**Drools Introduction**

When we implement a complex software, often we require maintaining a set of rules which will be applied to a set of data for action to be taken on them. In a regular term, we call them rule engine. If there is a small set of rules, we can create our own rule engine that will maintain a certain order and be applied on incoming data to take the decision or categorize the data.

The advantage of maintaining one's own rule engine is that we have a pure control over the rule engine's algorithm. This means that we can easily change the algorithm logic to be simple; we don’t have to rely on the third party for the pattern matching logic.

On the other hand, if the rules are ever-changing or there is a huge set of rules, we will probably not want to maintain our own rule engine as it increases our development cost. Who wants to take this responsibility?

It would be nice if we could delegate that work to some third party that is tested and trusted so we could clearly separate the data and logic.

The third party maintains the rule engine application and we just define the rules as a strategy. It would be nice if it were declarative so that business analysts could understand the logic.

**What Is Drools?**

Drools is a Business Logic integration Platform (BLiP). Drools is an open-source project written in Java. Red Hat and JBoss maintain Drools.

Drools has two main parts:

**1. Authoring**

By authoring, we create a rules file for Drools (.drl). This file contains the rule definition in a declarative way. In this file, we can write a set of rules that will be fired at the run time. It is the developer's responsibility to write these rules as per business requirements.

**2. Runtime**

With the runtime, we create a working memory. It is the same as a session in Hibernate. As a rules file contains a set of rules, the runtime creates memory load. These rules and apply to the incoming data. In Drools, we called incoming data as facts.

**Rule:**A rule is nothing but the logic that will be applied to incoming data. It has two main parts: when and then.   
**When:**Determines the condition on which the Rule will be fired.   
**Then:**The action; if the rules met the condition, they defines what work this rule performs.

This is the syntax:

Rule <Rule Name>

when

<condition>

then

<Action>

End

In this example, we will greet a person based on current time. We will define the rules in Drools files. Drool will load these rules and fire on the incoming data.

**Step 1:**Create a.drl (droolRule.drl)file where we will define the rules.

package com.rules

import com.example.droolsExample.Person

rule "Good Morning"

when

person: Person(time >= 0, time < 12)

then

person.setGreet("Good Morning " + person.getName());

end

rule "Good Afternoon"

when

person: Person(time >= 12, time < 16)

then

person.setGreet("Good Afternoon " + person.getName());

end

rule "Good Night"

when

person: Person(time >= 16, time <= 24)

then

person.setGreet("Good Night " + person.getName());

end

Please note that here we create three rules: “Good Morning”, “Good Afternoon “ and “Good Night.” In the **When** section, we check the current time based on the Person POJO’s time property. In the **Then** section, we set the greeting messages accordingly.

**Step 2:**Create Person POJO class.

package com.example.droolsExample;

public class Person {

private String name;

private int time;

private String greet;

public String getGreet() {

return greet;

}

public void setGreet(String greet) {

this.greet = greet;

}

public String getName() {

return name;

}

public void setName(String name) {

this.name = name;

}

public int getTime() {

return time;

}

public void setTime(int time) {

this.time = time;

}

}

**Step 3:** We create a class named DroolsTest.java.

3a. Load the rule file (i.e., droolsTest.drl) by using InputStream.

3b. Create a package using the above rule and add them into drools PackageBuilder.

3c. Create a RuleBase by using the above Package. Rulebase is the same as Sessionfactory in Hibernate; it is costly.

3d. Create a working memory from this RuleBase. It is same as Session class in Hibernate. This working memory manages the rules and incoming data. Apply the rules on the data.

3e. Add incoming data into working memory. Here, we create a Person Object and add it into Working Memory

3f. Fire all rules.

DroolsTest.java should look like the following:

package com.example.droolsExample;

import java.io.IOException;

import java.io.InputStream;

import java.io.InputStreamReader;

import java.io.Reader;

import org.drools.compiler.compiler.DroolsParserException;

import org.drools.compiler.compiler.PackageBuilder;

import org.drools.core.RuleBase;

import org.drools.core.RuleBaseFactory;

import org.drools.core.WorkingMemory;

public class DroolsTest {

public static void main(String[] args) throws DroolsParserException,

IOException {

DroolsTest droolsTest = new DroolsTest();

droolsTest.executeDrools();

}

public void executeDrools() throws DroolsParserException, IOException {

PackageBuilder packageBuilder = new PackageBuilder();

String ruleFile = "/com/rules/droolsRule.drl";

InputStream resourceAsStream = getClass().getResourceAsStream(ruleFile);

Reader reader = new InputStreamReader(resourceAsStream);

packageBuilder.addPackageFromDrl(reader);

org.drools.core.rule.Package rulesPackage = packageBuilder.getPackage();

RuleBase ruleBase = RuleBaseFactory.newRuleBase();

ruleBase.addPackage(rulesPackage);

WorkingMemory workingMemory = ruleBase.newStatefulSession();

Person person = new Person();

person.setName("Shamik Mitra");

person.setTime(7);

workingMemory.insert(person);

workingMemory.fireAllRules();

System.out.println(person.getGreet());

}

Output:

Good Morning Shamik Mitra.

We set the time for 7 a.m., so it satisfies the Good Morning Rule condition and fires this rule.

**Chapter 9. Examples**

[9.1. Getting the Examples](https://docs.jboss.org/drools/release/5.2.0.Final/drools-expert-docs/html/ch09.html#d0e8277)

[9.2. Hello World](https://docs.jboss.org/drools/release/5.2.0.Final/drools-expert-docs/html/ch09.html#d0e8284)

[9.3. State Example](https://docs.jboss.org/drools/release/5.2.0.Final/drools-expert-docs/html/ch09.html#d0e8552)

[9.3.1. Understanding the State Example](https://docs.jboss.org/drools/release/5.2.0.Final/drools-expert-docs/html/ch09.html#d0e8557)

[9.4. Fibonacci Example](https://docs.jboss.org/drools/release/5.2.0.Final/drools-expert-docs/html/ch09.html#d0e8838)

[9.5. Banking Tutorial](https://docs.jboss.org/drools/release/5.2.0.Final/drools-expert-docs/html/ch09.html#d0e9044)

[9.6. Pricing Rule Decision Table Example](https://docs.jboss.org/drools/release/5.2.0.Final/drools-expert-docs/html/ch09.html#d0e9371)

[9.6.1. Executing the example](https://docs.jboss.org/drools/release/5.2.0.Final/drools-expert-docs/html/ch09.html#d0e9395)

[9.6.2. The decision table](https://docs.jboss.org/drools/release/5.2.0.Final/drools-expert-docs/html/ch09.html#d0e9437)

[9.7. Pet Store Example](https://docs.jboss.org/drools/release/5.2.0.Final/drools-expert-docs/html/ch09.html#d0e9502)

[9.8. Honest Politician Example](https://docs.jboss.org/drools/release/5.2.0.Final/drools-expert-docs/html/ch09.html#d0e10064)

[9.9. Sudoku Example](https://docs.jboss.org/drools/release/5.2.0.Final/drools-expert-docs/html/ch09.html#d0e10193)

[9.9.1. Sudoku Overview](https://docs.jboss.org/drools/release/5.2.0.Final/drools-expert-docs/html/ch09.html#d0e10214)

[9.9.2. Running the Example](https://docs.jboss.org/drools/release/5.2.0.Final/drools-expert-docs/html/ch09.html#d0e10228)

[9.9.3. Java Source and Rules Overview](https://docs.jboss.org/drools/release/5.2.0.Final/drools-expert-docs/html/ch09.html#d0e10295)

[9.9.4. Sudoku Validator Rules (validate.drl)](https://docs.jboss.org/drools/release/5.2.0.Final/drools-expert-docs/html/ch09.html#d0e10425)

[9.9.5. Sudoku Solving Rules (sudoku.drl)](https://docs.jboss.org/drools/release/5.2.0.Final/drools-expert-docs/html/ch09.html#d0e10434)

[9.10. Number Guess](https://docs.jboss.org/drools/release/5.2.0.Final/drools-expert-docs/html/ch09.html#d0e10470)

[9.11. Miss Manners and Benchmarking](https://docs.jboss.org/drools/release/5.2.0.Final/drools-expert-docs/html/ch09.html#d0e10761)

[9.11.1. Introduction](https://docs.jboss.org/drools/release/5.2.0.Final/drools-expert-docs/html/ch09.html#d0e10783)

[9.11.2. Indepth Discussion](https://docs.jboss.org/drools/release/5.2.0.Final/drools-expert-docs/html/ch09.html#d0e10869)

[9.11.3. Output Summary](https://docs.jboss.org/drools/release/5.2.0.Final/drools-expert-docs/html/ch09.html#d0e11079)

[9.12. Conway's Game Of Life](https://docs.jboss.org/drools/release/5.2.0.Final/drools-expert-docs/html/ch09.html#d0e11134)

**9.1. Getting the Examples**

Make sure the Drools Eclipse plugin is installed, which needs the Graphical Editing Framework (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 set up the dependencies by hand, of course. Many, but not all of the examples are documented below, enjoy!

Some examples require Java 1.6 to run.

**9.2. Hello World**

**Name:** Hello World

**Main class:** org.drools.examples.helloworld.HelloWorldExample

**Module:** drools-examples

**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.

This example demonstrates how to build Knowledge Bases and Sessions. Also, audit logging and debug outputs are shown, which is ommitted from other examples as it's all very similar. A KnowledgeBuilder is used to turn a DRL source file into Package objects which the Knowledge Base can consume. The add method takes a Resource interface and a Resource Type as parameters. The Resource can be used to retrieve a DRL source file from various locations; in this case the DRL file is being retrieved from the classpath using aResourceFactory, but it could come from a disk file or a URL. Here, we only add a single DRL source file, but multiple DRL files can be added. Also, DRL files with different namespaces can be added, where the Knowledge Builder creates a package for each namespace. Multiple packages of different namespaces can be added to the same Knowledge Base. When all the DRL files have been added, we should check the builder for errors. While the Knowledge Base 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 KnowledgeBuilder instance. Once you know the builder is error free, get the Package collection, instantiate a KnowledgeBase from the KnowledgeBaseFactory and add the package collection.

**Example 9.1. HelloWorld: 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 System.err stream displayed in the Console window. Adding listeners to a Session is trivial, as shown below. The KnowledgeRuntimeLogger provides execution auditing, the result of which can be viewed in a graphical viewer. The logger is 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 9.2. HelloWorld: 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 one of the two integers HELLO or GOODBYE.

**Example 9.3. 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 text "Hello World" and the status HELLO and then inserted into the engine, at which point fireAllRules() is executed. Remember that all the network evaluation is done during the insert time, so that by the time the program execution reaches the fireAllRules() method call the engine already knows which rules are fully matches and able to fire.

**Example 9.4. HelloWorld: Execution**

**final** Message message = **new** Message();

message.setMessage("Hello World");

message.setStatus(Message.HELLO);

ksession.insert(message);

ksession.fireAllRules();

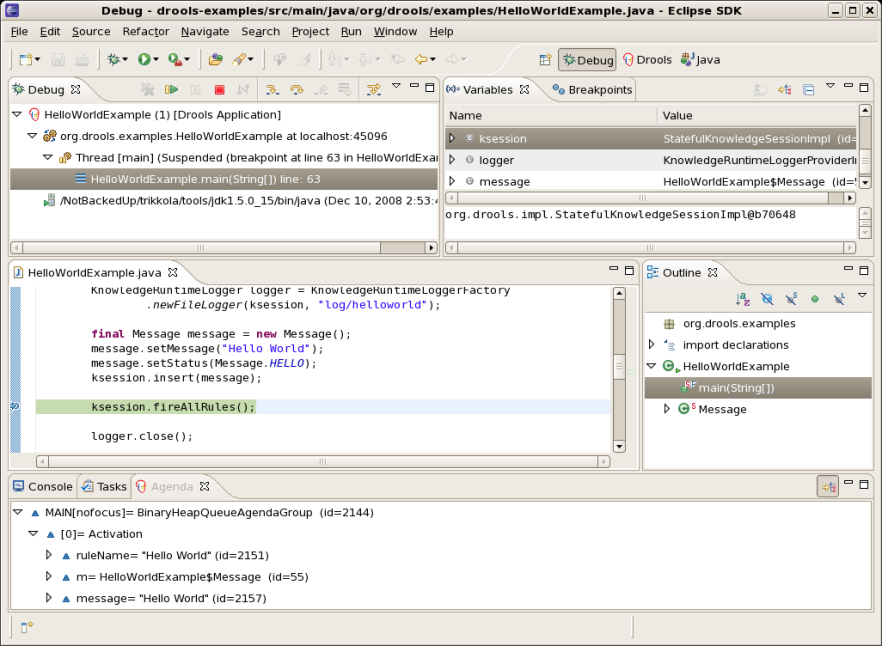
logger.close();

ksession.dispose();

To execute the example as a Java application:

1. Open the class org.drools.examples.helloworld.HelloWorldExample in your Eclipse IDE
2. Right-click the class and select "Run as..." and then "Java application"

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



**Figure 9.1. 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 9.5. HelloWorld: System.out in the Console window**

Hello World

Goodbye cruel world

**Example 9.6. HelloWorld: System.err in the Console window**

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

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

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

object=org.drools.examples.helloworld.HelloWorldExample$Message@17cec96]

[BeforeActivationFired: rule=Hello World;

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

==>[ActivationCreated(4): rule=Good Bye;

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

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

old\_object=org.drools.examples.helloworld.HelloWorldExample$Message@17cec96;

new\_object=org.drools.examples.helloworld.HelloWorldExample$Message@17cec96]

[AfterActivationFired(0): rule=Hello World]

[BeforeActivationFired: rule=Good Bye;

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

[AfterActivationFired(4): rule=Good Bye]

The LHS (after 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 bindings are created: the variable message is bound to the message attribute and the variable m is bound to the matched Message object itself.

