JBoss SOA Platform 4.3

JBPM Reference Guide

Your guide to using JBoss jBPM with the JBoss Enterprise SOA Platform

JBoss SOA Platform 4.3 JBPM Reference Guide
Your guide to using JBoss jBPM with the JBoss Enterprise SOA Platform
Edition 1

The GPG fingerprint of the security@redhat.com key is:

CA 20 86 86 2B D6 9D FC 65 F6 EC C4 21 91 80 CD DB 42 A6 0E

1801 Varsity Drive
Raleigh, NC 27606-2072USAPhone: +1 919 754 3700
Phone: 888 733 4281
Fax: +1 919 754 3701
PO Box 13588Research Triangle Park, NC 27709USA

The JBPM jPDL 3.2 user guide for use with the JBoss SOA Platform 4.3

Preface

1. Document Conventions

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

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

1.1. Typographic Conventions

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

`Mono-spaced Bold`

Used to highlight system input, including shell commands, file names and paths. Also used to highlight key caps and key-combinations. For example:

> To see the contents of the file **`my_next_bestselling_novel`** in your current working directory, enter the **`cat my_next_bestselling_novel`** command at the shell prompt and press **`Enter`** to execute the command.

The above includes a file name, a shell command and a key cap, all presented in Mono-spaced Bold and all distinguishable thanks to context.

Key-combinations can be distinguished from key caps by the hyphen connecting each part of a key-combination. For example:

> Press **`Enter`** to execute the command.
>
> Press **`Ctrl`**-**`Alt`**-**`F1`** to switch to the first virtual terminal. Press **`Ctrl`**-**`Alt`**-**`F7`** to return to your X-Windows session.

The first sentence highlights the particular key cap to press. The second highlights two sets of three key caps, each set pressed simultaneously.

If source code is discussed, class names, methods, functions, variable names and returned values mentioned within a paragraph will be presented as above, in **`Mono-spaced Bold`**. For example:

> File-related classes include **`filesystem`** for file systems, **`file`** for files, and **`dir`** for directories. Each class has its own associated set of permissions.

Proportional Bold

This denotes words or phrases encountered on a system, including application names; dialogue box text; labelled buttons; check-box and radio button labels; menu titles and sub-menu titles. For example:

[1] https://fedorahosted.org/liberation-fonts/

Choose **System > Preferences > Mouse** from the main menu bar to launch **Mouse Preferences**. In the **Buttons** tab, click the **Left-handed mouse** check box and click **Close** to switch the primary mouse button from the left to the right (making the mouse suitable for use in the left hand).

To insert a special character into a **gedit** file, choose **Applications > Accessories > Character Map** from the main menu bar. Next, choose **Search > Find…** from the **Character Map** menu bar, type the name of the character in the **Search** field and click **Next**. The character you sought will be highlighted in the **Character Table**. Double-click this highlighted character to place it in the **Text to copy** field and then click the **Copy** button. Now switch back to your document and choose **Edit > Paste** from the **gedit** menu bar.

The above text includes application names; system-wide menu names and items; application-specific menu names; and buttons and text found within a GUI interface, all presented in Proportional Bold and all distinguishable by context.

Note the **>** shorthand used to indicate traversal through a menu and its sub-menus. This is to avoid the difficult-to-follow 'Select **Mouse** from the **Preferences** sub-menu in the **System** menu of the main menu bar' approach.

Mono-spaced Bold Italic or ***Proportional Bold Italic***

Whether Mono-spaced Bold or Proportional Bold, the addition of Italics indicates replaceable or variable text. Italics denotes text you do not input literally or displayed text that changes depending on circumstance. For example:

To connect to a remote machine using ssh, type **ssh *username*@*domain.name*** at a shell prompt. If the remote machine is **example.com** and your username on that machine is john, type **ssh john@example.com**.

The **mount -o remount *file-system*** command remounts the named file system. For example, to remount the **/home** file system, the command is **mount -o remount /home**.

To see the version of a currently installed package, use the **rpm -q *package*** command. It will return a result as follows: ***package-version-release***.

Note the words in bold italics above — username, domain.name, file-system, package, version and release. Each word is a placeholder, either for text you enter when issuing a command or for text displayed by the system.

Aside from standard usage for presenting the title of a work, italics denotes the first use of a new and important term. For example:

When the Apache HTTP Server accepts requests, it dispatches child processes or threads to handle them. This group of child processes or threads is known as a *server-pool*. Under Apache HTTP Server 2.0, the responsibility for creating and maintaining these server-pools has been abstracted to a group of modules called *Multi-Processing Modules* (*MPMs*). Unlike other modules, only one module from the MPM group can be loaded by the Apache HTTP Server.

1.2. Pull-quote Conventions

Two, commonly multi-line, data types are set off visually from the surrounding text.

Output sent to a terminal is set in `Mono-spaced Roman` and presented thus:

```
books        Desktop   documentation  drafts  mss    photos   stuff  svn
books_tests  Desktop1  downloads      images  notes  scripts  svgs
```

Source-code listings are also set in `Mono-spaced Roman` but are presented and highlighted as follows:

```
package org.jboss.book.jca.ex1;

import javax.naming.InitialContext;

public class ExClient
{
   public static void main(String args[])
       throws Exception
   {
      InitialContext iniCtx = new InitialContext();
      Object         ref    = iniCtx.lookup("EchoBean");
      EchoHome       home   = (EchoHome) ref;
      Echo           echo   = home.create();

      System.out.println("Created Echo");

      System.out.println("Echo.echo('Hello') = " + echo.echo("Hello"));
   }

}
```

1.3. Notes and Warnings

Finally, we use three visual styles to draw attention to information that might otherwise be overlooked.

Note

A note is a tip or shortcut or alternative approach to the task at hand. Ignoring a note should have no negative consequences, but you might miss out on a trick that makes your life easier.

Important

Important boxes detail things that are easily missed: configuration changes that only apply to the current session, or services that need restarting before an update will apply. Ignoring Important boxes won't cause data loss but may cause irritation and frustration.

Warning

A Warning should not be ignored. Ignoring warnings will most likely cause data loss.

2. We Need Feedback!

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

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

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

Introduction

JBoss jBPM is a flexible, extensible framework for process languages. jPDL is one process language that is build on top of that common framework. It is an intuitive process language to express business processes graphically in terms of tasks, wait states for asynchronous communication, timers, automated actions,... To bind these operations together, jPDL has the most powerful and extensible control flow mechanism.

jPDL has minimal dependencies and can be used as easy as using a java library. But it can also be used in environments where extreme throughput is crucial by deploying it on a J2EE clustered application server.

jPDL can be configured with any database and it can be deployed on any application server.

1.1. Overview

The core workflow and BPM functionality is packaged as a simple java library. This library includes a service to manage and execute processes in the jPDL database.

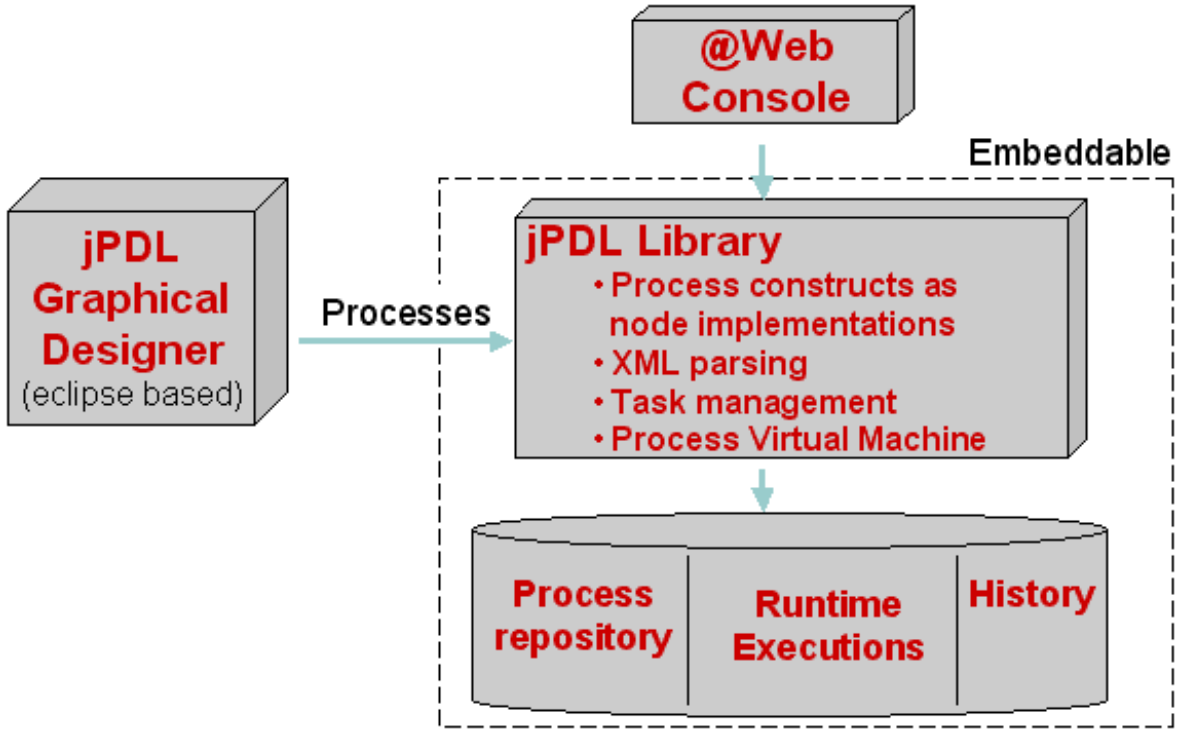

Figure 1.1. Overview of the jPDL components

1.2. The jPDL suite

The suite is a download that contains all the jBPM components bundled in one easy download. The download includes:

- **config**, configuration files for a standard java environment
- **db**, SQL scripts for DB creation and compatibility information

- **designer**, the eclipse plugin to author jPDL processes and installation scripts (this is not part of the plain jpdl download) See also *Section 1.3, "The jPDL graphical process designer"*.
- **doc**, userguide and javadocs
- **examples**
- **lib**, libraries on which jbpm depends. For more information on this see *Section 4.3, "Third party libraries"*
- **server**, a pre-configured jboss that contains jbpm inside the console web application (this is not part of the plain jpdl download)
- **src**, the jbpm and identity component java sources

The pre-configured JBoss application server has the following components installed :

- **The jBPM web console**, packaged as a web archive. That console can be used by process participants as well as jBPM administrators.
- **The Job Executor** for the execution of timers and messages. The job executor is a part of the console web application. There is a servlet that launches the Job Executor. The Job Executor spawns a thread pool for monitoring and executing timers and asynchronous messages.
- **The jBPM tables, in the database**: the default hypersonic database that contains the jBPM tables and already contains a process.
- **One example process** is already deployed into the jBPM database.
- **Identity component**. The identity component libraries are part of the console web application. The tables of the identity component are available in the database (those are the tables that start with JBPM_ID_...)

1.3. The jPDL graphical process designer

jPDL also includes a graphical designer tool. The designer is a graphical tool for authoring business processes. It's an eclipse plugin.

The most important feature of the graphical designer tool is that it includes support for both the business analyst as well as the technical developer. This enables a smooth transition from business process modeling to the practical implementation.

The plugin is available as a local update site (plain zip file) for installation via the standard eclipse software updates mechanism. And there is also a feature package that you can unzip in your eclipse home directory.

1.4. The jBPM console web application

The jBPM console web application serves two purposes. First, it serves as a central user interface for interacting with runtime tasks generated by the process executions. Secondly, it is an administration and monitoring console that allows to inspect and manipulate runtime instances. The third functionality is Business Activity Monitoring. These are statistics about process executions. This is useful information for managers to find bottlenecks or other kinds of optimizations.

1.5. The jBPM core library

The JBoss jBPM core component is the plain java (J2SE) library for managing process definitions and the runtime environment for execution of process instances.

JBoss jBPM is a java library. As a consequence, it can be used in any java environment like e.g. a web application, a swing application, an EJB, a webservice,... The jBPM library can also be packaged and exposed as a stateless session EJB. This allows clustered deployment and scalability for extreme high throughput. The stateless session EJB will be written against the J2EE 1.3 specifications so that it is deployable on any application server.

Depending on the functionality that you use, the library **`jbpm-jpdl.jar`** has some dependencies on other third party libraries such as e.g. hibernate, dom4j and others. We have done great efforts to require only those dependent libraries that you actually use. The dependencies are further documented in *Chapter 4, Deployment*

For its persistence, jBPM uses hibernate internally. Apart from traditional O/R mapping, hibernate also resolves the SQL dialect differences between the different databases, making jBPM portable across all current databases.

The JBoss jBPM API can be accessed from any custom java software in your project, like e.g. your web application, your EJB's, your web service components, your message driven beans or any other java component.

1.6. The JBoss jBPM identity component

JBoss jBPM can integrate with any company directory that contains users and other organizational information. But for projects where no organizational information component is readily available, JBoss jBPM includes this component. The model used in the identity component is richer than the traditional servlet-, ejb- and portlet models.

For more information, see *Section 10.11, “The identity component”*

1.7. The JBoss jBPM Job Executor

The JBoss jBPM Job Scheduler is a component for monitoring and executing jobs in a standard Java environment. Jobs are used for timers and asynchronous messages. In an enterprise environment, JMS and the EJB TimerService can be used for that purpose. But the Job Executor can be used in a standard environment.

The Job Executor component is packaged in the core jbpm-jpdl library, but it needs to be deployed in one of the following environments: either you have to configure the JbpmThreadsServlet to start the Job Executor or you have to start up a separate JVM and run the Job Executor thread in there.

Tutorial

This tutorial will show you basic process constructs in jpdl and the usage of the API for managing the runtime executions.

The format of this tutorial is explaining a set of examples. The examples focus on a particular topic and contain extensive comments. The examples can also be found in the jBPM download package in the directory **`src/java.examples`**.

The best way to learn is to create a project and experiment by creating variations on the examples given.

To get started for eclipse users: download jbpm-3.0-[version].zip and unzip it to your system. Then do "File" --> "Import..." --> "Existing Project into Workspace". Click "Next" Then, browse for the jBPM root directory and click "Finish". Now you have a jbpm.3 project in your workspace. You can now find the examples of the tutorial in **`src/java.examples/...`**. When you open these examples, you can run them with "Run" --> "Run As..." --> "JUnit Test"

jBPM includes a graphical designer tool for authoring the XML that is shown in the examples. You don't need the graphical designer tool to complete this tutorial.

State machines can be

2.1. Hello World example

A process definition is a directed graph, made up of nodes and transitions. The hello world process has 3 nodes. To see how the pieces fit together, we're going to start with a simple process without the use of the designer tool. The following picture shows the graphical representation of the hello world process:

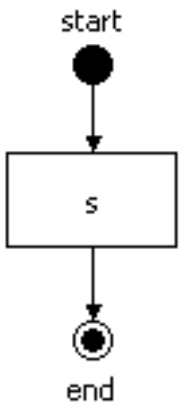

Figure 2.1. The hello world process graph

```
public void testHelloWorldProcess() {
// This method shows a process definition and one execution
// of the process definition.  The process definition has
// 3 nodes: an unnamed start-state, a state 's' and an
// end-state named 'end'.
// The next line parses a piece of xml text into a
// ProcessDefinition.  A ProcessDefinition is the formal
// description of a process represented as a java object.
```

```
ProcessDefinition processDefinition = ProcessDefinition.parseXmlString(
    "<process-definition>" +
    "  <start-state>" +
    "    <transition to='s' />" +
    "  </start-state>" +
    "  <state name='s'>" +
    "    <transition to='end' />" +
    "  </state>" +
    "  <end-state name='end' />" +
    "</process-definition>"
);

  // The next line creates one execution of the process definition.
  // After construction, the process execution has one main path
  // of execution (=the root token) that is positioned in the
  // start-state.
  ProcessInstance processInstance =
      new ProcessInstance(processDefinition);

  // After construction, the process execution has one main path
  // of execution (=the root token).
  Token token = processInstance.getRootToken();

  // Also after construction, the main path of execution is positioned
  // in the start-state of the process definition.
  assertSame(processDefinition.getStartState(), token.getNode());

  // Let's start the process execution, leaving the start-state
  // over its default transition.
  token.signal();
  // The signal method will block until the process execution
  // enters a wait state.

  // The process execution will have entered the first wait state
  // in state 's'. So the main path of execution is now
  // positioned in state 's'
  assertSame(processDefinition.getNode("s"), token.getNode());

  // Let's send another signal.  This will resume execution by
  // leaving the state 's' over its default transition.
  token.signal();
  // Now the signal method returned because the process instance
  // has arrived in the end-state.

  assertSame(processDefinition.getNode("end"), token.getNode());
}
```

2.2. Database example

One of the basic features of jBPM is the ability to persist executions of processes in the database when they are in a wait state. The next example will show you how to store a process instance in the

jBPM database. The example also suggests a context in which this might occur. Separate methods are created for different pieces of user code. E.g. an piece of user code in a web application starts a process and persists the execution in the database. Later, a message driven bean loads the process instance from the database and resumes its execution.

More about the jBPM persistence can be found in *Chapter 6, Persistence*.

```
public class HelloWorldDbTest extends TestCase {

  static JbpmConfiguration jbpmConfiguration = null;

  static {
    // An example configuration file such as this can be found in
    // 'src/config.files'.  Typically the configuration information is in
 the
    // resource file 'jbpm.cfg.xml', but here we pass in the
 configuration
    // information as an XML string.

    // First we create a JbpmConfiguration statically.  One
 JbpmConfiguration
    // can be used for all threads in the system, that is why we can
 safely
    // make it static.

    jbpmConfiguration = JbpmConfiguration.parseXmlString(
      "<jbpm-configuration>" +

      // A jbpm-context mechanism separates the jbpm core
      // engine from the services that jbpm uses from
      // the environment.

      "  <jbpm-context>" +
      "    <service name='persistence' " +
      "
 factory='org.jbpm.persistence.db.DbPersistenceServiceFactory' />" +
      "  </jbpm-context>" +

      // Also all the resource files that are used by jbpm are
      // referenced from the jbpm.cfg.xml

      "  <string name='resource.hibernate.cfg.xml' " +
      "          value='hibernate.cfg.xml' />" +
      "  <string name='resource.business.calendar' " +
      "          value='org/jbpm/calendar/
jbpm.business.calendar.properties' />" +
      "  <string name='resource.default.modules' " +
      "          value='org/jbpm/graph/def/
jbpm.default.modules.properties' />" +
      "  <string name='resource.converter' " +
```

```
      "          value='org/jbpm/db/hibernate/
jbpm.converter.properties' />" +
      "  <string name='resource.action.types' " +
      "          value='org/jbpm/graph/action/action.types.xml' />" +
      "  <string name='resource.node.types' " +
      "          value='org/jbpm/graph/node/node.types.xml' />" +
      "  <string name='resource.varmapping' " +
      "          value='org/jbpm/context/exe/jbpm.varmapping.xml' />" +
      "</jbpm-configuration>"
    );
  }

  public void setUp() {
    jbpmConfiguration.createSchema();
  }

  public void tearDown() {
    jbpmConfiguration.dropSchema();
  }

  public void testSimplePersistence() {
    // Between the 3 method calls below, all data is passed via the
    // database.  Here, in this unit test, these 3 methods are executed
    // right after each other because we want to test a complete process
    // scenario.  But in reality, these methods represent different
    // requests to a server.

    // Since we start with a clean, empty in-memory database, we have to
    // deploy the process first.  In reality, this is done once by the
    // process developer.
    deployProcessDefinition();

    // Suppose we want to start a process instance (=process execution)
    // when a user submits a form in a web application...
    processInstanceIsCreatedWhenUserSubmitsWebappForm();

    // Then, later, upon the arrival of an asynchronous message the
    // execution must continue.
    theProcessInstanceContinuesWhenAnAsyncMessageIsReceived();
  }

  public void deployProcessDefinition() {
    // This test shows a process definition and one execution
    // of the process definition.  The process definition has
    // 3 nodes: an unnamed start-state, a state 's' and an
    // end-state named 'end'.
    ProcessDefinition processDefinition =
 ProcessDefinition.parseXmlString(
      "<process-definition name='hello world'>" +
      "  <start-state name='start'>" +
      "    <transition to='s' />" +
```

```
      "  </start-state>" +
      "  <state name='s'>" +
      "    <transition to='end' />" +
      "  </state>" +
      "  <end-state name='end' />" +
      "</process-definition>"
    );

    // Lookup the POJO persistence context-builder that is configured
above
    JbpmContext jbpmContext = jbpmConfiguration.createJbpmContext();
    try {
      // Deploy the process definition in the database
      jbpmContext.deployProcessDefinition(processDefinition);

    } finally {
      // Tear down the POJO persistence context.
      // This includes flush the SQL for inserting the process definition

      // to the database.
      jbpmContext.close();
    }
  }

  public void processInstanceIsCreatedWhenUserSubmitsWebappForm() {
    // The code in this method could be inside a struts-action
    // or a JSF managed bean.

    // Lookup the POJO persistence context-builder that is configured
above
    JbpmContext jbpmContext = jbpmConfiguration.createJbpmContext();
    try {

      GraphSession graphSession = jbpmContext.getGraphSession();

      ProcessDefinition processDefinition =
          graphSession.findLatestProcessDefinition("hello world");

      // With the processDefinition that we retrieved from the database,
we
      // can create an execution of the process definition just like in
the
      // hello world example (which was without persistence).
      ProcessInstance processInstance =
          new ProcessInstance(processDefinition);

      Token token = processInstance.getRootToken();
      assertEquals("start", token.getNode().getName());
      // Let's start the process execution
      token.signal();
      // Now the process is in the state 's'.
```

```
      assertEquals("s", token.getNode().getName());

      // Now the processInstance is saved in the database.  So the
      // current state of the execution of the process is stored in the
      // database.
      jbpmContext.save(processInstance);
      // The method below will get the process instance back out
      // of the database and resume execution by providing another
      // external signal.

    } finally {
      // Tear down the POJO persistence context.
      jbpmContext.close();
    }
  }

  public void theProcessInstanceContinuesWhenAnAsyncMessageIsReceived() {
    // The code in this method could be the content of a message driven
bean.

    // Lookup the POJO persistence context-builder that is configured
above
    JbpmContext jbpmContext = jbpmConfiguration.createJbpmContext();
    try {

      GraphSession graphSession = jbpmContext.getGraphSession();
      // First, we need to get the process instance back out of the
database.
      // There are several options to know what process instance we are
dealing
      // with here.  The easiest in this simple test case is just to look
for
      // the full list of process instances.  That should give us only
one
      // result.  So let's look up the process definition.

      ProcessDefinition processDefinition =
          graphSession.findLatestProcessDefinition("hello world");

      // Now, we search for all process instances of this process
definition.
      List processInstances =
          graphSession.findProcessInstances(processDefinition.getId());

      // Because we know that in the context of this unit test, there is
      // only one execution.  In real life, the processInstanceId can be
      // extracted from the content of the message that arrived or from
      // the user making a choice.
      ProcessInstance processInstance =
          (ProcessInstance) processInstances.get(
<xslthl:number>0</xslthl:number>
```

```
);

      // Now we can continue the execution.  Note that the
 processInstance
      // delegates signals to the main path of execution (=the root
 token).
      processInstance.signal();

      // After this signal, we know the process execution should have
      // arrived in the end-state.
      assertTrue(processInstance.hasEnded());

      // Now we can update the state of the execution in the database
      jbpmContext.save(processInstance);

    } finally {
      // Tear down the POJO persistence context.
      jbpmContext.close();
    }
  }
}
```

2.3. Context example: process variables

The process variables contain the context information during process executions. The process variables are similar to a **`java.util.Map`** that maps variable names to values, which are java objects. The process variables are persisted as a part of the process instance. To keep things simple, in this example we only show the API to work with variables, without persistence.

More information about variables can be found in *Chapter 9, Context*

```
// This example also starts from the hello world process.
// This time even without modification.
ProcessDefinition processDefinition = ProcessDefinition.parseXmlString(
  "<process-definition>" +
  "  <start-state>" +
  "    <transition to='s' />" +
  "  </start-state>" +
  "  <state name='s'>" +
  "    <transition to='end' />" +
  "  </state>" +
  "  <end-state name='end' />" +
  "</process-definition>"
);

ProcessInstance processInstance =
  new ProcessInstance(processDefinition);

// Fetch the context instance from the process instance
// for working with the process variables.
ContextInstance contextInstance =
```

```
  processInstance.getContextInstance();

// Before the process has left the start-state,
// we are going to set some process variables in the
// context of the process instance.
contextInstance.setVariable("amount", new Integer(
<xslthl:number>500</xslthl:number>
));
contextInstance.setVariable("reason", "i met my deadline");

// From now on, these variables are associated with the
// process instance.  The process variables are now accessible
// by user code via the API shown here, but also in the actions
// and node implementations.  The process variables are also
// stored into the database as a part of the process instance.

processInstance.signal();

// The variables are accessible via the contextInstance.

assertEquals(new Integer(
<xslthl:number>500</xslthl:number>
),
             contextInstance.getVariable("amount"));
assertEquals("i met my deadline",
             contextInstance.getVariable("reason"));
```

2.4. Task assignment example

In the next example we'll show how you can assign a task to a user. Because of the separation between the jBPM workflow engine and the organizational model, an expression language for calculating actors would always be too limited. Therefore, you have to specify an implementation of AssignmentHandler for including the calculation of actors for tasks.

```
public void testTaskAssignment() {
  // The process shown below is based on the hello world process.
  // The state node is replaced by a task-node.  The task-node
  // is a node in JPDL that represents a wait state and generates
  // task(s) to be completed before the process can continue to
  // execute.
  ProcessDefinition processDefinition = ProcessDefinition.parseXmlString(
    "<process-definition name='the baby process'>" +
    "  <start-state>" +
    "    <transition name='baby cries' to='t' />" +
    "  </start-state>" +
    "  <task-node name='t'>" +
    "    <task name='change nappy'>" +
    "      <assignment
 class='org.jbpm.tutorial.taskmgmt.NappyAssignmentHandler' />" +
    "    </task>" +
    "    <transition to='end' />" +
```

```
      "  </task-node>" +
      "  <end-state name='end' />" +
      "</process-definition>"
    );

    // Create an execution of the process definition.
    ProcessInstance processInstance =
        new ProcessInstance(processDefinition);
    Token token = processInstance.getRootToken();

    // Let's start the process execution, leaving the start-state
    // over its default transition.
    token.signal();
    // The signal method will block until the process execution
    // enters a wait state.   In this case, that is the task-node.
    assertSame(processDefinition.getNode("t"), token.getNode());

    // When execution arrived in the task-node, a task 'change nappy'
    // was created and the NappyAssignmentHandler was called to determine
    // to whom the task should be assigned.  The NappyAssignmentHandler
    // returned 'papa'.

    // In a real environment, the tasks would be fetched from the
    // database with the methods in the org.jbpm.db.TaskMgmtSession.
    // Since we don't want to include the persistence complexity in
    // this example, we just take the first task-instance of this
    // process instance (we know there is only one in this test
    // scenario).
    TaskInstance taskInstance = (TaskInstance)
        processInstance
          .getTaskMgmtInstance()
          .getTaskInstances()
          .iterator().next();

    // Now, we check if the taskInstance was actually assigned to 'papa'.
    assertEquals("papa", taskInstance.getActorId() );

    // Now we suppose that 'papa' has done his duties and mark the task
    // as done.
    taskInstance.end();
    // Since this was the last (only) task to do, the completion of this
    // task triggered the continuation of the process instance execution.

    assertSame(processDefinition.getNode("end"), token.getNode());
  }
```

2.5. Custom action example

Actions are a mechanism to bind your custom java code into a jBPM process. Actions can be associated with its own nodes (if they are relevant in the graphical representation of the process). Or actions can be placed on events like e.g. taking a transition, leaving a node or entering a node. In that

case, the actions are not part of the graphical representation, but they are executed when execution fires the events in a runtime process execution.

We'll start with a look at the action implementation that we are going to use in our example : **`MyActionHandler`**. This action handler implementation does not do really spectacular things... it just sets the boolean variable **`isExecuted`** to **`true`**. The variable **`isExecuted`** is static so it can be accessed from within the action handler as well as from the action to verify it's value.

More information about actions can be found in *Section 8.5, "Actions"*

```
// MyActionHandler represents a class that could execute
// some user code during the execution of a jBPM process.
public class MyActionHandler implements ActionHandler {

  // Before each test (in the setUp), the isExecuted member
  // will be set to false.
  public static boolean isExecuted = false;

  // The action will set the isExecuted to true so the
  // unit test will be able to show when the action
  // is being executed.
  public void execute(ExecutionContext executionContext) {
    isExecuted = true;
  }
}
```

As mentioned before, before each test, we'll set the static field **`MyActionHandler.isExecuted`** to false;

```
  // Each test will start with setting the static isExecuted
  // member of MyActionHandler to false.
  public void setUp() {
    MyActionHandler.isExecuted = false;
  }
```

We'll start with an action on a transition.

```
public void testTransitionAction() {
    // The next process is a variant of the hello world process.
    // We have added an action on the transition from state 's'
    // to the end-state.  The purpose of this test is to show
    // how easy it is to integrate java code in a jBPM process.
    ProcessDefinition processDefinition =
 ProcessDefinition.parseXmlString(
      "<process-definition>" +
      "  <start-state>" +
      "    <transition to='s' />" +
      "  </start-state>" +
      "  <state name='s'>" +
      "    <transition to='end'>" +
```

```
      "      <action class='org.jbpm.tutorial.action.MyActionHandler' />"
 +
      "    </transition>" +
      "  </state>" +
      "  <end-state name='end' />" +
      "</process-definition>"
    );

    // Let's start a new execution for the process definition.
    ProcessInstance processInstance =
      new ProcessInstance(processDefinition);

    // The next signal will cause the execution to leave the start
    // state and enter the state 's'
    processInstance.signal();

    // Here we show that MyActionHandler was not yet executed.
    assertFalse(MyActionHandler.isExecuted);
    // ... and that the main path of execution is positioned in
    // the state 's'
    assertSame(processDefinition.getNode("s"),
               processInstance.getRootToken().getNode());

    // The next signal will trigger the execution of the root
    // token.  The token will take the transition with the
    // action and the action will be executed during the
    // call to the signal method.
    processInstance.signal();

    // Here we can see that MyActionHandler was executed during
    // the call to the signal method.
    assertTrue(MyActionHandler.isExecuted);
  }
```

The next example shows the same action, but now the actions are placed on the **`enter-node`** and **`leave-node`** events respectively. Note that a node has more than one event type in contrast to a transition, which has only one event. Therefore actions placed on a node should be put in an event element.

```
ProcessDefinition processDefinition = ProcessDefinition.parseXmlString(
  "<process-definition>" +
  "  <start-state>" +
  "    <transition to='s' />" +
  "  </start-state>" +
  "  <state name='s'>" +
  "    <event type='node-enter'>" +
  "      <action class='org.jbpm.tutorial.action.MyActionHandler' />" +
  "    </event>" +
  "    <event type='node-leave'>" +
  "      <action class='org.jbpm.tutorial.action.MyActionHandler' />" +
  "    </event>" +
```

```
  "    <transition to='end'/>" +
  "  </state>" +
  "  <end-state name='end' />" +
  "</process-definition>"
);

ProcessInstance processInstance =
  new ProcessInstance(processDefinition);

assertFalse(MyActionHandler.isExecuted);
// The next signal will cause the execution to leave the start
// state and enter the state 's'.  So the state 's' is entered
// and hence the action is executed.
processInstance.signal();
assertTrue(MyActionHandler.isExecuted);

// Let's reset the MyActionHandler.isExecuted
MyActionHandler.isExecuted = false;

// The next signal will trigger execution to leave the
// state 's'.  So the action will be executed again.
processInstance.signal();
// Voila.
assertTrue(MyActionHandler.isExecuted);
```

Graph Oriented Programming

3.1. Introduction

This chapter can be considered the manifest for JBoss jBPM. It gives a complete overview of the vision and ideas behind current strategy and future directions of the JBoss jBPM project. This vision significantly differs from the traditional approach.

First of all, we believe in multiple process languages. There are different environments and different purposes that require a their own specific process language.

Secondly, Graph Oriented Programming is a new implementation technique that serves as a basis for all graph based process languages.

The main benefit of our approach is that it defines one base technology for all types of process languages.

Current software development relies more and more on domain specific languages. A typical Java developer will use quite a few domain specific languages. The XML-files in a project that are input for various frameworks can be considered domain specific languages.

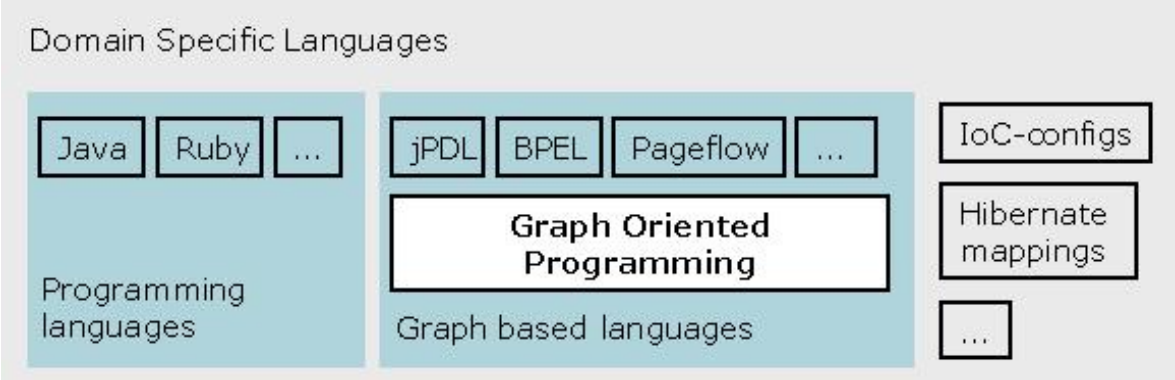

Figure 3.1. Positioning of graph based languages

Domain specific languages for workflow, BPM, orchestration and page-flow are based on the execution of a directed graph. Others like hibernate mapping files, ioc-configuration are not. Graph Oriented Programming is the foundation for all domain specific languages that are based on executing a graph.

Graph Oriented Programming is a very simple technique that describes how graphs can be defined and executed on a plain object-oriented programming language.

In *Section 3.5, "Application domains"*, we'll cover the most often used process languages that can be implemented using Graph Oriented Programming like workflow, BPM, orchestration and pageflow.

3.1.1. Domain specific languages

Each process language can be considered a Domain Specific Language (DSL). The DSL perspective gives developers good insight in how process languages are related to plain object-oriented programming.

This section might give the impression that we're focused solely on programming environments. None is less true. Graph Oriented Programming includes the whole BPM product continuum from API

libraries to fully fledged BPM suite products. BPM suite products are complete software development environments that are centered around business processes. In that type of products, coding in programming languages is avoided as much as possible.

An important aspect of domain specific languages is that each language has a certain grammar. That grammar can be expressed as a domain model. In case of java this is Class, Method, Field, Constructor,... In jPDL this is Node, Transition, Action,... In rules, this is condition, consequence,...

The main idea of DSL is that developers think in those grammars when authoring artifacts for a specific language. The IDE is built around the grammar of a language. Then, there can be different editors to author the artifacts. E.g. a jPDL process has a graphical editor and a XML source view editor. Also there can be different ways to store the same artifact: for jPDL, this could be a process XML file or the serialized object graph of nodes and transition objects. Another (theoretic) example is java: you could use the java class file format on the system. When a user starts the editor, the sources are generated. When a user saves the compiled class is saved....

Ten years ago, most of a developer's time was spend on writing code. Now a shift has taken place towards learning and using domain specific languages. This trend will still continue and the result is that developers will have a big choice between frameworks and writing software in the host platform. JBoss SEAM is a very big step in that direction.

Some of those languages are based on execution of a graph. E.g. jPDL for workflow in Java, BPEL for service orchestration, SEAM pageflow,... Graph Oriented Programming is a common foundation for all these type of domain specific languages.

In the future, for each language, a developer will be able to choose an editor that suites him/her best. E.g. a hard core programmer probably will prefer to edit java in the src file format cause that works really fast. But a less experienced java developer might choose a point and click editor to compose a functionality that will result in a java class. The java source editing will be much more flexible.

Another way of looking at these domain specific languages (including the programming languages) is from the perspective of structuring software. Object Oriented Programming (OOP) adds structure by grouping methods with their data. Aspect Oriented Programming (AOP) adds a way to extract cross cutting concerns. Dependency Injection (DI) and Inversion of Control (IoC) frameworks adds easy wiring of object graphs. Also graph based execution languages (as covered here) can be helpful to tackle complexity by structuring part of your software project around the execution of a graph.

An initial explanation on Domain Specific Languages (DSL) can be found *on Martin Fowler's bliki*[1]. But the vision behind it is better elaborated in *Martin's article about 'Language Workbenches'*[2].

3.1.2. Features of graph based languages

There are numerous graph based process languages. There are big differences in the environment and focus. For instance, BPEL is intended as an XML based service orchestration component on top of an Enterprise Service Bus (ESB) architecture. And a pageflow process language might define how the pages of a web application can be navigated. These are two completely different environments.

Despite all these differences, there are two features that you'll find in almost every process language: support for wait states and a graphical representation. This is no coincidence because it's exactly those two features that are not sufficiently supported in plain Object Oriented (OO) programming languages like Java.

[1] http://www.martinfowler.com/bliki/DomainSpecificLanguage.html
[2] http://www.martinfowler.com/articles/languageWorkbench.html

Graph Oriented Programming is a technique to implement these two features in an OO programming language. The dependency of Graph Oriented Programming on OO programming implies that all concrete process languages, implemented on top of Graph Oriented Programming, will have to be developed in OOP. But this does not mean that the process languages themselves expose any of this OOP nature. E.g. BPEL doesn't have any relation to OO programming and it can be implemented on top of Graph Oriented Programming.

3.1.2.1. Support for wait states

An imperative programming language like Java are used to express a sequence of instructions to be executed by one system. There is no wait instruction. An imperative language is perfect for describing e.g. one request response cycle in a server. The system is continuously executing the sequence of instructions till the request is handled and the response is complete.

But one such request is typically part of a bigger scenario. E.g. a client submits a purchase order, this purchase order is to be validated by a purchase order manager. After approval, the information must be entered in the ERP system. Many requests to the server are part of the same bigger scenario.

So process languages are languages to describe the bigger scenario. A very important distinction we must make here is scenarios that are executable on one system (orchestration) and scenarios that describe the protocol between multiple systems (choreography). The Graph Oriented Programming implementation technique only targets process languages that are executable on one machine (orchestration).

So an orchestration process describes the overall scenario in terms of one system. For example: A process is started when a client submits an order. The next step in the process is the order manager's approval. So the system must add an entry in the task list of the order manager and the **wait** till the order manager provides the required input. When the input is received, the process continues execution. Now a message is sent to the ERP system and again this system will **wait** until the response comes back.

So to describe the overall scenario for one system, we need a mechanism to cope with wait states.

In most of the application domains, the execution must be persisted during the wait states. That is why blocking threads is not sufficient. Clever Java programmers might think about the Object.wait() and Object.notify(); methods. Those could be used to simulate wait states but the problem is that threads are not able to be persisted.

Continuations is a technique to make the thread (and the context variables) able to be persisted. This could be a sufficient to solve the wait state problem. But as we will discuss in the next section, also a graphical representation is important for many of the application domains. And continuations is a technique that is based on imperative programming, so it's unsuitable for the graphical representation.

So an important aspect of the support for wait states is that executions need to be able to be persisted. Different application domains might have different requirements for persisting such an execution. For most workflow, BPM and orchestration applications, the execution needs to be persisted in a relational database. Typically, a state transition in the process execution will correspond with one transaction in the database.

3.1.2.2. Graphical representation

Some aspects of software development can benefit very well from a graph based approach. Business Process Management is one of the most obvious application domains of graph based languages. In that example, the communication between a business analyst and the developer is improved using

the graph based diagram of the business process as the common language. See also *Section 3.5.1, "Business Process Management (BPM)"*.

Another aspect that can benefit from a graphical representation is pageflow. In this case, the pages, navigation and action commands are shown and linked together in the graphical representation.

In Graph Oriented Programming we target graph diagrams that represent some form of execution. That is a clear differentiation with for instance UML class diagrams, which represent a static model of the OO data structure.

Also the graphical representation can be seen as a missing feature in OO programming. There is no sensible way in which the execution of an OO program can be represented graphically. So there is no direct relation between an OO program and the graphical view.

In Graph Oriented Programming, the description of the graph is central and it is a real software artifact like e.g. an XML file that describes the process graph. Since the graphical view is an intrinsic part of the software, it is always in sync. There is no need for a manual translation from the graphical requirements into a software design. The software is structured around the graph.

3.2. Graph Oriented Programming

What we present here is an implementation technique for graph based execution languages. The technique presented here is based on runtime interpretation of a graph. Other techniques for graph execution are based on message queues or code generation.

This section will explain the strategy on how graph execution can be implemented on top of an OO programming language. For those who are familiar with design patterns, it's a combination of the command pattern and the chain of responsibility pattern.

We'll start off with the simplest possible model and then extend it bit by bit.

3.2.1. The graph structure

First of all, the structure of the graph is represented with the classes **`Node`** and **`Transition`**. A transition has a direction so the nodes have leaving- and arriving transitions.

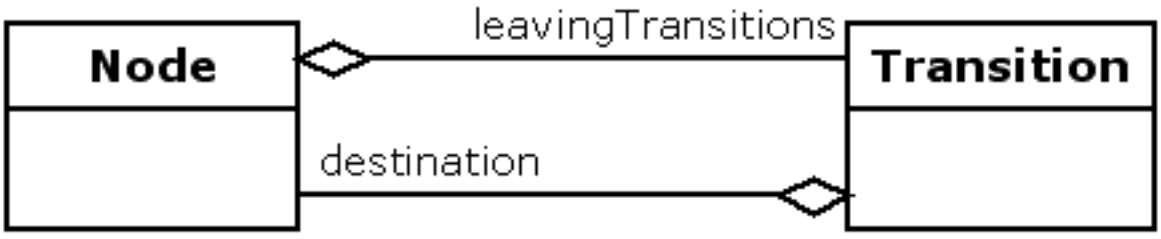

Figure 3.2. Node and Transition classes

A node is a command and has an **`execute`** method. Subclasses of Node are supposed to override the execute method to implement some specific behavior for that node type.

3.2.2. An execution

The execution model that we defined on this graph structure might look similar to finite state machines or UML state diagrams. In fact Graph Oriented Programming can be used to implement those kinds of behaviors, but it also can do much more.

An execution (also known as a token) is represented with a class called **`Execution`**. An execution has a reference to the current node.

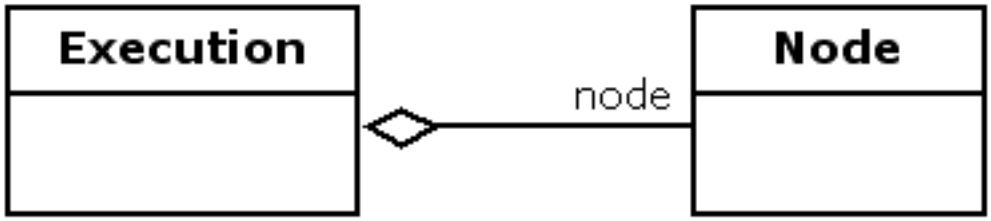

Figure 3.3. The Execution class

Transitions are able to pass the execution from a source node to a destination node with the method **`take`**.

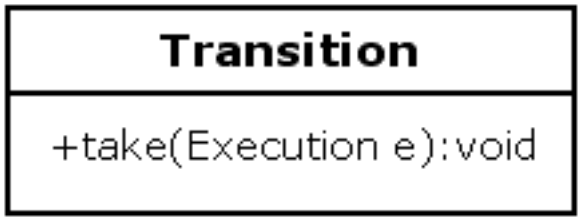

Figure 3.4. The Transition take method

When an execution arrives in a node, that node is executed. The Node's execute method is also responsible for propagating the execution. Propagating the execution means that a node can pass the execution that arrived in the node over one of its leaving transitions to the next node.

