JBoss Enterprise SOA Platform 4.3

JBPM Reference Guide

Your guide to using JBoss jBPM with the JBoss Enterprise SOA Platform 4.3 CP05

JBoss Enterprise SOA Platform 4.3 JBPM Reference Guide
Your guide to using JBoss jBPM with the JBoss Enterprise SOA Platform 4.3 CP05
Edition 4.3.5

Editor	Darrin Mison
Translator	Shigeaki Wakizaka
Translator	Takayoshi Osawa
Translator	Toshiya Kobayashi

Copyright © 2011 Red Hat, Inc.

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

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

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

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

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

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

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

All other trademarks are the property of their respective owners.

The JBPM jPDL 3.3 user guide for use with the JBoss SOA Platform

Preface vii
- 1. Document Conventions ... vii
  - 1.1. Typographic Conventions ... vii
  - 1.2. Pull-quote Conventions ... viii
  - 1.3. Notes and Warnings ... ix
- 2. We Need Feedback! ... ix

1. Introduction 1
- 1.1. Overview ... 1
- 1.2. The jPDL Suite ... 2
- 1.3. The jPDL Graphical Process Designer ... 3
- 1.4. The jBPM Console Web Application ... 3
- 1.5. The jBPM Core Library ... 3
- 1.6. The Identity Component ... 3

2. Tutorial 5
- 2.1. "Hello World" Example ... 5
- 2.2. Database Example ... 6
- 2.3. Contextual Example: Process Variables ... 9
- 2.4. Task Assignment Example ... 10
- 2.5. Example of a Custom Action ... 11

3. Configuration 15
- 3.1. Customizing Factories ... 18
- 3.2. Configuration Properties ... 18
- 3.3. Other Configuration Files ... 19
- 3.4. Logging Optimistic Concurrency Exceptions ... 19
- 3.5. Configuring the Job Executor ... 20
- 3.6. Object Factory ... 21

4. Persistence 25
- 4.1. The Persistence Application Programming Interface ... 25
  - 4.1.1. Relationship with the Configuration Framework ... 25
  - 4.1.2. Convenience Methods on JbpmContext ... 26
- 4.2. Configuring the Persistence Service ... 29
  - 4.2.1. Database Compatibility ... 29
  - 4.2.2. Programmatic Database Schema Operations ... 30
  - 4.2.3. The jBPM Schema Ant Task ... 30
  - 4.2.4. Combining Hibernate Classes ... 31

5. Java EE Application Server Facilities 33
- 5.1. Enterprise Beans ... 33

6. Process Modeling 37
- 6.1. Some Helpful Definitions ... 37
- 6.2. Process Graph ... 37
- 6.3. Nodes ... 39
  - 6.3.1. Node Responsibilities ... 39
  - 6.3.2. Node Type: Task Node ... 40
  - 6.3.3. Node Type: State ... 40
  - 6.3.4. Node Type: Decision ... 40
  - 6.3.5. Node Type: Fork ... 40
  - 6.3.6. Node Type: Join ... 40
  - 6.3.7. Node Type: Node ... 41
- 6.4. Transitions ... 41
- 6.5. Actions ... 41
  - 6.5.1. Action References ... 42

6.5.2. Events 43
6.5.3. Passing On Events 43
6.5.4. Scripts 43
6.5.5. Custom Events 44
6.6. Super-States 44
6.6.1. Super-State Transitions 44
6.6.2. Super-State Events 44
6.6.3. Hierarchical Names 45
6.7. Exception Handling 45
6.8. Process Composition 46
6.9. Custom Node Behavior 46
6.10. Graph Execution 47
6.11. Transaction Demarcation 48

7. The Context **51**
7.1. Accessing Process Variables 51
7.2. Lifes of Variables 51
7.3. Variable Persistence 52
7.4. Variable Scopes 52
7.4.1. Variable Overloading 52
7.4.2. Variable Over-Riding 52
7.4.3. Task Instance Variable Scope 52
7.5. Transient Variables 52

8. Task Management **55**
8.1. Tasks 55
8.2. Task instances 55
8.2.1. Task instance life-cycle 55
8.2.2. Task instances and graph execution 56
8.3. Assignment 57
8.3.1. Assignment interfaces 57
8.3.2. The assignment data model 57
8.3.3. The personal task list 58
8.3.4. The group task list 58
8.4. Task instance variables 59
8.5. Task controllers 59
8.6. Swimlanes 61
8.7. Swimlane in start task 61
8.8. Task events 61
8.9. Task timers 62
8.10. Customizing task instances 62
8.11. The identity component 62
8.11.1. The identity model 63
8.11.2. Assignment expressions 63
8.11.3. Removing the identity component 64

9. Scheduler **65**
9.1. Timers 65
9.2. Scheduler Deployment 65

10. Asynchronous Continuations **67**
10.1. The Concept 67
10.2. An Example 67
10.3. The Job Executor 70
10.4. jBPM's built-in asynchronous messaging 72

11. Business Calendar **75**
11.1. Due Date ... 75
11.1.1. Duration ... 75
11.1.2. Base Date ... 75
11.1.3. Due Date Examples ... 75
11.2. Calendar Configuration ... 76
11.3. Examples ... 76

12. E. Mail Support **79**
12.1. Mail in JPDL ... 79
12.1.1. Mail Action ... 79
12.1.2. Mail Node ... 80
12.1.3. "Task Assigned" E. Mail ... 80
12.1.4. "Task Reminder" E. Mail ... 80
12.2. Expressions in Mail ... 81
12.3. Specifying E. Mail Recipients ... 81
12.3.1. Multiple Recipients ... 81
12.3.2. Sending E. Mail to a BCC Address ... 81
12.3.3. Address Resolving ... 81
12.4. E. Mail Templates ... 82
12.5. Mail Server Configuration ... 83
12.6. "From" Address Configuration ... 83
12.7. Customizing E. Mail Support ... 83

13. Logging **85**
13.1. Log Creation ... 85
13.2. Log Configurations ... 86
13.3. Log Retrieval ... 87

14. jBPM Process Definition Language (JPDL) **89**
14.1. The process archive ... 89
14.1.1. Deploying a process archive ... 89
14.1.2. Process versioning ... 89
14.1.3. Changing deployed process definitions ... 90
14.1.4. Migrating process instances ... 90
14.2. Delegation ... 91
14.2.1. The jBPM class loader ... 91
14.2.2. The process class loader ... 91
14.2.3. Configuration of delegations ... 91
14.3. Expressions ... 93
14.4. jPDL XML Schema ... 93
14.4.1. Validation ... 93
14.4.2. process-definition ... 94
14.4.3. node ... 94
14.4.4. common node elements ... 94
14.4.5. start-state ... 95
14.4.6. end-state ... 95
14.4.7. state ... 96
14.4.8. task-node ... 96
14.4.9. process-state ... 96
14.4.10. super-state ... 97
14.4.11. fork ... 97
14.4.12. join ... 97
14.4.13. decision ... 97
14.4.14. event ... 98

14.4.15. transition ... 98
14.4.16. action ... 98
14.4.17. script ... 99
14.4.18. expression ... 100
14.4.19. variable ... 100
14.4.20. handler ... 100
14.4.21. timer ... 101
14.4.22. create-timer ... 101
14.4.23. cancel-timer ... 102
14.4.24. task ... 102
14.4.25. swimlane ... 103
14.4.26. assignment ... 103
14.4.27. controller ... 104
14.4.28. sub-process ... 104
14.4.29. condition ... 105
14.4.30. exception-handler ... 105

15. Test Driven Development for Workflow **107**
15.1. Introducing TDD for Workflow ... 107
15.2. XML Sources ... 108
15.2.1. Parsing a process archive ... 108
15.2.2. Parsing an XML file ... 108
15.2.3. Parsing an XML String ... 108

A. Revision History **111**

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 keycaps 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 keycap, all presented in mono-spaced bold and all distinguishable thanks to context.

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

> Press **`Enter`** to execute the command.
>
> Press **`Ctrl`**+**`Alt`**+**`F2`** to switch to the first virtual terminal. Press **`Ctrl`**+**`Alt`**+**`F1`** to return to your X-Windows session.

The first paragraph highlights the particular keycap to press. The second highlights two key combinations (each a set of three keycaps with 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; dialog box text; labeled buttons; check-box and radio button labels; menu titles and sub-menu titles. For example:

> 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

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

> **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.

`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:

> Publican is a *DocBook* publishing system.

1.2. Pull-quote Conventions

Terminal output and source code listings 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 add syntax highlighting 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

Notes are tips, shortcuts or alternative approaches 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 a box labeled 'Important' will not cause data loss but may cause irritation and frustration.

Warning

Warnings 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 Enterprise SOA Platform.**

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

This *Guide* has been written for developers and administrators. Read on in order to learn how to use jBPM and JPDL in your corporate setting. Note that this book not only teaches how to use the software but explains, in significant detail, how it works.

Note

This *Guide* contains a lot of terminology. Definitions for the key terms can be found in *Section 6.1, “ Some Helpful Definitions ”*.

The JBoss *Business Process Manager* (jBPM) is a flexible and extensible scaffolding for process languages. The *jBPM Process Definition Language* (JPDL) is one of the *process languages* that is built on top of this framework. It is an intuitive language, designed to enable the user to express business processes graphically. It does so by representing *tasks*, *wait states* (for asynchronous communication), *timers* and automated *actions*. To bind these operations together, the language has a powerful and extensible *control flow mechanism*.

The JPDL has few dependencies, making it is as easy to install as a Java library. To do so, deploy it on a *J2EE clustered application server*. One will find it particularly useful in environments in which extreme throughput is a crucial requirement.

Note

The JPDL can be configured for use with any database. It can also be deployed on any application server.

1.1. Overview

Read this section to gain an overview of the way in which the jBPM works.

The core workflow and business process management functionality is packaged in the form of a simple Java library. This library includes a service that manages and executes jPDL database processes.

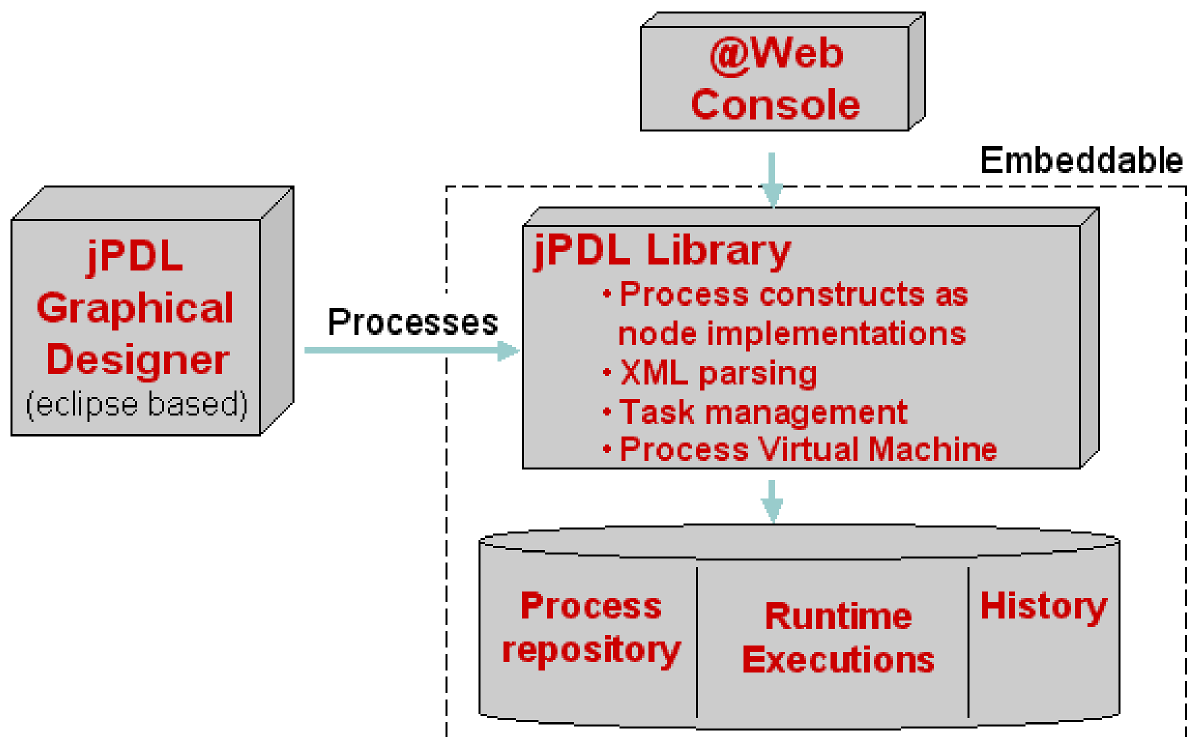


Figure 1.1. Overview of the jPDL components

1.2. The jPDL Suite

This suite contains all of the jBPM components and the following sub-directories:

- config
- database
- deploy
- designer
- examples
- lib
- src

The **JBoss Application Server** consists of the following components:

The jBPM Web Console
: This is packaged as a web archive. Both *process participants* and jBPM administrators can use this console.

The Job Executor
: This is part of the **Console Web Application**. It is launched by a *servlet*, then spawns a *thread pool*, whose job it is to monitor and executes timers and asynchronous messages.

The jBPM Tables
: These are contained in the default **Hypersonic** database. (It already contains a process.)

An Example Process
: One example process is already deployed to the jBPM database.

Identity Component
: The identity component libraries are part of the **Console Web Application**. It owns those tables found in the database which have the **`JBPM_ID_`** prefix.

1.3. The jPDL Graphical Process Designer

The jPDL also includes the **Graphical Process Designer Tool**. Use it to design business processes. (It is an **Eclipse** plug-in and is included with the **JBoss Developer Studio** product.)

It facilitates a smooth transition from business process modeling to practical implementation, making it of use to both the business analyst and the technical developer.

1.4. The jBPM Console Web Application

The **Console Web Application** serves three purposes. Firstly, it functions as a central user interface, allowing one to interact with those run-time tasks that have been generated by the process executions. Secondly, it is an administrative and monitoring console that allows one to inspect and manipulate run-time instances. The third role of this software is that of business activity monitor. In this role, it presents statistics about the execution of processes. This information is of use to managers seeking to optimize performance as it allows them to find and eliminate bottlenecks.

1.5. The jBPM Core Library

The Business Process Manager has two core components. These are the "plain Java" (J2SE) library, which manages process definitions, and the run-time environment, which executes process instances.

The jBPM, itself, is a Java library. Consequently, it can be used in any Java environment, be it a web or **Swing** application, an `Enterprise Java Bean` or a web service.

One can also package and expose the jBPM library as a *stateless session* `Enterprise Java Bean`. Do this if there is a need to create a clustered deployment or provide scalability for extremely high throughput. (The stateless session `Enterprise Java Bean` adheres to the **`J2EE 1.3`** specifications, mearning that it can be deployed on any application server.)

Be aware that some parts of the **`jbpm-jpdl.jar`** file are dependent upon third-party libraries such as **Hibernate** and **Dom4J**.

Hibernate provides the jBPM with *persistence* functionality. Also, apart from providing traditional *O/R mapping*, **Hibernate** resolves the differences between the Structured Query Language dialects used by competing databases. This ability makes the jBPM highly portable.

The Business Process Manager's application programming interface can be accessed from any custom Java code in your project, whether it be a web application, an Enterprise Java Bean, a web service component or a message-driven bean.

1.6. The Identity Component

The jBPM can integrate with any company directory that contains user (and other organizational) data. (For those projects for which no organizational information component is available, use the *Identity Component*. This component has a "richer" model than those used by traditional servlets, Enterprise Java Beans and portlets.)

Note

Read *Section 8.11, “The identity component”* to learn more about this topic.

Having read this chapter, you have gained a broad overview of the jBPM and its constituent components.

Tutorial

Study the following tutorial to learn how to use basic *process constructs* in the JPDL. The tutorial also demonstrates ways in which to manage run-time executions via the application programming interface.

Each of the extensively-commented examples in the tutorial can be found in the jBPM download package, located in the **`src/java.examples`** sub-directory.

Note

Red Hat recommends creating a project at this point. One can then freely experiment and create variations of each of the examples in turn.

