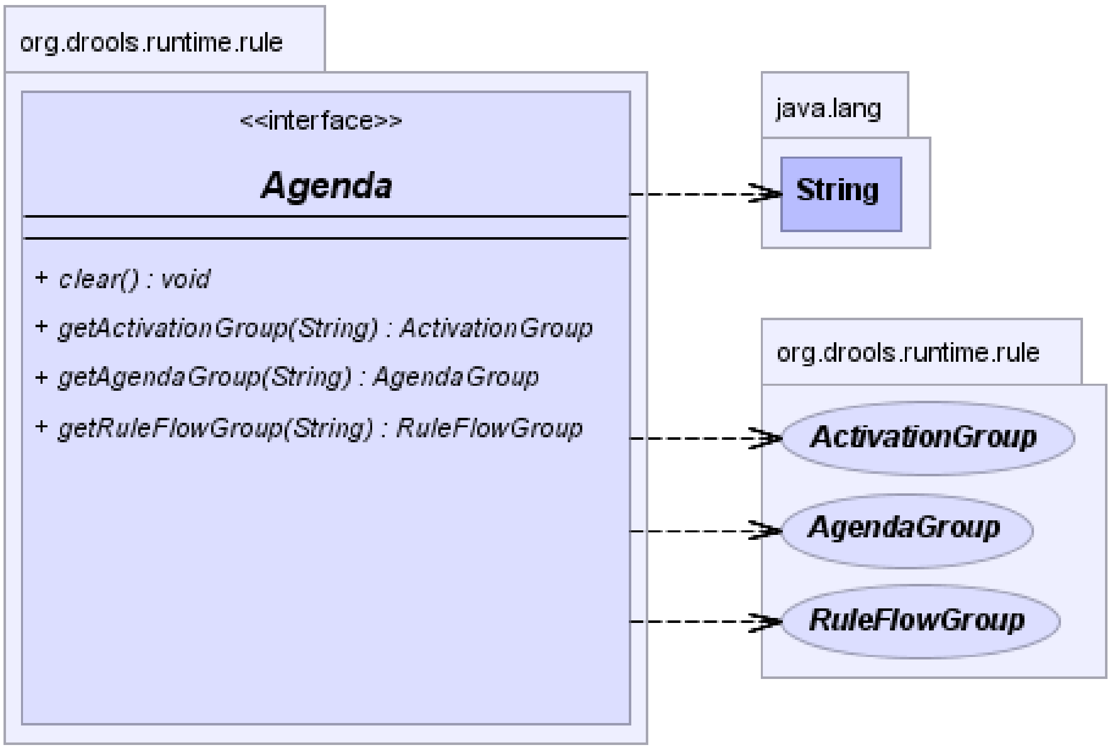


Figure 3.18. Agenda

3.3.4.1. Conflict Resolution

Conflict resolution is required when there are multiple rules on the agenda, the basics to this are covered in *Chapter 2, Quick Start* . As firing a rule may have side effects on working memory, the rule engine needs to know in what order the rules should fire. For example, firing **`ruleA`** may cause **`ruleB`** to be removed from the agenda.

The default conflict resolution strategies employed by JBoss Rules are: Salience and LIFO (last in, first out).

The most visible one is "salience" or priority, in which a user can specify that a certain rule has a higher priority (by giving it a higher number) than other rules. In that case, the rule with higher salience will be preferred.

LIFO priorities are based on the assigned *Working Memory Action counter value*, with all rules created during the same action receiving the same value. The execution order of a set of firings with the same priority value is arbitrary.

As a general rule, it is a good idea not to count on the rules firing in any particular order. Remember that you should not be authoring rules as though they are steps in a imperative process.

Note

Previous versions of JBoss Rules supported custom conflict resolution strategies. This capability still exists in version 5 but the API is not currently exposed.

3.3.4.2. AgendaGroup

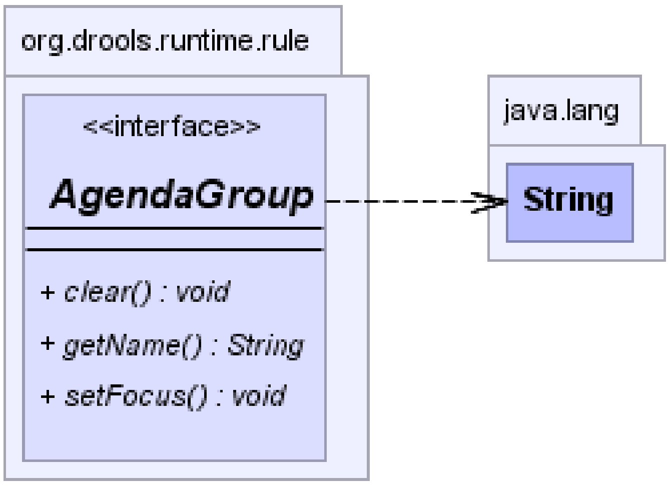


Figure 3.19. AgendaGroup

Agenda Groups are a way to partition Activations on the Agenda. Agenda Groups are known as "modules" in CLIPS terminology. At any time only one group can have "focus", and only the Activations belonging to that group will take effect.

Focus can be set from within a rule or by using the JBoss Rules API. Rules can also be set with "auto focus", so its Agenda Group will become focused when it becomes matched.

Agenda Groups are most commonly used to define phases of processing.

```
ksession.getAgenda().getAgendaGroup( "Group A" ).setFocus();
```

Each time `setFocus()` is called it pushes that Agenda Group onto a stack. When the focus group is empty it is popped from the stack and the focus group that is now on top evaluates. An Agenda Group can appear in multiple locations on the stack.

The default Agenda Group is "MAIN". It it the first group on the stack and has the initial focus. Any rule without a Agenda Group is automatically placed in this group.

3.3.4.3. ActivationGroup

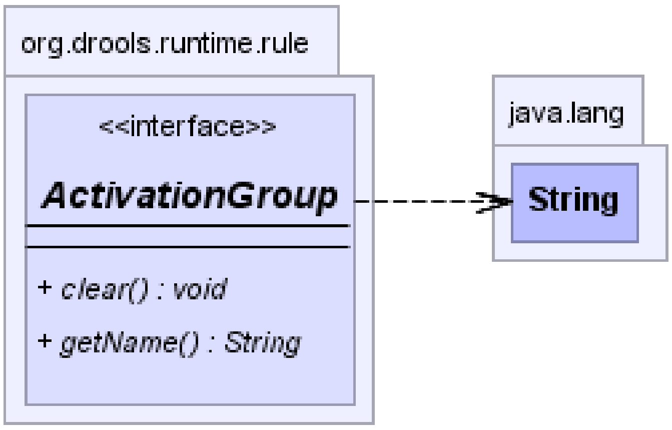


Figure 3.20. ActivationGroup

An activation group is group of rules associated together by the activation-group rule attribute. In this group only one rule can fire. After that rule has fired all the other rules are cancelled. The `clear()` method can be called at any time, which cancels all of the activations before one has a chance to fire.

```
ksession.getAgenda().getActivationGroup( "Group B" ).clear();
```

3.3.5. Event Model

The event package provides means to be notified of rule engine events, including rules firing, objects being asserted, etc. This allows you, for instance, to separate logging and auditing activities from the main part of your application and the rules.

The **`KnowlegeRuntimeEventManager`** is implemented by the **`KnowledgeRuntime`** which provides two interfaces, **`WorkingMemoryEventManager`** and **`ProcessEventManager`**. We will only cover the **`WorkingMemoryEventManager`** here.

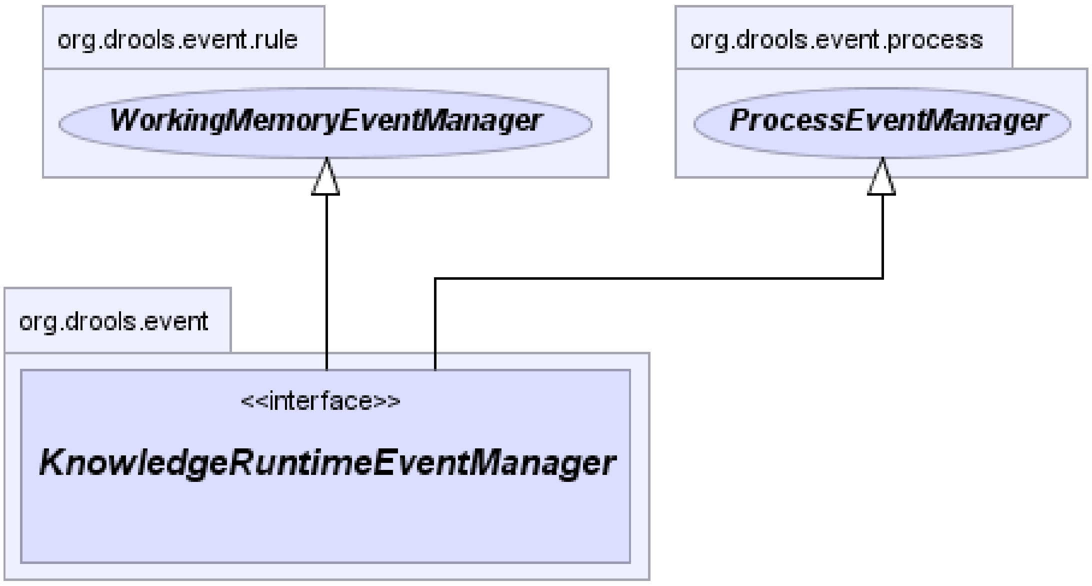


Figure 3.21. KnowledgeRuntimeEventManager

The **WorkingMemoryEventManager** allows for listeners to be added and removed, so that events for the working memory and the agenda can be listened to.

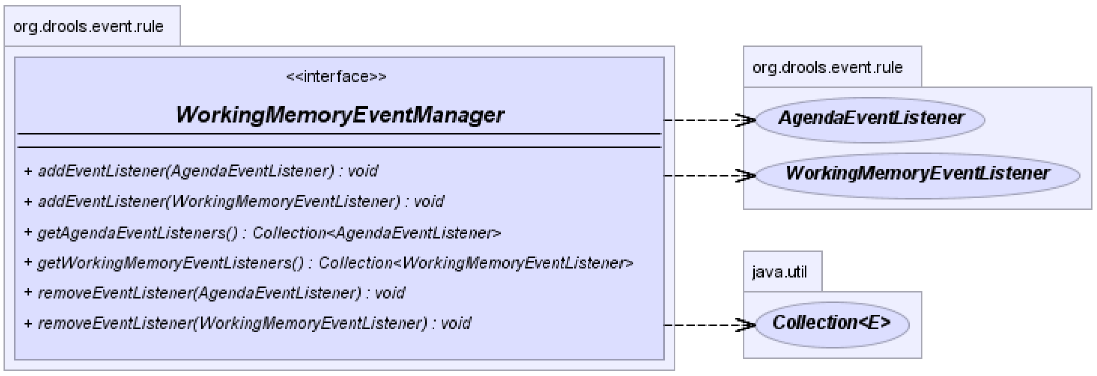

org.drools.event.rule
<<interface>>
WorkingMemoryEventManager
+ addEventListener(AgendaEventListener) : void
+ addEventListener(WorkingMemoryEventListener) : void
+ getAgendaEventListeners() : Collection<AgendaEventListener>
+ getWorkingMemoryEventListeners() : Collection<WorkingMemoryEventListener>
+ removeEventListener(AgendaEventListener) : void
+ removeEventListener(WorkingMemoryEventListener) : void
org.drools.event.rule
AgendaEventListener
WorkingMemoryEventListener
java.util
Collection<E>

Figure 3.22. WorkingMemoryEventManager

```
ksession.addEventListener( new DefaultAgendaEventListener() {
    public void afterActivationFired(AfterActivationFiredEvent event) {
        super.afterActivationFired( event );
        System.out.println( event );
    }
});
```

Example 3.26. Adding an AgendaEventListener

JBoss Rules also provides **DebugWorkingMemoryEventListener**, **DebugAgendaEventListener** which implement each method with a debug print statement. To print all Working Memory events, you add a listener like this:

```
ksession.addEventListener( new DebugWorkingMemoryEventListener() );
```

Example 3.27. Creating a new KnowledgeBuilder

All emitted events implement the `KnowlegeRuntimeEvent` interface which can be used to retrieve the actual **`KnowlegeRuntime`** the event originated from.

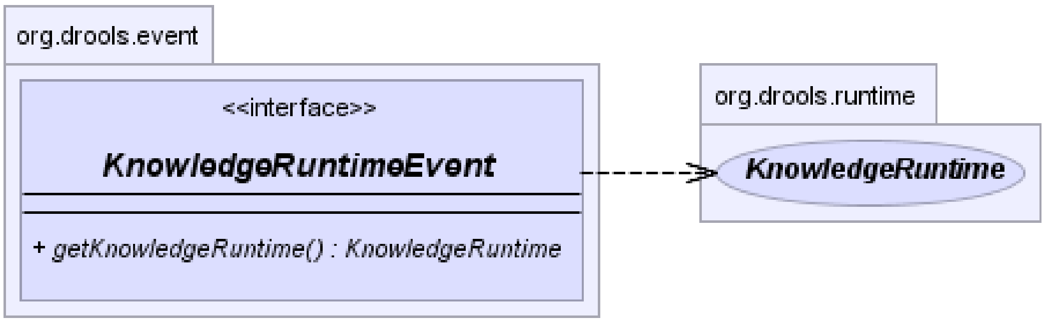


Figure 3.23. KnowlegeRuntimeEvent

The events currently supported are:

ActivationCreatedEvent
ActivationCancelledEvent
BeforeActivationFiredEvent
AfterActivationFiredEvent
AgendaGroupPushedEvent
AgendaGroupPoppedEvent
ObjectInsertEvent
ObjectRetractedEvent
ObjectUpdatedEvent
ProcessCompletedEvent
ProcessNodeLeftEvent
ProcessNodeTriggeredEvent
ProcessStartEvent

3.3.6. KnowledgeRuntimeLogger

The **`KnowledgeRuntimeLogger`** uses the comprehensive event system in JBoss Rules to create an audit log of the execution of rules. This log can be inspected in tools such as the Eclipse audit viewer.

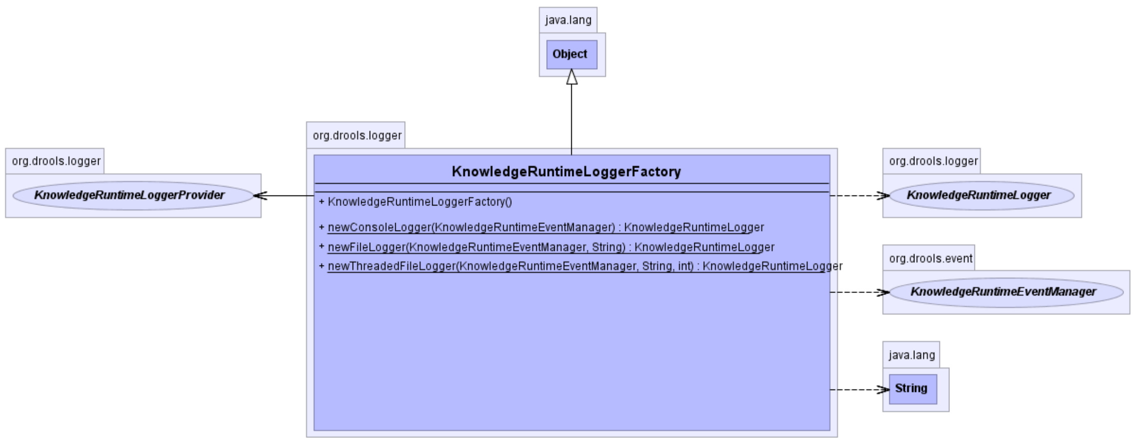

java.lang
Object
org.drools.logger
KnowledgeRuntimeLoggerFactory
+ KnowledgeRuntimeLoggerFactory()
+ newConsoleLogger(KnowledgeRuntimeEventManager) : KnowledgeRuntimeLogger
+ newFileLogger(KnowledgeRuntimeEventManager, String) : KnowledgeRuntimeLogger
+ newThreadedFileLogger(KnowledgeRuntimeEventManager, String, int) : KnowledgeRuntimeLogger
org.drools.logger
KnowledgeRuntimeLoggerProvider
org.drools.logger
KnowledgeRuntimeLogger
org.drools.event
KnowledgeRuntimeEventManager
java.lang
String

Figure 3.24. KnowledgeRuntimeLoggerFactory

```
KnowledgeRuntimeLogger logger =
 KnowledgeRuntimeLoggerFactory.newFileLogger(ksession, "logdir/mylogfile");
....
logger.close();
```

Example 3.28. FileLogger

3.3.7. StatelessKnowledgeSession

The **`StatelessKnowledgeSession`** wraps the **`StatefulKnowledgeSession`**. Its main focus is on decision service type scenarios. It removes the need to call `dispose()`.

Stateless sessions do not support iterative insertions and `fireAllRules()` from java code. The `execute()` internally instantiates a **`StatefullKnowledgeSession`**, adds all the user data and execute user commands, calls `fireAllRules()`, and then calls `dispose()`.

The usual way to work with this class is via the `BatchExecution` Command as supported by the `CommandExecutor` interface. However two convenience methods are provided for when simple object insertion is all that is required. The `CommandExecutor` and `BatchExecution` are talked about in detail in their own section.

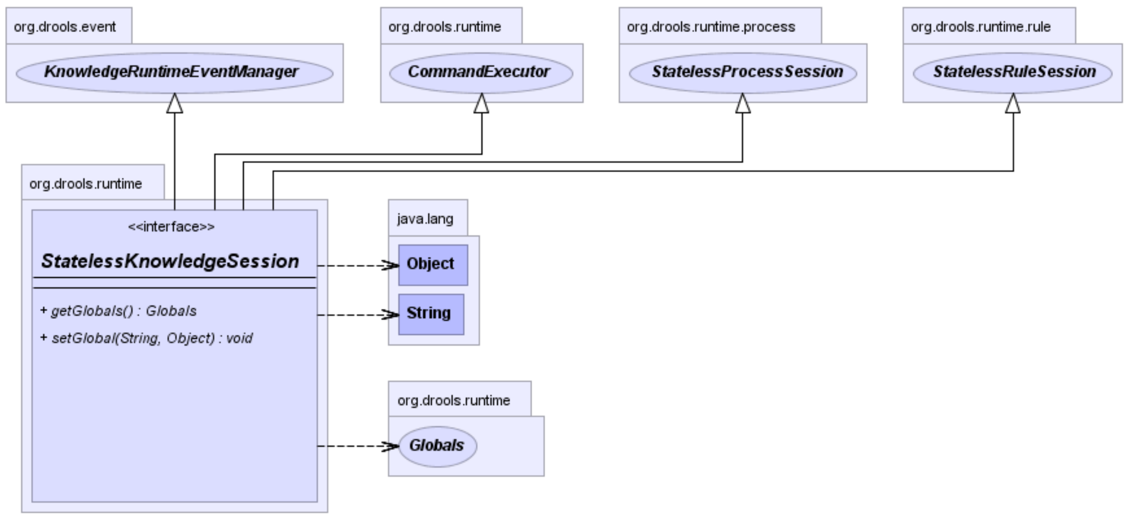


Figure 3.25. StatelessKnowledgeSession

Simple example showing a stateless session executing for a given collection of java objects using the convenience API. It will iterate the collection inserting each element in turn.

```
KnowledgeBuilder kbuilder = KnowledgeBuilderFactory.newKnowledgeBuilder();
kbuilder.add( ResourceFactory.newFileSystemResource( fileName ),
 ResourceType.DRL );
assertFalse( kbuilder.hasErrors() );
if (kbuilder.hasErrors() ) {
    System.out.println( kbuilder.getErrors() );
}
KnowledgeBase kbase = KnowledgeBaseFactory.newKnowledgeBase();
kbase.addKnowledgePackages( kbuilder.getKnowledgePackages() );

StatelessKnowledgeSession ksession = kbase.newStatelessKnowledgeSession();
ksession.execute( collection );
```

Example 3.29. Simple StatelessKnowledgeSession execution with a Collection

If this was done as a single Command it would be as follows:

```
ksession.execute( CommandFactory.newInsertElements( collection ) );
```

Example 3.30. Simple StatelessKnowledgeSession execution with InsertElements Command

Note if you wanted to insert the collection itself, and not the iterate and insert the elements, then you can use `CommandFactory.newInsert(collection)`.

The **`CommandFactory`** details the supported commands, all of which can marshalled using **`XStream`** and the **`BatchExecutionHelper`**. **`BatchExecutionHelper`** provides details on the xml format as well as how to use JBoss Rules Pipeline to automate the marshaling of **`BatchExecution`** and **`ExecutionResults`**.

`StatelessKnowledgeSessions` support globals, scoped in a number of ways. I'll cover the non-command way first, as commands are scoped to a specific execution call. Globals can be resolved in three ways. The **`StatelessKnowledgeSession`** supports `getGlobals()`, which returns a Globals instance. These globals are shared for *all* execution calls, so be especially careful of mutable globals in these cases - as often execution calls can be executing simultaneously in different threads. Globals also supports a delegate, which adds a second way of resolving globals. Calling of `setGlobal(String, Object)` will actually be set on an internal collection, identifiers in this internal collection will have priority over supplied delegate, if one is added. If an identifier cannot be found in the internal collection, it will then check the delegate Globals, if one has been set.

```
StatelessKnowledgeSession ksession = kbase.newStatelessKnowledgeSession();
// sets a global hibernate session, that can be used
// for DB interactions in the rules.
ksession.setGlobal( "hbnSession", hibernateSession );
// this will now execute and will be able to resolve the
// "hbnSession" identifier.
ksession.execute( collection );
```

Example 3.31. Session scoped global

The third way is execution scoped globals using the `CommandExecutor` and `SetGlobal` Commands.

```
StatelessKnowledgeSession ksession = kbase.newStatelessKnowledgeSession();
// sets a global hibernate session, that can be used
// for DB interactions in the rules.
ksession.setGlobal( "hbnSession", hibernateSession );
// this will now execute and will be able to resolve the
// "hbnSession" identifier.
ksession.execute( collection );
```

Example 3.32. Execute scoped global

The `CommandExecutor` interface also supports the ability to expert data via out parameters. Inserted facts, globals and query results can all be returned.

```

List cmds = new ArrayList();
cmds.add( CommandFactory.newSetGlobal( "list1", new ArrayList(), true ) );
cmds.add( CommandFactory.newInsert( new Person( "jon", 102 ), "person" ) );
cmds.add( CommandFactory.newQuery( "Get People" "getPeople" );

ExecutionResults results =
    ksession.execute( CommandFactory.newBatchExecution( cmds ) );
results.getValue( "list1" ); // returns the ArrayList
results.getValue( "person" ); // returns the inserted fact Person
results.getValue( "Get People" );// returns query as QueryResults instance
```

Example 3.33. Out identifiers

3.3.7.1. Sequential Mode

With Rete you have a stateful session where objects can be asserted and modified over time, and where rules can also be added and removed. Now what happens if we assume a stateless session, where after the initial data set no more data can be asserted or modified and rules cannot be added or removed? Certainly it won't be necessary to re-evaluate rules, and the engine will be able to operate in a simplified way.

Sequential mode can only be used with a StatelessSession and is off by default. To turn on either set the RuleBaseConfiguration.setSequential to true or set the rulebase.conf property drools.sequential to true. Sequential mode can fall back to a dynamic agenda with setSequentialAgenda to either SequentialAgenda.SEQUENTIAL or SequentialAgenda.DYNAMIC set by a call or via the "drools.sequential.agenda" property.

When in Sequential mode a session behaves as described below.

1. Order the Rules by salience and position in the ruleset (by setting a sequence attribute on the rule terminal node).

2. Create an array, one element for each possible rule activation; element position indicates firing order.

3. Turn off all node memories, except the right-input Object memory.

4. Disconnect the LeftInputAdapterNode propagation, and let the Object plus the Node be referenced in a Command object, which is added to a list on the WorkingMemory for later execution.

5. Assert all objects, and, when all assertions are finished and thus right-input node memories are populated, check the Command list and execute each in turn.

6. All resulting Activations should be placed in the array, based upon the determined sequence number of the Rule. Record the first and last populated elements, to reduce the iteration range.

7. Iterate the array of Activations, executing populated element in turn.

8. If we have a maximum number of allowed rule executions, we can exit our network evaluations early to fire all the rules in the array.

The LeftInputAdapterNode no longer creates a Tuple, adding the Object, and then propagate the Tuple – instead a Command Object is created and added to a list in the Working Memory. This Command Object holds a reference to the LeftInputAdapterNode and the propagated Object. This stops any left-input propagations at insertion time, so that we know that a right-input propagation will never need to attempt a join with the left-inputs (removing the need for left-input memory). All nodes have their memory turned off, including the left-input Tuple memory but excluding the right-input Object memory, which means that the only node remembering an insertion propagation is the right-input Object memory. Once all the assertions are finished and all right-input memories populated, we can then iterate the list of LeftInputAdatperNode Command objects calling each in turn; they will propagate down the network attempting to join with the right-input objects; not being remembered in the left input, as we know there will be no further object assertions and thus propagations into the right-input memory.

There is no longer an Agenda, with a priority queue to schedule the Tuples, instead there is simply an array for the number of rules. The sequence number of the RuleTerminalNode indicates the element with the array to place the Activation. Once all Command Objects have finished we can iterate our array checking each element in turn and firing the Activations if they exist. To improve performance in the array we remember the first and last populated cells. The network is constructed where each RuleTerminalNode is given a sequence number, based on a salience number and its order of being added to the network.

Typically the right-input node memories are HashMaps, for fast Object retraction; here, as we know there will be no Object retractions, we can use a list when the values of the Object are not indexed. For larger numbers of Objects indexed HashMaps provide a performance increase; if we know an Object type has a low number of instances then indexing is probably not of an advantage and an Object list can be used.

3.3.8. Pipeline

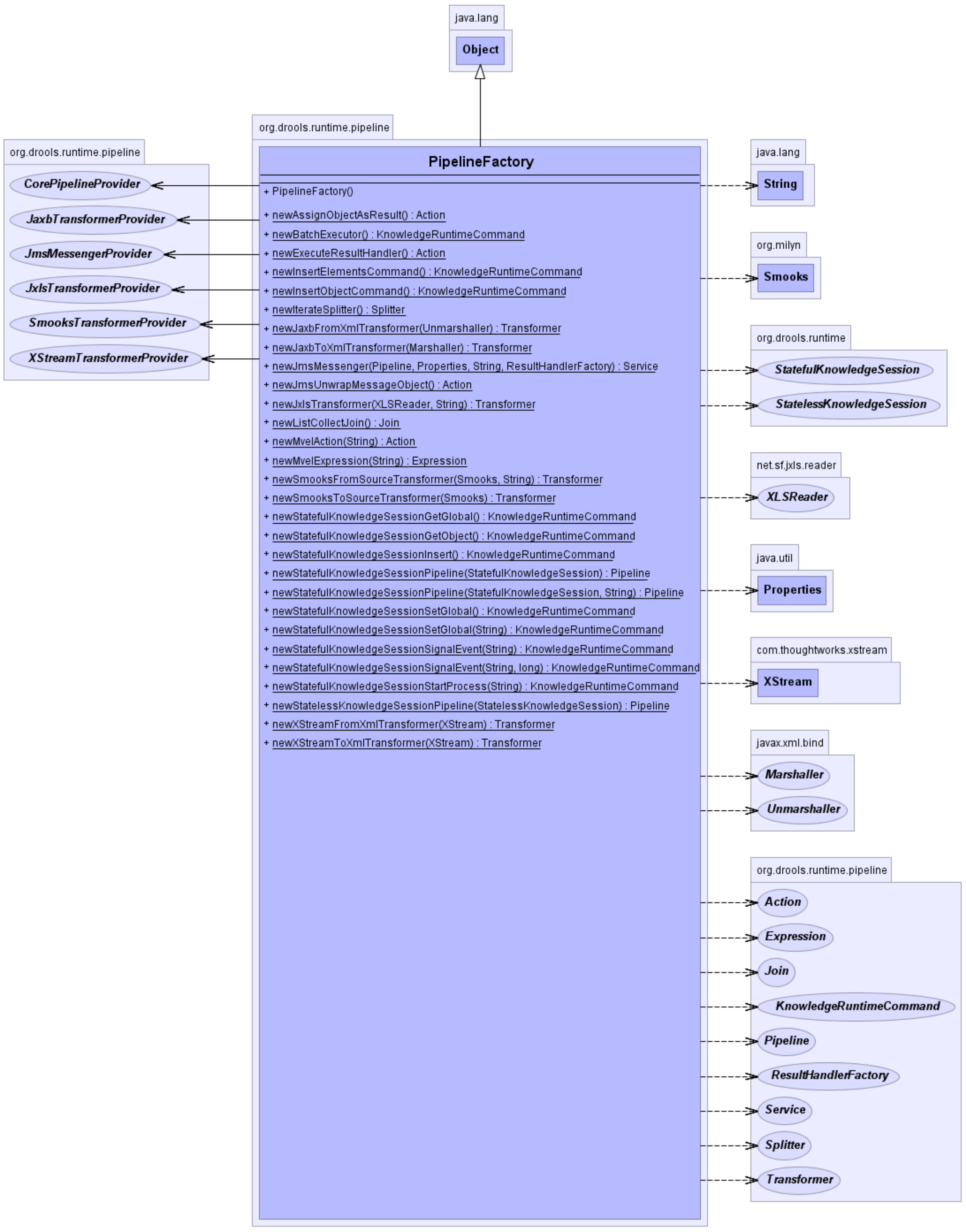

java.lang
Object
org.drools.runtime.pipeline
PipelineFactory
+ PipelineFactory()
+ newAssignObjectAsResult() : Action
+ newBatchExecutor() : KnowledgeRuntimeCommand
+ newExecuteResultHandler() : Action
+ newInsertElementsCommand() : KnowledgeRuntimeCommand
+ newInsertObjectCommand() : KnowledgeRuntimeCommand
+ newIterateSplitter() : Splitter
+ newJaxbFromXmlTransformer(Unmarshaller) : Transformer
+ newJaxbToXmlTransformer(Marshaller) : Transformer
+ newJmsMessenger(Pipeline, Properties, String, ResultHandlerFactory) : Service
+ newJmsUnwrapMessageObject() : Action
+ newJxlsTransformer(XLSReader, String) : Transformer
+ newListCollectJoin() : Join
+ newMvelAction(String) : Action
+ newMvelExpression(String) : Expression
+ newSmooksFromSourceTransformer(Smooks, String) : Transformer
+ newSmooksToSourceTransformer(Smooks) : Transformer
+ newStatefulKnowledgeSessionGetGlobal() : KnowledgeRuntimeCommand
+ newStatefulKnowledgeSessionGetObject() : KnowledgeRuntimeCommand
+ newStatefulKnowledgeSessionInsert() : KnowledgeRuntimeCommand
+ newStatefulKnowledgeSessionPipeline(StatefulKnowledgeSession) : Pipeline
+ newStatefulKnowledgeSessionPipeline(StatefulKnowledgeSession, String) : Pipeline
+ newStatefulKnowledgeSessionSetGlobal() : KnowledgeRuntimeCommand
+ newStatefulKnowledgeSessionSetGlobal(String) : KnowledgeRuntimeCommand
+ newStatefulKnowledgeSessionSignalEvent(String) : KnowledgeRuntimeCommand
+ newStatefulKnowledgeSessionSignalEvent(String, long) : KnowledgeRuntimeCommand
+ newStatefulKnowledgeSessionStartProcess(String) : KnowledgeRuntimeCommand
+ newStatelessKnowledgeSessionPipeline(StatelessKnowledgeSession) : Pipeline
+ newXStreamFromXmlTransformer(XStream) : Transformer
+ newXStreamToXmlTransformer(XStream) : Transformer
org.drools.runtime.pipeline
CorePipelineProvider
JaxbTransformerProvider
JmsMessengerProvider
JxlsTransformerProvider
SmooksTransformerProvider
XStreamTransformerProvider
java.lang
String
org.milyn
Smooks
org.drools.runtime
StatefulKnowledgeSession
StatelessKnowledgeSession
net.sf.jxls.reader
XLSReader
java.util
Properties
com.thoughtworks.xstream
XStream
javax.xml.bind
Marshaller
Unmarshaller
org.drools.runtime.pipeline
Action
Expression
Join
KnowledgeRuntimeCommand
Pipeline
ResultHandlerFactory
Service
Splitter
Transformer

Figure 3.26. PipelineFactory

The **`PipelineFactory`** and associated classes are there to help with the automation of getting information into and out of Drools, especially when using services, such as JMS, and non-POJO data sources. Transformers for Smooks, JAXB, Xstream and Jxls are povided. Smooks is an ETL tooling and can work with a variety of data sources, JAXB is a Java standard aimed at working with XSDs, while XStream is a simple and fast xml serialisation framework and finally Jxls allows for loading of POJOs from an Excel decision table. Minimal information on these technologies is provided here and it is expected for the user to consult the relevant user guide for each.

Pipeline is not meant as a replacement for products like the more powerful Camel, but is aimed as a complimentary framework that ultimately can be integrated into more powerful pipeline frameworks. Instead it is a simple framework aimed at the specific JBoss Rules use cases.

In JBoss Rules a pipeline is a series of stages that operate on and propagate a given payload. Typically this starts with a **`Pipeline`** instance which is responsible for taking the payload, creating a **`PipelineContext`** for it and propagating that to the first Receiver stage. Two types of Pipelines are provided, both requiring a different **`PipelineContexts`**. **`StatefulKnowledgeSessionPipeline`** and **`StatelessKnowledgeSessionPipeline`**. Notice that both factory methods take the relevant session as an argument.

```
Pipeline pipeline =
    PipelineFactory.newStatefulKnowledgeSessionPipeline( ksession );
pipeline.setReceiver( receiver );
```

Example 3.34. StatefulKnowledgeSessionPipeline

A pipeline is then made up of a chain of Stages that can implement both the `Emitter` and the `Receiver` interfaces. The `Emitter` interface means the stage can propagate a payload and the `Receiver` interface means it can receive a payload. This is why the `Pipeline` interface only implements `Emitter` and `Stage` and not `Receiver`, as it is the first instance in the chain. The `Stage` interface allows a custom exception handler to be set on the stage.

```
Transformer transformer =
 PipelineFactory.newXStreamFromXmlTransformer( xstream );
transformer.setStageExceptionHandler( new StageExceptionHandler()
 { .... } );
```

Example 3.35. StageExceptionHandler

The `Transformer` interface above extends both `Stage`, `Emitter` and `Receiver`, other than providing those interface methods as a single type, it's other role is that of a marker interface that indicates the role of the instance that implements it. We have several other marker interfaces such as `Expression` and `Action`, both of which also extend `Stage`, `Emitter` and `Receiver`. One of the stages should be responsible for setting a result value on the `PipelineContext`. It is the role of the `ResultHandler` interface, that the user implements that is responsible for executing on these results or simply setting them an object that the user can retrieve them from.

```java
ResultHandler resultHandler = new ResultHandlerImpl();
pipeline.insert( factHandle, resultHandler );
System.out.println( resultHandler );
...
public class ResultHandlerImpl implements ResultHandler {
    Object result;

    public void handleResult(Object result) {
        this.result = result;
    }

    public Object getResult() {
        return this.result;
    }
}
```

Example 3.36. StageExceptionHandler

While the above example shows a simple handler that simply assigns the result to a field that the user can access, it could do more complex work like sending the object as a message.

Pipeline is provides an adapter to insert the payload and internally create the correct PipelineContext. Two types of Pipelines are provided, both requiring a different PipelineContext. StatefulKnowledgeSessionPipeline and StatelessKnowledgeSessionPipeline. Pipeline itself implements both Stage and Emitter, this means it's a Stage in a pipeline and emits the payload to a receiver. It does not implement Receiver itself, as it the start adapter for the pipeline. PipelineFactory provides methods to create both of the two Pipeline. StatefulKnowledgeSessionPipeline is constructed as below, with the receiver set.

In general it easier to construct the pipelines in reverse, for example the following one handles loading xml data from disk, transforming it with xstream and then inserting the object:

```
// Make the results, in this case the FactHandles, available to the user
Action executeResultHandler = PipelineFactory.newExecuteResultHandler();

// Insert the transformed object into the session
// associated with the PipelineContext
KnowledgeRuntimeCommand insertStage =
    PipelineFactory.newStatefulKnowledgeSessionInsert();
insertStage.setReceiver( executeResultHandler );

// Create the transformer instance and create the Transformer stage,
// where we are going from Xml to Pojo.
XStream xstream = new XStream();
Transformer transformer =
    PipelineFactory.newXStreamFromXmlTransformer( xstream );
transformer.setReceiver( insertStage );

// Create the start adapter Pipeline for StatefulKnowledgeSessions
Pipeline pipeline =
    PipelineFactory.newStatefulKnowledgeSessionPipeline( ksession );
pipeline.setReceiver( transformer );

// Instantiate a simple result handler and load and insert the XML
ResultHandlerImpl resultHandler = new ResultHandlerImpl();
pipeline.insert( ResourceFactory.newClassPathResource(
    "path/facts.xml", getClass() ), resultHandler );
```

Example 3.37. Constructing a pipeline

While the above example is for loading a resource from disk it is also possible to work from a running messaging service. JBoss Rules currently provides a single Service for JMS, called JmsMessenger. Support for other Services will be added later. Below shows part of a unit test which illustrates part of the JmsMessenger in action:

```
// as this is a service, it's more likely the results will
// be logged or sent as a return message
Action resultHandlerStage = PipelineFactory.newExecuteResultHandler();

// Insert the transformed object into the session
// associated with the PipelineContext
KnowledgeRuntimeCommand insertStage =
    PipelineFactory.newStatefulKnowledgeSessionInsert();
insertStage.setReceiver( resultHandlerStage );

// Create the transformer instance and create the Transformer stage,
// where we are going from Xml to Pojo. Jaxb needs an array of the
// available classes
JAXBContext jaxbCtx =
    KnowledgeBuilderHelper.newJAXBContext( classNames,kbase );
Unmarshaller unmarshaller = jaxbCtx.createUnmarshaller();
Transformer transformer =
    PipelineFactory.newJaxbFromXmlTransformer( unmarshaller );
transformer.setReceiver( insertStage );

// payloads for JMS arrive in a Message wrapper,
// we need to unwrap this object
Action unwrapObjectStage = PipelineFactory.newJmsUnwrapMessageObject();
unwrapObjectStage.setReceiver( transformer );

// Create the start adapter Pipeline for StatefulKnowledgeSessions
Pipeline pipeline =
    PipelineFactory.newStatefulKnowledgeSessionPipeline( ksession );
pipeline.setReceiver( unwrapObjectStage );

// Services, like JmsMessenger take a ResultHandlerFactory
// implementation, this is because a result handler must be
// created for each incoming message
ResultHandleFactoryImpl factory = new ResultHandleFactoryImpl();
Service messenger =
   PipelineFactory.newJmsMessenger(pipeline,props,destinationName,factory);
messenger.start();
```

Example 3.38. Using JMS with Pipeline

3.3.8.1. Xstream Transformer

```
XStream xstream = new XStream();
Transformer transformer =
    PipelineFactory.newXStreamFromXmlTransformer( xstream );
transformer.setReceiver( nextStage );
```

Example 3.39. XStream FromXML transformer stage

```
XStream xstream = new XStream();
Transformer transformer =
 PipelineFactory.newXStreamToXmlTransformer( xstream );
transformer.setReceiver( receiver );
```

Example 3.40. XStream ToXML transformer stage

3.3.8.2. JAXB Transformer

```
Options xjcOpts = new Options();
xjcOpts.setSchemaLanguage( Language.XMLSCHEMA );
KnowledgeBuilder kbuilder = KnowledgeBuilderFactory.newKnowledgeBuilder();

String[] classNames = KnowledgeBuilderHelper.addXsdModel(
    ResourceFactory.newClassPathResource( "order.xsd", getClass() ),
    kbuilder, xjcOpts, "xsd" );
```

Example 3.41. JAXB XSD Generation into the KnowlegeBuilder

```
JAXBContext jaxbCtx =
    KnowledgeBuilderHelper.newJAXBContext( classNames, kbase );
Unmarshaller unmarshaller = jaxbCtx.createUnmarshaller();
Transformer transformer =
    PipelineFactory.newJaxbFromXmlTransformer( unmarshaller );
transformer.setReceiver( receiver );
```

Example 3.42. JAXB FromXML transformer stage

```
Marshaller marshaller = jaxbCtx.createMarshaller();
Transformer transformer =
    PipelineFactory.newJaxbToXmlTransformer( marshaller );
transformer.setReceiver( receiver );
```

Example 3.43. JAXB ToXML transformer stage

3.3.8.3. Smooks Transformer

```
Smooks smooks = new Smooks( getClass().getResourceAsStream( "smooks-
config.xml" ) );
Transformer transformer =
 PipelineFactory.newSmooksFromSourceTransformer( smooks,

  "orderItem" );
transformer.setReceiver( receiver );
```

Example 3.44. Smooks FromSource transformer stage

```
Smooks smooks = new Smooks( getClass().getResourceAsStream( "smooks-
config.xml" ) );

Transformer transformer =
 PipelineFactory.newSmooksToSourceTransformer( smooks );
transformer.setReceiver( receiver );
```

Example 3.45. Smooks ToSource transformer stage

3.3.8.4. JXLS (Excel/Calc/CSV) Transformer

Transforms from an Excel spreadsheet to a Map of POJOs using jXLS, the resulting map is set as the propagating object. You may need to use splitters and MVEL expressions to split up the transformation to insert individual pojos. Note you must provde an XLSReader, which references the mapping file and also an MVEL string which will instantiate the map. The mvel expression is pre-compiled but executedon each usage of the transformation.

```

XLSReader mainReader = ReaderBuilder.buildFromXML(
    ResourceFactory.newClassPathResource(
    "departments.xml", getClass() ).getInputStream() );

Transformer transformer =
    PipelineFactory.newJxlsTransformer(
        mainReader, "[ 'departments' : new java.util.ArrayList(),
        'company' : new org.drools.runtime.pipeline.impl.Company() ]");
```

Example 3.46. JXLS transformer stage

3.3.8.5. JMS Messenger

Creates a new JmsMessenger which runs as a service in it's own thread. It expects an existing JNDI entry for "ConnectionFactory" Which will be used to create the MessageConsumer which will feed into the specified pipeline.

```
// as this is a service, it's more likely the results
// will be logged or sent as a return message
Action resultHandlerStage = PipelineFactory.newExecuteResultHandler();

// Insert the transformed object into the session
// associated with the PipelineContext
KnowledgeRuntimeCommand insertStage =
    PipelineFactory.newStatefulKnowledgeSessionInsert();
insertStage.setReceiver( resultHandlerStage );

// Create the transformer instance and create the Transformer
// stage, where we are going from Xml to Pojo. Jaxb needs an
// array of the available classes
JAXBContext jaxbCtx =
    KnowledgeBuilderHelper.newJAXBContext( classNames, kbase );
Unmarshaller unmarshaller = jaxbCtx.createUnmarshaller();
Transformer transformer =
    PipelineFactory.newJaxbFromXmlTransformer( unmarshaller );
transformer.setReceiver( insertStage );

// payloads for JMS arrive in a Message wrapper,
// we need to unwrap this object
Action unwrapObjectStage =
    PipelineFactory.newJmsUnwrapMessageObject();
unwrapObjectStage.setReceiver( transformer );

// Create the start adapter Pipeline for StatefulKnowledgeSessions
Pipeline pipeline =
    PipelineFactory.newStatefulKnowledgeSessionPipeline( ksession );
pipeline.setReceiver( unwrapObjectStage );

// Services, like JmsMessenger take a ResultHandlerFactory
// implementation, this is because a result handler must be
// created for each incoming message.
ResultHandleFactoryImpl factory = new ResultHandleFactoryImpl();
Service messenger =
   PipelineFactory.newJmsMessenger(pipeline,props,destinationName,factory);
```

Example 3.47. JMS Messenger stage

3.3.9. Commands and the CommandExecutor

JBoss Rules has the concept of stateful or stateless sessions, we've already covered stateful. Where stateful is the standard working memory that can be worked with iteratively over time. Stateless is a one off execution of a working memory with a provided data set and optionally returning some results with the session disposed at the end, prohibiting further iterative interactions. You can think of stateless as treating a rule engine like a function call with optional return results.

In previous versions we supported these two paradigms but the way the user interacted with them was different. StatelessSession used an execute(...) method which would insert a collection of objects as facts. StatefulSession didn't have this method and insert used the more traditional insert(...) method.

The other issue was the StatelessSession did not return any results, the user was expected to map globals themselves to get results, and it wasn't possible to do anything else other than insert objects, users could not start processes or execute querries.

JBoss Rules 5.0 addresses all of these issues and more. The foundations for this is the CommandExecutor interface, which both the stateful and stateless interfaces extend creating consistency and ExecutionResults:

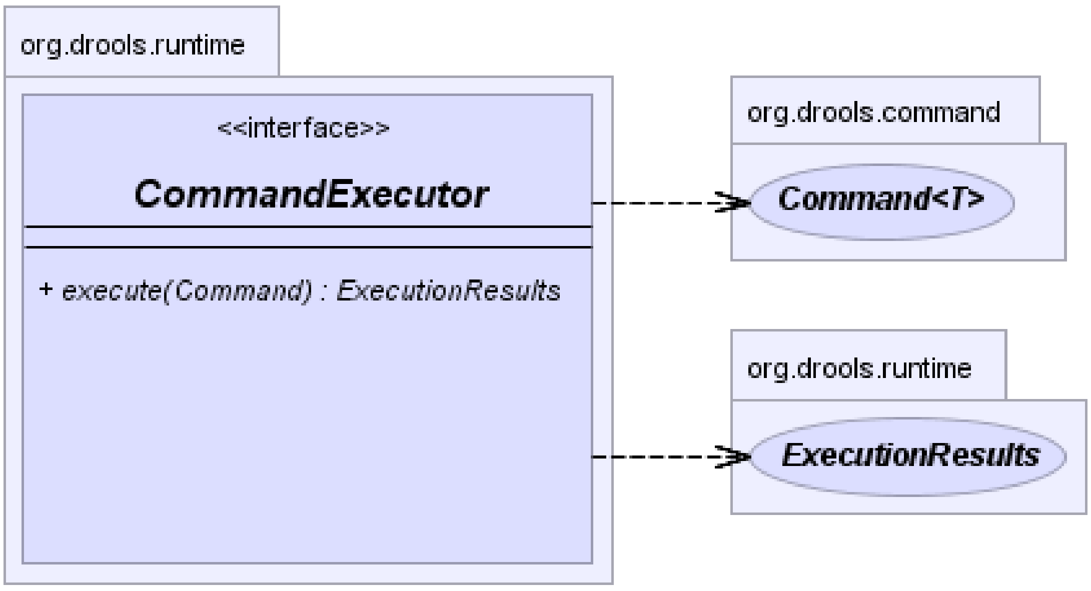


Figure 3.27. CommandExecutor

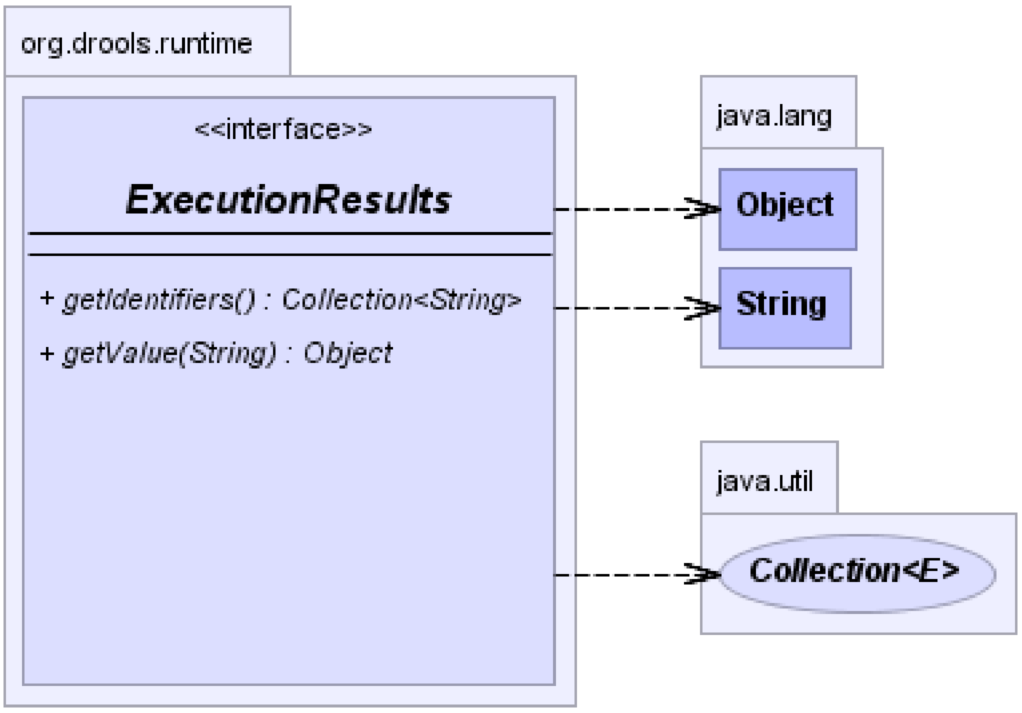


Figure 3.28. ExecutionResults

The CommandFactory allows for commands to be executed on those sessions, only only difference being the StatelessKnowledgeSession executes fireAllRules() at the end before disposing the session.

The current supported commands are:

FireAllRules	GetGlobal
SetGlobal	InsertObject
InsertElements	Query
StartProcess	BatchExecution

InsertObject will insert a single object, with an optional out identifier. InsertElements will iterate an Iterable inserting each of the elements. What this means is that StatelessKnowledgeSession are no longer limited to just inserting objects, they can now start processes or execute queries and in any order.

```
StatelessKnowledgeSession ksession =
    kbase.newStatelessKnowledgeSession();
ExecutionResults bresults = ksession.execute(
    CommandFactory.newInsert( new Cheese( "stilton" ), "stilton_id" ) );
Stilton stilton = bresults.getValue( "stilton_id" );
```

Example 3.48. Insert Command

The execute method always returns an ExecutionResults instance, which allows access to any command results if they specify an out identifier such as the "stilton_id" above.

```
StatelessKnowledgeSession ksession = kbase.newStatelessKnowledgeSession();
Command cmd = CommandFactory.newInsertElements(
    Arrays.asList( Object[] {
        new Cheese("stilton"), new Cheese("brie"), new Cheese("cheddar"),
    });

ExecutionResults bresults = ksession.execute( cmd );
```

Example 3.49. InsertElements Command

What you say, the method only allows for a single command? That's Where the BatchExecution comes in, this is a composite command that takes a list of commands and will iterate and execute each command in turn. This means you can insert some objects, start a process, call fireAllRules and execute a query all in a single execute(...) call - much more powerful.

As mentioned the StatelessKnowledgeSession by default will execute fireAllRules() automatically at the end. However the keen eyed reader probably has already noticed the FireAllRules command and wondering how that works with a StatelessKnowledgeSession. The FireAllRules command is allowed and using it will disable the automatic execution at the end, think of using it as a sort of manual override.

StatelessKnowledgeSession and StatefullKnowledgeSession work in a manner consistent with each other and also brought in support for more than just inserting objects. What about result handling? Rather than using parameters, like my first attempt which always bugged me, these commands support out identifiers. Any command that has an out identifier set on it will add it's results to the returned ExecutionResults instance. Let's look at a simple example to see how this works.

```
StatelessKnowledgeSession ksession = kbase.newStatelessKnowledgeSession();

List cmds = new ArrayList();
cmds.add(
    CommandFactory.newInsertObject(new Cheese("stilton", 1), "stilton"));
cmds.add( CommandFactory.newStartProcess("process cheeses"));
cmds.add( CommandFactory.newQuery("cheeses"));
ExecutionResults bresults =
    ksession.execute(CommandFactory.newBatchExecution(cmds));
QueryResults qresults = (QueryResults)bresults.getValue("cheeses");
Cheese stilton = ( Cheese ) bresults.getValue( "silton" );
```

Example 3.50. BatchExecution Command

So in the above example you saw how multiple commands where executed two of which populate the ExecutionResults. The query command defaults to use the same identifier as the query name, but it can also be mapped to a different identifier.

So now we have consistency across stateless and stateful sessions, ability to execute a variety of commands and an elegant way to deal with results. Does it get better than this? Absolutely we've built a custom XStream marshaller that can be used with the JBoss Rules' Pipeline to get XML scripting, which is perfect for services. Here are two simple XML samples for the BatchExecution and ExecutionResults.

```
<batch-execution>
   <insert out-identifier='outStilton'>
      <org.drools.Cheese>
         <type>stilton</type>
         <price>25</price>
         <oldPrice>0</oldPrice>
      </org.drools.Cheese>
   </insert>
</batch-execution>
```

Example 3.51. Simple BatchExecution XML

```
<execution-results>
   <result identifier='outStilton'>
      <org.drools.Cheese>
         <type>stilton</type>
         <oldPrice>0</oldPrice>
         <price>30</price>
      </org.drools.Cheese>
   </result>
</execution-results>
```

Example 3.52. Simple ExecutionResults XML

I've mentioned the pipeline previously, it allows for a series of stages to be used together to help with getting data into and out of sessions. There is a stage that supports the CommandExecutor interface and allows the pipeline to script either a stateful or stateless session. The pipeline setup is trivial:

```
Action executeResultHandler = PipelineFactory.newExecuteResultHandler();

Action assignResult = PipelineFactory.newAssignObjectAsResult();

assignResult.setReceiver( executeResultHandler );

Transformer outTransformer =
    PipelineFactory.newXStreamToXmlTransformer(
BatchExecutionHelper.newXStreamMarshaller() );

outTransformer.setReceiver( assignResult );

KnowledgeRuntimeCommand cmdExecution =
    PipelineFactory.newCommandExecutor();

batchExecution.setReceiver( cmdExecution );

Transformer inTransformer =
    PipelineFactory.newXStreamFromXmlTransformer(
        BatchExecutionHelper.newXStreamMarshaller() );

inTransformer.setReceiver( batchExecution );

Pipeline pipeline =
    PipelineFactory.newStatelessKnowledgeSessionPipeline( ksession );

pipeline.setReceiver( inTransformer );
```

Example 3.53. Pipeline use for CommandExecutor

The key thing here to note is the use of the BatchExecutionHelper to provide a specially configured XStream with custom converters for our Commands and the new BatchExecutor stage.

Using the pipeline is very simple, you must provide your own implementation of the ResultHandler, which is called if the pipeline executes the ExecuteResultHandler stage.

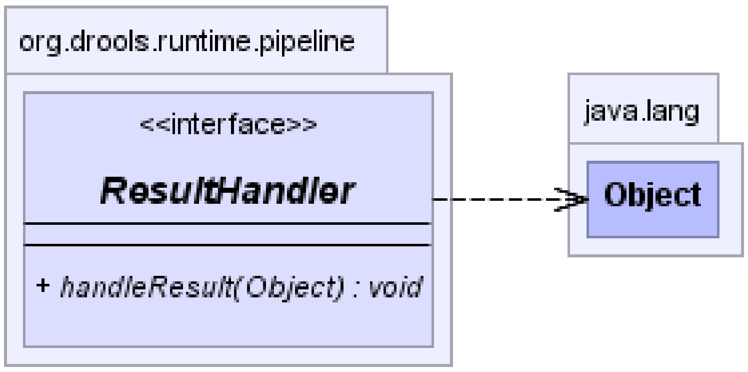


Figure 3.29. Pipeline ResultHandler

```
public static class ResultHandlerImpl implements ResultHandler
{
    Object object;

    public void handleResult(Object object)
    {
       this.object = object;
    }

    public Object getObject()
    {
        return this.object;
    }
}
```

Example 3.54. Simple Pipeline ResultHandler

```
ResultHandler resultHandler = new ResultHandlerImpl();
pipeline.insert( inXml, resultHandler );
```

Example 3.55. Using a Pipeline

Earlier a BatchExecution was created with Java to insert some objects and execute a query. The XML representation to be used with the pipeline for that example is shown below with parameters added to the query.

```
<batch-execution>
  <insert out-identifier="stilton">
    <org.drools.Cheese>
      <type>stilton</type>
      <price>1</price>
    </org.drools.Cheese>
  </insert>
  <query out-identifier='cheeses2' name='cheesesWithParams'>
    <string>stilton</string>
    <string>cheddar</string>
  </query>
</batch-execution>
```

Example 3.56. BatchExecution Marshalled to XML

The CommandExecutor returns an ExecutionResults, this too is handled by the pipeline code snippet. A similar output for the <batch-execution> xml sample above would be:

```

<execution-results>
  <result identifier="stilton">
    <org.drools.Cheese>
      <type>stilton</type>
      <price>2</price>
    </org.drools.Cheese>
  </result>
  <result identifier='cheeses2'>
    <query-results>
      <identifiers>
        <identifier>cheese</identifier>
      </identifiers>
      <row>
        <org.drools.Cheese>
          <type>cheddar</type>
          <price>2</price>
          <oldPrice>0</oldPrice>
        </org.drools.Cheese>
      </row>
      <row>
        <org.drools.Cheese>
          <type>cheddar</type>
          <price>1</price>
          <oldPrice>0</oldPrice>
        </org.drools.Cheese>
      </row>
    </query-results>
  </result>
</execution-results>
```

Example 3.57. ExecutionResults Marshalled to XML

The BatchExecutionHelper provides a configured XStream instance to support the marshaling of BatchExecutions, where the resulting xml can be used as a message format, as shown above. Configured converters only exist for the commands supported via the CommandFactory. The user may add other converters for their user objects. This is very useful for scripting stateless of stateful knowledge sessions, especially when services are involved.

There is current no XSD for schema validation, however we will try to outline the basic format here and the drools-transformer-xstream module has an illustrative unit test in the XStreamBatchExecutionTest unit test. The root element is <batch-execution> and it can contain zero or more commands elements.

```
<batch-execution>
...
</batch-execution>
```

Example 3.58. Root XML element

This contains a list of elements that represent commands, the supported commands is limited to those Commands provided by the CommandFactory. The most basic of these is the <insert> element, which inserts objects. The contents of the insert element is the user object, as dictated by XStream.

```
<batch-execution>
   <insert>
      ....
   </insert>
</batch-execution>
```

Example 3.59. Insert with Out Identifier Command

The insert element supports an 'out-identifier' attribute, this means the insert object will also be returned as part of the payload.

```
<batch-execution>
   <insert out-identifier='userVar'>
      ....
   </insert>
</batch-execution>
```

Example 3.60. Insert with Out Identifier Command

It's also possible to insert a collection of objects using the <insert-elements> element, however this command does not support an out-identifier. The org.domain.UserClass is just an example user object that xstream would serialize.

```
<batch-execution>
   <insert-elements>
      <org.domain.UserClass>
         ...
      </org.domain.UserClass>
      <org.domain.UserClass>
         ...
      </org.domain.UserClass>
      <org.domain.UserClass>
         ...
      </org.domain.UserClass>
   </insert-elements>
</batch-execution>
```

Example 3.61. Insert Elements command

Next there is the <set-global> element, which sets a global for the session.

```
<batch-execution>
   <set-global identifier='userVar'>
      <org.domain.UserClass>
         ...
      </org.domain.UserClass>
   </set-global>
</batch-execution>
```

Example 3.62. Insert Elements command

<set-global> also supports two other optional attributes 'out' and 'out-identifier'. 'out' is a boolean and when set the global will be added to the <batch-execution-results> payload using the name from the 'identifier' attribute. 'out-identifier' works like 'out' but additionally allows you to override the identifier used in the <batch-execution-results> payload.

```
<batch-execution>
   <set-global identifier='userVar1' out='true'>
      <org.domain.UserClass>
         ...
      </org.domain.UserClass>
   </set-global>
   <set-global identifier='userVar2' out-identifier='alternativeUserVar2'>
      <org.domain.UserClass>
         ...
      </org.domain.UserClass>
   </set-global>
</batch-execution>
```

Example 3.63. Set Global Command

There is also a <get-global> element, which has no contents but does support an 'out-identifier' attribute, there is no need for an 'out' attribute as we assume that a <get-global> is always an 'out'.

```
<batch-execution>
   <get-global identifier='userVar1' />
   <get-global identifier='userVar2' out-identifier='alternativeUserVar2'/>
</batch-execution>
```

Example 3.64. Get Global Command

While the 'out' attribute is useful in returning specific instances as a result payload, we often wish to run actual queries. Both parameter and parameterless queries are supported. The 'name' attribute is the name of the query to be called, and the 'out-identifier' is the identifier to be used for the query results in the <execution-results> payload.

```
<batch-execution>
   <query out-identifier='cheeses' name='cheeses'/>
   <query out-identifier='cheeses2' name='cheesesWithParams'>
      <string>stilton</string>
      <string>cheddar</string>
   </query>
</batch-execution>
```

Example 3.65. Query Command

JBoss Rules is no longer just about rules, as the <start-process> command is also supported and accepts optional parameters. Other process related methods will be added later, like interacting with work items.

```
<batch-execution>
   <startProcess processId='org.drools.actions'>
      <parameter identifier='person'>
         <org.drools.TestVariable>
            <name>John Doe</name>
          </org.drools.TestVariable>
       </parameter>
   </startProcess>
</batch-execution
```

Example 3.66. Start Process Command

Support for more commands will be added over time.

3.3.10. Marshaling

The MarshallerFactory is used to marshal and un-marshal StatefulKnowledgeSessions.

