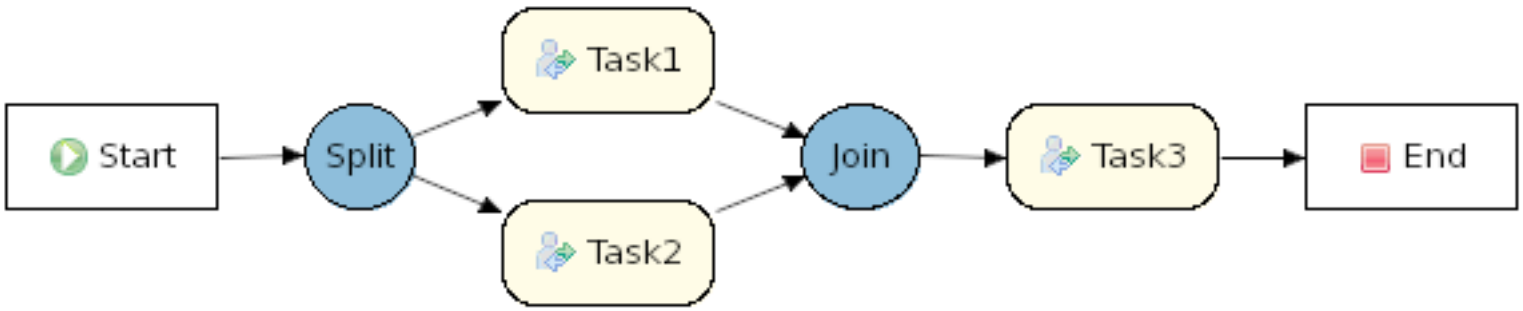
Chapter 1. Introduction

Drools Flow is a workflow or process engine that allows advanced integration of processes and rules. A process or a workflow describes the order in which a series of steps need to be executed, using a flow chart. For example, the following figure shows a process where first Task1 and Task2 need to be executed in parallel. After completion of both, Task3 needs to be executed.



The following chapters will learn you everything you need to know about Drools Flow. It's distinguishing characteristics are:

1. **Advanced integration of processes and rules:** Processes and rules are usually considered as two different paradigms when it comes to defining business logic. While loose coupling between a processes and rules is possible by integrating both a process and a rules engine, we provide advanced integration of processes and rules out-of-the-box. This allows users to use rules to define part of their business logic when defining their business processes and vice versa.

2. **Unification of processes and rules:** We consider rules, processes and event processing all as different types of knowledge. Not only do we allow the advanced integration of these three types, we also offer a unified API and unified tooling so that users should not learn three different products but can easily combine these three types using our knowledge-based API. The tooling also allows seamless integration of these different kinds of knowledge, including things like a unified knowledge repository, audit logs, debugging, etc.

3. **Declarative modelling:** Drools Flow tries to keep processes as declarative as possible, i.e. focussing on what should happen instead of how. As a result, we try to avoid having to hardcode details into your process but offer ways to describe your work in an abstract way (e.g. using pluggable work items, a business scripting language, etc.). We also allows users to easily create domain-specific extension, making it much easier to read, update or create these processes as they are using domain-specific concepts that are closely related to the problem you are trying to solve and can be understood by domain experts.

4. **Generic process engine supporting multiple process languages:** We do not believe that there is one process language that fits all purposes. Therefore, the Drools Flow engine is based on a generic process engine that allows the definition and execution of different types of

process languages, like for example our RuleFlow language, WS-BPEL (a standard targeted towards web service orchestration), OSWorkflow (another existing workflow language), jPDL (the process language defined by the jBPM project), etc. All these languages are based on the same set of core building blocks, making it easier to implement your own process language by reusing and combining these low-level building blocks the way you want to.

All these features (and many more) will be explained in the following chapters.

Chapter 2. Getting Started

This section describes how to get started with Drools Flow. It will guide you to create and execute your first Drools Flow process.

2.1. Installation

The best way to get started is to use the Drools Eclipse Plugin. This is a plugin for the Eclipse development environment that allows users to create, execute and debug Drools processes and rules. To get started with the plugin, you first need an Eclipse 3.4.x (as well as the Eclipse Graphical Editing Framework (GEF) plugin installed). Eclipse can be downloaded from the following link (if you do not know which version of eclipse you need, simply choose the "Eclipse IDE for Java Developers", and this one already includes the GEF plugin as well):

http://www.eclipse.org/downloads/

Next you need to install the Drools Eclipse plugin. Download the Drools Eclipse IDE plugin from the link below. Unzip the downloaded file in your main eclipse folder (do not just copy the file there, extract it so that the feature and plugin jars end up in the features and plugin directory of eclipse) and (re)start Eclipse.

http://www.jboss.org/drools/downloads.html

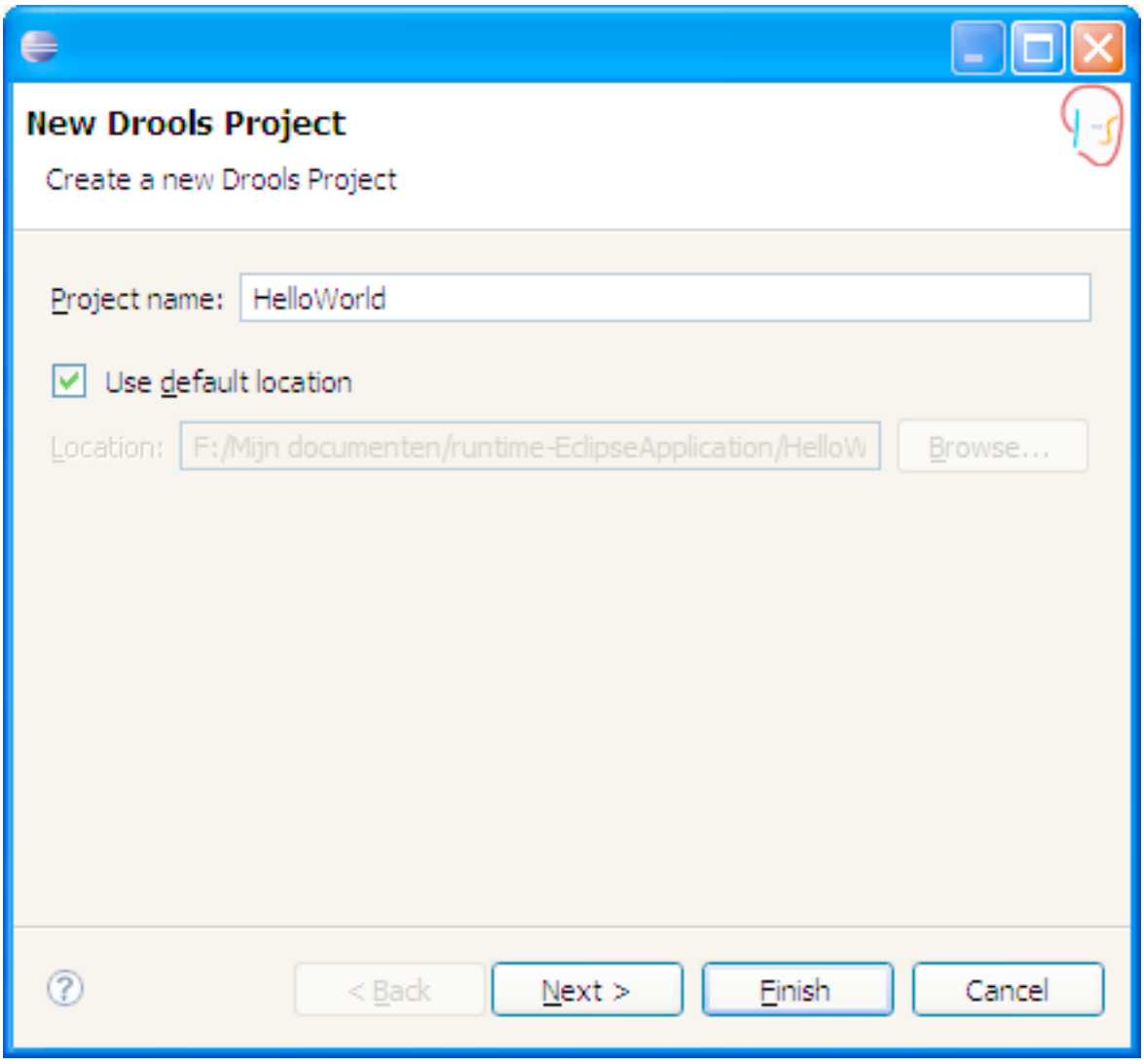
To check that the installation was successful, try opening the Drools perspective: Click the 'Open Perspective' button in the top right corner of your Eclipse window, select 'Other...' and pick the Drools perspective. If you cannot find the Drools perspective as one of the possible perspectives, the installation probably was unsuccessful. Check whether you executed each of the required steps correctly: Do you have the right version of Eclipse (3.4.x)? Do you have Eclipse GEF installed (check whether the org.eclipse.gef_3.4.*.jar exists in the plugins directory in your eclipse root folder)? Did you extract the Drools Eclipse plugin correctly (check whether the org.drools.eclipse_*.jar exists in the plugins directory in your eclipse root folder)? If you cannot find the problem, try contacting us (e.g. on irc or on the user mailing list), more info can be found no our homepage here:

http://www.jboss.org/drools/

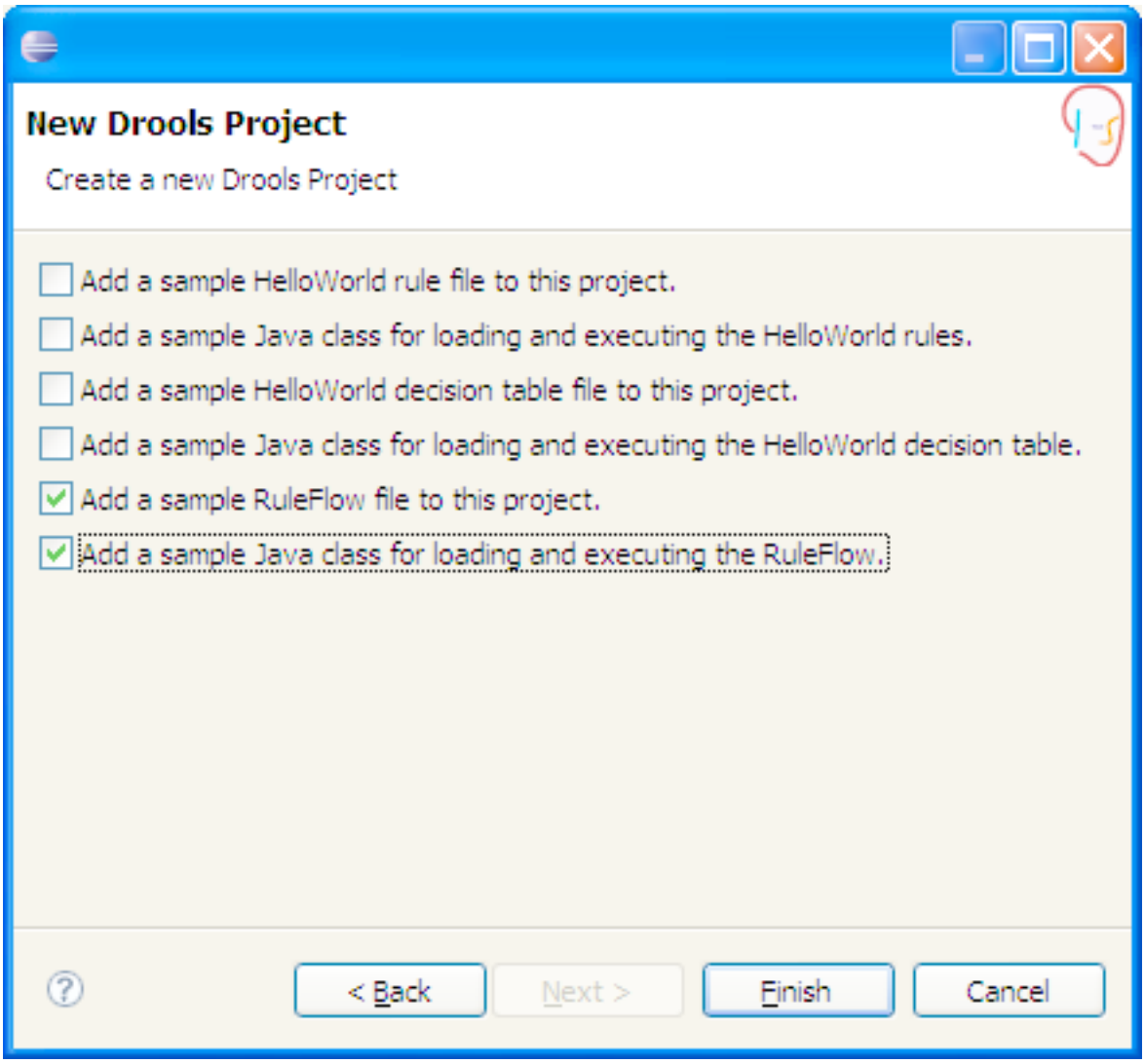
2.2. Creating your first process

A new Drools project wizard can be used to set up an executable project that contains the necessary files to get started easily with defining and executing processes. This wizard will setup a basic project structure, the classpath, a sample process and execution code to get you started. To create a new Drools Project, simply left-click on the Drools action button (with the Drools head) in the Eclipse toolbar and select "New Drools Project". [Note that the Drools action button only shows up in the Drools perspective. To open the Drools perspective (if you haven't done so already), click the 'Open Perspective' button in the top right corner of your Eclipse window, select 'Other...'

and pick the Drools perspective.] Alternatively, you could also select "File" -> "New" -> "Project ..." and in the Drools folder, select Drools Project. This should open the following dialog:

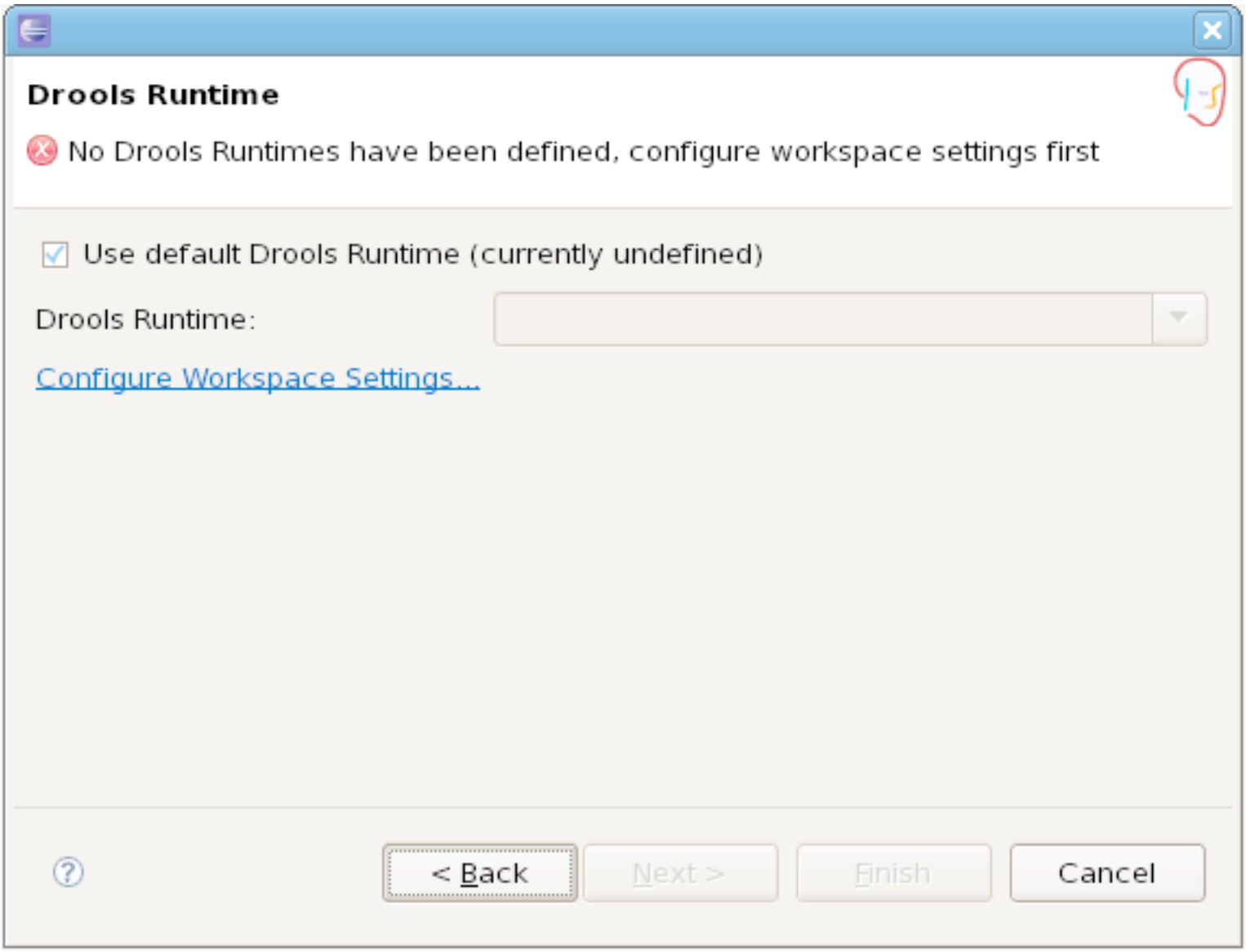


Give your project a name and click Next. In the following dialog you can select which elements are added to your project by default. Since we are creating a new process, deselect the first two check boxes and select the last two. This will generate a sample process and a Java class to execute this process.

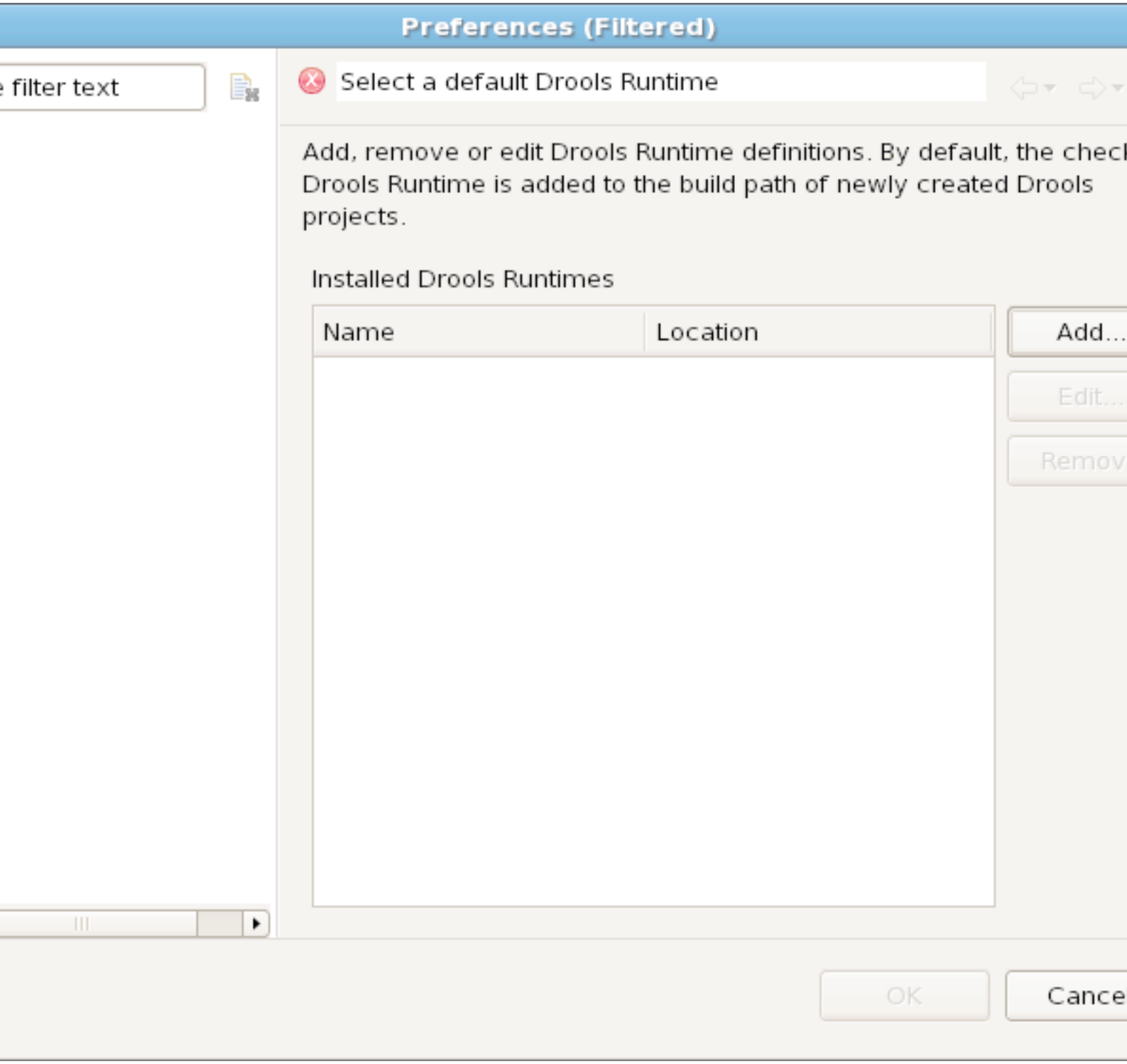


If you have not yet set up a Drools runtime, you should do this now. A Drools runtime is a collection of jars on your file system that represent one specific release of the Drools project jars. To create a runtime, you must either point the IDE to the release of your choice, or you can simply create a new runtime on your file system from the jars included in the Drools Eclipse plugin. Since we simply want to use the Drools version included in this plugin, we will do the latter. Note that you will only have to do this once, next time you create a Drools project, it will automatically use the default Drools runtime (unless you specify otherwise).

So if you have not yet set up a Drools runtime, click the Next button. The following dialog as shown below shows up, telling you that you have not yet defined a default Drools runtime and that you should configure the workspace settings first. Do this by clicking on the "Configure Workspace Settings ..." link.



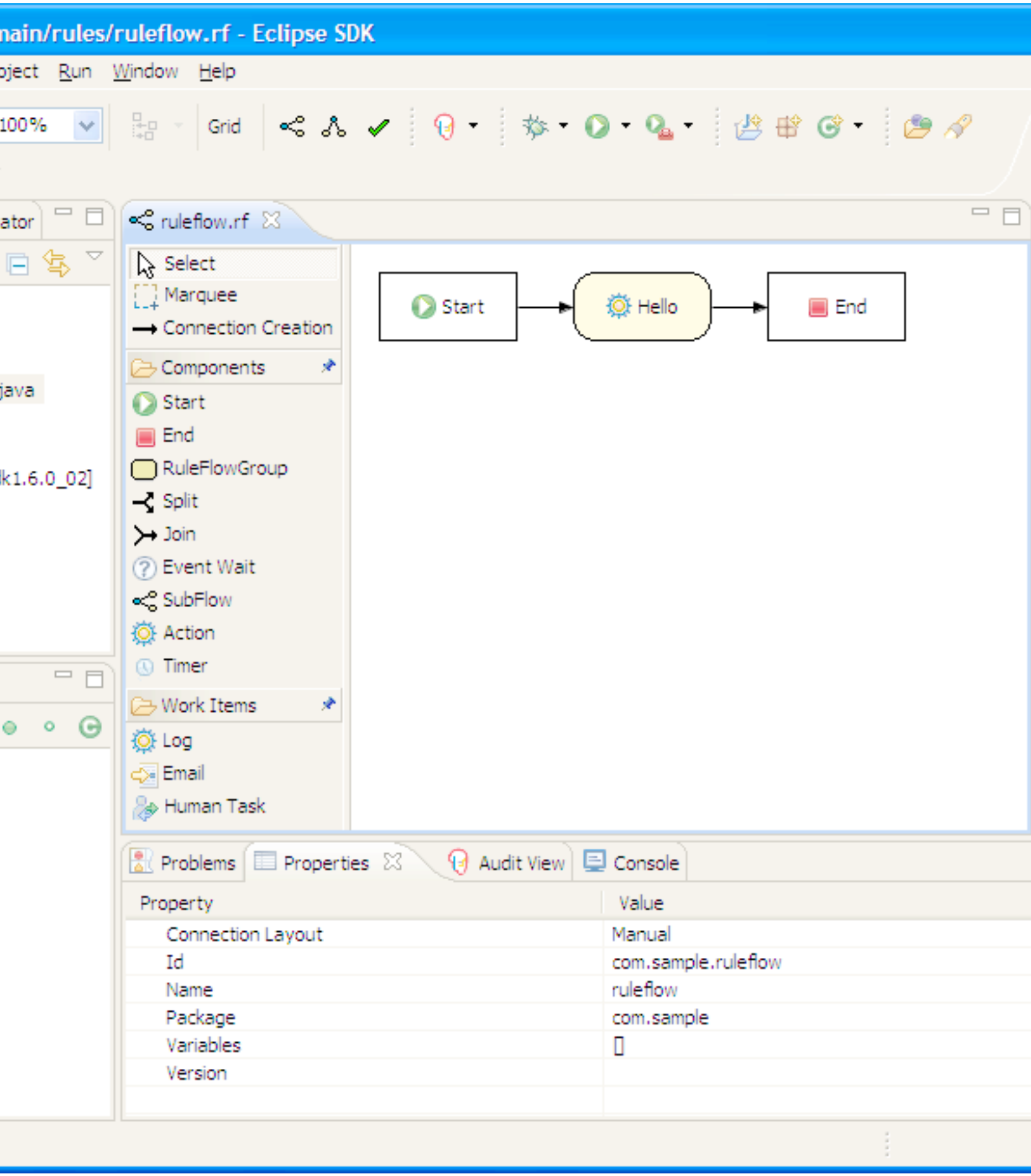
The dialog that pops up shows the workspace settings for Drools runtimes. The first time you do this, the list of installed Drools runtimes is probably empty, as shown below. To create a new runtime on your file system, click the "Add..." button. This shows a dialog where you should give the new runtime a name (e.g. "Drools 5.0.0 runtime"), and a path to your drools runtime on your file system. In this tutorial, we will simply create a new Drools 5 runtime from the jars embedded in the Drools Eclipse plugin. Click the "Create a new Drools 5 runtime ..." button and select the folder where you want this runtime to be stored and click the "OK" button. You will see the selected path show up in the previous dialog. So we're all done here as well, so click the "OK" button here as well. You will see the newly created runtime show up in your list of Drools runtimes. Select this runtime as the new default runtime by clicking on the check box in front of your runtime name and click "OK". After successfully setting up your runtime, you can now finish the project creation wizard by clicking on the "Finish" button.



The end result should look like this and contains:

1. ruleflow.rf : the process definition, which is a very simple process containing a start node (the entry point), an action node (that prints out 'Hello World') and an end node (the end of the process).
2. RuleFlowTest.java : a Java class that executes the process.

3. The necessary libraries are automatically added to the project classpath as a Drools library.


nain/rules/ruleflow.rf - Eclipse SDK
oject Run Window Help
100%
Grid
ruleflow.rf
Select
Marquee
Connection Creation
Components
Start
End
RuleFlowGroup
Split
Join
Event Wait
SubFlow
Action
Timer
Work Items
Log
Email
Human Task
Start
Hello
End
Problems
Properties
Audit View
Console
Property
Value
Connection Layout
Manual
Id
com.sample.ruleflow
Name
ruleflow
Package
com.sample
Variables
Version

By double-clicking the ruleflow.rf file, the process will be opened in the RuleFlow editor. The RuleFlow editor contains a graphical representation of your process definition. It consists of nodes that are connected to each other. The editor shows the overall control flow, while the details of each of the elements can be viewed (and edited) in the Properties View at the bottom. The editor contains a palette at the left that can be used to drag-and-drop new nodes, and an outline view at the right.

This process is a simple sequence of three nodes. The start node defines the start of the process. It is connected to an action node (called 'Hello' that simply prints out 'Hello World' to the standard output. You can see this by clicking on the Hello node and checking the action property in the properties view below. This node is then connected to an end node, signaling the end of the process.

While it is probably easier to edit processes using the graphical editor, user can also modify the underlying XML directly. The XML for our sample process is shown below (note that we did not include the graphical information here for simplicity). The process element contains parameters like the name and id of the process, and consists of three main subsections: a header (where information like variables, globals and imports can be defined), the nodes and the connections.

```
<?xml version="1.0" encoding="UTF-8"?>
<process xmlns="http://drools.org/drools-5.0/process"
         xmlns:xs="http://www.w3.org/2001/XMLSchema-instance"
         xs:schemaLocation="http://drools.org/drools-5.0/process
 drools-processes-5.0.xsd"
         type="RuleFlow" name="ruleflow" id="com.sample.ruleflow"
 package-name="com.sample" >

  <header>
  </header>

  <nodes>
    <start id="1" name="Start" x="16" y="16" />
    <actionNode id="2" name="Hello" x="128" y="16" >
      <action type="expression" dialect="mvel" >System.out.println("Hello
 World");</action>
    </actionNode>
    <end id="3" name="End" x="240" y="16" />
  </nodes>

  <connections>
    <connection from="1" to="2" />
    <connection from="2" to="3" />
  </connections>

</process>
```

2.3. Executing your first process

To execute this process, right-click on RuleFlowTest.java and select Run As - Java Application. When the process in executed, the following output should appear on the console:

```
Hello World
```

If you look at the RuleFlowTest code (see below), you will see that executing a process requires a few steps:

1. You should first create a knowledge base. A knowledge base contains all the knowledge (i.e. processes, rules, etc.) that are relevant in your application. This knowledge base can be created only once and can be reused. In this case, the knowledge base only consists of our sample process.

