JBoss Enterprise SOA Platform 4.3

JBoss Rules Reference Guide

Your guide to using JBoss Rules with the JBoss Enterprise SOA Platform 4.3 CP05

JBoss Enterprise SOA Platform 4.3 JBoss Rules Reference Guide
Your guide to using JBoss Rules with the JBoss Enterprise SOA Platform 4.3 CP05
Edition 4.3.5

The JBoss Rules Reference Guide for use with the JBoss SOA Platform

Preface

1. Document Conventions

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

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

1.1. Typographic Conventions

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

`Mono-spaced Bold`

Used to highlight system input, including shell commands, file names and paths. Also used to highlight 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. We Need Feedback!

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

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

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

JBoss Rules 4.0.7 Release Notes

1.1. What is New in Drools 4.0?

Drools 4.0 is a major update over the previous Drools 3.0.x series. A whole new set of features were developed with special focus on language expressiveness, engine performance and tools availability. The following is a list of the most interesting changes.

1.1.1. Language Expressiveness Enhancements

- New Conditional Elements: from, collect, accumulate and forall
- New Field Constraint operators: not matches, not contains, in, not in, memberOf, not memberOf
- New Implicit Self Reference field: this
- Full support for Conditional Elements nesting, for First Order Logic completeness.
- Support for multi-restrictions and constraint connectives && and ||
- Parser improvements to remove previous language limitations, like character escaping and keyword conflicts
- Support for pluggable dialects and full support for MVEL scripting language
- Complete rewrite of DSL engine, allowing for full l10n
- Fact attributes auto-verification for return value restrictions and inline-eval constraints
- Support for nested accessors, property navigation and simplified collection, arrays and maps syntax
- Improved support for XML rules

1.1.2. Core Engine Enhancements

- Native support for primitive types, avoiding constant autoboxing
- Support for transparent optional Shadow Facts
- Rete Network performance improvements for complex rules
- Support for Rule-Flows
- Support for Stateful and Stateless working memories (rule engine sessions)
- Support for Asynchronous Working Memory actions
- Rules Engine Agent for hot deployment and BRMS integration
- Dynamic salience for rules conflict resolution
- Support for Parameterized Queries
- Support for halt command
- Support for sequential execution mode

- Support for pluggable global variable resolver

1.1.3. IDE Enhancements

- Support for rule break-points on debugging
- WYSIWYG support for rule-flows
- New guided editor for rules authoring
- Upgrade to support all new engine features

1.1.4. Miscellaneous Enhancements

- Slimmed down dependencies and smaller memory footprint

1.2. Upgrade tips from Drools 3.0.x to Drools 4.0.x

Drools 4.0 is a major update over the previous Drools 3.0.x series. Unfortunately, in order to achieve the goals set for this release, some backward compatibility issues were introduced. This has been discussed at length in the mailing list and blogs.

This section of the manual documents a simple how-to on upgrading from Drools 3.0.x to Drools 4.0.x.

1.2.1. API changes

There are a few API changes that are visible to regular users and need to be fixed.

1.2.1.1. Working Memory creation

Drools 3.0.x had only one working memory type that functioned as a stateful working memory. Drools 4.0.x introduces separate APIs for Stateful and Stateless working memories. Stateful working memories are called Rule Sessions in Drools 4.0.x.

In Drools 3.0.x, the code to create a working memory was:

> Example 1.1. Drools 3.0.x: Working Memory Creation
>
> ```
> WorkingMemory wm = rulebase.newWorkingMemory();
> ```

In Drools 4.0.x it must be changed to:

> Example 1.2. Drools 4.0.x: Stateful Rule Session Creation
>
> ```
> StatefulSession wm = rulebase.newStatefulSession();
> ```

The StatefulSession object has the same behavior as the Drools 3.0.x WorkingMemory (it even extends the WorkingMemory interface), so there should be no other problems with this fix.

1.2.1.2. Working Memory Actions

Drools 4.0.x now supports pluggable dialects and has built-in support for Java and MVEL scripting language. In order to avoid keyword conflicts, the working memory actions were renamed as showed below:

Table 1.1. Working Memory Actions equivalent API methods

Drools 3.0.x	Drools 4.0.x
WorkingMemory.assertObject()	WorkingMemory.insert()
WorkingMemory.assertLogicalObject()	WorkingMemory.insertLogical()
WorkingMemory.modifyObject()	WorkingMemory.update()

1.2.2. Rule Language Changes

The DRL Rule Language also has some backward incompatible changes as detailed below.

1.2.2.1. Working Memory Actions

The Working Memory actions in rule consequences were also changed in a similar way to the change made in the API. The following table summarizes the changes:

Table 1.2. Working Memory Actions equivalent DRL commands

Drools 3.0.x	Drools 4.0.x
assert()	insert()
assertLogical()	insertLogical()
modify()	update()

1.2.2.2. Primitive support and unboxing

Drools 3.0.x did not have native support for primitive types and consequently, it auto-boxed all primitives in it's respective wrapper classes. Any use of a boxed variable binding required a manual unbox.

Drools 4.0.x has full support for primitive types and does not wrap values anymore. So, all previous unwrap method calls must be removed from the DRL.

Example 1.3. Drools 3.0.x manual unwrap

```
rule "Primitive int manual unbox"
when
    $c : Cheese( $price : price )
then
    $c.setPrice( $price.intValue() * 2 )
end
```

The above rule in 4.0.x would be:

Example 1.4. Drools 4.0.x primitive support

```
rule "Primitive support"
when
    $c : Cheese( $price : price )
then
    $c.setPrice( $price * 2 )
end
```

1.2.3. Drools Update Tool

The Drools Update tools is a simple program to help with the upgrade of DRL files from Drools 3.0.x to Drools 4.0.x.

Currently its main purpose is to upgrade the memory action calls from 3.0.x to 4.0.x, but future versions will conver additional scenarios. It is important to note that it does not perform a simple search and replace in the rules file, but parses the rules file to ensure it is not doing anything unexpected. For this reason it is a safe tool to use for upgrade large sets of rule files.

The drools update tool can be found as a maven project in the following Subversion source repository *http://anonsvn.jboss.org/repos/labs/labs/jbossrules/tags/4.0.7.19894.GA/experimental/drools-update/*

To build the update tool:

1. Checkout the Drools Update Tool Maven project from its Subversion repository:

```
$ svn co http://anonsvn.jboss.org/repos/labs/labs/jbossrules/tags/4.0.7.19894.GA/
experimental/drools-update/
```

2. execute the maven clean install action with the project's **`pom.xml`** file

```
$ mvn clean install
```

After you resolve any class path dependencies you are able to run the tool with the following command:

```
java -cp $CLASSPATH org.drools.tools.update.UpdateTool -f <filemask> [-d <basedir>] [-s
 <suffix>]
```

The program parameters are very easy to understand as following.

- -h,--help, Shows a very simple list the usage help
- -d your source base directory
- -f pattern for the files to be updated. The format is the same as used by ANT: * = single file, directory ** = any level of subdirectories EXAMPLE: src/main/resources/**/*.drl = matches all DRL files inside any subdirectory of /src/main/resources
- -s, --suffix, The suffix to be added to all updated files

1.2.4. DSL Grammars in Drools 4.0

It is important to note that the DSL template engine was rewritten from scratch to improve flexibility. One of the new features of DSL grammars is the support of regular expressions.

Now you can write your mappings using regular expressions to have additional flexibility, as explained in the DSL chapter. There are a number of characters which have special meanings in regular expressions such as **[** and *****. If you use these characters as part of a mapping without the intent of using the regular expression effect you will need to prefix each of those characters with a \. This is referred to as 'escaping' the characters. Note that character \ itself must be escaped unless you are using it to 'escape' another character.

For example if you previously had a mapping like:

Example 1.5. Drools 3.0.x mapping

```
[when][]- the {attr} is in [ {values} ]={attr} in ({values})
```

Now, you need to 'escape' **[** and **]** characters. So, the same mapping in Drools 4.0 would be:

Example 1.6. Drools 4.0.x mapping with escaped characters

```
[when][]- the {attr} is in \[ {values} \]={attr} in ({values})
```

1.2.5. Rule flow Update for 4.0.2

The Rule flow feature was updated for 4.0.2, and now all your ruleflows must declare a package name.

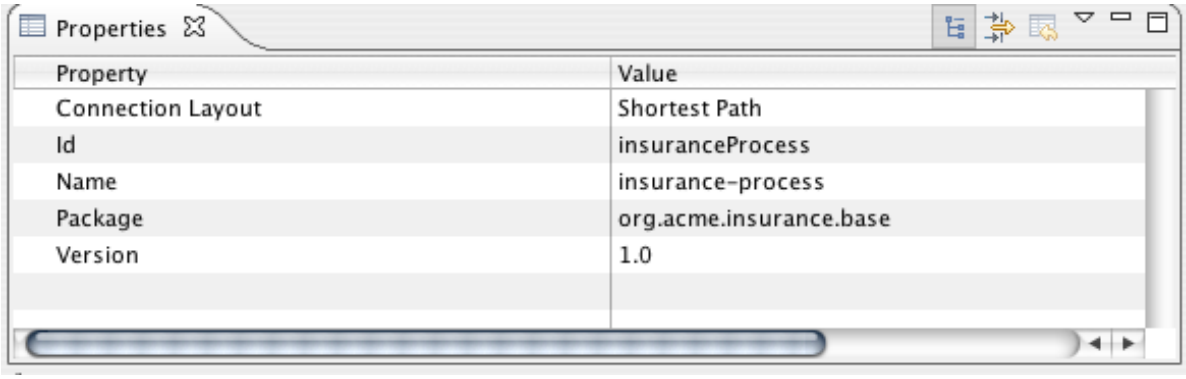


Figure 1.1. Rule Flow properties

The Rule Engine

2.1. What is a Rule Engine?

2.1.1. Introduction and Background

JBoss Rules is an advanced artificial intelligence system that utilises Turing-complete Rete algorithms to create and interpret *Production Rules*. Users can use this system to write and modify rules and procedures that reflect current business processes.

The software can then be used to manage, deploy and analyse these rules.

The "brain" of a Production Rules System is an Inference Engine that is able to scale to a large number of rules and facts. The Inference Engine matches facts and data, against Production Rules, also called Productions or just Rules, to infer conclusions which result in actions. A Production Rule is a two-part structure using First Order Logic for knowledge representation.

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

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

- Linear
- Rete
- Treat
- Leaps

The Drools Rete implementation is called **`ReteOO`**, an enhanced and optimized implementation of the Rete algorithm for Object Oriented systems.

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

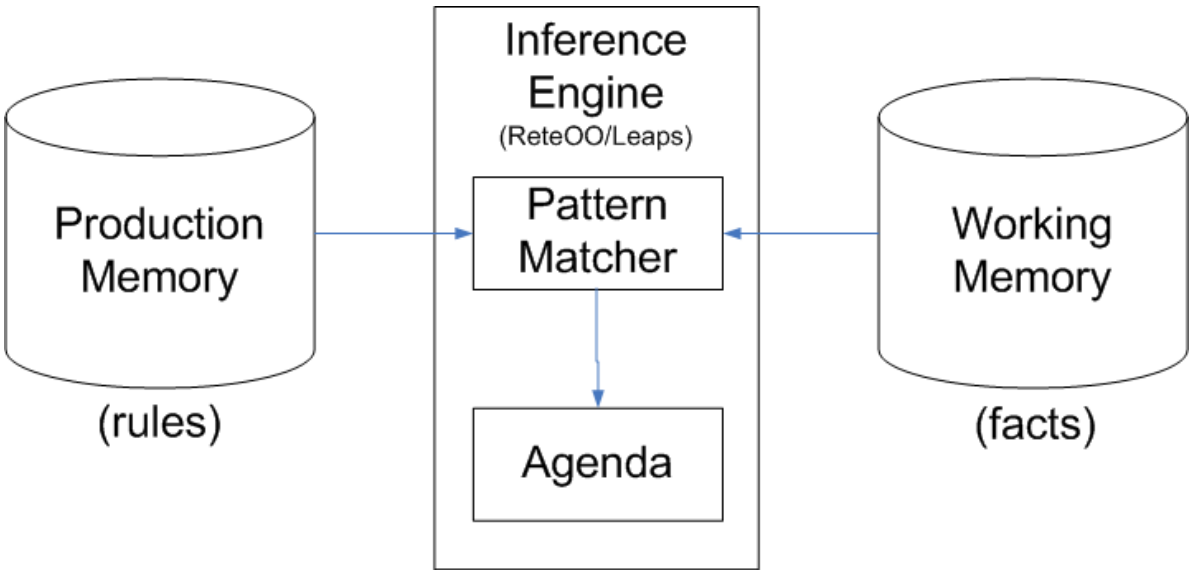


Figure 2.1. A Basic Rete network

A Production Rule System's Inference Engine is stateful and able to enforce truthfulness. This is called Truth Maintenance. A logical relationship can be declared by actions. This means the action's state depends on the inference remaining true. When it is no longer true the logical dependent action is undone.

There are two methods of execution for a Production Rule Systems, *Forward Chaining* and *Backward Chaining*. Systems that implement both methods are called *Hybrid Production Rule Systems*. s Understanding these two modes of operation are key to understanding why a Production Rule System is different and how to optimise them.

Forward Chaining

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

Drools is a Forward Chaining engine.

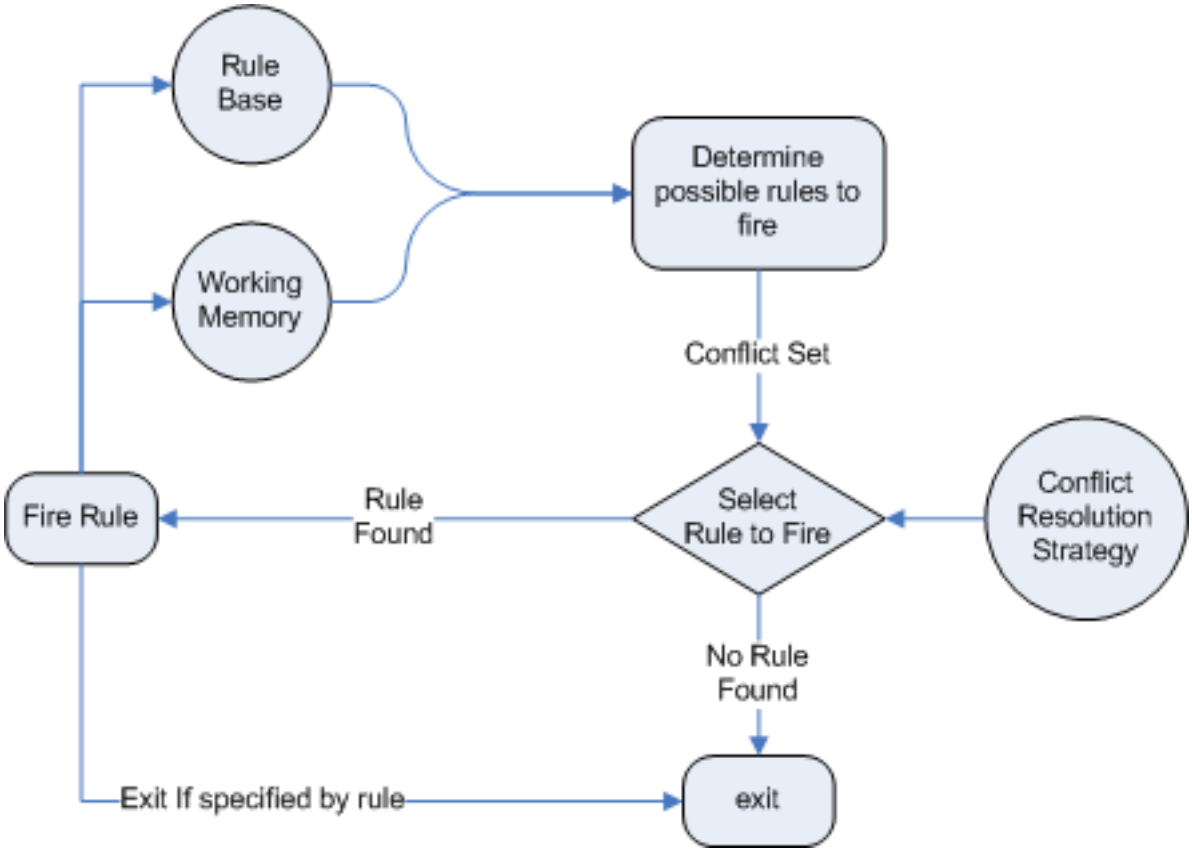


Figure 2.2. Forward Chaining

2.2. Why use a Rule Engine?

The most frequently asked questions regarding Rules Engines are:

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

We will attempt to address these questions below.

2.2.1. Advantages of a Rule Engine

- Declarative Programming

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

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

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

- Logic and Data Separation

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

- Speed and Scalability

  The Rete algorithm and its descendants such as Drools' ReteOO provide very efficient ways of matching rule patterns to your domain object data. These are especially efficient when you have datasets that do not change entirely as the rule engine can remember past matches. These algorithms are battle proven.

- Centralization of Knowledge

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

- Tool Integration

  Tools such as Eclipse (and in future, Web based UIs) provide ways to edit and manage rules and get immediate feedback, validation and content assistance. Auditing and debugging tools are also available.

- Explanation Facility

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

- Understandable Rules

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

2.2.2. When should you use a Rule Engine?

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

- The problem is just too complex for traditional code.

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

- The problem is beyond any obvious algorithm based solution.

  It is a complex problem to solve. There are no obvious traditional solutions or the problem isn't fully understood.

- The logic changes often

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

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

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

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

Typically in a modern object-orientated application you would use a rule engine to contain key parts of your business logic, although what that means of course depends on your application. This is an inversion of the object-orientated concept of encapsulating all the logic inside your objects. This is not to say that you throw out object-orientated practices, on the contrary in any real world application, business logic is just one part of the application. If you ever notice lots of "if", "else", "switch", an over abundance of strategy patterns or other messy logic in your code that you keep having to change then think about using rules. If you are faced with tough problems of which there are no algorithms or patterns for, consider using rules.

Rules could be used embedded in your application or perhaps as a service. Rules usually work best as "stateful" component - hence they are often an integral part of an application. However, there have been successful cases of creating reusable rule services which are stateless.

In your organization it is important to think about the process you will use for updating rules in systems that are in production. You have many options, but different organizations have different requirements which are out of the control of the application vendors and project teams.

2.2.3. When not to use a Rule Engine

In the excitement of working with rules engines, that people forget that a rules engine is only one piece of a complex application or solution.

Because rule engines are dynamic, in the sense that the rules can be stored and managed and updated as data, they are often looked at as a solution to the problem of deploying software (most IT departments seem to exist for the purpose of preventing software being rolled out). If this is the reason you wish to use a rule engine, be aware that rule engines work best when you are able to write declarative rules. As an alternative, you can consider data-driven designs (lookup tables), or script/process engines where the scripts are managed in a database and are able to be updated on the fly.

2.2.4. Scripting or Process Engines

Hopefully the preceding sections have explained when you may want to use a rule engine.

Alternatives are script-based engines that provide a mechanism for "changes on the fly" (there are many solutions here).

Alternatively Process Engines (also capable of workflow) such as jBPM allow you to graphically (or programmatically) describe steps in a process - those steps can also involve decision point which are in themselves a simple rule. Process engines and rules often can work nicely together, so it is not an either-or proposition.

One key point to note with rule engines, is that some rule-engines are really scripting engines. The downside of scripting engines is that you are tightly coupling your application to the scripts (if

they are rules, you are effectively calling rules directly) and this may cause more difficulty in future maintenance, as they tend to grow in complexity over time. The upside of scripting engines is they can be easier to implement at first and you can get quick results. They are also conceptually simpler for imperative programmers.

Many people have also implemented data-driven systems successfully in the past, often using control tables that store meta-data that determine your applications behavior. These can work well when the control remains very limited. However their complexity can can quickly grow out of control as the applications behaviour is extended or they cause the application to stagnate as they are too inflexible.

2.2.5. Strong and Loose Coupling

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

2.3. Knowledge Representation

2.3.1. First Order Logic

Rules are written using First Order Logic, or predicate logic, which extends Propositional Logic. *Emil Leon Post*[1] was the first to develop an inference based system using symbols to express logic - as a consequence of this he was able to prove that any logical system (including mathematics) could be expressed with such a system.

A proposition is a statement is either true or false. If the truth can be determined from statement alone it is said to be a "closed statement". In programming terms this is an expression that does not reference any variables:

```
10 == 2 * 5
```

Expressions that evaluate against one or more variables, the facts, are "open statements", in that we cannot determine whether the statement is true until we have a variable instance to evaluate against:

```
Person.sex == "male"
```

With SQL if we look at the conclusion's action as simply returning the matched fact to the user:

```
select * from People where People.sex == "male"
```

For any rows, which represent our facts, that are returned we have inferred that those facts are male people. The following diagram shows how the above SQL statement and People table can be represented in terms of an Inference Engine.

[1] http://en.wikipedia.org/wiki/Emil_Leon_Post

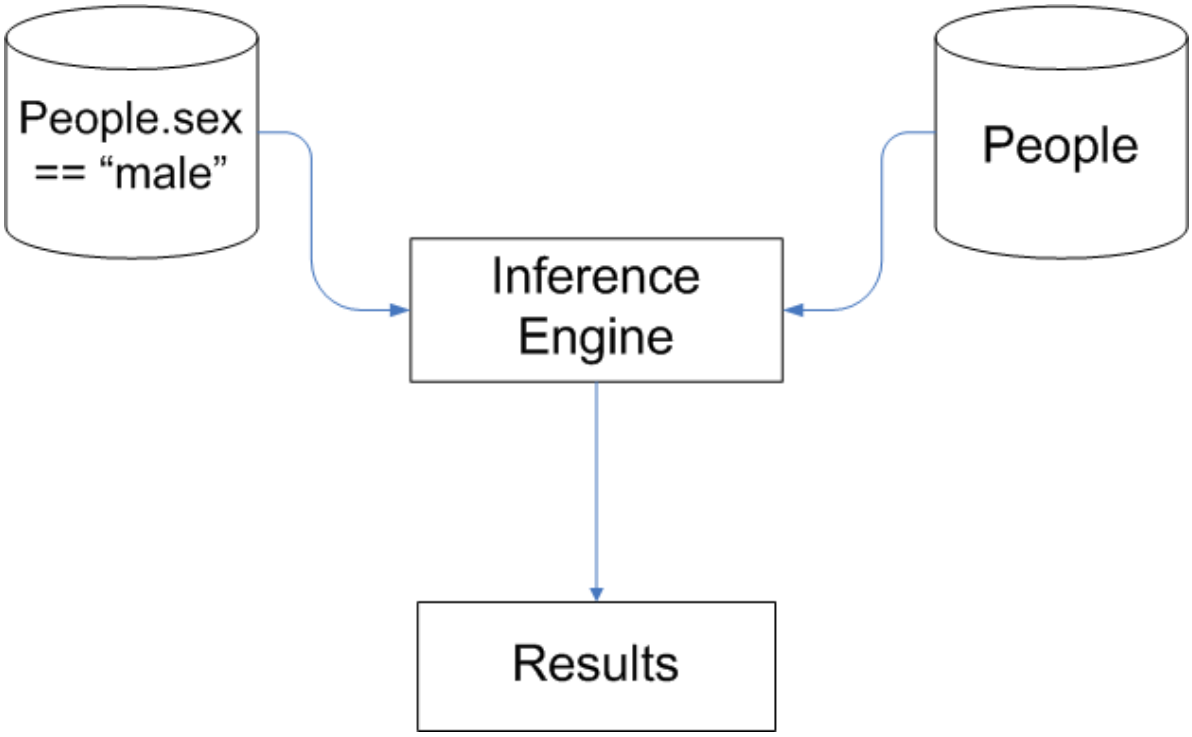


Figure 2.3. SQL as a simplistic Inference Engine

In Java we can say that a simple proposition is of the form "variable operator value", where we often refer to 'value' as being a literal value and a proposition can be thought as a field constraint. Propositions can be combined with conjunctive and disjunctive connectives, which is the logic theorists way of saying 'AND' and 'OR' (**&&** and **| |**). The following shows two open propositional statements connected together with a single disjunctive connective.

```
person.getEyeColor().equals("blue") || person.getEyeColor().equals("green")
```

This can be expressed using a disjunctive Conditional Element connective - which actually results in the generation of two rules, to represent the two possible logic outcomes.

```
Person( eyeColour == "blue" ) || Person( eyeColor == "green" )
```

Disjunctive field constraints connectives could also be used and would not result in multiple rule generation.

```
Person( eyeColour == "blue"|| == "green" )
```

Propositional Logic is not Turing complete, limiting the problems you can define, because it cannot express criteria for the structure of data. First Order Logic (FOL), or Predicate Logic, extends Propositional Logic with two new quantifier concepts to allow expressions defining structure - specifically universal and existential quantifiers. Universal quantifiers allow you to check that something is true for everything; normally supported by the 'forall' conditional element. Existential quantifiers check for the existence of something, in that it occurs at least once - this is supported with 'not' and 'exists' conditional elements.

Imagine we have two classes - Student and Module. Module represents each of the courses the Student attended for that semester, referenced by the List collection. At the end of the semester each Module has a score. If the Student has a Module score below 40 then they will fail that semester -

the existential quantifier can be used used with the "less than 40" open proposition to check for the existence of a Module that is true for the specified criteria.

```
public class Student {
    private String name;
    private List modules;
}
```

```
public class Module {
    private String name;
    private String studentName;
    private int score;
}
```

Java is Turing complete in that you can write code, among other things, to iterate data structures to check for existence. The following should return a List of students who have failed the semester.

```
List failedStudents = new ArrayList();

for ( Iterator studentIter = students.iterator(); studentIter.hasNext() {
    Student student = ( Student ) studentIter.next();
    for ( Iterator it = student.getModules.iterator(); it.hasNext(); ) {
        Module module = ( Module ) it.next();
        if ( module.getScore() < 40  ) {
            failedStudents.add( student ) ;
            break;
        }
    }
}
```

Early SQL implementations were not Turing complete as they did not provide quantifiers to access the structure of data. Modern SQL engines do allow nesting of SQL, which can be combined with keywords like 'exists' and 'in'. The following show SQL and a Rule to return a set of Students who have failed the semester.

```
select * from
    Students s
where exists (
    select * from
        Modules m
    where
        m.student_name = s.name and
        m.score < 40
)
```

```
rule "Failed_Students"
when
    exists( $student : Student() && Module( student == $student, score < 40 ) )
```

2.4. Rete Algorithm

The word RETE is Latin for "net" meaning network. The RETE algorithm can be broken into two parts: rule compilation and runtime execution.

The compilation algorithm describes how the Rules in the Production Memory to generate an efficient discrimination network. In non-technical terms, a discrimination network is used to filter data. The idea is to filter data as it propagates through the network. At the top of the network the nodes would have

many matches and as we go down the network, there would be fewer matches. At the very bottom of the network are the terminal nodes. There are four basic nodes: root, 1-input, 2-input and terminal.

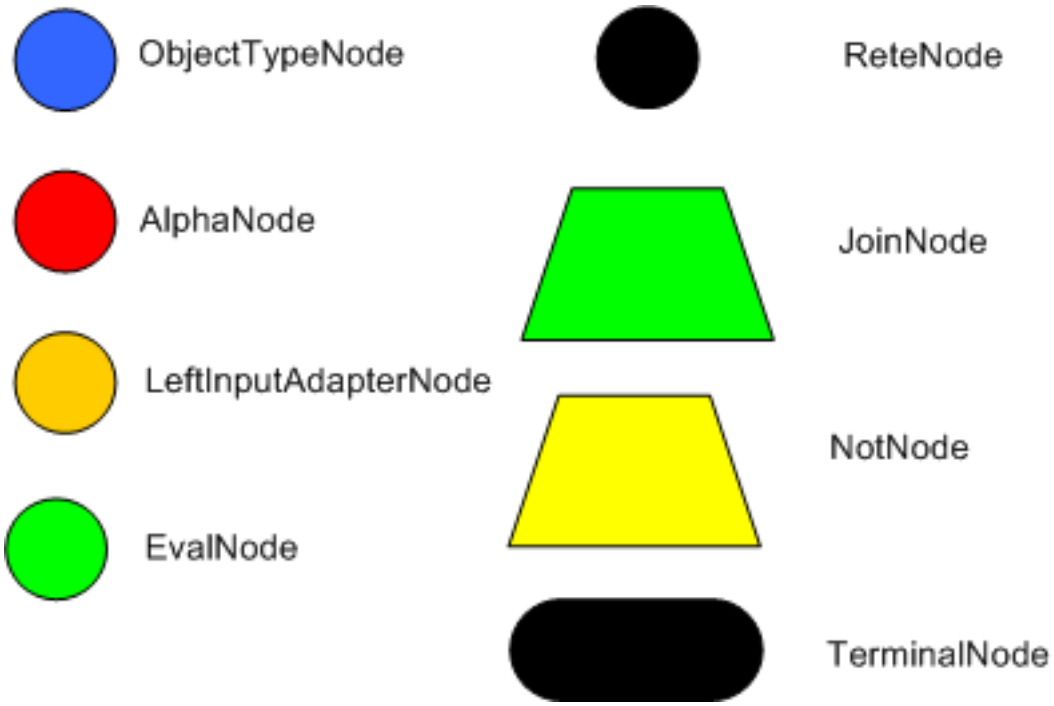


Figure 2.4. Rete Nodes

The root node is where all objects enter the network. From there, it immediately goes to the ObjectTypeNode. The purpose of the ObjectTypeNode is to make sure the engine doesn't do more work than it needs to. For example, say we have 2 objects: Account and Order. If the rule engine tried to evaluate every single node against every object, it would waste a lot of cycles. To make things efficient, the engine should only pass the object to the nodes that match the object type. The easiest way to do this is to create an ObjectTypeNode and have all 1-input and 2-input nodes descend from it. This way, if an application asserts a new account, it won't propagate to the nodes for the Order object. In Drools when an object is asserted it retrieves a list of valid ObjectTypesNodes via a lookup in a HashMap from the object's Class; if this list doesn't exist it scans all the ObjectTypde nodes finding valid matches which it caches in the list. This enables Drools to match against any Class type that matches with an **`instanceof`** check.

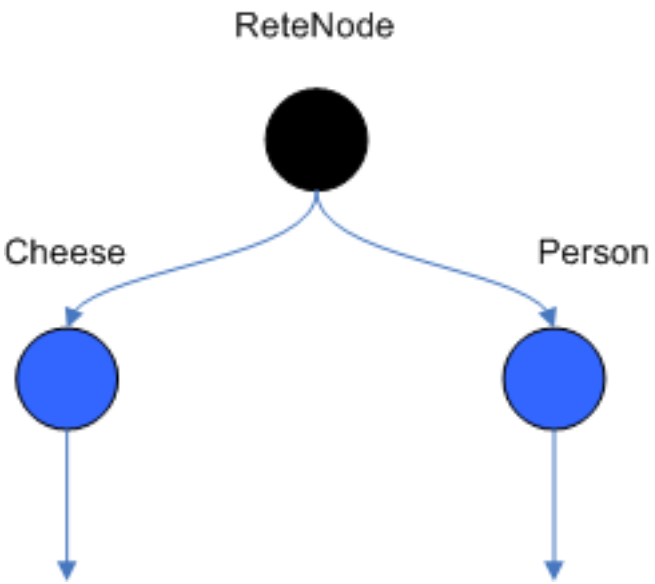


Figure 2.5. ObjectTypeNodes

ObjectTypeNodes can propagate to AlphaNodes, LeftInputAdapterNodes and BetaNodes. AlphaNodes are used to evaluate literal conditions. Although the 1982 paper only covers equality conditions, many RETE implementations support other operations. For example, Account.name == "Mr Trout" is a literal condition. When a rule has multiple literal conditions for a single object type, they are linked together. This means that if an application asserts an account object, it must first satisfy the first literal condition before it can proceed to the next AlphaNode. In Dr. Forgy's paper, he refers to these as IntraElement conditions. The following shows the AlphaNode combinations for Cheese(name == "cheddar, strength == "strong"):

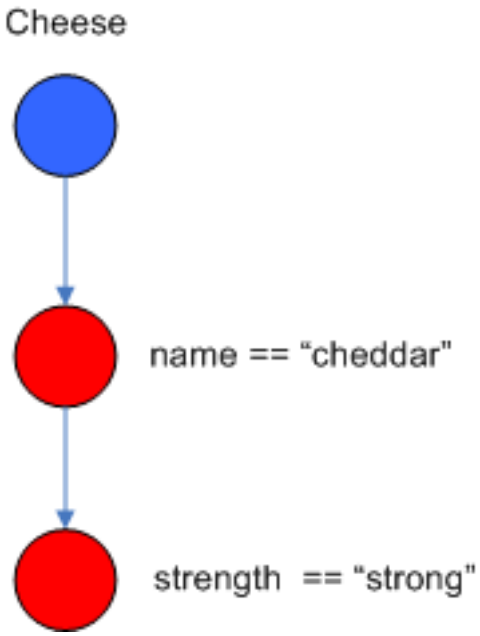


Figure 2.6. AlphaNodes

Drools extends Rete by optimizing the propagation from ObjectTypeNode to AlphaNode using hashing. Each time an AlphaNode is added to an ObjectTypdeNode it adds the literal value as a key to the HashMap with the AlphaNode as the value. When a new instance enters the ObjectTypde node, rather than propagating to each AlphaNode, it can instead retrieve the correct AlphaNode from the HashMap - avoiding unnecessary literal checks.

There are two two-input nodes; JoinNode and NotNode - both are types of BetaNodes. BetaNodes are use to compare 2 objects, and their fields, to each other. The objects may be the same or different types. By convention we refer to the two inputs as left and right. The left input for a BetaNode is generally a list of objects; in Drools this is a Tuple. The right input is a single object. Two Nots can be used to implement 'exists' checks. BetaNodes also have memory. The left input is called the Beta Memory and remembers all incoming tuples. The right input is called the Alpha Memory and remembers all incoming objects. Drools extends Rete by performing indexing on the BetaNodes. For instance, if we know that a BetaNode is performing a check on a String field, as each object enters we can do a hash lookup on that String value. This means when facts enter from the opposite side, instead of iterating over all the facts to find valid joins, we do a lookup returning potentially valid candidates. At any point a valid join is found the Tuple is joined with the Object; which is referred to as a partial match; and then propagated to the next node.

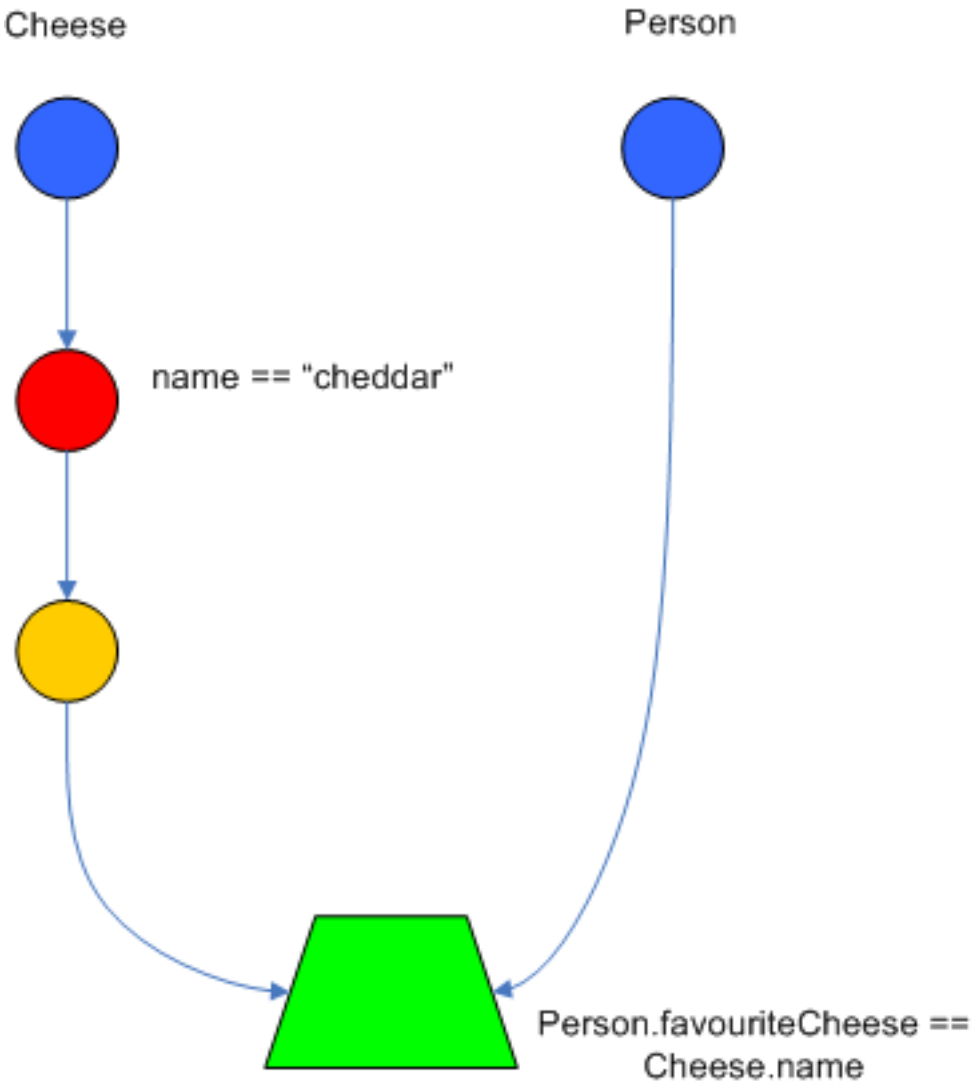


Figure 2.7. JoinNode

To enable the first Object, in the above case Cheese, to enter the network we use a LeftInputNodeAdapter - this takes an Object as an input and propagates a single Object Tuple.

Terminal nodes are used to indicate a single rule has matched all its conditions - at this point we say the rule has a full match. A rule with an 'or' conditional disjunctive connective results in subrule generation for each possible logically branch; thus one rule can have multiple terminal nodes.

Drools also performs node sharing. Many rules repeat the same patterns, node sharing allows us to collapse those patterns so that they don't have to be re-evaluated for every single instance. The following two rules share the first same pattern, but not the last:

```
rule
when
    Cheese( $chedddar : name == "cheddar" )
    $person : Person( favouriteCheese == $cheddar )
then
    System.out.println( $person.getName() + " likes cheddar" );
end
```

```
rule
when
    Cheese( $chedddar : name == "cheddar" )
    $person : Person( favouriteCheese != $cheddar )
then
    System.out.println( $person.getName() + " does not like cheddar" );
end
```

As you can see below, the compiled Rete network shows the alpha node is shared, but the beta nodes are not. Each beta node has its own TerminalNode. Had the second pattern been the same it would have also been shared.

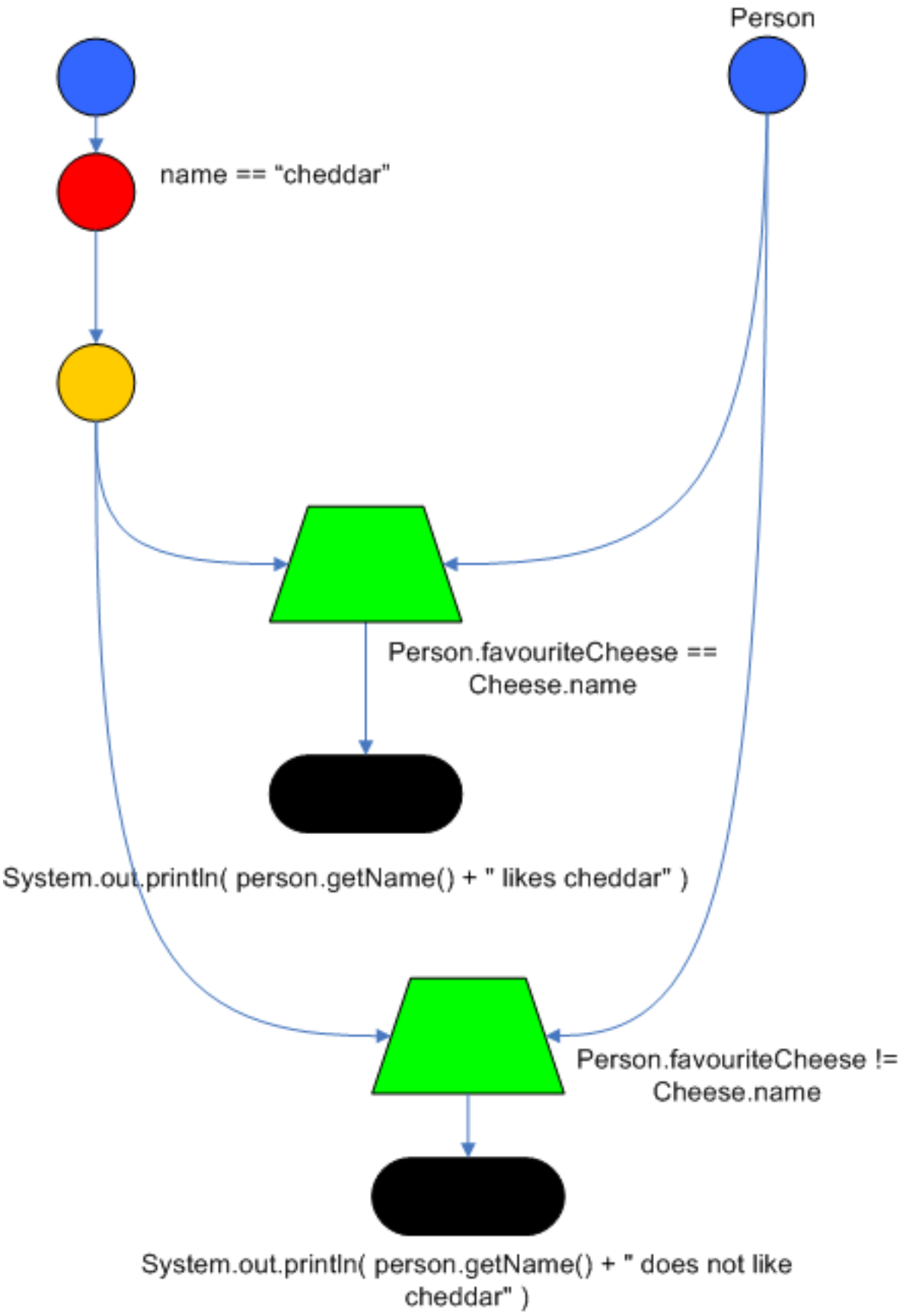


Figure 2.8. Node Sharing

2.5. The Drools Rule Engine

2.5.1. Overview

Drools is split into two main parts: Authoring and Runtime.

The authoring process involves the creation of DRL or XML files for rules which are fed into a parser, defined by an *Antlr 3* grammar. The parser checks for correctly formed grammar and produces an intermediate structure for the "descr"; where the "descr" indicates the AST that "describes" the rules. The AST is then passed to the Package Builder which produces Packages. Package Builder also undertakes any code generation and compilation that is necessary for the creation of the Package. A Package object is self contained and deployable, it is a serialized object consisting of one or more rules.

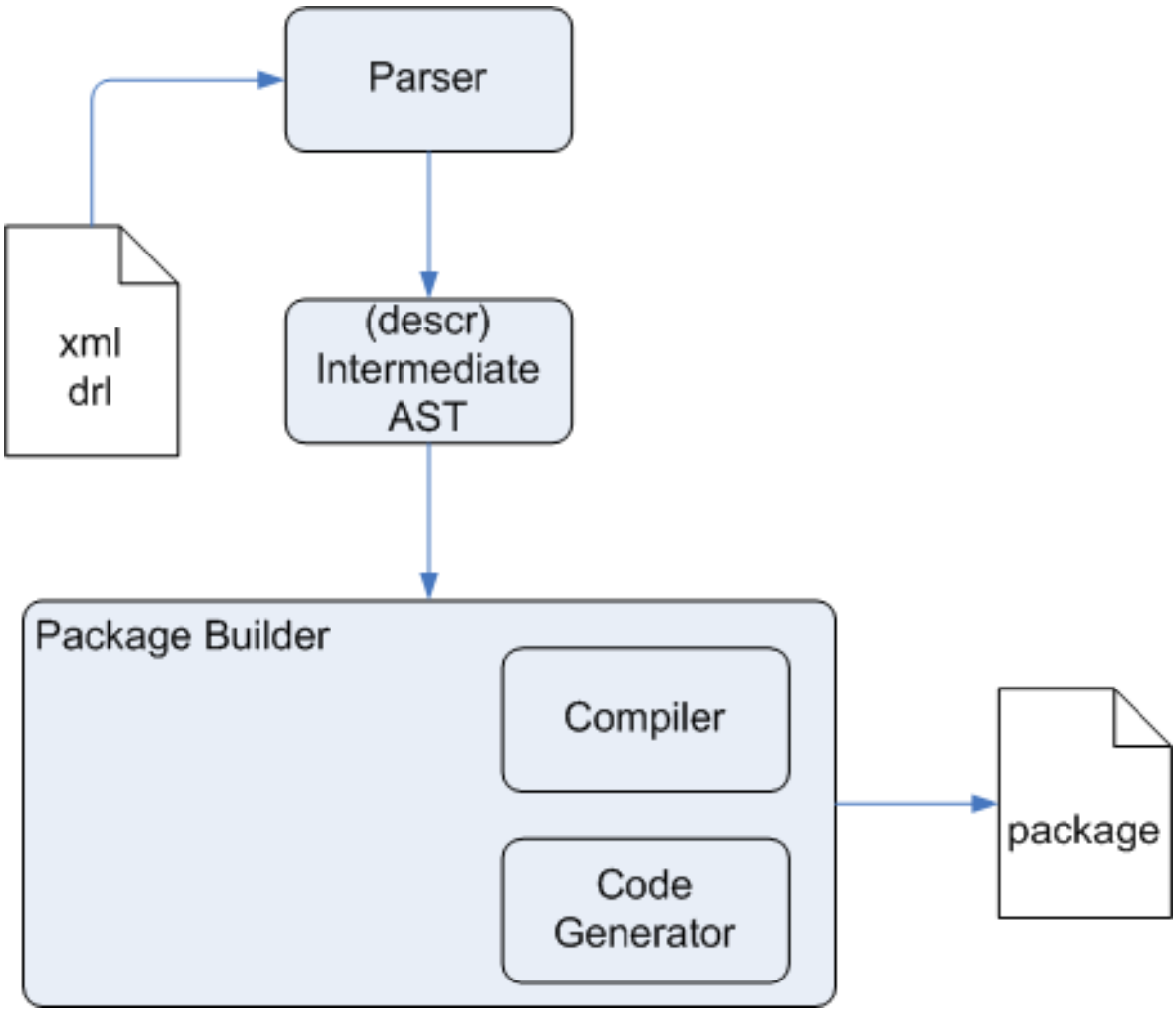


Figure 2.9. Authoring Components

A RuleBase is a runtime component which consists of one or more Packages. Packages can be added and removed from the RuleBase at any time. A RuleBase can instantiate one or more WorkingMemories at any time; a weak reference is maintained, unless configured otherwise. The Working Memory consists of a number of sub components, including Working Memory Event Support, Truth Maintenance System, Agenda and Agenda Event Support. Object insertion may result in the creation of one or more Activations. The Agenda is responsible for scheduling the execution of these Activations.

