

Nova Bonita BPM

CookBook

BPM Nova Bonita

Nova Bonita BPM

CookBook

Nova Bonita (aka Bonita v4)

Software

Mai 2009

Table Of Contents

TABLE OF CONTENTS.....	III
LIST OF FIGURES.....	VIII
CHAPTER 1.OVERVIEW.....	11
CHAPTER 2.GENERAL INFORMATION.....	12
2.1Nova Bonita introduction.....	12
2.2Feature List.....	12
2.3Restrictions.....	13
2.4Hardware prerequisites.....	13
2.5Software prerequisites.....	14
CHAPTER 3.CONCEPTS.....	15
3.1Terminology.....	15
3.2Package.....	15
3.2.1LifeCycle.....	16
3.2.2Versioning.....	16
3.3Process.....	16
3.3.1Process basics.....	16
3.3.2Life Cycles (process/instance).....	16
3.3.3Process definition & process Instances.....	17
3.3.4Versioning.....	17
3.3.5Concept of Hooks.....	17
3.3.6SubProcesses.....	18
3.4Activities.....	18
3.4.1Activity Basics.....	18
3.4.2Activities life cycle.....	19
3.4.3Life cycle for tasks (aka manual activity).....	19
3.4.4Transition between Activities.....	20
3.4.5Iterating Activities.....	20
3.4.6Activity multi-instantiation.....	21
3.4.7Concepts of hooks/connectors.....	22
3.4.8Activity/Hooks and Transactions.....	22
3.5Role Mappers Feature.....	23
3.5.1Overview.....	23
3.5.2Custom Mappers.....	24
3.5.3Instance Initiator.....	24
3.6Performer Assignment.....	24
3.6.1Overview.....	24

3.6.2Custom Performer Assignment.....	25
3.6.3Variable Performer Assignment.....	25
CHAPTER 4.CONFIGURATION AND SERVICES.....	26
4.1Services Container.....	26
4.2Services.....	27
4.2.1Persistence.....	27
4.2.2Identity.....	28
4.2.3Security.....	28
4.2.4Task Management.....	29
4.2.5Journal and History.....	29
4.2.6Timers.....	30
CHAPTER 5.INSTALLATION.....	31
5.1.1Distribution.....	31
5.2Standard vs Enterprise installation.....	32
5.2.1Standard installation (Bonita as a library).....	32
5.2.2Enterprise installation (Bonita as a server).....	32
5.2.2.1Jboss 4.x and 5.x installation and deployment.....	33
5.2.2.2JOnAS 4.x and 5.x installation and deployment.....	33
5.2.2.3EasyBeans EJB3 Installation and deployment.....	34
CHAPTER 6.DEFINITION.....	35
6.1Designing a xpdI process with ProEd.....	35
6.2Versioning Support in ProEd.....	35
6.3Starting ProEd and ProEd Modes of Operation.....	35
6.3.1Launching ProEd as a desktop application	36
6.3.2Launching ProEd as an Eclipse plugin.....	36
6.4Creating a BPM/BPM Eclipse project.....	37
6.5Using ProEd BPM editor	44
6.5.1Creating a New BPM Project.....	44
6.5.2Interface Overview.....	47
6.5.3Load/Save/SaveAs/Delete Projects.....	56
6.5.4Defining BPM Process Properties.....	60
6.5.5Adding Participants.....	60
6.5.6Creating and Defining Activities.....	64
6.5.7Creating Variables.....	68
6.5.8Adding Hooks.....	69
CHAPTER 7.DEVELOPMENT.....	75
7.1Nova Bonita APIs.....	75
7.1.1Getting started with Bonita APIs.....	75
7.1.2Nova Bonita APIs, playing with !.....	75
7.1.2.1Nova Bonita local vs remote applications !.....	76
7.2Running the examples.....	76

7.3Java Properties.....	77
7.4Administration operations.....	78
7.5Database configuration.....	78
7.6Changing the default database configuration.....	79
7.7Connecting with your Information System (Hooks).....	80
7.7.1Hooks Execution Time Scale	80
7.7.2Out-of-Timescale Hooks.....	81
7.7.3Hooks Capabilities.....	81
7.7.4BPM-Related Hook Actions.....	81
7.7.5Java-Environment-Related Hook Actions.....	81
7.7.6Hooks Logic.....	82
7.7.7Fault Management.....	82
7.7.8Activity/Hooks and Transactions.....	84
7.7.9Writing a Hook.....	84
7.7.10Hooks-Specific Operations.....	84
7.7.11Caveat Regarding Activity Deadline.....	85
7.7.12Use Case.....	85
7.7.12.1A Simple Hook.....	85
7.7.12.2A More Complex Hook.....	85
7.7.13Practical Steps for Hooks Usage.....	87
7.7.13.1Hook Loading and Compiling.....	87
7.7.13.2Hooks deployment.....	87
7.8Mapping BPM roles with users in your IS (Mappers).....	88
7.8.1Writing a Mapper.....	88
7.8.2Mapper Types: LDAP, Custom, and Properties.....	89
7.8.3Practical Steps for Using Custom Mappers.....	89
7.8.4Example of a Mapper.....	90
7.9Selecting actors in Activities (Performer Assignments).....	91
7.9.1Performer Assignment Types: Custom and Properties.....	91
7.9.1.1Custom Performer Assignment	91
7.9.1.2Variables Performer Assignment.....	91
7.9.2Practical Steps for Using Callback Performer Assignments.....	92
7.9.2.1Performer Assignment – Loading and Compiling	92
7.9.2.2Example of a Performer Assignment.....	93
7.10Assigning activities to multiple actors (Multi-Instantiators).....	93
7.10.1Practical Steps for Using Multi-Instantiators.....	94
7.10.1.1Multi-Instantiators – Loading and Compiling	94
7.10.1.2Example of a Multi-Instantiator.....	95
7.11Connectors.....	96
7.12How to use it.....	96
7.13How to create it.....	97
CHAPTER 8.ADMINISTRATION AND EXECUTION.....	106
8.1Installation.....	106
8.1.1Prerequisite.....	106
8.1.2Installation procedure.....	106
8.2Quick Start.....	107

8.2.1Console Start.....	107
8.2.2Deploy process.....	107
8.2.3Start Process	108
8.3Process Console Description.....	109
8.3.1Console Access.....	109
8.3.2Default users.....	109
8.3.3Console frames description.....	110
8.3.4Application's graphical organization description.....	111
8.4Accessing and Creating Processes.....	111
8.4.1Access the Workflow Process List.....	111
8.4.2Create a new Instance of a Bonita Process.....	111
8.5Access To Do / Done Tasks.....	112
8.5.1Consult the To Do Tasks List.....	112
8.5.2Perform / Pause / Resume a Task.....	113
8.5.2.1Perform.....	113
8.5.2.2Suspend.....	114
8.5.2.3Resume.....	114
8.5.3Consult the Done Tasks List.....	114
8.6Managing Process Models.....	114
8.6.1Access the Process Model List.....	114
8.6.2Deploy / Undeploy / Delete Processes Models.....	116
8.6.3Start Process Models.....	116
8.6.4Remove all Instances of a Process Model.....	116
8.7Managing Instances.....	117
8.7.1Access the Process Instances List	117
8.7.2Consult / Edit the variables of an instance.....	118
8.7.3Access the activities list of an instance.....	118
8.8Managing Activities.....	119
8.8.1Access the Activities List.....	119
8.8.2Start an Activity.....	120
8.8.3Suspend an Activity.....	121
8.8.4Resume an Activity.....	121
8.8.5Access the Variables List of an Activity.....	121
8.9Managing users, groups and memberships.....	121
8.9.1Add a new user to the Nova Bonita Console.....	122
8.9.2Set roles/permission of a user to access to the console.....	122
8.10Forms customization.....	124
8.10.1Overview.....	124
8.10.2Forms.xml syntax.....	124
8.10.2.1Tag List.....	125
<forms/>.....	125
<form/>.....	125
<activity/>.....	125
<resource-bundle/>.....	126
<customized-view/>.....	126
<submitbutton/>.....	127
<variable/>.....	127
<validator/>.....	129
<property/>.....	130
8.10.2.2Data Validators list.....	130
DateTime.....	130

EmailAddress.....	131
Expression.....	131
Number.....	132
NumberInRange.....	132
PositiveNumber.....	133
SpecialCharacter.....	133
StringLength.....	133
Float.....	134
8.10.3Internationalize your forms.....	135
8.10.3.1Syntax.....	135
Key list.....	135
CHAPTER 9.CHANGE HISTORY BETWEEN BONITA V3 AND V4.....	137
9.1Concept of package.....	137
9.1.1Package life cycle.....	137
9.2Processes, instances, activities and tasks life cycles.....	137
9.2.1Process life cycle.....	137
9.2.2Instance life cycle.....	137
9.2.3Activity life cycle.....	138
9.2.4Task life cycles.....	138
9.3APIs.....	138
9.4Hooks.....	138
9.4.1For tasks.....	138
9.4.2For automatic activities.....	139
9.4.3For processes.....	139
9.4.4Interactive hook.....	140
9.5Deadlines.....	140
9.6Mappers.....	140
9.7Performer assignments.....	140
9.8Variables.....	140
9.9Iterations.....	141

List of Figures

Creating a New BPM Project.....	37
Editing Java settings.....	38
BPM project structure.....	38
Creating a New XPD file.....	39
BPM process editor.....	40
BPM process definition through ProEd.....	40
Creating a Bonita Hook Java entity.....	41
Editing Java Hook settings.....	42
Java Hook Preview.....	42
Editing Hook java file.....	43
Bar file generation view.....	44
Creating a New BPM Project.....	44
ProEd Display for New Project.....	46
ProEd File Menu.....	47
ProEd Edit menu.....	48
ProEd Window Menu.....	48
ProEd Participant View.....	49
ProEd Activity View.....	49
ProEd Process Menu.....	50
ProEd Main toolbar.....	51
Projects View.....	52
Activity View.....	52
Participant View.....	56
Open File Dialog.....	57
Save File Dialog.....	58
ProEd Add Participant Window.....	61
Add Participant Search Window.....	62

New Participant Window.....	63
New Activity Window.....	65
Variable Menu.....	68
Add Hook Window.....	69
Iterations and Transitions Graph.....	71
Add Condition Window.....	72
Modifying Transition Properties.....	73
Login screen.....	107
Choose a process to deploy.....	107
Choose a process to deploy.....	107
Start Process Form.....	108
The created instance.....	108
Console login screen.....	109
Console description.....	110
Application description.....	111
Process list.....	111
New instance creation.....	112
ToDoList classical view.....	112
ToDoList advanced view.....	113
Perform a task.....	113
Done task list.....	114
Process Model List.....	115
Process model detailed view.....	115
Instances list for a given project.....	116
Instance detailed view.....	118
Instance Variables.....	118
Activities for a given instance.....	119
Activity detailed view.....	120
Activities variables	121

Administration page.....	122
Add a new user.....	122
Group selection.....	123
Add member dialog (1/2).....	124
Add member dialog (2/2).....	124

Chapter 1. Overview

This documentation is intended for Bonita end users (administrators, architects and developers). It introduces the Bonita v4 architecture, presents some of the main concepts in Bonita and also provides some useful installation and configuration instructions. If you are already familiar with previous Bonita versions you will find in the last chapter a change history between those versions and Bonita v4.

Chapter 2, General information describes the new version Bonita v4 called Nova Bonita

Chapter 3, Concepts describes main Bonita BPM concepts and features

Chapter 4, Configuration and Services describes main configuration features and services

Chapter 5, Installation guides you on installing the Bonita v4

Chapter 6, Definition illustrates how processes can be defined graphically in Bonita

Chapter 7, Development guides you through the discovery of Nova Bonita

Chapter 8, Administration and execution describes monitoring and end execution capabilities

Chapter 9, Change history between Bonita v3 and Bonita v4

Chapter 2. General information

2.1 Nova Bonita introduction

Nova Bonita is the name of new version of Bonita v4.

“Nova” technology is based on the “Process Virtual Machine” conceptual model for processes. The Process Virtual Machine defines a generic process engine enabling support for multiple process languages (such BPEL, XPD...).

On top of that, it leads to a pluggable and embeddable design of process engines that gives modelling freedom to the business analyst. Additionally, it enables the developer to leverage process technology embedded in a Java application.

For more information about the Process Virtual Machine, check Nova Bonita FAQs

<http://wiki.bonita.objectweb.org/xwiki/bin/view/Main/FAQ> on the Bonita web site
<http://bonita.ow2.org>

2.2 Feature List

Nova Bonita (aka Bonita v4) is a lightweight BPM solution that provides XPD 1.0 support.

Nova Bonita V4 comes with an enhanced XPD extension module, a rich BPM API, support for iterations and deadlines, multiple variables types support, activities multi instantiation, processes versioning and a set services such configurable journal, history and timers services.

Bonita 4.1.1 is the latest stable release. For a detailed list of improvements and bug fixes included in this version please check the releases notes.

Hereafter you can find the list of features available in Bonita v4:

- Powerful BPM API covering deployment, definition, runtime and history BPM data
- QueryAPI vs RuntimeAPIs for advanced resources (hooks, mappers and performer assignments)
- Standard (J2SE) vs Enterprise (J2EE) deployment
- JEE deployment includes support for both 1.4 and 1.5 standards
- Support for XPD 1.0 activities : Join, Split, Activity (Route, implementation no and subFlow) in both automatic and manual execution modes
- Support of main XPD 1.0 elements: Datafield, DataType, Participant, Transition, RedefinableHeader, Transition Restriction, Package...
- Support of advanced entities/resources: Hooks, mappers, performer assignments and activities multiinstantiators (via XPD extended attributes)
- Persistent execution (through a configurable persistence service, hibernate by default)
- Subprocesses support
- Activities multi-instantiation support
- Iterations/cycles support
- General information
- Activities deadlines support through the Process Virtual Machine generic and configurable Timer service
- Configurable journal and history BPM modules: history db vs history xml implementations
- Advanced process deployment capabilities including ".bar" file deployment and local vs global resources (hooks, mappers, performer assignments and instantiators)
- Processes and package versioning
- Standard security service based on JAAS LoginModules: Test, standard and J2EE login modules are included in the package
- Tasks (aka manual activities)assign and re-assign capabilities

- Unified life cycle for BPM activities (XPDL activities types) execution handling synchronization with Tasks, also known as manual activities, life cycle.
- Task Management module handling init, ready, executing, finished, dead, suspend and resume states
- Transitions conditions advanced support based on BeanShell scripting language and multiple variables types
- BPM data: both process and activity level variables support
- Integer, String, Float, Boolean, Date and Enumerated types are supported as variable types
- Default mapper implementation: Initiator Mapper
- Process Virtual Machine technology based
- BPM Designer (ProEd):
 - Eclipse and Desktop versions
 - "Easy BPM project" creation wizard available in Eclipse version
 - Graphical support for advanced Nova Bonita entities: hooks, mappers, performers and instantiators
 - Support for multiple variables types
 - "Smart" conditions editor: graphical definition of complex conditions based on multiple operators and variables types
 - Automatic generation of start and end BPM steps
- BPM Console
 - Web 2.0 console supporting both desktop and traditional portal layout modes
 - Monitoring vs Worklist applications (portlets)
 - Internal user repository handling access rights to applications
 - Automatic generation of forms vs customized forms
 - Console customization capabilities: on the fly page creation, add/remove applications and widgets, look and feel...
 - Applications (portlets) and widgets support

2.3 Restrictions

Nova Bonita comes out with an innovative architecture based on a generic and extensible engine, called "The Process Virtual Machine" and a powerful injection technology allowing services pluggability.

Nova Bonita includes support for elements defined in the XPDL 1.0 standard. Next versions will add support for XPDL 2.0 standard coverage as well as the following new features: asynchronous activities execution, process changes (instance modifications), native clustering... Check the roadmap <http://wiki.bonita.objectweb.org/xwiki/bin/view/Main/Roadmap> for more information about next developments.

This release does not yet support the following features available in Bonita v3:

- Block activities
- Process definition and process modifications via Java APIs (process changes on the fly)
- Hooks: processes hook (onInstantiate) as well as activity onCancelled hook are not yet supported

2.4 Hardware prerequisites

A 1GHz processor is recommended, with a minimum of 512 Mb of RAM. Windows users can avoid swap file adjustments and get improved performance by using 1Gb or more of RAM

2.5 Software prerequisites

Nova Bonita has been successfully tested in the following environments (should work in others but those ones are part of Nova Bonita continuous integration infrastructure):

- Operating Systems:
 - Solaris-10 (SunOS 5.10) x86
 - GNU/Linux kernel 2.6.25-2 x86 Debian
 - Windows XP
- Java Virtual Machines (jdk 1.5 and 1.6):
 - Sun-jdk1.5.0_13
 - Jrockit-R27.1.0-jdk1.5.0_08
 - Ibm-java2-i386-50
 - Jrockit-R27.2.0-jdk1.6.0
 - Sun-jdk1.6.0_06
- Relational Databases:
 - Mysql-server 5.0.51a-6
 - PostgreSQL 8.3.3-1
 - Oracle 11.1.0
 - H2 1.0.76
 - HSQL 1.8.0.7
- Application Servers:
 - Tomcat 5.5.26
 - JOnAS 5.10.3
 - Jboss-4.2.2.GA
 - JOnAS 5.1.0 RC2
 - Jboss-5.0.1.GA
 - Easybeans 1.0.1

Nova Bonita requires Apache Ant 1.7 or higher. Apache ant will allow users to deal with configuration and administration operations. It can be downloaded from <http://ant.apache.org>

Chapter 3. Concepts

3.1 Terminology

Bonita is an XPDL compliant BPM solution, so most of the concepts presented in this section are the ones included in the XPDL specification. Please, refer to this specification for more details. In the following lines we will briefly introduce those concepts and explain in details the way in which they are leveraged in Nova Bonita:

- **Package** issued from XPDL acts as a container for main BPM objects that can be shared by multiple BPM processes.
- **Process** (called BPM Process into XPDL) contains the elements that make up a BPM: activities, data fields, participants, transitions.....
- **Activity** is the base BPM entity to build a process. It contains others sub entities that will determine the behaviour of the activity (the implementation: no or subflow, the start mode: manual/automatic, the performer, the performer assignment, the transition restrictions: Split or Join).
- **Task** is a runtime object created as a specific activity type which is also called manual activity. BPM tasks could be managed by an independent module receiving tasks from other applications.
- **Participant** is an actor in a BPM process. The following types are supported: SYSTEM, HUMAN, ROLE. Participants are associated to tasks.
- **Transition** is a dependency expressing an order constraint between two activities. The notion of loop (also called iteration) is also represented via a transition.
- **Variable** (aka BPM Relevant Data in XPDL) is a BPM unit of data. Variables can be local to an activity or global to the process or a package. Nova Bonita supports the following data types: Enumeration, String, Float, Integer, Boolean, Datetime, Performer.
- **Hook** is user defined logic adding automatic or specific behaviour to activities and BPM processes
- **Mapper** is a unit of work allowing dynamic role resolution each time an activity with human task behaviour is created (instantiated).
- **Performer assignment** is a unit of work adding additional activity assignment rules at run time.

Each one of those entities is leveraged by both BPM definition and runtime environments. Definition data, runtime recorded data and archived data are so automatically managed by the engine.

To easily play with those three aspects that characterize BPM entities, Bonita has introduced UUID's (Universally Unique Identifier). Each entity has its own typed UUID that can be used when doing operations at both definition and runtime sides through the Bonita facade API's.

3.2 Package

Package element in an XPDL definition file contains: processes, participants, datafields..... The idea is to put together multiple processes definition in one single XPDL file that will be deployed once in the engine.

These processes can share participants and datafields. Process deployment implies to deploy at least a package (ie. the XPDL file that contains the Package element). Package is the minimal unit of deployment.

The notion of package concerns also the scope of deployed java classes/artifacts (i.e hooks, mappers and performer assignments). If java classes are deployed within a package, they will be visible for all processes included in this package (XPDL file).

Undeployment operation of a package will undeploy all these classes as well. Java classes can also be deployed globally (meaning at BPM server level) and so be acceded by any process/activity of any package. **QueryDefinitionAPI** provides access to deployment and undeployment package related data.

3.2.1 LifeCycle

A package has its own life cycle:

- **Deployed:** When deployment operation is successfully executed (through the Management API), state of the package is **deployed**.
- **Undeployed:** when undeploy() operation is successfully performed, the state of the package becomes **undeployed**. An 'undeployed-package-handler' (refer to section Configuration and Services for more info) is then called. This handler is responsible for storing undeployment related data into the archive/history repository through the default environment configuration.

3.2.2 Versioning

Packaging versioning is fully supported in this version. The Bonita API provides operations allowing to deploy and undeploy different package versions as well as to retrieve useful data from those packages.

The only constraint on regards to the version for the deployment is the following:

- It's forbidden to deploy two times in sequence a package with the same package Id if the version is the same in both packages (XPDL constraint)

3.3 Process

Processes are defined within a package and deployed into the engine by deploying the package.

3.3.1 Process basics

- **Process definition:** contains the BPM definition logic (elements that makes up a BPM). A process definition is instantiated.
- **Process Instance:** represents a specific execution of a BPM process. It may run as an implementation of an activity of type subflow.

3.3.2 Life Cycles (process/instance)

A process has the following life cycle:

- **Deployed:** when the package containing the process is successfully deployed, the process is created into the engine and its state is **deployed**.
- **Undeployed:** when the process is successfully undeployed via the undeployment of the package containing the process its state become **undeployed**.

A process instance has the following life cycle:

- **Initial:** once the process instance has been created, state is set to initial.

- **Started:** when `instantiateProcess()` method of the `RuntimeAPI` is called, firstly the instance is created (**Initial** state) and secondly the execution is automatically started which causes the state of the instance to become **started**.
- **Finished:** when the execution has reached the "bonitaEnd" activity (last activity in a BPM process), the instance state is set to finished.
- **Cancelled:** when the execution has been cancelled by a user.
- **Aborted:** when the execution has been aborted by Bonita. Bonita can abort executions when the process contains XOR Join or multi instantiation activities.

3.3.3 Process definition & process Instances

Common BPM scenarios are focused on the re-use of a process definition; in these scenarios, a long-time is spent when defining a generic process model that instantiates in the same way many times. These processes are called business processes (aka process models in Bonita).

A process is a specific definition of a process that may be instantiated multiple times. These processes are based on a process-instance BPM paradigm.

Processes are created into the engine via the **ManagmentAPI** (deployment operations giving an XPD file). **Java DefinitionAPI** allowing the creation of a process via a java API is not yet supported for this release..

When the process is created, the BPM users are able to instantiate the BPM process via the **RuntimeAPI** to create process instance(s), execute assigned tasks... Once the process instance(s) are created, BPM participants can access the **QueryRuntimeAPI** to accomplish the following: obtain their "ToDo", "done", "suspended" lists, or access the **QueryDefinitionAPI** to get definition (as a complement of runtime data) informations about process, activities, tasks, instances, participants A process keeps track of all its instances. BPM Instances related data can be obtained through the **QueryRuntimeAPI**

Bonita instantiation mechanism:

At process instantiation time a new process object issued from a deployed process definition is created. This object is initialized with definition elements and specific parameters such as variables. A root execution object pointing to the process object is created and started. It causes the execution to point to (and enter into) the first activity of the process. The root execution references a `ProcessInstance` object representing the runtime data being recorded in the journal all along the life of the process instance.

3.3.4 Versioning

Versioning of processes is fully supported. The only constraints on regards to the version for the deployment is:

- It's forbidden to deploy two processes with the same process id in the same package even if the versions are not the same (XPD constraint)

3.3.5 Concept of Hooks

Hooks are user-defined logic that can be triggered at some defined point in the process life cycle. Process Hooks **are not supported in this version**.

Process hooks types are:

- OnInstantiate hook is called when a BPM instance is created. The OnInstantiate hook is not considered to be in the same transaction as the process instantiation action.
- OnFinish hook is called automatically after BPM instance termination ends.
- OnCancel hook is called automatically when a user or a BPM administrator decide to cancel a BPM instance

3.3.6 SubProcesses

Sometimes, an independently existing business process can take part in another more sophisticated process.

Instead of redefining the activities, edges, properties, and hooks in the parent process, the independent process may run as an implementation of an activity of type subflow. As the execution logic is inside the subProcess, the subProcess activities are started and finished automatically by the BPM engine according to the subProcess state.

Creating a SubProcess Activity: When a subProcess activity is defined in the process, a specific activity with subflow behavior is created (process id of the process, local variables, in/out/in-out parameters of the sub...).

Instantiating a Process with a SubProcess Activity: At runtime, the execution enter into the subflow type activity, the following operations are done:

- An instance of the process referenced by the subflow activity is created.
- A new root execution is created into this instance and is automatically started (then execution enters in the first activity of the subflow).
- Local variables of the subflow activity (defined through extended attributes in XPDL 1.0) are created as global variables of the instance in the subflow (this is the default way to pass variables to a subflow when processes are defined using the **ProEd editor**). At the end of the subflow execution, global variables are automatically propagated to the parent process as local variables of the subprocess activity.
- If both **formal parameters** into the subflow process and **actual parameters** into the subflow activity have been defined, the list of actual parameters are mapped to the formal parameters (these XPDL definitions are supported by Bonita engine but not yet by ProEd editor).

3.4 Activities

Activity types in Bonita are the following: Task, Subflow, Route, and Automatic (detailed below). When the execution enters into the activity it executes the logic (behavior/type) of this activity.

3.4.1 Activity Basics

The activity is the basic unit of work within a process.

Bonita engine is supporting all kinds of activities specified within XPDL 1.0. Those activity types are the following:

- **Manual activity** (startMode = manual, Implementation = No): When the execution enters into a manual activity a **task** object (aka human task or user task) is created. QueryRuntimeAPI allows to get access to the task according to the task state. RuntimeAPI allows to manage the task state (start, suspend, resume and finish operations). In further releases tasks could be managed by an external and pluggable task module that will allow tasks creation coming from a BPM engine as well as from any other applications such a forum, an online manager....

- **Automatic activity** (startMode = automatic, Implementation = No) : the activity is automatically executed by the engine.
- **Route activity** (Route element): Route are specialized to express Transition Restriction (e.g. SPLIT is automatically executed by the engine).
- **Subflow activity** (implementation = Subflow startMode = automatic): the activity is automatically executed by the engine.
- **BlockActivity** (BlockActivity element): the activity references an ActivitySet (set of activities) and is automatically executed by the engine.

In addition to information determining the activity type, additional informations can be added to the activity definition depending on its type (this data can be accessed via the QueryDefinitionAPI):

- Name and Id: Id is unique within the process.
- Transition Restriction: logic of control for incoming or/and outgoing transitions (e.g. Split and Join). This attribute applies to any activity type. Routes behaviour only relates to Transition restriction.
- Performer: actor defined in charge of the activity (Human or Role)
- Deadline: due date for a particular activity
- Advanced Bonita features defined using extended attributes: local variables, role mapper, performer assignment
- Description, documentation, icon Runtime (recorded) data concerning activities is divided in a common part and a body. the common part is represented by an object called "ActivityInstance". This object is returned by operations included in the queryRuntimeAPI. The "Body" relates to the specific behaviour of each activity (TaskInstance, SubflowBody, RouteBody and AutomaticBody are the types supported) Route and subflow activity types can't execute hooks. Only Tasks and Automatic activities types are able to execute them. Refer to section Hooks here below for more details about hooks usage.

3.4.2 Activities life cycle

Bonita activities share the following states types:

- **Initial:** This is the default state of activity that are not yet ready to be executed
- **Ready:** This is the state of an activity ready to be started. There are two possible situations for this state to occur:
 - Activity for which there is no incoming transitions
 - Incoming connected activities of a particular activity are successfully finished and transition conditions were evaluated to true
- **Executing:** An activity under execution.
- **Finished:** An activity that has successfully finished.
- **Cancelled:** when the activity execution has been cancelled by a user.
- **Aborted:** when the activity execution has been aborted by Bonita. Bonita can abort activities when the process contains XOR Join or multi instantiation activities.

3.4.3 Life cycle for tasks (aka manual activity)

In addition to previous states, manual activities (known as tasks) also add the following one:

- **Suspended:** An activity having that was either in Ready or Executing state that has been suspended. Resume operation put back the activity with its initial state.

3.4.4 Transition between Activities

Most of the usual transition patterns can be achieved through Nova Bonita BPM. There are no special activities to achieve these patterns; however, any activity can behave as a routing node (obviously the case for Route activity). The transition pattern depends on the Transition Restrictions (e.g. Join, Split) and transition conditions defined. Transition Restrictions can be one or both of the followings:

- Join: describing the semantics of an activity with multiple (≥ 1) **ingoing** transitions;
- Split: describing the semantics of an activity with multiple (≥ 1) **outgoing** transitions.

For **Join**, possible types are:

- **AND** (also known as "synchronization" pattern): the activity is not initiated until the transition conditions on all incoming routes evaluate to true.
- **XOR** (also known as "cancelling discriminator" pattern): No synchronisation is required. the activity is executed when the first execution enter the activity. Other incoming branches are aborted.

For **Split**, allowed types are:

- **AND**: the number of child executions that will be created depends on the number of outgoing transitions and the conditions (evaluated to true) associated with each transition.
- **XOR**: A single transition route is selected. when evaluation the conditions on the outgoing transitions the first one evaluated to true is taken.



Note:

The current version of Nova Bonita Designer, only generates Join types . If more than one outgoing transitions is set (without Split within the Transition Restriction), an implicit Split And is constructed by the engine.

The transition patterns can be refined by defining conditions in edges between activities. A condition operates on the value of one or more variables of activities, and is expressed in Java. Any java expression statement is valid.

Assuming that the variable "Var" is defined for a given activity, any of the following constructs is a valid condition: `(str1.compareTo("initial value") == 0) && (enum1.compareTo("yes") == 0) && (float1.compareTo("1.0") == 0) && (int1.compareTo("123") == 0) && (boolean1.compareTo("true") == 0) && (date1.compareTo("2008-09-25T13:14:58") == 0)`

3.4.5 Iterating Activities

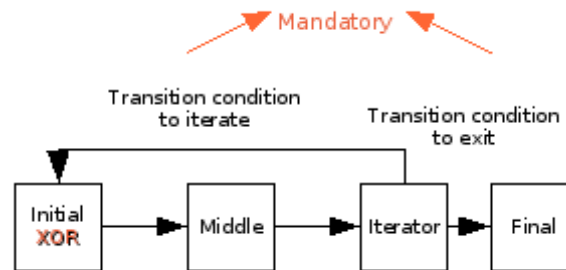
Bonita supports both structured (one entry and exit point in the cycle) and arbitrary cycles/iterations. Nova Bonita iteration feature is natively supported through the use of XPDL Transition element: a loop may be represented via a transition that returns to an activity that was on a path that led to the transition.

The following Guidelines explain how to design iterations in Nova Bonita:

Premise: It is not possible to continue execution inside iterations and exit at the same time.

- All transitions exiting from a node starting the iteration must meet a condition. If there is more than one transition for exiting from that node, all transitions must meet a condition.

