OpenBP User Guide

Version: 0.1

Product: OpenBP
Product - Version: 0.9.8

OpenBP User Guide

Document Version: 0.1

Publication date 05.03.2010 10:58:45.

Main Product: OpenBP

Version: 0.9.8

Revision History				
Rev Number	SW Number	Date	Author	Remark
Revision 0.1		01.10.2009	heiko	created

Table of Contents

Overview

1. Field of Application & USPs

1.1. Business Application Challenges

Programming complex business applications in a regular programming language such as Java has a few challenges regarding process execution:

- *State Management*

  When your program needs to wait on a particular event (e. g. a user entering some data, some external system responding to a request or some computation to be completed), it will block memory and CPU resources while waiting. Also, when somebody turns off the computer or the program crashes, the state of the program will get lost resulting in an uncompleted process.

  In order to get this problem resolved, you need to use a persistence mechanism (i. e. a database) to save the state of your program to and you need to implement the logic that suspends and resumes your program upon completion of the pending task. Usually, this complex logic becomes rather application-specific, requiring additional maintenance and extensive testing effort.

- *Business Functionality Definition*

  Java being an architecture-orientated language does not provide functionality that is really suitable for business specialists to describe business functionality. The business functionality (i. e. what - from a business point of view - needs to be done with the objects that are managed or processed by the program) often gets lost somewhere between frameworks, architecture patterns and technical configuration. Sometimes even smaller functional changes (e. g. adding an additional workflow step such as a confirmation by a manager) can proof a complex task.

1.2. Business Process, Workflow & Co.

According to different requirements, processes within software applications can be grouped in three categories:

- *Business Processes*

  A business process is a rather high-level process that defines how distributed systems interact in a heterogeneous environment. Business processes can span across companies, systems and programming platforms and languages. The focus of business process management is on enterprise application integration and safety. Business process engines need to integrate with middleware software and need to follow cross-platform standards.

  An example of a business process may be an order process that involves an ERP system, a CRM system and various external partners providing fulfillment and delivery services in a SOA environment.

- *Workflow Process*

  A workflow process defines the interaction of the different roles within an organization in order to perform a business task. Though roles can be fulfilled by persons as well as systems, workflows have an organizational rather than a technical focus.

  A typical workflow may determine how a vacation application needs to be processed when running through a department.

- *Application Process*

An application process is a process that defines how a particular business functionality within an application is to be performed.

An example might be the steps that need to be performed by a production machine in order to manufacture an item.

Though also in this type of process interaction with external systems or roles may be required, the focus is on the business functionality within the application, requiring an emphasis on performance and integration with the Java environment.

*Process engines*are frameworks that take a process definition in some formal language (usually XML-based) and execute it step by on a set of data (the process context). The process definition is either written manually or in some cases generated by a graphical process modeling tool.

1.3. What is OpenBP?

OpenBP is an open Java-based process and workflow platform. The platform consists of a sophisticated graphical process modelling tool (the modeler) and a process execution framework (the engine). The engine executes the XML-based business process definitions that are created and edited in the modeller.

The OpenBP modeler allows the developer or business process specialist to visually model business processes and workflows as flowchart or UML activity diagrams using simple drag & drop. The OpenBP Modeler is intended to be used by software developers as well as process specialists and consultants.

1.4. What's Different To Other Process Engines?

OpenBP provides the following USPs as compared to other process engines:

- *Lightweight application processes*

  OpenBP is focused on application processes. Its goal is to bring pragmatic and performant process support to any type of application, be it administrative, technical or even real-time.

  Pragmatic, fast and flexible approach for application processes

- *Sophisticated graphical process modeling*

  A sophisticated modeling tool enables comfortable process design, supporting all features of the modeling language. There is *no need for manual editing of the process XML.*

  Excellent tool support for business specialists as well as developers.

- *Extremely fast suspend & resume*

  As opposed to some other process engines that have a pretty complex database footprint, OpenBP processes can be suspended to the database or resumed extremely fast. In addition, the interaction of OpenBP with the database is always transparent. There is no indeterminable 'magic' behind the scenes.

  Support for high-performance process execution

- *Process debugger*

  In OpenBP, the execution of a process can be traced in the graphical modeling environment. Process debugging does not require debugging of the process engine on the Java level. Even the process context data can be viewed in the modeling tool, making process development a snap. In some deployment scenarios, process definitions can be updated in the modeling tool without even having to restart the running application.

Process-level debugging aids developer productivity considerably

- *Hot Deployment*

  OpenBP supports hot deployment, which eliminates the 'Build' and 'Deploy' step in your Development - Build - Deploy - Test cycle for the logic realized as OpenBP process. Any change in a process can be reflected instantaneous to the server that runs the OpenBP engine - in most cases even while debugging the process.

  Hot deployment saves a lot of time by eliminating the 'Build' and 'Deploy' steps for process logic

- *Layered Modeling*

  OpenBP supports different levels of interconnected models. For complex scenarios, you may start modeling an abstract business process (that may represent software-based elements as well as other activities) and drill down to the level of the technical process that will finally be executed by the process engine. Simply double-click a higher-level process element and the lower-level process will open.

  Layered modeling may help business and IT specialists communicate better

- *Use case flexibility*

  OpenBP provides a flexible API that allows using the framework for different use cases such as regular process execution, dynamic construction of GUI pages, definition of object lifecycles (states and transitions) etc.

  All-purpose process engine

OpenBP also provides integration and workflow features as far as needed for application processes, but this is not the primary focus of the framework.

2. Basic Concepts

2.1. Process Modeling and Process Execution

Similar to conventional software development, we differ between development and runtime.

During development, you create a process model that defines how your business processes should be executed. The visual development environment, the OpenBP Modeler, will aid you in this process. The result is a series of XML descriptions defining your process model. They are stored in a model repository, which might be located in a file system directory, stored in a database or contained within a jar file of the classpath.

At runtime, these XML files will be read from the repository by the process engine. On request, the process engine will execute the process model step by step.

2.2. The Process Lifecycle

TODO

2.3. Model Concept

OpenBP processes and supporting components are clustered in *models*. A model can be seen as container that holds objects that are semantically related.

For example, an application might contain a model `ErpExchange` that hosts all processes that deal with interfacing with an ERP system.

A model may refer other models (*model import*), defining a hierarchy of models. Similar to Java packages, this allows for structuring an application. It also supports specialization of a model (similar to Java subclassing) by overriding a component of an included model in the including model.

Models are stored in *model repositories*.

2.4. Model Components

Component	**Description**
Process	An OpenBP process that can be executed by the process engine.
Activity	A template representing an activity node (usually refering to or containing a piece of code) that can be used within a process.
Visual	A template representing a user interaction node that can be used within a process.
Data Type	The (optional) definition of a data type that may be used as process data.

3. Getting Started

3.1. Setting Up the OpenBP Distribution

TODO

TODO: Copy OpenBP-user.properties to HOME dir

3.2. Installing the OpenBP Distribution

TODO

4. The OpenBP Modeler

4.1. Starting The Modeler

In order to start the modeler, choose the `OpenBP Modeler` menu item in the `OpenBP` program group.