2.1. "Hello World" Example

A *process definition* is a *directed graph*, made up of nodes and transitions. The `Hello World` process definition has three of these nodes. (It is best to learn how the pieces fit together by studying this simple process without using the **Designer Tool**.) The following diagram presents a graphical representation of the `Hello World` process:

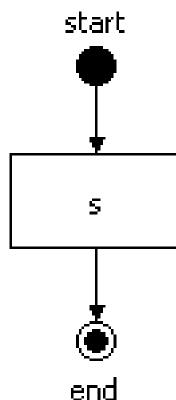


Figure 2.1. The Hello World Process Graph

```
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 jBPM's basic features is the ability to make the execution of database processes persist whilst they are in a `wait state`. The next example demonstrates this ability, storing a process instance in the jBPM database.

It works by creating separate `methods` for different pieces of user code. For instance, a piece of user code in a web application starts a process and "persists" the execution in the database. Later, a message-driven bean loads that process instance and resumes the execution of it.

Note

More information about jBPM persistence can be found in *Chapter 4, 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 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(0);

      // 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. Contextual Example: Process Variables

Whilst processes are executed, the context information is held in *process variables*. These are similar to **`java.util.Map`** classes, in that they map variable names to values, the latter being Java objects. (The process variables are "persisted" as part of the process instance.)

Note

In order to keep the following example simple, only the application programming interface that is needed to work with variables is shown (without any persistence functionality.)

Note

Find out more about variables by reading *Chapter 7, The 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(500));
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(500),
             contextInstance.getVariable("amount"));
assertEquals("i met my deadline",
             contextInstance.getVariable("reason"));
```

2.4. Task Assignment Example

The next example demonstrates how to assign a task to a user. Because of the separation between the jBPM workflow engine and the organizational model, expression languages will always be too limited to use to calculate actors. Instead, specify an implementation of **`AssignmentHandler`** and use it to include 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. Example of a Custom Action

Actions are mechanisms designed to bind custom Java code to jBPM processes. They can be associated with their own nodes (if these are relevant to the graphical representation of the process.) Alternatively, actions can be "placed on" events (for instance, when taking a transition, or entering or leaving a node.) If they are placed on events, the actions are not treated as part of the graphical representation (but they are still run when the events are "fired" during a run-time process execution.)

Firstly, look at the action handler implementation to be used in the next example: `MyActionHandler`. It is not particularly impressive of itself: it merely sets the Boolean variable `isExecuted` to **`true`**. Note that this variable is static so one can access it from within the action handler (and from the action itself) to verify its value.

Note

More information about "actions" can be found in *Section 6.5, "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;
  }
}
```

Important

Prior to each test, 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;
  }
```

The first example illustrates 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 now being placed on both the `enter-node` and `leave-node` events. Note that a node has more than one event type. This is in contrast to a *transition*, which has only one event. Hence, when placing actions on a node, always put them 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);
```

Configuration

Read this chapter and studied the examples to learn how to configure the jBPM.

The simplest way to configure the Business Process Manager is by putting the **`jbpm.cfg.xml`** configuration file into the root of the classpath. If the file is not available for use as a resource, the default minimal configuration will be used instead. This minimal configuration is included in the jBPM library (**`org/jbpm/default.jbpm.cfg.xml`**.) If a jBPM configuration file is provided, the values it contains will be used as the defaults. Hence, one only needs to specify the values that are to be different from those in the default configuration file.

The jBPM configuration is represented by a Java class called **`org.jbpm.JbpmConfiguration`**. Obtain it by making use of the `singleton` instance method (`JbpmConfiguration.getInstance()`.)

Note

Use the `JbpmConfiguration.parseXxxx` methods to load a configuration from another source.

```
static JbpmConfinguration jbpmConfiguration =
 JbpmConfinguration.parseResource("my.jbpm.cfg.xml");
```

The **`JbpmConfiguration`** is "thread safe" and, hence, can be kept in a *static member*.

Every thread can use a **`JbpmConfiguration`** as a *factory* for **`JbpmContext`** objects. A **`JbpmContext`** will usually represent one transaction. They make services available inside *context blocks* which 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 both a set of services and the configuration settings available to the Business Process Manager. The services are configured by the values in the **`jbpm.cfg.xml`** file. They make it possible for the jBPM to run in any Java environment, using whatever services are available within said environment.

Here are the default configuration settings for the **`JbpmContext`**:

```
<jbpm-configuration>

<jbpm-context>
    <service name='persistence'
      factory='org.jbpm.persistence.db.DbPersistenceServiceFactory' />
    <service name='message'
      factory='org.jbpm.msg.db.DbMessageServiceFactory' />
    <service name='scheduler'
      factory='org.jbpm.scheduler.db.DbSchedulerServiceFactory' />
    <service name='logging'
```

```
      factory='org.jbpm.logging.db.DbLoggingServiceFactory' />
    <service name='authentication'
      factory=
'org.jbpm.security.authentication.DefaultAuthenticationServiceFactory' />
</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' />
  <string name='resource.mail.templates'
    value='jbpm.mail.templates.xml' />

  <int name='jbpm.byte.block.size' value="1024" singleton="true" />
  <bean name='jbpm.task.instance.factory'
    class='org.jbpm.taskmgmt.impl.DefaultTaskInstanceFactoryImpl'
    singleton='true' />

  <bean name='jbpm.variable.resolver'
    class='org.jbpm.jpdl.el.impl.JbpmVariableResolver'
    singleton='true' />

    <string name='jbpm.mail.smtp.host' value='localhost' />

    <bean name='jbpm.mail.address.resolver'
        class='org.jbpm.identity.mail.IdentityAddressResolver'
        singleton='true' />
    <string name='jbpm.mail.from.address' value='jbpm@noreply' />

    <bean name='jbpm.job.executor'
        class='org.jbpm.job.executor.JobExecutor'>
      <field name='jbpmConfiguration'><ref bean='jbpmConfiguration' />
      </field>
      <field name='name'><string value='JbpmJobExecutor' /></field>
      <field name='nbrOfThreads'><int value='1' /></field>
      <field name='idleInterval'><int value='60000' /></field>
      <field name='retryInterval'><int value='4000' /></field>
      <!-- 1 hour -->
      <field name='maxIdleInterval'><int value='3600000' /></field>
      <field name='historyMaxSize'><int value='20' /></field>
      <!-- 10 minutes -->
      <field name='maxLockTime'><int value='600000' /></field>
      <!-- 1 minute -->
      <field name='lockMonitorInterval'><int value='60000' /></field>
       <!-- 5 seconds -->
      <field name='lockBufferTime'><int value='5000' /></field>
    </bean>
</jbpm-configuration>
```

The above file contains three parts:

1. a set of *service implementations* which configure the **`JbpmContext`**. (The possible configuration options are detailed in the chapters that cover specific service implementations.)

2. a series of references to configuration resources. If one wishes to customize one of the configuration files, update these mappings. To do so, always back up the default configuration file (**`jbpm-jpdl.jar`**) to another location on the classpath first. Then, update the reference in this file, pointing it to the customized version that the jBPM is to use.

3. miscellaneous configurations for use by the jBPM. (These are described in the chapters that cover the specific topics in question.)

The default configuration has been optimized for a simple web application environment which has minimal dependencies. The persistence service obtains a JDBC connection which is used by all of the other services. Hence, all of the workflow operations are centralized as they are placed in a single transaction on a JDBC connection (without the need for a transaction manager.)

`JbpmContext` contains *convenience methods* for most of the common process operations. They are demonstrated in this code sample:

```
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

There is no need to call any of the save methods explicitly because the `XxxForUpdate` methods are designed to register the loaded object for "auto-save."

It is possible to specify multiple **`jbpm-context`**s. To do so, make sure that each of them is given a unique name attribute. (Retrieve named contexts by using `JbpmConfiguration.createContext(String name);`.)

A service element specifies its own name and associated *service factory*. The service will only be created when requested to do so by `JbpmContext.getServices().getService(String name)`.

Note

One can also specfy the `factories` as *elements* instead of attributes. This is necessary when injecting some configuration information into factory objects.

Note that the component responsible for creating and wiring the objects and parsing the XML is called the **`object factory`**.

3.1. Customizing Factories

Warning

A mistake commonly made by people when they are trying to customize factories is to mix long and short notation together. (Examples of the short notation can be seen in the default configuration file.)

Hibernate logs `StateObjectStateException` exceptions and generates a `stack trace`. In order to remove the latter, set **`org.hibernate.event.def.AbstractFlushingEventListener`** to **`FATAL`**. (Alternatively, if using `log4j`, set the following line in the configuration: for that: **`log4j.logger.org.hibernate.event.def.AbstractFlushingEventListener=FATAL`**

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

Important

If one needs to note specific properties on a service, only the long notation can be used.

```
<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>
```

3.2. Configuration Properties

jbpm.byte.block.size

File attachments and binary variables are stored in the database in the form of a list of fixed-sized, binary objects. (The aim of this is to improve portability amongst different databases. It also allows one to embed the jBPM more easily.) This parameter controls the size of those fixed-length chunks.

jbpm.task.instance.factory

To customize the way in which task instances are created, specify a fully-qualified classname against this property. (This is often necessary when one intends to customize, and add new properties to, the **`TaskInstance`** bean.) Ensure that the specified classname implements the `org.jbpm.taskmgmt.TaskInstanceFactory` interface. (Refer to *Section 8.10, “Customizing task instances”* for more information.)

jbpm.variable.resolver
: Use this to customize the way in which jBPM looks for the first term in "JSF"-like expressions.

3.3. Other Configuration Files

There are a number of configuration files in the jBPM which can be customized:

`hibernate.cfg.xml`
: This contains references to, and configuration details for, the **Hibernate** mapping resource files.

  To specify a different file, configure the jbpm.hibernate.cfg.xml property in **`jbpm.properties`**. (The default **Hibernate** configuration file is located in the **`src/config.files/hibernate.cfg.xml`** sub-directory.)

`org/jbpm/db/hibernate.queries.hbm.xml`
: This file contains those **Hibernate** queries to be used in the jBPM sessions (**`org.jbpm.db.*Session`**.)

`org/jbpm/graph/node/node.types.xml`
: This file is used to map XML node elements to **`Node`** implementation classes.

`org/jbpm/graph/action/action.types.xml`
: This file is used to map XML action elements to **`Action`** implementation classes.

`org/jbpm/calendar/jbpm.business.calendar.properties`
: This contains the definitions of "business hours" and "free time."

`org/jbpm/context/exe/jbpm.varmapping.xml`
: This specifies the way in which the process variables values (Java objects) are converted to variable instances for storage in the jBPM database.

`org/jbpm/db/hibernate/jbpm.converter.properties`
: This specifies the **`id-to-classname`** mappings. The ids are stored in the database. The **`org.jbpm.db.hibernate.ConverterEnumType`** class is used to map the identifiers to the `singleton` objects.

`org/jbpm/graph/def/jbpm.default.modules.properties`
: This specifies which modules are to be added to a new **`ProcessDefinition`** by default.

`org/jbpm/jpdl/par/jbpm.parsers.xml`
: This specifies the phases of *process archive parsing*.

3.4. Logging Optimistic Concurrency Exceptions

When it is run in a cluster configuration, the jBPM synchronizes with the database by using *optimistic locking*. This means that each operation is performed in a transaction and if, at the end, a collision is detected, then the transaction in question is rolled back and has to be handled with a retry. This can cause `org.hibernate.StateObjectStateException` exceptions. If and when this happens, **Hibernate** will log the exceptions with a simple message,

```
optimistic locking
            failed
```

.

Hibernate can also log the `StateObjectStateException` with a stack trace. To remove these stack traces, set the **`org.hibernate.event.def.AbstractFlushingEventListener`** class to **`FATAL`**. Do so in **log4j** by using the following configuration:

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

In order to log jBPM stack traces, add the following line to the **`jbpm.cfg.xml`** file:

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

3.5. Configuring the Job Executor

The `job executor` exposes a few properties that one can tune. To change any of the values described below, edit the **`jbpm.job.executor`** bean (found in the **`jbpm.cfg.xml`** file.)

Table 3.1. Properties of the Job Executor

Property	Description	Default
nbrOfThreads	Size of the job executor thread pool	1
idleInterval	Period between checks for new jobs (milliseconds)	5 seconds
maxIdleInterval	When a job fails, the affected thread pauses for a period initially equal to **`idleInterval`**, which is increased twofold until it reaches **`maxIdleInterval`** (milliseconds)	1 hour
maxLockTime	Amount of time a job executor thread is allowed to hold a job before the job is released and offered to other threads (milliseconds)	10 minutes
lockMonitorInterval	Period between checks for job lock times (milliseconds)	1 minute

In addition to the job executor bean properties, you can indicate the number of times a failed job is retried. Set the jbpm.job.retries configuration entry to the desired value. The default is **3**.

```
<int name="jbpm.job.retries" value="3" />
```

Warning

Setting the retry count to a low value may cause process instances to get jam, whereas a high value causes jobs with unrecoverable exceptions (for instance, database connectivity problems) to be unduly reattempted.

Note

Alternative implementations of the `Message and Timer Service`, (an example being the `JCA Inflow Service`, included with JBoss Enterprise Service Bus) also recognise the jbpm.job.retries configuration entry.

3.6. Object Factory

The *Object Factory* can build objects to the specification contained in a "beans-like" XML configuration file. This file dictates how objects are to be created, configured and wired together to form a complete object graph. Also use the Object Factory to inject configurations and other beans into a single bean.

In its most elementary form, the Object Factory is able to create both basic *types* and Java beans from such a configuration, as shown in the following examples:

```
<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" />
    ObjectFactory of = ObjectFactory.parseXmlFromAbove();
</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"));]]>
```

This code shows how to configure lists:

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

This code demonstrates how to configure 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>
```

Use *direct field injection* and property `setter` methods to configure beans:

```
<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, an integer or any other kind. Here is some code that demonstrates this capability:

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

Beans can be built with any constructor, as this code shows:

```
<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>
```

Beans can be constructed using a `factory` method:

```
<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>
```

Beans can be constructed using 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>
```

Use the attribute **`singleton="true"`** to mark each named object as a `singleton`. Doing so will ensure that a given **`object factory`** always returns the same object for each request.

Note

`Singletons` cannot be shared between different object factories.

The `singleton` feature causes differentiation between the methods named `getObject` and `getNewObject`. Normally, one should use `getNewObject` as this clears the **`object factory`**'s *object cache* before the new object graph is constructed.

During construction of the object graph, the *non-singleton objects* are stored in the **`object factory`**'s cache. This allows references to one object to be shared. Bear in mind that the `singleton object cache` is different from the `plain object cache`. The `singleton` cache is never cleared, whilst the plain one is cleared every time a `getNewObject` method is started.

Having studied this chapter, one now has a thorough knowledge of the many ways in which the jBPM can be configured.

Persistence

This chapter provides the reader with detailed insight into the Business Process Manager's "persistence" functionality.

Most of the time, the jBPM is used to execute processes that span several transactions. The main purpose of the *persistence* functionality is to store process executions when *wait states* occur. It is helpful to think of the process executions as *state machines*. The intention is to move the process execution state machine from one state to the next within a single transaction.

A process definition can be represented in any of three different forms, namely XML, Java object or a jBPM database record. (Run-time data and log information can also be represented in either of the latter two formats.)

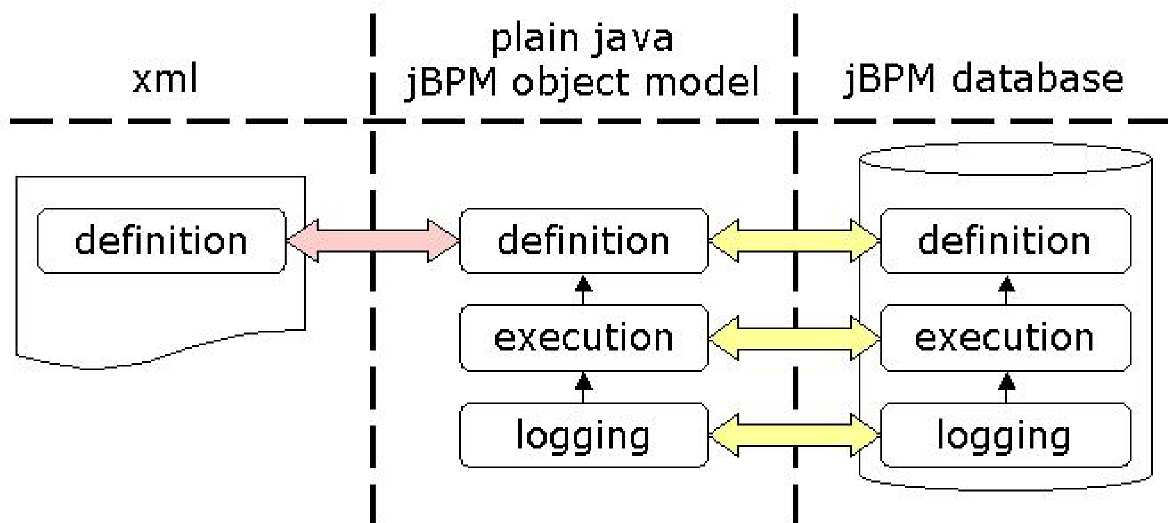


Figure 4.1. The Transformations and Different Forms

> **Note**
>
> To learn more about XML representations of process definitions and process archives, see *Chapter 14, jBPM Process Definition Language (JPDL)*.

> **Note**
>
> To learn more about how to deploy a process archive to the database, read *Section 14.1.1, "Deploying a process archive"*

4.1. The Persistence Application Programming Interface

4.1.1. Relationship with the Configuration Framework

The persistence application programming interface is integrated with the configuration framework, (see *Chapter 3, Configuration* .) This has been achieved by the exposure of some of the `convenience persistence` methods on the `JbpmContext`, allowing the jBPM `context block` to call persistence API operations.