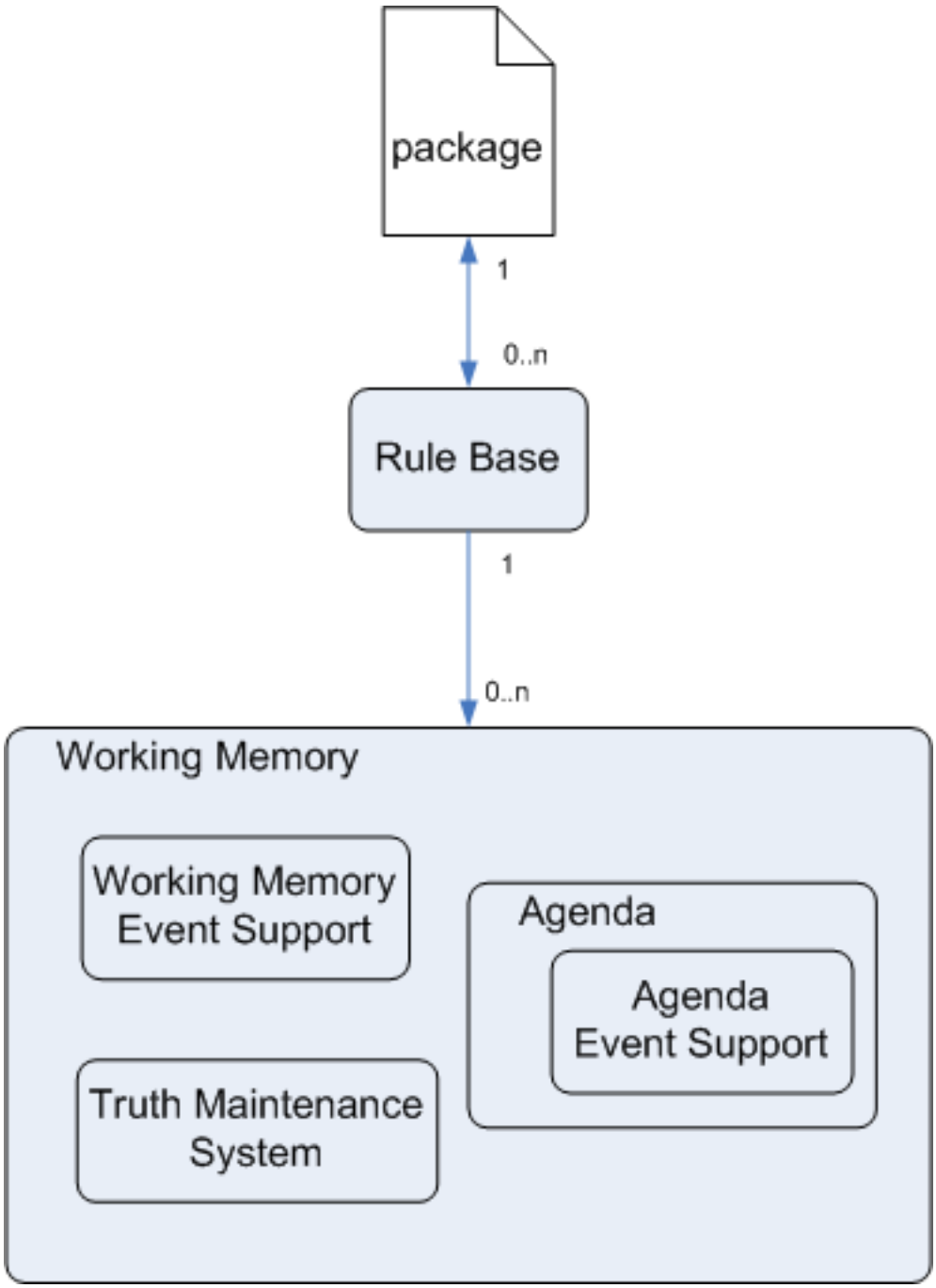


Figure 2.10. Runtime Components

2.5.2. Authoring

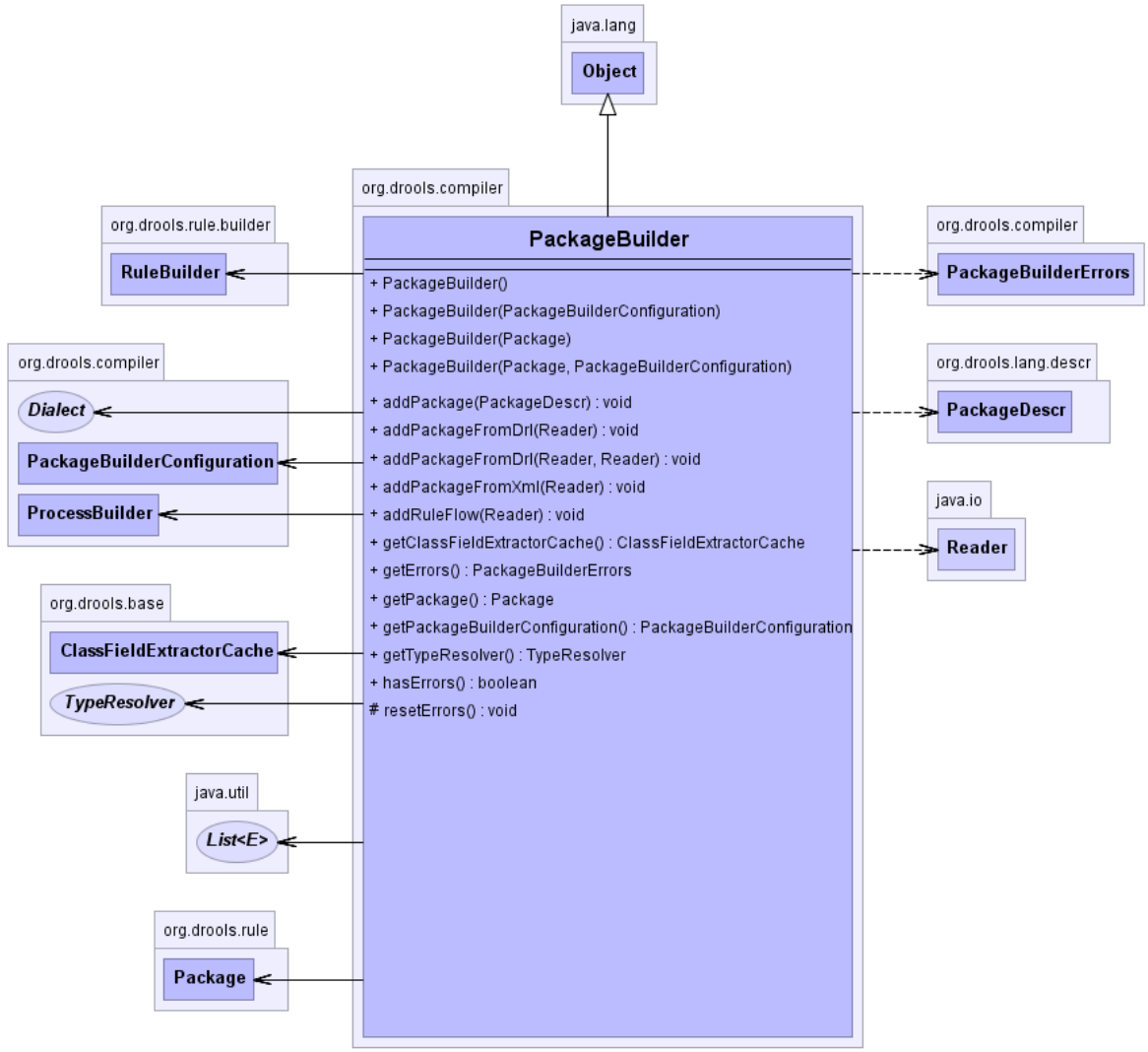


Figure 2.11. PackageBuilder

Four classes are used for authoring: DrlParser, XmlParser, ProcessBuilder and PackageBuilder. The two parser classes produce "descr" (description) AST models from a provided Reader instance. ProcessBuilder reads in an xstream serialisation representation of the Rule Flow. PackageBuilder provides convienience APIs so that you can mostly forget about those classes. The three convenience methods are "addPackageFromDrl", "addPackageFromXml" and addRuleFlow - all take an instance of Reader as an argument. The example below shows how to build a package that includes both XML, DRL and rule files and a ruleflow file, which are in the classpath. Note that all added package sources must be of the same package namespace for the current PackageBuilder instance!

Note

RuleFlow is not supported by the JBoss Enterprise SOA Platform for BPM workflow or service orchestration. Refer to *Section 5.8, "Rule Flow"* for more details.

Example 2.1. Building a Package from Multiple Sources

```
PackageBuilder builder = new PackageBuilder();
```

```
builder.addPackageFromDrl( new
 InputStreamReader( getClass().getResourceAsStream( "package1.drl" ) ) );
builder.addPackageFromXml( new
 InputStreamReader( getClass().getResourceAsStream( "package2.xml" ) ) );
builder.addRuleFlow( new InputStreamReader( getClass().getResourceAsStream( "ruleflow.rfm"
 ) ) );
Package pkg = builder.getPackage();
```

It is essential that you always check your PackageBuilder for errors before attempting to use it. While the ruleBase does throw an InvalidRulePackage when a broken Package is added, the detailed error information is stripped and only a toString() equivalent is available. If you interrogate the PackageBuilder itself much more information is available.

Example 2.2. Checking the PackageBuilder for errors

```
PackageBuilder builder = new PackageBuilder();
builder.addPackageFromDrl( new
 InputStreamReader( getClass().getResourceAsStream( "package1.drl" ) ) );
PackageBuilderErrors errors = builder.getErrors();
```

PackageBuilder is configurable using **`PackageBuilderConfiguration`** class.

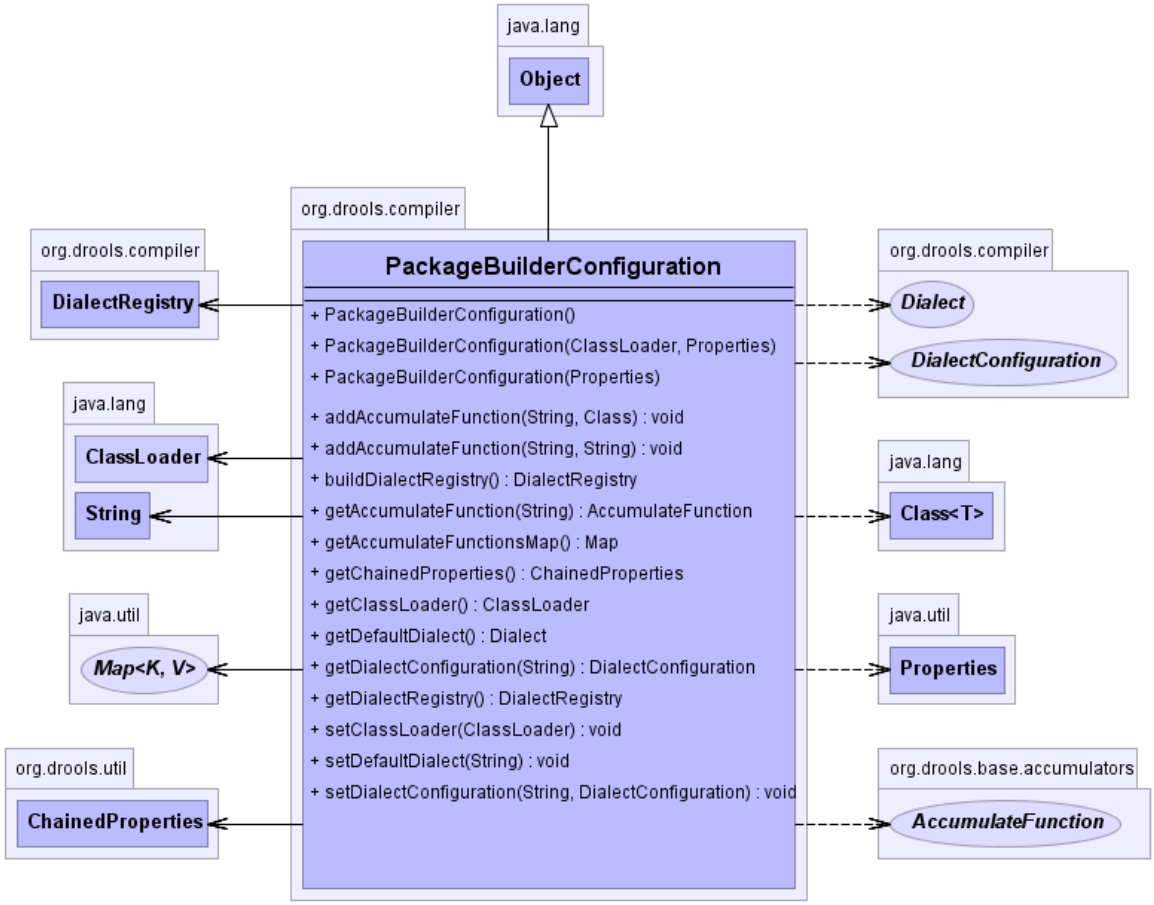


Figure 2.12. PackageBuilderConfiguration

It has default values that can be overridden programmatically via *setters* or on first use via property settings. At the heart of the settings is the **`ChainedProperties`** class which searches several locations looking for **`drools.packagebuilder.conf`** files. As it finds them it adds the properties

to the master propperties list, providing a level precedence. In order of precedence those locations are: System Properties, user defined file in System Properties, user home directory, working directory, various META-INF locations. Further to this the droosl-compiler jar has the default settings in its META-INF directory.

Currently the PackageBulderConfiguration handles the registry of Accumulate functions, registry of Dialects and the main ClassLoader.

Drools has a pluggeable Dialect system which allows other languages to compile and execute expressions and blocks. The two currently supported dialects are Java and MVEL. Each has its own DialectConfiguration Implementation. The javadocs provide details for each setter/getter and the property names used to configure them.

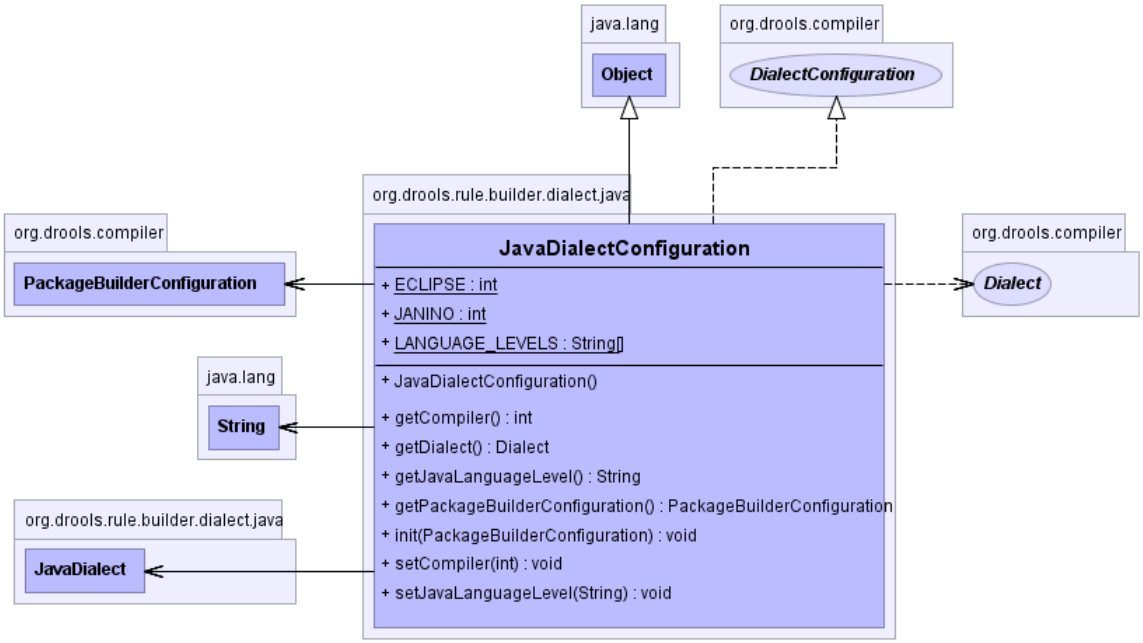


Figure 2.13. JavaDialectConfiguration

The **`JavaDialectConfiguration`** allows the compiler and language levels to be supported. You can override by setting the drools.dialect.java.compiler property in a **`packagebuilder.conf`** file that the **`ChainedProperties`** instance will find, or you can do it at runtime as shown below.

Example 2.3. Configuring the JavaDialectConfiguration to use JANINO via a setter

```
PackageBuilderConfiguration cfg = new PackageBuilderConfiguration( );
JavaDialectConfiguration javaConf = (JavaDialectConfiguration)
 cfg.getDialectConfiguration( "java" );
javaConf.setCompiler( JavaDialectConfiguration.JANINO );
```

if you do not have Eclipse JDT Core in your classpath you must override the compiler setting before you instantiate this **`PackageBuilder`**, you can either do that with a packagebuilder properties file the **`ChainedProperties`** class will find, or you can do it programmatically as shown below. In this example properties are used to inject the value for startup.

Example 2.4. Configuring the JavaDialectConfiguration to use JANINO

```
Properties properties = new Properties();
```

```
properties.setProperty( "drools.dialect.java.compiler","JANINO" );
PackageBuilderConfiguration cfg = new PackageBuilderConfiguration( properties );
JavaDialectConfiguration javaConf = (JavaDialectConfiguration)
 cfg.getDialectConfiguration( "java" );
// demonstrate that the compiler is correctly configured
assertEquals( JavaDialectConfiguration.JANINO,
              javaConf.getCompiler() );
```

Currently it allows alternative compilers (Janino, Eclipse JDT) to be specified, different JDK source levels ("1.4" and "1.5") and a parent class loader. The default compiler is Eclipse JDT Core at source level "1.4" with the parent class loader set to "Thread.currentThread().getContextClassLoader()".

The following show how to specify the JANINO compiler programmatically:

Example 2.5. Configuring the PackageBuilder to use JANINO via a property

```
PackageBuilderConfiguration conf = new PackageBuilderConfiguration();
conf.setCompiler( PackageBuilderConfiguration.JANINO );
PackageBuilder builder = new PackageBuilder( conf );
```

The MVELDialectConfiguration is much simpler and only allows strict mode to be turned on and off, by default strict is true; this means all method calls must be type safe either by inference or by explicit typing.

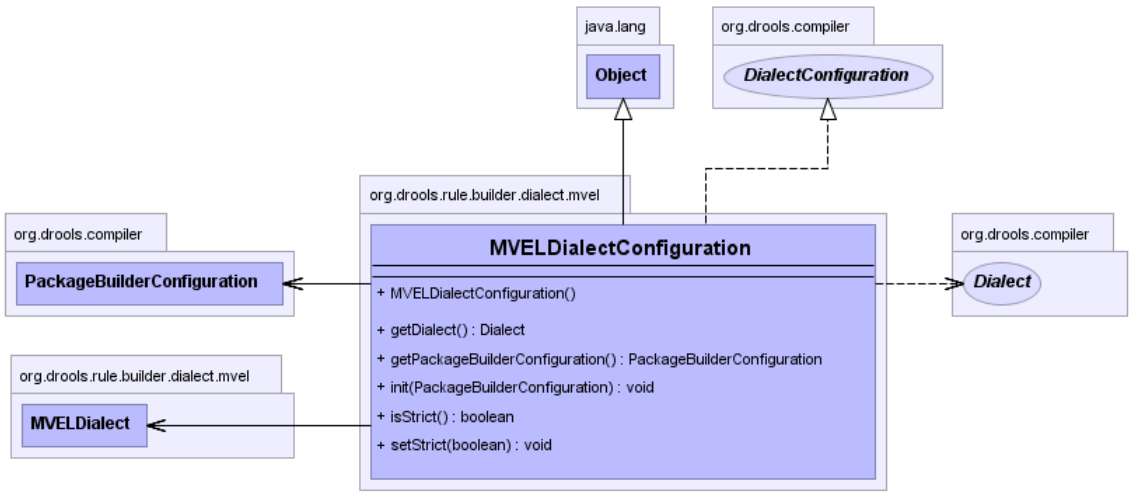


Figure 2.14. MvelDialectConfiguration

2.5.3. RuleBase

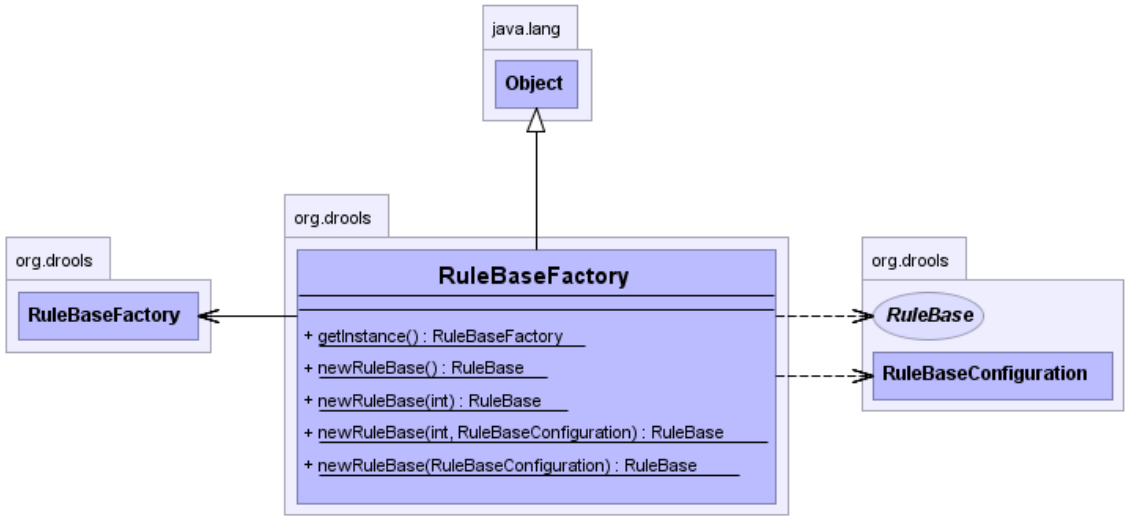


Figure 2.15. RuleBaseFactory

A RuleBase is instantiated using the RuleBaseFactory, by default this returns a ReteOO RuleBase. Packages are added, in turn, using the addPackage method. You may specify packages of any namespace and multiple packages of the same namespace may be added.

Example 2.6. Adding a Package to a new RuleBase

```
RuleBase ruleBase = RuleBaseFactory.newRuleBase();
ruleBase.addPackage( pkg  );
```

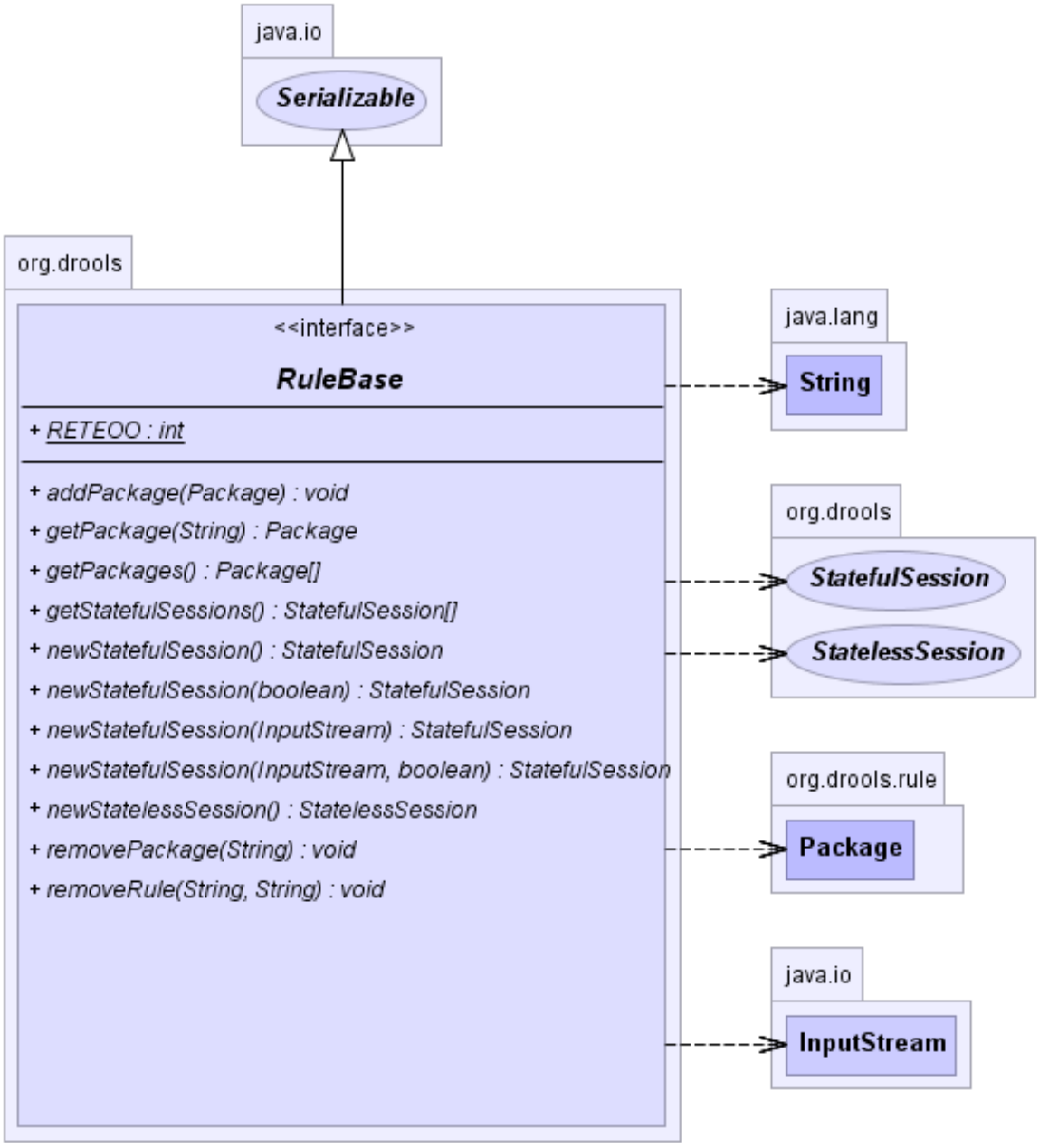


Figure 2.16. RuleBase

A RuleBase contains one or more more packages of rules, ready to be used, i.e., they have been validated/compiled etc. A Rule Base is serializable so it can be deployed to JNDI or other such services. Typically, a rulebase would be generated and cached on first use; to save on the continually re-generation of the Rule Base; which is expensive.

A Rule Base instance is thread safe, in the sense that you can have the one instance shared across threads in your application, which may be a web application, for instance. The most common operation on a rulebase is to create a new rule session; either stateful or stateless.

The Rule Base also holds references to any stateful session that it has spawned, so that if rules are changing (or being added/removed etc. for long running sessions), they can be updated with the latest rules (without necessarily having to restart the session). You can specify not to maintain a reference, but only do so if you know the Rule Base will not be updated. References are not stored for stateless sessions.

```
ruleBase.newStatefulSession();  // maintains a reference.
ruleBase.newStatefulSession( false ); // do not maintain a reference
```

Packages can be added and removed at any time and all changes will be propagated to the existing stateful sessions. Do not forget to call fireAllRules() for resulting Activations to fire.

```
ruleBase.addPackage( pkg );  // Add a package instance
ruleBase.removePackage( "org.com.sample" );  // remove a package, and all its parts, by it's
 namespace
ruleBase.removeRule( "org.com.sample", "my rule" ); // remove a specific rule from a
 namespace
```

While there is a method to remove an individual rule, there is no method to add an individual rule. You just add a new package with a single rule in it.

RuleBaseConfigurator can be used to specify additional behavior of the RuleBase. RuleBaseConfiguration is set to immutable after it has been added to a Rule Base. Nearly all the engine optimizations can be turned on and off from here, and also the execution behavior can be set. Users will generally be concerned with insertion behavior (identity or equality) and cross product behavior(remove or keep identity equals cross products).

```
RuleBaseConfiguration conf = new RuleBaseConfiguration();
conf.setAssertBehaviour( AssertBehaviour.IDENTITY );
conf.setRemoveIdentities( true );
RuleBase ruleBase = RuleBaseFactory.newRuleBase( conf );
```

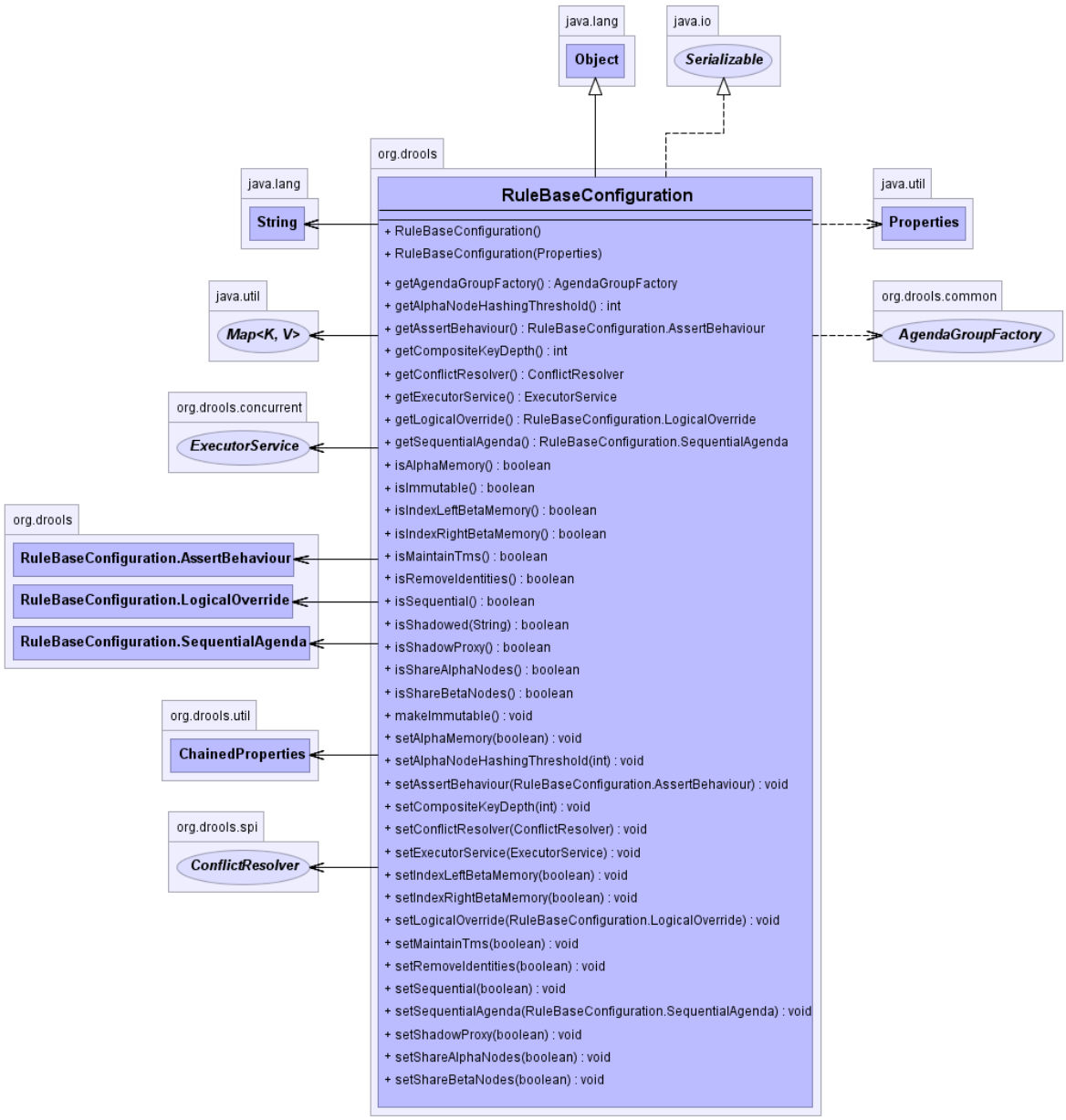


Figure 2.17. RuleBaseConfiguration

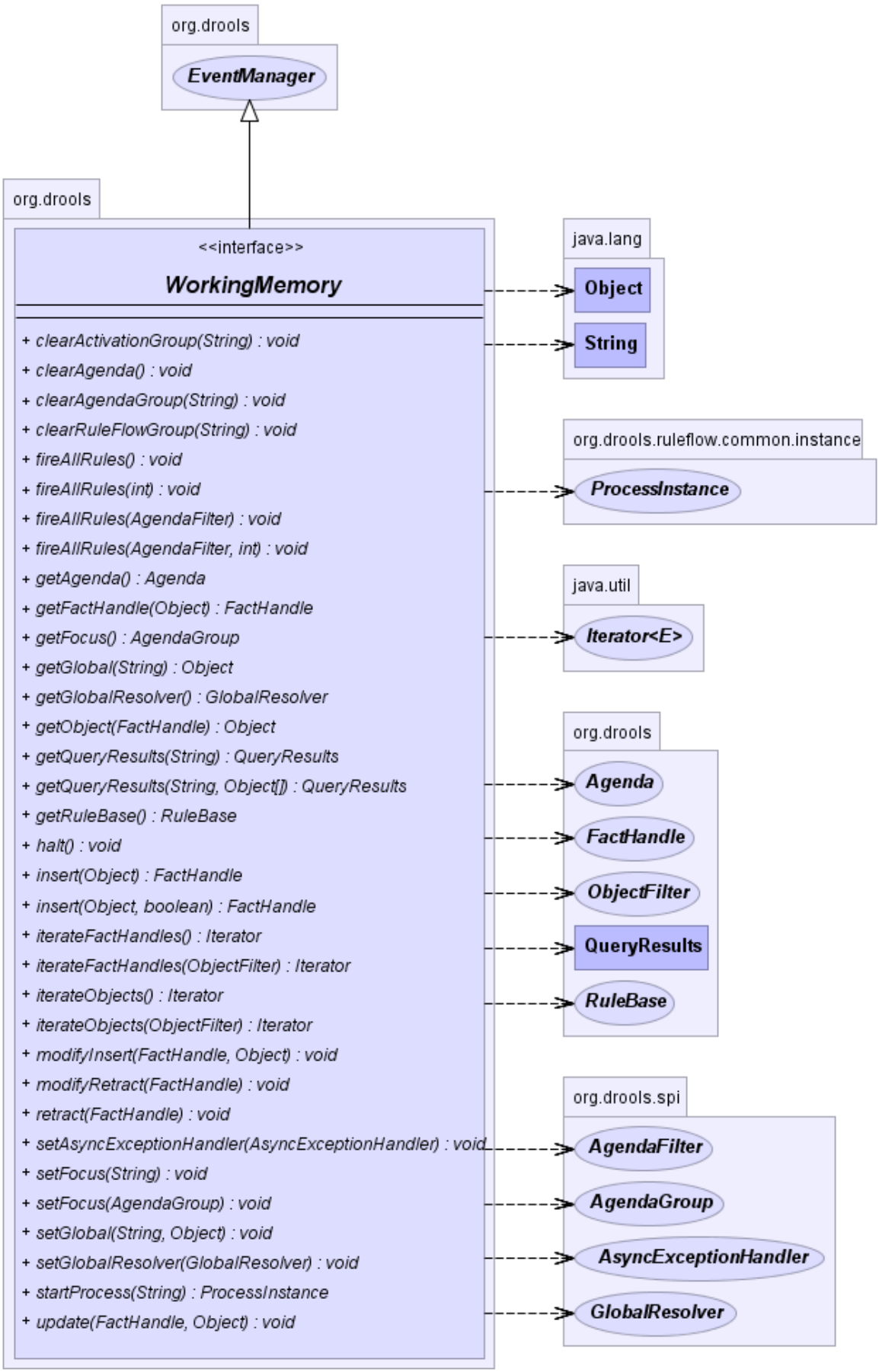


Figure 2.18. WorkingMemory

It holds references to all data that has been "inserted" into it (until retracted) and it is the place where the interaction with your application occurs. Working memories are stateful objects. They may be shortlived or longlived.

2.5.4.1. Facts

Facts are objects (beans) from your application that you insert into the working memory. Facts are any Java objects which the rules can access. The rule engine does not "clone" facts at all, it is all references/pointers at the end of the day. Facts are your applications data. Strings and other classes without getters and setters are not valid Facts and can't be used with Field Constraints which rely on the JavaBean standard of getters and setters to interact with the object.

2.5.4.2. Insertion

"Insert" is the act of telling the WorkingMemory about the facts. WorkingMemory.insert(yourObject) for example. When you insert a fact, it is examined for matches against the rules etc. This means ALL of the work is done during insertion; however, no rules are executed until you call "fireAllRules()". You don't call "fireAllRules()" until after you have finished inserting your facts. This is a common misunderstanding by people who think the work happens when you call "fireAllRules()". Expert systems typically use the term "assert" or "assertion" to refer to facts made available to the system, however due to the assert become a keyword in most languages we have moved to use the "Insert" keyword; so expect to hear the two used interchangeably.

When an Object is inserted it returns a FactHandle. This FactHandle is the token used to represent your insert Object inside the WorkingMemory, it is also how you will interact with the Working Memory when you wish to retract or modify an object.

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

As mentioned in the Rule Base section a Working Memory may operate in two assertions modes equality and identity - identity is default.

Identity means the Working Memory uses an IdentityHashMap to store all asserted Objects. New instance assertions always result in the return of a new FactHandle, if an instance is asserted twice then it returns the previous fact handle – i.e. it ignores the second insertion for the same fact.

Equality means the Working Memory uses a HashMap to store all asserted Objects. New instance assertions will only return a new FactHandle if a *not equal classes condition* has been asserted.

2.5.4.3. Retraction

"Retraction" is when you retract a fact from the Working Memory, which means it will no longer track and match that fact, and any rules that are activated and dependent on that fact will be cancelled. Note that it is possible to have rules that depend on the "non existence" of a fact, in which case retracting a fact may cause a rule to activate (see the 'not' and 'exist' keywords). Retraction is done using the FactHandle that was returned during the assert.

```
Cheese stilton = new Cheese("stilton");
FactHandle stiltonHandle = session.insert( stilton );
//...
session.retract( stiltonHandle );
```

2.5.4.4. Update

The Rule Engine must be notified of modified Facts, so that it can be re-process. Modification internally is actually a retract and then an insert; so it clears the WorkingMemory and then starts again. Use the modifyObject method to notify the Working Memory of changed objects, for objects that are not able to notify the Working Memory themselves. Notice modifyObject always takes the modified object as a second parameter - this allows you to specify new instances for immutable objects. The update() method can only be used with objects that have shadow proxies turned on. If you do not use shadow proxies then you must call session.modifyRestract() before making your changes and session.modifyInsert() after the changes.

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

2.5.4.5. Globals

Globals are named objects that can be passed in to the rule engine; without needing to insert them. Most often these are used for static information, or services that are used in the RHS of a rule, or perhaps a means to return objects from the rule engine. If you use a global on the LHS of a rule, make sure it is immutable. A global must first be declared in the drl before it can be set on the session.

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

With the Rule Base now aware of the global identifier and its type any sessions are now able to call session.setGlobal; failure to declare the global type and identifier first will result in an exception being thrown. to set the global on the session use session.setGlobal(identifier, value);

```
List list = new ArrayList();
session.setGlobal("list", list);
```

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

2.5.4.6. Shadow Facts

A shadow fact is a shallow copy of an asserted object. Shadow facts are cached copies of object asserted to the working memory. The term shadow facts is commonly known as a feature of JESS (Java Expert System Shell).

The origins of shadow facts traces back to the concept of truth maintenance. The basic idea is that an expert system should guarantee the derived conclusions are accurate. A running system may alter a fact during evaluation. When this occurs, the rule engine must know a modification occurred and handle the change appropriately. There are two ways to guarantee truthfulness. The first is to lock all the facts during the inference process. The second is to make a cache copy of an object and force all modifications to go through the rule engine. This way, the changes are processed in an orderly fashion. Shadow facts are particularly important in multi-threaded environments, where an engine is shared by multiple sessions. Without truth maintenance, a system has a difficult time proving the results are accurate. The primary benefit of shadow facts is it makes development easier. When developers are forced to keep track of fact modifications, it can lead to errors, which are difficult to debug. Building a moderately complex system using a rule engine is hard enough without adding the burden of tracking changes to facts and when they should notify the rule engine.

Drools 4.0 has full support for Shadow Facts implemented as transparent lazy proxies. Shadow facts are enable by default and are not visible from external code, not even inside code blocks on rules.

Note

Because Drools implements Shadow Facts as Proxies, the fact classes must either be immutable or should not be final, nor have final methods. If a fact class is final or have final methods and is still a mutable class, the engine is not able to create a proper shadow fact for it and results in unpredictable behavior.

Although shadow facts are a great way of ensuring the engine integrity, they add some overhead to the the reasoning process. As so, Drools 4.0 supports fine grained control over them with the ability to enable/disable them for each individual class.

2.5.4.6.1. When is it possible to disable Shadow Facts?

It is possible to disable shadow facts for your classes if you meet the following requirements:

1. Immutable classes are safe

   If a class is immutable it does not require shadow facts. Just to clarify, a class is immutable from the engine perspective if once an instance is asserted into the working memory, no attribute will change until it is retracted.

2. Inside your rules, attributes are only changed using modify() blocks

   Both Drools dialects (MVEL and Java) have the modify block construct. If all attribute value changes for a given class happen inside modify() blocks, you can disable shadow facts for that class.

Example 2.7. Example: modify() block using Java dialect

```
rule "Eat Cheese"
when
  $p: Person( status == "hungry" )
  $c: Cheese( )
then
  retract( $c );
  modify( $p ) {
      setStatus( "full" ),
      setAge( $p.getAge() + 1 )
  }
end
```

Example 2.8. Example: modify() block using MVEL dialect

```
rule "Eat Cheese"
  dialect "mvel"
when
  $p: Person( status == "hungry" )
  $c: Cheese( )
then
  retract( $c );
  modify( $p ) {
      status = "full",
      age = $p.age + 1
  }
end
```

3. In your application, attributes are only changed between calls to modifyRetract() and modifyInsert():

   In this case, the engine becomes aware that attributes will be changed and can prepare itself for them.

> Example 2.9. Example: safely modifying attributes in the application code
>
> ```
> // create session
> StatefulSession session = ruleBase.newStatefulSession();
>
> // get facts
> Person person = new Person( "Bob", 30 );
> person.setLikes( "cheese" );
>
> // insert facts
> FactHandle handle = session.insert( person );
>
> // do application stuff and/or fire rules
> session.fireAllRules();
>
> // wants to change attributes?
> // call modifyRetract() before doing changes
> session.modifyRetract( handle );
> person.setAge( 31 );
> person.setLikes( "chocolate" );
> // call modifyInsert() after the changes
> session.modifyInsert( handle, person );
> ```

2.5.4.6.2. How to disable Shadow Facts?

To disable shadow fact for all classes set the following property in a configuration file of system property:

```
drools.shadowProxy = false
```

Alternatively, it is possible to disable through an API call:

```
RuleBaseConfiguration conf = new RuleBaseConfiguration();
conf.setShadowProxy( false );
//...
RuleBase ruleBase = RuleBaseFactory.newRuleBase( conf );
```

To disable the shadow proxy for a list of classes only, use the following property instead, or the equivalent API:

```
drools.shadowproxy.exclude = org.domainy.* org.domainx.ClassZ
```

As shown above, a space separated list is used to specify more than one class, and '*' is used as a wild card.

