

JBoss Enterprise SOA Platform 5.2

JBoss Rules 5 Reference Guide

for JBoss Programmers and Business Rules Developers



JBoss Enterprise SOA Platform 5.2 JBoss Rules 5 Reference Guide

for JBoss Programmers and Business Rules Developers

Edition 5.2.0

Portions of this book are based on the *Drools Expert User Guide*, written by Mark Proctor, Michael Neale, Edson Tirelli and the Drools open source community. Further details about Drools can be found at the project's website <http://www.jboss.org/drools>.

Copyright © 2011 Red Hat, Inc..

The text of and illustrations in this document are licensed by Red Hat under the Apache Software License, Version 2. You may obtain a copy of the License at <http://www.apache.org/licenses/LICENSE-2.0>.

Red Hat, Red Hat Enterprise Linux, the Shadowman logo, JBoss, MetaMatrix, Fedora, the Infinity Logo, and RHCE are trademarks of Red Hat, Inc., registered in the United States and other countries.

Linux® is the registered trademark of Linus Torvalds in the United States and other countries.

All other trademarks are the property of their respective owners.

Use this book as a reference guide when developing business rules for your company.

| | |
|---|------------|
| Preface | vii |
| 1. Document Conventions | vii |
| 1.1. Typographic Conventions | vii |
| 1.2. Pull-quote Conventions | viii |
| 1.3. Notes and Warnings | ix |
| 2. Getting Help and Giving Feedback | ix |
| 2.1. Do You Need Help? | ix |
| 2.2. Give us Feedback | x |
| 3. Acknowledgements | x |
| 1. Introduction | 1 |
| 1.1. What Is a Rule Engine? | 1 |
| 1.1.1. Introduction and Background | 1 |
| 1.2. Strong and Loose Coupling | 4 |
| 2. Quick Start | 5 |
| 2.1. The Basics | 5 |
| 2.1.1. Stateless Knowledge Sessions | 5 |
| 2.1.2. Stateful Knowledge Sessions | 9 |
| 2.2. A Little Theory | 12 |
| 2.2.1. Methods and Rules | 12 |
| 2.2.2. Cross-Products | 13 |
| 2.2.3. Activations, Agendas and Conflict Sets | 14 |
| 2.2.4. Inference | 18 |
| 2.2.5. Inference and TruthMaintenance | 19 |
| 2.3. Further Comments on Building and Deploying | 20 |
| 2.3.1. Using Change-Sets to Add Rules | 20 |
| 2.3.2. The Knowledge Agent | 22 |
| 3. User Guide | 23 |
| 3.1. Building | 23 |
| 3.1.1. Building with Code | 24 |
| 3.1.2. Building via Configurations and the Change-Set XML | 27 |
| 3.2. Deploying | 30 |
| 3.2.1. The KnowledgePackage and Knowledge Definitions | 30 |
| 3.2.2. Knowledge Bases | 31 |
| 3.2.3. In-Process Building and Deployment | 32 |
| 3.2.4. Building and Deployment as Separate Processes | 33 |
| 3.2.5. Stateful Knowledge Sessions and Knowledge Base Modifications | 34 |
| 3.2.6. KnowledgeAgent | 34 |
| 3.3. Running | 38 |
| 3.3.1. The Knowledge Base | 38 |
| 3.3.2. StatefulKnowledgeSession | 38 |
| 3.3.3. KnowledgeRuntime | 38 |
| 3.3.4. Agenda | 47 |
| 3.3.5. Event Model | 51 |
| 3.3.6. KnowledgeRuntimeLogger | 53 |
| 3.3.7. StatelessKnowledgeSession | 54 |
| 3.3.8. Commands and the CommandExecutor | 58 |
| 3.3.9. Marshaling | 66 |
| 4. The Rule Language | 69 |
| 4.1. Overview | 69 |
| 4.1.1. A rule file | 69 |
| 4.1.2. Structure of a Rule | 69 |
| 4.2. Keywords | 70 |

| | |
|--|------------|
| 4.3. Comments | 71 |
| 4.4. Error Messages | 71 |
| 4.4.1. 101: No viable alternative | 72 |
| 4.4.2. 102: Mismatched input | 73 |
| 4.4.3. 103: Failed predicate | 74 |
| 4.4.4. 104: Trailing semi-colon not allowed | 75 |
| 4.4.5. 105: Early Exit | 75 |
| 4.5. Package | 75 |
| 4.5.1. import | 76 |
| 4.5.2. global | 76 |
| 4.6. Functions | 78 |
| 4.7. Type Declaration | 79 |
| 4.7.1. Declaring New Types | 80 |
| 4.7.2. Declaring Metadata | 81 |
| 4.7.3. Declaring Metadata for Existing Types | 82 |
| 4.7.4. Accessing Declared Types from the Application Code | 82 |
| 4.8. Rule | 84 |
| 4.8.1. Rule Attributes | 85 |
| 4.8.2. Timers and Calendars | 88 |
| 4.8.3. Left-Hand Side Conditional Elements | 89 |
| 4.8.4. The Right-Hand Side | 115 |
| 4.9. Query | 117 |
| 4.10. Domain-Specific Languages | 118 |
| 4.10.1. When to Use a Domain-Specific Language | 118 |
| 4.10.2. Creating a Domain-Specific Language | 118 |
| 4.10.3. Managing a Domain-Specific Language | 119 |
| 4.10.4. Adding Constraints to Facts | 121 |
| 4.10.5. DSL and DSLR Reference | 122 |
| 4.10.6. The Transformation of a DSLR File | 124 |
| 4.10.7. String Transformation Functions | 125 |
| 4.10.8. Domain-Specific Languages in the BRMS and in the IDE | 125 |
| 4.11. XML Rule Language | 126 |
| 4.11.1. When to use XML | 126 |
| 4.11.2. The XML format | 126 |
| 4.11.3. Automatic transforming between formats (XML and DRL) | 130 |
| 5. Using Spreadsheet Decision Tables | 131 |
| 5.1. When Should Decision Tables be Used? | 131 |
| 5.2. Overview | 131 |
| 5.3. How Decision Tables Work | 133 |
| 5.4. Keywords and Syntax | 136 |
| 5.4.1. Template Syntax | 136 |
| 5.4.2. Keywords | 139 |
| 5.5. Creating and Integrating Spreadsheet Based Decision Tables | 142 |
| 5.6. Managing Business Rules in Decision Tables | 143 |
| 5.6.1. Workflow and Collaboration | 143 |
| 5.6.2. Using Spreadsheet Features | 143 |
| 6. The Java Rule Engine Application Programming Interface | 145 |
| 6.1. Introduction | 145 |
| 6.2. How To Use the API | 145 |
| 6.2.1. Building and Registering RuleExecutionSets | 145 |
| 6.2.2. Using "Stateful" and "Stateless" Rule Sessions | 147 |
| 6.2.3. Globals | 148 |
| 6.3. References | 149 |

| | |
|--|------------|
| 7. JBoss Developer Studio | 151 |
| 7.1. Overview | 152 |
| 7.2. Drools Runtimes | 152 |
| 7.2.1. Defining a Drools Runtime | 152 |
| 7.2.2. Selecting a runtime for your Drools project | 155 |
| 7.3. Creating a Rule Project | 157 |
| 7.4. Creating a New Rule and Wizards | 159 |
| 7.5. Textual Rule Editor | 161 |
| 7.6. The Guided Editor | 163 |
| 7.7. JBoss Rules Views | 164 |
| 7.7.1. The Working Memory View | 165 |
| 7.7.2. The Audit View | 165 |
| 7.8. Domain-Specific Languages | 167 |
| 7.8.1. Editing languages | 167 |
| 7.9. The Rete View | 169 |
| 7.10. Large .dr1 Files | 170 |
| 7.11. Debugging Rules | 170 |
| 7.11.1. Creating Breakpoints | 170 |
| 7.11.2. Debugging Rules | 171 |
| 8. Examples | 175 |
| 8.1. HelloWorld Example | 175 |
| 8.2. State Example | 181 |
| 8.2.1. Understanding the State Example | 181 |
| 8.3. Fibonacci Example | 187 |
| 8.4. Banking Tutorial | 191 |
| 8.5. Pricing Rule Decision Table Example | 202 |
| 8.5.1. Executing the Example | 202 |
| 8.5.2. The Decision Table | 203 |
| 8.6. Pet Store Example | 205 |
| 8.7. Sudoku Example | 215 |
| 8.7.1. Overview of Sudoku | 216 |
| 8.7.2. Running the Example | 216 |
| 8.7.3. Java Source and Rules Overview | 219 |
| 8.7.4. Validation Rules | 220 |
| 8.7.5. Solving Rules | 220 |
| 8.7.6. Suggestions for Future Developments | 222 |
| 8.8. Number Guess | 223 |
| 8.9. Miss Manners and Benchmarking | 229 |
| 8.9.1. Introduction | 229 |
| 8.9.2. In-Depth Analysis | 231 |
| 8.9.3. Summary of Output | 236 |
| 8.10. <i>Conway's Game Of Life</i> Example | 238 |
| A. © 2011 | 247 |
| B. Revision History | 249 |

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](https://fedorahosted.org/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 keycaps 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 keycap, all presented in mono-spaced bold and all distinguishable thanks to context.

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

Press **Enter** to execute the command.

Press **Ctrl+Alt+F2** to switch to the first virtual terminal. Press **Ctrl+Alt+F1** to return to your X-Windows session.

The first paragraph highlights the particular keycap to press. The second highlights two key combinations (each a set of three keycaps with 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; dialog box text; labeled buttons; check-box and radio button labels; menu titles and sub-menu titles. For example:

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

¹ <https://fedorahosted.org/liberation-fonts/>

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.

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:

Publican is a *DocBook* publishing system.

1.2. Pull-quote Conventions

Terminal output and source code listings 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 add syntax highlighting 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

Notes are tips, shortcuts or alternative approaches 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 a box labeled 'Important' will not cause data loss but may cause irritation and frustration.



Warning

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

2. Getting Help and Giving Feedback

2.1. Do You Need Help?

If you experience difficulty with a procedure described in this documentation, visit the Red Hat Customer Portal at <http://access.redhat.com>. Through the customer portal, you can:

- search or browse through a knowledgebase of technical support articles about Red Hat products.
- submit a support case to Red Hat Global Support Services (GSS).
- access other product documentation.

Red Hat also hosts a large number of electronic mailing lists for discussion of Red Hat software and technology. You can find a list of publicly available mailing lists at <https://www.redhat.com/mailman/listinfo>. Click on the name of any mailing list to subscribe to that list or to access the list archives.

2.2. Give us Feedback

If you find a typographical error, or know how this guide can be improved, we would love to hear from you. Submit a report in Bugzilla against the product **JBoss Enterprise SOA Platform** and the component **doc-JBoss_Rules_5_Reference_Guide**. The following link will take you to a pre-filled bug report for this product: <http://bugzilla.redhat.com/>².

Fill out the following template in Bugzilla's **Description** field. Be as specific as possible when describing the issue; this will help ensure that we can fix it quickly.

| |
|------------------------------|
| Document URL: |
| Section Number and Name: |
| Describe the issue: |
| Suggestions for improvement: |
| Additional information: |

Be sure to give us your name so that you can receive full credit for reporting the issue.

3. Acknowledgements

Certain portions of this text first appeared in the work *Drools Expert* by Mark Proctor, Michael Neale, and Edson Tirelli, copyright © 2010 JBoss Inc, available from <http://www.jboss.org/drools>.

JBoss Enterprise BRMS Platform JBoss Rules 5 Reference Guide edited by Darrin Mison (Red Hat) and David Le Sage (Red Hat).

² https://bugzilla.redhat.com/enter_bug.cgi?product=JBoss%20Enterprise%20SOA%20Platform%205&component=doc-JBoss_Rules_5_Reference_Guide&version=52

Introduction

1.1. What Is a Rule Engine?

1.1.1. Introduction and Background

JBoss Rules is an advanced artificial intelligence system that utilises Turing-complete *Rete algorithms* to create and interpret *production rules*. Study this book to learn how to use this system to write and modify business rules and procedures as they evolve over time. Once rules have been written, use the software to manage, deploy and analyse them.

Read this first section to gain a broad understanding of how the software works. This overview introduces basic terms and some theory, explaining the major aspects of the system.

The "brain" of a rules system is an *inference engine* that is able to scale to a large number of production rules and facts.

Inference Engine

An inference engine matches facts and data, against the rules, to infer conclusions which result in actions.

Production Rule

A production rule is a two-part structure that uses first order logic to represent knowledge:

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

Pattern Matching

Pattern matching is the process of matching facts against rules. It is performed by the inference engine, using Linear, Rete, Treat and Leaps algorithms.

ReteOO

The Rete implementation used is called **ReteOO**. This is an enhanced and optimized implementation of the Rete algorithm specifically for Object Oriented systems.

Conflict Resolution Strategy

If a system has a large number of rules, sometimes more than one may be true for the same fact assertion. If so, these rules are said to be in conflict. The agenda manages these situations by using a conflict resolution strategy to dictate the order in which they are to be executed.

The rules are stored in the production memory whilst the facts are *asserted* into the working memory. Once the facts are in the working memory, one can modify or retract them.

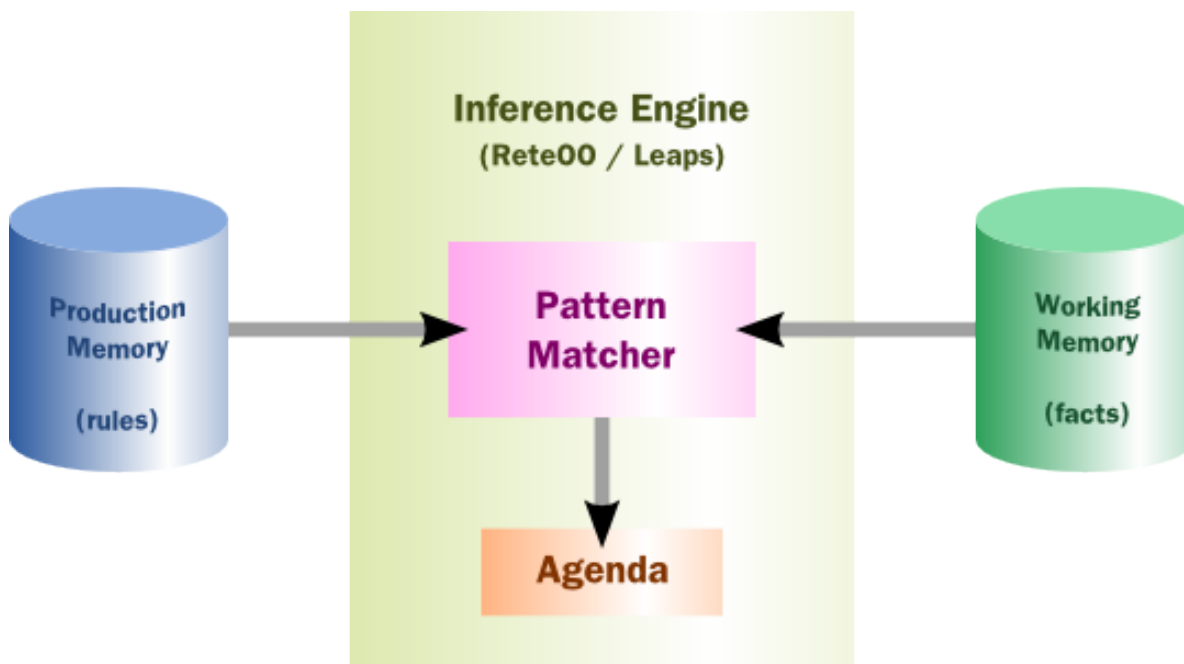


Figure 1.1. High-Level Overview of a Rules Engine

A production rule system's inference engine is *stateful* and is responsible for *truth maintenance*.
Truth maintenance

The inference engine's ability to enforce truthfulness.

Use actions to declare *logical relationships*.

Logical Relationship

A logical relationship exists when the action's state depends on the inference remaining true.
When it is no longer true, the dependent action is undone.

There are three types of production rule systems: *forward-chaining*, *backward-chaining* and *hybrids* which For

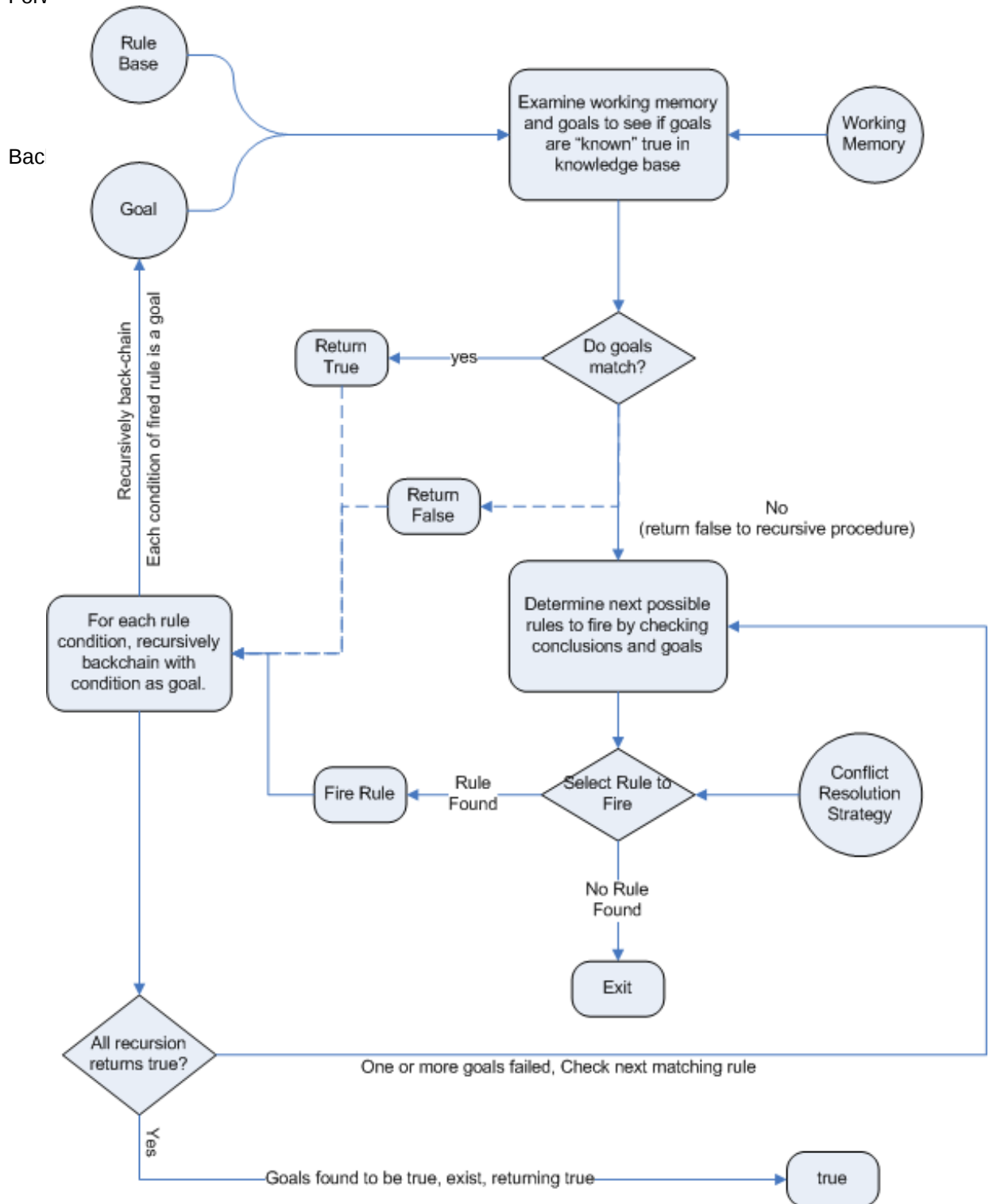


Figure 1.3. Backward-Chaining



Important

Understanding these modes of operation is the key to understanding the differences between systems and how best to optimize each of them.

1.2. Strong and Loose Coupling

Generally, a design exhibiting *loose-coupling* is preferable, as this grants a great deal of flexibility. If the rules are all strongly-coupled, they are likely to be inflexible. More significantly, it indicates that deploying a rule engine is overkill for the situation.

loose-coupling

A design in which the execution of one rule will not lead to the execution of another.

strong-coupling

If rules are strongly-coupled, it means that the firing of one rule will directly result in the firing of another and so on; in other words, there is a clear chain of logic.

A *clear chain* can be hard-coded, or implemented using a decision tree.



Note

Strong coupling is not inherently bad but remember the arguments against it when designing the way in which rules are to be captured.

A loosely-coupled system is more flexible and allows one to add, change and remove rules without a follow-on effect.

Quick Start

2.1. The Basics

New users sometimes find JBoss Rules a little overwhelming because there is so much functionality because the software has been designed to deal with many different use-cases. The purpose of this chapter is to introduce this functionality little by little. Some very simple examples are provided to help one learn.

2.1.1. Stateless Knowledge Sessions

The simplest use-case is known as a *stateless session*.
stateless session

A session without inference.

A stateless session can be called like a function, as one passes it some data and then receives the result back. There are many common use-cases for stateless sessions. Here are a few:

- validation, an example being, "Is this person eligible for a mortgage?"
- calculation, an example being, "Compute a mortgage premium for me."
- routing and filtering, an for example, "Filter my incoming messages into folders" or, "Send incoming messages to a destination."

Here is a simple example involving an application for a driver's license.

1. Gather the data needed, as this will form the set of *facts* that are going to be passed to the rule. In this case, there is only one piece of data:

```
package com.company.license;

public class Applicant
{
    private String name;
    private int age;
    private boolean valid;

    public Applicant (String name, int age, boolean valid)
    {
        this.name = name;
        this.age = age;
        this.valid = valid;
    }

    //add getters & setters here
}
```

2. Now that one has a data model, it is time to write a first rule. The purpose of this one will be to disqualify any applicant younger than eighteen years of age:

```
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, each rule's constraints evaluate it, looking for a match. (Note that there is always an implied constraint of "object type" after which there can be any number of explicit field constraints.)

Pattern

A collection of constraints is known as a pattern.

Pattern Matching

The process whereby each rule's group of constraints evaluates an object, looking for a match.

Matched

When an inserted object satisfied all of the constraints for a rule, it is said to be matched.

For example, In the **Is of valid age** rule there are two constraints:

- a. The fact being matched must be of type **Applicant**.
- b. The value of **Age** must be less than eighteen.

\$a is a binding variable. It exists to make possible a reference to the matched object in the rule's consequence (from which place the object's properties can be updated.)



Note

Use of the dollar sign (\$) is optional. It helps to differentiate between variable names and field names.



Note

Just for the moment, assume that the rules are in the same folder as the classes, so that the *class-path resource loader* can be used to build the first *knowledge base*.

Knowledge Base

A knowledge base is a collection of rules which have been compiled by the **KnowledgeBuilder**.

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

The piece of code quoted above uses the `newClassPathResource()` method to search the class-path for the **licenseApplication.drl** file. The **ResourceType** is written in the *Drools Rule Language*.

Drools Rule Language

Drools Rule Language (DRL) is JBoss Rules' native rules language.

4. Check the **KnowledgeBuilder** for any errors. If there are none, one is now ready to build the session.
5. Execute the data against the rules. (Since the applicant is under the age of eighteen, their application will be marked as "invalid.")

```

KnowledgeBase kbase = KnowledgeBaseFactory.newKnowledgeBase();
kbase.addKnowledgePackages( kbuilder.getKnowledgePackages() );
StatelessKnowledgeSession ksession = kbase.newStatelessKnowledgeSession();
Applicant applicant = new Applicant( "Mr John Smith", 16, true );

assertTrue( applicant.isValid() );
ksession.execute( applicant );
assertFalse( applicant.isValid() );

```

So far, the data has consisted of but a single object. What if one wanted to use more than this? It is possible to execute against any object-implementing **iterable**, such as a **collection**. In this next example, one will be taught how to add another class called **Application**, which contains the date of the driver's licence application. Another skill one will be taught is how to move the Boolean field entitled **valid** to the **Application** class.

1. Here is the code:

```

public class Applicant {
    private String name;
    private int age;

    public Applicant (String name, int age)
    {
        this.name = name;
        this.age = age;
    }
    // getter and setter methods here
}

public class Application {
    private Date dateApplied;
    private boolean valid;

    public Application (boolean valid)
    {
        this.valid = valid;
    }
    // getter and setter methods here
}

```

2. In order to check that the application was made within a legitimate time-frame, add this rule:

```

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
```

3. Unfortunately, Java arrays are unable to implement the `iterable` interface, so use the *JDK converter method* instead. (This method commences with the line, **Arrays.asList(...)**.)

The code shown below executes against an `iterable` list. Every collection element is 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(true);
assertTrue( application.isValid() );
ksession.execute( Arrays.asList( new Object[] {application, applicant} ));
assertFalse( application.isValid() );
```



Note

The `execute(Object object)` and `execute(Iterable objects)` methods are actually wrappers around a further method called `execute(Command command)` which comes from the `BatchExecutor` interface.

4. Use the **CommandFactory** to create instructions, so that the following is equivalent to `execute(Iterable it)`:

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

5. One will find the `BatchExecutor` and **CommandFactory** are particularly useful when working with many different commands and result output identifiers:

```
List<Command> cmds = new ArrayList<Command>();
cmds.add(
  CommandFactory.newInsertObject(new Person("Mr John Smith"), "mrSmith"));
cmds.add(
  CommandFactory.newInsertObject(new Person( "Mr John Doe" ), "mrDoe" ));

ExecutionResults results =
  ksession.execute( CommandFactory.newBatchExecution(cmds) );

assertEquals( new Person("Mr John Smith"), results.getValue("mrSmith" ) );
```



Note

CommandFactory supports many other commands that can be used in the `BatchExecutor`. Some of these are **StartProcess**, **Query** and **SetGlobal**.

2.1.2. Stateful Knowledge Sessions

Here are some of the many common use cases for *stateful sessions*:

Stateful Session

Stateful sessions allow one to make iterative changes to facts over time.

- monitoring: for instance, one can monitor a stock market and automate the buying process.
- diagnostics: for instance, one can use it to run fault-finding processes. It could also be used for medical diagnostic processes.
- logistical: for instance, it could be applied to problems involving parcel tracking and delivery provisioning.
- ensuring compliance: for instance, it could be used to validate the legality of market trades.



Warning

Ensure that the `dispose()` method is called afterward running a stateful session. This is to ensure that there are no memory leaks. This is due to the fact that knowledge bases will obtain references to stateful knowledge sessions when they are created.

As with the **StatelessKnowledgeSession**, the **StatefulKnowledgeSession** supports the `BatchExecutor` interface, the only difference being that, in this case, the **FireAllRules** command is not automatically called at the end.

To illustrate the "monitoring" use-case, here is an example involving the development of a fire alarm system.

1. Create a model representing the rooms in the 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 class 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
{
}
```

2. The rules must express the relationships between multiple objects, (to define things such as the presence of a sprinkler in a certain room.)

Achieve this by using a binding variable as a constraint in a pattern. Doing so results in a *cross-product*.

3. Create an instance of the **Fire** class. Insert the instance into the session.

The rule below adds a binding to the **Fire** object's room field to constrain matches. This so that only the sprinkler for that room is checked. When this rule fires and the consequence executes, the sprinkler activates:

```
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
```

Whereas the stateless session employed standard Java syntax to modify a field, the rule above uses the **modify** statement. (It acts much like a "with" statement.)

It contains a series of comma-separated Java expressions, which are, to all intents and purposes, calls to those object "**setters**" that have been selected by the **modify** statement's *control expression*. These **setters** modify the data and then make the engine aware of the changes so that it can "reason" its way through them once more. This process is known as *inference* and it is the key to understanding how a stateful session's operates. (By contrast, stateless sessions do not use inference, so the engine does not need to be aware of changes to data.)



Note

To deactivate inference use the *sequential mode*.

The tutorials have, thus far, show rules that operate when matches exist but what happens when nothing matches? How does one determine that a fire has been extinguished? The previous constraints were "sentences" according to *propositional logic*, whereby the engine constrains individual instances. However, **JBoss Rules** also supports *first order logic*, which allows one to look at sets of data. A pattern featuring the keyword **not** matches only when something does not exist.

This rule turns the sprinkler off when the fire is extinguished:

```
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
```

Whilst there is one sprinkler for each room, there is just one alarm for the entire building. An **Alarm** object is created when there is a fire, but only one **Alarm** is needed for the entire building, no matter how many fires there might be. **not**'s complement, **exists** can now be introduced. It matches one or more instances of a category:

```

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

```

If there are no more fires, the alarm must be deactivated. To turn it off, use **not** again:

```

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, this code print a general health status message when the application first starts and also when the alarm and all of the sprinklers have been deactivated:

```

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

```

Store the rules in a file called **fireAlarm.drl**. Save this file in a sub-directory on the class-path. Now build a knowledge base, using the new name, **fireAlarm.drl**:

```

KnowledgeBuilder kbuilder = KnowledgeBuilderFactory.newKnowledgeBuilder();
kbuilder.add( ResourceFactory.newClassPathResource( "fireAlarm.drl",
    getClass() ), ResourceType.DRL );

if ( kbuilder.hasErrors() )
    System.err.println( kbuilder.getErrors().toString() );

StatefulKnowledgeSession ksession = kbase.newStatefulKnowledgeSession();

```

In this next example, four room objects are created and inserted, along with one sprinkler object for each room. (The matching process has ended but no rules have yet been fired.)

1. Call `ksession.fireAllRules()`. This grants the matched rules permission to run but, since there is no fire, they will merely produce the health message:

```

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();
```

```
> Everything is Okay
```

2. Now create and insert two fires. (A *fact handle* will be kept.)

fact handle

A fact handle is an internal reference to the inserted instance. It allows instances to be retracted or modified at a later point in time.

3. With the **fires** now in the engine, call `fireAllRules()`. The alarm will be raised and the respective sprinklers will be 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
```

4. When the fires are extinguished, the fire objects are retracted and the sprinklers are turned off. At this point in time, the alarm is canceled and the health message displays once more:

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

ksession.fireAllRules();
```

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

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

These simple examples demonstrate some of the functionality of the rule system and given the user some idea of how to program for it.

2.2. A Little Theory

2.2.1. Methods and Rules

New users often confuse methods and rules. To summarise methods:

- they are called directly
- specific instances are passed

- a single call results in a single execution

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

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

To summarise rules:

- they execute by matching against any data that has been inserted into the engine
- they can never be called directly
- specific instances cannot be passed to a rule
- depending on the matches, a rule may fire once, several times or never at all

2.2.2. Cross-Products

Cross-products

When two or more sets of data are combined, the result is called a cross-product.

Consider the following rule from 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
```

This is analogous with the Structured Query Language command to **select * from Room, Sprinkler**, which instructs every row in the **Room** table to join every row in the **Sprinkler** table, thereby resulting in the following output:

```
room:office sprinkler: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
```

Cross-products can become huge and, therefore, have the potential to cause performance problems. To prevent this, use variable constraints to eliminate nonsensical results:

```
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** assigned to each **Room**. As written in SQL, the corresponding query would be **select * from Room, Sprinkler where Room == Sprinkler.room**

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

2.2.3. Activations, Agendas and Conflict Sets

So far, the data and the matching process have been small and relatively simple. However, over time, one will insert many facts and rules. At this point, the `rule` engine will need a way to manage the execution of outcomes. JBoss Rules achieves this using *activations*, *agendas* and a *conflict resolution strategy*.

This next, more complex example demonstrates the handling of cash-flow calculations over multiple date periods.



Note

It is assumed that one is comfortable with the Java code needed to create knowledge bases and populating a **StatefulKnowledgeSession** with facts, so that code will not be repeated here.

Diagrams will illustrate the state of the `rule` engine at key stages.

The data model consists of three classes, **Cashflow**, **Account** and **AccountPeriod**:

```
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
}
```

By now, you already know how to create knowledge bases and how to instantiate facts to populate the **StatefulKnowledgeSession**. Therefore, tables will be used to show the state of the inserted data, as this makes things clearer for illustrative purposes. The tables below show that a single fact was inserted for the **Account**. A series of debits and credits extending over two quarters were also inserted into the **Account** as **Cashflow** objects.

Figure 2.1, “Cash-Flows and the 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. Cash-Flows and the Account

The two rules which follow are used to, firstly, determine the debit and credit totals for the specified period and, secondly, update the account balance. (The && operator is used to avoid the need to repeat 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.setBalance(acc.getBalance() + $amount);
end
```

```
rule "decrease balance for debits"
when
    ap : AccountPeriod()
    acc : Account( $accountNo : accountNo )
```

Chapter 2. Quick Start

```

CashFlow( type == DEBIT,
  accountNo == $accountNo,
  date >= ap.start && <= ap.end,
  $amount : amount )
then
  acc.setBalance(acc.getBalance() - $amount);
end

```

As shown in [Figure 2.2, "Cash-Flows and the Account"](#), the accounting period start date is set to the 1st of January and the end is set to the 31st of March. This constrains the data to two **Cashflow** objects for credit and one for debit.

| 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. Cash-Flows and the Account

1. 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*.
2. Each activation is added to a list called the *agenda*.
3. Each activation on the agenda is executed when the `fireAllRules()` method 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. Cash-Flows and the Account

After all of the activations noted above are fired, the account will have a balance of minus twenty-five.

| Account | |
|-----------|---------|
| accountNo | balance |
| 1 | -25 |

Figure 2.4. Cash-Flows and the Account

If the accounting period is updated to the second quarter, one will only have a single matched row of data and, thus, 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. Cash-Flows and the Account

When the activation fires, the result will be a balance of twenty-five.

| accountNo | balance |
|-----------|---------|
| 1 | 25 |

Figure 2.6. Cash-Flows and the Account


When there are one or more activations on the agenda, they are said to be "in conflict", and a *conflict resolution 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 zero and the higher the value, the higher the priority shall be. The salience can also be a negative value. This lets one order the execution of rules relative to each other.


Note

The execution order for rules with the same salience value is still arbitrary. To illustrate this, the next step is to 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, thus, it will execute after the rules with the default salience value of zero.

```
rule "Print balance for AccountPeriod"
  salience -50
  when
    ap : AccountPeriod()
    acc : Account()
  then
    System.out.println( acc.getAccountNo() + " : " + acc.getBalance() );
  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, so that it will execute afterwards.


Important

JBoss Rules includes ruleflow-group attributes. Use these to declare work-flow diagrams in order to specify when rules can be fired. The screen-shot below is taken from the **JBoss Developer Studio**. It has two ruleflow-group nodes. These ensure that the calculation rules are executed before the reporting rules.

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

Figure 2.7. Cash-Flows and Account

2.2.4. Inference

Inference is the act of using one piece of data to infer something else. For instance, given a **Person** fact with an *age* field and a rule that provides age policy control, we can infer whether a Person is an adult or a child and act on this.

Example 2.1. Inferring Adulthood

```
rule "Infer Adult"
when
  $p : Person( age >= 18 )
then
  insert( new IsAdult( $p ) )
end
```

Every **Person** who is 18 or over will have an instance of *IsAdult* inserted for them. This kind of fact is known as a relation. Relations can use this inferred relation in any rule:

```
$p : Person()
IsAdult( person == $p )
```

2.2.4.1. Inference in Action

Example Government Department is responsible for issuing ID cards when people become adults. The ID department uses a decision table includes logic that states, when an adult living in London is 18 years old or over, issue the card:

| | RuleTable ID Card | | |
|-------------------------|-------------------|---------------|--------------------|
| | CONDITION | CONDITION | ACTION |
| | p : Person | | |
| | location | age >= 18 | issueIdCard(\$1) |
| | Select Person | Select Adults | Issue ID Card |
| Issue ID Card to Adults | London | 18 | p |

Figure 2.8. Monolithic Decision Table

The ID department does not set the policy on who an adult is. If the central government changes the age a person is considered to be an adult to 21, there is a change management process and this need to be communicated to the ID department to ensure their systems are updated to reflect the change.

The change management process can be costly and introduce errors. The ID department is maintaining more information than it needs with the decision table in [Figure 2.8, "Monolithic Decision Table"](#), by storing the age at which a person is considered an adult, the ID department must keep this information up to date.

It is possible to split, or de-couple, the authoring responsibilities, with each department maintaining their own rules. In effect this means, if the central government changes the age a person is considered an adult, the central government updates their central repository with the new rules, which others (the ID department) use:

| RuleTable Age Policy | | |
|----------------------|------------------|--------------------|
| CONDITION | ACTION | |
| p : Person | | |
| age >= \$1 | insert(\$1) | |
| | Adult Age Policy | Add Adult Relation |
| Infer Adult | 18 | new IsAdult(p) |

Figure 2.9. Rule Table Age Policy

The *IsAdult* fact, as discussed previously, is inferred from the policy rules. Because the central government now maintains the *IsAdult* fact, the ID department only needs to know if the person is an adult or not, and do not need to maintain their rules to stay inline with current policy.

| RuleTable ID Card | | | |
|-------------------------|---------------|--------------------|---------------|
| CONDITION | CONDITION | ACTION | |
| p : Person | IsAdult | | |
| location | person == \$1 | issueIdCard(\$1) | |
| | Select Person | Select Adults | Issue ID Card |
| Issue ID Card to Adults | London | p | p |

Figure 2.10. Rule Table ID Card

2.2.5. Inference and TruthMaintenance

Truth Maintenance and Logical Inserts (*logicalInsert*) can be used to provide a separation of concerns. The following example issues either a child or adult bus pass:

```

rule "Issue Child Bus Pass" when
    $p : Person( age < 16 )
then
    insert(new ChildBusPass( $p ) );
end

rule "Issue Adult Bus Pass" when
    $p : Person( age >= 16 )
then
    insert(new AdultBusPass( $p ) );
end
    
```

A separation of concerns can be achieved by using *logicalInsert*:

```

rule "Infer Child" when
    $p : Person( age < 16 )
then
    logicalInsert( new IsChild( $p ) )
end
    
```

```
rule "Infer Adult" when
  $p : Person( age >= 16 )
then
  logicalInsert( new IsAdult( $p ) )
end
```

The fact is logically inserted, dependent on the truth of the *when* clause. If the truth of the *when* clause changes to false, the fact is automatically retracted. This works well for rules that are mutually exclusive, for instance, when the person's age changes from 15 to 16, the *IsChild* fact is automatically retracted and the *IsAdult* fact is inserted.

We can now bring back in the code to issue the passes, these two can also be logically inserted, as the TMS supports chaining of logical insertions for a cascading set of retracts.

```
rule "Issue Child Bus Pass" when
  $p : Person( )
  IsChild( person =$p )
then
  logicalInsert(new ChildBusPass( $p ) );
end

rule "Issue Adult Bus Pass" when
  $p : Person( age >= 16 )
  IsAdult( person =$p )
then
  logicalInsert(new AdultBusPass( $p ) );
end
```

LogicalInsert can be combined with the *not* conditional element to handle notifications, in this situation a request could be sent for the return of the bus pass. When the *ChildBusPass* object is retracted a rule triggers and sends a request to the person:

```
rule "Return ChildBusPass Request "when
  $p : Person( )
  not( ChildBusPass( person == $p ) )
then
  requestChildBusPass( $p );
end
```

2.3. Further Comments on Building and Deploying

2.3.1. Using Change-Sets to Add Rules

The examples so far have all used the JBoss Rules API to build knowledge bases. They have done so by manually adding each rule. JBoss Rules also provide a means to declare the resources to be added to a knowledge base through an **XML** file. This feature is called a *change-set*.

The change-set **XML** file contains a list of the rule resources that can be added to a knowledge base. One can also point this file to another.



Important

At the current moment in time, change-sets only support the `<add>` element. Red Hat will add support for the `<remove>` and `<modify>` elements in the future.

```
<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>
```

URLs specify the location of each resource. Every protocol provided by *java.net.URL* is supported. A protocol called *classpath* can also be used. This protocol refers to the current processes class-path for the resource.



Important

The type attribute must always be specified for a resource but it is not inferred from the file name extension.



Note

When using the XML above, note that the code is almost identical as that depicted before, with the exception that the *ResourceType* has been altered to **CHANGE_SET**.

```
KnowledgeBuilder kbuilder = KnowledgeBuilderFactory.newKnowledgeBuilder();
kbuilder.add( ResourceFactory.newClasspathResource( "myChangeSet.xml",
  getClass() ), ResourceType.CHANGE_SET );

if ( kbuilder.hasErrors() ) {
  System.err.println( kbuilder.getErrors().toString() );
}
```

Change-sets can include any number of resources. One can also add additional configuration information for decision tables to them. The example below loads rules from both an HTTP uniform resource locator and from a decision table spreadsheet via the class-path 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>
```

To add all of the files in a directory, use its name as the resource source. (Note that all of 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. The Knowledge Agent

The **KnowledgeAgent** automatically loads, re-loads and caches rule resources. Configure it via its **properties** file.

If the resources used by a knowledge base change, the **KnowledgeAgent** can update or rebuild it. To set a strategy for these updates, re-configure the **KnowledgeAgentFactory**:

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

The **KnowledgeAgent** scans every resource added, the default polling interval being sixty seconds. If the "last-modified" date of a resource has changed, the **KnowledgeAgent** will rebuild the knowledge base. (If a directory has been set as one of the resources, then every contents of that directory will be scanned for changes.)



Important

The previous knowledge base reference will still exist after change, so one must call `getKnowledgeBase()` to access the newly-built version.

Having studied this chapter, the reader now understands how this software works in a little more detail, having learned of the differences between methods and rules, seen how agendas, activations and conflict-sets come into play and been taught how the **KnowledgeAgent** and knowledge bases interact. The reader should also now have a a more comprehensive understanding of stateless and stateful sessions.

User Guide

3.1. Building

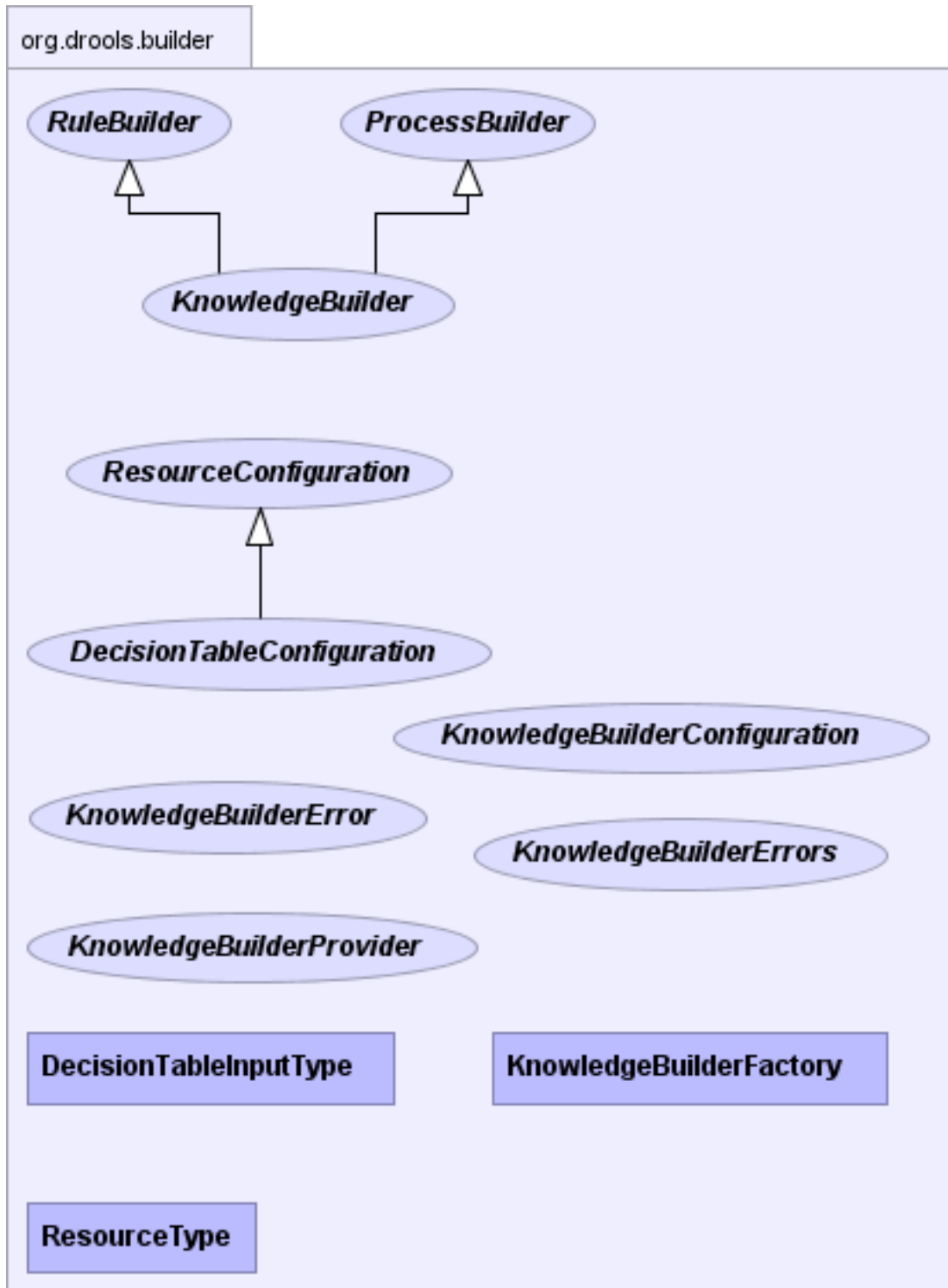


Figure 3.1. org.drools.builder

3.1.1. Building with Code

The Knowledge Builder is responsible for taking source data and turning it into a *knowledge package*. A knowledge package contains rule and process definitions which the **Knowledge Base** then consumes.



Note

As its name implies, the **ResourceType** object class indicates the type of resource being built.

The **ResourceFactory** provides the capability to load a resource from a number of sources, including a **reader**, a class-path, a uniform resource locator, a file or a **ByteArray**.



Important

When dealing with binaries, (such as decision tables), do not use a **Reader**-based resource handler. These are only suitable for use with plain text.

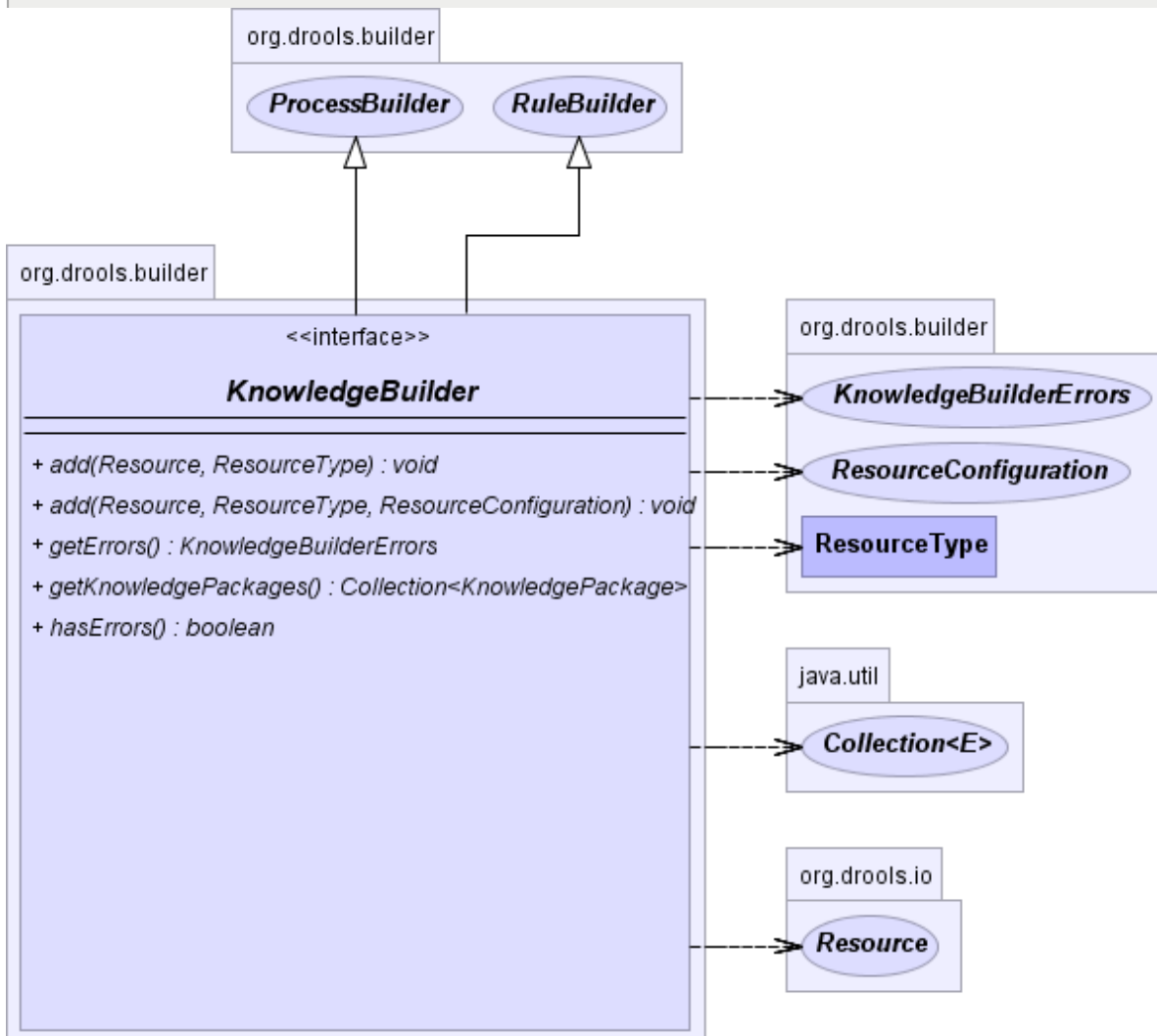


Figure 3.2. KnowledgeBuilder



Note

The Knowledge Builder is created by the **KnowledgeBuilderFactory**.

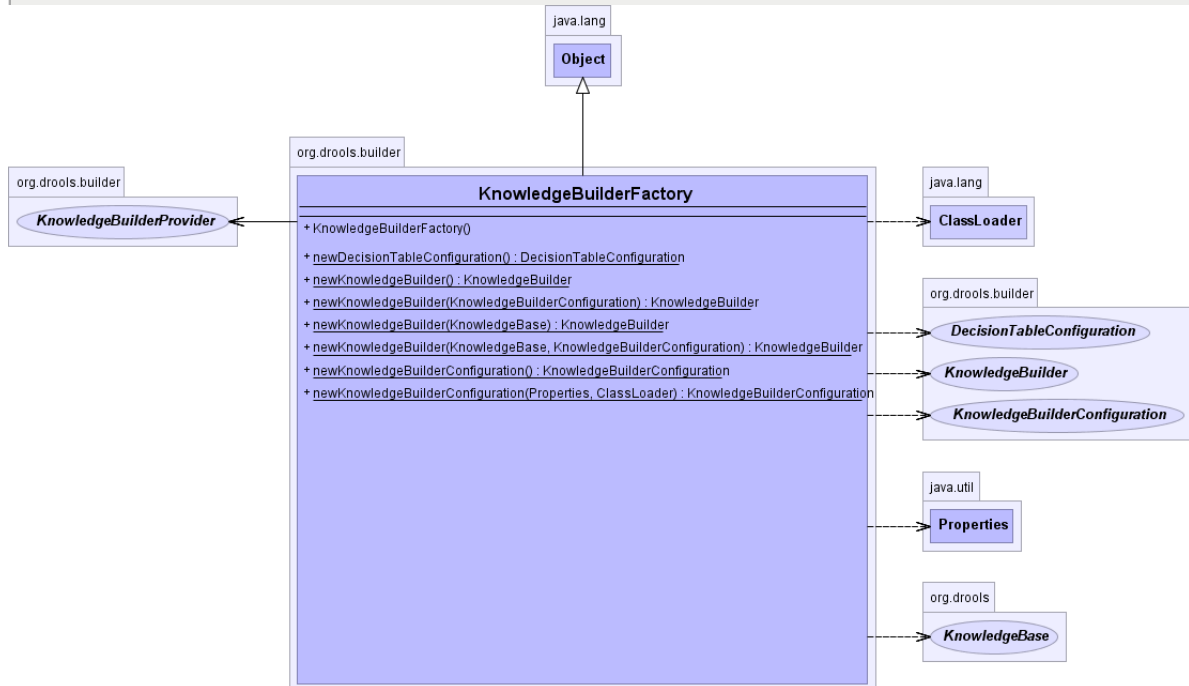


Figure 3.3. KnowledgeBuilderFactory

Create a Knowledge Builder by using the default configuration:

Example 3.1. Creating a new Knowledge Builder

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

Create a configuration by using the **KnowledgeBuilderFactory**. Such a configuration allows one to modify the behaviour of the Knowledge Builder.



Note

Many users do this to provide a custom *class loader* that allows the Knowledge Builder object to resolve classes that are not in the default path.

The first parameter is for properties. It is optional and can, therefore, be left null, in which case the default options will be used. The options parameter can be used for such tasks as changing the dialect and registering new accumulator functions.

Example 3.2. Creating a new Knowledge Builder with a Custom Class Loader

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

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

Resources of any type can be added on an iterative basis. In the example below, a **.drl** file is added.



Note

The Knowledge Builder can now handle multiple name-spaces, which was not the case with JBoss Rules 4.0 Package Builder. Therefore, one can just keep adding resources, regardless of the name-space.

Example 3.3. Adding DRL Resources

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



Important

Always check the `hasErrors()` method after making an addition. Do not add more resources or retrieve the **Knowledge Packages** if there are errors. (`getKnowledgePackages()` returns an empty list if there are errors.)

Example 3.4. Validating

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

Once all the resources have been added and there are no longer any errors, retrieve the *collection* of **Knowledge Packages**. (This is termed a "collection" because there is one **Knowledge Package** per package name-space.) These **Knowledge Packages** are *serializable* and are often used as a unit of deployment.

Example 3.5. Obtaining the Knowledge Packages

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

The final example combines all of these elements:

Example 3.6. Combining All Elements

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

```

KnowledgeBuilder kbuilder = KnowledgeBuilderFactory.newKnowledgeBuilder();
kbuilder.add( ResourceFactory.newFileResource( "/project/myrules1.dr1" ),
             ResourceType.DRL);
kbuilder.add( ResourceFactory.newFileResource( "/project/myrules2.dr1" ),
             ResourceType.DRL);

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

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

```

3.1.2. Building via Configurations and the Change-Set XML

It is possible to create definitions via configurations, rather than programming them by adding resources. You do so via the Change-Set XML. The simple XML file supports three elements: add, remove, and modify, each of which has a sequence of resource sub-elements which serve to define a configuration entity.



Warning

The following XML schema is not *normative*: it is included for illustrative purposes only.

Example 3.7. Schema for Change-Set XML (Not "Normative")

```

<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
           xmlns="http://drools.org/drools-5.0/change-set"
           targetNamespace="http://drools.org/drools-5.0/change-set">

  <xs:element name="change-set" type="ChangeSet"/>

  <xs:complexType name="ChangeSet">
    <xs:choice maxOccurs="unbounded">
      <xs:element name="add" type="Operation"/>
      <xs:element name="remove" type="Operation"/>
      <xs:element name="modify" type="Operation"/>
    </xs:choice>
  </xs:complexType>

  <xs:complexType name="Operation">
    <xs:sequence>
      <xs:element name="resource" type="Resource"
                 maxOccurs="unbounded"/>
    </xs:sequence>
  </xs:complexType>

  <xs:complexType name="Resource">
    <xs:sequence>
      <!-- To be used with <resource type="DTABLE"...&gt; -->
      <xs:element name="decisiontable-conf" type="DecTabConf"
                 minOccurs="0"/>
    </xs:sequence>
    <!-- java.net.URL, plus "classpath" protocol -->
    <xs:attribute name="source" type="xs:string"/>
    <xs:attribute name="type" type="ResourceType"/>
  </xs:complexType>

  <xs:complexType name="DecTabConf">
    <xs:attribute name="input-type" type="DecTabInpType"/>
  </xs:complexType>

```

```

<xs:attribute name="worksheet-name" type="xs:string"
              use="optional"/>
</xs:complexType>

<!-- according to org.drools.builder.ResourceType -->
<xs:simpleType name="ResourceType">
  <xs:restriction base="xs:string">
    <xs:enumeration value="DRL"/>
    <xs:enumeration value="XDRL"/>
    <xs:enumeration value="DSL"/>
    <xs:enumeration value="DSLX"/>
    <xs:enumeration value="DRF"/>
    <xs:enumeration value="DTABLE"/>
    <xs:enumeration value="PKG"/>
    <xs:enumeration value="BRL"/>
    <xs:enumeration value="CHANGE_SET"/>
  </xs:restriction>
</xs:simpleType>

<!-- according to org.drools.builder.DecisionTableInputType -->
<xs:simpleType name="DecTabInpType">
  <xs:restriction base="xs:string">
    <xs:enumeration value="XLS"/>
    <xs:enumeration value="CSV"/>
  </xs:restriction>
</xs:simpleType>

</xs:schema>

```



Important

Currently only the add element is supported. The others will soon be implemented so that iterative changes can be supported.

This example loads a single **.drl** file:

Example 3.8. Simple Change-Set XML

```

<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>

```

Take note of the file: prefix, as this signifies the protocol for the resource. The Change-Set supports all of the protocols provided by **java.net.URL**, such as `file` and `http`, as well as an additional version of `classpath`.



Important

Remember to always specify the type attribute for a resource, because it is not inferred from the filename extension.

Utilise the `ClassPath` resource loader in Java to specify the `class loader` to be used to locate the resource (this is not possible in XML.) The **class loader** to be used will, by default, be that which is employed by the `Knowledge Builder` (unless the Change-Set XML is loaded by the `ClassPath` resource. If so, the **class loader** specified for that resource will be used instead.)

Example 3.9. Loading the Change-Set XML

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

Any number of resources can be included in a change-set. Eventually, they will even support additional configuration information (though this use is currently restricted to decision tables only.) [Example 3.10, "Change-Set XML with Resource Configuration"](#) loads rules from both an HTTP uniform resource location and an Excel decision table found on the class-path.

Example 3.10. Change-Set XML with Resource Configuration

```
<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>
```

The Change-Set is especially useful when working with a **Knowledge Agent**, as it provides *change notification* functionality and automatically rebuilds the `Knowledge Base`. (These features are covered in more detail under the sub-heading "Deploying" in the section on the **Knowledge Agent**.)

One can also specify a directory. Do this in order to add all of the resources found within it. (The software expects that all of the resources will be of the same type.) If one uses the **Knowledge Agent**, it will continuously scan for changes to the resources. It will also rebuild the cached `Knowledge Base`.



Note

Change-sets can also be used in conjunction with the **Knowledge Agent**. Refer to [Section 3.2.6, "KnowledgeAgent"](#) for more information.

Example 3.11. Change-Set XML Code for Adding a Directory's Contents.