Following figure illustrates a typical design that matches our model:



To guarantee the premise, the iteration condition and edge condition must be exclusive. This means that when one is true the other is false. Only iterations from Iterator to initial are possible. This constraint is checked at runtime by Bonita.

Conditions can be a group of conditions like: $((...)) \ \&\& \ (...)) \ || \ (...))$. (Remember: only a single iteration between nodes is allowed.) There could be another iteration starting in the Iterator activity going to Middle or to Iterator itself.

A transition condition from Iterator to Final activity is strictly necessary and must be different to the iteration condition. If there are multiple edges outgoing from Iterator to other activities, all of them must meet a condition not equal to the iteration condition (this is necessary to accomplish the above premise)...

Note that:

- The initial node must be XOR type to allow to validate its entry condition either at the first entry and then to continue into the cycle. At each time only one execution can enters into the node.
- If the Iterator point has not Split Xor transition restriction a Warning is produced when deploying the process. Only a Split Xor setting can avoid threads of execution goes both outside the cycle and inside the cycle which is not authorized.
- At each cycle of the iteration, for activities inside the iteration path, new runtime records (ActivityInstance interface) are created/recorded (with distinct iteration id). If iterations have been performed the parameter: *activityInstanceUUID* must be specified into some methods of the QueryRuntimeAPI.

3.4.6 Activity multi-instantiation

Multi-instantiation of activities is a new powerful feature introduced by Nova Bonita.. This feature covers all types of activities previously described in section 4.1. The idea is to determine at runtime the number of instances to create for a particular activity. This feature is really useful in situations in which, at definition time, process designers do not know in advance the number of occurrences of a particular activity to be created.

The principle is based on the execution of an "Multi-Instantiator" class (added to the activity definition) that returns an object containing:

- A list of values which size is determining the number of instances to be created and so executed. This list of values is used to set for each created activity instance a dedicated activity variable (this activity variable is also added to the definition of the activity);
- The number of finished instances expected to take the transition (called joinNumber). This number must be **greater than 0** and **lesser than or equal to** the number of created instances. When this number is reached, remaining instances are aborted

The condition on the outgoing transition must not contain local variables.
Here after an example of multi-instantiation definition within activity definition.

```
<Activity Id="Approval" Name="Approval">
-.../...
  <ExtendedAttribute Name="MultiInstantiation">
    <Variable>performer</Variable>
    <MultiInstantiator>org.ow2.bonita.example.aw.instantiator.ApprovalInstantiator</
      MultiInstantiator>
    </ExtendedAttribute>
  <ExtendedAttribute Name="property" Value="performer" -/>
-.../...
</Activity>
```

The class given into the definition must implement the interface: MultiInstantiator. Refer to the javadoc API for MultiInstantiator interface (org.ow2.bonita.definition package).

3.4.7 Concepts of hooks/connectors

Hooks in Nova Bonita BPM context are external java classes performing user-defined operations. Hooks may be called at different moments in the activity lifetime. Only two types of activity allow setting of hooks: manual (task) and automatic activity. Hooks are prefixed by the type of activity for which they are associated.

In activities of type task, hooks may be called at different moments of the task lifecycle:

- Task:onReady: is called when the task becomes available.
- Task:onStart: is called as soon as the task is started.
- Task:onFinish: is called as soon as the task is finished.
- Task:onSuspend: is called when the task is suspended by a user.
- Task:onResume: is called when the task is resumed from a suspended state.

In automatic activities, hooks can only be called at one moment (when the activity is executed) :

- automatic:onEnter

Refer to the Development Chapter of this document to get more details on hooks (writing, compiling, deploying hooks as well as samples).



Note:

For deadline feature, the name of the class that implements the hook interface is specified within XPDL Deadline element. The event name (aka Hook type is called ON_DEADLINE).

3.4.8 Activity/Hooks and Transactions

Hooks are always executed within the transaction involved in the last activity change state (i.e. instantiateProcess, startTask, finishTask, suspendTask...). Transaction can involve more than one activity in synchronous executions (typically a task connected to one or more automatic activities). If the execution of the hook raises an exception, it will abort or not the transaction depending on the implemented interface (Hook vs TxHook):

- **Hook** interface, if an exception occurs it is caught by the engine and no rollback is performed. Hook interface is intended to execute methods of APIs that are allowed to perform read/query operations: QueryDefinitionAPI and QueryRuntimeAPI.
- **TxHook** interface, if an exception occurs it is raised by the engine and the transaction is not committed (rollback). TxHook interface is intended for executing Bonita APIs operations that are related to write/ set operations: RuntimeAPI, ManagementAPI, DefinitionAPI (not yet supported), CommandAPI.

This interface should also be used to call business logic in which transactions are involved. Refer to the javadoc on API for Hook/TxHook interfaces (org.ow2.bonita.definition package) as well as the developpementGuide for more details on Fault Management and on Activity/Hooks and Transactions features.

3.5 Role Mappers Feature

3.5.1 Overview

This feature can be added in the XPDL definition as an extended attribute inside the **"Participant" element of type Role**.

```
<Participant Id="manager" Name="manager">
  <ParticipantType Type="ROLE" -/>
  <ExtendedAttributes>
    <ExtendedAttribute Name="Mapper" Value="Custom" -/>
    <ExtendedAttribute Name="MapperClassName"
      Value="org.ow2.bonita.tests.functionnal.mappers.AdminRoleMapper" -/>
  </ExtendedAttributes>
</Participant>
-..../....
<Activity Id="myTask" Name="myTask">
  <Implementation>
    <No -/>
  </Implementation>
  <Performer>manager</Performer>
  <StartMode>
    <Manual -/>
  </StartMode>
-..../..
..
```

This feature is dedicated to manual activities (human tasks) where a "Performer" (element of XPDL Activity) referencing such type of participant has been defined. When the task runtime is created the mapper feature is executed allowing the dynamic resolution of the participant that will be assigned to the activity.

A **list of candidates users** for the task is filled-in with the returned values of the role mapper. In the case that the mapper is an 'Initiator Instance' mapper (default implementation) the user that creates the instance is assigned to the task (this default implementation is useful for testing purposes).

Two types of mappers are available, depending on the method employed to retrieve users in the system:

- Calling a java class to request a user's base (**Custom** mapper).
- Getting the initiator of the process instance (**Instance Initiator** mapper)

Role Mappers can be defined the ProEd editor application. The Bonita API allows to retrieve related data of performers defined in a XPDL file. The QueryDefinitionAPI allows to get the role mapper definition (getProcessParticipant operation).



Note:

- The execution of a mapper for a particular participant is performed each time a task is created (only for tasks having this performer assigned)
- If there is no role mapper defined for participant of type role, no assignment of task is done (meaning, the task is not yet assigned, but it could be done afterwards through the API)

3.5.2 Custom Mappers

This mapper type allows to perfectly match with the users-roles mappings and constraints available in organizations. When this type of mapper is selected, a call to a java class is performed. This java class must implement the "RoleMapper" interface. In particular the "searchMembers" method of this interface must be implemented and will return the collection of expected users (see the javadoc of interface RoleMapper).

The class name is specified into the extended attribute with Name MapperClassname. Refer to Development Chapter to get more details on the practical steps to follow to define and deploy mapper classes into Nova Bonita (developping, compiling and deploying steps)...

3.5.3 Instance Initiator

This type of mapper fills in the candidates list of a task with the user that created the BPM instance (based on the authenticated user that initiates the instance). This user is able to perform operations to this the task.

3.6 Performer Assignment

3.6.1 Overview

This feature extends first assignment rules for tasks that was done through mappers. Mappers resolutions can assign a task to a list of possible candidates users (those ones are able to see and perform operations over the task). This list of candidates can be refined for each particular activity through the use of performers assignments elements. Depending on the type of performer assignment the following functionalities can be added:

- Assign the activity (only tasks) to a user by calling a java class in charge to perform a user selection from the list of candidates (**Custom** performer assignment).
- Dynamically assign the activity to a user by using the value of a variable that has been previously set with the selected user Id (**Variable** performer assignment).

Once the performer assignment has been performed, the task is assigned to the selected user. This feature can be added though the ProEd editor application. QueryDefinitionAPI allows read access to the definition of the performer assignment.

3.6.2 Custom Performer Assignment

The java class must implement the "PerformerAssign" interface. In particular the "selectUser" method of this interface must be implemented. The return value of this method is the name of the selected user (see the javadoc of interface PerformerAssign).

The class name is specified in an XPDL "extended attribute" called Name PerformerAssign. Refer to the section Performer Assignments in the developpementGuide to get details on the practical steps to deploy and define performerAssign classes into Nova Bonita (developing, compiling and deploying steps). Notice that the candidates list is passed to the "selectUser" method to simplify the user selection. Of course other strategy could be considered.

3.6.3 Variable Performer Assignment

With this type, assignment depends on the value of a variable previously set.

Chapter 4. Configuration and Services

This chapter introduces the services configuration infrastructure provided by Nova Bonita as well as main services included in this 4.1 version.

4.1 Services Container

The Process Virtual Machine technology includes a services container allowing the injection of services and objects that will be required during BPM definition and execution. Objects and services used by the Bonita engine are defined through a XML file. A dedicated parser and a wiring framework are in charge of creating those objects. Security, identity, persistence, notifications, human task and timers are examples of pluggable services.

This services container (aka IoC container) can be configured through a configuration file. A default configuration file is included in the package under the /conf directory (bonita-environment.xml):

Currently, following objects implementations can be injected in the environment:

- **repository:** data repository storing BPM processes, instances, activities... Db persistence (class `org.ow2.bonita.repository.db.DbRepository`) implementation is included in this version.
- **recorder:** object responsible of BPM execution logs. Default implementation handles BPM logs in the command line console (`org.ow2.bonita.persistence.log.LoggerRecorder`). Recorder and Journal (see next) objects can be chained (new ones can be added as well on top of the recorder chainer). This give you a powerful mechanism to handle BPM execution data
- **journal:** object responsible for storing or retrieving BPM execution data. Db persistence (class `org.ow2.bonita.persistence.db.DbJournal`) implementation is provided by default.
- **archiver:** object intended for BPM logs archiving. Default implementation handles logs on BPM data archiving through the default implementation (class `org.ow2.bonita.persistence.log.LoggerArchiver`). Archiver and History (see next) objects can be chained (new ones can be added as well on top of the archiver chainer). This give you a powerful mechanism to handle BPM archived data
- **history:** object intended for storing or retrieving BPM archived data. Default implementation is provided and available in the following class: `org.ow2.bonita.persistence.db.DBHistory`. This class will store the BPM history in a relational database (commonly a dedicated database)
- **queryList:** object intended to configure how the QueryRuntimeAPI will retrieve the BPM execution data. This retrieval could be configured to chain with the expected order into the journal and the history.
- **finished-instance-handler:** action to perform when a BPM instance is finished. This object could chain two distinct actions: for a given BPM instance, deleting the runtime object including its tasks from the repository and then store data in the archive and remove data from journal. Default implementations are proposed for both chained actions.

- **undeployed-package-handler** action to perform when a BPM package is undeployed. Default implementation is proposed allowing to store undeployment related data into the archive
- **security**: object implementing the security strategy in Bonita. By default 4 different implementations are provided: `org.ow2.bonita.facade.AutoDetectSecurityContext`, `org.ow2.bonita.facade.StandardSecurityContext`, `org.ow2.bonita.facade.EJB2SecurityContext` and `org.ow2.bonita.facade.EJB3SecurityContext`. Those implementations are based on JAAS security.

AutoDetect meaning that Bonita will make the choice for you on the one to be used (i.e if your Bonita is deployed in Jboss 4.x is going to be `EJB2SecurityContext`).

For users that don't want to use JAAS security you can just add your own implementation of the `BonitaSecurityContext` interface. This interface just requires the implementation of the `getUser` operation so you can easily plug your own security in Bonita by leveraging this mechanism.



Note:

- The environment is divided in two different contexts: application and block. Objects declared inside the application context are created once and reused while objects declared inside the block context are created for each operation.
- As explained before persistence objects are provided as default implementations in the environment. Notice that in a persistence configuration additional resources are required, i.e for hibernate persistence you can specify mappings, cache configuration...

4.2 Services

Services in Nova Bonita is all about pluggability. Standard (StandAlone Java based) and Enterprise (JEE Server based) versions of Nova Bonita can be easily configured thanks to the services container. To allow that, each BPM related service has been thought in terms of an interface with different possible implementations. In the following lines you will find a description of main services supported in Nova Bonita:

4.2.1 Persistence

Persistence is one of key technical services injected into the services container. This service, as well as other major services in Nova Bonita, is based on a service interface. That means that multiple persistence implementations can be plugged on top.

The Persistence service interface (called `DbSession`) is responsible to save and load objects from a relational database. By default, a persistence implementation based on the Hibernate ORM framework (called `HibernateDbSession`) is provided (JPA and JCR would be other examples of persistence implementations).

The Process Virtual Machine core definition and execution elements (processes, nodes, transitions, events, actions, variables and executions) as well as the XPDL extension ones (join, split, manual and

automatic activities, conditions, variables...) are persisted through this service. Process Virtual Machine core elements are also cached by leveraging the default persistence service implementation (Hibernate based). BPM packages, processes, instances, tasks and advanced classes (such hooks or mappers) are stored through this persistence service. BPM repository is the term used in Nova Bonita to store those entities.

4.2.2 Identity

Identity service main objective is to give freedom to system administrators to leverage their favorite organization user repository. Traditional user repositories such LDAP, ActiveDirectory as well as any other user repository (database or API) can be plugged as implementations of this service.

By default, some user repositories implementations are provided for testing purposes: in memory, basic FileSystem based persistence, and basic database persistence (based on a predefined database schema).

Those implementations can also be used in production if there is no other user repository available.

The Identity service is so an extensible interface (known as IdentityServiceOp) build around three main concepts: Users, Groups and Memberships:

- User: a particular user inside an users repository. Users can be created, modified, removed and queried (some of those operations could be not allowed for some repositories (i.e LDAP) through the IdentityService API).
- Group: a group of users in a particular users repository. A group could contain either users security restrictions or hierarchical information. As for users, groupes can also be created, removed, modified and queried.
- Membership: a membership represents a user position in a particular group. An user could have two different membership in two different groups. Membership related operations concern set, remove or updates on users position inside groups.

Both Security and Human Task services will use the Identity one by checking user login/password and user rights (Security) and by resolving BPM logical roles with users and so to assign manual activities to users based on some hierarchical information (Tasks Management)

By default, Nova Bonita is packaged with a test based identity module based on a properties file. This file contains the user/login allowed to reach Nova Bonita APIs. This properties file is in fact a Test Login Module (see security module description below), meaning that the same properties file is used for security and identity configuration.

4.2.3 Security

The security service is based on JAAS standard. Main purpose of this service is to provide both authentication and authorization capabilities to the BPM engine. As security directly relates to users permissions, this service also relates to the identity one (commonly security is configured on top of the identity service).

As for other services, the Nova Bonita team is concerned on let you the freedom to choose and plug your favorite security implementation. At the same time we also want to provide one ore more default implementations that allow users to quickly set up and start playing with Nova Bonita.

For testing purposes Nova Bonita includes a default JAAS login module checking user/password values stored in a file. This easily allow to start playing with Nova Bonita in a testing security environment in which the login module acts as a lightweight users repository. This login module (org.ow2.novabpm.identity.auth.PlainLoginModule) is the one provided in Nova Bonita examples directory.

The current implementation of the security service allows you directly work with the default identity service to handle users authentication. Users must login before start calling the Bonita APIs. The Security service is composed by two different JAAS LoginModules. The first one (called

PlainLoginModule) is responsible to handle security authentication and authorization. This one could just be replaced by you favorite JAAS Login Module.

The second one (StorageLoginModule) is responsible to keep data of authenticated users (basically for security context initialization). Those login modules can be configured in both standard and enterprise environments (note that most of JEE servers already provides a Storage Login Module so you could just replace the one proposed by Nova Bonita by the one leveraged by you app server). Some examples of security configuration files for both standard and enterprise environments are included in the Bonita distribution (under the /conf directory).

4.2.4 Task Management

Task management is all about providing the right information to the right people at the right time !. This is one of the most important services that must be provided by a BPM solution.

As human task management can be re-used in other domains (not only by BPM solutions but by any Java based application) we wanted those features to be a service rather than an internal BPM module. As a result, this service is generic and extensible Task Management service that can be either used in Nova Bonita extension to handle manual task assignments and executions or either by any Java application or Domain Specific Language (i.e BPEL4People extension for instance).

Traditional features such users - roles/group mapping, delegation, scalation, task deadlines handling or manual activities execution life cycle are in scope of this service. Advanced features such configurable activity life cycle, interactions with other task managements system, services or collaborative applications and integration with organizational rules are also part of the main responsibilities of this service.

The current implementation focus on support of manual tasks (also known as manual activities) in Nova Bonita. Basic features such Bonita RoleMappers and Performer Assignments entities allowing users – roles mapping are already supported. Together with the identity and security service, users can login into the system, get their tasks todo list and execute them. As other service in Nova Bonita this module is executed in a persistent environment.

4.2.5 Journal and History

This module concerns the way in which the BPM data is stored during the BPM execution and archived when the execution is completed. This is indeed a crucial module in a BPM solution.

While in Bonita v3 journal data (aka execution BPM data) and history data (aka archived data) were handled by different mechanism, in Nova Bonita we decided to unify them as the underlying essence of both is to handle BPM data. For that to be done, we created the concept of BPM record. A record is a minimal set of attributes describing a BPM entity execution. That means that each BPM entity related to the execution has its own associated record: instance record, task record, hook record...

Those records are recorded during the BPM execution and stored depending on the persistence service implementation (db, xml...). The Nova Bonita API will retrieve record data from the records storage and sent them back to the users (meaning that records also acts as value objects in Nova Bonita APIs).

As soon as a BPM instance is finished, a typical scenario would be (by default) to move instance related BPM data from the production environment to a history one. While the physical device and the data structure could changed from one BPM engine deployment to another (XML, BI database...), the internal format could remain the same (records). This is exactly what is happening in Nova Bonita, when archiving data the engine just move execution records from the production to the history environment without data transformation inbetween.

4.2.6 Timers

To handle activities deadlines, a timer service is required that can schedule timers to be executed in the future. Timers must have the ability to contain some contextual information and reference the program logic that needs to be executed when the timer expires. A typical scenario would be a manual activity (task) that needs to be monitored with a timer. For example, if this task is not completed within 2 days, notify the manager.

This service, as well as any other asynchronous service in Nova Bonita is based on the Process Virtual Machine Job executor framework. Job executor framework is responsible for handling jobs. A job could be a timer scheduling or an asynchronous message for instance. When a job is created and stored in the database, the job executor starts a new transaction, fetch the job from the database and perform the instructions contained in the message.

Chapter 5. Installation

Nova Bonita V4 adds support for both standard and enterprise deployments. After unzipping this release you could easily use Nova Bonita "as a library" inside your web or rich client application or to deploy it into you favorite application server and use it remotely.

For a detailed information about deployment configurations supported by Bonita please see Chapter 2 (Prerequisites) of this guide.

5.1.1 Distribution

So, first of all you should start by unzipping the Bonita distribution package:

```
>unzip bonita-4.1.1.zip
```

A new directory bonita-4.1.1 will be created with the following structure:

```
README
build.xml
build.properties
License.txt
release_notes.txt
conf/
doc/
javadoc/
examples/
ear/
lib/
```

Let's describe those items :

- **README:** This file gives the basic information related to Nova Bonita
- **build.xml:** This file is an "ant" file (aka makefile) that provides tasks to deal with Nova Bonita administration operations (detailed commands instructions are given in following sections).
- **build.properties:** This file contains the J2EE properties required to deploy and to use Nova Bonita APIs deployed in a remote J2EE server (JOnAS, Jboss and EasyBeans properties are provided by default allowing to execute Nova Bonita samples remotely).
- **License.txt:** The license of Nova Bonita. Bonita is released under the LGPL license v2.1.
- **conf/:** This directory contains default configuration files for Nova Bonita. That includes "environment" xml files (including services and objects used as default by the engine), login modules configurations (JAAS compliant login modules samples) and hibernate persistence configuration (as a default implementation to handle Nova Bonita persistence). Standard (JSE) and Enterprise (JEE) versions are provided for JBoss and JOnAS application servers as well as with Easybeans EJB3 container
- **doc/:** This directory contains Nova Bonita documentation
 - Cookbook.pdf: This is the document that you are actually reading. The bonita cookbook gives a detailed overview of features, architecture as well as installations and configurations instructions.
 - quickStartGuide.pdf: This guide covers main features of Nova Bonita, and will help you to get started right away. This document is intended for users looking to get started quickly on Nova Bonita runtime and graphical tools.

- **javadoc/:** This directory contains developer's javadoc documentation of Nova Bonita. This javadoc describes in details Nova Bonita APIs.
- **lib/:** This directory contains the libraries used in Nova Bonita v4. Nova Bonita can be integrated in your application/IS in different ways (integrated in a web application, inside a rich client application, remotely deployed in a JEE application server...). Depending on your integration environment only some of those libraries will be required.
- **examples/:** This directory contains BPM examples provided with Nova Bonita package. Those samples applications illustrates how to use Nova Bonita APIs from within a client application. That includes the process definition (XPDL) files, java related BPM artifacts (Hooks, mappers and performer assigns) and client applications which illustrates how to deploy, execute and query BPM processes through Nova Bonita APIs in both JSE and J2EE environments.
 - Carpool sample: This example is a carpool simulation in which requesters and publishers are put in relationship to each other. This process illustrates the way in which deadlines and asynchronous services can be leveraged in Nova Bonita.
 - WebSales sample This example is a web sale simulation process in which a customer and an online shop agent/employee are involved in a purchase request process. In this sample iterations/loops as well as multiple types of variables are illustrated.
 - Approval Workflow sample: This is a generic Approval Workflow process. Two versions of this process exists. One with a single approval performed by the manager (which decides whether he accepts or rejects an hypothetical request) and an other involving the multi-instantiation for the approval step followed by a "CheckDecision" step (to decide what should be the decision). Both sample applications show how hooks entities can be used by both manual and automatic activities. The version with multiinstantiation illustrates additional features like activities multi-instances coupled with a performer assignment.

An advanced version of this sample is also included. In this version Nova Bonita engine is packaged in a .war file together with the approval workflow sample and a simple web application. This sample illustrates how simple it is to embed Nova Bonita in a web application running in a web container (i.e Tomcat). See "Java Properties" chapter below for more details about Tomcat configuration for Bonita

5.2 Standard vs Enterprise installation

Find hereafter some instructions about how to deploy and to reach Nova Bonita in both Standard and Enterprise environments.

5.2.1 Standard installation (Bonita as a library)

To use Bonita embedded in your java application just add the libraries located under /lib/server directory in your application classpath. The main library allowing to deal with Bonita is: **bonita-server.jar**

5.2.2 Enterprise installation (Bonita as a server)

This is intended for BPM deployments in which Bonita is going to be deployed in a dedicated server, meaning that different applications will reach Bonita remotely. In those cases, client applications only requires one bonita library called: **bonita-client.jar**

In this deployment configuration the Bonita server will often be deployed in a application server. Hereafter you will find the instructions to deploy Bonita as a BPM server.

Move to the Nova Bonita installation directory and:

1. call "ant ear.ejb2", "ant ear.ejb3" respectively tasks to generate the bonita.ear file corresponding to either JEE 1.4 or 1.5 specification.
2. deploy this ear into your favorite JEE 1.4 or 1.5 application server.



Note:

If you are using Jboss or JOnAS application servers or the EasyBeans EJB3 container you can directly deploy and start using Nova Bonita examples. Specific descriptors and classpath configurations for those servers are included in this distribution. In case you are using another JEE 1.4 application server (Weblogic, Websphere, Oracle, Geronimo, Glassfish...) just add specific descriptors for those application servers into the bonita.ear file and configure your client side to reach the Bonita APIs (take as example existing configurations for JOnAS and Jboss).

bonita.ear file generated through ant "ear.ejb3" task can be deployed in any EJB3 compliant application server. in this version of the specification, standard descriptors should work in any EJB3 environment.

5.2.2.1

Jboss 4.x and 5.x installation and deployment

Find hereafter required steps to deploy and run Nova Bonita in Jboss apps:

1. Download Jboss 4.x or 5.x from Jboss web site: <http://www.jboss.org> and follows jboss application server installation instructions
2. Edit build.properties file and set your Jboss configuration settings: URL provider (localhost with port 1099 by default) and set "jboss.home" and "jboss.client" properties values (those properties must be initialized with the path corresponding to the directory in which Jboss was installed and the path in which jboss client libraries are available).
3. Type "ant ear.ejb2" or "ant ear.ejb3" (depending on the jboss version you want to use) under your Bonita installation directory to generate the bonita.ear file
4. Copy the bonita.ear file generated into JBOSS_HOME/server/default/deploy directory (default configuration)
5. Start the Jboss application server by executing run.bat or run.sh under JBOSS_HOME/bin directory.
6. Bonita should be deployed at that time. Bonita is now up and running under Jboss

5.2.2.2

JOnAS 4.x and 5.x installation and deployment

Find hereafter required steps to deploy and run Nova Bonita in JOnAS apps:

1. Download JOnAS 4.x or 5.x from JOnAS web site: <http://jonas.ow2.org> and follows JOnAS application server installation instructions
2. Edit build.properties file and set your JOnAS configuration settings: URL provider (localhost with port 1099 by default) and set "jonas.root" and "jonas.lib" properties values (those properties must be initialized with the path corresponding to the directory in which JOnAS was installed and the path in which JOnAS client libraries are available).
3. Type "ant ear.ejb2" or "ant ear.ejb3" (depending on the JOnAS version you want to use) under your Bonita installation directory to generate the bonita.ear file
4. Copy the bonita.ear file generated into JONAS_ROOT/apps/autoload directory (default configuration)

5. Start the JOnAS application server by executing "jonas start.bat" or "jonas start.sh" under JONAS_ROOT/bin directory. Bonita should be deployed at that time
6. Bonita is now up and running under JOnAS

5.2.2.3 EasyBeans EJB3 Installation and deployment

Hereafter you will find steps required to deploy Nova Bonita (JEE version) in EasyBeans EJB3 container:

1. Download easybeans from v1.0 at <http://www.easybeans.net/xwiki/bin/view/Main/Downloads> (standalone version). EasyBeans is using a directory called 'easybeans-deploy' in the basedir to deploy new archives
2. Create a directory with this name in the folder from where you will start easybeans container and copy the bonita.ear file in to the created 'easybeans-deploy' directory
3. Be sure to have all security permissions in your java.policy file: permission java.security.AllPermission
4. Add novaBpmIdentity.jar in your CLASSPATH environment variable. This jar is available under /lib directory of Nova Bonita distribution
5. Then start easybeans : `java -classpath "lib\easybeans\novaBpmIdentity.jar;easybeans.jar" -Dorg.ow2.bonita.environment=bonita-environment.xml -Djava.security.manager -Djava.security.policy=java.policy org.ow2.easybeans.server.EasyBeans`

Chapter 6. Definition

Like in previous Bonita versions, processes could be created either through a java api or through a graphical editor : Nova Bonita designer (aka ProEd). The java api to build Bonita v4 processes is not yet developed, so processes should be deployed as .xpd files. That can easily be done under ProEd and then imported as xpd files.

6.1 Designing a xpd process with ProEd

ProEd (Process Editor), is a Java program used to define BPM models. The ProEd tool helps in the creation, updates, and visualization of BPM processes.

The ProEd graphics-based tool allows the user to visually describe a BPM process using a single graphical notation inspired from the [BPMN](#) standard (Business Process Modeling Notation) graphic notation. All elements of the BPM can be displayed, such as activities, transitions, iterations, etc. Values for performers, mappers, hooks, etc. can be set at the project or activity level as necessary. Finally, the BPM process can be saved using the XPD standard notation.

The XPD file can be saved locally on the computer workstation or in a file repository. The file repository provides a shared BPM storage location residing on the server.

6.2 Versioning Support in ProEd

ProEd supports versioning of the BPM process. Each BPM process contains an inherent attribute that describes its version.

The version consists of a major version number and a minor version number, and is represented in the conventional decimal notation of MajorVersion.MinorVersion. A new BPM project is created with an initial version of 1.0. If an existing BPM project that does not contain version information is opened, it will be given a version of 1.0.

Whenever the SaveAs operation is performed, the option is presented to increment either the major version or the minor version by one. Saving a BPM project to a new file in this manner is the only way to change the version number.

There is no special format requirement for the name of a BPM process's XPD file; however, the following format is recommended and will be proposed in the dialogs whenever a new file name is required:

BPMProcessName_version.xpd

For example:

MedicalBPM_1.0.xpd

6.3 Starting ProEd and ProEd Modes of Operation

ProEd is available in two different versions: as a desktop application or as an Eclipse plugin. Both versions are available to download at the Bonita forge: http://forge.objectweb.org/project/showfiles.php?group_id=56 (Nova Bonita subproject).

While ProEd desktop application (swing application) is more oriented to analysts, the Eclipse plugin is more intended for developers as they can easily integrate ProEd to their Java development environment.

6.3.1 Launching ProEd as a desktop application

Go to the Bonita download forge, http://forge.objectweb.org/project/showfiles.php?group_id=56 and get the ProEd designer version for Bonita 4.1.1 (bonita-desktopDesigner-4.1.1.zip file).

Unzip this file in your favorite drive and you are done.

In order to execute ProEd desktop application just move to the unzipped directory and type “ant”. The only prerequisite to run ProEd is to install Jakarta Ant version from 1.6.4 (go to <http://ant.apache.org/bindownload.cgi> for downloading this project).

6.3.2 Launching ProEd as an Eclipse plugin

Go to the Bonita download forge, http://forge.objectweb.org/project/showfiles.php?group_id=56 and get the ProEd designer version for Bonita 4.1.1 (bonita-eclipseDesigner-4.1.1.zip file).

In order to install the plugin in your Eclipse environment just unzip this file on your eclipse installation directory. ProEd for Bonita 4.1.1 has been released for Eclipse from 3.2 version.

Once unzipped, just restart your Eclipse, go to “File” menu and then either go to “New->Other ->Bonita BPM->ProEd XPD file” to create a new XPD file into your favorite Eclipse project or either “New->Project->New Bonita project” to create a new BPM project. Please, go to the next sections to know more about those options.

6.4 Creating a BPM/BPM Eclipse project

This feature is available in the Eclipse version of ProEd. Main purpose is to accelerate the process of creating a BPM project in Bonita. The idea is to help BPM and BPM designers on creating .bar files (remember that a .bar file in Bonita is a zipped file containing a BPM definition as well as the list of entities resources required to interact with users and your information system).