4.2. Initial Modeler Configuration

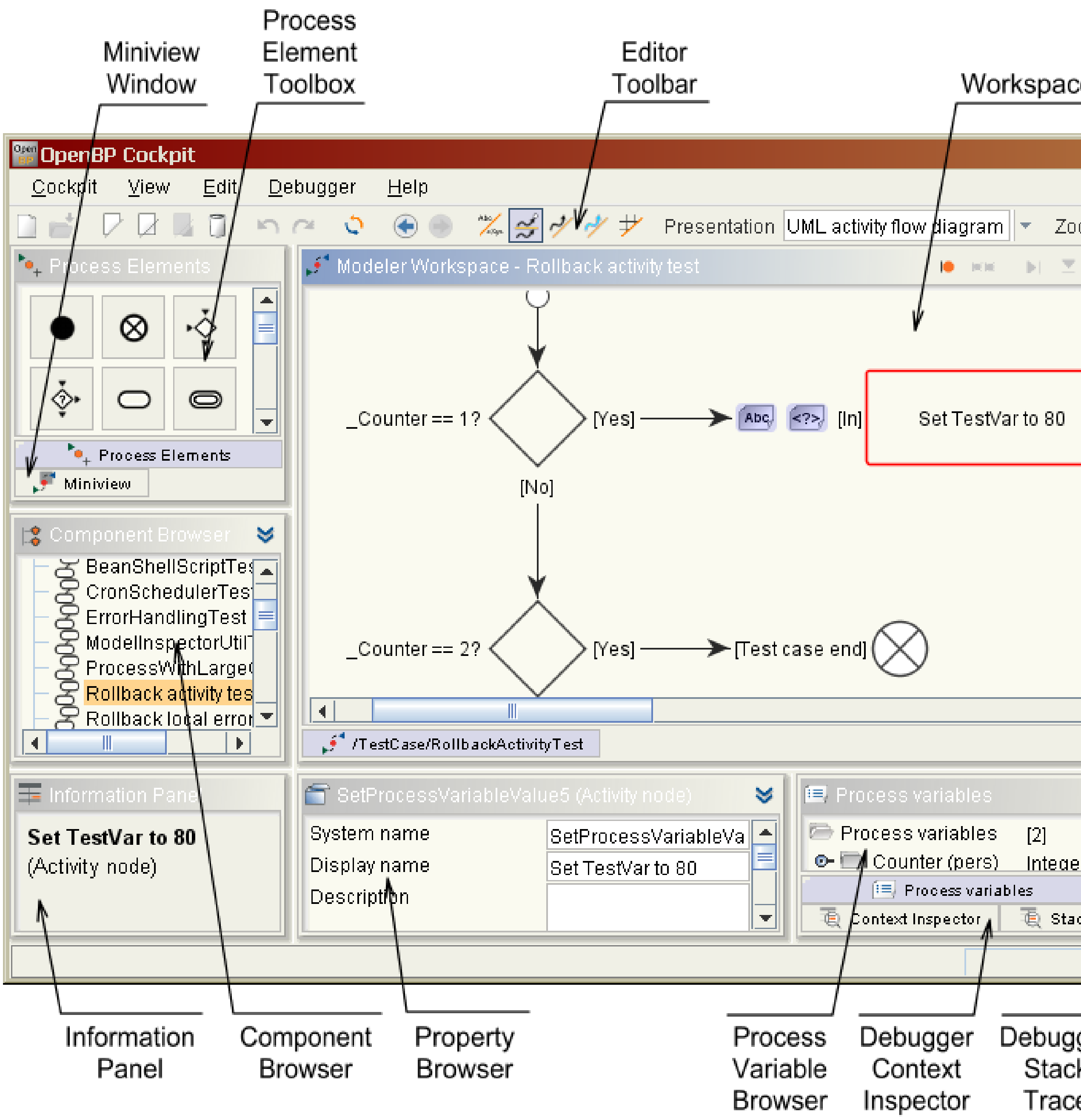
Select Cockpit|Options|Associations from the main menu.

This tab page allows for specifying programs that will be used to display certain types of documents. We recommend specifying viewers for the following document types:

- text/plain: Generic text documents
- text/x-source: Java source code
- text/html: HTML documents (e. g. Javadoc)
- application/pdf: PDF documents (e. g. manuals)

4.3. The Modeler Windows

The modeler is divided into several windows:



- The *Workspace* shows the process that is currently being editred. The modeler allows to edit multiple processes at once. Each process is displayed in its own tab page. You can switch between the tab pages by clicking on one of the tab titles below the workspace window.

- The *Editor Toolbar* provides the most important editor functions and allows for adjusting the zoom level and selection the presentation skin. Note that you may also change the zoom level by **CTRL +Mouse Wheel** or **CTRL+ALT+Mouse Movement**.

- The *Workspace* displays a particular portion of the current process.

- The *Miniview* window displays the entire process, providing an overview. The portion displayed by the workspace appears as light-blue rectangle.

You may click into the miniview to position the workspace view or you may drag a rectangle in the miniview to redefine the portion to be displayed by the workspace.

In the Cockpit|Options|Modeler dialog, you may choose how many processes are being displayed simultaneously in the miniview window.

- The *Process Element Toolbox* displays the standard components you may add to the process. Drag a component symbol from the component window to the workspace in order to create a new process node.

- The *Component Browser* lists all components add to the process. Drag a component symbol from the component window to the workspace in order to create a new process node.

- The *Information Panel* displays descriptive text about the currently selected element. For example, the information panel displays the display name and description of process elements in the workspace as well as components in the component browsers and also an help text for the currently selected field in the property browser.

- The *Property Browser* displays the properties of the currently selected element (i. e. an element in the workspace or a component in the component browser). The windows allows also for editing the component's properties.

- The *Process Variable Browser.* lists all process variables of the current process.

  You can create new process variables by dragging a data type of the System unit from the component browser to the process variable browser. Double-click an entry in order to view or edit its properties.

- The *Debugger Context Inspector* shows the current data context of a running process. Double-click an entry of the list to view any subordinate entries. Note that the inspector does not support changing the values of a running process.

- The *Debugger Stack Trace Window* shows the 'Stack Trace' of a process that has been called as sub process.

- The *Color Palette* allows assigning custom colors to several process elements such as activity nodes or sub process nodes.

  Simply choose a color in the color picker and drag the color from the preview field to the left of the dialog onto the component.

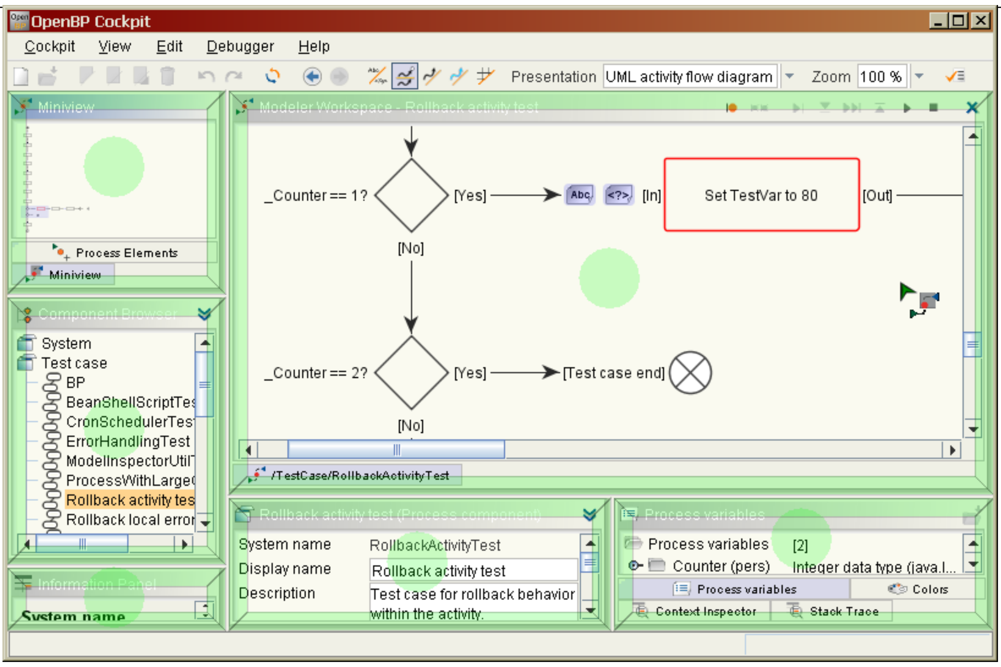
4.4. Customizing the Workspace

You can customize the modeler by changing window sizes or moving windows to different locations.

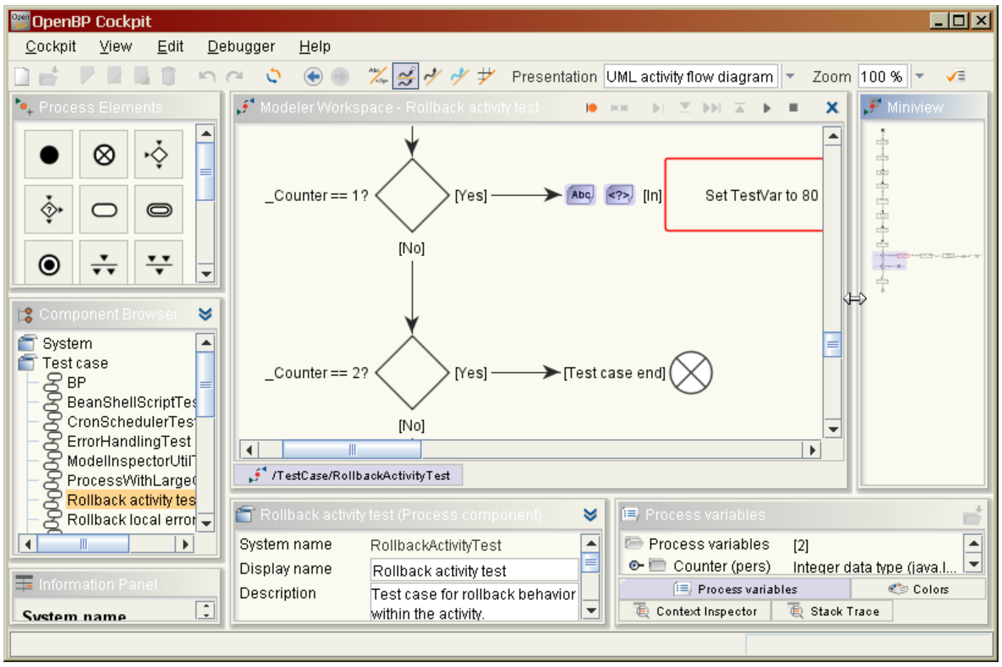
For example, suppose you might want to have the miniview in a separate window next to the workspace instead of having it hidden behind the process element toolbox.