```
<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>
```

3.2. Deploying

3.2.1. The KnowledgePackage and Knowledge Definitions

A *KnowledgePackage* is a collection of *Knowledge Definitions*, which is simply another term for rules and processes. A **KnowledgePackage** is created by the **KnowledgeBuilder**, as described in [Section 3.1, “Building”](#). **KnowledgePackages** are self-contained and serializable. They form the current basic deployment unit.

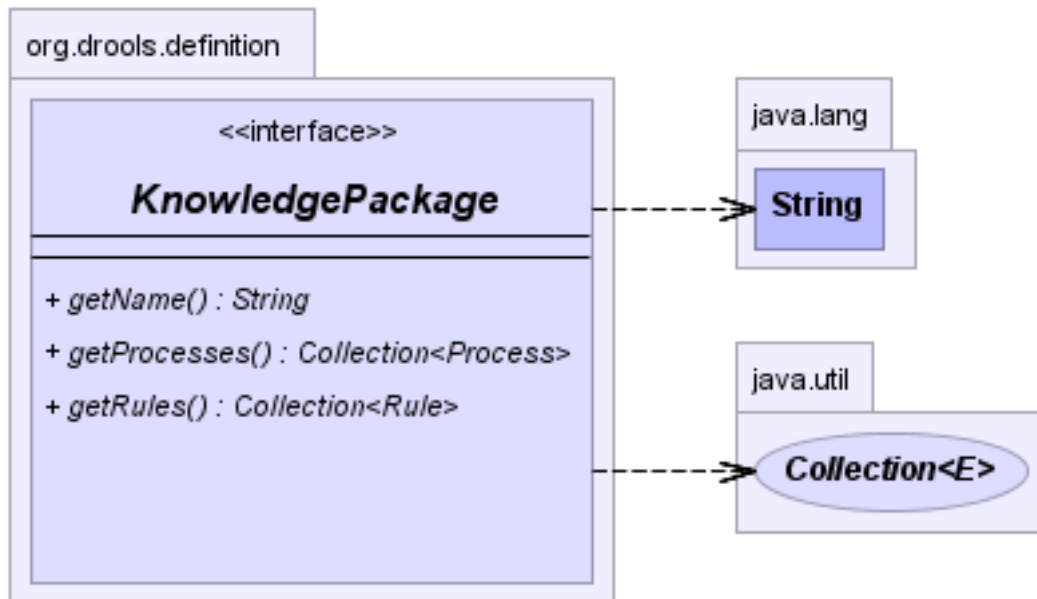


Figure 3.4. KnowledgePackage



Important

KnowledgePackages are added to the **Knowledge Base**. However, it is important to understand that a **KnowledgePackage** instance cannot be re-used once this has occurred. To add it to another **knowledge base**, try *serializing* it first and using the "cloned" result. This limitation will be removed in a future version of **JBoss Rules**.

3.2.2. Knowledge Bases

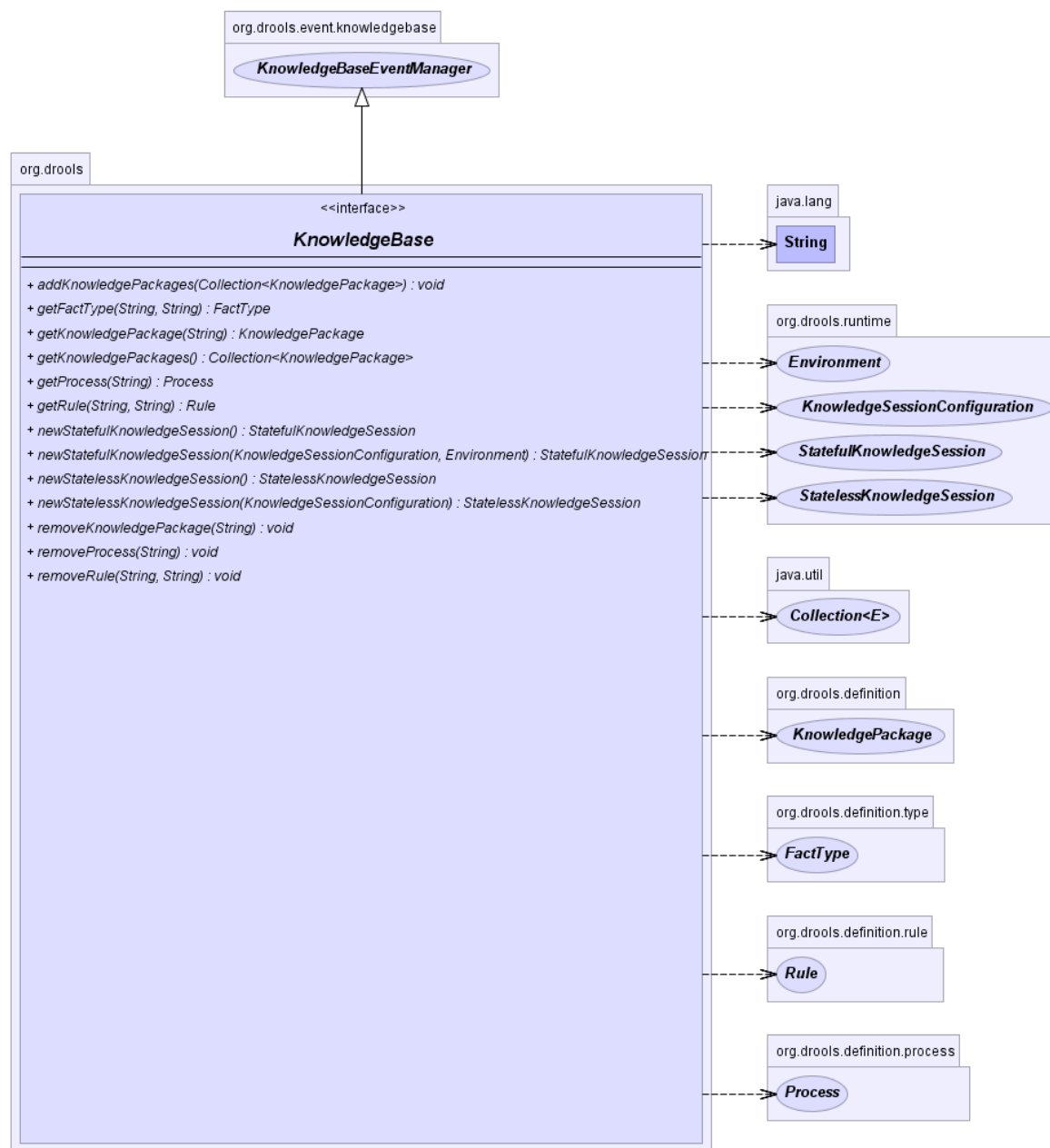


Figure 3.5. A Knowledge Base

A **knowledge base** is a repository that contains all of the application's *knowledge definitions*. It may contain rules, processes, functions and type models. The **knowledge base** itself does not contain "instance" data, (known as *facts*.) Instead, sessions are created from the **Knowledge Base** into which facts can be inserted and from which process instances can be commenced.



Important

Creating a **knowledge base** is a rather resource-intensive process, whereas creating a session is not. Therefore, Red Hat recommends caching **knowledge bases** where possible to facilitate the repeated creation of sessions.

A **knowledge base** object is also *serializable* so it may be preferable to build it and then store it. By so doing, one can treat it, rather than the `knowledge` packages, as the unit of deployment.

One way to create a **knowledge base** by using the `KnowledgeBuilderFactory` class:

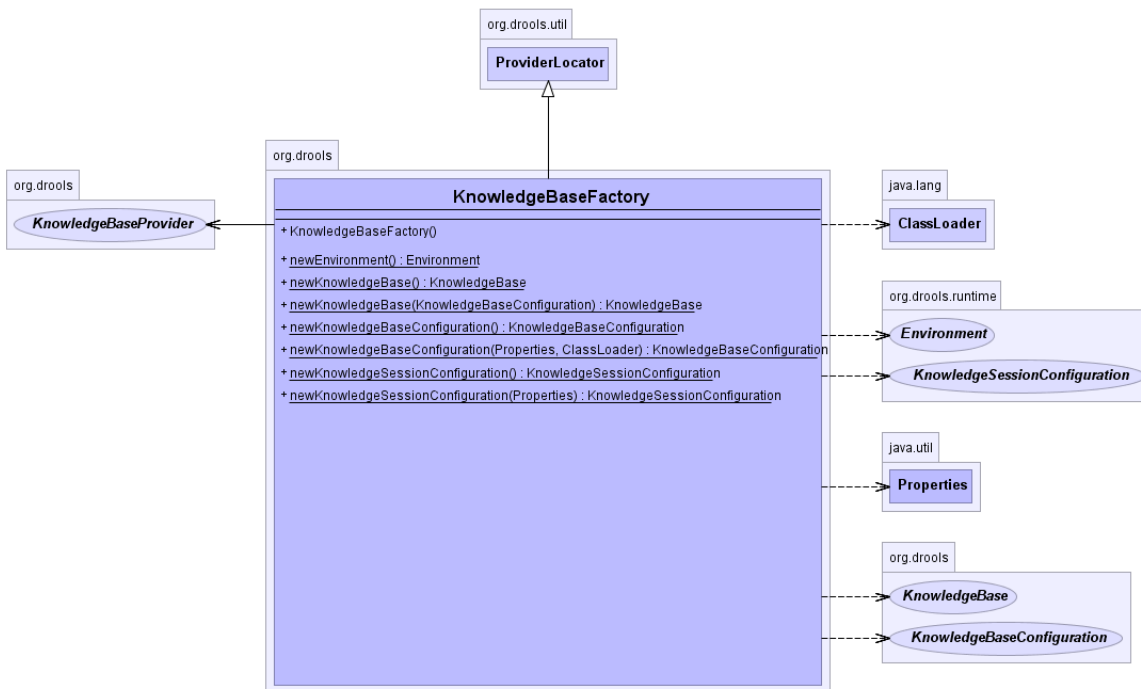


Figure 3.6. KnowledgeBuilderFactory

Another way to create one is by employing the default configuration:

Example 3.12. Creating a New Knowledge Base

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

If one wishes to use a customised class-loader in conjunction with the **Knowledge Builder** to resolve types that were not in the default loader, then set it on the **Knowledge Base**. (The technique for this is the same as that which applies to the **Knowledge Builder**.)

Example 3.13. Creating a New Knowledge Base with a Custom Class-Loader

```
KnowledgeBaseConfiguration kbaseConf =
    KnowledgeBuilderFactory.newKnowledgeBaseConfiguration( null, cl );
KnowledgeBase kbase = KnowledgeBuilderFactory.newKnowledgeBase( kbaseConf );
```

3.2.3. In-Process Building and Deployment

The simplest form of deployment is known as *in-process building*. In this case, the knowledge definitions are compiled and added to the **knowledge base** that is residing in the same Java Virtual Machine.



Important

Ensure that the **drools-core.jar** and **drools-compiler.jar** files are on the class-path when using this approach.

Example 3.14. Add Knowledge Packages to a Knowledge Base

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



Note

Understand that the `addKnowledgePackages(kpkgs)` method can be called on an iterative basis. Do so in order to add additional knowledge.

3.2.4. Building and Deployment as Separate Processes

Both the **Knowledge Base** and the **KnowledgePackage** are units of deployment. They can, therefore, be *serialized*. This means that one can assign one machine to undertake any necessary building that requires **drools-compiler.jar**, and have another machine reserved to deploy and execute everything. This second machine will only require **drools-core.jar**.

Although "serializing" is a standard Java practice, the examples below show one machine might write out the deployment unit and how another machine might read it in and use it.

Example 3.15. Writing the KnowledgePackage to an Output Stream

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

Example 3.16. Reading the KnowledgePackage from an Input Stream

```
ObjectInputStream in = new ObjectInputStream( new FileInputStream( fileName ) );
// The input stream might contain an individual
// package or a collection.
@SuppressWarnings( "unchecked" )
Collection<KnowledgePackage> kpkgs =
    ()in.readObject( Collection<KnowledgePackage> );
in.close();

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

The actual **knowledge base** itself is also serializable, so one may prefer to build and store it rather than the **knowledge packages**.



Note

Red Hat's server-side management system, Drools Guvnor , uses this deployment approach. After it has compiled and published serialized **knowledge packages** to a uniform resource location, it can use this address resource type to load them.

3.2.5. Stateful Knowledge Sessions and Knowledge Base Modifications

Stateful Knowledge Sessions are discussed in more detail in [Section 3.3.2, "StatefulKnowledgeSession"](#). The **Knowledge Base** creates and returns them. It also may, optionally, keep references to them. When the **Knowledge Base** is modified, these changes are applied to the data in the sessions. This is a weak, optional reference, controlled by a Boolean flag.

3.2.6. KnowledgeAgent

The **KnowledgeAgent** is a class that provides automatic loading, caching and re-loading of resources. It is configured via a properties files. The **KnowledgeAgent** can update or rebuild the **Knowledge Base**, as the resources it uses are changed. The factory's configuration determines the strategy that will be used (normally, it will typically be pull-based and use regular polling.)



Note

The capacity for push-based updates and rebuilds will be added in a future version.

The **KnowledgeAgent** continuously scans all of the added resources, using a default polling interval of sixty seconds. If the date of the last modification is updated, the cached **Knowledge Base** is automatically rebuilt using the new resources.

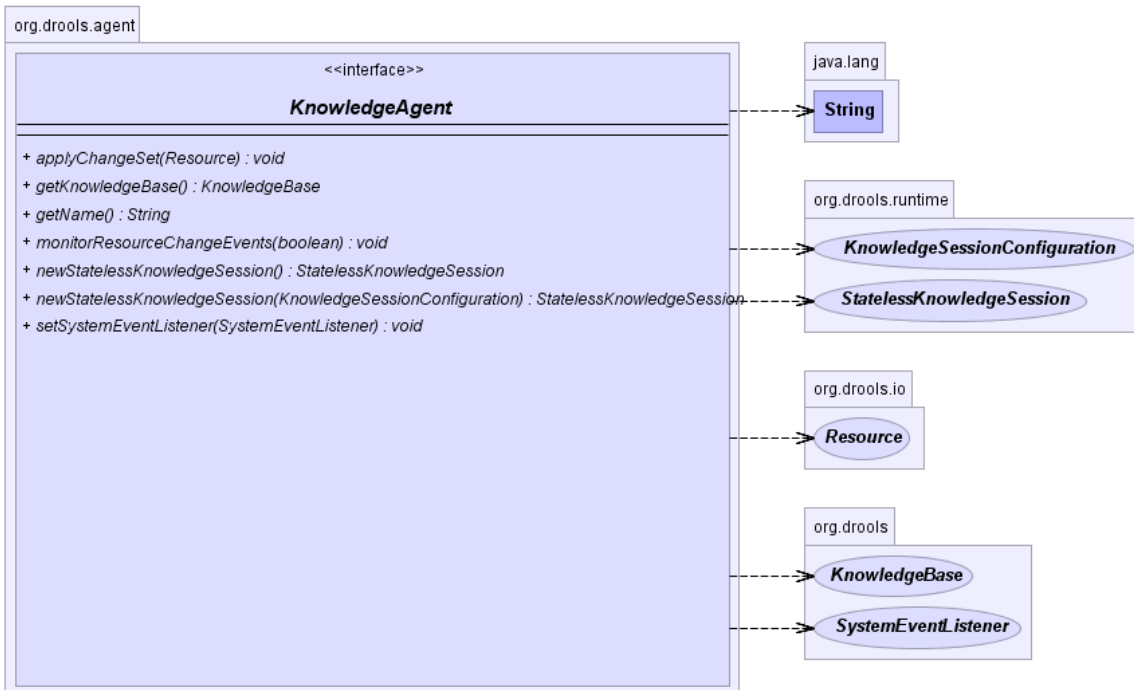


Figure 3.7. KnowledgeAgent

A **KnowledgeBuilderFactory** object is used to create the **Knowledge Builder**. The agent must specify a name because this will be needed by the log files. (This is so that the log entries can be associated against the correct agents.)

Example 3.17. Creating the KnowledgeAgent

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

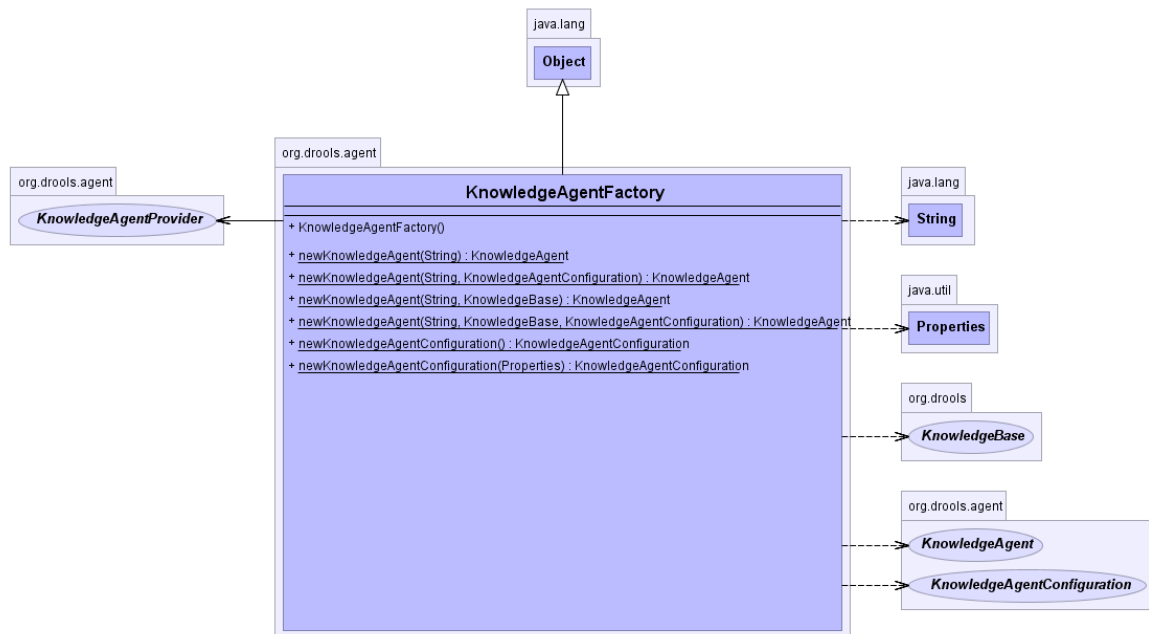


Figure 3.8. KnowledgeAgentFactory

The following example constructs an agent that will build a new **knowledge base** from the specified change-set.

Note

Refer to [Section 3.1.2, “ Building via Configurations and the Change-Set XML ”](#) for additional information about change-sets.

Note

The method can be called on an iterative basis. This enables one to add new resources over time.

The **KnowledgeAgent** polls the resources added from the **change set** every sixty seconds, (the default interval), to see if they are updated. Whenever changes are found, it will construct a new **Knowledge Base**. In addition, if a directory has been specified as the resource, its contents will be scanned.

Example 3.18. Writing the **KnowledgePackage** to an Output Stream

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

Resource scanning is switched off by default. It is a service, so it must be specifically started. The same is true of notifications. Activate both of these via the **ResourceFactory**.

Example 3.19. Starting the Scanning and Notification Services

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

Change the default resource scanning period via the **ResourceChangeScannerService** class. (An updated **ResourceChangeScannerConfiguration** object is passed to the service's `configure()` method, thereby allowing for the service to be reconfigured on demand.)

Example 3.20. Changing the Scanning Intervals

```
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 );
```

KnowledgeAgents can handle both empty and populated **Knowledge Bases**. If a populated **Knowledge Base** is provided, the **KnowledgeAgent** will run an *iterator* from within it and subscribe to each resource that it finds.



Warning

Whilst it is possible to make the **KnowledgeBuilder** build all of the resources in a directory, that information it will then lose that information. This means that those directories will not be continuously scanned. Only directories specified via the `applyChangeSet (Resource)` method are monitored.



Note

One of the advantages of using **Knowledge Base** as the starting point is that one can provide it with a **KnowledgeBaseConfiguration** class. When resource changes are detected and a new **Knowledge Base** is instantiated, it will use the **KnowledgeBaseConfiguration** class belonging to the previous **Knowledge Base** object.

Example 3.21. Using an Existing Knowledge Base

```

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

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

```

In the example above, the `getKnowledgeBase()` method returns the same **Knowledge Base** instance until resource changes are detected and a new **Knowledge Base** is built. When this happens, it is done with the **KnowledgeBaseConfiguration** that was provided to the previous **Knowledge Base**.

Example 3.22. Change-Set XML Which Adds the Contents of a Directory

```

<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>

```

**Note**

The `drools-compiler` dependency is not needed for the resource type entitled `PKG`, as the **KnowledgeAgent** is able to handle those with `drools-core` alone.

Use the **KnowledgeAgentConfiguration** to modify a **KnowledgeAgent**'s default behaviour. Do this to load the resources from a directory, whilst inhibiting the continuous scan of that directory for changes.

Example 3.23. Change the Scanning Behaviour

```

KnowledgeBase kbase = KnowledgeBaseFactory.newKnowledgeBase();

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

```

Previously, one was taught how the **JBoss Enterprise BRMS Platform** can build and publish serialized **Knowledge Packages** through a uniform resource location and also how the Change-Set XML can handle both URLs and packages. Taken together, these form an important deployment scenario for the **Knowledge Agent**.

3.3. Running

3.3.1. The Knowledge Base

The *KnowledgeBase* is a repository that contains all of the application's *knowledge definitions*. It may contain rules, processes, functions and type models. The **Knowledge Base** itself does not contain instance data, (known as *facts*.) Instead, sessions are created from the **KnowledgeBase** into which facts can be inserted and from where process instances may be started.



Note

Knowledge Base creation is a resource-intensive process, whereas session creation is not. Cache **Knowledge Bases** whenever possible to facilitate repeated session creation.

Example 3.24. Creating a New Knowledge Base

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

3.3.2. StatefulKnowledgeSession

The **StatefulKnowledgeSession** stores and executes the run-time data. It is created from the **KnowledgeBase**.

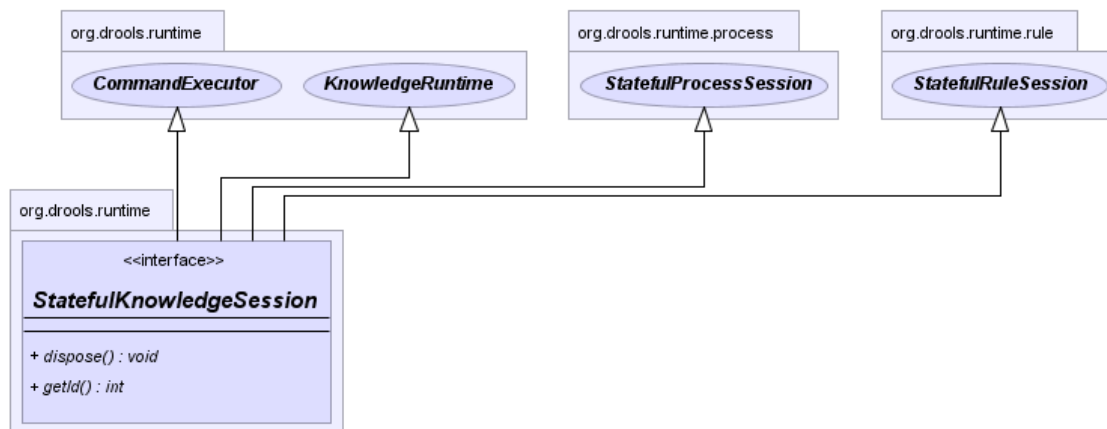


Figure 3.9. StatefulKnowledgeSession

Example 3.25. Create a StatefulKnowledgeSession from a KnowledgeBase

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

3.3.3. KnowledgeRuntime

3.3.3.1. WorkingMemoryEntryPoint

The **WorkingMemoryEntryPoint** provides the methods for inserting, updating and retrieving facts.



Note

The term, *entry point*, is related to the fact that there are multiple partitions in a working memory and one can choose into which of these the facts will be inserted. 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. While the focus is on the methods and interfaces related to rules, you'll notice that the `KnowledgeRuntime` inherits methods from both the `WorkingMemory` and the `ProcessRuntime`. This provides a unified API to work with processes and rules. When working with rules three interfaces form the `KnowledgeRuntime: WorkingMemoryEntryPoint`, `WorkingMemory`, and the `KnowledgeRuntime` itself.

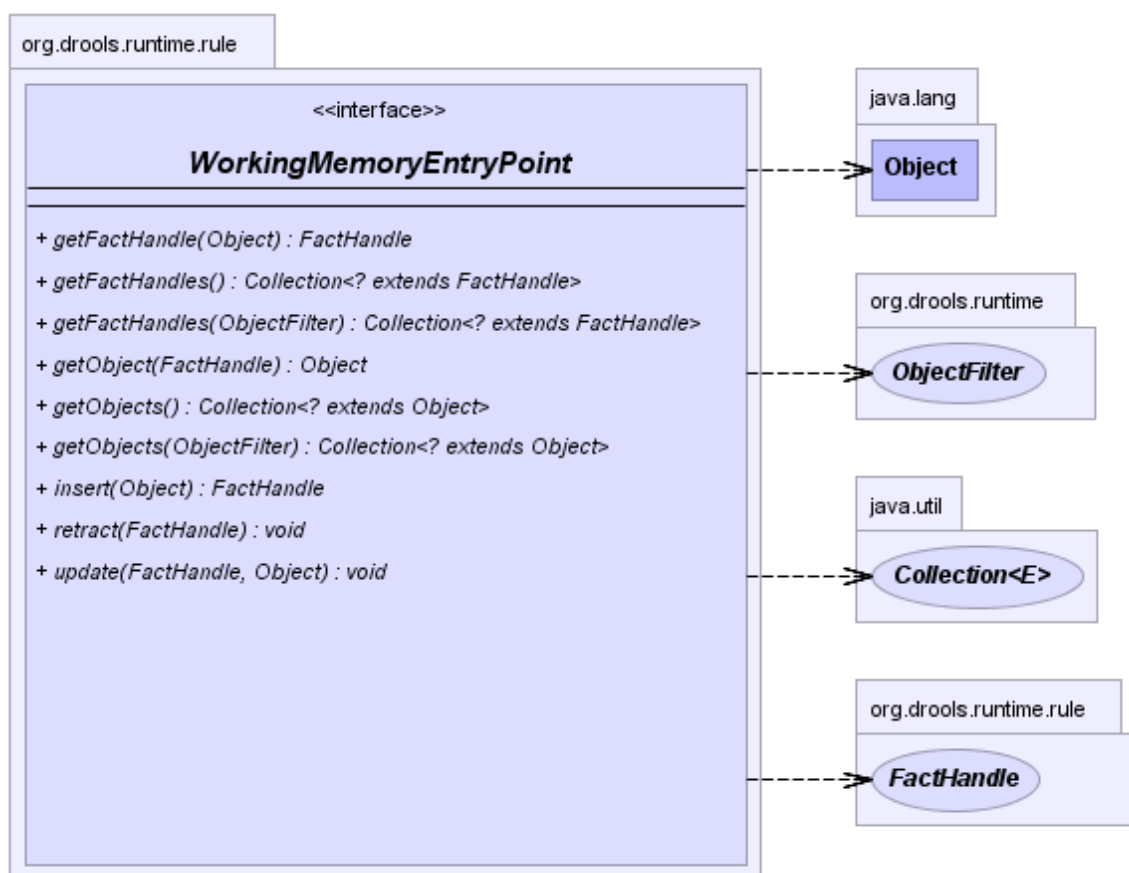


Figure 3.10. `WorkingMemoryEntryPoint`

3.3.3.1.1. Insertion

Insertion is the act of telling the `WorkingMemory` about a fact. (Here is an example: `ksession.insert(yourObject)`.) As they are inserted, the system examines each fact for matches against rules. All of the decisions about whether or not to fire a rule happen at the time of insertion. However, no rule is executed until `fireAllRules()` is called. Do this only after inserting all of the facts.



Note

In the past, users have sometimes erroneously held the belief that the condition evaluation occurs when `fireAllRules()` is called.



Note

The term *assert* or *assertion* is normally used in relation to expert systems to refer to facts that have been made available. However, due to "assert" being a keyword in most languages, Red Hat has decided to use the *insert* keyword to avoid clashes, so the two terms are often used interchangeably.

When an object is inserted it returns a *fact handle*. A `FactHandle` is the token used to represent the inserted object within the working memory. It is also used for interactions with the working memory when objects are modified or retracted.

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

A working memory can operate in either one of these two assertion modes: *equality* or *identity*. (Identity is the default one.)

- If *Identity* is used, the working memory utilises an **IdentityHashMap** to store all of the asserted objects. New instance assertions always result in the return of a new `FactHandle`. Repeated insertions of the same instance will simply return the original fact handle.
- If *Equality* is used, the working memory utilises a **HashMap** to store all of the asserted objects. New instance assertions will only return a new `FactHandle` if no equal objects have been asserted.

3.3.3.1.2. Retraction

The term *retraction* refers to the removal of a fact from the working memory. The fact will no longer be tracked or matched to rules. Furthermore, any rules that are activated and dependent on that fact will be cancelled. Retraction is achieved via utilisation of the `FactHandle` that was returned at the time of assertion.



Note

It is possible to create rules (using the **not** and **exist** keywords) that will fire when certain facts do not exist. In these cases, retracting a fact may cause the rule to activate.

```
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 it can reprocess them. When a fact which is identified as having been updated, it is automatically retracted from the working memory and inserted again.

If an modified object is unable to notify the working memory itself, use the update method to do so. The update method always takes the modified object as a secondary parameter. This allows one to specify new instances for *immutable objects*.



Note

The update method can only be used with objects for which *shadow proxies* have been turned on.



Important

The update method is only for use in conjunction with Java code. Within a rule, use the `modify` keyword as this provides calls to an object's "setter" methods.

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

3.3.3.2. Working Memory

The working memory provides access to the agenda, permits query executions and allows one to access named entry points.

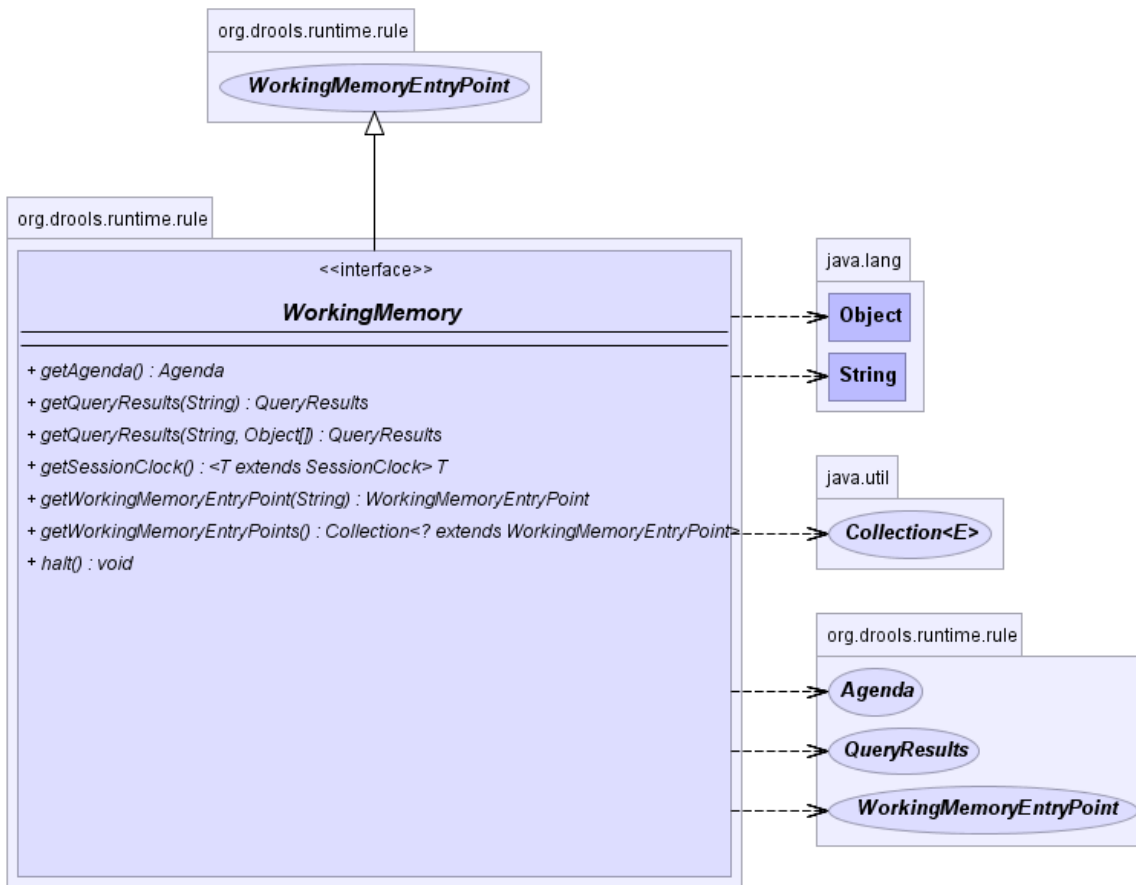


Figure 3.11. Working Memory

3.3.3.2.1. Query

Use *queries* to retrieve fact-sets. They are based on patterns as they are used in rules. These patterns may make use of optional parameters.

Define queries in the Knowledge Base, from which place they are called to return the matching results. Whilst iterating over the result collection, any bound identifier in the query can be accessed using the `get(String identifier)` method. Any `FactHandle` for that identifier can be retrieved using `getFactHandle(String identifier)`.

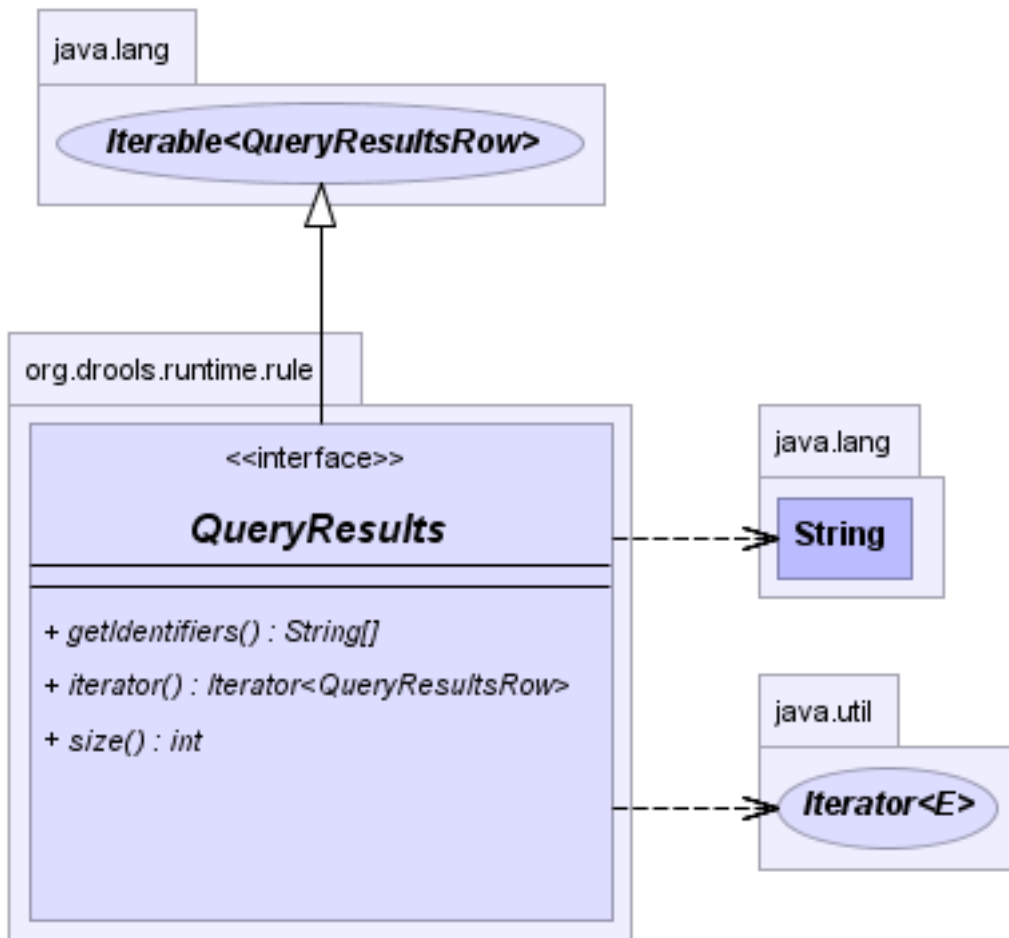


Figure 3.12. Query Results

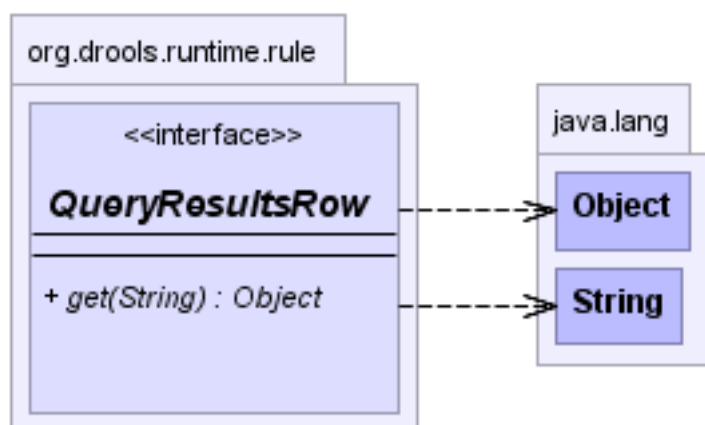


Figure 3.13. QueryResultsRow

Example 3.26. Simple Query Example

```

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

3.3.3.3. Live Queries

JBoss Enterprise BRMS 5.2 supports live queries.

Live Queries use an attached listener and remain open as a view and publish change events for the contents of this view. This makes it possible to execute a query with parameters and listen to changes in the resulting view.

Example 3.27. Implementing ViewChangedEventListener

```
final List updated = new ArrayList();
final List removed = new ArrayList();
final List added = new ArrayList();

ViewChangedEventListener listener = new ViewChangedEventListener() {
    public void rowUpdated(Row row) {
        updated.add( row.get( "$price" ) );
    }

    public void rowRemoved(Row row) {
        removed.add( row.get( "$price" ) );
    }

    public void rowAdded(Row row) {
        added.add( row.get( "$price" ) );
    }
};

// Open the LiveQuery
LiveQuery query = ksession.openLiveQuery( "cheeses",
                                         new Object[] { "cheddar", "stilton" },
                                         listener );

...
...
query.dispose() // make sure you call dispose when you want the query to close
```

3.3.3.4. KnowledgeRuntime

The **KnowledgeRuntime** provides further methods applicable to both rules and processes. Some examples are those for setting globals and registering **ExitPoints**.

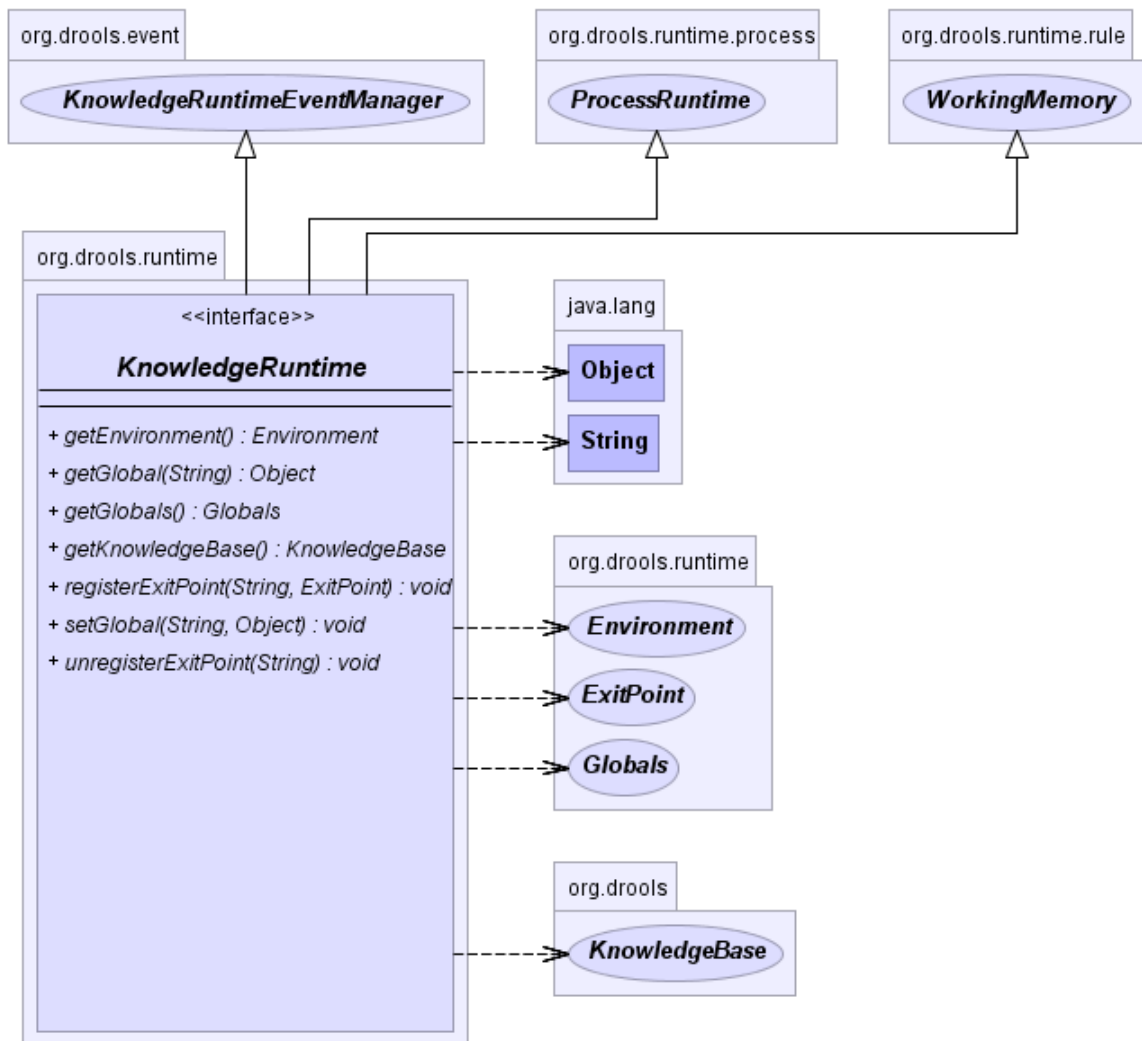


Figure 3.14. KnowledgeRuntime

3.3.3.4.1. Globals

Globals are named objects that can be passed to the rule engine. There is no need to insert them. Most often they are used for static information, or for services that are used in the right-hand side of a rule, or perhaps as a means to return objects from the rule engine.

To use a global on the left-hand side of a rule, follow these steps:

1. Make sure that it is immutable.
2. Declare it in a **rules** file before it is set on the session:

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

3. With the Knowledge Base now aware of the global identifier and its type, call `ksession.setGlobal` for any session.



Warning

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);
```



Warning

If a rule evaluates on a global before it is set, a `NullPointerException` exception will be thrown.

3.3.3.5. StatefulRuleSession

The `StatefulRuleSession` is inherited by the `StatefulKnowledgeSession`. It provides the rule-related methods that are applicable outside the engine.

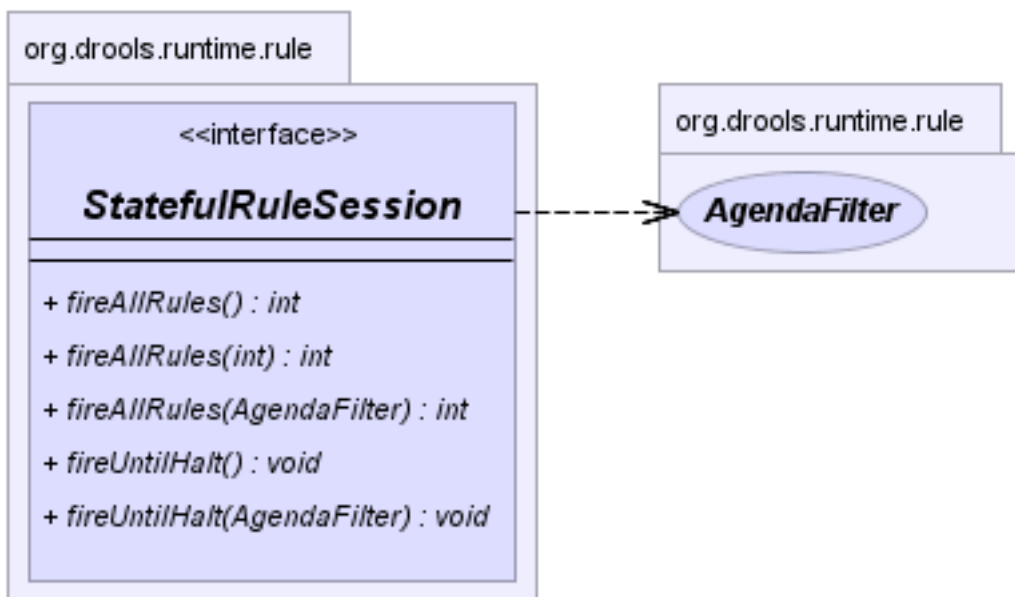


Figure 3.15. StatefulRuleSession

3.3.3.5.1. Agenda Filters

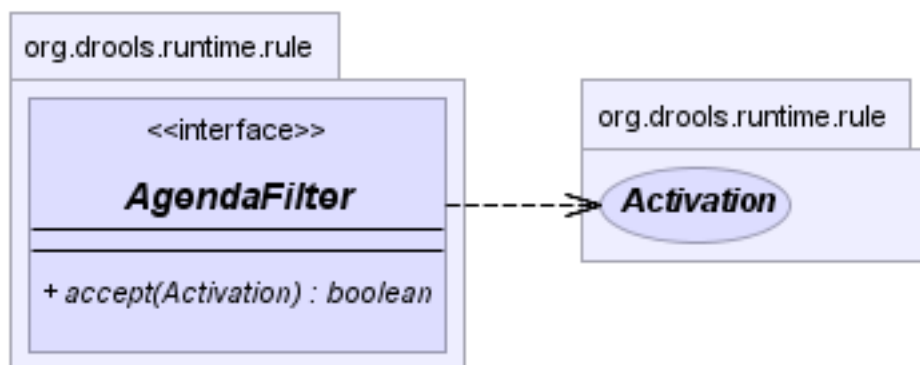


Figure 3.16. AgendaFilters

Agenda filters are optional implementations of the `filter` interface. Use them to allow or deny an activation the right to fire. (That which can be filtered is entirely dependent upon the implementation.)



Note

Earlier versions of **JBoss Rules** supplied several filters which are not provided in version 5.0. They are simple to implement. Refer to the **JBoss Rules 4** code base in order to do so.

To use a filter specify it when calling `fireAllRules()`. The following example permits only rules those ending in the string **Test** to fire. It filters out all others:

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

3.3.4. Agenda

The *Agenda* is a Rete feature. When actions are performed on the `working memory`, rules may become fully matched and, therefore, eligible for execution. A single `working memory` action can result in multiple rules being made eligible. When a rule is fully matched an activation is created. This references both the rule and the matched facts, and is placed onto the Agenda. The Agenda then determines the order of these Activations via a conflict resolution strategy.

The engine then cycles repeatedly through two phases:

1. The first is termed the *Working Memory Actions Phase*. Most of the work takes place at this time, either in the *consequence* (the right-hand side) 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 second phase.
2. This is termed the *Agenda Evaluation Phase*. At this time, the system searches for a rule to fire. If none is found it exits. Otherwise, it fires the rule it has found and then switches itself back to Working Memory Actions Phase.

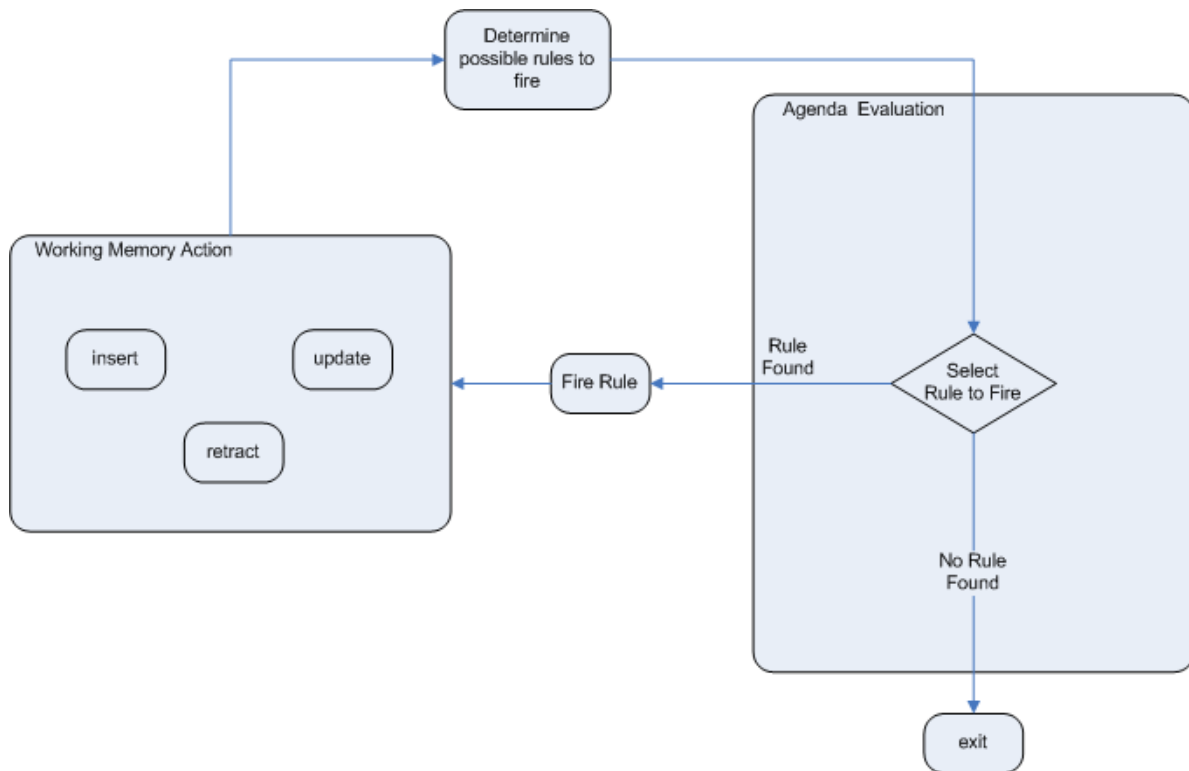


Figure 3.17. Two Phase Execution

3. The process repeats over and over until the agenda is cleared, at which time control is returned to the calling application.



Note

No rules are fired whilst working memory actions are taking place.

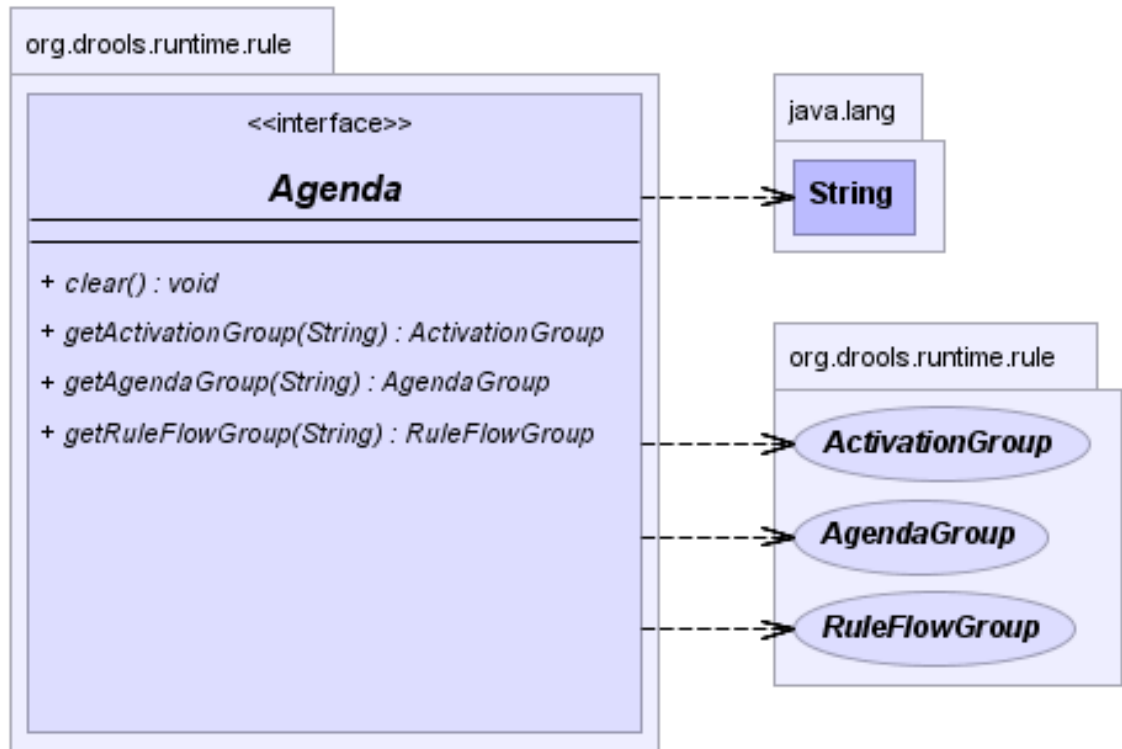


Figure 3.18. Agenda

3.3.4.1. Conflict Resolution

When there are multiple rules on the agenda a conflict resolution strategy is required. As the firing of a rule may have an impact upon the working memory, the rule engine needs to know in which order the rules are to be executed. (For example, firing **ruleA** may cause **ruleB** to be removed from the agenda.)

JBoss Rules employs two conflict resolution strategies. These are:

- Saliency
- LIFO (Last In, First Out)

Use the *saliency* strategy, to can specify that a certain rule has a higher priority (by giving it a higher number) than other rules. In this case, the rule with the higher saliency will be given preferential treatment.

The LIFO strategy prioritises based on the assigned working memory's action counter value, with each rule created during the same action receiving the same value. (If a set of firings has the same priority value, the execution order will be arbitrary.)



Important

Although sometimes unavoidable, try always to avoid writing rules that are reliant upon being fired in a specific order to work correctly. Do not think of rules in terms of being 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 application programming interface is no longer exposed.

3.3.4.2. AgendaGroup

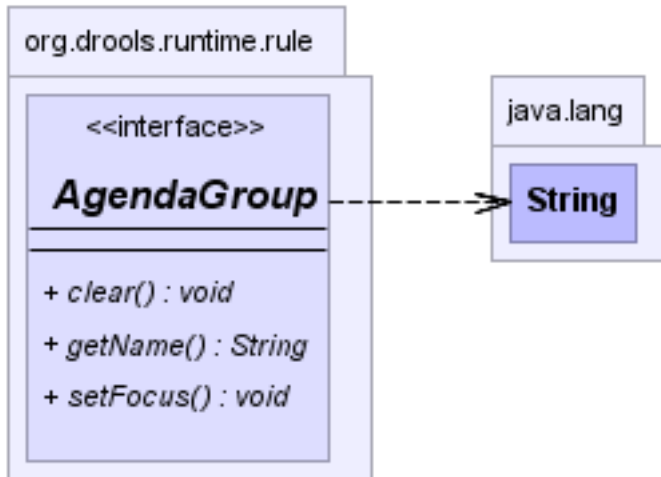


Figure 3.19. AgendaGroup

Use *agenda groups*, (known as "modules" in CLIPS terminology), to partition activations on the agenda. At any time, only one group can have "focus", and only the activations belonging to that group will be able to take effect.



Note

Agenda groups are most commonly used to define one or more subsets of rules that apply to specific circumstances, (such as phases of processing), and to control as to when these sets of rules can be applied.

Set focus either from within a rule or via the **JBoss Rules** application programming interface. (Another option is to set rules to use *auto-focus*. By so doing, the agenda group will become focused when it is matched.)

Each time `setFocus()` is called, it pushes an agenda group onto a *stack*. When the focus group is empty, it is removed from the stack and the next focus group (now the topmost one) is permitted to evaluate.



Note

An agenda group can appear in multiple locations on the stack.

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

The default agenda group is called **MAIN**. It is the first group on the stack and, hence, initially has the focus. Any rule without an agenda group is automatically placed in this group.

3.3.4.3. Activation Group

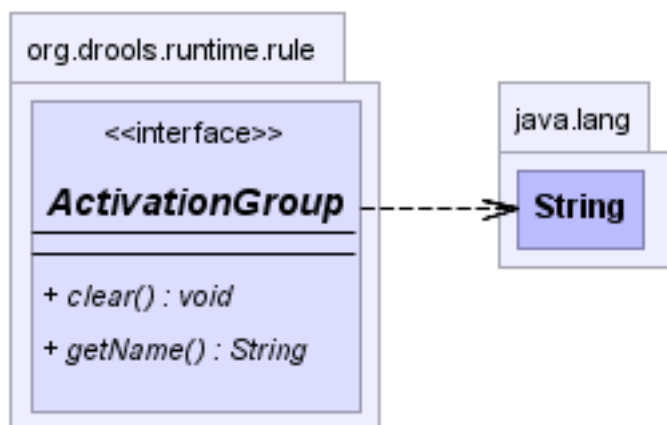


Figure 3.20. `ActivationGroup`

An activation group is set of rules bound together by the activation-group rule attribute. In this group only one rule can fire. After that rule has fired, all of the other rules are cancelled.



Note

Call the `clear()` method at any time, to cancel all of the activations before any have had a chance to fire.

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

3.3.5. Event Model

The **event** package notifies one of rule engine events. Use it to separate logging and auditing activities from the main part of the application and from the rules.

The `KnowledgeRuntimeEventManager` interface is implemented by the **KnowledgeRuntime** class which provides two interfaces, `WorkingMemoryEventManager` and `ProcessEventManger`.



Note

This book only covers the `WorkingMemoryEventManager`.

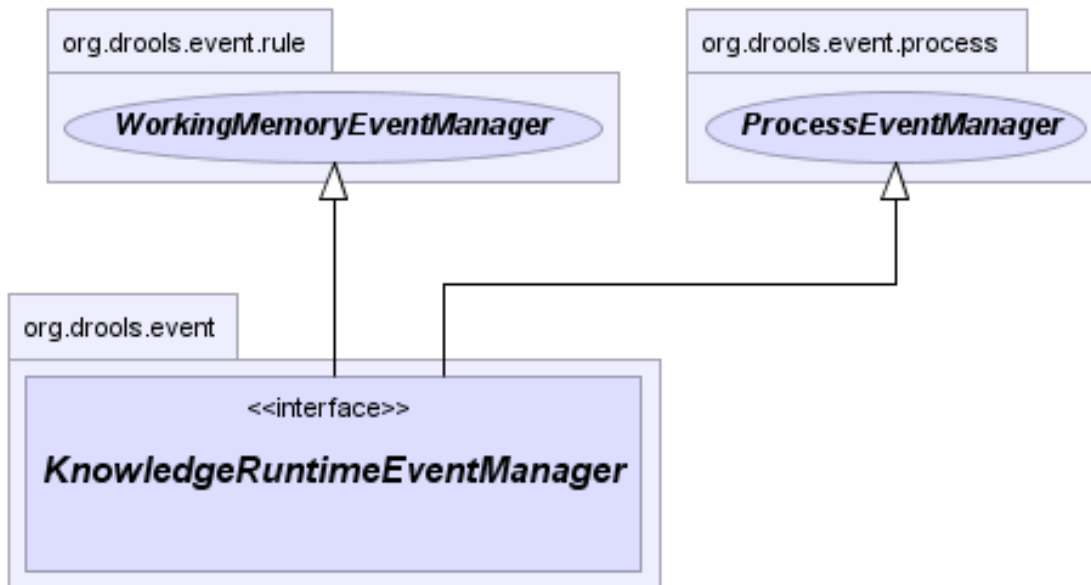


Figure 3.21. KnowledgeRuntimeEventManager

Use the WorkingMemoryEventManager to add and remove listeners. Adding a listener enables one to "listen" to events affecting the working memory and the agenda.

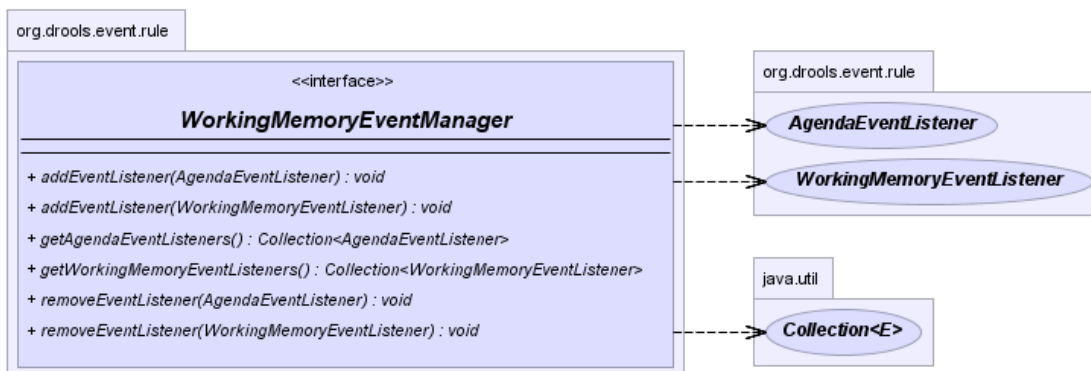


Figure 3.22. WorkingMemoryEventManager

The following code shows how to declare a simple agenda listener and attached it to a session. It prints activations after they have fired.

Example 3.28. Adding an AgendaEventListener