2.5.4.7. Property Change Listener

If your fact objects are Java Beans, you can implement a property change listener for them, and then tell the rule engine about it. This means that the engine will automatically know when a fact has changed, and behave accordingly (you don't need to tell it that it is modified). There are proxy libraries that can help automate this (a future version of drools will bundle some to make it easier). To use the Object in dynamic mode specify true for the second assertObject parameter.

```
Cheese stilton = new Cheese("stilton");
//specifies that this is a dynamic fact
FactHandle stiltonHandle = workingMemory.insert( stilton, true );
```

To make a JavaBean dynamic add a PropertyChangeSupport field memory along with two add/remove mothods and make sure that each setter notifies the PropertyChangeSupport instance of the change.

```
private final PropertyChangeSupport changes = new PropertyChangeSupport( this );

public void addPropertyChangeListener(final PropertyChangeListener l) {
    this.changes.addPropertyChangeListener( l );
}

public void removePropertyChangeListener(final PropertyChangeListener l) {
    this.changes.removePropertyChangeListener( l );
}

public void setState(final String newState) {
    String oldState = this.state;
    this.state = newState;
    this.changes.firePropertyChange( "state",oldState,newState );
```

2.5.4.8. Initial Fact

To support conditional elements like "not" (which will be covered in *Chapter 5, The Rule Language*), there is a need to "seed" the engine with something known as the "Initial Fact". This fact is a special fact that is not intended to be seen by the user.

On the first working memory action (assert, fireAllRules) on a fresh working memory, the Initial Fact will be propagated through the RETE network. This allows rules that have no LHS, or perhaps do not use normal facts (such as rules that use "from" to pull data from an external source). For instance, if a new working memory is created, and no facts are asserted, calling the fireAllRules will cause the Initial Fact to propagate, possibly activating rules (otherwise, nothing would happen as there area no other facts to start with).

2.5.5. StatefulSession

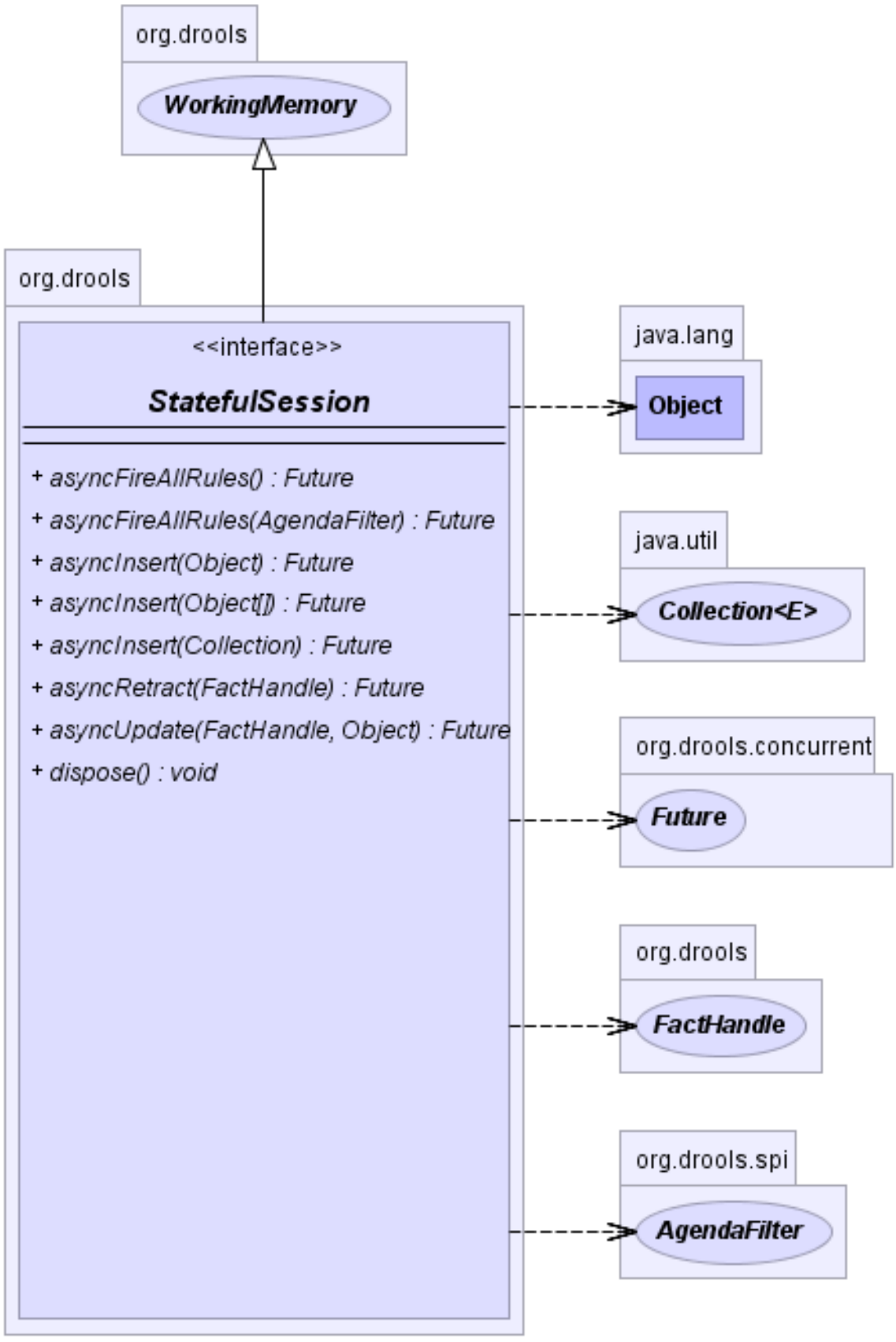


Figure 2.19. StatefulSession

The StatefulSession extends the WorkingMemory class. It simply adds async methods and a dispose() method. The ruleBase retains a reference to each StatefulSession is creates, so that it can update them when new rules are added, dispose() is needed to release the StatefulSession reference from the RuleBase, without it you can get memory leaks.

Example 2.10. Createing a StatefulSession

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

2.5.6. Stateless Session

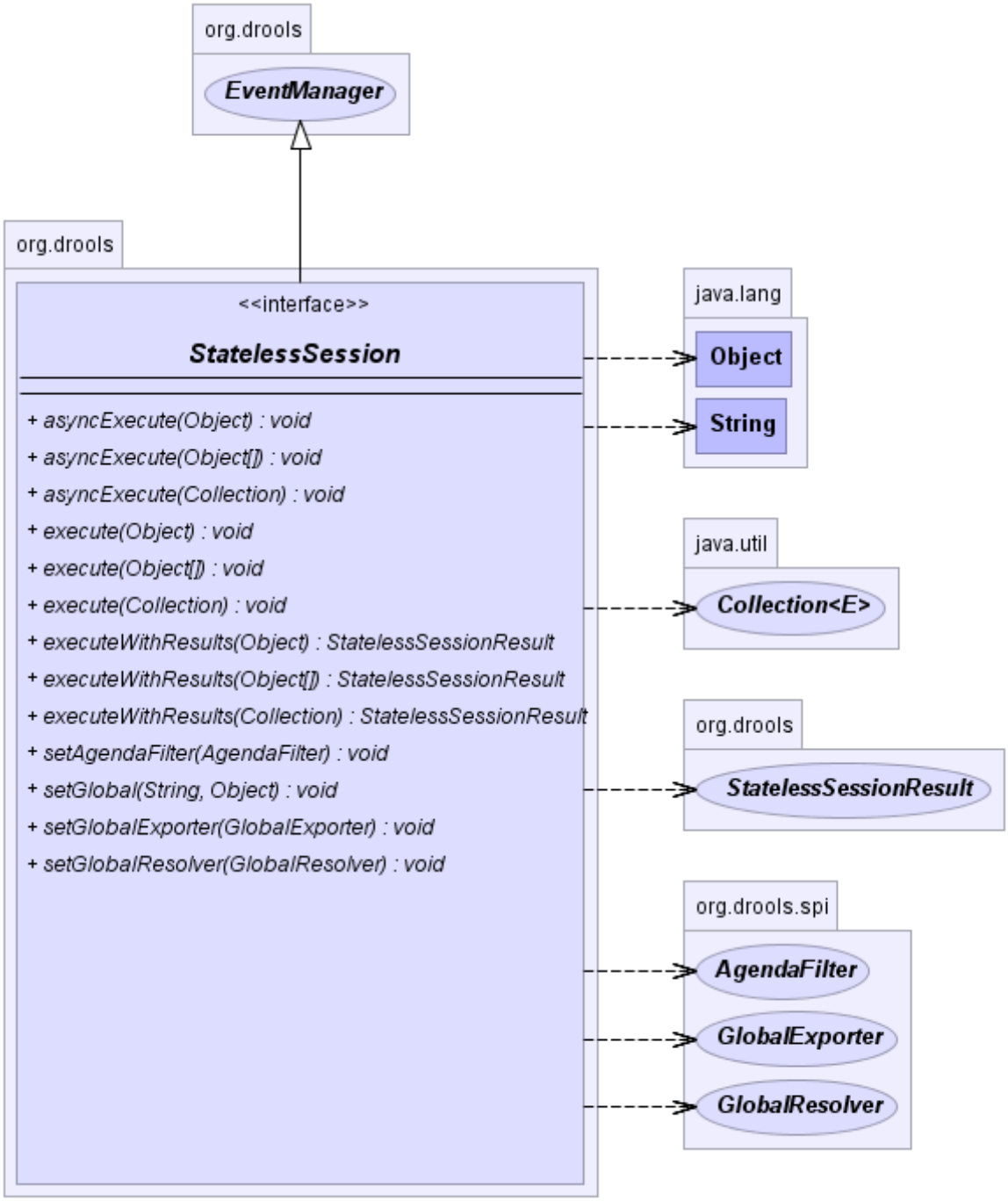


Figure 2.20. StatelessSession

The StatelessSession wraps the WorkingMemory, instead of extending it, its main focus is on decision service type scenarios.

Example 2.11. Createing a StatelessSession

```
StatelessSession session = ruleBase.newStatelessSession();
session.execute( new Cheese( "cheddar" ) );
```

The api is reduced for the problem domain and is much simpler making maintenance of those services easier. The RuleBase never retains a reference to the **`StatelessSession`**, thus dispose() is not needed, and they only have an execute() method that takes an object, an array of objects or a collection of objects - there is no insert or fireAllRules. The execute method iterates the objects inserting each and calling fireAllRules() at the end when the session is finished. Should the session need access to any results information they can use the executeWithResults method, which returns a StatelessSessionResult. The reason for this is in remoting situations you do not always want the return payload, so this way its optional.

setAgendaFilter, setGlobal and setGlobalResolver share their state across sessions; so each call to execute() will use the set AgendaFilter, or see any previous set globals etc.

StatelessSessions do not currently support propertyChangeListeners.

Async versions of the Execute method are supported, remember to override the ExecutorService implementation when in special managed thread environments such as JEE.

StatelessSessions also support sequential mode, which is a special optimised mode that uses less memory and executes faster. Refer to *Section 2.5.10, “Sequential Mode”* for more details.

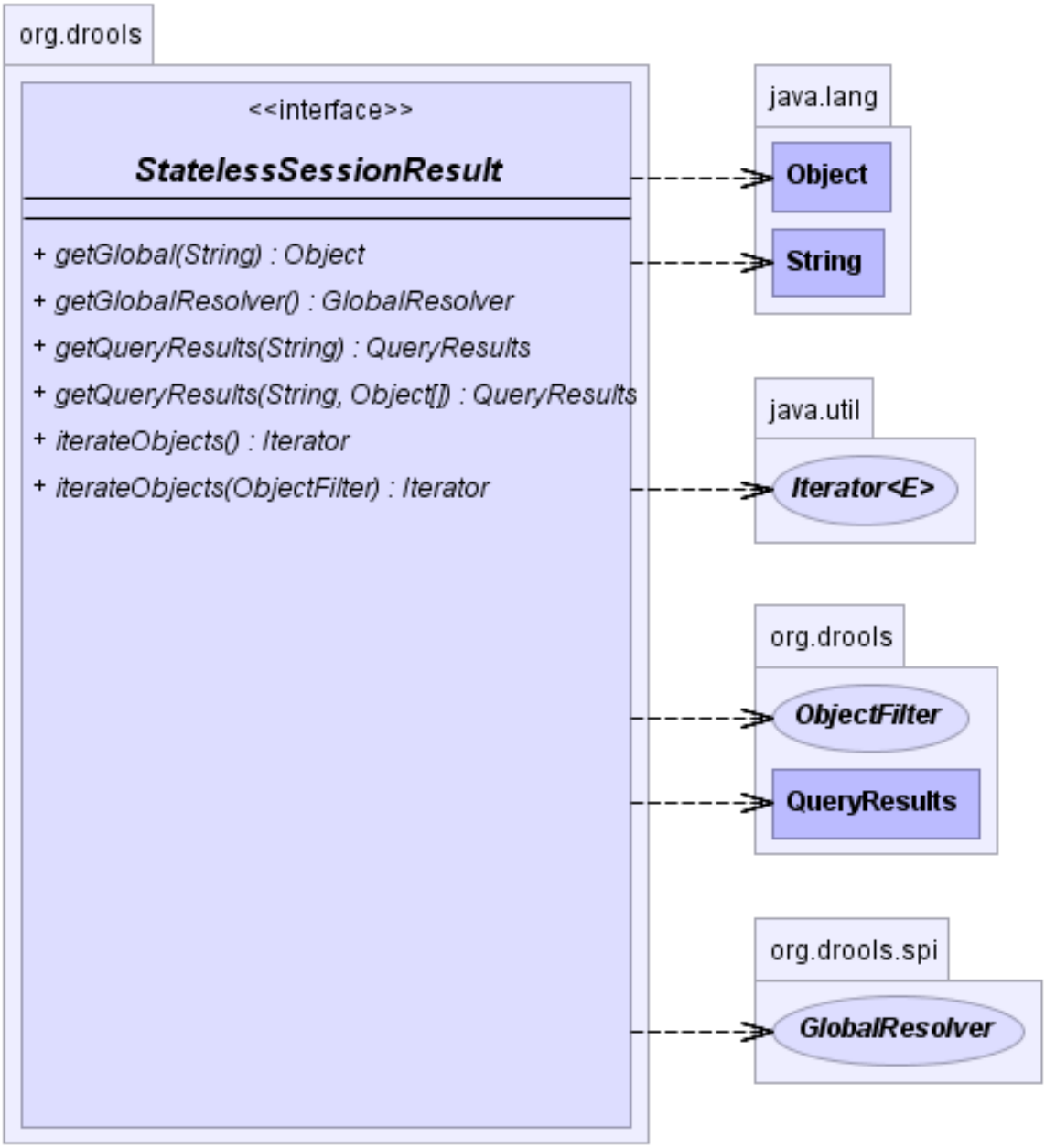


Figure 2.21. StatelessSessionResult

StatelessSession.executeWithResults(....) returns a minimal api to examine the sessions data. The inserted Objects can be iterated over, queries can be executed and globals retrieved. Once the StatelessSessionResult is serialized it loses the reference to the underlying WorkingMemory and RuleBase, so queries can no longer be executed, however globals can still be retrieved and objects iterated. To retrieve globals they must be exported from the StatelessSession; the GlobalExporter strategy is set with StatelessSession.setGlobalExporter(GlobalExporter globalExporter). Two implementations of GlobalExporter are available and users may implement their own strategies. CopyIdentifiersGlobalExporter copies named identifiers into a new GlobalResovler that is passed to the StatelessSessionResult; the constructor takes a String[] array of identifiers, if no identifiers are specified it copies all identifiers declaredin the RuleBase. ReferenceOriginalGlobalExporter just passes a reference to the original Global Resolver; the later should be used with care as identifier instances can be changed at any time by the StatelessSession and the GlobalResolver may not be able to be serialized.

Example 2.12. GlobalExporter with StatelessSessions

```
StatelessSession session = ruleBase.newStatelessSession();
session.setGlobalExporter( new CopyIdentifiersGlobalExporter( new String[]{"list"} ) );
StatelessSessionResult result = session.executeWithResults( new Cheese( "stilton" ) );
List list = ( List ) result.getGlobal( "list" );
```

2.5.7. Agenda

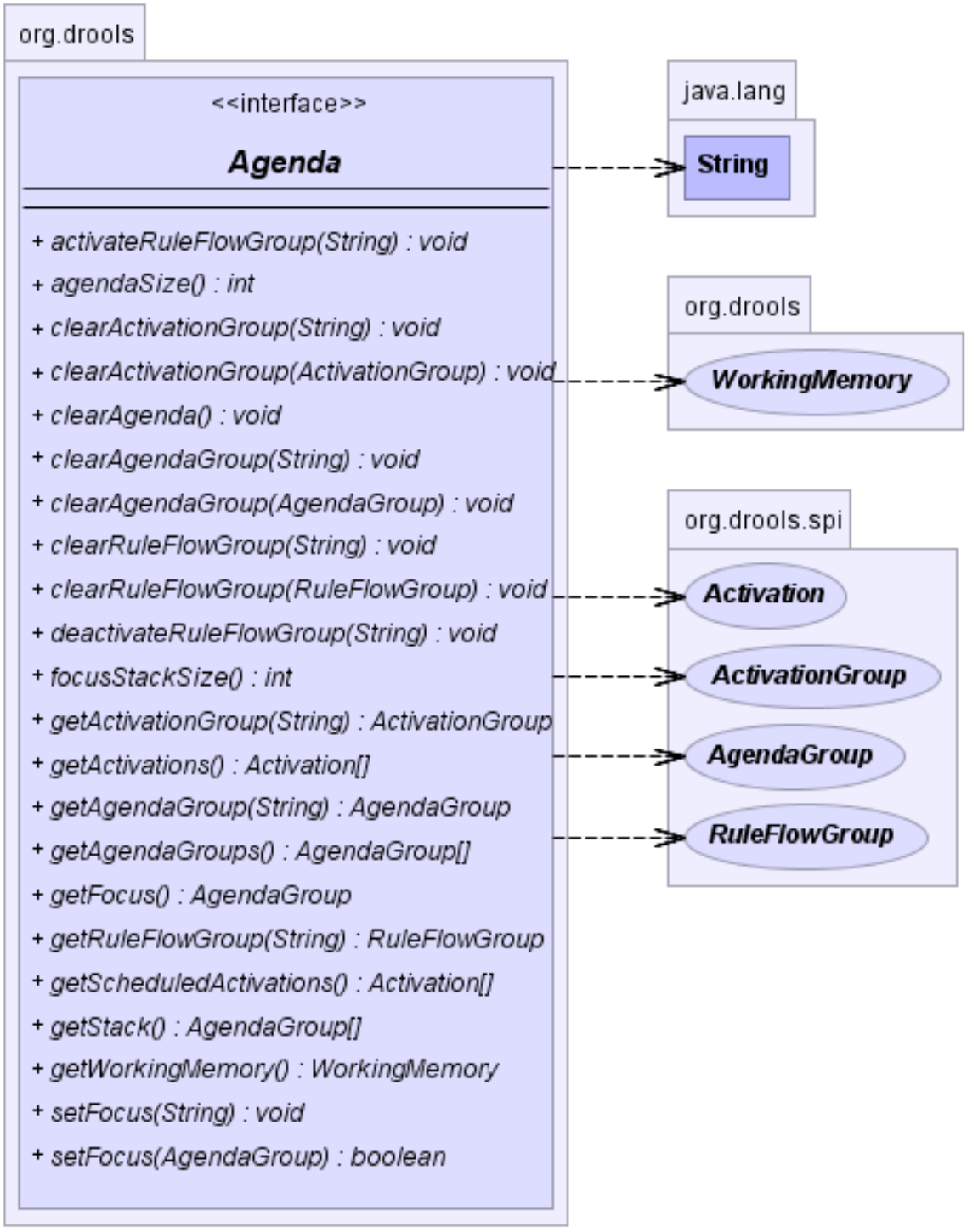


Figure 2.22. Two Phase Execution

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

The engine operates in a "2 phase" mode which is recursive:

1. Working Memory Action Phase

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

2. Agenda Evaluation Phase

   The engine attempts to select a rule to fire. If a rule is not found it exits, otherwise it attempts to fire the rule, switching the phase back to Working Memory Actions. The process repeats again until the Agenda is empty.

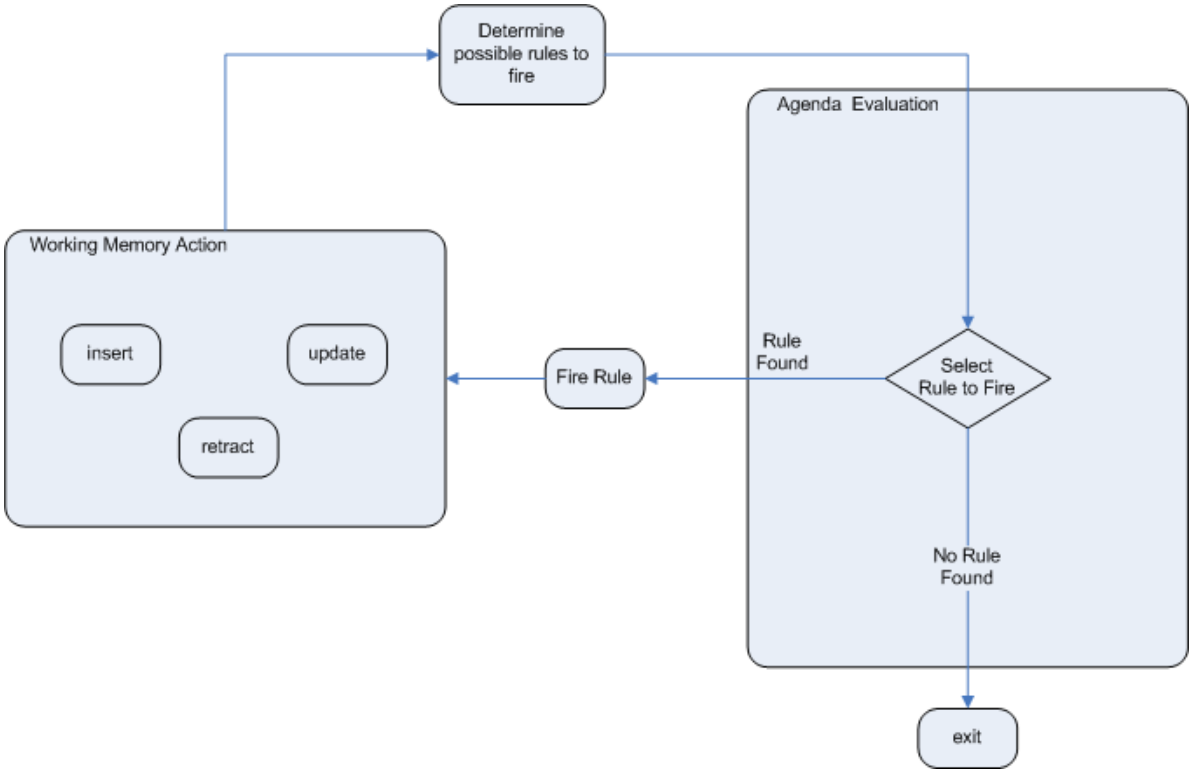


Figure 2.23. Two Phase Execution

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

2.5.7.1. Conflict Resolution

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

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

The most visible strategy is "salience" or priority. You can specify that a certain rule has a higher priority by giving it a higher number then other rules. In that case, the higher salience rule will always be preferred.

LIFO prioritizes based on the assigned Working Memory Action counter value. Multiple rules created from the same action have the same value and execution of these are considered arbitrary.

Custom conflict resolution strategies can be specified by setting the Class in the **`RuleBaseConfiguration`** method `setConflictResolver`, or using the property drools.conflictResolver.

Important

Although it is sometimes unavoidable, you should avoid relying on your rules being fired in a specific order. Remember that you should not be authoring rules as though they are steps in a imperative process.

2.5.7.2. Agenda Groups

Sometimes known as "modules" in CLIPS terminology, Agenda groups are a way to partition the activation of rules on the agenda. At any one time, only one group has "focus" which means that only the activations for rules in that group will take effect. You can also "auto focus" a rule, causing its agenda group to gain "focus" when that rule's conditions are true.

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 when these sets of rules apply.

You can change the group which has focus either from within the rule engine, or from the API. Each time `setFocus()` is called it pushes that Agenda Group onto a stack. When the group is empty it is popped off and the next one on the stack evaluates. An Agenda Group can appear in multiple locations on the stack. The default Agenda Group is `MAIN`, all rules which do not specify an Agenda Group are placed there, it is also always the first group on the stack and given focus as default.

2.5.7.3. Agenda Filters

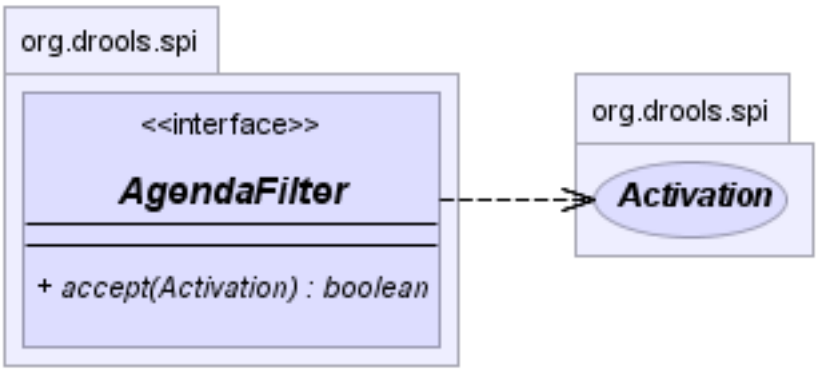


Figure 2.24. AgendaFilters

Filters are optional implementations of the filter interface which are used to allow/or deny an activation from firing. What you filter is entirely up to your implementation. Drools provides the following convenience default implementations

- RuleNameEndWithAgendaFilter
- RuleNameEqualsAgendaFilter
- RuleNameStartsWithAgendaFilter
- RuleNameMatchesAgendaFilter

To use a filter specify it while calling FireAllRules. The following example will filter out all rules ending with the text "Test":

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

2.5.8. Truth Maintenance with Logical Objects

In a regular insert, you need to explicitly retract a fact. With logical assertions, the fact that was asserted will be automatically retracted when the conditions that asserted it in the first place are no longer true and there are no possible conditions that could support the logical assertion.

Normal insertions are said to be *STATED*. Using a HashMap and a counter we track how many times a particular equality is STATED, essentially counting how many different instances are equal. When we logically insert an object we are said to *JUSTIFY* it and it is justified by the firing rule. For each logical insertion there can only be one equal object, each subsequent equal logical insertion increases the justification counter for this logical assertion. As each justification is removed when we have no more justifications the logical object is automatically retracted.

If we logically insert an object when there is an equal STATED object it will fail and return null. If we STATE an object that has an exist equal object that is JUSTIFIED we override the Fact. Eaxtly how this override works depends on the configuration setting WM_BEHAVIOR_PRESERVE. When the property is set to discard the default behaviour is to use the existing handle and replace the existing instance with the new Object. Otherwise we override it to STATED but we create an new *FactHandle*.

This can be confusing on a first read, so hopefully the flow charts below help. When it says that it returns a new FactHandle, this also indicates the Object was propagated through the network.

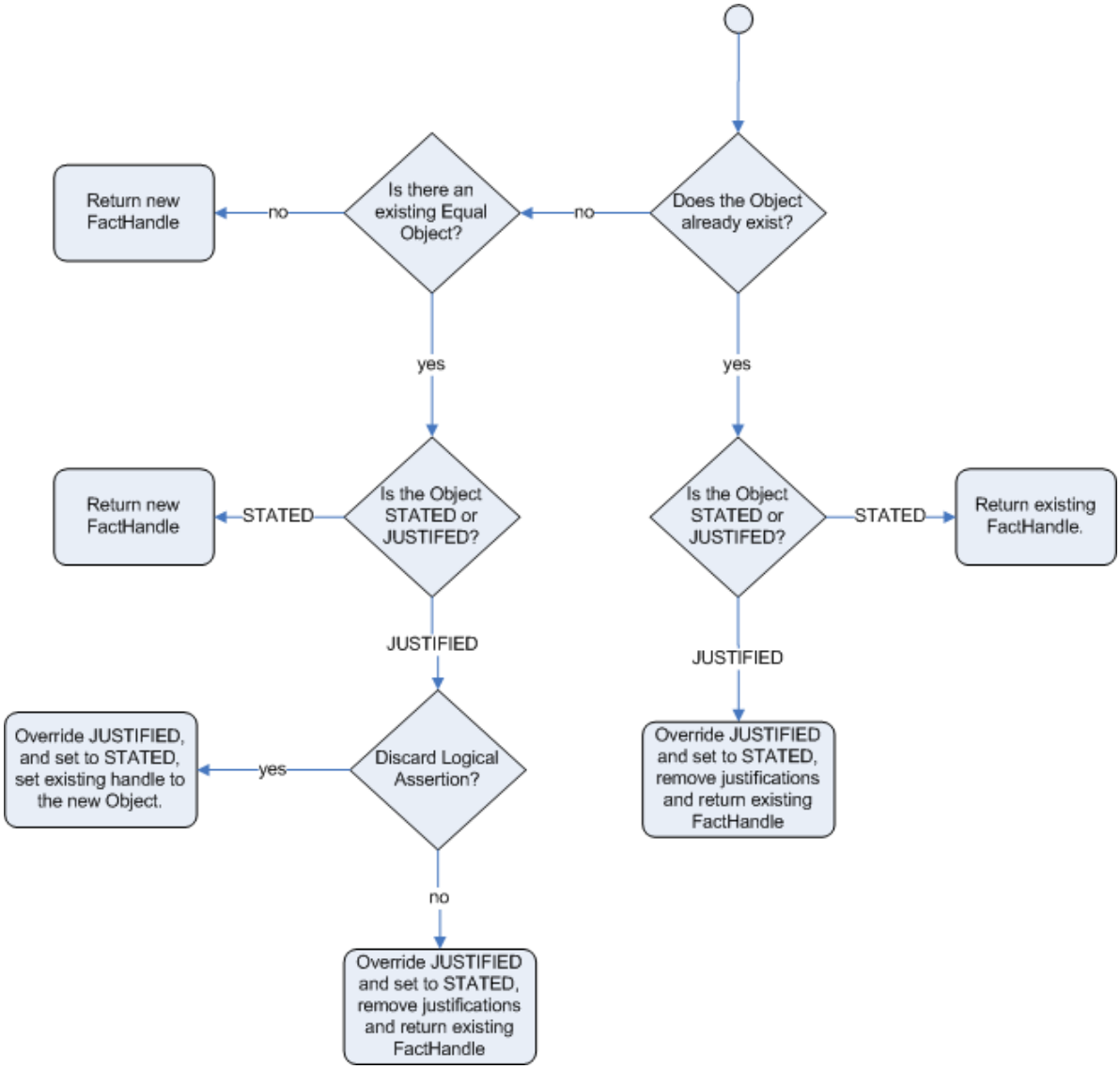


Figure 2.25. Stated Insertion

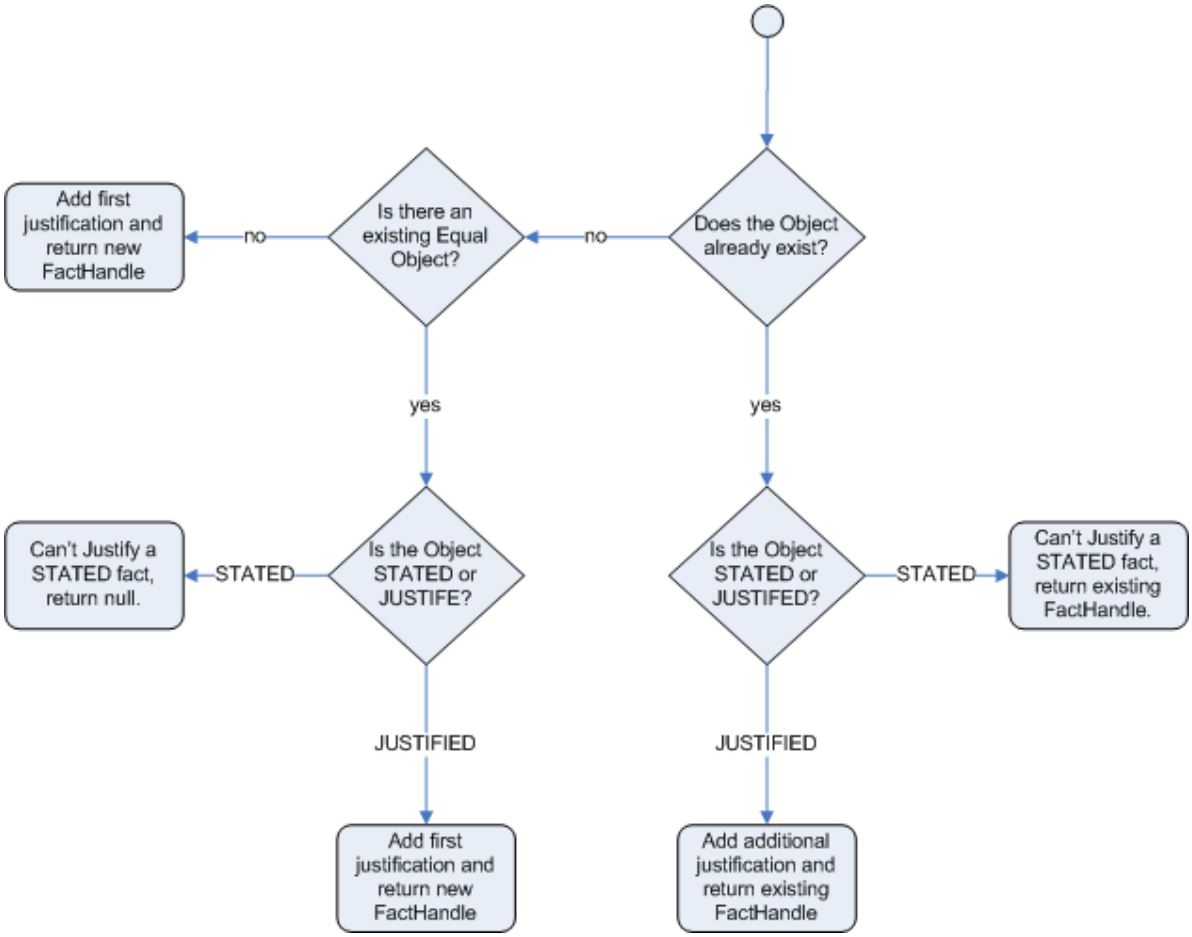


Figure 2.26. Logical Insertion

2.5.8.1. Example Scenario

An example may make things clearer. Imagine a credit card processing application, processing transactions for a given account and we have a working memory accumulating knowledge about a single accounts transaction. The rule engine is doing its best to decide if transactions are possibly fraudulent or not. Imagine this rule base basically has rules that activate when there is "reason to be suspicious" and when "everything is normal".

Of course there are many rules that operate no matter what performing standard calculations and other actions.There are possibly many reasons as to what could trigger a "reason to be suspicious": someone notifying the bank, a sequence of large transactions, transactions for geographically disparate locations or even reports of credit card theft. Rather then smattering all the little conditions in lots of rules, imagine there is a fact class called **`SuspiciousAccount`**.

Then there can be a series of rules whose job is to look for things that may raise suspicion, and if they fire, they simply insert a new SuspiciousAccount() instance. All the other rules just have conditions like "not SuspiciousAccount()" or "SuspiciousAccount()" depending on their needs. Note that this has the advantage of allowing there to be many rules around raising suspicion, without touching the other rules. When the facts causing the SuspiciousAccount() insertion are removed, the rule engine reverts back to the normal "mode" of operation and, for instance, a rule with "not SuspiciousAccount()" may activate to flush through any interrupted transactions.

If you have followed this far, you will note that truth maintenance, like logical assertions, allows rules to behave a little like a human would, and can certainly make the rules more manageable.

2.5.8.2. Important note: Equality for Java objects

It is important to note that for Truth Maintenance (and logical assertions) to work at all, your Fact objects (which may be Javabeans) must override equals and hashCode methods (from java.lang.Object) correctly. As the truth maintenance system needs to know when 2 different physical objects are equal in value, BOTH equals and hashCode must be overridden correctly, as per the Java standard.

Two objects are equal if and only if their equals methods return true for each other and if their hashCode methods return the same values. See the Java API for more details (but do keep in mind you MUST override both equals and hashCode).

2.5.9. Event Model

The event package provides machanisms to be notified of rule engine events such as rules firing & objects being asserted. This allows you to separate out the logging and auditing activities from the main part of your application as well as your rules. This seperation is important as as events are a concern which will cover many different parts of your application.

There are three types of event listeners:

1. WorkingMemoryEventListener
2. AgendaEventListener
3. RuleFlowEventListener

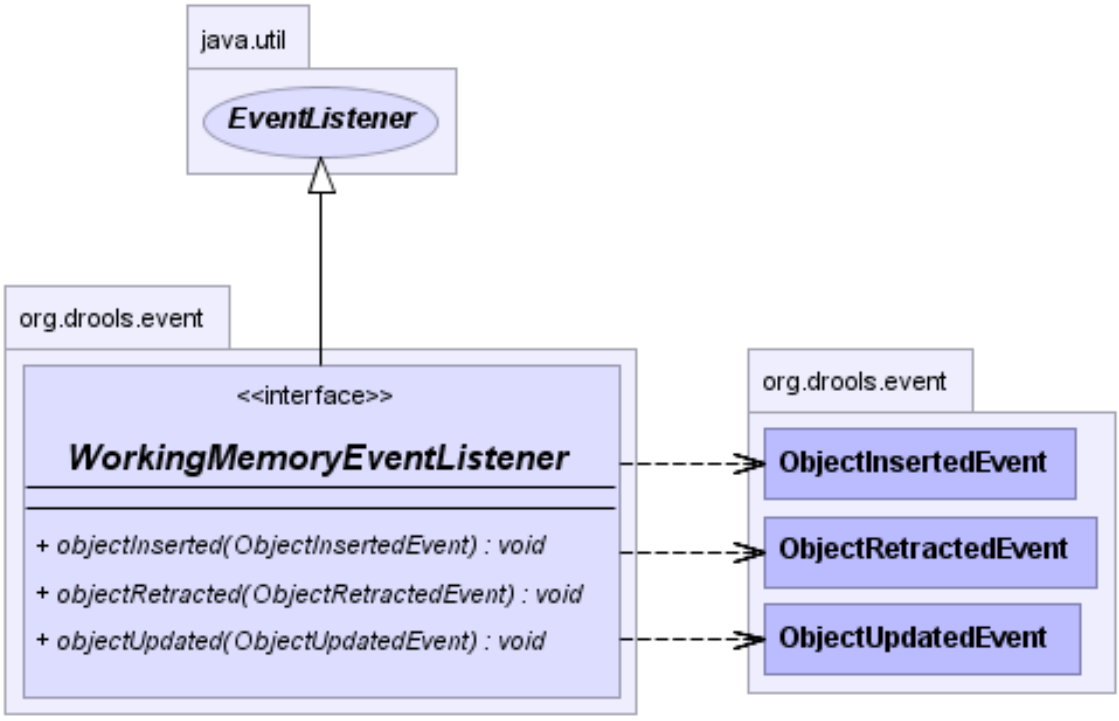


Figure 2.27. WorkingMemoryEventListener

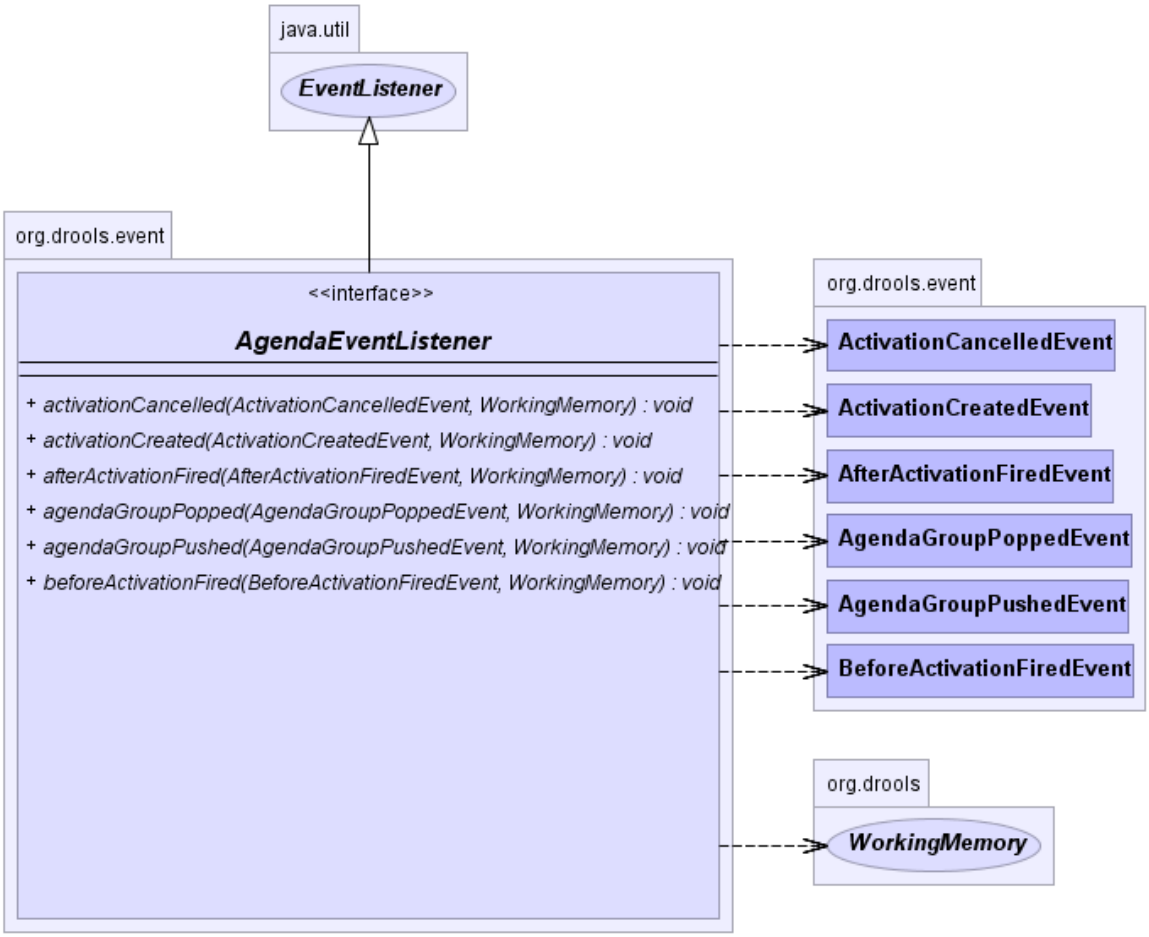


Figure 2.28. AgendaEventListener

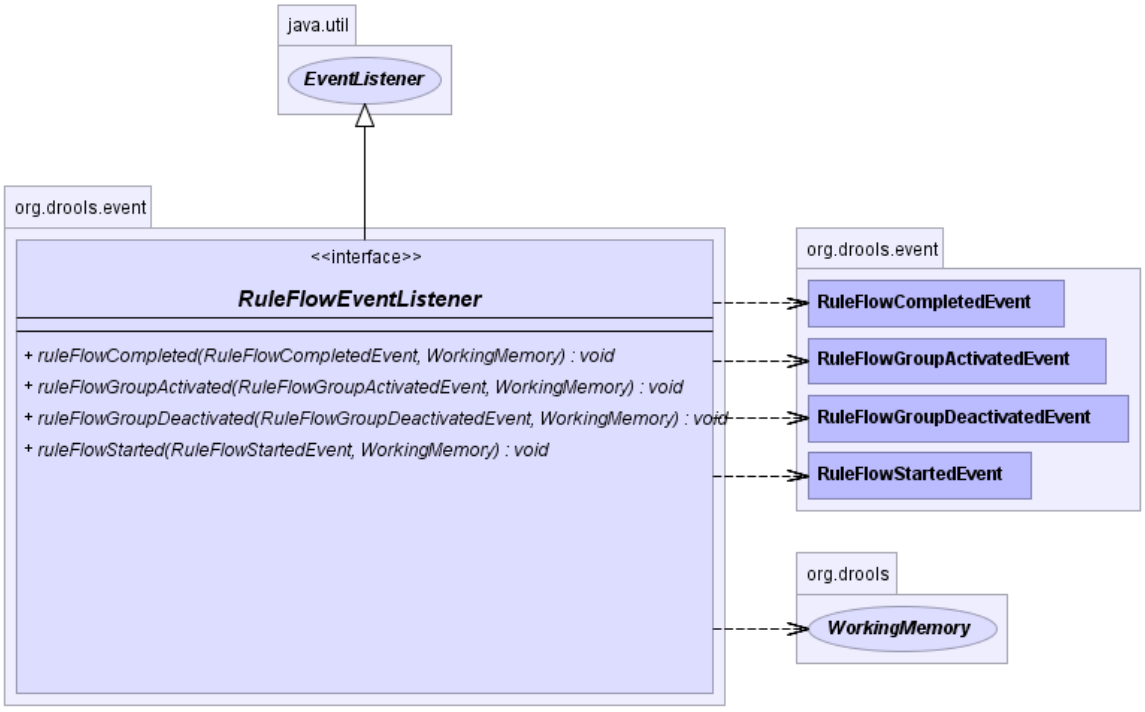


Figure 2.29. RuEventListener

Both stateful and statless sessions implement the EventManager interface, which allows event listeners to be added to the session.

