Red Hat Decision Manager 7.2

Getting started with decision services

Last Updated: 2020-05-26

Red Hat Decision Manager 7.2 Getting started with decision services

Red Hat Customer Content Services
brms-docs@redhat.com

Legal Notice

Copyright © 2020 Red Hat, Inc.

The text of and illustrations in this document are licensed by Red Hat under a Creative Commons Attribution–Share Alike 3.0 Unported license ("CC-BY-SA"). An explanation of CC-BY-SA is available at
http://creativecommons.org/licenses/by-sa/3.0/
. In accordance with CC-BY-SA, if you distribute this document or an adaptation of it, you must provide the URL for the original version.

Red Hat, as the licensor of this document, waives the right to enforce, and agrees not to assert, Section 4d of CC-BY-SA to the fullest extent permitted by applicable law.

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

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

Java ® is a registered trademark of Oracle and/or its affiliates.

XFS ® is a trademark of Silicon Graphics International Corp. or its subsidiaries in the United States and/or other countries.

MySQL ® is a registered trademark of MySQL AB in the United States, the European Union and other countries.

Node.js ® is an official trademark of Joyent. Red Hat is not formally related to or endorsed by the official Joyent Node.js open source or commercial project.

The OpenStack ® Word Mark and OpenStack logo are either registered trademarks/service marks or trademarks/service marks of the OpenStack Foundation, in the United States and other countries and are used with the OpenStack Foundation's permission. We are not affiliated with, endorsed or sponsored by the OpenStack Foundation, or the OpenStack community.

All other trademarks are the property of their respective owners.

Abstract

This document describes how to create and test an example traffic violation decision service in Red Hat Decision Manager 7.2.

Table of Contents

PREFACE 4

CHAPTER 1. CREATING THE TRAFFIC VIOLATIONS PROJECT 5

CHAPTER 2. DATA OBJECTS 6
2.1. CREATING THE VIOLATION DATA OBJECT 7
2.1.1. Adding the Violation data object constraints 7
2.2. CREATING THE DRIVER DATA OBJECT 9
2.2.1. Adding the Driver data object constraints 10

CHAPTER 3. GUIDED RULES 13
3.1. CREATING THE DRIVER LICENSE SUSPENSION RULE 13
3.2. SETTING THE SUSPENSION RULE CONDITIONS 14
3.3. SETTING THE SUSPENSION RULE ACTIONS 16

CHAPTER 4. GUIDED DECISION TABLES 18
4.1. CREATING A TRAFFIC VIOLATION GUIDED DECISION TABLE 18
4.1.1. Inserting Violation Type columns 19
4.1.2. Inserting Fine Amount and Points columns 23
4.1.3. Inserting guided decision table rows 24

CHAPTER 5. TEST SCENARIOS 26
5.1. TESTING THE SPEED LIMIT SCENARIO 26
5.2. TESTING THE DRIVER LICENSE SUSPENSION SCENARIO 28
5.3. TESTING THE MULTIPLE VIOLATIONS SCENARIO 30

CHAPTER 6. EXAMPLE PROJECTS AND BUSINESS ASSETS IN DECISION CENTRAL 32
6.1. ACCESSING EXAMPLE PROJECTS AND BUSINESS ASSETS IN DECISION CENTRAL 32
6.2. EXECUTING RULES 32

CHAPTER 7. EXAMPLE DECISIONS IN RED HAT DECISION MANAGER FOR AN IDE 38
7.1. IMPORTING AND EXECUTING RED HAT DECISION MANAGER EXAMPLE DECISIONS IN AN IDE 38
7.2. HELLO WORLD EXAMPLE DECISIONS (BASIC RULES AND DEBUGGING) 41
7.3. STATE EXAMPLE DECISIONS (FORWARD CHAINING AND CONFLICT RESOLUTION) 44
State example using salience 47
State example using agenda groups 50
Dynamic facts in the State example 51
7.4. FIBONACCI EXAMPLE DECISIONS (RECURSION AND CONFLICT RESOLUTION) 52
7.5. PRICING EXAMPLE DECISIONS (DECISION TABLES) 58
Spreadsheet decision table setup 59
Base pricing rules 62
Promotional discount rules 63
7.6. PET STORE EXAMPLE DECISIONS (AGENDA GROUPS, GLOBAL VARIABLES, CALLBACKS, AND GUI INTEGRATION) 63
Rule execution behavior in the Pet Store example 64
Pet Store rule file imports, global variables, and Java functions 65
Pet Store rules with agenda groups 67
Pet Store example execution 71
7.7. HONEST POLITICIAN EXAMPLE DECISIONS (TRUTH MAINTENANCE AND SALIENCE) 75
Politician and Hope classes 76
Rule definitions for politician honesty 77
Example execution and audit trail 78
7.8. SUDOKU EXAMPLE DECISIONS (COMPLEX PATTERN MATCHING, CALLBACKS, AND GUI

INTEGRATION) 81
Sudoku example execution and interaction 81
Sudoku example classes 87
Sudoku validation rules (validate.drl) 87
Sudoku solving rules (sudoku.drl) 88
7.9. CONWAY'S GAME OF LIFE EXAMPLE DECISIONS (RULEFLOW GROUPS AND GUI INTEGRATION) 95
Conway example execution and interaction 96
Conway example rules with ruleflow groups 97
7.10. HOUSE OF DOOM EXAMPLE DECISIONS (BACKWARD CHAINING AND RECURSION) 101
Recursive query and related rules 105
Transitive closure rule 106
Reactive query rule 107
Queries with unbound arguments in rules 108

APPENDIX A. VERSIONING INFORMATION 110

PREFACE

As a business rules developer, you can use Decision Central in Red Hat Decision Manager to design a variety of decision services. Red Hat Decision Manager provides example projects with example decisions directly in Decision Central as a reference and example decisions distributed as Java classes that you can import into your integrated development environment (IDE) for external testing.

This document focuses on how to create and test a new project in Decision Central for an example traffic violation decision service.

Prerequisites

- Red Hat JBoss Enterprise Application Platform 7.2 is installed. For installation information, see *Red Hat JBoss EAP 7.2.0 Installation Guide* .
- Red Hat Decision Manager is installed and configured with Decision Server. For more information, see *Installing and configuring Red Hat Decision Manager on Red Hat JBoss EAP* .
- Red Hat Decision Manager is running and you can log in to Decision Central with the **developer** role. For more information, see *Planning a Red Hat Decision Manager installation* .

CHAPTER 1. CREATING THE TRAFFIC VIOLATIONS PROJECT

A project is a container for assets such as data objects, guided decision tables, and guided rules.

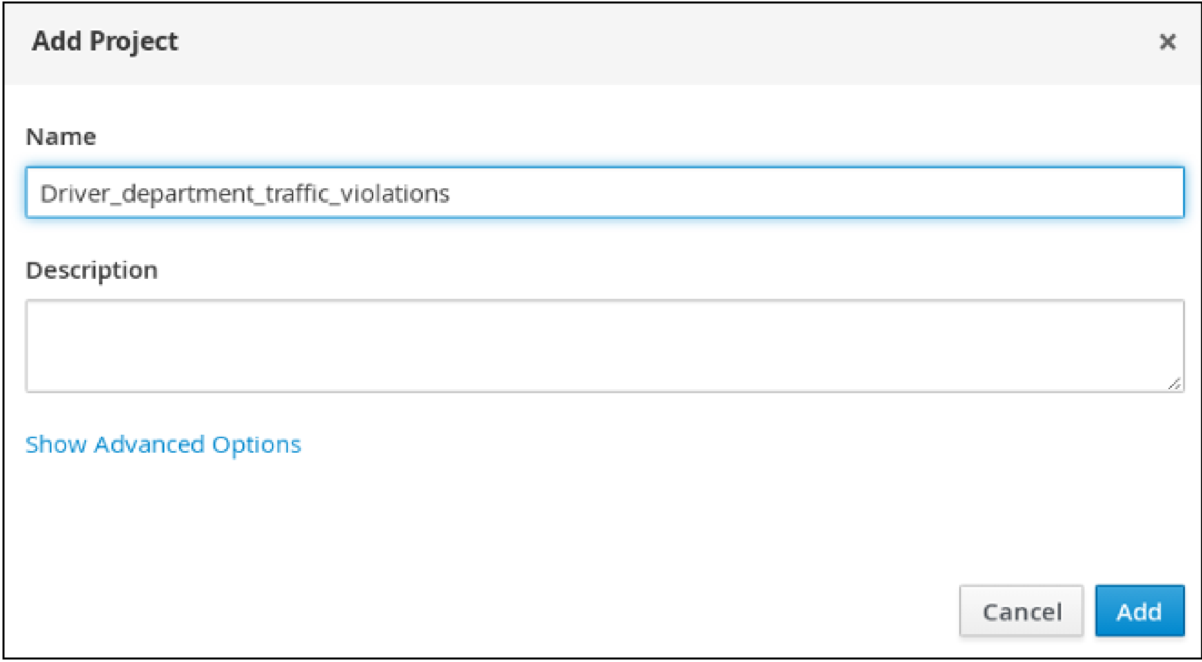
Procedure

1. Log in to Decision Central.
2. Go to **Menu → Design → Projects**.
   Red Hat Decision Manager provides a default space called **MySpace**, as shown in the following image. You can use the default space to create and test example projects.

   Figure 1.1. Default space

3. Click **Add Project**.
4. Enter **Driver_department_traffic_violations** in the **Name** field.
5. Click **Add**.

   Figure 1.2. Add Project window

The **Assets** view of the project opens.

CHAPTER 2. DATA OBJECTS

Data objects are the building blocks for the rule assets that you create. Data objects are custom data types implemented as Java objects in specified packages of your project. For example, you might create a **Person** object with data fields **Name**, **Address**, and **DateOfBirth** to specify personal details for loan application rules. These custom data types determine what data your assets and your decision services are based on.

The following tables show the **Violation** and **Driver** data objects that you will create for this project.

Table 2.1. Violation data object

ID	Label	Type
code	Code	String
points	Points	Integer
violationDate	Violation Date	Date
type	Type	String
fineAmount	Fine Amount	Double
speedLimit	Speed Limit	Integer
actualSpeed	Actual Speed	Integer

Table 2.2. Driver data object

ID	Label	Type
name	Name	String
age	Age	Integer
state	State	String
city	City	String
violations	Violations	Violation (org.jboss.example.traffic_violations.Violation) Note: The **Violations** field is set to **List** to hold multiple items for the given type.
fineAmount	Fine Amount	Double

ID	Label	Type
totalPoints	Total Points	Integer
reason	Reason	String

2.1. CREATING THE VIOLATION DATA OBJECT

The **Violation** data object contains data fields based on violation details, such as the **Violation Date**, **Fine Amount**, and **Speed Limit**.

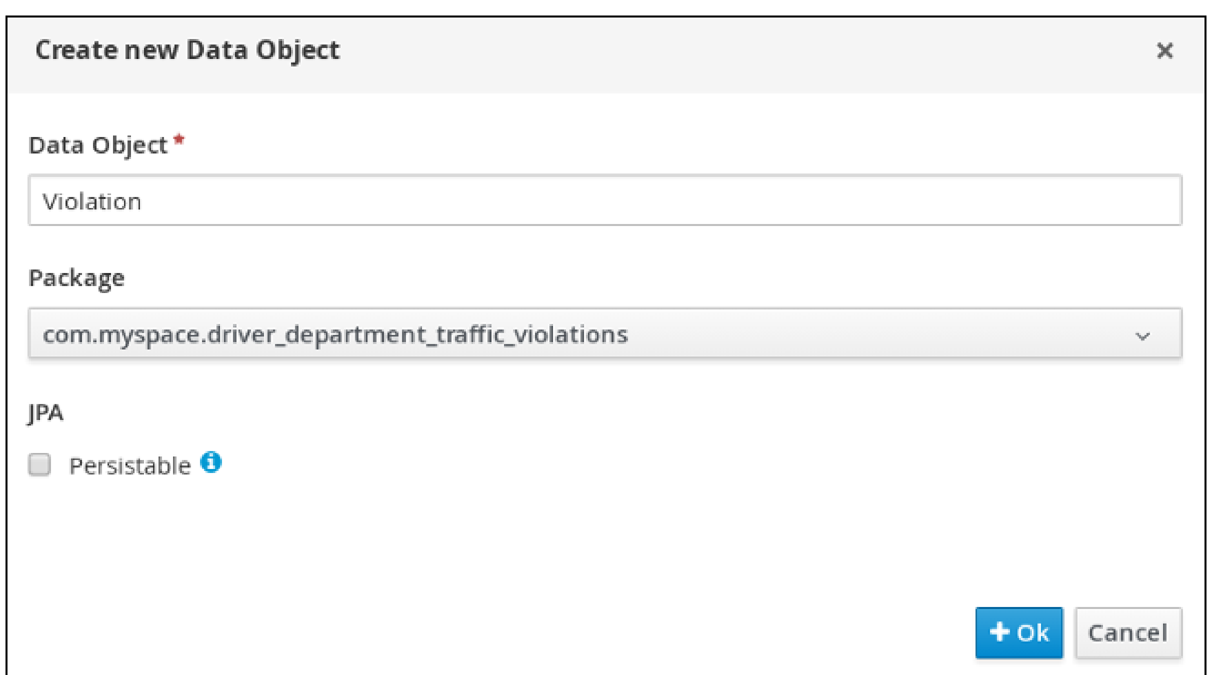
Prerequisites

You have created the **Driver_department_traffic_violations** project.

Procedure

1. Click **Add Asset → Data Object**.
2. In the **Create new Data Object** wizard, enter the following values:
   - **Data Object**: **Violation**
   - **Package**: select **com.myspace.driver_department_traffic_violations**
3. Click **Ok**.

   Figure 2.1. Create new Data Object window



2.1.1. Adding the Violation data object constraints

Populate the **Violation** data object fields with the constraints that you will select when you define your rules.

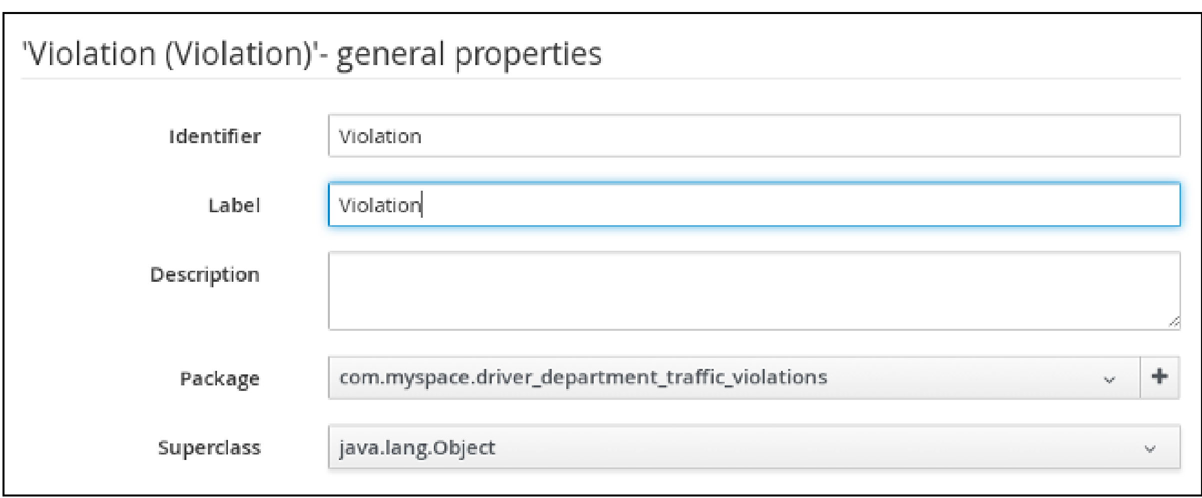
Prerequisites

You have created the **Violation** data object.

Procedure

1. In the **'Violation'-general properties** section, enter **Violation** in the **Label** field.

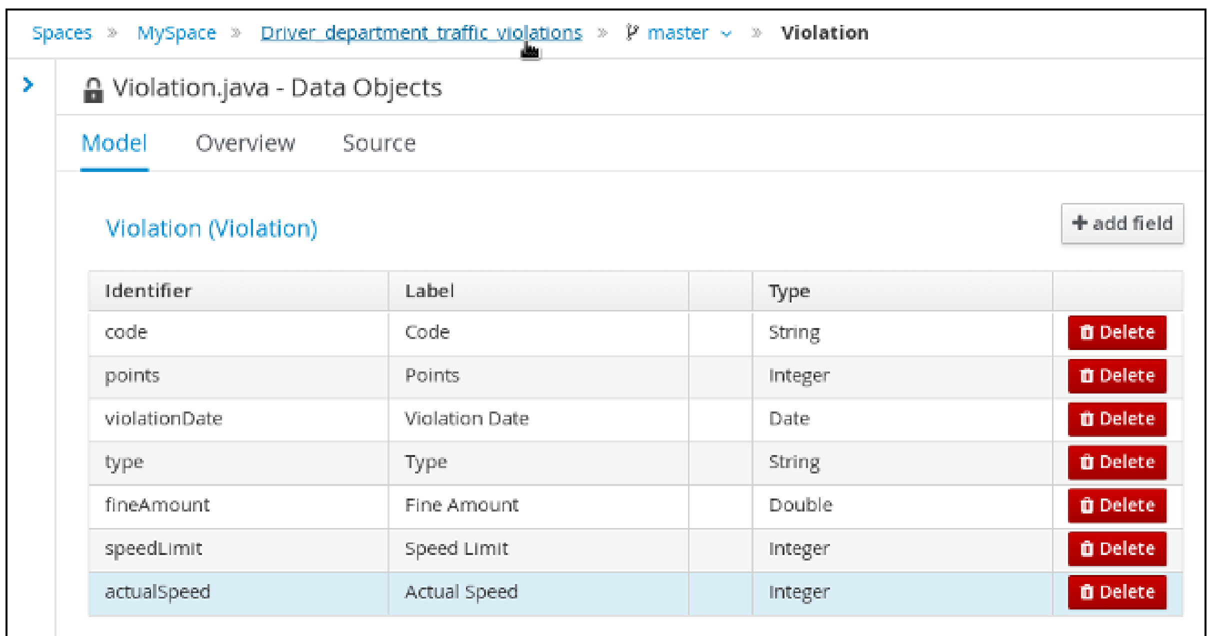
   Figure 2.2. General properties

2. Click **+ add field**.
3. Enter the following values:
   - **Id**: **code**
   - **Label**: **Code**
   - **Type**: **String**
4. Click **Create and continue**, then enter the following values:
   - **Id**: **points**
   - **Label**: **Points**
   - **Type**: **Integer**
5. Click **Create and continue**, then enter the following values:
   - **Id**: **violationDate**
   - **Label**: **Violation Date**
   - **Type**: **Date**
6. Click **Create and continue**, then enter the following values:
   - **Id**: **type**
   - **Label**: **Type**
   - **Type**: **String**

7. Click **Create and continue**, then enter the following values:
    - Id: **fineAmount**
    - Label: **Fine Amount**
    - Type: **Double**
8. Click **Create and continue**, then enter the following values:
    - Id: **speedLimit**
    - Label: **Speed Limit**
    - Type: **Integer**
9. Click **Create and continue**, then enter the following values:
    - Id: **actualSpeed**
    - Label: **Actual Speed**
    - Type: **Integer**
10. Click **Create**.
11. Click **Save**, and then click **Save** to confirm your changes.
12. Click the **Driver_department_traffic_violations** label to return to the **Assets** view of the project.

Figure 2.3. Violation data object fields



2.2. CREATING THE DRIVER DATA OBJECT

The **Driver** data object contains data fields based on driver details, such as the **Name**, **Age**, and **Total Points** of the driver.

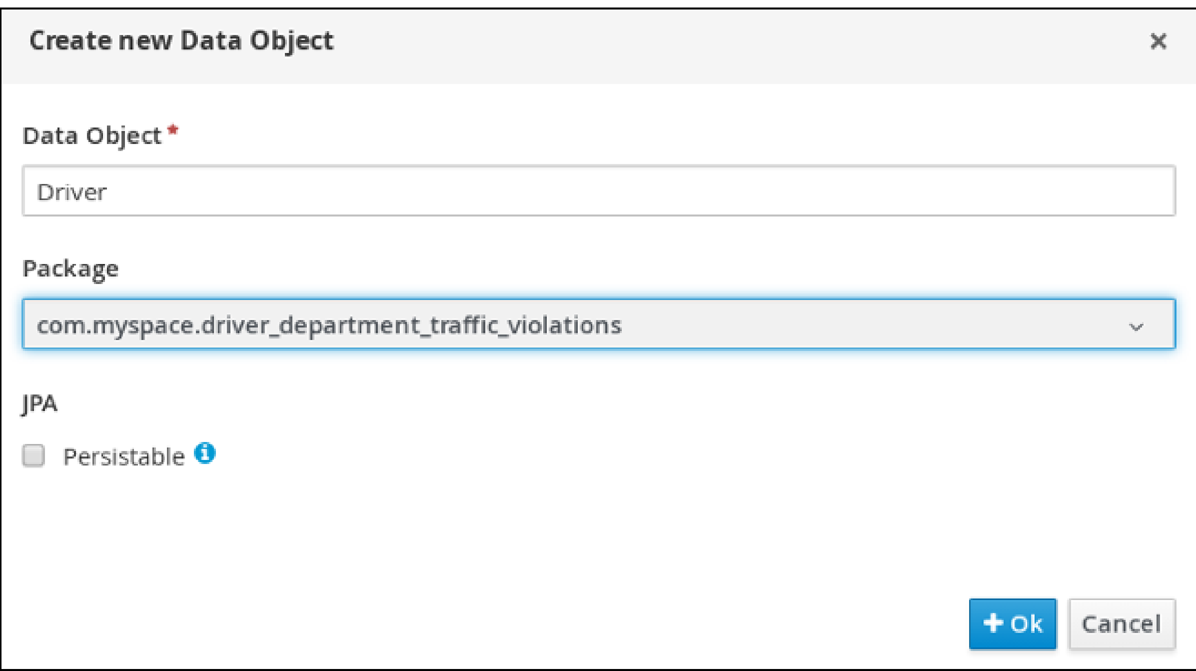
Prerequisites

You have created the **Driver_department_traffic_violations** project.

Procedure

1. Go to **Menu** → **Design** → **Projects** → **Driver_department_traffic_violations**.
2. Click **Add Asset** → **Data Object**.
3. In the **Create new Data Object** wizard, enter the following values:
   - **Data Object**: **Driver**
   - **Package**: select **com.myspace.driver_department_traffic_violations**
4. Click **Ok**.

   Figure 2.4. Create new Data Object window

2.2.1. Adding the Driver data object constraints

Populate the **Driver** data object fields with the constraints that you will select when you define your rules.

Prerequisites

You have created the **Driver** data object.

Procedure

1. In the **'Driver'-general properties** section, enter **Driver** in the **Label** field.
2. Click **+ add field**.
3. Enter the following values:
   - **Id**: **name**
   - **Label**: **Full Name**

- Type: **String**

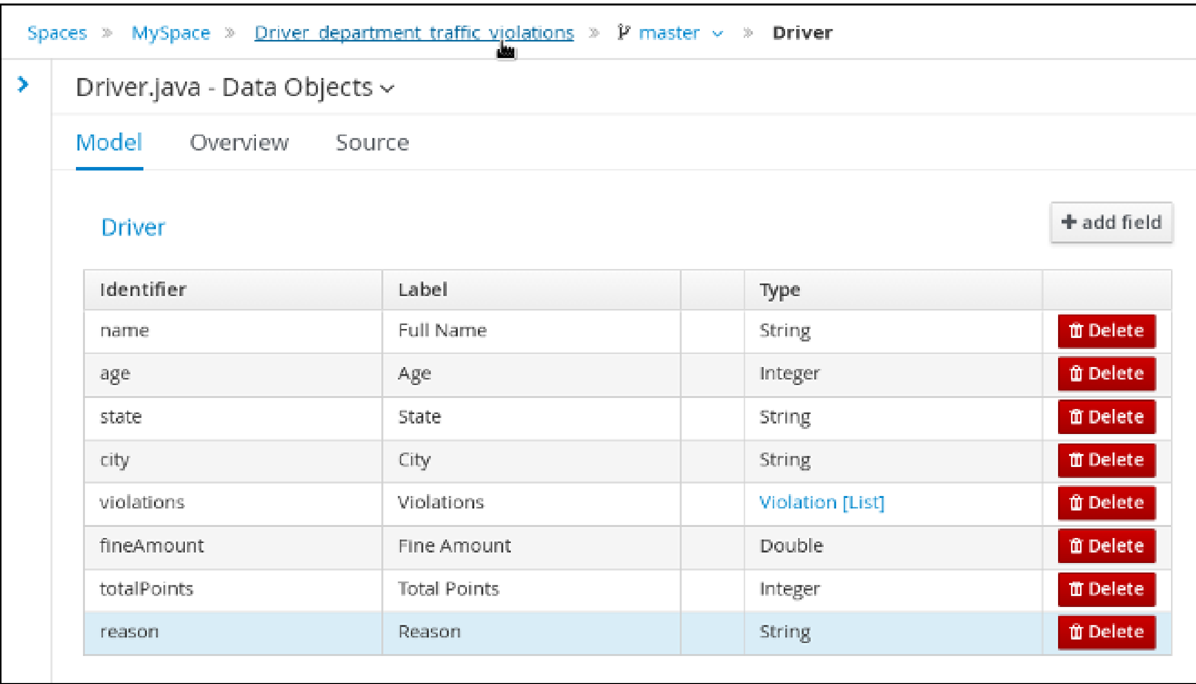
4. Click **Create and continue**, then enter the following values:
   - Id: **age**
   - Label: **Age**
   - Type: **Integer**
5. Click **Create and continue**, then enter the following values:
   - Id: **state**
   - Label: **State**
   - Type: **String**
6. Click **Create and continue**, then enter the following values:
   - Id: **city**
   - Label: **City**
   - Type: **String**
7. Click **Create and continue**, then enter the following values:
   - Id: **violations**
   - Label: **Violations**
   - Type: **Violation(com.myspace.driver_department_traffic_violations.Violation)**
   - **List**: Select this check box to enable the field to hold multiple items for the specified type.
8. Click **Create and continue**, then enter the following values:
   - Id: **fineAmount**
   - Label: **Fine Amount**
   - Type: **Double**
9. Click **Create and continue**, then enter the following values:
   - Id: **totalPoints**
   - Label: **Total Points**
   - Type: **Integer**
10. Click **Create and continue**, then enter the following values:
    - Id: **reason**
    - Label: **Reason**
    - Type: **String**

11. Click **Create**.

12. Click **Save**, and then click **Save** to confirm your changes.

13. Click the **Driver_department_traffic_violations** label to return to the **Assets** view of the project.

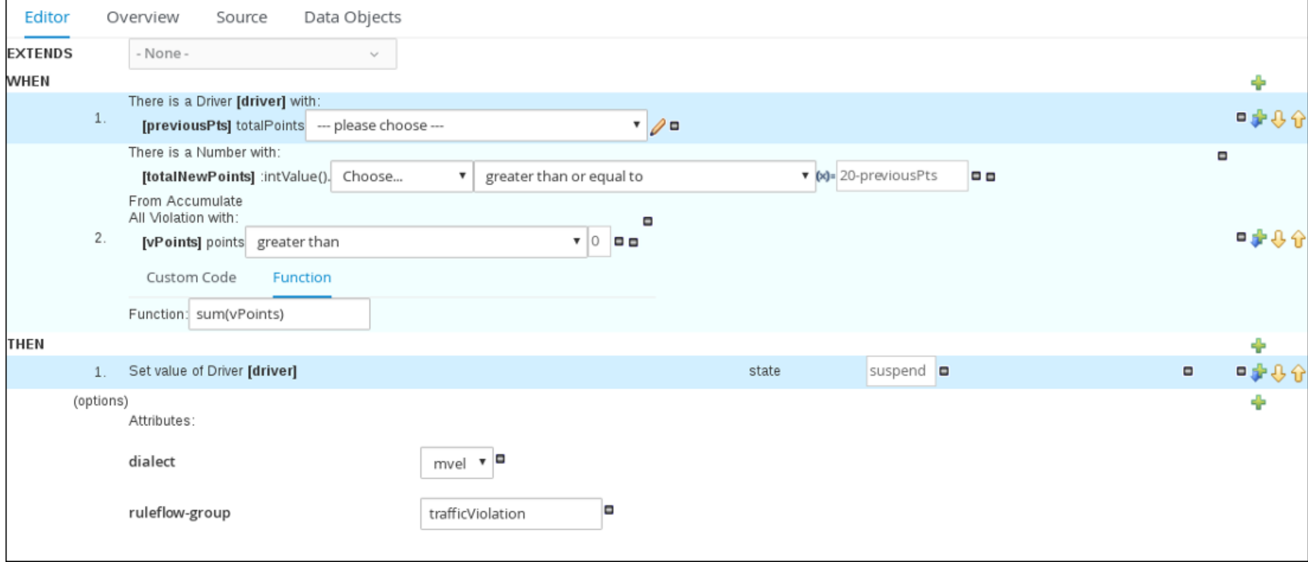
Figure 2.5. Driver data object fields



CHAPTER 3. GUIDED RULES

Guided rules are business rules that you can create in a UI-based Guided Rules designer that leads you through the rule creation process. The rule designer provides fields and options for acceptable input based on the object model of the rule being edited. All data objects related to the rule must be in the same project package as the rule. Assets in the same package are imported by default. You can use the **Data Objects** tab of the rule designer to verify that all required data objects are listed or to import any other needed data objects.

Figure 3.1. The Guided Rule designer



3.1. CREATING THE DRIVER LICENSE SUSPENSION RULE

Create the Driver license suspension rule using the guided rules designer.