```

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

```

JBoss Rules also provides two classes called **DebugWorkingMemoryEventListener** and **DebugAgendaEventListener** which implement each method with a debug print statement. To print every working memory event, add one of these listeners.

Example 3.29. Creating a new KnowledgeBuilder

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

Use the KnowledgeRuntimeEvent interface in order to retrieve the KnowledgeRuntime from which the event originated.



Figure 3.23. KnowledgeRuntimeEvent

These are the supported events:

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

3.3.6. KnowledgeRuntimeLogger

The KnowledgeRuntimeLogger uses JBoss Rules's event systems to create an audit log each time an application is executed. Inspect this log with a tools such as the JBoss Rules IDE's Audit Viewer.

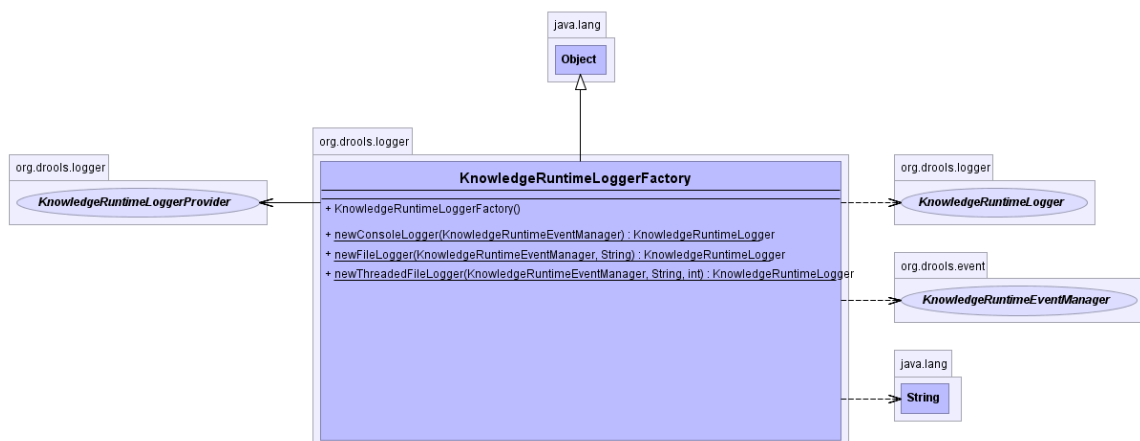


Figure 3.24. KnowledgeRuntimeLoggerFactory

Example 3.30. FileLogger

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

Use the `newFileLogger()` method to automatically append the file extension, `.log`, to any file.

3.3.7. StatelessKnowledgeSession

The `StatelessKnowledgeSession` wraps the `StatefulKnowledgeSession`. It is used in relation to decision service-type scenarios. Its presence mitigates the need to call `dispose()`.

One cannot perform iterative insertions or call the `fireAllRules()` method from Java code when using stateless sessions. The `execute()` method instantiates a `StatefulKnowledgeSession` internally, adds all of the user data and executes user commands. It then calls the `fireAllRules()` and `dispose()` methods.

The usual way to work with this class is via the `BatchExecution` command (as supported by the `CommandExecutor` interface.) However, two *convenience methods* have also been provided. Use these when only simple object insertion is required. (The `CommandExecutor` and `BatchExecution` are discussed in detail in their own sections.)

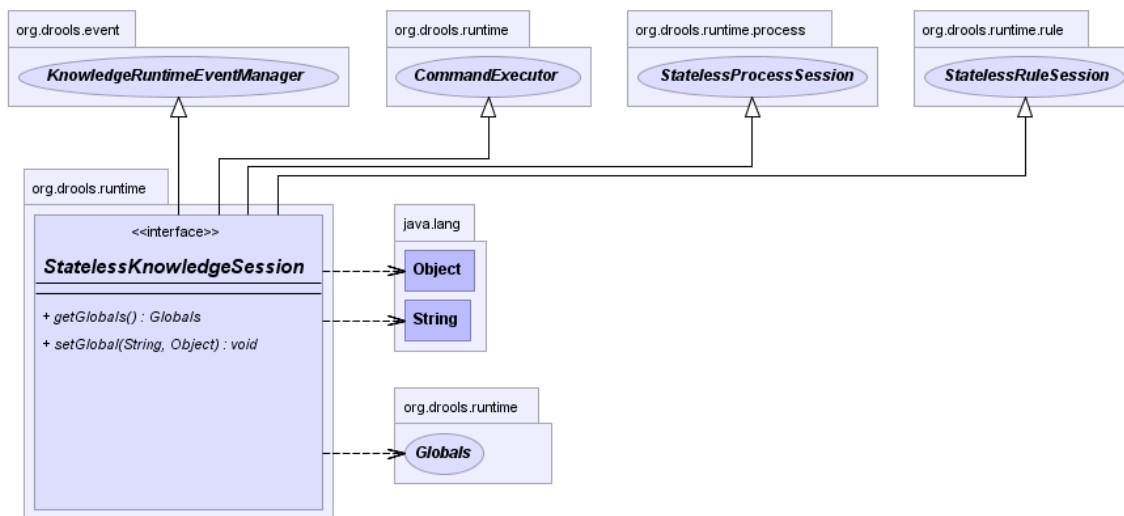


Figure 3.25. StatelessKnowledgeSession

This example shows a `stateless` session executing using the Convenience API to execute a given collection of Java objects. It iterates the collection, inserting each element in turn.

Example 3.31. Simple StatelessKnowledgeSession execution with a Collection

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



```

kbase.addKnowledgePackages( kbuilder.getKnowledgePackages() );
StatelessKnowledgeSession ksession = kbase.newStatelessKnowledgeSession();
ksession.execute( collection );
}

```

To do the same thing as a single command, use this code:

Example 3.32. Simple StatelessKnowledgeSession execution with InsertElements Command

```

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

```

To insert the collection itself without iterating it or inserting the elements, use `CommandFactory.newInsert(collection)`.

The **CommandFactory** contains details of the supported commands. To marshal any of them use **XStream** and the **BatchExecutionHelper**. Also use **BatchExecutionHelper** to learn details of the XML format being utilised. Use JBoss Rules Pipeline to automatically marshal the **BatchExecution** and **ExecutionResults**.

The **StatelessKnowledgeSession** allows one to scope globals in a number of ways. The first of these is the non-command way. Commands are scoped to a specific execution call. (Globals can be resolved in three ways.)

- The **StatelessKnowledgeSession's** `getGlobals()` method returns a `Globals` instance. As its name implies, this provides access to the session's globals. These are shared for *all* execution calls.



Warning

Exercise caution when handling *mutable globals* because execution calls can be run simultaneously in different threads.

Example 3.33. Session-Scoped Global

```

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

```

- Another way to perform global resolution is to use a delegate. Assigning a value to a global (with `setGlobal(String, Object)`) results in the value being stored in an internal collection, the purpose of which is to map identifiers to values. These identifiers will have priority over any delegate supplied: only if an identifier cannot be found will the delegate global (if, indeed, there is any) be used.
- The third way of resolving globals is to have *execution-scoped globals*. In this case, a command to set a global is passed to the `CommandExecutor`.

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

Example 3.34. Out identifiers

```
// Set up a list of commands
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" );

// Execute the list
ExecutionResults results =
    ksession.execute( CommandFactory.newBatchExecution( cmds ) );

// Retrieve the ArrayList
results.getValue( "list1" );
// Retrieve the inserted Person fact
results.getValue( "person" );
// Retrieve the query as a QueryResults instance.
results.getValue( "Get People" );
```

3.3.7.1. Sequential Mode

Rete provides a a stateful session into which objects can be asserted over time, and to which rules can also be added and removed. However, with a stateless session, after the initial data set has been provided, no more data can be asserted or modified and rules cannot be added and removed. In this case, it will not be necessary to re-evaluate rules, and the engine can be operated in a simplified way. Follow these steps:

1. Order the rules by salience and position in the rule-set (by setting a sequence attribute on the rule terminal node).
2. Create an array with one element for each possible rule activation. The position of the elements will indicate the firing order.
3. Turn off all of the *node memories*, except for the right-input *object memory*.
4. Disconnect the Left Input Adapter Node propagation and let a command object refer to the object and the nodes. Added this command object to a list in the working memory for later execution.
5. Assert all of the objects. When this has occurred and thus right-input node memories are populated, check the command list and execute each item in turn.
6. Place all resulting activations in the array, basing their order upon the sequence number determined for the rule. Record those elements populated first and last in order to reduce the iteration range.
7. Iterate the array of activations, thereby executing each populated element in turn.
8. If the maximum number of allowed rule executions exists, exit the network evaluations early in order to fire all of the rules in the array.

**Note**

The **LeftInputAdapterNode** no longer creates a tuple, adds the object or propagates the tuple. Rather, a command object is created and added to a list in the working memory. This object contains references to both the **LeftInputAdapterNode** and the propagated object. This stops any left-input propagations from occurring at insertion time which means that no right-input propagation will ever attempt a join with the left-inputs (thereby removing the need for left-input memory).

Nearly every node's memory is turned off, including the left-input tuple memory but excluding the right-input object memory, meaning that the latter is the only node which remembers an insertion propagation.

Once all of the assertions have concluded and, as a result, all of the right-input memories have been populated, iterate the list of **LeftInputAdapterNode** command objects by calling each in turn. They will be passed down the network and attempt to join with the right-input objects, but will not be remembered in the left input because there will be no further object assertions or propagations into the right-input memory.

**Note**

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 **RuleTerminalNode**'s sequence number indicates the element within the array upon which to place the activation.

Once every command object has been processed, iterate the array by checking each element in turn, and firing the activations (if they exist.)

**Important**

To improve performance, remember the first and the last populated cells in the array. The network is constructed, with each **RuleTerminalNode** being given a sequence number. This number is based on a salience number and the order in which it has been added to the network.

The right-input node memories are normally hash maps because this facilitates rapid object retraction. However, in this case, there will be no object retractions, so use a list as the object values are not indexed.

**Important**

For large numbers of indexed objects, hash maps provide a performance increase but if an object type has only a few instances, indexing will not be advantageous, so a list can be used.

Sequential mode can only be used with a Stateless Session and is turned off by default. To turn it on, either call `RuleBaseConfiguration.setSequential(true)`, or set the Rule base Configuration's `drools.sequential` property to **true**.



Note

Call `setSequentialAgenda` with `SequentialAgenda.DYNAMIC` to make the sequential mode fall back to a dynamic agenda. One may also set the `drools.sequential.agenda` property to `sequential` or `dynamic`.

3.3.8. Commands and the CommandExecutor

JBoss Rules makes use of stateful and stateless sessions. Stateful sessions use the standard working memory with which one can work *iteratively* over time. A stateless session is a one-off execution of a working memory with a provided data-set. It may return some results, and the session is disposed at the end, prohibiting further iterative interactions. Think of stateless sessions as a way in which to treat a rule engine as a function call with optional return results.

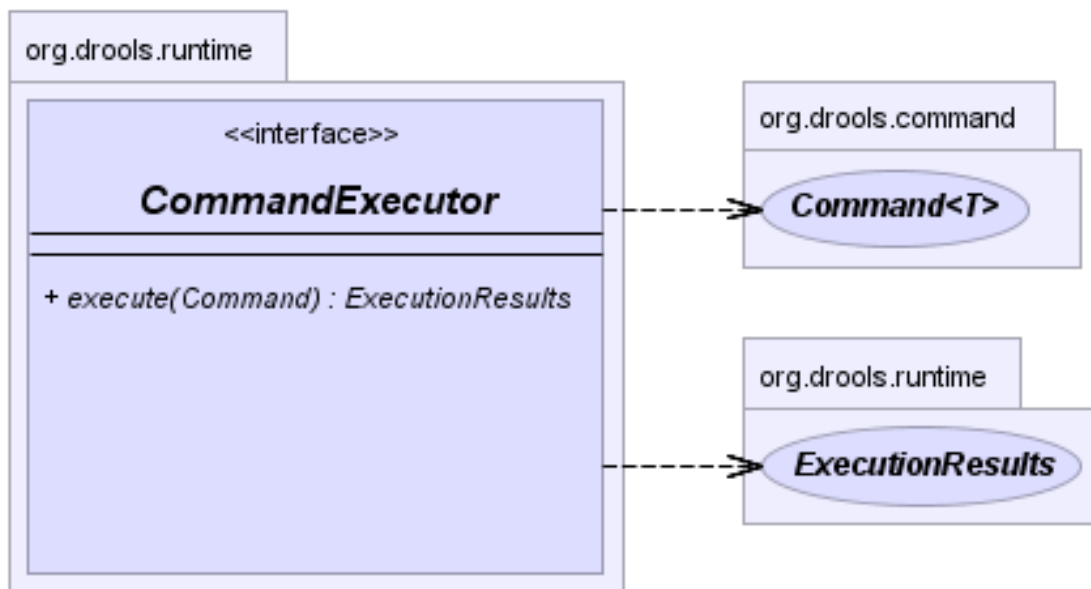


Figure 3.26. CommandExecutor

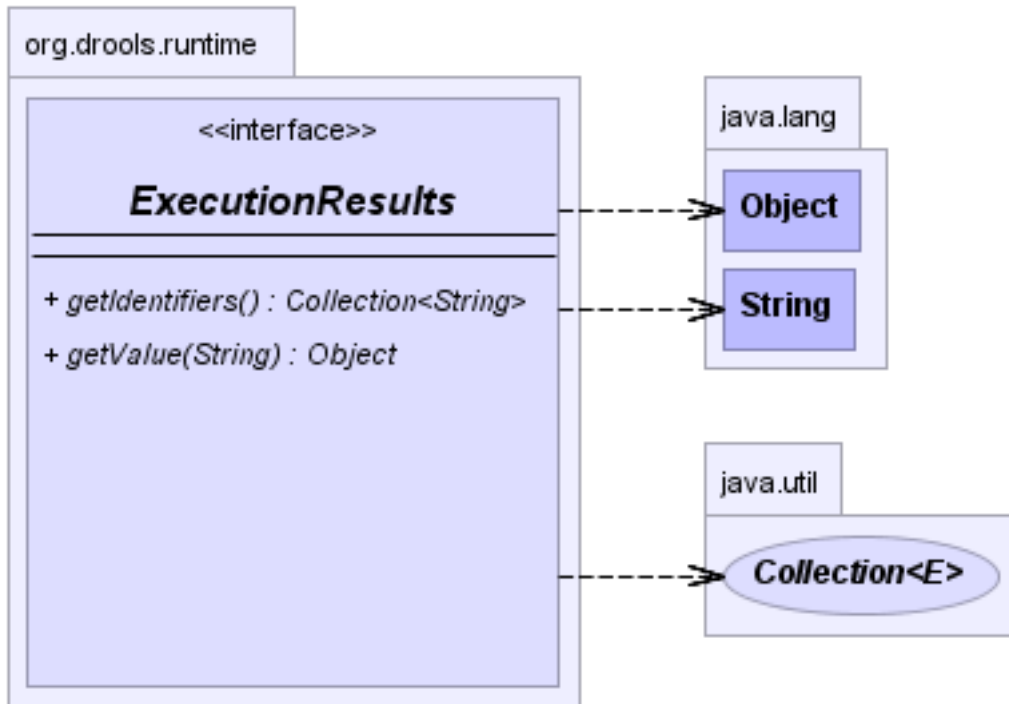


Figure 3.27. ExecutionResults

The **CommandFactory** allows one to execute commands on stateful and stateless sessions, (the only difference being that the stateless knowledge session executes `fireAllRules()` at the end before it is disposed.) These commands are currently supported:

- | | |
|----------------|----------------|
| FireAllRules | GetGlobal |
| SetGlobal | InsertObject |
| InsertElements | Query |
| StartProcess | BatchExecution |

As its name implies, `InsertObject` inserts a single object, with an optional out identifier. **InsertElements** runs through an iterable object, inserting each of the elements. As a result, one is no longer limited to just inserting objects into a stateless knowledge session, but can now start processes or execute queries and do this in any order.

Example 3.35. Insert Command

```

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

The `execute` method always returns an **ExecutionResults** instance. This allows one access to any command result if an out identifier, such as the "stilton_id" above, has been specified.

Example 3.36. InsertElements Command

```

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

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



Important

This method only allows for a single command. **BatchExecution** is a composite command that takes a list of instructions and iterates and execute each of these in turn. This means one can insert some objects, start a process, call **fireAllRules** and execute a query all in a single `execute(...)` call, making it much more powerful.

The stateless knowledge session executes `fireAllRules()` method automatically as it finishes processing. However, the **FireAllRules** command is also allowed, and using it will disable the automatic execution at the end - it is a manual override.

Commands support out identifiers. Any command upon which one will be set will add its results to the **ExecutionResults** instance that is returned. Here is an example that depicts the way in which it works:

Example 3.37. BatchExecution Command

```
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 ) );  
Cheese stilton = ( Cheese ) bresults.getValue( "stilton" );  
QueryResults qresults = ( QueryResults ) bresults.getValue( "cheeses" );
```

In this example, multiple commands are executed, two of which populate the **ExecutionResults**. The query command uses the same identifier as the query name by default, but it can also be mapped to a different identifier.

A customised XStream marshaller can be used in conjunction with the **JBoss Rules Pipeline** to provide XML scripting, making it ideal for services. Here are two simple XML samples, one for the **BatchExecution** and the other for the **ExecutionResults**.

Example 3.38. Simple BatchExecution XML

```
<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 3.39. Simple ExecutionResults XML

```
<execution-results>
```

```

<result identifier='outStilton'>
  <org.drools.Cheese>
    <type>stilton</type>
    <oldPrice>25</oldPrice>
    <price>30</price>
  </org.drools.Cheese>
</result>
</execution-results>

```

The pipeline allows one to use a series of stage objects. Combine these to more easily move data into and out of sessions.

There is a stage that implements the `CommandExecutor` interface. Use this to make the pipeline script either a stateful, or a stateless, session. Configure it in this way:

Example 3.40. Pipeline for CommandExecutor

```

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 );

```

BatchExecutionHelper is used to provide a specially-configured `XStream` with custom converters for command objects and the new **BatchExecutor** stage.

To use the pipeline, provide an implementation of the **ResultHandler**. This called when the pipeline executes the **ExecuteResultHandler** stage.

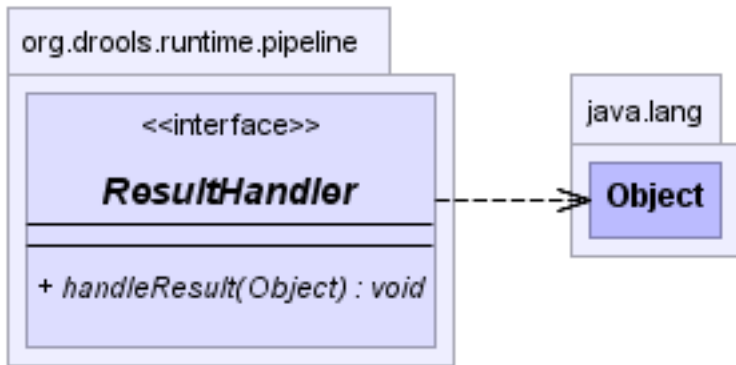


Figure 3.28. Pipeline ResultHandler

Example 3.41. Simple 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 3.42. Using a Pipeline

```

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

Now, use the **BatchExecution** created earlier to insert some objects and execute a query. The XML representation to be used with the pipeline example is shown below. Parameters have been added to the query:

Example 3.43. BatchExecution Marshalled to XML

```

<batch-execution>
  <insert out-identifier="stilton">
    <org.drools.Cheese>
      <type>stilton</type>
      <price>1</price>
      <oldPrice>0</oldPrice>
    </org.drools.Cheese>
  </insert>
  <query out-identifier='cheeses2' name='cheesesWithParams'>
    <string>stilton</string>
    <string>cheddar</string>
  </query>
</batch-execution>
  
```

The **CommandExecutor** returns the **ExecutionResults**, and this is handled by the pipeline code snippet as well.

Here is the similar output produced by the `<batch-execution>` XML sample:

Example 3.44. ExecutionResults Marshalled to XML

```
<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>
```

The **BatchExecutionHelper** provides a pre-configured `XStream` instance. Use it to support the marshalling of batch executions, (for which the resulting XML can be used as a message format, as shown above.) Only commands supported via the `Command Factory` have pre-configured converters. One can add other converters for user objects. (This is very useful when scripting for stateless or stateful knowledge sessions, especially when services are involved.)

No XML schema currently exists to support validation. The basic format is outlined here, and the `drools-transformer-xstream` module has a unit test called `XStreamBatchExecutionTest`. The root element is named `<batch-execution>` and it can contain any number of command elements:

Example 3.45. Root XML Element

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

This contains a list of elements that represent commands. The commands supported are limited to those provided by the `Command Factory`. 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`.

Example 3.46. Insert

```
<batch-execution>
  <insert>
```

```
...<!-- any user object -->
</insert>
</batch-execution>
```

The `insert` element features an attribute called `out-identifier`. This demands that the inserted object be returned as part of the result payload.

Example 3.47. Insert with Out Identifier Command

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

It is also possible to insert a collection of objects using the `<insert-elements>` element. This command does not support an `out-identifier`. (The `org.domain.UserClass` is just an illustrative user object that `XStream` can serialize.)

Example 3.48. Insert Elements command

```
<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>
```

As its name implies, the `<set-global>` element is used to set a global for the session:

Example 3.49. Insert Elements Command

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

`<set-global>` also supports two other optional attributes, `out` and `out-identifier`. A true value for the Boolean `out` will add the global to the `<batch-execution-results>` payload, using the name from the `identifier` attribute. `out-identifier` works like `out` but, additionally, allows one to override the identifier used in the `<batch-execution-results>` payload.

Example 3.50. Set Global Command

```

<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>

```

There is also a `<get-global>` element, without contents, with just an `out-identifier` attribute. (There is no need for an `out` attribute because retrieving the value is the sole purpose of a `<get-global>` element.)

Example 3.51. Get Global Command

```

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

```

The `out` attribute can only be used to return specific instances as a result payload. A different approach is needed to run actual queries. Fortunately, queries both with and without parameters 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.

Example 3.52. Query Command

```

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

```

**Note**

The `<start-process>` command accepts optional parameters.

Example 3.53. Start Process Command

```

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

```

```
</startProcess>
</batch-execution
```

Example 3.54. Signal Event Command

```
<signal-event process-instance-id='1' event-type='MyEvent'>
  <string>MyValue</string>
</signal-event>
```

Example 3.55. Complete Work Item Command

```
<complete-work-item id='" + workItem.getId() + "' >
  <result identifier='Result'>
    <string>SomeOtherString</string>
  </result>
</complete-work-item>
```

Example 3.56. Abort Work Item Command

```
<abort-work-item id='21' />
```



Note

More commands will be added over time.

3.3.9. Marshaling

Use the **MarshallerFactory** to marshal and unmarshal stateful knowledge sessions.

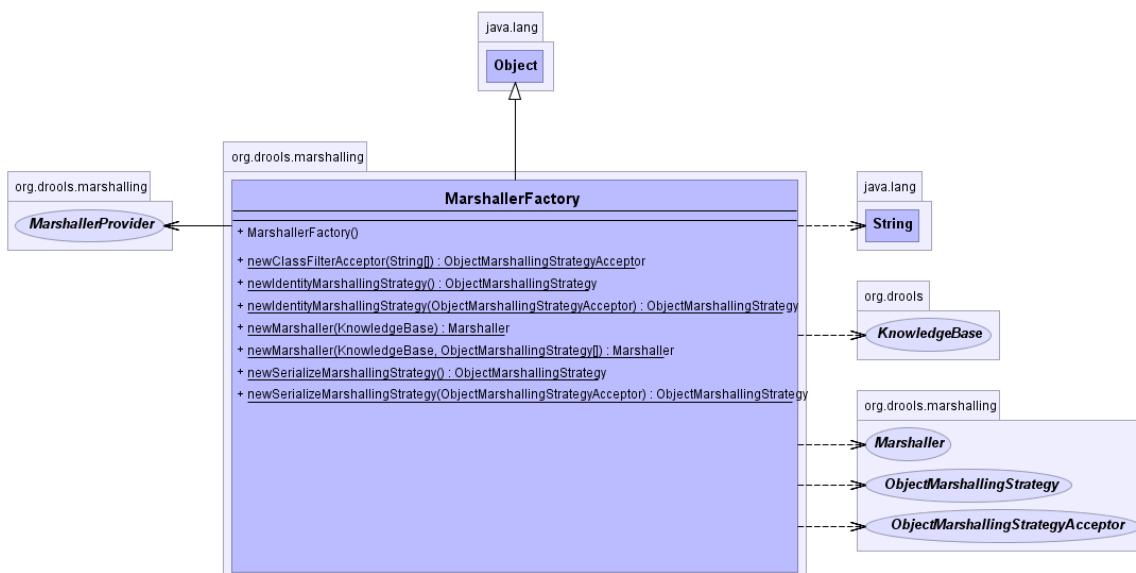


Figure 3.29. MarshalerFactory

This is the simplest way in which to use the **MarshallerFactory**:

Example 3.57. Simple Marshaller Example

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

However, ones need to have more flexibility when marshaling referenced user data. The `ObjectMarshallingStrategy` interface has been added to provide this. Two implementations of this interface are provided, and users can create their own additional ones. The two that are supplied are called:

- `IdentityMarshallingStrategy`
- `SerializeMarshallingStrategy`

The **`SerializeMarshallingStrategy`** is the default (it was used in the example above.) It simply calls either the `Serializable` or the `Externalizable` method on a user instance.

By contrast, the **`IdentityMarshallingStrategy`** creates an integer identification number for each user object and stores these in a map, whilst the identification is written to the stream. Whilst unmarshaling, it accesses the `IdentityMarshallingStrategy` map to retrieve the instance. (Hence, if the `IdentityMarshallingStrategy` is used, it remains stateful for the life of the `Marshaller` instance and will create identifiers and keep references to every objects that it attempts to marshal.) Here is the code to use with an `IdentityMarshallingStrategy`:

Example 3.58. IdentityMarshallingStrategy

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

For added flexibility the `ObjectMarshallingStrategyAcceptor` interface has also been provided. Each `ObjectMarshallingStrategy` contains this interface. The `Marshaller` has a chain of strategies, and when it attempts to read or write to or from a user object, it iterates the strategies, "asking" them if they accept responsibility for marshaling the user object. One of the implementations provided is called the **`ClassFilterAcceptor`**. It allows strings and wild cards to be used to match classnames. The default is `*.*` so, in the above example, the `IdentityMarshallingStrategy` to be used is that which has the default `*.*` acceptor.

To serialize every class bar one, (for which the identity look-up shall be used), do this:

Example 3.59. IdentityMarshallingStrategy with Acceptor

```
ByteArrayOutputStream baos = new ByteArrayOutputStream();
ObjectMarshallingStrategyAcceptor identityAcceptor =
    MarshallerFactory.newClassFilterAcceptor( new String[] { "org.domain.pkg1.*" } );
ObjectMarshallingStrategy identityStrategy =
    MarshallerFactory.newIdentityMarshallingStrategy( identityAcceptor );
ObjectMarshallingStrategy sms = MarshallerFactory.newSerializeMarshallingStrategy();
```

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



Note

The acceptance checking order is in the supplied array's natural order.

The Rule Language

4.1. Overview

Jboss Rules has a "native" rule language. This format is very light in terms of punctuation, and supports natural and domain specific languages via "expanders" that allow the language to adapt to your problem domain. This chapter explains this native rule format.

The diagrams used to present the syntax are known as *railroad* diagrams, and are like flow charts for the language terms. Interested readers can also refer to Antlr3 grammar for the rule language which is in **DRL.g** but this is not required. 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 and 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
```

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. Structure of a Rule

A rule has the following rough structure:

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

Punctuation is, for the most part, 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 the case of domain specific languages. When using a domain specific language each line is processed before the following line and spaces may be significant to the domain language.

4.2. Keywords

This section introduces the concept of *hard* and *soft keywords*.

Hard keywords are reserved, meaning that they cannot be used when naming domain objects, properties, methods, functions or any other elements that are used in the rule text.

Here is the list of hard keywords. Do not use these as identifiers when writing rules:

| | | |
|---------|-------|------------|
| true | false | accumulate |
| collect | from | null |
| over | then | when |

Soft keywords are only recognized in their immediate context. These means one can use these words in any other place.

Here is the 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 | |

Both hard and soft keywords can be used in method names, such as, for example, `notSomething()` and `accumulateSomething()`.

The DRL language also allows one to escape hard keywords on rule text. This feature enables one to use existing domain objects without worrying about keyword "collisions." To escape a word, simply enclose it in grave accents, like this:

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

Use the escape anywhere, except within code expressions in the left-hand side and right-hand side code blocks. Here are some examples of proper 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. Upon encountering them, it strips them out, unless they are inside *semantic code blocks*, like the right-hand side of a rule. There are two kinds, these being *single-line comments* and *multi-line comments*.

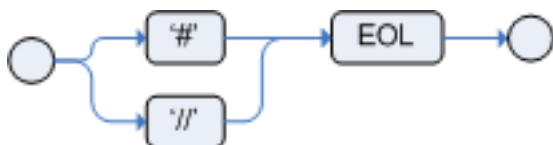


Figure 4.1. A single-line comment

Use either # or // to create single-line comments. The parser will ignore anything in the line after the comment symbol.

Here is an 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

```



Figure 4.2. A multi-line comment

Use multi-line comments to indicate blocks of text, both inside and outside of semantic code blocks.

Here is an 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 possesses standardized error message functionality. This helps users to find and resolve problems quickly and easily. Read this section to learn how to identify and interpret those error messages and to receive instruction on how to solve some of the problems that they report.

This is the format of an error message:

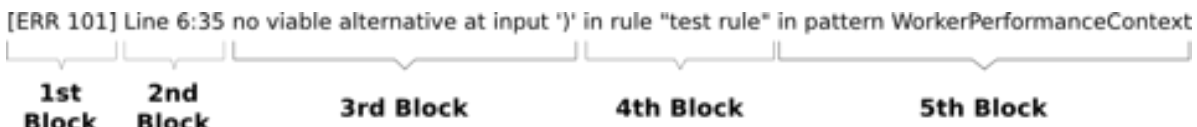


Figure 4.3. Error Message Format

1st Block

This area identifies the error code.

2nd Block

This area displays line and column information.

3rd Block

This area displays some text describing the problem.

4th Block

This is the first context. It usually indicates the rule, function, template or query in which the error occurred. (This block is not mandatory.)

5th Block

This identifies the pattern in which the error occurred. (This block is not mandatory.)

Here are all of the error messages:

4.4.1. 101: No viable alternative

This message appears for the most common errors. It occurs when the **parser** comes to a decision point but cannot identify an alternative. Here are some examples:

```
rule one
when
  exists Foo()
  exists Bar()
then
end
```

This first example generates this message:

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

At first glance, this may have seemed to be valid syntax but, in reality, it is not because **exists != exists**.

Here is another example:

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

The above code generates this error message:

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

This message means that the **parser** has encountered the `WHEN` token, (which is actually a "hard" keyword), but it is in the wrong place because the rule name is missing.

The same error message also appears when one makes a simple lexical mistake. Here is an example of just such a problem:

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

The closing quote is missing, so the 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 is accurate but, in some cases, the **parser** generates a **0: -1** position. If so, check that quotes quotes, apostrophes and parentheses have all been closed.

4.4.2. 102: Mismatched input

This error indicates that the **parser** was looking for a particular symbol that it could not find at the current input position. Here are some examples:

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

That code generates this message:

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

To fix this problem, complete the rule statement.



Note

A **0: -1** position, means that the **parser** has reached the end of source. The next piece of 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 (,) with an **&&** operator.



Note

If there is more than one error message, it is a good idea to try to fix them one by one, starting with the first. Some error messages are generated merely as consequences of others.

4.4.3. 103: Failed predicate

This means that a *validating semantic predicate* evaluated as **false**. Usually, these semantic predicates are used to identify "soft" keywords. This example shows that exact 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
```

Here is the error message produced by this sample:

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

fdsfdsfds is invalid text and the **parser** could not identify it as the soft keyword rule.



Note

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

4.4.4. 104: Trailing semi-colon not allowed

This error is associated with the eval clause. It occurs if the expression is incorrectly terminated with a semi-colon. Here is an example:

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

This error message appears due to the trailing semi-colon within the eval clause:

```
[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.

4.4.5. 105: Early Exit

This occurs if the recognizer encounters a sub-rule in the grammar that does not match any alternative. In other words, it means that the **parser** has entered a branch from which there is no way out. Here is an example that illustrates this scenario:

```
template test_error
    aa s 11;
end
```

Here is the resulting error message in full:

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

To fix this problem, 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.

Having studied this section, the user now knows the meaning of the error messages and ways of fixing the problems that they indicate.

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 contain multiple knowledge packages built on it. It is common practice 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 statement which must be at the top of the file. In all cases, the semicolons are optional.

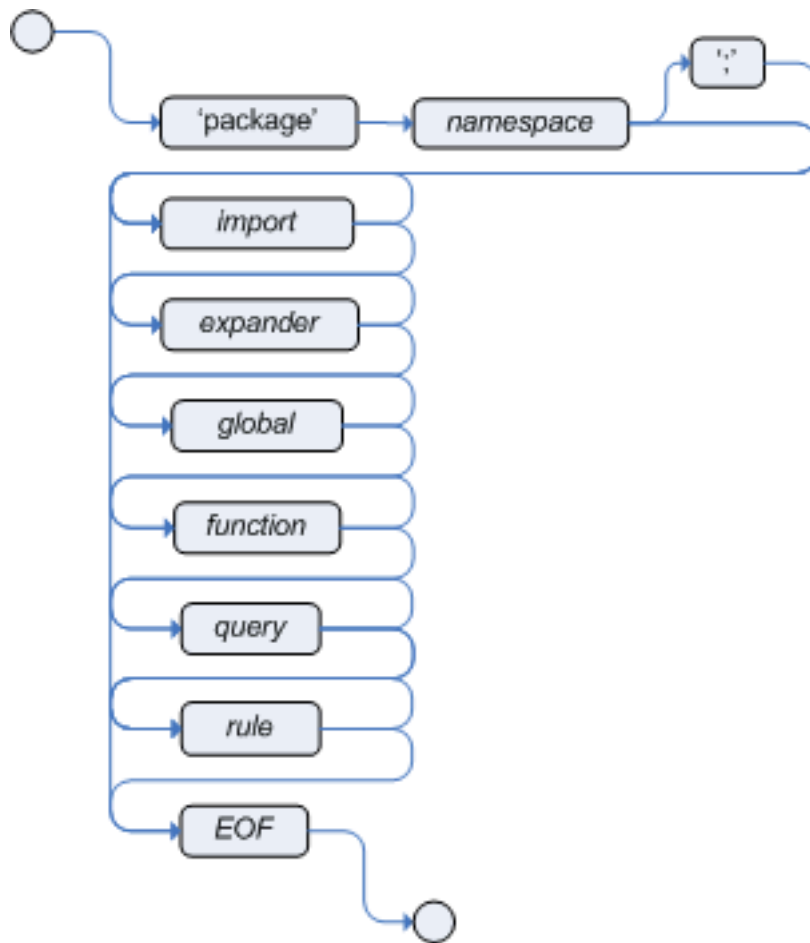


Figure 4.4. package

Note

Notice that any rule attribute (as described the section Rule Attributes) may also be written at package level, superseding the attribute's default value. The modified default may still be replaced by an attribute setting within a rule.

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 Java package of the same name, and also from the package `java.lang`.

4.5.2. global



Figure 4.6. global

With global you define global variables. They are used to make application objects available to the rules. Typically, they are used to provide data or services that the rules use, especially application services used in rule consequences, and to return data from the rules, like logs or values added in rule consequences, or for the rules to interact with the application, doing callbacks. Globals are not inserted into the Working Memory, and therefore a global should never be used to establish conditions in rules except when it 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 surprising results - surprising in a bad way.

If multiple knowledge 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 obtain 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 the 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 pass data from rule to rule, assert the data as facts into 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. Functions

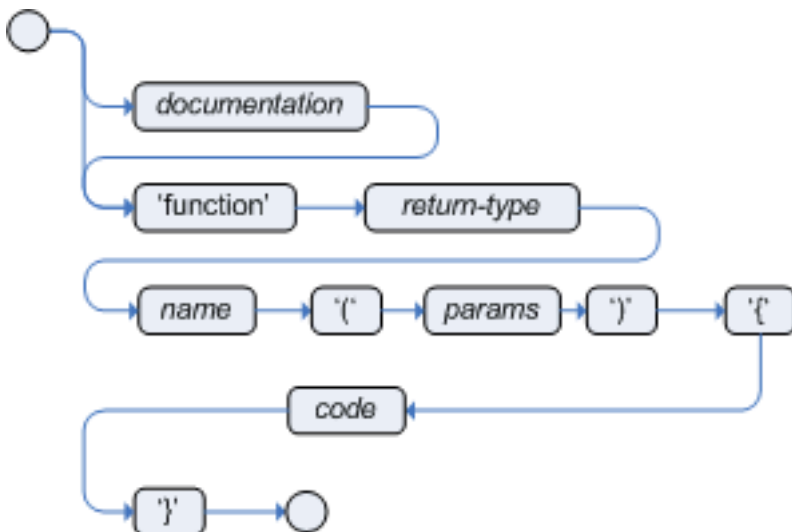


Figure 4.7. A function

Use *functions* to put semantic code into the rule source file, (rather than into normal Java classes.) They cannot do anything more than **helper** classes (in fact, the compiler generates the **helper** class "behind the scenes") but their main advantage stems from the fact that one can use them to keep all of the logic all in one place. Also one can change functions as one's needs alter (this can be both good and bad.)

Functions are most useful for invoking actions on a rule's consequence, especially if that particular action is used over and over (with, perhaps, only differing parameters for each rule; for example, the contents of an e. mail message.)

This is a standard function declaration:

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



Note

The **function** keyword is used, even though it is not actually a part of Java. Parameters are, to the function, just like normal methods (and one does not have to use parameters if they are not required.) *Return type* is just like a normal method.

An alternative is to use a static method in a helper class, like this: **Foo.hello()**. **JBoss Rules** allows one to use *function imports*. This code sample shows how to do so:

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

In both of these cases, to use the function, just call it by its name in either the consequence or inside a semantic code block. Here is a final example, showing this:

```
rule "using a static function"
when
    eval( true )
then
```



```
System.out.println( hello( "Bob" ) );
end
```

4.7. Type Declaration

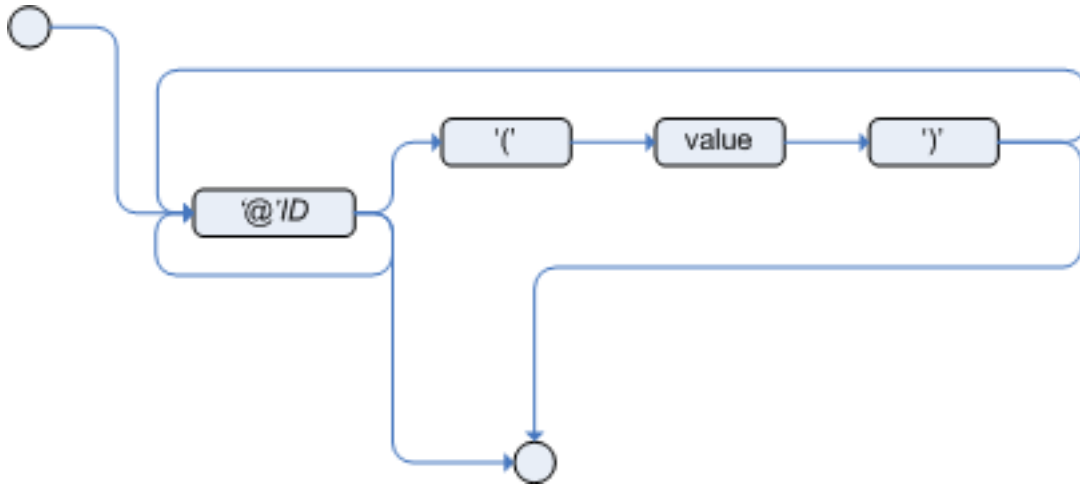


Figure 4.8. meta_data

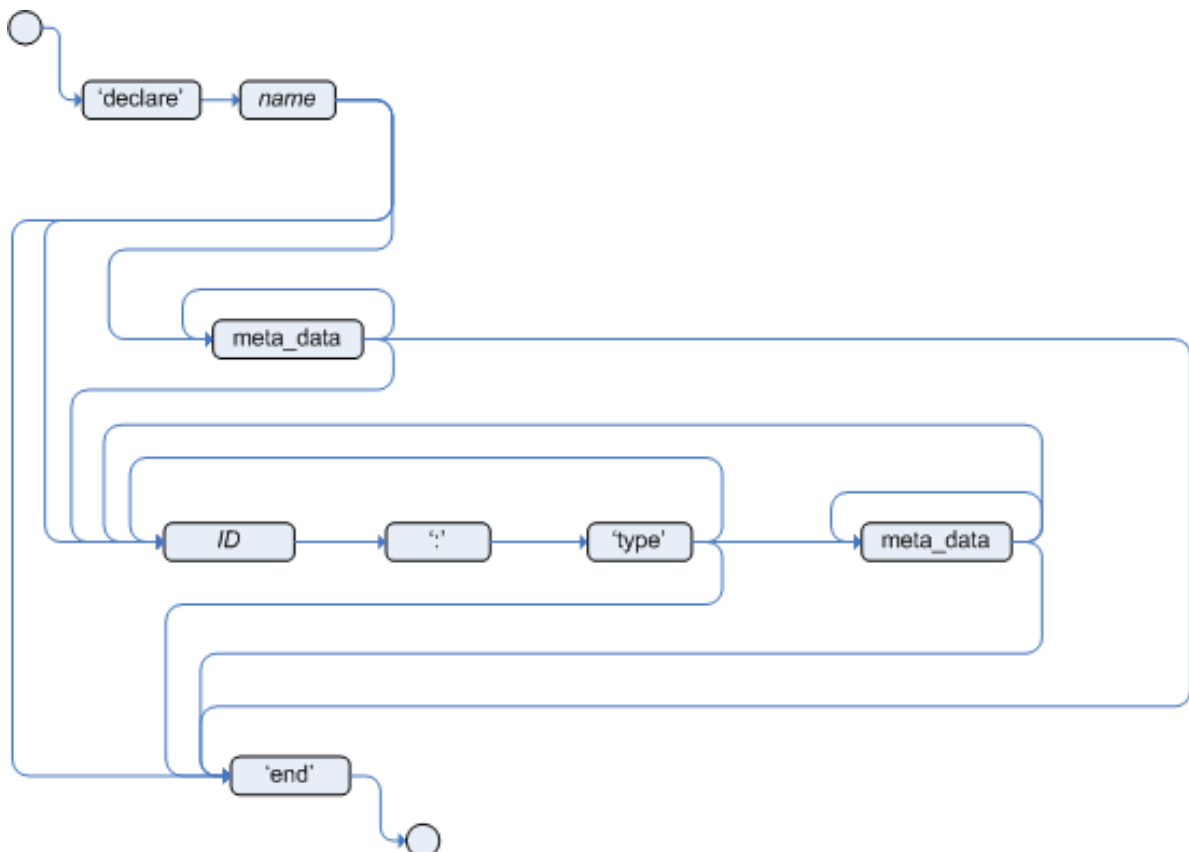


Figure 4.9. type_declaration

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**.

Example 4.1. declaring a new fact type: Address

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

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*:

Example 4.2. declaring a new fact type: Person

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

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.

Example 4.3. avoiding the need to use fully qualified class names by using import

```
import java.util.Date

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

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:

Example 4.4. generated Java class for the previous Person fact type declaration

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

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

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

Example 4.5. using the declared types in rules

```
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
```

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:

Example 4.6. declaring an arbitrary metadata attribute

```
@author( Bob )
```

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.

Example 4.7. declaring metadata attributes for fact types and attributes

```
import java.util.Date
```

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

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

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:

Example 4.8. declaring metadata for an existing type

```
import org.drools.examples.Person

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

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.

Example 4.9. declaring metadata using the fully qualified class name

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

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*.

Example 4.10. declaring a type in the org.drools.examples package

```
package org.drools.examples

import java.util.Date

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

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.

Example 4.11. handling declared fact types through the API

```
// 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" );
```

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

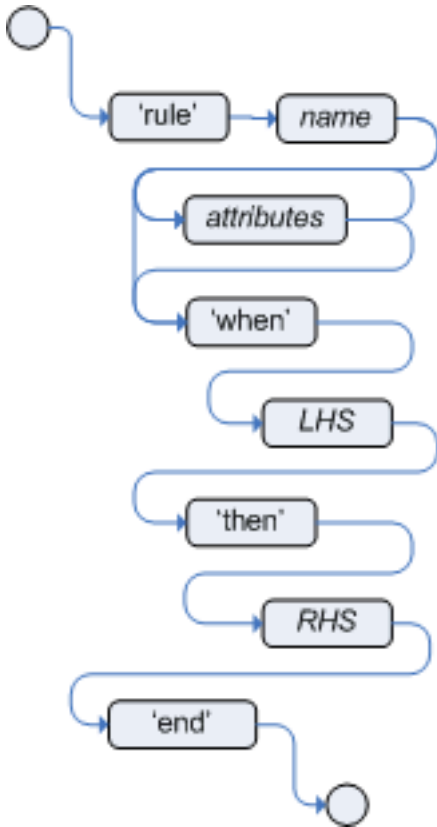


Figure 4.10. Rule

A rule specifies that when a particular set of conditions occur, specified in the left-hand side, then do what is specified as a list of actions in the right-hand side. 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, a condition is to be checked. In contrast, "when" indicates that the condition evaluation is not tied to a specific evaluation sequence or point in time, but that it happens continually, at any time during the life time of the engine; whenever the condition is met, the actions are executed.

A rule must have a name, unique within its rule package. If a rule is defined twice in a single DRL, an error will appear when one loads it. If a DRL that includes a rule name already in the package is added, the previous rule is replaced. If a rule name is to have spaces, then it will need to be enclosed in double quotes. (Always use double quotes).

Attributes are optional. Always write them on a one-per-line basis.

The left-hand side of the rule follows the when keyword (ideally on a new line), similarly the right-hand side follows the then keyword (again, ideally on a newline). The rule is terminated by the keyword end. Rules cannot be nested.

Example 4.12. Rule Syntax Overview

```
rule "<name>"
```

```

<attribute>*
when
  <conditional element>*
then
  <action>*
end

```

Example 4.13. A simple rule

```

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

```



Note

JBoss Rules attempts to preserve numbers in their primitive or object wrapper form, so a variable bound to an int primitive when used in a code block or expression will no longer need manual unboxing; unlike JBoss Rules 3.0 for which all primitives were "auto-boxed," requiring manual "unboxing." A variable bound to an object wrapper will remain as an object; the existing JDK 1.5 and JDK 5 rules to handle auto-boxing and 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.

4.8.1. Rule Attributes

These provide a declarative way to influence the behaviour of a rule. Some are quite simple, whilst others are part of complex sub-systems such as `RuleFlow`. To obtain the most value from **JBoss Rules** it is beneficial to gain a thorough understanding of each attribute. Read this section to do just that.

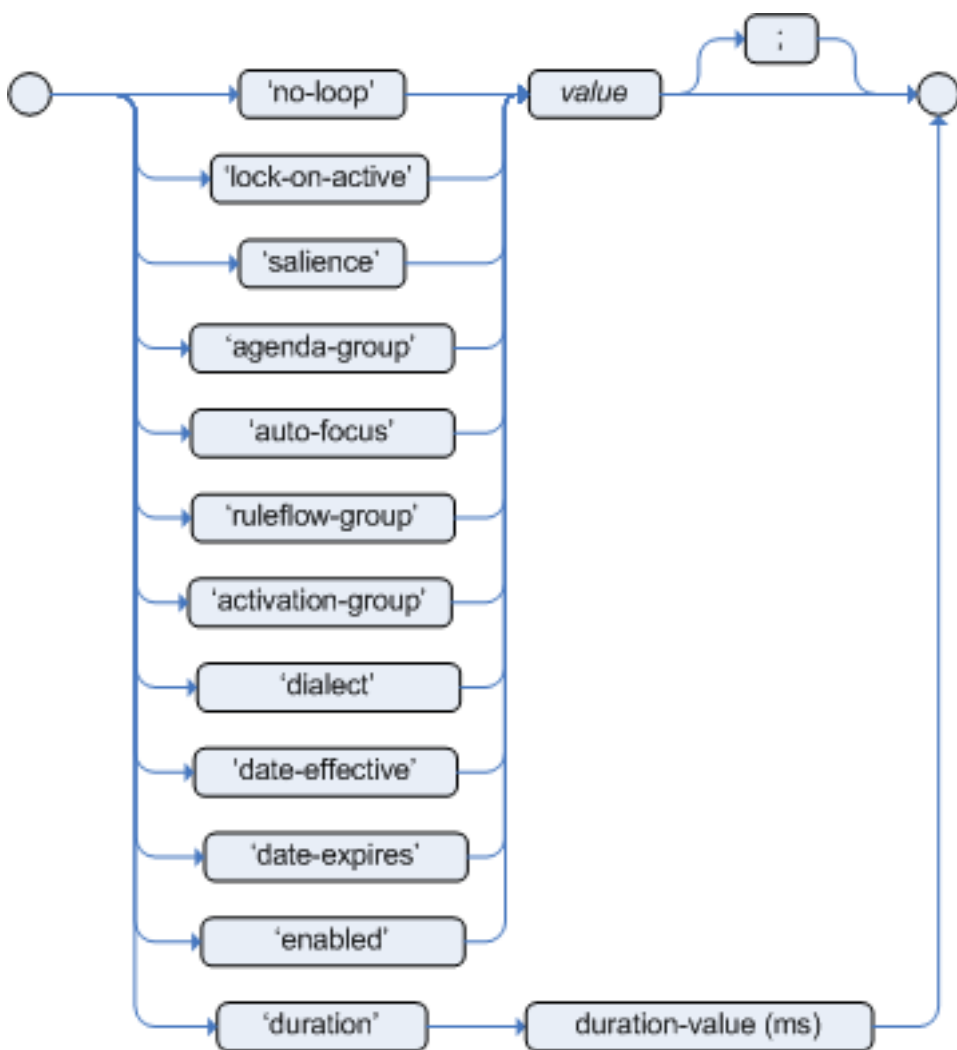



Figure 4.11. rule attributes

Table 4.1. Rule Attributes

| Attribute | Default Value | Type | Comments |
|----------------|---------------|---------|--|
| no-loop | false | Boolean | When the rule's consequence modifies a fact it may cause the rule to activate again, causing recursion. Set no-loop to true so that the attempt to create the activation for the current set of data will be ignored. |
| ruleflow-group | N/A | string | Use the <i>rule-flow</i> feature to exercise control over the firing of rules. (Rules that are assembled by the same ruleflow-group identifier will only fire when their group is active.) |
| lock-on-active | False | 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 |

| Attribute | Default Value | Type | Comments |
|------------------|-----------------------------|--|---|
| | | | 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 | 0 | integer | Each rule has a salience attribute that can be assigned an integer number, the default being zero. Salience is a form of priority whereby rules with higher values are given higher priority when ordered onto the activation queue. |
| agenda-group | MAIN | string | Agenda groups allow the user to partition the Agenda providing more execution control. Only rules in the agenda group that has acquired the focus are allowed to fire. |
| auto-focus | false | Boolean | If a rule, for which the auto-focus value is true , is activate and if the rule's agenda group does not yet have focus, then focus is granted to it, allowing the rule to potentially fire. |
| activation-group | N/A | string | This string value identifies rules that belong to the same activation group. Rules in such a group will only fire exclusively of each other. In other words, the first rule in an activation group to fire will cancel the other rules' activations, stopping them from firing in turn. <div data-bbox="938 1503 1439 1749" style="border: 1px solid gray; padding: 5px; margin-top: 10px;">  <p>Note</p> <p>This was once called the Xor group but, strictly speaking, it does not meet the definition of Xor.</p> </div> |
| dialect | as specified by the package | string, with possible values being java and mvel | Use this attribute to specify the language to be used for any code expression on either the left-hand side or the right-hand side. Currently two dialects are available, Java and the MVFLEX Expression Language. (Whilst the dialect can also be specified at the package level, this |

| Attribute | Default Value | Type | Comments |
|----------------|------------------|--|---|
| | | | attribute allows one to override the package definition for a rule.) |
| date-effective | N/A | string, containing date and time definitions | A rule can only activate if the current date and time indicate it is after the timestamp set in this attribute. |
| date-expires | N/A | string, containing date and time definitions | A rule will not activate if the current date and time indicate it is after the timestamp set in this attribute. |
| duration | no default value | long | Use this attribute to dictate that the rule will fire after a specified duration, provided that it is still true . |

This code extract shows how to put some common attributes into practice:

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

4.8.2. Timers and Calendars

Rule's now suport both interval and cron based timers, which replace the now deprecated duration attribute.