The RHS (after then) or consequence part of the rule is written using the MVEL expression language, as declared by the rule's attribute dialect. After printing the content of the bound variable message to System.out, the rule changes the values of the message and status attributes of the Message object bound to m. This is done MVEL's modify statement, which allows you to apply a block of assignments in one statement, with the engine being automatically notified of the changes at the end of the block.

**Example 9.7. HelloWorld: 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",

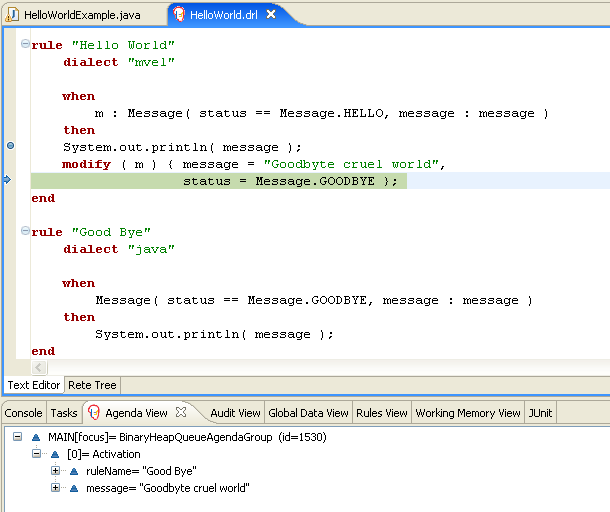
status = Message.GOODBYE };

end

We can set a breakpoint into the DRL, on the modify call, and inspect the Agenda view again during the execution of the rule's consequence. This time we start the execution via "Debug As" and "Drools application" and not by running a "Java application":

1. Open the class org.drools.examples.HelloWorld in your Eclipse IDE.
2. Right-click the class and select "Debug as..." and then "Drools application".

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



**Figure 9.2. Hello World: rule "Hello World" Agenda View**

The "Good Bye" rule, which specifies the "java" dialect, is similar to the "Hello World" rule except that it matches Message objects whose status is Message.GOODBYE.

**Example 9.8. HelloWorld: rule "Good Bye"**

rule "Good Bye"

dialect "java"

when

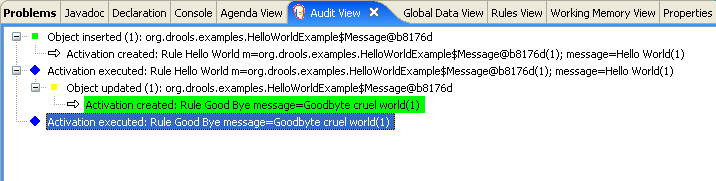
Message( status == Message.GOODBYE, message : message )

then

System.out.println( message );

end

Remember the Java code where we used the KnowledgeRuntimeLoggerFactory method 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 demostrate the example execution flow. In the view screen shot below we can see that the object is inserted, which creates an activation for the "Hello World" rule; the activation is then executed which updates the Message object causing the "Good Bye" rule to activate; finally the "Good Bye" rule also executes. Selecting an event in the Audit view highlights the origin event in green; therefore the "Activation created" event is highlighted in green as the origin of the "Activation executed" event.



**Figure 9.3. Hello World: Audit View**

**9.3. State Example**

This example is implemented in three different versions to demonstrate different ways of implementing the same basic behavior: 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.

**9.3.1. Understanding the State Example**

**Name:** State Example

**Main class:** org.drools.examples.state.StateExampleUsingSalience

**Module:** drools-examples

**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 the class org.drools.examples.state.State). The two possible states for each objects are:

* NOTRUN
* FINISHED

**Example 9.9. 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...

}

Ignoring the PropertyChangeSupport, which will be explained later, we see the creation of four State objects named A, B, C and D. Initially their states are set to NOTRUN, which is default for the used constructor. Each instance is asserted in turn into the Session and then fireAllRules() is called.

**Example 9.10. Salience State: 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 modify or update().

boolean dynamic = true;

session.insert( a, dynamic );

session.insert( b, dynamic );

session.insert( c, dynamic );

session.insert( d, dynamic );

session.fireAllRules();

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

To execute the application:

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

You will see the following output in the Eclipse console window:

**Example 9.11. Salience State: Console Output**

A finished

B finished

C finished

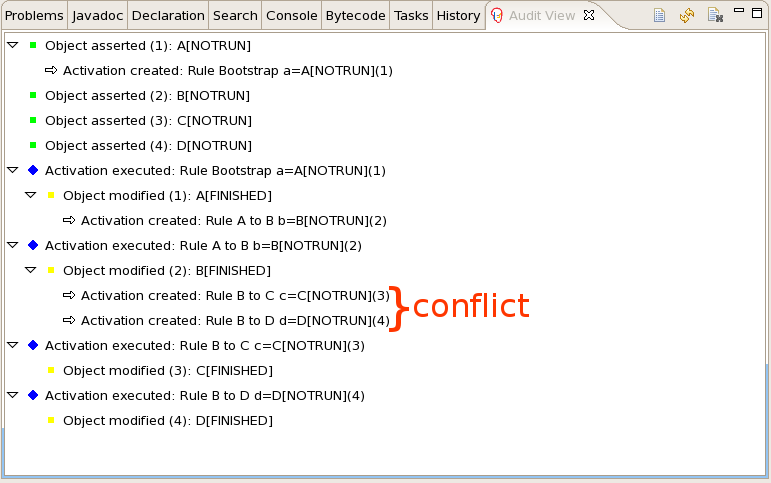
D finished

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

The best way to understand what is happening is to use the Audit Logging feature to graphically see the results of each operation. To view the Audit log generated by a run of this example:

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

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



**Figure 9.4. Salience State Example Audit View**

Reading the log in the "Audit View", top to bottom, we see every action and the corresponding changes in the Working Memory. This way we observe that the assertion of the State object A in the state NOTRUNactivates the Bootstrap rule, while the assertions of the other State objects have no immediate effect.

**Example 9.12. Salience State: 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 rule Bootstrap changes the state of A to FINISHED, which, in turn, activates rule "A to B".

**Example 9.13. Salience State: Rule "A to B"**

rule "A to B"

when

State(name == "A", state == State.FINISHED )

b : State(name == "B", state == State.NOTRUN )

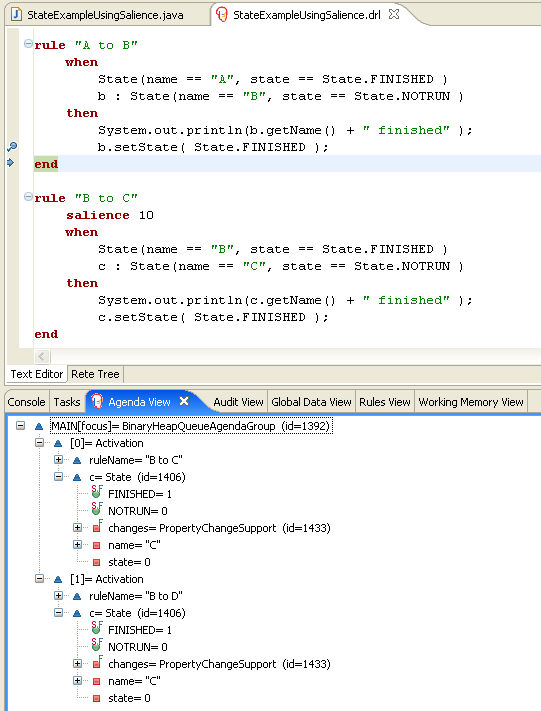
then

System.out.println(b.getName() + " finished" );

b.setState( State.FINISHED );

end

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



**Figure 9.5. State Example Agenda View**

**Example 9.14. Salience State: 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

Rule "B to D" fires last, modifying object D to state FINISHED.

**Example 9.15. Salience State: 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*, based on PropertyChangeListenerobjects. As described in the documentation, in order for the engine to see and react to changes of fact properties, the application must tell the engine that changes occurred. This can be done explicitly in the rules by using the modify statement, 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 modify statements in the rules. To make use of this feature, ensure that your facts implement PropertyChangeSupport, the same way the classorg.drools.example.State does, and use the following code to insert the facts into the Working Memory:

**Example 9.16. Inserting a Dynamic Fact**

// By setting dynamic to TRUE, Drools will use JavaBean

// PropertyChangeListeners so you don't have to call modify or update().

**final** boolean dynamic = true;

session.insert( fact, dynamic );

When using PropertyChangeListener objects, each setter must implement a little extra code for the notification. Here is the setter for state in the class org.drools.examples:

:

**Example 9.17. 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 classes in this example: 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. StateExampleWithDynamicRules shows how an additional rule can be added to an already running Working Memory with all the existing data applying to it at runtime.

Agenda groups are a way to partition the Agenda into groups and to control which groups can execute. By default, all rules are in the agenda group "MAIN". The "agenda-group" attribute lets you specify a different agenda group for the rule. Initially, a Working Memory has its focus on the Agenda group "MAIN". A group's rules will only fire when the group receives the focus. This can be achieved either ny using the method by setFocus() or the rule attribute auto-focus. "auto-focus" means that the rule automatically sets the focus to its agenda group when the rule is matched and activated. It is this "auto-focus" that enables rule "B to C" to fire before "B to D".

**Example 9.18. Agenda Group State Example: Rule "B to C"**

rule "B to C"

agenda-group "B to C"

auto-focus true

when

State(name == "B", state == State.FINISHED )

c : State(name == "C", state == State.NOTRUN )

then

System.out.println(c.getName() + " finished" );

c.setState( State.FINISHED );

kcontext.getKnowledgeRuntime().getAgenda().getAgendaGroup( "B to D" ).setFocus();

end

The rule "B to C" calls setFocus() on the agenda group "B to D", allowing its active rules to fire, which allows the rule "B to D" to fire.

**Example 9.19. 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 Rule Base after fireAllRules(). The added rule is just another state transition.

**Example 9.20. 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

This produces the following expected output:

**Example 9.21. Dynamic Sate Example Output**

A finished

B finished

C finished

D finished

E finished

**9.4. Fibonacci Example**

**Name:** Fibonacci

**Main class:** org.drools.examples.fibonacci.FibonacciExample

**Module:** drools-examples

**Type:** Java application

**Rules file:** Fibonacci.drl

**Objective:** Demonstrates Recursion,

the CE not and cross product matching

The Fibonacci Numbers (see <http://en.wikipedia.org/wiki/Fibonacci_number>) discovered by Leonardo of Pisa (see <http://en.wikipedia.org/wiki/Fibonacci>) is a sequence that starts with 0 and 1. The next Fibonacci number is obtained by adding the two preceding Fibonacci numbers. The Fibonacci sequence begins with 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987, 1597, 2584, 4181, 6765, 10946,... The Fibonacci Example demonstrates recursion and conflict resolution with salience values.

The single fact class Fibonacci is used in this example. It has two fields, sequence and value. The sequence field is used to indicate the position of the object in the Fibonacci number sequence. The value field shows the value of that Fibonacci object for that sequence position, using -1 to indicate a value that still needs to be computed.

**Example 9.22. Fibonacci Class**

**public** **static** **class** Fibonacci {

    **private** int  sequence;

    **private** long value;

    **public** Fibonacci( **final** int sequence ) {

        **this**.sequence = sequence;

        **this**.value = -1;

    }

    ... setters and getters go here...

}

Execute the example:

1. Open the class org.drools.examples.fibonacci.FibonacciExample in your Eclipse IDE.
2. Right-click the class and select "Run as..." and then "Java application"

Eclipse shows the following output in its console window (with "...snip..." indicating lines that were removed to save space):

**Example 9.23. 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 field of 50. A recursive rule is then used to insert the other 49 Fibonacci objects. This example doesn't use PropertyChangeSupport. It uses the MVEL dialect, which means we can use the modify keyword, which allows a block setter action which also notifies the engine of changes.

**Example 9.24. Fibonacci Example: Execution**

ksession.insert( **new** Fibonacci( 50 ) );

ksession.fireAllRules();

The rule Recurse is very simple. It matches each asserted Fibonacci object with a value of -1, creating and asserting a new Fibonacci object with a sequence of one less than the currently matched object. Each time a Fibonacci object is added while the one with a sequence field equal to 1 does not exist, the rule re-matches and fires again. The not conditional element is used to stop the rule's matching once we have all 50 Fibonacci objects in memory. The rule also has a salience value, because we need to have all 50 Fibonacciobjects asserted before we execute the Bootstrap rule.

**Example 9.25. Fibonacci Example: Rule "Recurse"**

rule Recurse

salience 10

when

f : Fibonacci ( value == -1 )

not ( Fibonacci ( sequence == 1 ) )

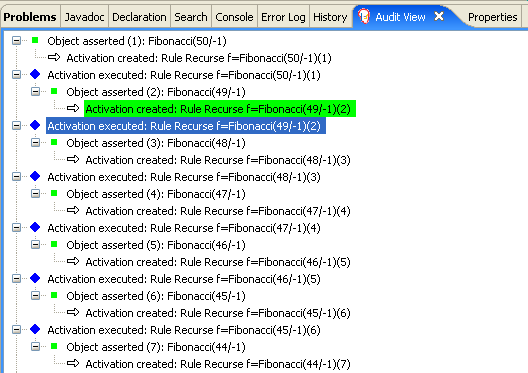
then

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 field of 50, done from Java code. From there on, the Audit view shows the continual recursion of the rule, where each asserted Fibonacci object causes the Recurse rule to become activated and to fire again.



**Figure 9.6. Fibonacci Example: "Recurse" Audit View 1**

When a Fibonacci object with a sequence field of 2 is asserted the "Bootstrap" rule is matched and activated along with the "Recurse" rule. Note the multi-restriction on field sequence, testing for equality with 1 or 2.

