Nova Bonita Workflow

Development Guide

Nova Bonita Workflow
Development Guide

Nova Bonita (aka Bonita v4)

Software

August 2008

Table of Contents

List of Figures

List of Tables

Chapter 1. Overview

This document describes the design and development process for the Nova Bonita Workflow. Although design and development activities may be performed by a single individual, they require the use of separate tools, and therefore are described as two separate roles.

The information in this document is organized as follows:

FOR THE DESIGNER

- The ProEd XPDL Workflow Process Editor
  Refer to Chapter 2.

FOR THE DEVELOPER

- Hooks
  Refer to Chapter 3.

- Mappers and Initiator Mappers
  Refer to Chapter 4.

- Performer Assignment
  Refer to Chapter 5.

1.1 Role of Designer

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

- To create or modify Process Models using the ProEd Workflow editor.

1.2 Role of Developer

At different points during the Workflow process, Nova Bonita Workflow process models may call external Java classes to perform specific tasks.

These Java classes are divided into four categories, according to the task they are to perform during a Workflow process. The four types of Java classes that can be involved in a Workflow process are:

- **Hooks**: triggering automatic actions at specific moments during the process or during an activity.
- **Mappers**: specifying the person(s) corresponding to a specific role.
- **Initiator Mapper**: specifying the person(s) allowed to start the process.
- **Performer assignment**: refining the assignment of a Participant to an activity.

This guide provides the developer with the information necessary to:

- Add hooks, mappers, initiator mappers, and performer assignment entities, to a Workflow process definition.
- Compile and deploy those entities in the Nova Bonita Workflow environment (where those entities are Java classes).

Chapter 2.   Using the ProEd Workflow Process Editor

2.1   ProEd Overview

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

The ProEd graphics-based tool allows the user to visually describe a Workflow process using standard BPMN (Business Process Modeling Notation) graphic notation. All elements of the Workflow 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 Workflow process can be saved using the XPDL standard notation.

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

2.2   Versioning Support in ProEd

ProEd now supports versioning of the Workflow process. Each Workflow 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 Workflow project is created with an initial version of 1.0. If an existing Workflow 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 Workflow 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 Workflow process's XPDL file; however, the following format is recommended and will be proposed in the dialogs whenever a new file name is required:

```
WorkflowProcessName_version.xpdl
```
For example:
```
MedicalWorkflow_1.0.xpdl
```

2.3   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.

2.3.1 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 4.1.2 version (proed_standalone_4.1.2.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 up to 1.6.4 (go to http://ant.apache.org/bindownload.cgi for downloading this project).

2.3.2 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 4.1.2 version (org.objectweb.proed4.1.2.zip file).

In order to install the plugin in your Eclipse environment just unzip this file on your eclipse installation directory. ProEd 4.1.2 has been released for Eclipse up to 3.2 version.

Once unzipped, just restart your Eclipse, go to "File" menu and then "New->Other" and select ProEd XPDL file. Then just follow the instructions to create and visualize your first XPDL file in ProEd.

2.4 Quickstart

2.4.1 Creating a New Workflow Project

A new Workflow 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 2-1.   Creating a New Workflow Project

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

- **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 Workflow Process. New Workflow Processes will be created with a version of 1.0.

- **Author:** enter the name of the Workflow Process model Designer.

- **State of Publication**: select the appropriate state of publication, depending on thee specific Workflow design progress.