First, bring the miniview to the fron by clicking at the tab page labelled 'Miniview'. Then move the cursor over the title bar that displays 'Miniview' and start dragging the title bar. You will notice light green rectangle sections and circles popping up. In order to dock the miniview as a separate window next to another window, drop it at the corresponding rectangle section. Drop it at a circle to add the view as a tab page to window beneath the circle.

For practice, drop the miniview on the rectangular section to the right of the workspace.



The miniview will be docked to the right of the workspace. You may reduce the width of the new miniview window in order to have more space available for the workspace window by simply dragging the edge of the window. Now, the editor should look like this:



Note that these settings are not permanent. So if you accidentally might have messed up yor workspace configuration, simply restart the OpenBP cockpit.

4.5. Modeler Keyboard Shortcuts

This section provides information about the keyboard shortcuts within the OpenBP Modeler that can help to make editing a lot quicker and easier.

4.5.1. Window Activation Shortcuts

Using one of the following shortcuts, you can activate a particualar window, giving the focus to it and make it come to the front if is hidden by another tab page.

Window	Keyboard Shortcut
Component Browser	F2 | CTRL+w b
Modeler Workspace	ALT+F2 | CTRL+w m
Options...	CTRL+F2 | CTRL+w o
Miniview	F3 | CTRL+w v
Process Elements	SHIFT+F3 | CTRL+w s
Toolbox	CTRL+w t
Process variables	CTRL+F3 | CTRL+w g
Context Inspector	ALT+F3 | CTRL+w d
Stack Trace	CTRL+w k
Colors	CTRL+w l
Property Browser	CTRL+w e

4.5.2. List Manipulation Shortcuts

You may use the following shortcuts to edit lists in browser windows, e. g. the comonent browser, the property browser or the process variable browser.

Function	Keyboard Shortcut	Description
Add Element	CTRL+n	Adds a new element to the end of the list
Copy Element	CTRL+c	Copies the selected element
Cut Element	CTRL+x	Cuts the selected element
Paste Element	CTRL+v	Pastes a copied element
Remove Element	DELETE	Removes the selected element (w/o confirmation!)
Move Element Up	CTRL+u	Moves the element upward in the list
Move Element Down	CTRL+d	Moves the element downward in the list

4.5.3. Component Browser Shortcuts

In the component browser, you may use the folloiwng special shortcuts.

Function	Keyboard Shortcut	Description
Functional Groups	CTRL+g	Show or hide the functional groups of the components
New...	CTRL+n	Create a new component or model
Remove	DELETE	Remove the selected component or model after confirmation

4.5.4. Workspace Shortcuts

These shortcuts are very useful in the modeler workspace.

Function	Keyboard Shortcut	Description
Save	CTRL+s	Save the edited process(es)

Function	Keyboard Shortcut	Description
Select All	CTRL+a	Select all objects
Normalize Diagram	CTRL+m n	Normalize the origin of the diagram
Flip orientation	CTRL+f	Flip the orientation of a data link. The link will be locked automatically to the new direction.
Control Anchors On/Off	CTRL+m a	Toggle the display of control link anchors
Control Links On/Off	CTRL+m c	Toggle the display of control links
Data Links On/Off	CTRL+m d	Toggle the display of data links
ModelerPage	CTRL+m g	Grid On/Off
Process Back	ALT+left	Goes backward in the process history
Process Forward	ALT+right	Goes forward in the process history
Undo	CTRL+z	Undo the last action
Redo	CTRL+y	Redo the last undo action
Server Reload	CTRL+r	Causes the server to reload any component classes and mapping files. Use this function after rebuilding the project in the Java IDE.

4.5.5. Debugger Shortcuts

These shortcuts provide convenient debugger control when stepping through a process.

Function	Keyboard Shortcut	Description
Toggle Breakpoints	F9	Toggle the breakpoints of the selected objects
Clear all Breakpoints	SHIFT+F9	Clear all breakpoints
Step Next	F5	Step to the next event
Step Into	SHIFT+F5	Step into a component
Step Over	F6	Step over a component
Step Out	F7	Step out of the current process
Continue	F8	Connects to a running intance of the OpenBP server or resumes a halted process
Stop	SHIFT+F8	Terminate a halted process

Tutorial

1. Tutorial Story

The concepts of OpenBP are best explained by building a small tutorial application, that makes use of the OpenBP process engine.

As an example that should be common to everybody, the tutorial will implement a simple business processes for managing vacation requests. Use cases for this process might be vacation or business travel planning. This type of processing will require some as well as workflow functionality.

The business process defines the business logic that the application provides:

- Create a task list item for the supervisor of the submitter
- Process the decision of the supervisor
- If accepted, submit a request to an external system that might adjust the capacity planning accordingly
- Create a task list item for the submitter stating the final result of the process.

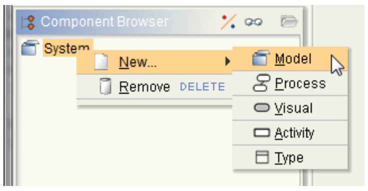
2. Initial Setup

2.1. Creating a New Model

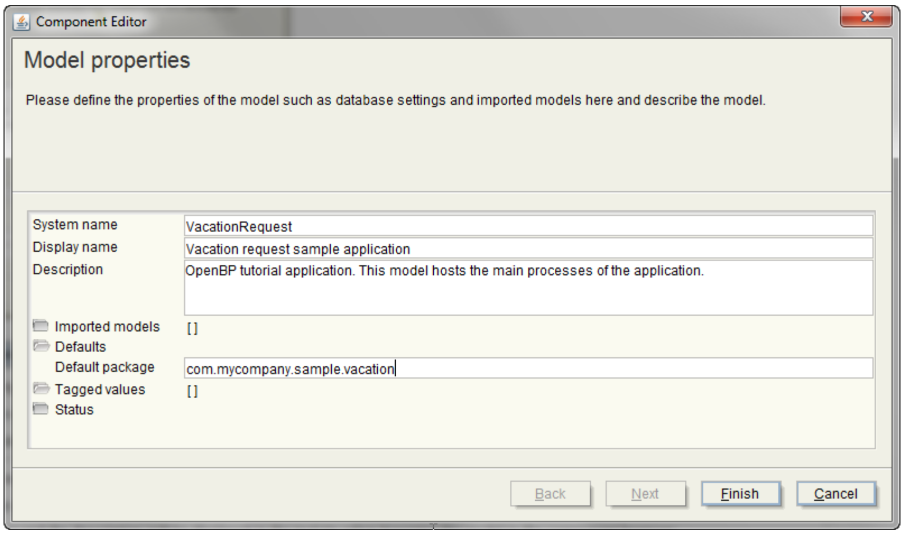
All OpenBP components need to be placed in a container named model. For now, consider a model some kind of domain wrapper for process functionality.

Follow these steps in order to create your model:

1. Select Cockpit|New|Model from the main menu or in the component browser right click the System model and select New|Model.



2. Define the properties of the new model as follows:.



Model properties

Please define the properties of the model such as database settings and imported models here and describe the model.

System name	VacationRequest
Display name	Vacation request sample application
Description	OpenBP tutorial application. This model hosts the main processes of the application.
Imported models	[]
Defaults	
Default package	com.mycompany.sample.vacation
Tagged values	[]
Status	

Property	Value/Description
System name	`VacationRequest` Technical name of the model (must be unique throughout all models). Though not necessary for models, for consistency with other objects we suggest to use the same syntax as Java class names for system names.
Display name	`Vacation request sample application` Human-readable name of the model. The display name and the description will be displayed in the tool tip when hovering the model in the component browser.
Descripition	`OpenBP tutorial application. This model hosts the main processes of the application.` Short descriptive text about the purpose of the model. You may generate a newline using the **CTRL +ENTER** key.
Defaults-Default package	`com.mycompany.sample.vacation` Name of the Java package that shall be associated with this model. Useful when creating source code for Java activity nodes.

3. Click Finish to create the unit. The new model will appear in the component browser.

3. Business Process Modeling

This sample is a small process that can be modeled on a technical level right away. However, we want to demonstrate OpenBP's layered modeling feature that can help you to develop a process starting from a bird's eye view down to the technical modeling level.

3.1. Creating the Business Process

Our business process will define the principal tasks of the application that we want to develop. A business process process is not to be executed by the process engine, it merely serves as the 'Big Picture' that structures lower level processes and non-IT-based activities.

1. Right click the new model in the component browser and select New|Process.

   Tip

   If you prefer using the keyboard, you may pop up the menu in the component browser by using the **SPACE** bar.

2. Choose which type of process you wish to create. If you select a type, a description of the type appears in the headline of the dialog.

Type	**/Description**
Business process	High level business process. Meant for modeling processes that may contain non-executable portions.
Top level process	Executable process that can be invoked by the application code. This is the most common type of process you may use within OpenBP models.
Sub process	Process that can be called by a top level process or another sub process. A sub process is used in cases where a particular functionality is needed in several top level processes or several locations within one top level process or if you want to enable overriding of a particular portion of a process.