**Example 9.26. Fibonacci Example: Rule "Bootstrap"**

rule Bootstrap

when

f : Fibonacci( sequence == 1 || == 2, value == -1 ) // multi-restriction

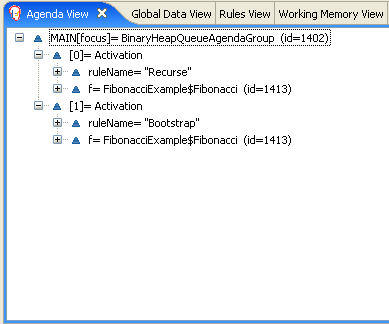
then

modify ( f ){ value = 1 };

System.out.println( f.sequence + " == " + f.value );

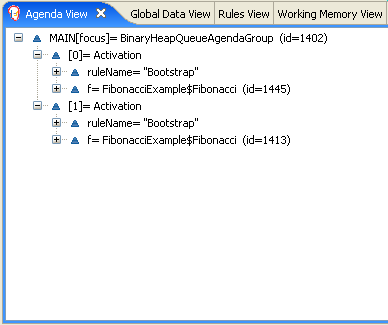
end

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



**Figure 9.7. Fibonacci Example: "Recurse" Agenda View 1**

When a Fibonacci object 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 notconditional element stops the rule's matching as soon as a Fibonacci object with a sequence of 1 exists.



**Figure 9.8. Fibonacci Example: "Recurse" Agenda View 2**

Once we have two Fibonacci objects with values not equal to -1 the "Calculate" rule is able to match. It was the "Bootstrap" rule that set the objects with sequence 1 and 2 to values of 1. At this point we have 50 Fibonacci objects in the Working Memory. Now we need to select a suitable triple to calculate each of their values in turn. Using three Fibonacci patterns in a rule without field constraints to confine the possible cross products would result in 50x49x48 possible combinations, leading to about 125,000 possible rule firings, most of them incorrect. The "Calculate" rule uses field constraints to correctly constraint the thee Fibonacci patterns 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 this, too, but it adds an additional field constraint to ensure that its sequence is greater by one than the Fibonacci bound to f1. When this rule fires for the first time, 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 sequence 2. The final pattern finds the Fibonacci with a value equal to -1 and 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 calculate the value for the third Fibonacci object that's bound to f3.

**Example 9.27. Fibonacci Example: Rule "Calculate"**

rule Calculate

when

// Bind f1 and s1

f1 : Fibonacci( s1 : sequence, value != -1 )

// Bind f2 and v2; refer to bound variable s1

f2 : Fibonacci( sequence == (s1 + 1), v2 : value != -1 )

// Bind f3 and s3; alternative reference of f2.sequence

f3 : Fibonacci( s3 : sequence == (f2.sequence + 1 ), value == -1 )

then

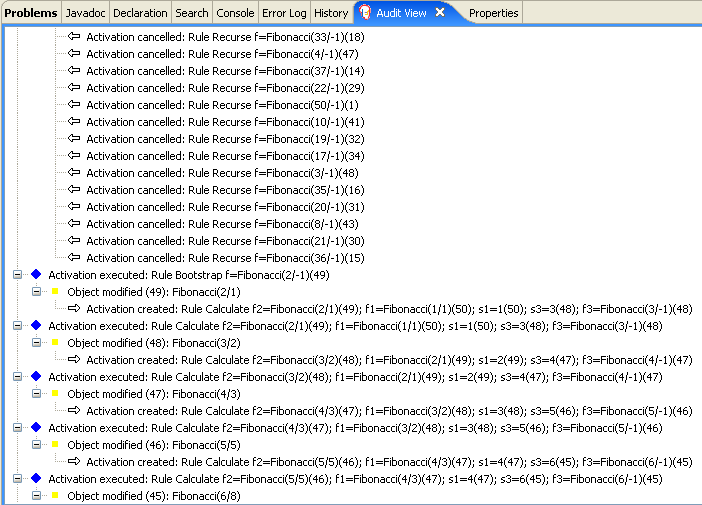
// Note the various referencing rechniques.

modify ( f3 ) { value = f1.value + v2 };

System.out.println( s3 + " == " + f3.value );

end

The modify statement updated the value of the Fibonacci object bound to f3. This means we now have another new Fibonacci object with a value not equal to -1, which allows the "Calculate" rule to rematch and calculate the next Fibonacci number. The Audit view below shows 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 match again. This continues till the value is set for all Fibonacciobjects.



**Figure 9.9. Fibonacci Example: "Bootstrap" Audit View**

**9.5. Banking Tutorial**

**Name:** BankingTutorial

**Main class:** org.drools.tutorials.banking.BankingExamplesApp.java

**Module:** drools-examples

**Type:** Java application

**Rules file:** org.drools.tutorials.banking.\*.drl

**Objective:** Demonstrate pattern matching, basic sorting and calculation rules.

This tutorial demonstrates the process of developing a complete personal banking application to handle credits and debits on multiple accounts. It uses a set of design patterns that have been created for the process.

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

**Example 9.28. Banking Tutorial: RuleRunner**

**public** **class** RuleRunner {

    **public** RuleRunner() {

    }

    **public** void runRules(String[] rules,

                         Object[] facts) **throws** Exception {

        KnowledgeBase kbase = KnowledgeBaseFactory.newKnowledgeBase();

        KnowledgeBuilder kbuilder = KnowledgeBuilderFactory.newKnowledgeBuilder();

        **for** ( int i = 0; i < rules.length; i++ ) {

            String ruleFile = rules[i];

            System.out.println( "Loading file: " + ruleFile );

            kbuilder.add( ResourceFactory.newClassPathResource( ruleFile,

                                                                RuleRunner.**class** ),

                          ResourceType.DRL );

        }

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

        kbase.addKnowledgePackages( pkgs );

        StatefulKnowledgeSession ksession = kbase.newStatefulKnowledgeSession();

        **for** ( int i = 0; i < facts.length; i++ ) {

            Object fact = facts[i];

            System.out.println( "Inserting fact: " + fact );

            ksession.insert( fact );

        }

        ksession.fireAllRules();

    }

}

The first of our sample Java classes loads and executes a single DRL file, **Example.drl**, but without inserting any data.

**Example 9.29. 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] );

    }

}

The first simple rule to execute has a single eval condition that will alway be true, so that this rule will match and fire, once, after the start.

**Example 9.30. Banking Tutorial: Rule in Example1.drl**

rule "Rule 01"

when

eval( 1==1 )

then

System.out.println( "Rule 01 Works" );

endh

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

**Example 9.31. Banking Tutorial: Output of Example1.java**

Loading file: Example1.drl

Rule 01 Works

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

**Example 9.32. 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 objects. This is not considered "best practice" as a number is not a useful fact, but we use it here to demonstrate basic techniques before more complexity is added.

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

**Example 9.33. Banking Tutorial: Rule in Example2.drl**

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 Number objects and prints out the values. Notice the use of the abstract class Number: we inserted Integer objects but we now look for any kind of number. The pattern matching engine is able to match interfaces and superclasses of 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 9.34. Banking Tutorial: Output of Example2.java**

Loading file: Example2.drl

Inserting fact: 3

Inserting fact: 1

Inserting fact: 4

Inserting fact: 1

Inserting fact: 5

Number found with value: 5

Number found with value: 1

Number found with value: 4

Number found with value: 1

Number found with value: 3

There are certainly many better ways to sort numbers than using rules, but since we will need to apply some cashflows in date order when we start looking at banking rules we'll develop simple rule based sorting technique.

**Example 9.35. Banking Tutorial: Example3.java**

**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 Integer objects, but this time the rule is slightly different:

**Example 9.36. Banking Tutorial: Rule in Example3.drl**

rule "Rule 03"

when

$number : Number( )

not Number( intValue < $number.intValue )

then

System.out.println("Number found with value: " + $number.intValue() );

retract( $number );

end

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

The resulting output shows that the numbers are now sorted numerically.

**Example 9.37. Banking Tutorial: Output of Example3.java**

Loading file: Example3.drl

Inserting fact: 3

Inserting fact: 1

Inserting fact: 4

Inserting fact: 1

Inserting fact: 5

Number found with value: 1

Number found with value: 1

Number found with value: 3

Number found with value: 4

Number found with value: 5

We are ready to start moving towards our personal accounting rules. The first step is to create a Cashflowobject.

**Example 9.38. Banking Tutorial: 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 + "]";

    }

}

Class Cashflow has two simple attributes, a date and an amount. (Note that using the type double for monetary units is generally *not* a good idea because floating point numbers cannot represent most numbers accurately.) There is also an overloaded constructor to set the values, and a method toString to print a cashflow. The Java code of **Example4.java** inserts five Cashflow objects, with varying dates and amounts.

**Example 9.39. Banking Tutorial: Example4.java**

**public** **class** Example4 {

    **public** **static** void main(String[] args) **throws** Exception {

        Object[] cashflows = {

            **new** Cashflow(**new** SimpleDate("01/01/2007"), 300.00),

            **new** Cashflow(**new** SimpleDate("05/01/2007"), 100.00),

            **new** Cashflow(**new** SimpleDate("11/01/2007"), 500.00),

            **new** Cashflow(**new** SimpleDate("07/01/2007"), 800.00),

            **new** Cashflow(**new** SimpleDate("02/01/2007"), 400.00),

        };

        **new** RuleRunner().runRules( **new** String[] { "Example4.drl" },

                                   cashflows );

    }

}

The convenience class SimpleDate extends java.util.Date, providing a constructor taking a String as input and defining a date format. The code is listed below

**Example 9.40. Banking Tutorial: Class SimpleDate**

**public** **class** SimpleDate **extends** Date {

    **private** **static** **final** SimpleDateFormat format = **new** SimpleDateFormat("dd/MM/yyyy");

    **public** SimpleDate(String datestr) **throws** Exception {

        setTime(format.parse(datestr).getTime());

    }

}

Now, let’s look at **Example4.drl** to see how we print the sorted Cashflow objects:

**Example 9.41. Banking Tutorial: Rule in Example4.drl**

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 rule we ensure that there is no Cashflow with an earlier date than the one found. In the consequence, we print the Cashflowthat satisfies the rule and then retract it, making way for the next earliest Cashflow. So, the output we generate is:

**Example 9.42. Banking Tutorial: Output of Example4.java**

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

Next, we extend our Cashflow, resulting in a TypedCashflow which can be a credit or a debit operation. (Normally, we would just add this to the Cashflow type, but we use extension to keep the previous version of the class intact.)

**Example 9.43. Banking Tutorial: 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.

Now let's create Example5, a class for running our code.

**Example 9.44. Banking Tutorial: Example5.java**

**public** **class** Example5 {

    **public** **static** void main(String[] args) **throws** Exception {

        Object[] cashflows = {

            **new** TypedCashflow(**new** SimpleDate("01/01/2007"),

                              TypedCashflow.CREDIT, 300.00),

            **new** TypedCashflow(**new** SimpleDate("05/01/2007"),

                              TypedCashflow.CREDIT, 100.00),

            **new** TypedCashflow(**new** SimpleDate("11/01/2007"),

                              TypedCashflow.CREDIT, 500.00),

            **new** TypedCashflow(**new** SimpleDate("07/01/2007"),

                              TypedCashflow.DEBIT, 800.00),

            **new** TypedCashflow(**new** SimpleDate("02/01/2007"),

                              TypedCashflow.DEBIT, 400.00),

        };

        **new** RuleRunner().runRules( **new** String[] { "Example5.drl" },

                                   cashflows );

    }

}

Here, we simply create a set of Cashflow objects which are either credit or debit operations. We supply them and **Example5.drl** to the RuleEngine.

Now, let’s look at a rule printing the sorted Cashflow objects.

**Example 9.45. Banking Tutorial: Rule in Example5.drl**

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 fact with a type of CREDIT and extract the date and the amount. In the second line of the rule we ensure that there is no Cashflow of the same type with an earlier date than the one found. In the consequence, we print the cashflow satisfying the patterns and then retract it, making way for the next earliest cashflow of type CREDIT.

So, the output we generate is

**Example 9.46. Banking Tutorial: Output of Example5.java**

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

Continuing our banking exercise, we are now going to process both credits and debits on two bank accounts, calculating the account balance. In order to do this, we create two separate Account objects and inject them into the Cashflows objects before passing them to the Rule Engine. The reason for this is to provide easy access to the correct account without having to resort to helper classes. Let’s take a look at the Account class first. This is a simple Java object with an account number and balance:

**Example 9.47. Banking Tutorial: 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, resulting in AllocatedCashflow, to include an Account reference.

**Example 9.48. Banking Tutorial: 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() + "]";

    }

}

The Java code of **Example5.java** creates two Account objects and passes one of them into each cashflow, in the constructor call.

