

Novell exteNd Composer

5.2

www.novell.com

PROCESS MANAGER USER'S GUIDE



Novell[®]

Legal Notices

Copyright © 2004 Novell, Inc. All rights reserved. No part of this publication may be reproduced, photocopied, stored on a retrieval system, or transmitted without the express written consent of the publisher. This manual, and any portion thereof, may not be copied without the express written permission of Novell, Inc.

Novell, Inc. makes no representations or warranties with respect to the contents or use of this documentation, and specifically disclaims any express or implied warranties of merchantability or fitness for any particular purpose. Further, Novell, Inc. reserves the right to revise this publication and to make changes to its content, at any time, without obligation to notify any person or entity of such revisions or changes.

Further, Novell, Inc. makes no representations or warranties with respect to any software, and specifically disclaims any express or implied warranties of merchantability or fitness for any particular purpose. Further, Novell, Inc. reserves the right to makes changes to any and all parts of Novell software, at any time, without any obligation to notify any person or entity of such changes.

This product may require export authorization from the U.S. Department of Commerce prior to exporting from the U.S. or Canada.

Copyright ©1997, 1998, 1999, 2000, 2001, 2002, 2003 SilverStream Software, LLC. All rights reserved.

SilverStream software products are copyrighted and all rights are reserved by SilverStream Software, LLC

Title to the Software and its documentation, and patents, copyrights and all other property rights applicable thereto, shall at all times remain solely and exclusively with SilverStream and its licensors, and you shall not take any action inconsistent with such title. The Software is protected by copyright laws and international treaty provisions. You shall not remove any copyright notices or other proprietary notices from the Software or its documentation, and you must reproduce such notices on all copies or extracts of the Software or its documentation. You do not acquire any rights of ownership in the Software.

Patent pending.

Novell, Inc.
404 Wyman Street, Suite 500
Waltham, MA 02451
U.S.A.

www.novell.com

exteNd Composer *Process Manager User's Guide*
[June 2004](#)

Online Documentation: To access the online documemntation for this and other Novell products, and to get updates, see www.novell.com/documentation.

Novell Trademarks

ConsoleOne is a registered trademark of Novell, Inc.
eDirectory is a trademark of Novell, Inc.
GroupWise is a registered trademark of Novell, Inc.
exteNd is a trademark of Novell, Inc.
exteNd Composer is a trademark of Novell, Inc.
exteNd Director is a trademark of Novell, Inc.
iChain is a registered trademark of Novell, Inc.
jBroker is a trademark of Novell, Inc.
NetWare is a registered trademark of Novell, Inc.
Novell is a registered trademark of Novell, Inc.
Novell eGuide is a trademark of Novell, Inc.

SilverStream Trademarks

SilverStream is a registered trademark of SilverStream Software, LLC.

Third-Party Trademarks

All third-party trademarks are the property of their respective owners.

Third-Party Software Legal Notices

The Apache Software License, Version 1.1

Copyright (c) 2000 The Apache Software Foundation. All rights reserved.

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met: 1. Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer. 2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution. 3. The end-user documentation included with the redistribution, if any, must include the following acknowledgment: "This product includes software developed by the Apache Software Foundation (<http://www.apache.org/>)." Alternately, this acknowledgment may appear in the software itself, if and wherever such third-party acknowledgments normally appear. 4. The names "Apache" and "Apache Software Foundation" must not be used to endorse or promote products derived from this software without prior written permission. For written permission, please contact apache@apache.org. 5. Products derived from this software may not be called "Apache", nor may "Apache" appear in their name, without prior written permission of the Apache Software Foundation.

THIS SOFTWARE IS PROVIDED ``AS IS" AND ANY EXPRESSED OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE APACHE SOFTWARE FOUNDATION OR ITS CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

JDOM.JAR

Copyright (C) 2000-2002 Brett McLaughlin & Jason Hunter. All rights reserved.

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met: 1. Redistributions of source code must retain the above copyright notice, this list of conditions, and the following disclaimer. 2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions, and the disclaimer that follows these conditions in the documentation and/or other materials provided with the distribution. 3. The name "JDOM" must not be used to endorse or promote products derived from this software without prior written permission. For written permission, please contact license@jdom.org. 4. Products derived from this software may not be called "JDOM", nor may "JDOM" appear in their name, without prior written permission from the JDOM Project Management (pm@jdom.org).

In addition, we request (but do not require) that you include in the end-user documentation provided with the redistribution and/or in the software itself an acknowledgement equivalent to the following: "This product includes software developed by the JDOM Project (<http://www.jdom.org/>)." Alternatively, the acknowledgment may be graphical using the logos available at <http://www.jdom.org/images/logos>.

THIS SOFTWARE IS PROVIDED ``AS IS" AND ANY EXPRESSED OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE JDOM AUTHORS OR THE PROJECT CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

Sun

Sun Microsystems, Inc. Sun, Sun Microsystems, the Sun Logo Sun, the Sun logo, Sun Microsystems, JavaBeans, Enterprise JavaBeans, JavaServer Pages, Java Naming and Directory Interface, JDK, JDBC, Java, HotJava, HotJava Views, Visual Java, Solaris, NEO, Joe, Netra, NFS, ONC, ONC+, OpenWindows, PC-NFS, SNM, SunNet Manager, Solaris sunburst design, Solstice, SunCore, SolarNet, SunWeb, Sun Workstation, The Network Is The Computer, ToolTalk, Ultra, Ultracomputing, Ultraserer, Where The Network Is Going, SunWorkShop, XView, Java WorkShop, the Java Coffee Cup logo, Visual Java, and NetBeans are trademarks or registered trademarks of Sun Microsystems, Inc. in the United States and other countries.

Indiana University Extreme! Lab Software License

Version 1.1.1

Copyright (c) 2002 Extreme! Lab, Indiana University. All rights reserved.

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met: 1. Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer. 2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution. 3. The end-user documentation included with the redistribution, if any, must include the following acknowledgment: "This product includes software developed by the Indiana University Extreme! Lab (<http://www.extreme.indiana.edu/>)." Alternately, this acknowledgment may appear in the software itself, if and wherever such third-party acknowledgments normally appear. 4. The names "Indiana University" and "Indiana University Extreme! Lab" must not be used to endorse or promote products derived from this software without prior written permission. For written permission, please contact <http://www.extreme.indiana.edu/>. 5. Products derived from this software may not use "Indiana University" name nor may "Indiana University" appear in their name, without prior written permission of the Indiana University.

THIS SOFTWARE IS PROVIDED "AS IS" AND ANY EXPRESSED OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE AUTHORS, COPYRIGHT HOLDERS OR ITS CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

Phaos

This Software is derived in part from the SSLava™ Toolkit, which is Copyright ©1996-1998 by Phaos Technology Corporation. All Rights Reserved. Customer is prohibited from accessing the functionality of the Phaos software.

W3C

W3C® SOFTWARE NOTICE AND LICENSE

This work (and included software, documentation such as READMEs, or other related items) is being provided by the copyright holders under the following license. By obtaining, using and/or copying this work, you (the licensee) agree that you have read, understood, and will comply with the following terms and conditions.

Permission to copy, modify, and distribute this software and its documentation, with or without modification, for any purpose and without fee or royalty is hereby granted, provided that you include the following on ALL copies of the software and documentation or portions thereof, including modifications: 1. The full text of this NOTICE in a location viewable to users of the redistributed or derivative work. 2. Any pre-existing intellectual property disclaimers, notices, or terms and conditions. If none exist, the W3C Software Short Notice should be included (hypertext is preferred, text is permitted) within the body of any redistributed or derivative code. 3. Notice of any changes or modifications to the files, including the date changes were made. (We recommend you provide URIs to the location from which the code is derived.)

THIS SOFTWARE AND DOCUMENTATION IS PROVIDED "AS IS," AND COPYRIGHT HOLDERS MAKE NO REPRESENTATIONS OR WARRANTIES, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO, WARRANTIES OF MERCHANTABILITY OR FITNESS FOR ANY PARTICULAR PURPOSE OR THAT THE USE OF THE SOFTWARE OR DOCUMENTATION WILL NOT INFRINGE ANY THIRD PARTY PATENTS, COPYRIGHTS, TRADEMARKS OR OTHER RIGHTS.

COPYRIGHT HOLDERS WILL NOT BE LIABLE FOR ANY DIRECT, INDIRECT, SPECIAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF ANY USE OF THE SOFTWARE OR DOCUMENTATION.

The name and trademarks of copyright holders may NOT be used in advertising or publicity pertaining to the software without specific, written prior permission. Title to copyright in this software and any associated documentation will at all times remain with copyright holders.

Contents

About This Book.	9
Getting Started with Process Manager: 5-Minute Tutorial.	12
How Do I Deploy It?	17
1 Welcome to Composer and Process Management.	19
What Is Process Management?	19
Why Automated Process Management?	20
Process Design versus Application Design	21
Modularity	21
Example of a Simple Straight-Through Process	22
Process Management and Emerging Technologies	22
How Does a Process Differ from a Service?	23
Process Management Terms and Concepts	25
Activities, Messages, and Links	26
Sequencing, Timing, and Process-Level Logic	28
Control Flow Logic	28
Deferred Mode versus Immediate Mode	30
Map Policy and Data Merging	32
Timeouts and Retries	32
Data Flow Patterns	32
Lifecycle Events	33
Process Manager Architectural Layers	33
Process Manager FAQ	35
2 Preparing to Model a Process	39
Process Server Execution Model	39
The Design-Time View	41
Flow Control Strategies	43
Branch Logic	43
Join Logic	45
Looping	46
How Safe Looping Can Be Accomplished	47
Process Architecture in Review	53
Taking a Best-Practices Approach	54
3 Creating and Testing Processes	57
Example: A Simple Straight-Through Process	57
Description	57
Process-Building Basics	58
Creating a New Process	58
About Service Provider Resources	60
About Service Provider Type Resources	64
Creating Activities	65
Creating Links	67
Message Mapping	68
Message Naming	68
How to Define Message Mappings	69
Data Mapping for Start and End Activities	71
Selecting a Process Input Template	71

Applying Flow Logic at the Activity Level	71
Timeouts and Retries	73
Map Policy	74
Fault Messages and Fault Handling	75
System Faults	75
Timeout Faults	76
Fault Handling	76
Animation and Testing	78
Aids to Debugging	81
Watching System Messages at Animation Time	82
Inspecting Messages	83
4 The Process Designer User Interface	85
Main Features	85
The Process Designer Window	85
Graph Elements	87
Menu Commands	89
Process Properties	94
Object Properties	94
Activity Properties	94
Composer Component	95
Activity Tab	95
Messages Tab	97
UI Tab	98
Web Service Send	99
Web Service Send Activity Tab	99
Web Service Receive	101
Web Service Receive Activity Tab	101
Subprocess	102
Synchronize Subprocesses	102
Link	103
Link Tab	103
UI Tab for Links	104
Graph Object Properties	105
Process Messages Tab	105
Graph UI Tab	106
Selected Node Properties on UI Tab	107
UI Tab (Selected Node Properties)	107
Text Object Properties	108
UI Tab	108
Layout Properties	110
General Layout Tips	111
Customizing the Canvas	112
5 Advanced Topics	115
Web Service Receive	115
Implementation Independence	117
Synchronize Subprocesses Activity	119
Data Mapping in the Synchronize Subprocesses Activity	120
Fault Handling	122
Waiting Activities	122
“Waiting Activity” Actions	123
6 Waiting Activities and Addressees	127
Understanding How Processes Are Triggered	127
Process-Related Actions	128
The Process Execute Action	128
How to Create a Process Execute Action	129
Deployment and the Process Execute Action	130

Find Waiting Activity Action	131
Finding a Waiting Activity	132
The Find Waiting Activity Dialog	133
Release Waiting Activity Action	135
The Release Waiting Activity Dialog	136
Human Participation in Processes	136
Addressees	137
The Role of the Web Service Receive Activity	138
Browse Waiting Activities Action	139
Where to Use the Browse Waiting Activities Action	139
Creating a Browse Waiting Activities Action	140
Lock/Unlock Waiting Activity	141
Prerequisites for Locking/Unlocking an Activity	142
Creating a Lock/Unlock Waiting Activity Action	142
The Reassign Addressee Action	143
Reassigning an Addressee	144
Creating a Reassign Addressee Action	144
7 Runtime Administration of Processes	147
Server Console Usage	147
Process Manager Console: Main Tab	147
Process Manager Console: Status Tab	152
Process Manager Console: Log Tab	154
Detail View for a Process Instance	154
A Testing	159
Environmental Differences between Design-Time Testing and Server Testing	159
B Performance Tuning	161
Configuration Options	161
Cache	161
Sleep Time	161
Cutoff Period	161
Total In-Memory Process Instances	161
C Process Management Glossary	163

About This Book

Purpose

This guide describes how to use the eXtend Composer Process Manager to build potentially largescale, long-running, automated processes that rely, in whole or in part, on Web Services. The guide is intended to be an adjunct to (not a replacement for) the eXtend Composer User's Guide.

Audience

This guide is aimed at persons tasked with design and deployment of coordinated systems of automated activities (that is, business process models). Anyone participating in the development of such systems should read this guide.

Prerequisites

You should be familiar with XML-related standards (including Schema, XSL, and XPath), the Document Object Model, and WSDL metaphors and motivations, in addition to basic J2EE concepts involving file packaging (JAR/EAR/WAR files).

Additional documentation

For the complete set of Novell exteNd user guides and other documentation, see the [Novell Documentation Web Site \(http://www.novell.com/documentation-index/index.jsp\)](http://www.novell.com/documentation-index/index.jsp).

Organization

This document is organized as follows:

Chapter	Description
Chapter 1, <i>Welcome to Composer and Process Management</i>	Gives a definition and overview of the Process Manager and key process-modelling concepts.
Chapter 2, <i>Preparing to Model a Process</i>	Briefly describes Process Manager design-time concepts and user-interface features.
Chapter 3, <i>Creating and Testing Processes</i>	Explains process deployment options and how to use the Process Administrator console to manage process instances.
Chapter 4, <i>The Process Designer User Interface</i>	Outlines the key factors that should be considered when designing a process, and presents various scenarios. A brief example is explained in walkthrough fashion.
Chapter 5, <i>Advanced Topics</i>	Discusses scenarios involving the Web Service Receive activity and the Synchronize Subprocesses activity, with information, also, about ways to implement human-centric workflows involving queued work items.
Chapter 6, <i>Waiting Activities and Addressees</i>	Discusses a wide variety of issues relevant to invocation and control of processes and activities.

Chapter	Description
Chapter 7, <i>Runtime Administration of Processes</i>	An introduction to the administrative consoles that can be used to monitor and control running processes.
Appendix A, <i>Testing</i>	A discussion of important differences between design-time and server-side testing.
Appendix B, <i>Performance Tuning</i>	This appendix explains the parameters that may be adjusted for obtaining better Process Server performance in an environment where performance is critical.
Appendix C, <i>Process Management Glossary</i>	Defines a variety of process management terms.

About the PDF Documentation

The PDF documentation can be viewed using Acrobat Reader 3.0 or higher. The current version of Reader (5.1 as of this writing) can be obtained free at:

<http://www.adobe.com/products/acrobat/readstep.html>

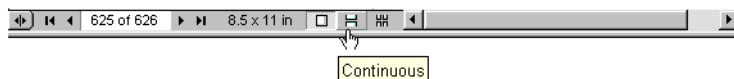
Navigation

Various navigational features are available when viewing this document in Acrobat Reader:

- ◆ The Bookmarks frame (left side of window) lists the contents of the document, by chapter name, heading, and subheading. Every topic listed in the content tree is a clickable link. To flip open the entire subtree (all children) under any tree node, *Control-click* the parent node. To toggle the visibility of the Bookmarks frame itself, press F5.
- ◆ Every item in the book's Table of Contents (page iii) is a clickable link that will take you directly to the text discussion. This is true also for Index entries.
- ◆ Wherever a web address (URL) appears, you will usually find that clicking on it will take you to the web site in your browser. *Even if the URI is not in blue or underlined, it will generally be a hot link.* You can test this by hovering the mouse over the URI. The cursor will change from an arrow to a finger cursor if the link is hot.
- ◆ *Cross-references* within and between chapters are also clickable. Again, this will be apparent from the cursor's appearance when you hover over a cross-reference.
- ◆ Use *Control-N* to go directly to a given page in the document. A dialog will prompt you for the page number.
- ◆ Use *Control-F* to execute a text search.

Copying and Pasting Text

- ◆ To copy PDF text to the clipboard, first enable the Text Selection tool (shortcut: 'V' on the keyboard), then click-drag to select text.
- ◆ To select large portions of text spanning several PDF pages, first click the "Continuous Pages Mode" icon in the button bar at the bottom of the Acrobat window (see below). Then shift-drag to select text (or use Control-A to Select All). You can then Copy the selected text to the clipboard.



- ◆ To keep text styling intact when Pasting clipboard selections into Word or other applications, choose Paste Special (if available) from the target application's Edit menu. In Microsoft Word, for example, this will allow you to paste clipboard text in RTF (Rich Text Format), retaining the text's styles.

Server-Side Installation and Setup

You will need to configure Composer Process Server's database options manually after doing the server-side install of Process Manager. To complete your installation, follow the steps shown below.

Prerequisites

Before undertaking the database-setup procedure, you should make sure the following conditions are met:

- ◆ A SilverStream, WebSphere, or WebLogic application server must be installed.
- ◆ A Sybase, Oracle, or IBM DB2 database system must be installed. (Check the Composer Process Manager Release Notes for the latest database compatibility matrix.)
- ◆ The application server must be set up to access an existing or new database:
 - ◆ Using your database system's administration facility, create a new database.
 - ◆ If necessary, create an ODBC data source for the database.
 - ◆ Using your application server's administration facility, add the newly created database as a data source for the application server, including a connection pool.

Setting Up a Process Manager Database

The Process Server uses a database for runtime storage of critical process data. You will need to designate a database to use for this purpose; then you must "bind" the Process Server to this database using the following procedure.

NOTE: Before performing the following steps, be sure you have created a dedicated database for the Process Manager as indicated in the previous section ("Prerequisites," above).

➤ To set up a Process Manager Database:

- 1 Start your application server.
- 2 Install the Composer Process Manager onto the application server if it is not already installed.
- 3 Using a Web Browser, access the Process Manager Console on the application server (i.e.: **http://<hostname>:<port>/eXtendComposerProcess/**). See "Process Database Info" on page 148 for console screen shots and additional information.
- 4 Confirm the unconfigured state of the Process Engine:
 - ◆ The Process Engine Status should read: "Shut Down"
 - ◆ The Process Database Info should read: "Invalid Configuration"
- 5 Press the **Configure** button and the Process Database Configuration screen appears. (See "Process Database Info", starting on page 148.)
- 6 Select the type of database for the Process Manager to use: Sybase, Oracle, or IBM DB2.
- 7 Enter the application server specific **Pool Name** for the database the Process Manager will use. For example:
 - ◆ SilverStream: Databases/<database name>/DataSource
 - ◆ WebSphere: jdbc/<DBPoolName>
 - ◆ WebLogic: an arbitrary JNDI data source nameWebSphere and WebLogic both use the JNDI Name specified by the user when creating the Connection Pool. So if the user creates a Connection Pool called *ProcessPool* with a JNDI name of *ProcessJNDI*, the user will enter *ProcessJNDI* in the Pool Name field of Composer's Process Console on the Process Database Configuration screen (see picture, page 149).
- 8 Enter the **User Name** (e.g. "dba") and **Password** for the database (e.g. <sql>).

NOTE: When configuring the Process engine's database using the Process Database Configuration screen (see page 149), the **Username** and **Password** for a SilverStream or WebSphere server will be the *database* Username/Password (for example, "dba/sql" for a Sybase database). But for WebLogic, the Username/Password needed to configure the Process database is the WebLogic *server* username/password ("system/weblogic," for example).

- 9 Press the **Save** button. If successful, the **Initialize** button appears.
- 10 To set up the Process Manager database tables, press the **Initialize** button.
- 11 If successful, the **Status** will read "Connected - Ready".
- 12 Press the **Return to Main** button to access the Process Manager Console.
- 13 To start the Process Manager engine, press the **Start** button. If successful, the Status under Process Engine Info should read: "Running".
- 14 For the Silverstream application server only, access the Server Management Console and synchronize the Process Manager database once the Process Manager engine is running.

NOTE: To reinitialize the database or change to another database, you must stop the engine first, and repeat the above steps.

Getting Started with Process Manager: 5-Minute Tutorial

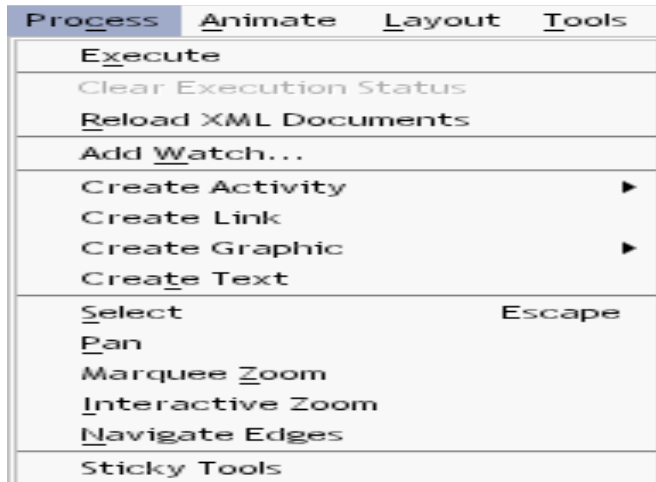
Creating a process in Process Manager is easy and straightforward. The steps below describe the basic procedure. For a fuller discussion of relevant concepts, you will obviously want to consult the chapters called "The Process Designer User Interface" and "Creating and Testing Processes" (as well as other relevant portions of this guide).

The basic procedure is always the same:

- ◆ Place activities on the process graph
- ◆ Link them together
- ◆ Specify relevant data mappings

➤ To create a Process:

- 1 Launch Composer. In the **File** menu, select **New > xObject**, then open the **Process/Service** tab, as shown below, and select **Process**.



- 2 The “Create a New Process Component” dialog will appear. Enter a **Name** for your process.

Create a New Process Component

Please enter a name and, optionally, a description for the Process Component. The name will appear in the Composer Detail Pane and in choice lists for XObjects in Composer. The name may not contain the characters: / : ? < > . | Names are case sensitive.

Name:
SampleProcess

Description:
Purpose:
Input:
Output:
Remarks:

Help Back Next Cancel

- 3 Click **Next** to bring up the second (and final) dialog of the wizard.

Create a New Process Component

Specify one or more XML Templates to help design Input to this Component or Web Service and only one to design Output. The sample XML Documents in each Template are design time aids to help you build Action Models for the component. The samples are not actually used at runtime after deployment to your application server. The Identifier is fixed and represents the name used to refer to the XML Document during component execution. Selecting System (ANY) allows you to use an empty template (i.e. accept any document as Input).

Input Message

Part	Template Category	Template Name
Input	{System}	{ANY}

Add Delete

Output Message

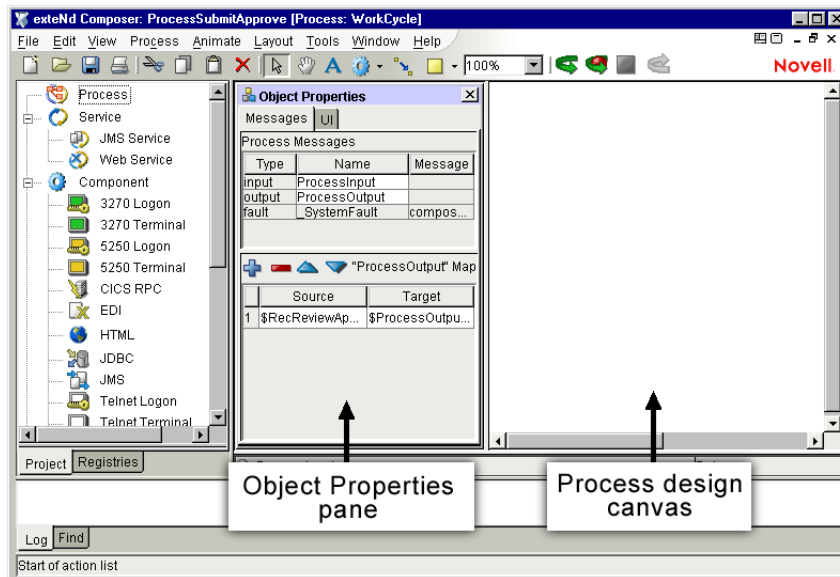
Part	Template Category	Template Name
Output	{System}	{ANY}

Add Delete

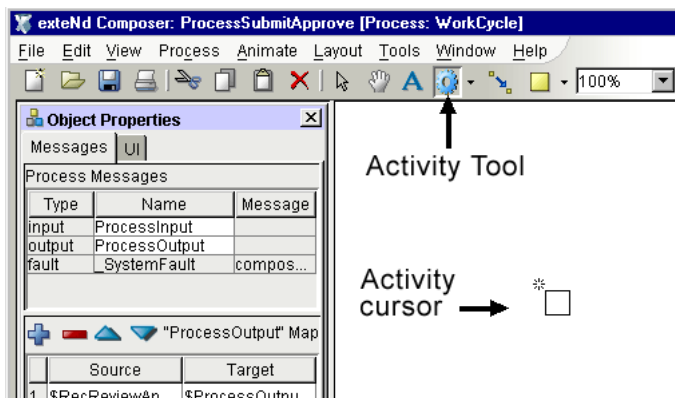
Help Back Finish Cancel

- 4 The second dialog allows you to choose **XML Templates** for your process input and output messages. Select these as you normally would when setting up any other Composer Component. (See the *Composer User's Guide*.)

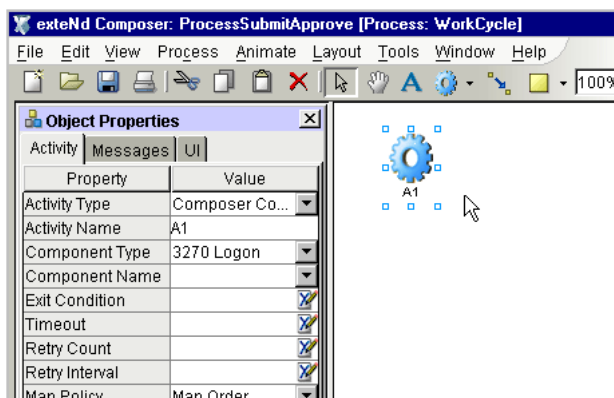
- Dismiss the dialog. A **blank canvas** (representing the area where you will draw your process) appears in what would ordinarily be the Native Environment Pane.



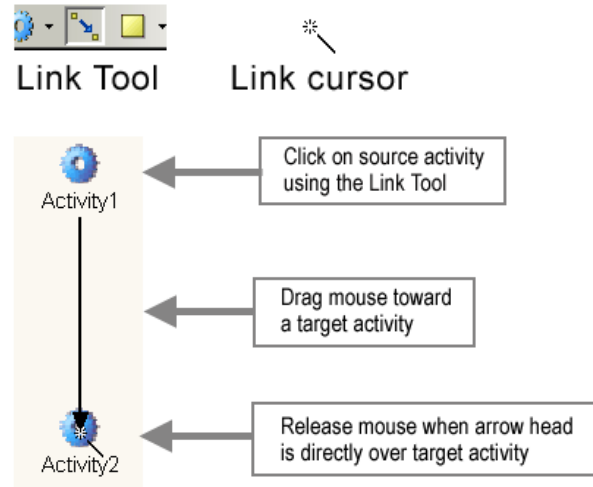
- The Object Properties pane should be visible (as above). If it is not, toggle its visibility using the **Object Properties** command under the **View** menu. Note that you can tear off (or undock) this pane if you want to drag it out of the way at any time.
- Click on the **Activity Tool** to select it. (See below.) The cursor will change appearance.



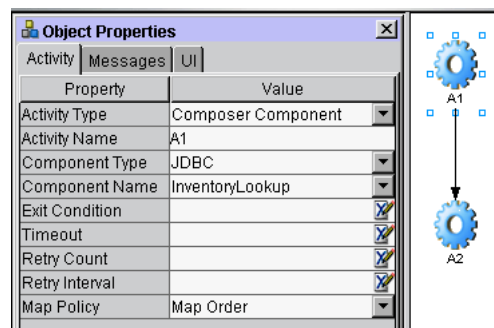
- Click anywhere on the blank canvas. A new activity is created, with blue stretch-handles positioned around its boundary.



- 9 With the new activity still selected (i.e., in focus), use **Copy** and **Paste** to create another copy of it (or use the Activity Tool again to create another activity on the canvas). You should now have two activities: **A1** and **A2**.
- 10 Select the **Link Tool** on the toolbar. Connect the two activities with a link in the manner shown below.

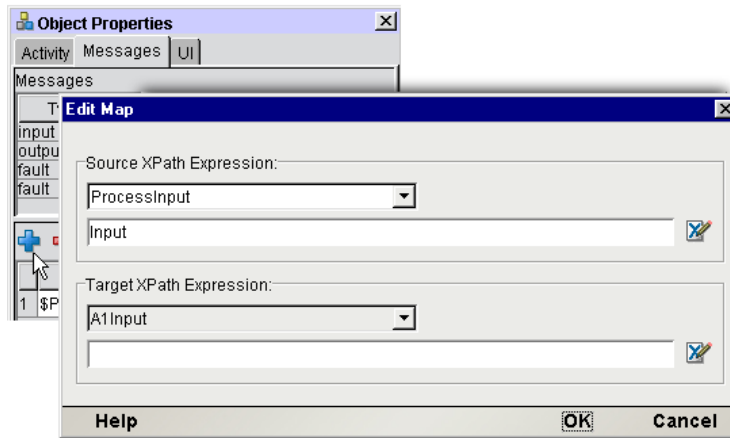


- 11 Now it is time to associate the activities with components (concrete implementations). Bring the Object Properties pane into view (with **View > Object Properties**) if it is not already visible. Click on the first activity. The Object Properties pane will update to show the current properties for that activity.
- 12 From the dropdown menu next to **Component Type**, select the *type of component* you would like to use as the implementation of your first activity: XML Map, JDBC, etc.
- 13 From the dropdown menu next to **Component Name**, select an actual component. (This list will be prepopulated with the names of components that already exist in the current project.) The graphic below shows what your object properties should now look like, assuming you chose a JDBC Component named *InventoryLookup*.



- 14 Click the **Messages** tab of the Object Properties pane.

- We want to associate input with this activity, so click the blue **Plus Sign** in the Messages tab. A dialog appears.