To create a new BPM process with ProEd just go to Eclipse File menu and select “New -> Project” feature. If ProEd plugin was successfully installed in your Eclipse environment, the following dialog should appear:

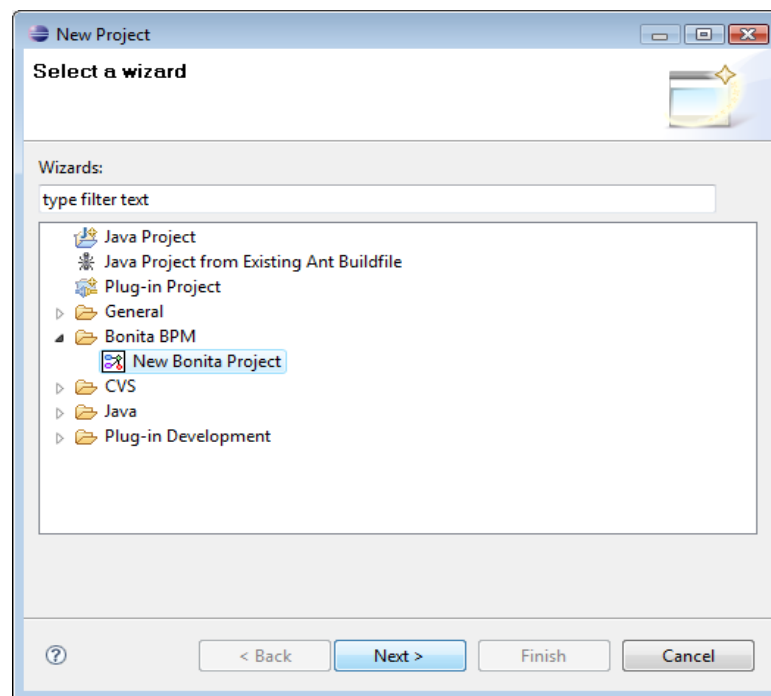


Figure 6-1. Creating a New BPM Project

Then select the “New Bonita Project” feature and click “Next”. In the next dialog you are allowed to introduce the name of your BPM project as well as the directory name in which the .bar file will be generated. Once finished click on “Next” and you will be redirected to the Java Settings dialog.

In this dialog, the wizard will suggest you a default directory structure for your BPM project composed by:

- forms directory: default directory in which forms associated to manual activities will be stored
- java directory: default directory in which you could add java related BPM entities such hooks, mappers, performers assignments and Multi-instantiators.
- xpdL directory: default directory to store xpdL files

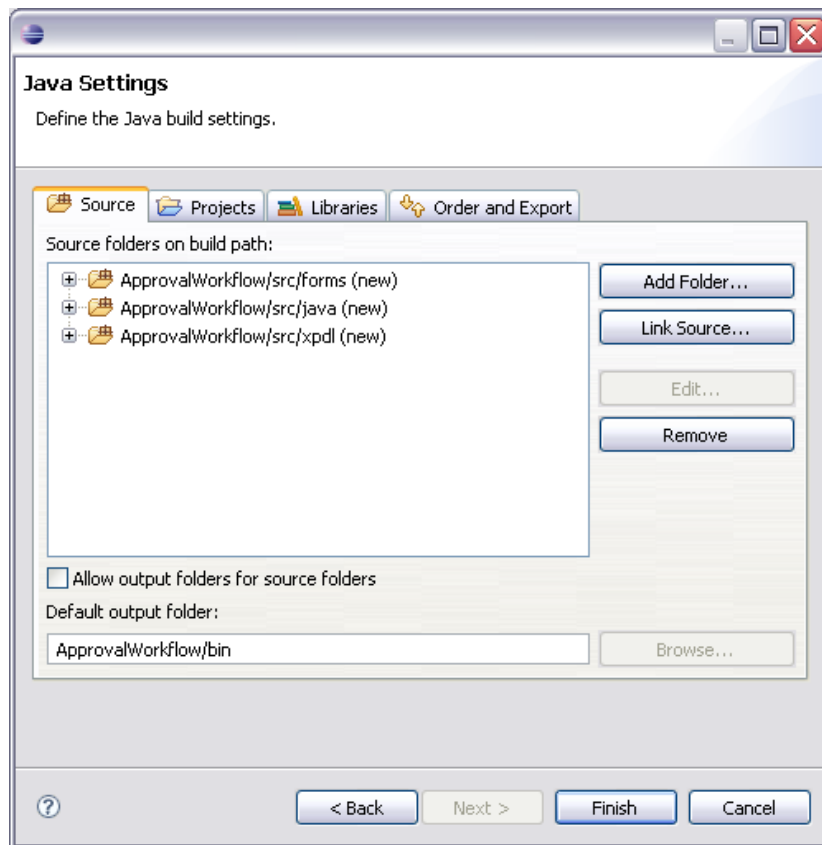


Figure 6-2. Editing Java settings

Click on the “Finish” button when you are done and this wizard will create a new project in Eclipse with the following structure:

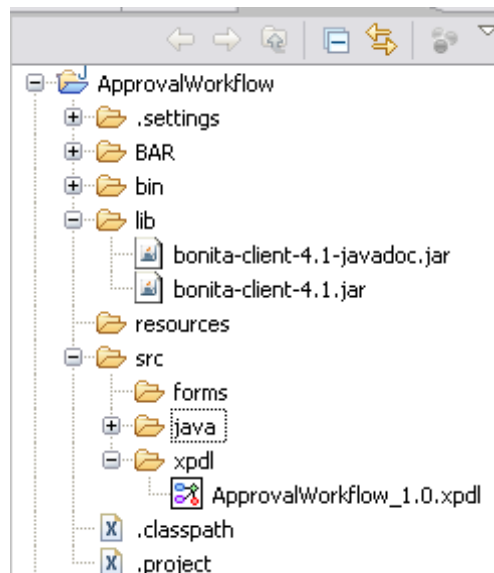


Figure 6-3. BPM project structure

In this sample ApprovalWorkflow was given as the name of the project. As you can see this wizard has automatically added required libraries to automatically compile java related BPM entities.

By default, the previous operation also creates an empty XPDL file associated to this BPM project. If you decided to do not check the “Create a new XPDL file” option you could do that as a separate operation at any time. For that to be done just select the xpd directory and click right button on “New -> Other”. This will show you the following dialog:

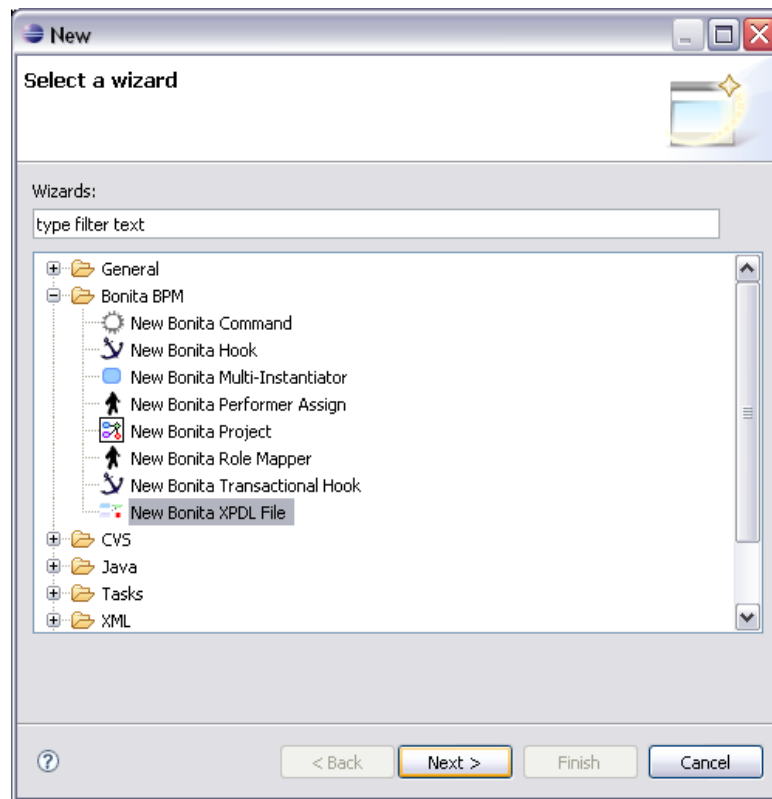


Figure 6-4. Creating a New XPDL file

In this dialog ProEd allows you to automatically create, among other options, a XPDL file (we will describe other features later on). Select so “New Bonita XPDL File” option and click “Next”.

On the next dialog just enter the name of your process (i.e ApprovalWorkflow) as well as a description (if required). This will automatically create an empty XPDL file in your project and will open the ProEd Eclipse editor:

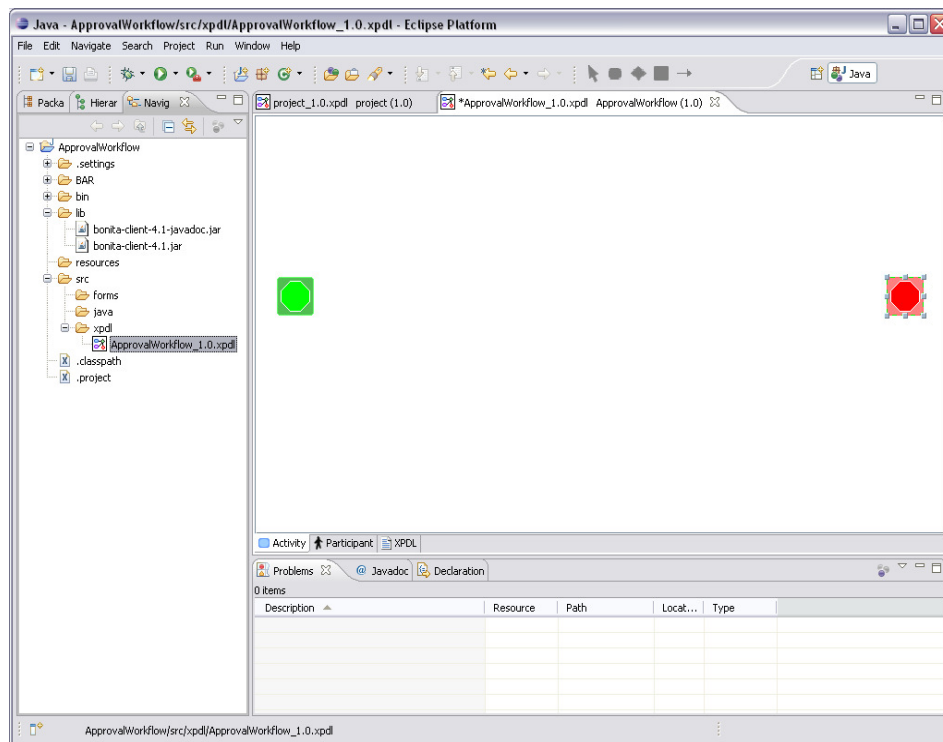


Figure 6-5. BPM process editor

You are now ready to graphically define a BPM process. For a detailed description on how to use ProEd graphical editor please take a look to next chapters.

To illustrate a BPM process definition we will re-use the ApprovalWorkflow sample provided in the Bonita distribution examples directory. After creating 2 manual and 2 automatic activities assigned to 2 different performers, the BPM definition for this process looks like follows:

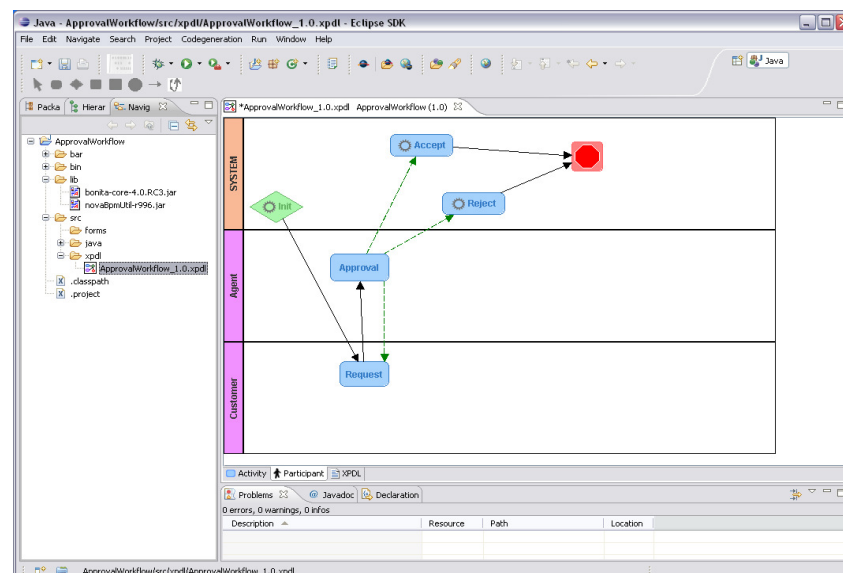


Figure 6-6. BPM process definition through ProEd

Time now to show some other interesting capabilities of this plugin: how to create hooks, mappers, performerAssigns and Multi-instantiators java entities.

Those three java entities can be created in a unified way. Just select the java directory in your BPM project and click right, then go to “New->Other” and select the BPM java entity you want to create:

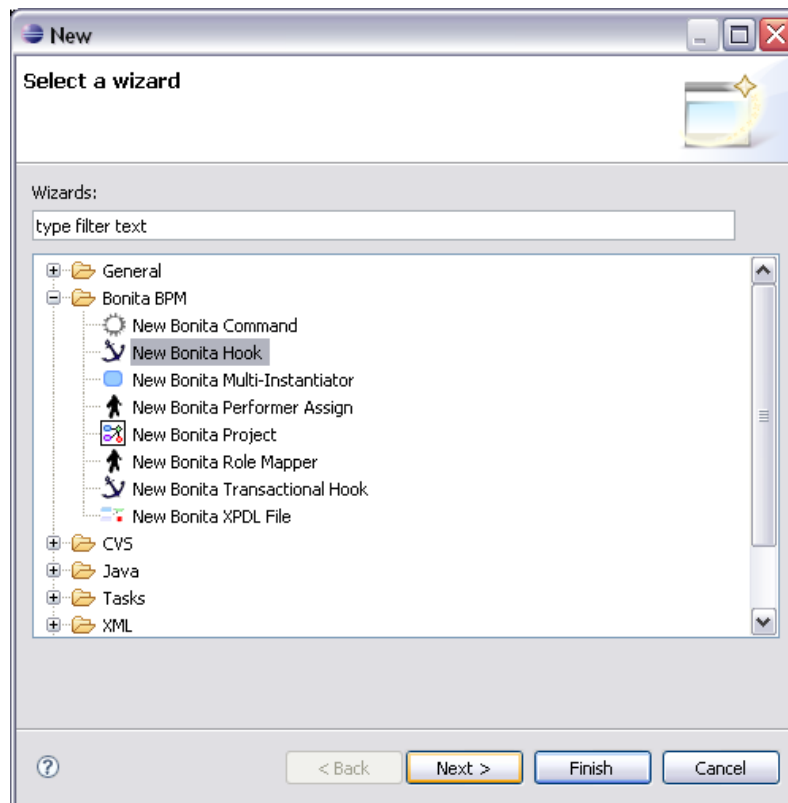


Figure 6-7. Creating a Bonita Hook Java entity

Let's focus on Hooks creation. As soon as you click “Next” on the previous dialog the plugin will move to the Hook creation wizard:

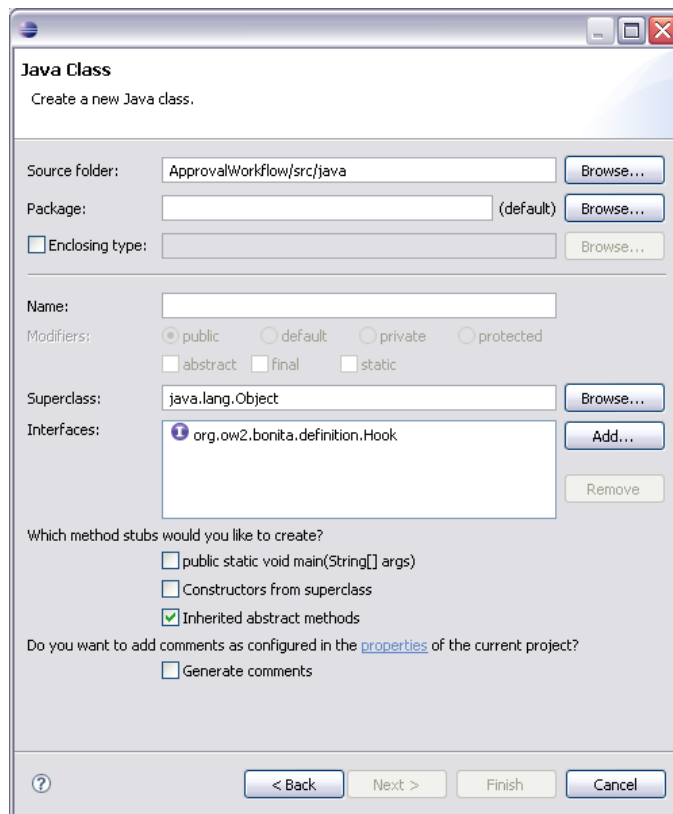


Figure 6-8. Editing Java Hook settings

In this wizard you are allowed to specify the name of your hook as well as the java package in which you want it to be. When you are done, just click on the “Next” button, this will automatically create a Java hook skeleton class:

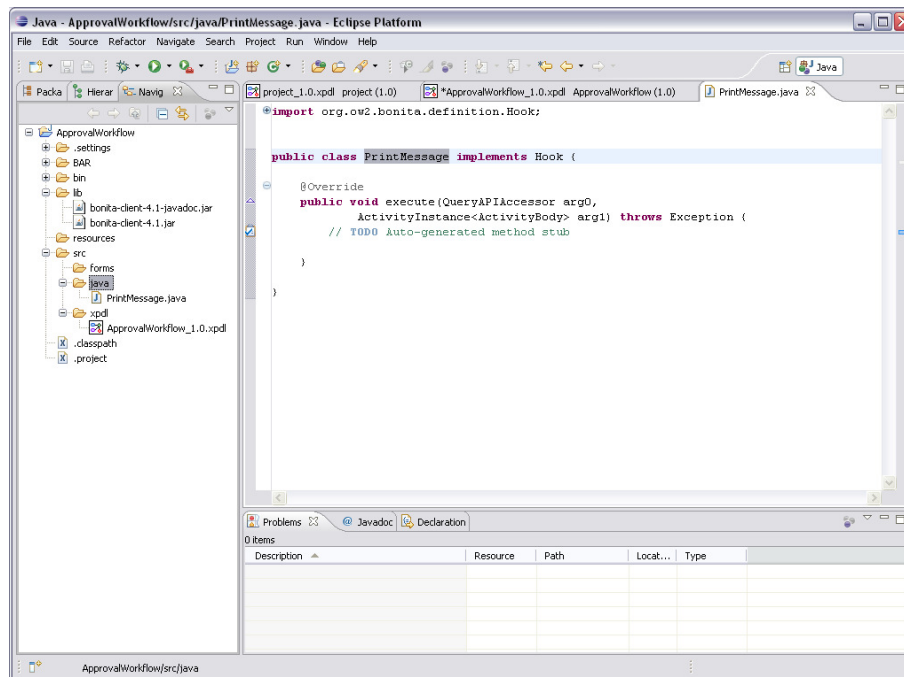


Figure 6-9. Java Hook Preview

In the picture above “PrintMessage” was given as the name of the Hook. Now you are free to implement this class (connector) using your own java code. For instance, we will just “print out” a message as a default implementation of this hook:

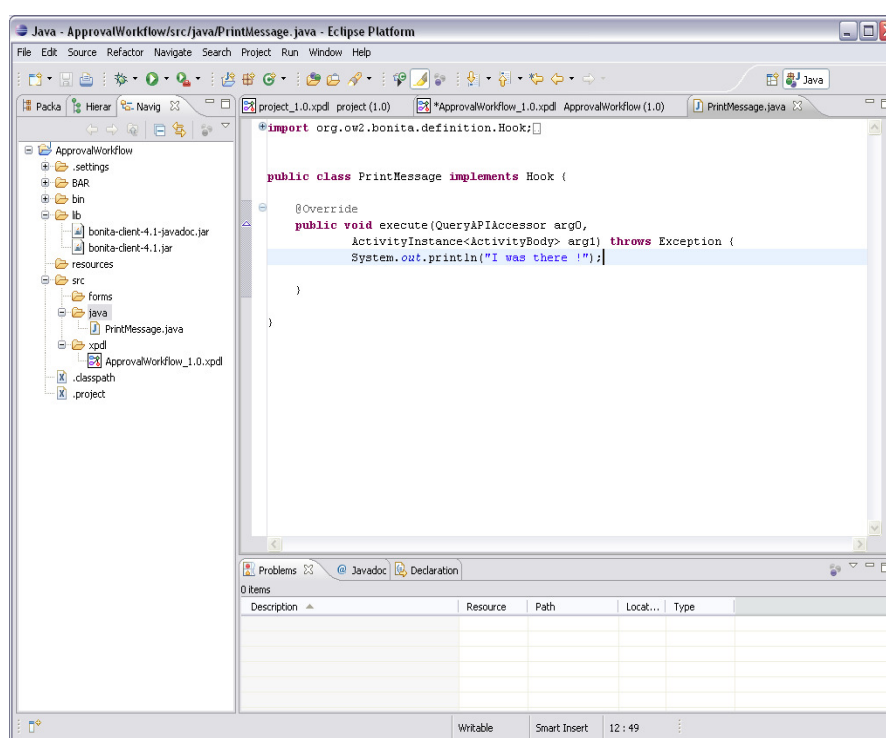


Figure 6-10. Editing Hook java file

Mappers and performer assignments can be created the same way, just leverage “New Bonita Role Mapper” and “New Bonita Performer Assign” features respectively.

Once created, Java BPM entities are ready to be used inside your BPM definition. Please check in the next chapter how to add hooks, mappers, performer assignments and Multi-instantiators to the BPM definition.



Note:

Hooks, as well as mappers, performer assignments and Multi-instantiators java entities are automatically compiled by the plugin as soon as you save them. If you are using external java libraries from within your hook just copy them into the lib directory.

Let’s now address how to generate the .bar file corresponding to this BPM project. Nothing easier than click on Eclipse save button. In fact, each save operation in your BPM project will automatically generate or update the .bar file of your process (by default this .bar file is located under the bar directory).

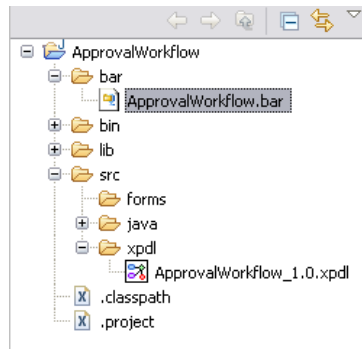


Figure 6-11. Bar file generation view

6.5 Using ProEd BPM editor

This chapter is dedicated to the desktop version of ProEd. However, most of the features you will find in the following lines will also be available in the Eclipse version.

6.5.1 Creating a New BPM Project

A new BPM project is created by selecting the File → New menu item or by clicking on the "New Project" button in the main toolbar. This displays the "New Project" dialog box as shown in the following figure.

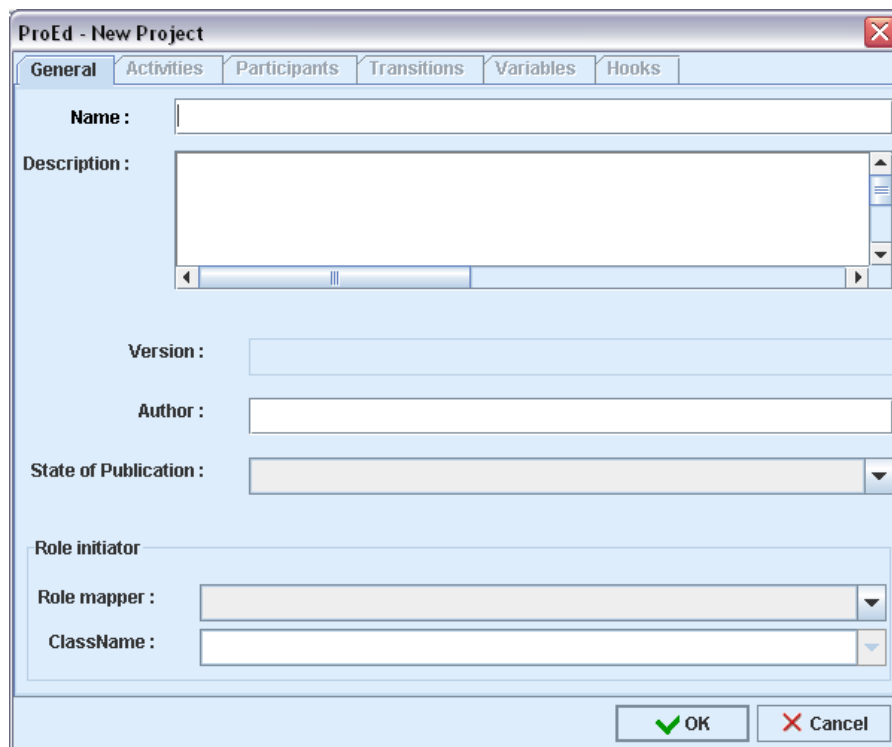


Figure 6-12. Creating a New BPM Project

A project **Name** must be entered in the "General" tab. All other fields are optional.

Field Descriptions

- **Name:** Assigned name of the project
- **Description:** enter more information about the Process.
- **Version:** This is a read-only field that displays the version information of the current BPM Process. New BPM Processes will be created with a version of 1.0.
- **Author:** enter the name of the BPM Process model Designer.
- **State of Publication:** select the appropriate state of publication, depending on the specific BPM design progress.
- **Role initiator:** this selection specifies which users are authorized to start the BPM Process in Nova Bonita.
- **Role mapper:** select the appropriate type of mapper:
 - **LDAP:** select "LDAP" to specify a group of users defined in the LDAP user directory.
 - **Custom:** select Custom to call a Java class listing specific users.
- **ClassName:**
 - For an LDAP mapper: select the group of users allowed to start the BPM Process.
 - For a Custom mapper:
 - If present, this field displays the list of implemented initiatorMapper Java classes (specifying a list of users allowed to start the Process). Select the appropriate Java class.
 - If the Java Mapper class is not yet implemented, type the Java classname to call.



Note:

The Java Mapper class must be created on the server with exactly the same name as before BPM Process deployment.

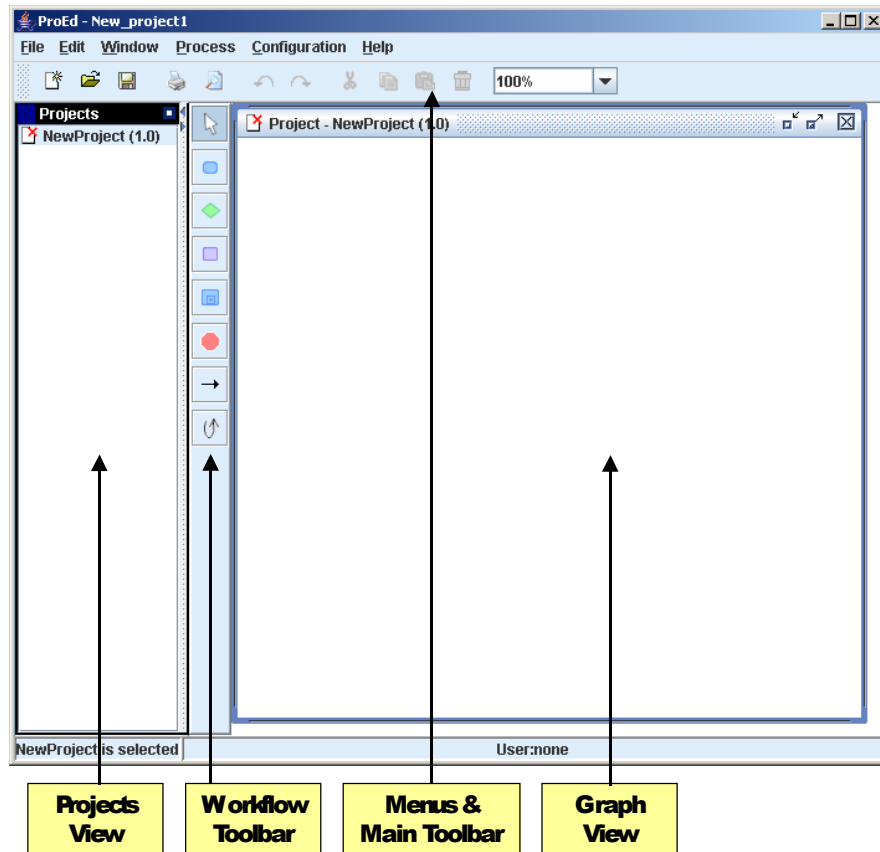


Figure 6-13. ProEd Display for New Project

Figure 4-2 displays the main ProEd frame for the newly created project.

- The menu and main toolbar at the top allow access to the main ProEd functions.
- The BPM toolbar in the middle of the window allows access to commonly used design functions.
- The Projects view on the left displays a list of all currently open projects.
- The Graph view on the right displays the BPMN representation of the current project.

See Section 6.5.2 for descriptions of these interface functions.

The screen area devoted to the projects view and the graph view can be resized by dragging the vertical divider either left or right between these two regions.

The status bar at the bottom displays the currently selected element and the user name, if in connected mode.

BPM elements can now be added to the project as described in the following sections.

6.5.2 Interface Overview

MENUS

File Menu

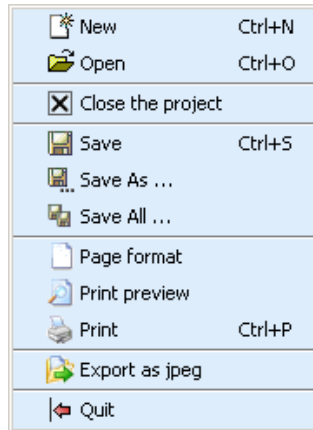


Figure 6-14. ProEd File Menu

- **Open:** opens a XPDL file containing process definition(s).
- **Close the project:** closes the current project.
- **Save:** saves the current process definition into a XPDL file.
- **Save as:** saves the current process into a XPDL file, after defining a new filename and location and/or incrementing the version.
- **Save all:** saves all currently opened projects.
- **Page Format:** defines page layout for printing.
- **Print preview:** previews the graph corresponding to the currently selected process with the defined **Page Format**.
- **Print:** prints the graph corresponding to the currently selected process.
- **Export as jpeg:** exports the current graph as a JPEG image file.
- **Quit:** exits ProEd.

Edit Menu







 U ndo	Ctrl-Z
 R edo	Ctrl-Y
 C ut	Ctrl-X
 C opy	Ctrl-C
 P aste	Ctrl-V
 D ele t e	D ele t e
S elect All	Ctrl-A
U nselect All	

Figure 6-15. ProEd Edit menu

- **Delete:** deletes the element selected on the graph view.

Window Menu



<input type="checkbox"/> Real Size	Ctrl+Alt+N
Zoom	▶
 Layout	
 G rid	Ctrl+G
Participant View	
✓ A ctivity View	

Figure 6-16. ProEd Window Menu

- **Real size:** reverts to the original size of the graph view (after zooming in or out).
- **Zoom:** select a value to zoom in or out on the graph view.
- **Grid:** display (or does not display) a grid on the graph.

- **Participant view:** organizes the graph of the BPM process by Participants as shown below.