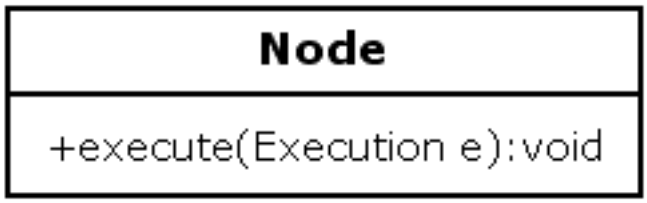

Figure 3.5. The Node execute method

When a node's execute method does not propagate the execution, it behaves as a wait state. Also when a new execution is created, it is initialized in some start node and then waits for an event.

An event is given to an execution and it can trigger the execution to start moving. If the event given to an execution relates to a leaving transition of the current node, the execution takes that transition. The execution then will continue to propagate until it enters another node that behaves as a wait state.

Figure 3.6. The Execution event method

3.2.3. A process language

So now we can already see that the two main features are supported : wait states and a graphical representation. During wait states, an Execution just points to a node in the graph. Both the process graph and the Execution can be persisted: E.g. to a relational database with an O/R mapper like hibernate or by serializing the object graph to a file. Also you can see that the nodes and transitions form a graph and hence there is a direct coupling with a graphical representation.

A process language is nothing more than a set of Node-implementations. Each Node-implementation corresponds with a process construct. The exact behavior of the process construct is implemented by overriding the execute method.

Here we show an example process language with 4 process constructs: a start state, a decision, a task and an end state. This example is unrelated to the jPDL process language.

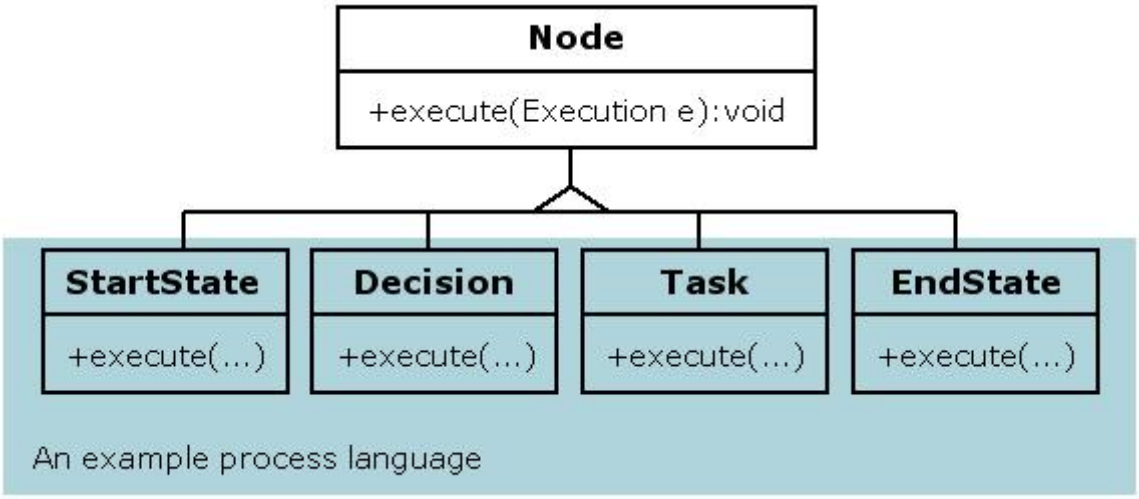

Figure 3.7. An example process language

Concrete node objects can now be used to create process graphs in our example process language.

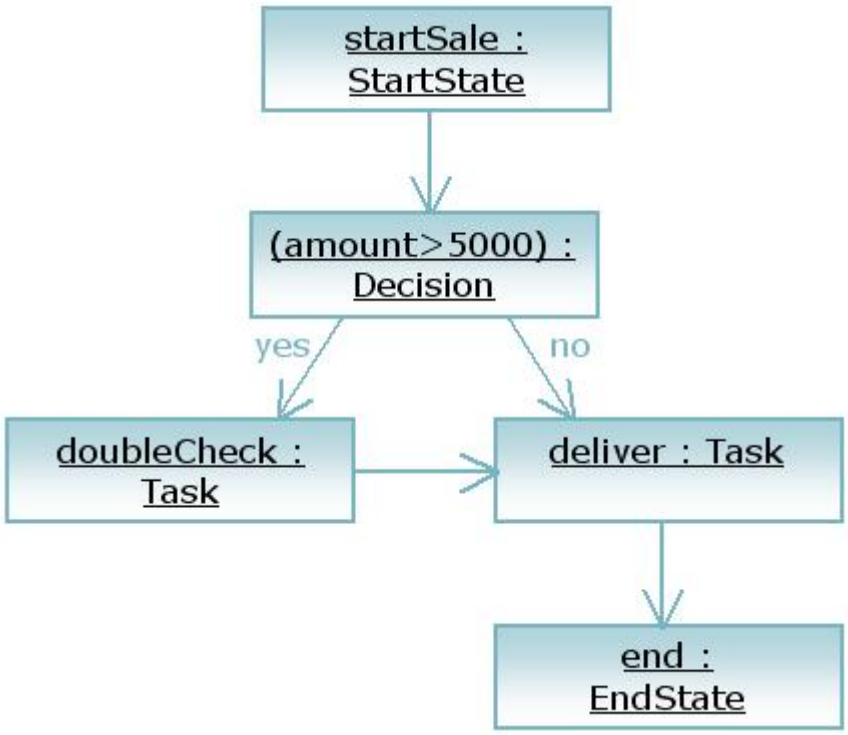

Figure 3.8. An example process

When creating a new execution for this process, we start by positioning the execution in the start node. So as long as the execution does not receive an event, the execution will remain positioned in the start state.

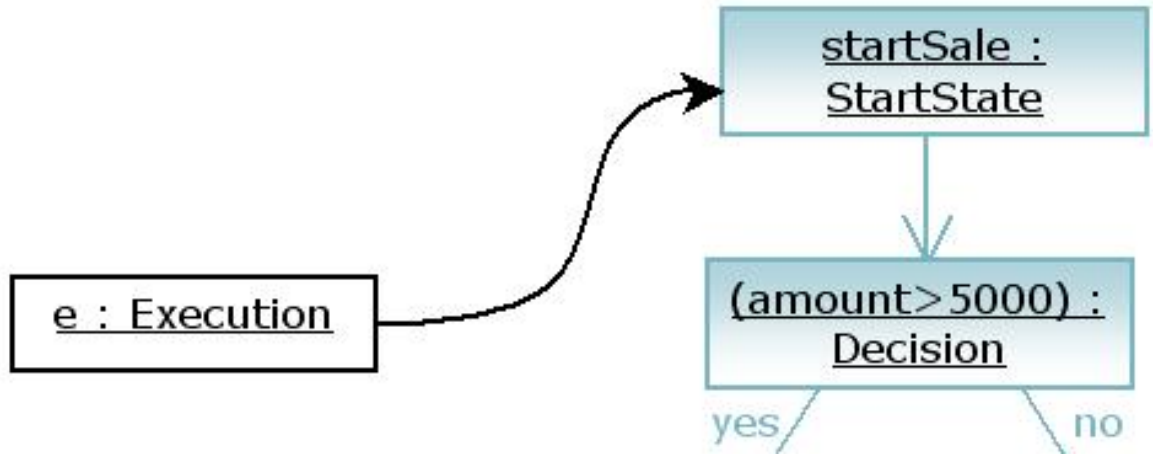

Figure 3.9. A new execution

Now let's look at what happens when an event is fired. In this initial situation, we fire the default event that will correspond with the default transition.

That is done by invoking the event method on the execution object. The event method will propagate find the default leaving transition and pass the execution over the transition by invoking the take method on the transition and passing itself in as a parameter.

The transition will pass on the execution to the decision node and invoke the execute method. Let's assume the decision's execute implementation performs a calculation and decides to propagate the

execution by sending the 'yes'-event to the execution. That will cause the execution to continue over the 'yes' transition and the execution will arrive in the task 'doubleCheck'.

Let's assume that the execute implementation of the doubleCheck's task node adds an entry into the checker's task list and then waits for the checker's input by not propagating the execution further.

Now, the execution will remain positioned in the doubleCheck task node. All nested invocations will start to return until the original event method returns.

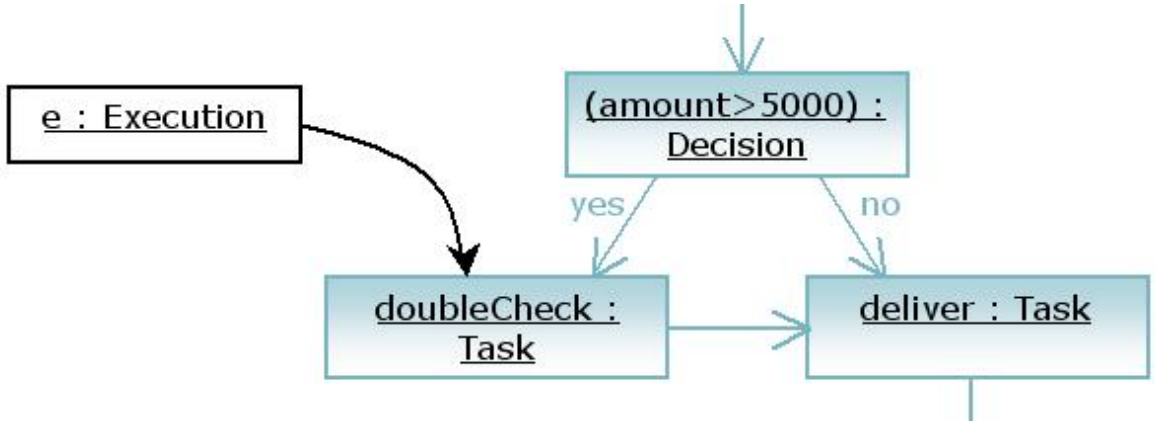

Figure 3.10. An execution in the 'doubleCheck' wait state

3.2.4. Actions

In some application domains there must be a way to include the execution of programming logic without introducing a node for it. In Business Process Management for example this is a very important aspect. The business analyst is in charge of the graphical representation and the developer is responsible for making it executable. It is not acceptable if the developer must change the graphical diagram to include a technical detail in which the business analyst is not interested.

An **`Action`** is also a command with an execute method. Actions can be associated with events.

There are 2 basic events fired by the Node class while an execution is executing: **`node-leave`** and **`node-enter`**. Along with the events that cause transitions to be taken this gives already a good freedom of injecting programming logic into the execution of a graph.

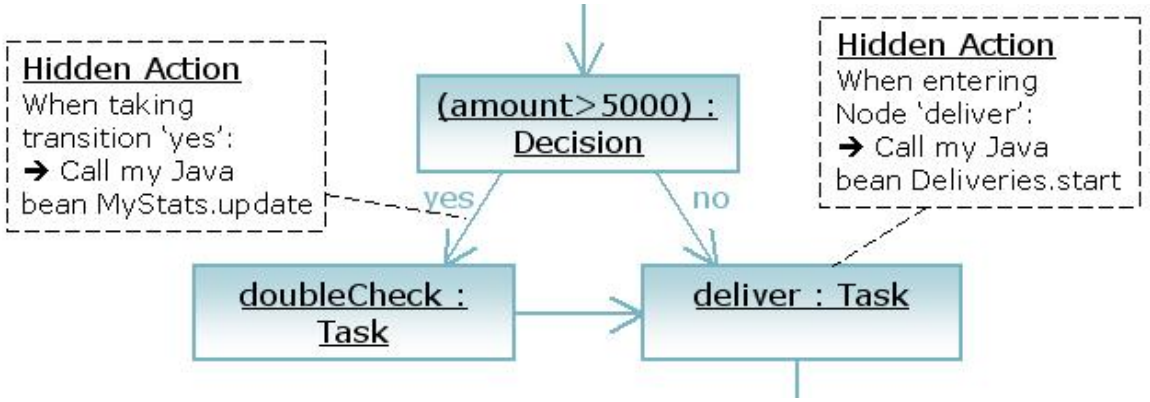

Figure 3.11. Actions that are normally hidden from the graphical view

Each event can be associated with a list of actions. All the actions will be executed when the event fires.

3.2.5. Synchronous execution

The default propagation of execution is synchronous. In *Section 3.3.4, "Asynchronous continuations"* we'll see how this default behavior can be changed.

An execution starts when an event is sent to the execution. That execution will start to propagate over a transition and enters a node. If the node decides to propagate the execution, the take method is invoked on a leaving transition and the execution propagates further. By default, all of these propagations are done as nested method calls. Which means that the original **`event`**-method will only return when the execution has entered a new wait state. So the execution can have travelled over multiple nodes during one invocation of the **`event`**-method.

Typically, a signal method is invoked inside of a transaction. This implies that in one transaction, the execution can potentially move over multiple nodes on the process graph. That results in significant performance benefits over systems that need one transaction per node.

Another benefit of synchronous execution is more options for exception handling. If all nodes are executed synchronously, all propagation's of executions will be nested method invocations. The caller that invoked the signal method will know that a new wait state has been reached without problems when the signal method returns.

3.2.6. Code example

In order for people to get acquainted with the principles of Graph Oriented Programming, we have developed these 4 classes in less then 130 lines of code. You can just read the code to get an idea or you can actually start playing with them and implement your own node types.

Here's the example code:

- ***`Execution.java`***[3]
- ***`Node.java`***[4]
- ***`Transition.java`***[5]
- ***`Action.java`***[6]

You can also *download the whole (297KB) source project*[7] and start playing with it yourself. It includes an eclipse project so just importing it in your eclipse as a project should get you going. Also there are a set of tests that show basic process execution and the advanced graph execution concepts covered in the next section.

3.3. Extending Graph Oriented Programming

The previous section introduced the plain Graph Oriented Programming model in its simplest form. This section will discuss various aspects of graph based languages and how Graph Oriented Programming can be used or extended to meet these requirements.

3.3.1. Process variables

Process variables maintain the contextual data of a process execution. In an insurance claim process, the 'claimed amount', 'approved amount' and 'isPaid' could be good examples of process variables. In many ways, they are similar to the member fields of a class.

[7] http://docs.jboss.com/jbpm/gop/jbpm.gop.zip

Graph Oriented Programming can be easily extended with support for process variables by associating a set of key-value pairs that are associated with an execution. Concurrent execution paths (see *Section 3.3.2, “Concurrent executions”*) and process composition (see *Section 3.3.3, “Process composition”*) will complicate things a bit. Scoping rules will define the visibility of process variables in case of concurrent paths of execution or sub-processes.

'Workflow Data Patterns'[8] is an extensive research report on the types of scoping that can be applied to process variables in the context of sub-processing and concurrent executions.

3.3.2. Concurrent executions

Suppose that you're developing a 'sale' process with a graph based process language for workflow. After the client submitted the order, there is a sequence of activities for billing the client and there's also a sequence of activities for shipping the items to the client. As you can imagine, the billing activities and shipping activities can be done in parallel.

In that case, one execution will not be sufficient to keep track of the whole process state. Let's go through the steps to extend the Graph Oriented Programming model and add support for concurrent executions.

First, let's rename the execution to an execution path. Then we can introduce a new concept called a process execution. A process execution represents one complete execution of a process and it contains many execution paths.

The execution paths can be ordered hierarchically. Meaning that one root execution path is created when a new process execution is instantiated. When the root execution path is forked into multiple concurrent execution paths, the root is the parent and the newly created execution paths are all children of the root. This way, implementation of a join can become straightforward: the implementation of the join just has to verify if all sibling-execution-paths are already positioned in the join node. If that is the case, the parent execution path can resume execution leaving the join node.

While the hierarchical execution paths and the join implementation based on sibling execution paths covers a large part of the use cases, other concurrency behavior might be desirable in specific circumstances. For example when multiple merges relate to one split. In such a situation, other combinations of runtime data and merge implementations are required.

[8] http://is.tm.tue.nl/research/patterns/download/data_patterns%20BETA%20TR.pdf

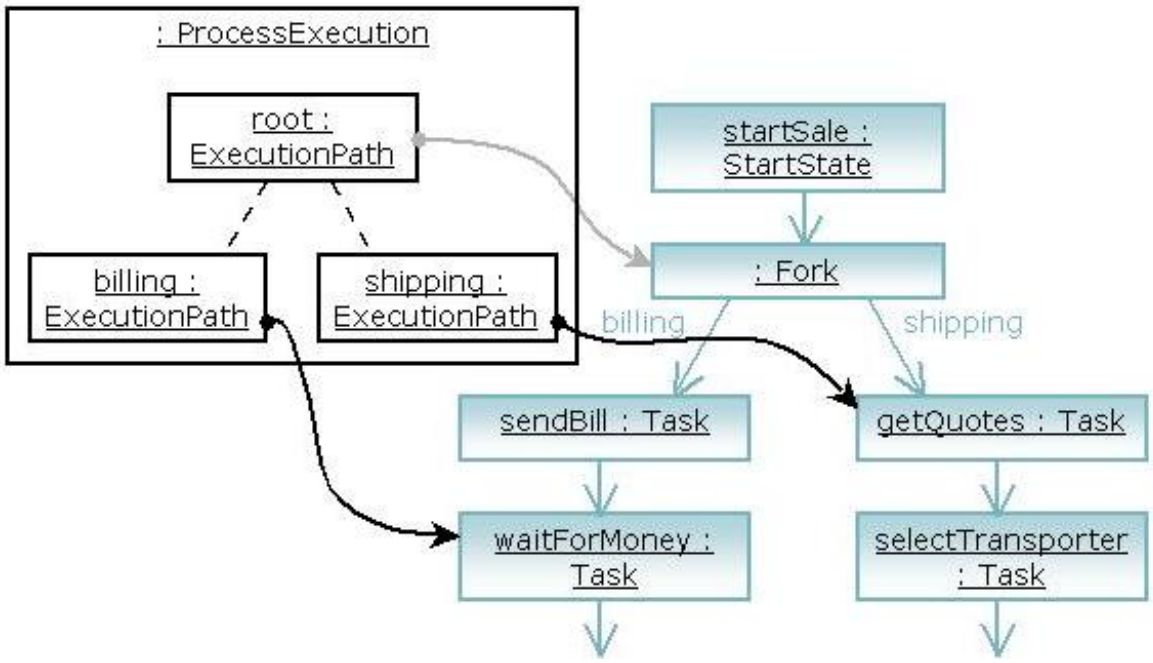

Figure 3.12. Concurrent paths of execution

Multiple concurrent paths of execution are often mixed up with multi-threaded programming. Especially in the context of workflow and BPM, these are quite different. A process specifies a state machine. Consider for a moment a state machine as being always in a stable state and state transitions are instantaneous. Then you can interpret concurrent paths of execution by looking at the events that cause the state transitions. Concurrent execution then means that the events that can be handled are unrelated between the concurrent paths of execution. Now let's assume that state transitions in the process execution relates to a database transition (as explained in *Section 3.3.5, "Persistence and Transactions"*), then you see that multi-threaded programming is actually not even required to support concurrent paths of execution.

3.3.3. Process composition

Process composition is the ability to include a sub process as part of a super process. This advanced feature makes it possible to add abstraction to process modeling. For the business analyst, this feature is important to handle break down large models in smaller blocks.

The main idea is that the super process has a node in the graph that represents a complete execution of the sub process. When an execution enters the sub-process-node in the super process, several things are to be considered:

- First of all, a new execution is created for the sub process.
- Optionally some of information stored in the process variables of the super process can be injected from the super process execution into the sub process execution. The most easy form is that the sub process node is configured with a set of variables that are just copied from the super process variables to the sub process variables.
- The start-node of the sub process should have only one leaving transition. Process languages that support multiple leaving transitions must have a mechanism to choose one of those transitions based on the process variables of the super process.

- The sub process execution is launched by sending an event that corresponds to the default leaving transition of its start state.

After the sub process entered a wait state, the super process execution will be pointing to the sub-process-node and the sub process execution will be pointing to some wait state.

When the sub process execution finishes, the super process execution can continue. The following aspects need to be considered at that time:

- Process variable information may need to be copied back from the sub process execution into the super process execution.
- The super process execution should continue. Typically, process languages allow only one leaving transition on a sub process node. In that case the super process execution is propagated over that default single leaving transition.
- In case a sub process node is allowed more than one leaving transition, a mechanism has to be introduced to select a leaving transition. This selection can be based on either the sub process execution's variables or the end state of the sub process (a typical state machine can have multiple end states).

WS-BPEL has an implicit notion of sub-processing, rather then an explicit. An **`invoke`** will start a new sub process. Then the super process will have a **`receive`** activity that will wait till the sub process ends. So the usual **`invoke`** and **`receive`** are used instead of a special activity.

3.3.4. Asynchronous continuations

Above, we saw that the default behavior is to execute processes synchronously until there is a wait state. And typically this overall state-change is packaged in one transaction. In this section, you'll see how you can demarcate transaction boundaries in the process language. Asynchronous continuations means that a process can continue asynchronously. This means that the first transaction will send a message. That message represents a continuation command. Then the message receiver executes the command in a second transaction. Then the process has continued its automatic execution, but it was split over 2 transactions.

To add asynchronous continuations to graph oriented programming, a messaging system is required. Such a system that integrates with your programming logic and allows for transactional sending and receiving of messages. Messaging systems are also know as message oriented middleware (MOM) and Java Message Service (JMS) is the standard API to use such systems.

There are 3 places where execution can be continued asynchronously:

- Just before the node's execute method. Which is after entering the node.
- When execution is about to be propagated over a transition. Which is before leaving a node.
- Every action can be executed asynchronously as well.

Let's consider the first situation in detail as it is indicated in the following figure. Suppose some event caused an execution to start propagating over the graph and now a transition is about to invoke the execute method on the 'generatePdf' node. Instead of invoking the execute method on the 'generatePdf' node directly, a new command message is being created with a pointer to the execution. The command message should be interpreted as "continue this execution by executing the node". This message is sent over the message queue to the command executor. The command executor

takes the message from the queue and invokes the node's execute method with the execution as a parameter.

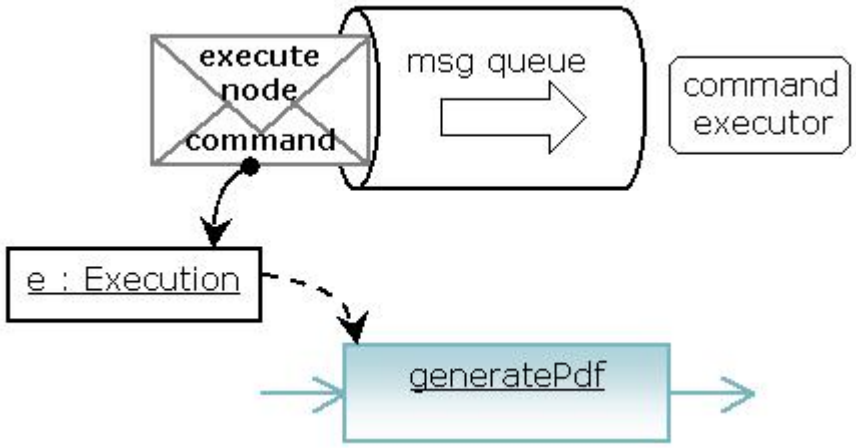

Figure 3.13. Asynchronous continuation

Note that there are two separate transactions involved now. One transaction that originated from the original event. That transaction contains moving the execution in the 'generatePdf' node and sending the command message. In a second transaction, the command message was consumed and the node's execute method was invoked with the execution as a parameter. Between the two transactions, the execution should be blocked for incoming events.

3.3.5. Persistence and Transactions

Both process definition information (like Node, Transition and Action) and execution information (like Execution) can be stored in a relational database. An ORM solution (like eg Hibernate/EJB3) can be used to perform the mapping between the database records and the OOP objects.

All process definition information is static. Hence it can be cached in memory. This gives a serious performance boost. Only the runtime execution data will have to be loaded from the DB in each transaction.

A transaction typically corresponds to the event method on the Execution. A transaction starts when an event is being processed. The event method will trigger execution to continue till a new wait state is reached. When that happens, the Execution's event method returns and the transaction can be ended.

The overall change of the event method invocation is that the Execution has moved it's node pointer from one node to another. The ORM solution can calculate the difference between the original database state and the updated java objects. Those changes are then flushed to the database at the end of the Execution's event method. In our example here this will be a SQL update statement on the execution, that sets the node pointer to the new (wait-state)node.

ORM solutions like hibernate/EJB3 work with a different set of objects in each session. This implies that all access to Node implementations is serialized and removes the necessity to write thread safe code as long as the node uses the execution data (and not static variables, for instance).

3.3.6. Services and environment

Nodes might want to make use of pluggable services or new node implementations might want to use new services, unknown at design time. To accommodate this, a services framework can be added to Graph Oriented Programming so that nodes can access arbitrary services and configurations.

Basically, there are 2 options:

- Passing down an execution context object (that would wrap the Execution object that is passed in the explanation above)
- A thread local execution context

The execution context contains access to services that are made available by 'the environment'. The environment is the client code (the code that invokes the **`Execution.event(String)`** plus an optional container in which this client code runs.

Examples of services are a timer service, an asynchronous messaging service, a database service (java.sql.Connection),...

3.4. Considerations

3.4.1. Runtime data isolation

Graph oriented programming clearly separates the definition data (nodes, transitions and actions) from the runtime data (execution).

So instead of just propagating the execution that entered the node, any node implementation can decide to rearrange the whole runtime data that represents the execution. This creates a lot of flexibility for implementing different flavors of fork.split and join/merge behavior.

Also, the definition information is static and never changes. This is important for all kinds of performance optimizations.

3.4.2. GOP compared to other techniques

In this section we describe how graph oriented programming compares to other implementation techniques used for graph based execution languages.

In MOM based execution engines, an execution is represented by a message that travels along message queues. Each node in a process graph is represented by a message queue in the system. Actually, graph oriented programming is a super-set of MOM based execution. In GOP, by default, the calculation to move an execution from one wait state to another is done synchronously. Later in this paper, we'll cover the asynchronous continuations extension that explains how MOM can be used to make one step in the process asynchronous. So MOM based execution is similar to graph oriented programming where all the nodes are executed asynchronously.

Another technique used to implement workflow, BPM and orchestration systems is code generation. In that approach, the graph based process is translated into imperative programming logic like Java. The generated programming logic has a method for each external trigger that can be given after a wait state. Those methods will calculate the transition to a new wait state. This technique is limitated in process versioning capabilities and in practice, the code generation has proved to be impractical and a bottleneck in the software development process.

3.4.3. GOP compared to petri nets

The academic world, for a long time, has focused on petri nets for workflow and business process modeling, mainly because petri nets was the only mathematically defined model that supports

concurrent paths of execution. Because of the mathematical foundations, many interesting algorithms for validation and completeness could be defined.

The biggest difference between petri nets and graph oriented programming is their nature. Petri nets is a mathematical model, while graph oriented programming is an implementation technique or a design pattern.

Graph oriented programming can be used to implement petri nets. Petri net places and petri net transitions can be implemented as two different node types. Petri net arcs correspond to GOP transitions. A petri net token corresponds to a GOP execution.

The higher level extensions that have been defined on top of petri nets can also be defined in terms of graph oriented programming.

Graph oriented programming by itself does not support analytical algorithms as they are defined on petri nets. That is because graph oriented programming does not’t have a concrete interpretation. Analytical algorithms can only be defined on models that have a deterministic design time interpretation. Graph oriented programming on the other hand also supports nodes that have an undeterministic design time interpretation. GOP node implementations can potentially do any type of calculation at runtime and decide only then how the execution is propagated. Analytical algorithms can only be defined on concrete process languages, for which the nodes implementations give a deterministic design-time interpretation to the node types.

3.5. Application domains

3.5.1. Business Process Management (BPM)

3.5.1.1. Different aspects of BPM

The goal of BPM is to make an organization run more efficient. The first step is analyzing and describing how work gets done in an organization. "Defining a business process" is a description of the way that people and systems work together to get a particular job done. Once business processes are described, the search for optimizations can begin.

Sometimes business processes have evolved organically and merely looking at the overall business process shows some obvious inefficiencies. Searching for modifications that make a business process more efficient is called Business Process Re-engineering (BPR). Once a large part of a business process is automated, statistics and audit trails can help to find and identify these inefficiencies.

Another way to improve efficiency can be to automate whole or parts of the business process using information technology.

Automating and modifying business processes are the most common ways of making an organization run more efficient. Important is to note that those are parts of business process management in general.

Managers continuously break down jobs into steps to be executed by their team members. For example a software development manager that organizes a team-building event. In that case, the description of the business process might be done only in the head of the manager. Other situations like handling an insurance claim for a large insurance company require a more formal approach to BPM.

The total gain that can be obtained from managing business processes is the efficiency improvements times the number of executions of the process. The cost of managing business processes formally is the extra effort that is spent on analyzing, describing, improving and automating the business processes. So that cost has to be taken into consideration when determining which processes will be selected for formal management and/or automation. This explains the focus on procedures with a high recurrence rate.

3.5.1.2. Goals of BPM systems

The main goal of BPM systems is to facilitate the automation of business processes. In building software for business processes, two roles can be distinguished: The business analyst and the developer. In small teams, these two roles can of course be fulfilled by one person. The business analyst studies and describes the business process and specifies the software requirements, while the developer creates executable software.

Traditional BPM suites try to start from the business analyst's model and work their way down towards executable software. They try to minimize the need for technical skills so that the business analyst can produce executable software. All of this is centralized around the graphical diagram so inevitably, technical details ripple through in the analyst's world.

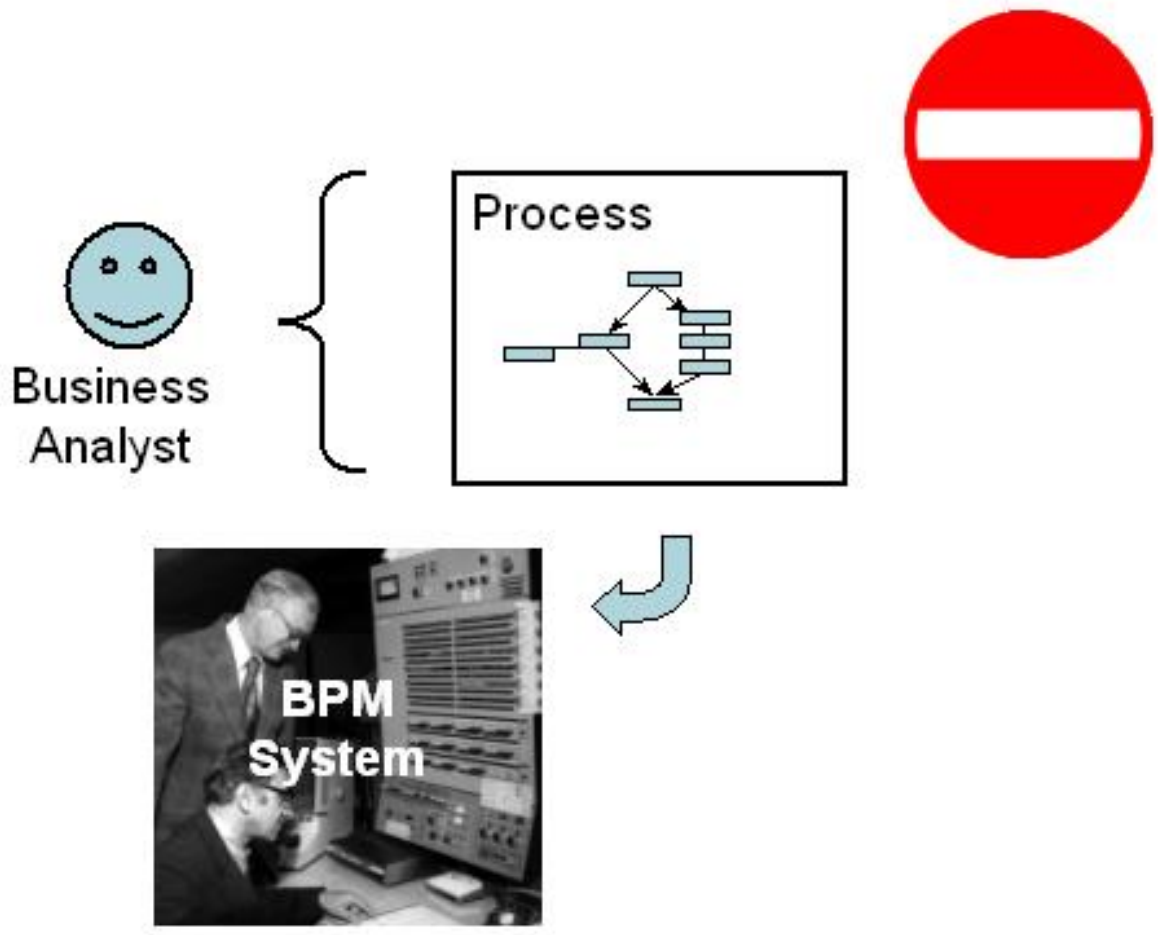

Figure 3.14. Traditional BPM approach

In our vision, the central idea is that the business analyst and the developer communicate in a common language with the help of the graphical view of the process. Technical skills will always be

necessary when developing software. The business analyst is responsible for the graphical view and should not be forced to deal with technical aspects of the process. Without those technical aspects the process will not be fully defined and hence it won't be executable. The developer is responsible for the technical implementation aspects. Technical aspects should not require diagram changes.

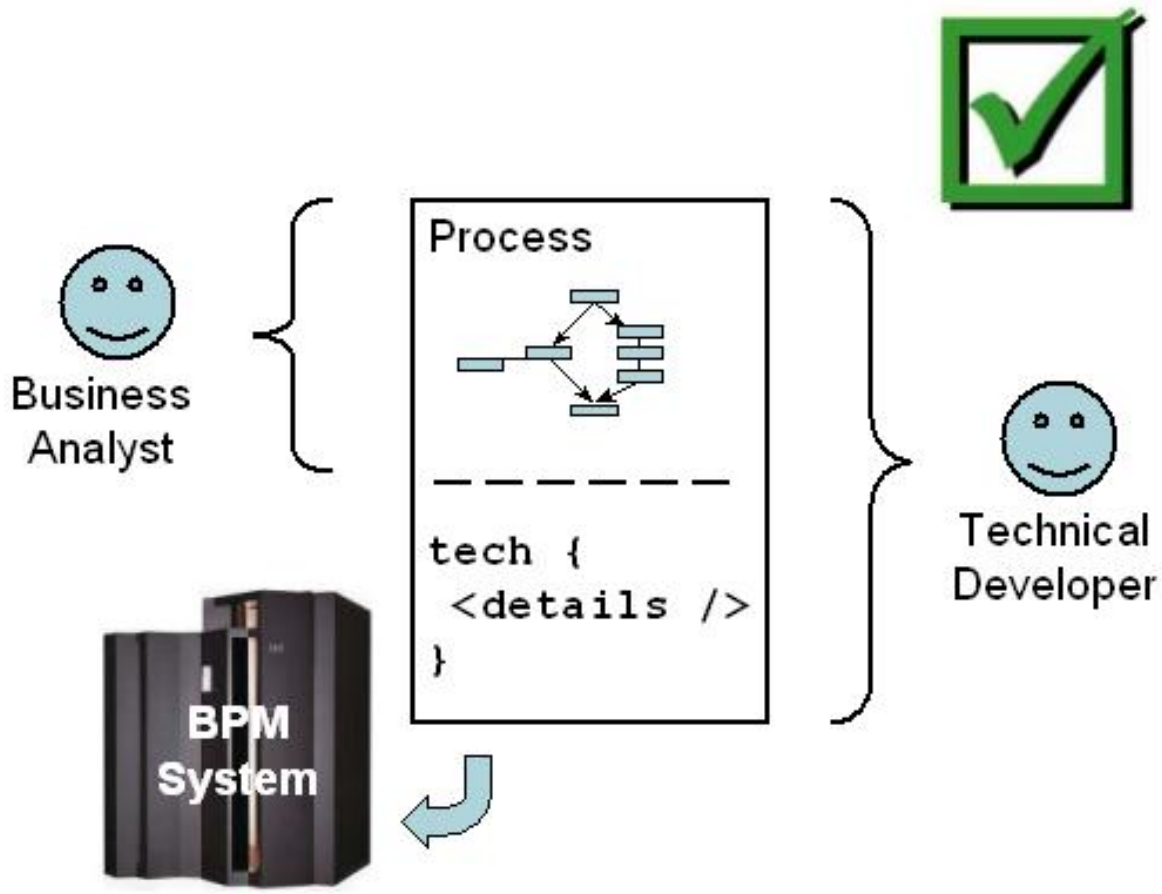

Figure 3.15. Improved BPM approach

3.5.2. Service orchestration

The most recognized name in service orchestration languages is BPEL. Service orchestration is to be seen in the context of an Enterprise Service Bus. An enterprise service bus is a central communication backbone on a corporate level. It integrates many diverse systems and it is based on XML technology.

Suppose you have services A, B and C on your enterprise service bus. Service orchestration is a graph based execution language for writing a new services as a function of existing services. E.g. A new service D can be written as a function of existing services A, B and C in an orchestration script.

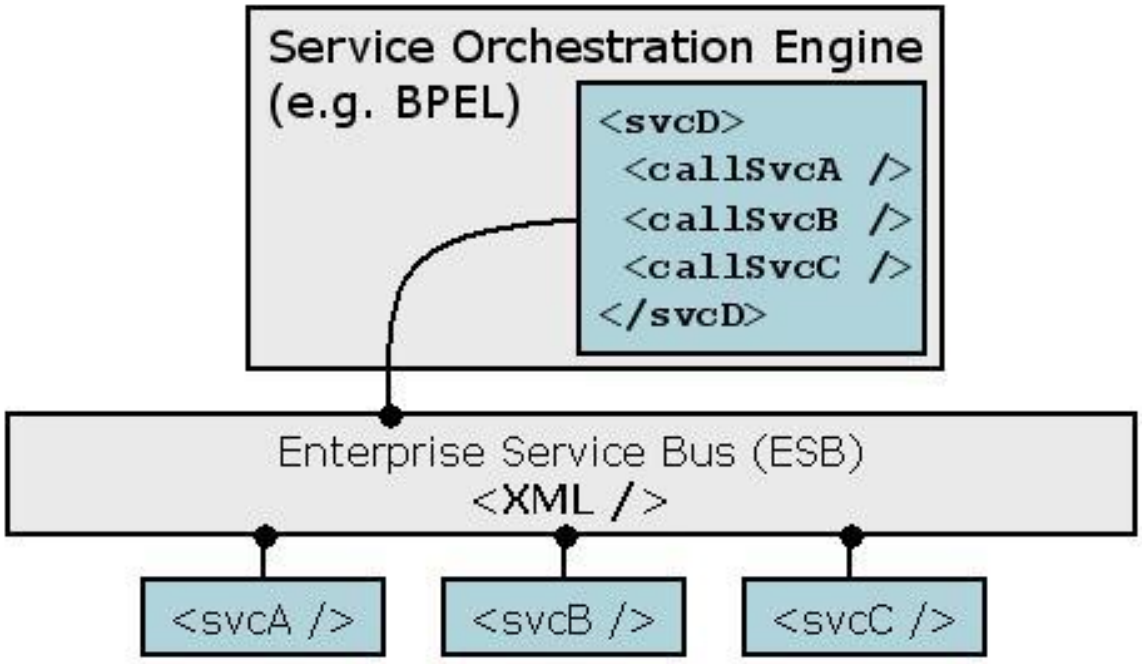

Figure 3.16. Service

3.6. Embedding graph based languages

When the BPM engine can be completely integrated into a software development project and when even the BPM engine's database tables are integrated into the project's database, then we speak of an Embeddable BPM engine. That is the goal we target with Graph Oriented Programming: a common foundation for implementing graph based languages.

3.7. Market

3.7.1. The ultimate process language

Traditionally, the vendors have been searching for the ultimate process language. The approach is to specify a process language as a set of constructs. Each construct has a graphical representation and a runtime behavior. In other words, each construct is a node type in the process graph. And a process language is just a set of node constructs.

The idea was that the vendors were searching for the best set of process constructs to form a universally applicable process language. This vision is still found a lot today and we call it searching for the ultimate process language.

We believe that the focus should not be on trying to find the ultimate process language, but rather in finding a common foundation that can be used as a basis for process languages in different scenarios and different environment. Graph Oriented Programming as we present it next is to be seen as such a foundation.

3.7.2. Fragmentation

The current landscape of workflow, BPM and orchestration solutions is completely fragmented. In this section we'll describe two dimensions in this fragmentation. The first dimension is called the BPM

product continuum and it's shown in the next picture. The term was originally coined by Derek Miers and Paul Harmon in 'The 2005 BPM Suites Report'.

On the left, you can see the programming languages. This side of the continuum is targeted towards the IT developers. Programming languages are the most flexible and it integrates completely with the other software developed for a particular project. But it takes quite a bit of programming to implement a business process.

On the right, there are the BPM suites. These BPM suites are complete software development environments targeted to be used by business analysts. Software is developed around business processes. No programming has to be done to create executable software in these BPM suites.

Figure 3.17. The BPM product continuum.

Traditional products mark 1 spot in the BPM product continuum. To be complete, these products tend to aim for the far right of the continuum. This is problematic because it results in monolithic system that is very hard to integrate into a project combines plain OOP software with business processes.

Graph Oriented Programming can be built as a simple library that integrates nice with plain programming environment. On the other hand, this library can be packaged and pre-deployed on a server to become a BPM server. Then other products are added and packaged together with the BPM server to become a complete BPM suite.

The net result is that solutions based on Graph Oriented Programming can target the whole continuum. Depending on the requirements in a particular project, the BPM suite can be peeled and customized to the right level of integration with the software development environment.

The other dimension of fragmentation is the application domain. As show above, a BPM application domain is completely different from service orchestration or pageflow. Also in this dimension, traditional products target one single application domain, where Graph Oriented Programming covers the whole range.

If we set this out in a graph, this gives a clear insight in the current market fragmentation. In the graph based languages market, prices are high and volumes are low. Consolidation is getting started and this technology aims to be a common foundation for what is now a fragmented and confusing market landscape.

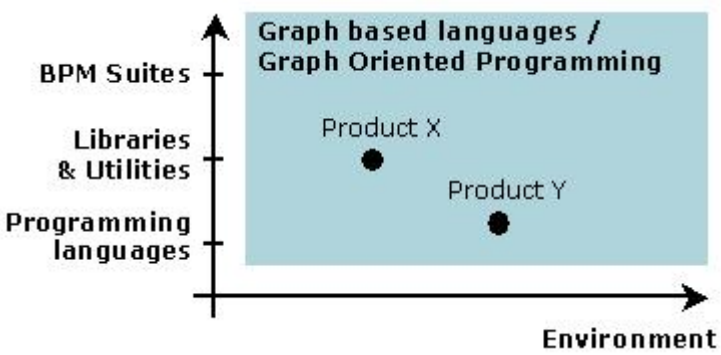

Figure 3.18. Two dimensions of fragmentation.

Deployment

jPDL is an embeddable BPM engine, which means that you can take the jPDL libraries and embed it into your own Java project, rather then installing a separate product and integrate with it. One of the key aspects that make this possible is minimizing the dependencies. This chapter discusses the jbpm libraries and their dependencies.

4.1. jBPM libraries

jbpm-jpdl.jar is the library with the core jpdl functionality.

jbpm-identity.jar is the (optional) library containing an identity component as described in *Section 10.11, "The identity component"*.

4.2. Java runtime environment

jBPM 3 requires J2SE 1.4.2+

4.3. Third party libraries

All the libraries on which jPDL might have a dependency, are located in the lib directory.

In a minimal deployment, you can create and run processes with jBPM by putting only the commons-logging and dom4j library in your classpath. Beware that persisting processes to a database is not supported. The dom4j library can be removed if you don't use the process xml parsing, but instead build your object graph programatically.