- Because this is the first activity in the process, we will want to specify *ProcessInput* as the message **Source**. (This will be the default.) *ProcessInput* will have the data structure corresponding to the XML Template that you specified for input in Step 3 earlier.
- Since the **Target** of our maps is **A1** (or whatever the currently selected activity is named), we will want to specify *A1Input* as the target message and *Input* as the target message *part* (as shown). You can think of the *Input part* as corresponding to the Input DOM in your component.
- Repeat Steps 10 through 16 for **Activity A2** (the second activity you created). Remember, once again, that you are mapping data *into* (not out of) the selected activity (A2). This time, the **Source** for the current activity's input will be the previous activity's output (*A1Output*). Therefore, your data mapping will probably look like:

"A2Input" Maps		
	Source	Target
1	\$A1 Output/Output	\$A2Input/Input

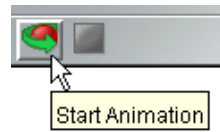
- The output of this simple process will simply be the second activity's output. This requires one more data mapping, to specify, explicitly, the mapping from A2 to *ProcessOutput*. To set this up, *click on an empty portion of the canvas* (to deselect all activities). Then go back to the **Object Properties** pane. You will see that it has changed to reflect the properties of the process-as-a-whole.
- Click the blue **Plus Sign** and create the data mapping shown below.

"ProcessOutput" Maps		
	Source	Target
1	\$A2Output/Output	\$ProcessOutput/Output

- Save** your work.

Congratulations: You have created your first Process!

Now might be a good time to animate through the process to be sure it does what you expect it to do. Using the **Animation Tool** on the toolbar (see below), you can begin stepping through the process from beginning to end.



Additional toolbar tools allow you to Step Into or Step Over the components that make up your activity implementations. If you Step Into one of those components, animation continues in real time at the action-model level. That is, you can step into or over any of the actions in the component, thus executing the component in stepwise fashion. When the final action executes, the component will return and you will be back at the process-graph level, where you can continue animating to the next activity, etc.

How Do I Deploy It?

Since a Process is just a Composer xObject (like any component or service created with Composer), it is deployed as part of a *project*, following the same procedure as with other projects (using the same Deployment Wizard). Obviously, the target app server must have both Composer Server and Process Server installed before your deployed processes can be run. Be sure you have done your server-side installs of all Composer products before deploying.

Just as *components* must be called from *services*, processes must also be invoked from Composer services. To do this, you simply place a Process Execute action in the action model of any service. Then deploy the service. (The service's input message can be passed straight through to the process. See the discussion of "The Process Execute Action" later in this guide.) If the service is deployed on a public URL, incoming requests will trigger new instances of the associated process. Those instances, and the status of all associated activities, can be monitored via Process Server Consoles. (See "Runtime Administration of Processes" elsewhere in this guide.)

For more information about deploying Composer services, be sure to consult the *eXtend Composer Server User's Guide* appropriate to your app server environment.

1

Welcome to Composer and Process Management

Welcome to the *SilverStream eXtend Composer Process Manager*. This Guide is a companion to the *eXtend Composer User's Guide*, which explains the core features of Composer. The rest of this guide assumes familiarity with core Composer functionality, so if you haven't looked at the *Composer User's Guide* yet, please familiarize yourself with it before using this Guide.

Before you begin working with the Process Manager, you must have it installed in your existing Composer environment. Likewise, before you can run any server-based processes, you must already have installed the Composer Process Server software on your application server.

To be successful with the Process Manager, you should be familiar with the following:

- ◆ Business Process Management (BPM) concepts
- ◆ The particular app server environment (e.g., SilverStream, WebSphere, or WebLogic) into which you will be deploying
- ◆ XML, XSD (schema), and XPath
- ◆ WSDL (the Web Services Description Language)
- ◆ Java WAR (Web Archive) files
- ◆ The use of eXtend Composer (and the eXtend Developer Workbench) to create and deploy services
- ◆ Basic structured-programming concepts and object-oriented design patterns

It will also help if you already have some knowledge of the Web Services Flow Language. The complete specification can be seen at

<http://www-4.ibm.com/software/solutions/webservices/pdf/WSFL.pdf>

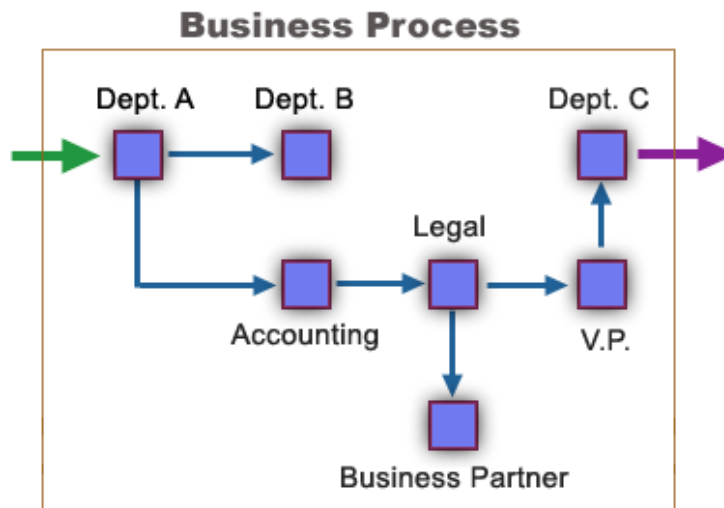
This chapter presents an overview of key BPM (or “workflow”) concepts so that you can better understand the relationship of Web Services, J2EE applications, and Composer applications to automated workflows.

What Is Process Management?

The aim of Business Process Management is to model business operations as well-defined *systems of tasks* in which participants interact according to a prescribed choreography to achieve a desired goal.

The top-level unit of work in such a model is usually called a *process*, emphasizing the dynamic nature of the underlying interactions. Because of the directed flow of work through such a system, the resulting model is often said to encapsulate a *workflow*.

A non-automated workflow might look something like the following.



In this hypothetical flow model, various parties (inside and outside the company) accomplish various tasks in a well-defined sequence. The input that triggers the overall process might be a phone call; the output might be a signed contract. The process has identifiable players with defined roles and responsibilities. If each participant does its job, the desired objective will be achieved.

An *automated* business process attempts to model the same interactions in terms of enterprise applications (each one, again, with its own roles and responsibilities). For maximum flexibility, the applications might be (but need not always be) implemented as Web Services. To accommodate human input, some of the applications might have a user-facing presentation layer. In some cases, the entire model might be realized in software, with no need for human intervention.

Why Automated Process Management?

The ultimate aim of BPM is to make possible the *automation* of complex and/or long-running business processes. The benefits of process automation go far beyond the obvious one of reducing demands on human resources. They include:

- ◆ **Scalable throughput**—Capacity no longer hinges on personnel headcount. Logjams during “busy times” can be avoided.
- ◆ **Consistency**—Once business rules are formalized as part of a process, they are adhered to reliably. TPA (Trading Partner Agreement) provisions can be enforced and company performance documented.
- ◆ **Adaptability**—Processes can be designed to automatically detect and route around unexpected bottlenecks.
- ◆ **Upgradability**—Processes can be rewired quickly to adapt to changes in business requirements. Individual components of a process can be modified or “changed out” without necessitating a total rewrite of the process itself.
- ◆ **Powerful audit capabilities**—Comprehensive reporting across activities, process instances, and business units is possible without the need to pull together disparate data sets from a variety of sources.
- ◆ **Better ability to respond to customer needs**—Processes can be initiated by the customer or trading partner in real time and executed on a 24/365 availability basis. Turnaround times can in some cases be shortened from days to hours, or hours to minutes.
- ◆ **New opportunities for Business Process Improvement**—The powerful audit and reporting capabilities afforded by BPM can yield new categories of process-related analytics that expose inefficiencies and opportunities for improvement within the organization.

Process Design versus Application Design

Process design and application design start from different points of reference. The design of an enterprise *application* usually involves a narrowly focused, data-centric view of a problem and a correspondingly scoped data-oriented solution. *Process* design, on the other hand, is motivated by the need to fulfill a business objective: patent an invention, process a claim, conduct an auction. The input to the process may be a phone call; the output might be 55-gallon drums on a truck. Carrying out the process may require completion of many tasks. Data requirements may vary greatly along the chain of tasks.

Process design is more than just “data in, data out.” It requires thinking about the Big Picture, including not only *which applications* one might use in modelling a process, but the time order in which those applications must run; the guarantees made by, and responsibilities of, the applications that make up a process; the possible interdependencies of the applications; and the various ways in which a process might terminate prematurely even though no application has failed.

Modularity

The concept of modularity is key to process modelling. For example:

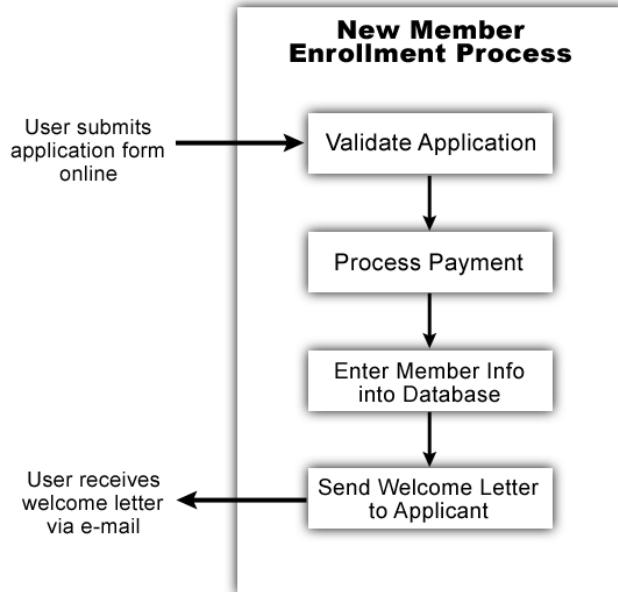
- ◆ The various constituent activities that make up a given business process (or workflow model) can, themselves, be processes. This is sometimes called *recursive composition*.
- ◆ A particular activity may play a role in multiple processes (which may be unrelated to each other). For example, the “credit check” activity of Process A might also be used by Process B and Process C. This is *activity reuse*.
- ◆ The implementation of an activity can be changed without affecting the process model itself. For example, new business logic, reflecting a change in company policy (or perhaps a change in algorithms), can be instituted in the “credit check” activity of a process; but the process itself doesn’t have to be modified.

NOTE: The principle of dividing large, custom-built chunks of work into smaller general-purpose chunks of work is well known to application developers as *factoring*. The goal of factoring is to promote *reuse* of costly resources.

The activities that make up a process may involve public-facing Web Services, or they may be limited to “behind-the-firewall” services running on a local app server. External trading partners may or may not be participants, and the process can be long-running, with lots of “callbacks” into the system, or it could be of relatively short duration (i.e., straight-through processing).

Example of a Simple Straight-Through Process

An example of an automated business process is shown in the following graphic.



In this scenario, a membership organization accepts member applications online. The applicant, upon submitting HTML form data to the organization's web site, triggers an automated process consisting of four activities:

- 1 The first activity checks the application for completeness and perhaps looks in a database to see if the applicant is already a member.
- 2 The second activity processes the user's electronic payment information.
- 3 Once payment has been received, relevant information about the new member is entered into the main membership database.
- 4 The final activity sends a personalized welcome message to the new member via e-mail.

In this admittedly simple example, any of the four component activities of the New Member Enrollment Process might represent an automated process in its own right. One of the activities (Process Payment) might very well rely on a Web Service offered by a business partner. Others might be local to the app server.

Process Management and Emerging Technologies

Modelling a high-level business function in terms of tasks that can be linked together via software is a powerful metaphor that plays well to the strengths of the Web Services model in particular and distributed computing in general. With the advent of technologies like XML, SOAP, WSDL, and UDDI, it becomes practical to design and deploy powerful, robust, sophisticated business applications that rely on the coordinated efforts of smaller, task-oriented units of work that can be "wired together" without respect to each unit's implementation details. (Separation of interfaces from implementations is a key feature of Web Services architectures.)

The eXtend Composer Process Manager leverages many of today's most important enterprise-computing technologies, including:

- ◆ **XML** (eXtensible Markup Language) for data portability
- ◆ **SOAP** (Simple Object Access Protocol) for platform-neutral handling of payloads and remote procedure calls
- ◆ **WSDL** (Web Services Definition Language) for describing the public interfaces to services
- ◆ **J2EE** (Java 2 Enterprise Edition) standards, for interoperability, security, scalability, and platform independence

In addition, Composer’s Process Manager runtime engine utilizes key features of the proposed Web Services Flow Language (WSFL) standard.

How Does a Process Differ from a Service?

Processes are dynamic, stateful systems characterized by a rules-driven flow of data between participating activities. From an input-output point of view, a process receives input data, transforms and/or augments that data, and produces output data, much like any other service. And in fact, if the process is exposed as a Web Service (described by WSDL), it looks to the world like any other Web Service.

What makes a process different from a conventional Web Service is that it ties together—and orchestrates the flow of control and data between—relatively large units of work to accomplish a particular business function. A process is, in this sense, a *meta-service* that directs the interaction of other services (including, potentially, services that are external to the organization).

Some of the important differences between a conventional Composer service and a *process* are summarized in the table below (and discussed in further detail in the sections to follow).

Conventional Service	Process
Short duration	Long-running
Performance is important	Rapid execution typically not as important
Execution depends on the server being “up”	Processes that rely on external services can continue executing while a server is down
Serial execution of logic. Relatively little reliance on asynchronous processes	Asynchronous processing and parallel execution of activities are commonplace
Few opportunities for unexpected data overwrite	Multiple activity outputs can map to the same target messages (or message parts). Hence, overwrite is a potential concern, and policies to deal with “who writes where and when” must be defined explicitly
Control-flow stoppages are handled as exceptions	Flow may “route around” blockage points, in some cases. In other cases, timeout/retry policies may kick in
Data flow and control flow are tightly coupled	Data and control are less tightly coupled
No “sleep” mechanism	A long-running process may go to sleep during idle periods. Over the process’s lifetime, it may go to sleep and wake up many times
Straightforward testing requirements	Control-flow paths may be too numerous to test; extensive coordination with business partners may be required
Administration centers on performance tuning and configuration issues	Administration centers on lifecycle events, status monitoring

Large versus Small Units of Work

The units of work in a *process* are relatively large. (They encompass whole applications or services.) Likewise, operations on data tend to be conducted at a coarse (rather than fine) scale, occurring at the level of whole documents or document aggregates, rather than at the level of, say, nodes or nodesets. Fine-scale data manipulations occur inside the activities that make up the process.

Long-Running versus Straight-Through

One key distinguishing characteristic of a *process* (as opposed to an ordinary application or service) is that it is typically *long-running*. This means the process could have an execution time measured in hours, days, or even weeks, due to reliance on partner interactions, scheduled batch operations, human intervention at various levels, etc. For example, a process that obtains bids from contractors might very well require weeks to run to completion, whereas a credit-check application is expected to execute quickly, in real time. The credit-check task is best implemented as a discrete, standalone app: It processes information in straight-through fashion while the caller waits for a reply. By contrast, an RFP process involving (potentially) dozens of bidders, each with its own internal procedures and constraints, constitutes a large-scale, long-running process, which might be difficult or impossible to implement robustly as a monolithic, self-contained Web Service.

Wait States and Persistence

Persistence of state information is important for automated processes not only from a recovery standpoint but for efficient use of resources. A long-running process has to be able to deal with suspensions and resumptions of service, whether brought about administratively or through hardware downtime (scheduled or otherwise).

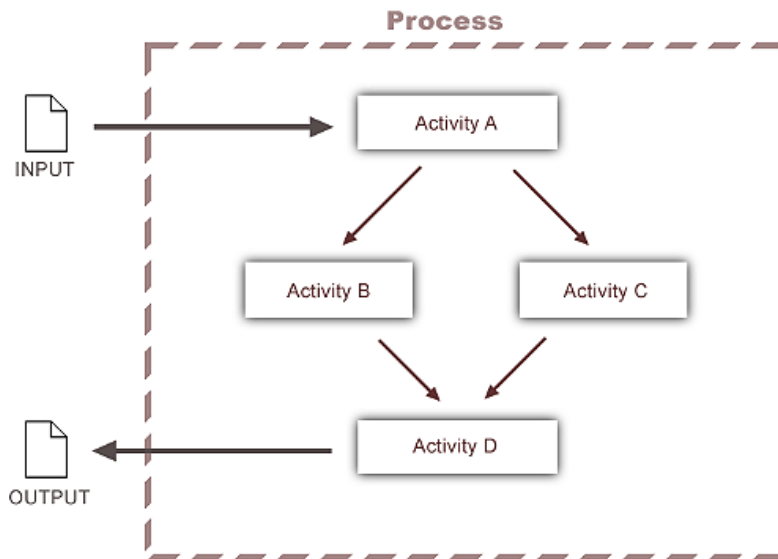
Composer's Process Manager persists process-instance info to database storage, so that processes can be put to sleep as needed and woken up again in response to appropriate events (such as the arrival of data from a just-finished activity), thus freeing valuable RAM and CPU resources during long waits.

NOTE: State info is persisted at every stage of process execution (not just when a process goes to sleep), so that a server restart, for example, is not disruptive.

Parallel Execution

Processes typically involve more than simple "straight-through" processing. If each task in a process requires, say, three days to complete, a straight-through execution chain would mean that the process could require nine days to run. This could be very inefficient if the tasks are not directly dependent on one another. Splits and merges (parallel execution and resynchronization) are a common feature, therefore, of process control flows.

These concepts become clearer with an example. The following figure depicts a process that relies on parallel execution of tasks.



In this example, an incoming request (which could be a SOAP request, form data received via HTTP POST, etc.) triggers a process instance to handle the request. Activity A performs the initial processing, then calls two more activities (B and C) to do additional processing. The output of Activities B and C form the input to Activity D. Finally, the latter sends output to the requester.

It's important to note that this diagram could cover a wide range of scenarios. For example:

- ◆ The request might come from inside the firewall or outside; and it could invoke the process *asynchronously*, or wait for the process to return (synchronous execution).
- ◆ The process might or might not be designed to send output to the original requester. The output might actually be directed elsewhere.
- ◆ Activity A might call B and/or C asynchronously, or synchronously.
- ◆ The process might be designed such that if Activity B does not respond within a given timeout period, a retry will occur.
- ◆ Activities B and C could be Web Services operated by remote business partners.
- ◆ Activity D might be designed to execute as soon as B or C finishes (whichever is first), or it might be required to block until both B and C have delivered data. In the latter case, D might be required to choose data from B *or* C, but not both.
- ◆ Any of the four activities shown could be processes in their own right. Or they could be Web Services, or Composer components; or any combination of the above.

Process Management Terms and Concepts

To be productive quickly with Composer's Process Manager, it's important that you understand certain key terms and concepts. This section explains the terms and concepts you'll most need to know when working with the Process Manager.

Note that most of the process automation idioms discussed below—as implemented by Composer's Process Manager—derive directly from the Web Services Flow Language (WSFL). For a more rigorous explanation of key terms, consult the WSFL specification at:

<http://www-4.ibm.com/software/solutions/webservices/pdf/WSFL.pdf>

Many of the fundamental concepts behind Composer's vision of process automation stem, also, from WSDL (the Web Services Description Language). For a detailed discussion of WSDL, see

<http://www.w3.org/TR/wsd1>

NOTE: The more you understand WSDL and Web Services, the easier it will be for you to understand Composer's vision of business process automation.

Activities, Messages, and Links

Composer implements processes in terms of activities and links. *Activities* are the units of work that carry out the steps of the process; they may be Web Services, applications, or other processes. *Links* establish the possible control-flow paths between activities. Data moves through the process model via *messages* that are passed (in whole or in part) from one activity to another.

Activities

Activities represent the fundamental unit of work inside an automated process. In a Process Manager process model, an activity can be:

- ◆ A Composer component
- ◆ A Web Service
- ◆ Another process

NOTE: The notion of a Web Service puts no restrictions on implementations. A Web Service can be implemented in any language, on any kind of platform, as long as it has an interface that can be described in WSDL.

In some cases, the components and services that you intend to use in a process model will have already been deployed on your server (or might already exist on the Web somewhere). That is to say, a process might simply “wire together” preexisting services. In other cases, you will develop components or services from scratch in order to meet process requirements. (Obviously, you can't test or *deploy* the process until all activities have been fully implemented.)

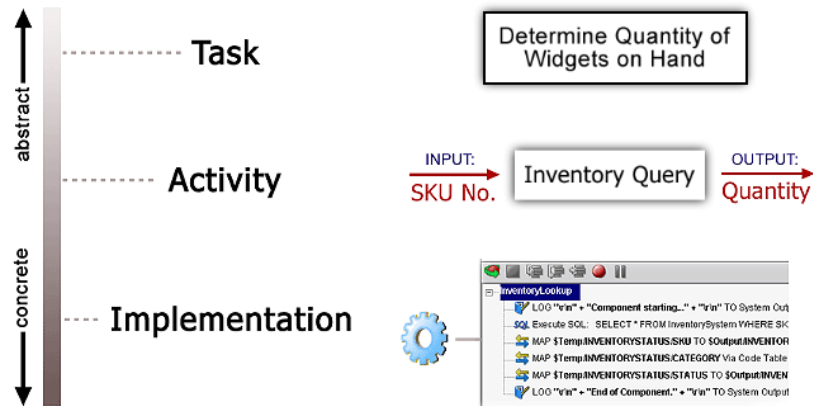
Start Activities and End Activities

The trigger for a process provides data that can be mapped (as messages) to one or more *start activities*. The final activities of a process model are said to be *end activities*. (There can be more than one start activity to a workflow; and also more than one end activity.) Start and end activities are like any other activities, except that a start activity has no incoming control link and an exit activity has no outgoing control link.

It's possible that some start activities might not have any outgoing control links. For example, one of a process model's startup activities might be a JMS messaging component (built using Composer's JMS Connect) that sends notifications to various queues asynchronously. If no downstream activities depend on data from the messaging activity, the messaging activity becomes both a start *and* an end activity.

Tasks, Activities, and Implementations

Composer's vision of process modelling makes a subtle but important distinction between tasks, activities, and implementations.



At the most abstract level, a *task* is a business function (which, in the real world, might go by a name like “Obtain Payment History,” “Issue Purchase Order,” or “Determine Back-Order Status”). Some business tasks are performed by humans; others are automated. Twentieth-century enterprise computing was concerned mainly with finding ways to automate or semi-automate business tasks.

Activities *carry out* tasks. The word “activity” implies a task that can be realized in software, *but it does not imply anything about the actual implementation*. One might know in advance what an activity’s required inputs and outputs will be, but this simply means that the activity’s *interface* needs are known. It doesn’t mean underlying implementation details are known.

Implementations of activities can take many forms. This is the key intuition behind Web Services: Collaboration depends on interfaces, not implementations. Composer’s Process Manager takes full advantage of the Web Services model, allowing any service that has a WSDL-described interface to be used as an activity in a process. No restrictions are put on how activities are implemented. An activity might be a C++ program running on a business partner’s web server, or it could be a custom EJB running on your own server, or a Composer JDBC component, etc.

Messages

Activities operate on *messages* which, in turn, are composed of logical parts. The parts have *name* and *type* characteristics as defined in a schema. (The types can be canonical XSD data types, *or* complex types of your own, defined in custom schemas.) You can think of message parts as corresponding to the input and output data for activities.

If you are accustomed to thinking of service inputs and outputs as XML documents or DOMs, the message metaphor simply extends the input-doc/output-doc idiom to include the notion of collaboration between participants. A message implies an *interface*—a predefined set of operations on specific kinds of data. The key intuition is that a message, far from being a static container of data, carries with it an implicit operational semantic arising from the fact that messages and their parts can be *named* and therefore designated in *operations* associated with an interface.

NOTE: The concept of “data as message” is fundamental in WSDL. If you are not familiar with Web Service Description Language concepts involving messages, message parts, types, etc., you will find it helpful to read the WSDL spec.

The message metaphor is extremely powerful, because it is concrete enough, in practice, to allow applications to specify their interface needs (and so make interoperability possible), yet abstract enough to keep participants from having to know anything about their respective implementation details. This means that applications can be developed entirely independent of one another, in different times and places, by different programmers, yet interoperate with one another as the need arises.

By exploiting the message metaphor, Composer’s Process Manager achieves the goal of keeping interfaces and implementations isolated, for maximum flexibility in “wiring activities together.”

Message Parts

Composer’s Process Manager “understands” the message-part semantics of WSDL-described services, in cases where activities are associated with WSDL.

Where activities consist of regular Composer components, message parts needn’t be explicitly defined in a schema. Component Input and Output DOMs are treated (by default) as messages.

Links

Links define the allowable control-flow paths in a process model. Unless an activity is a *start activity* or an *end activity* (see below), it will have one or more incoming links and zero or more outgoing links.

NOTE: The mere existence of a link does not mean that the link will actually be followed at runtime. Transition-condition logic actually determines this. (See discussion to follow.)

In an operational sense, links tell the Process Manager runtime engine “what to do next” when an activity finishes.

Links also provide a convenient metaphor for visualizing control flow between activities in a design-time environment, since links can be drawn as lines or arrows connecting boxes or icons that represent activities.

Sequencing, Timing, and Process-Level Logic

A process is more than just a collection of links and activities. The links in a process model are like the roads in a highway system: They define all the possible paths that can be traversed, but not *how they will actually be* traversed. In the real world, the pattern of traffic flow through a road system is affected by traffic laws, clearance limits on overpasses, etc. Likewise, the flow of execution through a process model is dependent on various designed-in rules and constraints that apply at runtime.

Factors that affect runtime flow patterns include:

- ◆ **Link-traversal logic**—Rules applied at the level of *link transition conditions*. (See below.)
- ◆ **Synchronization logic**—Rules that govern the triggering of activities that have more than one incoming link. In some cases, a “join” activity will want all potential input activities to finish executing before the join is evaluated. In other cases, the target activity may be designed to begin executing as soon as the first input (from any incoming activity) arrives.
- ◆ **Retry and timeout policies**—Some business interactions are required to adhere to elaborate try/timeout/retry requirements. For example: “Query this vendor and wait a maximum of two hours for acknowledgement. Query again, up to a total of three times.” Every activity can have (but doesn’t *have* to have) a timeout/retry policy.

These and other flow control factors are discussed in the sections immediately below.

Control Flow Logic

Control flow is mediated by logic that you can apply at three key points in a process: link transition conditions, activity exit conditions, and join conditions.

Link Transition Conditions

The logic that determines whether or not a given link is actually traversed at runtime is called a *transition condition*. The transition logic returns a *boolean* value based, typically, on inspection of the data coming into the link. If the transition condition evaluates to true, the link is traversed; otherwise it is not.

Note that links are not *required* to have transition logic. By default, a link is traversed straight-through.

In the example shown previously (see Figure), the arrows between Activities constitute *links*. Each link could have an associated transition condition (expressed via XPath). Data at Activity A might or might not trigger Activity B depending on (for example) the type of data received or particular values contained in the data.

Activity Exit Conditions

Every activity can have an *exit condition*. The exit condition is a logical expression that yields a boolean value. That value signifies whether or not the associated activity completed normally. If the exit condition evaluates to *true* at runtime, the outgoing link(s) can be followed. If it is false, the original activity will be reexecuted; but outgoing links will not be followed. (If an activity has no outgoing links, there is no exit condition.)

Join Conditions

When two or more links meet at the same target activity, logic needs to be applied in order for execution to continue. This logic takes the form of a *join condition* in conjunction with a map policy.

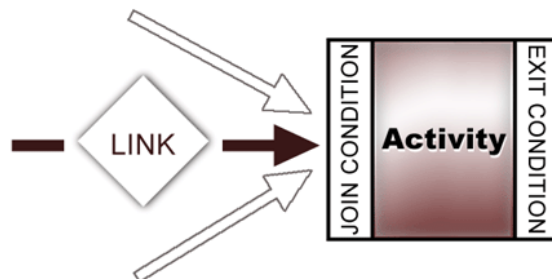
The join condition is an expression that returns true or false based on examination of the truth values of incoming links. (The truth value is the final value of the link condition.)

NOTE: While exit and link conditions are expressed in XPath, join conditions are specified in straightforward fashion using AND, OR, NOT, and parentheses (for grouping).

When an activity has a join condition, the join logic is consulted to determine how to proceed. Consider the following scenarios.

- ◆ Bids have been solicited from three suppliers. Company policy requires that bids must be received from *all three* suppliers before the rest of the process can be undertaken. The join condition specifies a logical-AND between link values. This pattern is called an AND-join.
- ◆ A company allows each of its employees to choose between two retirement plans. Each plan has associated with it an activity that generates appropriate paperwork for the employee. The paperwork contains data that will be passed to a join activity. The join condition specifies something like:
(Plan1 AND NOT Plan2) OR (NOT Plan1 AND Plan2)
This is an exclusive-OR (i.e., XOR) join.
- ◆ An activity receives input from any of several links. Any or all of the inputs can be used. This pattern is an OR-join.

The easiest way to visualize the relationship between the various kinds of flow logic is to think of the join condition as the “input-side” logic of an activity and the exit condition as the “output-side” logic.



The join condition exists for the primary purpose of implementing *synchronization logic* of the OR/XOR/AND type. Data from one or more activities can be inspected and used as the basis for deciding whether the next activity executes or doesn't execute.

The exit condition is strictly a mechanism for determining whether the associated activity (once it finishes executing) has produced data suitable for use by the next activity (or activities). If an exit condition evaluates to *true*, it means the activity's data output met the minimum criteria for continuation to any outgoing links. *All* outgoing links will be followed if the exit condition is met. *No* links will be followed if it is not met.

Transition conditions determine whether the next activity can be entered at all, using output from the source activity. Since there is no way for a link to "know" anything about other link targets, transition logic tends to be relatively simple (in many cases merely defaulting to *true*).

NOTE: Conditional *branching* can be implemented at the link level. See the discussion under "Branch Logic" in the next chapter.

Deferred Mode versus Immediate Mode

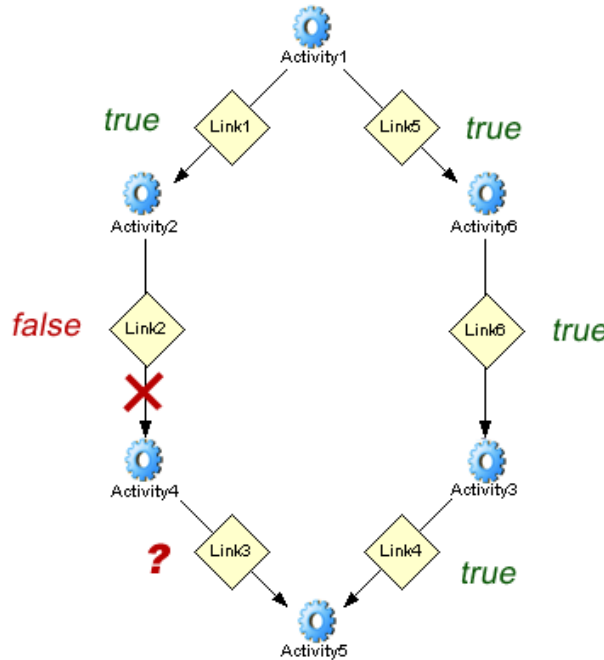
Joins can be fully synchronized (i.e., dependent on *all* source activities having finished executing), or *asynchronous* (allowing continuation as soon as data from any input activity arrives). By default, all joins occur in Deferred Mode, which means that all of a join's input activities must finish before the join condition can be evaluated. In this mode, a join condition will be evaluated exactly once.