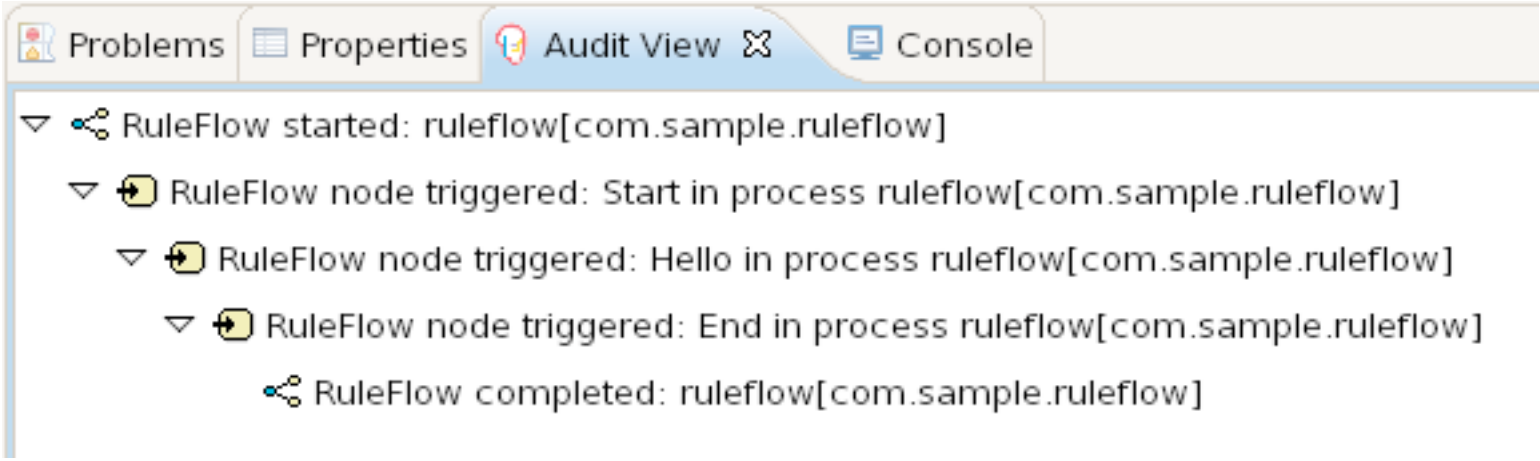
2. Next, you should create a session to interact with the engine. Note that we also add a logger to the session to log execution events and make it easier to visualize what is going on.

3. Finally, you can start a new instance of the process by invoking the startProcess("processId") method on the session. This will start the execution of your process instance. The process instance will execute the start node, action node and end node in this order, after which the process instance will be completed.

```
package com.sample;

import org.drools.KnowledgeBase;
import org.drools.KnowledgeBaseFactory;
import org.drools.builder.KnowledgeBuilder;
import org.drools.builder.KnowledgeBuilderError;
import org.drools.builder.KnowledgeBuilderErrors;
import org.drools.builder.KnowledgeBuilderFactory;
import org.drools.builder.KnowledgeType;
import org.drools.io.ResourceFactory;
import org.drools.logger.KnowledgeRuntimeLogger;
import org.drools.logger.KnowledgeRuntimeLoggerFactory;
import org.drools.runtime.StatefulKnowledgeSession;

/**
 * This is a sample file to launch a process.
 */
public class RuleFlowTest {

  public static final void main(String[] args) {
    try {
```

```
        // load up the knowledge base
        KnowledgeBase kbase = readKnowledgeBase();
        StatefulKnowledgeSession ksession =
  kbase.newStatefulKnowledgeSession();
        KnowledgeRuntimeLogger logger =
  KnowledgeRuntimeLoggerFactory.newFileLogger(ksession, "test");
        // start a new process instance
        ksession.startProcess("com.sample.ruleflow");
        logger.close();
      } catch (Throwable t) {
        t.printStackTrace();
      }
    }

    private static KnowledgeBase readKnowledgeBase() throws Exception {
      KnowledgeBuilder kbuilder =
  KnowledgeBuilderFactory.newKnowledgeBuilder();
      kbuilder.add(ResourceFactory.newClassPathResource("ruleflow.rf"),
  KnowledgeType.DRF);
      KnowledgeBuilderErrors errors = kbuilder.getErrors();
      if (errors.size() > 0) {
        for (KnowledgeBuilderError error: errors) {
          System.err.println(error);
        }
        throw new IllegalArgumentException("Could not parse knowledge.");
      }
      KnowledgeBase kbase = KnowledgeBaseFactory.newKnowledgeBase();
      kbase.addKnowledgePackages(kbuilder.getKnowledgePackages());
      return kbase;
    }

  }
```

Congratulations, you have successfully executed your first process! Because we added a logger to the session, you can easily check what happened internally by looking at the audit log. Select the "Audit View" tab on the bottom right, next to the Console tab. Click on the "Open Log" button (the first one one the right of the view) and navigate to the newly created "test.log" file in your project folder (if you are not sure where this project folder is located, right-click on the project folder and you will find the location in the "Resource" section). An image like the one below should be shown. It is a tree view of the events that occurred at runtime. Events that were executed as the direct result of another event are shown as the child of that event. This log shows that after starting the process, the start node, the action node and the end node were triggered in that order, after which the process instance was completed.



You can now start experimenting and designing your own process by modifying our example. Note that you can validate your process by clicking on the "Check the ruleflow model" button (the green check box action in the upper toolbar that shows up if you are editing a process). Processes will also be validated upon save and errors will be shown in the error view. Or you can continue reading our documentation to learn about our more advanced features.

Chapter 3. Rule Flow

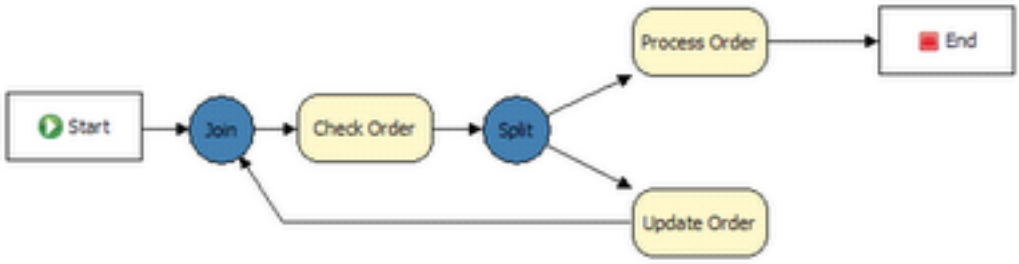


Figure 3.1. Ruleflow

A RuleFlow is a process that describes the order in which a series of steps need to be executed, using a flow chart. A process consists of a collection of nodes that are linked to each other using connections. Each of the nodes represents one step in the overall process while the connections specify how to transition from one node to the other. A large selection of predefined node types have been defined. This chapter describes how to define such processes and use them in your application.

3.1. Creating a RuleFlow process

Processes can be created by using one of the following three methods:

1. Using the graphical RuleFlow editor in the Drools plug-in for Eclipse
2. As an XML file, according to the XML process format as defined in the 'drools-processes' XML Schema Definition.
3. By directly creating a process using the Process API.

3.1.1. Using the graphical RuleFlow editor

The graphical RuleFlow editor is a editor that allows you to create a process by dragging and dropping different nodes on a canvas and editing the properties of these nodes. The graphical RuleFlow editor is part of the Drools plug-in for Eclipse. Once you have set up a Drools project (check the IDE chapter if you do not know how to do this), you can start adding processes. When in a project, launch the 'New' wizard (use "Ctrl+N" or by right-clicking the directory you would like to put your ruleflow in and selecting "New ... -> Other ...". Choose the section on "Drools" and then pick "RuleFlow file". This will create a new .rf file.

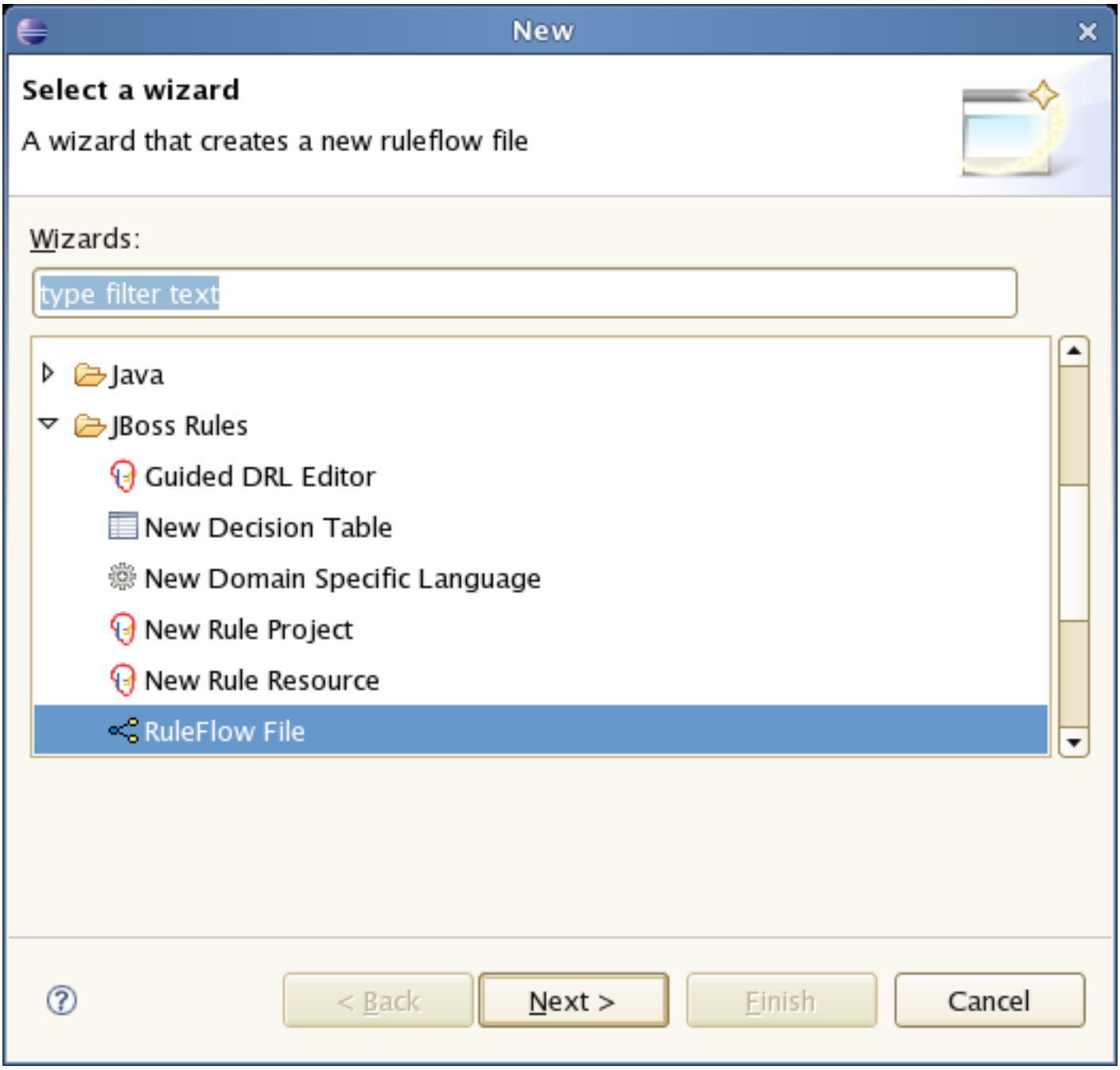


Figure 3.2. Creating a new RuleFlow file

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

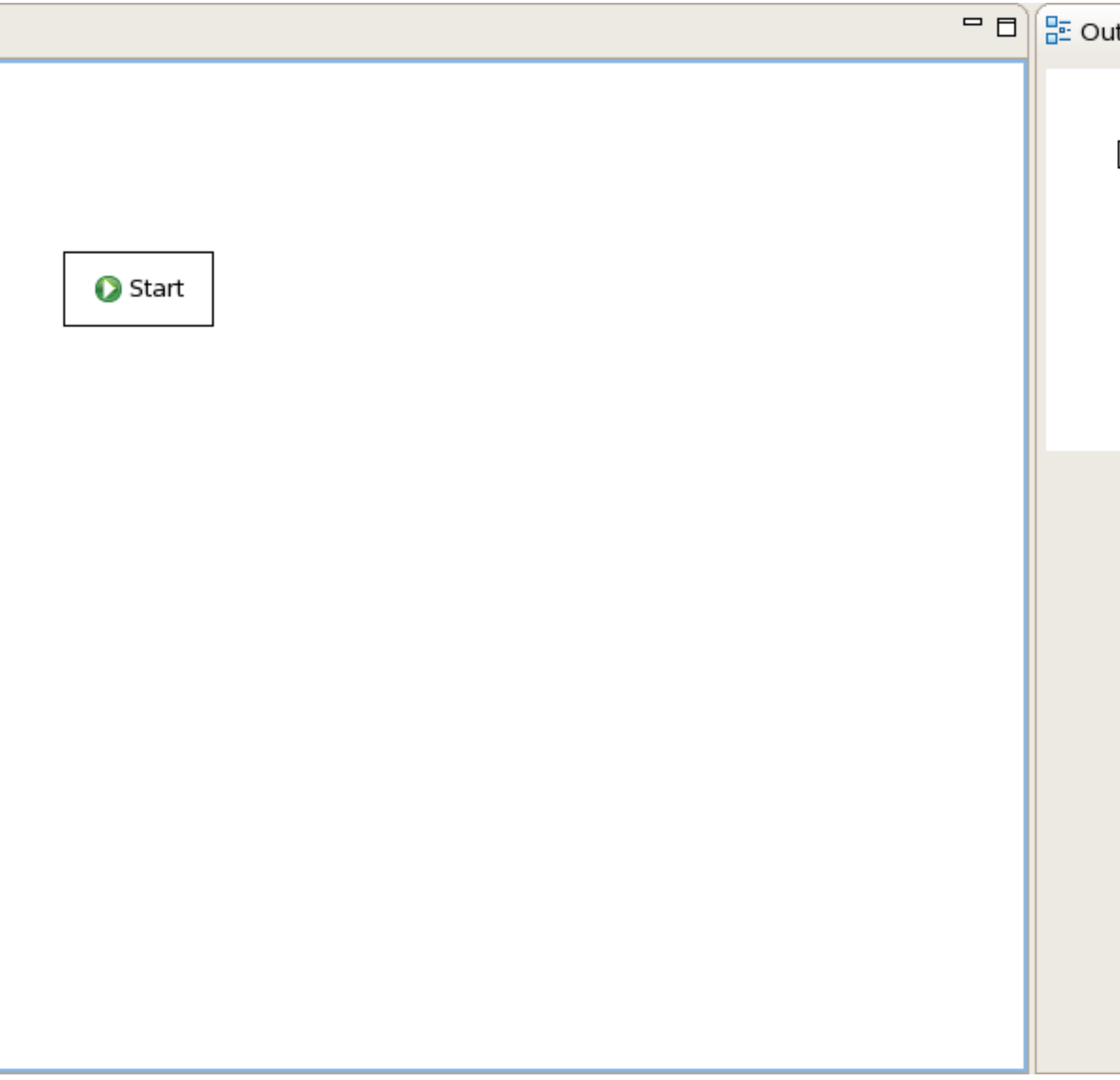


Figure 3.3. New RuleFlow process

The RuleFlow editor consists of a palette, a canvas and an outline view. To add new elements to the canvas, select the element you would like to create in the palette and then add them to the canvas by clicking on the preferred location. For example, click on the RuleFlowGroup icon

in the Component Pallette of the GUI - you can then draw a few rule flow groups. Clicking on an element in your ruleflow allows you to set the properties of that element.You can link the nodes together (as long as it is allowed by the different types of nodes) by using "Connection Creation" from the component palette.

You can keep adding nodes and connections to your process until it represents the business logic that you want to specify. You'll probably need to check the process for any missing information (by pressing the green "check" icon in the IDE menu bar) before trying to use it in your application.

3.1.2. Defining processes using XML

It is also possible to specify processes using the underlying XML directly. The syntax of these XML processes is defined using an XML Schema Definition. For example, the following XML fragment shows a simple process that contains a sequence of a start node, an action node that prints "Hello World" to the console, and an end node.

```
<?xml version="1.0" encoding="UTF-8"?>
<process xmlns="http://drools.org/drools-5.0/process"
         xmlns:xs="http://www.w3.org/2001/XMLSchema-instance"
         xs:schemaLocation="http://drools.org/drools-5.0/process
 drools-processes-5.0.xsd"
         type="RuleFlow" name="ruleflow" id="com.sample.ruleflow"
 package-name="com.sample" >

  <header>
  </header>

  <nodes>
    <start id="1" name="Start" x="16" y="16" />
    <actionNode id="2" name="Hello" x="128" y="16" >
      <action type="expression" dialect="mvel" >System.out.println("Hello
 World");</action>
    </actionNode>
    <end id="3" name="End" x="240" y="16" />
  </nodes>

  <connections>
    <connection from="1" to="2" />
    <connection from="2" to="3" />
  </connections>

</process>
```

The process XML file should consist of exactly one <process> element. This element contains parameters related to the process (the type, name, id and package name of the process), and consists of three main subsections: a <header> (where process-level information like variables, globals, imports and swimlanes can be defined), a <nodes> section that defines each of the nodes

in the process (there is a specific element for each of the different node types that defines the various parameters and possibly sub-elements for that node type), and a <connections> section that contains the connections between all the nodes in the process.

3.1.3. Defining processes using the Process API

While it is recommended to define processes using the graphical editor or the underlying XML (to shield yourself from internal APIs), it is also possible to define a process using the Process API directly. The most important process elements are defined in the org.drools.workflow.core and the org.drools.workflow.core.node packages. A "fluent API" is provided that allows you to easily construct processes in a readable manner using factories. At the end, you can validate the process that you were constructing manually. Some examples about how to build processes using this fluent API are added below.

3.1.3.1. Example 1

This is a simple example of a basic process with a ruleset node only:

```
      RuleFlowProcessFactory factory =

RuleFlowProcessFactory.createProcess("org.drools.HelloWorldRuleSet");
      factory
          // Header
          .name("HelloWorldRuleSet")
          .version("1.0")
          .packageName("org.drools")
          // Nodes
          .startNode(1).name("Start").done()
          .ruleSetNode(2)
              .name("RuleSet")
              .ruleFlowGroup("someGroup").done()
          .endNode(3).name("End").done()
          // Connections
          .connection(1, 2)
          .connection(2, 3);
      RuleFlowProcess process = factory.validate().getProcess();
```

You can see that we start by calling the static createProcess() method from the RuleFlowProcessFactory class. This method creates a new process with the given id and returns the RuleFlowProcessFactory that can be used to create the process. A typical process consists of three parts: a header part that contains global elements like the name of the process, imports, variables, etc. The nodes section contains all the different nodes that are part of the process and finally the connections section links these nodes to each other to create a flow chart.

So in this example, the header contains the name and the version of the process and the package name. After that you can start adding nodes to the current process. If you have auto-completion you can see that you different methods to create each of the supported node types at your disposal.

When you start adding nodes to the process, in this example by calling the startNode(), ruleSetNode() and endNode() methods, you can see that these methods return a specific NodeFactory, that allows you to set the properties of that node. Once you have finished configuring that specific node, the done() method returns you to the current RuleFlowProcessFactory so you can add more nodes if necessary.

When you finised adding nodes you must connect them by creating connections between them. This can be done by calling the connection method, which will link the earlier created nodes.

Finally, you can validate the generated process by calling the validate() method and retrieve the create RuleFlowProcess object.

3.1.3.2. Example 2

This example is using Split and Join nodes:

```
     RuleFlowProcessFactory factory =

RuleFlowProcessFactory.createProcess("org.drools.HelloWorldJoinSplit");
     factory
         // Header
         .name("HelloWorldJoinSplit")
         .version("1.0")
         .packageName("org.drools")
         // Nodes
         .startNode(1).name("Start").done()
         .splitNode(2).name("Split").type(Split.TYPE_AND).done()
         .actionNode(3).name("Action 1").action("mvel",
"System.out.println(\"Inside Action 1\")").done()
         .actionNode(4).name("Action 2").action("mvel",
"System.out.println(\"Inside Action 2\")").done()
         .joinNode(5).type(Join.TYPE_AND).done()
         .endNode(6).name("End").done()
         // Connections
         .connection(1, 2)
         .connection(2, 3)
         .connection(2, 4)
         .connection(3, 5)
         .connection(4, 5)
         .connection(5, 6);
     RuleFlowProcess process = factory.validate().getProcess();
```

This shows a simple split / join example. As you can see, split nodes can have multiple outgoing connections and joins multipe incomming connections. To understand the behaviour of the different types of split and join nodes, take a look at the documentation for each of these nodes.

3.1.3.3. Example 3

Now we show a more complex example with a ForEach node, where we have nested nodes:

```
      RuleFlowProcessFactory factory =

RuleFlowProcessFactory.createProcess("org.drools.HelloWorldForeach");
      factory
          // Header
          .name("HelloWorldForeach")
          .version("1.0")
          .packageName("org.drools")
          // Nodes
          .startNode(1).name("Start").done()
          .forEachNode(2)
              // Properties
              .linkIncomingConnections(3)
              .linkOutgoingConnections(4)
              .collectionExpression("persons")
              .variable("child", new ObjectDataType("org.drools.Person"))
              // Nodes
              .actionNode(3)
                  .action("mvel", "System.out.println(\"inside
action1\")").done()
              .actionNode(4)
                  .action("mvel", "System.out.println(\"inside
action2\")").done()
              // Connections
              .connection(3, 4)
              .done()
          .endNode(5).name("End").done()
          // Connections
          .connection(1, 2)
          .connection(2, 5);
      RuleFlowProcess process = factory.validate().getProcess();
```

Here you can see how we can include a ForEach node with nested action nodes. Note the linkIncommingConnections() and linkOutgoingConnections() methods that are called to link the foreach node with the internal action node. These methods are used to specify what the first and last nodes are inside the ForEach composite node.

3.2. Using a process in your application

There are two things you need to do to be able to execute processes from within your application: (1) you need to create a knowledge base that contain the definition of the process; and (2) you need to start the process by creating a session to communicate with the process engine and start the process.

1. *Creating a knowledge base*: Once you have a valid process, you can add the process to the knowledge base (note that this is almost identical to adding rules to the knowledge base, except for the type of knowledge added):

```
KnowledgeBuilder kbuilder = KnowledgeBuilderFactory.newKnowledgeBuilder();
kbuilder.add(ResourceFactory.newClassPathResource("MyProcess.rf"),
 ResourceType.DRF);
```

   After adding all your knowledge to the builder (you can add more than one process or even rules), you should probably check whether the process (and/or rules) have been parsed correctly and write out any errors like this:

```
KnowledgeBuilderErrors errors = kbuilder.getErrors();
if (errors.size() > 0) {
    for (KnowledgeBuilderError error: errors) {
        System.err.println(error);
    }
    throw new IllegalArgumentException("Could not parse knowledge.");
}
```

   Next you need to create the knowledge base that contains all the necessary processes (and rules) like this:

```
KnowledgeBase kbase = KnowledgeBaseFactory.newKnowledgeBase();
kbase.addKnowledgePackages(kbuilder.getKnowledgePackages());
```

2. *Starting a process*: Processes are only executed if you explicitly state that they should be executed. This is because you could potentially define a lot of processes in your knowledge base and the engine has no way to know when you would like to start each of these. To activate a particular process, you will need to start the process by calling the startProcess method on your session. For example:

```
StatefulKnowledgeSession ksession = kbase.newStatefulKnowledgeSession();
ksession.startProcess("com.sample.MyProcess");
```

The parameter of the startProcess method represents the id of the process that needs to be started (the process id needs to be specified as a property of the process, which are shown in the properties view when you click the background canvas of your process). If your process also requires the execution of rules during the execution of the process, you also need to call the ksession.fireAllRules() method to make sure the rules are executed as well. That's it!

You can also specify additional parameters that are used to pass on input data to the process, using the startProcess(String processId, Map parameters) method, that takes an additional set of parameters as name-value pairs. These parameters are then copied to the newly created process instance as top-level variables of the process.

You can also start a process from within a rule consequence, using

```
kcontext.getKnowledgeRuntime().startProcess("com.sample.MyProcess");
```

3.3. Detailed explanation of the different node types

A ruleflow process is a flow chart where different types of nodes are linked using connections. The process itself exposes the following properties:

- *Id*: The unique id of the process.
- *Name*: The display name of the process.
- *Version*: The version number of the process.
- *Package*: The package (namespace) the process is defined in.
- *Variables*: Variables can be defined to store data during the execution of your process (see the 'data' section for more details).
- *Swimlanes*: Specify the actor that is responsible for the execution of human tasks (see the 'human tasks' section for more details).
- *Exception Handlers*: Specify the behaviour when a fault occurs in the process (see the 'exceptions' section for more details).
- *Connection Layout*: Specify how the connections are visualized on the canvas using the connection layout property:
  - 'Manual' always draws your connections as lines going straight from their start to end point (with the possibility to use intermediate break points).
  - 'Shortest path' is similar, but it tries to go around any obstacles is might encounter between the start and end point (to avoid lines crossing nodes).
  - 'Manhatten' draws connections by only using horizontal and vertical lines.

A RuleFlow process supports different types of nodes:

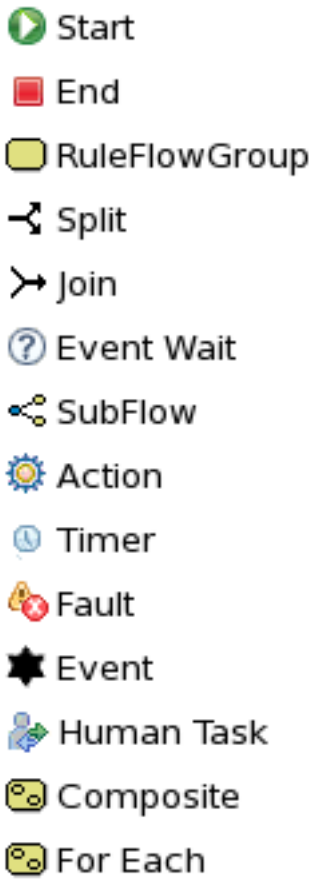


Figure 3.4. The different types of ruleflow nodes

1. **Start**: The start of the ruleflow. A ruleflow should have exactly one start node. The start node cannot have incoming connections and should have one outgoing connection. Whenever ruleflow process is started, the ruleflow will start executing here, and will then automatically continue to the first node linked to this start node, etc. It contains the following properties:
   - *Id*: The id of the node (which is unique within one node container).
   - *Name*: The display name of the node.
   - *Triggers*: A start node can also specify additional triggers that can be used to automatically start the process. Examples are a 'constraint' trigger that automatically starts the process if a given rule / contraint is satisfied, or an 'event' trigger that automatically starts the process if a specific event is signalled.
2. **End**: The end of the ruleflow. A ruleflow should have one or more end nodes. The end node should have one incoming connection and cannot have outgoing connections. It contains the following properties:
   - *Id*: The id of the node (which is unique within one node container).
   - *Name*: The display name of the node.

- *Terminate*: An end node can be terminating (default) or not. When a terminating end node is reached in the ruleflow, the ruleflow is terminated. If a ruleflow is terminated, all nodes that are still active in this ruleflow are cancelled first (which is possible if parallel paths are used). Non-terminating end nodes are simply end nodes in the process where the flow ends but other parallel paths still continue.

3. **RuleFlowGroup**: Represents a set of rules that need to be evaluated. The rules are evaluated when the node is reached. A RuleFlowGroup node should have one incoming connection and one outgoing connection. Rules can become part of a specific ruleflow group using the "ruleflow-group" attribute in the header. When a RuleSet node is reached in the ruleflow, the engine will start executing rules that are part of the corresponding ruleflow-group (if any). Execution will automatically continue to the next node if there are no more active rules in this ruleflow-group. This means that, during the execution of a ruleflow-group, it is possible that new activations belonging to the currently active ruleflow-group are added to the agenda due to changes made to the facts by the other rules. Note that the ruleflow will immediately continue with the next node if it encounters a ruleflow-group where there are no active rules at that point. If the ruleflow-group was already active, the ruleflow-group will remain active and exeution will only continue if all active rules of the ruleflow-group has been completed. It contains the following properties:

   - *Id*: The id of the node (which is unique within one node container).
   - *Name*: The display name of the node.
   - *RuleFlowGroup*: The name of the ruleflow-group that represents the set of rules of this RuleFlowGroup node.
   - *Timers*: Timers that are linked to this node (see the 'timers' section for more details).

4. **Split**: Allows you to create branches in your ruleflow. A split node should have one incoming connection and two or more outgoing connections. There are three types of splits currently supported:

   - AND means that the control flow will continue in all outgoing connections simultaneously (paralellism).
   - XOR means that exactly one of the outgoing connections will be chosen (decision). Which connection is decided by evaluating the constraints that are linked to each of the outgoing connections. Constraints are specified using the same syntax as the left-hand side of a rule. The constraint with the lowest priority number that evaluates to true is selected. Note that you should always make sure that at least one of the outgoing connections will evaluate to true at runtime (the ruleflow will throw an exception at runtime if it cannot find at least one outgoing connection). For example, you could use a connection which is always true (default) with a high priority number to specify what should happen if none of the other connections can be taken.
   - OR means that all outgoing connections whose condition evaluates to true are selected. Conditions are similar to the XOR split, except that the priorities are not taken into account.

Note that you should make sure that at least one of the outgoing connections will evaluate to true at runtime (the ruleflow will throw an exception at runtime if it cannot find an outgoing connection).

It contains the following properties:

- *Id*: The id of the node (which is unique within one node container).
- *Name*: The display name of the node.
- *Type*: The type of the split node, t.e. AND, XOR or OR (see above).
- *Constraints*: The constraints linked to each of the outgoing connections (in case of an (X)OR split).

5. **Join**: Allows you to synchronize multiple branches. A join node should have two or more incoming connections and one outgoing connection. There are three types of splits currently supported:
   - AND means that is will wait until all incoming branches are completed before continuing.
   - XOR means that it continues if one of its incoming branches has been completed. If it is triggered from more than one incoming connection, it will trigger the next node for each of those triggers.
   - Discriminator means that it continues if one of its incoming branches has been completed. At that point, it will wait until all other connections have been triggered as well. At that point, it will reset, so that it can trigger again when one of its incoming branches has been completed.
   - n-of-m means that it continues if n of its m incoming branches have been completed. The n variable could either be hardcoded to a fixed value, or could also refer to a process variable that will contain the number of incoming branches to wait for.

   It contains the following properties:
   - *Id*: The id of the node (which is unique within one node container).
   - *Name*: The display name of the node.
   - *Type*: The type of the join node, t.e. AND, XOR or Discriminator (see above).
   - *n*: The number of incoming connections to wait for (in case of a n-of-m join).
6. **Event wait (or milestone)**: Represents a wait state. An event wait should have one incoming connection and one outgoing connection. It specifies a constraint which defines how long the process should wait in this state before continuing. For example, a constraint in an order entry application might specify that the process should wait until no more errors are found in the given order. Constraints are specified using the same syntax as the left-hand side of a rule. When a wait node is reached in the ruleflow, the engine will check the associated constraint. If the constraint evaluates to true directly, the flow will continue imediately. Otherwise, the flow will

continue if the constraint is satisfied later on, for example when a fact is inserted in, updated or removed from the working memory. It contains the following properties:

- *Id*: The id of the node (which is unique within one node container).
- *Name*: The display name of the node.
- *Constraint*: Defines when the process can leave this state and continue.

