# Chapter 1. Acknowledgements

## 1.1. Documentation

We would like to thank the following persons that contributed either directly or indirectly in building the documentation that is presented here. The list is presented in no particular order and the appreciation of both the Drools team and the community is due.

- Peter Samouelian

If you contributed to the project in some way and your name is not listed above, please drop us a note with your name and e-mail and we will gladly add it.

# Chapter 2. The Rule Engine

## 2.1. What is a Rule Engine?

### 2.1.1. Introduction and Background

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

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

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

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

A Production Rule System is Turing complete with a focus on knowledge representation to express propositional and first order logic in a concise, non-ambiguous and declarative manner. The brain of a Production Rules System is an Inference Engine that is able to scale to a large number of rules and facts. The Inference Engine matches facts and data against Production Rules - also called Productions or 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

Drools implements and extends the Rete algorithm, Leaps used to be provided but was retired as it became unmaintained. The Drools Rete implementation is called ReteOO, signifying that Drools has an enhanced and optimized implementation of the Rete algorithm for Object Oriented systems. Other Rete based engines also have marketing terms for their proprietary enhancements to Rete, like RetePlus and Rete III. It is important to understand that names like Rete III are purely marketing where, unlike the original published Rete Algorithm, no details of the implementation are published. This makes questions such as "Does Drools implement Rete III?" nonsensical. The most common enhancements are covered in "Production Matching for Large Learning Systems (Rete/UL)" (1995) by Robert B. Doorenbos.

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

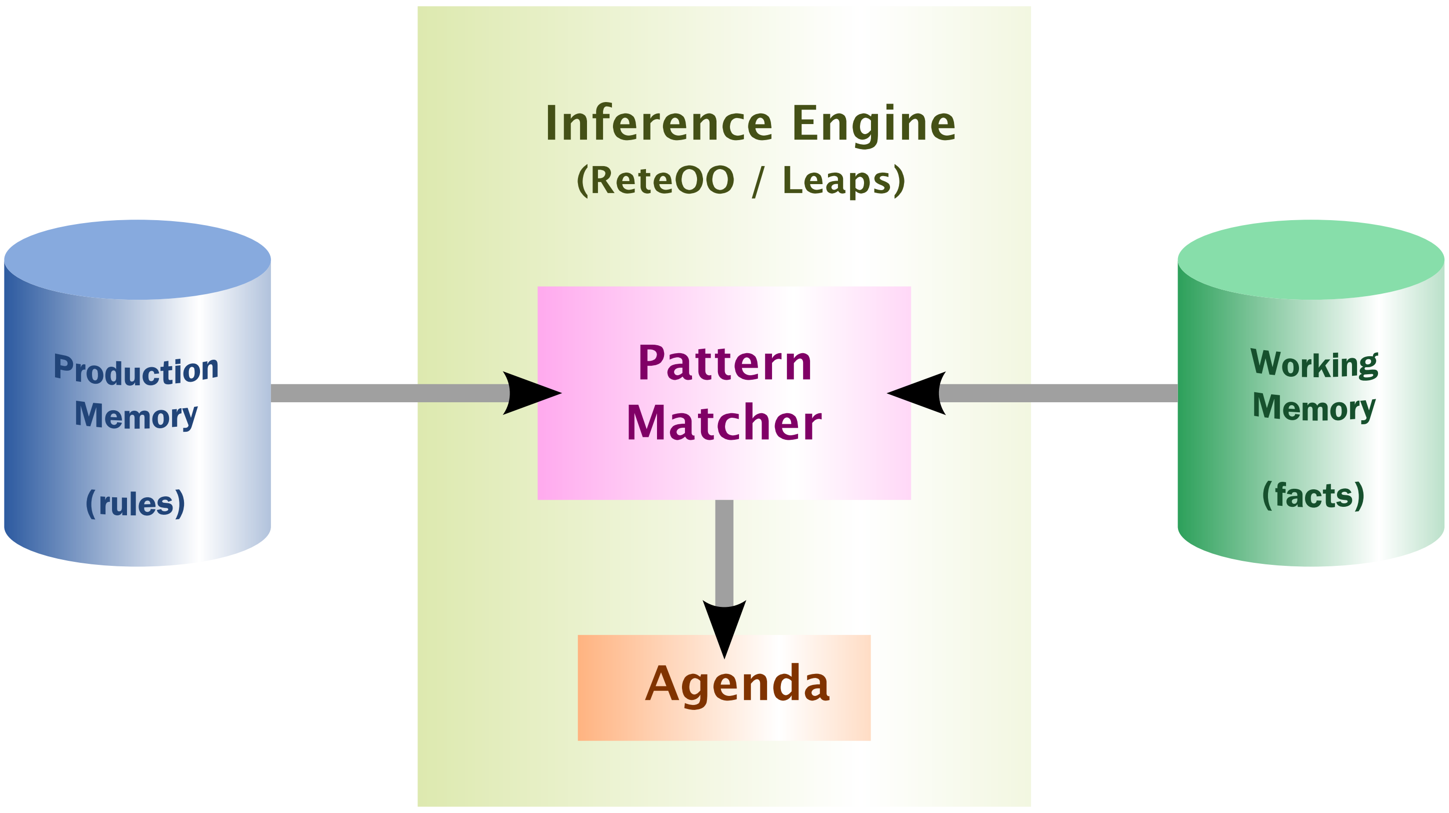
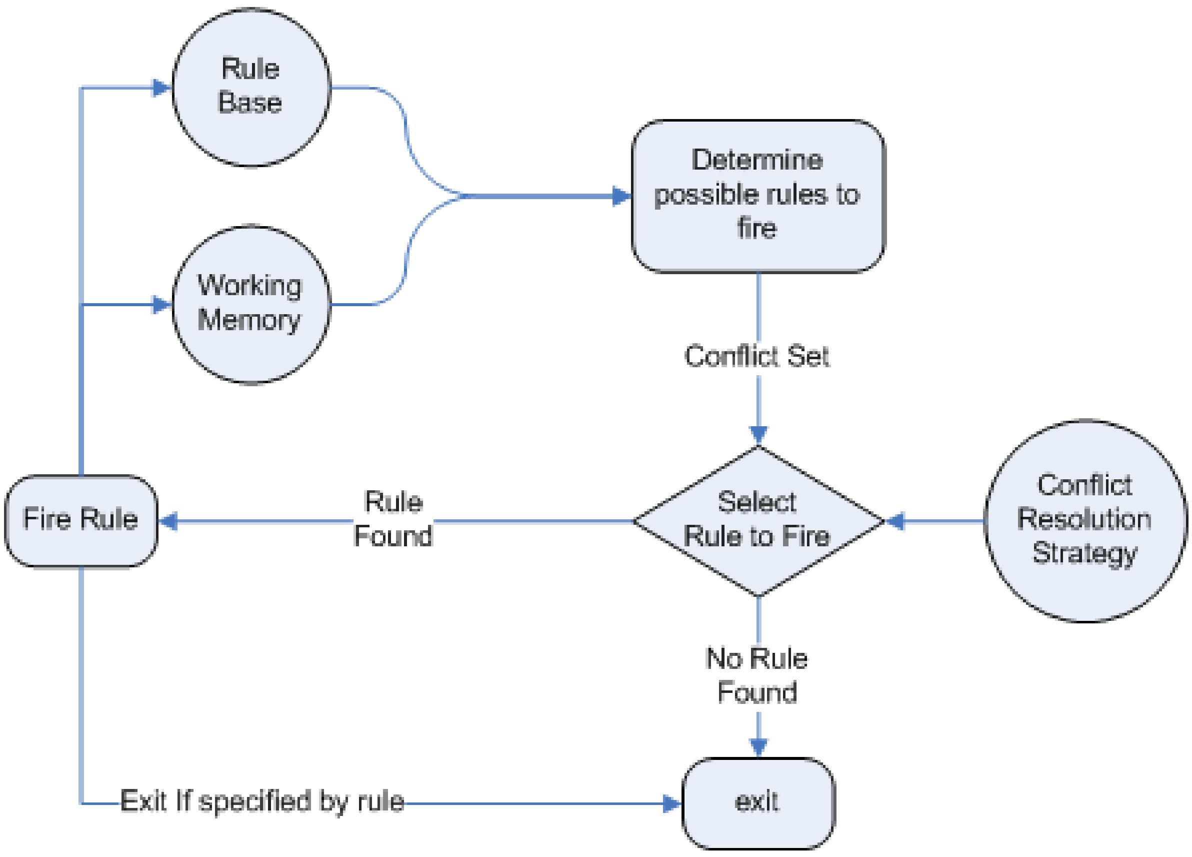


**Figure 2.1. High-level View of a Rule Engine**

A Production Rule System's Inference Engine is stateful and able to enforce truthfulness - called Truth Maintenance. A logical relationship can be declared by actions which means the action's state depends on the inference remaining true; when it is no longer true the logical dependent action is undone. The "Honest Politician" is an example of Truth Maintenance, which always ensures that hope can only exist for a democracy while we have honest politicians.

```
when
    an honest Politician exists
then
    logically assert Hope

when
   Hope exists
then
   print "Hurrah!!! Democracy Lives"

when
   Hope does not exist
then
   print "Democracy is Doomed"
```

There are two methods of execution for a Production Rule Systems - Forward Chaining and Backward Chaining; systems that implement both are called Hybrid Production Rule Systems. Understanding these two modes of operation are key to understanding why a Production Rule System is different and how to get the best from them. Forward chaining is 'data-driven' and thus reactionary - facts are asserted into the working memory which results in one or more rules being concurrently true and scheduled for execution by the Agenda - we start with a fact, it propagates and we end in a conclusion. Drools is a forward chaining engine.



## Figure 2.2. Forward Chaining

Backward chaining is 'goal-driven', meaning that we start with a conclusion which the engine tries to satisfy. If it can't it then searches for conclusions that it can, known as 'sub goals', that will help satisfy some unknown part of the current goal - it continues this process until either the initial conclusion is proven or there are no more sub goals. Prolog is an example of a Backward Chaining engine; Drools will be adding support for Backward Chaining in its next major release.

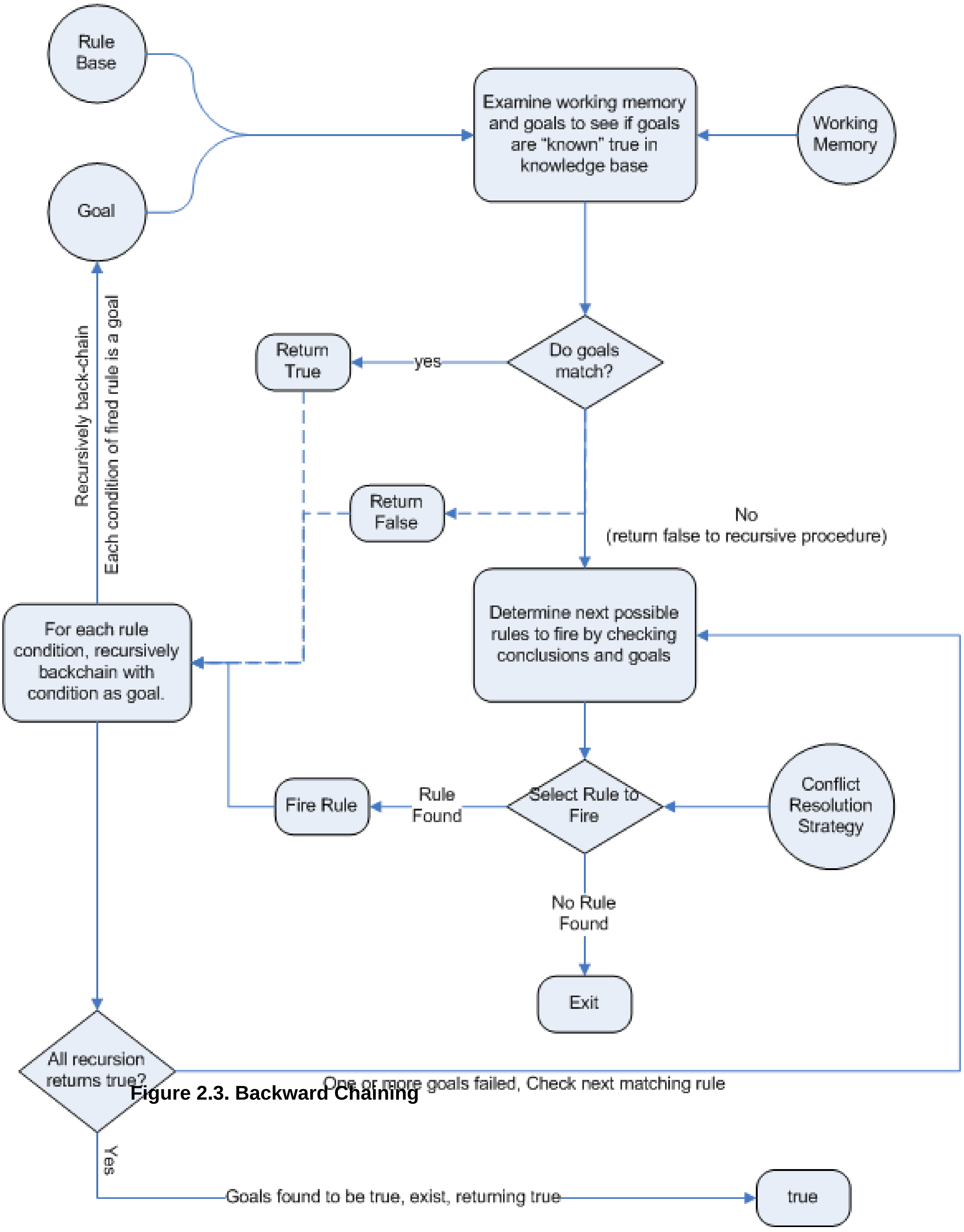

Rule Base
Goal
Examine working memory and goals to see if goals are "known" true in knowledge base
Working Memory
Do goals match?
yes
Return True
Return False
No (return false to recursive procedure)
Recursively back-chain
Each condition of fired rule is a goal
Determine next possible rules to fire by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

**Figure 2.3. Backward Chaining**

## 2.2. Why use a Rule Engine?

Some frequently asked questions:

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".

  The key advantage of this point is that using rules can make it easy to express solutions to difficult problems and consequently have those solutions verified (rules are much easier to read then code).

  Rule systems are capable of solving very, very hard problems, providing an explanation of how the solution was arrived at and why each "decision" along the way was made (not so easy with other of AI systems like neural networks or the human brain - I have no idea why I scratched the side of the car).

- Logic and Data Separation

  Your data is in your domain objects, the logic is in the rules. This is fundamentally breaking the OO coupling of data and logic, which can be an advantage or a disadvantage depending on your point of view. The upshot is that the logic can be much easier to maintain as there are changes in the future, as the logic 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, Leaps algorithm, and its descendants such as Drools' ReteOO (and Leaps), 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 (a knowledgebase) 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 effectively provide an "explanation facility" by being able to log 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 you can set yourself up to write rules that are very close to natural language. They lend themselves to logic that is understandable to, possibly nontechnical, domain experts as they are expressed in their language (as all the program plumbing, the "How", is in the usual code, hidden away).

### 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 fiddle for traditional code.

  The problem may not be complex, but you can't see a non-fragile 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 basically the problem isn't fully understood.

- The logic changes often

  The logic itself may be simple (but doesn't have to be) but the rules change quite often. In many organizations software releases are few and far between and rules can help provide the "agility" that is needed and expected in a reasonably safe way.

- Domain experts (or 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 "soft" nontechnical positions do not have training in formal logic, so be careful and work with them, as by codifying business knowledge in rules, you will often expose 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. It is not a trivial technology, but this document tries to make it easier to understand.

Typically in a modern OO application you would use a rule engine to contain key parts of your business logic (what that means of course depends on the application) - ESPECIALLY the REALLY MESSY parts!. This is an inversion of the OO concept of encapsulating all the logic inside your objects. This is not to say that you throw out OO 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 and/or other messy logic in your code that just doesn't feel right (and you keep coming back to fix it - either because you got it wrong, or the logic/your understanding changes) - 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. Often rules 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 (the options are many, but different organizations have different requirements - often they are out of the control of the application vendors/project teams).

## 2.2.3. When not to use a Rule Engine

To quote a Drools mailing list regular:

> It seems to me that 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. Rules engines are not really intended to handle workflow or process executions nor are workflow engines or process management tools designed to do rules. Use the right tool for the job. Sure, a pair of pliers can be used as a hammering tool in a pinch, but that's not what it's designed for.
>
> —Dave Hamu

As rule engines are dynamic (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 the dynamicness 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 (and conceptually simpler for imperative programmers!).

Many people have also implemented data-driven systems successfully in the past (where there are control tables that store meta-data that changes your applications behavior) - these can work well when the control can remain very limited. However, they can quickly grow out of control if extended too much (such that only the original creators can change the applications behavior) 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* [http://en.wikipedia.org/wiki/Emil_Leon_Post] 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 that can be classified as 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
```

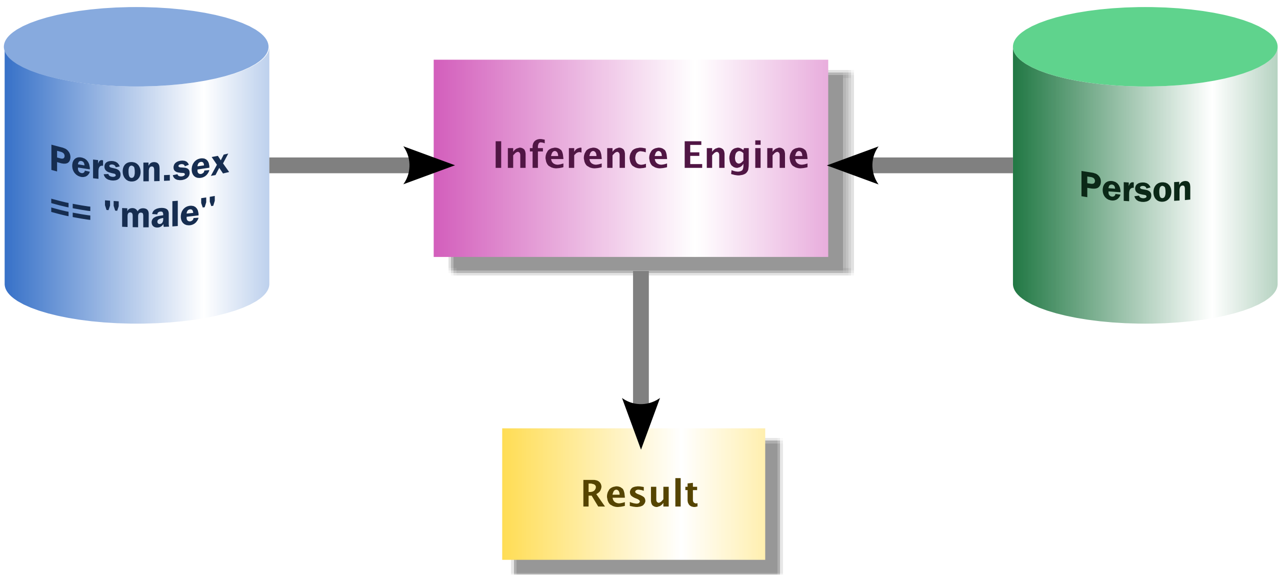
Expressions that evaluate against one or more variables, the facts, are "open statements", in that we cannot determine whether the statement is true until we have a variable instance to evaluate against:

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

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

```
select * from Person where Person.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.



## Figure 2.4. SQL as a simplistic Inference Engine

So 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 - a proposition can be thought as a field constraint. Further to this proposition can be combined with conjunctive and disjunctive connectives, which is the logic theorists way of saying '&&' 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 RETE algorithm was invented by Dr. Charles Forgy and documented in his PhD thesis in 1978-79. A simplified version of the paper was published in 1982 (*http://citeseer.ist.psu.edu/context/505087/0*). The word RETE is latin for "net" meaning network. The RETE algorithm can be broken into 2 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. In Dr. Forgy's 1982 paper, he described 4 basic nodes: root, 1-input, 2-input and terminal.

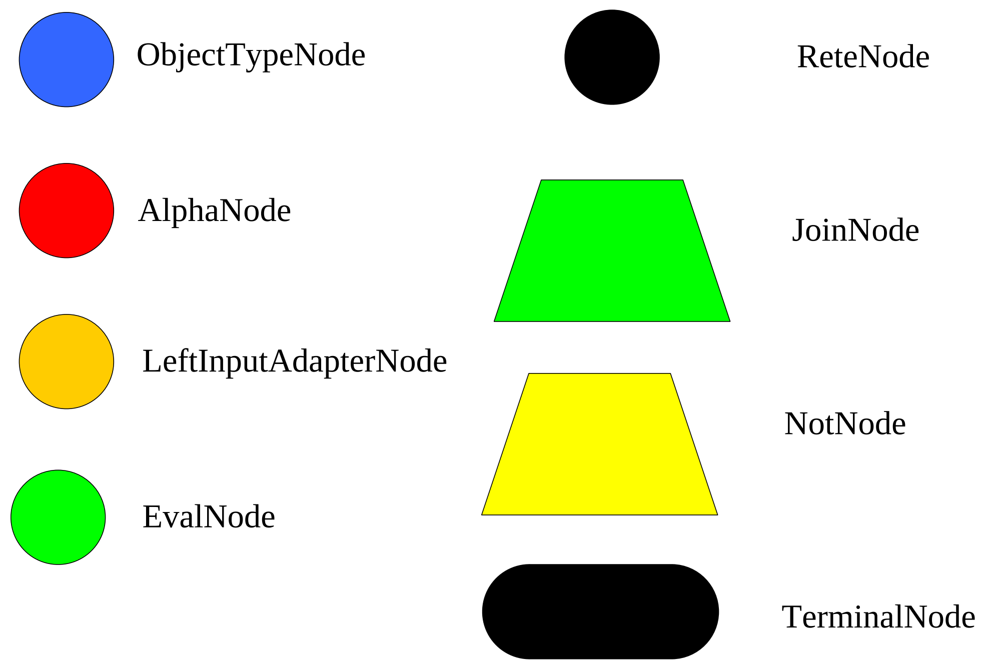


**Figure 2.5. Rete Nodes**

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 ObjectTypeNodes finding valid matches which it caches in the list. This enables Drools to match against any Class type that matches with an `instanceof` check.

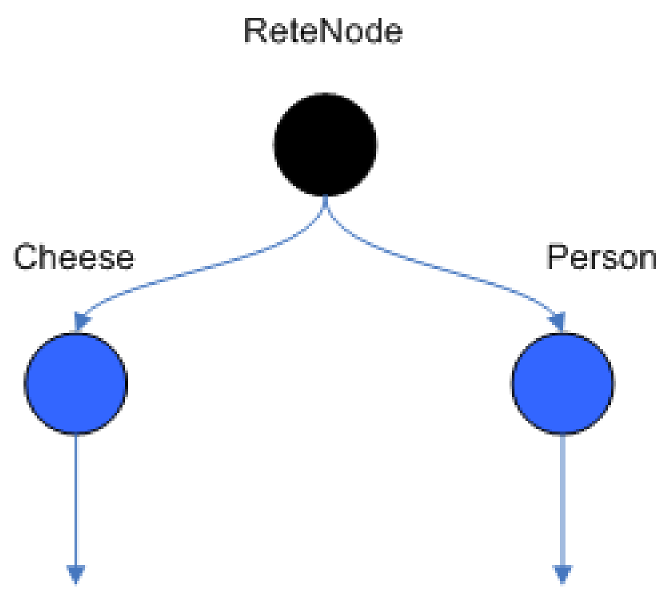


**Figure 2.6. ObjectTypeNodes**

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" ):

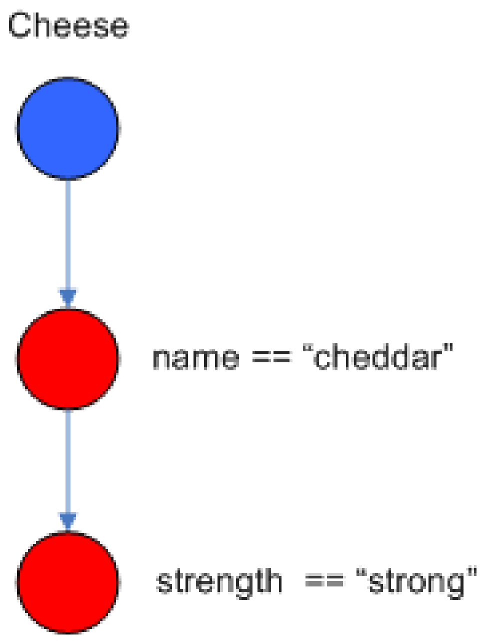
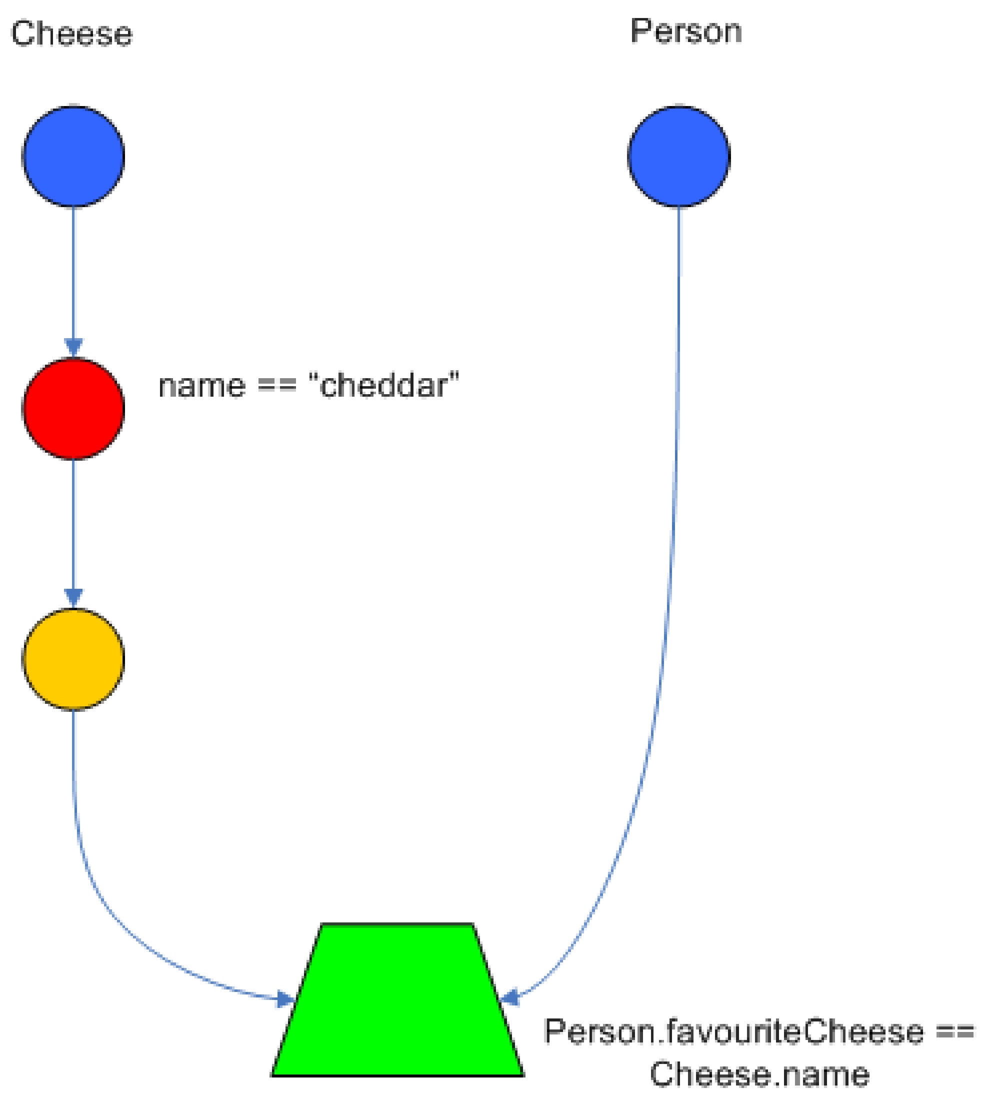


**Figure 2.7. AlphaNodes**

Drools extends Rete by optimizing the propagation from ObjectTypeNode to AlphaNode using hashing. Each time an AlphaNode is added to an ObjectTypeNode it adds the literal value as a key to the HashMap with the AlphaNode as the value. When a new instance enters the ObjectType 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 used 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 Nodes 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 2.8. 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.

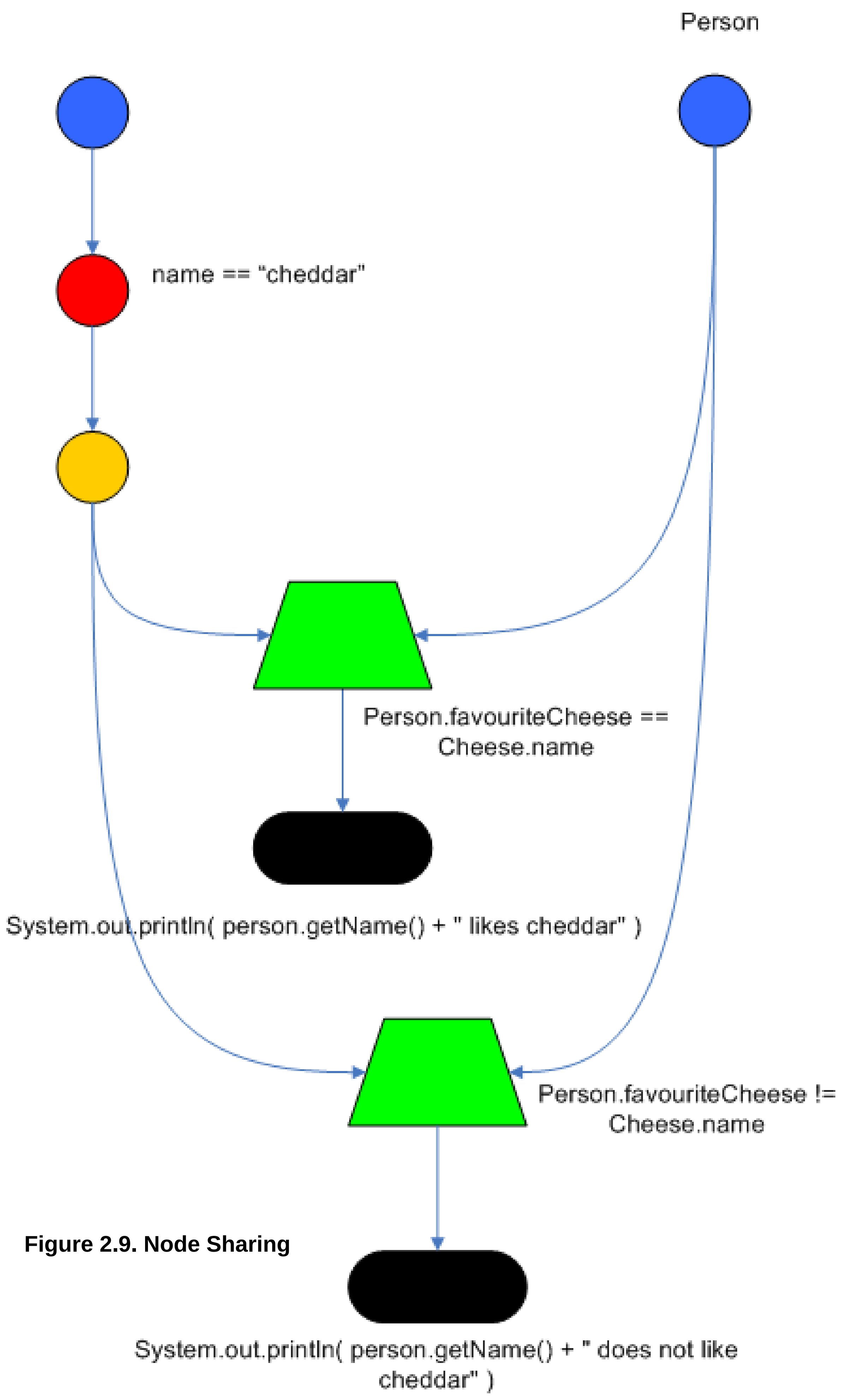

Person
name == "cheddar"
Person.favouriteCheese == Cheese.name
System.out.println( person.getName() + " likes cheddar" )
Person.favouriteCheese != Cheese.name
System.out.println( person.getName() + " does not like cheddar" )

**Figure 2.9. 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, in that it's a serialized object consisting of one or more rules.

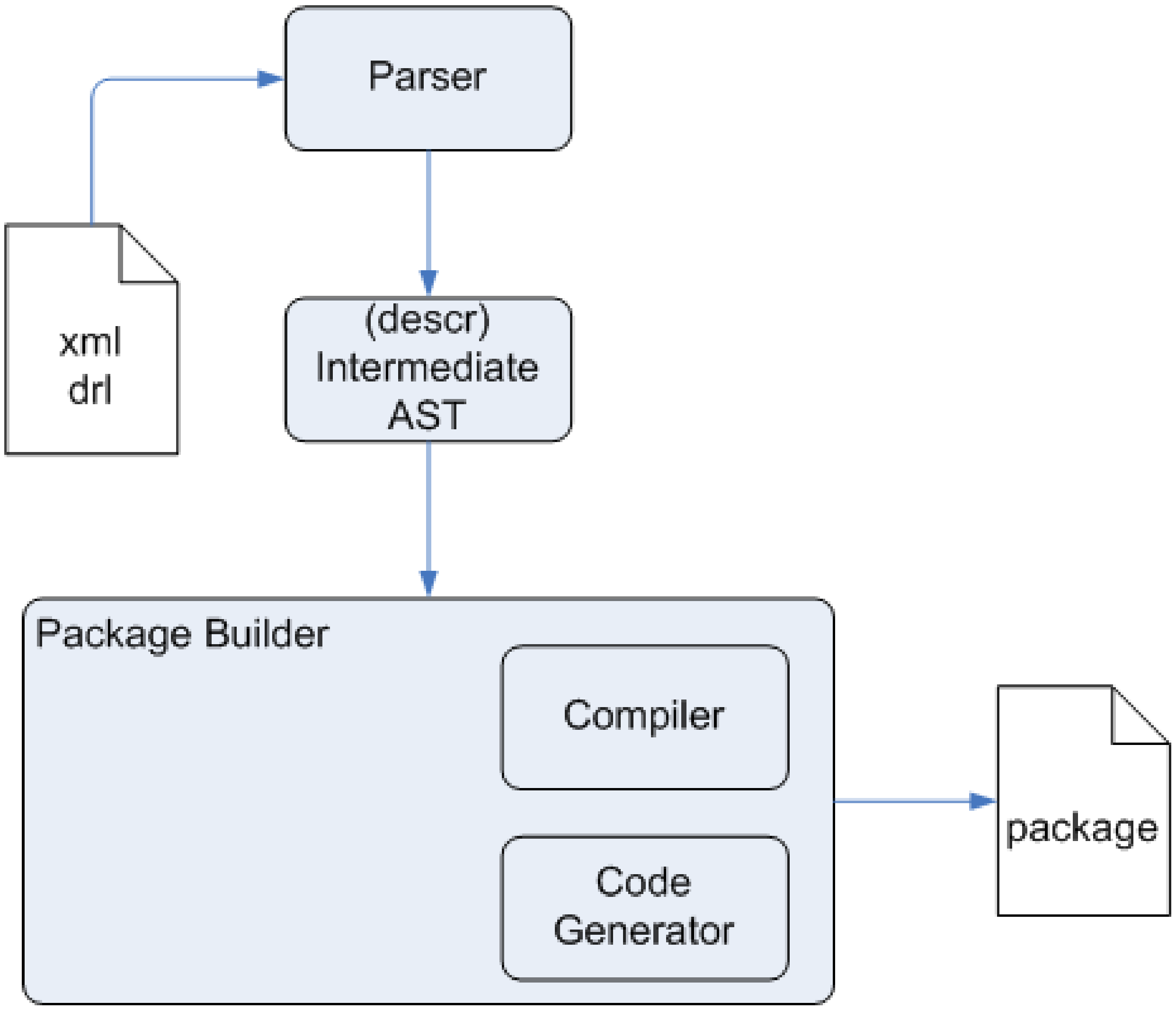


**Figure 2.10. Authoring Components**

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.

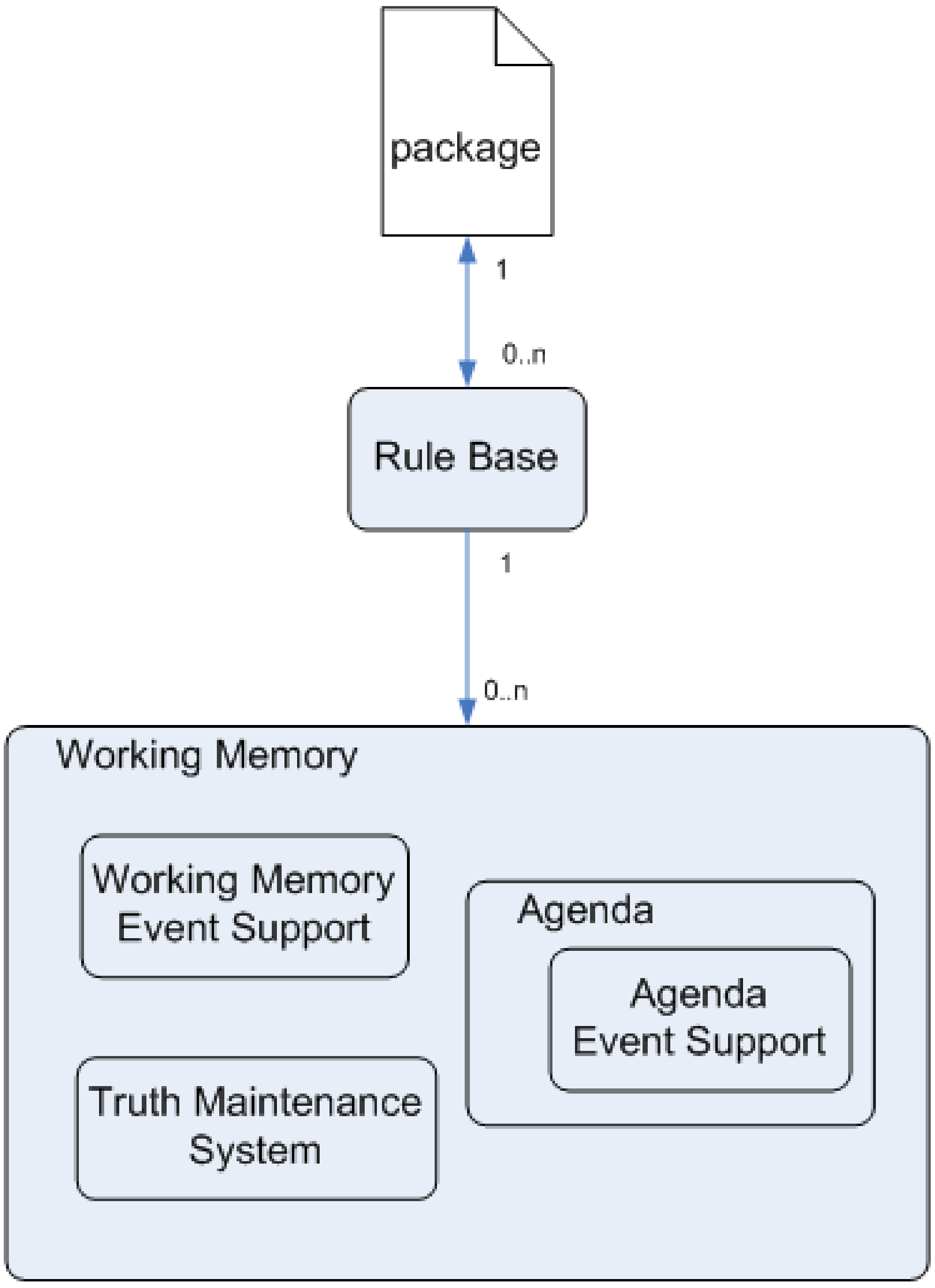


**Figure 2.11. Runtime Components**

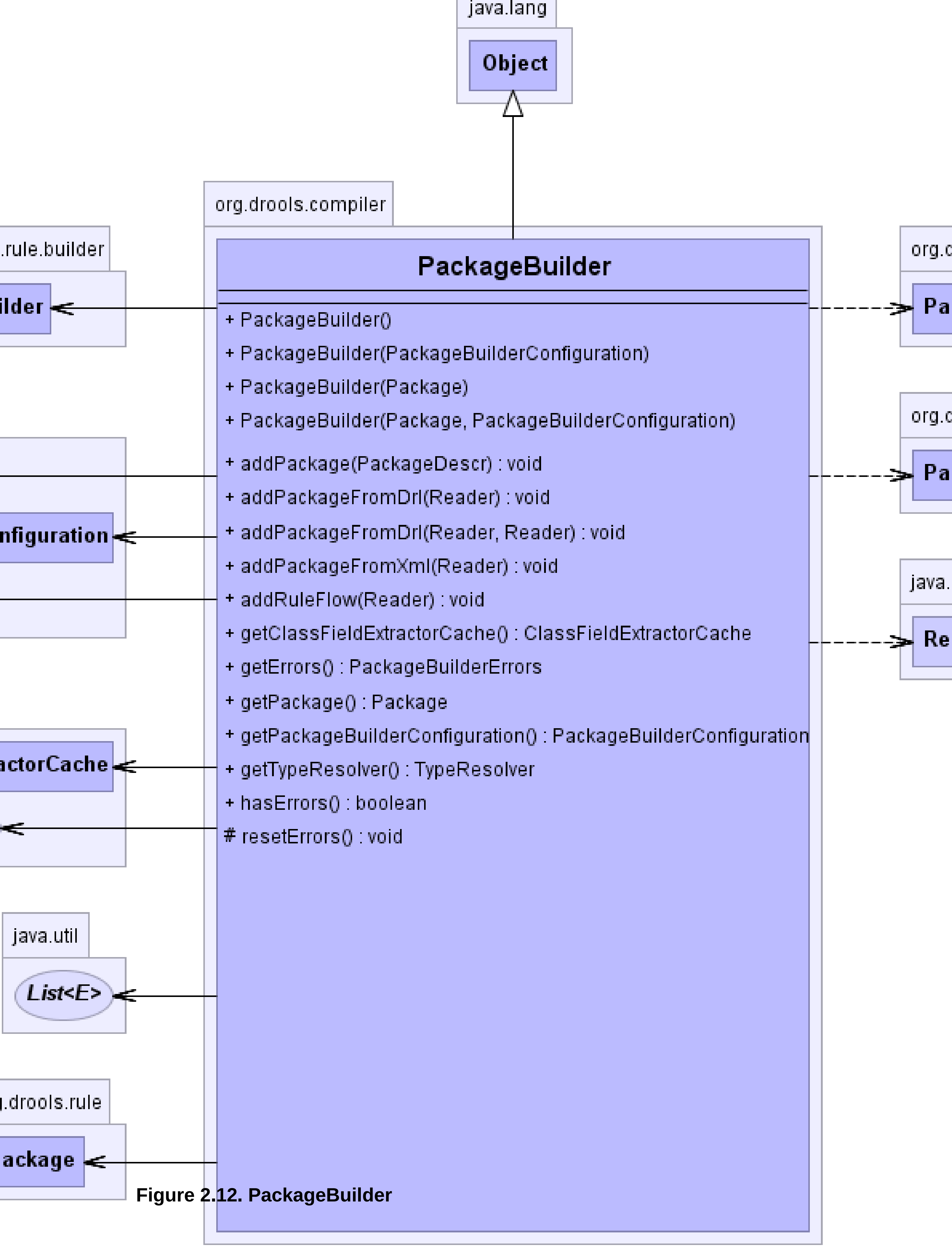

java.lang
Object
org.drools.compiler
PackageBuilder
+ PackageBuilder()
+ PackageBuilder(PackageBuilderConfiguration)
+ PackageBuilder(Package)
+ PackageBuilder(Package, PackageBuilderConfiguration)
+ addPackage(PackageDescr) : void
+ addPackageFromDrl(Reader) : void
+ addPackageFromDrl(Reader, Reader) : void
+ addPackageFromXml(Reader) : void
+ addRuleFlow(Reader) : void
+ getClassFieldExtractorCache() : ClassFieldExtractorCache
+ getErrors() : PackageBuilderErrors
+ getPackage() : Package
+ getPackageBuilderConfiguration() : PackageBuilderConfiguration
+ getTypeResolver() : TypeResolver
+ hasErrors() : boolean
# resetErrors() : void
.rule.builder
ilder
nfiguration
actorCache
java.util
List<E>
.drools.rule
ackage
org.c
Pa
org.c
Pa
java.
Re

**Figure 2.12. 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 serialization 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 and DRL 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!

## 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.

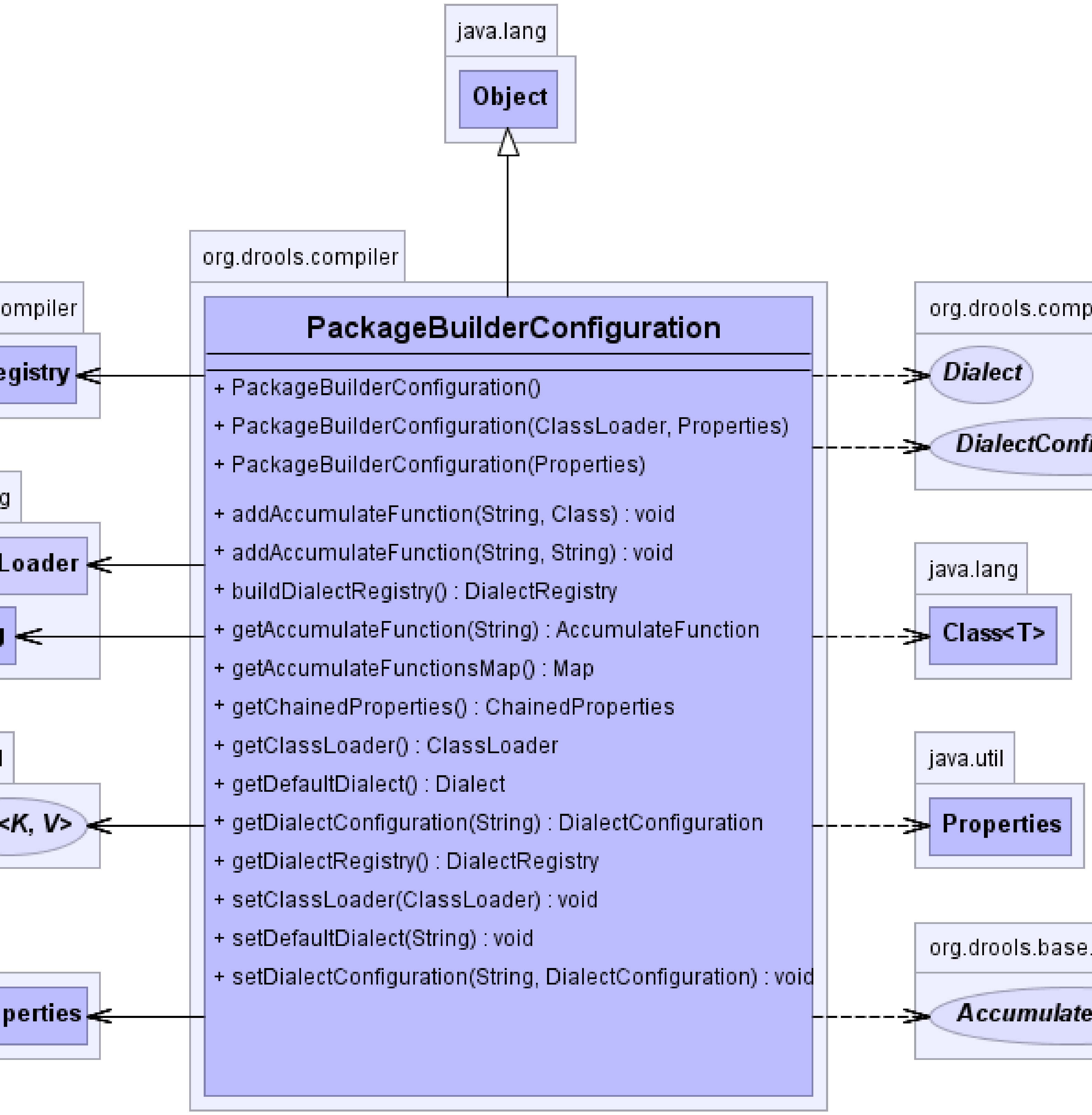

java.lang
Object
org.drools.compiler
PackageBuilderConfiguration
+ PackageBuilderConfiguration()
+ PackageBuilderConfiguration(ClassLoader, Properties)
+ PackageBuilderConfiguration(Properties)
+ addAccumulateFunction(String, Class) : void
+ addAccumulateFunction(String, String) : void
+ buildDialectRegistry() : DialectRegistry
+ getAccumulateFunction(String) : AccumulateFunction
+ getAccumulateFunctionsMap() : Map
+ getChainedProperties() : ChainedProperties
+ getClassLoader() : ClassLoader
+ getDefaultDialect() : Dialect
+ getDialectConfiguration(String) : DialectConfiguration
+ getDialectRegistry() : DialectRegistry
+ setClassLoader(ClassLoader) : void
+ setDefaultDialect(String) : void
+ setDialectConfiguration(String, DialectConfiguration) : void
ompiler
egistry
g
Loader
g
l
<K, V>
perties
org.drools.comp
Dialect
DialectConfi
java.lang
Class<T>
java.util
Properties
org.drools.base.
Accumulate

**Figure 2.13. 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 a number of locations looking for `drools.packagebuilder.conf` files; as it finds them it adds the properties to the master propperties list; this provides 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 `PackageBuilderConfiguration` handles the registry of Accumulate functions, registry of Dialects and the main ClassLoader.

Drools has a pluggable Dialect system, which allows other languages to compile and execution 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.

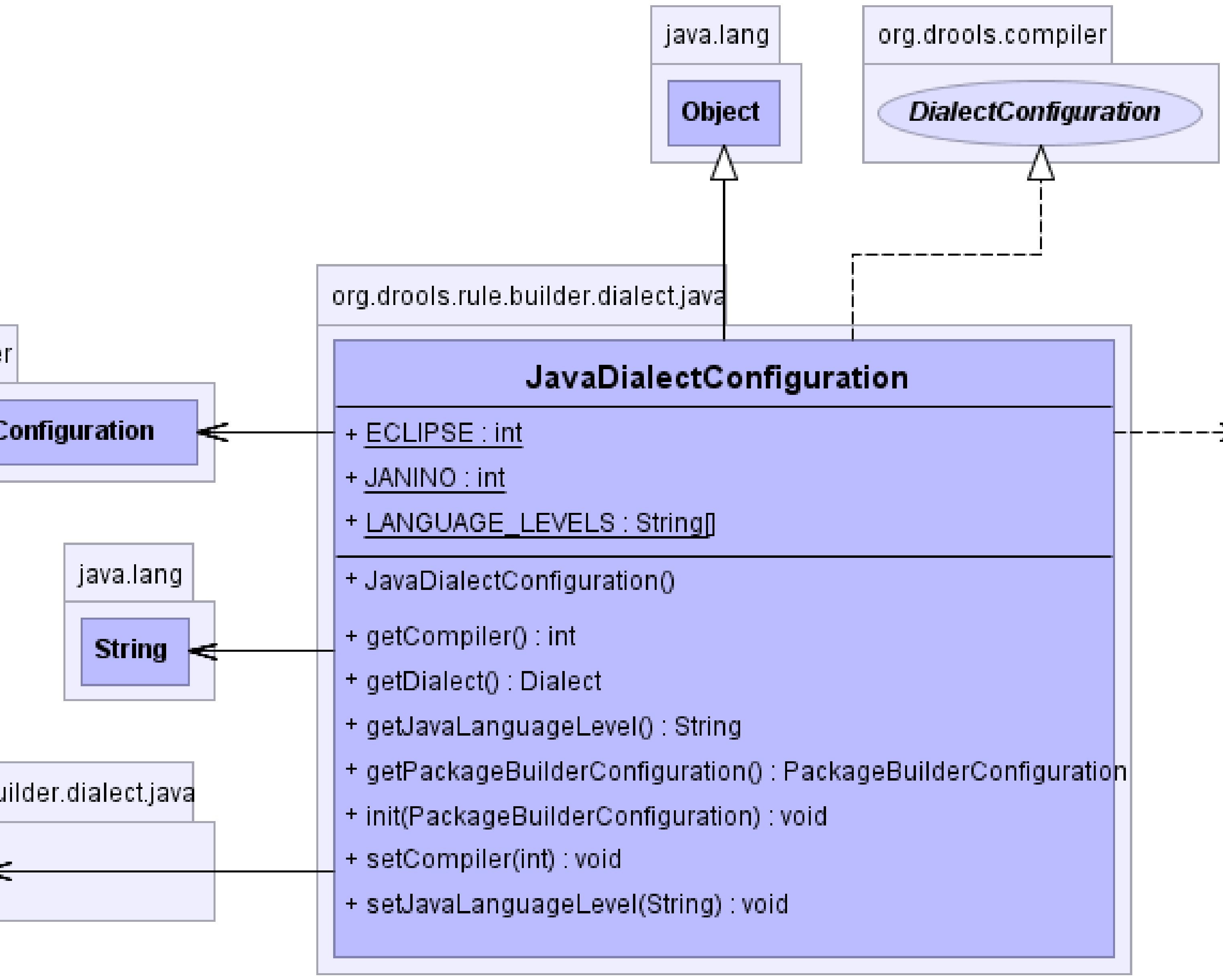

java.lang
Object
org.drools.compiler
DialectConfiguration
org.drools.rule.builder.dialect.java
JavaDialectConfiguration
+ ECLIPSE : int
+ JANINO : int
+ LANGUAGE_LEVELS : String[]
+ JavaDialectConfiguration()
+ getCompiler() : int
+ getDialect() : Dialect
+ getJavaLanguageLevel() : String
+ getPackageBuilderConfiguration() : PackageBuilderConfiguration
+ init(PackageBuilderConfiguration) : void
+ setCompiler(int) : void
+ setJavaLanguageLevel(String) : void
Configuration
java.lang
String
uilder.dialect.java

**Figure 2.14.** `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; note this time I use properties 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" );
assertEquals( JavaDialectConfiguration.JANINO,
              javaConf.getCompiler() ); // demonstrate that the compiler is
 correctly configured
```

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.

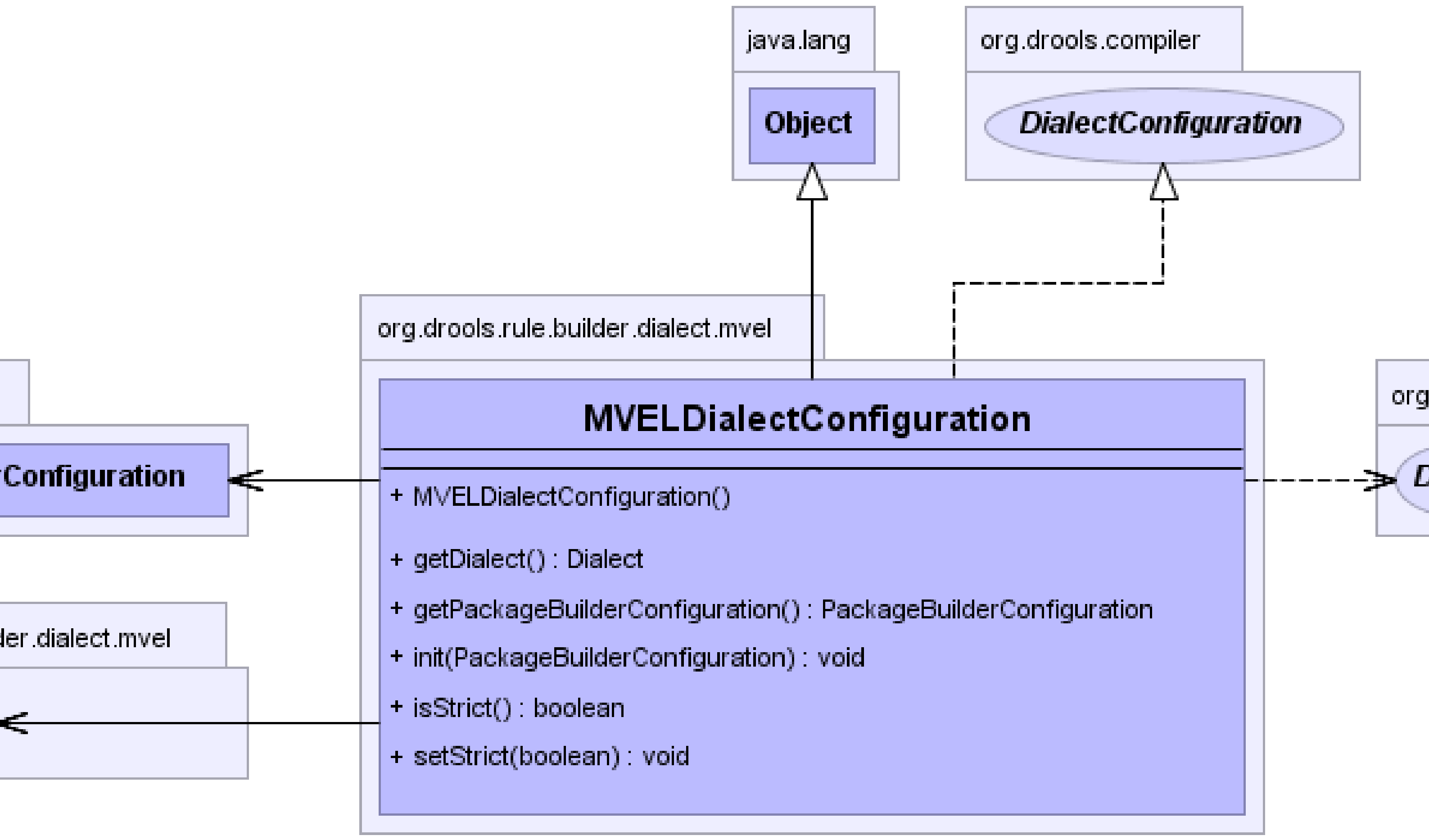

java.lang
Object
org.drools.compiler
DialectConfiguration
org.drools.rule.builder.dialect.mvel
MVELDialectConfiguration
+ MVELDialectConfiguration()
+ getDialect() : Dialect
+ getPackageBuilderConfiguration() : PackageBuilderConfiguration
+ init(PackageBuilderConfiguration) : void
+ isStrict() : boolean
+ setStrict(boolean) : void
Configuration
ler.dialect.mvel
org

**Figure 2.15.** `MvelDialectConfiguration`

### 2.5.3. RuleBase

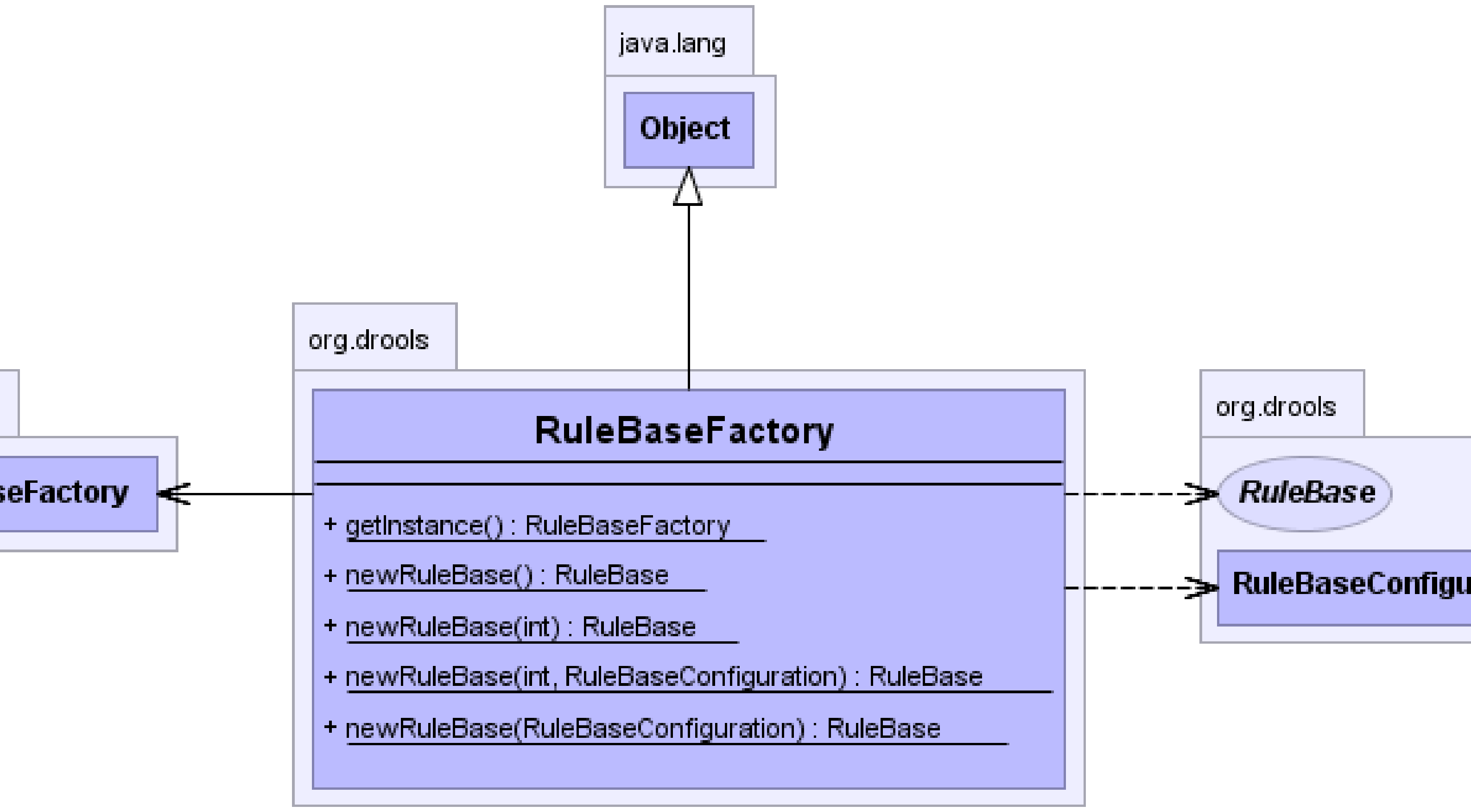

java.lang
Object
org.drools
RuleBaseFactory
+ getInstance() : RuleBaseFactory
+ newRuleBase() : RuleBase
+ newRuleBase(int) : RuleBase
+ newRuleBase(int, RuleBaseConfiguration) : RuleBase
+ newRuleBase(RuleBaseConfiguration) : RuleBase
eFactory
org.drools
RuleBase
RuleBaseConfigu

**Figure 2.16. `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  );
```

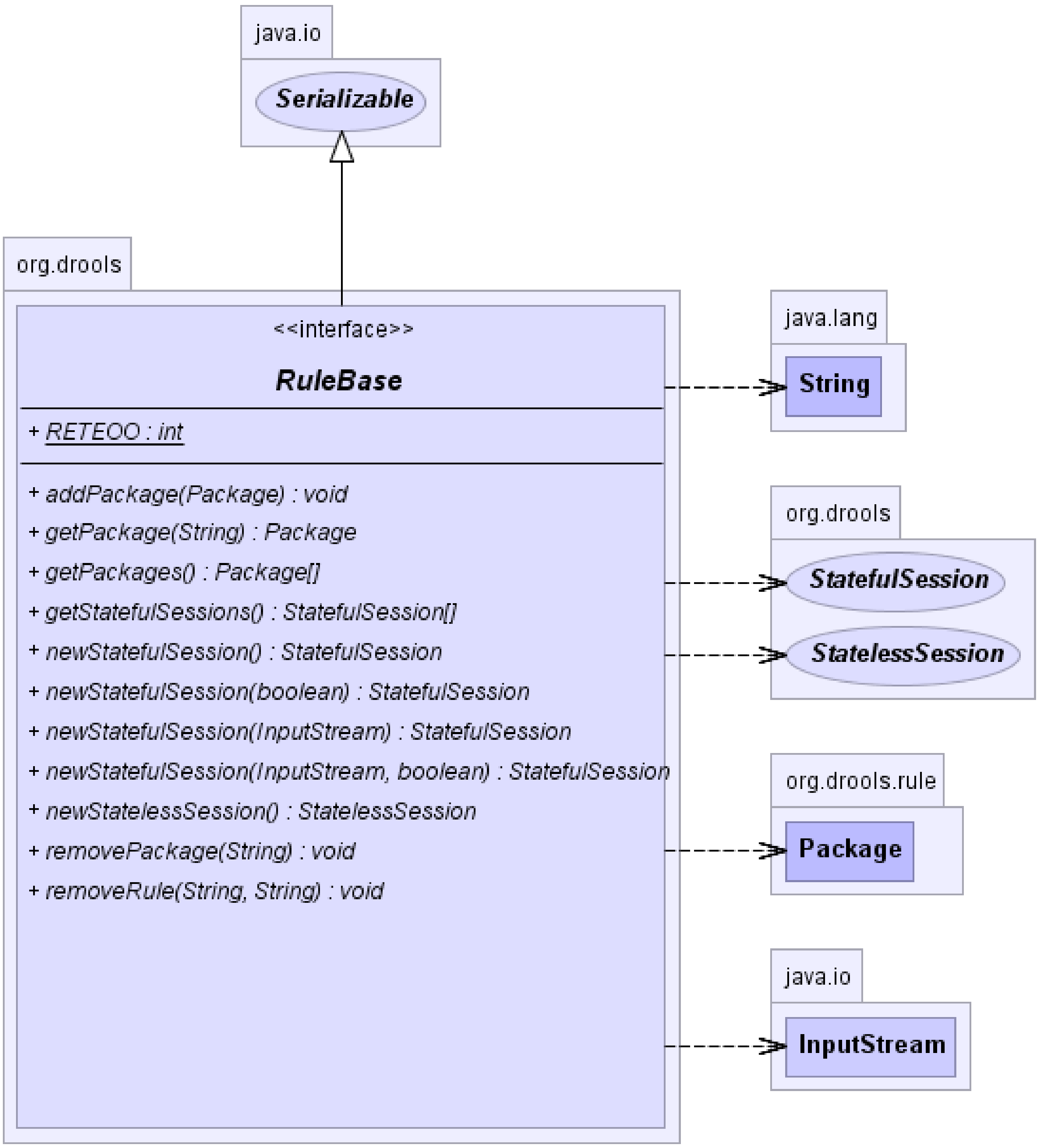

java.io
Serializable
org.drools
<<interface>>
RuleBase
+ RETEOO : int
+ addPackage(Package) : void
+ getPackage(String) : Package
+ getPackages() : Package[]
+ getStatefulSessions() : StatefulSession[]
+ newStatefulSession() : StatefulSession
+ newStatefulSession(boolean) : StatefulSession
+ newStatefulSession(InputStream) : StatefulSession
+ newStatefulSession(InputStream, boolean) : StatefulSession
+ newStatelessSession() : StatelessSession
+ removePackage(String) : void
+ removeRule(String, String) : void
java.lang
String
org.drools
StatefulSession
StatelessSession
org.drools.rule
Package
java.io
InputStream

**Figure 2.17. RuleBase**

A RuleBase contains one or more more packages of rules, ready to be used, i.e., they have been validated/compiled etc. A RuleBase 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 RuleBase; which is expensive.