Example 4.14. Sample timer attribute uses

```
timer ( int: <initial delay> <repeat interval?? )
timer ( int: 30s )
timer ( int: 30s 5m )

timer ( cron: <cron expression> )
timer ( cron: * 0/15 * * * ? )
```

Interval "int:" timers follow the JDK semantics for initial delay optionally followed by a repeat interval. Cron "cron:" timers follow standard cron expressions:

Example 4.15. A Cron Example

```
rule "Send SMS every 15 minutes"
  timer (cron:* 0/15 * * * ?)
  when
    $a : Alarm( on == true )
  then
    channels[ "sms" ].insert( new Sms( $a.mobileNumber, "The alarm is still on" );
  end
```

Example 4.16. Adapting a Quartz Calendar

```
Calendar weekDayCal = QuartzHelper.quartzCalendarAdapter(org.quartz.Calendar quartzCal)
```

Calendars are registered with the StatefulKnowledgeSession:

Example 4.17. Registering a Calendar

```
ksession.getCalendars().set( "week day", weekDayCal );
```

They can be used in conjunction with normal rules and rules including timers. The rule calendar attribute can have one or more comma calendar names.

Example 4.18. Using Calendars and Timers together

```
rule "weekdays are high priority"
  calendars "weekday"
  timer (int:0 1h)
  when
    Alarm()
  then
    send( "priority high - we have an alarm" );
  end

rule "weekend are low priority"
  calendars "weekend"
  timer (int:0 4h)
  when
    Alarm()
  then
    send( "priority low - we have an alarm" );
  end
```

4.8.3. Left-Hand Side Conditional Elements

The *left-hand side* (LHS) is a common name for the *conditional* (when) part of the rule. It consists of *conditional elements*. (It is possible to not have any. If the left-hand side is left empty, it is re-written as `eval(true)`. This means that the rule's condition will always remain true.)

The left-hand side will be activated once, this being when a new working memory session is created.



Figure 4.12. The Left-Hand Side

Here is a rule without a conditional element:

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

This is re-written internally as:

```
rule "no CEs"
  when
    eval( true )
```

```
then
  <action>*
end
```

Conditional elements work on one or more *patterns* (these are described in more detail below.) The most common one is **and**, which is implied when there are multiple, totally-unrelated patterns in the left-hand side of a rule.



Note

In contrast to the **or** pattern, an **and** cannot have a *leading declaration binding*. This is because a declaration can only refer to a single fact, and when the **and** is "satisfied", it matches more than one fact, hence it would not know to which of these it should be bound.

4.8.3.1. The Pattern

A *pattern* is the most important type of conditional element. The entity relationship diagram below provides an overview of the various parts that make up the pattern's constraints and shows how they work together. Later in this section each part is covered in more detail with further diagrams and sample code.

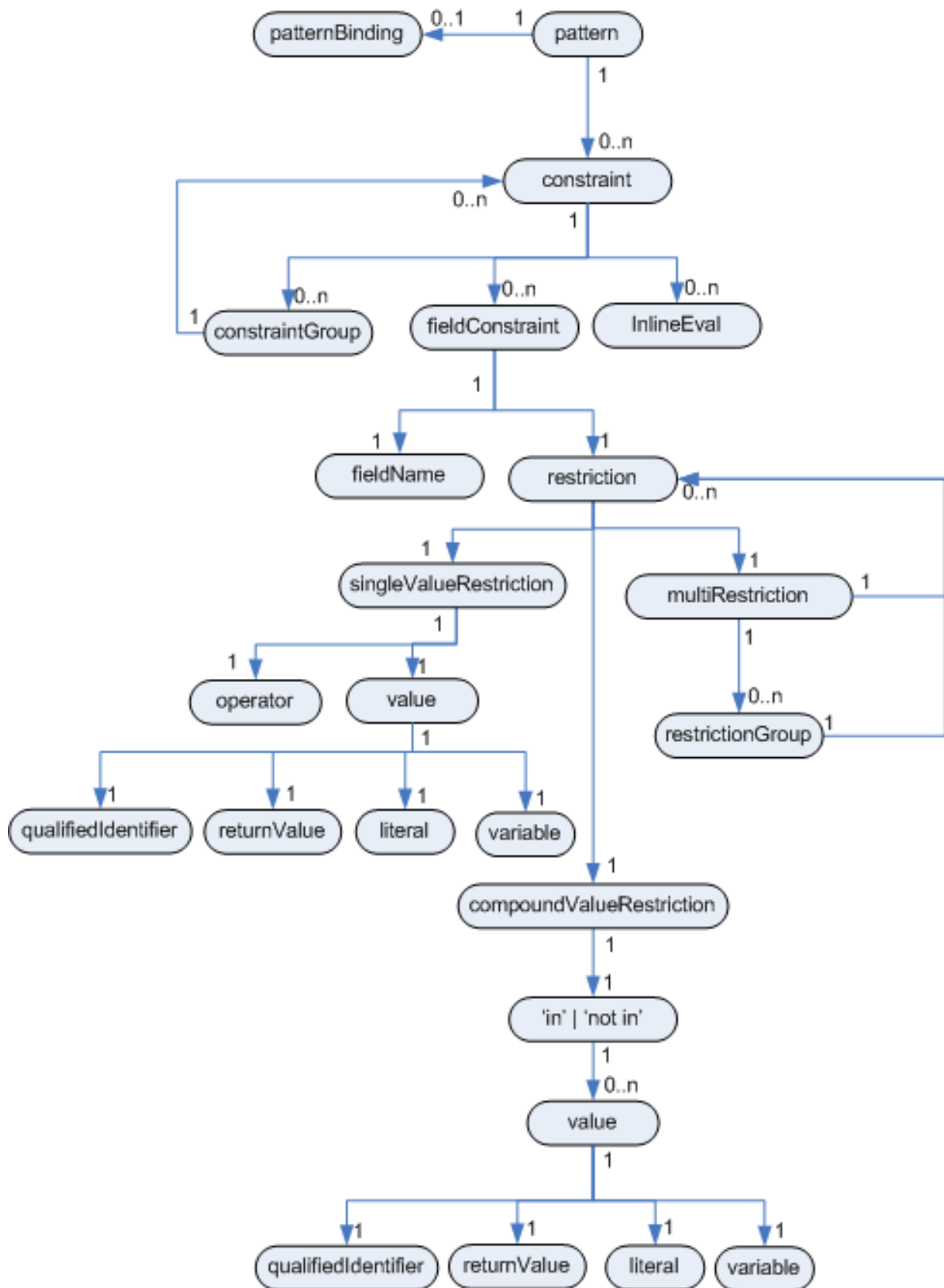


Figure 4.13. Pattern Entity Relationship Diagram

Look at the top of the entity relationship diagram and observe that the pattern consists of zero or more constraints and that pattern binding is optional. This next diagram shows the format of its syntax:



Figure 4.14. Pattern

In its simplest form, (meaning with no constraints), a pattern matches against a fact of a given type. In the following case the type is **Cheese**, which means that the pattern will match against every **Cheese** object in the working memory.



Note

The "type" need not be the actual class of some fact object. Patterns may refer to *super-classes* or even interfaces and, therefore, it is possible that they may match facts from many different classes.

```
Cheese( )
```

Use a pattern binding variable, such as **\$c**, to refer to the matched object. A helpful option is to use make use of the **\$** prefix. This can be advantageous when dealing with complex rules as it helps one to differentiate between variables and fields more easily.

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

The key element the syntax is the parentheses. It is within these that the constraints are placed. The key types are *field constraints*, *inline evals* and *constraint groups*.

Use any of **,** **&&** or **||** to separate constraints. However, note that they have slightly different abilities.



Figure 4.15. Constraints

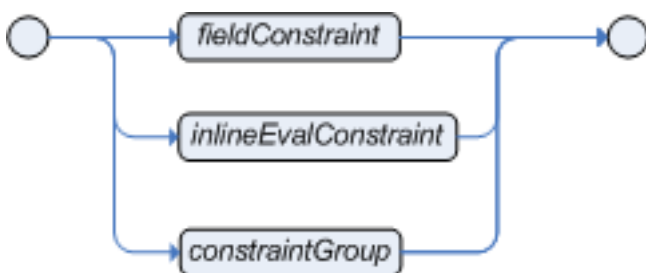


Figure 4.16. Constraint



Figure 4.17. constraintGroup

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" )
```

In this example, there are three constraint groups, each of which has a single constraint:

1. The type is stilton as **type == "stilton"**.
2. The price is less than ten, as **price < 10**.
3. The age is "mature," as **age == "mature"**.

The **&&** and **||** separators allow groups to have multiple constraints. Here is an example:

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

In this case, there are two constraint groups. The first has two constraints and the second has one constraint.

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

1. **&&**
2. **||**
3. **,**

To change the evaluation priority, use parentheses, as for any logic or mathematical expression. Here is an example:

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

In this case, the use of parentheses ensures that the **||** connective is evaluated before the **&&** connective.

Be aware that, besides having the same semantics, the **&&** and **,** connectives are resolved with different priorities. Hence **,** cannot be embedded in a composite constraint expression:

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

4.8.3.1.1. Field Constraints

Use *field constraints* to place a restriction on a named field. The field name can, optionally, have a variable binding.



Figure 4.18. fieldConstraint

Restrictions come in three forms:

- single-value restrictions

- compound-value restrictions
- multi-restrictions

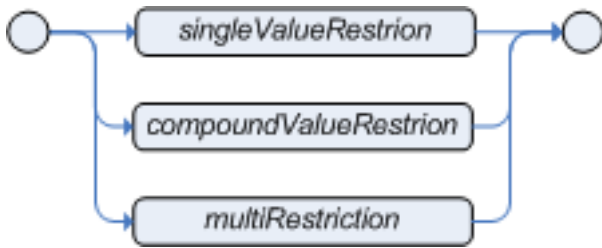


Figure 4.19. restriction

4.8.3.1.2. Java Beans as Facts

A field is derived from an object's *accessible method*. If a model objects follow the *Java bean pattern*, then fields are exposed using either the `getXXX` method or the `isXXX` method, in which cases these methods take no arguments but return something.

Access fields within patterns by using the bean naming convention, (so that, for instance, `getType` is accessed as `type`. (**JBoss Rules** uses the standard Java Development Kit **Introspector** class to undertake this mapping process.)

Referring back to the **Cheese** class example, the **Cheese(type == "brie")** pattern applies the `getType()` method to a **Cheese** instance. If a field name cannot be found, the compiler will resort to using the name as a method without arguments. Thus, the `toString()` is called due to a **Cheese(toString == "cheddar")** constraint. In this case, use the full name of the method with correct capitalization but without parentheses. Ensure that methods being accessed do not take parameters, and that they are, in fact, **accessors** which will not change the state of the object in a way that affects the rules. (Remember that the rule engine caches the results of its matching in between invocations for performance reasons.)

4.8.3.1.3. Values

The field constraints can take a variety of values, including:

- literals
- qualifiedIdentifiers (enums)
- variables
- returnValues

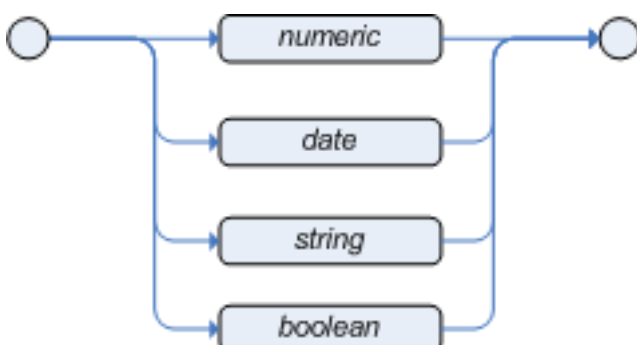


Figure 4.20. literal



Figure 4.21. qualifiedIdentifier



Figure 4.22. variable



Figure 4.23. returnValue

To checks against fields that are null, use `==` and `!=` in the usual way. The literal `null` keyword, (as in `Cheese(type != null)`, whereby the evaluator will not throw an exception will return `true` if the value is null.)

The system always attempts to undertake *type coercion* if the field and the value are of different types. If one attempts to do a "bad" coercion, an exception will occur. For instance, providing `ten` as a string in a numeric evaluator will cause an exception, whereas providing `10` will coerce to a numeric ten. (Coercion always favours the field type, not the value type.)

4.8.3.1.4. The Single Value Restriction

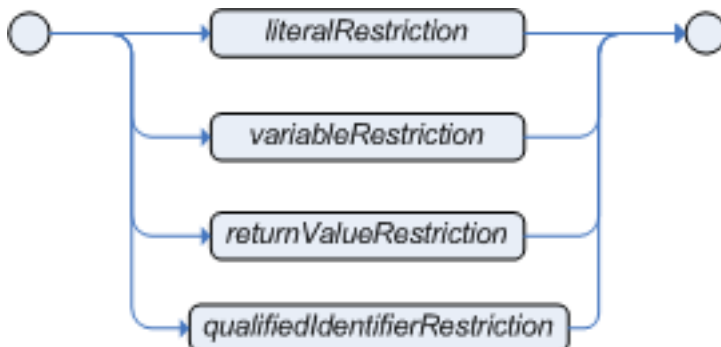


Figure 4.24. singleValueRestriction

4.8.3.1.5. Operators

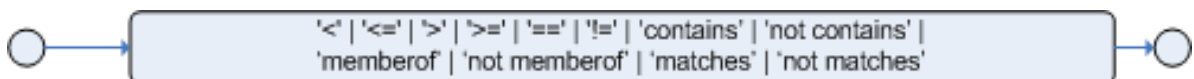


Figure 4.25. Operators

The `==` and `!=` operators are valid for all types. Other relational operators may be used whenever the type values are ordered; for date fields, `<` means "before." The pair of `matches` and `not matches` only apply to string fields, whereas `contains` and `not contains` require the field to be that of a `Collection` type. (It will attempt to coerce it to the correct value for the evaluator and the field.)

The matches Operator

This matches a field with any valid Java regular expression. The regular expression is normally a string literal, but one is also allowed to use variables that resolve to a valid regular expression.



Important

In contrast to Java, escapes are not needed within regular expressions written as string literals.

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

The **not matches** Operator

This operator returns **true** if the string does not match the regular expression. The same rules apply as for the **matches** operator. Here is an example of its use:

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

The **contains** Operator

Use this operator to check whether a collection or array field contains the specified value.

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

The **not contains** Operator

Use this operator to check whether a specified value is absent from a collection or array field.

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



Note

To ensure backward compatibility, the **excludes** operator is still supported. It is synonymous with **not contains**.

The **memberOf** Operator

Use this operator to check whether a field is a member of a collection or an array; note that the collection must be a variable.

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

The **not memberOf** Operator

Use this operator to check whether a field is not a member of a collection or an array; note that the collection must be a variable.

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

The **soundsLike** Operator

This operator is similar to **matches** but it checks as to whether or not a word has almost the same sound as the given value. To do so, it uses the the **Soundex** algorithm and is based on English pronunciations of the words.

```
// match cheese "fubar" or "foobar"
Cheese( name soundslike 'foobar' )
```

4.8.3.1.6. Literal Restrictions

These are the simplest form of restriction. They evaluate a field against a specified literal, which may be numeric, a date, a string or a Boolean.



Figure 4.26. literalRestriction

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

Numeric

All of the standard Java numeric primitives are supported.

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

Date

The default date format is **dd-mmm-yyyy**. To change this, provide an alternative date format mask for the `drools.dateformat` property. (If more control is required, use the `inline-eval` constraint.)

```
Cheese( bestBefore < "27-Oct-2013" )
```

String

Use any valid Java string.

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

Booleans

One can only use **true** or **false**; 0 and 1 are not acceptable. A lone Boolean field (as in **Cheese(smelly)**) is not permitted; it must be compared to a Boolean literal.

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

Qualified Identifier

Both Java Development Kit 1.4 and 1.5-style **enums** are supported but the latter can only be used with a JDK 5 environment.

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

4.8.3.1.7. Bound Variable Restriction



Figure 4.27. variableRestriction

One can bind variables to facts and their fields and then use them in subsequent field constraints. A bound variable is called a *declaration*. The type of the field being constrained

determines which operators are valid; coercion will be attempted where possible. Use the `==` operator to bind variable restrictions for very fast performance.

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

In this example, the `likes` variable is bound to the `favouriteCheese` field of every matching `Person` instance. (It then constrains the type of `Cheese` in the next pattern.) Any valid Java variable name can be used, and it may be prefixed with `$`, which is often used to help differentiate declarations from fields.

The example below shows a declaration for `$stilton`, bound to the object matching the first pattern. It is used in conjunction with a `contains` operator. (Note the optional use of `$`.)

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

4.8.3.1.8. Return Value Restriction



Figure 4.28. returnValueRestriction

A *return value restriction* is a parenthesized expression comprising of literals, any valid Java primitive or object, previously bound variables, function calls and operators. The functions used must not return time-dependent results.

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

4.8.3.1.9. Compound Value Restriction

Use the *compound value restriction* when there is more than one possible value to match. (Currently only the `in` and `not in` evaluators support this.)

The second operand must be a comma-separated list of values, enclosed in parentheses. Values may be given as variables, literals, return values or qualified identifiers. Both evaluators are actually "syntactic sugar", internally rewritten as a list of multiple restrictions using the `!=` and `==` operators.

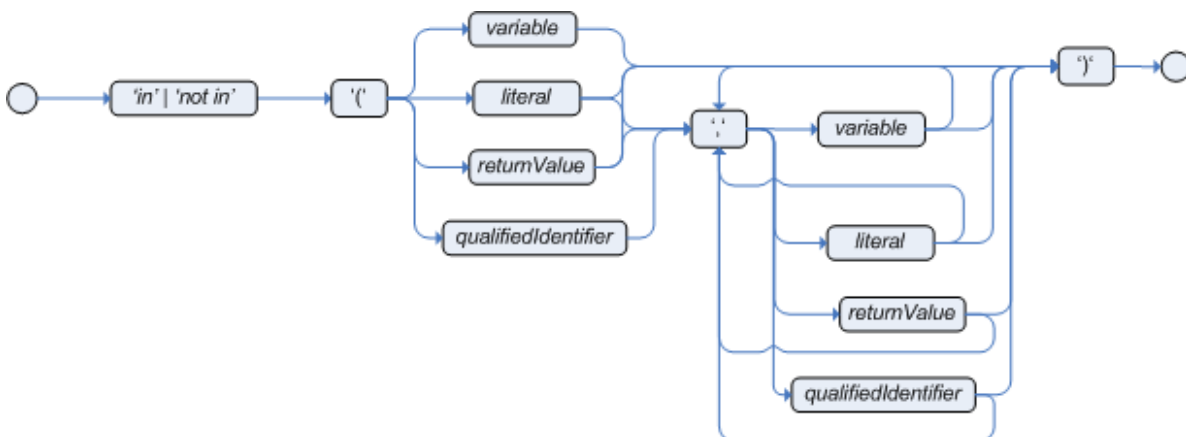


Figure 4.29. compoundValueRestriction

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

4.8.3.1.10. Multi-Restrictions

Use the *multi-restriction* constraint to place more than one restriction on a field (using the `&&` or `||` separators.) Grouping via parentheses is permitted; this will result in a recursive syntactical pattern.

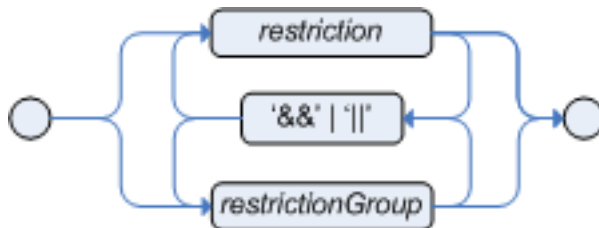


Figure 4.30. multiRestriction



Figure 4.31. restrictionGroup

```
// Simple multi restriction using a single &&
Person( age > 30 && < 40 )
// 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" )
```

4.8.3.1.11. Inline Eval Constraints



Figure 4.32. Inline Eval Expression

An *inline eval* constraint can use any valid dialect expression as long as it resolves to a primitive Boolean. The expression must be constant over time. Any previously-bound variable, from the current or previous pattern, can be used; *auto-vivification* is also used to automatically create field binding variables.



When an identifier that is not a current variable is found, the builder checks to determine 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.

This example will find all of the possible male-female pairs in which the male is two years older than the female; the **age** variable is auto-created in the second pattern by the auto-vivification process.

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

4.8.3.1.12. Nested Accessors

JBoss Rules allows one to use *nested accessors* in the field constraints. Use MVFLEX Expression Language accessor graph notation to do so. (Field constraints possessing nested accessors are actually re-written as MVFLEX Expression Language `inline-eval` constraints.)



Warning

Take care when using nested accessors as the working memory is not aware of any of the nested values and does not know when they change. Always regard them as immutable whilst any of their parent references are in working memory.

To modify a nested value, remove the parent objects first and re-assert them afterwards. If there is only one parent at the root of the graph, one can use the MVEL dialect's **modify** construct and its *block setters* to write the nested accessor assignments whilst retracting and inserting the the root parent object as required. (Nested accessors can be used on either side of the operator symbol.)

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

This is rewritten internally as an MVEL `inline eval`:

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



Warning

Use nested accessors carefully as they have a much greater performance impact than direct field accesses.

4.8.3.2. The and Conditional Element

Use the **and** conditional element to group other conditional elements into a logical conjunction.

The root element of the left-hand side is an implicit prefix **and**. It does not need to be specified.

JBoss Rules supports both **and** as both a prefix and as an infix but the prefix is the preferred option as its implicit grouping eliminates confusion.



Figure 4.33. prefixAnd

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

```
when
```

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

Should it be needed, infix **and** is supported, along with explicit grouping via parentheses.

Note

The **&&** symbol can be used as an alternative to **and**. This is deprecated but is still currently available for legacy support reasons.

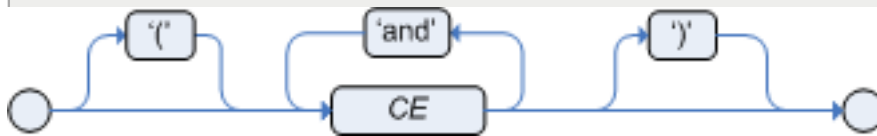


Figure 4.34. infixAnd

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

4.8.3.3. The or Conditional Element

Use the **or** conditional element to group other conditional element into a logical disjunction.

Note

JBoss Rules allows one to use **or** as either a prefix or as an infix, but the prefix is the preferred option as its implicit grouping avoids confusion.

The behaviour of this conditional element is different to that of the connective **||** for field constraints and restrictions. The engine actually has no understanding of **or**; rather, via a number of different logic transformations, a rule that uses **or** is rewritten as a number of sub-rules. This results in a rule that has a single root node **or** and one sub-rule for each of its conditional elements. Each sub-rule can activate and fire like any normal rule; there is no special behaviour or interaction between them, a fact which sometimes confuses new developers.



Figure 4.35. prefixOr

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

Infix **or** is supported along with explicit grouping with parentheses, should it be needed.



Note

The `||` symbol can be used as an alternative to `or`. This is deprecated but it is still currently available for legacy support reasons.



Figure 4.36. infixOr

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

One has the option of using *pattern binding* with `or`. This means that each resulting sub-rule will bind to the pattern. Each pattern must be bound separately, using eponymous variables, as in this example:

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

Myriad sub-rules are created when `or` is used, one for each possible outcome. The simple example shown above will generate two rules. These will function independently of each other within the working memory, meaning that they can both match, activate and fire. No short-cuts are taken.

It can be helpful to consider `or` as a way to generate two or more similar rules. A single rule may have multiple activations if two or more terms of the disjunction are true.

4.8.3.4. The eval Conditional Element



Figure 4.37. eval

The **eval** conditional element is essentially a "catch-all" which allows one to execute any semantic code that returns a primitive Boolean. This code can refer either to variables that were bound to the left-hand side of the rule, or to functions in the rule package.



Warning

Do not overuse **eval** because it reduces the declarativeness of the rules which can lead to a poorly performing engine. Whilst **eval** can be used anywhere in the patterns, best practice dictates that one should add it as the last conditional element in the left-hand side of a rule.

Evals cannot be indexed and, thus, are not as efficient as *field constraints*. However they are ideal for use as functions that return values which are subject change over time, (an ability which field constraints do not possess.)


```

p1 : Parameter()
p2 : Parameter()
eval( p1.getList().containsKey(p2.getItem()) )
// call function isValid in the LHS
eval( isValid(p1, p2) )

```

4.8.3.5. The not Conditional Element

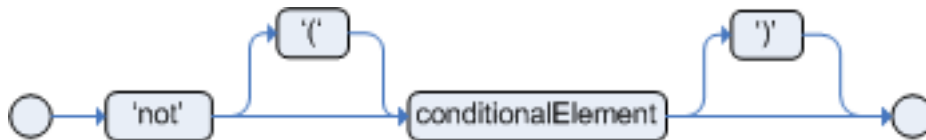


Figure 4.38. not

The **not** conditional element is the first-order logic's *non-existential quantifier*. Its purpose is to check that something does not exist in the working memory.

The **not** keyword must be followed by conditional element that are, themselves, in parentheses. (In the simplest use cases, these parentheses can be omitted.)

```
not Bus()
```

```

// Brackets are optional:
not Bus(color == "red")

// Brackets are optional:
not ( Bus(color == "red", number == 42) )

// "not" with nested infix and - two patterns,
// brackets are requires:
not ( Bus(color == "red") and Bus(color == "blue") )

```

4.8.3.6. The exists Conditional Element

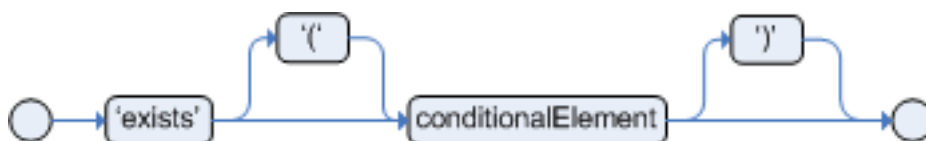


Figure 4.39. exists

The **exists** conditional element is the *first order logic's existential quantifier*. Its purpose is to check for the existence of something in the working memory. Some programmers find it easiest to think of **exists** as meaning "there is at least one". (It is different from just having the pattern on its own, which is more like saying "for each one of.")

When **exists** is used with a pattern, the rule will only activate once, regardless of how much data there is in working memory that matches its condition. Since only the very existence matters, no bindings will be established.

The **exists** keyword must be followed by the conditional elements to which it applies. These must be contained within parentheses. (In the simplest of single patterns, like that depicted below, one may have the option of omitting the parentheses.)

```
exists Bus()
```

```
exists Bus(color == "red")
// brackets are optional:
exists ( Bus(color == "red", number == 42) )
// "exists" with nested infix and,
// brackets are required:
exists ( Bus(color == "red") and
Bus(color == "blue") )
```

4.8.3.7. The forall Conditional Element



Figure 4.40. forall

The **forall** conditional element completes the first-order logic support in **JBoss Rules**. **forall** evaluates as **true** when all of the facts that match the first pattern also match every remaining pattern. Here is an 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
```

This rule selects every **Bus** object for which the type is **english**. Then, for each fact that matches this pattern, the following patterns are evaluated. If they, too, match, the **forall** conditional element will evaluate as **true**.

To state that every fact of a given type must match a set of constraints, write a simple single pattern like this:

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

By way of contrast, here are multiple patterns:

```
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
```

Forall can be nested inside other conditional elements such as **not**.



Important

Note that parentheses are only optional when dealing with single patterns, so a nested one must have them.

```
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
```



Note

As an aside, `not(forall(p1 p2 p3...))` is equivalent to this piece of code:

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



Important

Be aware that **forall** is a *scope delimiter*. Therefore, it can use any previously bound variable but no such variable bound within it will be available for use outside of it.

4.8.3.8. The from Conditional Element



Figure 4.41. from

Use the **from** conditional element to specify an arbitrary source for data to be matched by left-hand side patterns. Doing so allows the engine to "reason over" data not found in the working memory. The data source could be a sub-field on a bound variable or the result 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.

Use any expression that follows regular MVFLEX Expression Language syntax to define the object source. In this way, one can easily execute method calls, access maps and collections elements and utilise *object-property navigation*.

Here is a simple example that demonstrates both reasoning and binding to another pattern sub-field:

```
rule "validate zipcode"
when
  Person( $personAddress : address )
  Address( zipcode == "23920W") from $personAddress
then
```

Chapter 4. The Rule Language

```
# zip code is ok
end
```

This shows how to do the same thing using graph notation:

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

Previous examples were single-pattern evaluations. One can also use **from** on object sources to return a collection of objects. In this case, **from** will iterate over every objects in the collection and try to match each of them individually. Here is an example, featuring a rule designed to a ten percent discount to every item in an order:

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

The rule will fire once for every item with a value greater than one hundred on each given order.

Take caution, 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 Queensland (QLD) to sales region 1"
  ruleflow-group "test"
  lock-on-active true
when
  $p : Person( )
  $a : Address( state == "QLD") 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 Brisbane"
  ruleflow-group "test"
  lock-on-active true
when
  $p : Person( )
  $a : Address( city == "Brisbane") from $p.address
then
  modify ($p) {} #Apply discount to person in a modify block
end
```

In this example, persons in Brisbane, QLD are supposed to be assigned to Sales Region 1 and receive a discount (in other words, both rules are expected to activate and fire. However, one will find that only the second rule fires.

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

Follow these steps:

1. Review the need to use the above pattern.

It may be because there are many rules across different rule-flow groups. When rules modify working memory and other rules downstream of in rule-flow in question and needs must be reevaluated, the use of **modify** is critical. Do not, however, make other rules in the same rule-flow group place activations on one another recursively.

In this case, the no-loop attribute is ineffective, as it will only prevent a rule from activating itself recursively so use lock-on-active.

2. There are now a number of ways in which to address this issue:

- either avoid the use of **from** when it is possible to assert all facts into working memory or use nested object references in the constraint expressions (shown below).
- place the variable assigned for use in the **modify block** as the last sentence in the left-hand side condition.
- avoid the use of lock-on-active when it is possible to explicitly manage the way in which rules within the same rule-flow group place activations on one another. (This is explained below.)

Of these, the preferred solution is to minimize the use of **from** when it is possible to assert all facts directly into working memory.

In the example above, both the **Person** and the **Address** instances can be asserted into working memory. Because the graph is fairly simple, an even easier solution is to modify the rules in the following way:

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

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

3. Both rules will now fire as expected. However, it is not always possible to access nested facts in this way. Consider an example whereby a **Person** holds one or more **Addresses** and one wishes to use an *existential quantifier* to match people with at least one address that meets certain conditions. In this case, one will have to resort to the use of **from** to reason over the collection.

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

```
rule "Assign people in Queensland (QLD) to sales region 1"
  ruleflow-group "test"
  lock-on-active true
```

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

A slightly different approach does, however, exhibit the problem:

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

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

In this case, using **from** returns the `$addresses` variable. This example also introduces a new object, **assessment**, which points the way to a possible solution. If the `$addresses` variable is moved so that it becomes the last condition in each rule, both fire as expected.

Though the examples above demonstrate how to combine the use of **from** with lock-on-active without loss of rule activations, they carry the drawback of being dependent on the placement order of conditions on the left-hand side. In addition, the solutions present greater complexity for the rule author, who must suddenly keep track of conditions with the potential to cause issues.

4. A better alternative is to assert more facts into working memory. In this case, once the person's addresses are asserted into working memory it will no longer be necessary to use **from**.

Note

One will, however, encounter cases in which asserting all of the data into working memory will not be practical and other solutions will need to be found.

One option is to reevaluate the need for lock-on-active. An alternative is to directly manage the way in which rules within the same rule-flow group activate one another by including conditions in each rule that prevent them from activating each other recursively when working memory has been modified. For example, to use the example above once more, one would add a condition to the rule that checks whether the discount has already been applied and, if so, ensures that the rule does not activate.

4.8.3.9. The collect Conditional Element

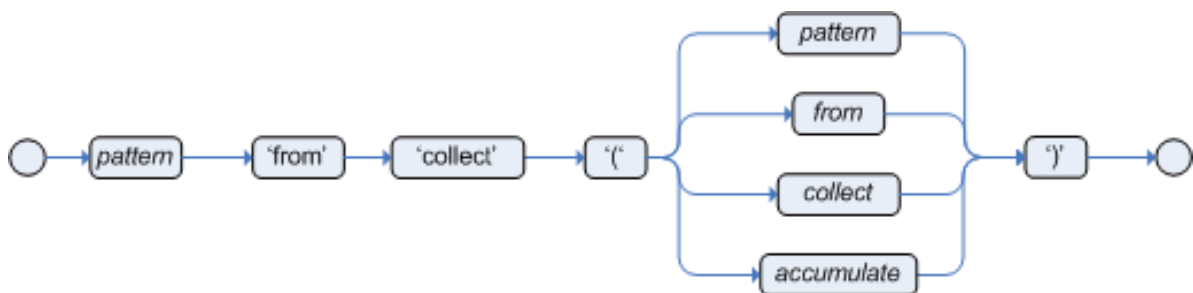


Figure 4.42. collect

Use the **collect** conditional element to make rules "reason" over a collection of objects that have been obtained from either a given source or from the working memory.

Note

In *first-order logic* terms this is known as the *cardinality quantifier*.

```

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 this case, the rule looks in the working memory for any pending alarms for each given system. It then groups them in **ArrayLists**. If it finds three or more alarms for a given system, it fires.

collect's result pattern can be any "concrete" class that implements the `java.util.Collection` interface and provides a default public constructor with no arguments. This means that one can use Java collections like **ArrayList**, **LinkedList**, **HashSet** or even a custom class, as long as it implements the `java.util.Collection` as long as it meets these requirements.



Note

One can constrain both source and result patterns as any other pattern.

Variables bound before the **collect** conditional element are in the scope of both the source and the result patterns. Use them to constrain these patterns. However, note that **collect** is a scope delimiter for bindings, so that any binding made inside of it is not available for use outside of it.

collect can accept nested **from** conditional elements. Hence, 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
```

4.8.3.10. The accumulate Conditional Element

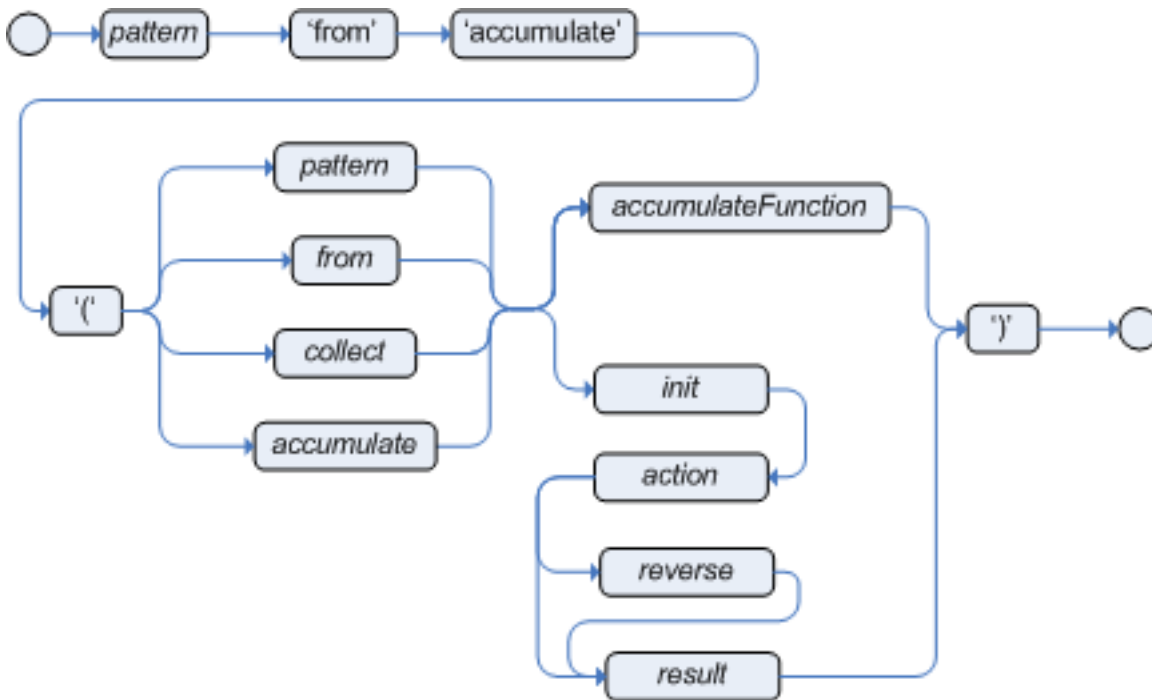


Figure 4.43. accumulate

The **accumulate** conditional element is a more flexible and powerful form of **collect**. It allows a rule to iterate over a collection of objects, executing custom actions for each of the elements. Upon completion, it returns a result object.

This is the general syntax of the **accumulate** conditional element:

```
<result pattern> from accumulate(<source pattern>,
    init( <init code> ),
    action( <action code> ),
    reverse( <reverse code> ),
    result( <result expression> ) )
```

Here is the meaning of each of these elements:

- **<source pattern>**: this is a regular pattern that the engine attempts to match with each of the source objects.
- **<init code>**: this is a semantic block of code in the selected dialect. It is 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 is executed for each of the source objects.
- **<reverse code>**: this is an optional semantic block of code in the selected dialect. If present, it is executed for each source object that no longer matches the source pattern. The objective is to undo any calculation performed in the **<action code>** block, so that the engine can do a decremental calculation when a source object is modified or retracted. This improves performance of these operations quite dramatically.
- **<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 with the object returned from the **<result expression>**. If it matches, the **accumulate** conditional element evaluates it as **true** and the engine proceeds to evaluate the next conditional element in the rule.

If it does not match, the **accumulate** conditional element evaluates it as **false** and the engine stops evaluating conditional elements for that rule.

Here is 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 this case, the following occurs:

1. The engine executes the *init code* for each **order** in the working memory. This initialises the total variable as zero.
2. It then iterates over all of the **OrderItem** objects for that order, executing the action for each one (in this case, it sums the total value of all of the items and puts this into the **total** variable.)
3. It returns the value corresponding with the **result expression** (the value of variable **total**.)

4. The engine tries to match the result with the **Number** pattern, and if the double value is greater than 100, the rule fires.



Important

That example used Java as the semantic dialect. Because of this, note that the usage of the semi-colon as the statement delimiter is mandatory in the **init**, **action** and **reverse** code blocks. The result is an expression and, as such, it does not admit **;**. If using any other dialect, always comply with its specific syntax.



Important

Remember that the **reverse code** is optional, but Red Hat strongly recommends using it in order to benefit from improved performance when using **update** and **retract**.

Use the **accumulate conditional element** to execute any action on source objects. The example in the next section instantiates and populates a custom object.

4.8.3.10.1. Accumulate Functions

The **accumulate conditional element** is very powerful CE, but it is particularly easy to use when utilising those predefined functions known as *accumulate functions*. They work almost exactly like **accumulate** with the difference that, instead of explicitly writing custom code in every **accumulate conditional element**, one can use the predefined code for common operations.

Here is an example. This demonstrates that the rule to apply a discount to orders can be programmed in the following way with **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 this case, **sum** is an **accumulate** function. As its name implies, it sums the **\$value** of every **OrderItem** and returns 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, to calculate the average profit on all of the items in an order, write a rule using the **average** function like this:

```

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

```

Every **accumulate** function is *pluggable*. This means that, if needed, customised, domain-specific functions can be added to the engine quite easily. The rules can then start to use them without any restrictions. To implement a new **accumulate** function all one needs to do is:

1. Create a Java class that implements the `org.drools.base.accumulators.AccumulateFunction` interface.
2. Add a line to the configuration file or set a system property to let the engine know about the new function.

The following example depicts an 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 is very simple because all of integration work is undertaken by the engine.

- To plug the functionality into the engine, add it to the **configuration** file:

```

drools.accumulate.function.average =
    org.drools.base.accumulators.AverageAccumulateFunction

```



Important

Always use the **drools.accumulate.function.** prefix. **average** dictates the way in which function will be used in the rule file, whilst **org.drools.base.accumulators.AverageAccumulateFunction** is the fully-qualified name of the class that implements the behaviour of the function.

4.8.4. The Right-Hand Side

4.8.4.1. Usage

The *right-hand side* is the common name for the *consequence* or action part of the rule. It is here that one places the list of actions that are to be executed.



Important

It is bad practice to use imperative or conditional code on the right-hand side because a rule should be *atomic* in nature ("When this, then do this", rather than "When this, maybe do this.")

The right-hand side of a rule should also be kept small, thus ensuring it remains declarative and readable. If it seems that imperative and/or conditional code is needed on the right-hand side, then consider breaking the rule down into a number of smaller rules.

Use the right-hand side to insert, retract or modify working memory data. That is its purpose. To assist with this, one can take advantage of the following *convenience methods* that modify working memory without the need for one to firstly reference a working memory instance:

- Use `update(object, handle)` to tell the engine that an object (that has been bound to something on the left-hand side) has changed and the rules may, therefore need to be "reconsidered."
- Use `update(object)` to make the *Knowledge Helper* look up the `facthandle` required. It does so by using identity-checking the passed object. (If one is providing the Java beans with property change `listeners`, one is inserting them into the engine, so there is no need to call `update()` when the object changes.)
- Use `insert(new Something())` to place a newly-created object in working memory.
- `insertLogical(new Something())` is similar to `insert`, with the difference that the object is automatically retracted when there are no more facts to support the truth of the currently-firing rule.
- Use `retract(handle)` to removes an object from working memory.

The *convenience methods* are, in fact, just macros that provide short cuts to the Knowledge Helper instance. By doing so, they allow one to access the working memory from the **rules** files.

The pre-defined `KnowledgeHelper` variable allows one to call several other useful methods:

- use `drools.halt()` to terminate rule execution immediately. Do this to return control to the point at which the current session was started with `fireUntilHalt()`.
- the `insert(Object o)`, `update(Object o)` and `retract(Object o)` methods can be called as well. (Due to their frequent use they can be called without the object reference.)
- use `drools.getWorkingMemory()` to return the working memory object.
- use `drools.setFocus(String s)` to set the focus upon the specified agenda group.
- use `drools.getRule().getName()` to return the name of the rule.
- use `drools.getTuple()` to return the *tuple* that matches the currently executing rule. `drools.getActivation()` returns the corresponding activation. (One will find these calls useful during the debugging process.)

The full Knowledge Runtime application programming interface is exposed through another predefined variable, `kcontext`, which is of the type **KnowledgeContext**. Its `getKnowledgeRuntime()` method delivers an object of the type **KnowledgeRuntime**, which, in turn, provides access to numerous methods, many of which are quite useful for coding right-hand side logic.

- use the `kcontext.getKnowledgeRuntime().halt()` call to terminate rule execution immediately.
- use the `getAgenda()` accessor to return a reference to this session's agenda. This, in turn will provide access to the various activation, agenda and rule-flow groups. A relatively common use is the activation of some agenda group, demonstrated here:

```
// give focus to the agenda group Cleanup
kcontext.getKnowledgeRuntime().getAgenda().getAgendaGroup( "Cleanup" ).setFocus();
```



Note

One can achieve the same thing in another way by using `drools.setFocus("Cleanup")`.

- to run a query, call `getQueryResults(String query)`, after which one may process the results in the ways explained in [Section 4.9, "Query"](#).
- there are a set of methods for dealing with event management that lets one add and remove working memory and agenda event listeners.
- use the `getKnowledgeBase()` method to return the **KnowledgeBase** object, which is the "backbone" of the system and, indeed, the originator of the current session.
- manage globals with `setGlobal(...)`, `getGlobal(...)` and `getGlobals()`.
- use `getEnvironment()` to return the run-time's *environment*. (This is much like an operating system's environment.)

4.8.4.2. The modify Statement

This is a language extension that provides a structured approach to undertaking **fact** updates. It combines the update operation with a number of `setter` calls that change the object's fields. Here is the syntactical schema for it:

```
modify ( <fact-expression> ) {
    <expression> [ , <expression> ]*
}
```



Important

The parenthesized **<fact-expression>** must yield a *fact-object reference*. Ensure that the block's expression list consists of `setter` calls for the given object. (These will be written without the usual object reference, which is automatically prep-ended by the compiler.)

Here is a simple example of fact modification in practice:

```
rule "modify stilton"
when
  $stilton : Cheese(type == "stilton")
then
  modify( $stilton ){
    setPrice( 20 ),
    setAge( "overripe" )
  }
end
```

4.9. Query

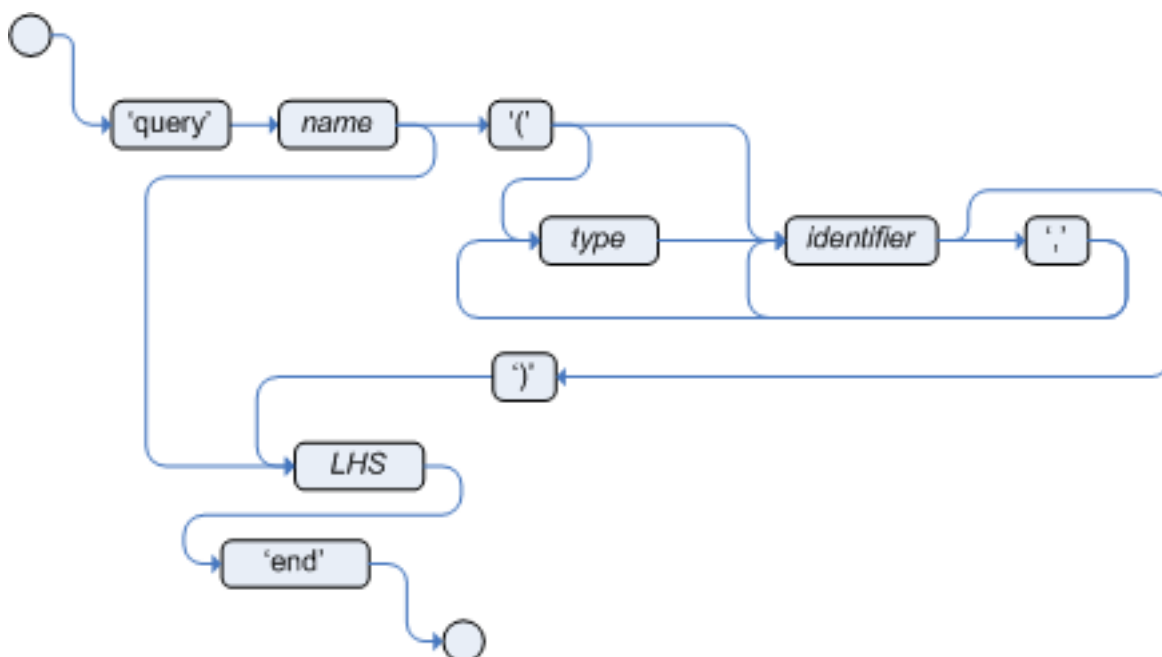


Figure 4.44. query

A query is a simple way to search the working memory for facts that match the stated conditions. Therefore, it contains only the structure of the LHS of a rule, so that you specify neither "when" nor "then". A query has an optional set of parameters, each of which can also be optionally typed. If the type is not given then the type Object is assumed. The engine will attempt to coerce the values as needed. Query names are global to the **KnowledgeBase**, so do not add queries of the same name to different knowledge packages for the same **RuleBase**.

To return the results use `ksession.getQueryResults("name")`, where "name" is the query's name. This returns a list of query results, which allow you to retrieve the objects that matched the query.

The first example is a simple query for all the people over the age of 30. The second one, using parameters, combines the age limit with a location.

Example 4.19. Query People over the age of 30

```
query "people over the age of 30"
  person : Person( age > 30 )
end
```

Example 4.20. Query People over the age of x, and who live in y

```
query "people over the age of x" (int x, String y)
  person : Person( age > x, location == y )
end
```

We iterate over the returned `QueryResults` using a standard for loop. Each element is a **QueryResultsRow** which we can use to access each of the columns in the tuple. These columns can be accessed by bound declaration name or index position.

Example 4.21. Query People over the age of 30

```
QueryResults results = ksession.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 ( QueryResultsRow row : results ) {
    Person person = ( Person ) row.get( "person" );
    System.out.println( person.getName() + "\n" );
}
```

4.10. Domain-Specific Languages

Domain Specific Languages (or DSLs) are a way of creating a rule language that is dedicated to your problem domain. A set of DSL definitions consists of transformations from DSL "sentences" to DRL constructs, which lets you use of all the underlying rule language and engine features. Given a DSL, you write rules in DSL rule (or DSLR) files, which will be translated into DRL files.

DSL and DSLR files are plain text files, and can be created or modified with a text editor. DSL and DSLRs can be used with both integrated development environments and the **Business Rules Management System** web user interface.

4.10.1. When to Use a Domain-Specific Language

Domain-specific languages provide the following advantages:

- They can serve as a layer of separation between rule authoring and the domain objects upon which the engine operates. This is useful if rules need to be read and validated by domain experts (such as business analysts) who are not programmers. DSLs hide implementation details and focuses on the rule logic.
- DSL sentences can also act as templates for conditional elements and consequence actions that are used repeatedly in rules.
- DSLs have no impact on the rule engine at runtime, they are a compile time feature, requiring a special parser and transformer.

4.10.2. Creating a Domain-Specific Language

Consider the following points when starting to develop a Domain-Specific Language:

- Technical and domain experts need to collaborate to create a domain-specific language.

- Initially write representative samples of the rules the application requires to gain an idea of their size and complexity.
- Identify similar and recurring statements in the rules and mark the variable parts as parameters.
- Test the rules as the language is developed.
- Writing rules is generally easier if the application's data model represents the data types as facts.
- Implementation decisions concerning conditions and actions may be postponed during the initial design phase by leaving conditional elements and actions in their DRL form by prefixing a line with a greater than sign (>). (This is also useful for inserting debugging statements.)
- Rules can be written by reusing the existing DSL definitions, or by adding a parameter to an existing condition or consequence entry
- Try to keep the number of DSL entries small. Using parameters lets you apply the same DSL sentence for similar rule patterns or constraints.

4.10.3. Managing a Domain-Specific Language

A domain-specific language's configuration is stored in a plain text file.

The DSL mechanism allows you to customize conditional expressions and consequence actions. A global substitution mechanism *keyword* is also available.

```
[when]Something is {color}=Something(color=="{color}")
```

The following applies to the previous example:

- The **[when]** keyword indicates the scope of the expression, i.e., whether it is valid for the LHS or the RHS of a rule.
- The part after the bracketed keyword is the expression to use in the rule; typically a natural language expression, but it doesn't have to be.
- The part to the right of the first equal sign = is the mapping of the expression into the rule language. The form of this string depends on its destination, RHS or LHS. If it is for the LHS, then it should be a term according to the regular LHS syntax; if it is for the RHS then it might be a Java statement.

Whenever the DSL parser matches a line from the rule file written in the DSL with an expression in the DSL definition, it performs three steps of string manipulation. First, it extracts the string values appearing where the expression contains variable names in braces (for instance, **{color}**). Then, the values obtained from these captures are interpolated wherever that name, again enclosed in braces, occurs on the right hand side of the mapping. Finally, the interpolated string replaces whatever was matched by the entire expression in the line of the DSL rule file.

Note that the expressions (i.e., the strings on the left hand side of the equal sign) are used as regular expressions in a pattern matching operation against a line of the DSL rule file, matching all or part of a line. This means you can use a **?** to indicate that the preceding character is optional. This helps to overcome variations in the natural language phrases of your DSL. however, as these expressions are regular expression patterns, this also means that all *magic* characters of Java's pattern syntax have to be escaped with a preceding backslash (\).



NOTE

It is important to note that the compiler transforms DSL rule files line by line. In the above example, all the text after "Something is " to the end of the line is captured as the replacement value for "{colour}", and this is used for interpolating the target string.

To merge different DSL expressions to generate a composite DRL pattern, it is necessary to transform a DSLR line in several independent operations. Do this by ensuring the captures are surrounded by characteristic text - words or even single characters. The matching operation performed by the parser extracts a substring from the line. In the example below, quotes are used as distinctive characters. The characters used to surround the capture are not included during interpolation, just the content between the characters.

Use quotes for textual data that a rule editor may want to enter. It is also possible to enclose the capture with words to ensure that the text is correctly matched.

For instance:

```
[when]This is "{something}" and "{another}"=Something(something=="{something}",
another=="{another}")
[when]This is {also} valid=Another(something=="{also}")
```

Avoid using punctuation (other than quotes) in the domain-specific language expressions. Punctuation is easily forgotten by rule authors using the DSL, and some punctuation marks (parentheses, period, and question mark) requiring escaping in the DSL definition.

In a DSL mapping, the curly braces { and } should only be used to enclose a variable definition or reference, resulting in a capture. If they should occur literally, either in the expression or within the replacement text on the right hand side, they must be escaped with a preceding backslash ("\"):

```
[then]do something= if (foo) \{ doSomething(); \}
```



NOTE

If curly braces { and } should appear in the replacement string of a DSL definition, escape them with a backslash (\).

When capturing plain text from a DSL rule line and want to use it as a string literal in the expansion, you must provide the quotes on the right hand side of the mapping.

Example 4.22. DSL Mapping Entries

```
#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
```

Given the [Example 4.22, "DSL Mapping Entries"](#), the following examples show the expansion of various DSLR snippets:

Example 4.23. DSL Mapping Entries Expansions

```

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")

```

4.10.4. Adding Constraints to Facts

A common requirement when writing rule conditions is to be able to add an arbitrary combination of constraints to a pattern. Given that a fact type may have many fields, having to provide an individual DSL statement for each combination could be extremely difficult.

The DSL facility allows constraints to be added to a pattern by adding a hyphen - to the beginning of the DSL expression. If the expression starts with a hyphen it is assumed to be a field constraint and is added to the last pattern line preceding it.

For example, with the **Cheese** class, which has the following fields: type, price, age, and country, it is possible to express some left-hand side conditions in a normal **DRL** file as follows:

Example 4.24. LHS Conditions in a DRL

```
Cheese(age < 5, price == 20, type=="stilton", country=="ch")
```

The DSL definitions given in [Example 4.25, "Adding Constraints"](#) result in three DSL phrases which may be used to create any combination of constraint involving these fields.

Example 4.25. Adding Constraints

```

[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}'

```

You can then write rules with conditions as follows:

Example 4.26. Writing Constraints

```

There is a Cheese with
  - age is less than 42
  - type is 'stilton'

```

The **parser** will pick up a line beginning with - and add it as a constraint to the preceding pattern, inserting a comma when it is required. For [Example 4.26, "Writing Constraints"](#) example, the resulting DRL is:

```
Cheese(age<42, type=='stilton')
```

Combining all all numeric fields with all relational operators (according to the DSL expression "age is less than..." in the preceding example) produces a lot of DSL entries. DSL Phrases can be defined for the various operators and even a generic expression that handles any field constraint, as shown below. (Notice that the expression definition contains a regular expression in addition to the variable name.)

```
[when][]is less than or equal to<=  
[when][]is less than=<  
[when][]is greater than or equal to>=  
[when][]is greater than=>  
[when][]is equal to===  
[when][]equals===  
[when][]There is a Cheese with=Cheese()
```

These definitions mean it is possible to write conditions textual (i.e., is less than).



Note

The order of the entries in the DSL is important if separate DSL expressions are intended to match the same line, one after the other.



Note

To make a filtered list of field constraints appear with the **Context Assistant**, press - followed by **Ctrl+Space**, and then choose an item from this list.)

Alter the domain-specific language code for the first item to read: **[when][org.drools.Cheese]-age is less than {age}**. Do the same to all of the other items in the example above.

The extra **[org.drools.Cheese]** code indicates that the sentence only applies to the main constraint directly above it (which, in this case reads **There is a Cheese with**.)

For example, if there is a class called **Cheese** and a constraint is being added via the **Content Assistance** approach, then only those items marked with an object-scope of **com.yourcompany.Something** are valid, so only they will appear in the list. This is entirely optional.

4.10.5. DSL and DSLR Reference

A DSL file is a text file in a line-oriented format. Its entries are used for transforming a DSLR file into a file with DRL syntax.

- A line starting with # or // (with or without preceding white space) is treated as a comment. A comment line starting with #/ is scanned for words requesting a debug option, see below.
- Any line starting with an opening square bracket [is assumed to be the first line of a DSL entry definition.
- Any other line is appended to the preceding DSL entry definition, with the line end replaced by a space.

A DSL entry consists of the following four parts:

- A scope definition:

- **[condition]** or **[when]**
- **[consequence]** or **[then]**
- **[keyword]**, for instance **rule** or **end**.

The keyword indicates the scope of the entry, whether it has global significance, i.e., it is recognized anywhere in a DSLR file.

- A type definition, written as a Java class name, enclosed in brackets. This part is optional unless the the next part begins with an opening bracket. An empty pair of brackets is also valid.
- A DSL expression consists of a (Java) regular expression, with any number of embedded *variable definitions*, terminated by an equal sign =. A variable definition is enclosed in curly braces { and }. It consists of a variable name and two optional attachments, separated by colons :. If there is one attachment, it is a regular expression for matching text that is to be assigned to the variable; if there are two attachments, the first one is a hint for the GUI editor and the second one the regular expression.

Note that all characters that are "magic" in regular expressions must be escaped with a preceding backslash \ if they should occur literally within the expression.

- The remaining part of the line after the delimiting equal sign is the replacement text for any DSLR text matching the regular expression. It may contain variable references, i.e., a variable name enclosed in curly braces. Optionally, the variable name may be followed by an exclamation mark ! and a transformation function, see below.

Note that curly braces { and } must be escaped with a preceding backslash \ if they should occur literally within the replacement string.

Debugging of DSL expansion can be turned on, selectively, by using a comment line starting with #/ which may contain one or more words from the table presented below. The resulting output is written to standard output.

Table 4.2. Debug options for DSL expansion

| Word | Description |
|---------|---|
| result | Prints the resulting DRL text, with line numbers. |
| steps | Prints each expansion step of condition and consequence lines. |
| keyword | Dumps the internal representation of all DSL entries with scope keyword . |
| when | Dumps the internal representation of all DSL entries with scope when or * . |
| then | Dumps the internal representation of all DSL entries with scope then or * . |
| usage | Displays a usage statistic of all DSL entries. |

Below are some sample DSL definitions, with comments describing the language features they illustrate.