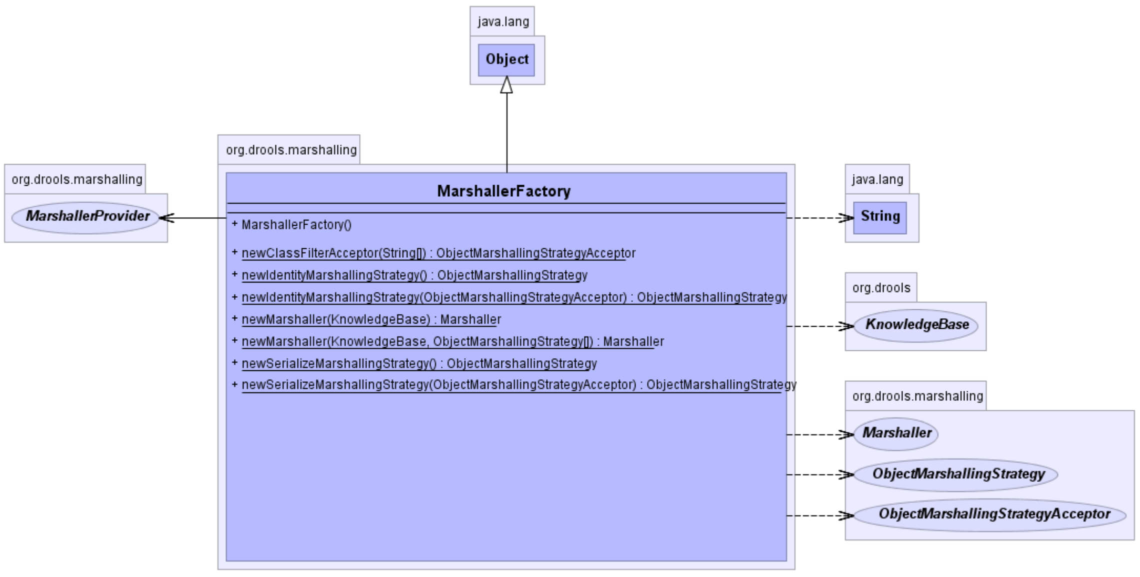

java.lang
Object
org.drools.marshalling
MarshallerProvider
org.drools.marshalling
MarshallerFactory
+ MarshallerFactory()
+ newClassFilterAcceptor(String[]) : ObjectMarshallingStrategyAcceptor
+ newIdentityMarshallingStrategy() : ObjectMarshallingStrategy
+ newIdentityMarshallingStrategy(ObjectMarshallingStrategyAcceptor) : ObjectMarshallingStrategy
+ newMarshaller(KnowledgeBase) : Marshaller
+ newMarshaller(KnowledgeBase, ObjectMarshallingStrategy[]) : Marshaller
+ newSerializeMarshallingStrategy() : ObjectMarshallingStrategy
+ newSerializeMarshallingStrategy(ObjectMarshallingStrategyAcceptor) : ObjectMarshallingStrategy
java.lang
String
org.drools
KnowledgeBase
org.drools.marshalling
Marshaller
ObjectMarshallingStrategy
ObjectMarshallingStrategyAcceptor

Figure 3.30. MarshallerFactory

At the simplest the MarshallerFactory can be used as follows:

```
// ksession is the StatefulKnowledgeSession
// kbase is the KnowledgeBase
ByteArrayOutputStream baos = new ByteArrayOutputStream();
Marshaller marshaller = MarshallerFactory.newMarshaller( kbase );
marshaller.marshall( baos, ksession );
baos.close();
```

Example 3.67. Simple Marshaller Example

However with marshaling you need more flexibility when dealing with referenced user data. To achieve this we have the ObjectMarshallingStrategy interface. Two implementations are provided, but the user can implement their own. The two supplied are IdentityMarshallingStrategy and SerializeMarshallingStrategy. SerializeMarshallingStrategy is the default, as used in the example above and it just calls the Serializable or Externalizable methods on a user instance. IdentityMarshallingStrategy instead creates an integer id for each user object and stores them in a Map the id is written to the stream. When un-marshaling it simply looks to the IdentityMarshallingStrategy map to retrieve the instance. This means that if you use the IdentityMarshallingStrategy it's stateful for the life of the Marshaller instance and will create ids and keep references to all objects that it attempts to marshal. Here is he code to use a IdentityMarshallingStrategy.

```
ByteArrayOutputStream baos = new ByteArrayOutputStream();
Marshaller marshaller = MarshallerFactory.newMarshaller( kbase, new
 ObjectMarshallingStrategy[]
 { MarshallerFactory.newIdentityMarshallingStrategy() } );
marshaller.marshall( baos, ksession );
baos.close();
```

Example 3.68. IdentityMarshallingStrategy

For added flexability we can't assume that a single strategy is suitable for this we have added the ObjectMarshallingStrategyAcceptor interface that each ObjectMarshallingStrategy has. The Marshaller has a chain of strategies and when it attempts to read or write a user object it iterates the strategies asking if they accept responsability for marshalling the user object. One one implementation is provided the ClassFilterAcceptor. This allows strings and wild cards to be used to match class names. The default is "*.*", so in the above the IdentityMarshallingStrategy is used which has a default "*.*" acceptor.

But lets say we want to serialise all classes except for one given package, where we will use identity lookup, we could do the following:

```
ByteArrayOutputStream baos = new ByteArrayOutputStream();

ObjectMarshallingStrategyAcceptor identityAceceptor =
    MarshallerFactory.newClassFilterAcceptor(
        new String[] { "org.domain.pkg1.*" } );

ObjectMarshallingStrategy identityStratetgy =
    MarshallerFactory.newIdentityMarshallingStrategy( identityAceceptor );

Marshaller marshaller =
    MarshallerFactory.newMarshaller( kbase,
        new ObjectMarshallingStrategy[] { identityStratetgy,
            MarshallerFactory.newSerializeMarshallingStrategy() } );

marshaller.marshall( baos, ksession );
baos.close();
```

Example 3.69. IdentityMarshallingStrategy with Acceptor

Note that the acceptance checking order is in the natural order of the supplied array.

3.3.11. Persistence and Transactions

Long term out of the box persistence with JPA is possible with JBoss Rules. You will need to have JTA installed. The Bitronix JTA Transaction Manager is suitable for development but JBoss Transactions is recommended for production environments.

You can find additional information about Bitronix at *http://docs.codehaus.org/display/BTM/Home*

```
Environment env = KnowledgeBaseFactory.newEnvironment();

env.set( EnvironmentName.ENTITY_MANAGER_FACTORY,
    Persistence.createEntityManagerFactory( "emf-name" ) );

env.set( EnvironmentName.TRANSACTION_MANAGER,
    TransactionManagerServices.getTransactionManager() );

// KnowledgeSessionConfiguration may be null, and a default will be used

StatefulKnowledgeSession ksession =
    JPAKnowledgeService.newStatefulKnowledgeSession( kbase, null, env );

int sessionId = ksession.getId();

UserTransaction ut = (UserTransaction) new
    InitialContext().lookup( "java:comp/UserTransaction" );

ut.begin();
ksession.insert( data1 );
ksession.insert( data2 );
ksession.startProcess( "process1" );
ut.commit();
```

Example 3.70. Simple example using transactions

To use a JPA the Environment must be set with both the EntityManagerFactory and the TransactionManager. If rollback occurs the ksession state is also rolled back, so you can continue to use it after a rollback. To load a previous persisted StatefulKnowledgeSession you'll need the id, as shown below:

```
StatefulKnowledgeSession ksession =
 JPAKnowledgeService.loadStatefulKnowledgeSession( sessionId, kbase, null,
 env );
```

Example 3.71. Loading a StatefulKnowledgeSession

To enable persistence the following classes must be added to your **`persistence.xml`**, as in the example below:

```
<persistence-unit name="org.drools.persistence.jpa" transaction-type="JTA">
    <provider>org.hibernate.ejb.HibernatePersistence</provider>
    <jta-data-source>jdbc/BitronixJTADataSource</jta-data-source>
    <class>org.drools.persistence.session.SessionInfo</class>
    <class>
        org.drools.persistence.processinstance.ProcessInstanceInfo
    </class>
    <class>
        org.drools.persistence.processinstance.ProcessInstanceEventInfo
    </class>
    <class>org.drools.persistence.processinstance.WorkItemInfo</class>
    <properties>
        <property name="hibernate.dialect"
            value="org.hibernate.dialect.H2Dialect"/>
        <property name="hibernate.max_fetch_depth" value="3"/>
        <property name="hibernate.hbm2ddl.auto" value="update" />
        <property name="hibernate.show_sql" value="true" />
        <property name="hibernate.transaction.manager_lookup_class"
           value="org.hibernate.transaction.BTMTransactionManagerLookup" />
    </properties>
</persistence-unit>
```

Example 3.72. Configuring JPA

The JBDC JTA data source would need to be previously bound. Bitronix provides a number of ways of doing this and it's docs should be consulted for more details, however for quick start help here is the programmatic approach:

```
PoolingDataSource ds = new PoolingDataSource();
ds.setUniqueName( "jdbc/BitronixJTADataSource" );
ds.setClassName( "org.h2.jdbcx.JdbcDataSource" );
ds.setMaxPoolSize( 3 );
ds.setAllowLocalTransactions( true );
ds.getDriverProperties().put( "user", "sa" );
ds.getDriverProperties().put( "password", "sasa" );
ds.getDriverProperties().put( "URL", "jdbc:h2:mem:mydb" );
ds.init();
```

Example 3.73. Configuring JTA DataSource

Bitronix also provides a simple embedded JNDI service, ideal for testing, to use it add a **`jndi.properties`** file to your **`META-INF`** and add the following line to it:

```
java.naming.factory.initial=bitronix.tm.jndi.BitronixInitialContextFactory
```

Example 3.74. JNDI properties

The Rule Language

4.1. Overview

Jboss Rules has a "native" rule language that is non XML textual format. This format is very light in terms of punctuation, and supports natural and domain specific languages via "expanders" that allow the language to morph to your problem domain. This chapter is mostly concerted with the native rule format. The Diagrams used are known as "rail road" diagrams, and are basically flow charts for the language terms. For the technically very keen, you can also refer to "DRL.g" which is the Antlr3 grammar for the rule language. If you use the Rule Workbench, a lot of the rule structure is done for you with content assistance, for example, type "ru" and press ctrl+space, and it will build the rule structure for you.

4.1.1. A rule file

A rule file is typically a file with a .drl extension. In a drl file you can have multiple rules, queries and functions, as well as some resource declarations like imports, globals and attributes that are assigned and used by your rules and queries. However, you are also able to spread your rules across multiple rule files (in that case, the extension .rule is suggested, but not required) - spreading rules across files can help with managing large numbers of rules. A DRL file is simply a text file.

The overall structure of a rule file is:

```
>package package-name

imports

globals

functions

queries

rules
```

Example 4.1. Rules file

The order in which the elements are declared is not important, except for the package name that, if declared, must be the first element in the rules file. All elements are optional, so you will use only those you need. We will discuss each of them in the following sections.

4.1.2. What makes a rule

A rule has the following rough structure:

```
rule "name"
    attributes
when
    LHS
then
```

```
    RHS
end
```

It's really that simple. Mostly punctuation is not needed, even the double quotes for "name" are optional, as are newlines. Attributes are simple (always optional) hints to how the rule should behave. LHS is the conditional parts of the rule, which follows a certain syntax which is covered below. RHS is basically a block that allows dialect specific semantic code to be executed.

It is important to note that white space is not important, *except* in these case of domain specific languages, in which case each line is processed before the following line (and spaces may be significant to the domain language).

4.2. Keywords

JBoss Rules 5 introduces the concept of Hard and Soft keywords.

Hard keywords are reserved, you cannot use any hard keyword when naming your domain objects, properties, methods, functions and other elements that are used in the rule text.

Here is a list of hard keywords that must be avoided as identifiers when writing rules:

true	false	accumulate
collect	from	null
over	then	when

Soft keywords are just recognized in their context, enabling you to use this words in any other place you wish. Here is a list of soft keywords:

lock-on-active	date-effective	date-expires
no-loop	auto-focus	activation-group
agenda-group	ruleflow-group	entry-point
duration	package	import
dialect	salience	enabled
attributes	rule	extend
template	query	declare
function	global	eval
not	in	or
and	exists	forall
action	reverse	result
end	init	

Of course, you can have these (hard and soft) words as part of a method name in camel case, like notSomething() or accumulateSomething() - there are no issues with that scenario.

Another improvement on DRL language is the ability to escape hard keywords on rule text. This new feature enables you to use your existing domain objects without worrying about keyword collision. To escape a word, simple type it between grave accents, like this:

```
Holiday( `when` == "july" )
```

The escape should be used everywhere in rule text, except within code expressions in the LHS or RHS code block. Here are examples of usage:

```
rule "validate holiday by eval"
dialect "mvel"
when
    h1 : Holiday( )
    eval( h1.when == "july" )
then
    System.out.println(h1.name + ":" + h1.when);
end
```

```
rule "validate holiday"
dialect "mvel"
when
    h1 : Holiday( `when` == "july" )
then
    System.out.println(h1.name + ":" + h1.when);
end
```

4.3. Comments

Comments are sections of text that are ignored by the rule engine. They are stripped out when they are encountered, except inside semantic code blocks, like the RHS of a rule.

4.3.1. Single line comment

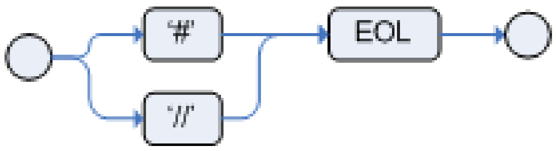


Figure 4.1. Single line comment

To create single line comments, you can use either '#' or '//'. The parser will ignore anything in the line after the comment symbol. Example:

```
rule "Testing Comments"
when
    # this is a single line comment
    // this is also a single line comment
    eval( true ) # this is a comment in the same line of a pattern
then
    // this is a comment inside a semantic code block
    # this is another comment in a semantic code block
end
```

4.3.2. Multi-line comment

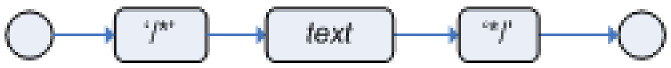


Figure 4.2. Multi-line comment

Multi-line comments are used to comment blocks of text, both in and outside semantic code blocks. Example:

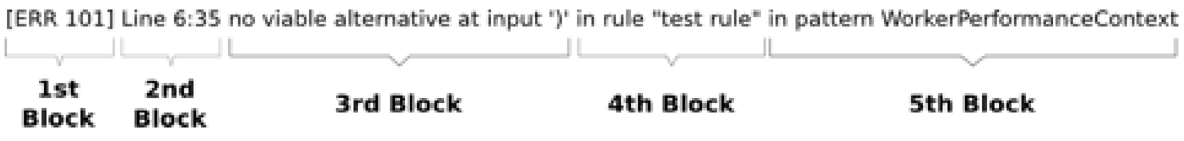
```
rule "Test Multi-line Comments"
when
    /* this is a multi-line comment
       in the left hand side of a rule */
    eval( true )
then
    /* and this is a multi-line comment
       in the right hand side of a rule */
end
```

4.4. *Error Messages*

JBoss Rules 5 introduces standardized error messages. This standardization aims to help users to find and resolve problems in a easier and faster way. In this section you will learn how to identify and interpret those error messages, and you will also receive some tips on how to solve the problems associated with them.

4.4.1. Message format

The standardization includes the error message format and to better explain this format, let's use the following example:



```
[ERR 101] Line 6:35 no viable alternative at input ')' in rule "test rule" in pattern WorkerPerformanceContext
```

Figure 4.3. Error Message Format

1st Block: This area identifies the error code.

2nd Block: Line:column information.

3rd Block: Some text describing the problem.

4th Block: This is the first context. Usually indicates the rule, function, template or query where the error occurred. This block is not mandatory.

5th Block: Identifies the pattern where the error occurred. This block is not mandatory.

4.4.2. Error Messages Description

4.4.2.1. 101: No viable alternative

Indicates the most common errors, where the parser came to a decision point but couldn't identify an alternative. Here are some examples:

```
rule one
when
```

```
    exists Foo()
    exits Bar()
then
end
```

The above example generates this message:

- [ERR 101] Line 4:4 no viable alternative at input 'exits' in rule one

At first glance this seems to be valid syntax, but it is not (exits != exists). Let's take a look at next example:

```
package org.drools;
rule
when
    Object()
then
    System.out.println("A RHS");
end
```

Now the above code generates this message:

- [ERR 101] Line 3:2 no viable alternative at input 'WHEN'

This message means that the parser could identify the **WHEN** hard keyword, but it is not an option here. In this case, it is missing the rule name.

No viable alternative error occurs also when you have lexical problems. Here is a sample of a lexical problem:

```
rule simple_rule
when
    Student( name == "Andy )
then
end
```

The above code misses to close the quotes and because of this the parser generates this error message:

- [ERR 101] Line 0:-1 no viable alternative at input '<eof>' in rule simple_rule in pattern Student

Note

Usually the Line and Column information are accurate, but in some cases (like unclosed quotes), the parser generates a 0:-1 position. In this case you should check whether you didn't forget to close quotes, apostrophes or parentheses.

4.4.2.2. 102: Mismatched input

Indicates that the parser was looking for a particular symbol that it didn’t find at the current input position. Here are some samples:

```
rule simple_rule
when
    foo3 : Bar(
```

The above example generates this message:

- [ERR 102] Line 0:-1 mismatched input '<eof>' expecting ')' in rule simple_rule in pattern Bar

To fix this problem, it is necessary to complete the rule statement.

Note

Usually when you get a 0:-1 position, it means that parser reached the end of source.

The following code generates more than one error message:

```
package org.drools;

rule "Avoid NPE on wrong syntax"
when
    not(Cheese((type=="stilton",price==10)||(type=="brie",price==15))
        from $cheeseList)
then
    System.out.println("OK");
end
```

These are the errors associated with this source:

- [ERR 102] Line 5:36 mismatched input ',' expecting ')' in rule "Avoid NPE on wrong syntax" in pattern Cheese
- [ERR 101] Line 5:57 no viable alternative at input 'type' in rule "Avoid NPE on wrong syntax"
- [ERR 102] Line 5:106 mismatched input ')' expecting 'then' in rule "Avoid NPE on wrong syntax"

Note that the second problem is related to the first. To fix it, just replace the commas (",") by AND operator ("&&").

Note

In some situations you can get more than one error message. Try to fix one by one, starting at the first one. Some error messages are generated merely as consequences of other errors.

4.4.2.3. 103: Failed predicate

A validating semantic predicate evaluated to false. Usually these semantic predicates are used to identify soft keywords. This sample shows exactly this situation:

```
package nesting;
```

```
dialect "mvel"

import org.drools.Person
import org.drools.Address

fdsfdsfds

rule "test something"
when
    p: Person( name=="Michael" )
then
    p.name = "other";
    System.out.println(p.name);
end
```

With this sample, we get this error message:

- [ERR 103] Line 7:0 rule 'rule_key' failed predicate: {(validateIdentifierKey(DroolsSoftKeywords.RULE))}? in rule

The **fdsfdsfds** text is invalid and the parser couldn't identify it as the soft keyword "rule".

Note

This error is very similar to 102: Mismatched input, but usually involves soft keywords.

4.4.2.4. 104: Trailing semi-colon not allowed

This error is associated with the "eval" clause, where its expression may not be terminated with a semi-colon. Check this example:

```
rule simple_rule
when
    eval(abc();)
then
end
```

Due to the trailing semi-colon within eval, we get this error message:

- [ERR 104] Line 3:4 trailing semi-colon not allowed in rule simple_rule

This problem is simple to fix: just remove the semi-colon.

4.4.2.5. 105: Early Exit

The recognizer came to a (..)+ EBNF subrule that must match an alternative at least once, but the subrule did not match anything. Simply put: the parser has entered a branch from where there is no way out. This example tries to illustrates it:

```
template test_error
```

```
    aa s  11;
end
```

This is the message associated to the above sample:

- [ERR 105] Line 2:2 required (...)+ loop did not match anything at input 'aa' in template test_error

To fix this problem it is necessary to remove the numeric value as it is neither a valid data type which might begin a new template slot nor a possible start for any other rule file construct.

4.4.3. Other Messages

If you get any other message, means that something bad happened, so please contact the development team.

4.5. Package

A package is a collection of rules and other related constructs, such as imports and globals. The package members are typically related to each other - perhaps HR rules, for instance. A package represents a namespace, which ideally is kept unique for a given grouping of rules. The package name itself is the namespace, and is not related to files or folders in any way.

It is possible to assemble rules from multiple rule sources, and have one top level package configuration that all the rules are kept under (when the rules are assembled). Although, it is not possible to merge into the same package resources declared under different names. A single Rulebase, can though, contain multiple packages built on it. A common structure, is to have all the rules for a package in the same file as the package declaration (so that is it entirely self contained).

The following railroad diagram shows all the components that may make up a package. Note that a package *must* have a namespace and be declared using standard Java conventions for package names; i.e., no spaces, unlike rule names which allow spaces. In terms of the order of elements, they can appear in any order in the rule file, with the exception of the "package" and "expander" statements being at the top of the file, before any rules appear. In all cases, the semicolons are optional.

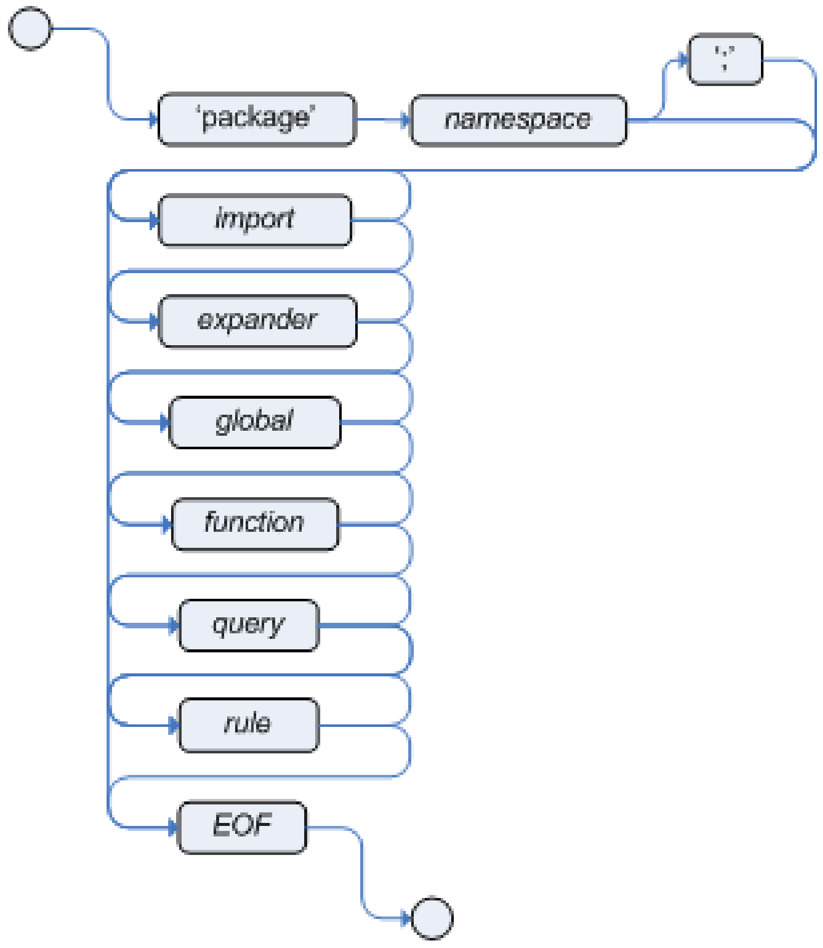


Figure 4.4. package

4.5.1. import

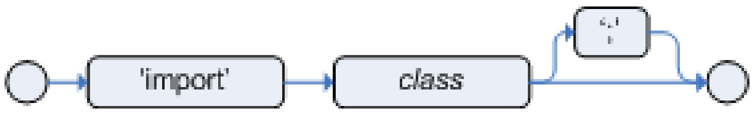


Figure 4.5. import

Import statements work like import statements in Java. You need to specify the fully qualified paths and type names for any objects you want to use in the rules. JBoss Rules automatically imports classes from the same named Java package and from the java.lang package.

4.5.2. expander

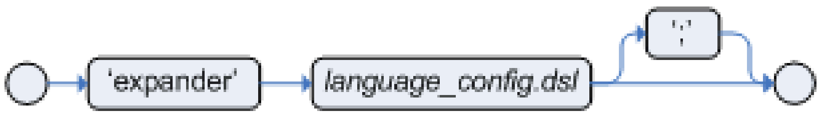


Figure 4.6. expander

The expander statement (optional) is used to specify domain specific language (DSL) configurations (which are normally stored in a separate file). This provides clues to the parser how to understand what you are raving on about in your rules. It is important to note that in JBoss Rules 5 the expander declaration is mandatory for the tools to provide you context assistance and to avoid error reporting, but the API allows the program to apply DSL templates, even if the expanders are not declared in the source file.

4.5.3. global

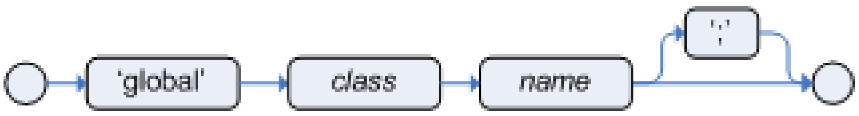


Figure 4.7. global

Globals are global variables. They are used to make application objects available to the rules, and are typically used to provide data or services that the rules use (specially application services used in rule consequences), to return data from the rules (like logs or values added in rules consequence) or for the rules to interact with the application doing callbacks. Globals are not inserted into the Working Memory so they should never be reasoned over. You should use them in rules' LHS only if the global has a constant immutable value. The engine cannot be notified about value changes of globals, and does not track their changes. Incorrect use of globals in constraints may yield unexpected results.

If multiple packages declare globals with the same identifier they must be of the same type and all of them will reference the same global value.

In order to use globals you must:

1. Declare your global variable in your rules file and use it in rules. Example:

```
global java.util.List myGlobalList;

rule "Using a global"
when
    eval( true )
then
    myGlobalList.add( "Hello World" );
end
```

2. Set the global value on your working memory. It is a best practice to set all global values before asserting any fact to the working memory. Example:

```
List list = new ArrayList();
WorkingMemory wm = rulebase.newStatefulSession();
wm.setGlobal( "myGlobalList", list );
```

Note that these are just named instances of objects that you pass in from your application to the working memory. This means you can pass in any object you want: you could pass in a service locator, or perhaps a service itself. With the new 'from' element it is now common to pass a Hibernate session as a global, to allow 'from' to pull data from a named Hibernate query.

One example may be an instance of a Email service. In your integration code that is calling the rule engine, you get your emailService object, and then set it in the working memory. In the DRL, you declare that you have a global of type EmailService, and give it a name "email". Then in your rule consequences, you can use things like email.sendSMS(number, message).

Globals are not designed to share data between rules and they should never be used for that purpose. Rules always reason and react to the working memory state, so if you want to "share" data between rules, assert the data to the working memory.

It is strongly discouraged to set (or change) a global value from inside your rules. We recommend to you always set the value from your application using the working memory interface.

4.6. Function

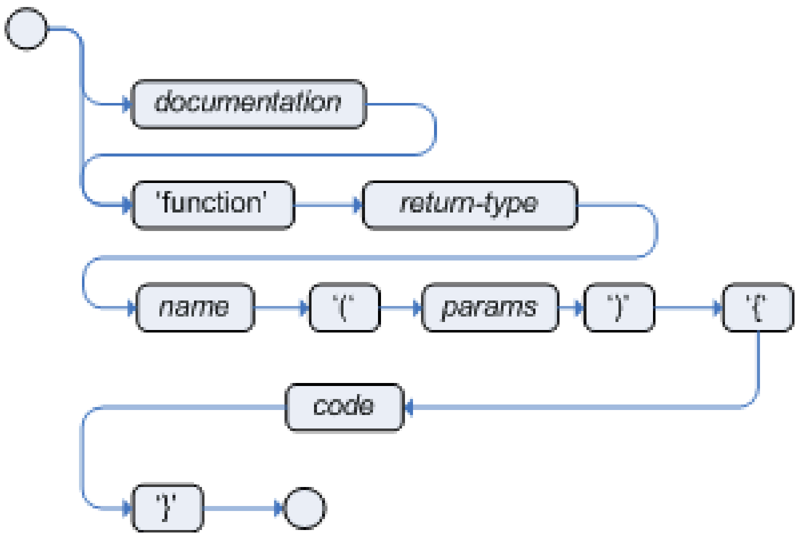


Figure 4.8. function

Functions are a way to put semantic code in your rule source file, as opposed to in normal Java classes. They can't do anything more then what you can do with helper classes (in fact, the compiler generates the helper class for you behind the scenes). The main advantage of using functions in a rule is that you can keep the logic all in one place, and you can change the functions as needed (this can be a good and bad thing). Functions are most useful for invoking actions on the consequence ("then") part of a rule, especially if that particular action is used over and over (perhaps with only differing parameters for each rule - for example the contents of an email message).

A typical function declaration looks like:

```
function String hello(String name) {
    return "Hello "+name+"!";
}
```

Note that the "function" keyword is used, even though its not really part of Java. Parameters to the function are just like a normal method (and you don't have to have parameters if they are not needed). Return type is just like a normal method.

An alternative to the use of a function, could be to use a static method in a helper class: Foo.hello(). JBoss Rules supports the use of function imports, so all you would need to do is:

```
import function my.package.Foo.hello
```

In both cases above, to use the function, just call it by its name in the consequence or inside a semantic code block. Example:

```
rule "using a static function"
when
    eval( true )
then
    System.out.println( hello( "Bob" ) );
end
```

4.7. Type Declaration

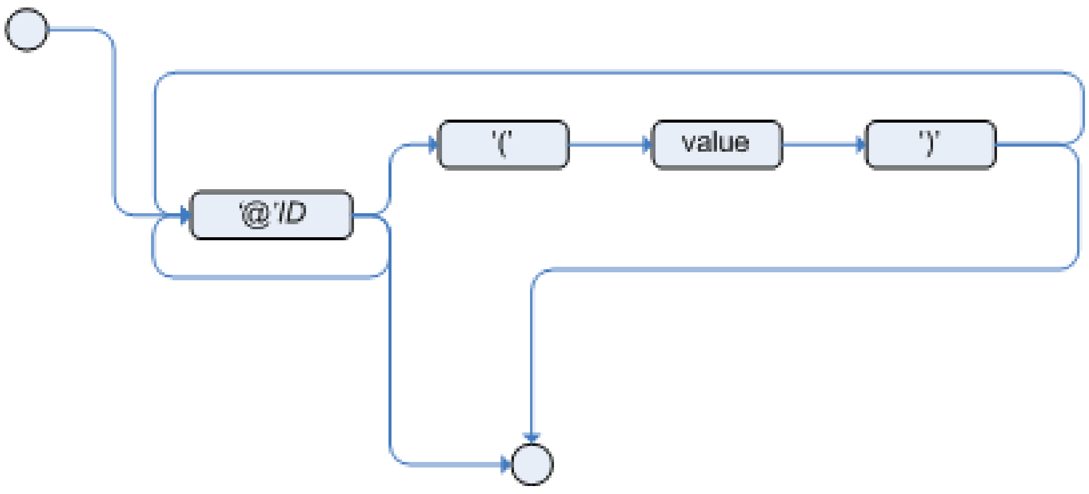


Figure 4.9. meta_data

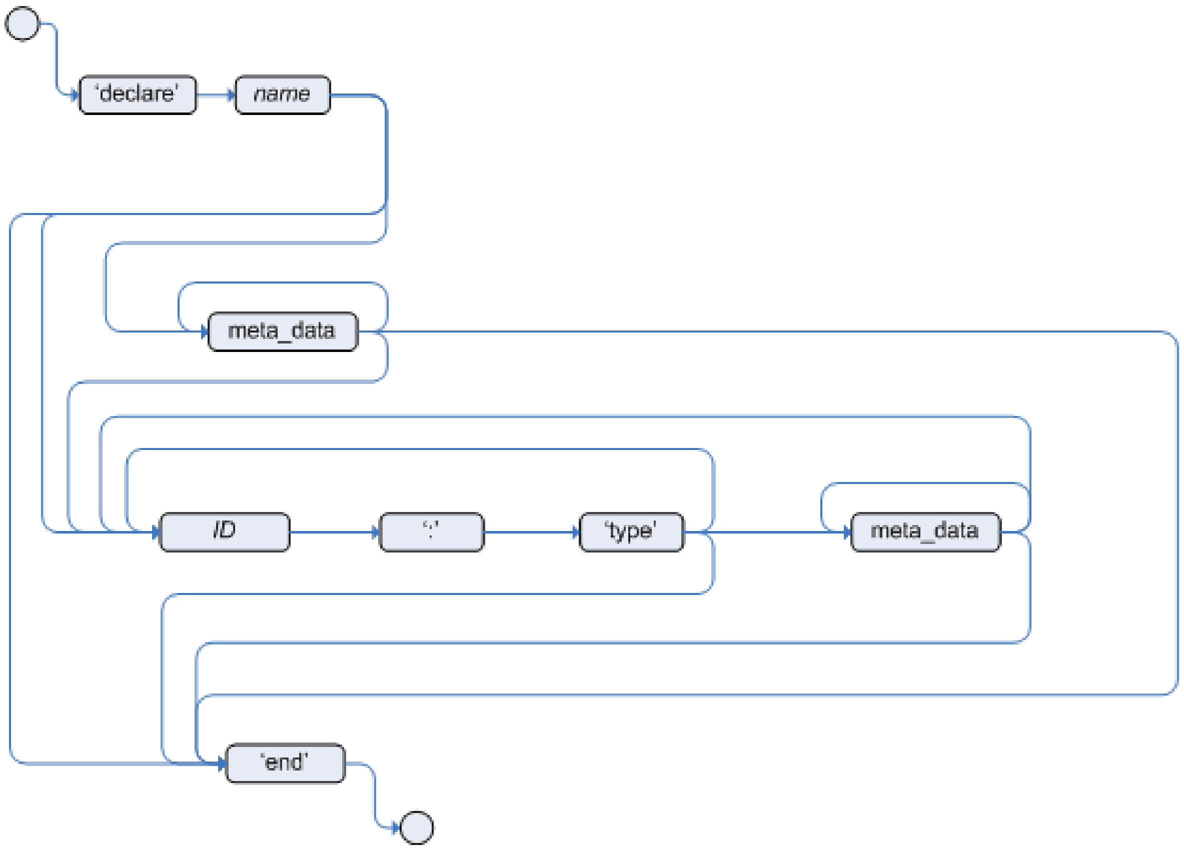


Figure 4.10. type_declaration

Type declarations have two main goals in the rules engine: to allow the declaration of new types, and to allow the declaration of metadata for types.

- **Declaring new types:** JBoss Rules works out of the box with plain POJOs as facts. Although, sometimes the users may want to define the model directly to the rules engine, without worrying to create their models in a lower level language like Java. At other times, there is a domain model already built, but eventually the user wants or needs to complement this model with additional entities that are used mainly during the reasoning process.
- **Declaring metadata:** facts may have meta information associated to them. Examples of meta information include any kind of data that is not represented by the fact attributes and are consistent among all instances of that fact type. This meta information may be queried at runtime by the engine and used in the reasoning process.

4.7.1. Declaring New Types

To declare a new type, all you need to do is use the keyword **declare**, followed by the list of fields and the keyword **end**.

```
declare Address
   number : int
   streetName : String
   city : String
end
```

Example 4.2. declaring a new fact type: Address

The previous example declares a new fact type called *Address*. This fact type will have 3 attributes: *number*, *streetName* and *city*. Each attribute has a type that can be any valid Java type, including any other class created by the user or even other fact types previously declared.

For instance, we may want to declare another fact type *Person*:

```

declare Person
    name : String
    dateOfBirth : java.util.Date
    address : Address
end
```

Example 4.3. declaring a new fact type: Person

As we can see on the previous example, *dateOfBirth* is of type `java.util.Date`, from the Java API, while *address* is of the previously defined fact type Address.

You may avoid having to write the fully qualified name of a class every time you write it by using the **import** clause, previously discussed.

```

import java.util.Date

declare Person
    name : String
    dateOfBirth : Date
    address : Address
end
```

Example 4.4. avoiding the need to use fully qualified class names by using import

When you declare a new fact type, JBoss Rules will, at compile time, generate bytecode implementing a POJO that represents the fact type. The generated Java class will be a one-to-one Java Bean mapping of the type definition. So, for the previous example, the generated Java class would be:

```
public class Person implements Serializable {
    private String name;
    private java.util.Date dateOfBirth;
    private Address address;

    // getters and setters
    // equals/hashCode
    // toString
}
```

Example 4.5. generated Java class for the previous Person fact type declaration

Since it is a simple POJO, the generated class can be used transparently in the rules, like any other fact.

```
rule "Using a declared Type"
when
    $p : Person( name == "Bob" )
then
    System.out.println( "The name of the person is "+ )
    // lets insert Mark, that is Bob's mate
    Person mark = new Person();
    mark.setName("Mark");
    insert( mark );
end
```

Example 4.6. using the declared types in rules

4.7.2. Declaring Metadata

Metadata may be assigned to several different constructions in JBoss Rules, like fact types, fact attributes and rules. JBoss Rules uses the @ symbol to introduce metadata, and it always uses the form:

```
@matadata_key( metadata_value )
```

The parenthesis and the metadata_value are optional.

For instance, if you want to declare a metadata attribute like *author*, whose value is *Bob*, you could simply write:

```
@author( Bob )
```

Example 4.7. declaring an arbitrary metadata attribute

JBoss Rules allows the declaration of any arbitrary metadata attribute, but some will have special meaning to the engine, while others are simply available for querying at runtime. JBoss Rules allows the declaration of metadata both for fact types and for fact attributes. Any metadata that is declared before the fields of a fact type are assigned to the fact type, while metadata declared after an attribute are assigned to the attribute in particular.

```
import java.util.Date

declare Person
    @author( Bob )
    @dateOfCreation( 01-Feb-2009 )

    name : String @key @maxLength( 30 )
    dateOfBirth : Date
    address : Address
end
```

Example 4.8. declaring metadata attributes for fact types and attributes

In the previous example, there are two metadata declared for the fact type (*@author* and *@dateOfCreation*), and two more defined for the name attribute (*@key* and *@maxLength*). Please note that the *@key* metadata has no value, and so the parenthesis and the value were omitted.

4.7.3. Declaring Metadata for Existing Types

JBoss Rules allows the declaration of metadata attributes for existing types in the same way as when declaring metadata attributes for new fact types. The only difference is that there are no fields in that declaration.

For instance, if there is a class org.drools.examples.Person, and one wants to declare metadata for it, just write the following code:

```
import org.drools.examples.Person

declare Person
    @author( Bob )
    @dateOfCreation( 01-Feb-2009 )
end
```

Example 4.9. declaring metadata for an existing type

Instead of using the import, it is also possible to reference the class by its fully qualified name, but since the class will also be referenced in the rules, usually it is shorter to add the import and use the short class name everywhere.

```
declare org.drools.examples.Person
    @author( Bob )
    @dateOfCreation( 01-Feb-2009 )
end
```

Example 4.10. declaring metadata using the fully qualified class name

4.7.4. Accessing Declared Types from the Application Code

Declared types are usually used inside rules files, while Java models are used when sharing the model between rules and applications. Although, sometimes, the application may need to access and handle facts from the declared types, specially when the application is wrapping the rules engine and providing higher level, domain specific, user interfaces for rules management.

In such cases, the generated classes can be handled as usual with the Java Reflection APIs, but as we know, that usually requires a lot of work for small results. This way, JBoss Rules provides a simplified API for the most common fact handling the application may want to do.

The first important thing to realize is that a declared fact will belong to the package where it was declared. So, for instance, in the example below, *Person* will belong to the *org.drools.examples* package, and so the generated class fully qualified name will be: *org.drools.examples.Person*.

```
package org.drools.examples

import java.util.Date

declare Person
    name : String
    dateOfBirth : Date
    address : Address
end
```

Example 4.11. declaring a type in the org.drools.examples package

Declared types, as discussed previously, are generated at knowledge base compilation time, i.e., the application will only have access to them at application run time. As so, these classes are not available for direct reference from the application.

JBoss Rules then provides an interface through which the users can handle declared types from the application code: org.drools.definition.type.FactType. Through this interface, the user can instantiate, read and write fields in the declared fact types.

```
// get a reference to a knowledge base with a declared type:
KnowledgeBase kbase = ...

// get the declared FactType
FactType personType = kbase.getFactType( "org.drools.examples",
                                         "Person" );

// handle the type as necessary:
// create instances:
Object bob = personType.newInstance();

// set attributes values
personType.set( bob,
                "name",
                "Bob" );
personType.set( bob,
                "age",
                42 );

// insert fact into a session
StatefulKnowledgeSession ksession = ...
ksession.insert( bob );
ksession.fireAllRules();

// read attributes
String name = personType.get( bob, "name" );
int age = personType.get( bob, "age" );
```

Example 4.12. handling declared fact types through the API

The API also includes other helpful methods, like setting all the attributes at once, reading values from a Map, or read all attributes at once, populating a Map.

Although the API is similar to Java reflection it does not use reflection. It instead relies on much faster bytecode generated accessors.

4.8. Rule

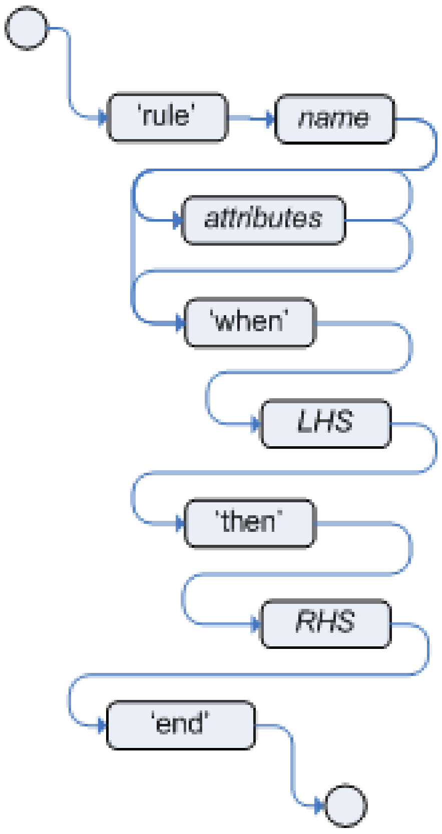


Figure 4.11. rule

A rule specifies that "when" a particular set of conditions occur, specified in the Left Hand Side (LHS), then do this, which is specified as a list of actions in the Right Hand Side (RHS). A common question from users is "why use when instead of if". "when" was chosen over "if" because "if" is normally part of a procedural execution flow, where at a specific point in time it checks the condition. Where as "when" indicates it's not tied to a specific evaluation sequence or point in time, at any time during the life time of the engine "when" this occurs, do that Rule.

A rule must have a name, unique within its rule package. If you define a rule twice in the same DRL it produces an error while loading. If you add a DRL that includes a rule name already in the package, it replaces the previous rule. If a rule name is to have spaces, then it will need to be enclosd in double quotes (it is best to always use double quotes).

Attributes are optional, and are described below (they are best kept as one per line).

The LHS of the rule follows the "when" keyword (ideally on a new line), similarly the RHS follows the "then" keyword (ideally on a newline). The rule is terminated by the keyword "end". Rules cannot be nested of course.

```
rule "<name>"
    <attribute>*
when
    <conditional element>*
then
    <action>*
end
```

Example 4.13. Rule Syntax Overview Example

```

rule "Approve if not rejected"
  salience -100
  agenda-group "approval"
    when
        not Rejection()
        p : Policy(approved == false, policyState:status)
        exists Driver(age > 25)
        Process(status == policyState)
    then
        log("APPROVED: due to no objections.");
        p.setApproved(true);
end
```

Example 4.14. A rule example

4.8.1. Rule Attributes

Rule attributes provide a declarative way to influence the behavior of the rule. Some are quite simple, while others are part of complex sub systems such as ruleflow. To get the most from JBoss Rules you should make sure you have a proper understanding of each attribute.

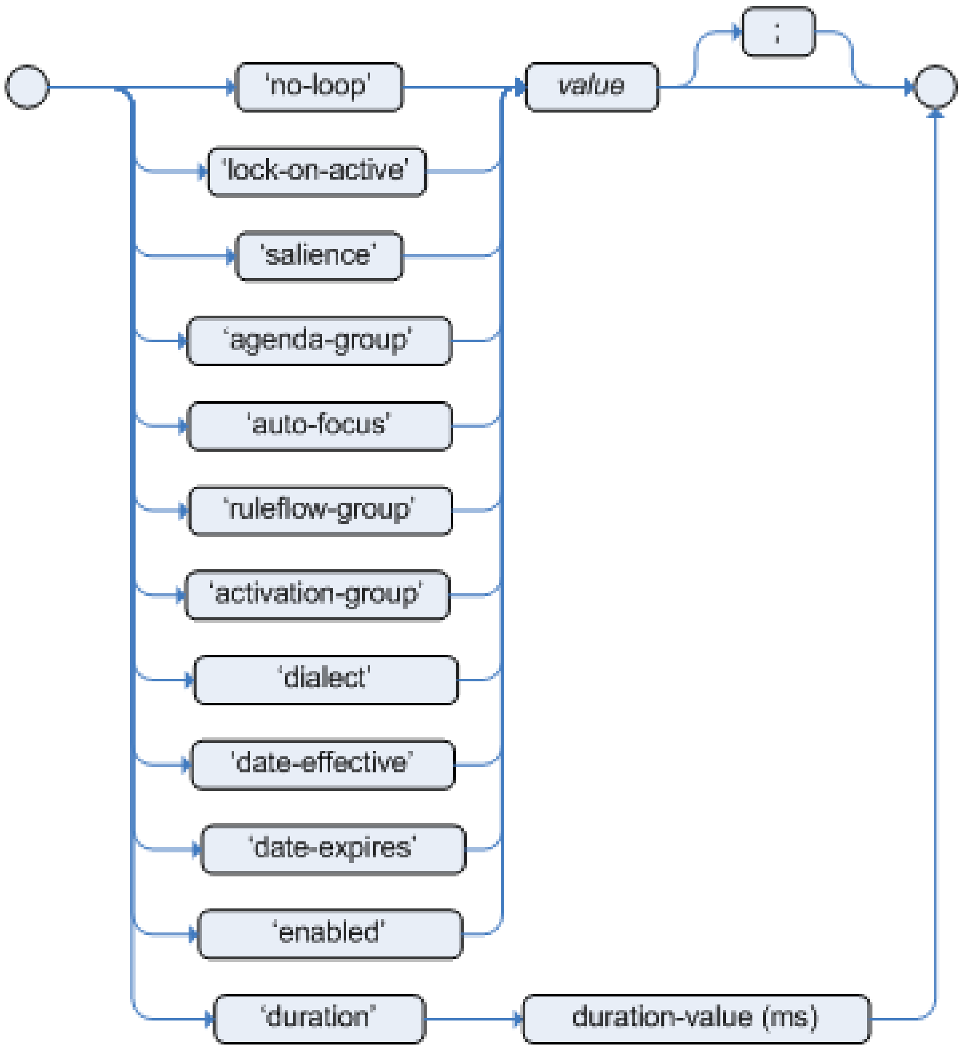


Figure 4.12. rule attributes

no-loop
: default value: false

  type: Boolean

  When the Rule's consequence modifies a fact it may cause the Rule to activate again, causing recursion. Setting no-loop to true means the attempt to create the Activation for the current set of data will be ignored.

lock-on-active
: default value: false

  type: Boolean

  Whenever a ruleflow-group becomes active or an agenda-group receives the focus, any rule within that group that has lock-on-active set to true will not be activated any more; irrespective of the origin of the update, the activation of a matching rule is discarded. This is a stronger version of no-loop, because the change could now be caused not only by the rule itself. It's ideal for calculation rules where you have a number of rules that modify a fact and you don't want any rule re-matching and firing again. Only when the ruleflow-group is no longer active or the agenda-group loses the focus those rules with lock-on-active set to true become eligible again for their activations to be placed onto the agenda.

salience
: default value : 0

  type : integer

  Each rule has a salience attribute that can be assigned an Integer number, defaults to zero, the Integer and can be negative or positive. Salience is a form of priority where rules with higher salience values are given higher priority when ordered in the Activation queue.

agenda-group
: default value: MAIN

  type: String

  Agenda group's allow the user to partition the Agenda providing more execution control. Only rules in the focus group are allowed to fire.

auto-focus
: default value: false

  type: Boolean

  When a rule is activated if the **`auto-focus value is true and the Rule's agenda-group`** does not have focus then it is given focus, allowing the rule to potentially fire.

activation-group
: default value: N/A

  type: String

  Rules that belong to the same named activation-group will only fire exclusively. In other words, the first rule in an activation-group to fire will cancel the other rules activations (stop them from firing). The Activation group attribute is any string, as long as the string is identical for all the rules you need to be in the one group.

  This used to be called Xor group, but technically its not quite an Xor, but you may hear people mention Xor group, just swap that term in your mind with activation-group.

dialect
: default value : as specified by the package

  type : String

  possible values: "java" or "mvel"

  The dialect species the language to be used for any code expressions in the LHS or the RHS code block. Currently two dialects are available, Java and MVEL. While the dialect can be specified at the package level, this attribute allows the package definition to be overridden.

date-effective
: default value: N/A

  type: String, which contains a date and time definition

A rule can only activate if the current date and time is after date-effective attribute.

date-expires
: default value: N/A

  type: String, which contains a date and time definition

  A rule cannot activate if the current date and time is after date-expires attribute.

duration
: default value: no default value

  type: long

  The duration dictates that the rule will fire after a specified duration, if it is still true.

```
rule "my rule"
    salience 42
    agenda-group "number 1"
    when ...
```

Example 4.15. Some attribute examples

4.8.2. Left Hand Side (when) Conditional Elements

The Left Hand Side (LHS) is a common name for the conditional part of the rule. It consists of zero or more Conditional Elements. If the LHS is left empty it is re-written as eval(true), which means the rule is always true, and will be activated when a new Working Memory session is created.

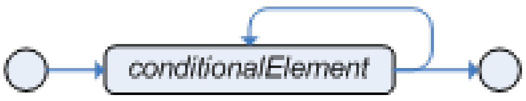


Figure 4.13. Left Hand Side

```
rule "no CEs"
when
then
        <action>*
end
```

Is internally re-written as:

```
rule "no CEs"
when
    eval( true )
then
    <action>*
end
```

Example 4.16. Rule without a Conditional Element

Conditional elements work on one or more *patterns* (which are described below). The most common one is "and" which is implicit when you have multiple patterns in the LHS of a rule that are not connected in any way. Note that an 'and' cannot have a leading declaration binding like 'or'. This is obvious, since a declaration can only reference a single fact, and when the 'and' is satisfied it matches more than one fact - which fact would the declaration bind to?

4.8.2.1. Pattern

The pattern element is the most important Conditional Element. The entity relationship diagram below provides an overview of the various parts that make up the pattern's constraints and how they work together; each is then covered in more detail with railroad diagrams and examples.

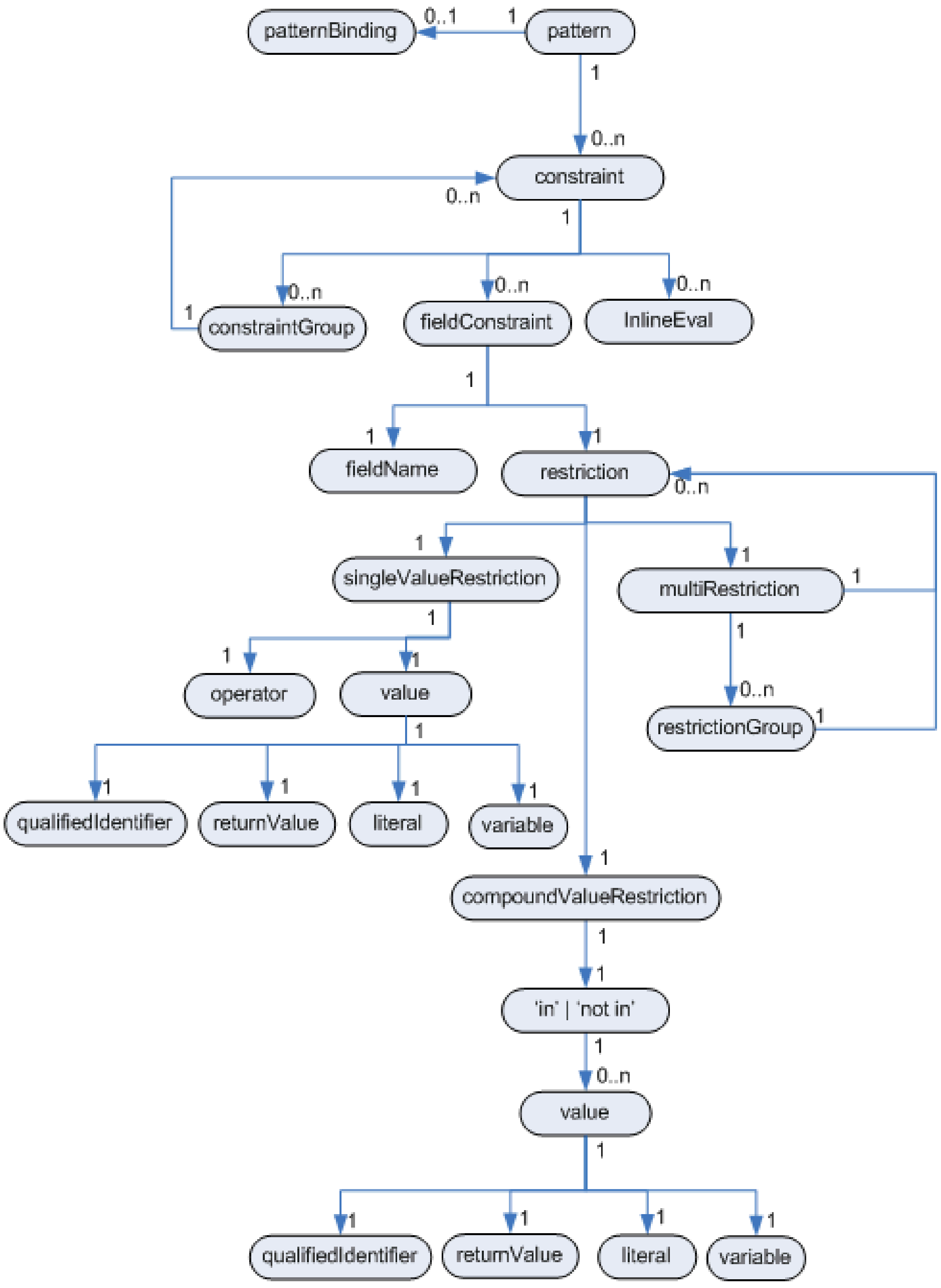


Figure 4.14. Pattern Entity Relationship Diagram

At the top of the ER diagram you can see that the pattern consists of zero or more constraints and has an optional pattern binding. The railroad diagram below shows the syntax for this.

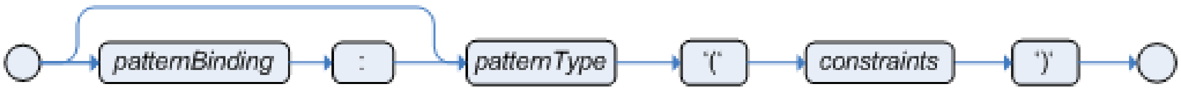


Figure 4.15. Pattern

In its simplest form, with no constraints, it simply matches against a type. In the following case the type is "Cheese", which means that the pattern will match against all Cheese objects in the Working Memory.

```
Cheese( )
```

Example 4.17. Simple Pattern

To be able to refer to the matched object use a pattern binding variable such as '$c'. While this example variable is prefixed with a $ symbol, it is optional, but can be useful in complex rules as it helps to more easily differentiation between variables and fields.

```
$c : Cheese( )
```

Example 4.18. Pattern with a binding variable

Inside of the pattern parenthesis is where all the action happens. A constraint can be either a Field Constraint, Inline Eval (called a predicate in 3.0) or a Constraint Group. Constraints can be separated by the following symbols: ',', '&&' or '||'.

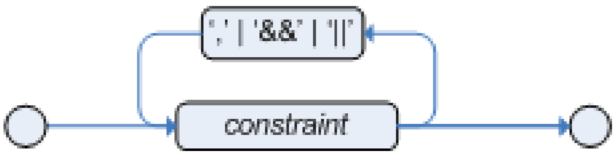


Figure 4.16. Constraints

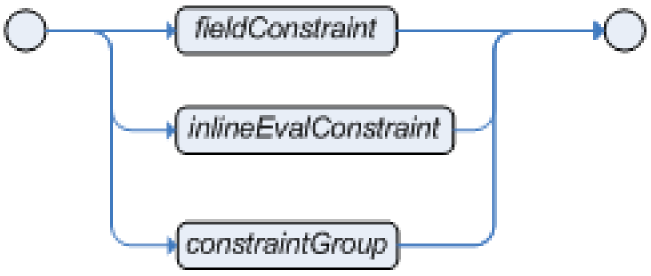


Figure 4.17. Constraint

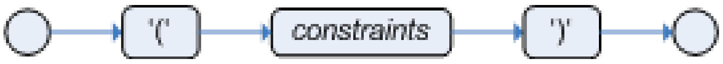


Figure 4.18. Group Constraint

The ',' (comma) character is used to separate constraint groups. It has implicit 'and' connective semantics.

```
# Cheese type is stilton and price < 10 and age is mature.
Cheese( type == "stilton", price < 10, age == "mature" )
```

Example 4.19. Constraint Group connective ',' The above example has 3 constraint groups, each with a single constraint:

- group 1: type is stilton -> type == "stilton"
- group 2: price is less than 10 -> price < 10
- group 3: age is mature -> age == "mature"

The '&&' (and) and '||' (or) constraint connectives allow constraint groups to have multiple constraints. Example:

```
Cheese( type == "stilton" && price < 10, age == "mature" ) // Cheese type
 is "stilton" and price < 10, and age is mature
Cheese( type == "stilton" || price < 10, age == "mature" ) // Cheese type
 is "stilton" or price < 10, and age is mature
```

Example 4.20. && and || Constraint Connectives

The above example has two constraint groups. The first has 2 constraints and the second has one constraint.

The connectives are evaluated in the following order, from first to last:

1. &&
2. ||
3. ,

It is possible to change the evaluation priority by using parenthesis, as in any logic or mathematical expression. Example:

```
# Cheese type is stilton and ( price is less than 20 or age is mature ).
Cheese( type == "stilton" && ( price < 20 || age == "mature" ) )
```

Example 4.21. Using parenthesis to change evaluation priority

In the above example, the use of parenthesis makes the || connective be evaluated before the && connective.

Also, it is important to note that besides having the same semantics, the connectives '&&' and ',' are resolved with different priorities and ',' cannot be embedded in a composite constraint expression.

```
Cheese( ( type == "stilton", price < 10 ) || age == "mature" ) // invalid
 as ',' cannot be embedded in an expression
Cheese( ( type == "stilton" && price < 10 ) || age == "mature") // valid as
 '&&' can be embedded in an expression
```

Example 4.22. Not Equivalent connectives

4.8.2.2. Field Constraints

A Field constraint specifies a restriction to be used on a field name; the field name can have an optional variable binding.

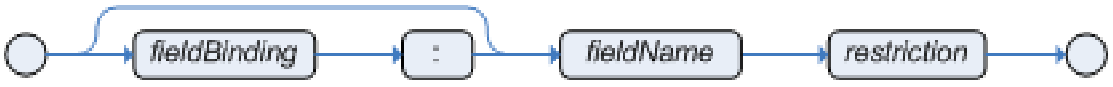


Figure 4.19. fieldConstraint

There are three types of restrictions; Single Value Restriction, Compound Value Restriction and Multi Restriction.

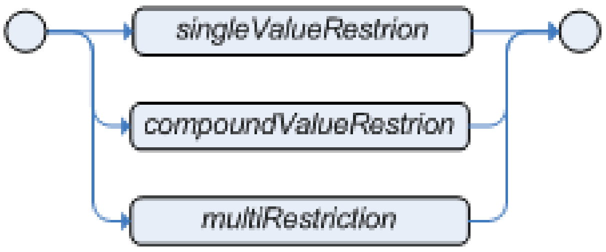


Figure 4.20. restriction

4.8.2.3. JavaBeans as facts

A field is an accessible method on the object. If your model objects follow the Java bean pattern, then fields are exposed using "getXXX" or "isXXX" methods (these are methods that take no arguments, and return something). You can access fields either by using the bean naming convention (so "getType" can be accessed as "type") - we use the standard Java **`Introspector`** class to do this mapping.

For example, referring to our Cheese class, the pattern Cheese(type == ...) applies the getType() method to a Cheese instance. If a field name cannot be found it will resort to calling the name as a no argument method; "toString()" on the Object for instance can be used with Cheese(toString == ..) - you use the full name of the method with correct capitalization, but without parentheses. Do please make sure that you are accessing methods that take no parameters, and are in-fact "accessors" which don't change the state of the object in a way that may effect the rules. Remember that the rule engine effectively caches the results of its matching in between invocations to make it faster.

4.8.2.4. Values

The field constraints can take a number of values; including literal, qualifiedIdentifier (enum), variable and returnValue.

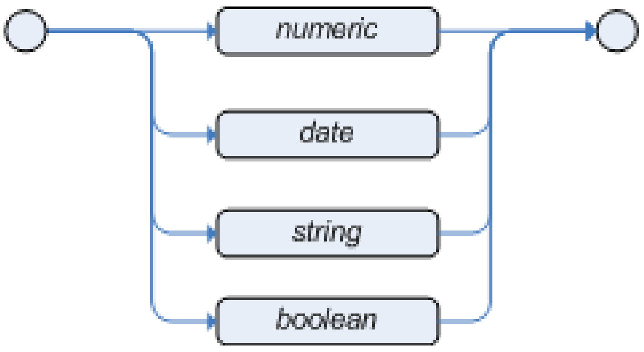


Figure 4.21. literal

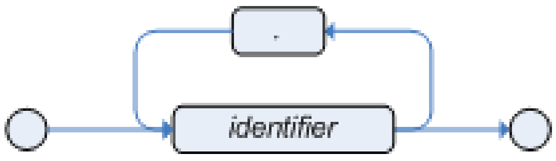


Figure 4.22. qualifiedIdentifier

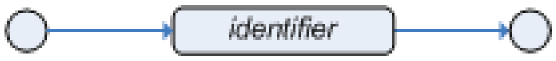


Figure 4.23. variable

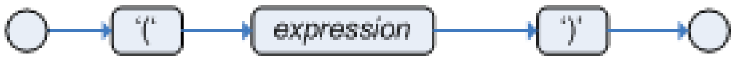


Figure 4.24. returnValue

You can do checks against fields that are or may be null, using == and != as you would expect, and the literal "null" keyword, like: Cheese(type != null). If a field is null the evaluator will not throw an exception and will only return true if the value is null. Coercion is always attempted if the field and the value are of different types; exceptions will be thrown if bad coercions are attempted. For instance, if "ten" is provided as a string in a number evaluator an exception is thrown, whereas "10" would coerce to a numeric 10. Coercion is always in favor of the field type and not the value type.