7. **SubFlow**: represents the invocation of another process from withing this process. A sub-process node should have one incoming connection and one outgoing connection. When a SubProcess node is reached in the ruleflow, the engine will start the process with the given id. It contains the following properties:
   - *Id*: The id of the node (which is unique within one node container).
   - *Name*: The display name of the node.
   - *ProcessId*: The id of the process that should be executed.
   - *Wait for completion*: If this property is true, the subflow node will only continue if that subflow process has terminated its execution (completed or aborted); otherwise it will continue immediately after starting the sub-process.
   - *Independent*: If this property is true, the sub-process is started as an independent process, which means that the subflow process will not be terminated if this process reaches an end node; otherwise the active sub-process will be cancelled on termination (or abortion) of the process.
   - *On entry/exit actions*: Actions that are executed upon entry / exit of this node.
   - *Parameter in/out mapping*: A SubFlow node can also define in- and out-mappings for variables. The value of variables in this process with given variable name in the in-mappings will be used as parameters (with the associated parameter name) when starting the process. The value of the variables in the sub-process with the given variable name in the out-mappings will be copied to the variables of this process when the sub-process has been completed. Note that can only use out-mappings when "Wait for completion" is set to true.
   - *Timers*: Timers that are linked to this node (see the 'timers' section for more details).
8. **Action**: represents an action that should be executed in this ruleflow. An action node should have one incoming connection and one outgoing connection. The associated action specifies what should be executed. An action should specify which dialect is used to specify the action (e.g. Java or MVEL), and the actual action code. The action code can refer to any globals, the special 'drools' variable which implements KnowledgeHelper (can for example be used to access the working memory (drools.getWorkingMemory())) and the special 'context' variable which implements the ProcessContext (can for example be used to access the current ProcessInstance or NodeInstance and get/set variables). When an action node is reached in

the ruleflow, it will execute the action and continue with the next node. It contains the following properties:

- *Id*: The id of the node (which is unique within one node container).
- *Name*: The display name of the node.
- *Action*: The action associated with this action node.

9. **Timer**: represents a timer that can trigger one or multiple times after a given period of time. A Timer node should have one incoming connection and one outgoing connection. The timer delay specifies how long (in milliseconds) the timer should wait before triggering the first time. The timerperiod specifies the time between two subsequenct triggers. A period of 0 means that the timer should only be triggered once. When a timer node is reached in the ruleflow, it will execute the associated timer. The timer is cancelled if the timer node is cancelled (e.g. by completing or aborting the process). Check out the section on timers to find out more information. The timer node contains the following properties:

   - *Id*: The id of the node (which is unique within one node container).
   - *Name*: T

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

   he display name of the node.

   - *Timer delay*: The delay (in milliseconds) that the node should wait before triggering the first time.
   - *Timer period*: The period (in milliseconds) between two subsequent triggers. If the period is 0, the timer should only be triggered once.

10. **Fault**: A fault node can be used to signal an exceptional condition in the process. A fault node should have one incoming connection and no outgoing connections. When a fault node is reached in the ruleflow, it will throw a fault with the given name. The process will search for an appropriate exception handler that is capable of handling this kind of fault. If no fault handler is found, the process instance will be aborted. It contains the following properties:

    - *Id*: The id of the node (which is unique within one node container).
    - *Name*: The display name of the node.

- *FaultName*: The name of the fault. This name is used to search for appriopriate exception handlers that is capable of handling this kind of fault.
- *FaultVariable*: The name of the variable that contains the data associated with this fault. This data is also passed on to the exception handler (if one is found).

11. **Event**: An event node can be used to respond to (internal/external) events during the execution of the process. An event node should have no incoming connections and one outgoing connection. An event node specifies the type of event that is expected. Whenever that type of event is detected, the node connected to this event node will be triggered. It contains the following properties:
    - *Id*: The id of the node (which is unique within one node container).
    - *Name*: The display name of the node.
    - *EventType*: The type of event that is expected.
    - *VariableName*: The name of the variable that will contain the data associated with this event (if any) when this event occurs.
    - *Scope*: An event could be used to listen to internal events only, i.e. events that are signalled to this process instance directly, using processInstance.signalEvent(String type, Object data). When an event node is defined as external, it will also be listening to (external) events that are signalled to the process engine directly, using workingMemory.signalEvent(String type, Object event).
12. **Human Task**: Processes can also involve tasks that need to executed by human actors. A task node represents an atomic task that needs to be executed by a human actor. A human task node should have one incoming connection and one outgoing connection. Human task nodes can be used in combination with swimlanes to assign multiple human tasks to similar actors. For more detail, check the 'human tasks' chapter. A human task node is actually nothing more than a specific type of work item node (of type "Human Task"). A human task node contains the following properties:
    - *Id*: The id of the node (which is unique within one node container).
    - *Name*: The display name of the node.
    - *TaskName*: The name of the human task.
    - *Priority*: An integer indicating the priority of the human task.
    - *Comment*: A comment associated with the human task.
    - *ActorId*: The actor id that is responsible for executing the human task. A list of actor id's can be specified using a comma (',') as separator.

- *Skippable*: Specifies whether the human task can be skipped (i.e. the actor decides not to execute the human task).
- *Content*: The data associated with this task.
- *Swimlane*: The swimlane this human task node is part of. Swimlanes make it easy to assign multiple human tasks to the same actor. See the human tasks chapter for more detail on how to use swimlanes.
- *Wait for completion*: If this property is true, the human task node will only continue if the human task has been terminated (i.e. completed or any other terminal state); otherwise it will continue immediately after creating the human task.
- *On entry/exit actions*: Actions that are executed upon entry / exit of this node.
- *Parameter mapping*: Allows copying the value of process variables to parameters of the human task. Upon creation of the human tasks, the values will be copied.
- *Result mapping*: Allows copying the value of result parameters of the human task to a process variable. Upon completion of the human task, the values will be copied. Note that can only use result mappings when "Wait for completion" is set to true. A human task has a result variable "Result" that contains the data returned by the human actor. The variable "ActorId" contains the id of the actor that actually executed the task.
- *Timers*: Timers that are linked to this node (see the 'timers' section for more details).

13. **Composite**: A composite node is a node that can contain other nodes (i.e. acts as a node container). It thus allows creating a part of the flow embedded inside a composite node. It also allows you to define additional variables and exception handlers that are accessible for all nodes inside this container. A composite node should have one incoming connection and one outgoing connection. It contains the following properties:
    - *Id*: The id of the node (which is unique within one node container).
    - *Name*: The display name of the node.
    - *StartNodeId*: The id of the node (inside this node container) that should be triggered when this node is triggered.
    - *EndNodeId*: The id of the node (inside this node container) that represents the end of the flow contained in this node. When this node is completed, the composite node will also be completed and trigger its outgoing connection. All other executing nodes within this composite node will be cancelled.
    - *Variables*: Additional variables can be defined to store data during the execution of this node (see the 'data' section for more details).

- *Exception Handlers*: Specify the behaviour when a fault occurs in this node container (see the 'exceptions' section for more details).

14. **For Each**: A for each node is a special kind of composite node that allows you to execute the contained flow multiple times, once for each element in a collection. A for each node should have one incoming connection and one outgoing connection. A for each node waits for completion of the embedded flow for each of its elements before continuing. It contains the following properties:

    - *Id*: The id of the node (which is unique within one node container).
    - *Name*: The display name of the node.
    - *StartNodeId*: The id of the node (inside this node container) that should be triggered for each of the elements in a collection.
    - *EndNodeId*: The id of the node (inside this node container) that represents the end of the flow contained in this node. When this node is completed, the execution of the for each node will also be completed for that element. and trigger its outgoing connection. All other executing nodes within this composite node will be cancelled.
    - *CollectionExpression*: The name of a variable that represents the collection of elements that should be iterated over. The collection variable should be of type java.util.Collection.
    - *VariableName*: The name of the variable that will contain the selected element from the collection. This can be used to gives nodes inside this composite node access to the selected element.

15. **Work Item**: Represents an (abstract) unit of work that should be executed in this process. All work that is executed outside the process engine should be represented (in a declarative way) using a work item. Different types of work items are predefined, like for example sending an email, logging a message, etc. However, the user can define domain-specific work items (using a unique name and by defining the paramaters (input) and results (output) that are associated with this type of work). See the chapter about domain-specific processes for a detailed explanation and illustrative examples of how to define and use work items in your processes. When a work item node is reached in the process, the associated work item is executed. A work item node should have one incoming connection and one outgoing connection.

    - *Id*: The id of the node (which is unique within one node container).
    - *Name*: The display name of the node.
    - *Wait for completion*: If the property "Wait for completion" is true, the WorkItem node will only continue if the created work item has terminated its execution (completed or aborted); otherwise it will continue immediately after starting the work item.
    - *Parameter mapping*: Allows copying the value of process variables to parameters of the work item. Upon creation of the work item, the values will be copied.

- *Result mapping*: Allows copying the value of result parameters of the work item to a process variable. Each type of work can define result parameters that will (potentially) be returned after the work item has been completed. A result mapping can be used to copy the value of the given result parameter to the given variable in this process. For example, the "FileFinder" work item returns a list of files that match the given search criteria as a result parameter 'Files'. This list of files can then be bound to a process variable for use within the process. Upon completion of the work item, the values will be copied. Note that can only use result mappings when "Wait for completion" is set to true.

- *On entry/exit actions*: Actions that are executed upon entry / exit of this node.

- *Timers*: Timers that are linked to this node (see the 'timers' section for more details).

- *Additional parameters*: Each type of work item can define additional parameters that are relevant for that type of work. For example, the "Email" work item defines additional parameters like 'From', 'To', 'Subject' and 'Body'. The user can either fill in values for these parameters directly, or define a parameter mapping that will copy the value of the given variable in this process to the given parameter (if both are specified, the mapping will have precedence). Parameters of type String can use #{expression} to embed a value in the String. The value will be retrieved when creating the work item and the #{...} will be replaced by the toString() value of the variable. The expression could simply be the name of a variable (in which case it will be resolved to the value of the variable), but more advanced MVEL expressions are possible as well, like #{person.name.firstname}.

3.4. Data

While the flow graph focusses on specifying the control flow of the process, it is usually also necessary to look at the process from a data perspective. During the execution of a process, data can retrieved, stored, passed on and (re)used throughout the entire process.

Runtime data can be stored during the execution of the process using variables. A variable is defined by a name and a data type. This could be a basic data types (e.g. boolean, integer, String) or any kind of Object. Variables can be defined inside a variable scope. The top-level scope is the variable scope of the process itself. Sub-scopes can be defined using a composite node. Variables that are defined in a sub-scope are only accessible for nodes within that scope.

Whenever a variable is accessed, the process will search for the appropriate variable scope that defines the variable. Nesting of variable scopes is allowed: a node will always search for a variable in its parent container. If the variable cannot be found, it will look in that one's parent container, etc. until the process instance itself is reached. If the variable cannot be found, either null will be returned (in case of a read) or an error message will be shown that the variable could not be found (in case of a write), after which the process will continue without setting the paramater.

Variables can be used in various ways:

- Process-level variables can be set when starting a process by providing a map of parameters when invoking the startProcess method. These parameters will be set as variables on the process scope.
- Actions can access variables directly (by simply using the name of the variable as a parameter name).

```
person.setAge(10); // with "person" a variable in the process
```

  Changing the value of a variable can be done through the knowledge context:

```
kcontext.setVariable(variableName, value);
```

- Work items and sub-flows can pass the value of parameters to the outside world by mapping the variable to one of the work item parameters (either using a parameter mapping or by using #{expression} directly inside a String parameter. The results of a work item can also be copied to a variable using a result mapping.
- Various other nodes can also access data. For example, event nodes can store the data associated to the event in a variable, exception handlers can read error data from a specific variable, etc. Check the properties of the different node types for more information.

Finally, processes and rules all have access to globals (globally defined variables that are considered immutable with regard to rule evaluation) and data in the knowledge session. The knowledge session can be accessed in actions using the knowledge context:

```
kcontext.getKnowledgeRuntime().insert( new Person("..") );
```

3.5. Constraints

Constraints can be used in various locations in your processes, like for example decision points (i.e. an (X)OR split), wait constraints, etc. Drools Flow supports two types of constraints:

- code: Code constraints are expressions that return a boolean value. They are evaluated directly whenever they are reached. We currently support two dialects for expressing these code constraints: java and MVEL. Both java and MVEL code constraints have direct access to the globals and variables defined in the process. An example of a valid java code constraint would for example be (with person a variable in the process):

```
return person.getAge() > 20;
```

  An similar example of a valid MVEL code constraint would be:

```
return person.age > 20;
```

- rule: Rule constraints are equals to normal Drools rule conditions. They use the Drools Rule Language syntax to express possibly complex constraints. These rules can (like any other rule) refer to data in the working memory. They can also refer to globals directly. An example of a valid rule constraint would for example be:

```
Person( age > 20 )
```

  which will search for a person older than 20 in the working memory.

Rule constraints do not have direct access to variables defined inside the process. It is however possible to refer to the current process instance inside a rule constraint, by adding the process instance to the working memory and matching to the process instance inside your rule constraint. We have added special logic to make sure that a variable "processInstance" of type WorkflowProcessInstance will only match to the current process instance and not to other process instances in the working memory. Note that you are however responsible yourself to insert (and possibly update) the process instance into the session (for example using Java code or an (on-entry or on-exit or explicit) action in your process). The following exampleof a rule constraint will search for a person with the same name as the value stored in the variable "name" of the process:

```
processInstance: WorkflowProcessInstance()
Person( name == ( processInstance.getVariable("name") ) )
# add more constraints here ...
```

3.6. Actions

Actions can be used in different ways:

- Action node
- On entry/exit actions
- Actions that specify the behaviour of exception handlers

Actions have access to globals and the variables that are defined for the process and the 'kcontext' variable. This variable is of type org.drools.runtime.process.ProcessContext and can be used for

- Getting the current node instance (if applicable). The node instance could be queried for data (name, type). You can also cancel the current node instance.
- Getting the current process instance. This process instance could be queried for data (name, id, processId, etc.), abort process instance, signal events (internal).
- Data: getting or setting the value of variables
- Accessing the KnowledgeRuntime: this allows you do things like starting a process, signalling events (external), inserting data, etc.

Drools currently supports two dialects: the java and the MVEL dialect. Java actions should be valid Java code. MVEL actions can use the business scripting language MVEL to express

the action. MVEL accepts any valid Java code but also provides aditional support for nested accesses of parameters (e.g. person.name instead of person.getName()), and many other scripting improvements. Therefore, MVEL usually allows more business user friendly action expressions. For example, an action that prints out the name of the person in the "requester" variable of the process would look like this:

```
// using the Java dialect
System.out.println( person.getName() );

// Similarly, using the MVEL dialect
System.out.println( person.name );
```

3.7. Events

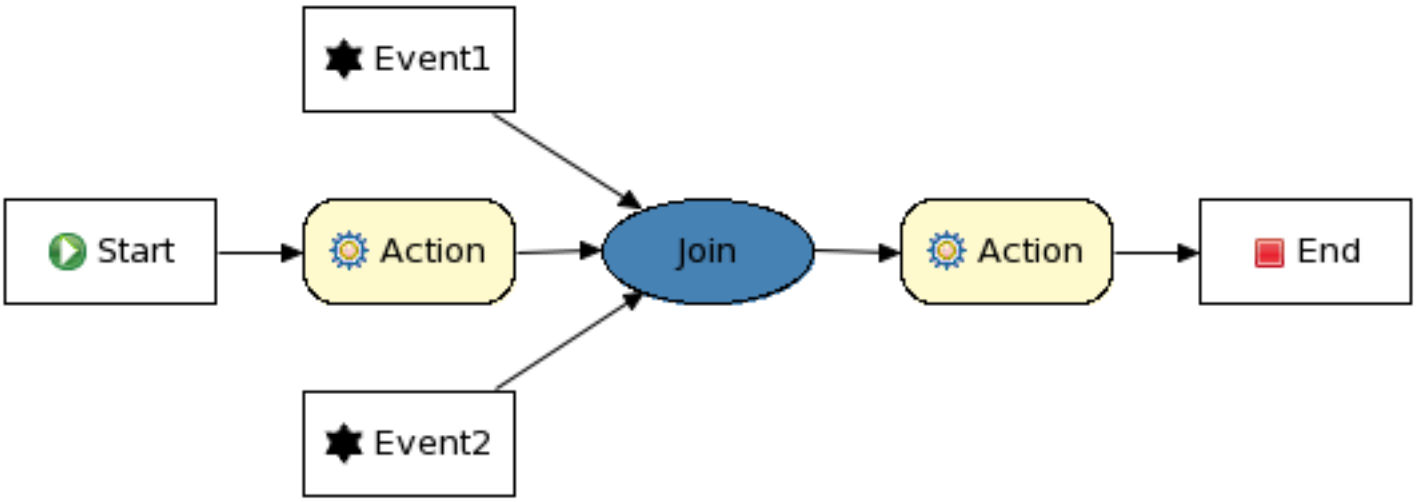


Figure 3.5. A sample process using events

During the execution of a process, the process engine makes sure that all the relevant tasks are executed according to the process plan, by requesting the execution of work items and waiting for the results. However, it is also possible that the process should respond to events that were not directly requested by the process engine. Explicitly representing these events in a process allows the process author to specify how the process should react whenever such events occur.

Events have a type and possibly data associated with the event. Users are free to define their own types of events and the data that is associated with this event.

A process can specify how to respond to events by using event nodes. An event node needs to specify the type of event the node is interested in. It can also define a variable name, which defines the variable that the data that is associated with the event will be copied to. This allows subsequent nodes in the process to access the event data and take appropriate action based on this data.

An event can be signalled to a running instance of a process in a number of ways:

- Internal event: Any action inside a process (e.g. the action of an action node, or on-entry or on-exit actions of nodes) can signal the occurence of an internal event to the surrounding process instance using

```
context.getProcessInstance().signalEvent(type, eventData);
```

- External event: A process instance can be notified of an event from outside using

```
processInstance.signalEvent(type, eventData);
```

- External event using event correlation: Instead of notifying a process instance directly, it is also possible to have the engine automatically determine which process instances might be interested in an event using event correlation (based on the event type). All process instances that have specified they are interested in receiving external events of that type (e.g. by having an event node that is listening to external events of that type) will be notified. You can signal such an event to the process engine using

```
workingMemory.signalEvent(type, eventData);
```

Events could also be used to start a process. Whenever a start node defines an event trigger of a specific type, a new process instance will be started every time that type of event is signalled to the process engine.

3.8. Exceptions

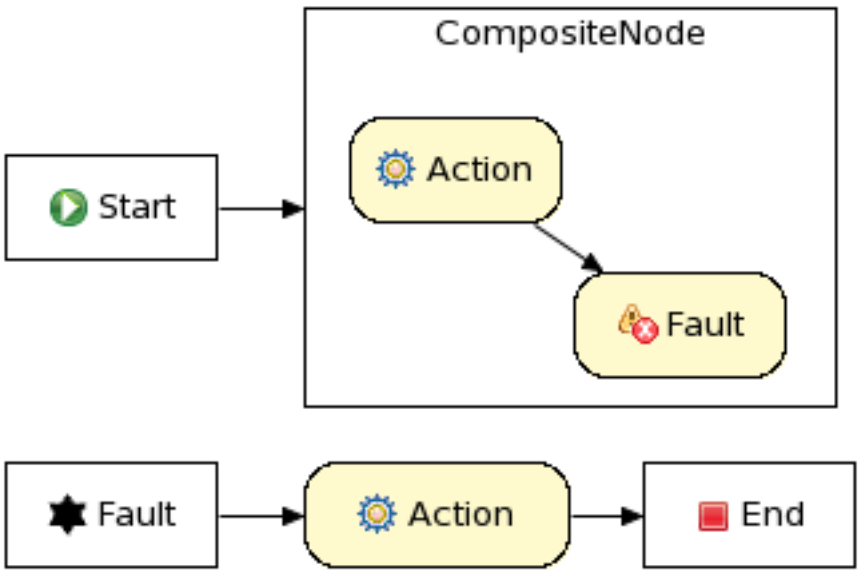


Figure 3.6. A sample process using exception handlers

Whenever an exceptional condition occurs during the execution of a process, a fault could be raised to signal the occurrence of this exception. The process will then search for an appropriate exception handler that is capable of handling such a fault.

Similar to events, faults also have a type and possibly data associated with the fault. Users are free to define their own types of faults and the data that is associated with this fault.

Faults can be created using a fault node: A fault node generates a fault of the given type (i.e. the fault name). If the fault node specifies a fault variable, the value of the given variable will be associated with the fault.

Whenever a fault is created, the process will search for an appropriate exception handler that is capable of handling the given type of fault. Processes and composite nodes both can define exception handlers for handling faults. Nesting of exception handlers is allowed: a node will always search for an appropriate exception handler in its parent container. If none is found, it will look in that one's parent container, etc. until the process instance itself is reached. If no exception handler can be found, the process instance will be aborted, resulting in the cancellation of all nodes inside the process.

Exception handlers can also specify a fault variable. The data associated with the fault (if any) will be copied to this variable if the exception handler is selected to handle the fault. This allows subsequent actions / nodes in the process to access the fault data and take appropriate action based on this data.

Exception handlers need to define an action that specifies how to respond to the given fault. In most cases, the behaviour that is needed to react to the given fault cannot be handled in one action. It is therefore recommended to have the exception handler signal an event of a specific type (in this case "Fault") using

```
context.getProcessInstance().signalEvent("FaultType",
 context.getVariable("FaultVariable");
```

3.9. Timers

Timers can be used to wait for a predefined amount of time, before triggering. They could be used to specify timeout behaviour, to trigger certain logic after a certain period or repeat it at regular intervals.

A timer needs to specify a delay and a period. The delay specifies the amount of time (in milliseconds) to wait after activation before triggering the timer the first time. The period defines the time between subsequent activations. If the period is 0, the timer will only be triggered once.

The timer service is responsible for making sure that timers get triggered at the appropriate times. Timers can also be cancelled, meaning that the timer will no longer be triggered.

Timers can be used in two ways inside a process:

- TimerNode: A timer node is a node that can be added to the process flow. When the timer node is triggered, the associated timer is activated. The timer node will trigger the next node whenever the timer is triggered. This means that the outgoing connection of a timer node can be triggered multiple times if a period is used. Cancelling a timer node also cancels the associated timer.
- Timers associated to nodes: it is also possible to add timers to event-based nodes like work items, sub-flows, etc. The timers associated with these nodes are activated once the node is triggered. The associated action is executed if the timer is triggered. This could for example be used to send out notifications at regular time intervals when the execution of a task takes too long, or signal an event or a fault in case of a time-out. When the node these timers are defined for is completed, the timers are automatically cancelled.

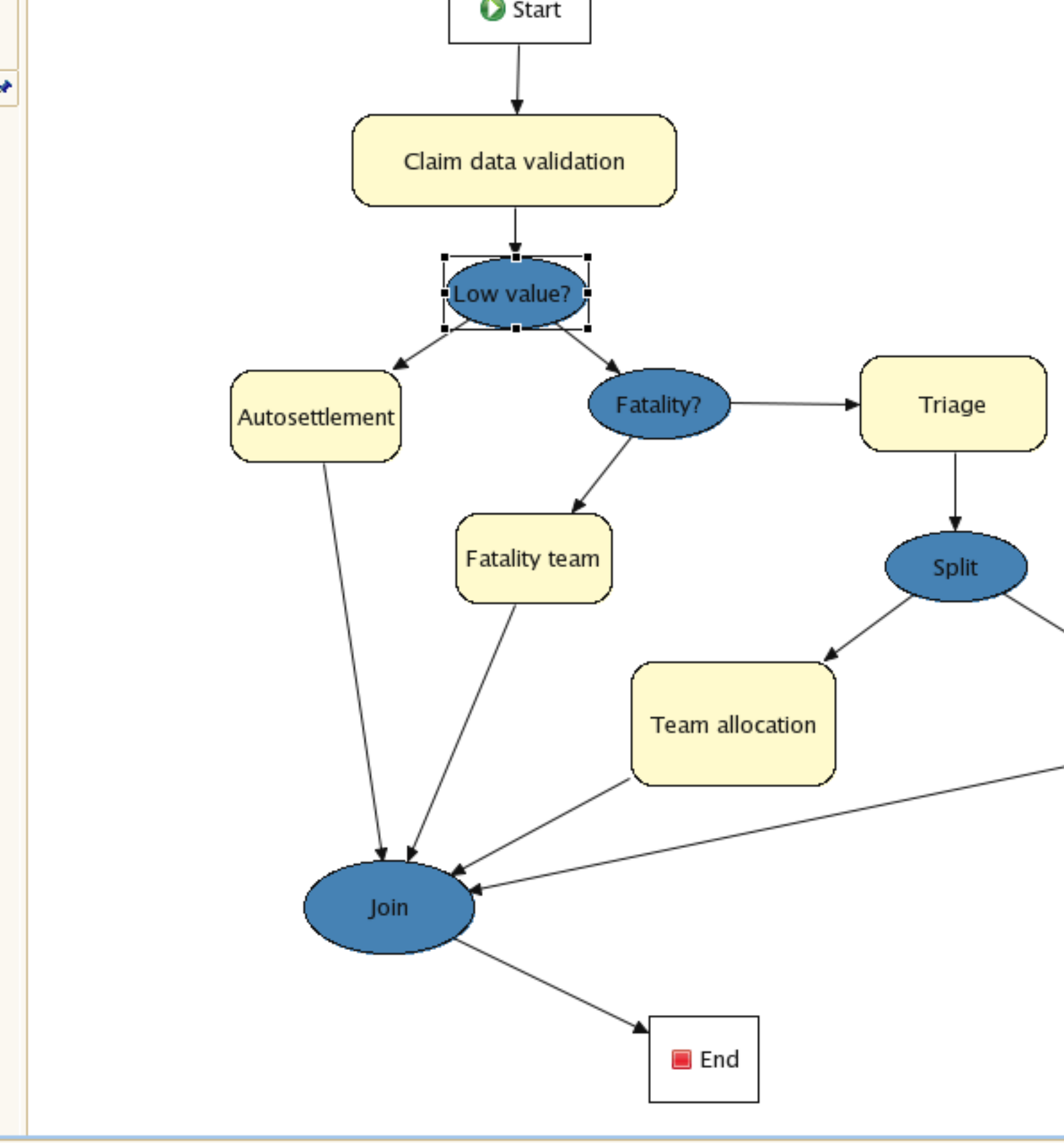
By default, the Drools engine is a passive component, meaning that it will only start processing if you tell it to (for example, you first insert the necessary data and then tell the engine to start processing). In passive mode, a timer that has been triggered will be put on the action queue. This means that it will be executed the next time the engine is told to start executing by the user (using fireAllRules() or if the engine is already / still running), in which case the timer will be executed automatically.

When using timers, it does usually make sense to make the Drools engine an active component, meaning that it will execute actions whenever they become available (and not wait until the user tells it to start executing again). This would mean a timer would be executed once it is triggered. To make the engine fire all actions continuously, you must call the fireUntilHalt() method. That means the engine will continue firing until the engine is halted. The following fragment shows how to do this (note that you should call fireUntilHalt() in a separate thread as it will only return if the engine has been halted (by either the user or some logic calling halt() on the session):

```
new Thread(new Runnable() {
  public void run() {
    ksession.fireUntilHalt();
  }
}).start();

// starting a new process instance
ksession.startProcess("...");
// any timer that will trigger will now be executed automatically
```

3.10. Assigning rules to a ruleflow group

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



erties | Console | Audit View | JUnit | Coverlipse Class View

Value

Low value?