Library	Usage	Description
commons-logging.jar	logging in jbpm and hibernate	The jBPM code logs to commons logging. The commons logging library can be configured to dispatch the logs to e.g. java 1.4 logging, log4j, ... See *the Apache commons user guide*[1] for more information on how to configure commons logging. if you're used to log4j, the easiest way is to put the log4j lib and a log4j.properties in the classpath. commons logging will automatically detect this and use that configuration.
dom4j.jar	process definitions and hibernate persistence	xml parsing

Table 4.1. Minimal Dependencies

A typical deployment for jBPM will include persistent storage of process definitions and process executions. In that case, jBPM does not have any dependencies outside hibernate and its dependent libraries.

Of course, Hibernate's required libraries depend on the environment and what features you use. For details refer to the hibernate documentation. The next table gives an indication for a plain standalone POJO development environment.

Library	Usage	Description
hibernate3.jar	hibernate persistence	the best O/R mapper
antlr-2.7.6rc1.jar	used in query parsing by hibernate persistence	parser library
cglib.jar	hibernate persistence	reflection library used for hibernate proxies
commons-collections.jar	hibernate persistence	
asm.jar	hibernate persistence	asm byte code library

Table 4.2. Typical Dependencies

The beanshell library is optional. If you don't include it, you won't be able to use the beanshell integration in the jBPM process language and you'll get a log message saying that jbpm couldn't load the Script class and hence, the script element won't be available.

Library	Usage	Description
bsh.jar	beanshell script interpreter	Only used in the script's and decision's. When you don't use these process elements, the beanshell lib can be removed, but then you have to comment out the Script.hbm.xml mapping line in the **`hibernate.cfg.xml`**

Table 4.3. Optional Dependencies

4.4. Web application

In the deploy directory of the downloads, you can find **jbpm-console.war**. That web console contains the jPDL libraries, configuration files and the required libraries to run this web application on JBoss.

This war file does NOT include the hibernate libraries. That is because JBoss already includes the hibernate libraries. To run this web application on other servers like Tomcat, all you have to do is get the hibernate libraries in the **`WEB-INF/lib`** directory in the war file. Simplest way to do that is to use the ant build script in this directory.

Also, this war file can give you a good indication of how you could deploy jbpm libraries and configuration files into your own web application.

In the web.xml of this web application, the **`JobExecutorServlet`** is configured. This will start the JobExecutor when the **`jbpm-console.war`** is deployed. The JobExecutor serves as the basis for executing timers and asynchronous messages on the standard java platform.

4.5. Enterprise archive

In the deploy directory of the downloads, you can find **jbpm-enterprise.ear**. That J2EE 1.4 compliant enterprise archive includes: jPDL libraries, jBPM configuration files, the jBPM web console, and a couple of enterprise beans. In this package, jBPM is configured for usage in an application server like e.g. JBoss. Asynchronous messaging service is here bound to JMS and the scheduler service is bound to the EJB Timer Service. So here in this .ear file, there is no JobExecutor started. Also the hibernate session that jBPM uses is configured to participate in the overall JTA transaction.

Within **`jbpm-enterprise.ear`** there are the following files:

- **jbpm-console.war** - the jbpm console web application
- **jbpm-enterprise.jar** - several jBPM EJB compnents
- **lib/jbpm-configs.jar** - jBPM configuration files
- **lib/jbpm-identity.jar** - jBPM identity component classes
- **lib/jbpm-jpdl.jar** - jBPM jpdl classes
- **meta-inf/application.xml**

jbpm-enterprise.jar contains the following EJB components:

- CommandListenerBean - a Message Driven Bean that listens on the jbpmCommandQueue for jBPM command messages.
- CommandServiceBean - a Stateless Session Bean that executes jBPM Commands.
- JobListenerBean - a Message Driven Bean that listeners on the jbpmJobQueue for jBPM job messages to support asynchronous continuations.
- TimerServiceBean - a TimerBean that implements the jBPM timer service.

These beans are J2EE 1.4 / EJB 2.1 compliant, to allow them to be deployed on a variety of application servers. Note however that jBPM only provide deployment descriptors for JBoss Application Servers. All beans are deployed without specifying transaction-attribute, therefore by default they have transaction-attribute "Required".

The CommandListenerBean delegates Command execution to the CommandServiceBean, which in turn delegates to a CommandServiceImpl class that executes the command class by calling it's execute method.

The source for these classes is in: src/enterprise. The javadocs in doc/javadoc-enterprise.

jbpm-enterprise.jar also contains JmsMessageServiceFactoryImpl, which is responsible for sending Jobs as JMS messages to support asynchronous continuations.

jbpm-configs.jar contains the following files:

- **jbpm.cfg.xml**
- **jbpm.mail.templates.xml**
- **hibernate.cfg.xml** includes the following configuration items that may require modification to support other databases or application servers.

```
<!-- hibernate dialect -->
    <property name="hibernate.dialect">
        org.hibernate.dialect.HSQLDialect
    </property>

    <!-- JDBC connection properties (begin) ===
    <property
 name="hibernate.connection.driver_class">org.hsqldb.jdbcDriver</
property>
```

```
    <property name="hibernate.connection.url">jdbc:hsqldb:mem:jbpm</
property>
    <property name="hibernate.connection.username">sa</property>
    <property name="hibernate.connection.password"></property>
    ==== JDBC connection properties (end) -->

    <property name="hibernate.cache.provider_class">
        org.hibernate.cache.HashtableCacheProvider
    </property>

    <!-- JBoss transaction manager lookup (begin) -->
    <property name="hibernate.transaction.manager_lookup_class">
        org.hibernate.transaction.JBossTransactionManagerLookup
    </property>
    <!-- JBoss transaction manager lookup (end) -->

    <!-- DataSource properties (begin) -->
    <property name="hibernate.connection.datasource">
        java:/JbpmDS
    </property>
    <!-- DataSource properties (end) -->

    <!-- JTA transaction properties (begin) ===
    <property
 name="hibernate.transaction.factory_class">org.hibernate.transaction.JTATransactionFac
property>
    <property name="jta.UserTransaction">java:comp/UserTransaction</
property>
    ==== JTA transaction properties (end) -->

    <!-- CMT transaction properties (begin) -->
    <property name="hibernate.transaction.factory_class">
        org.hibernate.transaction.CMTTransactionFactory
    </property>
    <property name="hibernate.transaction.manager_lookup_class">
        org.hibernate.transaction.JBossTransactionManagerLookup
    </property>
    <!-- CMT transaction properties (end) -->

    <!-- logging properties (begin) ===
    <property name="hibernate.show_sql">true</property>
    <property name="hibernate.format_sql">true</property>
    <property name="hibernate.use_sql_comments">true</property>
    ==== logging properties (end) -->
```

You may replace the hibernate.dialect with one that corresponds to your database management system.

You may replace the HashtableCacheProvider with other Hibernate supported cache providers, such as EhCache.

The transaction.manager.lookup may be replace with values appropriate to other applications servers, e.g. WebSphereTransactionManagerLookup or WebLogicTransactionManagerLookup when deploying to those application servers.

Similarly, if deploying on another application server you must change the name of the hibernate.connection.datasource to the jndi name of the datasource on that application server.

Out-of-the-box jBPM is configured to use CMTTransactionFactory. CMTTransactionFactory always assumes that the container has started a JTA transaction. This will be true if you use the jBPM CommandListener or CommandService beans, or your own EJBs that use container managed transaction. If this is not always the case, then change this configuration to use the JTATransactionFactory. When JTATransactionFactory is configured, Hibernate will use the JTA transaction if it already exists, but will start a JTA transaction if it does not.

- **`jbpm.cfg.xml`** included the following configuration items:

```
<jbpm-context>
  <service name="persistence">
    <factory>

  <bean class="org.jbpm.persistence.db.DbPersistenceServiceFactory">
        <field name="isCurrentSessionEnabled"><true /></field>
        <field name="isTransactionEnabled"><false /></field>
      </bean>
    </factory>
  </service>
  <service name="message">
    <factory>
      <bean class="org.jbpm.msg.jms.JmsMessageServiceFactoryImpl">
        <field name="connectionFactoryJndiName">
          <string value="java:/JmsXA"/>
        </field>
        <field name="destinationJndiName">
          <string value="queue/JbpmJobQueue"/>
        </field>
      </bean>
    </factory>
  </service>
<service name="scheduler"
  factory="org.jbpm.scheduler.ejbtimer.EjbSchedulerServiceFactory" />
```

isCurrentSessionEnabled true means jBPM will request Hibernate to use the current session associated with the current transaction. If there is no current transaction, an exception will be thrown stating no session is active. In this case, you may want to set this isCurrentSessionEnabled to false, and inject the current session into the JbpmContext via the JbpmContext.setSession(session) method. This will also insure that jBPM uses the same Hibernate session as other parts of your application. Note, the Hibernate session can be injected into a stateless session bean via a persistence context, for example.

isTransactionEnabled true means jBPM will begin Hibernate transaction upon JbpmConfiguraiton().createJbpmContext and commit Hibernate transactions and close Hibernate

sessions upon jbpmContext.close()..This is NOT the desired behavior when jBPM is deployed as an ear, hence the value of isTransactionEnabled is set to false by default in this configuration.

4.6. The jPDL Runtime and Suite

4.6.1. The runtime

The jPDL runtime is all you need to get started with jPDL: jpdl libraries, third party libraries, examples and documentation. It doesn't include the graphical designer and web console tooling, which is added in the suite package.

Directory	Content
config	Contains all the configuration files. Note that for easy testing and development, the current hibernate configuration points to the in-memory jbpm database.
db	Contains the scripts to create the jPDL tables in your DB. It includes a copy of the wiki page about database compatibility.
doc	Contains the userguide and the javadocs for the jpdl sources and identity sources
examples	Each example is a separate project that you can compile and run with ant or eclipse.
lib	All the third party libs and their licenses.
src	The sources for jpdl and the identity components.

Table 4.4. jPDL runtime directories

4.6.2. The suite

The jPDL suite is an extension of the jPDL runtime with 2 tools: a graphical designer plugin for eclipse and a JBoss server that is pre-configured with a deployed version of the jPDL runtime and console web app. The included tools are all pre-configured to work nicely together out of the box.

Directory	Content
designer	The designer is the eclipse plugin that allows for graphical process editing of jPDL process files. Look in the designer/readme.html for more instructions on installing the designer.
server	The server is actually a JBoss application server which has the jPDL runtime and the jPDL console web application deployed.

Table 4.5. jPDL suite extra directories

4.6.3. Configuring the logs in the suite server

If you want to see debug logs in the suite server, update file **`jpdl-suite-home/server/server/jbpm/config/log4j.xml`** Look for

```
<!-- ============================== -->
<!-- Append messages to the console -->
<!-- ============================== -->
```

```
<appender name="CONSOLE" class="org.apache.log4j.ConsoleAppender">
  <errorHandler class="org.jboss.logging.util.OnlyOnceErrorHandler"/>
  <param name="Target" value="System.out"/>
  <param name="Threshold" value="INFO"/>
```

And in param **`Threshold`**, change **`INFO`** to **`DEBUG`**.

Then you'll get debug logs of all the components. To limit the number of debug logs, look a bit further down that file until you see 'Limit categories'. You might want to add thresholds there for specific packages like e.g.

```
<category name="org.hibernate">
  <priority value="INFO"/>
</category>

<category name="org.jboss">
  <priority value="INFO"/>
</category>
```

4.6.4. Debugging a process in the suite

First of all, in case you're just starting to develop a new process, it is much easier to use plain JUnit tests and run the process in memory like explained in *Chapter 2, Tutorial*.

But if you want to run the process in the console and debug it there here are the 2 steps that you need to do:

1) in **`jpdl-suite-home/server/server/bin/run.bat`**, somewhere at the end, there is a line like this:

```
rem set JAVA_OPTS=-Xdebug -
Xrunjdwp:transport=dt_socket,address=8787,server=y,suspend=y %JAVA_OPTS%
```

For backup reasons, just start by making a copy of that line, then remove the first '**`rem`**' and change **`suspend=y`** to **`suspend=n`**. Then you get something like

```
rem set JAVA_OPTS=-Xdebug -
Xrunjdwp:transport=dt_socket,address=8787,server=y,suspend=y %JAVA_OPTS%
set JAVA_OPTS=-Xdebug -
Xrunjdwp:transport=dt_socket,address=8787,server=y,suspend=n %JAVA_OPTS%
```

2) In your IDE debug by connecting to a remote Java application on localhost on port 8787. Then you can start adding break points and run through the processes with the console until the breakpoint is hit.

Configuration

The simplest way to configure jBPM is by putting the **`jbpm.cfg.xml`** configuration file in the root of the classpath. If that file is not found as a resource, the default minimal configuration will be used that is included in the jbpm library. Note that the minimal configuration does not have any configurations for persistence.

The jBPM configuration is represented by the java class **`org.jbpm.JbpmConfiguration`**. Most easy way to get a hold of the JbpmConfiguration is to make use of the singleton instance method **`JbpmConfiguration.getInstance()`**.

If you want to load a configuration from another source, you can use the **`JbpmConfiguration.parseXxxx`** methods.

```
static JbpmConfinguration jbpmConfiguration =
 JbpmConfinguration.getInstance();
```

The JbpmConfiguration is threadsafe and hence can be kept in a static member. All threads can use the JbpmConfiguration as a factory for JbpmContext objects. A JbpmContext typically represents one transaction. The JbpmContext makes services available inside of a context block. A context block looks like this:

```
JbpmContext jbpmContext = jbpmConfiguration.createJbpmContext();
try {
  // This is what we call a context block.
  // Here you can perform workflow operations

} finally {
  jbpmContext.close();
}
```

The **`JbpmContext`** makes a set of services and the configuration available to jBPM. These services are configured in the **`jbpm.cfg.xml`** configuration file and make it possible for jBPM to run in any Java environment and use whatever services are available in that environment.

Here's a typical configuration for the JbpmContext as you can find it in **`src/config.files/jbpm.cfg.xml`**:

```
<jbpm-configuration>

  <jbpm-context>

  <service name='persistence' factory='org.jbpm.persistence.db.DbPersistenceService
 />

  <service name='message' factory='org.jbpm.msg.db.DbMessageServiceFactory'
 />

  <service name='scheduler' factory='org.jbpm.scheduler.db.DbSchedulerServiceFactor
 />
```

```
  <service name='logging' factory='org.jbpm.logging.db.DbLoggingServiceFactory'
 />

  <service name='authentication' factory='org.jbpm.security.authentication.DefaultAuthen
 />
  </jbpm-context>

  <!-- configuration resource files pointing to default configuration
 files in jbpm-{version}.jar -->
  <string name='resource.hibernate.cfg.xml' value='hibernate.cfg.xml' />
  <!-- <string name='resource.hibernate.properties'
 value='hibernate.properties' /> -->
  <string name='resource.business.calendar' value='org/jbpm/calendar/
jbpm.business.calendar.properties' />
  <string name='resource.default.modules' value='org/jbpm/graph/def/
jbpm.default.modules.properties' />
  <string name='resource.converter' value='org/jbpm/db/hibernate/
jbpm.converter.properties' />
  <string name='resource.action.types' value='org/jbpm/graph/action/
action.types.xml' />
  <string name='resource.node.types' value='org/jbpm/graph/node/
node.types.xml' />
  <string name='resource.parsers' value='org/jbpm/jpdl/par/
jbpm.parsers.xml' />
  <string name='resource.varmapping' value='org/jbpm/context/exe/
jbpm.varmapping.xml' />

  <int name='jbpm.byte.block.size' value="1024" singleton="true" />
  <bean name='jbpm.task.instance.factory' class='org.jbpm.taskmgmt.impl.DefaultTaskInsta
 />
  <bean name='jbpm.variable.resolver' class='org.jbpm.jpdl.el.impl.JbpmVariableResolver'
 />

</jbpm-configuration>
```

In this configuration file you can see 3 parts:

- The first part configures the jbpm context with a set of service implementations. The possible configuration options are covered in the chapters that cover the specific service implementations.
- The second part are all mappings of references to configuration resources. These resource references can be updated if you want to customize one of these configuration files. Typically, you make a copy the default configuration which is in the **`jbpm-3.x.jar`** and put it somewhere on the classpath. Then you update the reference in this file and jbpm will use your customized version of that configuration file.
- The third part are some miscellaneous configurations used in jbpm. These configuration options are described in the chapters that cover the specific topic.

The default configured set of services is targeted at a simple web-app environment and minimal dependencies. The persistence service will obtain a jdbc connection and all the other services will use

the same connection to perform their services. So all of your workflow operations are centralized into 1 transaction on a JDBC connection without the need for a transaction manager.

`JbpmContext` contains convenience methods for most of the common process operations:

```
public void deployProcessDefinition(ProcessDefinition processDefinition)
 {...}
  public List getTaskList() {...}
  public List getTaskList(String actorId) {...}
  public List getGroupTaskList(List actorIds) {...}
  public TaskInstance loadTaskInstance(long taskInstanceId) {...}
  public TaskInstance loadTaskInstanceForUpdate(long taskInstanceId)
 {...}
  public Token loadToken(long tokenId) {...}
  public Token loadTokenForUpdate(long tokenId) {...}
  public ProcessInstance loadProcessInstance(long processInstanceId)
 {...}
  public ProcessInstance loadProcessInstanceForUpdate(long
 processInstanceId) {...}
  public ProcessInstance newProcessInstance(String processDefinitionName)
 {...}
  public void save(ProcessInstance processInstance) {...}
  public void save(Token token) {...}
  public void save(TaskInstance taskInstance) {...}
  public void setRollbackOnly() {...}
```

Note that the **`XxxForUpdate`** methods will register the loaded object for auto-save so that you don't have to call one of the save methods explicitly.

It's possible to specify multiple **`jbpm-context`**s, but then you have to make sure that each **`jbpm-context`** is given a unique **`name`** attribute. Named contexts can be retrieved with **`JbpmConfiguration.createContext(String name);`**

A **`service`** element specifies the name of a service and the service factory for that service. The service will only be created in case it's asked for with **`JbpmContext.getServices().getService(String name)`**.

The factories can also be specified as an element instead of an attribute. That might be necessary to inject some configuration information in the factory objects. The component responsible for parsing the XML, creating and wiring the objects is called the object factory.

5.1. Customizing factories

A common mistake when customizing factories is to mix the short and the long notation. Examples of the short notation can be seen in the default configuration file and above: E.g.

```
<service name='persistence' factory='org.jbpm.persistence.db.DbPersistenceServiceFa
 />
```

If specific properties on a service need to be specified, the short notation can't be used, but instead, the long notation has to be used like this: E.g.

```
  <service name="persistence">
    <factory>
      <bean class="org.jbpm.persistence.db.DbPersistenceServiceFactory">
        <field name="dataSourceJndiName"><string value="java:/
myDataSource"/></field>
        <field name="isCurrentSessionEnabled"><true /></field>
        <field name="isTransactionEnabled"><false /></field>
      </bean>
    </factory>
  </service>
```

5.2. Configuration properties

jbpm.byte.block.size: File attachments and binary variables are stored in the database. Not as blobs, but as a list of fixed sized binary objects. This is done to improve portability amongst different databases and improve overall embeddability of jBPM. This parameter controls the size of the fixed length chunks.

jbpm.task.instance.factory: To customize the way that task instances are created, specify a fully qualified class name in this property. This might be necessary when you want to customize the TaskInstance bean and add new properties to it. See also *Section 10.10, "Customizing task instances"* The specified class should implement org.jbpm.taskmgmt.TaskInstanceFactory.

jbpm.variable.resolver: To customize the way that jBPM will look for the first term in JSF-like expressions.

5.3. Other configuration files

Here's a short description of all the configuration files that are customizable in jBPM.

5.3.1. Hibernate Configuration xml file

This file contains hibernate configurations and references to the hibernate mapping resource files.

Location: **`hibernate.cfg.xml`** unless specified otherwise in the jbpm.hibernate.cfg.xml property in the jbpm.properties file. In the jbpm project the default hibernate configuration xml file is located in directory **`src/config.files/hibernate.cfg.xml`**

5.3.2. Hibernate queries configuration file

This file contains hibernate queries that are used in the jBPM sessions **`org.jbpm.db.*Session`**.

Location: **`org/jbpm/db/hibernate.queries.hbm.xml`**

5.3.3. Node types configuration file

This file contains the mapping of XML node elements to Node implementation classes.

Location: **`org/jbpm/graph/node/node.types.xml`**

5.3.4. Action types configuration file

This file contains the mapping of XML action elements to Action implementation classes.

Location: **`org/jbpm/graph/action/action.types.xml`**

5.3.5. Business calendar configuration file

Contains the definition of business hours and free time.

Location: **`org/jbpm/calendar/jbpm.business.calendar.properties`**

5.3.6. Variable mapping configuration file

Specifies how the values of the process variables (java objects) are converted to variable instances for storage in the jbpm database.

Location: **`org/jbpm/context/exe/jbpm.varmapping.xml`**

5.3.7. Converter configuration file

Specifies the id-to-classname mappings. The id's are stored in the database. The org.jbpm.db.hibernate.ConverterEnumType is used to map the ids to the singleton objects.

Location: **`org/jbpm/db/hibernate/jbpm.converter.properties`**

5.3.8. Default modules configuration file

specifies which modules are added to a new ProcessDefinition by default.

Location: **`org/jbpm/graph/def/jbpm.default.modules.properties`**

5.3.9. Process archive parsers configuration file

specifies the phases of process archive parsing

Location: **`org/jbpm/jpdl/par/jbpm.parsers.xml`**

5.4. jBPM debug logs in JBoss

When running jPDL in JBoss and you want to see the debug logs of jBPM, replace the file **`conf/log4j.xml`** in the jboss server configuration take with the file **`deploy/log4j.xml`** in your jPDL distribution. In the suite, the full location of the file to be replaced is **`[jpdl.home]/server/server/jbpm/conf/log4j.xml`**.

5.5. Logging of optimistic concurrency exceptions

When running in a cluster, jBPM synchronizes on the database. By default with optimistic locking. This means that each operation is performed in a transaction. And if at the end a collision is detected, then the transaction is rolled back and has to be handled. E.g. by a retry. So optimistic locking exceptions are usually part of the normal operation. Therefor, by default, the **`org.hibernate.StateObjectStateException`**s the that hibernate throws in that case are not logged with error and a stack trace, but instead a simple info message 'optimistic locking failed' is displayed.

Hibernate itself will log the StateObjectStateException including a stack trace. If you want to get rid of these stack traces, put the level of

`org.hibernate.event.def.AbstractFlushingEventListener` to FATAL. If you use log4j following line of configuration can be used for that:

```
log4j.logger.org.hibernate.event.def.AbstractFlushingEventListener=FATAL
```

If you want to enable logging of the jBPM stack traces, add the following line to your **`jbpm.cfg.xml`**:

```
<boolean name="jbpm.hide.stale.object.exceptions" value="false" />
```

.

5.6. Object factory

The object factory can create objects according to a beans-like xml configuration file. The configuration file specifies how objects should be created, configured and wired together to form a complete object graph. The object factory can inject the configurations and other beans into a bean.

In its simplest form, the object factory is able to create basic types and java beans from such a configuration:

```
<beans>
  <bean name="task" class="org.jbpm.taskmgmt.exe.TaskInstance"/>
  <string name="greeting">hello world</string>
  <int name="answer">42</int>
  <boolean name="javaisold">true</boolean>
  <float name="percentage">10.2</float>
  <double name="salary">100000000.32</double>
  <char name="java">j</char>
  <null name="dusttodust" />
</beans>

-------------------------------------------------------

ObjectFactory of = ObjectFactory.parseXmlFromAbove();
assertEquals(TaskInstance.class, of.getNewObject("task").getClass());
assertEquals("hello world", of.getNewObject("greeting"));
assertEquals(new Integer(42), of.getNewObject("answer"));
assertEquals(Boolean.TRUE, of.getNewObject("javaisold"));
assertEquals(new Float(10.2), of.getNewObject("percentage"));
assertEquals(new Double(100000000.32), of.getNewObject("salary"));
assertEquals(new Character('j'), of.getNewObject("java"));
assertNull(of.getNewObject("dusttodust"));
```

Also you can configure lists:

```
<beans>
  <list name="numbers">
    <string>one</string>
    <string>two</string>
    <string>three</string>
```

```
  </list>
</beans>
```

and maps

```
<beans>
  <map name="numbers">
    <entry><key><int>1</int></key><value><string>one</string></value></
entry>
    <entry><key><int>2</int></key><value><string>two</string></value></
entry>
    <entry><key><int>3</int></key><value><string>three</string></value></
entry>
  </map>
</beans>
```

Beans can be configured with direct field injection and via property setters.

```
<beans>
  <bean name="task" class="org.jbpm.taskmgmt.exe.TaskInstance" >
    <field name="name"><string>do dishes</string></field>
    <property name="actorId"><string>theotherguy</string></property>
  </bean>
</beans>
```

Beans can be referenced. The referenced object doesn't have to be a bean, it can be a string, integer or any other object.

```
<beans>
  <bean name="a" class="org.jbpm.A" />
  <ref name="b" bean="a" />
</beans>
```

Beans can be constructed with any constructor

```
<beans>
  <bean name="task" class="org.jbpm.taskmgmt.exe.TaskInstance" >
    <constructor>
      <parameter class="java.lang.String">
        <string>do dishes</string>
      </parameter>
      <parameter class="java.lang.String">
        <string>theotherguy</string>
      </parameter>
    </constructor>
  </bean>
</beans>
```

... or with a factory method on a bean ...

```
<beans>
  <bean name="taskFactory"
        class="org.jbpm.UnexistingTaskInstanceFactory"
        singleton="true"/>

  <bean name="task" class="org.jbpm.taskmgmt.exe.TaskInstance" >
    <constructor factory="taskFactory" method="createTask" >
      <parameter class="java.lang.String">
        <string>do dishes</string>
      </parameter>
      <parameter class="java.lang.String">
        <string>theotherguy</string>
      </parameter>
    </constructor>
  </bean>
</beans>
```

... or with a static factory method on a class ...

```
<beans>
  <bean name="task" class="org.jbpm.taskmgmt.exe.TaskInstance" >
    <constructor factory-
class="org.jbpm.UnexistingTaskInstanceFactory" method="createTask" >
      <parameter class="java.lang.String">
        <string>do dishes</string>
      </parameter>
      <parameter class="java.lang.String">
        <string>theotherguy</string>
      </parameter>
    </constructor>
  </bean>
</beans>
```

Each named object can be marked as singleton with the attribute **`singleton="true"`**. That means that a given object factory will always return the same object for each request. Note that singletons are not shared between different object factories.

The singleton feature causes the differentiation between the methods **`getObject`** and **`getNewObject`**. Typical users of the object factory will use the **`getNewObject`**. This means that first the object factory's object cache is cleared before the new object graph is constructed. During construction of the object graph, the non-singleton objects are stored in the object factory's object cache to allow for shared references to one object. The singleton object cache is different from the plain object cache. The singleton cache is never cleared, while the plain object cache is cleared at the start of every **`getNewObject`** method.

Persistence

In most scenarios, jBPM is used to maintain execution of processes that span a long time. In this context, "a long time" means spanning several transactions. The main purpose of persistence is to store process executions during wait states. So think of the process executions as state machines. In one transaction, we want to move the process execution state machine from one state to the next.

A process definition can be represented in 3 different forms : as xml, as java objects and as records in the jBPM database. Execution (or runtime) information and logging information can be represented in 2 forms : as java objects and as records in the jBPM database.

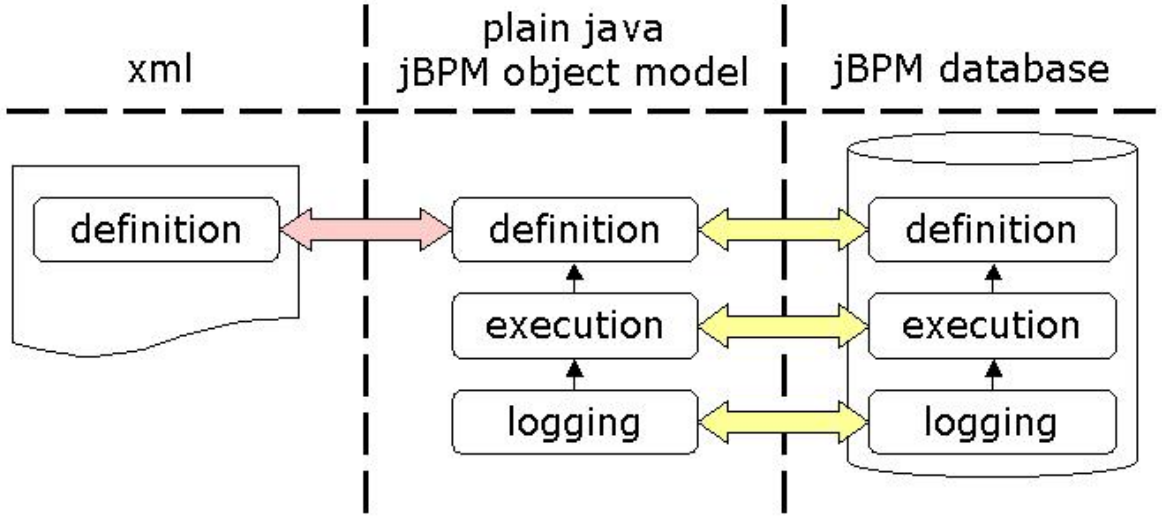

Figure 6.1. The transformations and different forms

For more information about the xml representation of process definitions and process archives, see *Chapter 17, jBPM Process Definition Language (JPDL)*.

More information on how to deploy a process archive to the database can be found in *Section 17.1.1, "Deploying a process archive"*

6.1. The persistence API

6.1.1. Relation to the configuration framework

The persistence API is an integrated with the configuration framework[1] by exposing some convenience persistence methods on the JbpmContext. Persistence API operations can therefore be called inside a jBPM context block like this:

```
JbpmContext jbpmContext = jbpmConfiguration.createJbpmContext();
try {

  // Invoke persistence operations here

} finally {
```

[1] *Chapter 5, Configuration*.

```
  jbpmContext.close();
}
```

In what follows, we suppose that the configuration includes a persistence service similar to this one (as in the example configuration file **`src/config.files/jbpm.cfg.xml`**):

```
<jbpm-configuration>

  <jbpm-context>

  <service name='persistence' factory='org.jbpm.persistence.db.DbPersistenceServiceFacto
 />
  </jbpm-context>
</jbpm-configuration>
```

6.1.2. Convenience methods on JbpmContext

The three most common persistence operations are:

- Deploying a process
- Starting a new execution of a process
- Continuing an execution

First deploying a process definition. Typically, this will be done directly from the graphical process designer or from the deployprocess ant task. But here you can see how this is done programmatically:

```
JbpmContext jbpmContext = jbpmConfiguration.createJbpmContext();
try {
  ProcessDefinition processDefinition = ...;
  jbpmContext.deployProcessDefinition(processDefinition);
} finally {
  jbpmContext.close();
}
```

For the creation of a new process execution, we need to specify of which process definition this execution will be an instance. The most common way to specify this is to refer to the name of the process and let jBPM find the latest version of that process in the database:

```
JbpmContext jbpmContext = jbpmConfiguration.createJbpmContext();
try {
  String processName = ...;
  ProcessInstance processInstance =
      jbpmContext.newProcessInstance(processName);
} finally {
  jbpmContext.close();
}
```

For continuing a process execution, we need to fetch the process instance, the token or the taskInstance from the database, invoke some methods on the POJO jBPM objects and afterward save the updates made to the processInstance into the database again.

```
JbpmContext jbpmContext = jbpmConfiguration.createJbpmContext();
try {
  long processInstanceId = ...;
  ProcessInstance processInstance =
      jbpmContext.loadProcessInstance(processInstanceId);
  processInstance.signal();
  jbpmContext.save(processInstance);
} finally {
  jbpmContext.close();
}
```

Note that if you use the xxx**ForUpdate** methods in the JbpmContext, an explicit invocation of the jbpmContext.save is not necessary any more because it will then occur automatically during the close of the jbpmContext. E.g. suppose we want to inform jBPM about a taskInstance that has been completed. Note that task instance completion can trigger execution to continue so the processInstance related to the taskInstance must be saved. The most convenient way to do this is to use the loadTaskInstanceForUpdate method:

```
JbpmContext jbpmContext = jbpmConfiguration.createJbpmContext();
try {
  long taskInstanceId = ...;
  TaskInstance taskInstance =
      jbpmContext.loadTaskInstanceForUpdate(taskInstanceId);
  taskInstance.end();
} finally {
  jbpmContext.close();
}
```

Just as background information, the next part is an explanation of how jBPM manages the persistence and uses hibernate.

The **`JbpmConfiguration`** maintains a set of **`ServiceFactory`**s. The service factories are configured in the **`jbpm.cfg.xml`** as shown above and instantiated lazy. The **`DbPersistenceServiceFactory`** is only instantiated the first time when it is needed. After that, service factories are maintained in the **`JbpmConfiguration`**. A **`DbPersistenceServiceFactory`** manages a hibernate **`SessionFactory`**. But also the hibernate session factory is created lazy when requested the first time.

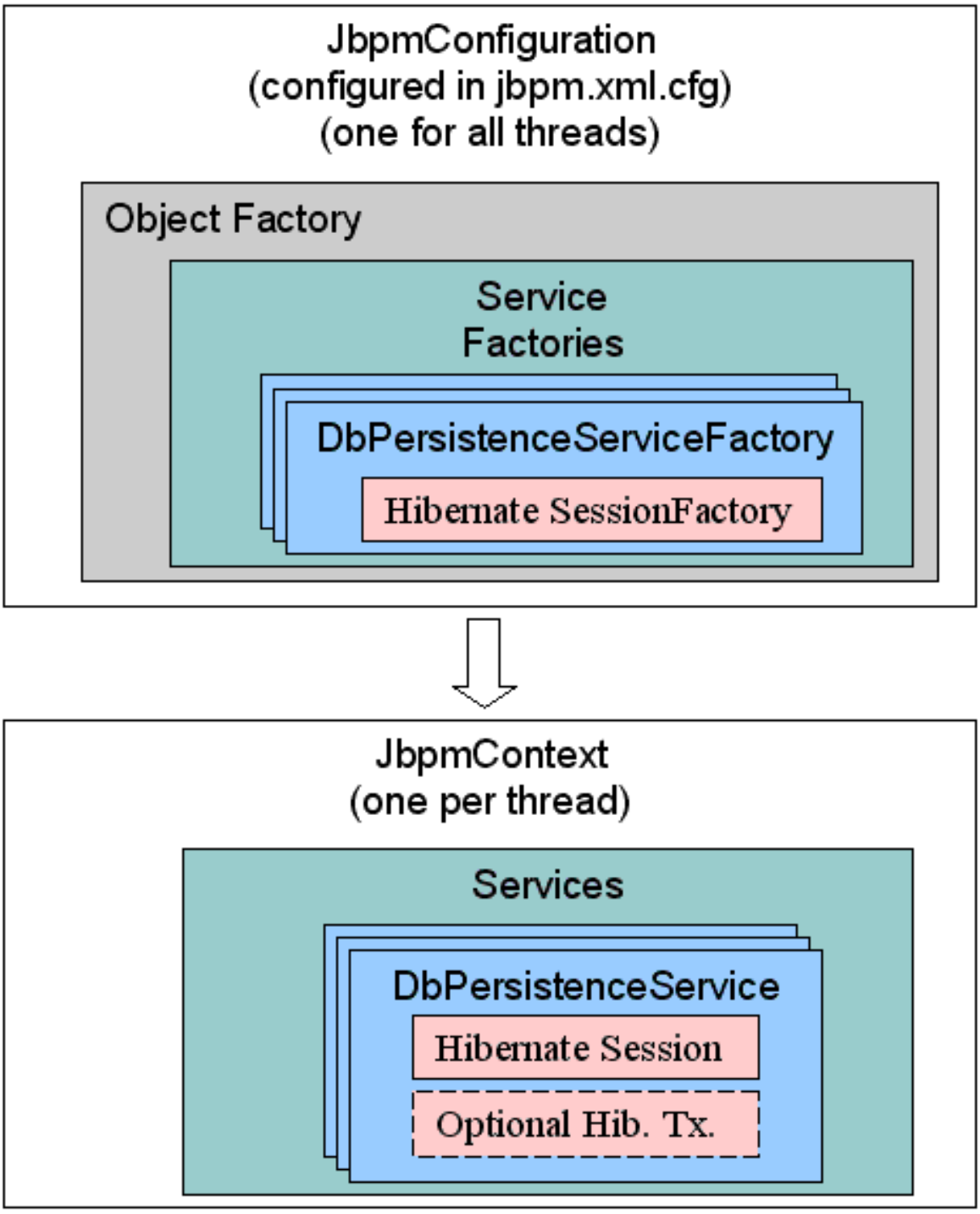

Figure 6.2. The persistence related classes

During the invocation of **`jbpmConfiguration.createJbpmContext()`**, only the **`JbpmContext`** is created. No further persistence related initializations are done at that time. The **`JbpmContext`** manages a **`DbPersistenceService`**, which is instantiated upon first request. The **`DbPersistenceService`** manages the hibernate session. Also the hibernate session inside the **`DbPersistenceService`** is created lazy. As a result, a hibernate session will be only be opened when the first operation is invoked that requires persistence and not earlier.

6.1.3. Managed transactions

The most common scenario for managed transactions is when using jBPM in a JEE application server like JBoss. The most common scenario is the following:

- Configure a DataSource in your application server
- Configure hibernate to use that data source for its connections
- Use container managed transactions
- Disable transactions in jBPM

A stateless session facade in front of jBPM is a good practice. The easiest way on how to bind the jbpm transaction to the container transaction is to make sure that the hibernate configuration used by jbpm refers to an xa-datasource. So jbpm will have its own hibernate session, there will only be 1 jdbc connection and 1 transaction.

The transaction attribute of the jbpm session facade methods should be 'required'

The the most important configuration property to specify in the **`hibernate.cfg.xml`** that is used by jbpm is hibernate.connection.datasource. Set this to you datasource JNDI name, e.g. java:/DefaultDS

More information on how to configure jdbc connections in hibernate, see *the hibernate reference manual, section 'Hibernate provided JDBC connections'*[2]

For more information on how to configure xa datasources in jboss, see *the jboss application server guide, section 'Configuring JDBC DataSources'*[3]

6.1.4. Injecting the hibernate session

In some scenarios, you already have a hibernate session and you want to combine all the persistence work from jBPM into that hibernate session.

Then the first thing to do is make sure that the hibernate configuration is aware of all the jBPM mapping files. You should make sure that all the hibernate mapping files that are referenced in the file **`src/config.files/hibernate.cfg.xml`** are provided in the used hibernate configuration.

Then, you can inject a hibernate session into the jBPM context as is shown in the following API snippet:

```
JbpmContext jbpmContext = jbpmConfiguration.createJbpmContext();
try {
  jbpmContext.setSession(SessionFactory.getCurrentSession());

  // your jBPM operations on jbpmContext

} finally {
  jbpmContext.close();
}
```

[2] http://www.hibernate.org/hib_docs/reference/en/html/session-configuration.html#configuration-hibernatejdbc
[3] http://docs.jboss.org/jbossas/jboss4guide/r4/html/ch7.chapt.html#ch7.jdbc.sect

That will pass in the current hibernate session used by the container to the jBPM context. No hibernate transaction is initiated when a session is injected in the context. So this can be used with the default configurations.

The hibernate session that is passed in, will **not** be closed in the jbpmContext.close() method. This is in line with the overall philosophy of programmatic injection which is explained in the next section.

6.1.5. Injecting resources programmatically

The configuration of jBPM provides the necessary information for jBPM to create a hibernate session factory, hibernate session, jdbc connections, jbpm required services,... But all of these resources can also be provided to jBPM programmatically. Just inject them in the jbpmContext. Injected resources always are taken before creating resources from the jbpm configuration information.

The main philosophy is that the API-user remains responsible for all the things that the user injects programmatically in the jbpmContext. On the other hand, all items that are opened by jBPM, will be closed by jBPM. There is one exception. That is when fetching a connection that was created by hibernate. When calling jbpmContext.getConnection(), this transfers responsibility for closing the connection from jBPM to the API user.

```
JbpmContext jbpmContext = jbpmConfiguration.createJbpmContext();
try {
  // to inject resources in the jbpmContext before they are used, you can
 use
  jbpmContext.setConnection(connection);
  // or
  jbpmContext.setSession(session);
  // or
  jbpmContext.setSessionFactory(sessionFactory);

} finally {
  jbpmContext.close();
}
```

6.1.6. Advanced API usage

The DbPersistenceService maintains a lazy initialized hibernate session. All database access is done through this hibernate session. All queries and updates done by jBPM are exposed by the XxxSession classes like e.g. GraphSession, SchedulerSession, LoggingSession,... These session classes refer to the hibernate queries and all use the same hibernate session underneath.

The XxxxSession classes are accessible via the JbpmContext as well.

6.2. Configuring the persistence service

6.2.1. The DbPersistenceServiceFactory

The DbPersistenceServiceFactory itself has 3 more configuration properties: isTransactionEnabled, sessionFactoryJndiName and dataSourceJndiName. To specify any of these properties in the jbpm.cfg.xml, you need to specify the service factory as a bean in the factory element like this:

IMPORTANT: don't mix the short and long notation for configuring the factories. See also *Section 5.1, "Customizing factories"*. If the factory is just a new instance of a class, you can use the factory attribute to refer to the factory class name. But if properties in a factory must be configured, the long notation must be used and factory and bean must be combined as nested elements. Like this:

```
<jbpm-context>
  <service name="persistence">
    <factory>

<bean class="org.jbpm.persistence.db.DbPersistenceServiceFactory">
       <field name="isTransactionEnabled"><false /></field>
       <field name="sessionFactoryJndiName">
         <string value="java:/myHibSessFactJndiName" />
       </field>
       <field name="dataSourceJndiName">
         <string value="java:/myDataSourceJndiName" />
       </field>
     </bean>
    </factory>
  </service>
  ...
</jbpm-context>
```

- **isTransactionEnabled**: by default, jBPM will begin a hibernate transaction when the session is fetched the first time and if the jbpmContext is closed, the hibernate transaction will be ended. The transaction is then committed or rolled back depending on wether jbpmContext.setRollbackOnly was called. The isRollbackOnly property is maintained in the TxService. To disable transactions and prohibit jBPM from managing transactions with hibernate, configure the isTransactionEnabled property to false as in the example above. This property only controls the behavior of the jbpmContext, you can still call the DbPersistenceService.beginTransaction() directly with the API, which ignores the isTransactionEnabled setting. For more info about transactions, see *Section 6.3, "Hibernate transactions"*.
- **sessionFactoryJndiName**: by default, this is null, meaning that the session factory is not fetched from JNDI. If set and a session factory is needed to create a hibernate session, the session factory will be fetched from jndi using the provided JNDI name.
- **dataSourceJndiName**: by default, this is null and creation of JDBC connections will be delegated to hibernate. By specifying a datasource, jBPM will fetch a JDBC connection from the datasource and provide that to hibernate while opening a new session.

6.2.2. The hibernate session factory

By default, the DbPersistenceServiceFactory will use the resource hibernate.cfg.xml in the root of the classpath to create the hibernate session factory. Note that the hibernate configuration file resource is mapped in the property 'jbpm.hibernate.cfg.xml' and can be customized in the jbpm.cfg.xml. This is the default configuration:

```
<jbpm-configuration>
  <!-- configuration resource files pointing to default configuration
 files in jbpm-{version}.jar -->
```

```
  <string name='resource.hibernate.cfg.xml'
          value='hibernate.cfg.xml' />
  <!-- <string name='resource.hibernate.properties'
                value='hibernate.properties' /> -->
</jbpm-configuration>
```

When the property **resource.hibernate.properties** is specified, the properties in that resource file will **overwrite all** the properties in the hibernate.cfg.xml. Instead of updating the hibernate.cfg.xml to point to your DB, the hibernate.properties can be used to handle jbpm upgrades conveniently: The hibernate.cfg.xml can then be copied without having to reapply the changes.