4.8.2.5. Single Value Restriction

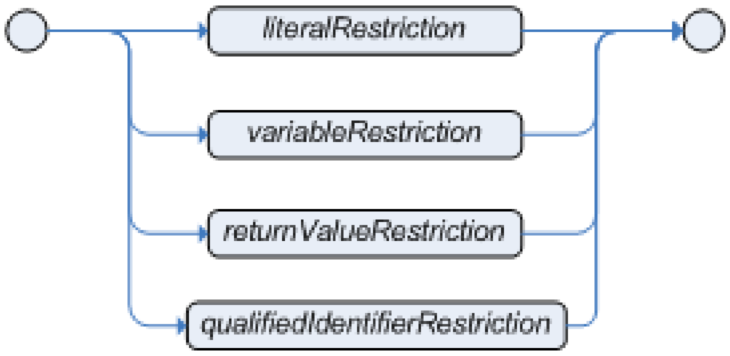


Figure 4.25. singleValueRestriction

4.8.2.6. Operators

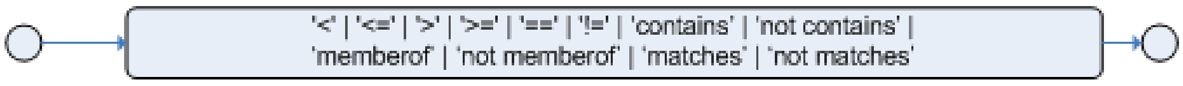

'<' | '<=' | '>' | '>=' | '==' | '!=' | 'contains' | 'not contains' |
'memberof' | 'not memberof' | 'matches' | 'not matches'

Figure 4.26. Operators

Valid operators are dependent on the field type. Generally they are self explanatory based on the type of data: for instance, for date fields, "<" means "before". "Matches" is only applicable to string fields, "contains" and "not contains" is only applicable to Collection type fields. These operators can be used with any value and coercion to the correct value for the evaluator and field will be attempted, as mentioned in the "Values" section.

Matches Operator

Matches a field against any valid Java Regular Expression. Typically that regexp is a String, but variables that resolve to a valid regexp are also allowed. It is important to note that, *different from Java*, if you write a String regexp directly on the source file, *you don't need to escape* '\'. Example:

```
Cheese( type matches "(Buffalo)?\S*Mozerella" )
```

Example 4.23. Regular Expression Constraint

Not Matches Operator

Any valid Java regular expression Regular Expression can be used to match String fields. This operator returns true when the match is false. Typically that regexp is a String, but variables that resolve to a valid regexp are also allowed. It is important to note that, *different from Java*, if you write a String regexp directly on the source file, *you don't need to escape* '\'. Example:

```
Cheese( type not matches "(Buffulo)?\S*Mozerella" )
```

Example 4.24. Regular Expression Constraint

Contains Operator

`'contains'` is used to check if a field's Collection or array contains the specified value.

```
CheeseCounter( cheeses contains "stilton" ) // contains with a String
 literal
                    CheeseCounter( cheeses contains $var ) // contains with
 a variable
```

Example 4.25. Contains with Collections

not contains

`'not contains'` is used to check if a field's Collection or array does not contain an object.

```
CheeseCounter( cheeses not contains "cheddar" ) // not contains with a
 String literal
                    CheeseCounter( cheeses not contains $var ) // not
 contains with a variable
```

Example 4.26. Literal Constraints with Collections

Note

For backward compatibility, the 'excludes' operator is supported as a synonym for 'not contains'.

memberOf

`'memberOf'` is used to check if a field is a member of a collection or array; that collection must be be a variable.

```
CheeseCounter( cheese memberOf $matureCheeses )
```

Example 4.27. Literal Constraints with Collections

not memberOf

'not memberOf' is used to check if a field is not a member of a collection or array; that collection must be be a variable.

```
CheeseCounter( cheese not memberOf $matureCheeses )
```

Example 4.28. Literal Constraints with Collections

soundslike

Similar to **'matches'**, but checks if a word has almost the same sound as the given value. Uses the 'Soundex' algorithm (**http://en.wikipedia.org/wiki/Soundex**).

```
Cheese( name soundslike 'foobar' )
```

Example 4.29. Text with soundslike (Sounds Like)

This will match a cheese with a name of "fubar"

4.8.2.7. Literal Restrictions

Literal restrictions are the simplest form of restrictions and evaluate a field against a specified literal; numeric, date, string or boolean.

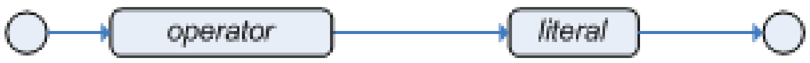


Figure 4.27. literalRestriction

Literal Restrictions using the '==' operator, provide for faster execution as we can index using hashing to improve performance.

Numeric

All standard Java numeric primitives are supported.

```
Cheese( quantity == 5 )
```

Example 4.30. Numeric Literal Restriction

Date

The date format "dd-mmm-yyyy" is supported by default. You can customize this by providing an alternative date format mask as a System property ("drools.dateformat" is the name of the property). If more control is required, use the inline-eval constraint.

```
Cheese( bestBefore < "27-Oct-2007" )
```

Example 4.31. Date Literal Restriction

String

Any valid Java String is allowed.

```
Cheese( type == "stilton" )
```

Example 4.32. String Literal Restriction

Boolean

Only true or false can be used; 0 and 1 are not recognized. A Boolean field alone (as in `Cheese( smelly )` is not permitted.

```
Cheese( smelly == true )
```

Example 4.33. Boolean Literal Restriction

Qualified Identifier

Enums can be used as well, both Java 1.4 and Java 5 style enumerations are supported. For the latter you must be executing on a Java 5 environment.

```
Cheese( smelly == SomeClass.TRUE )
```

Example 4.34. Boolean Literal Restriction

4.8.2.8. Bound Variable Restriction

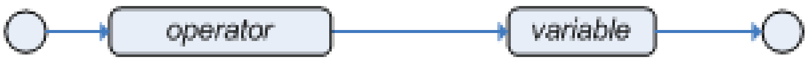


Figure 4.28. variableRestriction

Variables can be bound to Facts and their Fields and then used in subsequent Field Constraints. A bound variable is called a Declaration. Valid operators are determined by the type of the field being constrained; coercion will be attempted where possible. Bound Variable Restrictions using '==' operator, provide for very fast execution as we can index using hashing to improve performance.

```
Person( likes : favouriteCheese )
Cheese( type == likes )
```

Example 4.35. Bound Field using '==' operator

'likes' is our variable, our Declaration, that is bound to the favouriteCheese field for any matching Person instance and is used to constrain the type of Cheese in the following pattern. Any valid Java variable name can be used, and it may be prefixed with a '$', which you will often see used to help differentiate declarations from fields. The example below shows a declaration bound to the pattern's Object Type instance itself and used with a 'contains' operator. Note the optional use of '$'.

```
$stilton : Cheese( type == "stilton" )
Cheesery( cheeses contains $stilton )
```

Example 4.36. Bound Fact using 'contains' operator

4.8.2.9. Return Value Restriction

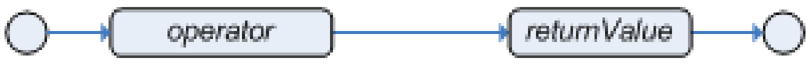


Figure 4.29. returnValueRestriction

A Return Value restriction can use any valid Java primitive or object. Avoid using any JBoss Rules keywords as Declaration identifiers. Functions used in a Return Value Restriction must return time constant results. Previously bound declarations can be used in the expression.

```
Person( girlAge : age, sex == "F" )
Person( age == ( girlAge + 2) ), sex == 'M' )
```

Example 4.37. Return Value Restriction

4.8.2.10. Compound Value Restriction

The compound value restriction is used where there is more than one possible value, currently only the 'in' and 'not in' evaluators support this. The operator takes a parenthesis enclosed comma separated list of values, which can be a variable, literal, return value or qualified identifier. The 'in' and 'not in' evaluators are actually sugar and are rewritten as a multi restriction list of != and == restrictions.

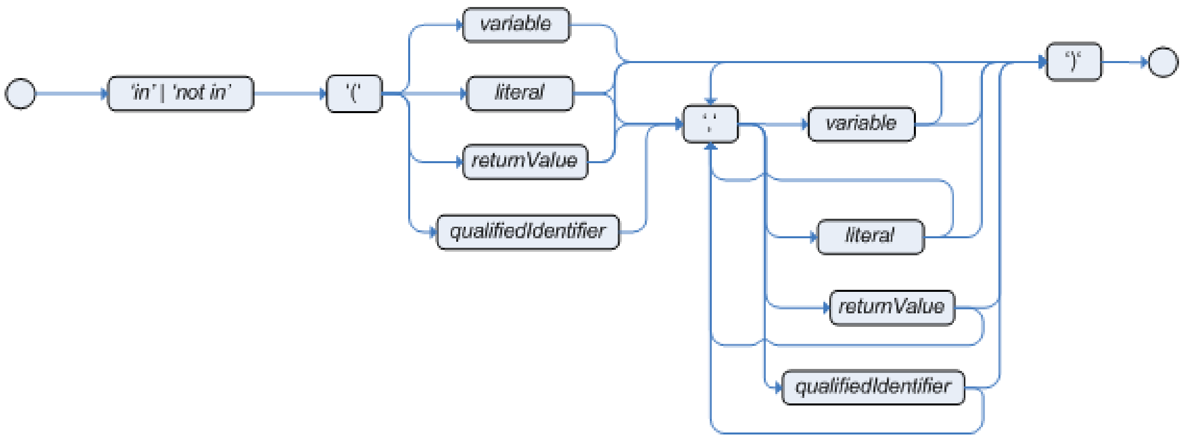


Figure 4.30. compoundValueRestriction

```
Person( $cheese : favouriteCheese )
Cheese( type in ( "stilton", "cheddar", $cheese )
```

Example 4.38. Compound Restriction using 'in'

4.8.2.11. Multi Restriction

Multi restriction allows you to place more than one restriction on a field using the '&&' or '||' restriction connectives. Grouping via parenthesis is also allowed; which adds a recursive nature to this restriction.

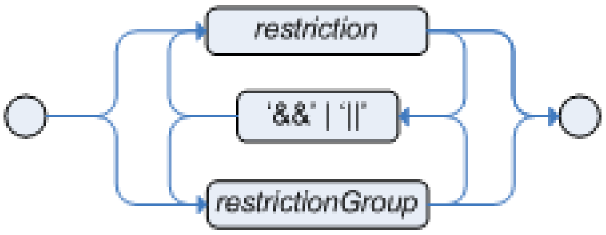


Figure 4.31. multiRestriction

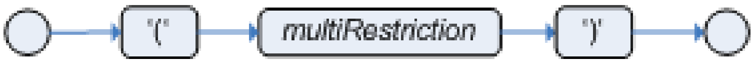


Figure 4.32. restrictionGroup

```
// simple multi restriction using a single &&
Person( age > 30 && < 40 )
// more complex multi restriction using groupings of multi restrictions
Person( age ( (> 30 && < 40) || (> 20 && < 25) ) )
// mixing multi restrictions with constraint connectives
Person( age > 30 && < 40 || location == "london" )
```

Example 4.39. Multi Restriction

4.8.2.12. Inline Eval Constraints

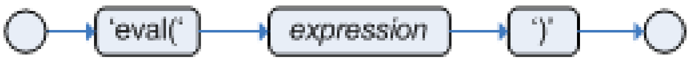


Figure 4.33. Inline Eval Expression

An inline-eval constraint can use any valid dialect expression as long as it is evaluated to a primitive boolean - avoid using any JBoss Rules keywords as Declaration identifiers. The expression must be time constant. Any previously bound variable, from the current or previous pattern, can be used; auto-vivification is also used to auto-create field binding variables. When an identifier is found that is not a current variable the builder looks to see if the identifier is a field on the current object type, if it is, the field binding is auto-created as a variable of the same name. This is called auto-vivification of field variables inside of inline evals.

This example will find all male-female pairs where the male is 2 years older than the female; the boyAge variable is auto-created by the auto-vivification process.

```
Person( girlAge : age, sex = "F" )
Person( eval( age == girlAge + 2 ), sex = 'M' )
```

Example 4.40. Return Value operator

4.8.2.13. Nested Accessors

JBoss Rules does allow for nested accessors in in the field constraints using the MVEL accessor graph notation. Field constraints involving nested accessors are actually re-written as an MVEL dialect inline-eval. Care should be taken when using nested accessors as the Working Memory is not aware of any of the nested values, and do not know when they change; they should be considered immutable while any of their parent references are inserted into the Working Memory. If you wish to modify a nested value you should remove the parent objects first and re-assert afterwards. If you only have a single parent at the root of the graph, when in the MVEL dialect, you can use the 'modify' keyword and its block setters to write the nested accessor assignments while retracting and inserting the the root parent object as required. Nested accessors can be used either side of the operator symbol.

```
$p : Person( )
// Find a pet who is older than their owners first born child
Pet( owner == $p, age > $p.children[0].age )
```

is internally rewriten as an MVEL inline eval:

```
$p : Person( )
// Find a pet who is older than their owners first born child
Pet( owner == $p, eval( age > $p.children[0].age ) )
```

Example 4.41. Nested Accessors

Note

Nested accessors have a much greater performance cost than direct field access, so use them carefully.

4.8.2.14. 'and'

The 'and' Conditional Element is used to group together other Conditional Elements. The root element of the LHS is an implicit prefix And and doesn't need to be specified. JBoss Rules supports both prefix and infix; although prefix is the preferred option as grouping is implicit which avoids confusion.

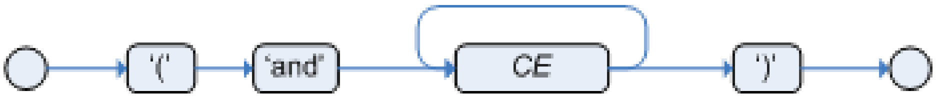


Figure 4.34. prefixAnd

```
(and Cheese( cheeseType : type )
Person( favouriteCheese == cheeseType ) )
```

Example 4.42. prefixAnd

```
when
Cheese( cheeseType : type )
Person( favouriteCheese == cheeseType )
```

Example 4.43. implicit root prefixAnd

Infix 'and' is supported along with explicit grouping with parenthesis, should it be needed. The '&&' symbol, as an alternative to 'and', is deprecated although it is still supported in the syntax for legacy support reasons.

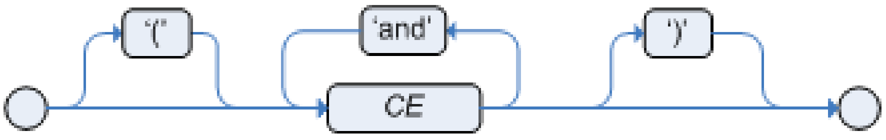


Figure 4.35. infixAnd

```
Cheese( cheeseType : type ) and
    Person( favouriteCheese == cheeseType ) //infixAnd
(Cheese( cheeseType : type ) and (Person( favouriteCheese == cheeseType )
 or
    Person( favouriteCheese == cheeseType  ) ) //infixAnd with grouping
```

Example 4.44. infixAnd

4.8.2.15. 'or'

The 'or' Conditional Element is used to group together other Conditional Elements. JBoss Rules supports both prefix and infix; although prefix is the preferred option as grouping is implicit which avoids confusion. The behavior of the 'or' Conditional Element is different than the '||' connective for constraints and restrictions in field constraints. The engine actually has no understanding of 'or' Conditional Elements, instead via a number of different logic transformations the rule is re-written as a number of sub-rules; the rule now has a single 'or' as the root node and a sub-rule per logical outcome. Each sub-rule can activate and fire like any normal rule, there is no special behavior or interactions between the sub-rules - this can be most confusing to new rule authors.

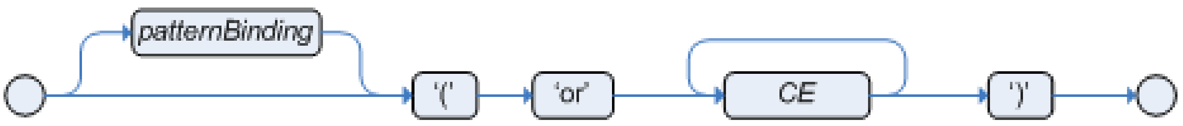


Figure 4.36. prefixOr

```
(or Person( sex == "f", age > 60 )
Person( sex == "m", age > 65 )
```

Example 4.45. prefixOr

Infix 'or' is supported along with explicit grouping with parenthesis, should it be needed. The '||' symbol, as an alternative to 'or', is deprecated although it is still supported in the syntax for legacy support reasons.

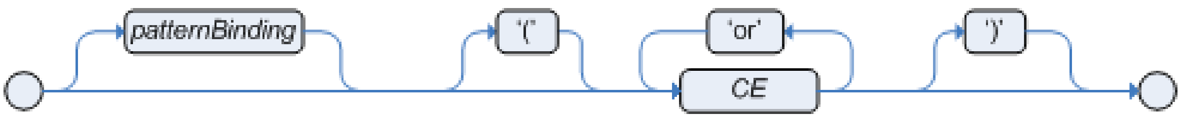


Figure 4.37. infixOr

```
Cheese( cheeseType : type ) or
    Person( favouriteCheese == cheeseType ) //infixOr
(Cheese( cheeseType : type ) or (Person( favouriteCheese == cheeseType )
 and
    Person( favouriteCheese == cheeseType  ) ) //infixOr with grouping
```

Example 4.46. infixOr

The 'or' Conditional Element also allows for optional pattern binding; which means each resulting subrule will bind its pattern to the pattern binding. Each pattern must be bound separately, using eponymous variables:

```
(or pensioner : Person( sex == "f", age > 60 )
pensioner : Person( sex == "m", age > 65 ) )
```

Example 4.47. or with binding

The 'or' conditional element results in multiple rule generation, called sub rules, for each possible logically outcome. The example above would result in the internal generation of two rules. These two rules work independently within the Working Memory, which means both can match, activate and fire - there is no shortcutting.

The best way to think of the OR conditional element is as a shortcut for generating two or more similar rules. When you think of it that way, it's clear that for a single rule there could be multiple activations if both sides of the OR conditional element are true.

4.8.2.16. 'eval'

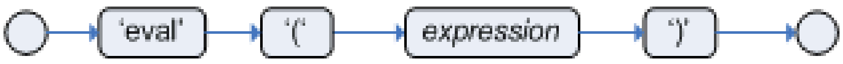


Figure 4.38. eval

Eval is essentially a catch all which allows any semantic code (that returns a primitive boolean) to be executed. This can refer to variables that were bound in the LHS of the rule, and functions in the rule package. Overuse of eval reduces the declarative nature of your rules and can result in a poor performing engine. While 'evals' can be used anywhere in the Pattern the best practice is to add it as the last conditional element in the LHS of a rule.

Evals cannot be indexed and thus are not as optimal as using Field Constraints. However this makes them ideal for being used when functions return values that change over time, which is not allowed within Field Constraints.

```
p1 : Parameter()
p2 : Parameter()
eval( p1.getList().containsKey(p2.getItem()) )
//this is how you call a function in the LHS - a function called "isValid"
eval( isValid(p1, p2) )
```

Example 4.48. eval

4.8.2.17. 'not'

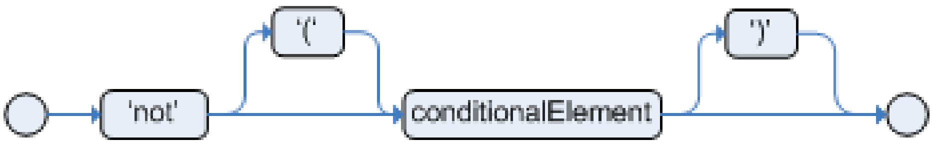


Figure 4.39. not

'not' is first order logic's Non-Existential Quantifier and checks for the non existence of something in the Working Memory. Think of 'not' as meaning "there must be none of...".

A 'not' statement must be followed by parentheses around the CEs that it applies to. In the simplest case of a single pattern (like below) you may optionally omit the parentheses.

```
not Bus()
```

Example 4.49. No Busses

```
not Bus(color == "red") //brackets are optional for this simple pattern
//brackets are optional for this simple case
not ( Bus(color == "red", number == 42) )
// not with nested 'and' infix used here
// as only two patterns (but brackets are required)
not ( Bus(color == "red") and Bus(color == "blue"))
```

Example 4.50. No red Busses

4.8.2.18. 'exists'

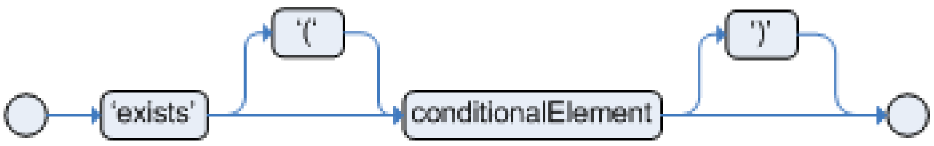


Figure 4.40. exists

'exists' is first order logic's Existential Quantifier and checks for the existence of something in the Working Memory. Think of exist as meaning "at least one..". It is different from just having the Pattern on its own; which is more like saying "for each one of...". If you use exist with a Pattern, then the rule will only activate at most once, regardless of how much data there is in working memory that matches that condition. Since only the existence matters, no bindings will be established.

An 'exist' statement must be followed by parentheses around the CEs that it applies to. In the simplest case of a single pattern (like below) you may optionally omit the parentheses.

```
exists Bus()
```

Example 4.51. At least one Bus

```
exists Bus(color == "red")
exists ( Bus(color == "red", number == 42) ) //brackets are optional
// exists with nested 'and' infix used here as only two patterns
exists ( Bus(color == "red") and Bus(color == "blue"))
```

Example 4.52. At least one red Bus

4.8.2.19. 'forall'

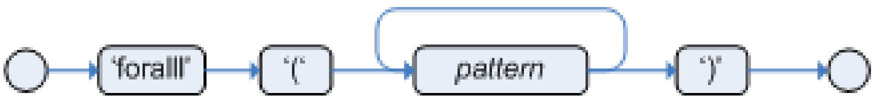


Figure 4.41. forall

The **forall** Conditional Element completes the First Order Logic support in JBoss Rules. The **forall** Conditional Element will evaluate to true when all facts that match the first pattern match all the remaining patterns. Example:

```
rule "All english buses are red"
when
    forall( $bus : Bus( type == 'english')
    Bus( this == $bus, color = 'red' ) )
then
    # all english buses are red
end
```

In the above rule, we "select" all Bus object whose type is "english". Then, for each fact that matches this pattern we evaluate the following patterns and if they match, the forall CE will evaluate to true.

To state that all facts of a given type in the working memory must match a set of constraints, forall can be written with a single pattern for simplicity. Example

```
rule "All Buses are Red"
when
    forall( Bus( color == 'red' ) )
then
    # all asserted Bus facts are red
end
```

Example 4.53. Single Pattern Forall

Another example of multi-pattern forall:

```
rule "all employees have health and dental care programs"
when
    forall( $emp : Employee()
        HealthCare( employee == $emp )
        DentalCare( employee == $emp )
    )
then
    # all employees have health and dental care
end
```

Example 4.54. Multi-Pattern Forall

Forall can be nested inside other CEs for complete expressiveness. For instance, **forall** can be used inside a **not** CE, note that only single patterns have optional parenthesis, so with a nested forall parenthesis must be used :

```
rule "not all employees have health and dental care"
when
    not ( forall( $emp : Employee()
        HealthCare( employee == $emp )
        DentalCare( employee == $emp ) )
    )
then
    # not all employees have health and dental care
end
```

Example 4.55. Combining Forall with Not CE

As a side note, `not( forall( p1 p2 p3...))` is equivalent to writing:

```
not(p1 and not(and p2 p3...))
```

Also, it is important to note that **forall is a scope delimiter**, so it can use any previously bound variable, but no variable bound inside it will be available to use outside of it.

4.8.2.20. From

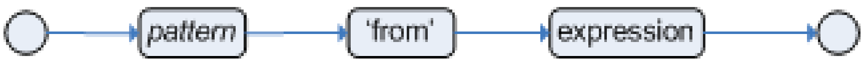


Figure 4.42. from

The **from** Conditional Element allows users to specify a source for patterns to reason over. This allows the engine to reason over data not in the Working Memory. This could be a sub-field on a bound variable or the results of a method call. It is a powerful construction that allows out of the box integration with other application components and frameworks. One common example is the integration with data retrieved on-demand from databases using hibernate named queries.

The expression used to define the object source is any expression that follows regular MVEL syntax. I.e., it allows you to easily use object property navigation, execute method calls and access maps and collections elements.

Here is a simple example of reasoning and binding on another pattern sub-field:

```
rule "validate zipcode"
when
    Person( $personAddress : address )
    Address( zipcode == "23920W") from $personAddress
then
    # zip code is ok
end
```

With all the flexibility from the new expressiveness in the JBoss Rules engine you can slice and dice this problem many ways. This is the same but shows how you can use a graph notation with the 'from':

```
rule "validate zipcode"
when
    $p : Person( )
    $a : Address( zipcode == "23920W") from $p.address
then
    # zip code is ok
end
```

Previous examples were reasoning over a single pattern. The **from** CE also support object sources that return a collection of objects. In that case, **from** will iterate over all objects in the collection and try to match each of them individually. For instance, if we want a rule that applies 10% discount to each item in an order, we could do:

```
rule "apply 10% discount to all items over US$ 100,00 in an order"
when
    $order : Order()
    $item  : OrderItem( value > 100 ) from $order.items
then
    # apply discount to $item
end
```

The above example will cause the rule to fire once for each item whose value is greater than 100 for each given order.

You must take caution, however, when using **from**, especially in conjunction with the **lock-on-active** rule attribute as it may produce unexpected results. Consider the example provided earlier, but now slightly modified as follows:

```
rule "Assign people in North Carolina (NC) to sales region 1"
    ruleflow-group "test"
    lock-on-active true
when
    $p : Person( )
    $a : Address( state == "NC") from $p.address
then
    modify ($p) {} #Assign person to sales region 1 in a modify block
end

rule "Apply a discount to people in the city of Raleigh"
    ruleflow-group "test"
    lock-on-active true
when
    $p : Person( )
    $a : Address( city == "Raleigh") from $p.address
then
    modify ($p) {} #Apply discount to person in a modify block
end
```

In the above example, persons in Raleigh, NC should be assigned to sales region 1 and receive a discount; i.e., you would expect both rules to activate and fire. Instead you will find that only the second rule fires.

If you were to turn on the audit log, you would also see that when the second rule fires, it deactivates the first rule. Since the rule attribute **lock-on-active** prevents a rule from creating new activations when a set of facts change, the first rule fails to reactivate. Though the set of facts have not changed, the use of **from** returns a new fact for all intents and purposes each time it is evaluated.

First, it's important to review why you would use the above pattern. You may have many rules across different rule-flow groups. When rules modify working memory and other rules downstream of your RuleFlow (in different rule-flow groups) need to be reevaluated, the use of **modify** is critical. You don't, however, want other rules in the same rule-flow group to place activations on one another recursively. In this case, the **no-loop** attribute is ineffective, as it would only prevent a rule from activating itself recursively. Hence, you resort to **lock-on-active**.

There are several ways to address this issue:

- Avoid the use of **from** when you can assert all facts into working memory or use nested object references in your constraint expressions (shown below)
- Place the variable assigned used in the modify block as the last sentence in your condition (LHS)
- Avoid the use of **lock-on-active** when you can explicitly manage how rules within the same rule-flow group place activations on one another (explained below)

The preferred solution is to minimize use of **from** when you can assert all your facts into working memory directly. In the example above, both the Person and Address instance can be asserted into working memory. In this case, because the graph is fairly simple, an even easier solution is to modify your rules as follows:

```
rule "Assign people in North Carolina (NC) to sales region 1"
    ruleflow-group "test"
    lock-on-active true
when
    $p : Person(address.state == "NC" )
then
    modify ($p) {} #Assign person to sales region 1 in a modify block
end

rule "Apply a discount to people in the city of Raleigh"
    ruleflow-group "test"
    lock-on-active true
when
    $p : Person(address.city == "Raleigh" )
then
    modify ($p) {} #Apply discount to person in a modify block
end
```

Now, you will find that both rules fire as expected. However, it is not always possible to access nested facts as above. Consider an example where a Person holds one or more Addresses and you wish to use an existential quantifier to match people with at least one address that meets certain conditions. In this case, you would have to resort to the use of **from** to reason over the collection.

There are several ways to use **from** to achieve this and not all of them exhibit an issue with the use of **lock-on-active**. For example, the following use of **from** causes both rules to fire as expected:

```
rule "Assign people in North Carolina (NC) to sales region 1"
    ruleflow-group "test"
    lock-on-active true
when
    $p : Person($addresses : addresses)
    exists (Address(state == "NC") from $addresses)
then
    modify ($p) {} #Assign person to sales region 1 in a modify block
end

rule "Apply a discount to people in the city of Raleigh"
    ruleflow-group "test"
    lock-on-active true
when
    $p : Person($addresses : addresses)
    exists (Address(city == "Raleigh") from $addresses)
then
    modify ($p) {} #Apply discount to person in a modify block
end
```

However, the following slightly different approach does exhibit the problem:

```
rule "Assign people in North Carolina (NC) to sales region 1"
    ruleflow-group "test"
    lock-on-active true
when
    $assessment : Assessment()
    $p : Person()
    $addresses : List() from $p.addresses
    exists (Address( state == "NC") from $addresses)
then
    modify ($assessment) {} #Modify assessment in a modify block
end

rule "Apply a discount to people in the city of Raleigh"
    ruleflow-group "test"
    lock-on-active true
when
    $assessment : Assessment()
    $p : Person()
    $addresses : List() from $p.addresses
    exists (Address( city == "Raleigh") from $addresses)
then
    modify ($assessment) {} #Modify assessment in a modify block
end
```

In the above example, the $addresses variable is returned from the use of **from**. The example also introduces a new object, assessment, to highlight one possible solution in this case. If the $assessment variable assigned in the condition (LHS) is moved to the last condition in each rule, both rules fire as expected.

Though the above examples demonstrate how to combine the use of **from** with **lock-on-active** where no loss of rule activations occurs, they carry the drawback of placing a dependency on the order of conditions on the LHS. In addition, the solutions present greater complexity for the rule author in terms of keeping track of which conditions may create issues.

A better alternative is to assert more facts into working memory. In this case, a person's addresses may be asserted into working memory and the use of **from** would not be necessary.

There are cases, however, where asserting all data into working memory is not practical and we need to find other solutions. Another option is to reevaluate the need for **lock-on-active**. An alternative to **lock-on-active** is to directly manage how rules within the same rule-flow group activate one another by including conditions in each rule that prevent rules from activating each other recursively when working memory is modified. For example, in the case above where a discount is applied to citizens of Raleigh, a condition may be added to the rule that checks whether the discount has already been applied. If so, the rule does not activate.

4.8.2.21. 'collect'

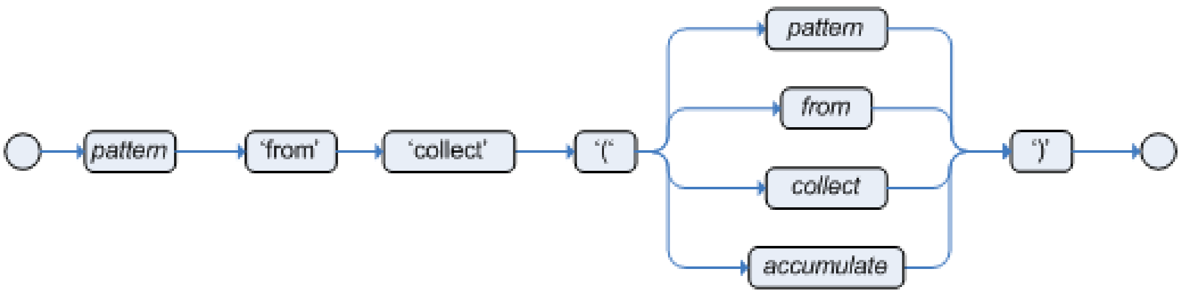


Figure 4.43. collect

The **collect** Conditional Element allows rules to reason over collection of objects collected from the given source or from the working memory. In first oder logic terms this is Cardinality Quantifier. A simple example:

```
import java.util.ArrayList

rule "Raise priority if system has more than 3 pending alarms"
when
    $system : System()
    $alarms : ArrayList( size >= 3 )
    from collect( Alarm( system == $system, status == 'pending' ) )
then
    # Raise priority, because system $system has
    # 3 or more alarms pending. The pending alarms
    # are $alarms.
end
```

In the above example, the rule will look for all pending alarms in the working memory for each given system and group them in **`ArrayLists`**. If 3 or more alarms are found for a given system, the rule will fire.

The **collect** CE result pattern can be any concrete class that implements the java.util.Collection interface and provides a default public constructor with no arguments. You can use default Java collections like ArrayList, LinkedList, HashSet, etc, or your own class, as long as it implements the java.util.Collection interface and provide a default public constructor with no arguments.

Both source and result patterns can be constrained as any other pattern.

Variables bound before the **collect** CE are in the scope of both source and result patterns and as so, you can use them to constrain both your source and result patterns. Although, the *collect(...)* is a scope delimiter for bindings, meaning that any binding made inside of it, is not available for use outside of it.

Collect accepts nested **from** elements, so the following example is a valid use of **collect**:

```
import java.util.LinkedList;

rule "Send a message to all mothers"
when
    $town : Town( name == 'Paris' )
    $mothers : LinkedList()
    from collect( Person( gender == 'F', children > 0 )
        from $town.getPeople()
        )
then
    # send a message to all mothers
end
```

4.8.2.22. 'accumulate'

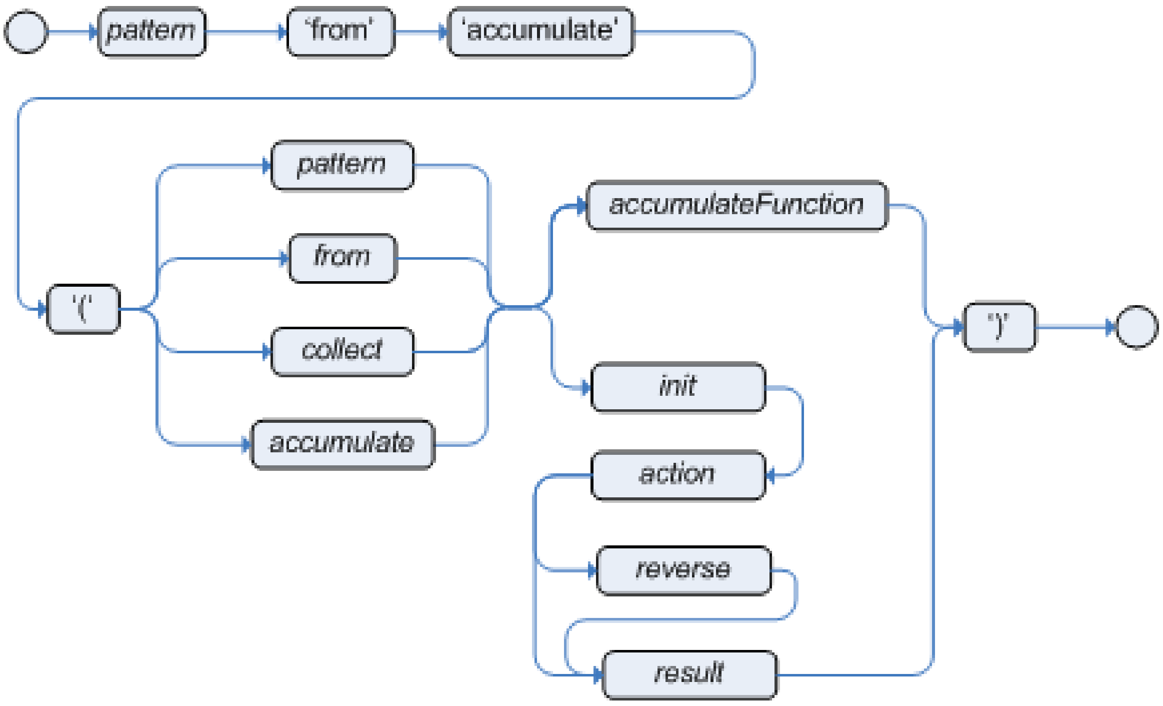


Figure 4.44. accumulate

The **accumulate** Conditional Element is a more flexible and powerful form of **collect** Conditional Element, in the sense that it can be used to do what **collect** CE does and also do things that **collect** CE is not capable to do. Basically what it does is it allows a rule to iterate over a collection of objects, executing custom actions for each of the elements, and at the end return a result object.

The general syntax of the **accumulate** CE is:

```
<result pattern> from accumulate( <source pattern>,
    init( <init code> ),
    action( <action code> ),
    reverse( <reverse code> ),
    result( <result expression> ) )
```

The meaning of each of the elements is the following:

- **<source pattern>**: the source pattern is a regular pattern that the engine will try to match against each of the source objects.
- **<init code>**: this is a semantic block of code in the selected dialect that will be executed once for each tuple, before iterating over the source objects.
- **<action code>**: this is a semantic block of code in the selected dialect that will be executed for each of the source objects.
- **<reverse code>**: this is an optional semantic block of code in the selected dialect that if present will be executed for each source object that no longer matches the source pattern. The objective of this code block is to "undo" any calculation done in the <action code> block, so that the engine can do decremental calculation when a source object is modified or retracted, hugely improving performance of these operations.
- **<result expression>**: this is a semantic expression in the selected dialect that is executed after all source objects are iterated.
- **<result pattern>**: this is a regular pattern that the engine tries to match against the object returned from the <result expression>. If it matches, the **accumulate** conditional element evaluates to **true** and the engine proceeds with the evaluation of the next CE in the rule. If it does not matches, the **accumulate** CE evaluates to **false** and the engine stops evaluating CEs for that rule.

It is easier to understand if we look at an example:

```
rule "Apply 10% discount to orders over US$ 100,00"
when
    $order : Order()
    $total : Number( doubleValue > 100 )
from accumulate( OrderItem( order == $order, $value : value ),
        init( double total = 0; ),
        action( total += $value; ),
        reverse( total -= $value; ),
        result( total ) )
then
    # apply discount to $order
end
```

In the above example, for each Order() in the working memory, the engine will execute the **init code** initializing the total variable to zero. Then it will iterate over all OrderItem() objects for that order, executing the **action** for each one (in the example, it will sum the value of all items into the total variable). After iterating over all OrderItem, it will return the value corresponding to the **result expression** (in the above example, the value of the total variable). Finally, the engine will try to match the result with the Number() pattern and if the double value is greater than 100, the rule will fire.

The example used Java as the semantic dialect, and as such, note that the usage of ';' is mandatory in the init, action and reverse code blocks. The result is an expression and as such, it does not admit ';'. If the user uses any other dialect, he must comply to that dialect specific syntax.

As mentioned before, the **reverse code** is optional, but it is strongly recommended that the user writes it in order to benefit from the *improved performance on update and retracts*.

The **accumulate** CE can be used to execute any action on source objects. The following example instantiates and populates a custom object:

```
rule "Accumulate using custom objects"
when
    $person   : Person( $likes : likes )
    $cheesery : Cheesery( totalAmount > 100 )
    from accumulate( $cheese : Cheese( type == $likes ),
        init( Cheesery cheesery = new Cheesery(); ),
        action( cheesery.addCheese( $cheese ); ),
        reverse( cheesery.removeCheese( $cheese ); ),
        result( cheesery ) );
then
    // do something
end
```

Accumulate Functions

The accumulate CE is a very powerful CE, but it gets real declarative and easy to use when using predefined functions that are known as Accumulate Functions. They work exactly like accumulate, but instead of explicitly writing custom code in every accumulate CE, the user can use predefined code for common operations.

For instance, the rule to apply discount on orders written in the previous section, could be written in the following way, using Accumulate Functions:

```
rule "Apply 10% discount to orders over US$ 100,00"
when
    $order : Order()
    $total : Number( doubleValue > 100 )
    from accumulate( OrderItem( order == $order, $value : value ),
    sum( $value ) )
then
    # apply discount to $order
end
```

In the above example, sum is an Accumulate Function and will sum the $value of all OrderItems and return the result.

JBoss Rules ships with the following built in accumulate functions:

- average
- min
- max

- count
- sum

These common functions accept any expression as input. For instance, if someone wants to calculate the average profit on all items of an order, a rule could be written using the average function:

```
rule "Average profit"
when
    $order : Order()
    $profit : Number()
    from accumulate( OrderItem( order == $order, $cost : cost, $price :
 price )
        average( 1 - $cost / $price ) )
then
    # average profit for $order is $profit
end
```

Accumulate Functions are all pluggable. That means that if needed, custom, domain specific functions can easily be added to the engine and rules can start to use them without any restrictions. To implement a new Accumulate Functions all one needs to do is to create a Java class that implements the org.drools.base.acumulators.AccumulateFunction interface and add a line to the configuration file or set a system property to let the engine know about the new function. As an example of an Accumulate Function implementation, the following is the implementation of the "average" function:

```
/*
 * Copyright 2007 JBoss Inc
 *
 * Licensed under the Apache License, Version 2.0 (the "License");
 * you may not use this file except in compliance with the License.
 * You may obtain a copy of the License at
 *
 *      http://www.apache.org/licenses/LICENSE-2.0
 *
 * Unless required by applicable law or agreed to in writing, software
 * distributed under the License is distributed on an "AS IS" BASIS,
 * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
 * See the License for the specific language governing permissions and
 * limitations under the License.
 *
 * Created on Jun 21, 2007
 */
package org.drools.base.accumulators;


/**
 * An implementation of an accumulator capable of calculating average
 values
 *
 * @author etirelli
 *
 */
public class AverageAccumulateFunction implements AccumulateFunction {

    protected static class AverageData {
        public int    count = 0;
        public double total = 0;
    }

    /* (non-Javadoc)
     * @see org.drools.base.accumulators.AccumulateFunction#createContext()
     */
    public Object createContext() {
        return new AverageData();
    }

    /* (non-Javadoc)
     * @see
 org.drools.base.accumulators.AccumulateFunction#init(java.lang.Object)
     */
    public void init(Object context) throws Exception {
        AverageData data = (AverageData) context;
        data.count = 0;
        data.total = 0;
    }

    /* (non-Javadoc)
     * @see
 org.drools.base.accumulators.AccumulateFunction#accumulate(java.lang.Object,
 java.lang.Object)
     */
    public void accumulate(Object context,
                           Object value) {
        AverageData data = (AverageData) context;
        data.count++;
        data.total += ((Number) value).doubleValue();
    }

    /* (non-Javadoc)
     * @see
 org.drools.base.accumulators.AccumulateFunction#reverse(java.lang.Object,
 java.lang.Object)
     */
    public void reverse(Object context,
                        Object value) throws Exception {
        AverageData data = (AverageData) context;
        data.count--;
        data.total -= ((Number) value).doubleValue();
    }

    /* (non-Javadoc)
```

```
     * @see
 org.drools.base.accumulators.AccumulateFunction#getResult(java.lang.Object)
     */
    public Object getResult(Object context) throws Exception {
        AverageData data = (AverageData) context;
        return new Double( data.count == 0 ? 0 : data.total / data.count );
    }

    /* (non-Javadoc)
     * @see
 org.drools.base.accumulators.AccumulateFunction#supportsReverse()
     */
    public boolean supportsReverse() {
        return true;
    }

}
```

The code for the function is very simple, as we could expect, as all the "dirty" integration work is done by the engine. Finally, to plug the function into the engine, we added it to the configuration file:

```
drools.accumulate.function.average =
 org.drools.base.accumulators.AverageAccumulateFunction
```

Where "drools.accumulate.function." is a prefix that must always be used, "average" is how the function will be used in the rule file, and "org.drools.base.accumulators.AverageAccumulateFunction" is the fully qualified name of the class that implements the function behavior.

4.8.3. The Right Hand Side (then)

The Right Hand Side (RHS) is a common name for the consequence or action part of the rule; this part should contain a list of actions to be executed. It is bad practice to use imperative or conditional code in the RHS of a rule; as a rule should be atomic in nature - "when this, then do this", not "when this, maybe do this". The RHS part of a rule should also be kept small, thus keeping it declarative and readable. If you find you need imperative and/or conditional code in the RHS, then maybe you should be breaking that rule down into multiple rules. The main purpose of the RHS is to insert, retractor modify working memory data. To assist with that there are a few convenience methods you can use to modify working memory; without having to first reference a working memory instance.

"update(object, handle);" will tell the engine that an object has changed (one that has been bound to something on the LHS) and rules may need to be reconsidered.

"update(object);" can also be used, here the KnowledgeHelper will lookup the facthandle for you, via an identity check, for the passed object.

"insert(new Something());" will place a new object of your creation in working memory.

"insertLogical(new Something());" is similar to insert, but the object will be automatically retracted when there are no more facts to support the truth of the currently firing rule.

"retract(handle);" removes an object from working memory.

These convenience methods are basically macros that provide short cuts to the KnowledgeHelper instance (refer to the KnowledgeHelper interface for more advanced operations). The KnowledgeHelper interface is made available to the RHS code block as a variable called "drools". If you provide "Property Change Listeners" to your Java beans that you are inserting into the engine, you can avoid the need to call "update" when the object changes.

4.8.4. A note on automatic boxing, unboxing and primitive types

JBoss Rules attempts to preserve numbers in their primitive or object wrapper form, so a variable bound to an integer primitive when used in a code block or expression will no longer need manual unboxing. A variable bound to an object wrapper will remain as an object; the existing Java 1.5 and Java 5 rules to handling auto boxing/unboxing apply in this case. When evaluating field constraints the system attempts to coerce one of the values into a comparable format; so a primitive is comparable to an object wrapper.

4.9. Query

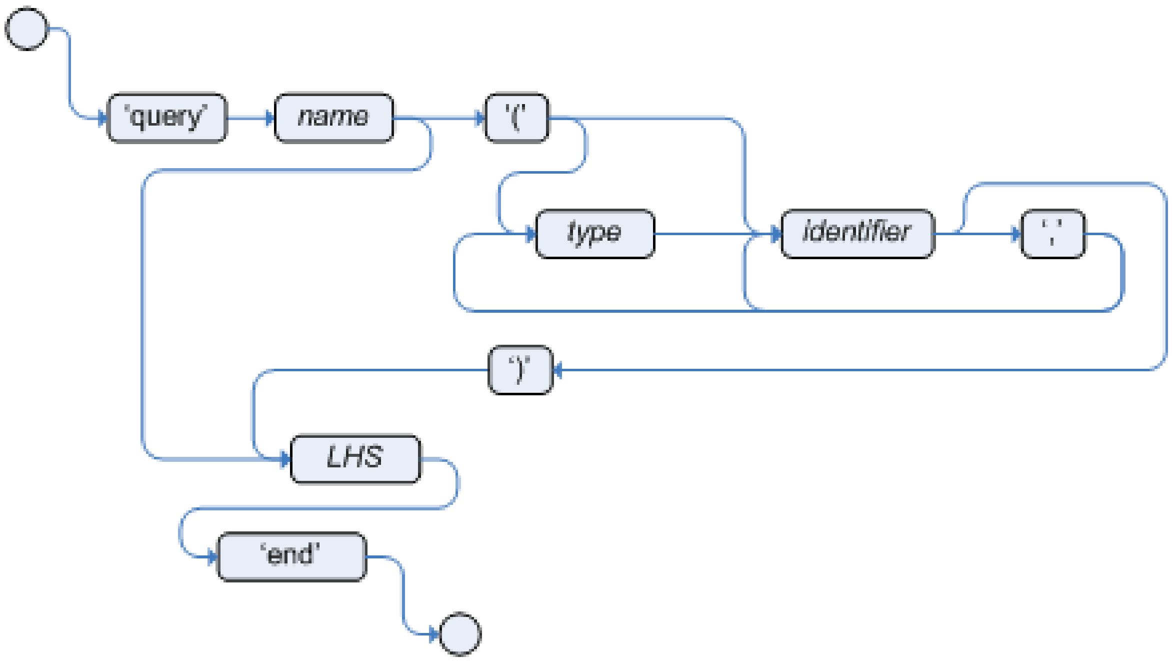


Figure 4.45. query

A query contains the structure of the LHS of a rule only (you don't specify "when" or "then"). It is simply a way to query the working memory for facts that match the conditions stated. A query has an optional set of parameters, each of which can also be optionally typed. If type is not given then Object type is assumed. The engine will attempt to coerce the values as needed. Query names are global to the RuleBase, so do not add queries of the same name to different packages for the same Rule Base.

To return the results use `WorkingMemory.getQueryResults("name")`, where "name" is the query's name. This returns a list of query results, which allow you to to get to the objects that matched the query.

The first example creates a simple query for all the people over the age of 30. The second one, using parameters, combines the age limit with a location.

```
query "people over the age of 30"
    person : Person( age > 30 )
end
```

Example 4.56. Query People over the age of 30

```
query "people over the age of x"  (int x, String y)
    person : Person( age > x, location == y )
end
```

Example 4.57. Query People over the age of x, and who live in y

We iterate over the returned QueryResults using a standard 'for' loop. Each element returns a QueryResult which we can use to access each of the columns in the Tuple. Those columns can be accessed by bound declaration name or index position.

```
QueryResults results =
    workingMemory.getQueryResults( "people over the age of 30" );
System.out.println("we have "+results.size()+" people over the age of 30");
System.out.println( "These people are are over 30:" );

for ( Iterator it = results.iterator; it.hasNext(); )
{
    QueryResult result = ( QueryResult ) it.next();
    Person person = ( Person ) result.get( "person" );
    System.out.println( person.getName() + "\n" );
}
```

Example 4.58. Query People over the age of 30

4.10. Domain Specific Languages

Domain Specific Languages (or DSLs) are a way of extending the rule language to your problem domain. They are wired in to the rule language for you, and can make use of all the underlying rule language and engine features.

DSLs are used both in the IDE, as well as the web based BRMS UI. Of course as rules are text, you can use them even without this tooling.

4.10.1. When to use a DSL

DSLs can serve as a layer of separation between rule authoring (and rule authors) and the domain objects that the engine operates on. DSLs can also act as "templates" of conditions or actions that are used over and over in your rules, perhaps only with parameters changing each time. If your rules need to be read and validated by less technical folk, (such as Business Analysts) the DSLs are definitely for you. If the conditions or consequences of your rules follow similar patterns which you can express in a template. You wish to hide away your implementation details, and focus on the business rule. You want to provide a controlled means of editing rules based on pre-defined templates.

DSLs have no impact on the rules at runtime, they are just a parse/compile time feature.

4.10.2. Editing and managing a DSL

A DSL's configuration like most things is stored in plain text. If you use the IDE, you get a nice graphical editor (with some validation), but the format of the file is quite simple, and is basically a properties file.

Note that since JBoss Rules 4.0, DSLs have become more powerful in allowing you to customize almost any part of the language, including keywords. Regular expressions can also be used to match words/sentences if needed (this is provided for enhanced localization). However, not all features are supported by all the tools (although you can use them, the content assistance just may not be 100% accurate in certain cases).

```
[when]This is {something}=Something(something=={something})
```

Example 4.59. Example DSL mapping

Referring to the above example, the [when] refers to the scope of the expression: i.e. does it belong on the LHS or the RHS of a rule. The part after the [scope] is the expression that you use in the rule (typically a natural language expression, but it doesn't have to be). The part on the right of the "=" is the mapping into the rule language (of course the form of this depends on if you are talking about the RHS or the LHS - if its the LHS, then its the normal LHS syntax, if its the RHS then its fragments of Java code for instance).

The parser will take the expression you specify, and extract the values that match where the {something} (named Tokens) appear in the input. The values that match the tokens are then interpolated with the corresponding {something} (named Tokens) on the right hand side of the mapping (the target expression that the rule engine actually uses).

Note also that the "sentences" above can be regular expressions. This means the parser will match the sentence fragments that match the expressions. This means you can use (for instance) the '?' to indicate the character before it is optional (think of each sentence as a regular expression pattern - this means if you want to use regular expression characters - you will need to escape them with a '\' of course.

It is important to note that the DSL expressions are processed one line at a time. This means that in the above example, all the text after "This is " to the end of the line will be included as the value for "{something}" when it is interpolated into the target string. This may not be exactly what you want, as you may want to "chain" together different DSL expressions to generate a target expression. The best way around this is to make sure that the {tokens} are enclosed with characters or words. This means that the parser will scan along the sentence, and pluck out the value BETWEEN the characters (in the example below they are double-quotes). Note that the characters that surround the token are not included in when interpolating, just the contents between them (rather then all the way to the end of the line, as would otherwise be the case).

As a rule of thumb, use quotes for textual data that a rule editor may want to enter. You can also wrap words around the {tokens} to make sure you enclose the data you want to capture (see other example).

```
[when]This is "{something}" and
 "{another}"=Something(something=="{something}", another=="{another}")
[when]This is {also} valid=Another(something=="{also}")
```

Example 4.60. Example with quotes

It is a good idea to try and avoid punctuation in your DSL expressions where possible, other then quotes and the like - keep it simple and things will be easier. Using a DSL can make debugging slightly harder when you are first building rules, but it can make the maintenance easier (and of course the readability of the rules).

The "{" and "}" characters should only be used on the left hand side of the mapping (the expression) to mark tokens. On the right hand side you can use "{" and "}" on their own if needed - such as

```
if (foo) \{ doSomething();\ }
```

as well as with the token names as shown above.

Important

If you want curly braces to appear literally as curly braces, then escape them with a backslash (\). Otherwise it may think it is a token to be replaced.

Don't forget that if you are capturing strings from users, you will also need the quotes on the right hand side of the mapping, just like a normal rule, as the result of the mapping must be a valid expression in the rule language.

```
#This is a comment to be ignored.
[when]There is a Person with name of "{name}"=Person(name=="{name}")
[when]Person is at least {age} years old and lives in
 "{location}"=Person(age > {age}, location=="{location}")
[then]Log "{message}"=System.out.println("{message}");
[when]And = and
```

Example 4.61. Some more examples

Referring to the above examples, this would render the following input as shown below:

```
There is a Person with name of "kitty" ---> Person(name="kitty")
Person is at least 42 years old and lives in "atlanta" ---> Person(age >
 42, location="atlanta")
Log "boo" ---> System.out.println("boo");
There is a Person with name of "bob" and Person is at least 30 years old
 and lives in "atlanta"
          ---> Person(name="kitty") and Person(age > 30,
 location="atlanta")
```

Example 4.62. Some examples as processed

4.10.3. Using a DSL in your rules

A good way to get started if you are new to Rules (and DSLs) is just write the rules as you normally would against your object model. You can unit test as you go (like a good agile citizen!). Once you feel comfortable, you can look at extracting a domain language to express what you are doing in the rules. Note that once you have started using the "expander" keyword, you will get errors if the parser does not recognize expressions you have in there - you need to move everything to the DSL. As a way around this, you can prefix each line with ">" and it will tell the parser to take that line literally, and not try and expand it (this is handy also if you are debugging why something isn't working).

Also, it is better to rename the extension of your rules file from ".drl" to ".dslr" when you start using DSLs, as that will allow the IDE to correctly recognize and work with your rules file.

As you work through building up your DSL, you will find that the DSL configuration stabilizes pretty quickly, and that as you add new rules and edit rules you are reusing the same DSL expressions over and over. The aim is to make things as fluent as possible.

To use the DSL when you want to compile and run the rules, you will need to pass the DSL configuration source along with the rule source.

```
// source is a reader for the rule source,
// dsl is a reader for the DSL configuration
PackageBuilder builder = new PackageBuilder();
builder.addPackageFromDrl( source, dsl );
```

You will also need to specify the expander by name in the rule source file:

```
expander your-expander.dsl
```

Typically you keep the DSL in the same directory as the rule, but this is not required if you are using the above API (you only need to pass a reader). Otherwise everything is just the same.

You can chain DSL expressions together on one line, as long as it is clear to the parser what the {tokens} are (otherwise you risk reading in too much text until the end of the line). The DSL expressions are processed according to the mapping file, top to bottom in order. You can also have the resulting rule expressions span lines - this means that you can do things like:

```
There is a person called Bob who is happy
  Or
There is a person called Mike who is sad
```

Example 4.63. Chaining DSL Expressions

Of course this assumes that "Or" is mapped to the "or" conditional element (which is a sensible thing to do).

4.10.4. Adding constraints to facts

A common requirement when writing rule conditions is to be able to add many constraints to fact declarations. A fact may have many (dozens) of fields, all of which could be used or not used at various times. To come up with every combination as separate DSL statements would in many cases not be feasible.

The DSL facility allows you to achieve this however, with a simple convention. If your DSL expression starts with a "-", then it will be assumed to be a field constraint, which will be added to the declaration that is above it (one per line).

This is easier to explain with an example. Lets take look at Cheese class, with the following fields: type, price, age, country. We can express some LHS condition in normal DRL like the following

```
Cheese(age < 5, price == 20, type=="stilton", country=="ch")
```

If you know ahead of time that you will use all the fields, all the time, it is easy to do a mapping using the above techniques. However, chances are that you will have many fields, and many combinations. If this is the case, you can setup your mappings like so:

```
[when]There is a Cheese with=Cheese()
[when]- age is less than {age}=age<{age}
[when]- type is '{type}'=type=='{type}'
[when]- country equal to '{country}'=country=='{country}'
```

Important

It is *NOT* possible to use the "-" feature after an *accumulate* statement to add constraints to the accumulate pattern. This limitation will be removed in a future version.

You can then write rules with conditions like the following:

```
There is a Cheese with
        - age is less than 42
        - type is 'stilton'
```

The parser will pick up the "-" lines (they have to be on their own line) and add them as constraints to the declaration above. So in this specific case, using the above mappings, is the equivalent to doing (in DRL):

```
Cheese(age<42, type=='stilton')
```

The parser will do all the work for you, meaning you just define mappings for individual constraints, and can combine them how you like (if you are using context assistant, if you press "-" followed by CTRL+space it will conveniently provide you with a filtered list of field constraints to choose from.

To take this further, after alter the DSL to have [when][org.drools.Cheese]- age is less than {age} ... (and similar to all the items in the example above).

The extra [org.drools.Cheese] indicates that the sentence only applies for the main constraint sentence above it (in this case "There is a Cheese with"). For example, if you have a class called "Cheese" - then if you are adding constraints to the rule (by typing "-" and waiting for content assistance) then it will know that only items marked as having an object-scope of "com.yourcompany.Something" are valid, and suggest only them. This is entirely optional (you can leave out that section if needed - OR it can be left blank).

4.10.5. How it works

DSLs kick in when the rule is parsed. The DSL configuration is read and supplied to the parser, so the parser can "expand" the DSL expressions into the real rule language expressions.

When the parser is processing the rules, it will check if an "expander" representing a DSL is enabled, if it is, it will try to expand the expression based on the context of where it is the rule. If an expression can not be expanded, then an error will be added to the results, and the line number recorded (this insures against typos when editing the rules with a DSL). At present, the DSL expander is fairly space sensitive, but this will be made more tolerant in future releases (including tolerance for a wide range of punctuation).

The expansion itself works by trying to match a line against the expression in the DSL configuration. The values that correspond to the token place holders are stored in a map based on the name of the token, and then interpolated to the target mapping. The values that match the token placeholders are extracted by either searching until the end of the line, or until a character or word after the token place holder is matched. The "{" and "}" are not included in the values that are extracted, they are only used to demarcate the tokens - you should not use these characters in the DSL expression (but you can in the target).

4.10.6. Creating a DSL from scratch

Rules engines require an object or a data model to operate on - in many cases you may know this up front. In other cases the model will be discovered with the rules. In any case, rules generally work better with simpler flatter object models. In some cases, this may mean having a rule object model which is a subset of the main applications model (perhaps mapped from it). Object models can often have complex relationships and hierarchies in them - for rules you will want to simplify and flatten the model where possible, and let the rule engine infer relationships (as it provides future flexibility). As stated previously, DSLs can have an advantage of providing some insulation between the object model and the rule language.

Coming up with a DSL is a collaborative approach for both technical and domain experts. Historically there was a role called "knowledge engineer" which is someone skilled in both the rule technology, and in capturing rules. Over a short period of time, your DSL should stabilize, which means that changes to rules are done entirely using the DSL. A suggested approach if you are starting from scratch is the following workflow:

- Capture rules as loose "if then" statements - this is really to get an idea of size and complexity (possibly in a text document).
- Look for recurring statements in the rules captured. Also look for the rule objects/fields (and match them up with what may already be known of the object model).
- Create a new DSL, and start adding statements from the above steps. Provide the "holes" for data to be edited (as many statements will be similar, with only some data changing).
- Use the above DSL, and try to write the rules just like that appear in the "if then" statements from the first and second steps. Iterate this process until patterns appear and things stabilize. At this stage, you are not so worried about the rule language underneath, just the DSL.
- At this stage you will need to look at the Objects, and the Fields that are needed for the rules, reconcile this with the datamodel so far.
- Map the DSL statements to the rule language, based on the object model. Then repeat the process. Obviously this is best done in small steps, to make sure that things are on the right track.

4.10.7. Scope and keywords

If you are editing the DSL with the GUI, or as text, you will notice there is a [scope] item at the start of each mapping line. This indicates if the sentence/word applies to the LHS, RHS or is a keyword. Valid values for this are [condition], [consequence] and [keyword] (with [when] and [then] being the same as [condition] and [consequence] respectively). When [keyword] is used, it means you can map any keyword of the language like "rule" or "end" to something else. Generally this is only used when you want to have a non English rule language (and you would ideally map it to a single word).

4.10.8. DSLs in the BRMS and IDE

You can use DSLs in the BRMS in both guided editor rules, and textual rules that use a DSL. (In fact, the same applies to the IDE).

In the guided editor - the DSLs generally have to be simpler - what you are doing is defining little "forms" to capture data from users in text fields (i.e. as you pick a DSL expression - it will add an item to the GUI which only allows you enter data in the {token} parts of a DSL expression). You can not use sophisticated regular expressions to match text. However, in textual rules (which have a .dslr extension in the IDE) you are free to use the full power as needed.

In the BRMS - when you build a package the DSLs are already included and all the work is done for you. In the IDE (or in any IDE) - you will either need to use the drools-ant task, or otherwise use the code shown in sections above.

4.11. XML Rule Language

As an option, JBoss Rules also supports a "native" XML rule language as an alternative to DRL. This allows you to capture and manage your rules as XML data. Just like the non-XML DRL format, the XML format is parsed into the internal "AST" representation - as fast as possible (using a SAX parser). There is no external transformation step required. All the features are available with XML that are available to DRL.

4.11.1. When to use XML

There are several scenarios that XML is desirable. However, we recommend that it is not a default choice, as XML is not readily human readable (unless you like headaches) and can create visually bloated rules.