For now, choose `Business process` and click Next.

3. Define the properties of the new process as follows:.

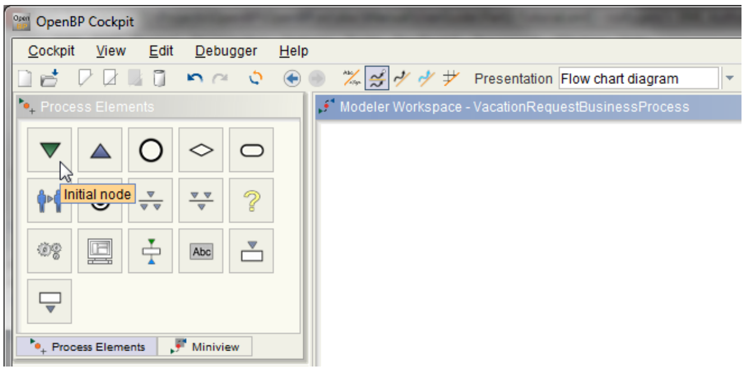
Property	**Value/Description**
System name	`VacationRequestBusinessProcess`
Descripition	`Abstract outline of the functionality of the application.`
Functional group	`Business process` You may use arbitrary functional group to sub-structure the components of you model semantically. For example, we have decided to structure our processes into 'Business process' and 'Technical process'.

A business process does not have parameters so, click Finish righ away. An empty workspace appears.

3.2. Initial Node

A process always starts with an initial node.

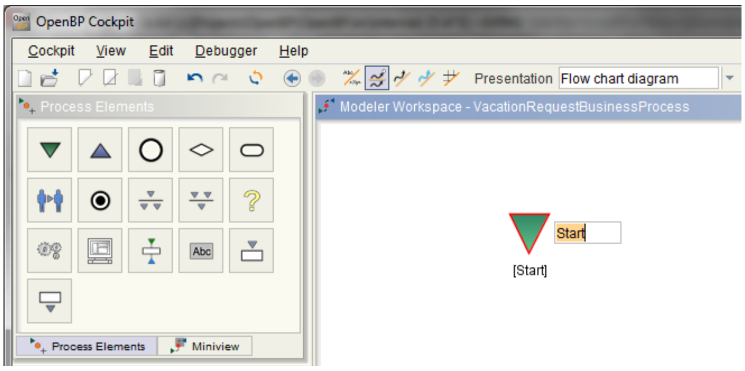
1. In order to create an initial node, simply point at the green triangle in the standards component browser (the top left view) and drag it the onto the workspace by pressing the left mouse button and releasing it over the workspace. While you drag, the cursor changes to reflect the type of item that is being dragged.


OpenBP Cockpit
Cockpit View Edit Debugger Help
Presentation Flow chart diagram
Process Elements
Modeler Workspace - VacationRequestBusinessProcess
Initial node
Abc
Process Elements
Miniview

Drop the node somewhere at the top of the workspace.

2. A small input widget (the 'in-place-editor') is displayed to the right of the new process element. Use it to change the system name of the node to `Start`.

   The result might look like this:


OpenBP Cockpit
Cockpit View Edit Debugger Help
Presentation Flow chart diagram
Process Elements
Modeler Workspace - VacationRequestBusinessProcess
Abc
Start
[Start]
Process Elements
Miniview

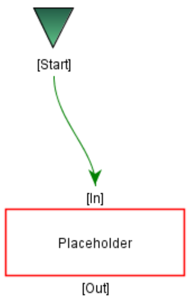
Note

To bring up the in-place-editor again, simply click the currently selected node once more. You may also use the property browser window to modify the properties of the currently selected element.

3.3. Functionality Nodes

On this level, our goal is to structure the process we would like to realize. We do not (yet) care about the details. So we use abstract element - placeholder notes - to define particular process steps.

1. Drag and drop the yellow question mark from the standards component browser onto the workspace below the initial node.



2. This process step shall create a task list entry for the supervisor of the submitter, who shall decide about the request. In the property browser change the system name of the placeholder to `Approve request.`

3. The connectors where control links between nodes can be attached are called sockets. A node has a single entry socket (the socket named `In` by default) and an arbitrary number of exit sockets. By default, a placeholder node has a single exit socket.

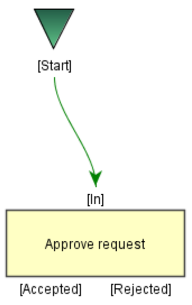
   The vacation request may be either approved or rejected. We will reflect these options by two different sockets.

   Select the only exit socket of the placeholder node (the lower one) and change its name in the properties browser to `Accepted`.

   Press the CTRL key to activate the socket movement mode and drag the exit socket a little to the left in the workspace.

   Drag the exit socket symbol from the standard components window to the bottom area of the placeholder and rename it to `Rejected`. You may use CTRL+Mouse to rotate the socket around the node. Place it to the right of the `Accepted` socket .

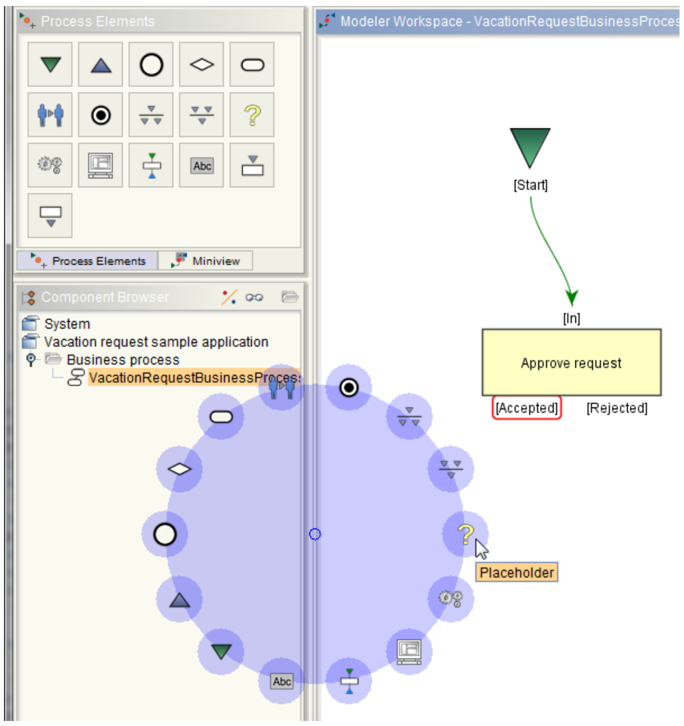
   Now, your process should look like this:

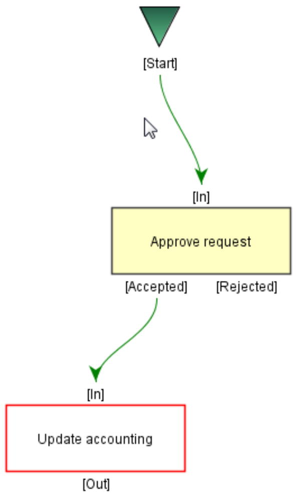
4. Now let's wire the `Accepted` socket with a new placeholder node in order to model the processing that should be performed if the supervisor of the requester agrees to the request. However, we will use a handy shortcut for inserting nodes.

   Select the `Accepted` socket and move the mouse somewhat below where you want the new placeholder node to be placed. Now press the **SPACE** bar. The flywheel menu will appear. It displays all elements from the standard component window that can be inserted at the present location. You can rotate the flywheel with the mouse wheel until the *placeholder* symbol appears under the cursor. Now simply click to insert the element.

   Select the `Accepted` socket and move the mouse somewhat below where The flywheel menu will remember the recently inserted element and will suggest this element by default the next time you open the flywheel menu. This allows for quickly inserting similar element types.

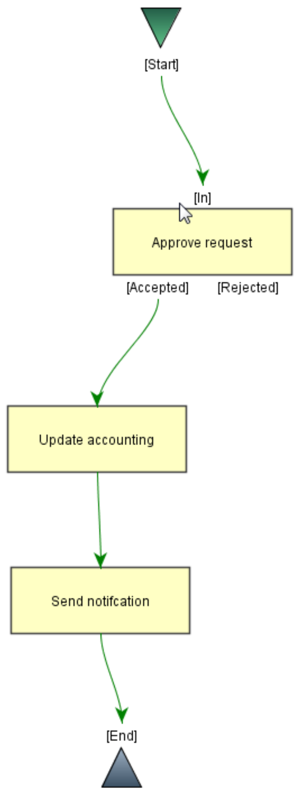


5. After you have insert the element, change its name to `Update accounting`. The task of the element will be to communicate the vacation request to the accounting system of the company.