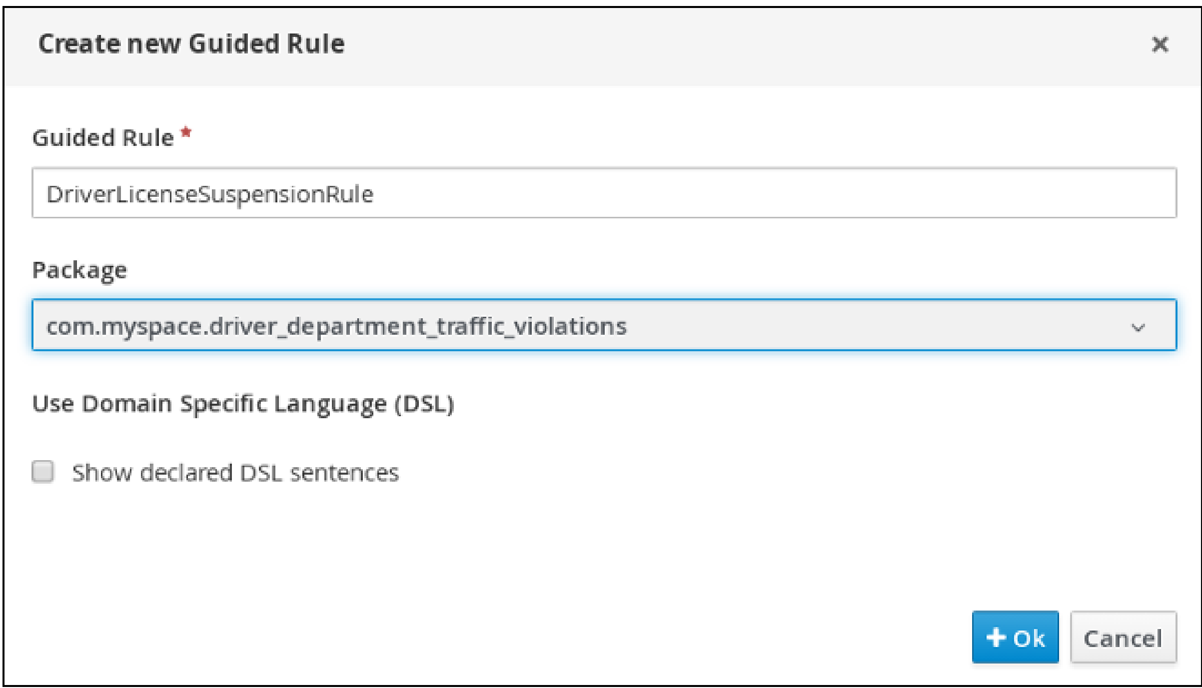
Prerequisites

You have created both the **Violation** and **Driver** data objects.

Procedure

1. Click **Menu → Design → Projects**, then **Driver_department_traffic_violations**.
2. Click **Add Asset → Guided Rule**, then enter the following values:
   - Guided Rule: **DriverLicenseSuspensionRule**
   - Package: **com.myspace.driver_department_traffic_violations**
3. Click **Ok** to open the **Guided Rule designer**.

Figure 3.2. Create new guided rule window



3.2. SETTING THE SUSPENSION RULE CONDITIONS

Set the **Suspension** rule conditions that are used to determine the driver's violation.

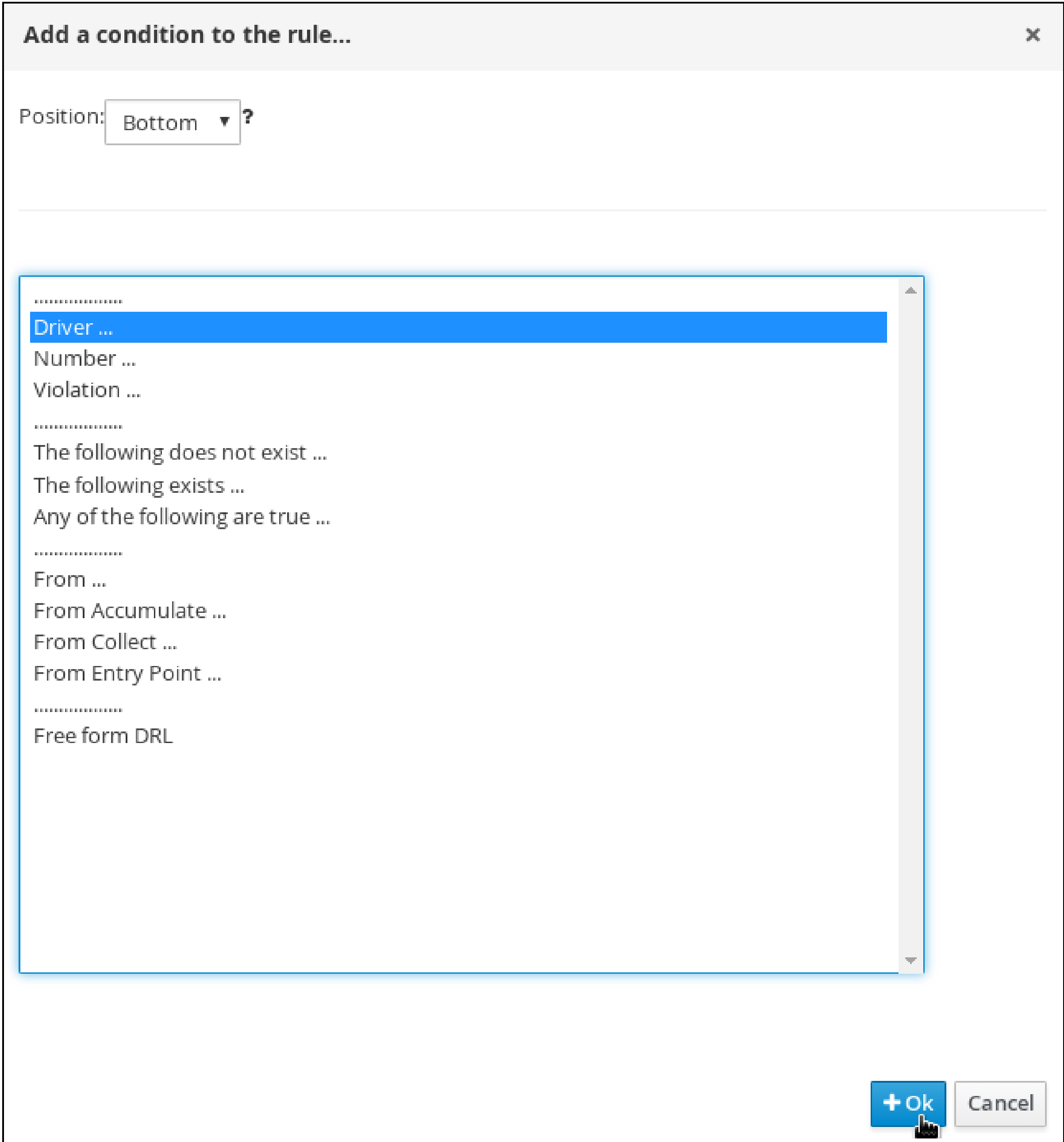
Prerequisites

You have created the Driver License Suspension rule.

Procedure

1. Click ( ) next to the **WHEN** label to open the **Add a condition to the rule**window.
2. Select **Driver** and click **Ok**.

Figure 3.3. Create new guided rule window



3. Click the **There is a Driver** label to open the **Modify constraints for Driver** window.

4. Enter **driver** in the **Variable name** field and click **Set**.

5. Click **There is a Driver[driver]** and click **Expression editor**.

6. Click **[not bound]** to open the **Expression editor**.

7. In the **Bind the Expression to a new variable** field, enter: **previousPts** and click **Set**.

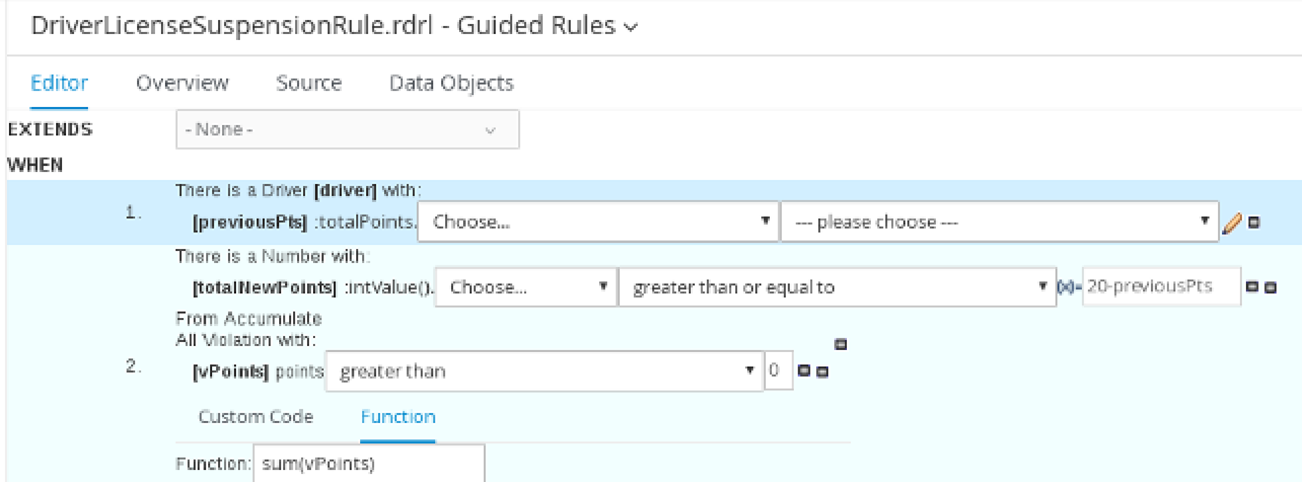
8. From the **Choose** menu, select **totalPoints**.

9. Click (✚) next to line 1 (the **previousPts** label) to open the **Add a condition to the rule** window.

10. Select **From Accumulate** and click **Ok**.

11. Click **click to add pattern** above the **From Accumulate** label and select **Number** from the **choose fact type** menu.
12. Click **There is a number** label to open the **Modify contraints for Number** window.
13. Click **Expression editor** and select **intValue()** from the **[not bound]: Choose** menu.
14. Click **[not bound]** to open the **Expression editor**.
15. In the **Bind the Expression to a new variable** field, enter: **totalNewPoints** and click **Set**.
16. Click **click to add pattern** and select **Violation** from the **choose fact type** menu.
17. Click **All Violation with:** to open the **Modify contraints for Violation** window and select **points** from the **Add a restriction on a field** menu.
18. Click **please choose** next to the **points** label and select **greater than**.
19. Click ( ), and then click **Literal value**.
20. Click the **points** label to open the **Add a field** window.
21. Enter **vPoints** and click **Set**.
22. In the **Function** field, enter **sum(vPoints)**.
23. Select **greater than or equal to** from the **totalNewPoints → please choose** menu.
24. Click ( ), click **New formula**, and enter **20-previousPts** in the new field.
25. Click **Save**, and then click **Save** to confirm your changes.

Figure 3.4. Suspension Rule conditions



3.3. SETTING THE SUSPENSION RULE ACTIONS

Set the **Suspension** rule actions that are used to determine a driver's penalties, including points and fine amounts. These values are based on the Suspension rule conditions.

Prerequisites

You have set the Suspension rule conditions.

Procedure

1. Click **(show options...)** next to the **THEN** label.

Figure 3.5. show options selection

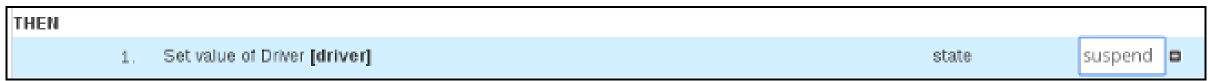
2. Click ( ) next to the **THEN** label and select **Change field values of driver**, and click **Ok**.

3. Click the **Set value of Driver [driver]** field and select **state** from the **Add field** menu.

4. Click ( ) next to **state** in the **Set value of Driver [driver]** section to open the **Field value** window.

5. Click **Literal value** and enter **suspend** in the new field.

Figure 3.6. New field



6. Click ( ) next to the **(options)** label below the **Set value of Driver [driver]** section.

7. From the **Add an option to the rule** window, select the **ruleflow-group** option from the **Attribute** menu.

8. Enter **trafficViolation** in to the **ruleflow-group** field.

9. Click **Save**, and then click **Save** to confirm your changes.

Figure 3.7. Suspension rule actions

CHAPTER 4. GUIDED DECISION TABLES

Guided decision tables are a wizard-led alternative to uploaded decision table spreadsheets for defining business rules in a tabular format. With guided decision tables, you are led by a UI-based wizard in Decision Central that helps you define rule attributes, metadata, conditions, and actions based on specified data objects in your project. After you create your guided decision tables, the rules you defined are compiled into Drools Rule Language (DRL) rules as with all other rule assets.

All data objects related to a guided decision table must be in the same project package as the guided decision table. Assets in the same package are imported by default. After you create the necessary data objects and the guided decision table, you can use the **Data Objects** tab of the guided decision tables designer to verify that all required data objects are listed or to import other existing data objects by adding a **New item**.

4.1. CREATING A TRAFFIC VIOLATION GUIDED DECISION TABLE

Use the Guided Decision Table designer to create the traffic violation guided decision table, which specifies the driver's specific violation and the resulting fine and points.

Prerequisites

You have created both the **Violation** and **Driver** data objects.

Procedure

1. Click **Menu → Design → Projects**, then **Driver_department_traffic_violations**.
2. Click **Add Asset → Guided Decision Table**, then enter the following values:
   - Guided Decision Table: **SpeedViolationRule**
   - Package: **com.myspace.driver_department_traffic_violations**
3. Select **Unique Hit** from the **Hit Policy** menu.
4. Select **Extended entry, values defined in table body** in the **Table format** section.
5. Click **Ok** to open the **Create new Guided Decision Table** window.

Figure 4.1. Guided Decision Table window

Create new Guided Decision Table

Guided Decision Table*

SpeedViolationRule

Package

com.myspace.driver_department_traffic_violations

Use Wizard

Hit Policy:

Unique Hit

Unique Hit

With unique hit policy each row has to be unique meaning there can be no overlap. There can never be a situation where two rows can fire, if there is the Verification feature warns about this on development time.

Table Format:

Extended entry, values defined in table body

Limited entry, values defined in columns

+ Ok Cancel

4.1.1. Inserting Violation Type columns

The **Violation Type** column contains the violation details such as the driver's speed and if the driver was under the influence of drugs or alcohol.

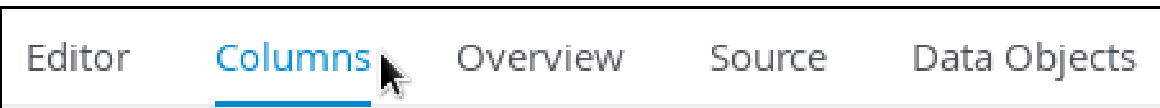
Prerequisites

You have created the traffic violation guided decision table.

Procedure

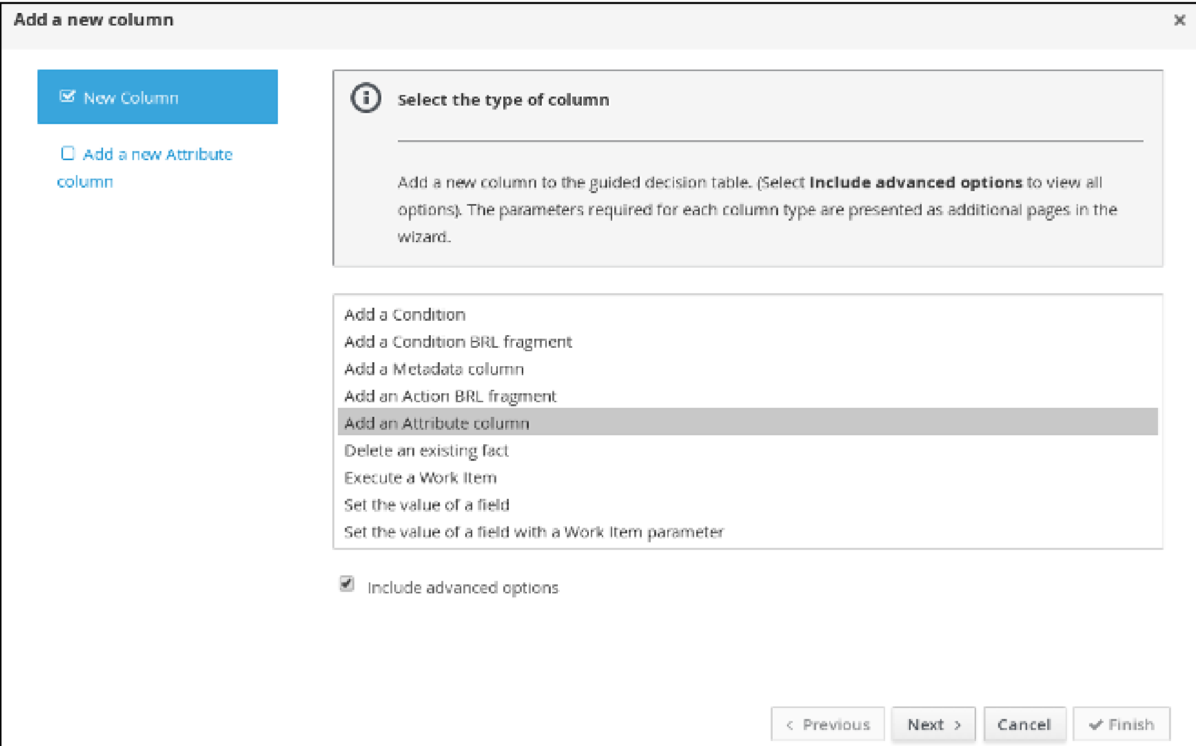
1. Click **Columns → Insert Column** and then select **Include advanced options**.

   Figure 4.2. Column tab

2. Select **Add an Attribute column**and click **Next**.

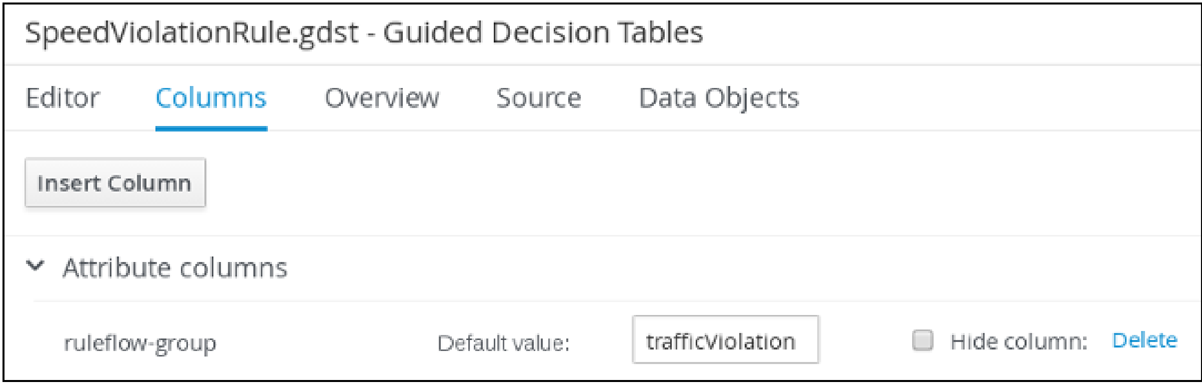
Figure 4.3. Add a new column window



3. Select **Ruleflow-group** and click **Finish**.

4. Expand **Attribute columns** and enter **trafficViolation** in the **Default value** field.

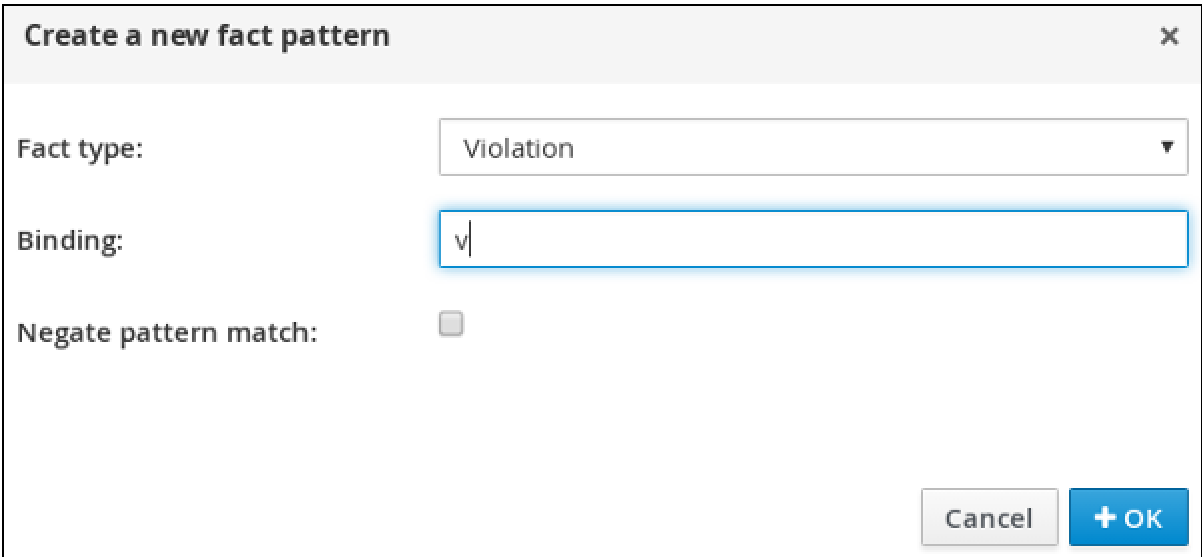
Figure 4.4. Attribute columns window



5. Click **Insert Column**, select **Add a Condition → Pattern → +Create a new Fact Pattern**

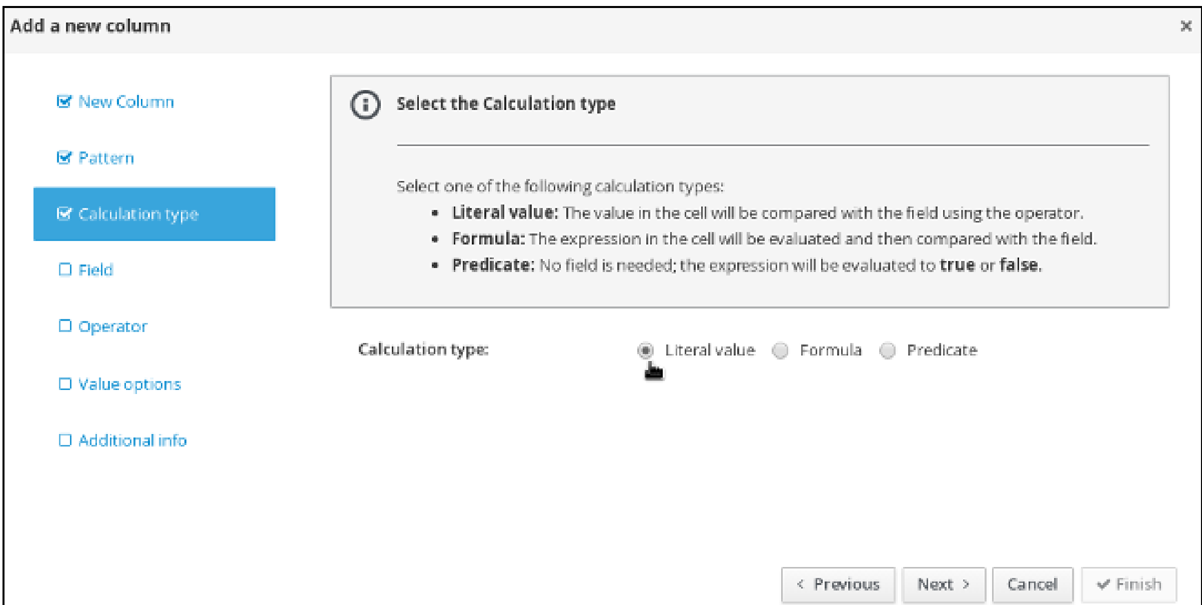
6. Select **Violation** from the **Fact type** menu, enter **v** in the **Binding** field, and click **OK**.

Figure 4.5. Create a new fact pattern window



7. Select **Calculation type → Literal value**.

Figure 4.6. Calculation type options



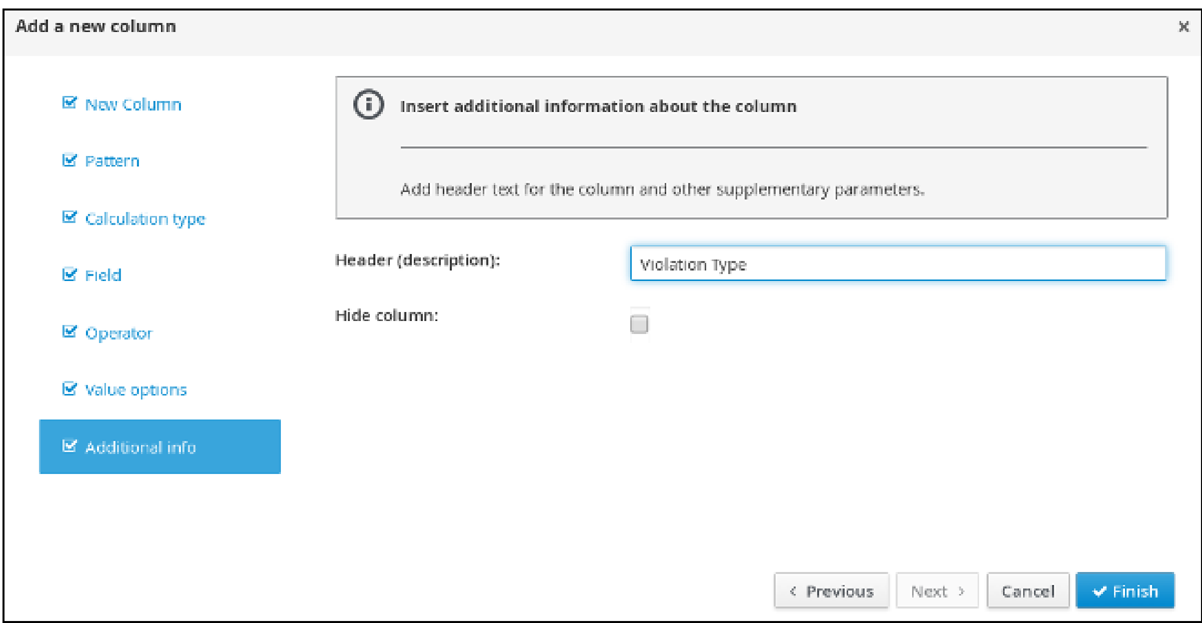
8. Select **Field** and then select **type** from the **Field** menu.

9. Select **Operator** and then **equal to** from the **Operator** menu.

10. Select **Value options** and enter **Speed,Driving while intoxicated,DWI=Driving while under the influence of drugs** in the **Value list (optional)** field.

11. Select **Additional info**, enter **Violation Type** in the **Header (description)** field, and click **Finish**.

Figure 4.7. Violation Type header



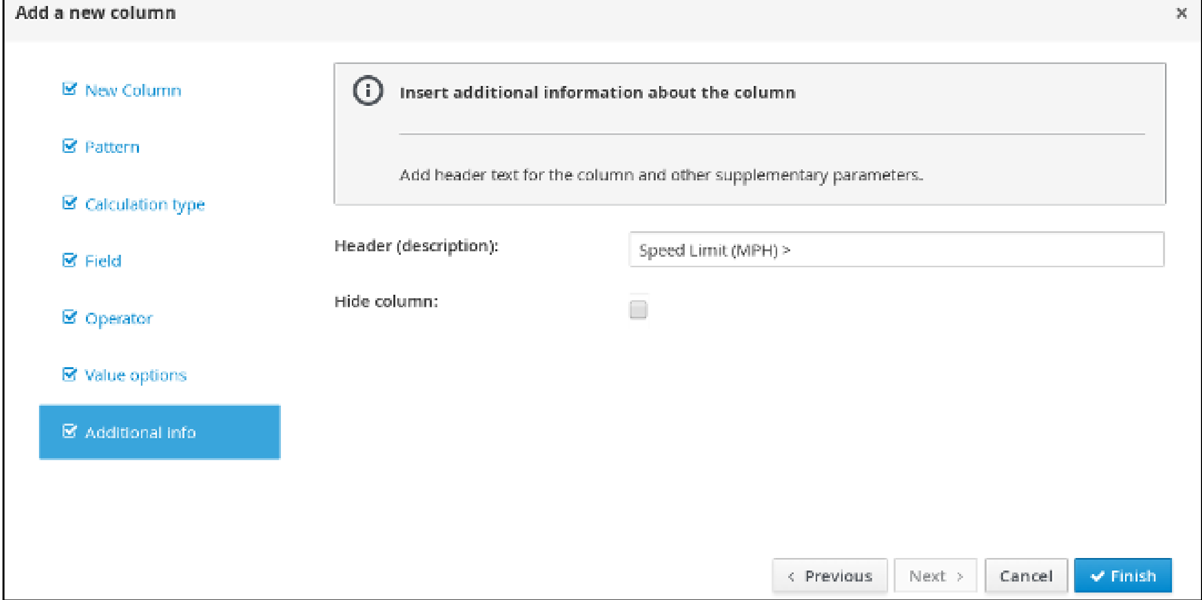
12. Click **Insert Column**, select **Add a Condition → Pattern**, and select **Violation[v]** from the **Pattern** menu.

13. Select **Calculation type → Predicate → Field** and enter **actualSpeed-speedLimit > $param**.

14. Select **Value options**, then select **Additional info** and enter **Speed Limit (MPH) >** in the **Header (description)** field.

15. Click **Finish**.

Figure 4.8. Speed Limit (MPH) > header



16. Click **Insert Column**, select **Add a Condition → Pattern**, and select **Violation[v]** from the **Pattern** menu.