For cases where the desired behavior is for a join activity to fire *prior* to the completion of all source activities, there is Immediate Mode. In this mode, the join condition is evaluated every time a source activity finishes. If there are multiple incoming links to a join, the join condition could be set to fire as soon as the first "true" link is known.

Composer Process Manager allows setting Deferred or Immediate Mode on an activity-by-activity basis.

Dead Links and Synchronization Failure

If a join condition is waiting on the truth value of an incoming link, but the link's condition is never evaluated (because flow was halted at some upstream point), the join will hang. Consider the following scenario:



In this flow graph, a join occurs at Activity5. In Deferred Mode, the join condition will not be evaluated until the truth conditions of Link3 and Link4 are both known. But assume that after Activity2 finishes normally, the link condition at Link2 evaluates to *false*. In that case, Activity4 will never fire; and if Activity4 never fires, Link3 will never be evaluated. (Link3 thereby becomes a *dead link*, and any segment of the flow graph that depends on it constitutes a *dead path*.) The net result is that the join at Activity 5 hangs.

To avoid this kind of synchronization failure, the Process Manager runtime engine performs a lookahead any time a condition expression evaluates to false. The lookahead is conducted as follows:

- Starting at the false link (or false join condition, as applicable), the engine traverses all downstream links until either a *join activity* or an *end activity* is reached, whichever occurs first. At this point, traversal stops.
- Each link on the traversal path is set to false.
- If the traversal path ends at a *join*, the engine determines whether the join condition can be evaluated (based on other link truth values and the join mode); if so, it is immediately evaluated with the incoming (dead) link having a value of *false*. Should the join condition then be *true*, the join is considered to be “alive” and no further dead-link traversal need occur. If the join is false—meaning that its outgoing links are dead—its status must be *set* to false, and the lookahead must continue downstream from that point.

This “dead-path elimination” procedure ensures that no false condition can cause a downstream join to hang. It is carried out automatically, as needed, by the runtime engine.

Map Policy and Data Merging

When multiple activities direct their output at a single activity, the potential exists for source activities to overwrite each other's data at the input to the target activity. A *map policy* specifies the mapping order and overwrite policy that will be followed for resolving conflicts.

There are three policy choices:

- ◆ **First writer wins (FWW)**—This means that the first data to be mapped into the activity's input template will be used as input to the join activity. Any subsequent messages cannot overwrite.
- ◆ **Last writer wins (LWW)**—The last data to arrive are mapped without regard to how any previous data were mapped.
- ◆ **Map Order**—Data mappings will occur in a user-specified order, without regard to time-of-arrival.

See “Map Policy” in the chapter called “Creating and Testing Processes” later in this guide.

Timeouts and Retries

An activity (whether it is part of a join or not) can have explicitly defined timeout/retry behavior. That is to say, if the activity doesn't produce usable output within a specified timeout period, a retry can be attempted, up to some maximum number of retries. Retries can be repeated at a user-specified interval.

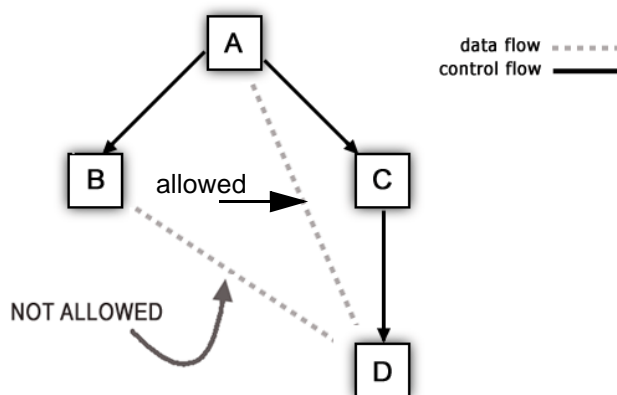
NOTE: Timeout/Retry behavior is available on a per-activity basis but is entirely optional.

Timeouts and retries are an important part of many standard business interactions and are formalized, in some cases, by industry standards, such as the Partner Interface Processes defined by RosettaNet. Composer's Process Designer allows timeout and retry options to be set on an activity-by-activity basis.

Data Flow Patterns

In automated business processes, as in human-mediated ones, the flow of data is coupled to the flow of control, but not always tightly. Some activities, for example, require data from outside the normal chain of control; the last activity to execute might need some piece of data from the first activity that executed, which could be many control links away. Other activities require data on the input side but have no “output data” *per se*. (The “output” of the activity might be physical goods loaded onto a truck.) There are many real-world situations in which data flow and control flow take different paths.

Composer's Process Manager allows data and control to follow their own paths, subject to one restriction: When an activity requires data from an activity to which it is not directly linked, the source activity must be reachable if one were to “swim upstream” (never downstream) through the control path, as shown in the following diagram.



In the example shown above, data can flow along the path from A to C to D, and it is also permissible for D to obtain data directly from A, because A lies on the (entirely one-way) path from A to C to D. But it is not permissible for D to obtain data from B. (The path from B to A to C to D involves first moving *upstream* from B to A, then *downstream* from A to C to D.)

It is not safe for D to obtain data from B, because the link topology does not ensure that B completes before D. One of the guarantees made by control links is that any activity on the “upstream” terminus of a link must execute *before* the activity on the downstream end can execute. In the above example, B might take three days to execute, but C might execute in a matter of seconds. The only safe way to get data from B to D is to create a control link between the two activities, thus making D a “join” activity.

Without getting too heavily into the details of data propagation, it should be mentioned in passing that data transfer (or mapping) across activities follows its own unique set of rules, distinct from the control-flow rules that Process Designer depicts with icons and arrows. (Flow graphs created in the Designer show *control* flow rather than data flow.) Data routing is easier to understand than control flow, but some unique twists apply; see the next chapter for details.

Lifecycle Events

A process can respond to any of several lifecycle events, so-called because they affect the *overall* execution of the process.

- ◆ **Spawn**—A *spawn* event invokes or instantiates a process in an *asynchronous* mode. The spawning agent does not want to wait (block) for the process to return, so after the process is activated the caller expects to return to whatever it was doing as soon as the spawned process returns a set of instance data (a “receipt”) to the caller, indicating that a unique process instance has been invoked successfully. The caller can, if necessary, later use this data to query the process for status updates, etc.
- ◆ **Call**—When an entity *calls* a process, the caller expects to receive data back from the process in real time (which is to say, synchronously). The call invokes the process, and the process runs to completion before returning. The output of the process is directed back at the caller.
- ◆ **Suspend**—A *suspend* stops, but does not destroy, an ongoing process. Control flow is temporarily interrupted. This type of lifecycle event typically occurs in an administrative context.
- ◆ **Resume**—The inverse of *suspend*. A process that was previously suspended continues operation. Again, this is an event of primarily administrative importance.
- ◆ **Inquire**—Queries a process for status information.
- ◆ **Terminate**—Aborts a process instance.

Process Manager Architectural Layers

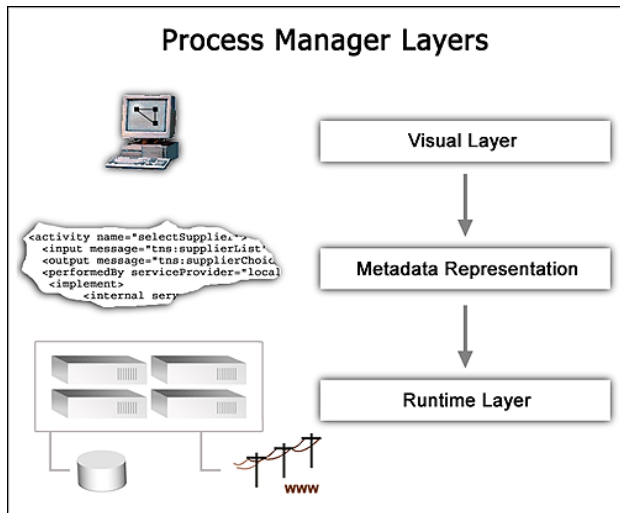
Processes you create using the eXtend Composer Process Manager are implemented and managed at three different levels.

Design Level: The design layer has the responsibility of managing the visual or user-interface representation of a process. This layer lets you define activities, connect activities via links, determine message mappings, and assign logic to transition points (links, joins, and exit conditions), using a rich set of visual design tools. The process model that you create here becomes the basis for the *metadata representation* of the process (see below) that the runtime engine uses for creating and managing process instances.

Metadata Level: At the non-visual level, a process model is stored in *metadata form* as an XML description of activities, links, input and output messages, etc. This metadata description provides all of the information needed to instantiate the process in a runtime environment. No presentation-related information is needed at this level.

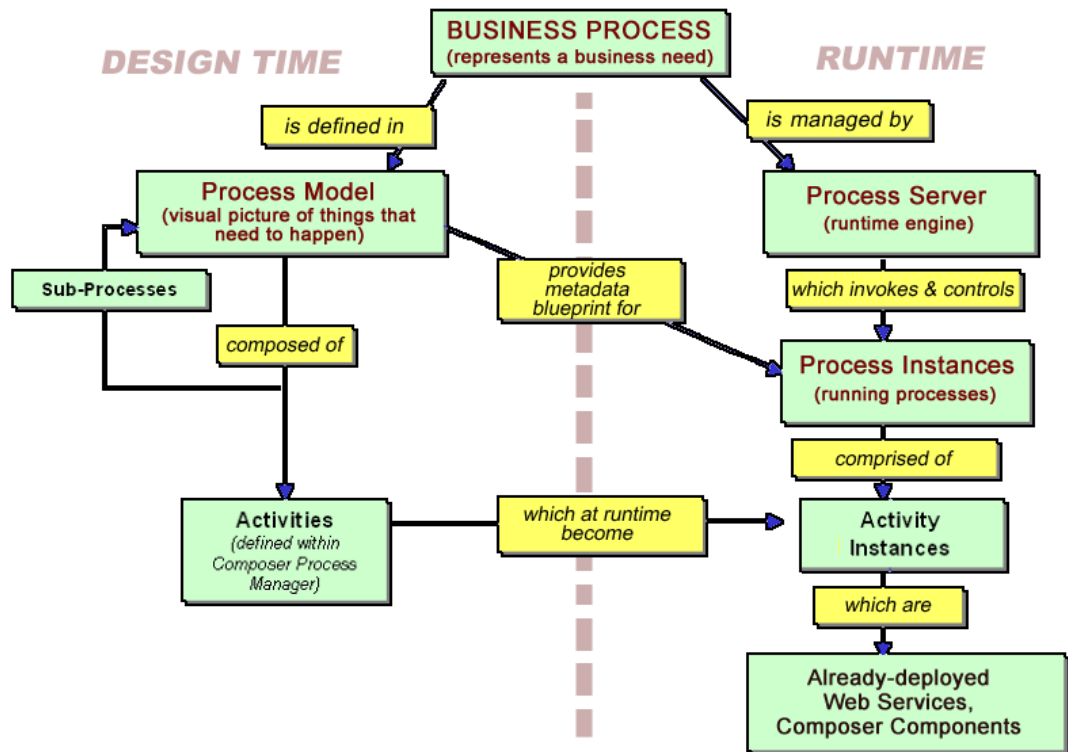
Runtime Level: The runtime layer manages the execution of process instances. It maintains state information, manages lifecycle events, implements timeout/retry behavior, mediates the flow of data and control between activities, and performs housekeeping tasks involving (among other things) persistence of instance data to a database. Administrative access to processes occurs via this layer.

The graphic below summarizes the relationship of these layers.



The layers drive each other from the top down (never from the bottom up). For example, the visual or design layer drives the creation of the process’s metadata representation, but the metadata layer never dictates a particular presentation. Likewise, the metadata layer sets the rules for the runtime layer, but the runtime engine never modifies the metadata; the metadata constitutes a blueprint of the process.

The design-time and runtime responsibilities of the layers (and their constituents) are shown in greater detail in the graphic below.



Notice that the process ultimately “sits on top of” and relies, for its concrete runtime implementation, on already-deployed services and components. (Some of these could be remote Web Services.) Deploying a process that uses prebuilt services can be likened to deploying a management framework whose sole job is to invoke existing applications according to special rules.

Existing applications might play roles in any number of processes. For example, there might be a Process A that uses Activities X, Y, and Z; and a Process B that uses Activities X, Z, and Q. If Activity Z is a Web Service with a public URI, it might actually play a role in a remote process in use at another company.

The plug-and-play nature of Web Services brings great power and flexibility to process management and is key to understanding how to use Composer’s Process Manager effectively.

Process Manager FAQ

By now, you are probably starting to have many questions about the ways in which processes can be modelled using Composer’s Process Manager and what the limitations are, if any, on process design. The answers to many questions will become clearer in subsequent chapters, but for now, here are a few quick answers to Frequently Asked Questions.

Can I Create or Edit Composer Components within Process Manager?

Yes. The Process Manager design-time editor runs entirely within Composer. You can have multiple components, services, and processes open at the same time and switch between windows freely. In fact, in animation mode, you can step over and into process activities, and if a given activity’s underlying implementation was built in Composer, you can *step into* the activity-implementation’s action model and step through it before returning to the process itself. You have the ability to debug action models and process models all in the same environment.

Can I Begin Designing a Process Even if Some Activities Have not Yet Been Implemented?

Yes. You can put placeholder activity icons on the process canvas and name them, draw links between them, etc., arbitrarily. To perform useful message-part mappings, of course, you will need to designate actual components or services for each activity, but even then, the components do not have to be completely built. If an activity is a Web Service, its mappings can be specified in the process model even before the service is built, so long as WSDL exists for the service.

Can I Run a Process in the Design-Time Environment for Test Purposes?

Yes. You can run a process within Composer, in animation mode, much the same way that you would execute a Composer Component in animation mode. *This is a unique capability among workflow and process automation tools.* In other workflow products, you may be able to create a “skeleton” process fairly quickly, but you usually cannot implement activity-layer functionality without leaving the design-time environment to do low-level programming; and when the activity layer has been implemented, you generally can’t test it in the original design environment. With Composer Process Manager, you can design, test, and debug activities *as well as processes* without leaving the design environment, dramatically shortening the development cycle.

Is It Possible to Import WSFL Flow Models Created in Another Environment?

No. Composer's Process Manager is not designed to import workflow models from other sources. WSFL is too immature a specification at this point to provide all the functionality required by users, and it's unlikely, therefore, that two vendors would implement two WSFL solutions in a compatible manner. In addition, the presentation (graphing) layer of Composer Process Manager is not directly driven by the metadata layer; in other words, no particular graphical representation of a process is inherent in a WSFL metadata model, and the Process Designer would not have any *a priori* notion of how to display your graph.

Can I Edit My Process-Model Metadata in an XML Editor?

You should never have to hand-edit the metadata descriptions of process models produced by Composer's Process Manager. Direct editing of the metadata is not recommended.

Does Process Manager Support Parallel Split, Exclusive Choice, and other Branching Constructs?

Yes. By allowing the designer to place boolean logic on the entry and exit sides of activities (in join and exit conditions) as well as on individual links, WSFL is able to accommodate arbitrarily complex flow patterns without having to define special-purpose constructs. So the short answer is that Composer Process Manager (following WSFL's lead) does not define special branch or join flow primitives. But you can easily achieve any desired branch/join behavior by means of appropriate transition conditions.

Does Process Manager Support Looping?

Yes, although backwards-facing links are *not* allowed. Links that connect downstream activities to source activities produce what's called a *cyclic graph*, which is not supported by WSFL because of the potential for reentrancy-related problems. (These problems are discussed more fully in the next chapter, along with the looping constructs actually supported by Process Manager.)

Can I Use the Process Manager for Document Routing and User Agent Functionality?

Queue-based workflows with human-facing activities can be created using Process Manager (see the "Advanced Topics" chapter). The concepts of queues containing work items, work-item priority, addressees (individuals) with roles, timeouts, locks, and administrative control over and browsing of queues are all supported by Process Manager. Also, the various actions that support these features are available for use across all component types (and all Component Editors) in Composer.

Will Automated Processes Put Huge Demands on My System?

No. The load and performance characteristics of a system running processes under Composer's Process Server are determined by the activities that make up the process. The Process Server *itself* incurs very little processing overhead because one instance of the Process Server controls any number of running processes. Also, since processes are typically long-running, it's usually the case that most of the pieces of an in-progress process instance spend the majority of their time in a sleep state. During these waiting periods, activities exist in persistent storage so that they do not actually consume CPU cycles.

Can I Start and Stop a Server While a Process is Running?

Yes, because process state information is persisted for each process instance on an ongoing basis. Also, processes are generally long-running and spend most of their time asleep. Suspension of a running process instance is supported by WSFL and by Process Manager. You can suspend any process at any time via the Process Server Console.

Must All Activities Be Implemented as Web Services?

No. Your activities can take the form of Composer Components *or* Web Services.

Must Processes be Exposed as Web Services?

No. They can be, but they don't have to be.

2

Preparing to Model a Process

This chapter attempts to make the abstract concepts of Chapter 1 more concrete by, first of all, examining runtime flow mechanics (as implemented by the Composer Process Server), then by showing how various use cases and design patterns can be implemented in Process Designer.

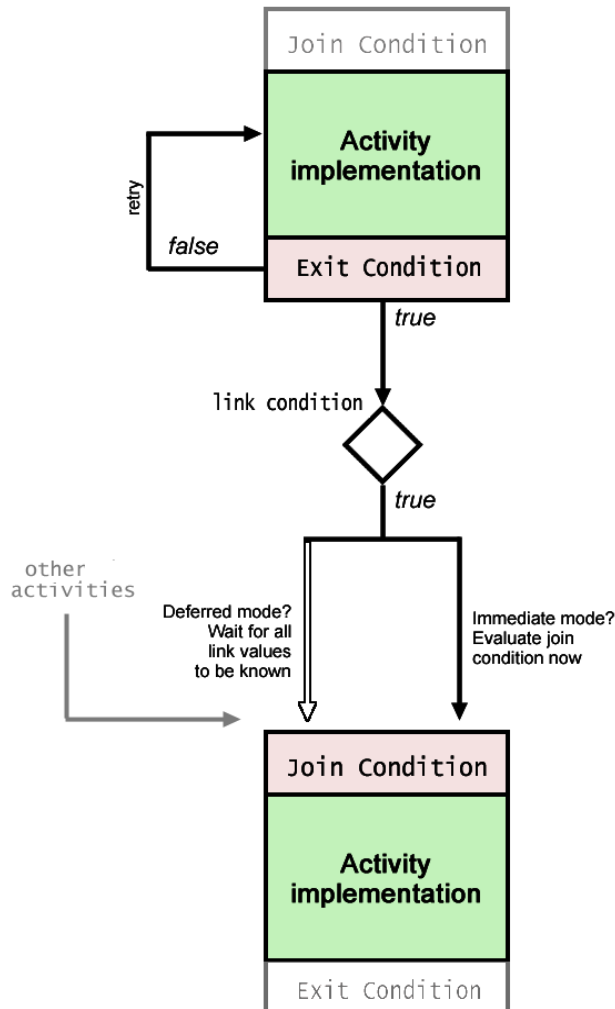
Process Server Execution Model

An understanding of the Process Server's basic execution algorithm is fundamental to understanding how to design a process.

The Process Server (or runtime engine) executes a process instance in the following manner.

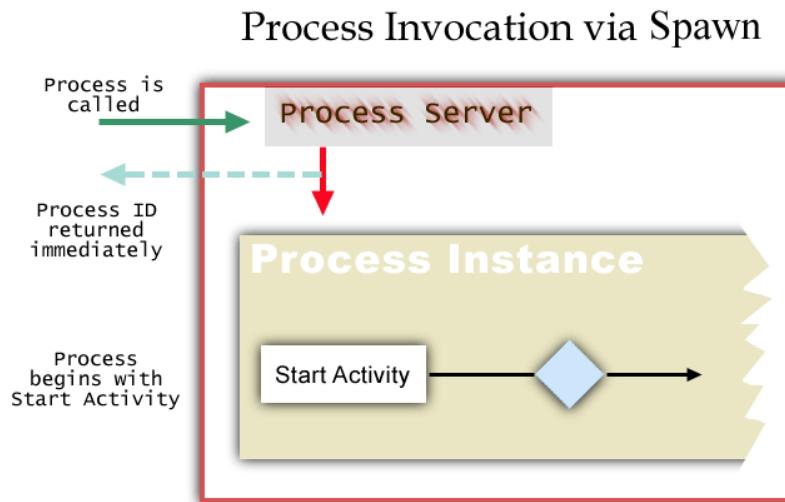
- 1 If the process was called asynchronously via a Spawn event, the Process server—upon instantiating a new process—returns a ProcessID (a “return receipt”) to the caller immediately. Otherwise, if the process was invoked with a Call, it is assumed that the caller will block until the process finishes.
- 2 The Process Server determines which of the process model's activities constitute *start activities*.
- 3 The input data to the process (one or more *message parts*) are mapped to the start activities.
- 4 Start activities are invoked.

- 5 Whenever any activity (whether it is a start activity or not) finishes, the Process Server consults the *exit condition* of the activity and evaluates the associated XPath expression. If the exit condition evaluates to *false*, the activity is executed again (with the same input as before). Execution repeats until a timeout occurs, or the exit condition is true, whichever occurs first. See diagram below.



- 6 If the exit condition of a completed activity is *true*, the Process Server determines which control links (if any) are connected to the outbound side of the activity, and the transition conditions of those control links are evaluated.
- 7 Data is mapped to the next activity (or activities).
- 8 For each control link whose transition condition was *true*, the Process Server evaluates the join condition of the link target. This evaluation takes place once, after all link conditions have been evaluated, if the join is in Deferred Mode (the default). If the join mode is Immediate, the join condition is evaluated multiple times: once each time a link's truth value has been computed. (In others words, as soon as a link condition has been evaluated—if even if the value is false—the engine will evaluate the join condition.)
- 9 If the join condition evaluates to true, the target activity fires; otherwise it does not.
NOTE: Regardless of synchronization mode (Immediate/Deferred), the target of a join *will not fire* until and unless the join condition is, at some point, true.
- 10 When the target activity of any link has finished executing, the cycle begins again at Step 5 above. Execution continues until there is nothing to do (i.e., the truth values of all end-activity exit conditions are known).

The following graphic shows typical process-startup mechanics for a process instance that has been invoked via *spawn*. (That is, the caller has elected to invoke the process in a “fire and forget” manner.)

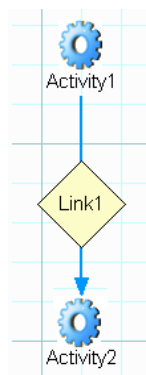


The Design-Time View

The runtime engine needs to know which activities a given process model will use, how they are linked together, what the data mappings are between them, etc. All of this information must be specified at design time in a process graph. You will use the Process Design to do this.

The Process Designer is a visual editing environment for creating graphical representations of processes, and for specifying data relationships between activities in a process. The tools that allow you to do this involve a combination of point-and-click *layout tools* plus text-based *property sheets*, which operate like non-modal dialogs. In addition to these GUI features (which are unique to the Process Designer), you have Composer’s standard menu commands, navigator frame, multi-document content frame, and output frame, just as you’d use when creating Composer components and services. In other words, the Process Designer runs entirely within Composer.

The Process Designer view of a simple two-activity process looks like:

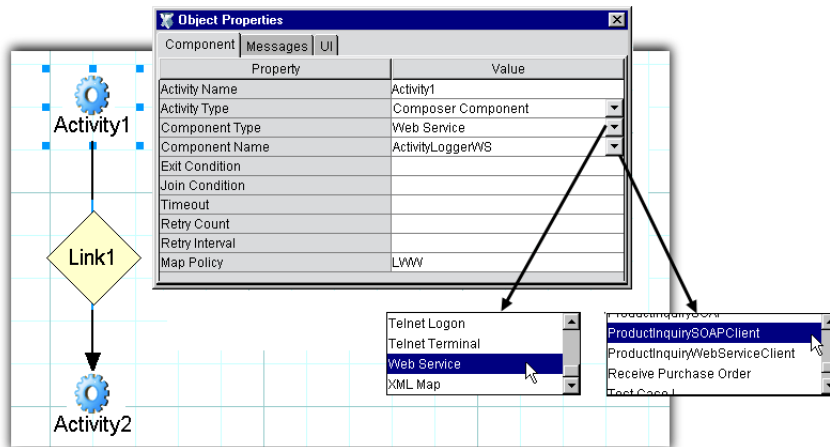


The activity icons, in this case, represent Composer Components. A link connects the two activities. The fact that the link icon is diamond-shaped means a custom transition condition has been specified for the link. (Links without custom conditions have no diamond icon and just show the word “Link”.)

We say that Activity1, in this diagram, is the Source Activity, whereas Activity2 is the Target Activity.

Just looking at this graph, it's not apparent whether Activity1 has a custom exit condition; whether a retry protocol applies to either activity; whether a mapping policy (such as Last Writer Wins) applies on the input to Activity2; and so on—to say nothing of how message parts are actually mapped from one activity to the other. The graph depicts control-flow relationships in a clear, direct, intuitive fashion, yet seems to hide data-related information.

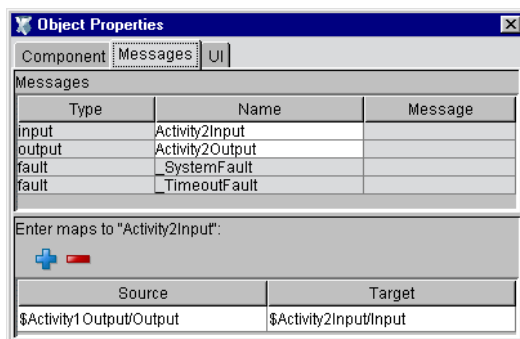
Data-link information is available via a non-modal (and dockable) Object Properties palette containing tabs for component-, message-, and UI-based information, as shown below.



Notice that in this graph, Activity1 has focus (as indicated by the handles around its periphery), and therefore the Object Properties panel (or “property sheet”) displays information appropriate to Activity1. If one were to give Activity2 focus (by clicking on it with the mouse), the Object Properties panel would update to show information specific to *that* activity. Likewise, clicking on Link1 would cause the panel to show information specific to the link. These panel updates happen in real time, automatically, so that information is available for any graph element at any time. The information is not simply read-only, however. The fields in the Object Properties panel are where you specify data-related and activity-level attribute values.

In the Component tab (shown here), you can view activity-level information: the activity name and type, the type of Component (in this case, a Web Service), the Component’s name, its Exit and Join conditions (if any), retry information, and map policy. Some of these values can be set using dropdown menus already populated with correct choices, as shown above. Others are text fields where you can type values directly into the panel.

By clicking on the Messages tab, you can view data-related information for the activity that has focus.



The top part of this panel shows the process-specific *name* for the activity’s input and output messages as well as the concrete Type and Message descriptions given in the WSDL for the service (i.e., the activity’s implementation). In other words, the Type and Message fields are automatically filled out with values taken from the WSDL *portType* segment.

The lower part of the panel is where you can specify exactly which Source message parts map to which Target message parts (using XPath). The above graphic applies to Activity2 in the previous flow graph and shows how input to Activity2 will be composed. The Source XPath, in this case, specifies that the Output message part from Activity1 will be mapped directly to Activity2's Input part. This means that when Activity2 fires, it will use as its input the Activity1 output. Obviously, this is a simple case. There could potentially be many intricate XPath mappings from Activity1's output message parts to Activity2's input parts.

The Object Properties panel will be discussed in greater detail later. For now, it's enough that you know that the Object Properties panel is where you can specify:

- ◆ Activity name
- ◆ Activity type (Web Service Send, Web Service Receive, Composer Component, Subprocess, or Synchronize Subprocesses)
- ◆ Exit condition for the activity
- ◆ Join condition for triggering the activity
- ◆ Timeout and retry settings
- ◆ Map policy (or overwrite policy) for situations where data from multiple incoming sources map to the same target message part(s)
- ◆ XPath-to-XPath mappings of data from source message parts to target message parts

Flow Control Strategies

Because the WSFL model attempts to “granulate” flow logic at the level of links and joins (rather than aggregating flow decision-making into higher-level constructs like “XOR-split”), it's not always obvious how one can specify conditional branches and other common control-flow patterns using a WSFL-based approach (as followed by Composer Process Manager). Nevertheless, it *is* possible to model virtually any kind of flow logic you can imagine using the Composer Process Designer.

This section looks at some of the more common flow idioms and how they can be implemented with the Process Designer.

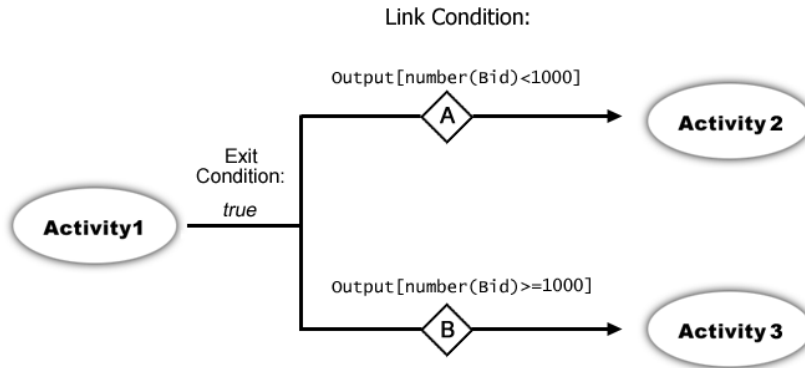
Branch Logic

Many workflow experts are accustomed to thinking in terms of branch logic as well as join logic. We will consider branching patterns in this section and join patterns in the next.

Conditional Branch (XOR-Split)

WSFL has no built-in notion of conditional branching *per se*, which means activities cannot, on their own, decide which link(s) to use when there is more than one link on the outbound side of an activity. Instead, the decision of which link to follow is determined by the *links themselves*. But no individual link can “know” what the transition conditions of other, parallel links might be. A link can only decide whether *its* path should be followed, based on the output of the previous activity.

Nevertheless, this flow-constraint mechanism is sufficient to model a conditional branch. For example:



In this scenario, Activity1 produces output containing bid information from a company. The link condition at A says that link A will be followed if /Bid is less than 1000. The condition at B says that link B will be followed if /Bid is greater than or equal to 1000. Clearly, if one link is followed, the other one will not be; so this represents an exclusive-OR split (or XOR split)—a conditional branch.