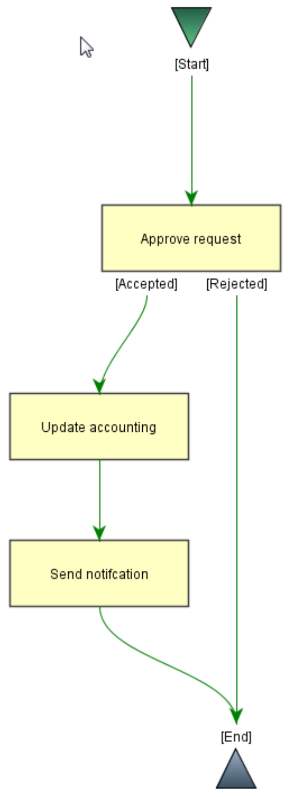
6. The next node should be a placeholder node named `Send notification` that resembles sending a notification to the requester to inform him about the outcome.

   Finally, insert a final node to distinguish the end of the workflow and name it `End`.

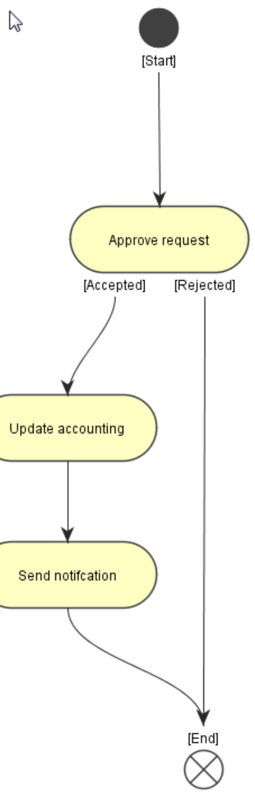


7. We also need to notify the requester if the request was declined. So select the `Rejected` socket and manually draw a control link to the `In` socket of the `Send notification` node.

   That's it so far for our simple business process. You might align the nodes a little to beautify the process. Finally press **CTRL+s** to save the process.



8. By default, the business process is displayed using a flowchart notation. However, you may switch to UML notation at any time by by selecting the appropriate presentation from the toolbar of the modeler.



4. Technical Process Modeling

4.1. Technical Concept

The vacation request application might run as part of a corporate intranet, which contains a page where employees may enter data for a planned business trip or a vacation they want to apply for.

The `Submit` button of the web page will transfer the data entered in the page to a request handler servlet. The servlet will construct an object that holds the form data and populate it with the data entered by the user. The object's class may look like this:

```
     package com.mycompany.sample.vacation;

import java.sql.Date;

public class VacationData
{
 public static final int STATE_NEW = 1;
 public static final int STATE_ACCEPTED = 2;
 public static final int STATE_REJECTED = 3;

 private String submitterName;
 private String submitterEmail;
 private int state;
 private String reason;
 private Date fromDate;
 private Date toDate;

 public String getSubmitterName()
 {
  return submitterName;
 }
 public void setSubmitterName(String submitterName)
 {
  this.submitterName = submitterName;
 }

 public String getSubmitterEmail()
 {
  return submitterEmail;
 }
 public void setSubmitterEmail(String submitterEmail)
 {
  this.submitterEmail = submitterEmail;
 }

 public int getState()
 {
  return state;
 }
 public void setState(int state)
 {
  this.state = state;
 }

 public String getReason()
 {
  return reason;
 }
 public void setReason(String reason)
 {
  this.reason = reason;
 }

 public Date getFromDate()
 {
  return fromDate;
 }
 public void setFromDate(Date fromDate)
 {
  this.fromDate = fromDate;
 }

 public Date getToDate()
 {
  return toDate;
 }
 public void setToDate(Date toDate)
 {
  this.toDate = toDate;
 }
}
```

The servlet will then create a new instance of an OpenBP process, provide the data object as parameter to it and pass the process instance to the OpenBP engine for execution. The engine will asynchronously execute the processing instructions defined by the business process one by one. Since this may take a lot of time, especially if the process contains interruptions such as workflow activities, the servlet will return immediately, notifying the user that his request will be processed.

4.2. Data Type Definitions

It might be useful to define a special data type that we want to use within the process. subordinate functionality.

Note

As an alternative to custom type definitions, you might simply use the OpenBP system type 'Object' for parameter definitions in your process, which will work just as well. In this case, you do not need to define custom data types.

Though this will probably save a lot of time, OpenBP cannot provide popup menus for data members of the particular type at design time and will also not be able to perform type checking at runtime. It will also require more discipline in terms of parameter naming so that the parameter type get obvious from its name.

This is a matter of personal preferences. In favor of clarity, we will use custom type definitions.

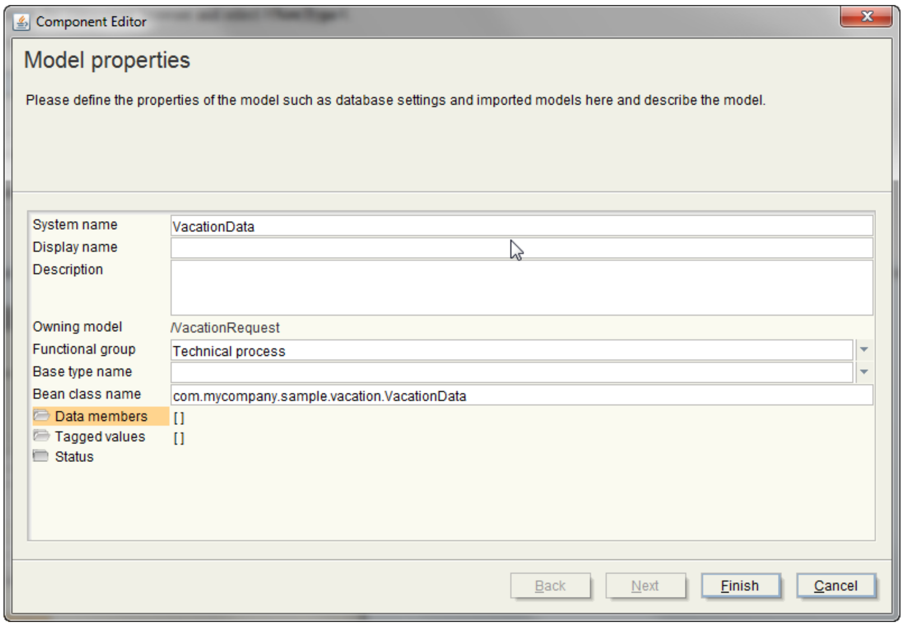
1. Right click the `VacationRequest` model in the component browser and select New|Type.

2. Enter the following data to specify the new type:

Property	**Value/Description**
System name	`VacationData`
Functional group	`Technical process`
Bean class name	`com.mycompany.sample.vacation.VacationData`

   Click `Finish`.

3. Now we will define the data members of our custom type. For each data member, right click the `Data members` label item in the component browser and select Add Element or press **CTRL +n** to add a new data member to the end of the member list.

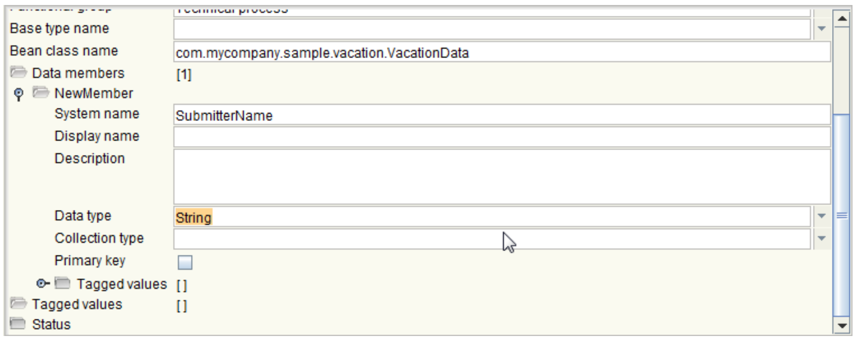


Model properties

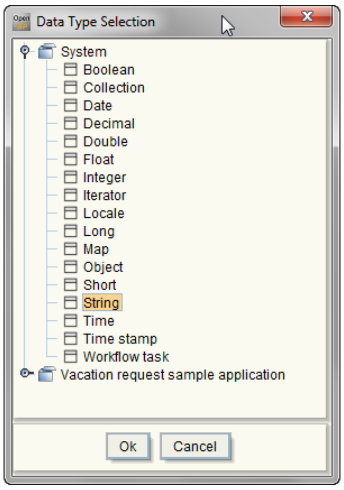
Please define the properties of the model such as database settings and imported models here and describe the model.

System name	VacationData
Display name	
Description	
Owning model	/VacationRequest
Functional group	Technical process
Base type name	
Bean class name	com.mycompany.sample.vacation.VacationData
Data members	[]
Tagged values	[]
Status	

For each data member, specify its properties in the property browser. The data type of a member itself must be referred to OpenBP type also. In order to open the popup dialog for type selection, click the arrow button of the `Data type` field or press **ALT+DOWN** when the field has the focus.