XOR

Figure 3.8. Complex ruleflow

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

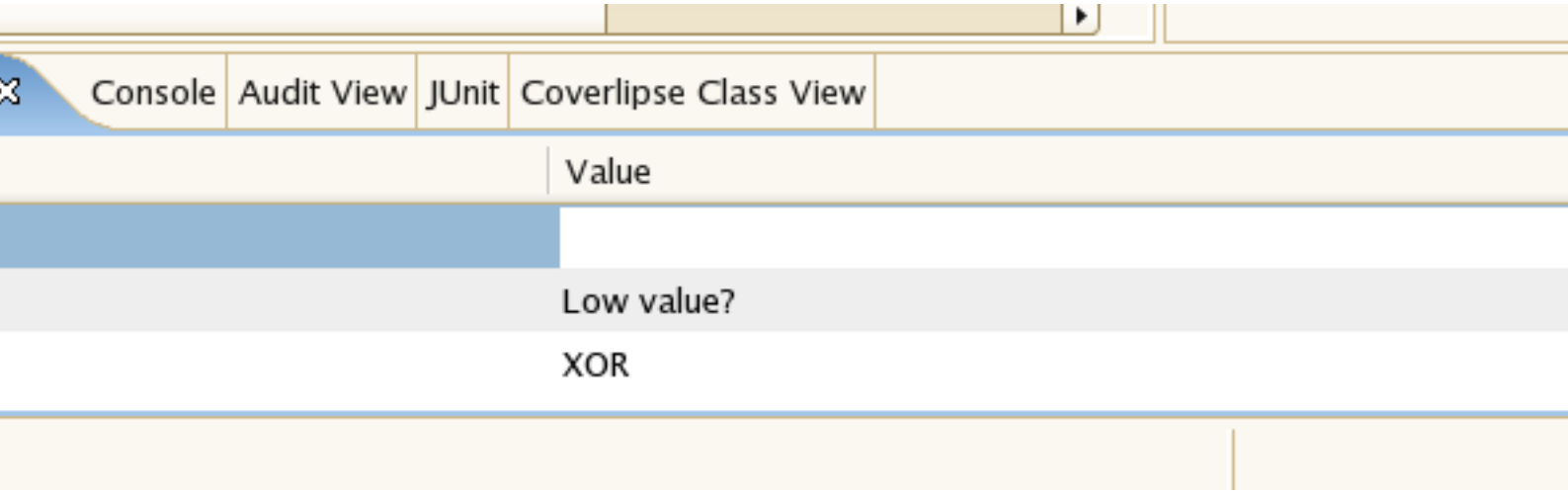


Figure 3.9. Split types

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

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

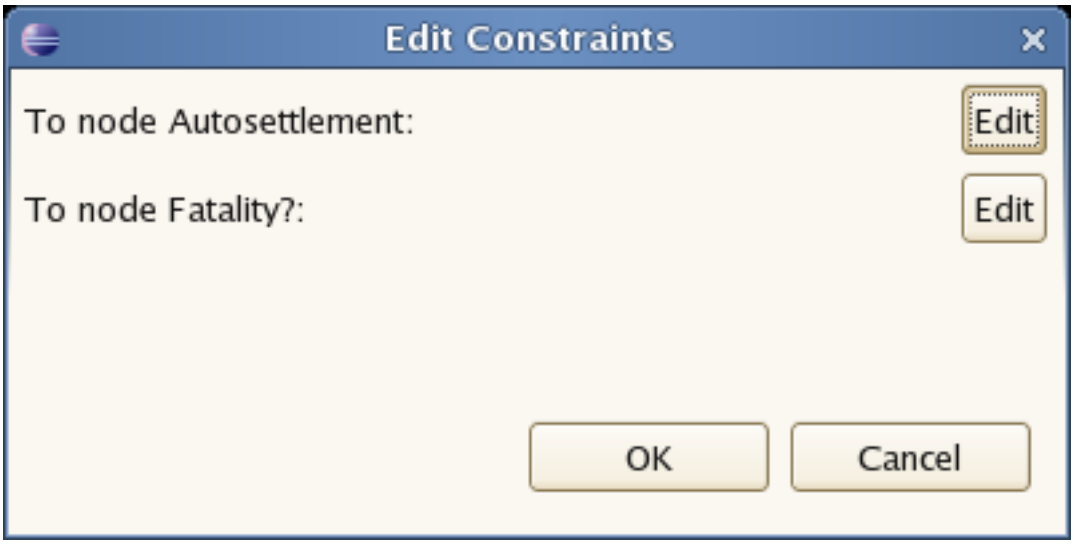


Figure 3.10. Edit constraints

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

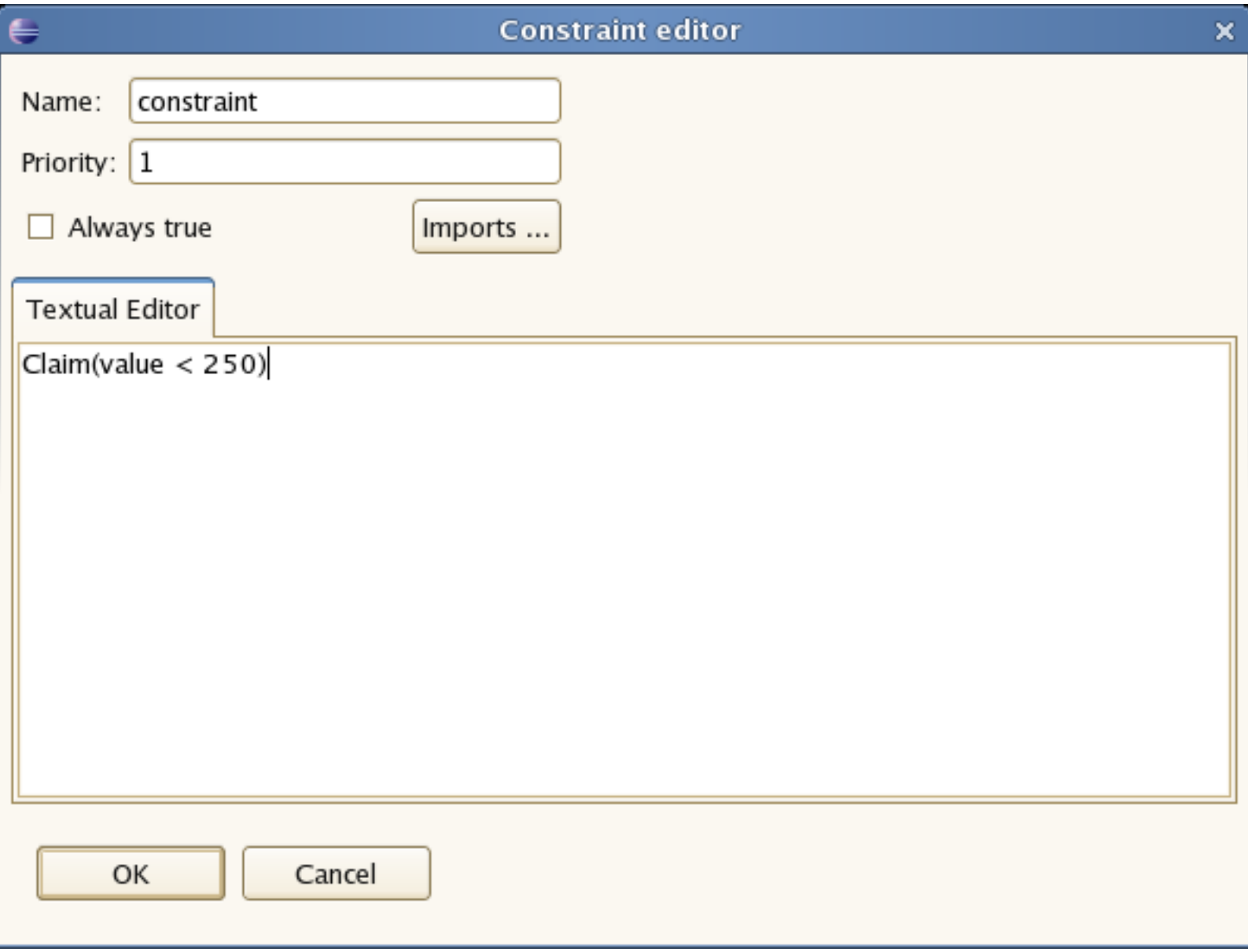


Figure 3.11. Constraint editor

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

Chapter 4. Drools Flow API

4.1. Knowledge Base

Our knowledge-based API allows you to first create a knowledge base that contains all the necessary knowledge. This includes of course all the relevant process definitions, but also other knowledge types like rules. The following code snippet shows how to create a knowledge base consisting of only one process definition: use a knowledge builder to add a resource, check for errors and create the knowledge base.

```
KnowledgeBuilder kbuilder = KnowledgeBuilderFactory.newKnowledgeBuilder();
kbuilder.add(ResourceFactory.newClassPathResource("ruleflow.rf"),
 ResourceType.DRF);
KnowledgeBuilderErrors errors = kbuilder.getErrors();
if (errors.size() > 0) {
  for (KnowledgeBuilderError error: errors) {
    System.err.println(error);
  }
  throw new IllegalArgumentException("Could not parse knowledge.");
}
KnowledgeBase kbase = KnowledgeBaseFactory.newKnowledgeBase();
kbase.addKnowledgePackages(kbuilder.getKnowledgePackages());
```

Note that the knowledge-based API allows users to add different types of resources (e.g. rules and processes) in almost identical ways into the same knowledge base. This allows user that know how to user Drools Flow to start using Expert of Fusion (and even integrate these different types of knowledge) almost instantaniously.

4.2. Session

Next, you should create a session to interact with the engine. Again, the API is knowledge-based, supporting different types of knowledge, with a specific extension for each knowledge type. The following code snippet shows how easy it is to create a session based on the earlier created knowledge base and start a process.

```
StatefulKnowledgeSession ksession = kbase.newStatefulKnowledgeSession();
ProcessInstance processInstance =
 ksession.startProcess("com.sample.ruleflow");
```

The ProcessRuntime interface defines all the methods on the session for interacting with processes, as shown below. Check out the JavaDocs to get a detailed explanation for each of the methods.

```
```

```
ProcessInstance startProcess(String processId);
ProcessInstance startProcess(String processId, Map<String, Object>
 parameters);
void signalEvent(String type, Object event);
Collection<ProcessInstance> getProcessInstances();
ProcessInstance getProcessInstance(long id);
WorkItemManager getWorkItemManager();
```

4.3. Events

Both the stateful and stateless knowledge session provide methods for registering (and removing) listeners. ProcessEventListeners can be used to listen to process-related events, like starting or completing a processes and triggering and leaving a node. Below the different methods of a ProcessEventListener are shown. The event object provides access to related information like the process instance and/or node instance linked to the event.

```
public interface ProcessEventListener {

  void beforeProcessStarted(ProcessStartedEvent event);
  void afterProcessStarted(ProcessStartedEvent event);
  void beforeProcessCompleted(ProcessCompletedEvent event);
  void afterProcessCompleted(ProcessCompletedEvent event);
  void beforeNodeTriggered(ProcessNodeTriggeredEvent event);
  void afterNodeTriggered(ProcessNodeTriggeredEvent event);
  void beforeNodeLeft(ProcessNodeLeftEvent event);
  void afterNodeLeft(ProcessNodeLeftEvent event);

}
```

An audit log can be created based on the information provided by these process listeners. We provide various default logger implementations:

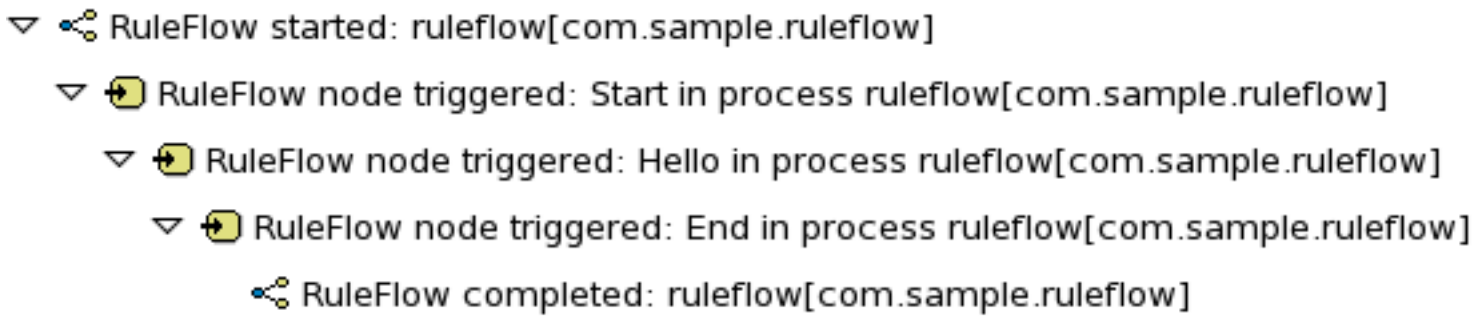
1. Console logger: This logger writes out all the events to the console.
2. File logger: This logger writes out all the events to a file using an XML representation. This log file can then for example be used in the IDE to generate a tree-based visualization of the events that occured during execution.
3. Threaded file logger: Because a file logger only writes the events to disk when closing the logger (or when the number of events in the logger reaches a predefined level), it cannot be used when debugging processes at runtime. A threaded file logger writes out the events to file at a specified time interval, making it possible to use the logger for example to visualize the progress when debugging processes in realtime.

The KnowledgeRuntimeLoggerFactory can be used to easily add a logger to your session, as shown below. When creating a console logger, the knowledge session for which the logger needs

to be created needs to be passes as an argument. The file logger also requires the name of the log file to be created, and the threaded file logger requires the interval (in milliseconds) after which the events should be saved.

```
KnowledgeRuntimeLogger logger =
 KnowledgeRuntimeLoggerFactory.newFileLogger(ksession, "test");
// add invocations to the process engine here, e.g.
 ksession.startProcess(processId);
...
logger.close();
```

The log file can be opened in the Eclipse when using the Audit View in the Drools Eclipse plugin, where the events are visualized in a tree-based manner (events that occur between the before and after event are shown as children of that event). The following screenshot shows a simple example, where a process is started, resulting in the triggering of the start node, an action node and an end node, after which the process was completed.



Chapter 5. Persistence

Drools Flow allows the persistent storage of the different kinds of information, i.e. the process runtime state, the process definitions and the history information.

5.1. Runtime state

Whenever a process is started, a process instance is created, which represents the execution of the process in that specific context. For example, when executing a process that specifies how to process a sales order, one process instance is created for each sales request. The process instance represents the current execution state in that specific context, and contains all the information related to that process instance. Note that it only contains the minimal runtime state that is needed to continue the execution of that process instance at some later time (and for example does not include information about the history of that process instance if that information is no longer needed in the process instance at some later stage, see the history log).

The runtime state of an executing process can be persisted, for example in a database. This allows for example to restore the state of execution of all running processes in case of unexpected failure, or to temporarily remove running instances from memory when they are no longer needed and restore them at some later time. Drools Flow allows you to plug in different persistence strategies. By default (if you do not configure the process engine otherwise), process instances are not persisted.

5.1.1. Binary persistence

But Drools Flow provides a binary persistence mechanism that allows you to persist the state of a process instance as a binary blob. This way, the state of all running process instances can always be stored in a persistent location. Note that these binary blobs usually are relatively small, as they only contain the minimal execution state of the process instance. For a simple process instance, this usually contains one or a few node instances (i.e. a node that is currently executing) and possibly some variable values.

5.1.2. Safe points

The state of a process instance is stored at so-called "safe points" during the execution of the process engine. Whenever a process instance is executed (for example by starting it, or by continuing after receiving a trigger it was waiting for), the engine continues execution of the process instance until no more actions can be performed. At that point, the engine has reached the next safe state and the state of the process instance (and all other process instances that might have been affected) is stored persistently.

5.1.3. Configuring persistence

By default, the engine does automatically persist the runtime data. It is however pretty straightforward to configure the engine to do this, by adding a config file and the necessary

dependencies. The persistence itself is based on the Java Persistence API (JPA) and can thus work with several persistence mechanisms. We are using Hibernate by default (but feel free to check out other alternatives). We're using the H2 database underneath to store the data (but again you're free to choose your own alternative).

First of all, you need to add the necessary dependencies to your classpath. If you're using the Eclipse IDE, you can do that by adding the jars to your Drools runtime directory (see the chapter on the Eclipse IDE), or by manually adding these dependencies to your project. First of all you need the drools-process-enterprise jar, as that contains the necessary code to persist the runtime state whenever necessary. Next, you also need various other dependencies, but these are different depending on the persistence solution and database you are using. In our case (using Hibernate and the H2 database), the following list of dependencies is needed:

1. drools-process-enterprise (org.drools)
2. persistence-api-1.0.jar (javax.persistence)
3. hibernate-entitymanager-3.4.0.GA.jar (org.hibernate)
4. hibernate-annotations-3.4.0.GA.jar (org.hibernate)
5. hibernate-commons-annotations-3.1.0.GA.jar (org.hibernate)
6. hibernate-core-3.3.0.SP1.jar (org.hibernate)
7. dom4j-1.6.1.jar (dom4j)
8. jta-1.0.1B.jar (javax.transaction)
9. javassist-3.4.GA.jar (javassist)
10. slf4j-api-1.5.2.jar (org.slf4j)
11. slf4j-jdk14-1.5.2.jar (org.slf4j)
12. h2-1.0.77.jar (com.h2database)
13. commons-collections-3.2.jar (commons-collections)
14. antlr-2.7.6.jar (antlr)

Next, you need to configure the Drools engine to persist the state of the engine whenever necessary. You can do this by simply specifying this in your session configuration. There are various ways to do this, but using a simple drools.session.conf file in a META-INF directory on your classpath is probably the easiest way (check the documentation for other ways of configuring your session, for example by providing a KnowledgeSessionConfiguration when first creating your session). In this config file you need to do two things: tell the engine that it needs to use a command service underneath (as commands are used to determine safe points during the execution of the engine), and use the JPA-based implementations of 3 internal components (the process instance manager, the work item manager and the signal manager), as these components will then be

able to look up the necessary information from persistence using JPA. The drools.session.conf file should thus look like this:

```
drools.commandService =
 org.drools.persistence.session.SingleSessionCommandService
drools.processInstanceManagerFactory =
 org.drools.persistence.processinstance.JPAProcessInstanceManagerFactory
drools.workItemManagerFactory =
 org.drools.persistence.processinstance.JPAWorkItemManagerFactory
drools.processSignalManagerFactory =
 org.drools.persistence.processinstance.JPASignalManagerFactory
```

By default, the drools-process-enterprise jar contains a configuration file that configures JPA to use hibernate and the H2 database, called persistence.xml in the META-INF directory, as shown below. You will need to override these if you want to change the default. We refer to the JPA and Hibernate documentation for more information on how to do this.

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<persistence
  version="1.0"
  xsi:schemaLocation=
    "http://java.sun.com/xml/ns/persistence
     http://java.sun.com/xml/ns/persistence/persistence_1_0.xsd
     http://java.sun.com/xml/ns/persistence/orm
     http://java.sun.com/xml/ns/persistence/orm_1_0.xsd"
  xmlns:orm="http://java.sun.com/xml/ns/persistence/orm"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns="http://java.sun.com/xml/ns/persistence">

  <persistence-unit name="org.drools.persistence.jpa">
    <provider>org.hibernate.ejb.HibernatePersistence</provider>
    <class>org.drools.persistence.jpa.ByteArrayObject</class>

    <properties>
      <property name="hibernate.dialect"
 value="org.hibernate.dialect.H2Dialect"/>
      <property name="hibernate.connection.driver_class"
 value="org.h2.Driver"/>
      <property name="hibernate.connection.url" value="jdbc:h2:mem:mydb" />
      <property name="hibernate.connection.username" value="sa"/>
      <property name="hibernate.connection.password" value="sasa"/>
      <property name="hibernate.connection.autocommit" value="false" />

      <property name="hibernate.max_fetch_depth" value="3"/>
      <property name="hibernate.hbm2ddl.auto" value="update" />
      <property name="hibernate.show_sql" value="false" />
    </properties>
```

```
  </persistence-unit>
</persistence>
```

After adding the necessary dependencies and the configuration file to your project, you can simply use the StatelessKnowledgeSession just the way you used to do. The engine will underneath translate your invocations to commands that will persist the state of the engine after each successful execution of a command. For example, the following code snippet shows how to create a session and start a process. Note that this snippet does not show anything about persistence (as the config file configures all that for you, and the engine takes care of it automatically), so the example looks just like normal Drools code. You can however always destroy your session and create a new one (or one session could continue the work that was started in another session), as the runtime state is persisted safely in a database, and can be retrieved whenever necessary.

```
 StatefulKnowledgeSession session = kbase.newStatefulKnowledgeSession();
 long processInstanceId =
session.startProcess("org.drools.test.TestProcess").getId();
 session.dispose();
```

Note that we only persist the minimal state that is needed to continue execution of the process instance at some later point. This for example means that it does not contain information about already executed nodes if that information is no longer relevant, or that process instances that have been completed or aborted are removed from the database. If you however want to search for history-related information, you should use the history log, as explained later.

5.2. Process definitions

Process definitions are usually stored in an XML format. These files can easily be stored on a file system during development. However, whenever you want to make your knowledge accessible to one or more engines in production, we recommend using a knowledge repository that (logically) centralizes your knowledge (in one or more knowledge repositories).

Guvnor is a sub-project that provides exactly that: it consists of a repository for storing different kinds of knowledge (not only process definitions but also rules, object models, etc.), allows easy retrieval of this knowledge (for example using WebDAV or by using a rule agent that automatically downloads the information from Guvnor when creating a knowledge base), and provides a web application that allows business users to view and possibly update the information in the knowledge repository. Check out the Drools Guvnor documentation for more information on how to do this.

5.3. History log

In many cases it is useful (if not necessary) to store information about the execution on process instances, so that this information can be used afterwards for example to verify what actions have been executed for a particular process instance, or to monitor and/or analyze the efficiency of a particular process, etc. Storing history information in the runtime database is usually not a

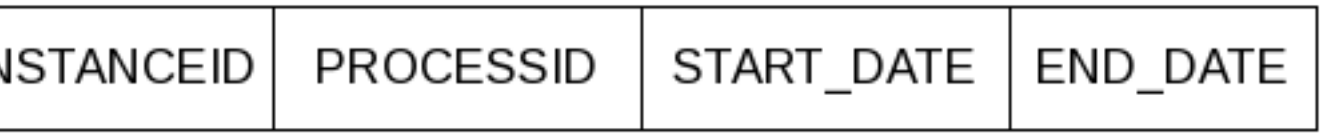
good idea (as this would result in ever-growing runtime data, and monitoring and analysis queries might influence the performance of your runtime engine). That is why history information about the execution of process instances is stored separately.

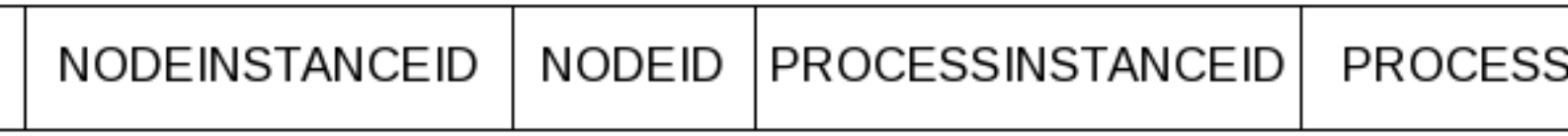
This history log of execution information is created based on the events generated by the process engine during execution. The Drools runtime engine provides a generic mechanism to listen to different kinds of events. The necessary information can easily be extracted from these events and persisted, for example in a database. Filters can be used to only store the information you find relevant.

5.3.1. Persisting process events in a database

The drools-bam module contains an event listener that stores process-related information in a database (using hibernate). The database contains two tables, one for process instance information and one for node instance information (see figure below):

1. *ProcessInstanceLog:* This lists the process instance id, the process (definition) id, the start date and (if applicable) the end date of all process instances.
2. *NodeInstanceLog:* This table contains more detailed information about which nodes were actually executed inside each process instance. Whenever a node instance is entered (from one of its incomming connections) or is exited (through one of its outgoing connections), that information is stored in this table. It therefore stores the process instance id and the process id (of the process instance it is being executed in), and the node instance id and corresponding node id (in the process definition) of the node instance in question. Finally, the type of event (0 = enter, 1 = exit) and the date of the event is stored as well.





To log process history information in a database like this, you need to register the logger on your session (or working memory) like this:

```
StatefulKnowledgeSession session = ...
new WorkingMemoryDbLogger(session);
```

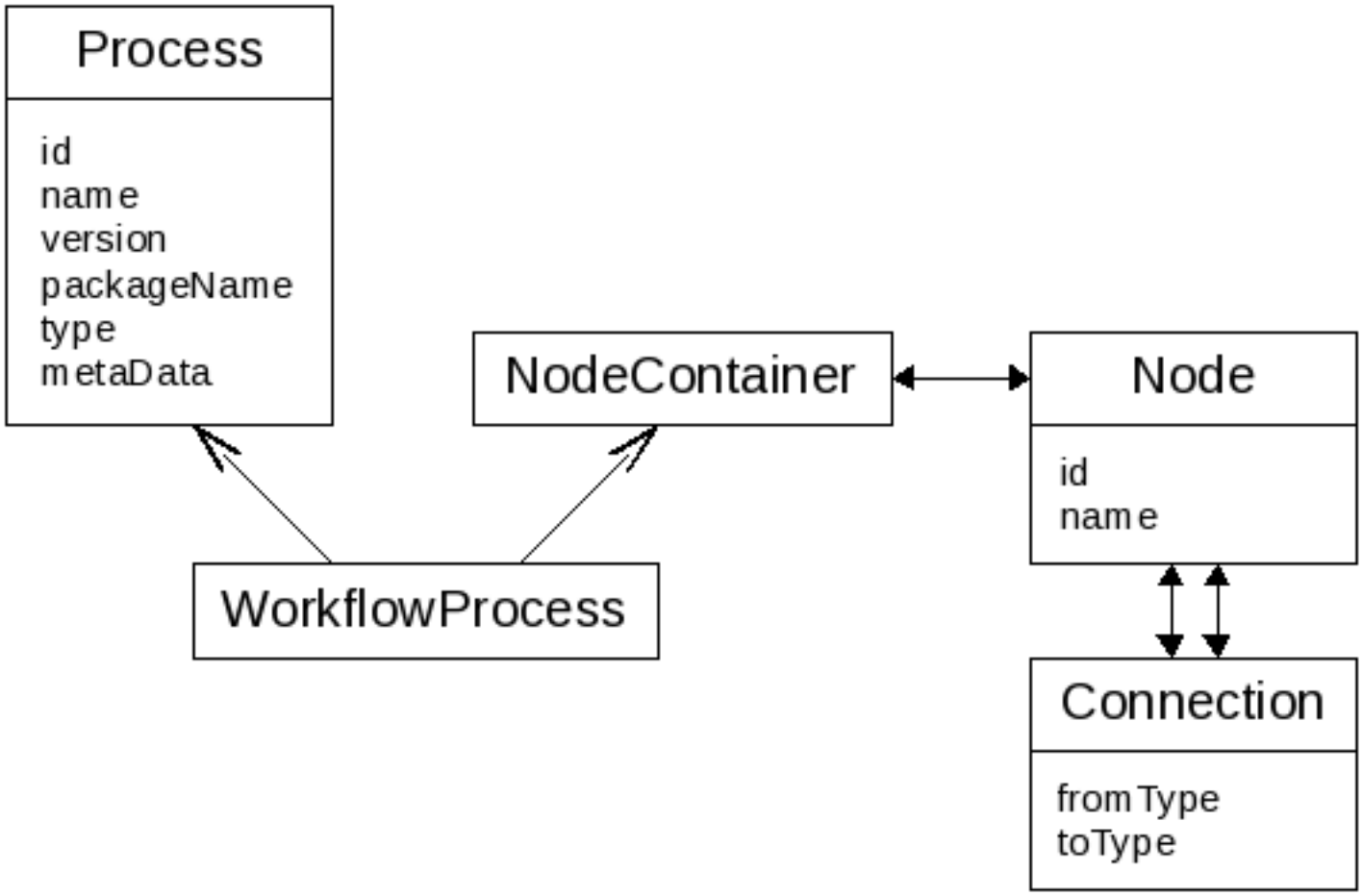
Note that this logger is just a logger like any other audit logger. This means you can add one or more filters using the addFilter method to make sure that only relevant information is stored in

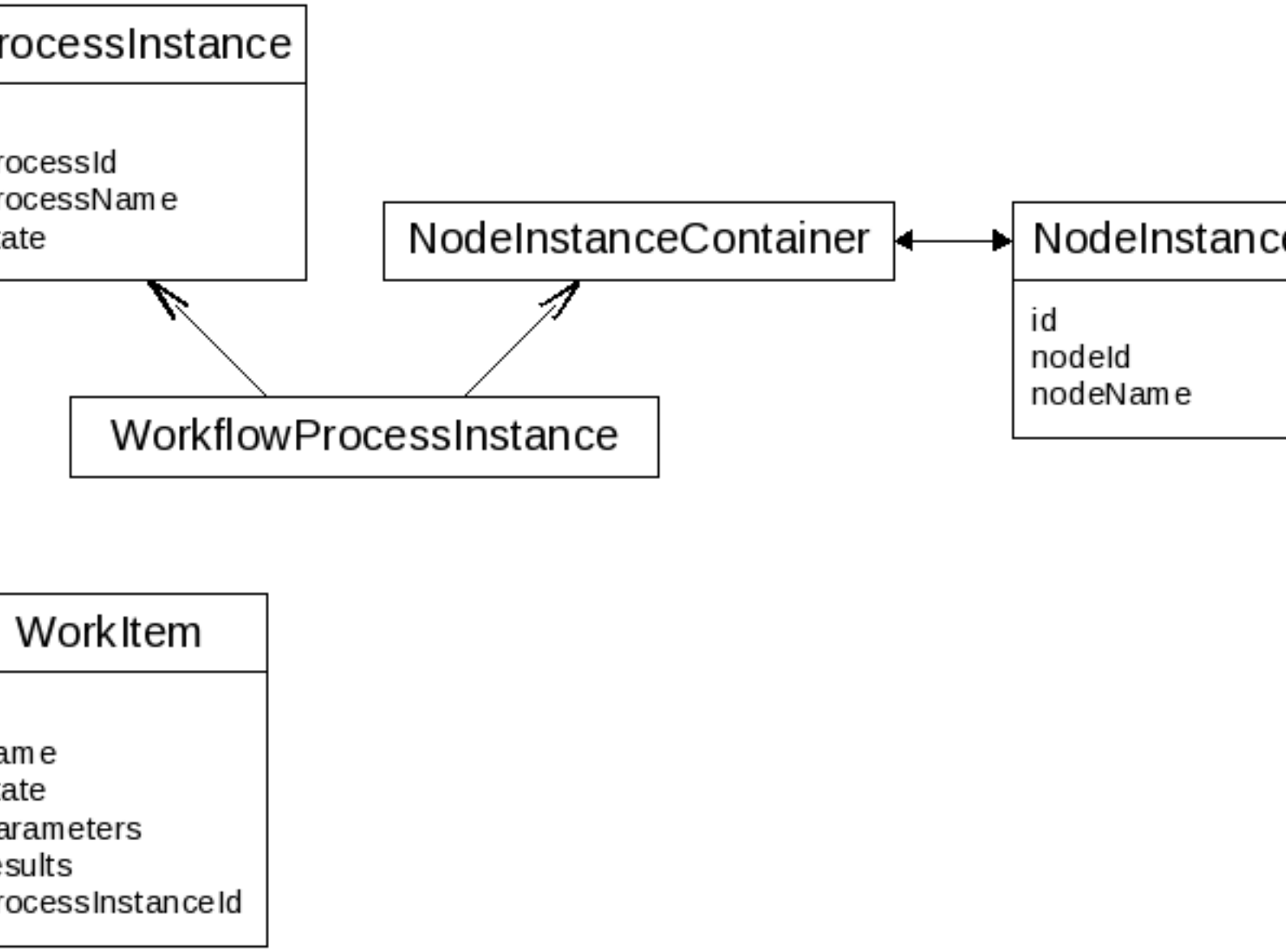
the database. If you use more than one filter, only information that is accepted by all your filters will be persisted in the database.

To specify the database where the information should be stored, modify the hibernate.cfg.xml file. By default, it uses an in memory database (H2). Check out the hibernate documentation if you do not know how to do this.

All this information can easily be queried and can be used in a lot of different use cases, ranging from creating a history log for one specific process instance to analyzing the performance of all instances of a specific process. The org.drools.process.auditProcessInstanceDbLog class shows some examples on how to retrieve all process instances, one specific process instance (by id), all process instances for one specific process, all node instances of a specific process instance, etc. You can of course easily create your own hibernate queries or access the information in the database directly.

Chapter 6. Drools Flow process model




rocessInstance
rocessId
rocessName
tate
NodeInstanceContainer
NodeInstanc
id
nodeId
nodeName
WorkflowProcessInstance
WorkItem
ame
tate
arameters
esults
rocessInstanceId

Chapter 7. Rules and Processes

Drools Flow is a workflow and process engine that allows advanced integration of processes and rules. This chapter gives an overview of how to integrate rules and processes, ranging from simple to advanced.

7.1. Why use rules in processes?

Workflow languages that depend purely on process constructs (like nodes and connections) to describe the business logic of (a part of) an application tend to be quite complex. While these workflow constructs are very well suited to describe the overall control flow of an application, it can be very difficult to describe complex logic and exceptional situations. Therefore, executable processes tend to become very complex. We believe that, by extending a process engine with support for declarative rules in combination with these regular process constructs, this complexity can be kept under control.

1. Simplicity: Complex decisions are usually easier to specify using a set of rules. Rules can describe complex business logic more easily by using an advanced constraint language. Multiple rules can be combined, each describing a part of the business logic.

2. Agility: Rules and processes can have a separate life cycle. This means that for example we can change the rules describing some crucial decision points without having to change the process itself. Rules can be added, removed or modified to fine-tune the behaviour of the process to the constantly evolving requirements and environment.

3. Different scope: Rules can be reused accross processes or outside processes. Therefore, your business logic is not locked inside your processes.

4. Declarative: Focus on describing 'what' instead of 'how'.

5. Granularity: It is easy to write simple rules that handle specific circumstances. Processes more suited to describe the overall control flow but tend to become very complex if they also need to describe a lot of exceptional situations.

6. Data-centric: Rules can easily handle large data sets.

7. Performance: Rule evaluation is optimized.

8. Advanced condition and action language: Rule languages supports advanced features like custom functions, collections, not, exists, for all, etc.

9. Higher-level: Using DSLs, business editors, decision tables, decision trees, etc. your business logic could be described in a way that can be understood (and possibly even modified) by business users.

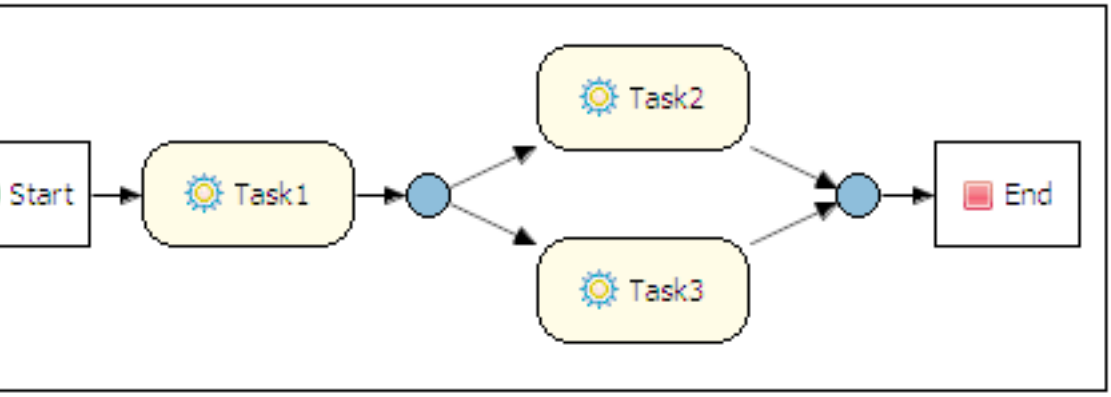
7.2. Why integrate rules and processes in a single engine?

Drools Flow combines a process and a rules engine in one software product. This offers several advantages (compared to trying to loosely coupling an existing process and rules product):

1. Simplicity: Easier for end user to combine both rules and processes.
2. Encapsulation: Sometimes close integration between processes and rules is beneficial.
3. Performance: No unnecessary passing, transformation or synchronization of data
4. Learning curve: Easier to learn one product.
5. Manageability: Easier to manage one product, rules and processes can be similar artefacts in a larger knowledge repository.
6. Integration of features: We provide an integrated IDE, audit log, web-based management platform, repository, debugging, etc.

7.3. Approach

Workflow languages describe the order in which activities should be performed using a flow chart. A process engine is responsible for selecting which activities should be executed based on the current state of the executing processes. Rules on the other hand are composed of a set of conditions that describe when a rule is applicable and an action that is executed when the rule is applicable. The rules engine is then responsible for evaluating and executing the rules. It decides which rules need to be executed based on the current state of the application.