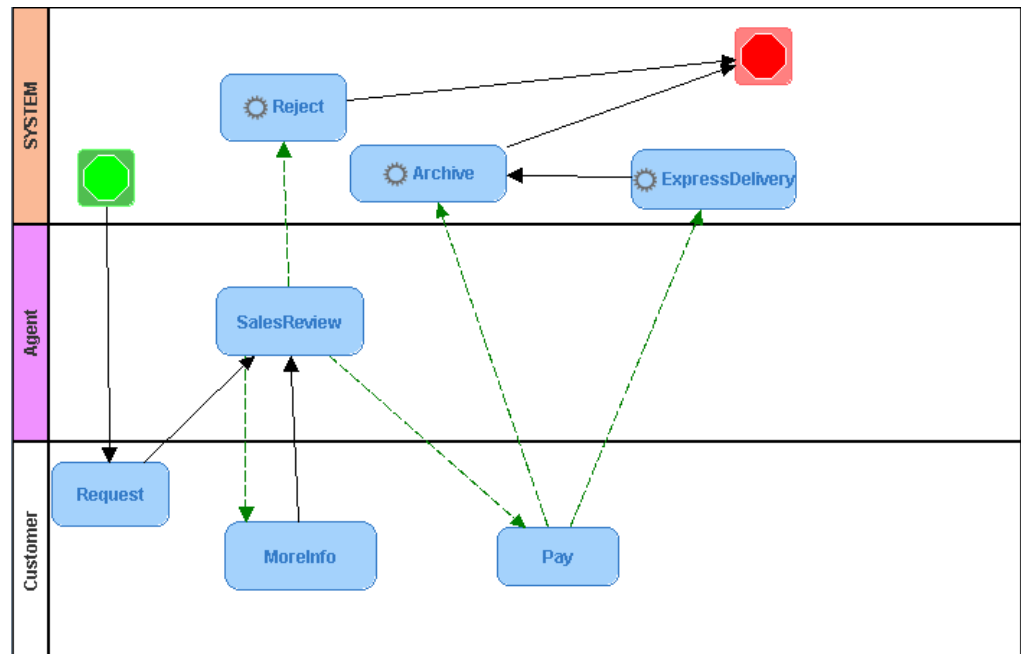


Figure 6-17. ProEd Participant View

- **Activity view:** organizes the graph of the BPM process by Activities as shown below.

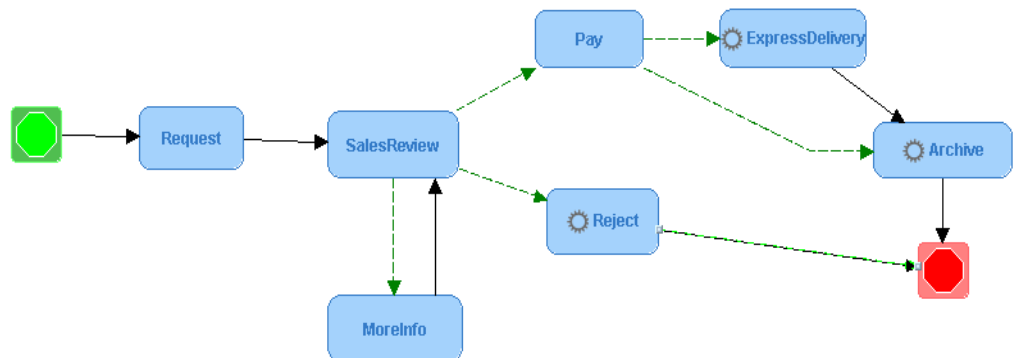


Figure 6-18. ProEd Activity View

Process Menu

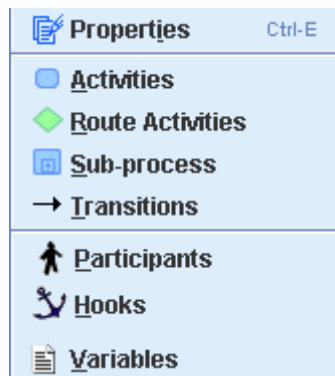


Figure 6-19. ProEd Process Menu

- **Properties:** displays the properties of the process.
- **Activities:** displays all basic activities of the process.
- **Sub-process:** displays all sub-processes of the process.
- **Route:** displays all route activities of the process
- **Transitions:** displays all transitions of the process.
- **Participants:** displays all participants of the process.
- **Hooks:** displays all hooks of the process.
- **Variables** displays all variables of the process.

Configuration Menu

- **Interface:**
 - **Change language:** change the language of the application (French, English, default see “INTERNATIONALIZATION” section).
 - **Change color:** change the color of the main window and of all dialog boxes and menus...
 - **Change look & feel:** change the look & feel of the application. This allows a user to select how the process window is represented.

Help Menu

- **Help:** displays the ProEd User's Manual.
- **About...:** displays ProEd version, release date, and copyrights.

TOOLBARS

Main Toolbar

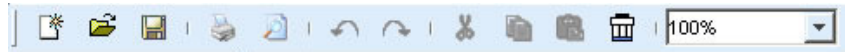


Figure 6-20. ProEd Main toolbar

BPM Toolbar

The BPM Toolbar groups all BPM design tools for easy access:






Button	Description
	Cursor: sets the pointer to its standard use.
	Add Basic Activity: creates a basic Activity (the smallest unit of work). See: "Creating and Defining Activities".
	Add Route Activity: creates a route Activity (synchronization Activity with complex transitional conditions). See: "Creating and Defining Activities".
	Add Sub-Process: creates a complete BPM Process model as an Activity. See: "Creating and Defining Activities".
	Add Transition: adds a Transition between two Activities. See: "CREATING AND DEFINING TRANSITIONS AND ITERATIONS".

Table 6-1. Description of BPM Toolbar Design Tools

VIEWS

Projects View

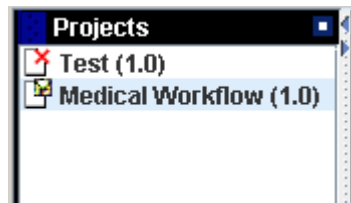


Figure 6-21. Projects View

The Projects View displays all processes present in the XPDL repository and / or processes that the user has created or opened. The version of the process is shown in parenthesis after the process name. Right-click on a process name to access either the process' graph view or to close the project. Double-click on the process name to display the process graph view. Click the black arrows to hide or display this view.

Graph View

The Graph View displays the graphic representation of the current process model. It can be organized in two different ways. The Activity View emphasizes the relationships between the activities. The Participant View emphasizes the participant involvement by grouping activities into participant swim lanes.

Activity View

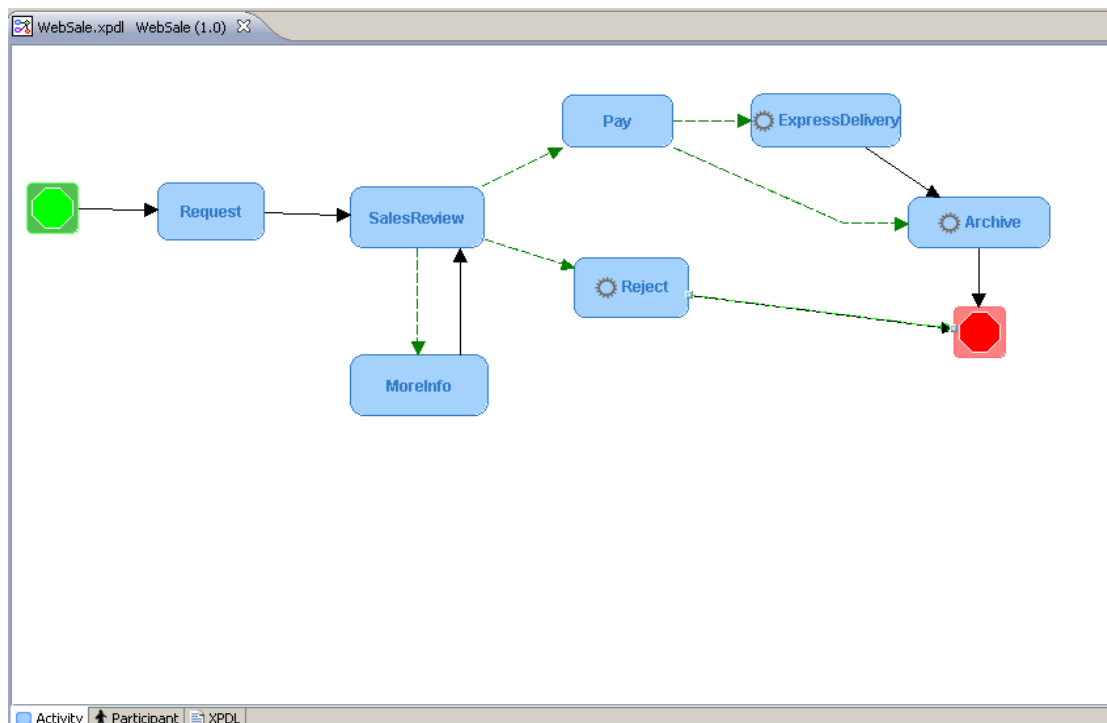


Figure 6-22. Activity View









Symbols	Description
	Basic activities are shown by blue rounded rectangles: automatic or manual start and various participants
	Route Activities are shown by green rounded rectangles: always automatic start and the SYSTEM participant
	Subflow activities are fat blue rectangles with a squared plus icon: always automatic start and the SYSTEM participant
	The start activity is shown by a green rounded square: always automatic start and the SYSTEM participant
	The end activity is shown by a red rounded square with a stop sign: always automatic start and the SYSTEM participant
	Ordinary transitions are shown by solid black arrows
	Transitions that have a condition are shown by dashed green arrows
	The gear symbol indicates an activity that has the automatic start mode

Table 6-2. Description of Activity View Graph Symbols

To add an activity (Basic activity, route, activity, Subflow or End activity):

- In the BPM toolbar, click on the activity button.
- The cursor changes to indicate the selected activity type.
- Click in the activity view at the location where the activity is to be added.
- The activity will be added at the specified location
- The activity's dialog opens to enter the activity name and other activity properties (except for the end activity, which has no properties or dialog)
- ProEd automatically changes back to the select mode, indicated by an arrow cursor.

To add a transition:

- In the BPM toolbar, click on the transition button
- The cursor changes to a cross and arrow.
- In the activity view, drag from the source activity to the target activity
- A transition is added between the two activities
- The transition's dialog opens to enter the transition name and other transition properties
- ProEd remains in the add transition mode, indicated by a cross and arrow cursor, and additional transitions may be added.

To delete an activity or and transition:

If ProEd is not in the select mode, indicated by an arrow cursor, press the top button in the BPM toolbar to enter the select mode.

- Select the desired (i.e. activity or transition) item by single clicking on it.
- Handles appear on the selected item to indicate its selection.
- Press the [Delete] key on the keyboard, or right click on the desired item, select Delete from the context menu, and answer Yes to the deletion confirmation dialog.

To reposition an activity:

- Drag the activity to the desired location.
- Transitions and iterations also move to remain attached to the activity in the new location.

To reposition a transition line:

The end points of a transition are fixed on the source and target activities; however the line that connects them may be re-positioned to avoid obstacles or un-clutter the diagram.

- Right click on the desired line or line segment.
- Select Add a Point.
- A new handle is added in the middle of the selected line or line segment.
- The new handle may be dragged to re-position the line.

To modify the properties of an activity or a transition:

- Double click on the item.
(does not apply to subflow activities)
- Right click on the item and select Properties from the context menu.
- The end activity has no properties that can be modified.

To access the items contained in subflow activity:

- Double click on the activity.
- A new graph window opens, showing the contents of the subflow.
- The contents of a subflow activity may not be edited in the new graph window

Participant View

The majority of the Activity View discussion in the previous section also applies to the Participant View.

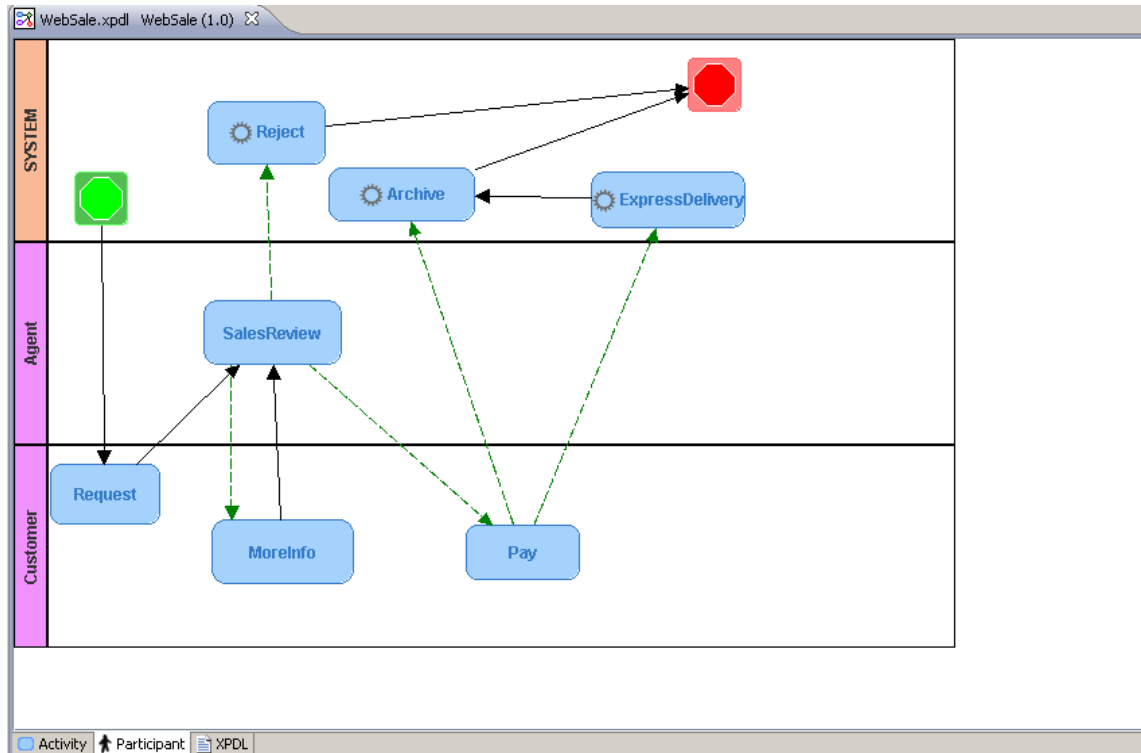


Figure 6-23. Participant View

The Participant View differs from the Activity View in the following ways:

- The activities are grouped by participant into "swim lanes".
- The SYSTEM participant lane is shown at the top of the diagram, with the other participants following in alphabetical order.
- Dragging a basic activity from one participant lane into another changes the activity's participant, and modifies the start mode as necessary to maintain compatibility.
- Route, subflow, and end activities must use the SYSTEM participant, so ProEd does not allow them to be dragged out of the SYSTEM lane.
- Activities are automatically re-positioned when it is necessary to place them in the proper participant lane, when two activities overlap, or when a participant view position has not been established for the activity.
- When participants are added or deleted, the resulting position of some of the existing participant lanes may change. The activities in these lanes are automatically re-positioned to move them into the new lane position, and the user may desire to re-arrange them in a more harmonious manner.

6.5.3 Load/Save/SaveAs/Delete Projects

The "file chooser" window is used to load, save, and delete the XPDL files corresponding to ProEd projects. It is accessed in the following ways:

- File → Open menu selection or "**Open**" toolbar button
- File → Save menu selection or "**Save**" toolbar button
- File → SaveAs menu selection
- "**Open File**" button in the Sub-Process dialog box.

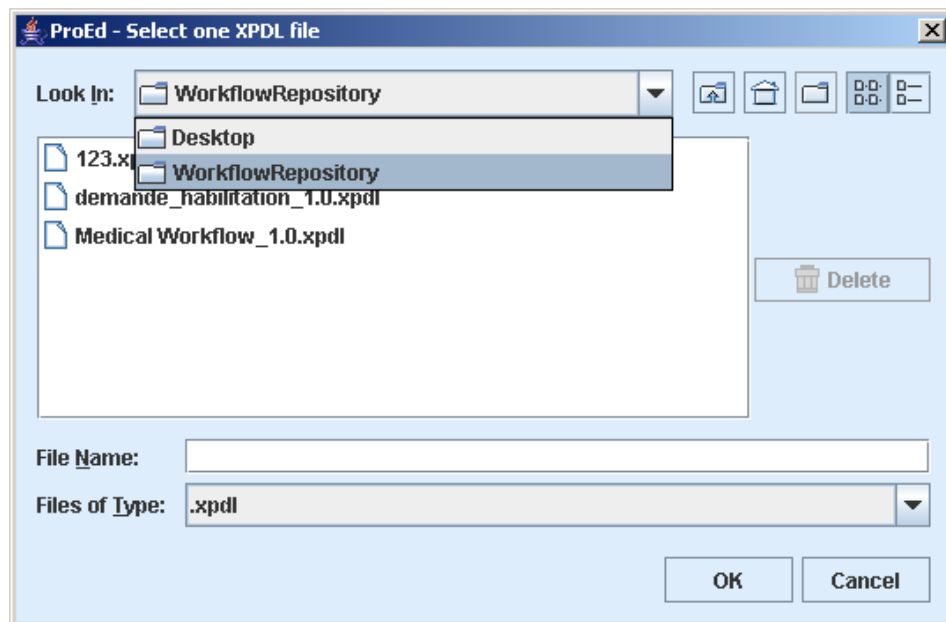


Figure 6-24. Open File Dialog

The text on the "**OK**" button will change to "**Open**" based on the operational context.

To open an existing file:

Use the combo box at the top of the dialog box to navigate to the desired directory. In "Connected" mode, the "BPMRepository" top-level directory entry is also available to allow selecting a file from the repository. After selecting the desired directory, select the desired file and click the "**OK**" button.

To save to the original file:

Doing "**Save**" on an existing project will save the project back into the original file without using a dialog. If this is a new project, doing "**Save**" will bring up the SaveAs Dialog to allow the initial file to be specified.

To delete an existing file:

This dialog box also allows the user to delete any XPDL file. Navigate to the file as above. When a XPDL file is selected, the "**Delete**" button at the right is enabled. Pressing the "**Delete**" button causes a dialog box to appear to confirm the intent to delete an existing file.



Note:

Note that the "**Delete**" button within ProEd can also be used to delete files contained in the repository.

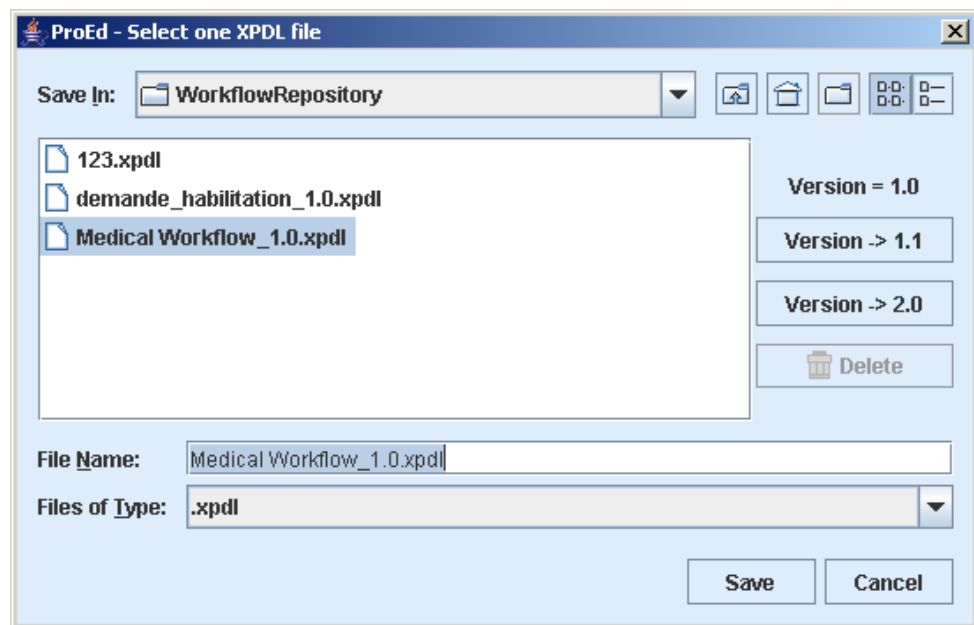


Figure 6-25. Save File Dialog

To save to an existing file:

When doing "**SaveAs**", it is possible to select an existing file to overwrite. Navigate to the file as shown in Figure 4-13, select an existing file and click the "**Save**" button. A dialog box appears to confirm the intent to overwrite the existing file.

- The Version field shows the existing version of the BPM Project.
- The upper Version button increments the minor version number in the saved file, as shown on the button.
- The lower Version button increments the major version number in the saved file, as shown on the button.

To save to a new version:

Using SaveAs to save an existing project into a new file with a new version is the only way to change the version. It is possible to increment the major version or the minor version number only by one.

The version is a property of a ProEd BPM Project, and is not dependent on the file name of the project's XPDL file.

Although it is not required, the recommended format for a ProEd XPDL file name is:

ProjectName_version.xpdL

For example:

Medical BPM_1.0.xpdL

When either of the Version buttons is used to increment the major or the minor version, a file name of this format will be automatically proposed.



Caution:

Note that attempting to "SaveAs" and selecting a file that is currently in use results in an error dialog box.

To save to a new file:

The "SaveAs" menu allows saving a project to a new file. Navigate to the desired directory as described above. Type a new file name in the "File Name" text box and click the "Save" button.

To save to an existing file:

When doing "SaveAs", it is possible to select an existing file to overwrite. Navigate to the file as described above and select an existing file and click the "Save" button. A dialog will appear to confirm the overwrite.



Caution:

Note that attempting to "SaveAs" and selecting a file that is currently in use results in an error dialog box.

6.5.4 Defining BPM Process Properties

A BPM Process Model is composed of the following:

Activities, Participants or Transitions between Activities
Process and Activity Variables,
Process and Activity Hooks

To define the Process Model Properties:

- Right-click the background in the **Graph view** and select **Properties** in the popup menu.
- Or select **Process** → **Properties** in the menus.

The BPM project dialog is divided into seven tabs:

- **Activity** tab: displays the list of all Activities included in this BPM Process model. To add an activity, refer to Section 6.5.6.
- **Participants** tab: displays the list of Participants defined for the entire BPM Process model and available for all Activities. Participants can be added, edited or deleted:
 - **Add** button: click to involve a Participant in the BPM Process model (see Section 6.5.5).
 - **Edit** button: click to modify the selected Participant.
 - **Delete** button: click to delete the selected Participant.
- **Transitions** tab: displays the list of Transitions involved in the BPM project. To add a Transition, see Section 6.5.8.
- **Variables** tab: defines the Process Variables used for the Process instantiation and propagated to all Activities in the project. Variables can be added, edited or deleted:
 - **Add** button: adds a new Variable at Process level: see Section 6.5.7.
 - **Edit** button: modifies the selected Variable.
 - **Delete** button: deletes the selected Variable.
- **Hooks** tab: This tab allows the definition of Process level Hooks (if needed). Process level hooks are instantiation hook and termination hook. Hooks can be added, edited or deleted:
 - **Add** button: adds a new Hook at the Process level: see Section 6.5.8.
 - **Edit** button: modify the selected Hook.
 - **Delete** button: deletes the selected Hook.

6.5.5 Adding Participants

Participants can be added at Process level or at Activity level. Whether a Participant is added to the whole project or only to a specific Activity, the participant becomes a Process model Participant and can be thus used for any other Activity created within the project (there is no need to add the Participant again to the project).

To add Participants:

- **At the Process level:** right click in the Project window, select **Participants** tab, and click the "Add" button.

- **At the Activity level:** in the Activity window, **General** tab, click the "**New Participant**" button.

CHOOSING AN EXISTING PARTICIPANT

Click the "**Existing participants**" checkbox in the "**Add Participants**" window to add an existing Participant: to choose the appropriate Participant, search the LDAP data or filter it.

The screenshot shows the "ProEd - Add participant" dialog box. It is divided into two main sections. The top section, "Existing Participants", is selected with a checked checkbox. It includes an "LDAP Search" button and a "Filter" dropdown menu. Below this is a table titled "Select participant" with columns for "Name", "Type", and "Description". The bottom section, "New Participant", is currently unselected. It contains a "Name" text field, a "Type" section with radio buttons for "Role", "Human" (which is selected), "Organizational Unit", and "System", and a "Description" text area. At the very bottom, there is a "Mapper" section with "Type" and "Class Name" dropdown menus, and "OK" and "Cancel" buttons.

Figure 6-26. ProEd Add Participant Window



Note:

The "**Existing participants**" checkbox is unavailable if no connection to the server is available.

LDAP Search

Click the "**LDAP Search**" button; the "**Participant Search**" window appears:

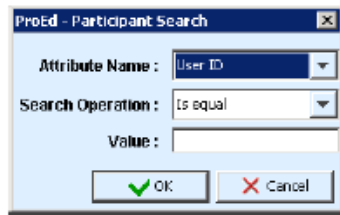


Figure 6-27. Add Participant Search Window

Enter data in the "**Value**" field to search participants. The User Directory can be searched on the following attributes:

- User ID
- Common Name
- Surname
- Given Name

Then, click the "**OK**" button.

The search results appear in the "**Select participant**" area.

The Mapper Type field in the Mapper area (at the bottom of the window) displays "**LDAP**".

Filter Search

Browse the "**Filter**" drop down menu to filter participants by type (Role, Human, Organizational Unit or System).

The filter results appear in the "**Select participant**" area.

In the "**Select participant**" area, select the desired Participant (Role, Person, Organization or System).

Click "**OK**" to add the Participant in the BPM Process model (Process level) or as the performer of an Activity (Activity level).

CHOOSING A NEW PARTICIPANT

ProEd offers the capability of adding a Participant that does not exist in the user directory but must be integrated in the BPM Process Model.

In the "Add Participants" window:

1. Check the "New participant" checkbox.
2. Fill in the fields in the "New Participant" area as follows:
 - **Name:** type the name of the new Participant. The name must be typed according to the user directory typographical rules.
 - **Type:** select the type of Participant to be created:
 - **Role:** a group of users
 - **Human:** a specific person
 - **Organizational Unit:** an organization
 - **System:** for automated Activities. The action is automatically performed, depending on the Activity Java class called (Hook).

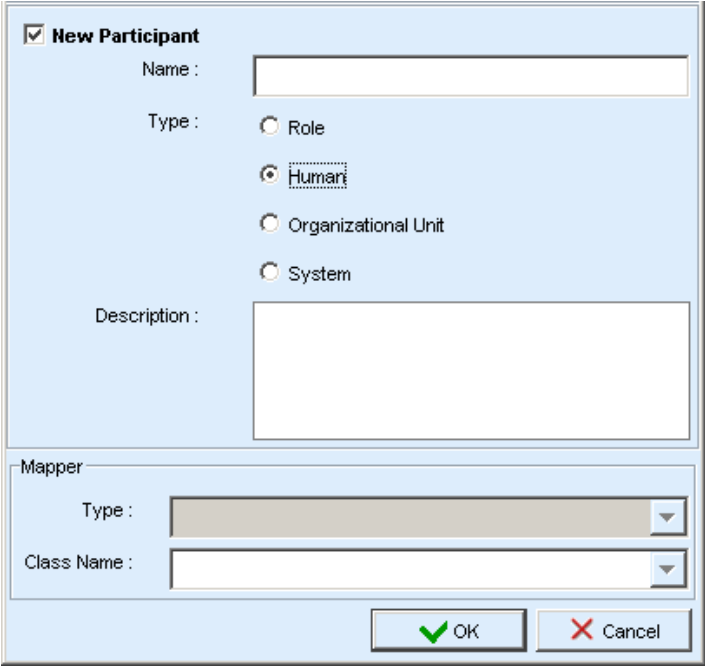


Figure 6-28. New Participant Window

3. If the **Role** or **Organizational Unit** Participant type is selected, supply a runtime Mapper in the **Mapper** area to allow the role-to-person association (not mandatory):
 - In the "**Type**" field:
 - Select **LDAP** to specify a group of users not yet created in the LDAP user directory. (In this case, the new Participant must be created in the user directory before deploying the BPM Process).
 - Select **Instance Initiator** to call the initiator of the project.
 - Select **Custom** to call a java class that specifies a list of users.
 - In the "**Class Name**" field:
 - For **LDAP** mapper (connected mode only): the field is unavailable. The group of users (role or organizational unit) will be picked in the user directory.
 - For **Instance Initiator** mapper: the field is unavailable. The initiator of the Process will be the one assigner to the role.
 - For **Custom** mapper:
If any, this field displays the list of available Java classes that call a specific list of users. Select the appropriate Java class.
If the Java class is not yet implemented, type the classname to be called.

Click **OK** to add the new Participant in the BPM Process model (Process level) or as the performer of an Activity (Activity level).

6.5.6 Creating and Defining Activities

Five types of activities are available within ProEd:

- **Basic Activity**: the smallest unit of work in a BPM process. The majority of Activities are basic.
- **Route Activity**: a flow control point or switch Activity used for synchronization and complex transitional conditions. This type of Activity does not contain Hooks.
- **Sub-process**: complete separate BPM Process model set as an Activity of the current BPM Process model. This type of Activity allows simplifying the graph of the BPM Process model, or to reuse an existing BPM Process model.

ProEd - Test

General | Deadline | Variables | Hooks

Name :

Type : Basic Activity

Start Mode : Automatic

Type of Join : AND

Execution Mode : Synchronous

Description :

Multiple Instance

Local variable to use:

Multi-Instantiator class:

Performer

Performer assignment

☒ **Type variable**

Select a variable:

☐ **Type custom**

Hook concerned:

Figure 6-29. New Activity Window

To create an activity:

1. In the BPM toolbar, click the Activity button: the pointer takes the shape of the Activity symbol.
2. Position the pointer on the graph view and click: the Activity creation window appears.
3. Fill in the "**General**" tab of the "**New Activity**" window as detailed in the following:
 - **Name:** type the Activity name. Name must be unique.
 - **Type:** this field displays the type of Activity selected (i.e. Basic).
 - **Start Mode:** select the start mode of the Activity:
 - **Manual:** a performer is required to start the Activity.
 - **Automatic:** the Activity is automatically performed by the system, depending on the Activity Java class called (Hook).

- **Type of Join:** this field specifies under what input condition the Activity becomes ready:
 - **AND** (default value): the Activity becomes ready only if all incoming transitional conditions are executed (synchronization between preceding Activities).
 - **XOR:** the Activity becomes ready if one incoming input transitional condition is executed.
- **Description:** enter information about the Activity.
- **Multiple Instance:** this area allows to choice of an instantiator class that will be resolved at runtime. A variable and the name of the java class corresponding to the instantiator class are required (this entity is not mandatory)
- **Performer** (manual Basic Activity only): this list allows assigning a performer (human, role or an organization) for the Activity being created. Click the "**Add Participant**" button and see Section 6.5.5 "Adding Participants".
- **Performer assignment** (manual Basic Activity only): this area allows the choice of the performer to be refined or deferred. The performer is determined at run time, depending on the value of an attribute, or by calling a java class:
 - **Variable:** this list displays all Variables of the Activity. The performer is determined at run time according to the value of the selected Variable.
 - **Custom:**

If specified, this list displays the deployed java classes for performer assignment. The performer is determined at run time by calling the selected java class.