**Example 9.49. Banking Tutorial: Example5.java**

**public** **class** Example6 {

    **public** **static** void main(String[] args) **throws** Exception {

        Account acc1 = **new** Account(1);

        Account acc2 = **new** Account(2);

        Object[] cashflows = {

            **new** AllocatedCashflow(acc1,**new** SimpleDate("01/01/2007"),

                                  TypedCashflow.CREDIT, 300.00),

            **new** AllocatedCashflow(acc1,**new** SimpleDate("05/02/2007"),

                                  TypedCashflow.CREDIT, 100.00),

            **new** AllocatedCashflow(acc2,**new** SimpleDate("11/03/2007"),

                                  TypedCashflow.CREDIT, 500.00),

            **new** AllocatedCashflow(acc1,**new** SimpleDate("07/02/2007"),

                                  TypedCashflow.DEBIT,  800.00),

            **new** AllocatedCashflow(acc2,**new** SimpleDate("02/03/2007"),

                                  TypedCashflow.DEBIT,  400.00),

            **new** AllocatedCashflow(acc1,**new** SimpleDate("01/04/2007"),

                                  TypedCashflow.CREDIT, 200.00),

            **new** AllocatedCashflow(acc1,**new** SimpleDate("05/04/2007"),

                                  TypedCashflow.CREDIT, 300.00),

            **new** AllocatedCashflow(acc2,**new** SimpleDate("11/05/2007"),

                                  TypedCashflow.CREDIT, 700.00),

            **new** AllocatedCashflow(acc1,**new** SimpleDate("07/05/2007"),

                                  TypedCashflow.DEBIT,  900.00),

            **new** AllocatedCashflow(acc2,**new** SimpleDate("02/05/2007"),

                                  TypedCashflow.DEBIT,  100.00)

        };

        **new** RuleRunner().runRules( **new** String[] { "Example6.drl" },

                                   cashflows );

    }

}

Now, let’s look at the rule in **Example6.drl** to see how we apply each cashflow in date order and calculate and print the balance.

**Example 9.50. Banking Tutorial: Rule in Example6.drl**

rule "Rule 06 - Credit"

when

$cashflow : AllocatedCashflow( $account : account,

$date : date,

$amount : amount,

type == TypedCashflow.CREDIT )

not AllocatedCashflow( account == $account, date < $date)

then

System.out.println("Credit: " + $date + " :: " + $amount);

$account.setBalance($account.getBalance()+$amount);

System.out.println("Account: " + $account.getAccountNo() +

" - new balance: " + $account.getBalance());

retract($cashflow);

end

rule "Rule 06 - Debit"

when

$cashflow : AllocatedCashflow( $account : account,

$date : date,

$amount : amount,

type == TypedCashflow.DEBIT )

not AllocatedCashflow( account == $account, date < $date)

then

System.out.println("Debit: " + $date + " :: " + $amount);

$account.setBalance($account.getBalance() - $amount);

System.out.println("Account: " + $account.getAccountNo() +

" - new balance: " + $account.getBalance());

retract($cashflow);

end

Although we have separate rules for credits and debits, but we do not specify a type when checking for earlier cashflows. This is so that all cashflows are applied in date order, regardless of the cashflow type. In the conditions we identify the account to work with, and in the consequences we update it with the cashflow amount.

**Example 9.51. Banking Tutorial: Output of Example6.java**

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

**9.6. Pricing Rule Decision Table Example**

The Pricing Rule decision table demonstrates the use of a decision table in a spreadsheet, in Excel's XLS format, in calculating the retail cost of an insurance policy. The purpose of the provide set of rules is to calculate a base price and a discount for a car driver applying for a specific policy. The driver's 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 discount percentage.

**Name:** Example Policy Pricing

**Main class:** org.drools.examples.decisiontable.PricingRuleDTExample

**Module:** drools-examples

**Type:** Java application

**Rules file:** ExamplePolicyPricing.xls

**Objective:** demonstrate spreadsheet-based decision tables.

**9.6.1. Executing the example**

Open the file **PricingRuleDTExample.java** and execute it as a Java application. It should produce the following output in the Console window:

Cheapest possible

BASE PRICE IS: 120

DISCOUNT IS: 20

The code to execute the example follows the usual pattern. The rules are loaded, the facts inserted and a Stateless Session is created. What is different is how the rules are added.

DecisionTableConfiguration dtableconfiguration =

    KnowledgeBuilderFactory.newDecisionTableConfiguration();

        dtableconfiguration.setInputType( DecisionTableInputType.XLS );

        KnowledgeBuilder kbuilder = KnowledgeBuilderFactory.newKnowledgeBuilder();

        Resource xlsRes = ResourceFactory.newClassPathResource( "ExamplePolicyPricing.xls",

                                                                getClass() );

        kbuilder.add( xlsRes,

                      ResourceType.DTABLE,

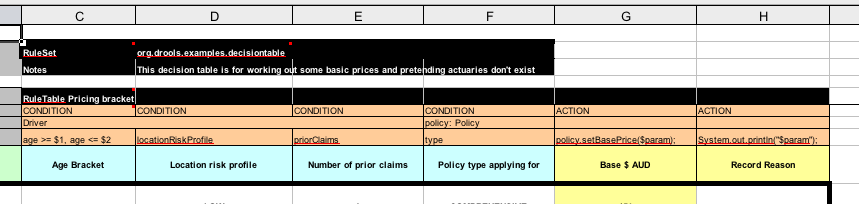
                      dtableconfiguration );

Note the use of the DecisionTableConfiguration object. Its input type is set to DecisionTableInputType.XLS. If you use the BRMS, all this is of course taken care of for you.

There are two fact types 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 it has not yet been approved.

**9.6.2. The decision table**

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

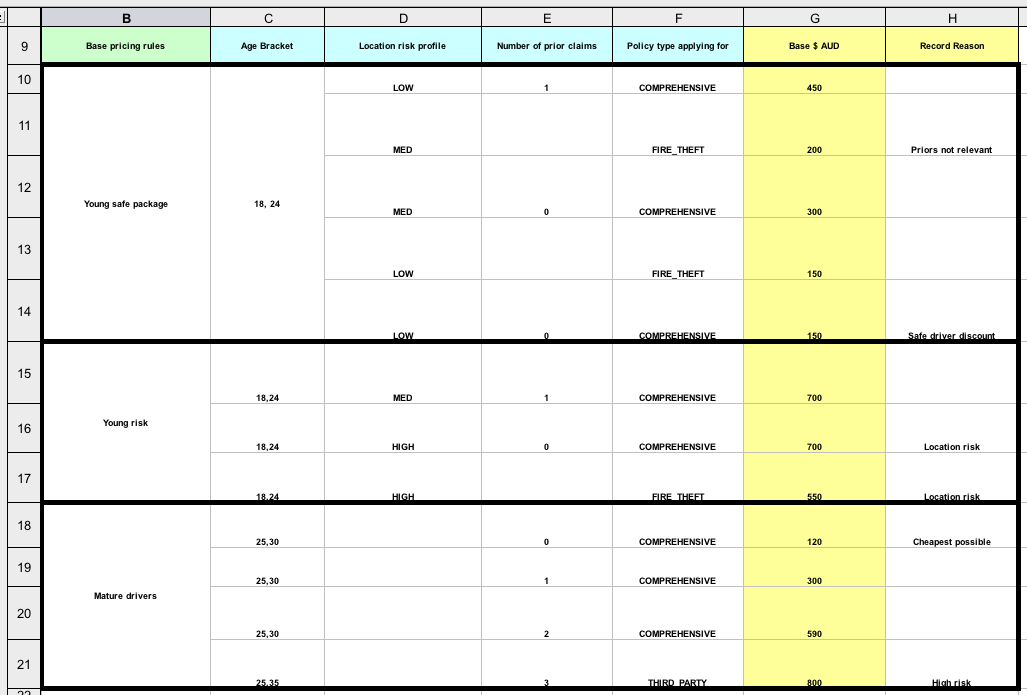


**Figure 9.10. Decision table configuration**

Referring to the spreadsheet show 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.

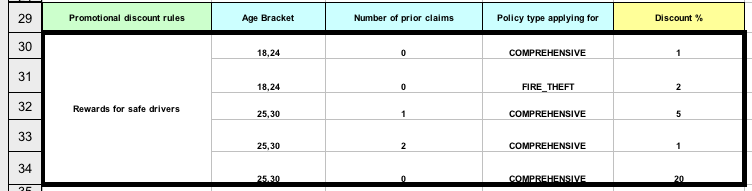
Moving further down, we can see the RuleTable declaration. The name after this (Pricing bracket) is used as the prefix for all the generated rules. Below that, we have "CONDITION or ACTION", indicating the purpose of the column, i.e., whether it forms part of the condition or the consequence of the rule that will be generated.

You can see that there is a driver, his data spanned across three cells, which means that the template expressions below it apply to that fact. We observe the driver's age range (which uses $1 and $2 with comma-separated values), locationRiskProfile, and priorClaims in the respective columns. In the action columns, we are set the policy base price and log a message.



**Figure 9.11. Base price calculation**

In the preceding spreadsheet section, there are broad category brackets, indicated by the comment in the leftmost column. As we know the details of our drivers and their policies, 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.



**Figure 9.12. Discount calculation**

The above section contains the conditions for the discount we might grant our driver. The discount results from the Age bracket, the number of prior claims, and the policy type. In our case, the driver is 30, with no prior claims, and is applying for a COMPREHENSIVE policy, which means we can give a discount of 20%. Note that this is actually a separate table, but in the same worksheet, so that 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 resulting in rules. This is a subtle difference that confuses some people. The evaluation of the rules is not necessarily in the given order, since all the normal mechanics of the rule engine still apply.

**9.7. Pet Store Example**

**Name:** Pet Store

**Main class:** org.drools.examples.petstore.PetStoreExample

**Module:** drools-examples

**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 demonstrates 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 illustrates the mixing of the Java and MVEL dialects within the rules, the use of accumulate functions and the way of calling Java functions from within the ruleset.

All of the Java code is contained in one file, **PetStore.java**, defining the following principal classes (in addition to several classes to handle Swing Events):

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

Much of the Java code is either plain JavaBeans or Swing-based. Only a few Swing-related points will be discussed in this section, but a good tutorial about Swing components can be found at Sun's Swing website, in [http://java.sun.com/docs/books/tutorial/uiswing/](https://docs.jboss.org/drools/release/5.2.0.Final/drools-expert-docs/html/ch09.html???).

The pieces of Java code in **Petstore.java** that relate to rules and facts are shown below.

**Example 9.52. Creating the PetStore RuleBase in PetStore.main**

KnowledgeBuilder kbuilder = KnowledgeBuilderFactory.newKnowledgeBuilder();

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

                                                    PetStore.**class** ),

              ResourceType.DRL );

KnowledgeBase kbase = KnowledgeBaseFactory.newKnowledgeBase();

kbase.addKnowledgePackages( kbuilder.getKnowledgePackages() );

// Create the stock.

Vector<Product> stock = **new** Vector<Product>();

stock.add( **new** Product( "Gold Fish", 5 ) );

stock.add( **new** Product( "Fish Tank", 25 ) );

stock.add( **new** Product( "Fish Food", 2 ) );

// A callback is responsible **for** populating the

// Working Memory and **for** firing all rules.

PetStoreUI ui = **new** PetStoreUI( stock,

                                **new** CheckoutCallback( kbase ) );

ui.createAndShowGUI();

The code shown above loads the rules from a DRL file on 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 a PetStoreUI object is created using a constructor accepting the Vector object stockcollecting our products, and an instance of the CheckoutCallback class containing the Rule Base that we have just loaded.

The Java code that fires the rules is within the CheckoutCallBack.checkout() method. This is triggered (eventually) when the Checkout button is pressed by the user.

**Example 9.53. Firing the Rules - extract from CheckoutCallBack.checkout()**

**public** String checkout(JFrame frame, List<Product> items) {

    Order order = **new** Order();

    // Iterate through list and add to cart

    **for** ( Product p: items ) {

        order.addItem( **new** Purchase( order, p ) );

    }

    // Add the JFrame to the ApplicationData to allow **for** user interaction

    StatefulKnowledgeSession ksession = kbase.newStatefulKnowledgeSession();

    ksession.setGlobal( "frame", frame );

    ksession.setGlobal( "textArea", **this**.output );

    ksession.insert( **new** Product( "Gold Fish", 5 ) );

    ksession.insert( **new** Product( "Fish Tank", 25 ) );

    ksession.insert( **new** Product( "Fish Food", 2 ) );

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

    ksession.insert( order );

    ksession.fireAllRules();

    // Return the state of the cart

    **return** order.toString();

}

Two items get passed into this method. One is the handle to the JFrame Swing component surrounding the output text frame, at the bottom of the GUI. The second is a list of order items; this comes from the TableModel storing 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 file **PetStore.java**. 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's Working Memory.

Within this code, there are nine calls to the Working Memory. The first of these creates a new Working Memory, as a Stateful Knowledge Session from the Knowledge Base. Remember that we passed in this Knowledge Base 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 the Swing frame used for writing messages.

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 9.54. Package, Imports, Globals and Dialect: extract from PetStore.drl**

**package** org.drools.examples

**import** org.drools.WorkingMemory

**import** org.drools.examples.petstore.PetStoreExample.Order

**import** org.drools.examples.petstore.PetStoreExample.Purchase

**import** org.drools.examples.petstore.PetStoreExample.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 file **PetStore.drl** contains the standard package and import statements to make various Java classes available to the rules. New to us are the two globals frame and textArea. They hold references to the Swing components JFrame and JTextArea components that were previously passed on by the Java code calling the setGlobal() method. Unlike variables in rules, which expire as soon as the rule has fired, global variables retain their value for the lifetime of the Session.

The next extract from the file **PetStore.drl** contains two functions that are referenced by the rules that we will look at shortly.

**Example 9.55. Java Functions in the Rules: extract 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 just makes the Pet Store example 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 import function my.package.Foo.hello.

The purpose of these two functions is:

* doCheckout() displays a dialog asking users whether they wish to checkout. If they do, focus is set to thecheckOut agenda-group, allowing rules in that group to (potentially) fire.
* requireTank() displays a dialog asking users whether they wish to buy a tank. If so, a new fish tankProduct is added to the order list in Working Memory.

We'll see the rules that call these functions later on. The next set of examples are from the Pet Store rules themselves. The first extract is the one that happens to fire first, partly because it has the auto-focusattribute set to true.