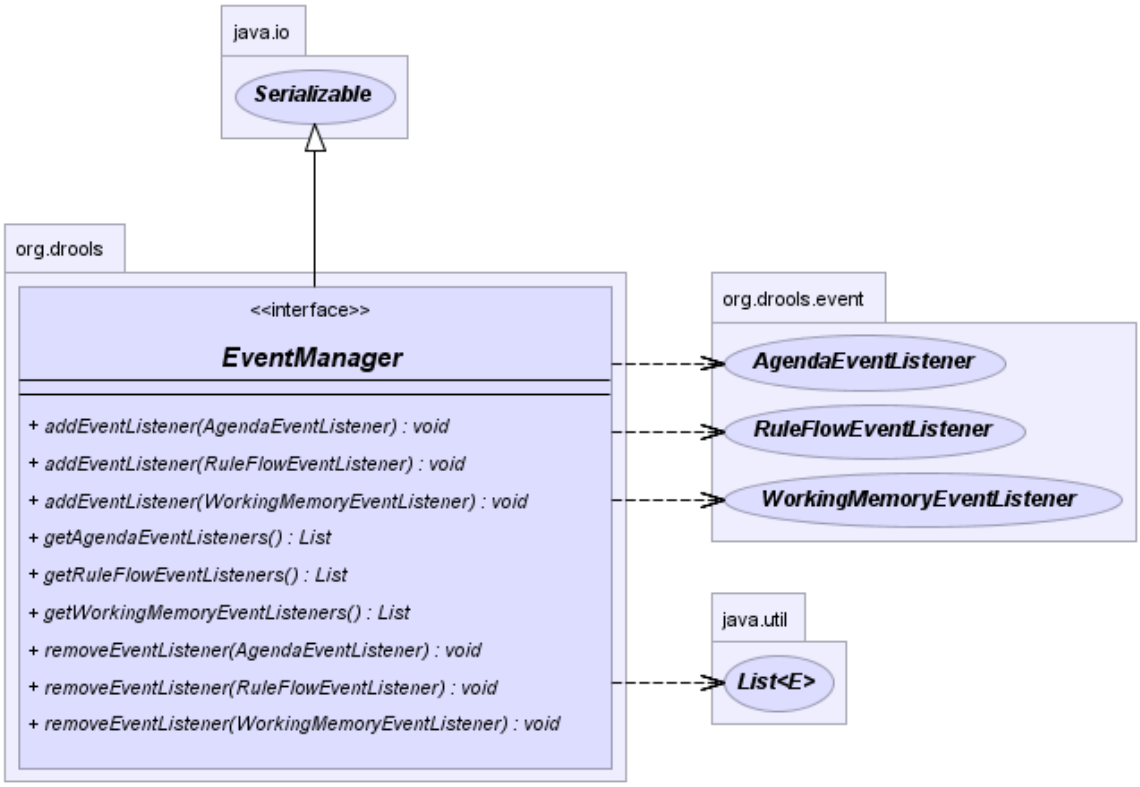


Figure 2.30. EventManager

All **`EventListeners`** have default implementations that implement each method, but do nothing, these are convienience classes that you can inherit from to save having to implement each method (DefaultAgendaEventListener, DefaultWorkingMemoryEventListener,DefaultRuleFlowEventListener). The following shows how to extend DefaultAgendaEventListener and add it to the session, the example prints statements for only when rules are fired:

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

Drools also provides DebugWorkingMemoryEventListener, DebugAgendaEventListener and DebugRuleFlowEventListener that implements each method with a debug print statement:

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

The Eclipse based Rule IDE also provides an audit logger and graphical viewer, so that the rule engine can log events for later viewing, and auditing.

2.5.10. Sequential Mode

With Rete you have a stateful session where objects can be asserted and modified over time, and rules can be added and removed. If we assume a stateless session, where after the initial data set

no more data can be asserted or modified (no rule re-evaluations) and rules cannot be added or removed, we can start to make assumptions to minimize what work the engine has to do.

1. Order the Rules by salience and position in the ruleset (just sets a sequence attribute on the rule terminal node).
2. Create an array, one element for each possible rule activation; element position indicates firing order.
3. Turn off all node memories, except the right-input Object memory.
4. Disconnect the LeftInputAdapterNode propagation, and have the Object plus the Node referenced in a Command object, which is added to a list on the WorkingMemory for later execution.
5. Assert all objects, when all assertions are finished and thus right-input node memories are populated check the Command list and execute each in turn.
6. All resulting Activations should be placed in the array, based upon the determined sequence number of the Rule. Record the first and last populated elements, to reduce the iteration range.
7. Iterate the array of Activations, executing populated element in turn.
8. If we have a maximum number of allowed rule executions, we can exit our network evaluations early to fire all the rules in the array.

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

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

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

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

Decision Tables

3.1. Decision Tables in Spreadsheets

Read this section to learn about *decision tables* and the ways in which they 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 one manage rules by storing them in a spreadsheet format, such as **CSV** or Microsoft Excel **.XLS**. Hence, one can use a spreadsheet program (such as **OpenOffice.org Calc** to manage these files.)

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.

3.1.1. 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.

3.1.2. Overview

Here are some examples of real-life decision tables:

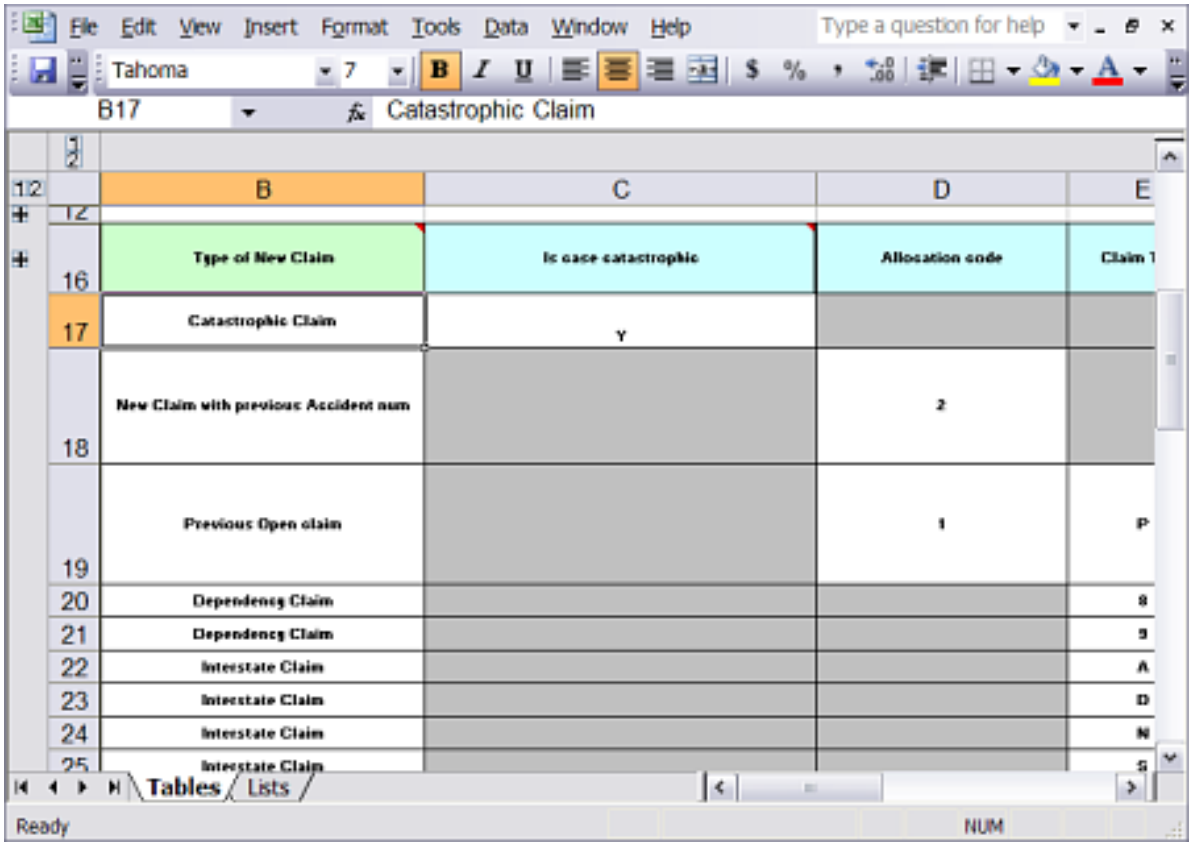


Figure 3.1. Using Excel to Edit a Decision Table

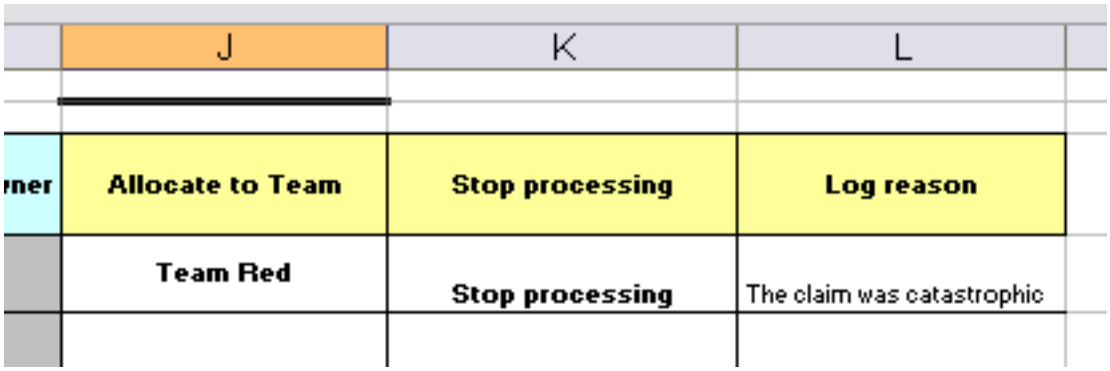
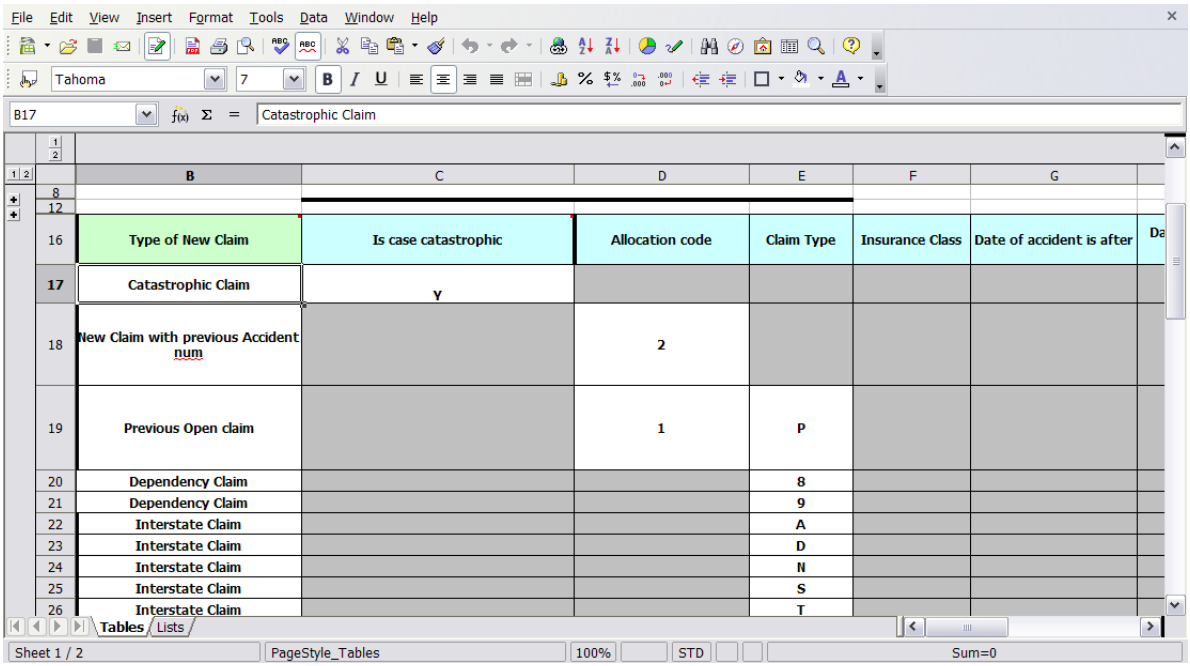


Figure 3.2. Multiple Actions for a Rule Row



	B	C	D	E	F	G
16	Type of New Claim	Is case catastrophic	Allocation code	Claim Type	Insurance Class	Date of accident is after
17	Catastrophic Claim	Y				
18	New Claim with previous Accident num		2			
19	Previous Open claim		1	P		
20	Dependency Claim			8		
21	Dependency Claim			9		
22	Interstate Claim			A		
23	Interstate Claim			D		
24	Interstate Claim			N		
25	Interstate Claim			S		
26	Interstate Claim			T		

Figure 3.3. 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 is a convention to use colour to indicate the meanings of different areas of the table.

Important

Although the decision tables look like they process 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

One 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.)

	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	ValidarAperturaCajaSucursalAbierta	Esta Regla tiene por Mision Validar que la sucursal de l se encuentre abierta Trabaja sobre la Caja que se intenta abrir, la Sucursa corresponde a esa caja y la Transacción de Ca apertura
		2	2000	ValidarAperturaCajaMismaFecha	Esta Regla tiene por Mision Validar que en la sucursal caja se encuentre abierta para la misma fecha de ape de la caja. Trabaja sobre la Caja que se intenta abrir, la Sucursa corresponde a esa caja y la Transacción de Ca 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 moment efectuarse el Cierre Contable de una Sucursal de FO todas las Cajas de esta última se encuentren en E Cerrado, es decir la Fecha de Cierre de Caja debe se a la Fecha de cierre de la entidad Registro_Cierre_Suc
3.ValidarTransaccionCaja(Caja, Transaccion_Caja)					
RuleTable[3] ValidarTransaccionCaja(CajaVO caja, MovimientoCajaVO movimientoCaja)					
ID_Caso de Uso	Caso de Uso	Identificador	Prioridad	Nombre	Descripcion

Figure 3.4. A Real-Life Example Using Multiple Tables to Group Like Rules

3.1.3. How Decision Tables Work

The key point to keep in mind is that in a decision table, each row is a rule, and each column in that row is either a condition or action for that rule.

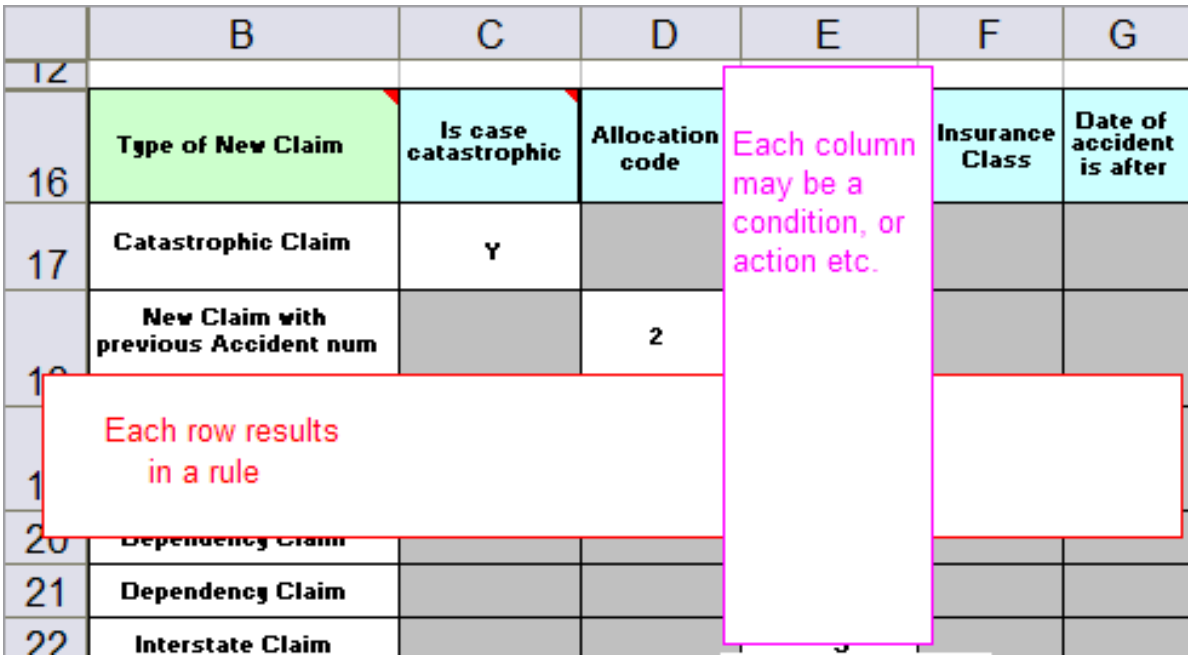


Figure 3.5. Rows and Columns

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 to see more if this helps in understanding it. Note the keywords in Column "C."

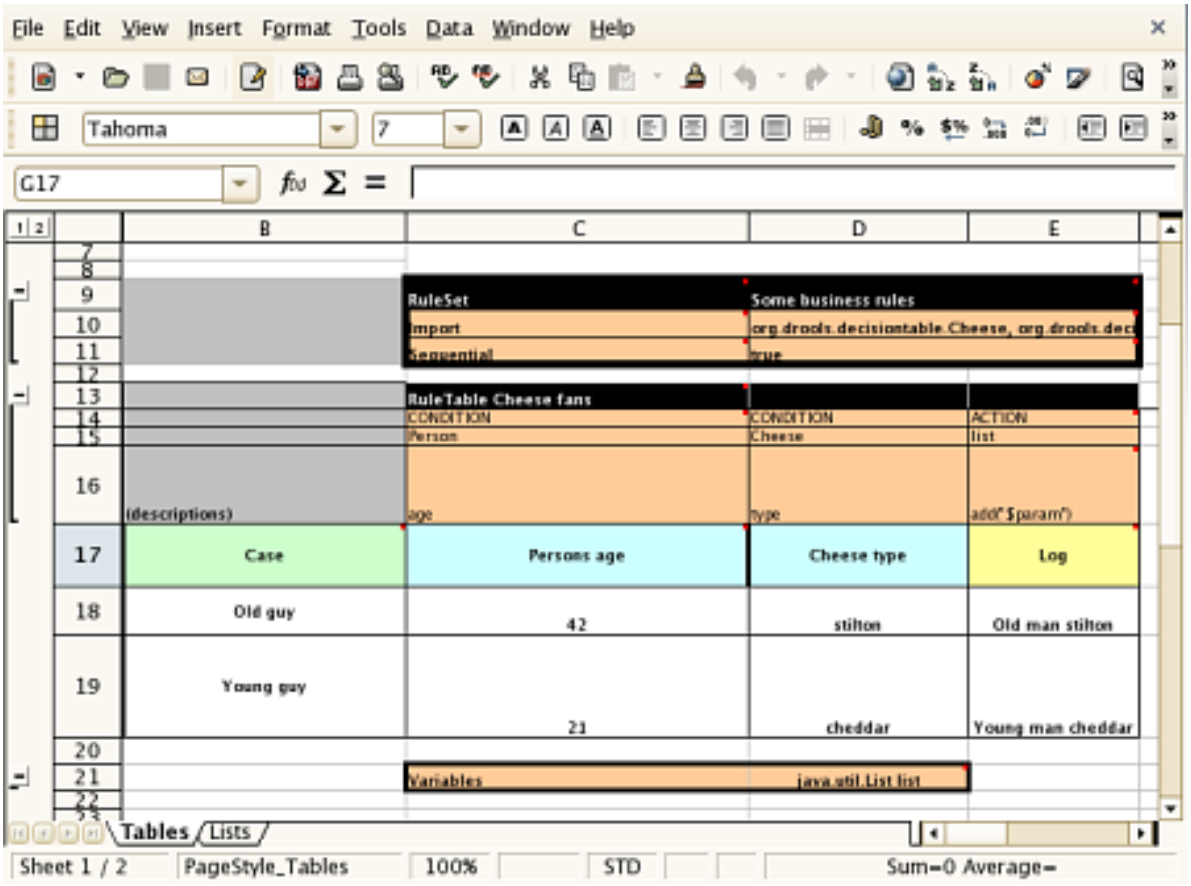


Figure 3.6. Expanded for Rule Templates

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 are 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, so if one does not intend to use it, leave a blank row. 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: note that they can use the **`$para`** place holder to indicate where data from the cells below is to populate. Use **`$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".)

3.1.4. Keywords and Syntax

3.1.4.1. 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 to indicate the point where data from the cell is to be interpolated. (**`$1`** can also be used for the same purpose. If the cell contains a comma-separated list of values, **`$1`**,

$2 and so forth may be used to indicate which positional parameter from the list of values in the cell is to be used.)

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 [42,42] then the result will be
 [Foo(bar > 42, baz ==42)]
```

For conditionals, the way in which snippets are rendered is dependent on the presence or absence of `ObjectType` declarations in the row above. If they are present, then the snippets are rendered as individual constraints on that `ObjectType`. If there are not any, then they are simply rendered as is (with values substituted.) If one adds a plain field (as in the example above) then it will assume equality is meant. Put another operator at the end of the snippet to force the values to be interpolated at the end of the constraint or it will look for **`$param`** as stated above.

For consequences, the way in which snippets are rendered is dependent on the presence or absence of anything in the row immediately above. If there is nothing there, 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 3.7. 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.

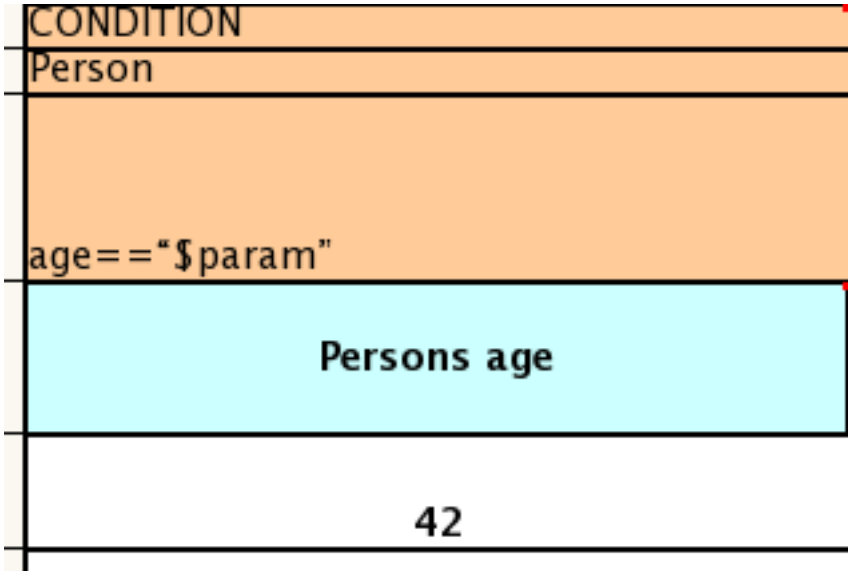


Figure 3.8. With Parameters

In this example, interpolation is used to place the values in the snippet (the result being **`Person(age == "42")`**.)

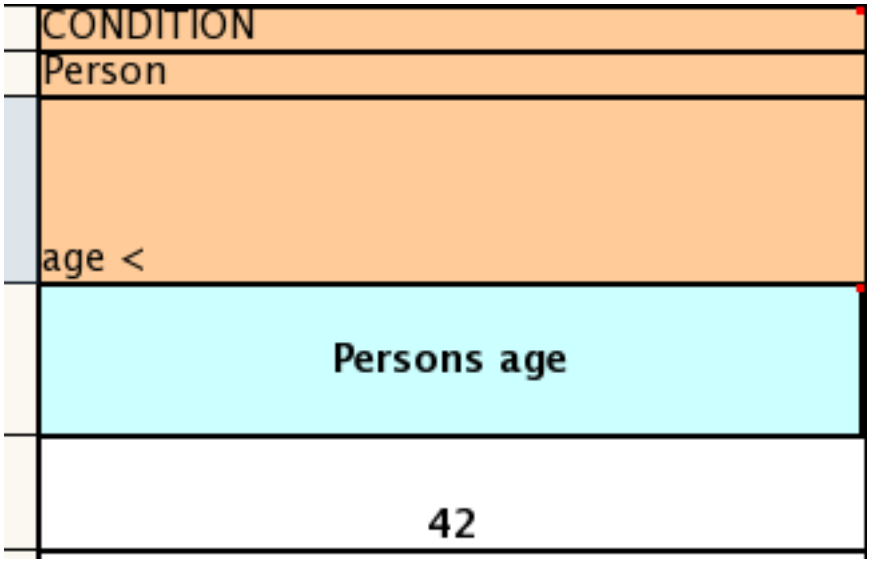


Figure 3.9. Operator Completion

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.

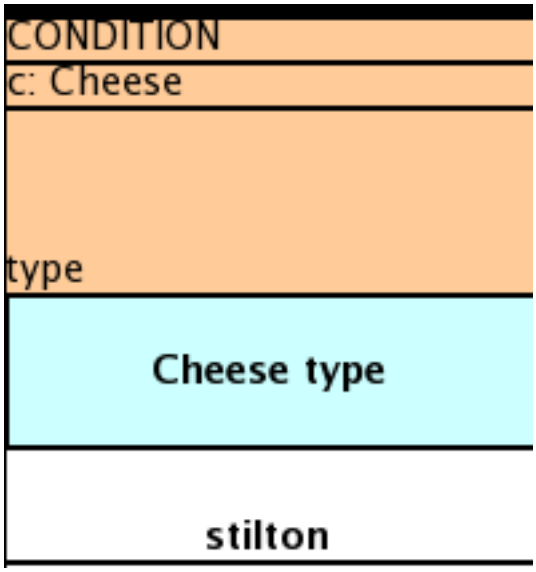


Figure 3.10. With Binding

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.)

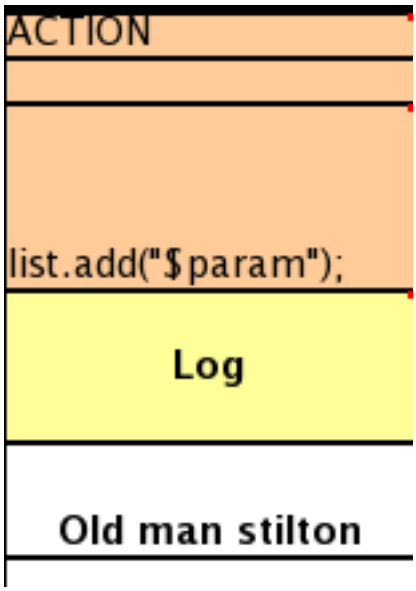


Figure 3.11. Consequence

This final example shows how the consequence can be achieved by a very simple form of interpolation by just 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.

3.1.4.2. Keywords

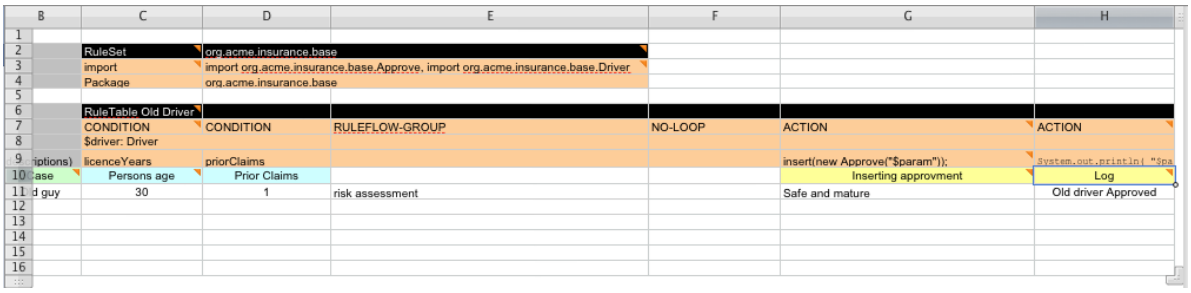
The following table describes the valid keywords that can be used.

Table 3.1. Keywords

Keyword	Description	Is mandatory?
RuleSet	The cell to the right of this contains the name of the rule-set	One only (if left out, it will default)

Keyword	Description	Is mandatory?
Sequential	The cell to the right of this can be true or false. If true, then salience is used to ensure that rules fire from the top down	optional
Import	The cell to the right contains a comma separated list of java classes to import	optional
RuleTable	A cell starting with RuleTable indicates the start of a definition of a rule table. The actual rule table starts the next row down. The rule table is read left-to-right, and top-down, until there is one BLANK ROW.	at least one. if there are more, then they are all added to the one ruleset
CONDITION	Indicates that this column will be for rule conditions	At least one per rule table
ACTION	Indicates that this column will be for rule consequences	At least one per rule table
PRIORITY	Indicates that this columns values will set the 'salience' values for the rule row. Over-rides the 'Sequential' flag.	optional
DURATION	Indicates that this columns values will set the duration values for the rule row.	optional
NAME	Indicates that this columns values will set the name for the rule generated from that row	optional
Functions	The cell immediately to the right can contain functions which can be used in the rule snippets. 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, comma separate them).	optional
NO-LOOP	if there is a column with the this keyword, the no-loop attribute will be set for the rule. Setting no-loop column means the	optional

Keyword	Description	Is mandatory?
	attempt to create the Activation for the current set of data will be ignored.	
ACTIVATION-GROUP	Cell values in this column mean that the rule-row belongs to the given XOR/activation group . An Activation group means that only one rule in the named group will fire (ie the first one to fire cancels the other rules activations).	optional
RULEFLOW-GROUP	Cell values in this column mean that the rule-row belongs to the given RULEFLOW-GROUP. Ruleflow allows you to specify the order in which rule sets should be evaluated by using a flow chart. See the documentation for ruleflow on this manual.	optional
Worksheet	By default, the first worksheet is only looked at for decision tables.	N/A



	B	C	D	E	F	G	H
1							
2		RuleSet	org.acme.insurance.base				
3		import	import org.acme.insurance.base.Approve, import org.acme.insurance.base.Driver				
4		Package	org.acme.insurance.base				
5							
6		RuleTable Old Driver					
7		CONDITION	CONDITION	RULEFLOW-GROUP	NO-LOOP	ACTION	ACTION
8		$driver: Driver					
9	iptions)	licenceYears	priorClaims			insert(new Approve("$param"));	System.out.println("$pa
10	ase	Persons age	Prior Claims			Inserting approvment	Log
11	d guy	30	1	risk assessment		Safe and mature	Old driver Approved
12							
13							
14							
15							
16							

Figure 3.12. Example usage of keywords for imports, functions etc.

RuleSet	Control Cajas[1]
Import	foo.Bar, bar.Baz
Variables	Parameters parametros, RulesResult resultado, EvalDate fecha

Functions	function boolean isRango(int iValor, int iRangoInicio, int iRangoFinal) {
	if (iRangoInicio <= iValor && iValor <= iRangoFinal)
	return true;
	return false;
	}
	
	function boolean isIgualTipo(TipoVO tipoVO, int p_tipo, boolean isNull) {
	if (tipoVO == null)
	return isNull;
	return tipoVO.getSecuencia().intValue() == p_tipo;
	}

Figure 3.13. Example usage of keywords for imports, functions etc.

3.1.5. Creating and Integrating Spreadsheet-Based Decision Tables

The API to use spreadsheet based decision tables is in the JBoss Rules-decisiontables module. There is really only one class to look at: SpreadsheetCompiler. This class will take spreadsheets in various formats, and generate rules in DRL (which you can then use in the normal way). Also note that if you like you can just use the SpreadsheetComiler to generate partial rule files, and assemble it into a complete rule package after the fact (this allows to you seperate technical and non technical aspects of the rules if needed).

To get started, you can find a sample spreadsheet and base it on that. Alternatively, if you are using the plug in (Rule Workbench IDE) the wizard can generate a spreadsheet for you from a template (to edit it you will need to use an xls compatible spreadsheet editor).

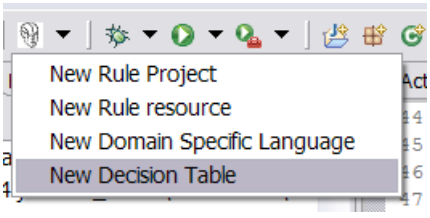


Figure 3.14. Wizard in the IDE

3.1.6. Managing business rules in decision tables.

3.1.6.1. Workflow and collaboration.

Spreadsheets are well established business tools (in use for over 25 years). Decision tables lend themselves to close collaboration between IT and domain experts, while making the business rules clear to business analysts, it is an ideal separation of concerns.

Typically, the whole process of authoring rules (coming up with a new decision table) would be something like:

1. Business analyst takes a template decision table (from a repository, or from IT)
2. Decision table business language descriptions are entered in the table(s)
3. Decision table rules (rows) are entered (roughly)
4. Decision table is handed to a technical resource, who maps the business language (descriptions) to scripts (this may involve software development of course, if it is a new application or data model !)
5. Technical person hands back and reviews the modifications with the business analyst.
6. The business analyst can continue editing the rule rows as needed (moving columns around is also fine etc).
7. In parallel, the technical person can develop test cases for the rules (liaising with business analysts) as these test cases can be used to verify rules and rule changes once the system is running.

3.1.6.2. Using spreadsheet features

You can use the features of applications like Excel to provide assistance in entering data into spreadsheets, such as validating fields. You can use lists that are stored in other worksheets to provide valid lists of values for cells, like in the following diagram.

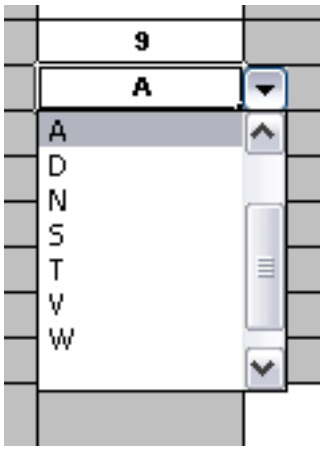


Figure 3.15. Wizard in the IDE

Some applications provide a limited ability to keep a history of changes, but it is recommended that an alternative means of revision control is also used. So when you are making changes to rules over time, older versions are archived (many solutions exist for this which are also open source, such as Subversion). http://www.JBoss Rules.org/Business+rules+in+decision+tables+explained

The (Eclipse based) Rule IDE

4.1. Introduction

The IDE provides developers (and very technical users) with an environment to edit and test rules in various formats, and integrate it deeply with their applications.

The Drools IDE is delivered as an eclipse plugin, which allows you to author and manage rules from within Eclipse, as well as integrate rules with your application. This is an optional tool, and not all components are required to be used, you can use what components are relevant to you. The Drools IDE is also a part of the JBoss Developer Studio (formerly known as JBoss IDE).

This guide will cover some of the features of JBoss Drools, in as far as the IDE touches on them (it is assumed that the reader has some familiarity with rule engines, and Drools in particular. It is important to note that none of the underlying features of the rule engine are dependent on Eclipse, and integrators are free to use their tools of choice, as always ! Plenty of people use IntelliJ with rules, for instance.

Note you can get the plug in either as a zip to download, or from an update site (refer to the chapter on installation).

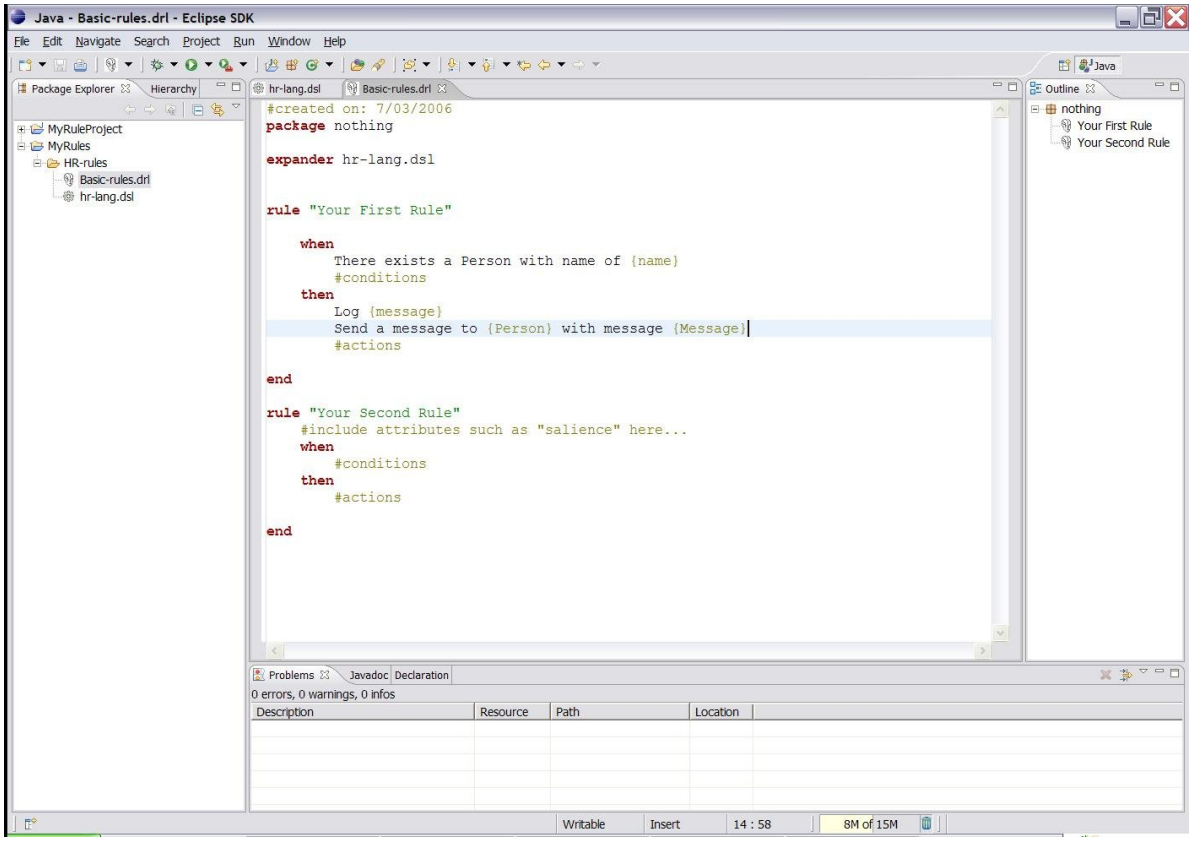


Figure 4.1. Overview

4.1.1. Features outline

The rules IDE has the following features

1. Textual/graphical rule editor

a. An editor that is aware of DRL syntax, and provides content assistance (including an outline view)

b. An editor that is aware of DSL (domain specific langauge) extensions, and provides content assistance.

2. RuleFlow graphical editor

   You can edit visual graphs which represent a process (a rule flow). The RuleFlow can then be applied to your rule package to have imperative control.

   RuleFlow is not supported by the JBoss Enterprise SOA Platform for BPM workflow or service orchestration. Refer to *Section 5.8, "Rule Flow"* for more details.

3. Wizards to accelerate and ...

   a. Help you quickly create a new "rules" project

   b. Create a new rule resource

   c. Create a new Domain Specific language

   d. Create a new decision table, guided editor, ruleflow

4. A domain specific language editor

   a. Create and manage mappings from your users language to the rule language

5. Rule validation

   a. As rules are entered, the rule is "built" in the background and errors reported via the problem "view" where possible

You can see the above features make use of Eclipse infrastructure and features. All of the power of eclipse is available.

4.1.2. Creating a Rule project

The aim of the new project wizard is to setup an executable scaffold project to start using rules immediately. This will setup a basic structure, classpath and sample rules and test case to get you started.

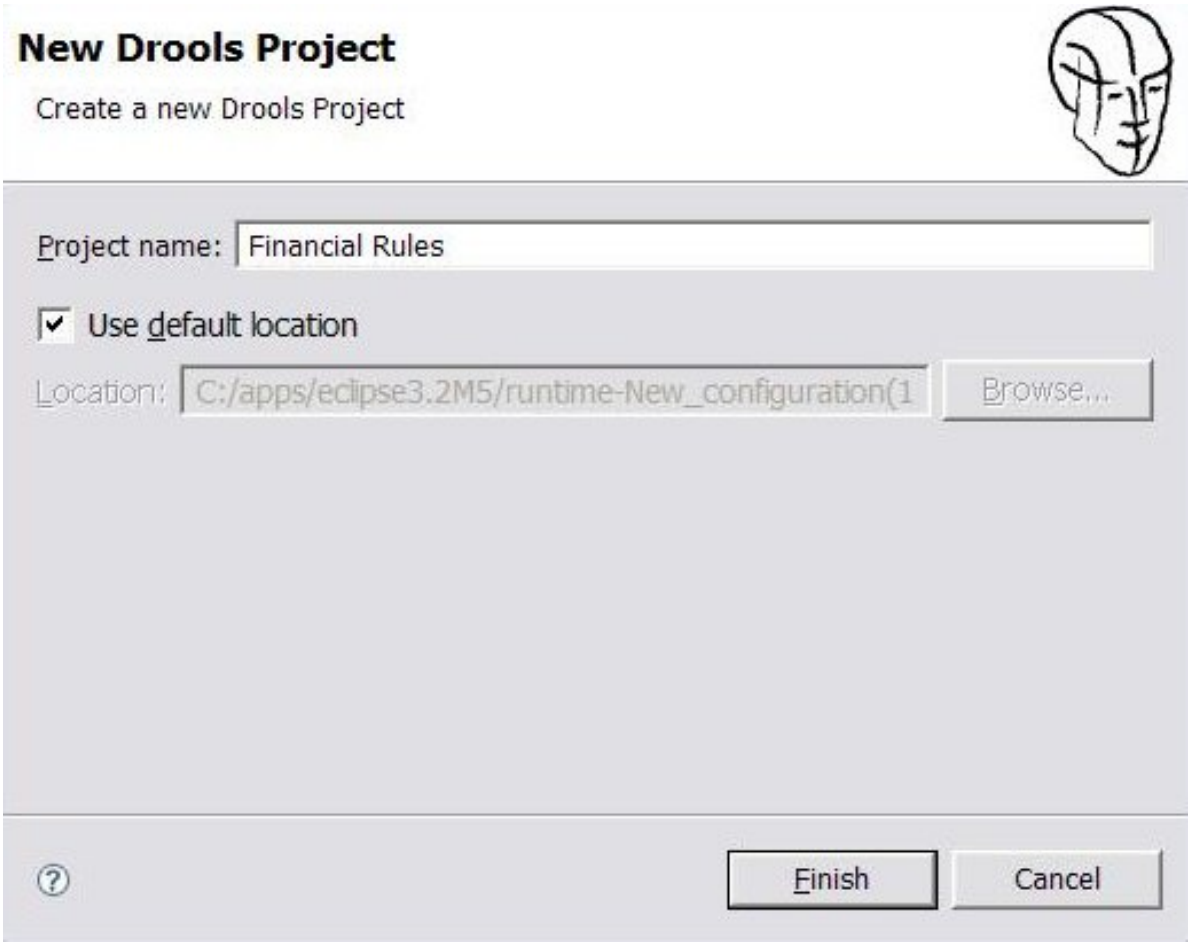


Figure 4.2. New rule project scaffolding

When you choose to create a new "rule project" - you will get a choice to add some default artifacts to it (like rules, decision tables etc). These can serve as a starting point, and will give you something executable to play with (which you can then modify and mould to your needs). The simplest case (a hello world rule) is shown below. Feel free to experiment with the plugin at this point.

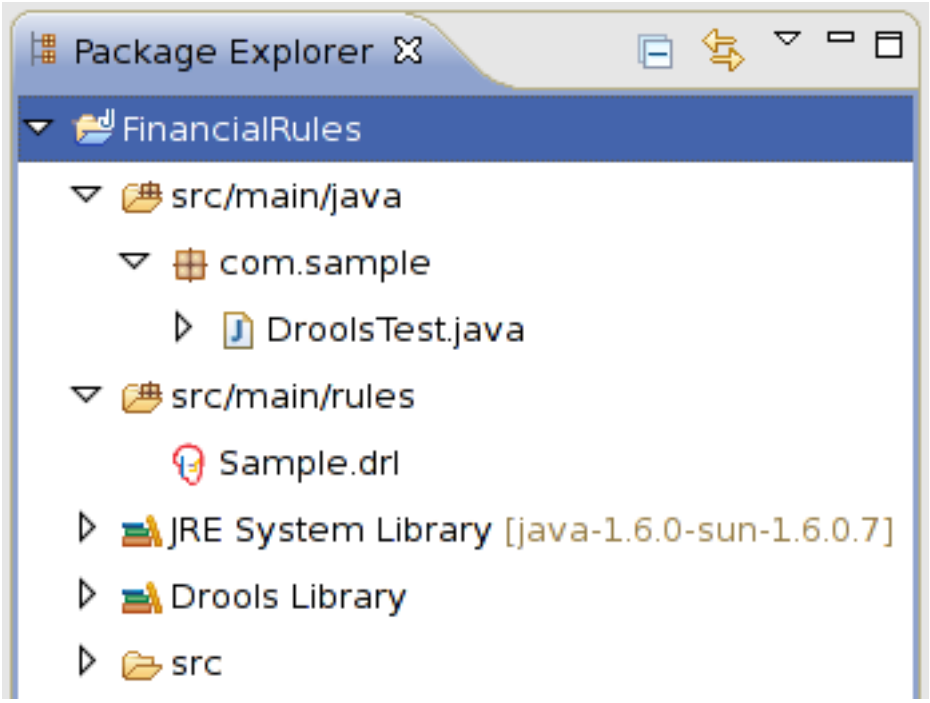


Figure 4.3. New rule project result

The newly created project contains an example rule file (Sample.drl) in the src/rules dir and an example java file (DroolsTest.java) that can be used to execute the rules in a Drools engine in the folder src/java, in the com.sample package. All the others jars that are necessary during execution are also added to the classpath in a custom classpath container called Drools Library. Rules do not have to be kept in "java" projects at all, this is just a convenience for people who are already using eclipse as their Java IDE.

Important note: The Drools plug in adds a "Drools Builder" capability to your eclipse instance. This means you can enable a builder on any project that will build and validate your rules when resources change. This happens automatically with the Rule Project Wizard, but you can also enable it manually on any project. One downside of this is if you have rule files that have a large number of rules (>500 rules per file) it means that the background builder may be doing a lot of work to build the rules on each change. An option here is to turn off the builder, or put the large rules into .rule files, where you can still use the rule editor, but it won't build them in the background - to fully validate the rules you will need to run them in a unit test of course.

4.1.3. Creating a new rule and wizards

You can create a rule simple as an empty text ".drl" file, or use the wizard to do so. The wizard menu can be invoked by Control+N, or choosing it from the toolbar (there will be a menu with the JBoss Drools icon).

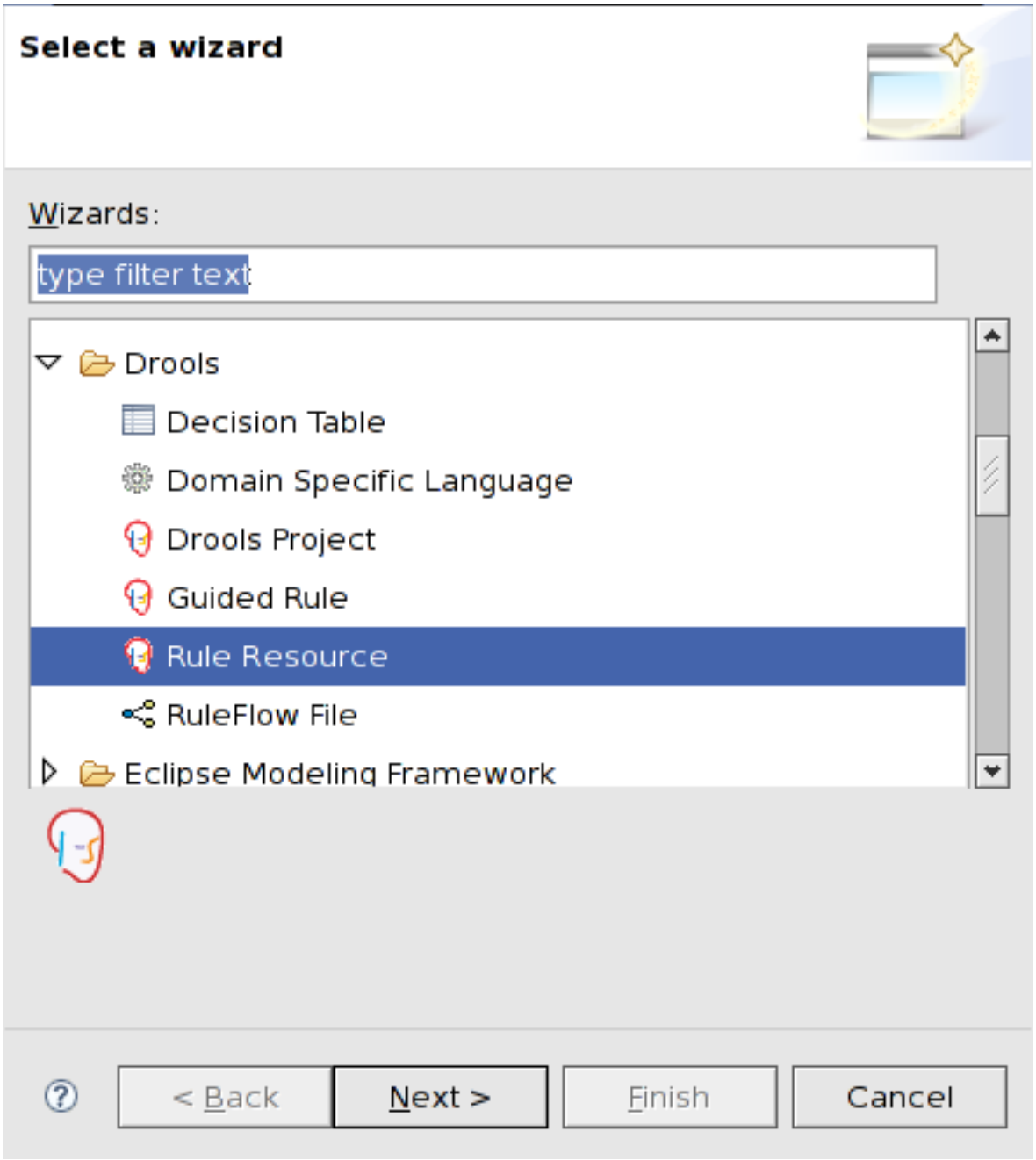


Figure 4.4. The wizard menu

The wizard will ask for some basic options for generating a rule resource. These are just hints, you can change your mind later !. In terms of location, typically you would create a top level /rules directory to store your rules if you are creating a rule project, and store it in a suitably named subdirectory. The package name is mandatory, and is similar to a package name in java (ie. its a namespace that groups like rules together).

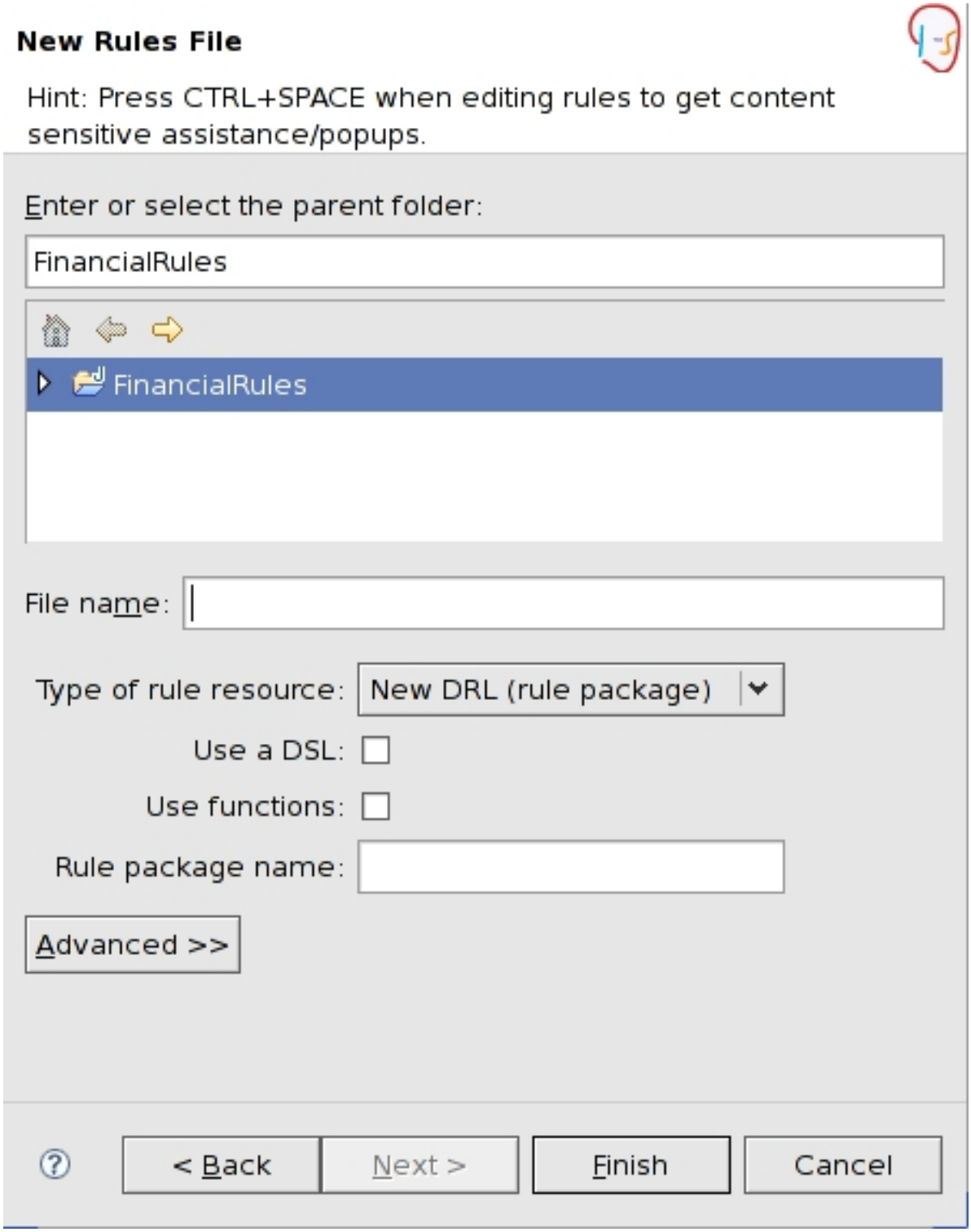


Figure 4.5. New rule wizard

This result of this wizard is to generate a rule skeleton to work from. As with all wizards, they are candy: you don't have to use them if you don't want !

4.1.4. Textual rule editor

The rule editor is where rule managers and developers will be spending most of their time. The rule editor follows the pattern of a normal text editor in eclipse, with all the normal features of a text editor. On top of this, the rule editor provides pop up content assistance. You invoke popup content assistance the "normal" way by pressing Control + Space at the same time.

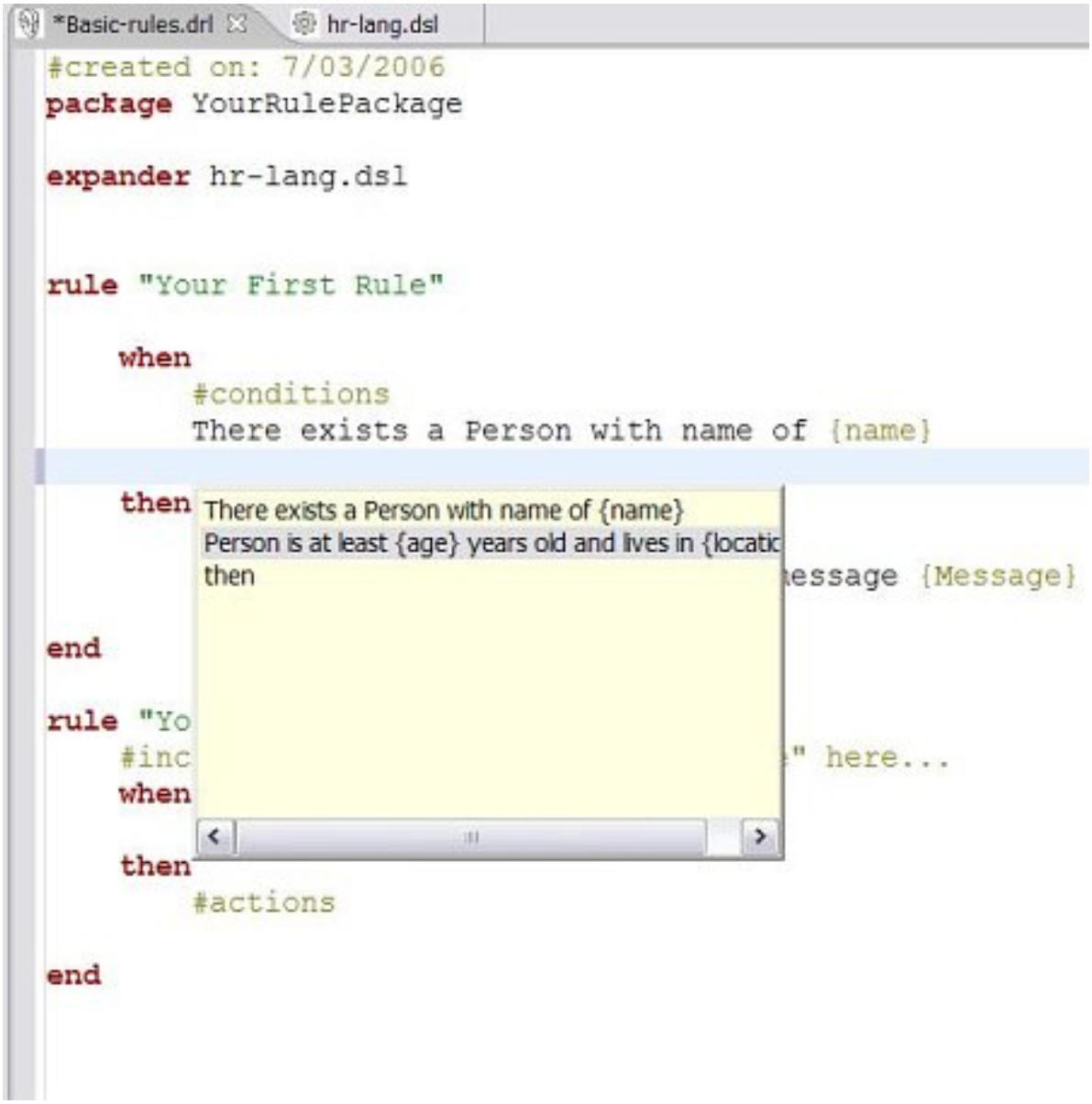


Figure 4.6. The rule editor in action

The rule editor works on files that have a .drl (or .rule) extension. Rules are generally grouped together as a "package" of rules (like the old ruleset construct). It will also be possible to have rules in individual files (grouped by being in the same package "namespace" if you like). These DRL files are plain text files.

You can see from the example above that the package is using a domain specific language (note the expander keyword, which tells the rule compiler to look for a dsl file of that name, to resolve the rule language). Even with the domain specific language (DSL) the rules are still stored as plain text as you see on screen, which allows simpler management of rules and versions (comparing versions of rules for instance).

The editor has an outline view that is kept in sync with the structure of the rules (updated on save). This provides a quick way of navigating around rules by name, in a file which may have hundreds of rules. The items are sorted alphabetically by default.

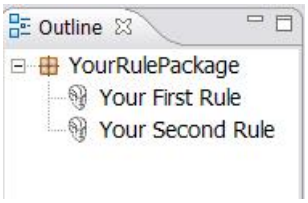


Figure 4.7. The rule outline view

4.1.5. Guided editor (rule GUI)

A new feature of the Drools IDE (since version 4) is the guided editor for rules. This allows you to build rules in a GUI driven fashion, based on your object model.

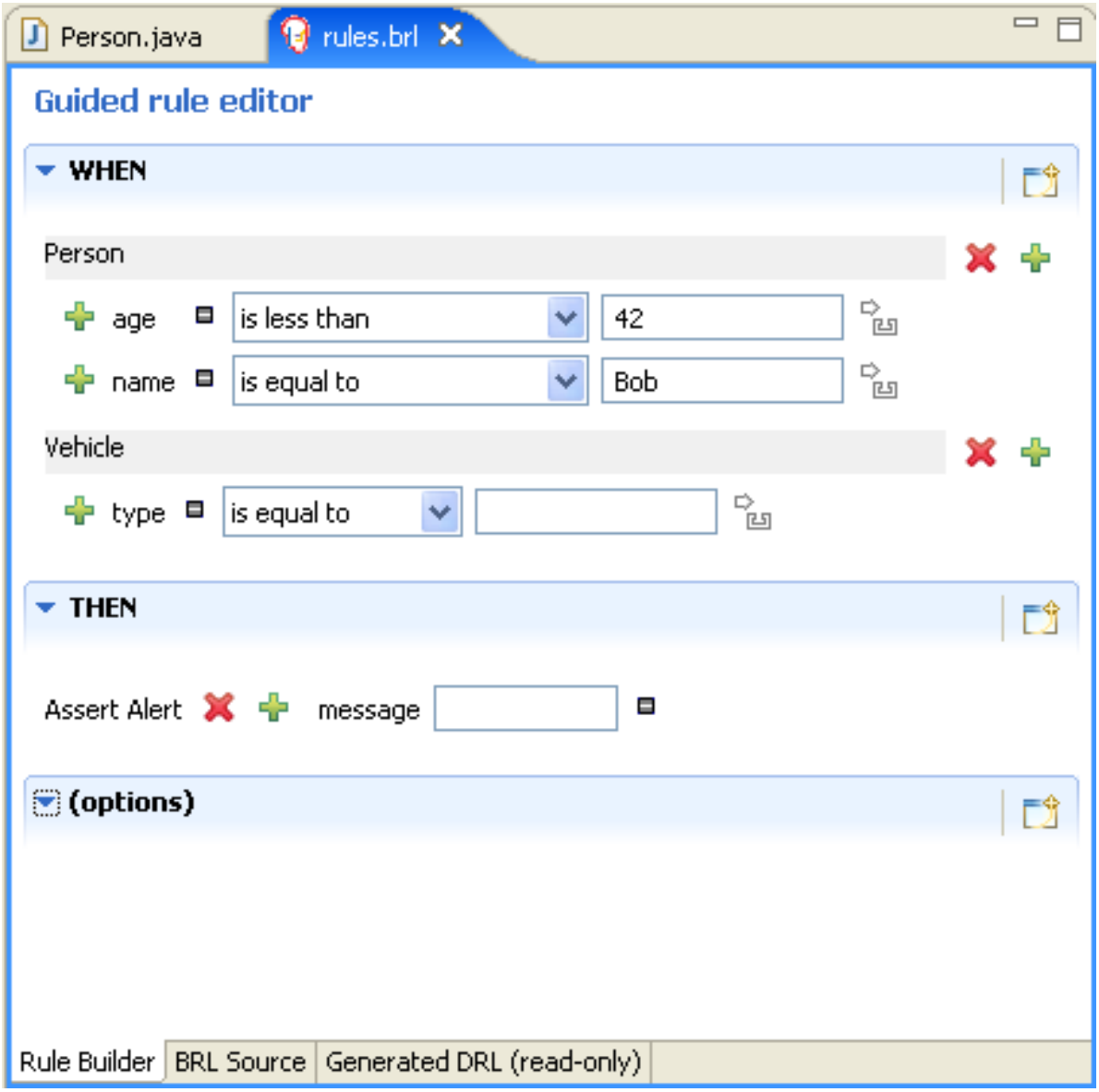


Figure 4.8. The guided editor

To create a rule this way, use the wizard menu. It will create a instance of a .brl file and open an editor. The guided editor works based on a .package file in the same directory as the .brl file. In this "package" file - you have the package name and import statements - just like you would in the top of a normal DRL file. So the first time you create a brl rule - you will need to ppulate the package file with

the fact classes you are interested in. Once you have this the guided editor will be able to prompt you with facts/fields and build rules graphically.

The guided editor works off the model classes (fact classes) that you configure. It then is able to "render" to DRL the rule that you have entered graphically. You can do this visually - and use it as a basis for learning DRL, or you can use it and build rules of the brl directly. To do this, you can either use the drools-ant module (it is an ant task that will build up all the rule assets in a folder as a rule package - so you can then deploy it as a binary file), OR you can use the following snippet of code to convert the brl to a drl rule:

```
BRXMLPersitence read = BRXMLPersitence.getInstance();
BRDRLPersistence write = BRDRLPersistence.getInstance();
String brl = ... read from the .brl file as needed...
String outputDRL = write.marshall(read.unmarshal(brl));
//then pass the outputDRL to the PackageBuilder as normal
```

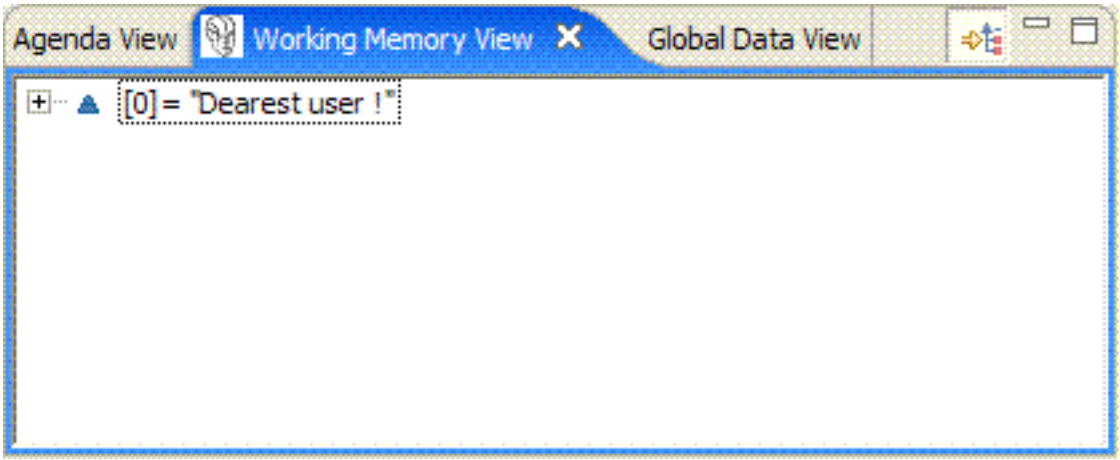
4.1.6. Views

When debugging an application using a Drools engine, these views can be used to check the state of the Drools engine itself: the Working Memory View, the Agenda View the Global Data View. To be able to use these views, create breakpoints in your code invoking the working memory. For example, the line where you call workingMemory.fireAllRules() is a good candidate. If the debugger halts at that joinpoint, you should select the working memory variable in the debug variables view. The following rules can then be used to show the details of the selected working memory:

1. The Working Memory shows all elements in the working memory of the Drools working memory.
2. The Agenda View shows all elements on the agenda. For each rule on the agenda, the rule name and bound variables are shown.
3. The Global Data View shows all global data currently defined in the Drools working memory.

The Audit view can be used to show audit logs that contain events that were logged during the execution of a rules engine in a tree view.

4.1.6.1. The Working Memory View

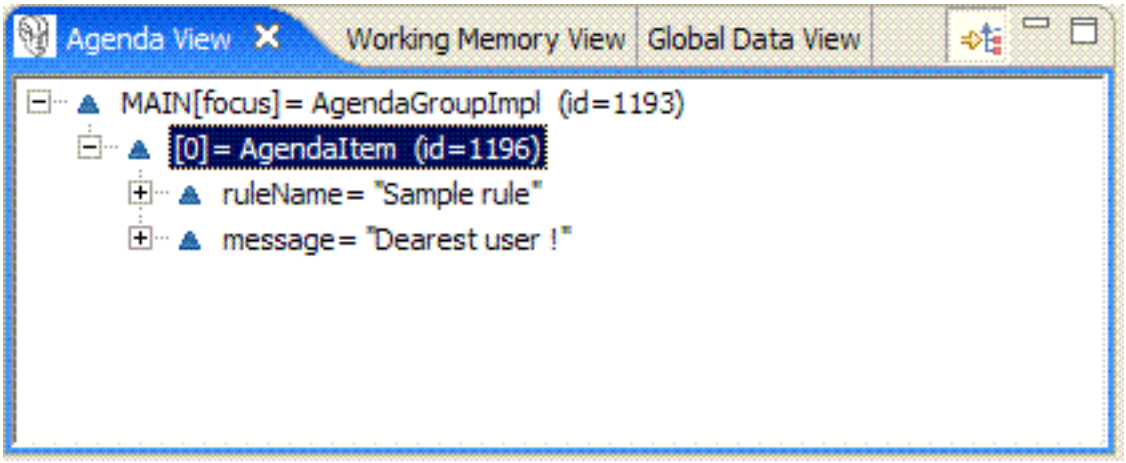


The Working Memory shows all elements in the working memory of the Drools engine.

An action is added to the right of the view, to customize what is shown:

1. The Show Logical Structure toggles showing the logical structure of the elements in the working memory, or all their details. Logical structures allow for example visualizing sets of elements in a more obvious way.

4.1.6.2. The Agenda View

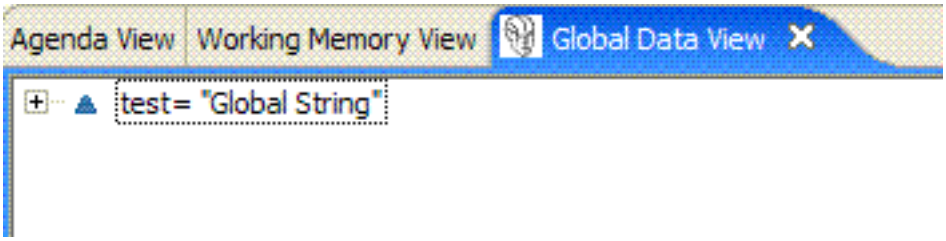


The Agenda View shows all elements on the agenda. For each rule on the agenda, the rule name and bound variables are shown.

An action is added to the right of the view, to customize what is shown:

1. The Show Logical Structure toggles showing the logical structure of the agenda item, or all their details. Logical structures allow for example visualizing sets of elements in a more obvious way. The logical structure of AgendaItems shows the rule that is represented by the AgendaItem, and the values of all the parameters used in the rule.

4.1.6.3. The Global Data View

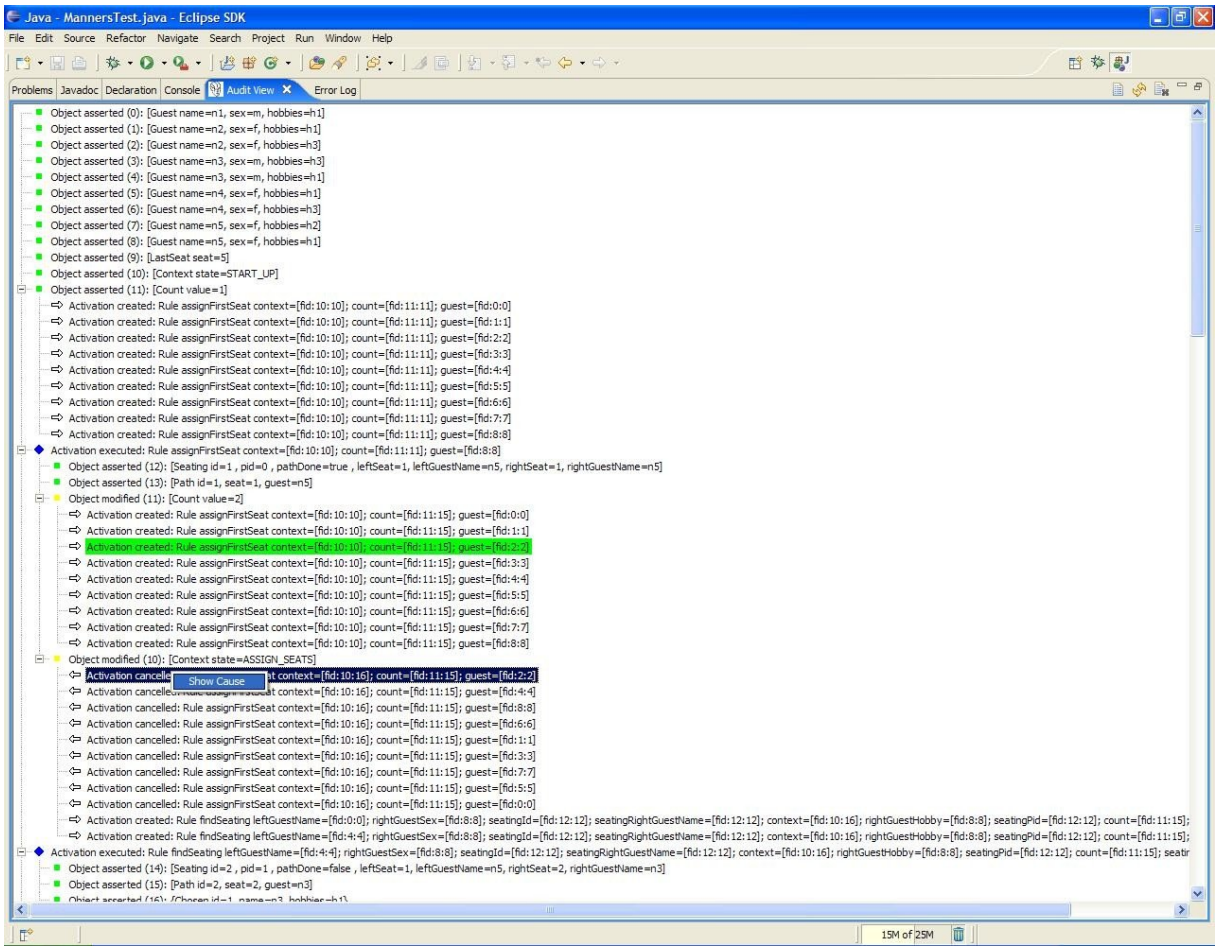


The Global Data View shows all global data currently defined in the Drools engine.

An action is added to the right of the view, to customize what is shown:

1. The Show Logical Structure toggles showing the logical structure of the elements in the working memory, or all their details. Logical structures allow for example visualizing sets of elements in a more obvious way.

4.1.6.4. The Audit View



The audit view can be used to visualize an audit log that can be created when executing the rules engine. To create an audit log, use the following code:

```
    WorkingMemory workingMemory = ruleBase.newWorkingMemory();
    // create a new Working Memory Logger, that logs to file.
    WorkingMemoryFileLogger logger = new WorkingMemoryFileLogger(workingMemory);
    // an event.log file is created in the log dir (which must exist)
    // in the working directory
    logger.setFileName("log/event");

    workingMemory.assertObject( ... );
    workingMemory.fireAllRules();

    // stop logging
    logger.writeToDisk();
```

Open the log by clicking the Open Log action (first action in the Audit View) and select the file. The Audit view now shows all events that where logged during the executing of the rules. There are different types of events (each with a different icon):

1. Object inserted (green square)
2. Object updated (yellow square)
3. Object removed (red square)
4. Activation created (arrow to the right)

5. Activation cancelled (arrow to the left)
6. Activation executed (blue diamond)
7. Ruleflow started / ended (process icon)
8. Ruleflow-group activated / deactivated (process icon)
9. Rule package added / removed (Drools icon)
10. Rule added / removed (Drools icon)

All these events show extra information concerning the event, like the id and toString representation of the object in case of working memory events (assert, modify and retract), the name of the rule and all the variables bound in the activation in case of an activation event (created, cancelled or executed). If an event occurs when executing an activation, it is shown as a child of the activation executed event. For some events, you can retrieve the "cause":

1. The cause of an object modified or retracted event is the last object event for that object. This is either the object asserted event, or the last object modified event for that object.
2. The cause of an activation cancelled or executed event is the corresponding activation created event.

When selecting an event, the cause of that event is shown in green in the audit view (if visible of course). You can also right click the action and select the "Show Cause" menu item. This will scroll you to the cause of the selected event.

4.1.7. Domain Specific Languages

Domain Specific Languages (dsl) allow you to create a language that allows your rules to look like, rules ! Most often the domain specific language reads like natural language. Typically you would look at how a business analyst would describe the rule, in their own words, and then map this to your object model via rule constructs. A side benefit of this is that it can provide an insulation layer between your domain objects, and the rules themselves (as we know you like to refactor !). A domain specific language will grow as the rules grow, and works best when there are common terms used over an over, with different parameters.

To aid with this, the rule workbench provides an editor for domain specific languages (they are stored in a plain text format, so you can use any editor of your choice - it uses a slightly enhanced version of the "Properties" file format, simply). The editor will be invoked on any files with a .dsl extension (there is also a wizard to create a sample DSL).

4.1.7.1. Editing languages

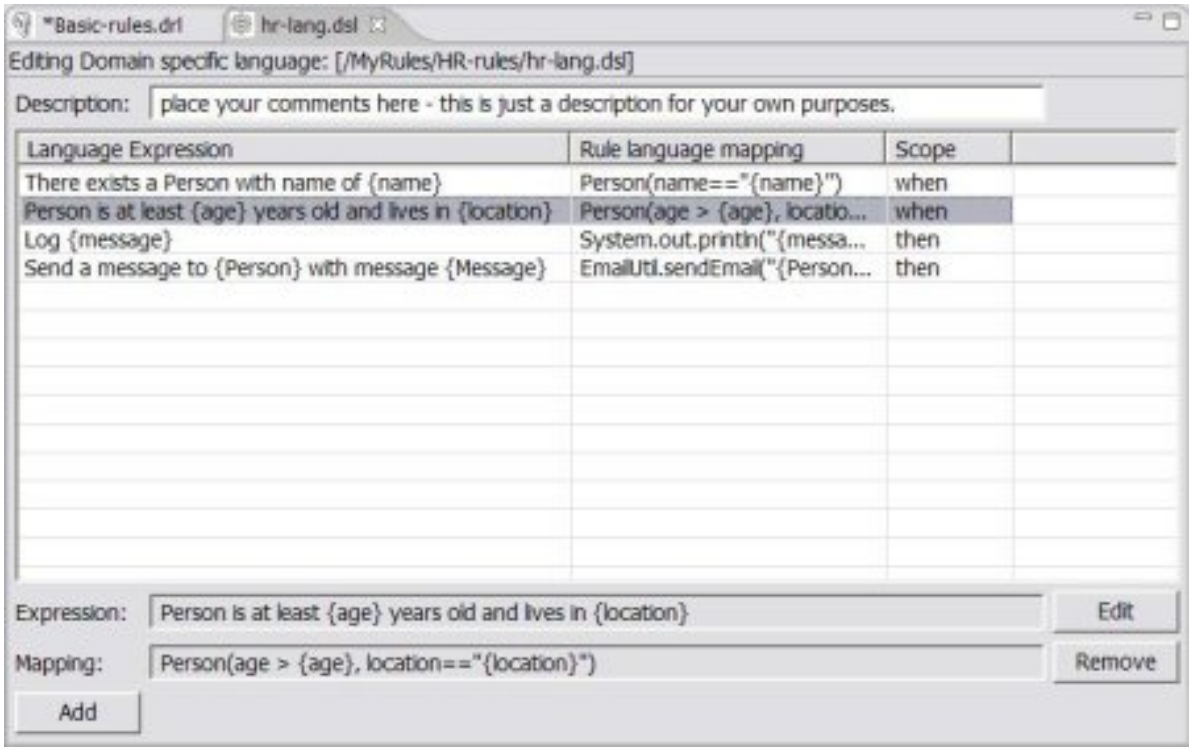


Figure 4.9. The Domain Specific Language editor

The DSL editor provides a table view of Language Expression to Rule Expression mapping. The Language expression is what is used in the rules. This also feeds the content assistance for the rule editor, so that it can suggest Language Expressions from the DSL configuration (the rule editor loads up the DSL configuration when the rule resource is loaded for editing). The Rule language mapping is the "code" for the rules - which the language expression will be compiled to by the rule engine compiler. For form of this Rule language depends if it is for a condition or action part of a rule (it may be a snippet of java, for instance). The "scope" item indicates where the expression is targeted: is it for the "when" part of the rule (LHS)? the "then" part (RHS)? Or anywhere?

By selecting a mapping item (a row in the table) you can see the expression and mapping in the greyed out fields below. Double clicking or pressing the edit button will open the edit dialog. You can remove items, and add new ones (you should generally only remove when you know that expression is no longer in use).

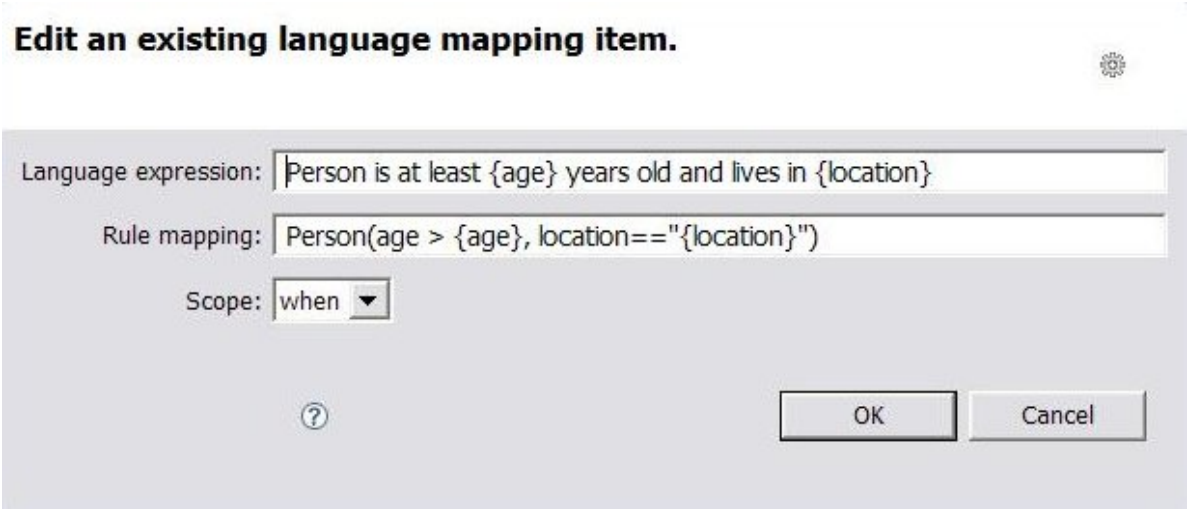


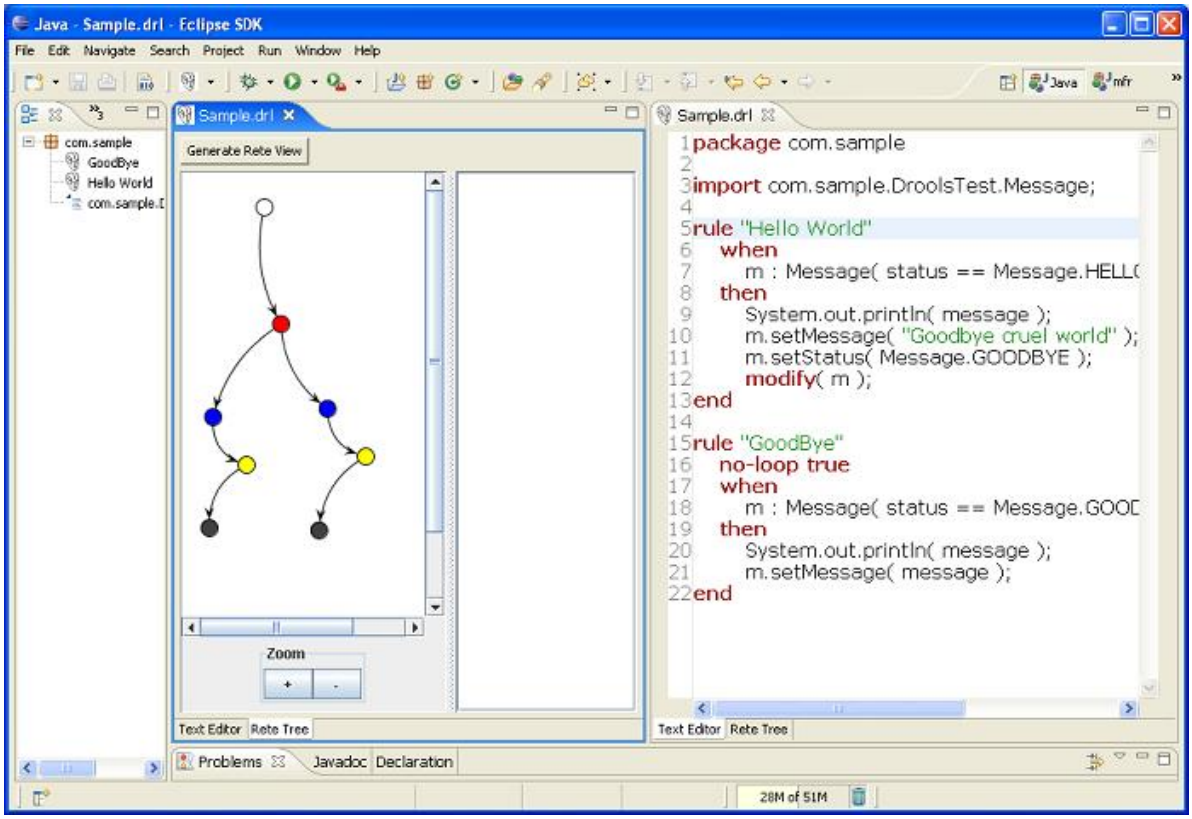
Figure 4.10. Language Mapping editor dialog

How it works: the "Language expression" is used to parse the rule language, depending on what the "scope" is set to. When it is found in a rule, the values that are market by the curly braces {value} are extracted from the rule source. These values are then interpolated with the "Rule mapping" expression, based on the names between the curly braces. So in the example above, the natural language expression maps to 2 constraints on a fact of type Person (ie the person object has the age field as less than {age}, and the location value is the string of {value}, where {age} and {value} are pulled out of the original rule source. The Rule mapping may be a java expression (such as if the scope was "then"). If you did not wish to use a language mapping for a particular rule in a drl, prefix the expression with > and the compiler will not try to translate it according to the language definition. Also note that domain specific languages are optional. When the rule is compiled, the .dsl file will also need to be available.

4.1.8. The Rete View

The Rete Tree View shows you the current Rete Network for your drl file. Just click on the tab "Rete Tree" below on the DRL Editor. Afterwards you can generate the current Rete Network visualization. You can push and pull the nodes to arrange your optimal network overview. If you got hundreds of nodes, select some of them with a frame. Then you can pull groups of them. You can zoom in and out, in case not all nodes are shown in the current view. For this press the button "+" oder "-".

There is no export function, which creates a gif or jpeg picture, in the current release. Please use ctrl + alt + print to create a copy of your current eclipse window and cut it off.



The graph is created with the Java Universal Network/Graph Framework (*JUNG*[1]). The Rete View is an advanced feature which is still in experimental state. It uses Swing inside eclipse. In future it will maybe improved using SWT or GEF.

The Rete view works only in Drools Rule Projects, where the Drools Builder is set in the project´s properties.

If you are using Drools in an other type of project, where you are not having a Drools Rule Project with the appropriate Drools Builder, you can create a little workaround:

Set up a little Drools Rule Project next to it, putting needed libraries into it and the drls you want to inspect with the Rete View. Just click on the right tab below in the DRL Editor, followed by a click on "Generate Rete View".

4.1.9. Large drl files

Depending on the JDK you use, it may be necessary to increase the permanent generation max size. Both SUN and IBM jdk have a permanent generation, whereas BEA JRockit does not.

To increase the permanent generation, start eclipse with -XX:MaxPermSize=###m

Example: c:\eclipse\eclipse.exe -XX:MaxPermSize=128m

Rulesets of 4,000 rules or greater should set the permanent generation to atleast 128Mb.

(note that this may also apply to compiling large numbers of rules in general - as there is generally one or more classes per rule).

[1] http://jung.sourceforge.net/

As an alternative to the above, you may put rules in a file with the ".rule" extension, and the background builder will not try to compile them with each change, which may provide performance improvements if your IDE becomes sluggish with very large numbers of rules.

4.1.10. Debugging rules

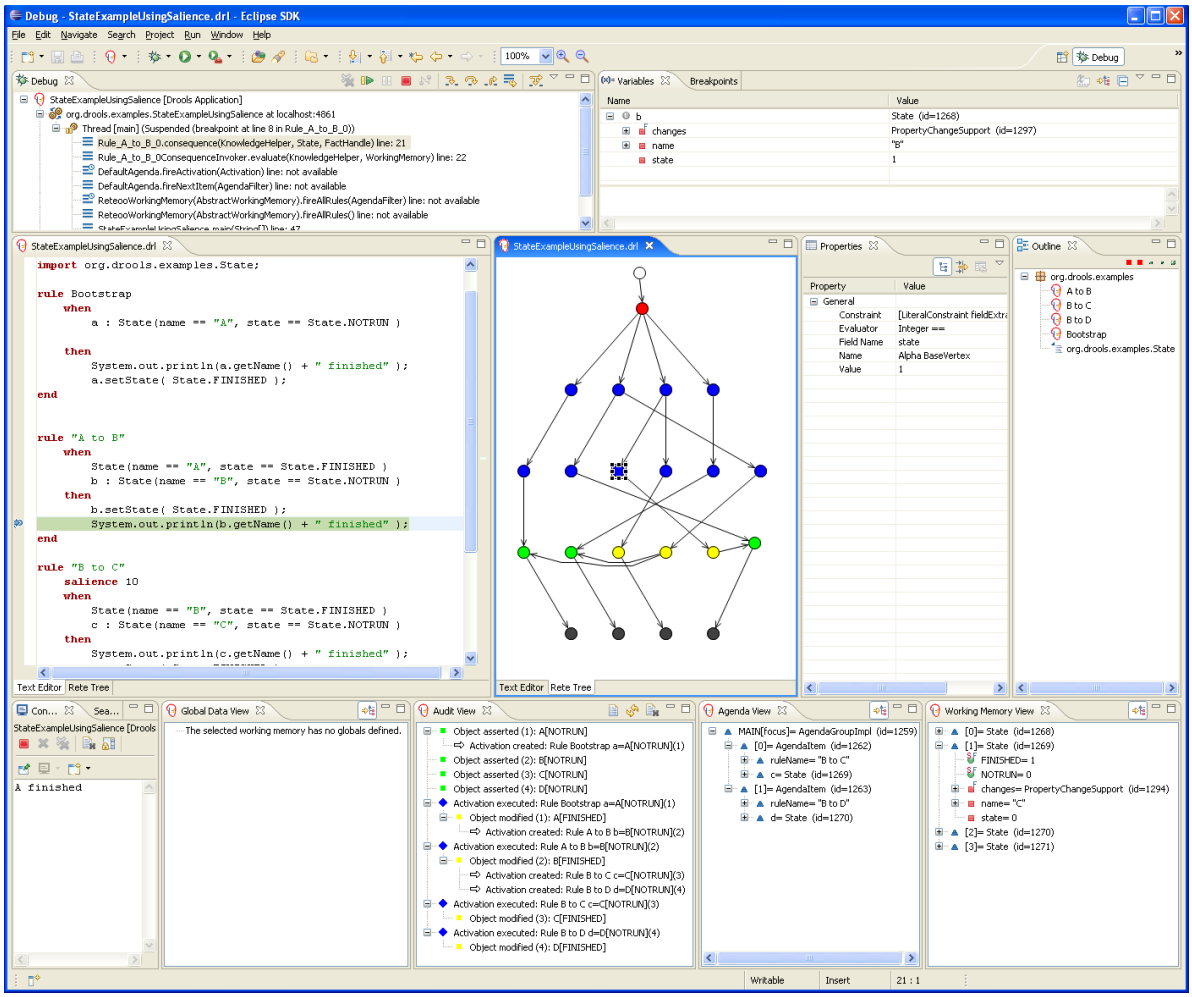


Figure 4.11. Debugging

You can debug rules during the execution of your Drools application. You can add breakpoints in the consequences of your rules, and whenever such a breakpoint is uncounted during the execution of the rules, the execution is halted. You can then inspect the variables known at that point and use any of the default debugging actions to decide what should happen next (step over, continue, etc.). You can also use the debug views to inspect the content of the working memory and agenda.

4.1.10.1. Creating breakpoints

You can add/remove rule breakpoints in drl files in two ways, similar to adding breakpoints to Java files:

1. Double-click the ruler of the DRL editor at the line where you want to add a breakpoint. Note that rule breakpoints can only be created in the consequence of a rule. Double-clicking on a line where no breakpoint is allowed will do nothing. A breakpoint can be removed by double-clicking the ruler once more.

2. If you right-click the ruler, a popup menu will show up, containing the "Toggle breakpoint" action. Note that rule breakpoints can only be created in the consequence of a rule. The action is

automatically disabled if no rule breakpoint is allowed at that line. Clicking the action will add a breakpoint at the selected line, or remove it if there was one already.

The Debug Perspective contains a Breakpoints view which can be used to see all defined breakpoints, get their properties, enable/disable or remove them, etc.

4.1.10.2. Debugging rules

Drools breakpoints are only enabled if you debug your application as a Drools Application. You can do this like this:

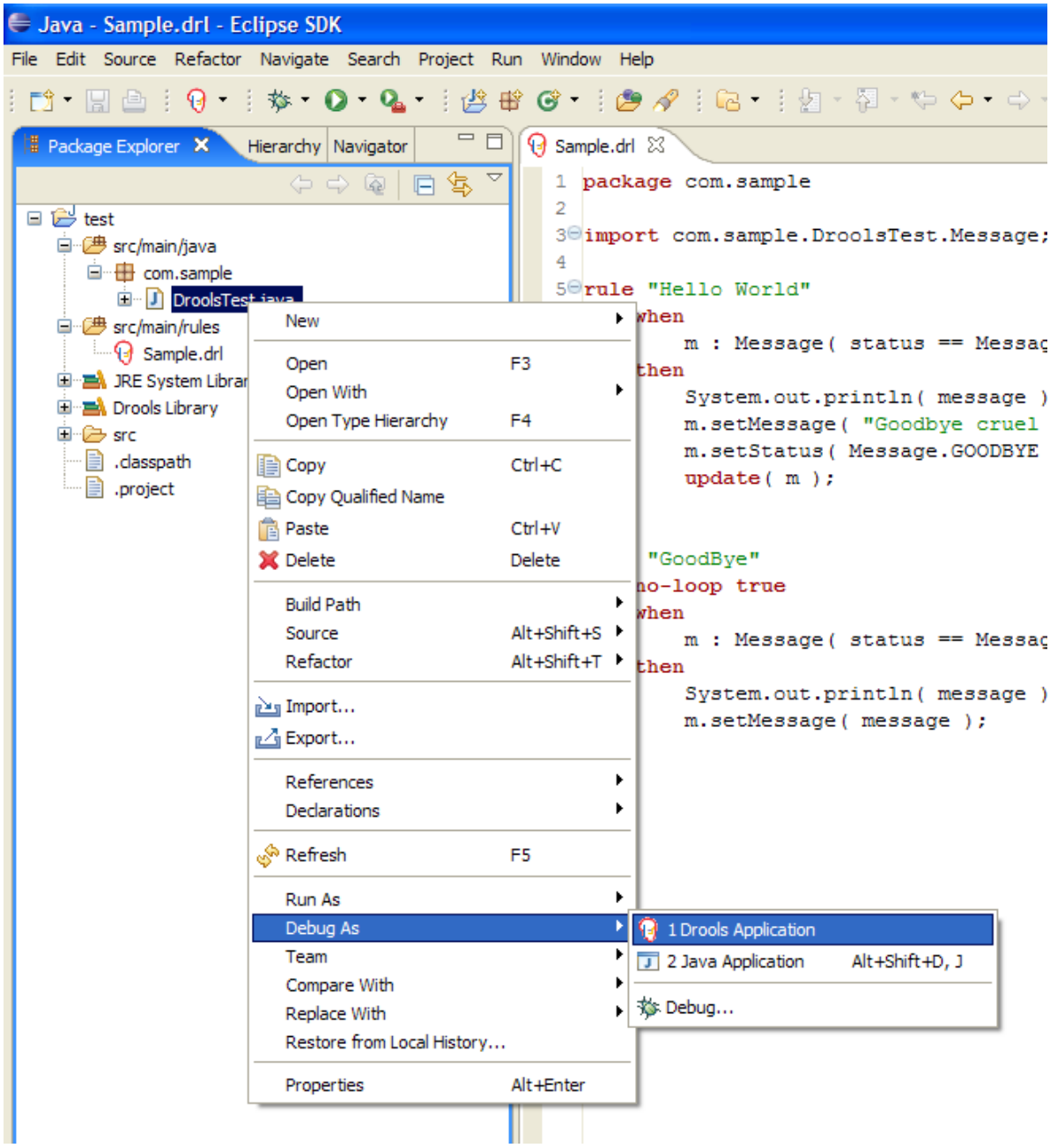


Figure 4.12. Debug as Drools Application

1. Select the main class of your application. Right click it and select the "Debug As >" sub-menu and select Drools Application. Alternatively, you can also select the "Debug ..." menu item to open a new dialog for creating, managing and running debug configurations (see screenshot below)

a. Select the "Drools Application" item in the left tree and click the "New launch configuration" button (leftmost icon in the toolbar above the tree). This will create a new configuration and already fill in some of the properties (like the project and main class) based on main class you selected in the beginning. All properties shown here are the same as any standard Java program.

b. Change the name of your debug configuration to something meaningful. You can just accept the defaults for all other properties. For more information about these properties, please check the eclipse jdt documentation.

c. Click the "Debug" button on the bottom to start debugging your application. You only have to define your debug configuration once. The next time you try to run your Drools application, you don't have to create a new one but select the one you defined previously by selecting it in the tree on the left, as a sub-element of the "Drools Application" tree node, and then click the Debug button. The eclipse toolbar also contains shortcut buttons to quickly re-execute the one of your previous configurations (at least when the Java, Java Debug, or Drools perspective has been selected).

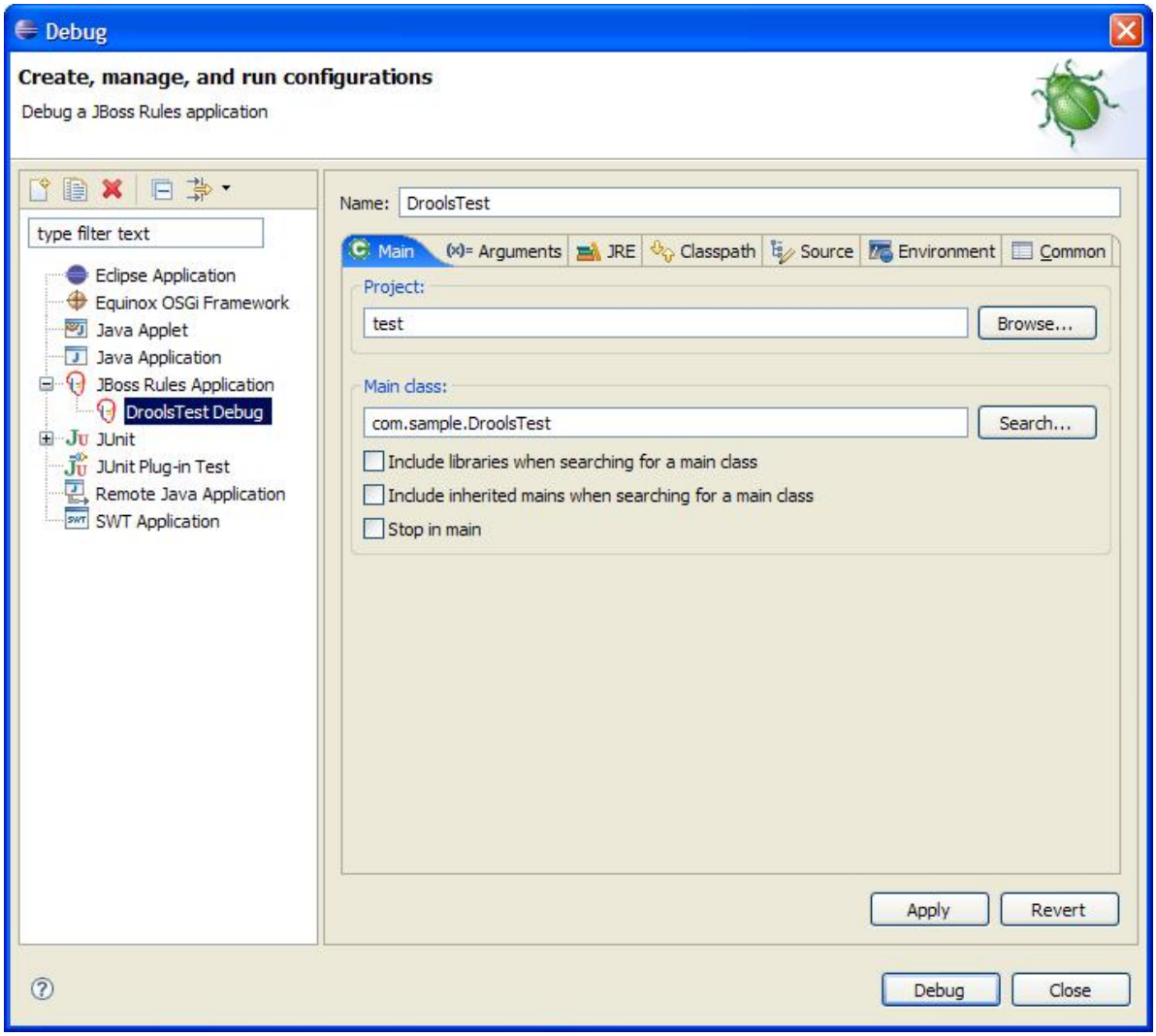


Figure 4.13. Debug as Drools Application Configuration

After clicking the "Debug" button, the application starts executing and will halt if any breakpoint is encountered. This can be a Drools rule breakpoint, or any other standard Java breakpoint. Whenever a Drools rule breakpoint is encountered, the corresponding drl file is opened and the active line is

highlighted. The Variables view also contains all rule parameters and their value. You can then use the default Java debug actions to decide what to do next (resume, terminate, step over, etc.). The debug views can also be used to determine the contents of the working memory and agenda at that time as well (you don't have to select a working memory now, the current executing working memory is automatically shown).

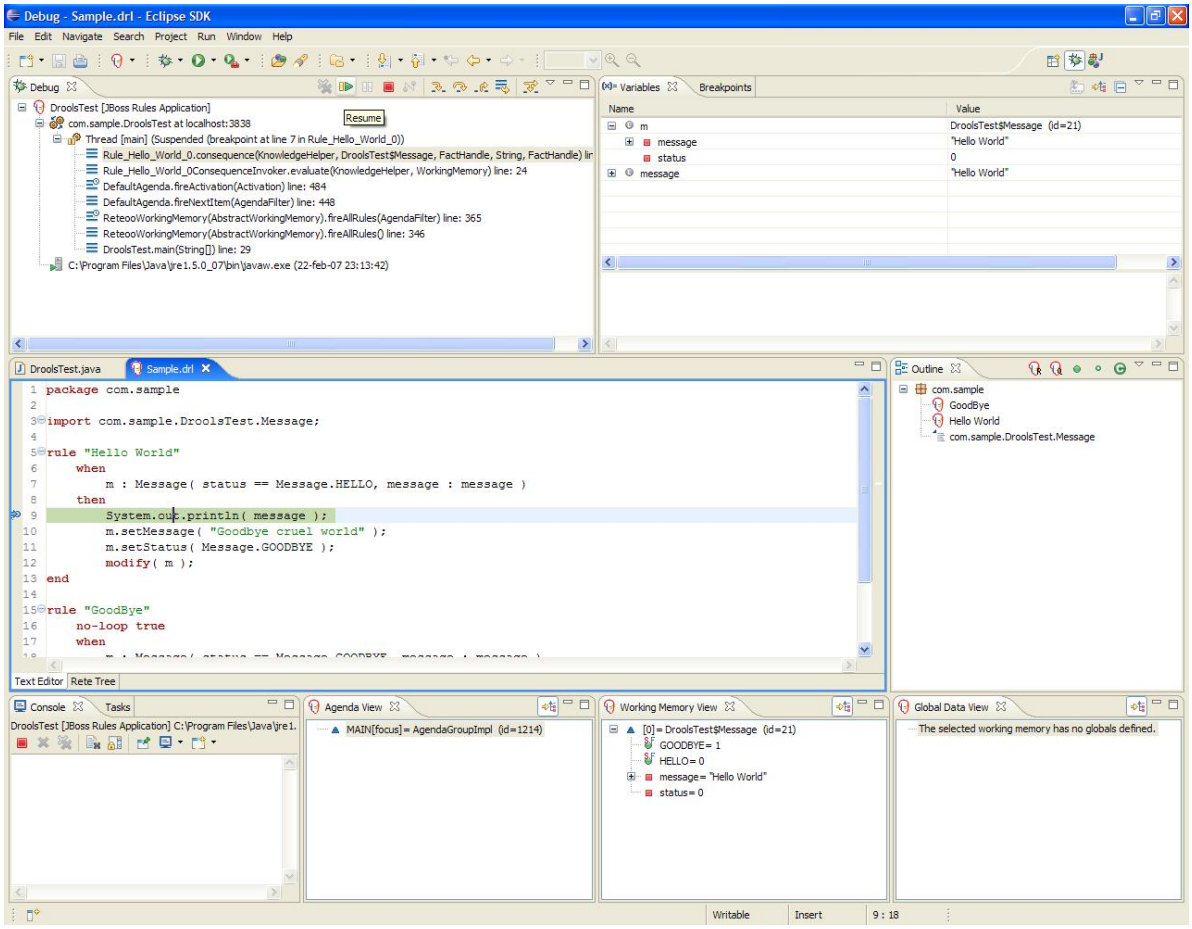


Figure 4.14. Debugging

The Rule Language

5.1. Overview

Drools 4.0 has a "native" rule language that is non XML textual format. This format is very light in terms of punctuation, and supports natural and domain specific languages via "expanders" that allow the language to morph to your problem domain. This chapter is mostly concerted with the native rule format. The Diagrams used are known as "rail road" diagrams, and are basically flow charts for the language terms. For the technically very keen, you can also refer to "DRL.g" which is the Antlr3 grammar for the rule language. If you use the Rule Workbench, a lot of the rule structure is done for you with content assistance, for example, type "ru" and press ctrl+space, and it will build the rule structure for you.

5.1.1. A rule file

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

The overall structure of a rule file is:

Example 5.1. Rules file