- **Role initiator**: this selection specifies which users are authorized to start the Workflow 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 Workflow 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 Workflow Process deployment.
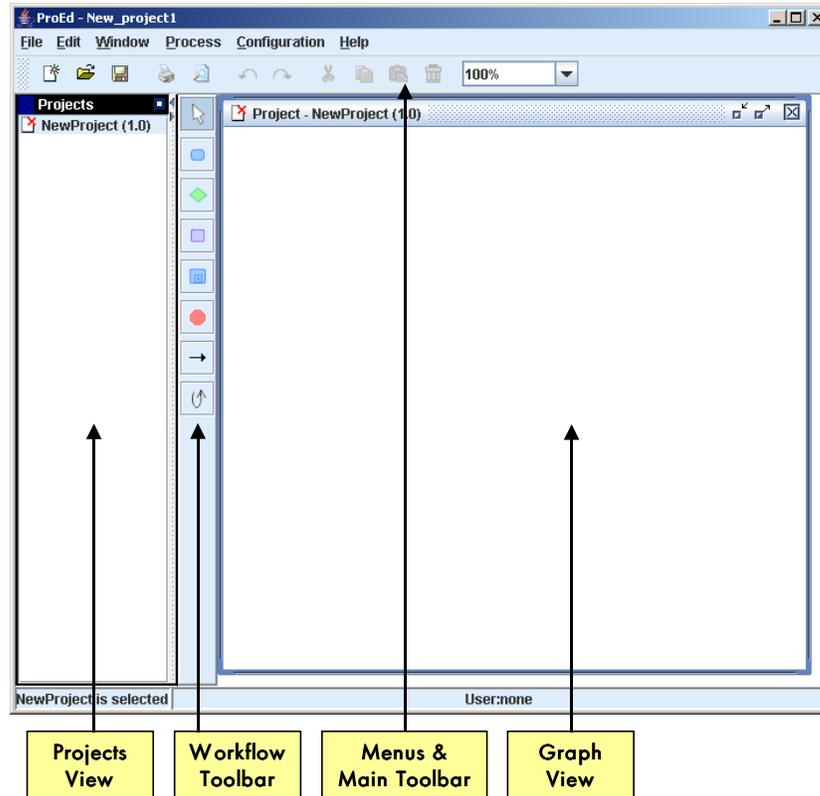
Figure 2-2.   ProEd Display for New Project

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 Workflow 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 2.4.2 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.

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

2.4.2　　　Interface Overview

File Menu

Figure 2-3.　　ProEd File Menu

- **New**: creates a new project. The **New Project** window appears (see "Creating a New Workflow Project").

- **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

Figure 2-4.   ProEd Edit menu

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

Window Menu

Figure 2-5.   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 Workflow process by Participants as shown below.
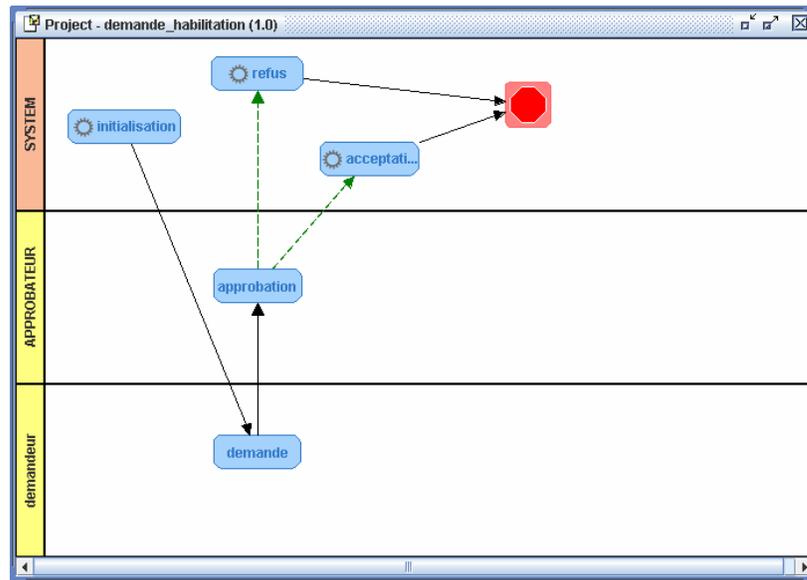
Figure 2-6.   ProEd Participant View

- **Activity view**: organizes the graph of the Workflow process by Activities as shown below.
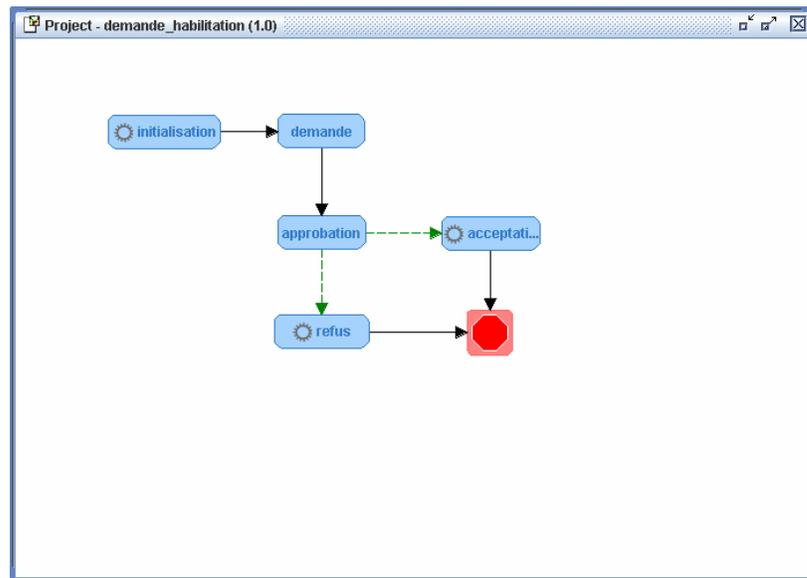
Figure 2-7.   ProEd Activity View