A RuleBase 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 RuleBase 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 RuleBase 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 - all changes will be propagated to the existing stateful sessions; don't 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 indivual rule, there is no method to add an individual rule - to achieve this 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 RuleBase. 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 );
```

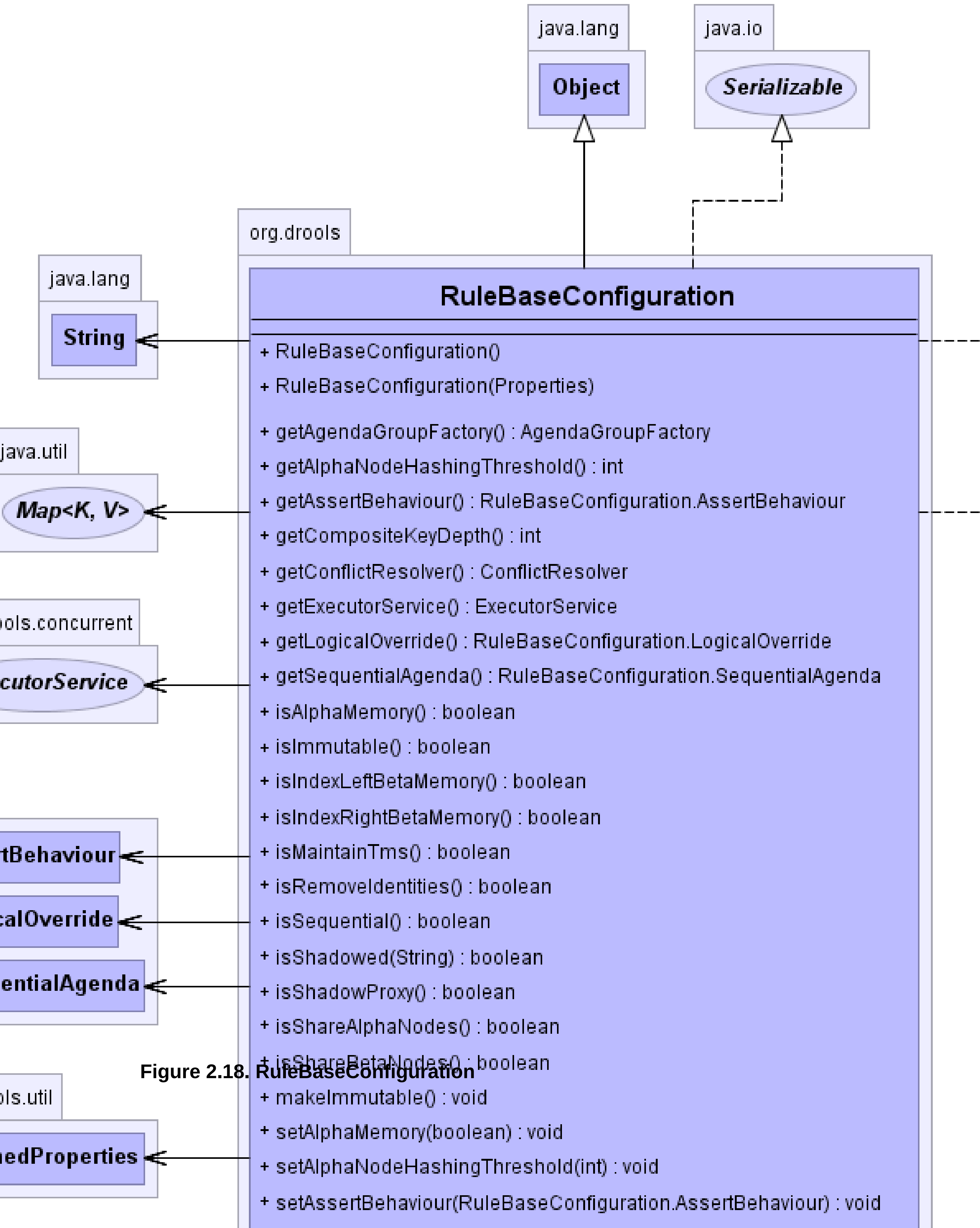

java.lang
Object
java.io
Serializable
org.drools
RuleBaseConfiguration
+ RuleBaseConfiguration()
+ RuleBaseConfiguration(Properties)
+ getAgendaGroupFactory() : AgendaGroupFactory
+ getAlphaNodeHashingThreshold() : int
+ getAssertBehaviour() : RuleBaseConfiguration.AssertBehaviour
+ getCompositeKeyDepth() : int
+ getConflictResolver() : ConflictResolver
+ getExecutorService() : ExecutorService
+ getLogicalOverride() : RuleBaseConfiguration.LogicalOverride
+ getSequentialAgenda() : RuleBaseConfiguration.SequentialAgenda
+ isAlphaMemory() : boolean
+ isImmutable() : boolean
+ isIndexLeftBetaMemory() : boolean
+ isIndexRightBetaMemory() : boolean
+ isMaintainTms() : boolean
+ isRemoveIdentities() : boolean
+ isSequential() : boolean
+ isShadowed(String) : boolean
+ isShadowProxy() : boolean
+ isShareAlphaNodes() : boolean
+ isShareBetaNodes() : boolean
+ makeImmutable() : void
+ setAlphaMemory(boolean) : void
+ setAlphaNodeHashingThreshold(int) : void
+ setAssertBehaviour(RuleBaseConfiguration.AssertBehaviour) : void
java.lang
String
java.util
Map<K, V>
ols.concurrent
cutorService
tBehaviour
calOverride
entialAgenda
ols.util
edProperties

Figure 2.18. RuleBaseConfiguration

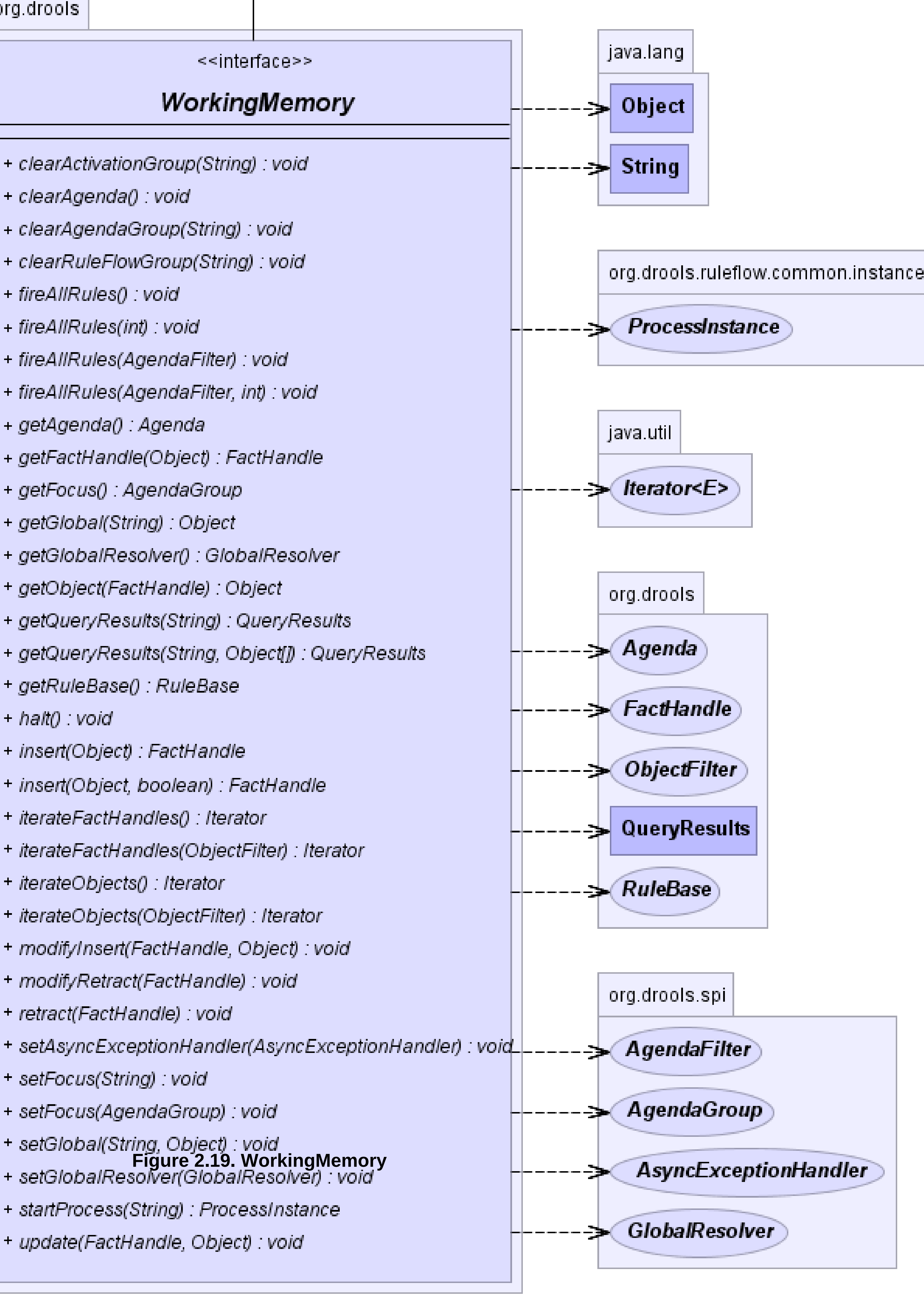

org.drools
<<interface>>
WorkingMemory
+ clearActivationGroup(String) : void
+ clearAgenda() : void
+ clearAgendaGroup(String) : void
+ clearRuleFlowGroup(String) : void
+ fireAllRules() : void
+ fireAllRules(int) : void
+ fireAllRules(AgendaFilter) : void
+ fireAllRules(AgendaFilter, int) : void
+ getAgenda() : Agenda
+ getFactHandle(Object) : FactHandle
+ getFocus() : AgendaGroup
+ getGlobal(String) : Object
+ getGlobalResolver() : GlobalResolver
+ getObject(FactHandle) : Object
+ getQueryResults(String) : QueryResults
+ getQueryResults(String, Object[]) : QueryResults
+ getRuleBase() : RuleBase
+ halt() : void
+ insert(Object) : FactHandle
+ insert(Object, boolean) : FactHandle
+ iterateFactHandles() : Iterator
+ iterateFactHandles(ObjectFilter) : Iterator
+ iterateObjects() : Iterator
+ iterateObjects(ObjectFilter) : Iterator
+ modifyInsert(FactHandle, Object) : void
+ modifyRetract(FactHandle) : void
+ retract(FactHandle) : void
+ setAsyncExceptionHandler(AsyncExceptionHandler) : void
+ setFocus(String) : void
+ setFocus(AgendaGroup) : void
+ setGlobal(String, Object) : void
+ setGlobalResolver(GlobalResolver) : void
+ startProcess(String) : ProcessInstance
+ update(FactHandle, Object) : void
java.lang
Object
String
org.drools.ruleflow.common.instance
ProcessInstance
java.util
Iterator<E>
org.drools
Agenda
FactHandle
ObjectFilter
QueryResults
RuleBase
org.drools.spi
AgendaFilter
AgendaGroup
AsyncExceptionHandler
GlobalResolver

**Figure 2.19. 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 will be used when interacting with the `WorkingMemory` 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 have 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-processed. 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.modifyRetract()` 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's generally 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.

**Important**

Since 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():** this way, 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?
           session.modifyRetract( handle ); // call modifyRetract() before
 doing changes
```

```
        person.setAge( 31 );
        person.setLikes( "chocolate" );
        session.modifyInsert( handle, person ); // call modifyInsert()
 after the changes
```

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

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 later on), 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 is no other fact to start with).

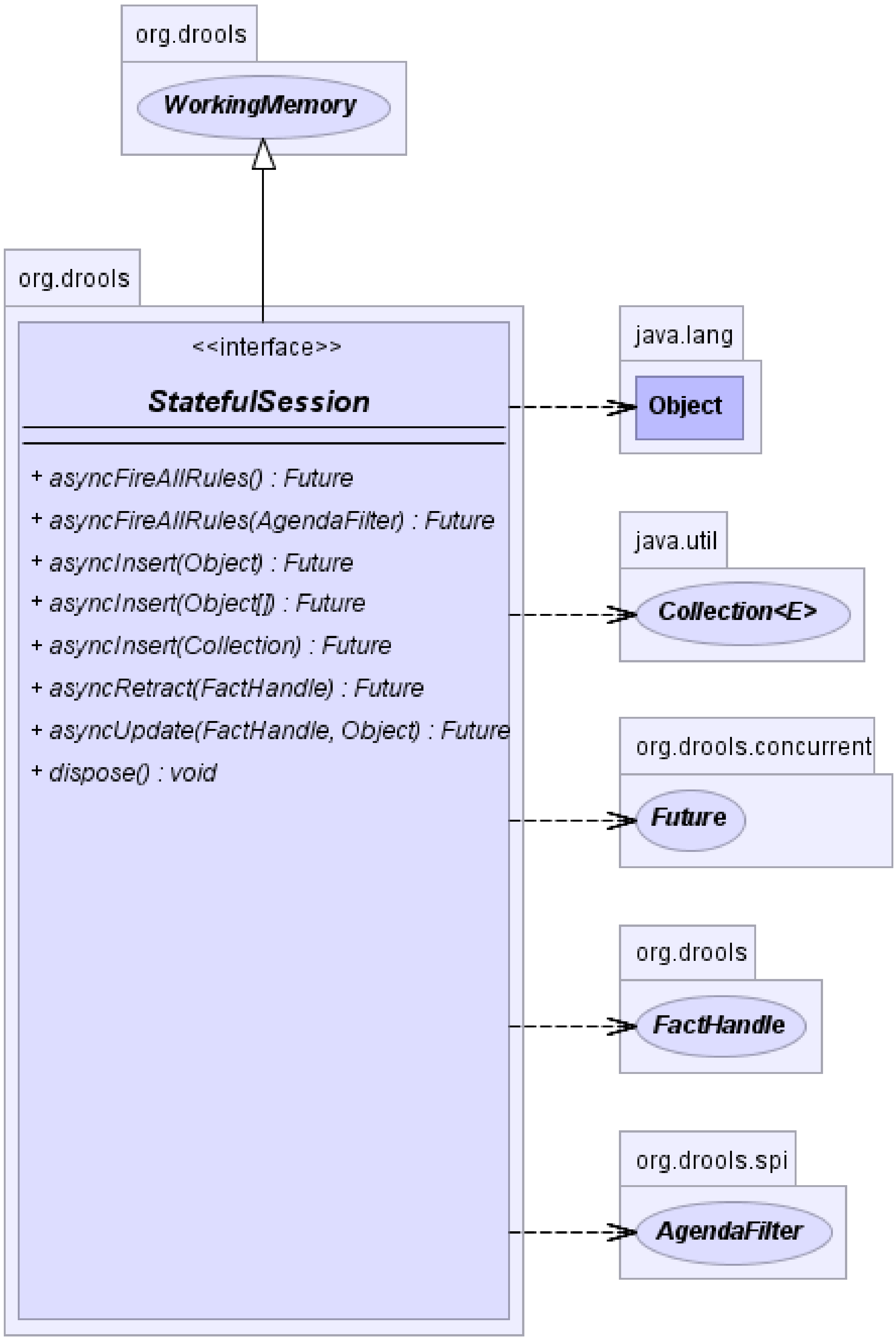

org.drools
WorkingMemory
org.drools
<<interface>>
StatefulSession
+ asyncFireAllRules() : Future
+ asyncFireAllRules(AgendaFilter) : Future
+ asyncInsert(Object) : Future
+ asyncInsert(Object[]) : Future
+ asyncInsert(Collection) : Future
+ asyncRetract(FactHandle) : Future
+ asyncUpdate(FactHandle, Object) : Future
+ dispose() : void
java.lang
Object
java.util
Collection<E>
org.drools.concurrent
Future
org.drools
FactHandle
org.drools.spi
AgendaFilter

**Figure 2.20. StatefulSession**

The `StatefulSession` extends the `WorkingMemory` class. It simply adds async methods and a `dispose()` method. The `ruleBase` retains a reference to each `StatefulSession` it 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();
```

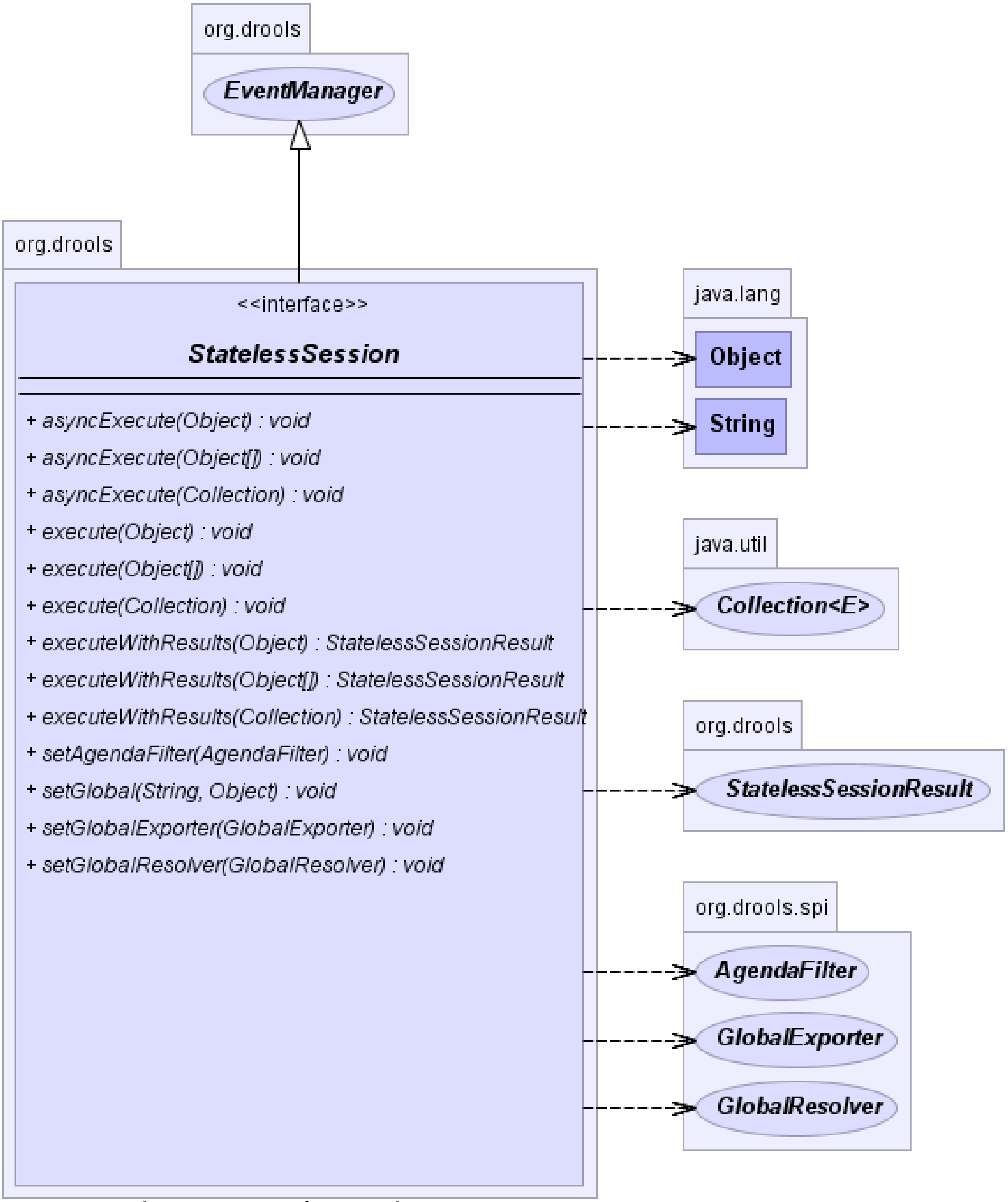

org.drools
EventManager
org.drools
<<interface>>
StatelessSession
+ asyncExecute(Object) : void
+ asyncExecute(Object[]) : void
+ asyncExecute(Collection) : void
+ execute(Object) : void
+ execute(Object[]) : void
+ execute(Collection) : void
+ executeWithResults(Object) : StatelessSessionResult
+ executeWithResults(Object[]) : StatelessSessionResult
+ executeWithResults(Collection) : StatelessSessionResult
+ setAgendaFilter(AgendaFilter) : void
+ setGlobal(String, Object) : void
+ setGlobalExporter(GlobalExporter) : void
+ setGlobalResolver(GlobalResolver) : void
java.lang
Object
String
java.util
Collection<E>
org.drools
StatelessSessionResult
org.drools.spi
AgendaFilter
GlobalExporter
GlobalResolver

**Figure 2.21. 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 thus much simpler; which in turn can make 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; session 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 it's 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 optimized mode that uses less memory and executes faster; please see the *Sequential* section for more details.

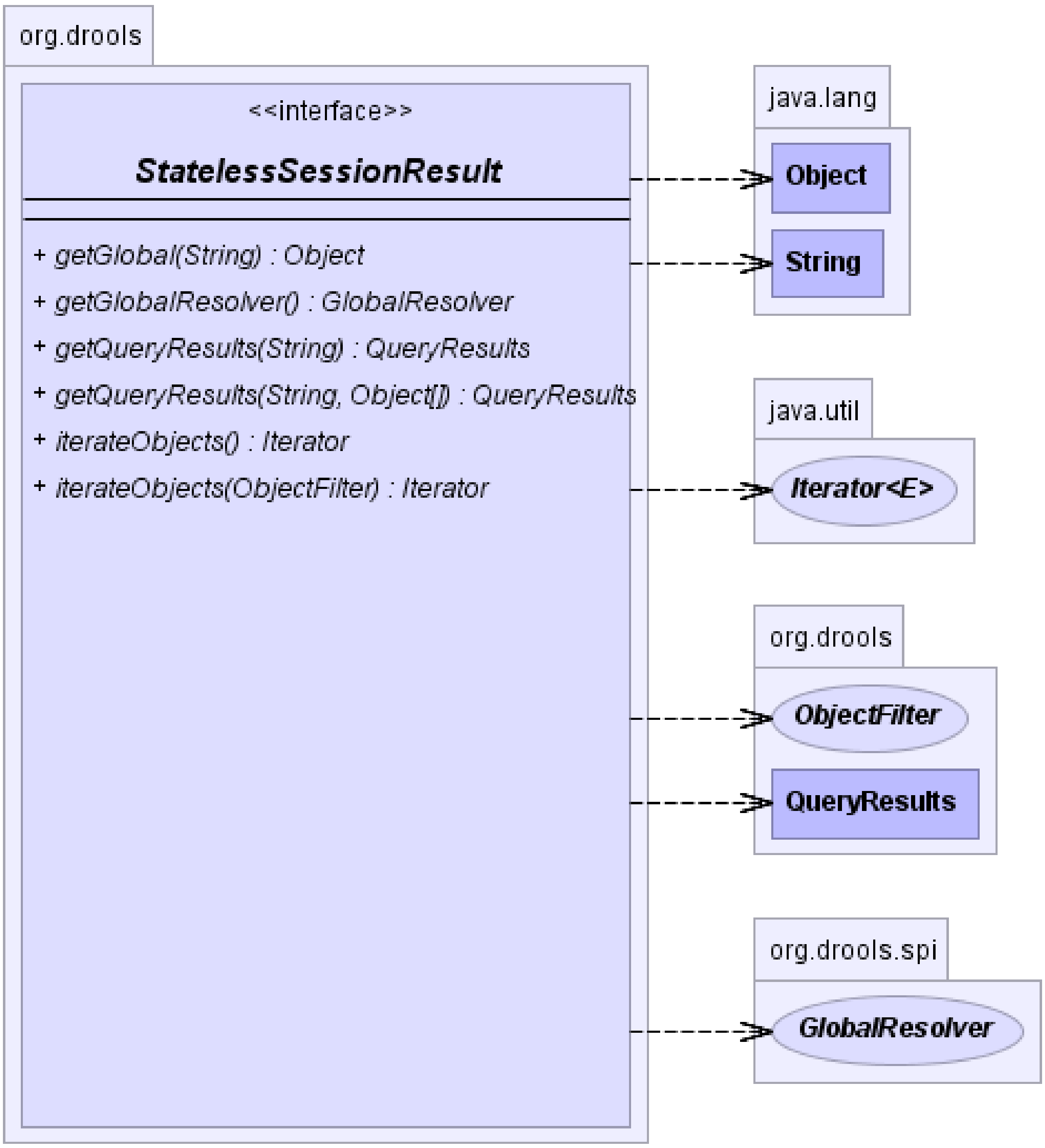

org.drools
<<interface>>
StatelessSessionResult
+ getGlobal(String) : Object
+ getGlobalResolver() : GlobalResolver
+ getQueryResults(String) : QueryResults
+ getQueryResults(String, Object[]) : QueryResults
+ iterateObjects() : Iterator
+ iterateObjects(ObjectFilter) : Iterator
java.lang
Object
String
java.util
Iterator<E>
org.drools
ObjectFilter
QueryResults
org.drools.spi
GlobalResolver

**Figure 2.22. StatelessSessionResult**

`StatelessSession.executeWithResults(....)` returns a minimal API to examine the session's 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 declared in the `RuleBase`. `ReferenceOriginalGlobalExporter` just passes a reference to the original `GlobalResolver`; the latter should be used with care as identifier instances can be changed at any time by the `StatelessSession` and the `GlobalResolver` may not be serializable-friendly.

### 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

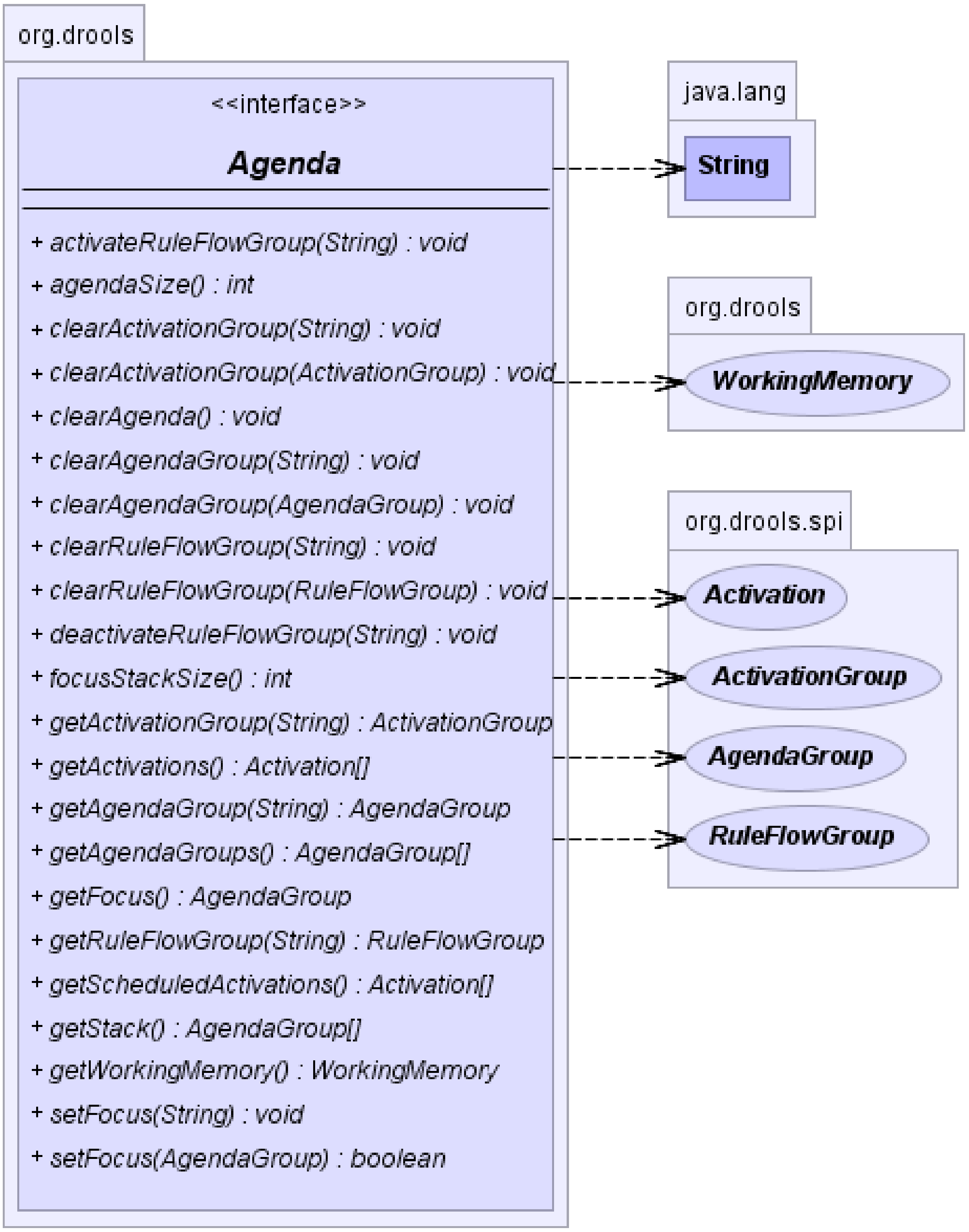

org.drools
<<interface>>
Agenda
+ activateRuleFlowGroup(String) : void
+ agendaSize() : int
+ clearActivationGroup(String) : void
+ clearActivationGroup(ActivationGroup) : void
+ clearAgenda() : void
+ clearAgendaGroup(String) : void
+ clearAgendaGroup(AgendaGroup) : void
+ clearRuleFlowGroup(String) : void
+ clearRuleFlowGroup(RuleFlowGroup) : void
+ deactivateRuleFlowGroup(String) : void
+ focusStackSize() : int
+ getActivationGroup(String) : ActivationGroup
+ getActivations() : Activation[]
+ getAgendaGroup(String) : AgendaGroup
+ getAgendaGroups() : AgendaGroup[]
+ getFocus() : AgendaGroup
+ getRuleFlowGroup(String) : RuleFlowGroup
+ getScheduledActivations() : Activation[]
+ getStack() : AgendaGroup[]
+ getWorkingMemory() : WorkingMemory
+ setFocus(String) : void
+ setFocus(AgendaGroup) : boolean
java.lang
String
org.drools
WorkingMemory
org.drools.spi
Activation
ActivationGroup
AgendaGroup
RuleFlowGroup

**Figure 2.23. 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 Actions - 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 - attempts to select a rule to fire, if a rule is not found it exits, otherwise it attempts to fire the found rule, switching the phase back to Working Memory Actions and the process repeats again until the Agenda is empty.

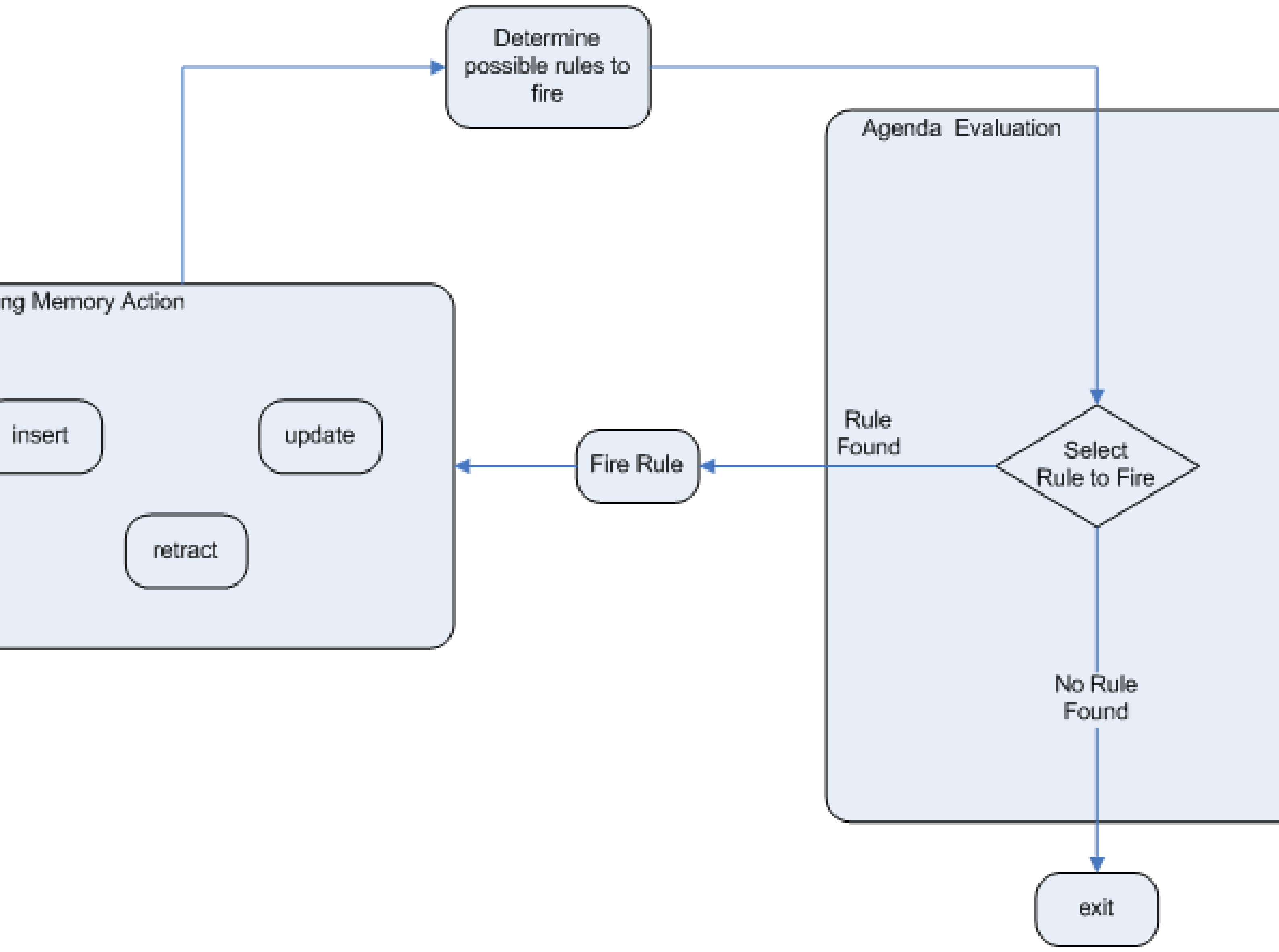


**Figure 2.24. Two Phase Execution**

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 Drools are: Salience and LIFO (last in, first out).

The most visible one is "salience" or priority, in which case a user can specify that a certain rule has a higher priority (by giving it a higher number) than other rules. In that case, the higher salience rule will always be preferred. LIFO priorities are based on the assigned Working Memory Action counter value, multiple rules created from the same action have the same value - execution of these are considered arbitrary.

As a general rule, it is a good idea not to count on the rules firing in any particular order, and try and author the rules without worrying about a "flow".

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

### 2.5.7.2. Agenda Groups

Agenda groups are a way to partition rules (activations, actually) on the agenda. At any one time, only one group has "focus" which means that the activations for rules in that group will only take effect - you can also have rules "auto focus" which means the focus for its agenda group is taken when that rules conditions are true.

They are sometimes known as "modules" in CLIPS terminology. Agenda groups are a handy way to create a "flow" between grouped rules. You can switch the group which has focus either from within the rule engine, or from the API. If your rules have a clear need for multiple "phases" or "sequences" of processing, consider using agenda-groups for this purpose.

Each time `setFocus(...)` is called it pushes that Agenda Group onto a stack, when the focus 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

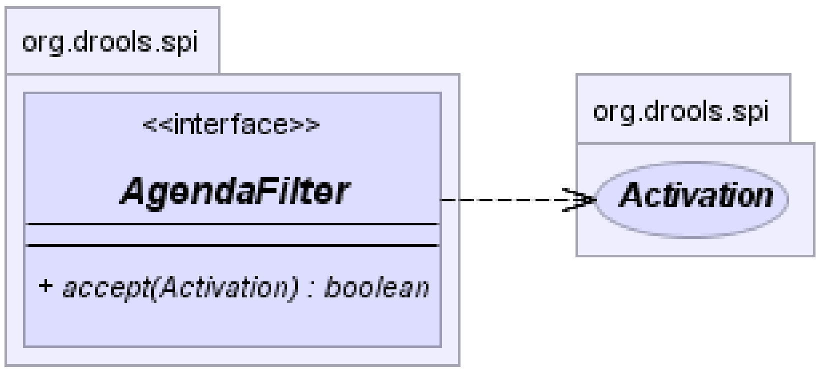


**Figure 2.25. AgendaFilters**

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

- `RuleNameEndsWithAgendaFilter`
- `RuleNameEqualsAgendaFilter`
- `RuleNameStartsWithAgendaFilter`
- `RuleNameMatchesAgendaFilter`

To use a filter specify it while calling `FireAllRules`. The following example will only allow activation of rules ending with the test *Test*. All others will be filtered out:

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

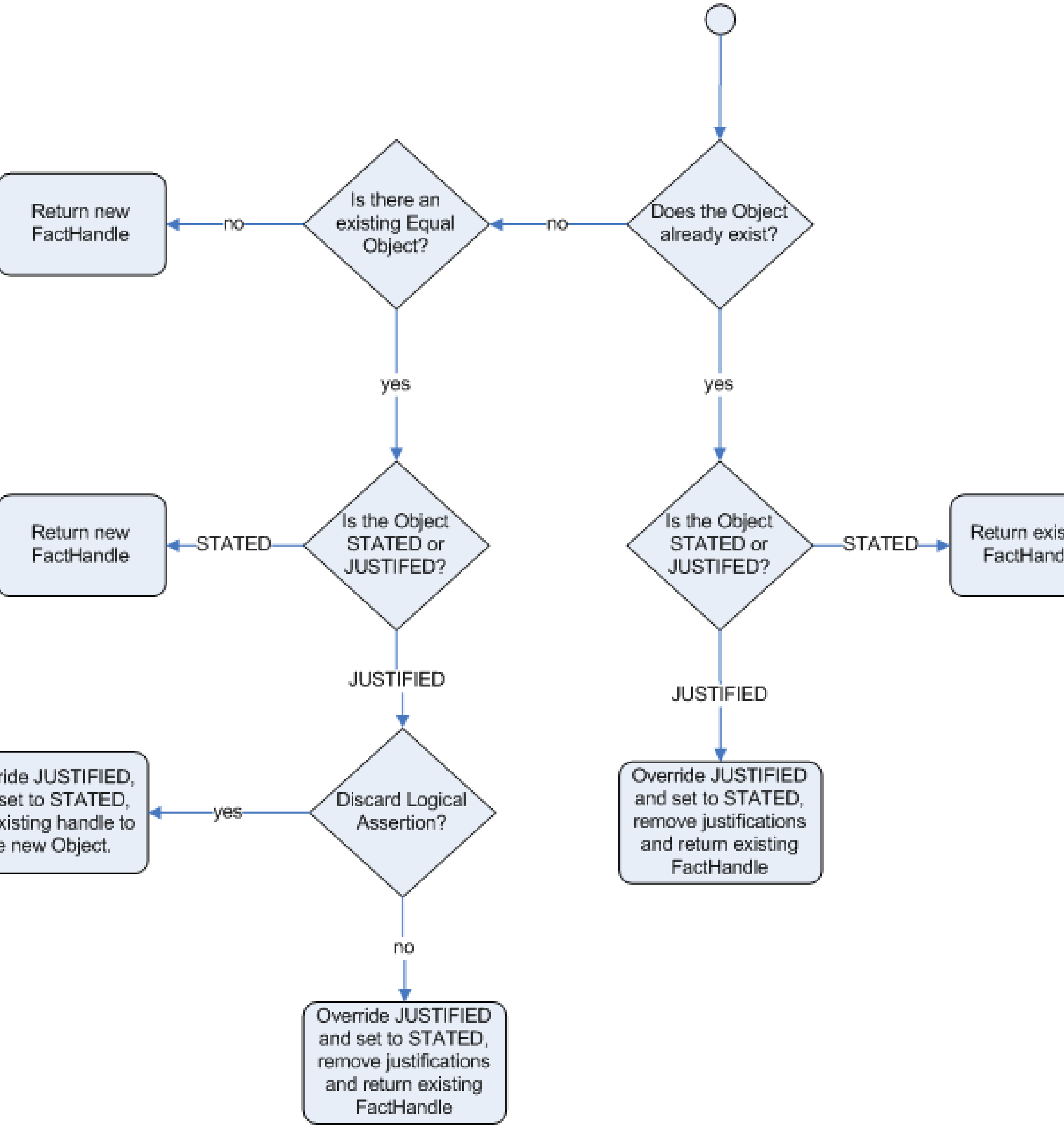
### 2.5.8. Truth Maintenance with Logical Objects

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. (It's actually more clever than that! If there are no possible conditions that could support the logical assertion, only then it will be retracted).

Normal insertions are said to be “STATED” (i.e. The Fact has been stated - just like the intuitive concept). Using a `HashMap` and a counter we track how many times a particular equality is STATED; this means we count 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 existing equal object that is JUSTIFIED we override the Fact - how this override works depends on the configuration setting `WM_BEHAVIOR_PRESERVE`. When the property is set to discard we use the existing handle and replace the existing instance with the new Object - this is the default behavior - 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.



Does the Object already exist?
no
Is there an existing Equal Object?
no
Return new FactHandle
yes
yes
Is the Object STATED or JUSTIFED?
STATED
Return new FactHandle
Is the Object STATED or JUSTIFED?
STATED
Return exis
FactHand
JUSTIFIED
JUSTIFIED
Discard Logical Assertion?
yes
ide JUSTIFIED, set to STATED, xisting handle to e new Object.
Override JUSTIFIED and set to STATED, remove justifications and return existing FactHandle
no
Override JUSTIFIED and set to STATED, remove justifications and return existing FactHandle

**Figure 2.26. Stated Insertion**

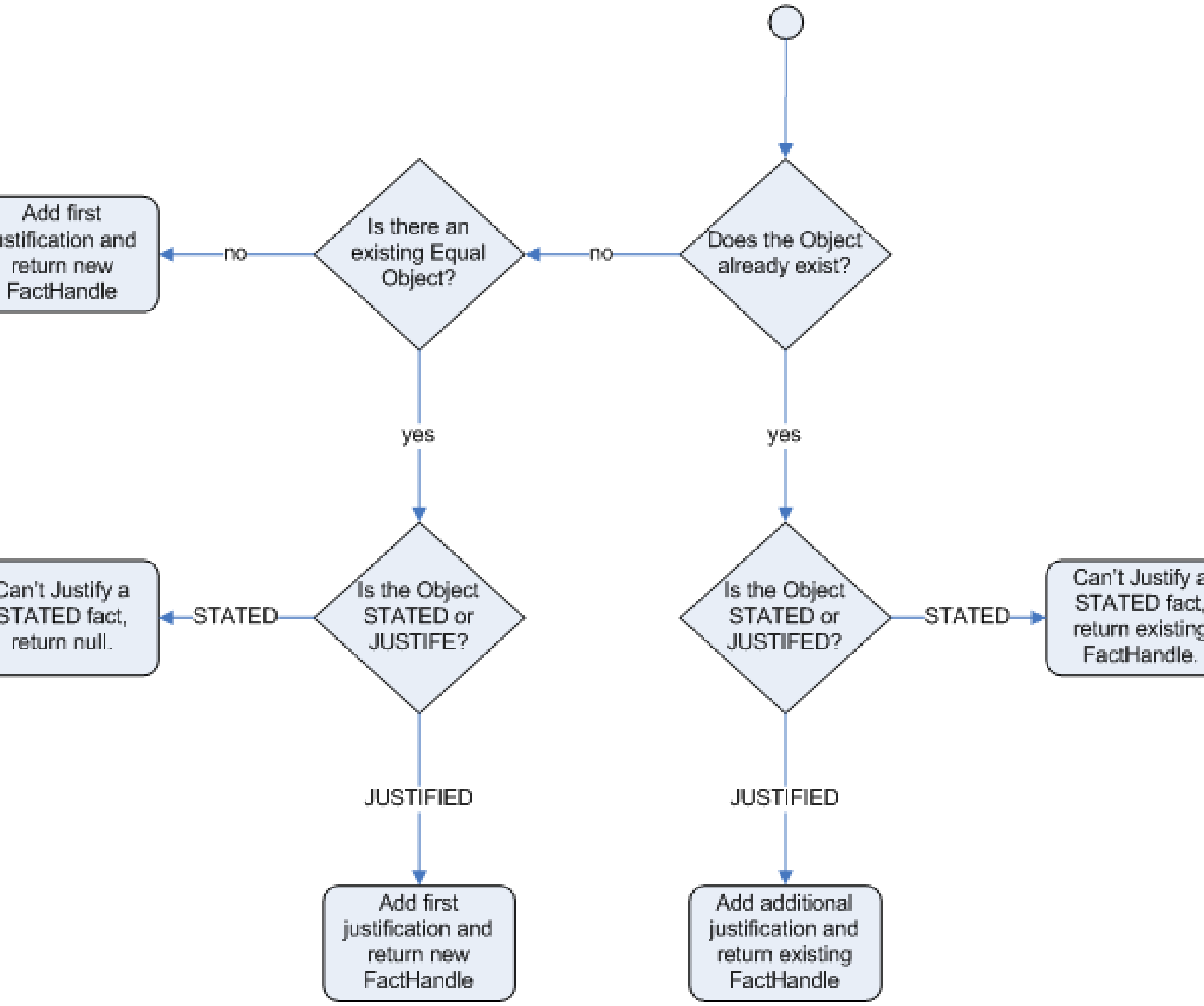

Does the Object already exist?
no
Is there an existing Equal Object?
no
Add first justification and return new FactHandle
yes
yes
Is the Object STATED or JUSTIFE?
STATED
Can't Justify a STATED fact, return null.
Is the Object STATED or JUSTIFED?
STATED
Can't Justify a STATED fact, return existing FactHandle.
JUSTIFIED
JUSTIFIED
Add first justification and return new FactHandle
Add additional justification and return existing FactHandle

**Figure 2.27. 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 kick in 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, etc.). Now 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 kick in which flushes 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) 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 means to be notified of rule engine events, including rules firing, objects being asserted, etc. This allows you to separate out logging/auditing activities from the main part of your application (and the rules) - as events are a cross cutting concern.

There are three types of event listeners - WorkingMemoryEventListener, AgendaEventListener RuleFlowEventListener.

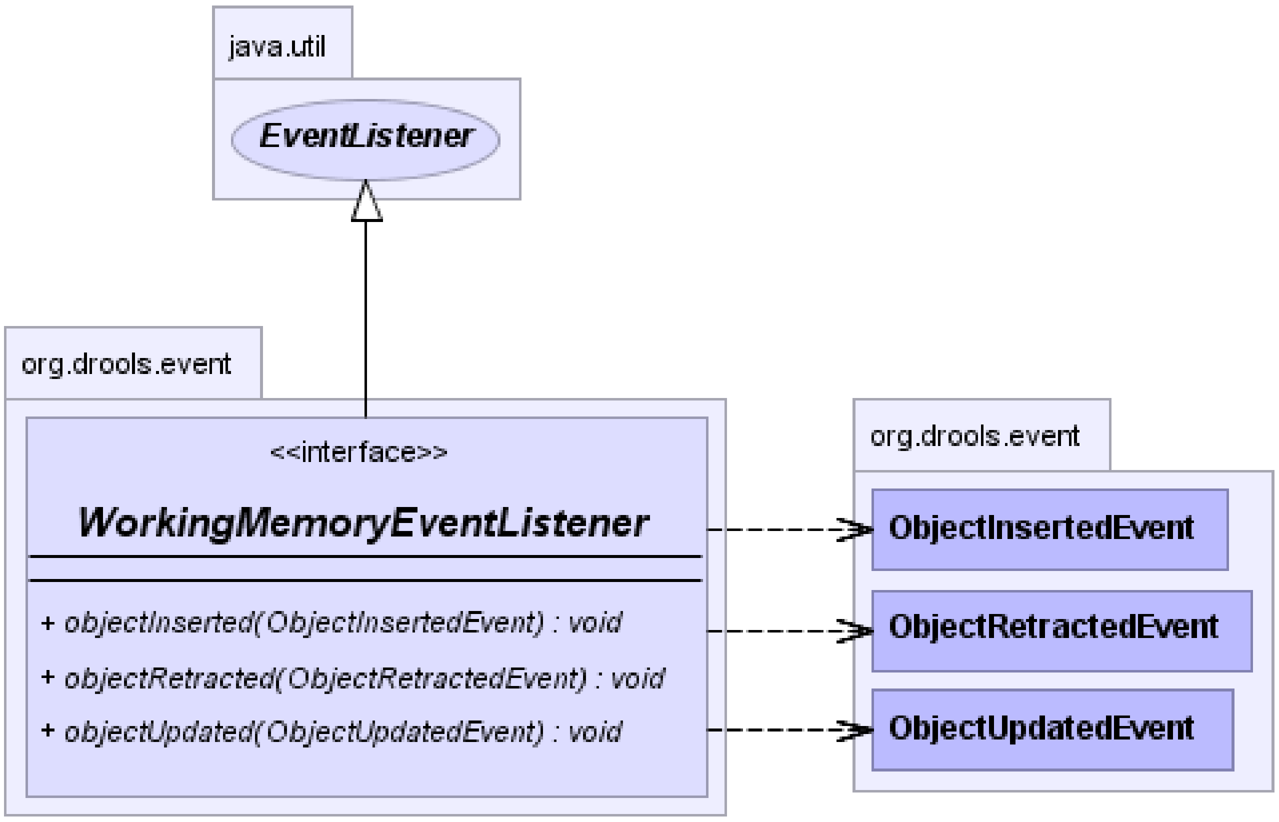
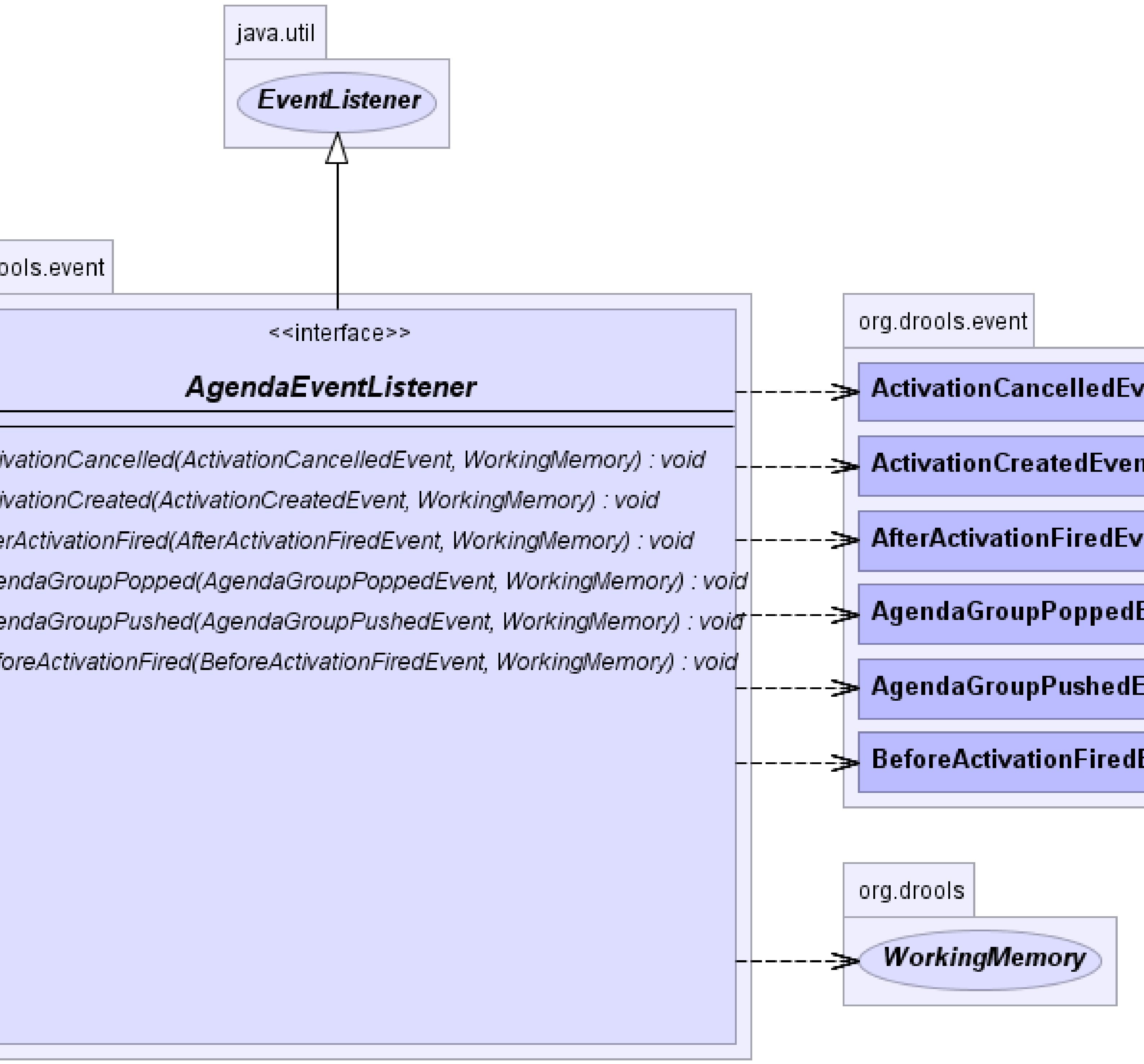

java.util
EventListener
org.drools.event
<<interface>>
WorkingMemoryEventListener
+ objectInserted(ObjectInsertedEvent) : void
+ objectRetracted(ObjectRetractedEvent) : void
+ objectUpdated(ObjectUpdatedEvent) : void
org.drools.event
ObjectInsertedEvent
ObjectRetractedEvent
ObjectUpdatedEvent

**Figure 2.28. WorkingMemoryEventListener**



java.util
EventListener
ools.event
<<interface>>
AgendaEventListener
ivationCancelled(ActivationCancelledEvent, WorkingMemory) : void
ivationCreated(ActivationCreatedEvent, WorkingMemory) : void
erActivationFired(AfterActivationFiredEvent, WorkingMemory) : void
endaGroupPopped(AgendaGroupPoppedEvent, WorkingMemory) : void
endaGroupPushed(AgendaGroupPushedEvent, WorkingMemory) : void
foreActivationFired(BeforeActivationFiredEvent, WorkingMemory) : void
org.drools.event
ActivationCancelledEv
ActivationCreatedEven
AfterActivationFiredEv
AgendaGroupPoppedE
AgendaGroupPushedE
BeforeActivationFiredE
org.drools
WorkingMemory

**Figure 2.29. AgendaEventListener**

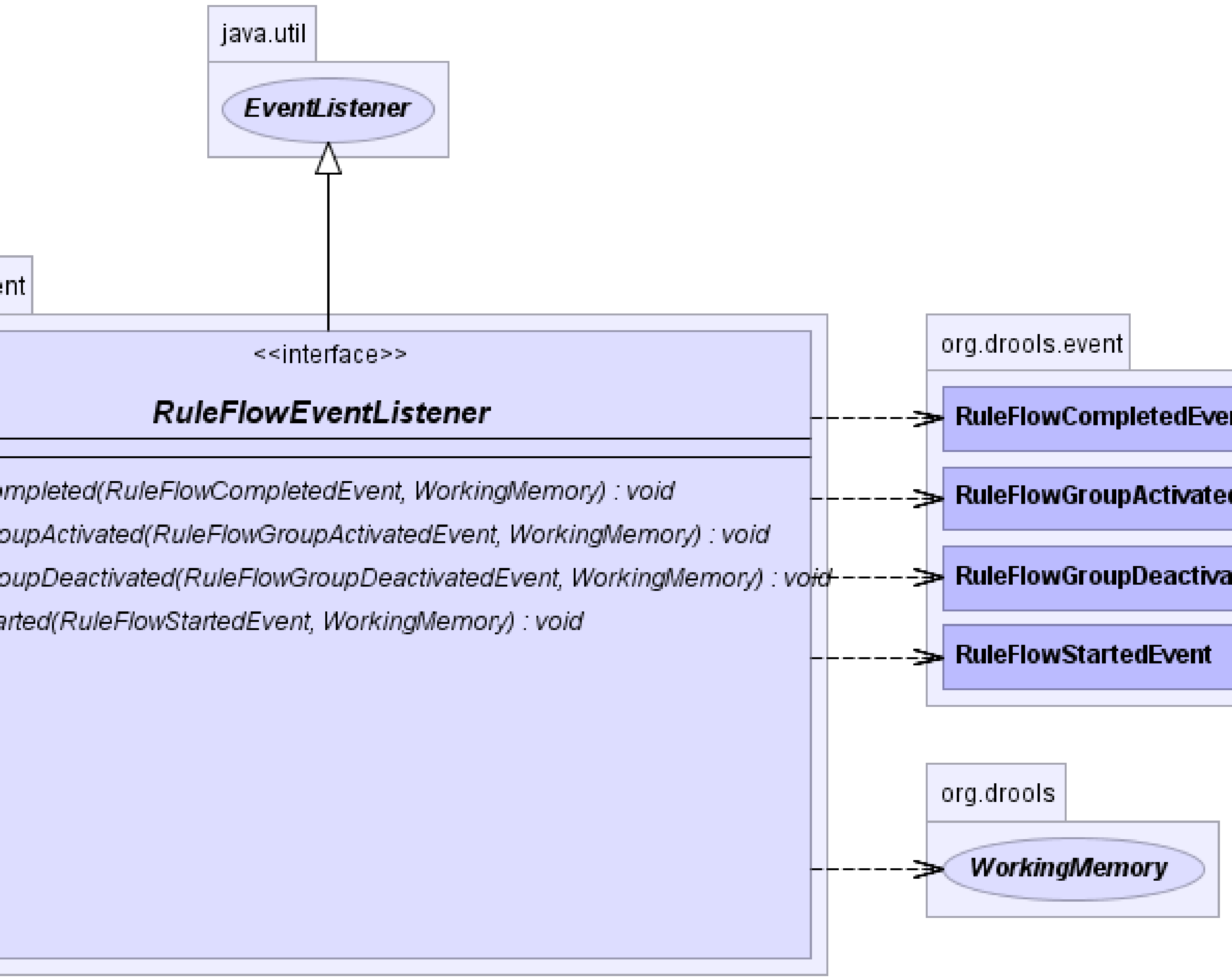

java.util
EventListener
ent
<<interface>>
RuleFlowEventListener
mpleted(RuleFlowCompletedEvent, WorkingMemory) : void
oupActivated(RuleFlowGroupActivatedEvent, WorkingMemory) : void
oupDeactivated(RuleFlowGroupDeactivatedEvent, WorkingMemory) : void
arted(RuleFlowStartedEvent, WorkingMemory) : void
org.drools.event
RuleFlowCompletedEven
RuleFlowGroupActivated
RuleFlowGroupDeactiva
RuleFlowStartedEvent
org.drools
WorkingMemory

**Figure 2.30. RuEventListener**

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

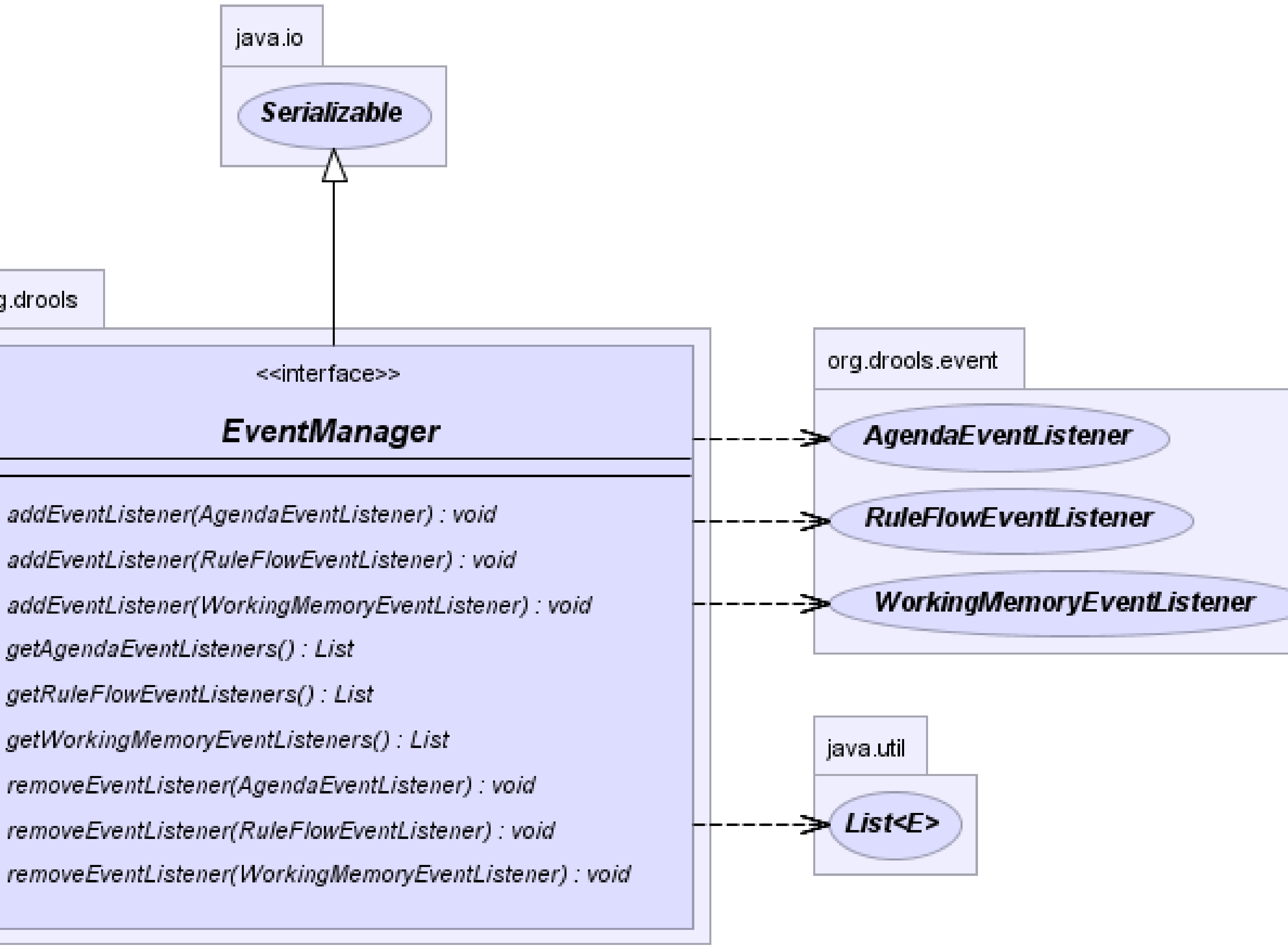

java.io
Serializable
g.drools
<<interface>>
EventManager
addEventListener(AgendaEventListener) : void
addEventListener(RuleFlowEventListener) : void
addEventListener(WorkingMemoryEventListener) : void
getAgendaEventListeners() : List
getRuleFlowEventListeners() : List
getWorkingMemoryEventListeners() : List
removeEventListener(AgendaEventListener) : void
removeEventListener(RuleFlowEventListener) : void
removeEventListener(WorkingMemoryEventListener) : void
org.drools.event
AgendaEventListener
RuleFlowEventListener
WorkingMemoryEventListener
java.util
List<E>

**Figure 2.31. 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, rules can also be added and removed. Now what happens if we assume a stateless session, where after the initial data set no more data can be asserted or modified (no rule re-evaluations) and rules cannot be added or removed? This means 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). 4
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, adding the Object, and then propagate the Tuple – instead a Command Object is created and added to a list in the Working Memory. This Command Object holds a reference to the LeftInputAdapterNode and the propagated Object. This stops any left-input propagations at insertion time, so that we know that a right-input propagation will never need to attempt a join with the left-inputs (removing the need for left-input memory). All nodes have their memory turned off, including the left-input Tuple memory but excluding the right-input Object memory – i.e. 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.setSequential 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

# Chapter 3. Decision tables in spreadsheets

Decision tables are a "precise yet compact" (ref. Wikipedia) way of representing conditional logic, and are well suited to *business* level rules.

Drools supports managing rules in a Spreadsheet format. Formats supported are Excel, and CSV. Meaning that a variety of spreadsheet programs (such as Microsoft Excel, OpenOffice.org Calc amongst others) can be utalized. It is expected that web based decision table editors will be included in a near future release.

Decision tables are an old concept (in software terms) but have proven useful over the years. Very briefly speaking, in Drools decision tables are a way to generate rules driven from the data entered into a spreadsheet. All the usual features of a spreadsheet for data capture and manipulation can be taken advantage of.

## 3.1. When to use Decision tables

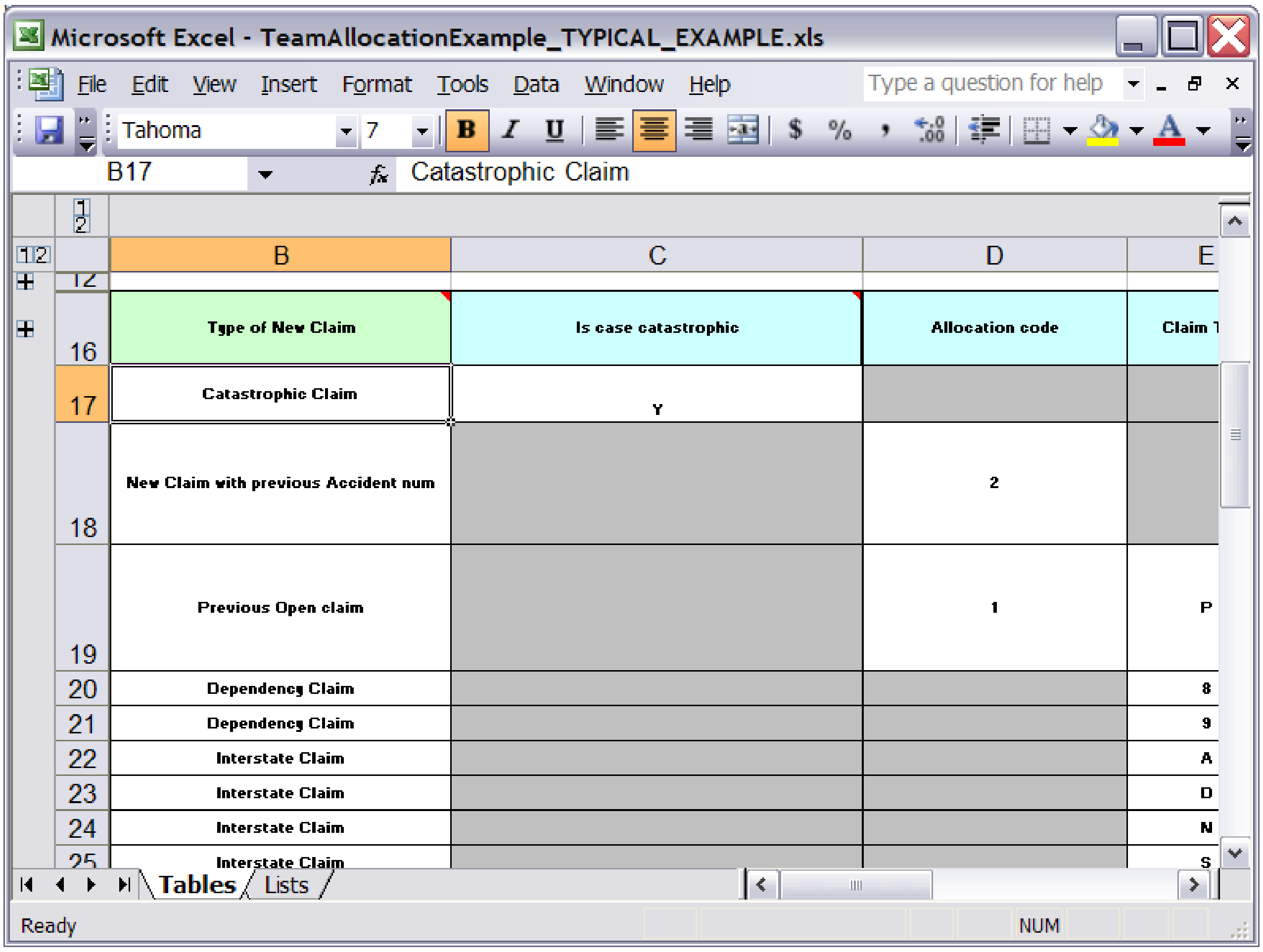
Decision tables my want to be considered as a course of action if rules exist that can be expressed as rule templates + data. In each row of a decision table, data is collected that is combined with the templates to generate a rule.

Many businesses already use spreadsheets for managing data, calculations etc. If you are happy to continue this way, you can also manage your business rules this way. This also assumes you are happy to manage packages of rules in .xls or .csv files. Decision tables are not recommenced for rules that do not follow a set of templates, or where there are a small number of rules (or if there is a dislike towards software like excel or open office). They are ideal in the sense that there can be control over what *parameters* of rules can be edited, without exposing the rules directly.

Decision tables also provide a degree of insulation from the underlying object model.

## 3.2. Overview

Here are some examples of real world decision tables (slightly edited to protect the innocent).

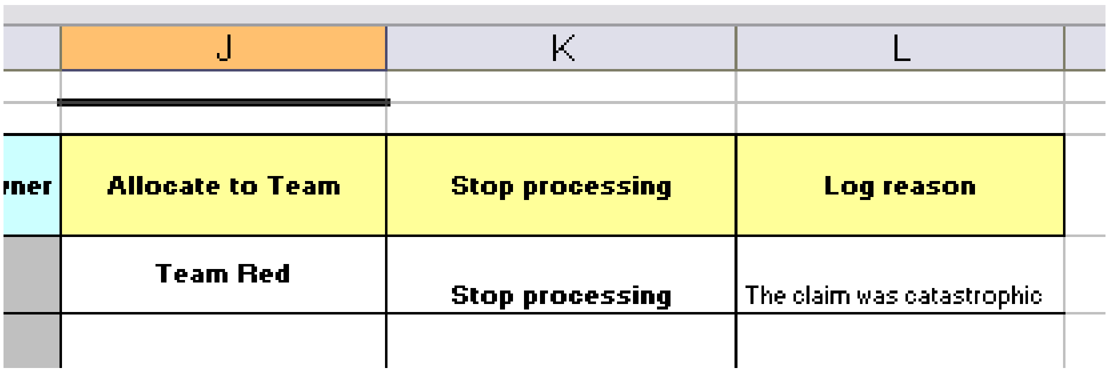

Microsoft Excel - TeamAllocationExample_TYPICAL_EXAMPLE.xls

B17 Catastrophic Claim

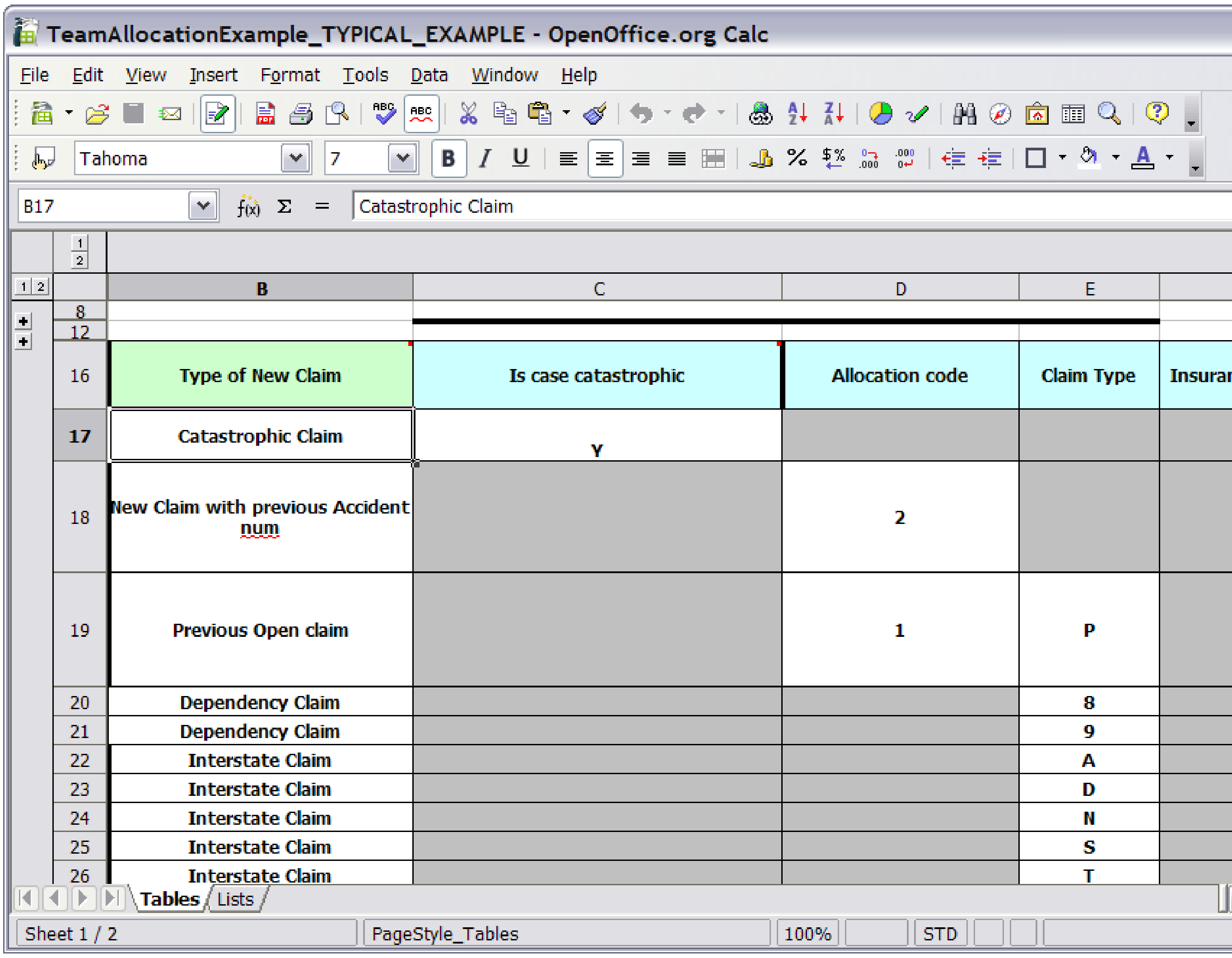
| | B | C | D | E |
|---|---|---|---|---|
| 12 | | | | |
| 16 | Type of New Claim | Is case catastrophic | Allocation code | Claim T |
| 17 | Catastrophic Claim | Y | | |
| 18 | New Claim with previous Accident num | | 2 | |
| 19 | Previous Open claim | | 1 | P |
| 20 | Dependency Claim | | | 8 |
| 21 | Dependency Claim | | | 9 |
| 22 | Interstate Claim | | | A |
| 23 | Interstate Claim | | | D |
| 24 | Interstate Claim | | | N |
| 25 | Interstate Claim | | | S |

Tables / Lists

Ready NUM



| | J | K | L |
|---|---|---|---|
| ner | Allocate to Team | Stop processing | Log reason |
| | Team Red | Stop processing | The claim was catastrophic |
| | | | |


TeamAllocationExample_TYPICAL_EXAMPLE - OpenOffice.org Calc
File Edit View Insert Format Tools Data Window Help
Tahoma
7
B17
Catastrophic Claim
Tables
Lists
Sheet 1 / 2
PageStyle_Tables
100%
STD

| | B | C | D | E | |
|---|---|---|---|---|---|
| 16 | Type of New Claim | Is case catastrophic | Allocation code | Claim Type | Insurar |
| 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 | |

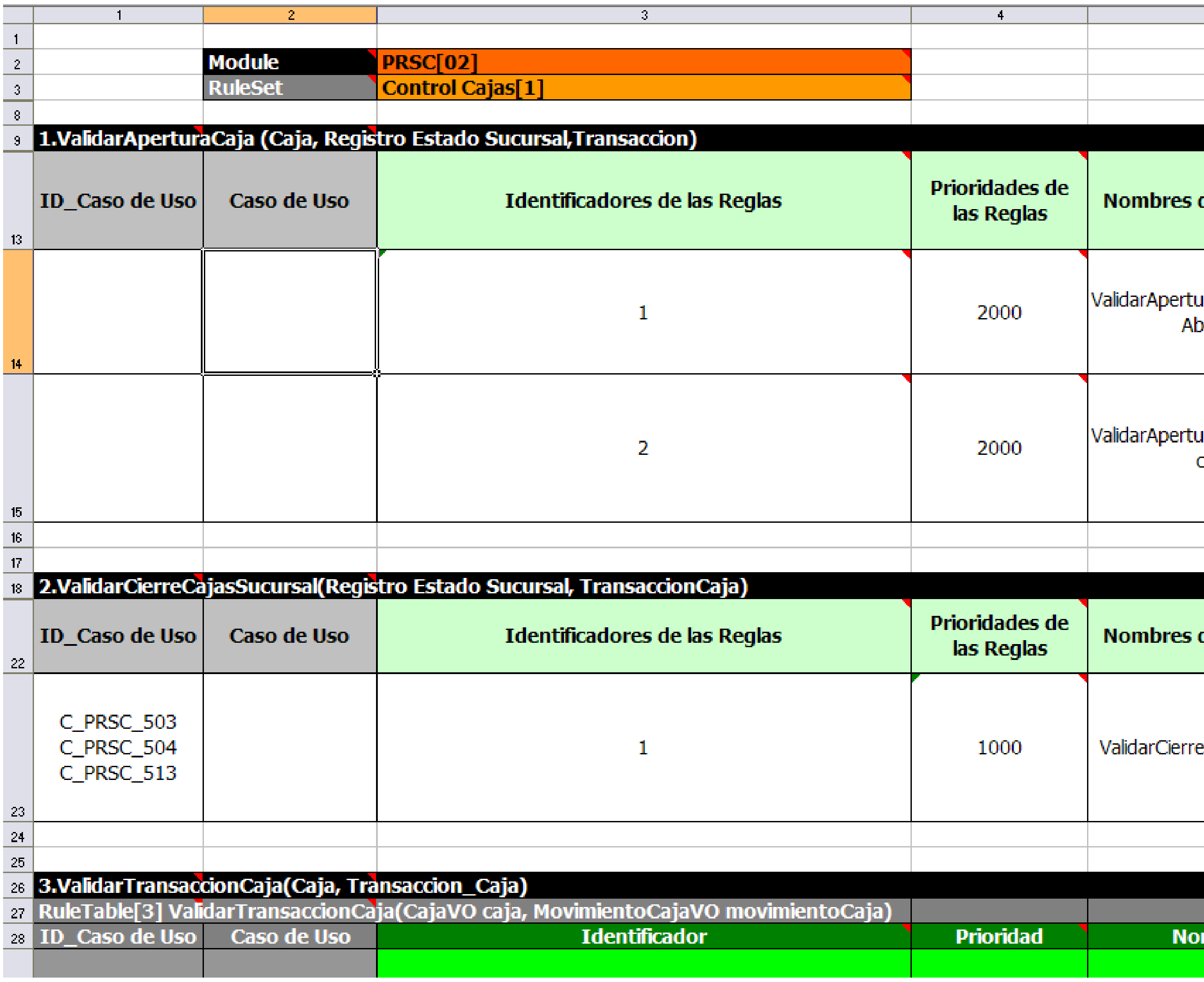
In the above examples, the technical aspects of the decision table have been collapsed away (standard spreadsheet feature).

The rules start from row 17 (each row results in a rule). The conditions are in column C, D, E etc.. (off screen are the actions). The value in the cells are quite simple, and have meaning when looking at the headers in Row 16. Column B is just a description. It is conventional to use color to make it obvious what the different areas of the table mean.

**Note**

Note that although the decision tables look like they process top down, this is not necessarily the case. Idealy, if the rules are able to be authored in such a way as order does not matter (simply as it makes maintenance easier, as rows will not need to be shifted around all the time).

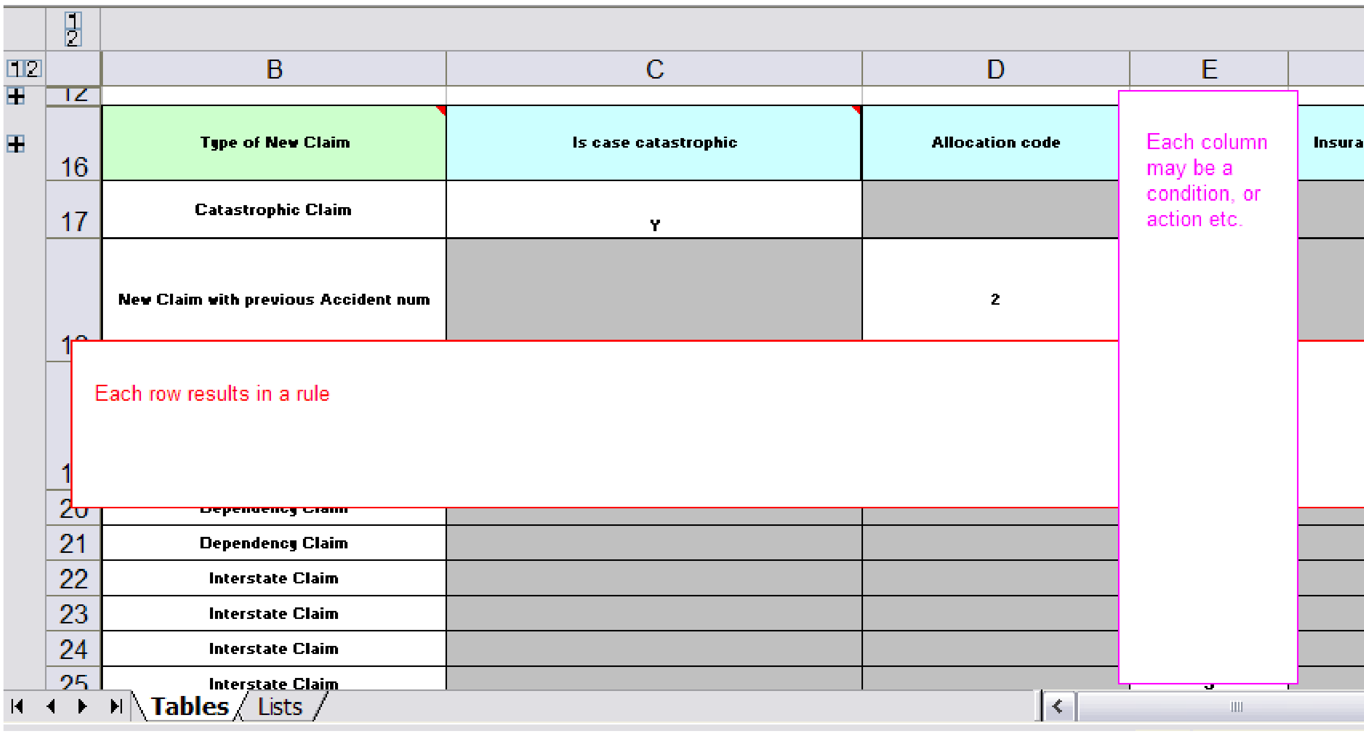
As each row is a rule, the same principles apply. As the rule engine processes the facts, any rules that match may fire (some people are confused by this. It is possible to clear the agenda when a rule fires and simulate a very simple decision table where the first match exists). Also note that you can have multiple tables on the one spreadsheet (so rules can be grouped where they share common templates, yet at the end of the day they are all combined into a one rule package). Decision tables are essentially a tool to generate DRL rules automatically.



| | 1 | 2 | 3 | 4 | |
|---|---|---|---|---|---|
| 1 | | | | | |
| 2 | | Module | PRSC[02] | | |
| 3 | | RuleSet | Control Cajas[1] | | |
| 8 | | | | | |
| 9 | 1.ValidarAperturaCaja (Caja, Registro Estado Sucursal,Transaccion) | | | | |
| 13 | ID_Caso de Uso | Caso de Uso | Identificadores de las Reglas | Prioridades de las Reglas | Nombres d |
| 14 | | | 1 | 2000 | ValidarApertu Ab |
| 15 | | | 2 | 2000 | ValidarApertu C |
| 16 | | | | | |
| 17 | | | | | |
| 18 | 2.ValidarCierreCajasSucursal(Registro Estado Sucursal, TransaccionCaja) | | | | |
| 22 | ID_Caso de Uso | Caso de Uso | Identificadores de las Reglas | Prioridades de las Reglas | Nombres d |
| 23 | C_PRSC_503 C_PRSC_504 C_PRSC_513 | | 1 | 1000 | ValidarCierre |
| 24 | | | | | |
| 25 | | | | | |
| 26 | 3.ValidarTransaccionCaja(Caja, Transaccion_Caja) | | | | |
| 27 | RuleTable[3] ValidarTransaccionCaja(CajaVO caja, MovimientoCajaVO movimientoCaja) | | | | |
| 28 | ID_Caso de Uso | Caso de Uso | Identificador | Prioridad | Nor |
| | | | | | |

## 3.3. How decision tables work

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

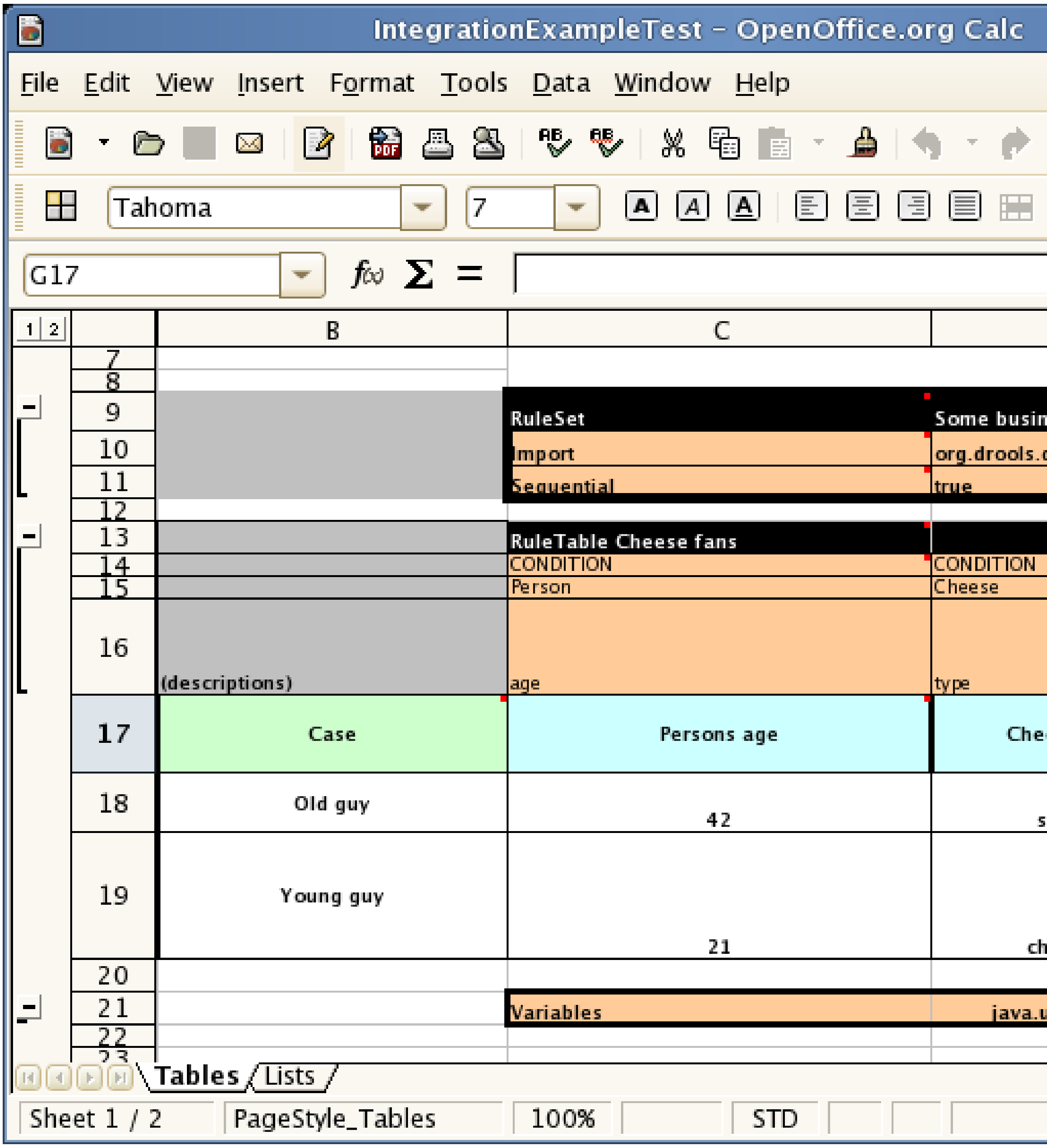


| | B | C | D | E |
|---|---|---|---|---|
| 16 | Type of New Claim | Is case catastrophic | Allocation code | Insura |
| 17 | Catastrophic Claim | Y | | |
| 18 | New Claim with previous Accident num | | 2 | |
| 20 | Dependency Claim | | | |
| 21 | Dependency Claim | | | |
| 22 | Interstate Claim | | | |
| 23 | Interstate Claim | | | |
| 24 | Interstate Claim | | | |
| 25 | Interstate Claim | | | |

Each row results in a rule
Each column may be a condition, or action etc.
Tables
Lists

The spreadsheet looks for the *RuleTable* keyword to indicate the start of a rule table (both the starting row and column). Other keywords are also used to define other package level attributes (covered later). It is important to keep the keywords in the one column. By convention the second column ("B") is used for this, but it can be any column (convention is to leave a margin on the left for notes). In the following diagram, C is actually the column where it starts. Everything to the left of this is ignored.

If we expand the hidden sections, it starts to make more sense how it works; note the keywords in column C.


IntegrationExampleTest - OpenOffice.org Calc
File Edit View Insert Format Tools Data Window Help
Tahoma
7
G17
Tables
Lists
Sheet 1 / 2
PageStyle_Tables
100%
STD

| | B | C | |
|---|---|---|---|
| 7 | | | |
| 8 | | | |
| 9 | | RuleSet | Some busin |
| 10 | | Import | org.drools.c |
| 11 | | Sequential | true |
| 12 | | | |
| 13 | | RuleTable Cheese fans | |
| 14 | | CONDITION | CONDITION |
| 15 | | Person | Cheese |
| 16 | (descriptions) | age | type |
| 17 | Case | Persons age | Che |
| 18 | Old guy | 42 | s |
| 19 | Young guy | 21 | ch |
| 20 | | | |
| 21 | | Variables | java.u |
| 22 | | | |

Now the hidden magic which makes it work can be seen. The RuleSet keyword indicates the name to be used in the *rule package* that all the rules will come under (the name is optional, it will have a default but it MUST have the *RuleSet* keyword) in the cell immediately to the right.

The other keywords visible in Column C are: Import, Sequential which will be covered later. The RuleTable keyword is important as it indicates that a chunk of rules will follow, based on some rule templates. After the RuleTable keyword there is a name - this name is used to prefix the generated rules names (the row numbers are appended to create unique rule names). The column of RuleTable indicates the column in which the rules start (columns to the left are ignored).

**Note**

In general the keywords make up name/value pairs.

Referring to row 14 (the row immediately after RuleTable): the keywords CONDITION and ACTION indicate that the data in the columns below are for either the LHS or the RHS parts of a rule. There are other attributes on the rule which can also be optionally set this way.

Row 15 contains declarations of *ObjectTypes* ; the content in this row is optional (if this option is not in use, a blank row must be left, however this option is usually found to be quite useful). When using this row, the values in the cells below (row 16) become constraints on that object type. In the above case, it will generate: Person(age=="42") etc (where 42 comes from row 18). In the above example, the "==" is implicit (if just a field name is given it will assume that it is to look for exact matches).

**Note**

An ObjectType declaration can span columns (via merged cells), meaning that all columns below the merged range will be combined into the one set of constraints.

Row 16 contains the rule templates themselves. They can use the "$para" place holder to indicate where data from the cells below will be populated ($param can be sued or $1, $2 etc to indicate parameters from a comma separated list in a cell below). Row 17 is ignored as it is textual descriptions of the rule template.

Row 18 to 19 shows data, which will be combined (interpolated) with the templates in row 15, to generate rules. If a cell contains no data, then its template is ignored (eg it means that condition, or action, does not apply for that rule-row). Rule rows are read until there is a BLANK row. Multiple RuleTables can exsist in a sheet. Row 20 contains another keyword, and a value. The row positions of keywords like this do not matter (most people put them at the top) but their column should be the same one where the RuleTable or RuleSet keywords should appear (in this case column C has been chosen to be significant, but column A could be used instead).

In the above example, rules would be rendered like the following (as it uses the "ObjectType" row):