**Example 9.56. Putting items into working memory: extract 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 );

kcontext.getKnowledgeRuntime().getAgenda().getAgendaGroup( "show items" ).setFocus();

kcontext.getKnowledgeRuntime().getAgenda().getAgendaGroup( "evaluate" ).setFocus();

end

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

* agenda-group "init" defines the name of the agenda group. In this case, there is only one rule in the group. However, neither the Java code nor a rule consequence sets the focus to this group, and therefore it relies on the next attribute for its chance to fire.
* auto-focus true ensures that this rule, while being the only rule in the agenda group, gets a chance to fire when fireAllRules() is called from the Java code.
* kcontext....setFocus() sets the focus to the "show items" and "evaluate" agenda groups in turn, permitting their rules 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 show the rules within the "show items" and evaluate agenda groups. We look at them in the order that they are called.

**Example 9.57. Show Items in the GUI - extract 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, called "Show Items" (note the difference in case). For each purchase on the order currently in the Working Memory (or 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 listed previously. This Agenda group has two rules, "Free Fish Food Sample" and "Suggest Tank", shown below.

**Example 9.58. Evaluate Agenda Group: extract from PetStore.drl**

// Free Fish Food sample when we buy a Gold Fish if we haven't already bought

// Fish Food and don't 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 don't 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 rule "Free Fish Food Sample" will only fire if

* we *don't*already have any fish food, *and*
* we *don't* already have a free fish food sample, *and*
* 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 rule "Suggest Tank" will only fire if

* we *don't*already have a Fish Tank in our order, *and*
* we *do* have 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 9.59. Doing the Checkout - extract (6) from PetStore.drl**

rule "do checkout"

dialect "java"

when

then

doCheckout(frame, drools.getWorkingMemory());

end

The rule "do checkout" has **no agenda group set and no auto-focus attribute**. As such, is is deemed part of the default (MAIN) agenda group. This group gets focus by default when all the rules in 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 thedoCheckout() function, the rule passes the global frame variable to give the function 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 9.60. Checkout Rules: extract 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" );

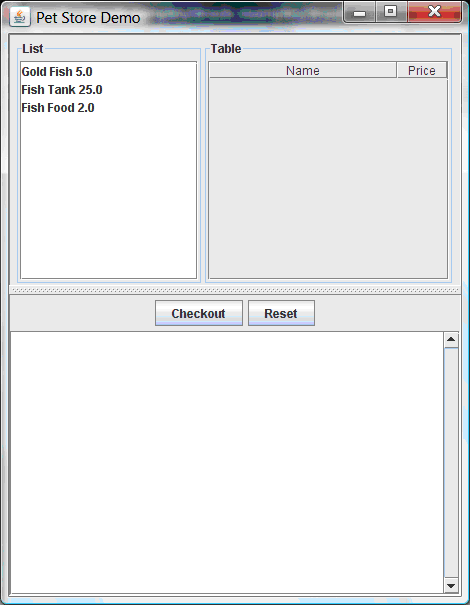
end

There are three rules in the checkout agenda-group:

* If we haven't already calculated the gross total, Gross Total accumulates the product prices into a total, puts this total into Working Memory, and displays it via the Swing JTextArea, using the textArea global variable yet again.
* If our gross total is between 10 and 20, "Apply 5% Discount" calculates the discounted total and adds it to the Working Memory and displays it in the text area.
* If our gross total is not less than 20, "Apply 10% Discount" calculates the discounted total and adds it to the Working Memory and displays it in the text area.

Now that we've run through what happens in the code, let's have a look at what happens when we actually run the code. The file **PetStore.java** contains a main() method, so that 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).

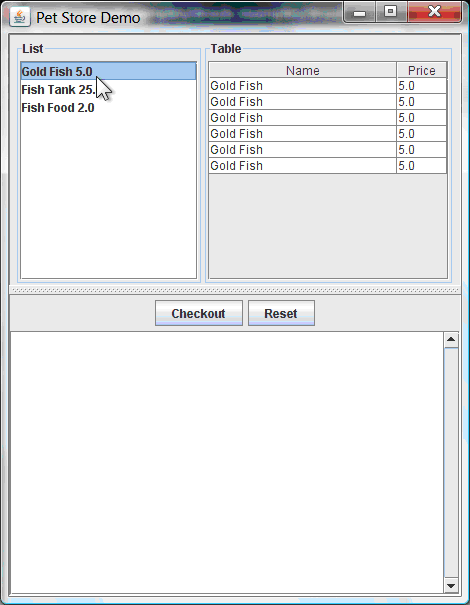


**Figure 9.13. PetStore Demo just after Launch**

To get to this point, the following things have happened:

1. The main() method has run and loaded the Rule Base *but not yet fired the rules*. So far, this is the only code in connection with rules that has been run.
2. A new PetStoreUI object has been created and given a handle to the Rule Base, 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.

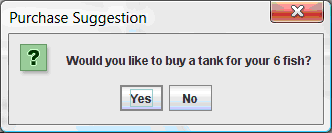


**Figure 9.14. PetStore Demo with Products Selected**

Note that *no rules code has been fired here*. This is only Swing code, listening for mouse click events, and adding some selected product to the TableModel object for display in the top right hand section. (As an aside, note that this is a classic use of the Model View Controller design pattern).

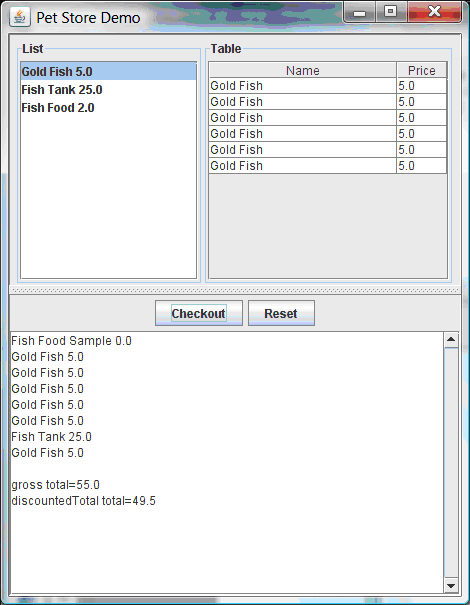
It is only when we press the "Checkout" button that we fire our business rules, in roughly the same order that we walked through the code earlier.

1. Method CheckOutCallBack.checkout() 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 inserts it into the Session's Working Memory. It then fires the rules.
2. The "Explode Cart" rule is the first to fire, given that it has auto-focus set to true. It loops through all the products in the cart, ensures that the products are in the Working Memory, and 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 (at the bottom of the window), decide whether or not to give us free fish food, and to ask us whether we want to buy a fish tank. This is shown in the figure below.



**Figure 9.15. 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 - see the figure below.



**Figure 9.16. Petstore Demo after all rules have fired.**

We could add more System.out calls to demonstrate this flow of events. The output, as it currently appears in the Console window, is given in the listing below.

**Example 9.61. 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

**9.8. Honest Politician Example**

**Name:** Honest Politician

**Main class:** org.drools.examples.honestpolitician.HonestPoliticianExample

**Module:** drools-examples

**Type:** Java application

**Rules file:** HonestPoliticianExample.drl

**Objective:** Illustrate the concept of "truth maintenance" based on the logical insertion of facts

The Honest Politician example demonstrates truth maintenance with logical assertions. The basic premise is that an object can only exist while a statement is true. A rule's consequence can logically insert an object with the insertLogical() method. This means the object will only remain in the Working Memory as long as the rule that logically inserted it remains true. When the rule is no longer true the object is automatically retracted.

In this example there is the class Politician, with a name and a boolean value for being honest. Four politicians with honest state set to true are inserted.

**Example 9.62. Class Politician**

**public** **class** Politician {

    **private** String name;

    **private** boolean honest;

    ...

}

**Example 9.63. Honest Politician: 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 window output shows that, while there is at least one honest politician, democracy lives. However, as each politician is in turn corrupted by an evil corporation, so that all politicians become dishonest, democracy is dead.

**Example 9.64. Honest Politician: 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 at least one honest politician in the Working Memory a new Hope object is logically asserted. This object will only exist while there is at least one honest politician. As soon as all politicians are dishonest, the Hope object will be automatically retracted. This rule is given a salience of 10 to ensure that it fires before any other rule, as at this stage the "Hope is Dead" rule is actually true.

**Example 9.65. Honest Politician: 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 9.66. Honest Politician: Rule "Hope Lives"**

rule "Hope Lives"

salience 10

when

exists( Hope() )

then

System.out.println("Hurrah!!! Democracy Lives");

end

Now that there is hope and we have, at the start, four honest politicians, we have four activations for this rule, all in conflict. They will fire 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 inserted (due to the last execution of new Hope()) is automatically retracted.

**Example 9.67. Honest Politician: 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 the Hope object being automatically retracted, via the truth maintenance system, the conditional element not applied to Hope is no longer true so that the following rule will match and fire.

**Example 9.68. Honest Politician: Rule "Hope is Dead"**

rule "Hope is Dead"

when

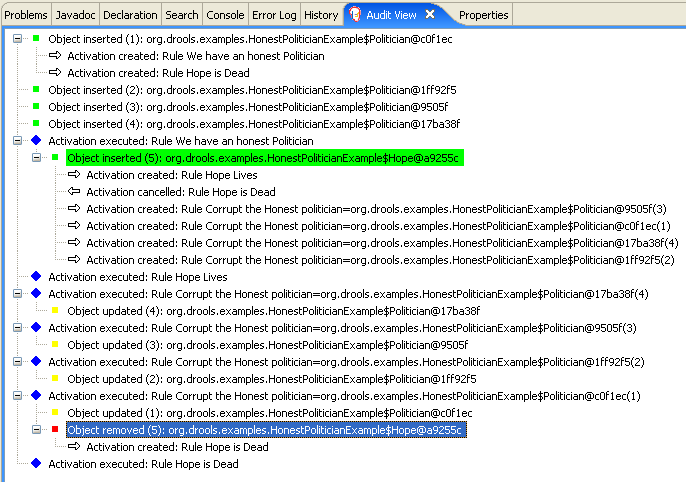
not( Hope() )

then

System.out.println( "We are all Doomed!!! Democracy is Dead" );

end

Let's take a look at the Audit trail for this application:



**Figure 9.17. Honest Politician Example Audit View**

The moment we insert the first politician we have two activations. The rule "We have an honest Politician" is activated only once for the first inserted politician because it uses an exists conditional element, which matches once for any number. The rule "Hope is Dead" is also activated at this stage, because we have not yet inserted the Hope object. Rule "We have an honest Politician" fires first, as it has a higher salience than "Hope is Dead", which inserts the Hope object. (That action is highlighted green.) The insertion of the Hopeobject activates "Hope Lives" and de-activates "Hope is Dead"; it also activates "Corrupt the Honest" for each inserted honest politician. Rule "Hope Lives" executes, printing "Hurrah!!! Democracy Lives". Then, for each politician, 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 politician's honest value to false. When the last honest politician is corrupted, Hope is automatically retracted, by the truth maintenance system, as shown by the blue highlighted area. The green highlighted area shows the origin of the currently selected blue highlighted area. Once the Hope fact is retracted, "Hope is dead" activates and fires printing "We are all Doomed!!! Democracy is Dead".

**9.9. Sudoku Example**

**Name:** Sudoku

**Main class:** org.drools.examples.sudoku.SudokuExample

**Type:** Java application

**Rules file:** sudoku.drl, validate.drl

**Objective:** Demonstrates the solving of logic problems, and complex pattern matching.

This example demonstrates how Drools can be used to find a solution in a large potential solution space based on a number of constraints. We use the popular puzzle of Sudoku. This example also shows how Drools can be integrated into a graphical interface and how callbacks can be used to interact with a running Drools rules engine in order to update the graphical interface based on changes in the Working Memory at runtime.

**9.9.1. 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.

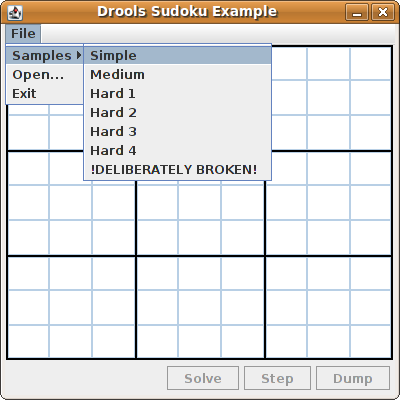
The general strategy to solve the problem is to ensure that when you insert a new number it should be unique in its particular 3x3 zone, row and column.

See [Wikipedia](http://en.wikipedia.org/wiki/Sudoku) for a more detailed description.

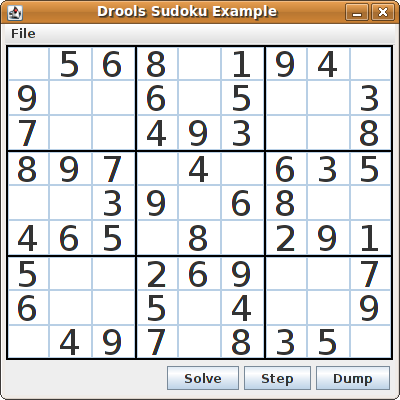
**9.9.2. Running the Example**

Download and install drools-examples as described above and then execute **java org.drools.examples.DroolsExamplesApp** and click on "SudokuExample".

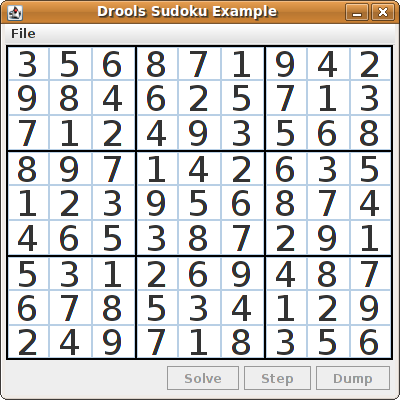
The window contains an empty grid, but the program comes with a number of grids stored internally which can be loaded and solved. Click on "File", then "Samples" and select "Simple" to load one of the examples. Note that all buttons are disabled until a grid is loaded.