6.2.3. Configuring a c3po connection pool

Please refer to the hibernate documentation: *http://www.hibernate.org/214.html*

6.2.4. Configuring a ehcache cache provider

If you want to configure jBPM with JBossCache, have a look at *the jBPM configuration wiki page*[4]

For more information about configuring a cache provider in hibernate, take a look at *the hibernate documentation, section 'Second level cache'*[5]

The **`hibernate.cfg.xml`** that ships with jBPM includes the following line:

```
<property name="hibernate.cache.provider_class">org.hibernate.cache.HashtableCacheProvid
property>
```

This is done to get people up and running as fast as possible without having to worry about classpaths. Note that hibernate contains a warning that states not to use the HashtableCacheProvider in production.

To use ehcache instead of the HashtableCacheProvider, simply remove that line and put ehcache.jar on the classpath. Note that you might have to search for the right ehcache library version that is compatible with your environmment. Previous incompatibilities between a JBoss version and a particular ehcache version were the reason to change the default to HashtableCacheProvider.

6.3. Hibernate transactions

By default, jBPM will delegate transaction to hibernate and use the session per transaction pattern. jBPM will begin a hibernate transaction when a hibernate session is opened. This will happen the first time when a persistent operation is invoked on the jbpmContext. The transaction will be committed right before the hibernate session is closed. That will happen inside the jbpmContext.close().

Use **`jbpmContext.setRollbackOnly()`** to mark a transaction for rollback. In that case, the transaction will be rolled back right before the session is closed inside of the jbpmContext.close().

To prohibit jBPM from invoking any of the transaction methods on the hibernate API, set the isTransactionEnabled property to false as explained in *Section 6.2.1, "The DbPersistenceServiceFactory"* above.

[4] http://wiki.jboss.org/wiki/Wiki.jsp?page=JbpmConfiguration
[5] http://www.hibernate.org/hib_docs/reference/en/html/performance.html#performance-cache

6.4. JTA transactions

The most common scenario for managed transactions is when using jBPM in a JEE application server like JBoss. The most common scenario to bind your transactions to JTA is the following:

```
  <jbpm-context>
    <service name="persistence">
      <factory>

  <bean class="org.jbpm.persistence.db.DbPersistenceServiceFactory">
         <field name="isTransactionEnabled"><false /></field>
         <field name="isCurrentSessionEnabled"><true /></field>
         <field name="sessionFactoryJndiName">
           <string value="java:/myHibSessFactJndiName" />
         </field>
       </bean>
      </factory>
    </service>
  </jbpm-context>
```

Then you should specify in your hibernate session factory to use a datasource and bind hibernate to the transaction manager. Make sure that you bind the datasource to an XA datasource in case you are using more than one resource. For more information about binding hibernate to your transaction manager, please, refer to *paragraph 'Transaction strategy configuration' in the hibernate documentation*[6].

```
<hibernate-configuration>
  <session-factory>

    <!-- hibernate dialect -->

  <property name="hibernate.dialect">org.hibernate.dialect.HSQLDialect</
property>

    <!-- DataSource properties (begin) -->
    <property name="hibernate.connection.datasource">java:/JbpmDS</
property>

    <!-- JTA transaction properties (begin) -->

  <property name="hibernate.transaction.factory_class">org.hibernate.transaction.JT
property>

  <property name="hibernate.transaction.manager_lookup_class">org.hibernate.transac
property>
    <property name="jta.UserTransaction">java:comp/UserTransaction</
property>

  </session-factory>
```

[6] http://www.hibernate.org/hib_docs/v3/reference/en/html_single/#configuration-optional-transactionstrategy

```
</hibernate-configuration>
```

Then make sure that you have configured hibernate to use an XA datasource.

These configurations allow for the enterprise beans to use CMT and still allow the web console to use BMT. That is why the property 'jta.UserTransaction' is also specified.

6.5. Customizing queries

All the HQL queries that jBPM uses are centralized in one configuration file. That resource file is referenced in the hibernate.cfg.xml configuration file like this:

```
<hibernate-configuration>
    <!-- hql queries and type defs -->
    <mapping resource="org/jbpm/db/hibernate.queries.hbm.xml" />
</hibernate-configuration>
```

To customize one or more of those queries, take a copy of the original file and put your customized version somewhere on the classpath. Then update the reference 'org/jbpm/db/hibernate.queries.hbm.xml' in the hibernate.cfg.xml to point to your customized version.

6.6. Database compatibility

jBPM runs on any database that is supported by hibernate.

The example configuration files in jBPM (**`src/config.files`**) specify the use of the hypersonic in-memory database. That database is ideal during development and for testing. The hypersonic in-memory database keeps all its data in memory and doesn't store it on disk.

6.6.1. Isolation level of the JDBC connection

Make sure that the database isolation level that you configure for your JDBC connection is at least READ_COMMITTED.

Almost all features run OK even with READ_UNCOMMITTED (isolation level 0 and the only isolation level supported by HSQLDB). But race conditions might occur in the job executor and with synchronizing multiple tokens.

6.6.2. Changing the jBPM DB

Following is an indicative list of things to do when changing jBPM to use a different database:

- put the jdbc-driver library archive in the classpath
- update the hibernate configuration used by jBPM
- create the schema in the new database

6.6.3. The jBPM DB schema

The jbpm.db sub-project, contains a number of drivers, instructions and scripts to help you getting started on your database of choice. Please, refer to the readme.html in the root of the jbpm.db project for more information.

While jBPM is capable of generating DDL scripts for all database, these schemas are not always optimized. So you might want to have your DBA review the DDL that is generated to optimize the column types and use of indexes.

In development you might be interested in the following hibernate configuration: If you set hibernate configuration property 'hibernate.hbm2ddl.auto' to 'create-drop' (e.g. in the hibernate.cfg.xml), the schema will be automatically created in the database the first time it is used in an application. When the application closes down, the schema will be dropped.

The schema generation can also be invoked programmatically with **`jbpmConfiguration.createSchema()`** and **`jbpmConfiguration.dropSchema()`**.

6.6.4. Known Issues

This section highlights the known-issues in databases that have been tested with jBPM.

6.6.4.1. Sybase Issues

Some Sybase distributions have a known issue with truncating binary files resulting in misbehavior of the system. This limitation is resulting from the storage mechanism of binaries into the Sybase database.

6.7. Combining your hibernate classes

In your project, you might use hibernate for your persistence. Combining your persistent classes with the jBPM persistent classes is optional. There are two major benefits when combining your hibernate persistence with jBPM's hibernate persistence:

First, session, connection and transaction management become easier. By combining jBPM and your persistence into one hibernate session factory, there is one hibernate session, one jdbc connection that handles both yours and jBPM's persistence. So automatically the jBPM updates are in the same transaction as the updates to your own domain model. This can eliminates the need for using a transaction manager.

Secondly, this enable you to drop your Hibernate persistent object in to the process variables without any further hassle.

The easiest way to integrate your persistent classes with the jBPM persistent classes is by creating one central hibernate.cfg.xml. You can take the jBPM **`src/config.files/hibernate.cfg.xml`** as a starting point and add references to your own hibernate mapping files in there.

6.8. Customizing the jBPM hibernate mapping files

To customize any of the jBPM hibernate mapping files, you can proceed as follows:

- copy the jBPM hibernate mapping file(s) you want to copy from the sources (**`src/java.jbpm/...`**) or from inside of the jbpm jar.
- put the copy anywhere you want on the classpath
- update the references to the customized mapping files in the hibernate.cfg.xml configuration file

6.9. Second level cache

jBPM uses Hibernate's second level cache for keeping the process definitions in memory after loading them once. The process definition classes and collections are configured in the jBPM hibernate mapping files with the cache element like this:

```
<cache usage="nonstrict-read-write"/>
```

Since process definitions (should) never change, it is ok to keep them in the second level cache. See also *Section 17.1.3, "Changing deployed process definitions"*.

The second level cache is an important aspect of the JBoss jBPM implementation. If it weren't for this cache, JBoss jBPM could have a serious drawback in comparison to the other techniques to implement a BPM engine.

The caching strategy is set to **`nonstrict-read-write`**. At runtime, the caching strategy could be set to **`read-only`**. But in that case, you would need a separate set of hibernate mapping files for deploying a process. That is why we opted for the nonstrict-read-write.

The jBPM Database

7.1. Switching the Database Backend

Switching the JBoss jBPM database backend is reasonably straightforward. We will step through this process using PostgreSQL and MySQL as an example. The process is identical for all other supported databases. For a number of these supported databases, a number of JDBC drivers, Hibernate configuration files and Ant build files to generate the database creation scripts are present in the jBPM distribution in the DB sub-project. If you cannot find these files for the database you wish to use, you should first make sure if Hibernate supports your database. If this is the case you can have a look at files for one of the databases present in the DB project and mimic this using your own database.

For this document, we will use the JBoss jBPM Starter's Kit distribution. We will assume that this starter's kit was extracted to a location on your machine named ${JBPM_SDK_HOME}. You will find the DB sub-project of jBPM in the ${JBPM_SDK_HOME}/jbpm-db.

After installing the database, you will have to run the database creation scripts. These will create the jBPM tables in the database. To make the default web app running with this new database, we will have to update some configuration files of the server included in the Starter's Kit. For these configuration changes, we will not go into too much detail. If you want to know more about the different configuration settings in the server, we advise you to have a look at the JBoss documentation.

7.1.1. Isolation level

Whatever database that you use, make sure that the isolation level of the configured JDBC connection is at least READ_COMMITTED, as explained in *Section 6.6.1, “Isolation level of the JDBC connection”*.

7.1.2. Installing the PostgreSQL Database Manager

To install PostgreSQL or any other database you may be using, we refer to the installation manual of these products. For Windows PostgreSQL installation is pretty straightforward. The installer creates a dedicated Windows user and allows to define the database administrator. PostgreSQL comes with an administration tool called pgAdmin III that we will use to create the jBPM database. A screenshot of this tool right after creating the JbpmDB database with it is shown in the figure below.

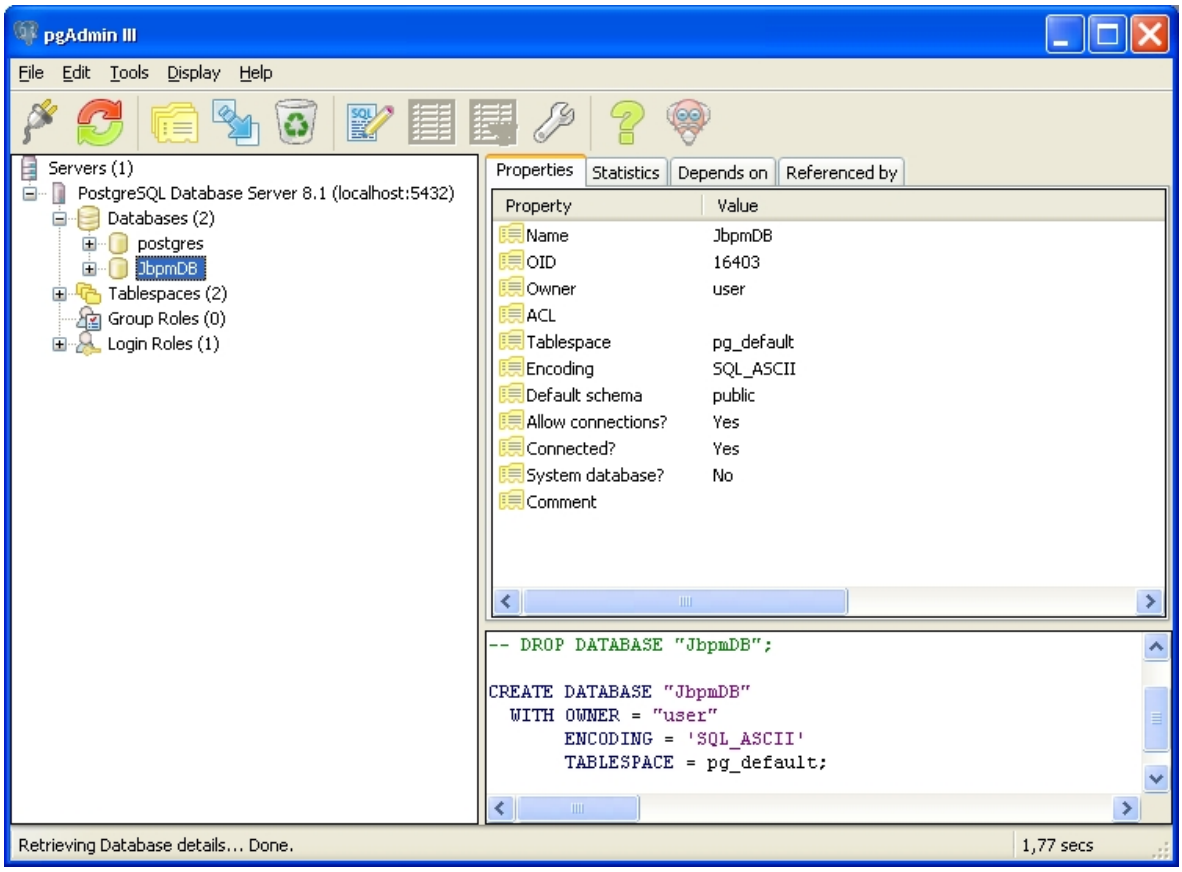

Figure 7.1. The PostgreSQL pgAdmin III tool after creating the JbpmDB database

After the installation of the database, we can use a database viewer tool like DBVisualizer to look at the contents of the database. Before you can define a database connection with DBVisualizer, you might have to add the PostgreSQL JDBC driver to the driver manager. Select 'Tools->Driver Manager...' to open the driver manager window. Look at the figure below for an example of how to add the PostgreSQL JDBC driver.

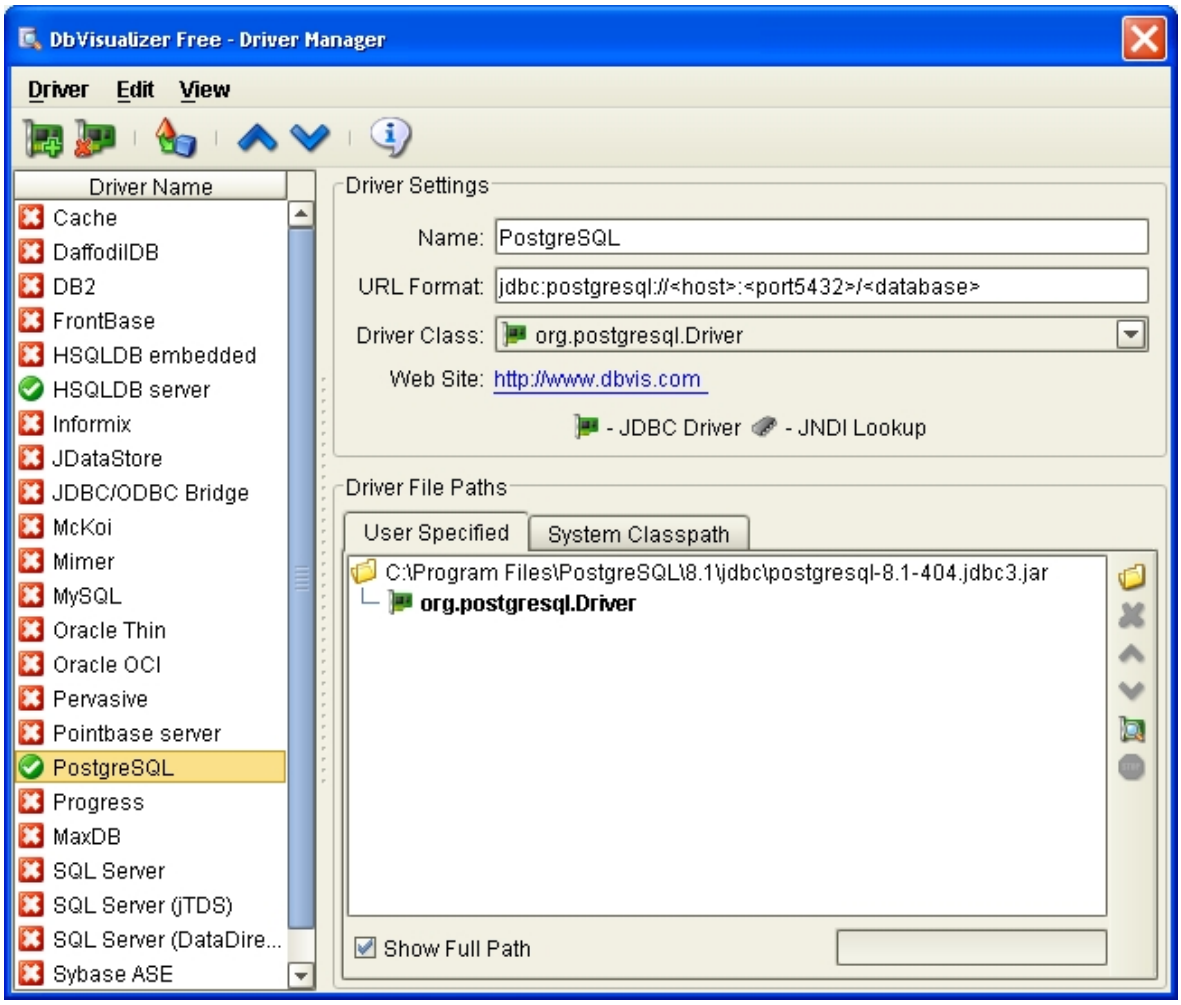

Figure 7.2. Adding the JDBC driver to the driver manager

Now everything is set to define a database connection in DBVisualizer to our newly created database. We will use this tool further in this document to make sure the creation scripts and process deployment are working as expected. For an example of creating the connection in DBVisualizer we refer to the following figure. As you can see, there are no tables present yet in this database. We will create them in the following section.

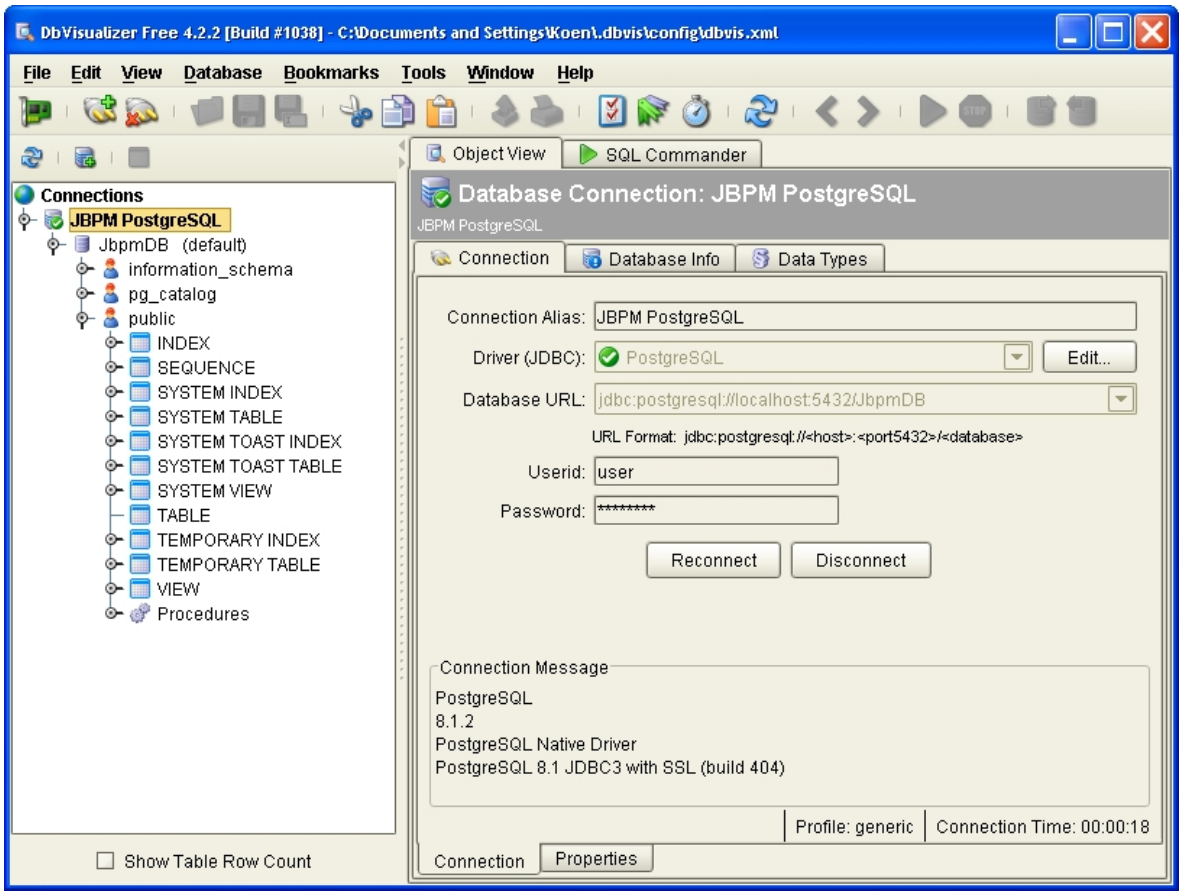

Figure 7.3. Create the connection to the jBPM database

Another thing worth mentioning is the Database URL above : 'jdbc:postgresql://localhost:5432/JbpmDB'. If you created the JbpmDB database with another name, or if PostgreSQL is not running on the localhost machine or on another port, you'll have to adapt your Database URL accordingly.

7.1.3. Installing the MySQL Database Manager

To install the MySQL database, please refer to the documentation provided by MySQL. The installation is very easy and straightforward and only takes a few minutes in windows. You will need to use the database Administration console provided by MySQL.

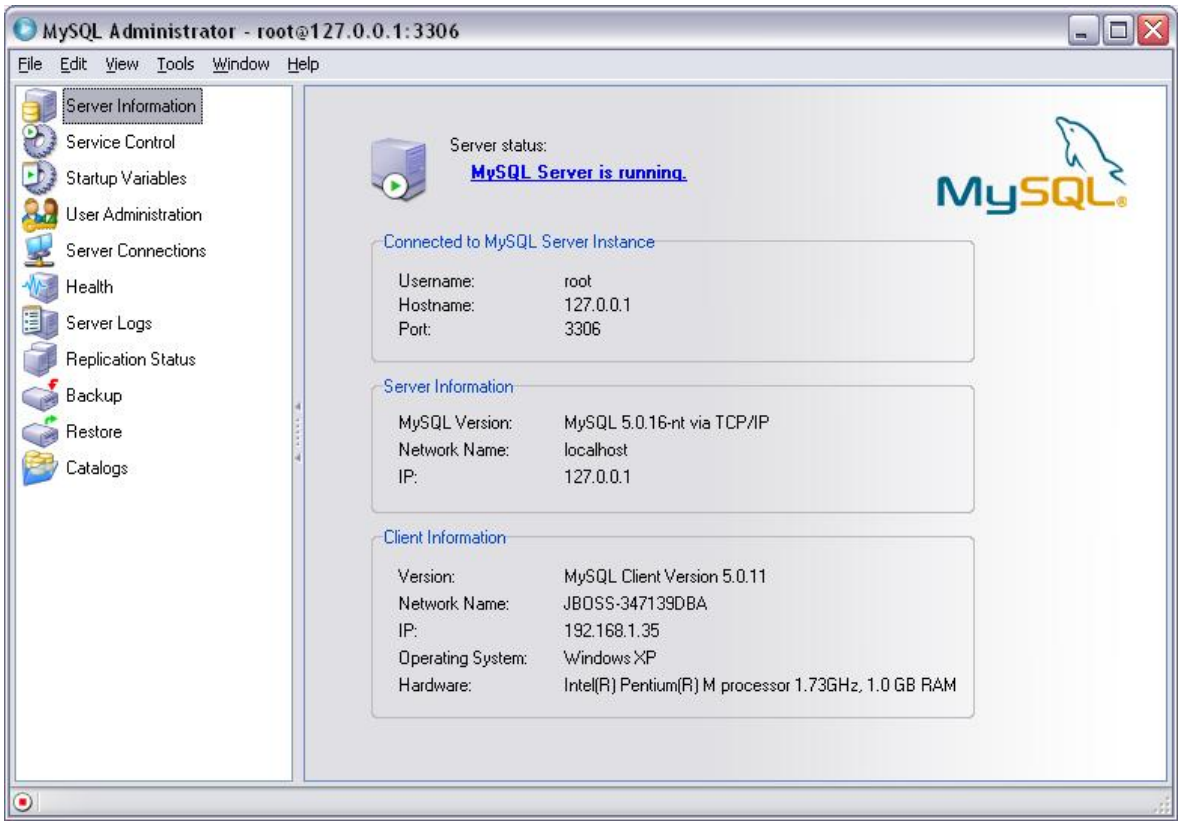

Figure 7.4. The MySQL Administrator

7.1.4. Creating the JBoss jBPM Database with your new PostGreSQL or MySQL

In order to get the proper database script generated for your database, you should use the scripts provided in the jBPM Starter's Kit. In the Starter's Kit under the ${JBPM_SSTARTERSKIT_HOME}/jbpm-db/build/${DATABASE_TYPE}/scripts you will find create scripts for all the major databases. Using your database admin console, navigate to the database and then open and execute the create script we just referenced. Below are screen shots doing this for PostGreSQL and MySQL under their respective admin consoles

7.1.4.1. Creating the JBoss jBPM Database with PostGreSQL

As already mentioned you will find the database scripts for a lot of the supported databases in the DB sub project. The database scripts for PostgreSQL are found in the folder '${JBPM_SDK_HOME}/jbpm-db/build/mysql/scripts. The creation script is called 'postgresql.create.sql'. Using DBVisualizer, you can load this script by switching to the 'SQL Commander' tab and then selecting 'File->Load...'. In the following dialog, navigate to the creation script file. The result of doing so is shown in the figure below.

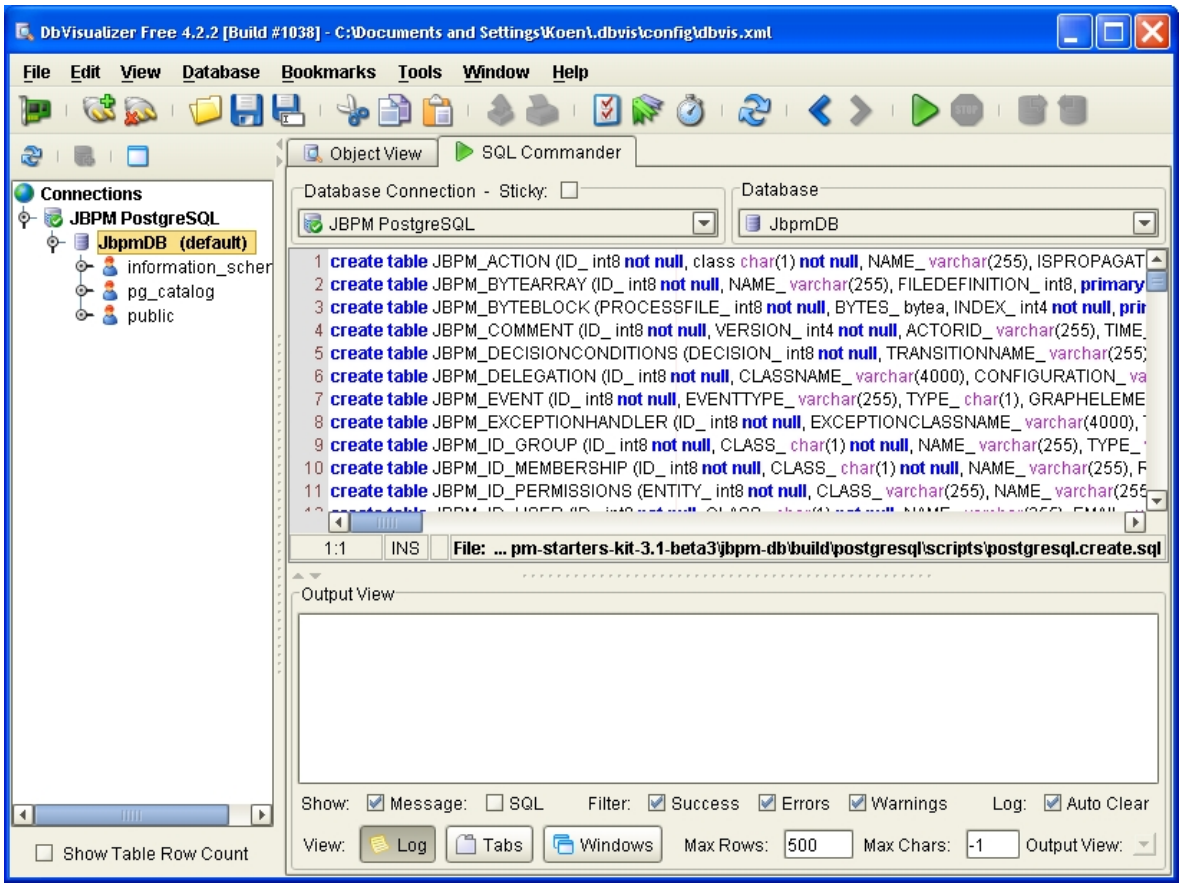

Figure 7.5. Load the database creation script

To execute this script with DBVisualizer, you select 'Database->Execute'. After this step all JBoss jBPM tables are created. The situation is illustrated in the figure below.

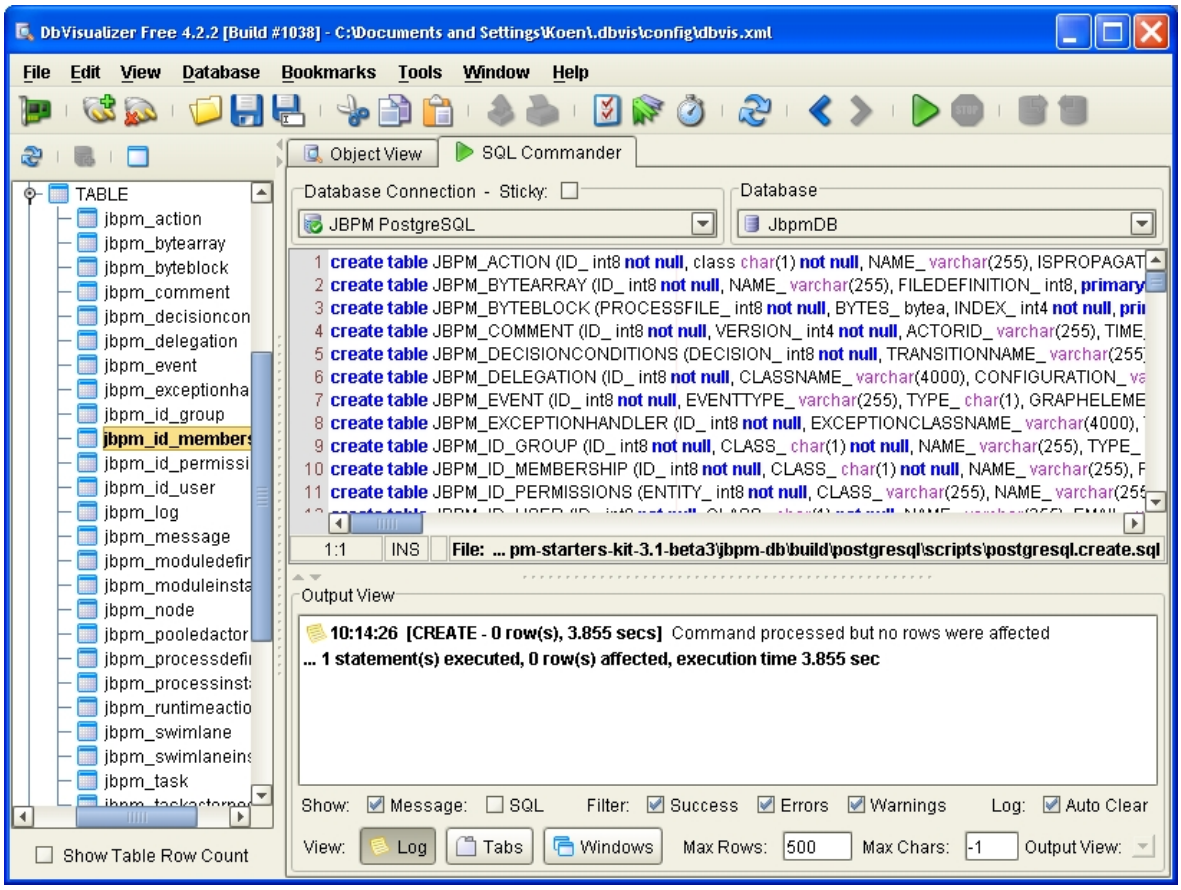

Figure 7.6. Running the database creation script

7.1.4.2. Creating the JBoss jBPM Database with your new MySQL

Once you have installed MySQL go ahead and create a jbpm database, use any name you like for this DB. In this example "jbpmdb" was used. A screenshot of the database is below.

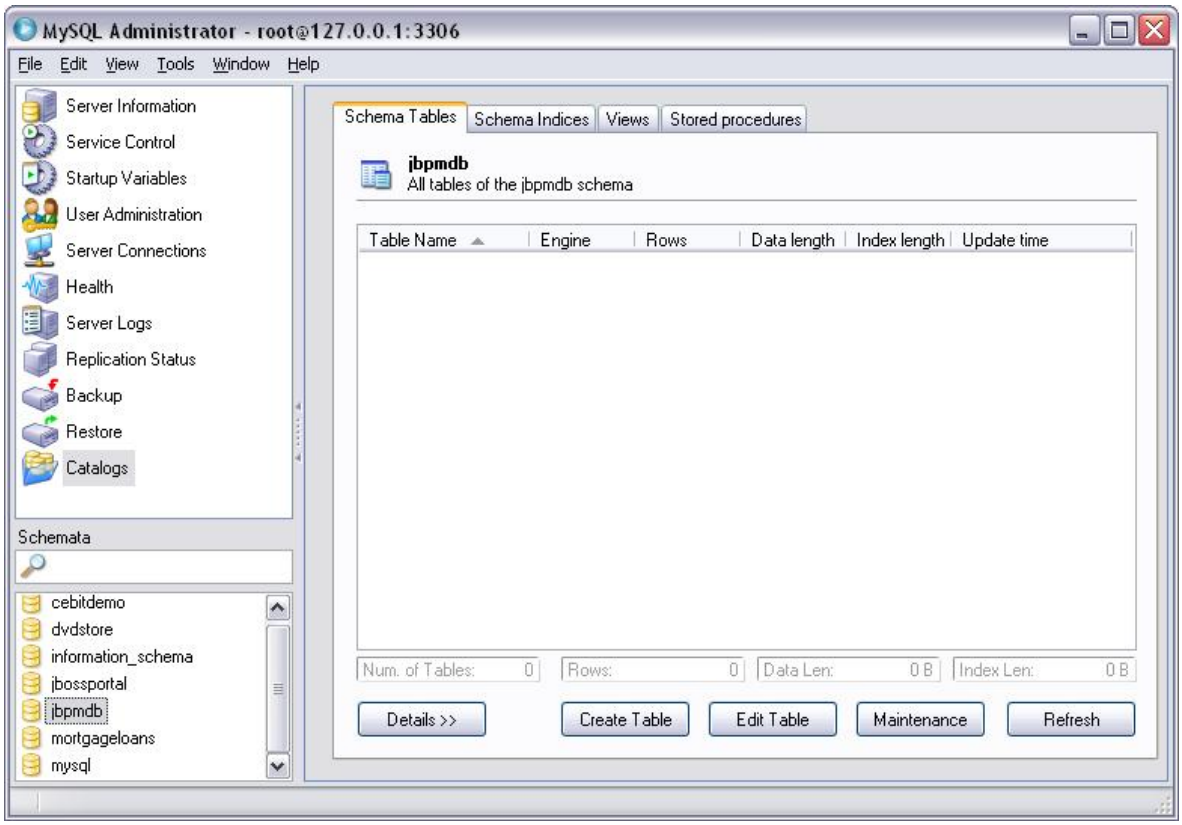

Figure 7.7. The MySQL Administrator after creating the jbpm database under MySQL

You will use the MySQL command line tool to load the database scripts. Open a DOS box or terminal window and type the following command:

```
$ mysql -u root -p
```

You will be prompted for your MySQL password for the root account or whatever account you are using to modify this database. After logging in, type the following command to use the newly created jbpmdb:

```
use jbpmdb
```

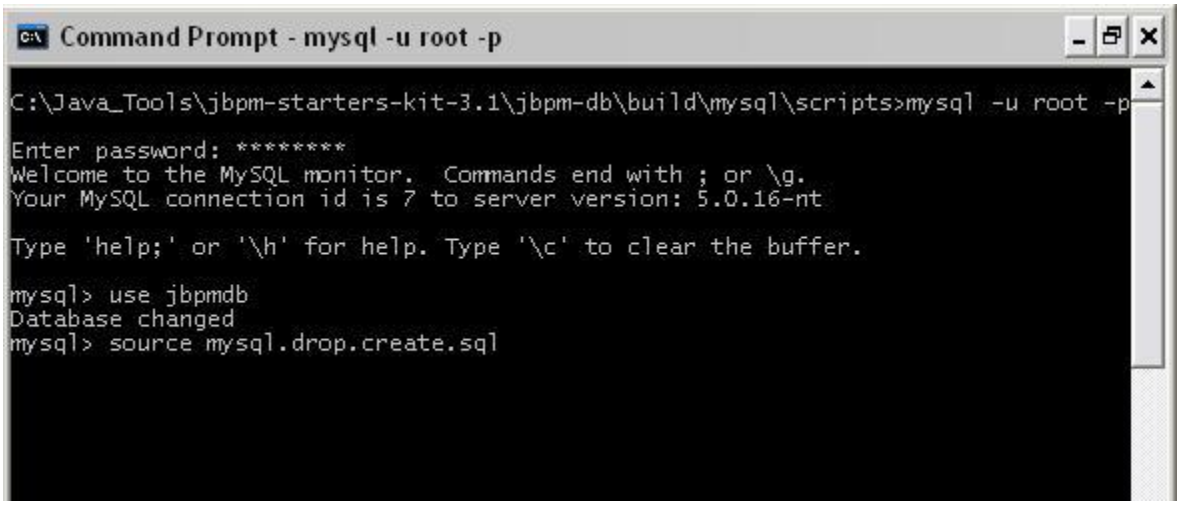

Figure 7.8. Loading the database create scripts for MySQL

Now you can load the database script for jBPM by executing the following command:

```
source mysql.drop.create.sql
```

Once the script executes, you should have the following output in the MySQL command window:

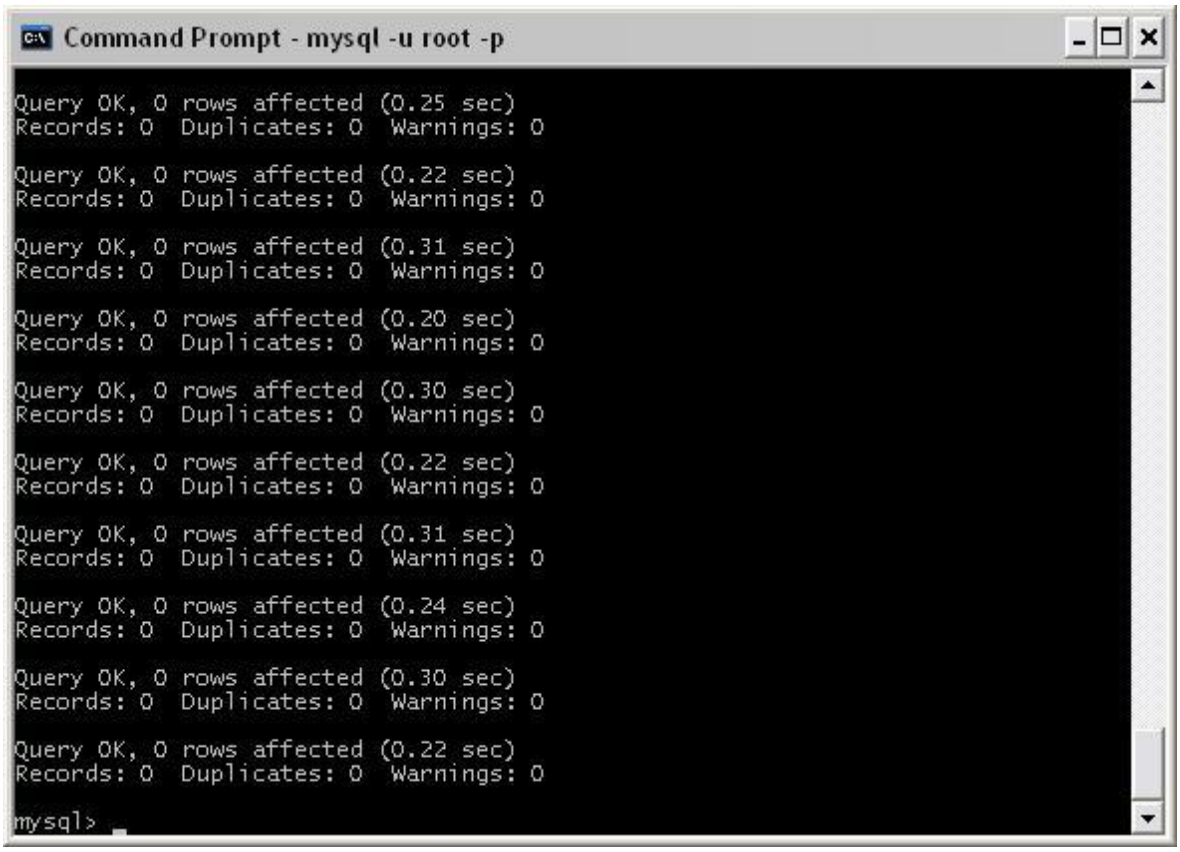

Figure 7.9. Loading the database create scripts for MySQL

7.1.5. Last Steps

After these steps, there is not yet any data present in the tables. For the jBPM web app to work, you should at least create some records in the jbpm_id_user table. In order to have exactly the same entries in this table as the default distribution of the starter's kit running on HSQLDB, we suggest to run the script below.

```
insert into JBPM_ID_USER (ID_, CLASS_, NAME_, EMAIL_, PASSWORD_)
       values ('1', 'U', 'user', 'sample.user@sample.domain', 'user');
insert into JBPM_ID_USER (ID_,CLASS_, NAME_, EMAIL_, PASSWORD_)
       values ('2', 'U', 'manager', 'sample.manager@sample.domain',
 'manager');
insert into JBPM_ID_USER (ID_,CLASS_, NAME_, EMAIL_, PASSWORD_)
       values ('3', 'U', 'shipper', 'sample.shipper@sample.domain',
 'shipper');
insert into JBPM_ID_USER (ID_,CLASS_, NAME_, EMAIL_, PASSWORD_)
       values ('4', 'U', 'admin', 'sample.admin@sample.domain', 'admin');
```

7.1.6. Update the JBoss jBPM Server Configuration

Before we can really use our newly created database with the JBoss jBPM default web app we will have to do some updates to the JBoss jBPM configuration. The location of the server is '${JBPM_SDK_HOME}/jbpm-server'. The first thing we will be doing to update this configuration is create a new datasource that points to our JbpmDB database. In a second step, we will make sure that the default web app is talking to this datasource and not to the HSQLDB datasource anymore.

```
<?xml version="1.0" encoding="UTF-8"?>

<datasources>
  <local-tx-datasource>
    <jndi-name>JbpmDS</jndi-name>
    <connection-url>jdbc:postgresql://localhost:5432/JbpmDB</connection-
url>
    <driver-class>org.postgresql.Driver</driver-class>
    <user-name>user</user-name>
    <password>password</password>
    <metadata>
      <type-mapping>PostgreSQL 8.1</type-mapping>
    </metadata>
  </local-tx-datasource>
</datasources>
```

For MySQL, the datasource definition would look as follows:

```
<?xml version="1.0" encoding="UTF-8"?>

<datasources>
  <local-tx-datasource>
    <jndi-name>JbpmDS</jndi-name>
    <connection-url>jdbc:mysql://localhost:3306/jbpmdb</connection-url>
    <driver-class>com.mysql.jdbc.Driver</driver-class>
```

```
    <user-name>root</user-name>
    <password>root</password>
    <metadata>
      <type-mapping>MySQL</type-mapping>
    </metadata>
  </local-tx-datasource>
</datasources>
```

In order to create a new datasource, you should create a file named e.g. jbpm-ds.xml with the contents shown in the program listing above. Of course it is possible that you have to change some of the values in this file to accommodate for your particular situation. You then simply save this file in the ${JBPM_SDK_HOME}/jbpm-server/server/jbpm/deploy folder. Congratulations, you just created a new DataSource for your JBoss jBPM server. Well, almost... To make things really work you will have to copy the correct JDBC driver to the ${JBPM_SDK_HOME}/jbpm-server/server/jbpm/lib folder. We already used this JDBC driver above when we were installing it in DBVisualizer to be able to browse our newly created database. The file is named 'postgresql-8.1-*.jdbc3.jar' and it can be found in the jdbc sub folder of your PostgreSQL installation folder.