```
JbpmContext jbpmContext = jbpmConfiguration.createJbpmContext();
try {
  // Invoke persistence operations here
} finally {
  jbpmContext.close();
}
```

4.1.2. Convenience Methods on JbpmContext

The three most commonly-performed persistence operations are:

1. process. deployment
2. new process execution commencement
3. process execution continuation

Process deployment is normally undertaken directly from the **Graphical Process Designer** or from the **`deployprocess` ant** task. However, to do it directly from Java, use this code:

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

Create a new process execution by specifying the process definition of which it will be an instance. The most common way to do this is by referring to the name of the process. The jBPM will then find the latest version of that process in the database. Here is some demonstration code:

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

To continue a process execution, fetch the process instance, the token or the **`taskInstance`** from the database and invoke some methods on the POJO (*Plain Old Java Object*) jBPM objects. Afterwards, save the updates made to the **`processInstance`** into the database.

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

Note that it is not necessary to explicitly invoke the `jbpmContext.save` method if the `ForUpdate` methods are used in the **`JbpmContext`** class. This is because the save process will run automatically when the **`jbpmContext`** class is closed. For example, one may wish to inform the jBPM that a

`taskInstance` has completed. This can cause an execution to continue, so the **`processInstance`** related to the **`taskInstance`** must be saved. The most convenient way to do this is by using the `loadTaskInstanceForUpdate` method:

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

Important

Read the following explanation to learn how the jBPM manages the persistence feature and uses **Hibernate**'s functionality.

The **`JbpmConfiguration`** maintains a set of `ServiceFactories`. They are configured via the **`jbpm.cfg.xml`** file and instantiated as they are needed.

The **`DbPersistenceServiceFactory`** is only instantiated the first time that it is needed. After that, `ServiceFactory`s are maintained in the **`JbpmConfiguration`**.

A **`DbPersistenceServiceFactory`** manages a **Hibernate** `ServiceFactory` but this is only instantiated the first time that it is requested.

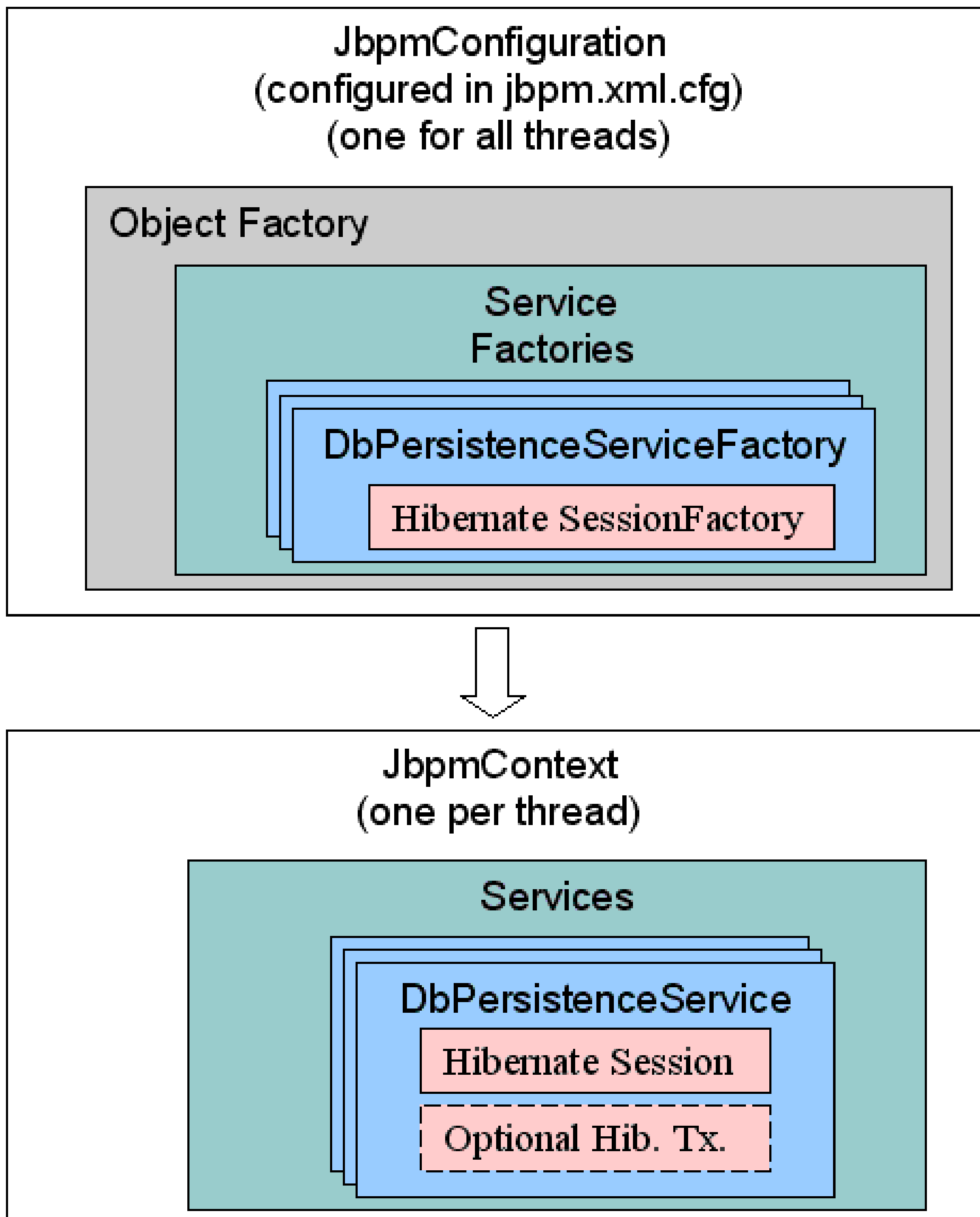


Figure 4.2. The Persistence-Related Classes

When the **`jbpmConfiguration.createJbpmContext()`** class is invoked, only the **`JbpmContext`** is created. No further persistence-related initializations occur at this time. The **`JbpmContext`** manages a **`DbPersistenceService`** class, which is instantiated when it is first requested. The **`DbPersistenceService`** class manages the **Hibernate** session, which is also only instantiated the first time it is required. (In other words, a **Hibernate** session will only be opened when the first operation that requires persistence is invoked.)

4.2. Configuring the Persistence Service

4.2.1. Database Compatibility

The jBPM runs on any database that is supported by **Hibernate**.

The example configuration file, `src/config.files`, specifies the use of the **Hypersonic** in-memory database, which is ideal for development and testing purposes. (**Hypersonic** retains all data in memory and does not store anything on disk.)

4.2.1.1. Isolation Level of the JDBC Connection

Set the database isolation level for the JDBC connection to at least `READ_COMMITTED`.

Warning

If it is set to `READ_UNCOMMITTED`, (isolation level zero, the only isolation level supported by **Hypersonic**), race conditions might occur in the `job executor` .These might also appear when synchronization of multiple tokens is occurring.

4.2.1.2. Changing the Database

In order to reconfigure Business Process Manger to use a different database, follow these steps:

- put the JDBC driver library archive in the classpath.
- update the **Hibernate** configuration used by jBPM.
- create a schema in the new database.

4.2.1.3. The Database Schema

The `jbpm.db` sub-project contains drivers, instructions and scripts to help the user to start using the database of his or her choice. Refer to the `readme.html` (found in the root of the `jbpm.db` project) for more information.

Note

Whilst the jBPM is capable of generating DDL scripts for any database, these schemas are not always optimized. Consider asking your corporation's Database Adminstrator to review the generated DDL, so that he or she can optimize the column types and indexes.

The following **Hibernate** configuration option may be of use in a development environment: set hibernate.hbm2ddl.auto to `create-drop` and the schema will be created automatically the first time the database is used in an application. When the application closes down, the schema will be dropped.

Note

This functionality can also be invoked programmatically with `jbpmConfiguration.createSchema()` and `jbpmConfiguration.dropSchema()`.

4.2.2. Programmatic Database Schema Operations

The Business Process Manager provides an application programming interface to use in order to create and drop database schemas. Access it through the **`org.jbpm.JbpmConfiguration`** methods, namely `createSchema` and `dropSchema`.

Note

Be aware that there is no constraint on invoking these methods other than those pertaining to the inherent privileges of the configured database user.

The aforementioned APIs constitute a "facade" to the broader functionality offered by the **`org.jbpm.db.JbpmSchema`** class. That functionality consists of the following options:

- Create, drop, update and clean (drop-create) the database schema
- Generate SQL scripts for the above operations
- List the mapped tables and query the existing tables in the database

Note

This function exists because, in production environments it is extremely wise not to grant table creation privileges to the database login account that runs the application. With this unavailable, the `JbpmSchema` API and the **`Ant task`** provide the means to create the schema.

4.2.3. The jBPM Schema Ant Task

As an alternative to programmatic schema manipulation, the jBPM provides an **`Ant`** task that generates SQL scripts. These scripts can be used to create, drop and update the database schema. The listing depicted below illustrates the task that generates the schema creation script and then saves it to the **`create.sql`** file. (The **Hibernate** configuration is read from the **`hibernate.cfg.xml`** resource.)

```
<taskdef name="jbpmschema" classname="org.jbpm.ant.JbpmSchemaTask">
  <classpath>
    <pathelement location="jbpm-jpdl.jar" />
    <pathelement location="hibernate.jar" />
    <pathelement location="dom4j.jar" />
    <pathelement location="commons-logging.jar"/>
    <pathelement location="commons-collections.jar"/>
  </classpath>
</taskdef>

<jbpmschema config="hibernate.cfg.xml" action="create" output="create.sql" />
```

These are the task parameters:

Table 4.1. jBPM Schema Task Parameters

Attribute	Description	Required
config	Hibernate configuration resource	No, default **`hibernate.cfg.xml`**
properties	Hibernate properties resource. These properties override property values from the `config` resource.	No
action	Database schema operation to script. Can be `create`, `drop`, `update` or `clean`.	No, default `create`
output	The output file. The generated script is written to this file.	Yes
delimiter	String that separates SQL statements	No, default `;`
delimiterType	Controls whether the delimiter should be placed on a line by itself. Can be `normal`, at the end of line or `row`, on a line by itself.	No, default `normal`

Note

The JbpmSchemaTask is packaged in **`jbpm-jpdl.jar`**. No script needs to be supplied: efine and run this task in your own scripts, and base any customisations on the sample snippet.

4.2.4. Combining Hibernate Classes

Combining **Hibernate** and jBPM persistent classes brings about two major benefits. Firstly, session, connection and transaction management become easier because, by combining them into one **Hibernate** session factory, there will be only one **Hibernate** session and one JDBC connection. Hence, the jBPM updates will be in the same transaction as the updates for the domain model. This eliminates the need for a transaction manager.

Secondly, it enables one to drop one's **Hibernate** persistence object into the process variables without any additional work.

To make this occur, create one central **`hibernate.cfg.xml`** file. It is easiest to use the default jBPM **`hibernate.cfg.xml`** as a starting point and add references to one's own **Hibernate** mapping files to customize it.

Java EE Application Server Facilities

Read this chapter to learn about the facilities offered by the jBPM to that can be used to leverage the Java EE infrastructure.

5.1. Enterprise Beans

The `CommandServiceBean` is a *stateless session bean* that runs Business Process Manager commands by calling its `execute` method within a separate jBPM context. The available environment entries and customizable resources are summarized in the following table:

Table 5.1. Command Service Bean Environment

Name	Type	Description
`JbpmCfgResource`	Environment Entry	This the classpath resource from which the jBPM configuration is read. Optional, defaults to **`jbpm.cfg.xml`**.
`ejb/TimerEntityBean`	EJB Reference	This is a link to the local entity bean that implements the scheduler service. Required for processes that contain timers.
`jdbc/JbpmDataSource`	Resource Manager Reference	This is the logical name of the data source that provides JDBC connections to the jBPM persistence service. Must match the hibernate.connection.datasource property in the Hibernate configuration file.
`jms/JbpmConnectionFactory`	Resource Manager Reference	This is the logical name of the factory that provides JMS connections to the jBPM message service. Required for processes that contain asynchronous continuations.
`jms/JobQueue`	Message Destination Reference	The jBPM message service sends job messages to this queue. To ensure this is the same queue from which the job listener bean receives messages, the `message-destination-link` points to a common logical destination, `JobQueue`.
`jms/CommandQueue`	Message Destination Reference	The command listener bean receives messages from this queue. To ensure this is the same queue to which command messages can be sent, the `message-destination-link element` points to a common logical destination, `CommandQueue`.

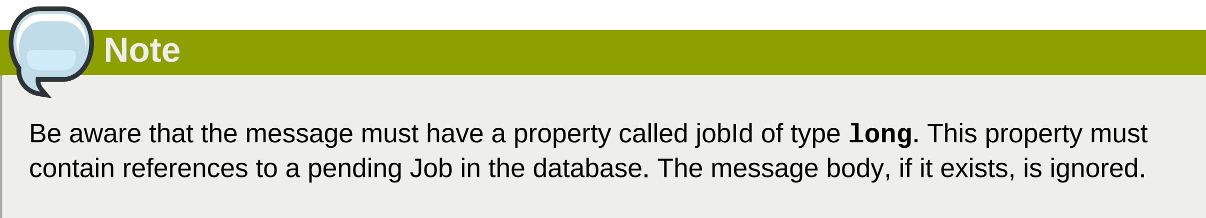
The `CommandListenerBean` is a message-driven bean that listens to the `CommandQueue` for command messages. It delegates command execution to the `CommandServiceBean`.

The body of the message must be a Java object that can implement the `org.jbpm.Command` interface. (The message properties, if any, are ignored.) If the message is not of the expected format, it is forwarded to the `DeadLetterQueue` and will not be processed any further. The message will also be rejected if the destination reference is absent.

If a received message specifies a **`replyTo`** destination, the command execution result will be wrapped in an `object message` and sent there.

The `command connection factory environment reference` points to the resource manager being used to supply Java Message Service connections.

Conversely, `JobListenerBean` is a message-driven bean that listens to the `JbpmJobQueue` for job messages, in order to support *asynchronous continuations*.



Note

Be aware that the message must have a property called jobId of type **`long`**. This property must contain references to a pending Job in the database. The message body, if it exists, is ignored.

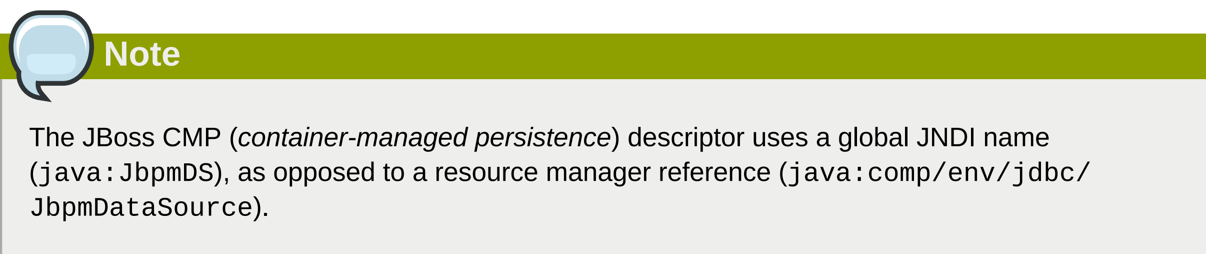
This bean extends the **`CommandListenerBean`**. It inherits the latter's environmental entries and those resource references that can be customized.

Table 5.2. Command/Job listener bean environment

Name	Type	Description
`ejb/LocalCommandServiceBean`	EJB Reference	This is a link to the local session bean that executes commands on a separate jBPM context.
`jms/JbpmConnectionFactory`	Resource Manager Reference	This is the logical name of the factory that provides Java Message Service connections for producing result messages. Required for command messages that indicate a reply destination.
`jms/DeadLetterQueue`	Message Destination Reference	Messages which do not contain a command are sent to the queue referenced here. It is optional. If it is absent, such messages are rejected, which may cause the container to redeliver.

The `TimerEntityBean` is used by the *Enterprise Java Bean timer service* for scheduling. When the bean expires, timer execution is delegated to the `command service` bean.

The `TimerEntityBean` requires access to the Business Process Manager's data source. The Enterprise Java Bean deployment descriptor does not define how an entity bean is to map to a database. (This is left to the container provider.) In the **JBoss Application Server**, the jbosscmp-jdbc.xml descriptor defines the data source's JNDI name and relational mapping data (such as the table and column names.)



Note

The JBoss CMP (*container-managed persistence*) descriptor uses a global JNDI name (`java:JbpmDS`), as opposed to a resource manager reference (`java:comp/env/jdbc/JbpmDataSource`).

Note

Earlier versions of the Business Process Manager used a stateless session bean called `TimerServiceBean` to interact with the Enterprise Java Bean timer service. The session approach had to be abandoned because it caused an unavoidable bottleneck for the `cancellation` methods. Because session beans have no identity, the timer service was forced to iterate through *all* the timers to find the ones it had to cancel.

The bean is still available for backwards compatibility purposes. It works in the same environment as the `TimerEntityBean`, so migration is easy.

Table 5.3. Timer Entity/Service Bean Environment

Name	Type	Description
`ejb/LocalCommandServiceBean`	EJB Reference	This is a link to the local session bean that executes timers on a separate jBPM context.

Having studied this chapter, you should now have a thorough understanding of the facilities offered by the jBPM that can be used to leverage the Java EE infrastructure and should be comfortable with testing some of these in your corporate environment.

Process Modeling

6.1. Some Helpful Definitions

Read this section to learn the terminology that you will find used throughout the rest of this book.

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 node type defines the run-time behavior. A process definition only has 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 generated 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 to execute the graph. When it receives 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 making the graph execution continue. Continuation of graph execution is achieved by making the token leave the node. Each type of node can implement a different behavior for the continuation of the graph execution. A node that does not pass on the execution will behave as a *state*.

Actions are pieces of Java code that are executed upon events during the process execution. The *graph* is an important instrument in the communication of software requirements but it is just one view (*projection*) of the software being produced. It hides many technical details. Actions are a mechanism one uses to add technical details beyond those 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`.

Having learned these definitions, read on to find out how process modelling works.

6.2. Process Graph

A process definition is a graph that is made up of nodes and transitions. This information is expressed in XML and found in a file called **`processdefinition.xml`**. Each node must have a *type* (examples being `state`, `decision`, `fork` and `join`.) Each node has a set of *leaving transitions*. Names can be given to the transitions that leave a node in order to make them distinct from each other. For example, the following diagram shows a process graph for an auction process.

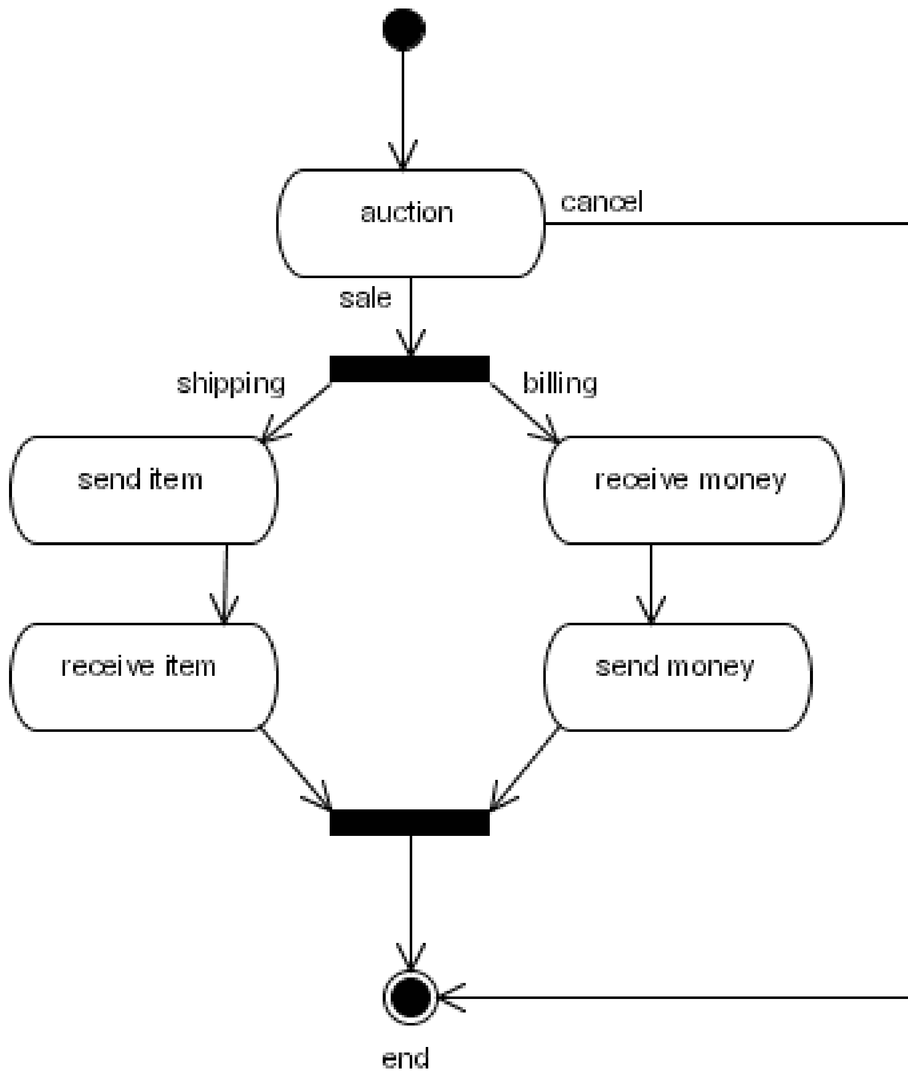


Figure 6.1. The auction process graph

Below is the process graph for the same auction process, albeit represented in XML this time:

```
<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>
```

6.3. Nodes

A process graph is made up of nodes and transitions. Each node is of a specific type. The node type determines what will happen when an execution arrives in the node at run-time. The Business Process Manager provides a set of node types to use. Alternatively, one can write custom code to implement a specific node behavior.

6.3.1. Node Responsibilities

Each node has two main responsibilities: firstly, it can execute plain Java code, code which will normally relate to the function of the node. Its second responsibility is to pass on the process execution.

A node may face the following options when it attempts to pass the process execution on. It will follow that course which is most applicable:

1. it can not propagate the execution. (The node behaves as a `wait state`.)
2. it can propagate the execution over one of the node's `leaving transitions`. (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 automatically in the sense that it will execute some custom programming logic and then continue the process execution automatically without waiting.
3. a node can "decide" to create new tokens, each of which will represent a new path of execution. Each of these new tokens can be launched over the node's `leaving transitions`. A good example of this kind of behavior is the `fork node`.
4. it can end the path of execution. This means that the token has concluded.
5. it can modify the whole *run-time structure* of the process instance. The run-time structure is a process instance that contains a tree of tokens, each of which 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.

The Business Process Manager contains a set of pre-implemented node types, each of which has a specific configuration and behavior. However, one can also write one's own node behavior and use it in a process.

6.3.2. Node Type: Task Node

A *task node* represents one or more tasks that are to be performed by humans. Thus, when the execution process arrives in a node, task instances will be created in the lists belonging to the workflow participants. After that, the node will enter a `wait state`. When the users complete their tasks, the execution will be triggered, making it resume.

6.3.3. Node Type: State

A *state* is a "bare bones" `wait state`. It differs from a task node in that no task instances will be created for any task list. This can be useful if the process is waiting for an external system. After that, the process will go into a wait state. When the external system send a response message, a `token.signal()` is normally invoked, triggering the resumption of the process execution.

6.3.4. Node Type: Decision

There are two ways in which one can model a decision, the choice as to which to use being left to the discretion of the user. The options are:

1. the decision is made by the process, and is therefore specified in the process definition,

2. an external entity decides.

When the decision is to be undertaken by the process, use a `decision node`. Specify the decision criteria in one of two ways, the simplest being to add condition elements to the transitions. (Conditions are EL expressions or beanshell scripts that return a Boolean value.)

At run-time, the decision node will, firstly, loop over those `leaving transitions` on which conditions have been specified. It will evaluate those transitions first in the order specified in the XML. The first transition for which the condition resolves to **`true`** will be taken. If the conditions for all transitions resolve to **`false`**, the default transition, (the first in the XML), will taken instead.

The second approach is to use an expression that returns the name of the transition to take. Use the expression attribute to specify an expression on the decision. This will need to resolve to one of the decision node's `leaving transitions`.

One can also use the handler element on the decision, as this element can be used to specify an implementation of the `DecisionHandler` interface that can be specified on the decision node. In this scenario, the decision is calculated by a Java class and the selected `leaving transition` is returned by the `decide` method, which belongs to the `DecisionHandler` implementation.

When the decision is undertaken by an external party, always use multiple transitions that will leave a `state` or `wait state` node. The leaving transition can then be provided in the external trigger that resumes execution after the `wait state` is finished (these might, for example, be `Token.signal(String transitionName)` or `TaskInstance.end(String transitionName)`.)

6.3.5. Node Type: Fork

A fork splits a single path of execution into multiple concurrent ones. By default, the fork creates a child token for each transition that leaves it, (thereby creating a parent-child relation between the tokens that arrives in the fork.)

6.3.6. Node Type: Join

By default, the join assumes that all tokens that arrive within itself are children of the same parent. (This situation occurs 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 it. It will then examine the parent-child relation of those tokens. When all sibling tokens have arrived in the join, the parent token will be passed through to the `leaving transition`. When there are still sibling tokens active, the join will behave as a `wait state`.

6.3.7. Node Type: Node

Use this node to avoid writing custom code. It expects only one sub-element action, which will be run when the execution arrives in the node. Custom code written in `actionhandler` can do anything but be aware that it is also responsible for passing on the execution. (See *Section 6.3.1, " Node Responsibilities "* for more information.)

This node can also be used when one is utilizing a Java API to implement some functional logic for a corporate business analyst. It is advantageous to do so this way because the node remains visible in the graphical representation of the process. (Use actions to add code that is invisible in the graphical representation of the process.)

6.4. Transitions

Transitions have both source and destination nodes. The source node is represented by the property from and the destination is represented by to.

A transition can, optionally, be given a name. (Indeed, most features of the Business Process Manager depend on transitions being given unique names.) If more than one transition has the same name, the first of these will be taken. (In case duplicate transition names occur in a node, the `Map getLeavingTransitionsMap()` method will return less elements than `List getLeavingTransitions()`.)

6.5. Actions

Important

There is a difference between an action that is placed on an event and an action that is placed in a node. Actions that are put in events are executed when the event fires. They have no way to influence the flow of control of the process. (It is similar to the *observer pattern*.) By contrast, an action placed on a node has the responsibility of passing on the execution.

Read this section to study an example of an action on an event. It demonstrates how to undertake a database update on a given transition. (The database update is technically vital but it is not of importance to the business analyst.)

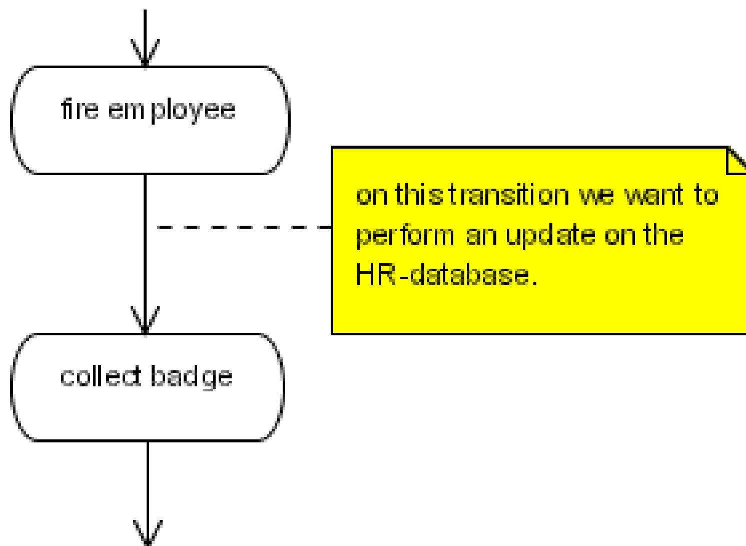


Figure 6.2. A database update action

```
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 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>
```

Note

To learn more about adding configurations to custom actions, see *Section 14.2.3, “Configuration of delegations”*

6.5.1. Action References

Actions can be given names. This allows for them be referenced from other locations in which actions are specified. Named actions can also be added to the process definition as *child elements*.

Use this feature to limit duplication of action configurations. (This is particularly helpful when the action has complicated configurations or when run-time actions have to be scheduled or executed.)

6.5.2. Events

Events are specific moments in the execution of the process. The Business Process Manager's engine will "fire" events during *graph execution*, which occurs when the software calculates the next state, (in other words, when it processes a signal.) An event is always relative to an element in the process definition.

Most process elements can fire different types of events. A node, for example, can fire both `node-enter` and `node-leave` events. (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.)

6.5.3. Passing On Events

A *super-state* creates a parent-child relation in the elements of a process definition. (Nodes and transitions contained in a super-state will have that superstate as a parent. Top-level elements have the process definition as their parent which, itself, does not have a further parent.) When an event is fired, the event will be passed up the parent hierarchy. This allows it both to capture all transition events in a process and to associate actions with these events via a centralized location.

6.5.4. Scripts

A *script* is an action that executes a **Beanshell** script. (For more information about **Beanshell**, see *http://www.beanshell.org/*.) By default, all process variables are available as script variables but no script variables will be written to the process variables. The following script-variables are 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, use the variable element as a sub-element of script. If doing so, also place the script expression into the script as a sub-element: 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 variable's access attribute contains **`read`**, the process variable will be loaded as a script variable before the script is evaluated. If the access attribute contains **`write`**, the script variable will be stored as a process variable after evaluation. The mapped-name attribute can make the process variable available under another name in the script. Use this when the process variable names contain spaces or other invalid characters.

6.5.5. Custom Events

Run custom events at will during the execution of a process by calling the `GraphElement.fireEvent(String eventType, ExecutionContext executionContext);` method. Choose the names of the event types freely.

6.6. Super-States

A super-state is a group of nodes. They can be nested recursively and are used to add a hierarchy to the process definition. (For example, use this functionality to group all of the nodes belonging to a process in phases.)

Actions can be associated with super-state events. A consequence of this is that a token can be in multiple nested nodes at any given time. This can be convenient when checking if a process execution is in, for example, the start-up phase. One is free to group any set of nodes into a super-state.

6.6.1. Super-State Transitions

Any of the transitions leaving a super-state can be taken by tokens in the nodes found within that same super state. Transitions can also arrive in super-states, in which case the token will be redirected to the first node in it. Furthermore, nodes which are outside the super-state can have transitions directly to nodes that are inside it and vice versa. Finally, super-states can also be self-referential.

6.6.2. Super-State Events

Two events are unique to super-states, these being `superstate-enter` and `superstate-leave`. They will be fired irrespective of which transitions the node has entered or left. (As long as a token takes transitions within the super-state, these events will not be fired.)

Note

There are separate event types for states and super-states. The software was designed this way in order to make it easy to distinguish between actual super-state events and node events which have been passed from within the super-state.

6.6.3. Hierarchical Names

Node names have to be unique (within their *scope*.) The scope of the node is its *node-collection*. (Both the process definition and the super-state are node collections.) To refer to nodes in super-states, 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 refer to a node in a super-state:

```
<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 shows how to travel up the super-state 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>
```

6.7. Exception Handling

The Business Process Manager's exception handling mechanism only works for Java exceptions. Graph execution cannot, of itself, result in problems. It is only when *delegation classes* are executed that exceptions can occur.

A list of `exception-handlers` can be specified on `process-definitions`, nodes and `transitions`. Each of these exception handlers has a list of actions. When an exception occurs in a delegation class, the process element's parent hierarchy is searched for an appropriate `exception-handler`, the actions for which are executed.

Important

The Business Process Manager's exception handling differs in some ways from 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 exception handling mechanism. The exception is either caught or it is not. Exceptions which have not been caught are thrown to the client that called the `token.signal()` method. For those exceptions that are caught, the graph execution continues as if nothing had occurred.

Note

Use `Token.setNode(Node node)` to put the token in an arbitrary node within the graph of an exception-handling `action`.

6.8. Process Composition

The Business Process Manager supports *process composition* by means of the `process-state`. This is a state that is associated with another process definition. When graph execution arrives in the `process-state`, a new instance of the sub-process is created. This sub-process is then associated with the path of execution that arrived in the process state. The super-process' path of execution will wait until the sub-process has ended and then 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>
```

In the example above, the **`hire`** process contains a `process-state` that spawns an **`interview`** process. When execution arrives in the **`first interview`**, a new execution (that is, process instance) of the **`interview`** process is created. If a version is not explicitly specified, the latest version of the sub-process is used. To make the Business Process Manager instantiate a specific version, specify the optional version attribute. To postpone binding the specified or latest version until the sub-process is actually created, set the optional binding attribute to **`late`**.