```
rule "RuleName"
    when
        // condition
    then
        // actions
end
```

Workflow processes are very good at describing the overall control flow of (possibly long-running) applications. However, processes that are used to define complex business decisions or contain a lot of exceptional situations or need to respond to various external events tend to become very complex. Rules on the other hand are very good at describing complex decisions and reasoning about large amounts of data or events. It is however not trivial to define long-running processes using rules.

In the past, users were forced to choose between defining their business logic using either a process or rules engine. Problems that required complex reasoning about large amounts of data used a rules engine, while users that wanted to focus on describing the control flow of their processes were forced to use a process engine. However, businesses nowadays might want to combine both processes and rules in order to be able to define all their business logic in the format that best suits their needs.

Basically, both a rules and a process engine will derive the next steps that need to be executed by looking at its knowledge base (a set of rules or processes respectively) and the current known state of the application (the data in the working memory or the state of the executing process instances respectively). If we want to integrate rules and processes, we need an engine that can decide the next steps taking into account the logic that is defined inside both the processes and the rules.

7.3.1. Teaching a rules engine about processes

It is very difficult (and probably very inefficient as well) to extend a process engine to also take rules into account: the process engine would need to check for rules that might need to be executed at every step and would have to keep the data that is used by the rules engine up to date. However, it is not that difficult to 'teach' a rules engine about processes. If the current state of the processes is also inserted as part of the data the rules engine reasons about, and we 'learn' the rules engine how to derive the next steps of an executing process, the rules engine will then be able to derive the next steps taking both rules and processes into account.

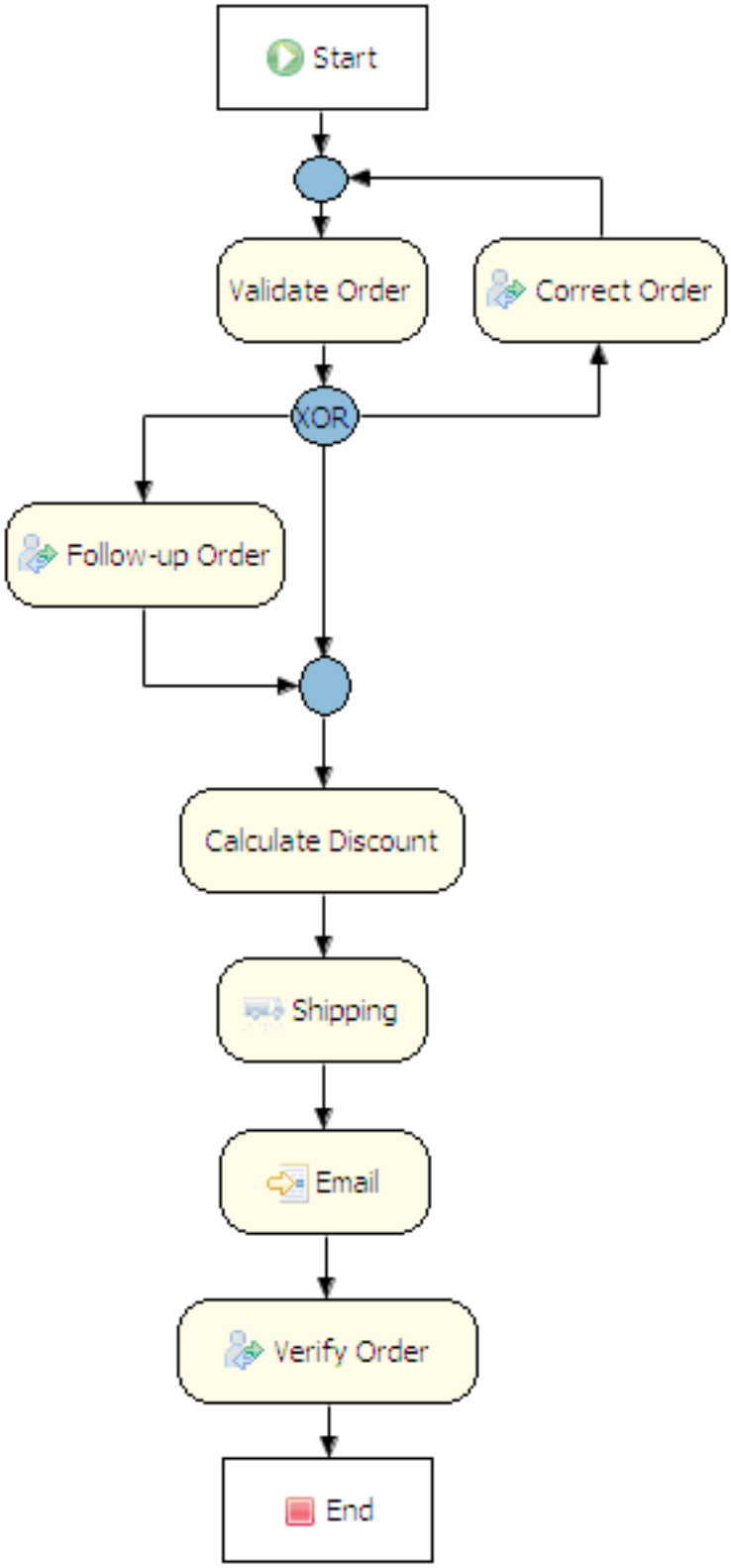
7.3.2. Inversion of control

From the process perspective, this means that there is an inversion of control. In a normal proces engine, the engine is in full control and derives the next steps based on the current state of the process instance. If needed, it can contact external services to retrieve additional information (e.g. invoke a rules engine to request a decision), but it solely decides which steps to take, and is responsible for executing these steps.

However, only our extended rules engine (that can reason about both rules and processes) is capable of deriving the next steps taking both rules and processes into account. If a part of the process needs to be executed, the rules engine will request the process engine to execute this step. Once this step has been performed, the process engine returns control to the rules engine to again derive the next steps. This means that the control on what to do next has been inverted: the process engine itself no longer decides the next step to take but our extended rules engine will be in control, notifying the process engine when to execute the next step.

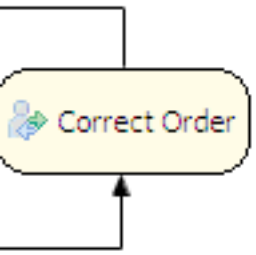
7.4. Example

The drools-examples project contains a sample process (org.drools.examples.process.order) that illustrates some of the advantages of being able to combine processes and rules. This process describes an order application where incoming orders are validated, possible discount are calculated and shipping of the goods is requested.

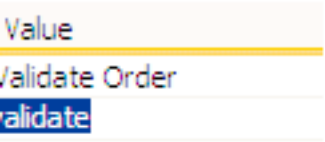

Start
Validate Order
Correct Order
XOR
Follow-up Order
Calculate Discount
Shipping
Email
Verify Order
End

7.4.1. Evaluating a set of rules in your process

Drools Flow can easily include a set of rules as part of the process. The rules that need to be evaluated should be grouped in a ruleflow group (using the ruleflow-group rule attribute) and a RuleSet node can be used to trigger the evaluation of these rules in your process. This example uses two RuleSet nodes in the process: one for the validation of the order and one for calculating the discount. For example, one of the rules for validiting an order looks like this (note the ruleflow-group attribute, which makes sure that this rule is evaluated as part of the RuleSet node with the same ruleflow-group):

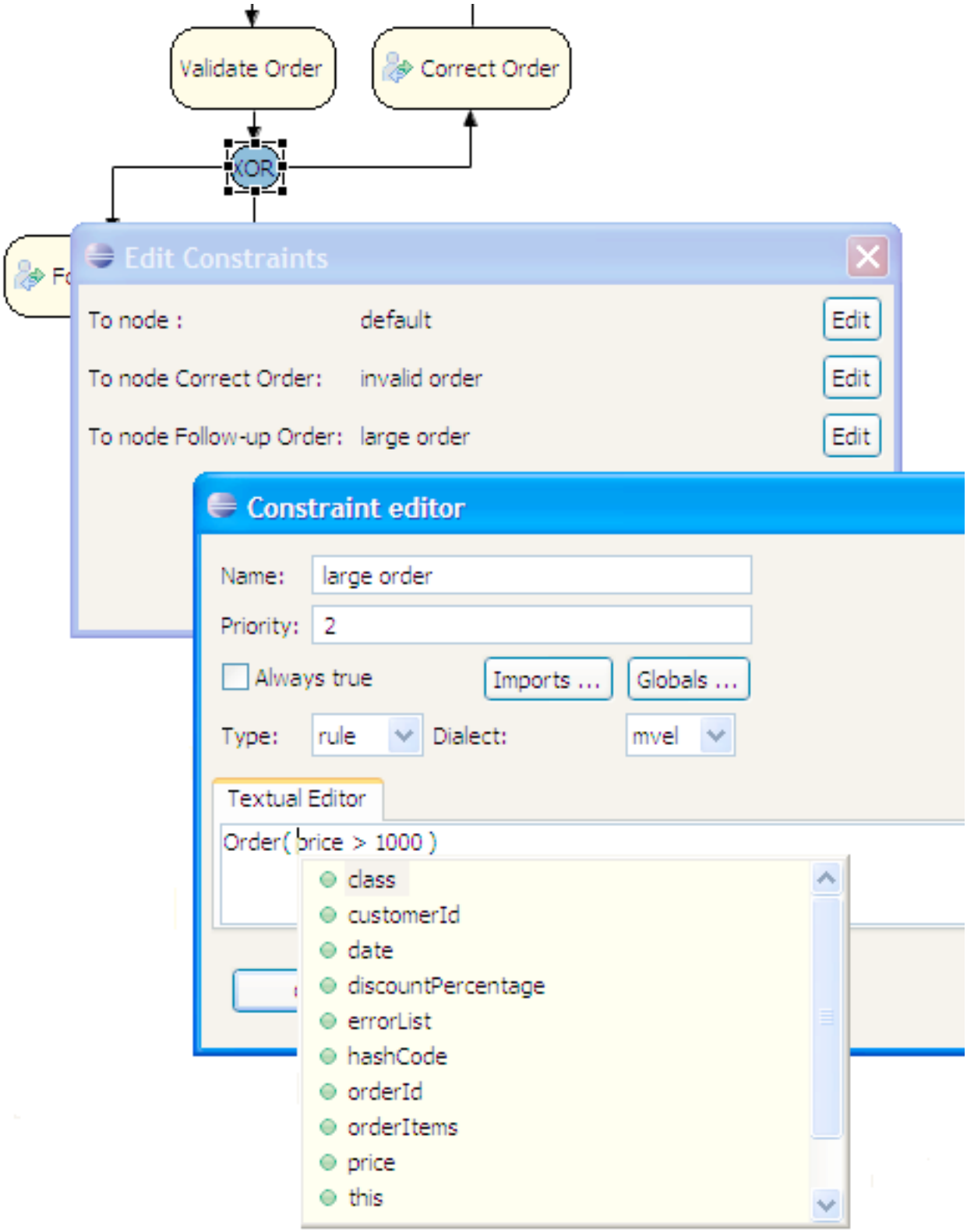



```
rule "Invalid item id" ruleflow-group "validate" lock-o
    when
        o: Order( )
        i: Order.OrderItem( ) from o.getOrderItems()
        not (Item( ) from itemCatalog.getItem(i.getItem
    then
        System.err.println("Invalid item id found!");
        o.addError("Invalid item id " + i.getItemId());
end
```



7.4.2. Using rules for evaluating constraints

Rules can be used for expressing and evaluating complex constraints in your process. For example, when a decision should be made which execution paths should be selected at a split, rules could be used to define these conditions. Similarly, a wait state could use a rule to define how long to wait. This example uses rules for deciding the next action after validating the order. If the order contains errors, a sales representative should try to correct the order. Orders with a value > 1000$ are more important and a senior sales representative should follow up the order. All other orders should just proceed normally. An decision node is used to select one of these alternatives, and rules are used to describe the constraints for each of them:


Validate Order
Correct Order
XOR
Edit Constraints
To node :
default
Edit
To node Correct Order:
invalid order
Edit
To node Follow-up Order:
large order
Edit
Constraint editor
Name:
large order
Priority:
2
Always true
Imports ...
Globals ...
Type:
rule
Dialect:
mvel
Textual Editor
Order(price > 1000)
class
customerId
date
discountPercentage
errorList
hashCode
orderId
orderItems
price
this

7.4.3. Assignment rules

Human tasks can be used in a process to describe work that needs to be executed by a human actor. Which actor could be based on the current state of the process, and the history. Assignment rules can be used to describe how to the actor based on this information. These assignment rules will then be applied automatically whenever a new human task needs to be executed.

```
*** Generic assignment rules **********/

sign 'Correct Order' to any sales representative" salience 30

There is a human task
- with task name "Correct Order"
- without actor id

Set actor id "Sales Representative"


*** Assignment rules for the RuleSetExample process **********/

sign 'Follow-up Order' to a senior sales representative" salience 40

Process "org.drools.examples.process.ruleset.RuleSetExample" contains a hu
- with task name "Follow-up Order"
- without actor id

Set actor id "Senior Sales Representative"
```

7.4.4. Describing exceptional situations using rules

Rules can be used for describing exceptional situations and how to respond to these situations. Adding all this information in the control flow of the main process would make the basic process much more complex. Rules can be used to handle each of these situations separately, without making the core process more complex. It also makes it much easier to adapt existing processes to take new unanticipated events into account.

7.4.5. Modularizing concerns using rules

The process defines the overall control flow. Rules could be used to add additional concerns to this process without making the overall control flow more complex. For example, rules could be

defined to log certain information during the execution of the process. The original process is not altered and all logging functionality is cleanly modularized as a set of rules. This greatly improves reusability (allows users to easily apply the same strategy on different processes), readability (control flow of the original process is still the same) and modifiability (you can easily add, remove or change the logging strategy by adding, removing or changing the rules, the process should not be modified).

7.4.6. Using rules to dynamically alter the behaviour of the process

Rules can be used to dynamically fine-tune the behaviour of your processes. For example, if a problem is encountered at runtime with one of the processes, new rules could be added at runtime to log additional information or handle specific cases of the process. Once the problem is solved or the circumstances have changed, these rules can easily be removed again. Based on the current status, different strategies could be selected dynamically. For example, based on the current load of all the services, rules could be used to optimize the process to the current load. This process contains a simple example that allows you to dynamically add or remove logging for the 'Check Order' task. When the 'Debugging output' checkbox in the main application window is checked, a rule is dynamically loaded to add a logging statement to the console whenever the 'Check Order' task is requested. Unchecking the box will dynamically remove the rule again.

```
 'Correct Order'" salience 25

ce: WorkItemNodeInstance( workItemId <= 0, node.name == "Correct Order" )
Impl( state == WorkItemImpl.PENDING ) from workItemNodeInstance.getWorkItem

nce variableScopeInstance =
Instance) workItemNodeInstance.getProcessInstance().getContextInstance(Vari
("LOGGING: Requesting the correction of " + variableScopeInstance.getVariab
```

7.4.7. Integrated tooling

Processes and rules are integrated in the Drools Eclipse IDE. Both processes and rules are simply considered as different types of business logic, but are managed almost identical. For example, loading a process or a set of rules into the engine is very similar:

```
private static RuleBase createKnowledgeBase() throws Exception {
    PackageBuilder builder = new PackageBuilder();
    Reader source = new InputStreamReader(
        OrderExample.class.getResourceAsStream("RuleSetExample.rf"));
    builder.addProcessFromXml(source);
    source = new InputStreamReader(
        OrderExample.class.getResourceAsStream("workflow_rules.drl"));
    builder.addPackageFromDrl(source);
```

Our audit log also contains an integrated view, showing how rules and processes are influencing each other. For example, a part of the log shows how the '5% discount' rule is executed as part of the 'Calculate Discount' node.

```
wGroup deactivated: validate[size=0]
Flow node triggered: XOR in process RuleSetExample[org.drools.examples.process.ruleset.RuleSetExample]
RuleFlow node triggered:  in process RuleSetExample[org.drools.examples.process.ruleset.RuleSetExample]
RuleFlow node triggered: Calculate Discount in process RuleSetExample[org.drools.examples.process.ruleset.RuleSetEx
  RuleFlowGroup activated: discount[size=1]
on executed: Rule 5% discount if order includes laptop and order after 18h o=Order Order-1(1); date=Wed Jul 02 21:30:30
wGroup deactivated: discount[size=0]
```

7.4.8. Domain-specific rules and processes

Rules do not need to be defined using the core rule language syntax, but they also can be defined using our more advanced rule editors like domain-specific languages, decision tables, guided editors, etc. Our examples defines a domain-specific language for describing assignment rules, based on the type of task, its properties, the process it is defined in, etc. This makes the assignment rules much more understandable for non-experts.

```
*** Generic assignment rules **********/

sign 'Correct Order' to any sales representative" salience 30

There is a human task
- with task name "Correct Order"
- without actor id

Set actor id "Sales Representative"


*** Assignment rules for the RuleSetExample process **********/

sign 'Follow-up Order' to a senior sales representative" salience 40

Process "org.drools.examples.process.ruleset.RuleSetExample" contains a hu
- with task name "Follow-up Order"
- without actor id

Set actor id "Senior Sales Representative"
```

Chapter 8. Domain-specific processes

8.1. Introduction

One of the goals of our unified rules and processes framework is to allow users to extend the default programming constructs with domain-specific extensions that simplify development in a particular application domain. While Drools has been offering constructs to create domain-specific rule languages for some time now, this tutorial describes our first steps towards domain-specific process languages.

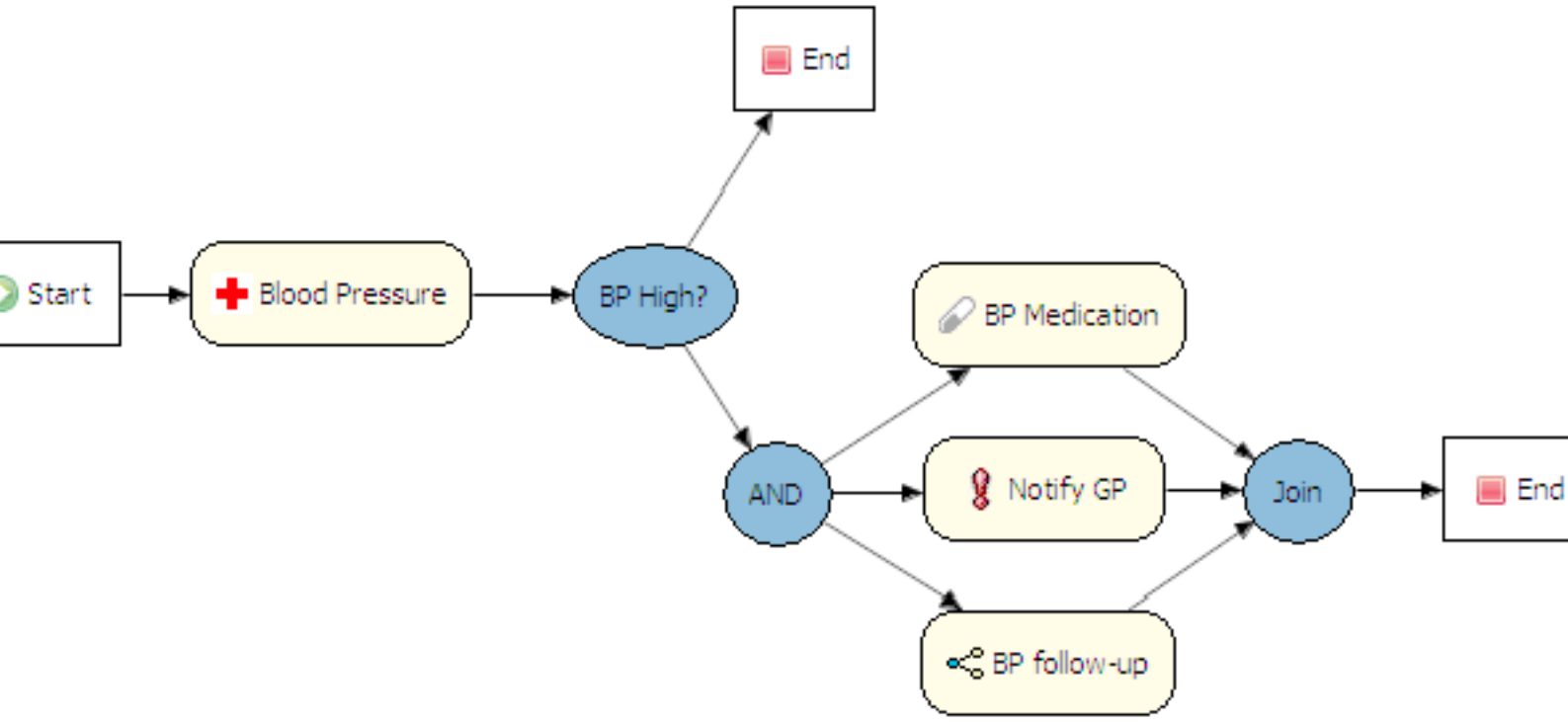
Most process languages offer some generic action (node) construct that allows plugging in custum user actions. However, these actions are usually low-level, where the user is required to write custom code to implement the work that should be incorporated in the process. The code is also closely linked to a specific target environment, making it difficult to reuse the process in different contexts.

Domain-specific languages are targeted to one particular application domain and therefore can offer constructs that are closely related to the problem the user is trying to solve. This makes the processes and easier to understand and self-documenting. We will show you how to define domain-specific work items, which represent atomic units of work that need to be executed. These work items specify the work that should be executed in the context of a process in a declarative manner, i.e. specifying what should be executed (and not how) on a higher level (no code) and hiding implementation details.

So we want work items that are:

1. domain-specific
2. declarative (what, not how)
3. high-level (no code)
4. customizable to the context

Users can easily define their own set of domain-specific work items and integrate them in our process language(s). For example, the next figure shows an example of a process in a healthcare context. The process includes domain-specific work items for ordering nursing tasks (e.g. measuring blood pressure), prescribing medication and notifying care providers.



8.2. Example: Notifications

Let's start by showing you how to include a simple work item for sending notifications. A work item represent an atomic unit of work in a declarative way. It is defined by a unique name and additional parameters that can be used to describe the work in more detail. Work items can also return information after they have been executed, specified as results. Our notification work item could thus be defined using a work definition with four parameters and no results:

```
Name: "Notification"
Parameters
From [String]
To [String]
Message [String]
Priority [String]
```

8.2.1. Creating the work definition

All work definitions must be specified in one or more configuration files in the project classpath, where all the properties are specified as name-value pairs. Parameters and results are maps where each parameter name is also mapped to the expected data type. Note that this configuration file also includes some additional user interface information, like the icon and the display name of the work item. (We use MVEL for reading in the configuration file, which allows us to do more advanced configuration files). Our MyWorkDefinitions.conf file looks like this:

```
import org.drools.process.core.datatype.impl.type.StringDataType;
[
  // the Notification work item
  [
    "name" : "Notification",
    "parameters" : [
      "Message" : new StringDataType(),
      "From" : new StringDataType(),
      "To" : new StringDataType(),
      "Priority" : new StringDataType(),
    ],
    "displayName" : "Notification",
    "icon" : "icons/notification.gif"
  ]

  // add more work items here ...
]
```

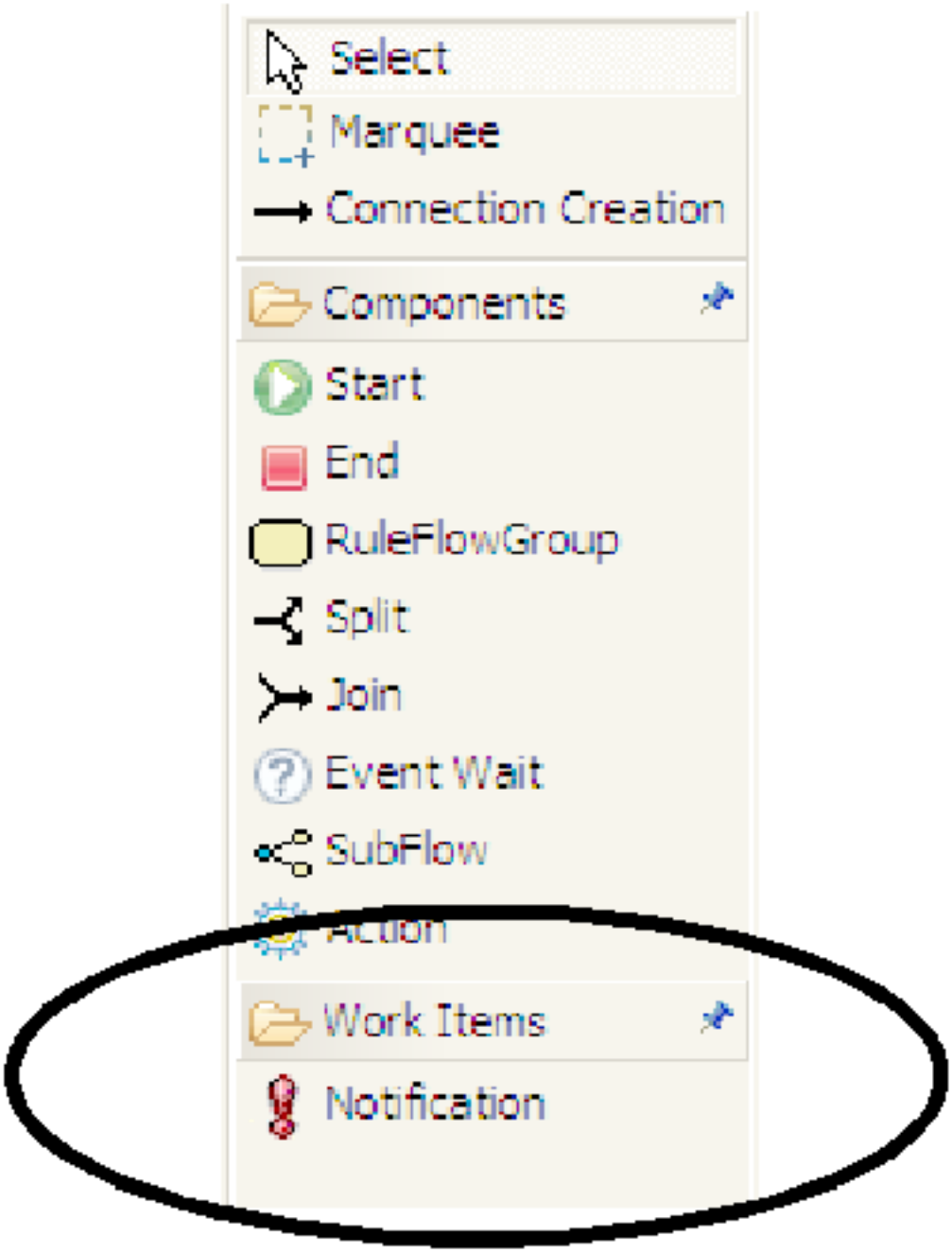
8.2.2. Registering the work definition

The Drools Configuration API can be used to register work definition files for your project using the drools.workDefinitions property, which represents a list of files containing work definitions (separated usings spaces). For example, include a drools.rulebase.conf file in the META-INF directory of your project and add the following line:

```
  drools.workDefinitions = MyWorkDefinitions.conf
```

8.2.3. Using your new work item in your processes

Once our work definition has been created and registered, we can start using it in our processes. The process editor contains a separate section in the palette where the different work items that have been defined for the project appear.



Using drag and drop, a notification node can be created inside your process. The properties can be filled in using the properties view.

Apart from the properties defined by for this work item, all work items also have these three properties:

1. Parameter Mapping: Allows you map the value of a variable in the process to a parameter of the work item. This allows you to customize the work item based on the current state of the actual process instance (for example, the priority of the notification could be dependent of some process-specific information).

2. Result Mapping: Allows you to map a result (returned once a work item has been executed) to a variable of the process. This allows you to use results in the remainder of the process.

3. Wait for completion: By default, the process waits until the requested work item has been completed before continuing with the process. It is also possible to continue immediately after the work item has been requested (and not waiting for the results) by setting "wait for completion" to false.

8.2.4. Executing work items

The Drools engine contains a WorkItemManager that is responsible for executing work items whenever necessary. The WorkItemManager is responsible for delegating the work items to WorkItemHandlers that execute the work item and notify the WorkItemManager when the work item has been completed. For executing notification work items, a NotificationWorkItemHandler should be created (implementing the WorkItemHandler interface):

```
package com.sample;

import org.drools.process.instance.WorkItem;
import org.drools.process.instance.WorkItemHandler;
import org.drools.process.instance.WorkItemManager;

public class NotificationWorkItemHandler implements WorkItemHandler {

  public void executeWorkItem(WorkItem workItem, WorkItemManager manager) {
    // extract parameters
    String from = (String) workItem.getParameter("From");
    String to = (String) workItem.getParameter("To");
    String message = (String) workItem.getParameter("Message");
    String priority = (String) workItem.getParameter("Priority");
    // send email
    EmailService service = ServiceRegistry.getInstance().getEmailService();
    service.sendEmail(from, to, "Notification", message);
    // notify manager that work item has been completed
    manager.completeWorkItem(workItem.getId(), null);
  }

  public void abortWorkItem(WorkItem workItem, WorkItemManager manager) {
    // Do nothing, notifications cannot be aborted
  }

}
```

This WorkItemHandler sends a notification as an email and then immediate notifies the WorkItemManager that the work item has been completed. Note that not all work items can be completed directly. In cases where executing a work item takes some time, execution can continue asynchronously and the work item manager can be notified later. In these situations, it might also

be possible that a work item is being aborted before it has been completed. The abort method can be used to specify how to abort such work items.

WorkItemHandlers should be registered at the WorkItemManager, using the following API:

```
workingMemory.getWorkItemManager().registerWorkItemHandler(
  "Notification", new NotificationWorkItemHandler());
```

Decoupling the execution of work items from the process itself has the following advantages:

1. The process is more declarative, specifying what should be executed, not how.
2. Changes to the environment can be implemented by adapting the work item handler. The process itself should not be changed. It is also possible to use the same process in different environments, where the work item handler is responsible for integrating with the right services.
3. It is easy to share work item handlers across processes and projects (which would be more difficult if the code would be embedded in the process itself).
4. Different work item handlers could be used depending on the context. For example, during testing or simulation, it might not be necessary to actually execute the work items. The next section shows an example of how to use specialized work item handlers during testing.

8.3. Testing processes using work items

Customizable execution depending on context, easier to manage changes in environment (by changing handler), sharing processes accross contexts (using different handlers), testing, simulation (custom test handlers)

8.4. Future

Our process framework is based on the (already well-known) idea of a Process Virtual Machine (PVM), where the process framework can be used as a basis for multiple process languages. This allows users to more easily create their own process languages, where common services provided by the process framework (e.g. persistence, audit) can be (re)used by the process language designer. Processes are represented as a graph of nodes, each node describing a part of the process logic. Different types of nodes are used for expressing different kinds of functionality, like creating or merging parallel flows (split and join), invoking a sub process, invoking external services, etc. One of our goals is creating a truly pluggable process language, where language designers can easily plug in their own node implementations.

Chapter 9. Human Tasks

An important aspect of work flow and BPM (business process management)is human task management. While some of the work performed in a process can be executed automatically, some tasks need to be executed with the interaction of human actors. Drools Flow supports the use of human tasks inside processes using a special human task node that will represent this interaction. This node allows process designers to define the type of task, the actor(s), the data associated with the task, etc. We also have implemented a task service that can be used to manage these human tasks. Users are however open to integrate any other solution if they want to, as this is fully pluggable.

To start using human tasks inside your processes, you first need to (1) include human task nodes inside your process, (2) integrate a task management component of your choice (e.g. the WS-HT implementation provided by us) and (3) have end users interact with the human task management component using some kind of user interface. These elements will be discussed in more detail in the next sections.

9.1. Human tasks inside processes

Drools Flow supports the use of human tasks inside processes using a special human task node (as shown in the figure above). A human task node represents an atomic task that needs to be executed by a human actor. Although Drools Flow has a special human task node for including human tasks inside a process, human tasks are simply considered as any other kind of external service that needs to be invoked and are therefore simply implemented as a special kind of work item. The only thing that is special about the human task node is that we have added support for swimlanes, making it easier to assign tasks to users (see below). A human task node contains the following properties:

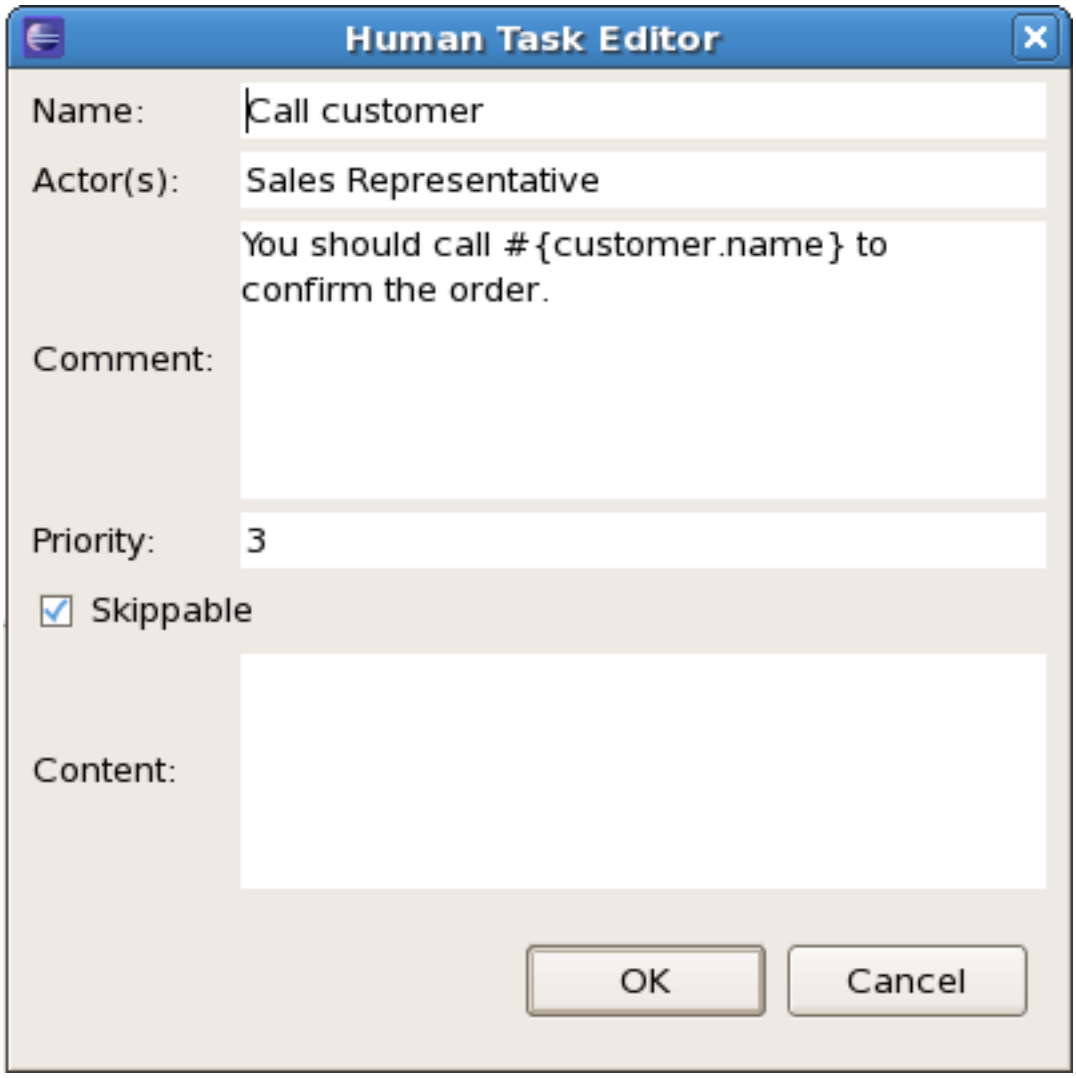
- *Id*: The id of the node (which is unique within one node container).
- *Name*: The display name of the node.
- *TaskName*: The name of the human task.
- *Priority*: An integer indicating the priority of the human task.
- *Comment*: A comment associated with the human task.
- *ActorId*: The actor id that is responsible for executing the human task. A list of actor id's can be specified using a comma (',') as separator.

- *Skippable*: Specifies whether the human task can be skipped (i.e. the actor decides not to execute the human task).
- *Content*: The data associated with this task.
- *Swimlane*: The swimlane this human task node is part of. Swimlanes make it easy to assign multiple human tasks to the same actor. See below for more detail on how to use swimlanes.
- *Wait for completion*: If this property is true, the human task node will only continue if the human task has been terminated (i.e. completed or any other terminal state); otherwise it will continue immediately after creating the human task.
- *On entry/exit actions*: Actions that are executed upon entry / exit of this node.
- *Parameter mapping*: Allows copying the value of process variables to parameters of the human task. Upon creation of the human tasks, the values will be copied.
- *Result mapping*: Allows copying the value of result parameters of the human task to a process variable. Upon completion of the human task, the values will be copied. Note that can only use result mappings when "Wait for completion" is set to true. A human task has a result variable "Result" that contains the data returned by the human actor. The variable "ActorId" contains the id of the actor that actually executed the task.
- *Timers*: Timers that are linked to this node (see the 'timers' section for more details).
- *ParentId*: Allows to specify the parent task id, in the case that this task is a sub task of another. (see the 'sub task' section for more details)

You can edit these variables in the properties view (see below) when selecting the human task node, or the most important properties can also be edited by double-clicking the human task node, after which a custom human task node editor is opened, as shown below as well.

Properties

Property	Value
ActorId	Sales Representative
Comment	You should call #{customer.name} to confirm the order.
Content	
Id	4
Name	Human Task
On Entry Actions	
On Exit Actions	
Parameter Mapping	{}
Priority	3
Result Mapping	{}
Skippable	true
Swimlane	
TaskName	Call customer
Timers	
Wait for completion	true



Note that you could either specify the values of the different parameters (actorId, priority, content, etc.) directly (in which case they will be the same for each execution of this process), or make them context-specific, based on the data inside the process instance. For example, parameters of type String can use #{expression} to embed a value in the String. The value will be retrieved when creating the work item and the #{...} will be replaced by the toString() value of the variable. The expression could simply be the name of a variable (in which case it will be resolved to the value of the variable), but more advanced MVEL expressions are possible as well, like #{person.name.firstname}. For example, when sending an email, the body of the email could contain something like "Dear #{customer.name}, ...". For other types of variables, it is possible to map the value of a variable to a parameter using the parameter mapping.

9.1.1. Swimlanes

Human task nodes can be used in combination with swimlanes to assign multiple human tasks to the similar actors. Tasks in the same swimlane will be assigned to the same actor. Whenever the first task in a swimlane is created, and that task has an actorId specified, that actorId will be

assigned to the swimlane as well. All other tasks that will be created in that swimlane will use that actorId as well, even if an actorId has been specified for the task as well.

Whenever a human task that is part of a swimlane is completed, the actorId of that swimlane is set to the actorId that executed that human task. This allows for example to assign a human task to a group of users, and to assign future tasks of that swimlame to the user that claimed the first task. This will also automatically change the assignment of tasks if at some point one of the tasks is reassigned to another user.

To add a human task to a swimlane, simply specify the name of the swimlane as the value of the "Swimlane" parameter of the human task node. A process must also define all the swimlanes that it contains. To do so, open the process properties by clicking on the background of the process and click on the "Swimlanes" property. You can add new swimlanes there.

9.2. Human task management component

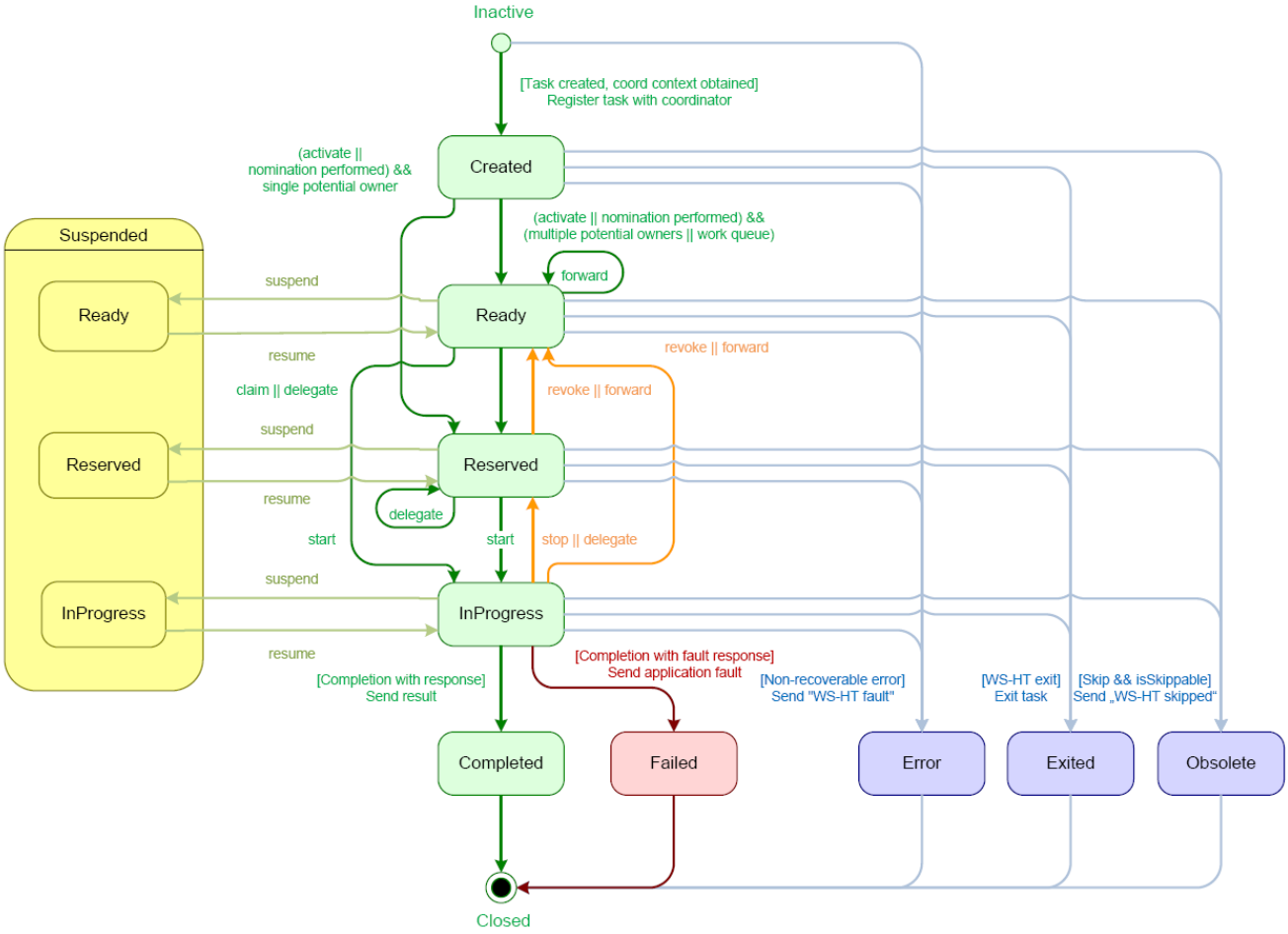
As far as the Drools Flow engine is concerned, human tasks are similar to any other external service that needs to be invoked and are implemented as an extension of normal work items. As a result, the process itself only contains an abstract description of the human tasks that need to be executed, and a work item handler is responsible for binding this abstract tasks to a specific implementation. Using our pluggable work item handler approach (see the chapter on domain-specific processes for more details), users can plug in any back-end implementation.

We do however provide an implementation of such a human task management component based on the WS-HumanTask specification. If you do not have the requirement to integrate a specific human task component yourself, you can use this service. It manages the task life cycle of the tasks (creation, claiming, completion, etc.) and stores the state of the task persistently. It also supports features like internationalization, calendar integration, different types of assignments, delegation, deadlines, etc.

Because we did not want to implement a custom solution when a standard is available, we chose to implement our service based on the WS-HumanTask (WS-HT) specification. This specification defines in detail the model of the tasks, the life cycle, and a lot of other features as the ones mentioned above. It is pretty comprehensive and can be found *here* [http://download.boulder.ibm.com/ibmdl/pub/software/dw/specs/ws-bpel4people/WS-HumanTask_v1.pdf].

9.2.1. Task life cycle

Looking from the perspective of the process, whenever a human task node is triggered during the execution of a process instance, a human task is created. The process will only continue from that point when that human task has been completed or aborted (unless of course you specify that the process does not need to wait for the human task to complete, by setting the "Wait for completion" property to true). However, the human task usually has a separate life cycle itself. We will now shortly introduce this life cycle, as shown in the figure below. For more details, check out the WS-HumanTask specification.



Whenever a task is created, it starts in the "Created" stage. It usually automatically transfers to the "Ready" state, at which point the task will show up on the task list of all the actors that are allowed to execute the task. There, it is waiting for one of these actors to claim the task, indicating that he or she will be executing the task. Once a user has claimed a task, the status is changed to "Reserved". Note that a task that only has one potential actor will automatically be assigned to that actor upon creation of that task. After claiming the task, that user can then at some point decide to start executing the task, in which case the task status is changed to "InProgress". Finally, once the task has been performed, the user must complete the task (and can specify the result data related to the task), in which case the status is changed to "Completed". If the task could not be completed, the user can also indicate this using a fault response (possibly with fault data associated), in which case the status is changed to "Failed".

The life cycle explained above is the normal life cycle. The service also allows a lot of other life cycle methods, like:

- Delegating or forwarding a task, in which case it is assigned to another actor
- Revoking a task, so it is no longer claimed by one specific actor but reappears on the task list of all potential actors
- Temporarily suspending and resuming a task

- Stopping a task in progress
- Skipping a task (if the task has been marked as skippable), in which case the task will not be executed

9.2.2. Linking the task component to the Drools Flow engine

The task management component needs to be integrated with the Drools Flow engine just like any other external service, by registering a work item handler that is responsible for translating the abstract work item (in this case a human task) to a specific invocation. We have implemented such a work item handler (org.drools.process.workitem.wsht.WSHumanTaskHandler in the drools-process-task module) so you can easily link this work item handler like this:

```
 StatefulKnowledgeSession session = ...;
 session.getWorkItemManager().registerWorkItemHandler("Human Task", new
WSHumanTaskHandler());
```

By default, this handler will connect to the human task management component on the local machine on port 9123, but you can easily change that by invoking the setConnection(ipAddress, port) method on the WSHumanTaskHandler.

At this moment WSHumanTaskHandler is using Mina *(http://mina.apache.org/)* [http://mina.apache.org/] for testing the behavior in a client/server architecture. Mina uses messages between client and server to enable the client comunicate with the server. That's why WSHumanTaskHandler have a MinaTaskClient that create different messages to give the user different actions that are executed for the server.

In the client (MinaTaskClient in this implementation) should see the implementation of the following methods to interact with Human Tasks:

```
 public void start(long taskId, String userId, TaskOperationResponseHandler
responseHandler)
 public void stop(long taskId, String userId, TaskOperationResponseHandler
responseHandler)
 public void release(long taskId, String userId,
TaskOperationResponseHandler responseHandler)
 public void suspend(long taskId, String userId,
TaskOperationResponseHandler responseHandler)
 public void resume(long taskId, String userId,
TaskOperationResponseHandler responseHandler)
 public void skip(long taskId, String userId, TaskOperationResponseHandler
responseHandler)
 public void delegate(long taskId, String userId, String targetUserId,
TaskOperationResponseHandler responseHandler)
 public void complete(long taskId, String userId, ContentData outputData,
TaskOperationResponseHandler responseHandler)
 ...
```

Using this methods we will implement any kind of GUI that the end user will use to do the task that they have assigned. If you take a look a this method signatures you will notice that almost all of this method takes the following arguments:

- **taskId: the id of the task with we are working. Probably you will pick this Id from the user task list in the UI (User Interface).**
- **userId**: the id of the user that is executing the action. Probably the Id of the user that is signed in the application.
- **responseHandler**: this is the handler have responsability to catch the response and get the results or just let us know that the task is already finished.

As you can imagine all the methods create a message that will be sended to the server, and the server will execute the logic that implement the correct action. A creation of one of this messages will be like this:

```
    public void complete(long taskId,
                         String userId,
                         ContentData outputData,
                         TaskOperationResponseHandler responseHandler) {
        List<Object> args = new ArrayList<Object>( 5 );
        args.add( Operation.Complete );
        args.add( taskId );
        args.add( userId );
        args.add( null );
        args.add( outputData );
        Command cmd = new Command( counter.getAndIncrement(),
                                   CommandName.OperationRequest,
                                   args );

        handler.addResponseHandler( cmd.getId(),
                                    responseHandler );

        session.write( cmd );
    }
```

Here we can see that a Command is created and the arguments of the method are inserted inside the command with the type of operation that we are trying to execute and then this command is sended to the server with session.write(cmd) method.

If we see the server implementation, when the command is recived, we find that depends of the operation type (here Operation.Complete) will be the logic that will be executed. If we look at the

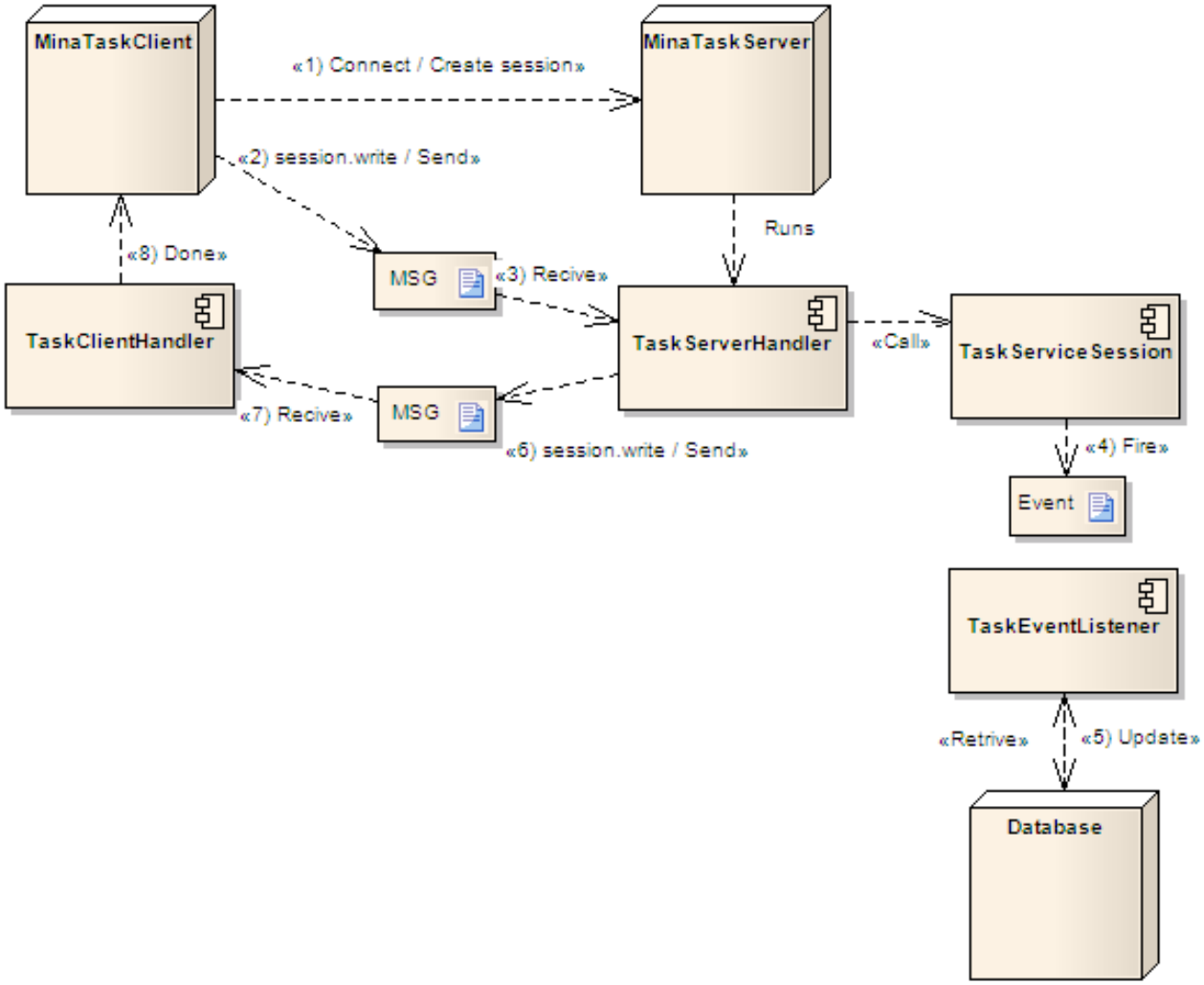
class TaskServerHandler in the messageReceived method the taskOperation is executed using the taskServiceSession that is the responsible for get, persist and manipulate all the Human Task Information when the tasks are created and the user is not interacting with them.

9.2.3. Starting the task management component

9.2.4. Interacting with the task management component

The task management component exposes various methods to manage the life cycle of the tasks through a Java API. This allows clients to integrate (at a low level) with the task management component. Note that end users should probably not interact with this low-level API directly but rather use one of the task list clients. These clients interact with the task management component using this API.

This interaction will be described with the following image:



As we can see in the image we have MinaTaskClient and MinaTaskServer. They communicate to each other sending messages to query and manipulate human tasks. Step by step the interaction will be something like this:

- Some client need to complete some task. So he/she needs to create an instace of MinaTaskClient and connect it to the MinaTaskServer to have a session to talk to each other. This is the step one in the image.
- Then the client can call the method complete() in MinaTaskClient with the corresponding arguments. This will generate a new Message (or Command) that will be inserted in the session that the client open when it connects to the server. This message must specify a type that the server recognize and know what to do when the message is recieved. This is the step two in the image.
- At this moment TaskServerHandler noticed that there is a new message in the session so an analysis about what kind of message is will take place. In this case is the type of Operation.Complete, because the client is finishing succesfully some task. So we need to complete the task that the user want to finish. This is achieved using the TaskServiceSession that will fire an specific type of event that will be procesed by an specific subclass of TaskEventListener. This are step three and four in the image.
- When the event is recived by TaskEventListener it will know how to modify the status of the task. This is achieved using the EntityManager to retrieve and modify the status of an specific task from the database. In this case, because we are finishing a task, the status will be updated to Completed. This is step five in the image.
- Now, when the changes are made we need to notify the client about that the task was succesfully ended and this is achieved creating a response message that TaskClientHandler will receive and inform MinaTaskClient. This are steps six, seven and eight in the image.

9.3. Human task management interface

9.3.1. Eclipse integration

9.3.2. Web-based task view

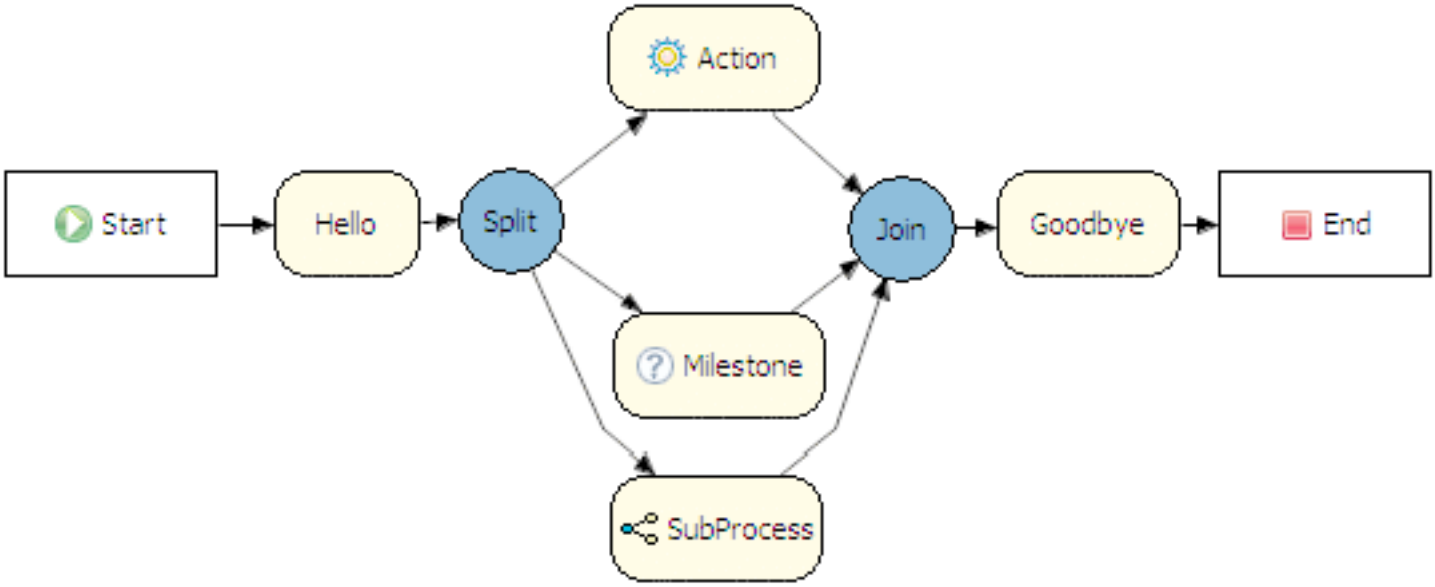
Chapter 10. Debugging processes

This section describes how to debug processes. This means that the current state of your running processes can be inspected and visualized during the execution. We use a simple example throughout this section to illustrate the debugging capabilities. The example will be introduced first, followed by an illustration on how to use the debugging capabilities.

10.1. A simple example

Our example contains two processes and some rules (used inside the ruleflow groups):

1. The main process contains some of the most common nodes: a start and end node (obviously), two ruleflow groups, an action (that simply prints a string to the default output), a milestone (a wait state that is trigger when a specific Event is inserted in the working memory) and a subprocess.



2. The SubProcess simply contains a milestone that also waits for (another) specific Event in the working memory.

3. There are only two rules (one for each ruleflow group) that simply print out either a hello world or goodbye world to default output.

We will simulate the execution of this process by starting the process, firing all rules (resulting in the executing of the hello rule), then adding the specific milestone events for both the milestones (in the main process and in the subprocess) and finally by firing all rules again (resulting in the executing of the goodbye rule). The console will look something like this:

```
Hello World
```

```
Executing action
Goodbye cruel world
```

10.2. Debugging the process

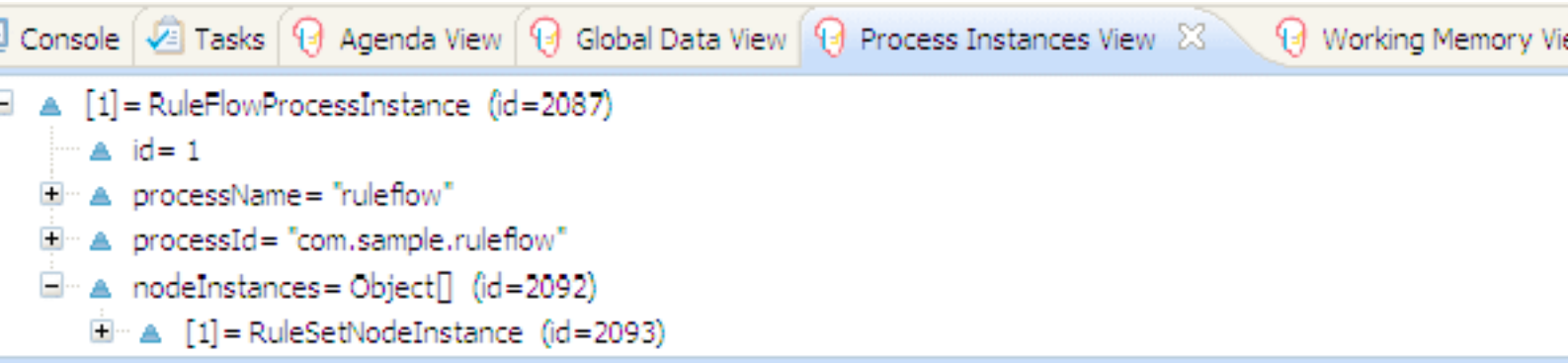
We now add four breakpoints during the execution of the process (in the order in which they will be encountered):

1. At the start of the consequence of the hello rule
2. Before inserting the triggering event for the milestone in the main process
3. Before inserting the triggering event for the milestone in the subprocess
4. At the start of the consequence of the goodbye rule

When debugging the application, one can use the following debug views to track the execution of the process:

1. The working memory view, showing the contents (data) in the working memory.
2. The agenda view, showing all activations in the agenda.
3. The global data view, showing the globals.
4. The default Java Debug views, showing the current line and the value of the known variables, and this both for normal Java code as for rules.
5. The process instances view, showing all running processes (and their state).
6. The audit view, showing the audit log.

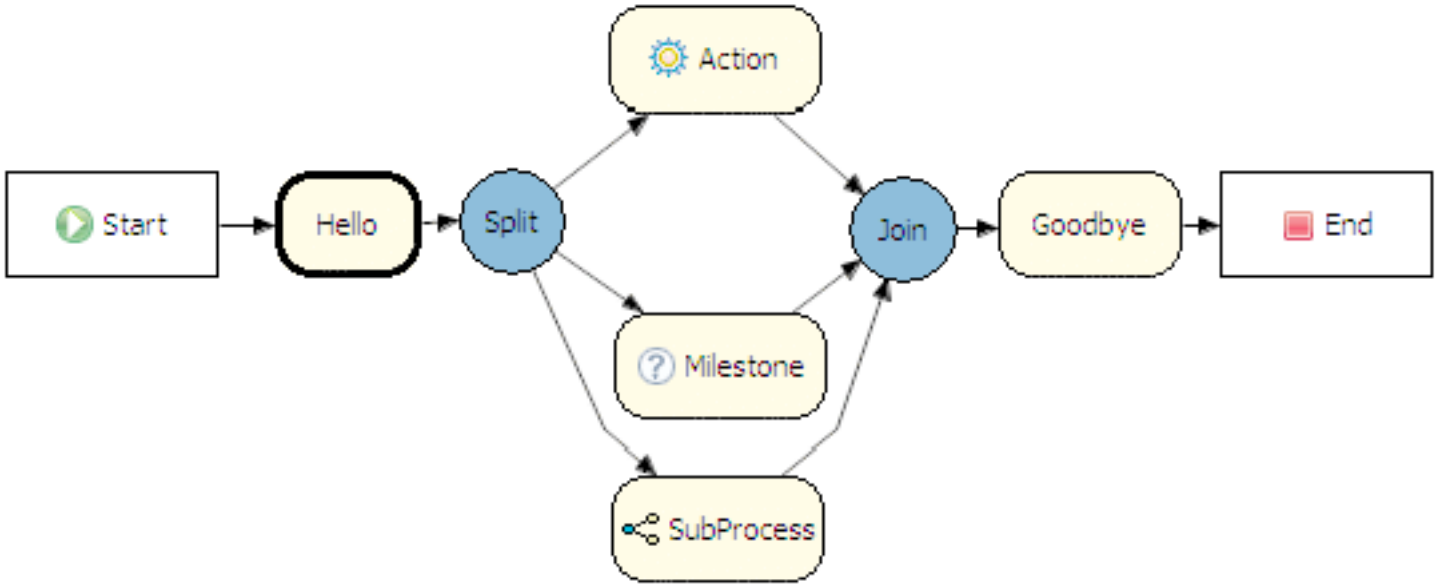
10.2.1. The Process Instances View



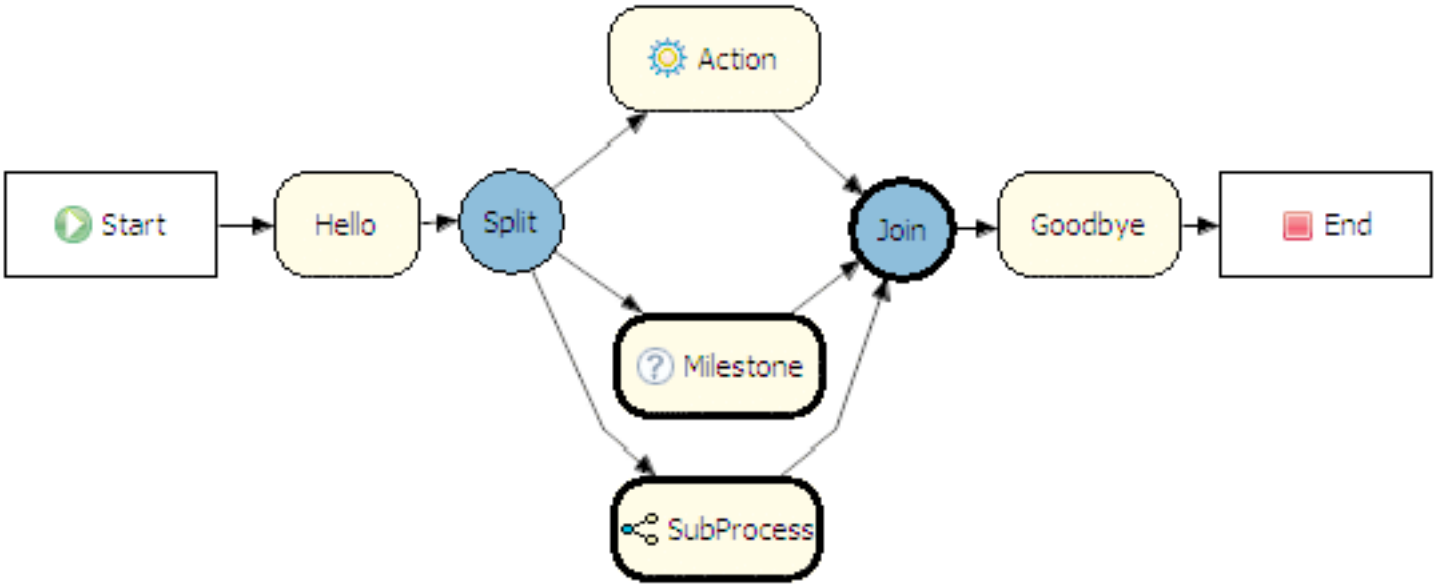
The process instances view shows the currently running process instances. The example shows that there is currently one running process (instance), currently executing one node (instance),

i.e. RuleSet node. When double-clicking a process instance, the process instance viewer will graphically show the progress of the process instance. At each of the breakpoints, this will look like:

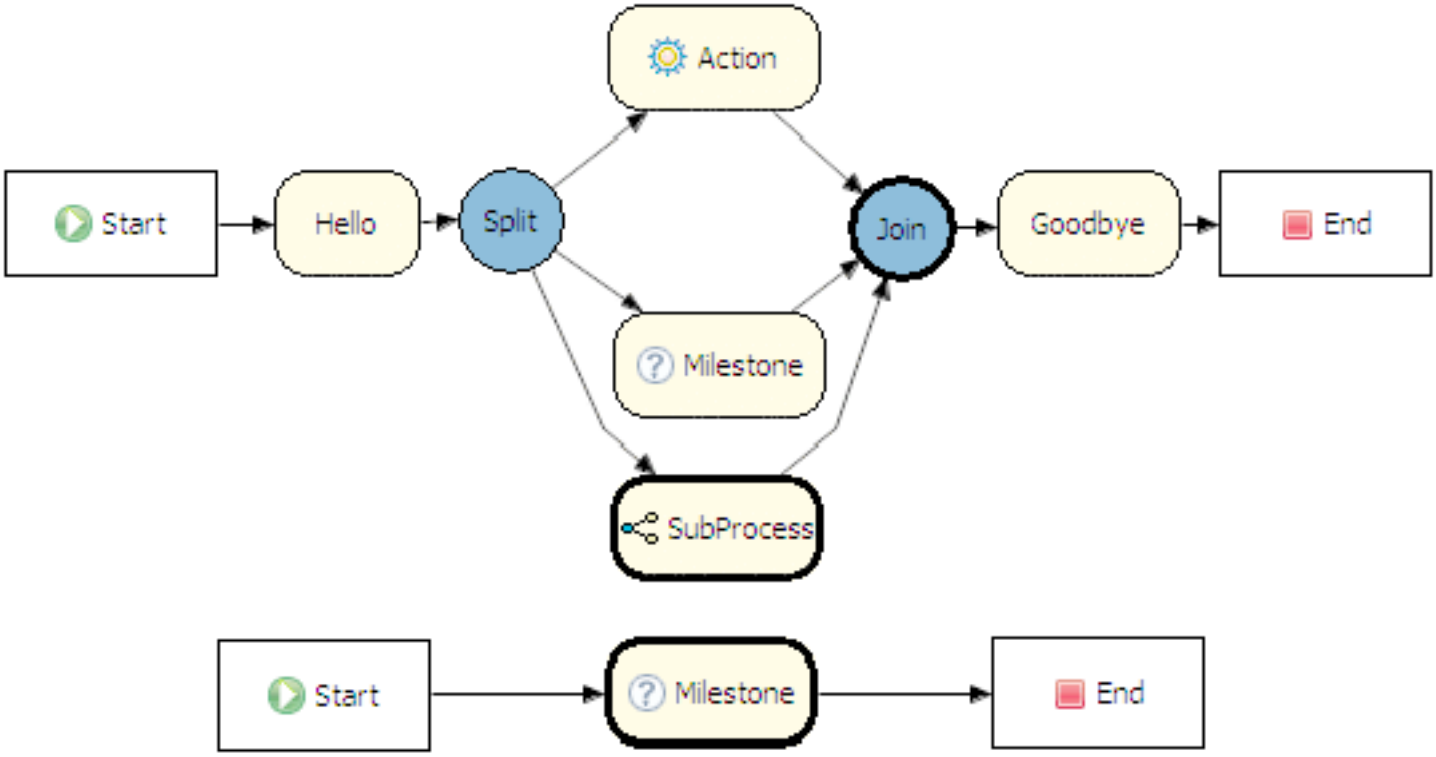
1. At the start of the consequence of the hello rule, only the hello ruleflow group is active, waiting on the execution of the hello rule:



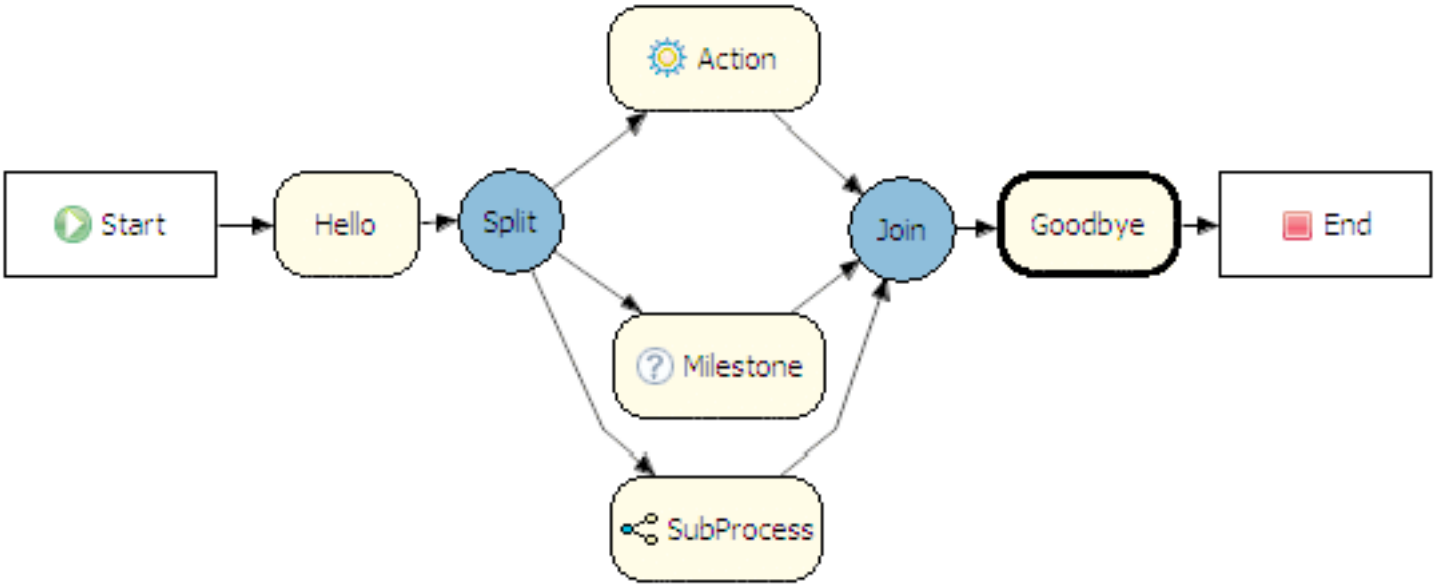
2. Once that rule has been executed, the action, the milestone and the subprocess will be triggered. The action will be executed immediately, triggering the join (which will simply wait until all incomming connections have been triggered). The subprocess will wait at the milestone. So, before inserting the triggering event for the milestone in the main process, there now are two process instances, looking like this:



3. When triggering the event for the milestone in the main process, this will also trigger the join (which will simply wait until all incomming connections have been triggered). So at that point (before inserting the triggering event for the milestone in the subprocess), the processes will look like this:



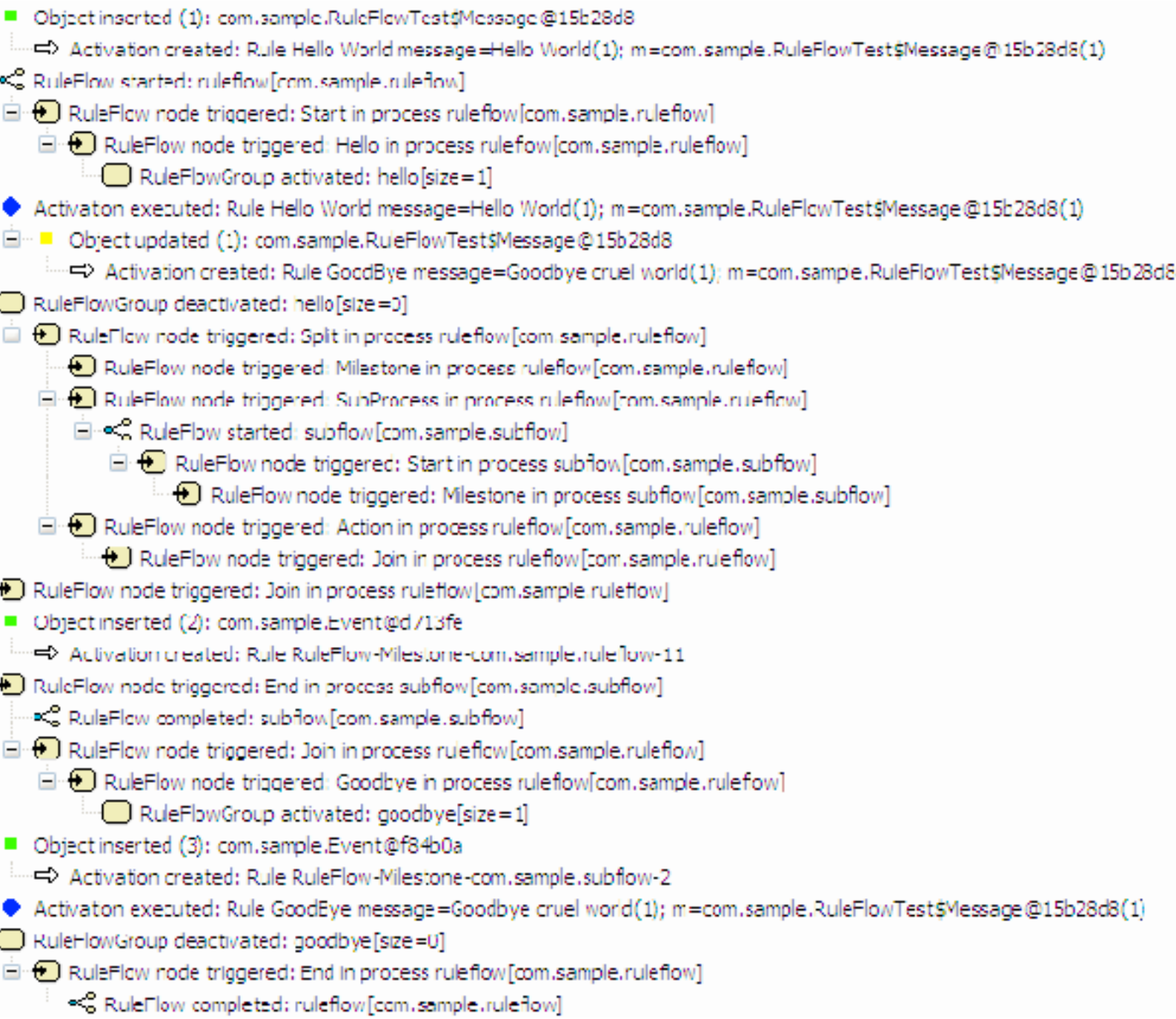
4. When triggering the event for the milestone in the subprocess, this process instance will be completed and this will also trigger the join, which will then continue and trigger the goodbye ruleflow group, as all its incomming connections have been triggered. Firing all the rules will trigger the breakpoint in the goodbye rule. At that point, the situation looks like this:



5. After executing the goodbye rule, the main process will also be completed and the execution will have reached the end.

10.2.2. The Audit View

For those who want to look at the result in the audit view, this will look something like this [Note: the object insertion events might seem a little out of place, which is caused by the fact that they are only logged after (and never before) they are inserted, making it difficult to exactly pinpoint their location.]


Object inserted (1): com.sample.RuleFlowTest$Message@15b28d8
Activation created: Rule Hello World message=Hello World(1); m=com.sample.RuleFlowTest$Message@15b28d8(1)
RuleFlow started: ruleflow[com.sample.ruleflow]
RuleFlow node triggered: Start in process ruleflow[com.sample.ruleflow]
RuleFlow node triggered: Hello in process rulefow[com.sample.ruleflow]
RuleFlowGroup activated: hello[size=1]
Activaton executed: Rule Hello World message=Hello World(1); m=com.sample.RuleFlowTest$Message@15b28d8(1)
Object updated (1): com.sample.RueFlowTest$Message@15b28d8
Activation created: Rule GoodBye message=Goodbye cruel world(1); m=com.sampe.RuleFlowTest$Message@15b28d8
RuleFlowGroup deactivated: hello[size=0]
RuleFlow node triggered: Split in process ruleflow[com.sample.ruleflow]
RuleFlow node triggered: Milestone in process ruleflow[com.sample.ruleflow]
RuleFlow node triggered: SubProcess in process ruleflow[com.sample.ruleflow]
RuleFlow started: subflow[com.sample.subflow]
RuleFlow node triggered: Start in process subflow[com.sample.subflow]
RuleFlow node triggered: Milestone in process subflow[com.sample.subflow]
RuleFlow node triggered: Action in process ruleflow[com.sample.ruleflow]
RuleFlow node triggered: Join in process ruleflow[com.sample.ruleflow]
RuleFlow node triggered: Join in process ruleflow[com.sample.ruleflow]
Object inserted (2): com.sample.Event@d713fe
Activation created: Rule RuleFlow-Milestone-com.sample.ruleflow-11
RuleFlow node triggered: End in process subflow[com.sample.subflow]
RuleFlow completed: subflow[com.sample.subflow]
RuleFlow node triggered: Join in process ruleflow[com.sample.ruleflow]
RuleFlow node triggered: Goodbye in process ruleflow[com.sample.rulefow]
RuleFlowGroup activated: goodbye[size=1]
Object inserted (3): com.sample.Event@f84b0a
Activation created: Rule RuleFlow-Milestone-com.sample.subflow-2
Activaton executed: Rule GoodEye message=Goodbye cruel word(1); m=com.sample.RuleFlowTest$Message@15b28d8(1)
RuleFlowGroup deactivated: goodbye[size=0]
RuleFlow node triggered: End in process ruleflow[com.sample.ruleflow]
RuleFlow completed: ruleflow[com.sample.ruleflow]

Chapter 11. Drools Eclipse IDE features

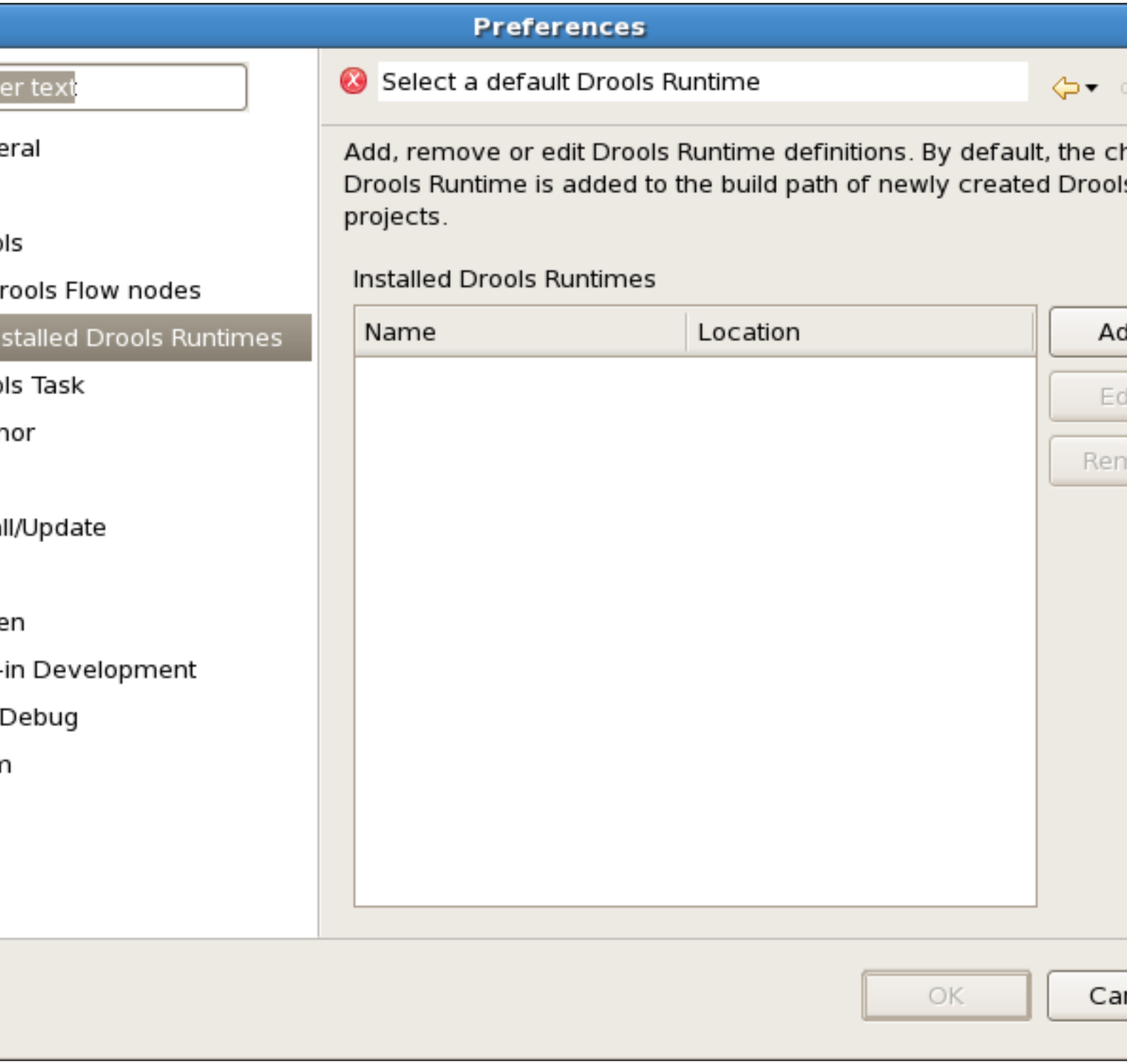
The Drools plugin for the Eclipse IDE provides a few additional features that might be interesting for developers.

11.1. Drools Runtimes

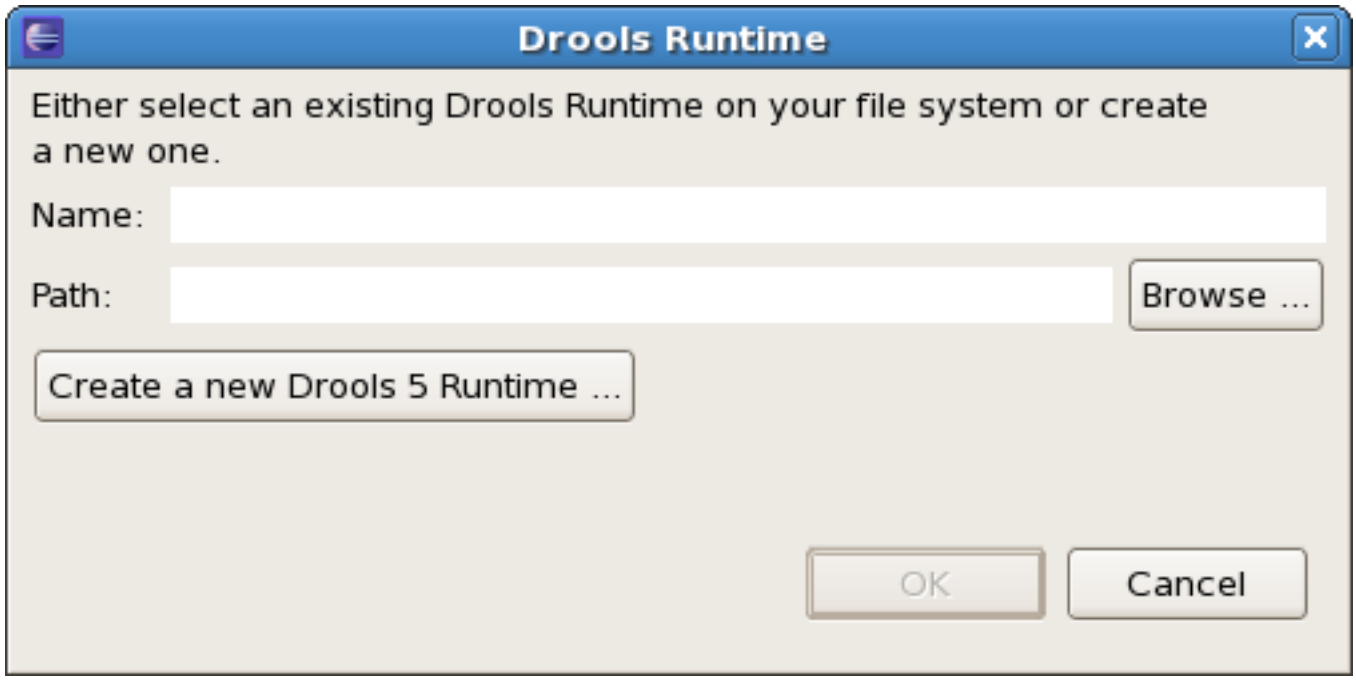