Process Menu

Figure 2-8.   ProEd Process Menu

- **Properties**: displays the properties of the process.
- **Activities**: displays all basic activities of the process.
- **Route activities**: displays all route activities of the process.
- **Block activities**: displays all block activities of the process.
- **Sub-process**: displays all sub-processes of the process.
- **Transitions**: displays all transitions of the process.
- **Iterations**: displays all iterations 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
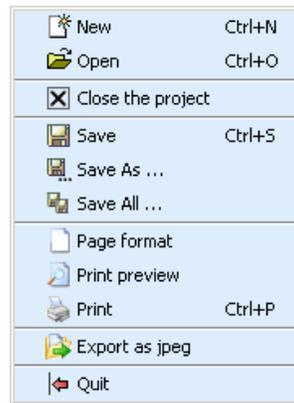
Main Toolbar

Figure 2-9.   ProEd Main toolbar

- **New**: creates a new project. The **New Project** window displays (see "Creating a New Workflow Project").

- **Open**: open a XPDL file containing process definition(s).

- **Save**: saves the current process definition into a XPDL file. If a filename has not been defined, the "**save as**" dialog box will open.

- **Print**: prints the graph corresponding to the currently selected process.

- **Print preview**: previews the graph corresponding to the currently selected process with the defined **Page Format**.

- **Zoom**: select a value from the drop down menu to zoom in or out on the graph view.

Workflow Toolbar

The Workflow Toolbar groups all Workflow 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 Block Activity: creates a set of Activities. See: "Creating and Defining Activities".
	Add Sub-Process: creates a complete Workflow Process model as an Activity. See: "Creating and Defining Activities".
	Add End Activity: creates an End Activity, which is the process end point which will terminate a Workflow instance when it is reached.
	Add Transition: adds a Transition between two Activities. See: "CREATING AND DEFINING TRANSITIONS AND ITERATIONS".
	Add Iteration: adds an iteration between two Activities. See: "CREATING AND DEFINING TRANSITIONS AND ITERATIONS".

Table 2-1.   Description of Workflow Toolbar Design Tools
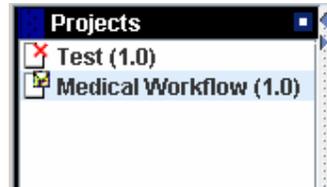
Projects View

Figure 2-10.   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
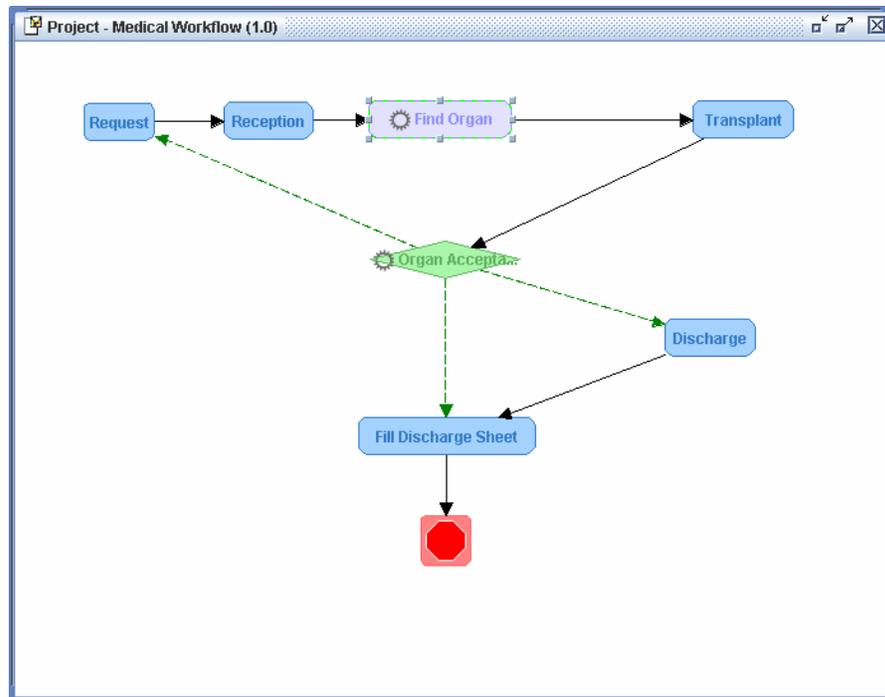
Figure 2-11.   Activity View