```
//row 18
   rule "Cheese_fans_18"
   when
   Person(age=="42")
   Cheese(type=="stilton")
   then
```

```
    list.add("Old man stilton");
    end
```

**Note**

The [age=="42"] and [type=="stilton"] are interpreted as single constraints to be added to the respective ObjectType in the cell above (if the cells above were spanned, then there could be multiple constraints on one "column".

## 3.4. Keywords and syntax

### 3.4.1. Syntax of templates

The syntax of what goes in the templates is dependent on if it is a CONDITION column or ACTION column. In most cases, it is identical to *vanilla* DRL for the LHS or RHS respectively. This means in the LHS, the constraint language must be used, and in the RHS it is a snippet of code to be executed.

The `$param` place holder is used in templates to indicate where data form the cell will be interpolated. You can also use `$1` to the same effect. If the cell contains a comma separated list of values, $1 and $2 etc. may be used to indicate which positional parameter from the list of values in the cell will be used.

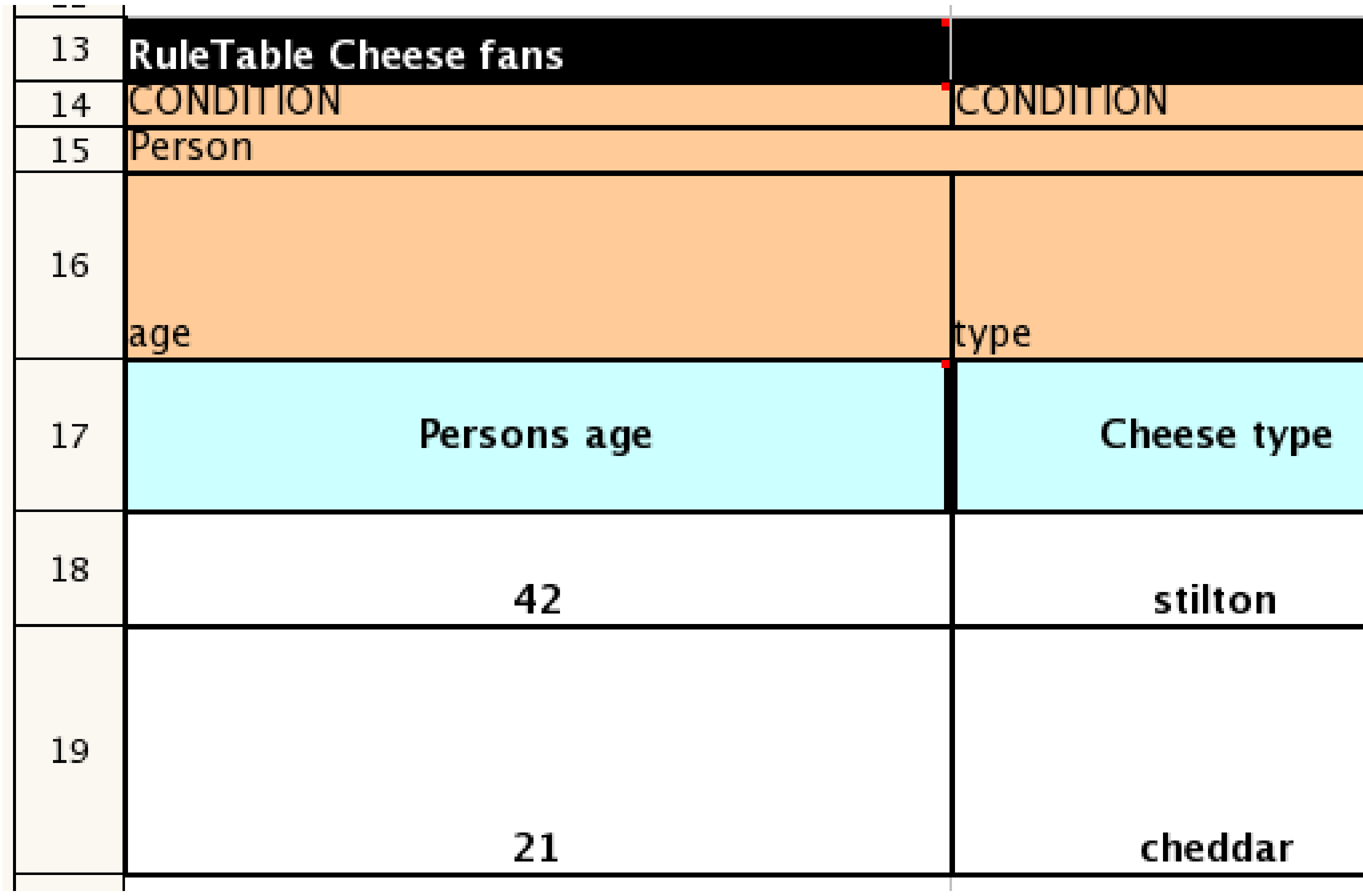
**Example 3.1.**

If the templates is [Foo(bar == $param)] and the cell is [ 42 ] then the result will be [Foo(bar == 42)] If the template is [Foo(bar < $1, baz == $2)] and the cell is [42,42] then the result will be [Foo(bar > 42, baz ==42)]

For conditions: How snippets are rendered depends on if there is anything in the row above (where ObjectType declarations may appear). If there is, then the snippets are rendered as individual constraints on that ObjectType. If there isn't, then they are just rendered as is (with values substituted). If just a plain field is entered (as in the example above) then it will assume this means equality. If another operator is placed at the end of the snippet, then the values will put interpolated at the end of the constraint, otherwise it will look for `$param` as outlined previously.

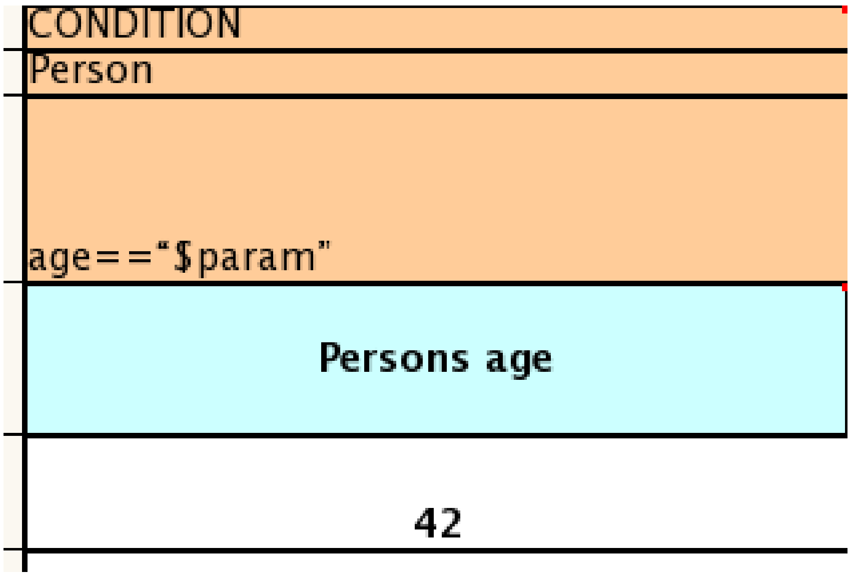
For consequences: How snippets are rendered also depends on if there is anything in the row immediately above it. If there is nothing there, the output is simple the interpolated snippets. If there is something there (which would typically be a bound variable or a global like in the example above) then it will append it as a method call on that object (refer to the above example).

This may be easiest to understand with some examples below.



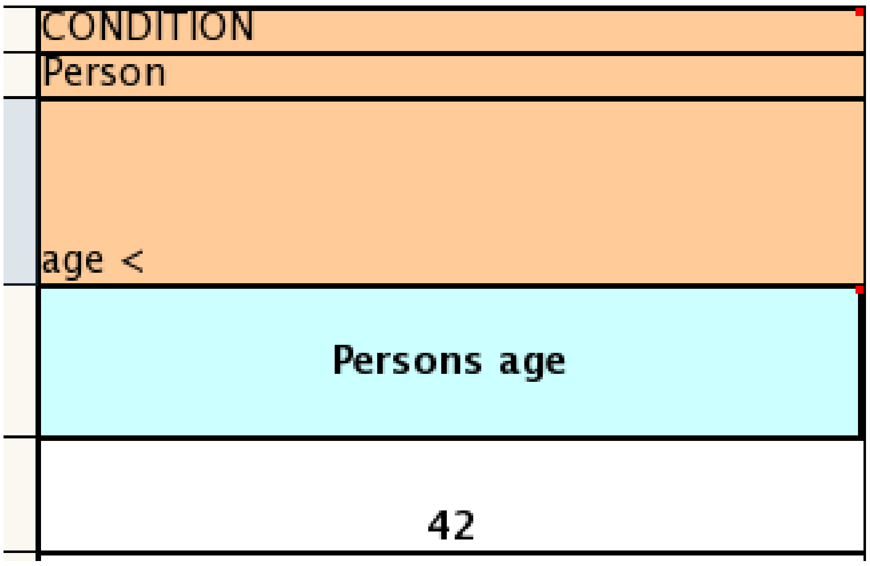
| | | |
|---|---|---|
| 13 | RuleTable Cheese fans | |
| 14 | CONDITION | CONDITION |
| 15 | Person | |
| 16 | age | type |
| 17 | Persons age | Cheese type |
| 18 | 42 | stilton |
| 19 | 21 | cheddar |

The above shows how the Person ObjectType declaration spans 2 columns in the spreadsheet, thus both constraints will appear as Person(age == ... , type == ...). As before, as only the field names are present in the snippet, they imply an equality test.



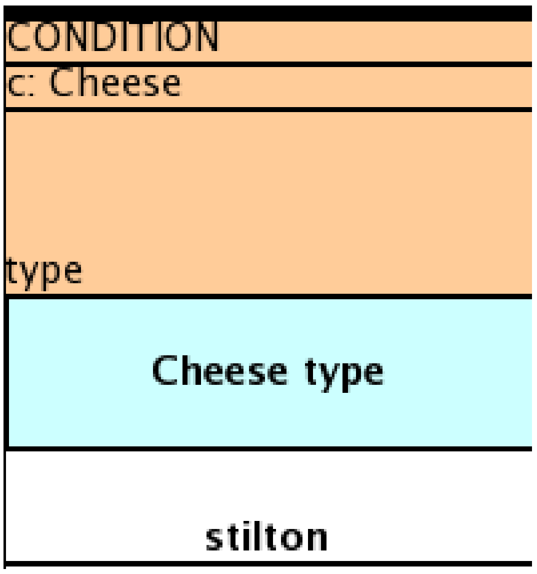
| |
|---|
| CONDITION |
| Person |
| age=="$param" |
| Persons age |
| 42 |

The above condition example shows how you use interpolation to place the values in the snippet (in this case it would result in Person(age == "42")).



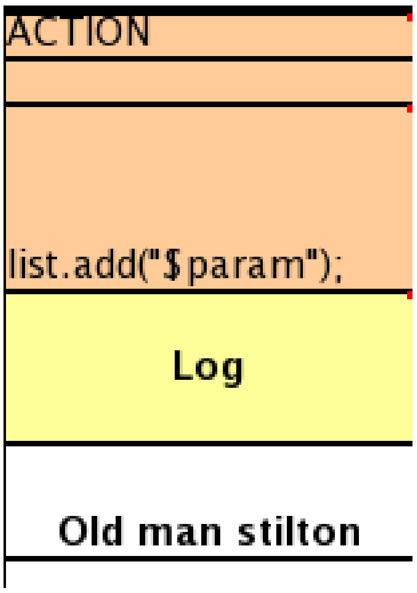
| |
|---|
| CONDITION |
| Person |
| age < |
| Persons age |
| 42 |

The above condition example show that if you put an operator on the end by itself, the values will be placed after the operator automatically.



| |
|---|
| CONDITION |
| c: Cheese |
| type |
| Cheese type |
| stilton |

A binding can be put in before the column (the constraints will be added from the cells below). Anything can be placed in the ObjectType row (eg it could be a pre condition for the columns in the spreadsheet columns that follow).



ACTION

list.add("$param");

Log

Old man stilton

This shows how the consequence could be done the by simple interpolation (just leave the cell above blank, the same applies to condition columns). With this style anything can be placed in the consequence (not just one method call).

### 3.4.2. Keywords

The following table describes the keywords that are pertinent to the rule table structure.

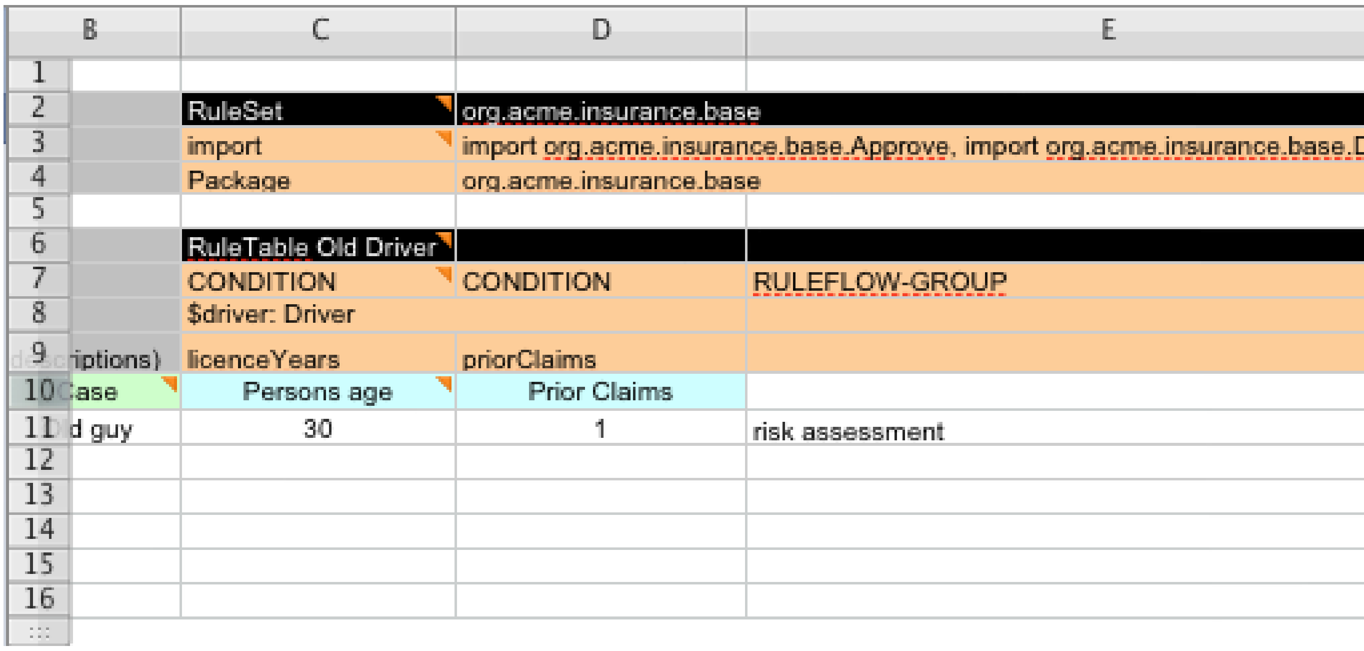
**Table 3.1. Keywords**

| Keyword | Description | Inclusion Status |
|---|---|---|
| RuleSet | The cell to the right of this contains the ruleset name | One only (if left out, it will default) |
| 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 | | At least one per rule table |

| Keyword | Description | Inclusion Status |
|---|---|---|
| | Indicates that this column will be for rule conditions | |
| 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. Overrides 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. Drools supports functions defined in the DRL, allowing logic to be embedded in the rule, and changed without hard coding, use with care. Same syntax as regular DRL. | optional |
| Variables | The cell immediately to the right can contain global declarations which Drools supports. This is a type, followed by a variable name. (if multiple variables are needed, comma separate them). | optional |
| No-loop or Unloop | Placed in the header of a table, no-loop or unloop will both complete the same function of not allowing a rule (row) to loop. For this option to function correctly, there must be a value (true or false) in the cell for the option to take effect. If the cell is left blank then this | optional |

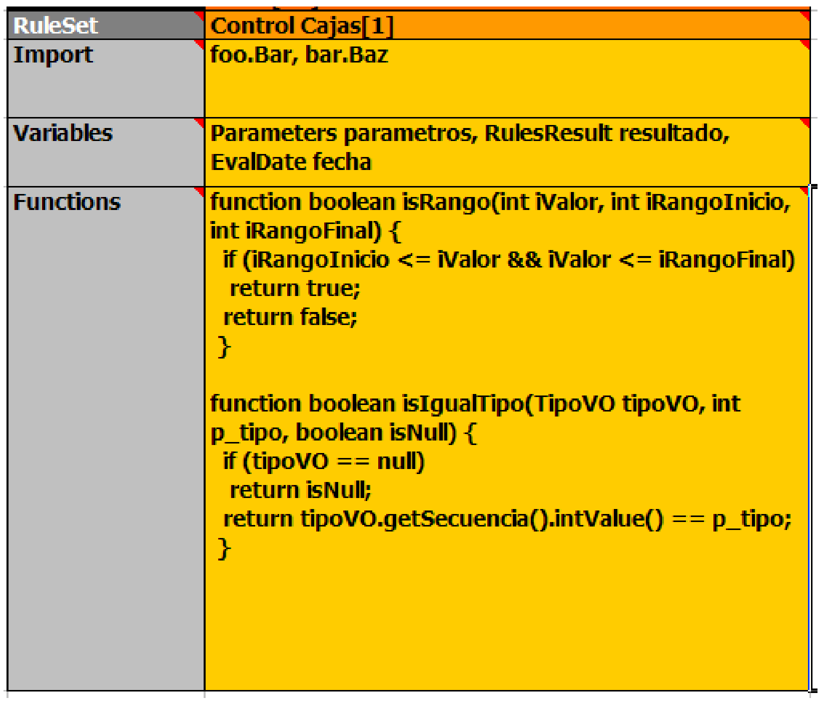
| Keyword | Description | Inclusion Status |
|---|---|---|
| | option will not be set for the row. | |
| XOR-GROUP | Cell values in this column mean that the rule-row belongs to the given XOR/ activation group . An Activation group means that only one rule in the named group will fire (ie the first one to fire cancels the other rules activations). | optional |
| AGENDA-GROUP | Cell values in this column mean that the rule-row belongs to the given agenda group (that is one way of controlling flow between groups of rules - see also "rule flow"). | optional |
| RULEFLOW-GROUP | Cell values in this column mean that the rule-row belongs to the given rule-flow group. | optional |
| Worksheet | By default, the first worksheet is only looked at for decision tables. | N/A |

Below you will find examples of using the HEADER keywords, which effects the rules generated for each row. Note that the header name is what is important in most cases. If no value appears in the cells below it, then the attribute will not apply (it will be ignored) for that specific row.



| | B | C | D | E |
|---|---|---|---|---|
| 1 | | | | |
| 2 | | RuleSet | org.acme.insurance.base | |
| 3 | | import | import org.acme.insurance.base.Approve, import org.acme.insurance.base.D | |
| 4 | | Package | org.acme.insurance.base | |
| 5 | | | | |
| 6 | | RuleTable Old Driver | | |
| 7 | | CONDITION | CONDITION | RULEFLOW-GROUP |
| 8 | | $driver: Driver | | |
| 9 | d criptions) | licenceYears | priorClaims | |
| 10 | Case | Persons age | Prior Claims | |
| 11 | d guy | 30 | 1 | risk assessment |
| 12 | | | | |
| 13 | | | | |
| 14 | | | | |
| 15 | | | | |
| 16 | | | | |

The following is an example of Import (comma delimited), Variables (gloabls) - also comma delimited, and a function block (can be multiple functions - just the usual drl syntax). This can appear in the same column as the "RuleSet" keyword, and can be below all the rule rows if you desire.



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

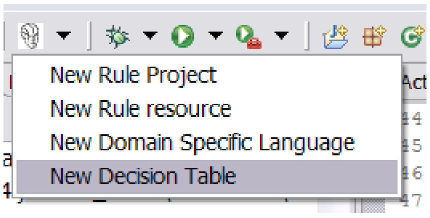
Functions

```
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;
 }
```

## 3.5. Creating and integrating Spreadsheet based Decision Tables

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

To get started, a sample spreadsheet can be used as base. Alternatively, if the plug-in is being used (Rule Workbench IDE), the wizard can generate a spreadsheet from a template (to edit it an xls compatible spreadsheet editor will need to be used).



## 3.6. Managing business rules in decision tables.

### 3.6.1. 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.

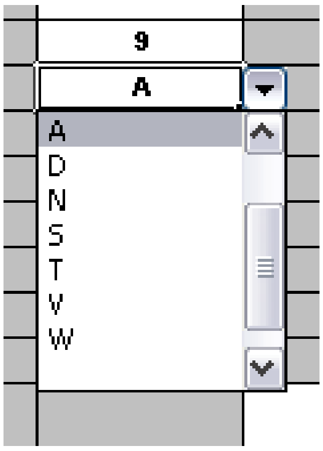
Typically, the whole process of authoring rules (coming up with a new decision table) would be something like:

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.

### 3.6.2. Using spreadsheet features

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



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

## 3.7. Rule Templates

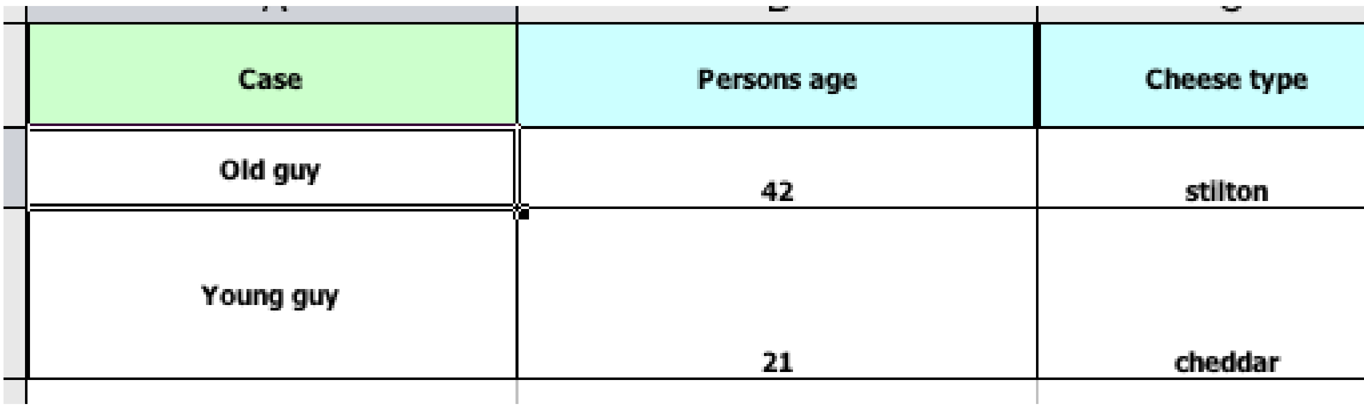
Related to decision tables (but not necessarily requiring a spreadsheet) are "Rule Templates" (in the drools-templates module). These use any tablular data source as a source of rule data - populating a template to generate many rules. This can allow both for more flexible spreadsheets, but also rules in existing databases, for instance (at the cost of developing the template up front to generate the rules).

With Rule Templates the data is separated from the rule and there are no restrictions on which part of the rule is data-driven. So whilst you can do everything you could do in decision tables you can also do the following:

- store your data in a database (or any other format)
- conditionally generate rules based on the values in the data
- use data for any part of your rules (e.g. condition operator, class name, property name)
- run different templates over the same data

### 3.7.1. A decision table-like example

As an example, a more classic decision table is shown, but without any hidden rows for the rule meta data (so the spreadsheet only contains the raw data to generate the rules).



| Case | Persons age | Cheese type |
|---|---|---|
| Old guy | 42 | stilton |
| Young guy | 21 | cheddar |

See the "ExampleCheese.xls" in the examples download for the above spreadsheet.

If this was a regular decision table there would be hidden rows before row 1 and between rows 1 and 2 containing rule metadata. With rule templates the data is completely separate from the rules. This has two handy consequences - you can apply multiple rule templates to the same data and your data is not tied to your rules at all. So what does the template look like?

```
template header
age
type
log

package org.drools.examples.templates;

global java.util.List list;

template "cheesefans"

rule "Cheese fans_@{row.rowNumber}"
when
   Person(age == @{age})
   Cheese(type == "@{type}")
then
   list.add("@{log}");
end

end template
```

Referring to the above:

```
Line 1: all rule templates start with "template header"
Lines 2-4: following the header is the list of columns in the order they
 appear in the data. In this case we are calling the first column "age", the
 second "type" and the third "log".
Lines 5: empty line signifying the end of the column definitions
Lines 6-9: standard rule header text. This is standard rule DRL and will
 appear at the top of the generated DRL. Put the package statement and any
 imports and global definitions
Line 10: The "template" keyword signals the start of a rule template. There
 can be more than one template in a template file. The template should have
 a unique name.
Lines 11-18: The rule template - see below
Line 20: "end template" signifies the end of the template.
```

The rule templates rely on MVEL to do substitution using the syntax @{token_name}. There is currently one built-in expression, @{row.rowNumber} which gives a unique number for each row of data and enables you to generate unique rule names. For each row of data a rule will be generated with the values in the data substituted for the tokens in the template. With the example data above the following rule file would be generated:

```
package org.drools.examples.templates;

global java.util.List list;

rule "Cheese fans_1"
when
  Person(age == 42)
  Cheese(type == "stilton")
then
  list.add("Old man stilton");
end

rule "Cheese fans_2"
when
  Person(age == 21)
  Cheese(type == "cheddar")
then
  list.add("Young man cheddar");
end
```

The code to run this is simple:

```
//first we compile the spreadsheet with the template
//to create a whole lot of rules.
```

```
final ExternalSpreadsheetCompiler converter = new
 ExternalSpreadsheetCompiler();
//the data we are interested in starts at row 2, column 2 (e.g. B2)
final String drl = converter.compile(getSpreadsheetStream(),
 getRulesStream(), 2, 2);
```

We create an ExternalSpreadsheetCompiler object and use it to merge the spreadsheet with the rules. The two integer parameters indicate the column and row where the data actually starts - in our case column 2, row 2 (i.e. B2)

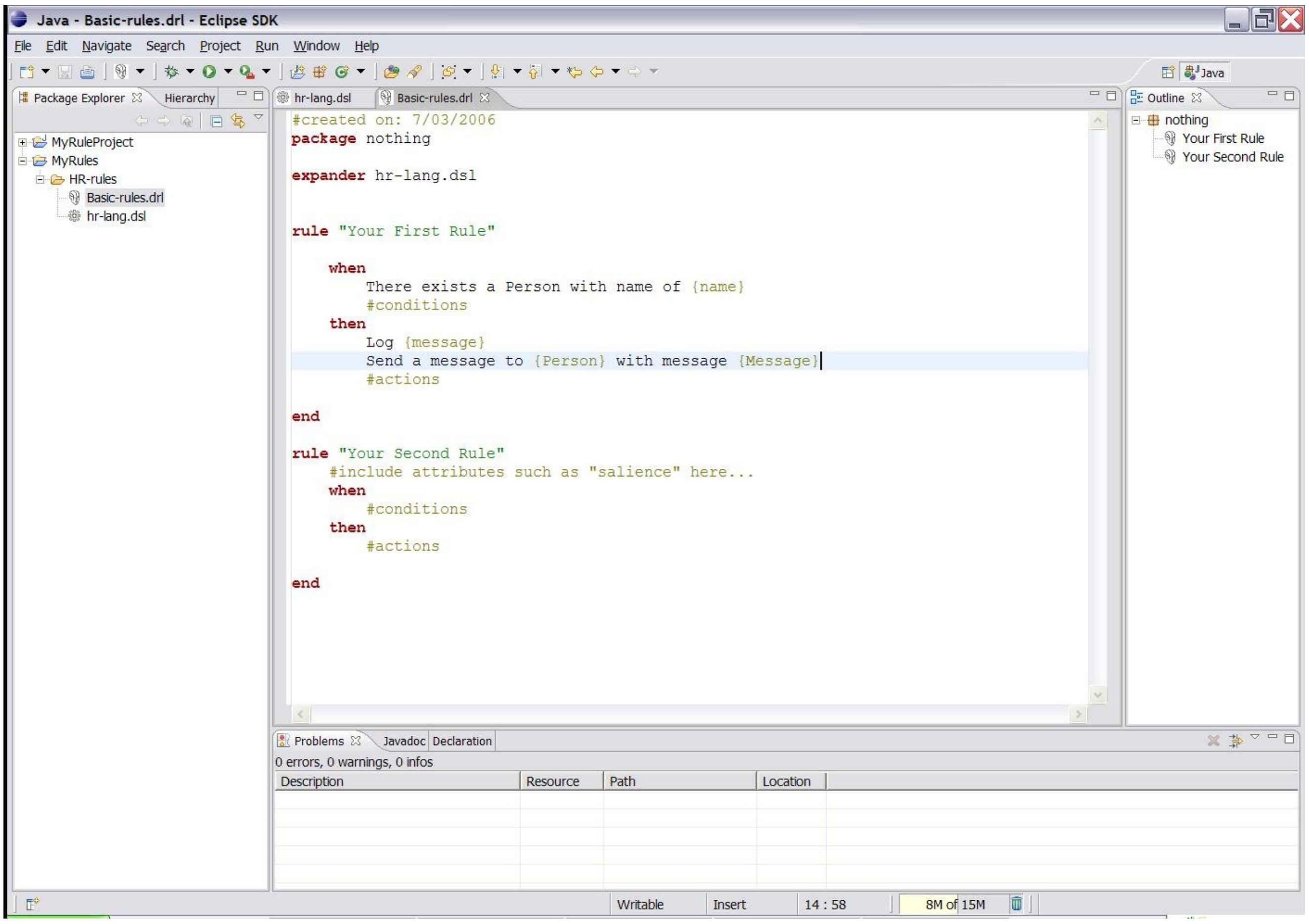
# Chapter 4. The (Eclipse based) Rule IDE

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. In cases where you prefer business rules and web tooling, you will want to look at the BRMS (but using the BRMS and the IDE together is not uncommon).

The Drools IDE is delivered as an Eclipse plug-in, 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 Red Hat 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).



```
#created on: 7/03/2006
package nothing

expander hr-lang.dsl


rule "Your First Rule"

    when
        There exists a Person with name of {name}
        #conditions
    then
        Log {message}
        Send a message to {Person} with message {Message}
        #actions

end

rule "Your Second Rule"
    #include attributes such as "salience" here...
    when
        #conditions
    then
        #actions

end
```

0 errors, 0 warnings, 0 infos

**Figure 4.1. Overview**

## 4.1. 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.

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

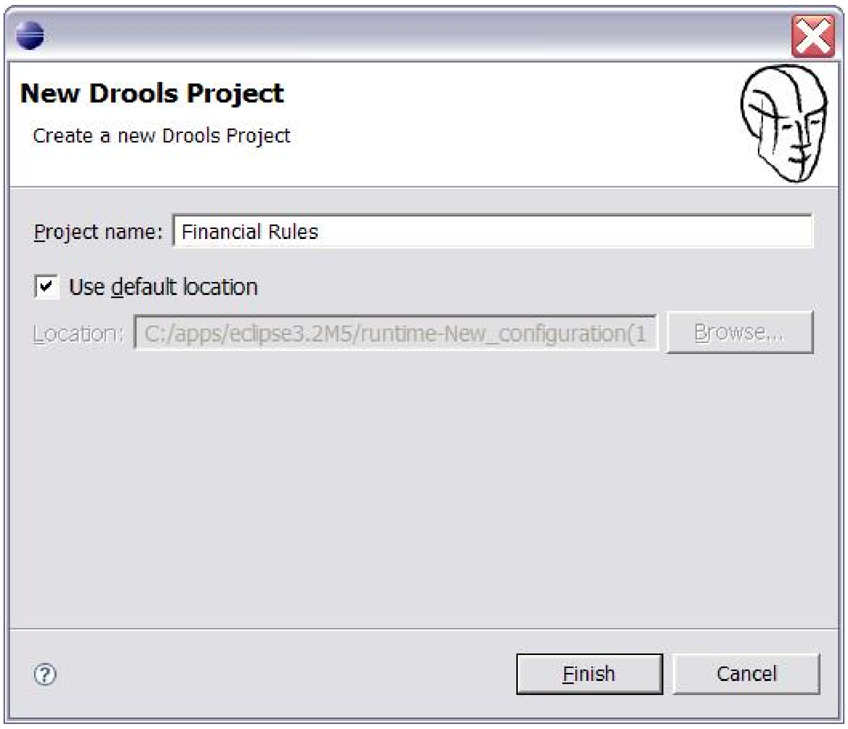
5. Rule validation

   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.

## 4.2. Creating a Rule project

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



New Drools Project

Create a new Drools Project

Project name: Financial Rules

[✓] Use default location

Location: C:/apps/eclipse3.2M5/runtime-New_configuration(1

**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, ruleflows etc). These can serve as a starting point, and will give you something executable to play with (which you can then modify and mould to your needs). The simplest case (a hello world rule) is shown below. Feel free to experiment with the plug-in at this point.

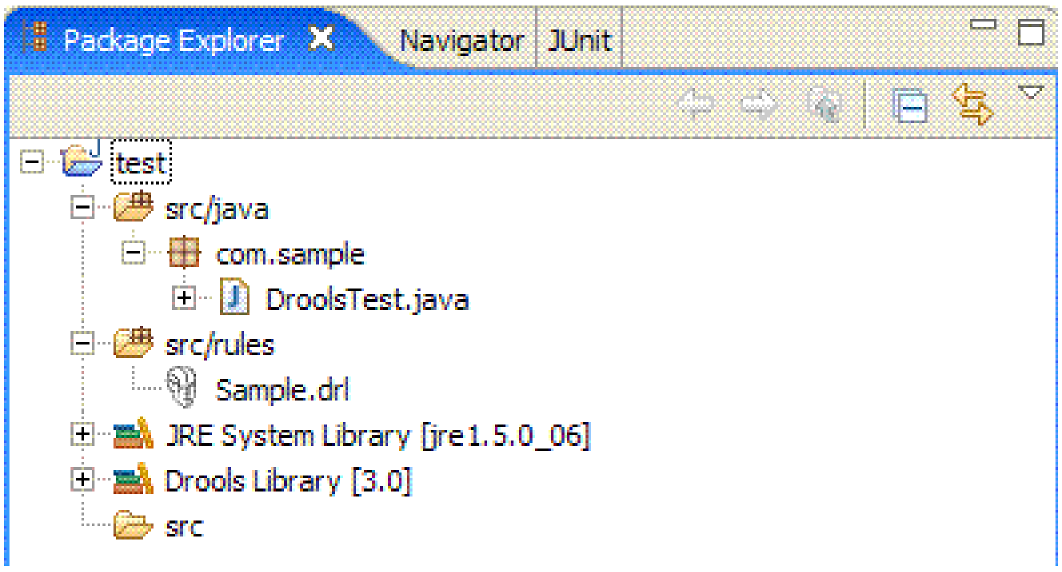


**Figure 4.3. New rule project result**

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.

## 4.3. 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).

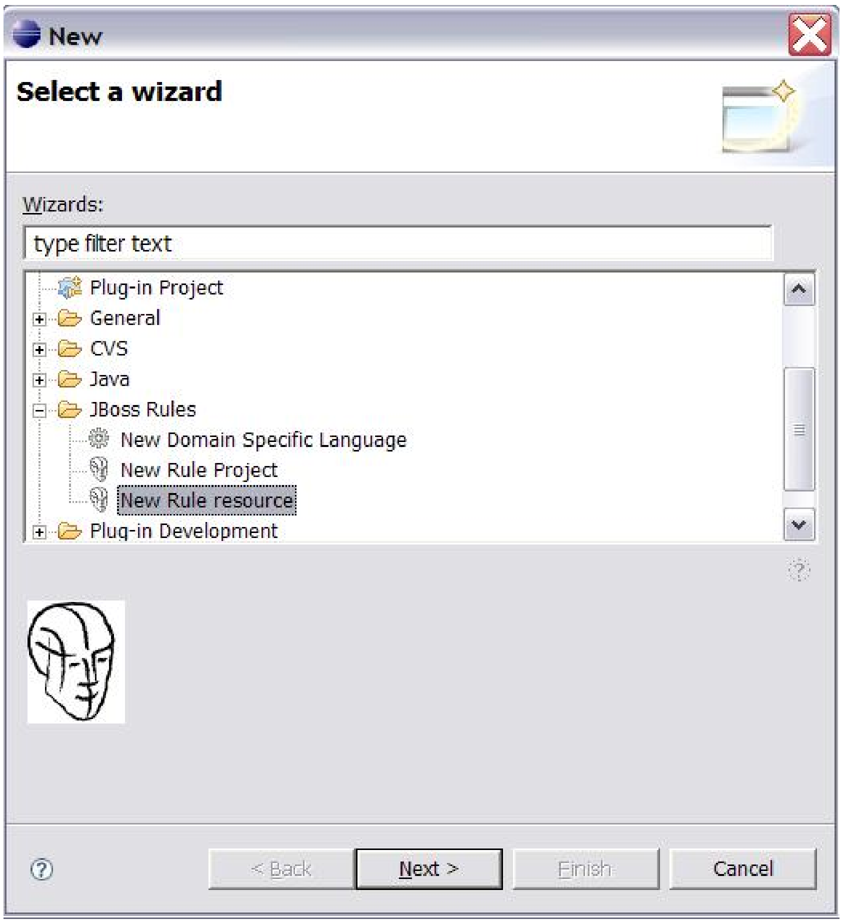
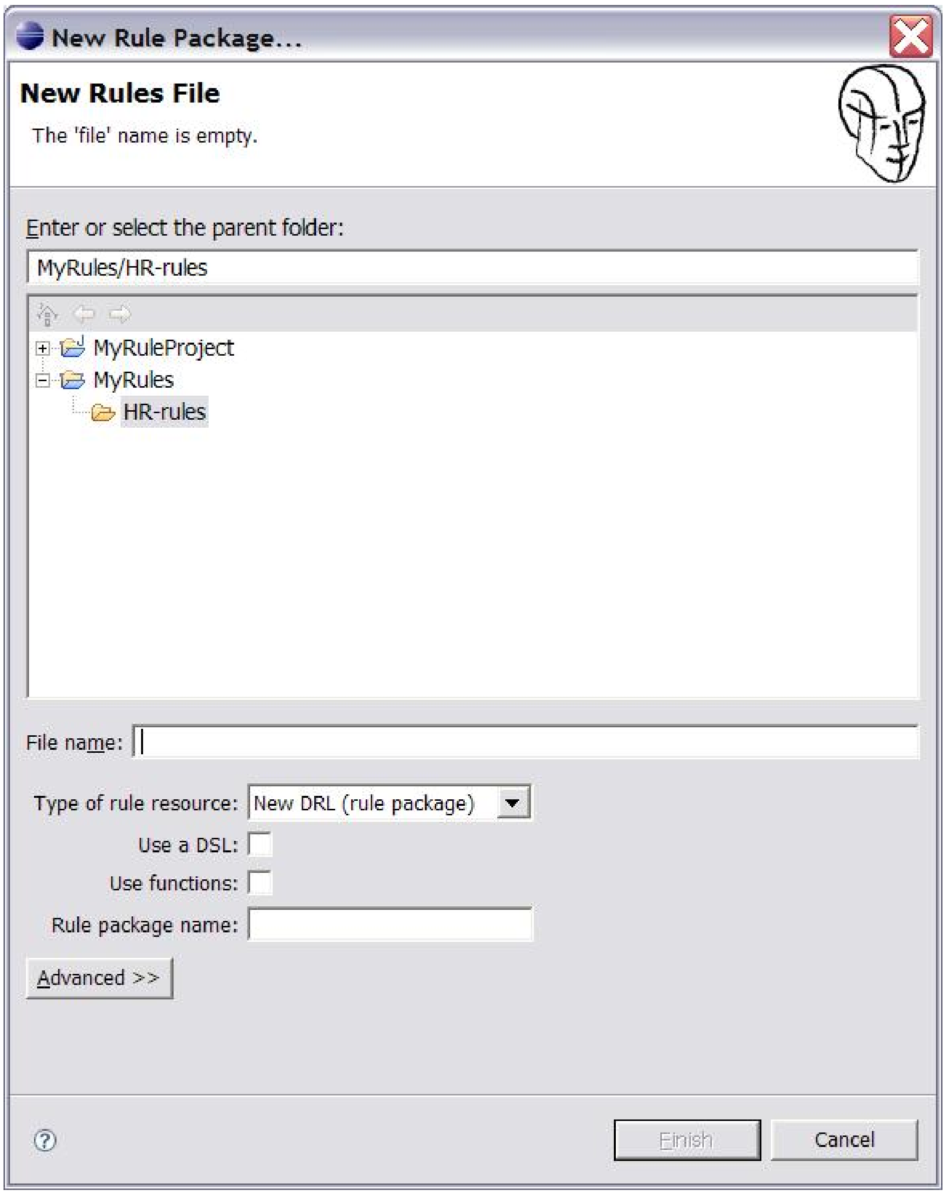


**Figure 4.4. The wizard menu**

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).


New Rule Package...

New Rules File

The 'file' name is empty.

Enter or select the parent folder:

MyRules/HR-rules

MyRuleProject

MyRules

HR-rules

File name:

Type of rule resource: New DRL (rule package)

Use a DSL:

Use functions:

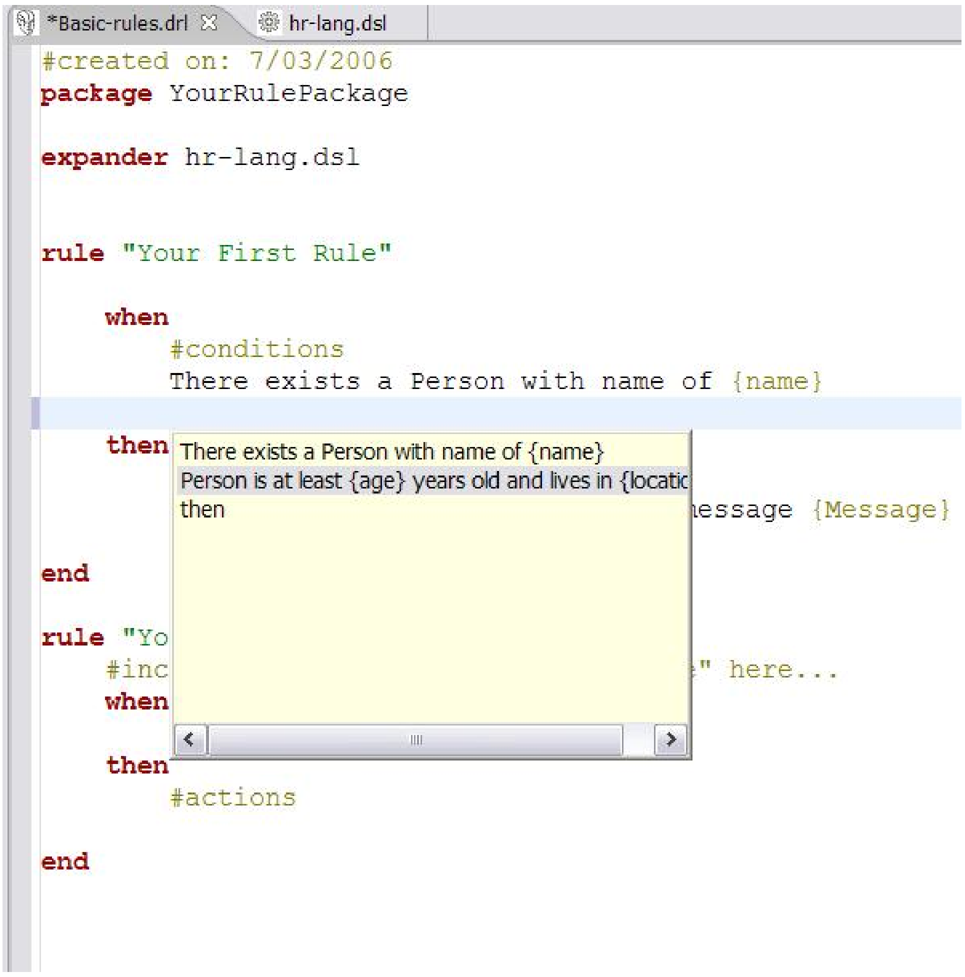
Rule package name:

**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 !

## 4.4. Textual rule editor

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



```
#created on: 7/03/2006
package YourRulePackage

expander hr-lang.dsl


rule "Your First Rule"

    when
        #conditions
        There exists a Person with name of {name}

    then
                                                  essage {Message}

end

rule "Yo
    #inc                                          " here...
    when

    then
        #actions

end
```

```
There exists a Person with name of {name}
Person is at least {age} years old and lives in {locatic
then
```

### Figure 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.

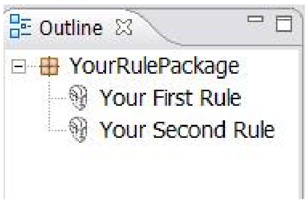


**Figure 4.7. The rule outline view**

## 4.5. Guided editor (rule GUI)

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

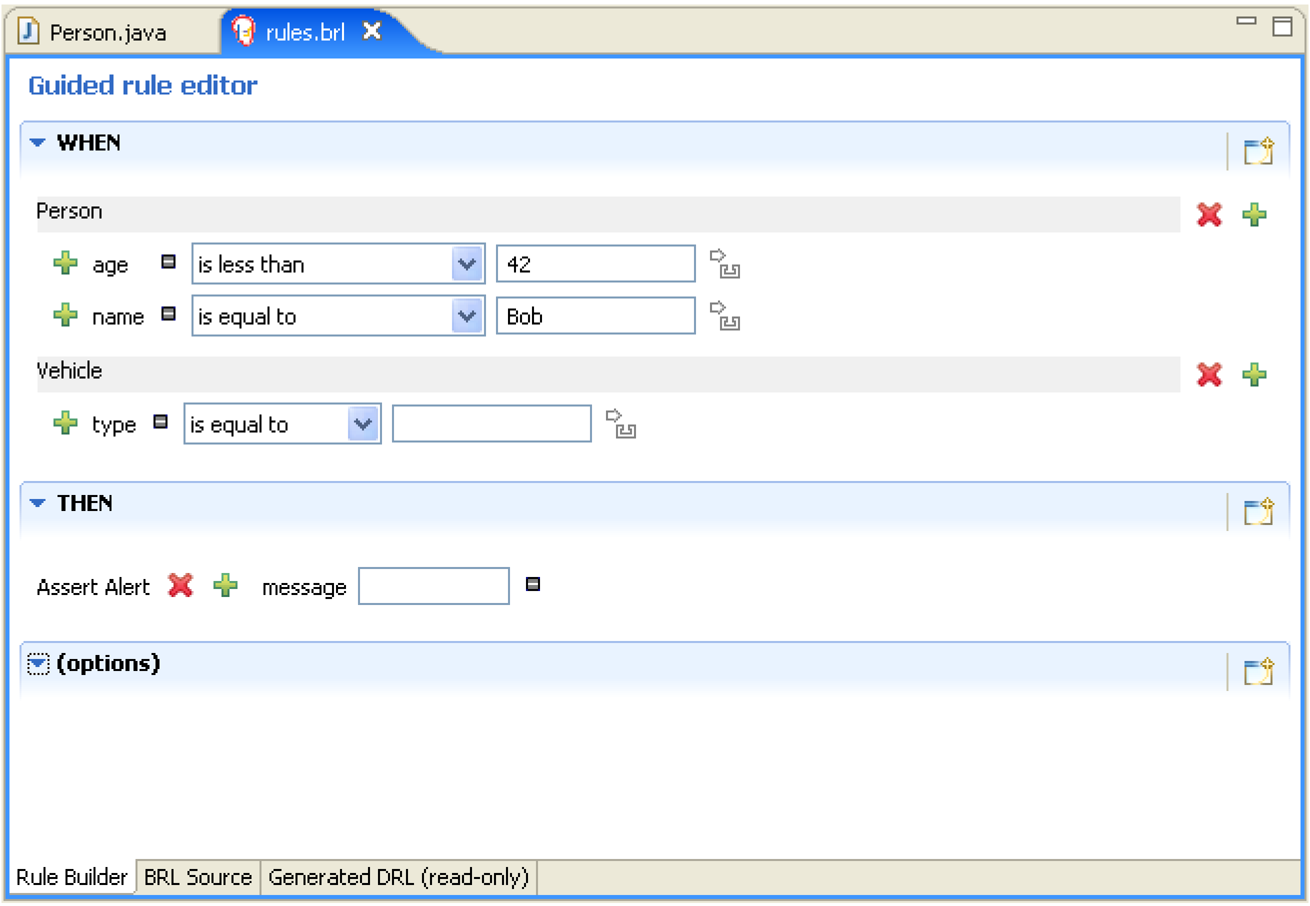


**Figure 4.8. The guided editor**

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
```

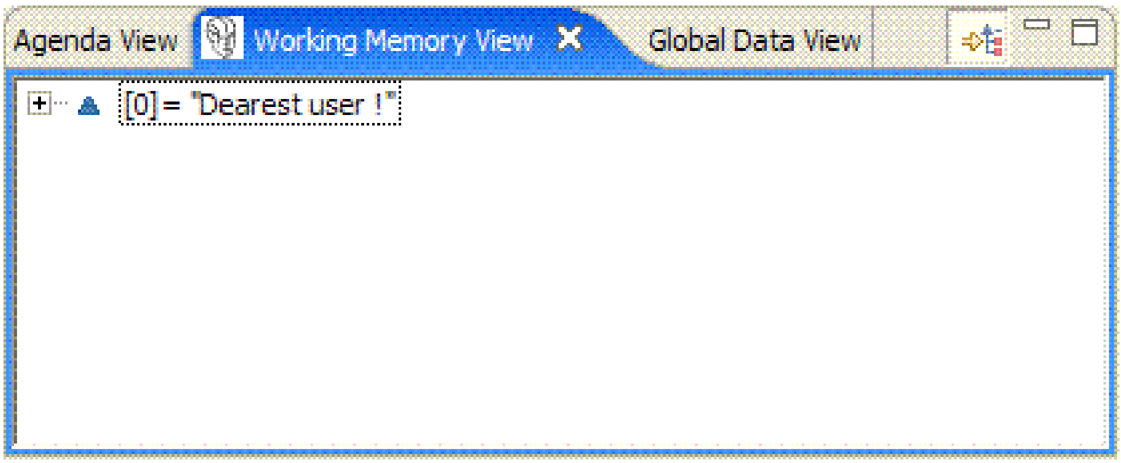
## 4.6. Views

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.

### 4.6.1. The Working Memory View

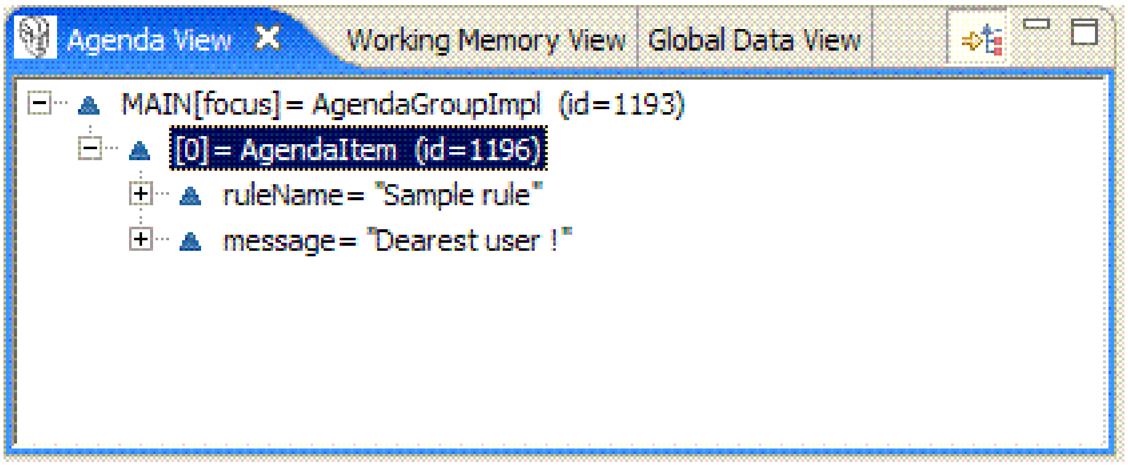


The Working Memory shows all elements in the working memory of the Drools engine.

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.