If the Java class is not implemented yet, type the classname to be called (remember to create the Java Class with exactly the same name before deploying the BPM process).

DEFINING ACTIVITY Properties

In the graph window, right-click the Activity to define and select "**Properties**": the Activity window appears.

Define the Activity as follows:

- **General** tab: this tab is filled at creation time: see "Creating and Defining Activities".
- **Variables** tab: variables can be added, edited or deleted (see Section 4.4.7 "Creating Variables")
 - **Add** button: adds a new Variable to the Activity.
 - **Edit** button: modifies the selected Variable.
 - **Delete** button: deletes the selected Variable.
 - **Inherited Variables** area: displays the names of all the inherited variables:
 - Variables defined at the Process level
- **Activity** tab (Sub-process Activities only): displays the list of all Activities included in the Activity.
- **Participants** tab (Sub-process Activity only): displays the list of all Participants involved in the Sub-process Activity (see Section 6.5.5, "Adding Participants").
- **Transitions** tab (Sub-process Activities only): displays the list of all Transitions included in the Activity (see below, "Creating and Defining Transitions and Iterations").
- **Hooks** tab: defines the Activity level Hooks. Hooks can be added, edited or deleted (see Section 6.5.8, "Adding Hooks"):
 - **Add** button: click to add a new Hook to the Activity.

- **Edit** button: click to modify the selected Hook.
- **Delete** button: click to delete the selected Hook.
- **Deadline** area: click the "**Add**" button to add a deadline:
 - **Condition:**

Time (relative time in days, hours, minutes and seconds): the deadline is set at the end of the elapsed time (from the Activity starting time) entered.

Date (fixed date): the deadline is set at the selected date and time. Click the Calendar button to select a fixed date.
 - **Exception** (mandatory field):

If any, this field displays the list of deployed Java classes that defines the action to execute if the deadline is missed. Select the appropriate Java class.

If the Java class is not implemented yet, type the classname to be called (remember to create the Java Class with exactly the same name before deploying the BPM process).

DELETING AN ACTIVITY

- Select an Activity in the graph view using the pointer.
- Right-click the Activity and select "**Delete**".
- Alternatively, click the "**Delete**" button in the menu or press the "**delete**" key.
- In the confirmation dialog box click "**OK**" to confirm deletion.

6.5.7 Creating Variables

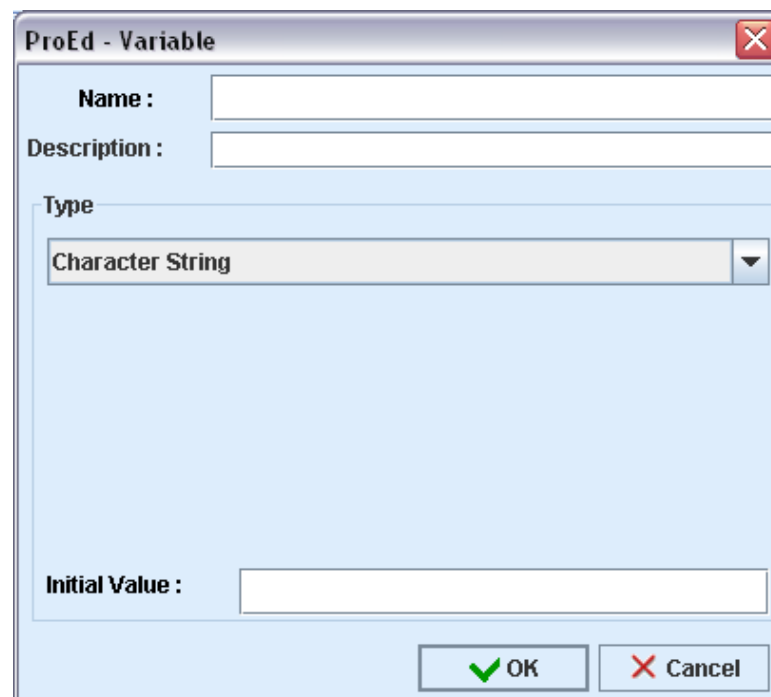
Variables can be created at the Process level or at the Activity level.

Variables can be:

- **String** type
- **Enumeration** type
- **Date** type
- **Boolean** type
- **Float** type
- **Integer** type

To add Process Variables to the BPM process model or to a specific Activity, do one of the following:

- At the Process level: in the Project view, **Variables** tab, click the **Add** button.
- At the Activity level: in the Activity window, **Variables** tab, click the **Add** button.



The image shows a dialog box titled "ProEd - Variable". It has a standard Windows-style title bar with a close button (X) in the top right corner. The dialog is divided into several sections. At the top, there is a "Name :" label followed by a text input field. Below that is a "Description :" label followed by another text input field. In the middle, there is a "Type" label above a dropdown menu that currently displays "Character String". At the bottom, there is an "Initial Value :" label followed by a text input field. The bottom right corner of the dialog contains two buttons: "OK" with a green checkmark icon and "Cancel" with a red X icon.

Figure 6-30. Variable Menu

- In the "**Variable**" window fill in the following fields:
 - **Name** (mandatory field): type a unique name for the Variable (\$\$, ll, and the space character are not allowed).
 - **Description**: enter a short description for the Variable.
 - **Type**: select one of the following types of Variable:
 - **Character String**: a character.
 - **Enumeration**: list of values. Click the Add button to add a value.
 - **Date**: a valid date (pop up)
 - **Boolean**: true or false values
 - **Float** : floating point
 - **Integer**: numeric value

6.5.8 Adding Hooks

Hooks are Java procedures that can be added to Process or Activity execution.

To add a Hook:

- At the Process level: in the Project window, **Hooks** tab, click the **Add** button.
- At the Activity level: in the Activity window, **Hooks** tab, click the **Add** button.

The "**Add Hook**" window is displayed.

The screenshot shows a dialog box titled "ProEd - Add hook". It has a standard Windows-style title bar with a close button (X). The dialog is divided into two main sections. The first section, labeled "Call point", contains two groups of radio buttons. The first group, "For tasks (aka manual activities)", has five options: "task:onReady" (which is selected), "task:onStart", "task:onFinish", "task:onSuspend", and "task:onResume". The second group, "For automatic activities", has one option: "automatic:onEnter". The second section, labeled "Hook Name", contains a text input field and a checkbox labeled "rollback". At the bottom of the dialog are two buttons: "OK" with a green checkmark icon and "Cancel" with a red X icon.

Figure 6-31. Add Hook Window



Notes:

Automatic call points are unavailable for Tasks and vice versa (as is the case in the task Hook window above).


To fill in the Add Hook window:

- **Call point section:**
 - **Task:onReady:** the Hook is called when the Task becomes available (i.e a manual activity is assigned to a user or a group of them)
 - **Task:onStart:** the Hook is called when an activity is executed by the user (manual activities)
 - **Task:onFinish:** the Hook is called when an activity is finished by the user (manual activities)
 - **Task:onSuspend:** the Hook is executed just after the user calls the suspend operation
 - **Task:onResume:** the Hook is executed just after the user calls the resume operation
 - **automatic:onEnter:** the Hook is executed once the automatic activity is executed
- **Error handling section:** a check box allows to define whether or not this hook will be rollbacked if an exception occurs. Rollback operation will also abort the activity state change.
- **Choose a hook section**
 - If available, this field displays the list of deployed Hooks available for use by the Activity or Process. Select a Hook from the list.
 - If the Java class is not yet implemented, type the Java classname to call (remember to create the Java Class with exactly the same name before deploying the BPM process).

CREATING AND DEFINING TRANSITIONS AND ITERATIONS

A **Transition** is a link between two activities; it allows the flow of control to pass from one Activity to another. Transitions can be created with or without an associated condition. A transition that has an attached condition will be displayed as a dotted line.

An iteration is set between two Activities and involves the repetitive execution of one or more BPM Activities until a condition is met. Iterations in Bonita 4.1.1 are defined through transitions

In the BPM toolbar, click the **Transition** 

1. In the graph window, click a source Activity and drag the pointer to a target Activity without releasing the mouse button.
 - **Transitions:** a straight arrow links the two activities, conditions may be added on the transition or modification of the properties of the transition (see following section).

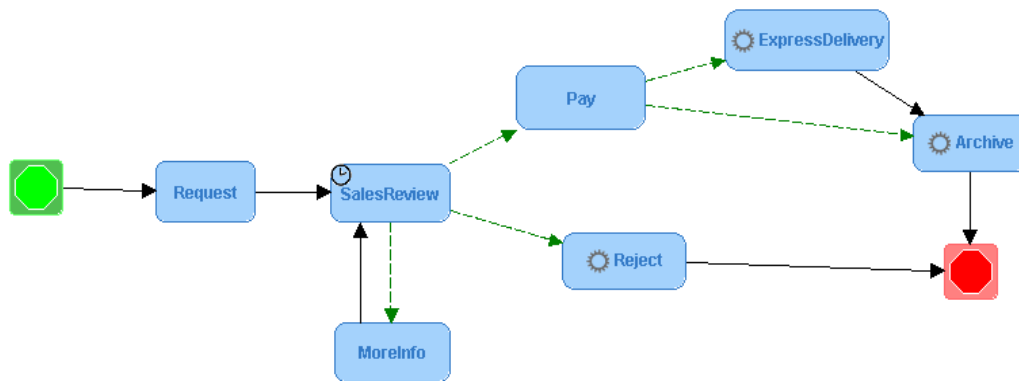


Figure 6-32. Iterations and Transitions Graph

2. Modify the shape of the arrow.
 - **Transitions:** right-click the Transition and select "**Add a point**"; a new point appears on the line, click and drag this point to change the shape of the arrow (repeat if needed). To delete a point, right-click a point and select "**Remove a point**". Note that the Activity View and Participant View have a separate set of added points so that the path of the transition line can be adjusted independently in each view.

ADDING / EDITING CONDITIONS ON TRANSITIONS

1. Right-click on a transition and select "**Add/Update condition**". The "**Add condition**" window appears:

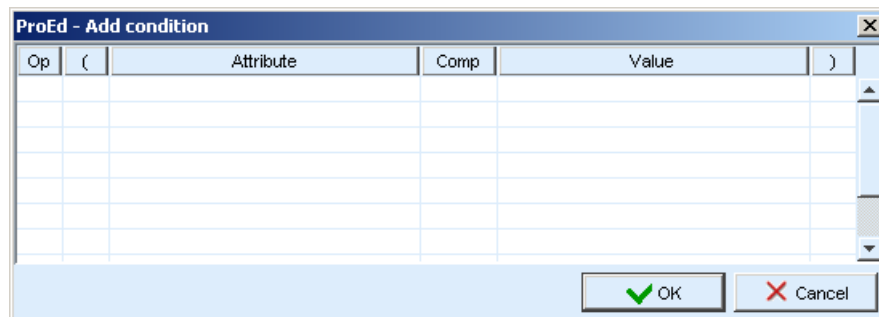


Figure 6-33. Add Condition Window

2. Fill in the table as described below:

Field	Value
Op (Operator)	Define an AND or OR operator between two conditions.
()	Select the appropriate number of brackets according to the number of conditions set for the Transition.
Variable	Select the Variable from the drop down list.
Comp (Comparator)	Select an operator: <ul style="list-style-type: none"> • = "is equal to" • != "is different from" (not equal) • > "is greater than" • < "lower than" • <= "lower or equals than" • >= "greater or equals than"
Value	Select the desired value that corresponds to the above Variable selected from the drop down list. A user-defined value can also be entered if the selected Variable is of the Character String type.

Table 6-3. Description of Condition Parameters

3. Click "**OK**" to create or update the condition: transitions with conditions as a dotted green arrow.

MODIFYING THE PROPERTIES OF A TRANSITION

1. Right-click a Transition arrow and select **Variable**.
The Transition window appears:

The screenshot shows a dialog box titled "ProEd - Request_Approval". It contains the following fields and sections:

- Name :** Request_Approval
- Source :** Request
- Target :** Approval
- Description :** A large text area for entering a description.
- Condition**: A table for defining conditions.

Op	(Variable	Comp	Value)

At the bottom right are "OK" and "Cancel" buttons.

Figure 6-34. Modifying Transition Properties

2. Fill-in the window:
 - **Name** (Transition only): type the name of the Transition.
 - **Source**: the name of the source Activity.
 - **Target**: the name of the target Activity.
 - **Description**: enter a description of the transition.
 - **Condition**: enter information in this table as explained in Section 4.3.11 "ADDING / EDITING CONDITIONS ON TRANSITIONS"
3. Click "**OK**" to update

INTERNATIONALIZATION

ProEd can be customized to display all static text in different languages. The base product supports English and French. All static text is contained in language-specific property files. A new language-specific property file can be created by starting with one of the existing property files and customizing it. At runtime, ProEd will scan for valid language files and allow selection of any language that is present.

Follow these steps to add a new language to ProEd:

1. Extract a base language property file from the ProEd.jar file. There is an ant task to assist in doing this. The build.xml file under the BPM installation contains a "proed-extract-lang" target. This can be used to extract any language file already in the ProEd package that resides in the delivery. From Bull, the package is delivered with 2 languages files, "ProEd_en.properties" for English, and "ProEd_fr.properties" for French. The script will ask for the name of the language file to extract. The file will be left in the current directory after the ant task is executed.
2. Rename this file by replacing the language code to be that for the desired new language. For example, if ProEd_en.properties (English) was extracted in step 1, and a new Spanish version is to be built, rename the file to ProEd_es.properties
3. Within the file, for each property ending in ".text", replace the property value with the language specific translation of the current value.
4. Save the modified property file specifying a name "ProEd_xx.properties", where "xx" is the java standard language code. (not necessary if the file was already renamed in step 2)
5. Add the new property file to the package. An ant task is provided to assist this step. With the new language property file in the current directory, run the target "proed-add-lang". This will ask for the name of the language file that will be added to the ProEd package. The ant task will add the file, as well as resign the jar files needed. No other steps are necessary by the user.

When ProEd is executed, the presence of the new language file is detected at runtime. When the "Change language" dialog is used, all languages found, including those added by the above steps, will be displayed in the list for selection as the target language for all ProEd panels and dialogs.



Note:

If a new version of BPM is installed, the above steps must be repeated.

Chapter 7. Development

This chapter describes how to start playing with Nova Bonita v4. More precisely it describes main steps to define and deploy BPM processes and how to start running them in Nova Bonita:

- How to deploy a process definition
- How to develop a simple application by leveraging Bonita APIs
- How to use advanced BPM capabilities

For end users, there is a dedicated document: [quickStartGuide.pdf](#) that illustrates the steps described above using a 100% graphical environment.

7.1 Nova Bonita APIs

If you are already familiar with previous Bonita versions and you have already developed your own applications on top of Bonita, we want to minimize your effort when migrating to Bonita v4. Compatibility from Bonita v3 to v4 is one of our main concerns.

At the same time, we took the chance to review and to improve Bonita v3 APIs in this new major version so basically the Bonita v3 API spirit is still there but we applied some improvements in Nova Bonita to simplify some operations and to add added value features.

Nova Bonita APIs has been refactored each intermediate release until become stable in this final version.

7.1.1 Getting started with Bonita APIs

Nova Bonita APIs are divided into 5 different areas:

- **DefinitionAPI:** to create/modify major process elements into the engine (packages, processes, activities, role mappers, variables by calling java methods instead of importing xpd files. It will allows also to modify the execution of runtime elements such as tasks and instances.
- **QueryDefinitionAPI:** to get useful BPM definition data from a particular process definition: packages, processes, activities, role mappers,
- **RuntimeAPI:** to manage process, instance and task life cycle operations as well as variables set/updates
- **QueryRuntimeAPI:** to get recorded/runtime informations for packages, processes, instances, activities, tasks (support for dynamic queries will be added in the future). It allows also to get tasks (aka manual activities) per state for a particular user and as well to get/list BPM instances.variables.
- **ManagementAPI:** to deploy BPM processes into the engine. XPD files and advanced entities such hooks, mappers and performer assignments can be deployed individually or in one shot.

There's also a generic API that allow to execute specific commands (queries) that could be needed (and not available in the APIs) in the context of a particular BPM based application:

- **CommandAPI:** to allow developpers to write and execute its own commands (mostly queries)

7.1.2 Nova Bonita APIs, playing with !

Nova Bonita v4 is an extensible and embeddable BPM solution that can be easily deployed in both standard (JSE) and Enterprise (JEE) environments.

- Nova Bonita can be easily integrated in your application as a BPM library.
- Nova Bonita can also be deployed in a JEE application server and so reached remotely by external applications

7.1.2.1 Nova Bonita local vs remote applications !

An utility class has been provided to unify access to Bonita APIs from both local and remote environments and so to avoid the use of lookups in JEE deployments: `org.ow2.bonita.util.AccessorUtil`.

Through this class, Nova Bonita APIs can be reached in a unified way in both local and remote applications. From a developer point of view Nova Bonita APIs are just Java POJOs.

The `AccessorUtil` object will also be leveraged inside BPM artifacts such hooks, mappers or performer assignments to deal with Nova Bonita APIs. By configuration you can specify the way in which the

Bonita APIs will be reached, the system property called `"org.ow2.bonita.api-type"` must be defined at client side to specify whether the APIs will be reached locally or remotely (possible values are `"Standard"`, `"AutoDetect"`, `"EJB2"` and `"EJB3"`).

Hereafter you will find an example on how to use the `AccessorUtil` class from your client application:

```
RuntimeAPI runtimeAPI = AccessorUtil.getRuntimeAPI();
QueryRuntimeAPI queryRuntimeAPI = AccessorUtil.getQueryRuntimeAPI();
```

You will find some samples applications leveraging this API under the `/examples` directory. For a detailed insight on Nova Bonita APIs, please take a look to the Nova Bonita javadoc apis (available under `/javadoc` directory).

7.2 Running the examples

The Nova Bonita package contains some complete BPM examples. Those examples can be easily compiled, deployed and executed (in both JSE and JEE modes), thanks to a set of "ant" tasks available on examples directory. Hereafter you will find some information about how to play with one of those examples (ApprovalWorkflow sample). For others (carpool and websales samples), you can proceed the same way by replacing the references to approval workflow by the one corresponding to your sample name:

- The Bonita basic Approval Workflow: a simple Approval Workflow application illustrating the workflow definition, workflow deployment and execution phases.
- The Bonita Multi-instance Approval Workflow: a version of the Approval Workflow application in which the "approval" activity is dynamically assigned to a set of users at runtime. This example illustrates activities multi-instantiation paradigm.

The `build.xml` in the root directory (`BONITA_HOME/examples/approvalWorkflow` directory) contains required targets to compile and launch the example in both standard (JSE) and enterprise (JEE) environments:

The Approval Workflow sample is configured to run together with the default hibernate persistence service. Main Process Virtual Machine entities, XPDLL extension as well as execution related data will be persisted in a relational database. By default Nova Bonita embeds HSQL database and uses it as a default database. You can easily change this default configuration and use your favorite database by modifying the `hibernate-core.properties` file located under `"conf"` directory.

>ant aw-standalone

This sample application leverages the Security and Identity services so you must provide a right user login/passwd to run the sample. The default identity module (based on a properties file) is provided with three users ("john", "jack" and "james" logins with "bpm", "bpm" and "bpm" as password). All three can login into to system but only john and jack are able to play in the Approval Workflow sample. This behaviour is due to the users - role mapping defined in the previous workflow sample.

The java example simply deploy the xpdL file as well as advanced java entities such hooks and mappers in the engine (deployBar method in ManagementAPI), then creates a BPM instance and calls getTasksList, startTask and finishTask methods from the RuntimeAPI

This sample application can be launched at both standard and enterprise modes. In the enterprise mode Nova Bonita APIs are available as Stateless Session Beans. Enterprise sample version can easily be executed with the following ant tasks:

>ant aw-jonas4 (for a deployment in JOnAS application server version 4)

>ant aw-jboss4 (for a deployment in JBoss AS version 4)

>ant aw-eb (for a deployment in EasyBeans EJB3 container)



Note:

The enterprise version requires Nova Bonita (bonita.ear file) to be deployed in an application server (see Enterprise Instalation chapter of this Guide to know more about how to deploy Nova Bonita in a JEE application server).

As for basic Approval Workflow sample, you can easily launch the multi-instantiated version as follows:

>ant aw-multiInst-standalone

Multi-instantiation version of Approval Workflow can also be launched remotely. Please type "ant -p" to get an overview of remote execution tasks.

An advanced version of this sample is also provided. This version package the approval workflow sample as well as the Nova Bonita engine (standalone mode) in a war file together with a simple web application.

This war file can be deployed in a web container (i.e Tomcat) as an standalone BPM application. This web application allows to deploy, undeploy and create instances of a deployed BPM sample to illustrates how to reach Nova Bonita APIs from a web application (API calls will directly leverage the Bonita POJO based API).

In order to generate this war file, just type:

>ant war (for aw.war file generation)

once this file is generated, just deploy it in your favorite web container (i.e Tomcat). The web application will be available by default at <http://localhost:8080/aw> if you are using Tomcat.

7.3 Java Properties

There are couple of considerations that must be taken into account when using Nova Bonita in both JSE and JEE versions. Security and Bonita environment configurations should be set when running

Bonita as a library in your web application (so to be added when running your Tomcat server). Those JAVA properties are concerned:

- `org.ow2.bonita.environment` = URL to the xml file containing the BPM environment configuration or filepath to this file or resource path. Nova Bonita engine environment lookup mechanism will check for the environment location following this order: first URL, second, filepath and third resource
- `java.security.auth.login.config` = PATH where the login configuration is available (default configurations are available under `/conf` directory).

`JAVA_OPTS` environment variable can be used for this purpose. `org.ow2.bonita.environment` property must also be set when using Bonita as a server (deployed in an application server). In this scenario, the client side application would require the security related property and the Bonita server side only the Bonita environment.

If those properties are not defined, for instance when starting JOnAS or Jboss servers, Nova Bonita will take as default the `bonita-environment.xml` file include in the `bonita.ear`.

7.4 Administration operations

The Nova Bonita distribution provides a set of administration utilities/commands that can be executed through ant tasks. Hereafter you will find main commands available:

- **deployBarDb** Deploy the bar file in standalone mode
- **deployBarEb** Deploy the bar file in enterprise mode
- **ear.ejb2** Generate an ear which can be use in an ejb2 environment (Jonas 4 or Jboss 4 application server)
- **ear.ejb3** Generate an ear which can be use in an ejb3 environment (easybeans, jonas 5, jboss 5)
- **info** display helpful information on this distribution
- **init-db** Generate database schema from the environment configuration
- **undeployBarDb Undeploy** the process in standalone mode
- **undeployBarEb Undeploy** the process in enterprise mode
- **usageDeploy** The usage to successfully deploy a bar file
- **usageUndeploy** The usage to successfully undeploy a bar file

Through those commands, BPM processes can be deployed , undeployed, J2E enterprise version can be generated, database schema can be initialized or still th test suite can be launched.

7.5 Database configuration

Nova Bonita distribution is configured by default to run with an embedded database (HSQL database). This allows you to start playing with Bonita without configuring your own database. The default database will be created in your filesystem and soon as you execute one of the samples provided in the distribution (default location would be the `"java.io.tmpdir"` system variable: i.e `/tmp` in Linux Systems or `Local Settings/tmp` in windows platforms).

Database configuration for HSQL is available under the `/conf` directory of the distribution (`hibernatecore.properties` file).

```
hibernate.dialect org.hibernate.dialect.HSQLDialect
hibernate.connection.driver_class org.hsqldb.jdbcDriver
hibernate.connection.url jdbc:hsqldb:file:${java.io.tmpdir}/bonita-db/
bonita_core.db;shutdown=true
hibernate.connection.username sa
hibernate.connection.password
```

```
hibernate.hbm2ddl.auto update
hibernate.cache.use_second_level_cache false
hibernate.cache.use_query_cache false
hibernate.cache.provider_class org.hibernate.cache.EhCacheProvider
hibernate.show_sql false
hibernate.format_sql false
hibernate.use_sql_comments false
```

This configuration (core) concerns the Nova Bonita definition and runtime information. As you can see in the /conf directory, there is also another file called hibernate-history.properties which contains the database configuration for the Nova Bonita history database. In fact process history data can be stored either in XML files or in a database (please check Configuration and Services chapter) and so this file concerns the choice of a database to store history data.

7.6 Changing the default database configuration

Hereafter the instructions to update the default database configuration in Nova Bonita to use your favorite relational database:

1. Copy your database driver into the Nova Bonita distribution /lib directory (i.e mysql.jar, oracle.jar)
2. Configure hibernate.properties file for your favorite database (i.e examples for MySQL, Oracle and Postgres are provided in both core and history properties files). Note that you can use different databases or instances for "core" and "history" configurations. By default, if you only update hibernatecore.properties file and you keep the default history service to use the XML service (see the bonita-environment.xml file) no history data will be stored in the database.
3. Open a command line and go to the Nova Bonita distribution main directory and type "ant init-db":

```
init-db:
[input]
[input] Which hibernate configuration to use to generate database -?
[input] Default value for bonita engine database is:
hibernate-configuration:core
[input] Default value for bonita DbHistory database is:
hibernate-configuration:history
[input] []
```

This command allows Nova Bonita administrators to initialize a particular database instance with the Nova Bonita database schema (including FK and indexes). Just select the database you want to initialize (core vs history) by copying either "hibernate-configuration:core" or "hibernate-configuration:history" chains.



Note:

Previous values names are correlated to the corresponding environment file (bonita-environment.xml) so if you change the values of this file you should take care to use the same during the DB initialization

7.7 Connecting with your Information System (Hooks)

Hooks in Nova Bonita BPM context are external java classes performing user-defined operations. At different moments in BPM process execution, hooks might be called by the BPM engine after instance activity state update.

Hooks may be called at different moments in the activity lifetime. Depending on the activity type (task or automatic) only a set of them can be used. Hooks are prefixed by the type of activity in which they are associated.

Table 7-4. Hooks Names

BPM Hook Name	Short Name	Rollback	Without Rollback
task:onReady	TOR	X	X
Task:onStart	TOS	X	X
Task:onFinish	TOF	X	X
task:onSuspend	TOS	X	X
task:onResume	TORE	X	X
automatic:onEnter	AOE	X	X



In Table 3-2, two of the hooks have a « Rollback » tag. This means that if an error occurs during execution (see Section 3.5.1 to determine how to signify that an error has occurred), all transactions performed on the BPM level are rolled back (e.g.: activity variables changed during the hook execution are reset to their previous values) to the beginning state of the transaction. Be aware that no « external » actions performed by the hook are rolled back. The BPM engine has no way to determine the potential effect of the actions performed in the hook, thus it is the hook developer's responsibility to anticipate and respond to possible failures during hook execution.

7.7.1 Hooks Execution Time Scale

In a « client-driven » BPM context, meaning a java program using the BPM engine API, Hooks are leveraged to link processes with already existing applications. Hook calls/executions are performed by the BPM engine itself, meaning that Hooks are always executed at BPM server side so they should not be used to communicate with client applications (i.e opening a browser).

However, previous behavior could be achieved in a standalone Nova Bonita deployment, meaning a deployment in which the BPM engine is embedded in the client application (i.e Tomcat based application).

As said, hooks are automatically triggered by the BPM engine based on activities types (known as activity “bodies” in Nova Bonita) life cycle.

Automatic activities (activities with body = automatic) can only trigger one hook called onEnter. This hook is automatically executed by the BPM engine once the activity runs.

7.7.2 Out-of-Timescale Hooks

The onDeadline (OD) hooks are not related to a particular activity state while are also automatically triggered by the engine.

The onDeadline (OD) hook is triggered when the set activity deadline expires. The deadline is set either by default (through ProEd editor) or by another hook/external program using the BPM API. Be sure to keep in mind that multiple deadlines may be set within the same activity.

In a normal case, this hook is not triggered or called by a « common user ».

7.7.3 Hooks Capabilities

As stated previously, hooks in Nova Bonita context are external java classes performing user-defined operations. Mainly, hooks are able to perform a wide array of actions involving two different « channels »: « BPM related», and what is called a « java-related environment».

7.7.4 BPM-Related Hook Actions

The hook likely interacts with the BPM engine through a dedicated API. The most commonly performed operations are the following:

- get an activity / process variable
 - methods: [getVariable](#)
- set an activity/process variable
 - methods: setVariable
- get the activity / process name
 - methods: getProcessActivity and getProcess
- start / finish an activity
 - methods : startTask, finishTask

For more detailed information about dedicated functions, see the Nova Bonita Javadoc API document (available under /doc/javadoc directory).

7.7.5 Java-Environment-Related Hook Actions

Hooks are java classes, thus any «standard » java operation may be performed. Therefore, external program calls, and some system calls, are feasible within a hook context. Depending on the execution environment (JSE vs JEE) hooks can directly call Java API without adding additional libraries.

For instance, in a Nova Bonita deployment in a JEE application server, hooks can directly reach external web services by leveraging the web services framework provided by the application server (i.e Axis)

7.7.6 Hooks Logic

Hooks are user-defined logic entities (java classes), which may be triggered at defined points of the corresponding activity body life cycle (automatic or task). Those defined points are:

- **task:onReady hook** is called when the task becomes available (activities of type task when it is assigned to a user or a group of them). The task state has moved from init to ready state
- **task:onStart hook** is called as soon as the task is executed. This hook is launched by the engine once the task moves to executing state. This is correlated to the startTask operation performed by a user.
- **task:onFinish hook** is called as soon as the task is finished. This hook is launched by the engine once the task moves to finished state. This is correlated to the finishTask operation performed by a user.
- **task:onSuspend hook** is called when the task is suspended by a user. This is correlated to the suspendTask operation.
- **task:onResume hook** is called when the task is resumed from a suspended state. This is correlated to the suspendTask operation.
- **automatic:onEnter hook** is called when an automatic activity is executed.
- **onDeadline hook** is called when an activity deadline expires. Deadlines are started when an activity becomes ready and throws an exception (called onDeadline hook) when the period of time is reached.

7.7.7 Fault Management

If an exception occurs during hook execution, it is propagated to the application triggering the execution of the hook.

Consider the following scenario:

An application calls the **finishTask** operation on “Activity1”; this triggers the execution of an **task:onFinish hook** which raises an exception; the exception is caught by the application.

Things may be a little bit problematic if automatic activities are used:

- Imagine that the finishTask statement on “Activity1” completes normally, and that “Activity1” has an outgoing transition towards an automatic activity “Activity 2”.
- “Activity 2” is started and finished automatically in the context of the first call related to “Activity1”.
- Therefore if “Activity 2” has a task:onFinish hook that raises an exception, it will interrupt the call related to “Activity1”.
- This means “Activity1” will not finish (the activity stays under the executing state) and the system throws an exception due to the “Activity2” execution error.