Symbols	Description
Basic Activity	Basic activities are shown by blue rounded rectangles - automatic or manual start and various participants
Route	Route activities are shown by green diamonds - always automatic start and the SYSTEM participant
Block Activity	Block Activities are shown by violet rounded rectangles - always automatic start and the SYSTEM participant
Subflow	Subflow activities are fat blue rectangles with a squared plus icon - always automatic start and the SYSTEM participant
(red octagon)	The end activity is shown by a red rounded square with a stop sign - always automatic start and the SYSTEM participant
(solid black arrow)	Ordinary transitions are shown by solid black arrows
(dashed green arrow)	Transitions that have a condition are shown by dashed green arrows
(curved blue arrow)	Iterations are shown by solid, curved blue arrows - an iteration may also curve back to the same activity it started from
(gear symbol)	The gear symbol indicates an activity that has the automatic start mode

Table 2-2.   Description of Activity View Graph Symbols

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

- In the Workflow 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 Workflow 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 add an iteration:

- In the Workflow toolbar, click on the iteration button

- The cursor changes to a cross and arrow.

- In the activity view, drag from the source activity to the target activity or
  Click on an activity that is both the source and target

- An iteration is added either between the two activities or looping back to the single activity

- The iteration dialog opens to enter the iteration name and other iteration properties.

- ProEd remains in the add iteration mode, indicated by a cross and arrow cursor, and additional
  iterations may be added.

To delete an activity, transition, or iteration:

If ProEd is not in the select mode, indicated by an arrow cursor, press the top button in the Workflow
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 or iteration line:

The end points of a transition or iteration 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, transition, or iteration:

- Double click on the item.
  (does not apply to block, or 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 a block or subflow activity:

- Double click on the activity.

- A new graph window opens, showing the contents of the block or subflow.

- The contents of a subflow activity may not be edited in the new graph window, but editing contents of a block activity is allowed.

- An end activity is not permitted in a block activity, but is permitted in a subflow.

Participant View

The majority of the Activity View discussion in the previous section also applies to the Participant View.
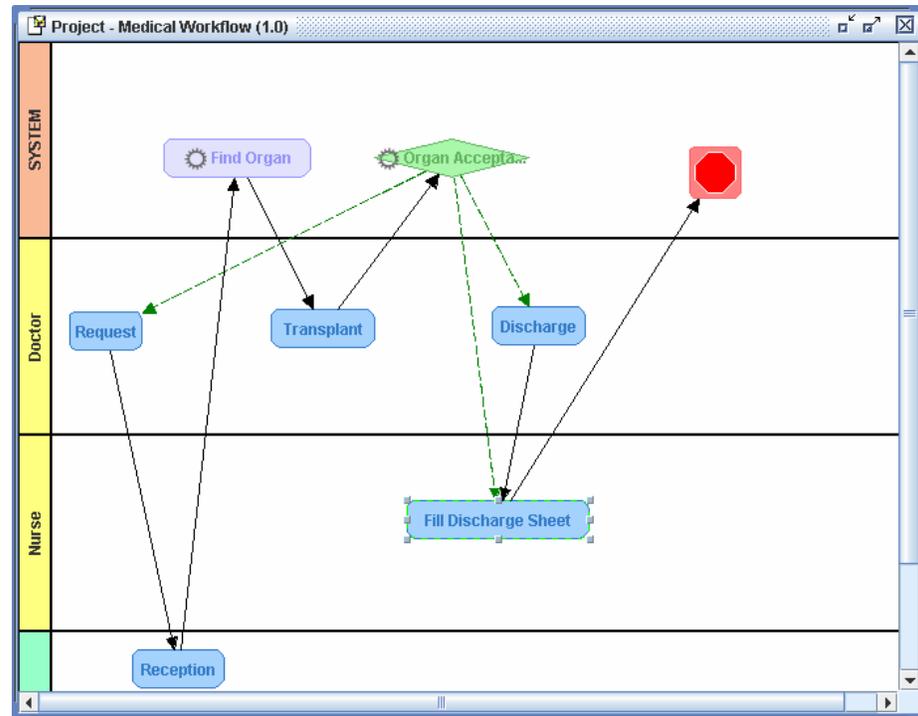
Figure 2-12.   Participant View

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.