## 4.6.2. The Agenda View



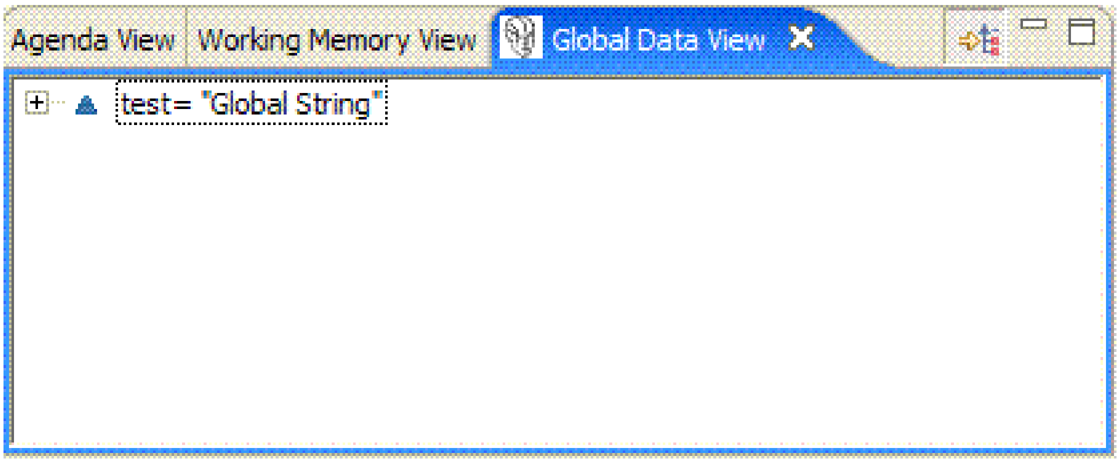
```
MAIN[focus] = AgendaGroupImpl  (id=1193)
    [0] = AgendaItem  (id=1196)
        ruleName= "Sample rule"
        message= "Dearest user !"
```

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

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

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

## 4.6.3. The Global Data View



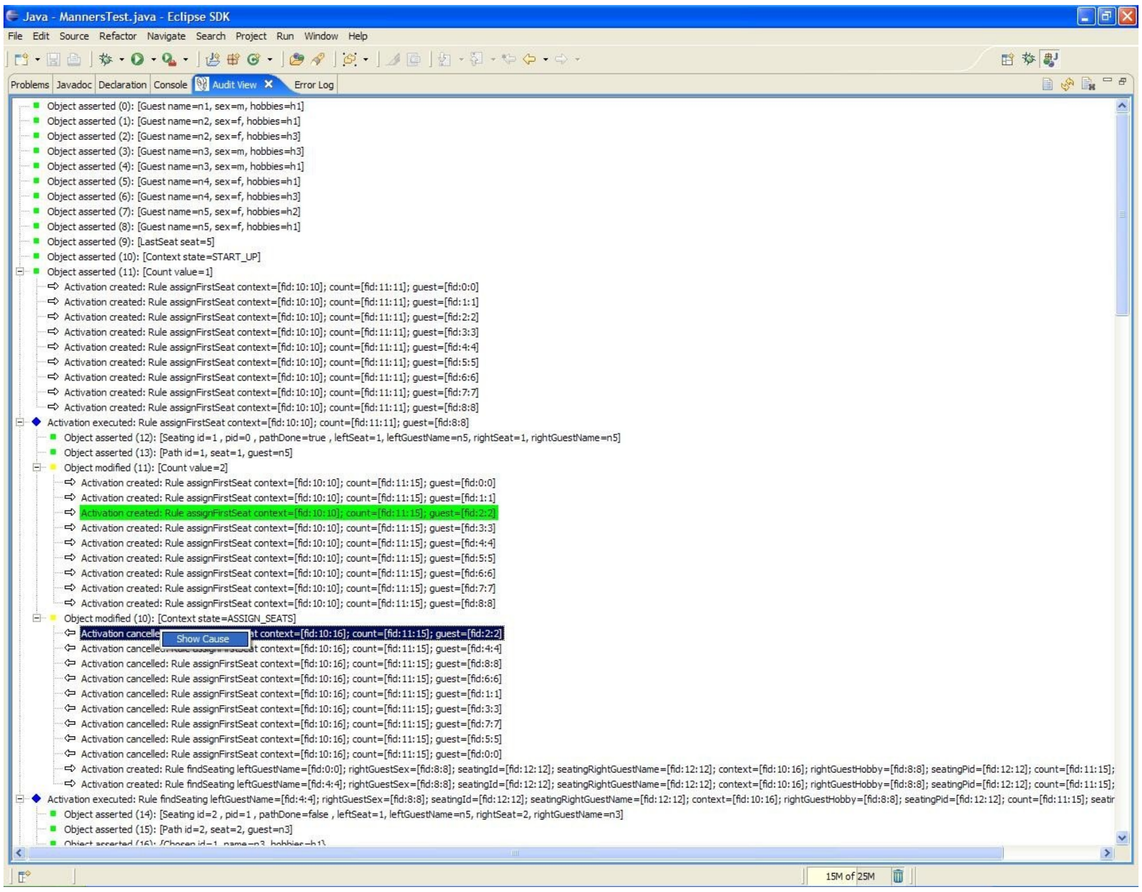
```
test= "Global String"
```

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

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

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

## 4.6.4. The Audit View


Java - MannersTest.java - Eclipse SDK
File Edit Source Refactor Navigate Search Project Run Window Help
Problems Javadoc Declaration Console Audit View Error Log
Show Cause
15M of 25M

```
Object asserted (0): [Guest name=n1, sex=m, hobbies=h1]
Object asserted (1): [Guest name=n2, sex=f, hobbies=h1]
Object asserted (2): [Guest name=n2, sex=f, hobbies=h3]
Object asserted (3): [Guest name=n3, sex=m, hobbies=h3]
Object asserted (4): [Guest name=n3, sex=m, hobbies=h1]
Object asserted (5): [Guest name=n4, sex=f, hobbies=h1]
Object asserted (6): [Guest name=n4, sex=f, hobbies=h3]
Object asserted (7): [Guest name=n5, sex=f, hobbies=h2]
Object asserted (8): [Guest name=n5, sex=f, hobbies=h1]
Object asserted (9): [LastSeat seat=5]
Object asserted (10): [Context state=START_UP]
Object asserted (11): [Count value=1]
  Activation created: Rule assignFirstSeat context=[fid:10:10]; count=[fid:11:11]; guest=[fid:0:0]
  Activation created: Rule assignFirstSeat context=[fid:10:10]; count=[fid:11:11]; guest=[fid:1:1]
  Activation created: Rule assignFirstSeat context=[fid:10:10]; count=[fid:11:11]; guest=[fid:2:2]
  Activation created: Rule assignFirstSeat context=[fid:10:10]; count=[fid:11:11]; guest=[fid:3:3]
  Activation created: Rule assignFirstSeat context=[fid:10:10]; count=[fid:11:11]; guest=[fid:4:4]
  Activation created: Rule assignFirstSeat context=[fid:10:10]; count=[fid:11:11]; guest=[fid:5:5]
  Activation created: Rule assignFirstSeat context=[fid:10:10]; count=[fid:11:11]; guest=[fid:6:6]
  Activation created: Rule assignFirstSeat context=[fid:10:10]; count=[fid:11:11]; guest=[fid:7:7]
  Activation created: Rule assignFirstSeat context=[fid:10:10]; count=[fid:11:11]; guest=[fid:8:8]
Activation executed: Rule assignFirstSeat context=[fid:10:10]; count=[fid:11:11]; guest=[fid:8:8]
  Object asserted (12): [Seating id=1 , pid=0 , pathDone=true , leftSeat=1, leftGuestName=n5, rightSeat=1, rightGuestName=n5]
  Object asserted (13): [Path id=1, seat=1, guest=n5]
  Object modified (11): [Count value=2]
    Activation created: Rule assignFirstSeat context=[fid:10:10]; count=[fid:11:15]; guest=[fid:0:0]
    Activation created: Rule assignFirstSeat context=[fid:10:10]; count=[fid:11:15]; guest=[fid:1:1]
    Activation created: Rule assignFirstSeat context=[fid:10:10]; count=[fid:11:15]; guest=[fid:2:2]
    Activation created: Rule assignFirstSeat context=[fid:10:10]; count=[fid:11:15]; guest=[fid:3:3]
    Activation created: Rule assignFirstSeat context=[fid:10:10]; count=[fid:11:15]; guest=[fid:4:4]
    Activation created: Rule assignFirstSeat context=[fid:10:10]; count=[fid:11:15]; guest=[fid:5:5]
    Activation created: Rule assignFirstSeat context=[fid:10:10]; count=[fid:11:15]; guest=[fid:6:6]
    Activation created: Rule assignFirstSeat context=[fid:10:10]; count=[fid:11:15]; guest=[fid:7:7]
    Activation created: Rule assignFirstSeat context=[fid:10:10]; count=[fid:11:15]; guest=[fid:8:8]
  Object modified (10): [Context state=ASSIGN_SEATS]
    Activation cancelle          t context=[fid:10:16]; count=[fid:11:15]; guest=[fid:2:2]
    Activation cancelle          t context=[fid:10:16]; count=[fid:11:15]; guest=[fid:4:4]
    Activation cancelled: Rule assignFirstSeat context=[fid:10:16]; count=[fid:11:15]; guest=[fid:8:8]
    Activation cancelled: Rule assignFirstSeat context=[fid:10:16]; count=[fid:11:15]; guest=[fid:6:6]
    Activation cancelled: Rule assignFirstSeat context=[fid:10:16]; count=[fid:11:15]; guest=[fid:1:1]
    Activation cancelled: Rule assignFirstSeat context=[fid:10:16]; count=[fid:11:15]; guest=[fid:3:3]
    Activation cancelled: Rule assignFirstSeat context=[fid:10:16]; count=[fid:11:15]; guest=[fid:7:7]
    Activation cancelled: Rule assignFirstSeat context=[fid:10:16]; count=[fid:11:15]; guest=[fid:5:5]
    Activation cancelled: Rule assignFirstSeat context=[fid:10:16]; count=[fid:11:15]; guest=[fid:0:0]
    Activation created: Rule findSeating leftGuestName=[fid:0:0]; rightGuestSex=[fid:8:8]; seatingId=[fid:12:12]; seatingRightGuestName=[fid:12:12]; context=[fid:10:16]; rightGuestHobby=[fid:8:8]; seatingPid=[fid:12:12]; count=[fid:11:15];
    Activation created: Rule findSeating leftGuestName=[fid:4:4]; rightGuestSex=[fid:8:8]; seatingId=[fid:12:12]; seatingRightGuestName=[fid:12:12]; context=[fid:10:16]; rightGuestHobby=[fid:8:8]; seatingPid=[fid:12:12]; count=[fid:11:15];
Activation executed: Rule findSeating leftGuestName=[fid:4:4]; rightGuestSex=[fid:8:8]; seatingId=[fid:12:12]; seatingRightGuestName=[fid:12:12]; context=[fid:10:16]; rightGuestHobby=[fid:8:8]; seatingPid=[fid:12:12]; count=[fid:11:15]; seatir
  Object asserted (14): [Seating id=2 , pid=1 , pathDone=false , leftSeat=1, leftGuestName=n5, rightSeat=2, rightGuestName=n3]
  Object asserted (15): [Path id=2, seat=2, guest=n3]
  Object asserted (16): [Chosen id=1, name=n3, hobbies=h1]
```

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.

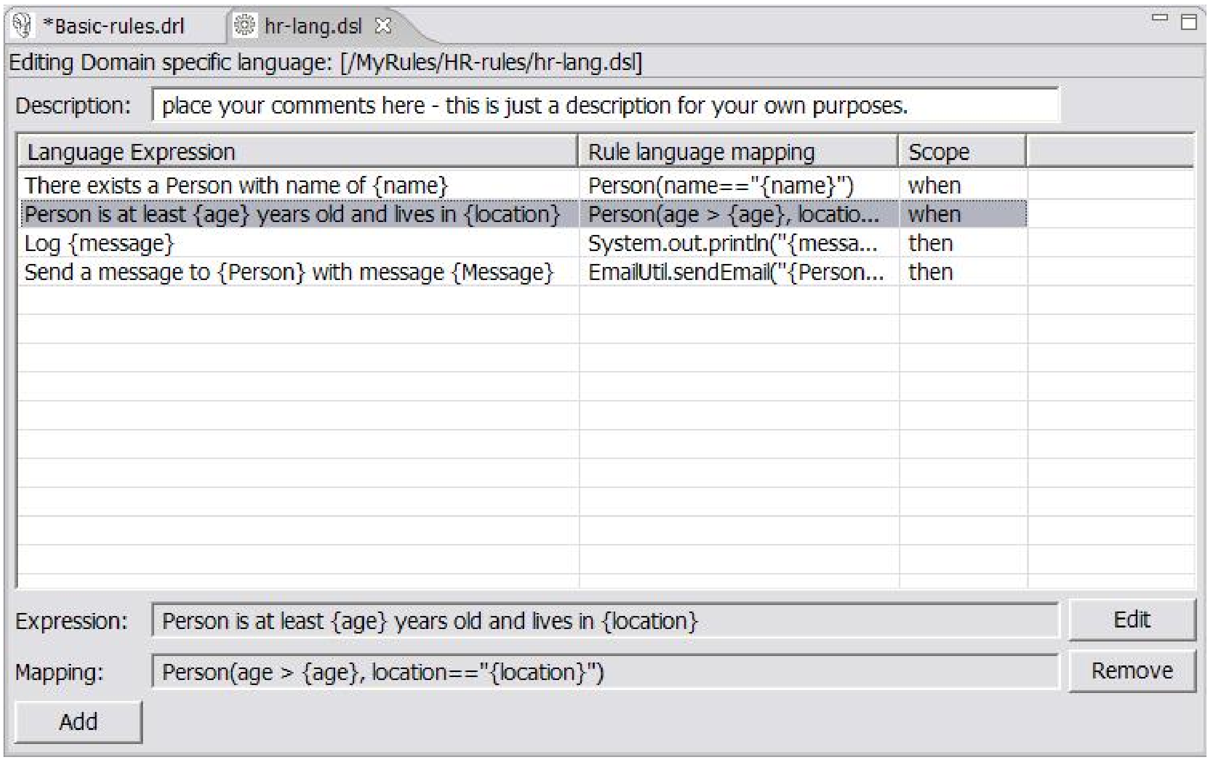
When selecting an event, the cause of that event is shown in green in the audit view (if visible of course). You can also right click the action and select the "Show Cause" menu item. This will scroll you to the cause of the selected event.

## 4.7. 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).

## 4.7.1. Editing languages



Editing Domain specific language: [/MyRules/HR-rules/hr-lang.dsl]

Description: place your comments here - this is just a description for your own purposes.

| Language Expression | Rule language mapping | Scope |
|---|---|---|
| There exists a Person with name of {name} | Person(name=="{name}") | when |
| Person is at least {age} years old and lives in {location} | Person(age > {age}, locatio... | when |
| Log {message} | System.out.println("{messa... | then |
| Send a message to {Person} with message {Message} | EmailUtil.sendEmail("{Person... | then |

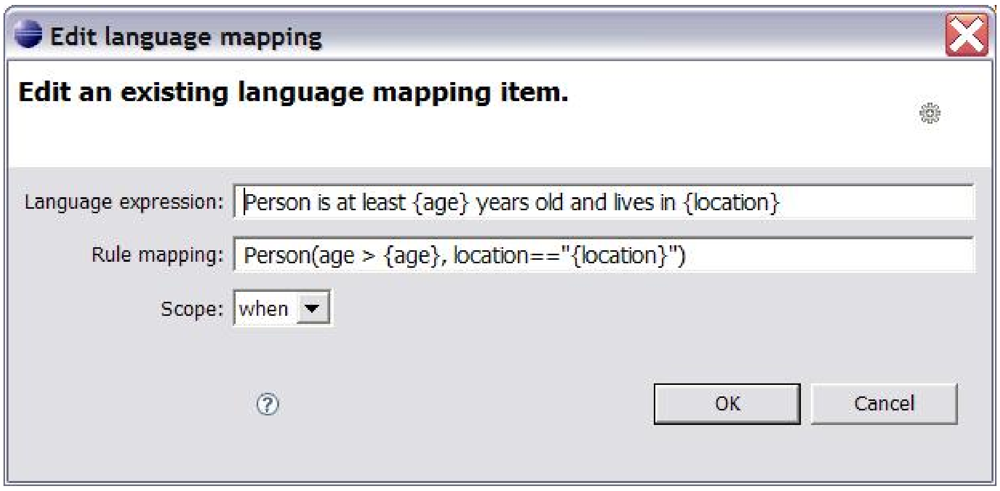
Expression: Person is at least {age} years old and lives in {location}

Mapping: Person(age > {age}, location=="{location}")

**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).


Edit language mapping

Edit an existing language mapping item.

Language expression: Person is at least {age} years old and lives in {location}

Rule mapping: Person(age > {age}, location=="{location}")

Scope: when

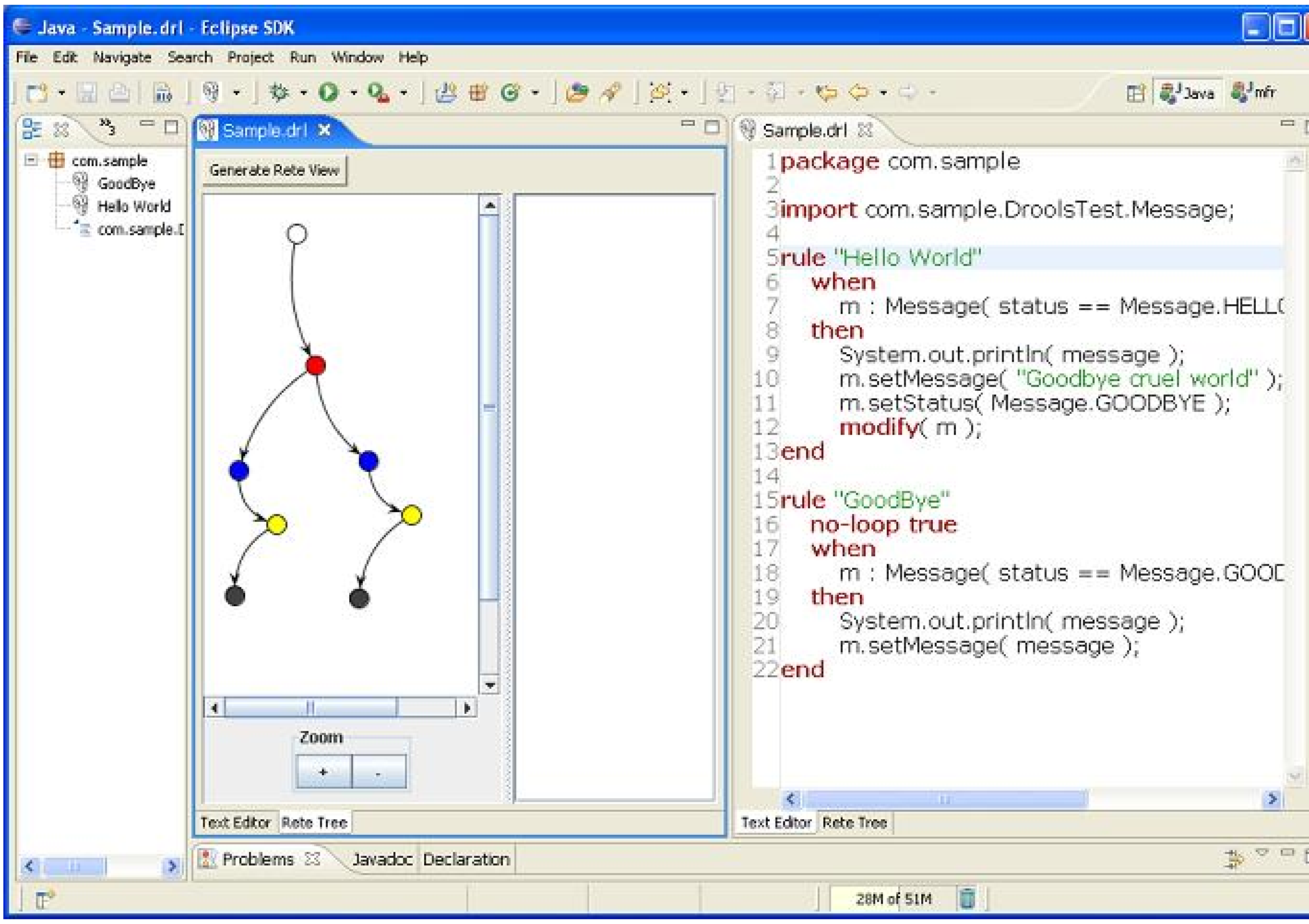
**Figure 4.10. Language Mapping editor dialog**

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.

## 4.8. The Rete View

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

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



```
package com.sample

import com.sample.DroolsTest.Message;

rule "Hello World"
   when
      m : Message( status == Message.HELL
   then
      System.out.println( message );
      m.setMessage( "Goodbye cruel world" );
      m.setStatus( Message.GOODBYE );
      modify( m );
end

rule "GoodBye"
   no-loop true
   when
      m : Message( status == Message.GOOD
   then
      System.out.println( message );
      m.setMessage( message );
end
```

The Rete View is an advanced feature which takes full advantage of the Eclipse Graphical Editing Framework (GEF).

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

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

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

## 4.9. Large drl files

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

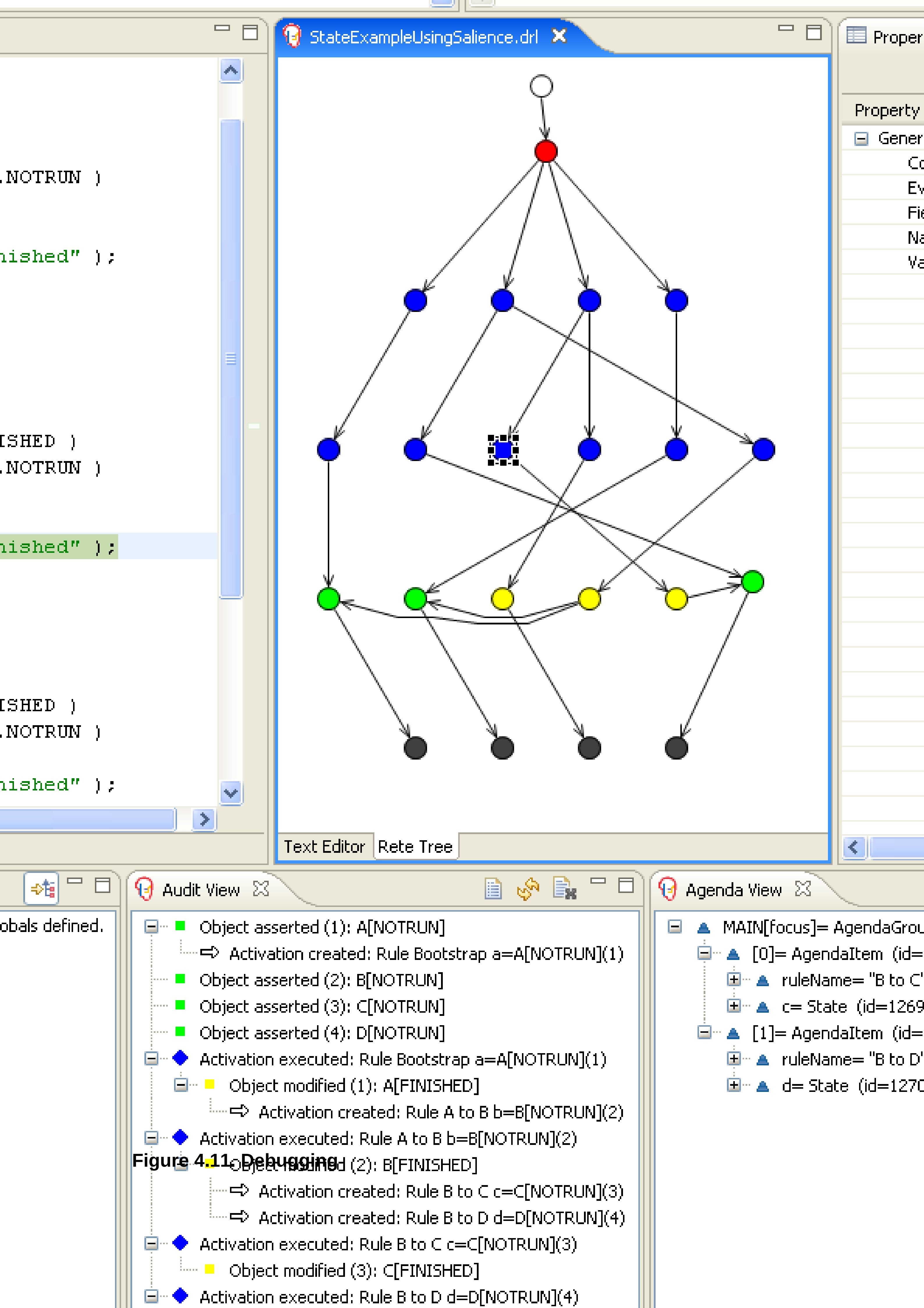
To increase the permanent generation, start Eclipse with -XX:MaxPermSize=###m

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

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

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

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



```
.NOTRUN )

nished" );

ISHED )
.NOTRUN )

nished" );

ISHED )
.NOTRUN )

nished" );
```

Audit View:

```
Object asserted (1): A[NOTRUN]
    Activation created: Rule Bootstrap a=A[NOTRUN](1)
Object asserted (2): B[NOTRUN]
Object asserted (3): C[NOTRUN]
Object asserted (4): D[NOTRUN]
Activation executed: Rule Bootstrap a=A[NOTRUN](1)
    Object modified (1): A[FINISHED]
        Activation created: Rule A to B b=B[NOTRUN](2)
Activation executed: Rule A to B b=B[NOTRUN](2)
    Object modified (2): B[FINISHED]
        Activation created: Rule B to C c=C[NOTRUN](3)
        Activation created: Rule B to D d=D[NOTRUN](4)
Activation executed: Rule B to C c=C[NOTRUN](3)
    Object modified (3): C[FINISHED]
Activation executed: Rule B to D d=D[NOTRUN](4)
```

Agenda View:

```
MAIN[focus]= AgendaGrou
    [0]= AgendaItem (id=
        ruleName= "B to C
        c= State (id=1269
    [1]= AgendaItem (id=
        ruleName= "B to D
        d= State (id=1270
```

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.

### 4.10.1. Creating breakpoints

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

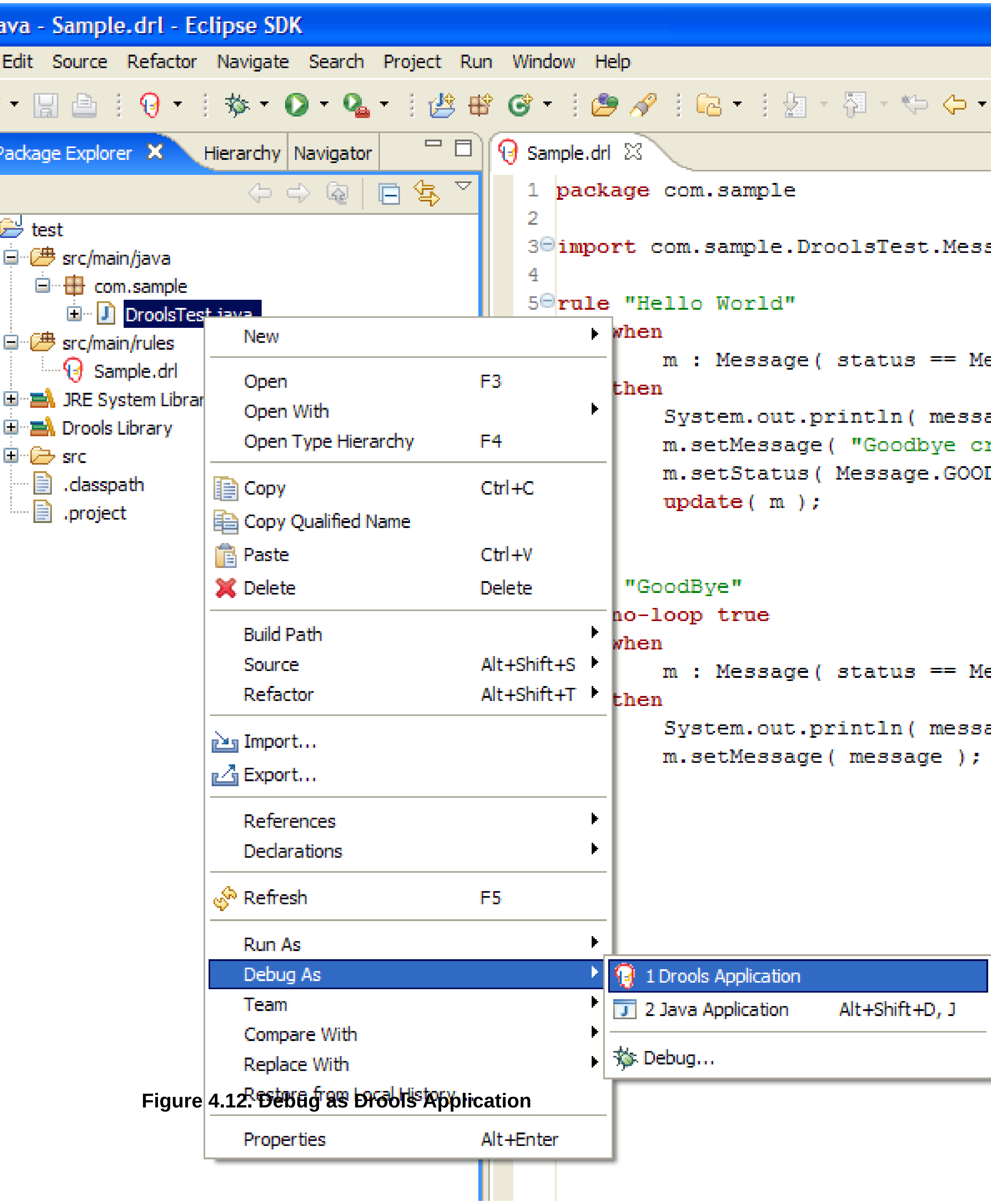
1. Double-click the ruler of the DRL editor at the line where you want to add a breakpoint. Note that rule breakpoints can only be created in the consequence of a rule. Double-clicking on a line where no breakpoint is allowed will do nothing. A breakpoint can be removed by double-clicking the ruler once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

### 4.10.2. Debugging rules

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



```
package com.sample

import com.sample.DroolsTest.Mess

rule "Hello World"
when
    m : Message( status == Me
then
    System.out.println( messa
    m.setMessage( "Goodbye cr
    m.setStatus( Message.GOOD
    update( m );

 "GoodBye"
no-loop true
when
    m : Message( status == Me
then
    System.out.println( messa
    m.setMessage( message );
```

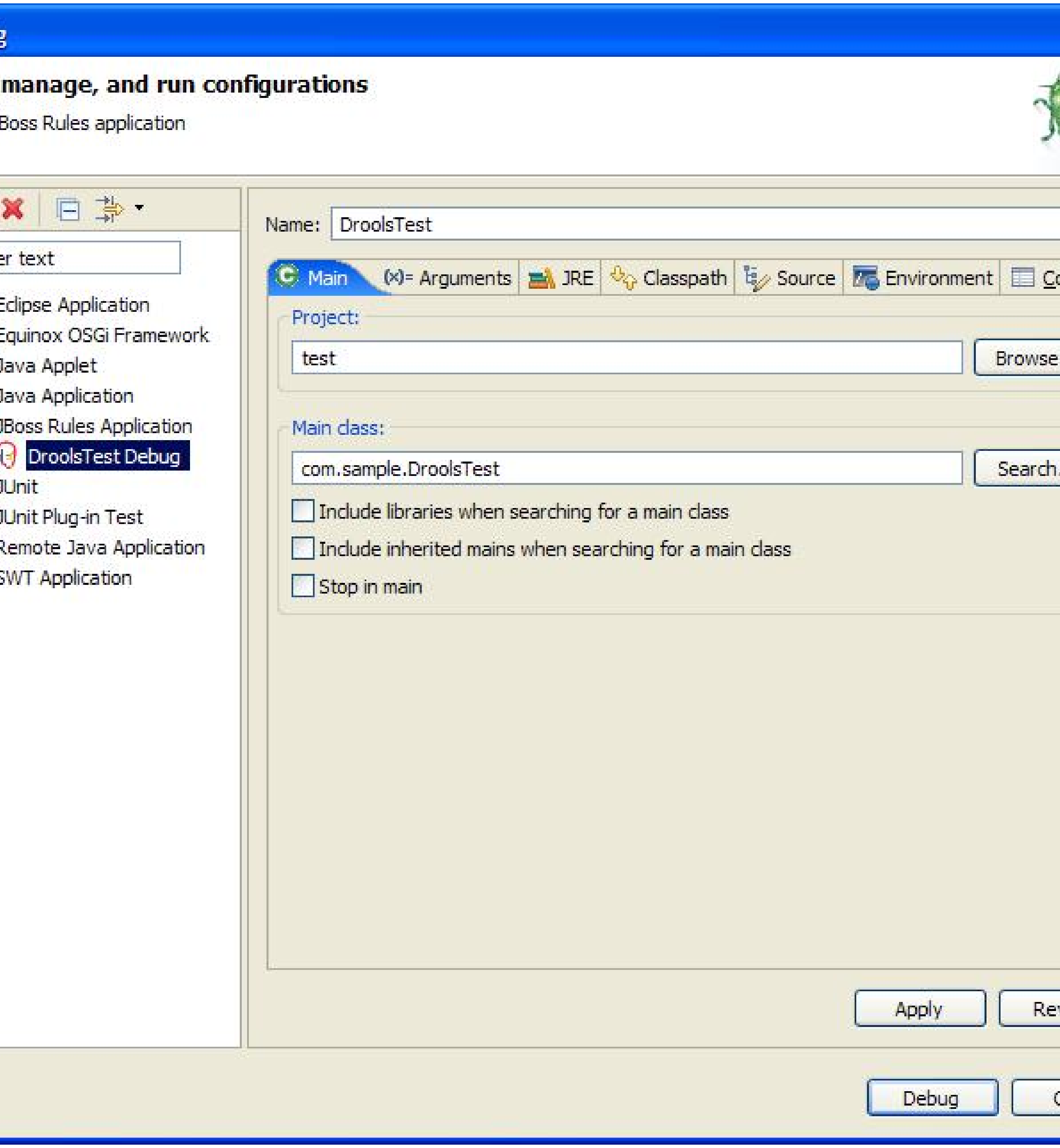
Figure 4.12. Debug as Drools Application

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).



manage, and run configurations

Boss Rules application

er text

Eclipse Application
Equinox OSGi Framework
Java Applet
Java Application
JBoss Rules Application
DroolsTest Debug
JUnit
JUnit Plug-in Test
Remote Java Application
SWT Application

Name: DroolsTest

Project:

test

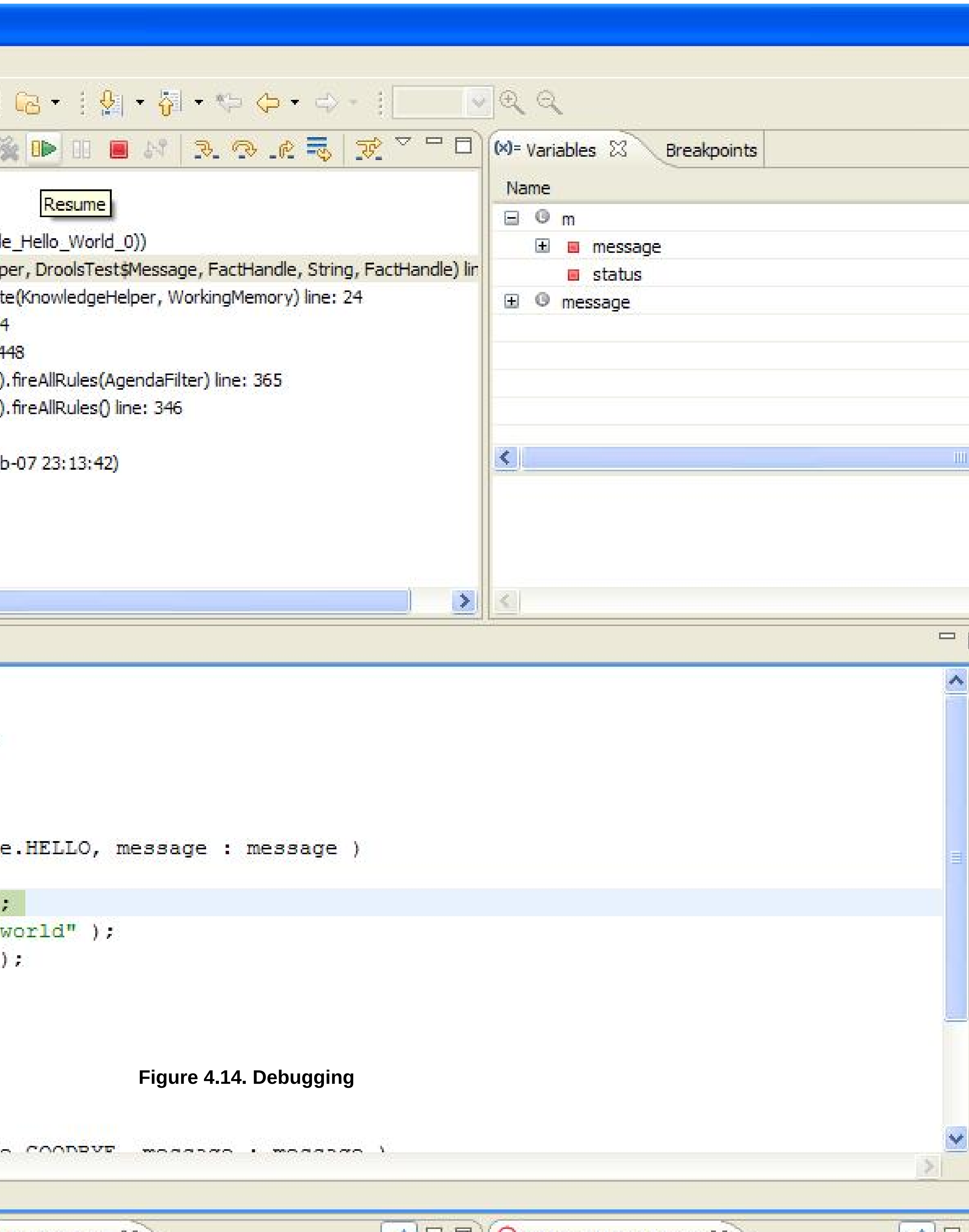
Main class:

com.sample.DroolsTest

- [ ] Include libraries when searching for a main class
- [ ] Include inherited mains when searching for a main class
- [ ] Stop in main

Figure 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).



```
le_Hello_World_0))
per, DroolsTest$Message, FactHandle, String, FactHandle) lir
te(KnowledgeHelper, WorkingMemory) line: 24
4
448
).fireAllRules(AgendaFilter) line: 365
).fireAllRules() line: 346

b-07 23:13:42)
```

```
m
    message
    status
message
```

```
e.HELLO, message : message )

;
world" );
);
```

Figure 4.14. Debugging

# Chapter 5. Deployment and Testing

## 5.1. 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).

The recommended way of deploying depends on how you are managing your rules. If you are using the BRMS, you should use the RuleAgent (the detailed documentation for this is in the chapter on the BRMS). You can still use the RuleAgent even without the BRMS, in which case you will need to use ant (the drools-ant task or something custom) to create serialized "Package" objects of your rules.

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.

### 5.1.1. Deployment using the RuleAgent

The easiest and most automated way to deploy rules is to use the RuleAgent. This is described in detail in the BRMS user guide. In short, the rule agent requires that you build binary packages of rules outside of "your" application (ie the application that is using rules).

The upside of this is that your application only needs to include drools-core.jar - no other dependencies (of course you need the classes that form the model that the rules use as well !). It also means the agent can be configured to automatically monitor for rule changes - directly to the BRMS, or from a file/directory.

To use the rule agent in your application use the following code:

```
RuleAgent agent = RuleAgent.newRuleAgent("/MyRules.properties");
RuleBase rb = agent.getRuleBase();
rb.newStatefulSession....
//now assert your facts into the session and away you go !
```

The MyRules.properties is a configuration file which (in the above case) should be on the root of your classpath:

```
##
```

```
## RuleAgent configuration file example
##



dir=/my/dir
url=http://some.url/here http://some.url/here
localCacheDir=/foo/bar/cache
poll=30


name=MyConfig
```

In the above config, the agent will look for binary package files in /my/dir, and also at the specified URLs. It will pick up any changes for these packages and apply them to the rulebase.

If you are using the BRMS, you can use the url feature. You can use "file" or "dir" if the packages need to be manually migrated to your production servers.

### 5.1.2. 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).

### 5.1.3. 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).

### 5.1.4. 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.

### 5.1.4.1. 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
```

### 5.1.4.2. 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
```

### 5.1.4.3. 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
```

#### 5.1.4.4. 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).

### 5.1.5. Deployment patterns

#### 5.1.5.1. 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.

#### 5.1.5.2. Out of process rule building

In this case, rules are built 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).

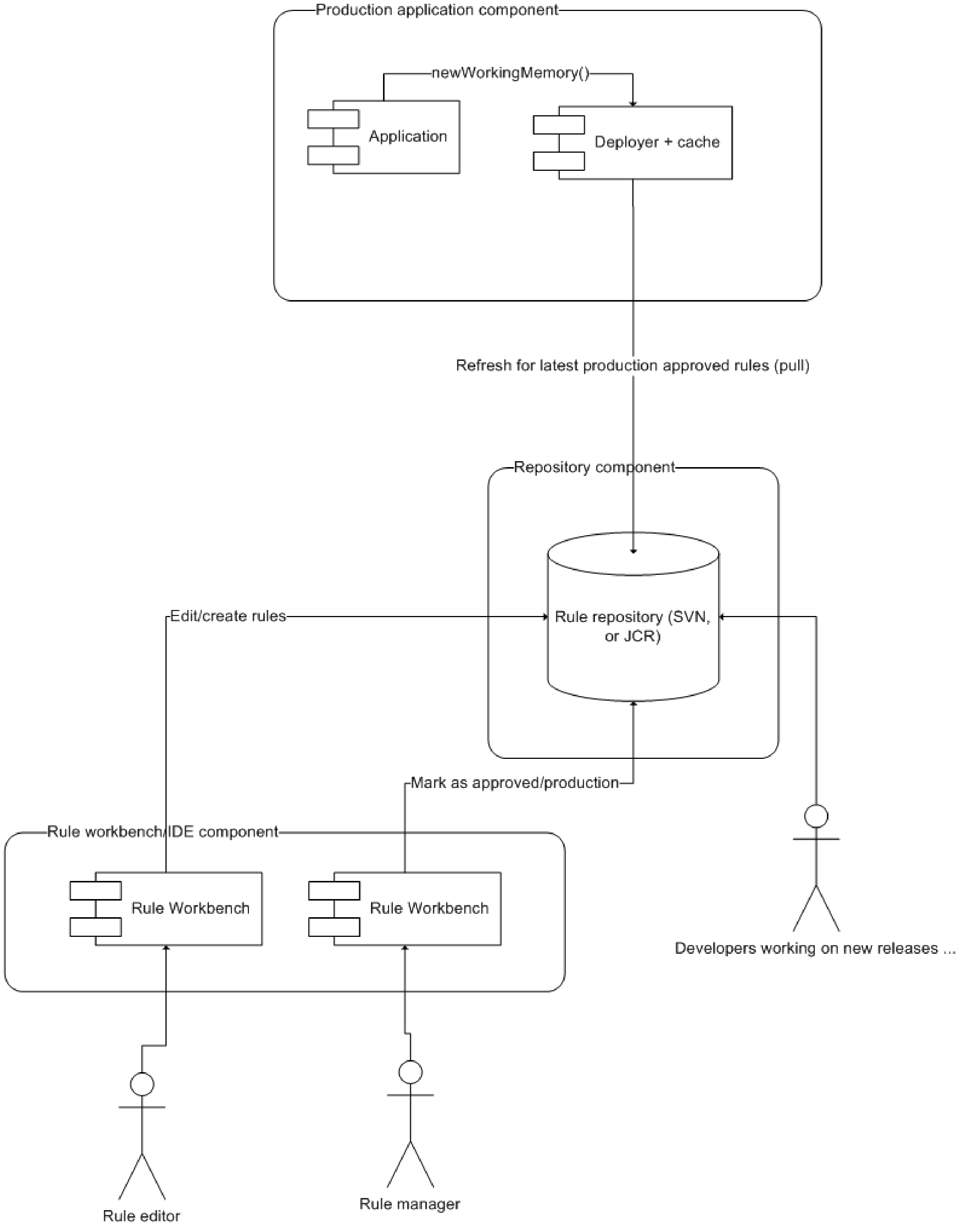
#### 5.1.5.3. Some deployment scenarios

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

##### 5.1.5.3.1. Pull style

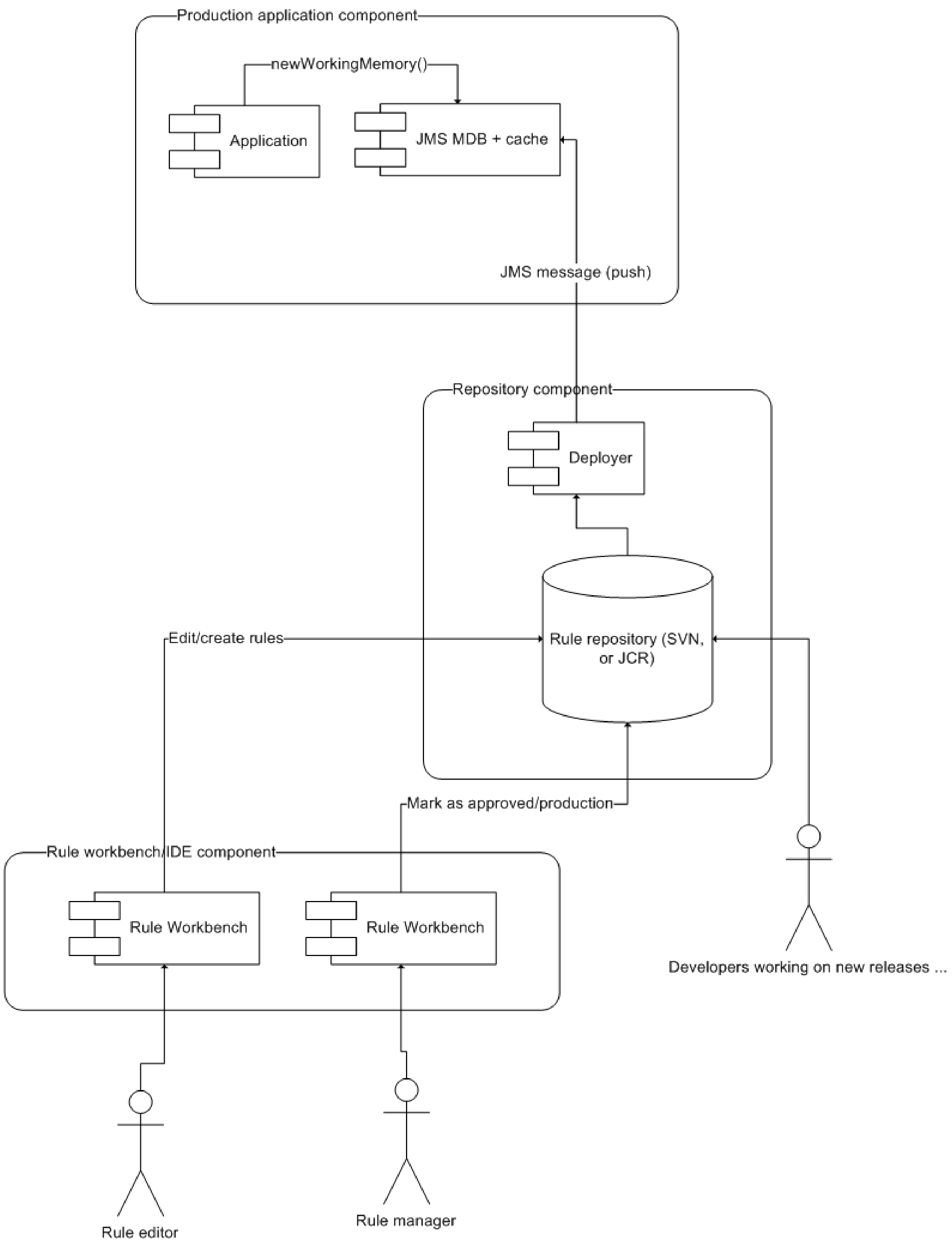
This pattern is what is used by the RuleAgent, by default.

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.


Production application component
newWorkingMemory()
Application
Deployer + cache
Refresh for latest production approved rules (pull)
Repository component
Rule repository (SVN, or JCR)
Edit/create rules
Mark as approved/production
Rule workbench/IDE component
Rule Workbench
Rule Workbench
Developers working on new releases ...
Rule editor
Rule manager

### 5.1.5.3.2. 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.


Production application component
newWorkingMemory()
Application
JMS MDB + cache
JMS message (push)
Repository component
Deployer
Rule repository (SVN, or JCR)
Edit/create rules
Mark as approved/production
Rule workbench/IDE component
Rule Workbench
Rule Workbench
Developers working on new releases ...
Rule editor
Rule manager

### 5.1.6. 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).

### 5.1.7. Future considerations

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

## 5.2. 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).

### 5.2.1. 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.

### 5.2.2. FIT for Rules - a rule testing framework

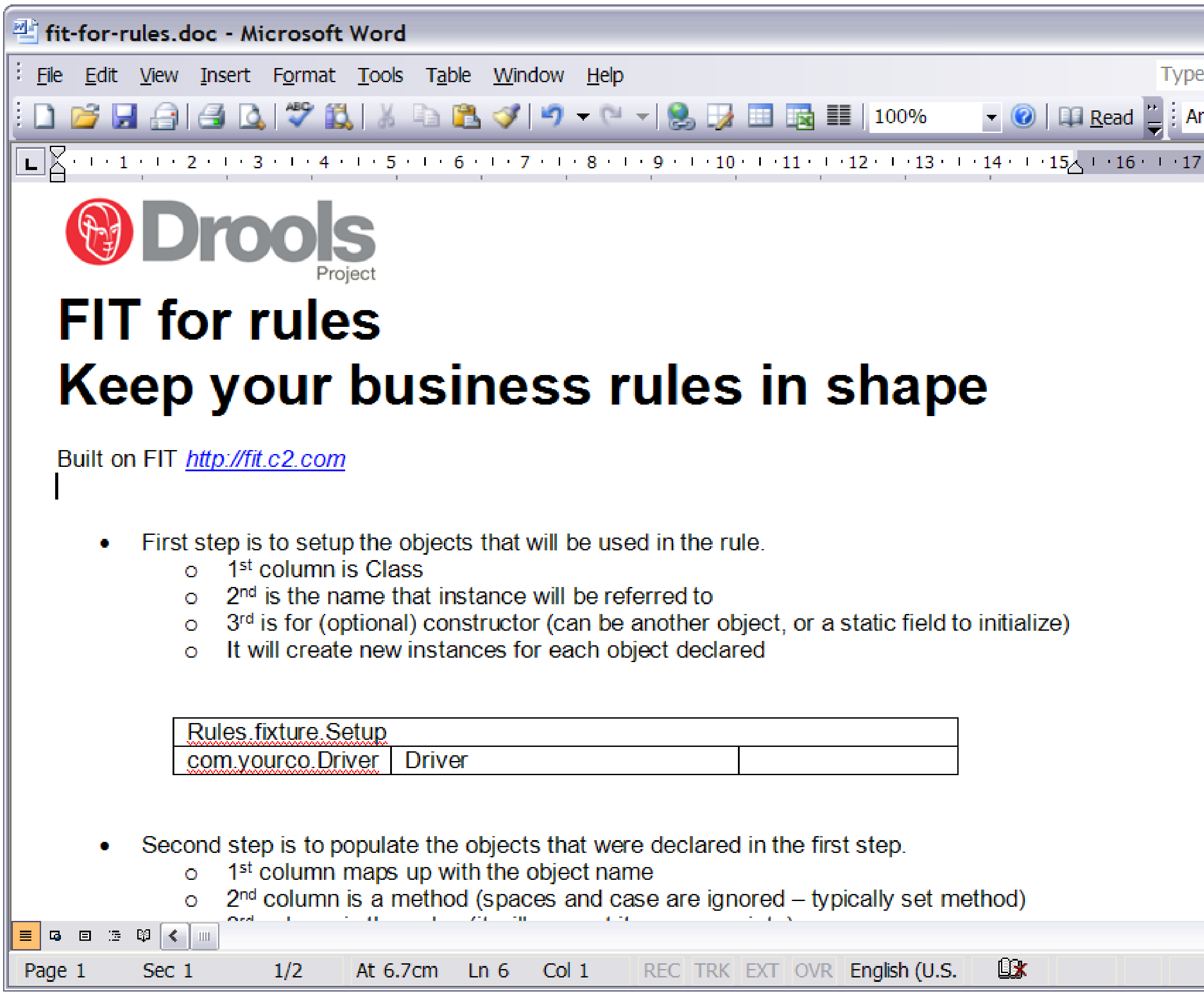
As a separate add-on, there is a testing framework available that is built on FIT (Framework for Integrated Testing). This allows rule test suites (functional) to be capture in Word documents, or

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

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


fit-for-rules.doc - Microsoft Word
File Edit View Insert Format Tools Table Window Help
100%
Read
Drools Project
Page 1 Sec 1 1/2 At 6.7cm Ln 6 Col 1 REC TRK EXT OVR English (U.S.

FIT for rules
Keep your business rules in shape

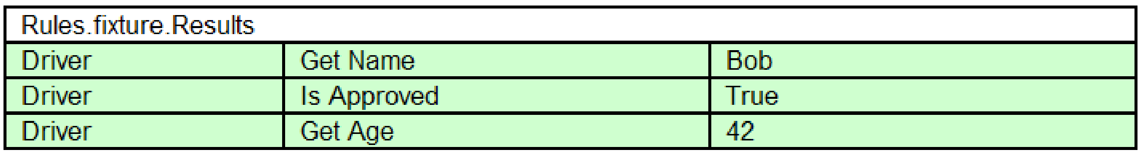
Built on FIT http://fit.c2.com

- First step is to setup the objects that will be used in the rule.
  - 1st column is Class
  - 2nd is the name that instance will be referred to
  - 3rd is for (optional) constructor (can be another object, or a static field to initialize)
  - It will create new instances for each object declared

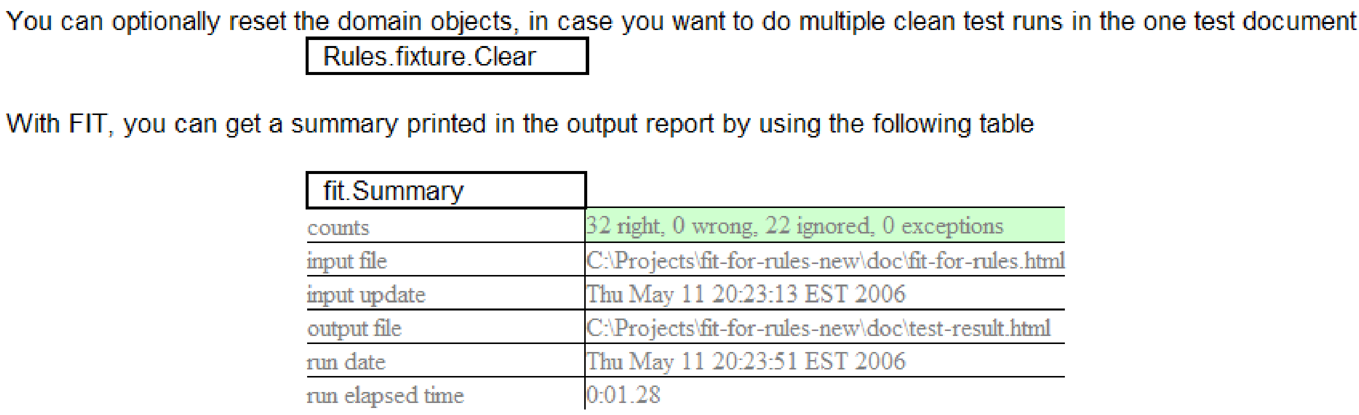
| Rules.fixture.Setup | | |
|---|---|---|
| com.yourco.Driver | Driver | |

- Second step is to populate the objects that were declared in the first step.
  - 1st column maps up with the object name
  - 2nd column is a method (spaces and case are ignored – typically set method)

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

# Chapter 6. The Java Rule Engine API

## 6.1. 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)* [http://www.w3.org/TR/2006/WD-rif-ucr-20060323/] and the OMG has started to work on a standard based on *RuleML* [http://ruleml.org/] , 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.

## 6.2. How To Use

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

### 6.2.1. Building and Registering RuleExecutionSets

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

#### Example 6.1. Automatic RuleServiceProvider Registration

```
// RuleServiceProviderImpl is registered to "http://drools.org/" via a
 static initialization block
Class.forName("org.drools.jsr94.rules.RuleServiceProviderImpl");

// Get the rule service provider from the provider manager.
```

```
RuleServiceProvider ruleServiceProvider =
 RuleServiceProviderManager.getRuleServiceProvider("http://drools.org/");
```

The RuleServiceProvider provides access to the RuleRuntime and RuleAdministration APIs. The RuleAdministration provides an administration API for the management of RuleExecutionSets, making it possible to register a RuleExecutionSet that can then be retrieved via the RuleRuntime.

First you need to create a RuleExecutionSet before it can be registered; RuleAdministrator provides factory methods to return an empty LocalRuleExecutionSetProvider or RuleExecutionSetProvider. The LocalRuleExecutionSetProvider should be used to load a RuleExecutionSets from local sources that are not serializable, like Streams. The RuleExecutionSetProvider can be used to load RuleExecutionSets from serializable sources, like DOM Elements or Packages. Both the "ruleAdministrator.getLocalRuleExecutionSetProvider( null );" and the "ruleAdministrator.getRuleExecutionSetProvider( null );" take null as a parameter, as the properties map for these methods is not currently used.

### Example 6.2. 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 6.3. 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 6.4. Register the RuleExecutionSet**

```
// Register the RuleExecutionSet with the RuleAdministrator
String uri = ruleExectionSet.getName();
ruleAdministrator.registerRuleExecutionSet(uri, ruleExecutionSet, null);
```

## 6.2.2. Using Stateful and Stateless RuleSessions

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

**Example 6.5. 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 6.6. Stateful Rule**

```
(StatefulRuleSession) session = ruleRuntime.createRuleSession( uri,
                                                               null,

 RuleRuntime.STATEFUL_SESSION_TYPE );
```

```
session.addObject( new PurchaseOrder( "lots of cheese" ) );
session.executeRules();
```

The StatelessRuleSession has a very simple API; you can only call executeRules(List list) passing a list of objects, and an optional filter, the resulting objects are then returned.

**Example 6.7. 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 );
```

#### 6.2.2.1. Globals

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

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

Do not forget to declare the global "list" in your DRL:

```
package SistersRules;
import org.drools.jsr94.rules.Person;
global java.util.List list
rule FindSisters
when
$person1 : Person ( $name1:name )
$person2 : Person ( $name2:name )
```

```
eval( $person1.hasSister($person2) )
then
list.add($person1.getName() + " and " + $person2.getName() +" are sisters");

assert( $person1.getName() + " and " + $person2.getName() +" are sisters");
end
```

## 6.3. References

If you need more information on JSR 94, please refer to the following references

1. Official JCP Specification for Java Rule Engine API (JSR 94)
   - *http://www.jcp.org/en/jsr/detail?id=94*
2. The Java Rule Engine API documentation
   - *http://www.javarules.org/api_doc/api/index.html*
3. The Logic From The Bottom Line: An Introduction to The Drools Project. By N. Alex Rupp, published on TheServiceSide.com in 2004
   - *http://www.theserverside.com/articles/article.tss?l=Drools*
4. Getting Started With the Java Rule Engine API (JSR 94): Toward Rule-Based Applications. By Dr. Qusay H. Mahmoud, published on Sun Developer Network in 2005
   - *http://java.sun.com/developer/technicalArticles/J2SE/JavaRule.html*
5. Jess and the javax.rules API. By Ernest Friedman-Hill, published on TheServerSide.com in 2003
   - *http://www.theserverside.com/articles/article.tss?l=Jess*

# Chapter 7. The Rule Language

## 7.1. Overview

**Note**

*(updated to Drools 4.0)*

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.

### 7.1.1. 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 7.1. 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.

### 7.1.2. 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).

## 7.2. Keywords

**Note**

*(updated to Drools 5.0)*

Drools 5 introduces the concept of Hard and Soft keywords.

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

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

- true
- false
- accumulate
- collect
- from

- null
- over
- then
- when

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

- lock-on-active
- date-effective
- date-expires
- no-loop
- auto-focus
- activation-group
- agenda-group
- ruleflow-group
- entry-point
- duration
- package
- import
- dialect
- salience
- enabled
- attributes
- rule
- extend
- template
- query
- declare

- function
- global
- eval
- not
- in
- or
- and
- exists
- forall
- action
- reverse
- result
- end
- init

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

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

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

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

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

```
rule "validate holiday"
```

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

## 7.3. Comments

**Note**

*(updated to Drools 4.0)*

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.

### 7.3.1. Single line comment

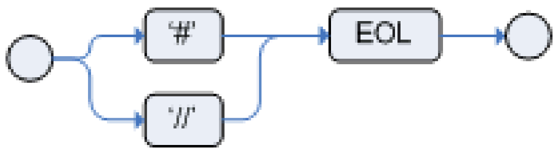


**Figure 7.1. Single line comment**

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
```

### 7.3.2. Multi line comment

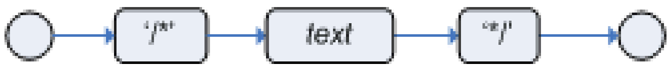


**Figure 7.2. Multi line comment**

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
```

## 7.4. Error Messages

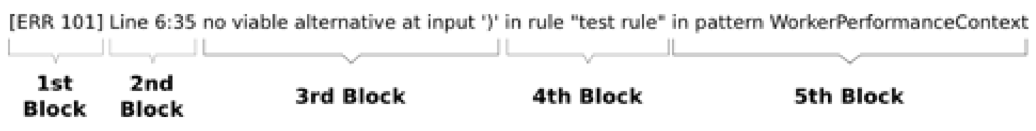
**Note**

*(updated to Drools 5.0)*

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

### 7.4.1. Message format

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



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

**Figure 7.3. Error Message Format**

**1st Block:** This area identifies the error code.

**2nd Block:** Line:column information.

**3rd Block:** Some text describing the problem.

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

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

## 7.4.2. Error Messages Description

### 7.4.2.1. 101: No viable alternative

Most common kind of error, means that the parser came to a decision point but it couldn't identify an alternative. Here as some examples:

#### Example 7.2.

```
1: rule one
2:   when
3:     exists Foo()
4:     exits Bar()
5:   then
6: end
```

The above example generates this message:

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

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

#### Example 7.3.

```
1: package org.drools;
2: rule
3:   when
4:     Object()
5:   then
6:     System.out.println("A RHS");
7: end
```

Now the above code generates this message:

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

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

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

#### Example 7.4.

```
1: rule simple_rule
```

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

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

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

**Tip**

Usually the Line and Column information are accurate, but in some cases (like unclosed quotes), the parser generates a 0:-1 position. In this case you should check if you didn't forget to close quotes nor brackets.

### 7.4.2.2. 102: Mismatched input

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

**Example 7.5.**

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

The above example generates this message:

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

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

**Tip**

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

The following code generates more than one error messages:

**Example 7.6.**

```
package org.drools;
```

```
2:
3: rule "Avoid NPE on wrong syntax"
4:   when
5:     not( Cheese( ( type == "stilton", price == 10 ) || ( type == "brie",
 price == 15 ) ) from $cheeseList )
6:   then
7:     System.out.println("OK");
8: end
```

These are the errors associated with this source:

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

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

**Tip**

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

### 7.4.2.3. 103: Failed predicate

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

**Example 7.7.**

```
 1: package nesting;
 2: dialect "mvel"
 3:
 4: import org.drools.Person
 5: import org.drools.Address
 6:
 7: fdsfdsfds
 8:
 9: rule "test something"
10:   when
11:     p: Person( name=="Michael" )
12:   then
```

```
13:     p.name = "other";
14:     System.out.println(p.name);
15: end
```

With this sample, we get this error message:

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

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

**Tip**

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

### 7.4.2.4. 104: Trailing semi-colon not allowed

This error is associated with EVAL clause (at EVAL expression you cannot use a semi-colon). Check this sample:

**Example 7.8.**

```
1: rule simple_rule
2:   when
3:     eval(abc();)
4:   then
5: end
```

Due its trailling semi-colon at eval, we get this error message:

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

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

### 7.4.2.5. 105: Early Exit

The recognizer came to a (..)+ EBNF subrule that must match an alternative at least once, but the subrule did not match anything. To simplify: the parser followed a path that there is no way out. This example tries to illustrates it:

**Example 7.9.**

```
1: template test_error
```

```
2:    aa s  11;
3: end
```

This is the message associated to the above sample:

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

To fix this problem it is necessary to remove the numeric value (as it is not a valid sentence nor a valid data type to begin a new slot).

### 7.4.3. Other Messages

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

## 7.5. Package

**Note**

*(updated to Drools 4.0)*

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.

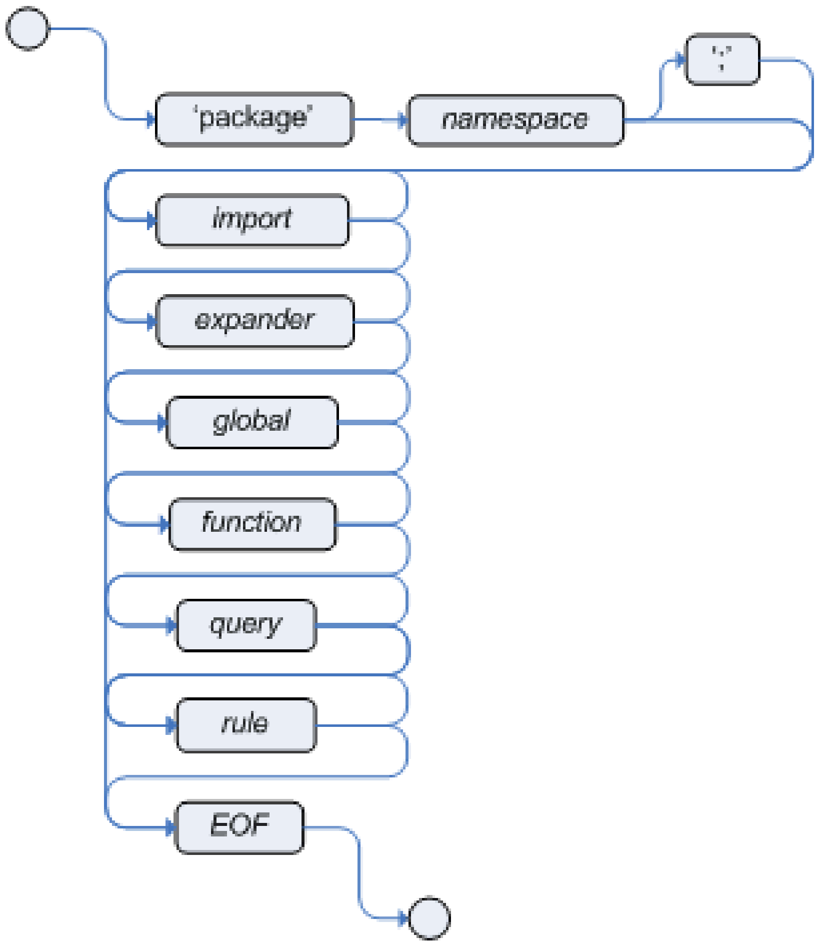


**Figure 7.4. package**

### 7.5.1. import

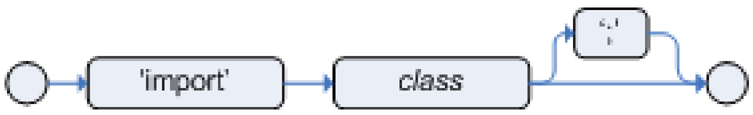


**Figure 7.5. import**

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

### 7.5.2. expander

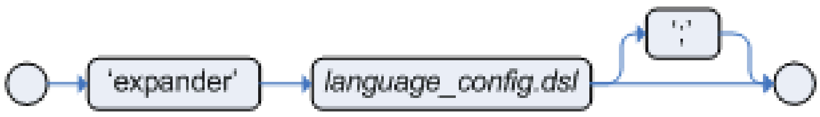


**Figure 7.6. expander**

The expander statement (optional) is used to specify domain specific language (DSL) configurations (which are normally stored in a separate file). This provides clues to the parser 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.

## 7.5.3. global

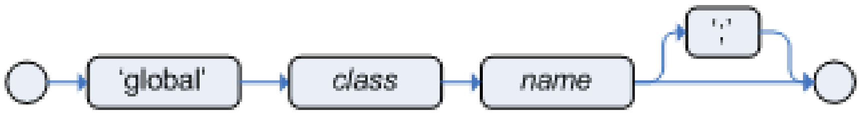


**Figure 7.7. global**

Globals are global variables. They are used to make application objects available to the rules, and are typically used to provide data or services that the rules use (specially application services used in rule consequences), to return data from the rules (like logs or values added in rules consequence) or for the rules to interact with the application doing callbacks. Globals are not inserted into the Working Memory so they should never be reasoned over, 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.

## 7.6. Function

**Note**

*(updated to Drools 4.0)*

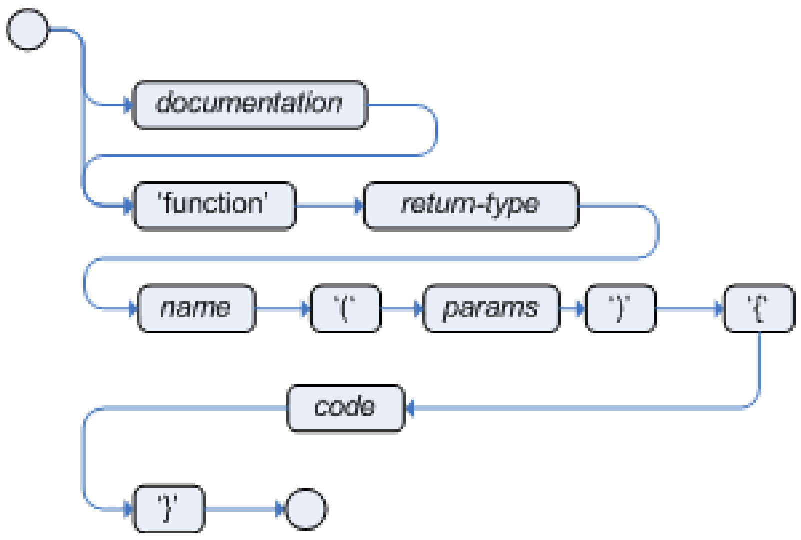


**Figure 7.8. function**

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

A typical function declaration looks like:

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

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

An alternative to the use of a function, could be to use a static method in a helper class: Foo.hello(). 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
```

## 7.7. Type Declaration

**Note**

*(updated to Drools 5.0)*

**Warning**

FIXME: add syntax diagram for declare

Type Declarations have two main goals in the rules engine: allow the declaration of new types and/or allow the declaration of metadata for types.

- **Declaring new types:** Drools works out of the box with plain POJOs as facts. Although, sometimes the users may want to define the model directly into the rules engine, without worrying to create their models in a lower level language like Java. Another times, there is a domain model already built, but eventually the user wants or needs to complement this model with additional entities that are used mainly during the reasoning process.

- **Declaring metadata:** facts may have meta information associated to them. Examples of meta information include any kind of data that is not represented by the fact attributes and are consistent among all instances of that fact type. This meta information may be queried at runtime by the engine and used in the reasoning process.

## 7.7.1. Declaring New Types

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

### Example 7.10. declaring a new fact type: Address

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

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

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

### Example 7.11. declaring a new fact type: Person

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

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

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

### Example 7.12. avoiding the need to use fully qualified class names by using import

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

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

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

### Example 7.13. generated Java class for the previous Person fact type declaration

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

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

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

### Example 7.14. using the declared types in rules

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

## 7.7.2. Declaring Metadata

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

```
@matadata_key( metadata_value )
```

The parenthesis and the metadata_value are optional.

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

#### Example 7.15. declaring an arbitrary metadata attribute

```
@author( Bob )
```

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

#### Example 7.16. declaring metadata attributes for fact types and attributes

```
import java.util.Date

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

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

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

### 7.7.3. Declaring Metadata for Existing Types

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

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

#### Example 7.17. declaring metadata for an existing type

```
import org.drools.examples.Person

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

```
end
```

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

**Example 7.18. declaring metadata using the fully qualified class name**

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

### 7.7.4. Accessing Declared Types from the Application Code

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

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

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

**Example 7.19. declaring a type in the org.drools.examples package**

```
package org.drools.examples

import java.util.Date

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

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

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

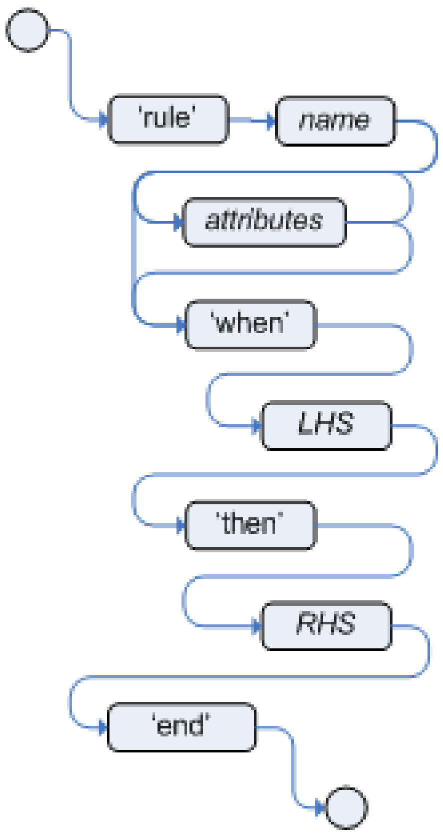
### Example 7.20. handling declared fact types through the API

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

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

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

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

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

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

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

Although the API is similar to Java reflection (yet much simpler to use), it does not use reflection underneath, relying in much more performant bytecode generated accessors.