The above examples show two error scenarios related to transactional hook execution. **Be aware that hooks can be executed in a transactional or in a non-transactional context, depending on the implemented interface (Hook vs TxHook).**

Transactional hooks are executed in the same transactional context as the activity invoking the hook. Transactional hooks are those ones implementing the TxHook interface:

- Any changes performed on a transactional resource are included in the existing transactional context.

- Any exception raised by a Hook aborts the existing transaction, so the activity will be re-executed later on. Furthermore, all operations executed by the hook before the exception was raised are rolled back (meaning BPM related operations)

The BPM engine also allows creation of hooks for execution without a transactional context. Those hooks (implementing the Hook interface) are executed outside the activity transactional context.

- It is strongly recommended that these types of hooks NOT be used to access either BPM APIs or other transactional APIs. Nova Bonita prevents internal issues by only exposing to developers of Non Transactional hooks the use of query APIs.
- If one of these hooks fails during its execution, the system throws an exception but the activity life cycle is updated without rolling back any operation or transaction.

Consider the last sample scenario as described above and change the use of the task:onFinish hook implementation from TxHook interface to Hook one, the execution is as follows:

- Imagine that the finishTask statement on “Activity1” completes normally, and that “Activity1” has an outgoing transition towards an automatic activity “Activity 2”.
- “Activity 2” is started and terminated automatically in the context of the first call related to “Activity1”.
- Therefore, if “Activity 2” has an task:onFinish non transactional hook that raises an exception, the hook does not interrupt the call related to “Activity1”.
- This means, “Activity1” finishes without problem, but the system throws an exception due to the “Activity2” execution error.

7.7.8 Activity/Hooks and Transactions

Any change of state (i.e. startTask, finishTask, suspendTask statements) performed against an activity is part of a transaction.

This transaction typically involves more than one activity in synchronous executions: for example, a finishTask statement performed on an activity triggers a change of state in all connected activities. The BPM engine therefore keeps transactional consistency across activities.

The BPM engine aborts a transaction in two cases:

- A failure at system level (e.g. impossibility to access the BPM database)
- An exception not caught by a transactional hook.

When hooks are executed in a transactional context:

- Any changes performed on a transactional resource are included in this existing transactional context.
- Any exception raised by the hook aborts the existing transaction.

7.7.9 Writing a Hook

Process and activity hooks are a java class and have to implement either Hook or TxHook interfaces. Those interfaces are located in the package org.ow2.bonita.definition.Hook.

A hook only requires one method to be implemented: execute. Parameters of this method will vary depending on the implemented interface.

For example the signature of a execute method for a hook implementing the hook interface looks as follows:

```
execute(QueryAPIAccessor accessor, ActivityInstance<ActivityBody> activityInstance)
```

while the same method signature for a hook implementing the Tx interface would be:

```
execute(APIAccessor accessor, ActivityInstance<ActivityBody> activityInstance)
```

While the first one gets a QueryAPIAccessor parameter only allowing access to the read only Nova Bonita APIs, the second one (TxHook interface) give access to the whole Nova Bonita API through the APIAccessor object.

Main purpose of this approach is to guarantees that exceptions in non transactional hooks will not cause any damage to BPM related data, meaning updating the database state inside hooks should be avoided in non transactional hooks.

The other parameter of the execute operation is an object containing the execution data of the current activity in which the hook is executed. This parameter represents the activity of a particular BPM instance.

7.7.10 Hooks-Specific Operations

Prior to coding a hook, the developer should organize step by step the needed operations and call the appropriate functions to achieve the goal. This code is enclosed in the hook main function.

The main way to return an error signal to the BPM, (the hook caller), is to send a Java Exception. The exception constructor takes a parameter string and BPM, upon receiving this exception, is able to deal with it by a rollback if the hook is transactional (Rollback type hook). The end user may also be informed that a problem occurred during the activity treatment.

7.7.11 Caveat Regarding Activity Deadline

Contrary to what might be intuitive, the deadline of an activity must be set prior to activity start. In fact, deadlines could be set in the previous manual activity in a task:onFinish hook (e.g. to set the deadline in the activity number 3, it is necessary to invoke a hook in the task:onFinish hook of activity 2). See example Set Deadline Hook below for set deadline code.

7.7.12 Use Case

7.7.12.1 A Simple Hook

For this example, “Hello world” will be printed in the BPM console

The code produced is the following:

```
import org.ow2.bonita.definition.Hook;
import org.ow2.bonita.facade.QueryAPIAccessor;
import org.ow2.bonita.facade.runtime.ActivityBody;
import org.ow2.bonita.facade.runtime.ActivityInstance;

public class Reject implements Hook {

    public void execute(QueryAPIAccessor accessor, ActivityInstance<ActivityBody> activityInstance) throws
    Exception {
        System.out.println("Hello World");
    }

}
```

Though very interesting, this hook is not very useful.

7.7.12.2 A More Complex Hook

In this case, the intent is to send an email to the process creator after registering a person (defined in the BPM process under the activity variable “Email_address“.

Check the related functions documentation for more details on their use.

```
import org.ow2.bonita.definition.TxHook;
import org.ow2.bonita.facade.APIAccessor;
import org.ow2.bonita.facade.QueryRuntimeAPI;
import org.ow2.bonita.facade.runtime.ActivityBody;
import org.ow2.bonita.facade.runtime.ActivityInstance;

import javax.naming.InitialContext;
import javax.mail.Session;
import javax.mail.Address;
import javax.mail.Transport;
import javax.mail.internet.InternetAddress;
import javax.mail.internet.MimeMessage;

public class Accept implements TxHook {
```

```

    public void execute(APIAccessor accessor, ActivityInstance<ActivityBody> activityInstance) throws Exception {

        QueryRuntimeAPI runtime = accessor.getQueryRuntimeAPI();
        String mailString = (String)runtime.getActivityVariable(activityInstance.getUUID(), "Email_address");

        // obtain JNDI initial context
        InitialContext ctx = new javax.naming.InitialContext();

        // use JNDI lookup to obtain Session (in the context of an application server)
        Session session = (javax.mail.Session) ctx.lookup("java:comp/env/mail/MailSession");
        MimeMessage m = new MimeMessage(session);
        m.setFrom();

        Address[] to = new InternetAddress[] { new InternetAddress(mailString)};
        m.setRecipients(javax.mail.Message.RecipientType.TO, to);
        m.setSentDate(new java.util.Date());
        String content = "";
        m.setContent(content, "text/plain");

        // Sending email
        Transport.send(m);
        System.out.println("Email was successfully sent");
    }
}

```

7.7.13 Practical Steps for Hooks Usage

7.7.13.1 Hook Loading and Compiling

Hooks are stored on the file system as standard java classes. It is necessary to load the code that has been written into the production environment in which Nova Bonita has been installed. The way to do this is as follows:

- Create the source .java file, i.e. *MyHook.java*
- Compile this java file by adding the bonita.jar library in your compilation classpath



Note:

If the java class uses user-defined libraries, include them in your classpath before compiling and deploying the hook.

7.7.13.2 Hooks deployment

Hooks deployment has been improved in Nova Bonita regarding previous versions. A centralized way to deploy hooks is available through the ManagementAPI façade. This API allows easily to deploy hooks but also any other advanced entities such mappers and performer assignments.

Depending on the Nova Bonita deployment environment (i.e Tomcat, Application Server, Spring application...) this façade can be leveraged as a POJO or as a Session Bean.

The idea is to provide different operations allowing hooks deployment:

- `deployClass`: this operation deploys a single hook class
- `deployClasses`: operation allowing to deploy two or more hooks in a single operation
- `deployClassesInJar`: operation getting as a parameter a jar file which include a set of hooks to be deployed

Those operations deploy hooks that could be leveraged afterwards by any BPM process deployed in Nova Bonita.

Hooks can also be deployed in the context of a BPM process, meaning only visible for a particular BPM process:

- `deployBar`: operation deploying a BPM process as well as hooks, mappers and performer assignments entities.

Remove and replace operations are also available in this API allowing hooks removing and hook classes updates:

- `removeClass`: remove a hook fro the production environment
- `replaceClass`: replace an already deployed hook from the production environment

7.8 Mapping BPM roles with users in your IS (Mappers)

Writing a mapper is very similar to writing a hook because both are Java classes than implements a well defined interface. In the case of mappers this java class is used to designate a person. A mapper Java class is used to designate the person(s) corresponding to a specific user-defined role.

7.8.1 Writing a Mapper

A **mapper** specifies person(s) corresponding to a specific role defined in the BPM process model by the process model designer. It is used to automatically fill-in users with a group of Participants defined in the Process model.

Three methods for filling are available (three types of mappers) depending on the method used to retrieve the users in the information system:

- getting groups/roles in an LDAP server (*LDAP mapper*)
- calling a java class to request a users repository (*custom mapper*)
- getting the initiator of the project instance (*Instance Initiator mapper*)

In fact LDAP and Instance Initiators mappers types are particular uses cases (implementations) of customs mappers.

LDAP mapper will be useful in production environments in which roles/groups defined at BPM definition time match with LDAP groups while Instance Initiators mappers are usually leveraged for testing purposes as they assign a particular role to the user that has created a particular BPM instance.

While mappers are assigned to roles/groups in processes, they are automatically resolved by Nova Bonita when a manual activity (called task) becomes available (ready state). This resolution users-roles resolution mechanism gives a lot of flexibility when assigning tasks to one users or a set of them in a particular BPM instance execution.

7.8.2

Mapper Types: LDAP, Custom, and Properties

LDAP MAPPER (not yet supported in Nova Bonita RC2 ! but can be implemented through a Custom mapper)

This mapper uses the LDAP directory to retrieve users that correspond to a specific role defined in Bonita BPM project.

LDAP mapper specifics:

- The location of the LDAP groups depends on the attributes: *roleDN* and *roleNameAttribute*.
- There is no mapping between roles/groups in the LDAP and roles in BPM database (same name for both bases).
- The attribute name: *uid* is used to achieve the mapping between the actor identifier in the LDAP base and the *userName* in the Workflow base.
- If the group does not exist, an exception is thrown.
- Users found in the groups must have been deployed before usage of the mapper function. Otherwise an exception is thrown.
- The name of the mapper is user-defined.

Limitations of this version:

- Groups cannot be recursive. Groups' inclusions are ignored.

CUSTOM MAPPER

This allows the process developer to request use of the user's storage base. When this type of mapper is added, a call to a java class is performed. The name of this mapper is the name of the called java class (ex.: *org.myProject..CustomMapper*). After retrieving these users they must be added to the project instance as well as added to the targeted role.

INSTANCE INITIATOR MAPPER

At present, this type of mapper auto fills the role with the user name of the creator of the instance (based on the authenticated user initiating the instance). This mapper is useful for testing purposes in order to assign the role specified in the property to the user instantiating the process. By leveraging this mapper in all the roles defined in a BPM process, developers can easily check that the BPM execution is working as expected.

7.8.3

Practical Steps for Using Custom Mappers

Mappers – loading and compiling

The BPM engine loads and executes these classes at runtime. To add a custom mapper, perform the following steps:

Mappers are stored on the file system as standard java classes. It is necessary to load the code that has been written into the production environment in which Nova Bonita has been installed. The way to do this is as follows:

- Create the source .java file, i.e. *MyMapper.java*

- Compile this java file by adding the bonita.jar library in your compilation classpath



Note:

If the java class uses user-defined libraries, include them in your classpath before compiling and deploying the mapper.

Mappers – deploying a mapper

Mappers deployment has been improved in Nova Bonita regarding previous versions. A centralized way to deploy mappers is available through the ManagementAPI façade. This API allows easily to deploy mappers but also any other advanced entities such hooks and performer assignments.

Depending on the Nova Bonita deployment environment (i.e Tomcat, Application Server, Spring application...) this façade can be leveraged as a POJO or as a Session Bean.

The idea is to provide different operations allowing mappers deployment:

- `deployClass`: this operation deploys a single mapper class
- `deployClasses`: operation allowing to deploy two or more mappers in a single operation
- `deployClassesInJar`: operation getting as a parameter a jar file which include a set of mappers to be deployed

Those operations deploy mappers that could be leveraged afterwards by any BPM process deployed in Nova Bonita.

Hooks can also be deployed in the context of a BPM process, meaning only visible for a particular BPM process:

- `deployBar`: operation deploying a BPM process as well as hooks, mappers and performer assignments entities.

Remove and replace operations are also available in this API allowing mappers removing and mappers classes updates:

- `removeClass`: remove a mapper fro the production environment
- `replaceClass`: replace an already deployed mapper from the production environment

7.8.4 Example of a Mapper

The following mapper returns “John” and “Jack” names as the "mapped persons". Of course, external requests to data storages are likely to be used in the mapper classes.

```
import java.util.HashSet;
import java.util.Set;

import org.ow2.bonita.definition.RoleMapper;
import org.ow2.bonita.facade.QueryAPIAccessor;
import org.ow2.bonita.facade.uuid.ProcessInstanceUUID;

public class MyMapper implements RoleMapper {
    public Set<String> searchMembers(QueryAPIAccessor accessor, ProcessInstanceUUID instanceUUID, String roleId) {
        Set<String> usersId = new HashSet<String>();
        usersId.add("John");
        usersId.add("Jack");
        return usersId;
    }
}
```

}

7.9 Selecting actors in Activities (Performer Assignments)

This feature provides a means within the BPM engine to modify the standard assignment rules for activities.

As mappers, performer assignments only concerns to manual activities (Tasks). While mappers returns a list of users of a role, performers returns only one user: the user responsible of a particular manual activity.

Performer assignments allows assigning additional assignment rules other than those in the standard BPM model (mappers). In the standard model, all users assigned to a group associated to the activity are able to see and execute this activity (*ToDo List*).

By adding this new functionality, it is possible to

- **Assign the activity to a user of a group** by calling a java class in charge to do the user selection into the user group (*custom performer assignment*)
- **Dynamically assign the activity to a user** by using an *activity property* (*variables performer assignment*)

When this functionality is added, the user could be notified (via mail notification for instance) that the activity is ready to be started. Others users of the groups (called role in Nova Bonita) associated to the activity will be considered as candidates and so will not see this activity (Task) in their todo list anymore.

Candidates (list of users resolved by a mapper) could be leverage by a performer assignment of type custom to make a selection of the user responsible for this manual activity execution.

7.9.1 Performer Assignment Types: Custom and Properties

7.9.1.1 Custom Performer Assignment

This allows the process developer to write a request with a user-defined algorithm. When this type of custom performer assignment is added, a call to a java class is performed.

The name of this callback performer assignment is the name of the called java class (ex.: *org.myApplication.performerAssign.CallbackSelectActors*):

7.9.1.2 Variables Performer Assignment

This allows the process developer to provide at **variables performer assignment** creation time, the activity variable used by the BPM engine to assign the activity. This activity variable is defined in the targeted activity to be assigned.

7.9.2 Practical Steps for Using Callback Performer Assignments

7.9.2.1 Performer Assignment – Loading and Compiling

As mappers, custom performer assignment are loaded and executed by the BPM engine. If adding a specific custom classes, follow these steps:

- Create the source .java file, i.e. *MyPerformer.java*
- Compile this java file by adding the bonita.jar library in your compilation classpath



Note:

If the java class uses user-defined libraries, include them in your classpath before compiling and deploying the performer assignment.

Performer assignment – deploying a performer assignment

Performer assignments deployment has been improved in Nova Bonita regarding previous versions. A centralized way to deploy those entities is available through the ManagementAPI façade. This API allows easily to deploy performers assignment (PA) but also any other advanced entities such hooks and mappers.

Depending on the Nova Bonita deployment environment (i.e Tomcat, Application Server, Spring application...) this façade can be leveraged as a POJO or as a Session Bean.

The idea is to provide different operations allowing PA deployment:

- `deployClass`: this operation deploys a single PA class
- `deployClasses`: operation allowing to deploy two or more PA in a single operation
- `deployClassesInJar`: operation getting as a parameter a jar file which include a set of PA to be deployed

Those operations deploy PA that could be leveraged afterwards by any BPM process deployed in Nova Bonita.

PA can also be deployed in the context of a BPM process, meaning only visible for a particular BPM process:

- `deployBar`: operation deploying a BPM process as well as hooks, mappers and performer assignments entities.

Remove and replace operations are also available in this API allowing PA removing and PA classes updates:

- `removeClass`: remove a PA fro the production environment
- `replaceClass`: replace an already deployed PA from the production environment

7.9.2.2 Example of a Performer Assignment

In the following performer assignment, the user “John” is set as the performer of the activity calling this custom performer assignment.

```
import java.util.Set;

import org.ow2.bonita.definition.PerformerAssign;
import org.ow2.bonita.facade.QueryAPIAccessor;
import org.ow2.bonita.facade.uuid.ProcessInstanceUUID;

public class MyPerformerAssign implements PerformerAssign {

    public String selectUser(QueryAPIAccessor accessor, ProcessInstanceUUID instanceUUID,
        String activityId, String iterationId, Set<String> candidates) {
        return "John";
    }

}
```

7.10 Assigning activities to multiple actors (Multi-Instantiators)

This feature allows to create and assign a number of activities to a set of actors (human and systems) at runtime.

Multi-instantiator concerns any activity type and are really useful in situations in which the number of occurrences of a particular activity is not known at definition time.

The principle is based on the execution of an "Multi-Instantiator" class (added to the activity definition) that returns an object containing:

- A list of values which size is determining the number of instances to be created and so executed. This list of values is used to set for each created activity instance a dedicated activity variable (this activity variable is also added to the definition of the activity)
- The number of finished instances expected to take the transition (called joinNumber). This number must be **greater than 0** and **lesser than or equal to** the number of created instances.

7.10.1 Practical Steps for Using Multi-Instantiators

7.10.1.1 Multi-Instantiators – Loading and Compiling

Multi-instantiators are loaded and executed by the BPM engine. If adding a specific custom class, follow these steps:

- Create the source .java file, i.e. *MyInstantiator.java*
- Compile this java file by adding the bonita.jar library in your compilation classpath



Note:

If the java class uses user-defined libraries, include them in your classpath before compiling and deploying the performer assignment.

Multi-Instantiator – deploying a Multi-Instantiator

Multi-Instantiator is a new feature added in Bonita v4. A centralized way to deploy those entities is available through the ManagementAPI façade. This API allows easily to deploy Multi-instantiators (MI) but also any other advanced entities such hooks, mappers....

Depending on the Nova Bonita deployment environment (i.e Tomcat, Application Server, Spring application...) this façade can be leveraged as a POJO or as a Session Bean.

The idea is to provide different operations allowing MI deployment:

- `deployClass`: this operation deploys a single MI class
- `deployClasses`: operation allowing to deploy two or more MI in a single operation
- `deployClassesInJar`: operation getting as a parameter a jar file which include a set of MI to be deployed

Those operations deploy MI that could be leveraged afterwards by any BPM process deployed in Nova Bonita.

Hooks can also be deployed in the context of a BPM process, meaning only visible for a particular BPM process:

- `deployBar`: operation deploying a BPM process as well as hooks, mappers and performer assignments entities.

Remove and replace operations are also available in this API allowing MI removing and MI classes updates:

- `removeClass`: remove a MI from the production environment
- `replaceClass`: replace an already deployed MI from the production environment

7.10.1.2 Example of a Multi-Instantiator

In the following example, users “John”, “jack” and “james” will get an instance of an activity at BPM runtime.

As specified in the return values, those three users as expected to take over the activity (three different instances in fact) before the Nova Bonita runtime decides to move forward:

```
import java.util.ArrayList;
import java.util.List;

import org.ow2.bonita.definition.MultiInstantiator;
import org.ow2.bonita.definition.MultiInstantiatorDescriptor;
import org.ow2.bonita.facade.QueryAPIAccessor;
import org.ow2.bonita.facade.uuid.ProcessInstanceUUID;

public class ApprovalInstantiator implements MultiInstantiator {

    public MultiInstantiatorDescriptor execute(QueryAPIAccessor accessor,
        ProcessInstanceUUID instanceUUID, String activityId, String iterationId)
        throws Exception {
        List<Object> variableValues = new ArrayList<Object>();
        variableValues.add("john");
        variableValues.add("jack");
        variableValues.add("james");
        return new MultiInstantiatorDescriptor(3, variableValues);
    }
}
```

7.11 Connectors

Connectors are a new feature of Bonita (preview feature at this time). They simplify the use of hooks in activities of a process. Connectors are parameterized hooks allowing an easy connection with the information system.

Connectors realize a well defined and repeated task, no more no less, such sending an email, accessing a user LDAP repository or calling a web services from a Bonita process.

At present, 9 connectors are available:

- a connector to send emails
- a connector to do LDAP searches
- 7 connectors to modify strings

7.12 How to use it

In order to realize its task, a connector may need input parameters and returns (if necessary) output values. A connector was designed to automate usual tasks.

To use a connector is really easy: *initialize it, execute it and that's it!*

A connector is a smart black box: when it is set, it checks itself parameter values. And if something wrong appends, it warns the user of his mistake(s). While the connector finds mistakes, the user cannot execute it.

Connectors can be used in several ways, for example in transactional hooks. Soon, they will be available to use through the Bonita designer without coding anything.

So how to use connectors in a hook:

- create a txHook
- in the “execute” method, declare a new connector
- initialize it
- execute it

In a same hook, a connector or a set of connectors can be added. Getter values can be used in setter methods of others connectors. Before executing a connector, it checks whether its setting is correct. If not, the execution cannot be performed.

This example shows how to link different connectors. First the LDAP connector searches an email address from a specific user. Then the second connector removes useless data from the search result in order to use it with the next connector. Finally, the email connector sends an email to a user using the converted string.


```

public class SendEmailFromLdapHook implements TxHook {

    public void execute(APIAccessor accessor,
        ActivityInstance<ActivityBody> activityInstance)
        throws Exception {
        LdapConnector ldap = new LdapConnector();
        ldap.setHost("localhost");
        ldap.setPort(10389);
        ldap.setProtocol(LdapProtocol.LDAP);
        ldap.setUserName("cn=Directory Manager");
        ldap.setPassword("bonita-secret");
        ldap.setBaseObject("ou=people,dc=bonita,dc=org");
        ldap.setScope(LdapScope.SUBTREE);
        ldap.setFilter("(uid=doej)");
        ldap.setAttributes(new String[] {"mail"});
        ldap.execute();

        ReplaceAllStringConnector replace =
            new ReplaceAllStringConnector();
        replace.setString(ldap.getResult());
        replace.setRegex("mail"+Connector.KEY_VALUE_SEPARATOR);
        replace.setReplacement("");
        replace.execute();

        EmailConnector email = new EmailConnector();
        email.setSmtphost("smtp.bonita.org");
        email.setSmtpport(10025);
        email.setTo(converter.getConvertedString());
        email.setSubject("It works!");
        email.setFrom("test@bonita.org");
        email.execute();
    }
}

```

7.13 How to create it

Before to create a new connector you should define the expected behavior (what the connector must do, what it can do, what it must not do ...) Then the connector should be coded (see steps defined below) and tested. (a static method checks the validity of the connector) Then the connector is set and executed after validation (done by the connector itself: automatic and dynamic validation at runtime)

prerequisites: the Bonita-template jar and at least a Java 2 Standard Edition 1.5.

Follow these steps in order to create a connector:

- create a connector class
- add attributes
- generate getter and setter methods
- add wrapped methods (if necessary)
- put annotations on getter and setter methods
- add value checking
- write the connector execution
- test it

create a connector class:

Find an explicit name for your class and extends Connector class.

For example:

If the aim of the connector is to calculate simple operations, call its SimpleCalculatorConnector (Connector is not mandatory but it indicates clearly that what is the class)

add attributes:

It is possible to define default values. (these values will be used by the Designer)

For example:

This simple calculator needs the first operand, the second operand, an operator and a result.

```
private double firstOperand;  
private double secondOperand;  
private boolean plus;  
private boolean minus;  
private boolean dividedBy;  
private boolean times;  
private double result;
```

generate getter and setter methods:

The name of setter methods must be `setAttributeName` because the Connector needs this format to do its checking. Do not add values checking, it will be done in another method.

For example:

- *if the attribute name is firstOperand the setter method name must be setFirstOperand(...) and its getter method name is getFirstOperand()*
- *if the attribute type is boolean, the getter method name begins with 'is' and not 'get': isMinus()*

add wrapped methods:

A connector can be used standalone. But to use it through Bonita, in some cases, it is mandatory to add other getter and setter methods in order to wrap existing methods.

Bonita supports only 5 Java types: Long, Double, Boolean, Date and String.

So methods to convert Bonita types to attribute types of the connector (setters) and methods to convert attribute types of the connector Bonita types (getters) are required.

For example:

firstOperand has a double type (a primitive one): had setFirstOperand(Double value) and convert the Double value in a double value. Here it is useful to check the null value of the parameter!

```
public void setFirstOperand(Double value) {  
    if (value == null) {  
        //whatever you want  
        firstOperand = 0.0;  
    } else {  
        firstOperand = value.doubleValue();  
    }  
}
```

If the connector contains a collection or an array:

Add appropriate separators between each value. The connector class contains three separators: the first one to separate key and value, the second one to separate attributes and the third one to separate objects.

For example:

```

private List<String> fruits = new ArrayList<String>();

public void setFruits(String fruit) {
    if (fruit == null) {
        fruits = new ArrayList<String>();
    } else {
        if (fruit.length() > 0) {
            String[] tokens = fruit.split(OBJECT_SEPARATOR);
            for (String token : tokens) {
                fruits.add(token);
            }
        }
    }
}

```

put annotations:

Annotations are used by the connector in order to check whether the connector is well-formed (static validation) and whether parameters given to the connector are correct. Annotations must be put only on methods.

There are four annotations:

@Input: this annotation indicates that this method sets an input field. This annotation can only be put on a setter method!

This annotation contains an attribute which defines if the input field is a technical one or a business one. Technical fields are fields to configure for example a server. Business ones are fields to execute a request (ie the others). By default, the value of this attribute is business. So it is not necessary to define business as input attribute.

For example:

- in *EmailConnector*, *smtpPort* is a technical field because it does not concern the email but how to send it. That is why it was declared technical

```

@Input("Technical")
public void setSmtpPort(int port) {...}

```

- with the example of the *SimpleCalculatorConnector*, add only **@Input** on *firstOperand*, *secondOperand* and each operator because they are the input parameters.

```

@Input
public void setFirstOperand(...)

```

@Output: this annotation indicates that this method gets an output field. This annotation can only be put on a getter method! It has to be on the method used by Bonita! (ie not the primitive one if the Bonita wrapped method exists)

For example:

The SimpleCalculatorConnector returns the result of the operation.

```

@Output
public Double getResult()

```

@Required: this annotation indicates that the input and/or output field is required. So add this annotation with input or output annotation.

This annotation contains an attribute which indicates when this field is required.

If this attribute is not set or empty, the field is required. Otherwise this attribute contains an expression containing connector attribute names and boolean operators (and = &, or = |, xor = ^, not = !). The result of this expression indicates whether the field is required depending on other attributes values.

For example:

- with the simple calculator, operands are required

```
@Input @Required
public void setFirstOperand(...)

@Input @Required
public void setSecondOperand(...)
```

operator is also required but in this example only one is required

```
@Input @Required("!(minus && !dividedBy && !times")
public void setPlus(...)

@Input @Required("!(plus | times | dividedBy)")
public void setMinus(...)
```

@Forbidden: this annotation indicates whether the field must be set. Add as well this annotation with input or output annotations.

This annotation contains the same attribute as @Required but the meaning is different if the result of the boolean expression is true, the field must not be set.

For example:

```
@Input @Required("!(plus | times | dividedBy)")
@Forbidden("plus | times | dividedBy")
public void setMinus(...)
```

Note:

- annotations are used by the designer and the connector verification.
- if the connector contains wrapped methods, do not copy annotations from the method to the wrapped one. Once is enough!

add value checking:

Connector has a dynamic checking. It verifies whether required fields are set ie different from `null` (required annotations must be on setter methods). But the null verification is necessary if no required annotation are put on the method and the field must not be `null`.

At the end of this checking, it returns a list of connector errors if the setting is incorrect. But this checking cannot check values of connector attributes. That's why this checking is performed in the `validateValues` method. Only the connector developer can code this method. For example he can check the validity of email addresses, if numbers are in the good range or greater than 0, ...)

For example:
in the simple calculator example, it is not allowed to divide by zero

```
protected List<ConnectorError> validateValues() {  
  
    List<ConnectorError> errors = new ArrayList<ConnectorError>();  
  
    if (isDividedBy() && getSecondOperand() == 0) {  
  
        errors.add(new ConnectorError("secondOperand",  
  
            new IllegalArgumentException("division by 0 are not allowed!")));  
  
    }  
  
    return errors;  
}
```

To add an error, use a List of ConnectorError. A ConnectorError contains the field name where the errors occurs and an Exception describing the error.

write the connector execution:

In the method executeConnector, write what the connector must do.

test it:

Finally test the connector with its static validation method and add all other tests. (more connectors and tests examples, can be found on the SVN repository)

For example:

```
Class<SimpleCalcualtorConnector> clazz = SimpleCalcualtorConnector.class;  
  
Assert.assertTrue(Connector.validateConnector(clazz).isEmpty());
```

This is what you get at the end:

```
package ...;  
  
import java.util.ArrayList;  
  
import java.util.List;  
  
import org.ow2.bonita.connector.Connector;  
  
import org.ow2.bonita.connector.ConnectorError;  
  
import org.ow2.bonita.connector.annotation.Forbidden;  
  
import org.ow2.bonita.connector.annotation.Input;  
  
import org.ow2.bonita.connector.annotation.Output;
```

```

import org.ow2.bonita.connector.annotation.Required;

public class SimpleCalcultorConnector extends Connector {

    private double firstOperand;

    private double secondOperand;

    private boolean plus;

    private boolean minus;

    private boolean times;

    private boolean dividedBy;

    private double result;

    public double getFirstOperand() {

        return firstOperand;

    }

    public double getSecondOperand() {

        return secondOperand;

    }

    public boolean isPlus() {

        return plus;

    }

    public boolean isMinus() {

        return minus;

    }

    public boolean isTimes() {

        return times;

    }

    public boolean isDividedBy() {

        return dividedBy;

    }

    public double getdoubleResult() {

        return result;

    }

    @Output

```

```

public Double getResult() {

    return result;

}

@Input @Required

public void setFirstOperand(double firstOperand) {

    this.firstOperand = firstOperand;

}

public void setFirstOperand(Double firstOperand) {

    if (firstOperand == null) {

        this.firstOperand = 0;

    } else {

        this.firstOperand = firstOperand;

    }

}

@Input @Required

public void setSecondOperand(double secondOperand) {

    this.secondOperand = secondOperand;

}

public void setSecondOperand(Double secondOperand) {

    if (secondOperand == null) {

        this.secondOperand = 1;

    } else {

        this.secondOperand = secondOperand;

    }

}

@Input @Required("!(minus | times | dividedBy)")

@Forbidden("minus | times | dividedBy")

public void setPlus(boolean plus) {

    this.plus = plus;

}

public void setPlus(Boolean plus) {

    if (plus == null) {

        this.plus = false;

    }

}

```

```

    } else {

        this.plus = plus;

    }

}

@Input @Required("!(plus | times | dividedBy)")

@Forbidden("plus | times | dividedBy")

public void setMinus(boolean minus) {

    this.minus = minus;

}

public void setMinus(Boolean minus) {

    if (minus == null) {

        this.minus = false;

    } else {

        this.minus = minus;

    }

}

@Input @Required("!(minus | plus | dividedBy)")

@Forbidden("minus | plus | dividedBy")

public void setTimes(boolean times) {

    this.times = times;

}

public void setTimes(Boolean times) {

    if (times == null) {

        this.times = false;

    } else {

        this.times = times;

    }

}

@Input @Required("!(minus | times | plus)")

@Forbidden("minus | times | plus")

public void setDividedBy(boolean dividedBy) {

    this.dividedBy = dividedBy;

}

```



```

public void setDividedBy(Boolean dividedBy) {

    if (dividedBy == null) {

        this.dividedBy = false;

    } else {

        this.dividedBy = dividedBy;

    }

}

public void setResult(double result) {

    this.result = result;

}

@Override

protected void executeConnector() throws Exception {

    if (isPlus()) {

        result = firstOperand + secondOperand;

    } else if (isMinus()) {

        result = firstOperand - secondOperand;

    } else if (isTimes()) {

        result = firstOperand * secondOperand;

    } else if (isDividedBy()) {

        result = firstOperand / secondOperand;

    }

}

@Override

protected List<ConnectorError> validateValues() {

    List<ConnectorError> errors = new ArrayList<ConnectorError>();

    if (isDividedBy() && getSecondOperand() == 0) {

        errors.add(new ConnectorError("secondOperand",

            new IllegalArgumentException("division by 0 are not allowed!")));

    }

    return errors;

}

}

```

Chapter 8. Administration and Execution

This chapter covers administration and execution features available in Bonita BPM through the web 2.0 console. Although these features may be performed by a single user, the Process Console provides those capabilities to users in a role based basis. BPM User and Operator roles will be leveraged in this guide.