- Block, route, subflow, and end activities must use the SYSTEM participant, so ProEd does not allow them to be dragged out of the SYSYEM 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.

2.4.3   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.
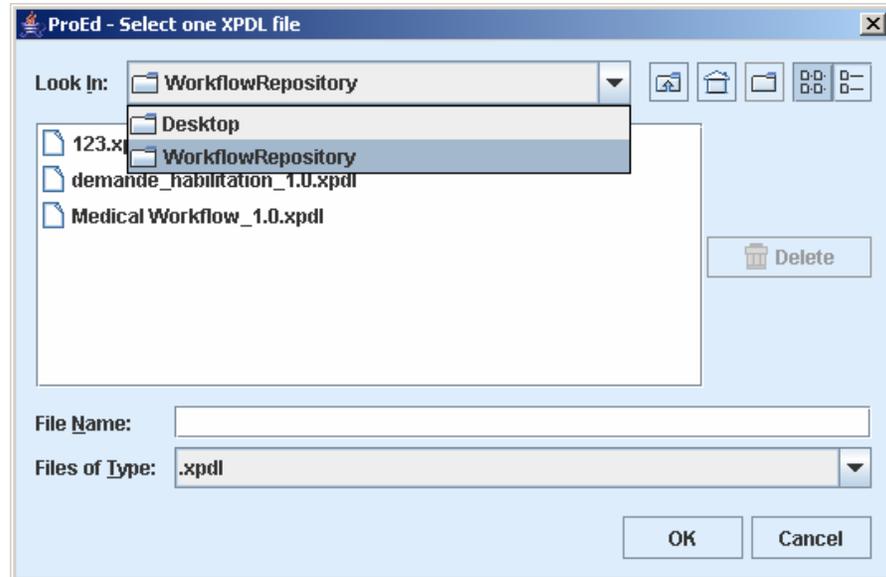
Figure 2-13.   Open File Dialog

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 "WorkflowRepository" 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 2-14.   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 Workflow 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 Workflow 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 Workflow_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.
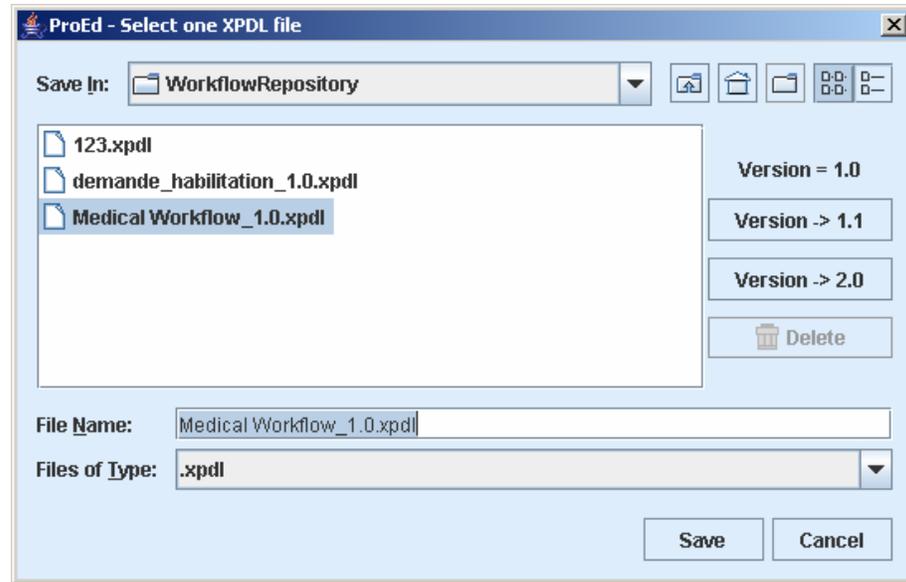
To save to an existing file:

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.