Next, **`hire`** process variable **`a`** is copied into **`interview`** process variable **`aa`**. In the same way, **`hire`** variable **`b`** is copied into interview variable **`bb`**. When the interview process finishes, only variable **`aa`** is copied back into the **`a`** variable.

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

6.9. Custom Node Behavior

Create custom nodes by using a special implementation of the **`ActionHandler`** that can execute any business logic, but also has the responsibility to pass on the graph execution. Here is an example that reads a value from an ERP system, adds an amount (from the process variables) and stores the result back in the ERP system. Based on the size of the amount, use either the `small amounts` or the `large amounts` transition to exit.

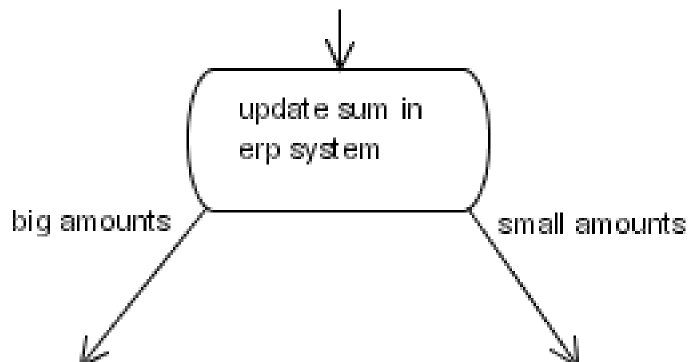


Figure 6.3. Process Snippet for Updating ERP Example

```
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 > 5000) {
      ctx.leaveNode(ctx, "big amounts");
    } else {
      ctx.leaveNode(ctx, "small amounts");
    }
  }
}
```

Note

One can also create and join tokens in custom node implementations. To learn how to do this, study the Fork and Join node implementation in the jBPM source code.

6.10. Graph Execution

The Business Process Manager's graph execution model is based on an interpretation of the process definition and the "chain of command" pattern.

The process definition data is stored in the database and is used during process execution.

Note

Be aware that **Hibernate**'s second level cache is used so as to avoid loading definition information at run-time. Since the process definitions do not change, **Hibernate** can cache them in memory.

The "chain of command pattern" makes each node in the graph responsible for passing on the process execution. If a node does not pass it on, it behaves as though it were a `wait state`.

Let the execution start on process instances and it will continue until it enters a `wait state`.

A token represents a path of execution. It has a pointer to a node in the process graph. During `wait state`, the tokens can be made to persist in the database.

This algorithm is used to calculate the execution of a token. Execution starts when a signal is sent to the tokenand it is then passed over the transitions and nodes via the chain of command pattern. These are the relevant methods:

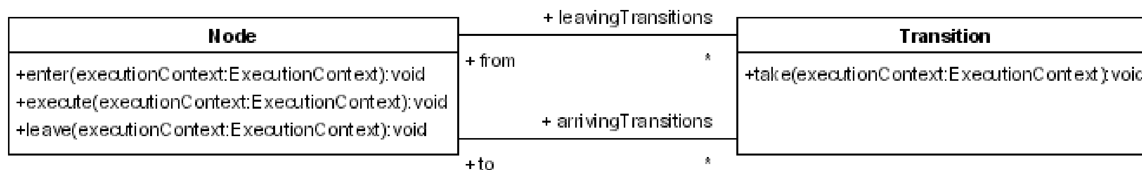


Figure 6.4. The graph execution-related methods

When a token is in a node, signals can be sent to it. A signal is treated as an instruction to start execution and must, therefore, specify a `leaving transition` from the token's current node. The first transition is the default. In a signal to a token, it takes its current node and calls the `Node.leave(ExecutionContext,Transition)` method. (It is best to think of the `ExecutionContext` as a token because the main object in it is a token.) The `Node.leave(ExecutionContext,Transition)` method will fire the `node-leave` event and call the `Transition.take(ExecutionContext)`. That method will then run the transition event and call the `Node.enter(ExecutionContext)` on the transition's destination node. That method will then fire the `node-enter` event and call the `Node.execute(ExecutionContext)`.

Every type of node has its own behaviour, these being implemented via the `execute` method. Each node is responsible for passing on the 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

The next state, including the invocation of the actions, is calculated via the client's thread. A common misconception is that all calculations must be undertaken in this way. Rather, as is the case with any *asynchronous invocation*, one can use *asynchronous messaging* (via Java Message Service) for that. When the message is sent in the same transaction as the process instance update, all synchronization issues are handled correctly. 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 to those wishing to maximise business process performance.

6.11. Transaction Demarcation

As explained in *Section 6.10, “ Graph Execution ”*, the Business Process Manager runs the process in the thread of the client and is, by nature, synchronous. In practice, this means that the

`token.signal()` or `taskInstance.end()` will only return when the process has entered a new `wait state`.

Note

To learn more about the jPDL feature being described in this section, read *Chapter 10, Asynchronous Continuations* .

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

Sometimes, in-process calculations take a lot of time, so this behavior might be undesirable. To cope with this issue, the Business Process Manager includes an asynchronous messaging system that allows it to continue a process in a manner, which is, as the name implies, asynchronous. (Of course, in a Java enterprise environment, jBPM can be configured to use a Java Message Service broker instead of the in-built messaging system.)

jPDL supports the **`async="true"`** attribute in every node. 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 (in other words, `token.signal()` or `taskInstance.end()` will be returned.)

The Business Process Manager's client code can now commit the transaction. Send messages in the same transaction as that containing the process updates. (The overall result of such a transaction will be that the token is moved to the next node (which has not yet been executed) and a **`org.jbpm.command.ExecuteNodeCommand`** message will be sent from the asynchronous messaging system to the `jBPM Command Executor`. This reads the commands from the queue and executes them. In the case of the **`org.jbpm.command.ExecuteNodeCommand`**, the process will be continued when the node is executed. (Each command is executed in a separate transaction.)

Important

Ensure that a `jBPM Command Executor` is running so that asynchronous processes can continue. Do so by configuring the web application's **`CommandExecutionServlet`**.

Note

Process modelers do not need to be excessively concerned with asynchronous messaging. The main point to remember is transaction demarcation: by default, the Business Process Manager will operate in the client transaction, undertaking the whole calculation until the process enters a `wait state`. (Use **`async="true"`** to demarcate a transaction in the process.)

Here is 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" />
  ...
```

The client code needed to both start and resume process executions is exactly the same as that needed for 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 occurs, the process execution's `root token` will point to **`node one`** and an **`ExecuteNodeCommand`** message is 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 pass the execution on or enter a `wait state`. If it chooses to pass it on, the transaction will be ended when the execution arrives at **`node two`**.

The Context

Read this chapter to learn about *process variables*. Process variables are key-value pairs that maintain process instance-related information.

Note

Since one must be able to store the *context* in a database, some minor limitations apply.

7.1. Accessing Process Variables

`org.jbpm.context.exe.ContextInstance` serves as the central interface for process variables. Obtain the `ContextInstance` from a process instance in this manner:

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

These are the basic operations:

```
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 name is **`java.lang.String`**. By default, the Business Process Manager supports the following value types. (It also supports any other class that can be persisted with **Hibernate**.)

`java.lang.String`
`java.lang.Character`
`java.lang.Double`
`java.lang.Byte`
`java.util.Date`
`java.io.Serializable`
`java.lang.Boolean`
`java.lang.Float`
`java.lang.Long`
`java.lang.Integer`
`byte[]`

Note

Untyped null values can also be stored persistently.

Warning

Do not save a process instance if there are any other types stored in the process variables as this will cause an exception error.

7.2. Lifes of Variables

Variables do not have to be declared in the process archive. At run-time, any Java object can simply be put in the variables. If a variable did not exist, it will be created, in the same way that this occurs for a plain `java.util.Map`. (Variables can also be deleted.)

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

Types can change automatically. This means that a type is allowed to overwrite a variable with a value of a different type. It is important to always try to limit the number of type changes since this generates more comunications with the database than a plain column update.

7.3. Variable Persistence

The variables are part of the process instance. Saving the process instance in the database will synchronise the database with the process instance. (The variables are created, updated and deleted by doing this.) For more information, see *Chapter 4, Persistence* .

7.4. Variable Scopes

Each path of execution (also known as a *token*) has its own set of process variables. Variables are always requested on a path of execution. Process instances have a tree of these paths. If a variable is requested but no path is specified, the `root token` will be used by default.

The variable look-up occurs recursively. It runs over the parents of the given path of execution. (This is similar to the way in which variables are scoped in programming languages.)

When a non-existent variable is set on a path of execution, the variable is created on the `root token`. (Hence, each variable has, by default, a process scope.) To make a variable token "local", create it explicitly, as per this example:

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

7.4.1. Variable Overloading

Variable overloading means that each path of execution can have its own copy of a variable with the same name. These copies are all treated independently of each other and can be of different types. Variable overloading can be interesting if one is launching multiple concurrent paths of execution over the same transition. This is because the only thing that will distinguish these paths will be their respective set of variables.

7.4.2. Variable Over-Riding

Variable over-riding simply means that variables in *nested paths of execution* over-ride variables in more global paths of execution. Generally, "nested paths of execution" relates 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, ione can over-ride a variable named **`contact`** in the process instance scope with this variable in the nested paths of execution **`shipping`** and **`billing`**.

7.4.3. Task Instance Variable Scope

To learn about task instance variables, read *Section 8.4, “Task instance variables”*.

7.5. Transient Variables

When a process instance is persisted in the database, so too are normal variables. However, at times one might want to use a variable in a delegation class without storing it in the database. This can be achieved with *transient variables*.

Note

The lifespan of a transient variable is the same as that of a `ProcessInstance` Java object.

Note

Because of their nature, transient variables are not related to paths of execution. Therefore, a process instance object will have only one map of them.

The transient variables are accessible through their own set of methods in the context instance. They do not need to be declared in the **`processdefinition.xml`** file.

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

This chapter has covered process variables in great detail. The reader should now be confident that he or she understands this topic.

Task Management

The core role 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.

8.1. 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.

8.2. 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.

8.2.1. 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.

8.2.2. 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 is 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.

8.3. 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, the push(personal task list) and pull(group task list) models of task assignment can be applied in combination. The process can determine those responsible for a task and push it to their task list. A task can also 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. Refer to *Section 8.3.3, "The personal task list"* and *Section 8.3.4, "The group task list"* for more details.

8.3.1. 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, see *Section 8.6, "Swimlanes"*.

To allow for the creation of reusable **`AssignmentHandler`**s, each usage of an **`AssignmentHandler`** can be configured in the **`processdefinition.xml`**. See *Section 14.2, "Delegation"* for more information on how to add configuration to assignment handlers.

8.3.2. 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.

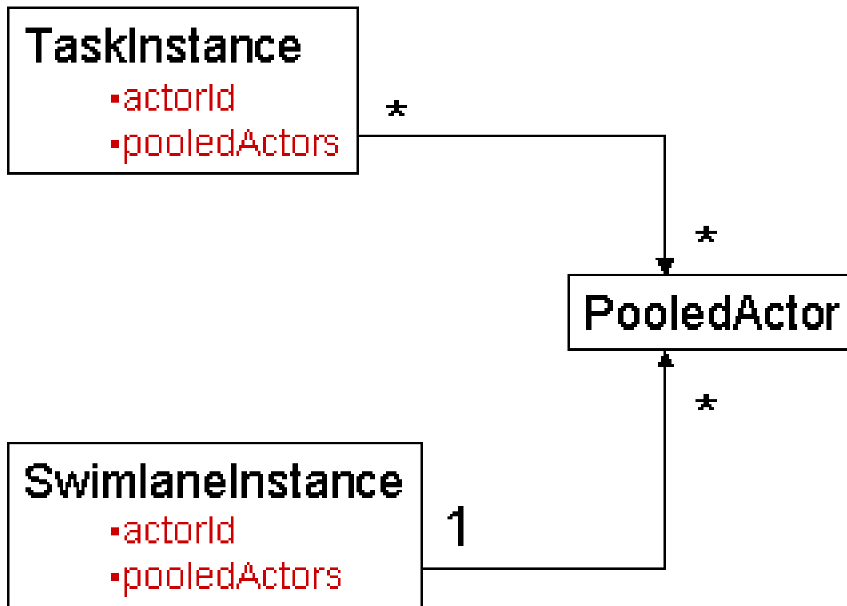


Figure 8.1. The assignment model class diagram

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.

8.3.3. 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)`**.

8.3.4. 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 setting the actorId property of the taskInstance to **`null`**.

8.4. 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 7.4, “ Variable 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.

8.5. 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:

- 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.
- 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.

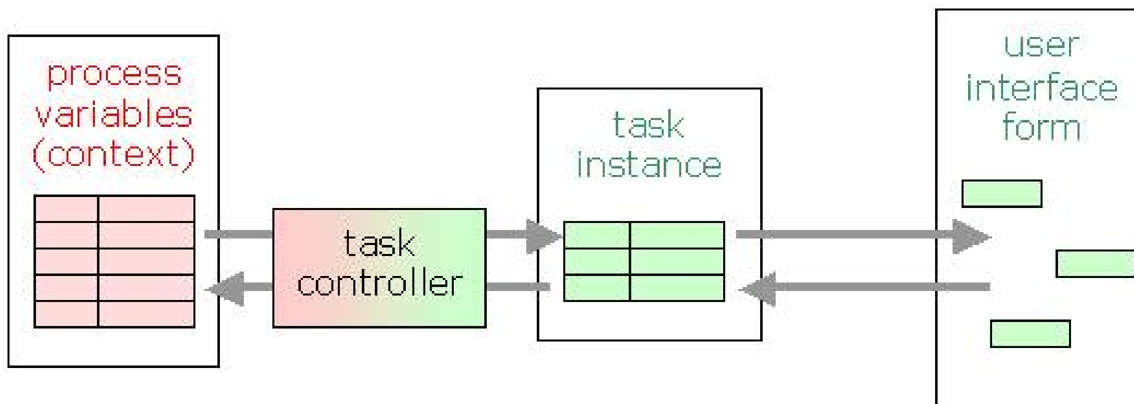


Figure 8.2. The task controllers

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=&quot;clean ceiling&quot;>
  <controller>
    <variable name=&quot;a&quot; access=&quot;read&quot; mapped-name=&quot;x&quot; />
    <variable name=&quot;b&quot; access=&quot;read,write,required&quot; mapped-
name=&quot;y&quot; />
    <variable name=&quot;c&quot; access=&quot;read,write&quot; />
  </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 one 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=&quot;clean ceiling&quot;>
  <controller class=&quot;com.yourcom.CleanCeilingTaskControllerHandler&quot;>
    -- here goes your task controller handler configuration --
  </controller>
</task>
```

8.6. 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**. See *Section 8.3, “Assignment”* for more details.

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.

8.7. Swimlane in start task

A swimlane can be associated with the start task to capture the process initiator.

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()**. and that actor will be stored in the swimlane of the start task.

```
<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 8.5, “Task controllers”*

8.8. 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 6.7, “ Exception Handling ”*.

8.9. Task timers

As on nodes, timers can be specified on tasks. See *Section 9.1, “ 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.

8.10. 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`**

8.11. 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.

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.