AND Split

The AND-split case (where *every* outgoing link is always followed) represents default behavior in Process Manager models.

An AND split, as defined here, is the case where every outgoing link *will fire* its target activity. This just means that every link has a value of *true*.

Non-Exclusive OR Split

There may be cases where an activity with multiple outgoing links could (depending on the output data) fire any number of target activities. For example, “Fire Activity 1 if such-and-so is true; also Fire Activity 2 if this-and-such is true; also Fire Activity 3 if so-and-so is true.” The number of activities that might actually fire at runtime could be zero, one, two, or three.

Again, this case is easily handled by distributed link logic. Every link can “look at” the source activity’s output and apply suitable XPath logic in order to arrive at a decision of whether to fire/not fire. In the end, the appropriate number of target activities fire.

Compound Branch Logic

It’s possible to combine the above patterns to handle complex cases such as “Traverse links A, B, and C always, but traverse D conditionally and traverse E only if D wasn’t followed.” To implement the case just stated, you would:

- ◆ Hard-wire links A, B, and C to *true*.
- ◆ Set a condition on D (using suitable XPath) of “if this node value is exactly such-and-such, fire the target activity.”
- ◆ Set a condition on E of “if [the same node that was tested in D] isn’t exactly such-and-such, fire the target.”

In pseudocode, the net result (in terms of the links that will fire their target activities) is:

(A AND B AND C) AND (D OR E)

Other complex cases are possible as well. But before getting involved too deeply in compound branching strategies, it’s important to step back and understand why such complex constructions are best avoided altogether.

In programming, complexity is a sign that a procedure or a block of code needs to be factored into smaller logical units. In Java code, conditionals involving many ANDs and ORs chained together are rare, because usually the desired actions can be carried out in a single switch/case block or a series of if/elses with simple conditions. If data dependencies are too complex to allow this, the dependencies themselves need to be broken out in such a way that the logic can be made simple. Data entanglements shouldn't be allowed to dictate tangled logic.

U.S. income-tax law provides many good examples of tangled logic involving complex data dependencies. The Internal Revenue Service must nevertheless produce tax forms that mere human beings can fill out correctly every year. They do this by, first of all, factoring out major dependencies into subject groupings, each with its own dedicated form (or "Schedule"). Within each form, there are major subdivisions (parts) that group calculations. The major parts are broken down, finally, into simple if/else statements. Some of the if/else statements point to other Schedules that must be completed before the if/else can be evaluated. (Each of *those* Schedules is a series of if/else statements grouped into parts.) Obviously, each tax form could, in theory, present all of its if/else logic in a *single compound expression* at the top of the form. But such a single-statement procedure wouldn't be human-readable.

Complex branch requirements should be a signal to you that the model you're creating needs to be factored into simpler logical units.

Join Logic

Link logic determines whether a target activity *can be* fired; not whether it actually will fire. The final decision as to whether an activity fires rests with the join condition.

In Deferred Mode (the default), no join condition can be evaluated—and therefore no join activity can be fired—until the *truth value* of every one of the activity's incoming links is known. When all link values are known, the join condition is evaluated. Only if the join condition is true can the target activity then fire.

In Immediate Mode, the join condition is evaluated every time a link's truth value is determined by the runtime engine. So if there are four inbound links to a target activity, it is possible that the activity's join condition will be evaluated four separate times. As soon as it is clear that the join condition is true (and can't change), the target activity is invoked.

NOTE: As explained in the previous chapter, join logic is the only logic touchpoint in the process model where XPath is *not* used. Links and exit conditions have access to upstream data and base their decision-making on XPath evaluations. The join condition knows only the truth values of incoming links; it does not use XPath and is not data-driven.

A join condition tends to look something like:

```
(Link1 OR Link2)
```

This is a simple non-exclusive OR condition. It means the join is true if one link *or* both links are true.

An exclusive-OR condition (i.e., the condition is true if one *and only one* link is true) would look like:

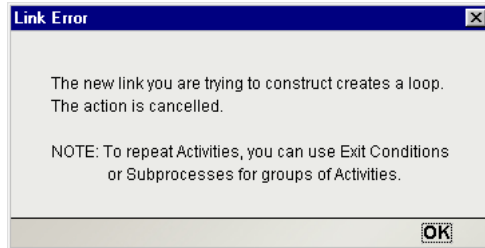
```
(Link1 AND NOT Link2) OR (NOT Link1 AND Link2)
```

In this case, *either* link could fire the activity, but if both links were true the activity would *not* fire. For this condition to work as intended, the join mode would have to be Deferred.

You can think of join conditions as a mechanism for deciding how many (and which precise combinations of) link values it takes to fire a target activity. This is important, because links have no way of knowing whether other (sibling) links exist, or which siblings evaluated to *true*. This knowledge exists only at the join.

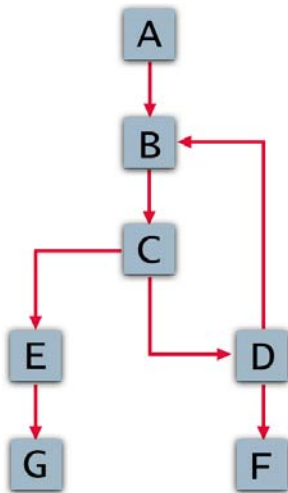
Looping

On occasion, you may find that you want to iterate on a given activity until a certain condition is met. For example, you may have some kind of batch operation to do. Your normal inclination might be to draw a link from a target back to one of its sources. But this kind of control flow (reentrant flow) is *not* allowed in WSFL, nor in the Process Designer. If you attempt to create a cyclic graph, you will see the following alert:



Iteration of an activity must be done by the implementation of the activity (or by the implementation of the calling activity) rather than at the process-logic level.

The reason looping is not allowed in a process model is that it opens the door to any number of ill-defined situations that would be difficult to manage. For example, consider the control-flow graph shown below, which has a loop-link from D back to B. The runtime engine must make some difficult decisions:



- B is a join node with input from two activities: A and D. In Deferred Mode, B must wait for *both* incoming links to have truth values before its join condition can evaluate. But since D requires the prior execution of B and C, it can't execute unless D executes. The model hangs while B waits for D to execute. The join at B would only work in Immediate mode.
- C will execute multiple times as part of the B-C-D loop. Each time it executes, the link from C to E is followed (in addition to the C-to-D link) and E will fire repeatedly if its link evaluates to true. Therefore E can unintentionally become part of the loop. To avoid this, the link logic for E's input link would have to "know" about the iteration status of the loop.
- If E is executed multiple times, it might trigger G multiple times; therefore, G also has to know about the loop. (And so on, for all activities downstream of E.)
- If the loop makes it back to C while before E returns from its first invocation, should a new instance of E be spawned?
- When D executes each time through the loop, should it fire F on every cycle?

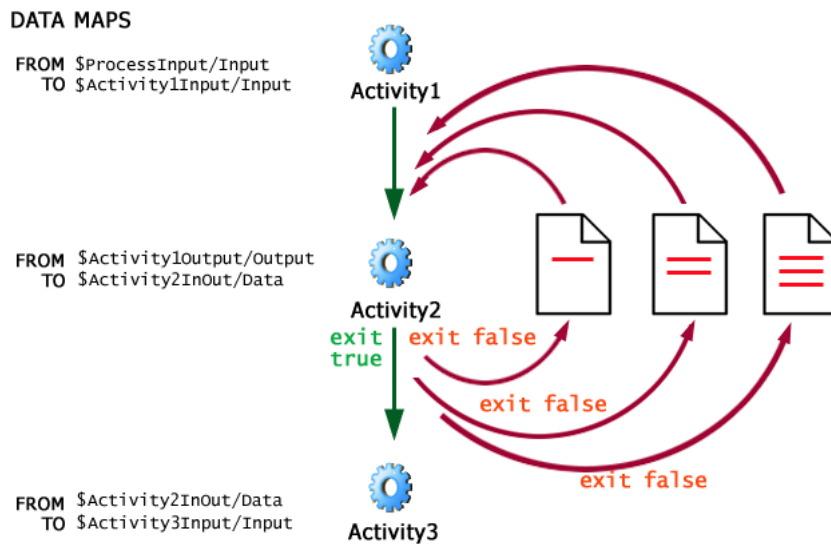
Even if these issues were resolved, the prospect of *testing* (then debugging) a model of the above sort could be daunting.

How Safe Looping Can Be Accomplished

A number of looping paradigms are possible in Process Manager. Some rely on WSFL's inherent retry-on-false-exit-condition mechanism; others delegate looping to activity implementations; and there is a special Process Manager activity to help with asynchronous fan-outs.

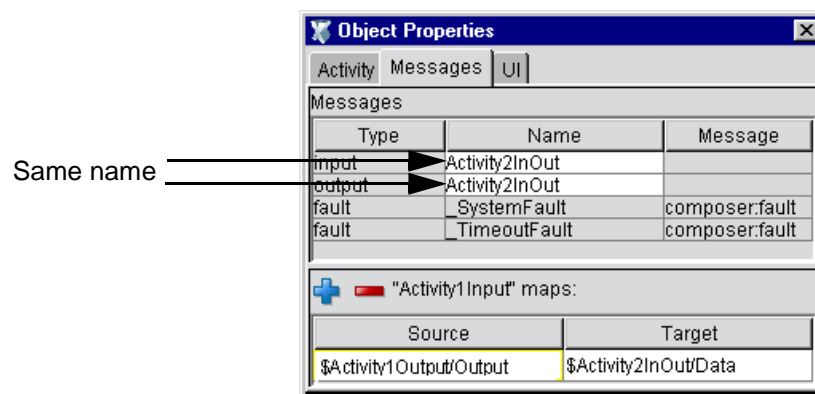
Mapping an Activity to Itself

While Process Manager does not permit control links to be used for looping, it does allow you to designate an activity's *output* as the data source to use for *input* in the event of a retry. This affords a type of looping, since standard WSFL and Process Manager behavior is to try an activity whenever the activity's exit condition is false. By mapping output back to input, the activity can loop on its own output as needed, until an exit condition of true is reached, at which point the looping stops and control proceeds down outgoing links. Diagrammatically, the scenario can be represented this way:



Activity2 has an input message named *Activity2InOut* and an output message of the same name. If Activity2 exits with an exit condition of *false*, it reexecutes using *Activity2InOut*. But *Activity2InOut*'s data was modified on the initial execution of the activity as part of a loop. (Perhaps new info from a database lookup was appended to the Data doc.) In any case, the exit condition on Activity2 might be an XPath expression that inspects a flag value in the output DOM. The flag value would signal the need either to iterate again or break out of the loop.

To implement this kind of mapping requires that you give the loop activity's output and input messages the same name, as shown here:



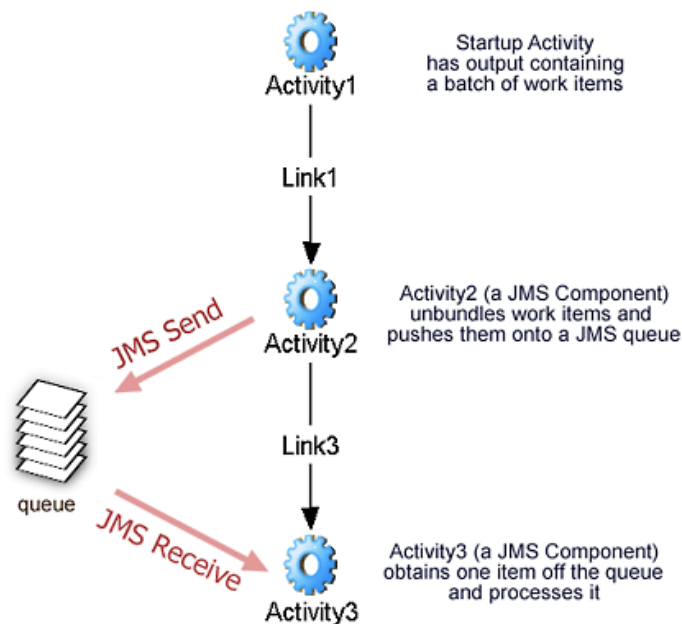
On the first invocation of Activity2, *Activity2InOut* gets populated with data from *Activity1Output/Output*. When Activity2 reexecutes due to a false exit condition, *Activity2InOut* (already populated with data) simply gets fed back in to the activity.

NOTE: When setting up this kind of mapping, do not forget to apply an exit condition (one that will successfully terminate the loop) to the loop activity. Otherwise, an infinite loop can result.

Iterating Against an External Data Store

The type of looping described above is useful when an output document containing loop results needs to be built incrementally, with new data added to output on each trip through the loop. But there are also times when work items merely need to be pulled off a queue and processed one at a time (one work item per trip through the loop), with no consolidation, *per se*, of data for a final output doc.

Imagine that a process has a start activity that produces, as output, a batch of work items. Each item needs to be processed *individually* by a given application designed for the purpose. This means the processing application must be invoked multiple times (once per work item). A possible process model is shown below.



In this scenario, Activities 2 and 3 are Composer JMS Components, but they could also be JDBC Components using a database instead of a queue; the concept can be adapted to other external stores as well. The idea is that Activity2 receives a batch of data (packaged as a WSDL message) as input. Activity2 unbundles (and possibly performs some kind of preprocessing on) the input. It also pushes every work item onto a message queue. Activity3 will inspect that queue.

In this example, the output from Activity2 contains a *JMSDestination* (in an element inside a message part) representing the location of the queue that Activity3 should operate against. No “work-item count” need be passed to Activity3. (Activity3 has been designed simply to fetch and process one work item.)

Activity3 does the following:

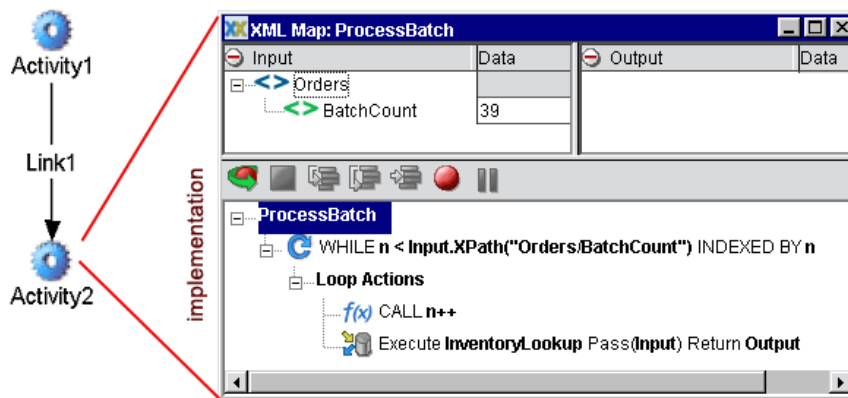
- 1 It executes exactly one JMS Receive action. The action either succeeds in pulling a waiting message off the queue or finds that the queue is empty.
- 2 If a message is successfully pulled off the queue, its data is processed and the activity exits with an exit condition of *false*, so that it executes again.
- 3 If no message was available (i.e., the *JMSMessageID* came back empty), the activity exits with condition of *true*.

The exit condition for Activity3 is based on whether the JMS Receive succeeded. If a message *was* processed, the exit is *false* so that Activity3 executes again. If *no* message was processed (i.e., the activity had nothing to do, because the queue was empty), the condition is *true* and Activity3 passes control to the next activity in the process flow.

This pattern does not involve a cyclic graph and does not violate WSFL’s restriction against backward links, because there is no control link that “points the wrong way”—no reentrancy. Repeated execution of Activity3 occurs because its *exit condition* is false until a certain criterion is met. The arrows labelled “JMS Send” and “JMS Receive” represent data flow outside the process model. The data do not enter into any message maps.

Delegating Loop Behavior to an Activity Implementation

An alternative to the foregoing strategy is to hide loop behavior within an activity’s implementation. For example, you could create a Composer service that uses Execute Component and Repeat While actions to call a given component repeatedly in a while loop. This strategy does not require the Process Server to manage any aspect of loop iteration.

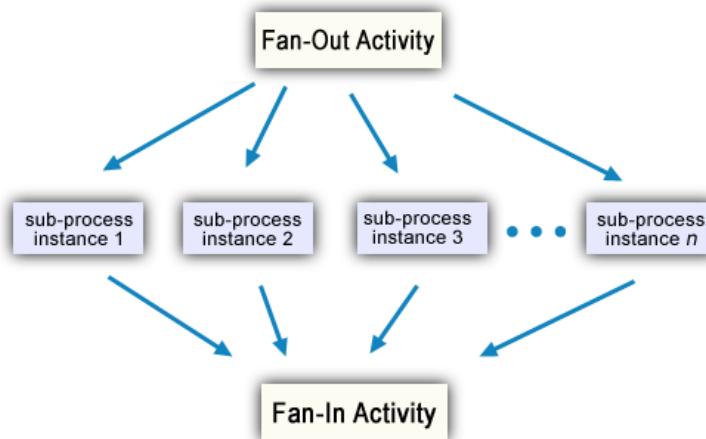


The flow graph is the same as before except that the Composer Component called *InventoryLookup* is called not by the process engine but by the action model of Activity2, as shown above.

Fan-Out

Rather than processing work items one at a time, you might find it advantageous to process them *in parallel*. Concurrent processing often results in significant performance gains.

The spawning of multiple concurrent processes is called a *fan-out*. The design pattern looks like this:



The Fan-Out Activity, on receiving a batch of work, unbundles the work items and spawns multiple instances of the appropriate target subprocess: one instance per work item. The subprocess might be called something like “DetermineQuantityOnHand” and the batch might be a collection of SKU numbers. One instance of *DetermineQuantityOnHand* is created for each SKU number.

Every subprocess instance is invoked via the WSFL-defined *spawn* mechanism, which means each instance runs in its own thread: i.e., parallel instances execute concurrently and finish whenever they happen to finish.

For this scenario to work, there has to be a “Fan-In” activity that collects output from every subprocess instance and waits until all instances are finished before passing control to the next activity in the process graph. The activity that does this is shown as the Consolidator activity in the above diagram.

As each subprocess instance finishes, it hands data to the merge component, which collects that data and merges it (typically) into one final document. When all subprocess instances have been accounted for, the merge component’s exit condition becomes true and control passes to the next point in the graph.

There are two problems with implementing this pattern in a flow graph. The first is that WSFL provides no native mechanism for creating arbitrary numbers of outbound links at runtime. The second is that, even supposing that links *could* be created in quantities known only at runtime, there is no native provision for specifying the kind of late-binding join logic that would be needed to handle the synchronization.

Fortunately, these problems can be overcome.

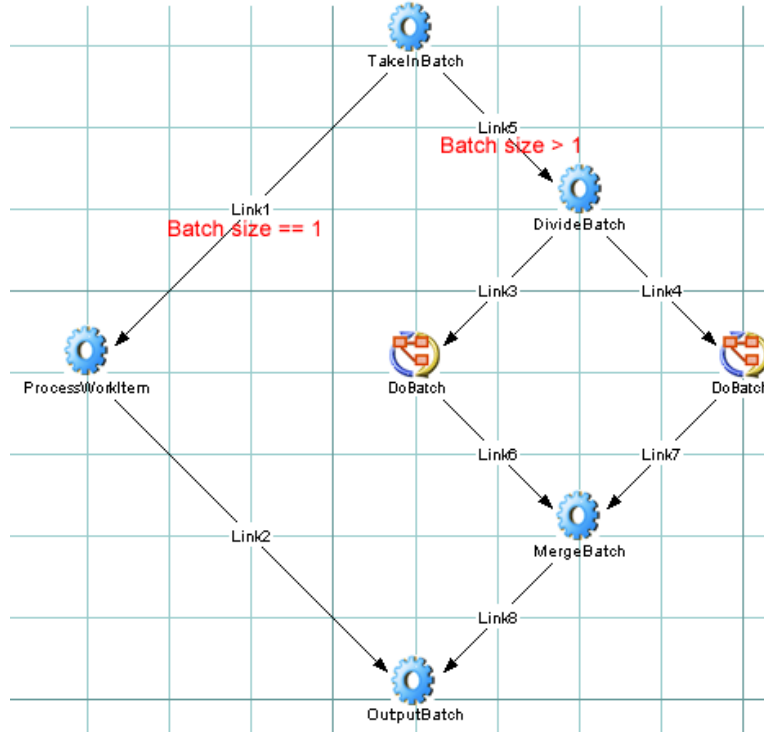
Component-Controlled Fan-Out/Fan-In

One strategy is to hide the fan-out in a component’s implementation. Imagine that a component’s action model contains a Repeat While loop that iterates over a batch, calling a Process Execute action on each work item. By specifying an execution method of “spawn” in the Process Execute action, every process is launched in its own thread, in fire-and-forget manner. The workhorse process instances (the “fanee processes”) can be designed in such a way that they post their results to a database, JMS message queue, or other external store. Synchronization would be handled by a second component (a “consolidator”). The “sync component” could follow either a listener metaphor or a periodic-polling metaphor. If the latter pattern is used, polling could take place in a continuous loop, or (less CPU-intensively) on a timed basis, where the component sleeps between data queries. The listener metaphor, on the other hand, can readily be implemented using a JMS Service.

Recursive Fan-Out/Fan-In

Fan-outs can be modelled in WSFL using a recursive graph. That is, a fan-out can be modelled as a process that *calls itself* until a suitable number of “fantee” activities has been invoked (at which point the results are accumulated via joins). Diagrammatically, a *recursive fan-out process* might look like this:

Process: "DoBatch"



The algorithm can be summarized as:

- 1 Take in a batch of work. If the “batch” contains more than one work item, split it into two smaller batches (that is, traverse Link5 shown above) and fire new instances of ourselves using the smaller batches as input. (That is, traverse Link3 and Link4 and fire two new instances of *DoBatch*.) This recursive invocation of new instances of *DoBatch* continues until incoming batches are no longer splittable.
- 2 If an incoming batch contains exactly *one* work item, traverse Link1. The target of Link1 is the component (*ProcessWorkItem*) that actually processes the work item.
- 3 The output of the *ProcessWorkItem* activity is handed to *OutputBatch*, which does any required post-processing and returns.
- 4 When a recursively called *DoBatch* instance returns, it traverses its outgoing link (Link6 or Link7, whichever applies). A deferred join then occurs at *MergeBatch*.
- 5 The *MergeBatch* component accumulates data arriving from Link6 and Link7 into one output message, which is sent to *OutputBatch*. The return/merge/return/merge cycle continues until every processed work item has been accumulated into one consolidated message (document).
- 6 Finally, the topmost instance of *DoBatch* returns the consolidated document.

Note that the link logic at the top of the graph effectively makes the split coming out of *TakeInBatch* an exclusive-OR split. (This also has the effect of making Link2 and Link8 mutually exclusive.) The algorithm is basically one of “split or work.” The *ProcessWorkItem* activity is not hit until the batch has been split into individual work items (at which point a corresponding number of instances of *ProcessWorkItem* fire up). The output docs are merged two-by-two, then four-by-four, etc., until the final output of the process is one consolidated document.

This is an example of a well-factored design that uses simple, discrete operations to accomplish a dynamically sized task. The flow can be diagrammed explicitly using ordinary WSFL constructs and hides nothing (other than business logic) in activity-level implementations. All data travels through ordinary data links (no special “off the graph” communication through external stores); all processing is concurrent; and all joins are synchronous.

Synchronize Subprocesses Activity

One of the native Activity types in Composer Process Manager is a specialized activity called *Synchronize Subprocesses*. This activity exists in order to provide “fan-in” capability (synchronization and consolidation of returns from asynchronous subprocesses).

A graph that uses the Synchronize Subprocesses activity will implement the following pattern:



Activity1 is a Composer Component that performs a fan-out by spawning multiple subprocess instances inside a Repeat While loop, as part of the component’s implementation. (This can be done via the Process Execute action, which has *spawn* as well as *call* modes.) Activity1 constitutes the fan-out. Activity2, the Synchronize Subprocesses activity (which has its own distinct icon), constitutes the fan-in.

When a subprocess is invoked via *spawn*, the Process Server returns a Flow Instance ID to the caller. Activity1 collects Flow Instance IDs for each subprocess that it invokes and passes the list of IDs to the Synchronize Subprocesses activity.

The Synchronize Subprocesses activity’s implementation consists of a Composer XML Map, JDBC, or other Component. This component receives, as input, the list of Flow Instance IDs mentioned above, plus a collation document. The latter is used at runtime to accumulate data returned by “fanee” subprocesses. The Process Server mechanics for this are described in a later chapter, but the key notion is that the fan-in component is notified (called by the runtime engine) each time a fanee returns, and the associated work-item data is added to the collation doc. When every fanee is accounted for, the component exits with a condition of true and its output (a completed batch of work) is mapped forward to the next activity or activities in the model.

Process Architecture in Review

Below is a brief recap of key concepts. You should keep these concepts in mind as you create models in Process Designer.

Activities can be of five types:

- ◆ Composer Component
- ◆ Subprocess
- ◆ Web Service Receive
- ◆ Web Service Send
- ◆ Synchronize Subprocesses

There are two flavors of Web Service activity. The Web Service *Send* type handles WSDL Solicit-Response and Notify patterns, whereas the Web Service *Receive* type handles WSDL Request-Response and One-Way operations.

Synchronize Subprocesses is a specialized type of activity (unique to Process Manager) that provides for synchronization of fan-outs.

A Subprocess is simply any Composer process that is being used as an activity inside a larger process. The use of processes as activities makes possible *hierarchical modelling* of business workflows.

A process model can coordinate the flow of data and control between a heterogeneous mix of local and offsite applications (including Web Services administered by business partners).

A Start Activity has no inbound links. An End Activity has no outgoing links. All other activities have one or more incoming links and zero or more outgoing links.

Data dependencies between activities are implemented by means of *data links*. In Process Designer, data links are not drawn by the user; they are created automatically when you map one activity's output message part(s) to another activity's input message part(s).

Time-order dependencies between activities are enforced by means of *control links*. A control link connects a source activity to a target activity. The link guarantees that the target cannot execute before the source does. A corollary of this is that cyclic graph patterns (where downstream activities have links pointing to upstream activities) is not allowed.

Synchronization of work is accomplished through *joins*.

Conditional flow of data is under the control of *link logic* (transition conditions).

Conditional triggering of an activity based on the completion of "feeder" activities is under the control of *join logic* (join conditions). In Deferred Mode, the join condition is not evaluated until the truth values of all incoming links are known. In Immediate Mode, the join condition evaluates every time a source link evaluates.

Data-overwrites can be controlled through the use of *map policies* (in cases where two source activities might target the same XPath location in the input to the next activity). The policy can be Last Writer Wins (arrival-order overwrite), First Writer Wins (first mapping is permanent; late data is ignored), and literal map-order.

Retry behavior is under the control of activity *exit conditions* and/or timeout values. If an exit condition is false, the activity reexecutes using the original input. The activity keeps reexecuting until the exit condition is true or a timeout occurs, whichever comes first.

Link and exit conditions must be specified using XPath. Join conditions cannot use XPath; they are specified via boolean expressions involving link truth values.

The Process Server is the runtime engine that manages the execution of process instances. It persists state information, instance data, etc. at all points in a process's life cycle.

Processes can be monitored and administered (suspended, resumed, etc.) via the Process Server Console.

WSFL defines (and Process Manager supports) lifecycle events *spawn*, *call*, *suspend*, *resume*, *enquire*, and *terminate*.

Suspend, *resume*, and *terminate* events can be controlled administratively via the Process Server Console. The *enquire* event (meant for status queries) is not labelled as such in the console; rather, complete status information is displayed in a Process Status view, available at any time. *Spawn* and *call* are under the control of a process's initiator, which might be a SOAP server responding to a request, a component that has executed a Process Execute action, etc.

Taking a Best-Practices Approach

The key feature of a WSFL-based process model is its reliance on units of work that know nothing about each other's implementation details, yet can interact cooperatively based on known interfaces. In this type of system, the units of work (activities) have no knowledge of—and *should need* no knowledge of—the context in which they are being used. Everything an activity needs to know is contained in the *input message* to the activity.

A good process model leverages this principle of separation of interfaces from implementations. This not only makes for efficient code reuse but opens the door to interoperability across technologies, platforms, partners, etc. It also greatly eases troubleshooting, testing, and maintenance.

Characteristics of a well-designed process model include:

- ◆ A well-factored activity layer. No single activity tries to “do too much.” No activity is monolithic in functional requirements.
- ◆ Every activity has been designed to run standalone, with no special knowledge of its neighbor activities.
- ◆ Every activity has clear-cut data-input needs and correspondingly clear data-output responsibilities.
- ◆ Activity-to-activity data dependencies are explicitly described in message mappings.
- ◆ Business logic is completely hidden inside activity implementations. No business logic is attempted in any message maps.

NOTE: Message mappings between activities should exhibit coarse granularity. Element-level transformations of the underlying XML (i.e., fine-granularity document manipulation) should be done inside activity implementations, not in the process model.

- ◆ The flow graph is easy to read and comprehend. If a graph starts to grow beyond a handful activities or joins, *consider factoring out related activities into subprocesses*. A model with dozens of splits, joins, activities, etc., may be extremely difficult to test or debug, whereas if the same model can be factored into three or four subprocesses, each with only three or four activities, the subprocesses can very likely be tested standalone, then combined into a final, unified model that is robust.

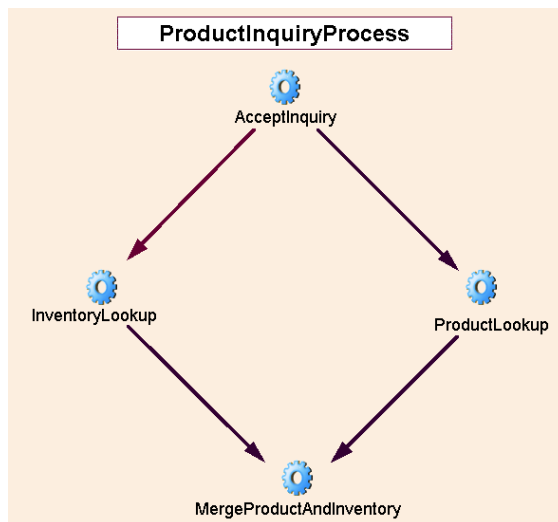
3 Creating and Testing Processes

In this chapter, we'll take a look at how to construct a process, specify data mappings, apply logic to links, control joins, and animate (step through or execute) a process in the design-time environment.

If this is your first time using the Process Designer, you should read this chapter before building your first process.

Example: A Simple Straight-Through Process

So that you can see how quickly a process model can be built and tested using the Composer Process Designer, we'll go step-by-step through the construction of a simple straight-through process, as represented by the graph shown below:



Description

The *ProductInquiryProcess* model handles a request for product information. The input to the process is an XML DOM containing a SKU (product ID) number. The output of the process is an XML DOM containing detailed information about the product in question. The needed product information is pulled from two sources (two databases) via JDBC.