This guide provides the User with the information necessary to be able to :

- Start Workflow Processes
- Perform / Suspend / Resume Tasks

This guide provides the Operator with the information necessary to be able to :

- Deploy / Undeploy / Start Bonita process models
- Access Bonita process model instance informations
- Delete Instance
- Edit Process Instance Variables
- Access Bonita Activities informations
- Perform / Suspend / Resume Activity in a specific instance
- Consult / Edit Activity Variables
- Consult Activity's Properties

8.1 Installation

8.1.1 Prerequisite

The Nova Bonita Console works with java 1.5. Be sure that your default JRE is 1.5

8.1.2 Installation procedure

- Get the last version of the Nova Bonita Console from this web page :
http://forge.objectweb.org/project/showfiles.php?group_id=56
- Extract it and go in the **bonita-console-4.1.1/bin** directory
- If your use Linux as Operating system do the following:

```
> chmod +x bonita-console-4.1.1/bin/*.sh
```

```
> unset CATALINA_HOME CATALINA_BASE
```

8.2 Quick Start

In this section we present a quick start documentation for the Nova Bonita Console. In the next chapters we will explain in more details the functionalities available in this release

8.2.1 Console Start

- Open a command line and execute the following under the **bonita-console-2.0/bin** directory:
 - For Linux : **./bpm.sh run**
 - For Windows : **bpm run**
- In your web browser connect to the following URL : <http://localhost:8080/portal/>



Figure 8-35. Login screen

- Connect with :
 - User Name : **root**
 - Password : **bpm**

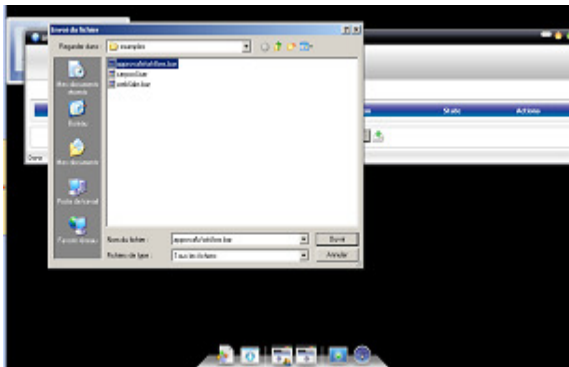


Figure 8-36. Choose a process to deploy




Figure 8-37. Choose a process to deploy


8.2.2 Deploy process

Open the **Bonita Management** application available in the doc bar.

A couple of BPM examples to deploy are available under **Bonita-console-4.1.1/examples/**.

Choose a process to deploy and click on 

8.2.3 Start Process

To start your first process click on the button 

Fill the displayed form and click on the submit button.

So a new instance is created. And the updated instances list is displayed.

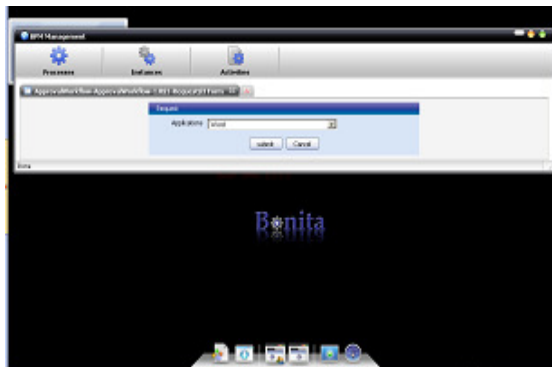


Figure 8-38. Start Process Form

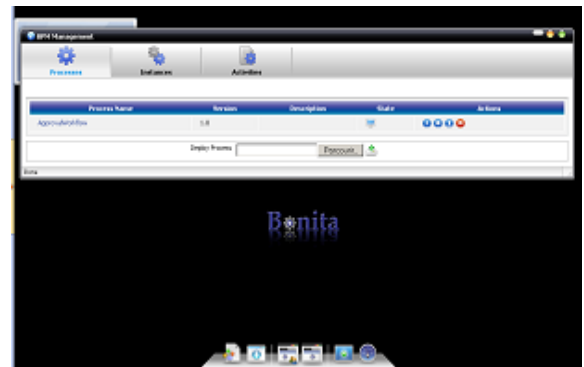


Figure 8-39. The created instance

8.3 Process Console Description

8.3.1 Console Access

To access the Process Nova Bonita Console, connect to the following URL :

`http://your_host:your_HttpPort/portal/`

(by default : <http://localhost:8080/portal/>)



Figure 8-40. Console login screen

8.3.2 Default users

The Nova Bonita Console has three default users types:

root :

This user has the rights to manage the console look and feel, he can also manage :

- The navigations and pages of the console
- The languages setting
- The users, groups and memberships
- The registry of all the Nova Bonita Console applications

Finally, this user has by default the two profiles User and Operator, so he access to all the associated applications.

admin :

This user has the Operator profile so he access to the functionalities described in Chapter 1.2

james, john and jack:

These users have the User profile so they access to the functionalities described in Chapter 1.1

8.3.3 Console frames description

After logging in, the Nova Bonita Console is available in the main frame, of your browser

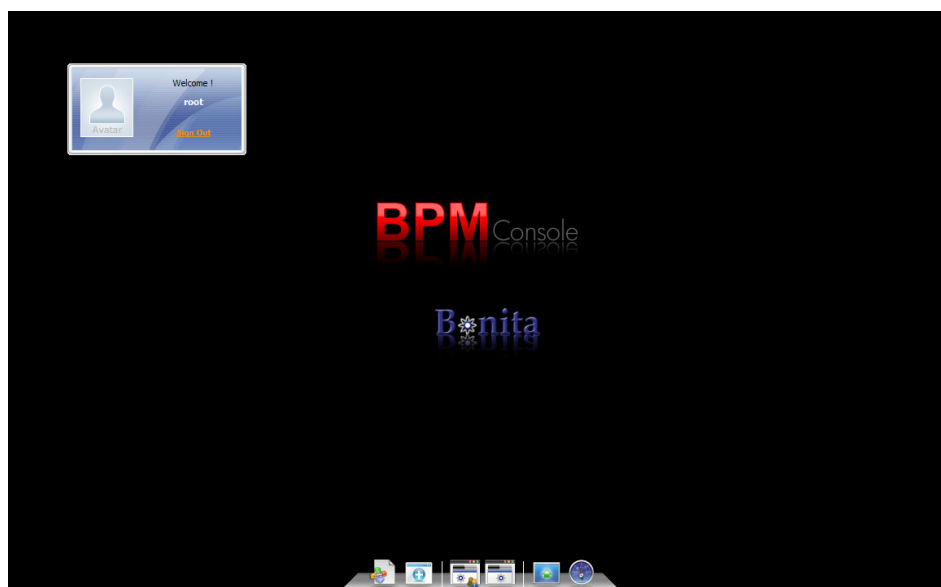


Figure 8-41. Console description

Desktop :

The Desktop is the workspace in which BPM and others applications (aka portlets) can be added, removed, configured... like in an OS

Doc Bar :

The Doc Bar is used to access to the available applications for a particular user type

Applications :

The Application allow to perform the business functionalities like the process management ...etc. Each application is independent from the others. It looks like a simple window.

Pages Navigation Button :

This button allow to display the list of the Console navigations and to navigate between the different pages.

Add Applications Button :

This button allow to add an application/widget in the Doc Bar/Desktop.

Show / Hide Applications :

This button allow to hide all the Applications / Widgets displayed on the desktop. A second click allow to display them again.

Welcome Widget :

The welcome widget display the logged user's name and allow to logout from the Nova Bonita Console.

8.3.4 Application's graphical organization description

Each Application is organized like this :

- **Main Tools :** The main banner that allow to access the main tools.
- **Tabulations :** A set of tabulation for the tools in use. A lot of tabulations can be opened at the same time.
- **Work Area :** The tools display area.

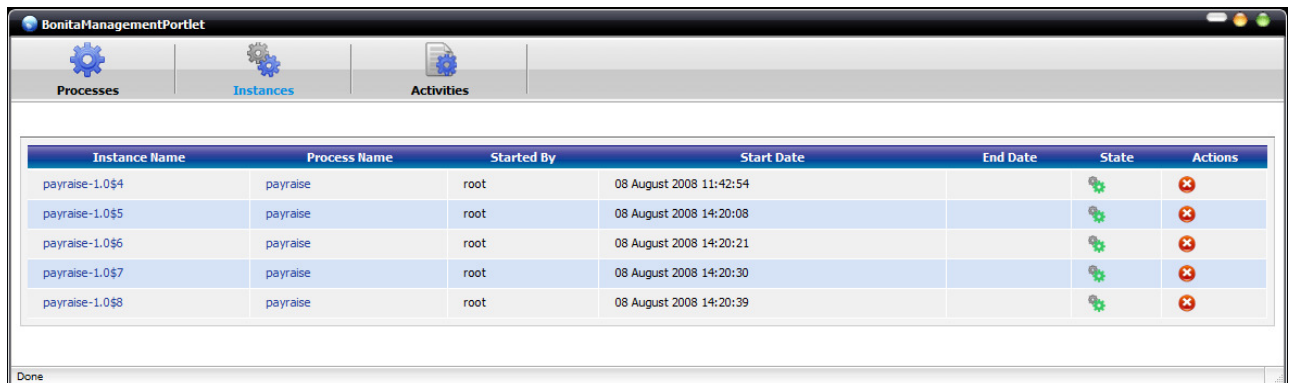


Figure 8-42. Application description

8.4 Accessing and Creating Processes

8.4.1 Access the Workflow Process List

Select the application **Users WorkList** in the **Doc Bar**, then select the "To Do List" tabulation



Todo List in the **Main Tools** to display the list of all the remaining tasks to be performed and all the accessible processes that can be launched.

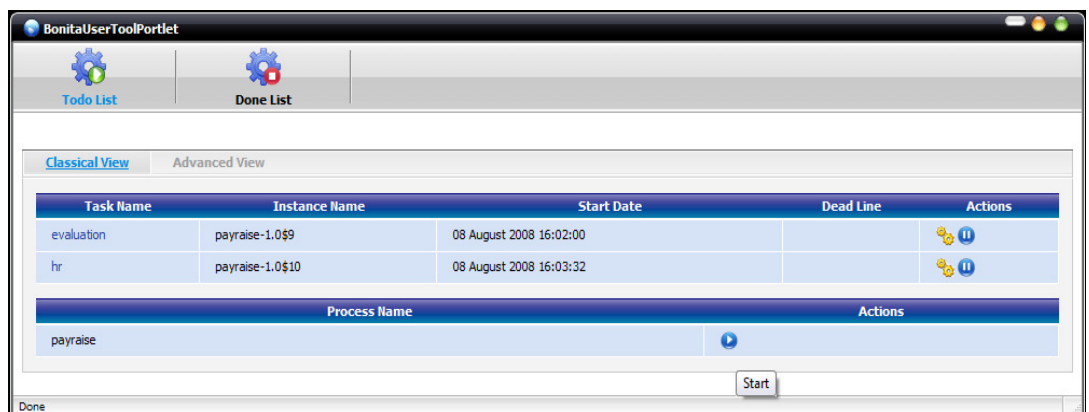


Figure 8-43. Process list

8.4.2 Create a new Instance of a Bonita Process

- Access the workflow process list (see section 4.1).
- To create a new instance of the process, click on the Start button.

- If the start of the process is manual, a form is displayed.
- Fill in the form and click on the submit button as shown in the example below.

The screenshot shows a web application window titled 'BonitaUserToolPortlet'. It has two tabs: 'Todo List' and 'Done List'. The 'Start payraise Form' is active. A modal dialog titled 'Request a pay raise' is displayed. It contains the following fields: 'Amount : *' with the value '150', 'Priority : important' (dropdown), 'Rewarded : ☒', and 'Reason : This year' (text area). At the bottom of the dialog are two buttons: 'Let's pray' and 'Cancel'.

Figure 8-44. New instance creation

8.5 Access To Do / Done Tasks

8.5.1 Consult the To Do Tasks List

- Select the application User Worklist in the **Doc Bar**, then select the "ToDo List" tabulation in the **Main Tools** to display the list of all the To Do tasks remaining to be performed.

- Two different views are possible for the To Do list :

Classical view : This view display the To Do tasks list in a simple grid mode.

The screenshot shows the 'Users Worklist' application window. It has two tabs: 'Todo List' and 'Done List'. The 'Classical View' is selected. It displays a table with the following data:

Task Name	Instance Name	Start Date	Actions
Approval	ApprovalWorkflow-ApprovalWorkflow-1.001	25 September 2008 14:49:20	[icon]
Approval	ApprovalWorkflow-ApprovalWorkflow-1.002	25 September 2008 15:00:53	[icon]
Approval	ApprovalWorkflow-ApprovalWorkflow-1.004	25 September 2008 15:04:00	[icon]
Approval	ApprovalWorkflow-ApprovalWorkflow-1.005	25 September 2008 15:04:03	[icon]
SalesReview	WebSale-WebSale-1.008	25 September 2008 15:05:45	[icon]

Below the main table, there is a summary section with two columns: 'Process Name' and 'Actions'.

Process Name	Actions
ApprovalWorkflow	[icon]
carpool	[icon]
WebSale	[icon]

Figure 8-45. TodoList classical view

Advanced view : This view display the users tasks list sorted by process and activity names. Once clicking in each activity name the application will list the available activities instances corresponding to a particular activity name.

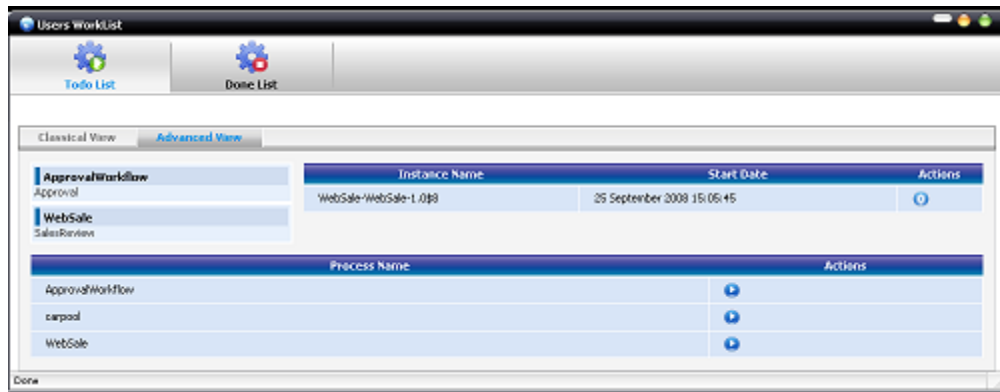


Figure 8-46. TodoList advanced view

- The following parameters are displayed for each task :

- **Task name :** The task name
- **Instance name :** The associated instance name
- **Start Date :** The start date of the task


- A detailed view of a task is available by clicking on the name of the task.

A description text can be displayed if exist.

8.5.2 Perform / Pause / Resume a Task

8.5.2.1 Perform

- Go to the To Do tasks list (see section 5.1).

- To perform a task, click on the  button (perform) in the Actions field of the line corresponding to the task you want to perform (as shown above on the example).

- If this task has properties to be set or read by the user, a form is displayed.

- Fill in the form, then click on the "submit" button as illustrated below.

Figure 8-47. Perform a task

8.5.2.2 Suspend

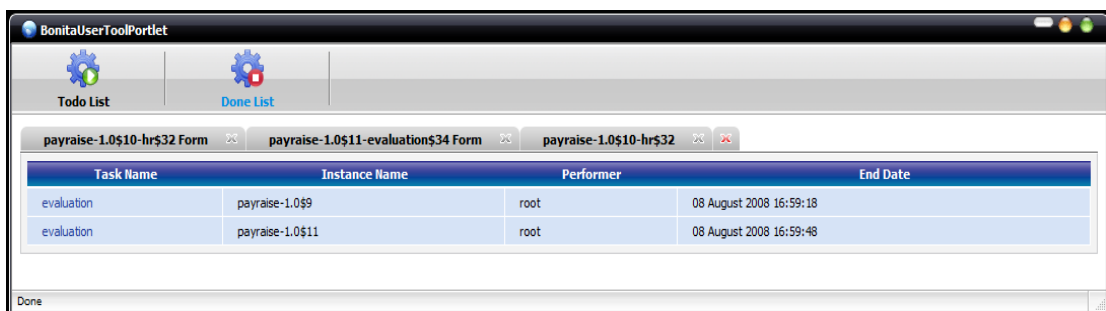
- Go to the To Do tasks list (see section 5.1).
- To suspend a task, click on the button at the end of the line of the task you want to suspend.

8.5.2.3 Resume

- Access the To Do tasks list (see section 5.1).
- To resume a suspended task, click on the button at the end of the line of the task you want to resume (as shown above on the example). Only suspended tasks can be resumed !.

8.5.3 Consult the Done Tasks List

Select the application **Users WorkList** in the **Doc Bar**, then select the "Done List" tabulation in the **Main Tools** to display the list tasks that has already been executed.



Task Name	Instance Name	Performer	End Date
evaluation	payraise-1.0\$9	root	08 August 2008 16:59:18
evaluation	payraise-1.0\$11	root	08 August 2008 16:59:48

Figure 8-48. Done task list

8.6 Managing Process Models

Those features are only accessible by Operators users.

8.6.1 Access the Process Model List

- Select the application **BPM Management** in the **Doc Bar**, then select the processes tabulation







in the **Main Tools** to display the list of all the accessible processes.

- The following parameters are displayed for each process model line :

- **Process name** : The name of the process
- **Version** : The version number of the process
- **Description** : Some description about the process

- The following actions are possible for each process model :

-  starting the process
-  removing all the running instances of this process model
-  deleting a process
-  undeploying a process

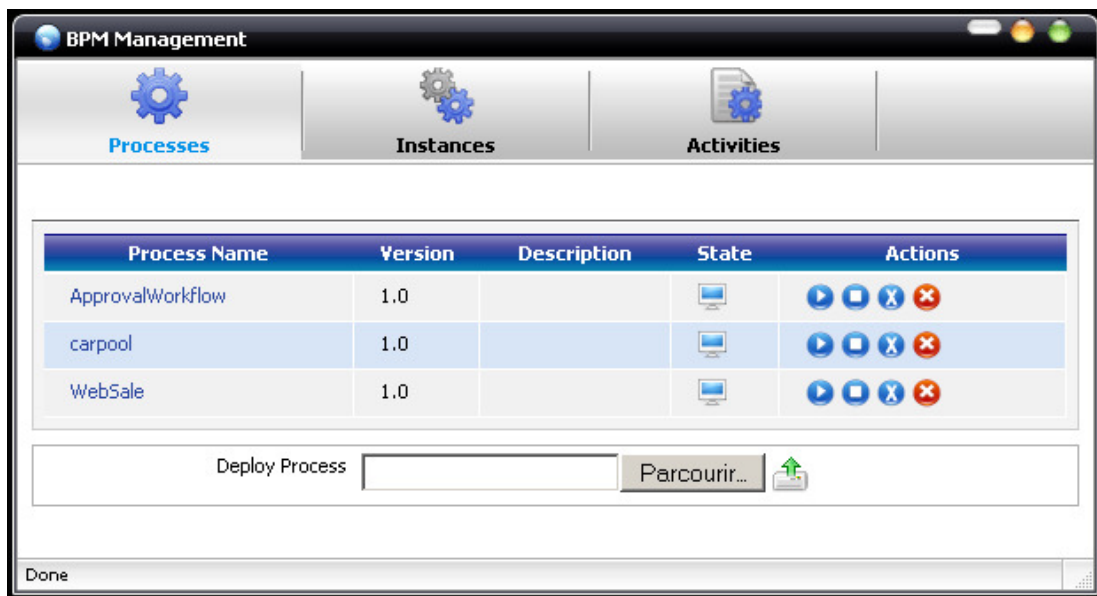


Figure 8-49. Process Model List

-A detailed view of a process model is available by clicking on the name of the process.



Figure 8-50. Process model detailed view

-Associated instances of the process model displayed in the detailed view are available by clicking on "Instances" sub-tabulation.

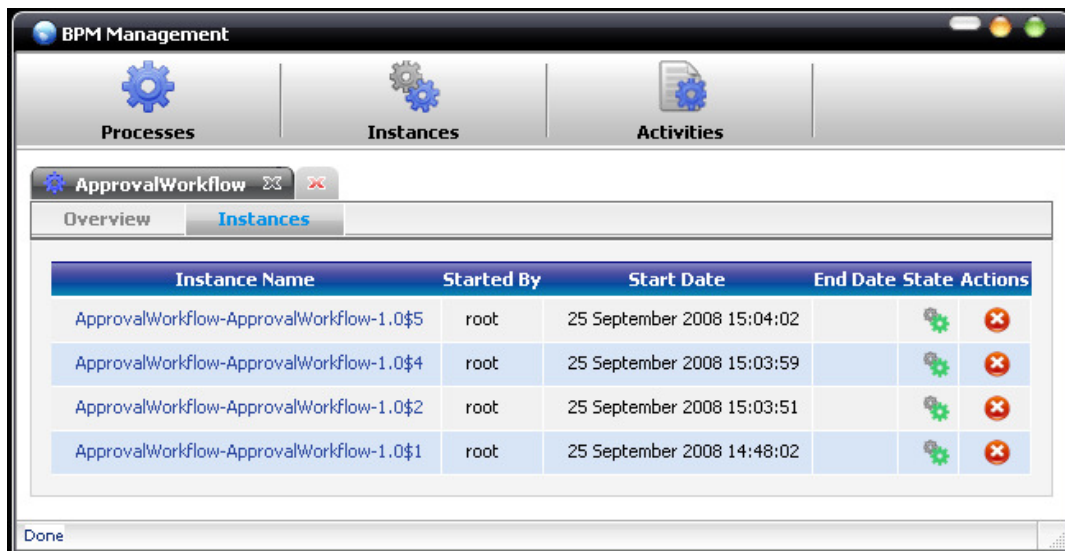


Figure 8-51. Instances list for a given project

8.6.2 Deploy / Undeploy / Delete Processes Models

Go to the process models list (see section 6.1).

Deploy :

To deploy a process model, click on the browse button and select the path to access your process

model, then click on the button (upload) . After deploying operation, the process models list is refreshed with the new imported process model.

Note : You may deploy a single class file or a library jar file with this fonctionnality

Undeploy :

To undeploy a process model, click on the button for the process you want to undeploy. You can also open the detailed view of the process model, then click on the undeploy button in the Actions field.

Note that the undeploying action will keep all historic data in your system.

Delete :

To delete a process model, click on button at the end line of the process you want to delete. You can also open the detailed view of a process model, then click on the delete button in the Actions field.


8.6.3 Start Process Models

- Access the workflow process list.

- To create a new instance of the process, click on the button (Start button). If the start of the process is manual, a form is displayed.

8.6.4 Remove all Instances of a Process Model

- Access the workflow process list

- To remove all the instances of a process, click on the  button

8.7 Managing Instances

8.7.1 Access the Process Instances List






Select the application **Bonita Management** in the **Doc Bar**, then select the Instances tabulation in the **Main Tools** to display the list of all the running instances.

The following parameters are displayed for each process instance :

- **Instance name** : The name of the instance (click to display the detailed view of the instance)
- **Process name** : The name of the associated process model (click to display the detailed view of the process)
- **Started By** : The name of the user that started the instance
- **Start Date** : The date in which the instance was started
- **End Date** : The finish date of the instance
- **State** : The state of the instance

In the following table we present the different states of instances.

State	Meaning
	Started : The instance is in started state
	Finished : The instance is in the finished state
	Initial : The instance is in the initial state

The following actions are possible for each process instance line :

-  Delete the instance

- It is possible to consult a detailed view of a process instance by clicking on the name of the instance.

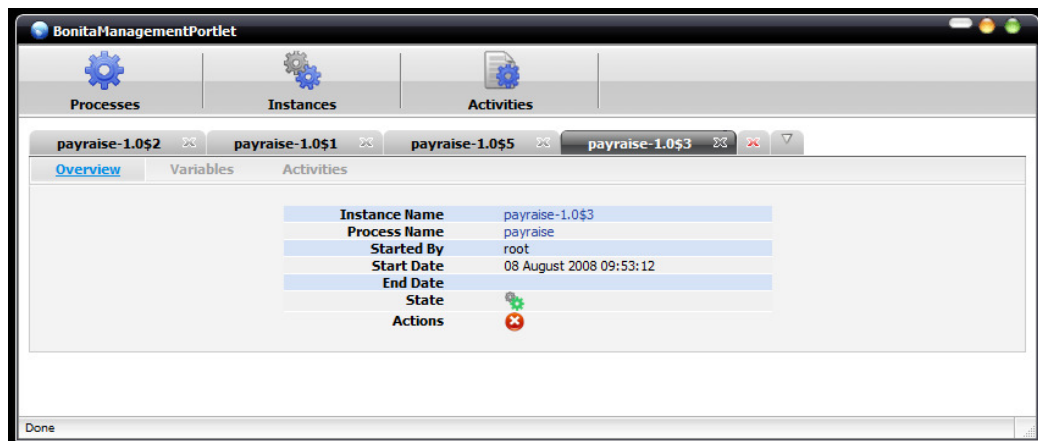


Figure 8-52. Instance detailed view

8.7.2 Consult / Edit the variables of an instance

- Access the instances list
- Click on the instance name to display the detailed view of the instance.
- Click on the "Variables" sub-tabulation to display all the associated variables of the instance. You can also EDIT those variables by clicking on the button and filling the new value in the variable edition popup.

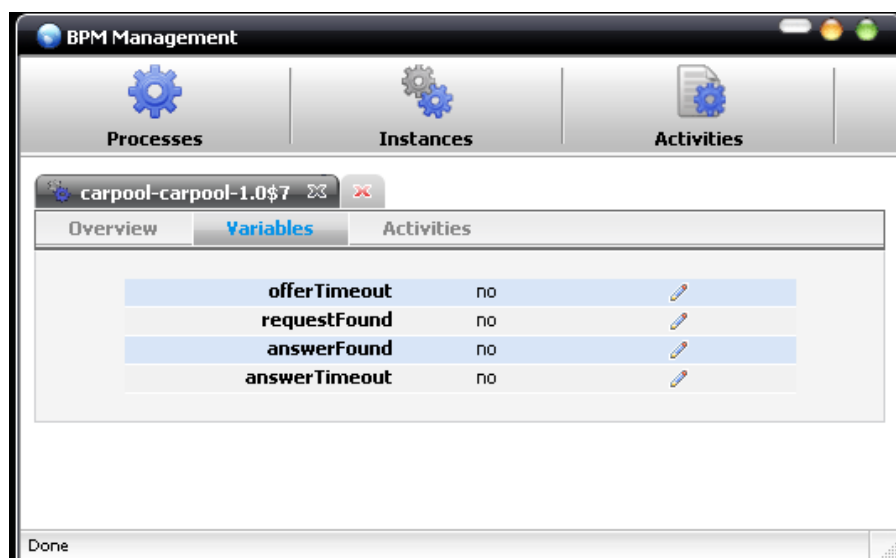


Figure 8-53. Instance Variables

8.7.3 Access the activities list of an instance

- Go to the instances list
- Click on the instance name to display the detailed view of the instance.
- Click on the "Activities" sub-tabulation to display all the associated activities of the instance.

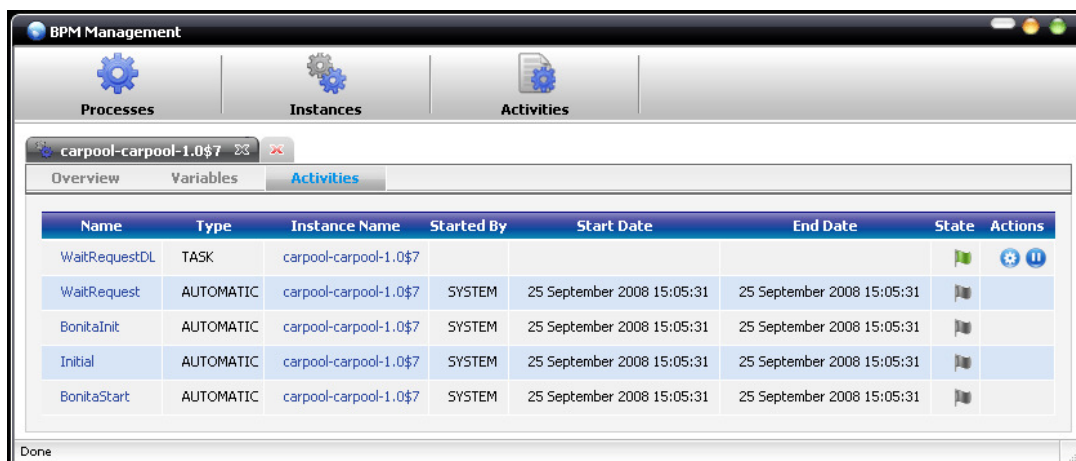


Figure 8-54. Activities for a given instance

8.8 Managing Activities

8.8.1 Access the Activities List







Select the application **Bonita Management** in the **Doc Bar**, then select the Activities tabulation in the **Main Tabulation** to display the list of all the Activities.