## 7.8. Rule



### Figure 7.9. 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 (it is best to always use double quotes).

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

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

### Example 7.21. Rule Syntax Overview Example

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

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

### Example 7.22. 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
```

## 7.8.1. Rule Attributes

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

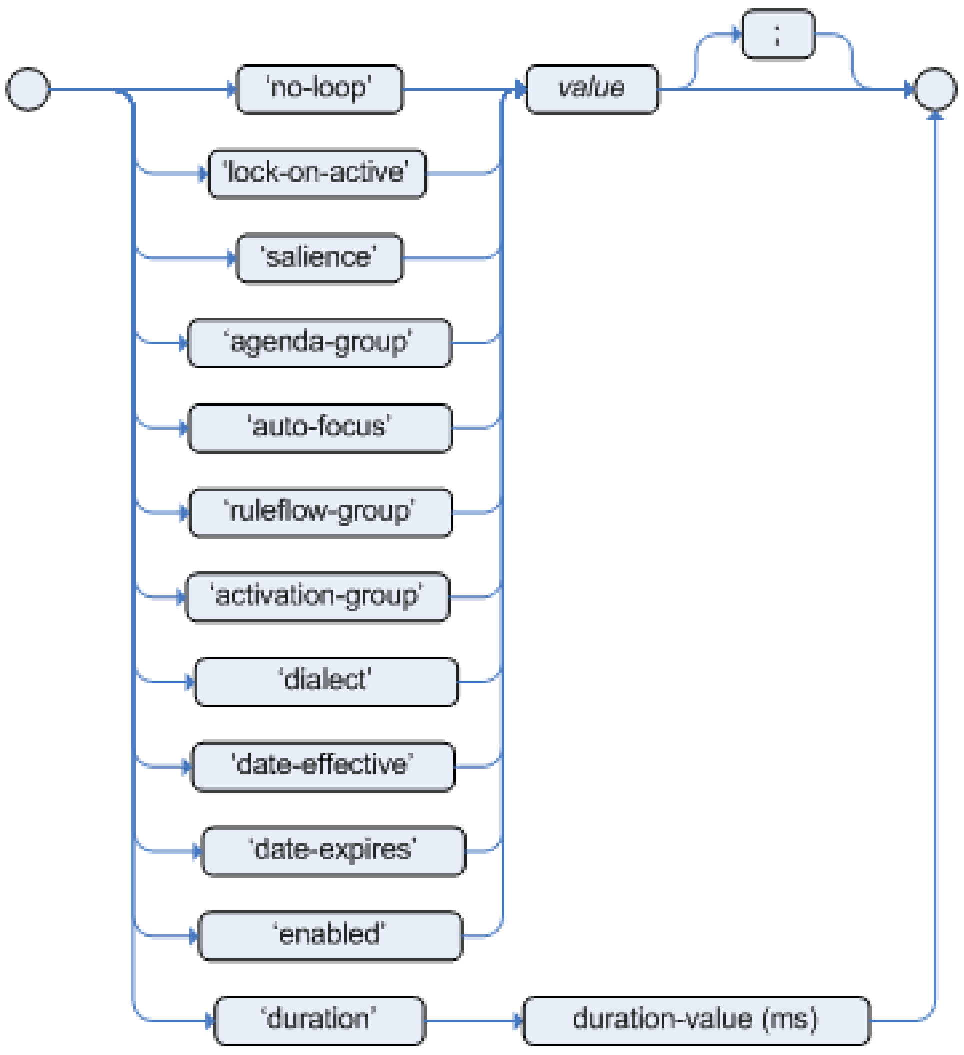


**Figure 7.10. rule attributes**

no-loop
: default value : false

  type : Boolean

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

lock-on-active
: default value : false

  type : Boolean

  when a ruleflow-group becomes active or an agenda-group receives the focus any rules that have lock-on-active set to try cannot place activations onto the agenda, the rules are matched and the resulting activations discarded. This is a stronger version of no-loop. It's ideally for calculation rules where you have a number of rules that will modify a fact and you don't want any rule re-matching and firing. In summary fire these currently active rules and only these rules, no matter how the data changes, do not allow any more activations for the rules with the attribute set to true. When the ruleflow-group is no longer active or agenda-group loses the focus those rules with lock-on-active set to true can once again add activations onto the agenda.

salience

: default value : 0

  type : integer

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

agenda-group

: default value : MAIN

  type : String

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

auto-focus

: default value false

  type : Boolean

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

activation-group

: default value : N/A

  type : String

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

  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.

dialect

: default value : as specified by the package type : String

possible values: "java" or "mvel"

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

date-effective
: default value : N/A

  type : String, which contains a Date/Time definition

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

date-expires
: default value : N/A

  type : String, which contains a Date/Time definition

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

duration
: default value : no default value

  type : long

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

**Example 7.23. Some attribute examples**

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

## 7.8.2. Left Hand Side (when) Conditional Elements

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

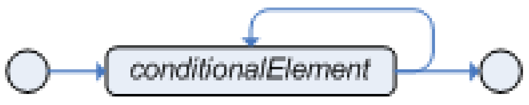


**Figure 7.11. Left Hand Side**

**Example 7.24. 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 bellow). 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?

### 7.8.2.1. Pattern

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

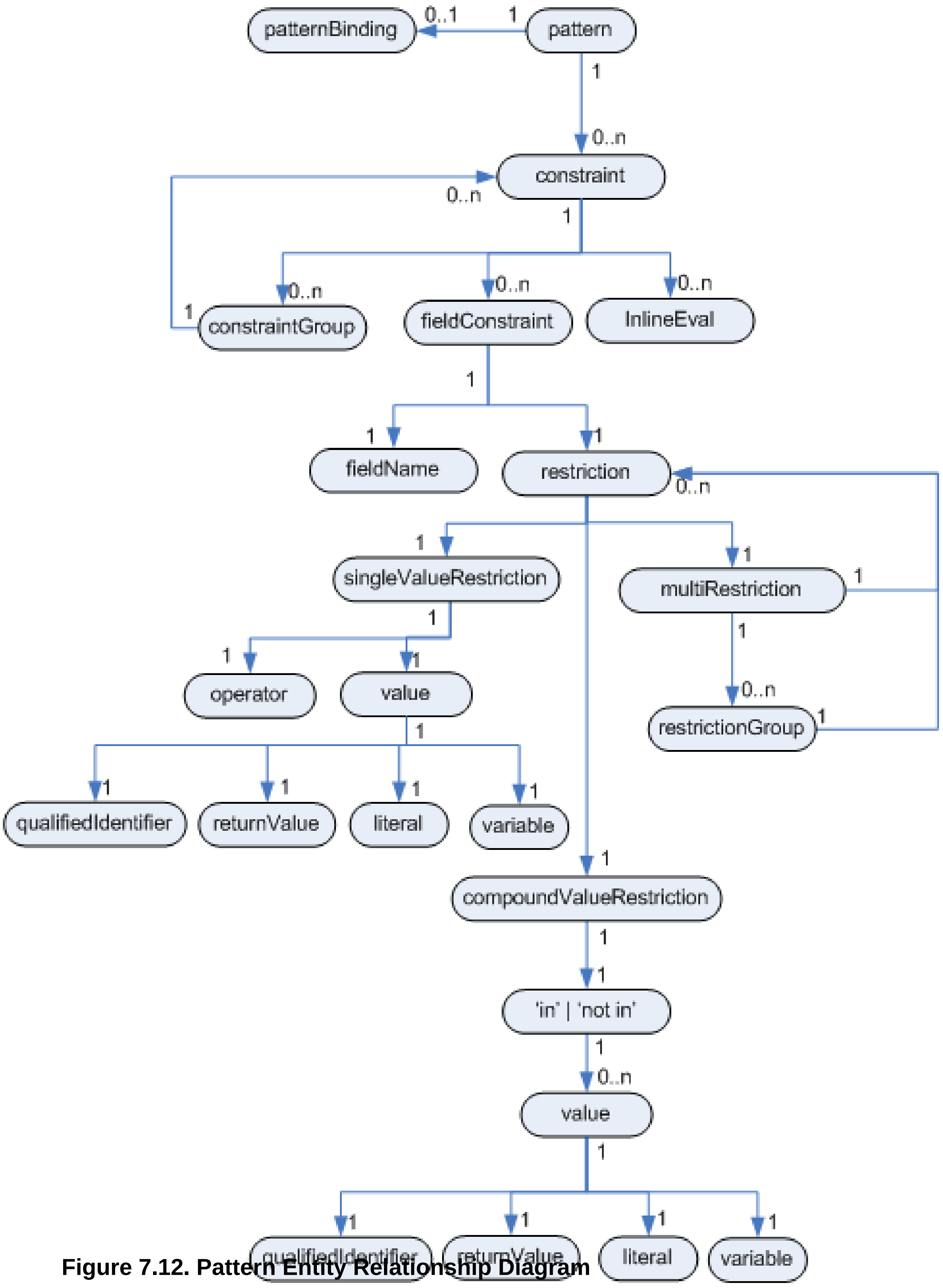


**Figure 7.12. Pattern Entity Relationship Diagram**

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.

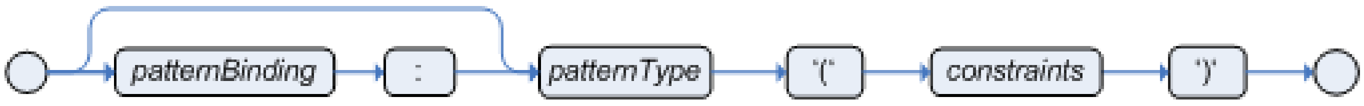


**Figure 7.13. Pattern**

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 7.25. 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 7.26. 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 '||'.

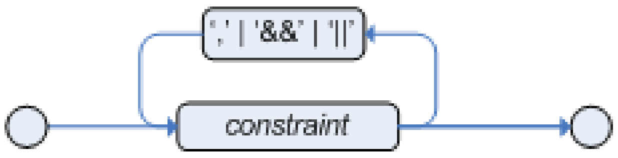


**Figure 7.14. Constraints**

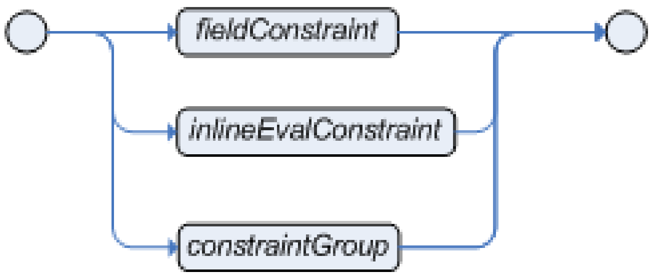
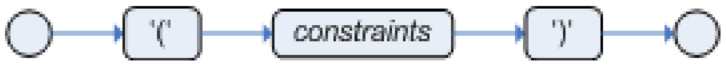


**Figure 7.15. Constraint**



### Figure 7.16. Group Constraint

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

### Example 7.27. 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 7.28. && 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 7.29. 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.

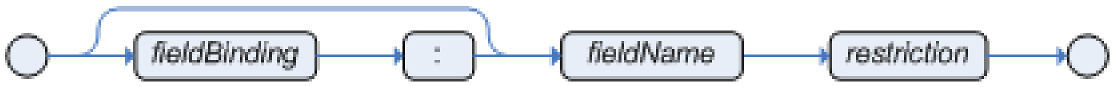
Also, it is important to note that besides having the same semantics, the connectives '&&' and ',' are resolved with different priorities and ',' cannot be embedded in a composite constraint expression.

### Example 7.30. 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
```

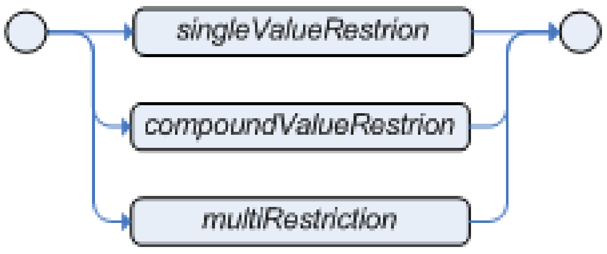
#### 7.8.2.1.1. Field Constraints

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



### Figure 7.17. fieldConstraint

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



### Figure 7.18. restriction

##### 7.8.2.1.1.1. 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).

### 7.8.2.1.1.2. Values

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

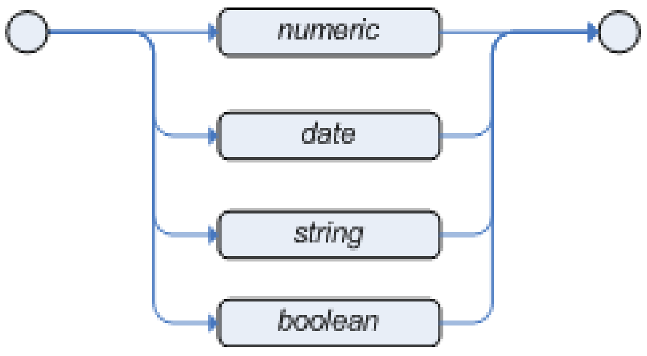


**Figure 7.19. literal**

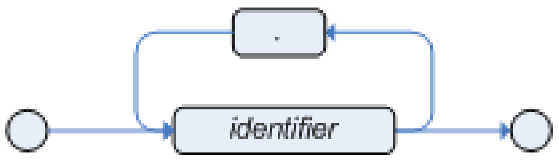


**Figure 7.20. qualifiedIdentifier**

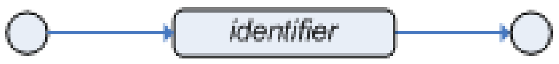


**Figure 7.21. variable**

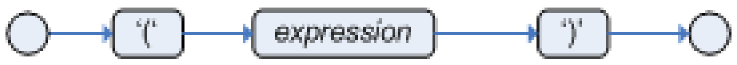


**Figure 7.22. returnValue**

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.

#### 7.8.2.1.1.3. Single Value Restriction

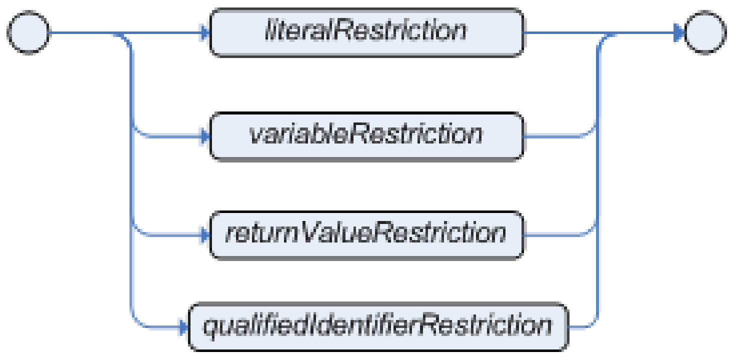


**Figure 7.23. singleValueRestriction**

##### 7.8.2.1.1.3.1. Operators

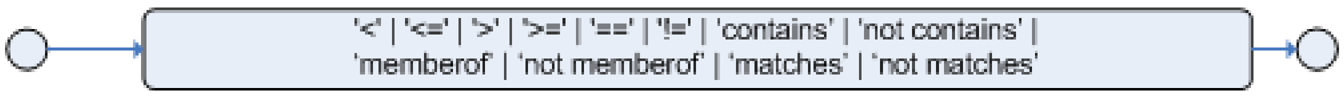

'<' | '<=' | '>' | '>=' | '==' | '!=' | 'contains' | 'not contains' |
'memberof' | 'not memberof' | 'matches' | 'not matches'

**Figure 7.24. 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 7.31. 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 7.32. 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 7.33. 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 7.34. 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 7.35. 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 7.36. 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)`

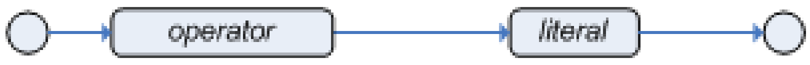
### Example 7.37. Text with soundslike (Sounds Like)

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

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

#### 7.8.2.1.1.3.2. Literal Restrictions

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



### Figure 7.25. 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 7.38. 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 7.39. Date Literal Restriction

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

#### String

Any valid Java String is allowed.

### Example 7.40. 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 7.41. Boolean Literal Restriction

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

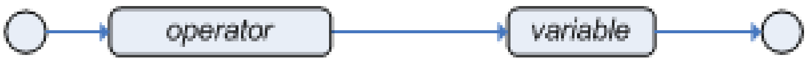
#### Qualified Identifier

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 7.42. Boolean Literal Restriction

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

#### 7.8.2.1.1.3.3. Bound Variable Restriction



### Figure 7.26. 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 7.43. Bound Field using '==' operator

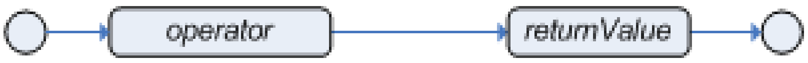
```
Person( likes : favouriteCheese )
Cheese( type == likes )
```

'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 7.44. Bound Fact using 'contains' operator

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

#### 7.8.2.1.1.3.4. Return Value Restriction



### Figure 7.27. 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 7.45. Return Value Restriction

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

#### 7.8.2.1.1.4. 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.

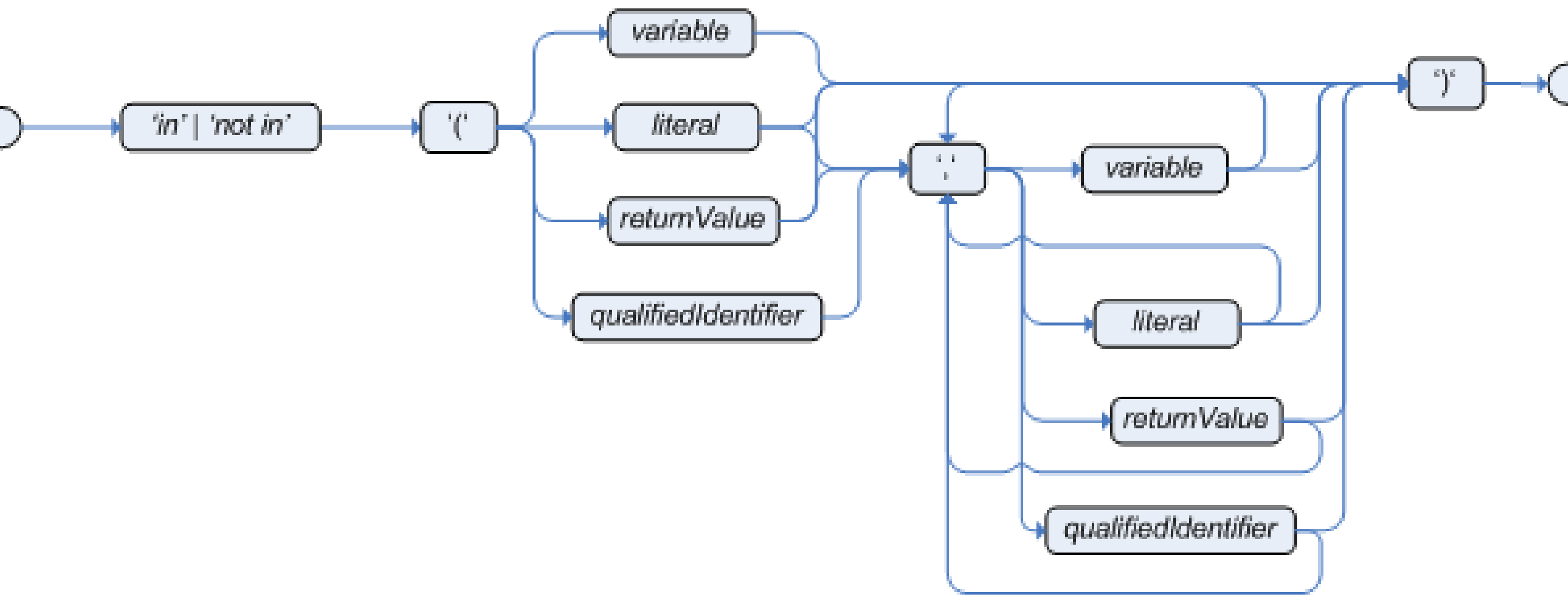


Figure 7.28. compoundValueRestriction

### Example 7.46. Compound Restriction using 'in'

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

#### 7.8.2.1.1.5. 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.

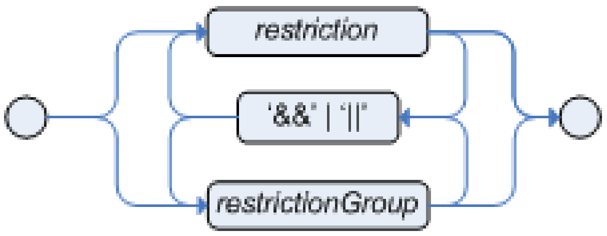


**Figure 7.29. multiRestriction**

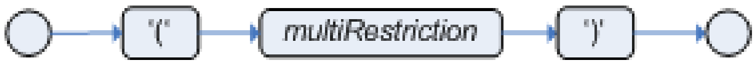


**Figure 7.30. restrictionGroup**

### Example 7.47. 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
```

### 7.8.2.1.2. Inline Eval Constraints

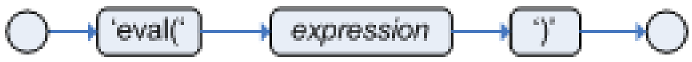


**Figure 7.31. Inline Eval Expression**

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 7.48. Return Value operator

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

#### 7.8.2.1.3. 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 the parent objects first and re-assert afterwards. If you only have a single parent at the root of the graph, when in the MVEL dialect, you can use the 'modify' keyword and its block setters to write the nested accessor assignments while retracting and inserting the the root parent object as required. Nested accessors can be used either side of the operator symbol.

**Example 7.49. 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
```

*NOTE: nested accessors have a much greater performance cost than direct field access, so use them carefully.*

### 7.8.2.2. '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.

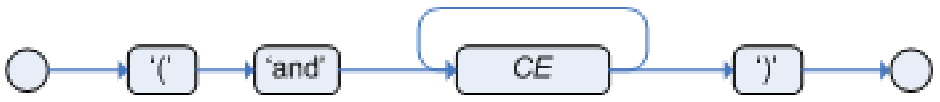


**Figure 7.32. prefixAnd**

**Example 7.50. prefixAnd**

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

**Example 7.51. 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.

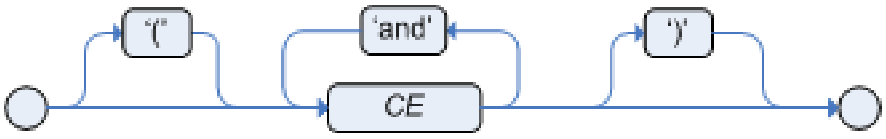


**Figure 7.33. infixAnd**

**Example 7.52. infixAnd**

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

### 7.8.2.3. '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.

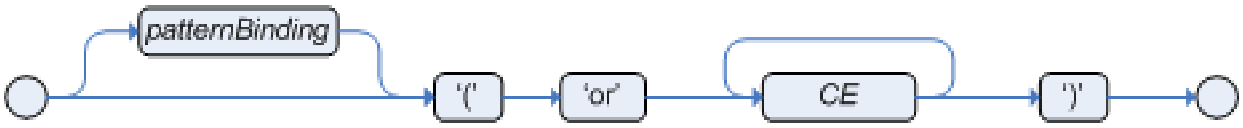


**Figure 7.34. prefixOr**

**Example 7.53. 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.

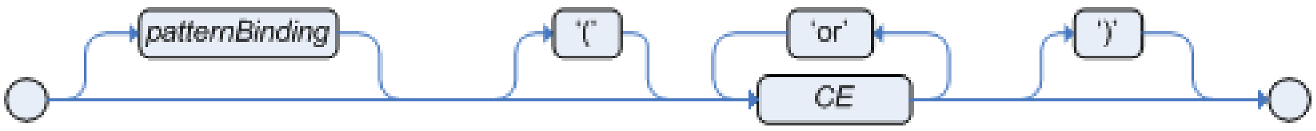


**Figure 7.35. infixOr**

**Example 7.54. 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 7.55. 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.

### 7.8.2.4. 'eval'

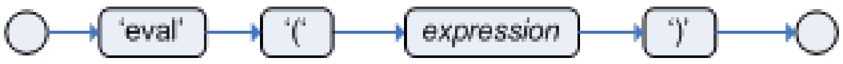


**Figure 7.36. eval**

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.

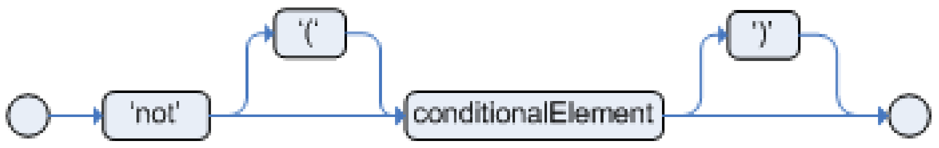
Evals cannot be indexed and thus are not as optimal as using Field Constraints. However this makes them ideal for being used when functions return values that change over time, which is not allowed within Field Constraints.

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 7.56. 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"
```

### 7.8.2.5. 'not'



### Figure 7.37. 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 7.57. No Busses

```
not Bus()
```

### Example 7.58. 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).
```

### 7.8.2.6. 'exists'

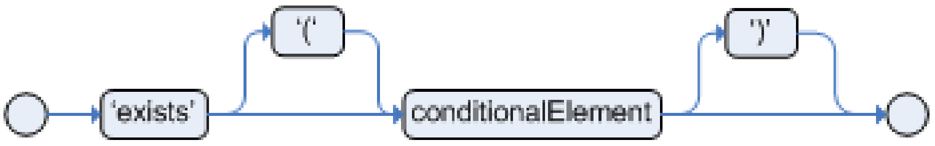


**Figure 7.38. exists**

'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 7.59. Atleast one Bus**

```
exists Bus()
```

**Example 7.60. 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
```

### 7.8.2.7. 'forall'

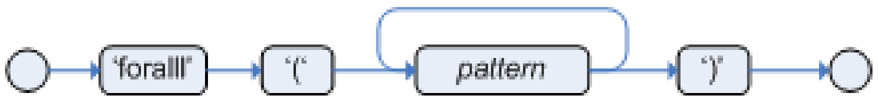


**Figure 7.39. forall**

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 7.61. 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 7.62. 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 7.63. Combining Forall with Not CE

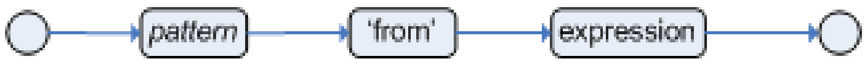
```
rule "not all employees have health and dental care"
when
    not ( forall( $emp : Employee()
                  HealthCare( employee == $emp )
                  DentalCare( employee == $emp ) )
          )
then
    # not all employees have health and dental care
end
```

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.

#### 7.8.2.8. From



### Figure 7.40. 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.

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

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

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

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

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

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

There are several ways to address this issue:

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

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

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

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

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

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

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

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

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

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

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

```
then
    modify ($assessment) {} #Modify assessment in a modify block
end
```

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

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

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

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

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:

### 7.8.2.9. 'collect'

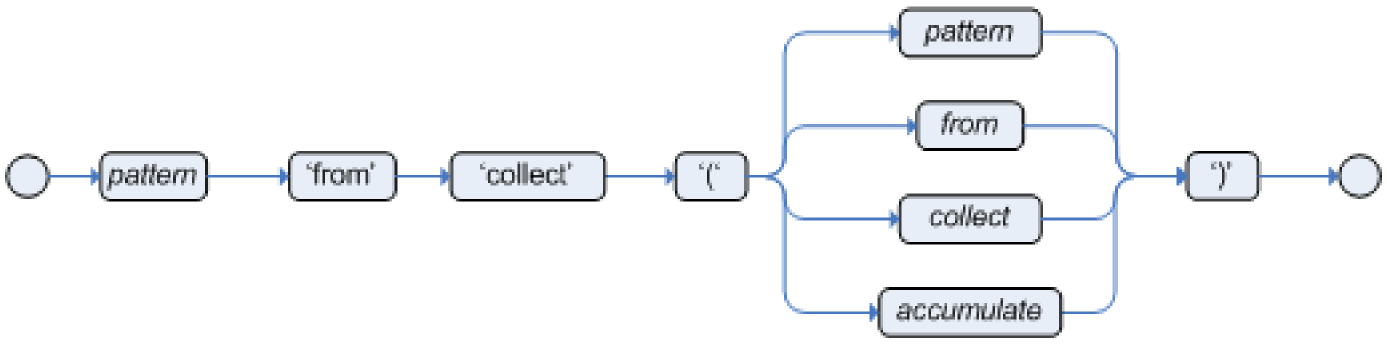


**Figure 7.41. collect**

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
```

### 7.8.2.10. 'accumulate'

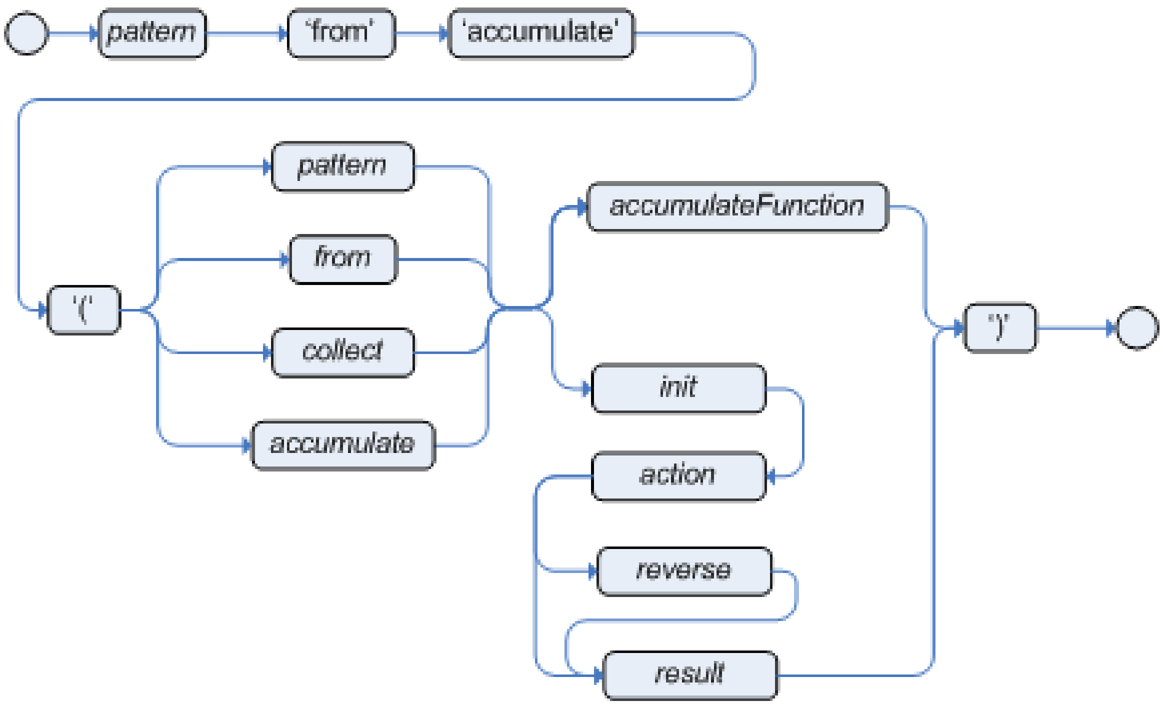


**Figure 7.42. accumulate**

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
```

### 7.8.2.10.1. 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.

## 7.8.3. The Right Hand Side (then)

The Right Hand Side (RHS) is a common name for the consequence or action part of the rule; this part should contain a list of actions to be executed. It is bad practice to use imperative or conditional code in the RHS of a rule; as a rule should be atomic in nature - "when this, then do this", not "when this, maybe do this". The RHS part of a rule should also be kept small, thus keeping it declarative and readable. If you find you need imperative and/or conditional code in the RHS, then maybe you should be breaking that rule down into multiple rules. The main purpose of the RHS is to insert, retractor modify working memory data. To assist with 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.

### 7.8.4. 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.

## 7.9. Query

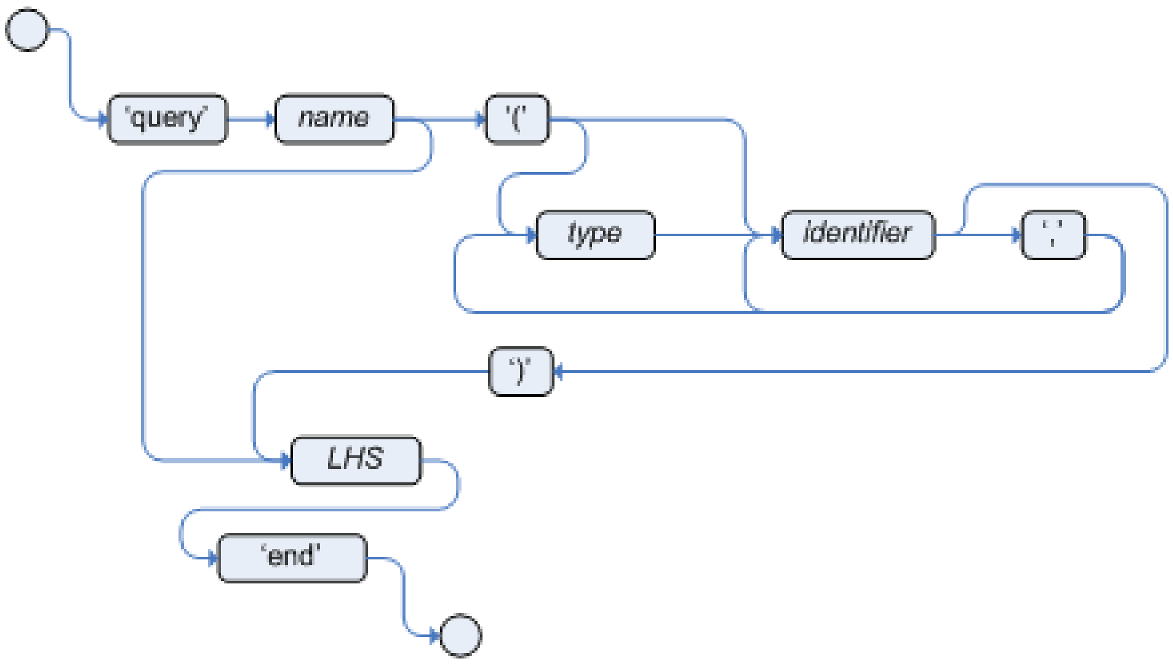


**Figure 7.43. query**

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 7.64. Query People over the age of 30

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

### Example 7.65. 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 7.66. 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" );
}
```

## 7.10. 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 both in the IDE, as well as the web based BRMS. Of course as rules are text, you can use them even without this tooling.

## 7.10.1. When to use a DSL

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

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

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.

## 7.10.2. Editing and managing a DSL

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

Note that since 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 7.67. 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 "This is " to the end of the line will be included as the value for "{something}" when it is interpolated into the target string. This may not be exactly what you want, as you may want to "chain" together different DSL expressions to generate a target expression. The best way around this is to make sure that the {tokens} are enclosed with characters or words. This means that the parser will scan along the sentence, and pluck out the value BETWEEN the characters (in the example below they are double-quotes). Note that the characters that surround the token are not included in when interpolating, just the contents between them (rather then all the way to the end of the line, as would otherwise be the case).

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

### Example 7.68. 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 and things will be easier. Using a DSL can make debugging slightly harder when you are first building rules, but it can make the maintenance easier (and of course the readability of the rules).

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

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

as well as with the token names as shown above.

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 7.69. 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 7.70. 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")
```

## 7.10.3. Using a DSL in your rules

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

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

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

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

```
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 7.71. Chaining DSL Expressions

```
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).

### 7.10.4. Adding constraints to facts

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

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

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

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

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

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

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

## 7.10.5. How it works

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

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

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

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

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

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

### 7.10.7. Scope and keywords

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

### 7.10.8. DSLs in the BRMS and IDE

You can use DSLs in the BRMS in both guided editor rules, and textual rules that use a dsl. (In fact, the same applies to the IDE).

In the guided editor - the DSLs generally have to be simpler - what you are doing is defining little "forms" to capture data from users in text fields (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.

In the BRMS - when you build a package the DSLs are already included and all the work is done for you. In the IDE (or in any IDE) - you will either need to use the drools-ant task, or otherwise use the code shown in sections above.

## 7.11. 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.

### 7.11.1. When to use XML

There are several scenarios that XML is desirable. However, we recommend that it is not a default choice, as XML is not readily human readable (unless you like headaches) and can create visually bloated rules.

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).

### 7.11.2. 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.

### Example 7.72. Example

```
<?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 7.73. 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 than the columns, as the rule engine has to evaluate it each time, but it is a "catch all" feature for when you can express what you need to do with Column constraints.

### 7.11.3. Legacy Drools 2.x XML rule format

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

### 7.11.4. 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.

# Chapter 8. Examples

## 8.1. Getting the examples

Make sure the Drools Eclipse plugin is installed, which needs GEF dependency installed first. Then download and extract the drools-examples zip file, which includes an already created Eclipse project. Import that project into a new Eclipse workspace. The rules all have example classes that execute the rules. If you want to try the examples in another project (or another IDE) then you will need to setup the dependencies by hand of course. Many, but not all of the examples are documented below, enjoy :)

## 8.2. Hello World

```
Name: Hello World
Main class: org.drools.examples.HelloWorldExample
Type: java application
Rules file: HelloWorld.drl
Objective: demonstrate basic rules in use
```

The "Hello World" example shows a simple example of rules usage, and both the MVEL and Java dialects.

In this example it will be shown how to build knowledgebases and sessions and how to add audit logging and debug outputs, this information is ommitted from other examples as it's all very similar. KnowledgeBuilder is used to turn a drl source file into Package objects which the KnowledgeBase can consume, add takes a Resource interface and ResourceType as parameters. Resource can be used to retrieve a source drl file from various locations, in this case the drl file is being retrieved from the classpath using ResourceFactory; but it could come from the disk or a url. In this case we only add a single drl source file, but multiple drl files can be added. Drl files with different namespaces can be added, KnowledgeBuilder creates a package for each namespace. Multiple packages of different namespaces can be added to the same KnowledgeBase. When all the drl files have been added we should check the builder for errors; while the KnowledgeBase will validate the package it will only have access to the error information as a String, so if you wish to debug the error information you should do it on the builder instance. Once we know the builder is error free get the Package collection, instantiate a KnowledgeBase from the KnowledgeBaseFactory and add the package collection.

### Example 8.1. HelloWorld example: Creating the KnowledgeBase and Session

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

```
// this will parse and compile in one step
kbuilder.add(ResourceFactory.newClassPathResource("HelloWorld.drl",
    HelloWorldExample.class), ResourceType.DRL);

// Check the builder for errors
if (kbuilder.hasErrors()) {
 System.out.println(kbuilder.getErrors().toString());
 throw new RuntimeException("Unable to compile \"HelloWorld.drl\".");
}

// get the compiled packages (which are serializable)
final Collection<KnowledgePackage> pkgs = kbuilder.getKnowledgePackages();

// add the packages to a knowledgebase (deploy the knowledge packages).
final KnowledgeBase kbase = KnowledgeBaseFactory.newKnowledgeBase();
kbase.addKnowledgePackages(pkgs);

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

Drools has an event model that exposes much of what's happening internally, two default debug listeners are supplied DebugAgendaEventListener and DebugWorkingMemoryEventListener which print out debug event information to the err console, adding listeners to a session is trivial and shown below. The KnowledgeRuntimeLogger provides execution auditing which can be viewed in a graphical viewer; it's actually a specialised implementation built on the agenda and working memory listeners, when the engine has finished executing logger.close() must be called.

Most of the examples use the Audit logging features of Drools to record execution flow for later inspection.

### Example 8.2. HelloWorld example: Event logging and Auditing

```
// setup the debug listeners
ksession.addEventListener( new DebugAgendaEventListener() );
ksession.addEventListener( new DebugWorkingMemoryEventListener() );

// setup the audit logging
KnowledgeRuntimeLogger logger =
 KnowledgeRuntimeLoggerFactory.newFileLogger(ksession, "log/helloworld");
```

The single class used in this example is very simple, it has two fields: the message, which is a String and the status which can be either the int HELLO or the int GOODBYE.

### Example 8.3. HelloWorld example: Message Class

```
public static class Message {
    public static final int HELLO   = 0;
    public static final int GOODBYE = 1;

    private String          message;
    private int             status;
    ...
}
```

A single Message object is created with the message "Hello World" and status HELLO and then inserted into the engine, at which point fireAllRules() is executed. Remember all the network evaluation is done during the insert time, by the time the program execution reaches the fireAllRules() method it already knows which rules are fully matches and able to fire.

### Example 8.4. HelloWorld example: Execution

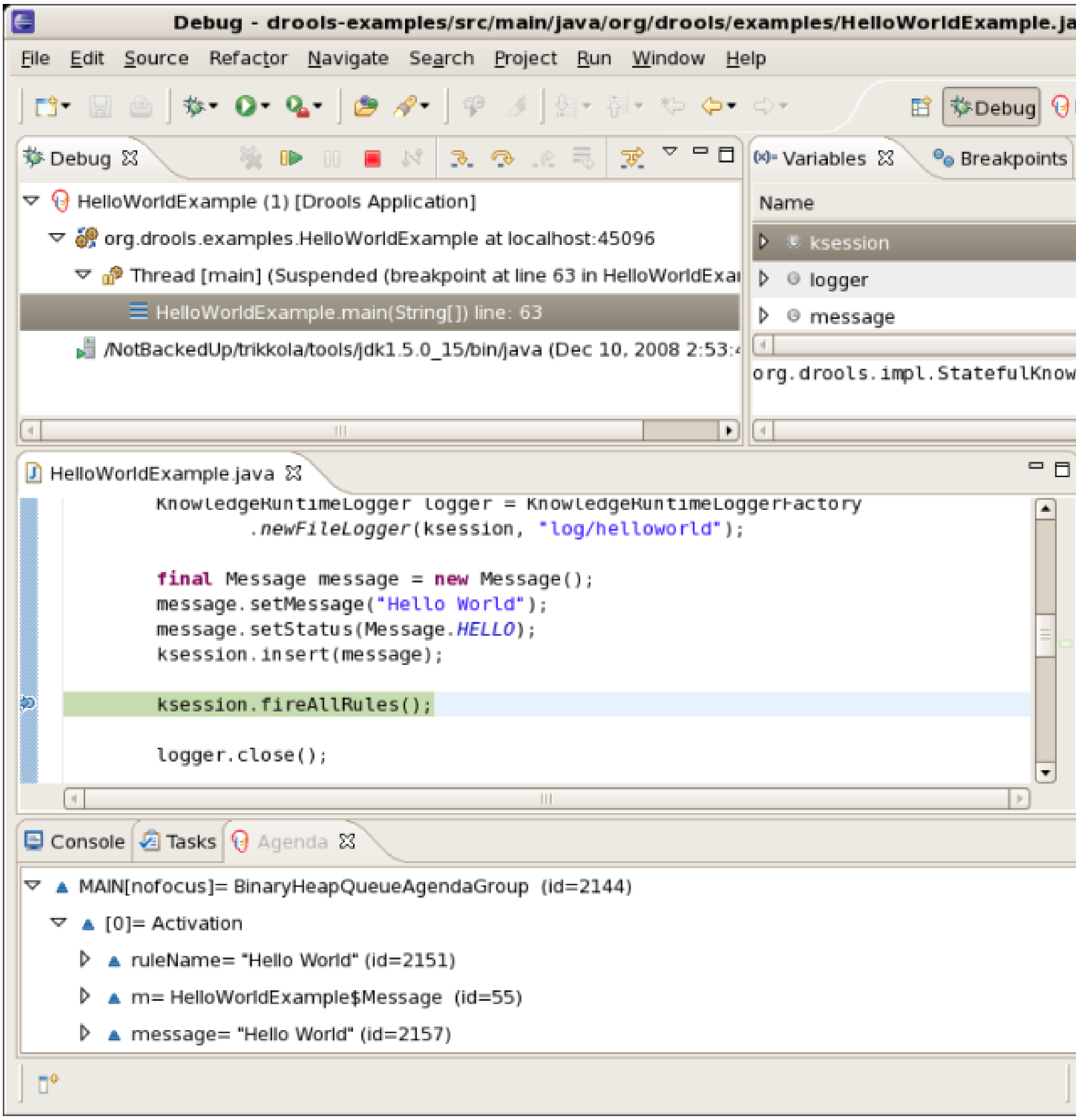
```
final Message message = new Message();
message.setMessage("Hello World");
message.setStatus(Message.HELLO);
ksession.insert(message);

ksession.fireAllRules();

logger.close();

ksession.dispose();
```

To execute the example from Java.

1. Open the class org.drools.examples.HelloWorldExample in your Eclipse IDE

2. Right-click the class an select "Run as..." -> "Java application"

If we put a breakpoint on the fireAllRules() method and select the ksession variable we can see that the "Hello World" view is already activated and on the Agenda, showing that all the pattern matching work was already done during the insert.


Debug - drools-examples/src/main/java/org/drools/examples/HelloWorldExample.ja
File Edit Source Refactor Navigate Search Project Run Window Help
Debug
Debug
Variables
Breakpoints
HelloWorldExample.java
Console
Tasks
Agenda

```
HelloWorldExample (1) [Drools Application]
  org.drools.examples.HelloWorldExample at localhost:45096
    Thread [main] (Suspended (breakpoint at line 63 in HelloWorldExa
      HelloWorldExample.main(String[]) line: 63
  /NotBackedUp/trikkola/tools/jdk1.5.0_15/bin/java (Dec 10, 2008 2:53:
```

```
Name
ksession
logger
message
org.drools.impl.StatefulKnow
```

```
        KnowledgeRuntimeLogger logger = KnowledgeRuntimeLoggerFactory
                .newFileLogger(ksession, "log/helloworld");

        final Message message = new Message();
        message.setMessage("Hello World");
        message.setStatus(Message.HELLO);
        ksession.insert(message);

        ksession.fireAllRules();

        logger.close();
```

```
MAIN[nofocus]= BinaryHeapQueueAgendaGroup  (id=2144)
  [0]= Activation
    ruleName= "Hello World" (id=2151)
    m= HelloWorldExample$Message  (id=55)
    message= "Hello World" (id=2157)
```

## Figure 8.1. Hello World : fireAllRules Agenda View

The may application print outs go to to System.out while the debug listener print outs go to System.err.

### Example 8.5. HelloWorld example: Console.out

```
Hello World
```

```
Goodbye cruel world
```

### Example 8.6. HelloWorld example: Console.err

```
==>[ActivationCreated(0): rule=Hello World;

 tuple=[fid:1:1:org.drools.examples.HelloWorldExample$Message@17cec96]]
[ObjectInserted:
 handle=[fid:1:1:org.drools.examples.HelloWorldExample$Message@17cec96];

 object=org.drools.examples.HelloWorldExample$Message@17cec96]
[BeforeActivationFired: rule=Hello World;

 tuple=[fid:1:1:org.drools.examples.HelloWorldExample$Message@17cec96]]
==>[ActivationCreated(4): rule=Good Bye;

 tuple=[fid:1:2:org.drools.examples.HelloWorldExample$Message@17cec96]]
[ObjectUpdated:
 handle=[fid:1:2:org.drools.examples.HelloWorldExample$Message@17cec96];

 old_object=org.drools.examples.HelloWorldExample$Message@17cec96;

 new_object=org.drools.examples.HelloWorldExample$Message@17cec96]
[AfterActivationFired(0): rule=Hello World]
[BeforeActivationFired: rule=Good Bye;

 tuple=[fid:1:2:org.drools.examples.HelloWorldExample$Message@17cec96]]
[AfterActivationFired(4): rule=Good Bye]
```

The **LHS (when)** section of the rule states that it will be activated for each *Message* object inserted into the working memory whose *status* is *Message.HELLO*. Besides that, two variable binds are created: "*message*" variable is bound to the *message* attribute and "*m*" variable is bound to the *object matched pattern* itself.

The **RHS (consequence, then)** section of the rule is written using the MVEL expression language, as declared by the rule's attribute *dialect*. After printing the content of the *message* bound variable to the default console, the rule changes the values of the *message* and *status* attributes of the *m* bound variable; using MVEL's 'modify' keyword which allows you to apply a block of setters in one statement, with the engine being automatically notified of the changes at the end of the block.

### Example 8.7. HelloWorld example: rule "Hello World"

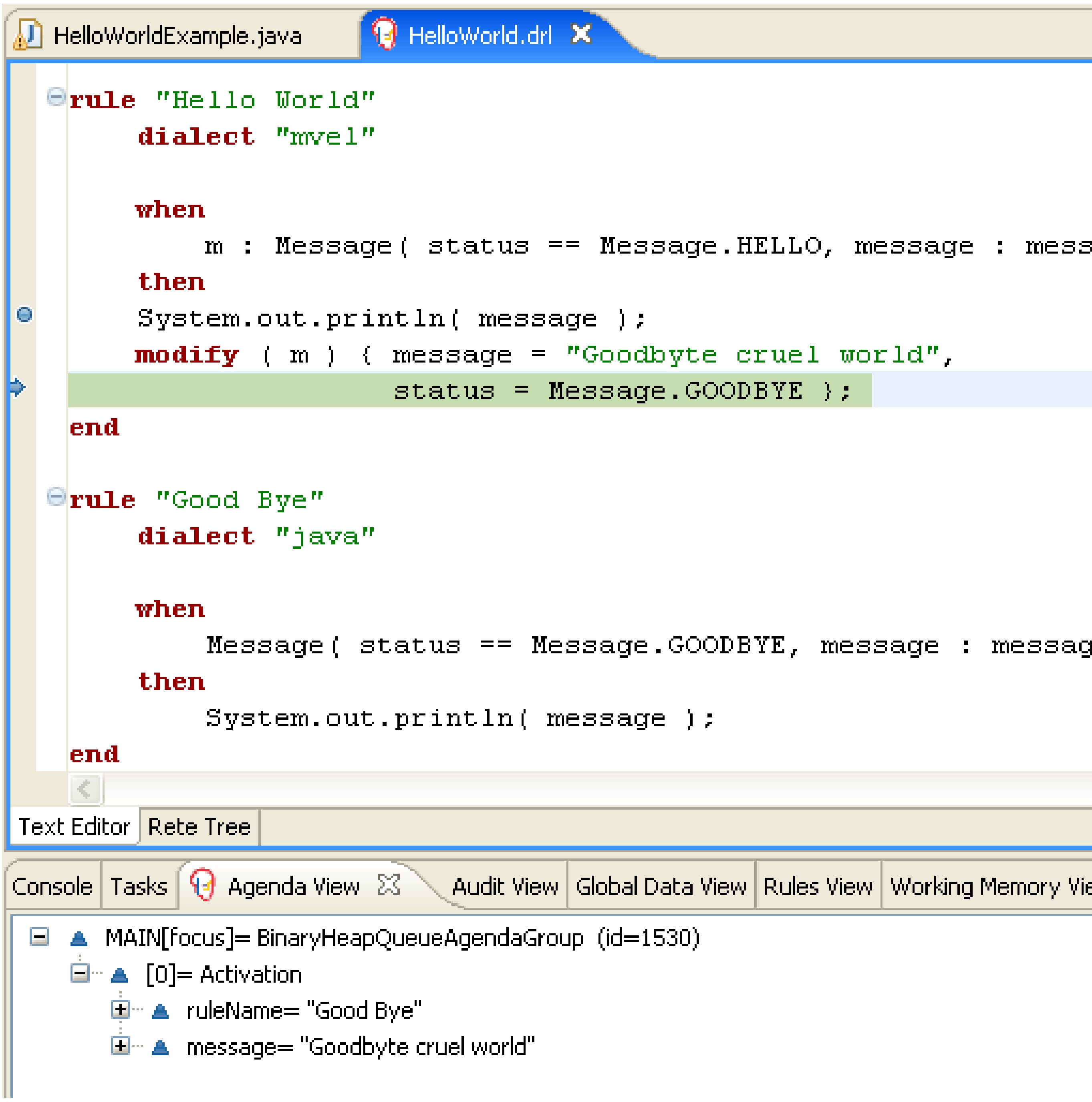
```
rule "Hello World"
      dialect "mvel"
  when
      m : Message( status == Message.HELLO, message : message )
  then
```

```
        System.out.println( message );
        modify ( m ) { message = "Goodbyte cruel world",
                       status = Message.GOODBYE };
  end
```

We can add a break point into the DRL for when modify is called during the execution of the "Hello World" consequence and inspect the Agenda view again. Notice this time we "Debug As" a "Drools application" and not a "Java application".

1. Open the class org.drools.examples.FibonacciExample in your Eclipse IDE
2. Right-click the class an select "Debug as..." -> "Drools application"

Now we can see that the other rule "Good Bye" which uses the java dialect is activated and placed on the agenda.



```
rule "Hello World"
    dialect "mvel"

    when
        m : Message( status == Message.HELLO, message : mess
    then
    System.out.println( message );
    modify ( m ) { message = "Goodbyte cruel world",
                   status = Message.GOODBYE };
end

rule "Good Bye"
    dialect "java"

    when
        Message( status == Message.GOODBYE, message : messag
    then
        System.out.println( message );
end
```

```
MAIN[focus]= BinaryHeapQueueAgendaGroup (id=1530)
    [0]= Activation
        ruleName= "Good Bye"
        message= "Goodbyte cruel world"
```

### Figure 8.2. Hello World : rule "Hello World" Agenda View

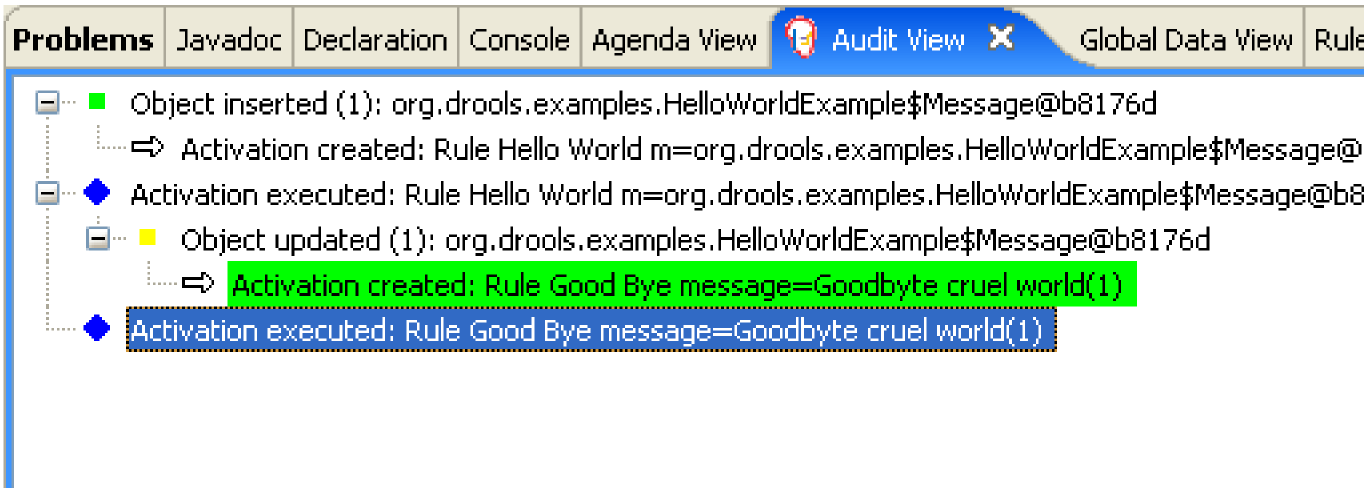
The "Good Bye" rule is similar to the "Hello World" rule but matches Message objects whose status is Message.GOODBYE instead, printing its message to the default console, it specifies the "java" dialect.

### Example 8.8. HelloWorld example: rule "Good Bye"

```
rule "Good Bye"
```

```
        dialect "java"
    when
        Message( status == Message.GOODBYE, message : message )
    then
        System.out.println( message );
end
```

If you remember at the start of this example in the java code we used KnowledgeRuntimeLoggerFactory.newFileLogger to create a KnowledgeRuntimeLogger and called logger.close() at the end, this created an audit log file that can be shown in the Audit view. We use the audit view in many of the examples to try and understand the example execution flow. In the view below we can see the object is inserted which creates an activation for the "Hello World" rule, the activation is then executed which updated the Message object causing the "Good Bye" rule to activate, the "Good Bye" rule then also executes. When an event in the Audit view is select it highlights the origin event in green, so below the Activation created event is highlighted in greed as the origin of the Activation executed event.



```
Object inserted (1): org.drools.examples.HelloWorldExample$Message@b8176d
    Activation created: Rule Hello World m=org.drools.examples.HelloWorldExample$Message@
Activation executed: Rule Hello World m=org.drools.examples.HelloWorldExample$Message@b8
    Object updated (1): org.drools.examples.HelloWorldExample$Message@b8176d
        Activation created: Rule Good Bye message=Goodbyte cruel world(1)
Activation executed: Rule Good Bye message=Goodbyte cruel world(1)
```

**Figure 8.3. Hello World : Audit View**

## 8.3. State Example

This example is actually implemented in three different versions to demonstrate different ways of implementing the same basic behavior: rules forward chaining, i.e., the ability the engine has to evaluate, activate and fire rules in sequence, based on changes on the facts in the working memory.

### 8.3.1. Understanding the State Example

```
Name: State Example
Main class: org.drools.examples.StateExampleUsingSalience
Type: java application
```

```
Rules file: StateExampleUsingSalience.drl
Objective: Demonstrates basic rule use and Conflict Resolution for rule
 firing priority.
```

Each State class has fields for its name and its current state (see org.drools.examples.State class). The two possible states for each objects are:

- NOTRUN
- FINISHED

### Example 8.9. State Class

```
public class State {
    public static final int       NOTRUN   = 0;
    public static final int       FINISHED = 1;

    private final PropertyChangeSupport changes  = new
 PropertyChangeSupport( this );

    private String                name;
    private int                   state;

    ... setters and getters go here...
}
```

Ignore the PropertyChangeSupport for now, that will be explained later. In the example we create four State objects with names: A, B, C and D. Initially all are set to state NOTRUN, which is default for the used constructor. Each instance is asserted in turn into the session and then fireAllRules() is called.

### Example 8.10. Salience State Example Execution

```
State a = new State( "A" );
State b = new State( "B" );
State c = new State( "C" );
final State d = new State( "D" );

// By setting dynamic to TRUE, Drools will use JavaBean
// PropertyChangeListeners so you don't have to call update().
boolean dynamic = true;

session.insert( a,
                dynamic );
session.insert( b,
                dynamic );
```

```
session.insert( c,
                dynamic );
session.insert( d,
                dynamic );

session.fireAllRules();
session.dispose(); // Stateful rule session must always be disposed when
 finished
```

To execute the application:

1. Open the class org.drools.examples.StateExampleUsingSalience in your Eclipse IDE
2. Right-click the class an select "Run as..." -> "Java application"

And you will see the following output in the Eclipse console output:

### Example 8.11. Salience State Example Console Output

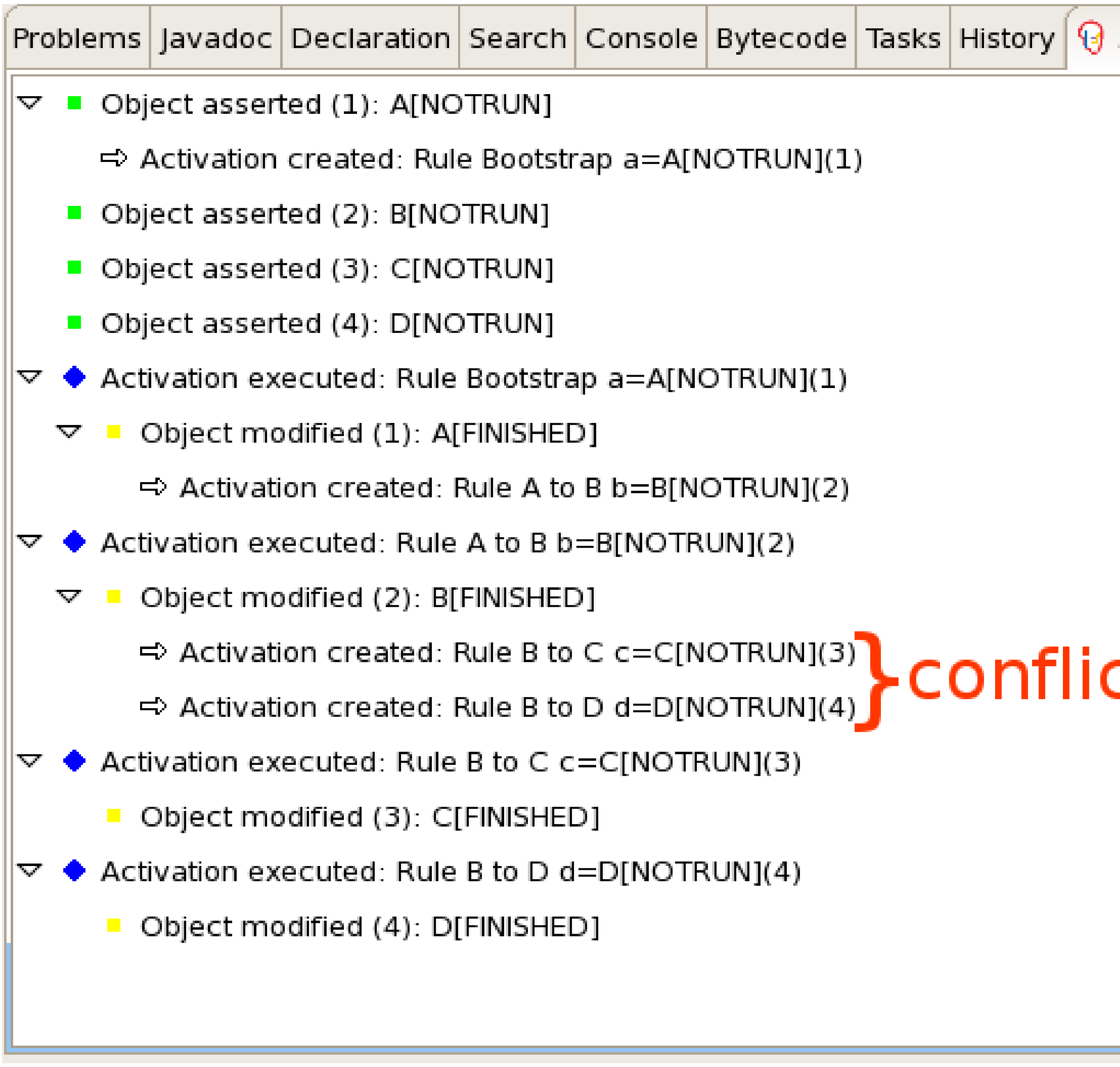
```
A finished
B finished
C finished
D finished
```

There are four rules in total, first a Bootstrap rule fires setting A to state FINISHED which then causes B to change to state FINISHED. C and D are both dependent on B - causing a conflict which is resolved by setting salience values. First lets look at how this was executed

The best way to understand what is happening is to use the "Audit Log" feature to graphically see the results of each operation. The Audit log was generated when the example was previously run. To view the Audit log in Eclipse:

1. If the "Audit View" is not visible, click on: "Window"->"Show View"->"Other..."->"Drools"->"Audit View"
2. In the "Audit View" click in the "Open Log" button and select the file "<drools-examples-drl-dir>/log/state.log"

After that, the "Audit view" will look like the following screenshot.



```
▽ ■ Object asserted (1): A[NOTRUN]
      ⇨ Activation created: Rule Bootstrap a=A[NOTRUN](1)
  ■ Object asserted (2): B[NOTRUN]
  ■ Object asserted (3): C[NOTRUN]
  ■ Object asserted (4): D[NOTRUN]
▽ ◆ Activation executed: Rule Bootstrap a=A[NOTRUN](1)
  ▽ ■ Object modified (1): A[FINISHED]
        ⇨ Activation created: Rule A to B b=B[NOTRUN](2)
▽ ◆ Activation executed: Rule A to B b=B[NOTRUN](2)
  ▽ ■ Object modified (2): B[FINISHED]
        ⇨ Activation created: Rule B to C c=C[NOTRUN](3)
        ⇨ Activation created: Rule B to D d=D[NOTRUN](4)
▽ ◆ Activation executed: Rule B to C c=C[NOTRUN](3)
      ■ Object modified (3): C[FINISHED]
▽ ◆ Activation executed: Rule B to D d=D[NOTRUN](4)
      ■ Object modified (4): D[FINISHED]
```

**Figure 8.4. Salience State Example Audit View**

Reading the log in the "Audit View", top to down, we see every action and the corresponding changes in the working memory. This way we see that the assertion of the State "A" object with the "NOTRUN" state activates the "Bootstrap" rule, while the assertions of the other state objects have no immediate effect.

### Example 8.12. Salience State Example: Rule "Bootstrap"

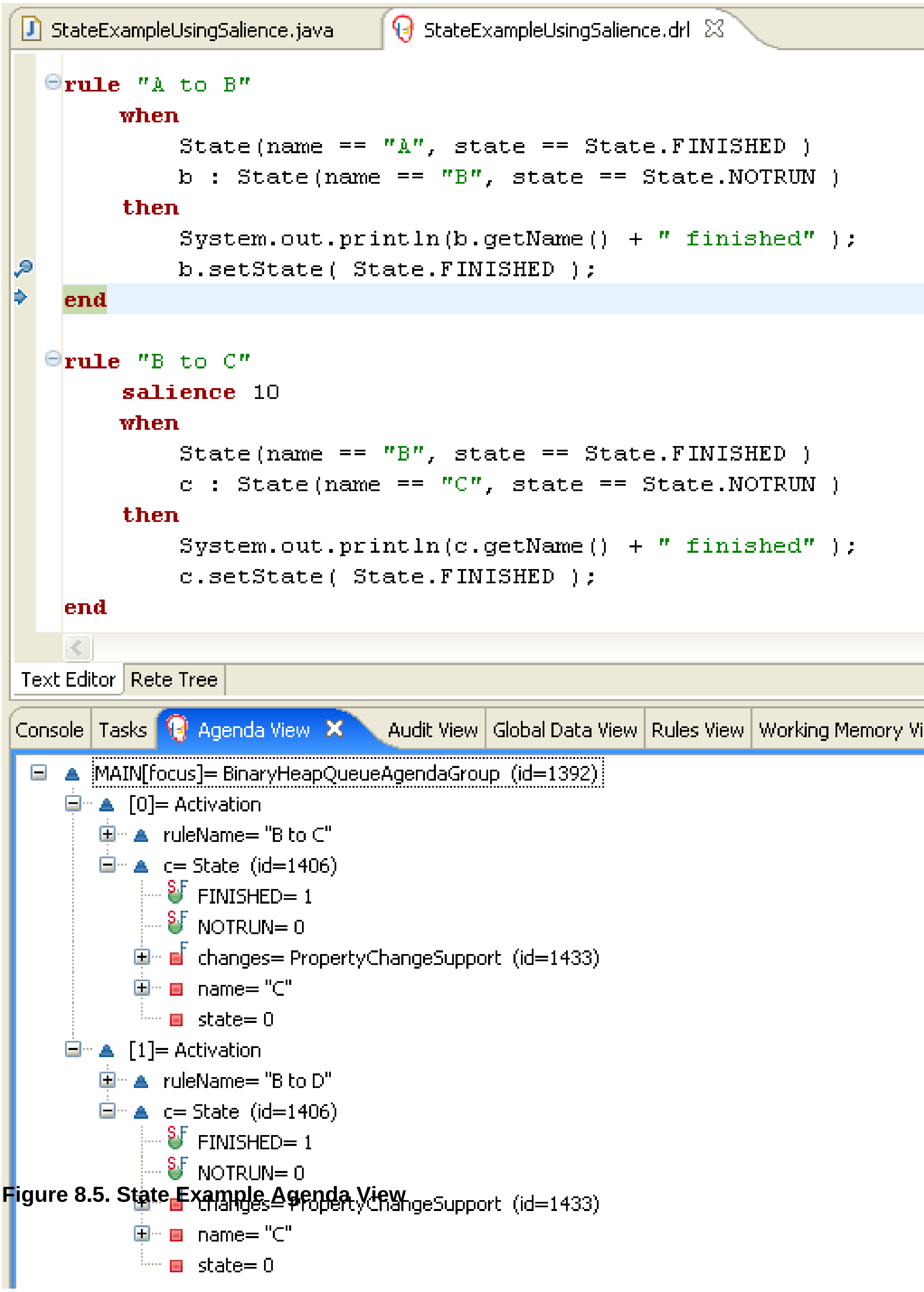
```
rule Bootstrap
    when
        a : State(name == "A", state == State.NOTRUN )
    then
        System.out.println(a.getName() + " finished" );
        a.setState( State.FINISHED );
end
```

The execution of "Bootstrap" rule changes the state of "A" to "FINISHED", that in turn activates the "A to B" rule.

### Example 8.13. Salience State Example: Rule "A to B"

```
rule "A to B"
    when
        State(name == "A", state == State.FINISHED )
        b : State(name == "B", state == State.NOTRUN )
    then
        System.out.println(b.getName() + " finished" );
        b.setState( State.FINISHED );
end
```

The execution of "A to B" rule changes the state of "B" to "FINISHED", which activates both rules "B to C" and "B to D", placing both Activations onto the Agenda. In this moment the two rules may fire and are said to be in conflict. The conflict resolution strategy allows the engine's Agenda to decide which rule to fire. As the "B to C" rule has a **higher salience value** (10 versus the default salience value of 0), it fires first, modifying the "C" object to state "FINISHED". The Audit view above shows the modification of the State object in the rule "A to B" which results in two highlighted activations being in conflict. The Agenda view can also be used to investigate the state of the Agenda, debug points can be placed in the rules themselves and the Agenda view opened; the screen shot below shows the break point in the rule "A to B" and the state of the Agenda with the two conflicting rules.