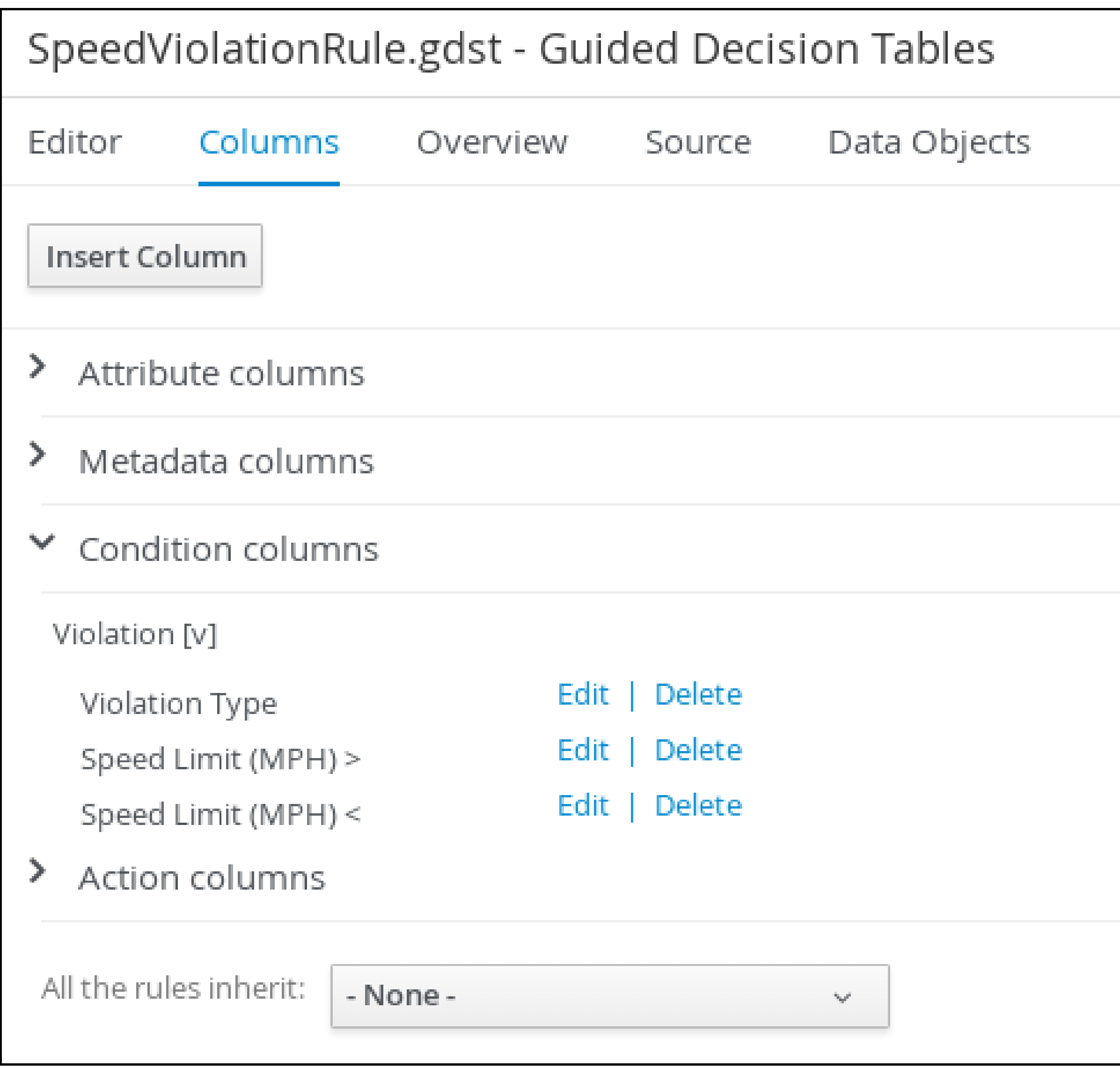
17. Select **Calculation type → Predicate**.

18. Select **Field** and enter **actualSpeed-speedLimit < $param** in the **Field** field.

19. Select **Operator**, select **Value options**, then select **Additional info**.

20. Enter **Speed Limit (MPH) <** in the **Header (description)** field and click **Finish**.

Figure 4.9. Condition columns



4.1.2. Inserting **Fine Amount and Points** columns

The **Fine Amount** and **Points** columns contain the fines and points based on the corresponding **Violation Type** field values.

Prerequisites

You have inserted the **Violation Type** column in to the traffic violation guided decision table.

Procedure

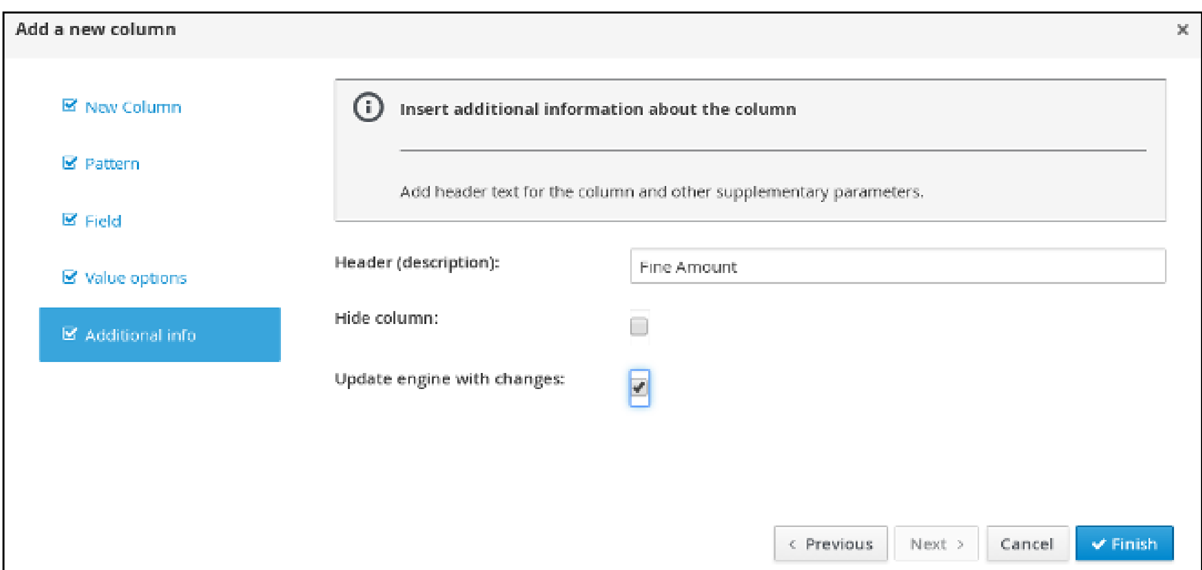
1. Click **Insert Column**, select **Set the value of a field→ Pattern**, and select **Violation[v]** from the **Pattern** menu.

2. Select **Field** and then **fineAmount** from the **Field** menu.

3. Select **Value options**, and then select **Additional info**, and enter **Fine Amount** in the **Header (description)** field.

4. Select the **Update engine with changes** option and click **Finish**.

Figure 4.10. Fine Amount header



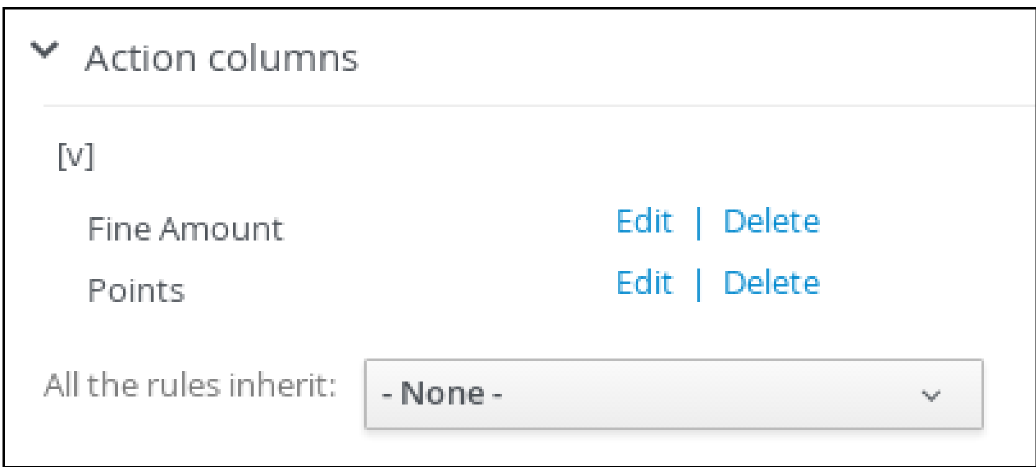
5. Click **Insert Column**, select **Set the value of a field→ Pattern**, and select **Violation[v]** from the **Pattern** menu.

6. Select **Field** and then **points** from the **Field** menu.

7. Select **Value options**, then select **Additional info** and enter **Points** in the **Header (description)** field.

8. Select the **Update engine with changes** option and click **Finish**.

Figure 4.11. Action columns



4.1.3. Inserting guided decision table rows

After you have created your columns in the guided decision table, you can add rows and define rules using the decision table designer.

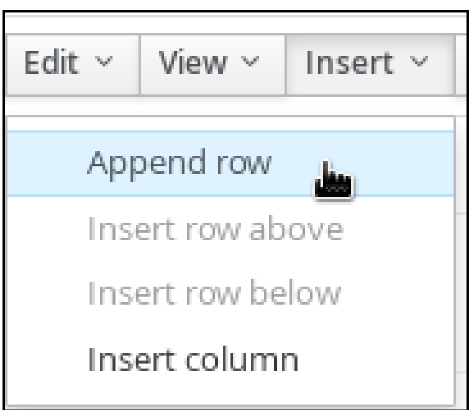
Prerequisites

You have created the **Violation Type**, **Fine Amount**, and **Points** columns in the traffic violation guided decision table.

Procedure

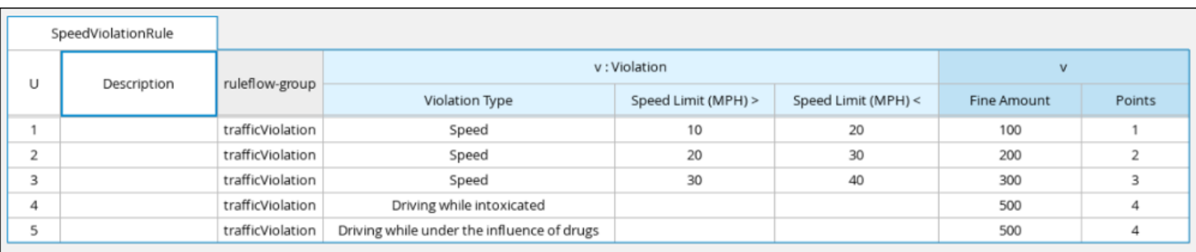
1. Click the **Model** tab to view the **SpeedViolationRule** table.
2. Click **Insert → Append row**. Repeat this step to add a total of five table rows.

Figure 4.12. Append row menu location



3. Fill out the table as shown in the following example:

Figure 4.13. Populated data fields



SpeedViolationRule

U	Description	ruleflow-group	v : Violation			v	
			Violation Type	Speed Limit (MPH) >	Speed Limit (MPH) <	Fine Amount	Points
1		trafficViolation	Speed	10	20	100	1
2		trafficViolation	Speed	20	30	200	2
3		trafficViolation	Speed	30	40	300	3
4		trafficViolation	Driving while intoxicated			500	4
5		trafficViolation	Driving while under the influence of drugs			500	4

4. Click **Save**, and then click **Save** to confirm your changes.

CHAPTER 5. TEST SCENARIOS

Test Scenarios in Red Hat Decision Manager enable you to validate the functionality of rules, models, and events before deploying them into production. A test scenario uses data for conditions that resemble an instance of your fact or project model. This data is matched against a specific set of rules and if the expected results match the actual results, the test is successful. If the expected results do not match the actual results, then the test fails.

After you run all test scenarios, the status of the scenarios is reported in a **Reporting** panel.

Test scenarios can be executed one at a time or as a group. The group execution contains all the scenarios from one package. Test scenarios are independent, so that one scenario cannot affect or modify the other.

5.1. TESTING THE SPEED LIMIT SCENARIO

Test the speed limit scenario using the data that you specified when you created the traffic violation guided decision table.

Prerequisites

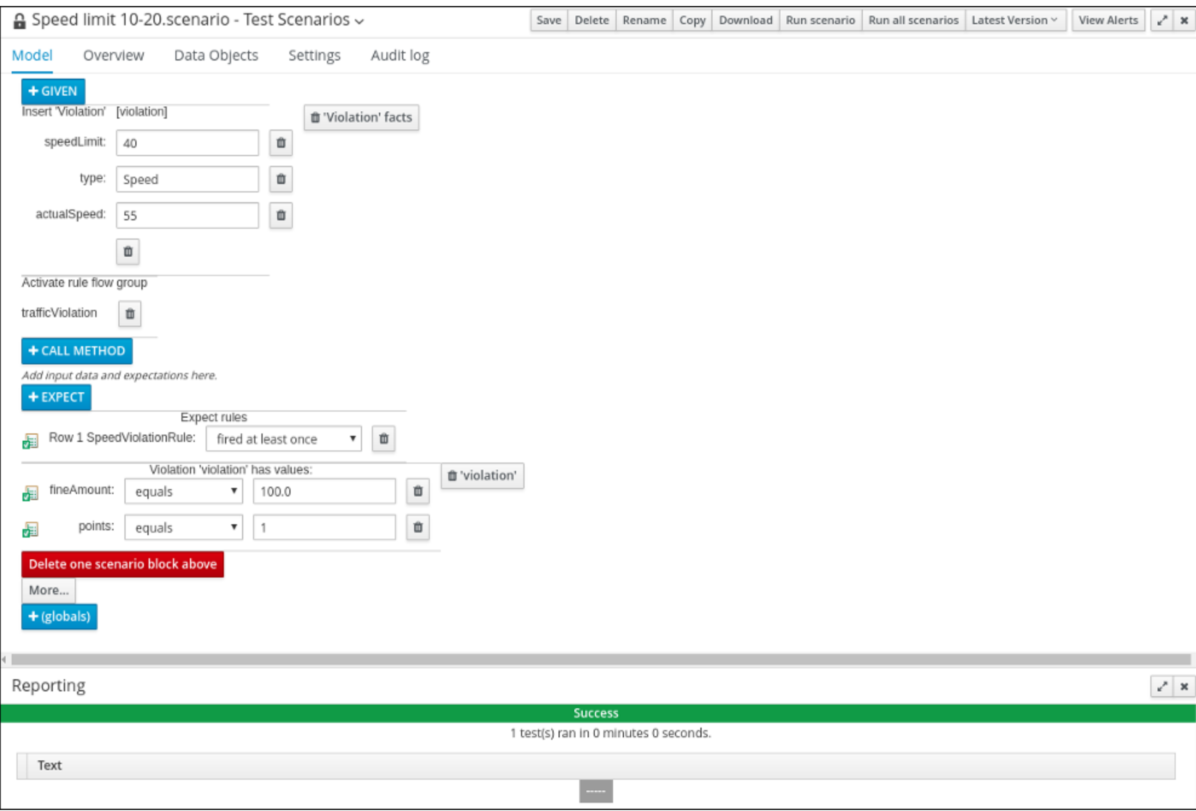
- You have created the **Driver_department_traffic_violations** project.
- You have created the **Violation** and **Driver** data objects.
- You have created the **SpeedViolationRule** guided decision table.

Procedure

1. Click **Menu** → **Design** → **Projects**, then **Driver_department_traffic_violations**.
2. Click **Add Asset** → **Test Scenario**.
3. In the **Create new Test Scenario window**, enter the following values:
    a. **Test Scenario**: **Speed limit 10-20**.
    b. **Package**: select **com.myspace.driver_department_traffic_violations**.
4. Click **Ok**.
5. Click **+GIVEN** to open the **New input** window.
6. Select **Violation** from the **Insert a new fact** menu.
7. Enter **violation** in the **Fact name** field and click **Add**.
8. Click **Add a field** located under **Insert 'Violation'[violation]** to open the **Choose a field to add** window.
9. Select **speedLimit** from the **Choose a field to add** menu and click **OK**.
10. Click ( ), and then click **Literal value**.
11. Enter **40** in the **speedLimit** field.

12. Click **Insert 'Violation'[violation]**.
13. Select **type** from the **Choose a field to add** menu in the **Choose a field to add** window, and click **OK**.
14. Click ( ), and then click **Literal value**.
15. Enter **Speed** in the **type** field.
16. Click **Insert 'Violation'[violation]**.
17. Select **actualSpeed** from the **Choose a field to add** menu, and click **OK**.
18. Click ( ), and then click **Literal value**.
19. Enter **55** in the **actualSpeed** field.
20. Click **+Expect** to open the **New expectation** window.
21. Expand the **Rule** menu, select **Row 1 SpeedViolationRule**, and click **OK**.
22. Click **+GIVEN** to open the **New input** window, enter **trafficViolation** in the **Activate rule flow group** field, and click **Add**.
23. Click **+Expect** to open the **New expectation** window and click **Add** next to **Fact value: violation**.
24. Click **Violation 'violation' has values:** to open the **Choose a field to add** window.
25. Select **fineAmount** from the **Choose a field to add** menu and click **OK**.
26. Enter **100.0** in the **fineAmount: equals** field.
27. Click **Violation 'violation' has values:** to open the **Choose a field to add** window.
28. Select **points** from the **Choose a field to add** menu and click **OK**.
29. Enter **1** in the **points: equals** field.
30. Click **Save**, and then click **Save** to confirm your changes.
31. Click **Run scenario**.

Figure 5.1. Speed test results screen



If the values and conditions set in the test scenario meet the requirements as specified in the speed violation guided decision table, the **Reporting** section at the bottom of the window displays a **Success** message.

5.2. TESTING THE DRIVER LICENSE SUSPENSION SCENARIO

Test the driver license suspension scenario using the data that you specified when you set the Driver License Suspension rules and actions.

Prerequisites

- You have created the **Driver_department_traffic_violations** project.
- You have created the **Violation** and **Driver** data objects.
- You have set the Driver License Suspension rules and actions.

Procedure

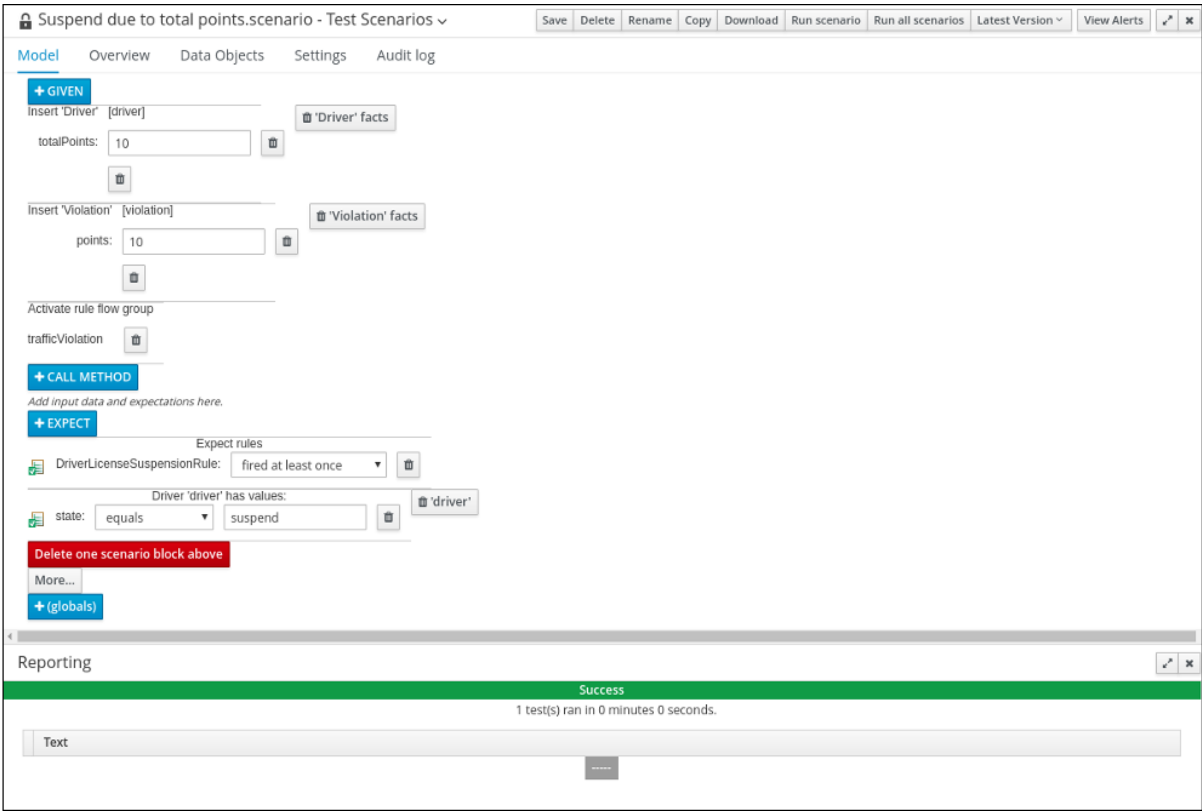
1. Click **Menu → Design → Projects**, then **Driver_department_traffic_violations**.
2. Click **Add Asset → Test Scenario**.
3. In the **Create new Test Scenario window**wizard, enter the following values:
    a. **Test Scenario**: **Suspend due to total points**.
    b. **Package**: select **com.myspace.driver_department_traffic_violations**.
4. Click **Ok**.

5. Click **+GIVEN** to open the **New input** window.
6. Select **Driver** from the **Insert a new fact** menu.
7. Enter **driver** in the **Fact name** field and click **Add**.
8. Click **Add a field** located under **'Driver'[driver]** to open the **Choose a field to add** window.
9. Select **totalPoints** from the **Choose a field to add** menu and click **OK**.
10. Click ( ) next to **totalPoints**, click **Literal value**, then enter **10** in the **totalPoints** field.
11. Click **+GIVEN** to open the **New input** window.
12. Select **Violation** from the **Insert a new fact** menu.
13. Enter **violation** in the **Fact name** field and click **Add**.
14. Click **Add a field** located under **Insert 'Violation'[violation]** to open the **Choose a field to add** window.
15. Select **points** from the **Choose a field to add** menu and click **OK**.
16. Click ( ), and then click **Literal value** next to **Literal value**.
17. Enter **10** in the **points** field.
18. Click **+Expect** to open the **New expectation** window.
19. Expand the **Rule** menu, select **DriverLicenseSuspensionRule**, and click **OK**.
20. Click **+GIVEN** to open the **New input** window, enter **trafficViolation** in the **Activate rule flow group** field, and click **Add**.
21. Click **+Expect** to open the **New expectation** window and click **Add** next to **Fact value: driver**.
22. Click **Driver 'driver' has values:** to open the **Choose a field to add** window.
23. Select **state** from the **Choose a field to add** menu and click **OK**.
24. Enter **suspend** in the **state: equals** field.
25. Click **Save**, and then click **Save** to confirm your changes.
26. Click **Run scenario**.

Result

The rule is fired and the driver’s license is suspended because the total number of points is >= 20.

Figure 5.2. Suspension test results screen



If the values and conditions set in the test scenario meet the requirements that you specified when you set the Driver License Suspension rules and actions, the **Reporting** section at the bottom of the window displays a **Success** message.

5.3. TESTING THE MULTIPLE VIOLATIONS SCENARIO

Copy the **Suspend due to total points**asset and modify it to create the driver license suspension scenario for drivers with multiple violations using the data that you specified when you set the Driver License Suspension rules and actions.

Prerequisites

- You have created the **Driver_department_traffic_violations** project.
- You have created the **Violation** and **Driver** data objects.
- You have set the driver license suspension rules and actions.

Procedure

1. Click **Menu** → **Design** → **Projects**, then **Driver_department_traffic_violations**.
2. Click **Suspend due to total points**→ **Copy**, enter **Suspend due to multiple violations** in the **New Asset Name** field, and click **Make a Copy**.
3. Click **Space** → **MySpace** → **Driver_department_traffic_violations**, and then select the **Suspend due to multiple violations** Asset.
4. Click **+GIVEN** to open the **New input** window.

5. Select **Violation** from the **Insert a new fact** menu.

6. Enter **violation2** in the **Fact name** field and click **Add**.

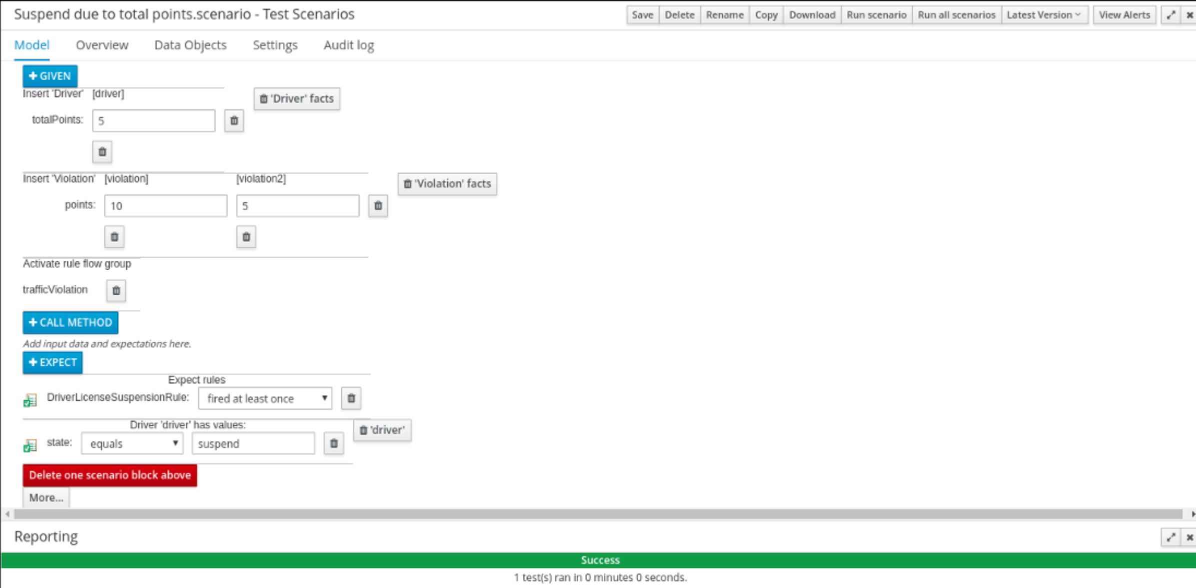
7. Click ( ) next to **points**, click **Literal value**, then enter **5** in the **points → violation2** field.

8. In the **totalPoints** field, change the value from **10** to **5**.

9. Click **Save**, and then click **Save** to confirm your changes.

10. Click **Run scenario**.

Figure 5.3. Suspension test results screen



If the values and conditions set in the test scenario meet the requirements that you specified when you set the Driver License Suspension rules and actions, the **Reporting** section at the bottom of the window displays a **Success** message.

CHAPTER 6. EXAMPLE PROJECTS AND BUSINESS ASSETS IN DECISION CENTRAL

Decision Central contains example projects with example business assets that you can use as a reference for the rules or other assets that you create in your own Red Hat Decision Manager projects. Each sample project is designed differently to demonstrate decision management or business optimization assets and logic in Red Hat Decision Manager.

The following example projects are available in Decision Central:

- **Mortgages**: (Decision management) Example loan approval process using decision assets. Determines loan eligibility based on applicant data and qualifications.
- **Employee_Rostering**: (Business optimization) Example employee rostering optimization using decision and solver assets. Assigns employees to shifts based on skills.
- **OptaCloud**: (Business optimization) Example resource allocation optimization using decision and solver assets. Assigns processes to computers with limited resources.

6.1. ACCESSING EXAMPLE PROJECTS AND BUSINESS ASSETS IN DECISION CENTRAL

You can use the example projects in Decision Central to explore example business assets as a reference for the rules or other assets that you create in your own Red Hat Decision Manager projects.

Prerequisites

- Decision Central is installed and running. For installation options, see *Planning a Red Hat Decision Manager installation*.

Procedure

1. In Decision Central, go to **Menu** → **Design** → **Projects** and click **Try Samples**.
   If a project already exists, click the three vertical dots in the upper-right corner of the **Projects** page and click **Try Samples**.
2. Review the descriptions for each sample project to determine which project you want to explore. Each sample project is designed differently to demonstrate decision management or business optimization assets and logic in Red Hat Decision Manager.
3. Select one or more sample projects and click **Ok** to add the projects to your space.
4. In the **Projects** page of your space, select one of the new example projects to view the example assets for that project.
5. Select each example asset to explore how the project is designed to achieve the specified goal or workflow.
6. In the upper-right corner of the project **Assets** page, click **Build** to build the sample project or **Deploy** to build the project and then deploy it to Decision Server.
   To review project deployment details (if applicable), go to **Menu** → **Deploy** → **Execution Servers**.

6.2. EXECUTING RULES

After you identify example rules or create your own rules in Decision Central, you can build and deploy the associated project and execute rules locally or on Decision Server to test the rules.

Prerequisites

- Decision Central and Decision Server are installed and running. For installation options, see *Planning a Red Hat Decision Manager installation* .

Procedure

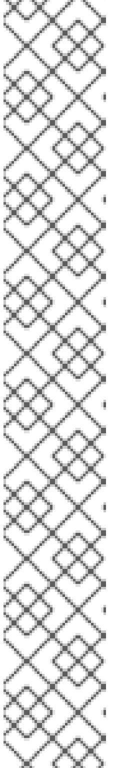
1. In Decision Central, go to **Menu** → **Design** → **Projects** and click the project name.
2. In the upper-right corner of the project **Assets** page, click **Deploy** to build the project and deploy it to Decision Server. If the build fails, address any problems described in the **Alerts** panel at the bottom of the screen.
   For more information about deploying projects, see *Packaging and deploying a Red Hat Decision Manager project*.
3. Create a Maven or Java project outside of Decision Central, if not created already, that you can use for executing rules locally or that you can use as a client application for executing rules on Decision Server. The project must contain a **pom.xml** file and any other required components for executing the project resources.
   For example test projects, see "Other methods for creating and executing DRL rules".
4. Open the **pom.xml** file of your test project or client application and add the following dependencies, if not added already:
   - **kie-ci**: Enables your client application to load Decision Central project data locally using **ReleaseId**
   - **kie-server-client**: Enables your client application to interact remotely with assets on Decision Server
   - **slf4j**: (Optional) Enables your client application to use Simple Logging Facade for Java (SLF4J) to return debug logging information after you interact with Decision Server

   Example dependencies for Red Hat Decision Manager 7.2 in a client application **pom.xml** file:

```
<!-- For local execution -->
<dependency>
  <groupId>org.kie</groupId>
  <artifactId>kie-ci</artifactId>
  <version>7.14.0.Final-redhat-00002</version>
</dependency>

<!-- For remote execution on Decision Server -->
<dependency>
  <groupId>org.kie.server</groupId>
  <artifactId>kie-server-client</artifactId>
  <version>7.14.0.Final-redhat-00002</version>
</dependency>

<!-- For debug logging (optional) -->
<dependency>
  <groupId>org.slf4j</groupId>
```

```
  <artifactId>slf4j-simple</artifactId>
  <version>1.7.25</version>
</dependency>
```

For available versions of these artifacts, search the group ID and artifact ID in the Nexus Repository Manager online.



NOTE

Instead of specifying a Red Hat Decision Manager **<version>** for individual dependencies, consider adding the Red Hat Business Automation bill of materials (BOM) dependency to your project **pom.xml** file. The Red Hat Business Automation BOM applies to both Red Hat Decision Manager and Red Hat Process Automation Manager. When you add the BOM files, the correct versions of transitive dependencies from the provided Maven repositories are included in the project.

Example BOM dependency:

```
<dependency>
  <groupId>com.redhat.ba</groupId>
  <artifactId>ba-platform-bom</artifactId>
  <version>7.2.0.GA-redhat-00002</version>
  <scope>import</scope>
  <type>pom</type>
</dependency>
```

For more information about the Red Hat Business Automation BOM, see What is the mapping between Red Hat Decision Manager and the Maven library version?.

5. Ensure that the dependencies for artifacts containing model classes are defined in the client application **pom.xml** file exactly as they appear in the **pom.xml** file of the deployed project. If dependencies for model classes differ between the client application and your projects, execution errors can occur.
   To access the project **pom.xml** file in Decision Central, select any existing asset in the project and then in the **Project Explorer** menu on the left side of the screen, click the **Customize View** gear icon and select **Repository View → pom.xml**.