Loading the "Simple" example fills the grid according to the puzzle's initial state.



Click on the "Solve" button and the Drools-based engine will fill out the remaining values, and the buttons are inactive once more.



Alternatively, you may click on the "Step" button to see the next digit found by the rule set. The Console window will display detailed information about the rules which are executing to solve the step in a human readable form. Some examples of these messages are presented below.

single 8 at [0,1]

column elimination due to [1,2]: remove 9 from [4,2]

hidden single 9 at [1,2]

row elimination due to [2,8]: remove 7 from [2,4]

remove 6 from [3,8] due to naked pair at [3,2] and [3,7]

hidden pair in row at [4,6] and [4,4]

Click on the "Dump" button to see the state of the grid, with cells showing either the established value or the remaining possibilitiescandidates.

Col: 0 Col: 1 Col: 2 Col: 3 Col: 4 Col: 5 Col: 6 Col: 7 Col: 8

Row 0: 2 4 7 9 2 456 4567 9 23 56 9 --- 5 --- --- 1 --- 3 67 9 --- 8 --- 4 67

Row 1: 12 7 9 --- 8 --- 1 67 9 23 6 9 --- 4 --- 23 67 1 3 67 9 3 67 9 --- 5 ---

Row 2: 1 4 7 9 1 456 --- 3 --- 56 89 5 78 5678 --- 2 --- 4 67 9 1 4 67

Row 3: 1234 12345 1 45 12 5 8 --- 6 --- 2 5 78 5 78 45 7 --- 9 ---

Row 4: --- 6 --- --- 7 --- 5 --- 4 --- 2 5 8 --- 9 --- 5 8 --- 1 --- --- 3 ---

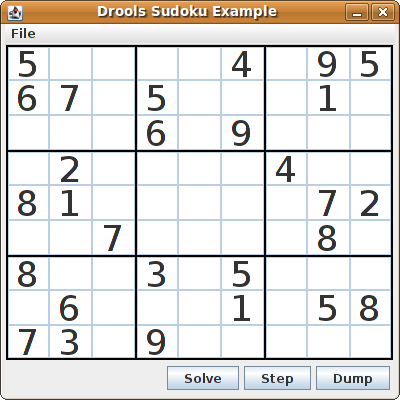
Row 5: --- 8 --- 12 45 1 45 9 12 5 --- 3 --- 2 5 7 567 4567 2 4 67

Row 6: 1 3 7 1 3 6 --- 2 --- 3 56 8 5 8 3 56 8 --- 4 --- 3 567 9 1 678

Row 7: --- 5 --- 1 34 6 1 4 678 3 6 8 --- 9 --- 34 6 8 1 3 678 --- 2 --- 1 678

Row 8: 34 --- 9 --- 4 6 8 --- 7 --- --- 1 --- 23456 8 3 56 8 3 56 6 8

Now, let us load a Sudoku grid that is deliberately invalid. Click on "File", "Samples" and "!DELIBERATELY BROKEN!". Note that this grid starts with some issues, for example the value 5 appears twice in the first row.



A few simple rules perform a sanity check, right after loading a grid. In this case, the following messages are printed on standard output:

cell [0,8]: 5 has a duplicate in row 0

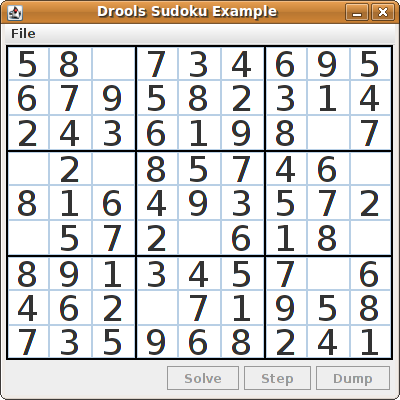
cell [0,0]: 5 has a duplicate in row 0

cell [6,0]: 8 has a duplicate in col 0

cell [4,0]: 8 has a duplicate in col 0

Validation complete.

Nevertheless, click on the "Solve" button to apply the solving rules to this invalid grid. This will not complete; some cells remain empty.



The solving functionality has been achieved by the use of rules that implement standard solving techniques. They are based on the sets of values that are still candidates for a cell. If, for instance, such a set contains a single value, then this is the value for the cell. A little less obvious is the single occurrence of a value in one of the groups of nine cells. The rules detecting these situations insert a fact of type Setting with the solution value for some specific cell. This fact causes the elimination of this value from all other cells in any of the groups the cell belongs to. Finally, it is retracted.

Other rules merely reduce the permissible values for some cells. Rules "naked pair", "hidden pair in row", "hidden pair in column" and "hidden pair in square" merely eliminate possibilities but do not establish solutions. More sophisticated eliminations are done by "X-wings in rows", "X-wings in columns", "intersection removal row" and "intersection removal column".

**9.9.3. 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.

The package 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 Cellobjects. SudokuGridView is a Swing component which can visualize 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 demonstration purposes.

Package org.drools.examples.sudoku.rules contains a utility class with a method for compiling DRL files.

The package org.drools.examples.sudoku contains a set of classes implementing the elementary Cell object and its various aggregations: the CellFile subtypes CellRow and CellCol as well as CellSqr, all of which are subtypes of CellGroup. It's interesting to note that Cell and CellGroup are subclasses of SetOfNine, which provides a property free with the type Set<Integer>. For a Cell it represents the individual candidate set; for a CellGroup the set is the union of all candidate sets of its cells, or, simply, the set of digits that still need to be allocated.

With 81 Cell and 27 CellGroup objects and the linkage provided by the Cell properties cellRow, cellCol and cellSqr and the CellGroup property cells, a list of Cell objects, it is possible to write rules that detect the specific sitations that permit the allocation of a value to a cell or the elimination of a value from some candidate set.

An object of class Setting is used for triggering the operations that accompany the allocation of a value: its removal from the candidata sets of sibling cells and associated cell groups. Moreover, the presence of a Setting fact is used in all rules that should detect a new situation; this is to avoid reactions to inconsistent intermediary states.

An object of class Stepping is used in a low priority rule to execute an emergency halt when a "Step" does not terminate regularly. This indicates that the puzzle cannot be solved by the program.

The class org.drools.examples.sudoku.SudokuExample implements a Java application combining the components desribed.

**9.9.4. Sudoku Validator Rules (validate.drl)**

Validation rules detect duplicate numbers in cell groups. They are combined in an agenda group which enables us to activate them, explicitly, after loading a puzzle.

The three rules "duplicate in cell..." are very similar. The first pattern locates a cell with an allocated value. The second pattern pulls in any of the three cell groups the cell belongs to. The final pattern would find a cell (other than the first one) with the same value as the first cell and in the same row, column or square, respectively.

Rule "terminate group" fires last. It prints a message and calls halt.

**9.9.5. Sudoku Solving Rules (sudoku.drl)**

There are three types of rules in this file: one group handles the allocation of a number to a cell, another group detects feasible allocations, and the third group eliminates values from candidate sets.

Rules "set a value", "eliminate a value from Cell" and "retract setting" depend on the presence of a Settingobject. The first rule handles the assignment to the cell and the operations for removing the value from the "free" sets of the cell's three groups. Also, it decrements a counter that, when zero, returns control to the Java application that has called fireUntilHalt(). The purpose of rule "eliminate a value from Cell" is to reduce the candidate lists of all cells that are related to the newly assigned cell. Finally, when all eliminations have been made, rule "retract setting" retracts the triggering Setting fact.

There are just two rules that detect a situation where an allocation of a number to a cell is possible. Rule "single" fires for a Cell with a candidate set containing a single number. Rule "hidden single" fires when there is no cell with a single candidate but when there is a cell containing a candidate but this candidate is absent from all other cells in one of the three groups the cell belongs to. Both rules create and insert aSetting fact.

Rules from the largest group of rules implement, singly or in groups of two or three, various solving techniques, as they are employed when solving Sudoku puzzles manually.

Rule "naked pair" detects identical candidate sets of size 2 in two cells of a group; these two values may be removed from all other candidate sets of that group.

A similar idea motivates the three rules "hidden pair in..."; here, the rules look for a subset of two numbers in exactly two cells of a group, with neither value occurring in any of the other cells of this group. This, then, means that all other candidates can be eliminated from the two cells harbouring the hidden pair.

A pair of rules deals with "X-wings" in rows and columns. When there are only two possible cells for a value in each of two different rows (or columns) and these candidates lie also in the same columns (or rows), then all other candidates for this value in the columns (or rows) can be eliminated. If you follow the pattern sequence in one of these rules, you will see how the conditions that are conveniently expressed by words such as "same" or "only" result in patterns with suitable constraints or prefixed with "not".

The rule pair "intersection removal..." is based on the restricted occurrence of some number within one square, either in a single row or in a single column. This means that this number must be in one of those two or three cells of the row or column; hence it can be removed from the candidate sets of all other cells of the group. The pattern establishes the restricted occurrence and then fires for each cell outside the square and within the same cell file.

These rules are sufficient for many but certainly not for all Sudoku puzzles. To solve very difficult grids, the rule set would need to be extended with more complex rules. (Ultimately, there are puzzles that cannot be solved except by trial and error.)

**9.10. Number Guess**

**Name:** Number Guess

**Main class:** org.drools.examples.numberguess.NumberGuessExample

**Module:** droolsjbpm-integration-examples (Note: this is in a different download, the droolsjbpm-integration download.)

**Type:** Java application

**Rules file:** NumberGuess.drl

**Objective:** Demonstrate use of Rule Flow to organise Rules

The "Number Guess" example shows the use of Rule Flow, a way of controlling the order in which rules are fired. It uses widely understood workflow diagrams for defining the order in which groups of rules will be executed.

**Example 9.69. Creating the Number Guess RuleBase: NumberGuessExample.main() - part 1**

**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 an additional line to add the Rule Flow (**NumberGuess.rf**), which provides the option of specifying different rule flows for the same Knowledge Base. Otherwise, the Knowledge Base is created in the same manner as before.

**Example 9.70. Starting the RuleFlow: NumberGuessExample.main() - part 2**

**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();

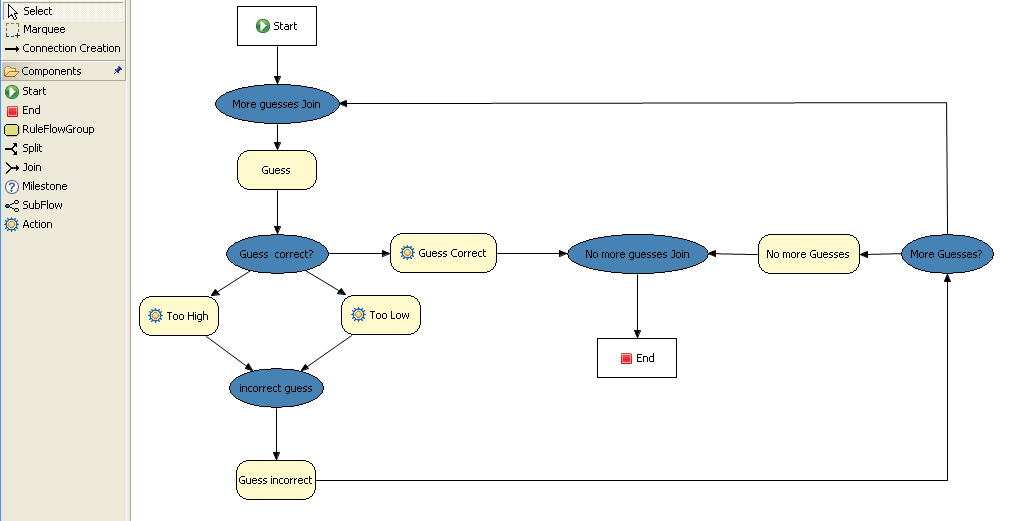
logger.close();

ksession.dispose();

Once we have a Knowledge Base, we can use it to obtain a Stateful Session. Into our session we insert our facts, i.e., standard Java objects. (For simplicity, in this sample, these classes are all contained within our**NumberGuessExample.java** file. Class GameRules provides the maximum range and the number of guesses allowed. Class RandomNumber automatically generates a number between 0 and 100 and makes it available to our rules, by insertion via the getValue() method. Class Game keeps track of the guesses we have made before, and their number.

Note that before we call the standard fireAllRules() method, we also start the process that we loaded earlier, via the startProcess() method. We'll learn where to obtain the parameter we pass ("Number Guess", i.e., the identifier of the rule flow) when we talk about the rule flow file and the graphical Rule Flow Editor below.

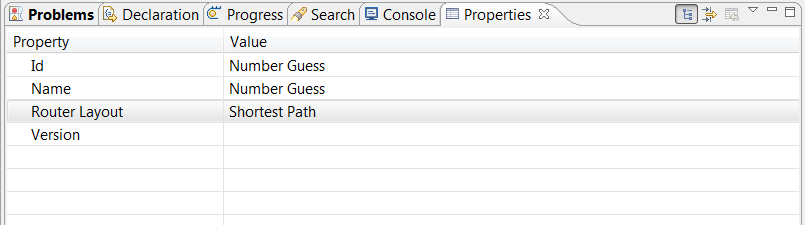
Before we finish the discussion of our Java code, we note that in some real-life application we would examine the final state of the objects. (Here, we could retrieve the number of guesses, to add it to a high score table.) For this example we are content to ensure that the Working Memory session is cleared by calling the dispose() method.



**Figure 9.18. RuleFlow for the NumberGuess Example**

If you open the **NumberGuess.rf** file in the Drools IDE (provided you 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 in 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 called the*palette*. This diagram is saved in XML, an (almost) human readable format, using XStream.