A Drools runtime is a collection of jars on your file system that represent one specific release of the Drools project jars. To create a runtime, you must point the IDE to the release of your choice. If you want to create a new runtime based on the latest Drools project jars included in the plugin itself, you can also easily do that. You are required to specify a default Drools runtime for your Eclipse workspace, but each individual project can override the default and select the appropriate runtime for that project specifically.

11.1.1. Defining a Drools runtime

You are required to define one or more Drools runtimes using the Eclipse preferences view. To open up your preferences, in the menu Window select the Preferences menu item. A new preferences dialog should show all your preferences. On the left side of this dialog, under the Drools category, select "Installed Drools runtimes". The panel on the right should then show the currently defined Drools runtimes. If you have not yet defined any runtimes, it should like something like the figure below.



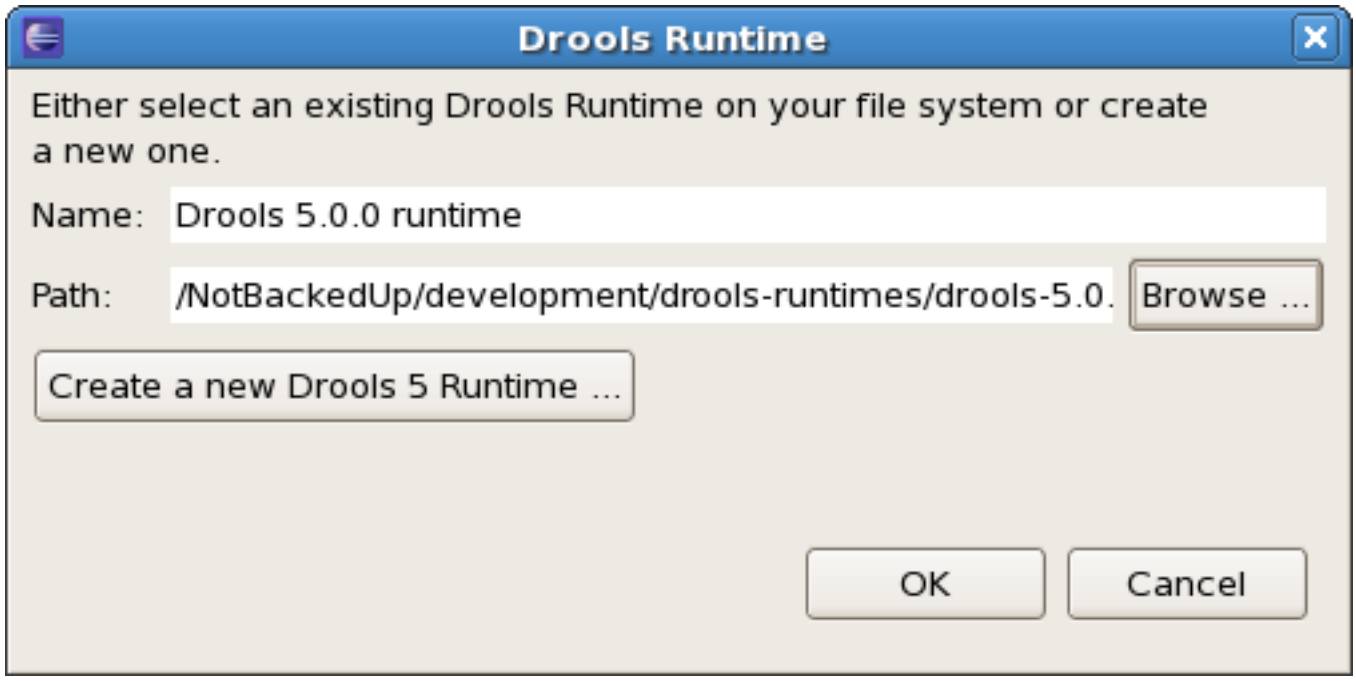
To define a new Drools runtime, click on the add button. A dialog as shown below should pop up, requiring the name for your runtime and the location on your file system where it can be found.



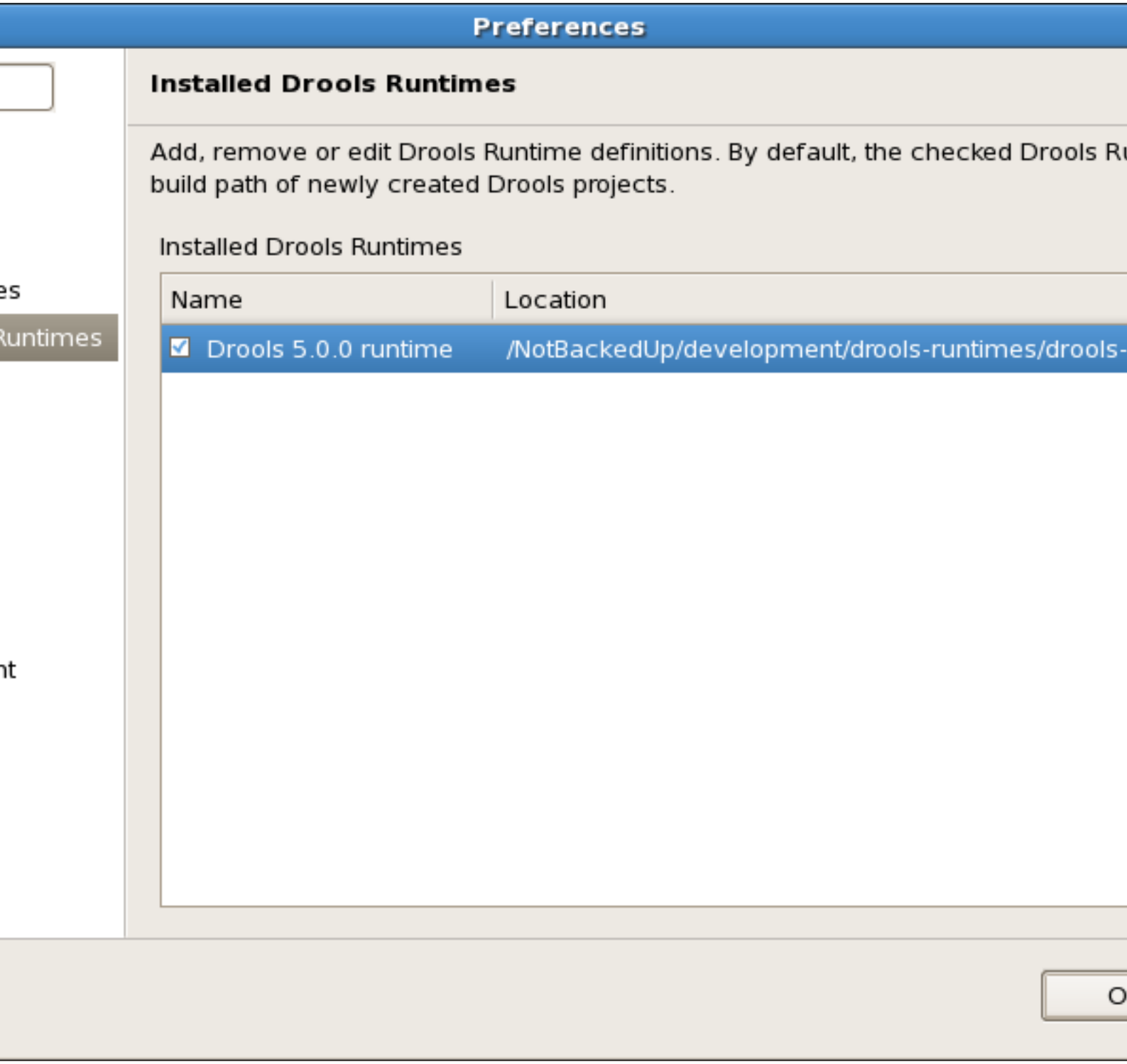
In general, you have two options:

1. If you simply want to use the default jars as included in the Drools Eclipse plugin, you can create a new Drools runtime automatically by clicking the "Create a new Drools 5 runtime ..." button. A file browser will show up, asking you to select the folder on your file system where you want this runtime to be created. The plugin will then automatically copy all required dependencies to the specified folder. After selecting this folder, the dialog should look like the figure shown below.

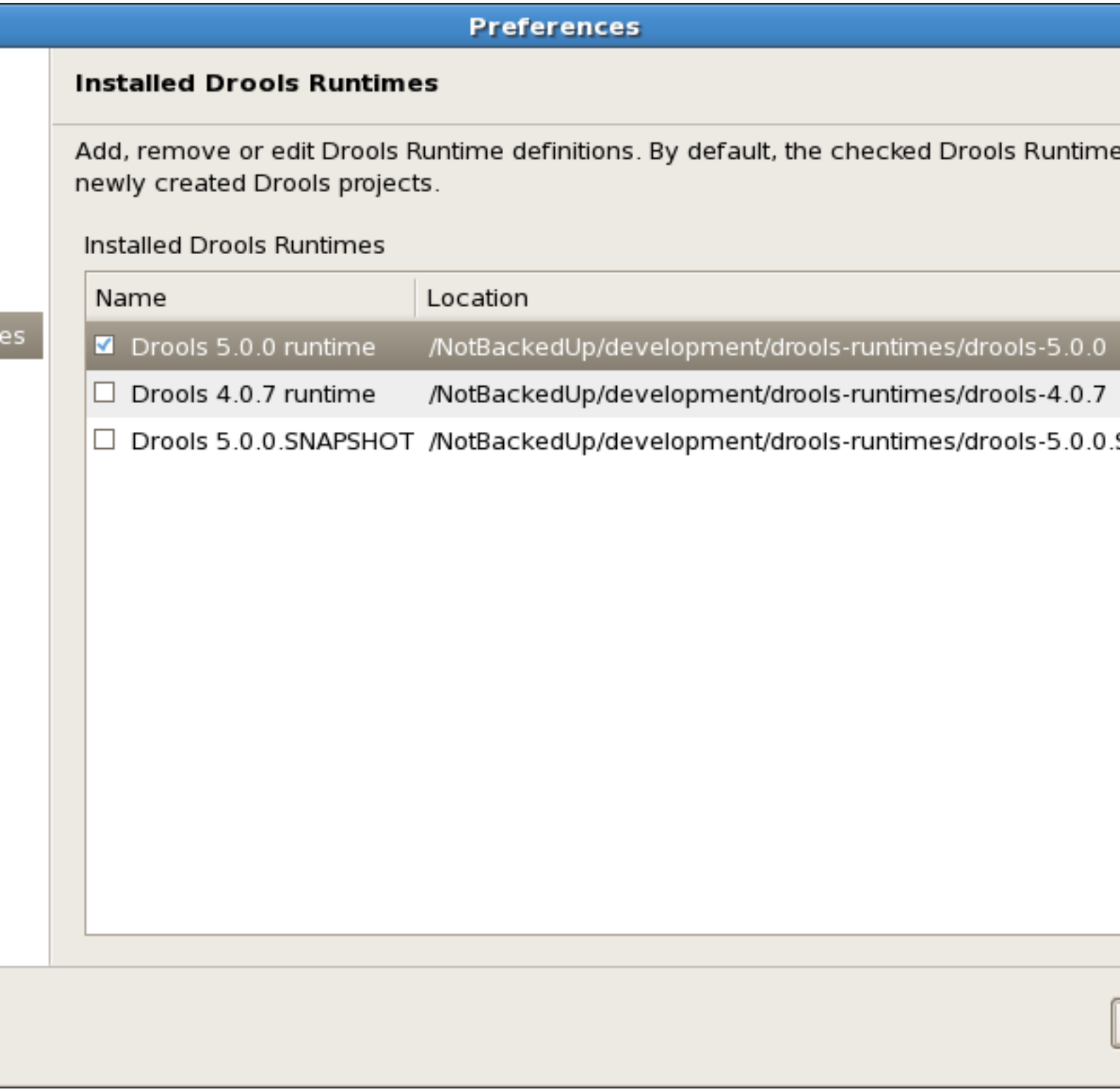
2. If you want to use one specific release of the Drools project, you should create a folder on your file system that contains all the necessary Drools libraries and dependencies. Instead of creating a new Drools runtime as explained above, give your runtime a name and select the location of this folder containing all the required jars.



After clicking the OK button, the runtime should show up in your table of installed Drools runtimes, as shown below. Click on checkbox in front of the newly created runtime to make it the default Drools runtime. The default Drools runtime will be used as the runtime of all your Drools project that have not selected a project-specific runtime.



You can add as many Drools runtimes as you need. For example, the screenshot below shows a configuration where three runtimes have been defined: a Drools 4.0.7 runtime, a Drools 5.0.0 runtime and a Drools 5.0.0.SNAPSHOT runtime. The Drools 5.0.0 runtime is selected as the default one.

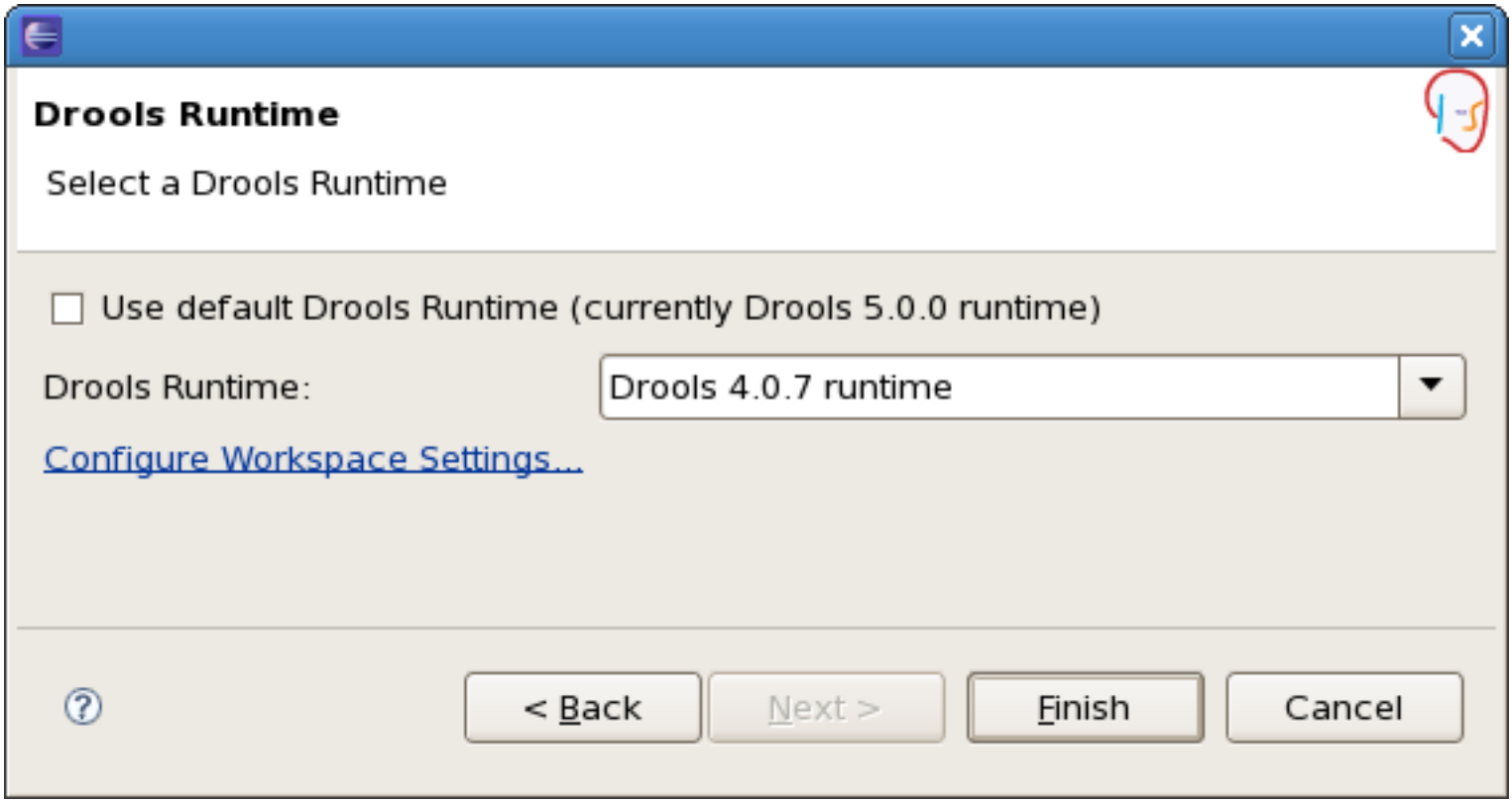


Note that you will need to restart Eclipse if you changed the default runtime and you want to make sure that all the projects that are using the default runtime update their classpath accordingly.

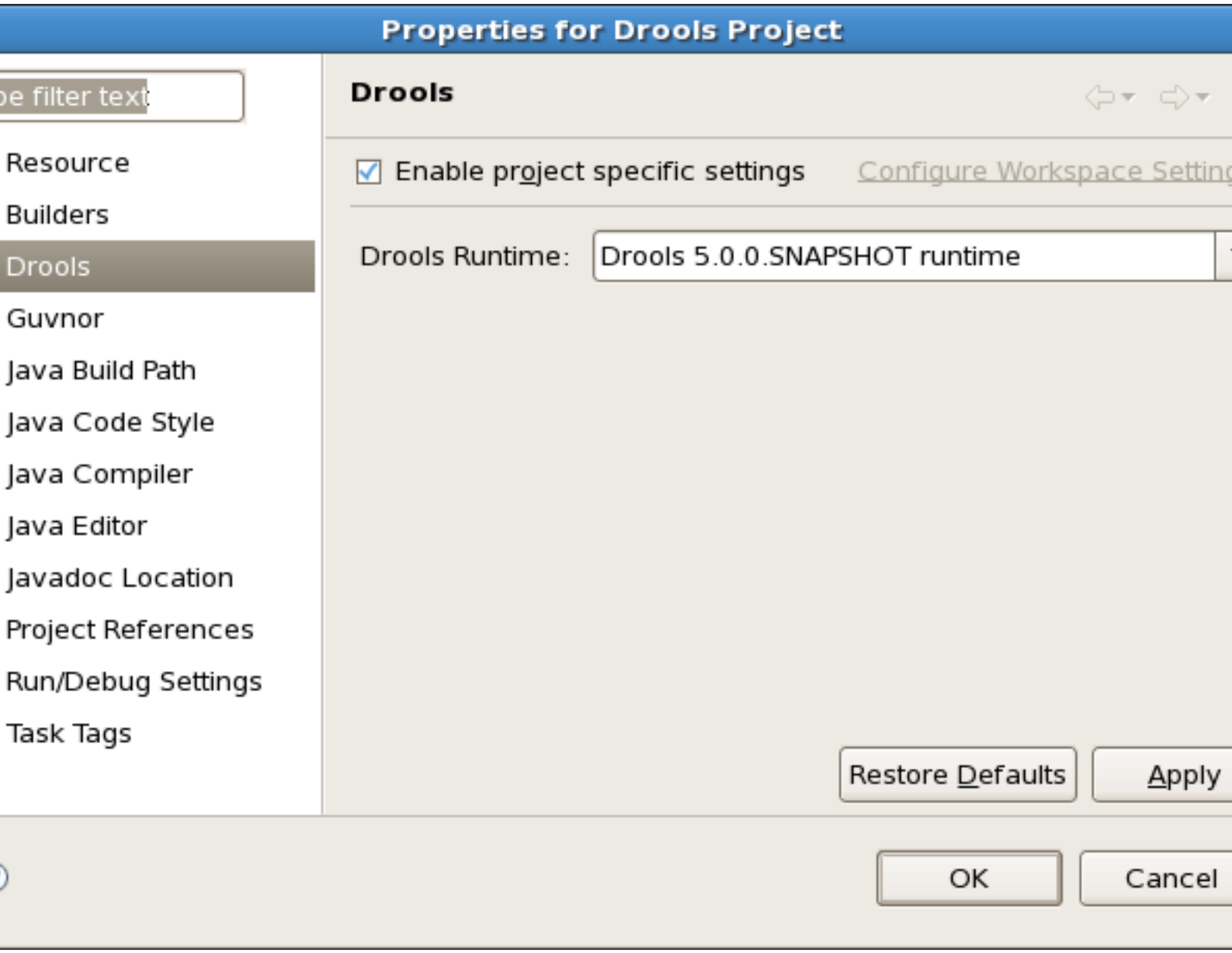
11.1.2. Selecting a runtime for your Drools project

Whenever you create a Drools project (using the New Drools Project wizard or by converting an existing Java project to a Drools project using the "Convert to Drools Project" action that is shown when you are in the Drools perspective and you right-click an existing Java project), the plugin will automatically add all the required jars to the classpath of your project.

When creating a new Drools project, the plugin will automatically use the default Drools runtime for that project, unless you specify a project-specific one. You can do this in the final step of the New Drools Project wizard, as shown below, by deselecting the "Use default Drools runtime" checkbox and selecting the appropriate runtime in the drop-down box. If you click the "Configure workspace settings ..." link, the workspace preferences showing the currently installed Drools runtimes will be opened, so you can add new runtimes there.



You can change the runtime of a Drools project at any time by opening the project properties (right-click the project and select Properties) and selecting the Drools category, as shown below. Check the "Enable project specific settings" checkbox and select the appropriate runtime from the drop-down box. If you click the "Configure workspace settings ..." link, the workspace preferences showing the currently installed Drools runtimes will be opened, so you can add new runtimes there. If you deselect the "Enable project specific settings" checkbox, it will use the default runtime as defined in your global preferences.

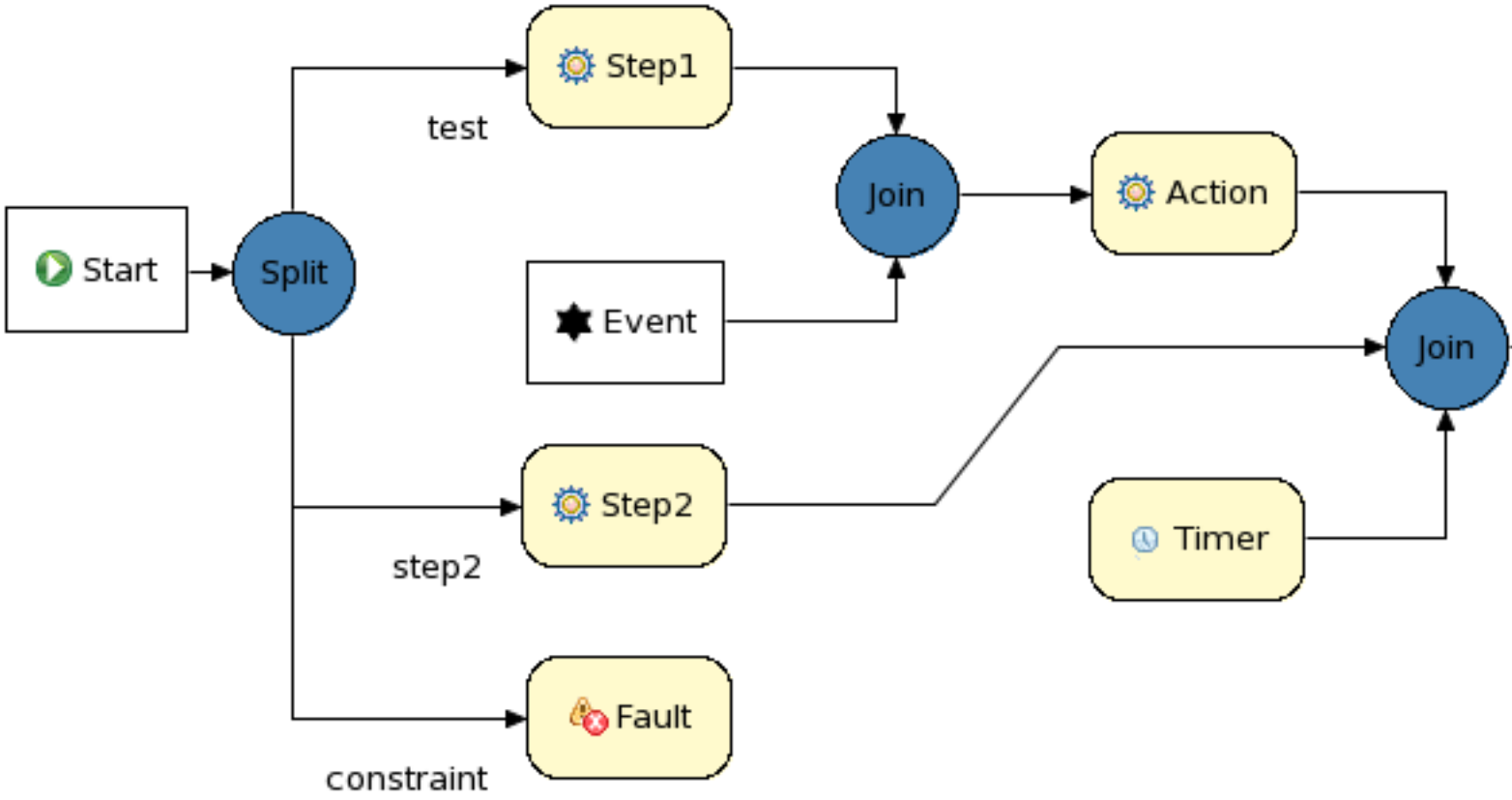


11.2. Process skins

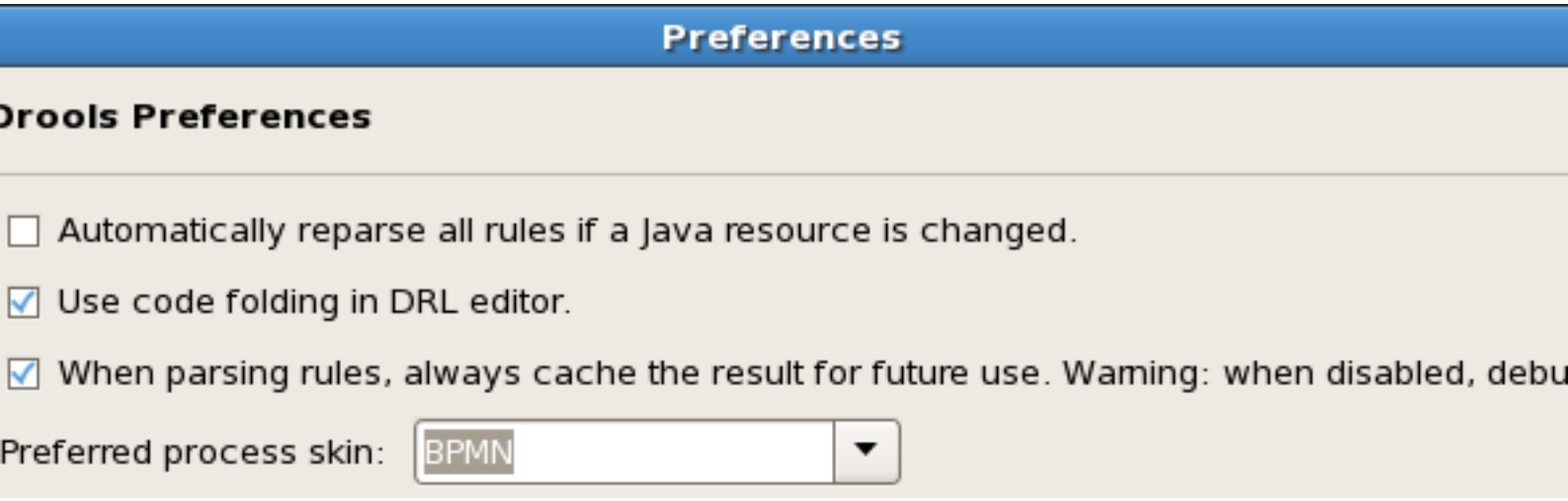
We have introduced the concept of a (process) skin, which controls how the different nodes inside a process are visualized. This allows you to change the visualization of the different node types the way you like them (by implementing your own SkinProvider).

BPMN is a popular language used by business users for modeling business processes. BPMN defines terminology, different types of nodes, how these should be visualized, etc. People who are familiar with BPMN might find it easier to implement an executable process (possibly based on a BPMN process diagram) using a similar visualization. We have therefore created a BPMN skin that maps the Drools Flow concepts to the equivalent BPMN visualization.

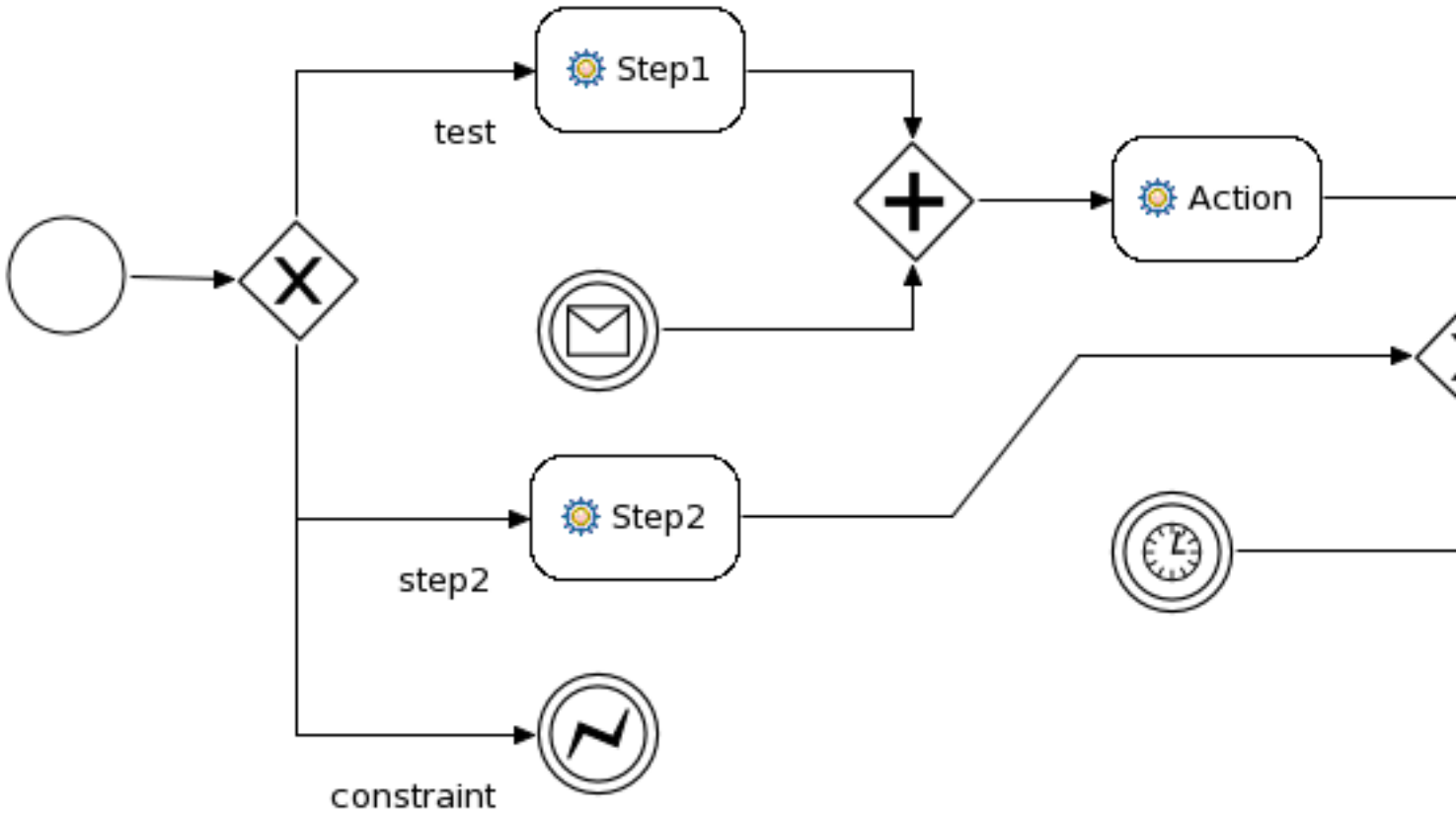
For example, the following figure shows a process using some of the different types of nodes in the RuleFlow language using the default skin ...



Simply by switching the preferred process skin in the Drools preferences ...



and then reopening the editor shows the same process using the BPMN skin ...


Step1
test
Action
Step2
step2
constraint

Index