   For example, the following **Person** class dependency appears in both the client and deployed project **pom.xml** files:

```
<dependency>
  <groupId>com.sample</groupId>
  <artifactId>Person</artifactId>
  <version>1.0.0</version>
</dependency>
```

6. If you added the **slf4j** dependency to the client application **pom.xml** file for debug logging, create a **simplelogger.properties** file on the relevant classpath (for example, in **src/main/resources/META-INF** in Maven) with the following content:

```
org.slf4j.simpleLogger.defaultLogLevel=debug
```

7. In your client application, create a **.java** main class containing the necessary imports and a **main()** method to load the KIE base, insert facts, and execute the rules.
   For example, a **Person** object in a project contains getter and setter methods to set and retrieve the first name, last name, hourly rate, and the wage of a person. The following **Wage** rule in a project calculates the wage and hourly rate values and displays a message based on the result:

```
package com.sample;

import com.sample.Person;

dialect "java"

rule "Wage"
  when
    Person(hourlyRate * wage > 100)
    Person(name : firstName, surname : lastName)
  then
    System.out.println("Hello" + " " + name + " " + surname + "!");
    System.out.println("You are rich!");
end
```

To test this rule locally outside of Decision Server (if desired), configure the **.java** class to import KIE services, a KIE container, and a KIE session, and then use the **main()** method to fire all rules against a defined fact model:

Executing rules locally

```
import org.kie.api.KieServices;
import org.kie.api.runtime.KieContainer;
import org.kie.api.runtime.KieSession;

public class RulesTest {

  public static final void main(String[] args) {
    try {
      // Identify the project in the local repository:
      ReleaseId rid = new ReleaseId();
      rid.setGroupId("com.myspace");
      rid.setArtifactId("MyProject");
      rid.setVersion("1.0.0");

      // Load the KIE base:
      KieServices ks = KieServices.Factory.get();
      KieContainer kContainer = ks.newKieContainer(rid);
      KieSession kSession = kContainer.newKieSession();

      // Set up the fact model:
      Person p = new Person();
      p.setWage(12);
      p.setFirstName("Tom");
      p.setLastName("Summers");
      p.setHourlyRate(10);

      // Insert the person into the session:
      kSession.insert(p);
```

```
            // Fire all rules:
            kSession.fireAllRules();
            kSession.dispose();
        }

        catch (Throwable t) {
            t.printStackTrace();
        }
    }
}
```

To test this rule on Decision Server, configure the **.java** class with the imports and rule execution information similarly to the local example, and additionally specify KIE services configuration and KIE services client details:

Executing rules on Decision Server

```
package com.sample;

import java.util.ArrayList;
import java.util.HashSet;
import java.util.List;
import java.util.Set;

import org.kie.api.command.BatchExecutionCommand;
import org.kie.api.command.Command;
import org.kie.api.KieServices;
import org.kie.api.runtime.ExecutionResults;
import org.kie.api.runtime.KieContainer;
import org.kie.api.runtime.KieSession;
import org.kie.server.api.marshalling.MarshallingFormat;
import org.kie.server.api.model.ServiceResponse;
import org.kie.server.client.KieServicesClient;
import org.kie.server.client.KieServicesConfiguration;
import org.kie.server.client.KieServicesFactory;
import org.kie.server.client.RuleServicesClient;

import com.sample.Person;

public class RulesTest {

  private static final String containerName = "testProject";
  private static final String sessionName = "myStatelessSession";

  public static final void main(String[] args) {
    try {
      // Define KIE services configuration and client:
      Set<Class<?>> allClasses = new HashSet<Class<?>>();
      allClasses.add(Person.class);
      String serverUrl = "http://$HOST:$PORT/kie-server/services/rest/server";
      String username = "$USERNAME";
      String password = "$PASSWORD";
      KieServicesConfiguration config =
        KieServicesFactory.newRestConfiguration(serverUrl,
                                                username,
```

```
                                    password);
      config.setMarshallingFormat(MarshallingFormat.JAXB);
      config.addExtraClasses(allClasses);
      KieServicesClient kieServicesClient =
        KieServicesFactory.newKieServicesClient(config);

      // Set up the fact model:
      Person p = new Person();
      p.setWage(12);
      p.setFirstName("Tom");
      p.setLastName("Summers");
      p.setHourlyRate(10);

      // Insert Person into the session:
      KieCommands kieCommands = KieServices.Factory.get().getCommands();
      List<Command> commandList = new ArrayList<Command>();
      commandList.add(kieCommands.newInsert(p, "personReturnId"));

      // Fire all rules:
      commandList.add(kieCommands.newFireAllRules("numberOfFiredRules"));
      BatchExecutionCommand batch = kieCommands.newBatchExecution(commandList,
sessionName);

      // Use rule services client to send request:
      RuleServicesClient ruleClient =
kieServicesClient.getServicesClient(RuleServicesClient.class);
      ServiceResponse<ExecutionResults> executeResponse =
ruleClient.executeCommandsWithResults(containerName, batch);
      System.out.println("number of fired rules:" +
executeResponse.getResult().getValue("numberOfFiredRules"));
    }

    catch (Throwable t) {
      t.printStackTrace();
    }
  }
}
```

8. Run the configured **.java** class from your project directory. You can run the file in your development platform (such as Red Hat JBoss Developer Studio) or in the command line. Example Maven execution (within project directory):

   ```
   mvn clean install exec:java -Dexec.mainClass="com.sample.app.RulesTest"
   ```

   Example Java execution (within project directory)

   ```
   javac -classpath "./$DEPENDENCIES/*:." RulesTest.java
   java -classpath "./$DEPENDENCIES/*:." RulesTest
   ```

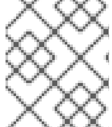
9. Review the rule execution status in the command line and in the server log. If any rules do not execute as expected, review the configured rules in the project and the main class configuration to validate the data provided.

CHAPTER 7. EXAMPLE DECISIONS IN RED HAT DECISION MANAGER FOR AN IDE

Red Hat Decision Manager provides example decisions distributed as Java classes that you can import into your integrated development environment (IDE). You can use these examples to better understand Red Hat Decision Manager decision engine capabilities or use them as a reference for the decisions that you define in your own Red Hat Decision Manager projects.

The following example decision sets are some of the examples available in Red Hat Decision Manager:

- **Hello World example**: Demonstrates basic rule execution and use of debug output
- **State example**: Demonstrates forward chaining and conflict resolution through rule salience and agenda groups
- **Fibonacci example**: Demonstrates recursion and conflict resolution through rule salience
- **Banking example**: Demonstrates pattern matching, basic sorting, and calculation
- **Pet Store example**: Demonstrates rule agenda groups, global variables, callbacks, and GUI integration
- **Sudoku example**: Demonstrates complex pattern matching, problem solving, callbacks, and GUI integration
- **House of Doom example**: Demonstrates backward chaining and recursion



NOTE

For optimization examples provided with Red Hat Business Optimizer, see *Getting started with Red Hat Business Optimizer*.

7.1. IMPORTING AND EXECUTING RED HAT DECISION MANAGER EXAMPLE DECISIONS IN AN IDE

You can import Red Hat Decision Manager example decisions into your integrated development environment (IDE) and execute them to explore how the rules and code function. You can use these examples to better understand Red Hat Decision Manager decision engine capabilities or use them as a reference for the decisions that you define in your own Red Hat Decision Manager projects.

Prerequisites

- Java 8 or later is installed.
- Maven 3.5.x or later is installed.
- An IDE is installed, such as Red Hat JBoss Developer Studio.

Procedure

1. Download and unzip the **Red Hat Decision Manager 7.2.0 Source Distribution**from the Red Hat Customer Portal to a temporary directory, such as **/rhdm-7.2.0-sources**.

2. Open your IDE and select **File → Import → Maven → Existing Maven Projects**, or the equivalent option for importing a Maven project.

3. Click **Browse**, navigate to **~/rhdm-7.2.0-sources/src/drools-$VERSION/drools-examples** (or, for the Conway's Game of Life example, **~/rhdm-7.2.0-sources/src/droolsjbpm-integration-$VERSION/droolsjbpm-integration-examples**), and import the project.

4. Navigate to the example package that you want to run and find the Java class with the **main** method.

5. Right-click the Java class and select **Run As → Java Application** to run the example.
   To run all examples through a basic user interface, run the **DroolsExamplesApp.java** class (or, for Conway's Game of Life, the **DroolsJbpmIntegrationExamplesApp.java** class) in the **org.drools.examples** main class.

Figure 7.1. Interface for all examples in drools-examples (DroolsExamplesApp.java)

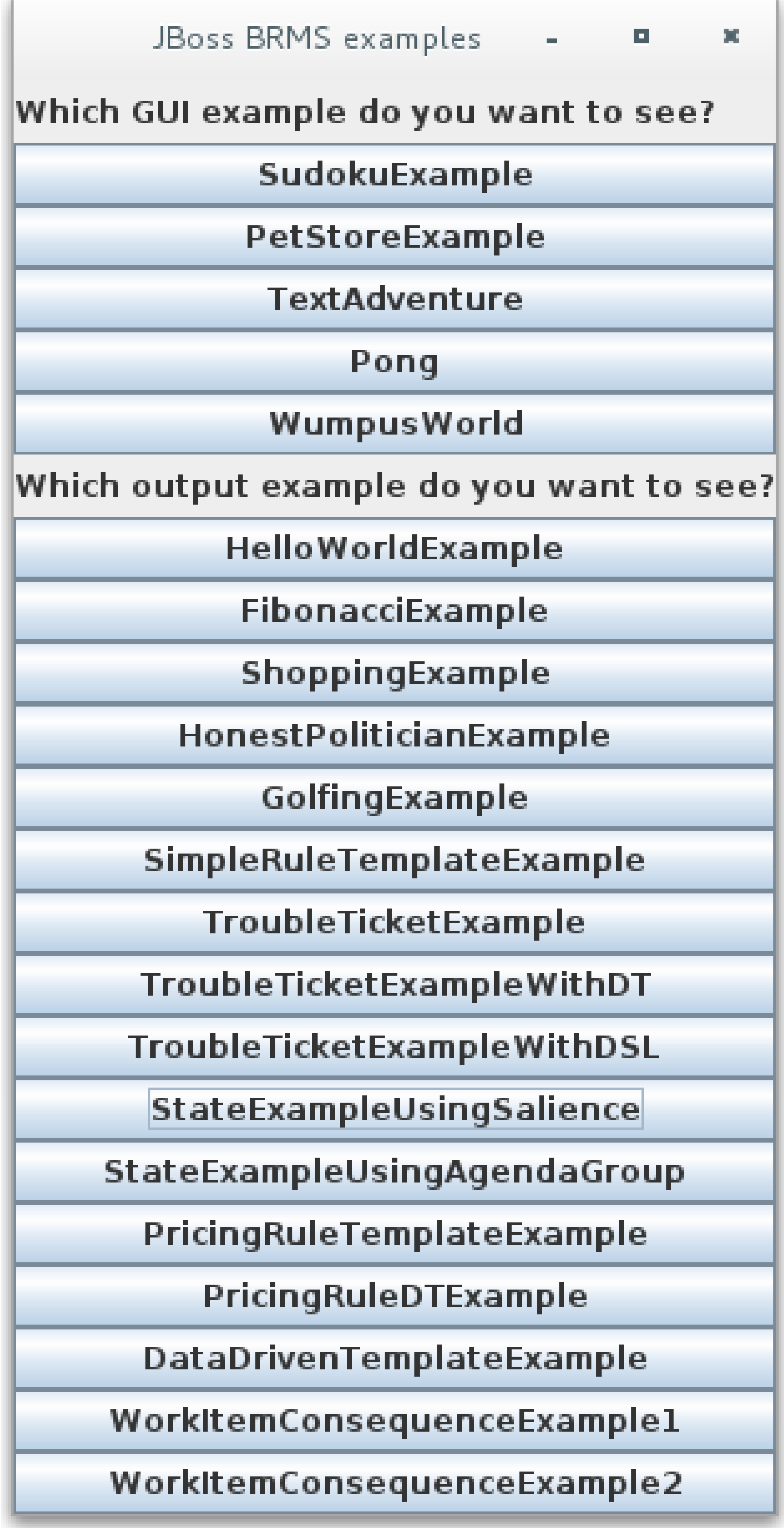
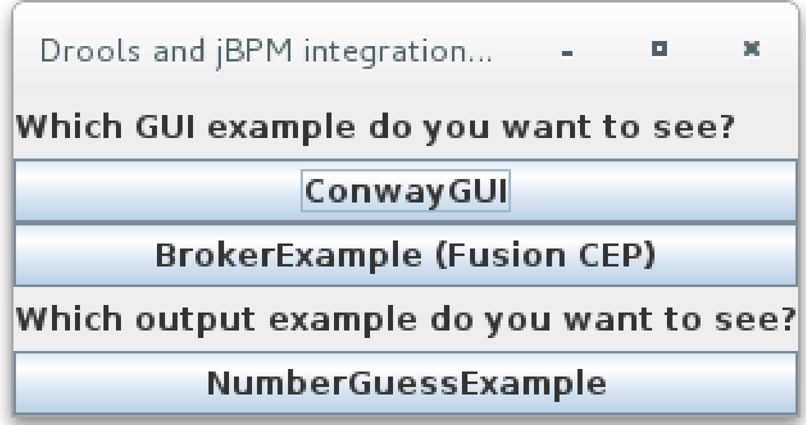


Figure 7.2. Interface for all examples in droolsjbpm-integration-examples (DroolsJbpmIntegrationExamplesApp.java)



7.2. HELLO WORLD EXAMPLE DECISIONS (BASIC RULES AND DEBUGGING)

The Hello World example decision set demonstrates how to insert objects into the Red Hat Decision Manager decision engine working memory, how to match the objects using rules, and how to configure logging to trace the internal activity of the engine.

The following is an overview of the Hello World example:

- Name: **helloworld**
- Main class: **org.drools.examples.helloworld.HelloWorldExample** (in **src/main/java**)
- Module: **drools-examples**
- **Type**: Java application
- Rule file: **org.drools.examples.helloworld.HelloWorld.drl** (in **src/main/resources**)
- **Objective**: Demonstrates basic rule execution and use of debug output

In the Hello World example, a KIE session is generated to enable rule execution. All rules require a KIE session for execution.

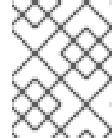
KIE session for rule execution

```
KieServices ks = KieServices.Factory.get(); 1
KieContainer kc = ks.getKieClasspathContainer(); 2
KieSession ksession = kc.newKieSession("HelloWorldKS"); 3
```

1. Obtains the **KieServices** factory. This is the main interface that applications use to interact with the engine.

2. Creates a **KieContainer** from the project class path. This detects a **/META-INF/kmodule.xml** file from which it configures and instantiates a **KieContainer** with a **KieModule**.

3. Creates a **KieSession** based on the **"HelloWorldKS"** KIE session configuration defined in the **/META-INF/kmodule.xml** file.



NOTE

For more information about Red Hat Decision Manager project packaging, see *Packaging and deploying a Red Hat Decision Manager project*.

Red Hat Decision Manager has an event model that exposes internal engine activity. Two default debug listeners, **DebugAgendaEventListener** and **DebugWorkingMemoryEventListener**, print debug event information to the **System.err** output. The **KieRuntimeLogger** provides execution auditing, the result of which you can view in a graphical viewer.

Debug listeners and audit loggers

```
// Set up listeners.
ksession.addEventListener( new DebugAgendaEventListener() );
ksession.addEventListener( new DebugRuleRuntimeEventListener() );

// Set up a file-based audit logger.
KieRuntimeLogger logger = KieServices.get().getLoggers().newFileLogger( ksession,
"./target/helloworld" );

// Set up a ThreadedFileLogger so that the audit view reflects events while debugging.
KieRuntimeLogger logger = ks.getLoggers().newThreadedFileLogger( ksession, "./target/helloworld",
1000 );
```

The logger is a specialized implementation built on the **Agenda** and **RuleRuntime** listeners. When the engine has finished executing, **logger.close()** is called.

The example creates a single **Message** object with the message **"Hello World"**, inserts the status **HELLO** into the **KieSession**, executes rules with **fireAllRules()**.

Data insertion and execution

```
// Insert facts into the KIE session.
final Message message = new Message();
message.setMessage( "Hello World" );
message.setStatus( Message.HELLO );
ksession.insert( message );

// Fire the rules.
ksession.fireAllRules();
```

Rule execution uses a data model to pass data as inputs and outputs to the **KieSession**. The data model in this example has two fields: the **message**, which is a **String**, and the **status**, which can be **HELLO** or **GOODBYE**.

Data model class

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

    private String         message;
```

```
    private int          status;
    ...
}
```

The two rules are located in the file **src/main/resources/org/drools/examples/helloworld/HelloWorld.drl**.

The **when** condition of the **"Hello World"** rule states that the rule is activated for each **Message** object inserted into the KIE session that has the status **Message.HELLO**. Additionally, 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 **then** action of the rule is written using the MVEL expression language, as declared by the rule **dialect** attribute. 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**. The rule uses the MVEL **modify** statement to apply a block of assignments in one statement and to notify the engine of the changes at the end of the block.

"Hello World" rule

```
rule "Hello World"
    dialect "mvel"
  when
    m : Message( status == Message.HELLO, message : message )
  then
    System.out.println( message );
    modify ( m ) { message = "Goodbye cruel world",
                   status = Message.GOODBYE };
end
```

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

"Good Bye" rule

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

To execute the example, run the **org.drools.examples.helloworld.HelloWorldExample** class as a Java application in your IDE. The rule writes to **System.out**, the debug listener writes to **System.err**, and the audit logger creates a log file in **target/helloworld.log**.

System.out output in the IDE console

```
Hello World
Goodbye cruel world
```

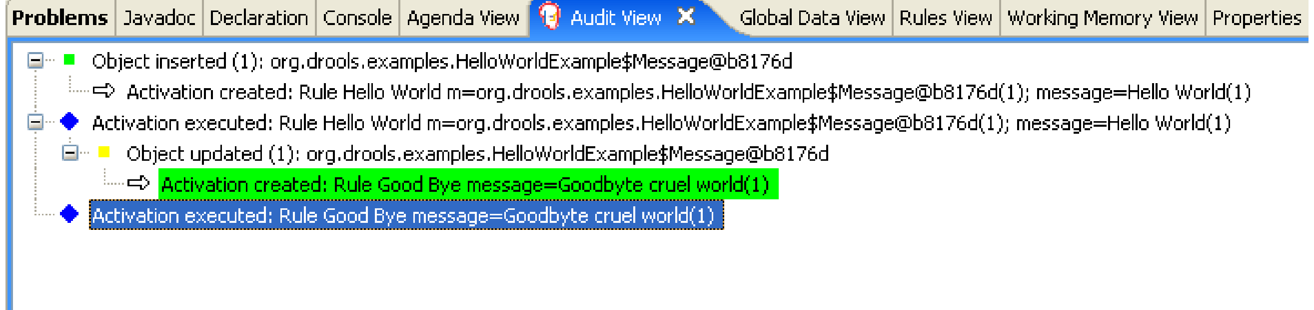
System.err output in the IDE console

```
==>[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]
```

To better understand the execution flow of this example, you can load the audit log file from **target/helloworld.log** into your IDE debug view or **Audit View**, if available (for example, in **Window → Show View** in some IDEs).

In this example, the **Audit view** shows that the object is inserted, which creates an activation for the **"Hello World"** rule. The activation is then executed, which updates the **Message** object and causes the **"Good Bye"** rule to activate. Finally, the **"Good Bye"** rule is executed. When you select an event in the **Audit View**, the origin event, which is the **"Activation created"** event in this example, is highlighted in green.

Figure 7.3. Hello World example Audit View



7.3. STATE EXAMPLE DECISIONS (FORWARD CHAINING AND CONFLICT RESOLUTION)

The State example decision set demonstrates how the decision engine uses forward chaining and any changes to facts in the working memory to resolve execution conflicts for rules in a sequence. The example focuses on resolving conflicts through salience values or through agenda groups that you can define in rules.

The following is an overview of the State example:

- Name: **state**
- Main classes: **org.drools.examples.state.StateExampleUsingSalience**, **org.drools.examples.state.StateExampleUsingAgendaGroup** (in **src/main/java**)

- Module: **drools-examples**
- **Type**: Java application
- Rule files: **org.drools.examples.state.*.drl** (in **src/main/resources**)
- **Objective**: Demonstrates forward chaining and conflict resolution through rule salience and agenda groups

A forward-chaining rule system is a data-driven system that starts with a fact in the working memory of the decision engine and reacts to changes to that fact. When objects are inserted into working memory, any rule conditions that become true as a result of the change are scheduled for execution by the agenda.

In contrast, a backward-chaining rule system is a goal-driven system that starts with a conclusion that the decision engine attempts to satisfy, often using recursion. If the system cannot reach the conclusion or goal, it searches for subgoals, which are conclusions that complete part of the current goal. The system continues this process until either the initial conclusion is satisfied or all subgoals are satisfied.

The decision engine in Red Hat Decision Manager uses both forward and backward chaining to evaluate rules.

The following diagram illustrates how the decision engine evaluates rules using forward chaining overall with a backward-chaining segment in the logic flow:

Figure 7.4. Rule evaluation logic using forward and backward chaining

In the State example, each **State** class has fields for its name and its current state (see the class **org.drools.examples.state.State**). The following states are the two possible states for each object:

- **NOTRUN**
- **FINISHED**

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...
}
```

The State example contains two versions of the same example to resolve rule execution conflicts:

- A **StateExampleUsingSalience** version that resolves conflicts by using rule salience
- A **StateExampleUsingAgendaGroups** version that resolves conflicts by using rule agenda groups

Both versions of the state example involve four **State** objects: **A**, **B**, **C**, and **D**. Initially, their states are set to **NOTRUN**, which is the default value for the constructor that the example uses.

State example using salience

The **StateExampleUsingSalience** version of the State example uses salience values in rules to resolve rule execution conflicts. Rules with a higher salience value are given higher priority when ordered in the activation queue.

The example inserts each **State** instance into the KIE session and then calls **fireAllRules()**.

Salience State example execution

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

ksession.insert( a );
ksession.insert( b );
ksession.insert( c );
ksession.insert( d );

ksession.fireAllRules();

// Dispose KIE session if stateful (not required if stateless).
ksession.dispose();
```

To execute the example, run the **org.drools.examples.state.StateExampleUsingSalience** class as a Java application in your IDE.

After the execution, the following output appears in the IDE console window:

Salience State example output in the IDE console

```
A finished
B finished
```

```
C finished
D finished
```

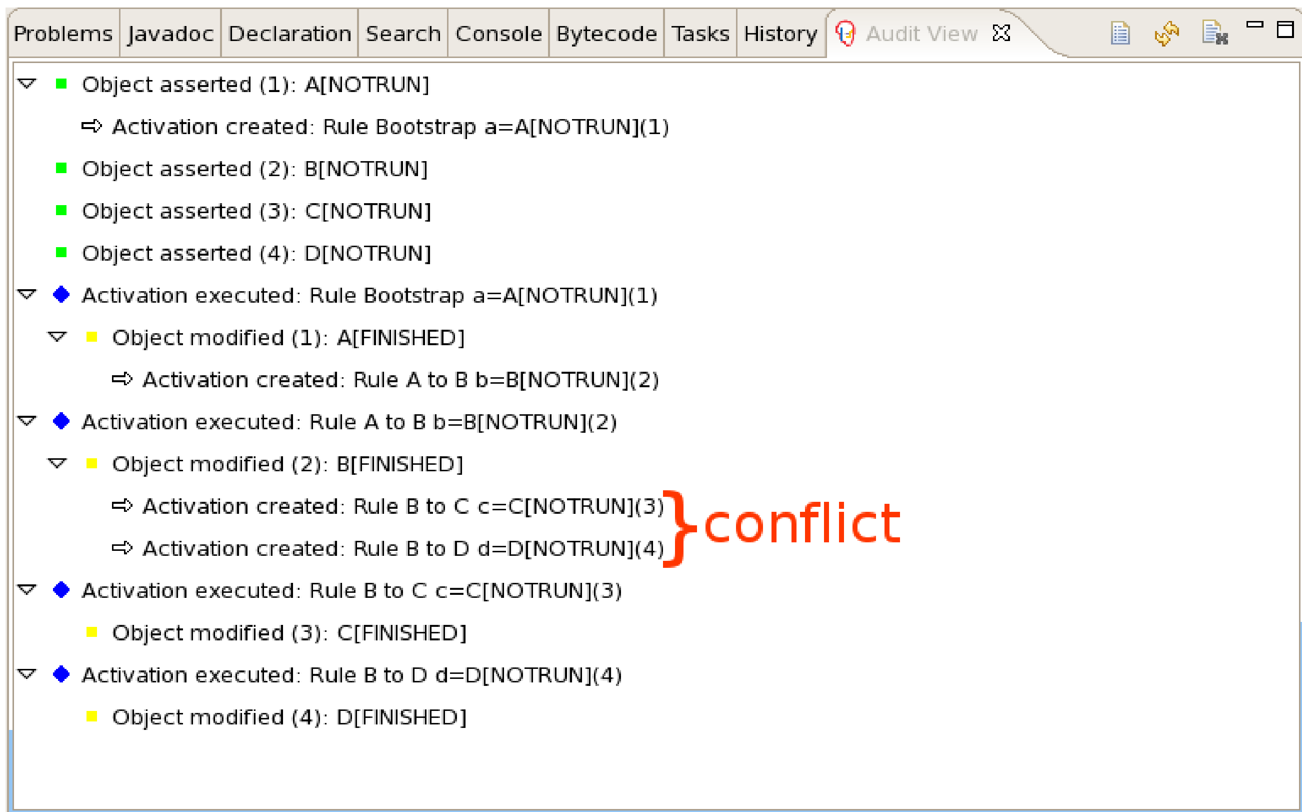
Four rules are present.

First, the **"Bootstrap"** rule fires, setting **A** to state **FINISHED**, which then causes **B** to change its state to **FINISHED**. Objects **C** and **D** are both dependent on **B**, causing a conflict that is resolved by the salience values.

To better understand the execution flow of this example, you can load the audit log file from **target/state.log** into your IDE debug view or **Audit View**, if available (for example, in **Window → Show View** in some IDEs).

In this example, the **Audit View** shows that the assertion of the object **A** in the state **NOTRUN** activates the **"Bootstrap"** rule, while the assertions of the other objects have no immediate effect.

Figure 7.5. Salience State example Audit View



Rule "Bootstrap" in salience State example

```
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 the **"Bootstrap"** rule changes the state of **A** to **FINISHED**, which activates rule **"A to B"**.

Rule "A to B" in salience State example

```
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 engine agenda.

Rules "B to C" and "B to D" in salience State example

```
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"
  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
```

From this point on, both rules may fire and, therefore, the rules are in conflict. The conflict resolution strategy enables the engine agenda to decide which rule to fire. Rule **"B to C"** has the higher salience value (**10** versus the default salience value of **0**), so it fires first, modifying object **C** to state **FINISHED**.

The **Audit View** in your IDE shows the modification of the **State** object in the rule **"A to B"**, which results in two activations being in conflict.

You can also use the **Agenda View** in your IDE to investigate the state of the engine agenda. In this example, the **Agenda View** shows the breakpoint in the rule **"A to B"** and the state of the agenda with the two conflicting rules. Rule **"B to D"** fires last, modifying object **D** to state **FINISHED**.

Figure 7.6. Salience State example Agenda View

StateExampleUsingSalience.java | StateExampleUsingSalience.drl

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

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

Text Editor | Rete Tree

Console | Tasks | Agenda View | Audit View | Global Data View | Rules View | Working Memory View

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

State example using agenda groups

The **StateExampleUsingAgendaGroups** version of the State example uses agenda groups in rules to resolve rule execution conflicts. Agenda groups enable you to partition the engine agenda to provide more execution control over groups of rules. By default, all rules are in the agenda group **MAIN**. You can use the **agenda-group** attribute to specify a different agenda group for the rule.

Initially, a working memory has its focus on the agenda group **MAIN**. Rules in an agenda group only fire when the group receives the focus. You can set the focus either by using the method **setFocus()** or the rule attribute **auto-focus**. The **auto-focus** attribute enables the rule to be given a focus automatically for its agenda group when the rule is matched and activated.

In this example, the **auto-focus** attribute enables rule **"B to C"** to fire before **"B to D"**.

Rule "B to C" in agenda group State example

```
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"**, enabling its active rules to fire, which then enables the rule **"B to D"** to fire.

Rule "B to D" in agenda group State example

```
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
```

To execute the example, run the **org.drools.examples.state.StateExampleUsingAgendaGroups** class as a Java application in your IDE.

After the execution, the following output appears in the IDE console window (same as the salience version of the State example):

Agenda group State example output in the IDE console

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

Dynamic facts in the State example

Another notable concept in this State example is the use of *dynamic facts*, based on objects that implement a **PropertyChangeListener** object. In order for the engine to see and react to changes of fact properties, the application must notify the engine that changes occurred. You can configure this

communication explicitly in the rules by using the **modify** statement, or implicitly by specifying that the facts implement the **PropertyChangeSupport** interface as defined by the JavaBeans specification.

This example demonstrates how to use the **PropertyChangeSupport** interface to avoid the need for explicit **modify** statements in the rules. To make use of this interface, ensure that your facts implement **PropertyChangeSupport** in the same way that the class **org.drools.example.State** implements it, and then use the following code in the DRL rule file to configure the engine to listen for property changes on those facts:

Declaring a dynamic fact

```
declare type State
  @propertyChangeSupport
end
```

When you use **PropertyChangeListener** objects, each setter must implement additional code for the notification. For example, the following setter for **state** is in the class **org.drools.examples**:

Setter example with PropertyChangeSupport

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

7.4. FIBONACCI EXAMPLE DECISIONS (RECURSION AND CONFLICT RESOLUTION)

The Fibonacci example decision set demonstrates how the decision engine uses recursion to resolve execution conflicts for rules in a sequence. The example focuses on resolving conflicts through salience values that you can define in rules.

The following is an overview of the Fibonacci example:

- Name: **fibonacci**
- Main class: **org.drools.examples.fibonacci.FibonacciExample** (in **src/main/java**)
- Module: **drools-examples**
- **Type**: Java application
- Rule file: **org.drools.examples.fibonacci.Fibonacci.drl** (in **src/main/resources**)
- **Objective**: Demonstrates recursion and conflict resolution through rule salience

The Fibonacci Numbers form a sequence starting with 0 and 1. The next Fibonacci number is obtained by adding the two preceding Fibonacci numbers: 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987, 1597, 2584, 4181, 6765, 10946, and so on.