2.4.4　Defining Workflow Process Properties

A Workflow Process Model is composed of the following:

Activities, Participants, Transitions or Iterations 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 Workflow project dialog is divided into seven tabs:

- **General** tab: this tab is filled at creation time (as described in Section 2.4.1 "Creating a New Workflow Project")

- **Activity** tab: displays the list of all Activities included in this Workflow Process model. To add an activity, refer to Section 2.4.6.

- **Participants** tab: displays the list of Participants defined for the entire Workflow Process model and available for all Activities. Participants can be added, edited or deleted:

  - **Add** button: click to involve a Participant in the Workflow Process model (see Section 2.4.5).

  - **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 Workflow project. To add a Transition, see Section 0.

- **Iterations** tab: displays the list of iterations involved in the Workflow project. To add an iteration, see Section 0.

- **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 2.4.7.

  - **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 2.4.8.

  – **Edit** button: modify the selected Hook.

  – **Delete** button: deletes the selected Hook.

2.4.5   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.

Figure 2-15.   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 2-16. 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 Workflow Process model (Process level) or as the performer of an Activity (Activity level).

CHOOSING A NEW PARTICIPANT

ProEd offers the capability of addng a Participant that does not exist in the user directory but must be integrated in the Workflow 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 2-17.   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 Workflow 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 Workflow Process model (Process level) or as the performer of an Activity (Activity level).

2.4.6   Creating and Defining Activities

Five types of activities are available within ProEd:

• **Basic Activity**: the smallest unit of work in a Workflow 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.

• **Block Activity**: set of Activities grouped to simplify the creation and understanding of the Workflow process.

• **Sub-process**: complete separate Workflow Process model set as an Activity of the current Workflow Process model. This type of Activity allows simplifying the graph of the Workflow Process model, or to reuse an existing Workflow Process model.

• **End Activity**: The process end point that terminates a Workflow instance when it is reached.

To create an activity:

1. In the Workflow 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.

   - **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 2.4.5 "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 Workflow process).

– **Deadline** area: click the "**Add**" button to add a deadline:

- Condition:

  Time (relative time in days and hours): 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 Workflow process).

BLOCK ACTIVITIES

After creating a block activity, the elements grouped in it must be defined.

1. In the graph view, right-click the Block Activity to be defined and select "**Edit**": the graph window of the Block Activity appears.

2. Design the Activities in the Block Activity graph window:

   – Create and define Activities: see Section 2.4.6, "Creating and Defining Activities".
   Note that a Block Activity graph may not contain an End Activity.

   – - Create and define Transitions and/or Iterations between Activities: see Section0.

3. Close the Block Activity graph window.

4. Define the properties of the Block Activity as explained in Section 2.4.6.

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 (Block and 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 2.4.5, "Adding Participants").

- **Transitions** tab (Block and Sub-process Activities only): displays the list of all Transitions included in the Activity (see below, "Creating and Defining Transitions and Iterations").

- **Iterations** tab (Block and Sub-process Activities only): displays the list of all iterations included in the Block 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 2.4.8, "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.

- 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.

2.4.7   Creating Variables

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

Variables can be:

- **String** type

- **Static enumeration** type, where the enumeration values are set during Variable creation

- **Dynamic enumeration** type, where the enumeration values are set dynamically, by means of a Hook.

To add Process Variables to the Workflow 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.
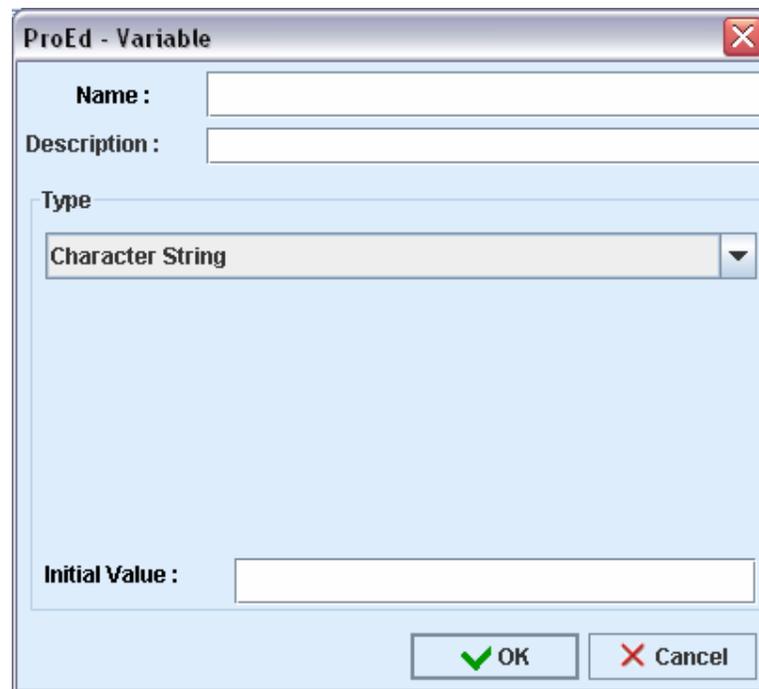
Figure 2-18.   Variable Menu