```
# Comment: DSL examples
```

```
#!/ debug: display result and usage

# keyword definition: replaces "regula" by "rule"
[keyword][]regula=rule

# conditional element: "T" or "t", "a" or "an", convert matched word
[when][][Tt]here is an? {entity:\w+}=
    ${entity!lc}: ${entity!ucfirst} ()

# consequence statement: convert matched word, literal braces
[then][][update {entity:\w+}=modify( ${entity!lc} )\{ \}
```

4.10.6. The Transformation of a DSLR File

The transformation of a DSLR file proceeds as follows:

1. The text is read into memory.
2. Each of the **keyword** entries is applied to the entire text. First, the regular expression from the keyword definition is modified by replacing white space sequences with a pattern matching any number of white space characters, and by replacing variable definitions with a capture made from the regular expression provided with the definition, or with the default ". *?". Then, the DSLR text is searched exhaustively for occurrences of strings matching the modified regular expression. Substrings of a matching string corresponding to variable captures are extracted and replace variable references in the corresponding replacement text, and this text replaces the matching string in the DSLR text.
3. Sections of the DSLR text between **when** and **then**, and **then** and **end**, respectively, are located and processed in a uniform manner, line by line, as described below.

For a line, each DSL entry pertaining to the line's section is taken in turn, in the order it appears in the DSL file. Its regular expression part is modified: white space is replaced by a pattern matching any number of white space characters; variable definitions with a regular expression are replaced by a capture with this regular expression, its default being ". *?". If the resulting regular expression matches all or part of the line, the matched part is replaced by the suitably modified replacement text.

Modification of the replacement text is done by replacing variable references with the text corresponding to the regular expression capture. This text may be modified according to the string transformation function given in the variable reference; see below for details.

If there is a variable reference naming a variable that is not defined in the same entry, the expander substitutes a value bound to a variable of that name, provided it was defined in one of the preceding lines of the current rule.

4. If a DSLR line in a condition is written with a leading hyphen, the expanded result is inserted into the last line, which should contain a pattern CE, i.e., a type name followed by a pair of parentheses. If this pair is empty, the expanded line (which should contain a valid constraint) is simply inserted, otherwise a comma , is inserted beforehand.

If a DSLR line in a consequence is written with a leading hyphen, the expanded result is inserted into the last line, which should contain a **modify** statement, ending in a pair of curly braces { and }. If this pair is empty, the expanded line (which should contain a valid method call) is simply inserted, otherwise a comma , is inserted beforehand.



Note

It is currently *not* possible to use a line with a leading hyphen to insert text into other conditional element forms (e.g., **accumulate**) or it may only work for the first insertion (e.g., **eval**).

4.10.7. String Transformation Functions

All string transformation functions are described in [Table 4.3, “String transformation functions”](#).

Table 4.3. String transformation functions

| Name | Description |
|---------|--|
| uc | Converts all letters to upper case. |
| lc | Converts all letters to lower case. |
| ucfirst | Converts the first letter to upper case, and all other letters to lower case. |
| num | Extracts all digits and - from the string. If the last two digits in the original string are preceded by . or , , a decimal period is inserted in the corresponding position. |
| a?b/c | Compares the string with string a, and if they are equal, replaces it with b, otherwise with c. But c can be another triplet a, b, c, so that the entire structure is, in fact, a translation table. |

The following DSL examples show how to use string transformation functions.

Example 4.27. DSL String Transformation Functions

```
# definitions for conditions
[when][ ]There is an? {entity}=${entity!lc}: {entity!ucfirst}()
[when][ ]- with an? {attr} greater than {amount}={attr} <= {amount!num}
[when][ ]- with a {what} {attr}={attr} {what!positive?>0/negative?%lt;0/zero?==0/ERROR}
```

A file containing a DSL definition is customarily given the extension **.dsl**. It is passed to the Knowledge Builder with **ResourceType.DSL**. For a file using DSL definition, the extension **.dslr** should be used. The Knowledge Builder expects **ResourceType.DSLR**. The IDE, however, relies on file extensions to correctly recognize and work with your rules file.

The DSL must be passed to the Knowledge Builder ahead of any rules file using the DSL.

```
KnowledgeBuilder kBuilder = new KnowledgeBuilder();
Resource dsl = ResourceFactory.newClassPathResource( dslPath, getClass() );
kBuilder.add( dsl, ResourceType.DSL );
Resource dslr = ResourceFactory.newClassPathResource( dslrPath, getClass() );
kBuilder.add( dslr, ResourceType.DSLR );
```

For parsing and expanding a DSLR file the DSL configuration is read and supplied to the parser. The parser can recognize the DSL expressions and transform them into native rule language expressions.

4.10.8. Domain-Specific Languages in the BRMS and in the IDE

If you are using the **Guided Editor** to develop rules, domain-specific languages can still be used.

**Important**

Keep them as simple as possible because the **Guided Editor** cannot handle some complex expressions.

The **Guided Editor** allows you to define little data-capture text field "forms." (i.e., upon picking a domain-specific language expression, it will add an item to the GUI which only allows you to enter data to the `{token}`).

The domain-specific languages will be included automatically when a package is built in the BRMS.

To include domain-specific languages in the integrated development environment, use the **drools-ant** task, or alternatively, incorporate the code shown in [Section 4.11, "XML Rule Language"](#).

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>
    </not>
  </lhs>
</rule>

```

```

    <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.

Example 4.28. Detail of rule element

```
<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>
```

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.

Using Spreadsheet Decision Tables

Read this chapter to learn about the ways in which *decision tables* can be used.

Decision tables are a way of representing *conditional logic*, and are well-suited to the task of depicting business-level rules.

JBoss Rules lets you manage rules by storing them in a spreadsheet format, such as **CSV** or **.XLS**.

JBoss Rules uses decision tables to generate rules derived the data entered into the spreadsheet. One can take advantage of all the usual data capture and manipulation features of a spreadsheet to build these data sets.

5.1. When Should Decision Tables be Used?

Consider using decision tables if there are rules that can be expressed as templates and data. In each row of a decision table, data is collected. It is then combined with the templates to generate a rule.

Do not use decision tables if the rules in question do not follow a set of templates, or where there are a small number of rules. It also comes down to personal preference: some users simply prefer using spreadsheet applications and some do not.

Decision tables also insulate the user underlying object model, which may or may not be preferable.

5.2. Overview

Here are some example decision tables:

The screenshot shows an Excel spreadsheet with the following data:

| | B | C | D | E |
|----|--------------------------------------|----------------------|-----------------|---------|
| 16 | Type of New Claim | Is case catastrophic | Allocation code | Claim 1 |
| 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 5.1. Using Excel to Edit a Decision Table

| | | | |
|-----|------------------|-----------------|----------------------------|
| | J | K | L |
| mer | Allocate to Team | Stop processing | Log reason |
| | Team Red | Stop processing | The claim was catastrophic |

Figure 5.2. Multiple Actions for a Rule Row

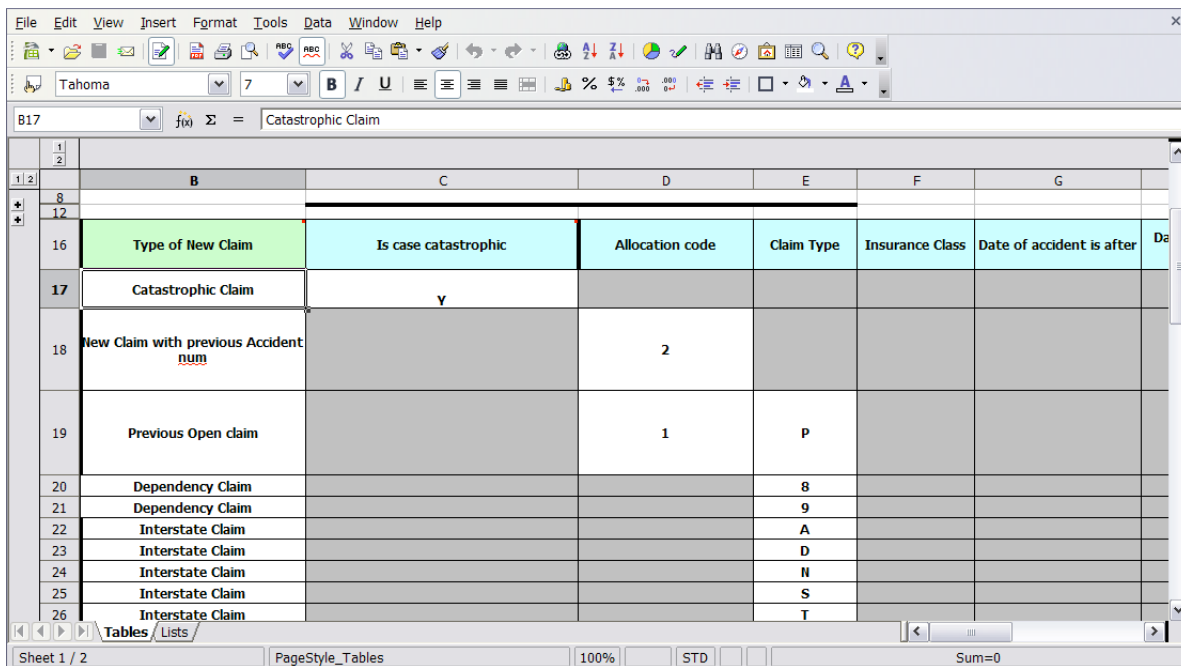


Figure 5.3. Using OpenOffice.org Calc



Note

In the above examples, the technical aspects of the decision table have been collapsed (a standard spreadsheet feature).

The rules start from row seventeen. (Each row results in a rule.) The conditions are in column C, D, E and so forth. (The actions are off-screen.) As can be seen, the values in the cells are quite simple, and have meaning when one observes the headers in row sixteen. (Column B is just a description.)



Note

It can sometimes be helpful to use colour to indicate the meanings of different areas of the table.

Important

Although the decision tables look like they process the data from the top down, this is not necessarily the case. It is best practice to write rules in such a way that order does not matter (simply because it will make maintenance easier and eliminate the need to constantly shift rows around.)

Each row is a rule and so, hence, the same principles apply. As the rule engine processes the facts, any rules that match will "fire." Users are sometimes confused by this; it is possible to clear the agenda when a rule fires and simulate a very simple decision table at the point where the first match exists. Decision tables are simply a tool to generate DRL packages automatically.

Note

You can have multiple tables on the one spreadsheet. This is helpful because rules can be grouped when they share common templates, yet still ultimately be combined into a single rule package.)

| 1 | 2 | 3 | 4 | 5 | 6 |
|---|------------------|-------------------------------|---------------------------|-------------------------------------|--|
| Module | PRSC[02] | | | | |
| RuleSet | Control Cajas[1] | | | | |
| 1.ValidarAperturaCaja (Caja, Registro Estado Sucursal,Transaccion) | | | | | |
| ID_Caso de Uso | Caso de Uso | Identificadores de las Reglas | Prioridades de las Reglas | Nombres de las Reglas | Descripciones |
| | | 1 | 2000 | ValidarAperturaCajaSucursal Abierta | Esta Regla tiene por Mision Validar que la sucursal de la se encuentre abierta Trabaja sobre la Caja que se intenta abrir, la Sucursal corresponde a esa caja y la Transacción de Caja apertura |
| | | 2 | 2000 | ValidarAperturaCajaMismaFecha | Esta Regla tiene por Mision Validar que en la sucursal caja se encuentre abierta para la misma fecha de apertura de la caja. Trabaja sobre la Caja que se intenta abrir, la Sucursal corresponde a esa caja y la Transacción de Caja apertura |
| 2.ValidarCierreCajasSucursal(Registro Estado Sucursal, TransaccionCaja) | | | | | |
| ID_Caso de Uso | Caso de Uso | Identificadores de las Reglas | Prioridades de las Reglas | Nombres de las Reglas | Descripciones |
| C_PRSC_503 C_PRSC_504 C_PRSC_513 | | 1 | 1000 | ValidarCierreCajasSucursal | Esta Regla tiene por Misión Validar que al momento efectuarse el Cierre Contable de una Sucursal de FOL todas las Cajas de esta última se encuentren en E Cerrado, es decir la Fecha de Cierre de Caja debe ser a la Fecha de cierre de la entidad Registro_Cierre_Suc |
| 3.ValidarTransaccionCaja(Caja, Transaccion_Caja) | | | | | |
| RuleTable[3] ValidarTransaccionCaja(CajaVO caja, MovimientoCajaVO movimientoCaja) | | | | | |
| ID_Caso de Uso | Caso de Uso | Identificador | Prioridad | Nombre | Descripcion |

Figure 5.4. A Real-Life Example Using Multiple Tables to Group Like Rules

5.3. How Decision Tables Work

Important

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 | Each column may be a condition, or action etc. | Insurance Class | Date of accident is after |
| 17 | Catastrophic Claim | y | | | | |
| 18 | New Claim with previous Accident num | | 2 | | | |
| Each row results in a rule | | | | | | |
| 20 | Dependency Claim | | | | | |
| 21 | Dependency Claim | | | | | |
| 22 | Interstate Claim | | | | | |
| 23 | Interstate Claim | | | | | |
| 24 | Interstate Claim | | | | | |
| 25 | Interstate Claim | | | | | |

Figure 5.5. Rows and Columns

The spreadsheet looks for the **RuleTable** keyword to indicate the starting row and column of a rule table. (Other keywords used to define other package level attributes are covered later in this chapter.) 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 (it is also a 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.



Note

Expand the hidden sections so that more can be seen if this helps one to understand it.

Note the keywords in Column "C."

| | B | C | D | E |
|----|----------------|------------------------------|---|-------------------|
| 7 | | | | |
| 8 | | | | |
| 9 | | RuleSet | Some business rules | |
| 10 | | import | org.drools.decisiontable.Cheese, org.drools.dec | |
| 11 | | Sequential | true | |
| 12 | | | | |
| 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 |
| 20 | | | | |
| 21 | | Variables | java.util.List list | |
| 22 | | | | |
| 23 | | | | |

Figure 5.6. Expanded for Rule Templates

The **RuleSet** keyword indicates the name to be used in the `rule` package under which all of the rules are to be grouped. (Not that the name is optional. It will have a default but the **RuleSet** keyword must be present.) The other keywords visible in Column C are **Import** and **Sequential**, which will be covered later in this chapter. At this stage, just note that, in general, the keywords make up name/value pairs.

The **RuleTable** keyword is important as it indicates that a group of rules will follow, and that these will be based on some rule templates.

After the **RuleTable** keyword there is a name. This is used as a prefix of the rules names that are generated. (The row numbers are appended to create unique rule names.) The **RuleTable** column indicates the column in which the rules start (the columns to the left of it are ignored.)

The **CONDITION** and **ACTION** keywords in Row 14 indicate that the data in the columns below is either for the LHS or the RHS part of a rule. (There are other attributes that can also be optionally set in this way.)

Row 15 contains declarations of `ObjectTypes`. The content in this row is optional, leave a blank row if you do not want to use it. When this row is used, the values in the cells below (in Row 16) become constraints on that object type. In the above case, it will generate: **Person(age=="42")** (where **42** comes from Row 18). In the above example, the `==` is implicit, if you just put a field name, it will assume that you are looking for exact matches.

**Note**

It is possible to make the `ObjectType` declaration span columns (by merging cells). This results in all of those columns below the merged range being combined into a single set of constraints.

Row 16 contains the rule templates themselves. They can use the **\$para** place holder to indicate where data from the cells will be populated. Use **\$param**, or **\$1**, **\$2** and so forth to indicate parameters from a comma-separated list located in a cell below.)

Row 17 is ignored; it contains a textual description of the rule template.

Row 18 to 19 show data, which will be combined (interpolated) with the templates in Row 15, to generate the actual rules. If a cell contains no data, then its template is ignored. Rule rows are read until a blank row is encountered. (One can have multiple `RuleTables` in a sheet.)

Row 20 contains another keyword and a value. (Remember that the row positions of keywords like this do not matter but it is best practice to put them at the top. However, their column should be the same one as that in which the **RuleTable** or **RuleSet** keywords appear (in this case column C has been chosen but one can use Column A if this is preferred.)

In the above example, rules will be rendered like this (as the `ObjectType` row is being used):

```
//row 18
rule "Cheese_fans_18"
  when
    Person(age=="42")
    Cheese(type=="stilton")
  then
    list.add("Old man stilton");
  end
```

Note that `[age=="42"]` and `[type=="stilton"]` are interpreted as single constraints to be added to the respective `ObjectTypes` in the cell above (if the cells above were spanned, then there would be multiple constraints on one "column".)

5.4. Keywords and Syntax

5.4.1. Template Syntax

The syntax used is slightly differs between the **CONDITION** column and **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 intended for execution.

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, the symbols **\$1**, **\$2**, etc. may be used to indicate which positional parameter from the list of values in the cell will be used. The **forall(DELIMITER) {SNIPPET}** function can be used to loop over all available comma separated values.

Here is an example:

```
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 [42,43] then the result will be [Foo(bar > 42, baz ==43)]

For conditionals, snippets are rendered dependent on the presence or absence of `ObjectType` declarations in the row above. If they are present, the snippets are rendered as individual constraints on that `ObjectType`. If there are not any, they are simply rendered as is (with values substituted.) If a plain field (as in the example above) is entered, it will assume this means equality. If another operator is placed at the end of the snippet, the values will be interpolated at the end of the constraint, otherwise it will look for **\$param** as outlined previously.

For consequences, snippets are rendered dependent on the presence or absence of anything in the row immediately above it. If there is no entry, the output is simply the interpolated snippets. If there is something there, such as a bound variable or a global (like in the example above), then it will be appended as a method call on that object.

Here are some more examples:

| | | |
|----|------------------------------|--------------------|
| 13 | RuleTable Cheese fans | |
| 14 | CONDITION | CONDITION |
| 15 | Person | |
| 16 | age | type |
| 17 | Persons age | Cheese type |
| 18 | 42 | stilton |
| 19 | 21 | cheddar |

Figure 5.7. Spanned Column

The example above shows how the **Person** `ObjectType` declaration spans two spreadsheet columns. Thus, both constraints will appear as **Person(age == ... , type == ...)**. As before, only the field names are present in the snippet, implying an equality test.

| |
|------------------|
| CONDITION |
| Person |
| age == "\$param" |
| Persons age |
| 42 |

Figure 5.8. With Parameters

In this example, interpolation is used to place the values in the snippet (the result being **Person(age == "42")**.)

| |
|-------------|
| CONDITION |
| Person |
| age < |
| Persons age |
| 42 |

Figure 5.9. Operator Completion

The conditional example above demonstrates that if an operator is put on the end by itself, the values will be placed after the operator automatically.

| |
|--------------------|
| CONDITION |
| c: Cheese |
| |
| type |
| Cheese type |
| |
| stilton |

Figure 5.10. With Binding

It is possible to put a binding in before the column (the constraints will be added from the cells below.) Anything can be placed in the `ObjectType` row, an example being a pre-condition for the columns that follow.)

| |
|------------------------|
| ACTION |
| |
| |
| list.add("\$param"); |
| Log |
| |
| Old man stilton |

Figure 5.11. Consequence

This final example shows how the consequence can be achieved by simple interpolation, just by leaving the cell above blank (the same applies to condition columns.) Using this method, anything can put in the consequence, not just one method call.

5.4.2. Keywords

The following table describes the keywords that are necessary for the rule table structure.

Table 5.1. Keywords

| Keyword | Description | Inclusion Status |
|---------|--|--|
| RuleSet | The cell to the right of this contains the ruleset name. | One only (if left out, it will default). |

| Keyword | Description | Inclusion Status |
|-------------------|---|---|
| Sequential | The cell to the right of this can be true or false . If true, then salience is used to ensure the 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 | RuleTable indicates the start of a rule table definition. (The actual rule table starts on the next row down.) The rule table is read from left to right, top to bottom, until the next blank row is encountered. | 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 column's values will set the 'salience' values for the rule row. Overrides the 'Sequential' flag. | Optional |
| DURATION | Indicates that this column's values will set the duration values for the rule row. | Optional |
| NAME | Indicates that this column's 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. JBoss Rules 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 JBoss Rules supports. This is a type, followed by a variable name. If multiple variables are needed, separate them with commas. | 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 | Optional |

| Keyword | Description | Inclusion Status |
|----------------|---|------------------|
| | 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. | |
| XOR-GROUP | Cell values in this column mean that the rule row belongs to the given Activation group . An Activation group means that only one rule in the named group will fire (i.e., 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. (This is one way of controlling flow between groups of rules - see also "rule flow"). | Optional |
| 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 |

Here are some use-cases for the HEADER keyword, which affects the rules generated for each row. Note that the header name itself is the most significant thing in most cases. If no value appears in the cells below it, the attribute will not be applied to 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 | licenceYears | priorClaims | | | | |
| 10 | Persons age | Prior Claims | | | insert(new Approver("\$param")); | System.out.println("Spa |
| 11 | guy | 30 | 1 | risk assessment | Safe and mature | Log |
| 12 | | | | | | Old driver Approved |
| 13 | | | | | | |
| 14 | | | | | | |
| 15 | | | | | | |
| 16 | | | | | | |

Figure 5.12. Example Usage of Keywords

This example demonstrates the following keywords: `Import` (which is comma-delimited), `Variables` (which is a global and also comma-delimited) and `function block` (which can be comprised of multiple functions and uses the normal DRL syntax. It can appear in the same column as the `RuleSet` keyword or be below all of the rule rows.)

| | |
|------------------|---|
| RuleSet | Control Cajas[1] |
| Import | foo.Bar, bar.Baz |
| Variables | Parameters parametros, RulesResult resultado, EvalDate fecha |
| Functions | <pre>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; }</pre> |

Figure 5.13. Example Usage of Keywords for Functions

5.5. Creating and Integrating Spreadsheet Based Decision Tables

Find the application programming interface used in conjunction with spreadsheet based decision tables in the `drools-decisiontables` module. Only one class is of relevance, this being **SpreadsheetCompiler**. This class takes spreadsheets in various formats and generates DRL rules which can then be used in the normal way.

The **SpreadsheetCompiler** can also be used to generate partial rule files which can later be assembled into a complete rule package. (Use this to separate the technical and non-technical aspects of the rules.)

Base them on a sample spreadsheet or, if using the **Rule Workbench IDE** plug-in, utilise its in-built Wizard to generate a spreadsheet from a template and then edit it with an **XLS**- compatible spreadsheet application.

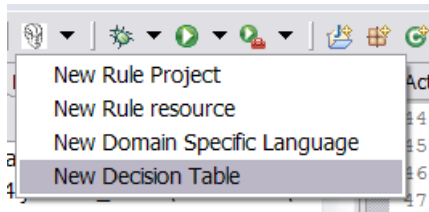


Figure 5.14. Using the Integrated Development Environment

5.6. Managing Business Rules in Decision Tables

5.6.1. Workflow and Collaboration

Decision tables are ideal for situations where there is a need for close collaboration between IT and domain experts, using decision tables keeps the business rules clear for analysts.

To create business rules, follow this process:

1. The business analyst obtains a decision table template (from a repository or from IT staff).
2. The business analyst enters decision table business language descriptions into the template.
3. Decision table rules (rows) are entered (as a rough draft)
4. The decision table is handed to a programmer, who maps the business language (descriptions) to scripts (this may involve software development, if it is a new application or data model)
5. The programmer reviews the modifications with the business analyst.
6. The business analyst can continue editing the rule rows as needed (moving columns, etc).
7. The programmer can develop test cases for the rules to be used to verify rule changes once the system is running.

5.6.2. Using Spreadsheet Features

Use LibreOffice's **Calc** features to assist in entering spreadsheet data. Lists stored in other worksheets can be used to provide valid lists of values for cells.

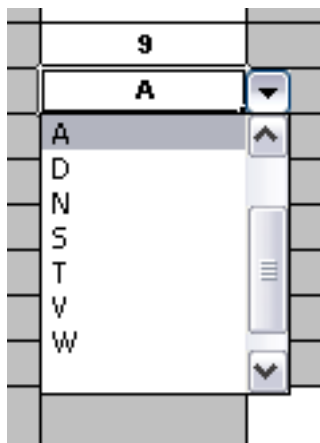


Figure 5.15. Using Worksheet Lists



Important

Red Hat recommends using a version control system to maintain a history of the changes.

The Java Rule Engine Application Programming Interface

6.1. Introduction

JBoss Rules provides an implementation of *JSR94*, the Java rule engine *application programming interface* (API.) Multiple rule engines can be run with this single API. Read this chapter to learn more about the capabilities of this API.



Note

JSR94 does not, in any way, interact with the rule language itself. It is important to remember that the JSR94 standard represents the "lowest common denominator" in terms of features across rule engines. This means that there is less functionality in the JSR94 API than can be found in the standard **JBoss Rules** API. Hence, by using JSR94, one will forfeit some of the capabilities granted by **JBoss Rules'** rule engine.

To access fuller functionality (including globals and **DRL**, **DSL** and **XML** files), use property maps. Note that, by doing this, non-portable functionality is, of course, introduced. Furthermore, as JSR94 does not provide a rule language, one is only reducing complexity by a small fraction when switching rule engines. There is very little to gain from the move. Therefore, whilst Red Hat provides support for JSR94 if one insists upon using it, programmers are strongly recommended to use the **JBoss Rules** API instead.

6.2. How To Use the API

JSR94 consists of two parts. The first of these is the Administrative API, which is used to build and register `RuleExecutionSets`. The second part is the run-time session, used to execute those same `RuleExecutionSets`.

6.2.1. Building and Registering `RuleExecutionSets`

The `RuleServiceProviderManager` manages the registration and retrieval of `RuleExecutionSets`. The **JBoss Rules `RuleServiceProvider`** implementation is automatically registered via a *static block* when the class is loaded using `Class.forName`. (This occurs in much the same way as it does for JDBC drivers.)

Example 6.1. Automatic `RuleServiceProvider` Registration

```
// 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/");
```

The `RuleServiceProvider` provides access to the `RuleRuntime` and `RuleAdministration` APIs. The `RuleAdministration` provides an administration API for the management of

RuleExecutionSets. This makes it possible to register a RuleExecutionSet that can then be retrieved via the RuleRuntime.

To create and register a RuleExecutionSet, follow these steps:

1. Create a RuleExecutionSet; the RuleAdministrator provides factory methods to return either an empty **LocalRuleExecutionSetProvider** or **RuleExecutionSetProvider**.
2. Use the **LocalRuleExecutionSetProvider** to load a RuleExecutionSet from a local, non-serialisable source, such as a stream.

The **RuleExecutionSetProvider** can be used to load RuleExecutionSets from serializable sources, like DOM elements or knowledge packages.



Note

The `ruleAdministrator.getLocalRuleExecutionSetProvider(null);` and `ruleAdministrator.getRuleExecutionSetProvider(null);` methods both accept **null** as a parameter. This is because the `properties` map for these methods is not currently being used.

Example 6.2. Registering a LocalRuleExecutionSet with the RuleAdministrator API

```
// 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 );
```

In the example above, the `ruleExecutionSetProvider.createRuleExecutionSet(reader, null)` takes a null parameter for the `properties` map; (however, it can actually be used to provide configuration information for the incoming source.) When **null** is passed, the default is used to load the input from a **DRL** file. The keys which one is allowed to use for a map are `source` and `dsl`. `source` takes **dr1** or **xml** as its value. Simply set `source` to **dr1** to load a **DRL** file and, likewise, set it to **xml** to load an **XML** file. (**xml** will ignore any **dsl** key/value settings.) The `dsl` key can use either a reader or a string (the contents of the domain-specific language) as a value.

Example 6.3. Specifying a Domain-Specific Language When Registering a LocalRuleExecutionSet

```
// 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 );
```



Important

The name to be used for the retrieval of a **RuleExecutionSet** must be specified when it is registered. (There is also a field intended to allow one to "pass" properties; as this is currently unused, just pass **null**.)

Example 6.4. Register the RuleExecutionSet

```
// Register the RuleExecutionSet with the RuleAdministrator
String uri = ruleExecutionSet.getName();
ruleAdministrator.registerRuleExecutionSet(uri, ruleExecutionSet, null);
```

6.2.2. Using "Stateful" and "Stateless" Rule Sessions

The run-time is obtained from the **RuleServiceProvider**. It is used to create *stateful* and *stateless* rule engine sessions.

Example 6.5. Obtaining the RuleRuntime

```
RuleRuntime ruleRuntime = ruleServiceProvider.getRuleRuntime();
```

In order to create a *rule session*, follow these instructions:

1. Use either one of the two public constants for **RuleRuntime**, namely **RuleRuntime.STATEFUL_SESSION_TYPE** or **RuleRuntime.STATELESS_SESSION_TYPE**.
2. Provide the uniform resource indicator for the **RuleExecutionSet** to be used to instantiate the **RuleSession**.
3. Either set the **properties map** to **null** or use it to specify globals. (This is shown in the next section.)
4. The **createRuleSession(...)** method returns a **RuleSession** instance. Cast this to either **StatefulRuleSession** or **StatelessRuleSession**.

Example 6.6. Stateful Rule

```
(StatefulRuleSession) session =
    ruleRuntime.createRuleSession( uri,
                                   null,
                                   RuleRuntime.STATEFUL_SESSION_TYPE );
session.addObject( new PurchaseOrder( "lots of cheese" ) );
```

```
session.executeRules();
```

The **StatelessRuleSession** has a very simple API; use it to call `executeRules(List list)` (which passes a list of objects) and, optionally, a filter. The resulting objects will then be returned.

Example 6.7. Stateless Rule

```
(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 );
```

6.2.3. Globals

It is possible to support globals with JSR94, albeit in a non-portable manner. To achieve this, use a method that passes the `properties` map to the `RuleSession` factory. Firstly, define the globals in either the **DRL** or the **XML** file, lest an exception be thrown.

The key represents the identifier declared in either the **DRL** or the **XML** file. The value of this key is the instance to use in the execution. In the following example, the results are collected in a global `java.util.List` list:

Example 6.8. Globals

```
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 the **DRL** file. Do so in this way:

Example 6.9. Global List

```
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

To learn more about JSR94, please refer to one or more of the following documents:

- *Official JCP Specification for Java Rule Engine API (JSR 94)*
<http://www.jcp.org/en/jsr/detail?id=94>
- *The Java Rule Engine API Documentation*
http://www.javarules.org/api_doc/api/index.html
- Friedman-Hill, E. *Jess and the javax.rules API*. TheServerSide.com, 2003
<http://www.theserverside.com/articles/article.tss?l=Jess>
- Mahmoud, Q. H. *Getting Started with the Java Rule Engine API (JSR 94): Toward Rule-Based Applications*. Sun Developer Network, 2005
<http://java.sun.com/developer/technicalArticles/J2SE/JavaRule.html>
- Rupp, N. A. *The Logic From The Bottom Line: an Introduction to the Drools Project*. TheServiceSide.com, 2004
<http://www.theserverside.com/articles/article.tss?l=Drools>

JBoss Developer Studio

The **JBoss Developer Studio** application is the only supported *integrated development environment* (IDE) for **JBoss Rules**. It provides a set of features that many programmers find very helpful. Read this chapter to learn how to use it.

Note

The **JBoss Rules IDE's** components are also available separately as **Eclipse** plug-ins.

Note

The **JBoss Developer Studio** is not required to write rules and the **JBoss Rules** engine is in no way dependent on the **Eclipse** environment.

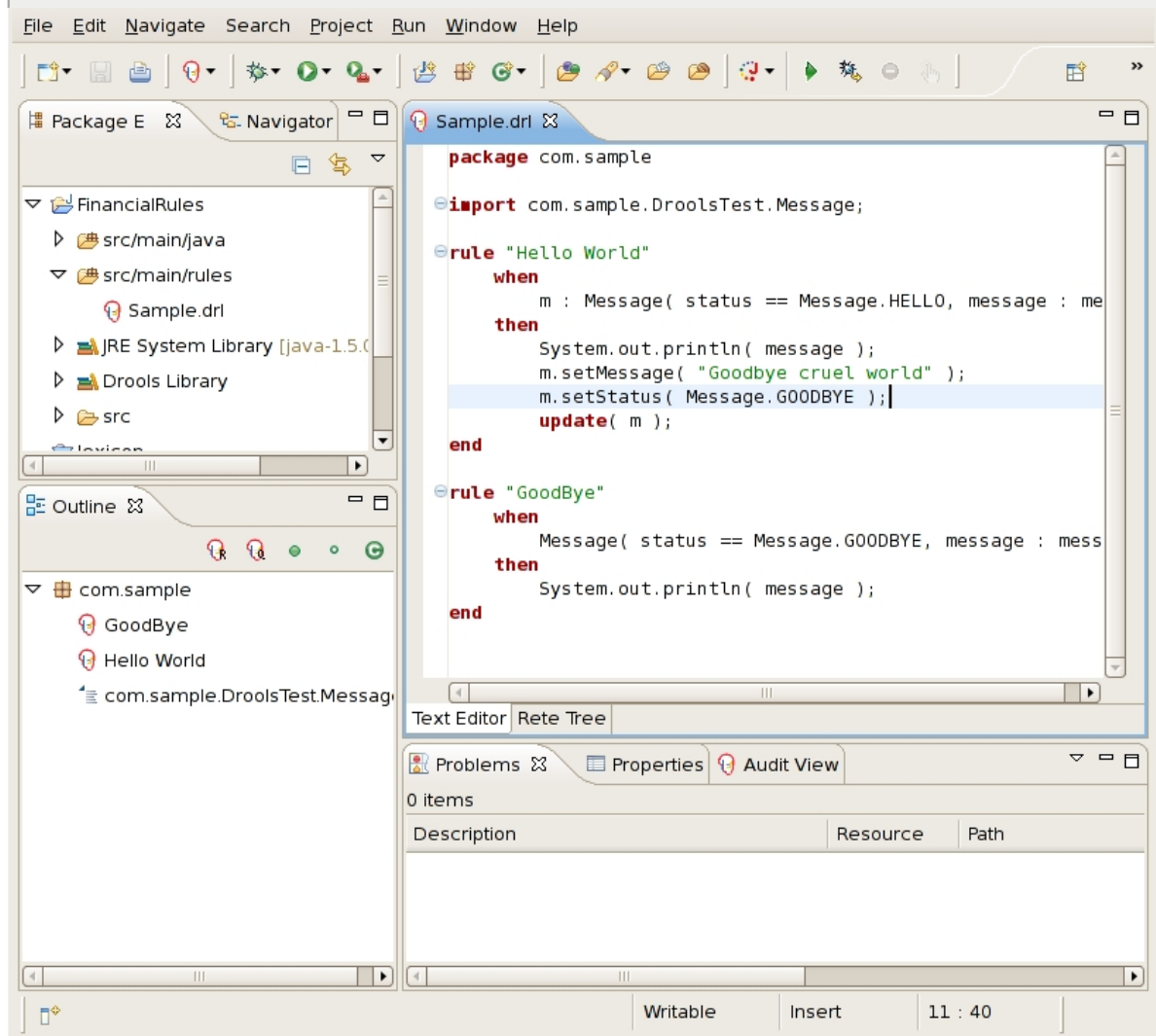


Figure 7.1. Overview

7.1. Overview

JBoss Developer Studio possesses the following features:

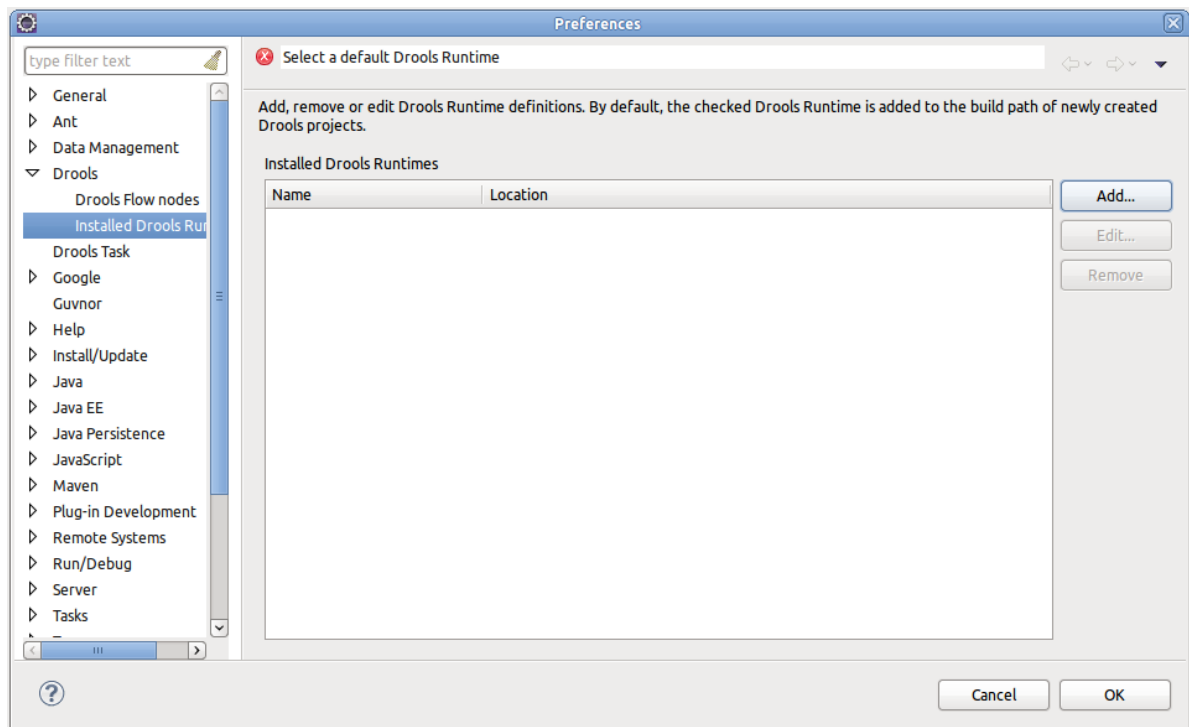
- A **DRL** syntax aware-editor that provides *content assistance* functionality (including an **outline view**)
- A *domain-specific language* extension-aware editor that also provides *content assistance* functionality
- A **Rule-Flow Graphical Editor** for editing rule-flow graphs (which represent processes). These can then be applied to the rule packages, granting them *imperative control*.
- Wizards to create the following:
 - "rules" projects
 - Rule resources, (in the form of either **DRL** or **BRL** files.
 - Domain Specific language.
 - Decision tables.
 - Rule-flows.
- a **domain-specific language editor** for creating and managing mappings between the custom language and the rule language.
- Rule validation automatically re-builds the rule every time a change is made to it. It reports any errors encountered via the **Problem View**.

7.2. Drools Runtimes

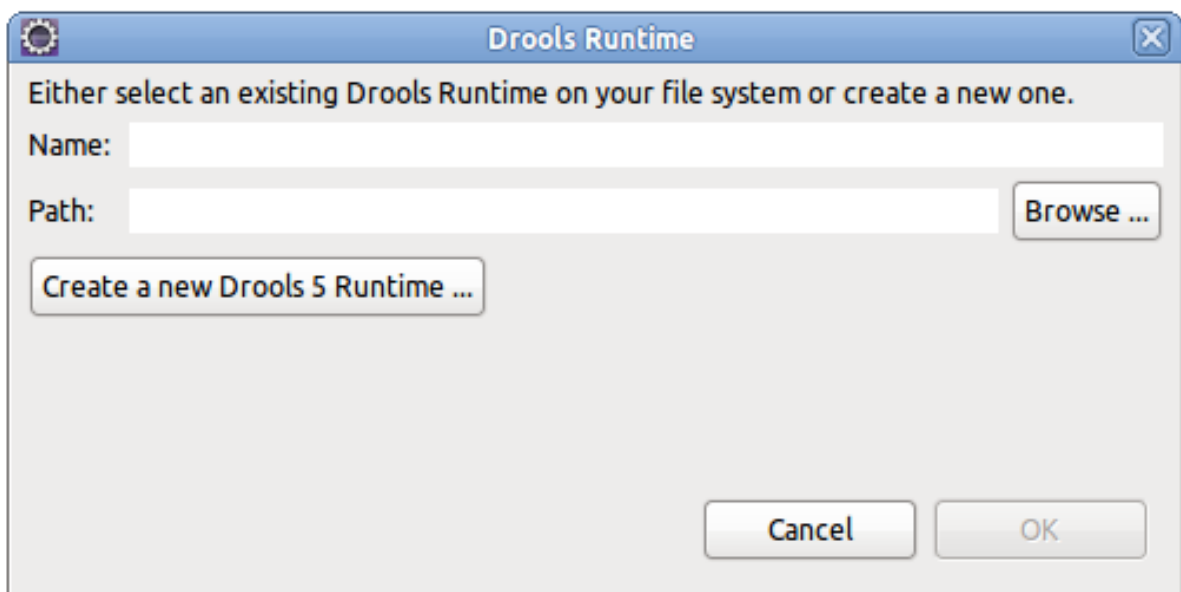
A Drools runtime is a collection of jar files that represent one specific release of the Drools project jars. To create a runtime, you must point the IDE to the release of your choice. If you want to create a new runtime based on the latest Drools project jars included in the plugin itself, you can also easily do that. You are required to specify a default Drools runtime for your Eclipse workspace, but each individual project can override the default and select the appropriate runtime for that project specifically.

7.2.1. Defining a Drools Runtime

To define one or more Drools runtimes using the Eclipse preferences view you open up your Preferences, by selecting the "Preferences" menu item in the menu "Window". A "Preferences" dialog should show all your settings. On the left side of this dialog, under the Drools category, select "Installed Drools runtimes". The panel on the right should then show the currently defined Drools runtimes. If you have not yet defined any runtimes, it should look like the figure below.



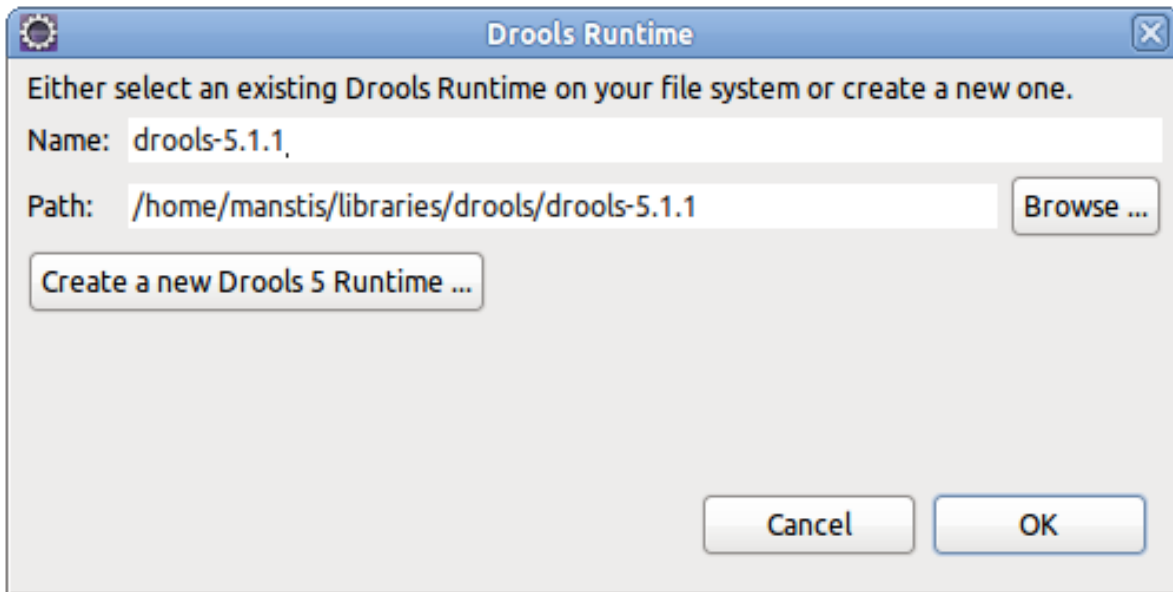
To define a new Drools runtime, click on the add button. A dialog such as the one shown below should pop up, asking for the name of your runtime and the location on your file system where it can be found.



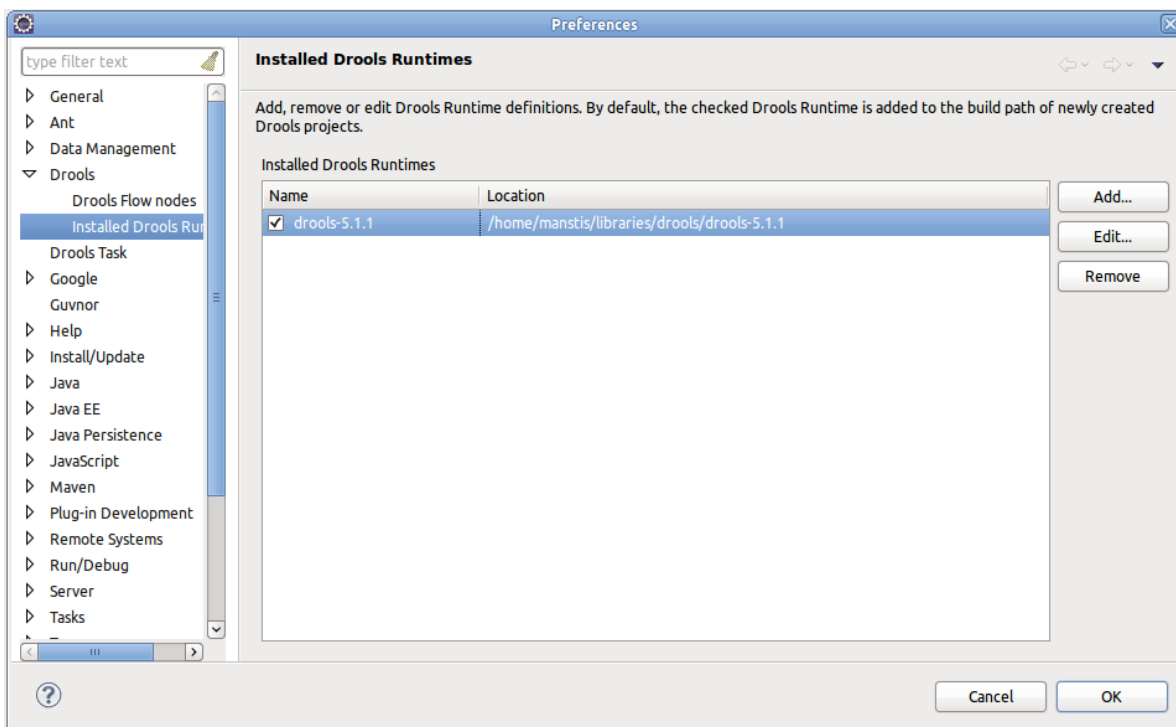
In general, you have two options:

1. To use the default jar files as included in the Drools Eclipse plug-in, you can create a new Drools runtime automatically by clicking the "Create a new Drools 5 runtime ..." button. A file browser will show up, asking you to select the folder on your file system where you want this runtime to be created. The plug-in will then automatically copy all required dependencies to the specified folder. After selecting this folder, the dialog should look like the figure shown below.
2. If you want to use one specific release of the Drools project, you should create a folder on your file system that contains all the necessary Drools libraries and dependencies. Instead of creating a

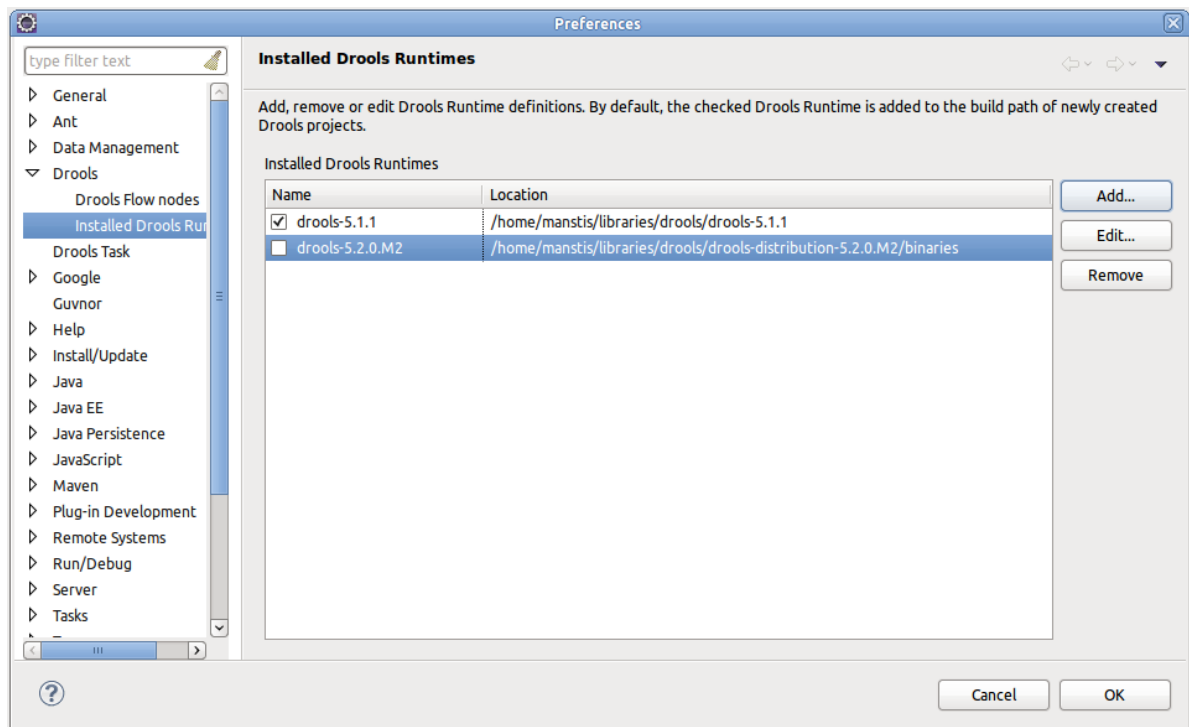
new Drools runtime as explained above, give your runtime a name and select the location of this folder containing all the required jars.



After clicking the OK button, the runtime should show up in your table of installed Drools runtimes, as shown below. Click on checkbox in front of the newly created runtime to make it the default Drools runtime. The default Drools runtime will be used as the runtime of all your Drools project that have not selected a project-specific runtime.



You can add as many Drools runtimes as you need. For example, the screenshot below shows a configuration where two runtimes have been defined: a Drools 5.1.1 runtime and a Drools 5.2.0.M2 runtime. The Drools 5.1.1 runtime is selected as the default one.

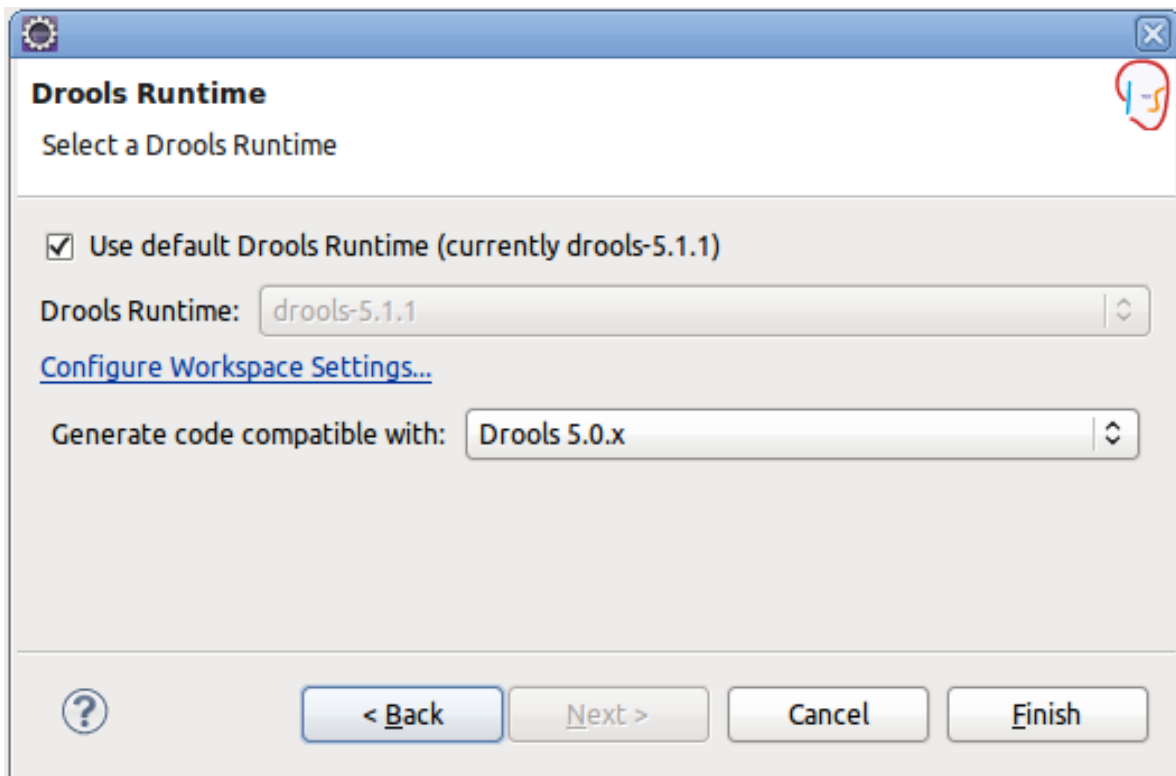


Note that you will need to restart Eclipse if you changed the default runtime and you want to make sure that all the projects that are using the default runtime update their classpath accordingly.

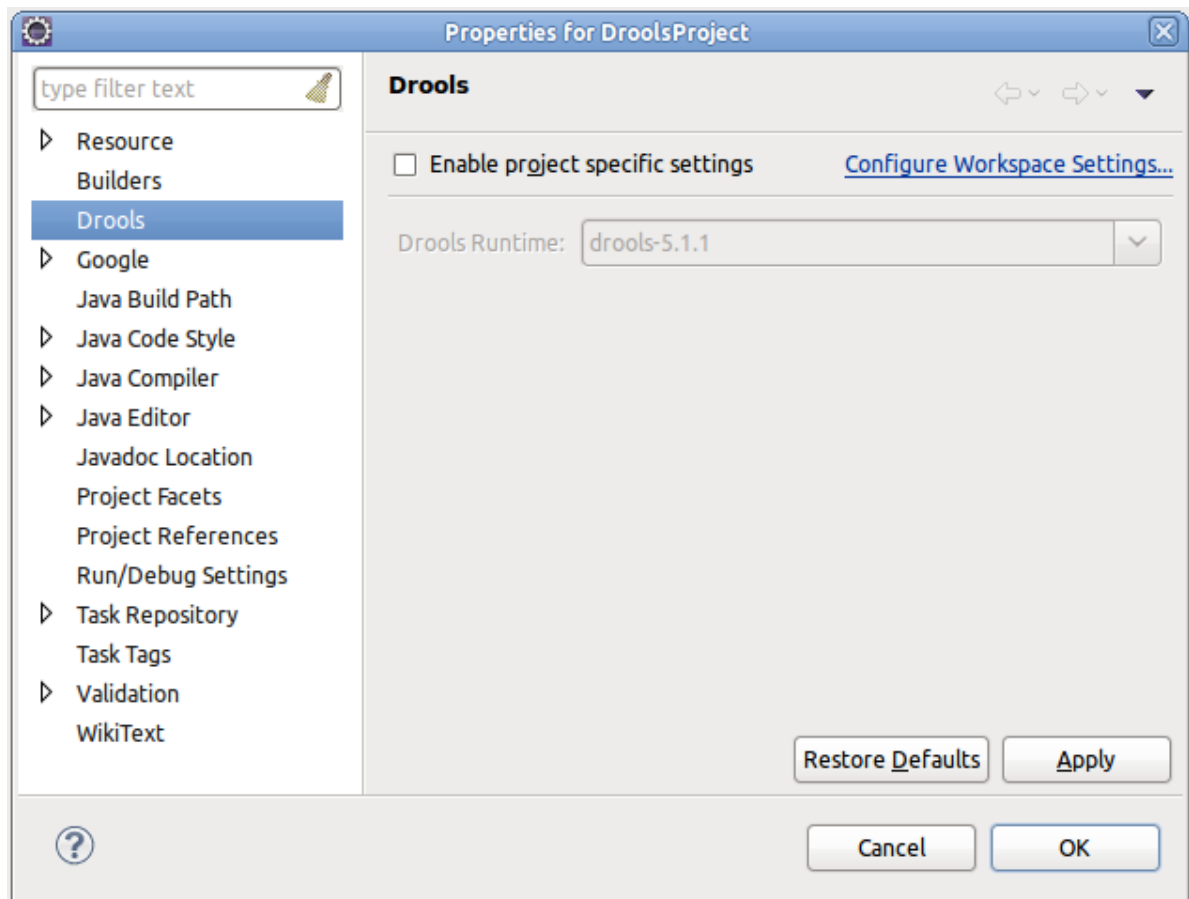
7.2.2. Selecting a runtime for your Drools project

Whenever you create a Drools project (using the New Drools Project wizard or by converting an existing Java project to a Drools project using the action "Convert to Drools Project" that is shown when you are in the Drools perspective and you right-click an existing Java project), the plugin will automatically add all the required jars to the classpath of your project.

When creating a new Drools project, the plugin will automatically use the default Drools runtime for that project, unless you specify a project-specific one. You can do this in the final step of the New Drools Project wizard, as shown below, by deselecting the "Use default Drools runtime" checkbox and selecting the appropriate runtime in the drop-down box. If you click the "Configure workspace settings ..." link, the workspace preferences showing the currently installed Drools runtimes will be opened, so you can add new runtimes there.



You can change the runtime of a Drools project at any time by opening the project properties and selecting the Drools category, as shown below. Mark the "Enable project specific settings" checkbox and select the appropriate runtime from the drop-down box. If you click the "Configure workspace settings ..." link, the workspace preferences showing the currently installed Drools runtimes will be opened, so you can add new runtimes there. If you deselect the "Enable project specific settings" checkbox, it will use the default runtime as defined in your global preferences.



7.3. Creating a Rule Project

The aim of the new project wizard is to set up an executable scaffold project to start using rules immediately. This will set up a basic structure, the classpath, sample rules and a test case to get you started.