Other scenarios where you may want to use the XML format are if you have a tool that generates rules from some input (programmatically generated rules), or perhaps interchange from another rule language, or from another tool that emits XML (using XSLT you can easily transform between XML formats). Note you can always generate normal DRL as well.

Alternatively you may be embedding JBoss Rules in a product that already uses XML for configuration, so you would like the rules to be in an XML format. You may be creating your own rule language on XML - note that you can always use the AST objects directly to create your own rule language as well (the options are many, due to the open architecture).

4.11.2. The XML format

A full W3C standards (XML Schema) compliant XSD is provided that describes the XML language, which will not be repeated here verbatim. A summary of the language follows.

```
<?xml version="1.0" encoding="UTF-8"?>

<package name="com.sample"
  xmlns="http://drools.org/drools-4.0"
  xmlns:xs="http://www.w3.org/2001/XMLSchema-instance"
  xs:schemaLocation="http://drools.org/drools-4.0 drools-4.0.xsd">
```

```
<import name="java.util.HashMap" />
<import name="org.drools.*" />

<global identifier="x" type="com.sample.X" />
<global identifier="yada" type="com.sample.Yada" />

<function return-type="void" name="myFunc">
  <parameter identifier="foo" type="Bar" />
  <parameter identifier="bada" type="Bing" />
  <body>System.out.println("hello world");</body>
</function>

<rule name="simple_rule">
  <rule-attribute name="salience" value="10" />
  <rule-attribute name="no-loop" value="true" />
  <rule-attribute name="agenda-group" value="agenda-group" />
  <rule-attribute name="activation-group" value="activation-group" />

  <lhs>
    <pattern identifier="foo2" object-type="Bar" >
      <or-constraint-connective>
        <and-constraint-connective>
          <field-constraint field-name="a">
            <or-restriction-connective>
              <and-restriction-connective>
                <literal-restriction evaluator=">" value="60" />
                <literal-restriction evaluator="<" value="70" />
              </and-restriction-connective>
              <and-restriction-connective>
                <literal-restriction evaluator="<" value="50" />
                <literal-restriction evaluator=">" value="55" />
              </and-restriction-connective>
            </or-restriction-connective>
          </field-constraint>

          <field-constraint field-name="a3">
            <literal-restriction evaluator="==" value="black" />
          </field-constraint>
        </and-constraint-connective>

        <and-constraint-connective>
          <field-constraint field-name="a">
            <literal-restriction evaluator="==" value="40" />
          </field-constraint>

          <field-constraint field-name="a3">
            <literal-restriction evaluator="==" value="pink" />
          </field-constraint>
        </and-constraint-connective>
```

```
          <and-constraint-connective>
            <field-constraint field-name="a">
              <literal-restriction evaluator="==" value="12"/>
            </field-constraint>

            <field-constraint field-name="a3">
              <or-restriction-connective>
                <literal-restriction evaluator="==" value="yellow"/>
                <literal-restriction evaluator="==" value="blue" />
              </or-restriction-connective>
            </field-constraint>
          </and-constraint-connective>
        </or-constraint-connective>
      </pattern>

      <not>
        <pattern object-type="Person">
          <field-constraint field-name="likes">
            <variable-restriction evaluator="==" identifier="type"/>
          </field-constraint>
        </pattern>

        <exists>
          <pattern object-type="Person">
            <field-constraint field-name="likes">
              <variable-restriction evaluator="==" identifier="type"/>
            </field-constraint>
          </pattern>
        </exists>
      </not>

      <or-conditional-element>
        <pattern identifier="foo3" object-type="Bar" >
          <field-constraint field-name="a">
            <or-restriction-connective>
              <literal-restriction evaluator="==" value="3" />
              <literal-restriction evaluator="==" value="4" />
            </or-restriction-connective>
          </field-constraint>
          <field-constraint field-name="a3">
            <literal-restriction evaluator="==" value="hello" />
          </field-constraint>
          <field-constraint field-name="a4">
            <literal-restriction evaluator="==" value="null" />
          </field-constraint>
        </pattern>

        <pattern identifier="foo4" object-type="Bar" >
          <field-binding field-name="a" identifier="a4" />
          <field-constraint field-name="a">
            <literal-restriction evaluator="!=" value="4" />
            <literal-restriction evaluator="!=" value="5" />
          </field-constraint>
        </pattern>
      </or-conditional-element>

      <pattern identifier="foo5" object-type="Bar" >
        <field-constraint field-name="b">
          <or-restriction-connective>
            <return-value-restriction evaluator="==" >
              a4 + 1
            </return-value-restriction>
            <variable-restriction evaluator=">" identifier="a4" />
            <qualified-identifier-restriction evaluator="==">
              org.drools.Bar.BAR_ENUM_VALUE
            </qualified-identifier-restriction>
          </or-restriction-connective>
        </field-constraint>
      </pattern>

      <pattern identifier="foo6" object-type="Bar" >
        <field-binding field-name="a" identifier="a4" />
        <field-constraint field-name="b">
          <literal-restriction evaluator="==" value="6" />
        </field-constraint>
      </pattern>
    </lhs>
    <rhs>
      if ( a == b ) {
      assert( foo3 );
      } else {
      retract( foo4 );
      }
      System.out.println( a4 );
    </rhs>
  </rule>

</package>
```

In the preceding XML text you will see the typical XML element, the package declaration, imports, globals, functions, and the rule itself. Most of the elements are self explanatory if you have some understanding of the JBoss Rules features.

The `import` elements import the types you wish to use in the rule.

The `global` elements define global objects that can be referred to in the rules.

The `function` contains a function declaration, for a function to be used in the rules. You have to specify a return type, a unique name and parameters, in the body goes a snippet of code.

The rule is discussed below.

```
<rule name="simple_rule">
<rule-attribute name="salience" value="10" />
<rule-attribute name="no-loop" value="true" />
<rule-attribute name="agenda-group" value="agenda-group" />
<rule-attribute name="activation-group" value="activation-group" />

<lhs>
    <pattern identifier="cheese" object-type="Cheese">
        <from>
            <accumulate>
                <pattern object-type="Person"></pattern>
                <init>
                    int total = 0;
                </init>
                <action>
                    total += $cheese.getPrice();
                </action>
                <result>
                    new Integer( total ) );
                </result>
            </accumulate>
        </from>
    </pattern>

    <pattern identifier="max" object-type="Number">
        <from>
            <accumulate>
               <pattern identifier="cheese" object-type="Cheese"></pattern>
               <external-function evaluator="max" expression="$price"/>
            </accumulate>
        </from>
    </pattern>
</lhs>
<rhs>
    list1.add( $cheese );
</rhs>
</rule>
```

Example 4.64. Detail of rule element

In the above detail of the rule we see that the rule has LHS and RHS (conditions and consequence) sections. The RHS is simple, it is just a block of semantic code that will be executed when the rule is activated. The LHS is slightly more complicated as it contains nested elements for conditional elements, constraints and restrictions.

A key element of the LHS is the Pattern element. This allows you to specify a type (class) and perhaps bind a variable to an instance of that class. Nested under the pattern object are constraints and restrictions that have to be met. The Predicate and Return Value constraints allow Java expressions to be embedded.

That leaves the conditional elements, not, exists, and, or etc. They work like their DRL counterparts. Elements that are nested under and an "and" element are logically "anded" together. Likewise with "or" (and you can nest things further). "Exists" and "Not" work around patterns, to check for the existence or nonexistence of a fact meeting the pattern's constraints.

The Eval element allows the execution of a valid snippet of Java code - as long as it evaluates to a boolean (do not end it with a semi-colon, as it is just a fragment) - this can include calling a function. The Eval is less efficient than the columns, as the rule engine has to evaluate it each time, but it is a "catch all" feature for when you can express what you need to do with Column constraints.

4.11.3. Automatic transforming between formats (XML and DRL)

JBoss Rules comes with some utility classes to transform between formats. This works by parsing the rules from the source format into the AST, and then "dumping" out to the appropriate target format. This allows you, for example, to write rules in DRL, and when needed, export to XML if necessary at some point in the future.

The classes to look at if you need to do this are:

```
XmlDumper - for exporting XML.
DrlDumper - for exporting DRL.
DrlParser - reading DRL.
XmlPackageReader - reading XML.
```

Using combinations of the above, you can convert between any format (including round trip). Note that DSLs will not be preserved (from DRLs that are using a DSL) - but they will be able to be converted.

Feel free to make use of XSLT to provide all sorts of possibilities for XML, XSLT and its ilk are what make XML powerful.

Authoring

5.1. Decision tables in spreadsheets

Decision tables are a "precise yet compact" way of representing conditional logic, and are well suited to *business* level rules.

Drools supports managing rules in a Spreadsheet format. Formats supported are Excel, and CSV. Meaning that a variety of spreadsheet programs (such as Microsoft Excel, OpenOffice.org Calc amongst others) can be utalized. It is expected that web based decision table editors will be included in a near future release.

Decision tables are an old concept (in software terms) but have proven useful over the years. Very briefly speaking, in Drools decision tables are a way to generate rules driven from the data entered into a spreadsheet. All the usual features of a spreadsheet for data capture and manipulation can be taken advantage of.

5.1.1. When to use Decision tables

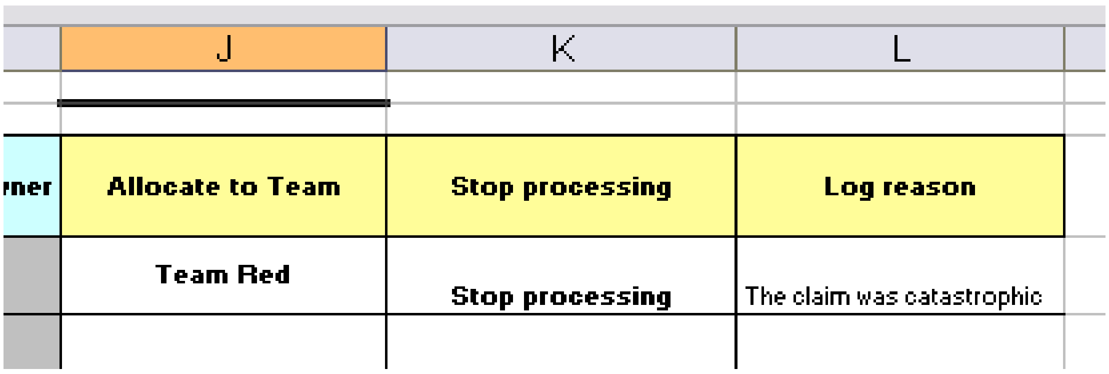
Decision tables my want to be considered as a course of action if rules exist that can be expressed as rule templates + data. In each row of a decision table, data is collected that is combined with the templates to generate a rule.

Many businesses already use spreadsheets for managing data, calculations etc. If you are happy to continue this way, you can also manage your business rules this way. This also assumes you are happy to manage packages of rules in .xls or .csv files. Decision tables are not recommenced for rules that do not follow a set of templates, or where there are a small number of rules (or if there is a dislike towards software like excel or open office). They are ideal in the sense that there can be control over what *parameters* of rules can be edited, without exposing the rules directly.

Decision tables also provide a degree of insulation from the underlying object model.

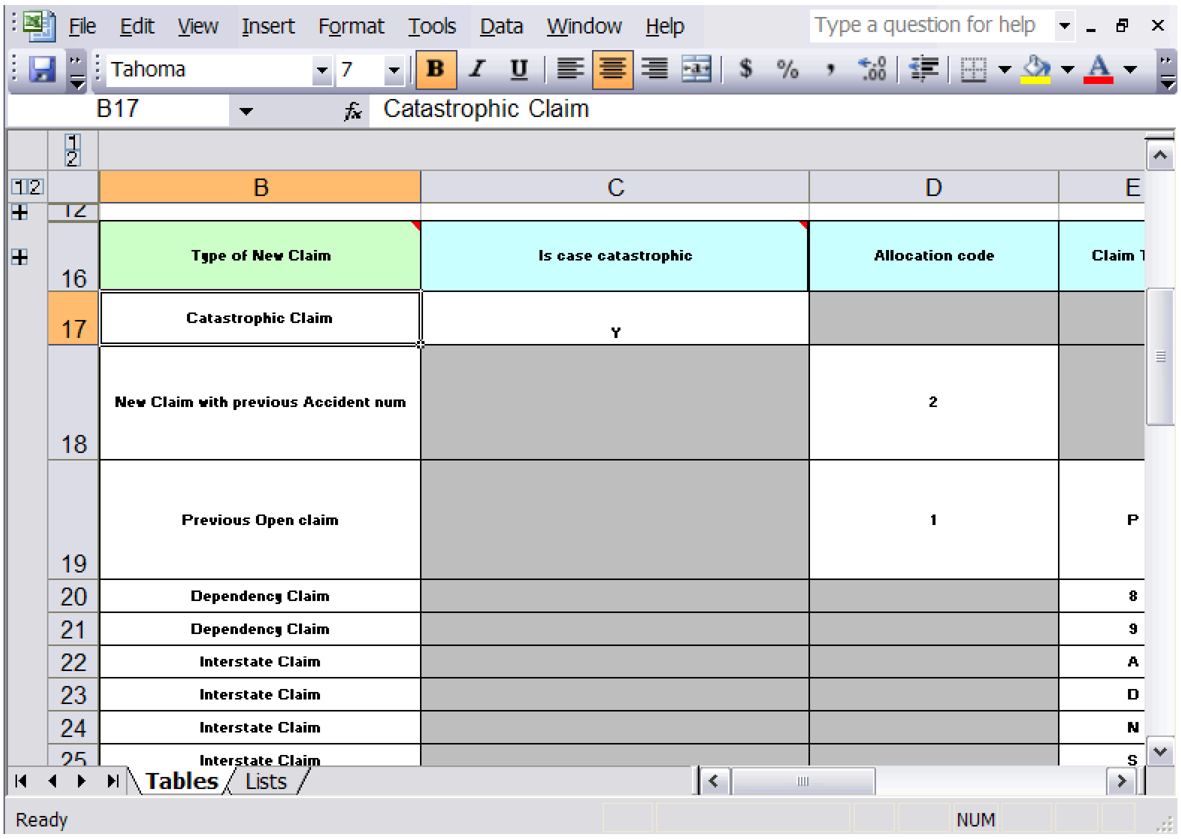
5.1.2. Overview

Here are some examples of real world decision tables.



	J	K	L
rner	Allocate to Team	Stop processing	Log reason
	Team Red	Stop processing	The claim was catastrophic

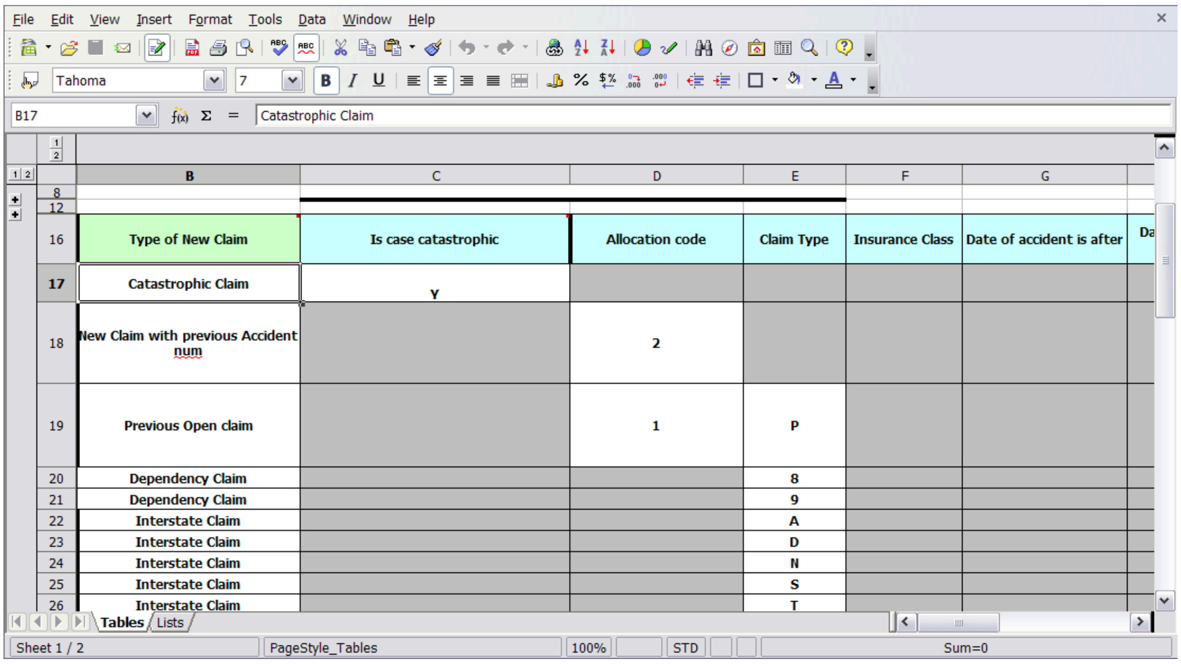
Figure 5.1. Can have multiple actions for a rule row



B17 Catastrophic Claim

	B	C	D	E
16	Type of New Claim	Is case catastrophic	Allocation code	Claim T
17	Catastrophic Claim	Y		
18	New Claim with previous Accident num		2	
19	Previous Open claim		1	P
20	Dependency Claim			8
21	Dependency Claim			9
22	Interstate Claim			A
23	Interstate Claim			D
24	Interstate Claim			N
25	Interstate Claim			S

Figure 5.2. Using excel to edit a decision table



B17 Catastrophic Claim

	B	C	D	E	F	G	
16	Type of New Claim	Is case catastrophic	Allocation code	Claim Type	Insurance Class	Date of accident is after	Da
17	Catastrophic Claim	Y					
18	New Claim with previous Accident num		2				
19	Previous Open claim		1	P			
20	Dependency Claim			8			
21	Dependency Claim			9			
22	Interstate Claim			A			
23	Interstate Claim			D			
24	Interstate Claim			N			
25	Interstate Claim			S			
26	Interstate Claim			T			

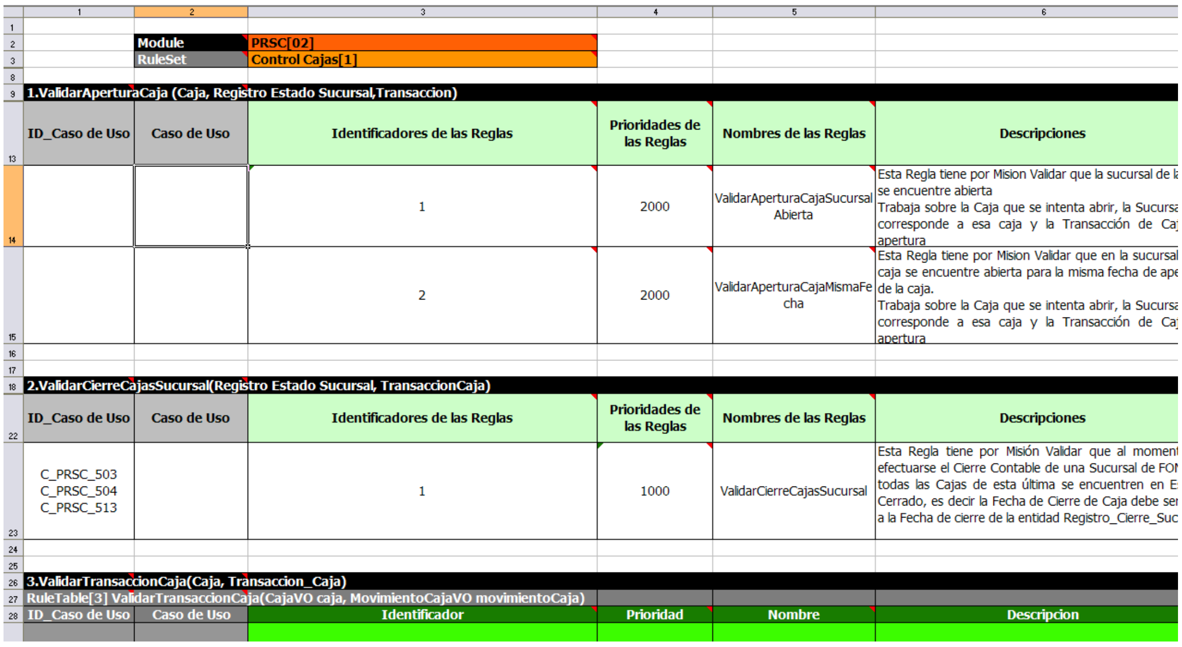
Figure 5.3. Using OpenOffice

In the above examples, the technical aspects of the decision table have been collapsed away (standard spreadsheet feature).

The rules start from row 17 (each row results in a rule). The conditions are in column C, D, E etc.. (off screen are the actions). The value in the cells are quite simple, and have meaning when looking at the headers in Row 16. Column B is just a description. It is conventional to use color to make it obvious what the different areas of the table mean.

> **Note**
>
> Note that although the decision tables look like they process top down, this is not necessarily the case. Ideally, if the rules are able to be authored in such a way as order does not matter (simply as it makes maintenance easier, as rows will not need to be shifted around all the time).

As each row is a rule, the same principles apply. As the rule engine processes the facts, any rules that match may fire (some people are confused by this. It is possible to clear the agenda when a rule fires and simulate a very simple decision table where the first match exists). Also note that you can have multiple tables on the one spreadsheet (so rules can be grouped where they share common templates, yet at the end of the day they are all combined into a one rule package). Decision tables are essentially a tool to generate DRL rules automatically.

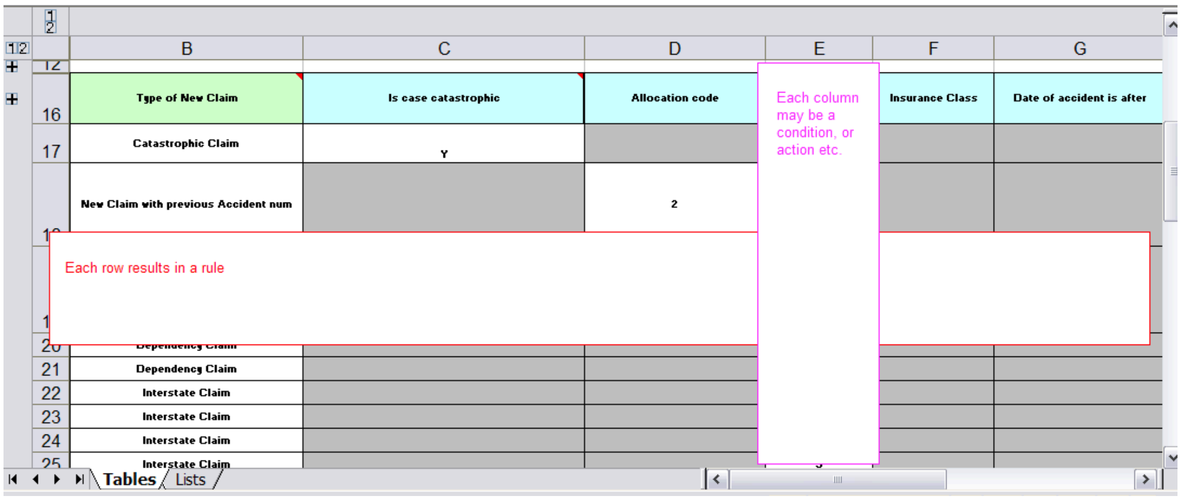


	1	2	3	4	5	6
1						
2		Module	PRSC[02]			
3		RuleSet	Control Cajas[1]			
8						
9	1.ValidarAperturaCaja (Caja, Registro Estado Sucursal,Transaccion)					
13	ID_Caso de Uso	Caso de Uso	Identificadores de las Reglas	Prioridades de las Reglas	Nombres de las Reglas	Descripciones
14			1	2000	ValidarAperturaCajaSucursal Abierta	Esta Regla tiene por Mision Validar que la sucursal de l se encuentre abierta Trabaja sobre la Caja que se intenta abrir, la Sucurs corresponde a esa caja y la Transacción de Ca apertura
15			2	2000	ValidarAperturaCajaMismaFe cha	Esta Regla tiene por Mision Validar que en la sucursal caja se encuentre abierta para la misma fecha de ape de la caja. Trabaja sobre la Caja que se intenta abrir, la Sucurs corresponde a esa caja y la Transacción de Ca apertura
16						
17						
18	2.ValidarCierreCajasSucursal(Registro Estado Sucursal, TransaccionCaja)					
22	ID_Caso de Uso	Caso de Uso	Identificadores de las Reglas	Prioridades de las Reglas	Nombres de las Reglas	Descripciones
23	C_PRSC_503 C_PRSC_504 C_PRSC_513		1	1000	ValidarCierreCajasSucursal	Esta Regla tiene por Misión Validar que al moment efectuarse el Cierre Contable de una Sucursal de FO todas las Cajas de esta última se encuentren en E Cerrado, es decir la Fecha de Cierre de Caja debe se a la Fecha de cierre de la entidad Registro_Cierre_Suc
24						
25						
26	3.ValidarTransaccionCaja(Caja, Transaccion_Caja)					
27	RuleTable[3] ValidarTransaccionCaja(CajaVO caja, MovimientoCajaVO movimientoCaja)					
28	ID_Caso de Uso	Caso de Uso	Identificador	Prioridad	Nombre	Descripcion

Figure 5.4. Using multiple tables for grouping similar rules

5.1.3. How decision tables work

The key point to keep in mind is that in a decision table, each row is a rule, and each column in that row is either a condition or action for that rule.



	B	C	D	E	F	G
16	Type of New Claim	Is case catastrophic	Allocation code		Insurance Class	Date of accident is after
17	Catastrophic Claim	Y				
18	New Claim with previous Accident num		2			
21	Dependency Claim					
22	Interstate Claim					
23	Interstate Claim					
24	Interstate Claim					
25	Interstate Claim					

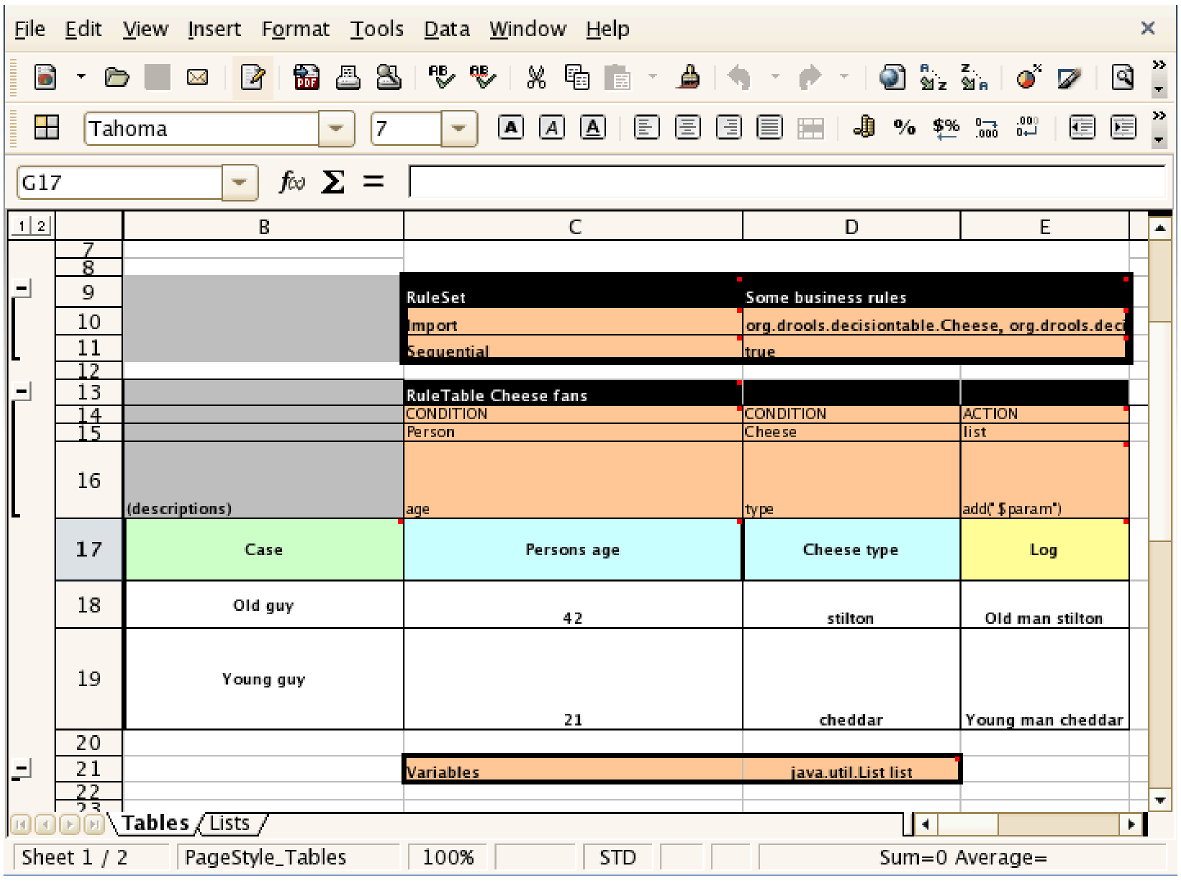
Each column may be a condition, or action etc.

Each row results in a rule

Figure 5.5. Rows and columns

The spreadsheet looks for the *RuleTable* keyword to indicate the start of a rule table (both the starting row and column). Other keywords are also used to define other package level attributes (covered later). It is important to keep the keywords in the one column. By convention the second column ("B") is used for this, but it can be any column (convention is to leave a margin on the left for notes). In the following diagram, C is actually the column where it starts. Everything to the left of this is ignored.

If we expand the hidden sections, it starts to make more sense how it works; note the keywords in column C.



	B	C	D	E
9		RuleSet	Some business rules	
10		Import	org.drools.decisiontable.Cheese, org.drools.deci	
11		Sequential	true	
13		RuleTable Cheese fans		
14		CONDITION	CONDITION	ACTION
15		Person	Cheese	list
16	(descriptions)	age	type	add("$param")
17	Case	Persons age	Cheese type	Log
18	Old guy	42	stilton	Old man stilton
19	Young guy	21	cheddar	Young man cheddar
21		Variables	java.util.List list	

Figure 5.6. Expanded for rule templates

Now the hidden magic which makes it work can be seen. The RuleSet keyword indicates the name to be used in the *rule package* that all the rules will come under (the name is optional, it will have a default but it MUST have the *RuleSet* keyword) in the cell immediately to the right.

The other keywords visible in Column C are: Import, Sequential which will be covered later. The RuleTable keyword is important as it indicates that a chunk of rules will follow, based on some rule templates. After the RuleTable keyword there is a name - this name is used to prefix the generated rules names (the row numbers are appended to create unique rule names). The column of RuleTable indicates the column in which the rules start (columns to the left are ignored).

Note

In general the keywords make up name/value pairs.

Referring to row 14 (the row immediately after RuleTable): the keywords CONDITION and ACTION indicate that the data in the columns below are for either the LHS or the RHS parts of a rule. There are other attributes on the rule which can also be optionally set this way.

Row 15 contains declarations of *ObjectTypes* ; the content in this row is optional (if this option is not in use, a blank row must be left, however this option is usually found to be quite useful). When using this row, the values in the cells below (row 16) become constraints on that object type. In the above case, it will generate: Person(age=="42") etc (where 42 comes from row 18). In the above example, the "==" is implicit (if just a field name is given it will assume that it is to look for exact matches).

Note

An ObjectType declaration can span columns (via merged cells), meaning that all columns below the merged range will be combined into the one set of constraints.

Row 16 contains the rule templates themselves. They can use the "$para" place holder to indicate where data from the cells below will be populated ($param can be sued or $1, $2 etc to indicate parameters from a comma separated list in a cell below). Row 17 is ignored as it is textual descriptions of the rule template.

Row 18 to 19 shows data, which will be combined (interpolated) with the templates in row 15, to generate rules. If a cell contains no data, then its template is ignored (e.g. it means that condition, or action, does not apply for that rule-row). Rule rows are read until there is a BLANK row. Multiple RuleTables can exsist in a sheet. Row 20 contains another keyword, and a value. The row positions of keywords like this do not matter (most people put them at the top) but their column should be the same one where the RuleTable or RuleSet keywords should appear (in this case column C has been chosen to be significant, but column A could be used instead).

In the above example, rules would be rendered like the following (as it uses the "ObjectType" row):

```
//row 18
rule "Cheese_fans_18"
when
    Person(age=="42")
    Cheese(type=="stilton")
then
    list.add("Old man stilton");
end
```

Note

The [age=="42"] and [type=="stilton"] are interpreted as single constraints to be added to the respective ObjectType in the cell above (if the cells above were spanned, then there could be multiple constraints on one "column".

5.1.4. Keywords and syntax

5.1.4.1. Syntax of templates

The syntax of what goes in the templates is dependent on if it is a CONDITION column or ACTION column. In most cases, it is identical to *vanilla* DRL for the LHS or RHS respectively. This means in the LHS, the constraint language must be used, and in the RHS it is a snippet of code to be executed.

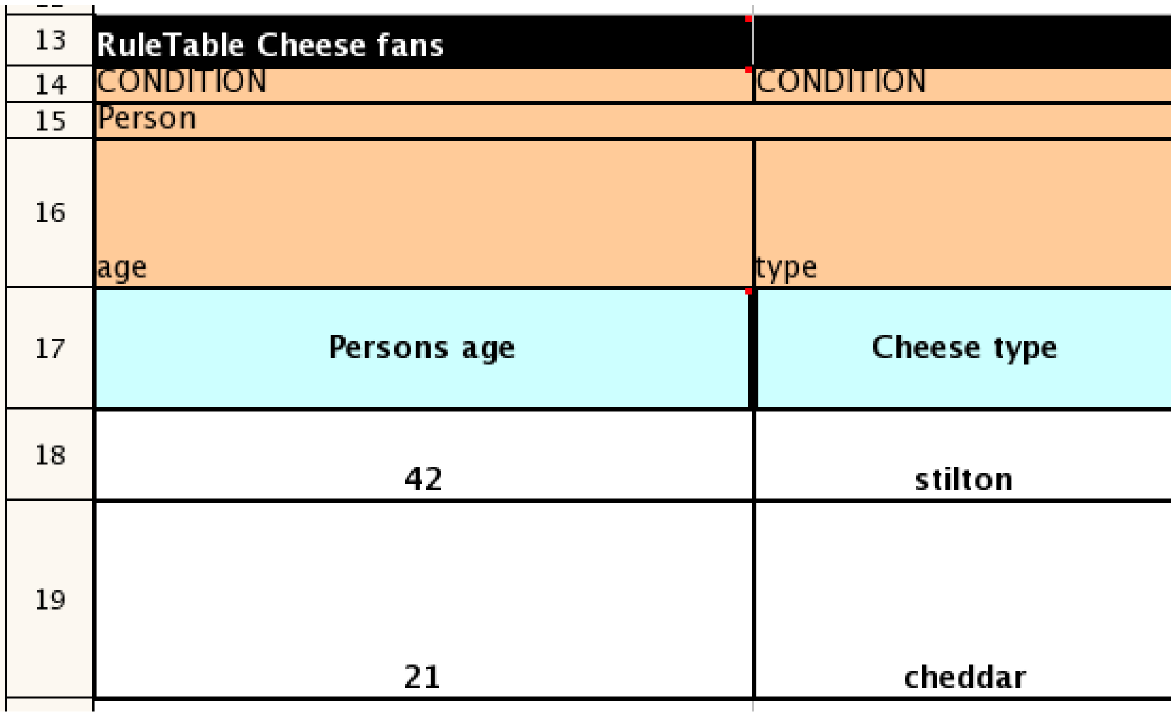
The `$param` place holder is used in templates to indicate where data form the cell will be interpolated. You can also use `$1` to the same effect. If the cell contains a comma separated list of values, $1 and $2 etc. may be used to indicate which positional parameter from the list of values in the cell will be used.

If the templates is [Foo(bar == $param)] and the cell is [42] then the result will be [Foo(bar == 42)] If the template is [Foo(bar < $1, baz == $2)] and the cell is [42,42] then the result will be [Foo(bar > 42, baz ==42)]

For conditions: How snippets are rendered depends on if there is anything in the row above (where ObjectType declarations may appear). If there is, then the snippets are rendered as individual constraints on that ObjectType. If there isn't, then they are just rendered as is (with values substituted). If just a plain field is entered (as in the example above) then it will assume this means equality. If another operator is placed at the end of the snippet, then the values will put interpolated at the end of the constraint, otherwise it will look for `$param` as outlined previously.

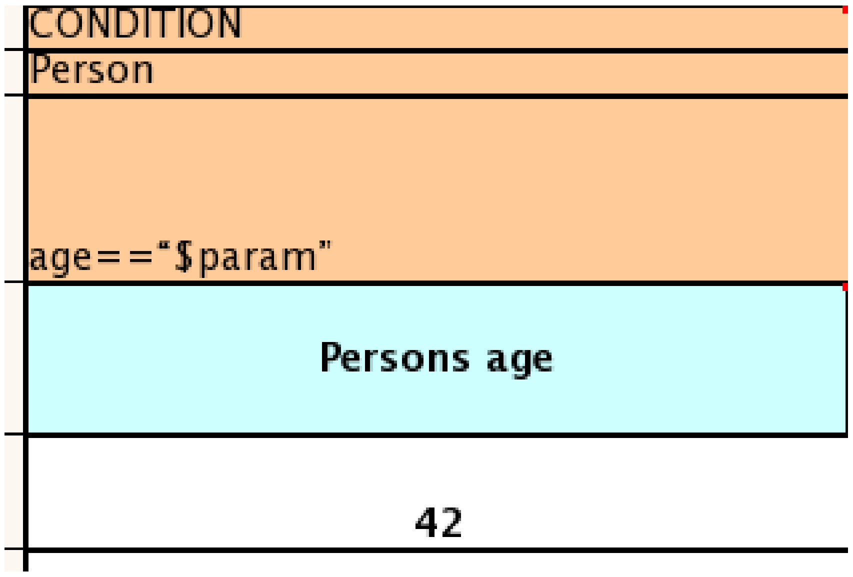
For consequences: How snippets are rendered also depends on if there is anything in the row immediately above it. If there is nothing there, the output is simple the interpolated snippets. If there is something there (which would typically be a bound variable or a global like in the example above) then it will append it as a method call on that object (refer to the above example).

This may be easiest to understand with some examples below.



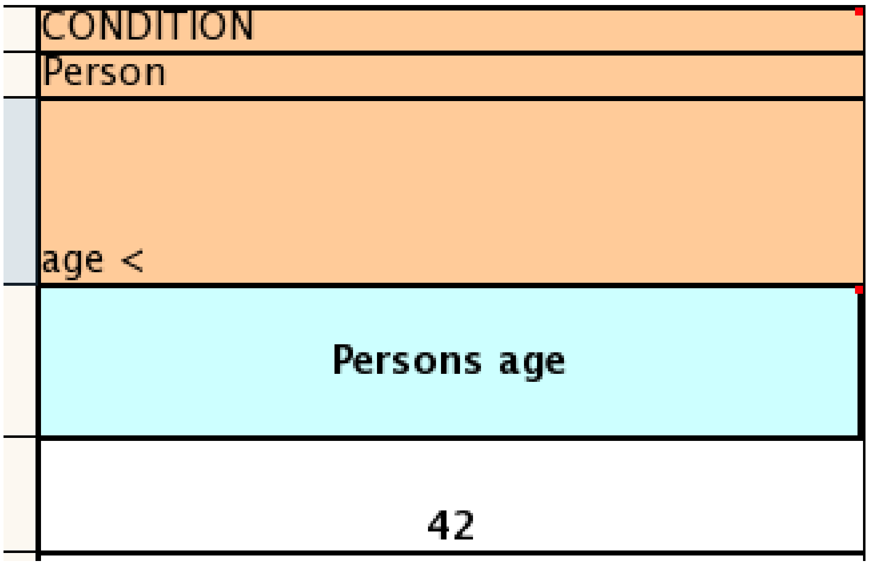
13	RuleTable Cheese fans	
14	CONDITION	CONDITION
15	Person	
16	age	type
17	Persons age	Cheese type
18	42	stilton
19	21	cheddar

The above shows how the Person ObjectType declaration spans 2 columns in the spreadsheet, thus both constraints will appear as Person(age == ... , type == ...). As before, as only the field names are present in the snippet, they imply an equality test.



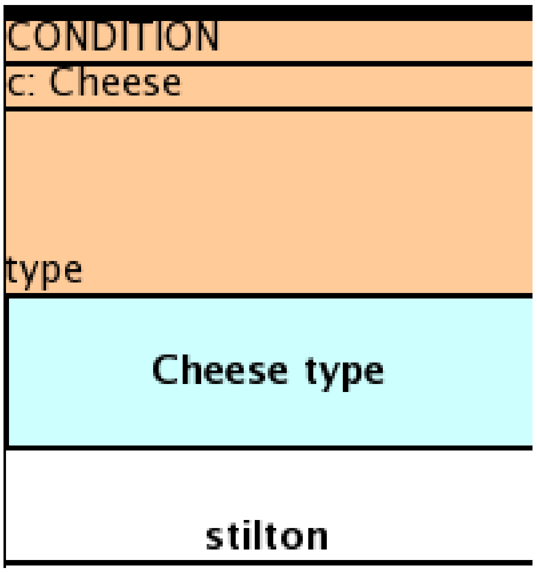
CONDITION
Person
age==“$param”
Persons age
42

The above condition example shows how you use interpolation to place the values in the snippet (in this case it would result in Person(age == "42")).



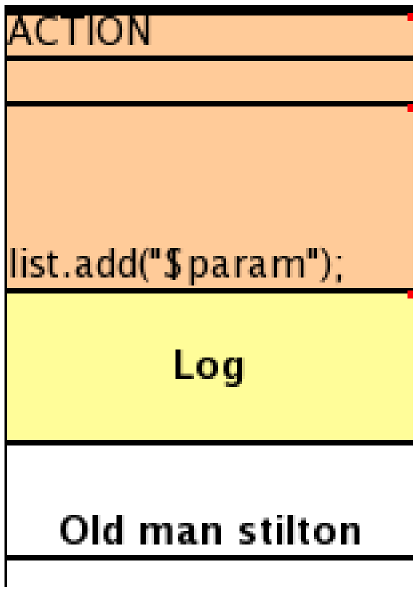
CONDITION
Person
age <
Persons age
42

The above condition example show that if you put an operator on the end by itself, the values will be placed after the operator automatically.



CONDITION
c: Cheese
type
Cheese type
stilton

A binding can be put in before the column (the constraints will be added from the cells below). Anything can be placed in the ObjectType row (e.g. it could be a pre condition for the columns in the spreadsheet columns that follow).


ACTION

list.add("$param");

Log

Old man stilton

This shows how the consequence could be done the by simple interpolation (just leave the cell above blank, the same applies to condition columns). With this style anything can be placed in the consequence (not just one method call).

5.1.4.2. Keywords

The following table describes the keywords that are pertinent to the rule table structure.

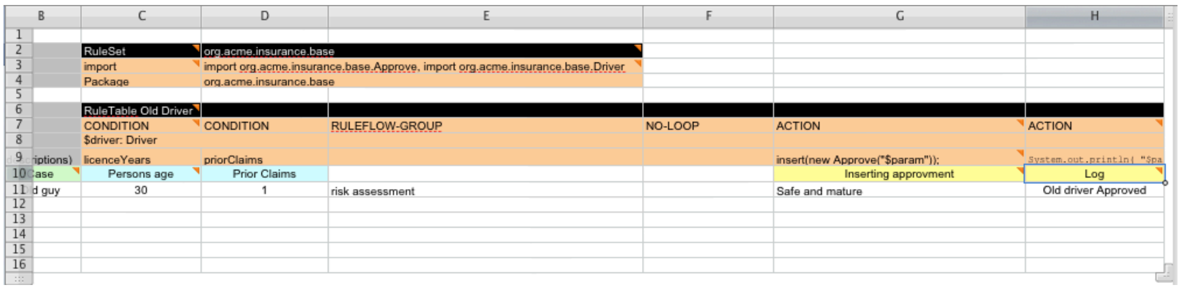
Keyword	Description	Inclusion Status
RuleSet	The cell to the right of this contains the ruleset name	One only (if left out, it will default)
Sequential	The cell to the right of this can be true or false. If true, then salience is used to ensure that rules fire from the top down	optional
Import	The cell to the right contains a comma separated list of Java classes to import	optional
RuleTable	A cell starting with RuleTable indicates the start of a definition of a rule table. The actual rule table starts the next row down. The rule table is read left-to-right, and top-down, until there is one BLANK ROW.	at least one. if there are more, then they are all added to the one ruleset
CONDITION	Indicates that this column will be for rule conditions	At least one per rule table
ACTION	Indicates that this column will be for rule consequences	At least one per rule table
PRIORITY	Indicates that this columns values will set the 'salience'	optional

Keyword	Description	Inclusion Status
	values for the rule row. Over-rides the 'Sequential' flag.	
DURATION	Indicates that this columns values will set the duration values for the rule row.	optional
NAME	Indicates that this columns values will set the name for the rule generated from that row	optional
Functions	The cell immediately to the right can contain functions which can be used in the rule snippets. Drools supports functions defined in the DRL, allowing logic to be embedded in the rule, and changed without hard coding, use with care. Same syntax as regular DRL.	optional
Variables	The cell immediately to the right can contain global declarations which Drools supports. This is a type, followed by a variable name. (if multiple variables are needed, comma separate them).	optional
No-loop or Unloop	Placed in the header of a table, no-loop or unloop will both complete the same function of not allowing a rule (row) to loop. For this option to function correctly, there must be a value (true or false) in the cell for the option to take effect. If the cell is left blank then this option will not be set for the row.	optional
XOR-GROUP	Cell values in this column mean that the rule-row belongs to the given XOR/activation group . An Activation group means that only one rule in the named group will fire (ie the first one to fire cancels the other rules activations).	optional
AGENDA-GROUP	Cell values in this column mean that the rule-row belongs to the given agenda group (that is one way of controlling flow between	optional

Keyword	Description	Inclusion Status
	groups of rules - see also "rule flow").	
RULEFLOW-GROUP	Cell values in this column mean that the rule-row belongs to the given rule-flow group.	optional
Worksheet	By default, the first worksheet is only looked at for decision tables.	N/A

Table 5.1. Keywords

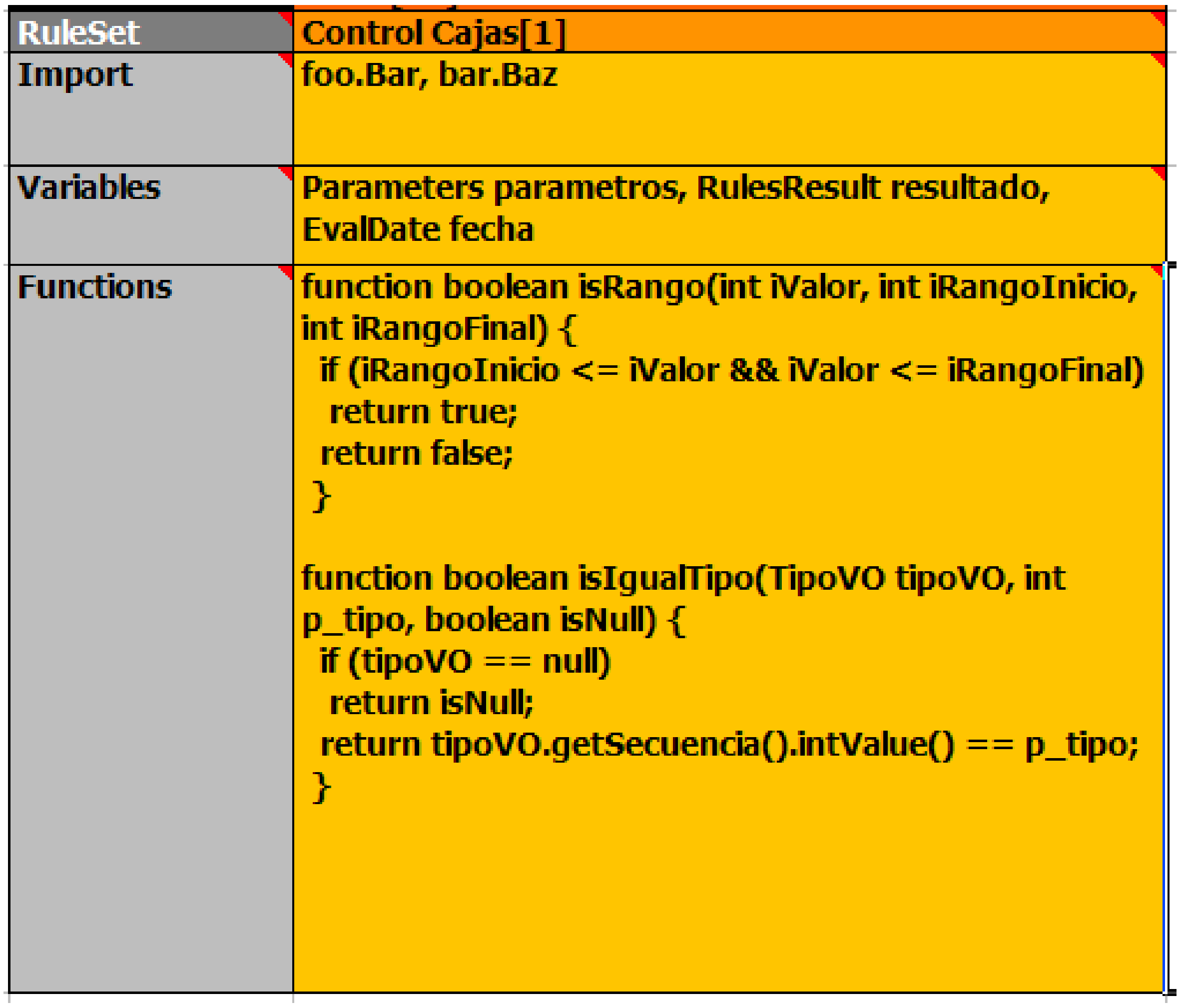
Below you will find examples of using the HEADER keywords, which effects the rules generated for each row. Note that the header name is what is important in most cases. If no value appears in the cells below it, then the attribute will not apply (it will be ignored) for that specific row.



	B	C	D	E	F	G	H
1							
2		RuleSet	org.acme.insurance.base				
3		import	import org.acme.insurance.base.Approve, import org.acme.insurance.base.Driver				
4		Package	org.acme.insurance.base				
5							
6		RuleTable Old Driver					
7		CONDITION	CONDITION	RULEFLOW-GROUP	NO-LOOP	ACTION	ACTION
8		$driver: Driver					
9	iptions)	licenceYears	priorClaims			insert(new Approve("$param"));	System.out.println("$pa
10	ase	Persons age	Prior Claims			Inserting approvment	Log
11	d guy	30	1	risk assessment		Safe and mature	Old driver Approved
12							
13							
14							
15							
16							

Figure 5.7. Example usage of keywords for imports, headers etc

The following is an example of Import (comma delimited), Variables (gloabls) - also comma delimited, and a function block (can be multiple functions - just the usual drl syntax). This can appear in the same column as the "RuleSet" keyword, and can be below all the rule rows if you desire.



RuleSet	Control Cajas[1]
Import	foo.Bar, bar.Baz
Variables	Parameters parametros, RulesResult resultado, EvalDate fecha
Functions	function boolean isRango(int iValor, int iRangoInicio, int iRangoFinal) { if (iRangoInicio <= iValor && iValor <= iRangoFinal) return true; return false; } function boolean isIgualTipo(TipoVO tipoVO, int p_tipo, boolean isNull) { if (tipoVO == null) return isNull; return tipoVO.getSecuencia().intValue() == p_tipo; }

Figure 5.8. Example usage of keywords for functions etc.

5.1.5. Creating and integrating Spreadsheet based Decision Tables

The API to use spreadsheet based decision tables is in the drools-decisiontables module. There is really only one class to look at: SpreadsheetCompiler. This class will take spreadsheets in various formats, and generate rules in DRL (which you can then use in the normal way). The SpreadsheetComiler can just be used to generate partial rule files if it is wished, and assemble it into a complete rule package after the fact (this allows the separation of technical and non-technical aspects of the rules if needed).

To get started, a sample spreadsheet can be used as base. Alternatively, if the plug-in is being used (Rule Workbench IDE), the wizard can generate a spreadsheet from a template (to edit it an xls compatible spreadsheet editor will need to be used).

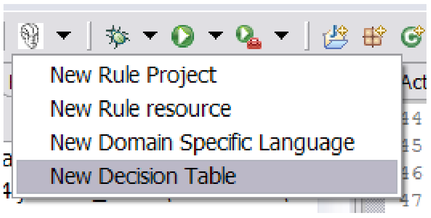


Figure 5.9. Wizard in the IDE

5.1.6. Managing business rules in decision tables.

5.1.6.1. Workflow and collaboration.

Spreadsheets are well established business tools (in use for over 25 years). Decision tables lend themselves to close collaboration between IT and domain experts, while making the business rules clear to business analysts, it is an ideal separation of concerns.

Typically, the whole process of authoring rules (coming up with a new decision table) would be something like:

1. Business analyst takes a template decision table (from a repository, or from IT)
2. Decision table business language descriptions are entered in the table(s)
3. Decision table rules (rows) are entered (roughly)
4. Decision table is handed to a technical resource, who maps the business language (descriptions) to scripts (this may involve software development of course, if it is a new application or data model)
5. Technical person hands back and reviews the modifications with the business analyst.
6. The business analyst can continue editing the rule rows as needed (moving columns around is also fine etc).
7. In parallel, the technical person can develop test cases for the rules (liaising with business analysts) as these test cases can be used to verify rules and rule changes once the system is running.

5.1.6.2. Using spreadsheet features

Features of applications like Excel can be used to provide assistance in entering data into spreadsheets, such as validating fields. Lists that are stored in other worksheets can be used to provide valid lists of values for cells, like in the following diagram.

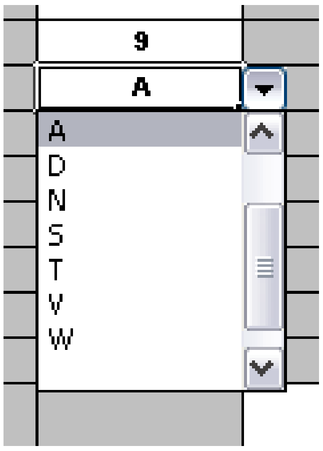


Figure 5.10. Wizard in the IDE

Some applications provide a limited ability to keep a history of changes, but it is recommended that an alternative means of revision control is also used. When changes are being made to rules over time, older versions are archived (many solutions exist for this which are also open source, such as Subversion). *http://www.drools.org/Business+rules+in+decision+tables+explained*

5.1.7. Rule Templates

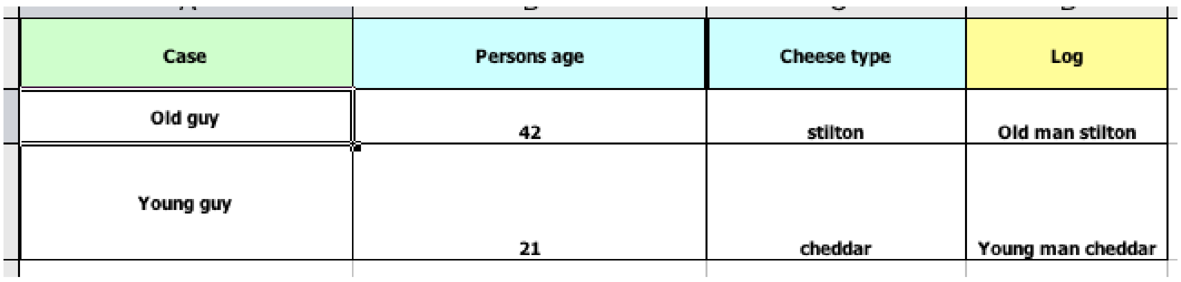
Related to decision tables (but not necessarily requiring a spreadsheet) are "Rule Templates" (in the drools-templates module). These use any tabular data source as a source of rule data - populating a template to generate many rules. This can allow both for more flexible spreadsheets, but also rules in existing databases, for instance (at the cost of developing the template up front to generate the rules).

With Rule Templates the data is separated from the rule and there are no restrictions on which part of the rule is data-driven. So whilst you can do everything you could do in decision tables you can also do the following:

- store your data in a database (or any other format)
- conditionally generate rules based on the values in the data
- use data for any part of your rules (e.g. condition operator, class name, property name)
- run different templates over the same data

5.1.7.1. A decision table-like example

As an example, a more classic decision table is shown, but without any hidden rows for the rule meta data (so the spreadsheet only contains the raw data to generate the rules).



Case	Persons age	Cheese type	Log
Old guy	42	stilton	Old man stilton
Young guy	21	cheddar	Young man cheddar

Figure 5.11. Template data

See the "ExampleCheese.xls" in the examples download for the above spreadsheet.

If this was a regular decision table there would be hidden rows before row 1 and between rows 1 and 2 containing rule metadata. With rule templates the data is completely separate from the rules. This has two handy consequences - you can apply multiple rule templates to the same data and your data is not tied to your rules at all. So what does the template look like?

```
template header
age
type
log
package org.drools.examples.templates;

global java.util.List list;

template "cheesefans"
```

```
rule "Cheese fans_@{row.rowNumber}"
when
    Person(age == @{age})
    Cheese(type == "@{type}")
then
    list.add("@{log}");
end

end template
```

Referring to the above:

Line 1
: all rule templates start with "template header"

Lines 2-4
: following the header is the list of columns in the order they appear in the data. In this case we are calling the first column "age", the second "type" and the third "log".

Lines 5
: empty line signifying the end of the column definitions

Lines 6-9
: standard rule header text. This is standard rule DRL and will appear at the top of the generated DRL. Put the package statement and any imports and global definitions

Line 10
: The "template" keyword signals the start of a rule template. There can be more than one template in a template file. The template should have a unique name.

Lines 11-18
: The rule template - see below

Line 20
: "end template" signifies the end of the template.

The rule templates rely on MVEL to do substitution using the syntax @{token_name}. There is currently one built-in expression, @{row.rowNumber} which gives a unique number for each row of data and enables you to generate unique rule names. For each row of data a rule will be generated with the values in the data substituted for the tokens in the template. With the example data above the following rule file would be generated:

```
package org.drools.examples.templates;

global java.util.List list;

rule "Cheese fans_1"
when
    Person(age == 42)
    Cheese(type == "stilton")
then
```

```
    list.add("Old man stilton");
end

rule "Cheese fans_2"
when
    Person(age == 21)
    Cheese(type == "cheddar")
then
    list.add("Young man cheddar");
end
```

The code to run this is simple:

```
KnowledgeBuilder kbuilder = KnowledgeBuilderFactory.newKnowledgeBuilder();
DecisionTableConfiguration dtconf =
    KnowledgeBuilderFactory.newDecisionTableConfiguration();
dtconf.setInputType( DecisionTableInputType.XLS );
dtconf.setWorksheetName( "Tables_2" );
kbuilder.add( ResourceFactory.newInputStreamResource(
    getSpreadsheetStream() ),
    ResourceType.DTABLE,
    dtconf );

Collection<KnowlegePackage> kpkg = kbuilder.getKnowlegePackages();
```

We create an **`ExternalSpreadsheetCompiler`** object and use it to merge the spreadsheet with the rules. The two integer parameters indicate the column and row where the data actually starts - in our case column 2, row 2 (i.e. B2)

The Java Rule Engine API

6.1. Introduction

JBoss Rules provides an implementation of JSR94, the Java Rule Engine API, which allows for support of multiple rule engines from a single API. JSR94 does not deal in anyway with the rule language itself. W3C is working on the Rule Interchange Format (RIF) *http://www.w3.org/TR/2006/WD-rif-ucr-20060323* and the OMG has started to work on a standard based on RuleML, *http://ruleml.org* Haley Systems has also proposed a rule language standard called RML.

It should be remembered that the JSR94 standard represents the lowest common denominator in features across rule engines. This means there is less functionality in the JSR94 API than in the standard JBoss Rules API. So by using JSR94 you are restricting yourself in taking advantage of using the full capabilities of the JBoss Rule Engine. It is necessary to expose further functionality, like globals and support for drl, dsl and xml via properties maps due to the very basic feature set of JSR94. This introduces non portable functionality. Further to this, as JSR94 does not provide a rule language, you are only solving a small fraction of the complexity of switching rule engines with very little gain. So while we support JSR94, for those that insist on using it, we strongly recommend you program against the JBoss Rules API.

6.2. How To Use

There are two parts to working with JSR94. The first part is the administrative api that deals with building and register RuleExecutionSets, the second part is runtime session execution of those RuleExecutionSets.

6.2.1. Building and Registering RuleExecutionSets

The RuleServiceProviderManager manages the registration and retrieval of RuleServiceProviders. The Drools RuleServiceProvider implementation is automatically registered via a static block when the class is loaded using Class.forName; in much the same way as JDBC drivers.

```


// RuleServiceProviderImpl is registered to "http://drools.org/"
// via a static initialization block
Class.forName("org.drools.jsr94.rules.RuleServiceProviderImpl");

// Get the rule service provider from the provider manager.
RuleServiceProvider ruleServiceProvider =
RuleServiceProviderManager.getRuleServiceProvider("http://drools.org/");
```

Example 6.1. Automatic RuleServiceProvider Registration

The RuleServiceProvider provides access to the RuleRuntime and RuleAdministration APIs. The RuleAdministration provides an administration API for the management of RuleExecutionSets, making it possible to register a RuleExecutionSet that can then be retrieved via the RuleRuntime.

First you need to create a RuleExecutionSet before it can be registered; RuleAdministrator provides factory methods to return an empty LocalRuleExecutionSetProvider or RuleExecutionSetProvider. The LocalRuleExecutionSetProvider should be used to load a RuleExecutionSets from local sources that are not serializable, like Streams. The RuleExecutionSetProvider can be used to load RuleExecutionSets from serializable sources, like DOM Elements or Packages. Both the "ruleAdministrator.getLocalRuleExecutionSetProvider(null);" and the "ruleAdministrator.getRuleExecutionSetProvider(null);" take null as a parameter, as the properties map for these methods is not currently used.

```
// Get the RuleAdministration
RuleAdministrator ruleAdministrator =
    ruleServiceProvider.getRuleAdministrator();
LocalRuleExecutionSetProvider ruleExecutionSetProvider =
    ruleAdministrator.getLocalRuleExecutionSetProvider( null );

// Create a Reader for the drl
URL drlUrl = new URL("http://mydomain.org/sources/myrules.drl");
Reader drlReader = new InputStreamReader(  drlUrl.openStream()  );

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

Example 6.2. Registering a LocalRuleExecutionSet with the RuleAdministration API

"ruleExecutionSetProvider.createRuleExecutionSet(reader, null)" in the above example takes a null parameter for the properties map; however it can actually be used to provide configuration for the incoming source. When null is passed the default is used to load the input as a drl. Allowed keys for a map are "source" and "dsl". "source" takes "drl" or "xml" as its value; set "source" to "drl" to load a drl or to "xml" to load an xml source; xml will ignore any "dsl" key/value settings. The "dsl" key can take a Reader or a String (the contents of the dsl) as a value.