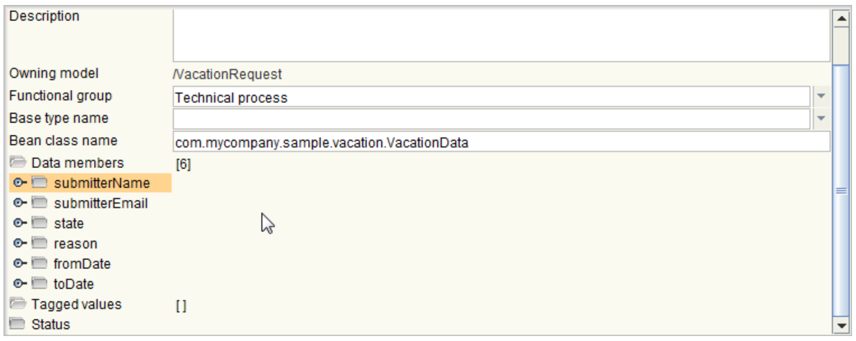


Base type name	
Bean class name	com.mycompany.sample.vacation.VacationData
Data members	[1]
NewMember	
System name	SubmitterName
Display name	
Description	
Data type	String
Collection type	
Primary key	☐
Tagged values	[]
Tagged values	[]
Status	



To create the next data member, simply press **CTRL+n**. When you are finally done, click `Finish`.

4. The new type appears in the component browser. You may double-click the type once more to open it in the component editor. Its properties should look like this:



Description	
Owning model	/VacationRequest
Functional group	Technical process
Base type name	
Bean class name	com.mycompany.sample.vacation.VacationData
Data members	[6]
submitterName	
submitterEmail	
state	
reason	
fromDate	
toDate	
Tagged values	[]
Status	

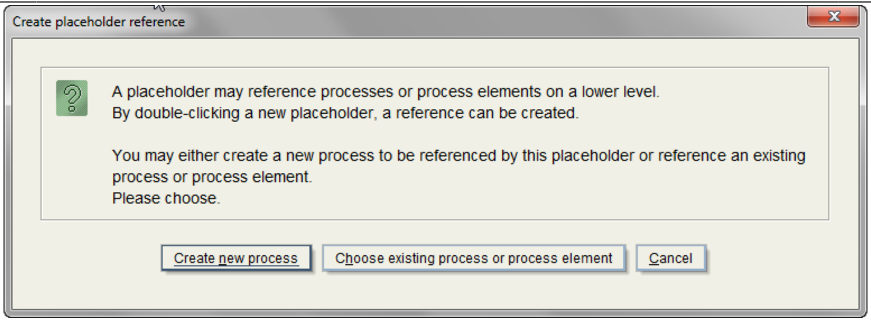
4.3. Creating a Technical Process

The placeholder elements that we used in the business process are abstract proxies of some subordinate functionality. Originating from the business process, we will create technical processes that implement the functionality indicated by the placeholder(s).

Note

If you prefer not have any connection between the business process and the technical process, simply create the technical process as you did in section Creating the Business Process. In this example, we will apply the top-down principle, orignating from our business process down to the technical level.

1. Double-click the `Approve request` placeholder node in order to create a technical process. In the dialog


Create placeholder reference

A placeholder may reference processes or process elements on a lower level.
By double-clicking a new placeholder, a reference can be created.

You may either create a new process to be referenced by this placeholder or reference an existing process or process element.
Please choose.

choose the option `Create a new process`.

2. In the subsequent process wizard, select 'Top level process' as process type and enter the following process properties:

Property	**Value/Description**
System name	`HandleVacationRequest`
Descripition	`Defines the actual vacation request processing.`
Functional group	`Technical process`

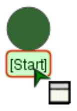
Click `Finish`.

4.4. Providing start parameters to the process

First, we need a point to begin. A process always starts at an *initial node*. Create an initial node named `Start`.

Now, provide a `RequestData` object as parameter to the node. Just drag the `RequestData` component from the component browser and drop it onto the exit socket of the start node.

.



Change the name of the parameter from `RequestData` to `Data`.

4.5. Activity Node

We will use an activity node to implement a simple verification of the input data.

1. Select the initial node in the workspace and drag the activity node symbol ▭ from the toolbox to the workspace below the initial node. Name the new activity `VerifyVacationData`.

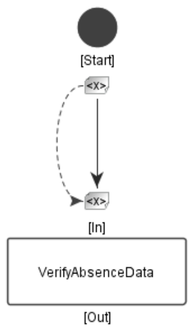
2. The activity will expect some data item to verify. We could define this parameter in the same way we have defined it for the initial node. However, since we have changed the default name suggested by the OpenBP modeler, we prefer to copy it.

   Right-click the parameter of the initial node and choose Copy or select it and press **CTRL+c**.

   Now right-click the entry socket of the new activity node and choose Paste or select it and press **CTRL+v** to paste the parameter to the socket.

   Note: Sockets that that have default names (either `In` or `Out`) and are connected to other sockets already are not displayed by default in order to save screen space. In order to make it visible, simply select the node that owns the socket. You might also choose to disable this optimization logic by toggling the button.

3. Connect the parameter of the initial node to the parameter of the activity by simply dragging a data link between the two. When you start dragging from the parameter, the areas where you can connect will appear in a light green overlay.

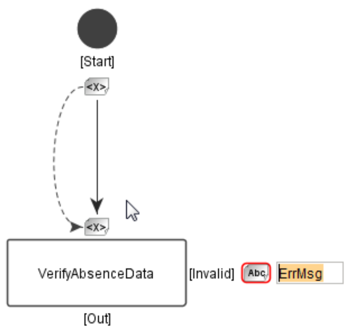


4. The outcome of the verification might be positive or negative. Depending on the result, we would like to take different paths of execution in the process. The activity already has a default exit socket called `Out`. We choose to use this socket in case of a successful verification.

   Create a second exit socket by dragging the exit socket symbol from the toolbox and drop it onto the activity. Name the new socket `Invalid`. Finally, bring the socket in the position you would like it to have. Move the cursor over the socket and press the **CTRL** key. Move the mouse to rotate the socket around the body of the activity. If you press the **SHIFT** in addition to the **CTRL** key, the socket will lock at 45° angles. Rotate the socket to the right of the activity rectangle.

   We would like to pass back a validation error message in case of validation errors. Let's define an output parameter that will receive the message text. Drag a `String` data type from the `System` unit in the component browser and drop it on the the socket. Name the parameter `ErrMsg`.

   The model should look lik this:



4.6. Using Scripts to Implement an Activity

For convenience, we will scripting to implement the (rather simple) activity logic. Later you will learn how to create a Java-based activity.

OpenBP currently supports the BeanShell language as scripting language. BeanShell was intended as full-features Java scripting language and should become a scripting standard with JSR 274. However, this JSR hasn't been closed but it also does not move forward. We might support different scripting languages in addition to the BeanShell in the future.

First, select the activity. In the property browser window below the workspace, you may edit the property `Bean shell script` in order to type the Bean shell script.

Tip

Note that the property browser does not support syntax highlighting or code completion. You might want to make life a little easier by editing your code with an external editor and copy/pasting it to the the property browser.

BeanShell supports most features of the Java 1.2 language. For details, see http://www.beanshell.org. OpenBP's Bean Shell support adds the following features:

- Input parameter support

  The input parameters of the entry socket of the node are copied to the Bean Shell's data context. You may access the parameter value by using a variable that has the same name as the parameter. Thus, you should name your variables in a way that they adhere to Java identifier syntax.

- Output parameter support

  Variables from the Bean Shell's data context will be copied to output variables of the exit socket that is executed when leaving the activity code (see below). In order to pass a value from the activity code to the process, simply assign the value to a Bean Shell variable and define an output parameter that has the same name on the exit socket that is to be taken.

- Exit socket selection

  When you do not specify an exit socket, execution will continue with the default socket of the activity. However, if you want to specify a different execution path, you can assign the name of the alternative exit socket to a special variable named `exit`.

- Entry socket detection

  Accompanying the exit socket selection mechanism, you may access the name of the entry socket through a special variable named `entry` in cases where an activity has more than one entry socket and you need to distringuish the entry path to the activity within the activity code.

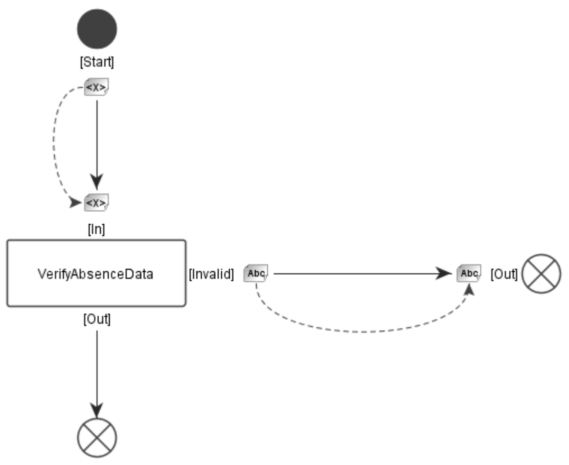
Insert the following Bean Shell code for the activity:

```
if (Data.state -!= 1)
{
    ErrMsg = -"Request has already been processed.";
    exit = -"Invalid";
}
```

4.7. Final Node

Finally, insert two final nodes in the process and connect them to the activity. Name them `Accepted` and `Failed`. Provide the `Failed` node with the error message string that has been returned by the activity.