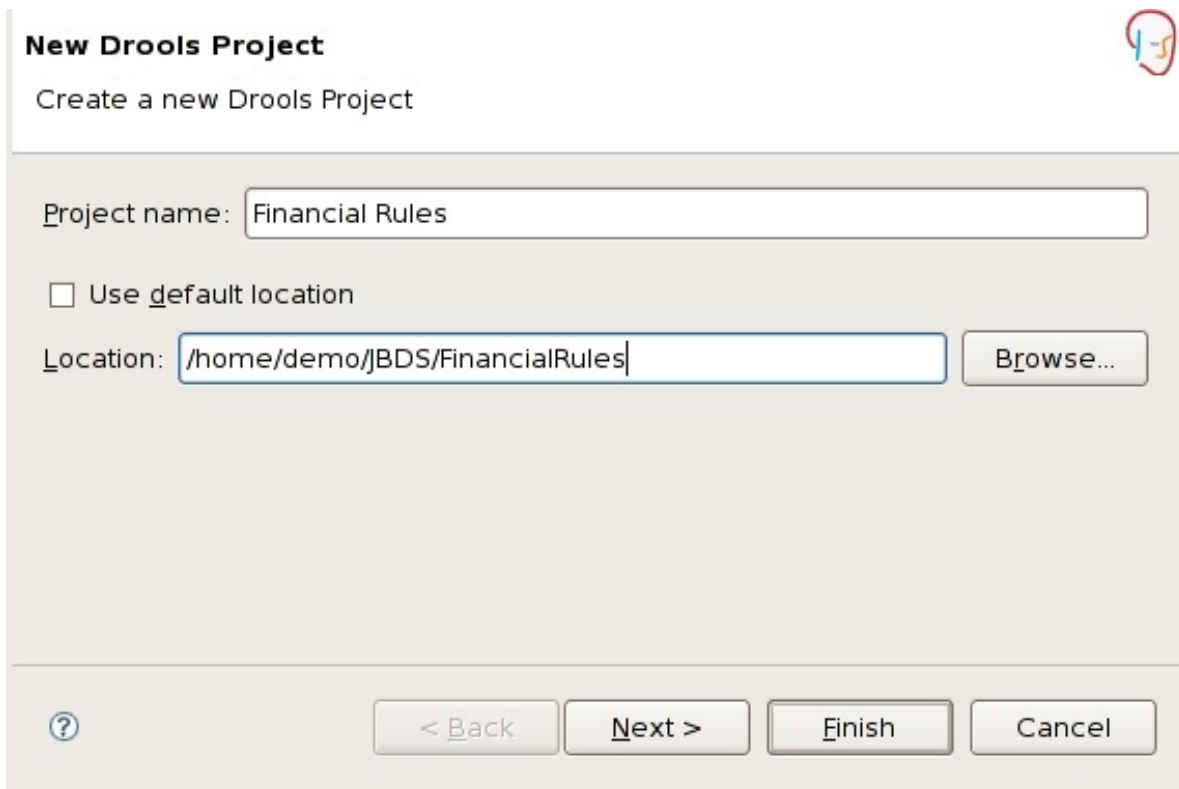


Figure 7.2. Scaffolding for a New Rules Project

When you creating a new rule project, choose whether to add default artifacts such as rules, decision tables and rule flows. These will serve as starting points and will give one an executable almost immediately. Treat this as a scaffold to customize. Study this simple Hello world rule:

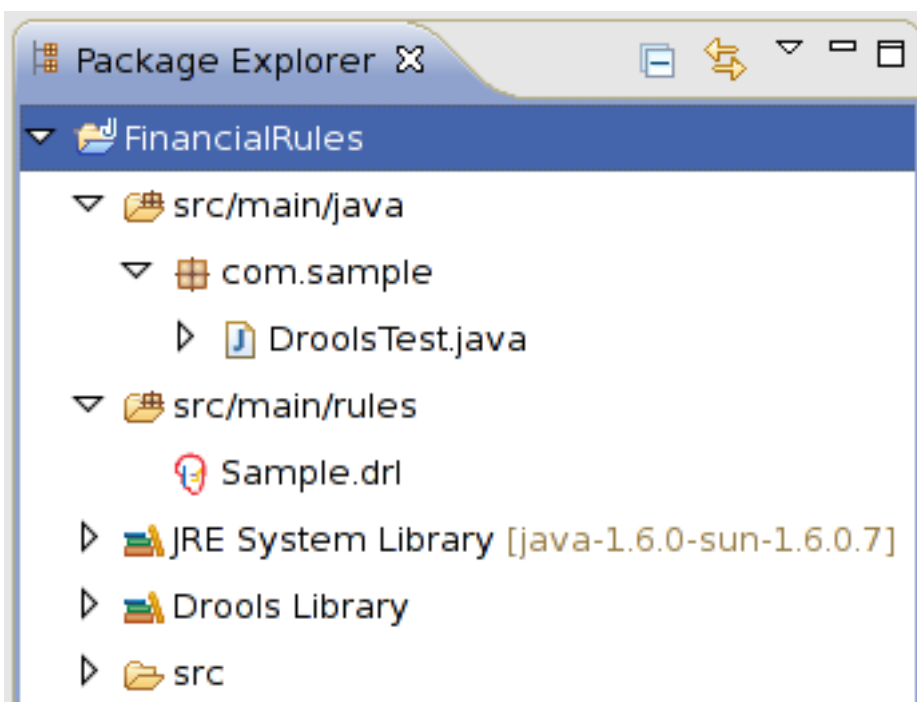


Figure 7.3. New Rule Project Result

The newly created project contains an example rule file (Sample.drl) in the src/rules directory and an example Java file (DroolsTest.java) that can be used to execute the rules in a Drools engine. You'll find this in the folder src/java, in the com.sample package. All the other 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.



Note

Strictly speaking, rules do not have to be kept in Java projects at all. This is just a convenience for those readers who are already using **JBoss Developer Studio** as their Java IDE.



Important

The **JBoss Developer Studio** provides a feature called the **JBoss Rules Builder** which automatically re-builds and validates rules every time the resources they use change. When projects are created using the **Rule Project Wizard**, this feature is enabled by default. One can also enable it manually for any other kind of project.



Important

Significantly more processing is incurred when files have large numbers of rules (typically more than five hundred.) This is because each rule will be rebuilt every time there is a file change. If this becomes a problem, one has two options. The easiest solution is to temporarily disable the builder. The alternative is to move the large rules into **.rule** files. These files are ignored by the builder but one will need to run them in a unit test to validate the rules they contain.

7.4. Creating a New Rule and Wizards

Create a rule either by generating an empty text file with a **.dr1** file extension or by using the **Wizard**. Invoke the **Wizard's** menu by pressing **Control+N** or simply click on the toolbar's **JBoss Rules** icon.

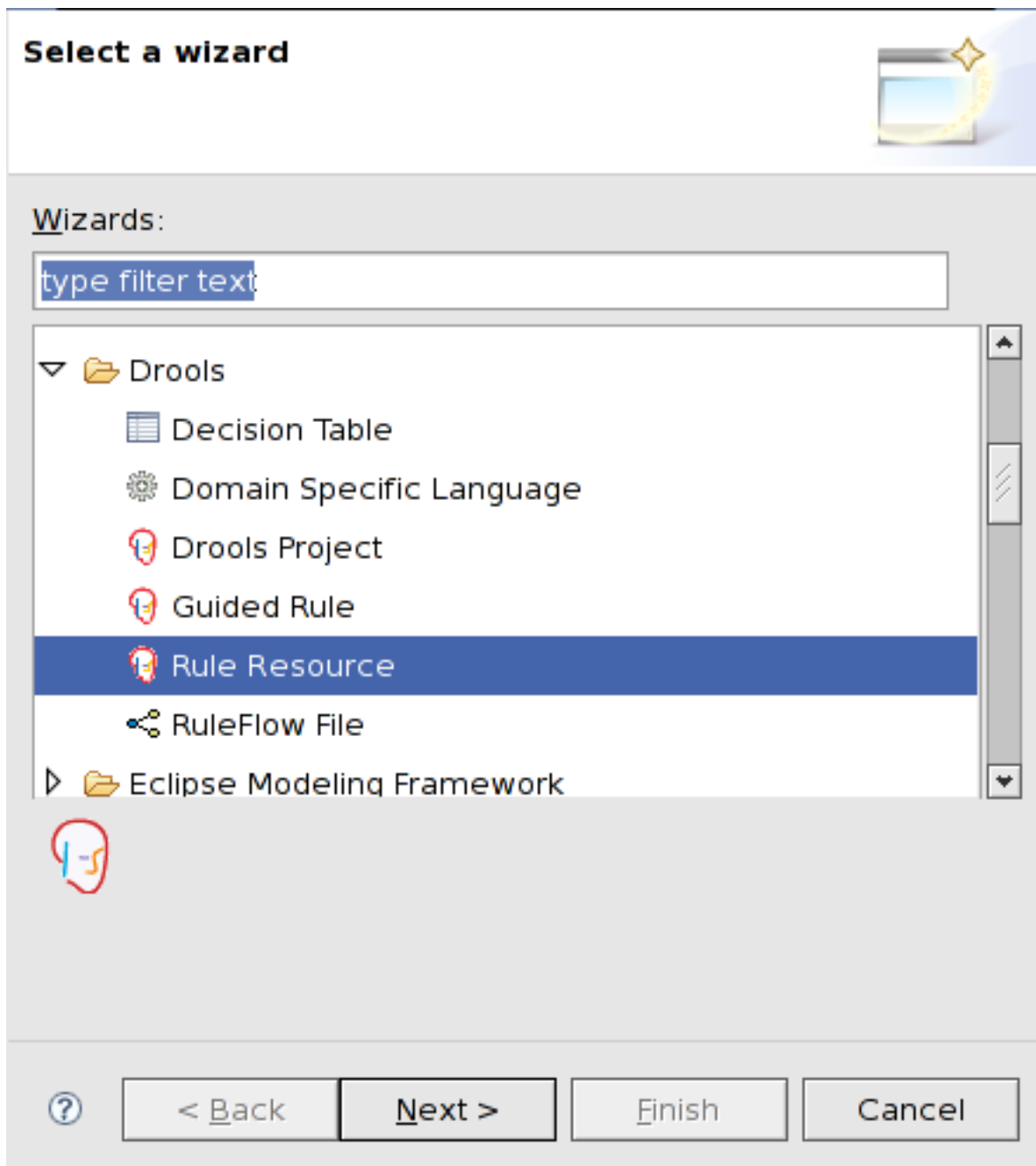


Figure 7.4. The Wizard Menu

The **Wizard** will then prompt the user for input by presenting some options related to generating a rule resource. (If unsure of what to input, note that the responses can be changed later.)

To store rule files, create a directory called **src/rules** and add suitably named subdirectories. Note that the package name is mandatory and is similar to that for a package in Java (in other words, it establishes a name-space in which to group related rules.)

New Rules File

Hint: Press CTRL+SPACE when editing rules to get content sensitive assistance/popups.

Enter or select the parent folder:

FinancialRules

Home ← →

▶ FinancialRules

File name: |

Type of rule resource: New DRL (rule package) ▾

Use a DSL:

Use functions:

Rule package name: |

Advanced >>

? < Back Next > Finish Cancel

Figure 7.5. New Rule Wizard

Having run the **Wizard**, a scaffold or skeleton has been created, which one can now "flesh out." As with all wizards, it is merely an optional helper; there is no obligation to use it if one does not desire to do so.

7.5. Textual Rule Editor

The **Rule Editor** is the tool which rule managers and developers will be using the most. The **Rule Editor** possesses the standard features of a normal **JBoss Developer Studio** text editor. In addition

to these, it provides provides "pop-up" contextual assistance. To access this functionality, press the Control and Space keys simultaneously.

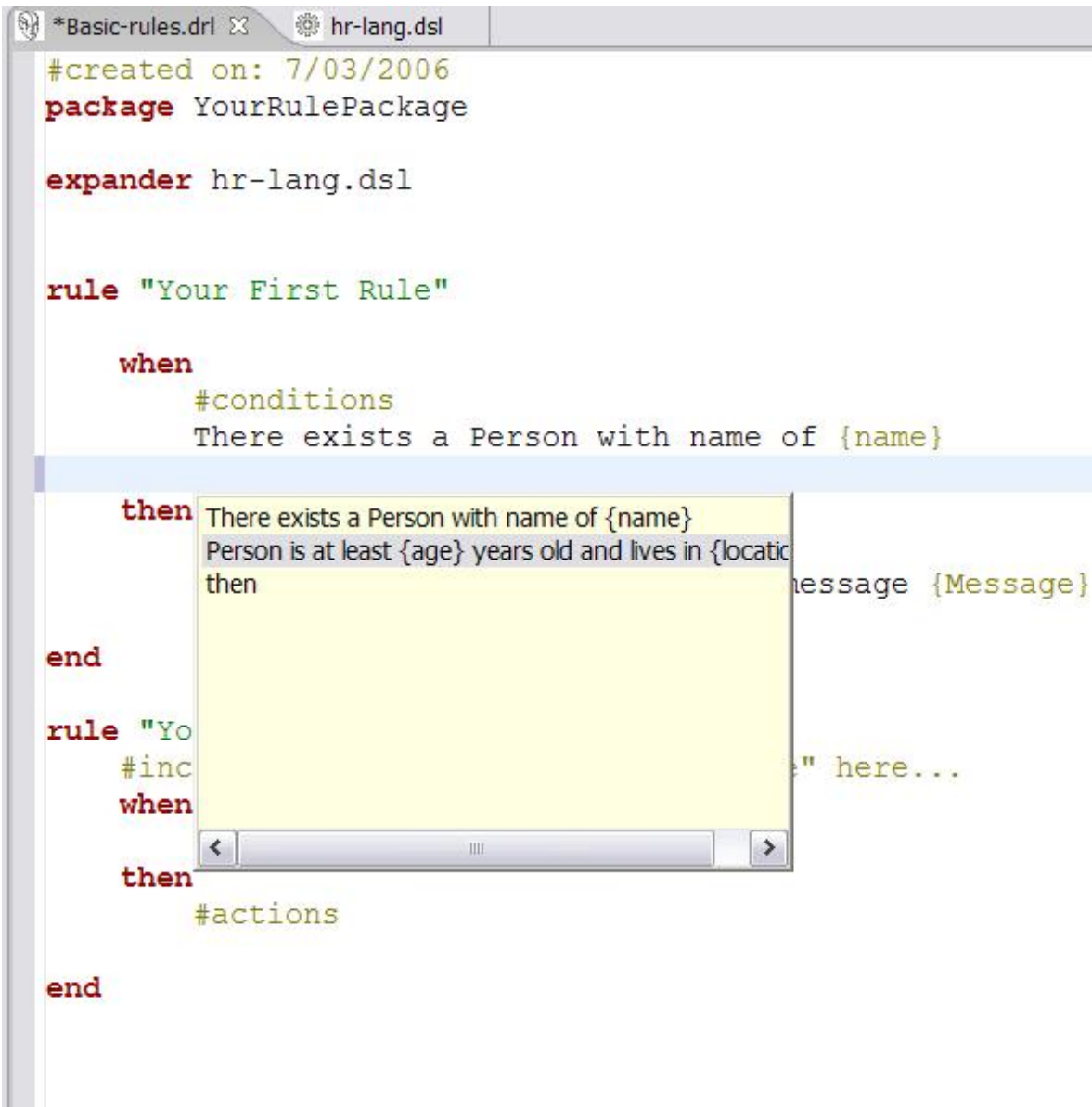


Figure 7.6. The Rule Editor in Action

The **Rule Editor** can open files that have a **.drl** or **.rule** extension. Usually these contain a number of related rules but it is also possible to have each rule in an individual file, grouped by virtue of being in the same package namespace.



Note

Data in **DRL** files is stored in plain text format.

In the example above, the rule group is using a domain-specific language. Note the presence of the **expander** keyword, which tells the rule compiler to look for a **.dsl** file of that name, the purpose of which is to resolve the rule language. Even with the domain-specific language available, the rules are still stored as plain text, mirroring what can be seen onscreen. This makes management of rules much more simple when, for instance, comparing versions.

The **editor** features an **outline view**. This remains synchronised with the rule structure (it updates each time the file is saved.) Use it to quickly and efficiently navigating among rules by name. It is particularly helpful in larger files with many hundreds of rules. Note that, by default, it lists items alphabetically.

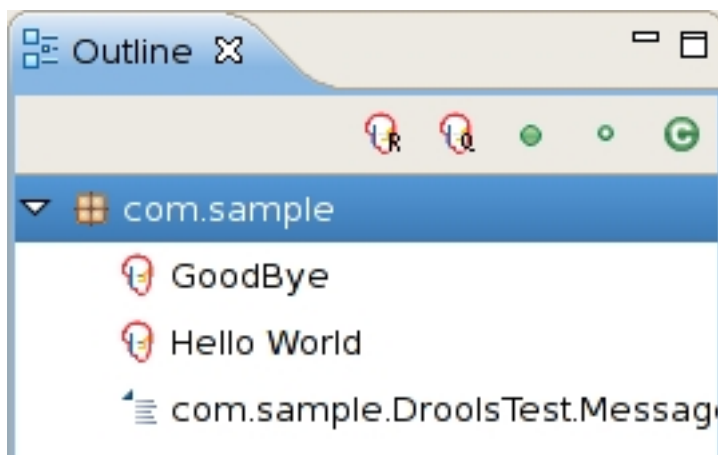


Figure 7.7. The Rule Outline View

7.6. The Guided Editor

The **JBoss Developer Studio** also possesses a feature known as the **Guided Editor**. This is similar to the web based editor available in the BRMS. It allows one to build rules graphically.

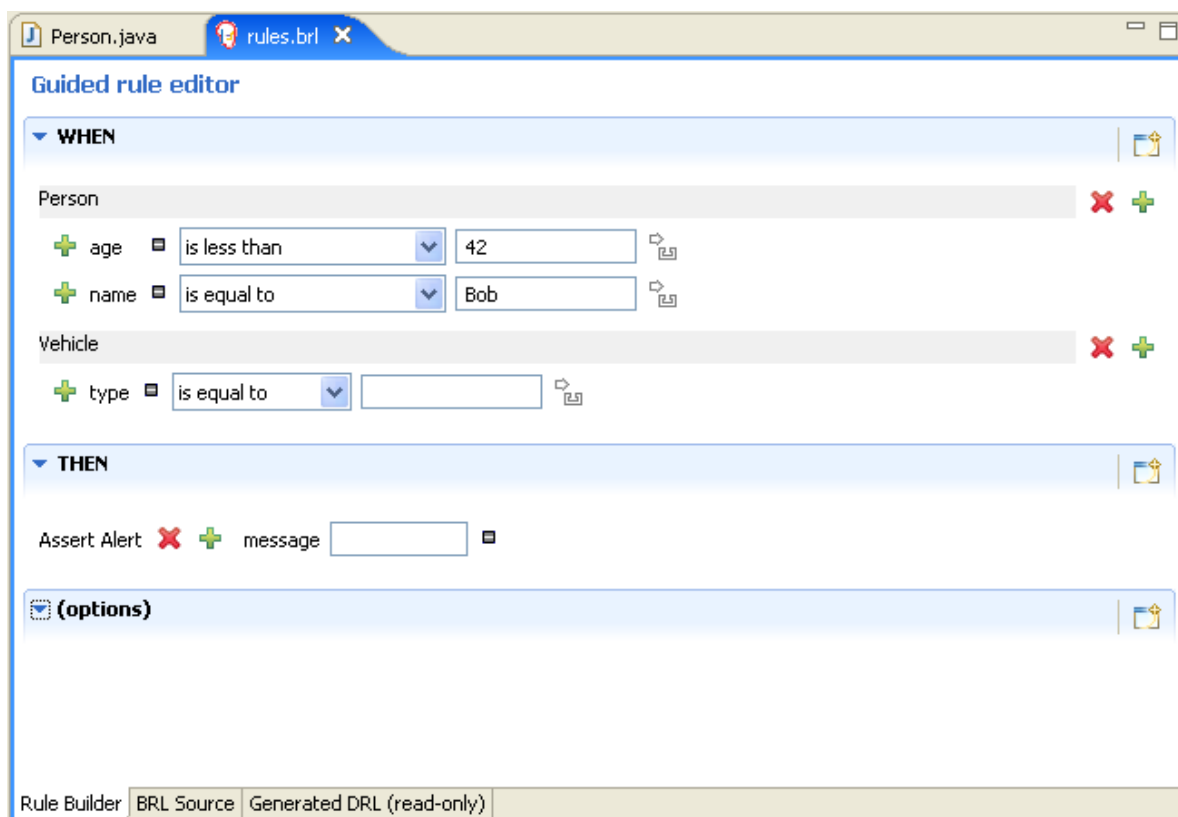


Figure 7.8. The Guided Editor

To create a rule using this tool, follow these steps:

1. Click on the **Wizard** menu.

2. Create a **.br1** file and open it in the **Guided Editor**



Note

The **Editor** works by using a **.package** file in the same directory as the **.br1** file. In this file resides the package name and import statements, just like those one finds at the top of a normal **.drl** file.

3. Populate the package file with the required **fact** classes.
4. Having added this information, follow the prompts presented by the **Guided Editor** as it presents facts and their associated fields.

Once the **model** or **fact** classes have been supplied, **Guided Editor** is able to render a graphical representation of the rule. Alternatively, one can use it and then build rules through direct use of the business rules language. One way to achieve this is by using the `drools-ant` module, which creates all of the rule assets as a rule package in a directory, enabling one to deploy it as a binary file. Alternatively, use the following snippet of code to convert the **BRL** file to a **.drl** rule.

Example 7.1. Conversion Code

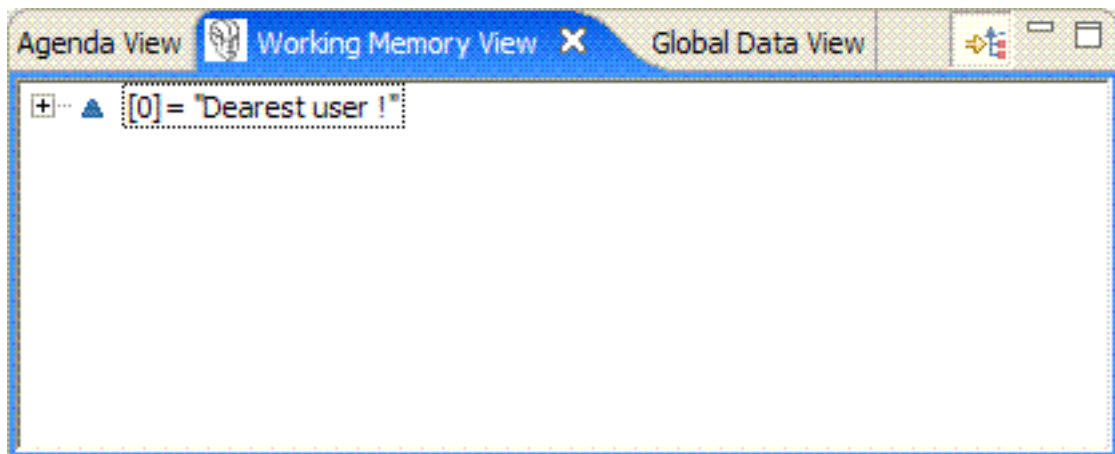
```
BRXMLPersistence read = BRXMLPersistence.getInstance();
BRDRLPersistence write = BRDRLPersistence.getInstance();
String br1 = ... // read from the .br1 file as needed...
String outputDRL = write.marshall(read.unmarshal(br1));
// Pass the outputDRL to the PackageBuilder, as usual
```

7.7. JBoss Rules Views

Use views to check the state of the **JBoss Rules** engine when debugging an application. Three views are provided, namely **working memory**, the **Working Memory View**, the **Agenda View** and the **Global Data View**. (There is also an **Audit View**.) To use them, create break-points in the code that will invoke the working memory. (The line that calls `workingMemory.fireAllRules()` is a good candidate.) If the debugger halts at that **joinpoint**, select the working memory variable in the **Debugging Variables** view. Next, use the following features to show the details of the selected working memory:

- The **Working Memory View** will show all of the elements in **JBoss Rules'** working memory.
- The **Agenda View** will, as its name implies, show all of the elements on the agenda. (The name and bound variables for each rule are shown.)
- The **Global Data View** displays all of the global data currently defined in **JBoss Rules'** working memory.
- The **Audit View** displays, in the form of a tree, the audit logs that were generated when the rules engine executed.

7.7.1. The Working Memory View

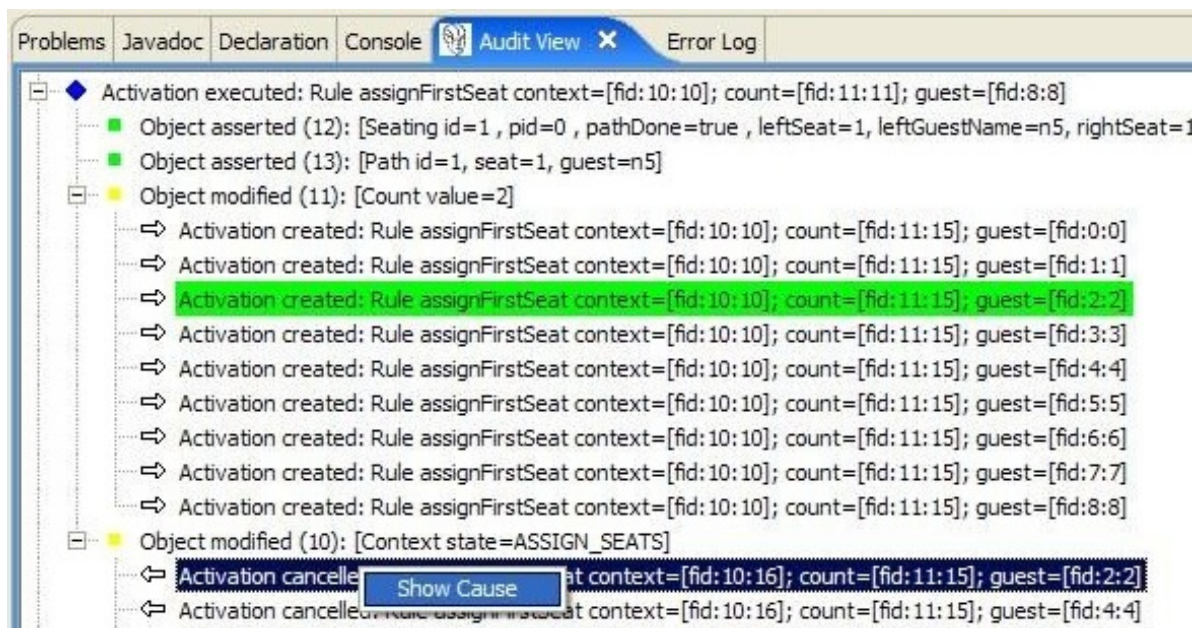


The Working Memory View shows all elements in the working memory of the Drools engine.

An action is added to the right of the view, to customize what is shown:

Click the **Show Logical Structure** icon to toggle between two options, these being that of showing the *logical structure* of each of the elements in the working memory, and that of showing the details of these elements. Logical structures help one to visualise sets of elements easily. The logical structure of AgendaItems shows both its rule and the values of all the parameters used by that rule.

7.7.2. The Audit View



Use the following code to create an audit log:

Example 7.2. Set Up Audit Log

```
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 subdirectory log (which must exist)
// of the working directory.
logger.setFileName( "log/event" );
```



```
workingMemory.assertObject(...);
workingMemory.fireAllRules();

// stop logging
logger.writeToDisk();
```

Open the log by clicking the Open Log action, the first icon in the Audit View, and select the file. The Audit View now shows all events that were 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 (right arrow)
5. Activation canceled (left arrow)
6. Activation executed (blue diamond)
7. Rule-flow started or ended ("process" icon)
8. Rule-flow group activated or deactivated ("activity" icon)
9. Rule package added or removed ("JBoss Rules" icon)
10. Rule added or removed (also the "JBoss Rules" icon)

All of these event records provide extra information about what has occurred. In the case of working memory events (such as insert, modify and retract), these details include the **id** and **toString** representation of the object. In case of an activation event (created, cancelled or executed), these include the name of the rule and all the variables bound in the activation.



Note

If an event occurs whilst an activation is being executed, it is shown as a child of that execution. To find out the cause of an event, select it. The cause, if available, will be displayed in green. Alternatively, right-click on the action and select the **Show Cause** menu entry. This will cause the cursor to jump down to the point in the log at which the cause is recorded.



Note

The cause of an object "modification" or "retraction" is recorded as the last event for that object. This is either the "object asserted" or the last "object modified" event against that same object.

The cause of an "activation canceled" or "executed" event is the corresponding "activation created" event.

7.8. Domain-Specific Languages

Domain-specific language functionality allows one to create a custom language in which English-language rules can be written. In other words, the domain-specific language reads like a natural language. To utilise, follow this process:

1. Note how a business analyst describes the rule in his own words.
2. Map this to the object model via rule constructs. (An additional benefit of this is that it can provide an insulation layer between the domain objects and the rules themselves.)

Note

A domain-specific language will grow as the number of rules expands. It is most efficient when common terms are used repeatedly, albeit with different parameters.

Note

The **Rule Workbench** provides an editor for domain-specific languages. (As the languages are stored in plain text format, one can use any editor that one desires; however, the **Rule Workbench** tool has the advantage of providing a slightly-enhanced version of the **Properties** file format.)

The **Editor** will be invoked on any file with a **.dsl** extension (there is also a **wizard** to create a sample **.dsl** file).

7.8.1. Editing languages

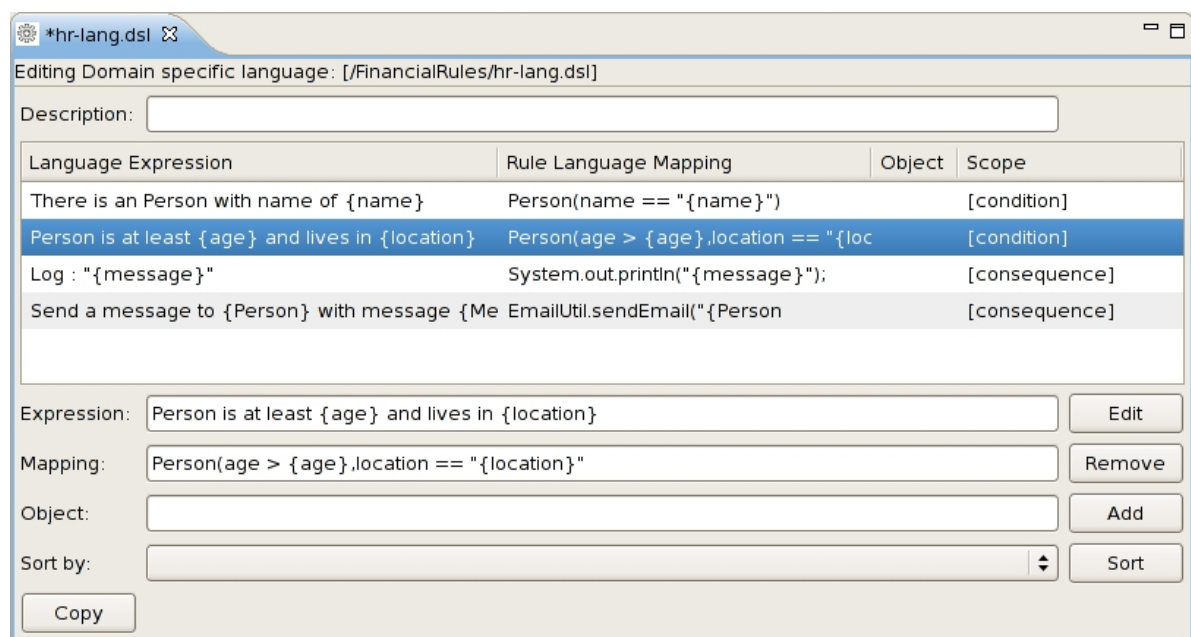


Figure 7.9. The Domain-Specific Language Editor

The **Domain-Specific Language Editor** provides a tabular view of the mapping of language to rule expressions. (The "Language Expressions" are those that are used in the rules.) The **Domain-Specific Language Editor** also feeds the *content assistance* for the **Rule Editor**. This is so that it can

suggest language expressions to the domain-specific language configuration. (The **Rule Editor** loads this configuration when the `rule` resource file is opened.) The rule's language mapping defines the "code" into which the language expressions will be compiled by the `rule` engine.

The form taken by a rule language expression depends upon whether it is intended for the "condition" or the "action" part of the rule. (For the right-hand side it may, for instance, be a snippet of Java.) The **scope** item indicates where the expression belongs: **when** indicates the left-hand side, **then** the right-hand side, and ***** means "anywhere."



Note

It is also possible to create aliases for keywords.

Select a mapping item (that is, a row in the table) to see the expression and mapping in the text fields below the table. Double-click it or press the **edit** button and the **Edit** dialogue box will open. From here, one can remove items or add new ones.



Warning

Only remove items when certain that the expression is no longer in use.

Edit an existing language mapping item.



Language expression:

Rule mapping:

Object:

Scope:



OK

Cancel

Figure 7.10. Language Mapping Editor Dialogue

The translation process occurs in the following manner:

1. The parser reads the rule text in a **DSL** file, line by line, and tries to match it against some language expressions, depending on the scope.
2. After a match is made, the values that correspond to a placeholder between braces (such as **{age}**) are extracted from the rule source.
3. The placeholders in the "Rule Expression" are replaced by their corresponding value. (In the example above, the natural language expression maps to two constraints on a fact of the type "Person," based on the fields "age" and "location," and the **{age}** and **{location}** values that are extracted from the original rule text.)



Note

If you does not wish to use a language mapping for a particular rule in a **.drl** file, prefix the expression with **>** and the compiler will ignore it. Also, please note that domain-specific languages are optional.

When the rule is compiled, the **.dsl** file will also need to be available.

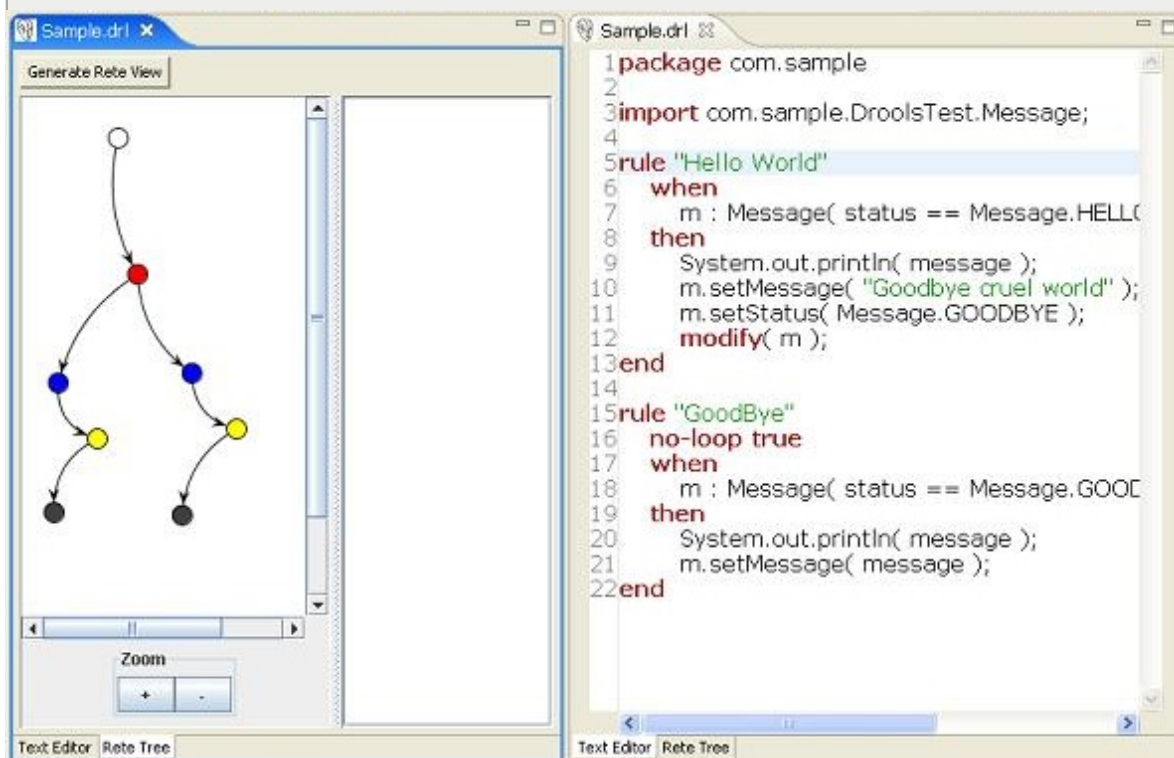
7.9. The Rete View

The *Rete Tree View* shows you the *Rete Network* for the current **.drl** file. To display it, click on the tab entitled **Rete Tree** at the bottom of the **DRL Editor** window. Once it is open, "drag-and-drop" individual nodes to arrange an optimal overview. (One can also select multiple nodes by dragging a rectangle over them; in that way, the entire group can be moved around.) The **JBoss Rules IDE** toolbar magnification icons can be used in the customary manner.



Note

A future version will allow exporting the **Rete Tree** as an image. Until this feature becomes available, take screen-shots as a workaround.



The **Rete View** is an advanced feature which takes full advantage of the **JBoss Developer Studio's Graphical Editing Framework**.



Important

This functionality can only be used with **JBoss Rules** projects, in which case the **JBoss Rules Builder** is configured in the project's properties.

7.10. Large .drl Files

Depending on the **Java Development Kit** being used, it may be necessary to increase the permanent generation setting's maximum size. Both SUN and IBM JDKs have a permanent generation setting, whereas BEA's JRockit does not.

To increase the permanent generation size, start the **JBoss Rules IDE** with `-XX:MaxPermSize=###m`. Here is an example that shows how to do so:

Example: `c:\Eclipse\Eclipse.exe -XX:MaxPermSize=128m`

Set the permanent generation to at least **128 Mb** in cases where there are more than four thousand rules.



Note

This may also apply more generally when you compiling large numbers of rules. This is because there are generally one or more classes per rule.

Alternatively, put the rules in a file with the `.rule` extension. Having done so, the background builder will not try to compile them upon each change. This may result in performance improvements, particularly if the **IDE** is becoming sluggish when processing very large volumes of rules.

7.11. Debugging Rules

It is possible to debug rules whilst **JBoss Rules** is executing. You can add break-points to the **consequences** of rules. Whenever such a break-point is encountered during the execution of the rules, the processing will halted, allowing one to inspect the variables known at that point and use any of the default debugging actions to decide what should happen next. You can also use the **Debugging View** to inspect the content of the working memory and the agenda.

7.11.1. Creating Breakpoints

Add or remove rule breakpoints in one of the following two ways:

1. by double-clicking on the **Ruler** in the **DRL Editor** when on the line on which the break-point is to be added.



Important

Such breakpoints can only be created in the **consequence** of a rule. Double-clicking on a line at which no breakpoint is allowed will do nothing.

To remove a breakpoint, double-click on the **Ruler** once more.

- 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 **Breakpoint View**. Use this to see all of the defined breakpoints, obtain their properties, and enable, disable or remove them.

7.11.2. Debugging Rules

To enable the breakpoints, the program must be debugged as a **JBoss Rules Application**.

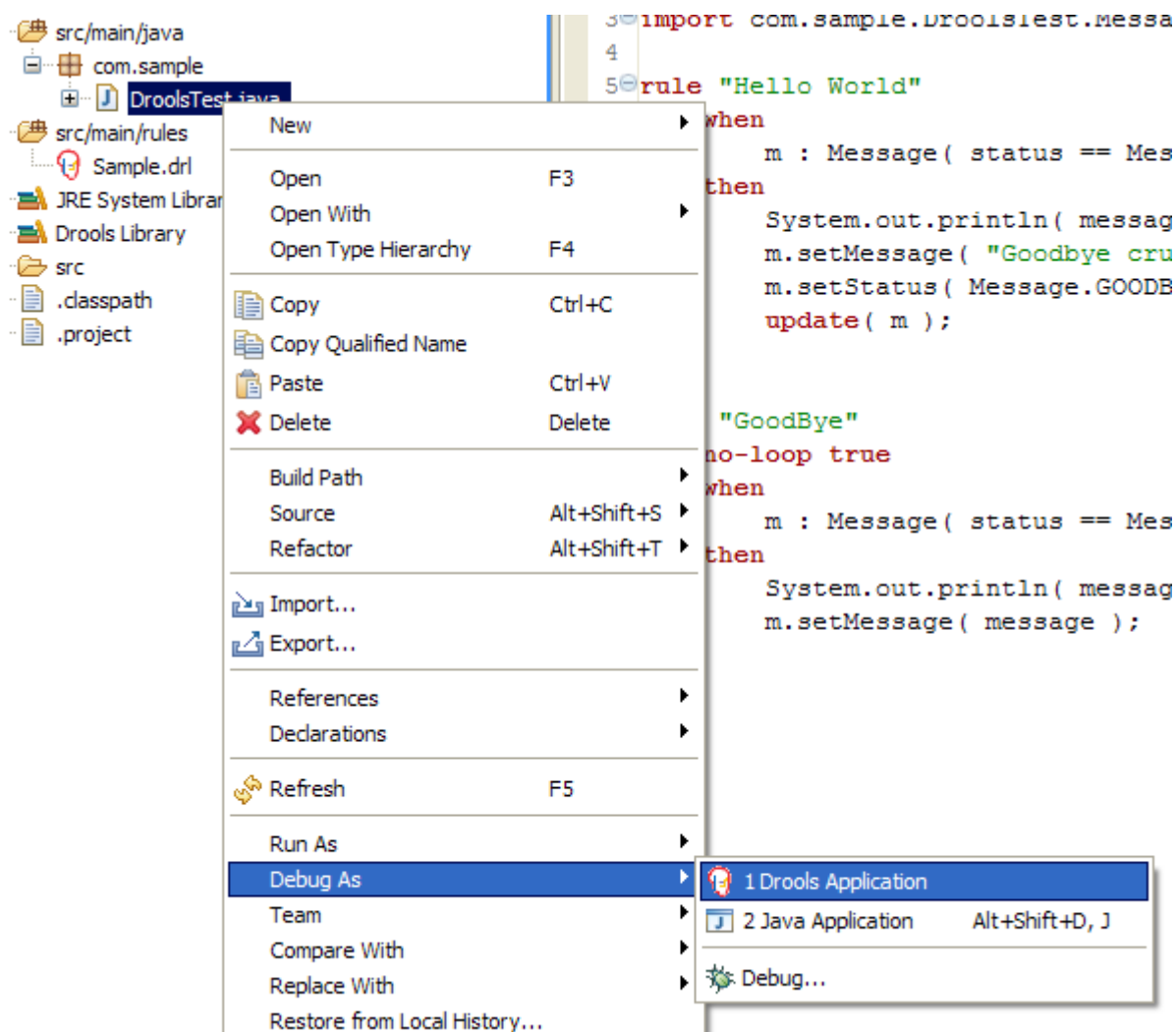


Figure 7.11. Debug as JBoss Rules Application

- Select the application's main class, then right click it, select the **Debug As** sub-menu, and then choose **JBoss Rules Application**.

Alternatively select the **Debug ...** menu item. A new dialogue box for creating, managing and running debug configurations appears (see the screenshot below.)

- Select the **JBoss Rules Application** item in the left-hand side **tree** and click the **New Launch Configuration** button (the leftmost icon in the toolbar above the **tree**.) This will create a new configuration with some of the properties (like project and main class) already set, (based on the main class selected at the beginning.) All of these properties are the same as those for any standard Java program.

3. Change the name of the debug configuration to something meaningful.
4. To start debugging the application, click on the **Debug** button at the bottom of the window.

The debug configuration only needs to be defined once, use the same configuration the next time you need to debug.



Note

The **JBoss Rules IDE** toolbar also contains shortcut buttons to quickly re-execute one of your previous configurations (at least when one of the Java, Java Debug, or JBoss Rules perspectives has been selected).

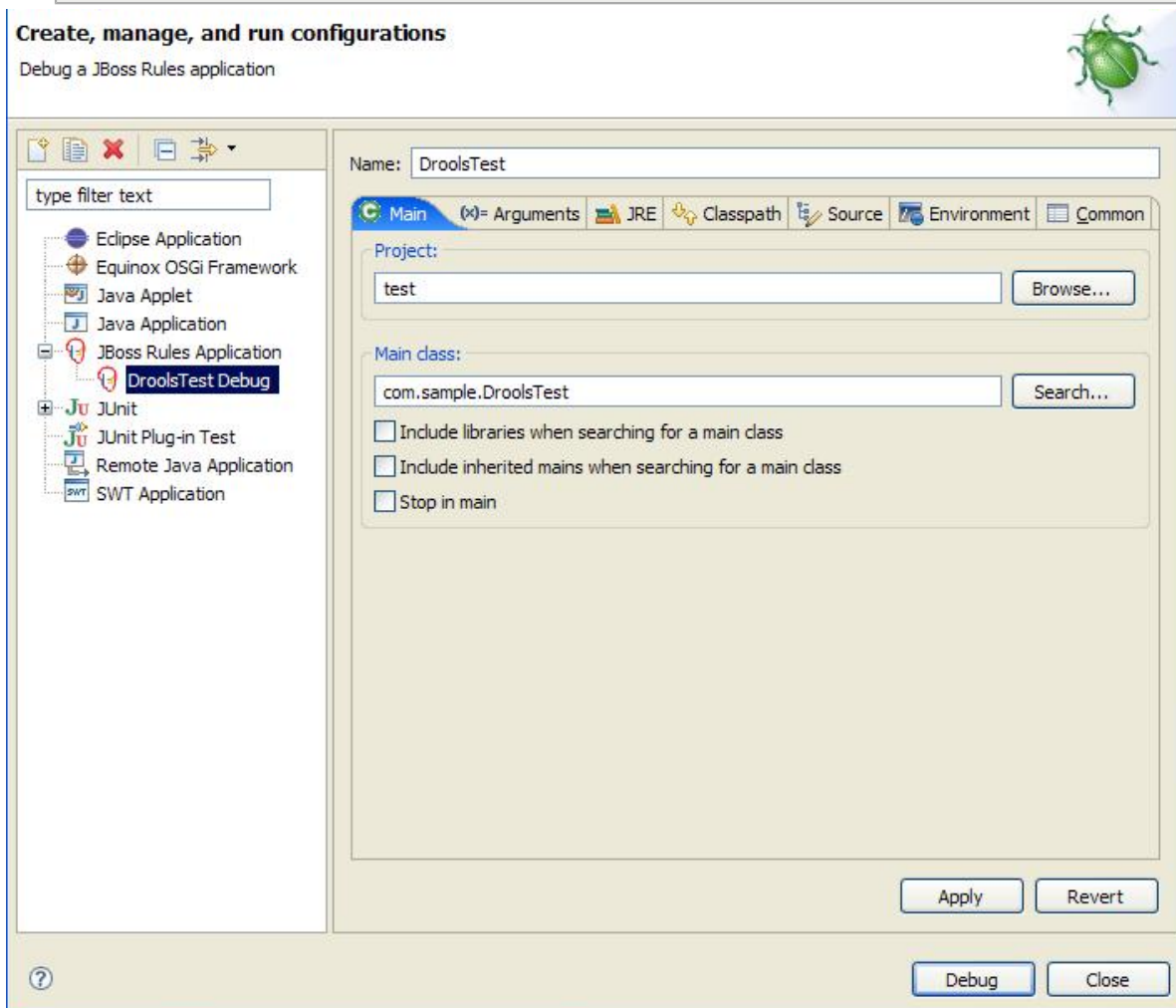


Figure 7.12. "Debug as JBoss Rules Application" Configuration

Having clicked the **Debug** button, the application will start executing and will halt if any break-point is encountered. (This can be a **JBoss Rules** rule break-point, or any other standard Java break-point.) Whenever a **JBoss Rules** rule break-point is encountered, the corresponding DRL file will open, with the active line highlighted. The **Variables View** also contains all of the rule parameters and the values associated with them.

Use the default Java debug actions to decide what to do next, be it to resume, terminate or step over the line. The **Debug View** can also be used to inspect the contents of the working memory and the

agenda at that time as well. (There is no need to select a working memory now, as that which is currently executing is displayed automatically.)

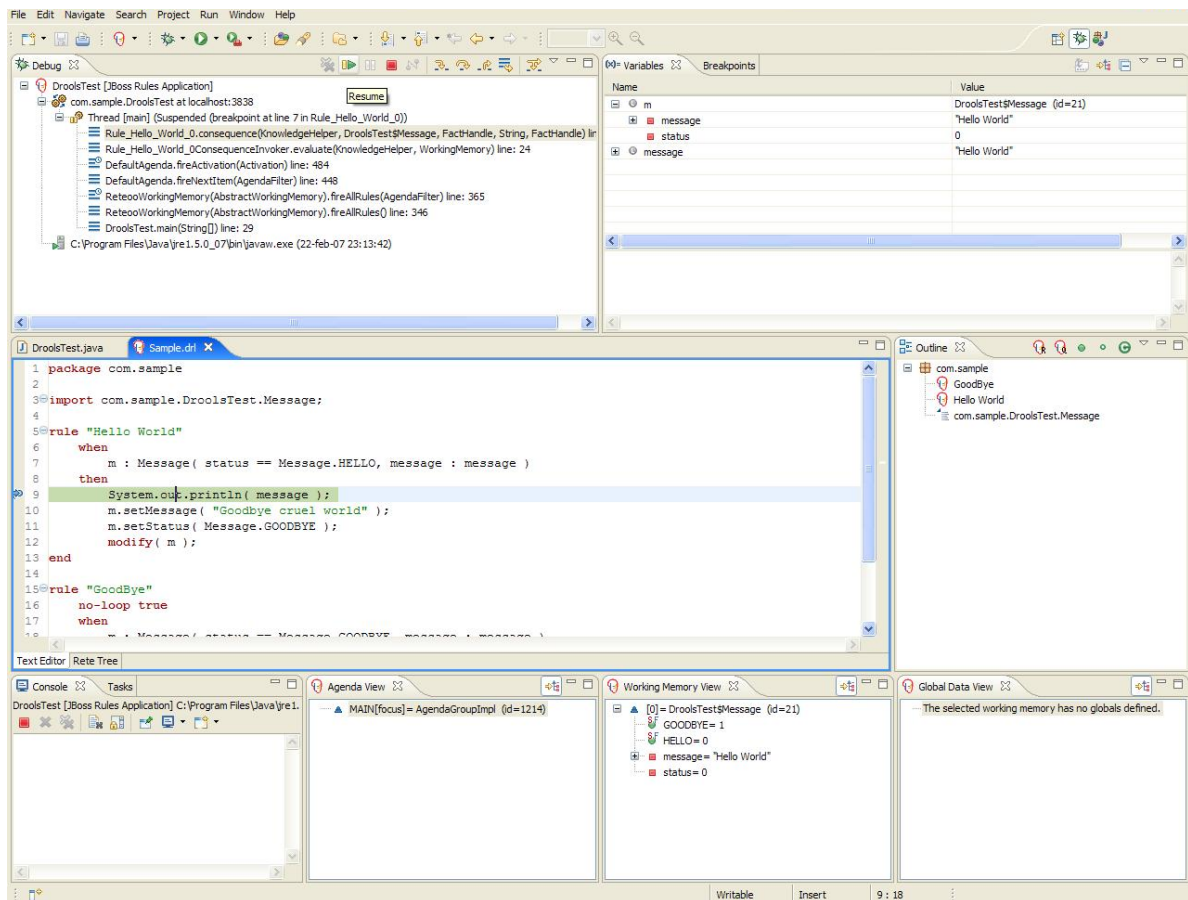


Figure 7.13. Debugging

Examples

This book ends with a series of tutorials that provide practical examples of how to use some of the functionality taught in this *Guide*. To follow the examples in this chapter, download the **Examples ZIP** archive file from <http://download.jboss.org/drools/release/5.0.1.26597.FINAL/drools-5.0-examples.zip>

8.1. HelloWorld Example

| | |
|-------------|---|
| Name: | HelloWorld Example |
| Main class: | org.drools.examples.helloworld.HelloWorldExample |
| Type: | Java application |
| Objective: | Tutorial that demonstrates simple Rules usage. |

This tutorial provides a simple example of rules usage and demonstrates both the MVFLEX Expression Language and Java dialects. It also teaches the reader how to build knowledge bases and sessions and demonstrates the audit logging and debugging output (both of which are omitted from other the other examples.)

A Knowledge Builder is used to turn a *Drools Rule Language* (DRL) source file into multiple Package objects which a knowledge base can then consume.

The add method takes both a Resource interface and a Resource Type as parameters. Use the Resource interface to retrieve the **DRL** source file from the class-path via a ResourceFactory.



Note

Although not demonstrated here, the Resource interface can also be used to retrieve **DRL** files from other locations, such as URL address. More than one file can be added if necessary.

Also, one can add **DRL** files with different name-spaces. (In this scenario, the Knowledge Builder will create a package for each name-space.) Multiple knowledge packages, each with a different name-space, can all be added to the same knowledge base.

Having added all of the **DRL**, check the Knowledge Builder for errors. (Whilst the knowledge base will validate the package, it will only have access to the error information in the form of a string so one must debug it through the the Knowledge Builder instance.)

Once any errors have been rectified, obtain the Package collection, instantiate a Knowledge Builder from the **KnowledgeBuilderFactory** and add the collection of knowledge packages.

Example 8.1. HelloWorld Example: Creating the Knowledge Base and Session

```
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();

```

JBoss Rules's *event model* exposes most of its own internal processes. Two default debug listeners, `DebugAgendaEventListener` and `DebugWorkingMemoryEventListener`, are supplied. These output debugging information to the **Error Console**. (It is easy to add listeners to a session and the process for doing so is discussed later in this section.)

The `KnowledgeRuntimeLogger` is a specialised derivative of the the `Agenda` and `WorkingMemory` listeners. It provides *execution auditing*, the output of which can be viewed on a graphical display.



Important

When the engine has finished executing, `logger.close()` must be called.



Note

Most of the examples in this book use JBoss Rules's audit logging features to record execution flow for future inspection.

Example 8.2. HelloWorld Example: Event Logging and Auditing

```

// setup the debug listeners
ksession.addEventListener( new DebugAgendaEventListener() );
ksession.addEventListener( new DebugWorkingMemoryEventListener() );

// setup the audit logging
KnowledgeRuntimeLogger logger =
    KnowledgeRuntimeLoggerFactory.newFileLogger(ksession, "log/helloworld");

```

This is a simple example of a single class with only two fields, these being the message, which is a string, and the status, which can be either of the two integers HELLO or GOODBYE.

Example 8.3. HelloWorld Example: Message Class

```

public static class Message
{
    public static final int HELLO    = 0;
    public static final int GOODBYE = 1;
}

```

```
private String      message;  
private int        status;  
...  
}
```

This creates a single Message object containing the words **Hello World** and possessing a status of HELLO. This object is then inserted into the engine, at which point `fireAllRules()` is executed.



Note

Remember that all network evaluation is undertaken during the period of insertion so that, by the time the executing program reaches the `fireAllRules()` method call, the engine already knows which rules are full matches that can be fired legitimately.

Example 8.4. Execution

```
final Message message = new Message();  
message.setMessage("Hello World");  
message.setStatus(Message.HELLO);  
ksession.insert(message);  
  
ksession.fireAllRules();  
  
logger.close();  
  
ksession.dispose();
```

In order to execute the example as a Java application, follow these steps:

1. Open the `org.drools.examples.helloworld.HelloWorldExample` class in the **JBoss Rules IDE**.
2. Right-click on the class and select **Run as...** and then **Java application** from the **context menu**.



Note

By adding a break-point to the `fireAllRules()` method and selecting the `ksession` variable, one will see that the Hello World view has already been activated and added to the **Agenda**. (This confirms that all of the pattern-matching work was already undertaken during the insert.)

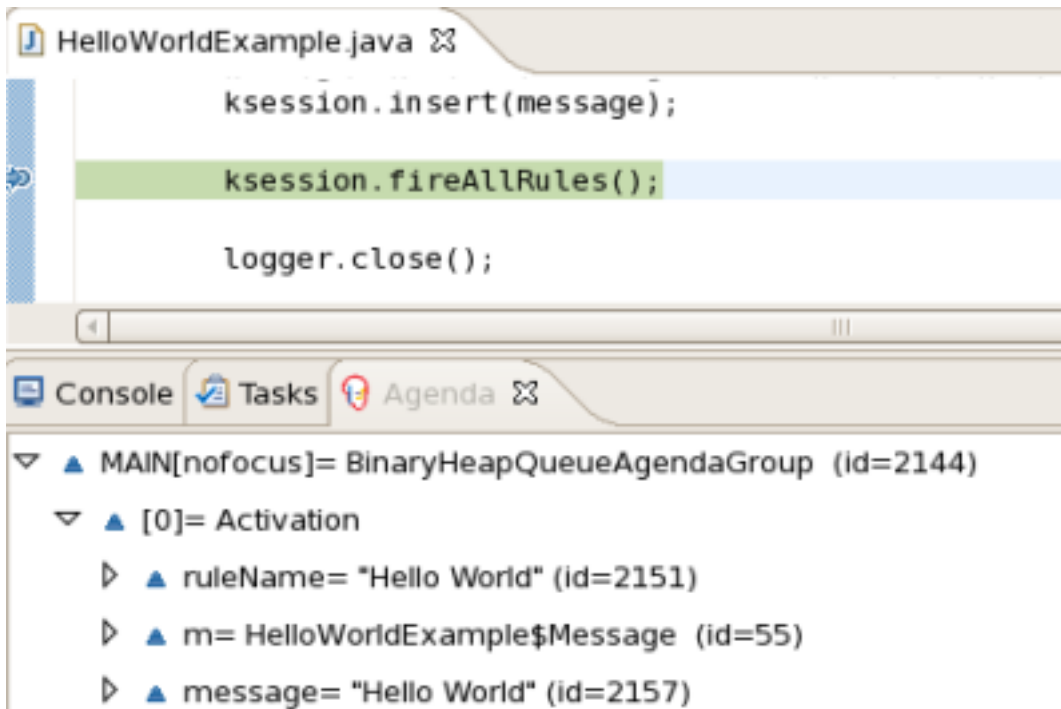


Figure 8.1. fireAllRules Agenda View

Any application print-outs are sent to `System.out`, whilst the debug listener print-outs are directed to `System.err`.