```
// Get the RuleAdministration
RuleAdministration ruleAdministrator =
    ruleServiceProvider.getRuleAdministrator();
LocalRuleExecutionSetProvider ruleExecutionSetProvider =
    ruleAdministrator.getLocalRuleExecutionSetProvider( null );

// Create a Reader for the drl
URL drlUrl = new URL("http://mydomain.org/sources/myrules.drl");
Reader drlReader = new InputStreamReader(  drlUrl.openStream()  );

// Create a Reader for the dsl and a put in the properties map
URL dslUrl = new URL("http://mydomain.org/sources/myrules.dsl");
Reader dslReader = new InputStreamReader( dslUrl.openStream()  );
Map properties = new HashMap();
properties.put( "source", "drl" );
properties.put( "dsl", dslReader );

// Create the RuleExecutionSet for the drl and dsl
RuleExecutionSet ruleExecutionSet =
    ruleExecutionSetProvider.createRuleExecutionSet( reader, properties );
```

Example 6.3. Specifying a DSL when registering a LocalRuleExecutionSet

When registering a RuleExecutionSet you must specify the name, to be used for its retrieval. There is also a field to pass properties, this is currently unused so just pass null.

```
// Register the RuleExecutionSet with the RuleAdministrator
String uri = ruleExecutionSet.getName();
ruleAdministrator.registerRuleExecutionSet(uri, ruleExecutionSet, null);
```

Example 6.4. Register the RuleExecutionSet

6.2.2. Using Stateful and Stateless RuleSessions

The Runtime, obtained from the RuleServiceProvider, is used to create stateful and stateless rule engine sessions.

```
RuleRuntime ruleRuntime = ruleServiceProvider.getRuleRuntime();
```

Example 6.5. Getting the RuleRuntime

To create a rule session you must use one of the two RuleRuntime public constants - "RuleRuntime.STATEFUL_SESSION_TYPE" and "RuleRuntime.STATELESS_SESSION_TYPE" along with the uri to the RuleExecutionSet you wish to instantiate a RuleSession for. The properties map can be null, or it can be used to specify globals, as shown in the next section. The createRuleSession(....) method returns a RuleSession instance which must then be cast to StatefulRuleSession or StatelessRuleSession.

```
(StatefulRuleSession) session = ruleRuntime.createRuleSession(
    uri, null, RuleRuntime.STATEFUL_SESSION_TYPE );
session.addObject( new PurchaseOrder( "lots of cheese" ) );
session.executeRules();
```

Example 6.6. Stateful Rule

The StatelessRuleSession has a very simple API; you can only call executeRules(List list) passing a list of objects, and an optional filter, the resulting objects are then returned.

```
(StatelessRuleSession) session = ruleRuntime.createRuleSession(
    uri, null, RuleRuntime.STATELESS_SESSION_TYPE );
List list = new ArrayList();
list.add( new PurchaseOrder( "even more cheese" ) );

List results = new ArrayList();
results = session.executeRules( list );
```

Example 6.7. Stateless

6.2.2.1. Globals

It is possible to support globals with JSR94, in a none portable manner, by using the properties map passed to the RuleSession factory method. Globals must be defined in the drl or xml file first, otherwise an Exception will be thrown. the key represents the identifier declared in the drl or xml and the value is the instance you wish to be used in the execution. In the following example the results are collected in an java.util.List which is used as global:

```
java.util.List globalList = new java.util.ArrayList( );
java.util.Map map = new java.util.HashMap( );
map.put( "list", globalList );
//Open a stateless Session StatelessRuleSession srs =
//    (StatelessRuleSession) runtime.createRuleSession(
//    "SistersRules", map, RuleRuntime.STATELESS_SESSION_TYPE );
...
// Persons added to List
// call executeRules( ) giving a List of Objects as parameter
// There are rules which will put Objects in the List
// fetch the list from the map
List list = (java.util.List) map.get("list");
```

Do not forget to declare the global "list" in your DRL:

```
package SistersRules;
import org.drools.jsr94.rules.Person;
global java.util.List list
rule FindSisters
when
    $person1 : Person ( $name1:name )
    $person2 : Person ( $name2:name )
    eval( $person1.hasSister($person2) )
then
    list.add($person1.getName()+" and "+$person2.getName()+" are sisters");
    assert( $person1.getName()+" and "+$person2.getName()+" are sisters");
end
```

6.3. References

If you need more information on JSR 94, please refer to the following references

1. Official JCP Specification for Java Rule Engine API (JSR 94)
   - *http://www.jcp.org/en/jsr/detail?id=94*
2. The Java Rule Engine API documentation
   - *http://www.javarules.org/api_doc/api/index.html*
3. The Logic From The Bottom Line: An Introduction to The Drools Project. By N. Alex Rupp, published on TheServiceSide.com in 2004
   - *http://www.theserverside.com/articles/article.tss?l=Drools*
4. Getting Started With the Java Rule Engine API (JSR 94): Toward Rule-Based Applications. By Dr. Qusay H. Mahmoud, published on Sun Developer Network in 2005
   - *http://java.sun.com/developer/technicalArticles/J2SE/JavaRule.html*

5. Jess and the javax.rules API. By Ernest Friedman-Hill, published on TheServerSide.com in 2003

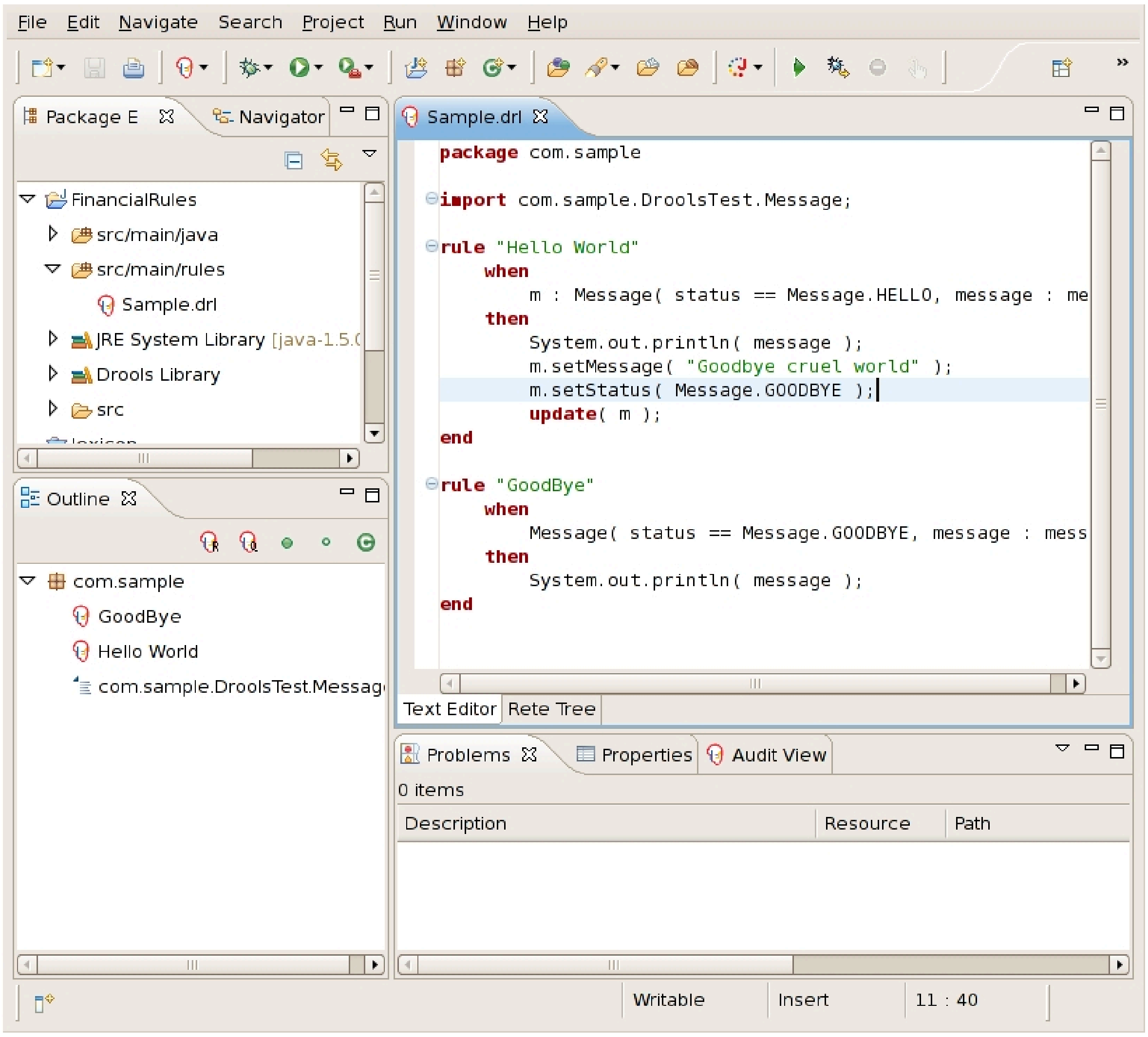
   - *http://www.theserverside.com/articles/article.tss?l=Jess*

The Rule IDE

The JBoss Developer Studio IDE provides developers with an environment to edit and test rules for JBoss Rules in various formats. The JBoss Rules IDE components are also available separately as an Eclipse plug-in. The use of the IDE is optional and not all the components are required.

Important

None of the underlying features of the JBoss Rules engine are dependent on Eclipse, and integrators are free to use their tools of choice.



```
package com.sample

import com.sample.DroolsTest.Message;

rule "Hello World"
    when
        m : Message( status == Message.HELLO, message : me
    then
        System.out.println( message );
        m.setMessage( "Goodbye cruel world" );
        m.setStatus( Message.GOODBYE );
        update( m );
end

rule "GoodBye"
    when
        Message( status == Message.GOODBYE, message : mess
    then
        System.out.println( message );
end
```

Figure 7.1. Overview

7.1. Features outline

The rules IDE has the following features

1. Textual/graphical rule editor that is aware of:

   a. DRL syntax, and provides content assistance (including an outline view)

   b. DSL (domain specific language) extensions, and provides content assistance.

2. Wizards to aid you to:

   a. quickly create a new "rules" project

   b. create a new rule resource,

   c. create a new Domain Specific language,

   d. create a new decision table, guided editor

3. A domain specific language editor

   a. Create and manage mappings from your users language to the rule language

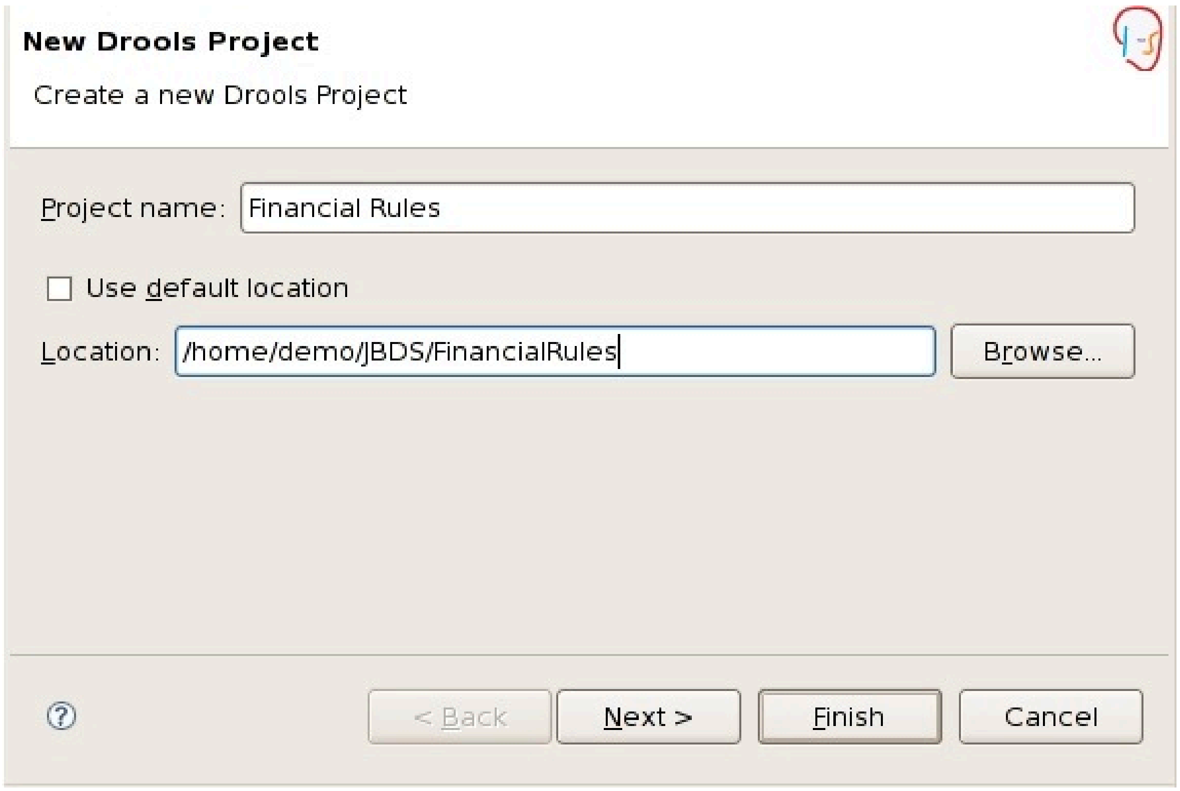
4. Rule validation

   a. As rules are entered, the rule is "built" in the background and errors reported via the problem "view" where possible

All of the power and flexibility of Eclipse is available.

7.2. Creating a Rule project

The aim of the new project wizard is to setup an executable scaffold project to start using rules immediately. This will setup a basic structure, classpath and sample rules and test case to get you started.


New Drools Project

Create a new Drools Project

Project name: Financial Rules

[ ] Use default location

Location: /home/demo/JBDS/FinancialRules

Figure 7.2. New rule project scaffolding

When you choose to create a new "rule project" - you will get a choice to add some default artifacts to it (like rules, decision tables, ruleflows etc). These can serve as a starting point, and will give you something executable to play with (which you can then modify and mould to your needs). The simplest case (a hello world rule) is shown below. Feel free to experiment with the plug-in at this point.

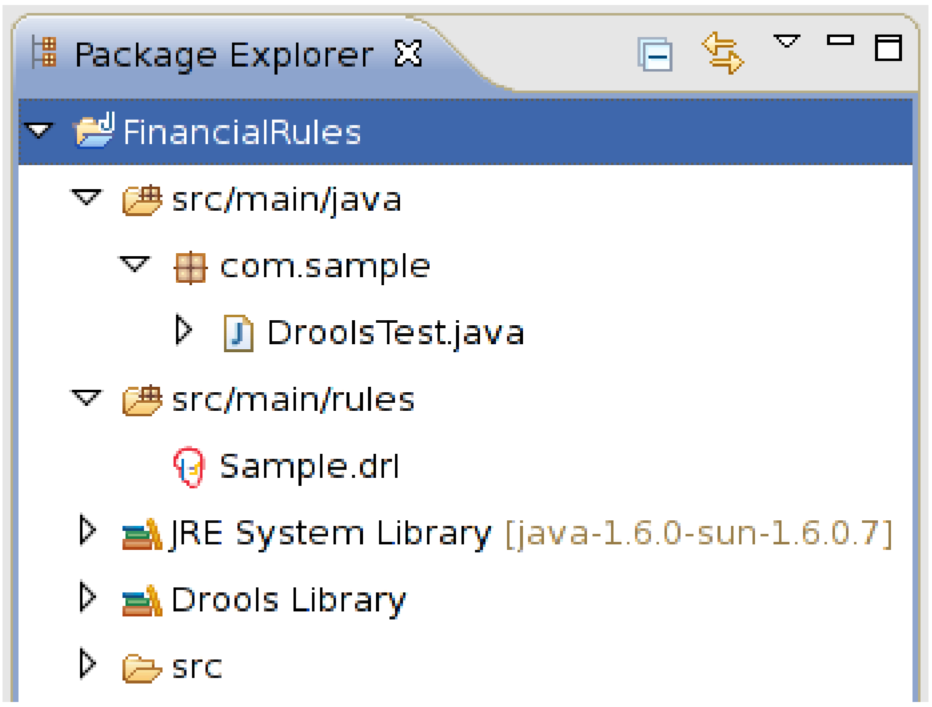


Figure 7.3. New rule project result

The newly created project contains an example rule file (Sample.drl) in the src/rules dir and an example Java file (DroolsTest.Java) that can be used to execute the rules in a Drools engine in the folder src/Java, in the com.sample package. All the others jars that are necessary during execution are also added to the classpath in a custom classpath container called Drools Library. Rules do not have to be kept in "Java" projects at all, this is just a convenience for people who are already using Eclipse as their Java IDE.

Important note: The Drools plug-in adds a "Drools Builder" capability to your Eclipse instance. This means you can enable a builder on any project that will build and validate your rules when resources change. This happens automatically with the Rule Project Wizard, but you can also enable it manually on any project. One downside of this is if you have rule files that have a large number of rules (>500 rules per file) it means that the background builder may be doing a lot of work to build the rules on each change. An option here is to turn off the builder, or put the large rules into .rule files, where you can still use the rule editor, but it won't build them in the background - to fully validate the rules you will need to run them in a unit test of course.

7.3. Creating a new rule and wizards

You can create a rule simple as an empty text ".drl" file, or use the wizard to do so. The wizard menu can be invoked by Control+N, or choosing it from the toolbar (there will be a menu with the JBoss Drools icon).

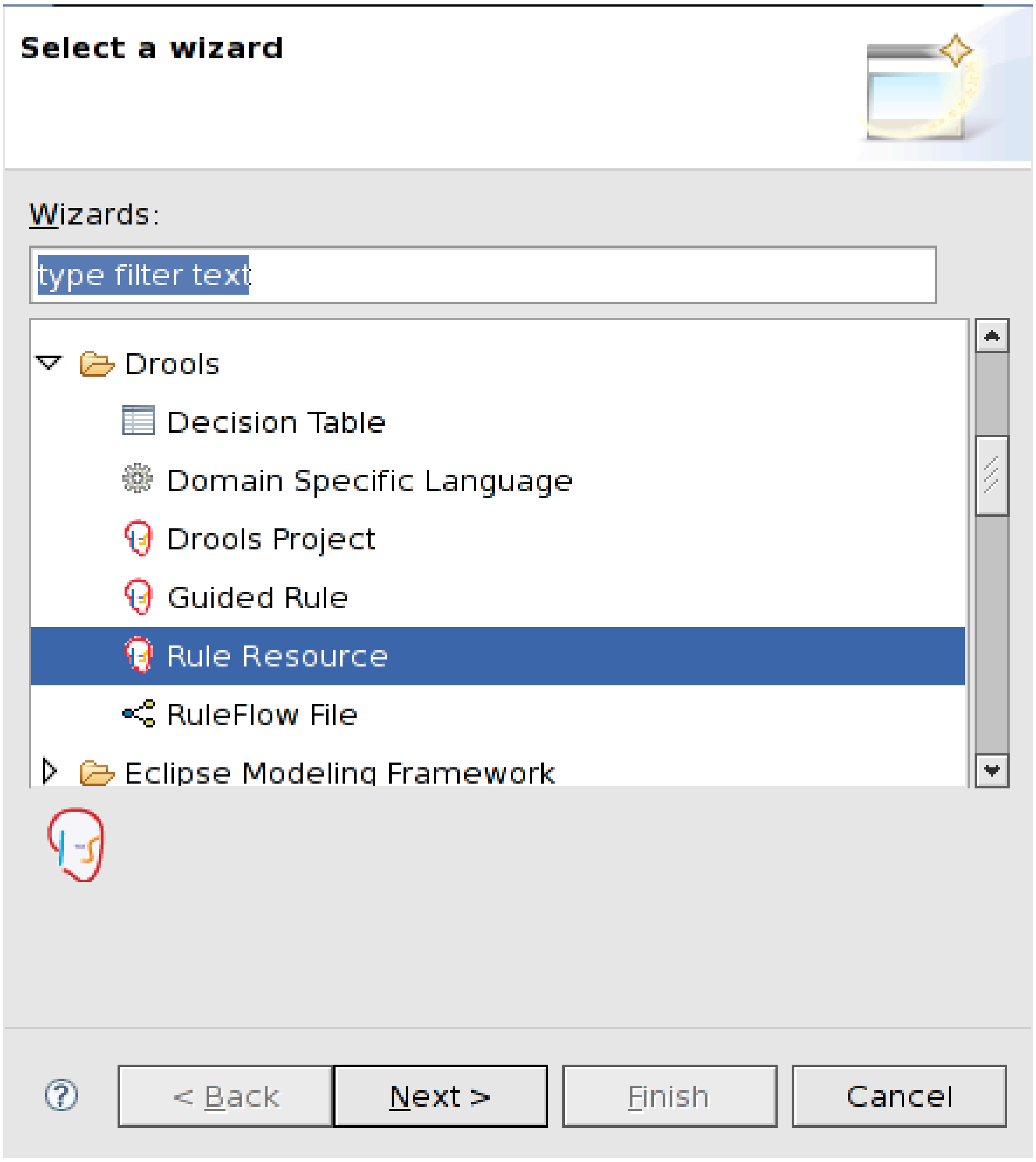
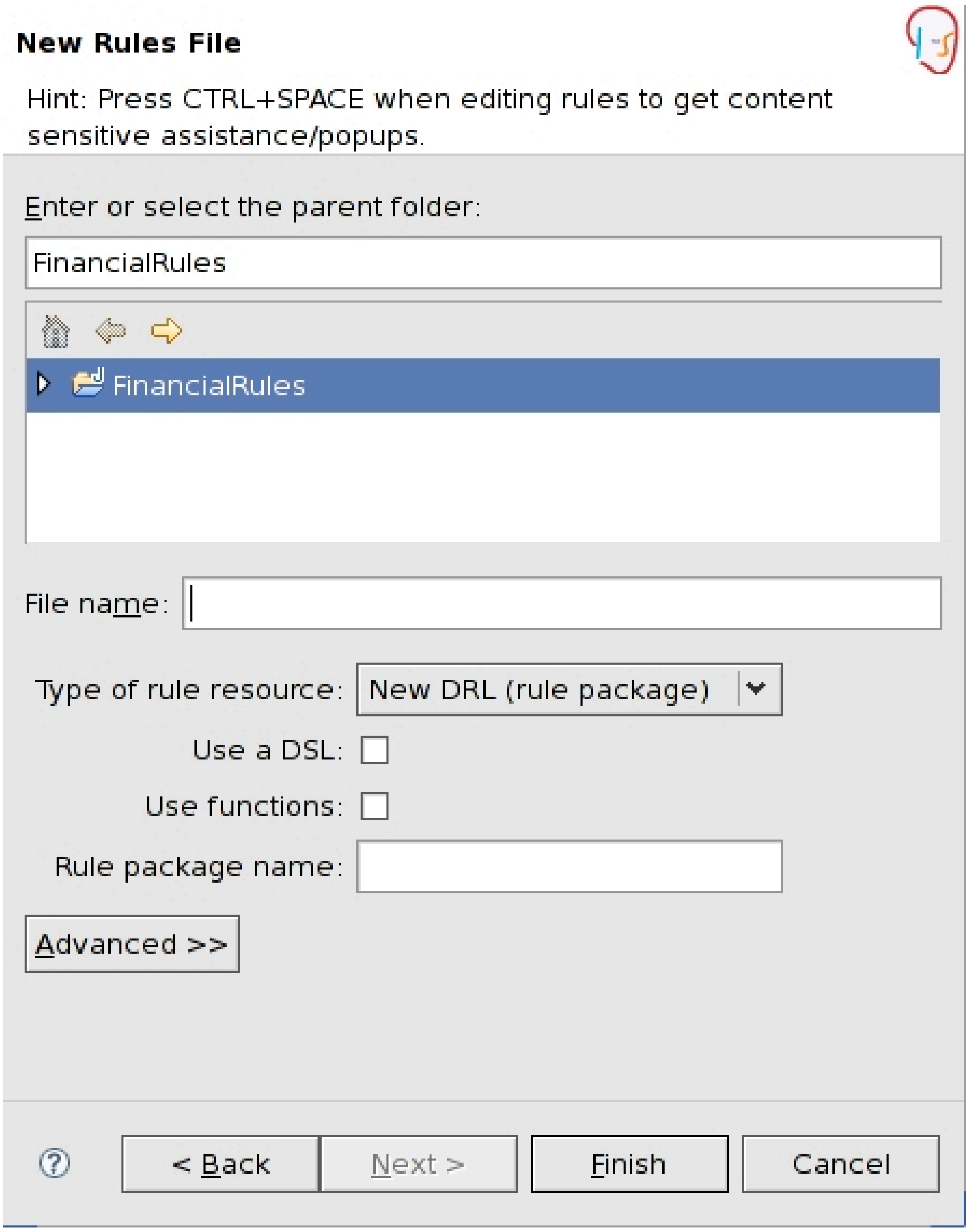


Figure 7.4. The wizard menu

The wizard will ask for some basic options for generating a rule resource. These are just hints, you can change your mind later !. In terms of location, typically you would create a top level /rules directory to store your rules if you are creating a rule project, and store it in a suitably named subdirectory. The package name is mandatory, and is similar to a package name in Java (ie. its a namespace that groups like rules together).



New Rules File

Hint: Press CTRL+SPACE when editing rules to get content sensitive assistance/popups.

Enter or select the parent folder: FinancialRules

File name:

Type of rule resource: New DRL (rule package)

Use a DSL: ☐

Use functions: ☐

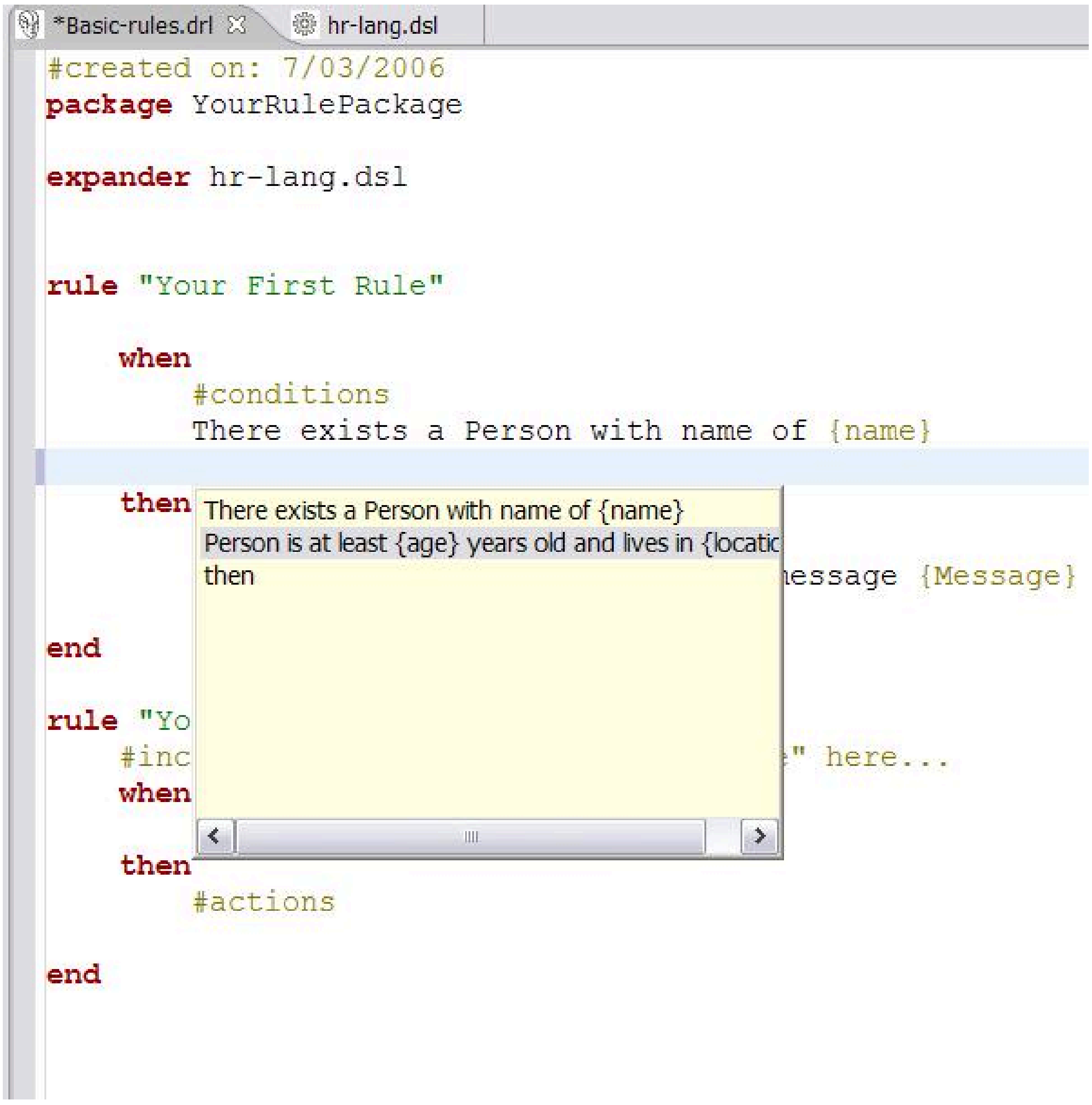
Rule package name:

Figure 7.5. New rule wizard

This result of this wizard is to generate a rule skeleton to work from.

7.4. Textual rule editor

The rule editor is where rule managers and developers will be spending most of their time. The rule editor follows the pattern of a normal text editor in Eclipse, with all the normal features of a text editor. On top of this, the rule editor provides pop up content assistance. You invoke popup content assistance the "normal" way by pressing Control + Space at the same time.



```
#created on: 7/03/2006
package YourRulePackage

expander hr-lang.dsl


rule "Your First Rule"

    when
        #conditions
        There exists a Person with name of {name}

    then                                           essage {Message}

end

rule "Yo
    #inc                                          " here...
    when

    then
        #actions

end
```

```
There exists a Person with name of {name}
Person is at least {age} years old and lives in {locatic
then
```

Figure 7.6. The rule editor in action

The rule editor works on files that have a .drl (or .rule) extension. Rules are generally grouped together as a "package" of rules (like the old ruleset construct). It will also be possible to have rules in individual files (grouped by being in the same package "namespace" if you like). These DRL files are plain text files.

You can see from the example above that the package is using a domain specific language (note the expander keyword, which tells the rule compiler to look for a dsl file of that name, to resolve the rule language). Even with the domain specific language (DSL) the rules are still stored as plain text as you see on screen, which allows simpler management of rules and versions (comparing versions of rules for instance).

The editor has an outline view that is kept in sync with the structure of the rules (updated on save). This provides a quick way of navigating around rules by name, in a file which may have hundreds of rules. The items are sorted alphabetically by default.

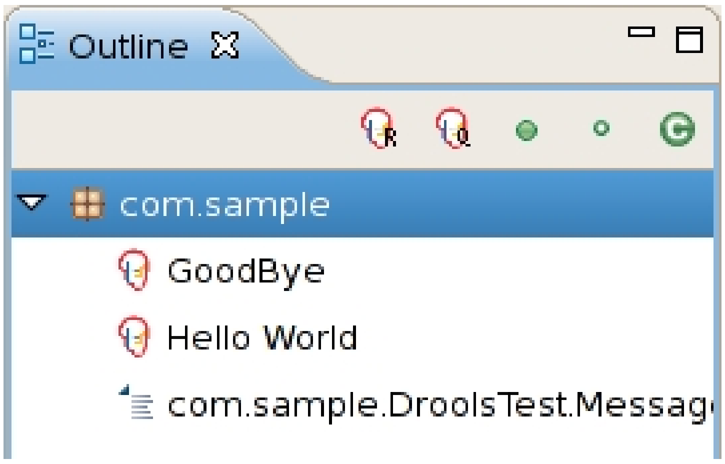


Figure 7.7. The rule outline view

7.5. Guided editor (rule GUI)

A new feature of the Drools IDE (since version 4) is the guided editor for rules. This is similar to the web based editor that is available in the BRMS. This allows you to build rules in a GUI driven fashion, based on your object model.

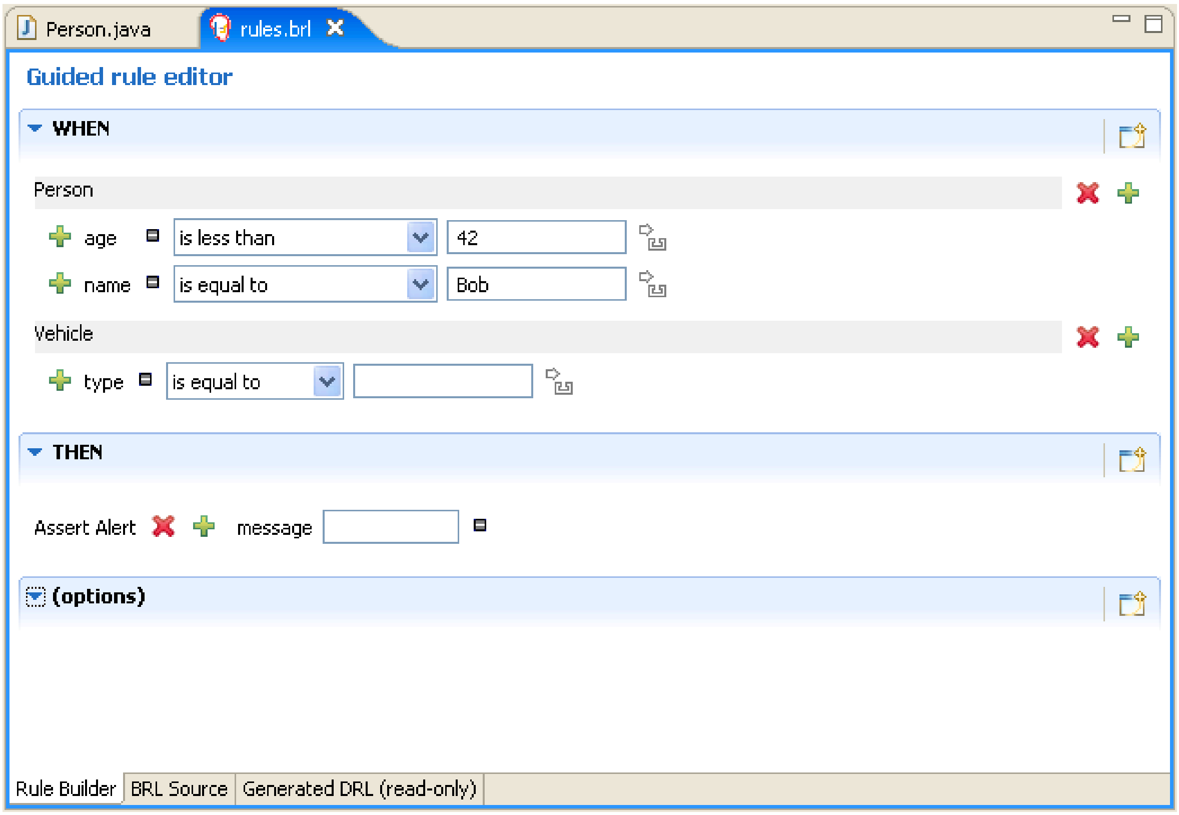


Figure 7.8. The guided editor

To create a rule this way, use the wizard menu. It will create a instance of a .brl file and open an editor. The guided editor works based on a .package file in the same directory as the .brl file. In this "package" file - you have the package name and import statements - just like you would in the top of a normal DRL file. So the first time you create a brl rule - you will need to populate the package file with the fact classes you are interested in. Once you have this the guided editor will be able to prompt you with facts/fields and build rules graphically.

The guided editor works off the model classes (fact classes) that you configure. It then is able to "render" to DRL the rule that you have entered graphically. You can do this visually - and use it as a basis for learning DRL, or you can use it and build rules of the brl directly. To do this, you can either use the drools-ant module (it is an ant task that will build up all the rule assets in a folder as a rule package - so you can then deploy it as a binary file), OR you can use the following snippet of code to convert the brl to a drl rule:

```
BRXMLPersitence read = BRXMLPersitence.getInstance();
BRDRLPersistence write = BRDRLPersistence.getInstance();
String brl = ... read from the .brl file as needed...
String outputDRL = write.marshall(read.unmarshal(brl));
//then pass the outputDRL to the PackageBuilder as normal
```

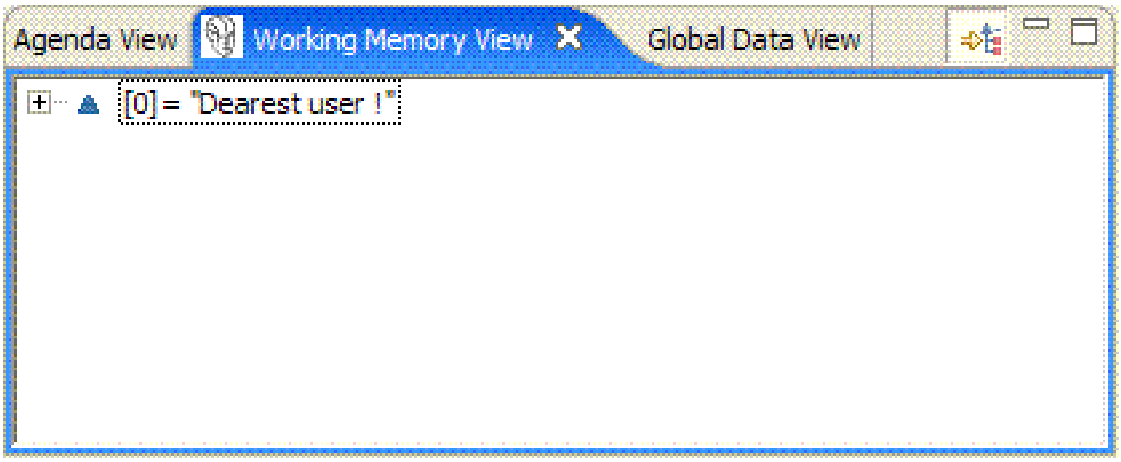
7.6. Views

When debugging an application using a Drools engine, these views can be used to check the state of the Drools engine itself: the Working Memory View, the Agenda View the Global Data View. To be able to use these views, create breakpoints in your code invoking the working memory. For example, the line where you call workingMemory.fireAllRules() is a good candidate. If the debugger halts at that joinpoint, you should select the working memory variable in the debug variables view. The following rules can then be used to show the details of the selected working memory:

1. The Working Memory shows all elements in the working memory of the Drools working memory.
2. The Agenda View shows all elements on the agenda. For each rule on the agenda, the rule name and bound variables are shown.
3. The Global Data View shows all global data currently defined in the Drools working memory.

The Audit view can be used to show audit logs that contain events that were logged during the execution of a rules engine in a tree view.

7.6.1. The Working Memory View



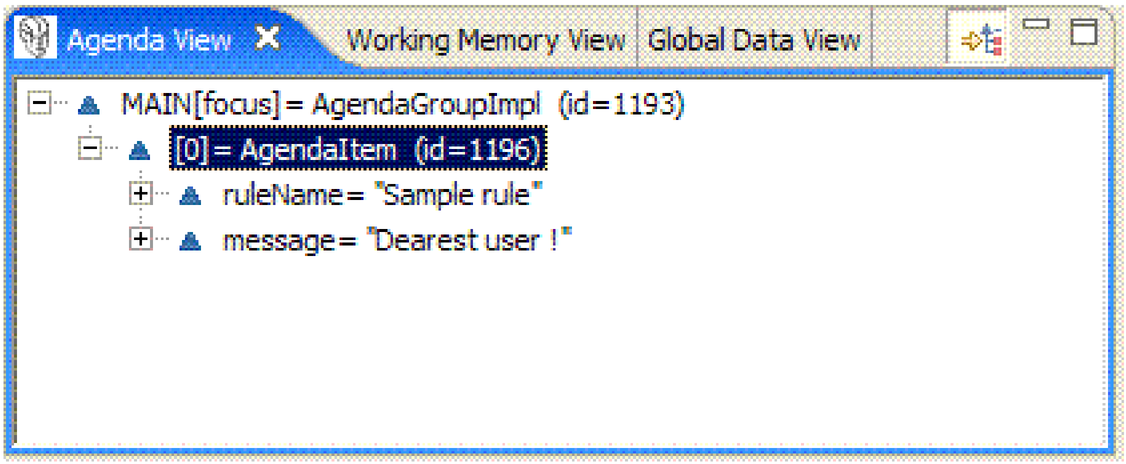
```
[0]= "Dearest user !"
```

The Working Memory shows all elements in the working memory of the Drools engine.

An action is added to the right of the view, to customize what is shown:

1. The Show Logical Structure toggles showing the logical structure of the elements in the working memory, or all their details. Logical structures allow for example visualizing sets of elements in a more obvious way.

7.6.2. The Agenda View



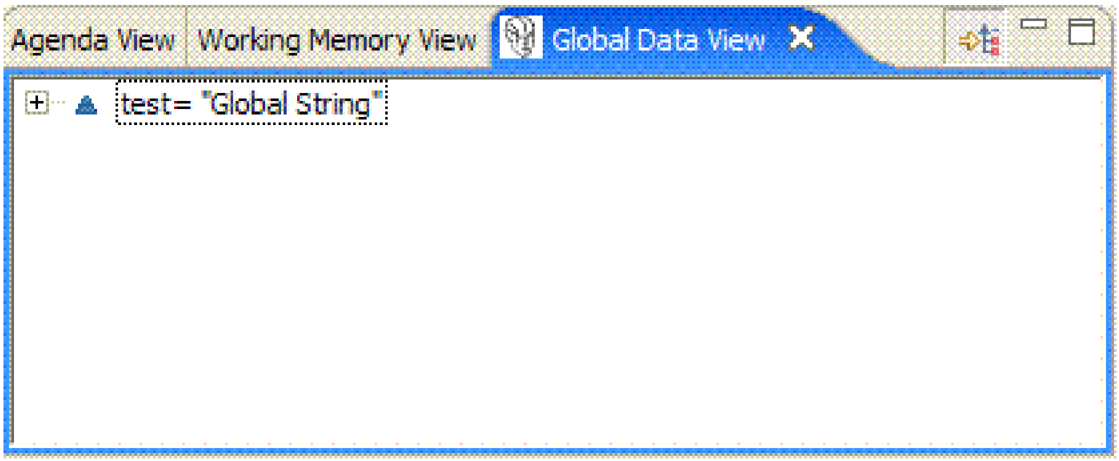
```
MAIN[focus]= AgendaGroupImpl  (id=1193)
    [0]= AgendaItem  (id=1196)
        ruleName= "Sample rule"
        message= "Dearest user !"
```

The Agenda View shows all elements on the agenda. For each rule on the agenda, the rule name and bound variables are shown.

An action is added to the right of the view, to customize what is shown:

1. The Show Logical Structure toggles showing the logical structure of the agenda item, or all their details. Logical structures allow for example visualizing sets of elements in a more obvious way. The logical structure of AgendaItems shows the rule that is represented by the AgendaItem, and the values of all the parameters used in the rule.

7.6.3. The Global Data View



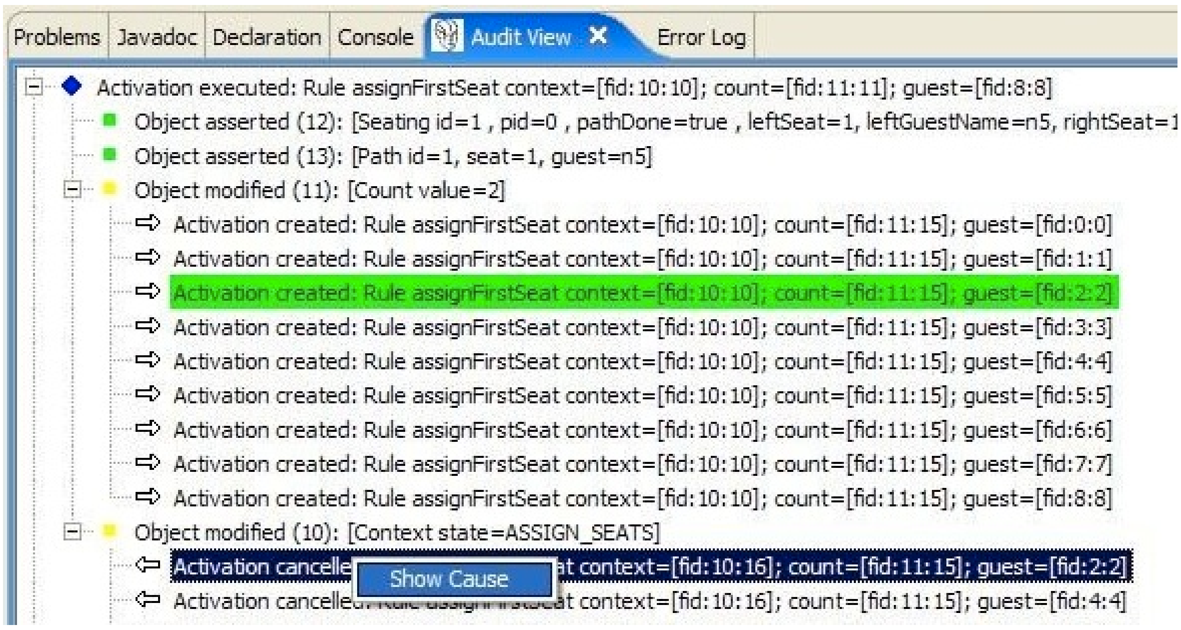
```
test= "Global String"
```

The Global Data View shows all global data currently defined in the Drools engine.

An action is added to the right of the view, to customize what is shown:

1. The Show Logical Structure toggles showing the logical structure of the elements in the working memory, or all their details. Logical structures allow for example visualizing sets of elements in a more obvious way.

7.6.4. The Audit View



```
Activation executed: Rule assignFirstSeat context=[fid:10:10]; count=[fid:11:11]; guest=[fid:8:8]
    Object asserted (12): [Seating id=1 , pid=0 , pathDone=true , leftSeat=1, leftGuestName=n5, rightSeat=1
    Object asserted (13): [Path id=1, seat=1, guest=n5]
    Object modified (11): [Count value=2]
        Activation created: Rule assignFirstSeat context=[fid:10:10]; count=[fid:11:15]; guest=[fid:0:0]
        Activation created: Rule assignFirstSeat context=[fid:10:10]; count=[fid:11:15]; guest=[fid:1:1]
        Activation created: Rule assignFirstSeat context=[fid:10:10]; count=[fid:11:15]; guest=[fid:2:2]
        Activation created: Rule assignFirstSeat context=[fid:10:10]; count=[fid:11:15]; guest=[fid:3:3]
        Activation created: Rule assignFirstSeat context=[fid:10:10]; count=[fid:11:15]; guest=[fid:4:4]
        Activation created: Rule assignFirstSeat context=[fid:10:10]; count=[fid:11:15]; guest=[fid:5:5]
        Activation created: Rule assignFirstSeat context=[fid:10:10]; count=[fid:11:15]; guest=[fid:6:6]
        Activation created: Rule assignFirstSeat context=[fid:10:10]; count=[fid:11:15]; guest=[fid:7:7]
        Activation created: Rule assignFirstSeat context=[fid:10:10]; count=[fid:11:15]; guest=[fid:8:8]
    Object modified (10): [Context state=ASSIGN_SEATS]
        Activation cancelle                    at context=[fid:10:16]; count=[fid:11:15]; guest=[fid:2:2]
        Activation cancelle                    at context=[fid:10:16]; count=[fid:11:15]; guest=[fid:4:4]
```

The audit view can be used to visualize an audit log that can be created when executing the rules engine. To create an audit log, use the following code:

```
WorkingMemory workingMemory = ruleBase.newWorkingMemory();
// create a new Working Memory Logger, that logs to file.
WorkingMemoryFileLogger logger = new
 WorkingMemoryFileLogger(workingMemory);
// an event.log file is created in the log dir (which must exist)
// in the working directory
logger.setFileName("log/event");

workingMemory.assertObject( ... );
workingMemory.fireAllRules();

// stop logging
logger.writeToDisk();
```

Open the log by clicking the Open Log action (first action in the Audit View) and select the file. The Audit view now shows all events that where logged during the executing of the rules. There are different types of events (each with a different icon):

1. Object inserted (green square)
2. Object updated (yellow square)
3. Object removed (red square)
4. Activation created (arrow to the right)
5. Activation cancelled (arrow to the left)
6. Activation executed (blue diamond)
7. Ruleflow started / ended (process icon)
8. Ruleflow-group activated / deactivated (process icon)
9. Rule package added / removed (Drools icon)
10. Rule added / removed (Drools icon)

All these events show extra information concerning the event, like the id and toString representation of the object in case of working memory events (assert, modify and retract), the name of the rule and all the variables bound in the activation in case of an activation event (created, cancelled or executed). If an event occurs when executing an activation, it is shown as a child of the activation executed event. For some events, you can retrieve the "cause":

1. The cause of an object modified or retracted event is the last object event for that object. This is either the object asserted event, or the last object modified event for that object.
2. The cause of an activation cancelled or executed event is the corresponding activation created event.

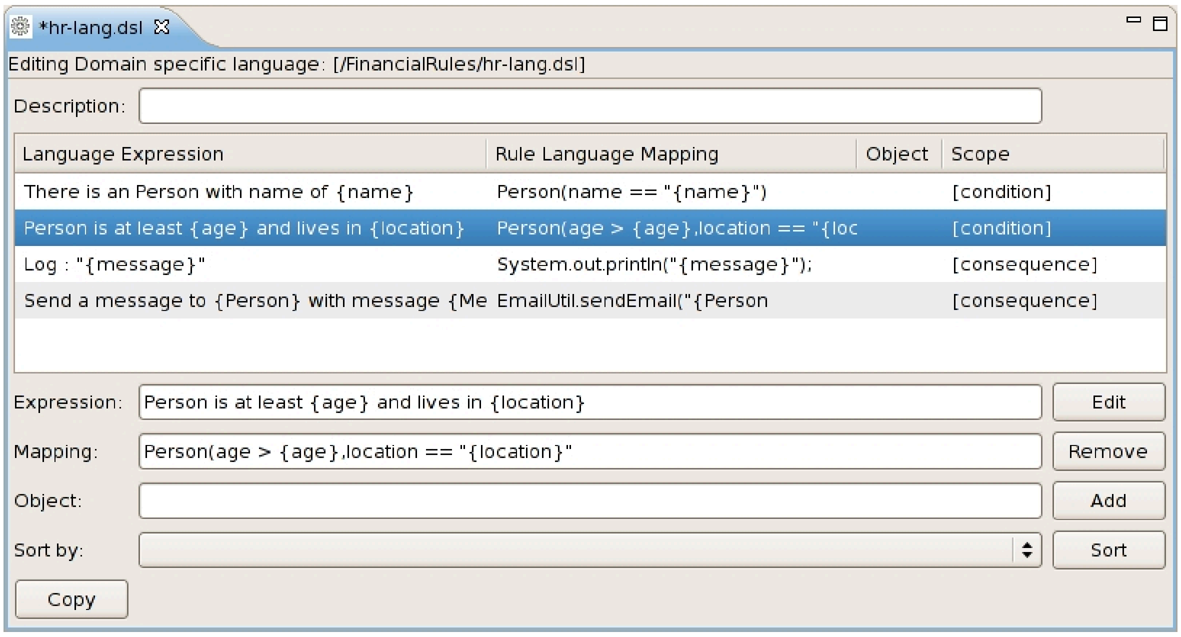
When selecting an event, the cause of that event is shown in green in the audit view (if visible of course). You can also right click the action and select the "Show Cause" menu item. This will scroll you to the cause of the selected event.

7.7. Domain Specific Languages

Domain Specific Languages (dsl) allow you to create a language that allows your rules to look like... rules ! Most often the domain specific language reads like natural language. Typically you would look at how a business analyst would describe the rule, in their own words, and then map this to your object model via rule constructs. A side benefit of this is that it can provide an insulation layer between your domain objects, and the rules themselves. A domain specific language will grow as the rules grow, and works best when there are common terms used over an over, with different parameters.

To aid with this, the rule workbench provides an editor for domain specific languages (they are stored in a plain text format, so you can use any editor of your choice - it uses a slightly enhanced version of the "Properties" file format, simply). The editor will be invoked on any files with a .dsl extension (there is also a wizard to create a sample DSL).

7.7.1. Editing languages



Editing Domain specific language: [/FinancialRules/hr-lang.dsl]

Description:

Language Expression	Rule Language Mapping	Object	Scope
There is an Person with name of {name}	Person(name == "{name}")		[condition]
Person is at least {age} and lives in {location}	Person(age > {age},location == "{loc		[condition]
Log : "{message}"	System.out.println("{message}");		[consequence]
Send a message to {Person} with message {Me	EmailUtil.sendEmail("{Person		[consequence]

Expression: Person is at least {age} and lives in {location}

Mapping: Person(age > {age},location == "{location}"

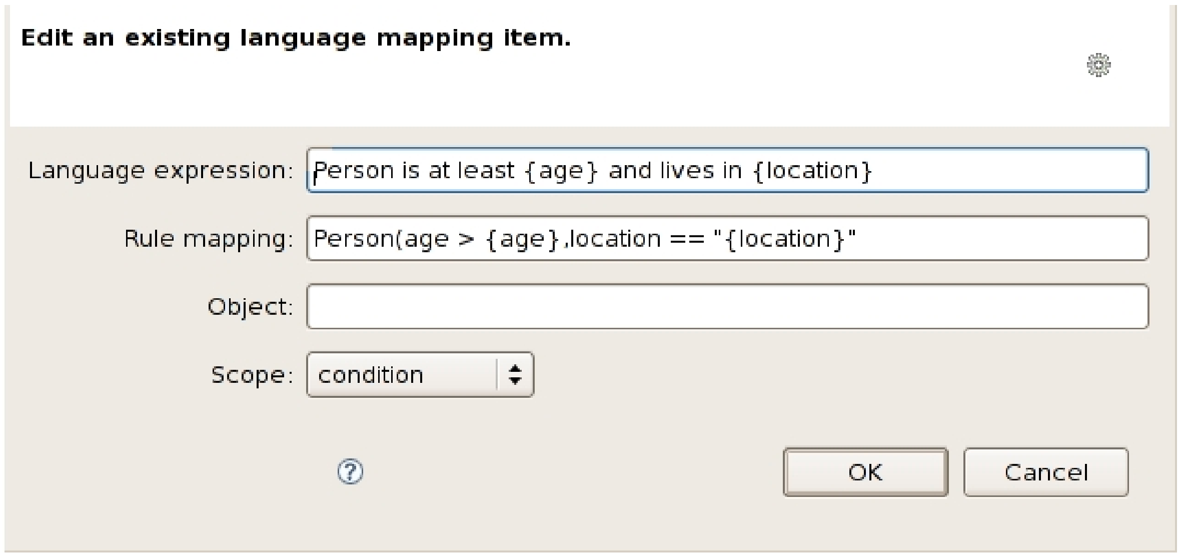
Object:

Sort by:

Figure 7.9. The Domain Specific Language editor

The DSL editor provides a table view of Language Expression to Rule Expression mapping. The Language expression is what is used in the rules. This also feeds the content assistance for the rule editor, so that it can suggest Language Expressions from the DSL configuration (the rule editor loads up the DSL configuration when the rule resource is loaded for editing). The Rule language mapping is the "code" for the rules - which the language expression will be compiled to by the rule engine compiler. For form of this Rule language depends if it is for a condition or action part of a rule (it may be a snippet of Java, for instance). The "scope" item indicates where the expression is targeted: is it for the "when" part of the rule (LHS)? the "then" part (RHS)? Or anywhere?

By selecting a mapping item (a row in the table) you can see the expression and mapping in the grayed out fields below. Double clicking or pressing the edit button will open the edit dialog. You can remove items, and add new ones (you should generally only remove when you know that expression is no longer in use).


Edit an existing language mapping item.

Language expression: Person is at least {age} and lives in {location}

Rule mapping: Person(age > {age},location == "{location}"

Object:

Scope: condition

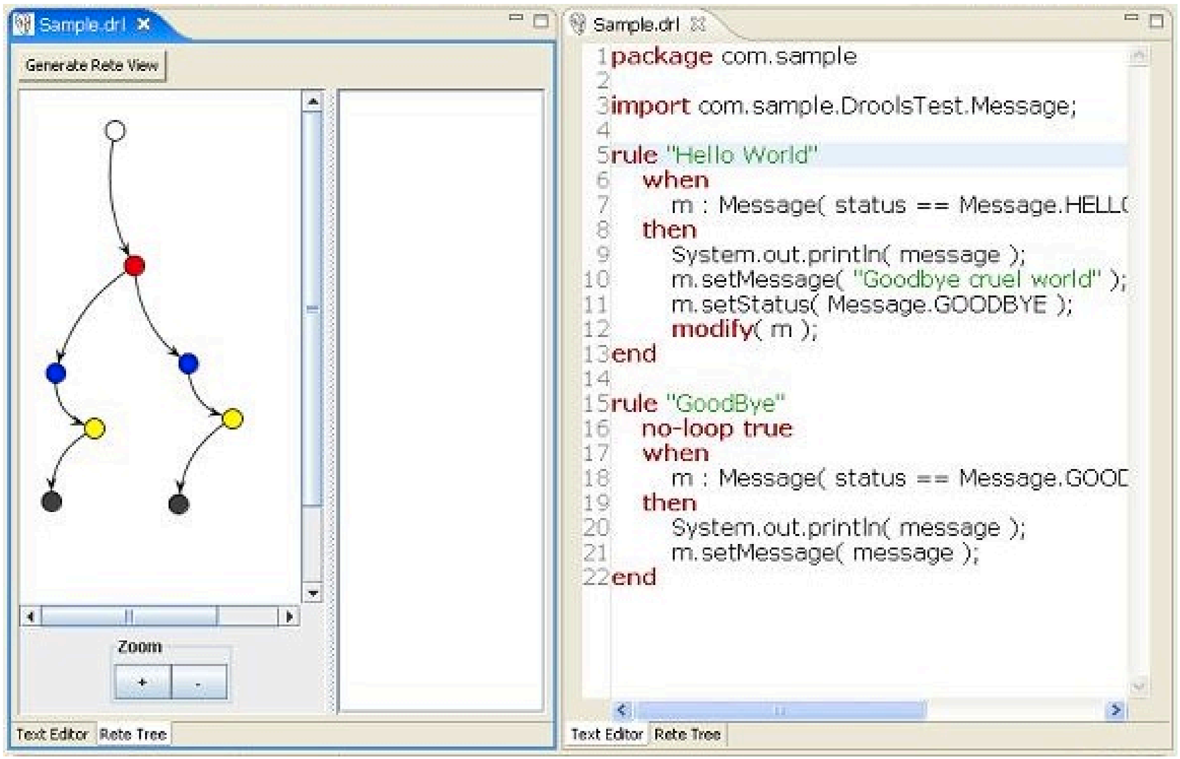
Figure 7.10. Language Mapping editor dialog

How it works: the "Language expression" is used to parse the rule language, depending on what the "scope" is set to. When it is found in a rule, the values that are market by the curly braces {value} are extracted from the rule source. These values are then interpolated with the "Rule mapping" expression, based on the names between the curly braces. So in the example above, the natural language expression maps to 2 constraints on a fact of type Person (i.e. the person object has the age field as less than {age}, and the location value is the string of {value}, where {age} and {value} are pulled out of the original rule source. The Rule mapping may be a Java expression (such as if the scope was "then"). If you did not wish to use a language mapping for a particular rule in a drl, prefix the expression with > and the compiler will not try to translate it according to the language definition. Also note that domain specific languages are optional. When the rule is compiled, the .dsl file will also need to be available.

7.8. The Rete View

The Rete Tree View shows you the current Rete Network for your drl file. Just click on the tab "Rete Tree" below on the DRL Editor. Afterwards you can generate the current Rete Network visualization. You can push and pull the nodes to arrange your optimal network overview. If you got hundreds of nodes, select some of them with a frame. Then you can pull groups of them. You can zoom in and out, in case not all nodes are shown in the current view. For this press the button "+" oder "-".

There is no export function, which creates a gif or jpeg picture, in the current release. Please use ctrl + alt + print to create a copy of your current Eclipse window and cut it off.