8.11.1. The identity model

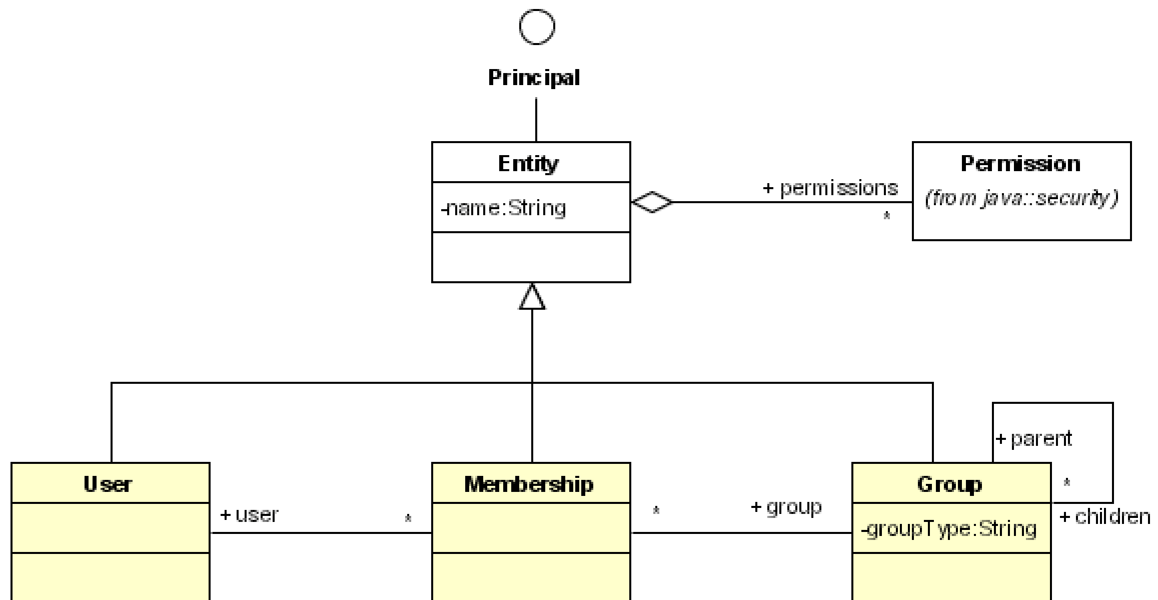


Figure 8.3. The identity model class diagram

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.

8.11.2. 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>

      <para>Syntax of the assignment expression is like this:</para>
      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)
</programlisting>
```

8.11.2.1. 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.

8.11.2.2. 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.

8.11.3. 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 remove the jBPM identity component. The only thing you need to do is make sure that you delete the following line from the **hibernate.cfg.xml**.

```
<mapping resource="org/jbpm/identity/User.hbm.xml"/>
<mapping resource="org/jbpm/identity/Group.hbm.xml"/>
<mapping resource="org/jbpm/identity/Membership.hbm.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);
```

Scheduler

Read this chapter to learn about the role of *timers* in the Business Process Manager.

Timers can be created upon events in the process. Set them to trigger either action executions or event transitions.

9.1. Timers

The easiest way to set a timer is by adding a *timer element* to the node. This sample code shows how to do so:

```
<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>
  <transition name='time-out-transition' to='...' />
</state>
```

A timer specified on a node is not executed after that node is exited. Both the transition and the action are optional. When a timer is executed, the following events occur in sequence:

1. an event of type `timer` is fired.
2. if an action is specified, it executes.
3. a signal is to resume execution over any specified transition.

Every timer must have a unique name. If no name is specified in the `timer` element, the name of the node is used by default.

Use the timer action to support any action element (such as `action` or `script`.)

Timers are created and canceled by actions. The two pertinent `action-elements` are `create-timer` and `cancel-timer`. In actual fact, the timer element shown above is just short-hand notation for a `create-timer` action on `node-enter` and a `cancel-timer` action on `node-leave`.

9.2. Scheduler Deployment

Process executions create and cancel timers, storing them in a *timer store*. A separate `timer runner` checks this store and execute each timers at the due moment.

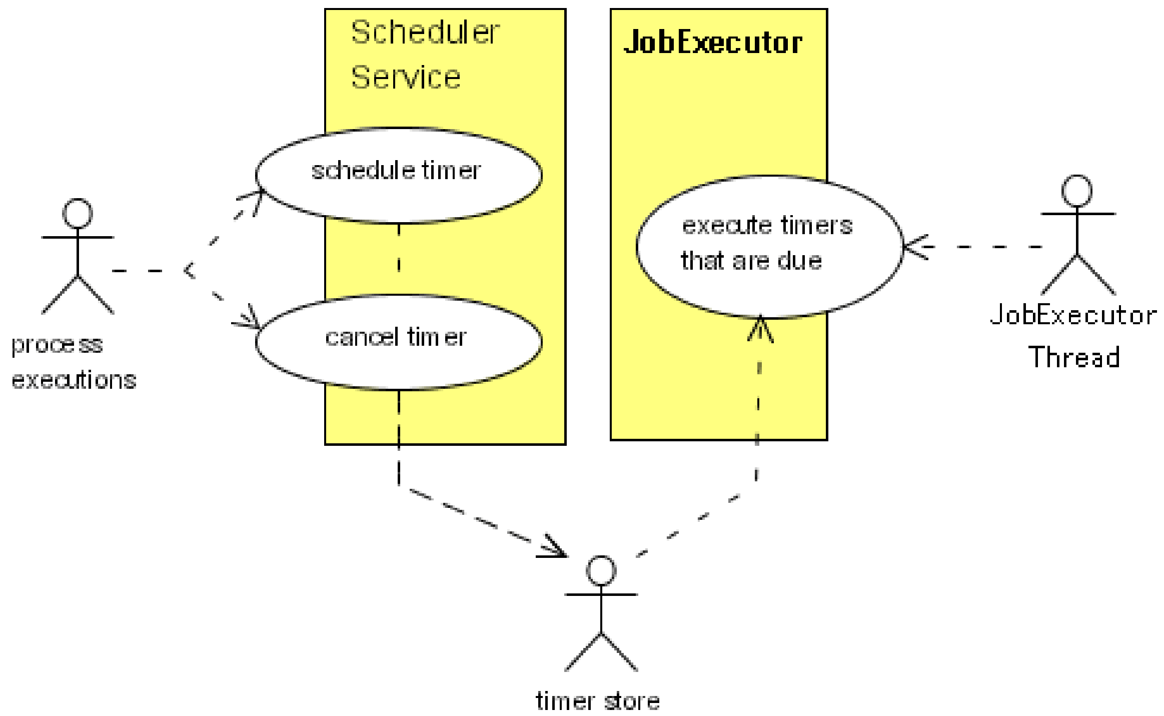


Figure 9.1. Scheduler Components Overview

Asynchronous Continuations

10.1. 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 specified execution algorithm, all state transitions are undertaken in a single thread client operation. By default, it is a good approach to perform state transitions in the thread of the client because it fits naturally with server-side transactions. The process execution moves from one "wait" state to another in the space of one transaction.

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"`** is supported only when it is triggered in an event but can be specified on all node types and all action types.

10.2. An Example

Normally, a node is always executed after a token has entered it. Hence, the node is executed in the client's thread. One will explore asynchronous continuations by looking at two examples. The first example is part of a process with three 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 diagram 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.

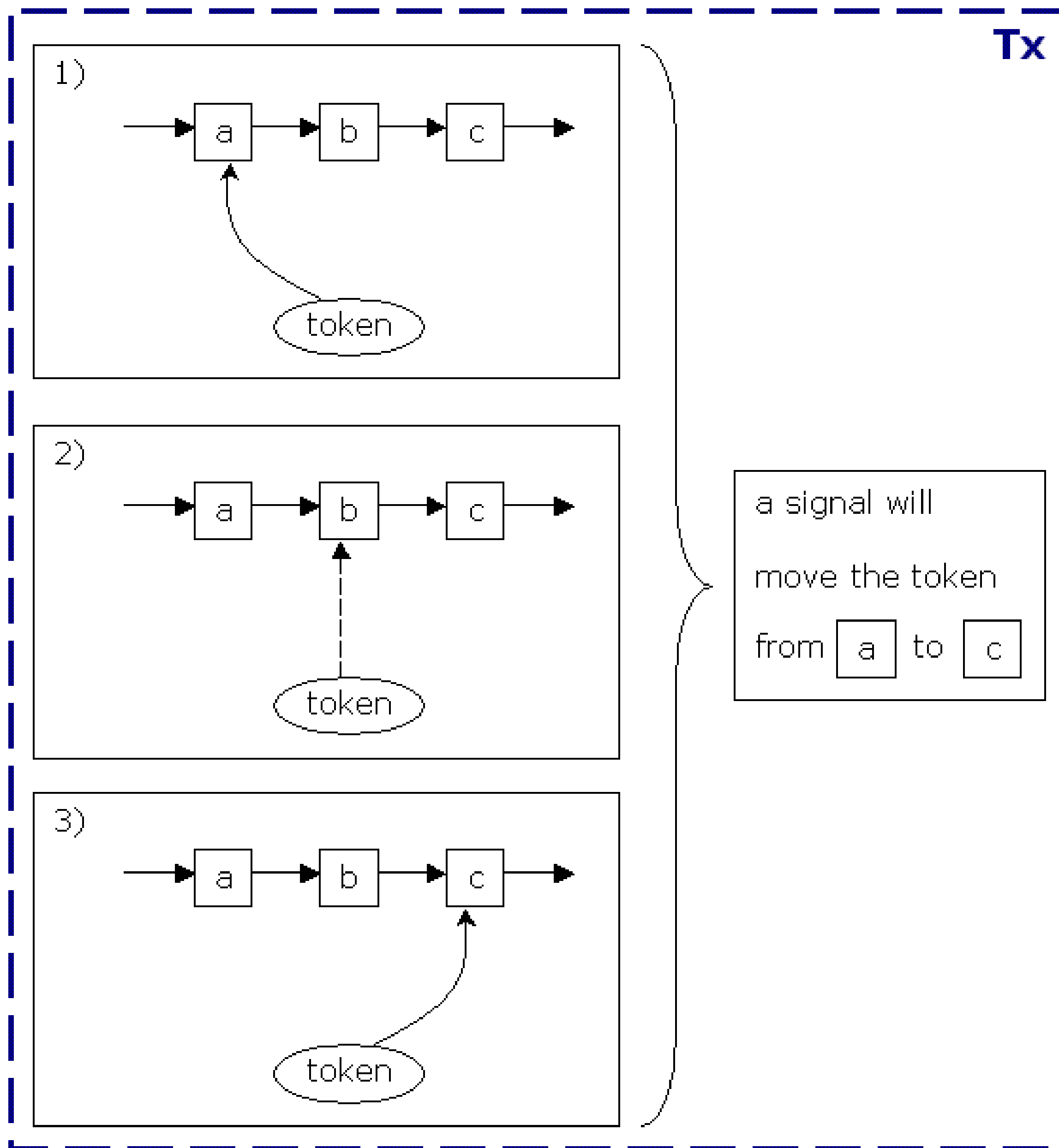


Figure 10.1. Example One: Process without Asynchronous Continuation

Whilst "persistence" is not mandatory in jBPM, most commonly a signal will be called within a transaction. Look at the updates of that transaction. Initially, 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. Secondly, in case the automated action accesses and updates some transactional resources, such updates should be combined or made part of the same transaction.

The second example is a variant of the first 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 into two parts. The first of these will execute the process up to the point at which node 'b' is to be executed. The second part will execute node 'b.' That execution will stop in wait state 'c'.

The transaction will hence be split into two separate transactions, one 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.

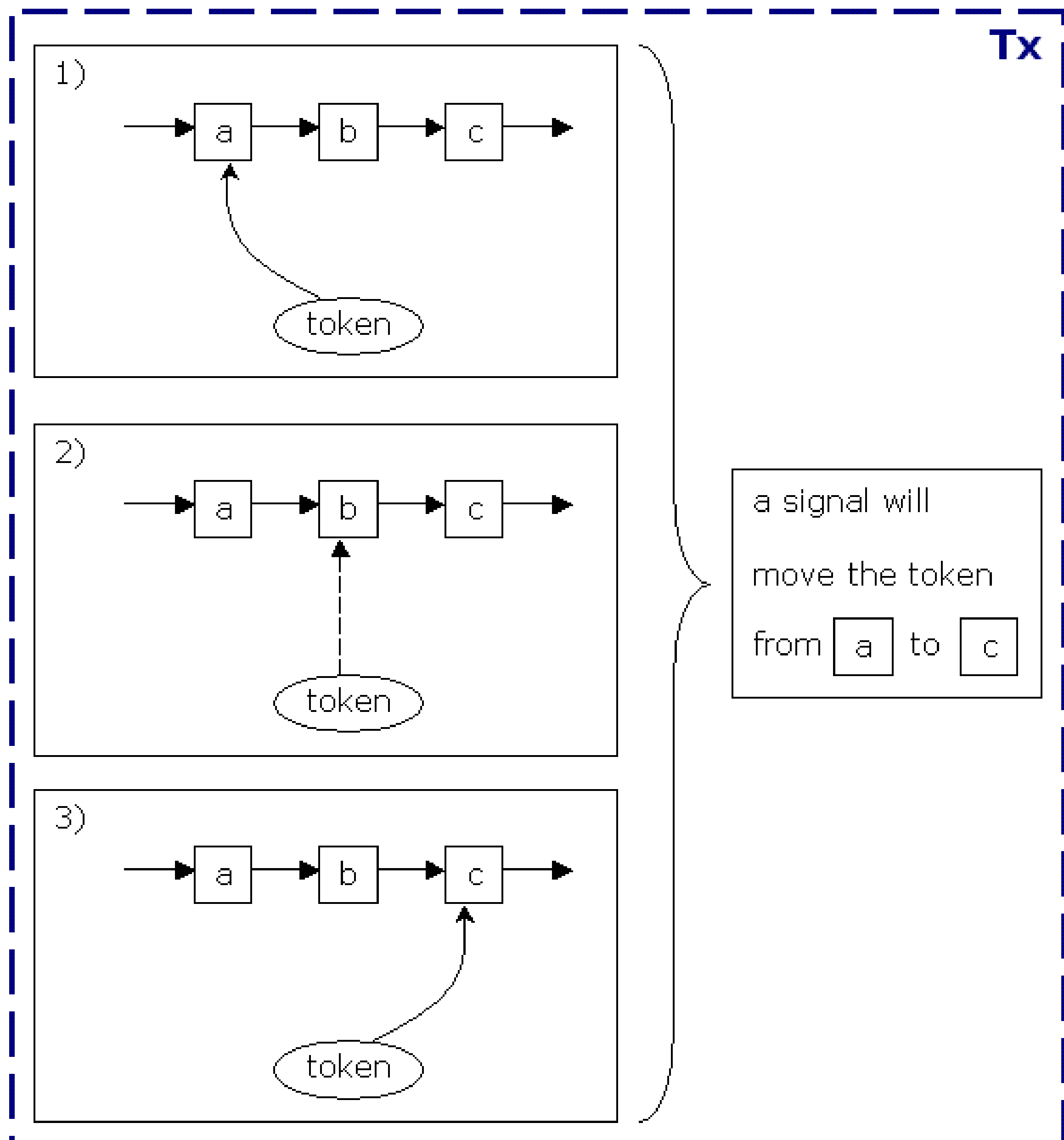


Figure 10.2. A Process with Asynchronous Continuations

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.

10.3. The Job Executor

The *job executor* is the component that resumes process executions asynchronously. It waits for job messages to arrive over an asynchronous messaging system and executes them. The two job messages used for asynchronous continuations are **ExecuteNodeJob** and **ExecuteActionJob**.

These job messages are produced by the process execution. During process execution, for each node or action that has to be executed asynchronously, a **Job** (Plain Old Java Object) will be dispatched to the **MessageService**. The message service is associated with the **JbpmContext** and it just collects all the messages that have to be sent.

The messages will be sent as part of **JbpmContext.close()**. That method cascades the **close()** invocation to all of the associated services. The actual services can be configured in **jbpm.cfg.xml**. One of the services, **DbMessageService**, is configured by default and will notify the job executor that new job messages are available.

The graph execution mechanism uses the interfaces **MessageServiceFactory** and **MessageService** to send messages. This is to make the asynchronous messaging service configurable (also in **jbpm.cfg.xml**). In Java EE environments, the **DbMessageService** can be replaced with the **JmsMessageService** to leverage the application server's capabilities.

The following is a brief summary of the way in which the job executor works.

"Jobs" are records in the database. Furthermore, they are objects and can be executed. Both timers and asynchronous messages are jobs. For asynchronous messages, the dueDate is simply set to the current time when they are inserted. The job executor must execute the jobs. This is done in two phases:

- A job dispatcher thread must acquire a job.
- The executor thread must execute the job.

Acquiring and executing the job is a two-transaction processs. The dispatcher thread acquires jobs from the database on behalf of all the executor threads on this node. When the executor thread takes the job, it adds its name into the owner field of the job. Each thread has a unique name based on IP address and sequence number.

A thread could die between acquisition and execution of a job. To clean-up after those situations, there is one lock-monitor thread per job executor that checks the lock times. The lock monitor thread will unlock any jobs that have been locked for more than ten minutes, so that they can be executed by another job executor thread.

The isolation level must be set to **REPEATABLE_READ** for Hibernate's optimistic locking to work correctly. **REPEATABLE_READ** guarantees that this query will only update one row in exactly one of the competing transactions.