```
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.

5.1.2. What makes a rule

For the inpatients, just as an early view, a rule has the following rough structure:

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

Its really that simple. Mostly punctuation is not needed, even the double quotes for "name" are optional, as are newlines. Attributes are simple (always optional) hints to how the rule should behave.

LHS is the conditional parts of the rule, which follows a certain syntax which is covered below. RHS is basically a block that allows dialect specific semantic code to be executed.

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

5.1.3. Reserved words

There are some reserved keywords that are used in the rule language. It is wise to avoid collisions with these words when naming your domain objects, properties, methods, functions and other elements that are used in the rule text. The parser is a bit smart and sometimes knows when a keyword is not being used as a keyword, but avoiding the use of them might prevent some headaches.

The following is a list of keywords that must be avoided as identifiers when writing rules:

rule	null	collect	false
query	and	accumulate	eval
when	or	from	
then	not	forall	
end	exists	true	

The following list are keywords that you should try and avoid in the rule contents if possible, but the parser usually will work fine, even if you use them for something else.

package	salience	excludes	auto-focus
function	duration	memberOf	activation-group
global	init	matches	agenda-group
import	action	in	dialect
template	reverse	date-effective	rule-flow-group
attributes	result	date-expires	
enabled	contains	no-loop	

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

5.2. Comments

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

5.2.1. Single line comment

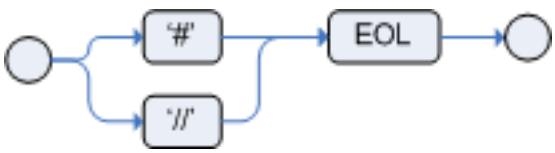


Figure 5.1. Single line comment

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

```
rule "Testing Comments"
```

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

5.2.2. Multi line comment

Figure 5.2. Multi line comment

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

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

5.3. Package

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

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

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

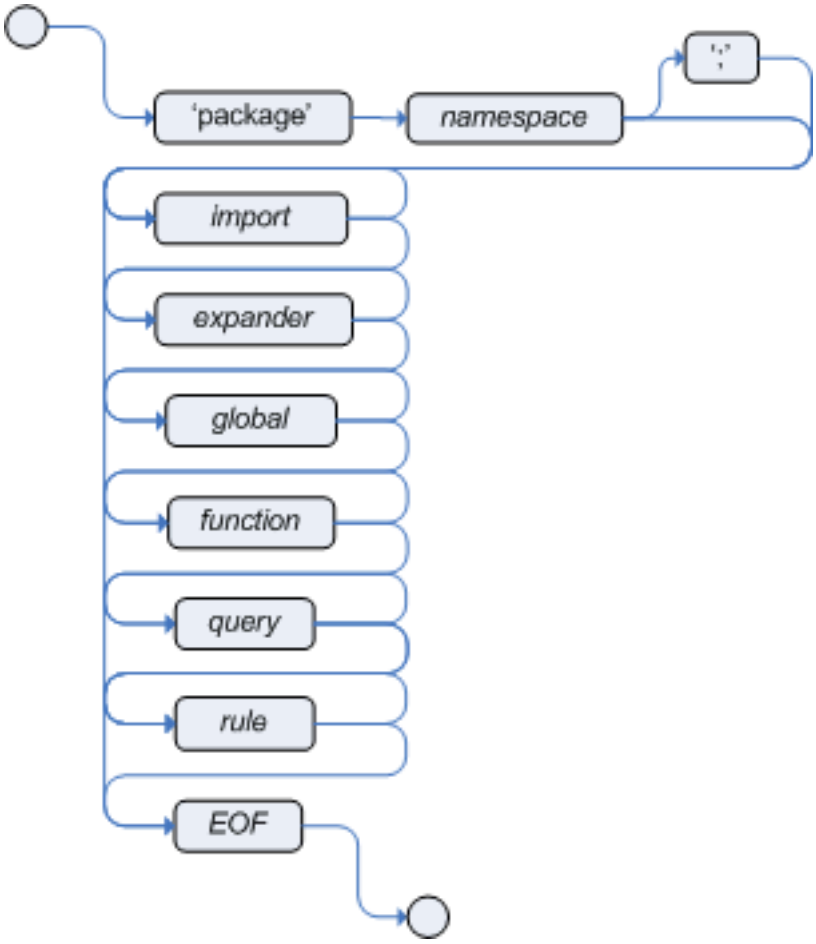


Figure 5.3. package

5.3.1. import

Figure 5.4. 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. Drools automatically imports classes from the same named java package and from the java.lang package.

5.3.2. expander

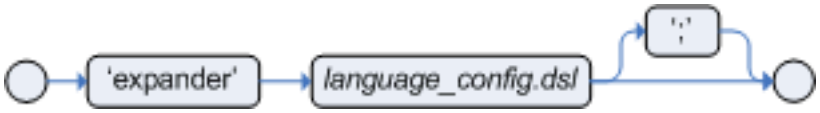


Figure 5.5. expander

The expander statement (optional) is used to specify domain specific language (DSL) configurations (which are normally stored in a separate file). This provides clues to the parser as how to understand what you are raving on about in your rules. It is important to note that in Drools 4.0 (that is different from Drools 3.x) the expander declaration is mandatory for the tools to provide you context assist and

avoiding error reporting, but the API allows the program to apply DSL templates, even if the expanders are not declared in the source file.

5.3.3. global

Figure 5.6. global

Globals are global variables. They are used to make application objects available to the rules, and are typically used to provide data or services that the rules use (specially application services used in rule consequences), to return data from the rules (like logs or values added in rules consequence) or for the rules to interact with the application doing callbacks. Globals are not inserted into the Working Memory so they should never be reasoned over, and only use them in rules LHS if the global has a constant immutable value. The engine is not notified and does not track globals value changes. Incorrect use of globals in constraints may yield surprising results - surprising in a bad way, like when a doctor says "thats interesting" to a chest XRay of yours.

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

In order to use globals you must:

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

```
global java.util.List myGlobalList;

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

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

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

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

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

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

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

5.4. Function

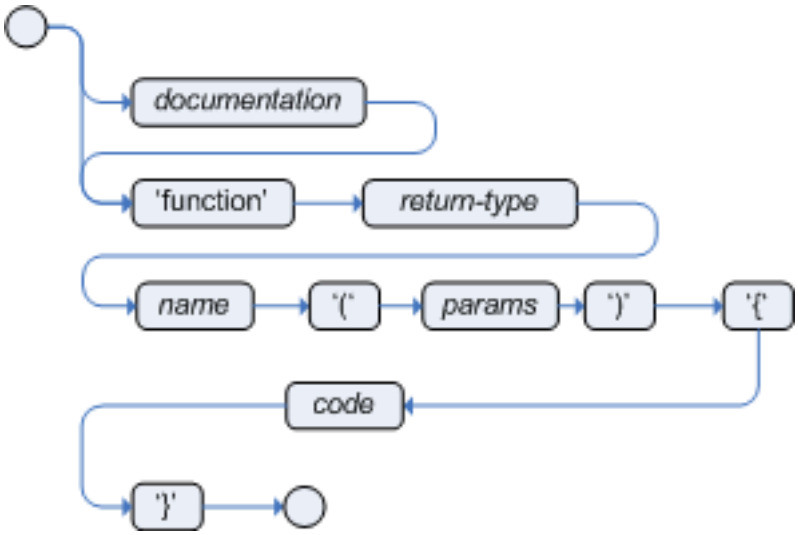


Figure 5.7. function

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

A typical function declaration looks like:

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

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

An alternative to the use of a function, could be to use a static method in a helper class: Foo.hello(). Drools 4.0 supports the use of function imports, so all you would need to do is:

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

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

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

```
end
```

5.5. Rule

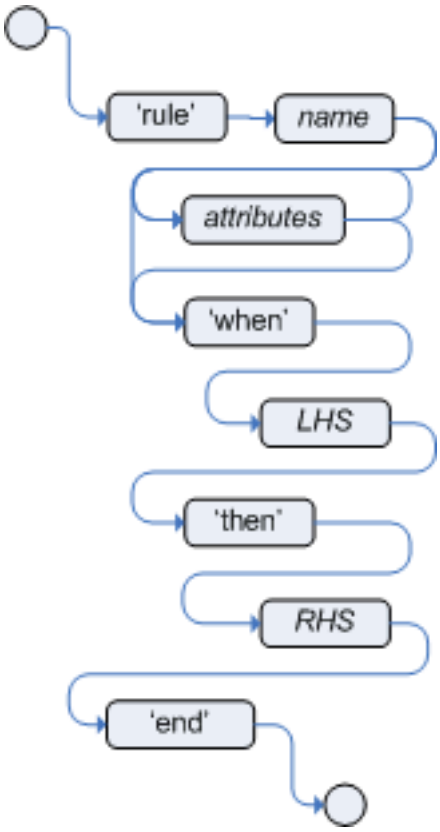


Figure 5.8. rule

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

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

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

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

Example 5.2. Rule Syntax Overview Example

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

```
    <action>*
end
```

Example 5.3. A rule example

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

5.5.1. Rule Attributes

Rule attributes provide a declarative way to influence the behavior of the rule, some are quite simple, while others are part of complex sub system. To get the most from Drools you should make sure you have a proper understanding of each attribute.

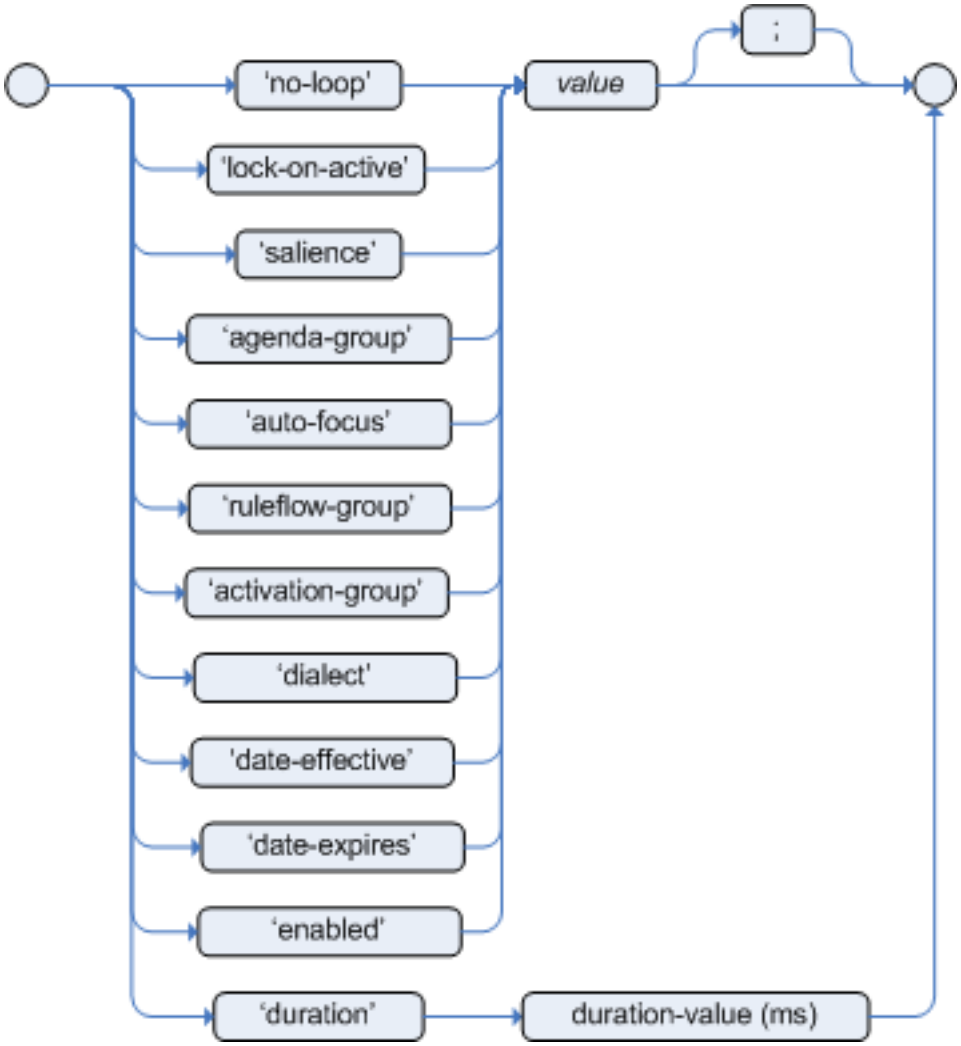


Figure 5.9. rule attributes

5.5.1.1. no-loop

default value : false

type : Boolean

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

5.5.1.2. lock-on-active

default value : false

type : Boolean

Rules with the lock-on-active attribute set to **`true`** can only place an activation onto the agenda once while their agenda-group is focused. While the agenda-group is still focused these rules will not be able to fire again.

lock-on-active also applies to RuleFlow-groups in addition to agenda-groups.

5.5.1.3. salience

default value : 0

type : integer

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

5.5.1.4. agenda-group

default value : MAIN

type : String

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

5.5.1.5. auto-focus

default value false

type : Boolean

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

5.5.1.6. activation-group

default value : N/A

type : String

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

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

5.5.1.7. dialect

default value : as specified by the package

type : String

possible values: "java" or "mvel"

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

5.5.1.8. date-effective

default value : N/A

type : String, which contains a Date/Time definition

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

5.5.1.9. date-exptires

default value : N/A

type : String, which contains a Date/Time definition

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

5.5.1.10. duration

default value : no default value

type : long

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

Example 5.4. Some attribute examples

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

5.5.2. Left Hand Side (when) Conditional Elements

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

Figure 5.10. Left Hand Side

Example 5.5. Rule Syntax Overview Example

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

Is internally re-written as:

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

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

5.5.2.1. Pattern

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

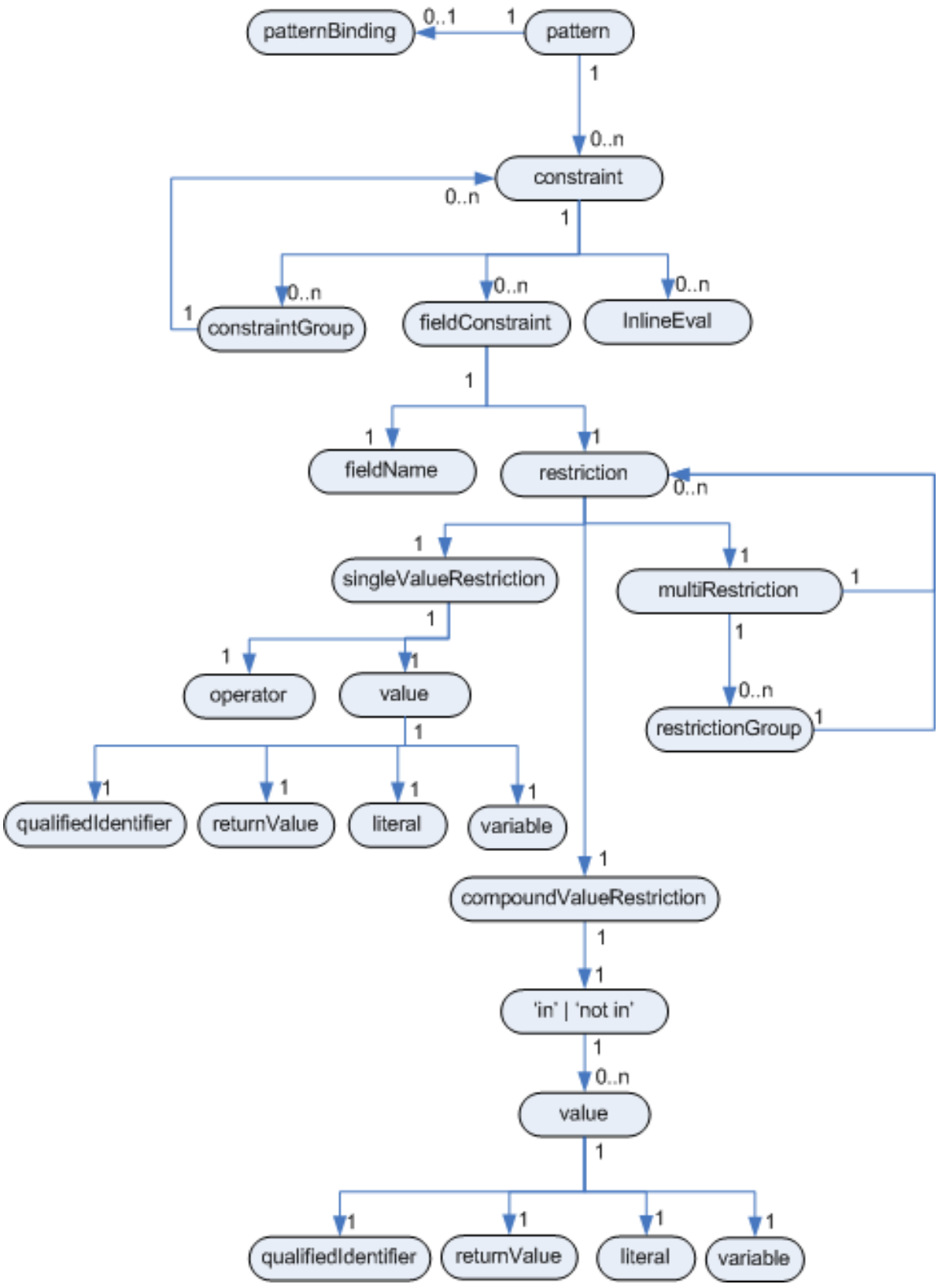


Figure 5.11. Pattern Entity Relationship Diagram

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

Figure 5.12. Pattern

At the simplest, with no constraints, it simply matches against a type, in the following case the type is "Cheese". This means the pattern will match against all Cheese objects in the Working Memory.

Example 5.6. Pattern

```
Cheese( )
```

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

Example 5.7. Pattern

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

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

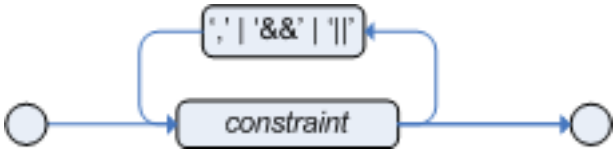


Figure 5.13. Constraints

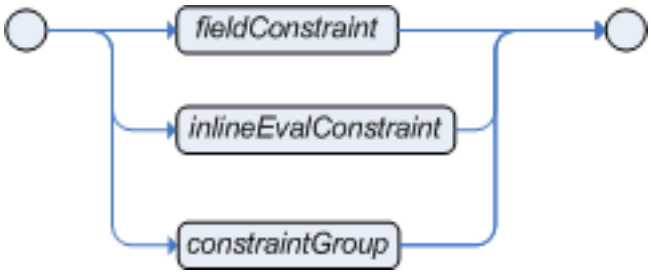


Figure 5.14. Constraint

Figure 5.15. Group Constraint

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

Example 5.8. Constraint Group connective ','

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

The above example has 3 constraint groups, each with a single constraint:

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

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

Example 5.9. && and || Constraint Connectives

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

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

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

1. &&
2. ||
3. ,

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

Example 5.10. Using parenthesis to change evaluation priority

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

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

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

Example 5.11. Not Equivalent connectives

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

5.5.2.1.1. Field Constraints

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

Figure 5.16. fieldConstraint

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

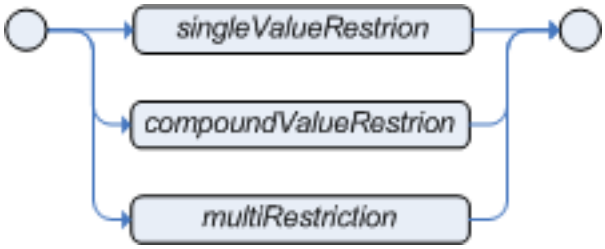


Figure 5.17. restriction

5.5.2.1.1.1. JavaBeans as facts

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

For example, referring to our Cheese class, the following : Cheese(type == ...) uses the getType() method on the a cheese instance. If a field name cannot be found it will resort to calling the name as a no argument method; "toString()" on the Object for instance can be used with Cheese(toString == ..) - you use the full name of the method with correct capitalization, but not brackets. Do please make sure that you are accessing methods that take no parameters, and are in-fact "accessors" (as in, they don't change the state of the object in a way that may effect the rules - remember that the rule engine effectively caches the results of its matching in between invocations to make it faster).

5.5.2.1.1.2. Values

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

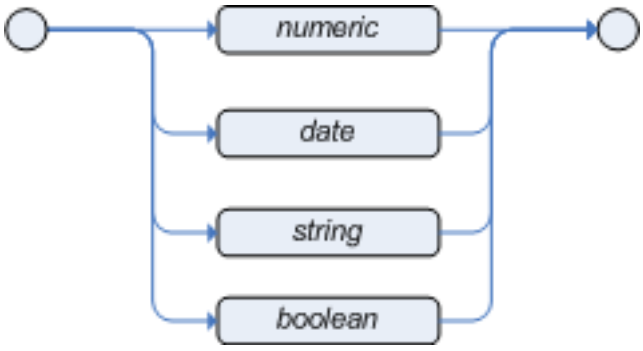


Figure 5.18. literal

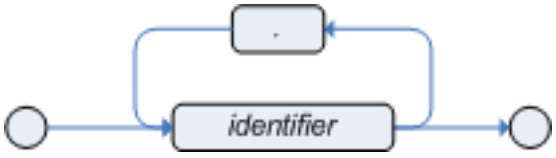


Figure 5.19. qualifiedIdentifier

Figure 5.20. variable

Figure 5.21. returnValue

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

5.5.2.1.1.3. Single Value Restriction

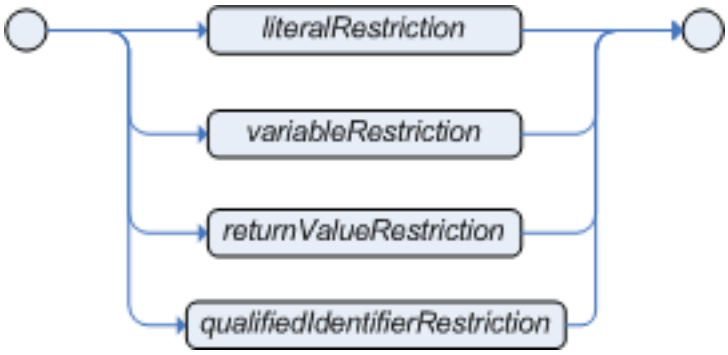


Figure 5.22. singleValueRestriction

5.5.2.1.1.3.1. Operators

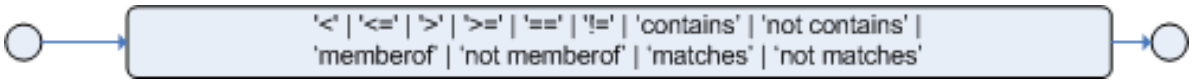


Figure 5.23. Operators

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

Matches Operator

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

> Example 5.12. Regular Expression Constraint
>
> ```
> Cheese( type matches "(Buffalo)?\S*Mozerella" )
> ```

Not Matches Operator

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

Example 5.13. Regular Expression Constraint

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

Contains Operator

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

Example 5.14. Contains with Collections

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

not contains

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

Example 5.15. Literal Constraints with Collections

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

> **NOTE:** for backward compatibility, the '**excludes**' operator is supported as a synonym for '**not contains**'.

memberOf

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

Example 5.16. Literal Constraints with Collections

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

not memberOf

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

Example 5.17. Literal Constraints with Collections

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

soundslike

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

Example 5.18. Text with soundslike (Sounds Like)

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

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

5.5.2.1.1.3.2. Literal Restrictions

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

Figure 5.24. literalRestriction

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

Numeric

All standard java numeric primitives are supported.

Example 5.19. Numeric Literal Restriction

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

Date

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

Example 5.20. Date Literal Restriction

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

String

Any valid Java String is allowed.

Example 5.21. String Literal Restriction

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

Boolean

only true or false can be used. 0 and 1 are not recognized, nor is **`Cheese ( smelly )`** is allowed

Example 5.22. Boolean Literal Restriction

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

Qualified Identifier

Enums can be used as well, both jdk1.4 and jdk5 style enums are supported - for the later you must be executing on a jdk5 environment.

Example 5.23. Boolean Literal Restriction

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

5.5.2.1.1.3.3. Bound Variable Restriction

Figure 5.25. variableRestriction

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

Example 5.24. Bound Field using '==' operator

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

'likes' is our variable, our Declaration, that is bound to the favouriteCheese field for any matching Person instance and is used to constrain the type of Cheese in the following Pattern. Any valid java variable name can be used, including '$'; which you will often see used to help differentiate declarations from fields. The example below shows a declaration bound to the Patterns Object Type instance itself and used with a 'contains' operator, note the optional use of '$' this time.

Example 5.25. Bound Fact using 'contains' operator

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

5.5.2.1.1.3.4. Return Value Restriction

Figure 5.26. returnValueRestriction

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

Example 5.26. Return Value Restriction

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

5.5.2.1.1.4. Compound Value Restriction

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

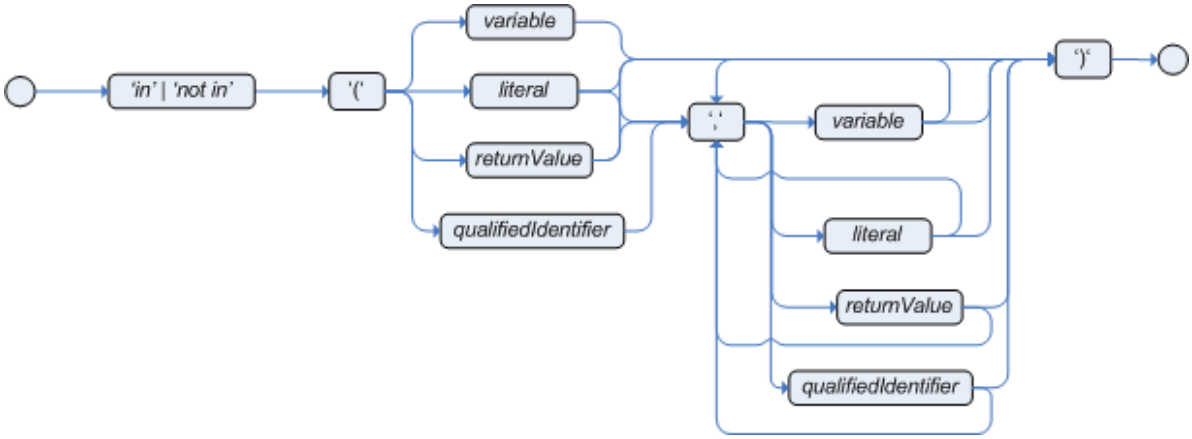


Figure 5.27. compoundValueRestriction

Example 5.27. Compound Restriction using 'in'

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

5.5.2.1.1.5. Multi Restriction

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

Figure 5.28. multiRestriction

Figure 5.29. restrictionGroup

Example 5.28. Multi Restriction

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

5.5.2.1.2. Inline Eval Constraints

Figure 5.30. Inline Eval Expression

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

This example will find all pairs of male/femal people where the male is 2 years older than the female; the boyAge variable is auto created as part of the autovivification process.

Example 5.29. Return Value operator

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

5.5.2.1.3. Nested Accessors

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

Example 5.30. Nested Accessors

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

is internally rewriten as an MVEL inline eval:

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

Important

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

5.5.2.2. 'and'

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

Figure 5.31. prefixAnd

Example 5.31. prefixAnd

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

Example 5.32. implicit root prefixAnd

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

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

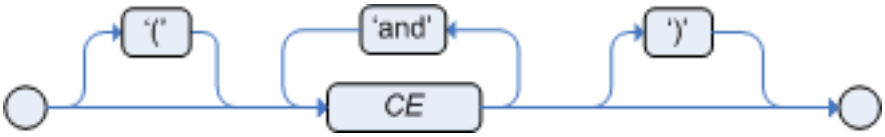


Figure 5.32. infixAnd

Example 5.33. infixAnd

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

5.5.2.3. 'or'

The 'or' Conditional Element is used to group together other Conditional Elements. Drools supports both prefix and infix; although prefix is the preferred option as grouping is implicit which avoids confusion. The behavior of the 'or' Conditional Element is different than the '||' connective for constraints and restrictions in field constraints. The engine actually has no understanding of 'or' Conditional Elements, instead via a number of different logic transformations the rule is re-written as a

number of subrules; the rule now has a single 'or' as the root node and a subrule per logical outcome. Each subrule can activate and fire like any normal rule, there is no special behavior or interactions between the subrules - this can be most confusing to new rule authors.

Figure 5.33. prefixOr

Example 5.34. prefixOr

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

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

Figure 5.34. infixOr

Example 5.35. infixAnd

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

The 'or' Conditional Element also allows for optional pattern binding; which means each resulting subrule will bind it's pattern to the pattern binding.

Example 5.36. or with binding

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

Explicit binding on each Pattern is also allowed.

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

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

The best way to think of the OR conditional element is as a shortcut for generating 2 additional rules. When you think of it that way, its clear that for a single rule there could be multiple activations if both sides of the OR conditional element are true.

5.5.2.4. 'eval'

Figure 5.35. eval

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

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

For folks who are familiar with Drools 2.x lineage, the old Drools parameter and condition tags are equivalent to binding a variable to an appropriate type, and then using it in an eval node.

Example 5.37. eval

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

5.5.2.5. 'not'

Figure 5.36. not

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

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

Example 5.38. No Busses

```
not Bus()
```

Example 5.39. No red Busses

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

5.5.2.6. 'exists'

Figure 5.37. exists

'exists' is first order logic's Existential Quantifier and checks for the existence of something in the Working Memory. Think of exist as meaning "at least one..". It is different from just having the Pattern on its own; which is more like saying "for each one of...". if you use exist with a Pattern, then the rule will only activate once regardless of how much data there is in working memory that matches that condition.

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

Example 5.40. Atleast one Bus

```
exists Bus()
```

Example 5.41. Atleast one red Bus

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

5.5.2.7. 'forall'

Figure 5.38. forall

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

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

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

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

Example 5.42. Single Pattern Forall

```
rule "All Buses are Red"
```

```
when
    forall( Bus( color == 'red' ) )
then
    # all asserted Bus facts are red
end
```

The above is exactly the same as writing:

Another example of multi-pattern forall:

Example 5.43. Multi-Pattern Forall

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

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

Example 5.44. Combining Forall with Not CE

```
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
```

As a side note, forall Conditional Element is equivalent to writing:

```
not( <first pattern> and not ( and <remaining patterns> ) )
```

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

5.5.2.8. From

Figure 5.39. from

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

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

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

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

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

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

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

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

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

The next example shows how we can reason over the results of a hibernate query. The Restaurant pattern will reason over and bind with each result in turn:

5.5.2.9. 'collect'

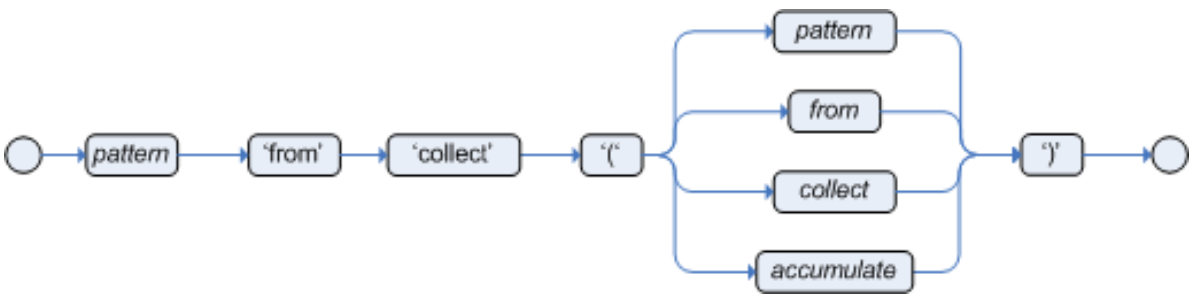


Figure 5.40. collect

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

```
import java.util.ArrayList

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

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

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

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

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

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

```
import java.util.LinkedList;

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

5.5.2.10. 'accumulate'

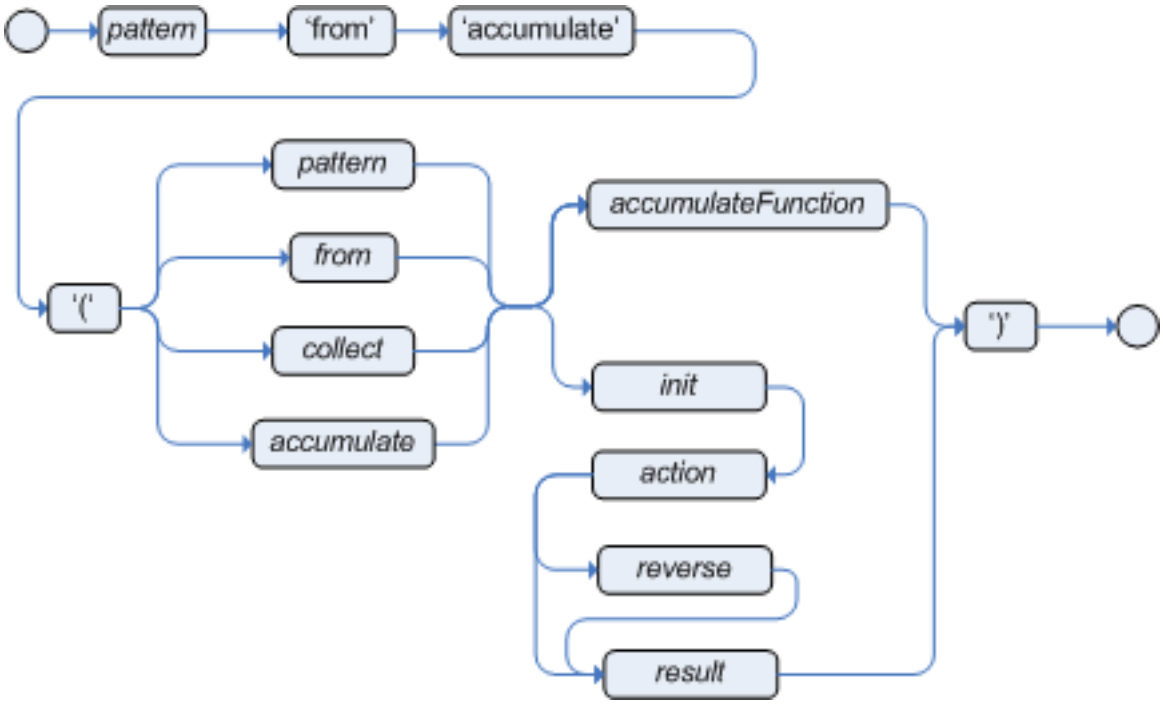


Figure 5.41. accumulate

The **accumulate** Conditional Element is a more flexible and powerful form of **collect** Conditional Element, in the sense that it can be used to do what **collect** CE does and also do things that **collect** CE is not capable to do. Basically what it does is it allows a rule to iterate over a collection of objects, executing custom actions for each of the elements, and at the end return a result object.

The general syntax of the **accumulate** CE is:

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

The meaning of each of the elements is the following:

- **<source pattern>**: the source pattern is a regular pattern that the engine will try to match against each of the source objects.
- **<init code>**: this is a semantic block of code in the selected dialect that will be executed once for each tuple, before iterating over the source objects.
- **<action code>**: this is a semantic block of code in the selected dialect that will be executed for each of the source objects.
- **<reverse code>**: this is an optional semantic block of code in the selected dialect that if present will be executed for each source object that no longer matches the source pattern. The objective of this code block is to "undo" any calculation done in the <action code> block, so that the engine can do decremental calculation when a source object is modified or retracted, hugely improving performance of these operations.
- **<result expression>**: this is a semantic expression in the selected dialect that is executed after all source objects are iterated.

- **<result pattern>**: this is a regular pattern that the engine tries to match against the object returned from the <result expression>. If it matches, the **accumulate** conditional element evaluates to **true** and the engine proceeds with the evaluation of the next CE in the rule. If it does not matches, the **accumulate** CE evaluates to **false** and the engine stops evaluating CEs for that rule.

It is easier to understand if we look at an example:

```
rule "Apply 10% discount to orders over US$ 100,00"
when
    $order : Order()
    $total : Number( doubleValue > 100 )
             from accumulate( OrderItem( order == $order, $value : value ),
                              init( double total = 0; ),
                              action( total += $value; ),
                              reverse( total -= $value; ),
                              result( total ) )
then
    # apply discount to $order
end
```

In the above example, for each Order() in the working memory, the engine will execute the **init code** initializing the total variable to zero. Then it will iterate over all OrderItem() objects for that order, executing the **action** for each one (in the example, it will sum the value of all items into the total variable). After iterating over all OrderItem, it will return the value corresponding to the **result expression** (in the above example, the value of the total variable). Finally, the engine will try to match the result with the Number() pattern and if the double value is greater than 100, the rule will fire.

The example used java as the semantic dialect, and as such, note that the usage of ';' is mandatory in the init, action and reverse code blocks. The result is an expression and as such, it does not admit ';'. If the user uses any other dialect, he must comply to that dialect specific syntax.

As mentioned before, the **reverse code** is optional, but it is strongly recommended that the user writes it in order to benefit from the *improved performance on update and retracts*.

The **accumulate** CE can be used to execute any action on source objects. The following example instantiates and populates a custom object:

```
rule "Accumulate using custom objects"
when
    $person   : Person( $likes : likes )
    $cheesery : Cheesery( totalAmount > 100 )
                from accumulate( $cheese : Cheese( type == $likes ),
                                 init( Cheesery cheesery = new Cheesery(); ),
                                 action( cheesery.addCheese( $cheese ); ),
                                 reverse( cheesery.removeCheese( $cheese ); ),
                                 result( cheesery ) );
then
    // do something
end
```

5.5.2.10.1. Accumulate Functions

The accumulate CE is a very powerful CE, but it gets real declarative and easy to use when using predefined functions that are known as Accumulate Functions. They work exactly like accumulate, but instead of explicitly writing custom code in every accumulate CE, the user can use predefined code for common operations.

For instance, the rule to apply discount on orders written in the previous section, could be written in the following way, using Accumulate Functions:

```
rule "Apply 10% discount to orders over US$ 100,00"
when
    $order : Order()
    $total : Number( doubleValue > 100 )
             from accumulate( OrderItem( order == $order, $value : value ),
                              sum( $value ) )
then
    # apply discount to $order
end
```

In the above example, sum is an AccumulateFunction and will sum the $value of all OrderItems and return the result.

Drools 4.0 ships with the following built in accumulate functions:

- average
- min
- max
- count
- sum

These common functions accept any expression as input. For instance, if someone wants to calculate the average profit on all items of an order, a rule could be written using the average function:

```
rule "Average profit"
when
    $order : Order()
    $profit : Number()
              from accumulate( OrderItem( order == $order, $cost : cost, $price : price )
                               average( 1 - $cost / $price ) )
then
    # average profit for $order is $profit
end
```

Accumulate Functions are all pluggable. That means that if needed, custom, domain specific functions can easily be added to the engine and rules can start to use them without any restrictions. To implement a new Accumulate Functions all one needs to do is to create a java class that implements the org.drools.base.acumulators.AccumulateFunction interface and add a line to the configuration file or set a system property to let the engine know about the new function. As an example of an Accumulate Function implementation, the following is the implementation of the "average" function:

```
/*
 * Copyright 2007 JBoss Inc
 *
 * Licensed under the Apache License, Version 2.0 (the "License");
 * you may not use this file except in compliance with the License.
 * You may obtain a copy of the License at
 *
 *      http://www.apache.org/licenses/LICENSE-2.0
 *
 * Unless required by applicable law or agreed to in writing, software
 * distributed under the License is distributed on an "AS IS" BASIS,
 * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
 * See the License for the specific language governing permissions and
 * limitations under the License.
 *
 * Created on Jun 21, 2007
 */
```

```
package org.drools.base.accumulators;


/**
 * An implementation of an accumulator capable of calculating average values
 *
 * @author etirelli
 *
 */
public class AverageAccumulateFunction implements AccumulateFunction {

    protected static class AverageData {
        public int    count = 0;
        public double total = 0;
    }

    /* (non-Javadoc)
     * @see org.drools.base.accumulators.AccumulateFunction#createContext()
     */
    public Object createContext() {
        return new AverageData();
    }

    /* (non-Javadoc)
     * @see org.drools.base.accumulators.AccumulateFunction#init(java.lang.Object)
     */
    public void init(Object context) throws Exception {
        AverageData data = (AverageData) context;
        data.count = 0;
        data.total = 0;
    }

    /* (non-Javadoc)
     * @see org.drools.base.accumulators.AccumulateFunction#accumulate(java.lang.Object,
 java.lang.Object)
     */
    public void accumulate(Object context,
                           Object value) {
        AverageData data = (AverageData) context;
        data.count++;
        data.total += ((Number) value).doubleValue();
    }

    /* (non-Javadoc)
     * @see org.drools.base.accumulators.AccumulateFunction#reverse(java.lang.Object,
 java.lang.Object)
     */
    public void reverse(Object context,
                        Object value) throws Exception {
        AverageData data = (AverageData) context;
        data.count--;
        data.total -= ((Number) value).doubleValue();
    }

    /* (non-Javadoc)
     * @see org.drools.base.accumulators.AccumulateFunction#getResult(java.lang.Object)
     */
    public Object getResult(Object context) throws Exception {
        AverageData data = (AverageData) context;
        return new Double( data.count == 0 ? 0 : data.total / data.count );
    }

    /* (non-Javadoc)
     * @see org.drools.base.accumulators.AccumulateFunction#supportsReverse()
     */
    public boolean supportsReverse() {
        return true;
```

```
    }

}
```

The code for the function is very simple, as we could expect, as all the "dirty" integration work is done by the engine. Finally, to plug the function into the engine, we added it to the configuration file:

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

Where "drools.accumulate.function." is a prefix that must always be used, "average" is how the function will be used in the rule file, and "org.drools.base.accumulators.AverageAccumulateFunction" is the fully qualified name of the class that implements the function behavior.

5.5.3. The Right Hand Side (then)

The Right Hand Side (RHS) is a common name for the consequence or action part of the rule; this part should contain a list of actions to be executed. It is bad practice to use imperative or conditional code in the RHS of a rule; as a rule should be atomic in nature - "when this, then do this", not "when this, maybe do this". The RHS part of a rule should also be kept small, thus keeping it declarative and readable. If you find you need imperative and/or conditional code in the RHS, then maybe you should be breaking that rule down into multiple rules. The main purpose of the RHS is to insert, retractor modify working memory data. To assist with there there are a few convenience methods you can use to modify working memory; without having to first reference a working memory instance.

"update(object, handle);" will tell the engine that an object has changed (one that has been bound to something on the LHS) and rules may need to be reconsidered.

"update(object);" can also be used, here the KnowledgeHelper will lookup the facthandle for you, via an identity check, for the passed object.

"insert(new Something());" will place a new object of your creation in working memory.

"insertLogical(new Something());" is similar to insert, but the object will be automatically retracted when there are no more facts to support the truth of the currently firing rule.

"retract(handle);" removes an object from working memory.

These convenience methods are basically macros that provide short cuts to the KnowldgeHelper instance (refer to the KnowledgeHelper interface for more advanced operations). The KnowledgeHelper interface is made available to the RHS code block as a variable called "drools". If you provide "Property Change Listeners" to your java beans that you are inserting into the engine, you can avoid the need to call "update" when the object changes.

5.5.4. A note on auto boxing/unboxing and primitive types

Drools 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 Drools 3.0 where all primitives was autoboxed, requiring manual unboxing. A variable bound to an object wrapper will remain as an object; the existing jdk1.5 and jdk5 rules to handling auto boxing/unboxing apply in this case. When evaluating field constraints the system attempts to coerce one of the values into a comparable format; so a primitive is comparable to an object wrapper.

5.6. Query

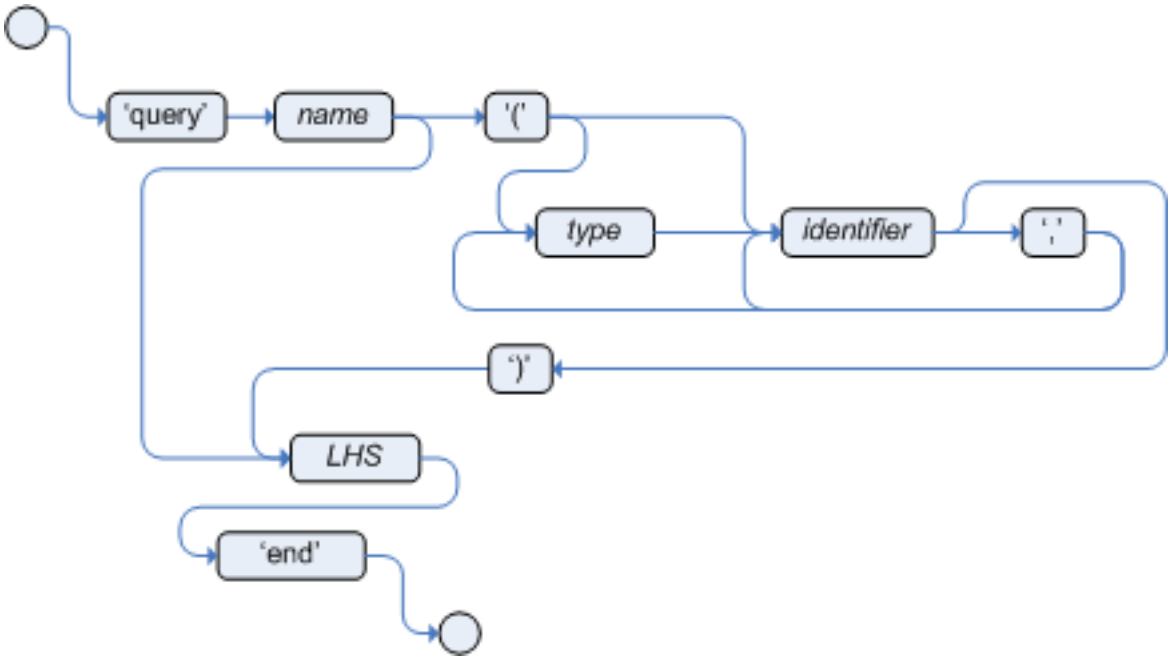


Figure 5.42. query

A query contains the structure of the LHS of a rule only (you don't specify "when" or "then"). It is simply a way to query the working memory for facts that match the conditions stated. A query has an optional set of parameters, that can also be optionally typed, if type is not given then Object type is assumed and the engine will attempt to co-erce the values as needed.

To return the results use WorkingMemory.getQueryResults("name") - where "name" is query name. Query names are global to the RuleBase, so do not add queries of the same name to different packages for the same Rule Base. This contains a list of query results, which allow you to to get to the objects that matched the query.

This example creates a simple query for all the people over the age of 30

Example 5.45. Query People over the age of 30

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

Example 5.46. 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 row returns a QueryResult which we can use to access each of the columns in the Tuple. Those columns can be access by bound declaration name or index position.

Example 5.47. Query People over the age of 30

```
QueryResults results = workingMemory.getQueryResults( "people over the age of 30" );
```

```
System.out.println( "we have " + results.size() + " people over the age  of 30" );

System.out.println( "These people are are over 30:" );

for ( Iterator it = results.iterator; it.hasNext(); ) {
    QueryResult result = ( QueryResult ) it.next();
    Person person = ( Person ) result.get( "person" );
    System.out.println( person.getName() + "\n" );
}
```

5.7. Domain Specific Languages

As mentioned previously,
(or DSLs) are a way of extending the rule language to your problem domain. They are wired in to the rule language for you, and can make use of all the underlying rule language and engine features.

DSLs are used in the IDE. Of course as rules are text, you can use them even without this tooling.

5.7.1. When to use a DSL

DSLs can serve as a layer of separation between rule authoring (and rule authors) and the domain objects that the engine operates on. DSLs can also act as "templates" of conditions or actions that are used over and over in your rules, perhaps only with parameters changing each time. If your rules need to be read and validated by less technical folk, (such as Business Analysts) the DSLs are definitely for you. If the conditions or consequences of your rules follow similar patterns which you can express in a template. You wish to hide away your implementation details, and focus on the business rule. You want to provide a controlled means of editing rules based on pre-defined templates.

DSLs have no impact on the rules at runtime, they are just a parse/compile time feature.

Note that Drools 4 DSLs are quite different from Drools 2 XML based DSLs. It is still possible to do Drools 2 style XML languages - if you require this, then take a look at the Drools 4 XML rule language, and consider using XSLT to map from your XML language to the Drools 4 XML language.

5.7.2. Editing and managing a DSL

A DSL's configuration like most things is stored in plain text. If you use the IDE, you get a nice graphical editor (with some validation), but the format of the file is quite simple, and is basically a properties file.

Note that since Drools 4.0, DSLs have become more powerful in allowing you to customise almost any part of the language, including keywords. Regular expressions can also be used to match words/sentences if needed (this is provided for enhanced localisation). However, not all features are supported by all the tools (although you can use them, the content assistance just may not be 100% accurate in certain cases).

Example 5.48. Example mapping

```
[when]This is {something}=Something(something=={something})
```

Referring to the above example, the [when] refers to the scope of the expression: ie does it belong on the LHS or the RHS of a rule. The part after the [scope] is the expression that you use in the rule (typically a natural language expression, but it doesn't have to be). The part on the right of the "=" is the mapping into the rule language (of course the form of this depends on if you are talking about the

RHS or the LHS - if its the LHS, then its the normal LHS syntax, if its the RHS then its fragments of java code for instance).

The parser will take the expression you specify, and extract the values that match where the {something} (named Tokens) appear in the input. The values that match the tokens are then interpolated with the corresponding {something} (named Tokens) on the right hand side of the mapping (the target expression that the rule engine actually uses).

Note also that the "sentences" above can be regular expressions. This means the parser will match the sentence fragements that match the expressions. This means you can use (for instance) the '?' to indicate the character before it is optional (think of each sentence as a regular expression pattern - this means if you want to use regex characters - you will need to escape them with a '\' of course.

It is important to note that the DSL expressions are processed one line at a time. This means that in the above example, all the text after "There is " to the end of the line will be included as the value for "{something}" when it is interpolated into the target string. This may not be exactly what you want, as you may want to "chain" together different DSL expressions to generate a target expression. The best way around this is to make sure that the {tokens} are enclosed with characters or words. This means that the parser will scan along the sentence, and pluck out the value BETWEEN the characters (in the example below they are double-quotes). Note that the characters that surround the token are not included in when interpolating, just the contents between them (rather then all the way to the end of the line, as would otherwise be the case).

As a rule of thumb, use quotes for textual data that a rule editor may want to enter. You can also wrap words around the {tokens} to make sure you enclose the data you want to capture (see other example).

Example 5.49. Example with quotes

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

It is a good idea to try and avoid punctuation in your DSL expressions where possible, other then quotes and the like - keep it simple it things will be easier. Using a DSL can make debugging slightly harder when you are first building rules, but it can make the maintenance easier (and of course the readability of the rules).

The "{" and "}" characters should only be used on the left hand side of the mapping (the expression) to mark tokens. On the right hand side you can use "{" and "}" on their own if needed - such as

```
if (foo) {
    doSomething(); }
```

as well as with the token names as shown above.

Don't forget that if you are capturing strings from users, you will also need the quotes on the right hand side of the mapping, just like a normal rule, as the result of the mapping must be a valid expression in the rule language.

Example 5.50. Some more examples

```
#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
```

Referring to the above examples, this would render the following input as shown below:

Example 5.51. Some examples as processed

```
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")
```

5.7.3. Using a DSL in your rules

A good way to get started if you are new to Rules (and DSLs) is just write the rules as you normally would against your object model. You can unit test as you go (like a good agile citizen!). Once you feel comfortable, you can look at extracting a domain language to express what you are doing in the rules. Note that once you have started using the "expander" keyword, you will get errors if the parser does not recognize expressions you have in there - you need to move everything to the DSL. As a way around this, you can prefix each line with ">" and it will tell the parser to take that line literally, and not try and expand it (this is handy also if you are debugging why something isn't working).

Also, it is better to rename the extension of your rules file from ".drl" to ".dslr" when you start using DSLs, as that will allow the IDE to correctly recognize and work with your rules file.

As you work through building up your DSL, you will find that the DSL configuration stabilizes pretty quickly, and that as you add new rules and edit rules you are reusing the same DSL expressions over and over. The aim is to make things as fluent as possible.

To use the DSL when you want to compile and run the rules, you will need to pass the DSL configuration source along with the rule source.

```
PackageBuilder builder = new PackageBuilder();
builder.addPackageFromDrl( source, dsl );
//source is a reader for the rule source, dsl is a reader for the DSL configuration
```

You will also need to specify the expander by name in the rule source file:

```
expander your-expander.dsl
```

Typically you keep the DSL in the same directory as the rule, but this is not required if you are using the above API (you only need to pass a reader). Otherwise everything is just the same.

You can chain DSL expressions together on one line, as long as it is clear to the parser what the {tokens} are (otherwise you risk reading in too much text until the end of the line). The DSL expressions are processed according to the mapping file, top to bottom in order. You can also have the resulting rule expressions span lines - this means that you can do things like:

Example 5.52. Rule Expression spanning lines

```
There is a person called Bob who is happy
  Or
```

```
There is a person called Mike who is sad
```

Of course this assumes that "Or" is mapped to the "or" conditional element (which is a sensible thing to do).

5.7.4. Adding constraints to facts

A common requirement when writing rule conditions is to be able to add many constraints to fact declarations. A fact may have many (dozens) of fields, all of which could be used or not used at various times. To come up with every combination as separate DSL statements would in many cases not be feasible.

The DSL facility allows you to achieve this however, with a simple convention. If your DSL expression starts with a "-", then it will be assumed to be a field constraint, which will be added to the declaration that is above it (one per line).

This is easier to explain with an example. Lets take look at Cheese class, with the following fields: type, price, age, country. We can express some LHS condition in normal DRL like the following

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

If you know ahead of time that you will use all the fields, all the time, it is easy to do a mapping using the above techniques. However, chances are that you will have many fields, and many combinations. If this is the case, you can setup your mappings like so:

```
[when]There is a Cheese with=Cheese()
[when]- age is less than {age}=age<{age}
[when]- type is '{type}'=type=='{type}'
[when]- country equal to '{country}'=country=='{country}'
```

IMPORTANT: It is NOT possible to use the "-" feature after an **accumulate** statement to add constraints to the accumulate pattern. This limitation will be removed in the future.

You can then write rules with conditions like the following:

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

The parser will pick up the "-" lines (they have to be on their own line) and add them as constraints to the declaration above. So in this specific case, using the above mappings, is the equivalent to doing (in DRL):

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

The parser will do all the work for you, meaning you just define mappings for individual constraints, and can combine them how you like (if you are using context assistant, if you press "-" followed by CTRL+space it will conveniently provide you with a filtered list of field constraints to choose from.

To take this further, after alter the DSL to have [when][org.drools.Cheese]- age is less than {age} ... (and similar to all the items in the example above).

The extra [org.drools.Cheese] indicates that the sentence only applies for the main constraint sentence above it (in this case "There is a Cheese with"). For example, if you have a class called "Cheese" - then if you are adding contraints to the rule (by typing "-" and waiting for

content assistance) then it will know that only items marked as having an object-scope of "com.yourcompany.Something" are valid, and suggest only them. This is entirely optional (you can leave out that section if needed - OR it can be left blank).

5.7.5. How it works

DSLs kick in when the rule is parsed. The DSL configuration is read and supplied to the parser, so the parser can "expand" the DSL expressions into the real rule language expressions.

When the parser is processing the rules, it will check if an "expander" representing a DSL is enabled, if it is, it will try to expand the expression based on the context of where it is the rule. If an expression can not be expanded, then an error will be added to the results, and the line number recorded (this insures against typos when editing the rules with a DSL). At present, the DSL expander is fairly space sensitive, but this will be made more tolerant in future releases (including tolerance for a wide range of punctuation).

The expansion itself works by trying to match a line against the expression in the DSL configuration. The values that correspond to the token place holders are stored in a map based on the name of the token, and then interpolated to the target mapping. The values that match the token placeholders are extracted by either searching until the end of the line, or until a character or word after the token place holder is matched. The "{" and "}" are not included in the values that are extracted, they are only used to demarcate the tokens - you should not use these characters in the DSL expression (but you can in the target).

5.7.6. Creating a DSL from scratch

DSLs can be aid with capturing rules if the rules are well known, just not in any technically usable format (ie. sitting around in people brains). Until we are able to have those little sockets in our necks like in the Matrix, our means of getting stuff into computers is still the old fashioned way.

Rules engines require a object or data model to operate on - in many cases you may know this up front. In other cases the model will be discovered with the rules. In any case, rules generally work better with simpler flatter object models. In some cases, this may mean having a rule object model which is a subset of the main applications model (perhaps mapped from it). Object models can often have complex relationships and hierarchies in them - for rules you will want to simplify and flatten the model where possible, and let the rule engine infer relationships (as it provides future flexibility). As stated previously, DSLs can have an advantage of providing some insulation between the object model and the rule language.

Coming up with a DSL is a collaborative approach for both technical and domain experts. Historically there was a role called "knowledge engineer" which is someone skilled in both the rule technology, and in capturing rules. Over a short period of time, your DSL should stabilize, which means that changes to rules are done entirely using the DSL. A suggested approach if you are starting from scratch is the following workflow:

- Capture rules as loose "if then" statements - this is really to get an idea of size and complexity (possibly in a text document).
- Look for recurring statements in the rules captured. Also look for the rule objects/fields (and match them up with what may already be known of the object model).
- Create a new DSL, and start adding statements from the above steps. Provide the "holes" for data to be edited (as many statements will be similar, with only some data changing).
- Use the above DSL, and try to write the rules just like that appear in the "if then" statements from the first and second steps. Iterate this process until patterns appear and things stabilize. At this stage, you are not so worried about the rule language underneath, just the DSL.

- At this stage you will need to look at the Objects, and the Fields that are needed for the rules, reconcile this with the datamodel so far.
- Map the DSL statements to the rule language, based on the object model. Then repeat the process. Obviously this is best done in small steps, to make sure that things are on the right track.

5.7.7. Scope and keywords

If you are editing the DSL with the GUI, or as text, you will notice there is a [scope] item at the start of each mapping line. This indicates if the sentence/word applies to the LHS, RHS or is a keyword. Valid values for this are [condition], [consequence] and [keyword] (with [when] and [then] being the same as [condition] and [consequence] respectively). When [keyword] is used, it means you can map any keyword of the language like "rule" or "end" to something else. Generally this is only used when you want to have a non English rule language (and you would ideally map it to a single word).

5.7.8. DSLs in the IDE

You can use DSLs in both guided editor rules, and textual rules that use a dsl.

In the guided editor - the DSLs generally have to be simpler - what you are doing is defining little "forms" to capture data from users in text fields (ie as you pick a DSL expression - it will add an item to the GUI which only allows you enter data in the {token} parts of a DSL expression). You can not use sophisticated regular expressions to match text. However, in textual rules (which have a .dslr extension in the IDE) you are free to use the full power as needed.

The DSLs are not automatically included when building a package in the IDE. You need to either use the drools-ant task or use the code shown in the sections above.

5.8. Rule Flow

Important

The JBoss Enterprise SOA Platform does not support the use of RuleFlow for BPM workflow or service orchestration.

JBoss jBPM is the supported means of service orchestration and BPM workflow. Refer to the jBPM Reference Guide [1] for additional information.

JBoss Rules already provides some functionality to define the order in which rules should be executed, like salience, activation groups, etc. When dealing with (possibly a lot of) large rule-sets, managing the order in which rules are evaluated might become complex. Ruleflow allows you to specify the order in which rule sets should be evaluated by using a flow chart. This allows you to define which rule sets should be evaluated in sequence or in parallel, to specify conditions under which rule sets should be evaluated, etc. This chapter contains a few ruleflow examples.

A rule flow is a graphical description of a sequence of steps that the rule engine needs to take, where the order is important. The ruleflow can also deal with conditional branching, parallelism, synchronization, etc.

[1] The JBoss Enterprise SOA Platform JBPM Reference Guide is provided as the file **`JBPM_Reference_Guide.pdf`** or can be viewed online at *http://www.redhat.com/docs/en-US/JBoss_SOA_Platform/*

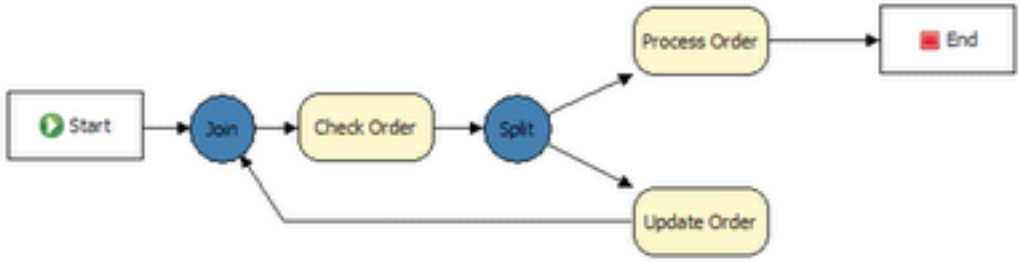


Figure 5.43. Ruleflow

To use a ruleflow to describe the order in which rules should be evaluated, you should first group rules into rulefow-groups using the ruleflow-group rule attribute ("options" in the GUI). Then you should create a ruleflow graph (which is a flow chart) that graphically describe the order in which the rules should be considered (by specifying the order in which the ruleflow-groups should be evaluated).

5.8.1. Assigning rules to a ruleflow group

```
rule 'YourRule'
    ruleflow-group 'group1'
when
    ...
then
    ...
end
```

This rule will then be placed in the ruleflow-group called "group1".

5.8.2. A simple ruleflow

Figure 5.44. Ruleflow

The above rule flow specifies that the rules in the group "Check Order" must be executed before the rules in the group "Process Order". This means that only rules which are marked as having a ruleflow-group of "Check Order" will be considered first, and then "Process Order". That's about it. You could achieve similar results with either using salience (setting priorities, but this is harder to maintain, and makes the time-relationship implicit in the rules), or agenda groups. However, using a ruleflow makes the order of processing explicit, almost like a meta-rule, and makes managing more complex situations a lot easier. The various elements that can be used inside a ruleflow will be explained in more detail later.

5.8.3. How to build a rule flow

Ruleflows can only be created by using the graphical ruleflow editor which is part of the Drools plugin for Eclipse. JBoss Developer Studio includes this functionality.

Once you have set up a JBoss Rules project (check the IDE chapter if you do not know how to do this), you can start adding ruleflows. When in a project, use "control+N" to launch the new wizard, or right-click the directory you would like to put your ruleflow in and select "New ... Other ...":

Choose the section on "Drools" and then pick "RuleFlow file". This will create a new .rf file.

Next you will see the graphical ruleflow editor. Now would be a good time to switch to the "Drools perspective" (if you haven't done so already) - this will tweak the UI so it is optimal for rules. Then ensure that you can see the "properties" panel down the bottom of the eclipse window, as it will be necessary to fill in the different properties of the elements in your ruleflow. If you cannot see the properties view, open it using the Menu Window - Show View - Other ..., and under the General folder select the Properties view.

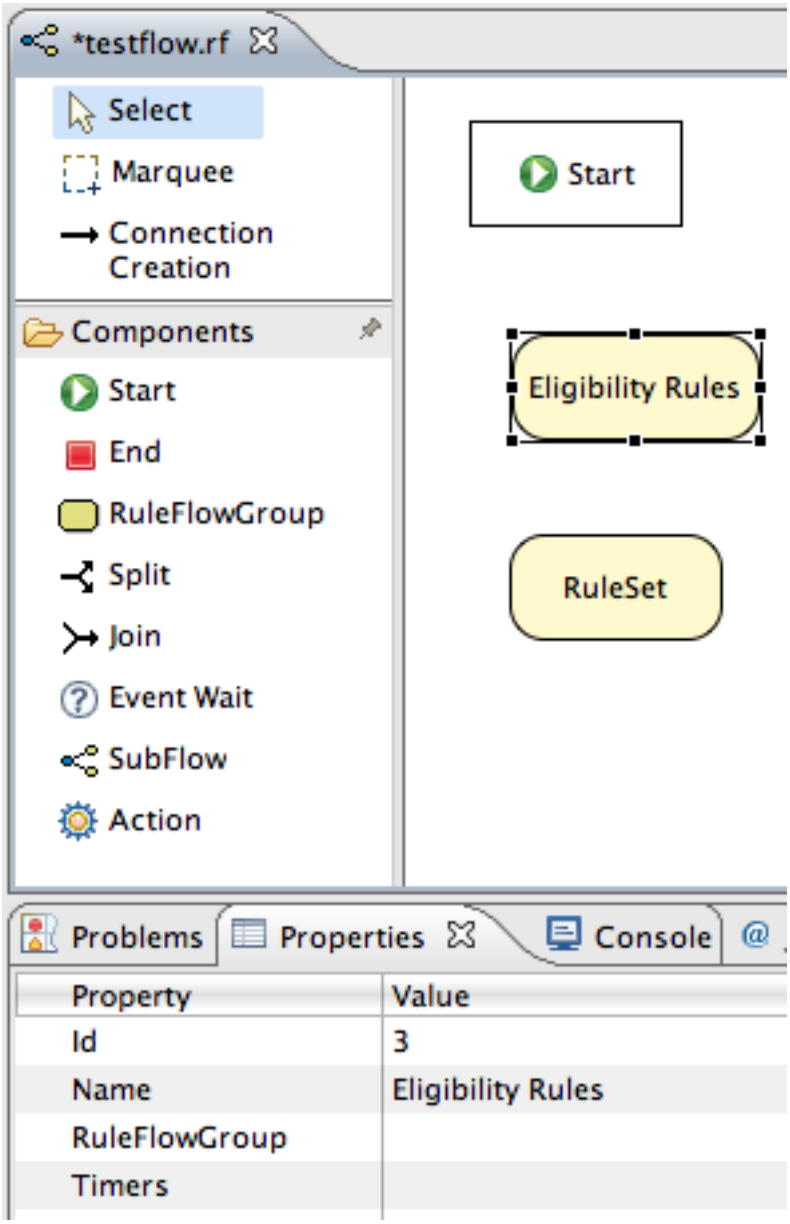


Figure 5.45. Groups

The RuleFlow editor consists of a palette, a canvas and an outline view. To add new elements to the canvas, select the element you would like to create in the palette and then add them to the canvas by clicking on the preferred location. For example, click on the RuleFlowGroup icon in the Component Palette of the GUI - you can then draw a few rule flow groups. Clicking on an element in your ruleflow allows you to set the properties of that element.

Click on a ruleflow group, and you should see the following:

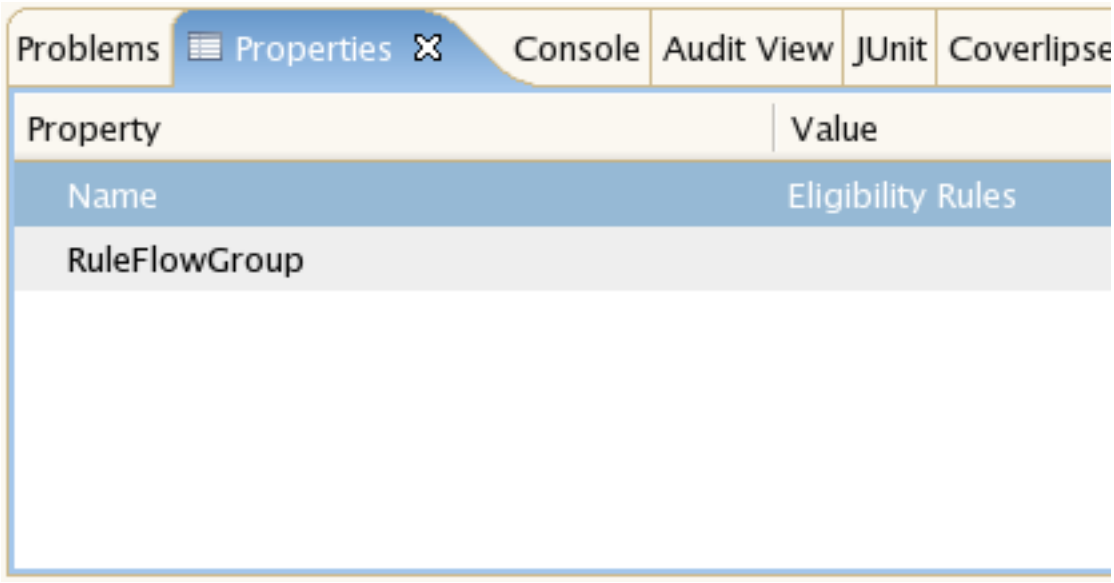


Figure 5.46. Group properties

You can see here you set the visible name, but you also need to set the actual group name that is used in the rules.

Next step is to join the groups together (if its a simple sequence of steps) - you use this by using "create connection" from the component palette. You should also create an "End" node (also from the component palette).

In practice, if you are using ruleflow, you will most likely be doing more then setting a simple sequence of groups to progress though. You are more likely modeling branches of processing. In this case you use "Split" and "Join" items from the component palette. You use connections to connect from the start to ruleflow groups, or to Splits, and from splits to groups, joins etc. (i.e. basically like a simple flow chart that models your processing). You can work entirely graphically until you get the graph approximately right.

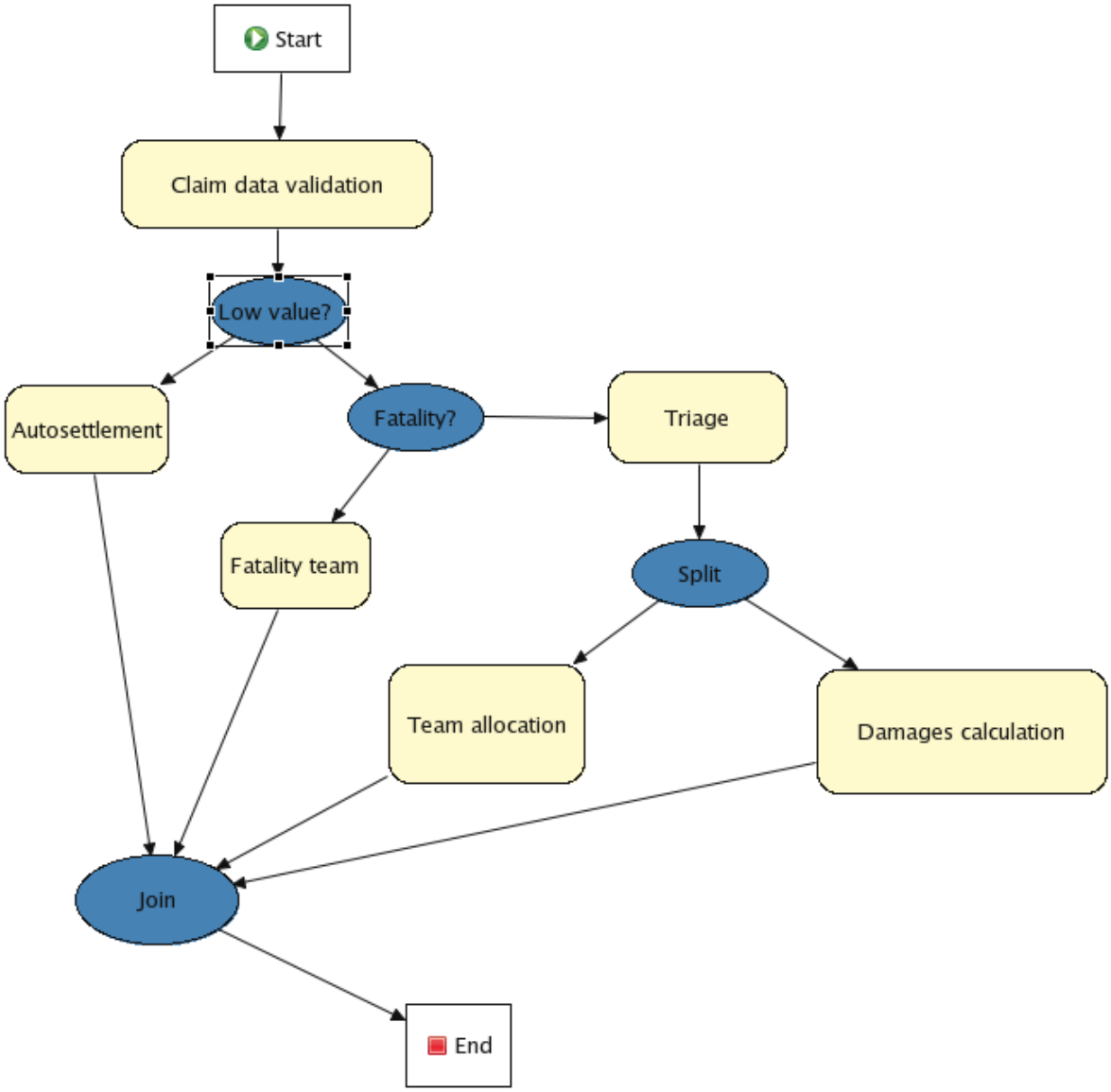


Figure 5.47. Complex ruleflow

The above flow is a more complex example. This example is an insurance claim processing rule flow. A description: Initially the claim data validation rules are processed (these check for data integrity and consistency, that all the information is there). Next there is a decision "split" - based on a condition which the rule flow checks (the value of the claim), it will either move on to an "auto-settlement" group, or to another "split", which checks if there was a fatality in the claim. If there was a fatality then it determines if the "regular" of fatality specific rules will take effect. And so on. What you can see from this is based on a few conditions in the rule flow the steps that the processing takes can be very different. Note that all the rules can be in one package - making maintenance easy. You can separate out the flow control from the actual rules.

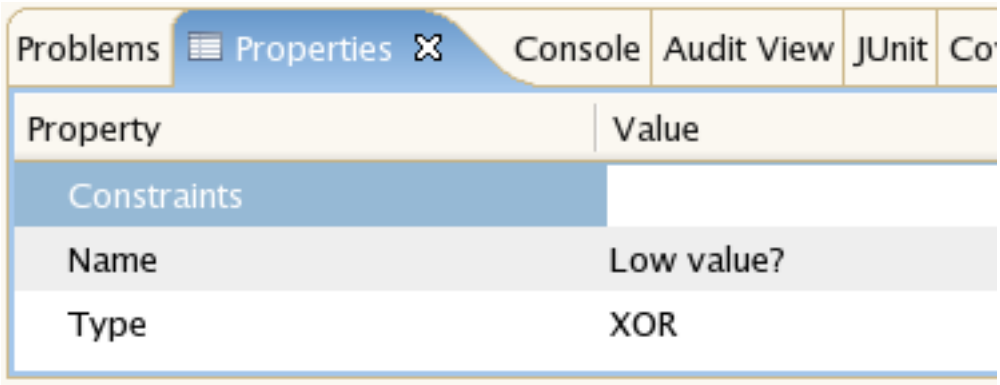


Figure 5.48. Split types

Split types (referring to the above): When you click on a split, you will see the above properties panel. You then have to choose the type: AND, OR, and XOR. The interesting ones are OR and XOR: if you choose OR, then any of the "outputs" of the split can happen (ie processing can proceed in parallel down more then one path). If you chose XOR, then it will be only one path.

If you choose OR or XOR, then in the row that has constraints, you will see a button on the right hand side that has "..." - click on this, and you will see the constraint editor. From this constraint editor, you set the conditions which the split will use to decide which "output path" will be chosen.

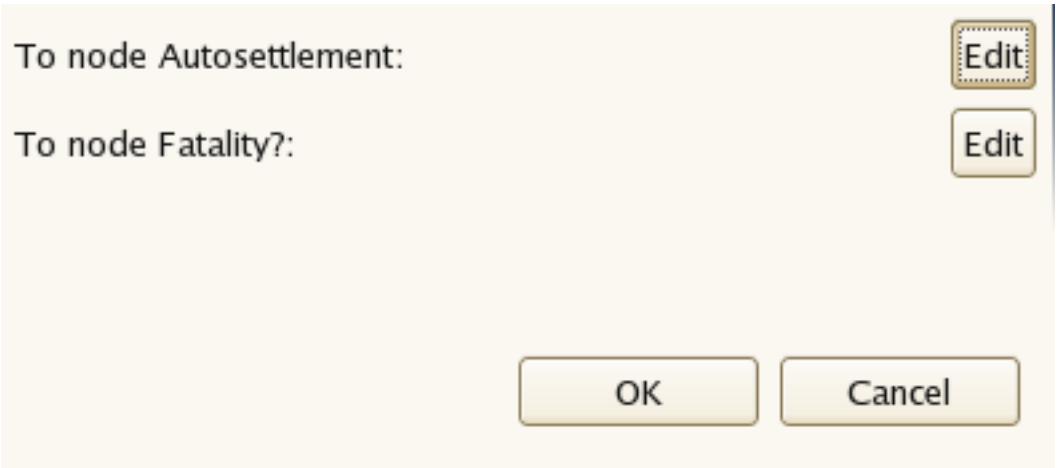


Figure 5.49. Edit constraints

Choose the output path you want to set the constraints for (e.g. Autosettlement), and then you should see the following constraint editor:

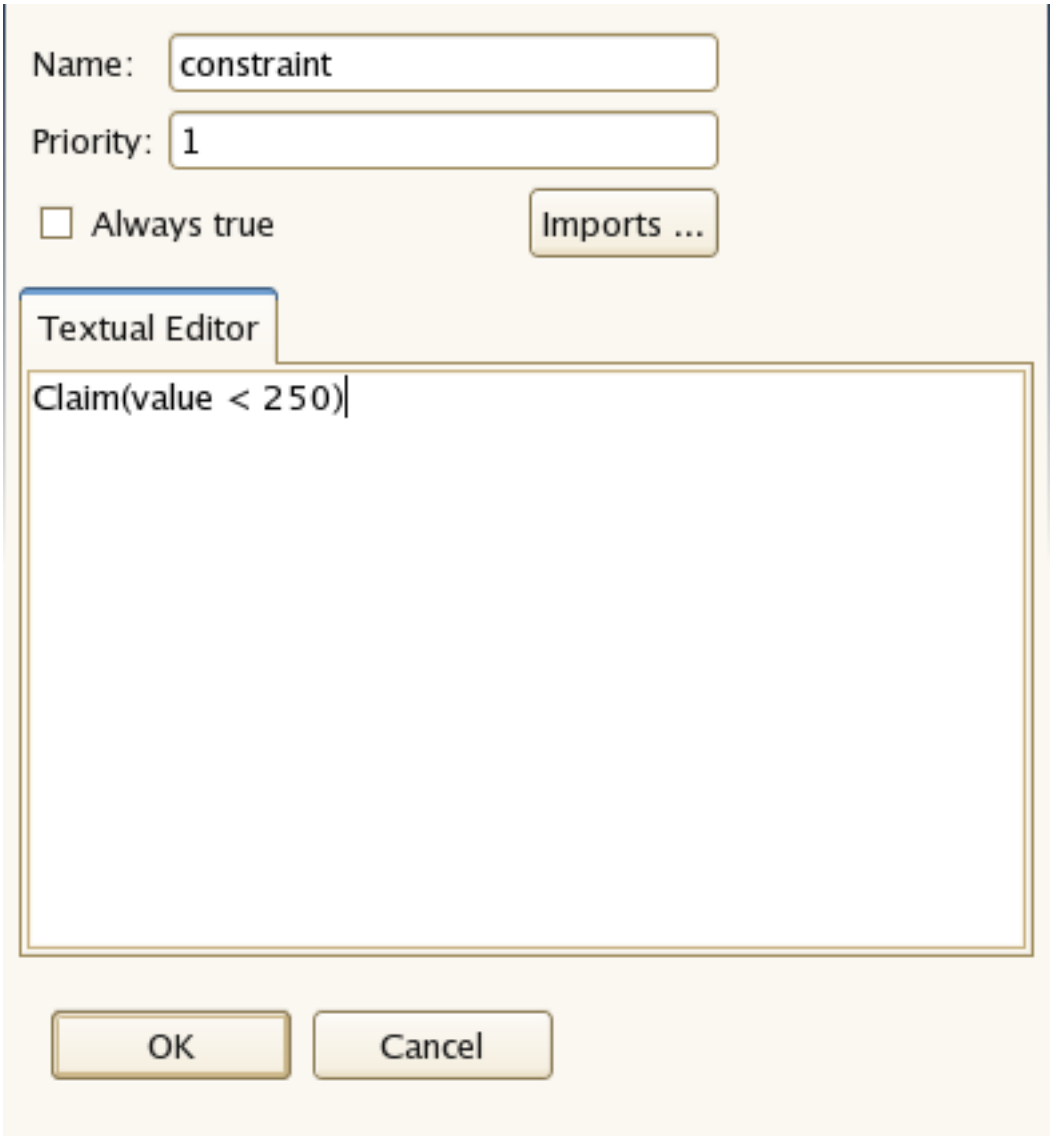


Figure 5.50. Constraint editor

This is a text editor where the constraints (which are like the condition part of a rule) are entered. These constraints operate on facts in the working memory (e.g. in the above example, it is checking for claims with a value of less than 250). Should this condition be true, then the path specified by it will be followed.

5.8.4. Using a rule flow in your application

Once you have a valid ruleflow (you can check its valid by pressing the green "tick" icon in the IDE), you can add a rule flow to a package just like a drl. However, the IDE creates two versions of your ruleflow: one containing the ruleflow definition (*.rfm) and one containing additional graphical information (*.rf). When adding a ruleflow to a package, you should make sure that you are adding the .rfm file to your ruleflow (and not the .rf file).