- In the "**Variable**" window fill in the following fields:

  - **Name** (mandatory field): type a unique name for the Variable ($$, ||, 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 or an integer string.

    - **Static Enumeration**: list of values. Click the Add button to add a value.

    - **Dynamic Enumeration**: list of values. Click the Add button to add a value.
      Note that the designer can still define explicit values, as an aid during development.

  - **Initial value**:

    - For **String** Variable: type an initial value for the Variable.

    - For **Enumeration**: the list contains the values defined for the enumeration. Select the desired initial value that will be presented to the user.

2.4.8   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.

Figure 2-19.   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 Workflow 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 Workflow Activities until a condition is met. Iterations require creation of at least one condition.

NOTE: Iteration is a deprecated feature that is still supported for compatibility purposes. In Bonita v4, transitions can be used to create cycles (aka iterations). Iteration graphical element will be removed from the palette in the next version.

In the Workflow toolbar, click the **Transition** ![→] or **Iteration** ![↺] button.

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).

   –   **Iterations**: the iteration properties window appears; fill in the fields as described in the following section.
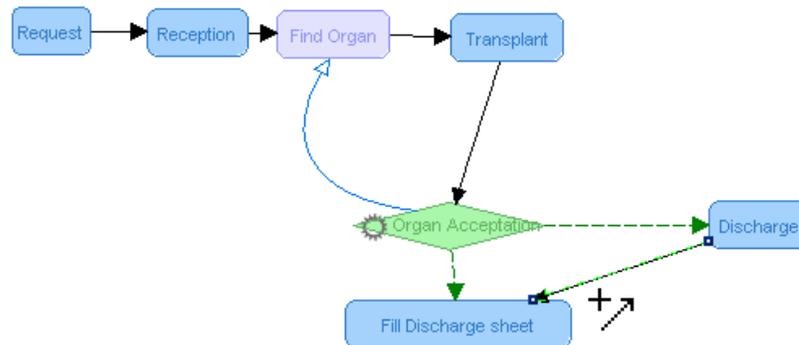
Figure 2-20.   Iterations and Transitions Graph

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.

   –   **Iterations**: Iterations arrows are curved and already have a vector point; click and drag this point to modify the shape of the arrow.

ADDING / EDITING CONDITIONS ON TRANSITIONS AND ITERATIONS

1. Right-click on a transition or an iteration and select "**Add/Update condition**". The "**Add condition**" window appears:
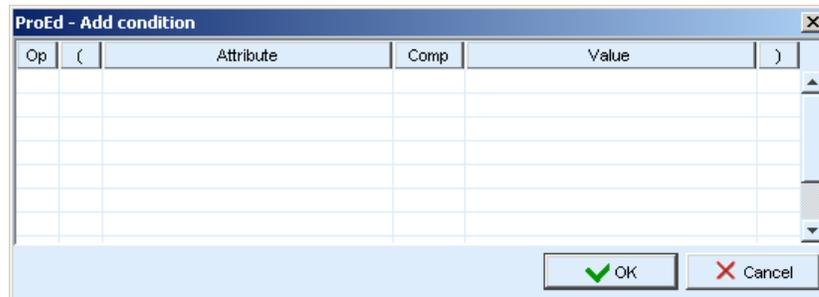
Figure 2-21.   Add Condition Window

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 / Iteration.
Variable	Select the Variable from the drop down list.
Comp (Comparator)	Select an operator: **=** "is equal to" **!=** "is different from" (not equal)
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 2-3.   Description of Condition Parameters

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

MODIFYING THE PROPERTIES OF A TRANSITION OR AN ITERATION

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

Figure 2-22.   Modifying Transition or Iteration 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 AND ITERATIONS"

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 Workflow 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 Workflow is installed, the above steps must be repeated.