For MySQL, copy the jdbc driver installed from the MySQL ConnectorJ package. The version you need to use is currently the MySQL Connector/J 3.1 available from http://www.mysql.com/products/connector/j/

If you are not using PostgreSQL or MySQL and are wondering how to create your own data source definition for your particular database, you can find sample data source definitions in the JBoss Application Server distribution under the 'docs/examples/jca' folder. if your database has a JDBC driver available for it, you should have no problems using it with jBPM.

Making the default web app talk to the correct datasource is again not very difficult. The first step in doing this is simply locate the 'jboss-service.xml' file in the folder '${JBPM_SDK_HOME}/jbpm-server/server/jbpm/deploy/jbpm.sar/META-INF'. Change the contents of this file with the contents of the listing below. An attentive reader will notice that the only difference is an exchange of the token 'DefaultDS' by 'JbpmDS'.

```
<?xml version="1.0" encoding="UTF-8"?>

<server>
  <mbean code="org.jbpm.db.jmx.JbpmService"
         name="jboss.jbpm:name=DefaultJbpm,service=JbpmService"
         description="Default jBPM Service">
    <attribute name="JndiName">java:/jbpm/JbpmConfiguration</attribute>
    <depends>jboss.jca:service=DataSourceBinding,name=JbpmDS</depends>
  </mbean>
</server>
```

The last thing we have to do to make everything run is a manipulation of the 'jbpm.sar.cfg.jar' file in the '${JBPM_SDK_HOME}/jbpm-server/server/jbpm/deploy/jbpm.sar' folder. You have to extract this file somewhere and open the file named 'hibernate.cfg.xml'. Then replace the section containing the jdbc connection properties. This section should look like shown in the listing below. There are two changes in this file : the hibernate.connection.datasource property should point to the JbpmDS datasource we created as the first step in this section and the hibernate.dialect property should match the PostgreSQL or MySQL dialect.

Below is a sample of the 2 changes required, comment out the version of the dialect you don't need depending on the database you are using. You can get a list of supported database Dialect types from here http://www.hibernate.org/hib_docs/v3/reference/en/html/session-configuration.html#configuration-optional-dialects

```
<?xml version='1.0' encoding='utf-8'?>

<!DOCTYPE hibernate-configuration PUBLIC
          "-//Hibernate/Hibernate Configuration DTD 3.0//EN"
          "http://hibernate.sourceforge.net/hibernate-
configuration-3.0.dtd">

<hibernate-configuration>
  <session-factory>

    <!-- jdbc connection properties -->
    <!-- comment out the dialect not needed! -->

  <property name="hibernate.dialect">org.hibernate.dialect.PostgreSQLDialect</
property>

  <property name="hibernate.dialect">org.hibernate.dialect.MySQLDialect</
property>
    <property name="hibernate.connection.datasource">java:/JbpmDS</
property>

    <!-- other hibernate properties
    <property name="hibernate.show_sql">true</property>
    <property name="hibernate.format_sql">true</property>
    -->

    <!-- ############################################ -->
    <!-- # mapping files with external dependencies # -->
    <!-- ############################################ -->

    ...

  </session-factory>
</hibernate-configuration>
```

Now we are ready to fire up the server, and look if the web app works. You will not be able to start any processes yet, as there are no processes deployed yet. To do this we refer to the document on process definition deployment.

7.2. Database upgrades

In the jbpm.db sub project, you can find:

- An SQL script to create the jBPM 3.0.2 schema (for Hypersonic)
- An SQL script to create the jBPM 3.1 schema (for Hypersonic)

- An SQL script to upgrade from a jBPM 3.0.2 schema to a jBPM 3.1 schema (for Hypersonic)
- The ant scripts to create the schema update

The schema SQL scripts can be found in directory **`hsqldb/upgrade.scripts`**.

To run the schema update tool for your database, follow these guidelines:

- Prerequisite: Make sure you installed the jbpm.db project right besides the jbpm project. In the starters-kit, this is automatically the case. If jbpm is installed in a different location, update the jbpm.3.location dir in build.properties accordingly.
- Prerequisite: You should have the proper JDBC driver jar for your database.
- Update the properties in the build.properties in the root of the jbpm.db project:
  - **upgrade.hibernate.properties**: a properties file that contains the connection properties to your database in hibernate style.
  - **upgrade.libdir**: the directory containing the jdbc driver jars for your database.
  - **upgrade.old.schema.script**: the schema generation script to create the old database schema. (if it already exists, you don't need this property.)
- For creating the old schema and then calculating the differences, run the ant script **`'ant upgrade.db.script'`**
- For only calculating the update script without first loading the old db schema, run the ant script **`'ant upgrade.hibernate.schema.update'`**
- After successful completion, you'll find the upgrade script in **`build/database.upgrade.sql`**

For upgrading from jBPM 3.0.2 to jBPM 3.1, the generated upgrade SQL script (for HSQLDB) is illustrated in the listing below:

```
# New JBPM_MESSAGE table
create table JBPM_MESSAGE (
  ID_ bigint generated by default as identity (start with 1),
  CLASS_ char(1) not null,
  DESTINATION_ varchar(255),
  EXCEPTION_ varchar(255),
  ISSUSPENDED_ bit,
  TOKEN_ bigint,
  TEXT_ varchar(255),
  ACTION_ bigint,
  NODE_ bigint,
  TRANSITIONNAME_ varchar(255),
  TASKINSTANCE_ bigint,
  primary key (ID_)
);

# Added columns
alter table JBPM_ACTION add column ACTIONEXPRESSION_ varchar(255);
alter table JBPM_ACTION add column ISASYNC_ bit;
```

```
alter table JBPM_COMMENT add column VERSION_ integer;
alter table JBPM_ID_GROUP add column PARENT_ bigint;
alter table JBPM_NODE add column ISASYNC_ bit;
alter table JBPM_NODE add column DECISIONEXPRESSION_ varchar(255);
alter table JBPM_NODE add column ENDTASKS_ bit;
alter table JBPM_PROCESSINSTANCE add column VERSION_ integer;
alter table JBPM_PROCESSINSTANCE add column ISSUSPENDED_ bit;
alter table JBPM_RUNTIMEACTION add column VERSION_ integer;
alter table JBPM_SWIMLANE add column ACTORIDEXPRESSION_ varchar(255);
alter table JBPM_SWIMLANE add column POOLEDACTORSEXPRESSION_
 varchar(255);
alter table JBPM_TASK add column ISSIGNALLING_ bit;
alter table JBPM_TASK add column ACTORIDEXPRESSION_ varchar(255);
alter table JBPM_TASK add column POOLEDACTORSEXPRESSION_ varchar(255);
alter table JBPM_TASKINSTANCE add column CLASS_ char(1);
alter table JBPM_TASKINSTANCE add column ISSUSPENDED_ bit;
alter table JBPM_TASKINSTANCE add column ISOPEN_ bit;
alter table JBPM_TIMER add column ISSUSPENDED_ bit;
alter table JBPM_TOKEN add column VERSION_ integer;
alter table JBPM_TOKEN add column ISSUSPENDED_ bit;
alter table JBPM_TOKEN add column SUBPROCESSINSTANCE_ bigint;
alter table JBPM_VARIABLEINSTANCE add column TASKINSTANCE_ bigint;

# Added constraints
alter table JBPM_ID_GROUP add constraint FK_ID_GRP_PARENT foreign key
 (PARENT_) references JBPM_ID_GROUP;
alter table JBPM_MESSAGE add constraint FK_MSG_TOKEN foreign key (TOKEN_)
 references JBPM_TOKEN;
alter table JBPM_MESSAGE add constraint FK_CMD_NODE foreign key (NODE_)
 references JBPM_NODE;
alter table JBPM_MESSAGE add constraint FK_CMD_ACTION foreign key
 (ACTION_) references JBPM_ACTION;
alter table JBPM_MESSAGE add constraint FK_CMD_TASKINST foreign key
 (TASKINSTANCE_) references JBPM_TASKINSTANCE;
alter table JBPM_TOKEN add constraint FK_TOKEN_SUBPI foreign key
 (SUBPROCESSINSTANCE_) references JBPM_PROCESSINSTANCE;
alter table JBPM_VARIABLEINSTANCE add constraint FK_VAR_TSKINST foreign
 key (TASKINSTANCE_) references JBPM_TASKINSTANCE;
```

7.3. Starting hsqldb manager on JBoss

Not really crucial for jBPM, but in some situations during development, it can be convenient to open the hypersonic database manager that gives you access to the data in the JBoss hypersonic database.

Start by opening a browser and navigating to the jBPM server JMX console. The URL you should use in your browser for doing this is : http://localhost:8080/jmx-console. Of course this will look slightly different if you are running jBPM on another machine or on another port than the default one. A screenshot of the resulting page is shown in the figure below.

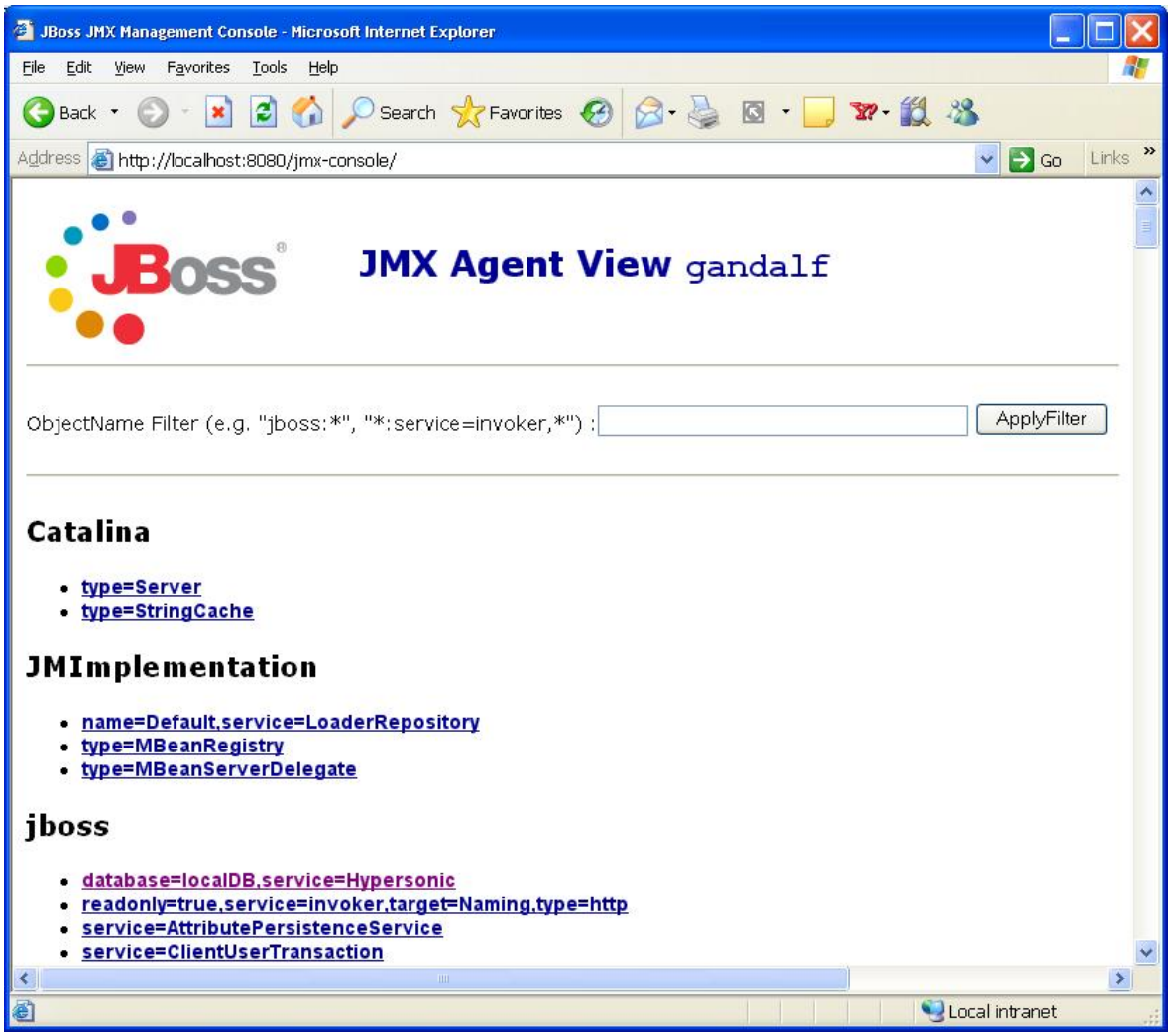

Figure 7.10. The JBoss jBPM JMX Console

If you click on the link 'database=jbpmDB,service=Hypersonic' under the JBoss entries, you will see the JMX MBean view of the HSQLDB database manager. Scrolling a bit down on this page, in the operations section, you will see the 'startDatabaseManager()' operation. This is illustrated in the screenshot below.

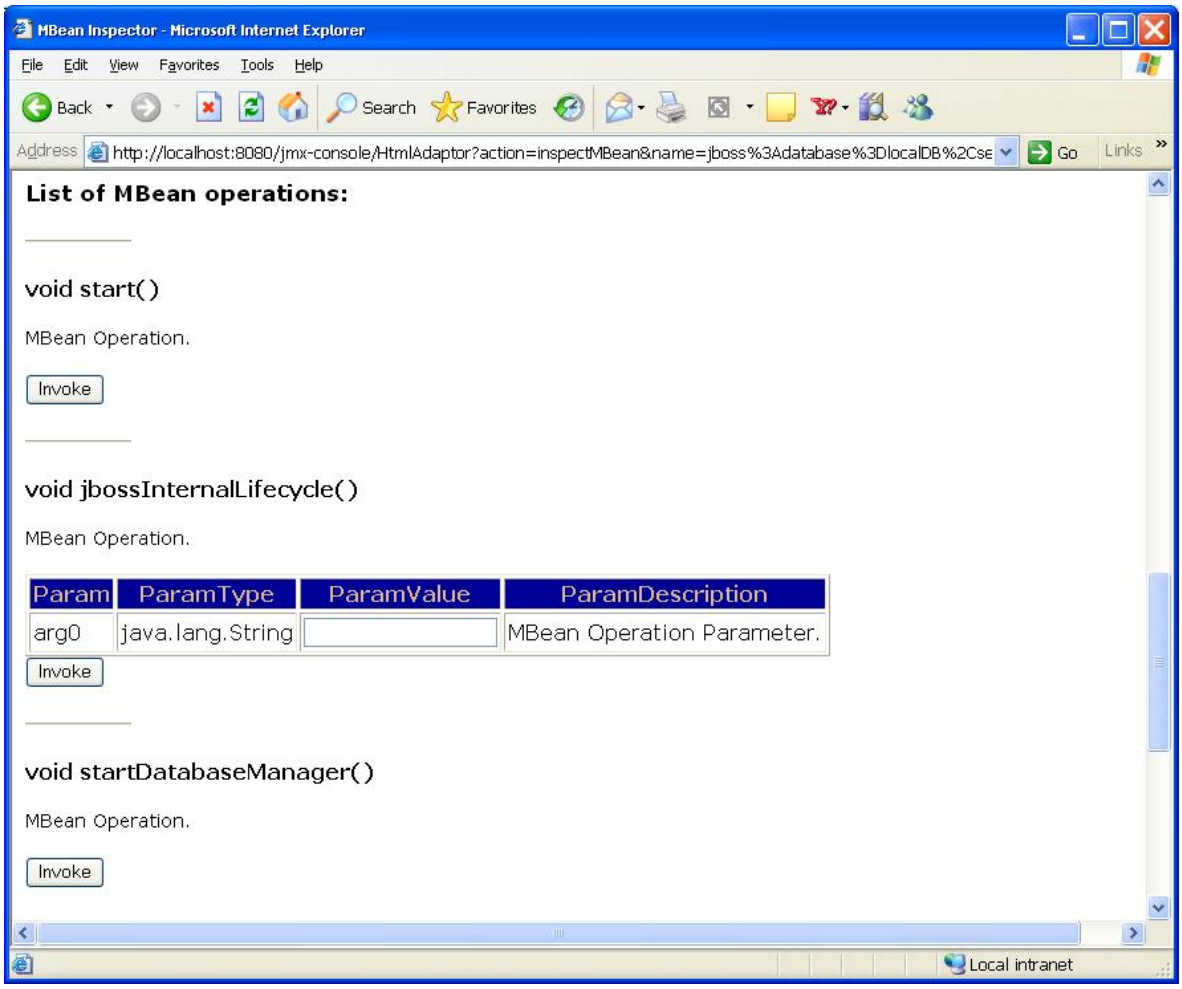

Figure 7.11. The HSQLDB MBean

Clicking the invoke button will start the HSQLDB Database Manager application. This is a rather harsh database client tool, but it works adequately for our purposes of executing this generated script. You may have to ALT-TAB to get to view this application as it may be covered by another window. The figure below shows this application with the above script loaded and ready to execute. Pushing the 'Execute SQL' button will execute the script and effectively update your database.

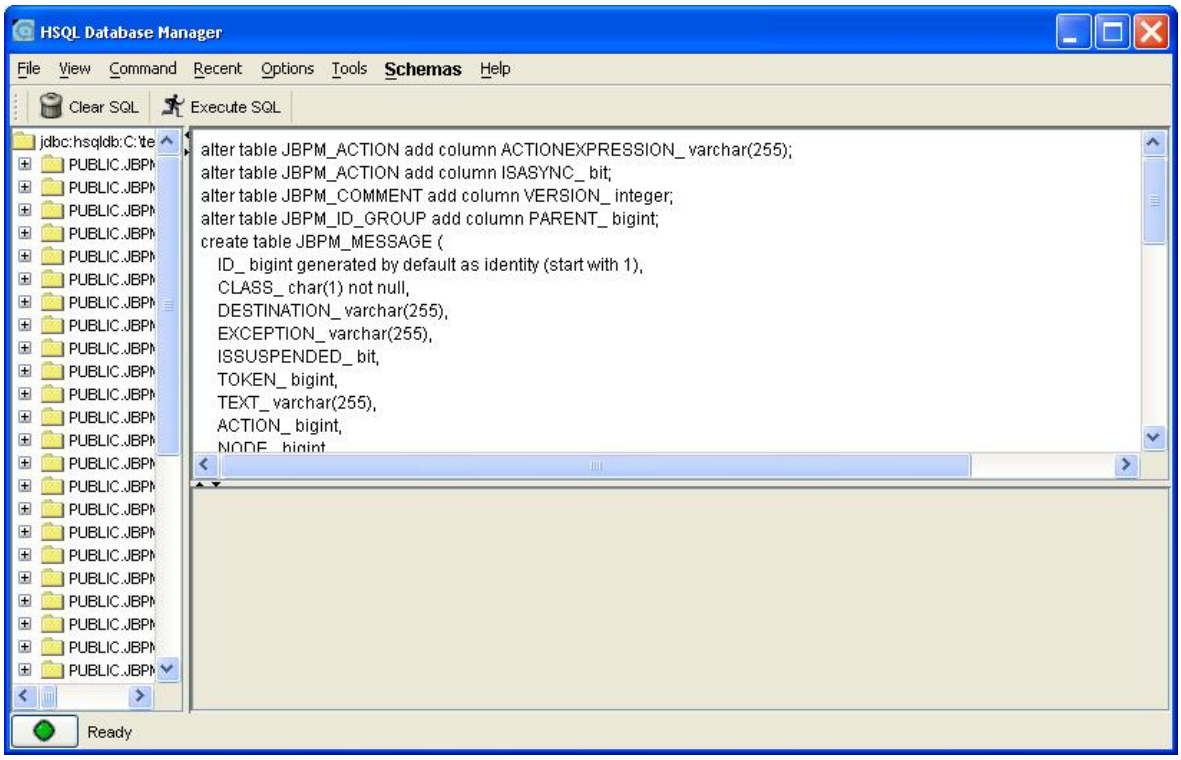

Figure 7.12. The HSQLDB Database Manager

Process Modeling

8.1. Overview

A process definition represents a formal specification of a business process and is based on a directed graph. The graph is composed of nodes and transitions. Every node in the graph is of a specific type. The type of the node defines the runtime behavior. A process definition has exactly one start state.

A token is one path of execution. A token is the runtime concept that maintains a pointer to a node in the graph.

A process instance is one execution of a process definition. When a process instance is created, a token is created for the main path of execution. This token is called the root token of the process instance and it is positioned in the start state of the process definition.

A signal instructs a token to continue graph execution. When receiving an unnamed signal, the token will leave its current node over the default leaving transition. When a transition-name is specified in the signal, the token will leave its node over the specified transition. A signal given to the process instance is delegated to the root token.

After the token has entered a node, the node is executed. Nodes themselves are responsible for the continuation of the graph execution. Continuation of graph execution is done by making the token leave the node. Each node type can implement a different behavior for the continuation of the graph execution. A node that does not propagate execution will behave as a state.

Actions are pieces of java code that are executed upon events in the process execution. The graph is an important instrument in the communication about software requirements. But the graph is just one view (projection) of the software being produced. It hides many technical details. Actions are a mechanism to add technical details outside of the graphical representation. Once the graph is put in place, it can be decorated with actions. The main event types are entering a node, leaving a node and taking a transition.

8.2. Process graph

The basis of a process definition is a graph that is made up of nodes and transitions. That information is expressed in an xml file called **`processdefinition.xml`**. Each node has a type like e.g. state, decision, fork, join,... Each node has a set of leaving transitions. A name can be given to the transitions that leave a node in order to make them distinct. For example: The following diagram shows a process graph of the jBAY auction process.

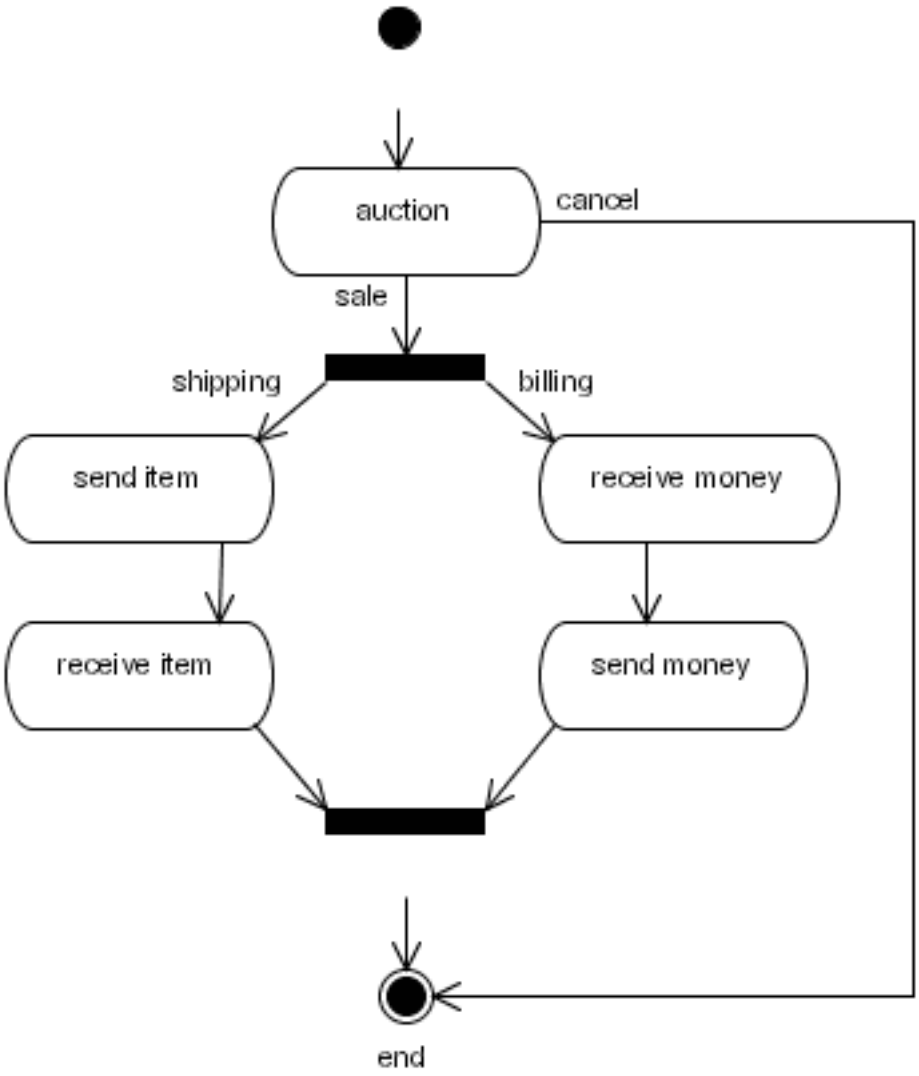

Figure 8.1. The auction process graph

Below is the process graph of the jBAY auction process represented as xml:

```
<process-definition>

  <start-state>
    <transition to="auction" />
  </start-state>

  <state name="auction">
    <transition name="auction ends" to="salefork" />
    <transition name="cancel" to="end" />
  </state>

  <fork name="salefork">
    <transition name="shipping" to="send item" />
    <transition name="billing" to="receive money" />
  </fork>
```

```
  <state name="send item">
    <transition to="receive item" />
  </state>

  <state name="receive item">
    <transition to="salejoin" />
  </state>

  <state name="receive money">
    <transition to="send money" />
  </state>

  <state name="send money">
    <transition to="salejoin" />
  </state>

  <join name="salejoin">
    <transition to="end" />
  </join>

  <end-state name="end" />

</process-definition>
```

8.3. Nodes

A process graph is made up of nodes and transitions. For more information about the graph and its execution model, refer to *Chapter 3, Graph Oriented Programming*.

Each node has a specific type. The node type determines what will happen when an execution arrives in the node at runtime. jBPM has a set of node types that you can use. Alternatively, you can write custom code for implementing your own specific node behavior.

8.3.1. Node responsibilities

Each node has 2 main responsibilities: First, it can execute plain java code. Typically the plain java code relates to the function of the node. E.g. creating a few task instances, sending a notification, updating a database,... Secondly, a node is responsible for propagating the process execution. Basically, each node has the following options for propagating the process execution:

- **1. not propagate the execution.** In that case the node behaves as a wait state.
- **2. propagate the execution over one of the leaving transitions of the node.** This means that the token that originally arrived in the node is passed over one of the leaving transitions with the API call executionContext.leaveNode(String). The node will now act as an automatic node in the sense it can execute some custom programming logic and then continue process execution automatically without waiting.
- **3. create new paths of execution.** A node can decide to create new tokens. Each new token represents a new path of execution and each new token can be launched over the node's leaving transitions. A good example of this kind of behavior is the fork node.

- **4. end paths of execution.** A node can decide to end a path of execution. That means that the token is ended and the path of execution is finished.
- **5. more general, a node can modify the whole runtime structure of the process instance.** The runtime structure is a process instance that contains a tree of tokens. Each token represents a path of execution. A node can create and end tokens, put each token in a node of the graph and launch tokens over transitions.

jBPM contains --as any workflow and BPM engine-- a set of pre-implemented node types that have a specific documented configuration and behavior. But the unique thing about jBPM and the Graph Oriented Programming foundation[1] is that we open up the model for developers. Developers can write their own node behavior very easy and use it in a process.

That is where traditional workflow and BPM systems are much more closed. They usually supply a fixed set of node types (called the process language). Their process language is closed and the execution model is hidden in the runtime environment. Research of *workflow patterns*[2] has shown that any process language is not powerful enough. We have decided for a simple model and allow developers to write their own node types. That way the JPDL process language is open ended.

Next, we discuss the most important node types of JPDL.

8.3.2. Nodetype task-node

A task node represents one or more tasks that are to be performed by humans. So when execution arrives in a task node, task instances will be created in the task lists of the workflow participants. After that, the node will behave as a wait state. So when the users perform their task, the task completion will trigger the resuming of the execution. In other words, that leads to a new signal being called on the token.

8.3.3. Nodetype state

A state is a bare-bones wait state. The difference with a task node is that no task instances will be created in any task list. This can be useful if the process should wait for an external system. E.g. upon entry of the node (via an action on the node-enter event), a message could be sent to the external system. After that, the process will go into a wait state. When the external system send a response message, this can lead to a token.signal(), which triggers resuming of the process execution.

8.3.4. Nodetype decision

Actually there are 2 ways to model a decision. The distinction between the two is based on *who* is making the decision. Should the decision made by the process (read: specified in the process definition). Or should an external entity provide the result of the decision.

When the decision is to be taken by the process, a decision node should be used. There are basically 2 ways to specify the decision criteria. Simplest is by adding condition elements on the transitions. Conditions are EL expressions or beanshell scripts that return a boolean.

At runtime the decision node will FIRST loop over its leaving transitions THAT HAVE a condition specified. It will evaluate those transitions first in the order as specified in the xml. The first transition for which the condition resolves to 'true' will be taken. If all transitions with a condition resolve to false, the default transition (the first in the XML) is taken.

[1] *Chapter 3, Graph Oriented Programming*.
[2] http://www.workflowpatterns.com

Another approach is to use an expression that returns the name of the transition to take. With the 'expression' attribute, you can specify an expression on the decision that has to resolve to one of the leaving transitions of the decision node.

Next approach is the 'handler' element on the decision, that element can be used to specify an implementation of the DecisionHandler interface can be specified on the decision node. Then the decision is calculated in a java class and the selected leaving transition is returned by the decide-method of the DecisionHandler implementation.

When the decision is taken by an external party (meaning: not part of the process definition), you should use multiple transitions leaving a state or wait state node. Then the leaving transition can be provided in the external trigger that resumes execution after the wait state is finished. E.g. **`Token.signal(String transitionName)`** and **`TaskInstance.end(String transitionName)`**.

8.3.5. Nodetype fork

A fork splits one path of execution into multiple concurrent paths of execution. The default fork behavior is to create a child token for each transition that leaves the fork, creating a parent-child relation between the token that arrives in the fork.

8.3.6. Nodetype join

The default join assumes that all tokens that arrive in the join are children of the same parent. This situation is created when using the fork as mentioned above and when all tokens created by a fork arrive in the same join. A join will end every token that enters the join. Then the join will examine the parent-child relation of the token that enters the join. When all sibling tokens have arrived in the join, the parent token will be propagated over the (unique!) leaving transition. When there are still sibling tokens active, the join will behave as a wait state.

8.3.7. Nodetype node

The type node serves the situation where you want to write your own code in a node. The nodetype node expects one sub-element action. The action is executed when the execution arrives in the node. The code you write in the actionhandler can do anything you want but it is also responsible for propagating the execution[3].

This node can be used if you want to use a JavaAPI to implement some functional logic that is important for the business analyst. By using a node, the node is visible in the graphical representation of the process. For comparison, actions --covered next-- will allow you to add code that is invisible in the graphical representation of the process, in case that logic is not important for the business analyst.

8.4. Transitions

Transitions have a source node and a destination node. The source node is represented with the property **`from`** and the destination node is represented by the property **`to`**.

A transition can optionally have a name. Note that most of the jBPM features depend on the uniqueness of the transition name. If more than one transition has the same name, the first transition with the given name is taken. In case duplicate transition names occur in a node,

[3] *Section 8.3.1, "Node responsibilities"*.

the method **`Map getLeavingTransitionsMap()`** will return less elements than **`List getLeavingTransitions()`**.

The default transition is the first transition in the list.

8.5. Actions

Actions are pieces of java code that are executed upon events in the process execution. The graph is an important instrument in the communication about software requirements. But the graph is just one view (projection) of the software being produced. It hides many technical details. Actions are a mechanism to add technical details outside of the graphical representation. Once the graph is put in place, it can be decorated with actions. This means that java code can be associated with the graph without changing the structure of the graph. The main event types are entering a node, leaving a node and taking a transition.

Note the difference between an action that is placed in an event versus an action that is placed in a node. Actions that are put in an event are executed when the event fires. Actions on events have no way to influence the flow of control of the process. It is similar to the observer pattern. On the other hand, an action that is put on a node[4] has the responsibility of propagating the execution[5].

Let's look at an example of an action on an event. Suppose we want to do a database update on a given transition. The database update is technically vital but it is not important to the business analyst.

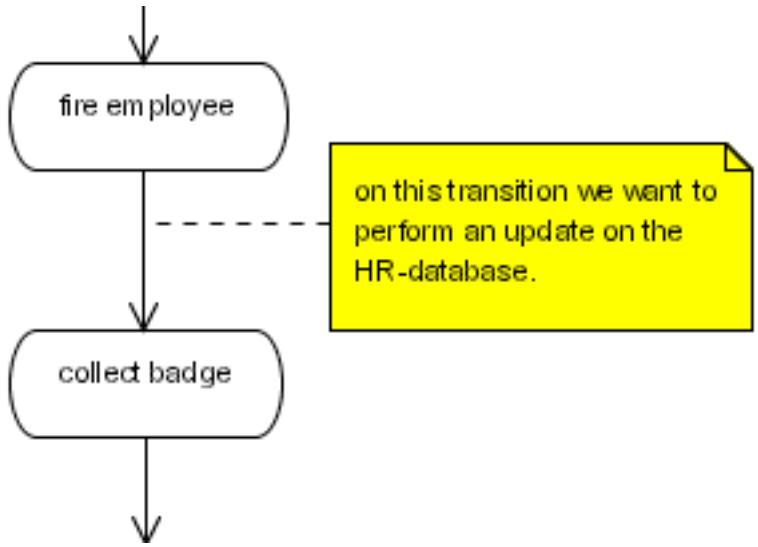

Figure 8.2. A database update action

```
public class RemoveEmployeeUpdate implements ActionHandler {
  public void execute(ExecutionContext ctx) throws Exception {
    // get the fired employee from the process variables.
    String firedEmployee = (String)
 ctx.getContextInstance().getVariable("fired employee");

    // by taking the same database connection as used for the jbpm
 updates, we
```

[4] *Section 8.3.7, "Nodetype node"*.
[5] *Section 8.3.1, "Node responsibilities"*.

```
    // reuse the jbpm transaction for our database update.
    Connection connection =
 ctx.getProcessInstance().getJbpmSession().getSession().getConnection();
    Statement statement = connection.createStatement();
    statement.execute("DELETE FROM EMPLOYEE WHERE ...");
    statement.execute();
    statement.close();
  }
}
```

```
<process-definition name="yearly evaluation">
  <state name="fire employee">
    <transition to="collect badge">
      <action class="com.nomercy.hr.RemoveEmployeeUpdate" />
    </transition>
  </state>

  <state name="collect badge">

</process-definition>
```

8.5.1. Action configuration

For more information about adding configurations to your custom actions and how to specify the configuration in the **`processdefinition.xml`**, see *Section 17.2.3, "Configuration of delegations"*

8.5.2. Action references

Actions can be given a name. Named actions can be referenced from other locations where actions can be specified. Named actions can also be put as child elements in the process definition.

This feature is interesting if you want to limit duplication of action configurations (e.g. when the action has complicated configurations). Another use case is execution or scheduling of runtime actions.

8.5.3. Events

Events specify moments in the execution of the process. The jBPM engine will fire events during graph execution. This occurs when jbpm calculates the next state (read: processing a signal). An event is always relative to an element in the process definition like e.g. the process definition, a node or a transition. Most process elements can fire different types of events. A node for example can fire a **`node-enter`** event and a **`node-leave`** event. Events are the hooks for actions. Each event has a list of actions. When the jBPM engine fires an event, the list of actions is executed.

8.5.4. Event propagation

Superstates create a parent-child relation in the elements of a process definition. Nodes and transitions contained in a superstate have that superstate as a parent. Top level elements have the process definition as a parent. The process definition does not have a parent. When an event is fired, the event will be propagated up the parent hierarchy. This allows e.g. to capture all transition events in a process and associate actions with these events in a centralized location.

8.5.5. Script

A script is an action that executes a beanshell script. For more information about beanshell, see *the beanshell website*[6]. By default, all process variables are available as script-variables and no script-variables will be written to the process variables. Also the following script-variables will be available :

- executionContext
- token
- node
- task
- taskInstance

```
<process-definition>
  <event type="node-enter">
    <script>
      System.out.println("this script is entering node "+node);
    </script>
  </event>
  ...
</process-definition>
```

To customize the default behavior of loading and storing variables into the script, the **`variable`** element can be used as a sub-element of script. In that case, the script expression also has to be put in a sub-element of script: **`expression`**.

```
<process-definition>
  <event type="process-end">
    <script>
      <expression>
        a = b + c;
      </expression>
      <variable name='XXX' access='write' mapped-name='a' />
      <variable name='YYY' access='read' mapped-name='b' />
      <variable name='ZZZ' access='read' mapped-name='c' />
    </script>
  </event>
  ...
</process-definition>
```

Before the script starts, the process variables **YYY** and **ZZZ** will be made available to the script as script-variables **b** and **c** respectively. After the script is finished, the value of script-variable **a** is stored into the process variable **XXX**.

If the **`access`** attribute of **`variable`** contains '**`read`**', the process variable will be loaded as a script-variable before script evaluation. If the **`access`** attribute contains '**`write`**', the script-variable will be stored as a process variable after evaluation. The attribute **`mapped-name`** can make the process

[6] http://www.beanshell.org/

variable available under another name in the script. This can be handy when your process variable names contain spaces or other invalid script-literal-characters.

8.5.6. Custom events

Note that it's possible to fire your own custom events at will during the execution of a process. Events are uniquely defined by the combination of a graph element (nodes, transitions, process definitions and superstates are graph elements) and an event-type (java.lang.String). jBPM defines a set of events that are fired for nodes, transitions and other graph elements. But as a user, you are free to fire your own events. In actions, in your own custom node implementations, or even outside the execution of a process instance, you can call the **`GraphElement.fireEvent(String eventType, ExecutionContext executionContext);`**. The names of the event types can be chosen freely.

8.6. Superstates

A Superstate is a group of nodes. Superstates can be nested recursively. Superstates can be used to bring some hierarchy in the process definition. For example, one application could be to group all the nodes of a process in phases. Actions can be associated with superstate events. A consequence is that a token can be in multiple nested nodes at a given time. This can be convenient to check whether a process execution is e.g. in the start-up phase. In the jBPM model, you are free to group any set of nodes in a superstate.

8.6.1. Superstate transitions

All transitions leaving a superstate can be taken by tokens in nodes contained within the super state. Transitions can also arrive in superstates. In that case, the token will be redirected to the first node in the superstate. Nodes from outside the superstate can have transitions directly to nodes inside the superstate. Also, the other way round, nodes within superstates can have transitions to nodes outside the superstate or to the superstate itself. Superstates also can have self references.

8.6.2. Superstate events

There are 2 events unique to superstates: **`superstate-enter`** and **`superstate-leave`**. These events will be fired no matter over which transitions the node is entered or left respectively. As long as a token takes transitions within the superstate, these events are not fired.

Note that we have created separate event types for states and superstates. This is to make it easy to distinct between superstate events and node events that are propagated from within the superstate.

8.6.3. Hierarchical names

Node names have to be unique in their scope. The scope of the node is its node-collection. Both the process definition and the superstate are node collections. To refer to nodes in superstates, you have to specify the relative, slash (/) separated name. The slash separates the node names. Use '..' to refer to an upper level. The next example shows how to reference a node in a superstate:

```
<process-definition>
  <state name="preparation">
    <transition to="phase one/invite murphy"/>
  </state>
  <super-state name="phase one">
```

```
    <state name="invite murphy"/>
  </super-state>
</process-definition>
```

The next example will show how to go up the superstate hierarchy

```
<process-definition>
  <super-state name="phase one">
    <state name="preparation">
      <transition to="../phase two/invite murphy"/>
    </state>
  </super-state>
  <super-state name="phase two">
    <state name="invite murphy"/>
  </super-state>
</process-definition>
```

8.7. Exception handling

The exception handling mechanism of jBPM only applies to java exceptions. Graph execution on itself cannot result in problems. It is only the execution of delegation classes that can lead to exceptions.

On **`process-definition`**s, **`node`**s and **`transition`**s, a list of **`exception-handler`**s can be specified. Each **`exception-handler`** has a list of actions. When an exception occurs in a delegation class, the process element parent hierarchy is searched for an appropriate **`exception-handler`**. When it is found, the actions of the **`exception-handler`** are executed.

Note that the exception handling mechanism of jBPM is not completely similar to the java exception handling. In java, a caught exception can have an influence on the control flow. In the case of jBPM, control flow cannot be changed by the jBPM exception handling mechanism. The exception is either caught or uncaught. Uncaught exceptions are thrown to the client (e.g. the client that called the **`token.signal()`**) or the exception is caught by a jBPM **`exception-handler`**. For caught exceptions, the graph execution continues as if no exception has occurred.

Note that in an **`action`** that handles an exception, it is possible to put the token in an arbitrary node in the graph with **`Token.setNode(Node node)`**.

8.8. Process composition

Process composition is supported in jBPM by means of the **`process-state`**. The process state is a state that is associated with another process definition. When graph execution arrives in the process state, a new process instance of the sub-process is created and it is associated with the path of execution that arrived in the process state. The path of execution of the super process will wait until the sub process instance has ended. When the sub process instance ends, the path of execution of the super process will leave the process state and continue graph execution in the super process.

```
<process-definition name="hire">
  <start-state>
    <transition to="initial interview" />
  </start-state>
  <process-state name="initial interview">
```

```
    <sub-process name="interview" />
    <variable name="a" access="read,write" mapped-name="aa" />
    <variable name="b" access="read" mapped-name="bb" />
    <transition to="..." />
  </process-state>
  ...
</process-definition>
```

This 'hire' process contains a **`process-state`** that spawns an 'interview' process. When execution arrives in the 'first interview', a new execution (=process instance) of the 'interview' process is created. If no explicit version is specified, the latest version of the sub process as known when deploying the 'hire' process is used. To make jBPM instantiate a specific version the optional **`version`** attribute can be specified. To postpone binding the specified or latest version until actually creating the sub process, the optional **`binding`** attribute should be set to **`late`**. Then variable 'a' from the hire process is copied into variable 'aa' from the interview process. The same way, hire variable 'b' is copied into interview variable 'bb'. When the interview process finishes, only variable 'aa' from the interview process is copied back into the 'a' variable of the hire process.

In general, When a sub-process is started, all **`variable`**s with **`read`** access are read from the super process and fed into the newly created sub process before the signal is given to leave the start state. When the sub process instances is finished, all the **`variable`**s with **`write`** access will be copied from the sub process to the super process. The **`mapped-name`** attribute of the **`variable`** element allows you to specify the variable name that should be used in the sub process.

8.9. Custom node behavior

In jBPM, it's quite easy to write your own custom nodes. For creating custom nodes, an implementation of the ActionHandler has to be written. The implementation can execute any business logic, but also has the responsibility to propagate the graph execution. Let's look at an example that will update an ERP-system. We'll read an amount from the ERP-system, add an amount that is stored in the process variables and store the result back in the ERP-system. Based on the size of the amount, we have to leave the node via the 'small amounts' or the 'large amounts' transition.