Our process uses four activities, all of them Composer Components. The roles and responsibilities of the activities are as follows:

AcceptInquiry (XML Map Component)—Takes in an input DOM containing SKU information and simply writes that information straight to an output DOM, along with a tracking number.

InventoryLookup (JDBC Component)—Using the output of *AcceptInquiry*, this component performs a database lookup against an inventory control system to obtain Category and Status information about the product whose SKU number was passed in.

ProductLookup (JDBC Component)—Using the output of *AcceptInquiry*, this component performs a database lookup against a marketing database to obtain detailed product info, including price, color, text description, and so on.

MergeProductAndInventory (XML Map Component)—This component merges the incoming data from the two JDBC components. Its output constitutes the overall process output.

NOTE: If you are familiar with the Composer Tutorial, the above components (except for *AcceptInquiry*, which is specific to this process) are the same ones that are used in the Composer Tutorial.

Process-Building Basics

Until you have gained familiarity with the Process Designer, we recommend that you construct your first process models following the steps shown below.

➤ **To create a Process Model:**

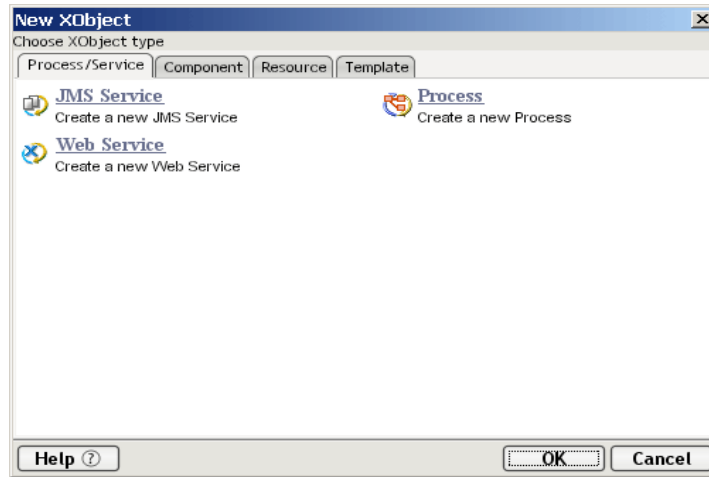
- 1 Create the new, blank process graph.
- 2 Optionally, create any Service Provider or Service Provider Type resources you may need.
- 3 Create and position all activity icons.
- 4 Connect the activities with links.
- 5 Create the message mappings between activities.
- 6 Specify any *link conditions* that might apply at various points in the process.
- 7 Specify any *exit conditions* that might apply.
- 8 Specify any *join conditions* that might apply.
- 9 Set any other attributes (Timeout/Retry values, Map Policy, etc.) that might apply to any activity in the process model.
- 10 Build, test, and debug all individual activity *implementations* (that is, the underlying Composer Components, Web Services, or Subprocesses that constitute the executables for the activities), if you have not already done so.
- 11 Finally, test the model.

Creating a New Process

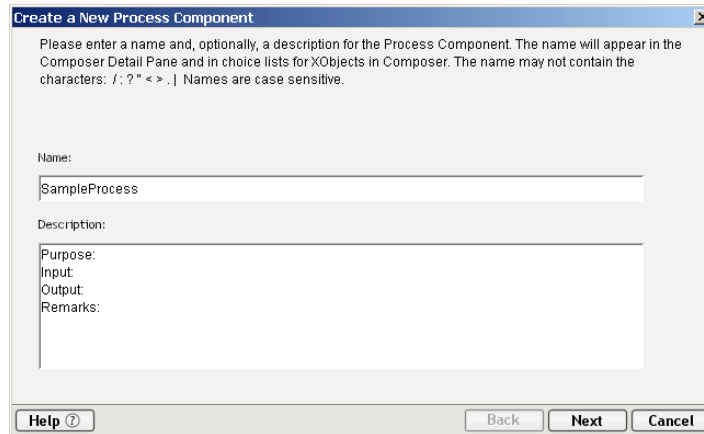
If you have created Composer Components and Services before, you will find the procedure for creating a new process quite familiar.

➤ **To create a new process:**

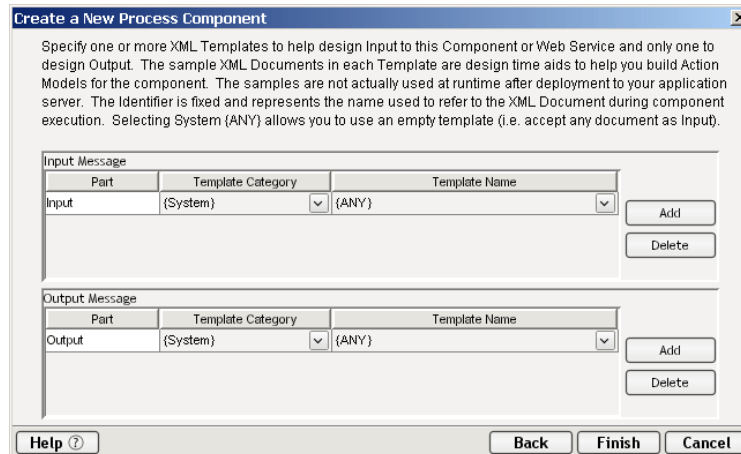
- 1 Launch Composer, if it is not already running.
- 2 From the **File** menu, select **New > xObject**, then open the **Process/Service** tab, as shown below and select **Process**.



A dialog appears, prompting you for a process name.



- 3 Enter a **Name** for the process. Optionally enter any additional descriptive info that you want to associate with this process.
- 4 :Click **Next**. A new dialog appears.

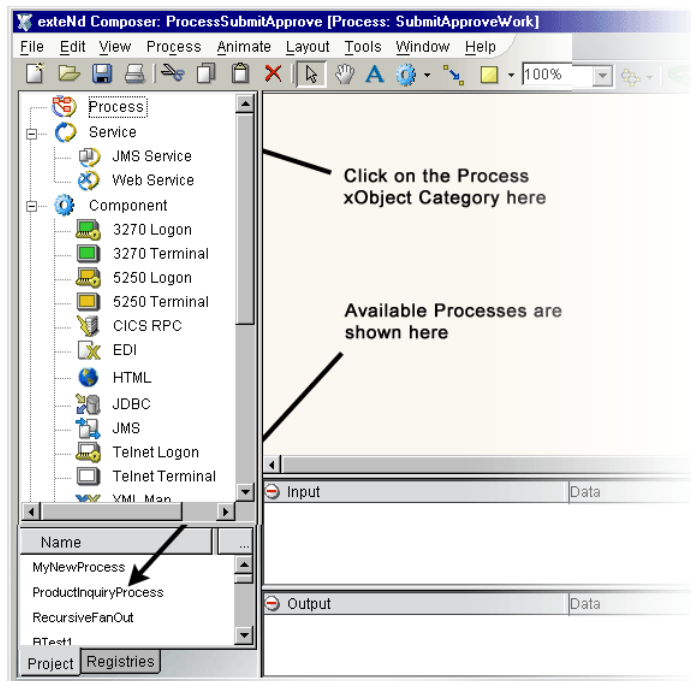


- As an aid to the design-time testing of your process, add whatever **XML Templates** you would like to use for process inputs and output. (These documents will not actually be used after deployment of the process to a server. They are design-time aids only.)

NOTE: XML Templates are used by almost every component type in Composer. If you are not familiar with the creation and use of templates, consult the chapter on XML Templates in the *Composer User's Guide*.

Usually, the template(s) you would specify here would be identical to the ones used by the start activity of your process (assuming that the start activity is a Composer component). In this case, you want a template, or templates, that are capable of providing sample input data to the process.

- Click **Finish**. The Process Designer window opens, with an empty canvas. See below.



About Service Provider Resources

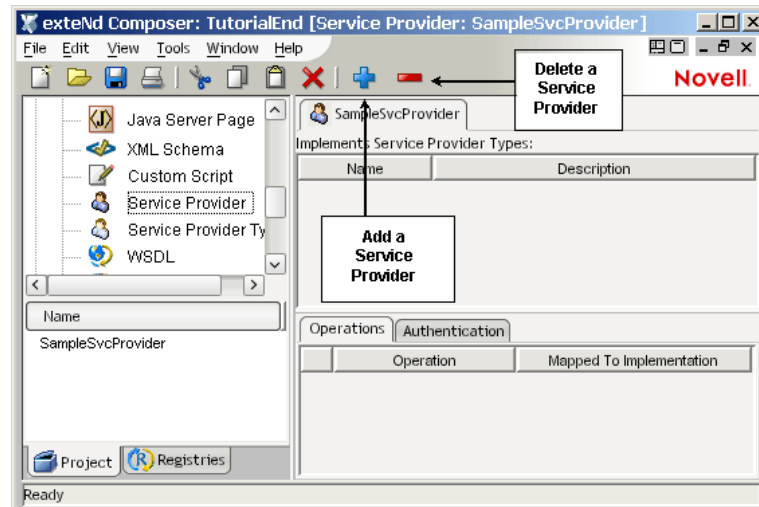
A Service Provider is the party responsible for performing a particular activity within a business process. Composer allows you to identify Service Providers for use with your process.

NOTE: You must have WSDL resources and Service Provider Types (see below) in place before creating new Service Provider Type Resources.

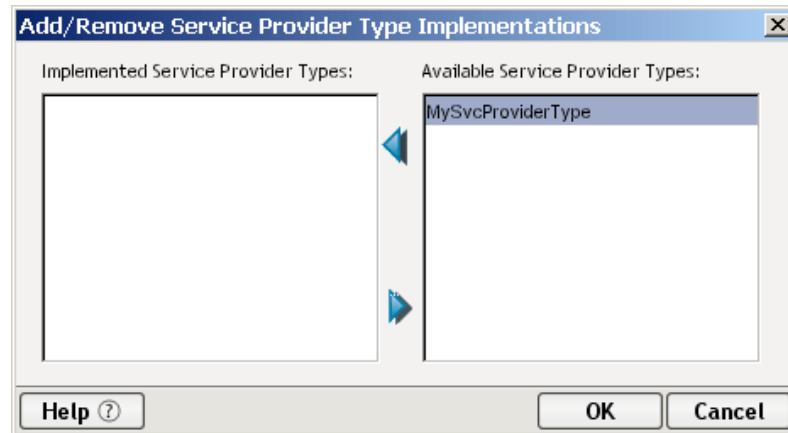
➤ To identify a new Service Provider:

- From Composer's **File** menu, select **New**, then **xObject**. From the **Resource** tab, select **Service Provider**.
 - ◆ or
 Right-click on the **Service Provider** Resource icon in the Category pane, and choose **New**.
- Provide a name and, optionally, a description to identify the Service Provider.

- 3 Click **Finish** to create the Service Provider and open it in the Editor pane.

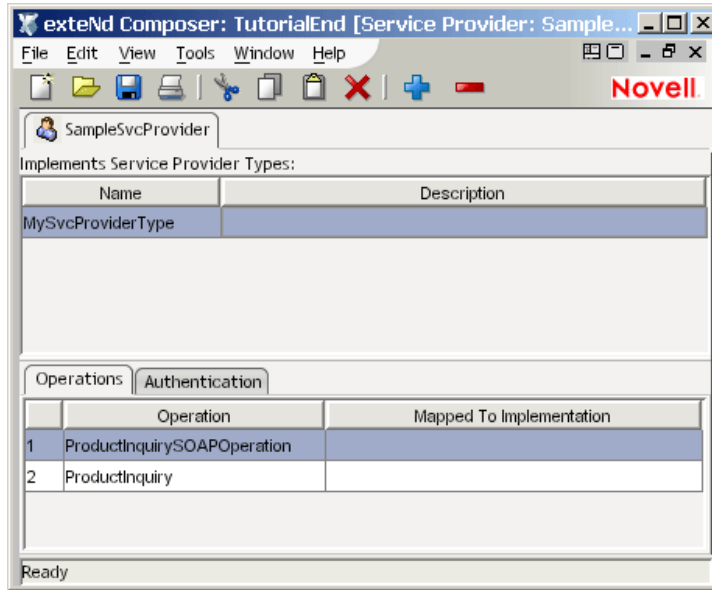


- 4 Click on Add to create a new Service Provider, which causes the Service Provider Type Selector to appear.

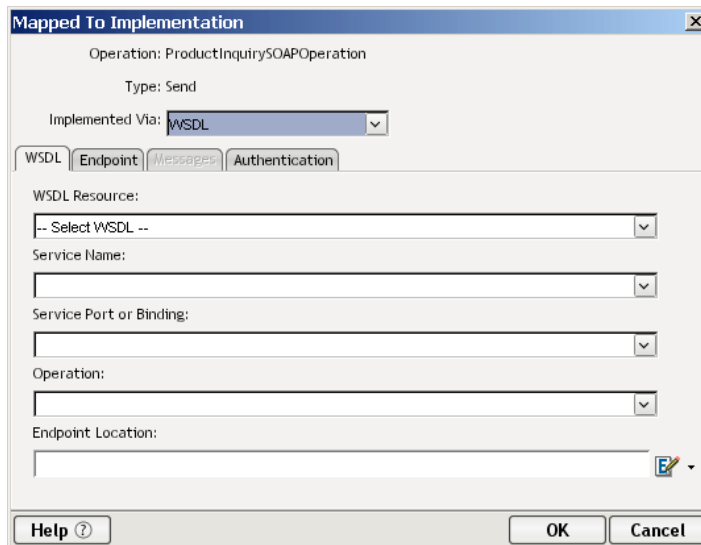


- 5 Use the right and left arrows to select or deselect the appropriate Service Provider Types and click on **OK** when your selection is complete.

- This will return you to the Editor pane, where the Service Provider Type will now appear, and the Name and Operations fields of the Operations tab will be completed.



- Click on Mapped to Implementation and select the appropriate implementation. Choices include: WSDL, Component and Process. The dialog will differ depending on your choice. Below is an example of the dialog when WSDL has been chosen as the implementation method.



8 In all cases, you will need to provide fill in the Endpoint information:

Property	Value
Timeout	
Retry Count	
Retry Interval	
HTTP Params	Edit...

- ◆ Specify a **Timeout Value** or use the XPath Expression Builder to identify one.
- ◆ Specify a **Retry Count**.
- ◆ Specify a **Retry Interval**.
- ◆ Identify your **HTTP Header Params** by clicking on Edit.

9 If the implementation method is WSDL, the **Authentication** Tab must also be filled in.

Connection: -- None --

User ID:

Password:

Client Certificate: Browse

Client Private Key: Browse

Private Key Password:

Connection Timeout(sec):

- ◆ Select either an Endpoint-Defined or Service-Provider-Defined **Connection**.
- ◆ Fill in a Userid and Password as appropriate.
- ◆ Identify a **Client Certificate**, **Client Private Key** and **Private Key Password** as appropriate.
- ◆ Type in a valu, in seconds, to be used for the **Connection Timeout**.

10 Click **OK** when you have finished selecting the appropriate choices for each field to return to the editor pane.

11 Select **File>Save**, or click the **Save** button to save your Service Provider.

About Service Provider Type Resources

Service Providers can be classified into Types. Each Service Provider Type describes its interface(s) using WSDL. According to the WSFL specification (<http://www-106.ibm.com/developerworks/webservices/library/ws-ref4/>), “Service providers must properly implement the appropriate Web service interface in order to be classified as the appropriate type of service provider to handle a particular activity in the business process.”

WSFL requires the process designer to explicitly specify the roles as part of the process implementation. Composer allows you to do this by creating Service Provider Type Resources.

The serviceProviderType element identifies each type of role with the context of a given business process model and the specific Web service interfaces (in the form of WSDL-defined portTypes) that must be implemented by a Web service provider in order to fulfill that role.

NOTE: You must have WSDL resources in place to create new Service Provider Type Resources.

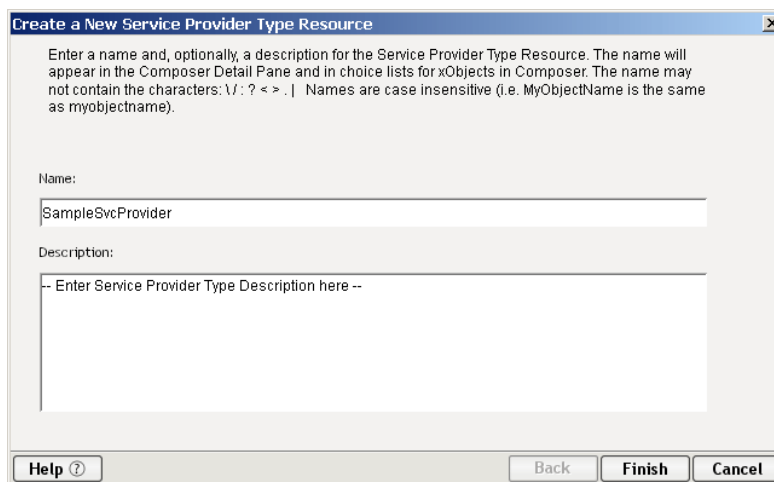
➤ To create a new Service Provider type:

- 1 From Composer’s **File** menu, select **New**, then **xObject**. From the **Resource** tab, select **Service Provider Type**.

◆ or

Right-click on the **Service Provider Type** Resource icon in the Category pane, and choose **New**.

- 2 This will cause the Create a New Service Provider Type dialog to appear.



Enter a name and, optionally, a description for the Service Provider Type Resource. The name will appear in the Composer Detail Pane and in choice lists for xObjects in Composer. The name may not contain the characters: \/: ? < > . | Names are case insensitive (i.e. MyObjectName is the same as myobjectname).

Name:
SampleSvcProvider

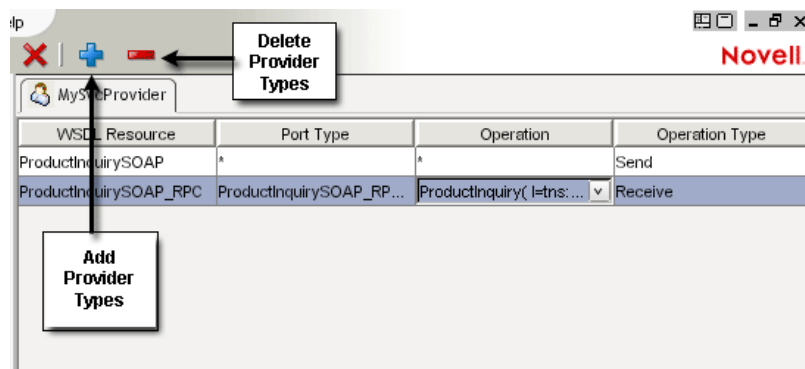
Description:
-- Enter Service Provider Type Description here --

Help ? Back Finish Cancel

- 3 Click on **Finish** to add the new Service Provider Type to the list and to open it in the Editor panel.

➤ To add Service Provider Types in the Editor:

- 1 Click on Add to create a new Service Provider Type.



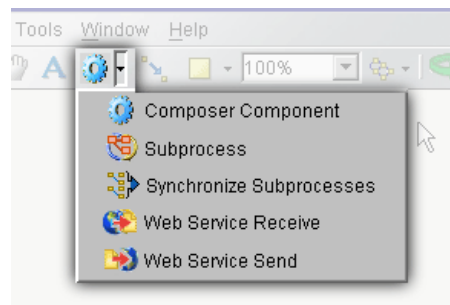
- 2 Select your **WSDL Resource** from the dropdown list.
- 3 Select the **Port Type**.
- 4 Select the **Operation** to perform.
- 5 Select the **Operation Type** (i.e., Send, Receive).
- 6 Repeat the previous steps until you've added all your data.
- 7 Select **File>Save**, or click the **Save** button.

Creating Activities

Even if your activity implementations (Composer Components, Subprocesses, etc.) have not yet been built, you can begin laying down activity icons at any time. For this example, we will assume that the activities consist of prebuilt Composer Components.

➤ **To create an Activity:**

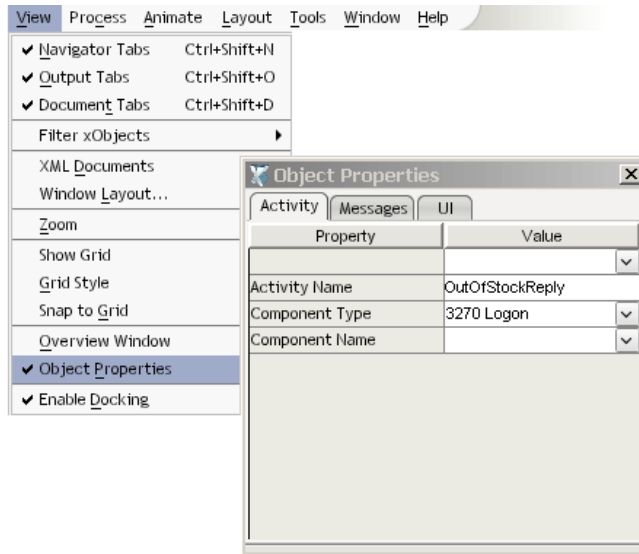
- 1 Choose the appropriate **Activity Tool** type from the toolbar. To see a flyout icon list, click the small triangle next to the current activity tool:



- 2 **Click** on the canvas. A new activity appears.
NOTE: To reposition the activity icon on the canvas, first choose the **Pointer Tool**, then click and drag the activity icon. You can control snap-to-grid behavior by using the Grid submenu under Composer's View menu.
- 3 Repeat the preceding steps as necessary to create additional activities for the process model.

➤ **To associate an implementation with an Activity:**

- 1 **Select** the activity by single-clicking on its icon.
- 2 Under Composer's View menu, choose **Object Properties** in order to bring the Object Properties panel into view. See graphic below.



- 3 Select the **Activity tab** (or Subprocess tab, etc., as applicable, depending on the type of activity) of the Object Properties panel.
- 4 Select the appropriate **Activity Type** (Composer Component, Subprocess, etc.) from the pulldown menu provided, if the type that it showing is not what you expected.

NOTE: For purposes of this discussion, we will assume that the activity implementation is a Composer Component.

- 5 Next to **Component Type**, select the desired type of component (JDBC, XML Map, or whatever applies).
- 6 Next to **Component Name**, use the pulldown menu to select among the already built components in the current project that match the Component Type specified in the previous step. (If your current Composer project has four XML Map Components and you have selected XML Map as the Component Type in Step 4, you will see the names of your four XML Map Components in the pulldown menu.)

➤ **To rename an Activity:**

- 1 **Select** (click on) the activity with the Pointer Tool. Resize-handles (small blue squares) will appear around the activity icon, indicating that the icon has focus.
- 2 Click directly on the activity's **name**. A text-entry field will appear, with the activity name highlighted:



- 3 **Type** the new name for the activity.
- 4 **Click** off to the side to deselect (remove focus from) the activity.

NOTE: Activities carry their own names, separate from their underlying implementations.

Creating Links

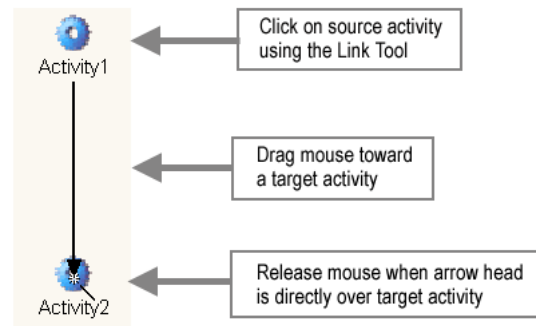
Once your activities have been placed on the canvas, you will want to connect them via control links. As explained in Chapter 1, control links control the flow of execution in a process. We will discuss data flow further below in the section on Message Mapping.

➤ To create a Link:

- 1 Select the **Link Tool** from the Process Designer toolbar.



- 2 **Click** on an activity. Doing so will designate the activity as the *source* for the link.
- 3 Without letting up on the mouse button, **drag the cursor** from the source activity to any activity that you want to be the *target* activity. As you drag around the canvas, the link arrow will “rubber-band” out as it tracks the mouse.
- 4 With the cursor directly over the target activity, **release** the mouse button. The link will change color and redraw immediately to show the connection between the two activities’ bounding boxes.

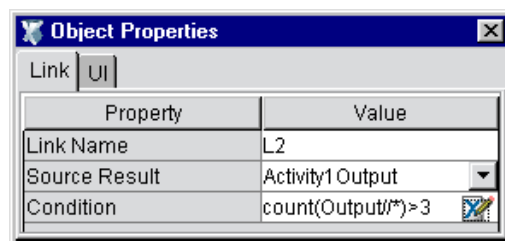


Link Transition Logic

Because link conditions are specified in XPath and therefore require knowledge of the source activity’s output message structure, it is usually best not to specify link conditions until after all data mappings have been specified. (See discussion further below.) Nevertheless, if you already understand the data relationships between source and target activities, you can specify a link condition at any time.

➤ To specify a Link Condition:

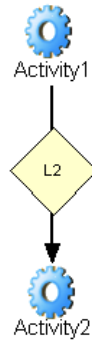
- 1 **Select** a link by clicking on it.
- 2 Bring the **Object Properties** panel into view if it is not already visible. (Toggle its visibility using the Object Properties command under Composer’s View menu.)



- 3 Select the **Link** tab if it is not already selected.
- 4 Next to **Condition**, type an XPath expression that will evaluate to a boolean value.

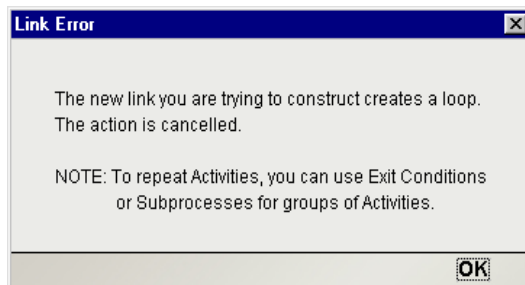
NOTE: If you enter nothing, the runtime engine will assume that the value of the link is *true* by default.

- 5 **Close** the Object Properties panel if desired. Notice that the link’s onscreen representation has changed to include a diamond, indicating that XPath logic has been associated with the link.



Links That Point the “Wrong Way”

If you attempt to draw a link connecting a target activity to one of its upstream sources (i.e., a backwards-facing link), you will get an error:



Cyclic graphic patterns (reentrant loops) are not supported by Composer Process Manager. See discussion in Chapter 2 for details (“Looping”, page 46).

Message Mapping

The transfer of data from one activity to another occurs via *data links*. Unlike control links, data links are not “drawable.” They have no visual symbology on a process graph. Instead, data links are established via *message maps*. These maps are simply XPath-to-XPath correlations between a source activity’s output and a target activity’s input. In other words, they are defined much the same way as Map Actions in an ordinary Composer XML Map Component.

Message Naming

Composer Process Manager uses a default naming scheme to label message sources and targets. When you place the first activity on a new canvas, Process Manager assigns a default name of *Activity1* to the activity. (Subsequent activities are named *Activity2*, *Activity3*, etc.) Accordingly, Process Manager assigns a default name of *Activity1Input* to *Activity1*’s input message and the name *Activity1Output* to the activity’s output message. Even if you later change the name of *Activity1* to *CodeRedFireAlarm*, the name of its input and output *messages* do not change, unless you change them manually (see procedure below). They continue to have the default names of *Activity1Input* and *Activity1Output*.

DOMs are associated with messages, and DOM names (Input, Input1, Temp, Output, etc.) are referenced off the message name. From there, normal XPath rules apply. For example:

```
Activity1Output/Output/PRODUCTREQUEST/SKU
```

means the XPath node `/PRODUCTREQUEST/SKU` on the Output DOM of the message named *Activity1Output*. You will see how this works in subsequent examples and screenshots.

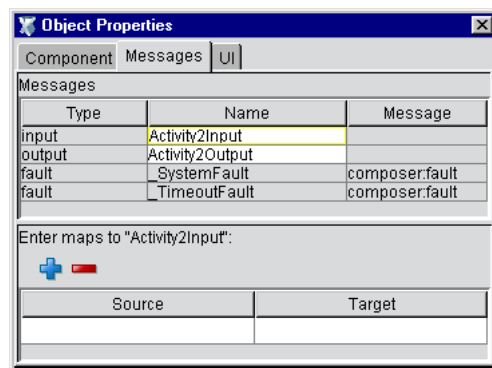
How to Define Message Mappings

To send data from a source activity to a target activity, you need to define at least one message map.

NOTE: All message maps are defined at the *target activity* (the “receiver” of incoming data), as described below.

➤ To define a Message Map:

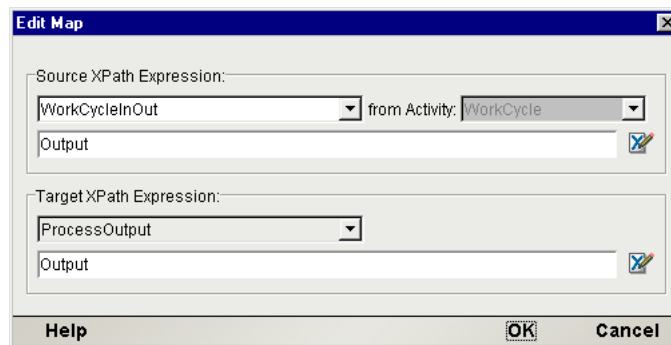
- 1 Select (click on) an activity. This is the activity whose data source(s) you will specify.
- 2 Bring the **Object Properties** panel into view if it is not already visible. (Toggle its visibility using the Object Properties command under Composer’s View menu.)
- 3 Select the **Messages** tab. The tab have an appearance similar to this:



- 4 In the upper half of the panel, you will see **Type**, **Name**, and **Message** information, with the default Names showing for input, output, and fault messages. You can enter a new Name for any message at this time if desired.

NOTE: Fault messages are discussed separately, later in this chapter.

- 5 In the lower half of the panel, you can define Source-to-Target message mappings using XPath. Click the **Plus** icon to add a mapping. A dialog appears.

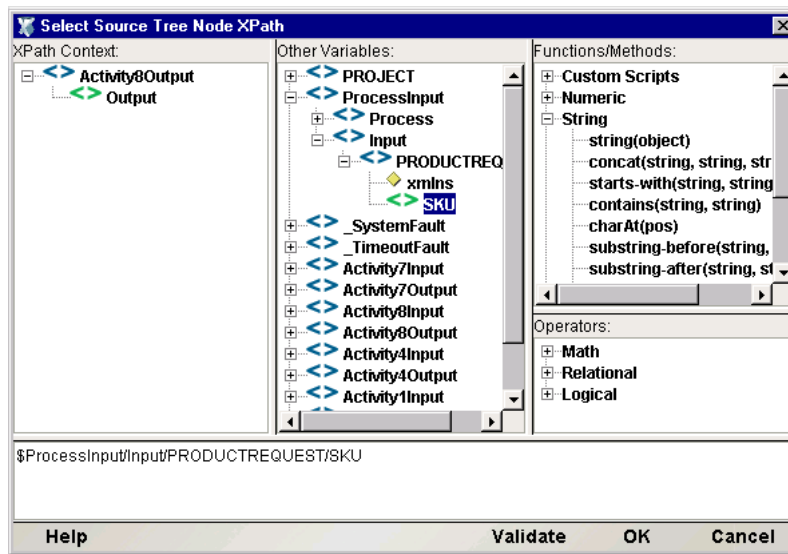


- 6 Using the pulldown menu immediately under **Source XPath Expression**, select the message that you want to use as the data source for this mapping. The prepopulated list will contain output message names from all available (legal) data sources. (In other words, you can choose to map data from any activity that can be reached by back-traversal of links. This includes the *ProcessInput* message itself.)