```
update JBPM_JOB job
set job.version = 2
    job.lockOwner = '192.168.1.3:2'
where
    job.version = 1
```

Non-Repeatable Reads can lead to the following anomaly. 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 because it allows for Non-Repeatable reads to occur. So **REPEATABLE_READ** is required if you configure more than one job executor thread.

Configuration properties related to the job executor are:

jbpmConfiguration
: The bean from which configuration is retrieved.

name
: The name of this executor.

Important

This name should be unique for each node, when more than one jBPM instance is started on a single machine.

nbrOfThreads
: The number of executor threads that are started.

idleInterval
: The interval that the dispatcher thread will wait before checking the job queue, if there are no jobs pending.

Note

The dispatcher thread is automatically notifed when jobs are added to the queue.

retryInterval
: The interval that a job will wait between retries, if it fails during execution.

Note

The maximum number of retries is configured by jbpm.job.retries.

maxIdleInterval
: The maximum period for idleInterval.

historyMaxSize
: This property is deprecated, and has no affect.

maxLockTime
: The macimum time that a job can be locked, before the lock-monitor thread will unlock it.

lockMonitorInterval
: The period for which the lock-monitor thread will sleep between checking for locked jobs.

lockBufferTime
: This property is deprecated, and has no affect.

10.4. jBPM's built-in asynchronous messaging

When using jBPM's built-in asynchronous messaging, job messages will be sent by persisting them to the database. This message persisting can be done in the same transaction or JDBC connection as the jBPM process updates.

The job messages will be stored in the **`JBPM_JOB`** table.

The POJO command executor (**`org.jbpm.msg.command.CommandExecutor`**) will read the messages from the database table and execute them. 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.

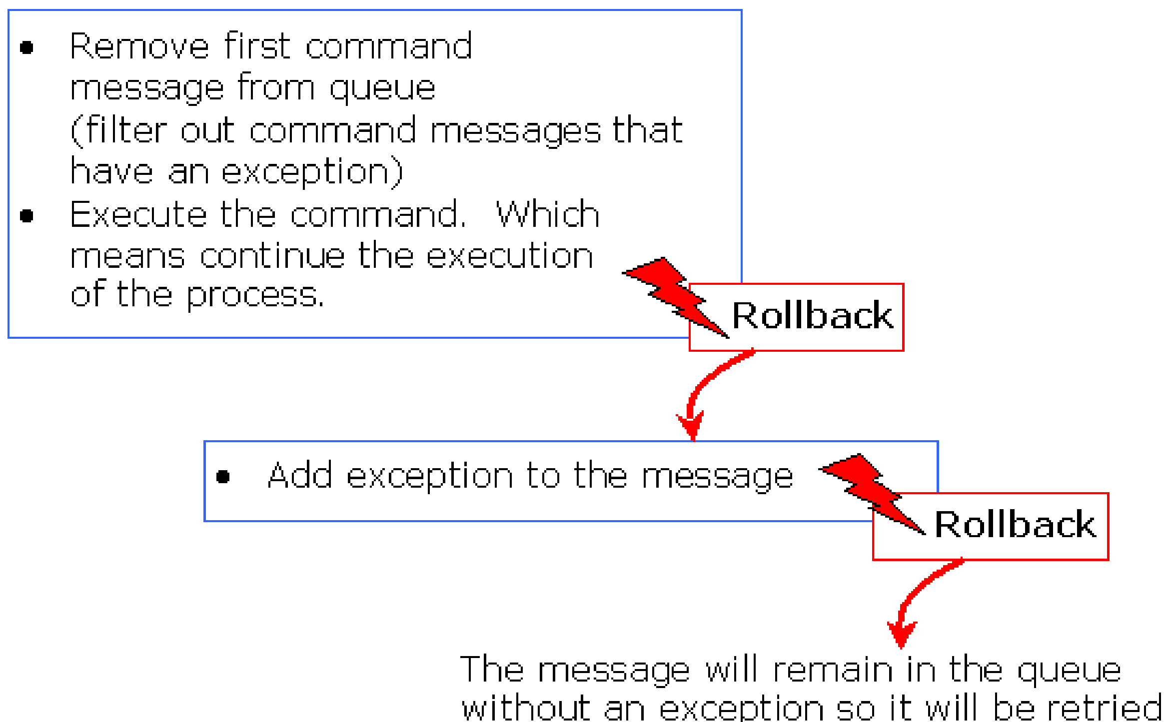


Figure 10.3. POJO command executor transactions

If the transaction that adds the exception to the command message fails, it is rolled back. The message will remain in the queue without an exception and will be retried later.

Important

jBPM's built-in asynchronous messaging system does not support multi-node locking. You cannot deploy the POJO command executor multiple times and have them configured to use the same database.

Business Calendar

Read this chapter to learn about the Business Process Manager's calendar functionality, which is used to calculate due dates for tasks and timers.

It does so by adding or subtracting a duration with a base date. (If the base date is omitted, the current date is used by default.)

11.1. Due Date

The due date is comprised of a duration and a base date. The formula used is: **`duedate ::= [<basedate> +/-] <duration>`**

11.1.1. Duration

A duration is specified in either absolute or business hours by use of this formula: **`duration ::= <quantity> [business] <unit>`**

In the calculation above, **`<quantity>`** must be a piece of text that is parsable with **`Double.parseDouble(quantity)`**. **`<unit>`** will be one of: second, seconds, minute, minutes, hour, hours, day, days, week, weeks, month, months, year or years. Adding the optional **`business`** flag will mean that only business hours will be taken into account for this duration. (Without it, the duration will be interpreted as an absolute time period.)

11.1.2. Base Date

The base date is calculated in this way: **`basedate ::= <EL>`**.

In the formula above, **`<EL>`** can be any Java Expression Language expression that resolves to a `Java Date` or `Calendar` object.

Warning

Do not reference variables of any other object types, as this will result in a `JbpmException` error.

The base date is supported in a number of places, these being a plain timer's duedate attributes, on a task reminder and the timer within a task. However, it is not supported on the repeat attributes of these elements.

11.1.3. Due Date Examples

The following uses are all valid:

```
<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" />
```

11.2. Calendar Configuration

Define the business hours in the **org/jbpm/calendar/jbpm.business.calendar.properties** file. (To customize this configuration file, place a modified copy in the root of the classpath.)

This is the default business hour specification found in **jbpm.business.calendar.properties**:

```
hour.format=HH:mm
#weekday ::= [<daypart> [&amp; <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 &amp; 12:30-17:00
weekday.tuesday=   9:00-12:00 &amp; 12:30-17:00
weekday.wednesday= 9:00-12:00 &amp; 12:30-17:00
weekday.thursday=  9:00-12:00 &amp; 12:30-17:00
weekday.friday=    9:00-12:00 &amp; 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
```

11.3. Examples

The following examples demonstrate different ways in which it can be used:

```
<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" />
```

```
hour.format=HH:mm
#weekday ::= [<daypart> [&amp; <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 &amp; 12:30-17:00
```

```
weekday.tuesday=   9:00-12:00 &amp; 12:30-17:00
weekday.wednesday= 9:00-12:00 &amp; 12:30-17:00
weekday.thursday=  9:00-12:00 &amp; 12:30-17:00
weekday.friday=    9:00-12:00 &amp; 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
```

Having studied this chapter, the reader now understands how the Business Calendar works.

E. Mail Support

This chapter describes the "out-of-the-box" e. mail support available in Business Process Manager JPDL. Read this information to learn how to configure different aspects of the mail functionality.

12.1. Mail in JPDL

There are four ways in which one can specify the point in time at which e. mails are to be sent from a process. Each shall be examined in turn.

12.1.1. Mail Action

Use a *mail action* if there is a reason not to show the e. mail as a node in the process graph.

Note

A mail action can be added to the process anywhere that a normal action can be added.

```
&lt;mail actors="#{president}" subject="readmylips" text="nomoretaxes" /&gt;
```

Specify the subject and text attributes as an element like this:

```
<mail actors="#{president}" >
    <subject>readmylips</subject>
    <text>nomoretaxes</text>
</mail>
```

Each of the fields can contain JSF-like expressions:

```
<mail
    to='#{initiator}'
    subject='websale'
    text='your websale of #{quantity} #{item} was approved' />
```

Note

To learn more about expressions, study *Section 14.3, “Expressions”*.

Two attribute specify the recipients: actors and to. The to attribute should "resolve" to a semi-colon separated list of e. mail addresses. The actors attribute should resolve to a semi-colon separated list of actorIds. These actorIds will, in turn, resolve to e. mail addresses. (Refer to *Section 12.3.3, “ Address Resolving ”* for more details.)

```
<mail
    to='admin@mycompany.com'
    subject='urgent'
    text='the mailserver is down :-)' />
```

Note

To learn how to specify recipients, read *Section 12.3, " Specifying E. Mail Recipients "*

E. mails can be defined by the use of templates. Overwrite template properties in this way:

```
<mail template='sillystatement' actors="#{president}" />
```

Note

Learn more about templates by reading *Section 12.4, " E. Mail Templates "*

12.1.2. Mail Node

As with mail actions, the action of sending an e. mail can be modeled as a node. In this case, the run-time behavior will be identical identical but the e. mail will display as a node in the *process graph*.

Mail nodes support exactly the same attributes and elements as the `mail action`. (See *Section 12.1.1, " Mail Action "* to find out more.)

```
<mail-node  name="send email"
            to="#{president}"
            subject="readmylips"
            text="nomoretaxes">
    <transition to="the next node" />
</mail-node>
```

Important

Always ensure that mail nodes have exactly one *leaving* transition.

12.1.3. "Task Assigned" E. Mail

A notification e. mail can be sent when a task is assigned to an actor. To configure this feature, add the **`notify="yes"`** attribute to a task in the following manner:

```
<task-node name='a'>
    <task name='laundry' swimlane="grandma" notify='yes' />
    <transition to='b' />
</task-node>
```

Set notify to **`yes`**, **`true`** or **`on`** to make the Business Process Manager send an e. mail to the actor being assigned to the task. (Note that this e. mail is based on a template and contains a link to the web application's task page.)

12.1.4. "Task Reminder" E. Mail

E. mails can be sent as task reminders. JPDL's reminder element utilizes the timer. The most commonly used attributes are duedate and repeat. Note that actions do not have 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>
```

12.2. Expressions in Mail

The fields **`to`**, **`recipients`**, **`subject`** and **`text`** can contain JSF-like expressions. (For more information about expressions, see *Section 14.3, "Expressions"*.)

One can use the following variables in expressions: swimlanes, process variables and transient variables beans. Configure them via the **`jbpm.cfg.xml`** file.

Expressions can be combined with *address resolving* functionality. (Refer to *Section 12.3.3, " Address Resolving "*. for more information.)

This example pre-supposes the existence of a swimlane called **`president`**:

```
<mail   actors="#{president}"
        subject="readmylips"
        text="nomoretaxes" />
```

The code will send an e. mail to the person that acts as the **`president`** for that particular process execution.

12.3. Specifying E. Mail Recipients

12.3.1. Multiple Recipients

Multiple recipients can be listed in the actors and to fields. Separate items in the list with either a colon or a semi-colon.

12.3.2. Sending E. Mail to a BCC Address

In order to send messages to a *Blind Carbon Copy* (BCC) recipient, use either the bccActors or the bcc attribute in the process definition.

```
<mail to='#{initiator}'
      bcc='bcc@mycompany.com'
      subject='websale'
      text='your websale of #{quantity} #{item} was approved' />
```

An alternative approach is to always send BCC messages to some location that has been centrally configured in **`jbpm.cfg.xml`**. This example demonstrates how to do so:

```
<jbpm-configuration>
    ...
    <string name="jbpm.mail.bcc.address" value="bcc@mycompany.com" />
</jbpm-configuration>
```

12.3.3. Address Resolving

Throughout the Business Process Manager, actors are referenced by `actorIds`. These are strings that serves to identify process participants. An *address resolver* translates `actorIds` into e. mail addresses.

Use the attribute actors to apply address resolving. Conversely, use the to attribute if adding addresses directly as it will not run apply address resolving.

Make sure the address resolver implements the following interface:

```
public interface AddressResolver extends Serializable {
 Object resolveAddress(String actorId);
}
```

An address resolver will return one of the following three types: a string, a collection of strings or an array of strings. (Strings must always represent e. mail addresses for the given `actorId`.)

Ensure that the address resolver implementation is a bean. This bean must be configured in the **`jbpm.cfg.xml`** file with name `jbpm.mail.address.resolver`, as per this example:

```
<jbpm-configuration>
    <bean name='jbpm.mail.address.resolver'
        class='org.jbpm.identity.mail.IdentityAddressResolver'
        singleton='true' />
</jbpm-configuration>
```

The Business Process Manager's `identity` component includes an address resolver. This address resolver will look for the given `actorId`'s user. If the user exists, his or her e. mail address will be returned. If not, null will be returned.

Note

To learn more about the identity component, read *Section 8.11, "The identity component"*.

12.4. E. Mail Templates

Instead of using the **`processdefinition.xml`** file to specify e. mails, one can use a template. In this case, each of the fields can still be overwritten by **`processdefinition.xml`**. Specify a templates like this:

```
<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 per the above, extra variables can be defined in the mail templates and these will be available in the expressions.

Configure the resource that contains the templates via the **jbpm.cfg.xml** like this:

```
<jbpm-configuration>
  <string name="resource.mail.templates" value="jbpm.mail.templates.xml" />
</jbpm-configuration>
```

12.5. Mail Server Configuration

Configure the mail server by setting the jbpm.mail.smtp.host property in the **jbpm.cfg.xml** file, as per this example code:

```
<jbpm-configuration>
    <string name="jbpm.mail.smtp.host" value="localhost" />
</jbpm-configuration>
```

Alternatively, when more properties need to be specified, give a resource reference to a properties file in this way:

```
<jbpm-configuration>
   <string name='resource.mail.properties' value='jbpm.mail.properties' />
</jbpm-configuration>
```

12.6. "From" Address Configuration

The default value for the `From` address field **jbpm@noreply**. Configure it via the **jbpm.xfg.xml** file with key `jbpm.mail.from.address` like this:

```
<jbpm-configuration>
    <string name='jbpm.mail.from.address' value='jbpm@yourcompany.com' />
</jbpm-configuration>
```

12.7. Customizing E. Mail Support

All of the Business Process Manager's e. mail support is centralized in one class, namely **org.jbpm.mail.Mail** This class is an `ActionHandler` implementation. Whenever an e. mail is specified in the `process` XML, a delegation to the **mail** class will result. It is possible to inherit from the **mail** class and customize certain behavior for specific needs. To configure a 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. Actions will be configured in the process that reference the configured (or the default) mail classname. Hence, if the property is changed, all the processes that were already deployed will still refer to the old mail classname. Alter them simply by sending an update statement directed at the jBPM database.

This chapter has provided detailed information on how to configure various e. mail settings. Having studied the examples carefully, one can now practice configuring one's own environment

Logging

Read this chapter to learn about the logging functionality present in the Business Process Manager and the various ways in which it can be utilized.

The purpose of logging is to record the history of a process execution. As the run-time data of each process execution alters, the changes are stored in the logs.

Note

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 the purpose of debugging it.) Process logging, by contast, traces the execution of process instances.

There are many ways in which process logging information can be useful. Most obvious of these is the consulting of the process history by process execution participants.

Another use case is that of *Business Activity Monitoring* (BAM). This can be used to query or analyze the logs of process executions to find useful statistical information about the business process. This information is key to implementing "real" business process management in an organization. (Real business process management is about how an organization manages its processes, how these processes are supported by information technology and how these two can be usede improve each other in an iterative process.)

Process logs can also be used to implement "undos." Since the logs contain a record of all run-time information changes, they can be "played" in reverse order to bring a process back into a previous state.

13.1. Log Creation

Business Process Manager modules produce logs when they run 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`, which is an optional extension of the `ProcessInstance`.

The Business Process Manager generates many different kinds of log, these being graph execution logs, context logs and task management logs. A good starting point is **`org.jbpm.logging.log.ProcessLog`** since one can use that to navigate down the `inheritance tree`.

The `LoggingInstance` collects all log entries. When the `ProcessInstance` is saved, they are flushed from here to the database. (The `ProcessInstance`'s logs field is not mapped to **Hibernate**. This is so as to avoid those logs that are retrieved from the database in each transaction.)

Each `ProcessInstance` is made in the context of a path of execution and hence, the `ProcessLog` refers to that token, which also serves as an *index sequence generator* it. (This is important for log retrieval as it means that logs produced in subsequent transactions shall have sequential sequence numbers.

Use this API method to add process logs:

```
public class LoggingInstance extends ModuleInstance {
  ...
```

```
  public void addLog(ProcessLog processLog) {...}
  ...
}
```

This is the UML diagram for information logging:

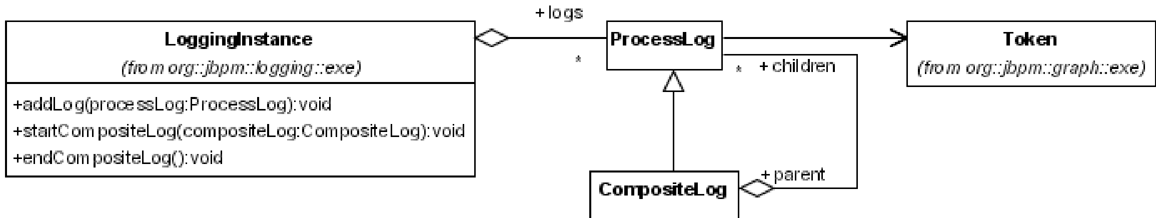


Figure 13.1. The jBPM logging information class diagram

A *CompositeLog* is a special case. It serves as the parent log for a number of children, thereby creating the means for a hierarchical structure to be applied. The following application programming interface is used to insert a log:

```
public class LoggingInstance extends ModuleInstance {
  ...
  public void startCompositeLog(CompositeLog compositeLog) {...}
  public void endCompositeLog() {...}
  ...
}
```

The `CompositeLogs` should always be called in a `try-finally-block` to make sure that the hierarchical structure is consistent. For example:

```
startCompositeLog(new MyCompositeLog());
try {
  ...
} finally {
  endCompositeLog();
}
```

13.2. Log Configurations

If logs are not important for a particular deployment, simply remove the logging line from the jbpm-context section of the **`jbpm.cfg.xml`** configuration file:

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

In order to filter the logs, write a custom implementation of the **`LoggingService`** (this is a subclass of **`DbLoggingService`**.) Having done so, create a custom `ServiceFactory` for logging and specify it in the `factory` attribute.

13.3. Log Retrieval

Process instance logs must always be retrieved via database queries. There are two methods to achieve this through `LoggingSession`.

The first method retrieves all logs for a process instance. These logs will be grouped by token in a map. This map will associate a list of `ProcessLogs` with every token in the process instance. The list will contain the `ProcessLogs` in the same order as that in which they were created.