The Fibonacci example uses the single fact class **Fibonacci** with the following two fields:

- **sequence**
- **value**

The **sequence** field indicates the position of the object in the Fibonacci number sequence. The **value** field shows the value of that Fibonacci object for that sequence position, where **-1** indicates a value that still needs to be computed.

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...
}
```

To execute the example, run the **org.drools.examples.fibonacci.FibonacciExample** class as a Java application in your IDE.

After the execution, the following output appears in the IDE console window:

Fibonacci example output in the IDE console

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

To achieve this behavior in Java, the example inserts a single **Fibonacci** object with a sequence field of **50**. The example then uses a recursive rule to insert the other 49 **Fibonacci** objects.

Instead of implementing the **PropertyChangeSupport** interface to use dynamic facts, this example uses the MVEL dialect **modify** keyword to enable a block setter action and notify the engine of changes.

Fibonacci example execution

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

This example uses the following three rules:

- **"Recurse"**
- **"Bootstrap"**
- **"Calculate"**

The rule **"Recurse"** 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 matching once you have all 50 Fibonacci objects in memory. The rule also has a **salience** value because you need to have all 50 **Fibonacci** objects asserted before you execute the **"Bootstrap"** rule.

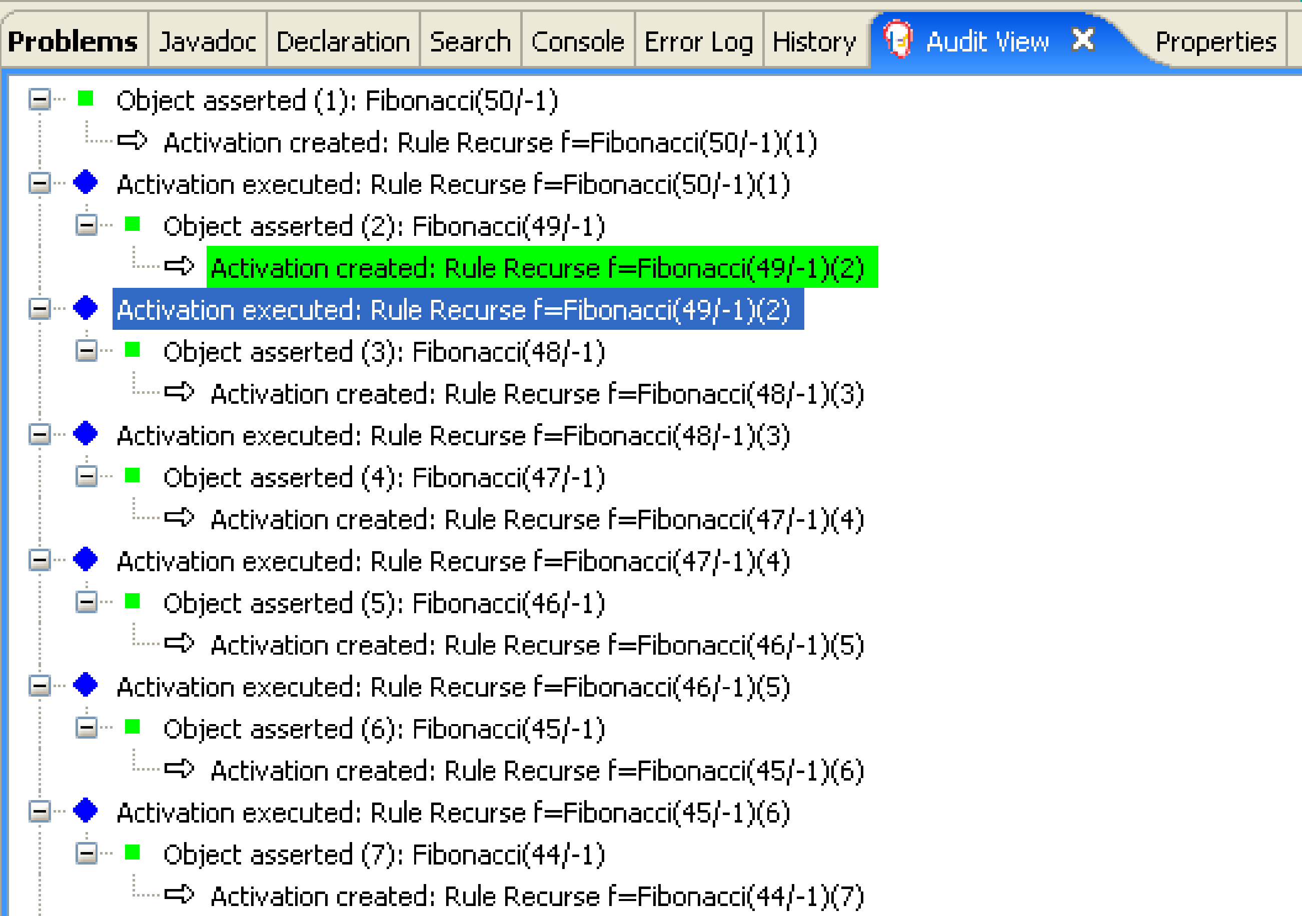
Rule "Recurse"

```
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
```

To better understand the execution flow of this example, you can load the audit log file from **target/fibonacci.log** into your IDE debug view or **Audit View**, if available (for example, in **Window → Show View** in some IDEs).

In this example, 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 7.7. Rule "Recurse" in Audit View



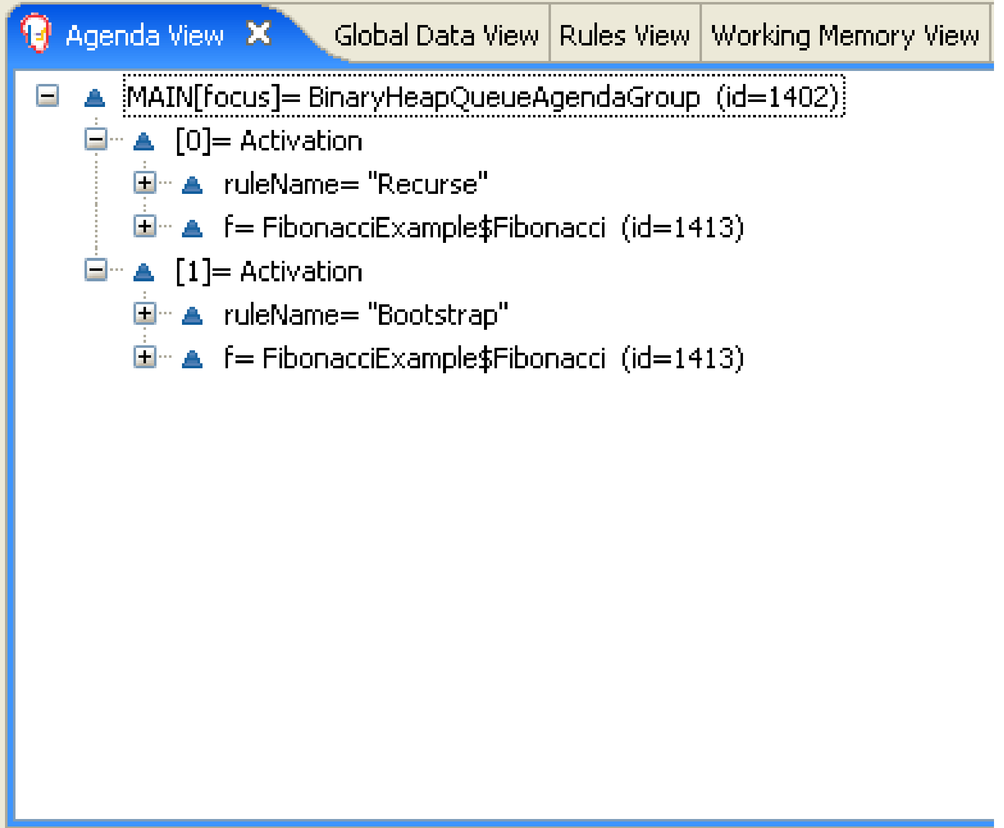
When a **Fibonacci** object with a **sequence** field of **2** is asserted, the **"Bootstrap"** rule is matched and activated along with the **"Recurse"** rule. Notice the multiple restrictions on field **sequence** that test for equality with **1** or **2**:

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
```

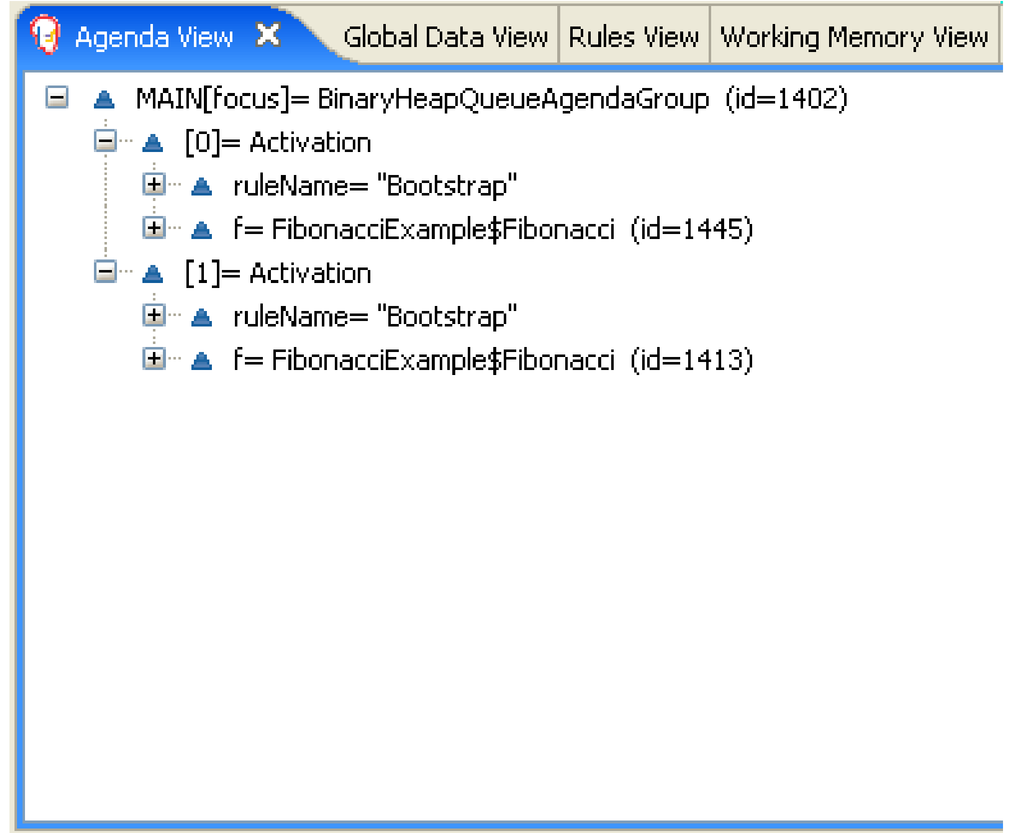
You can also use the **Agenda View** in your IDE to investigate the state of the engine agenda. The **"Bootstrap"** rule does not fire yet because the **"Recurse"** rule has a higher salience value.

Figure 7.8. Rules "Recurse" and "Bootstrap" in 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. The **"Recurse"** rule does not match and activate because the **not** conditional element stops the rule matching as soon as a **Fibonacci** object with a **sequence** of **1** exists.

Figure 7.9. Rules "Recurse" and "Bootstrap" in Agenda View 2



The **"Bootstrap"** rule sets the objects with a **sequence** of **1** and **2** to a value of **1**. Now that you have two **Fibonacci** objects with values not equal to **-1**, the **"Calculate"** rule is able to match.

At this point in the example, nearly 50 **Fibonacci** objects exist in the working memory. You need to select a suitable triple to calculate each of their values in turn. If you use three Fibonacci patterns in a rule without field constraints to confine the possible cross products, the result would be 50x49x48 possible combinations, leading to about 125,000 possible rule firings, most of them incorrect.

The **"Calculate"** rule uses field constraints to evaluate the three Fibonacci patterns in the correct order. This technique is called *cross-product matching*.

The first pattern finds any **Fibonacci** object with a value **!= -1** and binds both the pattern and the field. The second **Fibonacci** object does the same thing, but adds an additional field constraint to ensure that its sequence is greater by one than the **Fibonacci** object bound to **f1**. When this rule fires for the first time, you know that only sequences **1** and **2** have values of **1**, and the two constraints ensure that **f1** references sequence **1** and that **f2** references sequence **2**.

The final pattern finds the **Fibonacci** object with a value equal to **-1** and with a sequence one greater than **f2**.

At this point in the example, three **Fibonacci** objects are correctly selected from the available cross products, and you can calculate the value for the third **Fibonacci** object that is bound to **f3**.

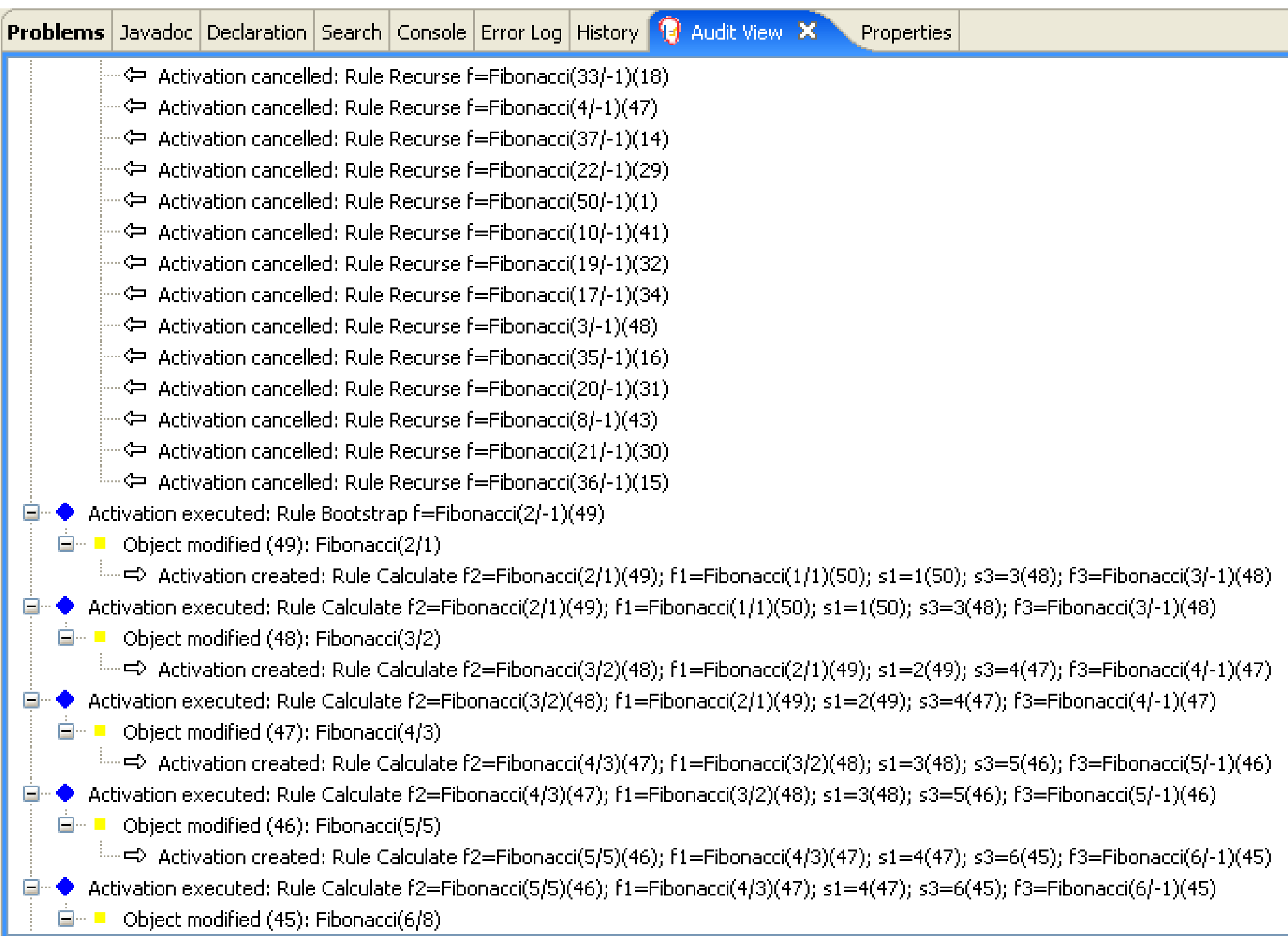
Rule "Calculate"

```
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 techniques.
    modify ( f3 ) { value = f1.value + v2 };
    System.out.println( s3 + " == " + f3.value );
end
```

The **modify** statement updates the value of the **Fibonacci** object bound to **f3**. This means that you now have another new **Fibonacci** object with a value not equal to **-1**, which allows the **"Calculate"** rule to re-match and calculate the next Fibonacci number.

The debug view or **Audit View** of your IDE shows how the firing of the last **"Bootstrap"** rule modifies the **Fibonacci** object, enabling the **"Calculate"** rule to match, which then modifies another **Fibonacci** object that enables the **"Calculate"** rule to match again. This process continues until the value is set for all **Fibonacci** objects.

Figure 7.10. Rules in Audit View



7.5. PRICING EXAMPLE DECISIONS (DECISION TABLES)

The Pricing example decision set demonstrates how to use a spreadsheet decision table for calculating the retail cost of an insurance policy in tabular format instead of directly in a DRL file.

The following is an overview of the Pricing example:

- Name: **decisiontable**
- Main class: **org.drools.examples.decisiontable.PricingRuleDTExample** (in **src/main/java**)
- Module: **drools-examples**
- **Type**: Java application
- Rule file: **org.drools.examples.decisiontable.ExamplePolicyPricing.xls** (in **src/main/resources**)
- **Objective**: Demonstrates use of spreadsheet decision tables to define rules

Decision tables are XLS or XLSX spreadsheets that you can use to define business rules in a tabular format and that you can include in your Red Hat Decision Manager project or upload in Decision Central. Each row in the spreadsheet is a rule, and each column is a condition, an action, or another rule attribute. After you create and upload your decision tables into your Red Hat Decision Manager project, the rules you defined are compiled into Drools Rule Language (DRL) rules as with all other rule assets.

The purpose of the Pricing example is to provide a set of business rules to calculate the base price and a

discount for a car driver applying for a specific type of insurance policy. The driver's age and history and the policy type all contribute to calculate the basic premium, and additional rules calculate potential discounts for which the driver might be eligible.

To execute the example, run the **org.drools.examples.decisiontable.PricingRuleDTExample** class as a Java application in your IDE.

After the execution, the following output appears in the IDE console window:

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

The code to execute the example follows the typical execution pattern: the rules are loaded, the facts are inserted, and a stateless KIE session is created. The difference in this example is that the rules are defined in an **ExamplePolicyPricing.xls** file instead of a DRL file or other source. The spreadsheet file is loaded into the decision engine using templates and DRL rules.

Spreadsheet decision table setup

The **ExamplePolicyPricing.xls** spreadsheet contains two decision tables in the first tab:

- **Base pricing rules**
- **Promotional discount rules**

As the example spreadsheet demonstrates, you can use only the first tab of a spreadsheet to create decision tables, but multiple tables can be within a single tab. Decision tables do not necessarily follow top-down logic, but are more of a means to capture data resulting in rules. The evaluation of the rules is not necessarily in the given order, because all of the normal mechanics of the decision engine still apply. This is why you can have multiple decision tables in the same tab of a spreadsheet.

The decision tables are executed through the corresponding rule template files **BasePricing.drt** and **PromotionalPricing.drt**. These template files reference the decision tables through their template parameter and directly reference the various headers for the conditions and actions in the decision tables.

BasePricing.drt rule template file

```
template header
age[]
profile
priorClaims
policyType
base
reason

package org.drools.examples.decisiontable;

template "Pricing bracket"
age
policyType
base

rule "Pricing bracket_@{row.rowNumber}"
  when
    Driver(age >= @{age0}, age <= @{age1}
```

```
        , priorClaims == "@{priorClaims}"
        , locationRiskProfile == "@{profile}"
    )
    policy: Policy(type == "@{policyType}")
  then
    policy.setBasePrice(@{base});
    System.out.println("@{reason}");
end
end template
```

PromotionalPricing.drt rule template file

```
template header
age[]
priorClaims
policyType
discount

package org.drools.examples.decisiontable;

template "discounts"
age
priorClaims
policyType
discount

rule "Discounts_@{row.rowNumber}"
  when
    Driver(age >= @{age0}, age <= @{age1}, priorClaims == "@{priorClaims}")
    policy: Policy(type == "@{policyType}")
  then
    policy.applyDiscount(@{discount});
end
end template
```

The rules are executed through the **kmodule.xml** reference of the KIE Session **DTableWithTemplateKB**, which specifically mentions the **ExamplePolicyPricing.xls** spreadsheet and is required for successful execution of the rules. This execution method enables you to execute the rules as a standalone unit (as in this example) or to include the rules in a packaged knowledge JAR (KJAR) file, so that the spreadsheet is packaged along with the rules for execution.

The following section of the **kmodule.xml** file is required for the execution of the rules and spreadsheet to work successfully:

```
    <kbase name="DecisionTableKB" packages="org.drools.examples.decisiontable">
        <ksession name="DecisionTableKS" type="stateless"/>
    </kbase>

    <kbase name="DTableWithTemplateKB" packages="org.drools.examples.decisiontable-template">
        <ruleTemplate dtable="org/drools/examples/decisiontable-
template/ExamplePolicyPricingTemplateData.xls"
                      template="org/drools/examples/decisiontable-template/BasePricing.drt"
                      row="3" col="3"/>
        <ruleTemplate dtable="org/drools/examples/decisiontable-
template/ExamplePolicyPricingTemplateData.xls"
```

```
            template="org/drools/examples/decisiontable-template/PromotionalPricing.drt"
            row="18" col="3"/>
    <ksession name="DTableWithTemplateKS"/>
  </kbase>
```

As an alternative to executing the decision tables using rule template files, you can use the **DecisionTableConfiguration** object and specify an input spreadsheet as the input type, such as **DecisionTableInputType.xls**:

```
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 );
```

The Pricing example uses two fact types:

- **Driver**
- **Policy**.

The example sets the default values for both facts in their respective Java classes **Driver.java** and **Policy.java**. The **Driver** is 30 years old, has had no prior claims, and currently has a risk profile of **LOW**. The **Policy** that the driver is applying for is **COMPREHENSIVE**.

In any decision table, each row is considered a different rule and each column is a condition or an action. Each row is evaluated in a decision table unless the agenda is cleared upon execution.

Decision table spreadsheets require two key areas that define rule data:

- A **RuleSet** area
- A **RuleTable** area

The **RuleSet** area of the spreadsheet defines elements that you want to apply globally to all rules in the same package (not only the spreadsheet), such as a rule set name or universal rule attributes. The **RuleTable** area defines the actual rules (rows) and the conditions, actions, and other rule attributes (columns) that constitute that rule table within the specified rule set. A decision table spreadsheet can contain multiple **RuleTable** areas, but only one **RuleSet** area.

Figure 7.11. Decision table configuration

C	D	E	F	G	H
RuleSet	org.drools.examples.decisiontable				
Notes	This decision table is for working out some basic prices and pretending actuaries don't exist				
RuleTable Pricing bracket					
CONDITION	CONDITION	CONDITION	CONDITION	ACTION	ACTION
Driver			policy: Policy		
age >= $1, age <= $2	locationRiskProfile	priorClaims	type	policy.setBasePrice($param);	System.out.println("$param");
Age Bracket	Location risk profile	Number of prior claims	Policy type applying for	Base $ AUD	Record Reason

The **RuleTable** area also defines the objects to which the rule attributes apply, in this case **Driver** and **Policy**, followed by constraints on the objects. For example, the **Driver** object constraint that defines the **Age Bracket** column is **age >= $1, age <= $2**, where the comma-separated range is defined in the table column values, such as **18,24**.

Base pricing rules

The **Base pricing rules** decision table in the Pricing example evaluates the age, risk profile, number of claims, and policy type of the driver and produces the base price of the policy based on these conditions.

Figure 7.12. Base price calculation

	B	C	D	E	F	G	H
9	Base pricing rules	Age Bracket	Location risk profile	Number of prior claims	Policy type applying for	Base $ AUD	Record Reason
10			LOW	1	COMPREHENSIVE	450	
11			MED		FIRE_THEFT	200	Priors not relevant
12	Young safe package	18, 24	MED	0	COMPREHENSIVE	300	
13			LOW		FIRE_THEFT	150	
14			LOW	0	COMPREHENSIVE	150	Safe driver discount
15		18,24	MED	1	COMPREHENSIVE	700	
16	Young risk	18,24	HIGH	0	COMPREHENSIVE	700	Location risk
17		18,24	HIGH		FIRE_THEFT	550	Location risk
18		25,30		0	COMPREHENSIVE	120	Cheapest possible
19		25,30		1	COMPREHENSIVE	300	
20	Mature drivers	25,30		2	COMPREHENSIVE	590	
21		25,35		3	THIRD_PARTY	800	High risk

The **Driver** attributes are defined in the following table columns:

- **Age Bracket**: The age bracket has a definition for the condition **age >=$1, age <=$2**, which defines the condition boundaries for the driver's age. This condition column highlights the use of **$1 and $2**, which is comma delimited in the spreadsheet. You can write these values as **18,24** or **18, 24** and both formats work in the execution of the business rules.

- **Location risk profile**: The risk profile is a string that the example program passes always as **LOW** but can be changed to reflect **MED** or **HIGH**.

- **Number of prior claims**: The number of claims is defined as an integer that the condition column must exactly equal to trigger the action. The value is not a range, only exact matches.

The **Policy** of the decision table is used in both the conditions and the actions of the rule and has attributes defined in the following table columns:

- **Policy type applying for**: The policy type is a condition that is passed as a string that defines the type of coverage: **COMPREHENSIVE**, **FIRE_THEFT**, or **THIRD_PARTY**.

- **Base $ AUD**: The **basePrice** is defined as an **ACTION** that sets the price through the constraint **policy.setBasePrice($param);** based on the spreadsheet cells corresponding to this value. When you execute the corresponding DRL rule for this decision table, the **then** portion of the rule executes this action statement on the true conditions matching the facts and sets the base price to the corresponding value.

- **Record Reason**: When the rule successfully executes, this action generates an output message to the **System.out** console reflecting which rule fired. This is later captured in the application and printed.

The example also uses the first column on the left to categorize rules. This column is for annotation only and has no affect on rule execution.

Promotional discount rules

The **Promotional discount rules** decision table in the Pricing example evaluates the age, number of prior claims, and policy type of the driver to generate a potential discount on the price of the insurance policy.

Figure 7.13. Discount calculation

29	Promotional discount rules	Age Bracket	Number of prior claims	Policy type applying for	Discount %
30	Rewards for safe drivers	18,24	0	COMPREHENSIVE	1
31		18,24	0	FIRE_THEFT	2
32		25,30	1	COMPREHENSIVE	5
33		25,30	2	COMPREHENSIVE	1
34		25,30	0	COMPREHENSIVE	20

This decision table contains the conditions for the discount for which the driver might be eligible. Similar to the base price calculation, this table evaluates the **Age**, **Number of prior claims** of the driver, and the **Policy type applying for** to determine a **Discount %** rate to be applied. For example, if the driver is 30 years old, has no prior claims, and is applying for a **COMPREHENSIVE** policy, the driver is given a discount of **20** percent.

7.6. PET STORE EXAMPLE DECISIONS (AGENDA GROUPS, GLOBAL VARIABLES, CALLBACKS, AND GUI INTEGRATION)

The Pet Store example decision set demonstrates how to use agenda groups and global variables in rules and how to integrate Red Hat Decision Manager rules with a graphical user interface (GUI), in this case a Swing-based desktop application. The example also demonstrates how to use callbacks to interact with a running decision engine to update the GUI based on changes in the working memory at run time.

The following is an overview of the Pet Store example:

- Name: **petstore**
- Main class: **org.drools.examples.petstore.PetStoreExample** (in **src/main/java**)
- Module: **drools-examples**
- **Type**: Java application
- Rule file: **org.drools.examples.petstore.PetStore.drl** (in **src/main/resources**)

- **Objective**: Demonstrates rule agenda groups, global variables, callbacks, and GUI integration

In the Pet Store example, the sample **PetStoreExample.java** class defines the following principal classes (in addition to several classes to handle Swing events):

- **Petstore** contains the **main()** method.
- **PetStoreUI** is responsible for creating and displaying the Swing-based GUI. This class contains several smaller classes, mainly for responding to various GUI events, such as user mouse clicks.
- **TableModel** holds the table data. This class is essentially a JavaBean that extends the Swing class **AbstractTableModel**.
- **CheckoutCallback** enables the GUI to interact with the rules.
- **Ordershow** keeps the items that you want to buy.
- **Purchase** stores details of the order and the products that you are buying.
- **Product** is a JavaBean containing details of the product available for purchase and its price.

Much of the Java code in this example is either plain JavaBean or Swing based. For more information about Swing components, see the Java tutorial on Creating a GUI with JFC/Swing .

Rule execution behavior in the Pet Store example

Unlike other example decision sets where the facts are asserted and fired immediately, the Pet Store example does not execute the rules until more facts are gathered based on user interaction. The example executes rules through a **PetStoreUI** object, created by a constructor, that accepts the **Vector** object **stock** for collecting the products. The example then uses an instance of the **CheckoutCallback** class containing the rule base that was previously loaded.

Pet Store KIE container and fact execution setup

```
// KieServices is the factory for all KIE services.
KieServices ks = KieServices.Factory.get();

// Create a KIE container on the class path.
KieContainer kc = ks.getKieClasspathContainer();

// 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( kc ) );
ui.createAndShowGUI();
```

The Java code that fires the rules is in the **CheckoutCallBack.checkout()** method. This method is triggered when the user clicks **Checkout** in the UI.

Rule execution 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.

        // From the KIE container, a KIE session is created based on
        // its definition and configuration in the META-INF/kmodule.xml file.
        KieSession ksession = kcontainer.newKieSession("PetStoreKS");

        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 );

        // Execute rules.
        ksession.fireAllRules();

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

The example code passes two elements into the **CheckoutCallBack.checkout()** method. One element is the handle for the **JFrame** Swing component surrounding the output text frame, found at the bottom of the GUI. The second element is a list of order items, which comes from the **TableModel** that stores the information from the **Table** area at the upper-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 **PetStoreExample.java**.

In this case, the rule is firing in a stateless KIE session because all of the data is stored in Swing components and is not executed until the user clicks **Checkout** in the UI. Each time the user clicks **Checkout**, the content of the list is moved from the Swing **TableModel** into the KIE session working memory and is then executed with the **ksession.fireAllRules()** method.