If you got it right, you process might look like this:



4.8. Text Elements

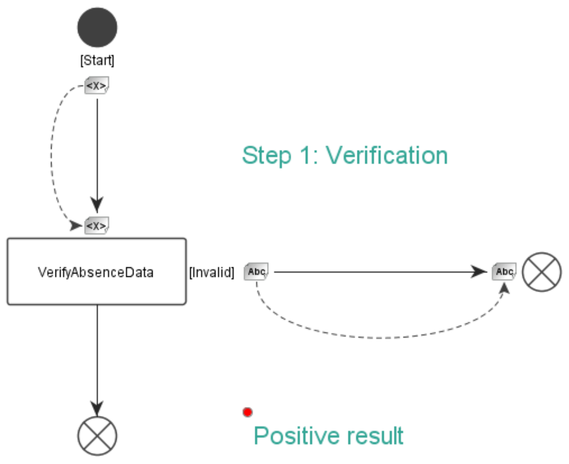
You might use text elements to add some documentation to the process. Just drag the text element symbol from the toolbox and drop it somewhere on the workspace. Edit the multi-line text in the property browser. You may enlarge the font by clicking on the red dot near the text and dragging it to the right. You may also assign the text a different color:

Select the Color tab of the window to the lower right. Click somewhere in the color space or move the color slider to the right to choose a color. You will notice that the preview area changes its color. In order to apply the color, drag it from the preview area and drop it onto the text in the workspace.

Tip

You can do this also with some node instead of the text element. Simply try it out...

Finally, we have a well-documented process:



5. Running Your First Process

5.1. The Sample Project

OpenBP is not a stand alone process execution environment by default. It is meant to be integrated into your software as a lightweight process engine.

For convenience and a jumpstart, we have provided a sample Eclipse project including maven pom file that implements a simple JUnit test case that runs an the sample process we are about to create. Simply copy the project directory `openbp-sample` to your environment and adopt it to your needs. When transferring it to your application, you should consider the following artifacts:

pom.xml	Maven project file Contains the dependencies to use OpenBP with Hibernate as persistence mechanism and the HSQLDB JDBC driver. *Copy the dependencies to you project's POM file*
.classpath	Eclipse classpath definition file *Copy the entries to your .classpath file or (better) regenerate from the pom.xml file*
src/main/ resources/OpenBP-Server.properties	Property file that defines several properties for the OpenBP engine. See section TODO for adjusting debugger-relevant values in this file. *Copy to your application's resource directory.*
src/main/resources/ OpenBP-Server-Sample.spring.xml	Spring configuration file that wires and configures the OpenBP components for running with Hibernate. For an explanation of OpenBP's Spring context, see section TODO *Copy to your application's resource directory.*
src/main/ resources/OpenBP-Quartz.properties	Configuration file for the Quartz scheduler for timed execution of OpenBP processes. Note required for the sample. *Copy to your application's resource directory.*
src/main/resources/ log4j.properties	Logger properties file. *Copy to your application's resource directory or copy the OpenBP portions to your own log4j configuration file.*
src/main/resources/ hibernate.cfg.xml	Hibernate configuration file containing the OpenBP entity definitions. You will probably have this yourself. Note that you do not have to add the OpenBP entity definitions to your Hibernate configuration, OpenBP does this itself upon initialization.
src/main/java/ com/mycompany/ sample/vacation/ VacationSampleTestCase.java	Sample JUnit test source code. Through this is a test case, it has been place in the main/java folder for this demo. Just copy the code portions you need to your application code.
src/main/java/ com/mycompany/ sample/vacation/ VacationData.java	A data object definition that is being used within the process. Just for demo.

5.2. Moving the Model to an Executable Environment

By default, the OpenBP modeler creates a new model in the `Model` folder in the OpenBP installation directory.

For most applications, it makes sense to place the process model as resource in the classpath of the application. So we have to move the model from its default location to our project location.

In order to do this, simply copy the `Model` directory from the OpenBP installation directory to the the `src/main/resources` directory.

We do not want to perform this action each time we change the model, so we instruct the OpenBP Modeler to look for the model in another location by supplying an environment variable to the Java VM of the OpenBP Cockpit. You might copy and modify the $OPENBP_HOME/bin/StartCockpit script to supply the parameter `-Dopenbp.FileSystemModelMgr.ModelPath=PATH` where `PATH` specifies the path to the model directory that you have copied to your project.

Alternatively, you may also modify the property `openbp.FileSystemModelMgr.ModelPath` in the OpenBP-Cockpit.properties file.

5.3. The OpenBP Process Management API

Let's take a look at a simple process invocation:

The following code creates an instance of the process server, which is used to execute processes (a.k.a. also known as the process engine). You may supply the name of a Spring configuration resource that contains your particular OpenBP wiring configuration.

```
// Create a process server from the supplied Spring confiugration
processServer = new ProcessServerFactory().createProcessServer("OpenBP-Server-
Hibernate.spring.xml");
ProcessFacade processFacade = processServer.getProcessFacade();
```

Caution

Creating a process server is an expansive operation. Thus you should do this once at application startup. The process facade is reentrant (i. e. stateless), so you may use a single instance in multiple threads.

A token context is an object that holds the state of an executing process including the position within the process, the data that is processed and various configuration information. Before we can start a process, we need to create a new token.

```
// Create the process context and start the process
TokenContext token = processFacade.createToken();
```

Note

When creating a token, the token will be written to the database if you have configured a persistent token context provider in the Spring configuration file. Note that when performing this operation for the first time, the database needs to be initialized first. The object-relational mapping frameworks such as Hibernate usually require some time for initialization, so this step might take a while if executed for the first time. Subsequent calls will execute pretty fast.

Setting the debugger id to a value not equal to null means that we enable the process for debugging. An OpenBP process debugger that provides the same debugger id will be able to attach itself to the process and monitor its execution (see below).

```
token.setDebuggerId("Deb1");
```

The process parameters are passed as map of values. The map keys must correspond to the names that have been used as parameter names of the initial node within the OpenBP process. The type of the parameters must correspond to the parameter types expected by the activities of the process. Otherwise, ClassCastExceptions or similar errors may appear.

```
Map<String, Object> inputParam = new HashMap<String, Object>();
inputParam.put("Data", data);
processFacade.startToken(token, -"/VacationRequest/AcceptVacationRequest.Start",
inputParam);
```

The `startToken` method will pass the token to the OpenBP engine for execution.

Note that the process will not be executed right away. Since a process engine is meant for background exeuction of business processes, you might see this more like an execution order that your are submitting. If you are using a persistent (i. e. database-oriented) token context service, this method call will cause the new token context to be saved to the database. You may configure the OpenBP Spring context regarding the type of token context service.

```
processFacade.startToken(token, -"/VacationRequest/AcceptVacationRequest.Start",
inputParam);
```

```
Usually, some worker thread of the OpenBP engine would scan the token database for
pending tokens and execute the ones that are waiting for execution.
However, in this test case, we want to have control over what's happening and
want to check the result of a process execution in the same thread as the process
invocation. For test cases and for the cases of synchronous process execution that
might arise from time to time, OpenBP provides a method that will scan for pending
token context objects and try to execute them before returning.
```

```
// Note: This should be done by some worker thread; for the test case, we do it
right here.
processFacade.executePendingContextsInThisThread();
```

```
Finally, after the process has been finished, we would like to have the process'
output parameters in a hash table so we can access them.
```

```
Map<String, Object> outputParam = new HashMap<String, Object>();
processFacade.retrieveOutputParameters(token, outputParam);
```

Finally, we should gracefully shut down the OpenBP server. Typically, you would do this when exiting the application, e. g. when unloading a web application or inside the shutdown hook of the Java VM.

```
processServer.shutdown(true);
```

5.4. Debugging a Process

OpenBP is able to track the execution of a process inside the OpenBP modeler. Similar to a Java debugger, the OpenBP modeler connects to the OpenBP engine running inside your application and monitors and controls the currently running process. OpenBP uses a RMI connection for the debugger to engine communication. Note that the ports in the OpenBP-Server.properties in your application's resources and the OpenBP-Cockpit.properties in the OpenBP home directory must match regarding the ports. OpenBP uses the following default settiings:

openbp.RMIRegistry.port=10088

Note

Make sure to enable the selected port in your firewall or the components will not be able to communicate.

In order to debug a process, follow these steps:

- Before you run an OpenBP application for the first time, make sure that all database tables required by OpenBP have been created.