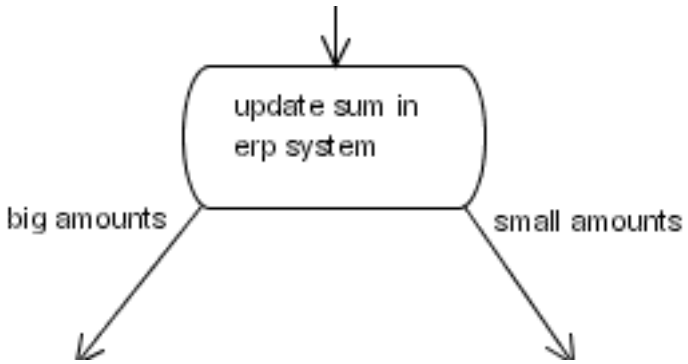

Figure 8.3. The update erp example process snippet

```
public class AmountUpdate implements ActionHandler {
  public void execute(ExecutionContext ctx) throws Exception {
    // business logic
    Float erpAmount = ...get amount from erp-system...;
    Float processAmount = (Float)
 ctx.getContextInstance().getVariable("amount");
```

```
    float result = erpAmount.floatValue() + processAmount.floatValue();
    ...update erp-system with the result...;

    // graph execution propagation
    if (result >
<xslthl:number>5000</xslthl:number>
) {
      ctx.leaveNode(ctx, "big amounts");
    } else {
      ctx.leaveNode(ctx, "small amounts");
    }
  }
}
```

It is also possible to create and join tokens in custom node implementations. For an example on how to do this, check out the Fork and Join node implementation in the jbpm source code.

8.10. Graph execution

The graph execution model of jBPM is based on interpretation of the process definition and the chain of command pattern.

Interpretation of the process definition means that the process definition data is stored in the database. At runtime the process definition information is used during process execution. Note for the concerned : we use Hibernate's second level cache to avoid loading of definition information at runtime. Since the process definitions don't change (see process versioning) hibernate can cache the process definitions in memory.

The chain of command pattern means that each node in the graph is responsible for propagating the process execution. If a node does not propagate execution, it behaves as a wait state.

The idea is to start execution on process instances and that the execution continues till it enters a wait state.

A token represents a path of execution. A token has a pointer to a node in the process graph. During wait states, the tokens can be persisted in the database. Now we are going to look at the algorithm for calculating the execution of a token. Execution starts when a signal is sent to a token. The execution is then passed over the transitions and nodes via the chain of command pattern. These are the relevant methods in a class diagram.

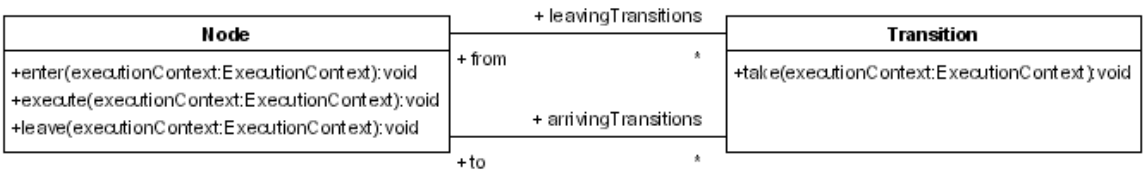

Figure 8.4. The graph execution related methods

When a token is in a node, signals can be sent to the token. Sending a signal is an instruction to start execution. A signal must therefore specify a leaving transition of the token's current

node. The first transition is the default. In a signal to a token, the token takes its current node and calls the **`Node.leave(ExecutionContext,Transition)`** method. Think of the ExecutionContext as a Token because the main object in an ExecutionContext is a Token. The **`Node.leave(ExecutionContext,Transition)`** method will fire the **`node-leave`** event and call the **`Transition.take(ExecutionContext)`**. That method will fire the **`transition`** event and call the **`Node.enter(ExecutionContext)`** on the destination node of the transition. That method will fire the **`node-enter`** event and call the **`Node.execute(ExecutionContext)`**. Each type of node has its own behaviour that is implemented in the execute method. Each node is responsible for propagating graph execution by calling the **`Node.leave(ExecutionContext,Transition)`** again. In summary:

- Token.signal(Transition)
- --> Node.leave(ExecutionContext,Transition)
- --> Transition.take(ExecutionContext)
- --> Node.enter(ExecutionContext)
- --> Node.execute(ExecutionContext)

Note that the complete calculation of the next state, including the invocation of the actions is done in the thread of the client. A common misconception is that all calculations *must* be done in the thread of the client. As with any asynchronous invocation, you can use asynchronous messaging (JMS) for that. When the message is sent in the same transaction as the process instance update, all synchronization issues are taken care of. Some workflow systems use asynchronous messaging between all nodes in the graph. But in high throughput environments, this algorithm gives much more control and flexibility for tweaking performance of a business process.

8.11. Transaction demarcation

As explained in *Section 8.10, "Graph execution"* and *Chapter 3, Graph Oriented Programming*, jBPM runs the process in the thread of the client and is by nature synchronous. Meaning that the **`token.signal()`** or **`taskInstance.end()`** will only return when the process has entered a new wait state.

The jPDL feature that we describe here from a modeling perspective is *Section 3.3.4, "Asynchronous continuations"*.

In most situations this is the most straightforward approach because the process execution can easily be bound to server side transactions: the process moves from one state to the next in one transaction.

In some scenarios where in-process calculations take a lot of time, this behavior might be undesirable. To cope with this, jBPM includes an asynchronous messaging system that allows to continue a process in an asynchronous manner. Of course, in a java enterprise environment, jBPM can be configured to use a JMS message broker instead of the built in messaging system.

In any node, jPDL supports the attribute **`async="true"`**. Asynchronous nodes will not be executed in the thread of the client. Instead, a message is sent over the asynchronous messaging system and the thread is returned to the client (meaning that the **`token.signal()`** or **`taskInstance.end()`** will return).

Note that the jbpm client code can now commit the transaction. The sending of the message should be done in the same transaction as the process updates. So the net result of the

transaction is that the token has moved to the next node (which has not yet been executed) and a **`org.jbpm.command.ExecuteNodeCommand`**-message has been sent on the asynchronous messaging system to the jBPM Command Executor.

The jBPM Command Executor reads commands from the queue and executes them. In the case of the **`org.jbpm.command.ExecuteNodeCommand`**, the process will be continued with executing the node. Each command is executed in a separate transaction.

So in order for asynchronous processes to continue, a jBPM Command Executor needs to be running. The simplest way to do that is to configure the **`CommandExecutionServlet`** in your web application. Alternatively, you should make sure that the CommandExecutor thread is up and running in any other way.

As a process modeler, you should not really be concerned with all this asynchronous messaging. The main point to remember is transaction demarcation: By default jBPM will operate in the transaction of the client, doing the whole calculation until the process enters a wait state. Use **`async="true"`** to demarcate a transaction in the process.

Let's look at an example:

```
<start-state>
  <transition to="one" />
</start-state>
<node async="true" name="one">
  <action class="com...MyAutomaticAction" />
  <transition to="two" />
</node>
<node async="true" name="two">
  <action class="com...MyAutomaticAction" />
  <transition to="three" />
</node>
<node async="true" name="three">
  <action class="com...MyAutomaticAction" />
  <transition to="end" />
</node>
<end-state name="end" />
...
```

Client code to interact with process executions (starting and resuming) is exactly the same as with normal (synchronous) processes:

```
//start a transaction
JbpmContext jbpmContext = jbpmConfiguration.createContext();
try {
  ProcessInstance processInstance = jbpmContext.newProcessInstance("my
 async process");
  processInstance.signal();
  jbpmContext.save(processInstance);
} finally {
  jbpmContext.close();
}
```

After this first transaction, the root token of the process instance will point to node **one** and a **`ExecuteNodeCommand`**message will have been sent to the command executor.

In a subsequent transaction, the command executor will read the message from the queue and execute node **one**. The action can decide to propagate the execution or enter a wait state. If the action decides to propagate the execution, the transaction will be ended when the execution arrives at node two. And so on, and so on...

Context

Context is about process variables. Process variables are key-value pairs that maintain information related to the process instance. Since the context must be able to be stored in a database, some minor limitations apply.

9.1. Accessing variables

org.jbpm.context.exe.ContextInstance serves as the central interface to work with process variables. You can obtain the ContextInstance from a ProcessInstance like this :

```
ProcessInstance processInstance = ...;
ContextInstance contextInstance = (ContextInstance)
 processInstance.getInstance(ContextInstance.class);
```

The most basic operations are

```
void ContextInstance.setVariable(String variableName, Object value);
void ContextInstance.setVariable(String variableName, Object value, Token
 token);
Object ContextInstance.getVariable(String variableName);
Object ContextInstance.getVariable(String variableName, Token token);
```

The variable names are java.lang.String. By default, jBPM supports the following value types:

- **java.lang.String**
- **java.lang.Boolean**
- **java.lang.Character**
- **java.lang.Float**
- **java.lang.Double**
- **java.lang.Long**
- **java.lang.Byte**
- **java.lang.Short**
- **java.lang.Integer**
- **java.util.Date**
- **byte[]**
- **java.io.Serializable**
- **classes that are able to be persisted with hibernate**

Also an untyped null value can be stored persistently.

All other types can be stored in the process variables without any problem. But it will cause an exception when you try to save the process instance.

To configure jBPM for storing hibernate persistent objects in the variables, see Storing hibernate persistent objects.

9.2. Variable lifetime

Variables do not have to be declared in the process archive. At runtime, you can just put any java object in the variables. If that variable was not present, it will be created. Just the same as with a plain **`java.util.Map`**.

Variables can be deleted with

```
ContextInstance.deleteVariable(String variableName);
ContextInstance.deleteVariable(String variableName, Token token);
```

Automatic changing of types is now supported. This means that it is allowed to overwrite a variable with a value of a different type. Of course, you should try to limit the number of type changes since this creates a more db communication then a plain update of a column.

9.3. Variable persistence

The variables are a part of the process instance. Saving the process instance in the database, brings the database in sync with the process instance. The variables are created, updated and deleted from the database as a result of saving (=updating) the process instance in the database. For more information, see *Chapter 6, Persistence*.

9.4. Variables scopes

Each path of execution (read: token) has its own set of process variables. Requesting a variable is always done on a token. Process instances have a tree of tokens (see *Chapter 3, Graph Oriented Programming*). When requesting a variable without specifying a token, the default token is the root token.

The variable lookup is done recursively over the parents of the given token. The behavior is similar to the scoping of variables in programming languages.

When a non-existing variable is set on a token, the variable is created on the root-token. This means that each variable has by default process scope. To make a variable token-local, you have to create it explicitly with:

```
ContextInstance.createVariable(String name, Object value, Token token);
```

9.4.1. Variables overloading

Variable overloading means that each path of execution can have its own copy of a variable with the same name. They are treated as independent and hence can be of different types. Variable overloading can be interesting if you launch multiple concurrent paths of execution over the same transition. Then the only thing that distinguishes those paths of executions are their respective set of variables.

9.4.2. Variables overriding

Variable overriding means that variables of nested paths of execution override variables in more global paths of execution. Generally, nested paths of execution relate to concurrency : the paths of execution between a fork and a join are children (nested) of the path of execution that arrived in the fork. For example, if you have a variable 'contact' in the process instance scope, you can override this variable in the nested paths of execution 'shipping' and 'billing'.

9.4.3. Task instance variable scope

For more info on task instance variables, see *Section 10.4, "Task instance variables"*.

9.5. Transient variables

When a process instance is persisted in the database, normal variables are also persisted as part of the process instance. In some situations you might want to use a variable in a delegation class, but you don't want to store it in the database. An example could be a database connection that you want to pass from outside of jBPM to a delegation class. This can be done with transient variables.

The lifetime of transient variables is the same as the ProcessInstance java object.

Because of their nature, transient variables are not related to a token. So there is only one map of transient variables for a process instance object.

The transient variables are accessible with their own set of methods in the context instance, and don't need to be declared in the processdefinition.xml

```
Object ContextInstance.getTransientVariable(String name);
void ContextInstance.setTransientVariable(String name, Object value);
```

9.6. Customizing variable persistence

Variables are stored in the database in a 2-step approach :

```
user-java-object <---> converter <---> variable instance
```

Variables are stored in **`VariableInstance`**s. The members of **`VariableInstance`**s are mapped to fields in the database with hibernate. In the default configuration of jBPM, 6 types of VariableInstances are used:

- **`DateInstance`** (with one java.lang.Date field that is mapped to a **`Types.TIMESTAMP`** in the database)
- **`DoubleInstance`** (with one java.lang.Double field that is mapped to a **`Types.DOUBLE`** in the database)
- **`StringInstance`** (with one java.lang.String field that is mapped to a **`Types.VARCHAR`** in the database)
- **`LongInstance`** (with one java.lang.Long field that is mapped to a **`Types.BIGINT`** in the database)
- **`HibernateLongInstance`** (this is used for hibernatable types with a long id field. One java.lang.Object field is mapped as a reference to a hibernate entity in the database)

- **HibernateStringInstance** (this is used for hibernatable types with a string id field. One java.lang.Object field is mapped as a reference to a hibernate entity in the database)

Converters convert between java-user-objects and the java objects that can be stored by the **VariableInstance**s. So when a process variable is set with e.g. **ContextInstance.setVariable(String variableName, Object value)**, the value will optionally be converted with a converter. Then the converted object will be stored in a **VariableInstance**. **Converter**s are implementations of the following interface:

```
public interface Converter extends Serializable {
  boolean supports(Object value);
  Object convert(Object o);
  Object revert(Object o);
}
```

Converters are optional. Converters must be available to the jBPM class loader. See *Section 17.2.1, "The jBPM class loader"*

The way that user-java-objects are converted and stored in variable instances is configured in the file **org/jbpm/context/exe/jbpm.varmapping.properties**. To customize this property file, put a modified version in the root of the classpath, as explained in *Section 5.3, "Other configuration files"* Each line of the properties file specifies 2 or 3 class-names separated by spaces : the class name of the user-java-object, optionally the class name of the converter and the class name of the variable instance. When you refer your custom converters, make sure they are in the jBPM class path (see *Section 17.2.1, "The jBPM class loader"*). When you refer to your custom variable instances, they also have to be in the the jBPM class path and the hibernate mapping file for **org/jbpm/context/exe/VariableInstance.hbm.xml** has to be updated to include the custom subclass of VariableInstance.

For example, take a look at the following xml snippet in the file **org/jbpm/context/exe/jbpm.varmapping.xml**.

```
    <jbpm-type>
      <matcher>
        <bean class="org.jbpm.context.exe.matcher.ClassNameMatcher">
          <field name="className"><string value="java.lang.Boolean" /></
field>
        </bean>
      </matcher>

  <converter class="org.jbpm.context.exe.converter.BooleanToStringConverter"
 />
      <variable-
instance class="org.jbpm.context.exe.variableinstance.StringInstance" />
    </jbpm-type>
```

This snippet specifies that all objects of type **java.lang.Boolean** have to be converted with the converter **BooleanToStringConverter** and that the resulting object (a String) will be stored in a variable instance object of type StringInstance.

If no converter is specified as in

```
    <jbpm-type>
      <matcher>
        <bean class="org.jbpm.context.exe.matcher.ClassNameMatcher">
          <field name="className"><string value="java.lang.Long" /></
field>
        </bean>
      </matcher>
      <variable-
instance class="org.jbpm.context.exe.variableinstance.LongInstance" />
    </jbpm-type>
```

that means that the Long objects that are put in the variables are just stored in a variable instance of type LongInstance without being converted.

Task management

The core business of jBPM is the ability to persist the execution of a process. A situation in which this feature is extremely useful is the management of tasks and task-lists for people. jBPM allows to specify a piece of software describing an overall process which can have wait states for human tasks.

10.1. Tasks

Tasks are part of the process definition and they define how task instances must be created and assigned during process executions.

Tasks can be defined in **`task-node`**s and in the **`process-definition`**. The most common way is to define one or more **`task`**s in a **`task-node`**. In that case the **`task-node`** represents a task to be done by the user and the process execution should wait until the actor completes the task. When the actor completes the task, process execution should continue. When more tasks are specified in a **`task-node`**, the default behavior is to wait for all the tasks to complete.

Tasks can also be specified on the **`process-definition`**. Tasks specified on the process definition can be looked up by name and referenced from within **`task-node`**s or used from inside actions. In fact, all tasks (also in task-nodes) that are given a name can be looked up by name in the process-definition.

Task names must be unique in the whole process definition. Tasks can be given a **`priority`**. This priority will be used as the initial priority for each task instance that is created for this task. TaskInstances can change this initial priority afterward.

10.2. Task instances

A task instance can be assigned to an actorId (java.lang.String). All task instances are stored in one table of the database (JBPM_TASKINSTANCE). By querying this table for all task instances for a given actorId, you get the task list for that particular user.

The jBPM task list mechanism can combine jBPM tasks with other tasks, even when those tasks are unrelated to a process execution. That way jBPM developers can easily combine jBPM-process-tasks with tasks of other applications in one centralized task-list-repository.

10.2.1. Task instance life-cycle

The task instance life-cycle is straightforward: After creation, task instances can optionally be started. Then, task instances can be ended, which means that the task instance is marked as completed.

Note that for flexibility, assignment is not part of the life cycle. So task instances can be assigned or not assigned. Task instance assignment does not have an influence on the task instance life cycle.

Task instances are typically created by the process execution entering a **`task-node`** (with the method **`TaskMgmtInstance.createTaskInstance(...)`**).
Then, a user interface component will query the database for the task lists using the **`TaskMgmtSession.findTaskInstancesByActorId(...)`**. Then, after collecting input from the user, the UI component calls **`TaskInstance.assign(String)`**, **`TaskInstance.start()`** or **`TaskInstance.end(...)`**.

A task instance maintains its state by means of date-properties : **`create`**, **`start`** and **`end`**. Those properties can be accessed by their respective getters on the **`TaskInstance`**.

Currently, completed task instances are marked with an end date so that they are not fetched with subsequent queries for tasks lists. But they remain in the JBPM_TASKINSTANCE table.

10.2.2. Task instances and graph execution

Task instances are the items in an actor's task list. Task instances can be signalling. A signalling task instance is a task instance that, when completed, can send a signal to its token to continue the process execution. Task instances can be blocking, meaning that the related token (=path of execution) is not allowed to leave the task-node before the task instance is completed. By default task instances are signalling and non-blocking.

In case more than one task instance are associated with a task-node, the process developer can specify how completion of the task instances affects continuation of the process. Following is the list of values that can be given to the signal-property of a task-node.

- **last**: This is the default. Proceeds execution when the last task instance is completed. When no tasks are created on entrance of this node, execution is continued.
- **last-wait**: Proceeds execution when the last task instance is completed. When no tasks are created on entrance of this node, execution waits in the task node until tasks are created.
- **first**: Proceeds execution when the first task instance is completed. When no tasks are created on entrance of this node, execution is continued.
- **first-wait**: Proceeds execution when the first task instance is completed. When no tasks are created on entrance of this node, execution waits in the task node until tasks are created.
- **unsynchronized**: Execution always continues, regardless whether tasks are created or still unfinished.
- **never**: Execution never continues, regardless whether tasks are created or still unfinished.

Task instance creation might be based upon a runtime calculation. In that case, add an **`ActionHandler`** on the **`node-enter`** event of the **`task-node`** and set the attribute **`create-tasks="false"`**. Here is an example of such an action handler implementation:

```
public class CreateTasks implements ActionHandler {
  public void execute(ExecutionContext executionContext) throws Exception
 {
    Token token = executionContext.getToken();
    TaskMgmtInstance tmi = executionContext.getTaskMgmtInstance();

    TaskNode taskNode = (TaskNode) executionContext.getNode();
    Task changeNappy = taskNode.getTask("change nappy");

    // now, 2 task instances are created for the same task.
    tmi.createTaskInstance(changeNappy, token);
    tmi.createTaskInstance(changeNappy, token);
  }
}
```

As shown in the example the tasks to be created can be specified in the task-node. They could also be specified in the **`process-definition`** and fetched from the **`TaskMgmtDefinition`**. **`TaskMgmtDefinition`** extends the ProcessDefinition with task management information.

The API method for marking task instances as completed is **`TaskInstance.end()`**. Optionally, you can specify a transition in the end method. In case the completion of this task instance triggers continuation of the execution, the task-node is left over the specified transition.

10.3. Assignment

A process definition contains task nodes. A **`task-node`** contains zero or more tasks. Tasks are a static description as part of the process definition. At runtime, tasks result in the creation of task instances. A task instance corresponds to one entry in a person's task list.

With jBPM, push (personal task list)[1] and pull (group task list)[2] model (see below) of task assignment can be applied in combination. The process can calculate the responsible for a task and push it in his/her task list. Or alternatively, a task can be assigned to a pool of actors, in which case each of the actors in the pool can pull the task and put it in the actor's personal task list.

10.3.1. Assignment interfaces

Assigning task instances is done via the interface **`AssignmentHandler`**:

```
public interface AssignmentHandler extends Serializable {
  void assign( Assignable assignable, ExecutionContext
 executionContext );
}
```

An assignment handler implementation is called when a task instance is created. At that time, the task instance can be assigned to one or more actors. The **`AssignmentHandler`** implementation should call the **`Assignable`** methods (**`setActorId`** or **`setPooledActors`**) to assign a task. The **`Assignable`** is either a **`TaskInstance`** or a **`SwimlaneInstance`** (=process role).

```
public interface Assignable {
  public void setActorId(String actorId);
  public void setPooledActors(String[] pooledActors);
}
```

Both **`TaskInstance`**s and **`SwimlaneInstance`**s can be assigned to a specific user or to a pool of actors. To assign a TaskInstance to a user, call **`Assignable.setActorId(String actorId)`**. To assign a TaskInstance to a pool of candidate actors, call **`Assignable.setPooledActors(String[] actorIds)`**.

Each task in the process definition can be associated with an assignment handler implementation to perform the assignment at runtime.

When more than one task in a process should be assigned to the same person or group of actors, consider the usage of a swimlane[3]

[1] *Section 10.3.3, “The personal task list”*.
[2] *Section 10.3.4, “The group task list”*.
[3] *Section 10.6, “Swimlanes”*.

To allow for the creation of reusable **`AssignmentHandler`**s, each usage of an **`AssignmentHandler`** can be configured in the **`processdefinition.xml`**. See *Section 17.2, "Delegation"* for more information on how to add configuration to assignment handlers.

10.3.2. The assignment data model

The data model for managing assignments of task instances and swimlane instances to actors is the following. Each **`TaskInstance`** has an actorId and a set of pooled actors.

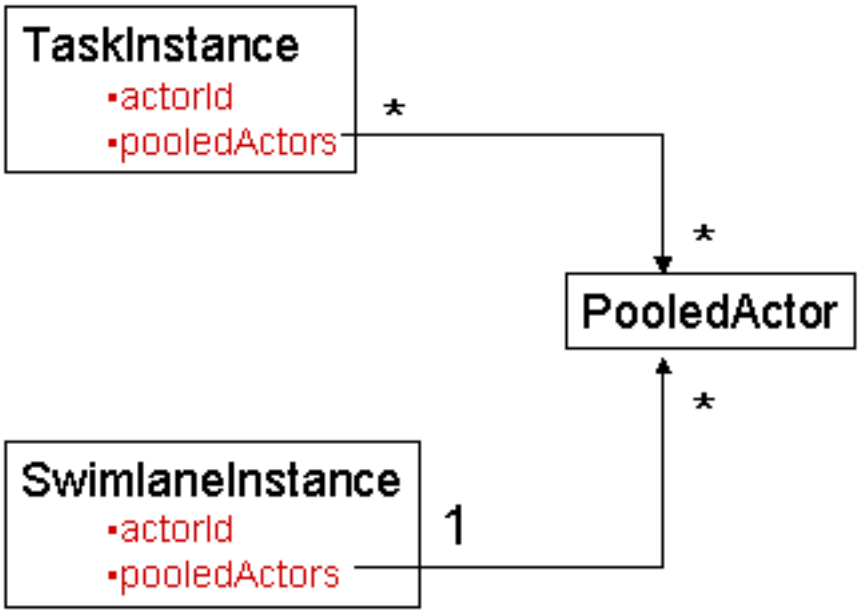

Figure 10.1. The assignment model class diagram

The actorId is the responsible for the task, while the set of pooled actors represents a collection of candidates that can become responsible if they would take the task. Both actorId and pooledActors are optional and can also be combined.

10.3.3. The personal task list

The personal task list denotes all the task instances that are assigned to a specific individual. This is indicated with the property **`actorId`** on a **`TaskInstance`**. So to put a TaskInstance in someone's personal task list, you just use one of the following ways:

- Specify an expression in the attribute **`actor-id`** of the task element in the process
- Use TaskInstance.setActorId(String) from anywhere in your code
- Use assignable.setActorId(String) in an AssignmentHandler

To fetch the personal task list for a given user, use **`TaskMgmtSession.findTaskInstances(String actorId)`**.

10.3.4. The group task list

The pooled actors denote the candidates for the task instance. This means that the task is offered to many users and one candidate has to step up and take the task. Users can not start working on

tasks in their group task list immediately. That would result in a potential conflict that many people start working on the same task. To prevent this, users can only take task instances of their group task list and move them into their personal task list. Users are only allowed to start working on tasks that are in their personal task list.

To put a taskInstance in someone's group task list, you must put the user's actorId or one of the user's groupIds in the pooledActorIds. To specify the pooled actors, use one of the following:

- Specify an expression in the attribute **`pooled-actor-ids`** of the task element in the process
- Use TaskInstance.setPooledActorIds(String[]) from anywhere in your code
- Use assignable.setPooledActorIds(String[]) in an AssignmentHandler

To fetch the group task list for a given user, proceed as follows: Make a collection that includes the user's actorId and all the ids of groups that the user belongs to. With **`TaskMgmtSession.findPooledTaskInstances(String actorId)`** or **`TaskMgmtSession.findPooledTaskInstances(List actorIds)`** you can search for task instances that are not in a personal task list (actorId==null) and for which there is a match in the pooled actorIds.

The motivation behind this is that we want to separate the identity component from jBPM task assignment. jBPM only stores Strings as actorIds and doesn't know the relation between the users, groups and other identity information.

The actorId will always override the pooled actors. So a taskInstance that has an actorId and a list of pooledActorIds, will only show up in the actor's personal task list. Keeping the pooledActorIds around allows a user to put a task instance back into the group by just erasing the actorId property of the taskInstance.

10.4. Task instance variables

A task instance can have its own set of variables and a task instance can also 'see' the process variables. Task instances are usually created in an execution path (=token). This creates a parent-child relation between the token and the task instance similar to the parent-child relation between the tokens themselves. The normal scoping rules apply between the variables of a task instance and the process variables of the related token. More info about scoping can be found in *Section 9.4, “Variables scopes”*.

This means that a task instance can 'see' its own variables plus all the variables of its related token.

The controller can be used to create, populate and submit variables between the task instance scope and the process scoped variables.

10.5. Task controllers

At creation of a task instance, the task controllers can populate the task instance variables and when the task instance is finished, the task controller can submit the data of the task instance into the process variables.

Note that you are not forced to use task controllers. Task instances also are able to 'see' the process variables related to its token. Use task controllers when you want to:

- a) create copies of variables in the task instances so that intermediate updates to the task instance variables don't affect the process variables until the process is finished and the copies are submitted back into the process variables.
- b) the task instance variables do not relate one-on-one with the process variables. E.g. suppose the process has variables 'sales in January' 'sales in February' and 'sales in march'. Then the form for the task instance might need to show the average sales in the 3 months.

Tasks are intended to collect input from users. But there are many user interfaces which could be used to present the tasks to the users. E.g. a web application, a swing application, an instant messenger, an email form,... So the task controllers make the bridge between the process variables (=process context) and the user interface application. The task controllers provide a view of process variables to the user interface application.

The task controller makes the translation (if any) from the process variables to the task variables. When a task instance is created, the task controller is responsible for extracting information from the process variables and creating the task variables. The task variables serve as the input for the user interface form. And the user input can be stored in the task variables. When the user ends the task, the task controller is responsible for updating the process variables based on the task instance data.

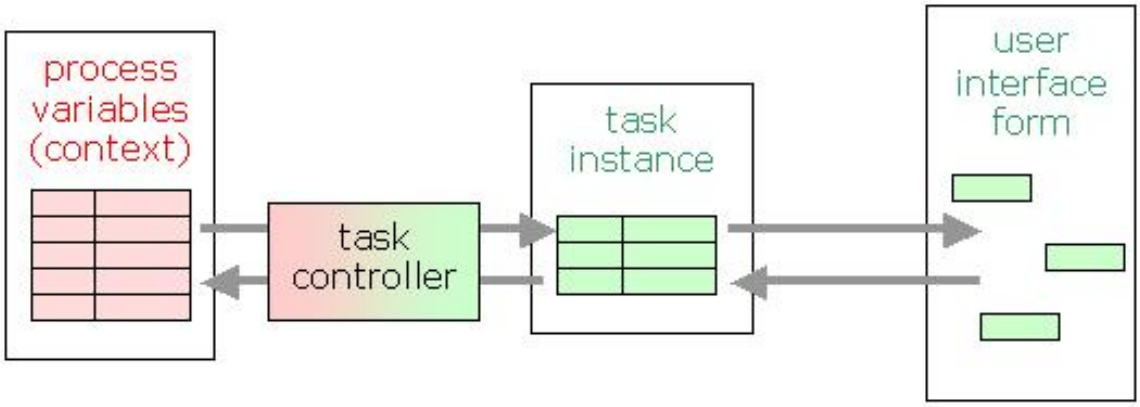

Figure 10.2. The task controllers

In a simple scenario, there is a one-on-one mapping between process variables and the form parameters. Task controllers are specified in a task element. In this case, the default jBPM task controller can be used and it takes a list of **`variable`** elements inside. The variable elements express how the process variables are copied in the task variables.

The next example shows how you can create separate task instance variable copies based on the process variables:

```
<task name="clean ceiling">
  <controller>
    <variable name="a" access="read" mapped-name="x" />
    <variable name="b" access="read,write,required" mapped-name="y" />
    <variable name="c" access="read,write" />
  </controller>
</task>
```

The **`name`** attribute refers to the name of the process variable. The **`mapped-name`** is optional and refers to the name of the task instance variable. If the mapped-name attribute is omitted, mapped-

name defaults to the name. Note that the mapped-name also is used as the label for the fields in the task instance form of the web application.

The **access** attribute specifies if the variable is copied at task instance creation, will be written back to the process variables at task end and whether it is required. This information can be used by the user interface to generate the proper form controls. The access attribute is optional and the default access is 'read,write'.

A **task-node** can have many tasks and a **start-state** can have 1 task.

If the simple one-to-one mapping between process variables and form parameters is too limiting, you can also write your own TaskControllerHandler implementation. Here's the TaskControllerHandler interface:

```
public interface TaskControllerHandler extends Serializable {
  void initializeTaskVariables(TaskInstance taskInstance, ContextInstance
 contextInstance, Token token);
  void submitTaskVariables(TaskInstance taskInstance, ContextInstance
 contextInstance, Token token);
}
```

And here's how to configure your custom task controller implementation in a task:

```
<task name="clean ceiling">
  <controller class="com.yourcom.CleanCeilingTaskControllerHandler">
    -- here goes your task controller handler configuration --
  </controller>
</task>
```

10.6. Swimlanes

A swimlane is a process role. It is a mechanism to specify that multiple tasks in the process should be done by the same actor. So after the first task instance is created for a given swimlane, the actor should be remembered in the process for all subsequent tasks that are in the same swimlane. A swimlane therefore has one **assignment**[4] and all tasks that reference a swimlane should not specify an **assignment**[5].

When the first task in a given swimlane is created, the **AssignmentHandler** of the swimlane is called. The **Assignable** that is passed to the **AssignmentHandler** will be the **SwimlaneInstance**. Important to know is that all assignments that are done on the task instances in a given swimlane will propagate to the swimlane instance. This behavior is implemented as the default because the person that takes a task to fulfilling a certain process role will have the knowledge of that particular process. So all subsequent assignments of task instances to that swimlane are done automatically to that user.

Swimlane is a terminology borrowed from UML activity diagrams.

10.7. Swimlane in start task

A swimlane can be associated with the start task to capture the process initiator.

[4] *Section 10.3, "Assignment"*.
[5] *Section 10.3, "Assignment"*.

A task can be specified in a start-state. That task be associated with a swimlane. When a new task instance is created for such a task, the current authenticated actor will be captured with Authentication.getAuthenticatedActorId()[6] and that actor will be stored in the swimlane of the start task.

For example:

```
<process-definition>
  <swimlane name='initiator' />
  <start-state>
    <task swimlane='initiator' />
    <transition to='...' />
  </start-state>
</process-definition>
```

Also variables can be added to the start task as with any other task to define the form associated with the task. See *Section 10.5, "Task controllers"*

10.8. Task events

Tasks can have actions associated with them. There are 4 standard event types defined for tasks: **`task-create`**, **`task-assign`**, **`task-start`** and **`task-end`**.

`task-create` is fired when a task instance is created.

`task-assign` is fired when a task instance is being assigned. Note that in actions that are executed on this event, you can access the previous actor with **`executionContext.getTaskInstance().getPreviousActorId();`**

`task-start` is fired when **`TaskInstance.start()`** is called. This can be used to indicate that the user is actually starting to work on this task instance. Starting a task is optional.

`task-end` is fired when **`TaskInstance.end(...)`** is called. This marks the completion of the task. If the task is related to a process execution, this call might trigger the resuming of the process execution.

Since tasks can have events and actions associated with them, also exception handlers can be specified on a task. For more information about exception handling, see *Section 8.7, "Exception handling"*.

10.9. Task timers

As on nodes, timers can be specified on tasks. See *Section 12.1, "Timers"*.

The special thing about timers for tasks is that the **`cancel-event`** for task timers can be customized. By default, a timer on a task will be canceled when the task is ended (=completed). But with the **`cancel-event`** attribute on the timer, process developers can customize that to e.g. **`task-assign`** or **`task-start`**. The **`cancel-event`** supports multiple events. The **`cancel-event`** types can be combined by specifying them in a comma separated list in the attribute.

[6] *Section 18.2, "Authentication"*.

10.10. Customizing task instances

Task instances can be customized. The easiest way to do this is to create a subclass of **`TaskInstance`**. Then create a **`org.jbpm.taskmgmt.TaskInstanceFactory`** implementation and configure it by setting the configuration property **`jbpm.task.instance.factory`** to the fully qualified class name in the jbpm.cfg.xml. If you use a subclass of TaskInstance, also create a hibernate mapping file for the subclass (using the hibernate **`extends="org.jbpm.taskmgmt.exe.TaskInstance"`**). Then add that mapping file to the list of mapping files in the **`hibernate.cfg.xml`**

10.11. The identity component

Management of users, groups and permissions is commonly known as identity management. jBPM includes an optional identity component that can be easily replaced by a company's own identity data store.

The jBPM identity management component includes knowledge of the organizational model. Task assignment is typically done with organizational knowledge. So this implies knowledge of an organizational model, describing the users, groups, systems and the relations between them. Optionally, permissions and roles can be included too in an organizational model. Various academic research attempts failed, proving that no generic organizational model can be created that fits every organization.

The way jBPM handles this is by defining an actor as an actual participant in a process. An actor is identified by its ID called an actorId. jBPM has only knowledge about actorIds and they are represented as **`java.lang.String`**s for maximum flexibility. So any knowledge about the organizational model and the structure of that data is outside the scope of the jBPM core engine.

As an extension to jBPM we will provide (in the future) a component to manage that simple user-roles model. This many to many relation between users and roles is the same model as is defined in the J2EE and the servlet specs and it could serve as a starting point in new developments. People interested in contributing should check the jboss jbpm jira issue tracker for more details.

Note that the user-roles model as it is used in the servlet, ejb and portlet specifications, is not sufficiently powerful for handling task assignments. That model is a many-to-many relation between users and roles. This doesn't include information about the teams and the organizational structure of users involved in a process.

10.11.1. The identity model

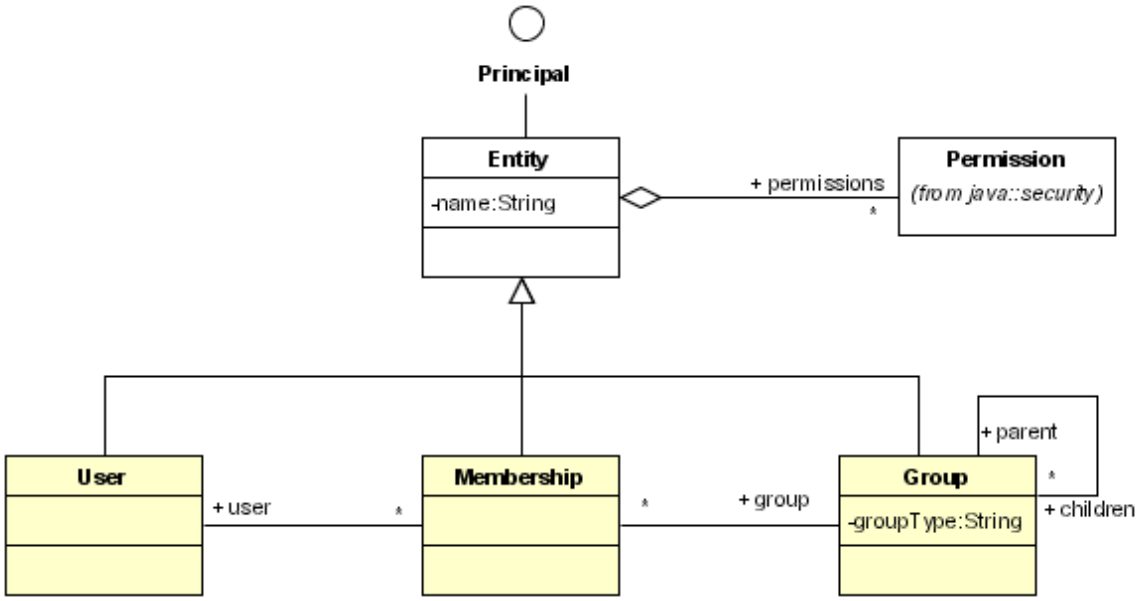

Figure 10.3. The identity model class diagram

The classes in yellow are the relevant classes for the expression assignment handler that is discussed next.

A **`User`** represents a user or a service. A **`Group`** is any kind of group of users. Groups can be nested to model the relation between a team, a business unit and the whole company. Groups have a type to differentiate between the hierarchical groups and e.g. hair color groups. **`Membership`**s represent the many-to-many relation between users and groups. A membership can be used to represent a position in a company. The name of the membership can be used to indicate the role that the user fulfills in the group.

10.11.2. Assignment expressions

The identity component comes with one implementation that evaluates an expression for the calculation of actors during assignment of tasks. Here's an example of using the assignment expression in a process definition:

```
<process-definition>
  <task-node name='a'>
    <task name='laundry'>
      <assignment expression='previous --> group(hierarchy) -->
 member(boss)' />
    </task>
    <transition to='b' />
  </task-node>
```

Syntax of the assignment expression is like this:

```
first-term --> next-term --> next-term --> ... --> next-term
```

```
where

first-term ::= previous |
               swimlane(swimlane-name) |
               variable(variable-name) |
               user(user-name) |
               group(group-name)

and

next-term ::= group(group-type) |
              member(role-name)
```

10.11.2.1. First terms

An expression is resolved from left to right. The first-term specifies a **`User`** or **`Group`** in the identity model. Subsequent terms calculate the next term from the intermediate user or group.

`previous` means the task is assigned to the current authenticated actor. This means the actor that performed the previous step in the process.

`swimlane(swimlane-name)` means the user or group is taken from the specified swimlane instance.

`variable(variable-name)` means the user or group is taken from the specified variable instance. The variable instance can contain a **`java.lang.String`**, in which case that user or group is fetched from the identity component. Or the variable instance contains a **`User`** or **`Group`** object.

`user(user-name)` means the given user is taken from the identity component.

`group(group-name)` means the given group is taken from the identity component.

10.11.2.2. Next terms

`group(group-type)` gets the group for a user. Meaning that previous terms must have resulted in a **`User`**. It searches for the the group with the given group-type in all the memberships for the user.

`member(role-name)` gets the user that performs a given role for a group. The previous terms must have resulted in a **`Group`**. This term searches for the user with a membership to the group for which the name of the membership matches the given role-name.

10.11.3. Removing the identity component

When you want to use your own datasource for organizational information such as your company's user database or LDAP system, you can just rip out the jBPM identity component. The only thing you need to do is make sure that you delete the line ...

```
<mapping resource="org/jbpm/identity/User.hbm.xml"/>
<mapping resource="org/jbpm/identity/Group.hbm.xml"/>
<mapping resource="org/jbpm/identity/Membership.hbm.xml"/>
```

from the **`hibernate.cfg.xml`**

The **ExpressionAssignmentHandler** is dependent on the identity component so you will not be able to use it as is. In case you want to reuse the **ExpressionAssignmentHandler** and bind it to your user data store, you can extend from the **ExpressionAssignmentHandler** and override the method **getExpressionSession**.

```
protected ExpressionSession getExpressionSession(AssignmentContext
 assignmentContext);
```

Document management

This is still an experimental feature.

To enable this feature, you need to un-comment the following line in the hibernate.cfg.xml:

```
<mapping resource="org/jbpm/context/exe/variableinstance/
JcrNodeInstance.hbm.xml"/>
```

The document management support of jBPM is based on *Java Content Repository*[1]. That is a standard java specification for integrating document management systems into Java. The basic idea is that jBPM supports storage of JCR nodes as process variables.

To store a node, the session, repository and path are extracted from the node like this:

```
Session session = node.getSession();
Repository repo = session.getRepository();
Workspace wspace = session.getWorkspace();

// THE NODE REPOSITORY AND WORKSPACE NAME GOT TO CORRESPOND WITH A JBPM
 SERVICE NAME
repository = repo.getDescriptor(Repository.REP_NAME_DESC);
workspace = wspace.getName();
path = node.getPath();
```

IMPORTANT NOTE: The name of the jbpm context service MUST correspond with the name of the repository (repository.getDescriptor(Repository.REP_NAME_DESC)). This is to make the match between the reference stored in the jbpm process variables and the repository when a node-variable is being loaded from the jBPM DB. When the JCR node process variable is retrieved, each service name in the jbpm context will be matched against the repository and workspace name stored. The matching between jbpm context service and the JCR session/repository names will go like this:

- if there is a jbpm context service named 'jcr' (lower case) that one will be taken
- a service name that is equal to the repository name matches
- a service that starts with the repository name and ends with the workspace name matches and takes preference over a service with the repository name

The typical use case for this feature is a document approval process. A document needs to be approved and updated. That document (e.g. a word document), can be stored in a JCR-content-repository-node. The node contains all the versions of the document. So that later in the process, people still can consult the historical versions of the document.

This feature was only tested with Jackrabbit. Please refer to the JCR implementation documentation for more information about library dependencies.

[1] http://www.jcp.org/en/jsr/detail?id=170

Scheduler

This chapter describes how to work with timers in jBPM.

Upon events in the process, timers can be created. When a timer expires, an action can be executed or a transition can be taken.

12.1. Timers

The easiest way to specify a timer is by adding a timer element to the node.

```
<state name='catch crooks'>
  <timer name='reminder'
         duedate='3 business hours'
         repeat='10 business minutes'
         transition='time-out-transition' >
    <action class='the-remainder-action-class-name' />
  </timer>
</state>
```

A timer that is specified on a node, is not executed after the node is left. Both the transition and the action are optional. When a timer is executed, the following events occur in sequence :

- an event is fired of type **`timer`**
- if an action is specified, the action is executed.
- if a transition is specified, a signal will be sent to resume execution over the given transition.

Every timer must have a unique name. If no name is specified in the **`timer`** element, the name of the node is taken as the name of the timer.

The timer action can be any supported action element like e.g. **`action`** or **`script`**.