Within this code, there are nine calls to **KieSession**. The first of these creates a new **KieSession** from the **KieContainer** (the example passed in this **KieContainer** from the **CheckoutCallBack** class in the **main()** method). The next two calls pass in the two objects that hold the 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 **KieSession**, as well as the order list. The final call is the standard **fireAllRules()**.

Pet Store rule file imports, global variables, and Java functions

The **PetStore.drl** file contains the standard package and import statements to make various Java classes available to the rules. The rule file also includes *global variables* to be used within the rules, defined as **frame** and **textArea**. The global variables hold references to the Swing components **JFrame** and **JTextArea** components that were previously passed on by the Java code that called the

setGlobal() method. Unlike standard variables in rules, which expire as soon as the rule has fired, global variables retain their value for the lifetime of the KIE session. This means the contents of these global variables are available for evaluation on all subsequent rules.

PetStore.drl package, imports, and global variables

```
package org.drools.examples;

import org.kie.api.runtime.KieRuntime;
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 **PetStore.drl** file also contains two functions that the rules in the file use:

PetStore.drl Java functions

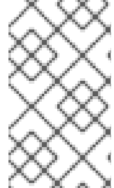
```
function void doCheckout(JFrame frame, KieRuntime krt) {
        Object[] options = {"Yes",
                            "No"};

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

       if (n == 0) {
            krt.getAgenda().getAgendaGroup( "checkout" ).setFocus();
       }
}

function boolean requireTank(JFrame frame, KieRuntime krt, 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 );
            krt.insert( purchase );
            order.addItem( purchase );
            System.out.println( "Yes" );
        } else {
            System.out.println( "No" );
        }
        return true;
    }
```

The two functions perform the following actions:

- **doCheckout()** displays a dialog that asks the user if she or he wants to check out. If the user does, the focus is set to the **checkout** agenda group, enabling rules in that group to (potentially) fire.

- **requireTank()** displays a dialog that asks the user if she or he wants to buy a fish tank. If the user does, a new fish tank **Product** is added to the order list in the working memory.



NOTE

For this example, all rules and functions are within the same rule file for efficiency. In a production environment, you typically separate the rules and functions in different files or build a static Java method and import the files using the import function, such as **import function my.package.name.hello**.

Pet Store rules with agenda groups

Most of the rules in the Pet Store example use agenda groups to control rule execution. Agenda groups allow you to partition the engine agenda to provide more execution control over groups of rules. By default, all rules are in the agenda group **MAIN**. You can use the **agenda-group** attribute to specify a different agenda group for the rule.

Initially, a working memory has its focus on the agenda group **MAIN**. Rules in an agenda group only fire when the group receives the focus. You can set the focus either by using the method **setFocus()** or the rule attribute **auto-focus**. The **auto-focus** attribute enables the rule to be given a focus automatically for its agenda group when the rule is matched and activated.

The Pet Store example uses the following agenda groups for rules:

- **"init"**
- **"evaluate"**
- **"show items"**
- **"checkout"**

For example, the sample rule **"Explode Cart"** uses the **"init"** agenda group to ensure that it has the option to fire and insert shopping cart items into the KIE session working memory:

Rule "Explode Cart"

```
// 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. The execution loops for each purchase item in that order.

The rule uses the following features related to its agenda group:

- **agenda-group "init"** defines the name of the agenda group. In this case, only one rule is in the group. However, neither the Java code nor a rule consequence sets the focus to this group, and therefore it relies on the **auto-focus** 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, enabling their rules to fire. In practice, you loop through all items in the order, insert them into memory, and then fire the other rules after each insertion.

The **"show items"** agenda group contains only one rule, **"Show Items"**. For each purchase in the order currently in the KIE session working memory, the rule logs details to the text area at the bottom of the GUI, based on the **textArea** variable defined in the rule file.

Rule "Show Items"

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

The **"evaluate"** agenda group also gains focus from the **"Explode Cart"** rule. This agenda group contains two rules, **"Free Fish Food Sample"** and **"Suggest Tank"**, which are executed in that order.

Rule "Free Fish Food Sample"

```
// Free fish food sample when users buy a goldfish if they did not already buy
// fish food and do not already have a fish food sample.
rule "Free Fish Food Sample"
```

```
    agenda-group "evaluate" 1
    dialect "mvel"
  when
    $order : Order()
    not ( $p : Product( name == "Fish Food") && Purchase( product == $p ) ) 2
    not ( $p : Product( name == "Fish Food Sample") && Purchase( product == $p ) ) 3
    exists ( $p : Product( name == "Gold Fish") && Purchase( product == $p ) ) 4
    $fishFoodSample : Product( name == "Fish Food Sample" );
  then
    System.out.println( "Adding free Fish Food Sample to cart" );
    purchase = new Purchase($order, $fishFoodSample);
    insert( purchase );
    $order.addItem( purchase );
end
```

The rule **"Free Fish Food Sample"** fires only if all of the following conditions are true:

1. The agenda group **"evaluate"** is being evaluated in the rules execution.
2. User does not already have fish food.
3. User does not already have a free fish food sample.
4. User has a goldfish in the order.

If the order facts meet all of these requirements, then a new product is created (Fish Food Sample) and is added to the order in working memory.

Rule "Suggest Tank"

```
// Suggest a fish tank if users buy more than five goldfish and
// do not already have a tank.
rule "Suggest Tank"
    agenda-group "evaluate"
    dialect "java"
  when
    $order : Order()
    not ( $p : Product( name == "Fish Tank") && Purchase( product == $p ) ) 1
    ArrayList( $total : size > 5 ) from collect( Purchase( product.name == "Gold Fish" ) ) 2
    $fishTank : Product( name == "Fish Tank" )
  then
    requireTank(frame, kcontext.getKieRuntime(), $order, $fishTank, $total);
end
```

The rule **"Suggest Tank"** fires only if the following conditions are true:

1. User does not have a fish tank in the order.
2. User has more than five fish in the order.

When the rule fires, it calls the **requireTank()** function defined in the rule file. This function displays a dialog that asks the user if she or he wants to buy a fish tank. If the user does, a new fish tank **Product** is added to the order list in the working memory. When the rule calls the **requireTank()** function, the rule

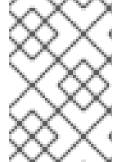
passes the **frame** global variable so that the function has a handle for the Swing GUI.

The **"do checkout"** rule in the Pet Store example has no agenda group and no **when** conditions, so the rule is always executed and considered part of the default **MAIN** agenda group.

Rule "do checkout"

```
rule "do checkout"
    dialect "java"
  when
  then
    doCheckout(frame, kcontext.getKieRuntime());
end
```

When the rule fires, it calls the **doCheckout()** function defined in the rule file. This function displays a dialog that asks the user if she or he wants to check out. If the user does, the focus is set to the **checkout** agenda group, enabling rules in that group to (potentially) fire. When the rule calls the **doCheckout()** function, the rule passes the **frame** global variable so that the function has a handle for the Swing GUI.



NOTE

This example also demonstrates a troubleshooting technique if results are not executing as you expect: You can remove the conditions from the **when** statement of a rule and test the action in the **then** statement to verify that the action is performed correctly.

The **"checkout"** agenda group contains three rules for processing the order checkout and applying any discounts: **"Gross Total"**, **"Apply 5% Discount"**, and **"Apply 10% Discount"**.

Rules "Gross Total", "Apply 5% Discount", and "Apply 10% Discount"

```
rule "Gross Total"
    agenda-group "checkout"
    dialect "mvel"
  when
    $order : Order( grossTotal == -1)
    Number( total : doubleValue ) from accumulate( Purchase( $price : product.price ),
                                                   sum( $price ) )
  then
    modify( $order ) { grossTotal = total }
    textArea.append( "\ngross total=" + total + "\n" );
end

rule "Apply 5% Discount"
    agenda-group "checkout"
    dialect "mvel"
  when
    $order : Order( grossTotal >= 10 && < 20 )
  then
    $order.discountedTotal = $order.grossTotal * 0.95;
    textArea.append( "discountedTotal total=" + $order.discountedTotal + "\n" );
end

rule "Apply 10% Discount"
    agenda-group "checkout"
```

```
    dialect "mvel"
  when
    $order : Order( grossTotal >= 20 )
  then
    $order.discountedTotal = $order.grossTotal * 0.90;
    textArea.append( "discountedTotal total=" + $order.discountedTotal + "\n" );
end
```

If the user has not already calculated the gross total, the **Gross Total** accumulates the product prices into a total, puts this total into the KIE session, and displays it through the Swing **JTextArea** using the **textArea** global variable.

If the gross total is between **10** and **20** (currency units), the **"Apply 5% Discount"** rule calculates the discounted total, adds it to the KIE session, and displays it in the text area.

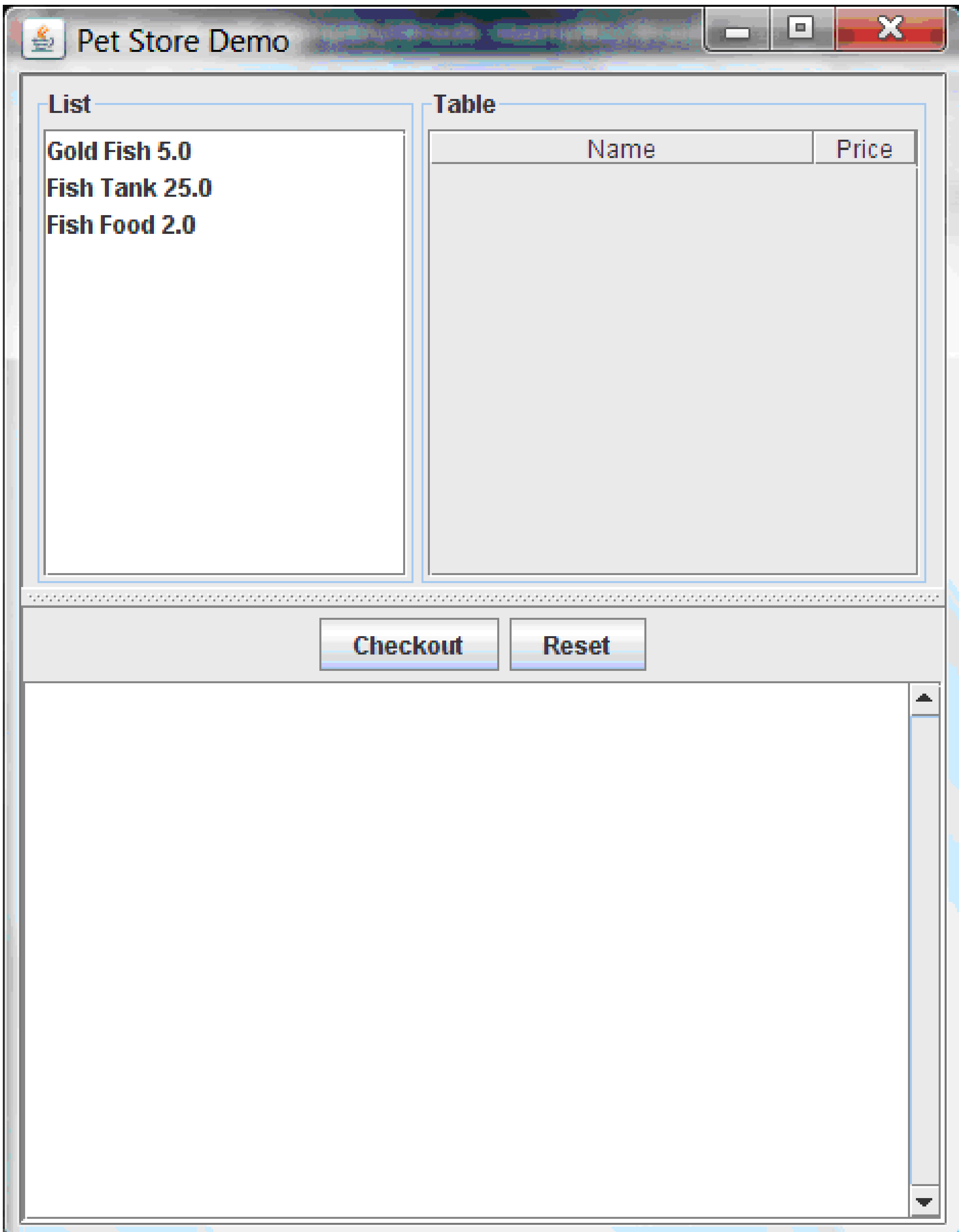
If the gross total is not less than **20**, the **"Apply 10% Discount"** rule calculates the discounted total, adds it to the KIE session, and displays it in the text area.

Pet Store example execution

Similar to other Red Hat Decision Manager decision examples, you execute the Pet Store example by running the **org.drools.examples.petstore.PetStoreExample** class as a Java application in your IDE.

When you execute the Pet Store example, the **Pet Store Demo** GUI window appears. This window displays a list of available products (upper left), an empty list of selected products (upper right), **Checkout** and **Reset** buttons (middle), and an empty system messages area (bottom).

Figure 7.14. Pet Store example GUI after launch



The following events occurred in this example to establish this execution behavior:

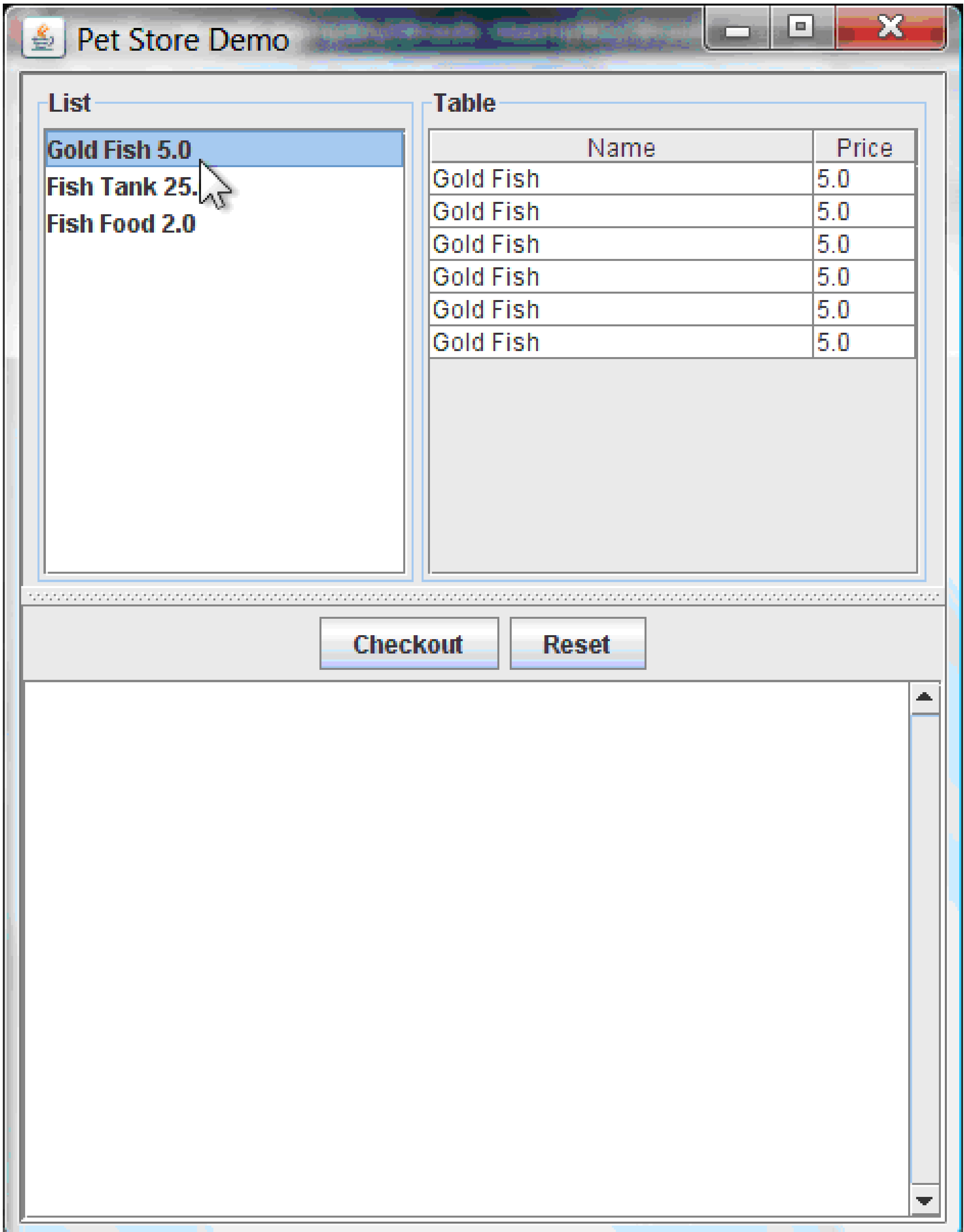
1. The **main()** method has run and loaded the rule base but has not yet fired the rules. So far, this is the only code in connection with rules that has been run.

2. A new **PetStoreUI** object has been created and given a handle for the rule base, for later use.

3. Various Swing components have performed their functions, and the initial UI screen is displayed and waits for user input.

You can click on various products from the list to explore the UI setup:

Figure 7.15. Explore the Pet Store example GUI



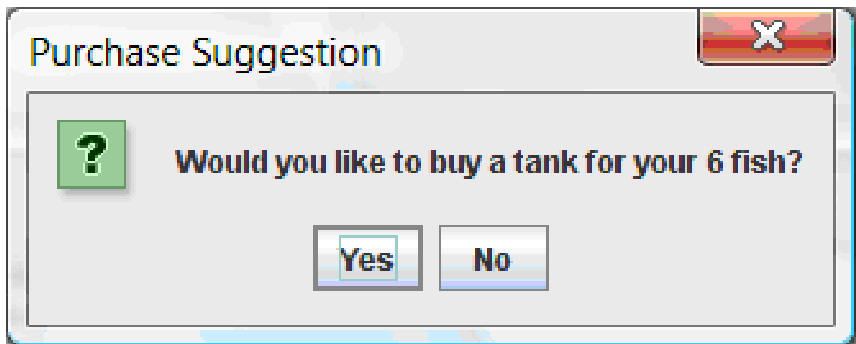
No rules code has been fired yet. The UI uses Swing code to detect user mouse clicks and add selected products to the **TableModel** object for display in the upper-right corner of the UI. This example illustrates the Model-View-Controller design pattern.

When you click **Checkout**, the rules are then fired in the following way:

1. Method **CheckOutCallBack.checkout()** is called (eventually) by the Swing class waiting for the click on **Checkout**. This inserts the data from the **TableModel** object (upper-right corner of the UI) into the KIE session working memory. The method then fires the rules.

2. The **"Explode Cart"** rule is the first to fire, with the **auto-focus** attribute set to **true**. The rule loops through all of the products in the cart, ensures that the products are in the working memory, and then gives the **"show Items"** and **"evaluate"** agenda groups the option to fire. The rules in these groups add the contents of the cart to the text area (bottom of the UI), evaluate if you are eligible for free fish food, and determine whether to ask if you want to buy a fish tank.

Figure 7.16. Fish tank qualification



3. The **"do checkout"** rule is the next to fire because no other agenda group currently has focus and because it is part of the default **MAIN** agenda group. This rule always calls the **doCheckout()** function, which asks you if you want to check out.

4. The **doCheckout()** function sets the focus to the **"checkout"** agenda group, giving the rules in that group the option to fire.

5. The rules in the **"checkout"** agenda group display the contents of the cart and apply the appropriate discount.

6. Swing then waits for user input to either select more products (and cause the rules to fire again) or to close the UI.

Figure 7.17. Pet Store example GUI after all rules have fired

You can add more **System.out** calls to demonstrate this flow of events in your IDE console:

System.out output in the IDE console

```
Adding free Fish Food Sample to cart
SUGGESTION: Would you like to buy a tank for your 6 fish? - Yes
```

7.7. HONEST POLITICIAN EXAMPLE DECISIONS (TRUTH MAINTENANCE AND SALIENCE)

The Honest Politician example decision set demonstrates the concept of truth maintenance with logical insertions and the use of salience in rules.

The following is an overview of the Honest Politician example:

- Name: **honestpolitician**
- Main class: **org.drools.examples.honestpolitician.HonestPoliticianExample** (in **src/main/java**)
- Module: **drools-examples**
- **Type**: Java application
- Rule file: **org.drools.examples.honestpolitician.HonestPolitician.drl** (in **src/main/resources**)
- **Objective**: Demonstrates the concept of truth maintenance based on the logical insertion of facts and the use of salience in rules

The basic premise of the Honest Politician example is that an object can only exist while a statement is true. A rule consequence can logically insert an object with the **insertLogical()** method. This means the object remains in the KIE session 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, rule execution causes a group of politicians to change from being honest to being dishonest as a result of a corrupt corporation. As each politician is evaluated, they start out with their honesty attribute being set to **true**, but a rule fires that makes the politicians no longer honest. As they switch their state from being honest to dishonest, they are then removed from the working memory. The rule salience notifies the engine how to prioritize any rules that have a salience defined for them, otherwise utilizing the default salience value of **0**. Rules with a higher salience value are given higher priority when ordered in the activation queue.

Politician and Hope classes

The sample class **Politician** in the example is configured for an honest politician. The **Politician** class is made up of a String item **name** and a Boolean item **honest**:

Politician class

```
public class Politician {
    private String name;
    private boolean honest;
    ...
}
```

The **Hope** class determines if a **Hope** object exists. This class has no meaningful members, but is present in the working memory as long as society has hope.

Hope class

```
public class Hope {

    public Hope() {

    }
  }
```

Rule definitions for politician honesty

In the Honest Politician example, when at least one honest politician exists in the working memory, the **"We have an honest Politician"** rule logically inserts a new **Hope** object. As soon as all politicians become dishonest, the **Hope** object is automatically retracted. This rule has a **salience** attribute with a value of **10** to ensure that it fires before any other rule, because at that stage the **"Hope is Dead"** rule is true.

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. This rule also has a **salience** value of **10** so that it takes priority over the **"Corrupt the Honest"** rule.

Rule "Hope Lives"

```
rule "Hope Lives"
    salience 10
  when
    exists( Hope() )
  then
    System.out.println("Hurrah!!! Democracy Lives");
end
```

Initially, four honest politicians exist so this rule has four activations, all in conflict. Each rule fires in turn, corrupting each politician so that they are no longer honest. When all four politicians have been corrupted, no politicians have the property **honest == true**. The rule **"We have an honest Politician"** is no longer true and the object it logically inserted (due to the last execution of **new Hope()**) is automatically retracted.

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 automatically retracted through the truth maintenance system, the conditional element **not** applied to **Hope** is no longer true so that the **"Hope is Dead"** rule matches and fires.

Rule "Hope is Dead"

```
rule "Hope is Dead"
  when
```

```
    not( Hope() )
  then
    System.out.println( "We are all Doomed!!! Democracy is Dead" );
end
```

Example execution and audit trail

In the **HonestPoliticianExample.java** class, the four politicians with the honest state set to **true** are inserted for evaluation against the defined business rules:

HonestPoliticianExample.java class execution

```
public static void execute( KieContainer kc ) {
        KieSession ksession = kc.newKieSession("HonestPoliticianKS");

        final Politician p1 = new Politician( "President of Umpa Lumpa", true );
        final Politician p2 = new Politician( "Prime Minster of Cheeseland", true );
        final Politician p3 = new Politician( "Tsar of Pringapopaloo", true );
        final Politician p4 = new Politician( "Omnipotence Om", true );

        ksession.insert( p1 );
        ksession.insert( p2 );
        ksession.insert( p3 );
        ksession.insert( p4 );

        ksession.fireAllRules();

        ksession.dispose();
    }
```

To execute the example, run the **org.drools.examples.honestpolitician.HonestPoliticianExample** class as a Java application in your IDE.

After the execution, the following output appears in the IDE console window:

Execution output in the IDE console

```
Hurrah!!! Democracy Lives
I'm an evil corporation and I have corrupted President of Umpa Lumpa
I'm an evil corporation and I have corrupted Prime Minster of Cheeseland
I'm an evil corporation and I have corrupted Tsar of Pringapopaloo
I'm an evil corporation and I have corrupted Omnipotence Om
We are all Doomed!!! Democracy is Dead
```

The output shows that, while there is at least one honest politician, democracy lives. However, as each politician is corrupted by some corporation, all politicians become dishonest, and democracy is dead.

To better understand the execution flow of this example, you can modify the **HonestPoliticianExample.java** class to include a **RuleRuntime** listener and an audit logger to view execution details:

HonestPoliticianExample.java class with an audit logger

```
package org.drools.examples.honestpolitician;

import org.kie.api.KieServices;
```

```
import org.kie.api.event.rule.DebugAgendaEventListener; 1
import org.kie.api.event.rule.DebugRuleRuntimeEventListener;
import org.kie.api.runtime.KieContainer;
import org.kie.api.runtime.KieSession;

public class HonestPoliticianExample {

    /**
     * @param args
     */
    public static void main(final String[] args) {
     KieServices ks = KieServices.Factory.get(); 2
     //ks = KieServices.Factory.get();
        KieContainer kc = KieServices.Factory.get().getKieClasspathContainer();
        System.out.println(kc.verify().getMessages().toString());
        //execute( kc );
        execute( ks, kc); 3
    }

    public static void execute( KieServices ks, KieContainer kc ) { 4
        KieSession ksession = kc.newKieSession("HonestPoliticianKS");

        final Politician p1 = new Politician( "President of Umpa Lumpa", true );
        final Politician p2 = new Politician( "Prime Minster of Cheeseland", true );
        final Politician p3 = new Politician( "Tsar of Pringapopaloo", true );
        final Politician p4 = new Politician( "Omnipotence Om", true );

        ksession.insert( p1 );
        ksession.insert( p2 );
        ksession.insert( p3 );
        ksession.insert( p4 );

        // The application can also setup listeners 5
        ksession.addEventListener( new DebugAgendaEventListener() );
        ksession.addEventListener( new DebugRuleRuntimeEventListener() );

        // Set up a file-based audit logger.
        ks.getLoggers().newFileLogger( ksession, "./target/honestpolitician" ); 6

        ksession.fireAllRules();

        ksession.dispose();
    }

}
```

1. Adds to your imports the packages that handle the **DebugAgendaEventListener** and **DebugRuleRuntimeEventListener**
2. Creates a **KieServices Factory** and a **ks** element to produce the logs because this audit log is not available at the **KieContainer** level
3. Modifies the **execute** method to use both **KieServices** and **KieContainer**
4. Modifies the **execute** method to pass in **KieServices** in addition to the **KieContainer**

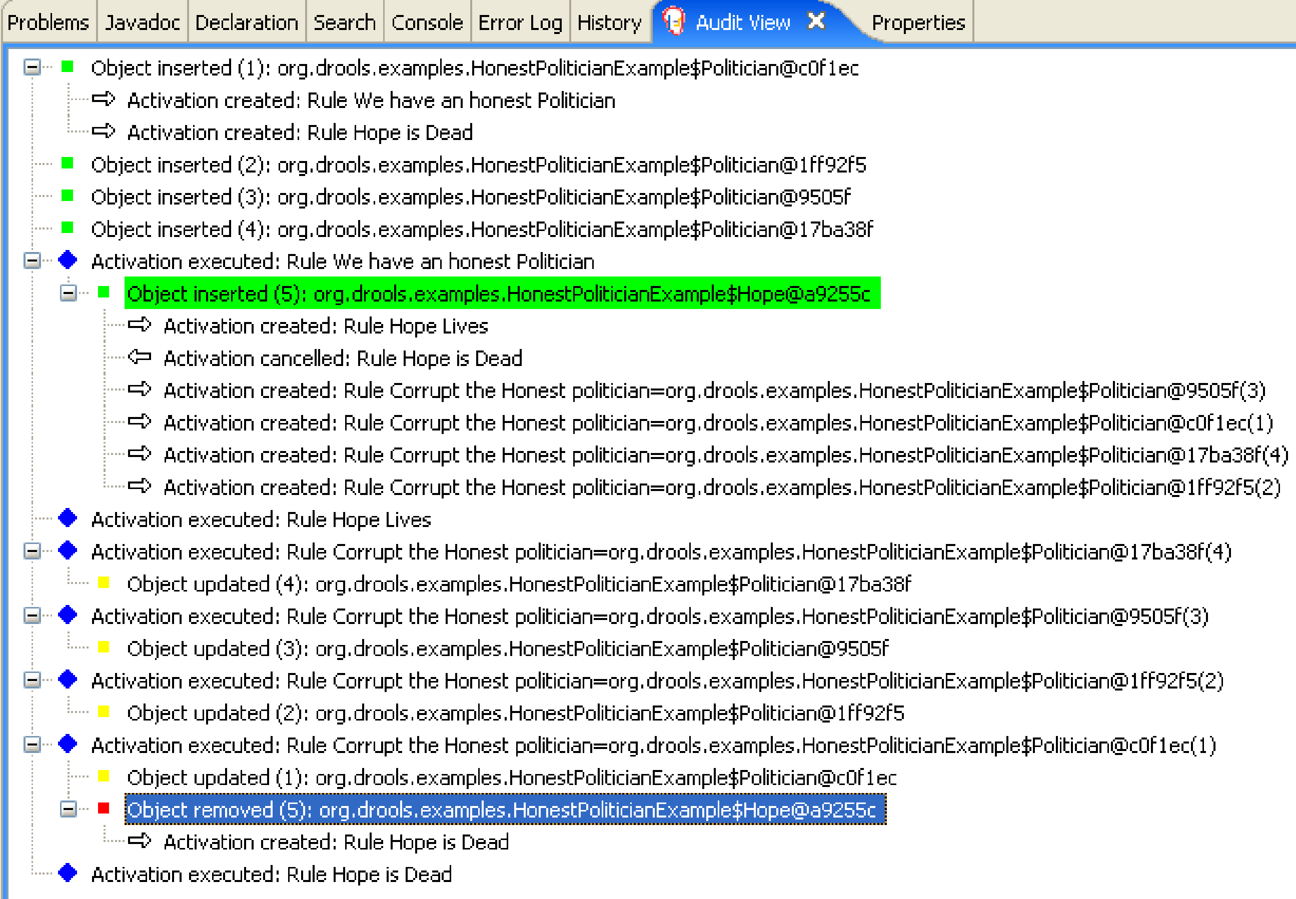
5. Creates the listeners

6. Builds the log that can be passed into the debug view or **Audit View** or your IDE after executing of the rules

When you run the Honest Politician with this modified logging capability, you can load the audit log file from **target/honestpolitician.log** into your IDE debug view or **Audit View**, if available (for example, in **Window → Show View** in some IDEs).