The following parameters are displayed for each activity line :

- **Id** : The id of the activity (click to display the detailed view of the activity)
- **Type** : The type of the activity
 - AUTOMATIC : Automatic execution of the activity
 - TASK : manual activity that requires human interaction
 - SUBFLOW : subprocess activity that creates a subprocess
- **Instance name** : The name of the instance that triggered the activity (click on to display the detailed view of the instance)
- **Started By** : The name of the user that started the activity
- **Start Date** : The lunch date of the activity
- **End Date** : The finish date of the activity
- **State** : The state of the activity

In the following table you will find the different states of the activities.

State	Description
	Ready : The activity is ready to be started

	Initial : The activity is in the initial state
	Executing : The activity is in execution
	Finished : The activity has already been finished
	Suspended : The activity has been suspended


- A detailed view of an activity is available by clicking on the name of the activity.



Figure 8-55. Activity detailed view

8.8.2 Start an Activity

- Go to the activities list


- To perform an activity, click on the  button in the Actions field of the line corresponding to the activity you want to start (as shown above on the example).

- If this activity has properties to be set or read by the user, a form is displayed.

- Fill in the form, then click on the "submit" button.

8.8.3 Suspend an Activity

- Access the activities list.

- To suspend an activity, click on the  button at the end of the line of the activity you want to suspend (as shown above on the example).

8.8.4 Resume an Activity

- Access the activities list.

- To resume a suspended activity, click again on the  button at the end of the line of the activity you want to resume.

8.8.5 Access the Variables List of an Activity

- Access the activities list.

- Click on the instance name to display the detailed view of the activity.

- Click on the "Variables" sub-tabulation to display all the associated variables of the activity.

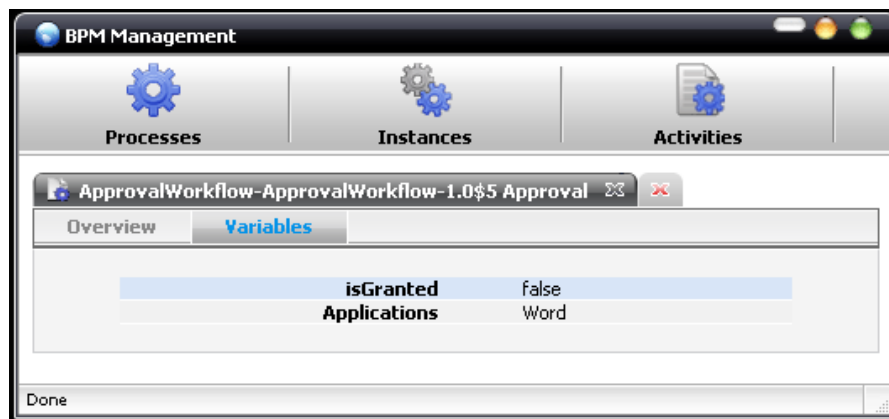


Figure 8-56. Activities variables

8.9 Managing users, groups and memberships

Those functionalities can be done only by the **root** user.

This user is allowed to view the left side workspace bar that gives access to administration features of the Console.

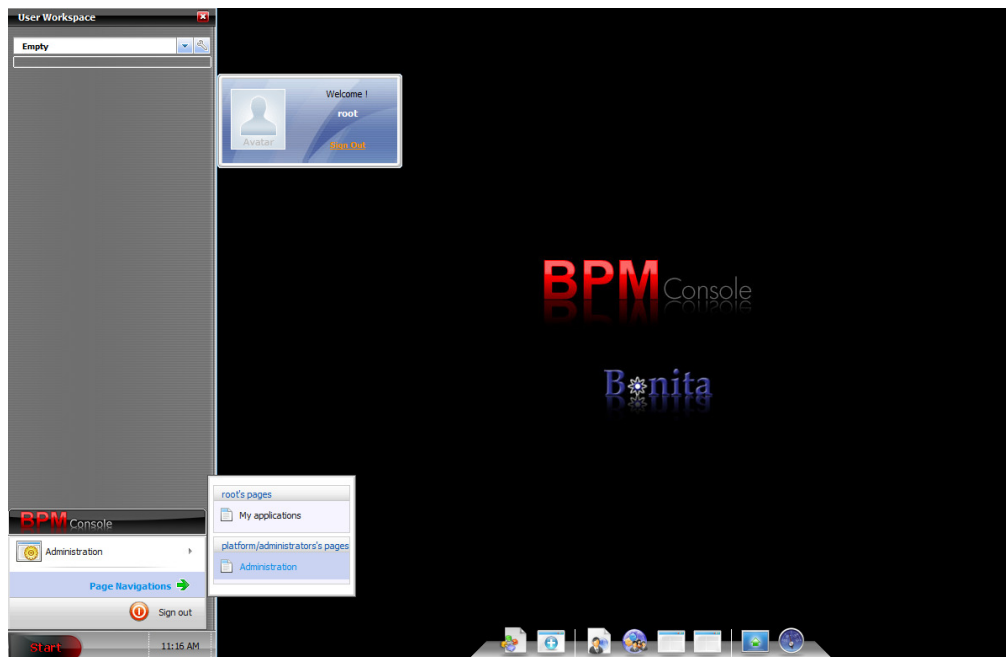


Figure 8-57. Administration page

8.9.1 Add a new user to the Nova Bonita Console

Then open the "New Account" application, and fill the form with the informations of the new user and finally click on "Save " button to validate.

Figure 8-58. Add a new user

8.9.2 Set roles/permission of a user to access to the console



- Go to the Administration page with the "**Pages Navigation**" button or the root navigation
- Then open the "**Community Management**" application.
- Click on "**Group Management**" tabulation.
- Select the group **Platform > Console > Bonita**
- After that just add the user you want to give access to the console with at least the membership "**user**". And repeat the operation if you want that user has the role "**operator**".

Example : Setting the user “rodrigue” as operator of the console means that we need to put it in the group Bonita with 2 memberships : user and operator.

Step 1 : Select the right group

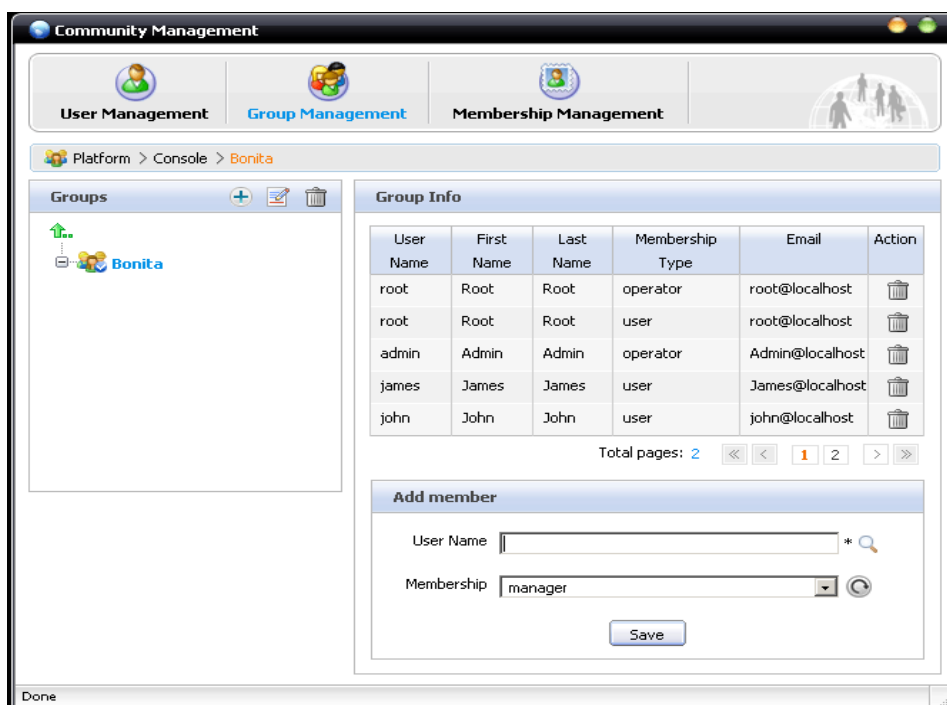


Figure 8-59. Group selection

Step 2: fill the Add member form with user name rodrigue and Membership user and click on Save

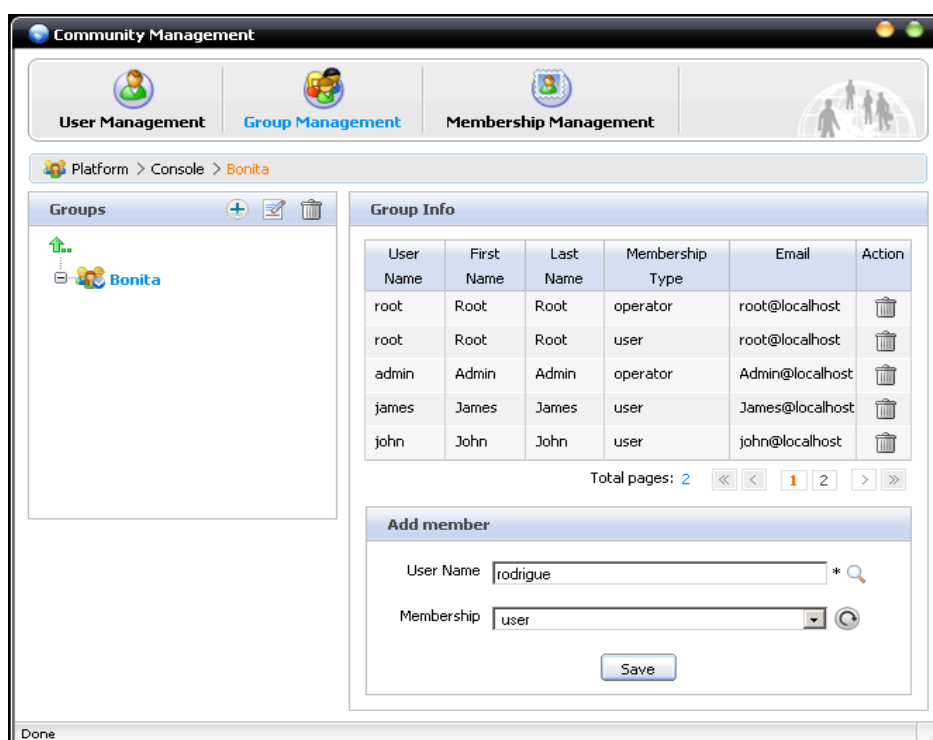


Figure 8-60. Add member dialog (1/2)

Step 3 : fill the Add member form with user name rodrigue and Membership operator and click on Save

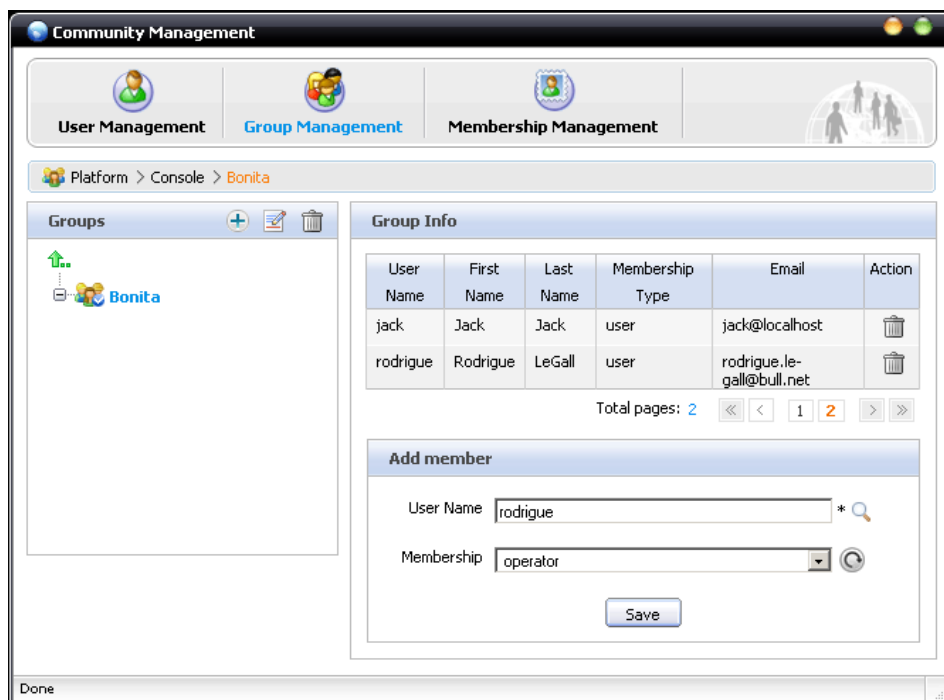


Figure 8-61. Add member dialog (2/2)

The user is now an operator of the Nova Bonita Console.

8.10 Forms customization

8.10.1 Overview

The Bonita console is built with an automated form generator. This functionality is useful during the conception and testing phases of your processes. The major inconvenient of the automated form generation is that generated forms are not user friendly. In order to solve this problem, the form generator is customizable via few configuration files.

To customize the forms of a Bpm process, you just need to write a description file named **forms.xml**. Each web form can be internationalized by means of a properties files. All those files must be located in the root directory of the bar (Business Archive) file of your process.

This chapter will explain how to write **forms.xml** file and the internationalized property files.

8.10.2 Forms.xml syntax

The forms.xml syntax is based on an xml syntax initially derived from the xFormsyntax.

The form generator suppose that you have only one form by manual activity in your process.

Here is the abstract of all tags you need to know to customize your forms

```

<forms>
  <form>
    <activity/>
    <resource-bundle/>
    <customized-view/>
    <variable>
      <validator>
        <property/>
      </validator>
    </variable>
    <submitbutton/>
    <message/>
  </form>
</forms>

```

8.10.2.1

Tag List

<forms/>

Description : top level tag of the forms description.

Mandatory : true

Properties : none

Childs : <form/>

<form/>

Description : include the description of a specific activity form

Mandatory : true

Properties : none

Childs : <activity/>, <resource-bundle/>, <customized-view/>, <variable/>, <message/>

<activity/>

Description : include the id of the activity in your process model. If you want to write a form to start a process, you need to enter an empty value.

Mandatory : true

Properties : none

Childs : none

Examples :

Write a form for the activity "validation" of your process:

```

<forms>
  <form>
    <activity>validation</activity>
    ...
  </form>
  ....
</forms>

```

Write a form to start your process :

```

<forms>
  <form>
    <activity></activity>
    ...
  </form>
  ....
</forms>

```

<resource-bundle/>

Description : the resource bundle includes the translation of each variable names into human friendly language. The value refere to a .properties file : <resource-bundle>.properties

Mandatory : false

Properties : none

Childs : none

Examples :

Write a form for the activity "validation" that will use the following files for translation in english, french and spanish : validation.i18n.properties , validation.i18n_en.properties, validation.i18n_fr.properties, validation.i18n_es.properties

```

<forms>
  <form>
    <activity>validation</activity>
    <resource-bundle>validation.i18n</resource-bundle>
    ...
  </form>
  ....
</forms>

```

<customized-view/>

Description : specify the local path of a template file that will be used for the facing of your form. This template have to be written in groovy.

Mandatory : false

Properties : none

Childs : none

<submitbutton/>

Description : define a custom submit button that will set the variable of the process or activity.

Mandatory : false

Properties :

- name

Description : the name of the button. This is also the value that will be put in the variable target

Mandatory : true

Possible values : none

Default value : none

- variable

Description : the variable of the process or activity that will be modify after the submit with the name value.

Mandatory : true

Possible values : none

Default value : none

Childs : none

<variable/>

Description : define a variable of the process that will be used in the form.

Mandatory : false

Properties :

- name

Description : the id of the variable in the process

Mandatory : true

Possible values : none

Default value : none

Example :

Write a form for an activity "validation" that use a variable "comment".

```

<forms>
  <form>
    <activity>validation</activity>
    <variable name="comment"/>
  </form>
  ...
</forms>

```

- component

Description : the type of widget you want to use in your form to interact with your variable

Mandatory : false

Possible values : text, textarea, select, checkbox, radiobox, wysiwyg, date, date-time

Default value : text

Example :

Write a form for an activity "validation" that use a variable "grant" with a checkbox.

```

<forms>
  <form>
    <activity>validation</activity>
    <variable name="grant" component="checkbox"/>
  </form>
  ...
</forms>

```

- editable

Description : specify if the variable is editable or not

Mandatory : true

Possible values : true, false

Default value : true

Example :

Write a form for an activity "validation" that use a variable "explanation" that should only be printed and not modified.

```

<forms>
  <form>
    <activity>validation</activity>
    <variable name="explanation" editable="false"/>
  </form>
  ...
</forms>

```

- mandatory

Description : specify if the variable must be filled or not

Mandatory : false

Possible value : true, false

Default value : false

Example :

Write a form for an activity "validation" that use a variable "amount" that must be filled.

```
<forms>
  <form>
    <activity>validation</activity>
    <variable name="amount" mandatory="true"/>
  </form>
  ....
</forms>
```

Childs : <validator/>

<validator/>

Description : add a validator for the data of the parent variable. If the validator failed, an error is show in the form.

Mandatory : false

Properties :

- name

Description : the name of the validator

Mandatory : true

Possible values : DateTime, EmailAddress, Expression, Number, NumberInRange, PositiveNumber, SpecialCharacter, StringLength, Float

Default value : none

Childs : <property/>

Example :

Wite a form for activity "validation" with a variable "amount" that must be a Float and mandatory.

```
<forms>
  <form>
    <activity>validation</activity>
    <variable name="amount" mandatory="true">
      <validator name="Float"/>
    </variable>
  </form>
  ....
</forms>
```

<property/>

Description : define a property needed by a validator

Mandatory : false

Properties :

- name

Description : the name of the property

Mandatory : true

Possible values : none

Default value : none

- value

Description : the value of the property

Mandatory : true

Possible values : none

Default value : none

Childs : none

Example :

Write a form for activity "validation" with a variable "amount" that must be between 100 and 10000.

```
<forms>
  <form>
    <activity>validation</activity>
    <variable name="amount">
      <validator name="NumberInRange">
        <property name="min" value="100"/>
        <property name="max" value="10000"/>
      </validator>
    </variable>
  </form>
  ....
</forms>
```

8.10.2.2

Data Validators list

DateTime

Description : Valides that the data is a date

Properties : none

Example :

Write a form for an activity "schedule" with a variable meeting

```
<forms>
  <form>
    <activity>schedule</activity>
    <variable name="meetingDate" component="date-time">
      <validator name="DateTime"/>
    </variable>
  </form>
  ....
</forms>
```

EmailAddress

Description : Validates that the data is an email address

Properties : none

Example :

Write a form for an activity "signup" with a variable "email".

```
<forms>
  <form>
    <activity>signup</activity>
    <variable name="email" mandatory="true">
      <validator name="EmailAddress"/>
    </variable>
  </form>
  ....
</forms>
```

Expression

Description : Validates that the data matches one regular expression.

See <http://java.sun.com/j2se/1.5.0/docs/api/java/util/regex/Pattern.html#sum> for the regular expression format.

Properties :

- expression

Description : this is the regular expression that is conform to the format describe here <http://java.sun.com/j2se/1.5.0/docs/api/java/util/regex/Pattern.html#sum>

Mandatory : true

Example :

Write a form for an activity "signup" with a variable "email" validated with your own email pattern.

```

<forms>
  <form>
    <activity>signup</activity>
    <variable name="email" mandatory="true">
      <validator name="Expression">
        <property name="expression" value="^[A-Z0-9._%+-]+@[A-
Z0-9.-]+\.[A-Z]{2,6}$"/>
      </validator>
    </variable>
  </form>
  ....
</forms>

```

Number

Description : Validates that the data is a number (integer).

Properties : none

Example :

Write a form for activity "validation" with a variable "amount" that must be a number.

```

<forms>
  <form>
    <activity>validation</activity>
    <variable name="amount">
      <validator name="Number"/>
    </variable>
  </form>
  ....
</forms>

```

NumberInRange

Description : Validates that the data is a number in a specified range

Properties :

- min

Description : the minimum of the range

Mandatory : true

- max

Description : the maximum of the range

Mandatory : true

Example :

Write a form for activity "validation" with a variable "amount" that must be between 100 and 10000.

```

<forms>
  <form>
    <activity>validation</activity>
    <variable name="amount">
      <validator name="NumberInRange">
        <property name="min" value="100"/>
        <property name="max" value="10000"/>
      </validator>
    </variable>
  </form>
  ...
</forms>

```

PositiveNumber

Description : Validates that the data is a positive number

Properties : none

Example :

Write a form for activity "validation" with a variable "amount" that must be a positive number.

```

<forms>
  <form>
    <activity>validation</activity>
    <variable name="amount">
      <validator name="PositiveNumber"/>
    </variable>
  </form>
  ...
</forms>

```

SpecialCharacter

Description : Validates that the data only contains letters, digits, '-', '_' or space

Properties : none

Example :

Write a form for activity "validation" with a variable "id" that must not contain any special characters

```

<forms>
  <form>
    <activity>validation</activity>
    <variable name="id">
      <validator name="SpecialCharacter"/>
    </variable>
  </form>
  ...
</forms>

```

StringLength

Description : Validates that the data is a string with a length in a range

Properties :

- min

Description : the minimum of the range

Mandatory : false

Default value : 0

- max

Description : the maximum of the range

Mandatory : true

Example :

Write a form for activity "signup" with a variable "login" with a length between 4 and 15.

```
<forms>
  <form>
    <activity>signup</activity>
    <variable name="login">
      <validator name="StringLength">
        <property name="min" value="4"/>
        <property name="max" value="15"/>
      </validator>
    </variable>
  </form>
  ....
</forms>
```

Float

Description : Validates that the data is a float (ie: 100.45)

Properties : none

Example :

Write a form for activity "validation" with a variable "amount" that must be a Float.

```
<forms>
  <form>
    <activity>validation</activity>
    <variable name="amount">
      <validator name="Float"/>
    </variable>
  </form>
  ....
</forms>
```

8.10.3 Internationalize your forms

To internationalize a form, you need to provide a resource bundle (see **<resource-bundle/>**). The name of the resource bundle corresponds to the principal part of the name of internationalized property file.

Example: the resource bundle for the activity "**validation**" is "**validation.i18n**", then the corresponding files will be **validation.i18n.properties** or **validation.i18n_<locale>.properties** where locale is en for english, fr for french, es for spanish, ...

So if you want to internationalize you forms, you need to write a file by language and by activity and refer them in the **forms.xml** file by using **<resource-bundle/>** tag. These files must be included in the root directory of the BAR (Business Archive) file of your process.

8.10.3.1 Syntax

The syntax of the *.properties files are very simple, each line may contains a couple key/value like this : **key=value** .

To comment a line you need to start this line with #.

Example :

```
task-name=update the salary system
title=Update Salary

submit=Confirm update
cancel=Cancel this form

#variable title and label

initiator.label=Employee :
amount-granted.label=Amount to be added :
```

Key list

Use Case	Key
Task name	task-name
Form title	title
Submit button	submit
<Submitbutton/> button	buttonName. submit
Cancel button	cancel
Variable label	<i>variableId</i> . label
Checkbox value	<i>variableId</i> . checkbox

Radiobox values	<i>variableId</i> . radiobox-0 <i>variableId</i> . radiobox-1 ... <i>variableId</i> . radiobox-<i>n</i>
Select values	<i>variableId</i> . select-0.label <i>variableId</i> . select-0.value <i>variableId</i> . select-1.label <i>variableId</i> . select-1.value ... <i>variableId</i> . select-<i>n</i>.label <i>variableId</i> . select-<i>n</i>.value

Chapter 9. Change history between Bonita v3 and v4

Main concepts and features that made the friendly usage and the Bonita v3 brand have been kept: hooks, role mapper, performer assignments, local/global variables, rich and powerful API. Most of these features have been revisited in order to become even more efficient thanks to the PVM execution environment.

Aim was to be the most compatible with the last version but of course some changes are required. Goal of this chapter is to list/focus all these differences.

9.1 Concept of package

The concept of package has been introduced by XPDL specification from the WfMC in order to be a container for main workflow objects that can be shared by multiple workflow processes that can support an overall business application. Amongst these elements are: participants, datafields, others process workflows/subflows.

This concept has been natively taken in account by Nova Bonita engine. According the requirements and needs of our customers this concept should be enforced.

9.1.1 Package life cycle

States for package: UNDEPLOYED, DEPLOYED

9.2 Processes, instances, activities and tasks life cycles

One major change concerns the adding of task entity. If the activity is manual (ie. startMode>manual) when the execution enters the graph node of the activity a task is created. This task has its own life cycle with some synchronisation with the activity entity. Within this version task is still managed by the engine but in the future, it will be possible to plug an external task module to manage the tasks.

9.2.1 Process life cycle

States for process: UNDEPLOYED, DEPLOYED

Deployment of processes implies deployment of a package. Same thing for the undeployment. Package can be deployed and undeployed several times in order to make modifications onto its contained elements (process, participants, activities, ...). This is the way to maintain processes before the introduction of versioning in next version.

9.2.2 Instance life cycle

States for process instance: INITIAL, STARTED, FINISHED

No difference with bonita v3.

9.2.3 Activity life cycle

Activities state depends on the type of behavior defined within the activity. A specific body of the activity is created according the type of the activity (Task, Subflow, Route, Automatic). The state is implemented by the body.

Activities states are: INITIAL, READY, EXECUTING, FINISHED, CANCELED and ABORTED

9.2.4 Task life cycles

State SUSPENDED has been introduced. This state can be reached either from READY or from EXECUTING ones.

Tasks are particular types of activities (such subflow, route...) associated to human actors.

9.3 APIs

Bonita v3 APIs were divided into 5 different areas and can be compared to bonita v4 API:

- **ProjectSessionBean:** is covered by both DefinitionAPI (for set/add methods) and QueryDefinitionAPI (for get methods)
- **UserSessionBean:** is covered by both RuntimeAPI and QueryRuntimeAPI
- **AdminSessionBean:** has fonctions that could be found into QueryDefinitionAPI and QueryRuntimeAPI according on the type of information (runtime or definition information). At now there's no check for admin role
- **UsersRegistrationBrean:** is not relevant for bonita v4 because user base is not managed by the engine.
- **HistoryAPI:** is covered by QueryRuntimeAPI.

Bonita v3 can only be acceded as a remote workflow server. Bonita v4 supports both java workflow library and remote workflow server.

A new API has also be added for improving workflow processes deployment as well as advanced entities deployment: hooks, mappers and performers assignments. This API is called ManagementAPI. No need anymore to deploy xpd1 first and the compile and copy by hand advanced entities in a particular server directory. Any deployment/undeployment operation can be performed through the ManagementAPI.

Furthermore Nova Bonita v4 provides extensibility to the APIs by the addition of the commandAPI. Developers are now free to write and execute its own commands and consequently can extends the proposed API. This is a service oriented feature and it also should avoid to provide a querier language for complex requests (involving requests with multiple criteria).

9.4 Hooks

Pieces of end-users java code that are executed at particular moments of either a process instance or a task or an "automatic activity" (route and subflow type activity cannot have hook).

9.4.1 For tasks

Xpdl definition of hooks has changed in order to extend rollbacking capabilities to all hook types making the way the usage of hook simpler. An example of the new one is given here after:

```
<ExtendedAttribute Name="hook"
  value="org.ow2.bonita.integration.hook.OnReadyHook">
  <HookEventName>task:onReady</HookEventName>
  <rollback>false</rollback>
</ExtendedAttribute>
```

The element rollback has been introduced to indicate if the hook will be or not rollbacked. Hook events have also been adapted to match the constraints of task life cycle.

- task:onReady
- task:onStart
- task:onFinish
- task:onSuspend
- task:onResume

Main change is the suppression of before/after for terminate and before/after start types because of the introduction of the rollback parameter. Another change is the introduction of new events due to the new state: SUSPENDED.

Note: if using proEd, the designer can select for each hook:

- rollback=true (case1)
- rollback=false (case2)

Hooks are always executed into a transaction. In case1, if an exception has occurred the exception is raised by the engine and the transaction is rollbacked. In case2, the occurring exception is caught by the engine.

To implement a hook class the developer has the choice between two interfaces. Look at the javadoc of these interfaces for more details:

- org.bonita.ow2.definition.TxHook
- org.bonita.ow2.definition.Hook

If rollback=false has been previously defined, only Hook interface can be implemented otherwise an exception is raised at runtime. Then it prevents the use of TxHook interface. These hooks are intended to execute not critical operations for the workflow. Only query API are proposed to be acceded into the parameters of the execute() method of the interface.

If modification on hook class is required it can be hot deployed to replace the previous one (see the ManagementAPI). It can be also deployed within the bar archive or independently. It can be also undeployed if the class is not required by a deployed process.

9.4.2 For automatic activities

One type of event can be defined: automatic:onEnter

9.4.3 For processes

ON_INSTANTIATE hook (set within the process element of XPDL definition) is not yet supported. ON_CANCELLED hook is not yet supported.

9.4.4 Interactive hook

Interactive hooks (also called Bean Shell) are not yet supported for activity and process. Those hooks will be implemented soon adding support for others scripting languages such Groovy

9.5 Deadlines

Deadline feature within Bonita v4 is the same as for Bonita v3. `org.ow2.bonita.definition.TxHook` or `org.ow2.bonita.definition.Hook` interface must be implemented in case of deadline hook (ON_DEADLINE event). See javadoc for more details.

9.6 Mappers

`org.ow2.bonita.definition.RoleMapper` interface must be implemented (see javadoc for more details).

Main difference concerns the moment in which the `searchMembers()` method is executed. In Bonita v3 it was executed at process instantiation since in Bonita v4 it is at the creation of the task from which the activity has been defined with a role mapper. It has the advantage to take in account modification of the groups within the external user base

9.7 Performer assignments

`org.ow2.bonita.definition.PerformerAssign` interface must be implemented (see javadoc for more details).

9.8 Variables

Properties entity in Bonita v3 has been renamed to Variables in Bonita v4. This seems a more natural way to work with workflow relevant data.

Variables support and flexibility in Bonita v3 was too limited. Only String and enumerated types were supported. In Bonita v4 support for common variables types as well as as advanced ones (including own Java based ones) will be added in next releases (currently, the v4 version support same types than v3 and adds: Float, Integer, Boolean, Datetime, Performer).

Getting and Setting variables operations directly handles Java Objects, meaning that a get operation returns an Object so the developer only needs to use the `instanceOf` operator to determine the type of a particular variable.

9.9 Iterations

Iterations support in Nova Bonita follows the innovative mechanism included in Bonita v3, meaning supporting complex and advanced uses cases: unstructured iterations or arbitrary cycles.

Main difference between Bonita v3 and v4 related to iterations is that in v4 there is no need anymore for a dedicated entity called iteration. Transitions can be used in Bonita v4 to create a cycle in a process.

For compatibility reasons iterations entities defined as XPDL extended attributes (Bonita v3) are still supported.

The current implementation has some restrictions:

- A cycle must have at least one XOR entry point
- Split activities as exit points are only supported in case of XOR
- Join XOR inside iterations do not cancel/delete non selected execution path

Those restrictions will be fixed in the next release with the addition of a new behaviour for XOR activities in which non selected execution path will be automatically deleted/removed