```
package com.sample

import com.sample.DroolsTest.Message;

rule "Hello World"
    when
        m : Message( status == Message.HELL(
    then
        System.out.println( message );
        m.setMessage( "Goodbye cruel world" );
        m.setStatus( Message.GOODBYE );
        modify( m );
end

rule "GoodBye"
    no-loop true
    when
        m : Message( status == Message.GOOD
    then
        System.out.println( message );
        m.setMessage( message );
end
```

The Rete View is an advanced feature which takes full advantage of the Eclipse Graphical Editing Framework (GEF).

The Rete view works only in Drools Rule Projects, where the Drools Builder is set in the project´s properties.

If you are using Drools in an other type of project, where you are not having a Drools Rule Project with the appropriate Drools Builder, you can create a little workaround:

Set up a little Drools Rule Project next to it, putting needed libraries into it and the drls you want to inspect with the Rete View. Just click on the right tab below in the DRL Editor, followed by a click on "Generate Rete View".

7.9. Large drl files

Depending on the JDK you use, it may be necessary to increase the permanent generation max size. Both SUN and IBM JDK have a permanent generation, whereas BEA JRockit does not.

To increase the permanent generation, start Eclipse with -XX:MaxPermSize=###m

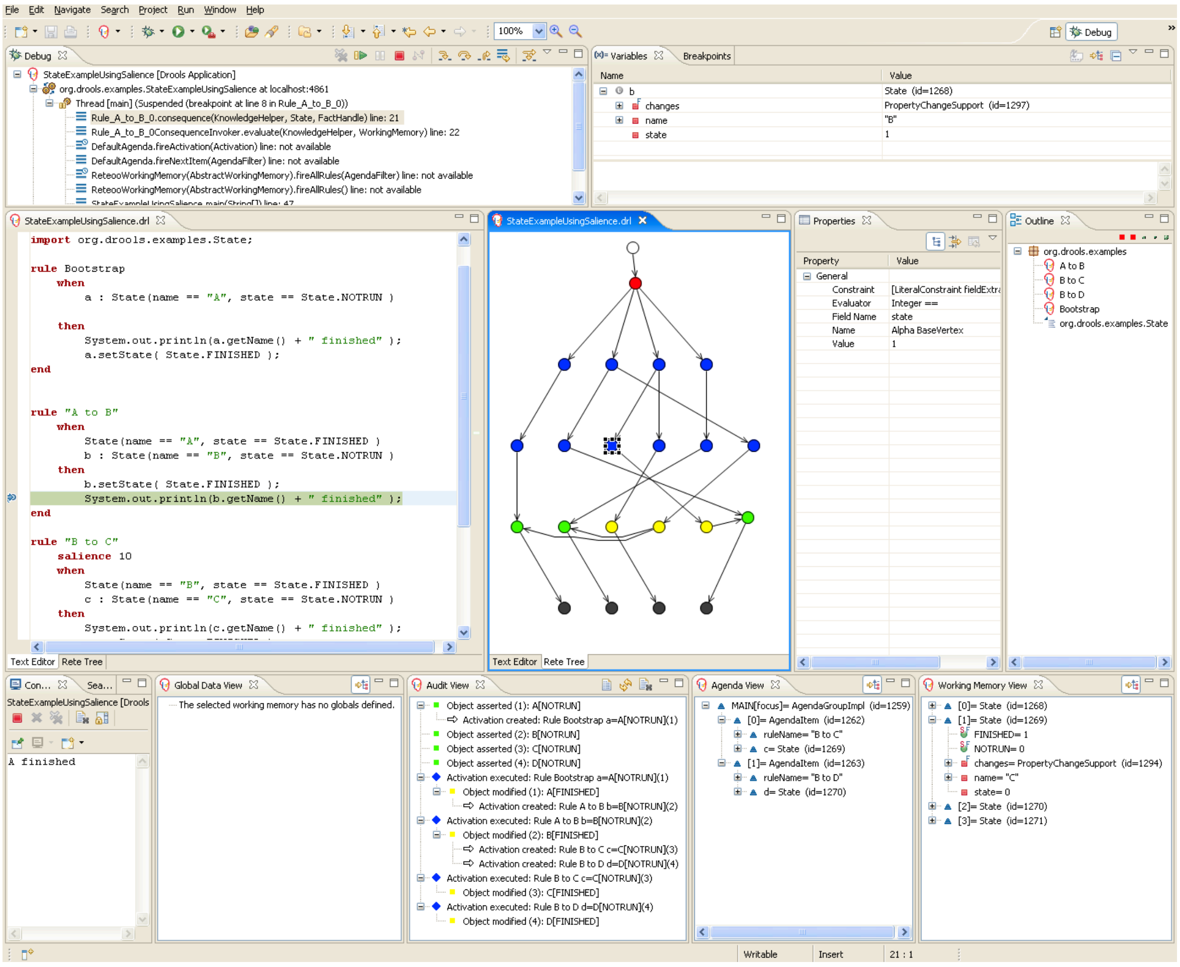
Example: c:\Eclipse\Eclipse.exe -XX:MaxPermSize=128m

Rulesets of 4,000 rules or greater should set the permanent generation to atleast 128Mb.

(note that this may also apply to compiling large numbers of rules in general - as there is generally one or more classes per rule).

As an alternative to the above, you may put rules in a file with the ".rule" extension, and the background builder will not try to compile them with each change, which may provide performance improvements if your IDE becomes sluggish with very large numbers of rules.

7.10. Debugging rules



```
StateExampleUsingSalience [Drools Application]
  org.drools.examples.StateExampleUsingSalience at localhost:4861
    Thread [main] (Suspended (breakpoint at line 8 in Rule_A_to_B_0))
      Rule_A_to_B_0.consequence(KnowledgeHelper, State, FactHandle) line: 21
      Rule_A_to_B_0ConsequenceInvoker.evaluate(KnowledgeHelper, WorkingMemory) line: 22
      DefaultAgenda.fireActivation(Activation) line: not available
      DefaultAgenda.fireNextItem(AgendaFilter) line: not available
      ReteooWorkingMemory(AbstractWorkingMemory).fireAllRules(AgendaFilter) line: not available
      ReteooWorkingMemory(AbstractWorkingMemory).fireAllRules() line: not available
      StateExampleUsingSalience.main(String[]) line: 47
```

Name	Value
b	State (id=1268)
changes	PropertyChangeSupport (id=1297)
name	"B"
state	1

```
import org.drools.examples.State;

rule Bootstrap
    when
        a : State(name == "A", state == State.NOTRUN )

    then
        System.out.println(a.getName() + " finished" );
        a.setState( State.FINISHED );
end


rule "A to B"
    when
        State(name == "A", state == State.FINISHED )
        b : State(name == "B", state == State.NOTRUN )
    then
        b.setState( State.FINISHED );
        System.out.println(b.getName() + " finished" );
end

rule "B to C"
    salience 10
    when
        State(name == "B", state == State.FINISHED )
        c : State(name == "C", state == State.NOTRUN )
    then
        System.out.println(c.getName() + " finished" );
```

Property	Value
General	
Constraint	[LiteralConstraint fieldExtra
Evaluator	Integer ==
Field Name	state
Name	Alpha BaseVertex
Value	1

```
org.drools.examples
  A to B
  B to C
  B to D
  Bootstrap
  org.drools.examples.State
```

```
StateExampleUsingSalience [Drools
A finished
```

The selected working memory has no globals defined.

```
Object asserted (1): A[NOTRUN]
  Activation created: Rule Bootstrap a=A[NOTRUN](1)
Object asserted (2): B[NOTRUN]
Object asserted (3): C[NOTRUN]
Object asserted (4): D[NOTRUN]
Activation executed: Rule Bootstrap a=A[NOTRUN](1)
  Object modified (1): A[FINISHED]
    Activation created: Rule A to B b=B[NOTRUN](2)
Activation executed: Rule A to B b=B[NOTRUN](2)
  Object modified (2): B[FINISHED]
    Activation created: Rule B to C c=C[NOTRUN](3)
    Activation created: Rule B to D d=D[NOTRUN](4)
Activation executed: Rule B to C c=C[NOTRUN](3)
  Object modified (3): C[FINISHED]
Activation executed: Rule B to D d=D[NOTRUN](4)
  Object modified (4): D[FINISHED]
```

```
MAIN[focus]= AgendaGroupImpl  (id=1259)
  [0]= AgendaItem  (id=1262)
    ruleName= "B to C"
    c= State  (id=1269)
  [1]= AgendaItem  (id=1263)
    ruleName= "B to D"
    d= State  (id=1270)
```

```
[0]= State  (id=1268)
[1]= State  (id=1269)
  FINISHED= 1
  NOTRUN= 0
  changes= PropertyChangeSupport  (id=1294)
  name= "C"
  state= 0
[2]= State  (id=1270)
[3]= State  (id=1271)
```

Figure 7.11. Debugging

You can debug rules during the execution of your Drools application. You can add breakpoints in the consequences of your rules, and whenever such a breakpoint is uncounted during the execution of the rules, the execution is halted. You can then inspect the variables known at that point and use any of the default debugging actions to decide what should happen next (step over, continue, etc.). You can also use the debug views to inspect the content of the working memory and agenda.

7.10.1. Creating breakpoints

You can add/remove rule breakpoints in drl files in two ways, similar to adding breakpoints to Java files:

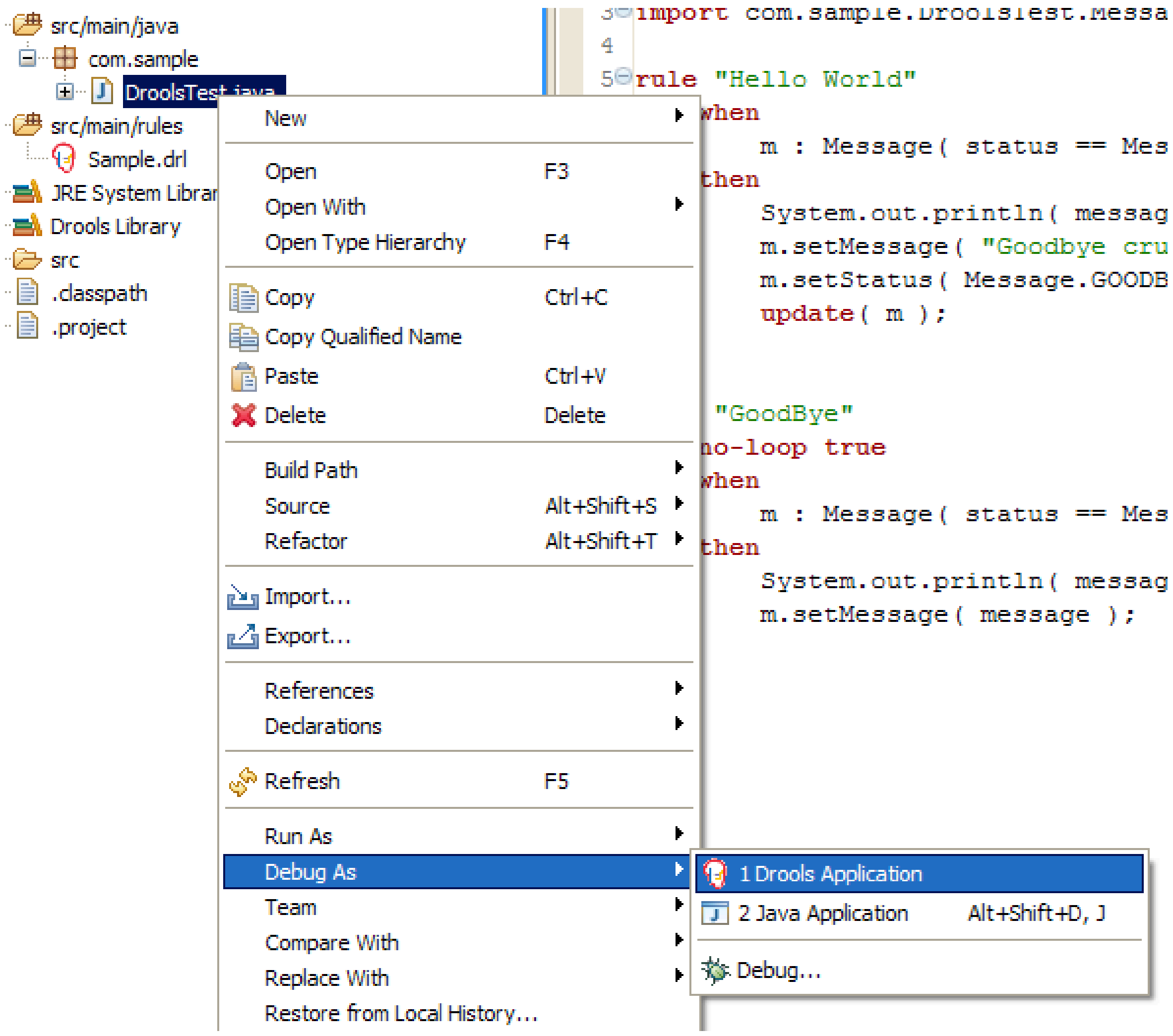
1. Double-click the ruler of the DRL editor at the line where you want to add a breakpoint. Note that rule breakpoints can only be created in the consequence of a rule. Double-clicking on a line where no breakpoint is allowed will do nothing. A breakpoint can be removed by double-clicking the ruler once more.

2. If you right-click the ruler, a popup menu will show up, containing the "Toggle breakpoint" action. Note that rule breakpoints can only be created in the consequence of a rule. The action is automatically disabled if no rule breakpoint is allowed at that line. Clicking the action will add a breakpoint at the selected line, or remove it if there was one already.

The Debug Perspective contains a Breakpoints view which can be used to see all defined breakpoints, get their properties, enable/disable or remove them, etc.

7.10.2. Debugging rules

Drools breakpoints are only enabled if you debug your application as a Drools Application. You can do this like this:



```
import com.sample.DroolsTest.Messa

rule "Hello World"
when
    m : Message( status == Mes
then
    System.out.println( messag
    m.setMessage( "Goodbye cru
    m.setStatus( Message.GOODB
    update( m );


 "GoodBye"
no-loop true
when
    m : Message( status == Mes
then
    System.out.println( messag
    m.setMessage( message );
```

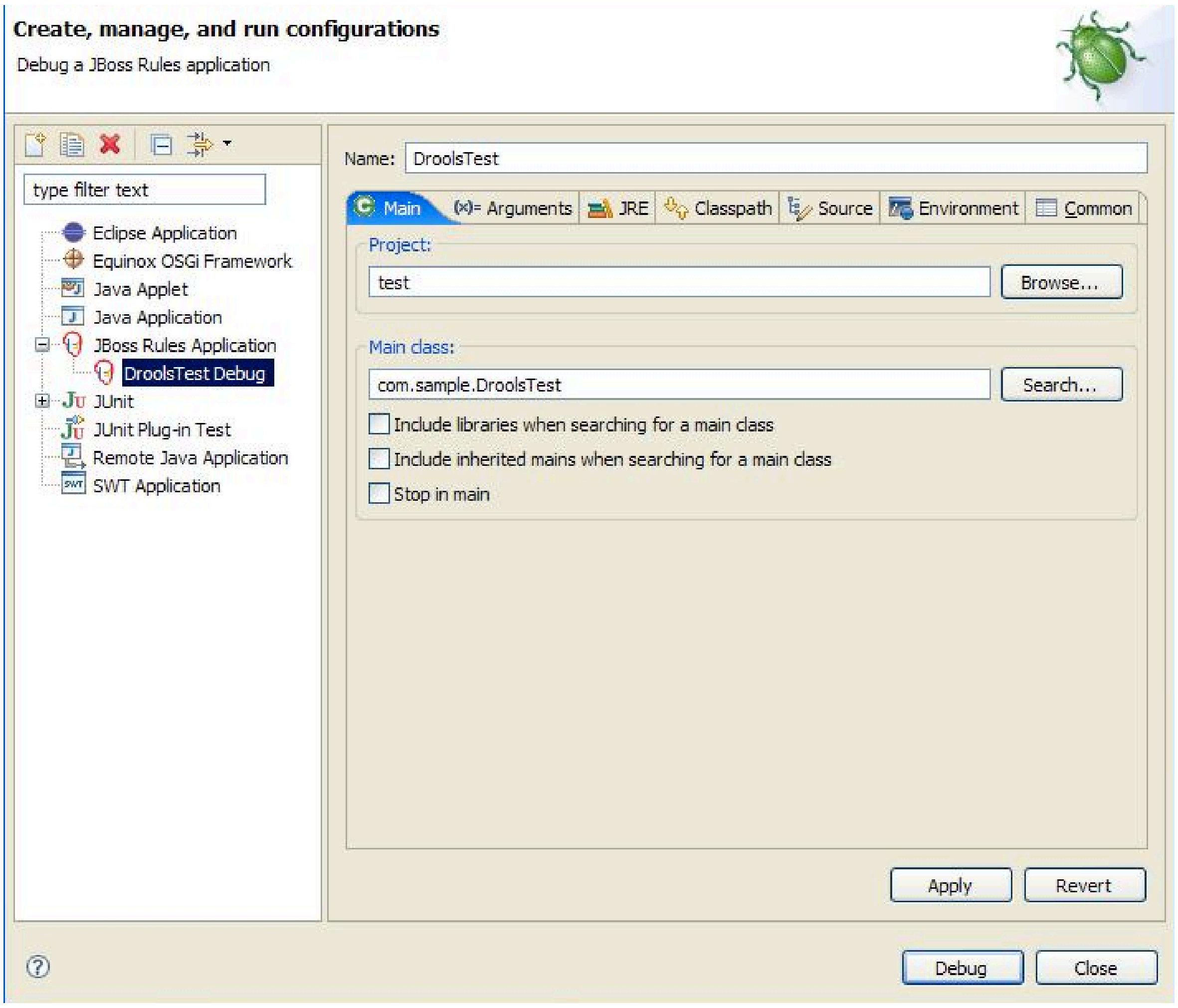
Figure 7.12. Debug as Drools Application

1. Select the main class of your application. Right click it and select the "Debug As >" sub-menu and select Drools Application. Alternatively, you can also select the "Debug ..." menu item to open a new dialog for creating, managing and running debug configurations (see screenshot below)

    a. Select the "Drools Application" item in the left tree and click the "New launch configuration" button (leftmost icon in the toolbar above the tree). This will create a new configuration and already fill in some of the properties (like the project and main class) based on main class you selected in the beginning. All properties shown here are the same as any standard Java program.

b. Change the name of your debug configuration to something meaningful. You can just accept the defaults for all other properties. For more information about these properties, please check the Eclipse jdt documentation.

c. Click the "Debug" button on the bottom to start debugging your application. You only have to define your debug configuration once. The next time you try to run your Drools application, you don't have to create a new one but select the one you defined previously by selecting it in the tree on the left, as a sub-element of the "Drools Application" tree node, and then click the Debug button. The Eclipse toolbar also contains shortcut buttons to quickly re-execute the one of your previous configurations (at least when the Java, Java Debug, or Drools perspective has been selected).


Create, manage, and run configurations

Debug a JBoss Rules application

type filter text

- Eclipse Application
- Equinox OSGi Framework
- Java Applet
- Java Application
- JBoss Rules Application
  - DroolsTest Debug
- JUnit
- JUnit Plug-in Test
- Remote Java Application
- SWT Application

Name: DroolsTest

Main | Arguments | JRE | Classpath | Source | Environment | Common

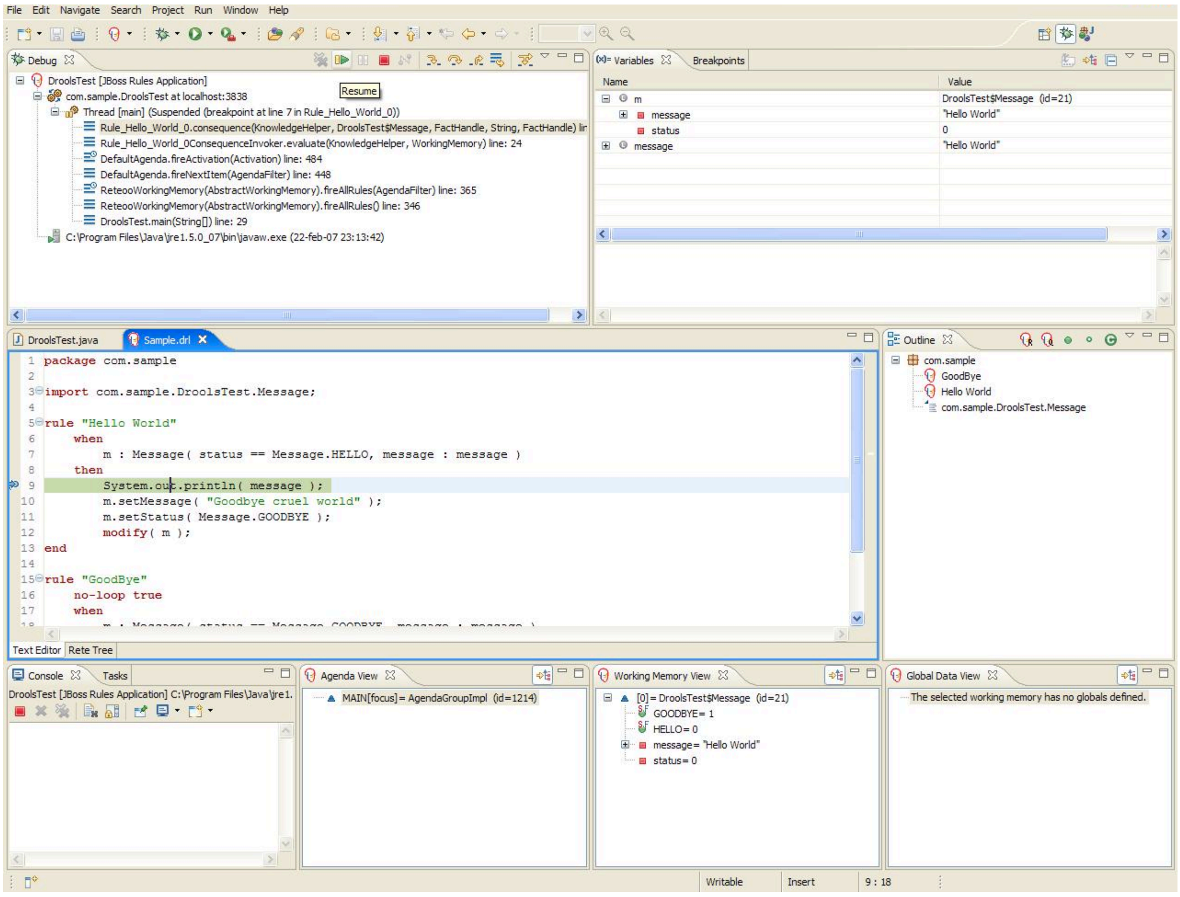
Project:
test

Main class:
com.sample.DroolsTest

- [ ] Include libraries when searching for a main class
- [ ] Include inherited mains when searching for a main class
- [ ] Stop in main

Figure 7.13. Debug as Drools Application Configuration

After clicking the "Debug" button, the application starts executing and will halt if any breakpoint is encountered. This can be a Drools rule breakpoint, or any other standard Java breakpoint. Whenever a Drools rule breakpoint is encountered, the corresponding drl file is opened and the active line is highlighted. The Variables view also contains all rule parameters and their value. You can then use the default Java debug actions to decide what to do next (resume, terminate, step over, etc.). The debug views can also be used to determine the contents of the working memory and agenda at that time as well (you don't have to select a working memory now, the current executing working memory is automatically shown).



```
DroolsTest [JBoss Rules Application]
  com.sample.DroolsTest at localhost:3838
    Thread [main] (Suspended (breakpoint at line 7 in Rule_Hello_World_0))
      Rule_Hello_World_0.consequence(KnowledgeHelper, DroolsTest$Message, FactHandle, String, FactHandle) lir
      Rule_Hello_World_0ConsequenceInvoker.evaluate(KnowledgeHelper, WorkingMemory) line: 24
      DefaultAgenda.fireActivation(Activation) line: 484
      DefaultAgenda.fireNextItem(AgendaFilter) line: 448
      ReteooWorkingMemory(AbstractWorkingMemory).fireAllRules(AgendaFilter) line: 365
      ReteooWorkingMemory(AbstractWorkingMemory).fireAllRules() line: 346
      DroolsTest.main(String[]) line: 29
  C:\Program Files\Java\jre1.5.0_07\bin\javaw.exe (22-feb-07 23:13:42)
```

Name	Value
m	DroolsTest$Message (id=21)
message	"Hello World"
status	0
message	"Hello World"

```
package com.sample

import com.sample.DroolsTest.Message;

rule "Hello World"
    when
        m : Message( status == Message.HELLO, message : message )
    then
        System.out.println( message );
        m.setMessage( "Goodbye cruel world" );
        m.setStatus( Message.GOODBYE );
        modify( m );
end

rule "GoodBye"
    no-loop true
    when
```

```
com.sample
  GoodBye
  Hello World
  com.sample.DroolsTest.Message
```

DroolsTest [JBoss Rules Application] C:\Program Files\Java\jre1.

```
MAIN[focus]= AgendaGroupImpl  (id=1214)
```

```
[0]= DroolsTest$Message  (id=21)
  GOODBYE= 1
  HELLO= 0
  message= "Hello World"
  status= 0
```

The selected working memory has no globals defined.

Figure 7.14. Debugging

Examples

8.1. Hello World

Name:	Banking Tutorial
Main class:	**`org.drools.tutorials.banking`**
Type:	java application
Objective:	Tutorial that builds up knowledge of pattern matching, basic sorting and calculation rules.

The "Hello World" example shows a simple example of rules usage, and both the MVEL and Java dialects. In this example it will be shown how to build knowledge bases and sessions and how to add audit logging and debug outputs. This information is omitted from other examples as it's all very similar.

KnowledgeBuilder is used to convert a drl source file into Package objects that the KnowledgeBase can consume, and takes a Resource interface and ResourceType as parameters. Resource can be used to retrieve a source drl file from various locations, in this case the drl file is being retrieved from the classpath using ResourceFactory; but it could come from the disk or a url.

In this case we only add a single drl source file, however multiple drl files can be added. Drl files with different namespaces can be added; KnowledgeBuilder creates a package for each namespace. Additionally, multiple packages of different namespaces can be added to the same KnowledgeBase.

When all the drl files have been added, we should check the builder for errors. The KnowledgeBase will validate the package, however it will only have access to the error information as a String. If you wish to debug the error information you should do it on the builder instance. Once we know the builder is error, we can do the following: get the Package collection; instantiate a KnowledgeBase from the KnowledgeBaseFactory; and add the package collection.

```
final KnowledgeBuilder kbuilder =
 KnowledgeBuilderFactory.newKnowledgeBuilder();

// this will parse and compile in one step
kbuilder.add(ResourceFactory.newClassPathResource
 ("HelloWorld.drl",HelloWorldExample.class), ResourceType.DRL);

// Check the builder for errors
if (kbuilder.hasErrors())
{
 System.out.println(kbuilder.getErrors().toString());
 throw new RuntimeException("Unable to compile \"HelloWorld.drl\".");
}

// get the compiled packages (which are serializable)
final Collection<KnowledgePackage> pkgs = kbuilder.getKnowledgePackages();
```

```
// add the packages to a knowledgebase (deploy the knowledge packages).
final KnowledgeBase kbase = KnowledgeBaseFactory.newKnowledgeBase();
kbase.addKnowledgePackages(pkgs);

final StatefulKnowledgeSession ksession =
 kbase.newStatefulKnowledgeSession();
```

Drools has an event model that exposes much of what's happening internally. Two default debug listeners are supplied that print out debug event information to the err console: DebugAgendaEventListener, and DebugWorkingMemoryEventListener. Adding listeners to a session is trivial and is discussed later. The KnowledgeRuntimeLogger is a specialized implementation built on the agenda and working memory listeners, and provides execution auditing which can be viewed in a graphical viewer. When the engine has finished executing, logger.close() must be called.

Most of the examples use the Audit logging features of Drools to record execution flow for later inspection.

```
// setup the debug listeners
ksession.addEventListener( new DebugAgendaEventListener() );
ksession.addEventListener( new DebugWorkingMemoryEventListener() );

// setup the audit logging
KnowledgeRuntimeLogger logger =
 KnowledgeRuntimeLoggerFactory.newFileLogger(ksession,"log/helloworld");
```

Example 8.1. Event Logging and Auditing

The single class used in this example only has two fields: the message, which is a String; and the status, which can be either the int HELLO or the int GOODBYE.

```
public static class Message
{
    public static final int HELLO   = 0;
    public static final int GOODBYE = 1;

    private String          message;
    private int             status;
    ...
}
```

Example 8.2. Message Class

A single Message object is created with the message "Hello World" and status HELLO and then inserted into the engine, at which point fireAllRules() is executed. Remember all the network evaluation is done during the insert time, by the time the program execution reaches the fireAllRules() method it already knows which rules are fully matches and able to fire.

```
final Message message = new Message();
message.setMessage("Hello World");
message.setStatus(Message.HELLO);
ksession.insert(message);

ksession.fireAllRules();

logger.close();

ksession.dispose();
```

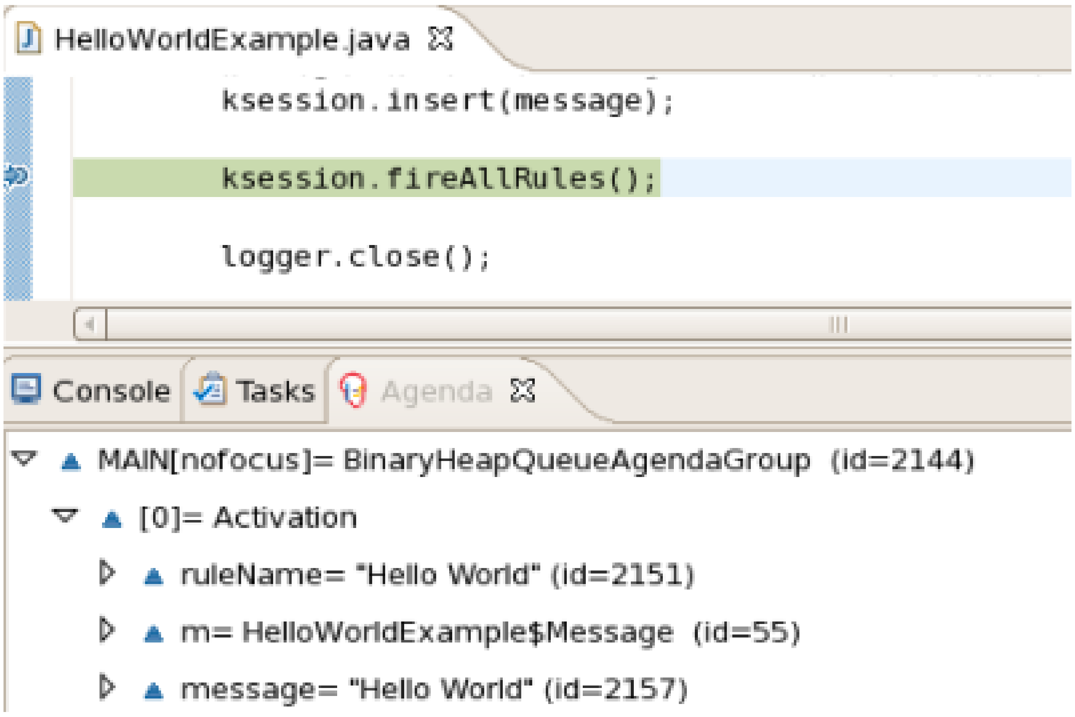
Example 8.3. Execution

To execute the example from Java.

1. Open the class org.drools.examples.HelloWorldExample in your Eclipse IDE.

2. Right-click the class an select Run as... > Java application.

If we put a breakpoint on the fireAllRules() method and select the ksession variable we can see that the "Hello World" view is already activated and on the Agenda, showing that all the pattern matching work was already done during the insert.



```
        ksession.insert(message);

        ksession.fireAllRules();

        logger.close();
```

```
MAIN[nofocus]= BinaryHeapQueueAgendaGroup  (id=2144)
  [0]= Activation
    ruleName= "Hello World" (id=2151)
    m= HelloWorldExample$Message  (id=55)
    message= "Hello World" (id=2157)
```

Figure 8.1. fireAllRules Agenda View

The may application printouts go to System.out, while the debug listener printouts go to System.err.

```
Hello World
Goodbye cruel world
```

Example 8.4. Console.out

```
==>[ActivationCreated(0): rule=Hello World;

tuple=[fid:1:1:org.drools.examples.HelloWorldExample$Message@17cec96]]
[ObjectInserted: handle=[fid:1:1:org.drools.examples.HelloWorldExample
$Message@17cec96];
object=org.drools.examples.HelloWorldExample$Message@17cec96]
[BeforeActivationFired: rule=Hello World;

tuple=[fid:1:1:org.drools.examples.HelloWorldExample$Message@17cec96]]

==>[ActivationCreated(4): rule=Good Bye;
tuple=[fid:1:2:org.drools.examples.HelloWorldExample$Message@17cec96]]
[ObjectUpdated: handle=[fid:1:2:org.drools.examples.HelloWorldExample
$Message@17cec96];
old_object=org.drools.examples.HelloWorldExample$Message@17cec96;

new_object=org.drools.examples.HelloWorldExample$Message@17cec96]
[AfterActivationFired(0): rule=Hello World]
[BeforeActivationFired: rule=Good Bye;

tuple=[fid:1:2:org.drools.examples.HelloWorldExample$Message@17cec96]]
[AfterActivationFired(4): rule=Good Bye]
```

Example 8.5. Console.err

The *LHS (when)* section of the rule states that it will be activated for each *Message* object inserted into the working memory whose *status* is *Message.HELLO*. Besides that, two variable binds are created: "*message*" variable is bound to the *message* attribute and "*m*" variable is bound to the *object matched pattern* itself.

The *RHS (consequence, then)* section of the rule is written using the MVEL expression language, as declared by the rule's attribute *dialect*. After printing the content of the *message* bound variable to the default console, the rule changes the values of the *message* and *status* attributes of the *m* bound variable; using MVEL's 'modify' keyword which allows you to apply a block of setters in one statement, with the engine being automatically notified of the changes at the end of the block.

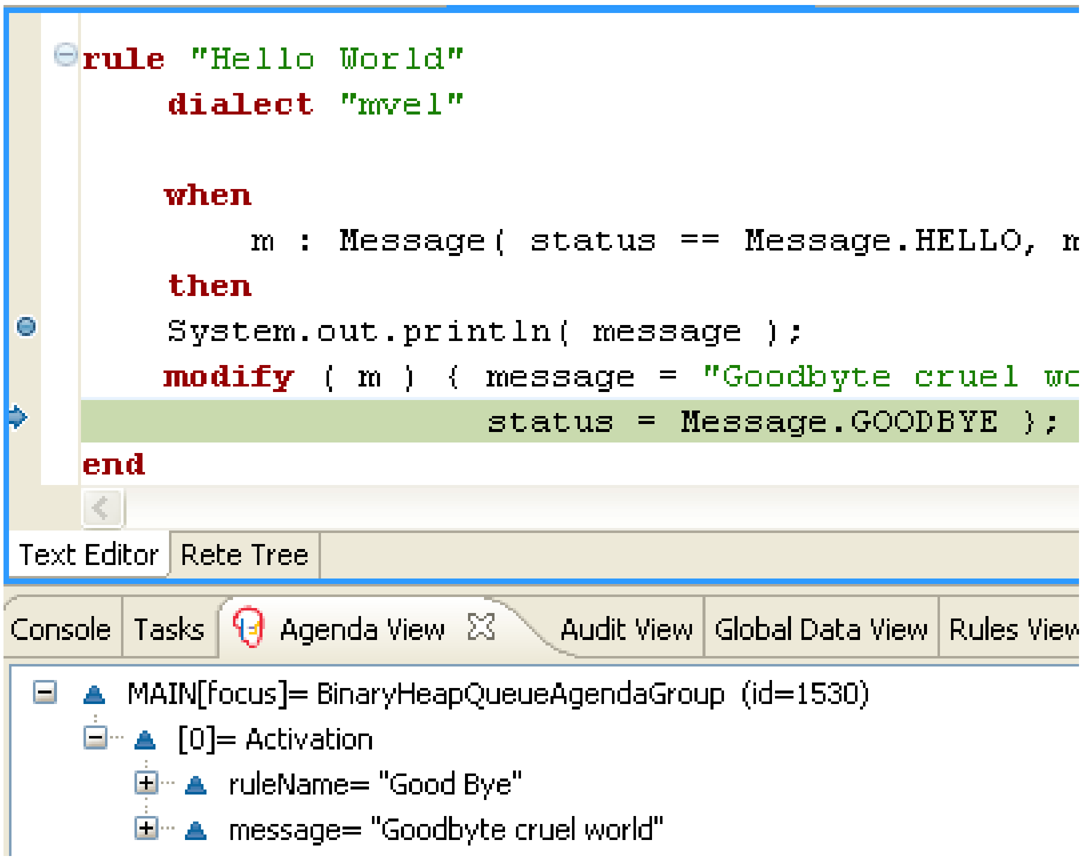
```
rule "Hello World"
 dialect "mvel"
when
 m : Message( status == Message.HELLO, message : message )
then
 System.out.println( message );
 modify (m) { message="Goodbye cruel world", status=Message.GOODBYE };
end
```

Example 8.6. Rule "Hello World"

We can add a break point into the DRL for when modify is called during the execution of the "Hello World" consequence and inspect the Agenda view again. Notice this time we debug as a Drools application and not a Java application.

1. Open the class org.drools.examples.FibonacciExample in your Eclipse IDE.
2. Right-click the class an select Debug as... > Drools application.

Now we can see that the other rule "Good Bye" which uses the java dialect is activated and placed on the agenda.



```
rule "Hello World"
    dialect "mvel"

    when
        m : Message( status == Message.HELLO, m
    then
    System.out.println( message );
    modify ( m ) { message = "Goodbyte cruel wo
                    status = Message.GOODBYE };
end
```

```
MAIN[focus]= BinaryHeapQueueAgendaGroup (id=1530)
    [0]= Activation
        ruleName= "Good Bye"
        message= "Goodbyte cruel world"
```

Figure 8.2. Rule "Hello World" Agenda View

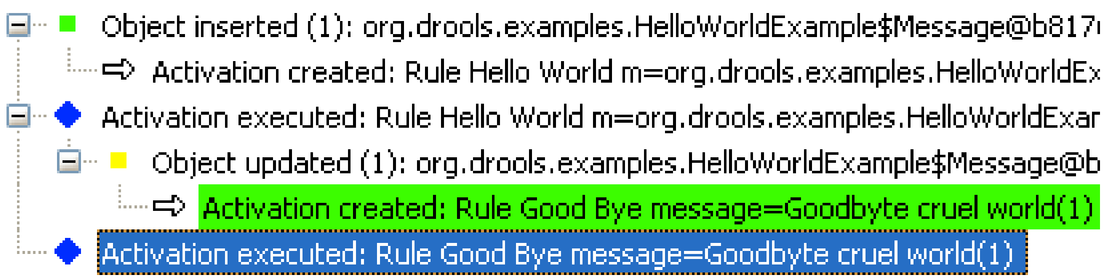
The "Good Bye" rule is similar to the "Hello World" rule; the difference is it matches Message objects whose status is Message.GOODBYE. It then prints its message to the default console, and specifies the "java" dialect.

```
rule "Good Bye"
 dialect "java"
when
 Message( status == Message.GOODBYE, message : message )
then
 System.out.println( message );
end
```

Example 8.7. Rule "Good Bye"

If you remember at the start of this example in the java code we used KnowledgeRuntimeLoggerFactory.newFileLogger to create a KnowledgeRuntimeLogger and called logger.close() at the end. This created an audit log file that can be shown in the Audit view. We use the audit view in many of the examples to try and understand the example execution flow.

In the following view, we can see an object is inserted that creates an activation for the "Hello World" rule. The activation is then executed, which updates the Message object causing the "Good Bye" rule to activate, and subsequently execute. When an event in the Audit view is selected, it highlights the origin event in green. The Activation created event is highlighted in green as the origin of the Activation executed event.



```
Object inserted (1): org.drools.examples.HelloWorldExample$Message@b817
    Activation created: Rule Hello World m=org.drools.examples.HelloWorldEx
Activation executed: Rule Hello World m=org.drools.examples.HelloWorldExar
    Object updated (1): org.drools.examples.HelloWorldExample$Message@b
        Activation created: Rule Good Bye message=Goodbyte cruel world(1)
Activation executed: Rule Good Bye message=Goodbyte cruel world(1)
```

Figure 8.3. Audit View

8.2. State Example

This example is implemented in three different versions to demonstrate different ways of implementing rules forward chaining behaviour. The behaviour provides the engine the ability to evaluate, activate, and fire rules in sequence, based on changes on the facts in working memory.

8.2.1. Understanding the State Example

Name:	State Example
Main class:	**`org.drools.examples.StateExampleUsingSalience`**
Type:	java application
Rules file:	**`StateExampleUsingSalience.drl`**
Objective:	Demonstrate basic rule use and conflict resolution for rule firing priority.

Each State class has fields for its name and its current state (see org.drools.examples.State class). The two possible states for each objects are NOTRUN and FINISHED.

```
public class State
{
 public static final int       NOTRUN   = 0;
 public static final int       FINISHED = 1;

 private final PropertyChangeSupport changes  =
  new PropertyChangeSupport( this );

 private String                name;
 private int                   state;

 //... setters and getters go here...
}
```

Example 8.8. State Class

Ignore the PropertyChangeSupport for now; this concept will be explained later. In the example, we create four State objects with names: A, B, C and D. All are set to state NOTRUN initially, which is default for this constructor. Each instance is asserted in turn into the session and then fireAllRules() is called.

```
State a = new State( "A" );
State b = new State( "B" );
State c = new State( "C" );
final State d = new State( "D" );

// By setting dynamic to TRUE, Drools will use JavaBean
// PropertyChangeListeners so you don't have to call update().
boolean dynamic = true;

session.insert( a,dynamic );
session.insert( b,dynamic );
session.insert( c,dynamic );
session.insert( d,dynamic );

session.fireAllRules();
// Stateful rule session must always be disposed when finished.
session.dispose();
```

Example 8.9. Salience State Example Execution

To execute the application:

1. Open the class org.drools.examples.StateExampleUsingSalience in your Eclipse IDE.

2. Right-click the class an select Run as... > Java application.

The following output is displayed in the Eclipse console output:

```
A finished
B finished
C finished
D finished
```

Example 8.10. Console Output

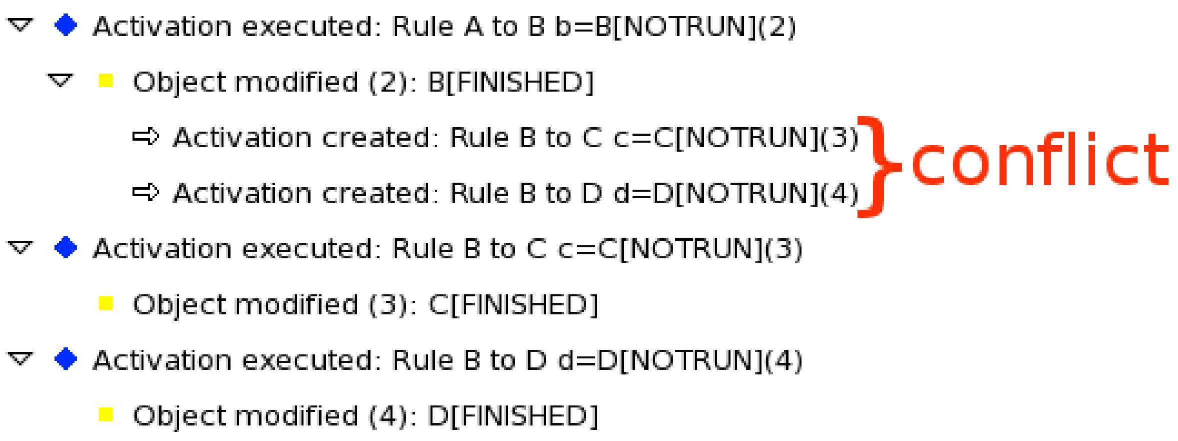
There are four rules in total, first a Bootstrap rule fires setting A to state FINISHED which then causes B to change to state FINISHED. C and D are both dependent on B - causing a conflict which is resolved by setting salience values. Lets look at how this was executed.

The best way to understand what is happening is to use the "Audit Log" feature to graphically see the results of each operation. The audit log was generated when the example was previously run. To view the audit log in Eclipse:

1. If the "Audit View" is not visible, select Window > Show View > Other... > Drools > Audit View.

2. In the "Audit View" click in the Open Log button and select File < drools-examples-drl-dir > /log/ state.log.

The "Audit view" will look similar to the following screenshot.



```
Activation executed: Rule A to B b=B[NOTRUN](2)
    Object modified (2): B[FINISHED]
        Activation created: Rule B to C c=C[NOTRUN](3)
        Activation created: Rule B to D d=D[NOTRUN](4)
Activation executed: Rule B to C c=C[NOTRUN](3)
    Object modified (3): C[FINISHED]
Activation executed: Rule B to D d=D[NOTRUN](4)
    Object modified (4): D[FINISHED]
```

Figure 8.4. Audit View

Reading the log in the "Audit View", top to down, we see every action and the corresponding changes in the working memory. This way we see that the assertion of the State "A" object with the "NOTRUN" state activates the "Bootstrap" rule, while the assertions of the other state objects have no immediate effect.

```
rule Bootstrap
when
 a : State(name == "A", state == State.NOTRUN )
then
 System.out.println(a.getName() + " finished" );
 a.setState( State.FINISHED );
end
```

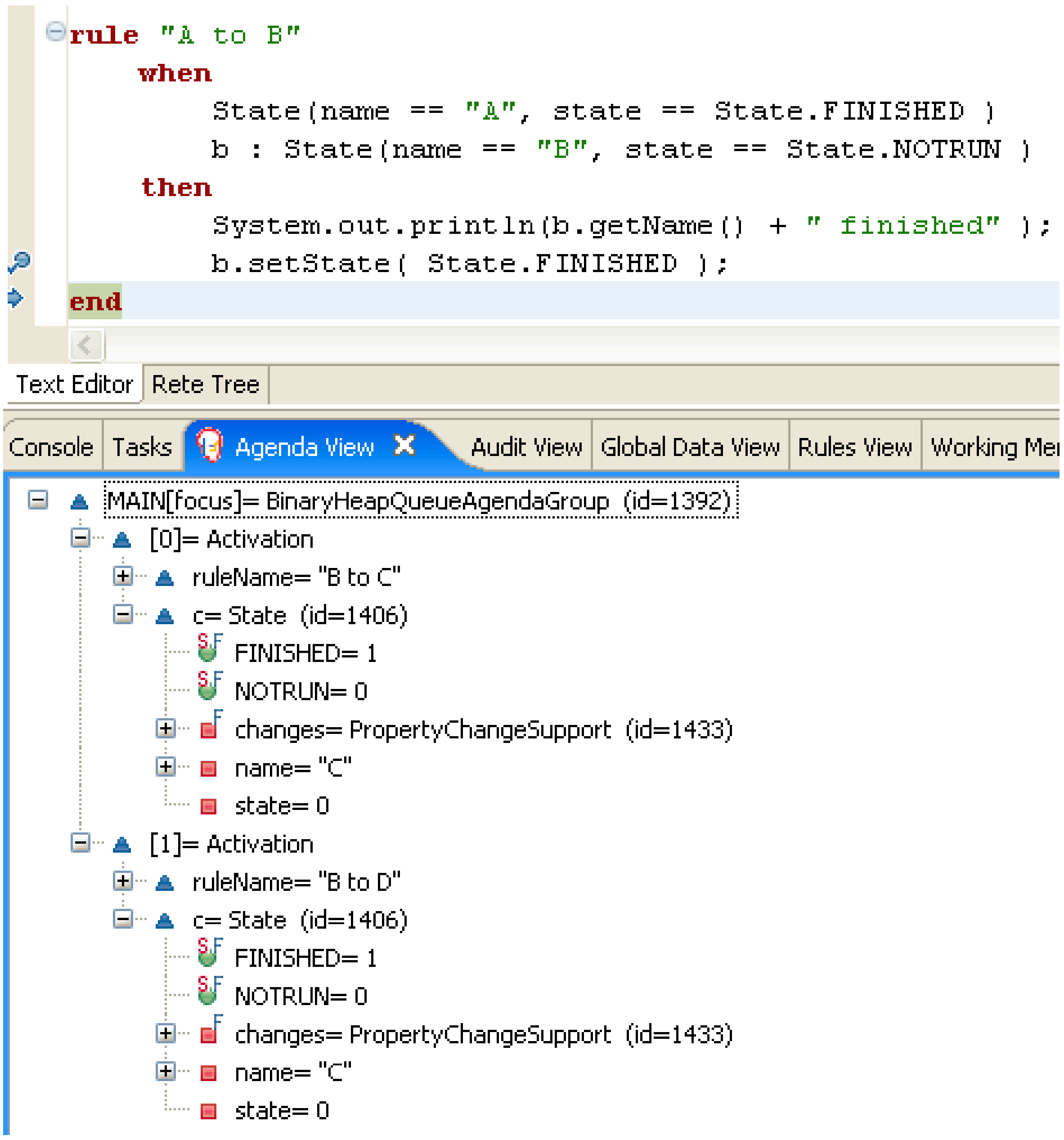
Example 8.11. Rule "Bootstrap"

The execution of "Bootstrap" rule changes the state of "A" to "FINISHED", that in turn activates the "A to B" rule.

```
rule "A to B"
when
 State(name == "A", state == State.FINISHED )
 b : State(name == "B", state == State.NOTRUN )
then
 System.out.println(b.getName() + " finished" );
 b.setState( State.FINISHED );
end
```

Example 8.12. Rule "A to B"

The execution of "A to B" rule changes the state of "B" to "FINISHED", which activates both rules "B to C" and "B to D", placing both Activations onto the Agenda. In this moment the two rules may fire and are said to be in conflict. The conflict resolution strategy allows the engine's Agenda to decide which rule to fire. As the "B to C" rule has a *higher salience value* (10 versus the default salience value of 0), it fires first, modifying the "C" object to state "FINISHED". The Audit view above shows the modification of the State object in the rule "A to B" which results in two highlighted activations being in conflict. The Agenda view can also be used to investigate the state of the Agenda, debug points can be placed in the rules themselves and the Agenda view opened; the screen shot below shows the break point in the rule "A to B" and the state of the Agenda with the two conflicting rules.



```
rule "A to B"
    when
        State(name == "A", state == State.FINISHED )
        b : State(name == "B", state == State.NOTRUN )
    then
        System.out.println(b.getName() + " finished" );
        b.setState( State.FINISHED );
end
```

```
MAIN[focus]= BinaryHeapQueueAgendaGroup  (id=1392)
    [0]= Activation
        ruleName= "B to C"
        c= State  (id=1406)
            FINISHED= 1
            NOTRUN= 0
            changes= PropertyChangeSupport  (id=1433)
            name= "C"
            state= 0
    [1]= Activation
        ruleName= "B to D"
        c= State  (id=1406)
            FINISHED= 1
            NOTRUN= 0
            changes= PropertyChangeSupport  (id=1433)
            name= "C"
            state= 0
```

Figure 8.5. Agenda View

```
rule "B to C"
 salience 10
when
 State(name == "B", state == State.FINISHED )
 c : State(name == "C", state == State.NOTRUN )
then
 System.out.println(c.getName() + " finished" );
 c.setState( State.FINISHED );
end
```

Example 8.13. Rule "B to C"

The "B to D" rule fires last, modifying the "D" object to state "FINISHED".

```
rule "B to D"
when
 State(name == "B", state == State.FINISHED )
 d : State(name == "D", state == State.NOTRUN )
then
 System.out.println(d.getName() + " finished" );
 d.setState( State.FINISHED );
end
```

Example 8.14. Rule "B to D"

There are no more rules to execute and so the engine stops.

Another notable concept in this example is the use of *dynamic facts*, which is the PropertyChangeListener part. As mentioned previously in the documentation, in order for the engine to see and react to fact's properties change, the application must tell the engine that changes occurred. This can be done explicitly in the rules, by calling the *update()* memory action, or implicitly by letting the engine know that the facts implement PropertyChangeSupport as defined by the *Javabeans* specification. This example demonstrates how to use PropertyChangeSupport to avoid the need for explicit update() calls in the rules. To make use of this feature, make sure your facts implement the PropertyChangeSupport as the org.drools.example.State class does and use the following code to insert the facts into the working memory:

```
// By setting dynamic to TRUE, Drools will use JavaBean
// PropertyChangeListeners so you don't have to call update().
final boolean dynamic = true;
session.insert( fact, dynamic );
```

Example 8.15. Inserting a Dynamic Fact

When using PropertyChangeListeners, each setter must implement extra code to do the notification. Here is the state setter for the org.drools.examples.State class:

```
public void setState(final int newState) {
    int oldState = this.state;
    this.state = newState;
    this.changes.firePropertyChange( "state",oldState,newState );
}
```

Example 8.16. Setter Example with PropertyChangeSupport

There are two other State examples: StateExampleUsingAgendGroup and StateExampleWithDynamicRules. Both execute from A to B to C to D, as just shown. StateExampleUsingAgendGroup uses agenda-groups to control the rule conflict and which one fires first. StateExampleWithDynamicRules shows how an additional rule can be added to an already running WorkingMemory with all the existing data applying to it at runtime.

Agenda groups are a way to partition the agenda into groups and controlling which groups can execute. All rules by default are in the "MAIN" agenda group, by simply using the "agenda-group" attribute you specify a different agenda group for the rule. A working memory initially only has focus on the "MAIN" agenda group, only when other groups are given the focus can their rules fire. This can be achieved by either using the method setFocus() or the rule attribute "auto-focus". "auto-focus" means that the rule automatically sets the focus to it's agenda group when the rule is matched and activated. It is this "auto-focus" that enables "B to C" to fire before "B to D".

```
rule "B to C"
 agenda-group "B to C"
 auto-focus true
when
 State(name == "B", state == State.FINISHED )
 c : State(name == "C", state == State.NOTRUN )
then
 System.out.println(c.getName() + " finished" );
 c.setState( State.FINISHED );
 drools.setFocus( "B to D" );
end
```

Example 8.17. Agenda Group State Example: Rule "B to C"

The rule "B to C" calls "drools.setFocus("B to D");" which gives the agenda group "B to D" focus allowing its active rules to fire; which allows the rule "B to D" to fire.

```
rule "B to D"
 agenda-group "B to D"
when
 State(name == "B", state == State.FINISHED )
 d : State(name == "D", state == State.NOTRUN )
then
 System.out.println(d.getName() + " finished" );
 d.setState( State.FINISHED );
end
```

Example 8.18. Agenda Group State Example: Rule "B to D"

The example StateExampleWithDynamicRules adds another rule to the RuleBase after fireAllRules(), the rule it adds is just another State transition.

```
rule "D to E"
when
 State(name == "D", state == State.FINISHED )
 e : State(name == "E", state == State.NOTRUN )
then
 System.out.println(e.getName() + " finished" );
 e.setState( State.FINISHED );
end
```

Example 8.19. Dynamic State Example: Rule "D to E"

It gives the following expected output:

```
A finished
B finished
C finished
D finished
E finished
```

Example 8.20. Dynamic Sate Example Output

8.3. Fibonacci Example

Name:	Fibonacci
Main class:	**`org.drools.examples.FibonacciExample`**
Type:	java application
Rules file:	**`Fibonacci.drl`**
Objective:	Demonsrate Recursion, 'not' CEs and Cross Product Matching.

The Fibonacci Numbers *http://en.wikipedia.org/wiki/Fibonacci_number*, invented by Leonardo of Pisa *http://en.wikipedia.org/wiki/Fibonacci*, are obtained by starting with 0 and 1, and then produce the next Fibonacci number by adding the two previous Fibonacci numbers. The first Fibonacci numbers for n = 0, 1,... are: * 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987, 1597, 2584, 4181, 6765, 10946... The Fibonacci Example demonstrates recursion and conflict resolution with Salience values.

A single fact Class is used in this example: Fibonacci. It has two fields, sequence and value. The sequence field is used to indicate the position of the object in the Fibonacci number sequence and the value field shows the value of that Fibonacci object for that sequence position.

```
public static class Fibonacci
{
    private int  sequence;
    private long value;

    ... setters and getters go here...
}
```

Example 8.21. Fibonacci Class

Execute the example:

1. Open the class **`org.drools.examples.FibonacciExample`** in your Eclipse IDE
2. Right-click the class an select Run as... > Java application

Eclipse shows the following output in its console, "...snip..." shows repeated bits removed to save space:

```
recurse for 50
recurse for 49
recurse for 48
recurse for 47
...snip...
recurse for 5
recurse for 4
recurse for 3
recurse for 2
1 == 1
2 == 1
3 == 2
4 == 3
5 == 5
6 == 8
...snip...
47 == 2971215073
48 == 4807526976
49 == 7778742049
50 == 12586269025
```

Example 8.22. Console Output

To kick this off from java we only insert a single Fibonacci object, with a sequence of 50, a recurse rule is then used to insert the other 49 Fibonacci objects. This example does not use PropertyChangeSupport and uses the MVEL dialect, this means we can use the *modify* keyword, which allows a block setter action which also notifies the engine of changes.

```
ksession.insert( new Fibonacci( 50 ) );
ksession.fireAllRules();
```

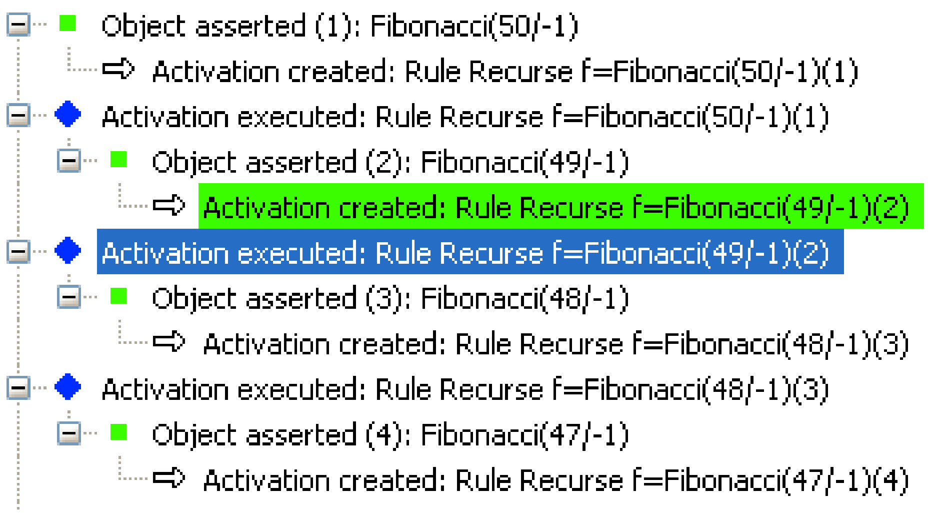
Example 8.23. Fibonacci Example Execution

The recurse rule is very simple, it matches each asserted Fibonacci object with a value of -1, it then creates and asserts a new Fibonacci object with a sequence of one less than the currently matched object. Each time a Fibonacci object is added, as long as one with a "sequence == 1" does not exist, the rule re-matches again and fires; causing the recursion. The 'not' conditional element is used to stop the rule matching once we have all 50 Fibonacci objects in memory. The rule also has a salience value, this is because we need to have all 50 Fibonacci objects asserted before we execute the Bootstrap rule.

```
rule Recurse
 salience 10
when
 f : Fibonacci ( value == -1 )
 not ( Fibonacci ( sequence == 1 ) )
then
 insert( new Fibonacci( f.sequence - 1 ) );
 System.out.println( "recurse for " + f.sequence );
end
```

Example 8.24. Rule "Recurse"

The audit view shows the original assertion of the Fibonacci object with a sequence of 50, this was done from Java land. From there the audit view shows the continual recursion of the rule, each asserted Fibonacci causes the "Recurse" rule to become activate again, which then fires.



```
Object asserted (1): Fibonacci(50/-1)
  Activation created: Rule Recurse f=Fibonacci(50/-1)(1)
Activation executed: Rule Recurse f=Fibonacci(50/-1)(1)
  Object asserted (2): Fibonacci(49/-1)
    Activation created: Rule Recurse f=Fibonacci(49/-1)(2)
Activation executed: Rule Recurse f=Fibonacci(49/-1)(2)
  Object asserted (3): Fibonacci(48/-1)
    Activation created: Rule Recurse f=Fibonacci(48/-1)(3)
Activation executed: Rule Recurse f=Fibonacci(48/-1)(3)
  Object asserted (4): Fibonacci(47/-1)
    Activation created: Rule Recurse f=Fibonacci(47/-1)(4)
```

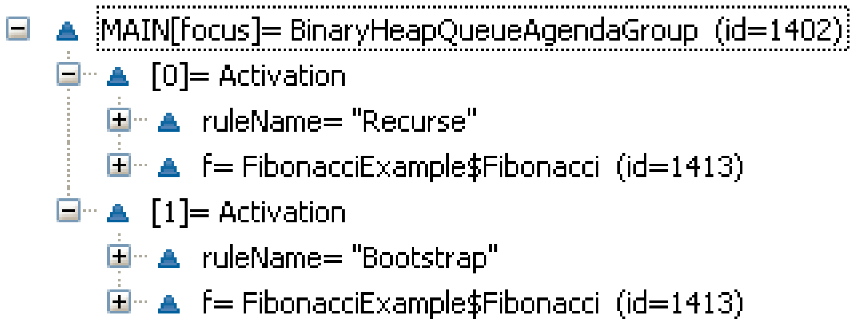
Figure 8.6. "Recurse" Audit View 1

When a Fibonacci with a sequence of 2 is asserted, the "Bootstrap" rule is matched and activated along with the "Recurse" rule.

```
rule Bootstrap
when
 f : Fibonacci( sequence == 1 || == 2, value == -1 )
 // this is a multi-restriction || on a single field
then
 modify ( f ){ value = 1 };
 System.out.println( f.sequence + " == " + f.value );
end
```

Example 8.25. Rule "Bootstrap"

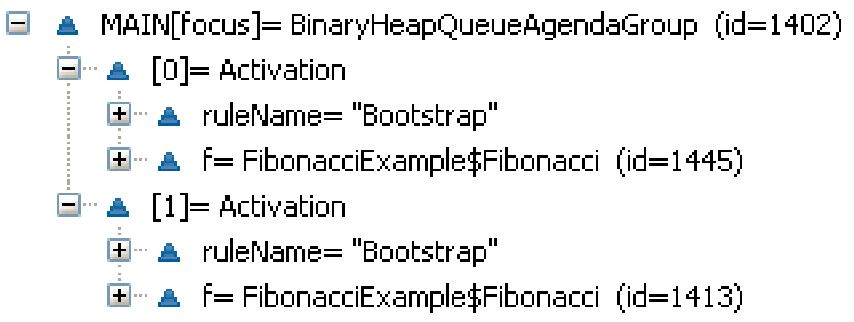
At this point the Agenda looks like the following figure. However the "Bootstrap" rule does not fire as the "Recurse" rule has a higher salience.



```
MAIN[focus]= BinaryHeapQueueAgendaGroup  (id=1402)
  [0]= Activation
    ruleName= "Recurse"
    f= FibonacciExample$Fibonacci  (id=1413)
  [1]= Activation
    ruleName= "Bootstrap"
    f= FibonacciExample$Fibonacci  (id=1413)
```

Figure 8.7. "Recurse" Agenda View 1

When a Fibonacci with a sequence of 1 is asserted the "Bootstrap" rule is matched again, causing two activations for this rule; note that the "Recurse" rule does not match and activate because the 'not conditional' element stops the rule matching when a Fibonacci with a sequence of 1 exists.