In this example, the **Audit View** shows the flow of executions, insertions, and retractions as defined in the example classes and rules:

Figure 7.18. Honest Politician example Audit View



When the first politician is inserted, two activations occur. The rule **"We have an honest Politician"** is activated only one time for the first inserted politician because it uses an **exists** conditional element, which matches when at least one politician is inserted. The rule **"Hope is Dead"** is also activated at this stage because the **Hope** object is not yet inserted. The rule **"We have an honest Politician"** fires first because it has a higher **salience** value than the rule **"Hope is Dead"**, and inserts the **Hope** object (highlighted in green). The insertion of the **Hope** object activates the rule **"Hope Lives"** and deactivates the rule **"Hope is Dead"**. The insertion also activates the rule **"Corrupt the Honest"** for each inserted honest politician. The rule **"Hope Lives"** is executed and prints **"Hurrah!!! Democracy Lives"**.

Next, for each politician, the rule **"Corrupt the Honest"** fires, printing **"I'm an evil corporation and I have corrupted X"**, where **X** is the name of the politician, and modifies the politician honesty value to **false**. When the last honest politician is corrupted, **Hope** is automatically retracted by the truth maintenance system (highlighted in blue). The green highlighted area shows the origin of the currently selected blue highlighted area. After the **Hope** fact is retracted, the rule **"Hope is dead"** fires, printing **"We are all Doomed!!! Democracy is Dead"**.

7.8. SUDOKU EXAMPLE DECISIONS (COMPLEX PATTERN MATCHING, CALLBACKS, AND GUI INTEGRATION)

The Sudoku example decision set, based on the popular number puzzle Sudoku, demonstrates how to use rules in Red Hat Decision Manager to find a solution in a large potential solution space based on various constraints. This example also shows how to integrate Red Hat Decision Manager rules into a graphical user interface (GUI), in this case a Swing-based desktop application, and how to use callbacks to interact with a running decision engine to update the GUI based on changes in the working memory at run time.

The following is an overview of the Sudoku example:

- Name: **sudoku**
- Main class: **org.drools.examples.sudoku.SudokuExample** (in **src/main/java**)
- Module: **drools-examples**
- **Type**: Java application
- Rule files: **org.drools.examples.sudoku.*.drl** (in **src/main/resources**)
- **Objective**: Demonstrates complex pattern matching, problem solving, callbacks, and GUI integration

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 only one time. 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 must be unique in its particular 3x3 zone, row, and column. This Sudoku example decision set uses Red Hat Decision Manager rules to solve Sudoku puzzles from a range of difficulty levels, and to attempt to resolve flawed puzzles that contain invalid entries.

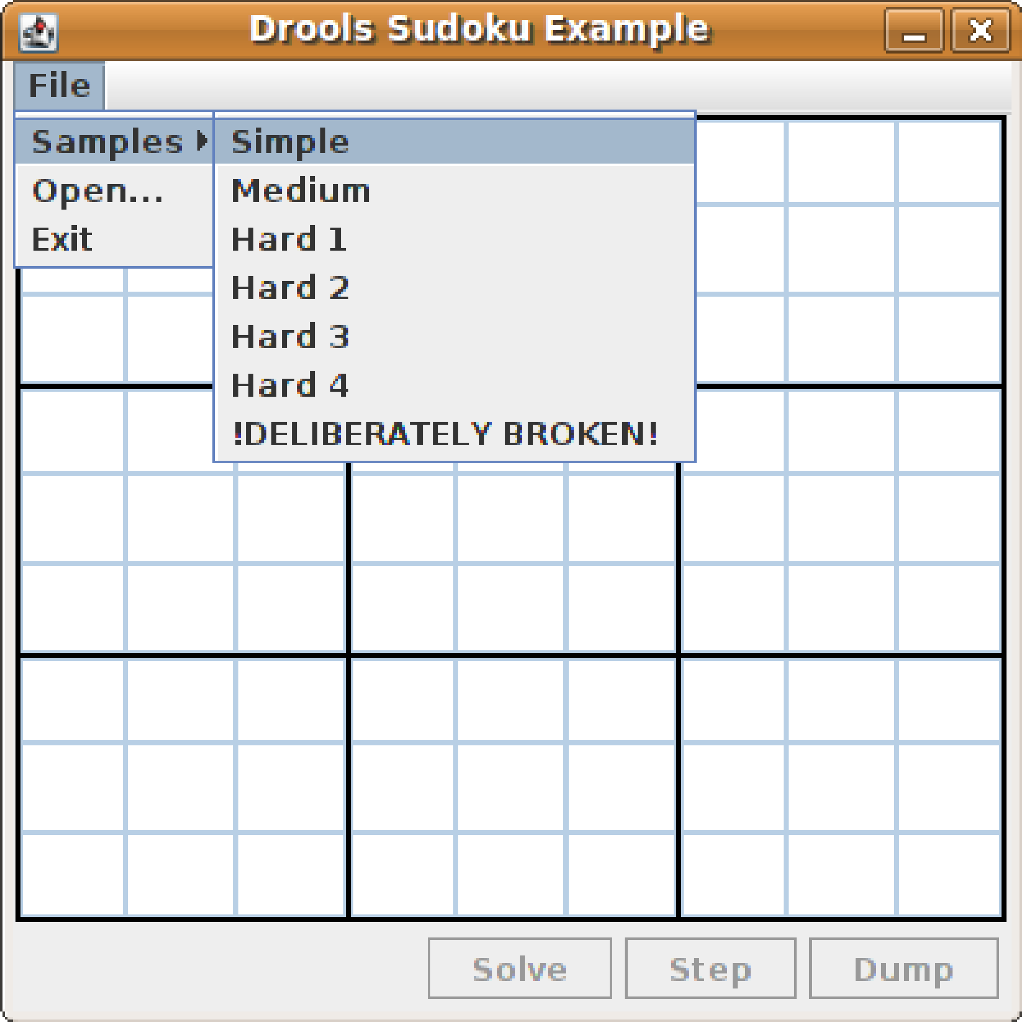
Sudoku example execution and interaction

Similar to other Red Hat Decision Manager decision examples, you execute the Sudoku example by running the **org.drools.examples.sudoku.SudokuExample** class as a Java application in your IDE.

When you execute the Sudoku example, the **Drools Sudoku Example** GUI window appears. This window contains an empty grid, but the program comes with various grids stored internally that you can load and solve.

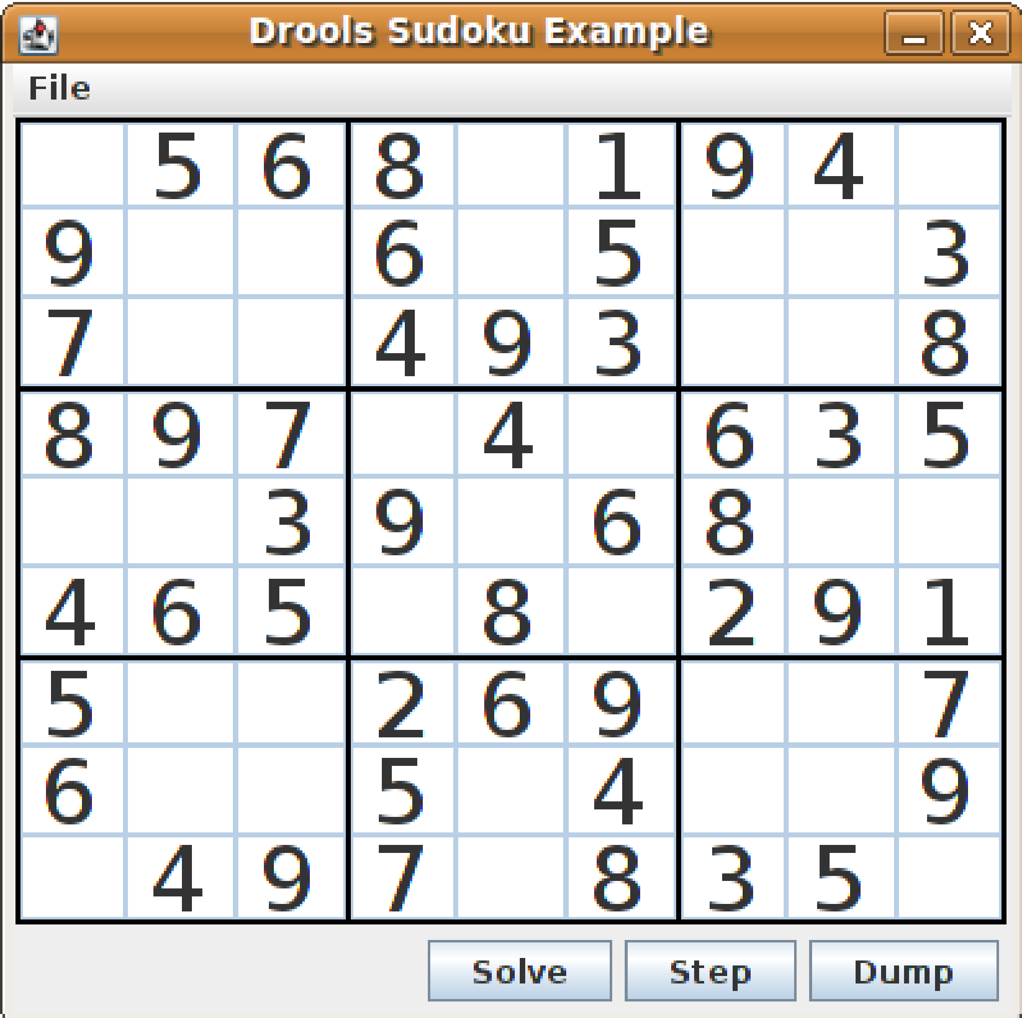
Click **File → Samples → Simple** to load one of the examples. Notice that all buttons are disabled until a grid is loaded.

Figure 7.19. Sudoku example GUI after launch



When you load the **Simple** example, the grid is filled according to the puzzle's initial state.

Figure 7.20. Sudoku example GUI after loading Simple sample



Choose from the following options:

- Click **Solve** to fire the rules defined in the Sudoku example that fill out the remaining values and that make the buttons inactive again.

Figure 7.21. Simple sample solved

- Click **Step** to see the next digit found by the rule set. The console window in your IDE displays detailed information about the rules that are executing to solve the step.

Step execution output in the IDE console

```
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 **Dump** to see the state of the grid, with cells showing either the established value or the remaining possibilities.

Dump execution output in the IDE console

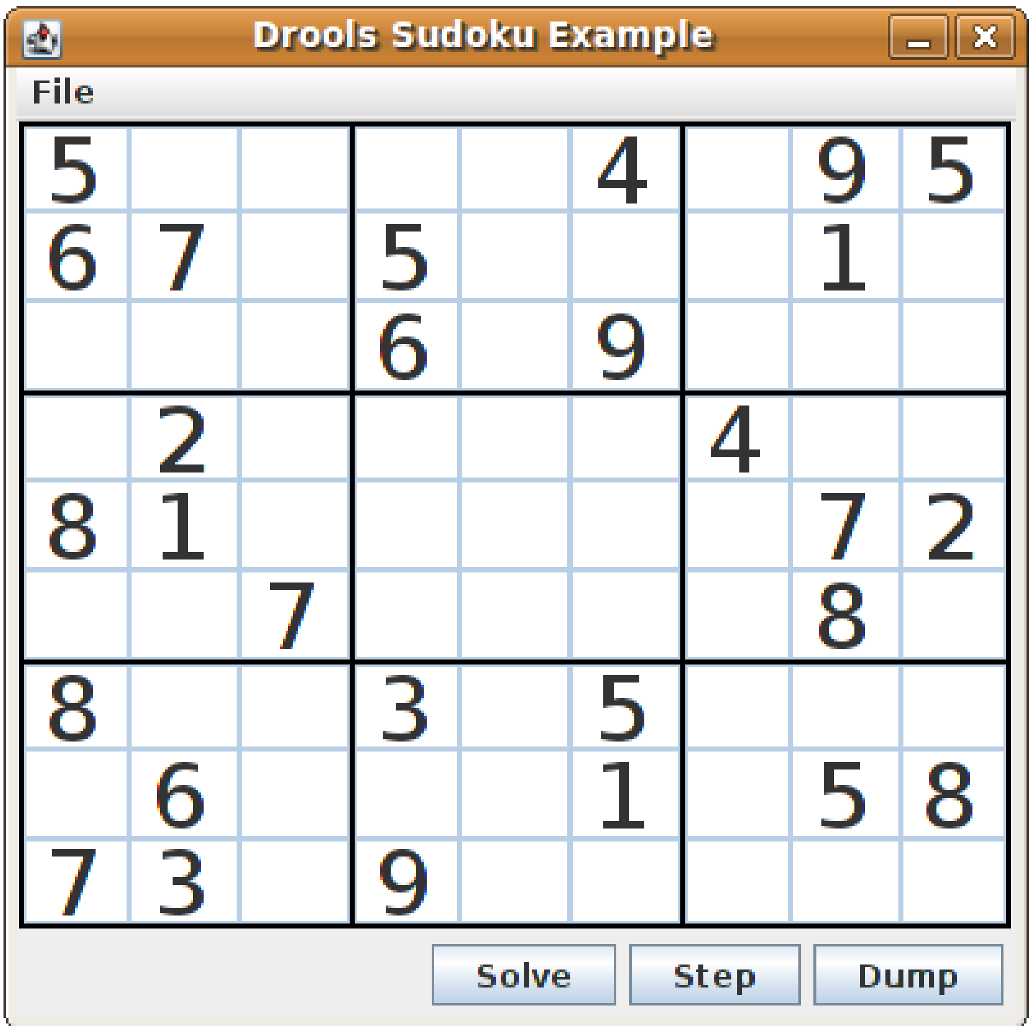
```
       Col: 0     Col: 1     Col: 2     Col: 3     Col: 4     Col: 5     Col: 6     Col: 7     Col: 8
Row 0:  123456789  --- 5 ---  --- 6 ---  --- 8 ---  123456789  --- 1 ---  --- 9 ---  --- 4 ---
123456789
Row 1:  --- 9 ---  123456789  123456789  --- 6 ---  123456789  --- 5 ---  123456789
123456789  --- 3 ---
Row 2:  --- 7 ---  123456789  123456789  --- 4 ---  --- 9 ---  --- 3 ---  123456789  123456789
--- 8 ---
```

```
Row 3:  --- 8 ---  --- 9 ---  --- 7 ---  123456789  --- 4 ---  123456789  --- 6 ---  --- 3 ---  --- 5 ---
Row 4:  123456789  123456789  --- 3 ---  --- 9 ---  123456789  --- 6 ---  --- 8 ---  123456789
123456789
Row 5:  --- 4 ---  --- 6 ---  --- 5 ---  123456789  --- 8 ---  123456789  --- 2 ---  --- 9 ---  --- 1 ---
Row 6:  --- 5 ---  123456789  123456789  --- 2 ---  --- 6 ---  --- 9 ---  123456789  123456789
--- 7 ---
Row 7:  --- 6 ---  123456789  123456789  --- 5 ---  123456789  --- 4 ---  123456789
123456789  --- 9 ---
Row 8:  123456789  --- 4 ---  --- 9 ---  --- 7 ---  123456789  --- 8 ---  --- 3 ---  --- 5 ---
123456789
```

The Sudoku example includes a deliberately broken sample file that the rules defined in the example can resolve.

Click **File → Samples → !DELIBERATELY BROKEN!** to load the broken sample. The grid starts with some issues, for example, the value **5** appears two times in the first row, which is not allowed.

Figure 7.22. Broken Sudoku example initial state



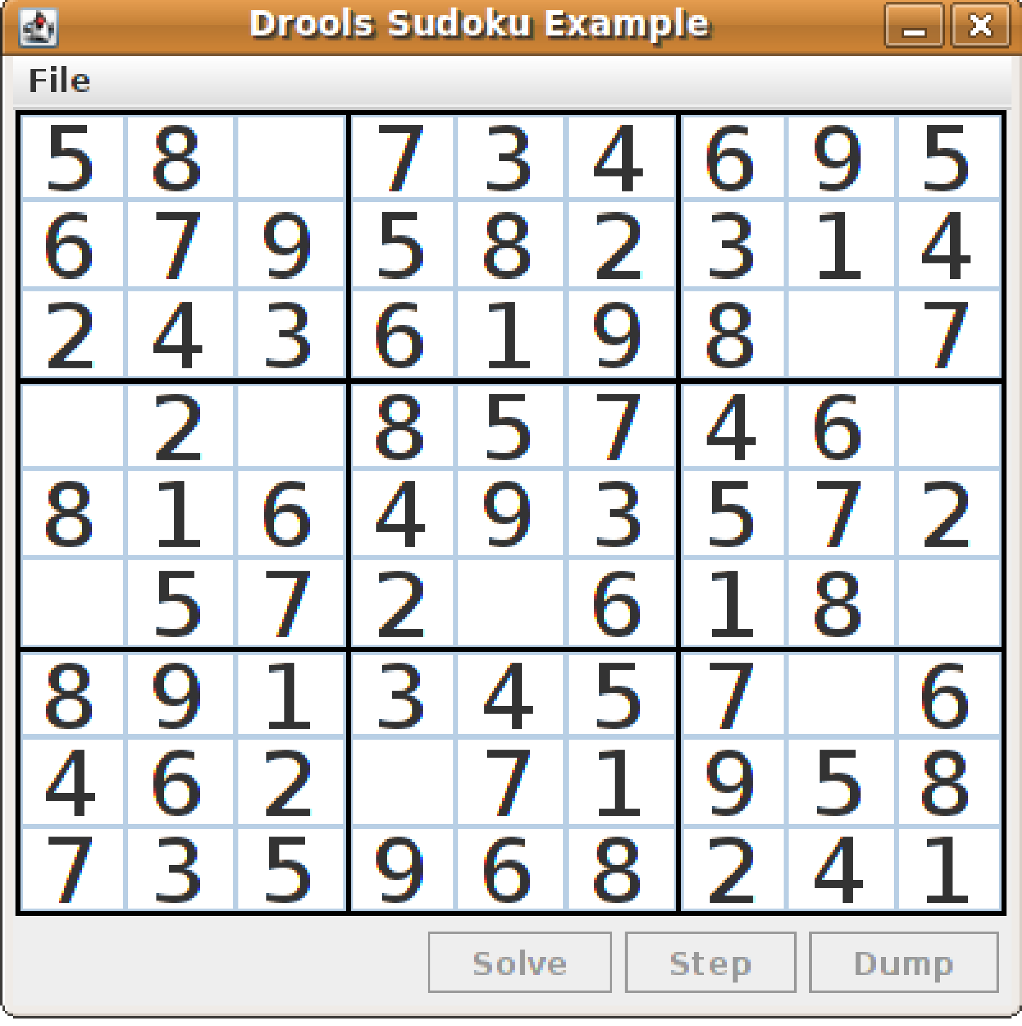
Click **Solve** to apply the solving rules to this invalid grid. The associated solving rules in the Sudoku example detect the issues in the sample and attempts to solve the puzzle as far as possible. This process does not complete and leaves some cells empty.

The solving rule activity is displayed in the IDE console window:

Detected issues in the broken sample

> 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.

Figure 7.23. Broken sample solution attempt



The sample Sudoku files labeled **Hard** are more complex and the solving rules might not be able to solve them. The unsuccessful solution attempt is displayed in the IDE console window:

Hard sample unresolved

> Validation complete.
> ...
> Sorry - can't solve this grid.

The rules that work to solve the broken sample implement standard solving techniques based on the sets of values that are still candidates for a cell. For example, if a set contains a single value, then this is the value for the cell. For a single occurrence of a value in one of the groups of nine cells, the rules 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 and the value is retracted.

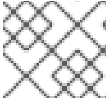
Other rules in the example reduce the permissible values for some cells. The rules **"naked pair"**, **"hidden pair in row"**, **"hidden pair in column"**, and **"hidden pair in square"** eliminate possibilities but do not establish solutions. The rules **"X-wings in rows"**, "`X-wings in columns"`, **"intersection removal**

row", and **"intersection removal column"** perform more sophisticated eliminations.

Sudoku example classes

The package **org.drools.examples.sudoku.swing** contains the following core set of classes that implement a framework for Sudoku puzzles:

- The **SudokuGridModel** class defines an interface that is implemented to store a Sudoku puzzle as a 9x9 grid of **Cell** objects.
- The **SudokuGridView** class is a Swing component that can visualize any implementation of the **SudokuGridModel** class.
- The **SudokuGridEvent** and **SudokuGridListener** classes communicate state changes between the model and the view. Events are fired when a cell value is resolved or changed.
- The **SudokuGridSamples** class provides partially filled Sudoku puzzles for demonstration purposes.



NOTE

This package does not have any dependencies on Red Hat Decision Manager libraries.

The package **org.drools.examples.sudoku** contains the following core set of classes that implement the elementary **Cell** object and its various aggregations:

- The **CellFile** class, with subtypes **CellRow**, **CellCol**, and **CellSqr**, all of which are subtypes of the **CellGroup** class.
- The **Cell** and **CellGroup** subclasses of **SetOfNine**, which provides a property **free** with the type **Set<Integer>**. For a **Cell** class, the set represents the individual candidate set. For a **CellGroup** class, the set is the union of all candidate sets of its cells (the set of digits that still need to be allocated).
  In the Sudoku example are 81 **Cell** and 27 **CellGroup** objects and a linkage provided by the **Cell** properties **cellRow**, **cellCol**, and **cellSqr**, and by the **CellGroup** property **cells** (a list of **Cell** objects). With these components, you can write rules that detect the specific situations that permit the allocation of a value to a cell or the elimination of a value from some candidate set.
- The **Setting** class is used to trigger the operations that accompany the allocation of a value. The presence of a **Setting** fact is used in all rules that detect a new situation in order to avoid reactions to inconsistent intermediary states.
- The **Stepping** class is used in a low priority rule to execute an emergency halt when a **"Step"** does not terminate regularly. This behavior indicates that the program cannot solve the puzzle.
- The main class **org.drools.examples.sudoku.SudokuExample** implements a Java application combining all of these components.

Sudoku validation rules (validate.drl)

The **validate.drl** file in the Sudoku example contains validation rules that detect duplicate numbers in cell groups. They are combined in a **"validate"** agenda group that enables the rules to be explicitly activated after a user loads the puzzle.

The **when** conditions of the three rules **"duplicate in cell …"** all function in the following ways:

- The first condition in the rule locates a cell with an allocated value.

- The second condition in the rule pulls in any of the three cell groups to which the cell belongs.
- The final condition finds a cell (other than the first one) with the same value as the first cell and in the same row, column, or square, depending on the rule.

Rules "duplicate in cell ..."

```
rule "duplicate in cell row"
  when
    $c: Cell( $v: value != null )
    $cr: CellRow( cells contains $c )
    exists Cell( this != $c, value == $v, cellRow == $cr )
  then
    System.out.println( "cell " + $c.toString() + " has a duplicate in row " + $cr.getNumber() );
end

rule "duplicate in cell col"
  when
    $c: Cell( $v: value != null )
    $cc: CellCol( cells contains $c )
    exists Cell( this != $c, value == $v, cellCol == $cc )
  then
    System.out.println( "cell " + $c.toString() + " has a duplicate in col " + $cc.getNumber() );
end

rule "duplicate in cell sqr"
  when
    $c: Cell( $v: value != null )
    $cs: CellSqr( cells contains $c )
    exists Cell( this != $c, value == $v, cellSqr == $cs )
  then
    System.out.println( "cell " + $c.toString() + " has duplicate in its square of nine." );
end
```

The rule **"terminate group"** is the last to fire. This rule prints a message and stops the sequence.

Rule "terminate group"

```
rule "terminate group"
    salience -100
  when
  then
    System.out.println( "Validation complete." );
    drools.halt();
end
```

Sudoku solving rules (sudoku.drl)

The **sudoku.drl** file in the Sudoku example contains three types of rules: 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.

The rules **"set a value"**, **"eliminate a value from Cell"**, and **"retract setting"** depend on the presence of a **Setting** object. The first rule handles the assignment to the cell and the operations for removing the value from the **free** sets of the three groups of the cell. This group also reduces a counter that, when zero, returns control to the Java application that has called **fireUntilHalt()**.

The purpose of the 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, the rule **"retract setting"** retracts the triggering **Setting** fact.

Rules "set a value", "eliminate a value from a Cell", and "retract setting"

```
// A Setting object is inserted to define the value of a Cell.
// Rule for updating the cell and all cell groups that contain it
rule "set a value"
  when
    // A Setting with row and column number, and a value
    $s: Setting( $rn: rowNo, $cn: colNo, $v: value )

    // A matching Cell, with no value set
    $c: Cell( rowNo == $rn, colNo == $cn, value == null,
              $cr: cellRow, $cc: cellCol, $cs: cellSqr )

    // Count down
    $ctr: Counter( $count: count )
  then
    // Modify the Cell by setting its value.
    modify( $c ){ setValue( $v ) }
    // System.out.println( "set cell " + $c.toString() );
    modify( $cr ){ blockValue( $v ) }
    modify( $cc ){ blockValue( $v ) }
    modify( $cs ){ blockValue( $v ) }
    modify( $ctr ){ setCount( $count - 1 ) }
end

// Rule for removing a value from all cells that are siblings
// in one of the three cell groups
rule "eliminate a value from Cell"
  when
    // A Setting with row and column number, and a value
    $s: Setting( $rn: rowNo, $cn: colNo, $v: value )

    // The matching Cell, with the value already set
    Cell( rowNo == $rn, colNo == $cn, value == $v, $exCells: exCells )

    // For all Cells that are associated with the updated cell
    $c: Cell( free contains $v ) from $exCells
  then
    // System.out.println( "clear " + $v + " from cell " + $c.posAsString()  );
    // Modify a related Cell by blocking the assigned value.
    modify( $c ){ blockValue( $v ) }
end

// Rule for eliminating the Setting fact
rule "retract setting"
  when
    // A Setting with row and column number, and a value
    $s: Setting( $rn: rowNo, $cn: colNo, $v: value )

    // The matching Cell, with the value already set
    $c: Cell( rowNo == $rn, colNo == $cn, value == $v )
```

```
    // This is the negation of the last pattern in the previous rule.
    // Now the Setting fact can be safely retracted.
    not( $x: Cell( free contains $v )
         and
         Cell( this == $c, exCells contains $x ) )
  then
    // System.out.println( "done setting cell " + $c.toString() );
    // Discard the Setter fact.
    delete( $s );
    // Sudoku.sudoku.consistencyCheck();
end
```

Two solving rules detect a situation where an allocation of a number to a cell is possible. The rule **"single"** fires for a **Cell** with a candidate set containing a single number. The rule **"hidden single"** fires when no cell exists with a single candidate, but when a cell exists containing a candidate, this candidate is absent from all other cells in one of the three groups to which the cell belongs. Both rules create and insert a **Setting** fact.

Rules "single" and "hidden single"

```
// Detect a set of candidate values with cardinality 1 for some Cell.
// This is the value to be set.
rule "single"
  when
    // Currently no setting underway
    not Setting()

    // One element in the "free" set
    $c: Cell( $rn: rowNo, $cn: colNo, freeCount == 1 )
  then
    Integer i = $c.getFreeValue();
    if (explain) System.out.println( "single " + i + " at " + $c.posAsString() );
    // Insert another Setter fact.
    insert( new Setting( $rn, $cn, i ) );
end

// Detect a set of candidate values with a value that is the only one
// in one of its groups. This is the value to be set.
rule "hidden single"
  when
    // Currently no setting underway
    not Setting()
    not Cell( freeCount == 1 )

    // Some integer
    $i: Integer()

    // The "free" set contains this number
    $c: Cell( $rn: rowNo, $cn: colNo, freeCount > 1, free contains $i )

    // A cell group contains this cell $c.
    $cg: CellGroup( cells contains $c )
    // No other cell from that group contains $i.
    not ( Cell( this != $c, free contains $i ) from $cg.getCells() )
  then
    if (explain) System.out.println( "hidden single " + $i + " at " + $c.posAsString() );
```

```
    // Insert another Setter fact.
    insert( new Setting( $rn, $cn, $i ) );
end
```

Rules from the largest group, either individually or in groups of two or three, implement various solving techniques used for solving Sudoku puzzles manually.

The 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.

Rule "naked pair"

```
// A "naked pair" is two cells in some cell group with their sets of
// permissible values being equal with cardinality 2. These two values
// can be removed from all other candidate lists in the group.
rule "naked pair"
  when
    // Currently no setting underway
    not Setting()
    not Cell( freeCount == 1 )

    // One cell with two candidates
    $c1: Cell( freeCount == 2, $f1: free, $r1: cellRow, $rn1: rowNo, $cn1: colNo, $b1: cellSqr )

    // The containing cell group
    $cg: CellGroup( freeCount > 2, cells contains $c1 )

    // Another cell with two candidates, not the one we already have
    $c2: Cell( this != $c1, free == $f1 /*** , rowNo >= $rn1, colNo >= $cn1 ***/ ) from $cg.cells

    // Get one of the "naked pair".
    Integer( $v: intValue ) from $c1.getFree()

    // Get some other cell with a candidate equal to one from the pair.
    $c3: Cell( this != $c1 && != $c2, freeCount > 1, free contains $v ) from $cg.cells
  then
    if (explain) System.out.println( "remove " + $v + " from " + $c3.posAsString() + " due to naked pair
at " + $c1.posAsString() + " and " + $c2.posAsString() );
    // Remove the value.
    modify( $c3 ){ blockValue( $v ) }
end
```

The three rules **"hidden pair in …"** functions similarly to the rule **"naked pair"**. These rules detect a subset of two numbers in exactly two cells of a group, with neither value occurring in any of the other cells of the group. This means that all other candidates can be eliminated from the two cells harboring the hidden pair.

Rules "hidden pair in ..."