```
public class LoggingSession {
  ...
  public Map findLogsByProcessInstance(long processInstanceId) {...}
  ...
}
```

The second method retrieves the logs for a specific token. The list will contain the `ProcessLogs` in the same order as that in which they were created.

```
public class LoggingSession {
  public List findLogsByToken(long tokenId) {...}
  ...
}
```

Having perused this chapter, the reader will now know how logging works in jBPM and has some idea of the various uses to which it can be put.

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.

14.1. 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 or UI forms for tasks.

14.1.1. 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 starter's kit. Right click on the process archive folder to find the "Deploy process archive" option. The starter's kit server contains the jBPM 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 process attribute itself is optional. Other attributes of the ant task are listed below.

jbpmcfg
: Optional. The default value is **jbpm.cfg.xml**. The JBPM configuration file can specify the location of the Hibernate configuration file (default value is **hibernate.cfg.xml**) that contains the JDBC connection properties for the database and the mapping files.

properties
: Optional. Overwrites *all* Hibernate properties as found in the **hibernate.cfg.xml**

createschema
: When set to true, the jBPM database schema is created before processes get deployed.

Process archives can also be deployed programmatically with the class **org.jbpm.jpdl.par.ProcessArchiveDeployer**

14.1.2. 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 14.2, “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.

14.1.3. 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. Migrating process instances to a new definition is the preferred solution. See *Section 14.1.4, “Migrating process instances”*.

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.

14.1.4. 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 because 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.

14.2. Delegation

Delegation is the mechanism used to include the users' custom code in the execution of processes.

14.2.1. 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 webapplications.

14.2.2. 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.

14.2.3. 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:

14.2.3.1. 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 elementnames 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 than 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>
...
```

14.2.3.2. config-type bean

Same as **config-type field** but then the properties are set via setter methods, rather than directly on the fields. The same conversions are applied.

14.2.3.3. config-type constructor

This method takes the complete contents of the delegation XML element and passes this as text in the delegation class constructor.

14.2.3.4. config-type configuration-property

First, the default constructor is used, then this method will take the complete contents of the delegation XML element, and pass it as text in method **`void configure(String);`**. (as in jBPM 2)

14.3. 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 at *http://java.sun.com/j2ee/1.4/docs/tutorial/doc/JSPIntro7.html*.

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*[1], all of your backed beans, EJB's and other **`one-kind-of-stuff`** becomes available right inside of your process definition.

14.4. jPDL XML Schema

The jPDL schema is the schema used in the file **`processdefinition.xml`** in the process archive.

14.4.1. Validation

When parsing a jPDL XML document, jBPM will validate your document against the jPDL schema when two conditions are met.

1. The schema has to be referenced in the XML document.

```
<process-definition xmlns="urn:jbpm.org:jpdl-3.2">
  ...
```

[1] http://www.jboss.com/products/seam

```
</process-definition>
```

2. The Xerces 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*.

14.4.2. process-definition

Table 14.1. Process Definition Schema

Name	Type	Multiplicity	Description
name	attribute	optional	the name of the process
swimlane	element	[0..*]	the swimlanes used in this process. The swimlanes represent process roles and they are used for task assignments.
start-state	element	[0..1]	the start state of the process. Note that a process without a start-state is valid, but cannot be executed.
{end-state|state|node|task-node|process-state|super-state|fork|join|decision}	element	[0..*]	the nodes of the process definition. Note that a process without nodes is valid, but cannot be executed.
event	element	[0..*]	the process events that serve as a container for actions
{action|script|create-timer|cancel-timer}	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.
task	element	[0..*]	global defined tasks that can be used in e.g. actions.
exception-handler	element	[0..*]	a list of exception handlers that applies to all exceptions thrown by delegation classes thrown in this process definition.

14.4.3. node

Table 14.2. Node Schema

Name	Type	Multiplicity	Description
{action|script|create-timer|cancel-timer}	element	1	a custom action that represents the behavior for this node
common node elements			*Section 14.4.4, “common node elements”*

14.4.4. common node elements

Table 14.3. Common Node Schema

Name	Type	Multiplicity	Description
name	attribute	required	the name of the node

Name	Type	Multiplicity	Description
async	attribute	{ true | false }, false is the default	If set to true, this node will be executed asynchronously. See also *Chapter 10, Asynchronous Continuations*
transition	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 specifed is called the default transition. The default transition is taken when the node is left without specifying a transition.
event	element	[0..*]	supported event types: {node-enter|node-leave}
exception-handler	element	[0..*]	a list of exception handlers that applies to all exceptions thrown by delegation classes thrown in this process node.
timer	element	[0..*]	specifies a timer that monitors the duration of an execution in this node.

14.4.5. start-state

Table 14.4. Start State Schema

Name	Type	Multiplicity	Description
name	attribute	optional	the name of the node
task	element	[0..1]	The task to start a new instance for this process or to capture the process initiator. See *Section 8.7, "Swimlane in start task"*
event	element	[0..*]	supported event types: {node-leave}
transition	element	[0..*]	the leaving transitions. Each transition leaving a node *must* have a distinct name.
exception-handler	element	[0..*]	a list of exception handlers that applies to all exceptions thrown by delegation classes thrown in this process node.

14.4.6. end-state

Table 14.5. End State Schema

Name	Type	Multiplicity	Description
name	attribute	required	the name of the end-state
end-complete-process	attribute	optional	By default end-complete-process is false which means that only the token ending this end-state is ended. If this token was the last child to end, the parent token is ended recursively. If you set this property to true, then the full process instance is ended.
event	element	[0..*]	supported event types: {node-enter}

Name	Type	Multiplicity	Description
exception-handler	element	[0..*]	a list of exception handlers that applies to all exceptions thrown by delegation classes thrown in this process node.

14.4.7. state

Table 14.6. State Schema

Name	Type	Multiplicity	Description
common node elements			See *Section 14.4.4, "common node elements"*

14.4.8. task-node

Table 14.7. Task Node Schema

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	element	[0..*]	the tasks that should be created when execution arrives in this task node.
common node elements			See *Section 14.4.4, "common node elements"*

14.4.9. process-state

Table 14.8. Process State Schema

Name	Type	Multiplicity	Description
binding	attribute	optional	Defines the moment a subprocess is resolved. {late|*} defaults to resolving deploytime
sub-process	element	1	the sub process that is associated with this node
variable	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			See *Section 14.4.4, "common node elements"*

14.4.10. super-state

Table 14.9. Super State Schema

Name	Type	Multiplicity	Description
{end-state|state|node|task-node|process-state|super-state|fork|join|decision}	element	[0..*]	the nodes of the superstate. superstates can be nested.
common node elements			See *Section 14.4.4, "common node elements"*

14.4.11. fork

Table 14.10. Fork Schema

Name	Type	Multiplicity	Description
common node elements			See *Section 14.4.4, "common node elements"*

14.4.12. join

Table 14.11. Join Schema

Name	Type	Multiplicity	Description
common node elements			See *Section 14.4.4, "common node elements"*

14.4.13. decision

Table 14.12. Decision Schema

Name	Type	Multiplicity	Description
handler	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 14.4.29, "condition"*
common node elements			See *Section 14.4.4, "common node elements"*

14.4.14. event

Table 14.13. Event Schema

Name	Type	Multiplicity	Description
type	attribute	required	the event type that is expressed relative to the element on which the event is placed
{action|script|create-timer|cancel-timer}	element	[0..*]	the list of actions that should be executed on this event

14.4.15. transition

Table 14.14. Transition Schema

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 6.6.3, " Hierarchical Names "*
condition	attribute or element text	optional	a guard condition expression. These condition attributes (or child elements) can be used in decision nodes, or to calculate the available transitions on a token at runtime.
{action|script|create-timer|cancel-timer}	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	element	[0..*]	a list of exception handlers that applies to all exceptions thrown by delegation classes thrown in this process node.

14.4.16. action

Table 14.15. Action Schema

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	attibute	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	attibute	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.

Name	Type	Multiplicity	Description
expression	attibute	either this, a class or a ref-name	A jPDL expression that resolves to a method. See also *Section 14.3, “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 events that were fired on this action's element. for more information, see *Section 6.5.3, “ Passing On Events ”*
config-type	attribute	optional	{field|bean|constructor|configuration-property}. 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}	**'async="true"** is only supported in **action** when it is triggered in an event. The default value 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 14.2.3, “Configuration of delegations”*.

14.4.17. script

Table 14.16. Script Schema

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 6.5.3, “ Passing On Events ”*
expression	element	[0..1]	the beanshell script. If you don't specify variable elements, you can write the expression as the content of the script element (omitting the expression element tag).
variable	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.

14.4.18. expression

Table 14.17. Expression Schema

Name	Type	Multiplicity	Description
	{content}		a bean shell script.

14.4.19. variable

Table 14.18. Variable Schema

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`**. "required" is only relevant when one is a submitting task variable to a process variable.
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.

14.4.20. handler

Table 14.19. Handler Schema

Name	Type	Multiplicity	Description
expression	attibute	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 14.3, "Expressions"*.
class	attibute	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|bean|constructor|configuration-property}. 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.
	{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

Name	Type	Multiplicity	Description
			delegation configuration, see *Section 14.2.3, “Configuration of delegations”*.

14.4.21. timer

Table 14.20. Timer Schema

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 11.1.1, “ 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 11.1.1, “ 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 cancelled. 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|script|create-timer|cancel-timer}	element	[0..1]	an action that should be executed when this timer fires

14.4.22. create-timer

Table 14.21. Create Timer Schema

Name	Type	Multiplicity	Description
name	attribute	optional	the name of the timer. The name can be used for cancelling 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 11.1.1, “ Duration ”* for the syntax.
repeat	attribute	optional	{duration | 'yes' | 'true'}after a timer has been executed on the duedate, 'repeat' optionally

Name	Type	Multiplicity	Description
			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 11.1.1, " 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).

14.4.23. cancel-timer

Table 14.22. Cancel Timer Schema

Name	Type	Multiplicity	Description
name	attribute	optional	the name of the timer to be cancelled.

14.4.24. task

Table 14.23. Task Schema

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 trigering the continuation of the token.
duedate	attribute	optional	is a duration expressed in absolute or business hours as explained in *Chapter 11, Business Calendar*
swimlane	attribute	optional	reference to a swimlane. 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	element	optional	describes a delegation that will assign the task to an actor when the task is created.
event	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`**

Name	Type	Multiplicity	Description
exception-handler	element	[0..*]	a list of exception handlers that applies to all exceptions thrown by delegation classes thrown in this process node.
timer	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	element	[0..1]	specifies how the process variables are transformed into task form parameters. the task form paramaters are used by the user interface to render a task form to the user.

14.4.25. swimlane

Table 14.24. Swimlane Schema

Name	Type	Multiplicity	Description
name	attribute	required	the name of the swimlane. Swimlanes can be referenced and looked up via the **`TaskMgmtDefinition`**
assignment	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.

14.4.26. assignment

Table 14.25. Assignment Schema

Name	Type	Multiplicity	Description
expression	attribute	optional	For historical reasons, this attribute expression does *not* refer to the jPDL expression, 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 8.11.2, “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. 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

Name	Type	Multiplicity	Description
			this: **`pooled-actors="chicagobulls, pointersisters"`**. The pooled-actors will be resolved as an expression. 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|bean|constructor|configuration-property}. 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 14.2.3, "Configuration of delegations"*.

14.4.27. controller

Table 14.26. Controller Schema

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|bean|constructor|configuration-property}. 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	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.

14.4.28. sub-process

Table 14.27. Sub Process Schema

Name	Type	Multiplicity	Description
name	attribute	required	The name of the **`sub-process`**. It can be an EL expression but it must resolve to a **`String`**.

Name	Type	Multiplicity	Description
			Powerful especially with late binding in the process-state.
version	attribute	optional	The version of the **`sub-process`**. If version is not specified, jBPM will use the latest version of the given process as known while deploying the parent process-state.
binding	attribute	optional	Controls when the version of the **`sub-process`** is determined. The default behavior is to determine the version when deploying the parent **`process-state`**. If binding is defined as **`late`** (**`binding="late"`**) then it is determined when the **`sub-process`** is invoked. If binding is set to "late" and version is specified, then the value of version will be ignored and the latest version will be used.

14.4.29. condition

Table 14.28. Condition Schema

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 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.

14.4.30. exception-handler

Table 14.29. Exception Handler Schema

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	element	[1..*]	a list of actions to be executed when an exception is being handled by this exception handler.

Test Driven Development for Workflow

15.1. 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 15.1. The auction test process

Next, w 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());
  }
}
```

15.2. XML Sources

Before you can start writing execution scenarios, one must write 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. One then obtains the various parsing methods. There are basically three ways to write XML that can be parsed to a `ProcessDefinition` object:

15.2.1. Parsing a process archive

A process archive is a zip file that contains the process XML as the file **`processdefinition.xml`**. The jBPM process designer reads and writes process archives.

```
static ProcessDefinition auctionProcess =
    ProcessDefinition.parseParResource("org/jbpm/tdd/auction.par");
```

15.2.2. 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 an Ant script. In that case, you can use the **`JpdlXmlReader`**

```
static ProcessDefinition auctionProcess =
    ProcessDefinition.parseXmlResource("org/jbpm/tdd/auction.xml");
```

15.2.3. 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>");
```

Appendix A. Revision History

Revision 1.5 **Mon Mar 21 2011** **David Le Sage** *dlesage@redhat.com*

Updated for 4.3.CP05 Release

Revision 1.4 **Tue Apr 27 2010** **David Le Sage** *dlesage@redhat.com*

Updated for SOA 4.3.CP04
SOA-1998 - Added Configuring the Job Executor. Section 3.5
SOA-1285 - Added Persistence/Schema information. Sections 4.2.2 and 4.2.3
SOA-1898 - Clarified details of variable attribute. Section 14.4.19

Revision 1.3 **Tue Apr 20 2010** **David Le Sage** *dlesage@redhat.com*

Updated for SOA 4.3.CP03

Revision 1.2 **Mon Aug 17 2009** **Darrin Mison** *dmison@redhat.com*

Updated for SOA 4.3.CP02
SOA-1436 - Clarified details of sub-process attributes. Section 16.4.28
SOA-1435 - Updated process archive deployment details. Section 16.1.1
SOA-1434 - Updated Asynchronous Continuations content. Section 3.3.4, 12.1 and 16.4.16

Revision 1.1 **Fri Feb 27 2009** **Darrin Mison** *dmison@redhat.com*

Updated for SOA 4.3.CP01
Updated database upgrades chapter

Revision 1.0 **Thu Sep 04 2008** **Joshua Wulf** *jwulf@redhat.com*

Converted to publican format