```
MAIN[focus]= BinaryHeapQueueAgendaGroup  (id=1402)
  [0]= Activation
    ruleName= "Bootstrap"
    f= FibonacciExample$Fibonacci  (id=1445)
  [1]= Activation
    ruleName= "Bootstrap"
    f= FibonacciExample$Fibonacci  (id=1413)
```

Figure 8.8. "Recurse" Agenda View 2

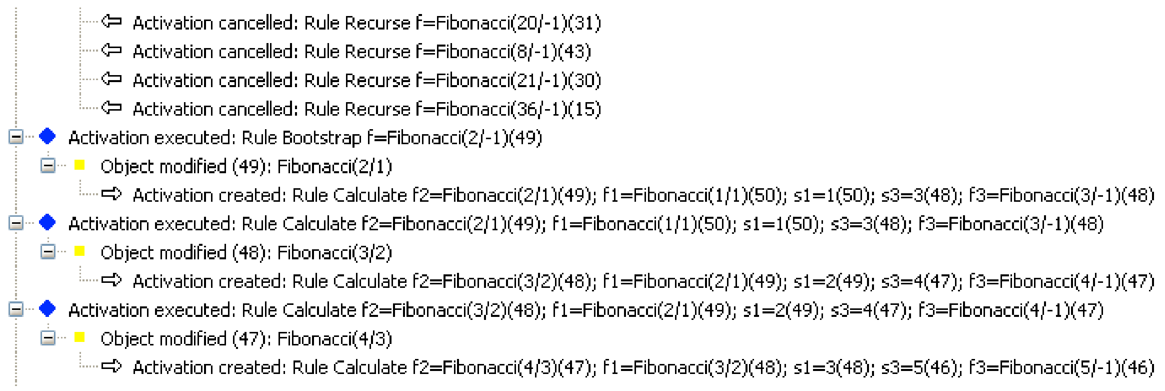
Once we have two Fibonacci objects both with values not equal to -1, the "calculate" rule is able to match; remember it was the "Bootstrap" rule that set the Fibonacci's with sequences 1 and 2 to values of 1. At this point we have 50 Fibonacci objects in the Working Memory and we some how need to select the correct ones to calculate each of their values in turn. With three Fibonacci patterns in a rule with no field constraints to correctly constrain the available cross products, we have 50x50x50 possible permutations, that is 125,000 possible rule firings.

The "Calculate" rule uses the field constraints to correctly constraint the thee Fibonacci patterns and in the correct order; this technique is called "cross product matching". The first pattern finds any Fibonacci with a value != -1 and binds both the pattern and the field. The second Fibonacci does too but it adds an additional field constraint to make sure that its sequence is one greater than the Fibonacci bound to f1. When this rule first fires we know that only sequences 1 and 2 have values of 1 and the two constraints ensure that f1 references sequence 1 and f2 references sequence2. The final pattern finds the Fibonacci of a value == -1 with a sequence one greater than f2. At this point we have three Fibonacci objects correctly selected from the available cross products and we can do the maths calculating the value for Fibonacci sequence = 3.

```
rule Calculate
when
 f1 : Fibonacci( s1 : sequence, value != -1 )
 // here we bind sequence
 f2 : Fibonacci( sequence == (s1 + 1 ), value != -1 )
 // here we don't, just to demonstrate the different way
 // bindings can be used
 f3 : Fibonacci( s3 : sequence == (f2.sequence + 1 ), value == -1 )
then
 modify ( f3 ) { value = f1.value + f2.value };
 System.out.println( s3 + " == " + f3.value );
 // see how you can access pattern and field  bindings
end
```

Example 8.26. Rule "Calculate"

The MVEL modify keyword updated the value of the Fibonacci object bound to f3. This means we have a new Fibonacci object with a value != -1, which allows the "Calculate" rule to rematch and calculate the next Fibonacci number. The Audit view below shows the how the firing of the last "Bootstrap" modifies the Fibonacci object enabling the "Calculate" rule to match, which then modifies another Fibonacci object allowing the "Calculate" rule to rematch. This continues till the value is set for all Fibonacci objects.



```
        Activation cancelled: Rule Recurse f=Fibonacci(20/-1)(31)
        Activation cancelled: Rule Recurse f=Fibonacci(8/-1)(43)
        Activation cancelled: Rule Recurse f=Fibonacci(21/-1)(30)
        Activation cancelled: Rule Recurse f=Fibonacci(36/-1)(15)
Activation executed: Rule Bootstrap f=Fibonacci(2/-1)(49)
    Object modified (49): Fibonacci(2/1)
        Activation created: Rule Calculate f2=Fibonacci(2/1)(49); f1=Fibonacci(1/1)(50); s1=1(50); s3=3(48); f3=Fibonacci(3/-1)(48)
Activation executed: Rule Calculate f2=Fibonacci(2/1)(49); f1=Fibonacci(1/1)(50); s1=1(50); s3=3(48); f3=Fibonacci(3/-1)(48)
    Object modified (48): Fibonacci(3/2)
        Activation created: Rule Calculate f2=Fibonacci(3/2)(48); f1=Fibonacci(2/1)(49); s1=2(49); s3=4(47); f3=Fibonacci(4/-1)(47)
Activation executed: Rule Calculate f2=Fibonacci(3/2)(48); f1=Fibonacci(2/1)(49); s1=2(49); s3=4(47); f3=Fibonacci(4/-1)(47)
    Object modified (47): Fibonacci(4/3)
        Activation created: Rule Calculate f2=Fibonacci(4/3)(47); f1=Fibonacci(3/2)(48); s1=3(48); s3=5(46); f3=Fibonacci(5/-1)(46)
```

Figure 8.9. "Bootstrap" Audit View 1

8.4. Banking Tutorial

Name:	Banking Tutorial
Main class:	**`org.drools.tutorials.banking`**
Type:	java application

Rules file:	**`org.drools.tutorials.banking.`**
Objective:	Increase knowledge of pattern matching, basic sorting and calculation rules.

This tutorial will demonstrate the process of developing a complete personal banking application that will handle credits, debits, currencies and that will use a set of design patterns that have been created for the process. In order to make the examples documented here clear and modular, I will try and steer away from re-visiting existing code to add new functionality, and will instead extend and inject where appropriate.

The RuleRunner class is a simple harness to execute one or more drl's against a set of data.

```
public class RuleRunner {

 public RuleRunner() {}

 public void runRules(String[] rules,Object[] facts) throws Exception
 {
  KnowledgeBase kbase = KnowledgeBaseFactory.newKnowledgeBase();
  KnowledgeBuilder kbuilder =
   KnowledgeBuilderFactory.newKnowledgeBuilder();

  for ( int i = 0; i < rules.length; i++ ) {
   String ruleFile = rules[i];
   System.out.println( "Loading file: " + ruleFile );
   kbuilder.add( ResourceFactory.newClassPathResource
    ( ruleFile, RuleRunner.class ),ResourceType.DRL );
  }

    Collection<KnowledgePackage> pkgs = kbuilder.getKnowledgePackages();
       kbase.addKnowledgePackages( pkgs );
       StatefulKnowledgeSession ksession =
   kbase.newStatefulKnowledgeSession();

  for ( int i = 0; i < facts.length; i++ ) {
   Object fact = facts[i];
   System.out.println( "Inserting fact: " + fact );
   ksession.insert( fact );
  }
  ksession.fireAllRules();
 }
}
```

Example 8.27. RuleRunner

It compiles the Packages and creates the KnowledgeBase for each execution, this allows us to easy execute each scenario and see the outputs. In reality, this is not a good solution for a production system where the KnowledgeBase should be built just once and cached, but for the purposes of this tutorial it shall suffice.

This is our first **`Example1.java`** class that loads and executes a single drl file "Example.drl" but inserts no data.

```
public class Example1
{
 public static void main(String[] args) throws Exception
 {
  new RuleRunner().runRules(new String[]{"Example1.drl"},
   new Object[0] );
 }
}
```

Example 8.28. Java Example1

This is the first simple rule to execute. It has a single "eval" condition that will always be true, therefore will always match and fire.

```
rule "Rule 01"
when
 eval (1==1)
then
 System.out.println("Rule 01 Works");
end
```

Example 8.29. Rule Example1

The output for the rule is described in the following example. The rule matches and executes the single print statement.

```
Loading file: Example1.drl
Rule 01 Works
```

Example 8.30. Output Example1

The next step is to assert some simple facts and print them out.

```
public class Example2
{
 private static Integer wrap(int i) {return new Integer(i);}

 public static void main(String[] args) throws Exception
 {
  Number[] numbers = new Number[] {wrap(3), wrap(1), wrap(4),
   wrap(1), wrap(5)};
  new RuleRunner().runRules(new String[] { "Example2.drl" },numbers);
 }
}
```

Example 8.31. Java Example2

This doesn’t use any specific facts, but instead asserts a set of java.lang.Integer’s. This is not considered "best practice" as a number of a collection is not a fact or a thing. A bank acount has a number, its balance, therefore the Account is the fact. Initially, asserting Integers shall suffice for demonstration purposes as the complexity is built up.

Now we will create a simple rule to print out these numbers.

```
rule "Rule 02"
when
 Number( $intValue : intValue )
then
 System.out.println("Number found with value: " + $intValue);
end
```

Example 8.32. Rule Example2

Once again, this rule does nothing special. It identifies any facts that are Numbers and prints out the values. Note the use of interfaces here; we inserted Integers but the pattern matching engine is able to match the interfaces and super classes of the asserted objects.

The output shows the drl being loaded, the facts inserted and then the matched and fired rules. We can see that each inserted number is matched and fired and thus printed.

```
Loading file: Example2.drl
Inserting fact: 3
Inserting fact: 1
Inserting fact: 4
Inserting fact: 1
Inserting fact: 5
Number found with value: 5
Number found with value: 1
Number found with value: 4
Number found with value: 1
Number found with value: 3
```

Example 8.33. Output Example2

There are better ways to sort numbers, however we will need to apply some cash flows in date order when we start looking at banking rules. Let’s look at a simple rule-based example.

```
public class Example3
{
    private static Integer wrap(int i) {return new Integer(i);}

 public static void main(String[] args) throws Exception
 {
  Number[] numbers = new Number[] {wrap(3), wrap(1), wrap(4),
   wrap(1), wrap(5)};

  new RuleRunner().runRules(new String[]{ "Example3.drl"},numbers);
 }
}
```

Example 8.34. Java Example3

Again we insert our Integers as before, however this time the rule is slightly different:

```
rule "Rule 03"
when
 $number : Number( )
 not Number( intValue &lt; $number.intValue )
then
 System.out.println("Number found with value: "+$number.intValue() );
 retract( $number );
end
```

Example 8.35. Rule Example3

The first line of the rules identifies a Number and extracts the value. The second line ensures that a smaller number than the one found does not exist. By executing this rule, we might expect to find only one number - the smallest in the set. However, the retraction of the number after it has been printed means that the smallest number has been removed, revealing the next smallest number, and so on.

The generated output is showing in the following example. Note the numbers are now sorted numerically.

```
Loading file: Example3.drl
Inserting fact: 3
Inserting fact: 1
Inserting fact: 4
Inserting fact: 1
Inserting fact: 5
Number found with value: 1
Number found with value: 1
Number found with value: 3
Number found with value: 4
Number found with value: 5
```

Example 8.36. Output Example3

Now we want to start moving towards our personal accounting rules. The first step is to create a Cashflow POJO.

```
public class Cashflow
{
    private Date   date;
    private double amount;

    public Cashflow() {}

    public Cashflow(Date date,double amount)
 {
    this.date = date;this.amount = amount;
 }

    public Date getDate()   { return date;  }
    public void setDate(Date date) { this.date = date; }

    public double getAmount()     { return amount;   }
    public void setAmount(double amount) { this.amount = amount; }

    public String toString()
 {
  return "Cashflow[date=" + date + ",amount=" + amount + "]";
 }
}
```

Example 8.37. Class Cashflow

The Cashflow has two simple attributes: a date and an amount. I have added a toString method to print it and overloaded the constructor to set the values. The following example inserts 5 Cashflow objects with varying dates and amounts.

```
public class Example4
{
    public static void main(String[] args) throws Exception
 {
        Object[] cashflows = {
            new Cashflow(new SimpleDate("01/01/2007"), 300.00),
            new Cashflow(new SimpleDate("05/01/2007"), 100.00),
            new Cashflow(new SimpleDate("11/01/2007"), 500.00),
            new Cashflow(new SimpleDate("07/01/2007"), 800.00),
            new Cashflow(new SimpleDate("02/01/2007"), 400.00),
        };

        new RuleRunner().runRules(new String[] {"Example4.drl"},cashflows;
 }
}
```

Example 8.38. Java Example4

SimpleDate is a simple class that extends Date and takes a String as input. It allows for pre-formatted Data classes, for convienience. The code is listed in the following example.

```
public class SimpleDate extends Date
{
    private static final SimpleDateFormat format =
     new SimpleDateFormat("dd/MM/yyyy");

    public SimpleDate(String datestr) throws Exception
 {
        setTime(format.parse(datestr).getTime());
 }
}
```

Example 8.39. Java SimpleDate

Now, let's look at rule04.drl to see how we print the sorted Cashflows:

```
rule "Rule 04"
when
 $cashflow : Cashflow( $date : date, $amount : amount )
 not Cashflow( date &lt; $date)
then
 System.out.println("Cashflow: "+$date+" :: "+$amount);
 retract($cashflow);
end
```

Example 8.40. Rule Example4

Here, we identify a Cashflow and extract the date and the amount. In the second line of the rules we ensure that a Cashflow does not exist with an earlier date than the one found. In the consequences, we print the Cashflow that satisfies the rules and then retract it, making way for the next earliest Cashflow. The output is described in the following example.

```
Loading file: Example4.drl
Inserting fact: Cashflow[date=Mon Jan 01 00:00:00 GMT 2007,amount=300.0]
Inserting fact: Cashflow[date=Fri Jan 05 00:00:00 GMT 2007,amount=100.0]
Inserting fact: Cashflow[date=Thu Jan 11 00:00:00 GMT 2007,amount=500.0]
Inserting fact: Cashflow[date=Sun Jan 07 00:00:00 GMT 2007,amount=800.0]
Inserting fact: Cashflow[date=Tue Jan 02 00:00:00 GMT 2007,amount=400.0]
Cashflow: Mon Jan 01 00:00:00 GMT 2007 :: 300.0
Cashflow: Tue Jan 02 00:00:00 GMT 2007 :: 400.0
Cashflow: Fri Jan 05 00:00:00 GMT 2007 :: 100.0
Cashflow: Sun Jan 07 00:00:00 GMT 2007 :: 800.0
Cashflow: Thu Jan 11 00:00:00 GMT 2007 :: 500.0
```

Example 8.41. Output Example4

Here we extend our Cashflow to give a TypedCashflow which can be CREDIT or DEBIT. Ideally, we would just add this to the Cashflow type, but so that we can keep all the examples simple, we will go with the extensions.

```
public class TypedCashflow extends Cashflow {
    public static final int CREDIT = 0;
    public static final int DEBIT  = 1;

    private int type;

    public TypedCashflow() { }

    public TypedCashflow(Date date, int type, double amount)
 {
        super( date, amount );
        this.type = type;
    }

    public int getType()
 {
        return type;
    }

    public void setType(int type)
 {
        this.type = type;
    }

    public String toString()
 {
        return "TypedCashflow[date=" + getDate()
   + ",type=" + (type == CREDIT ? "Credit" : "Debit")
   + ",amount=" + getAmount()
   + "]";
    }
}
```

There are lots of ways to improve this code, but for the sake of the example this will do.

Now lets create the Example5 runner.

```
public class Example5
{
    public static void main(String[] args) throws Exception
 {
        Object[] cashflows = {
            new TypedCashflow(new SimpleDate("01/01/2007"),
         TypedCashflow.CREDIT, 300.00),
            new TypedCashflow(new SimpleDate("05/01/2007"),
                                TypedCashflow.CREDIT, 100.00),
            new TypedCashflow(new SimpleDate("11/01/2007"),
                                TypedCashflow.CREDIT, 500.00),
            new TypedCashflow(new SimpleDate("07/01/2007"),
                                TypedCashflow.DEBIT, 800.00),
            new TypedCashflow(new SimpleDate("02/01/2007"),
                                TypedCashflow.DEBIT, 400.00),
        };

        new RuleRunner().runRules(
   new String[] { "Example5.drl" }, cashflows );
    }
}
```

Example 8.42. Java Example5

Here, we simply create a set of Cashflows which are either CREDIT or DEBIT Cashflows and supply them and rule05.drl to the RuleEngine.

Now, let’s look at rule0 Example5.drl to see how we print the sorted Cashflows.

```

rule "Rule 05"
when
 $cashflow : TypedCashflow( $date : date, $amount : amount,
  type == TypedCashflow.CREDIT )
 not TypedCashflow( date &lt; $date, type == TypedCashflow.CREDIT )
then
 System.out.println("Credit: "+$date+" :: "+$amount);
 retract($cashflow);
end
```

Example 8.43. Rule Example5

Here, we identify a Cashflow with a type of CREDIT and extract the date and the amount. In the second line of the rules we ensure that there is not a Cashflow of type CREDIT with an earlier date than the one found. In the consequences, we print the Cashflow that satisfies the rules and then retract it, making way for the next earliest Cashflow of type CREDIT.

The generated output is described in the following example.

```
Loading file: Example5.drl
Inserting fact: TypedCashflow[date=Mon Jan 01 00:00:00 GMT
 2007,type=Credit,amount=300.0]
Inserting fact: TypedCashflow[date=Fri Jan 05 00:00:00 GMT
 2007,type=Credit,amount=100.0]
Inserting fact: TypedCashflow[date=Thu Jan 11 00:00:00 GMT
 2007,type=Credit,amount=500.0]
Inserting fact: TypedCashflow[date=Sun Jan 07 00:00:00 GMT
 2007,type=Debit,amount=800.0]
Inserting fact: TypedCashflow[date=Tue Jan 02 00:00:00 GMT
 2007,type=Debit,amount=400.0]
Credit: Mon Jan 01 00:00:00 GMT 2007 :: 300.0
Credit: Fri Jan 05 00:00:00 GMT 2007 :: 100.0
Credit: Thu Jan 11 00:00:00 GMT 2007 :: 500.0
```

Example 8.44. Output Example5

Here we are going to process both Credits and Debits on 2 bank accounts to calculate the account balance. In order to do this, I am going to create two separate Account Objects and inject them into the Cashflows before passing them to the Rule Engine. The reason for this is to provide easy access to the correct bank accounts without having to resort to Helper classes. Let’s take a look at the Account class first. This is a simple POJO with an account number and balance:

```
public class Account
{
    private long   accountNo;
    private double balance = 0;

    public Account() { }

    public Account(long accountNo)
 {
        this.accountNo = accountNo;
    }

    public long getAccountNo()
 {
        return accountNo;
    }

    public void setAccountNo(long accountNo)
 {
        this.accountNo = accountNo;
    }

    public double getBalance()
 {
        return balance;
    }
    public void setBalance(double balance)
 {
        this.balance = balance;
    }

    public String toString()
 {
        return "Account[" + "accountNo=" + accountNo
   + ",balance=" + balance + "]";
    }
}
```

Now let’s extend our TypedCashflow to give AllocatedCashflow (allocated to an account).

```

public class AllocatedCashflow extends TypedCashflow
{
 private Account account;

 public AllocatedCashflow() {}

 public AllocatedCashflow(Account account, Date date,
  int type, double amount)
 {
  super( date, type, amount );
  this.account = account;
 }

 public Account getAccount()
 {
  return account;
 }

 public void setAccount(Account account)
 {
  this.account = account;
 }

 public String toString()
 {
  return "AllocatedCashflow["
   + "account=" + account
   + ",date=" + getDate()
   + ",type=" + (getType() == CREDIT ? "Credit" : "Debit")
   + ",amount=" + getAmount()
   + "]";
 }
}
```

Example 8.45. Class AllocatedCashflow

Now, let's java code for Example5 execution. Here we create two Account objects and inject one into each cashflow as appropriate. For simplicity I have simply included them in the constructor.

```
public class Example6
{
 public static void main(String[] args) throws Exception
 {
  Account acc1 = new Account(1);
  Account acc2 = new Account(2);

  Object[] cashflows =
  {
   new AllocatedCashflow(acc1,new SimpleDate("01/01/2007"),
           TypedCashflow.CREDIT, 300.00),
   new AllocatedCashflow(acc1,new SimpleDate("05/02/2007"),
           TypedCashflow.CREDIT, 100.00),
   new AllocatedCashflow(acc2,new SimpleDate("11/03/2007"),
           TypedCashflow.CREDIT, 500.00),
   new AllocatedCashflow(acc1,new SimpleDate("07/02/2007"),
           TypedCashflow.DEBIT,  800.00),
   new AllocatedCashflow(acc2,new SimpleDate("02/03/2007"),
           TypedCashflow.DEBIT,  400.00),
   new AllocatedCashflow(acc1,new SimpleDate("01/04/2007"),
           TypedCashflow.CREDIT, 200.00),
   new AllocatedCashflow(acc1,new SimpleDate("05/04/2007"),
           TypedCashflow.CREDIT, 300.00),
   new AllocatedCashflow(acc2,new SimpleDate("11/05/2007"),
           TypedCashflow.CREDIT, 700.00),
   new AllocatedCashflow(acc1,new SimpleDate("07/05/2007"),
           TypedCashflow.DEBIT,  900.00),
   new AllocatedCashflow(acc2,new SimpleDate("02/05/2007"),
           TypedCashflow.DEBIT,  100.00)
  };

  new RuleRunner().runRules(new String[]{"Example6.drl"},cashflows);
 }
}
```

Example 8.46. Java Example5

Now, let's look at rule Example06.drl to see how we apply each cashflow in date order and calculate and print the balance.

```
rule "Rule 06 - Credit"
when
 $cashflow : AllocatedCashflow( $account : account,
  $date : date, $amount : amount, type==TypedCashflow.CREDIT )
 not AllocatedCashflow( account == $account, date < $date)
then
 System.out.println("Credit: " + $date + " :: " + $amount);
 $account.setBalance($account.getBalance()+$amount);
 System.out.println("Account: " + $account.getAccountNo() +
  " - new balance: " + $account.getBalance());
 retract($cashflow);
end

rule "Rule 06 - Debit"
when
 $cashflow : AllocatedCashflow( $account : account,
 $date : date, $amount : amount, type==TypedCashflow.DEBIT )
 not AllocatedCashflow( account == $account, date < $date)
then
 System.out.println("Debit: " + $date + " :: " + $amount);
 $account.setBalance($account.getBalance() - $amount);
 System.out.println("Account: " + $account.getAccountNo() +
  " - new balance: " + $account.getBalance());
 retract($cashflow);
end
```

Here, we have separate rules for Credits and Debits, however we do not specify a type when checking for earlier cashflows. This is so that all cashflows are applied in date order regardless of which type of cashflow type they are. In the rule section we identify the correct account to work with and in the consequences we update it with the cashflow amount.

```
Loading file: Example6.drl
Inserting fact:
 AllocatedCashflow[account=Account[accountNo=1,balance=0.0],date=Mon Jan 01
 00:00:00 GMT 2007,type=Credit,amount=300.0]
Inserting fact:
 AllocatedCashflow[account=Account[accountNo=1,balance=0.0],date=Mon Feb 05
 00:00:00 GMT 2007,type=Credit,amount=100.0]
Inserting fact:
 AllocatedCashflow[account=Account[accountNo=2,balance=0.0],date=Sun Mar 11
 00:00:00 GMT 2007,type=Credit,amount=500.0]
Inserting fact:
 AllocatedCashflow[account=Account[accountNo=1,balance=0.0],date=Wed Feb 07
 00:00:00 GMT 2007,type=Debit,amount=800.0]
Inserting fact:
 AllocatedCashflow[account=Account[accountNo=2,balance=0.0],date=Fri Mar 02
 00:00:00 GMT 2007,type=Debit,amount=400.0]
Inserting fact:
 AllocatedCashflow[account=Account[accountNo=1,balance=0.0],date=Sun Apr 01
 00:00:00 BST 2007,type=Credit,amount=200.0]
Inserting fact:
 AllocatedCashflow[account=Account[accountNo=1,balance=0.0],date=Thu Apr 05
 00:00:00 BST 2007,type=Credit,amount=300.0]
Inserting fact:
 AllocatedCashflow[account=Account[accountNo=2,balance=0.0],date=Fri May 11
 00:00:00 BST 2007,type=Credit,amount=700.0]
Inserting fact:
 AllocatedCashflow[account=Account[accountNo=1,balance=0.0],date=Mon May 07
 00:00:00 BST 2007,type=Debit,amount=900.0]
```

```
Inserting fact:
 AllocatedCashflow[account=Account[accountNo=2,balance=0.0],date=Wed May 02
 00:00:00 BST 2007,type=Debit,amount=100.0]
Debit: Fri Mar 02 00:00:00 GMT 2007 :: 400.0
Account: 2 - new balance: -400.0
Credit: Sun Mar 11 00:00:00 GMT 2007 :: 500.0
Account: 2 - new balance: 100.0
Debit: Wed May 02 00:00:00 BST 2007 :: 100.0
Account: 2 - new balance: 0.0
Credit: Fri May 11 00:00:00 BST 2007 :: 700.0
Account: 2 - new balance: 700.0
Credit: Mon Jan 01 00:00:00 GMT 2007 :: 300.0
Account: 1 - new balance: 300.0
Credit: Mon Feb 05 00:00:00 GMT 2007 :: 100.0
Account: 1 - new balance: 400.0
Debit: Wed Feb 07 00:00:00 GMT 2007 :: 800.0
Account: 1 - new balance: -400.0
Credit: Sun Apr 01 00:00:00 BST 2007 :: 200.0
Account: 1 - new balance: -200.0
Credit: Thu Apr 05 00:00:00 BST 2007 :: 300.0
Account: 1 - new balance: 100.0
Debit: Mon May 07 00:00:00 BST 2007 :: 900.0
Account: 1 - new balance: -800.0
```

8.5. Pricing Rule Decision Table Example

The Pricing Rule decision table demonstrates the use of a decision table in a spreadsheet (XLS format) in calculating the retail cost of an insurance policy. The purpose of the set of rules provided is to calculate a base price, and an additional discount for a car driver applying for a specific policy. The drivers age, history, and the policy type all contribute to what the basic premium is. An additional chunk of rules deals with refining this with a subtractive percentage discount.

Name:	Example Policy Pricing
Main class:	**`org.drools.examples.PricingRuleDTExample`**
Type:	java application
Rules file:	**`ExamplePolicyPricing.xls`**
Objective:	Demonstrate spreadsheet based decision tables.

8.5.1. Executing the Example

Open the PricingRuleDTExample.java and execute it as a Java application. It should produce the following console output.

```
Cheapest possible
BASE PRICE IS: 120
DISCOUNT IS: 20
```

The code to execute the example is very similar to the other examples. The rules are loaded, the facts inserted, and a stateless session is used. What is different is how the rules are added.

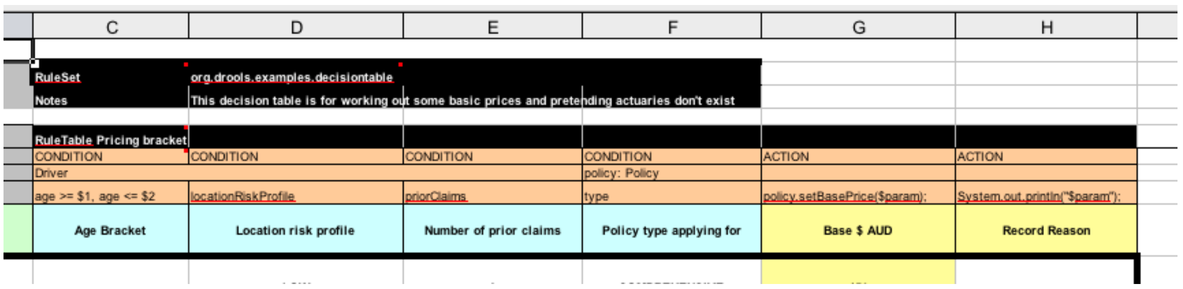
```
DecisionTableConfiguration dtableconfiguration =
 KnowledgeBuilderFactory.newDecisionTableConfiguration();
dtableconfiguration.setInputType( DecisionTableInputType.XLS );

KnowledgeBuilder kbuilder = KnowledgeBuilderFactory.newKnowledgeBuilder();

kbuilder.add( ResourceFactory.newClassPathResource(
 "ExamplePolicyPricing.xls", getClass() ),
 ResourceType.DTABLE, dtableconfiguration );
```

Note the use of the DecisionTableConfiguration class; it takes the DecisionTableInputType.XLS as input type. If you use the BRMS, all this is of course taken care of for you.

There are 2 facts used in this example: Driver, and Policy. Both are used with their default values. The Driver is 30 years old, has had no prior claims and currently has a risk profile of LOW. The Policy being applied for is COMPREHENSIVE, and the policy has not yet been approved.

8.5.2. The Decision Table

In this decision table, each row is a rule, and each column is a condition or an action.



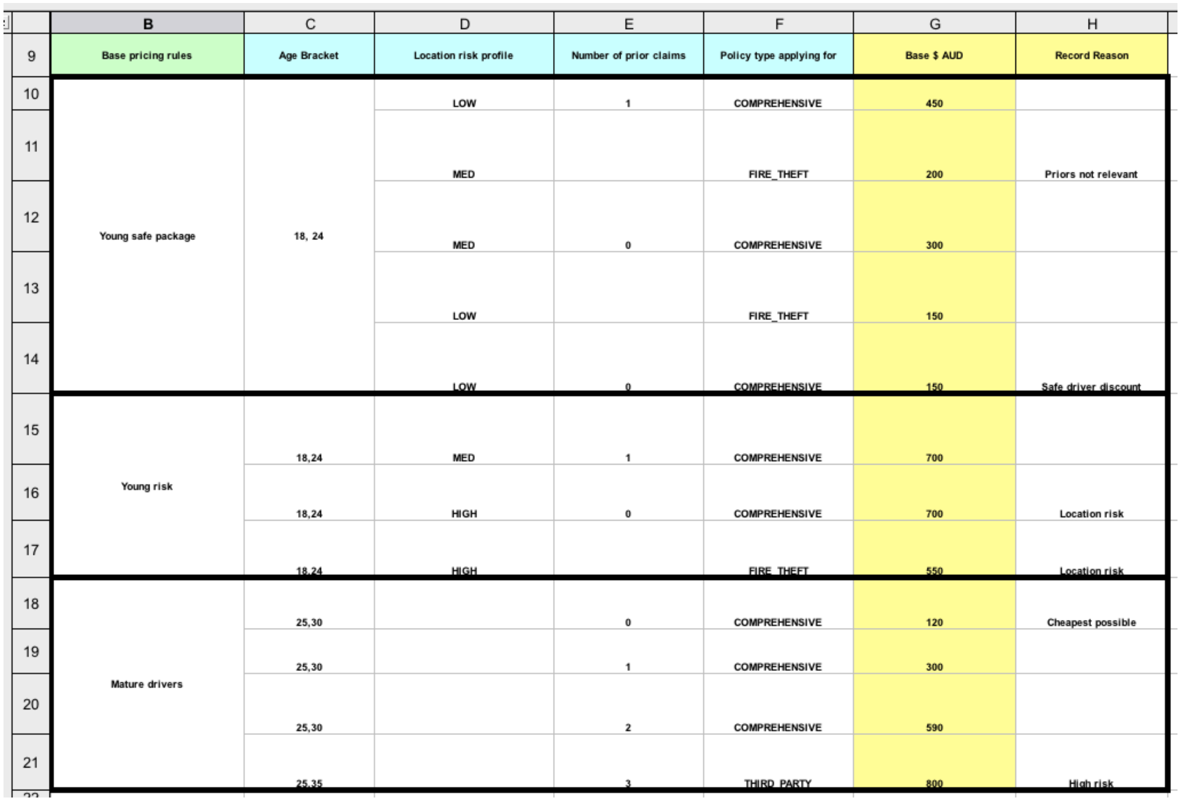
C	D	E	F	G	H
RuleSet	org.drools.examples.decisiontable				
Notes	This decision table is for working out some basic prices and pretending actuaries don't exist				
RuleTable Pricing bracket					
CONDITION	CONDITION	CONDITION	CONDITION	ACTION	ACTION
Driver			policy: Policy		
age >= $1, age <= $2	locationRiskProfile	priorClaims	type	policy.setBasePrice($param);	System.out.println("$param");
Age Bracket	Location risk profile	Number of prior claims	Policy type applying for	Base $ AUD	Record Reason

Figure 8.10. Decision Table Configuration

Referring to the above, we have the RuleSet declaration, which provides the package name. There are also other optional items you can have here, such as Variables for global variables, and Imports for importing classes. In this case, the namespace of the rules is the same as the fact classes we are using, so we can omit it.

Moving further down, we can see the RuleTable declaration. The name after this (Pricing bracket) is used as the prefix for all the generated rules. Below that, we have CONDITION or ACTION - this indicates the purpose of the column (ie does it form part of the condition, or an action of a rule that will be generated).

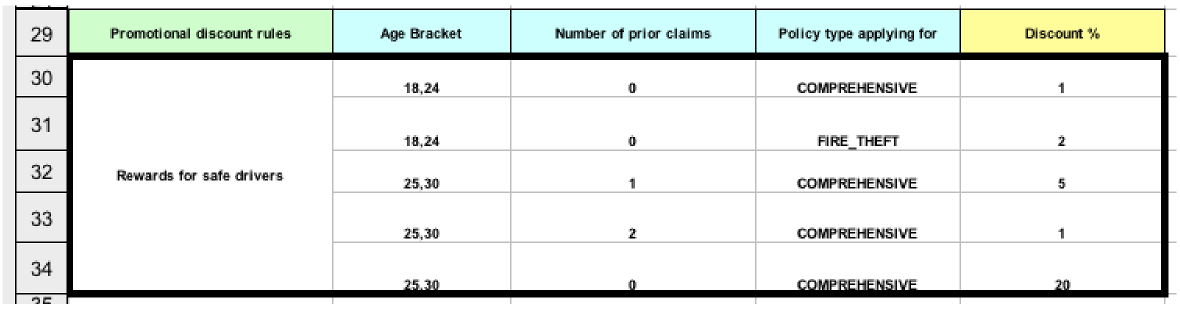
You can see there is a Driver which is spanned across 3 cells, this means the template expressions below it apply to that fact. So we look at the drivers age range (which uses $1 and $2 with comma separated values), locationRiskProfile, and priorClaims in the respective columns. In the action columns, we are setting the policy base price, and then logging a message.



	B	C	D	E	F	G	H
9	Base pricing rules	Age Bracket	Location risk profile	Number of prior claims	Policy type applying for	Base $ AUD	Record Reason
10	Young safe package	18, 24	LOW	1	COMPREHENSIVE	450	
11			MED		FIRE_THEFT	200	Priors not relevant
12			MED	0	COMPREHENSIVE	300	
13			LOW		FIRE_THEFT	150	
14			LOW	0	COMPREHENSIVE	150	Safe driver discount
15	Young risk	18,24	MED	1	COMPREHENSIVE	700	
16		18,24	HIGH	0	COMPREHENSIVE	700	Location risk
17		18,24	HIGH		FIRE_THEFT	550	Location risk
18	Mature drivers	25,30		0	COMPREHENSIVE	120	Cheapest possible
19		25,30		1	COMPREHENSIVE	300	
20		25,30		2	COMPREHENSIVE	590	
21		25,35		3	THIRD_PARTY	800	High risk

Figure 8.11. Base Price Calculation

In the example, we can see there are broad category brackets (indicated by the comment in the left most column). As we know, the details of our driver and their policy, we can tell (with a bit of thought) that they should match row number 18, as they have no prior accidents, and are 30 years old. This gives us a base price of 120.



29	Promotional discount rules	Age Bracket	Number of prior claims	Policy type applying for	Discount %
30	Rewards for safe drivers	18,24	0	COMPREHENSIVE	1
31		18,24	0	FIRE_THEFT	2
32		25,30	1	COMPREHENSIVE	5
33		25,30	2	COMPREHENSIVE	1
34		25,30	0	COMPREHENSIVE	20

Figure 8.12. Discount Calculation

Referring to the example, we can see if there are any discounts we can give our driver. Based on the Age bracket, number of prior claims, and the policy type, a discount is provided. In our case, the drive is 3, with no priors, and they are applying for COMPREHENSIVE. This means we can give a discount of 20%. Note that this is actually a separate table, but in the same worksheet: different templates apply.

It is important to note that decision tables generate rules; they aren't simply top down logic, but more a means to capture data that generate rules (this is a subtle difference that can be confusing). The evaluation of the rules is not "top down" necessarily, all the normal indexing and mechanics of the rule engine still apply.

8.6. Pet Store Example

Name:	Pet Store
Main class:	**`org.drools.examples.PetStore`**
Type:	java application
Rules file:	**`PetStore.drl`**
Objective:	Demonstrate use of Agenda Groups, Global Variables and integration with a GUI (including callbacks from within the Rules).

The Pet Store example shows how to integrate Rules with a GUI (in this case a Swing based Desktop application). Within the rules file, it shows how to use agenda groups and auto-focus to control which of a set of rules is allowed to fire at any given time. It also shows mixing of Java and MVEL dialects within the rules, the use of accumulate functions and calling of Java functions from within the ruleset.

Like the rest of the the samples, all the Java Code is contained in one file. The PetStore.java contains the following principal classes (in addition to several minor classes to handle Swing Events)

- *Petstore* - containing the main() method that we will look at shortly.
- *PetStoreUI* - responsible for creating and displaying the Swing based GUI. It contains several smaller classes , mainly for responding to various GUI events such as mouse and button clicks.
- *TabelModel* - for holding the table data. Think of it as a JavaBean that extends the Swing AbstractTableModel class.
- *CheckoutCallback* - Allows the GUI to interact with the Rules.
- *Ordershow* - the items that we wish to buy.
- *Purchase* - Details of the order and the products we are buying.
- *Product* - JavaBean holding details of the product available for purchase, and it's price.

Much of the Java code is either JavaBeans (simple enough to understand) or Swing based. We will touch on some Swing related points in the this tutorial , but a good place to get more Swing component information is available at the Sun Swing website*`http://java.sun.com/docs/books/tutorial/uiswing/`*.

There are two important Rules related pieces of Java code in *Petstore.java*.

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

kbuilder.add( ResourceFactory.newClassPathResource(
 "PetStore.drl", PetStore.class ),ResourceType.DRL );

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

//RuleB
Vector<Product> stock = new Vector<Product>();
stock.add( new Product( "Gold Fish",5 ) );
stock.add( new Product( "Fish Tank",25 ) );
stock.add( new Product( "Fish Food",2 ) );

//The callback is responsible for populating working memory and
// fireing all rules
PetStoreUI ui = new PetStoreUI( stock,new CheckoutCallback( kbase ) );
ui.createAndShowGUI();
```

Example 8.47. Creating the PetStore RuleBase - Extract from the PetStore.java main() Method

This code above loads the rules (drl) file from the classpath. Unlike other examples where the facts are asserted and fired straight away, this example defers this step to later. The way it does this is via the second last line where the PetStoreUI is created using a constructor the passes in the Vector called stock containing products, and an instance of the CheckoutCallback class containing the RuleBase that we have just loaded.

The actual Javacode that fires the rules is within the *CheckoutCallBack.checkout()* method. This is triggered (eventually) when the 'Checkout' button is pressed by the user.

```
public String checkout(JFrame frame, List<Product> items)
{
   Order order = new Order();

   //Iterate through list and add to cart
   for ( int i = 0; i < items.size(); i++ )
   {
      order.addItem( new Purchase( order, (Product) items.get( i ) ) );
   }

   //add the JFrame to the ApplicationData to allow for user interaction
   StatefulKnowledgeSession ksession = kbase.newStatefulKnowledgeSession();
   ksession.setGlobal( "frame", frame );
   ksession.setGlobal( "textArea", this.output );
   ksession.insert( new Product( "Gold Fish",
   ksession.insert( new Product( "Fish Tank", 25 ) );
   ksession.insert( new Product( "Fish Food", 2 ) );
   ksession.insert( new Product( "Fish Food Sample", 0 ) );
   ksession.insert( order );
   ksession.fireAllRules();
```

```
    //returns the state of the cart
    return order.toString();
}
```

Two items get passed into this method. A handle is passed to the JFrame Swing Component surrounding the output text frame (bottom of the GUI if / when you run the component). The second item is a list of order items; this comes from the TableModel the stores the information from the 'Table' area at the top right section of the GUI.

The *for()* loop transforms the list of order items coming from the GUI into the Order JavaBean (also contained in the PetStore.java file). It is possible to refer to the Swing dataset directly within the rules, but it is better coding practice to use Simple Java Objects. By using Simple Java Objects, we are not tied to Swing if we wanted to transform the sample into a Web application.

It is important to note that *all state in this example is stored in the Swing components, and that the rules are effectively stateless*. Each time the 'Checkout' button is pressed, this code copies the contents of the Swing *TableModel* into the Session/Working Memory.

Within this code, there are nine calls to the working memory. The first of these creates a new workingMemory (StatefulKnowledgeSession) from the Knowledgebase - remember that we passed in this Knowledgebase when we created the CheckoutCallBack class in the *main()* method. The next two calls pass in two objects that we will hold as Global variables in the rules - the Swing text area and Swing frame that we will use for writing messages later.

More inserts put information on products into the working memory, as well as the order list. The final call is the standard *fireAllRules()*. Next, we look at what this method causes to happen within the Rules file.

```

package org.drools.examples

import org.drools.WorkingMemory
import org.drools.examples.PetStore.Order
import org.drools.examples.PetStore.Purchase
import org.drools.examples.PetStore.Product
import java.util.ArrayList
import javax.swing.JOptionPane;

import javax.swing.JFrame

global JFrame frame
global javax.swing.JTextArea textArea

```

Example 8.48. Package, Imports, Globals, and Dialect - Extract (1) from the PetStore.drl

The first part of the *PetStore.drl* file contains the standard package and import statement to make various Java classes available to the rules. What is new are the two globals *frame and textArea*. These hold references to the Swing JFrame and Textarea components that were previous passed by the Java code calling the *setGlobal()* method. Unlike normal variables in Rules , which expire as soon as the rule has fired, Global variables retain their value for the lifetime of the (Stateful in this case) Session.

The following example is taken from the *end* of the PetStore.drl file. It contains two functions that are referenced by the rules that we will look at shortly.

```
function void doCheckout(JFrame frame, WorkingMemory workingMemory)
{
 Object[] options = {"Yes","No"};

 int n = JOptionPane.showOptionDialog(frame,
  "Would you like to checkout?","",
  JOptionPane.YES_NO_OPTION,
  JOptionPane.QUESTION_MESSAGE,
  null,options,options[0]);

 if (n == 0) {workingMemory.setFocus( "checkout" );}
}

function boolean requireTank(JFrame frame, WorkingMemory workingMemory,
 Order order, Product fishTank, int total)
{
 Object[] options = {"Yes","No"};

 int n = JOptionPane.showOptionDialog(frame,
  "Would you like to buy a tank for your " +
  total + " fish?",
  "Purchase Suggestion",
  JOptionPane.YES_NO_OPTION,
  JOptionPane.QUESTION_MESSAGE,
  null,options,options[0]);

 System.out.print( "SUGGESTION: Would you like to buy a tank for your "
  + total + " fish? - " );

 if (n == 0) {
  Purchase purchase = new Purchase( order, fishTank );
  workingMemory.insert( purchase );
  order.addItem( purchase );
  System.out.println( "Yes" );
 } else {
  System.out.println( "No" );
 }
 return true;
}
```

Having these functions in the rules file makes the PetStore sample more compact.. In real life you would probably have the functions in a file of their own (within the same rules package), or as a static method on a standard Java class (and import them using the *import function my.package.Foo.hello* syntax).

The above functions are:

- *doCheckout()* - Displays a dialog asking the user if they wish to checkout. If they do, focus is set to the *checkOut* agenda-group, allowing rules in that group to (potentially) fire.

- *requireTank()* - Displays a dialog asking the user if they wish to buy a tank. If so, a new FishTank *Product* is added to the orderlist in working memory.

We'll see later the rules that call these functions. The next set of examples are from the PetStore rules themselves. The first extract is the one that happens to fire first (partly because it has the *auto-focus* attibute set to true).

```
// insert each item in the shopping cart into the Working Memory
rule "Explode Cart"
 agenda-group "init"
 auto-focus true
 salience 10
 dialect "java"
when
 $order : Order( grossTotal == -1 )
 $item : Purchase() from $order.items
then
 insert( $item );
 drools.getKnowledgeRuntime().getAgenda().getAgendaGroup
  ( "show items" ).setFocus();
 drools.getKnowledgeRuntime().getAgenda().getAgendaGroup
  ( "evaluate" ).setFocus();
end
```

Example 8.49. Putting Each Individual Item into Working Memory - Extract (3) from the PetStore.drl

This rule matches against all orders that do not yet have an Order.grossTotal calculated . It loops for each purchase item in that order. Some of the *Explode Cart* rule should be familiar; the rule name, the salience (suggesting of the order that the rules should be fired in) and the dialect set to *java*. There are three new items in the rule:

- *agenda-group "init"* - the name of the agenda group. In this case, there is only one rule in the group. However, nothing in Java code, or a rule, sets the focus to this group. It relies on the next attibute for it's chance to fire.

- *auto-focus true* - This is the only rule in the sample, so when *fireAllRules()* is called from within the Java code, this rule is the first to get a chance to fire.

- *drools.setFocus()* - This sets the focus to the *show items* and *evaluate* agenda groups in turn, giving their rules a chance to fire. In practice, we loop through all items on the order, inserting them into memory, then firing the other rules after each insert.

The next two listings shows the rules within the *show items* and *evaluate* agenda groups. We look at them in the order that they are called.

```
rule "Show Items"
 agenda-group "show items"
 dialect "mvel"
when
 $order : Order( )
 $p : Purchase( order == $order )
then
 textArea.append( $p.product + "\n");
end
```

Example 8.50. Show Items in the GUI Extract (4) from the PetStore.drl

The *show items* agenda-group has only one rule, also called *Show Items* (note the difference in case). For each purchase on the order currently in the working memory (session) it logs details to the text area (at the bottom of the GUI). The *textArea* variable used to do this is one of the Global Variables we looked at earlier.

The *evaluate* Agenda group also gains focus from the *explode cart* rule above. This Agenda group has two rules (shown in the following example): *Free Fish Food Sample* and *Suggest Tank*.

```
// Free Fish Food sample when we buy a Gold Fish if we haven't already
//bought Fish Food and dont already have a Fish Food Sample
rule "Free Fish Food Sample"
 agenda-group "evaluate"
 dialect "mvel"
when
 $order : Order()
 not ( $p : Product( name == "Fish Food") &&
 Purchase( product == $p ) )
  not ( $p : Product( name == "Fish Food Sample") &&
  Purchase( product == $p ) )
  exists ( $p : Product( name == "Gold Fish") &&
  Purchase( product == $p ) )
  $fishFoodSample : Product( name == "Fish Food Sample" );
then
 System.out.println( "Adding free Fish Food Sample to cart" );
 purchase = new Purchase($order, $fishFoodSample);
 insert( purchase );
 $order.addItem( purchase );
end

// Suggest a tank if we have bought more than 5 gold fish and dont
// already have one
rule "Suggest Tank"
 agenda-group "evaluate"
 dialect "java"
when
 $order : Order()
 not ( $p : Product( name == "Fish Tank") &&
 Purchase( product == $p ) )
 ArrayList( $total : size &gt; 5 ) from collect( Purchase
 ( product.name == "Gold Fish" ) )
 $fishTank : Product( name == "Fish Tank" )
then
 requireTank(frame, drools.getWorkingMemory(),
 $order, $fishTank, $total);
end
```

The *Free Fish Food Sample* rule will only fire if:

- We *do not*already have any fish food.
- We *do not* already have a free fish food sample.
- We *do* have a Gold Fish in our order.

If the rule does fire, it creates a new product (Fish Food Sample), and adds it to the Order in working memory.

The *Suggest Tank* rule will only fire if

- We *do not* already have a Fish Tank in our order.
- If we *can* find more than 5 Gold Fish Products in our order.

If the rule does fire, it calls the *requireTank()* function that we looked at earlier (showing a Dialog to the user, and adding a Tank to the order/working memory if confirmed). When calling the *requireTank()* function the rule passes the global *frame* variable so that the function has a handle to the Swing GUI.

The next rule we look at is *do checkout.*

```

rule "do checkout"
 dialect "java"
when
then
 doCheckout(frame, drools.getWorkingMemory());
end

```

Example 8.51. Doing the Checkout - Extract (6) from the PetStore.drl

The *do checkout* rule has *no* agenda-group set and *no* auto-focus attribute. As such, is is deemed part of the default (MAIN) agenda-group - the same as the other non PetStore examples where agenda groups are not used. This group gets focus by default when all the rules/agenda-groups that explicitly had focus set to them have run their course.

There is no LHS to the rule, so the RHS will always call the *doCheckout()* function. When calling the *doCheckout()* function the rule passes the global *frame* variable so the function has a handle to the Swing GUI. As we saw earlier, the *doCheckout()* function shows a confirmation dialog to the user. If confirmed, the function sets the focus to the *checkout* agenda-group, allowing the next lot of rules to fire.

```
rule "Gross Total"
 agenda-group "checkout"
 dialect "mvel"
when
 $order : Order( grossTotal == -1)
 Number( total : doubleValue ) from accumulate( Purchase ( $price :
 product.price ),sum( $price ) )
then
 modify( $order ) { grossTotal = total };
 textArea.append( "\ngross total=" + total + "\n" );
end

rule "Apply 5% Discount"
 agenda-group "checkout"
 dialect "mvel"
when
 $order : Order( grossTotal &gt;= 10 &amp;&amp; &lt; 20 )
then
 $order.discountedTotal = $order.grossTotal * 0.95;
 textArea.append("discountedTotal total="+$order.discountedTotal+"\n");
end

rule "Apply 10% Discount"
 agenda-group "checkout"
 dialect "mvel"
when
 $order : Order( grossTotal &gt;= 20 )
then
 $order.discountedTotal = $order.grossTotal * 0.90;
 textArea.append("discountedTotal total="+$order.discountedTotal+"\n");
end
```

There are three rules in the *checkout* agenda-group:

- *Gross Total* - if we haven't already calculated the gross total, accumulates the product prices into a total, puts this total into working memory, and displays it via the Swing TextArea (using the *textArea* global variable yet again).
- *Apply 5% Discount* - if the gross total is between 10 and 20, then calculate the discounted total and add it to working memory/display in the text area.
- *Apply 10% Discount* - if the gross total is equal to or greater than 20, calculate the discounted total and add it to working memory/display in the text area.

Now we've run through what happens in the code, lets have a look at what happens when we run the code for real. The *PetStore.java* example contains a *main()* method, so it can be run as a standard Java application (either from the command line or via the IDE). This assumes you have your classpath set correctly (see the start of the examples section for more information).

The first screen that we see is the Pet Store Demo. It has a List of available products (top left) , an empty list of selected products (top right), checkout and reset buttons (middle) and an empty system messages area (bottom).

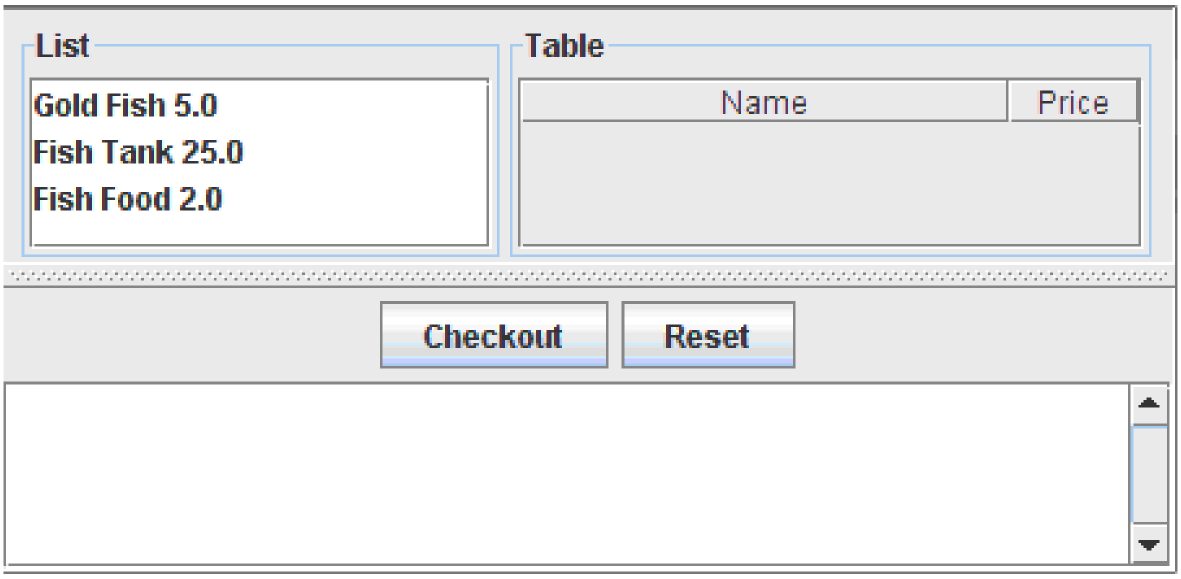
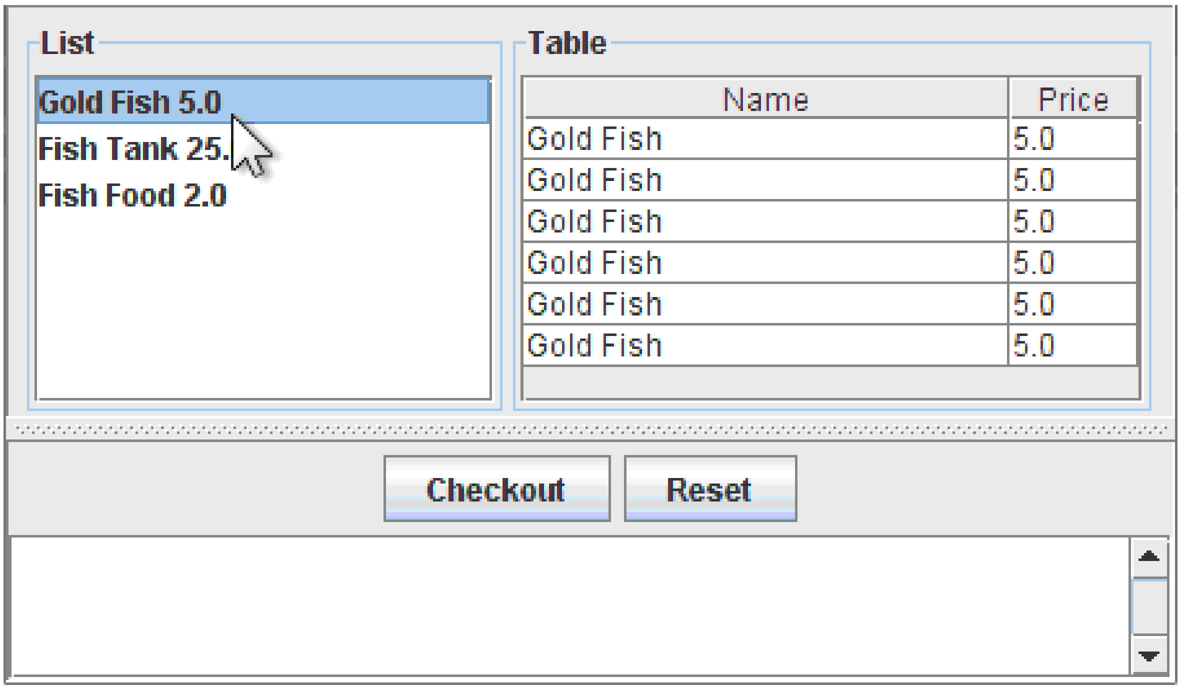


Figure 8.13. PetStore Demo just after Launch

To get to this point, the following things have happened:

1. The *main()* method has run and loaded the RuleBase *but not yet fired the rules*. This is the only rules related code to run so far.
2. A new *PetStoreUI* class is created and given a handle to the RuleBase (for later use).
3. Various Swing Components do their stuff, and the above screen is shown and *waits for user input*.

Clicking on various products from the list might give you a screen similar to the one below.



Name	Price
Gold Fish	5.0
Gold Fish	5.0
Gold Fish	5.0
Gold Fish	5.0
Gold Fish	5.0
Gold Fish	5.0

Figure 8.14. PetStore Demo with Products Selected

Note that *no rules* code has been fired here. This is only swing code, listening for the mouse click event, and added the clicked product to the *TableModel* object for display in the top right hand section (as an aside, this is a classic use of the Model View Controller - MVC - design pattern).

It is only when we press the *Checkout* that we fire our business rules, in roughly the same order that we walked through the code earlier.

1. The *CheckOutCallBack.checkout()* method is called (eventually) by the Swing class waiting for the click on the checkout button. This inserts the data from the *TableModel* object (top right hand side of the GUI), and handles from the GUI into the session/working memory. It then fires the rules.

2. The *Explode Cart* rule is the first to fire, given that has *auto-focus* set to true. It loops through all the products in the cart, makes sure the products are in the working memory, then gives the *Show Items* and *Evaluation* agenda groups a chance to fire. The rules in these groups, add the contents of the cart to the text area (bottom), decide whether or not to give us free fish food, and whether to ask if we want to buy a fish tank.

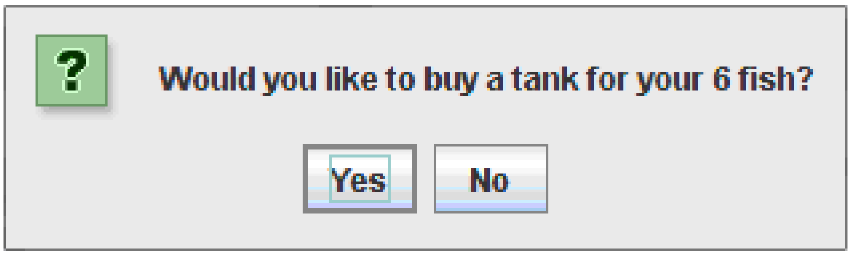
   
   Would you like to buy a tank for your 6 fish?

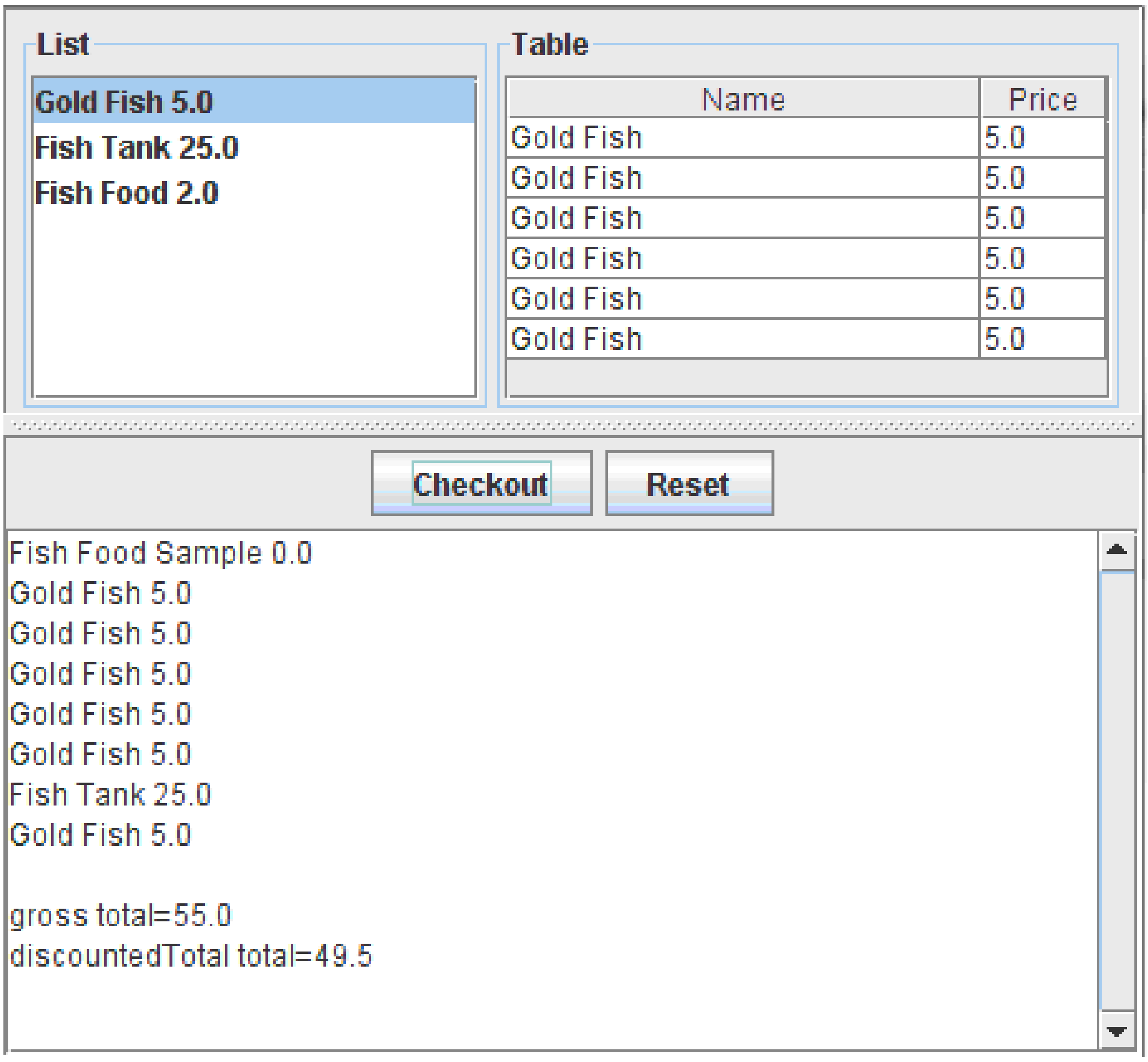
   Figure 8.15. Do we want to buy a fish tank?

3. The *Do Checkout* rule is the next to fire as it (a) No other agenda group currently has focus and (b) it is part of the default (MAIN) agenda group. It always calls the *doCheckout() function* which displays a 'Would you like to Checkout?' Dialog Box.

4. The *doCheckout() function* sets the focus to the *checkout* agenda-group, giving the rules in that group the option to fire.

5. The rules in the the *checkout* agenda-group, display the contents of the cart and apply the appropriate discount.

6. *Swing then waits for user input* to either checkout more products (and to cause the rules to fire again) or to close the GUI - Figure 4 below.



Name	Price
Gold Fish	5.0
Gold Fish	5.0
Gold Fish	5.0
Gold Fish	5.0
Gold Fish	5.0
Gold Fish	5.0

```
Fish Food Sample 0.0
Gold Fish 5.0
Gold Fish 5.0
Gold Fish 5.0
Gold Fish 5.0
Gold Fish 5.0
Fish Tank 25.0
Gold Fish 5.0

gross total=55.0
discountedTotal total=49.5
```

Figure 8.16. Petstore Demo after all rules have fired.