If it is not already open, ensure that the Properties View is visible in the IDE. It can be opened by clicking "Window", then "Show View" and "Other", where you can select the "Properties" view. If you do this *before*you select any item on the rule flow (or click on the blank space in the rule flow) you should be presented with the following set of properties.



**Figure 9.19. Properties for the Number Guess Rule Flow**

Keep an eye on the Properties View as we progress through the example's rule flow, as it presents valuable information. In this case, it provides us with the identification of the Rule Flow Process that we used in our earlier code snippet, when we called session.startProcess().

In the "Number Guess" Rule Flow we encounter several node types, many of them identified by an icon.

* The Start node (white arrow in a green circle) and the End node (red box) mark beginning and end of the rule flow.
* A Rule Flow Group box (yellow, without an icon) represents a Rule Flow Groups defined in our rules (DRL) file that we will look at later. For example, when the flow reaches the Rule Flow Group "Too High", only those rules marked with an attribute of ruleflow-group "Too High" can potentially fire.
* Action nodes (yellow, cog-shaped icon) perform standard Java method calls. Most action nodes in this example call System.out.println(), indicating the program's progress to the user.
* Split and Join Nodes (blue ovals, no icon) such as "Guess Correct?" and "More guesses Join" mark places where the flow of control can split, according to various conditions, and rejoin, respectively
* Arrows indicate the flow between the various nodes.

The various nodes in combination with the rules make the Number Guess game work. For example, the "Guess" Rule Flow Group allows only the rule "Get user Guess" to fire, because only that rule has a matching attribute of ruleflow-group "Guess".

**Example 9.71. A Rule firing only at a specific point in the Rule Flow: 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 ) );

i = br.readLine();

modify ( game ) { guessCount = game.guessCount + 1 }

insert( new Guess( i ) );

end

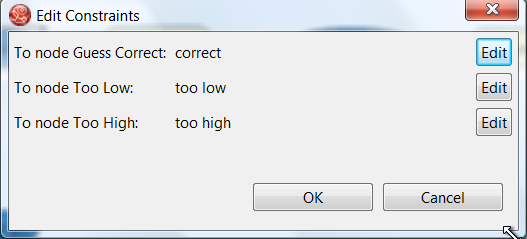
The rest of this rule is fairly standard. The LHS section (after when) of the rule states that it will be activated for each RandomNumber object inserted into the Working Memory where guessCount is less than allowedGuessesfrom the GameRules object and where the user has not guessed the correct number.

The RHS section (or consequence, after then) prints a message to the user and then awaits user input from System.in. After obtaining this input (the readLine() method call blocks until the return key is pressed) it modifies the guess count and inserts the new guess, making both available to the Working Memory.

The rest of the rules file is fairly standard: the package declares the dialect as MVEL, and various Java classes are imported. In total, there are five rules in this file:

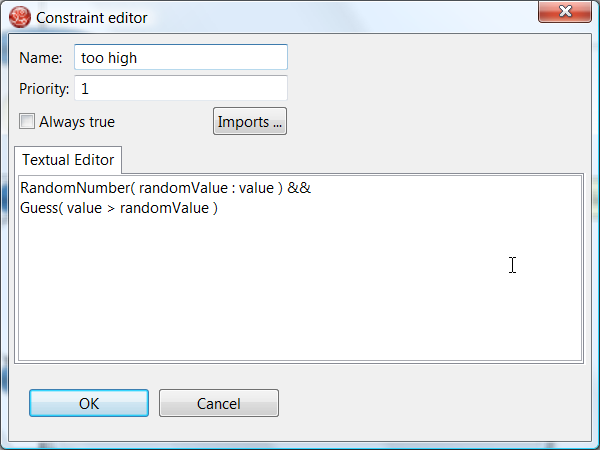
1. Get User Guess, the Rule we examined above.
2. A Rule to record the highest guess.
3. A Rule to record the lowest guess.
4. A Rule to inspect the guess and retract it from memory if incorrect.
5. A Rule that notifies the user that all guesses have been used up.

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 (.drl) file and the Rules Flow .rf files* is that the Split Nodes (the blue ovals) can use values in the 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 by clicking the button at the right that appears once you click on the "Constraints" property line. You should see something similar to the diagram below.



**Figure 9.20. Edit Constraints for the "Guess Correct" Node**

Click on the "Edit" button beside "To node Too High" and you'll see a dialog like the one below. The values in the "Textual Editor" window follow the standard rule format for the 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.



**Figure 9.21. Constraint Editor for the "Guess Correct" Node: value too high**

Since the file **NumberGuess.java** 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 9.72. 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. The main() method of **NumberGuessExample.java** loads a Rule Base, creates a Stateful Session and inserts Game, GameRules and RandomNumber (containing the target number) objects into it. The method also sets the process flow we are going to use, and fires all rules. Control passes to the Rule Flow.
2. File **NumberGuess.rf**, the Rule Flow, 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 Rule Flow Group ("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 Working 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 to take.

If the guess in step 4 was too high or too low, flow proceeds along a path which has an action node with normal Java code printing a suitable message and a Rule Flow Group causing a highest guess or lowest guess rule to be triggered. Flow passes from these nodes to step 6.

If the guess in step 4 was 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 Rule Flow.

1. Control passes as per the Rule Flow via a join node, a guess incorrect Rule Flow Group (triggering a rule to retract a guess from Working Memory) onto the "More guesses" decision node.
2. The "More guesses" decision node (on the right hand side of the rule flow) 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 rule flow, via a Rule Flow Group that triggers a rule stating "you have no more guesses".
3. The loop over steps 3 to 7 continues until the number is guessed correctly, or we run out of guesses.

**9.11. Miss Manners and Benchmarking**

**Name:** Miss Manners

**Main class:** org.drools.benchmark.manners.MannersBenchmark

**Module:** drools-examples

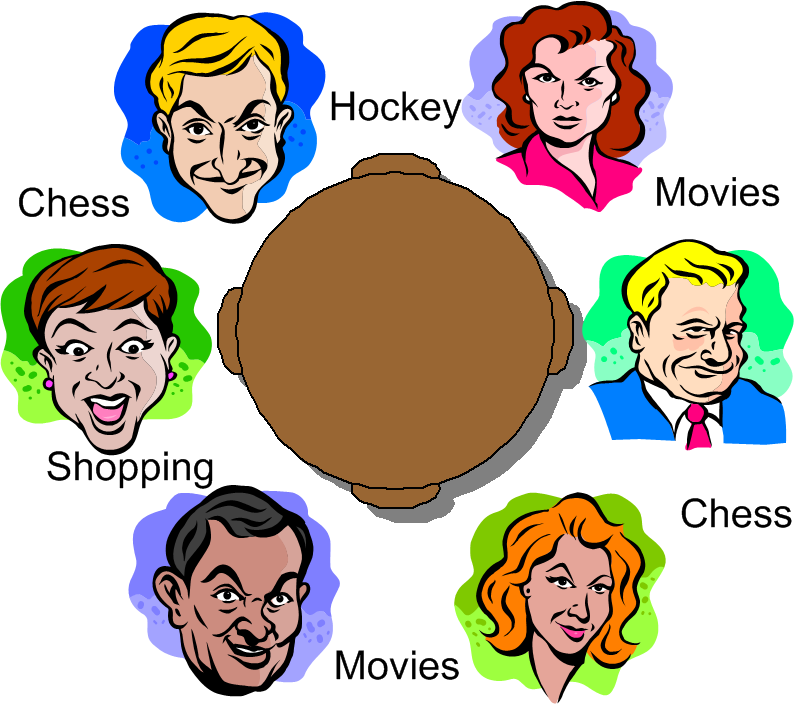
**Type:** Java application

**Rules file:** manners.drl

**Objective:** Advanced walkthrough on the Manners benchmark, covers Depth conflict resolution in depth.

**9.11.1. Introduction**

Miss Manners is throwing a party and, being a 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? She decides to note the hobby of each guest so she can then arrange guests not only pairing them according to alternating sex but also ensuring that a guest has someone with a common hobby, at least on one side.



**Figure 9.22. Miss Manners' Guests**

**9.11.1.1. BenchMarking**

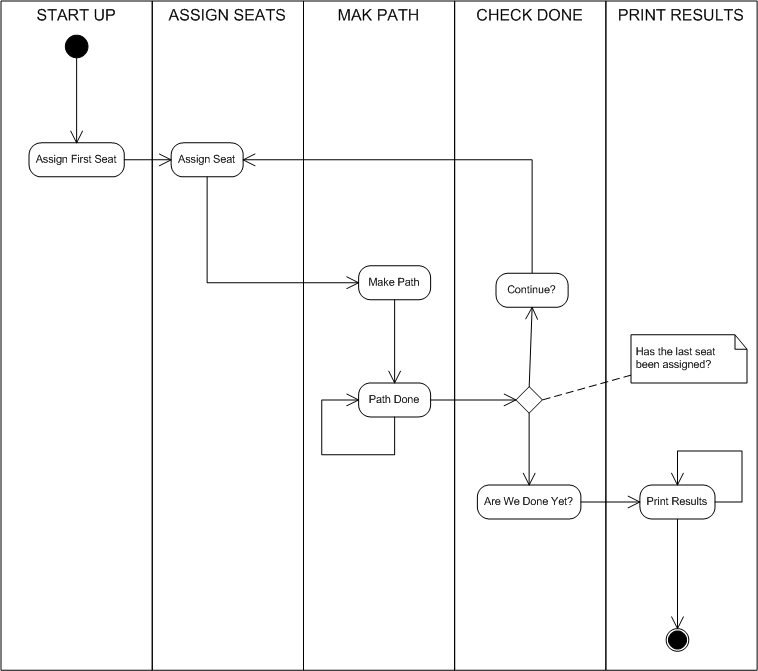
Five benchmarks were established in the 1991 paper "Effects of Database Size on Rule System Performance: Five Case Studies" by David Brant, Timothy Grose, Bernie Lofaso and Daniel P. Miranker:

* **Manners** uses a depth-first search approach to determine the seating arrangements alternating women and men and ensuring one common hobby for neighbors.
* **Waltz** establishes a three-dimensional interpretation of a line drawing by line labeling by constraint propagation.
* **WaltzDB** is a more general version of Waltz, supporting junctions of more than three lines and using a database.
* **ARP** is a route planner for a robotic air vehicle using the A\* search algorithm to achieve minimal cost.
* **Weaver** VLSI router for channels and boxes using a black-board technique.

Manners has become the de facto rule engine benchmark. Its behavior, however, 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 five benchmarks are also published at the University of Texas <http://www.cs.utexas.edu/ftp/pub/ops5-benchmark-suite/>.

**9.11.1.2. 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 9.23. Manners Activity Diagram**

**9.11.1.3. The Data and Results**

Before going deeper into the rules, let's first take a look at the asserted data and the resulting seating arrangement. The data is a simple set of five guests who should be arranged so that sexes alternate and neighbors have a common hobby.

**The Data**

The data is given in OPS5 syntax, with a parenthesized list of name and value pairs for each attribute. Each person has only one hobby.

(guest (name n1) (sex m) (hobby  h1)  )  
(guest (name n2) (sex f) (hobby  h1)  )  
(guest (name n2) (sex f) (hobby  h3)  )  
(guest (name n3) (sex m) (hobby  h3)  )  
(guest (name n4) (sex m) (hobby  h1)  )  
(guest (name n4) (sex f) (hobby  h2)  )  
(guest (name n4) (sex f) (hobby  h3)  )  
(guest (name n5) (sex f) (hobby  h2)  )  
(guest (name n5) (sex f) (hobby  h1)  )  
(last\_seat (seat 5)  )

**The Results**

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 a "pid" value one greater than the last. (The significance of this will be explained in the discussion of the rule "Assign Seating".) The "ls", "rs", "ln" and "rn" refer to the left and right seat and neighbor's name, respectively. The actual implementation uses longer attribute names (e.g., leftGuestName, but here we'll stick to the notation from the original implementation.

[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]

**9.11.2. Indepth Discussion**

**9.11.2.1. Cheating**

Manners has been designed to exercise cross product joins and Agenda activities. Many people not understanding this tweak the example to achieve better performance, making their port of the Manners benchmark pointless. Known cheats or porting errors for Miss Manners are:

* Using arrays for a guests hobbies, instead of asserting each one as a single fact massively reduces the cross products.
* Altering the sequence of data can also reduce the amount of matching, increasing execution speed.
* It's possible to change the not Conditional Element so that the test algorithm only uses the "first-best-match", which is, basically, transforming the test algorithm to backward chaining. The results are only comparable to other backward chaining rule engines or ports of Manners.
* Removing the context so the rule engine matches the guests and seats prematurely. A proper port will prevent facts from matching using the context start.
* It's possible to prevent the rule engine from performing combinatorial pattern matching.
* If no facts are retracted in the reasoning cycle, as a result of the not CE, the port is incorrect.

**9.11.2.2. Conflict Resolution**

The 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 and complexity. The recency part of the strategy drives the depth first (LIFO) firing order. The CLIPS manual documents the Recency strategy as follows:

|  |  |  |
| --- | --- | --- |
|  | 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 entity 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.

**9.11.2.3. Rule "assignFirstSeat"**

Once the context is changed to START\_UP, activations are created for all asserted guest. 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 object would have a higher fact time tag, and its Activation would fire because it has the highest accumulated fact time tag. The execution order in this rule has little importance, but has a big impact in the rule "Assign Seat". The activation fires and asserts the first Seating arrangement and a Path, and then sets the Context attribute state to create an activation for rule findSeating.