Timers are created and canceled by actions. The 2 action-elements are **`create-timer`** and **`cancel-timer`**. Actually, the timer element shown above is just a short notation for a create-timer action on **`node-enter`** and a cancel-timer action on **`node-leave`**.

12.2. Scheduler deployment

Process executions create and cancel timers. The timers are stored in a timer store. A separate timer runner must check the timer store and execute the timers when they are due.

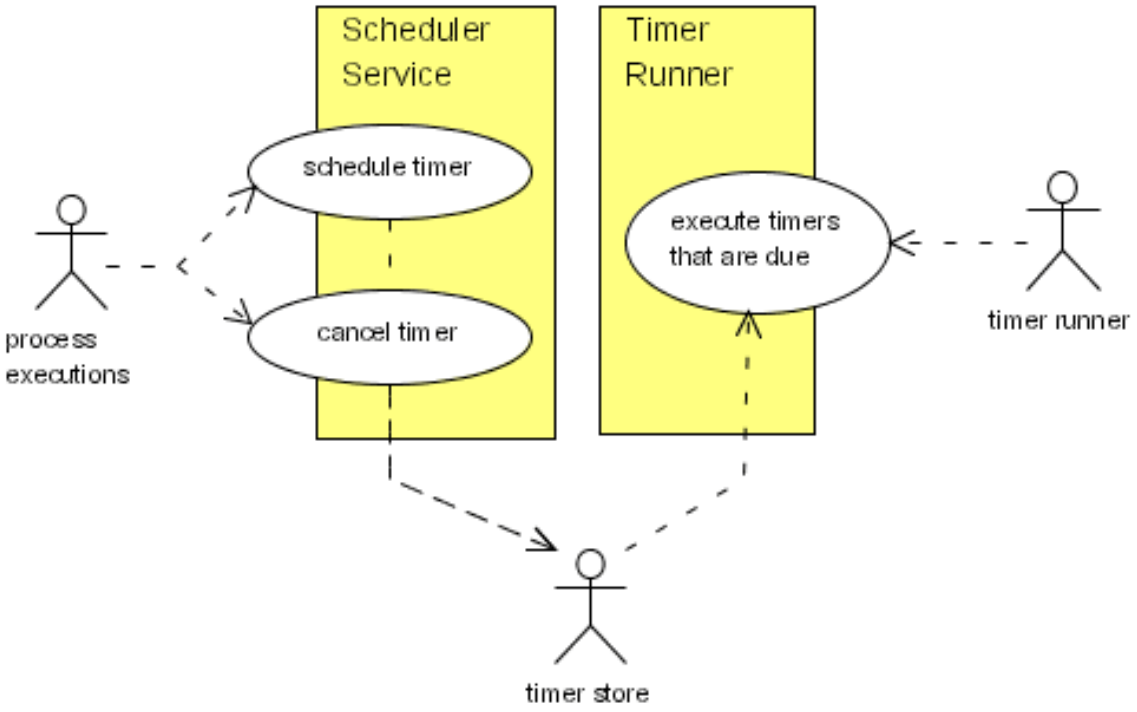

Figure 12.1. Scheduler components overview

The following class diagram shows the classes that are involved in the scheduler deployment. The interfaces **`SchedulerService`** and **`TimerExecutor`** are specified to make the timer execution mechanism pluggable.

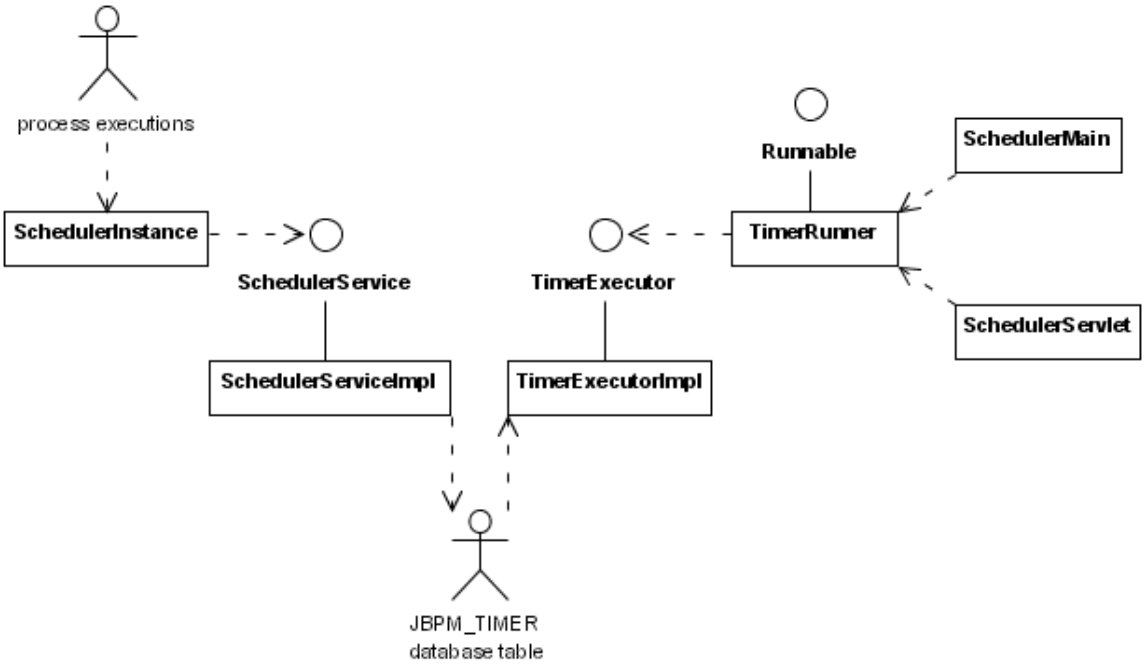

Figure 12.2. Scheduler classes overview

Asynchronous continuations

13.1. The concept

jBPM is based on Graph Oriented Programming (GOP). Basically, GOP specifies a simple state machine that can handle concurrent paths of execution. But in the execution algorithm specified in GOP, all state transitions are done in a single operation in the thread of the client. If you're not familiar with the execution algorithm defined in *Chapter 3, Graph Oriented Programming*, please read that first. By default, this performing state transitions in the thread of the client is a good approach cause it fits naturally with server side transactions. The process execution moves from one wait state to another wait state in one transaction.

But in some situations, a developer might want to fine-tune the transaction demarcation in the process definition. In jPDL, it is possible to specify that the process execution should continue asynchronously with the attribute **`async="true"`**. **`async="true"`** can be specified on all node types and all action types.

13.2. An example

Normally, a node is always executed after a token has entered the node. So the node is executed in the thread of the client. We'll explore asynchronous continuations by looking two examples. The first example is a part of a process with 3 nodes. Node 'a' is a wait state, node 'b' is an automated step and node 'c' is again a wait state. This process does not contain any asynchronous behavior and it is represented in the picture below.

The first frame, shows the starting situation. The token points to node 'a', meaning that the path of execution is waiting for an external trigger. That trigger must be given by sending a signal to the token. When the signal arrives, the token will be passed from node 'a' over the transition to node 'b'. After the token arrived in node 'b', node 'b' is executed. Recall that node 'b' is an automated step that does not behave as a wait state (e.g. sending an email). So the second frame is a snapshot taken when node 'b' is being executed. Since node 'b' is an automated step in the process, the execute of node 'b' will include the propagation of the token over the transition to node 'c'. Node 'c' is a wait state so the third frame shows the final situation after the signal method returns.

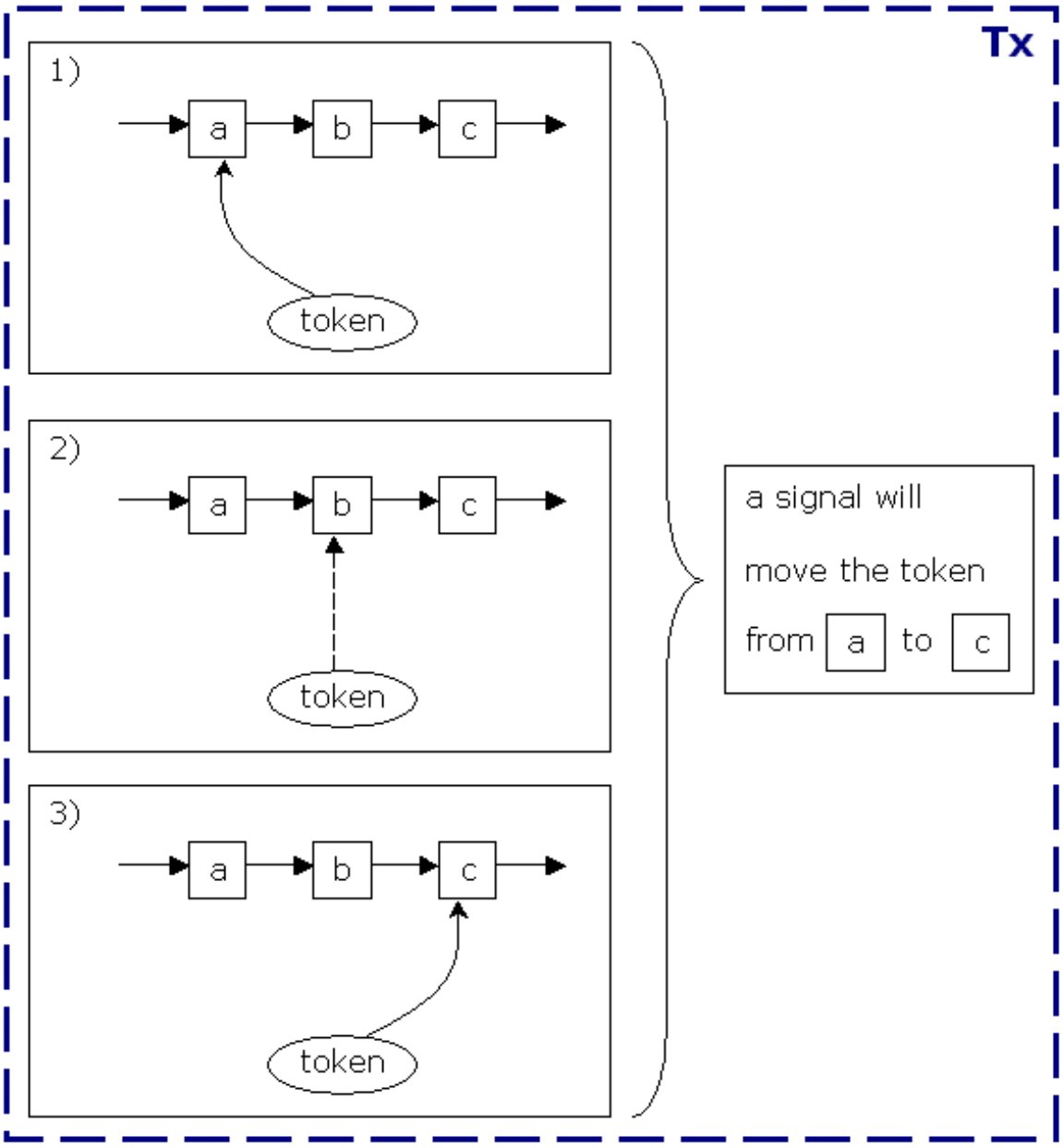

Figure 13.1. Example 1: Process without asynchronous continuation

While persistence is not mandatory in jBPM, the most common scenario is that a signal is called within a transaction. Let's have a look at the updates of that transaction. First of all, the token is updated to point to node 'c'. These updates are generated by hibernate as a result of the **`GraphSession.saveProcessInstance`** on a JDBC connection. Second, in case the automated action would access and update some transactional resources, those transactional updates should be combined or part of the same transaction.

Now, we are going to look at the second example, the second example is a variant of the first example and introduces an asynchronous continuation in node 'b'. Nodes 'a' and 'c' behave the same as in the first example, namely they behave as wait states. In jPDL, a node is marked as asynchronous by setting the attribute **`async="true"`**.

The result of adding **`async="true"`** to node 'b' is that the process execution will be split up into 2 parts. The first part will execute the process up to the point where node 'b' is to be executed. The second part will execute node 'b' and that execution will stop in wait state 'c'.

The transaction will hence be split up into 2 separate transactions. One transaction for each part. While it requires an external trigger (the invocation of the **`Token.signal`** method) to leave node 'a' in the first transaction, jBPM will automatically trigger and perform the second transaction.

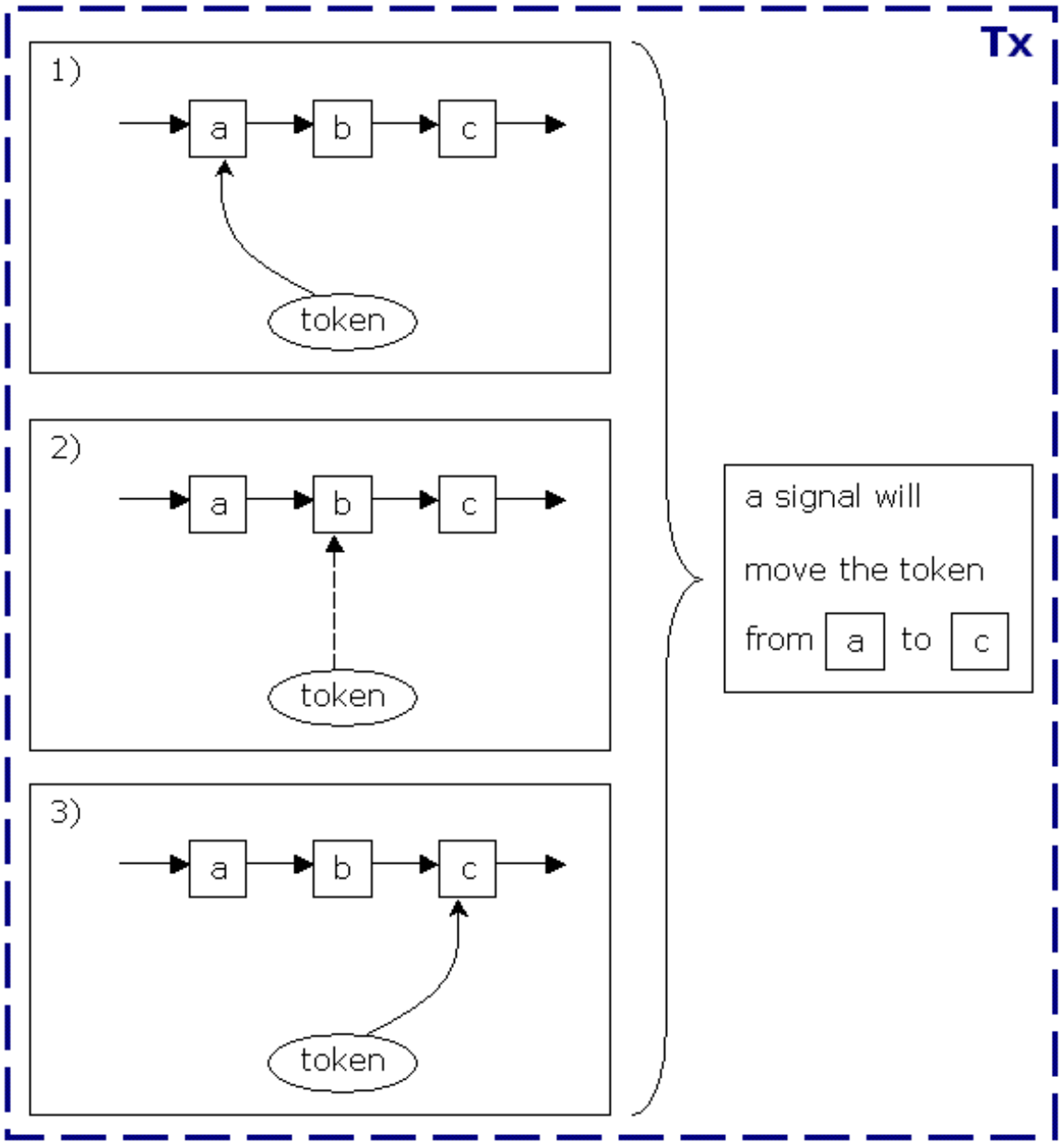

Figure 13.2. Example 2: A process with asynchronous continuations

For actions, the principle is similar. Actions that are marked with the attribute **`async="true"`** are executed outside of the thread that executes the process. If persistence is configured (it is by default), the actions will be executed in a separate transaction.

In jBPM, asynchronous continuations are realized by using an asynchronous messaging system. When the process execution arrives at a point that should be executed asynchronously, jBPM will

suspend the execution, produces a command message and send it to the command executor. The command executor is a separate component that, upon receipt of a message, will resume the execution of the process where it got suspended.

jBPM can be configured to use a JMS provider or its built-in asynchronous messaging system. The built-in messaging system is quite limited in functionality, but allows this feature to be supported on environments where JMS is unavailable.

13.3. The command executor

The command executor is the component that resumes process executions asynchronously. It waits for command messages to arrive over an asynchronous messaging system and executes them. The two commands used for asynchronous continuations are **`ExecuteNodeCommand`** and **`ExecuteActionCommand`**.

These commands are produced by the process execution. During process execution, for each node that has to be executed asynchronously, an **`ExecuteNodeCommand`** (POJO) will be created in the **`MessageInstance`**. The message instance is a non-persistent extension of the ProcessInstance and it just collects all the messages that have to be sent.

The messages will be sent as part of the **`GraphSession.saveProcessInstance`**. The implementation of that method includes a context builder that acts as an aspect on the saveProcessInstance method. The actual interceptors can be configured in the **`jbpm.cfg.xml`**. One of the interceptors, **`SendMessagesInterceptor`**, is configured by default and will read the messages from the **`MessageInstance`** and send them over the configurable asynchronous messaging system.

The **`SendMessagesInterceptor`** uses the interfaces **`MessageServiceFactory`** and **`MessageService`** to send messages. This is to make the asynchronous messaging implementation configurable (also in **`jbpm.cfg.xml`**).

Here's how the job executor works in a nutshell:

Jobs are records in the database. Jobs are commands and can be executed. Both timers and asynchronous messages are jobs. For asynchronous messages, the dueDate is simply set to now when they are inserted. The job executor must execute the jobs. This is done in 2 phases: 1) a job executor thread must acquire a job and 2) the thread that acquired the job must execute it.

Acquiring a job and executing the job are done in 2 separate transactions. A thread acquires a job by putting its name into the owner field of the job. Each thread has a unique name based on ip-address and sequence number. Hibernate's optimistic locking is enabled on **`Job`**-objects. So if 2 threads try to acquire a job concurrently, one of them will get a StaleObjectException and rollback. Only the first one will succeed. The thread that succeeds in acquiring a job is now responsible for executing it in a separate transaction.

A thread could die between acquisition and execution of a job. For clean-up of those situations, there is 1 lock-monitor thread per job executor that checks the lock times. By default, jobs that are locked for more then 30 minutes will be unlocked so that they can be executed by another job.

The required isolation level should be set to REPEATABLE_READ for Hibernate's optimistic locking to work correctly. That isolation level will guarantee that

```
update JBPM_JOB job
set job.version = 2
```

```
    job.lockOwner = '192.168.1.3:2'
where
    job.version = 1
```

will only return result 1 row updated in exactly 1 of the competing transactions.

Non-Repeatable Reads means that the following anomaly can happen: A transaction re-reads data it has previously read and finds that data has been modified by another transaction, one that has been committed since the transaction's previous read.

Non-Repeatable reads are a problem for optimistic locking and therefore, isolation level READ_COMMITTED is not enough as it allows for Non-Repeatable reads to occur. So REPEATABLE_READ is required if you configure more then 1 job executor thread.

13.4. jBPM's built-in asynchronous messaging

When using jBPM's built-in asynchronous messaging, messages will be sent by persisting them to the database. This message persisting can be done in the same transaction/jdbc connection as the jBPM process updates.

The command messages will be stored in the **`JBPM_MESSAGE`** table.

The POJO command executor (**`org.jbpm.msg.command.CommandExecutor`**) will read the messages from the database table and execute them. So the typical transaction of the POJO command executor looks like this: 1) read next command message 2) execute command message 3) delete command message.

If execution of a command message fails, the transaction will be rolled back. After that, a new transaction will be started that adds the error message to the message in the database. The command executor filters out all messages that contain an exception.

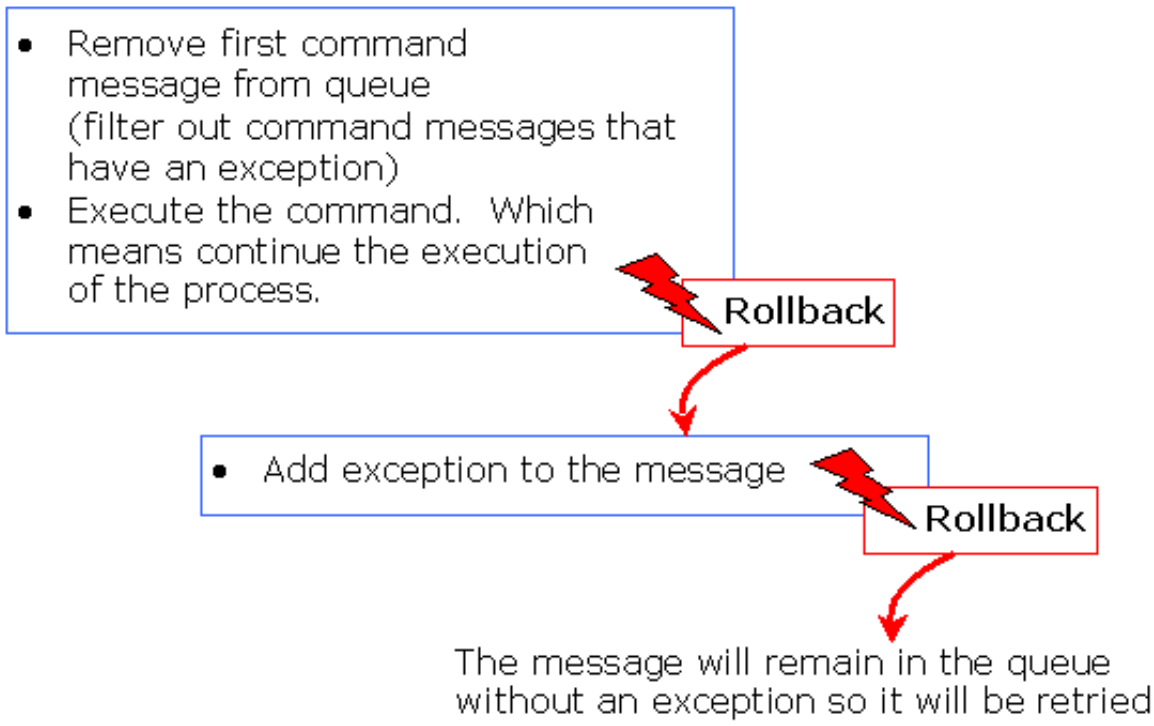

Figure 13.3. POJO command executor transactions

If for some reason or another, the transaction that adds the exception to the command message would fail, it is rolled back as well. In that case, the message remains in the queue without an exception so it will be retried later.

Limitation: beware that jBPM's built-in asynchronous messaging system does not support multi-node locking. So you cannot just deploy the POJO command executor multiple times and have them configured to use the same database.

13.5. JMS for asynchronous architectures

The asynchronous continuations feature opens up a new world of jBPM usage scenarios. Where typically, jBPM is used for modelling business processes, it can now be used from a more technical perspective.

Imagine that you have an application with quite some asynchronous processing. That typically requires quite a bit of difficult setup to bind all the message producing and message consuming pieces of software together. With jBPM it now becomes possible to create a picture of the overall asynchronous architecture, have all your code in POJO's and add transaction demarcation in the overall process file. jBPM will now take care of binding the senders to the receivers without the need for writing all the JMS or MDB code yourself.

13.6. Future directions

Future versions will add support for multiple queues. So that it becomes possible to specify a queue for each node or action that is marked as asynchronous. Also it would be great to produce message for a set of queues in a round-robin. Since all of this should be configurable for both the JMS and the built-in messaging systems, this will require some thought on how to do all this configurations. The process definitions should not have to depend on any of the 2 possible implementations.

Business calendar

This chapter describes the business calendar of jBPM. The business calendar knows about business hours and is used in calculation of due dates for tasks and timers.

The business calendar is able to calculate a due date by adding a duration to or subtracting it from a base date. If the base date is omitted, the 'current' date is used.

14.1. Duedate

As mentioned the due date is composed of a duration and a base date. If this base date is omitted, the duration is relative to the date (and time) at the moment of calculating the duedate. The format is:

`duedate ::= [<basedate> +/-] <duration>`

14.1.1. Duration

A duration is specified in absolute or in business hours. Let's look at the syntax:

`duration ::= <quantity> [business] <unit>`

Where **`<quantity>`** is a piece of text that is parsable with Double.parseDouble(quantity). **`<unit>`** is one of {second, seconds, minute, minutes, hour, hours, day, days, week, weeks, month, months, year, years}. And adding the optional indication **`business`** means that only business hours should be taken into account for this duration. Without the indication **`business`**, the duration will be interpreted as an absolute time period.

14.1.2. Base date

A duration is specified in absolute or in business hours. Let's look at the syntax:

`basedate ::= <EL> +/-`

Where **`<EL>`** is any JAVA Expression Language expression that resolves to a JAVA Date or Calendar object. Referencing variable of other object types, even a String in a date format like '2036-02-12', will throw a JbpmException

NOTE: This basedate is supported on the duedate attributes of a plain timer, on the reminder of a task and the timer within a task. It is **not** supported on the repeat attributes of these elements.

14.1.3. Examples

The following examples of the usage are all possible

```
<timer name="daysBeforeHoliday" duedate="5 business days">...</timer>

<timer name="pensionDate" duedate="#{dateOfBirth} + 65 years" >...</
timer>

<timer name="pensionReminder" duedate="#{dateOfPension} - 1 year" >...</
timer>
```

```
<timer name="fireWorks" duedate="#{chineseNewYear} repeat="1 year" >...</
timer>

<reminder name="hitBoss" duedate="#{payRaiseDay} + 3 days" repeat="1
 week" />
```

14.2. Calendar configuration

The file **`org/jbpm/calendar/jbpm.business.calendar.properties`** specifies what business hours are. The configuration file can be customized and a modified copy can be placed in the root of the classpath.

This is the example business hour specification that is shipped by default in **`jbpm.business.calendar.properties`**:

```
hour.format=HH:mm
#weekday ::= [<daypart> [& <daypart>]*]
#daypart ::= <start-hour>-<to-hour>
#start-hour and to-hour must be in the hour.format
#dayparts have to be ordered
weekday.monday=    9:00-12:00 & 12:30-17:00
weekday.tuesday=   9:00-12:00 & 12:30-17:00
weekday.wednesday= 9:00-12:00 & 12:30-17:00
weekday.thursday=  9:00-12:00 & 12:30-17:00
weekday.friday=    9:00-12:00 & 12:30-17:00
weekday.saturday=
weekday.sunday=

day.format=dd/MM/yyyy
# holiday syntax: <holiday>
# holiday period syntax: <start-day>-<end-day>
# below are the belgian official holidays
holiday.1=  01/01/2005 # nieuwjaar
holiday.2=  27/3/2005  # pasen
holiday.3=  28/3/2005  # paasmaandag
holiday.4=  1/5/2005   # feest van de arbeid
holiday.5=  5/5/2005   # hemelvaart
holiday.6=  15/5/2005  # pinksteren
holiday.7=  16/5/2005  # pinkstermaandag
holiday.8=  21/7/2005  # my birthday
holiday.9=  15/8/2005  # moederkesdag
holiday.10= 1/11/2005  # allerheiligen
holiday.11= 11/11/2005 # wapenstilstand
holiday.12= 25/12/2005 # kerstmis

business.day.expressed.in.hours=            8
business.week.expressed.in.hours=          40
business.month.expressed.in.business.days=  21
business.year.expressed.in.business.days=  220
```

Email support

This chapter describes the out-of-the-box email support in jBPM jPDL.

15.1. Mail in jPDL

There are four ways of specifying when emails should be sent from a process.

15.1.1. Mail action

A mail action can be used when the sending of this email should not be shown as a node in the process graph.

Anywhere you are allowed to specify actions in the process, you can specify a mail action like this:

```
<mail actors="#{president}" subject="readmylips" text="nomoretaxes" />
```

The subject and text attributes can also be specified as an element like this:

```
<mail actors="#{president}" >
  <subject>readmylips</subject>
  <text>nomoretaxes</text>
</mail>
```

Each of the fields can contain JSF like expressions. For example:

```
<mail to='#{initiator}' subject='websale' text='your websale of
 #{quantity} #{item} was approved' />
```

For more information about expressions, see *Section 17.3, "Expressions"*.

There are two attribute to specify recipients: **`actors`** and **`to`**. The **`to`** attribute should resolve to a semicolon separated list of email addresses. The **`actors`** attribute should resolve to a semicolon separated list of actorIds. Those actorIds will be resolved to email addresses with by means of address resolving[1].

```
<mail to='admin@mycompany.com' subject='urgent' text='the mailserver is
 down :-)' />
```

For more about how to specify recipients, see *Section 15.3, "Specifying mail recipients"*

Mails can be defined in templates and in the process you can overwrite properties of the templates like this:

```
<mail template='sillystatement' actors="#{president}" />
```

More about templates can be found in *Section 15.4, "Mail templates"*

[1] *Section 15.3.2, "Address resolving"*.

15.1.2. Mail node

Just the same as with mail actions, sending of an email can also be modeled as a node. In that case, the runtime behavior is just the same, but the email will show up as a node in the process graph.

The attributes and elements supported by mail nodes are exactly the same as with the mail actions[2].

```
<mail-node name="send
 email" to="#{president}" subject="readmylips" text="nomoretaxes">
  <transition to="the next node" />
</mail-node>
```

Mail nodes should have exactly one leaving transition.

15.1.3. Task assign mails

A notification email can be send when a task gets assigned to an actor. Just use the **`notify="yes"`** attribute on a task like this:

```
<task-node name='a'>
  <task name='laundry' swimlane="grandma" notify='yes' />
  <transition to='b' />
</task-node>
```

Setting notify to yes, true or on will cause jBPM to send an email to the actor that will be assigned to this task. The email is based on a template (see *Section 15.4, "Mail templates"*) and contains a link to the task page of the web application.

15.1.4. Task reminder mails

Similarly as with assignments, emails can be sent as a task reminder. The **`reminder`** element in jPDL is based upon the timer. The most common attributes will be the **`duedate`** and the **`repeat`**. The only difference is that no action has to be specified.

```
<task-node name='a'>
  <task name='laundry' swimlane="grandma" notify='yes'>
    <reminder duedate="2 business days" repeat="2 business hours"/>
  </task>
  <transition to='b' />
</task-node>
```

15.2. Expressions in mails

The fields **`to`**, **`recipients`**, **`subject`** and **`text`** can contain JSF-like expressions. For more information about expressions, see *Section 17.3, "Expressions"*

The variables in the expressions can be : swimlanes, process variables, transient variables beans configured in the jbpm.cfg.xml, ...

[2] *Section 15.1.1, "Mail action"*.

These expressions can be combined with the address resolving[3] that is explained later in this chapter. For example, suppose that you have a swimlane called president in your process, then look at the following mail specification:

```
<mail actors="#{president}" subject="readmylips" text="nomoretaxes" />
```

That will send an email to to the person that acts as the president for that particular process execution.

15.3. Specifying mail recipients

15.3.1. Multiple recipients

In the **actors** and **to** fields, multiple recipients can be separated with a semi colon (;) or a colon (:).

15.3.2. Address resolving

In all of jBPM, actors are referenced by actorId's. This is a string that serves as the identifier of the process participant. An address resolver translates actorId's into email addresses.

Use the attribute actors in case you want to apply address resolving and use the attribute to in case you are specifying email addresses directly and don't want to apply address resolving.

An address resolver should implement the following interface:

```
public interface AddressResolver extends Serializable {
  Object resolveAddress(String actorId);
}
```

An address resolver should return 1 of 3 types: a String, a Collection of Strings or an array of Strings. All strings should represent email addresses for the given actorId.

The address resolver implementation should be a bean configured in the jbpm.cfg.xml with name **jbpm.mail.address.resolver** like this:

```
<jbpm-configuration>
  <bean name='jbpm.mail.address.resolver' class='org.jbpm.identity.mail.IdentityAdd
 />
</jbpm-configuration>
```

The identity component of jBPM includes an address resolver. That address resolver will look for the User of the given actorId. If the user exists, the user's email is returned, otherwise null. More on the identity component can be found in *Section 10.11, "The identity component"*.

15.4. Mail templates

Instead of specifying mails in the processdefinition.xml, mails can be specified in a template file. When a template is used, each of the fields can still be overwritten in the processdefinition.xml. The mail templates should be specified in an XML file like this:

[3] *Section 15.3.2, "Address resolving"*.

```
<mail-templates>

  <variable name="BaseTaskListURL" value="http://localhost:8080/jbpm/
task?id=" />

  <mail-template name='task-assign'>
    <actors>#{taskInstance.actorId}</actors>
    <subject>Task '#{taskInstance.name}'</subject>
    <text><![CDATA[Hi,
Task '#{taskInstance.name}' has been assigned to you.
Go for it: #{BaseTaskListURL}#{taskInstance.id}
Thanks.
---powered by JBoss jBPM---]]></text>
  </mail-template>

  <mail-template name='task-reminder'>
    <actors>#{taskInstance.actorId}</actors>
    <subject>Task '#{taskInstance.name}' !</subject>
    <text><![CDATA[Hey,
Don't forget about #{BaseTaskListURL}#{taskInstance.id}
Get going !
---powered by JBoss jBPM---]]></text>
  </mail-template>

</mail-templates>
```

As you can see in this example (BaseTaskListURL), extra variables can be defined in the mail templates that will be available in the expressions.

The resource that contains the templates should be configured in the jbpm.cfg.xml like this:

```
<jbpm-configuration>
  <string name="resource.mail.templates" value="jbpm.mail.templates.xml"
 />
</jbpm-configuration>
```

15.5. Mail server configuration

The simplest way to configure the mail server is with the configuration property **`jbpm.mail.smtp.host`** in the jbpm.cfg.xml like this:

```
<jbpm-configuration>
  <string name="jbpm.mail.smtp.host" value="localhost" />
</jbpm-configuration>
```

Alternatively, when more properties need to be specified, a resource reference to a properties file can be given with the key '' like this:

```
<jbpm-configuration>
```

```
  <string name='resource.mail.properties' value='jbpm.mail.properties' />
</jbpm-configuration>
```

15.6. From address configuration

The default value for the From address used in jPDL mails is **`jbpm@noreply`**. The from address of mails can be configured in the jBPM configuration file jbpm.xfg.xml with key 'jbpm.mail.from.address' like this:

```
<jbpm-configuration>
  <string name='jbpm.mail.from.address' value='jbpm@yourcompany.com' />
</jbpm-configuration>
```

15.7. Customizing mail support

All the mail support in jBPM is centralized in one class: **`org.jbpm.mail.Mail`** This is an ActionHandler implementation. Whenever an mail is specified in the process xml, this will result in a delegation to the mail class. It is possible to inherit from the Mail class and customize certain behavior for your particular needs. To configure your class to be used for mail delegations, specify a 'jbpm.mail.class.name' configuration string in the jbpm.cfg.xml like this:

```
<jbpm-configuration>
  <string name='jbpm.mail.class.name' value='com.your.specific.CustomMail'
 />
</jbpm-configuration>
```

The customized mail class will be read during parsing and actions will be configured in the process that reference the configured (or the default) mail classname. So if you change the property, all the processes that were already deployed will still refer to the old mail class name. But they can be easily updated with one simple update statement to the jbpm database.

15.8. Mail server

If you need a mail server that is easy to install, checkout *JBossMail Server*[4] or *Apache James*[5]

[4] http://www.jboss.org/products/mailservices
[5] http://james.apache.org/

Logging

The purpose of logging is to keep track of the history of a process execution. As the runtime data of a process execution changes, all the deltas are stored in the logs.

Process logging, which is covered in this chapter, is not to be confused with software logging. Software logging traces the execution of a software program (usually for debugging purposes). Process logging traces the execution of process instances.

There are various use cases for process logging information. Most obvious is the consulting of the process history by participants of a process execution.

Another use case is Business Activity Monitoring (BAM). BAM will query or analyze the logs of process executions to find useful statistical information about the business process. E.g. how much time is spent on average in each step of the process, where the bottlenecks in the process are etc. This information is key to implement real business process management in an organization. Real business process management is about how an organization manages their processes, how these are supported by information technology *and* how these two improve the other in an iterative process.

Next use case is the undo functionality. Process logs can be used to implement the undo. Since the logs contain the deltas of the runtime information, the logs can be played in reverse order to bring the process back into a previous state.

16.1. Creation of logs

Logs are produced by jBPM modules while they are running process executions.
But also users can insert process logs. A log entry is a java object that inherits from **`org.jbpm.logging.log.ProcessLog`**. Process log entries are added to the **`LoggingInstance`**. The **`LoggingInstance`** is an optional extension of the **`ProcessInstance`**.

Various kinds of logs are generated by jBPM : graph execution logs, context logs and task management logs. For more information about the specific data contained in those logs, we refer to the javadocs. A good starting point is the class **`org.jbpm.logging.log.ProcessLog`** since from that class you can navigate down the inheritance tree.

The **`LoggingInstance`** will collect all the log entries. When the **`ProcessInstance`** is saved, all the logs in the **`LoggingInstance`** will be flushed to the database. The **`logs`**-field of a **`ProcessInstance`** is not mapped with hibernate to avoid that logs are retrieved from the database in each transactions. Each **`ProcessLog`** is made in the context of a path of execution (**`Token`**) and hence, the **`ProcessLog`** refers to that token. The **`Token`** also serves as an index-sequence generator for the index of the **`ProcessLog`** in the **`Token`**. This will be important for log retrieval. That way, logs that are produced in subsequent transactions will have sequential sequence numbers. (wow, that a lot of seq's in there :-s).

The API method for adding process logs is the following.

```
public class LoggingInstance extends ModuleInstance {
  ...
  public void addLog(ProcessLog processLog) {...}
  ...
}
```

The UML diagram for logging information looks like this:

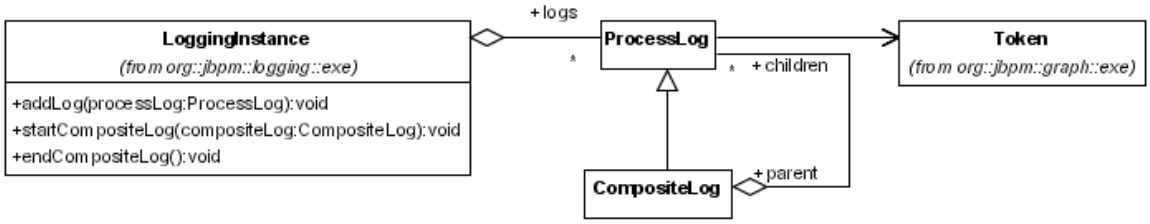

Figure 16.1. The jBPM logging information class diagram

A **`CompositeLog`** is a special kind of log entry. It serves as a parent log for a number of child logs, thereby creating the means for a hierarchical structure in the logs. The API for inserting a log is the following.

```
public class LoggingInstance extends ModuleInstance {
  ...
  public void startCompositeLog(CompositeLog compositeLog) {...}
  public void endCompositeLog() {...}
  ...
}
```

The **`CompositeLog`**s should always be called in a **`try-finally`**-block to make sure that the hierarchical structure of logs is consistent. For example:

```
startCompositeLog(new MyCompositeLog());
try {
  ...
} finally {
  endCompositeLog();
}
```

16.2. Log configurations

For deployments where logs are not important, it suffices to remove the logging line in the jbpm-context section of the jbpm.cfg.xml configuration file:

```
<service name='logging' factory='org.jbpm.logging.db.DbLoggingServiceFactory'
 />
```

In case you want to filter the logs, you need to write a custom implementation of the LoggingService that is a subclass of DbLoggingService. Also you need to create a custom logging ServiceFactory and specify that one in the factory attribute.

16.3. Log retrieval

As said before, logs cannot be retrieved from the database by navigating the LoggingInstance to its logs. Instead, logs of a process instance should always be queried from the database. The **`LoggingSession`** has 2 methods that serve this purpose.

The first method retrieves all the logs for a process instance. These logs will be grouped by token in a Map. The map will associate a List of ProcessLogs with every Token in the process instance. The list will contain the ProcessLogs in the same ordered as they were created.

```
public class LoggingSession {
  ...
  public Map findLogsByProcessInstance(long processInstanceId) {...}
  ...
}
```

The second method retrieves the logs for a specific Token. The returned list will contain the ProcessLogs in the same ordered as they were created.

```
public class LoggingSession {
  public List findLogsByToken(long tokenId) {...}
  ...
}
```

16.4. Database warehousing

Sometimes you may want to apply data warehousing techniques to the jbpm process logs. Data warehousing means that you create a separate database containing the process logs to be used for various purposes.

There may be many reasons why you want to create a data warehouse with the process log information. Sometimes it might be to offload heavy queries from the 'live' production database. In other situations it might be to do some extensive analysis. Data warehousing even might be done on a modified database schema which is optimized for its purpose.

In this section, we only want to propose the technique of warehousing in the context of jBPM. The purposes are too diverse, preventing a generic solution to be included in jBPM that could cover all those requirements.

jBPM Process Definition Language (JPDL)

JPDL specifies an xml schema and the mechanism to package all the process definition related files into a process archive.

17.1. The process archive

A process archive is a zip file. The central file in the process archive is **`processdefinition.xml`**. The main information in that file is the process graph. The **`processdefinition.xml`** also contains information about actions and tasks. A process archive can also contain other process related files such as classes, ui-forms for tasks, ...

17.1.1. Deploying a process archive

Deploying process archives can be done in 3 ways: with the process designer tool, with an ant task or programatically.

Deploying a process archive with the designer tool is supported in the starters-kit. Right click on the process archive folder to find the "Deploy process archive" option. The starters-kit server contains the jBPM web application, which has a servlet to upload process archives called ProcessUploadServlet. This servlet is capable of uploading process archives and deploying them to the default jBPM instance configured.

Deploying a process archive with an ant task can be done as follows:

```
<target name="deploy.par">
  <taskdef name="deploypar" classname="org.jbpm.ant.DeployProcessTask">
    <classpath --make sure the jbpm-[version].jar is in this classpath--/
>
  </taskdef>
  <deploypar par="build/myprocess.par" />
</target>
```

To deploy more process archives at once, use the nested fileset elements. The file attribute itself is optional. Other attributes of the ant task are:

- **cfg**: cfg is optional, the default value is 'hibernate.cfg.xml'. The hibernate configuration file that contains the jdbc connection properties to the database and the mapping files.
- **properties**: properties is optional and overwrites *all* hibernate properties as found in the hibernate.cfg.xml
- **createschema**: if set to true, the jbpm database schema is created before the processes get deployed.

Process archives can also be deployed programmatically with the class **`org.jbpm.jpdl.par.ProcessArchiveDeployer`**

17.1.2. Process versioning

What happens when we have a process definition deployed, many executions are not yet finished and we have a new version of the process definition that we want to deploy ?

Process instances always execute to the process definition that they are started in. But jBPM allows for multiple process definitions of the same name to coexist in the database. So typically, a process instance is started in the latest version available at that time and it will keep on executing in that same process definition for its complete lifetime. When a newer version is deployed, newly created instances will be started in the newest version, while older process instances keep on executing in the older process definitions.

If the process includes references to Java classes, the java classes can be made available to the jBPM runtime environment in 2 ways : by making sure these classes are visible to the jBPM classloader. This usually means that you can put your delegation classes in a **`.jar`** file next to the **`jbpm-[version].jar`**. In that case, all the process definitions will see that same class file. The java classes can also be included in the process archive. When you include your delegation classes in the process archive (and they are not visible to the jbpm classloader), jBPM will also version these classes inside the process definition. More information about process classloading can be found in *Section 17.2, "Delegation"*

When a process archive gets deployed, it creates a process definition in the jBPM database. Process definitions can be versioned on the basis of the process definition name. When a named process archive gets deployed, the deployer will assign a version number. To assign this number, the deployer will look up the highest version number for process definitions with the same name and adds 1. Unnamed process definitions will always have version number -1.