- In the **Source XPath Expression** text field area, enter any desired XPath statement to specify a source element, nodetree, whole DOM, etc., coming from the activity shown in the “from Activity” pulldown menu. (This menu will be greyed out if there is only one incoming link to the current activity. On join targets, this menu will be prepopulated with the names of all available incoming messages.)

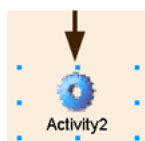
NOTE: It is a common case to specify “Output” (the source activity’s Output DOM) as the sole incoming message part, as shown here.

- (Optional) If you would like to generate XPath using Composer’s Expression Builder, click the small “pencil and X” icon to the right of the text field. This will bring up the XPath Expression Builder window:



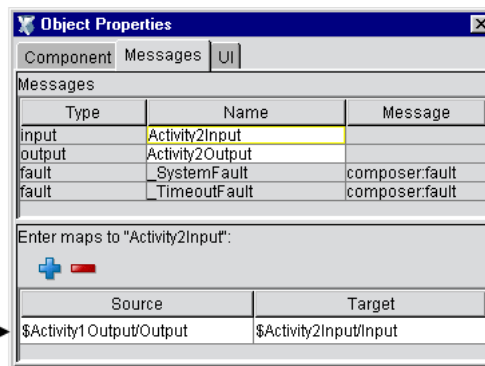
The upper panes of this editor window are prepopulated with message trees, XPath native script methods, etc., for your convenience in building XPath expressions. Doubleclick any node in any tree to make the correct sub-expression appear in the edit field. Click OK to go back to the Edit Map dialog.

- In the **Target XPath Expression** text field area, enter any desired XPath statement to specify a target to receive data from the input message.
- NOTE:** It is a common case to specify “Input” (the target activity’s Input DOM) as the target message part. This is equivalent to mapping Source data to the Input DOM of the activity.
- Click **OK** to dismiss the dialog.



On the graph, Activity2 is selected

Activity1’s Output DOM has been mapped to Activity2’s Input DOM



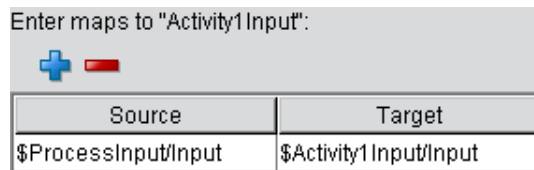
The Source and Target mapping information that you just specified are now visible in the Messages tab of the Object Properties window, as shown above. (A summary view of the info is also available in a rollover tooltip if you let the mouse loiter over these fields.)

Mappings of this sort continue throughout the chain of activities shown in the process graph. You will need to perform this mapping procedure at least once for every activity that receives data.

Data Mapping for Start and End Activities

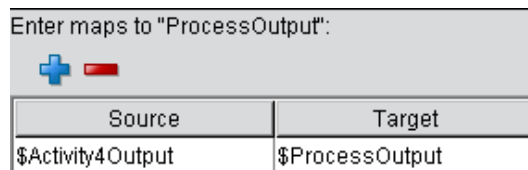
To specify the input to your process's start activity (or activities), simply click on the start activity, bring the Object Properties panel into view, and specify a mapping from *ProcessInput* to the start activity, using the procedure given above.

If your start activity is named Activity1, the resulting map specification might look like:



Source	Target
\$ProcessInput/Input	\$Activity1 Input/Input

To specify a mapping from an *end* activity to *ProcessOutput*, click anywhere on bare canvas, bring the Object Properties panel into view, and specify a mapping from the end activity's output message to the *ProcessOutput* message. The result might look like:



Source	Target
\$Activity4Output	\$ProcessOutput

Selecting a Process Input Template

As mentioned earlier (in the discussion of how to create a new process), you can specify an input template document for *ProcessInput* (for design-time testing purposes) during the initial creation of the process. If you did not specify any XML Templates at that time, or you now want to use a different template, simply go to the **File** menu and select the **Properties...** command. A dialog will appear. Select the **Messages** tab within that dialog. There, you will be able to add or remove templates as desired.

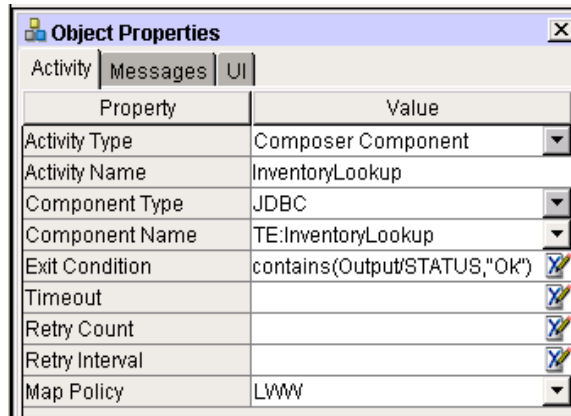
Applying Flow Logic at the Activity Level

Activity flow logic (join conditions and exit conditions) can be specified in the Object Properties panel. These conditions are optional: By default, the runtime engine will assume that an empty condition is *true*.

➤ **To specify an Exit Condition:**

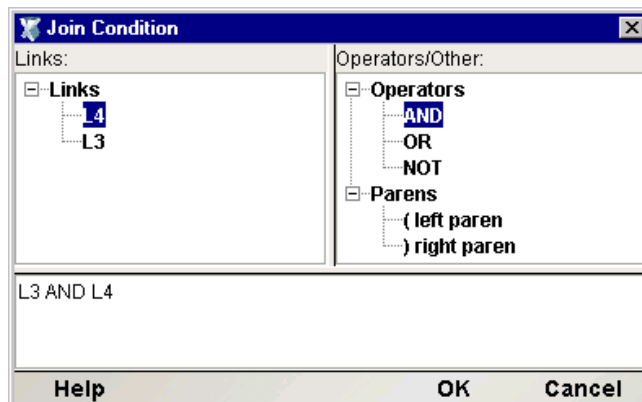
- 1 **Select** (click on) an activity.
- 2 Bring the **Object Properties** panel into view if it is not already visible. (Toggle its visibility using the Object Properties command under Composer's View menu.)
- 3 Select the **Activity tab** (or Subprocess, etc., as applicable to the selected activity).

- Next to **Exit Condition**, enter an XPath expression. This condition must evaluate to *true* or *false* at runtime. If it evaluates to false, the activity will execute again using the original input data. (See discussion in Chapter 2.) The activity will continue to reexecute until the exit condition is true or a timeout occurs.



➤ **To specify a Join Condition:**

- Select** (click on) a join activity—that is, any activity that has more than one incoming link.
- Bring the **Object Properties** panel into view if it is not already visible. (Toggle its visibility using the Object Properties command under Composer’s View menu.)
- Select the **Activity tab** (or Subprocess, etc., as applicable to the selected activity).
- Next to **Join Condition**, enter a join expression based on the truth values of incoming links. This condition must evaluate to *true* or *false* at runtime. Optionally use the **Expression Builder** to build the join condition. Click the blue icon at the right edge of the text field. The Expression Builder dialog appears.



- The Links tree in the upper left is prepopulated with the names of available incoming links. Link-expression syntax helpers are in the upper right. Doubleclick on any leaf node in any tree to build an expression in the text-edit field below. Then dismiss the dialog.

Recall that a join activity cannot fire until the join condition is true. In Deferred Mode (default), a join condition is evaluated exactly once, when the truth values of all incoming links are known. In Immediate Mode (which you can select on the Object Properties panel), the join condition is evaluated as truth values become known, and as soon as it is true, the join activity fires regardless of whether all source activities have finished executing.

NOTE: If, during a design session, you assign a join condition to an activity and later remove one or more incoming links, the join logic may no longer function as intended. Be sure to remember to update join conditions any time the input links to a join have been removed or replaced.

Timeouts and Retries

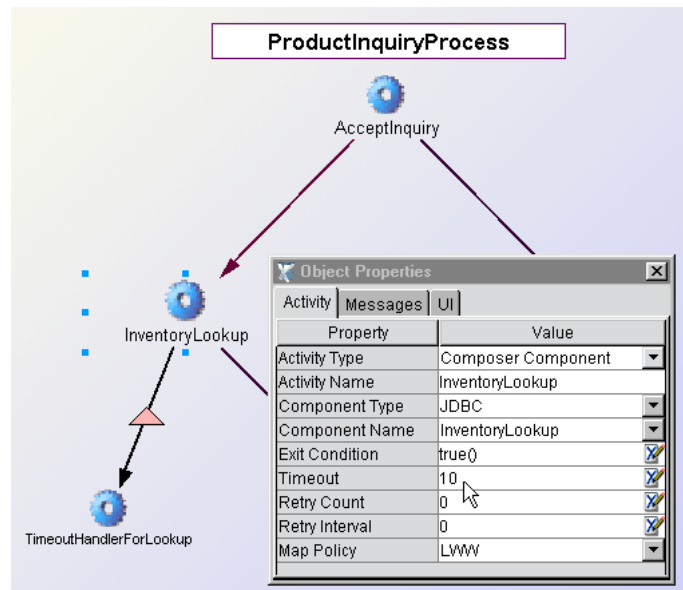
Timeout, Retry Count, and Retry Interval parameters are supported by Process Manager so as to allow for complex choreographies between partners involving timings and roundtrip interactions that are not addressed by simple Request/Response and Solicit/Response scenarios.

NOTE: It's important to keep in mind that Retry Count and Retry Interval come into play only when a Timeout value has been specified. Otherwise, Retry Count and Retry Interval are ignored.

See the discussion of fault trapping further below for a more detailed explanation of how and under what conditions timeout faults can occur.

➤ To specify Timeout and Retry Parameters

- 1 Click on an activity for which you wish to set Timeout and Retry parameters.
- 2 Bring the **Object Properties** panel into view if it is not already visible. (Toggle its visibility using the Object Properties command under Composer's View menu.)



- 3 Select the **Activity** tab.
- 4 Next to **Timeout**, enter a value in days, minutes, or seconds (suffix values 'd', 'm', or 's', respectively). Example: To specify 7 days, enter "7d".
NOTE: If you use a unit specifier, you must enclose the entire value in quotation marks. If you enter a number without units (and without quotation marks) it will be interpreted as *seconds*.
- 5 If you entered a Timeout value, optionally enter a number in the text field next to **Retry Count**. This is the number of times the activity will be retried after the first try times out. *If you do not enter a number, zero retries will be attempted.*
- 6 In the text field next to **Retry Interval**, enter a value representing the wait time between retries (in seconds). The default is *zero*, meaning that as soon as the activity times out, it will be retried with no wait. If the Retry Interval is non-zero, Process Manager will wait the specified amount of time between the timeout and the retry.

Map Policy

Map Policy comes into play when multiple data sources (incoming messages) have parts that map to the same location in the target activity's input message(s). For example, consider a process in which Activity1 and Activity2 have links to a join activity, Activity3. If *Output/ShipmentMode* from Activity1Output maps to *Activity3Input/Input/ShipVia*, and *Output/Carrier* from Activity2Output also maps to *Activity3Input/Input/ShipVia*, the potential exists for a collision. The result depends on whether you want to keep the last-arriving data (allowing overwrites in time order, as they occur) or keep only the *first*-arriving data. To specify this, you must set the Map Policy to LWW (Last Writer Wins) or FWW (First Writer Wins), as appropriate.

NOTE: Recall that an activity *does not have to be a join activity* in order to receive data from multiple upstream sources. Therefore, it's possible for map policies to come into play even when there is only one incoming control link to a target activity.

LWW, FWW, and Map Order

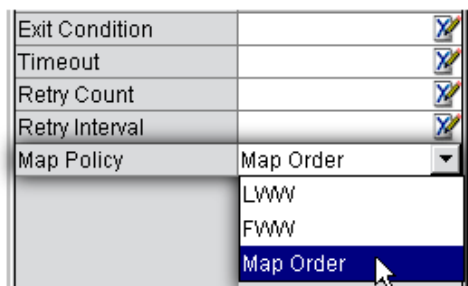
The choices for Map Policy are LWW (Last Writer Wins) or FWW (First Writer Wins), or Map Order. The meanings of the first two choices are self-evident. Map Order requires further explanation.

Map Order means that regardless of the timestamp on incoming messages, XPath-to-XPath mappings will occur in the order in which the mappings are specified in the Messages tab of the activity's property sheet, going top to bottom. Timestamps, in other words, are ignored. Messages are cached when they arrive, and then—when mapping takes place—every message part is mapped according to the literal order in which you have specified the mappings.

You would typically use this option as a way of dealing with overwrites when you care more about where messages are coming from than you do about their actual arrival order. For example, if several activities feed into a join, and one particular source activity should always have write-preference over other feeder activities, then you could use Map Order to give the preferred source a higher precedence (for overwrite) than the others.

➤ To set a Map Policy:

- 1 Click on an activity for which you wish to set a Map Policy.
- 2 Bring the **Object Properties** panel into view if it is not already visible. (Toggle its visibility using the Object Properties command under Composer's View menu.)
- 3 Select the **Activity** tab.
- 4 Use the pulldown menu next to **Map Policy** to specify **Last Writer Wins (LWW)**, **First Writer Wins (FWW)**, or **Map Order**.



Fault Messages and Fault Handling

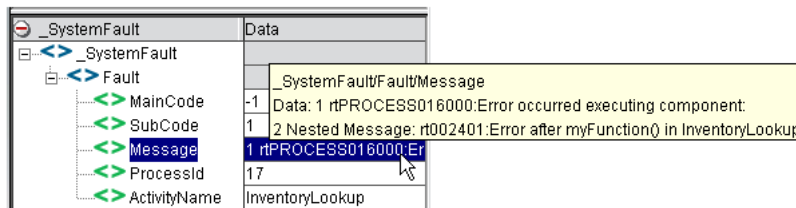
Faults generated by Process Manager are of two general kinds: System and Timeout. Both are generated as special messages. The fault messages take the place of the Activity Output message for the activity that raised the fault. In other words, an activity implementation that faults out is not considered to produce Output. An activity will therefore either produce an Output message or Fault message, but not both.

System Faults

The runtime engine raises a *System fault* under the following circumstances:

- ◆ An activity implementation generates an unhandled exception
- ◆ A subprocess activity returns a fault message
- ◆ The runtime engine encounters a message or message type that it doesn't know how to handle
- ◆ A Timeout fault occurred and was not handled by an activity designed for that purpose. (In this case, two faults are actually generated: one Timeout and one System.)

When a System fault occurs, the process instance produces a message called `_SystemFault`, with a part name called (also) `_SystemFault`. The DOM view of the message looks like:



Every System fault contains `MainCode`, `SubCode`, `Message`, `ProcessID`, and `ActivityName` elements. The content of each element is visible in a rollover tooltip as shown above for the `Message` element. Notice that the `Fault/Message` element contains a `Nested Message`. The value of this `Nested Message` is whatever custom string value you put in any `Log` or `Raise Error` action (assuming the implementation is a `Composer` service or component).

Regardless of the cause, a fault (of any kind) will result in termination of a running process, *unless the fault is handled by an activity designed for that purpose*. In this respect, faults are similar to exceptions. If no handler exists, the fault “bubbles up” to the process engine and the process simply allows the process instance to exit with a fault message. Any activity instances in existence at the time of the unhandled fault are aborted.

Fault Codes

`MainCode` values currently implemented include:

SYSTEM_FAULT_MAINCODE	-1
TIMEOUT_FAULT_MAINCODE	-2

`SubCode` values currently implemented include:

COMPONENT_FAULT_SUBCODE	1
UNHANDLED_MESSAGE_SUBCODE	2

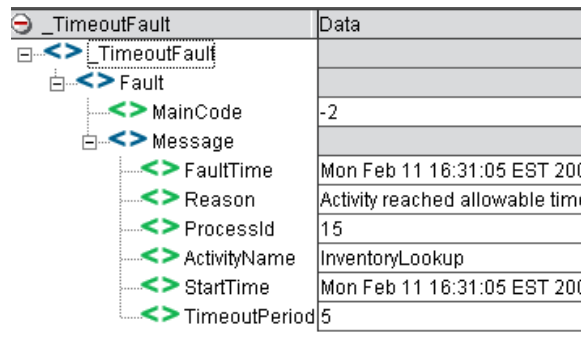
Timeout Faults

The runtime engine enforces the following behavior when a Timeout value exists on an activity:

- When the activity is launched, a timer begins.
- If the activity completes with an exit value of *true* prior to the timeout period, control passes to outgoing link(s).
- If the activity completes with an exit value of *false* prior to the timeout period, the activity is reexecuted immediately (which is the normal action for all activities that finish with a false exit condition).
- If the activity hasn't finished running when the timeout is reached, the runtime engine halts the activity and consults the Retry Count parameter. If the Retry Count is non-zero, the Retry Interval parameter (if applicable) is consulted, and the runtime engine waits for the time specified in Retry Interval; then it resets the Timeout clock and reexecutes the activity using the original data mappings. This execute-wait-retry cycle is repeated until the Retry Count has been reached, at which point the engine raises a Timeout fault.

If a Timeout fault is not handled by an activity, it will cause the runtime engine to terminate the process.

The Timeout fault message has this appearance, in tree view:

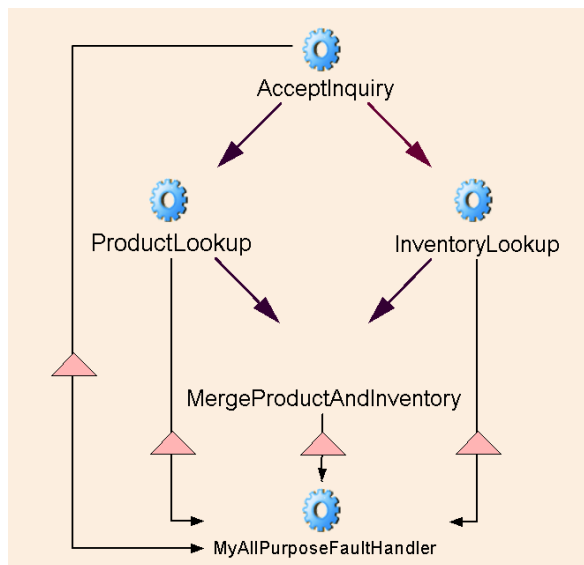


	Data
TimeoutFault	
Fault	
Message	
MainCode	-2
FaultTime	Mon Feb 11 16:31:05 EST 200...
Reason	Activity reached allowable time...
ProcessId	15
ActivityName	InventoryLookup
StartTime	Mon Feb 11 16:31:05 EST 200...
TimeoutPeriod	5

The message elements are self-explanatory. The `MainCode` value is -2 for Timeout (as explained above).

Fault Handling

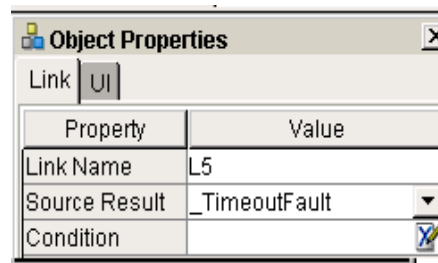
Because faults can be handled by custom-designed activities (which in turn can be implemented as Web Services, Composer Components, etc.), fault logic can be as sophisticated as it needs to be. You can designate one fault handler activity for each activity that needs one (its implementation could even consist of the same Composer component in each case); or you could have a single fault-handling activity that handles all faults for the entire process. An example of the latter is shown in the graph below. Every activity has a link to *MyAllPurposeFaultHandler*, which handles faults for the entire process.



The triangle shape on each link signifies that the link has been designed to handle fault flow. The procedure below tells how to create the necessary control and data links to handle faults.

➤ **To attach a fault handler to an activity:**

- 1 Create the implementation of the *fault handler activity*, if you have not already done so. (Since this activity will generally be local to the app server, it usually makes sense to implement it as a Composer Component.)
- 2 Place the activity icon for the fault handler on the process graph.
- 3 Draw a link from the appropriate *source activity* (the activity that generates the fault) to the fault-handler activity.
- 4 Click on the link you just drew, to select it.
- 5 Bring the **Object Properties** panel into view, if it is not already visible.
- 6 Click the **Link tab**. You should now be looking at something similar to the following:



- 7 Next to **Source Result**, use the pulldown menu to select the appropriate fault type. In this case, `_TimeoutFault` was chosen.

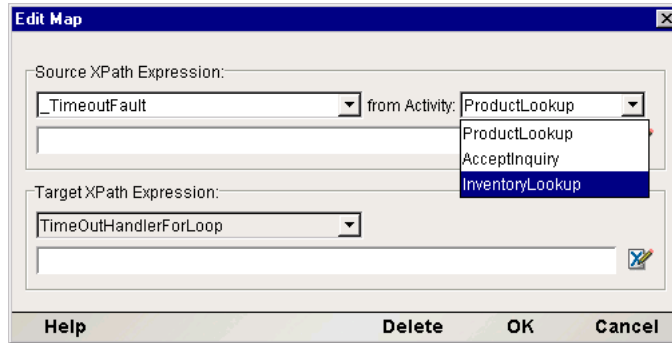
NOTE: On the graph, the link will acquire a triangle icon at this point.

- 8 If a special condition applies to this link, enter an appropriate **XPath** expression.
- 9 Save your work.

➤ **To create data mappings into a fault handler:**

- 1 Click on the fault-handler activity to select it.
- 2 Bring the **Object Properties** panel into view.
- 3 Click the **Messages** tab.

- In the bottom portion of the tab, click the **Plus-sign** icon to add a message. The following dialog appears.



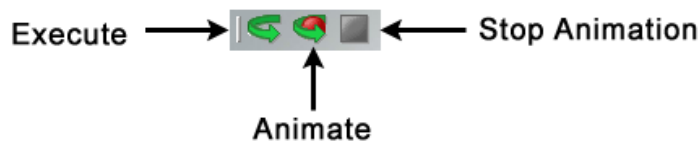
- Select **SystemFault** or **TimeoutFault**, as appropriate, from the top left pulldown menu, under **Source XPath Expression**.
- In the pulldown menu next to **from Activity**, select the source activity for this message.

NOTE: All possible source activities will be listed—that is, any upstream activity that is reachable by simple one-way back-traversal of links. When you select one of these activities, you are creating a data link from that activity to the fault handler. The source activity you choose from this list does *not* have to be directly connected to the target activity by a control link, but in most cases, you *will* want such a flow-control connection, since a data link, by itself, isn't enough to fire a fault handler. Bottom line: If you data-map a fault message to an activity input, be sure, also, to draw a *control link* from the source activity to the fault activity so that the fault activity will actually fire.
- Dismiss the dialog by clicking **OK**.
- Repeat Steps 4 through 7 for each activity that will feed into this fault handler.

Animation and Testing

A unique and powerful feature of the Process Manager is that it allows you to run and debug processes (step into or over activities, etc.) *in the design-time environment*. And because Process Designer runs within Composer itself, you can step directly into any Composer Components that make up the implementation(s) of activities. Once inside the component, you can step through the action model just as you would during a component design session, watch DOMs change in real time, set breakpoints, etc. You can debug activities at the same time that you test and debug your process.

You can either animate or execute a process via the special toolbar buttons provided for this purpose:

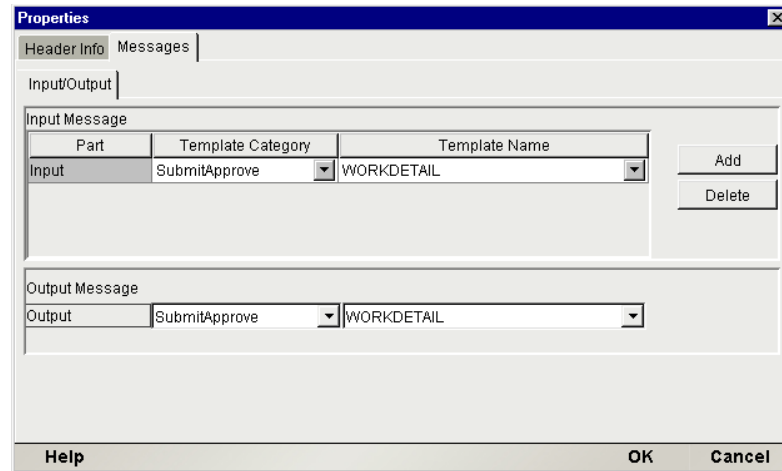


The difference between Execute and Animate is that Execute runs the process from start to finish without interruption, whereas Animate allows you to step through the process.

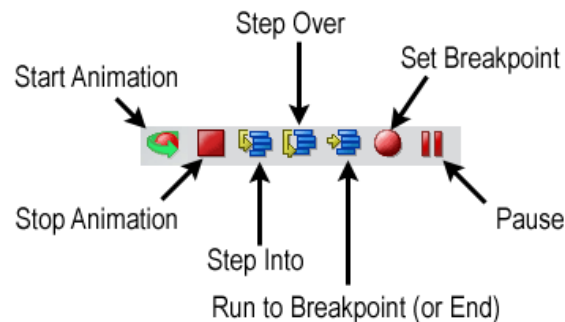
Process Designer gives valuable visual feedback during animation. Whether you Execute or Animate a process, you can see individual control links highlight (become thicker) as control passes from one highlighted activity to the next, and if a link cannot be followed (because its condition is false), that link's representation changes from a solid line to a dashed line. Therefore it's easy to see, at a glance, which links are being followed and which activities are executing.

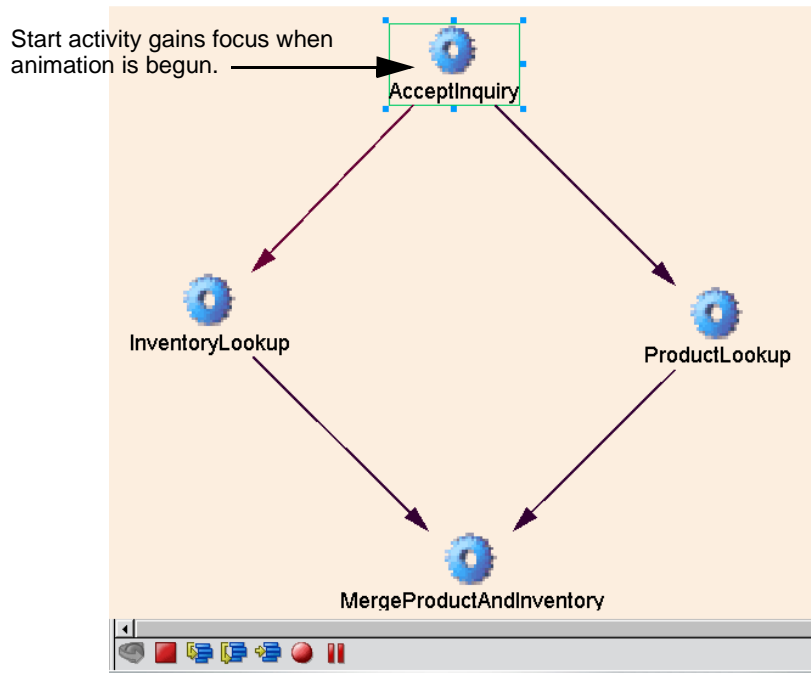
➤ **To animate a process:**

- 1 (Optional) Clear the output pane (where system messages appear) at the bottom of the Composer main window. To do this, click inside the pane, type Control-A (Select All), and hit Backspace.
- 2 If you have not already assigned a `ProcessInput` data template to the process for test purposes, select **Properties** from the **File** menu, then click the **Messages** tab. Otherwise, skip to Step 6 below.



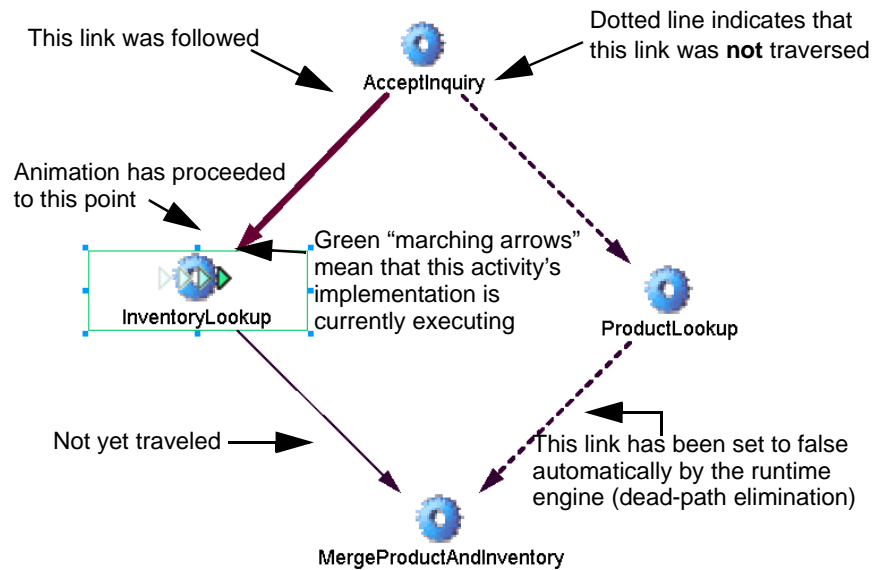
- 3 Choose and/or **Add** an Input Message by selecting from available XML Templates using the pulldown menus in the upper half of the dialog. Likewise, choose an Output template if needed.
- 4 Dismiss the Properties dialog by clicking **OK**.
- 5 Create data mappings between `ProcessInput` and your start activity. (The procedure for this was discussed previously. See “Data Mapping for Start and End Activities” earlier in this chapter.)
- 6 Click the **Animate** button in the Process Designer toolbar. The start activity will highlight. (See below.) Also notice that a new toolbar appears at the bottom of the graph window. The icons, from left to right, are Animate (dimmed when active), **Stop**, **Step Into**, **Step Over**, **Run to Breakpoint**, **Set Breakpoint**, and **Pause**:





- 7 If you wish to step into the activity implementation, click the **Step Into** button. This will open the activity's underlying component in the appropriate component editor environment within Composer. You can then step through the component's action list as you normally would in Composer. After you step through the last action in the action list, the process graph window will reappear.
- 8 If you wish to step over the currently highlighted activity, click the **Step Over** button. The appropriate link(s) will be followed and links will change appearance either to a double-thickness solid line (for true links) or a dotted line (for false links). Execution will stop at the target activity (or activities). You can then use Step Over or Step Into again, and so on.
- 9 To run to the end of the process, click the **Run to End** icon. When the process is finished running, a small alert dialog will appear, explaining whether the process finished normally or errored out in some manner.