```
rule "A to B"
    when
        State(name == "A", state == State.FINISHED )
        b : State(name == "B", state == State.NOTRUN )
    then
        System.out.println(b.getName() + " finished" );
        b.setState( State.FINISHED );
end

rule "B to C"
    salience 10
    when
        State(name == "B", state == State.FINISHED )
        c : State(name == "C", state == State.NOTRUN )
    then
        System.out.println(c.getName() + " finished" );
        c.setState( State.FINISHED );
end
```

```
MAIN[focus]= BinaryHeapQueueAgendaGroup (id=1392)
    [0]= Activation
        ruleName= "B to C"
        c= State (id=1406)
            FINISHED= 1
            NOTRUN= 0
            changes= PropertyChangeSupport (id=1433)
            name= "C"
            state= 0
    [1]= Activation
        ruleName= "B to D"
        c= State (id=1406)
            FINISHED= 1
            NOTRUN= 0
            changes= PropertyChangeSupport (id=1433)
            name= "C"
            state= 0
```

**Figure 8.5. State Example Agenda View**

### Example 8.14. Salience State Example: Rule "B to C"

```
rule "B to C"
        salience 10
    when
        State(name == "B", state == State.FINISHED )
        c : State(name == "C", state == State.NOTRUN )
    then
        System.out.println(c.getName() + " finished" );
        c.setState( State.FINISHED );
end
```

The "B to D" rule fires last, modifying the "D" object to state "FINISHED".

### Example 8.15. Salience State Example: Rule "B to D"

```
rule "B to D"
    when
        State(name == "B", state == State.FINISHED )
        d : State(name == "D", state == State.NOTRUN )
    then
        System.out.println(d.getName() + " finished" );
        d.setState( State.FINISHED );
end
```

There are no more rules to execute and so the engine stops.

Another notable concept in this example is the use of **dynamic facts**, which is the PropertyChangeListener part. As mentioned previously in the documentation, in order for the engine to see and react to fact's properties change, the application must tell the engine that changes occurred. This can be done explicitly in the rules, by calling the **update()** memory action, or implicitly by letting the engine know that the facts implement PropertyChangeSupport as defined by the *Javabeans specification*. This example demonstrates how to use PropertyChangeSupport to avoid the need for explicit update() calls in the rules. To make use of this feature, make sure your facts implement the PropertyChangeSupport as the org.drools.example.State class does and use the following code to insert the facts into the working memory:

### Example 8.16. Inserting a Dynamic Fact

```
// By setting dynamic to TRUE, Drools will use JavaBean
// PropertyChangeListeners so you don't have to call update().
final boolean dynamic = true;

session.insert( fact,
                dynamic );
```

When using PropertyChangeListeners each setter must implement a little extra code to do the notification, here is the state setter for thte org.drools.examples.State class:

### Example 8.17. Setter Example with PropertyChangeSupport

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

There are two other State examples: StateExampleUsingAgendGroup and StateExampleWithDynamicRules. Both execute from A to B to C to D, as just shown, the StateExampleUsingAgendGroup uses agenda-groups to control the rule conflict and which one fires first and StateExampleWithDynamicRules shows how an additional rule can be added to an already running WorkingMemory with all the existing data applying to it at runtime.

Agenda groups are a way to partition the agenda into groups and controlling which groups can execute. All rules by default are in the "MAIN" agenda group, by simply using the "agenda-group" attribute you specify a different agenda group for the rule. A working memory initially only has focus on the "MAIN" agenda group, only when other groups are given the focus can their rules fire; this can be achieved by either using the method setFocus() or the rule attribute "auto-focus". "auto-focus" means that the rule automatically sets the focus to it's agenda group when the rule is matched and activated. It is this "auto-focus" that enables "B to C" to fire before "B to D".

### Example 8.18. Agenda Group State Example: Rule "B to C"

```
rule "B to C"
      agenda-group "B to C"
      auto-focus true
  when
      State(name == "B", state == State.FINISHED )
      c : State(name == "C", state == State.NOTRUN )
  then
      System.out.println(c.getName() + " finished" );
      c.setState( State.FINISHED );
      drools.setFocus( "B to D" );
end
```

The rule "B to C" calls "drools.setFocus( "B to D" );" which gives the agenda group "B to D" focus allowing its active rules to fire; which allows the rule "B to D" to fire.

### Example 8.19. Agenda Group State Example: Rule "B to D"

```
rule "B to D"
      agenda-group "B to D"
  when
      State(name == "B", state == State.FINISHED )
      d : State(name == "D", state == State.NOTRUN )
  then
      System.out.println(d.getName() + " finished" );
      d.setState( State.FINISHED );
end
```

The example StateExampleWithDynamicRules adds another rule to the RuleBase after fireAllRules(), the rule it adds is just another State transition.

### Example 8.20. Dynamic State Example: Rule "D to E"

```
rule "D to E"
  when
      State(name == "D", state == State.FINISHED )
      e : State(name == "E", state == State.NOTRUN )
  then
      System.out.println(e.getName() + " finished" );
      e.setState( State.FINISHED );
end
```

It gives the following expected output:

### Example 8.21. Dynamic Sate Example Output

```
A finished
B finished
C finished
D finished
E finished
```

## 8.4. Fibonacci Example

```
Name: Fibonacci
Main class: org.drools.examples.FibonacciExample
Type: java application
Rules file: Fibonacci.drl
Objective: Demonsrates Recursion, 'not' CEs and Cross Product Matching
```

The Fibonacci Numbers, *http://en.wikipedia.org/wiki/Fibonacci_number*, invented by Leonardo of Pisa, *http://en.wikipedia.org/wiki/Fibonacci*, are obtained by starting with 0 and 1, and then produce the next Fibonacci number by adding the two previous Fibonacci numbers. The first Fibonacci numbers for n = 0, 1,... are: * 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987, 1597, 2584, 4181, 6765, 10946... The Fibonacci Example demonstrates recursion and conflict resolution with Salience values.

A single fact Class is used in this example, Fibonacci. It has two fields, sequence and value. The sequence field is used to indicate the position of the object in the Fibonacci number sequence and the value field shows the value of that Fibonacci object for that sequence position.

### Example 8.22. Fibonacci Class

```
public static class Fibonacci {
    private int  sequence;
    private long value;

    ... setters and getters go here...
}
```

Execute the example:

1. Open the class `org.drools.examples.FibonacciExample` in your Eclipse IDE
2. Right-click the class an select "Run as..." -> "Java application"

And Eclipse shows the following output in its console, "...snip..." shows repeated bits removed to save space:

### Example 8.23. Fibonacci Example Console Output

```
recurse for 50
recurse for 49
recurse for 48
recurse for 47
...snip...
recurse for 5
recurse for 4
recurse for 3
recurse for 2
1 == 1
2 == 1
3 == 2
4 == 3
5 == 5
6 == 8
...snip...
47 == 2971215073
48 == 4807526976
49 == 7778742049
50 == 12586269025
```

To kick this off from java we only insert a single Fibonacci object, with a sequence of 50, a recurse rule is then used to insert the other 49 Fibonacci objects. This example doesn't use PropertyChangeSupport and uses the MVEL dialect, this means we can use the **modify** keyword, which allows a block setter action which also notifies the engine of changes.

### Example 8.24. Fibonacci Example Execution

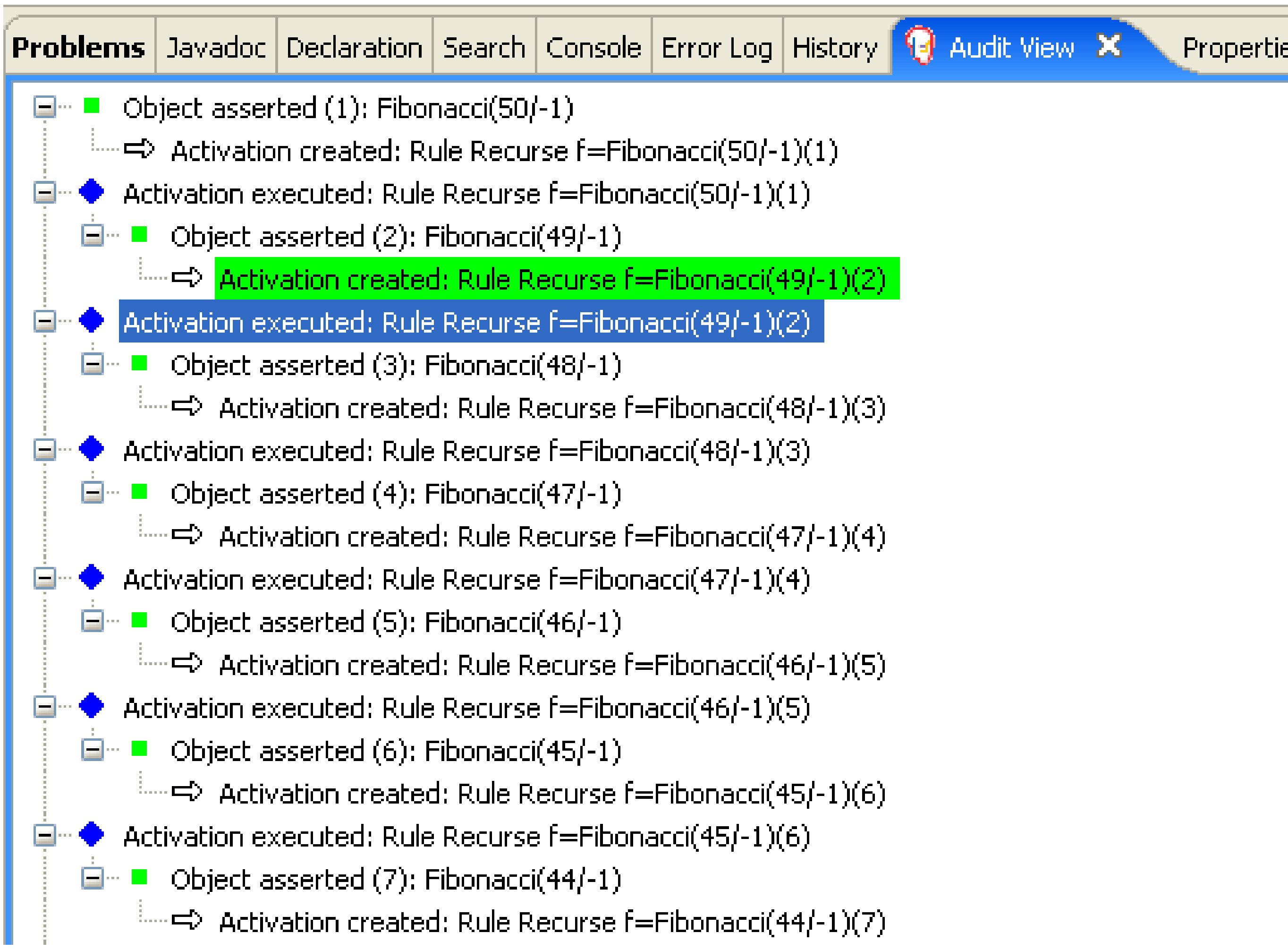
```
ksession.insert( new Fibonacci( 50 ) );
ksession.fireAllRules();
```

The recurse rule is very simple, it matches each asserted Fibonacci object with a value of -1, it then creates and asserts a new Fibonacci object with a sequence of one less than the currently matched object. Each time a Fibonacci object is added, as long as one with a "sequence == 1" does not exist, the rule re-matches again and fires; causing the recursion. The 'not' conditional element is used to stop the rule matching once we have all 50 Fibonacci objects in memory. The rule also has a salience value, this is because we need to have all 50 Fibonacci objects asserted before we execute the Bootstrap rule.

### Example 8.25. Fibonacci Example : Rule "Recurse"

```
rule Recurse
    salience 10
    when
        f : Fibonacci ( value == -1 )
        not ( Fibonacci ( sequence == 1 ) )
    then
        insert( new Fibonacci( f.sequence - 1 ) );
        System.out.println( "recurse for " + f.sequence );
end
```

The audit view shows the original assertion of the Fibonacci object with a sequence of 50, this was done from Java land. From there the audit view shows the continual recursion of the rule, each asserted Fibonacci causes the "Recurse" rule to become activate again, which then fires.



```
Object asserted (1): Fibonacci(50/-1)
    Activation created: Rule Recurse f=Fibonacci(50/-1)(1)
Activation executed: Rule Recurse f=Fibonacci(50/-1)(1)
    Object asserted (2): Fibonacci(49/-1)
        Activation created: Rule Recurse f=Fibonacci(49/-1)(2)
Activation executed: Rule Recurse f=Fibonacci(49/-1)(2)
    Object asserted (3): Fibonacci(48/-1)
        Activation created: Rule Recurse f=Fibonacci(48/-1)(3)
Activation executed: Rule Recurse f=Fibonacci(48/-1)(3)
    Object asserted (4): Fibonacci(47/-1)
        Activation created: Rule Recurse f=Fibonacci(47/-1)(4)
Activation executed: Rule Recurse f=Fibonacci(47/-1)(4)
    Object asserted (5): Fibonacci(46/-1)
        Activation created: Rule Recurse f=Fibonacci(46/-1)(5)
Activation executed: Rule Recurse f=Fibonacci(46/-1)(5)
    Object asserted (6): Fibonacci(45/-1)
        Activation created: Rule Recurse f=Fibonacci(45/-1)(6)
Activation executed: Rule Recurse f=Fibonacci(45/-1)(6)
    Object asserted (7): Fibonacci(44/-1)
        Activation created: Rule Recurse f=Fibonacci(44/-1)(7)
```

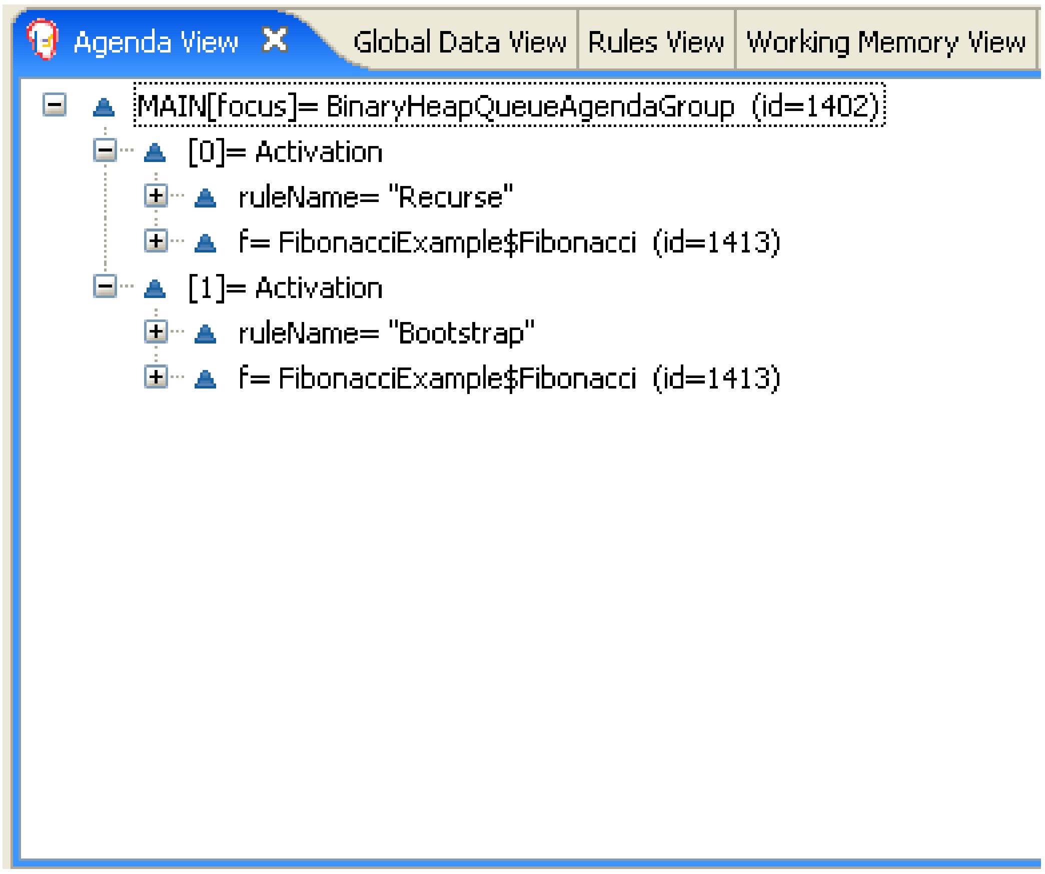
### Figure 8.6. Fibonacci Example "Recurse" Audit View 1

When a Fibonacci with a sequence of 2 is asserted the "Bootstrap" rule is matched and activated along with the "Recurse" rule.

### Example 8.26. Fibonacci Example : Rule "Bootstrap"

```
rule Bootstrap
    when
        f : Fibonacci( sequence == 1 || == 2, value == -1 ) // this is a
 multi-restriction || on a single field
    then
        modify ( f ){ value = 1 };
        System.out.println( f.sequence + " == " + f.value );
end
```

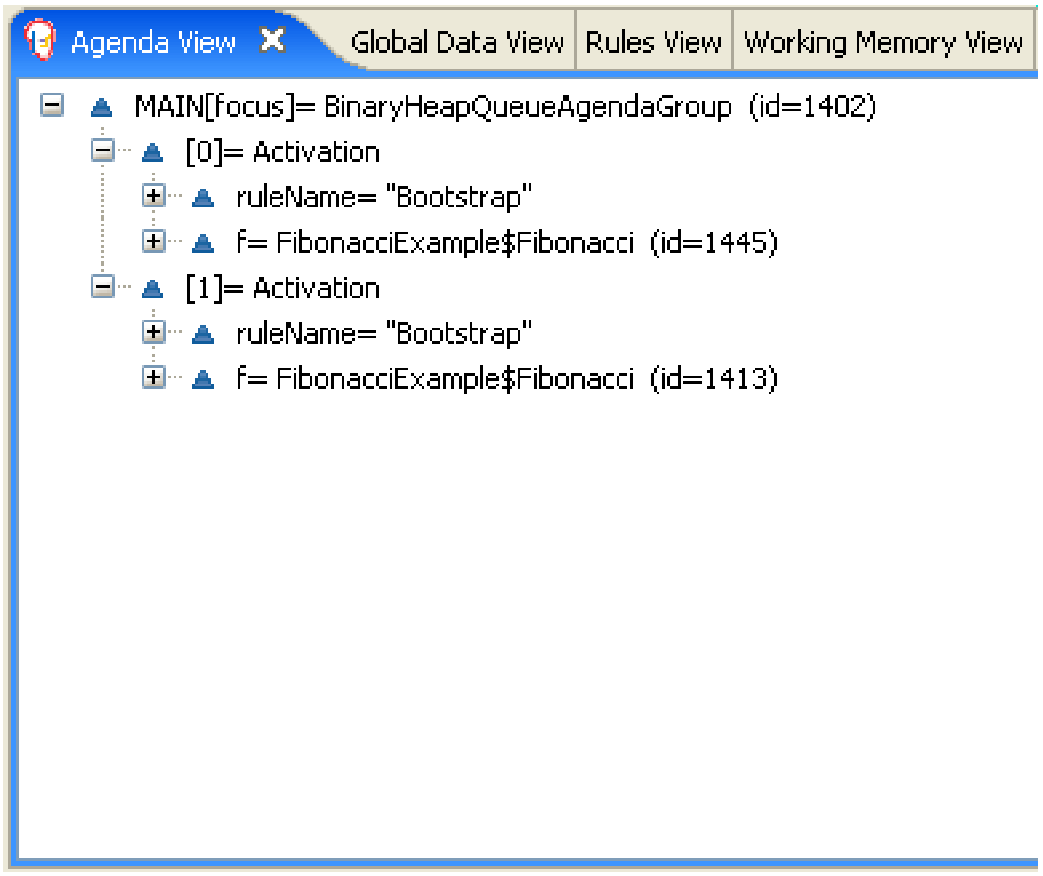
At this point the Agenda looks like the figure shown below. However the "Bootstrap" rule does not fire as the "Recurse" rule has a higher salience.



```
MAIN[focus]= BinaryHeapQueueAgendaGroup  (id=1402)
    [0]= Activation
        ruleName= "Recurse"
        f= FibonacciExample$Fibonacci  (id=1413)
    [1]= Activation
        ruleName= "Bootstrap"
        f= FibonacciExample$Fibonacci  (id=1413)
```

## Figure 8.7. Fibonacci Example "Recurse" Agenda View 1

When a Fibonacci with a sequence of 1 is asserted the "Bootstrap" rule is matched again, causing two activations for this rule; note that the "Recurse" rule does not match and activate because the 'not conditional element stops the rule matching when a Fibonacci with a sequence of 1 exists.



```
MAIN[focus]= BinaryHeapQueueAgendaGroup  (id=1402)
    [0]= Activation
        ruleName= "Bootstrap"
        f= FibonacciExample$Fibonacci  (id=1445)
    [1]= Activation
        ruleName= "Bootstrap"
        f= FibonacciExample$Fibonacci  (id=1413)
```

## Figure 8.8. Fibonacci Example "Recurse" Agenda View 2

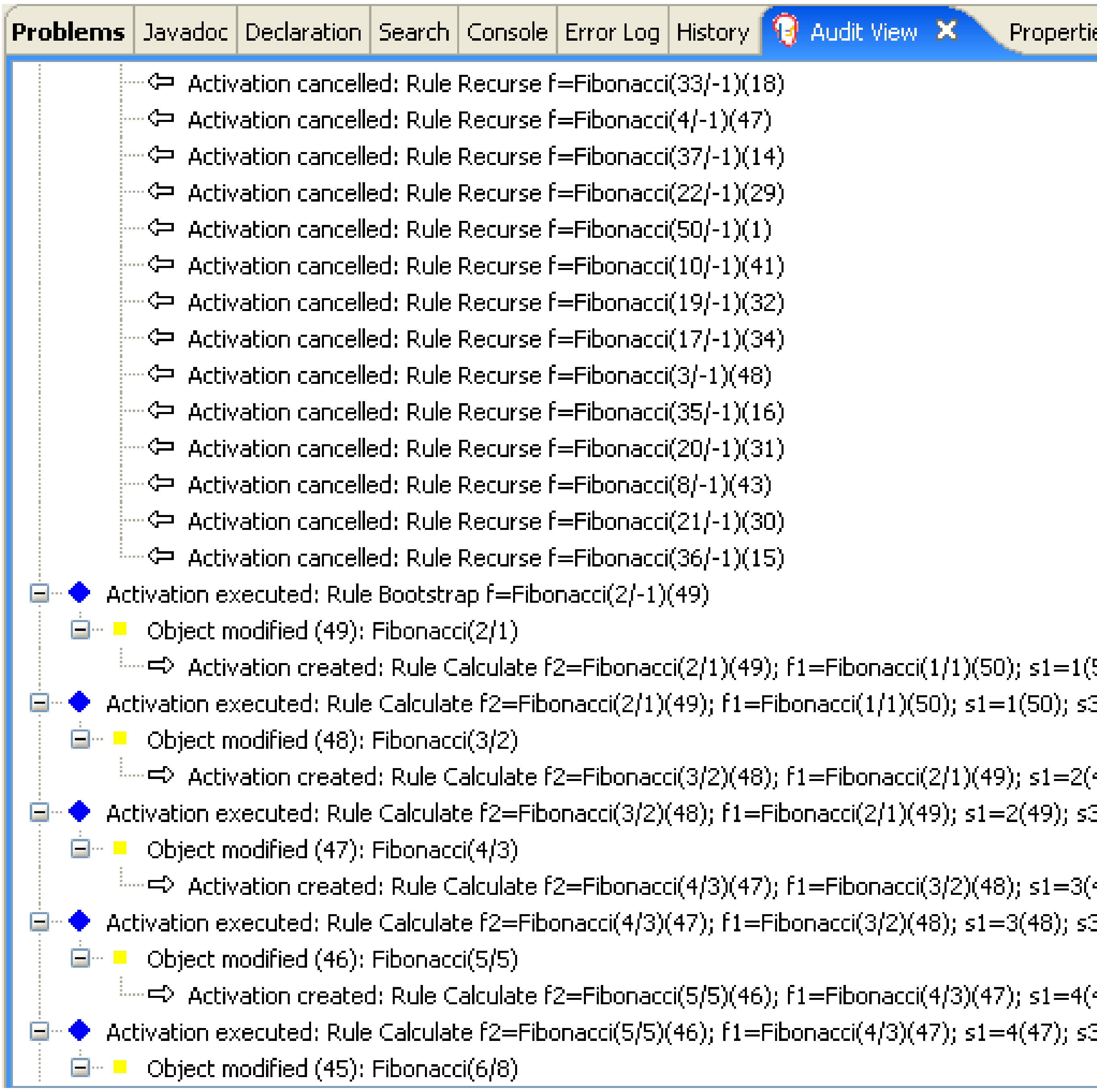
Once we have two Fibonacci objects both with values not equal to -1 the "calculate" rule is able to match; remember it was the "Bootstrap" rule that set the Fibonacci's with sequences 1 and 2 to values of 1. At this point we have 50 Fibonacci objects in the Working Memory and we some how need to select the correct ones to calculate each of their values in turn. With three Fibonacci patterns in a rule with no field constriants to correctly constrain the available cross products we have 50x50x50 possible permutations, thats 125K possible rule firings. The "Calculate" rule uses the field constraints to correctly constraint the thee Fibonacci patterns and in the correct order; this technique is called "cross product matching". The first pattern finds any Fibonacci with a value != -1 and binds both the pattern and the field. The second Fibonacci does too but it adds an additional field constraint to make sure that its sequence is one greater than the Fibonacci bound to f1. When this rule first fires we know that only sequences 1 and 2 have values of 1 and the two constraints ensure that f1 references sequence 1 and f2 references sequence2. The final pattern finds the Fibonacci of a value == -1 with a sequence one greater than f2. At this point we have three Fibonacci objects correctly selected from the available cross products and we can do the maths calculating the value for Fibonacci sequence = 3.

## Example 8.27. Fibonacci Example : Rule "Calculate"

```
rule Calculate
    when
```

```
        f1 : Fibonacci( s1 : sequence, value != -1 ) // here we bind
 sequence
        f2 : Fibonacci( sequence == (s1 + 1 ), value != -1 ) // here we
 don't, just to demonstrate the different way bindings can be used
        f3 : Fibonacci( s3 : sequence == (f2.sequence + 1 ), value == -1 )

    then
        modify ( f3 ) { value = f1.value + f2.value };
        System.out.println( s3 + " == " + f3.value ); // see how you can
 access pattern and field  bindings
end
```

The MVEL modify keyword updated the value of the Fibonacci object bound to f3, this means we have a new Fibonacci object with a value != -1, this allows the "Calculate" rule to rematch and calculate the next Fibonacci number. The Audit view below shows the how the firing of the last "Bootstrap" modifies the Fibonacci object enabling the "Calculate" rule to match, which then modifies another Fibonacci object allowing the "Calculate" rule to rematch. This continues till the value is set for all Fibonacci objects.



```
Activation cancelled: Rule Recurse f=Fibonacci(33/-1)(18)
Activation cancelled: Rule Recurse f=Fibonacci(4/-1)(47)
Activation cancelled: Rule Recurse f=Fibonacci(37/-1)(14)
Activation cancelled: Rule Recurse f=Fibonacci(22/-1)(29)
Activation cancelled: Rule Recurse f=Fibonacci(50/-1)(1)
Activation cancelled: Rule Recurse f=Fibonacci(10/-1)(41)
Activation cancelled: Rule Recurse f=Fibonacci(19/-1)(32)
Activation cancelled: Rule Recurse f=Fibonacci(17/-1)(34)
Activation cancelled: Rule Recurse f=Fibonacci(3/-1)(48)
Activation cancelled: Rule Recurse f=Fibonacci(35/-1)(16)
Activation cancelled: Rule Recurse f=Fibonacci(20/-1)(31)
Activation cancelled: Rule Recurse f=Fibonacci(8/-1)(43)
Activation cancelled: Rule Recurse f=Fibonacci(21/-1)(30)
Activation cancelled: Rule Recurse f=Fibonacci(36/-1)(15)
Activation executed: Rule Bootstrap f=Fibonacci(2/-1)(49)
Object modified (49): Fibonacci(2/1)
Activation created: Rule Calculate f2=Fibonacci(2/1)(49); f1=Fibonacci(1/1)(50); s1=1(
Activation executed: Rule Calculate f2=Fibonacci(2/1)(49); f1=Fibonacci(1/1)(50); s1=1(50); s3
Object modified (48): Fibonacci(3/2)
Activation created: Rule Calculate f2=Fibonacci(3/2)(48); f1=Fibonacci(2/1)(49); s1=2(
Activation executed: Rule Calculate f2=Fibonacci(3/2)(48); f1=Fibonacci(2/1)(49); s1=2(49); s3
Object modified (47): Fibonacci(4/3)
Activation created: Rule Calculate f2=Fibonacci(4/3)(47); f1=Fibonacci(3/2)(48); s1=3(
Activation executed: Rule Calculate f2=Fibonacci(4/3)(47); f1=Fibonacci(3/2)(48); s1=3(48); s3
Object modified (46): Fibonacci(5/5)
Activation created: Rule Calculate f2=Fibonacci(5/5)(46); f1=Fibonacci(4/3)(47); s1=4(
Activation executed: Rule Calculate f2=Fibonacci(5/5)(46); f1=Fibonacci(4/3)(47); s1=4(47); s3
Object modified (45): Fibonacci(6/8)
```

**Figure 8.9. Fibonacci Example "Bootstrap" Audit View 1**

## 8.5. Banking Tutorial

```
Name: BankingTutorial
Main class: org.drools.tutorials.banking.*
Type: java application
Rules file: org.drools.tutorials.banking.*
```

```
Objective: tutorial that builds up knowledge of pattern matching, basic
 sorting and calculation rules.
```

This tutorial will demonstrate the process of developing a complete personal banking application that will handle credits, debits, currencies and that will use a set of design patterns that have been created for the process. In order to make the examples documented here clear and modular, I will try and steer away from re-visiting existing code to add new functionality, and will instead extend and inject where appropriate.

The RuleRunner class is a simple harness to execute one or more drls against a set of data. It compiles the Packages and creates the KnowledgeBase for each execution, this allows us to easy execute each scenario and see the outputs. In reality this is not a good solution for a production system where the KnowledgeBase should be built just once and cached, but for the purposes of this tutorial it shall suffice.

## Example 8.28. Banking Tutorial : RuleRunner

```
public class RuleRunner {

    public RuleRunner() {
    }

    public void runRules(String[] rules,
                         Object[] facts) throws Exception {

        KnowledgeBase kbase = KnowledgeBaseFactory.newKnowledgeBase();
        KnowledgeBuilder kbuilder =
 KnowledgeBuilderFactory.newKnowledgeBuilder();

        for ( int i = 0; i < rules.length; i++ ) {
            String ruleFile = rules[i];
            System.out.println( "Loading file: " + ruleFile );
            kbuilder.add( ResourceFactory.newClassPathResource( ruleFile,

 RuleRunner.class ),
                                      ResourceType.DRL );
        }

        Collection<KnowledgePackage> pkgs = kbuilder.getKnowledgePackages();
        kbase.addKnowledgePackages( pkgs );
        StatefulKnowledgeSession ksession =
 kbase.newStatefulKnowledgeSession();

        for ( int i = 0; i < facts.length; i++ ) {
            Object fact = facts[i];
            System.out.println( "Inserting fact: " + fact );
            ksession.insert( fact );
        }
        ksession.fireAllRules();
    }
}
```

This is our first Example1.java class it loads and executes a single drl file "Example.drl" but inserts no data.

### Example 8.29. Banking Tutorial : Java Example1

```
public class Example1 {
    public static void main(String[] args) throws Exception {
        new RuleRunner().runRules( new String[] { "Example1.drl" },
                                   new Object[0] );
    }
}
```

And this is the first simple rule to execute. It has a single "eval" condition that will alway be true, thus this rul will always match and fire.

### Example 8.30. Banking Tutorial : Rule Example1

```
rule "Rule 01"
    when
        eval (1==1)
    then
        System.out.println("Rule 01 Works");
endh
```

The output for the rule is below, the rule matches and executes the single print statement.

### Example 8.31. Banking Tutorial : Output Example1

```
Loading file: Example1.drl
Rule 01 Works
```

The next step is to assert some simple facts and print them out.

### Example 8.32. Banking Tutorial : Java Example2

```
public class Example2 {
    public static void main(String[] args) throws Exception {
        Number[] numbers = new Number[] {wrap(3), wrap(1), wrap(4), wrap(1),
 wrap(5)};
        new RuleRunner().runRules( new String[] { "Example2.drl" },
                                   numbers );
    }

    private static Integer wrap(int i) {
        return new Integer(i);
    }
}
```

This doesn't use any specific facts but instead asserts a set of java.lang.Integer's. This is not considered "best practice" as a number of a collection is not a fact, it is not a thing. A Bank acount has a number, its balance, thus the Account is the fact; but to get started asserting Integers shall suffice for demonstration purposes as the complexity is built up.

Now we will create a simple rule to print out these numbers.

### Example 8.33. Banking Tutorial : Rule Example2

```
rule "Rule 02"
    when
        Number( $intValue : intValue )
    then
        System.out.println("Number found with value: " + $intValue);
end
```

Once again, this rule does nothing special. It identifies any facts that are Numbers and prints out the values. Notice the user of interfaces here, we inserted Integers but the pattern matching engine is able to match the interfaces and super classes of the asserted objects.

The output shows the drl being loaded, the facts inserted and then the matched and fired rules. We can see that each inserted number is matched and fired and thus printed.

### Example 8.34. Banking Tutorial : Output Example2

```
Loading file: Example2.drl
Inserting fact: 3
Inserting fact: 1
Inserting fact: 4
Inserting fact: 1
Inserting fact: 5
Number found with value: 5
Number found with value: 1
Number found with value: 4
Number found with value: 1
Number found with value: 3
```

here are probably a hundred and one better ways to sort numbers; but we will need to apply some cashflows in date order when we start looking at banking rules so let's look at a simple rule based example.

### Example 8.35. Banking Tutorial : Java Example3

```
public class Example3 {
    public static void main(String[] args) throws Exception {
        Number[] numbers = new Number[] {wrap(3), wrap(1), wrap(4), wrap(1),
 wrap(5)};
        new RuleRunner().runRules( new String[] { "Example3.drl" },
                                   numbers );
    }

    private static Integer wrap(int i) {
        return new Integer(i);
    }
}
```

Again we insert our Integers as before, this time the rule is slightly different:

### Example 8.36. Banking Tutorial : Rule Example3

```
rule "Rule 03"
    when
        $number : Number( )
        not Number( intValue < $number.intValue )
    then
        System.out.println("Number found with value: " + $number.intValue()
 );
        retract( $number );
end
```

The first line of the rules identifies a Number and extracts the value. The second line ensures that there does not exist a smaller number than the one found. By executing this rule, we might expect to find only one number - the smallest in the set. However, the retraction of the number after it has been printed, means that the smallest number has been removed, revealing the next smallest number, and so on.

So, the output we generate is, notice the numbers are now sorted numerically.

### Example 8.37. Banking Tutorial : Output Example3

```
Loading file: Example3.drl
Inserting fact: 3
Inserting fact: 1
Inserting fact: 4
Inserting fact: 1
Inserting fact: 5
Number found with value: 1
Number found with value: 1
Number found with value: 3
Number found with value: 4
Number found with value: 5
```

Now we want to start moving towards our personal accounting rules. The first step is to create a Cashflow POJO.

### Example 8.38. Banking Tutoria : Class Cashflow

```
public class Cashflow {
    private Date   date;
    private double amount;

    public Cashflow() {
    }

    public Cashflow(Date date,
                    double amount) {
        this.date = date;
        this.amount = amount;
    }

    public Date getDate() {
        return date;
    }

    public void setDate(Date date) {
        this.date = date;
    }

    public double getAmount() {
        return amount;
    }

    public void setAmount(double amount) {
        this.amount = amount;
    }

    public String toString() {
        return "Cashflow[date=" + date + ",amount=" + amount + "]";
    }
}
```

The Cashflow has two simple attributes, a date and an amount. I have added a toString method to print it and overloaded the constructor to set the values. The Example4 java code inserts 5 Cashflow objecst with varying dates and amounts.

### Example 8.39. Banking Tutorial : Java Example4

```
public class Example4 {
    public static void main(String[] args) throws Exception {
        Object[] cashflows = {
            new Cashflow(new SimpleDate("01/01/2007"), 300.00),
            new Cashflow(new SimpleDate("05/01/2007"), 100.00),
            new Cashflow(new SimpleDate("11/01/2007"), 500.00),
            new Cashflow(new SimpleDate("07/01/2007"), 800.00),
            new Cashflow(new SimpleDate("02/01/2007"), 400.00),
        };

        new RuleRunner().runRules( new String[] { "Example4.drl" },
                                   cashflows );
    }
}
```

SimpleDate is a simple class that extends Date and takes a String as input. It allows for pre-formatted Data classes, for convienience. The code is listed below

### Example 8.40. Banking Tutorial : Java SimpleDate

```
public class SimpleDate extends Date {
    private static final SimpleDateFormat format = new
 SimpleDateFormat("dd/MM/yyyy");

    public SimpleDate(String datestr) throws Exception {
        setTime(format.parse(datestr).getTime());
    }
}
```

Now, let’s look at rule04.drl to see how we print the sorted Cashflows:

### Example 8.41. Banking Tutorial : Rule Example4

```
rule "Rule 04"
    when
        $cashflow : Cashflow( $date : date, $amount : amount )
        not Cashflow( date < $date)
    then
        System.out.println("Cashflow: "+$date+" :: "+$amount);
        retract($cashflow);
end
```

Here, we identify a Cashflow and extract the date and the amount. In the second line of the rules we ensure that there is not a Cashflow with an earlier date than the one found. In the consequences, we print the Cashflow that satisfies the rules and then retract it, making way for the next earliest Cashflow. So, the output we generate is:

### Example 8.42. Banking Tutorial : Output Example4

```
Loading file: Example4.drl
```

```
Inserting fact: Cashflow[date=Mon Jan 01 00:00:00 GMT 2007,amount=300.0]
Inserting fact: Cashflow[date=Fri Jan 05 00:00:00 GMT 2007,amount=100.0]
Inserting fact: Cashflow[date=Thu Jan 11 00:00:00 GMT 2007,amount=500.0]
Inserting fact: Cashflow[date=Sun Jan 07 00:00:00 GMT 2007,amount=800.0]
Inserting fact: Cashflow[date=Tue Jan 02 00:00:00 GMT 2007,amount=400.0]
Cashflow: Mon Jan 01 00:00:00 GMT 2007 :: 300.0
Cashflow: Tue Jan 02 00:00:00 GMT 2007 :: 400.0
Cashflow: Fri Jan 05 00:00:00 GMT 2007 :: 100.0
Cashflow: Sun Jan 07 00:00:00 GMT 2007 :: 800.0
Cashflow: Thu Jan 11 00:00:00 GMT 2007 :: 500.0
```

Here we extend our Cashflow to give a TypedCashflow which can be CREDIT or DEBIT. Ideally, we would just add this to the Cashflow type, but so that we can keep all the examples simple, we will go with the extensions.

## Example 8.43. Banking Tutoria : Class TypedCashflow

```
public class TypedCashflow extends Cashflow {
    public static final int CREDIT = 0;
    public static final int DEBIT  = 1;

    private int             type;

    public TypedCashflow() {
    }

    public TypedCashflow(Date date,
                         int type,
                         double amount) {
        super( date,
               amount );
        this.type = type;
    }

    public int getType() {
        return type;
    }

    public void setType(int type) {
        this.type = type;
    }

    public String toString() {
        return "TypedCashflow[date=" + getDate() + ",type=" + (type ==
 CREDIT ? "Credit" : "Debit") + ",amount=" + getAmount() + "]";
    }
}
```

There are lots of ways to improve this code, but for the sake of the example this will do.

Nows lets create the Example5 runner.

### Example 8.44. Banking Tutorial : Java Example5

```
public class Example5 {
    public static void main(String[] args) throws Exception {
        Object[] cashflows = {
            new TypedCashflow(new SimpleDate("01/01/2007"),
                              TypedCashflow.CREDIT, 300.00),
            new TypedCashflow(new SimpleDate("05/01/2007"),
                              TypedCashflow.CREDIT, 100.00),
            new TypedCashflow(new SimpleDate("11/01/2007"),
                              TypedCashflow.CREDIT, 500.00),
            new TypedCashflow(new SimpleDate("07/01/2007"),
                              TypedCashflow.DEBIT, 800.00),
            new TypedCashflow(new SimpleDate("02/01/2007"),
                              TypedCashflow.DEBIT, 400.00),
        };

        new RuleRunner().runRules( new String[] { "Example5.drl" },
                                   cashflows );
    }
}
```

Here, we simply create a set of Cashflows which are either CREDIT or DEBIT Cashflows and supply them and rule05.drl to the RuleEngine.

Now, let’s look at rule0 Example5.drl to see how we print the sorted Cashflows:

### Example 8.45. Banking Tutorial : Rule Example5

```
rule "Rule 05"
    when
        $cashflow : TypedCashflow( $date : date,
                                   $amount : amount,
                                   type == TypedCashflow.CREDIT )
        not TypedCashflow( date < $date,
                           type == TypedCashflow.CREDIT )
    then
        System.out.println("Credit: "+$date+" :: "+$amount);
        retract($cashflow);
end
```

Here, we identify a Cashflow with a type of CREDIT and extract the date and the amount. In the second line of the rules we ensure that there is not a Cashflow of type CREDIT with an earlier date than the one found. In the consequences, we print the Cashflow that satisfies the rules and then retract it, making way for the next earliest Cashflow of type CREDIT.

So, the output we generate is

### Example 8.46. Banking Tutorial : Output Example5

```
Loading file: Example5.drl
Inserting fact: TypedCashflow[date=Mon Jan 01 00:00:00 GMT
 2007,type=Credit,amount=300.0]
Inserting fact: TypedCashflow[date=Fri Jan 05 00:00:00 GMT
 2007,type=Credit,amount=100.0]
Inserting fact: TypedCashflow[date=Thu Jan 11 00:00:00 GMT
 2007,type=Credit,amount=500.0]
Inserting fact: TypedCashflow[date=Sun Jan 07 00:00:00 GMT
 2007,type=Debit,amount=800.0]
Inserting fact: TypedCashflow[date=Tue Jan 02 00:00:00 GMT
 2007,type=Debit,amount=400.0]
Credit: Mon Jan 01 00:00:00 GMT 2007 :: 300.0
Credit: Fri Jan 05 00:00:00 GMT 2007 :: 100.0
Credit: Thu Jan 11 00:00:00 GMT 2007 :: 500.0
```

Here we are going to process both CREDITs and DEBITs on 2 bank accounts to calculate the account balance. In order to do this, I am going to create two separate Account Objects and inject them into the Cashflows before passing them to the Rule Engine. The reason for this is to provide easy access to the correct Bank Accounts without having to resort to Helper classes. Let's take a look at the Account class first. This is a simple POJO with an account number and balance:

### Example 8.47. Banking Tutoria : Class Account

```
public class Account {
    private long   accountNo;
    private double balance = 0;

    public Account() {
    }

    public Account(long accountNo) {
        this.accountNo = accountNo;
    }

    public long getAccountNo() {
        return accountNo;
    }

    public void setAccountNo(long accountNo) {
        this.accountNo = accountNo;
    }

    public double getBalance() {
        return balance;
    }
    public void setBalance(double balance) {
        this.balance = balance;
    }

    public String toString() {
        return "Account[" + "accountNo=" + accountNo + ",balance=" + balance
 + "]";
    }
}
```

Now let’s extend our TypedCashflow to give AllocatedCashflow (allocated to an account).

### Example 8.48. Banking Tutoria : Class AllocatedCashflow

```
public class AllocatedCashflow extends TypedCashflow {
    private Account account;

    public AllocatedCashflow() {
    }

    public AllocatedCashflow(Account account,
                             Date date,
                             int type,
                             double amount) {
        super( date,
               type,
               amount );
        this.account = account;
    }

    public Account getAccount() {
        return account;
    }

    public void setAccount(Account account) {
        this.account = account;
    }

    public String toString() {
        return "AllocatedCashflow[" + "account=" + account + ",date=" +
 getDate() +
                                  ",type=" + (getType() == CREDIT ? "Credit"
 : "Debit") +
                                  ",amount=" + getAmount() + "]";
    }
}
```

Now, let’s java code for Example5 execution. Here we create two Account objects and inject one into each cashflow as appropriate. For simplicity I have simply included them in the constructor.

### Example 8.49. Banking Tutorial : Java Example5

```
public class Example6 {
    public static void main(String[] args) throws Exception {
        Account acc1 = new Account(1);
        Account acc2 = new Account(2);

        Object[] cashflows = {
            new AllocatedCashflow(acc1,new SimpleDate("01/01/2007"),
                                  TypedCashflow.CREDIT, 300.00),
            new AllocatedCashflow(acc1,new SimpleDate("05/02/2007"),
                                  TypedCashflow.CREDIT, 100.00),
            new AllocatedCashflow(acc2,new SimpleDate("11/03/2007"),
                                  TypedCashflow.CREDIT, 500.00),
            new AllocatedCashflow(acc1,new SimpleDate("07/02/2007"),
                                  TypedCashflow.DEBIT,  800.00),
            new AllocatedCashflow(acc2,new SimpleDate("02/03/2007"),
                                  TypedCashflow.DEBIT,  400.00),
            new AllocatedCashflow(acc1,new SimpleDate("01/04/2007"),
                                  TypedCashflow.CREDIT, 200.00),
            new AllocatedCashflow(acc1,new SimpleDate("05/04/2007"),
                                  TypedCashflow.CREDIT, 300.00),
            new AllocatedCashflow(acc2,new SimpleDate("11/05/2007"),
                                  TypedCashflow.CREDIT, 700.00),
            new AllocatedCashflow(acc1,new SimpleDate("07/05/2007"),
                                  TypedCashflow.DEBIT,  900.00),
            new AllocatedCashflow(acc2,new SimpleDate("02/05/2007"),
                                  TypedCashflow.DEBIT,  100.00)
        };

        new RuleRunner().runRules( new String[] { "Example6.drl" },
                                   cashflows );
    }
}
```

Now, let’s look at rule Example06.drl to see how we apply each cashflow in date order and calculate and print the balance.

### Example 8.50. Banking Tutorial : Rule Example6

```
rule "Rule 06 - Credit"
    when
        $cashflow : AllocatedCashflow( $account : account,
                                       $date : date, $amount : amount,
                                        type==TypedCashflow.CREDIT )
        not AllocatedCashflow( account == $account, date < $date)
    then
        System.out.println("Credit: " + $date + " :: " + $amount);
        $account.setBalance($account.getBalance()+$amount);
        System.out.println("Account: " + $account.getAccountNo() +
                           " - new balance: " + $account.getBalance());

        retract($cashflow);
end

rule "Rule 06 - Debit"
    when
        $cashflow : AllocatedCashflow( $account : account,
                            $date : date, $amount : amount,
                            type==TypedCashflow.DEBIT )
        not AllocatedCashflow( account == $account, date < $date)
    then
        System.out.println("Debit: " + $date + " :: " + $amount);
        $account.setBalance($account.getBalance() - $amount);
        System.out.println("Account: " + $account.getAccountNo() +
                           " - new balance: " + $account.getBalance());

        retract($cashflow);
end
```

Here, we have separate rules for CREDITs and DEBITs, however we do not specify a type when checking for earlier cashflows. This is so that all cashflows are applied in date order regardless of which type of cashflow type they are. In the rule section we identify the correct account to work with and in the consequences we update it with the cashflow amount.

### Example 8.51. Banking Tutorial : Output Example6

```
Loading file: Example6.drl
Inserting fact:
 AllocatedCashflow[account=Account[accountNo=1,balance=0.0],date=Mon Jan 01
 00:00:00 GMT 2007,type=Credit,amount=300.0]
Inserting fact:
 AllocatedCashflow[account=Account[accountNo=1,balance=0.0],date=Mon Feb 05
 00:00:00 GMT 2007,type=Credit,amount=100.0]
Inserting fact:
 AllocatedCashflow[account=Account[accountNo=2,balance=0.0],date=Sun Mar 11
 00:00:00 GMT 2007,type=Credit,amount=500.0]
Inserting fact:
 AllocatedCashflow[account=Account[accountNo=1,balance=0.0],date=Wed Feb 07
 00:00:00 GMT 2007,type=Debit,amount=800.0]
Inserting fact:
 AllocatedCashflow[account=Account[accountNo=2,balance=0.0],date=Fri Mar 02
 00:00:00 GMT 2007,type=Debit,amount=400.0]
Inserting fact:
 AllocatedCashflow[account=Account[accountNo=1,balance=0.0],date=Sun Apr 01
 00:00:00 BST 2007,type=Credit,amount=200.0]
```

```
Inserting fact:
 AllocatedCashflow[account=Account[accountNo=1,balance=0.0],date=Thu Apr 05
 00:00:00 BST 2007,type=Credit,amount=300.0]
Inserting fact:
 AllocatedCashflow[account=Account[accountNo=2,balance=0.0],date=Fri May 11
 00:00:00 BST 2007,type=Credit,amount=700.0]
Inserting fact:
 AllocatedCashflow[account=Account[accountNo=1,balance=0.0],date=Mon May 07
 00:00:00 BST 2007,type=Debit,amount=900.0]
Inserting fact:
 AllocatedCashflow[account=Account[accountNo=2,balance=0.0],date=Wed May 02
 00:00:00 BST 2007,type=Debit,amount=100.0]
Debit: Fri Mar 02 00:00:00 GMT 2007 :: 400.0
Account: 2 - new balance: -400.0
Credit: Sun Mar 11 00:00:00 GMT 2007 :: 500.0
Account: 2 - new balance: 100.0
Debit: Wed May 02 00:00:00 BST 2007 :: 100.0
Account: 2 - new balance: 0.0
Credit: Fri May 11 00:00:00 BST 2007 :: 700.0
Account: 2 - new balance: 700.0
Credit: Mon Jan 01 00:00:00 GMT 2007 :: 300.0
Account: 1 - new balance: 300.0
Credit: Mon Feb 05 00:00:00 GMT 2007 :: 100.0
Account: 1 - new balance: 400.0
Debit: Wed Feb 07 00:00:00 GMT 2007 :: 800.0
Account: 1 - new balance: -400.0
Credit: Sun Apr 01 00:00:00 BST 2007 :: 200.0
Account: 1 - new balance: -200.0
Credit: Thu Apr 05 00:00:00 BST 2007 :: 300.0
Account: 1 - new balance: 100.0
Debit: Mon May 07 00:00:00 BST 2007 :: 900.0
Account: 1 - new balance: -800.0
```

## 8.6. Pricing Rule Decision Table Example

The Pricing Rule decision table demonstrates the use of a decision table in a spreadsheet (XLS format) in calculating the retail cost of an insurance policy. The purpose of the set of rules provided is to calculate a base price, and an additional discount for a car driver applying for a specific policy. The drivers age, history and the policy type all contribute to what the basic premium is, and an additional chunk of rules deals with refining this with a subtractive percentage discount.

```
Name: Example Policy Pricing
Main class: org.drools.examples.PricingRuleDTExample
Type: java application
Rules file: ExamplePolicyPricing.xls
Objective: demonstrate spreadsheet based decision tables.
```

### 8.6.1. Executing the example

Open the PricingRuleDTExample.java and execute it as a Java application. It should produce the following console output:

```
Cheapest possible
BASE PRICE IS: 120
DISCOUNT IS: 20
```

The code to the execute the example is very similar to the other examples. The rules are loaded, the facts inserted and a stateless session is used. What is different is how the rules are added:

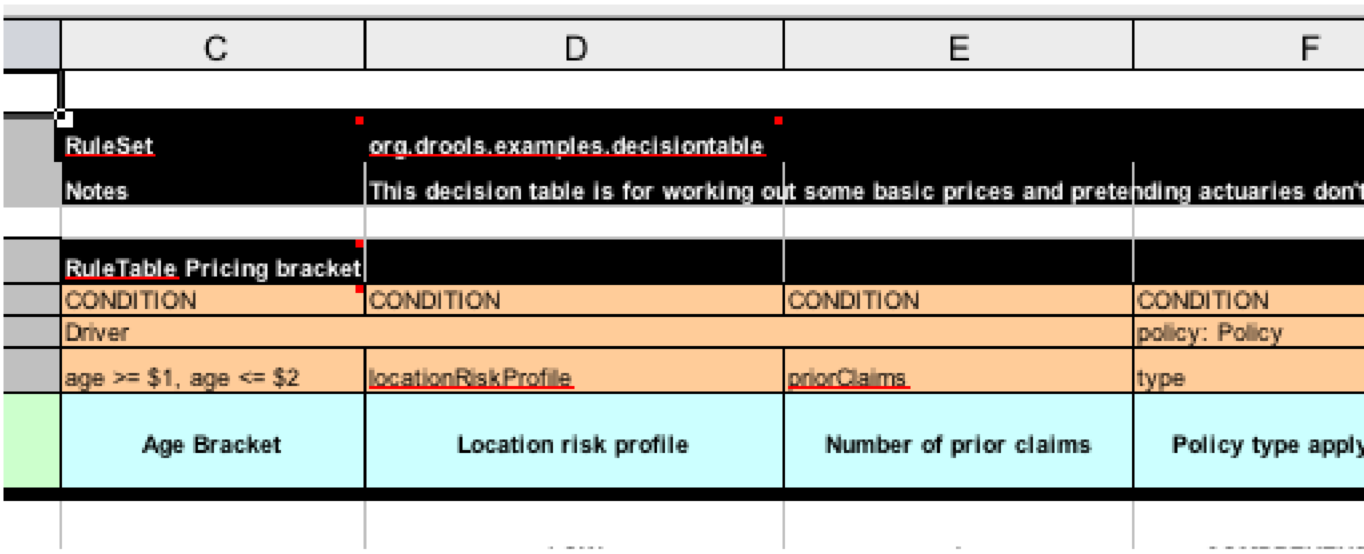
```
DecisionTableConfiguration dtableconfiguration =
 KnowledgeBuilderFactory.newDecisionTableConfiguration();
        dtableconfiguration.setInputType( DecisionTableInputType.XLS );

        KnowledgeBuilder kbuilder =
 KnowledgeBuilderFactory.newKnowledgeBuilder();

        kbuilder.add( ResourceFactory.newClassPathResource(
 "ExamplePolicyPricing.xls", getClass() ),
                              ResourceType.DTABLE,
                              dtableconfiguration );
```

Note the use of the DecisionTableConfiguration class. It is what takes the DecisionTableInputType.XLS as input type. If you use the BRMS, all this is of course taken care of for you.

There are 2 facts used in this example, Driver, and Policy. Both are used with their default values. The Driver is 30 years old, has had no prior claims and currently has a risk profile of LOW. The Policy being applied for is COMPREHENSIVE, and the policy has not yet been approved.

### 8.6.2. The decision table

In this decision table, each row is a rule, and each column is a condition or an action.



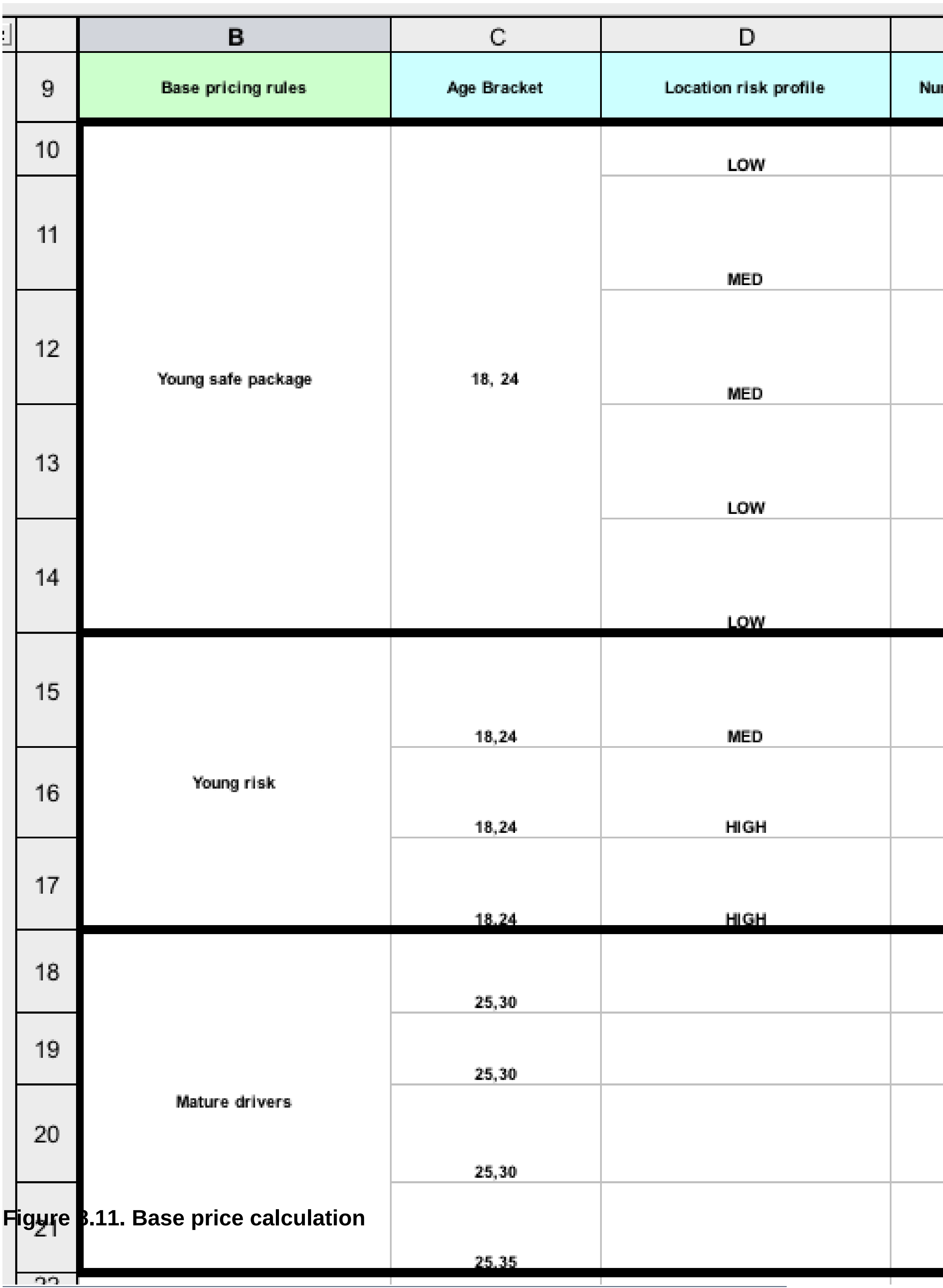
|  | C | D | E | F |
|---|---|---|---|---|
|  | RuleSet | org.drools.examples.decisiontable |  |  |
|  | Notes | This decision table is for working out some basic prices and pretending actuaries don' |  |  |
|  |  |  |  |  |
|  | RuleTable Pricing bracket |  |  |  |
|  | CONDITION | CONDITION | CONDITION | CONDITION |
|  | Driver |  |  | policy: Policy |
|  | age >= $1, age <= $2 | locationRiskProfile | priorClaims | type |
|  | Age Bracket | Location risk profile | Number of prior claims | Policy type apply |

## Figure 8.10. Decision table configuration

Referring to the above, we have the RuleSet declaration, which provides the package name. There are also other optional items you can have here, such as Variables for global variables, and Imports for importing classes. In this case, the namespace of the rules is the same as the fact classes we are using, so we can omit it.

Moving further down, we can see the RuleTable declaration. The name after this (Pricing bracket) is used as the prefix for all the generated rules. Below that, we have CONDITION or ACTION - this indicates the purpose of the column (ie does it form part of the condition, or an action of a rule that will be generated).

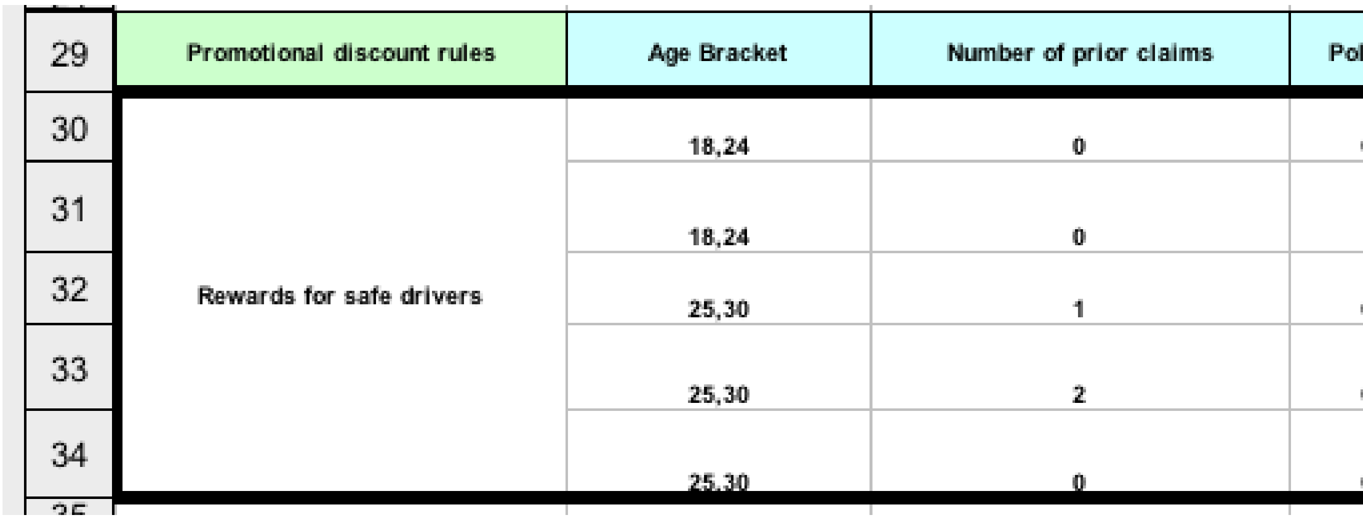
You can see there is a Driver which is spanned across 3 cells, this means the template expressions below it apply to that fact. So we look at the drivers age range (which uses $1 and $2 with comma separated values), locationRiskProfile, and priorClaims in the respective columns. In the action columns, we are setting the policy base price, and then logging a message.



| | B | C | D | |
|---|---|---|---|---|
| 9 | Base pricing rules | Age Bracket | Location risk profile | Nu |
| 10 | Young safe package | 18, 24 | LOW | |
| 11 | | | MED | |
| 12 | | | MED | |
| 13 | | | LOW | |
| 14 | | | LOW | |
| 15 | Young risk | 18,24 | MED | |
| 16 | | 18,24 | HIGH | |
| 17 | | 18,24 | HIGH | |
| 18 | Mature drivers | 25,30 | | |
| 19 | | 25,30 | | |
| 20 | | 25,30 | | |
| 21 | | 25,35 | | |

**Figure 3.11. Base price calculation**

Referring to the above, we can see there are broad category brackets (indicated by the comment in the left most column). As we know the details of our driver and their policy, we can tell (with a bit of thought) that they should match row number 18, as they have no prior accidents, and are 30 years old. This gives us a base price of 120.



| | | | |
|---|---|---|---|
| 29 | Promotional discount rules | Age Bracket | Number of prior claims |
| 30 | Rewards for safe drivers | 18,24 | 0 |
| 31 | | 18,24 | 0 |
| 32 | | 25,30 | 1 |
| 33 | | 25,30 | 2 |
| 34 | | 25,30 | 0 |

Figure 8.12. Discount calculation

Referring to the above, we are seeing if there is any discount we can give our driver. Based on the Age bracket, number of priot claims, and the policy type, a discount is provided. In our case, the drive is 3, with no priors, and they are applying for COMPREHENSIVE, this means we can give a discount of 20%. Note that this is actually a separate table, but in the same worksheet. This different templates apply.

It is important to note that decision tables generate rules, this means they aren't simply top down logic, but more a means to capture data that generate rules (this is a subtle difference that confuses some people). The evaluation of the rules is not "top down" necessarily, all the normal indexing and mechanics of the rule engine still apply.

## 8.7. Pet Store Example

```
Name: Pet Store
Main class: org.drools.examples.PetStore
Type: Java application
Rules file: PetStore.drl
Objective: Demonstrate use of Agenda Groups, Global Variables and
 integration with a GUI (including callbacks from within the Rules)
```

The Pet Store example shows how to integrate Rules with a GUI (in this case a Swing based Desktop application). Within the rules file, it shows how to use agenda groups and auto-focus to control which of a set of rules is allowed to fire at any given time. It also shows mixing of Java and MVEL dialects within the rules, the use of accumulate functions and calling of Java functions from within the ruleset.

Like the rest of the the samples, all the Java Code is contained in one file. The PetStore.java contains the following principal classes (in addition to several minor classes to handle Swing Events)

- *Petstore* - containing the main() method that we will look at shortly.
- *PetStoreUI* - responsible for creating and displaying the Swing based GUI. It contains several smaller classes , mainly for responding to various GUI events such as mouse and button clicks.
- *TabelModel* - for holding the table data. Think of it as a JavaBean that extends the Swing AbstractTableModel class.
- *CheckoutCallback* - Allows the GUI to interact with the Rules.
- *Ordershow* - the items that we wish to buy.
- *Purchase* - Details of the order and the products we are buying.
- *Product* - JavaBean holding details of the product available for purchase, and it's price.

Much of the Java code is either JavaBeans (simple enough to understand) or Swing based. We will touch on some Swing related points in the this tutorial , but a good place to get more Swing component information is `http://java.sun.com/docs/books/tutorial/uiswing/` *available at the Sun Swing website.[]* [???]

There are two important Rules related pieces of Java code in *Petstore.java*.

### Example 8.52. Creating the PetStore RuleBase - extract from PetStore.java main() method

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

            kbuilder.add( ResourceFactory.newClassPathResource(
 "PetStore.drl",

 PetStore.class ),
                                  ResourceType.DRL );
            KnowledgeBase kbase = KnowledgeBaseFactory.newKnowledgeBase();
            kbase.addKnowledgePackages( kbuilder.getKnowledgePackages() );

            //RuleB
            Vector<Product> stock = new Vector<Product>();
            stock.add( new Product( "Gold Fish",
                                    5 ) );
            stock.add( new Product( "Fish Tank",
                                    25 ) );
            stock.add( new Product( "Fish Food",
                                    2 ) );
```

```
              //The callback is responsible for populating working memory and
              // fireing all rules
              PetStoreUI ui = new PetStoreUI( stock,
                                              new CheckoutCallback( kbase ) );
              ui.createAndShowGUI();
```

This code above loads the rules (drl) file from the classpath. Unlike other examples where the facts are asserted and fired straight away, this example defers this step to later. The way it does this is via the second last line where the PetStoreUI is created using a constructor the passes in the Vector called stock containing products, and an instance of the CheckoutCallback class containing the RuleBase that we have just loaded.

The actual Javacode that fires the rules is within the *CheckoutCallBack.checkout()* method. This is triggered (eventually) when the 'Checkout' button is pressed by the user.

**Example 8.53. Firing the Rules - extract from the CheckOutCallBack.checkout() method**

```
public String checkout(JFrame frame, List<Product> items) {
            Order order = new Order();

            //Iterate through list and add to cart
            for ( int i = 0; i < items.size(); i++ ) {
                order.addItem( new Purchase( order,
                                             (Product) items.get( i ) ) );
            }

            //add the JFrame to the ApplicationData to allow for user
 interaction

            StatefulKnowledgeSession ksession =
 kbase.newStatefulKnowledgeSession();
            ksession.setGlobal( "frame",
                                frame );
            ksession.setGlobal( "textArea",
                                this.output );

            ksession.insert( new Product( "Gold Fish",
                                          5 ) );
            ksession.insert( new Product( "Fish Tank",
                                          25 ) );
            ksession.insert( new Product( "Fish Food",
                                          2 ) );

            ksession.insert( new Product( "Fish Food Sample",
                                          0 ) );

            ksession.insert( order );
            ksession.fireAllRules();

            //returns the state of the cart
            return order.toString();
        }
```

Two items get passed into this method; A handle to the JFrame Swing Component surrounding the output text frame (bottom of the GUI if / when you run the component). The second item is a list of order items; this comes from the TableModel the stores the information from the 'Table' area at the top right section of the GUI.

The *for()* loop transforms the list of order items coming from the GUI into the Order JavaBean (also contained in the PetStore.java file). Note that it would be possible to refer to the Swing dataset directly within the rules, but it is better coding practice to do it this way (using Simple Java Objects). It means that we are not tied to Swing if we wanted to transform the sample into a Web application.

It is important to note that **all state in this example is stored in the Swing components, and that the rules are effectively stateless.** Each time the 'Checkout' button is pressed, this code copies the contents of the Swing *TableModel* into the Session / Working Memory.

Within this code, there are nine calls to the working memory. The first of these creates a new workingMemory (StatefulKnowledgeSession) from the Knowledgebase - remember that we passed in this Knowledgebase when we created the CheckoutCallBack class in the *main()* method. The next two calls pass in two objects that we will hold as Global variables in the rules - the Swing text area and Swing frame that we will use for writing messages later.

More inserts put information on products into the working memory, as well as the order list. The final call is the standard *fireAllRules()*. Next, we look at what this method causes to happen within the Rules file.

### Example 8.54. Package, Imports , Globals and Dialect - extract (1) from PetStore.drl

```
package org.drools.examples

import org.drools.WorkingMemory
import org.drools.examples.PetStore.Order
import org.drools.examples.PetStore.Purchase
import org.drools.examples.PetStore.Product
import java.util.ArrayList
import javax.swing.JOptionPane;

import javax.swing.JFrame

global JFrame frame
global javax.swing.JTextArea textArea
```

The first part of the *PetStore.drl* file contains the standard package and import statement to make various Java classes available to the rules. What is new are the two globals *frame and textArea.* These hold references to the Swing JFrame and Textarea components that were previous passed by the Java code calling the *setGlobal()* method. Unlike normal variables in Rules , which expire as soon as the rule has fired, Global variables retain their value for the lifetime of the (Stateful in this case) Session.

The next extract (below) is from the **end** of the PetStore.drl file. It contains two functions that are referenced by the rules that we will look at shortly.

### Example 8.55. Java Functions in the Rules - extract (2) from PetStore.drl

```
function void doCheckout(JFrame frame, WorkingMemory workingMemory) {
    Object[] options = {"Yes",
                        "No"};

    int n = JOptionPane.showOptionDialog(frame,
                                         "Would you like to checkout?",
                                         "",
                                         JOptionPane.YES_NO_OPTION,
                                         JOptionPane.QUESTION_MESSAGE,
                                         null,
                                         options,
                                         options[0]);

    if (n == 0) {
        workingMemory.setFocus( "checkout" );
    }
}

function boolean requireTank(JFrame frame, WorkingMemory workingMemory,
 Order order, Product fishTank, int total) {
    Object[] options = {"Yes",
                        "No"};

    int n = JOptionPane.showOptionDialog(frame,
                                         "Would you like to buy a tank for
 your " + total + " fish?",
                                         "Purchase Suggestion",
                                         JOptionPane.YES_NO_OPTION,
                                         JOptionPane.QUESTION_MESSAGE,
                                         null,
                                         options,
                                         options[0]);

    System.out.print( "SUGGESTION: Would you like to buy a tank for your "
                      + total + " fish? - " );

    if (n == 0) {
        Purchase purchase = new Purchase( order, fishTank );
        workingMemory.insert( purchase );
        order.addItem( purchase );
        System.out.println( "Yes" );
    } else {
        System.out.println( "No" );
    }
    return true;
}
```

Having these functions in the rules file makes the PetStore sample more compact - in real life you probably have the functions in a file of their own (within the same rules package), or as a static method on a standard Java class (and import them using the **import function my.package.Foo.hello** syntax).

The above functions are

- *doCheckout()* - Displays a dialog asking the user if they wish to checkout. If they do, focus is set to the *checkOut* agenda-group, allowing rules in that group to (potentially) fire.
- *requireTank()* - Displays a dialog asking the user if they wish to buy a tank. If so, a new FishTank *Product* added to the orderlist in working memory.

We'll see later the rules that call these functions.The next set of examples are from the PetStore rules themselves. The first extract is the one that happens to fire first (partly because it has the *auto-focus* attibute set to true).

### Example 8.56. Putting each (individual) item into working memory - extract (3) from PetStore.drl

```
// insert each item in the shopping cart into the Working Memory
// insert each item in the shopping cart into the Working Memory
rule "Explode Cart"
    agenda-group "init"
 auto-focus true
    salience 10
 dialect "java"
 when
     $order : Order( grossTotal == -1 )
  $item : Purchase() from $order.items
 then
  insert( $item );
  drools.getKnowledgeRuntime().getAgenda().getAgendaGroup( "show items"
 ).setFocus();
  drools.getKnowledgeRuntime().getAgenda().getAgendaGroup( "evaluate"
 ).setFocus();
end
```

This rule matches against all orders that do not yet have an Order.grossTotal calculated . It loops for each purchase item in that order. Some of the *Explode Cart* Rule should be familiar ; the rule name, the salience (suggesting of the order that the rules should be fired in) and the dialect set to *java*. There are three new items:

- **agenda-group "init"** - the name of the agenda group. In this case, there is only one rule in the group. However, nothing in Java code / nor a rule sets the focus to this group , so it relies on the next attibute for it's chance to fire.
- **auto-focus true -** This is the only rule in the sample, so when *fireAllRules()* is called from within the Java code, this rule is the first to get a chance to fire.
- **drools.setFocus()** This sets the focus to the *show items* and *evaluate* agenda groups in turn , giving their rules a chance to fire. In practice , we loop through all items on the order, inserting them into memory, then firing the other rules after each insert.

The next two listings shows the rules within the *show items* and *evaluate* agenda groups. We look at them in the order that they are called.