Example 8.5. System.out in the Console Window

```
Hello world
Goodbye cruel world
```

Example 8.6. System.err in the Console Window

```
==>[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]
```

The "left-hand" portion of the rule (after **when**) states that it will be activated for each Message object with a status of **Message.HELLO** upon insertion into working memory.

Additionally, the left-hand portion of the code dictates that two variable bindings be created: **message** (which is bound to the message attribute) and **m** (which is bound to the matched Message object itself.)

The right-hand side (after **then**) is the "consequence" part of the rule. Observe that it is written in MVEL, (as declared in the rule's dialect attribute.) This part of the rule sends the contents of the bound variable **message** to **System.out**. After that, it changes the message and status attributes values contained in the message object bound to **m** via MVEL's **modify** statement. This statement allows one to apply a block of assignments all at once, (with the engine being automatically notified of the changes once the block is completed.)

Example 8.7. Hello World Rule

```
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
```

To inspect the **Agenda** view again whilst the rule consequence is executing, follow these steps:

1. Set a break-point on the **modify** call in the **DRL** file.
2. Open the **org.drools.examples.HelloWorld** class in the **JBoss Rules IDE**.
3. Start executing it by going to the context menu and clicking on **Debug As...** and then **JBoss Rules Application**.

The other rule, entitled Good Bye, uses the Java. It is now activated and placed on the agenda.

The screenshot shows a rule editor with a text editor and an agenda view. The text editor contains the following code:

```

rule "Hello World"
  dialect "mvel"

  when
    m : Message( status == Message.HELLO, m
  then
    System.out.println( message );
    modify ( m ) { message = "Goodbyte cruel wc
                    status = Message.GOODBYE };
  end

```

The agenda view shows the following structure:

```

MAIN[focus]= BinaryHeapQueueAgendaGroup (id=1530)
├── [0]= Activation
│   ├── ruleName= "Good Bye"
│   └── message= "Goodbyte cruel world"

```

Figure 8.2. Hello World Rule Agenda View

The Good Bye rule, is similar to the Hello World rule except that it matches those Message objects with a status of `Message.GOODBYE`.

Example 8.8. The Good Bye Rule

```

rule "Good Bye"
  dialect "java"
  when
    Message( status == Message.GOODBYE, message : message )
  then
    System.out.println( message );
  end

```

Think back to the Java code which used the `KnowledgeRuntimeLoggerFactory` method's `newFileLogger` to create a `KnowledgeRuntimeLogger`. It then called `logger.close()` at the end. In doing so, it created an audit log file that can be seen in the **Audit** view.



Note

The **Audit** view is used in many of the examples as it provides a way of showing the execution flow.

Look at the screen-shot below. It depicts the following items:

1. The object is inserted, creating an activation for the Hello World rule

- The activation is then executed which updates the Message object which subsequently causes the Good Bye rule to activate and execute.
- Selecting an event in the **Audit** view highlights the original event in green. (In this example, the Activation created event is highlighted in green as the origin of the Activation executed event.)



Figure 8.3. Audit View

8.2. State Example

| | |
|-------------|--|
| Name: | State Example |
| Main class: | org.drools.examples.state.StateExampleUsingSalience |
| Type: | Java application |
| Rules file: | StateExampleUsingSalience.drl |
| Objective: | demonstrate basic rule use and how to resolve conflicts in rule firing priority. |

There are three different implementations of this example, each of which demonstrates an alternative way of implementing the same basic behavior, called *forward chaining*. Forward chaining is the engine's ability to evaluate, activate and fire rules in sequence, based on changes to the facts in working memory.

8.2.1. Understanding the State Example

The **org.drools.examples.state.State** class has two fields, these being its name and current status. The current status can be one of:

- **NOTRUN**
- **FINISHED**

Example 8.9. State Class

```
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...
}
```

Study the example above. Ignoring the `PropertyChangeSupport`, (which will be explained later), one can see that four **State** objects named **A**, **B**, **C** and **D** have been created. Initially, their states are set to **NOTRUN**, this being the default for the constructor used. Each instance is asserted, in turn, into the session and then `fireAllRules()` is called.

Example 8.10. Saliency State Example Execution

```
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, JBoss Rules will use JavaBean
// PropertyChangeListeners so you do not have to call modify or update().
boolean dynamic = true;

session.insert( a, dynamic );
session.insert( b, dynamic );
session.insert( c, dynamic );
session.insert( d, dynamic );

session.fireAllRules();
session.dispose(); // Stateful rule session must always be disposed when finished</
programlisting>
```

In order to execute the application:

1. Open the `org.drools.examples.state.StateExampleUsingSaliency` class in the **JBoss Rules IDE**.
2. Right-click on the **class** and select **Run as...** and then **Java Application**.

The following output appears in the **JBoss Rules IDE** Console window:

Example 8.11. Saliency State Console Output

```
A finished
B finished
C finished
D finished
```

There are four rules in total. The `Bootstrap` rule runs first, setting **A** to the **FINISHED** state, which then causes **B** to also become **FINISHED**. (**C** and **D** are both dependent on **B**, causing a temporary conflict which is resolved by the saliency values.)

Next, analyse the way in which this process was executed. To do so, use the *audit logging* feature. This allows one to view a graphical representation of the results of each operation. Follow these steps to obtain an audit log:

1. If the **Audit View** is not visible, click on **Window** and then select **Show View, Other..., JBoss Rules** and, finally, **Audit View**.
2. Once in the **Audit View**, click on the **Open Log** button and select the file entitled `drools-examples-dr1-dir>/log/state.log`.

At this point, the **Audit View** onscreen will look like this:

- ▼ ◆ 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) } conflict
- ▼ ◆ 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. Saliency State Example **Audit View**

Read the log in the **Audit View**, from top to bottom. As one can see, every action, and the corresponding change it has wrought in working memory, has been recorded. Hence, one can observe that asserting **State** object **A** with a status of **NOTRUN** activates the **Bootstrap** rule, whilst asserting the other **State** objects has no immediate effect.

Example 8.12. Saliency State: Bootstrap Rule

```
rule Bootstrap
when
  a : State(name == "A", state == State.NOTRUN )
then
  System.out.println(a.getName() + " finished" );
  a.setState( State.FINISHED );
end
```

Executing the **Bootstrap** rule changes the state of **A** to **FINISHED**, which, in turn, activates **A to B** rule.

Example 8.13. 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
```

Executing the **A to B** rule changes the state of **B** to **FINISHED**, which, in turn, activates both the **B to C** and **B to D** rules, and places their activations on the agenda.

From this moment on, both rules may fire and, therefore, they can be said to be "in conflict." The conflict resolution strategy allows the engine's agenda to decide which rule to fire. As rule **B to C** has the higher saliency value (ten, as opposed to the default value of zero), it fires first, setting object **C** to a state of **FINISHED**.

The **Audit View** depicted above shows the modification to the **State** object in the **A to B** rule, which results in two activations being in conflict. The **Agenda View** can also be used to investigate the state of the agenda, as it allows one to place debugging points within the rules themselves whilst the **Audit**

View is open. The screen-shot below shows a break-point in the **A to B** rule. It also illustrates the state of the agenda whilst the two rules are in conflict.

```

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
  
```

Text Editor Rete Tree

Console Tasks **Agenda View** Audit View Global Data View Rules View Working Mem

- 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. State Example: **Agenda View**

Example 8.14. B to C Rule

```

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
  
```

The **B to D** rule fires last of all, modifying the status of object **D** to **FINISHED**.

Example 8.15. B to D Rule

```
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
```

At this point in time, there are no more rules to execute and, thus, the engine stops.

A notable facet of this example is the use of *dynamic facts*, which are based upon `PropertyChangeListener` objects. In order for the engine to "see" and react to changes in fact properties, it must be informed of these by the application. This can either be undertaken explicitly via the rules (via the **modify** statement) or, implicitly, (by letting the engine know through `PropertyChangeSupport` that the facts have implemented.)

**Note**

`PropertyChangeSupport` is defined in the *Java Beans Specification*.

Study the next example to learn how to use `PropertyChangeSupport`. (By making use of it, one can avoid cluttering one's rules with **modify** statements.) To use this feature, firstly make the facts implement `PropertyChangeSupport`, in the same way that the `org.drools.example.State` class does. Then, use the following code to insert the facts into working memory:

Example 8.16. Inserting a Dynamic Fact

```
// By setting dynamic to TRUE, JBoss Rules will use JavaBean
// PropertyChangeListeners so that one does not have to call modify or update().
final boolean dynamic = true;

session.insert( fact, dynamic );
```

When the `PropertyChangeListener` objects are being used, each *setter* must implement a little extra code (for the notification.) Here is the setter for the `State` in the `org.drools.examples` class:

Example 8.17. PropertyChangeListener Support in a "Setter"

```
public void setState(final int newState) {
    int oldState = this.state;
    this.state = newState;
    this.changes.firePropertyChange( "state",oldState,newState );
}
```

There are two other classes in this example. They are called `StateExampleUsingAgendGroup` and `StateExampleWithDynamicRules`. Both execute from A to B to C to D, as demonstrated above.

- The `StateExampleUsingAgendGroup` class uses agenda groups to control the conflict of rules and to determine which one shall fire first.
- The `StateExampleWithDynamicRules` class shows how an additional rule can be added to a running working memory session.

Use *agenda groups* to partition the agenda into groups and to determine which of these groups have permission to execute. By default, all rules are in the agenda group entitled **MAIN**. The agenda-group attribute lets one specify a different agenda group for the rule. Initially, the **MAIN** agenda group is used by working memory.



Important

A group's rules will only fire when that group receives the focus. Achieve this by using either the `setFocus()` method or the auto-focus rule attribute. (With the latter technique, the rule automatically sets the focus to its agenda group when it is matched and activated, hence the name *auto-focus*. It is this method that enables Rule B to C to fire before Rule B to D.)

Example 8.18. Agenda Group State Example: Rule B to C

```
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 );
    kcontext.getKnowledgeRuntime().getAgenda().getAgendaGroup( "B to D" ).setFocus();
  end
```

B to C calls `setFocus()` on the B to D agenda group. This allows it to fire its active rules, which, in turn, triggers those belonging to B to D.

Example 8.19. Agenda Group State Example: Rule B to D

```
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
```

The `StateExampleWithDynamicRules` example adds another rule to the base after `fireAllRules()` is run. This new rule is another *state transition*:

Example 8.20. Dynamic State Example: Rule D to E

```
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
```

It produces the following, final piece of output:

Example 8.21. Dynamic Sate Example Output

```
A finished
B finished
C finished
D finished
E finished
```

8.3. Fibonacci Example

| | |
|-------------|--|
| Name: | Fibonacci |
| Main class: | org.drools.examples.fibonacci.FibonacciExample |
| Type: | java application |
| Rules file: | Fibonacci.dr1 |
| Objective: | Demonstrate Recursion, 'not' CEs and Cross Product Matching. |

The Fibonacci Numbers (http://en.wikipedia.org/wiki/Fibonacci_number), discovered by Leonardo of Pisa (see <http://en.wikipedia.org/wiki/Fibonacci>), are a sequence that start with zero and one. The next Fibonacci number is obtained by adding the two preceding ones together. Therefore, the Fibonacci sequence begins with 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987, 1597, 2584, 4181, 6765, 10946, and continues infinitely onwards following that pattern. This example uses the number sequence to demonstrate *recursion* and how use of *salience values* can resolve conflicts.

This example uses the **Fibonacci** single-fact class. It has two fields, sequence and value. The sequence field is used to indicate the position of the object in the Fibonacci number sequence. The value field shows the value of a Fibonacci object for a specific sequence position, (using **-1** to indicate a value that still needs to be computed.)

Example 8.22. Fibonacci Class

```
public static class Fibonacci {
    private int sequence;
    private long value;

    public Fibonacci( final int sequence ) {
        this.sequence = sequence;
        this.value = -1;
    }

    ... setters and getters go here...
}
```

Follow these steps to execute the example:

1. open the **org.drools.examples.FibonacciExample** class in the **JBoss Rules** integrated development environment.

- right-click on the class and select **Run as...** and then **Java application**

The following output will be displayed in the **JBoss Rules IDE Console** window (with `...snip...` indicating lines where lines have been removed to save space):

Example 8.23. Fibonacci Example: Console Output

```
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
```

To run it from Java, insert a single Fibonacci object with a sequence field of **fifty**. A recursive rule is then run, inserting the other forty-nine objects automatically.



Note

This example does not use **PropertyChangeSupport**. Rather, it utilises the MVFLEX Expression Language, meaning that the `modify` keyword can be exploited. This keyword allows one to utilise a *block setter action*. (It also notifies the engine of changes.)

Example 8.24. Fibonacci Example: Execution

```
ksession.insert( new Fibonacci( 50 ) );
ksession.fireAllRules();
```

The Recurse rule is very simple. It matches each asserted Fibonacci object with a value of **-1**, thereby creating and asserting a new Fibonacci object with a sequence of one less than the current one. Each time a Fibonacci object is added whilst none with a sequence field of **1** exist, the rule fires again.

The not conditional element is used to stop the rule's matching once all fifty Fibonacci objects are in memory. The rule also has a salience value, because you need to have all fifty **Fibonacci** objects asserted before you execute the **Bootstrap** rule.

Example 8.25. Fibonacci Example: "Recurse" Rule

```
rule Recurse
```

```

saliency 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

```

The **Audit** view shows the original assertion (with a sequence field of **50**.) After that, it shows the continual recursion of the rule, whereby each asserted Fibonacci object causes the Recurse rule to run again and again.

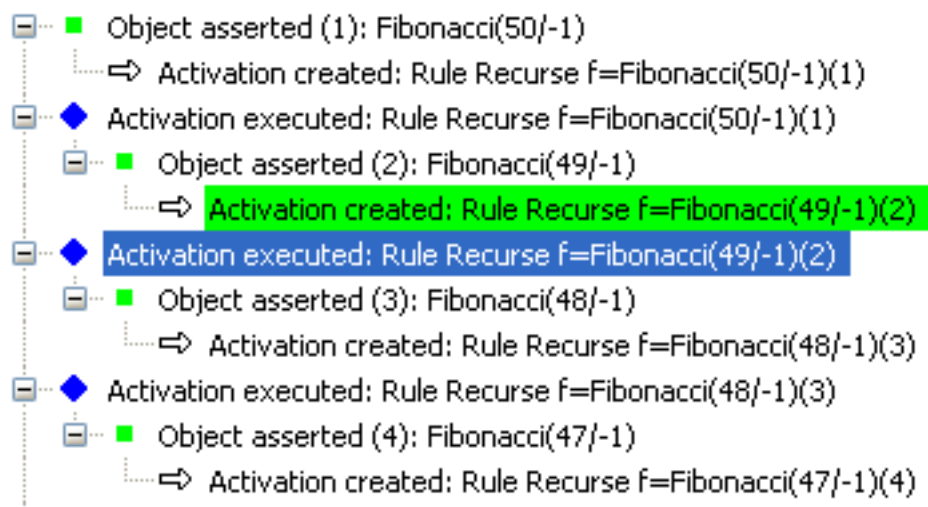


Figure 8.6. Fibonacci Example: Recurse **Audit View One**

When a Fibonacci object with a sequence field value of **2** is asserted, it will match the Bootstrap rule, activating this alongside the Recurse rule.



Note

There are multiple restrictions on the sequence field; they test if the value of the field equals **1** or **2**.

Example 8.26. Fibonacci Example: Bootstrap 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

```

At this point, the agenda looks as it does in the following diagram. However, the Bootstrap rule will not run because the Recurse rule has a higher saliency.

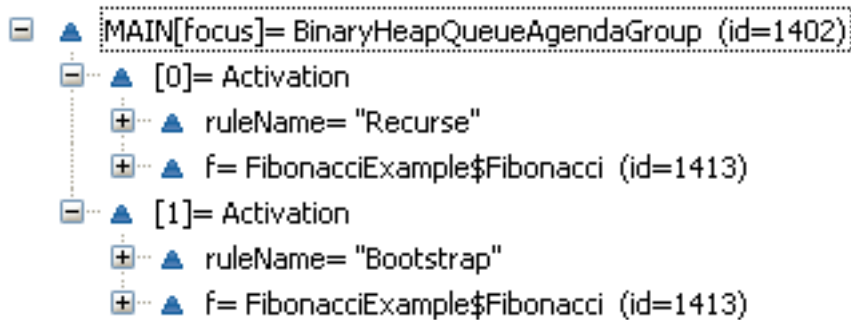


Figure 8.7. Fibonacci Example: Recurse Agenda View One

When a Fibonacci object with a sequence value of **1** is asserted, there will be a match against Recurse rule again, causing it to activate twice.



Note

The Recurse rule does not match and activate because the not conditional element prevents the rules from matching as soon as a Fibonacci object with a sequence value of **1** comes into existence.

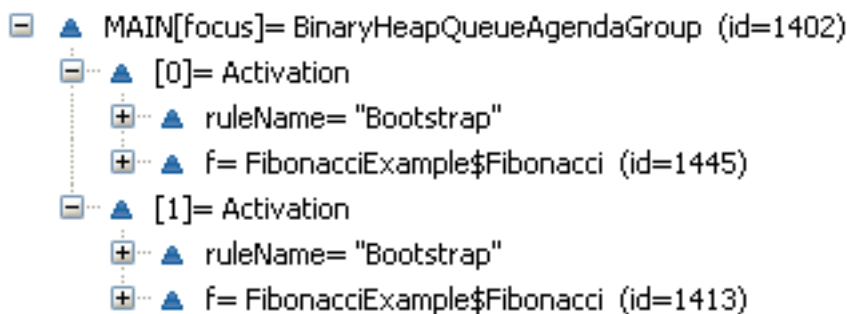


Figure 8.8. Fibonacci Example: Recurse Agenda View Two

Once there are two objects with values that do not equal **-1**, the Calculate rule will run. (Remember that it was the Bootstrap rule that set the objects with sequence values of **1** and **2** to **1**.)

At this point, there are fifty Fibonacci objects in working memory. Select a suitable "triple" to calculate the values of each in turn.

If there were three Fibonacci patterns in a rule that did not utilise field constraints to confine the possible cross-products, $50 \times 49 \times 48$ possible combinations would exist, potentially leading to 125,000 rule executions, that majority of which would be incorrect. To prevent this problem arising, the Calculate rule uses field constraints to restrict the three Fibonacci patterns to a correct order: this technique is called *cross-product matching*. This is how it works:

1. The first pattern finds any Fibonacci object with a value **!= -1** and binds both the pattern and the field together.
2. The second Fibonacci object does this too, but, in addition, it adds an extra field constraint. This is in order to ensure that its sequence is greater by one than the Fibonacci object bound to **f1**. When this rule fires for the first time, only the first two sequences have values of **1**. The two constraints ensure that **f1** references the first sequence and **f2** references sequence two.
3. The final pattern finds the Fibonacci object with a value equal to **-1** and with a sequence value one greater than that contained in **f2**.

At this point, there are three Fibonacci objects correctly selected from the available cross-products, so a value for the third object (bound to **f3**) can be calculated.

Example 8.27. Fibonacci Example: Calculate Rule

```
rule Calculate
  when
    // Bind f1 and s1
    f1 : Fibonacci( s1 : sequence, value != -1 )
    // Bind f2 and v2; refer to bound variable s1
    f2 : Fibonacci( sequence == (s1 + 1), v2 : value != -1 )
    // Bind f3 and s3; alternative reference of f2.sequence
    f3 : Fibonacci( s3 : sequence == (f2.sequence + 1 ), value == -1 )
  then
    // Note the various referencing techniques.
    modify ( f3 ) { value = f1.value + v2 };
    System.out.println( s3 + " == " + f3.value );
  end
end
```

The **Modify** statement updates the value of the object bound to **f3**. As a result, there is now another new object with a value not equal to **-1**. The very creation of this object allows the Calculate rule to re-match. It will then process the next Fibonacci number. The **Audit** view depicted in the next image summarises this process. It shows how the last execution of the Bootstrap rule modifies the Fibonacci object, triggering the Calculate rule. This then modifies another Fibonacci object allowing the Calculate rule to run again. This cycle continues until the values are set for all of the objects.

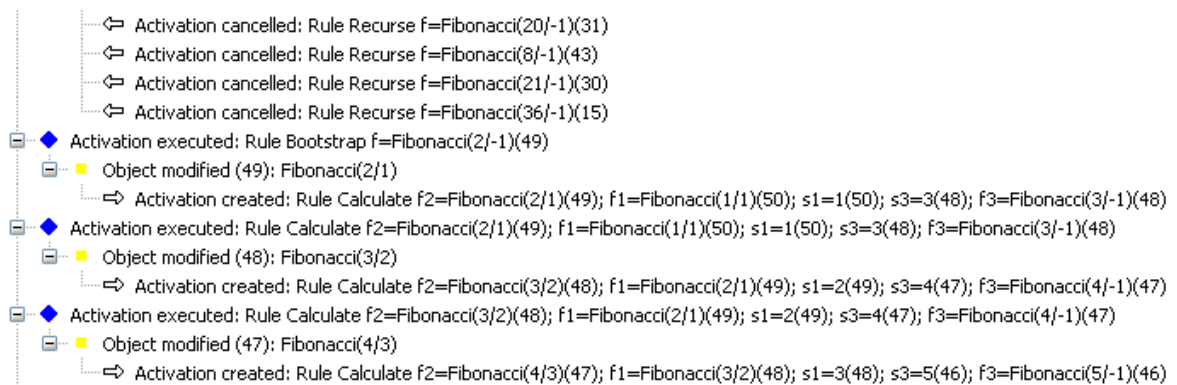


Figure 8.9. Fibonacci Example: Bootstrap Audit View

8.4. Banking Tutorial

| | |
|-------------|--|
| Name: | Banking Tutorial |
| Main class: | org.drools.tutorials.banking.BankingExamplesApp.java |
| Type: | java application |
| Rules file: | org.drools.tutorials.banking.*.drl |
| Objective: | Demonstrate pattern matching, basic sorting and calculation rules. |

This tutorial demonstrates the process of developing a complete personal banking application to handle credits and debits on multiple accounts. It uses a set of design patterns that have been created for this process.



Important

The **RuleRunner** class executes one or more DRL files against a set of data, by compiling the knowledge packages and creating a knowledge base for each execution. This is fine for testing and tutorial purposes, but it is important to realise that this is not a good solution for a production system, where the knowledge base should be built just once and then cached.

Example 8.28. Banking Tutorial: RuleRunner

```
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();
    }
}
```

The first Java class loads and executes a single DRL file, namely **Example.drl**; however, it does so without inserting any data.

Example 8.29. Banking Tutorial: Java Example One

```
public class Example1
{
    public static void main(String[] args) throws Exception
    {
        new RuleRunner().runRules(new String[]{"Example1.drl"},
            new Object[0] );
    }
}
```

This is the first simple rule to execute. It has a single eval condition this will always be **true**. This means that it will always match and "fire" at once after starting.

Example 8.30. Banking Tutorial: Rule in `Example1.dr1`

```
rule "Rule 01"
when
  eval( 1==1 )
then
  System.out.println( "Rule 01 Works" );
end
```

Here is the output, showing that the rule matches and executes the single **print** statement:

Example 8.31. Banking Tutorial: Output of `Example1.java`

```
Loading file: Example1.dr1
Rule 01 Works
```

Next, assert some simple facts and print them out:

Example 8.32. Banking Tutorial: Java Example Two

```
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.dr1" },numbers);
  }
}
```

This does not use any specific facts. Instead, it utilises a set of **java.lang.Integer** classes. (This is not considered "best practice" as a collection number is neither a "fact" nor a "thing." A bank account has a number, this being the balance it contains, therefore in that case, the account is the "fact." However, asserting integers shall suffice for the purposes of these initial demonstrations.)

Next, create a simple rule to print out these numbers:

Example 8.33. Banking Tutorial: Rule in `Example2.dr1`

```
rule "Rule 02"
when
  Number( $intValue : intValue )
then
  System.out.println("Number found with value: " + $intValue);
end
```

Once again, this is a very simple rule. It identifies any "facts" that are numbers and prints them out. Note the use of interfaces here: integers were inserted but the pattern-matching engine was able to match the interfaces and super-classes of the objects that have been asserted.

The output firstly shows the DRL being loaded, then the facts being inserted and then the rules being matched and fired. (Each inserted number is matched and fired and, thus, printed.)

Example 8.34. Banking Tutorial: Output of `Example2.java`

```
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
```

There are certainly many better ways to sort numbers than by using rules, but since one task will be to apply some **CashFlow** classes in date order later in this example, it is best to learn the process now.

Example 8.35. Banking Tutorial: `Example3.java`

```
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);
    }
}
```

This time the rule is slightly different:

Example 8.36. Banking Tutorial: Rule in `Example3.drl`

```
rule "Rule 03"
when
    $number : Number( )
    not Number( intValue <& $number.intValue )
then
    System.out.println("Number found with value: "+$number.intValue() );
    retract( $number );
end
```

The first line of the rule identifies a number and extracts the value. The second line ensures that there does not exist a smaller number than that found by the first pattern. One may be expecting to match 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.

Here is the generated output. (Note that the numbers are now sorted numerically.)

Example 8.37. Banking Tutorial: Output of `Example3.java`

```
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
```

Now is the time to start developing some personal accounting rules. The first step is to create a **Cashflow** class object.

Example 8.38. Banking Tutorial: **Cashflow** Class

```
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 + "];"
    }
}
```

The **Cashflow** class has two simple attributes: a date and an amount. A `toString` method has been added, so that it can be printed. (Note that using the **double** type for monetary units is generally *not* a good idea because the floating point form cannot represent most types of number accurately.)

There is also an "overloaded" constructor that can be used to set the values: the following example inserts five **Cashflow** objects, each possessing a different date and amount.

Example 8.39. Banking Tutorial: **Example4.java**

```
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.dr1"}, cashflows);
    }
}
```

The **SimpleDate** "convenience" class extends the **java.util.Date** class by providing a constructor that takes an input string and defines a date format. The code is listed below:

Example 8.40. Banking Tutorial: Class **SimpleDate**

```
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());
    }
}
```

Now, examine the **rule04.dr1** file in order to learn how the sorted **cashflows** were printed:

Example 8.41. Banking Tutorial: Rule in **Example4.dr1**

```
rule "Rule 04"
when
    $cashflow : Cashflow( $date : date, $amount : amount )
    not Cashflow( date < $date)
then
    System.out.println("Cashflow: "+$date+" :: "+$amount);
    retract($cashflow);
end
```

It is at this point that a **cashflow** can be identified and its date and amount extracted. In the second line of the rule, ensure that there is no **cashflow** with a date earlier than the one found. In the consequence, print the **cashflow** that satisfies the rule and then retract it, making way for the next earliest one.

Example 8.42. Banking Tutorial: Output of Example4.java

```

Loading file: Example4.dr1
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

```

The next step is to extend the **Cashflow** class to create a **TypedCashflow** (this can be either a credit or debit operation.) (Normally, it is simplest to just add this to the **Cashflow** class but here the extension will be used in order to keep the previous version of the class intact.)

```

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()
+ "];"
    }
}

```

This code can be improved in a multitude of ways, but for the sake of the example this will suffice for the present.

Next, create a class for running the code.

Example 8.43. Banking Tutorial: **Example5.java**

```

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.dr1" }, cashflows );
    }
}

```

Here, a set of **Cashflow** objects has been created, each of which is either a credit or debit operation. They have then be supplied, along with **Example5.dr1**, to the RuleEngine.

Now, examine this rule. It prints the sorted **Cashflow** objects.

Example 8.44. Banking Tutorial: Rule in **Example5.dr1**

```

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

```

It is now possible to identify a **Cashflow** fact with a type of CREDIT and extract the date and the amount. In the second line of the rule, ensure that there is no **Cashflow** of the same type with a date earlier than that which is found. In the consequence, print the **Cashflow** satisfying the patterns and then retract it, making way for the next earliest one of type CREDIT.

The output generated is described in the following example:

Example 8.45. Banking Tutorial: Output of **Example5.java**

```

Loading file: Example5.dr1
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
```

The next step is to process both the credits and debits on two bank accounts, calculating the account balances for each. In order to do this, firstly create two separate `Account` objects and inject them into the **Cashflow** class before passing them to the `RuleEngine`. (The reason for doing this is to provide easy access to the correct account without having to resort to **helper** classes.)

Study the **Account** class first. This is a simple Java object and has both an account number and a 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, extend the **TypedCashflow**, to create an **AllocatedCashflow** class, by including an **Account** reference.

Example 8.46. AllocatedCashflow Class

```
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()
        + "];"
}
}

```

Example5.java creates two `Account` objects and, with the constructor call, passes one of them into each `Cashflow`.

Example 8.47. Banking Tutorial: **Example5.java**

```

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.dr1"},cashflows);
}
}

```

Now, take look at the rule in the **Example6.dr1** file to see how each **Cashflow** should be applied in date order, then calculate and print out 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

```

Although there are now separate rules for credits and debits, do not specify a type when checking for earlier **Cashflows**. This is so that all **Cashflows** are checked in date order, regardless of type. The conditions have been used to identify the account with which to work, and the consequences have been used to update it with the **Cashflow** amount.

```

Loading file: Example6.dr1
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

| | |
|-------------|--|
| Name: | Example Policy Pricing |
| Main class: | org.drools.examples.decisiontable.PricingRuleDTEExample |
| Type: | Java application |
| Rules file: | ExamplePolicyPricing.xls |
| Objective: | Demonstrate spreadsheet-based decision tables. |

This tutorial demonstrates how a spreadsheet-based *decision table* can be used to calculate the retail cost of an insurance policy. The set of rules that are provided calculate a base price and discount for a motorist who is applying for a specific policy. The driver's age and history, along with the policy type are factors that are taken into account when determining the basic premium amount. A number of additional rules then refine this result by calculating a discount percentage.

8.5.1. Executing the Example

Open the **PricingRuleDTEExample.java** file and run it as a Java application. The following output will appear in the **Console** window:

```

Cheapest possible
BASE PRICE IS: 120
DISCOUNT IS: 20
    
```

The execution code adheres to the standard pattern: the rules are loaded, the facts are inserted and a `stateless` session is created. The difference lies in the way in which the rules are added.

```

DecisionTableConfiguration dtableconfiguration =
    KnowledgeBuilderFactory.newDecisionTableConfiguration();
    dtableconfiguration.setInputType( DecisionTableInputType.XLS );
    
```

```

KnowledgeBuilder kbuilder = KnowledgeBuilderFactory.newKnowledgeBuilder();

Resource xlsRes = ResourceFactory.newClassPathResource( "ExamplePolicyPricing.xls",
                                                       getClass() );

kbuilder.add( xlsRes,
              ResourceType.DTABLE,
              dtableconfiguration );
    
```

Note the use of the DecisionTableConfiguration object. Its input type is set to DecisionTableInputType.XLS.



Note

If one is using the **Business Rules Management System**, this is all configured automatically. Two fact types are utilized in this example, namely **Driver** and **Policy**. The default values of both are used. The **Driver** is thirty years of age, has had no prior claims and has a **LOW** risk profile. The **Policy** for which the driver is applying is **COMPREHENSIVE**. It has not yet been approved.

8.5.2. The Decision Table

In this decision table, each row represents a rule and each column represents either 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 pending 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

Study the spreadsheet shown above. Note that the **RuleSet** declaration provides the package name. Other optional items can also be added here, such as **Variables** (for global variables) and **Imports** (used to import classes.) In this case, the rules name-space is the same as that for the fact classes, so it is omitted.

Further down, there is a **RuleTable** declaration. The next words, **Pricing Bracket**, are assigned as a prefix to the name of every rule generated.

Next is the **CONDITION** or **ACTION**. This indicates the purpose of a column. In other words, it dictates whether the column will form part of the condition or the consequence of the generated rule.

Observe that the data about the motorist spans three cells. The template expressions below each fact are applied to this data. The driver's age-range data uses **\$1** and **\$2**, (populated with comma-separated values), **locationRiskProfile** and **priorClaims**, which are found in their respective columns.

The policy base price is set in the action columns. One can also log messages there.

Chapter 8. Examples

| | 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

Broad category brackets are indicated by the comments in the leftmost column. Now, use the known facts about the motorist and the policies to manually determine the base cost.



Note

The answer is Row Eighteen, (as the motorist has had no prior accidents) and, as they are thirty years of age, the base price equals **120**.

| | B | C | D | E | F | G |
|----|----------------------------|-------------|---|------------------------|--------------------------|------------|
| 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

The next step is to calculate any discount, based on the conditions listed above. The discount results from a mixture of factors, including the Age bracket, the number of prior claims and the policy type. In this example case, the driver is thirty, has had no prior claims and is applying for a **COMPREHENSIVE** policy. This results in a twenty percent discount.

**Note**

The discount information is stored in a separate table in the same worksheet. This is so that different templates can be applied.

**Important**

It is important to understand that decision tables generate rules. As a result, they do not simply employ "top-down" logic. Think of them as a means to capture the data from which the rules are created. This is a subtle difference that has confused some users. The evaluation of the rules is not necessarily in the given order, since all the normal mechanics of the rule engine still apply.

8.6. Pet Store Example

| | |
|-------------|--|
| Name: | Pet Store |
| Main class: | org.drools.examples.petstore.PetStoreExample |
| Type: | Java application |
| Rules file: | PetStore.drl |
| Objective: | demonstrate the use of agenda groups, global variables and graphical user interface integration (including callbacks from within rules.) |

This example shows how to use rules in conjunction with a program possessing a graphical user interface (in this case, it is a **Swing**-based desktop application).

Within the **Rules** file, there is an example which teaches how to use *agenda groups* and *auto-focus* functionality in order to dictate which member of a set of rules is permitted to run at a given time. This example also shows how Java and MVFLEX Expression Language dialects can be mixed together, and also demonstrates the use of **accumulate** and how Java functions can be called from within the rule-set.

All of the Java code is contained in the **PetStore.java** file. This code defines the principal classes listed below. (It also contains several unlisted minor classes which are used to handle **Swing** events.)

- **Petstore** - this contains the `main()` method.
- **PetStoreUI** - this is responsible for creating and displaying the **Swing**-based graphical user interface. It contains several smaller classes, which are mainly responsible for responding to various GUI events such as mouse and button clicks.
- **TableModel** - this holds the table data. Consider it a Java Bean that extends the **Swing AbstractTableModel** class.
- **CheckoutCallback** - this class allows the graphical user interface to interact with the rules.
- **Ordershow** - this holds the items that the user wishes to buy.
- **Purchase** - this stores details of the order and the products being bought.
- **Product** - this is a *Java Bean* and holds the pricing information and other details of the products available for purchase.



Note

Much of the code is either **Swing**-based or in the form of plain Java Beans. **Swing** will not be discussed in very much detail but a good tutorial on its use is found here on the Sun Microsystems website: <http://java.sun.com/docs/books/tutorial/uiswing/>.

Here are the pieces of Java code in the **Petstore.java** file that relate to rules and facts:

Example 8.48. Creating the Pet Store RuleBase in PetStore.main

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

kbuilder.add( ResourceFactory.newClassPathResource( "PetStore.drl",
                                                    PetStore.class ),
             ResourceType.DRL );
KnowledgeBase kbase = KnowledgeBaseFactory.newKnowledgeBase();
kbase.addKnowledgePackages( kbuilder.getKnowledgePackages() );

// Create the stock.
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 ) );

// A callback is responsible for populating the
// Working Memory and for firing all rules.
PetStoreUI ui = new PetStoreUI( stock,
                               new CheckoutCallback( kbase ) );
ui.createAndShowGUI();
```

The code shown above loads the rules from a **DRL** file that is located on the class-path. In contrast to the other examples, in which the facts are asserted and executed immediately, this time, that step is deferred until later. This is dictated by the second last line in which a **PetStoreUI** object is created using a constructor accepting the **Vector** object, **stock**. This collects the products and an instance of the **CheckoutCallback** class containing the rule-base which loaded just prior to this.

The actual Java code that fires the rules is called by the `CheckoutCallback.checkout()` method. It triggers when the user clicks the **Checkout** button.

```
public String checkout(JFrame frame, List<Product> items) {
    Order order = new Order();

    // Iterate through list and add to cart
    for ( Product p: items ) {
        order.addItem( new Purchase( order, p ) );
    }

    // 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", 5 ) );
    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();

// Return the state of the cart
return order.toString();
}

```

Two items are passed via this method. One is the handle for the *JFrame* **Swing** component that surrounds the output text frame (at the bottom of the graphical user interface.) The second is a list of order items. This list comes from the **TableModel**, which holds the information displayed in the **Table** area in the top right section of the screen.

The **for** loop transforms the list of order items into the Order Java Bean, (which is found in the **PetStore.java** file.)



Note

It is possible to refer directly to the **Swing** data-set in the rules but it is better to do it this way, using simple Java objects. As a result of following this method, one is not bound **Swing** if one wishes to transform the sample into a web application.



Note

All of the *states* depicted in this example are stored in the **Swing** components; the rules themselves are effectively "stateless." Each time the **Checkout** button is clicked, the contents of the **Swing TableModel** are copied into the session's working memory.

Within this code, there are nine calls to the working memory. The first of these creates a new working memory *stateful knowledge session* (in the Knowledge Base.) (Remember, this Knowledge Base was passed in the **CheckoutCallback** class was created in the `main()` method.)

The next two calls pass in two objects that will be held by the rules as global variables. These objects are the **Swing** text area and the **Swing** frame and are used to write messages.

More *inserts* put information on the products themselves into both working memory and the order list. The final call is the standard `fireAllRules()` method. Next, look at what this method does when the rules file:

Example 8.49. Package, Imports, Globals and Dialect - Extracts from the **PetStore.dr1**

```

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
```

The first part of the **PetStore.dr1** file contains the standard **package** and **import** statements (which are used to make various Java classes available to the rules.) In addition to these, there are the two *global variables*, namely **frame** and **textArea**. These hold the references to **Swing's** JFrame and JTextArea components. (Unlike Rules variables, which expire as soon as they have been fired, global variables retain their values for the lifetime of the session.)

The following example is taken from the end portion of the **PetStore.dr1** file. It contains two functions that are referenced by the rules. (These two functions will be the topic of study in the next section.)

```
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;
}
```



Note

Having these functions in the **Rules** file is simply to make the Pet Store example more compact. In real life scenarios, one should put the functions in a separate file, either within the same rules package or as a static method on a standard Java class. Import them in this way: **import function my.package.Foo.hello.**

The purpose of these two functions is as follows:

- `doCheckout()` displays a dialogue box that asks the user if they wish to check out. If he or she does, focus is set to the **checkOut** agenda group, giving the rules in that group the potential permission to fire.
- `requireTank()` displays a dialogue box that asks the user if they wish to buy a fish tank. If so, one is added to the order list in working memory.

The rules that call upon these functions are taught later in this tutorial. The next set of examples are, themselves, derived from the pet store rules. The first extract is that which runs first, (partly because the **auto-focus** attribute has been set to **true**.)

Example 8.50. Putting Items into Working Memory - Extract from the **PetStore.dr1** File

```

/// Insert each item in the shopping cart into the Working Memory
// 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 );
  kcontext.getKnowledgeRuntime().getAgenda().getAgendaGroup( "show items" ).setFocus();
  kcontext.getKnowledgeRuntime().getAgenda().getAgendaGroup( "evaluate" ).setFocus();
end

```

This rule matches against all orders for which the gross total (**Order.grossTotal**) has not yet been calculated. It loops for each purchase item in order. Some parts of the `Explode Cart` rule should be familiar, such as the rule name, the *salience* (which suggests the order in which rules should be fired) and the dialect, (which is set to Java.) There are also three new items in the rule:

- **agenda-group "init"** - this is the name of the agenda group. In this case, there is only one rule in the group. However, neither the Java code nor a rule consequence sets the focus to this group and, therefore, it is reliant upon the next attribute for it to be given an opportunity to fire.
- **auto-focus true** ensures that this rule, whilst being the only one in the agenda group, receives a chance to fire when `fireAllRules()` is called from the Java code.
- `drools.setFocus()` gives the focus to the `show items` and `evaluate` agenda groups in turn, permitting them to execute their rules. (In practice, all order items are looped, ensuring that they are inserted into memory. The other rules are fired each subsequent time.)

The next two listings show the rules in the `show items` and `evaluate agenda` groups.

Example 8.51. Show Items in the GUI - Extract from the `PetStore.dr1` File

```
rule "Show Items"
  agenda-group "show items"
  dialect "mvel"
  when
    $order : Order( )
    $p : Purchase( order == $order )
  then
    textArea.append( $p.product + "\n");
  end
```

The `show items` agenda group is called first. It has only one rule, which is called `Show Items` (note the difference in case.) This rule directs log details of each purchase to the text area (at the bottom of the GUI screen). (The `textArea` variable used to do this is one of the globals discussed earlier.)

The `evaluate agenda` group also gains focus from the `explode cart` code listed previously. This agenda group has two rules: `Free Fish Food Sample` and `Suggest Tank`.

Example 8.52. Evaluate Agenda Group: Extract from the `PetStore.dr1` File

```
// Free Fish Food sample when we buy a Gold Fish if we have not 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 do not
// 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 > 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 execute if:

- the user does not already have any fish food
- the user does not already have a free fish food sample
- the user's order includes goldfish.

If these conditions are met, the rule fires and a new product (called `Fish Food Sample`) is generated. It is added to the order in working memory.

Likewise, the **Suggest Tank** rule will only fire if these two conditions are met:

- the user has not already ordered a fish tank
- the user has ordered more than five goldfish products

If these conditions are met, the rule fires and calls the `requireTank()` function, which in turn presents the user with a dialogue box. Then a tank is added to the order if confirmed. Note that the rule passes the global frame variable when it calls the `requireTank()` function. This is so that the function has a handle on the **Swing** graphical user interface.

The next rule is entitled `do checkout`.

Example 8.53. Undertaking the Check-Out: Extract from the `PetStore.dr1` File

```
rule "do checkout"
  dialect "java"
  when
  then
    doCheckout(frame, drools.getWorkingMemory());
  end
```

The `do checkout` rule has no set agenda group or auto-focus attribute. As such, it is deemed part of the default agenda group, which automatically receives focus when all of the rules that had been set to receive explicit focus have completed.

There is no left-hand side to the rule, so the right-hand side will always call the `doCheckout()` function. When it does so, the rule passes the global frame variable to give the function a handle on the **Swing** graphical user interface. (As detailed above, the `doCheckout()` function displays a confirmation dialogue box to the user. If confirmed, the function then passes the focus to the `checkout` agenda group, allowing the next set of rules to execute.)

Example 8.54. Checkout Rules: Extract from the `PetStore.dr1` File

```
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 >= 10 && < 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 >= 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 it has not already been done, this accumulates the product prices into a total, puts this total into working memory, and displays it in the **Swing** Text Area (using the **textArea** global variable yet again.)
- if the gross total is between ten and twenty, **Apply 5% Discount** calculates the discounted total, adds it to working memory and displays it in the text area.
- if the gross total is more than twenty, **Apply 10% Discount** calculates the discounted total, adds it to the working memory and displays it in the text area.

That completes the summary of how the code works from a theoretical point of view. Now, one must observe what happens in practice. The file named **PetStore.java** contains a **main()** method, meaning that it can be run as a standard Java application from either the command line or within the IDE (assuming the class-path has been set correctly.)

The first screen contains the **Pet Store Demo**. It has a list of available products (on the top left), an empty list of selected products (top right), checkout and reset buttons (in the middle) and an empty system messages area (at the bottom.)

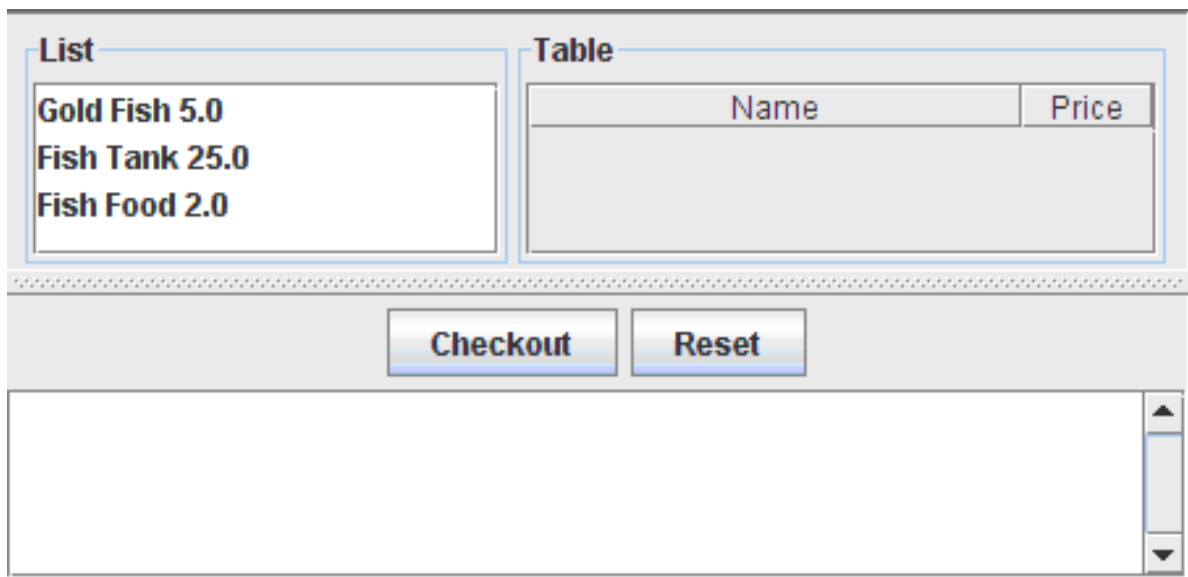


Figure 8.13. Depiction of Pet Store Demonstration Immediately After Launch

In order to reach this point, the following events have transpired:

1. The `main()` method has run and loaded the rule-base but it has not yet fired the rules. (So far, this is the only code connected to the rules in any way which has run.)
2. A new **PetStoreUI** object has been created and given a handle on the rule-base. It will use this later.
3. Various **Swing** components have performed their operations. It is only then that the above screen is shown and begins to await user input.

Click on various products from the list to see screens similar to that shown below.

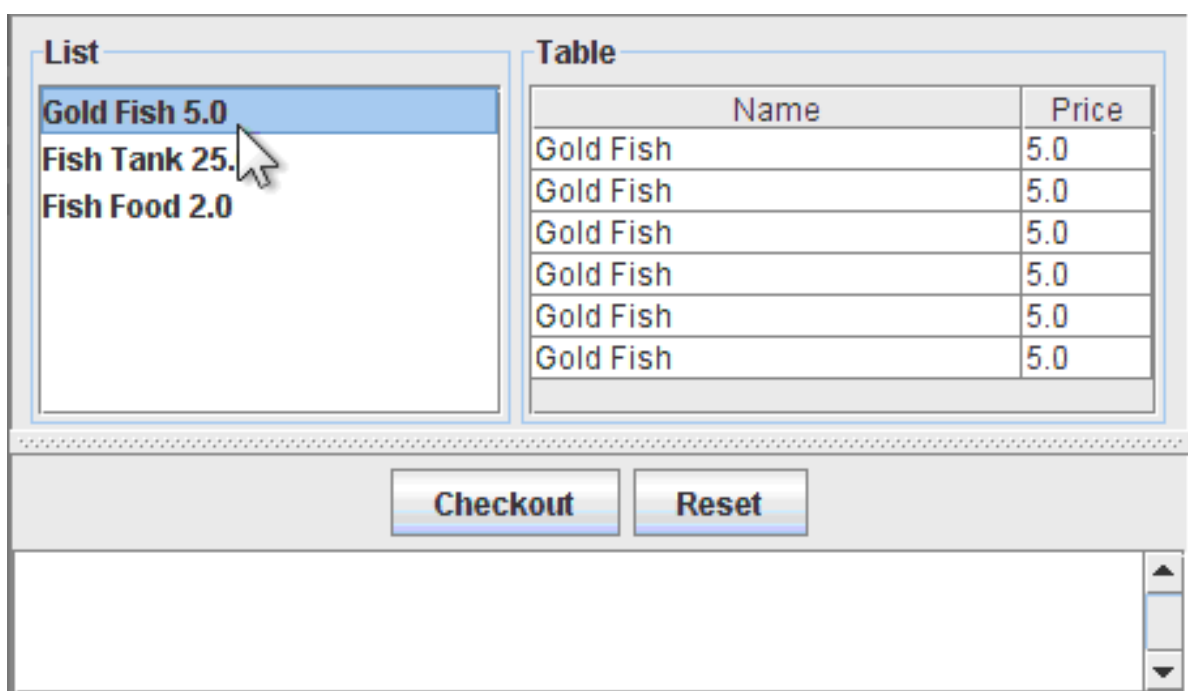


Figure 8.14. Depiction of Pet Store Demonstration with Products Selected



Note

To reiterate, no rules code has been fired yet. This is only **Swing** code, whose role it is to "listen" for mouse click events and, subsequently, add some selected products to the `TableModel` object for display in the top, right-hand section. (As an aside, note that this is a classic implementation of the *Model View Controller* design pattern.)

It is only when **Checkout** is clicked that the business rules are executed, in roughly the same order as that described earlier.

1. When the **Checkout** button is clicked, the `CheckoutCallback.checkout()` method is called by the **Swing** class. It inserts the data from the `TableModel` object (represented on the top right-hand side of the graphical user interface), and also places it in the session's working memory. It then fires the rules.
2. The `Explode Cart` rule is the first to fire, because its auto-focus setting is **true**. It loops through all of the products in the cart, ensuring that they are in the working memory. It then gives the `Show Items` and `Evaluation` agenda groups permission to fire. The rules in these groups add the contents of the cart to the text area (at the bottom of the window) and determine whether or not to give the customer free fish food and ask them if they desire to buy a fish tank. This latter step is depicted below:



Figure 8.15. Do You Want to Buy a Fish Tank?

3. The `Do Checkout` rule is the next to fire as, firstly, no other agenda group currently has focus and, secondly, it is part of the default agenda group. It always calls the `doCheckout()` function, which displays a dialogue box containing this question:

"Would you like to Checkout?"

The `doCheckout()` function sets the focus to the checkout agenda group, giving the rules in that group the option to fire.

4. The rules in the checkout agenda group display the contents of the "cart" and apply the appropriate discount.
5. **Swing** waits for user input, based upon which it will either check out more products (thus causing the rules to fire again) or close the graphical user interface, as per the final image:

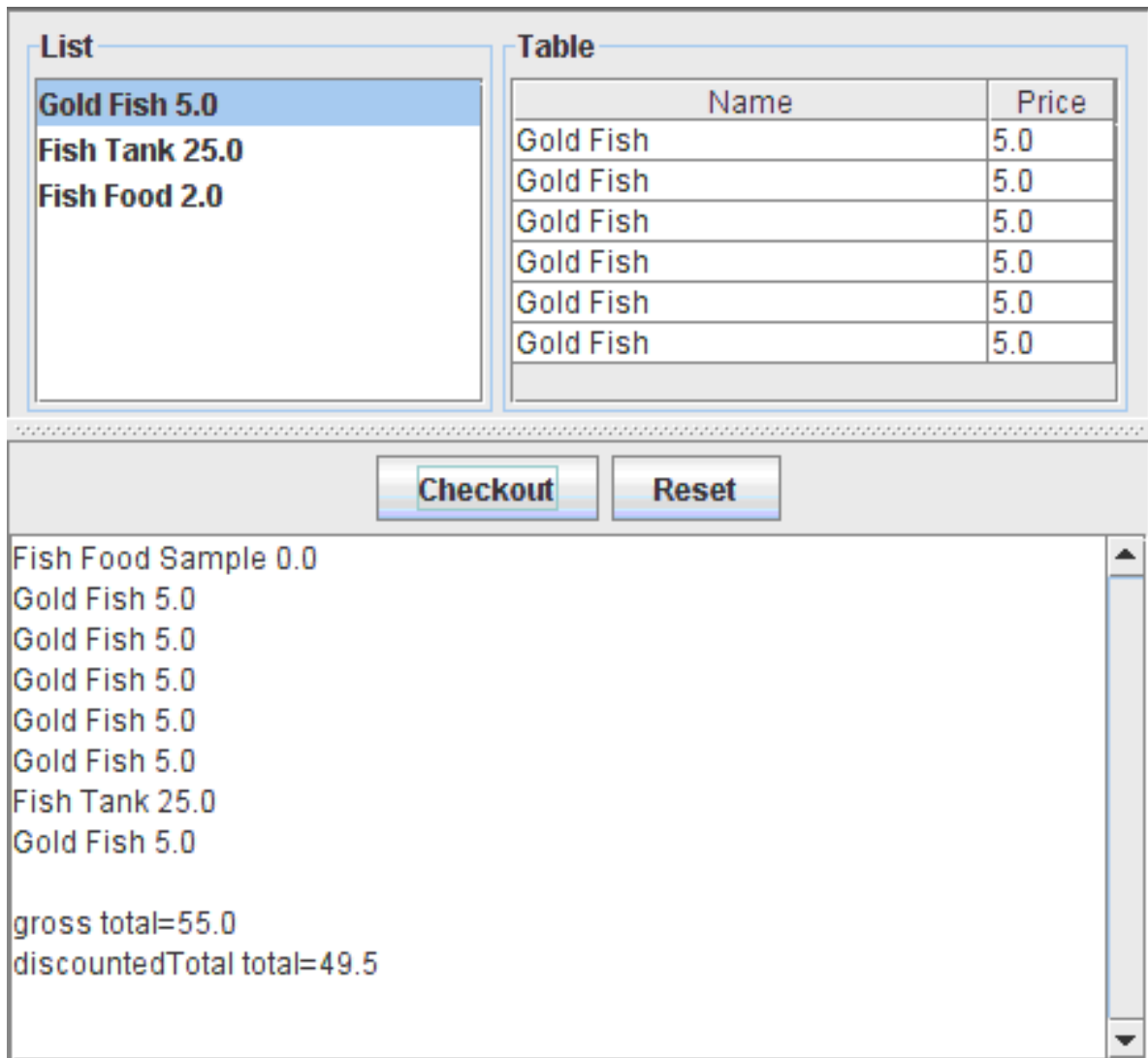


Figure 8.16. All Rules Having Fired, the Application Closes.

One could also add more `System.out` calls to demonstrate this flow of events. The output of the **Console** depicted in the above sample is mirrored in this listing:

Example 8.55. Console Output After Running the Pet Store GUI

```
Adding free Fish Food Sample to cart
SUGGESTION: Would you like to buy a tank for your 6 fish? - Yes
```

8.7. Sudoku Example

| | |
|-------------|---|
| Name: | Sudoku |
| Main class: | <code>org.drools.examples.sudoku.Main</code> |
| Type: | Java application |
| Rules file: | <code>sudokuSolver.dr1</code> , <code>sudokuValidator.dr1</code> |
| Objective: | demonstrates how to solve logic problem and shows how to employ complex pattern matching. |

This example demonstrates how to use **JBoss Rules** to find an answer in a potentially-large *solution-space* that is derived from a number of constraints. It also teaches how to integrate **JBoss Rules** with a graphical user interface-based application by using *callbacks* to update the display based on changes to the working memory at run-time.

8.7.1. Overview of Sudoku

Sudoku is a logic-based number-placement puzzle that originated in Japan. The objective is to fill a 9x9 square grid so that each column, row and 3x3 "zone" contains the digits from one to nine once and once only.

The player is presented with a partially-completed grid and given the task of completing it whilst abiding by the rules.

Each new number the player adds must be simultaneously unique in its particular row, column and 3x3 square.

8.7.2. Running the Example

Download and install **drools-examples** file as per the procedure described earlier. Having done so, execute `java org.drools.examples.sudoku.Main`. (Note that this example requires **Java 5**.) A relatively simple, partially-filled grid will appear in a window:



Figure 8.17. Partially-Filled Grid

Click on the **Solve** button and the `rules` engine will fill in the remaining values. The **Console** will display detailed information about the rules as they are executed, thereby demonstrating how the puzzle is solved.

```
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 "solving logic" rules have been activated and executed, the engine processes a second rule base. This one checks that the solution is complete and valid. In the case of the example, it finds that all is well, so, consequentially, the **Solve** button is disabled and this message appears:

Solved (1052ms)

File

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 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 |

Solved (954 ms)

Figure 8.18. Solved Grid

The example comes with a number of grids which can be loaded and solved. Click on **File**, then **Samples** and finally **Medium** to load a more challenging grid. (Note that the **Solve** button becomes enabled once more when the new grid loads.)

Having experimented with it a little, go now and load a deliberately invalid grid by clicking on **File**, then **Samples** and finally **!DELIBERATELY BROKEN!**. Have a look for some of the errors. (For example, **5** appears twice in the first row.)