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

**9.11.2.4. Rule "findSeating"**

This rule determines each of the Seating arrangements. The rule creates cross product solutions for *all*asserted Seating arrangements against *all* the asserted guests except against itself or any already assigned chosen solutions.

rule findSeating

when

context : Context( state == Context.ASSIGN\_SEATS )

$s : Seating( pathDone == true )

$g1 : Guest( name == $s.rightGuestName )

$g2 : Guest( sex != $g1.sex, hobby == $g1.hobby )

count : Count()

not ( Path( id == $s.id, guestName == $g2.name) )

not ( Chosen( id == $s.id, guestName == $g2.name, hobby == $g1.hobby) )

then

int rightSeat = $s.getRightSeat();

int seatId = $s.getId();

int countValue = count.getValue();

Seating seating =

new Seating( countValue, seatId, false, rightSeat,

$s.getRightGuestName(), rightSeat + 1, $g2.getName() );

insert( seating );

Path path = new Path( countValue, rightSeat + 1, $g2.getName() );

insert( path );

Chosen chosen = new Chosen( seatId, $g2.getName(), $g1.getHobby() );

insert( chosen );

System.err.println( "find seating : " + seating + " : " + path +

" : " + chosen);

modify( count ) {setValue( countValue + 1 )}

modify( context ) {setState( Context.MAKE\_PATH )}

end

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. How can this be possible if we have activations, of the same time tag, for nearly all existing Seating and Guest objects? For example, on the third iteration of findDeating the produced activations will be as shown below. 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 the Context object to ASSIGN\_SEATS. With OPS5 and LEX it would correctly fire the activation with the Seating asserted last. With Depth, the accumulated fact time tag ensures that the activation with the last asserted Seating fires.

**9.11.2.5. Rules "makePath" and "pathDone"**

Rule makePath must always fire before pathDone. A Path object is asserted for each Seating arrangement, up to the last asserted Seating. Notice that the conditions in pathDone are a subset of those in makePath - so how do we ensure that makePath 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 )

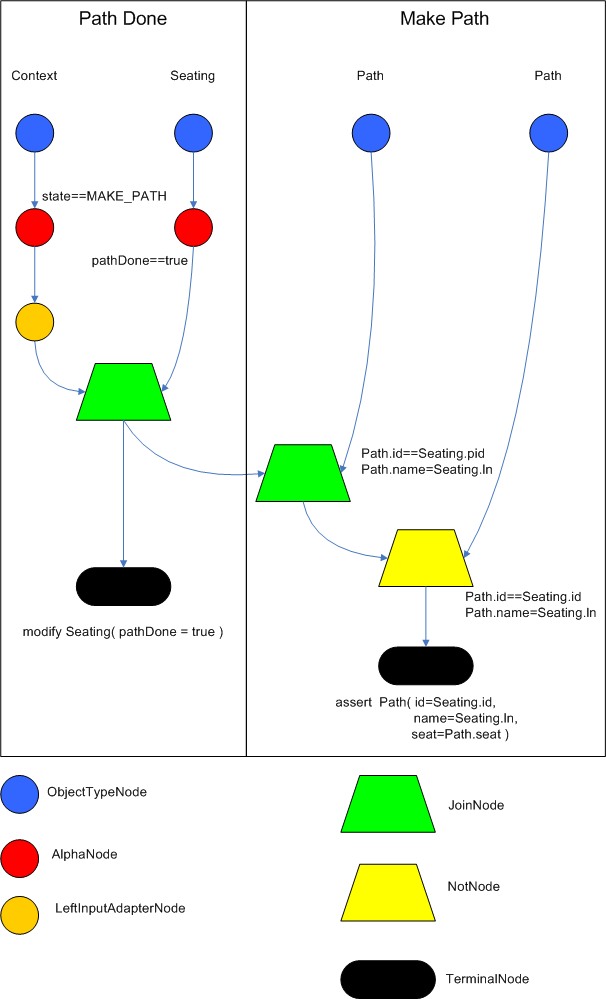
seating : Seating( pathDone == false )

then

modify( seating ) {setPathDone( true )}

modify( context ) {setState( Context.CHECK\_DONE)}

end



**Figure 9.24. 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.

**9.11.2.6. Rules "continue" and "areWeDone"**

Rule areWeDone only activates when the last seat is assigned, at which point both rules will be activated. For the same reason that makePath always wins over path Done, areWeDone will take priority over rule 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

**9.11.3. 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]

**9.12. Conway's Game Of Life**

**Name:** Conway's Game Of Life

**Main class:** org.drools.examples.conway.ConwayAgendaGroupRun

org.drools.examples.conway.ConwayRuleFlowGroupRun

**Module:** droolsjbpm-integration-examples (Note: this is in a different download, the droolsjbpm-integration download.)

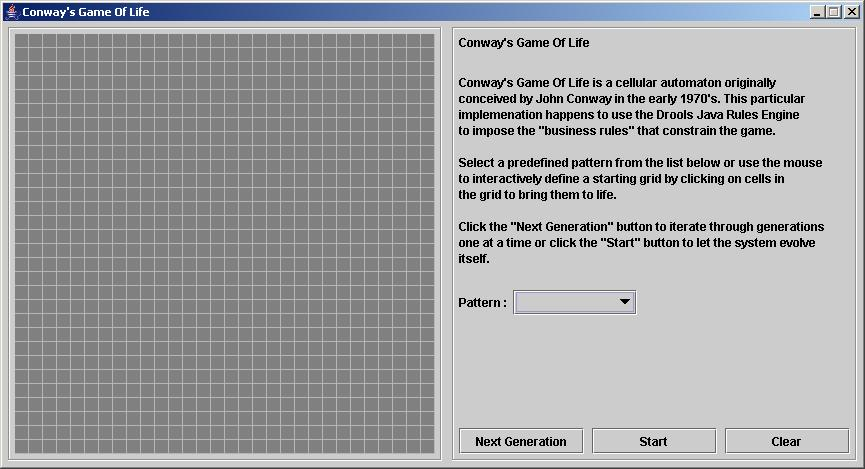
**Type:** Java application

**Rules file:** conway-ruleflow.drl conway-agendagroup.drl

**Objective:** Demonstrates 'accumulate', 'collect' and 'from'

Conway's Game Of Life, described in <http://en.wikipedia.org/wiki/Conway's_Game_of_Life> and in<http://www.math.com/students/wonders/life/life.html>, is a famous cellular automaton conceived in the early 1970's by the 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 simulation of a form of life. 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 presented 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 parts of the implementation.

We'll first introduce the grid view, shown below, designed for the visualisation of the game, showing the "arena" where the life simuation takes place. Initially the grid is empty, meaning that there are no live cells in the system. Each cell is either alive or dead, with live cells showing a green ball. Preselected patterns of live cells can be chosen from the "Pattern" drop-down list. Alternatively, individual cells can be doubled-clicked to toggle them between live and dead. It's important to understand that each cell is related to its neighboring cells, which is fundamental for the game's rules. Neighbors include not only cells to the left, right, top and bottom but also cells that are connected diagonally, so that each cell has a total of 8 neighbors. Exceptions are the four corner cells which have only three neighbors, and the cells along the four border, with five neighbors each.



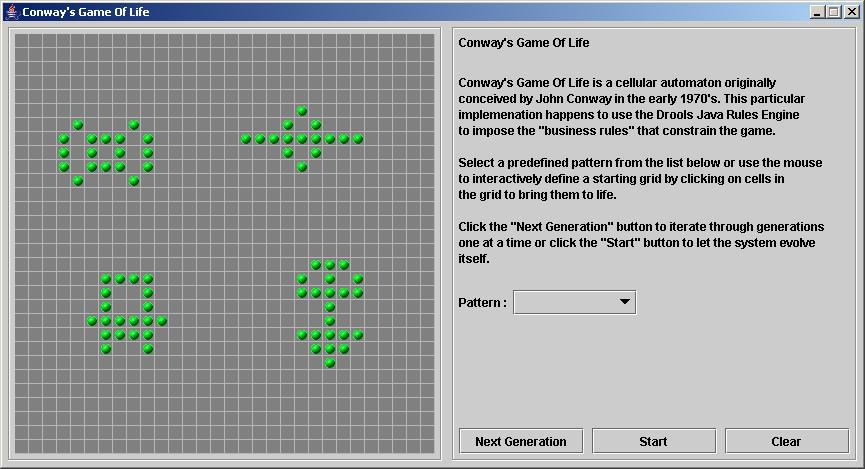
**Figure 9.25. Conway's Game of Life: Starting a new game**

So what are the basic rules that govern this game? Its goal is to show the development of a population, generation by generation. Each generation results from the preceding one, based on the simultaneous evaluation of all cells. This is the 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 screenshot 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). Step 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.



**Figure 9.26. Conway's Game of Life: 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 are able to connect the presented highlight, so that we'll just focus at a high level overview. The example has two ways to execute, one way uses Agenda Groups to manage execution flow, and the other one uses Rule Flow Groups to manage execution flow. These two versions are implemented in ConwayAgendaGroupRun and ConwayRuleFlowGroupRun, respectively. Here, we'll discuss the Rule Flow version, as it's what most people will use.

All the Cell objects are inserted into the Session and the rules in the ruleflow-group "register neighbor" are allowed to execute by the Rule Flow process. This group of four rules creates Neighbor relations between some cell and its northeastern, northern, northwestern and western neighbors. This relation is bidirectional, which takes care of the other four directions. Border cells don't need any special treatment - they simply won't be paired with neighboring cells where there isn't any. By the time all activations have fired for these rules, all cells are related to all their neighboring cells.

**Example 9.73. Conway's Game of Life: Register Cell Neighbour relations**

rule "register north east"

ruleflow-group "register neighbor"

when

$cell: Cell( $row : row, $col : col )

$northEast : Cell( row == ($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, $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, $col : col )

$northWest : Cell( row == ($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, $col : col )

$west : Cell( row == $row, col == ( $col - 1 ) )

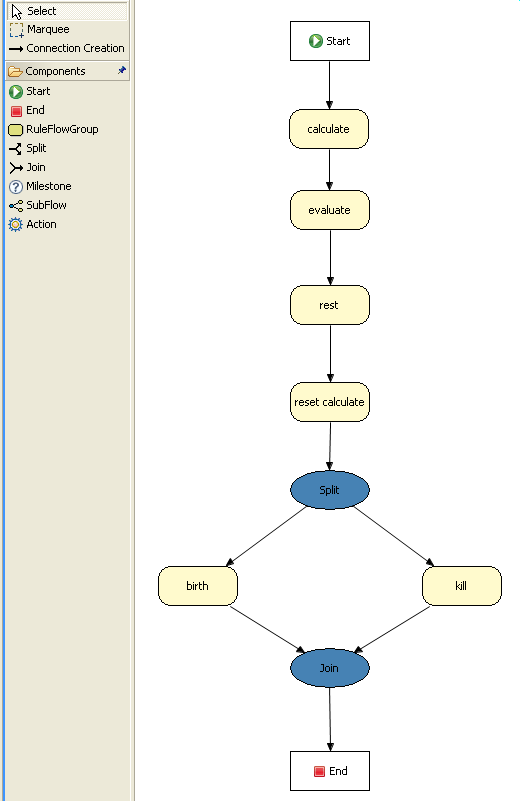
then

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 9.27. Conway's Game of Life: rule flow "Generation"**

The rule flow process first enters the "evaluate" group, which means that any active rule in the group can fire. The rules in this group apply the Game-of-Life rules discussed in the beginning of the example, determining the cells to be killed and the ones to be given life. We use the "phase" attribute to drive the reasoning of the Cell by specific groups of rules; typically the phase is tied to a Rule Flow Group in the Rule Flow process definition. Notice that it doesn't actually change the state of any Cell objectss 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, used later to control actions applied to the Cell object.

**Example 9.74. Conway's Game of Life: 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

modify( theCell ){

setPhase( Phase.KILL );

}

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

modify( theCell ){

setPhase( Phase.KILL );

}

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

modify( theCell ){

theCell.setPhase( Phase.BIRTH );

}

end

Once all Cell objects in the grid have been evaluated, we first clear any calculation activations that occured from any previous data changes. This is done via the "reset calculate" rule, which clears any activations in the "calculate" group. We then enter a split in the rule flow which allows any activations in both the "kill" and the "birth" group to fire. These rules are responsible for applying the state change.

**Example 9.75. Conway's Game of Life: 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

modify( theCell ){

setCellState( CellState.DEAD ),

setPhase( Phase.DONE );

}

end

rule "birth"

ruleflow-group "birth"

no-loop

when

theCell: Cell( phase == Phase.BIRTH )

then

modify( theCell ){

setCellState( CellState.LIVE ),

setPhase( Phase.DONE );

}

end

At this stage, a number of Cell objects have been modified with the state changed to either LIVE or DEAD. Now we get to see the power of the Neighbor facts defining the cell relations. When a cell becomes live or dead, we use the Neighbor relation to iterate over all surrounding cells, increasing or decreasing theliveNeighbor count. Any cell that has its count changed is also set to to the EVALUATE phase, to make sure it is included in the reasoning during the evaluation stage of the Rule Flow Process. Notice that we don't have to do any iteration ourselves; simply by applying the relations in the rules we make the rule engine do all the hard work for us, with a minimal amount of code. Once the live count has been determined and set for all cells, the Rule Flow Process comes to and end. If the user has initially clicked the "Start" button, the engine will restart the rule flow; otherwise the user may request another generation.

**Example 9.76. Conway's Game of Life: 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

modify( $neighbor ){

setLiveNeighbors( $neighbor.getLiveNeighbors() + 1 ),

setPhase( Phase.EVALUATE );

}

end

rule "Calculate Dead"

ruleflow-group "calculate"

lock-on-active

when

theCell: Cell( cellState == CellState.DEAD )

Neighbor( cell == theCell, $neighbor : neighbor )

then

modify( $neighbor ){

setLiveNeighbors( $neighbor.getLiveNeighbors() - 1 ),

setPhase( Phase.EVALUATE );

}

end