## Example 8.57. Show Items in the GUI extract (4) from PetStore.drl

```
rule "Show Items"
    agenda-group "show items"
    dialect "mvel"
when
    $order : Order( )
    $p : Purchase( order == $order )
then
   textArea.append( $p.product + "\n");
end
```

The *show items* agenda-group has only one rule, also called *Show Items* (note the difference in case). For each purchase on the order currently in the working memory (session) it logs details to the text area (at the bottom of the GUI). The *textArea* variable used to do this is one of the Global Variables we looked at earlier.

The *evaluate* Agenda group also gains focus from the *explode cart* rule above. This Agenda group has two rules (below) *Free Fish Food Sample* and *Suggest Tank*.

## Example 8.58. Evaluate Agenda Group extract (5) from PetStore.drl

```
// Free Fish Food sample when we buy a Gold Fish if we haven't already
 bought
// Fish Food and dont already have a Fish Food Sample
rule "Free Fish Food Sample"
    agenda-group "evaluate"
    dialect "mvel"
when
```

```
    $order : Order()
    not ( $p : Product( name == "Fish Food") &amp;&amp; Purchase( product ==
 $p ) )
    not ( $p : Product( name == "Fish Food Sample") &amp;&amp; Purchase(
 product == $p ) )
    exists ( $p : Product( name == "Gold Fish") &amp;&amp; Purchase( product
 == $p ) )
    $fishFoodSample : Product( name == "Fish Food Sample" );
then
    System.out.println( "Adding free Fish Food Sample to cart" );
    purchase = new Purchase($order, $fishFoodSample);
    insert( purchase );
    $order.addItem( purchase );
end

// Suggest a tank if we have bought more than 5 gold fish and dont already
 have one
rule "Suggest Tank"
    agenda-group "evaluate"
    dialect "java"
when
    $order : Order()
    not ( $p : Product( name == "Fish Tank") &amp;&amp; Purchase( product ==
 $p ) )
    ArrayList( $total : size &gt; 5 ) from collect( Purchase( product.name
 == "Gold Fish" ) )
    $fishTank : Product( name == "Fish Tank" )
then
    requireTank(frame, drools.getWorkingMemory(), $order, $fishTank,
 $total);
end
```

The *Free Fish Food Sample* rule will only fire if

- We *don't* already have any fish food.
- We *don't* already have a free fish food sample.
- We *do* have a Gold Fish in our order.

If the rule does fire, it creates a new product (Fish Food Sample), and adds it to the Order in working memory.

The *Suggest Tank* rule will only fire if

- We *don't* already have a Fish Tank in our order
- If we *can* find more than 5 Gold Fish Products in our order.

If the rule does fire, it calls the *requireTank*() function that we looked at earlier (showing a Dialog to the user, and adding a Tank to the order / working memory if confirmed). When calling the *requireTank*() function the rule passes the global *frame* variable so that the function has a handle to the Swing GUI.

The next rule we look at is *do checkout.*

### Example 8.59. Doing the Checkout - extract (6) from PetStore.drl

```
rule "do checkout"
    dialect "java"
    when
    then
        doCheckout(frame, drools.getWorkingMemory());
end
```

The *do checkout* rule has **no agenda-group set and no auto-focus attribute**. As such, is is deemed part of the default (MAIN) agenda-group - the same as the other non PetStore examples where agenda groups are not used. This group gets focus by default when all the rules/agenda-groups that explicity had focus set to them have run their course.

There is no LHS to the rule, so the RHS will always call the *doCheckout*() function. When calling the *doCheckout*() function the rule passes the global *frame* variable so the function has a handle to the Swing GUI. As we saw earlier, the *doCheckout*() function shows a confirmation dialog to the user. If confirmed, the function sets the focus to the *checkout* agenda-group, allowing the next lot of rules to fire.

### Example 8.60. Checkout Rules- extract (7) from PetStore.drl

```
rule "Gross Total"
    agenda-group "checkout"
    dialect "mvel"
when
   $order : Order( grossTotal == -1)
   Number( total : doubleValue ) from accumulate( Purchase( $price :
 product.price ),
                 sum( $price ) )
then
    modify( $order ) { grossTotal = total };
    textArea.append( "\ngross total=" + total + "\n" );
end

rule "Apply 5% Discount"
    agenda-group "checkout"
dialect "mvel"
when
   $order : Order( grossTotal &gt;= 10 &amp;&amp; &lt; 20 )
then
   $order.discountedTotal = $order.grossTotal * 0.95;
   textArea.append( "discountedTotal total=" + $order.discountedTotal + "\n"
 );
```

```
end


rule "Apply 10% Discount"
    agenda-group "checkout"
dialect "mvel"
when
   $order : Order( grossTotal &gt;= 20 )
then
   $order.discountedTotal = $order.grossTotal * 0.90;
   textArea.append( "discountedTotal total=" + $order.discountedTotal + "\n"
  );
end
```

There are three rules in the *checkout* agenda-group

- **Gross Total** - if we haven't already calculated the gross total, accumulates the product prices into a total, puts this total into working memory, and displays it via the Swing TextArea (using the *textArea* global variable yet again).
- **Apply 5% Discount** - if our gross total is between 10 and 20, then calculate the discounted total and add it to working memory / display in the text area.
- **Apply 10% Discount** - if our gross total is equal to or greater than 20, calculate the discounted total and add it to working memory / display in the text area.

Now we've run through what happens in the code, lets have a look at what happens when we run the code for real. The *PetStore.java* example contains a *main()* method, so it can be run as a standard Java application (either from the command line or via the IDE). This assumes you have your classpath set correctly (see the start of the examples section for more information).

The first screen that we see is the Pet Store Demo. It has a List of available products (top left) , an empty list of selected products (top right), checkout and reset buttons (middle) and an empty system messages area (bottom).

## Figure 8.13. Figure 1 - PetStore Demo just after Launch

To get to this point, the following things have happened:

1. The *main()* method has run and loaded the RuleBase **but not yet fired the rules**. This is the only rules related code to run so far.
2. A new *PetStoreUI* class is created and given a handle to the RuleBase (for later use).
3. Various Swing Components do their stuff, and the above screen is shown and **waits for user input**.

Clicking on various products from the list might give you a screen similar to the one below.

### Figure 8.14. Figure 2 - PetStore Demo with Products Selected

Note that **no rules code has been fired here**. This is only swing code, listening for the mouse click event, and added the clicked product to the *TableModel* object for display in the top right hand section (as an aside , this is a classic use of the Model View Controller - MVC - design pattern).

It is only when we press the **Checkout** that we fire our business rules, in roughly the same order that we walked through the code earlier.

1. The *CheckOutCallBack.checkout()* method is called (eventually) by the Swing class waiting for the click on the checkout button. This inserts the data from the *TableModel* object (top right hand side of the GUI), and handles from the GUI into the session / working memory. It then fires the rules.
2. The *Explode Cart* rule is the first to fire, given that has *auto-focus* set to true. It loops through all the products in the cart, makes sure the products are in the working memory, then gives the *Show Items* and *Evaluation* agenda groups a chance to fire. The rules in these groups, add the contents of the cart to the text area (bottom), decide whether or not to give us free fish food and whether to ask if we want to buy a fish tank (Figure 3 below).

### Figure 8.15. Figure 3 - Do we want to buy a fish tank?

1. The *Do Checkout* rule is the next to fire as it (a) No other agenda group currently has focus and (b) it is part of the default (MAIN) agenda group. It always calls the *doCheckout() function* which displays a 'Would you like to Checkout?' Dialog Box.
2. The *doCheckout() function* sets the focus to the *checkout* agenda-group, giving the rules in that group the option to fire.
3. The rules in the the *checkout* agenda-group, display the contents of the cart and apply the appropriate discount.
4. **Swing then waits for user input** to either checkout more products (and to cause the rules to fire again) or to close the GUI - Figure 4 below.

### Figure 8.16. Figure 4 - Petstore Demo after all rules have fired.

Should we choose, we could add more System.out calls to demonstrate this flow of events. The current output of the console of the above sample is as per the listing below.

### Example 8.61. Console (System.out) from running the PetStore GUI

```
Adding free Fish Food Sample to cart
SUGGESTION: Would you like to buy a tank for your 6 fish? - Yes
```

## 8.8. Honest Politician Example

The honest politician example demonstrates truth maintenance with logical assertions, the basic premise is that an object can only exist while a statement is true. A rule's consequence can logical insert an object with the insertLogical method, this means the object will only remain in the working memory as long as the rule that logically inserted it remains true, when the rule is no longer true the object is automatically retracted.

In this example there is Politician class with a name and a boolean value for honest state, four politicians with honest state set to true are inserted.

### Example 8.62. Politician Class

```
public class Politician {
    private String name;
    private boolean honest;
    ...
}
```

### Example 8.63. Honest Politician Example Execution

```
Politician blair  = new Politician("blair", true);
Politician bush  = new Politician("bush", true);
Politician chirac  = new Politician("chirac", true);
Politician schroder   = new Politician("schroder", true);

ksession.insert( blair );
ksession.insert( bush );
ksession.insert( chirac );
ksession.insert( schroder );

ksession.fireAllRules();
```

The console out shows that while there is atleast one honest polician democracy lives, however as each politician is in turn corrupted by an evil corporation, when all politicians are dishonest democracy is dead.

### Example 8.64. Honest Politician Example Console Output

```
Hurrah!!! Democracy Lives
I'm an evil corporation and I have corrupted schroder
I'm an evil corporation and I have corrupted chirac
I'm an evil corporation and I have corrupted bush
I'm an evil corporation and I have corrupted blair
We are all Doomed!!! Democracy is Dead
```

As soon as there is one more more honest politcians in the working memory a new Hope object is logically asserted, this object will only exist while there is at least one or more honest politicians, the moment all politicians are dishonest then the Hope object will be automatically retracted. This rule is given a salience of 10 to make sure it fires before any other rules, as at this stage the "Hope is Dead" rule is actually true.

### Example 8.65. Honest Politician Example : Rule "We have an honest politician"

```
rule "We have an honest Politician"
    salience 10
    when
        exists( Politician( honest == true ) )
    then
        insertLogical( new Hope() );
end
```

As soon as a Hope object exists the "Hope Lives" rule matches, and fires, it has a salience of 10 so that it takes priority over "Corrupt the Honest".

### Example 8.66. Honest Politician Example : Rule "Hope Lives"

```
rule "Hope Lives"
    salience 10
    when
        exists( Hope() )
    then
        System.out.println("Hurrah!!! Democracy Lives");
end
```

Now that hope exists and we have, at the start, four honest politicians we have 4 activations for this rule all in conflict. This rule iterates over those rules firing each one in turn, corrupting each politician so that they are no longer honest. When all four politicians have been corrupted we have no politicians with the property "honest == true" thus the rule "We have an honest Politician" is no longer true and the object it logical inserts "new Hope()" is automatically retracted.

### Example 8.67. Honest Politician Example : Rule "Corrupt the Honest"

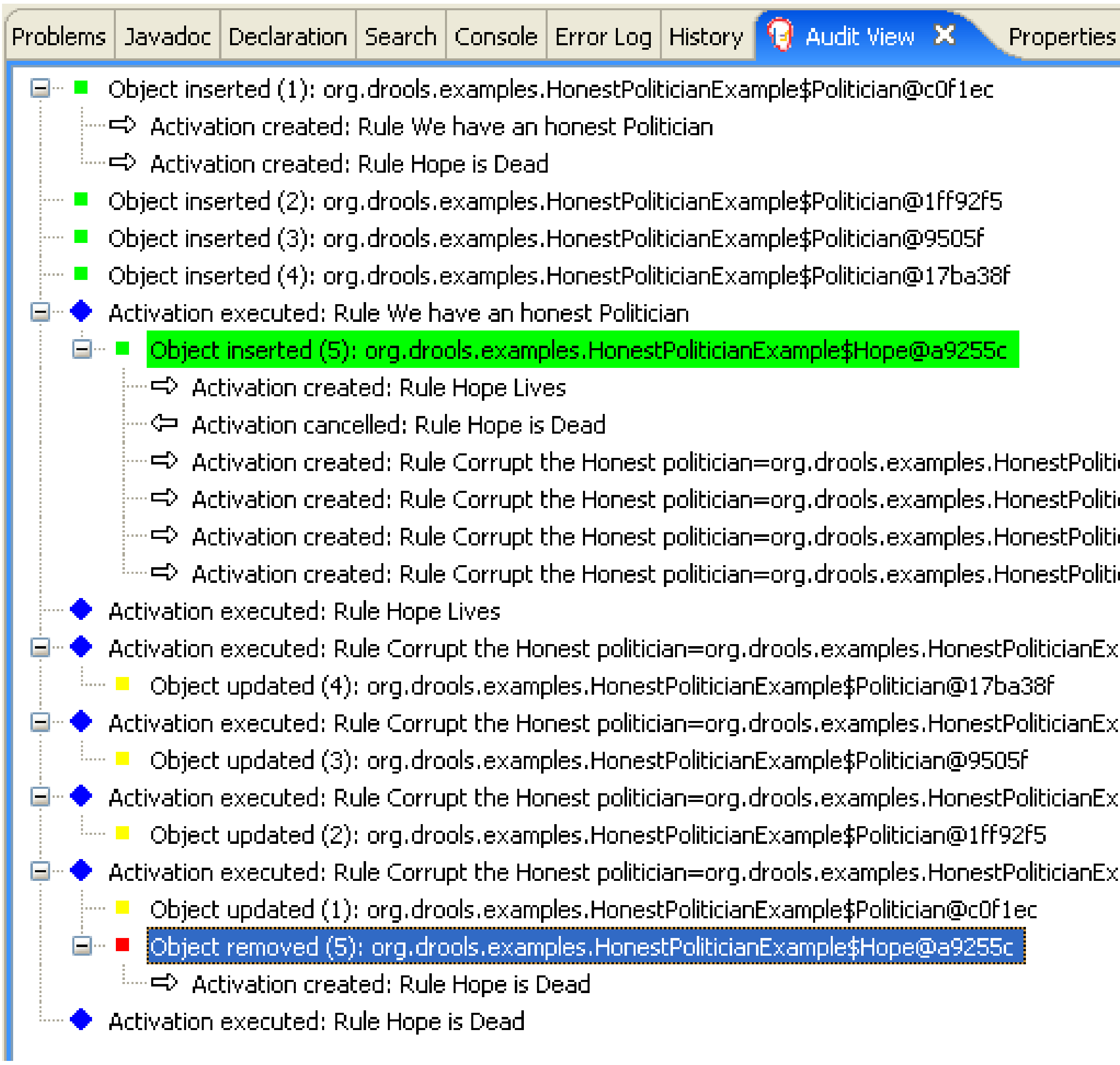
```
rule "Corrupt the Honest"
    when
        politician : Politician( honest == true )
        exists( Hope() )
    then
        System.out.println( "I'm an evil corporation and I have corrupted "
 + politician.getName() );
        modify ( politician ) { honest = false };
end
```

With Hope being automatically retracted, via the truth maintenance system, then Hope no longer exists in the system and this rule will match and fire.

### Example 8.68. Honest Politician Example : Rule "Hope is Dead"

```
rule "Hope is Dead"
    when
        not( Hope() )
    then
        System.out.println( "We are all Doomed!!! Democracy is Dead" );
end
```

lets take a look the audit trail for this application:


Problems | Javadoc | Declaration | Search | Console | Error Log | History | Audit View | Properties

```
Object inserted (1): org.drools.examples.HonestPoliticianExample$Politician@c0f1ec
    Activation created: Rule We have an honest Politician
    Activation created: Rule Hope is Dead
Object inserted (2): org.drools.examples.HonestPoliticianExample$Politician@1ff92f5
Object inserted (3): org.drools.examples.HonestPoliticianExample$Politician@9505f
Object inserted (4): org.drools.examples.HonestPoliticianExample$Politician@17ba38f
Activation executed: Rule We have an honest Politician
    Object inserted (5): org.drools.examples.HonestPoliticianExample$Hope@a9255c
        Activation created: Rule Hope Lives
        Activation cancelled: Rule Hope is Dead
        Activation created: Rule Corrupt the Honest politician=org.drools.examples.HonestPoliti
        Activation created: Rule Corrupt the Honest politician=org.drools.examples.HonestPoliti
        Activation created: Rule Corrupt the Honest politician=org.drools.examples.HonestPoliti
        Activation created: Rule Corrupt the Honest politician=org.drools.examples.HonestPoliti
Activation executed: Rule Hope Lives
Activation executed: Rule Corrupt the Honest politician=org.drools.examples.HonestPoliticianEx
    Object updated (4): org.drools.examples.HonestPoliticianExample$Politician@17ba38f
Activation executed: Rule Corrupt the Honest politician=org.drools.examples.HonestPoliticianEx
    Object updated (3): org.drools.examples.HonestPoliticianExample$Politician@9505f
Activation executed: Rule Corrupt the Honest politician=org.drools.examples.HonestPoliticianEx
    Object updated (2): org.drools.examples.HonestPoliticianExample$Politician@1ff92f5
Activation executed: Rule Corrupt the Honest politician=org.drools.examples.HonestPoliticianEx
    Object updated (1): org.drools.examples.HonestPoliticianExample$Politician@c0f1ec
    Object removed (5): org.drools.examples.HonestPoliticianExample$Hope@a9255c
        Activation created: Rule Hope is Dead
Activation executed: Rule Hope is Dead
```

**Figure 8.17. Honest Politician Example Audit View**

The moment we insert the first politician we have two activations, the "We have an honest Politician" is activated only once for the first inserted politician because it uses an existential 'exists' conditional element which only matches. the rule "Hope is Dead" is also activated at this stage, because as of yet we have not inserted the Hope object. "We have an honest Politician" fires first, as it has a higher salience over "Hope is Dead" which inserts the Hope object, that action is highlighted green above. The insertion of the Hope object activates "Hope Lives" and de-activates "Hope is Dead", it also actives "Corrupt the Honest" for each inserted honested politician. "Rule Hope Lives" executes printing "Hurrah!!! Democracy Lives". Then for each politician the rule "Corrupt the Honest" fires printing "I'm an evil corporation and I have corrupted X", where X is the name of the politician, and modifies the politicians honest value to false. When the last honest polician is corrupted Hope is automatically retracted, by the truth maintenance system, as shown by the blue highlighted area. The green highlighted area shows the origin of the currently selected blue highlighted area. Once Hope is retracted "Hope is dead" activates and fires printing "We are all Doomed!!! Democracy is Dead".

## 8.9. Sudoku Example

```
Name: Sudoku
Main class: org.drools.examples.sudoku.Main
Type: java application
Rules file: sudokuSolver.drl, sudokuValidator.drl
Objective: Demonstrates the solving of logic problems, and complex pattern
 matching.e
```

This example demonstrates how Drools can be used to find a solution in a large potential solution space based on a number of constraints. We use the popular puzzle of Sudoku. This example also shows how Drools can be integrated into a graphical interface and how callbacks can be used to interact with a running Drools rules engine in order to update the graphical interface based on changes in the working memory at runtime.

### 8.9.1. Sudoku Overview

Sudoku is a logic-based number placement puzzle. The objective is to fill a 9x9 grid so that each column, each row, and each of the nine 3x3 zones contains the digits from 1 to 9 once and only once.

The puzzle setter provides a partially completed grid and the puzzle solver's task is to complete the grid with these constraints.

The general strategy to solve the problem is to ensure that when you insert a new number it should be unique in that particular region(blocks) and also in that particular row and column.

See

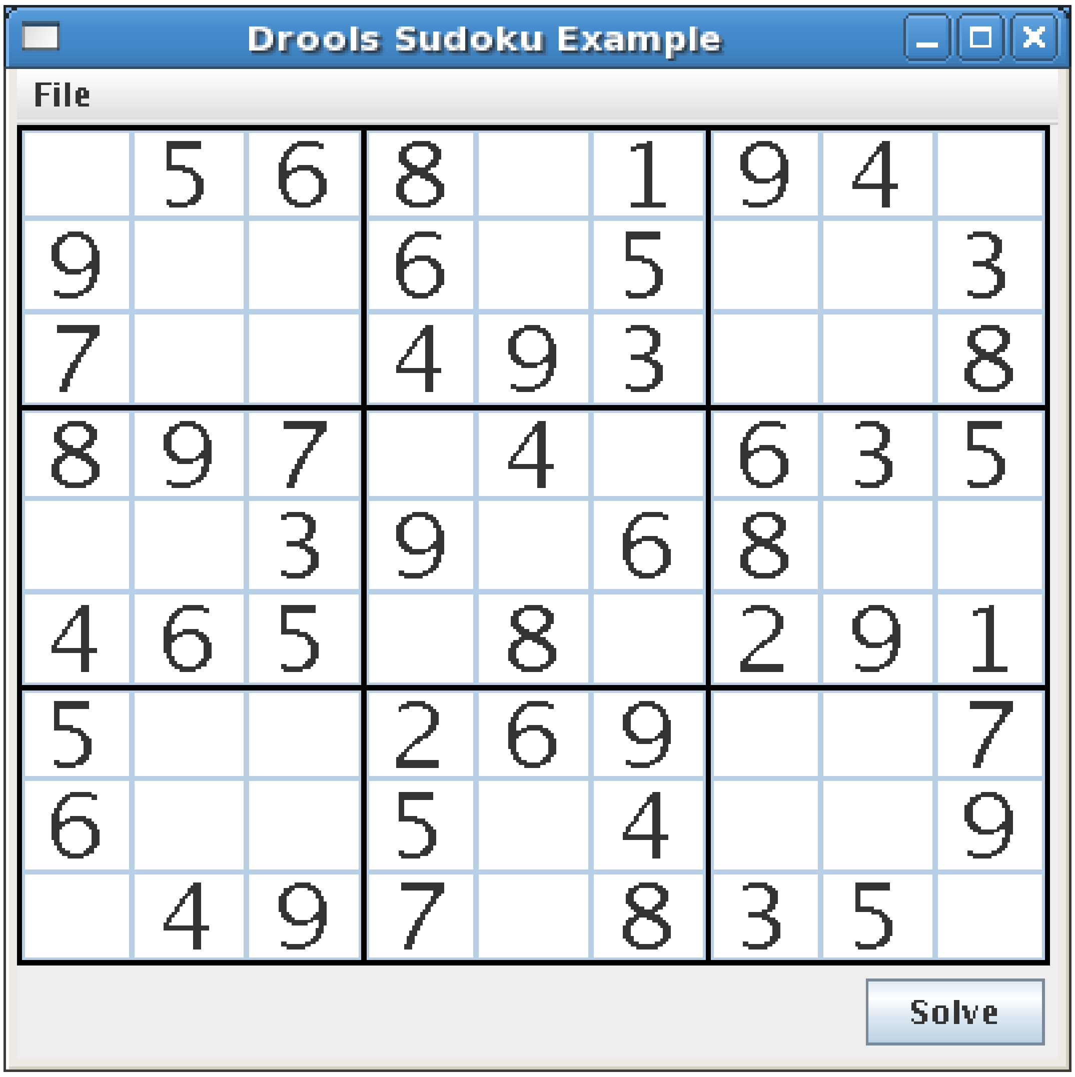
```
URL: http://en.wikipedia.org/wiki/Sudoku
```

for a more detailed description.

### 8.9.2. Running the Example

Download and install drools-examples as described above and then execute java org.drools.examples.sudoku.Main (this example requires Java 5).

A window will be displayed with a relatively simple partially filled grid.

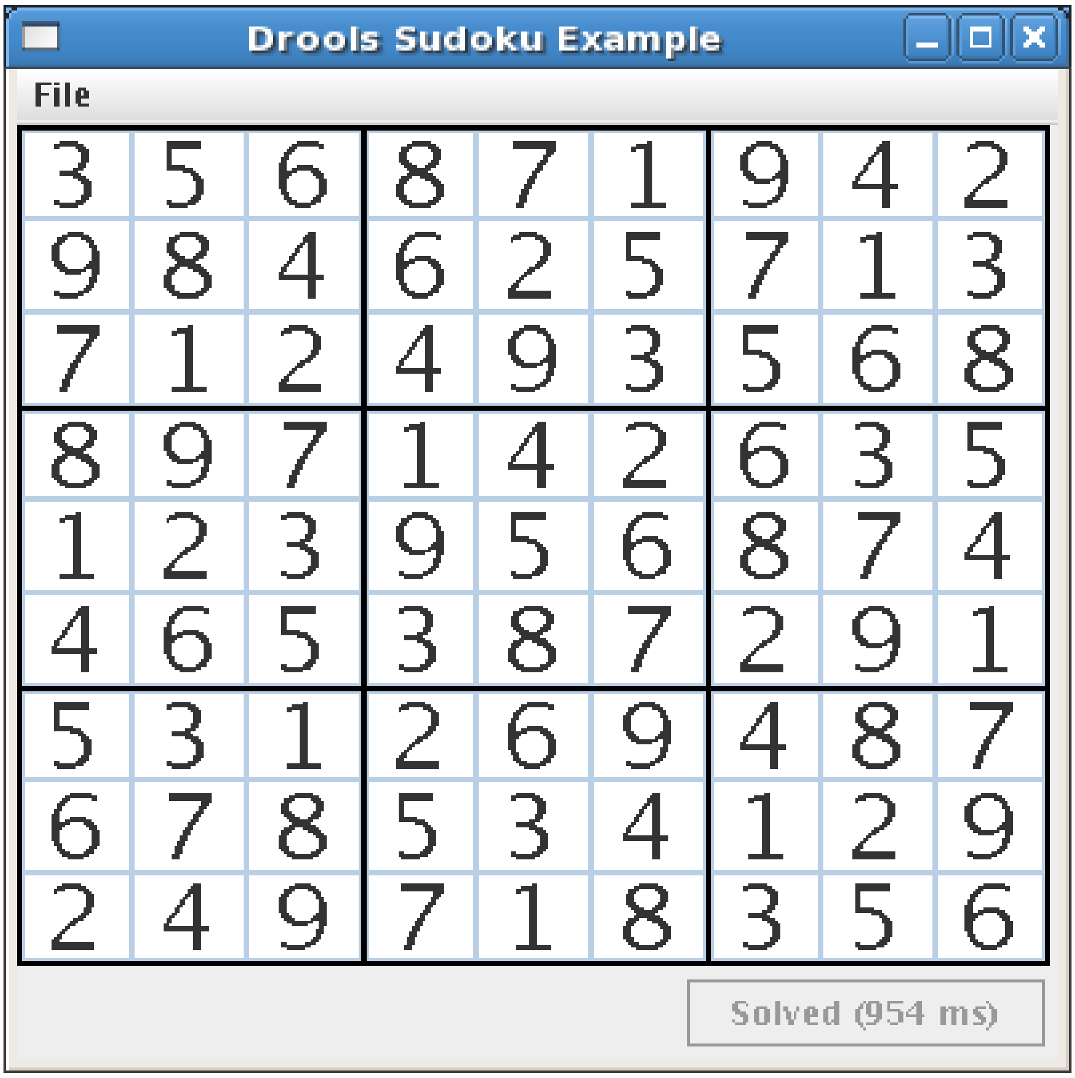


| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| | 5 | 6 | 8 | | 1 | 9 | 4 | |
| 9 | | | 6 | | 5 | | | 3 |
| 7 | | | 4 | 9 | 3 | | | 8 |
| 8 | 9 | 7 | | 4 | | 6 | 3 | 5 |
| | | 3 | 9 | | 6 | 8 | | |
| 4 | 6 | 5 | | 8 | | 2 | 9 | 1 |
| 5 | | | 2 | 6 | 9 | | | 7 |
| 6 | | | 5 | | 4 | | | 9 |
| | 4 | 9 | 7 | | 8 | 3 | 5 | |

Click on the Solve button and the Drools-based engine will fill out the remaining values. The console will display detailed information of the rules which are executing to solve the puzzle in a human readable form.

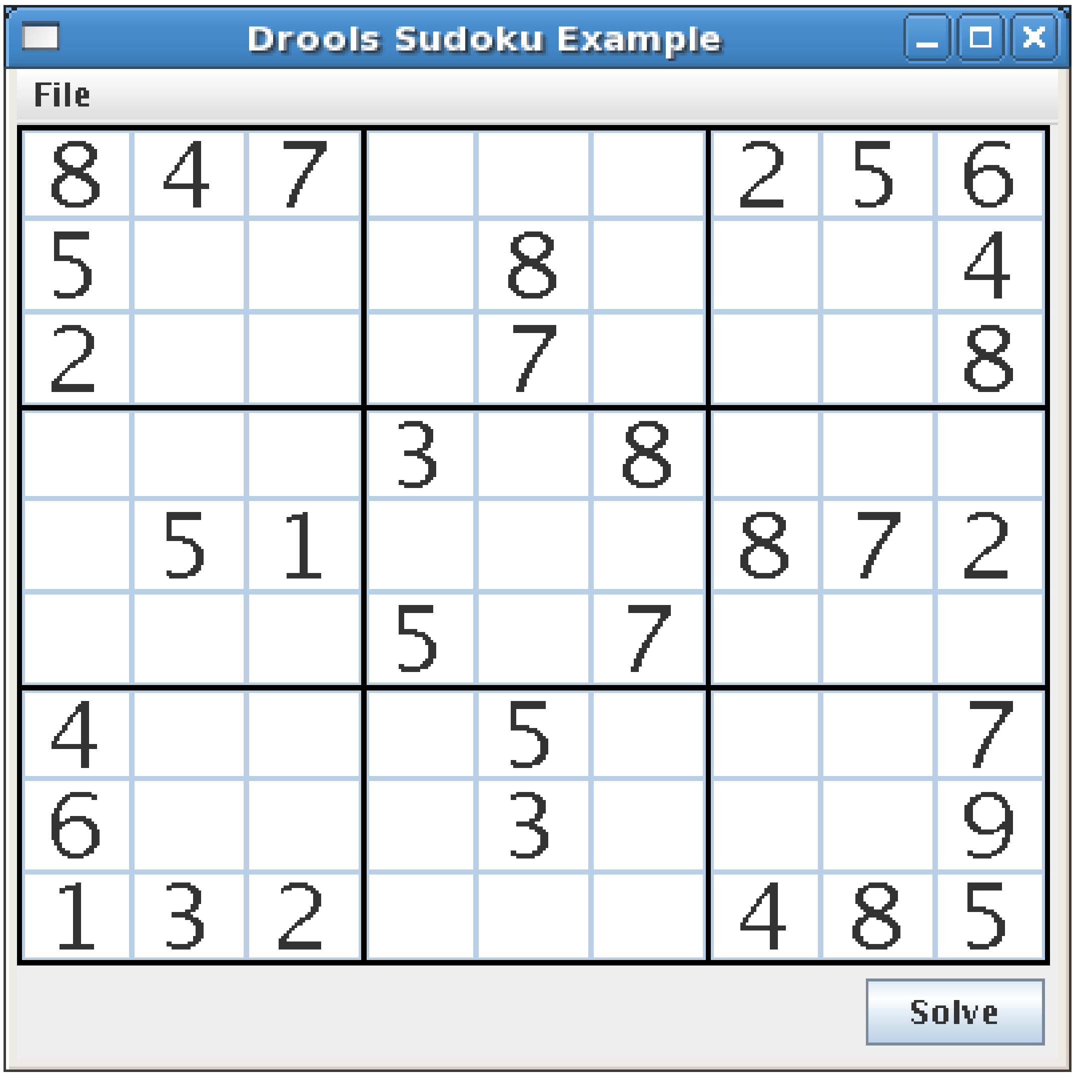
```
Rule #3 determined the value at (4,1) could not be 4 as this value already exists
in the same column at (8,1) Rule #3 determined the value at (5,5) could not be 2
as this value already exists in the same row at (5,6) Rule #7 determined (3,5)
is 2 as this is the only possible cell in the column that can have this value
Rule #1 cleared the other PossibleCellValues for (3,5) as a ResolvedCellValue of
2 exists for this cell. Rule #1 cleared the other PossibleCellValues for (3,5)
as a ResolvedCellValue of 2 exists for this cell. ... Rule #3 determined the
value at (1,1) could not be 1 as this value already exists in the same zone at
(2,1) Rule #6 determined (1,7) is 1 as this is the only possible cell in the row
that can have this value Rule #1 cleared the other PossibleCellValues for (1,7)
as a ResolvedCellValue of 1 exists for this cell. Rule #6 determined (1,1) is 8
as this is the only possible cell in the row that can have this value
```

Once all of the activated rules for the solving logic have executed, the engine executes a second rule base to check that the solution is complete and valid. In this case it is, and the "Solve" button is disabled and displays the text "Solved (1052ms)".



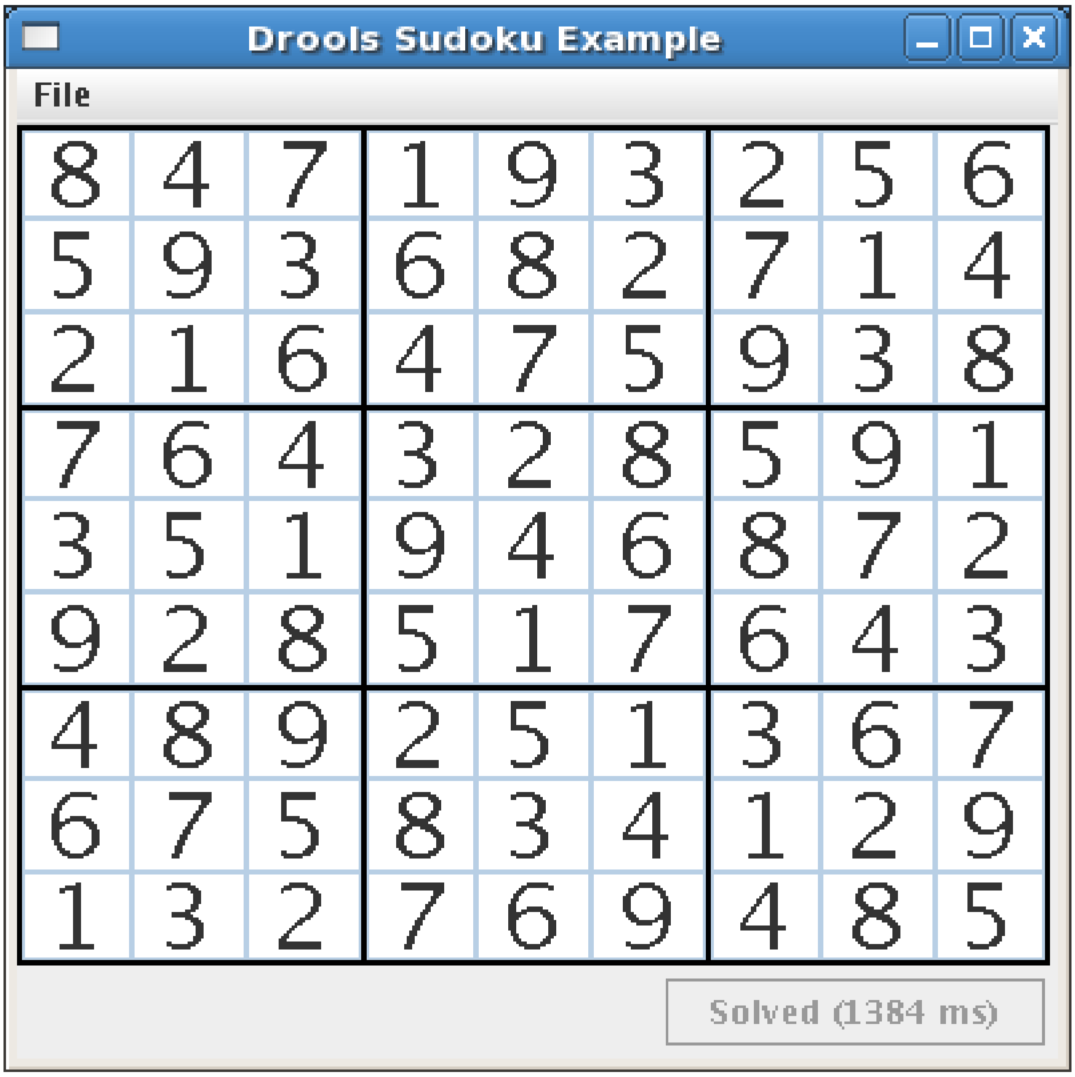
| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 3 | 5 | 6 | 8 | 7 | 1 | 9 | 4 | 2 |
| 9 | 8 | 4 | 6 | 2 | 5 | 7 | 1 | 3 |
| 7 | 1 | 2 | 4 | 9 | 3 | 5 | 6 | 8 |
| 8 | 9 | 7 | 1 | 4 | 2 | 6 | 3 | 5 |
| 1 | 2 | 3 | 9 | 5 | 6 | 8 | 7 | 4 |
| 4 | 6 | 5 | 3 | 8 | 7 | 2 | 9 | 1 |
| 5 | 3 | 1 | 2 | 6 | 9 | 4 | 8 | 7 |
| 6 | 7 | 8 | 5 | 3 | 4 | 1 | 2 | 9 |
| 2 | 4 | 9 | 7 | 1 | 8 | 3 | 5 | 6 |

The example comes with a number of grids which can be loaded and solved. Click on File->Samples->Medium to load a more challenging grid. Note that the solve button is enabled when the new grid is loaded.



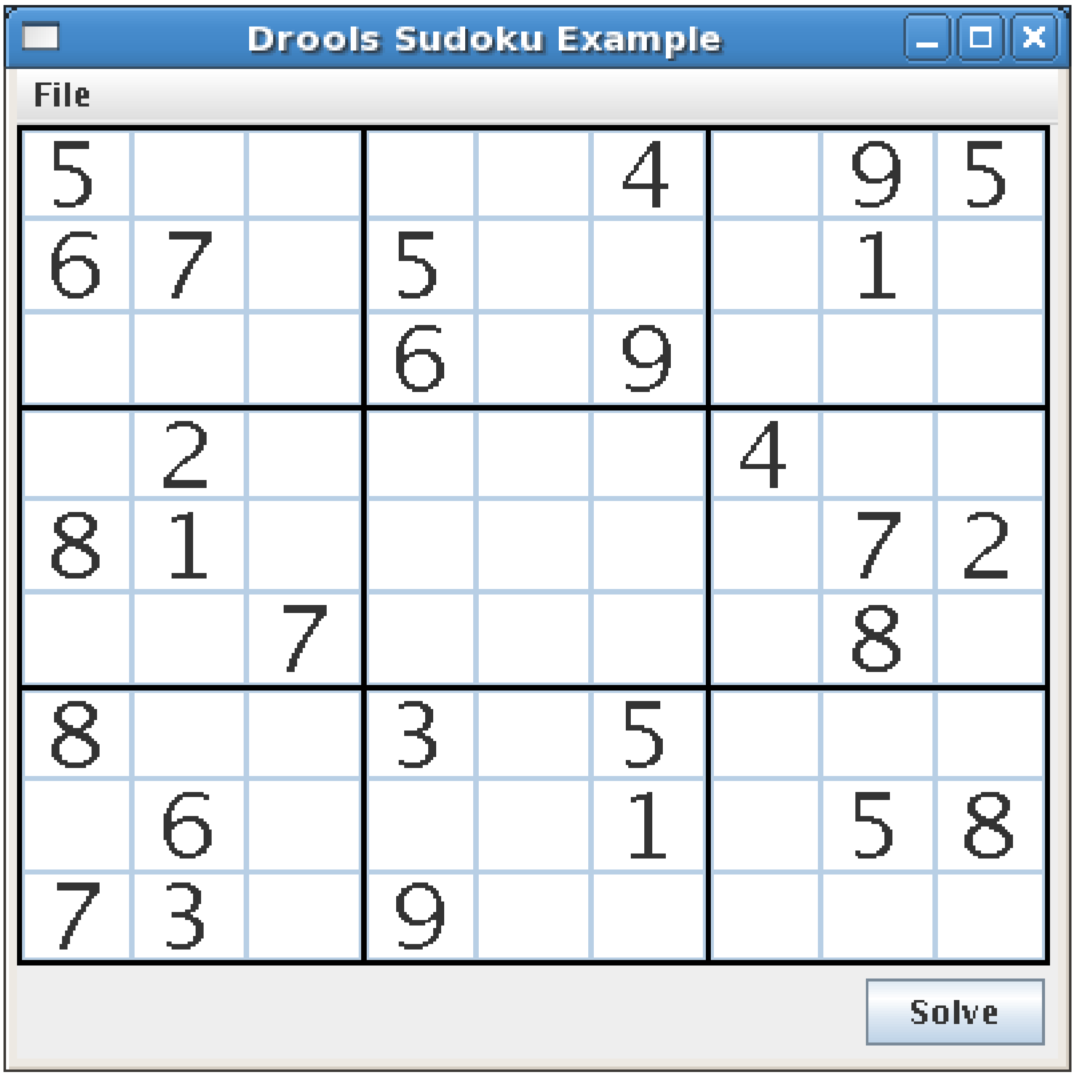
| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 8 | 4 | 7 | | | | 2 | 5 | 6 |
| 5 | | | | 8 | | | | 4 |
| 2 | | | | 7 | | | | 8 |
| | | | 3 | | 8 | | | |
| | 5 | 1 | | | | 8 | 7 | 2 |
| | | | 5 | | 7 | | | |
| 4 | | | | 5 | | | | 7 |
| 6 | | | | 3 | | | | 9 |
| 1 | 3 | 2 | | | | 4 | 8 | 5 |

Click on the "Solve" button again to solve this new grid.



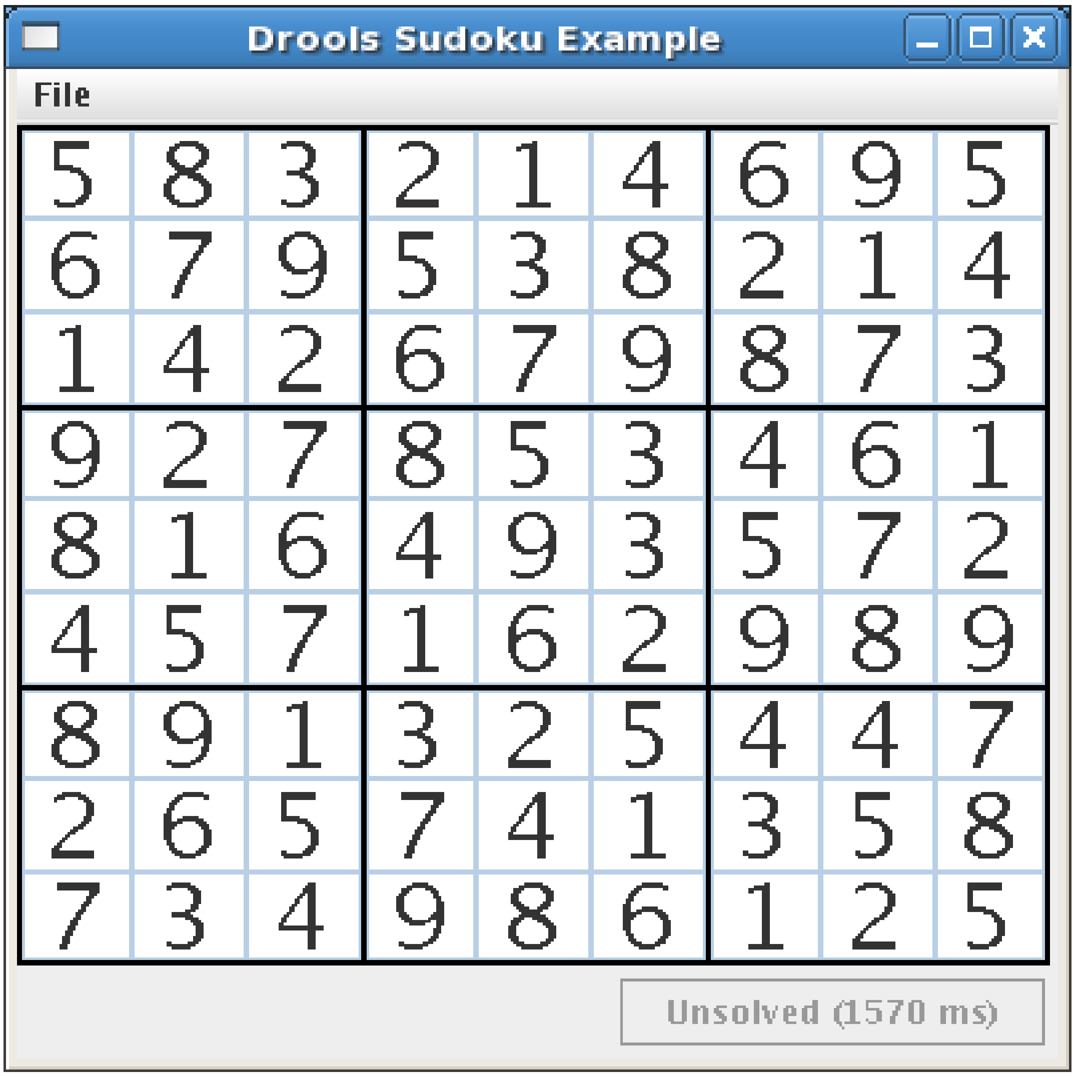
| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 8 | 4 | 7 | 1 | 9 | 3 | 2 | 5 | 6 |
| 5 | 9 | 3 | 6 | 8 | 2 | 7 | 1 | 4 |
| 2 | 1 | 6 | 4 | 7 | 5 | 9 | 3 | 8 |
| 7 | 6 | 4 | 3 | 2 | 8 | 5 | 9 | 1 |
| 3 | 5 | 1 | 9 | 4 | 6 | 8 | 7 | 2 |
| 9 | 2 | 8 | 5 | 1 | 7 | 6 | 4 | 3 |
| 4 | 8 | 9 | 2 | 5 | 1 | 3 | 6 | 7 |
| 6 | 7 | 5 | 8 | 3 | 4 | 1 | 2 | 9 |
| 1 | 3 | 2 | 7 | 6 | 9 | 4 | 8 | 5 |

Now, let us load a Sudoku grid that is deliberately invalid. Click on File->Samples->!DELIBERATELY BROKEN!. Note that this grid starts with some issues, for example the value 5 appears twice in the first row.



| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 5 | | | | | 4 | | 9 | 5 |
| 6 | 7 | | 5 | | | | 1 | |
| | | | 6 | | 9 | | | |
| | 2 | | | | | 4 | | |
| 8 | 1 | | | | | | 7 | 2 |
| | | 7 | | | | | 8 | |
| 8 | | | 3 | | 5 | | | |
| | 6 | | | | 1 | | 5 | 8 |
| 7 | 3 | | 9 | | | | | |

Nevertheless, click on the "Solve" button to apply the solving rules to this invalid Grid. Note that the "Solve" button is relabelled to indicate that the resulting solution is invalid.

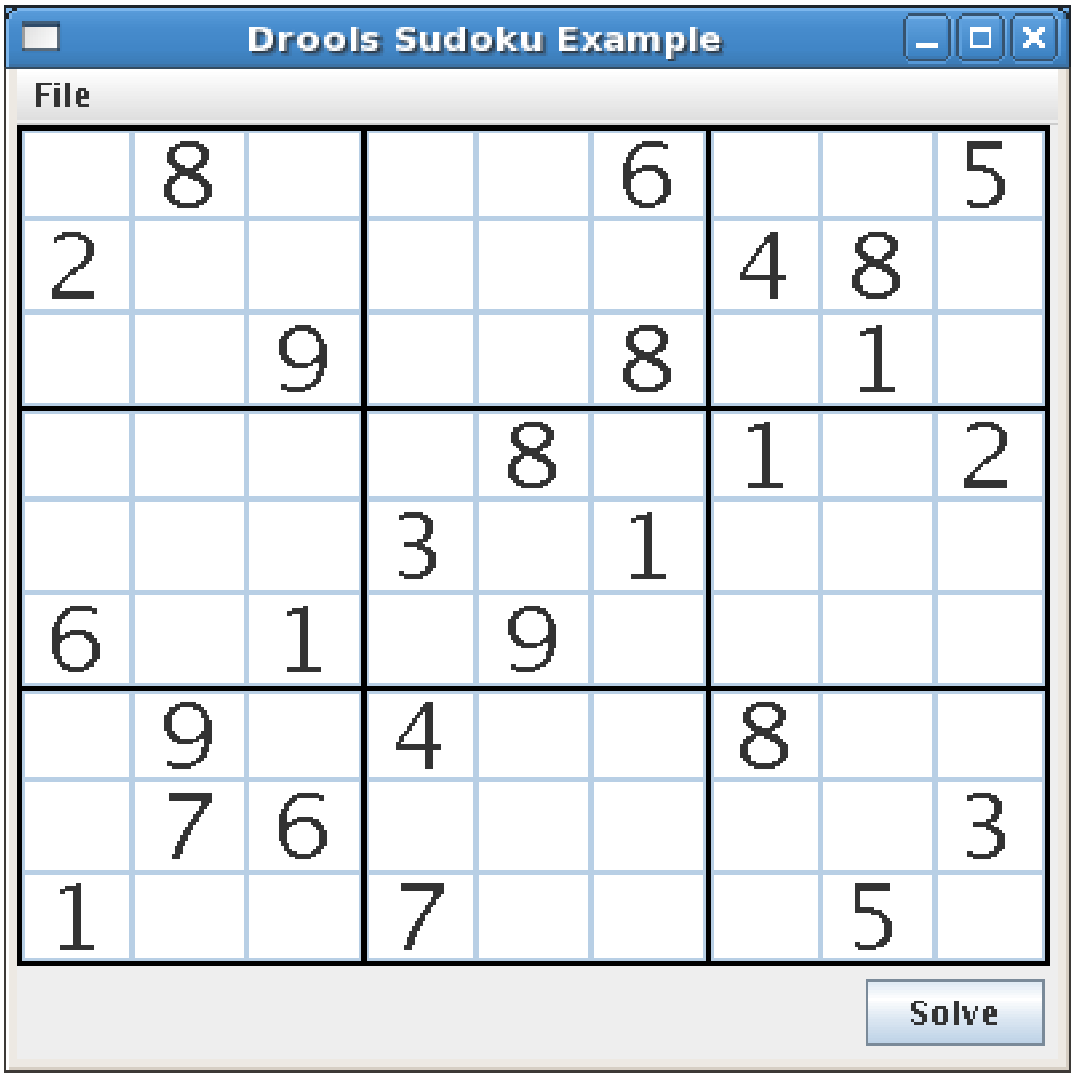


|   |   |   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|---|---|
| 5 | 8 | 3 | 2 | 1 | 4 | 6 | 9 | 5 |
| 6 | 7 | 9 | 5 | 3 | 8 | 2 | 1 | 4 |
| 1 | 4 | 2 | 6 | 7 | 9 | 8 | 7 | 3 |
| 9 | 2 | 7 | 8 | 5 | 3 | 4 | 6 | 1 |
| 8 | 1 | 6 | 4 | 9 | 3 | 5 | 7 | 2 |
| 4 | 5 | 7 | 1 | 6 | 2 | 9 | 8 | 9 |
| 8 | 9 | 1 | 3 | 2 | 5 | 4 | 4 | 7 |
| 2 | 6 | 5 | 7 | 4 | 1 | 3 | 5 | 8 |
| 7 | 3 | 4 | 9 | 8 | 6 | 1 | 2 | 5 |

In addition, the validation rule set outputs all of the issues which are discovered to the console.

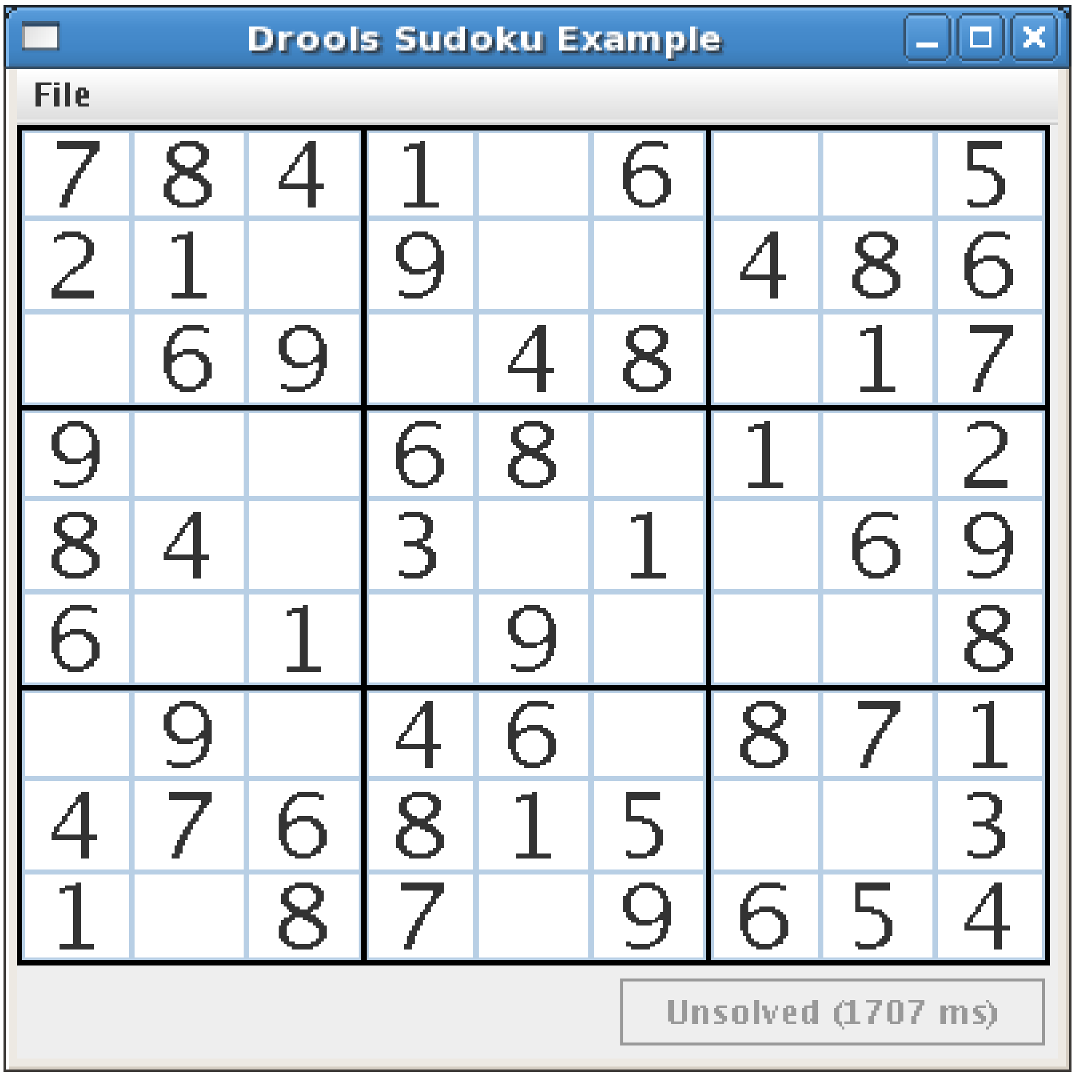
```
There are two cells on the same column with the same value at (6,0) and
 (4,0)
There are two cells on the same column with the same value at (4,0) and
 (6,0)
There are two cells on the same row with the same value at (2,4) and (2,2)
There are two cells on the same row with the same value at (2,2) and (2,4)
There are two cells on the same row with the same value at (6,3) and (6,8)
There are two cells on the same row with the same value at (6,8) and (6,3)
There are two cells on the same column with the same value at (7,4) and
 (0,4)
There are two cells on the same column with the same value at (0,4) and
 (7,4)
There are two cells on the same row with the same value at (0,8) and (0,0)
There are two cells on the same row with the same value at (0,0) and (0,8)
There are two cells on the same column with the same value at (1,2) and
 (3,2)
There are two cells on the same column with the same value at (3,2) and
 (1,2)
There are two cells in the same zone with the same value at (6,3) and (7,3)
There are two cells in the same zone with the same value at (7,3) and (6,3)
There are two cells on the same column with the same value at (7,3) and
 (6,3)
There are two cells on the same column with the same value at (6,3) and
 (7,3)
```

We will look at the solving rule set later in this section, but for the moment we should note that some theoretically solvable solutions can not be solved by the engine as it stands. Click on File->Samples->Hard 3 to load a sparsely populated Grid.



| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| | 8 | | | | 6 | | | 5 |
| 2 | | | | | | 4 | 8 | |
| | | 9 | | | 8 | | 1 | |
| | | | | 8 | | 1 | | 2 |
| | | | 3 | | 1 | | | |
| 6 | | 1 | | 9 | | | | |
| | 9 | | 4 | | | 8 | | |
| | 7 | 6 | | | | | | 3 |
| 1 | | | 7 | | | | 5 | |

Now click on the "Solve" button and note that the current rules are unable to complete the grid, even though (if you are a Sudoku afficiando) you may be able to see a way forward with the solution.



| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 7 | 8 | 4 | 1 | | 6 | | | 5 |
| 2 | 1 | | 9 | | | 4 | 8 | 6 |
| | 6 | 9 | | 4 | 8 | | 1 | 7 |
| 9 | | | 6 | 8 | | 1 | | 2 |
| 8 | 4 | | 3 | | 1 | | 6 | 9 |
| 6 | | 1 | | 9 | | | | 8 |
| | 9 | | 4 | 6 | | 8 | 7 | 1 |
| 4 | 7 | 6 | 8 | 1 | 5 | | | 3 |
| 1 | | 8 | 7 | | 9 | 6 | 5 | 4 |

At the present time, the solving functionality has been achieved by the use of ten rules. This rule set could be extended to enable the engine to tackle more complex logic for filling grids such as this.

### 8.9.3. Java Source and Rules Overview

The Java source code can be found in the /src/main/java/org/drools/examples/sudoku directory, with the two DRL files defining the rules located in the /src/main/rules/org/drools/examples/sudoku directory.

org.drools.examples.sudoku.swing contains a set of classes which implement a framework for Sudoku puzzles. Note that this package does not have any dependencies on the Drools libraries. SudokuGridModel defines an interface which can be implemented to store a Sudoku puzzle as a 9x9 grid of Integer values, some of which may be null, indicating that the value for the cell has not yet been resolved. SudokuGridView is a Swing component which can visualise any implementation of SudokuGridModel. SudokuGridEvent and SudokuGridListener are used to communicate state changes between the model and the view, events are fired when a cell's value is resolved or changed. If you are familiar with the model-view-controller patterns in other Swing components such as JTable then this pattern should be familiar. SudokuGridSamples provides a number of partially filled Sudoku puzzles for demo purposes.

org.drools.examples.sudoku.rules contains an implementation of SudokuGridModel which is based on Drools. Two POJOs are used, both of which extend AbstractCellValue and represent a value for a specific cell in the grid, including the row and column location of the cell, an index of the 3x3 zone the cell is contained in and the value of the cell. PossibleCellValue indicates that we do not currently know for sure what the value in a cell is. There can be 2-9 PossibleCellValues for a given cell. ResolvedCellValue indicates that we have determined what the value for a cell must be. There can only be 1 ResolvedCellValue for a given cell. DroolsSudokuGridModel implements SudokuGridModel and is responsible for converting an initial two dimensional array of partially specified cells into a set of CellValue POJOs, creating a working memory based on solverSudoku.drl and inserting the CellValue POJOs into the working memory. When the solve() method is called it calls fireAllRules() on this working memory to try to solve the puzzle. DroolsSudokuGridModel attaches a WorkingMemoryListener to the working memory, which allows it to be called back on insert() and retract() events as the puzzle is solved. When a new ResolvedCellValue is inserted into the working memory, this call back allows the implementation to fire a SudokuGridEvent to its SudokuGridListeners which can then update themselves in realtime. Once all the rules fired by the solver working memory have executed, DroolsSudokuGridModel runs a second set of rules, based on validatorSudoku.drl which works with the same set of POJOs to determine if the resulting grid is a valid and full solution.

org.drools.examples.sudoku.Main implements a Java application which hooks the components desribed above together.

org.drools.examples.sudoku contains two DRL files. solverSudoku.drl defines the rules which attempt to solve a Sudoku puzzle and validator.drl defines the rules which determin whether the current state of the working memory represents a valid solution. Both use PossibleCellValue and ResolvedCellValue POJOs as their facts and both output information to the console as their rules fire. In a real-world situation we would insert() logging information and use the WorkingMemoryListener to display this information to a user rather than use the console in this fashion.

## 8.9.4. Sudoku Validator Rules (validatorSudoku.drl)

We start with the validator rules as this rule set is shorter and simpler than the solver rule set.

The first rule simply checks that no PossibleCellValue objects remain in the working memory. Once the puzzle is solved, only ResolvedCellValue objects should be present, one for each cell.

The other three rules each match all of the ResolvedCellValue objects and store them in thenew_remote_sitetes instance variable $resolved. They then look respectively for ResolvedCellValues that contain the same value and are located, respectively, in the same row, column or 3x3 zone. If these rules are fired they add a message to a global List of Strings describing the reason the solution is invalid. DroolsSudokoGridModel injects this List before it runs the rule set and checks whether it is empty or not having called fireAllRules(). If it is not empty then it prints all the Strings in the list and sets a flag to indicate that the Grid is not solved.

### 8.9.5. Sudoku Solving Rules (solverSudoku.drl)

Now let us look at the more complex rule set used to solve Sudoku puzzles.

Rule #1 is basically a "book-keeping" rule. Several of the other rules insert() ResolvedCellValues into the working memory at specific rows and columns once they have determined that a given cell must have a certain value. At this point, it is important to clear the working memory of any inserted PossibleCellValues at the same row and column with invalid values. This rule is therefore given a higher salience than the remaining rules to ensure that as soon as the LHS is true, activations for the rule move to the top of the agenda and are fired. In turn this prevents the spurious firing of other rules due to the combination of a ResolvedCellValue and one or more PossibleCellValues being present in the same cell. This rule also calls update() on the ResolvedCellValue, even though its value has not in fact been modified to ensure that Drools fires an event to any WorkingMemoryListeners attached to the working memory so that they can update themselves - in this case so that the GUI can display the new state of the grid.

Rule #2 identifies cells in the grid which have only one possible value. The first line of the when caluse matches all of the PossibleCellValue objects in the working memory. The second line demonstrates a use of the not keyword. This rule will only fire if no other PossibleCellValue objects exist in the working memory at the same row and column but with a different value. When the rule fires, the single PossibleCellValue at the row and column is retracted from the working memory and is replaced by a new ResolvedCellValue at the same row and column with the same value.

Rule #3 removes PossibleCellValues with a given value from a row when they have the same value as a ResolvedCellValue. In other words, when a cell is filled out with a resolved value, we need to remove the possibility of any other cell on the same row having this value. The first line of the when clause matches all ResolvedCellValue objects in the working memory. The second line matches PossibleCellValues which have both the same row and the same value as these ResolvedCellValue objects. If any are found, the rule activates and, when fired retracts the PossibleCellValue which can no longer be a solution for that cell.

Rules #4 and #5 act in the same way as Rule #3 but check for redundant PossibleCellValues in a given column and a given zone of the grid as a ResolvedCellValue respectively.

Rule #6 checks for the scenario where a possible cell value only appears once in a given row. The first line of the LHS matches against all PossibleCellValues in the working memory, storing the result in a number of local variables. The second line checks that no other PossibleCellValues with the same value exist on this row. The third to fifth lines check that there is not a ResolvedCellValue with the same value in the same zone, row or column so that this rule does not fire prematurely. Interestingly we could remove lines 3-5 and give rules #3,#4 and #5 a higher salience to make sure they always fired before rules #6,#7 and #8. When the rule fires, we know that $possible must represent the value for the cell so, as in Rule #2 we retract $possible and replace it with the equivalent, new ResolvedCellValue.

Rules #7 and #8 act in the same way as Rule #2 but check for single PossibleCellValues in a given column and a given zone of the grid respectively.

Rule #9 represents the most complex currently implemented rule. This rule implements the logic that, if we know that a pair of given values can only occur in two cells on a specific row, (for example we have determined the values of 4 and 6 can only appear in the first row in cells 0,3 and 0,5) and this pair of cells can not hold other values then, although we do not know which of the pair contains a four and which contains a six we know that the 4 and the 6 must be in these two cells and hence can remove the possibility of them occuring anywhere else in the same row (phew!). TODO: more detail here and I think the rule can be cleaned up in the DRL file before fully documenting it.

Rules #10 and #11 act in the same way as Rule #9 but check for the existance of only two possible values in a given column and zone respectively.

To solve harder grids, the rule set would need to be extended further with more complex rules that encapsulated more complex reasoning.

### 8.9.6. Suggestions for Future Developments

There are a number of ways in which this example could be developed. The reader is encouraged to consider these as excercises.

- Agenda-group: agenda groups are a great declarative tool for phased execution. In this example, it is easy to see we have 2 phases: "resolution" and "validation". Right now, they are executed by creating two separate rule bases, each for one "job". I think it would be better for us to define agenda-groups for all the rules, spliting them in "resolution" rules and "validation" rules, all loaded in a single rule base. The engine executes resolution and right after that, executes validation.
- Auto-focus: auto focus is a great way of handling exceptions to the regular rules execution. In our case, if we detect an inconsistency, either in the input data or in the resolution rules, why should we spend time continuing the execution if it will be invalid anyway? I think it is better to simply (and immediatly) report the inconsistency as soon as it is found. To do that, since we now have a single rulebase with all rules, we simply need to define auto-focus attribute for all rules validating puzzle consistency.
- Logical insert: an inconsistency only exists while wrong data is in the working memory. As so, we could state that the the validation rules logically insert inconsistencies and as soon as the offending data is retracted, the inconsistency no longer exists.
- session.iterateObjects(): although a valid use case having a global list to add the found problems, I think it would be more interesting to ask the stateful session by the desired list of problems, using session.iterateObjects( new ClassObjectFilter( Inconsistency.class ) ); Having the inconsistency class can also allow us to paint in RED the offending cells in the GUI.
- drools.halt(): even reporting the error as soon as it is found, we need a way to tell the engine to stop evaluating rules. We can do that creating a rule that in the presence of Inconsistencies, calls drools.halt() to stop evaluation.

- queries: looking at the method getPossibleCellValues(int row, int col) in DroolsSudokuGridModel, we see it iterating over all CellValues and looking for the few it wants. That, IMO, is a great opportunity to teach drools queries. We just define a query to return the objects we want and iterate over it. Clean and nice. Other queries may be defined as needed.
- session.iterateObjects(): although a valid use case having a global list to add the found problems, I think it would be more interesting to ask the stateful session by the desired list of problems, using session.iterateObjects( new ClassObjectFilter( Inconsistency.class ) ); Having the inconsistency class can also allow us to paint in RED the offending cells in the GUI.
- Globals as services: the main objective of this change is to attend the next change I will propose, but it is nice by its own I guess. :) In order to teach the use of "globals" as services, it would be nice to setup a call back, so that each rule that finds the ResolvedCellValue for a given cell can call, to notify and update the corresponding cell in the GUI, providing immediate feedback for the user. Also, the last found cell could have its number painted in a different color to facilitate the identification of the rules conclusions.
- Step by step execution: now that we have immediate user feedback, we can make use of the restricted run feature in drools. I.e., we could add a button in the GUI, so that the user clicks and causes the execution of a single rule, by calling fireAllRules( 1 ). This way, the user can see, step by step, what the engine is doing.

## 8.10. Number Guess

```
Name: Number Guess
Main class: org.drools.examples.NumberGuessExample
Type: java application
Rules file: NumberGuess.drl
Objective: Demonstrate use of Rule Flow to organise Rules
```

The "Number Guess" example shows the use of RuleFlow, a way of controlling the order in which rules are fired. It uses widely understood workflow diagrams to make clear the order that groups of rules will be executed.

### Example 8.69. Creating the Number Guess RuleBase - extract 1 from NumberGuessExample.java main() method

```
final KnowledgeBuilder kbuilder =
 KnowledgeBuilderFactory.newKnowledgeBuilder();
        kbuilder.add( ResourceFactory.newClassPathResource(
 "NumberGuess.drl",

 ShoppingExample.class ),
                             ResourceType.DRL );
        kbuilder.add( ResourceFactory.newClassPathResource(
 "NumberGuess.rf",
 ShoppingExample.class ),
                             ResourceType.DRF );

        final KnowledgeBase kbase = KnowledgeBaseFactory.newKnowledgeBase();
        kbase.addKnowledgePackages( kbuilder.getKnowledgePackages() );
```

The creation of the package, and the loading of the rules (using the add() method ) is the same as the previous examples. There is a additional line to add the RuleFlow (NumberGuess.rf) as you have the option of specifying different ruleflows for the same KnowledgeBase. Otherwise the KnowledgeBase is created in the same manner as before.