17.1.3. Changing deployed process definitions

Changing process definitions after they are deployed into the jBPM database has many potential pitfalls. Therefor, this is highly discouraged.

Actually, there is a whole variety of possible changes that can be made to a process definition. Some of those process definitions are harmless, but some other changes have implications far beyond the expected and desirable.

So please consider migrating process instances to a new definition (see *Section 17.1.4, "Migrating process instances"*)over this approach.

In case you would consider it, these are the points to take into consideration:

Use Hibernate's update: You can just load a process definition, change it and save it with the hibernate session. The hibernate session can be accessed with the method **`JbpmContext.getSession()`**.

The second level cache: A process definition would need to be removed from the second level cache after you've updated an existing process definition. See also *Section 6.9, "Second level cache"*

17.1.4. Migrating process instances

An alternative approach to changing process definitions might be to convert the executions to a new process definition. Please take into account that this is not trivial due to the long-lived nature of business processes. Currently, this is an experimental area so for which there are not yet much out-of-the-box support.

As you know there is a clear distinction between process definition data, process instance data (the runtime data) and the logging data. With this approach, you create a separate new process definition in the jBPM database (by e.g. deploying a new version of the same process). Then the runtime information is converted to the new process definition. This might involve a translation cause tokens in the old process might be pointing to nodes that have been removed in the new version. So only new data is created in the database. But one execution of a process is spread over two process instance objects. This might become a bit tricky for the tools and statistics calculations. When resources permit us, we are going to add support for this in the future. E.g. a pointer could be added from one process instance to it's predecessor.

17.1.5. Process conversion

A conversion class has been made available to assist you with converting your jBPM 2.0 process archives into jBPM 3.0 compatible process archives. Create an output directory to hold the converted process archives. Enter the following command line from the build directory of the jBPM 3.0 distribution:

java -jar converter.jar indirectory outdirectory

Substitute the input directory where your jBPM 2.0 process archives reside for "indirectory". Substitute the output directory for the one you created to hold the newly converted process archives for "outdirectory".

17.2. Delegation

Delegation is the mechanism used to include the users' custom code in the execution of processes.

17.2.1. The jBPM class loader

The jBPM class loader is the class loader that loads the jBPM classes. Meaning, the classloader that has the library **`jbpm-3.x.jar`** in its classpath. To make classes visible to the jBPM classloader, put them in a jar file and put the jar file besides the **`jbpm-3.x.jar`**. E.g. in the WEB-INF/lib folder in the case of web applications.

17.2.2. The process class loader

Delegation classes are loaded with the process class loader of their respective process definition. The process class loader is a class loader that has the jBPM classloader as a parent. The process class loader adds all the classes of one particular process definition. You can add classes to a process definition by putting them in the **`/classes`** folder in the process archive. Note that this is only useful when you want to version the classes that you add to the process definition. If versioning is not necessary, it is much more efficient to make the classes available to the jBPM class loader.

If the resource name doesn't start with a slash, resources are also loaded from the **`/classes`** directory in the process archive. If you want to load resources outside of the classes directory, start with a double slash (**`//`**). For example to load resource **`data.xml`** which is located next to the processdefinition.xml on the root of the process archive file, you can do **`class.getResource("//data.xml")`** or **`classLoader.getResourceAsStream("//data.xml")`** or any of those variants.

17.2.3. Configuration of delegations

Delegation classes contain user code that is called from within the execution of a process. The most common example is an action. In the case of action, an implementation of the interface **`ActionHandler`** can be called on an event in the process. Delegations are specified in the **`processdefinition.xml`**. 3 pieces of data can be supplied when specifying a delegation :

1. the class name (required) : the fully qualified class name of the delegation class.
2. configuration type (optional) : specifies the way to instantiate and configure the delegation object. By default the default constructor is used and the configuration information is ignored.
3. configuration (optional) : the configuration of the delegation object in the format as required by the configuration type.

Next is a description of all the configuration types:

17.2.3.1. config-type field

This is the default configuration type. The **`config-type field`** will first instantiate an object of the delegation class and then set values in the fields of the object as specified in the configuration. The configuration is xml, where the element names have to correspond with the field names of the class. The content text of the element is put in the corresponding field. If necessary and possible, the content text of the element is converted to the field type.

Supported type conversions:

- String doesn't need converting, of course. But it is trimmed.
- primitive types such as int, long, float, double, ...
- and the basic wrapper classes for the primitive types.
- lists, sets and collections. In that case each element of the xml-content is considered as an element of the collection and is parsed, recursively applying the conversions. If the type of the elements is different from **`java.lang.String`** this can be indicated by specifying a type attribute with the fully qualified type name. For example, following snippet will inject an ArrayList of Strings into field 'numbers':

```
<numbers>
  <element>one</element>
  <element>two</element>
  <element>three</element>
</numbers>
```

The text in the elements can be converted to any object that has a String constructor. To use another type then String, specify the **`element-type`** in the field element ('numbers' in this case).

Here's another example of a map:

```
<numbers>
  <entry><key>one</key><value>1</value></entry>
  <entry><key>two</key><value>2</value></entry>
  <entry><key>three</key><value>3</value></entry>
```

```
  </numbers>
```

- maps. In this case, each element of the field-element is expected to have one sub-element **`key`** and one element **`value`**. The key and element are both parsed using the conversion rules recursively. Just the same as with collections, a conversion to **`java.lang.String`** is assumed if no **`type`** attribute is specified.
- org.dom4j.Element
- for any other type, the string constructor is used.

For example in the following class...

```
public class MyAction implements ActionHandler {
  // access specifiers can be private, default, protected or public
  private String city;
  Integer rounds;
  ...
}
```

...this is a valid configuration:

```
...
<action class="org.test.MyAction">
  <city>Atlanta</city>
  <rounds>5</rounds>
</action>
...
```

17.2.3.2. config-type bean

Same as **`config-type field`** but then the properties are set via setter methods, rather then directly on the fields. The same conversions are applied.

17.2.3.3. config-type constructor

This instantiator will take the complete contents of the delegation xml element and passes this as text in the delegation class constructor.

17.2.3.4. config-type configuration-property

First, the default constructor is used, then this instantiator will take the complete contents of the delegation xml element, and pass it as text in method **`void configure(String);`**. (as in jBPM 2)

17.3. Expressions

For some of the delegations, there is support for a JSP/JSF EL like expression language.
In actions, assignments and decision conditions, you can write an expression like e.g.
`expression="#{myVar.handler[assignments].assign}"`

The basics of this expression language can be found *in the J2EE tutorial*[1].

The jPDL expression language is similar to the JSF expression language. Meaning that jPDL EL is based on JSP EL, but it uses **`#{...}`** notation and that it includes support for method binding.

Depending on the context, the process variables or task instance variables can be used as starting variables along with the following implicit objects:

- taskInstance (org.jbpm.taskmgmt.exe.TaskInstance)
- processInstance (org.jbpm.graph.exe.ProcessInstance)
- processDefinition (org.jbpm.graph.def.ProcessDefinition)
- token (org.jbpm.graph.exe.Token)
- taskMgmtInstance (org.jbpm.taskmgmt.exe.TaskMgmtInstance)
- contextInstance (org.jbpm.context.exe.ContextInstance)

This feature becomes really powerful in a JBoss SEAM environment. Because of the integration between jBPM and *JBoss SEAM*[2], all of your backed beans, EJB's and other **`one-kind-of-stuff`** becomes available right inside of your process definition. Thanks Gavin ! Absolutely awesome ! :-)

17.4. jPDL xml schema

The jPDL schema is the schema used in the file **`processdefinition.xml`** in the process archive.

17.4.1. Validation

When parsing a jPDL XML document, jBPM will validate your document against the jPDL schema when two conditions are met: first, the schema has to be referenced in the XML document like this

```
<process-definition xmlns="urn:jbpm.org:jpdl-3.2">
  ...
</process-definition>
```

And second, the Xerxes parser has to be on the classpath.

The jPDL schema can be found in **`${jbpm.home}/src/java.jbpm/org/jbpm/jpdl/xml/jpdl-3.2.xsd`** or at *http://jbpm.org/jpdl-3.2.xsd*.

17.4.2. process-definition

Name	Type	Multiplicity	Description
name	attribute	optional	the name of the process
swimlane[1]	element	[0..*]	the swimlanes used in this process. The swimlanes represent process roles and they are used for task assignments.

[1] http://java.sun.com/j2ee/1.4/docs/tutorial/doc/JSPIntro7.html
[2] http://www.jboss.com/products/seam

Name	Type	Multiplicity	Description
start-state[2]	element	[0..1]	the start state of the process. Note that a process without a start-state is valid, but cannot be executed.
task[3]	element	[0..*]	global defined tasks that can be used in e.g. actions.
exception-handler[4]	element	[0..*]	a list of exception handlers that applies to all exceptions thrown by delegation classes thrown in this process definition.
{end-state [5]|state[6]| node[7]|task-node[8]| process-state[9]|super-state[10]|fork[11]|join[12]| decision[13]}	element	[0..*]	the nodes of the process definition. Note that a process without nodes is valid, but cannot be executed.
event[14]	element	[0..*]	the process events that serve as a container for actions
{action[15]|script[16]| create-timer[17]|cancel-timer[18]}	element	[0..*]	global defined actions that can be referenced from events and transitions. Note that these actions must specify a name in order to be referenced.

[1] *Section 17.4.25, "swimlane"*.
[2] *Section 17.4.5, "start-state"*.
[3] *Section 17.4.24, "task"*.
[4] *Section 17.4.30, "exception-handler"*.
[5] *Section 17.4.6, "end-state"*.
[6] *Section 17.4.7, "state"*.
[7] *Section 17.4.3, "node"*.
[8] *Section 17.4.8, "task-node"*.
[9] *Section 17.4.9, "process-state"*.
[10] *Section 17.4.10, "super-state"*.
[11] *Section 17.4.11, "fork"*.
[12] *Section 17.4.12, "join"*.
[13] *Section 17.4.13, "decision"*.
[14] *Section 17.4.14, "event"*.
[15] *Section 17.4.16, "action"*.
[16] *Section 17.4.17, "script"*.
[17] *Section 17.4.22, "create-timer"*.
[18] *Section 17.4.23, "cancel-timer"*.

Table 17.1. Process Definition Schema

17.4.3. node

Name	Type	Multiplicity	Description
{action[1]|script[2]|create-timer[3]|cancel-timer[4]}	element	1	a custom action that represents the behavior for this node
common node elements[5]			See *Section 17.4.4, "common node elements"*

[1] *Section 17.4.16, "action"*.
[2] *Section 17.4.17, "script"*.
[3] *Section 17.4.22, "create-timer"*.

[4] *Section 17.4.23, "cancel-timer"*.
[5] *Section 17.4.4, "common node elements"*.
Table 17.2. Node Schema

17.4.4. common node elements

Name	Type	Multiplicity	Description
name	attribute	required	the name of the node
async	attribute	{ true | false }, false is the default	If set to true, this node will be executed asynchronously. See also *Section 3.3.4, "Asynchronous continuations"*
transition[1]	element	[0..*]	the leaving transitions. Each transition leaving a node *must* have a distinct name. A maximum of one of the leaving transitions is allowed to have no name. The first transition that is specified is called the default transition. The default transition is taken when the node is left without specifying a transition.
event[2]	element	[0..*]	supported event types: {node-enter|node-leave}
exception-handler[3]	element	[0..*]	a list of exception handlers that applies to all exceptions thrown by delegation classes thrown in this process node.
timer[4]	element	[0..*]	specifies a timer that monitors the duration of an execution in this node.

[1] *Section 17.4.15, "transition"*.
[2] *Section 17.4.14, "event"*.
[3] *Section 17.4.30, "exception-handler"*.
[4] *Section 17.4.21, "timer"*.
Table 17.3. Common Node Schema

17.4.5. start-state

Name	Type	Multiplicity	Description
name	attribute	optional	the name of the node
task[1]	element	[0..1]	the task to start a new instance for this process or to capture the process initiator. See *Section 10.7, "Swimlane in start task"*
event[2]	element	[0..*]	supported event types: {node-leave}
transition[3]	element	[0..*]	the leaving transitions. Each transition leaving a node *must* have a distinct name.
exception-handler[4]	element	[0..*]	a list of exception handlers that applies to all exceptions thrown by delegation classes thrown in this process node.

[1] *Section 17.4.24, "task"*.
[2] *Section 17.4.14, "event"*.

[3] *Section 17.4.15, "transition"*.
[4] *Section 17.4.30, "exception-handler"*.
Table 17.4. Start State Schema

17.4.6. end-state

Name	Type	Multiplicity	Description
name	attribute	required	the name of the end-state
event[1]	element	[0..*]	supported event types: {node-enter}
exception-handler[2]	element	[0..*]	a list of exception handlers that applies to all exceptions thrown by delegation classes thrown in this process node.

[1] *Section 17.4.14, "event"*.
[2] *Section 17.4.30, "exception-handler"*.
Table 17.5. End State Schema

17.4.7. state

Name	Type	Multiplicity	Description
common node elements[1]			See *Section 17.4.4, "common node elements"*

[1] *Section 17.4.4, "common node elements"*.
Table 17.6. State Schema

17.4.8. task-node

Name	Type	Multiplicity	Description
signal	attribute	optional	{unsynchronized|never|first|first-wait|last|last-wait}, default is **`last`**. signal specifies the effect of task completion on the process execution continuation.
create-tasks	attribute	optional	{yes|no|true|false}, default is **`true`**. can be set to false when a runtime calculation has to determine which of the tasks have to be created. in that case, add an action on **`node-enter`**, create the tasks in the action and set **`create-tasks`** to **`false`**.
end-tasks	attribute	optional	{yes|no|true|false}, default is **`false`**. In case remove-tasks is set to true, on **`node-leave`**, all the tasks that are still open are ended.
task[1]	element	[0..*]	the tasks that should be created when execution arrives in this task node.
common node elements[2]			See *Section 17.4.4, "common node elements"*

[1] *Section 17.4.24, "task"*.
[2] *Section 17.4.4, "common node elements"*.
Table 17.7. Task Node Schema

17.4.9. process-state

Name	Type	Multiplicity	Description
binding	attribute	optional	Defines the moment a sub-process is resolved. {late|*} defaults to resolving deploy time
sub-process[1]	element	1	the sub process that is associated with this node
variable[2]	element	[0..*]	specifies how data should be copied from the super process to the sub process at the start and from the sub process to the super process upon completion of the sub process.
common node elements[3]			See *Section 17.4.4, "common node elements"*

[1] *Section 17.4.28, "sub-process"*.
[2] *Section 17.4.19, "variable"*.
[3] *Section 17.4.4, "common node elements"*.

Table 17.8. Process State Schema

17.4.10. super-state

Name	Type	Multiplicity	Description
{end-state[1]|state[2]| node[3]|task-node[4]| process-state[5]|super-state[6]|fork[7]|join[8]| decision[9]}	element	[0..*]	the nodes of the superstate. superstates can be nested.
common node elements[10]			See *Section 17.4.4, "common node elements"*

[1] *Section 17.4.6, "end-state"*.
[2] *Section 17.4.7, "state"*.
[3] *Section 17.4.3, "node"*.
[4] *Section 17.4.8, "task-node"*.
[5] *Section 17.4.9, "process-state"*.
[6] *Section 17.4.10, "super-state"*.
[7] *Section 17.4.11, "fork"*.
[8] *Section 17.4.12, "join"*.
[9] *Section 17.4.13, "decision"*.
[10] *Section 17.4.4, "common node elements"*.

Table 17.9. Super State Schema

17.4.11. fork

Name	Type	Multiplicity	Description
common node elements[1]			See *Section 17.4.4, "common node elements"*

[1] *Section 17.4.4, "common node elements"*.

Table 17.10. Fork Schema

17.4.12. join

Name	Type	Multiplicity	Description
common node elements[1]			See *Section 17.4.4, "common node elements"*

[1] *Section 17.4.4, "common node elements"*.

Table 17.11. Join Schema

17.4.13. decision

Name	Type	Multiplicity	Description
handler[1]	element	either a 'handler' element or conditions on the transitions should be specified	the name of a **`org.jbpm.jpdl.Def.DecisionHandler`** implementation
transition conditions	attribute or element text on the transitions leaving a decision		the leaving transitions. Each leaving transitions of a node can have a condition. The decision will use these conditions to look for the first transition for which the condition evaluates to true. The first transition represents the otherwise branch. So first, all transitions with a condition are evaluated. If one of those evaluate to true, that transition is taken. If no transition with a condition resolves to true, the default transition (=the first one) is taken. See *Section 17.4.29, "condition"*
common node elements[2]			See *Section 17.4.4, "common node elements"*

[1] *Section 17.4.20, "handler"*.

[2] *Section 17.4.4, "common node elements"*.

Table 17.12. Decision Schema

17.4.14. event

Name	Type	Multiplicity	Description
type	attribute	required	the event type that is expressed relative to the element on which the event is placed
{action[1]|script[2]|create-timer[3]|cancel-timer[4]}	element	[0..*]	the list of actions that should be executed on this event

[1] *Section 17.4.16, "action"*.

[2] *Section 17.4.17, "script"*.

[3] *Section 17.4.22, "create-timer"*.

[4] *Section 17.4.23, "cancel-timer"*.

Table 17.13. Event Schema

17.4.15. transition

Name	Type	Multiplicity	Description
name	attribute	optional	the name of the transition. Note that each transition leaving a node *must* have a distinct name.
to	attribute	required	the hierarchical name of the destination node. For more information about hierarchical names, see *Section 8.6.3, "Hierarchical names"*
condition	attribute or element text	optional	a guard condition expression[1]. These condition attributes (or child elements) can be used in decision nodes, or to calculate the available transitions on a token at runtime.
{action[2]|script[3]|create-timer[4]|cancel-timer[5]}	element	[0..*]	the actions to be executed upon taking this transition. Note that the actions of a transition do not need to be put in an event (because there is only one)
exception-handler[6]	element	[0..*]	a list of exception handlers that applies to all exceptions thrown by delegation classes thrown in this process node.

[1] *Section 17.3, "Expressions"*.
[2] *Section 17.4.16, "action"*.
[3] *Section 17.4.17, "script"*.
[4] *Section 17.4.22, "create-timer"*.
[5] *Section 17.4.23, "cancel-timer"*.
[6] *Section 17.4.30, "exception-handler"*.

Table 17.14. Transition Schema

17.4.16. action

Name	Type	Multiplicity	Description
name	attribute	optional	the name of the action. When actions are given names, they can be looked up from the process definition. This can be useful for runtime actions and declaring actions only once.
class	attribute	either, a ref-name or an expression	the fully qualified class name of the class that implements the **`org.jbpm.graph.def.ActionHandler`** interface.
ref-name	attribute	either this or class	the name of the referenced action. The content of this action is not processed further if a referenced action is specified.
expression	attribute	either this, a class or a ref-name	A jPDL expression that resolves to a method. See also *Section 17.3, "Expressions"*
accept-propagated-events	attribute	optional	{yes|no|true|false}. Default is yes|true. If set to false, the action will only be executed on

Name	Type	Multiplicity	Description
			events that were fired on this action's element. for more information, see *Section 8.5.4, "Event propagation"*
config-type	attribute	optional	{field[1]|bean[2]|constructor[3]|configuration-property[4]}. Specifies how the action-object should be constructed and how the content of this element should be used as configuration information for that action-object.
async	attribute	{true|false}	Default is false, which means that the action is executed in the thread of the execution. If set to true, a message will be sent to the command executor and that component will execute the action asynchronously in a separate transaction.
	{content}	optional	the content of the action can be used as configuration information for your custom action implementations. This allows the creation of reusable delegation classes. For more about delegation configuration, see *Section 17.2.3, "Configuration of delegations"*.

[1] *Section 17.2.3.1, "config-type field"*.
[2] *Section 17.2.3.2, "config-type bean"*.
[3] *Section 17.2.3.3, "config-type constructor"*.
[4] *Section 17.2.3.4, "config-type configuration-property"*.

Table 17.15. Action Schema

17.4.17. script

Name	Type	Multiplicity	Description
name	attribute	optional	the name of the script-action. When actions are given names, they can be looked up from the process definition. This can be useful for runtime actions and declaring actions only once.
accept-propagated-events	attribute	optional [0..*]	{yes|no|true|false}. Default is yes|true. If set to false, the action will only be executed on events that were fired on this action's element. for more information, see *Section 8.5.4, "Event propagation"*
expression[1]	element	[0..1]	the beanshell script. If you don't specify variable[2] elements, you can write the expression as the content of the script element (omitting the expression element tag).
variable[3]	element	[0..*]	in variable for the script. If no in variables are specified, all the variables of the current token will be loaded into the script evaluation. Use the in variables if you want to limit the number of variables loaded into the script evaluation.

[1] *Section 17.4.18, "expression"*.

[2] *Section 17.4.19, "variable"*.
[3] *Section 17.4.19, "variable"*.
Table 17.16. Script Schema

17.4.18. expression

Name	Type	Multiplicity	Description
	{content}		a bean shell script.

Table 17.17. Expression Schema

17.4.19. variable

Name	Type	Multiplicity	Description
name	attribute	required	the process variable name
access	attribute	optional	default is **`read,write`**. It is a comma separated list of access specifiers. The only access specifiers used so far are **`read`**, **`write`** and **`required`**.
mapped-name	attribute	optional	this defaults to the variable name. it specifies a name to which the variable name is mapped. the meaning of the mapped-name is dependent on the context in which this element is used. for a script, this will be the script-variable-name. for a task controller, this will be the label of the task form parameter and for a process-state, this will be the variable name used in the sub-process.

Table 17.18. Variable Schema

17.4.20. handler

Name	Type	Multiplicity	Description
expression	attribute	either this or a class	A jPDL expression. The returned result is transformed to a string with the toString() method. The resulting string should match one of the leaving transitions. See also *Section 17.3, "Expressions"*.
class	attribute	either this or ref-name	the fully qualified class name of the class that implements the **`org.jbpm.graph.node.DecisionHandler`** interface.
config-type	attribute	optional	{field[1]|bean[2]|constructor[3]|configuration-property[4]}. Specifies how the action-object should be constructed and how the content of this element should be used as configuration information for that action-object.

Name	Type	Multiplicity	Description
	{content}	optional	the content of the handler can be used as configuration information for your custom handler implementations. This allows the creation of reusable delegation classes. For more about delegation configuration, see *Section 17.2.3, "Configuration of delegations"*.

[1] *Section 17.2.3.1, "config-type field"*.
[2] *Section 17.2.3.2, "config-type bean"*.
[3] *Section 17.2.3.3, "config-type constructor"*.
[4] *Section 17.2.3.4, "config-type configuration-property"*.

Table 17.19. Handler Schema

17.4.21. timer

Name	Type	Multiplicity	Description
name	attribute	optional	the name of the timer. If no name is specified, the name of the enclosing node is taken. Note that every timer should have a unique name.
duedate	attribute	required	the duration (optionally expressed in business hours) that specifies the time period between the creation of the timer and the execution of the timer. See *Section 14.1.1, "Duration"* for the syntax.
repeat	attribute	optional	{duration | 'yes' | 'true'}after a timer has been executed on the duedate, 'repeat' optionally specifies duration between repeating timer executions until the node is left. If **`yes`** or **`true`** is specified, the same duration as for the due date is taken for the repeat. See *Section 14.1.1, "Duration"* for the syntax.
transition	attribute	optional	a transition-name to be taken when the timer executes, after firing the timer event and executing the action (if any).
cancel-event	attribute	optional	this attribute is only to be used in timers of tasks. it specifies the event on which the timer should be canceled. by default, this is the **`task-end`** event, but it can be set to e.g. **`task-assign`** or **`task-start`**. The **`cancel-event`** types can be combined by specifying them in a comma separated list in the attribute.
{action[1]|script[2]|create-timer[3]|cancel-timer[4]}	element	[0..1]	an action that should be executed when this timer fires

[1] *Section 17.4.16, "action"*.
[2] *Section 17.4.17, "script"*.
[3] *Section 17.4.22, "create-timer"*.
[4] *Section 17.4.23, "cancel-timer"*.

Table 17.20. Timer Schema

17.4.22. create-timer

Name	Type	Multiplicity	Description
name	attribute	optional	the name of the timer. The name can be used for canceling the timer with a cancel-timer action.
duedate	attribute	required	the duration (optionally expressed in business hours) that specifies the the time period between the creation of the timer and the execution of the timer. See *Section 14.1.1, "Duration"* for the syntax.
repeat	attribute	optional	{duration | 'yes' | 'true'}after a timer has been executed on the duedate, 'repeat' optionally specifies duration between repeating timer executions until the node is left. If **`yes`** of **`true`** is specified, the same duration as for the due date is taken for the repeat. See *Section 14.1.1, "Duration"* for the syntax.
transition	attribute	optional	a transition-name to be taken when the timer executes, after firing the the timer event and executing the action (if any).

Table 17.21. Create Timer Schema

17.4.23. cancel-timer

Name	Type	Multiplicity	Description
name	attribute	optional	the name of the timer to be canceled.

Table 17.22. Cancel Timer Schema

17.4.24. task

Name	Type	Multiplicity	Description
name	attribute	optional	the name of the task. Named tasks can be referenced and looked up via the **`TaskMgmtDefinition`**
blocking	attribute	optional	{yes|no|true|false}, default is false. If blocking is set to true, the node cannot be left when the task is not finished. If set to false (default) a signal on the token is allowed to continue execution and leave the node. The default is set to false, because blocking is normally forced by the user interface.
signalling	attribute	optional	{yes|no|true|false}, default is true. If signalling is set to false, this task will never have the capability of triggering the continuation of the token.

Name	Type	Multiplicity	Description
duedate	attribute	optional	is a duration expressed in absolute or business hours as explained in *Chapter 14, Business calendar*
swimlane	attribute	optional	reference to a swimlane[1]. If a swimlane is specified on a task, the assignment is ignored.
priority	attribute	optional	one of {highest, high, normal, low, lowest}. alternatively, any integer number can be specified for the priority. FYI: (highest=1, lowest=5)
assignment[2]	element	optional	describes a delegation[3] that will assign the task to an actor when the task is created.
event[4]	element	[0..*]	supported event types: {task-create|task-start|task-assign|task-end}. Especially for the **`task-assign`** we have added a non-persisted property **`previousActorId`** to the **`TaskInstance`**
exception-handler[5]	element	[0..*]	a list of exception handlers that applies to all exceptions thrown by delegation classes thrown in this process node.
timer[6]	element	[0..*]	specifies a timer that monitors the duration of an execution in this task. special for task timers, the **`cancel-event`** can be specified. by default the **`cancel-event`** is **`task-end`**, but it can be customized to e.g. **`task-assign`** or **`task-start`**.
controller[7]	element	[0..1]	specifies how the process variables are transformed into task form parameters. the task form parameters are used by the user interface to render a task form to the user.

[1] *Section 17.4.25, "swimlane"*.
[2] *Section 17.4.26, "assignment"*.
[3] *Section 17.2, "Delegation"*.
[4] *Section 17.4.14, "event"*.
[5] *Section 17.4.30, "exception-handler"*.
[6] *Section 17.4.21, "timer"*.
[7] *Section 17.4.27, "controller"*.

Table 17.23. Task Schema

17.4.25. swimlane

Name	Type	Multiplicity	Description
name	attribute	required	the name of the swimlane. Swimlanes can be referenced and looked up via the **`TaskMgmtDefinition`**
assignment[1]	element	[1..1]	specifies a the assignment of this swimlane. the assignment will be performed when the first task instance is created in this swimlane.

[1] *Section 17.4.26, "assignment"*.

Table 17.24. Swimlane Schema

17.4.26. assignment

Name	Type	Multiplicity	Description
expression	attribute	optional	For historical reasons, this attribute expression does **not** refer to the jPDL expression[1], but instead, it is an assignment expression for the jBPM identity component. For more information on how to write jBPM identity component expressions, see *Section 10.11.2, "Assignment expressions"*. Note that this implementation has a dependency on the jbpm identity component.
actor-id	attribute	optional	An actorId. Can be used in conjunction with pooled-actors. The actor-id is resolved as an expression[2]. So you can refer to a fixed actorId like this **`actor-id="bobthebuilder"`**. Or you can refer to a property or method that returns a String like this: **`actor-id="myVar.actorId"`**, which will invoke the getActorId method on the task instance variable "myVar".
pooled-actors	attribute	optional	A comma separated list of actorIds. Can be used in conjunction with actor-id. A fixed set of pooled actors can be specified like this: **`pooled-actors="chicagobulls, pointersisters"`**. The pooled-actors will be resolved as an expression[3]. So you can also refer to a property or method that has to return, a String[], a Collection or a comma separated list of pooled actors.
class	attribute	optional	the fully qualified classname of an implementation of **`org.jbpm.taskmgmt.def.AssignmentHandler`**
config-type	attribute	optional	{field[4]|bean[5]|constructor[6]|configuration-property[7]}. Specifies how the assignment-handler-object should be constructed and how the content of this element should be used as configuration information for that assignment-handler-object.
	{content}	optional	the content of the assignment-element can be used as configuration information for your AssignmentHandler implementations. This allows the creation of reusable delegation classes. for more about delegation configuration, see *Section 17.2.3, "Configuration of delegations"*.

[1] *Section 17.3, "Expressions"*.
[2] *Section 17.3, "Expressions"*.
[3] *Section 17.3, "Expressions"*.
[4] *Section 17.2.3.1, "config-type field"*.
[5] *Section 17.2.3.2, "config-type bean"*.
[6] *Section 17.2.3.3, "config-type constructor"*.

[7] *Section 17.2.3.4, "config-type configuration-property"*.

Table 17.25. Assignment Schema

17.4.27. controller

Name	Type	Multiplicity	Description
class	attribute	optional	the fully qualified classname of an implementation of **`org.jbpm.taskmgmt.def.TaskControllerHandler`**
config-type	attribute	optional	{field[1]|bean[2]|constructor[3]|configuration-property[4]}. Specifies how the assignment-handler-object should be constructed and how the content of this element should be used as configuration information for that assignment-handler-object.
	{content}		either the content of the controller is the configuration of the specified task controller handler (if the class attribute is specified. if no task controller handler is specified, the content must be a list of variable elements.
variable[5]	element	[0..*]	in case no task controller handler is specified by the class attribute, the content of the controller element must be a list of variables.

[1] *Section 17.2.3.1, "config-type field"*.

[2] *Section 17.2.3.2, "config-type bean"*.

[3] *Section 17.2.3.3, "config-type constructor"*.

[4] *Section 17.2.3.4, "config-type configuration-property"*.

[5] *Section 17.4.19, "variable"*.

Table 17.26. Controller Schema

17.4.28. sub-process

Name	Type	Multiplicity	Description
name	attribute	required	the name of the sub process. Can be an EL expression, as long as it resolves to a String. Powerful especially with late binding in the process-state. To know how you can test sub-processes, see *Section 19.3, "Testing sub processes"*
version	attribute	optional	the version of the sub process. If no version is specified, the latest version of the given process as known while deploying the parent process-state[1] will be taken.
binding	attribute	optional	indicates if the version of the sub process should be determined when deploying the parent process-state[2] (default behavior), or when actually invoking the sub process (**`binding="late"`**). When both **`version`** and **`binding="late"`** are given then jBPM will use the version as

Name	Type	Multiplicity	Description
			requested, but will not yet try to find the sub process when the parent process-state is deployed.

[1] *Section 17.4.9, "process-state"*.

[2] *Section 17.4.9, "process-state"*.

Table 17.27. Sub Process Schema

17.4.29. condition

Name	Type	Multiplicity	Description
	{content} For backwards compatibility, the condition can also be entered with the 'expression' attribute, but that attribute is deprecated since 3.2	required	The contents of the condition element is a jPDL expression[1] that should evaluate to a boolean. A decision takes the first transition (as ordered in the processdefinition.xml) for which the expression resolves to **`true`**. If none of the conditions resolve to true, the default leaving transition (== the first one) will be taken.

[1] *Section 17.3, "Expressions"*.

Table 17.28. Condition Schema

17.4.30. exception-handler

Name	Type	Multiplicity	Description
exception-class	attribute	optional	specifies the fully qualified name of the java throwable class that should match this exception handler. If this attribute is not specified, it matches all exceptions (**`java.lang.Throwable`**).
action[1]	element	[1..*]	a list of actions to be executed when an exception is being handled by this exception handler.

[1] *Section 17.4.16, "action"*.

Table 17.29. Exception Handler Schema

Security

Security features of jBPM are still in alpha stage. This chapter documents the pluggable authentication and authorization. And what parts of the framework are finished and what parts not yet.

18.1. TODOS

On the framework part, we still need to define a set of permissions that are verified by the jbpm engine while a process is being executed. Currently you can check your own permissions, but there is not yet a jbpm default set of permissions.

Only one default authentication implementation is finished. Other authentication implementations are envisioned, but not yet implemented. Authorization is optional, and there is no authorization implementation yet. Also for authorization, there are a number of authorization implementations envisioned, but they are not yet worked out.

But for both authentication and authorization, the framework is there to plug in your own authentication and authorization mechanism.

18.2. Authentication

Authentication is the process of knowing on who's behalf the code is running. In case of jBPM this information should be made available from the environment to jBPM. Cause jBPM is always executed in a specific environment like a web application, an EJB, a swing application or some other environment, it is always the surrounding environment that should perform authentication.

In a few situations, jBPM needs to know who is running the code. E.g. to add authentication information in the process logs to know who did what and when. Another example is calculation of an actor based on the current authenticated actor.

In each situation where jBPM needs to know who is running the code, the central method **`org.jbpm.security.Authentication.getAuthenticatedActorId()`** is called. That method will delegate to an implementation of **`org.jbpm.security.authenticator.Authenticator`**. By specifying an implementation of the authenticator, you can configure how jBPM retrieves the currently authenticated actor from the environment.

The default authenticator is **`org.jbpm.security.authenticator.JbpmDefaultAuthenticator`**. That implementation will maintain a **`ThreadLocal`** stack of authenticated actorId's. Authenticated blocks can be marked with the methods **`JbpmDefaultAuthenticator.pushAuthenticatedActorId(String)`** and **`JbpmDefaultAuthenticator.popAuthenticatedActorId()`**. Be sure to always put these demarcations in a try-finally block. For the push and pop methods of this authenticator implementation, there are convenience methods supplied on the base Authentication class. The reason that the JbpmDefaultAuthenticator maintains a stack of actorIds instead of just one actorId is simple: it allows the jBPM code to distinct between code that is executed on behalf of the user and code that is executed on behalf of the jbpm engine.

See the javadocs for more information.

18.3. Authorization

Authorization is validating if an authenticated user is allowed to perform a secured operation.

The jBPM engine and user code can verify if a user is allowed to perform a given operation with the API method **`org.jbpm.security.Authorization.checkPermission(Permission)`**.

The Authorization class will also delegate that call to a configurable implementation. The interface for plugging in different authorization strategies is **`org.jbpm.security.authorizer.Authorizer`**.

In the package org.jbpm.security.authorizer there are some examples that show intentions of authorized implementations. Most are not fully implemented and none of them are tested.

Also still to do is the definition of a set of jBPM permissions and the verification of those permissions by the jBPM engine. An example could be verifying that the current authenticated user has sufficient privileges to end a task by calling **`Authorization.checkPermission(new TaskPermission("end", Long.toString(id)))`** in the TaskInstance.end() method.

Test Driven Development for Workflow

19.1. Introducing TDD for workflow

Since developing process oriented software is no different from developing any other software, we believe that process definitions should be easily testable. This chapter shows how you can use plain JUnit without any extensions to unit test the process definitions that you author.

The development cycle should be kept as short as possible. Changes made to the sources of software should be immediately verifiable. Preferably, without any intermediate build steps. The examples given below will show you how to develop and test jBPM processes without intermediate steps.

Mostly the unit tests of process definitions are execution scenarios. Each scenario is executed in one JUnit test method and will feed in the external triggers (read: signals) into a process execution and verifies after each signal if the process is in the expected state.

Let's look at an example of such a test. We take a simplified version of the auction process with the following graphical representation:

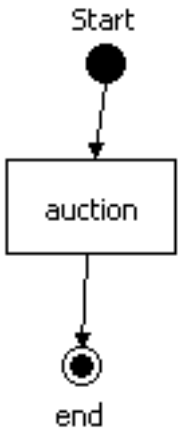

Figure 19.1. The auction test process

Now, let's write a test that executes the main scenario:

```
public class AuctionTest extends TestCase {

  // parse the process definition
  static ProcessDefinition auctionProcess =
      ProcessDefinition.parseParResource("org/jbpm/tdd/auction.par");

  // get the nodes for easy asserting
  static StartState start = auctionProcess.getStartState();
  static State auction = (State) auctionProcess.getNode("auction");
  static EndState end = (EndState) auctionProcess.getNode("end");

  // the process instance
  ProcessInstance processInstance;
```

```
  // the main path of execution
  Token token;

  public void setUp() {
    // create a new process instance for the given process definition
    processInstance = new ProcessInstance(auctionProcess);

    // the main path of execution is the root token
    token = processInstance.getRootToken();
  }

  public void testMainScenario() {
    // after process instance creation, the main path of
    // execution is positioned in the start state.
    assertSame(start, token.getNode());

    token.signal();

    // after the signal, the main path of execution has
    // moved to the auction state
    assertSame(auction, token.getNode());

    token.signal();

    // after the signal, the main path of execution has
    // moved to the end state and the process has ended
    assertSame(end, token.getNode());
    assertTrue(processInstance.hasEnded());
  }
}
```

19.2. XML sources

Before you can start writing execution scenario's, you need a **`ProcessDefinition`**. The easiest way to get a **`ProcessDefinition`** object is by parsing xml. If you have code completion, type **`ProcessDefinition.parse`** and activate code completion. Then you get the various parsing methods. There are basically 3 ways to write xml that can be parsed to a **`ProcessDefinition`** object:

19.2.1. Parsing a process archive

A process archive is a zip file that contains the process xml in a file called **`processdefinition.xml`**. The jBPM process designer reads and writes process archives. For example:

```
static ProcessDefinition auctionProcess =
    ProcessDefinition.parseParResource("org/jbpm/tdd/auction.par");
```

19.2.2. Parsing an xml file

In other situations, you might want to write the processdefinition.xml file by hand and later package the zip file with e.g. an ant script. In that case, you can use the **`JpdlXmlReader`**

```
static ProcessDefinition auctionProcess =
    ProcessDefinition.parseXmlResource("org/jbpm/tdd/auction.xml");
```

19.2.3. Parsing an xml String

The simplest option is to parse the xml in the unit test inline from a plain String.

```
static ProcessDefinition auctionProcess =
    ProcessDefinition.parseXmlString(
  "<process-definition>" +
  "  <start-state name='start'>" +
  "    <transition to='auction'/>" +
  "  </start-state>" +
  "  <state name='auction'>" +
  "    <transition to='end'/>" +
  "  </state>" +
  "  <end-state name='end'/>" +
  "</process-definition>");
...
```

19.3. Testing sub processes

TODO (see test/java/org/jbpm/graph/exe/ProcessStateTest.java)

Pluggable architecture

The functionality of jBPM is split into modules. Each module has a definition and an execution (or runtime) part. The central module is the graph module, made up of the **`ProcessDefinition`** and the **`ProcessInstance`**. The process definition contains a graph and the process instance represents one execution of the graph. All other functions of jBPM are grouped into optional modules. Optional modules can extend the graph module with extra features like context (process variables), task management, timers, ...

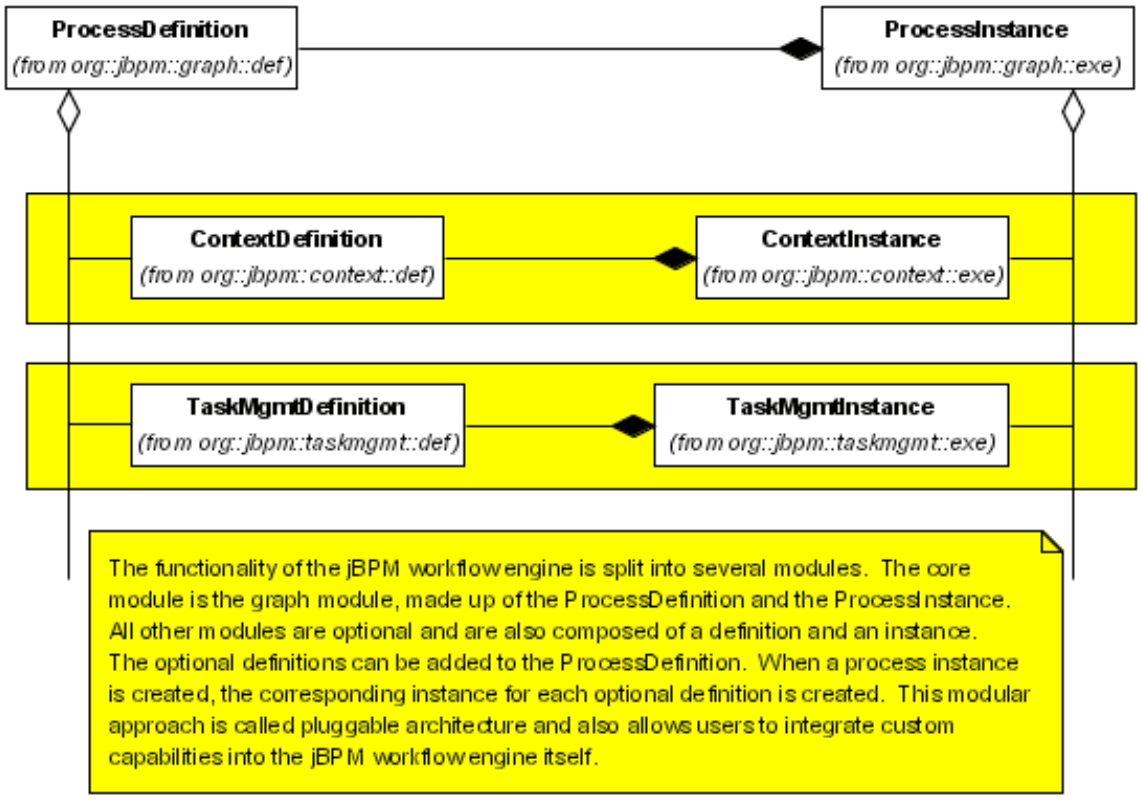

Figure 20.1. The pluggable architecture

The pluggable architecture in jBPM is also a unique mechanism to add custom capabilities to the jBPM engine. Custom process definition information can be added by adding a **`ModuleDefinition`** implementation to the process definition. When the **`ProcessInstance`** is created, it will create an instance for every **`ModuleDefinition`** in the **`ProcessDefinition`**. The **`ModuleDefinition`** is used as a factory for **`ModuleInstances`**.

The most integrated way to extend the process definition information is by adding the information to the process archive and implementing a **`ProcessArchiveParser`**. The **`ProcessArchiveParser`** can parse the information added to the process archive, create your custom **`ModuleDefinition`** and add it to the **`ProcessDefinition`**.

```
public interface ProcessArchiveParser {

  void writeToArchive(ProcessDefinition processDefinition, ProcessArchive
 archive);
```

```
   ProcessDefinition readFromArchive(ProcessArchive archive,
  ProcessDefinition processDefinition);

 }
```

To do its work, the custom **`ModuleInstance`** must be notified of relevant events during process execution. The custom **`ModuleDefinition`** might add **`ActionHandler`** implementations upon events in the process that serve as callback handlers for these process events.

Alternatively, a custom module might use AOP to bind the custom instance into the process execution. JBoss AOP is very well suited for this job since it is mature, easy to learn and also part of the JBoss stack.

Appendix A. Revision History

Revision History
Revision 1.0 Thu Sep 04 2008 JoshuaWulf*jwulf@redhat.com*
Converted to publican format

Index

F