  In the `openbp-server-VERSION-ddl.zip` (`VERSION` to be replaced by the actual OpenBP version) there are DDL scripts for various database dialects that will create the OpenBP tables. Execute theses scripts in your SQL execution environment. Watch for any errors that might appear.

- Start the application that you want to trace. After initializing the OpenBP process server, the application needs to wait until the OpenBP Cockpit has connected to the OpenBP engine.

  Basically, there are two alternatives:

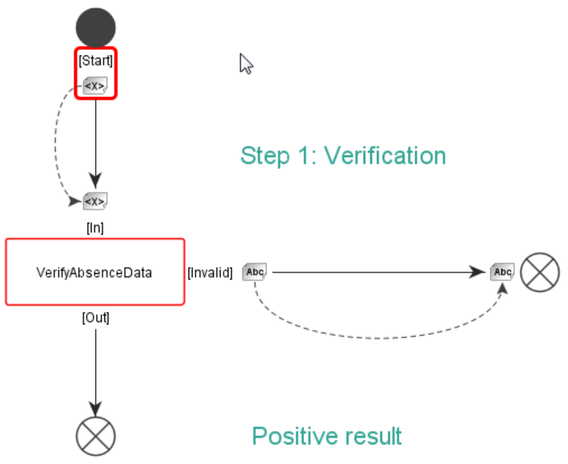
If you are running your application under control of a Java debugger, you may set a breakpoint before calling `processFacade.startToken`.

Otherwise, you can supply the argument `-Dopenbp.debuggerControl=waitAtFirstEvent` to the Java VM running your application that instructs the OpenBP engine to stop and wait until the OpenBP debugger has connected. When the engine is about to execute some process, it will print the message `Process waiting for debugger (debugger id 'Deb1')...` to the log output.

In either case, the execution of the program will halt before the process is being started.

- Start the OpenBP Cockpit. The Cockpit will automatically try to connect to the server.

  On successful connect, the Cockpit will display a red markation around the current position of the debugged process.



  If the Cockpit is already running, select Cockpit | Server Reload or press **CTRL+r** to reconnect to the OpenBP Server. On success, the Cockpit will print a message that it has connected to the server.

- The debugger provides the following commands. When the process has been halted, you may use one of the following The commands are available from the Debugger menu, through a toolbar inside the title bar (right hand side) of the workspace window and through keyboard shortcuts. The shortcuts have been chosen with Eclipse debugger compatibility in mind. Note that some commands are valid only during the execution of a process.

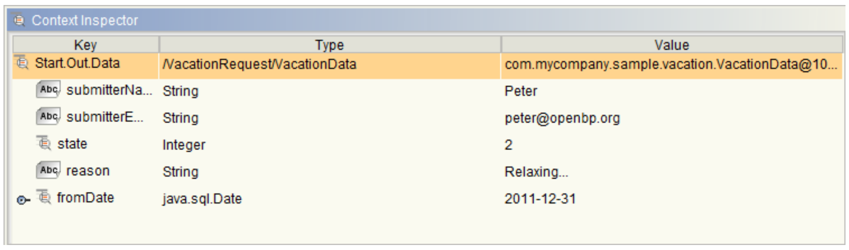
Icon	Command	Description	Shortcut
▶❘	Step Next	Continues execution to up to the next event. Available during process execution only. This is the most fine-granular debugger command. The debugger will visualize any process executions that are being executed by the engine, i. e. control link executions, data link executions, entry and exit socket positions and exceptions. Use this command when you want to learn how process execution works or when you want to track down a problem that occurs in a data link execution - or when you like to watch the dot flitting... :-)	**F5**
▼	Step Into	Continues execution to up to the next process node, stepping into sub processes.	**SF5**

Icon	Command	Description	Shortcut
		Available during process execution only. The process will stop at the next node. No data or control link animations will be displayed. When the current node is a sub process node, the execution will stop at the first node of the sub process.	
▶▶	Step Over	Continues execution to up to the next process nod, skipping over sub processese. Available during process execution only. The process will stop at the next node. No data or control link animations will be displayed. When the current node is a sub process node, the execution will stop at the first node after the sub process node (except when a breakpoint inside the subprocess has been hit).	k
▲	Step Out	Continues execution of a sub process until it returns. Available during process execution only. The process will stop at the first node of the parent process after the sub process node. When this function is applied in a top-level process, it has the same effect as executing the Resume function.	**F7**
▶●	Step until	Continues execution of a sub process until the selected node has been hit. Available during process execution only. When the node does not lie within the current execution path of the process, the function has the same effect as executing the Resume function. The function is available from the context menu of an initial or final node or socket only.	
▶	Resume	Continues execution of a process. Available during process execution only. Process execution will continue until a breakpoint has been hit, the process has ended (i. e. a final node has been executed) or is suspended (by a wait state or workflow node) or an exception has been thrown.	**F8**
■	Stop	Stops the execution of the process. Available during process execution only. The process is being terminated ungracefully. Usually, a rollback will be performed and the lifecycle state of the process will be set to `LifeCycleState.ERROR`. This means that the process will not be continued by the engine unless its lifecycle state is being changed programmatically.	**SHIFT+F8**
●	Toggle breakpoint	Sets or clears breakpoints at all selected nodes.	**F9**

Icon	Command	Description	Shortcut
		Select a socket to set a breakpoint at an individual socket or a single node or a group of nodes to set breakpoints at all sockets of the node(s).	
	Clear all breakpoints	Clears all breakpoints that have been set.	**SHIFT+F9**
	Break on top level	Toggle that automatically halts any process that is being started as top level process.	
	Break on workflow	Toggle that automatically halts any process that is about to execute a workflow node.	
	Break on exception	Toggle that automatically halts any process that has thrown an exception.	

- In the lower right of the Modeler, you will see a tab page named Context Inspector.

  It displays the current data context of a halted process.



Key	Type	Value
Start.Out.Data	/VacationRequest/VacationData	com.mycompany.sample.vacation.VacationData@10...
submitterNa...	String	Peter
submitterE...	String	peter@openbp.org
state	Integer	2
reason	String	Relaxing...
fromDate	java.sql.Date	2011-12-31

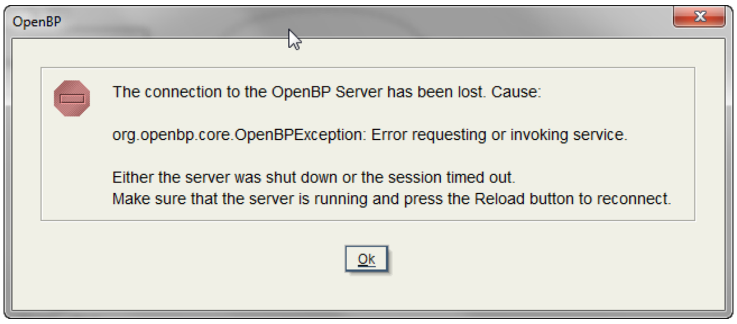
The Key column of the process data table denotes the name of the data item in the process' data table. If the key contains the dot ('.') character, it identifies a parameter variable having the form `NODE.SOCKET.PARAM`. As you single-step through the process you will see that these variables are of temporary nature and are present while the node is executing only. Process variables (see below) are listed with a leading underscore character.

The Type column indicates the data type. Complex data types that have been defined as OpenBP data type appear as qualified component name in the form `/MODEL/TYPE` where `TYPE` is the name of the data type and `MODEL` is the model that contains the data type definition. Other objects are represented by their Java class name. Primitive data types (`java.lang.String`, `java.lang.Integer`, ...) are displayed as simply `String`, `Integer` etc.

The Value column finally displays the `toString()` value of the object. Data items that have the `null` value appear empty.

Note that the context inspector is read-only. You cannot not change the displayed data.

- After our process terminates, the test case will also terminate. The Cockpit will detect this and print a notification message. To repeat the debugging session, simply restart the test case and press **CTRL +R** to reconnect to the new engine instance.



The connection to the OpenBP Server has been lost. Cause:

org.openbp.core.OpenBPException: Error requesting or invoking service.

Either the server was shut down or the session timed out.
Make sure that the server is running and press the Reload button to reconnect.

6. Process Data Handling

6.1. Using Process Variables

Work in progress...

Note

In the context inspector, process variables are listed with a leading underscore character when displaying the data of a halted process in order to determine them from socket parameters.

6.2. Decision Node

6.2.1. The Expression Language

Work in progress...

6.3. Merge Node

Work in progress...

7. Generating a Skeleton For Your Activity

7.1. Retrieving and Returning Activity Parameters

Work in progress...

7.2. Exception Management

Work in progress...

7.3. Running the Code-Aware Process

Work in progress...

8. Advanced Process Execution

8.1. Subprocesses

Work in progress...

8.2. Process Suspension and Resumption

Work in progress...

8.3. Process Execution Sequence Diagram

Work in progress...

8.4. Long-Running Processes

8.4.1. Wait State Node

Work in progress...