Should we choose, we could add more System.out calls to demonstrate this flow of events. The current output of the console of the above sample is as per the listing below.

```
Adding free Fish Food Sample to cart
SUGGESTION: Would you like to buy a tank for your 6 fish? - Yes
```

Example 8.52. Console (System.out) from running the PetStore GUI

8.7. Honest Politician Example

The honest politician example demonstrates truth maintenance with logical assertions. The basic premise is that an object can only exist while a statement is true. A rule's consequence can logically insert an object with the *insertLogical* method. This means the object will only remain in the working memory as long as the rule that logically inserted it remains true, when the rule is no longer true the object is automatically retracted.

In this example there is Politician class with a name and a boolean value for honest state, four politicians with honest state set to true are inserted.

```
public class Politician
{
    private String name;
    private boolean honest;
    //...
}
```

Example 8.53. Politician Class

```
Politician blair  = new Politician("blair", true);
Politician bush  = new Politician("bush", true);
Politician chirac  = new Politician("chirac", true);
Politician schroder   = new Politician("schroder", true);

ksession.insert( blair );
ksession.insert( bush );
ksession.insert( chirac );
ksession.insert( schroder );

ksession.fireAllRules();
```

Example 8.54. Honest Politician Execution

The console out shows that while there is at least one honest polician democracy lives. Democracy is dead when all honest politicians are corrupted by an evil corporation, and become dishonest.

```
Hurrah!!! Democracy Lives
I'm an evil corporation and I have corrupted schroder
I'm an evil corporation and I have corrupted chirac
I'm an evil corporation and I have corrupted bush
I'm an evil corporation and I have corrupted blair
We are all Doomed!!! Democracy is Dead
```

Example 8.55. Console Output

As soon as there is one more more honest politcians in the working memory a new Hope object is logically asserted, this object will only exist while there is at least one or more honest politicians, the moment all politicians are dishonest then the Hope object will be automatically retracted. This rule is given a salience of 10 to make sure it fires before any other rules, as at this stage the "Hope is Dead" rule is actually true.

```
rule "We have an honest Politician"
 salience 10
when
 exists( Politician( honest == true ) )
then
 insertLogical( new Hope() );
end
```

Example 8.56. Rule "We have an honest politician"

As soon as a Hope object exists, the "Hope Lives" rule matches, and fires. Because it has a salience of 10, it takes priority over "Corrupt the Honest".

```
rule "Hope Lives"
 salience 10
when
 exists( Hope() )
then
 System.out.println("Hurrah!!! Democracy Lives");
end
```

Example 8.57. Rule "Hope Lives"

Now that hope exists and we have, at the start, four honest politicians, we have 4 activations for this rule all in conflict. This rule iterates over those rules firing each one in turn, corrupting each politician so that they are no longer honest. When all four politicians have been corrupted we have no politicians with the property "honest == true", therefore the rule "We have an honest Politician" is no longer true and the object it logically inserts "new Hope()" is automatically retracted.

```
rule "Corrupt the Honest"
when
 politician : Politician( honest == true )
 exists( Hope() )
then
 System.out.println( "I'm an evil corporation and I have corrupted "
  + politician.getName() );
 modify ( politician ) { honest = false };
end
```

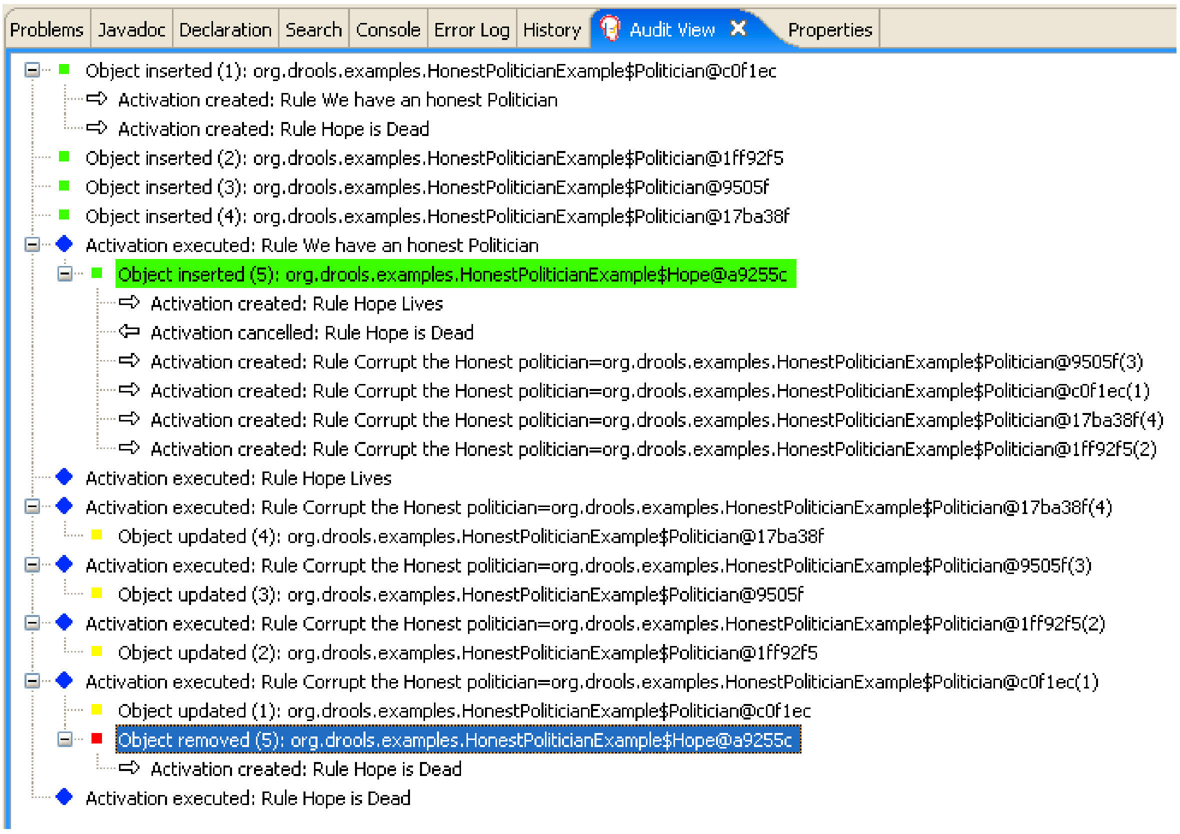
Example 8.58. Rule "Corrupt the Honest"

With Hope being automatically retracted, via the truth maintenance system, then Hope no longer exists in the system and this rule will match and fire.

```
rule "Hope is Dead"
when
 not( Hope() )
then
 System.out.println( "We are all Doomed!!! Democracy is Dead" );
end
```

Example 8.59. Rule "Hope is Dead"

Lets take a look at the audit trail for this application:



```
Object inserted (1): org.drools.examples.HonestPoliticianExample$Politician@c0f1ec
    Activation created: Rule We have an honest Politician
    Activation created: Rule Hope is Dead
Object inserted (2): org.drools.examples.HonestPoliticianExample$Politician@1ff92f5
Object inserted (3): org.drools.examples.HonestPoliticianExample$Politician@9505f
Object inserted (4): org.drools.examples.HonestPoliticianExample$Politician@17ba38f
Activation executed: Rule We have an honest Politician
    Object inserted (5): org.drools.examples.HonestPoliticianExample$Hope@a9255c
        Activation created: Rule Hope Lives
        Activation cancelled: Rule Hope is Dead
        Activation created: Rule Corrupt the Honest politician=org.drools.examples.HonestPoliticianExample$Politician@9505f(3)
        Activation created: Rule Corrupt the Honest politician=org.drools.examples.HonestPoliticianExample$Politician@c0f1ec(1)
        Activation created: Rule Corrupt the Honest politician=org.drools.examples.HonestPoliticianExample$Politician@17ba38f(4)
        Activation created: Rule Corrupt the Honest politician=org.drools.examples.HonestPoliticianExample$Politician@1ff92f5(2)
Activation executed: Rule Hope Lives
Activation executed: Rule Corrupt the Honest politician=org.drools.examples.HonestPoliticianExample$Politician@17ba38f(4)
    Object updated (4): org.drools.examples.HonestPoliticianExample$Politician@17ba38f
Activation executed: Rule Corrupt the Honest politician=org.drools.examples.HonestPoliticianExample$Politician@9505f(3)
    Object updated (3): org.drools.examples.HonestPoliticianExample$Politician@9505f
Activation executed: Rule Corrupt the Honest politician=org.drools.examples.HonestPoliticianExample$Politician@1ff92f5(2)
    Object updated (2): org.drools.examples.HonestPoliticianExample$Politician@1ff92f5
Activation executed: Rule Corrupt the Honest politician=org.drools.examples.HonestPoliticianExample$Politician@c0f1ec(1)
    Object updated (1): org.drools.examples.HonestPoliticianExample$Politician@c0f1ec
    Object removed (5): org.drools.examples.HonestPoliticianExample$Hope@a9255c
        Activation created: Rule Hope is Dead
Activation executed: Rule Hope is Dead
```

Figure 8.17. Audit View

The moment we insert the first politician we have two activations. The "We have an honest Politician" is activated only once for the first inserted politician because it uses an existential 'exists' conditional element which only matches. The rule "Hope is Dead" is also activated at this stage, because we have not inserted the Hope object yet.

"We have an honest Politician" fires first, because it has a higher salience over "Hope is Dead" which inserts the Hope object (highlighted in green in the example). The insertion of the Hope object activates "Hope Lives" and de-activates "Hope is Dead", it also actives "Corrupt the Honest" for each inserted honest politician. "Rule Hope Lives" executes printing "Hurrah!!! Democracy Lives".

Then for each politician the rule "Corrupt the Honest" fires printing "I'm an evil corporation and I have corrupted X", where X is the name of the politician, and modifies the politicians honest value to false. When the last honest politician is corrupted Hope is automatically retracted, by the truth maintenance system, as shown by the blue highlighted area.

The green highlighted area shows the origin of the currently selected blue highlighted area. Once Hope is retracted "Hope is dead" activates and fires printing "We are all Doomed!!! Democracy is Dead".

8.8. Sudoku Example

Name:	Sudoku
Main class:	**`org.drools.examples.sudoku.Main`**
Type:	java application
Rules file:	**`sudokuSolver.drl, sudokuValidator.drl`**
Objective:	Demonstrates the solving of logic problems, and complex pattern matching.

This example demonstrates how Drools can be used to find a solution in a large potential solution space based on a number of constraints. We use the popular puzzle of Sudoku. This example also shows how Drools can be integrated into a graphical interface and how callbacks can be used to interact with a running Drools rules engine in order to update the graphical interface based on changes in the working memory at runtime.

8.8.1. Sudoku Overview

Sudoku is a logic-based number placement puzzle. The objective is to fill a 9x9 grid so that each column, each row, and each of the nine 3x3 zones contains the digits from 1 to 9 once and only once.

The puzzle setter provides a partially completed grid and the puzzle solver's task is to complete the grid with these constraints.

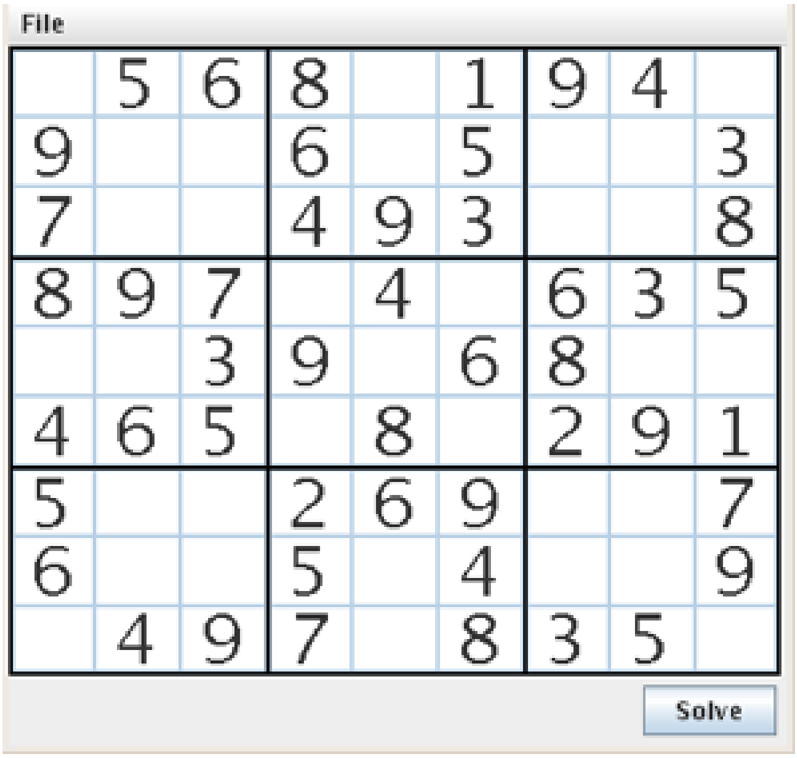
The general strategy to solve the problem is to ensure that when you insert a new number it should be unique in that particular region(blocks) and also in that particular row and column.

You can refer to *http://en.wikipedia.org/wiki/Sudoku* for a more detailed description.

8.8.2. Running the Example

Download and install drools-examples as described above and then execute: **`java org.drools.examples.sudoku.Main`**

A window is displayed with a relatively simple partially filled grid.



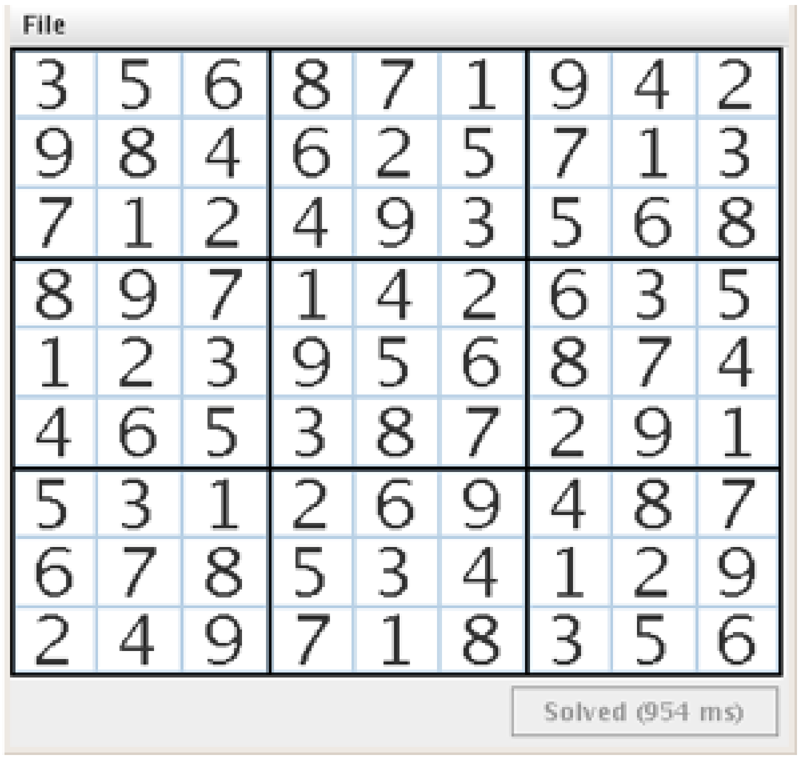
	5	6	8		1	9	4	
9			6		5			3
7			4	9	3			8
8	9	7		4		6	3	5
		3	9		6	8		
4	6	5		8		2	9	1
5			2	6	9			7
6			5		4			9
	4	9	7		8	3	5	

Figure 8.18. Partially Filled Grid

Click on the Solve button and the rules engine will fill out the remaining values. The console will display detailed information of the rules which are executing to solve the puzzle in a human readable form.

```
Rule #3 determined the value at (4,1) could not be 4 as this value already
 exists in the same column at (8,1)
Rule #3 determined the value at (5,5) could not be 2 as this value already
 exists in the same row at (5,6)
Rule #7 determined (3,5) is 2 as this is the only possible cell in the
 column that can have this value
Rule #1 cleared the other PossibleCellValues for (3,5) as a
 ResolvedCellValue of 2 exists for this cell.
Rule #1 cleared the other PossibleCellValues for (3,5) as a
 ResolvedCellValue of 2 exists for this cell. ...
Rule #3 determined the value at (1,1) could not be 1 as this value already
 exists in the same zone at (2,1)
Rule #6 determined (1,7) is 1 as this is the only possible cell in the row
 that can have this value
Rule #1 cleared the other PossibleCellValues for (1,7) as a
 ResolvedCellValue of 1 exists for this cell.
Rule #6 determined (1,1) is 8 as this is the only possible cell in the row
 that can have this value
```

Once all of the activated rules for the solving logic have executed, the engine executes a second rule base to check that the solution is complete and valid. In this case it is, and the "Solve" button is disabled and displays the text "Solved (1052ms)".

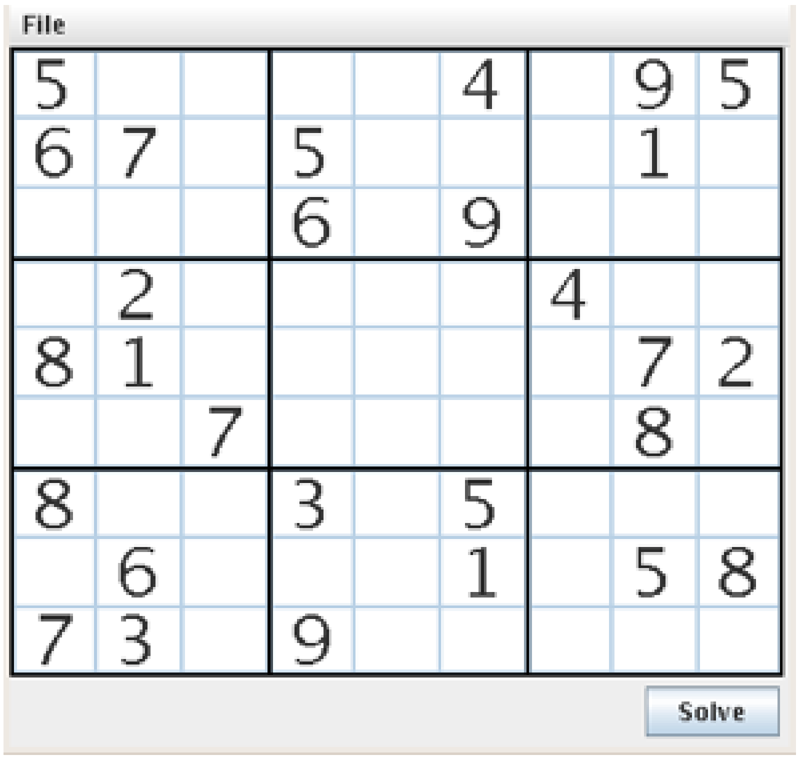


3	5	6	8	7	1	9	4	2
9	8	4	6	2	5	7	1	3
7	1	2	4	9	3	5	6	8
8	9	7	1	4	2	6	3	5
1	2	3	9	5	6	8	7	4
4	6	5	3	8	7	2	9	1
5	3	1	2	6	9	4	8	7
6	7	8	5	3	4	1	2	9
2	4	9	7	1	8	3	5	6

Figure 8.19. Solved Grid

The example comes with a number of grids which can be loaded and solved. Click on File > Samples > Medium to load a more challenging grid. Note that the solve button is enabled when the new grid is loaded. Click on the "Solve" button again to solve this new grid.

Now, let us load a Sudoku grid that is deliberately invalid. Click on File > Samples > !DELIBERATELY BROKEN!. Note that this grid starts with some issues, for example the value 5 appears twice in the first row.



5					4		9	5
6	7		5				1	
			6		9			
	2					4		
8	1						7	2
		7					8	
8			3		5			
	6				1		5	8
7	3		9					

Figure 8.20. Broken Grid

Nevertheless, click on the "Solve" button to apply the solving rules to this invalid Grid. Note that the "Solve" button is relabelled to indicate that the resulting solution is invalid.

In addition, the validation rule set outputs all of the issues which are discovered to the console.

```
There are two cells on the same column with the same value at (6,0) and
 (4,0)
There are two cells on the same column with the same value at (4,0) and
 (6,0)
There are two cells on the same row with the same value at (2,4) and (2,2)
There are two cells on the same row with the same value at (2,2) and (2,4)
There are two cells on the same row with the same value at (6,3) and (6,8)
There are two cells on the same row with the same value at (6,8) and (6,3)
There are two cells on the same column with the same value at (7,4) and
 (0,4)
There are two cells on the same column with the same value at (0,4) and
 (7,4)
There are two cells on the same row with the same value at (0,8) and (0,0)
There are two cells on the same row with the same value at (0,0) and (0,8)
There are two cells on the same column with the same value at (1,2) and
 (3,2)
There are two cells on the same column with the same value at (3,2) and
 (1,2)
There are two cells in the same zone with the same value at (6,3) and (7,3)
There are two cells in the same zone with the same value at (7,3) and (6,3)
There are two cells on the same column with the same value at (7,3) and
 (6,3)
```

```
There are two cells on the same column with the same value at (6,3) and
 (7,3)
```

We will look at the solving rule set later in this section, but for the moment we should note that some theoretically solvable solutions can not be solved by the engine as it stands. Click on File > Samples > Hard 3 to load a sparsely populated Grid.

Now click on the "Solve" button and note that the current rules are unable to complete the grid, even though you may be able to see a way forward with the solution.

At the present time, the solving functionality has been achieved by the use of ten rules. This rule set could be extended to enable the engine to tackle more complex logic for filling grids such as this.

8.8.3. Java Source and Rules Overview

The Java source code can be found in the /src/main/java/org/drools/examples/sudoku directory, with the two DRL files defining the rules located in the /src/main/rules/org/drools/examples/sudoku directory.

org.drools.examples.sudoku.swing contains a set of classes which implement a framework for Sudoku puzzles. This package does not have any dependencies on the Drools libraries.

SudokuGridModel defines an interface which can be implemented to store a Sudoku puzzle as a 9x9 grid of Integer values, some of which may be null, indicating that the value for the cell has not yet been resolved.

SudokuGridView is a Swing component which can visualise any implementation of SudokuGridModel.

SudokuGridEvent and SudokuGridListener are used to communicate state changes between the model and the view, events are fired when a cell's value is resolved or changed. If you are familiar with the model-view-controller patterns in other Swing components such as JTable then this pattern should be familiar. SudokuGridSamples provides a number of partially filled Sudoku puzzles for demo purposes.

org.drools.examples.sudoku.rules contains an implementation of SudokuGridModel which is based on Drools. Two POJOs are used, both of which extend AbstractCellValue and represent a value for a specific cell in the grid, including the row and column location of the cell, an index of the 3x3 zone the cell is contained in and the value of the cell.

PossibleCellValue indicates that we do not currently know for sure what the value in a cell is. There can be 2-9 PossibleCellValues for a given cell. ResolvedCellValue indicates that we have determined what the value for a cell must be. There can only be 1 ResolvedCellValue for a given cell.

DroolsSudokuGridModel implements SudokuGridModel and is responsible for converting an initial two dimensional array of partially specified cells into a set of CellValue POJOs, creating a working memory based on solverSudoku.drl and inserting the CellValue POJOs into the working memory. When the solve() method is called it calls fireAllRules() on this working memory to try to solve the puzzle. DroolsSudokuGridModel attaches a WorkingMemoryListener to the working memory, which allows it to be called back on insert() and retract() events as the puzzle is solved.

When a new ResolvedCellValue is inserted into the working memory, this call back allows the implementation to fire a SudokuGridEvent to its SudokuGridListeners which can then update themselves in realtime. Once all the rules fired by the solver working memory have executed, DroolsSudokuGridModel runs a second set of rules, based on validatorSudoku.drl which works with the same set of POJOs to determine if the resulting grid is a valid and full solution.

org.drools.examples.sudoku.Main implements a Java application which hooks the components desribed above together.

org.drools.examples.sudoku contains two DRL files. solverSudoku.drl defines the rules which attempt to solve a Sudoku puzzle and validator.drl defines the rules which determin whether the current state of the working memory represents a valid solution. Both use PossibleCellValue and ResolvedCellValue POJOs as their facts and both output information to the console as their rules fire. In a real-world situation we would insert() logging information and use the WorkingMemoryListener to display this information to a user rather than use the console in this fashion.

8.8.4. Sudoku Validator Rules (validatorSudoku.drl)

We start with the validator rules as this rule set is shorter and simpler than the solver rule set.

The first rule simply checks that no PossibleCellValue objects remain in the working memory. Once the puzzle is solved, only ResolvedCellValue objects should be present, one for each cell.

The other three rules each match all of the ResolvedCellValue objects and store them in thenew_remote_sitetes instance variable $resolved. They then look respectively for ResolvedCellValues that contain the same value and are located, respectively, in the same row, column or 3x3 zone. If these rules are fired they add a message to a global List of Strings describing the reason the solution is invalid. DroolsSudokoGridModel injects this List before it runs the rule set and checks whether it is empty or not having called fireAllRules(). If it is not empty then it prints all the Strings in the list and sets a flag to indicate that the Grid is not solved.

8.8.5. Sudoku Solving Rules (solverSudoku.drl)

Now let us look at the more complex rule set used to solve Sudoku puzzles.

Rule #1 is basically a "book-keeping" rule. Several of the other rules insert() ResolvedCellValues into the working memory at specific rows and columns once they have determined that a given cell must have a certain value. At this point, it is important to clear the working memory of any inserted PossibleCellValues at the same row and column with invalid values. This rule is therefore given a higher salience than the remaining rules to ensure that as soon as the LHS is true, activations for the rule move to the top of the agenda and are fired. In turn this prevents the spurious firing of other rules due to the combination of a ResolvedCellValue and one or more PossibleCellValues being present in the same cell. This rule also calls update() on the ResolvedCellValue, even though its value has not in fact been modified to ensure that Drools fires an event to any WorkingMemoryListeners attached to the working memory so that they can update themselves - in this case so that the GUI can display the new state of the grid.

Rule #2 identifies cells in the grid which have only one possible value. The first line of the when caluse matches all of the PossibleCellValue objects in the working memory. The second line demonstrates a use of the not keyword. This rule will only fire if no other PossibleCellValue objects exist in the working memory at the same row and column but with a different value. When the rule fires, the single PossibleCellValue at the row and column is retracted from the working memory and is replaced by a new ResolvedCellValue at the same row and column with the same value.

Rule #3 removes PossibleCellValues with a given value from a row when they have the same value as a ResolvedCellValue. In other words, when a cell is filled out with a resolved value, we need to remove the possibility of any other cell on the same row having this value. The first line of the when clause matches all ResolvedCellValue objects in the working memory. The second line matches PossibleCellValues which have both the same row and the same value as these ResolvedCellValue objects. If any are found, the rule activates and, when fired retracts the PossibleCellValue which can no longer be a solution for that cell.

Rules #4 and #5 act in the same way as Rule #3 but check for redundant PossibleCellValues in a given column and a given zone of the grid as a ResolvedCellValue respectively.

Rule #6 checks for the scenario where a possible cell value only appears once in a given row. The first line of the LHS matches against all PossibleCellValues in the working memory, storing the result in a number of local variables. The second line checks that no other PossibleCellValues with the same value exist on this row. The third to fifth lines check that there is not a ResolvedCellValue with the same value in the same zone, row or column so that this rule does not fire prematurely. Interestingly we could remove lines 3-5 and give rules #3,#4 and #5 a higher salience to make sure they always fired before rules #6,#7 and #8. When the rule fires, we know that $possible must represent the value for the cell so, as in Rule #2 we retract $possible and replace it with the equivalent, new ResolvedCellValue.

Rules #7 and #8 act in the same way as Rule #2 but check for single PossibleCellValues in a given column and a given zone of the grid respectively.

Rule #9 represents the most complex currently implemented rule. This rule implements the logic that, if we know that a pair of given values can only occur in two cells on a specific row, (for example we have determined the values of 4 and 6 can only appear in the first row in cells 0,3 and 0,5) and this pair of cells can not hold other values then, although we do not know which of the pair contains a four and which contains a six we know that the 4 and the 6 must be in these two cells and hence can remove the possibility of them occuring anywhere else in the same row.

Rules #10 and #11 act in the same way as Rule #9 but check for the existence of only two possible values in a given column and zone respectively.

To solve harder grids, the rule set would need to be extended further with more complex rules that encapsulated more complex reasoning.

8.8.6. Suggestions for Future Developments

There are a number of ways in which this example could be developed. The reader is encouraged to consider these as excercises.

- Agenda-group: agenda groups are a great declarative tool for phased execution. In this example, it is easy to see we have 2 phases: "resolution" and "validation". Right now, they are executed by creating two separate rule bases, each for one "job". I think it would be better for us to define agenda-groups for all the rules, spliting them in "resolution" rules and "validation" rules, all loaded in a single rule base. The engine executes resolution and right after that, executes validation.

- Auto-focus: auto focus is a great way of handling exceptions to the regular rules execution. In our case, if we detect an inconsistency, either in the input data or in the resolution rules, why should we spend time continuing the execution if it will be invalid anyway? I think it is better to simply (and immediatly) report the inconsistency as soon as it is found. To do that, since we now have a single rulebase with all rules, we simply need to define auto-focus attribute for all rules validating puzzle consistency.

- Logical insert: an inconsistency only exists while wrong data is in the working memory. As so, we could state that the the validation rules logically insert inconsistencies and as soon as the offending data is retracted, the inconsistency no longer exists.

- session.iterateObjects(): although a valid use case having a global list to add the found problems, I think it would be more interesting to ask the stateful session by the desired list of problems, using session.iterateObjects(new ClassObjectFilter(Inconsistency.class)); Having the inconsistency class can also allow us to paint in RED the offending cells in the GUI.
- drools.halt(): even reporting the error as soon as it is found, we need a way to tell the engine to stop evaluating rules. We can do that creating a rule that in the presence of Inconsistencies, calls drools.halt() to stop evaluation.
- queries: looking at the method getPossibleCellValues(int row, int col) in DroolsSudokuGridModel, we see it iterating over all CellValues and looking for the few it wants. That, IMO, is a great opportunity to teach drools queries. We just define a query to return the objects we want and iterate over it. Clean and nice. Other queries may be defined as needed.
- session.iterateObjects(): although a valid use case having a global list to add the found problems, I think it would be more interesting to ask the stateful session by the desired list of problems, using session.iterateObjects(new ClassObjectFilter(Inconsistency.class)); Having the inconsistency class can also allow us to paint in RED the offending cells in the GUI.
- Globals as services: the main objective of this change is to attend the next change I will propose, but it is nice by its own I guess. :) In order to teach the use of "globals" as services, it would be nice to setup a call back, so that each rule that finds the ResolvedCellValue for a given cell can call, to notify and update the corresponding cell in the GUI, providing immediate feedback for the user. Also, the last found cell could have its number painted in a different color to facilitate the identification of the rules conclusions.
- Step by step execution: now that we have immediate user feedback, we can make use of the restricted run feature in drools. I.e., we could add a button in the GUI, so that the user clicks and causes the execution of a single rule, by calling fireAllRules(1). This way, the user can see, step by step, what the engine is doing.

8.9. Miss Manners and Benchmarking

Name:	Miss Manners
Main class:	**`org.drools.benchmark.manners.MannersBenchmark`**
Type:	java application
Rules file:	**`manners.drl`**
Objective:	Advanced walkthrough on the Manners benchmark, covers Depth conflict resolution in depth.

8.9.1. Introduction

Miss Manners is throwing a party and, being the good host, she wants to arrange good seating. Her initial design arranges everyone in male female pairs, but then she worries about whether or not people will have things to talk about. What is a good host to do? She decides to note the hobby of each guest so she can then arrange her guests in not only male and female pairs but also ensure that a guest has someone to talk about a common hobby, from either their left or right side.

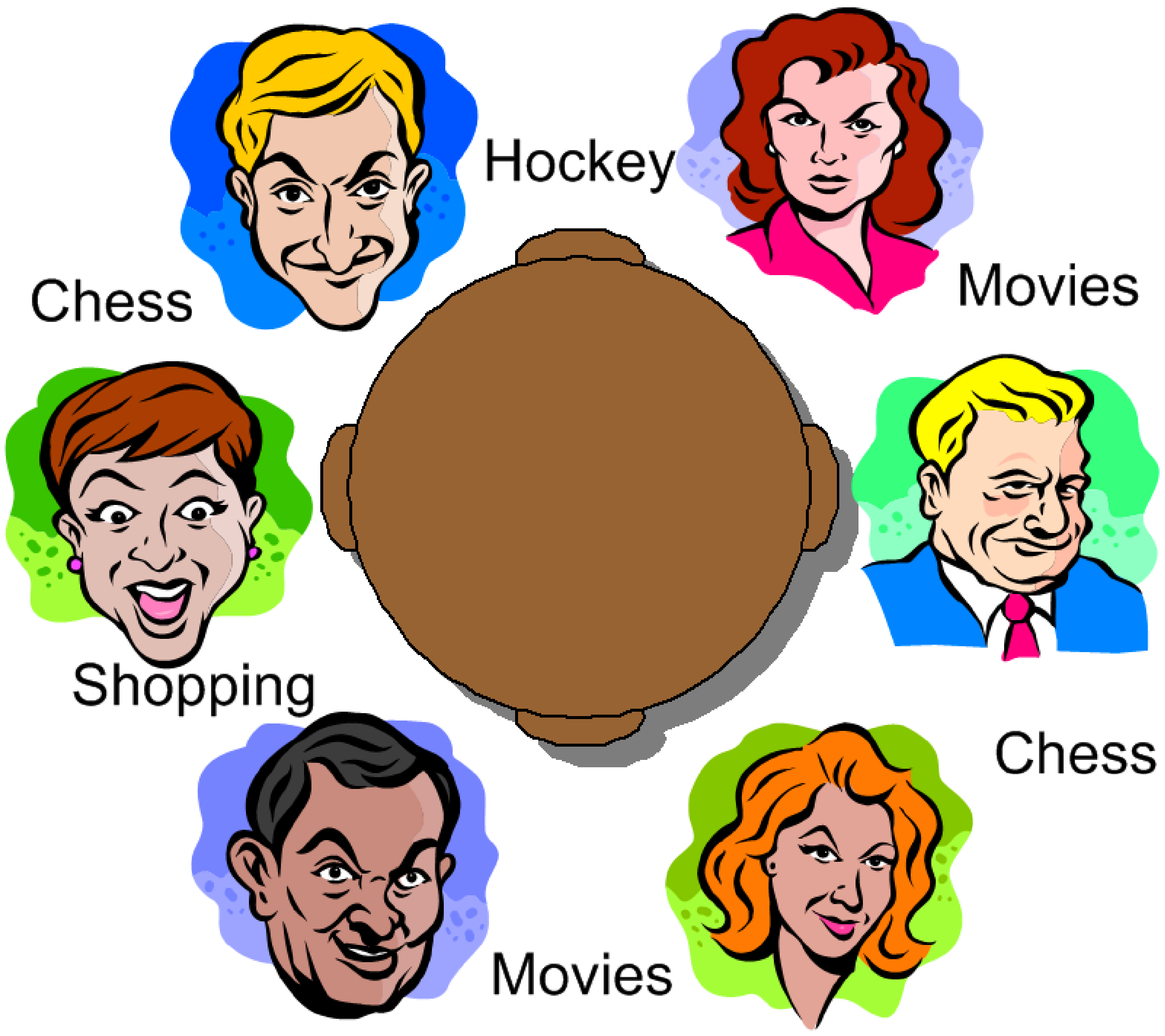


Figure 8.21. Miss Manners' Guests

8.9.1.1. BenchMarking

Five benchmarks were established in the 1991 paper "Effects of Database Size on Rule System Performance: Five Case Studies" by Brant, Timothy Grose, Bernie Lofaso, & Daniel P. Miranker.

- Manners

  Uses a depth-first search approach to determine the seating arrangements of boy/girl, and one common hobby for dinner guests.

- Waltz

  Line labeling for simple scenes by constraint propagation.

- WaltzDB

  More general version of Waltz to be able to adapt to a database of facts.

- ARP

  Route planner for a robotic air vehicle using the A* search algorithm.

- Weavera

  VLSI router for channels and boxes using a black-board technique.

Manners has become the de facto rule engine benchmark; however it's behavior is now well known and many engines optimize for this thus negating its usefulness as a benchmark which is why Waltz is becoming more favorable. These 5 benchmarks are also published at the University of Texas *http://www.cs.utexas.edu/ftp/pub/ops5-benchmark-suite/*.

8.9.1.2. Miss Manners Execution Flow

After the first Seating arrangement has been assigned, a depth-first recursion occurs, which repeatedly assigns correct Seating arrangements until the last seat is assigned. Manners uses a Context instance to control execution flow; the activity diagram is partitioned to show the relation of the rule execution to the current Context state.

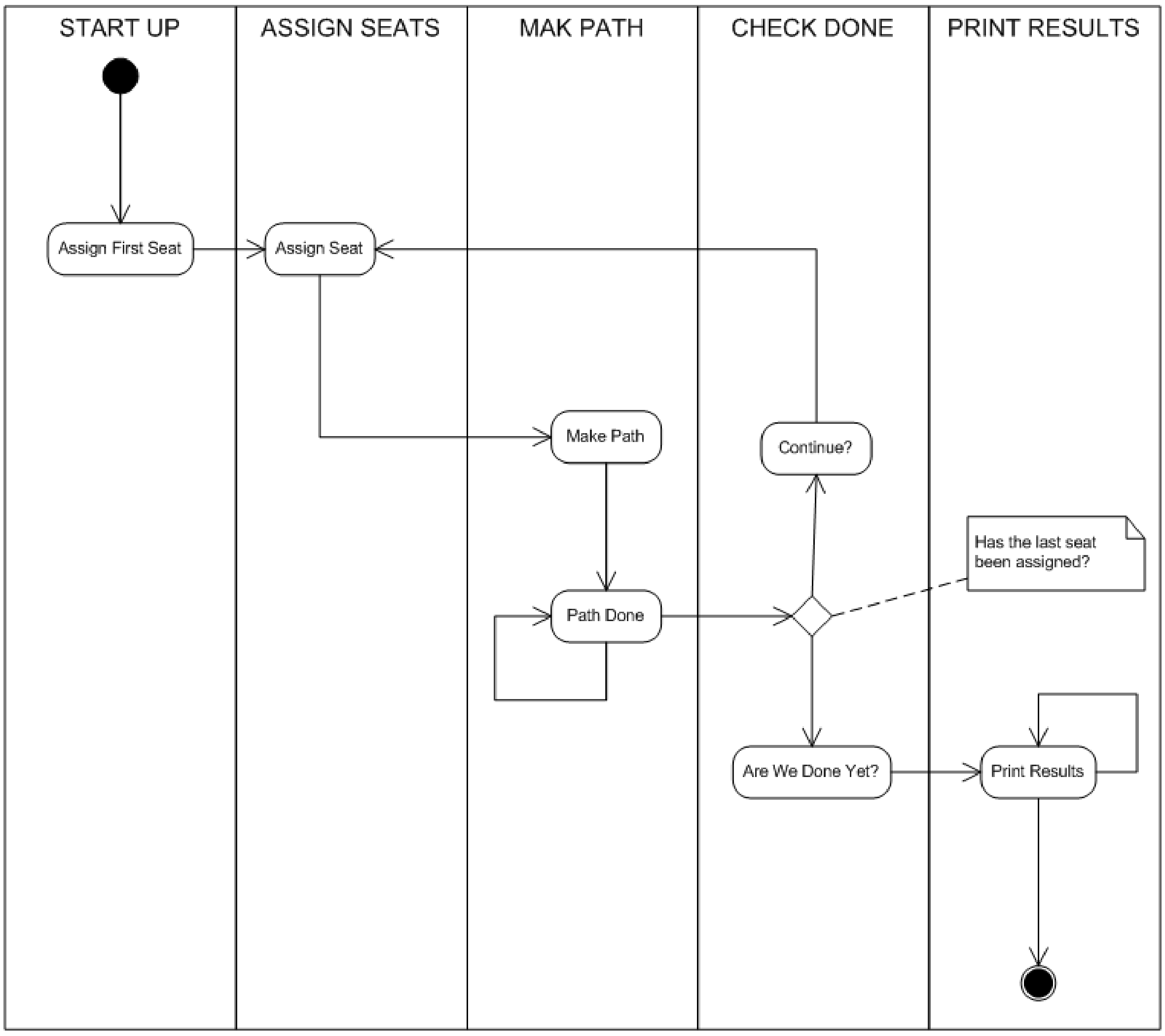


Figure 8.22. Manners Activity Diagram

8.9.1.3. The Data and Results

Before going deeper into the rules, lets first take a look at the asserted data and the resulting Seating arrangement. The data is a simple set of 5 guests who should be arranged in male/female pairs with common hobbies.

Each line of the results list is printed per execution of the "Assign Seat" rule. They key is that each line has a pid one greater than the last, the significance of this will be explained in the "Assign Seating" rule description. The 'l' and the 'r' refer to the left and right, 's' is sean and 'n' is the guest name. In my actual implementation I used longer notation, 'leftGuestName', but this is not practice in a printed article. I found the notation of left and right preferable to the original OPS5 '1' and '2

```
(guest (name n1) (sex m) (hobby  h1)  )
(guest (name n2) (sex f) (hobby  h1)  )
(guest (name n2) (sex f) (hobby  h3)  )
(guest (name n3) (sex m) (hobby  h3)  )
(guest (name n4) (sex m) (hobby  h1)  )
(guest (name n4) (sex f) (hobby  h2)  )
(guest (name n4) (sex f) (hobby  h3)  )
(guest (name n5) (sex f) (hobby  h2)  )
(guest (name n5) (sex f) (hobby  h1)  )
(last_seat (seat 5)  )
```

The Results

```
[Seating id=1, pid=0, done=true, ls=1, ln=n5, rs=1, rn=n5]
[Seating id=2, pid=1, done=false, ls=1, ln=n5, rs=2, rn=n4]
[Seating id=3, pid=2, done=false, ls=2, ln=n4, rs=3, rn=n3]
[Seating id=4, pid=3, done=false, ls=3, rn=n3, rs=4, rn=n2]
[Seating id=5, pid=4, done=false, ls=4, ln=n2, rs=5, rn=n1]
```

8.9.2. In-depth Look

8.9.2.1. Cheating

Manners has been around a long time and is a contrived benchmark meant to exercise the cross product joins and agenda, many people not understanding this tweak the example to achieve better performance, making their use of the Manners benchmark pointless. Known cheats to Miss Manners are:

- Using arrays for a guests hobbies, instead of asserting each one as a single fact. This massively reduces the cross products.
- The altering of the sequence of data can also reducing the amount of matching increase execution speed.
- Changing NOT CE (conditional element) such that the test algorithm only uses the "first-best-match". Basically, changing the test algorithm to backward chaining. the results are only comparable to other backward chaining rule engines or ports of Manners.

- Removing the context so the rule engine matches the guests and seats prematurely. A proper port will prevent facts from matching using the context start.
- Any change which prevents the rule engine from performing combinatorial pattern matching.
- If no facts are retracted in the reasoning cycle, as a result of NOT CE, the port is incorrect.

8.9.2.2. Conflict Resolution

Manners benchmark was written for OPS5, which has two conflict resolution strategies: LEX, and MEA. LEX is a chain of several strategies including Salience, Recency, Complexity. The Recency part of the strategy drives the depth first (LIFO) firing order. The Clips Reference Manual documents the recency strategy as:

> Every fact and instance is marked internally with a "time tag" to indicate its relative recency with respect to every other fact and instance in the system. The pattern entities associated with each rule activation are sorted in descending order for determining placement. An activation with a more recent pattern entities is placed before activations with less recent pattern entities. To determine the placement order of two activations, compare the sorted time tags of the two activations one by one starting with the largest time tags. The comparison should continue until one activation’s time tag is greater than the other activation’s corresponding time tag. The activation with the greater time tag is placed before the other activation on the agenda. If one activation has more pattern entities than the other activation and the compared time tags are all identical, then the activation with more time tags is placed before the other activation on the agenda.

However Jess and Clips both use the Depth strategy, which is simpler and lighter, which Drools also adopted. The Clips Reference Manual documents the Depth strategy as:

> Newly activated rules are placed above all rules of the same salience. For example, given that fact-a activates rule-1 and rule-2 and fact-b activates rule-3 and rule-4, then if fact-a is asserted before fact-b, rule-3 and rule-4 will be above rule-1 and rule-2 on the agenda. However, the position of rule-1 relative to rule-2 and rule-3 relative to rule-4 will be arbitrary.

The initial Drools implementation for the Depth strategy would not work for manners without the use of salience on the "make_path" rule, the Clips Support Forum had this to say:

> The default conflict resolution strategy for CLIPS, depth, is different than the default conflict resolution strategy used by OPS5. Therefore if you directly translate an OPS5 program to CLIPS, but use the default depth conflict resolution strategy, you're only likely to get the correct behavior by coincidence. The lex and mea conflict resolution strategies are provided in CLIPS to allow you to quickly convert and correctly run an OPS5 program in CLIPS

Investigation into the Clips code reveals there is undocumented functionality in the Depth strategy. There is an accumulated time tag used in this strategy; it's not an extensively fact by fact comparison as in the recency strategy, it simply adds the total of all the time tags for each activation and compares.

8.9.2.3. Assign First Seat

Once the context is changed to START_UP, Activations are created for all asserted Guests. Because all Activations are created as the result of a single Working Memory action, they all have the same Activation time tag. The last asserted Guest would have a higher fact time tag and its Activation would fire, because it has the highest accumulated fact time tag. The execution order in this rule has little importance, but has a big impact in the rule "Assign Seat". The Activation fires and asserts the first Seating arrangement, a Path, and the Context's state to create Activation for "Assign Seat".

```
rule assignFirstSeat
when
 context : Context( state == Context.START_UP )
 guest : Guest()
 count : Count()
then
 String guestName = guest.getName();

 Seating seating =  new Seating( count.getValue(),
         1,
         true,
         1,
         guestName,
         1,
         guestName);
 insert( seating );

 Path path = new Path( count.getValue(), 1, guestName );
 insert( path );

 modify( count ) { setValue ( count.getValue() + 1 )  }

 System.out.println( "assign first seat :  "+seating+" : "+path);

 modify( context ) { setState( Context.ASSIGN_SEATS ) }
end
```

8.9.2.4. Assign Seat

This rule determines each of the Seating arrangements. The Rule creates cross product solutions for ALL asserted Seating arrangements against ALL the asserted guests; except against itself or any already assigned chosen solutions.

```
rule findSeating
when
 context : Context( state == Context.ASSIGN_SEATS )
 $s      : Seating( pathDone == true )
 $g1     : Guest( name == $s.rightGuestName )
 $g2     : Guest( sex != $g1.sex, hobby == $g1.hobby )
 count   : Count()
 not ( Path( id == $s.id, guestName == $g2.name) )
 not ( Chosen( id == $s.id, guestName == $g2.name, hobby == $g1.hobby) )
```

```
then
 int rightSeat = $s.getRightSeat();
 int seatId = $s.getId();
 int countValue = count.getValue();

 Seating seating = new Seating( countValue,
         seatId,
         false,
         rightSeat,
         $s.getRightGuestName(),
         rightSeat + 1,
         $g2.getName()
         );
 insert( seating );

 Path path = new Path( countValue, rightSeat + 1, $g2.getName()  );
 insert( path );

 Chosen chosen = new Chosen( seatId, $g2.getName(), $g1.getHobby() );
 insert( chosen  );

 System.err.println( "find seating : "+seating+" : "+path+" : "+chosen);

 modify( count ) {setValue(  countValue + 1 )}
 modify( context ) {setState( Context.MAKE_PATH )}
end
```

The printed results shown earlier suggest it is essential that only the Seating with the highest pid cross product be chosen. How could this be achieved if we have Activations of the same time tag, for nearly all existing Seating and Guests. On the third iteration of "Assign Seat", these are the produced Activations. Remember this is from a very small data set; larger data sets would potentially contain more possible Activated Seating solutions, with multiple solutions per pid.

```
=>[ActivationCreated(35): rule=findSeating
[fid:19:33]:[Seating id=3, pid=2, done=true, ls=2, ln=n4, rs=3, rn=n3]
[fid:4:4]:[Guest name=n3, sex=m, hobbies=h3]
[fid:3:3]:[Guest name=n2, sex=f, hobbies=h3]

=>[ActivationCreated(35): rule=findSeating
[fid:15:23]:[Seating id=2, pid=1, done=true, ls=1, ln=n5, rs=2, rn=n4]
[fid:5:5]:[Guest name=n4, sex=m, hobbies=h1]
[fid:2:2]:[Guest name=n2, sex=f, hobbies=h1]

=>[ActivationCreated(35): rule=findSeating
[fid:13:13]:[Seating id=1, pid=0, done=true, ls=1, ln=n5, rs=1, rn=n5]
[fid:9:9]:[Guest name=n5, sex=f, hobbies=h1]
[fid:1:1]:[Guest name=n1, sex=m, hobbies=h1]
```

The creation of all these redundant Activations might seem pointless, but it must be remembered that Manners is not about good rule design; it's purposefully designed as a bad ruleset to fully stress test the cross product matching process and the agenda, which this clearly does. Notice that each Activation has the same time tag of 35, as they were all activated by the change in Context to ASSIGN_SEATS. With OPS5 and LEX, it would correctly fire the Activation with the last asserted Seating. With Depth, the accumulated fact time tag ensures the Activation with the last asserted Seating fires.

8.9.2.5. Make Path and Path Done

"Make Path" must always fire before "Path Done". A Path is asserted for each Seating arrangement up to the last asserted Seating. Notice that "Path Done" is a subset of "Make Path", so how do we ensure that "Make Path" fires first?

```
rule makePath
when
 Context( state == Context.MAKE_PATH )
 Seating( seatingId:id, seatingPid:pid, pathDone == false )
 Path( id == seatingPid, pathGuestName:guestName, pathSeat:seat )
 not Path( id == seatingId, guestName == pathGuestName )
then
 insert( new Path( seatingId, pathSeat, pathGuestName ) );
end
```

```
rule pathDone
when
 context : Context( state == Context.MAKE_PATH )
 seating : Seating( pathDone == false )
then
 modify( seating ) {setPathDone( true )}
 modify( context ) {setState( Context.CHECK_DONE)}
end
```

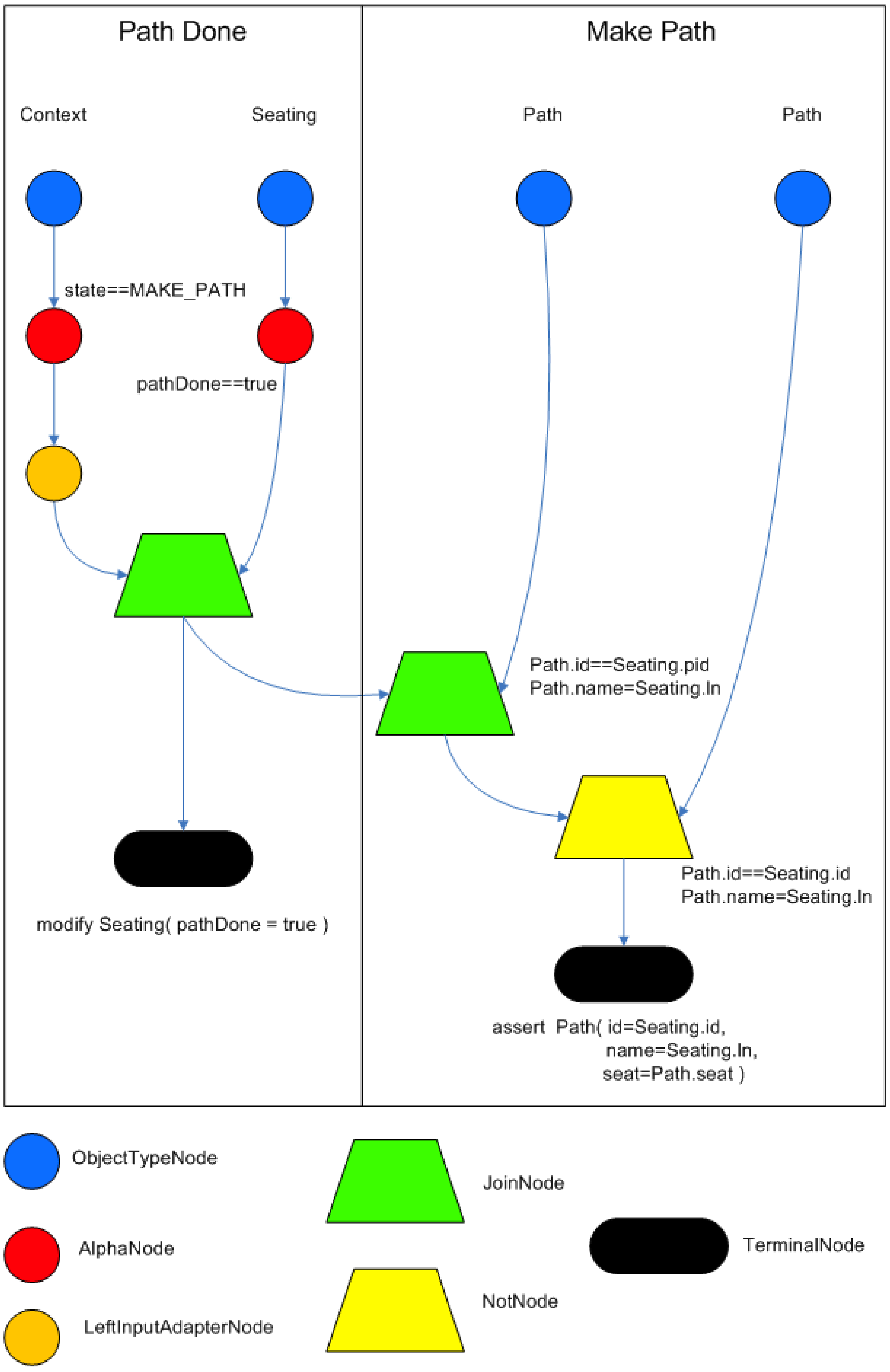


Figure 8.23. Rete Diagram

Both rules end up on the Agenda in conflict and with identical activation time tags, however the accumulate fact time tag is greater for "Make Path" so it gets priority.

8.9.2.6. Continue and Are We Done

"Are We Done" only activates when the last seat is assigned, at which point both rules will be activated. For the same reason that "Make Path" always wins over "Path Done", "Are We Done" will take priority over "Continue".

```
rule areWeDone
when
 context : Context( state == Context.CHECK_DONE )
 LastSeat( lastSeat: seat )
 Seating( rightSeat == lastSeat )
then
 modify( context ) {setState(Context.PRINT_RESULTS )}
end
```

```
rule continue
when
 context : Context( state == Context.CHECK_DONE )
then
 context.setState( Context.ASSIGN_SEATS );
 update( context );
end
```

8.9.3. Output Summary

```
Assign First Seat

=>[fid:13:13]:[Seating id=1, pid=0, done=true, ls=1, ln=n5, rs=1, rn=n5]
=>[fid:14:14]:[Path id=1, seat=1, guest=n5]

==>[ActivationCreated(16): rule=findSeating
[fid:13:13]:[Seating id=1, pid=0, done=true, ls=1, ln=n5, rs=1, rn=n5]
[fid:9:9]:[Guest name=n5, sex=f, hobbies=h1]
[fid:1:1]:[Guest name=n1, sex=m, hobbies=h1]

==>[ActivationCreated(16): rule=findSeating
[fid:13:13]:[Seating id=1 , pid=0, done=true, ls=1, ln=n5, rs=1, rn=n5]
[fid:9:9]:[Guest name=n5, sex=f, hobbies=h1]
[fid:5:5]:[Guest name=n4, sex=m, hobbies=h1]*

Assign Seating

=>[fid:15:17] :[Seating id=2 , pid=1 , done=false, ls=1, lg=n5, rs=2,
 rn=n4]
=>[fid:16:18]:[Path id=2, seat=2, guest=n4]
=>[fid:17:19]:[Chosen id=1, name=n4, hobbies=h1]
```

```
=>[ActivationCreated(21): rule=makePath
[fid:15:17] : [Seating id=2, pid=1, done=false, ls=1, ln=n5, rs=2, rn=n4]
[fid:14:14] : [Path id=1, seat=1, guest=n5]*

==>[ActivationCreated(21): rule=pathDone
[Seating id=2, pid=1, done=false, ls=1, ln=n5, rs=2, rn=n4]*

Make Path

=>[fid:18:22:[Path id=2, seat=1, guest=n5]]

Path Done

Continue Process

=>[ActivationCreated(25): rule=findSeating
[fid:15:23]:[Seating id=2, pid=1, done=true, ls=1, ln=n5, rs=2, rn=n4]
[fid:7:7]:[Guest name=n4, sex=f, hobbies=h3]
[fid:4:4] : [Guest name=n3, sex=m, hobbies=h3]*

=>[ActivationCreated(25): rule=findSeating
[fid:15:23]:[Seating id=2, pid=1, done=true, ls=1, ln=n5, rs=2, rn=n4]
[fid:5:5]:[Guest name=n4, sex=m, hobbies=h1]
[fid:2:2]:[Guest name=n2, sex=f, hobbies=h1], [fid:12:20] : [Count value=3]

=>[ActivationCreated(25): rule=findSeating
[fid:13:13]:[Seating id=1, pid=0, done=true, ls=1, ln=n5, rs=1, rn=n5]
[fid:9:9]:[Guest name=n5, sex=f, hobbies=h1]
[fid:1:1]:[Guest name=n1, sex=m, hobbies=h1]

Assign Seating

=>[fid:19:26]:[Seating id=3, pid=2, done=false, ls=2, lnn4, rs=3, rn=n3]]
=>[fid:20:27]:[Path id=3, seat=3, guest=n3]]
=>[fid:21:28]:[Chosen id=2, name=n3, hobbies=h3}]

=>[ActivationCreated(30): rule=makePath
[fid:19:26]:[Seating id=3, pid=2, done=false, ls=2, ln=n4, rs=3, rn=n3]
[fid:18:22]:[Path id=2, seat=1, guest=n5]*

=>[ActivationCreated(30): rule=makePath
[fid:19:26]:[Seating id=3, pid=2, done=false, ls=2, ln=n4, rs=3, rn=n3]
[fid:16:18]:[Path id=2, seat=2, guest=n4]*

=>[ActivationCreated(30): rule=done
[fid:19:26]:[Seating id=3, pid=2, done=false, ls=2, ln=n4, rs=3, rn=n3]*

Make Path

=>[fid:22:31]:[Path id=3, seat=1, guest=n5]
```

```
Make Path

=>[fid:23:32] [Path id=3, seat=2, guest=n4]

Path Done

Continue Processing

=>[ActivationCreated(35): rule=findSeating
[fid:19:33]:[Seating id=3, pid=2, done=true, ls=2, ln=n4, rs=3, rn=n3]
[fid:4:4]:[Guest name=n3, sex=m, hobbies=h3]
[fid:3:3]:[Guest name=n2, sex=f, hobbies=h3], [fid:12:29]*

=>[ActivationCreated(35): rule=findSeating
[fid:15:23]:[Seating id=2, pid=1, done=true, ls=1, ln=n5, rs=2, rn=n4]
[fid:5:5]:[Guest name=n4, sex=m, hobbies=h1]
[fid:2:2]:[Guest name=n2, sex=f, hobbies=h1]

=>[ActivationCreated(35): rule=findSeating
[fid:13:13]:[Seating id=1, pid=0, done=true, ls=1, ln=n5, rs=1, rn=n5]
[fid:9:9]:[Guest name=n5, sex=f, hobbies=h1], [fid:1:1] : [Guest name=n1,
 sex=m, hobbies=h1]

Assign Seating

=>[fid:24:36]:[Seating id=4, pid=3, done=false, ls=3, ln=n3, rs=4, rn=n2]]
=>[fid:25:37]:[Path id=4, seat=4, guest=n2]]
=>[fid:26:38]:[Chosen id=3, name=n2, hobbies=h3]

==>[ActivationCreated(40): rule=makePath
[fid:24:36]:[Seating id=4, pid=3, done=false, ls=3, ln=n3, rs=4, rn=n2]
[fid:23:32]:[Path id=3, seat=2, guest=n4]*

==>[ActivationCreated(40): rule=makePath
[fid:24:36]:[Seating id=4, pid=3, done=false, ls=3, ln=n3, rs=4, rn=n2]
[fid:20:27]:[Path id=3, seat=3, guest=n3]*

=>[ActivationCreated(40): rule=makePath
[fid:24:36]:[Seating id=4, pid=3, done=false, ls=3, ln=n3, rs=4, rn=n2]
[fid:22:31]:[Path id=3, seat=1, guest=n5]*

=>[ActivationCreated(40): rule=done
[fid:24:36]:[Seating id=4, pid=3, done=false, ls=3, ln=n3, rs=4, rn=n2]*

Make Path

=>fid:27:41:[Path id=4, seat=2, guest=n4]

Make Path
=>fid:28:42]:[Path id=4, seat=1, guest=n5]]

Make Path

=>fid:29:43]:[Path id=4, seat=3, guest=n3]]

Path Done

Continue Processing

=>[ActivationCreated(46): rule=findSeating
[fid:15:23]:[Seating id=2, pid=1, done=true, ls=1, ln=n5, rs=2, rn=n4]
[fid:5:5]:[Guest name=n4, sex=m, hobbies=h1], [fid:2:2]
[Guest name=n2, sex=f, hobbies=h1]

=>[ActivationCreated(46): rule=findSeating
[fid:24:44]:[Seating id=4, pid=3, done=true, ls=3, ln=n3, rs=4, rn=n2]
[fid:2:2]:[Guest name=n2, sex=f, hobbies=h1]
[fid:1:1]:[Guest name=n1, sex=m, hobbies=h1]*

=>[ActivationCreated(46): rule=findSeating
[fid:13:13]:[Seating id=1, pid=0, done=true, ls=1, ln=n5, rs=1, rn=n5]
[fid:9:9]:[Guest name=n5, sex=f, hobbies=h1]
[fid:1:1]:[Guest name=n1, sex=m, hobbies=h1]

Assign Seating

=>[fid:30:47]:[Seating id=5, pid=4, done=false, ls=4, ln=n2, rs=5, rn=n1]
=>[fid:31:48]:[Path id=5, seat=5, guest=n1]
=>[fid:32:49]:[Chosen id=4, name=n1, hobbies=h1]
```

Appendix A. Revision History

Revision 1.0	Tue May 5 2009	Darrin Mison *dmison@redhat.com*
Published		