### Example 8.70. Starting the RuleFlow - extract 2 from NumberGuessExample.java main() method

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

        KnowledgeRuntimeLogger logger =
 KnowledgeRuntimeLoggerFactory.newFileLogger(ksession, "log/numberguess");

        ksession.insert( new GameRules( 100,
                                        5 ) );
        ksession.insert( new RandomNumber() );
        ksession.insert( new Game() );

        ksession.startProcess( "Number Guess" );
        ksession.fireAllRules();

        logger.close();

        ksession.dispose();
```

Once we have a KnowledgeBase we can use it to obtain a stateful session. Into our session we insert our facts (standard Java Objects). For simplicity in this sample, these classes are all contained within our NumberGuessExample.java file. The GameRules class provides the maximum range and the number of guesses allowed. The RandomNumber class automatically generates a number between 0 and 100 and makes it available to our rules after insertion (via the getValue() method). The Game class keeps track of the guesses we have made before, and the number of guesses we have made.

Note that before we call the standard fireAllRules() method, we also start the process that we loaded earlier (via the startProcess() method). We explain where to obtain the parameter we pass ("Number Guess" - the id of the ruleflow) when we talk about the RuleFlow file and the graphical RuleFlow editor below.

Before we finish we our Java code , we note that In 'real life' we would examine the final state of the objects (e.g. how many guesses it took, so that we could add it to a high score table). For this example we are content to ensure the working memory session is cleared by calling the dispose() method.

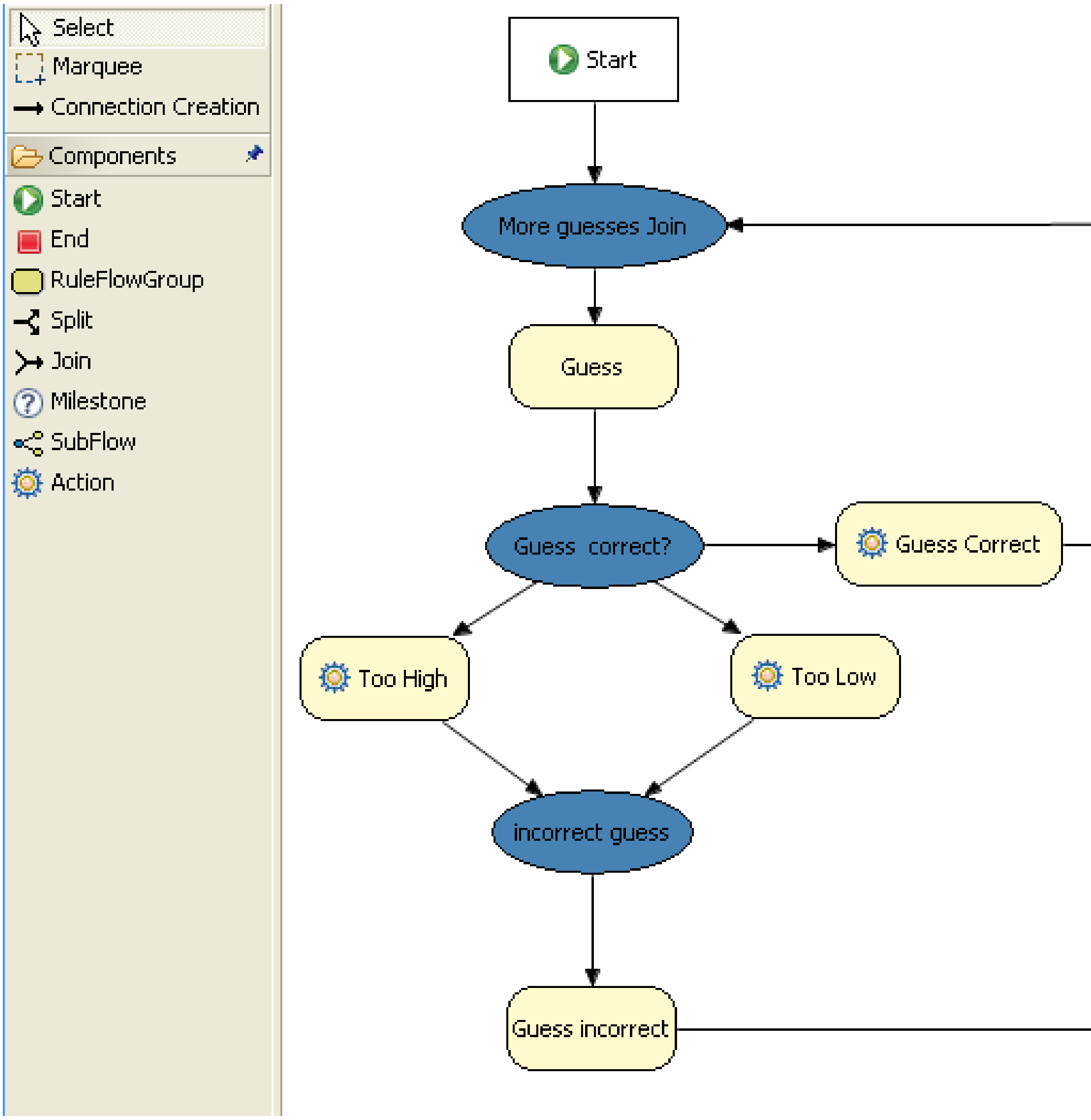
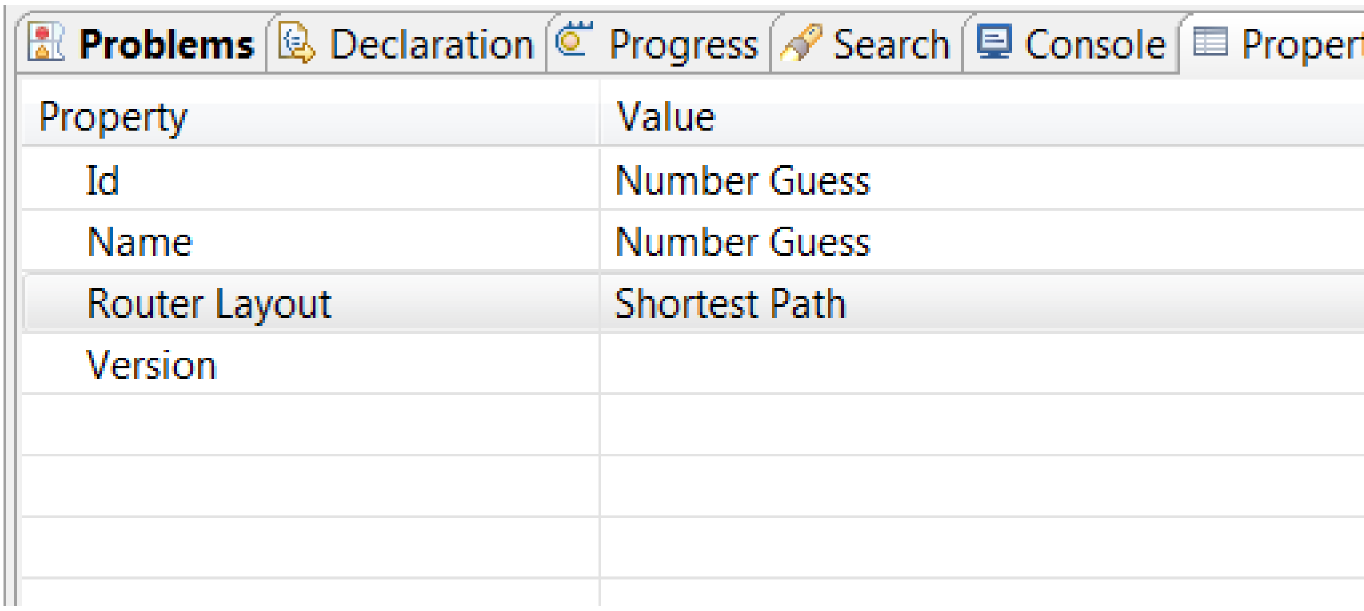


**Figure 8.18. RuleFlow for the NumberGuess Example**

If you open the NumberGuess.rf file open in the Drools IDE (and have the JBoss Rules extensions installed correctly in Eclipse) you should see the above diagram, similar to a standard flowchart.

Its icons are similar (but not exactly the same) as the JBoss jBPM workflow product. Should you wish to edit the diagram, a menu of available components should be available to the left of the diagram in the IDE, which is call the pallete. This diagram is saved in a (almost human) readable xml format, using xstream.

If it is not already open, ensure the properties view is visible in the IDE. It can opened by selecting Window -> Show View -> Other and then select the Properties view. If you do this **before** you select any item on the RuleFlow (or click on blank space in the RuleFlow) you should be presented with the following set of properties.



| Property | Value |
|---|---|
| Id | Number Guess |
| Name | Number Guess |
| Router Layout | Shortest Path |
| Version | |

## Figure 8.19. Properties for the Number Guess RuleFlow

Keep an eye on the properties view as we progress through the example RuleFlow as it gives valuable information. In this case it provides us with the ID of the RuleFlow process that we used in our earlier code example when we called session.startprocess().

To give an overview of each of the node types (boxes) in the NumberGuess RuleFlow.

- The Start and End nodes (green arrow and red box) are where the RuleFlow starts and ends.
- RuleFlowGroup (simple yellow box). These map to the RuleFlowGroups in our rules (DRL) file that we will look at later. For example when the flow reaches the 'Too High' RuleFlowGroup, only those rules marked with an attribute of **ruleflow-group "Too High"** can potentially fire.
- Action Nodes (yellow box with cog like icon). These can perform standard Java method calls. Most action nodes in this example call System.out.println to give an indication to the user of what is going on.
- Split and Join Nodes (Blue Ovals) such as "Guess Correct" and "More Guesses Join" where the flow of control can split (according to various conditions) and / or rejoin.

- Arrows that indicate the flow between the various nodes.

These various nodes work together with the Rules to make the Number Guess game work. For example, the "Guess" RuleFlowGroup allows only the rule "Get user Guess" to fire (details below) as only that Rule has a matching attribute of **ruleflow-group "Guess"**

### Example 8.71. A Rule that will fire only a specific point in the RuleFlow - extract from NumberGuess.drl

```
rule "Get user Guess"
 ruleflow-group "Guess"
 no-loop
 when
     $r : RandomNumber()
     rules : GameRules( allowed : allowedGuesses )
     game : Game( guessCount < allowed )
     not ( Guess() )
 then
     System.out.println( "You have " + ( rules.allowedGuesses -
 game.guessCount )
     + " out of " + rules.allowedGuesses + " guesses left.\nPlease enter
 your guess
     from 0 to " + rules.maxRange );
        br = new BufferedReader( new InputStreamReader( System.in ) );
        modify ( game ) { guessCount = game.guessCount + 1 }
        i = br.readLine();
    insert( new Guess( i ) );
end
```

The rest of this rule is fairly standard : The **LHS (when)** section of the rule states that it will be activated for each *RandomNumber* object inserted into the working memory where *guessCount* is less than the *allowedGuesses* ( read from the GameRules Class) and where the user has not guessed the correct number.

The **RHS (consequence, then)** prints a message to the user, then awaits user input from *System.in.* After getting this input (as System.in blocks until the <return> key is pressed) it updates/modifes the guess count, the actual guess and makes both available in the working memory.

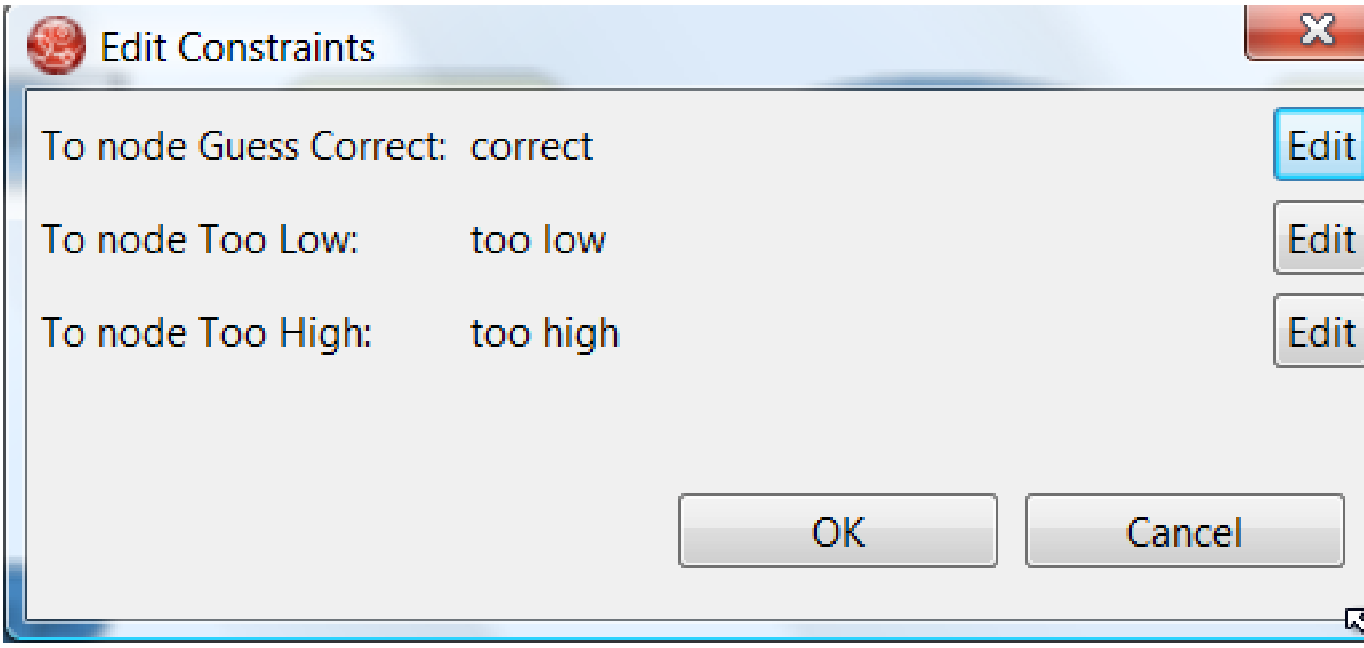
The rest of the Rules file is fairly standard ; the package declares the dialect is set to MVEL, various Java classes are imported. In total, there are five rules in this file:

1. Get User Guess, the Rule we examined above.
2. A Rule to record the highest guess.
3. A Rule to record the lowest guess.

4. A Rule to inspect the guess and retract it from memory if incorrect.

5. A Rule that notifies the user that all guesses have been used up.

One point of integration between the standard Rules and the RuleFlow is via the 'ruleflow-group' attribute on the rules (as dicussed above). A **second point of integration between the Rules File (drl) and the Rules Flow .rf files** is that the Split Nodes (the blue ovals) can use values in working memory (as updated by the Rules) to decide which flow of action to take. To see how this works click on the "Guess Correct Node" ; then within the properties view, open the constraints editor (the button at the right that appears once you click on the 'Constraints' property line). You should see something similar to the Diagram below.

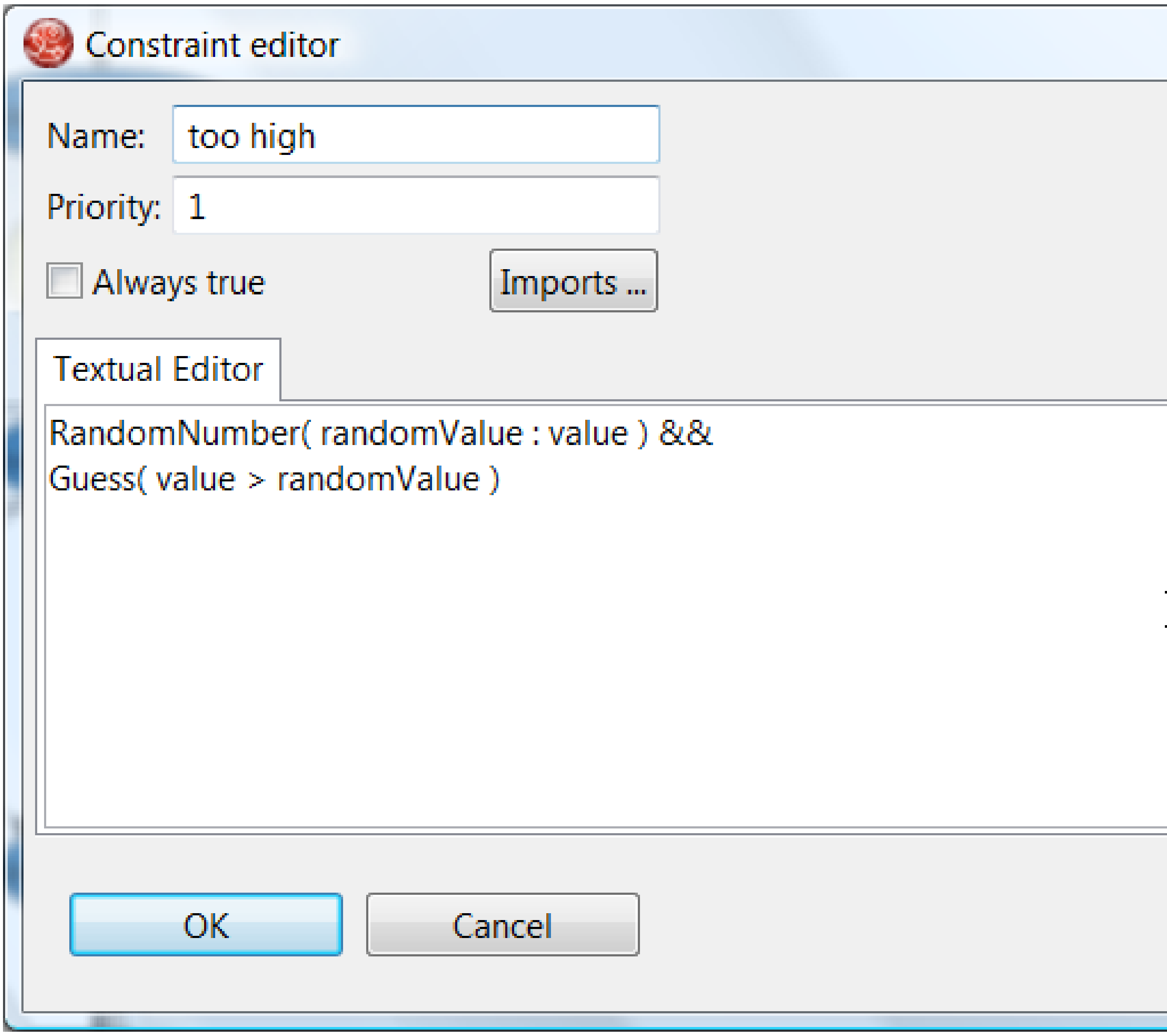


Edit Constraints

|  |  |  |
|---|---|---|
| To node Guess Correct: | correct | Edit |
| To node Too Low: | too low | Edit |
| To node Too High: | too high | Edit |

**Figure 8.20. Edit Constraints for the GuessCorrect Node**

Click on 'Edit' beside 'To node Too High' and you see a dialog like the one below. The values in the 'Textual Editor' follow the standard Rule Format (LHS) and can refer to objects in working memory. The consequence (RHS) is that the flow of control follows this node (i.e. To node Too high') if the LHS expression evaluates to true.



Name: too high

Priority: 1

```
RandomNumber( randomValue : value ) &&
Guess( value > randomValue )
```

**Figure 8.21. Constraints Editor for the GuessCorrect Node / value too high**

Since the NumberGuess.java example contains a main() method, it can be run as a standard Java application (either from the command line or via the IDE). A typical game might result in the interaction below (the numbers in bold are typed in by the user).

**Example 8.72. Example Console output where the Number Guess Example beat the human!**

```
You have 5 out of 5 guesses left.
Please enter your guess from 0 to 100
50
Your guess was too high
You have 4 out of 5 guesses left.
```

```
Please enter your guess from 0 to 100
25
Your guess was too low
You have 3 out of 5 guesses left.
Please enter your guess from 0 to 100
37
Your guess was too low
You have 2 out of 5 guesses left.
Please enter your guess from 0 to 100
44
Your guess was too low
You have 1 out of 5 guesses left.
Please enter your guess from 0 to 100
47
Your guess was too low
You have no more guesses
The correct guess was 48
```

A summary of what is happening in this sample is:

1. Main() method of NumberGuessExample.java loads RuleBase, gets a StatefulSession and inserts Game, GameRules and RandomNumber (containing the target number) objects into it. This method sets the process flow we are going to use, and fires all rules. Control passes to the RuleFlow.

2. The NumberGuess.rf RuleFlow begins at the Start node.

3. Control passes (via the "more guesses" join node) to the Guess Node..

4. At the Guess node, the appropriate RuleFlowGroup ("Get user Guess") is enabled. In this case the Rule "Guess" (in the NumberGuess.drl file) is triggered. This rule displays a message to the user, takes the response, and puts it into memory. Flow passes to the next Rule Flow Node.

5. At the next node , "Guess Correct", constraints inspect the current session and decide which path we take next.

   If the guess in step 4 was too high / too low flow procees along a path which has (i) An action node with normal Java code prints a too high / too low statement and (ii) a RuleFlowGroup causes a highest guess / lowest guess Rule to be triggered in the Rules file. Flow passes from these nodes to step 6.

   If the guess in step 4 just right we proceed along the path towards the end of the Rule Flow. Before we get there, an action node with normal Java code prints a statement "you guessed correctly". There is a join node here (just before the Rule Flow End) so that our no-more-guesses path (step 7) can also terminate the RuleFlow.

6. Control passes as per the RuleFlow via a join node, a guess incorrect RuleFlowGroup (triggers a rule to retract a guess from working memory) onto the "more guesses" decision node.

7. The "more guesses" decision node (right hand side of ruleflow) uses constraints (again looking at values that the Rules have put into the working memory) to decide if we have more guesses and if so, goto step 3. If not we proceed to the end of the workflow, via a RuleFlowGroup that triggers a rule stating "you have no more guesses".

8. The Loop 3-7 continues until the number is guessed correctly, or we run out of guesses.

## 8.11. Miss Manners and Benchmarking

```
Name: Miss Manners
Main class: org.drools.benchmark.manners.MannersBenchmark
Type: java application
Rules file: manners.drl
Objective: Advanced walkthrough on the Manners benchmark, covers Depth
 conflict resolution in depth.
```

### 8.11.1. Introduction

Miss Manners is throwing a party and being the good host she wants to arrange good seating. Her initial design arranges everyone in male female pairs, but then she worries about people have things to talk about; what is a good host to do? So she decides to note the hobby of each guest so she can then arrange guests in not only male and female pairs but also ensure that a guest has someone to talk about a common hobby, from either their left or right side.

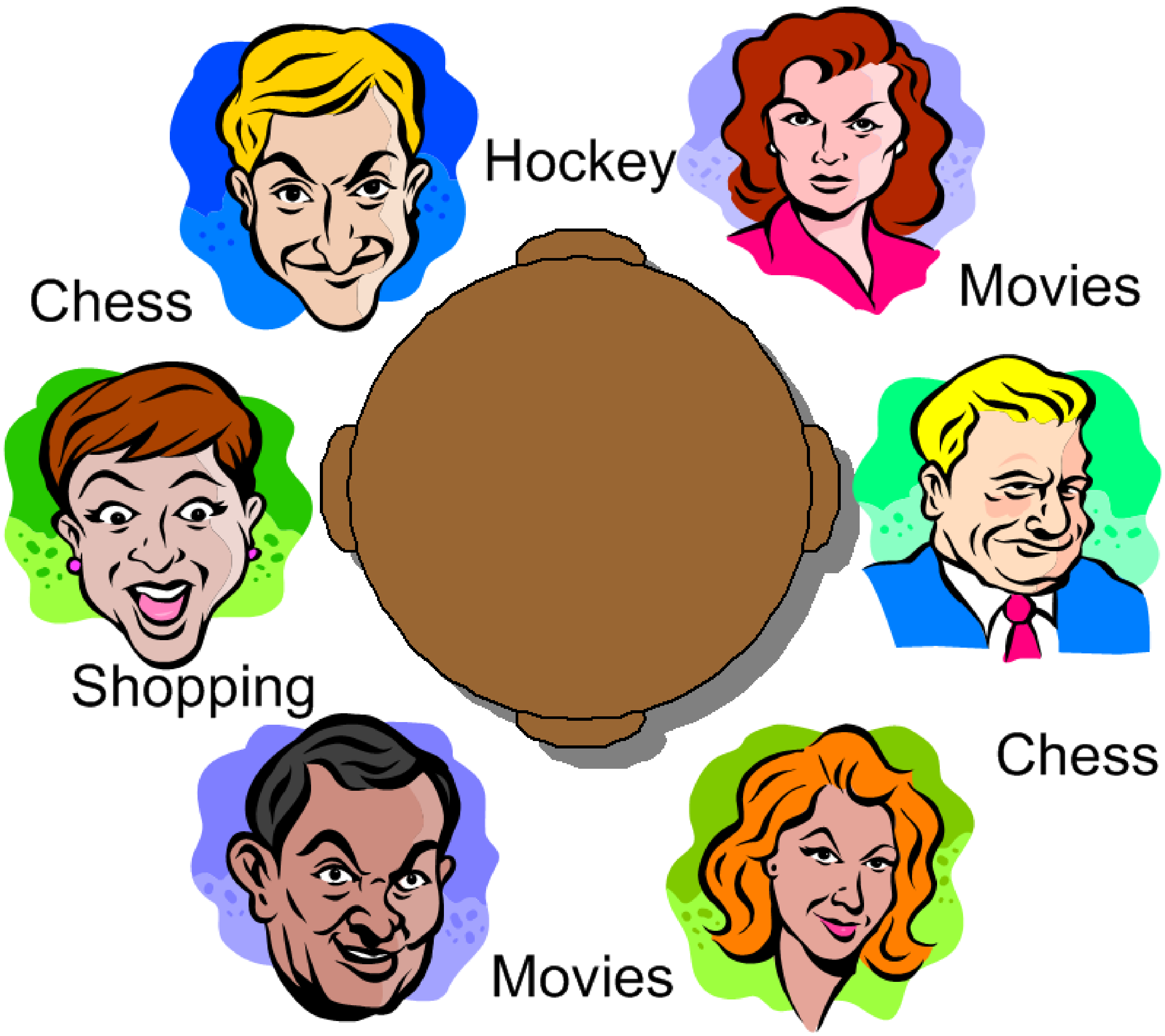


**Figure 8.22. Miss Manners' Guests**

#### 8.11.1.1. BenchMarking

5 benchmarks were established in the 1991 paper "Effects of Database Size on Rule System Performance: Five Case Studies" by Brant, Timothy Grose, Bernie Lofaso, & Daniel P. Miranker.

- Manners
  - Uses a depth-first search approach to determine the seating arrangements of boy/girl and one common hobby for dinner guests
- Waltz
  - line labeling for simple scenes by constraint propagation

- WaltzDB
  - More general version of Walts to be able to adapt to a database of facts
- ARP
  - Route planner for a robotic air vehicle using the A* search algorithm
- Weavera
  - VLSI router for channels and boxes using a black-board technique

Manners has become the de facto rule engine benchmark; however it's behavior is now well known and many engines optimize for this thus negating its usefulness as a benchmark which is why Waltz is becoming more favorable. These 5 benchmarks are also published at the University of Texas *http://www.cs.utexas.edu/ftp/pub/ops5-benchmark-suite/*.

#### 8.11.1.2. Miss Manners Execution Flow

After the first Seating arrangement has been assigned a depth-first recursion occurs which repeatedly assigns correct Seating arrangements until the last seat is assigned. Manners uses a Context instance to control execution flow; the activity diagram is partitioned to show the relation of the rule execution to the current Context state.

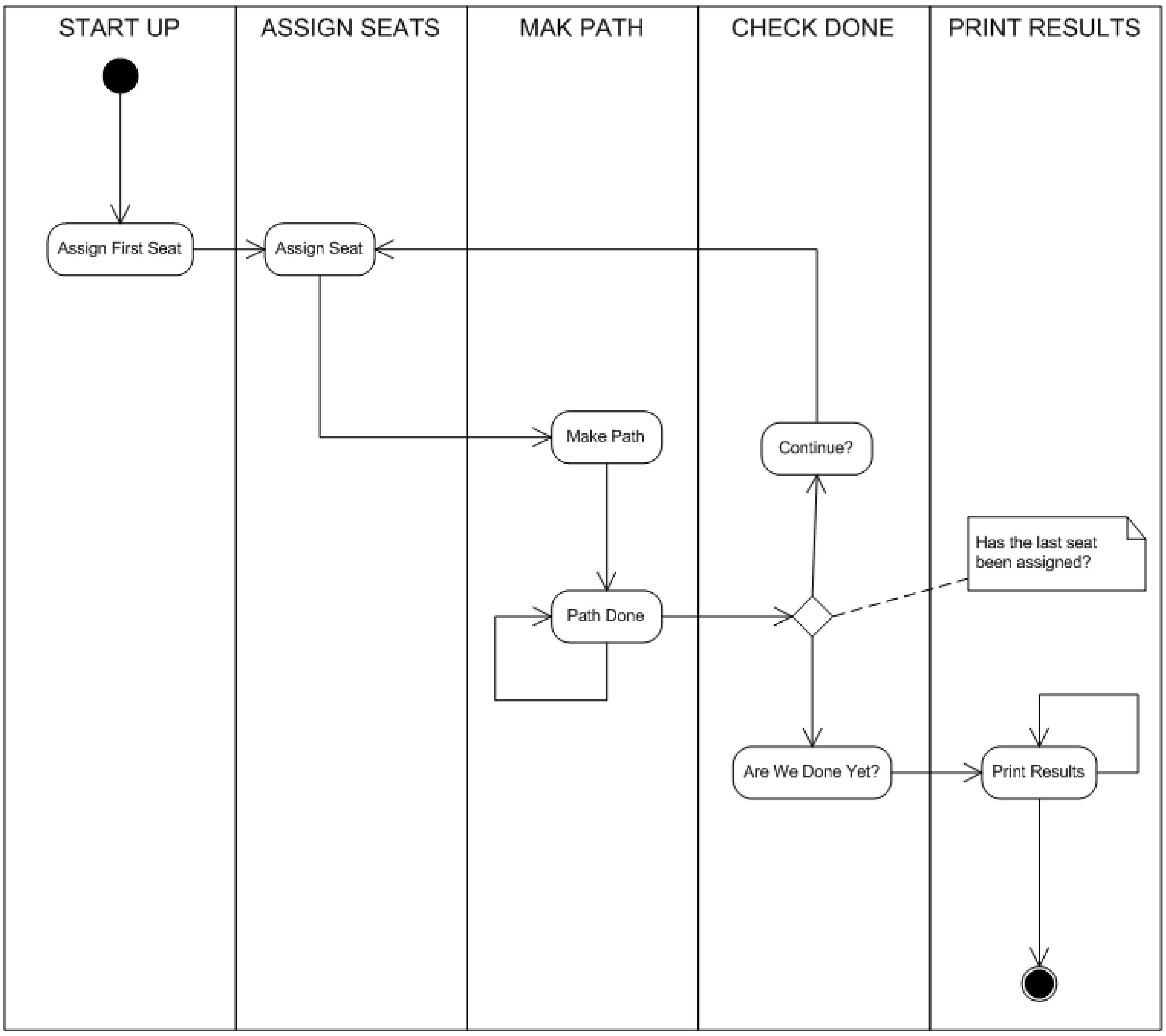


**Figure 8.23. Manners Activity Diagram**

### 8.11.1.3. The Data and Results

Before going deeper into the rules lets first take a look at the asserted data and the resulting Seating arrangement. The data is a simple set of 5 guests who should be arranged in male/female pairs with common hobbies.

**The Data**

Each line of the results list is printed per execution of the "Assign Seat" rule. They key bit to notice is that each line has pid one greater than the last, the significance of this will be explained in t he "Assign Seating" rule description. The 'l' and the 'r' refer to the left and right, 's' is sean and 'n' is the guest name. In my actual implementation I used longer notation, 'leftGuestName', but this is not practice in a printed article. I found the notation of left and right preferable to the original OPS5 '1' and '2

```
(guest (name n1) (sex m) (hobby  h1)  )
(guest (name n2) (sex f) (hobby  h1)  )
(guest (name n2) (sex f) (hobby  h3)  )
(guest (name n3) (sex m) (hobby  h3)  )
(guest (name n4) (sex m) (hobby  h1)  )
(guest (name n4) (sex f) (hobby  h2)  )
(guest (name n4) (sex f) (hobby  h3)  )
(guest (name n5) (sex f) (hobby  h2)  )
(guest (name n5) (sex f) (hobby  h1)  )
(last_seat (seat 5)  )
```

**The Results**

```
[Seating id=1, pid=0, done=true, ls=1, ln=n5, rs=1, rn=n5]
[Seating id=2, pid=1, done=false, ls=1, ln=n5, rs=2, rn=n4]
[Seating id=3, pid=2, done=false, ls=2, ln=n4, rs=3, rn=n3]
[Seating id=4, pid=3, done=false, ls=3, rn=n3, rs=4, rn=n2]
[Seating id=5, pid=4, done=false, ls=4, ln=n2, rs=5, rn=n1]
```

## 8.11.2. Indepth look

### 8.11.2.1. Cheating

Manners has been around a long time and is a contrived benchmark meant to exercise the cross product joins and agenda, many people not understanding this tweak the example to achieve better performance, making their use of the Manners benchmark pointless. Known cheats to Miss Manners are:

- Using arrays for a guests hobbies, instead of asserting each one as a single fact. This massively reduces the cross products.
- The altering of the sequence of data can also reducing the amount of matching increase execution speed
- Changing NOT CE (conditional element) such that the test algorithm only uses the "first-best-match". Basically, changing the test algorithm to backward chaining. the results are only comparable to other backward chaining rule engines or ports of Manners.
- Removing the context so the rule engine matches the guests and seats pre-maturely. A proper port will prevent facts from matching using the context start.
- Any change which prevents the rule engine from performing combinatorial pattern matching

- If no facts are retracted in the reasoning cycle, as a result of NOT CE, the port is incorrect.

#### 8.11.2.2. Conflict Resolution

Manners benchmark was written for OPS5 which has two conflict resolution strategies, LEX and MEA; LEX is a chain of several strategies including Salience, Recency, Complexity. The Recency part of the strategy drives the depth first (LIFO) firing order. The Clips manual documents the recency strategy as:

> Every fact and instance is marked internally with a "time tag" to indicate its relative recency with respect to every other fact and instance in the system. The pattern entities associated with each rule activation are sorted in descending order for determining placement. An activation with a more recent pattern entities is placed before activations with less recent pattern entities. To determine the placement order of two activations, compare the sorted time tags of the two activations one by one starting with the largest time tags. The comparison should continue until one activation’s time tag is greater than the other activation’s corresponding time tag. The activation with the greater time tag is placed before the other activation on the agenda. If one activation has more pattern entities than the other activation and the compared time tags are all identical, then the activation with more time tags is placed before the other activation on the agenda.
>
> —Clips Reference Manual

However Jess and Clips both use the Depth strategy, which is simpler and lighter, which Drools also adopted. The Clips manual documents the Depth strategy as:

> Newly activated rules are placed above all rules of the same salience. For example, given that fact-a activates rule-1 and rule-2 and fact-b activates rule-3 and rule-4, then if fact-a is asserted before fact-b, rule-3 and rule-4 will be above rule-1 and rule-2 on the agenda. However, the position of rule-1 relative to rule-2 and rule-3 relative to rule-4 will be arbitrary.
>
> —Clips Reference Manual

The initial Drools implementation for the Depth strategy would not work for manners without the use of salience on the "make_path" rule, the Clips support team had this to say:

> The default conflict resolution strategy for CLIPS, depth, is different than the default conflict resolution strategy used by OPS5. Therefore if you directly translate an OPS5 program to CLIPS, but use the default depth conflict resolution strategy, you're only likely to get the correct behavior by coincidence. The lex and mea conflict resolution strategies are provided in CLIPS to allow you to quickly convert and correctly run an OPS5 program in CLIPS
>
> —Clips Support Forum

Investigation into the Clips code reveals there is undocumented functionality in the Depth strategy. There is an accumulated time tag used in this strategy; it's not an extensively fact by fact comparison as in the recency strategy, it simply adds the total of all the time tags for each activation and compares.

### 8.11.2.3. Assign First Seat

Once the context is changed to START_UP Activations are created for all asserted Guests; because all Activations are created as the result of a single Working Memory action, they all have the same Activation time tag. The last asserted Guest would have a higher fact time tag and its Activation would fire, becuase it has the highest accumulated fact time tag. The execution order in this rule has little importance, but has a big impact in the rule "Assign Seat". The Activation fires and asserts the first Seating arrangement, a Path and then sets the Context's state to create Activation for "Assign Seat".

```
rule assignFirstSeat
    when
        context : Context( state == Context.START_UP )
        guest : Guest()
        count : Count()
    then
        String guestName = guest.getName();

        Seating seating =  new Seating( count.getValue(), 1, true, 1,
 guestName, 1, guestName);
        insert( seating );

        Path path = new Path( count.getValue(), 1, guestName );
        insert( path );

        modify( count ) { setValue ( count.getValue() + 1 )  }

  System.out.println( "assign first seat :  " + seating + " : " + path );

        modify( context ) {
            setState( Context.ASSIGN_SEATS )
        }
end
```

### 8.11.2.4. Assign Seat

This rule determines each of the Seating arrangements. The Rule creates cross product solutions for ALL asserted Seating arrangements against ALL the asserted guests; accept against itself or any already assigned Chosen solutions.

```
rule findSeating
   when
       context : Context( state == Context.ASSIGN_SEATS )
       $s      : Seating( pathDone == true )
       $g1     : Guest( name == $s.rightGuestName )
       $g2     : Guest( sex != $g1.sex, hobby == $g1.hobby )

       count   : Count()
```

```
        not ( Path( id == $s.id, guestName == $g2.name) )
        not ( Chosen( id == $s.id, guestName == $g2.name, hobby == $g1.hobby)
 )
    then
        int rightSeat = $s.getRightSeat();
        int seatId = $s.getId();
        int countValue = count.getValue();

        Seating seating = new Seating( countValue, seatId, false, rightSeat,
 $s.getRightGuestName(), rightSeat + 1, $g2.getName() );
        insert( seating );

        Path path = new Path( countValue, rightSeat + 1, $g2.getName()  );
        insert( path );

        Chosen chosen = new Chosen( seatId, $g2.getName(), $g1.getHobby() );
        insert( chosen  );

     System.err.println( "find seating : " + seating + " : " + path + " : " +
 chosen);

        modify( count ) {setValue(  countValue + 1 )}
        modify( context ) {setState( Context.MAKE_PATH )}
end
```

However, as can be seen from the printed results shown earlier, it is essential that only the Seating with the highest pid cross product be chosen – yet how can this be possible if we have Activations, of the same time tag, for nearly all existing Seating and Guests. For example on the third iteration of "Assing Seat" these are the produced Activations, remember this is from a very small data set and with larger data sets there would be many more possible Activated Seating solutions, with multiple solutions per pid:

```
=>[ActivationCreated(35): rule=findSeating
[fid:19:33]:[Seating id=3, pid=2, done=true, ls=2, ln=n4, rs=3, rn=n3]
[fid:4:4]:[Guest name=n3, sex=m, hobbies=h3]
[fid:3:3]:[Guest name=n2, sex=f, hobbies=h3]

=>[ActivationCreated(35): rule=findSeating
[fid:15:23]:[Seating id=2, pid=1, done=true, ls=1, ln=n5, rs=2, rn=n4]
[fid:5:5]:[Guest name=n4, sex=m, hobbies=h1]
[fid:2:2]:[Guest name=n2, sex=f, hobbies=h1]

=>[ActivationCreated(35): rule=findSeating
[fid:13:13]:[Seating id=1, pid=0, done=true, ls=1, ln=n5, rs=1, rn=n5]
[fid:9:9]:[Guest name=n5, sex=f, hobbies=h1]
[fid:1:1]:[Guest name=n1, sex=m, hobbies=h1]
```

The creation of all these redundant Activations might seem pointless, but it must be remembered that Manners is not about good rule design; it's purposefully designed as a bad ruleset to fully stress test the cross product matching process and the agenda, which this clearly does. Notice that each Activation has the same time tag of 35, as they were all activated by the change in Context to ASSIGN_SEATS. With OPS5 and LEX it would correctly fire the Activation with the last asserted Seating. With Depth the accumulated fact time tag ensures the Activation with the last asserted Seating fires.

#### 8.11.2.5. Make Path and Path Done

"Make Path" must always fires before "Path Done". A Path is asserted for each Seating arrangement up to the last asserted Seating. Notice that "Path Done" is a subset of "Make Path", so how do we ensure that "Make Path" fires first?

```
rule makePath
    when
        Context( state == Context.MAKE_PATH )
        Seating( seatingId:id, seatingPid:pid, pathDone == false )
        Path( id == seatingPid, pathGuestName:guestName, pathSeat:seat )
        not Path( id == seatingId, guestName == pathGuestName )
    then
        insert( new Path( seatingId, pathSeat, pathGuestName ) );
end
```

```
rule pathDone
    when
        context : Context( state == Context.MAKE_PATH )
        seating : Seating( pathDone == false )
    then
        modify( seating ) {setPathDone( true )}

  modify( context ) {setState( Context.CHECK_DONE)}
end
```

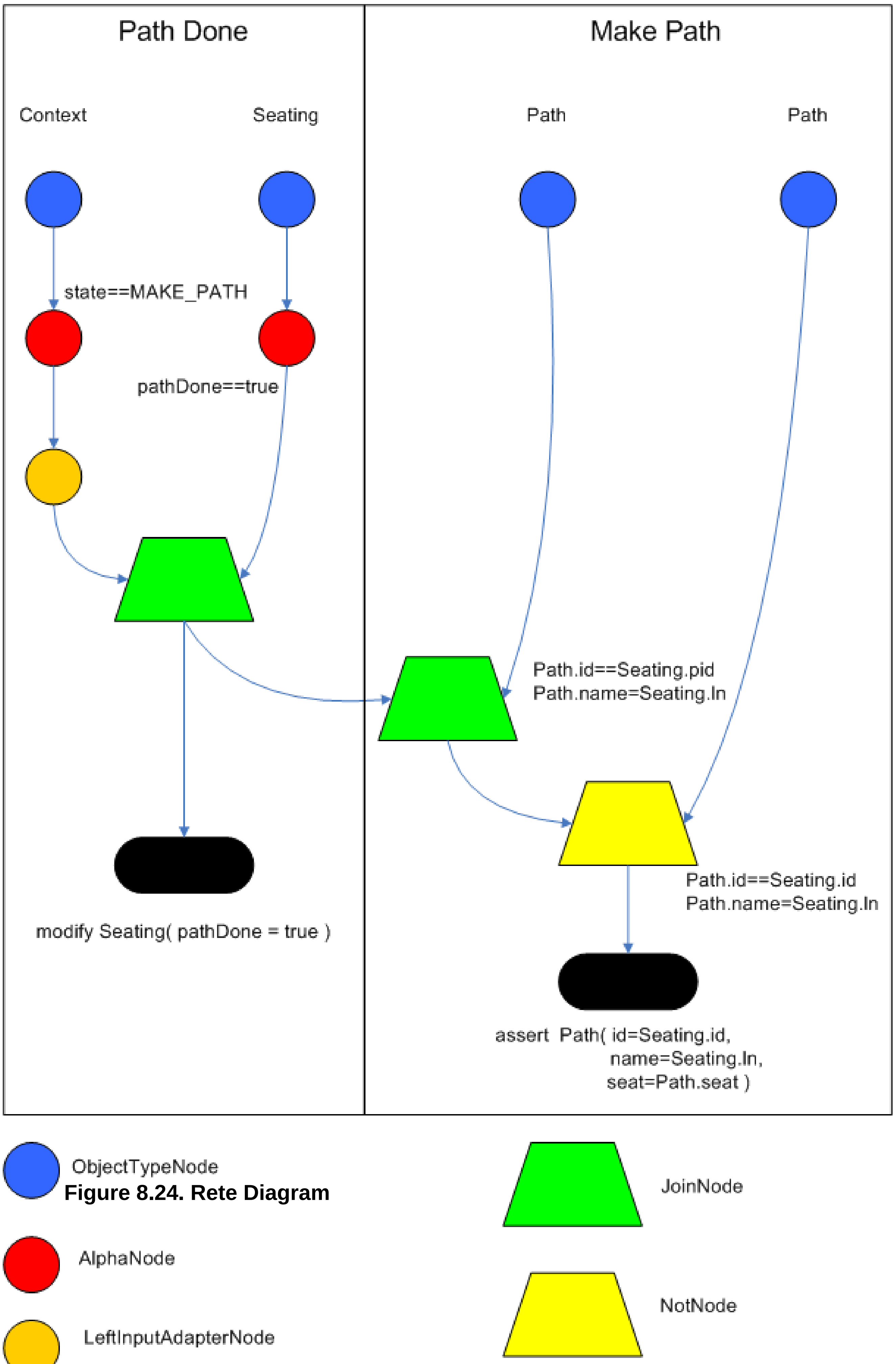


Figure 8.24. Rete Diagram

Both rules end up on the Agenda in conflict and with identical activation time tags, however the accumulate fact time tag is greater for "Make Path" so it gets priority.

### 8.11.2.6. Continue and Are We Done

"Are We Done" only activates when the last seat is assigned, at which point both rules will be activated. For the same reason that "Make Path" always wins over "Path Done" "Are We Done" will take priority over "Continue".

```
rule areWeDone
    when
        context : Context( state == Context.CHECK_DONE )
        LastSeat( lastSeat: seat )
        Seating( rightSeat == lastSeat )
    then
        modify( context ) {setState(Context.PRINT_RESULTS )}
end
```

```
rule continue
    when
        context : Context( state == Context.CHECK_DONE )
    then
        modify( context ) {setState( Context.ASSIGN_SEATS )}
end
```

## 8.11.3. Output Summary

```
Assign First seat
=>[fid:13:13]:[Seating id=1, pid=0, done=true, ls=1, ln=n5, rs=1, rn=n5]
=>[fid:14:14]:[Path id=1, seat=1, guest=n5]

==>[ActivationCreated(16): rule=findSeating
[fid:13:13]:[Seating id=1, pid=0, done=true, ls=1, ln=n5, rs=1, rn=n5]
[fid:9:9]:[Guest name=n5, sex=f, hobbies=h1]
[fid:1:1]:[Guest name=n1, sex=m, hobbies=h1]

==>[ActivationCreated(16): rule=findSeating
[fid:13:13]:[Seating id=1 , pid=0, done=true, ls=1, ln=n5, rs=1, rn=n5]
[fid:9:9]:[Guest name=n5, sex=f, hobbies=h1]
[fid:5:5]:[Guest name=n4, sex=m, hobbies=h1]*

Assign Seating
=>[fid:15:17] :[Seating id=2 , pid=1 , done=false, ls=1, lg=n5, rs=2, rn=n4]
=>[fid:16:18]:[Path id=2, seat=2, guest=n4]
=>[fid:17:19]:[Chosen id=1, name=n4, hobbies=h1]
```

```
=>[ActivationCreated(21): rule=makePath
[fid:15:17] : [Seating id=2, pid=1, done=false, ls=1, ln=n5, rs=2, rn=n4]
[fid:14:14] : [Path id=1, seat=1, guest=n5]*

==>[ActivationCreated(21): rule=pathDone
[Seating id=2, pid=1, done=false, ls=1, ln=n5, rs=2, rn=n4]*

Make Path
=>[fid:18:22:[Path id=2, seat=1, guest=n5]]

Path Done

Continue Process
=>[ActivationCreated(25): rule=findSeating
[fid:15:23]:[Seating id=2, pid=1, done=true, ls=1, ln=n5, rs=2, rn=n4]
[fid:7:7]:[Guest name=n4, sex=f, hobbies=h3]
[fid:4:4] : [Guest name=n3, sex=m, hobbies=h3]*

=>[ActivationCreated(25): rule=findSeating
[fid:15:23]:[Seating id=2, pid=1, done=true, ls=1, ln=n5, rs=2, rn=n4]
[fid:5:5]:[Guest name=n4, sex=m, hobbies=h1]
[fid:2:2]:[Guest name=n2, sex=f, hobbies=h1], [fid:12:20] : [Count value=3]

=>[ActivationCreated(25): rule=findSeating
[fid:13:13]:[Seating id=1, pid=0, done=true, ls=1, ln=n5, rs=1, rn=n5]
[fid:9:9]:[Guest name=n5, sex=f, hobbies=h1]
[fid:1:1]:[Guest name=n1, sex=m, hobbies=h1]

Assign Seating
=>[fid:19:26]:[Seating id=3, pid=2, done=false, ls=2, lnn4, rs=3, rn=n3]]
=>[fid:20:27]:[Path id=3, seat=3, guest=n3]]
=>[fid:21:28]:[Chosen id=2, name=n3, hobbies=h3}]

=>[ActivationCreated(30): rule=makePath
[fid:19:26]:[Seating id=3, pid=2, done=false, ls=2, ln=n4, rs=3, rn=n3]
[fid:18:22]:[Path id=2, seat=1, guest=n5]*

=>[ActivationCreated(30): rule=makePath
[fid:19:26]:[Seating id=3, pid=2, done=false, ls=2, ln=n4, rs=3, rn=n3]
[fid:16:18]:[Path id=2, seat=2, guest=n4]*

=>[ActivationCreated(30): rule=done
[fid:19:26]:[Seating id=3, pid=2, done=false, ls=2, ln=n4, rs=3, rn=n3]*
```

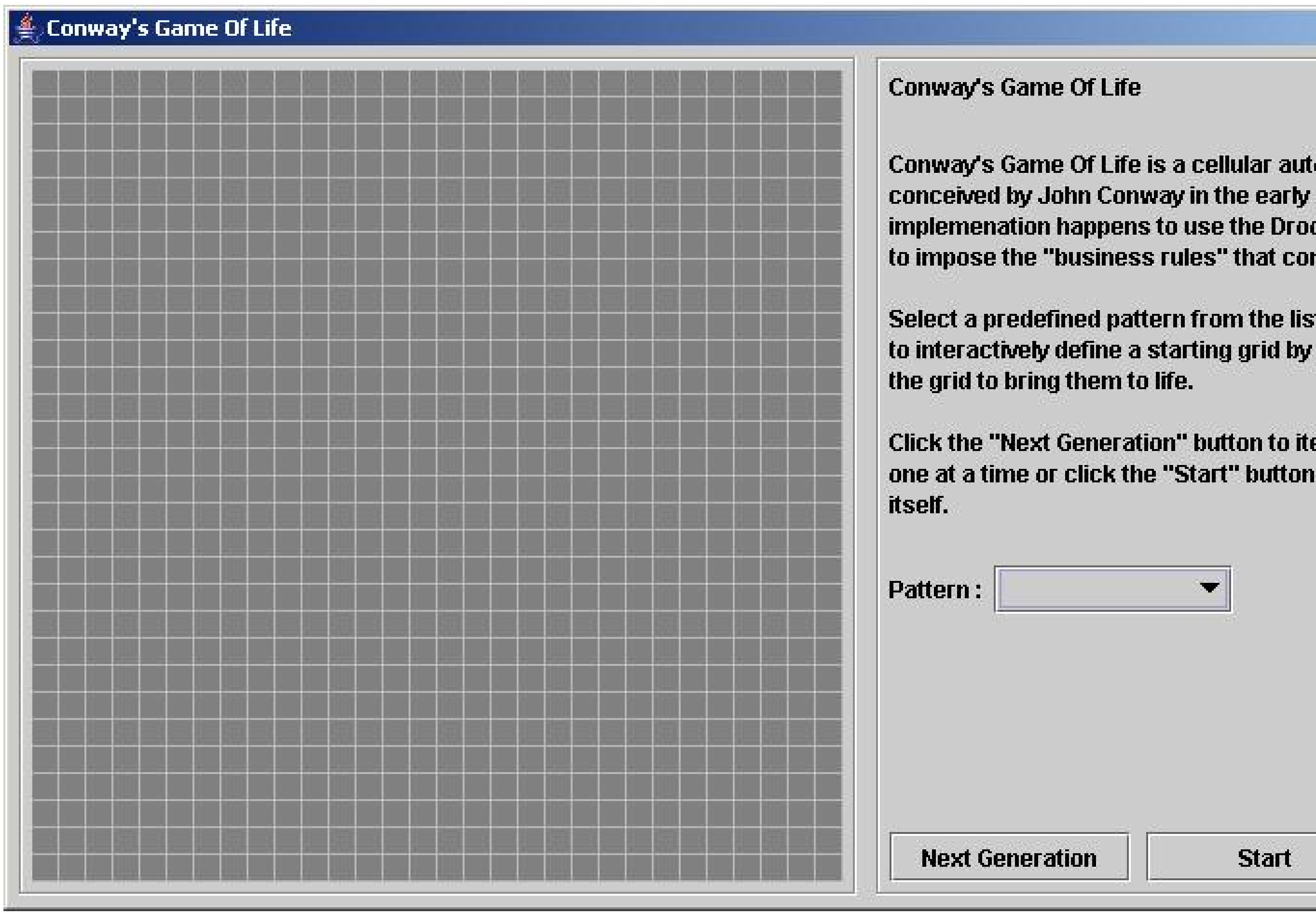
```
Make Path
=>[fid:22:31]:[Path id=3, seat=1, guest=n5]

Make Path
=>[fid:23:32] [Path id=3, seat=2, guest=n4]

Path Done

Continue Processing
=>[ActivationCreated(35): rule=findSeating
[fid:19:33]:[Seating id=3, pid=2, done=true, ls=2, ln=n4, rs=3, rn=n3]
[fid:4:4]:[Guest name=n3, sex=m, hobbies=h3]
[fid:3:3]:[Guest name=n2, sex=f, hobbies=h3], [fid:12:29]*

=>[ActivationCreated(35): rule=findSeating
[fid:15:23]:[Seating id=2, pid=1, done=true, ls=1, ln=n5, rs=2, rn=n4]
[fid:5:5]:[Guest name=n4, sex=m, hobbies=h1]
[fid:2:2]:[Guest name=n2, sex=f, hobbies=h1]

=>[ActivationCreated(35): rule=findSeating
[fid:13:13]:[Seating id=1, pid=0, done=true, ls=1, ln=n5, rs=1, rn=n5]
[fid:9:9]:[Guest name=n5, sex=f, hobbies=h1], [fid:1:1] : [Guest name=n1, sex=m,
 hobbies=h1]

Assign Seating
=>[fid:24:36]:[Seating id=4, pid=3, done=false, ls=3, ln=n3, rs=4, rn=n2]]
=>[fid:25:37]:[Path id=4, seat=4, guest=n2]]
=>[fid:26:38]:[Chosen id=3, name=n2, hobbies=h3]

==>[ActivationCreated(40): rule=makePath
[fid:24:36]:[Seating id=4, pid=3, done=false, ls=3, ln=n3, rs=4, rn=n2]
[fid:23:32]:[Path id=3, seat=2, guest=n4]*

==>[ActivationCreated(40): rule=makePath
[fid:24:36]:[Seating id=4, pid=3, done=false, ls=3, ln=n3, rs=4, rn=n2]
[fid:20:27]:[Path id=3, seat=3, guest=n3]*

=>[ActivationCreated(40): rule=makePath
[fid:24:36]:[Seating id=4, pid=3, done=false, ls=3, ln=n3, rs=4, rn=n2]
[fid:22:31]:[Path id=3, seat=1, guest=n5]*

=>[ActivationCreated(40): rule=done
[fid:24:36]:[Seating id=4, pid=3, done=false, ls=3, ln=n3, rs=4, rn=n2]*
Make Path
=>fid:27:41:[Path id=4, seat=2, guest=n4]

Make Path
=>fid:28:42]:[Path id=4, seat=1, guest=n5]]

Make Path
=>fid:29:43]:[Path id=4, seat=3, guest=n3]]

Path Done

Continue  Processing
=>[ActivationCreated(46): rule=findSeating
[fid:15:23]:[Seating id=2, pid=1, done=true, ls=1, ln=n5, rs=2, rn=n4]
[fid:5:5]:[Guest name=n4, sex=m, hobbies=h1], [fid:2:2]
[Guest name=n2, sex=f, hobbies=h1]

=>[ActivationCreated(46): rule=findSeating
[fid:24:44]:[Seating id=4, pid=3, done=true, ls=3, ln=n3, rs=4, rn=n2]
[fid:2:2]:[Guest name=n2, sex=f, hobbies=h1]
[fid:1:1]:[Guest name=n1, sex=m, hobbies=h1]*

=>[ActivationCreated(46): rule=findSeating
[fid:13:13]:[Seating id=1, pid=0, done=true, ls=1, ln=n5, rs=1, rn=n5]
[fid:9:9]:[Guest name=n5, sex=f, hobbies=h1]
[fid:1:1]:[Guest name=n1, sex=m, hobbies=h1]

Assign Seating
=>[fid:30:47]:[Seating id=5, pid=4, done=false, ls=4, ln=n2, rs=5, rn=n1]
=>[fid:31:48]:[Path id=5, seat=5, guest=n1]
=>[fid:32:49]:[Chosen id=4, name=n1, hobbies=h1]
```

## 8.12. Conways Game Of Life Example

```
Name: Conways Game Of Life
Main class: org.drools.examples.conway.ConwayAgendaGroupRun
 org.drools.examples.conway.ConwayRuleFlowGroupRun
Type: java application
Rules file: conway-ruleflow.drl conway-agendagroup.drl
Objective: Demonstrates 'accumulate', 'collect' and 'from'
```

Conway's Game Of Life, *http://en.wikipedia.org/wiki/Conway's_Game_of_Life* *http://www.math.com/students/wonders/life/life.html*, is a famous cellular automaton conceived in the early 1970's by mathematician John Conway. While the system is well known as "Conway's Game Of Life", it really isn't a game at all. Conway's system is more like a life simulation. Don't be intimidated. The system is terribly simple and terribly interesting. Math and Computer Science students alike have marvelled over Conway's system for more than 30 years now. The application represented here is a Swing based implementation of Conway's Game of Life. The rules that govern the system are implemented as business rules using Drools. This document will explain the rules that drive the simulation and discuss the Drools specific parts of the implementation.

We'll first introduce the grid view, shown below, to help visualisation of the problem; this is where the life simuation takes place. Initially the grid is empty, meaning that there are no live cells in the system; ech cell can be considered "LIVE" or "DEAD", live cells have a green ball in them. Pre-selected patterns of live cells can be selected from the "Pattern" drop down or cells can be doubled-clicked to toggle them between LIVE and DEAD. It's important to understand that each cell is related to it's neighbour cells, which is a core part of the game's rules and will be explained in a moment. Neighbors include not only cells to the left, right, top and bottom but also cells that are connected diagonally. Each cell has a total of 8 neighbors except the 4 corner cells and all of the other cells along the 4 edges. Corner cells have 3 neighbors and other edge cells have 5 neighbors.



Conway's Game Of Life

Conway's Game Of Life is a cellular aut
conceived by John Conway in the early
implemenation happens to use the Droo
to impose the "business rules" that co

Select a predefined pattern from the lis
to interactively define a starting grid by
the grid to bring them to life.

Click the "Next Generation" button to ite
one at a time or click the "Start" button
itself.

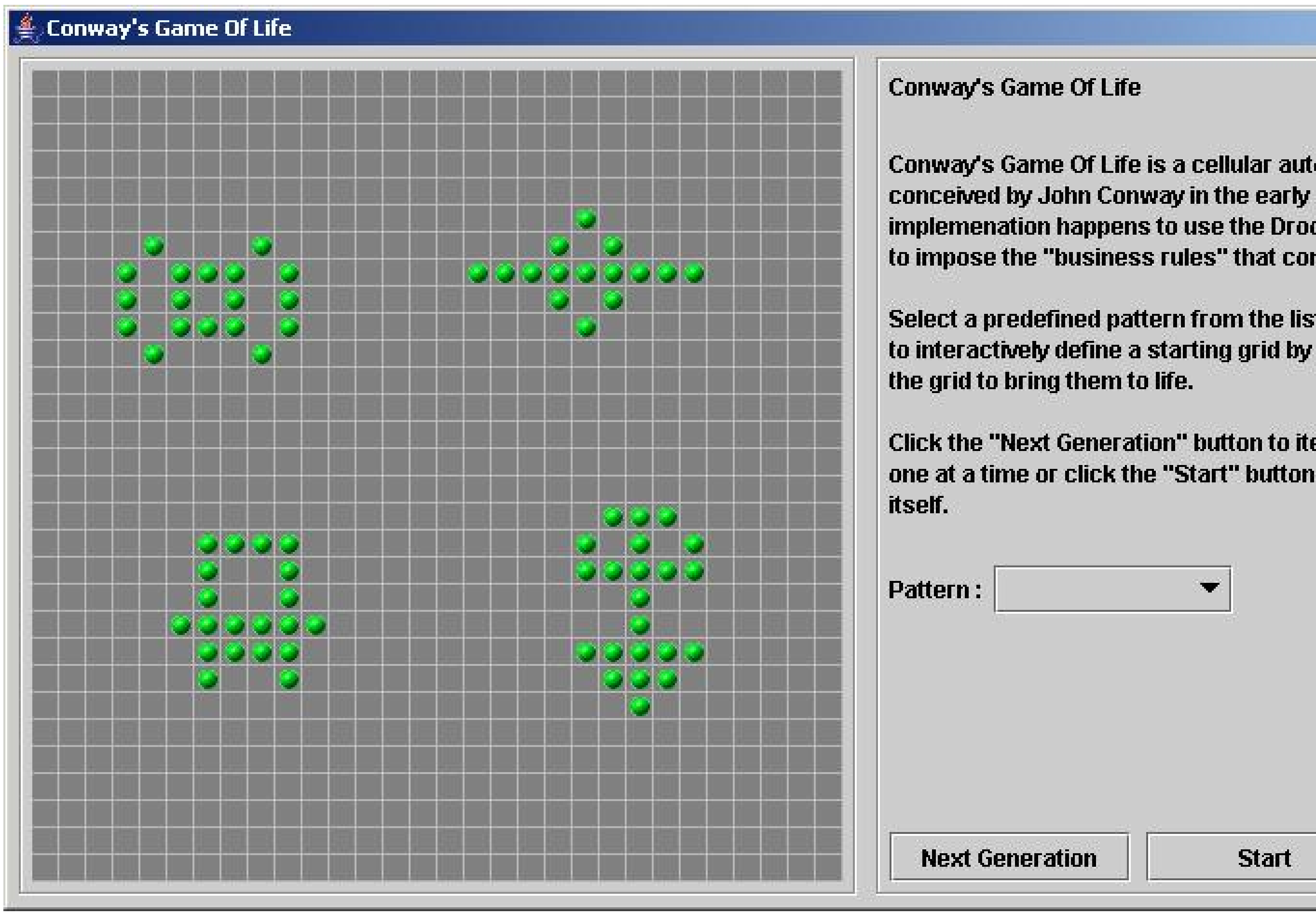
Figure 8.25. Conways Example : Starting a new game

So what are the basic rules that govern this game? Each generation, i.e. completion iteration and evalution of all cells, the system evolves and cells may be born or killed, there are a very simple set of rules that govern what the next generation will look like.

- If a live cell has fewer than 2 live neighbors, it dies of loneliness
- If a live cell has more than 3 live neighbors, it dies from overcrowding
- If a dead cell has exactly 3 live neighbors, it comes to life

That is all there is to it. Any cell that doesn't meet any of those criteria is left as is for the next generation. With those simple rules in mind, go back and play with the system a little bit more and step through some generations one at a time and notice these rules taking their effect.

The screnshot below shows an example generation, with a number of live cells. Don't worry about matching the exact patterns represented in the screen shot. Just get some groups of cells added to the grid. Once you have groups of live cells in the grid, or select a pre-designed pattern, click the "Next Generation" button and notice what happens. Some of the live cells are killed (the green ball disappears) and some dead cells come to life (a green ball appears). Cycle through several generations and see if you notice any patterns. If you click on the "Start" button, the system will evolve itself so you don't need to click the "Next Generation" button over and over. Play with the system a little and then come back here for more details of how the application works.



Conway's Game Of Life

Conway's Game Of Life is a cellular aut
conceived by John Conway in the early
implemenation happens to use the Droo
to impose the "business rules" that cor

Select a predefined pattern from the lis
to interactively define a starting grid by
the grid to bring them to life.

Click the "Next Generation" button to ite
one at a time or click the "Start" button
itself.

### Figure 8.26. Conways Example : A running game

Now lets delve into the code, as this is an advanced example we'll assume that by now you know your way around the Drools framework and able to connect many of the dots, so we'll just focus at a hgh level overview.The example has two ways to execute, one way uses AgendaGroups to manage execution flow the other uses RuleFlowGroups to manage execution flow - so it's a great way to see the differences. - that's ConwayAgendaGroupRun and ConwayRuleFlowGroupRun respectively. For this example I'll cover the ruleflow version, as its what most people will use.

All the Cells are inserted into the session and the rules in the ruleflow-group "register neighbor" are allowed to execute by the ruleflow process. What this group of rules does is for each cell it registers the north east, north, north west and west cells using a Neighbor relation class, notice this relation is bi-drectional which is why we don't have to do any rules for southern facing cells. Note that the constraints make sure we stay one column back from the end and 1 row back from the top. By the time all activations have fired for these rules all cells are related to all their neighboring cells.

### Example 8.73. Conways Example : Register all Cell Neighbour relations

```
rule "register north east"
    ruleflow-group "register neighbor"
```

```
when
    CellGrid( $numberOfColumns : numberOfColumns )
    $cell: Cell( $row : row > 0, $col : col < ( $numberOfColumns - 1 ) )

    $northEast : Cell( row  == ($row - 1), col == ( $col + 1 ) )
then
    insert( new Neighbor( $cell, $northEast ) );
    insert( new Neighbor( $northEast, $cell ) );
end

rule "register north"
    ruleflow-group "register neighbor"
when
    $cell: Cell( $row : row > 0, $col : col )
    $north : Cell( row  == ($row - 1), col == $col )
then
    insert( new Neighbor( $cell, $north ) );
    insert( new Neighbor( $north, $cell ) );
end

rule "register north west"
    ruleflow-group "register neighbor"
when
    $cell: Cell( $row : row > 0, $col : col > 0 )
    $northWest : Cell( row  == ($row - 1), col == ( $col - 1 ) )

then
    insert( new Neighbor( $cell, $northWest ) );
    insert( new Neighbor( $northWest, $cell ) );
end

rule "register west"
    ruleflow-group "register neighbor"
when
    $cell: Cell( $row : row >= 0, $col : col > 0 )
    $west : Cell( row  == $row, col == ( $col - 1 ) )
then
    insert( new Neighbor( $cell, $west ) );
    insert( new Neighbor( $west, $cell ) );
end
```

Once all the cells are inserted some java code applies the pattern to the grid setting certain cells to Live. Then when the user clicks "start" or "next generation" it executes the "Generation" ruleflow. This ruleflow is responsible for the management of all changes of cells in each generation cycle.

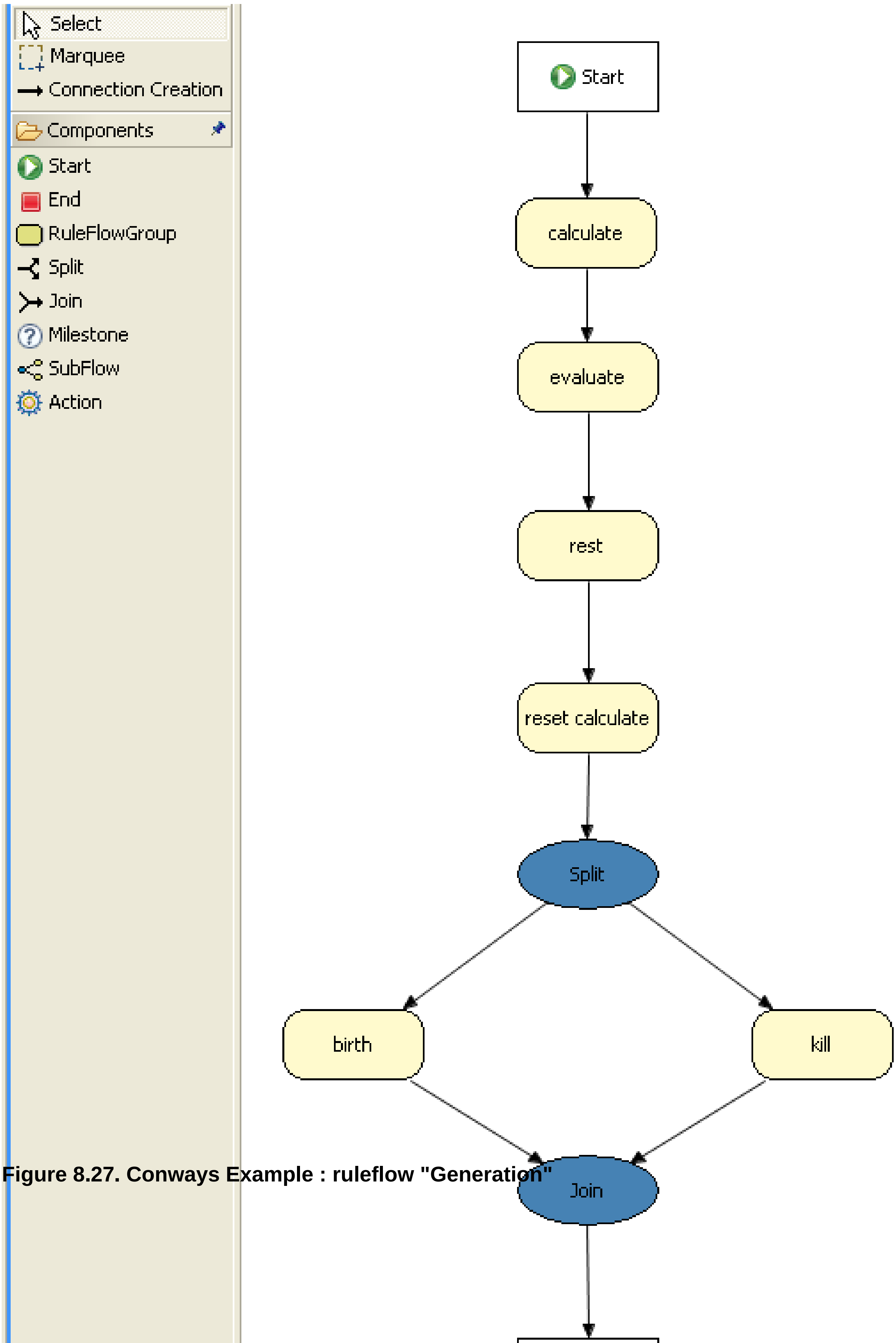


**Figure 8.27. Conways Example : ruleflow "Generation"**

The ruleflow process first enters the "evaluate" group, this means any active rule in that group can fire. The rules in this group apply the main game of life rules discussed in the beginning of the example, where it determines what cells will be killed and which ones given life. We use the "phase" attribute to drives the reasoning of the Cell by specific groups of rules; typical the phase is tied to a RuleFlowGroup. in the ruleflow process definition. Notice that it doesn't actually change the state of any Cells at this point; this is because it's evaluating the Grid in turn and it must complete the full evaluation until those changes can be applied. To achieve this it sets the cell to a "phase" which is either Phase.KILL or Phase.BIRTH, which is used later to control actions applied to the Cell and when.

## Example 8.74. Conways Example : Evaluate Cells with state changes

```
rule "Kill The Lonely"
    ruleflow-group "evaluate"
    no-loop
when
#   A live cell has fewer than 2 live neighbors
    theCell: Cell(liveNeighbors < 2, cellState == CellState.LIVE, phase ==
 Phase.EVALUATE)
then
    theCell.setPhase(Phase.KILL);
    update( theCell );
end

rule "Kill The Overcrowded"
    ruleflow-group "evaluate"
    no-loop
when
#   A live cell has more than 3 live neighbors
    theCell: Cell(liveNeighbors > 3, cellState == CellState.LIVE, phase ==
 Phase.EVALUATE)
then
    theCell.setPhase(Phase.KILL);
    update( theCell );
end

rule "Give Birth"
    ruleflow-group "evaluate"
    no-loop
when
#   A dead cell has 3 live neighbors
    theCell: Cell(liveNeighbors == 3, cellState == CellState.DEAD, phase ==
 Phase.EVALUATE)
then
    theCell.setPhase(Phase.BIRTH);
    update( theCell );
end
```

Once all Cells in the grid have been evaluated we first clear any calculation activations, that occured from any previous data changes, via the "reset calculate" rule, which clears any activations in the "calculate" group. We then enter a split which allows any activations in the "kill" groups and "birth" groups to fire, these rules are responsible for applying the state change.

### Example 8.75. Conways Example : Apply the state changes

```
rule "reset calculate"
    ruleflow-group "reset calculate"
when
then
    WorkingMemory wm = drools.getWorkingMemory();
    wm.clearRuleFlowGroup( "calculate" );
end

rule "kill"
    ruleflow-group "kill"
    no-loop
when
    theCell: Cell(phase == Phase.KILL)
then
    theCell.setCellState(CellState.DEAD);
    theCell.setPhase(Phase.DONE);
    update( theCell );
end

rule "birth"
    ruleflow-group "birth"
    no-loop
when
    theCell: Cell(phase == Phase.BIRTH)
then
    theCell.setCellState(CellState.LIVE);
    theCell.setPhase(Phase.DONE);
    update( theCell );
end
```

At this stage a number of Cells have been modified with the state changed to either LIVE or DEAD, this is where we get to see the power of the Neighbour cell and relational programming. When a cell becomes LIVE or DEAD we use the Neigbor relation drive the iteration over all surrounding Cells increasing or decreasing the LIVE neighbour count, any cell who has their count changed is also set to to the EVALUATE phase, to make sure they are reasoned over duing the evaluate stage of the ruleflow process. Notice that we don't have to do any iteration ourselves, by simpy applying the relations in the rules we can get the rule engine to do all the hard work for us in a minimal amount of code - very nice :) Once the live count for all Cells has been determiend and set the ruleflow process comes to and end; the user can either tell it to evaluate another generation, of if "start" was clicked the engine will start the ruleflow process again.

**Example 8.76. Conways Example : Evaluate Cells with state changes**

```
rule "Calculate Live"
    ruleflow-group "calculate"
    lock-on-active
when
    theCell: Cell(cellState == CellState.LIVE)
    Neighbor(cell == theCell, $neighbor : neighbor)
then
    $neighbor.setLiveNeighbors( $neighbor.getLiveNeighbors() + 1 );
    $neighbor.setPhase( Phase.EVALUATE );
    update( $neighbor );
end

rule "Calculate Dead"
    ruleflow-group "calculate"
    lock-on-active
when
    theCell: Cell(cellState == CellState.DEAD)
    Neighbor(cell == theCell, $neighbor : neighbor )
then
    $neighbor.setLiveNeighbors( $neighbor.getLiveNeighbors() - 1 );
    $neighbor.setPhase( Phase.EVALUATE );
    update( $neighbor );
end
```

# Index