```
Reader rfm = ... (rule flow reader, select your .RFM file here)
packageBuilder.addRuleFlow(rfm);
```

Ruleflows are only executed if you explicitly state that they should be executed. This is because you could potentially define a lot of ruleflows in your package and the engine has no way to know when you would like to start each of these. To activate a particular ruleflow, you will need to start the

process by calling the startProcess method on the working memory. For example, if you want to start a particular workflow after you have asserted your facts into the working memory, use:

```
workingMemory.startProcess("ID_From_your_Ruleflow_properties");
```

The ruleflow id can be specified in the properties view when you click the background canvas of your ruleflow. When you call fireAllRules(), this will start executing rules, taking the order specified in the ruleflow into account.

You can also start a ruleflow process from within a rule consequence using:

```
drools.getWorkingMemory().startProcess("ID_From_your_Ruleflow_properties");
```

5.8.5. Different types of nodes in a ruleflow

A ruleflow is a flow chart where different types of nodes are linked using connections. It has the following properties: id, name and version. You can also specify how the connections are visualized on the canvas using the connection layout property.

Connection Layout Property Options

1. Manual always draws your connections as lines going straight from their start to end point. You can also use intermediate break points.
2. Shortest path is similar, but it tries to go around any obstacles is might encounter between the start and end point to avoid lines crossing nodes.
3. Manhatten draws connections by only using horizontal and vertical lines.

Ruleflow supports eight types of nodes: **`Start`**, **`End`**, **`RuleFlowGroup`**, **`Split`**, **`Join`**, **`Milestone`**, **`Subflow`**, and **`Action`**.

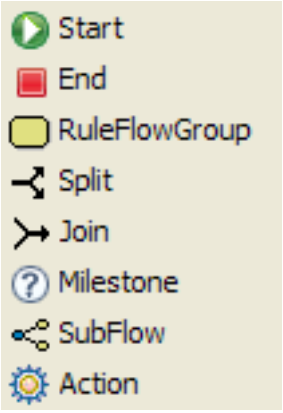


Figure 5.51. The different types of RuleFlow nodes

Start

`Start` is the start of the ruleflow. A ruleflow should have exactly one start node. The start node cannot have incoming cnnections and should have one outgoing connection. It contains one property "name" which is the display name of the node. Whenever ruleflow process is started, the ruleflow will start executing here and automatically continue to the first node linked to this start node.

End

End is the end of the ruleflow. A ruleflow should have one or more end nodes. The end node should have one incoming connection and cannot have outgoing connections. It contains one property "name" which is the display name of the node. When an end node is reached in the ruleflow, the ruleflow is terminated (including other remaining active nodes when parallelism is used).

RuleFlowGroup

RuleFlowGroup represents a set of rules. A RuleFlowGroup node should have one incoming connection and one outgoing connection. It contains a property "name" which is the display name of the node, and the property ruleflow-group which is used to specify the name of the ruleflow-group that represents the set of rules of this RuleFlowGroup node. When a RuleFlowGroup node is reached in the ruleflow, the engine will start executing rules that are part of the corresponding ruleflow-group. Execution will automatically continue to the next node if there are no more active rules in this ruleflow-group. This means that, during the execution of a ruleflow-group, it is possible that new activations belonging to the currently active ruleflow-group are added to the agenda due to changes made to the facts by the other rules. Note that the ruleflow will immediately continue with the next node if it encounters a ruleflow-group where there are no active rules at that point.

Split

Split allows you to create branches in your ruleflow. A split node should have one incoming connection and two or more outgoing connections. It contains a property "name" which is the display name of the node. There are three types of splits currently supported, **AND**, **XOR** and **OR**.

AND means that the control flow will continue in all outgoing connections simultaneously

XOR means that exactly one of the outgoing connections will be chosen. Connections are chosen by evaluating the constraints that are linked to each of the outgoing connections. Constraints are specified using the same syntax as the left-had side of a rule. The constraint with the lowest priority number that evaluates to true is selected. Note that you should make sure that at least one of the outgoing connections will evaluate to true at runtime, or the ruleflow will throw an exception at runtime if it cannot find an outgoing connection. For example, you could use a connection which is always true with a high priority number to specify what should happen if none of the other connections can be taken.

OR means that all outgoing connections whose condition evaluates to true are selected. Conditions are similar to the **XOR** split, except that the priorities are not taken into account. Note that you should make sure that at least one of the outgoing connections will evaluate to true at runtime, or the ruleflow will throw an exception at runtime if it cannot find an outgoing connection.

Join

Join allows you to synchronize multiple branches. A join node should have two or more incoming connections and one outgoing connection. It contains a property "name" which is the display name of the node. There are two types of splits currently supported, **AND** and **XOR**.

AND means that is will wait until all incoming branches are completed before continuing

XOR means that it continues if one of its incoming branches has been completed

Milestone

Milestone represents a wait state. A milestone should have one incoming connection and one outgoing connection. It contains a property "name" which is the display name of the node, and the

property "constraint" which specifies how long the ruleflow should wait in this state before continuing. For example, a milestone constraint in an order entry application might specify that the ruleflow should wait until (a fact in the working memory specifies that) no more errors are found in the given order. Constraints are specified using the same syntax as the left-had side of a rule. When a Milestone node is reached in the ruleflow, the engine will check the associated constraint. If the constraint evaluates to true directly, the flow will continue imediately. Otherwise, the flow will continue if the constraint is satisfied later on, for example when a facts in the working memory is inserted, updated or removed.

Subflow

`Subflow` represents the invocation of another ruleflow from withing this ruleflow. A subflow node should have one incoming connection and one outgoing connection. It contains a property "name" which is the display name of the node, and the property "processId" which specifies the id of the process that should be executed. When a Subflow node is reached in the ruleflow, the engine will start the process with the given id. The subflow node will only continue if that subflow process has terminated its execution. Note that the subflow process is started as an independent process, which means that the subflow process will not be terminated if this process reaches an end node.

Action

`Action` represents an action that should be executed in this ruleflow. An action node should have one incoming connection and one outgoing connection. It contains a property "name" which is the display name of the node, and the property "action" which specifies the action that should be executed. When an action node is reached in the ruleflow, it will execute the action and continue with the next node. An action should be specified as a piece of (valid) MVEL code.

5.9. XML Rule Language

As an option, Drools also supports a "native"
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.

5.9.1. 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.

If you do want to edit XML by hand, use a good schema aware editor that provides nice hierarchical views of the XML, ideally visually (commercial tools like XMLSpy, Oxygen etc are good, but cost money, but then so do headache tablets).

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 drools 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).

5.9.2. The XML format

A full W3C standards (XMLSchema) compliant XSD is provided that describes the XML language, which will not be repeated here verbatim. A summary of the language follows.

```
<?xml version="1.0" encoding="UTF-8"?>
<package name="com.sample"
  xmlns="http://drools.org/drools-4.0"
  xmlns:xs="http://www.w3.org/2001/XMLSchema-instance"
  xs:schemaLocation="http://drools.org/drools-4.0 drools-4.0.xsd">

  <import name="java.util.HashMap" />
  <import name="org.drools.*" />

  <global identifier="x" type="com.sample.X" />
  <global identifier="yada" type="com.sample.Yada" />

  <function return-type="void" name="myFunc">
    <parameter identifier="foo" type="Bar" />
    <parameter identifier="bada" type="Bing" />
    <body>System.out.println("hello world");</body>
  </function>

  <rule name="simple_rule">
    <rule-attribute name="salience" value="10" />
    <rule-attribute name="no-loop" value="true" />
    <rule-attribute name="agenda-group" value="agenda-group" />
    <rule-attribute name="activation-group" value="activation-group" />

    <lhs>
      <pattern identifier="foo2" object-type="Bar" >
        <or-constraint-connective>
          <and-constraint-connective>
            <field-constraint field-name="a">
              <or-restriction-connective>
                <and-restriction-connective>
                  <literal-restriction evaluator=">" value="60" />
                  <literal-restriction evaluator="<" value="70" />
                </and-restriction-connective>
                <and-restriction-connective>
                  <literal-restriction evaluator="<" value="50" />
                  <literal-restriction evaluator=">" value="55" />
                </and-restriction-connective>
              </or-restriction-connective>
            </field-constraint>
            <field-constraint field-name="a3">
              <literal-restriction evaluator="==" value="black" />
            </field-constraint>
          </and-constraint-connective>
          <and-constraint-connective>
            <field-constraint field-name="a">
              <literal-restriction evaluator="==" value="40" />
            </field-constraint>

            <field-constraint field-name="a3">
              <literal-restriction evaluator="==" value="pink" />
            </field-constraint>
          </and-constraint-connective>

          <and-constraint-connective>
            <field-constraint field-name="a">
              <literal-restriction evaluator="==" value="12"/>
            </field-constraint>

            <field-constraint field-name="a3">
              <or-restriction-connective>
```

```
            <literal-restriction evaluator="==" value="yellow"/>
            <literal-restriction evaluator="==" value="blue" />
          </or-restriction-connective>
        </field-constraint>
      </and-constraint-connective>
    </or-constraint-connective>
  </pattern>

  <not>
    <pattern object-type="Person">
      <field-constraint field-name="likes">
        <variable-restriction evaluator="==" identifier="type"/>
      </field-constraint>
    </pattern>

    <exists>
      <pattern object-type="Person">
        <field-constraint field-name="likes">
          <variable-restriction evaluator="==" identifier="type"/>
        </field-constraint>
      </pattern>
    </exists>
  </not>

  <or-conditional-element>
    <pattern identifier="foo3" object-type="Bar" >
      <field-constraint field-name="a">
        <or-restriction-connective>
          <literal-restriction evaluator="==" value="3" />
          <literal-restriction evaluator="==" value="4" />
        </or-restriction-connective>
      </field-constraint>
      <field-constraint field-name="a3">
        <literal-restriction evaluator="==" value="hello" />
      </field-constraint>
      <field-constraint field-name="a4">
        <literal-restriction evaluator="==" value="null" />
      </field-constraint>
    </pattern>

    <pattern identifier="foo4" object-type="Bar" >
      <field-binding field-name="a" identifier="a4" />
      <field-constraint field-name="a">
        <literal-restriction evaluator="!=" value="4" />
        <literal-restriction evaluator="!=" value="5" />
      </field-constraint>
    </pattern>
  </or-conditional-element>

  <pattern identifier="foo5" object-type="Bar" >
    <field-constraint field-name="b">
      <or-restriction-connective>
        <return-value-restriction evaluator="==" >
          a4 + 1
        </return-value-restriction>
        <variable-restriction evaluator=">" identifier="a4" />
        <qualified-identifier-restriction evaluator="==">
          org.drools.Bar.BAR_ENUM_VALUE
        </qualified-identifier-restriction>
      </or-restriction-connective>
    </field-constraint>
  </pattern>

  <pattern identifier="foo6" object-type="Bar" >
    <field-binding field-name="a" identifier="a4" />
    <field-constraint field-name="b">
      <literal-restriction evaluator="==" value="6" />
```

```
        </field-constraint>
      </pattern>
    </lhs>
    <rhs>
      if ( a == b ) {
      assert( foo3 );
      } else {
      retract( foo4 );
      }
      System.out.println( a4 );
    </rhs>
  </rule>
</package>
```

Referring to the above example: Notice the key parts, the declaration for the Drools 4, schema, imports, globals, functions, and the rules. Most of the elements are self explanatory if you have some understanding of the Drools 4 features.

Imports: import the types you wish to use in the rule.

Globals: These are global objects that can be referred to in the rules.

Functions: this is a declaration of functions to be used in the rules. You have to specify return types, a unique name and parameters, in the body goes a snippet of code.

Rule: see below.

Example 5.53. 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>
```

Referring to the above rule detail:

The rule has a 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, certainly more so then past versions.

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 conditional elements 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 non existence of a fact meeting its 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 then then 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.

5.9.3. Legacy Drools 2.x XML rule format

The Drools 2.x legacy XML format is no longer supported by Drools XML parser

5.9.4. Automatic transforming between formats (XML and DRL)

Drools 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.

Deployment and Testing

6.1. Deployment options

Once you have rules integrated in your application (or ideally before) you will need to plan how to deploy rules along with your application. Typically rules are used to allow changes to application business logic without re-deploying the whole application. This means that the rules must be provided to the application as data, not as part of the application (eg embedded in the classpath).

As every organization is subtly different, and different deployment patterns will be needed. Many organizations have (or should have) configuration management processes for changes to production systems. It is best to think of rules as "data" rather then software in that regard. However, as rules can contain a considerable amount of powerful logic, proper procedures should be used in testing and verifying rule changes, and approving changes before exposing them to the world. If you need to "roll your own" deployment, or have specific deployment needs, the information is provided in this chapter for your reference, but for the most part, people should be able to deploy either as the agent, or in the classpath.

6.1.1. Deployment using drl source

In some cases people may wish to deploy drl source. In that case all the drools-compiler dependencies will need to be on the classpath for your application. You can then load drl from file, classpath, or a database (for example) and compile as needed. The trick, as always, is knowing when rules change (this is also called "in process" deployment as described below).

6.1.2. Deploying rules in your classpath

If you have rules which do not change separate to your application, you can put packaged into your classpath. This can be done either as source (in which case the drl can be compiled, and the rulebase cached the first time it is needed) or else you can pre-compile packages, and just include the binary packages in the classpath.

Keep in mind with this approach to make a rule change, you will both need to deploy your app (and if its a server - restart the application).

6.1.3. Deployable objects, RuleBase, Package etc.

In the simplest possible scenario, you would compile and construct a rulebase inside your application (from drl source), and then cache that rulebase. That rulebase can be shared across threads, spawning new working memories to process transactions (working memories are then discarded). This is essentially the stateless mode. To update the rulebase, a new rulebase is loaded, and then swapped out with the cached rulebase (any existing threads that happen to be using the old rulebase will continue to use it until they are finished, in which case it will eventually be garbage collected).

There are many more sophisticated approaches to the above - Drools rule engine is very dynamic, meaning pretty much all the components can be swapped out on the fly (rules, packages) even when there are *existing* working memories in use. For instance rules can be retracted from a rulebase which has many in-use working memories - the RETE network will then be adjusted to remove that rule without having to assert all the facts again. Long running working memories are useful for complex applications where the rule engine builds up knowledge over time to assist with decision making for instance - it is in these cases that the dynamic-ness of the engine can really shine.

6.1.3.1. DRL and PackageDescr

One option is to deploy the rules in source form. This leaves the runtime engine (which must include the compiler components) to compile the rules, and build the rule base. A similar approach is to deploy the "PackageDescr" object, which means that the rules are pre-parsed (for syntactic errors) but not compiled into the binary form. Use the PackageBuilder class to achieve this. You can of course use the XML form for the rules if needed.

```
PackageDescr, PackageBuilder, RuleBaseLoader
```

6.1.3.2. Package

This option is the most flexible. In this case, Packages are built from DRL source using PackageBuilder - but it is the binary Package objects that are actually deployed. Packages can be merged together. That means a package containing perhaps a single new rule, or a change to an existing rule, can be built on its own, and then merged in with an existing package in an existing RuleBase. The rulebase can then notify existing working memories that a new rule exists (as the RuleBase keeps "weak" links back to the Working Memory instances that it spawned). The rulebase keeps a list of Packages, and to merge into a package, you will need to know which package you need to merge into (as obviously, only rules from the same package name can be merged together).

Package objects themselves are serializable, hence they can be sent over a network, or bound to JNDI, Session etc.

```
PackageBuilder, RuleBase, org.drools.rule.Package
```

6.1.3.3. RuleBase

Compiled Packages are added to rulebases. RuleBases are serializable, so they can be a binary deployment unit themselves. This can be a useful option for when rulebases are updated as a whole - for short lived working memories. If existing working memories need to have rules changed on the fly, then it is best to deploy Package objects. Also beware that rulebases take more processing effort to serialize (may be an issue for some large rulebases).

```
RuleBase, RuleBaseLoader
```

6.1.3.4. Serializing

Practically all of the rulebase related objects in Drools are serializable. For a working memory to be serializable, all of your objects must of course be serializable. So it is always possible to deploy remotely, and "bind" rule assets to JNDI as a means of using them in a container environment.

Please note that when using package builder, you may want to check the hasError() flag before continuing deploying your rules (if there are errors, you can get them from the package builder - rather then letting it fail later on when you try to deploy).

6.1.4. Deployment patterns

6.1.4.1. In process rule building

In this case, rules are provided to the runtime system in source form. The runtime system contains the drools-compiler component to build the rules. This is the simplest approach.

6.1.4.2. Out of process rule building

In this case, rules are build into their binary process outside of the runtime system (for example in a deployment server). The chief advantage of deploying from an outside process is that the runtime system can have minimal dependencies (just one jar). It also means that any errors to do with compiling are well contained and and known before deployment to the running system is attempted.

Use the PackageBuilder class out of process, and then use getPackage() to get the Package object. You can then (for example) serialize the Package object to a file (using standard java serialization). The runtime system, which only needs drools-core, can then load the file using RuleBaseFactory.newRuleBase().addPackage(deserialized package object).

6.1.4.3. Some deployment scenarios

This section contains some suggested deployment scenarios, of course you can use a variety of technologies as alternatives to the ones in the diagram.

6.1.4.3.1. Pull style

This pattern is what is used by the RuleAgent, by default.

In this scenario, rules are pulled from the rule repository into the runtime system. The repository can be as simple as a file system, or a database. The trigger to pull the rules could be a timed task (to check for changes) or a request to the runtime system (perhaps via a JMX interface). This is possibly the more common scenario.

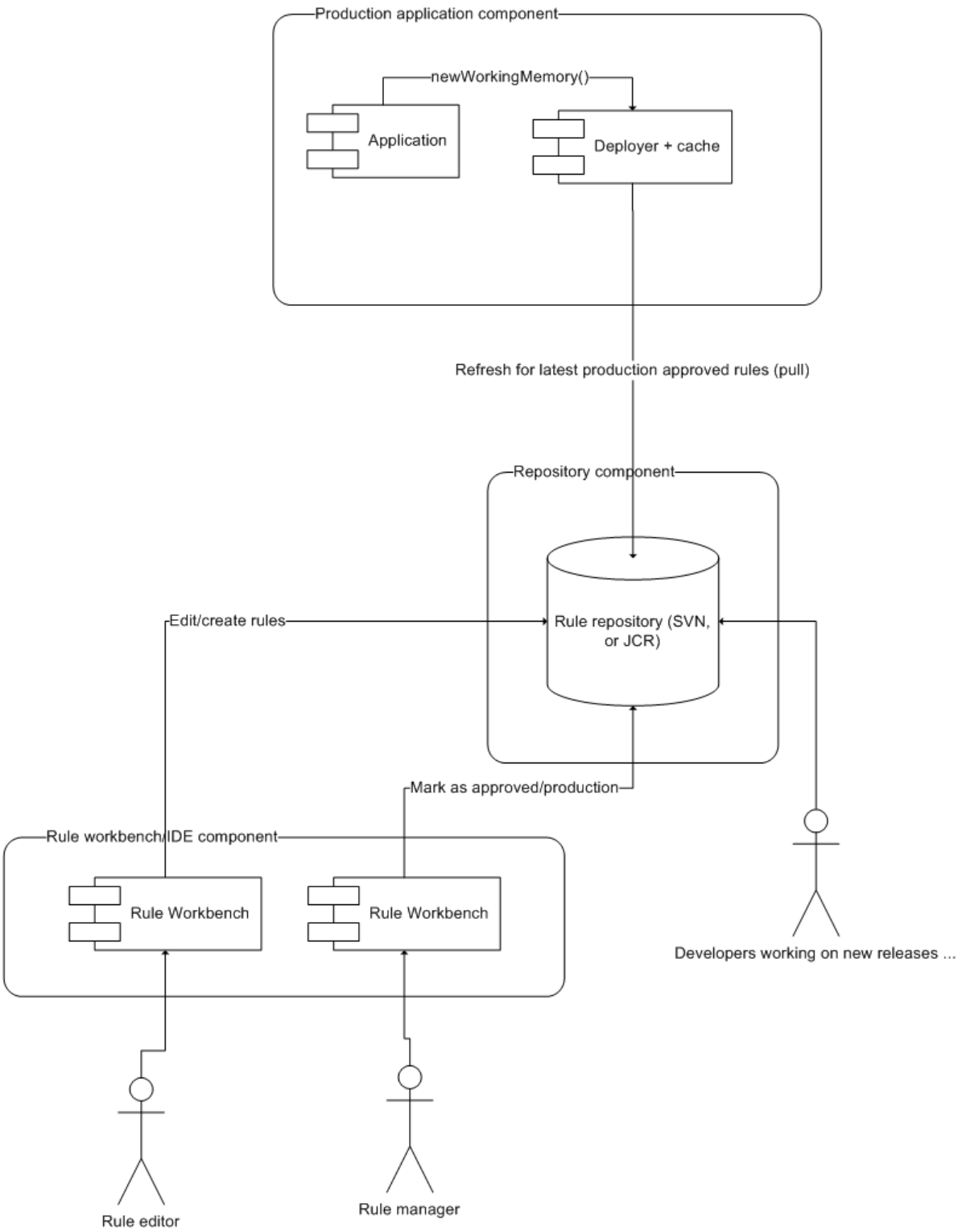


Figure 6.1. Pull Deployment Pattern

6.1.4.3.2. Push style

In this scenario, the rule deployment process/repository "pushes" rules into the runtime system (either in source or binary form, as described above). This gives more control as to when the new rules take effect.

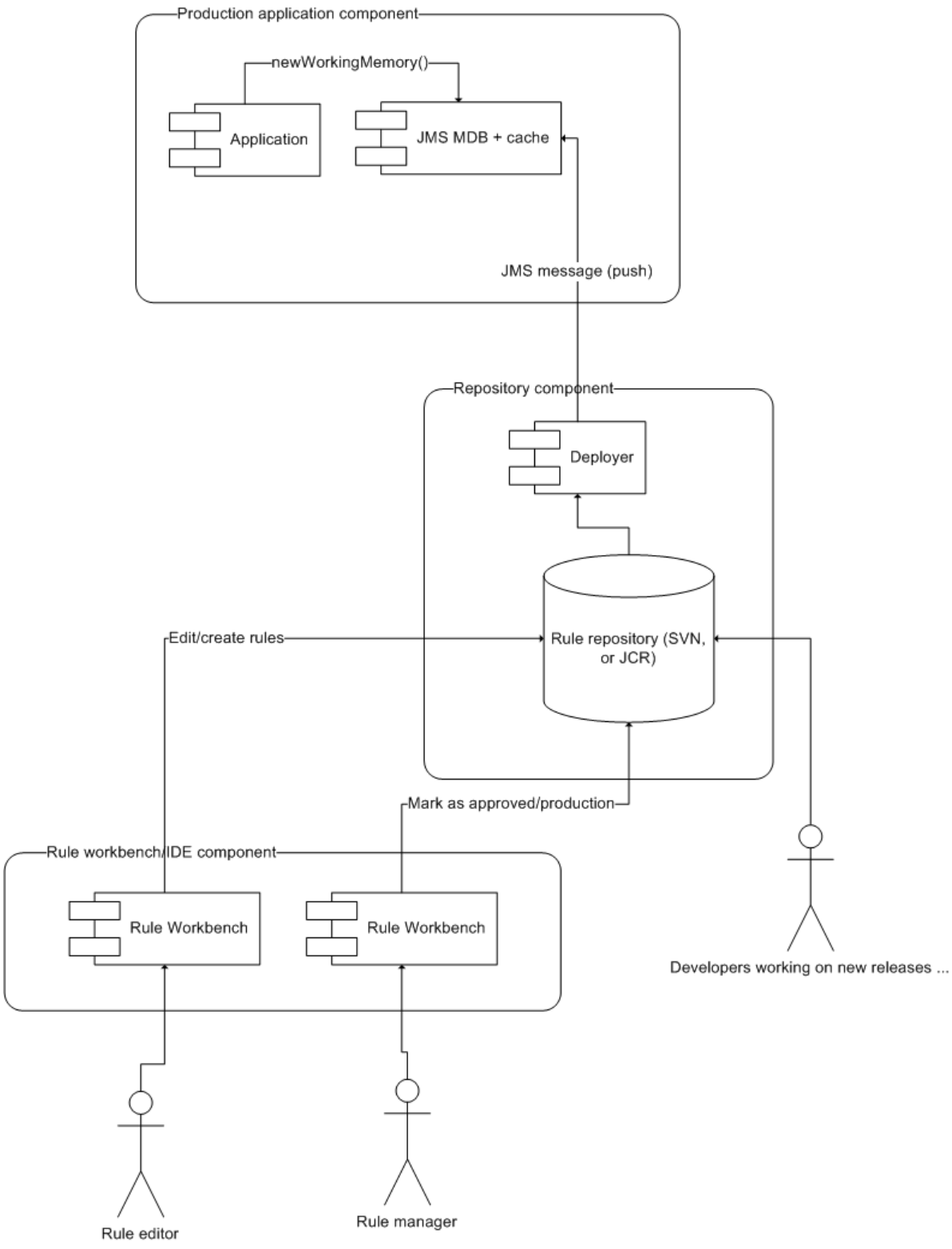


Figure 6.2. Push Deployment Pattern

6.1.5. Web Services

A possible deployment pattern for rules are to expose the rules as a web service. There a many ways to achieve this, but possibly the simplest way at present do achieve it is to use an interface-first process: Define the "facts" classes/templates that the rules will use in terms of XML Schema - and then use binding technologies to generate binding objects for the rules to actually operate against.

A reverse possibility is to use a XSD/WSDL generator to generate XML bindings for classes that are hand built (which the rules work against). It is expected in a future version there will be an automated tool to expose rules as web services (and possibly use XSDs as facts for the rules to operate on).

6.1.6. Future considerations

A future release of Drools will contain a rule repository (server) component that will directly support the above patterns, and more.

6.2. Testing

In recent years, practices such as Test Driven Development have become increasingly mainstream, as the value and quality that these techniques bring to software development has been realized. In a sense, rules are code (although at a high level), and a lot of the same principles apply.

You can provide tests as a means to specify rule behavior before rules are even written. Further to this, tests are even more important in environments where rules change frequently. Tests can provide a baseline of confidence that the rule changes are consistent with what is specified in the tests. Of course, the rules may change in such a way as the tests are now wrong (or perhaps new tests need to be written to cover the new rule behavior). As in TDD practices, tests should be run often, and in a rule driven environment, this means that they should be run every time the rules change (even though the software may be static).

6.2.1. Testing frameworks

For developers, clearly JUnit (or TestNG) are popular tools for testing code, and these can also apply to rules. Keep in mind that rule changes may happen out of sync with code changes, so you should be prepared to keep these unit tests up to date with rules (may not be possible in all environments). Also, the best idea is to target testing some core features of the rule sets that are not as likely to change over time.

Obviously, for rule tests, other non source code driven frameworks would be preferable to test rules in some environments. The following section outlines a rule testing component add on.

6.2.2. FIT for Rules - a rule testing framework

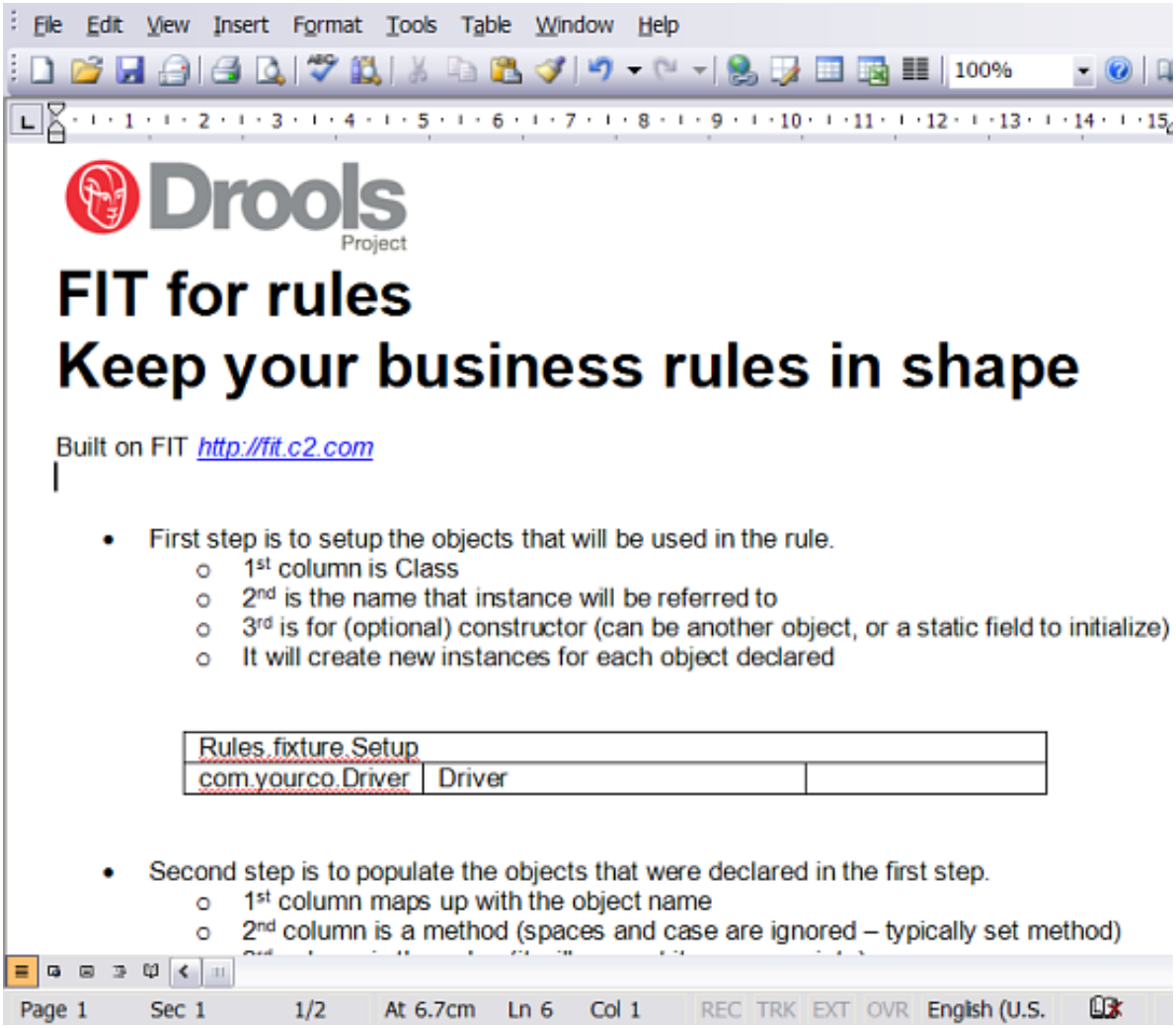
As a separate add-on, there is a testing framework available that is built on FIT (Framework for Integrated Testing). This allows rule test suites (functional) to be capture in Word documents, or Excel spreadsheets (in fact any tool that can save as HTML). It utilizes a tabular layout to capture input data, and make assertions over the rules of a rulesets execution for the given facts. As the tests are stored in documents, the scenarios and requirements can be (optionally) kept in the same documents, providing a single point of truth for rule behavior.

Also, as the test documents are not code, they can be updated frequently, and kept with the rules, used to validate rule changes etc. As the input format is fairly simple to people familiar with the domain of the rules, it also facilitates "scenario testing" where different scenarios can be tried out with the rules - all external to the application that the rules are used in. These scenarios can then be kept as tests to increase confidence that a rule change is consistent with the users understanding.

This testing framework is built on FIT and JSR-94, and is kept as a separate project to JBoss Rules. Due to it being built on FIT, it requires a different license (but is still open source). You can download and read more about this tool from this web page: *Fit for rules* [1] http://fit-for-rules.sourceforge.net/

[1] http://fit-for-rules.sourceforge.net/

The following screen captures show the fit for rules framework in action.



Using Fit for rules, you capture test data, pass it to the rule engine and then verify the results (with documentation woven in with the test). It is expected that in future, the Drools Server tools will provide a similar integrated framework for testing (green means good ! red means a failure - with the expected values placed in the cell). Refer to http://fit.c2.com for more information on the FIT framework itself.

Rules.fixture.Results		
Driver	Get Name	Bob
Driver	Is Approved	True
Driver	Get Age	42

You can optionally reset the domain objects, in case you want to do multiple clean test runs in the one test document

Rules.fixture.Clear

With FIT, you can get a summary printed in the output report by using the following table

fit.Summary	
counts	32 right, 0 wrong, 22 ignored, 0 exceptions
input file	C:\Projects\fit-for-rules-new\doc\fit-for-rules.html
input update	Thu May 11 20:23:13 EST 2006
output file	C:\Projects\fit-for-rules-new\doc\test-result.html
run date	Thu May 11 20:23:51 EST 2006
run elapsed time	0:01.28

More information and downloads from *Here* [2]

[2] http://fit-for-rules.sourceforge.net/

The Java Rule Engine API

7.1. Introduction

Drools provides an implementation of the Java Rule Engine API (known as JSR94), which allows for support of multiple rule engines from a single API. JSR94 does not deal in anyway with the rule language itself. W3C is working on the *Rule Interchange Format (RIF)*[1] and the OMG has started to work on a standard based on *RuleML*[2], recently Haley Systems has also proposed a rule language standard called RML.

It should be remembered that the JSR94 standard represents the "least common denominator" in features across rule engines - this means there is less functionality in the JSR94 api than in the standard Drools api. So by using JSR94 you are restricting yourself in taking advantage of using the full capabilities of the Drools Rule Engine. It is necessary to expose further functionality, like globals and support for drl, dsl and xml via properties maps due to the very basic feature set of JSR94 - this introduces non portable functionality. Further to this, as JSR94 does not provide a rule language, you are only solving a small fraction of the complexity of switching rule engines with very little gain. So while we support JSR94, for those that insist on using it, we strongly recommend you program against the Drools API.

7.2. How To Use

There are two parts to working with JSR94. The first part is the administrative api that deals with building and register RuleExecutionSets, the second part is runtime session execution of those RuleExecutionSets.

7.2.1. Building and Registering RuleExecutionSets

The RuleServiceProviderManager manages the registration and retrieval of RuleServiceProviders. The Drools RuleServiceProvider implementation is automatically registered via a static block when the class is loaded using Class.forName; in much the same way as JDBC drivers.

Example 7.1. 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, making it possible to register a RuleExecutionSet that can then be retrieved via the RuleRuntime.

First you need to create a RuleExecutionSet before it can be registered; RuleAdministrator provides factory methods to return an empty LocalRuleExecutionSetProvider or RuleExecutionSetProvider.

[1] http://www.w3.org/TR/2006/WD-rif-ucr-20060323/
[2] http://ruleml.org/

The LocalRuleExecutionSetProvider should be used to load a RuleExecutionSets from local sources that are not serializable, like Streams. The RuleExecutionSetProvider can be used to load RuleExecutionSets from serializable sources, like DOM Elements or Packages. Both the "ruleAdministrator.getLocalRuleExecutionSetProvider(null);" and the "ruleAdministrator.getRuleExecutionSetProvider(null);" take null as a parameter, as the properties map for these methods is not currently used.

Example 7.2. Registering a LocalRuleExecutionSet with the RuleAdministration API

```
// 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 the RuleExecutionSet for the drl
RuleExecutionSet ruleExecutionSet =
 ruleExecutionSetProvider.createRuleExecutionSet( drlReader, null );
```

"ruleExecutionSetProvider.createRuleExecutionSet(reader, null)" in the above example takes a null parameter for the properties map; however it can actually be used to provide configuration for the incoming source. When null is passed the default is used to load the input as a drl. Allowed keys for a map are "source" and "dsl". "source" takes "drl" or "xml" as its value; set "source" to "drl" to load a drl or to "xml" to load an xml source; xml will ignore any "dsl" key/value settings. The "dsl" key can take a Reader or a String (the contents of the dsl) as a value.

Example 7.3. Specifying a DSL 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 );
```

When registering a RuleExecutionSet you must specify the name, to be used for its retrieval. There is also a field to pass properties, this is currently unused so just pass null.

Example 7.4. Register the RuleExecutionSet

```
// Register the RuleExecutionSet with the RuleAdministrator
```

```
String uri = ruleExectionSet.getName();
ruleAdministrator.registerRuleExecutionSet(uri, ruleExecutionSet, null);
```

7.2.2. Using Stateful and Stateless RuleSessions

The Runtime, obtained from the RuleServiceProvider, is used to create stateful and stateless rule engine sessions.

Example 7.5. Getting the RuleRuntime

```
RuleRuntime ruleRuntime = ruleServiceProvider.getRuleRuntime();
```

To create a rule session you must use one of the two RuleRuntime public constants - "RuleRuntime.STATEFUL_SESSION_TYPE" and "RuleRuntime.STATELESS_SESSION_TYPE" along with the uri to the RuleExecutionSet you wish to instantiate a RuleSession for. The properties map can be null, or it can be used to specify globals, as shown in the next section. The createRuleSession(....) method returns a RuleSession instance which must then be cast to StatefulRuleSession or StatelessRuleSession.

Example 7.6. 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; you can only call executeRules(List list) passing a list of objects, and an optional filter, the resulting objects are then returned.

Example 7.7. Stateless

```
(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 );
```

7.2.2.1. Globals

It is possible to support globals with JSR94, in a none portable manner, by using the properties map passed to the RuleSession factory method. Globals must be defined in the drl or xml file first, otherwise an Exception will be thrown. the key represents the identifier declared in the drl or xml and the value is the instance you wish to be used in the execution. In the following example the results are collected in an java.util.List which is used as global:

```
java.util.List globalList = new java.util.ArrayList( );
java.util.Map map = new java.util.HashMap( );
map.put( "list", globalList );
```

```
//Open a stateless Session StatelessRuleSession srs = (StatelessRuleSession)
 runtime.createRuleSession( "SistersRules", map, RuleRuntime.STATELESS_SESSION_TYPE );
...
// Persons added to List
// call executeRules( ) giving a List of Objects as parameter
// There are rules which will put Objects in the List
// fetch the list from the map
List list = (java.util.List) map.get("list");
```

Do not forget to declare the global "list" in your DRL:

```
package SistersRules;
import org.drools.jsr94.rules.Person;
global java.util.List list
rule FindSisters
when
$person1 : Person ( $name1:name )
$person2 : Person ( $name2:name )
eval( $person1.hasSister($person2) )
then
list.add($person1.getName() + " and " + $person2.getName() +" are sisters");
assert( $person1.getName() + " and " + $person2.getName() +" are sisters");
end
```

7.3. References

If you need more information on JSR 94, please refer to the following references

1. Official JCP Specification for Java Rule Engine API (JSR 94)
   - *http://www.jcp.org/en/jsr/detail?id=94*
2. The Logic From The Bottom Line: An Introduction to The Drools Project. By N. Alex Rupp, published on TheServiceSide.com in 2004
   - *http://www.theserverside.com/articles/article.tss?l=Drools*
3. Getting Started With the Java Rule Engine API (JSR 94): Toward Rule-Based Applications. By Dr. Qusay H. Mahmoud, published on Sun Developer Network in 2005
   - *http://java.sun.com/developer/technicalArticles/J2SE/JavaRule.html*
4. Jess and the javax.rules API. By Ernest Friedman-Hill, published on TheServerSide.com in 2003
   - *http://www.theserverside.com/articles/article.tss?l=Jess*

Appendix A. Revision History

Revision 1.5 **Mon Mar 21 2011** **David Le Sage** *dlesage@redhat.com*

Updated for 4.3.CP05 Release

Revision 1.4 **Tue Apr 27 2010** **David Le Sage** *dlesage@redhat.com*

Updated for 4.3.CP04

Revision 1.3 **Tue Apr 20 2010** **David Le Sage** *dlesage@redhat.com*

Updated for SOA 4.3.CP03

Revision 1.2 **Mon July 13 2009** **Darrin Mison** *dmison@redhat.com*

Updated for 4.3.CP02
SOA-1199 - Updated RuleFlow content to clarify what is supported.

Revision 1.1 **Thu Mar 5 2009** **Darrin Mison** *dmison@redhat.com*

Updated for 4.3.CP01
Removed information on non-supported RuleFlow Functionality
Fixed formatting of XML example in XML Rule Language

Revision 1.0 **Tue Sep 2 2008** **Joshua Wulf** *jwulf@redhat.com*

Rebuilt with publican 0.35