Note that as you step through the chain of control, various links will highlight and change appearance to reflect the actual path followed during execution. For example, in the graph shown below, animation has proceeded past the start activity to the next activity in the flow. One of the two outgoing links from the start activity has been followed (namely, the dark, solid link on the left); the other link (with a dotted line appearance; right) was not followed, because its transition condition was false.



Notice that because the link from *AcceptInquiry* to *ProductLookup* was not followed, the link from *ProductLookup* to *MergeProductAndInventory* is also shown as a dotted line (even though execution has not proceeded to this point yet), through dead path elimination. The process engine knows that if the link from *AcceptInquiry* to *ProductLookup* is false, *there is no way the link from ProductLookup to MergeProductAndInventory can ever be followed*—hence this downstream link can be (and in fact must be) set to false as well. The reason this must occur is that the join condition at *MergeProductAndInventory* will never evaluate if it is waiting on the truth value of a *feeder link* that will never evaluate. (See the discussion of “Dead Links and Synchronization Failure”, Chapter 1.)

Aids to Debugging

Process Designer provides many ways to monitor the step-by-step execution of a process. For example, valuable realtime feedback is given (in plain text form) in the Log pane of the Composer window, and you can look at any activity’s input or output DOMs (or even the ProcessInput and ProcessOutput) along with DOM views of fault messages, to see exactly what data values were produced at various points in the flow.

Watching System Messages at Animation Time

Any time you execute or animate a process in Process Designer, system messages will appear in the Log pane at the bottom of the main Composer window. See below.

```
10(2): The process has started (ProductInquiryProcess)
10: executing data link: ProcessInputAcceptInquiry (ProcessInputAcceptInquiry)
10(6): The activity has started (AcceptInquiry)
+++++ Thu Feb 14 11:20:14 EST 2002 USER LOG FROM AcceptInquiry
-----
activityReturn(10, AcceptInquiry)
10: Evaluating condition for link L2 (L2)
10: following control link: L2 (L2)
10: Evaluating condition for link L1 (L1)
10: following control link: L1 (L1)
10(7): The activity has completed (AcceptInquiry)
10: executing data link: AcceptInquiryInventoryLookup (AcceptInquiryInventoryLookup)
10(6): The activity has started (InventoryLookup)
10: executing data link: AcceptInquiryProductLookup (AcceptInquiryProductLookup)
10(6): The activity has started (ProductLookup)
Log Find
```

The information in this pane is quite detailed. Every activity startup, link evaluation, join evaluation, activity completion, activity error—every event—is logged so that you can go back through the chain of events and see exactly what executed and when, and what failed and why.

NOTE: Two numbers (one of them in parentheses) precede every message. The first number is the ProcessID for the current instance. The second number, in parens, is the event code for the event in question (6 for activity start, 7 for activity complete, and so on).

If a fault occurs, you can easily identify the offending activity; and you will also see the complete fault message (in XML form):

```
12(6): The activity has started (InventoryLookup)
12: executing data link: AcceptInquiryProductLookup (AcceptInquiryProductLookup)
12(6): The activity has started (ProductLookup)
com.sssw.b2b.ee.process.rt.GNVProcessException: Error occurred executing component;
---> nested Error after call to myFunction() in InventoryLookup
12: Error during activity execution: Error occurred executing component: (InventoryLookup)
activityReturn(12, InventoryLookup)
12(7): The activity has completed (InventoryLookup)
12(4): The process has completed (ProductInquiryProcess)
12: Process "ProductInquiryProcess" ended with fault message:
<?xml version="1.0" encoding="UTF-8"?>
<_SystemFault>
  <Fault>
    <MainCode>-1</MainCode>
    <SubCode>1</SubCode>
    <Message>1 rtPROCESS016000:Error occurred executing component:
2 Nested Message: rt002401:Error after call to myFunction() in InventoryLookup</Message>
    <ProcessId>12</ProcessId>
    <ActivityName>InventoryLookup</ActivityName>
  </Fault>
</_SystemFault>
(ProductInquiryProcess)
```

If an activity was implemented as a Composer Component or Subprocess, you can doubleclick on the activity in question (right on the process graph), and the component will open in the appropriate component editor. You can then make changes to the component, save it, and return to the Process Designer for another animation session.

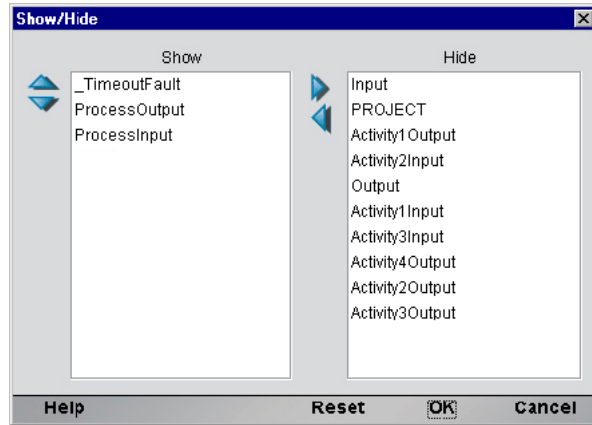
NOTE: When you have made changes to an action model, be sure to Save the component (save the changes) before reexecuting the process. Otherwise you will get the same error(s) again.

Inspecting Messages

Any message produced at any point in a process can be displayed in DOM view, text view, or stylized view in its own pane. This includes ProcessInput, ProcessOutput, _TimeoutFault, and _SystemFault messages as well as all activity input and output messages.

➤ **To make a message visible (or to hide an existing one):**

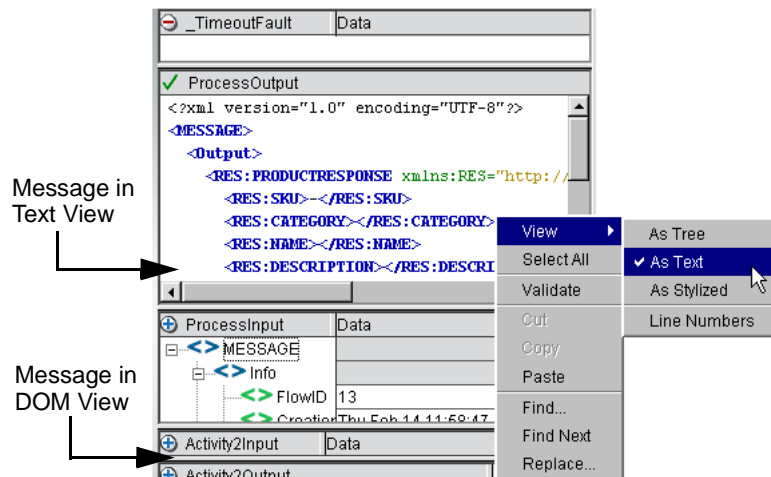
- 1 From the main menu bar, select **View > XML Documents > Show/Hide:**



- 2 In the dialog that appears, use the **left-right arrow** buttons to move messages to the Hide or Show columns as necessary.

NOTE: The prepopulated list on the right will contain the names of only those messages that were actually produced or used in the execution of the process. If a process terminates early, it is possible that some activities' messages won't be listed.

- 3 (Optional) Use the **up-down arrow** buttons to reorder the Show items as desired.
- 4 Dismiss the dialog by clicking **OK**.
- 5 The messages that you designated under Show will now appear in their own data panes. See below.



Note that you can obtain different views of any DOM by doing a right-mouse-click on the DOM in question, then choosing **View > As Text** (or Tree, or Stylized) from the context menu, as shown above.

4 The Process Designer User Interface

This chapter describes the user interface functionality of Composer Process Designer, which is the design-time environment in which you will create your process models.

Main Features

The Process Designer is a visual editing environment for creating process models represented by directed-edge graphs. In this environment, you can quickly create and arrange activities (represented by icons), draw links between activities, and designate data mappings, link conditions, etc., between and among activities. The point-and-click nature of the drawing environment allows for rapid creation of flow graphs.

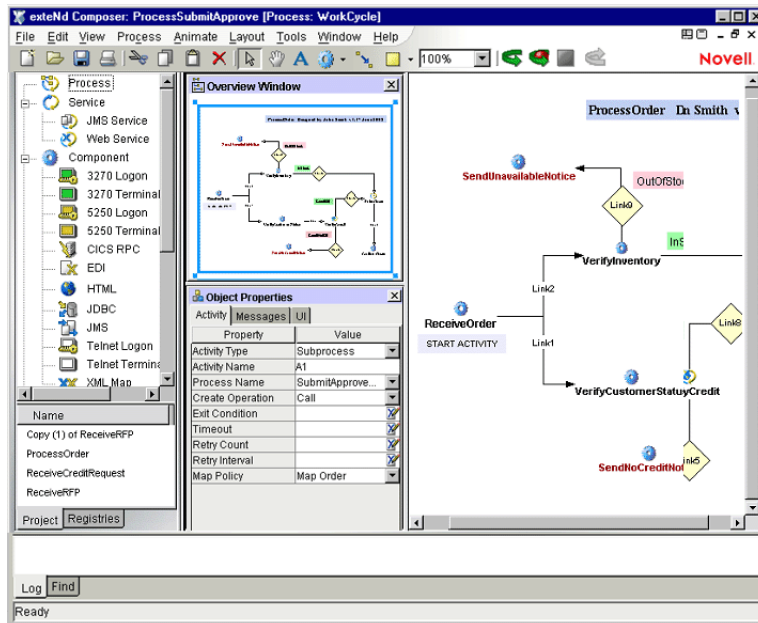
An important feature of Process Designer is that any process can be run in animation mode at design time, so that process models can be designed, tested, and debugged in a single session. In animation mode, you can step into or over activities, set breakpoints, watch data transformations as they occur, see log messages, observe the behavior of splits and joins, etc., all in real time. You can also drill down on activity implementations, make changes to action models in components, edit message maps or documents, modify link or join logic, and so forth, interactively, without leaving the session. This capability greatly speeds development.

The Process Designer Window

The Process Designer runs inside Composer (along with other component editors), so the overall environment should look familiar to any Composer user. (See graphic, below.)

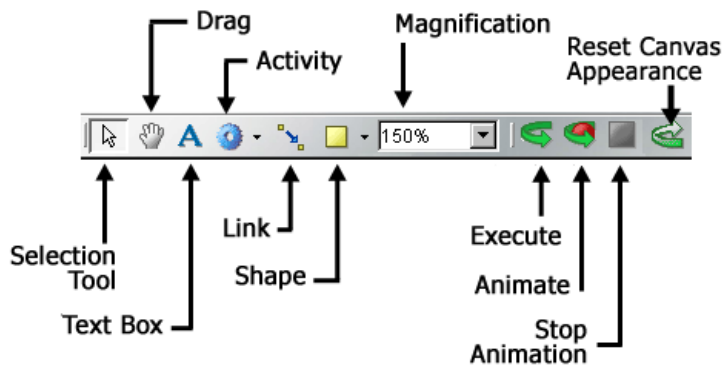
When Process Designer is the front editor, three new panes are visible:




- ◆ The *Process Model Pane* (also called the canvas), where you draw the process model graph. This is the largest pane.
- ◆ The *Object Properties Pane*, in which you can specify property values for various elements of the process model (e.g., activities, links, text labels, and shapes).
- ◆ The *Overview Pane*, which contains a “bombsight view” of the main canvas. By holding the mouse down and dragging the blue rectangle within this mini-window, you can pan across the main canvas, setting the visual focus to a particular region instantly, without using scrollbars.




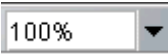




Toolbar

Composer Process Designer adds several new tools to the Composer toolbar, as shown below.



Tool	Icon	Usage
Selection Tool		This pointer allows you to select items by clicking on them. Once selected, an item can be dragged to any spot on the canvas. (You can shift-click to select multiple elements individually.)
Drag		The drag tool allows you to pull the entire canvas around, so that you can “pan across” the canvas and bring any region of interest into view.
Text Box		Click on the canvas with this tool to create a text label inside a rectangle.






Tool	Icon	Usage
Activity		This tool allows you to place new activity elements on the canvas.
Link		This tool lets you connect any two activities with an arrow, representing a control-flow link.
Shape		You can place resizable filled rectangles or ovals on the canvas with this tool.
Magnification		A dropdown menu allows you to choose from several preset viewing magnifications. You can also enter a custom magnification factor into the text field next to the dropdown.
Execute, Animate, Stop Animation, and Reset	  Reset	These buttons allow you to start or stop a process (for testing purposes) within the design-time environment. The Reset button (lower left) is greyed out until an animation has finished running; pressing it resets the graphic appearance of the flow diagram.

Graph Elements

Graph elements include activities, links, text boxes, and shapes (rect and oval). The creation tools for these elements can be accessed via the Process menu on the main Composer menubar or via tool icons on the main toolbar. They operate in point-and-click fashion.

NOTE: The appearance characteristics (colors, borders, text justification, etc.) for each of the different graph-element types discussed below can be adjusted by means of controls located in the UI tab of the Object Properties pane. (You can toggle this pane's visibility by means of the Object Properties command under the View menu.)

Activities

-  Composer Component
-  Subprocess
-  Synchronize Subprocesses
-  Web Service Receive
-  Web Service Send

Activities can be of five types, as depicted above. The various activity types are briefly described below.

Activity Type	Description
Component Activity	The Component Activity provides for runtime interaction with a Component or Service to interact with one or more external systems using one or more Composer Components (e.g. JDBC, 3270, 5250, CICS RPC, JMS, HTML, Telnet, EDI or XML Map as well as Composer JMS Services or Composer Web Services). One can drill down on a Component Activity to view and edit the Composer Component's action model.
Web Service Receive Activity	The Web Service Receive Activity provides for runtime interaction with a published Web Service and correlates a received message with a current process instance.
Web Service Send Activity	The Web Service Send Activity provides for runtime interaction with a published Web Service. It enables the Process Manager user to select the Web Service's WSDL Resource, Service Name, Binding, Operation, Endpoint Locator and Connection. This is similar to the WS Interchange Action introduced in Composer 3.0.
Subprocess Activity	A Subprocess Activity represents any process created in Process Designer. This effectively means a process can call another process. It allows for a layered, hierarchical flow architecture. One can doubleclick on a subprocess Activity to view and edit the subprocess graph.
Synchronize Subprocesses	This is a specialized activity type that allows the merging of information returned from a repetitively executed subprocess.

To create an instance of one of these activity types in your graph, simply select the corresponding tool icon from the main toolbar (or the flyout icon list under the icon), then click anywhere on the main canvas.

Links

The Link tool allows you to connect activities with a directed edge (arrow). Its operation is very simple. First, select the tool from the toolbar. Then click on any activity; this becomes the source activity for the link. With the mouse still down, *drag out a line to the desired target activity*. (Be sure the line extends not just to the activity, but actually over the middle of the activity icon.) When you let go of the mouse, an arrowhead will appear on the "target end" of the link and the two activities will be linked in terms of control flow. At this point, if you use the Selection cursor to drag either activity around the canvas, the link will automatically expand and/or reorient as necessary so that both activities remain connected.

Text Boxes

The Text tool allows you to place text boxes on the canvas. When you click on the canvas, a rectangle will appear with the word "Untitled." You can then change the text in the box, set its background and outline colors, etc., by entering appropriate settings in the UI tab of the Object Properties pane.

Text boxes are simply arbitrary text labels that you can use at various spots around the canvas to document activity characteristics, control-flow intents, etc., or to indicate titles, author info, revision dates, and so on. Text boxes can be repositioned (by dragging) at any time and have no effect on control flow. Their use is optional.

By using the controls in the UI tab of the Object Properties pane, you can change a text box's appearance, not only with regard to colors, resizable, margins and centering, etc., but also involving text size, font, and style.

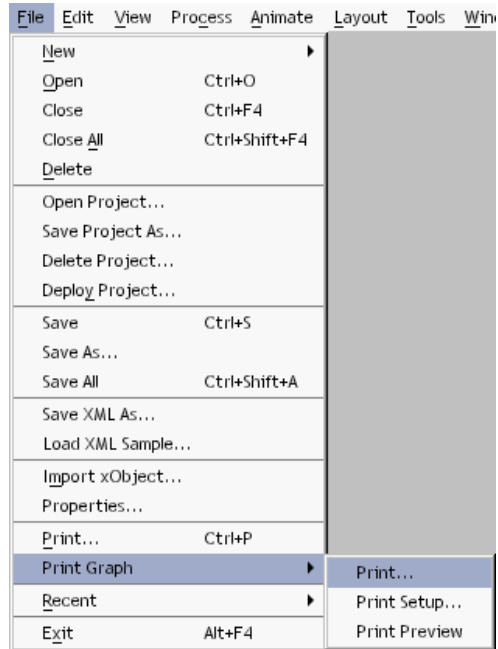
Shapes

The Shapes tool will let you put rectangles, ovals, or your own .jpg or .gif graphics anywhere on the canvas. These elements are strictly decorative and have no effect on process runtime dynamics.

Menu Commands

In Composer, when the Process Manager is the front editor, a number of process-specific menu commands appear in Composer's menus. The **File**, **View**, **Process**, and **Layout** menu structures are illustrated and discussed below.

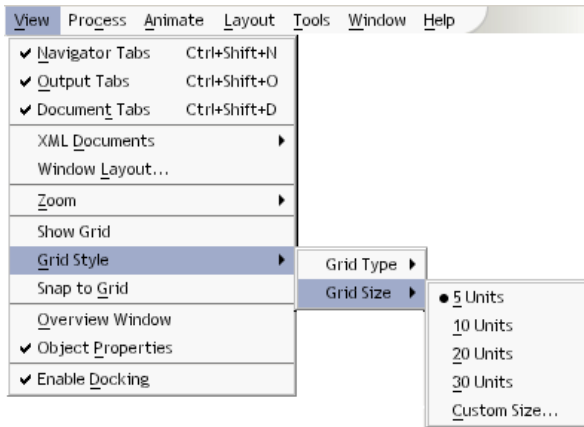
File menu:



The only addition to Composer's File menu is the **Print Graph** command.

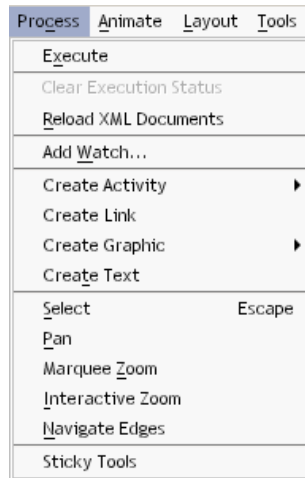
Menu	Submenu	Command	Description
File	Print Graph	Print	This selection allows you to print the complete or selected graph and descriptions
File	Print Graph	Print Setup	This selection allows you to determine what portion of the process should be printed – see dialog below for more information.
File	Print Graph	Print Preview	This selection allows you to preview the selected items before printing.

View menu:



Menu	Submenu	Command	Description
View	XML Documents		Brings up submenu allowing you to change visibility, order and view of XML documents
View	Zoom	Many	The Zoom tool lets you to set the view magnification (on a percentage basis) for the canvas. Several preset values are available via pulldown menu. You can also specify any arbitrary percentage by selecting Custom Zoom.
View	Show/Hide Grid		Toggles the grid's visibility (see below).
View	Grid Style	Grid Type	You can choose to either have a blank background or (in conjunction with the Grid Size option) a grid view. The default is None.
View	Grid Style	Grid Size	When in grid view mode, this command sets the spacing between lines or dots.
View	Snap to Grid		Align process objects to grid lines
View	Overview Window		Toggles the visibility of the Overview pane ("bombsight view") while creating or editing a layout.
View	Object Properties		Toggles the visibility of the Object Properties pane while creating or editing a layout. This pane is where data mappings (messages) are specified.
View	Enable Docking		Allows modal windows described above to be docked if they are brought near an edge of the graph. The default is On.

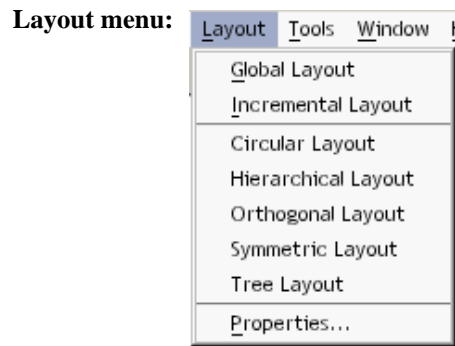
Process menu:



.The Process Menu commands are explained below.

Menu	Submenu	Command	Description
Process	Execute		Runs a process from start to finish.
Process	Clear Execution Status		This menu command duplicates the functionality of the Reset button on the far right side of the toolbar; it resets the graphics state(s) of all icons, links, etc. to the original pre-animation state(s).
Process	Reload XML Documents		Performs the same function as Reload in the Component menu item when in a Component Editor.
Process	Add Watch		Allows you to identify certain data items and examine their data values during the execution of a component as a debugging aid.
Process	Create Activity	Composer Component	Changes the active tool to the Activity tool and configures it so that a click on the canvas will create a new Component Activity.
Process	Create Activity	Subprocess	Changes the active tool to the Activity tool and configures it so that a click on the canvas will create a new Subprocess.
Process	Create Activity	Synchronize Subprocesses	Changes the active tool to the Activity tool and configures it so that a click on the canvas will create a new Synchronize Subprocesses Activity.
Process	Create Activity	Web Service Receive	Changes the active tool to the Activity tool and configures it so that a click on the canvas will create a new Web Service Receive Activity.
Process	Create Activity	Web Service Send	Changes the active tool to the Activity tool and puts the tool in Composer Component mode so that a click on the canvas will create a new Web Service Send Activity.
Process	Create Link		Changes the active tool to the Link tool.
Process	Create Graphic	Rectangle	Changes the active tool to the Graphics tool and configures it so that a click on the canvas will create a resizable rectangle.

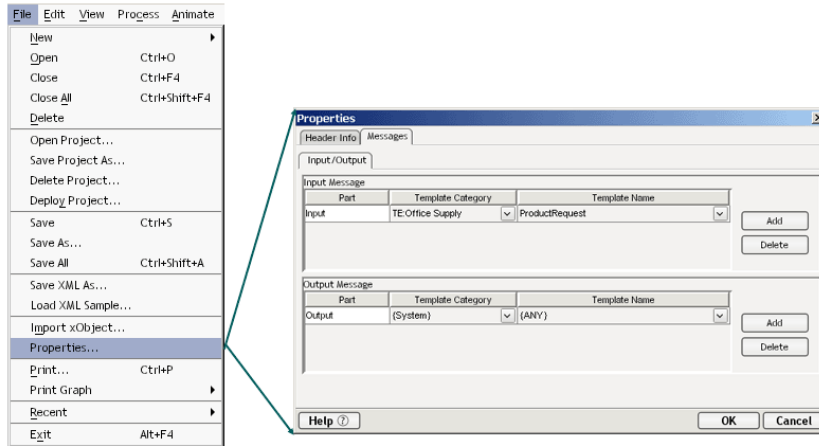
Process	Create Graphic	Oval	Changes the active tool to the Graphics tool and configures it so that a click on the canvas will create a resizable oval.
Process	Create Graphic	Rounded Rectangle	Changes the active tool to the Graphics tool and configures it so that a click on the canvas will create a resizable, rounded rectangle.
Process	Create Graphic	Diamond	Changes the active tool to the Graphics tool and configures it so that a click on the canvas will create a resizable diamond shape.
Process	Create Graphic	Picture	Changes the active tool to the Graphics tool and configures it so that a click on the canvas will cause an image file (.jpg or .gif) to be placed. You can specify the actual image file in the UI tab of the canvas's property sheet (see end of this chapter).
Process	Create Text		Changes the active tool to the Text tool.
Process	Select		Changes the current tool to the arrow cursor (for selection of graph items).
Process	Pan		Changes the current tool to the Hand tool to allow canvas panning for fast navigation of large graphs.
Process	Marquee Zoom		This option is useful only when the Overview Window (View > Overview Window) is in view. When this option is active, you can click outside the blue marquee box to zoom the canvas to larger magnification.
Process	Interactive Zoom		Similar to the above, but allows you to drag the corner handles of the marquee box (blue box) to "resize the view."
Process	Navigate Edges		Changes the active tool in such a way that you can click on any activity and see the graph animate (without executing any activities) along link paths. No executables are run.
Process	Sticky Tools		Allows you to select a tool once and have it remain the selected tool. This will allow you to drop multiple activities on the canvas or draw multiple links without selecting the Link tool multiple times.



Menu	Command	Description
Layout	Global Layout	Default layout mode: Entire graph is cached in memory at design time.
Layout	Incremental Layout	Optional layout mode that uses memory more efficiently.
Layout	Circular Layout	Arranges nodes in a hub-and-spoke manner. See discussion elsewhere under "Layout Properties".
Layout	Hierarchical Layout	Applies the familiar "organizational chart" style of diagramming, in which top-down relationships are emphasized.
Layout	Orthogonal Layout	Constrains nodes and links to a row-and-column motif.
Layout	Symmetric Layout	Edge crossings are minimized and node distributions are made uniform so that <i>symmetrical relationships</i> are emphasized.
Layout	Tree Layout	Applies the familiar "family tree" layout to a graph, similar to the hierarchical style described above, except that links are not parallel and seldom run perfectly horizontal or vertical.
Layout	Properties	Brings up a preferences dialog for fine-tuning the above settings.

Process Properties

General info for a whole process can be accessed via **File > Properties**. The dialog that appears has two tabs, Header Info and Messages. The Header Info tab gives Name and comment-type information about the process in question. The Messages tab contains XML Template information for the input and output messages of the process.



Object Properties

Each type of object depicted in a directed edge graph created in Process Designer has its own set of properties. The properties are context-sensitive: they vary according to the type of object that you have selected on the canvas. To see the current properties for any object, simply select an object (by clicking on it using the Pointer tool) and toggle **Object Properties** under the **View** menu (if the Object Properties palette is not already visible).

The Object Properties palette (equivalently referred to as the *property sheet* for an object) is where you can specify such important activity attributes as:

- ◆ Exit Condition
- ◆ Join Condition
- ◆ Timeout
- ◆ Retry Count
- ◆ Retry Interval
- ◆ Map Policy
- ◆ And more (see below)

The following sections describe what the property sheets for the various process elements look like when the appropriate type of object has focus.

Activity Properties

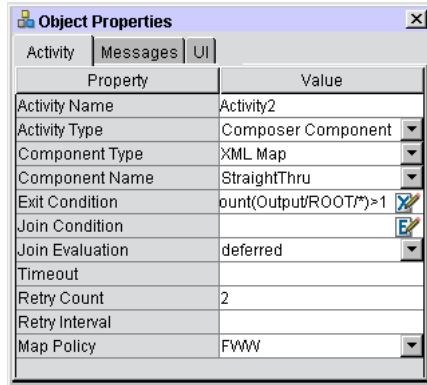
The Process Manager supports five activity types, each with its own set of object properties: Composer Component, Web Service Send, Web Service Receive, Subprocess, and Synchronize Subprocesses. The property sheets for each are discussed in some detail below.

Composer Component

The Component Activity Object Properties panel has three tabs: Activity; Messages, and UI. Their appearances are illustrated below; their functionality is discussed in the tables that follow.

Note that all Object Properties tabs and panels are *context-aware*: Their contents update automatically to reflect the attributes of the activity that you have selected on the canvas. Likewise, any changes you make in any of the property settings will take effect in real time, as soon as the field in question loses focus. (You may have to click outside of a property field in order for a change to take effect.)

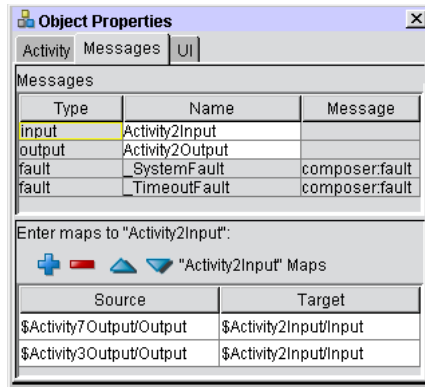
Activity Tab



Property	Control Type	Usage
Activity Name	Text field	This is the name shown under the activity icon on the canvas.
Activity Type	Dropdown	A dropdown list allows you to change the activity type of the currently selected activity. The dropdown shows the five categories of activity type.
Component Type	Dropdown	A dropdown displays a list of available Composer Component types (XML Map, Web Service, JDBC Component, and so on).
Component Name	Dropdown	The dropdown displays a list of Component Names corresponding to any components of the chosen Component Type (above) that you have already built in the current project.
Exit Condition	Text Field	<p>An Exit Condition is a Boolean XPath expression, the purpose of which is to determine whether the Activity has finished normally. The Exit Condition's expression can refer to the output message of the Activity or to output of any activity that ran before the Activity on the same control path.</p> <p>If the Exit Condition evaluates to <i>true</i>, the activity is treated as "Complete." If the Activity is complete, the process resumes normal flow of control; otherwise, the Activity is executed again.</p> <p>The Activity will be executed X number of times where X is the Retry Count defined below.</p> <p>The Retry Interval defines the time between execution retries.</p>

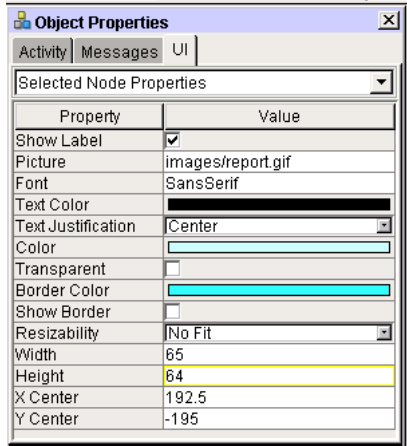
Join Condition (appears only on join targets)	Text Field	<p>A Join Condition is a Boolean expression in simple OR/AND/NOT syntax, the purpose of which is to synchronize parallel work based on the truth values of incoming links.</p> <p>An Activity is called a Join Activity if it has more than one incoming link. The Join Activity will fire if and only if the Join Condition is true. The default, if no condition is explicitly specified, is true.</p>
Join Evaluation (appears only on join targets)	Dropdown	The choices are Deferred and Immediate. For the meaning of these options, see Chapter 1.
Timeout	Text Field	<p>The Timeout attribute defines a time interval in which an Activity must complete its work. Once a time-out occurs, the Retry Count (if any) will apply and the activity will be reexecuted.</p> <p>After a timeout occurs, the Process Server will wait a certain length of time (specified in the Retry Interval) before kicking off the next retry. The Retry Interval, below, defines the wait time before an activity can be retried.</p> <p>Timeout and Retry settings are optional. The default is zero retries and a retry interval of zero.</p>
Retry Count	Text Field	The number of times to retry an Activity.
Retry Interval	Text Field	The length of time to wait between retries, should a retry be necessary.
Map Policy	Text Field	<i>Last Writer Wins, First Writer Wins, or Map Order.</i> Note that this value is important only when there is the potential for two activities to overwrite each other's data (i.e., two source activities contend for the same XPath locations in the target activity's input message).