```
// If two cells within the same cell group contain candidate sets with more than
// two values, with two values being in both of them but in none of the other
// cells, then we have a "hidden pair". We can remove all other candidates from
// these two cells.
rule "hidden pair in row"
  when
```

```
    // Currently no setting underway
    not Setting()
    not Cell( freeCount == 1 )

    // Establish a pair of Integer facts.
    $i1: Integer()
    $i2: Integer( this > $i1 )

    // Look for a Cell with these two among its candidates. (The upper bound on
    // the number of candidates avoids a lot of useless work during startup.)
    $c1: Cell( $rn1: rowNo, $cn1: colNo, freeCount > 2 && < 9, free contains $i1 && contains $i2,
$cellRow: cellRow )

    // Get another one from the same row, with the same pair among its candidates.
    $c2: Cell( this != $c1, cellRow == $cellRow, freeCount > 2, free contains $i1 && contains $i2 )

    // Ascertain that no other cell in the group has one of these two values.
    not( Cell( this != $c1 && != $c2, free contains $i1 || contains $i2 ) from $cellRow.getCells() )
  then
    if( explain) System.out.println( "hidden pair in row at " + $c1.posAsString() + " and " +
$c2.posAsString() );
    // Set the candidate lists of these two Cells to the "hidden pair".
    modify( $c1 ){ blockExcept( $i1, $i2 ) }
    modify( $c2 ){ blockExcept( $i1, $i2 ) }
end

rule "hidden pair in column"
  when
    not Setting()
    not Cell( freeCount == 1 )

    $i1: Integer()
    $i2: Integer( this > $i1 )
    $c1: Cell( $rn1: rowNo, $cn1: colNo, freeCount > 2 && < 9, free contains $i1 && contains $i2,
$cellCol: cellCol )
    $c2: Cell( this != $c1, cellCol == $cellCol, freeCount > 2, free contains $i1 && contains $i2 )
    not( Cell( this != $c1 && != $c2, free contains $i1 || contains $i2 ) from $cellCol.getCells() )
  then
    if (explain) System.out.println( "hidden pair in column at " + $c1.posAsString() + " and " +
$c2.posAsString() );
    modify( $c1 ){ blockExcept( $i1, $i2 ) }
    modify( $c2 ){ blockExcept( $i1, $i2 ) }
end

rule "hidden pair in square"
  when
    not Setting()
    not Cell( freeCount == 1 )

    $i1: Integer()
    $i2: Integer( this > $i1 )
    $c1: Cell( $rn1: rowNo, $cn1: colNo, freeCount > 2 && < 9, free contains $i1 && contains $i2,
              $cellSqr: cellSqr )
    $c2: Cell( this != $c1, cellSqr == $cellSqr, freeCount > 2, free contains $i1 && contains $i2 )
    not( Cell( this != $c1 && != $c2, free contains $i1 || contains $i2 ) from $cellSqr.getCells() )
  then
```

```
    if (explain) System.out.println( "hidden pair in square " + $c1.posAsString() + " and " +
$c2.posAsString() );
    modify( $c1 ){ blockExcept( $i1, $i2 ) }
    modify( $c2 ){ blockExcept( $i1, $i2 ) }
end
```

Two rules deal with **"X-wings"** in rows and columns. When only two possible cells for a value exist 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. When you follow the pattern sequence in one of these rules, notice how the conditions that are conveniently expressed by words such as **same** or **only** result in patterns with suitable constraints or that are prefixed with **not**.

Rules "X-wings in ..."

```
rule "X-wings in rows"
  when
    not Setting()
    not Cell( freeCount == 1 )

    $i: Integer()
    $ca1: Cell( freeCount > 1, free contains $i,
                $ra: cellRow, $rano: rowNo,         $c1: cellCol,        $c1no: colNo )
    $cb1: Cell( freeCount > 1, free contains $i,
                $rb: cellRow, $rbno: rowNo > $rano,      cellCol == $c1 )
    not( Cell( this != $ca1 && != $cb1, free contains $i ) from $c1.getCells() )

    $ca2: Cell( freeCount > 1, free contains $i,
                cellRow == $ra, $c2: cellCol,       $c2no: colNo > $c1no )
    $cb2: Cell( freeCount > 1, free contains $i,
                cellRow == $rb,      cellCol == $c2 )
    not( Cell( this != $ca2 && != $cb2, free contains $i ) from $c2.getCells() )

    $cx: Cell( rowNo == $rano || == $rbno, colNo != $c1no && != $c2no,
               freeCount > 1, free contains $i )
  then
    if (explain) {
        System.out.println( "X-wing with " + $i + " in rows " +
            $ca1.posAsString() + " - " + $cb1.posAsString() +
            $ca2.posAsString() + " - " + $cb2.posAsString() + ", remove from " + $cx.posAsString() );
    }
    modify( $cx ){ blockValue( $i ) }
end

rule "X-wings in columns"
  when
    not Setting()
    not Cell( freeCount == 1 )

    $i: Integer()
    $ca1: Cell( freeCount > 1, free contains $i,
                $c1: cellCol, $c1no: colNo,         $ra: cellRow,        $rano: rowNo )
    $ca2: Cell( freeCount > 1, free contains $i,
                $c2: cellCol, $c2no: colNo > $c1no,      cellRow == $ra )
    not( Cell( this != $ca1 && != $ca2, free contains $i ) from $ra.getCells() )

    $cb1: Cell( freeCount > 1, free contains $i,
```

```
            cellCol == $c1, $rb: cellRow,  $rbno: rowNo > $rano )
    $cb2: Cell( freeCount > 1, free contains $i,
            cellCol == $c2,      cellRow == $rb )
    not( Cell( this != $cb1 && != $cb2, free contains $i ) from $rb.getCells() )

    $cx: Cell( colNo == $c1no || == $c2no, rowNo != $rano && != $rbno,
           freeCount > 1, free contains $i )
  then
    if (explain) {
        System.out.println( "X-wing with " + $i + " in columns " +
            $ca1.posAsString() + " - " + $ca2.posAsString() +
            $cb1.posAsString() + " - " + $cb2.posAsString() + ", remove from " + $cx.posAsString()  );
    }
    modify( $cx ){ blockValue( $i ) }
end
```

The two rules **"intersection removal ..."** are 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 and 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 of the square and within the same cell file.

Rules "intersection removal ..."

```
rule "intersection removal column"
  when
    not Setting()
    not Cell( freeCount == 1 )

    $i: Integer()
    // Occurs in a Cell
    $c: Cell( free contains $i, $cs: cellSqr, $cc: cellCol )
    // Does not occur in another cell of the same square and a different column
    not Cell( this != $c, free contains $i, cellSqr == $cs, cellCol != $cc )

    // A cell exists in the same column and another square containing this value.
    $cx: Cell( freeCount > 1, free contains $i, cellCol == $cc, cellSqr != $cs )
  then
    // Remove the value from that other cell.
    if (explain) {
        System.out.println( "column elimination due to " + $c.posAsString() +
                            ": remove " + $i + " from " + $cx.posAsString() );
    }
    modify( $cx ){ blockValue( $i ) }
end

rule "intersection removal row"
  when
    not Setting()
    not Cell( freeCount == 1 )

    $i: Integer()
    // Occurs in a Cell
    $c: Cell( free contains $i, $cs: cellSqr, $cr: cellRow )
    // Does not occur in another cell of the same square and a different row.
    not Cell( this != $c, free contains $i, cellSqr == $cs, cellRow != $cr )
```

```
    // A cell exists in the same row and another square containing this value.
    $cx: Cell( freeCount > 1, free contains $i, cellRow == $cr, cellSqr != $cs )
  then
    // Remove the value from that other cell.
    if (explain) {
        System.out.println( "row elimination due to " + $c.posAsString() +
                            ": remove " + $i + " from " + $cx.posAsString() );
    }
    modify( $cx ){ blockValue( $i ) }
end
```

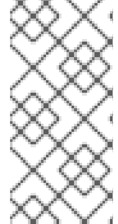
These rules are sufficient for many but not all Sudoku puzzles. To solve very difficult grids, the rule set requires more complex rules. (Ultimately, some puzzles can be solved only by trial and error.)

7.9. CONWAY'S GAME OF LIFE EXAMPLE DECISIONS (RULEFLOW GROUPS AND GUI INTEGRATION)

The Conway's Game of Life example decision set, based on the famous cellular automaton by John Conway, demonstrates how to use ruleflow groups in rules to control rule execution. The example also demonstrates how to integrate Red Hat Decision Manager rules with a graphical user interface (GUI), in this case a Swing-based implementation of Conway's Game of Life.

The following is an overview of the Conway's Game of Life (Conway) example:

- **Name**: **conway**
- **Main classes**: **org.drools.examples.conway.ConwayRuleFlowGroupRun**, **org.drools.examples.conway.ConwayAgendaGroupRun** (in **src/main/java**)
- **Module**: **droolsjbpm-integration-examples**
- **Type**: Java application
- **Rule files**: **org.drools.examples.conway.*.drl** (in **src/main/resources**)
- **Objective**: Demonstrates ruleflow groups and GUI integration



NOTE

The Conway's Game of Life example is separate from most of the other example decision sets in Red Hat Decision Manager and is located in **~/rhdm-7.2.0-sources/src/droolsjbpm-integration-$VERSION/droolsjbpm-integration-examples** of the **Red Hat Decision Manager 7.2.0 Source Distribution**from the Red Hat Customer Portal.

In Conway's Game of Life, a user interacts with the game by creating an initial configuration or an advanced pattern with defined properties and then observing how the initial state evolves. The objective of the game 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.

The following basic rules govern what the next generation looks like:

- If a live cell has fewer than two live neighbors, it dies of loneliness.

- If a live cell has more than three live neighbors, it dies from overcrowding.
- If a dead cell has exactly three live neighbors, it comes to life.

Any cell that does not meet any of those criteria is left as is for the next generation.

The Conway's Game of Life example uses Red Hat Decision Manager rules with **ruleflow-group** attributes to define the pattern implemented in the game. The example also contains a version of the decision set that achieves the same behavior using agenda groups. Agenda groups enable you to partition the engine agenda to provide execution control over groups of rules. By default, all rules are in the agenda group **MAIN**. You can use the **agenda-group** attribute to specify a different agenda group for the rule.

This overview does not explore the version of the Conway example using agenda groups. For more information about agenda groups, see the Red Hat Decision Manager example decision sets that specifically address agenda groups.

Conway example execution and interaction

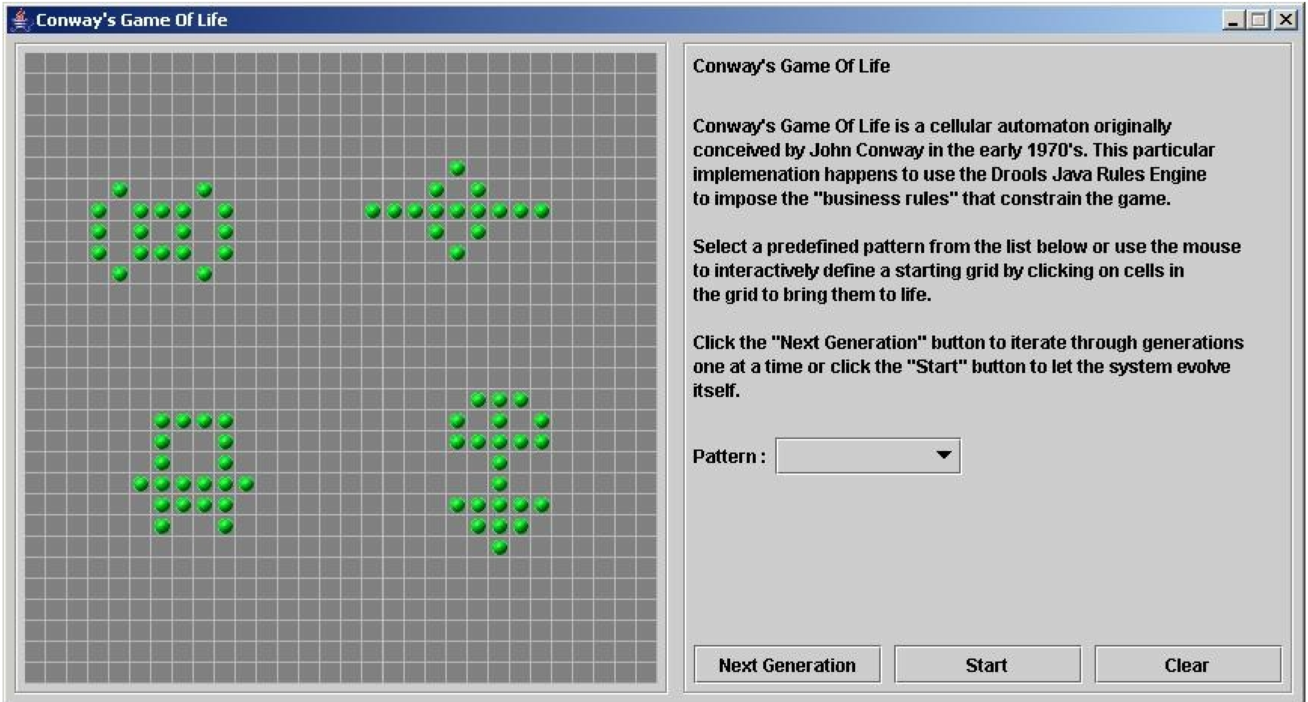
Similar to other Red Hat Decision Manager decision examples, you execute the Conway ruleflow example by running the **org.drools.examples.conway.ConwayRuleFlowGroupRun** class as a Java application in your IDE.

When you execute the Conway example, the **Conway's Game of Life** GUI window appears. This window contains an empty grid, or "arena" where the life simulation takes place. Initially the grid is empty because no live cells are in the system yet.

Figure 7.24. Conway example GUI after launch

Select a predefined pattern from the **Pattern** drop-down menu and click **Next Generation** to click through each population generation. Each cell is either alive or dead, where live cells contain a green ball. As the population evolves from the initial pattern, cells live or die relative to neighboring cells, according to the rules of the game.

Figure 7.25. Generation evolution in Conway example



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 eight neighbors. Exceptions are the corner cells, which have only three neighbors, and the cells along the four borders, with five neighbors each.

You can manually intervene to create or kill cells by clicking the cell.

To run through an evolution automatically from the initial pattern, click **Start**.

Conway example rules with ruleflow groups

The rules in the **ConwayRuleFlowGroupRun** example use ruleflow groups to control rule execution. A ruleflow group is a group of rules associated by the **ruleflow-group** rule attribute. These rules can only fire when the group is activated. The group itself can only become active when the elaboration of the ruleflow diagram reaches the node representing the group.

The Conway example uses the following ruleflow groups for rules:

- **"register neighbor"**
- **"evaluate"**
- **"calculate"**
- **"reset calculate"**
- **"birth"**
- **"kill"**
- **"kill all"**

All of the **Cell** objects are inserted into the KIE session and the **"register ..."** rules in the ruleflow group **"register neighbor"** are allowed to execute by the ruleflow process. This group of four rules creates **Neighbor** relations between some cell and its northeastern, northern, northwestern, and western neighbors.

This relation is bidirectional and handles the other four directions. Border cells do not require any special treatment. These cells are not paired with neighboring cells where there is not any.

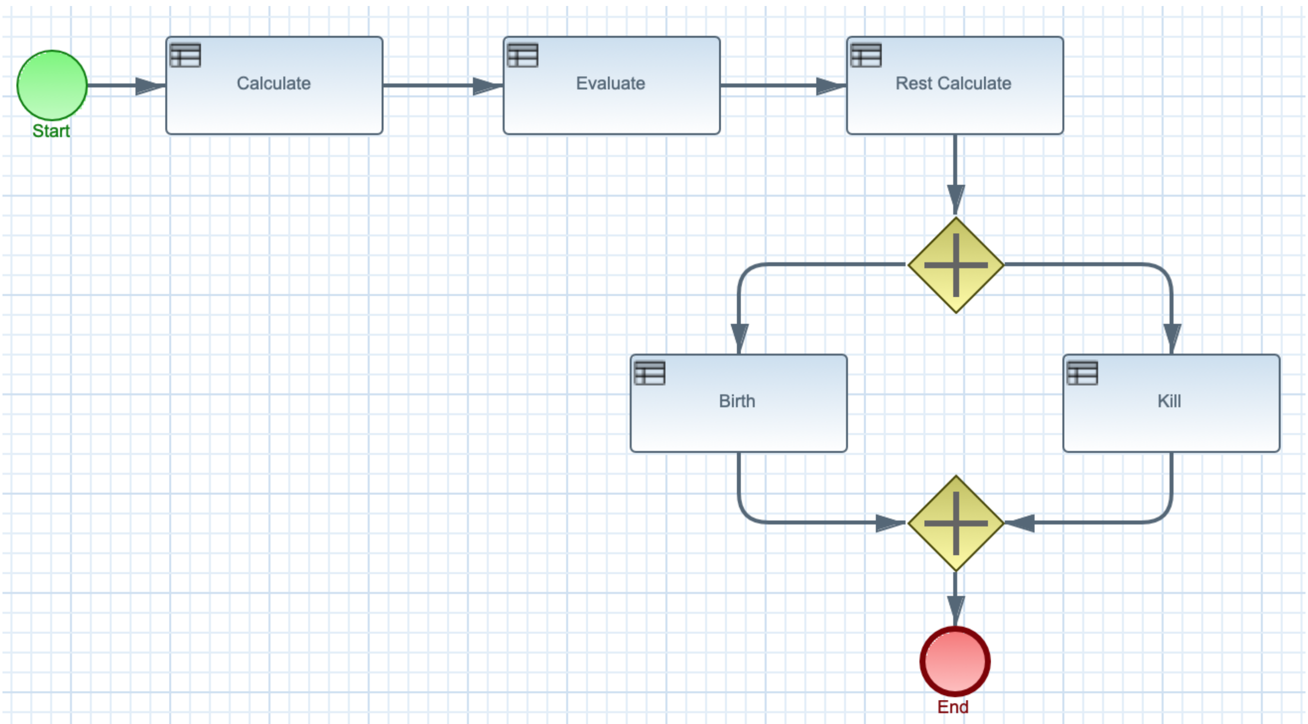
By the time all activations have fired for these rules, all cells are related to all their neighboring cells.

Rules "register ..."

```
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
```

After 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**, the example executes the **Generation** ruleflow. This ruleflow manages all changes of cells in each generation cycle.

Figure 7.26. Generation ruleflow



The ruleflow process enters the **"evaluate"** ruleflow group and any active rules in the group can fire. The rules **"Kill the ..."** and **"Give Birth"** in this group apply the game rules to birth or kill cells. The example uses the **phase** attribute to drive the reasoning of the **Cell** object by specific groups of rules. Typically, the phase is tied to a ruleflow group in the ruleflow process definition.

Notice that the example does not change the state of any **Cell** objects at this point because it must complete the full evaluation before those changes can be applied. The example sets the cell to a **phase** that is either **Phase.KILL** or **Phase.BIRTH**, which is used later to control actions applied to the **Cell** object.

Rules "Kill the ..." and "Give Birth"

```
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
```

After all **Cell** objects in the grid have been evaluated, the example uses the **"reset calculate"** rule to clear any activations in the **"calculate"** ruleflow group. The example then enters a split in the ruleflow that enables the rules **"kill"** and **"birth"** to fire, if the ruleflow group is activated. These rules apply the state change.

Rules "reset calculate", "kill", and "birth"

```
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, several **Cell** objects have been modified with the state changed to either **LIVE** or **DEAD**. When a cell becomes live or dead, the example uses the **Neighbor** relation in the rules **"Calculate ..."** to iterate over all surrounding cells, increasing or decreasing the **liveNeighbor** 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 ruleflow process.

  After the live count has been determined and set for all cells, the ruleflow process ends. If the user initially clicked **Start**, the engine restarts the ruleflow at that point. If the user initially clicked **Next Generation**, the user can request another generation.

Rules "Calculate ..."

```
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
```

7.10. HOUSE OF DOOM EXAMPLE DECISIONS (BACKWARD CHAINING AND RECURSION)

The House of Doom example decision set demonstrates how the decision engine uses backward chaining and recursion to reach defined goals or subgoals in a hierarchical system.

The following is an overview of the House of Doom example:

- Name: **backwardchaining**
- Main class: **org.drools.examples.backwardchaining.HouseOfDoomMain** (in **src/main/java**)
- Module: **drools-examples**
- **Type**: Java application

- Rule file: **org.drools.examples.backwardchaining.BC-Example.drl** (in **src/main/resources**)
- **Objective**: Demonstrates backward chaining and recursion

A backward-chaining rule system is a goal-driven system that starts with a conclusion that the decision engine attempts to satisfy, often using recursion. If the system cannot reach the conclusion or goal, it searches for subgoals, which are conclusions that complete part of the current goal. The system continues this process until either the initial conclusion is satisfied or all subgoals are satisfied.

In contrast, a forward-chaining rule system is a data-driven system that starts with a fact in the working memory of the decision engine and reacts to changes to that fact. When objects are inserted into working memory, any rule conditions that become true as a result of the change are scheduled for execution by the agenda.

The decision engine in Red Hat Decision Manager uses both forward and backward chaining to evaluate rules.

The following diagram illustrates how the decision engine evaluates rules using forward chaining overall with a backward-chaining segment in the logic flow:

Figure 7.27. Rule evaluation logic using forward and backward chaining

EVALUATE RULES
Rules
Facts
Done
NO
Does a new rule match exist?
YES
NO
Does any pattern match some fact?
YES
NO
Does the rule contain a goal?
YES
BACKWARD CHAINING
Is the goal a fact?
NO
Does a matching rule exist for the goal?
YES
YES
NO
Goal has been reached. Return TRUE and any matching fact.
Mark the matching rule condition as a new fact.
Goal cannot be reached. Return FALSE.
TRUE
FALSE
Execute the rule action.
Mark rule match as evaluated.
RHDM_2_0319

The House of Doom example uses rules with various types of queries to find the location of rooms and items within the house. The sample class **Location.java** contains the **item** and **location** elements used in the example. The sample class **HouseOfDoomMain.java** inserts the items or rooms in their respective locations in the house and executes the rules.

Items and locations in HouseOfDoomMain.java class

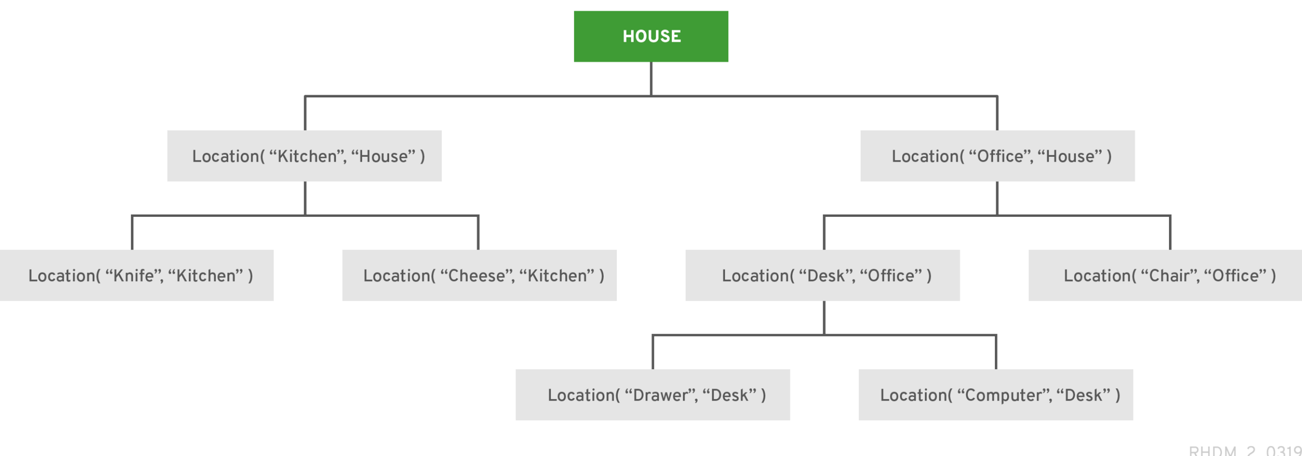
```
ksession.insert( new Location("Office", "House") );
ksession.insert( new Location("Kitchen", "House") );
```

```
ksession.insert( new Location("Knife", "Kitchen") );
ksession.insert( new Location("Cheese", "Kitchen") );
ksession.insert( new Location("Desk", "Office") );
ksession.insert( new Location("Chair", "Office") );
ksession.insert( new Location("Computer", "Desk") );
ksession.insert( new Location("Drawer", "Desk") );
```

The example rules rely on backward chaining and recursion to determine the location of all items and rooms in the house structure.

The following diagram illustrates the structure of the House of Doom and the items and rooms within it:

Figure 7.28. House of Doom structure



To execute the example, run the **org.drools.examples.backwardchaining.HouseOfDoomMain** class as a Java application in your IDE.

After the execution, the following output appears in the IDE console window:

Execution output in the IDE console

```
go1
Office is in the House
---
go2
Drawer is in the House
---
go3
---
Key is in the Office
---
go4
Chair is in the Office
Desk is in the Office
Key is in the Office
Computer is in the Office
Drawer is in the Office
---
go5
Chair is in Office
Desk is in Office
```

```
Drawer is in Desk
Key is in Drawer
Kitchen is in House
Cheese is in Kitchen
Knife is in Kitchen
Computer is in Desk
Office is in House
Key is in Office
Drawer is in House
Computer is in House
Key is in House
Desk is in House
Chair is in House
Knife is in House
Cheese is in House
Computer is in Office
Drawer is in Office
Key is in Desk
```

All rules in the example have fired to detect the location of all items in the house and to print the location of each in the output.

Recursive query and related rules

A recursive query repeatedly searches through the hierarchy of a data structure for relationships between elements.

In the House of Doom example, the **BC-Example.drl** file contains an **isContainedIn** query that most of the rules in the example use to recursively evaluate the house data structure for data inserted into the decision engine:

Recursive query in BC-Example.drl

```
query isContainedIn( String x, String y )
  Location( x, y; )
  or
  ( Location( z, y; ) and isContainedIn( x, z; ) )
end
```

The rule **"go"** prints every string inserted into the system to determine how items are implemented, and the rule **"go1"** calls the query **isContainedIn**:

Rules "go" and "go1"

```
rule "go" salience 10
  when
    $s : String( )
  then
    System.out.println( $s );
end

rule "go1"
  when
    String( this == "go1" )
    isContainedIn("Office", "House"; )
```

```
  then
    System.out.println( "Office is in the House" );
end
```

The example inserts the **"go1"** string into the engine and activates the **"go1"** rule to detect that item **Office** is in the location **House**:

Insert string and fire rules

```
ksession.insert( "go1" );
ksession.fireAllRules();
```

Rule "go1" output in the IDE console

```
go1
Office is in the House
```

Transitive closure rule

Transitive closure is a relationship between an element contained in a parent element that is multiple levels higher in a hierarchical structure.

The rule **"go2"** identifies the transitive closure relationship of the **Drawer** and the **House**: The **Drawer** is in the **Desk** in the **Office** in the **House**.

```
rule "go2"
  when
    String( this == "go2" )
    isContainedIn("Drawer", "House"; )
  then
    System.out.println( "Drawer is in the House" );
end
```

The example inserts the **"go2"** string into the engine and activates the **"go2"** rule to detect that item **Drawer** is ultimately within the location **House**:

Insert string and fire rules

```
ksession.insert( "go2" );
ksession.fireAllRules();
```

Rule "go2" output in the IDE console

```
go2
Drawer is in the House
```

The engine determines this outcome based on the following logic:

1. The query recursively searches through several levels in the house to detect the transitive closure between **Drawer** and **House**.

2. Instead of using **Location(x, y;)**, the query uses the value of **(z, y;)** because **Drawer** is not directly in **House**.

3. The **z** argument is currently unbound, which means it has no value and returns everything that is in the argument.
4. The **y** argument is currently bound to **House**, so **z** returns **Office** and **Kitchen**.
5. The query gathers information from the **Office** and checks recursively if the **Drawer** is in the **Office**. The query line **isContainedIn(x, z;)** is called for these parameters.
6. No instance of **Drawer** exists directly in **Office**, so no match is found.
7. With **z** unbound, the query returns data within the **Office** and determines that **z == Desk**.

```
isContainedIn(x==drawer, z==desk)
```

8. The **isContainedIn** query recursively searches three times, and on the third time, the query detects an instance of **Drawer** in **Desk**.

```
Location(x==drawer, y==desk)
```

9. After this match on the first location, the query recursively searches back up the structure to determine that the **Drawer** is in the **Desk**, the **Desk** is in the **Office**, and the **Office** is in the **House**. Therefore, the **Drawer** is in the **House** and the rule is satisfied.

Reactive query rule

A reactive query searches through the hierarchy of a data structure for relationships between elements and is dynamically updated when elements in the structure are modified.

The rule **"go3"** functions as a reactive query that detects if a new item **Key** ever becomes present in the **Office** by transitive closure: A **Key** in the **Drawer** in the **Office**.

Rule "go3"

```
rule "go3"
  when
    String( this == "go3" )
    isContainedIn("Key", "Office"; )
  then
    System.out.println( "Key is in the Office" );
end
```

The example inserts the **"go3"** string into the engine and activates the **"go3"** rule. Initially, this rule is not satisfied because no item **Key** exists in the house structure, so the rule produces no output.

Insert string and fire rules

```
ksession.insert( "go3" );
ksession.fireAllRules();
```

Rule "go3" output in the IDE console (unsatisfied)

```
go3
```

The example then inserts a new item **Key** in the location **Drawer**, which is in **Office**. This change satisfies the transitive closure in the **"go3"** rule and the output is populated accordingly.

Insert new item location and fire rules

```
ksession.insert( new Location("Key", "Drawer") );
ksession.fireAllRules();
```

Rule "go3" output in the IDE console (satisfied)

```
Key is in the Office
```

This change also adds another level in the structure that the query includes in subsequent recursive searches.

Queries with unbound arguments in rules

A query with one or more unbound arguments returns all undefined (unbound) items within a defined (bound) argument of the query. If all arguments in a query are unbound, then the query returns all items within the scope of the query.

The rule **"go4"** uses an unbound argument **thing** to search for all items within the bound argument **Office**, instead of using a bound argument to search for a specific item in the **Office**:

Rule "go4"

```
rule "go4"
  when
    String( this == "go4" )
    isContainedIn(thing, "Office"; )
  then
    System.out.println( thing + "is in the Office" );
end
```

The example inserts the **"go4"** string into the engine and activates the **"go4"** rule to return all items in the **Office**:

Insert string and fire rules

```
ksession.insert( "go4" );
ksession.fireAllRules();
```

Rule "go4" output in the IDE console

```
go4
Chair is in the Office
Desk is in the Office
Key is in the Office
Computer is in the Office
Drawer is in the Office
```

The rule **"go5"** uses both unbound arguments **thing** and **location** to search for all items and their locations in the entire **House** data structure:

Rule "go5"

```
rule "go5"
```

```
  when
    String( this == "go5" )
    isContainedIn(thing, location; )
  then
    System.out.println(thing + " is in " + location );
end
```

The example inserts the **"go5"** string into the engine and activates the **"go5"** rule to return all items and their locations in the **House** data structure:

Insert string and fire rules

```
ksession.insert( "go5" );
ksession.fireAllRules();
```

Rule "go5" output in the IDE console

```
go5
Chair is in Office
Desk is in Office
Drawer is in Desk
Key is in Drawer
Kitchen is in House
Cheese is in Kitchen
Knife is in Kitchen
Computer is in Desk
Office is in House
Key is in Office
Drawer is in House
Computer is in House
Key is in House
Desk is in House
Chair is in House
Knife is in House
Cheese is in House
Computer is in Office
Drawer is in Office
Key is in Desk
```

APPENDIX A. VERSIONING INFORMATION

Documentation last updated on Friday, May 22, 2020.