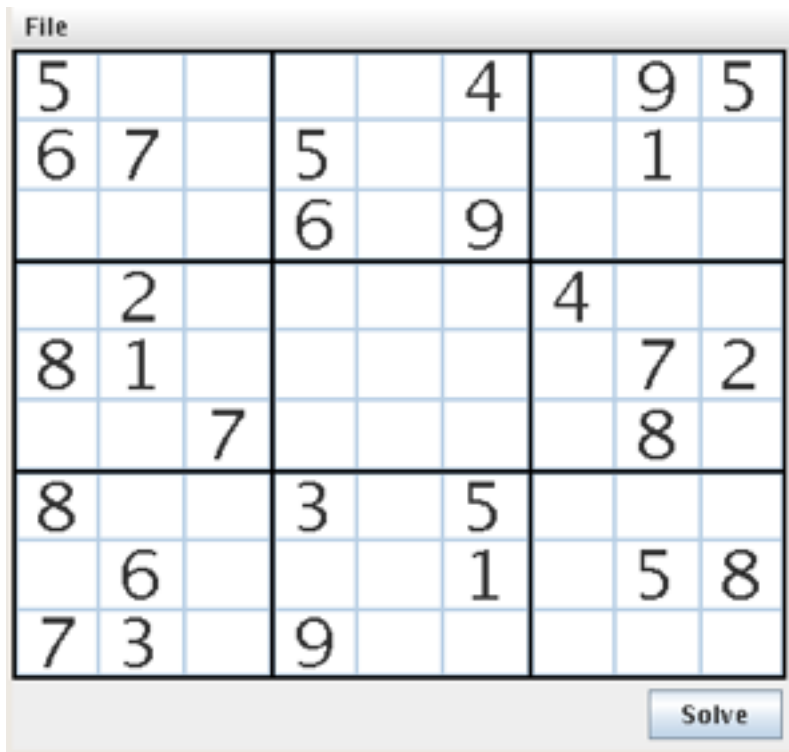


Figure 8.19. Broken Grid

Knowing that it will not work, click on the **Solve** button anyway to learn what happens when the "solving rules" are applied to this invalid grid.

Observe that the **Solve** button is relabeled to indicate that the resulting solution is invalid and that the Validation Rule Set outputs all of the problems it discovers 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)
    
```

The answers to some puzzles that are theoretically solvable cannot, in practice, be found by the engine in its current state. To see an example, click on **File**, then **Samples** and finally **Hard 3**. A sparsely-populated grid is loaded.

Now click on the **Solve** button and observe that the current rules are unable to complete the grid, even though a human being may be able to see the way to a solution.

Up until this point, the program has used a ten-rule set to deduce the solutions. This rule set must now be extended to give the engine the logic it needs to tackle more complex puzzles like this one.

8.7.3. Java Source and Rules Overview

The Java source code is found in the `/src/main/java/org/drools/examples/sudoku` directory. The two **DRL** rule definition files are located in the `/src/main/rules/org/drools/examples/sudoku` directory.

The `org.drools.examples.sudoku.swing` package contains a set of classes which implement a framework for Sudoku puzzles.



Note

This package does not depend on the **JBoss Rules** libraries.

Implement the `SudokuGridModel` interface 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 determined.)

The `SudokuGridView` is a **Swing** component. It can depict any implementation of the `SudokuGridModel` graphically.

The `SudokuGridEvent` and the `SudokuGridListener` are used to communicate model "state" changes to the view; events are fired when a cell's value is resolved or changed. This will be familiar to those readers acquainted with the model-view-controller patterns used in other **Swing** components such as `JTable`. (`SudokuGridSamples` provides a number of partially-completed puzzles to demonstrate these concepts.)

The `org.drools.examples.sudoku.rules` package contains an implementation of the `SudokuGridModel` based on **JBoss Rules**. Two Java objects are used, both of which extend `AbstractCellValue` and both of which represent a value for a specific cell in the grid. This value includes the row and column location of the cell, an index number for the 3x3 zone in which it is contained and, thirdly, and the actual number held in the cell.

`PossibleCellValue` indicates that the value of a cell is currently unknown. (There can be between two and nine possible values for any given cell.)

`ResolvedCellValue` indicates that the value of the cell has been determined. There can be only one resolved value for a given cell.

`DroolsSudokuGridModel` implements the `SudokuGridModel`. It is responsible for converting an (initially two-dimensional) array of partially-specified cells into a set of `CellValue` Java objects. Doing so creates a working memory session based on the `solverSudoku.drl` rules file. It also inserts the `CellValue` objects into working memory.

When the `solve()` method is called it, in turn, calls `fireAllRules()`, which attempts 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 callback allows the implementation to fire a `SudokuGridEvent` to its `SudokuGridListener` clientele, updating them in real time.

Once all of the rules fired by the "solver" working memory have run, the `DroolsSudokuGridModel` executes a second set of rules. (These come from the `validatorSudoku.drl` file). They work with the same set of Java objects and their purpose is to determine if the resulting grid is a valid and a complete solution.



Note

The `org.drools.examples.sudoku.Main` class implements a Java application that combines the components described above.

The `org.drools.examples.sudoku` package contains two **DRL** files. These are, namely, `solverSudoku.drl`, (which defines the rules used to solve a Sudoku puzzle) and `validator.drl`, (which defines the rules which test as to whether or not the current state of the working memory represents a valid solution.) Both of these rule-sets use the `PossibleCellValue` and `ResolvedCellValue` objects as their facts. Also, they both output data to the **Console** window as they run.



Note

In a real-life situation, one would insert logging information and utilize the `WorkingMemoryListener` to display the output to users, rather than use the **Console** in this manner.

8.7.4. Validation Rules

This is the process that the validation rules found in the `validatorSudoku.drl` file follow:

1. 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.)
2. The other three rules match all of the `ResolvedCellValue` objects and bind them to the variable entitled `$resolved1`. They then look for the `ResolvedCellValue`s that both contain the same value and are located, respectively, in the same row, column and 3x3 zone.
3. If these rules are fired they add a message, describing the reason why the solution is invalid, to a global list of strings. The `DroolsSudokuGridModel` injects this list before it runs the rule-set. (It also checks whether or not the list is empty after having called `fireAllRules()`. If it is not empty, then it prints all of the strings in the list and sets a flag to indicate that the grid has not been solved.)

8.7.5. Solving Rules

This is the somewhat more complex process followed by the "solving" rules found in the `solverSudoku.drl` file:

1. Rule #1 clears the working memory of any `PossibleCellValue`s that correspond with rows and columns holding invalid answers. Several of the other rules insert `ResolvedCellValue`s into the working memory at specific rows and columns after they have determined that a given cell must have a certain value.

Because of the importance of this rule, it is given a higher salience than the others. This ensures that, as soon as the left-hand side is **true**, the activations are moved to the top of the agenda and fired. This, in turn, prevents the other rules from firing spuriously.

This rule also runs `update()` on the `ResolvedCellValue`. (This happens even though the rule has not been modified to make **JBoss Rules** send the event to any `WorkingMemoryListeners`

attached to the `working` memory. Firing such an event would enable them to update themselves.) This makes the graphical user interface display the new state of the grid.

2. Rule #2 identifies cells in the grid for which there can be only one possible value. The first line of the **when** clause matches all of the `PossibleCellValue` objects in the `working` memory. The second line demonstrates use of the **not** keyword.



Note

This rule only fires if there are no other **PossibleCellValue** objects in the same row and column with differing values.

When it fires, the single `PossibleCellValue` in that row and that column is retracted from the `working` memory and replaced with the `ResolvedCellValue` of the same value.

3. Rule #3 removes `PossibleCellValue`s from a row if they are the same value as a `ResolvedCellValue`. In other words, when a cell contains a resolved value, any other cells in the same row containing the same value must be cleared so as to adhere to the rules of the puzzle. The first line of the **when** clause finds all of the **ResolvedCellValue** objects in the `working` memory. The second line locates `PossibleCellValue`s 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 them since they cannot be the correct solutions for those cells after all.
4. Rules #4 and #5 act in the same way as Rule #3 but check for columns and zones for the redundant `PossibleCellValue`s that clash with `ResolvedCellValue`s.
5. Rule #6 checks for the scenario whereby a cell's possible value appears only once in a given row. The first line on the left-hand side matches against all `PossibleCellValue` facts in the `working` memory and stores the results in a number of local variables. The second line checks that no other `PossibleCellValue` objects with the same value exist on the same row. The third to fifth lines check that there is not a `ResolvedCellValue` with the same value in the same zone, row or column. (This so that this rule does not fire prematurely.)



Note

As an aside, one could also remove lines three to five and give Rules #3, #4 and #5 a higher salience to make sure they always fire before Rules #6, #7 and #8.

When the rule fires, **\$possible** must represent the value for the cell; so retract it and replace it with the equivalent `ResolvedCellValue`. (This is the same process as that detailed in Rule #2.)

6. Rules #7 and #8 act in the same way as Rule #2 but check for single `PossibleCellValue`s in a given column and zone of the grid respectively.
7. Rule #9 is the most complex. It implements the logic that dictates, "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 do know that these two values must be in these two cells. Hence we can remove the possibility of them occurring 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 or zone respectively.

In order to solve more difficult grid puzzles, extend the rule-set further by coding custom laws that encapsulate more complex reasoning. The following section provides some suggestions on ways to do this.

8.7.6. Suggestions for Future Developments

There are a number of ways in which this example could be developed. The reader is encouraged to consider the following propositions as exercises to try in order to build skills.

- Agenda Groups*: use this *declarative tool for phased execution*. In this example, it is easy to see there are two phases, namely "resolution" and "validation". At present, they are executed via two separate rule base files. It would be a better practice to define agenda groups for all the rules, by splitting them into "resolution" and "validation" categories. They would then all be loaded from a single rule-base. The engine would execute them one immediately after the other.
- Auto-focus*: use this method to handle exceptions to regular rule execution. In the present case, if an inconsistency is found in either the input data or the resolution rules, it would be better to report it immediately than waste time continuing to run the rules futilely. To do this, having already created the single rule-base, simply define the auto-focus attribute for every rule used to validate puzzle consistency.
- Logical insert*: an inconsistency only exists whilst wrong data is in working memory. As such, one can state that the validation rules *logically insert* inconsistencies. (As soon as the offending data is retracted, the inconsistency no longer exists.)
- `session.iterateObjects()`: at the moment, a global list is used to record problems but it would be more interesting to ask the stateful session to call the desired issues by via `session.iterateObjects(new ClassObjectFilter(Inconsistency.class))`. Using the **inconsistency** class also enables one to paint the offending cells a colour (such as red), making them easy to spot.
- `kcontext.getKnowledgeRuntime().halt()`: even if the software reports the error as soon as it is found, one needs a way to tell the engine to stop evaluating rules. Create one by programming a rule that, in the presence of inconsistencies, calls the `halt()` code.
- Queries*: look at the `getPossibleCellValues(int row, int col)` method in the **DroolsSudokuGridModel**. One can see that it iterates over all `CellValue` objects as it searches for the few that it actually wants. This process can be made more efficient by the use of a **JBoss Rules** query. Simply define a query to return the wanted objects and cleanly iterate over it. (Define other queries as they are needed.)
- Globals as services*: the main objective of this change is to facilitate that which follows, but it is also useful in its own right. In order to learn the use of *globals as services*, one should, ideally, first have a callback configured. This change means that each rule that finds the `ResolvedCellValue` for a given cell can "call" the graphical user interface and update it, providing the user with immediate feedback. Also, the number in the last cell found can be "painted" a different colour so that the conclusions of the different rules can be identified quickly.
- Step-by-step execution*: having set the callback so that immediate user feedback is given, one can make use of **JBoss Rules's** *restricted run* feature. Add a button to the graphical user interface, that, when clicked, executes a single rule by calling `fireAllRules(1)`. This way, users will be able to see what the engine is doing on a "step-by-step" basis.

8.8. Number Guess

| | |
|-------------|---|
| Name: | Number Guess |
| Main class: | org.drools.examples.numberguess.NumberGuessExample |
| Type: | Java application |
| Rules file: | NumberGuess.drl |
| Objective: | to demonstrate the use of a rule-flow to organize rules. |

Study the Number Guess example in this section to learn how to use rule-flow, which provides a way of controlling the order in which rules are fired. Standard work-flow diagrams are used to clearly indicate the order in which groups of rules are to be executed.

Example 8.56. Creating the Number Guess Rule-Base

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

kbuilder.add( ResourceFactory.newClassPathResource( "NumberGuess.drl",
    ShoppingExample.class ), ResourceType.DRL );

kbuilder.add( ResourceFactory.newClassPathResource( "NumberGuess.rf",
    ShoppingExample.class ), ResourceType.DRF );

final KnowledgeBase kbase = KnowledgeBaseFactory.newKnowledgeBase();
kbase.addKnowledgePackages( kbuilder.getKnowledgePackages() );
```

The process of creating this package and the loading of the rules (via the `add()` method) is identical to that for the previous examples, with one optional further step: to specify the ability to use different rule-flows with the one knowledge base, append the **NumberGuess.rf** line to the current rule-flow. (Omitting this step simple means that the the knowledge base will be created in the same manner as before.)

Example 8.57. Starting the Rule-Flow

```
final StatefulKnowledgeSession ksession =
    kbase.newStatefulKnowledgeSession();
KnowledgeRuntimeLogger logger = KnowledgeRuntimeLoggerFactory.newFileLogger
    (ksession, "log/numberguess");

ksession.insert( new GameRules( 100,5 ) );
ksession.insert( new RandomNumber() );
ksession.insert( new Game() );

ksession.startProcess( "Number Guess" );
ksession.fireAllRules();

logger.close();
ksession.dispose();
```

Once generated, use the knowledge base to obtain a "stateful" session. Into this session insert the *facts* (this is another term for standard Java objects.) For the sake of simplicity, in this sample, all of these classes are contained within a single file, entitled **NumberGuessExample.java**:

- the **GameRules** class provides the maximum range and the number of guesses allowed
- the **RandomNumber** class automatically generates a number between zero and one hundred and makes it available to the rules after insertion via the `getValue()` method.
- the **Game** class tracks the guesses that have been made previously, and the number of guesses being made currently.



Important

Before calling the standard `fireAllRules()` method, start the process that was loaded earlier (via the `startProcess()` method).



Note

In a real-life scenario, one would make further use of the objects in their final state (for instance, the number of guesses, so that this figure could be added to a table of the highest scores.) However, for the purposes of this tutorial, it is enough simply to ensure that the working memory session is cleared by a call to the `dispose()` method.

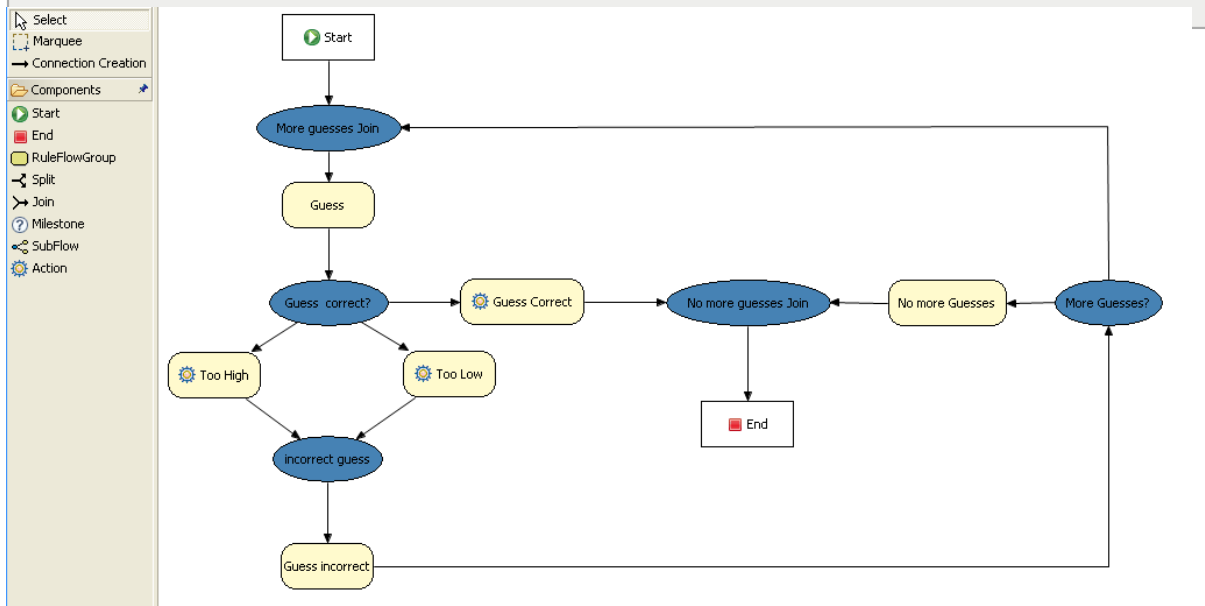


Figure 8.20. Rule-Flow for the Number Guess Example

To see the above diagram, open the **NumberGuess.rf** file in the **JBoss Rules IDE**. At first glance, it can be seen that this image is very much like a standard flowchart, with icons that are very similar (though not identical) to those found in the **JBoss jBPM Workflow** application.

To edit the diagram, use the menu of components found to the left of screen. This area is called the *palette*. Diagrams are saved in a human-readable XML format, through `X-Stream`.

Open the **Properties** view if it is not already visible. Do so by selecting **Window > Show View > Other**, and then clicking on the **Properties** view. Do this *before* selecting any item on the rule-flow (or after clicking on a blank space in the rule-flow.) The following set of properties will then be available:

| Property | Value |
|---------------|---------------|
| Id | Number Guess |
| Name | Number Guess |
| Router Layout | Shortest Path |
| Version | |
| | |

Figure 8.21. Properties for the Number Guess Rule-Flow



Note

It is a good idea to continually observe the **Properties** view whilst working through the rest of this tutorial as it provides valuable information. At this stage, it is displaying the identity number for the rule-flow process that was started when the `session.startprocess()` method was initially run.

These are the node types in the Number Guess rule-flow:

- the **start** and **end** nodes (the green arrow and red box respectively) indicate where the rule-flow starts and ends.
- the **RuleFlowGroup** (indicated by the plain yellow box maps to the RuleFlowGroups in the **DRL** file discussed later in this section. For example, when the flow reaches the Too High RuleFlowGroup, only those rules marked with the complementary ruleflow-group Too High attribute will be allowed to potentially fire.
- **action nodes** (indicated by the yellow boxes with cog-like emblems) can perform standard Java method calls. Most of the action nodes in this example call `System.out.println` to give the user an indication of what is occurring.
- **split** and **join nodes** (blue ovals) such as Guess Correct and More Guesses Join indicate where the flow of control can split (subject to various conditions) and/or be rejoined.
- **arrows** indicate the direction of flow between the various nodes.

These various nodes work in conjunction with the rules to form the Number Guess Game. For example, the Guess RuleFlowGroup permits only the Get User Guess rule to fire (more details below) as that is the only rule to possess the matching attribute of ruleflow-group "Guess".

Example 8.58. Example of a Rule that Will Only Fire at a Specific Point in the Rule-Flow

```
rule "Get user Guess"
  ruleflow-group "Guess"
  no-loop
  when
    $r : RandomNumber()
    rules : GameRules( allowed : allowedGuesses )
    game : Game( guessCount &lt; allowed )
    not ( Guess() )
  then
    System.out.println( "You have " + ( rules.allowedGuesses - game.guessCount )
```

```

        + " out of " + rules.allowedGuesses
        + " guesses left.\nPlease enter your guess from 0 to "
        + rules.maxRange );
    br = new BufferedReader( new InputStreamReader( System.in ) );
    i = br.readLine();
    modify ( game ) { guessCount = game.guessCount + 1 }
    insert( new Guess( i ) );
end

```

The rest of this rule is relatively straightforward. The left-hand side (the **when** section) dictates that it will be activated for every *RandomNumber* object is inserted into the *working memory*, where **guessCount** is less than the number of **allowedGuesses** (read from the **GameRules** class) and where the user has not yet guessed the correct number.

The right-hand side (known as the *consequence*, and featuring the keyword **then**) prints a message for the user, then awaits the user's response to arrive via `System.in`. After receiving this input (as `System.in` blocks until the **return** key is pressed) it updates/modifies the guess count and the actual guess and makes both available in the *working memory*.

The rest of the **Rules** file is rather straightforward: the package declares the dialect is set to MVEL and various Java classes are imported. In total, there are five rules in this file:

1. Get User Guess (which is the rule examined above.)
2. a rule to record the highest guess.
3. a rule to record the lowest guess.
4. a rule to inspect the guess and retract it from memory if it is incorrect.
5. a rule that notifies the user that all guesses have been used.

One point of integration between the standard rules and the *rule-flow* is via the *ruleflow-group* attribute. A second is that between the **DRL** rules file and the **rule-flow.rf** files. The **split nodes** (the blue ovals) can use values in *working memory* (as updated by the rules) to decide which flow of action to take. To see how this works, click on **Guess Correct Node**. From within the **Properties** view, open the **Constraints Editor** (by pressing the button at the right that appears after one has clicked on the **Constraints** property line). The following will appear:

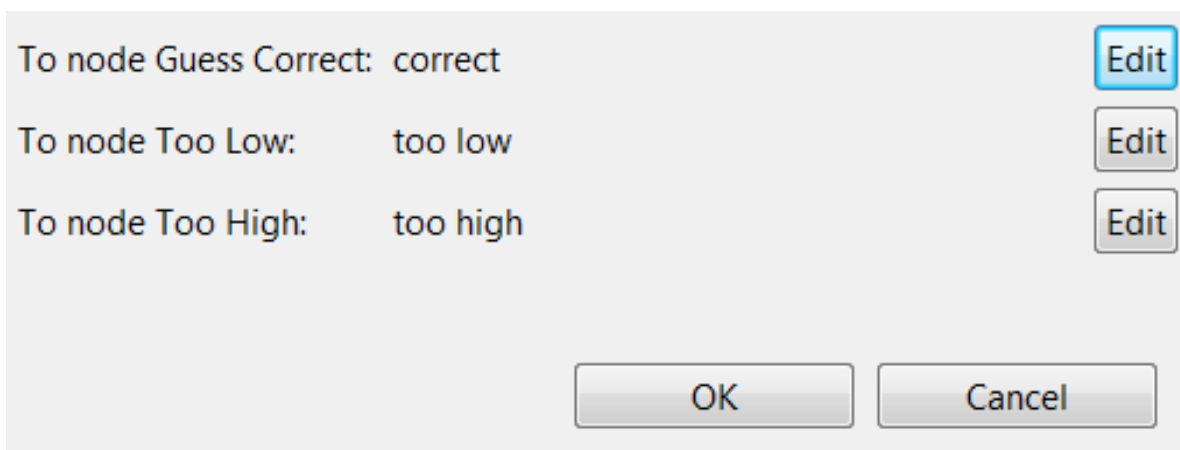


Figure 8.22. Edit Constraints for the GuessCorrect Node

Click on **Edit**. The dialogue box below shall appear. The values in the **Textual Editor** follow the standard left-hand side rule format and can refer to objects in working memory. The consequence (coded on the right-hand side) is that the flow of control will follow this node if the left-hand expression is found to be **true**.

Name:

Priority:

Always true

Textual Editor

```
RandomNumber( randomValue : value ) &&
Guess( value > randomValue )
```

Figure 8.23. **Constraints Editor** for the `GuessCorrect` Node

Since the `NumberGuess.java` example contains a `main()` method, it can be run as a standard Java application (either from **BASH** or via the **IDE**.) A typical game will feature the following interaction (the numbers are typed in by the user):

Example 8.59. Example Console Output for a Game in Which the Number Guess Program Beats the Human

```
You have 5 out of 5 guesses left.
Please enter your guess from 0 to 100
50
Your guess was too high
You have 4 out of 5 guesses left.
Please enter your guess from 0 to 100
25
Your guess was too low
You have 3 out of 5 guesses left.
Please enter your guess from 0 to 100
37
Your guess was too low
You have 2 out of 5 guesses left.
Please enter your guess from 0 to 100
44
Your guess was too low
You have 1 out of 5 guesses left.
Please enter your guess from 0 to 100
47
Your guess was too low
```

```
You have no more guesses  
The correct guess was 48
```

In summary, here are the key points:

1. **NumberGuessExample.java**'s `Main()` method loads the `RuleBase`, obtains a `StatefulSession` and inserts the `Game`, `GameRules` and `RandomNumber` (the latter containing the target number) objects into it. This method sets the process flow to be used and fires all of the rules. Control then passes to the `RuleFlow`.
2. The **NumberGuessExample.rf** `RuleFlow` begins on the `Start` node.
3. Control passes to the `Guess` node, transiting through the `More Guesses` join node.
4. At the `Guess` node, the `Get User Guess RuleFlowGroup` is enabled. In this case, the `Guess` rule (found in the **NumberGuess.dr1** file) is triggered. It displays a message to the user, takes the response, and puts it into memory. Control then flows on to the next node, `Guess Correct`.
5. At this node, constraints inspect the current session and determine the path to take.

If the guess in Step Four was too high or too low, flow proceeds along a path which has both:

- an action node which uses normal Java code to print one of the following two statements:

```
Too high
```

```
Too low
```

- a `RuleFlowGroup` which causes a highest- or lowest-guess rule to be triggered from within the **Rules** file.

Flow passes from these nodes to that designated in Step Six.

If the guess in Step Four is correct, an action node with normal Java code prints this statement:

```
You guessed correctly.
```

There is a join node here (just before the `Rule Flow End`) so that the `no-more-guesses` path (Step Seven) will also terminate the `RuleFlow`.

6. Control passes to the `RuleFlow` via a join node and then onto a `Guess Incorrect RuleFlowGroup` item (which triggers a rule to retract the guess from working memory.) It then moves onto the `More Guesses` decision node.
7. The `More Guesses` decision node (the right-hand side of the rule-flow) uses constraints (by again looking at values that the rules have placed in working memory) to decide if more guesses are still to be made available and if so, skips back to Step Three. If there are none left, control proceeds to the end of the the work-flow, via a `RuleFlowGroup` that triggers a rule which states:

```
You have no more guesses
```

8. The loop of Steps Three to Seven continues until the number is guessed correctly, or the number of guesses available is exhausted.

8.9. Miss Manners and Benchmarking

| | |
|-------------|---|
| Name: | Miss Manners |
| Main class: | <code>org.drools.benchmark.manners.MannersBenchmark</code> |
| Type: | Java application |
| Rules file: | <code>manners.dr1</code> |
| Objective: | Advanced walk-through of the Manners benchmark, covering <i>depth conflict resolution</i> . |

8.9.1. Introduction

Miss Manners is throwing a party and, being a good hostess, she wants to arrange seating appropriately. Her initial design arranges her guests in male-female pairs but then she worries that people may not have mutual topics of interest to discuss. What is a good hostess to do? She decides to note the hobby of each guest. She can then arrange them by alternating gender and ensure that everybody is seated, on at least one side, next to someone with whom they have a hobby in common.

8.9.1.1. Bench-Marking Scripts

- *Miss Manners* uses a *depth-first* search approach to determine the seating arrangements. It alternates the placement of ladies and gents, whilst ensuring one mutual hobby between neighbours.
- *Waltz* establishes a three-dimensional interpretation of a line drawing. It does so by labeling lines and using *constraint propagation*.
- *WaltzDB* is a more generalised version of *WaItz*, in that it supports junctions of more than three lines and it uses a database.
- *Automatic Route Planner (ARP)* is a route planner that has been designed for a robotic vehicle. It uses the A* search algorithm.
- *Weavera* is a *Very Large Scale Integration (VLSI)* router for channels and boxes. It uses a *blackboard* technique.



Note

Miss Manners is the rule engine industry's de facto standard bench-marking test. Its behaviour, however, is now well-known so many engines have been optimised to run it efficiently, thereby negating its usefulness. For this reason, *WaItz* is becoming more and more popular.

8.9.1.2. Miss Manners' Execution Flow

After the first seating arrangement has been assigned, the system executes the depth-first recursion code. This repeatedly processes the correct seating arrangements until the last seat is assigned. Miss Manners uses a context instance to control execution flow. The following activity diagram is partitioned in order to show the relationship between the rule execution and the current state of the context.

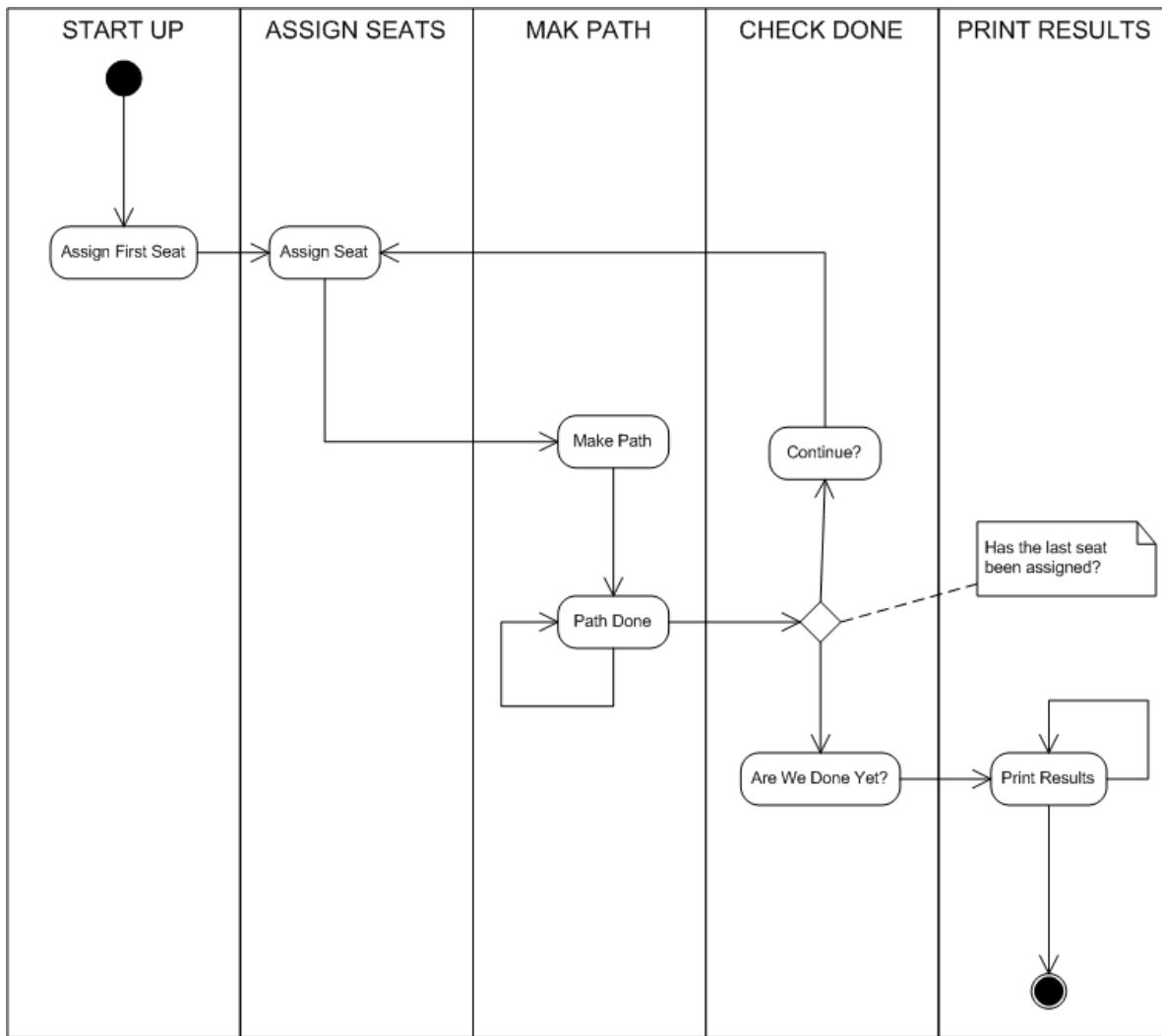


Figure 8.24. Miss Manners Activity Diagram

8.9.1.3. The Data and the Results

Before exploring the rules in detail, take a look at the asserted data and the resulting seating arrangement. The data is a simple set of five guests who are to be arranged so that genders alternate and neighbours have a common hobby.

8.9.1.4. The Data

The data is given in OPS5 (*Official Production System 5*) syntax. Each attribute possesses a parenthesised list of name- and value-pairs. Each person has only one hobby.

```

(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) )
  
```


8.9.1.5. The Results

Each line of the results list is printed when the `Assign Seat` rule is executed. The key element to which one should pay attention is that each line has a *pid value* one greater than that preceding it. (The significance of this will be explained in the discussion of the `Assign Seat` rule later on in this section.) The **ls**, **rs**, **ln** and **rn** abbreviations refer to the seats to the left and right and the names of the neighbours in these positions. The actual implementation uses longer attribute names (such as **leftGuestName**, but in this *Guide* the notation used in the original implementation is retained.

```
[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 Analysis

8.9.2.1. Cheating

`Miss Manners` has been designed to exercise *cross-product joins* and **Agenda** activities. Sometimes people, failing to understanding this, try to "tweak" the example in an attempt to achieve better performance. This makes their port of the `Manners` benchmark pointless. Here is a list of known cheats and porting errors for `Miss Manners`:

- using arrays for a guest's hobbies, instead of asserting each one as a single fact. This massively reduces the cross products.
- altering the sequence of data. This reduces the amount of matching, thereby increasing execution speed.
- changing the **not** conditional element so that the test algorithm only uses the first-best-match code. This is, basically, transforming the test algorithm so that it performs *backward-chaining*. The results of this are only comparable to other backward-chaining rule engines or ports of `Miss Manners`.
- removing the context so that the rule engine matches the guests and seats prematurely. A proper port will prevent facts from matching because of the `context start`.
- preventing the rule engine from performing *combinational* pattern-matching.
- the port is flawed if no facts are retracted in the *reasoning cycle* as a result of **NOT CE**.

8.9.2.2. Assign First Seat

Once the context changes to **START_UP**, activations are created for every asserted guest. Because each and every activation is created as the result of a single working memory action, they all have the same `activation time tag`. (The last guest to be asserted will have a higher `fact time tag`.)



Note

The execution order has little bearing upon this rule but has a significant impact upon the `Assign Seat` rule.

The activation fires and asserts the first seating arrangement and a path. It then sets the Context attribute's state to create an activation for the findSeating rule.

```
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
```

8.9.2.3. "findSeating" Rule

The findSeating rule determines the seating arrangements. When run, it generates cross-product solutions for every asserted seating arrangement against each asserted guests with the exception of itself and 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

```

```

=>[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]

```



Note

Creating redundant activations might seem pointless but it must be remembered that good rule design was not a motive in creating Miss Manners ; it was intentionally designed as a badly written rule-set to fully stress-test the cross-product matching process and the **Agenda** functionality.



Note

Be aware that each activation has an identical time tag value of **35**. This is because they were all activated by the change in the Context object to **ASSIGN_SEATS**. With OPS5 and LEX, it would correctly fire the activation with the seating asserted last. With Depth, the accumulated fact time tag ensures that the activation for the last seat to be asserted fires.

8.9.2.4. The "makePath" and "pathDone" Rules

The makePath rule must always execute before pathDone. A path object is asserted for each seating arrangement, up to the very last. (Note that the conditions in pathDone are a subset of those in makePath, which may lead one to wonder how it is that makePath can be ensured to execute 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
```

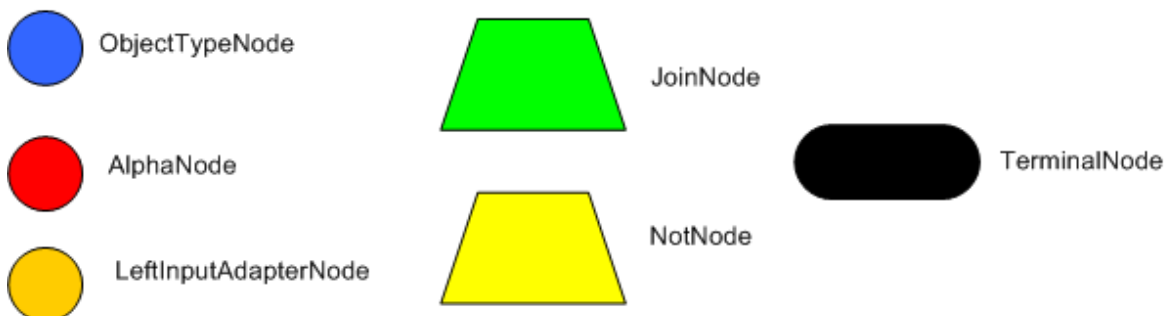
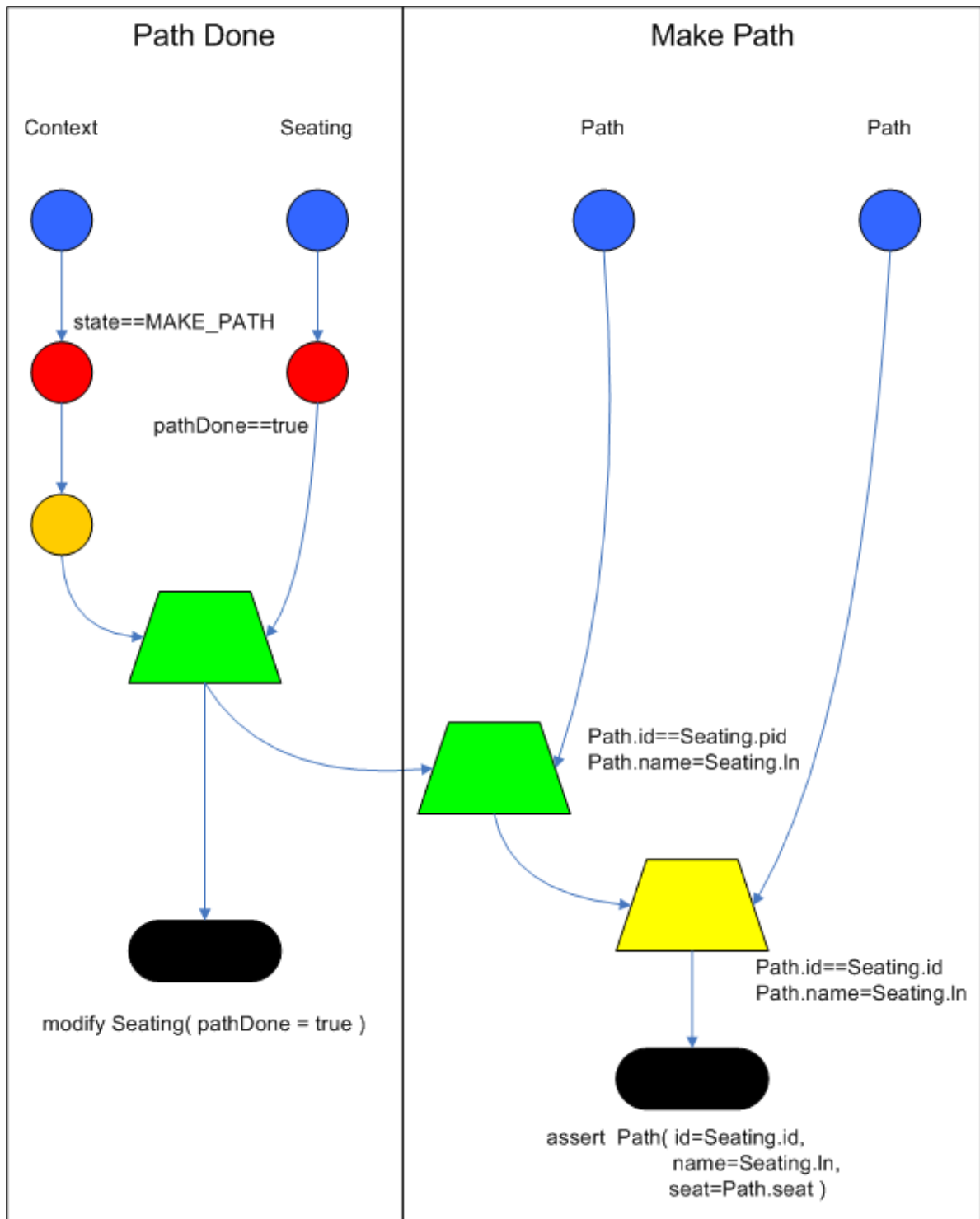


Figure 8.25. Rete Diagram

Both rules end up conflicting in the **Agenda**. Both will have identical activation time tags but the accumulate fact time tag is greater for the makePath rule so it is given precedence.

8.9.2.5. The "Continue" and "Are We Done?" Rules

Are We Done only activates when the last seat is assigned, at which point both rules are executed. For the same reason that makePath always has precedence over pathDone, Are We Done has 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. Summary of Output

```
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]
```

Having studied this section, the reader now knows how to the Miss Manners's bench-marking script works and some of the caveats and pitfalls of which to be aware when using it.

8.10. Conway's Game Of Life Example

| | |
|-------------|--|
| Name: | <i>Conway's Game Of Life</i> |
| Main class: | org.drools.examples.conway.ConwayAgendaGroupRun org.drools.examples.conway.ConwayRuleFlowGroupRun |
| Type: | Java application |
| Rules file: | conway-ruleflow.dr1 conway-agendagroup.dr1 |

| | |
|------------|--|
| Objective: | Demonstrates accumulate, collect and from. |
|------------|--|

Conway's Game Of Life is a well-known simulation model. The application presented here is a **Swing**-based implementation of the it. The laws governing the *Game* are implemented using **JBoss Rules**. Read this document to learn how this implementation works.

First, look at the grid shown below. It helps one to visualise the game, by showing the "arena" in which the life simulation takes place. Initially the grid is empty, meaning that there are no live cells in the system. Each cell is either "alive" or "dead," with living cells depicted as green balls. Pre-selected patterns of live cells can be chosen from the **Pattern** drop-down list. (Alternatively, individual cells can be doubled-clicked, which has the effect of toggling them between "live" and "dead" states.)

It is important to understand that each cell is related to its neighbours. This is fundamental to the *Game's* rules. "Neighbours" include not only cells to the left, right, top and bottom but also those cells that are connected diagonally. Thus, each cell has a total of eight neighbours. The exceptions are the four corner cells which have only three neighbours apiece, and the cells along the four borders, which each have only five neighbours.

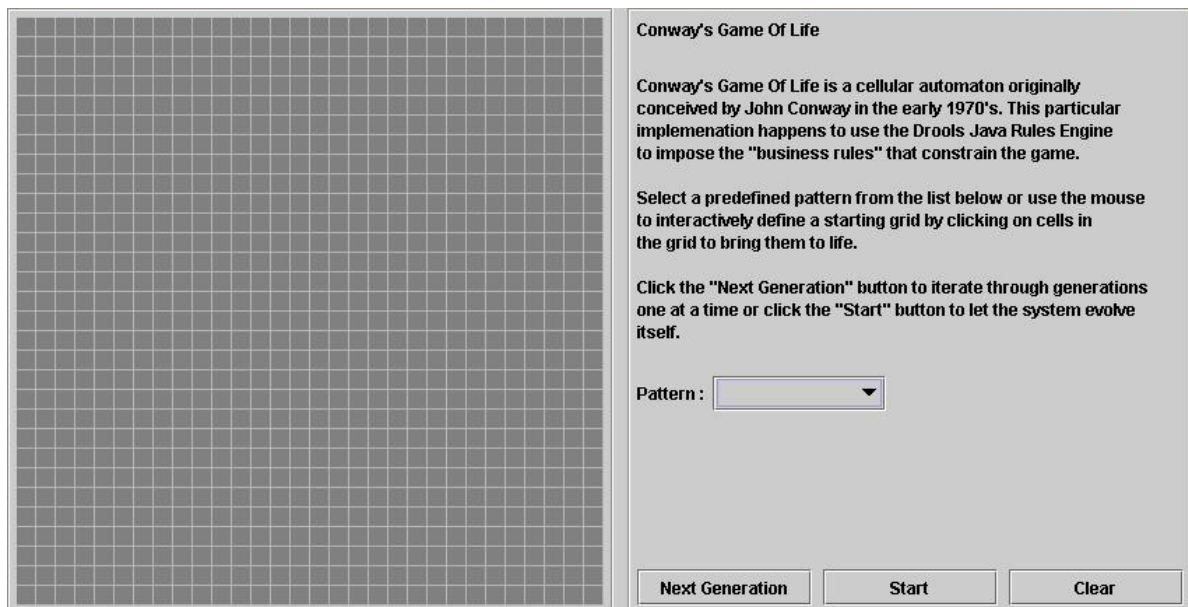


Figure 8.26. Starting a New Game

For each generation (a complete iteration and evaluation of all cells), the system evolves and cells may be born or killed. There are a very simple set of rules that govern what the next generation will look like.

- if a living cell has fewer than two live neighbours, it dies of loneliness.
- if a living cell has more than three live neighbours, it dies from overcrowding.
- if a dead cell has exactly three live neighbours, it comes back to life.

Any cell that fails to meet any of those criteria is left as is until the next iteration. With those simple rules in mind, go and play with the system for a while. Step through some iterations one at a time and observe the rules take effect.

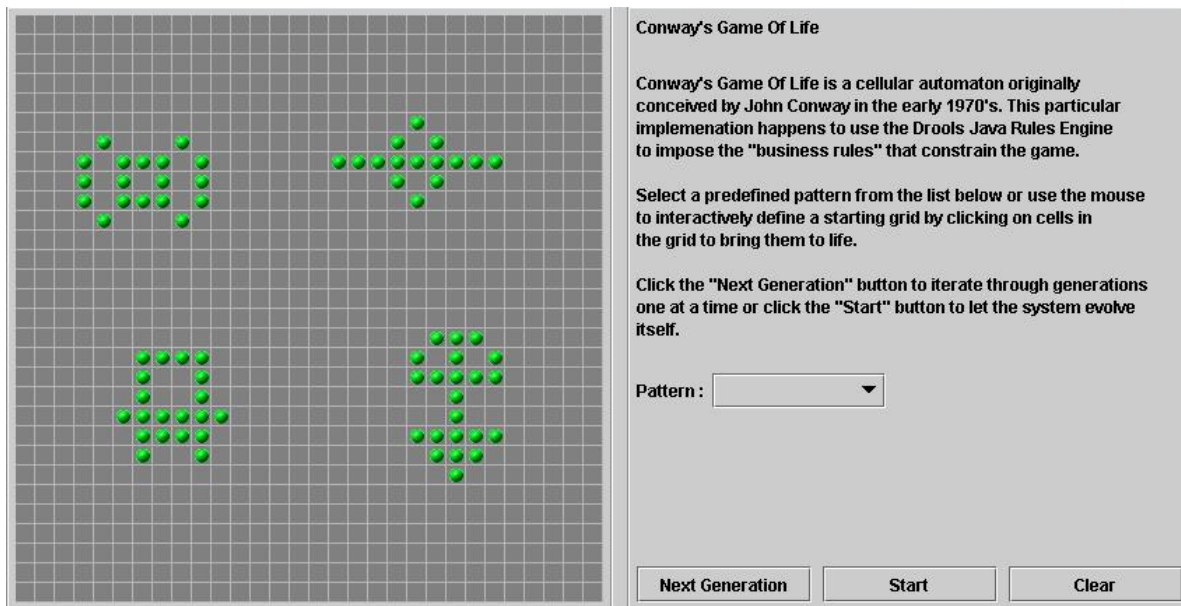


Figure 8.27. A Running Game

Now it is time to study the code. (Because this is an advanced example, it is assumed that the reader by now has an understanding of the **JBoss Rules** framework.) The example presents two ways in which to manage execution flow: via AgendaGroups (**ConwayAgendaGroupRun**) and RuleFlowGroups (**ConwayRuleFlowGroupRun**.) It is very instructive to compare them both to see the differences. This chapter will discuss the rule-flow version, as it is what most readers will use.

All of the cells are inserted into the session. The rules in the group called `register_neighbour` are granted permission to run by the rule-flow process. The rule group registers the north-eastern, northern, north-western, and western neighbours of each cell by using the **Neighbour Relation** class. Notice this relation is bi-directional, which is why there is no need to create rules for southwards-facing cells. Also note that the constraints ensure that cells are placed one column back from the end and one row back from the top. By the time all of the activations have run, all of the cells will be related to all of their neighbours.

Example 8.60. Register all Cell Neighbour Relations

```

rule "register north east"
  ruleflow-group "register neighbour"
when
  CellGrid( $numberOfColumns : numberOfColumns )
  $cell: Cell( $row : row > 0, $col : col &lt;
    ( $numberOfColumns - 1 ) )
  $northEast : Cell( row == ( $row - 1 ), col == ( $col + 1 ) )
then
  insert( new Neighbor( $cell, $northEast ) );
  insert( new Neighbor( $northEast, $cell ) );
end

rule "register north"
  ruleflow-group "register neighbour"
when
  $cell: Cell( $row : row > 0, $col : col )
  $north : Cell( row == ( $row - 1 ), col == $col )
then
  insert( new Neighbor( $cell, $north ) );
  insert( new Neighbor( $north, $cell ) );
end

```

```
rule "register north west"
  ruleflow-group "register neighbour"
  when
    $cell: Cell( $row : row > 0, $col : col > 0 )
    $northWest : Cell( row == ($row - 1), col == ( $col - 1 ) )
  then
    insert( new Neighbor( $cell, $northWest ) );
    insert( new Neighbor( $northWest, $cell ) );
  end

rule "register west"
  ruleflow-group "register neighbour"
  when
    $cell: Cell( $row : row >= 0, $col : col > 0 )
    $west : Cell( row == $row, col == ( $col - 1 ) )
  then
    insert( new Neighbor( $cell, $west ) );
    insert( new Neighbor( $west, $cell ) );
  end
```

Once all of the cells are inserted, some Java code runs, applying the pattern to the grid and setting certain cells to "alive." Clicking **Start** or **Next Generation** makes the Generation rule-flow run. This rule-flow is responsible for the managing all changes to cells in each iterative cycle.

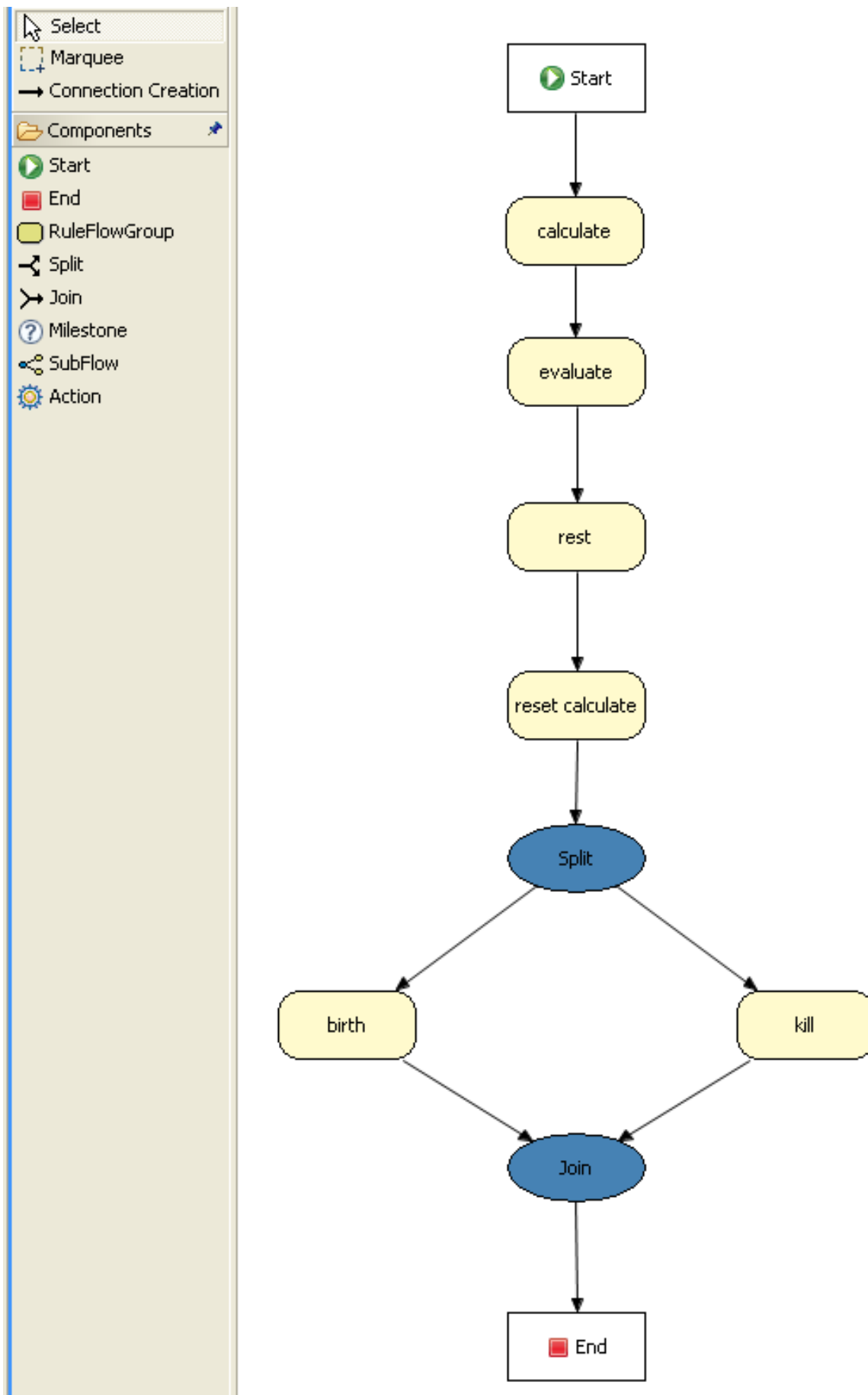


Figure 8.28. Generation Rule-Flow

The rule-flow process first runs through the evaluate group of rules. As a result, any active rule in that group can fire. (The rules in this group apply the main *Game* rules discussed at the beginning of this section. They determines which cells must die and which ones must be brought back to life.

The phase attribute dictates the way in which each cell responds to specific groups of rules. This attribute is normally tied to one of the RuleFlowGroups defined in the rule-flow process.

This attribute does not actually change the state of any cells at this point; this is because it must fully evaluate all of the grid before it applies any edits. Rather, it temporarily sets the cell to one of these two phases: Phase.KILL or Phase.BIRTH. These are used to apply the changes later.

Example 8.61. Evaluate Cells with State Changes

```

    rule "Kill The Lonely"
      ruleflow-group "evaluate"
      no-loop
    when
      # A live cell has fewer than 2 live neighbors
      theCell: Cell(liveNeighbors < 2, cellState ==
        CellState.LIVE, phase == Phase.EVALUATE)then
        theCell.setPhase(Phase.KILL);
        update( theCell );
    end

    rule "Kill The Overcrowded"
      ruleflow-group "evaluate"
      no-loop
    when
      # A live cell has more than 3 live neighbors
      theCell: Cell(liveNeighbors > 3, cellState ==
        CellState.LIVE, phase == Phase.EVALUATE)then
        theCell.setPhase(Phase.KILL);
        update( theCell );
    end

    rule "Give Birth"
      ruleflow-group "evaluate"
      no-loop
    when
      # A dead cell has 3 live neighbors
      theCell: Cell(liveNeighbors == 3, cellState ==
        CellState.DEAD, phase == Phase.EVALUATE)then
        theCell.setPhase(Phase.BIRTH);
        update( theCell );
    end

```

Once all the cells have been evaluated, the `reset calculate` rule clears any calculation activations that resulted from previous data changes from the `calculate` group. Next, a "split" is entered. This allows all activations in both the "kill" and "birth" groups to fire. (These rules are responsible for applying the state change.)

Example 8.62. Applying the Changes

```

    rule "reset calculate"
      ruleflow-group "reset calculate"
    when

```

```

then
  WorkingMemory wm = drools.getWorkingMemory();
  wm.clearRuleFlowGroup( "calculate" );
end

rule "kill"
  ruleflow-group "kill"
  no-loop
when
  theCell: Cell( phase == Phase.KILL )
then
  modify( theCell ){
    setCellState( CellState.DEAD ),
    setPhase( Phase.DONE );
  }
end

rule "birth"
  ruleflow-group "birth"
  no-loop
when
  theCell: Cell( phase == Phase.BIRTH )
then
  modify( theCell ){
    setCellState( CellState.LIVE ),
    setPhase( Phase.DONE );
  }
}

```

At this stage, a number of cells have been modified due to their states having been changed to either "live" or "dead." Next, the `neighbour` relation is used to drive the iteration over all surrounding cells, increasing or decreasing the living neighbour count. Any cell for which the count has changed will be set to an `evaluate` phase. This means they will be evaluated during the next stage of the rule-flow process.

Notice that there is no need to run the iteration process manually. Simply by applying the relations in the rules, the rule engine can be made to do all of the hard work and even it only needs a minimal amount of code. Once the "live" count for all cells has been determined and set, the rule-flow process comes to an end. The user can either tell it to evaluate another generation or, if **start** was clicked, the engine will run the rule-flow process again.

Example 8.63. Evaluating Cells for Which There Have Been State Changes

```

rule "Calculate Live"
  ruleflow-group "calculate"
  lock-on-active
when
  theCell: Cell( cellState == CellState.LIVE )
  Neighbor( cell == theCell, $neighbor : neighbor )
then
  modify( $neighbor ){
    setLiveNeighbors( $neighbor.getLiveNeighbors() + 1 ),
    setPhase( Phase.EVALUATE );
  }
end

rule "Calculate Dead"
  ruleflow-group "calculate"
  lock-on-active
when

```

```
theCell: Cell( cellState == CellState.DEAD )
Neighbor( cell == theCell, $neighbor : neighbor )
then
  modify( $neighbor ){
    setLiveNeighbors( $neighbor.getLiveNeighbors() - 1 ),
    setPhase( Phase.EVALUATE );
  }
end
```

Appendix A. © 2011

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.

Appendix B. Revision History

| | | |
|--|------------------------|---|
| Revision 5.2.0-0 | Wed Dec 15 2010 | L Carlon lcarlon@redhat.com |
| Updated for 5.2.0 | | |
| Revision 5.1.0-0 | Wed Dec 15 2010 | David Le Sage dlesage@redhat.com |
| Updated for 5.1.0 | | |
| Revision 5.0.2-0 | Tue Jun 29 2010 | David Le Sage dlesage@redhat.com |
| BRMS 341 - Removed archaic information. Sections 3.3.8 and 3.3.11. | | |
| Revision 5.0.2-0 | Wed May 5 2010 | Darrin Mison dmison@redhat.com |
| Updated for 5.0.2. | | |
| Revision 5.0.1-0 | Tue Oct 6 2009 | David Le Sage dlesage@redhat.com |
| 5.0.1 updates. First phase of grammar clean-up of this document. | | |
| Revision 5.0.0-0 | Mon May 18 2009 | Darrin Mison dmison@redhat.com |
| Published | | |