Chapter 3.   Hooks

3.1   Introduction to Hooks

Hooks in Nova Bonita Workflow context are external java classes performing user-defined operations. At different moments in Workflow process execution, hooks might be called by the workflow 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 3-1.   Hooks Names

Workflow 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 Workflow 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 Workflow 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.

3.1.1   Hooks Execution Time Scale

In a « client-driven » Workflow context, meaning a java program using the Workflow engine API, Hooks are leveraged to link processes with already existing applications. Hook calls/executions are performed by the workflow engine itself, meaning that Hooks are always executed at workflow 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 workflow engine is embedded in the client application (i.e Tomcat based application).

As said, hooks are automatically triggered by the workflow engine based on activities types (known as activity "bodies" in Nova Bonita) life cycle. Tasks (aka manual activities) follows the life cycle below:
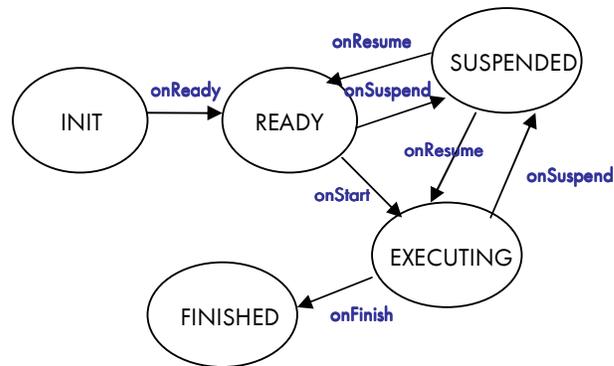
In Figure 3-1, possible hooks types are related to the Nova Bonita task life cycle (activities with body = task).

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

3.1.2 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 Workflow 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 ».

3.2 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 »: « Workflow related», and what is called a « java-related environment».

3.2.1 Workflow-Related Hook Actions

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

- get an activivty / process variable
  - methods: getVariable

- set an activity/process variabke
  - 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).

3.2.2 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)

3.3 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.

3.3.1 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 workflow related operations)

The Workflow 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 Workflow 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.

3.3.2   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 Workflow engine therefore keeps transactional consistency across activities.

The Workflow engine aborts a transaction in two cases:

- A failure at system level (e.g. impossibility to access the Workflow 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.

3.4   Writing a Hook

Process and activity hooks are a java class and have to implement either Node or TxNode 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 workflow 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 workflow instance.

3.5   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 Workflow, (the hook caller), is to send a Java Exception. The exception constructor takes a parameter string and Workflow, 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.

3.6 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.

3.7 Use Case

3.7.1 A Simple Hook

For this example, "Hello world" will be printed in the Workflow console

The code produced is the following:

```java
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.

3.7.2 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 Workflow process under the activity variable "Email_address".

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

```java
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;
```

```java
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");
  }


}
```

3.8   Practical Steps for Hooks Usage

3.8.1   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.

3.8.2   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 assigments.

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 workflow process deployed in Nova Bonita.

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

- deployBar: operation deploying a workflow 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

Chapter 4.   Mappers

4.1   Introduction

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.

4.2   Writing a Mapper

A **mapper** specifies person(s) corresponding to a specific role defined in the Workflow 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 (I*nstance 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 workflow 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 workflow 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 workflow instance execution.

4.2.1 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 Workflow 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 Workflow 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 Workflw 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 workflow process, developers can easily check that the workflow execution is working as expected.

4.2.2 Practical Steps for Using Custom Mappers

Mappers – loading and compiling

The Workflow 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 assigments.

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 workflow process deployed in Nova Bonita.

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

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

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

- removeClass: remove a mapper fro the production environment

- replaceClass: replace an already deployed mapper from the production environment

4.2.3   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.

```java
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;
  }
}
```

Chapter 5.   Performer Assignment

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

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

5.1   Introduction

This feature allows assigning additional assignment rules other than those in the standard Workflow 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.

5.2 Performer Assignment Types: Custom and Properties

5.2.1 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*):

5.2.2 Variables Performer Assignment

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

5.3   Practical Steps for Using Callback Performer Assignments

5.3.1   Performer Assignment – Loading and Compiling

As mappers, custom performer assignment are loaded and executed by the Workflow engine. If adding a specific custom classs, 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 workflow process deployed in Nova Bonita.

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

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

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

- removeClass: remove a PA from the production environment

- replaceClass: replace an already deployed PA from the production environment

5.3.2 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.

```java
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";
  }

}
```