Messages Tab



Property	Control Type	Usage
Messages	Three columns: <ul style="list-style-type: none"> Type (non-editable) Name Message (non-editable) 	If WSDL exists, "Type" and "Message" are pulled from the WSDL's Port Type Operations Input and Output elements. "Name" defaults to the default Activity Name appended by the type (e.g. <i>Activity2Output</i>).
Maps:	+ and - icons	Add and Delete mappings from last activity's output to current input.
Source	Button	The Source XPath expression (applies to output from previous activity in the graph)
Target	Button	The Target XPath expression (applies to currently selected activity's input)

UI Tab



Property	Control Type	Usage
Show Label	Checkbox	Determines whether a text label (name) appears below the currently selected activity object.
Picture	Text Field	The path to the image (Gif or JPEG) that will be used for the display of the currently selected activity object. Use this to point to custom icon art, if desired. (This is for design time only. Your art will not be deployed in any jar files.)
Font	Dialog appears	Clicking on the Value field causes the "Choose Font" dialog to be displayed. This dialog has three dropdowns which allow for the selection of a font, style (Plain, Bold, Italic, Bold Italic) and point size.
Text Color	Color picker	Displays the color to be used for text associated with the current object. Clicking on this bar causes a color picker dialog to appear.
Text Justification	Dropdown menu	Left Center (default) Right
Color	Color picker	This is the background color for the selected object. Clicking on this bar causes a color picker dialog to appear.
Transparent	Checkbox	Checked = Transparent object, Unchecked = Opaque
Border Color	Color picker	Border color for the selected object. Clicking on this bar causes a color picker dialog to appear.
Show Border	Checkbox	Checked = Border Displayed; Unchecked = Border not displayed.

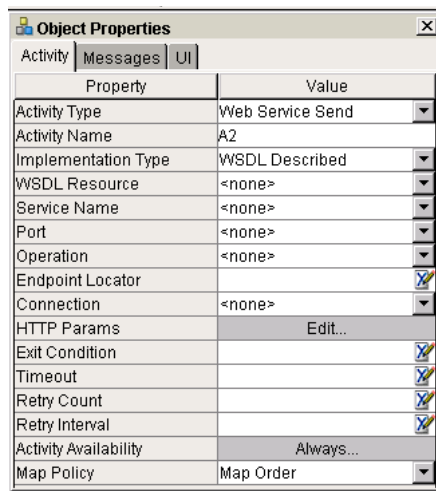
Resizability	Dropdown:	Choices are: No Fit Tight Fit Tight Width Tight Height Tight Fit Preserve Aspect Preserve Aspect
Width	Text Field	Item width. 40.0 (Default)
Height	Text Field	Item height. 32.0 (Default)
X Center	Text Field	Position X coordinate
Y Center	Text Field	Position Y coordinate

Web Service Send

The Web Service Send activity has its own unique object properties, which are reflected in the Activity tab on the Object Properties panel.

NOTE: The Messages and UI tabs for this activity are the same as for the Component activity described above. Only the Activity tab will be described below.

Web Service Send Activity Tab



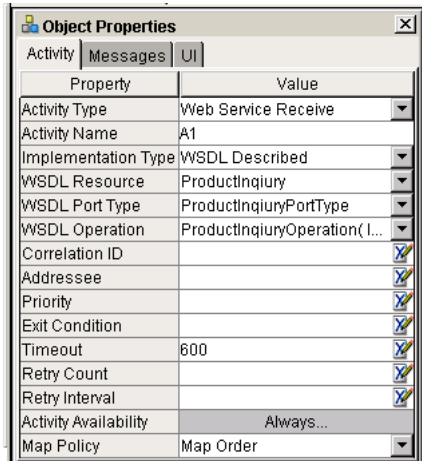
Property	Control Type	Usage
Activity Type	Dropdown	A dropdown list of Activity Types
Activity Name	Activity 1...n (default)	The name of the Activity.
WSDL Resource	Dropdown	A dropdown list of the available WSDL Resources within the Composer project.
Service Name	Dropdown	A dropdown list of the available Web Services within the WSDL Resource.

Service Port or Binding	Dropdown	A dropdown list of the Binding Names within the WSDL Resource.
Operation	Dropdown	A dropdown list of the Operation Names within the WSDL Resource.
Endpoint Locator	XPath Expression	Enter the Endpoint Location (usually a URL pointing at a servlet) for the Web Service you wish to use, wrapped in quotation marks. (Alternatively, enter an XPath expression that will evaluate to an Endpoint Location at runtime.)
Connection	Connection	A dropdown list of Connections.
HTTP Params	Pushbutton	This displays the 'HTTP Header Parameters' dialog, where you can specify content-length and other common HTTP parameters.
Exit Condition	Text Field	See discussion under "Exit Condition", page 95.
Join Condition (as applicable)	Text Field	See discussion under "Join Condition", page 95.
Join Evaluation	Dropdown	Like the Join Condition field, this field will only appear when the target activity is a join activity. The dropdown choices (Immediate, Deferred) determine the join's evaluation mode.
Timeout	Text Field	See discussion under "Exit Condition" on page 95.
Retry Count	Numeric Field	See discussion under "Retry Count", page 96.
Retry Interval	Text Field	See discussion under "Retry Interval", page 96.
Map Policy	Text Field	See discussion under "Map Policy", page 96.

Messages and UI Tabs for Web Service Send

The settings on these tabs work the same as described for the Component Activity (already discussed).

Web Service Receive



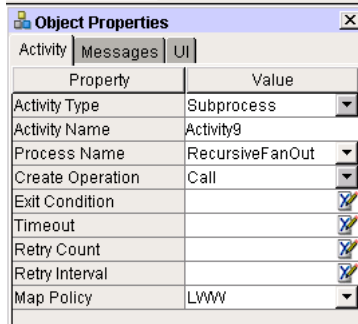
Web Service Receive Activity Tab

Property	Control Type	Usage
Activity Type	Dropdown	A dropdown list of available Activity Types
Activity Name	Activity 1...n (default)	The name of the Activity.
Implementation Type	Dropdown	One of: Web Service, JMS Service, or External.
WSDL Resource	Dropdown	A dropdown list of the available WSDL Resources within the Composer project.
WSDL Port Type	Dropdown	The port type for this service.
WSDL Operation	Dropdown	A dropdown list of the Operation Names within the WSDL Resource.
Correlation ID	Text Field	Arbitrary user-defined value, used to uniquely identify a transaction
Addressee	Text Field	Arbitrary string label, typically to define the "owner" (name of an individual) associated with this particular transaction or activity
Priority	Text Field	Some arbitrary numeric value relating, typically, to the importance of this activity or work item
Exit Condition	Text Field	See discussion under "Exit Condition", page 95.
Join Condition (as applicable)	Text Field	See discussion under "Join Condition", page 95.
Timeout	Text Field	See discussion under "Exit Condition" on page 95.
Retry Count	Numeric Field	See discussion under "Retry Count", page 96.
Retry Interval	Text Field	See discussion under "Retry Interval", page 96.
Map Policy	Text Field	See discussion under "Map Policy", page 96.

Messages and UI Tabs for Web Service Receive

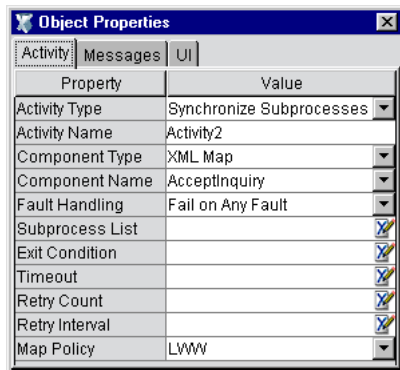
The settings on these tabs operate the same as described earlier for the Component Activity (see above).

Subprocess



All properties on all tabs of the Object Properties panel for Subprocess have exactly the same names (and operate the same way) as for the Composer Component properties, except for the *Create Operation* property, which is one of *spawn* or *call*, to reflect whether the subprocess should be invoked asynchronously (“fire and forget”) or synchronously (poll until response comes).

Synchronize Subprocesses



The Synchronize Subprocesses activity type is a specialized activity that coordinates the “fan-in” of multiple results from fanned-out subprocesses. See the discussion of “Synchronize Subprocesses Activity” in a later chapter.

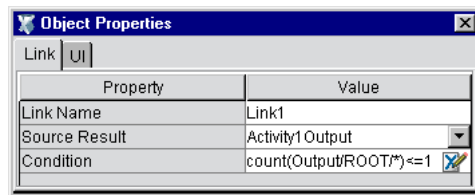
Property	Control Type	Usage
Activity Type	Dropdown	A dropdown list of available Activity Types
Activity Name	Activity 1...n (default)	The name of this Activity.
Component Type	Dropdown	A list of available components in this Composer project.
Fault Handling	Dropdown	Two choices: Fail on Any Fault, or Fail If All Fail.

Subprocess List	Text Field (XPath)	XPath locations of the ProcessInfo data for fanned out subprocesses.
Exit Condition	Text Field	See discussion under “Exit Condition”, page 95.
Join Condition (as applicable)	Text Field	See discussion under “Join Condition”, page 95.
Timeout	Text Field	See discussion under “Exit Condition” on page 95.
Retry Count	Numeric Field	See discussion under “Retry Count”, page 96.
Retry Interval	Text Field	See discussion under “Retry Interval”, page 96.
Map Policy	Text Field	See discussion under “Map Policy”, page 96.

Link

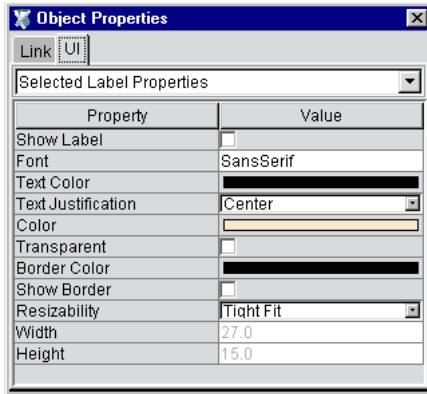
The Link Object Properties has two tabs: Link and UI.

Link Tab



Property	Control	Usage
Link Name	Text Field	The link’s name. This name is also used in join-condition expressions.
Source Result	Dropdown	Designates the source activity of the link.
Condition	Text Field	Specifies the XPath condition for the link.

UI Tab for Links



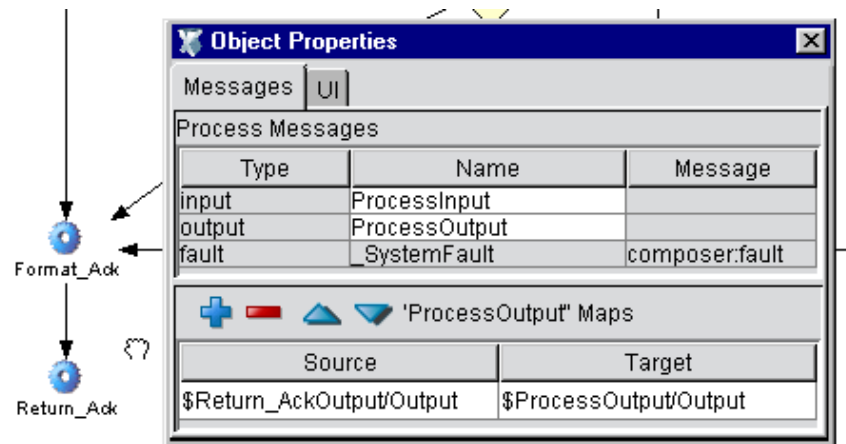
Property	Control	Usage
Show Label	Checkbox	Toggles the visibility of the link name.
Font	Text Field	Clicking this field will cause a dialog to appear. In the dialog, you can set various font properties.
Text Color	Color Picker	Allows you to set the color of the text (link name) associated with a link.
Text Justification	Dropdown Menu	Center, Left Justify, Right Justify.
Transparent	Checkbox	Toggles the link's transparency on/off.
Border Color	Color Picker	Allows you to select the color of the outline of the link.
Show Border	Checkbox	Toggles the border (draw/no-draw).
Resizability	Dropdown Menu	Allows you to specify various link drawing policies.
Width	Text Field	Allows you to specify the overall width of the link.
Height	Text Field	Allows you to specify the overall height of the link.

Graph Object Properties

The Process Object (or graph) property sheet has a Messages tab and a UI tab. To see the graph's properties, click anywhere on the bare canvas, then bring the Object Properties palette into view (use the **View** menu's **Object Properties** command). You will use this window to set overall process input, output, and fault message mappings, and customize the appearance of the graph.

Process Messages Tab

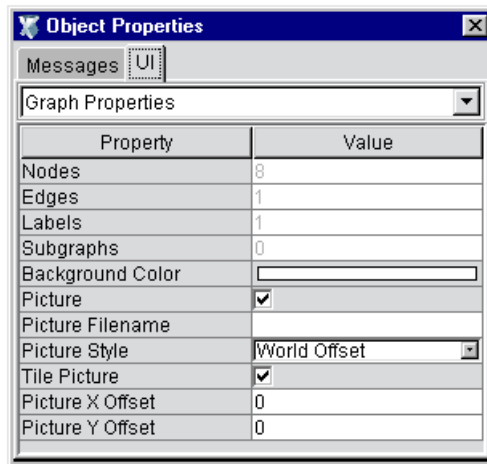
The Messages tab is where you will typically specify the end-activity-to-process-output data mapping(s). For example, if Activity4 on your graph is the *end activity* for the process (the final activity to execute), and you want the process to return a message containing Activity4's output, this is where you would specify the ProcessOutput mapping. See example below.



In the above example, the activity *Return_Ack* is the end activity for the process. Its output is mapped to `$ProcessOutput/Output`, as shown at the bottom of the Object Properties pane. The first step in setting up this pane was to *click on the bare canvas* (thus deselecting all activities, links, and other graphic elements). This makes the Object Properties pane reflect the properties of the *process-as-a-whole*. (Notice the input and output messages are simply *ProcessInput* and *ProcessOutput*.)

Graph UI Tab

The graph UI tab has two purposes: It allows you to define custom appearance-related settings for the overall graph, and it provides summary information about the number of nodes on the graph, the number of links, labels, etc.



Remember that this set of properties is reachable only when you click on bare canvas.

NOTE: For additional information about how to customize the appearance of a graph, see the section “Layout Properties” further below.

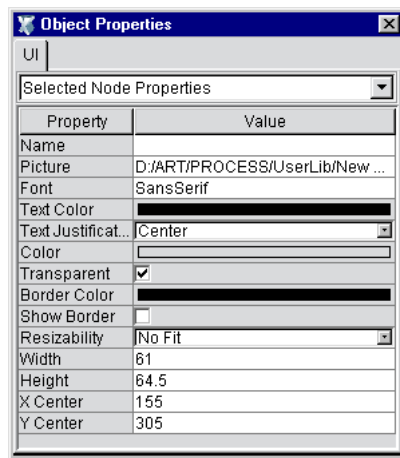
The following table describes the properties available in the UI tab of the process Object Properties panel.

Attribute	Value	Description
Nodes	0—n	This field is not editable. It provides a numeric count of the graph's Nodes.
Edges	0—n	This field is not editable. It provides a numeric count of the graph's Edges.
Labels	0—n	This field is not editable. It provides a numeric count of the graph's Labels.
Subgraphs	0—n	This field is not editable. It provides a numeric count of the graph's Subgraphs.
Background Color	Displays the color itself. The default is white.	This is the background color.
Picture	Check Box	Checked = display a picture on the graph, Not Checked = don't display a picture on the graph.
Picture Filename	The filename of the Picture.	The full path name of the picture file which may be a JPEG or a Gif.
Picture Style	World Offset or Device Offset	World Offset displays the picture in the middle of the diagram. Device Offset displays the picture at the offset defined by the Picture X Offset and the Picture Y Offset.

Tile Picture	Check Box	Checked = tile the picture, Not Checked = display the picture at the offset.
Picture X Offset	0.0	Used to change the onscreen x-offset.
Picture Y Offset	0.0	Used to change the onscreen y-offset.

Selected Node Properties on UI Tab

The Selected Node Properties UI tab is for inspecting or setting *appearance attributes* on objects shown on the graph. Single-click an object to select it, then select the UI tab from the Object Properties pane; then choose Selected Node Properties from the dropdown menu control at the top of the tab. See below.



UI Tab (Selected Node Properties)

Attribute	Value	Description
Name	Activity1...n (Default)	This is the name of the Activity. It defaults to Activity1...n.
Font	Dialog <ul style="list-style-type: none"> • SanSerif (default) • Serif • MonoSpaced • DialogInput 	Clicking on the Value field causes the 'Choose Font' dialog to be displayed. This dialog has three dropdowns which allow for the selection of a font, font style (Plain, Bold, Italic, Bold Italic) and Font Size.
Text Color	Displays the color itself. The default is black.	Click on the Value field causes the 'Choose Color' dialog to be displayed.
Text Justification	Left Center (default) Right	This is a dropdown.
Color	Displays the color itself. The default is yellow.	This is the background color. Click on the Value field causes the 'Choose Color' dialog to be displayed.

Transparent	Checkbox	Checked = Transparent, Unchecked = Opaque
Border Color	Displays the color itself. The default is black.	Click on the Value field causes the 'Choose Color' dialog to be displayed.
Show Border	Checkbox	Checked = Border Displayed; Unchecked = Border not displayed.
Resizability	Dropdown: <ul style="list-style-type: none"> • No Fit • Tight Fit • Tight Width • Tight Height • Tight Fit Preserve Aspect • Preserve Aspect 	
Width	Text Field	40.0 (Default)
Height	Text Field	40.0 (Default)
X Center	Text Field	X coordinate
Y Center	Text Field	Y coordinate

Text Object Properties

The UI tab for Text objects, Shapes, etc., has a Selected Node Properties pane with attributes similar to those described above. The table below describes the properties in detail.

UI Tab

Attribute	Value	Description
Name	Untitled	This is the Name of the text object as well as the Text/Caption/Label itself.
Margin Width	3.0 (default)	This is the width of the margin to the left and right of the text.
Margin Height	1.0 (default)	This is the height of the margin to the top and bottom of the text.
Font	Dialog <ul style="list-style-type: none"> • SanSerif (default) • Serif • MonoSpaced • DialogInput 	Clicking on the Value field causes the 'Choose Font' dialog to be displayed. This dialog has three dropdowns, which allow for the selection of a font, font style (Plain, Bold, Italic, Bold Italic) and Font Size.
Text Color	Displays the color itself. The default is black.	Click on the Value field causes the 'Choose Color' dialog to be displayed.

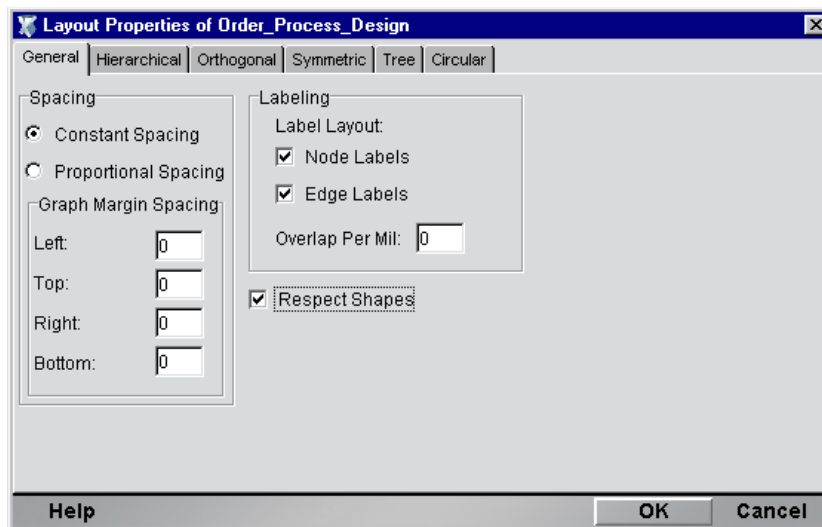
Text Justification	Left Center (default) Right	This is a dropdown.
Color	Displays the color itself. The default is white.	This is the background color. Click on the Value field causes the 'Choose Color' dialog to be displayed.
Transparent	Checkbox	Checked = Transparent, Unchecked = Opaque
Border Color	Displays the color itself. The default is black.	Click on the Value field causes the 'Choose Color' dialog to be displayed.
Show Border	Checkbox	Checked = Border Displayed; Unchecked = Border not displayed.
Resizability	Dropdown: <ul style="list-style-type: none"> ◆ No Fit ◆ Tight Fit ◆ Tight Width ◆ Tight Height ◆ Tight Fit & Preserve Aspect ◆ Preserve Aspect 	
Width	Text Field	48.0 (Default) This field is not enabled. The width will change as the text is changed from the default 'untitled' and as the margin widths and fonts are changed.
Height	Text Field	19.0 (Default) This field is not enabled. The height will change as the margin height and fonts are changed.
X Center	Text Field	X coordinate
Y Center	Text Field	Y coordinate

Layout Properties

Process Designer will (if you wish) automatically reformat your graph according to any of five flow-diagramming algorithms:

- ◆ **Circular**—Arranges nodes in a hub-and-spoke manner whenever possible, with spokes having identical lengths. This type of layout is appropriate when *clustering* is the predominant architectural feature (e.g., as in depicting a LAN or WAN layout).
- ◆ **Hierarchical**—This is the familiar “organizational chart” style of diagramming, in which top-down relationships are emphasized. (You can, however, configure this layout option to show left-to-right or other flow polarities.) This layout option is appropriate for graphs in which *hierarchical relationships* need to be emphasized.
- ◆ **Orthogonal**—This style constrains nodes and links to a row-and-column motif. Links are constrained to run parallel to x- and y-axes. Also note that nodes with more than one incoming link may be magnified in appearance relative to other nodes. This layout strategy is appropriate for situations where a *grid or lattice* relationship between elements needs to be emphasized, as opposed to hierarchical relationships.
- ◆ **Symmetric**—In this style of graphing, edge crossings are minimized and node distributions are made uniform so that *symmetrical relationships* can be emphasized.
- ◆ **Tree**—This diagram style is appropriate when the predominant need is to show parent/child relationships. It uses the familiar “family tree” type of layout, very similar to the hierarchical style described above, except that links are not parallel and seldom run perfectly horizontal or vertical.

All of the above diagramming styles can be extensively customized by means of preferences exposed in the **Layout > Properties** dialog. To bring up this dialog, go to Composer’s main menubar and choose **Properties** from the **Layout** menu.



The dialog has six tabs: a General Preferences tab, and five tabs corresponding to the five autolayout styles just described. Each tab contains a wealth of controls and settings to allow you to exercise fine control over the many constraints that characterize a particular style of graphing.

General Layout Tips

The following tips are aimed at helping you achieve maximum productivity with Process Designer.

Snap and Grid Behavior

- ◆ By default, everything you draw or position snaps to an invisible 5-pixel-by-5-pixel grid. But you can override this behavior at any time *by holding the Alt key down*. (You can override it permanently by setting Grid Size to one, using **View > Grid Style > Grid Size > Custom Size**.)
- ◆ You can toggle the visibility of the grid by using the **View** menu's **Hide Grid** or **Show Grid** commands. (There is actually only one command; its name changes dynamically depending on which mode you just entered.) Grid Size and Style (dots versus lines) can also be set at any time through View menu commands .
- ◆ You can instantly align graph nodes to the grid, at any time, by using **View > Snap to Grid**. You will see graph items suddenly “jump” to the closest grid lines.

Multiple Undo

Multiple Undo/Redo is available for all layout gestures.

Sticky Tools

Normally, a tool reverts to the arrow cursor after one use. For example, if you select the Activity Tool, then click on the canvas to put down a new activity icon, the tool will immediately revert to the arrow (or Selection Tool) when you let go of the mouse. You can override this behavior and make the tool mode persist across mouse clicks by turning on the Sticky Tools option. Look under the **Process** menu for **Sticky Tools**.

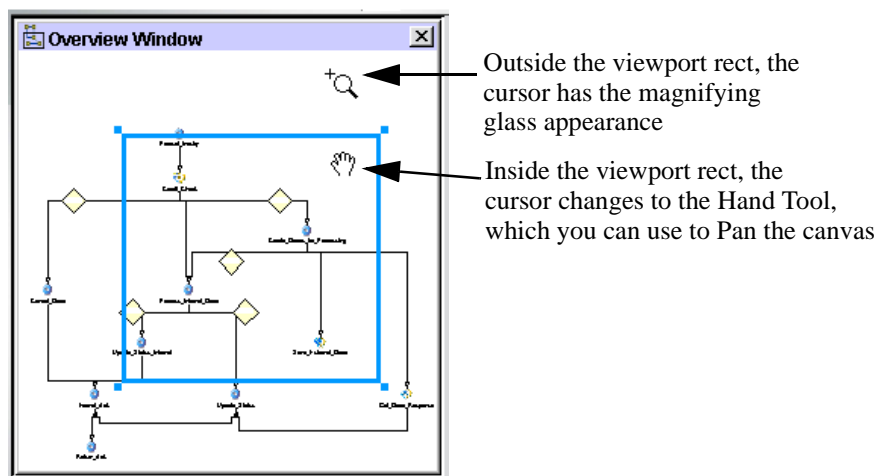
Overview Window

Exceptional control over pan and zoom can be had by using the Overview Window (see below). Toggle this pane's visibility by using **View > Overview Window**.

Two behaviors are available from the overview window:

- ◆ You can drag the blue “viewport rect” around the overview pane to pan the canvas in real time.
- ◆ You can click-drag just outside the viewport rect to interactively zoom the canvas to a bigger or smaller size.

Notice that the cursor changes appearance depending on the position of the mouse (inside or outside the viewport rect).



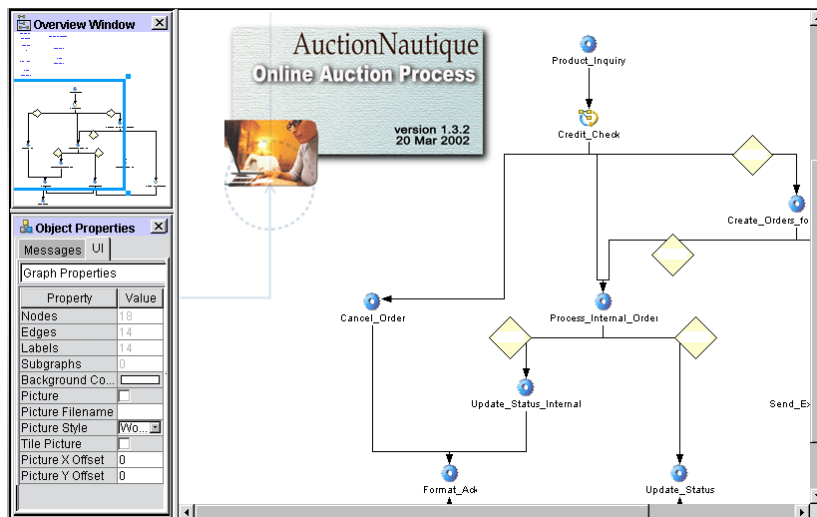
Customizing the Canvas

Note that you can customize the canvas in various ways. For example, you can specify a background image; change the appearance of any activity to use a custom image; and/or add any number of decorative images or logos to the canvas, and use Send to Back or Bring to Front to “stack” images in any order. These features allow you to build presentation-quality process graphs for use in meetings, demonstrations, etc.

NOTE: To access canvas properties, click anywhere on bare canvas, then choose the UI Tab in the Object Properties panel.

Using Custom Backgrounds

One way in which the canvas can be customized is to add a custom background, consisting of a **.gif** or **.jpg** image. The following illustration shows a canvas that contains a **.jpg** background.

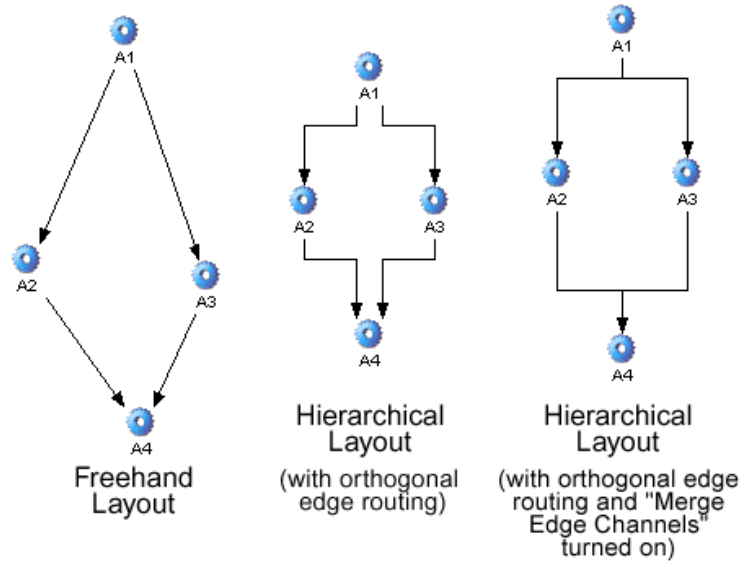


➤ To add a picture to a canvas:

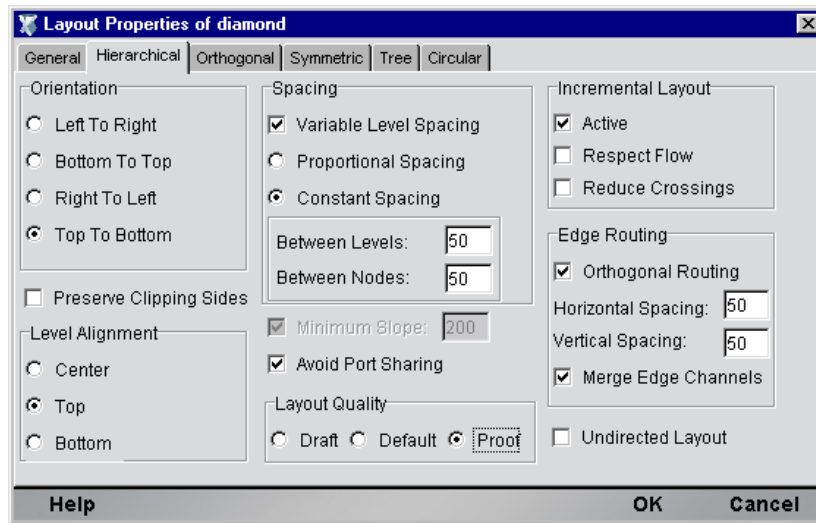
- 1 Click anywhere on bare canvas.
- 2 Toggle the **Object Properties** panel into view.
- 3 Choose the **UI Tab**.
- 4 Click the white area to the right of **Picture Filename**. A navigation dialog will appear.
- 5 Navigate your hard disk or network and find a **.jpg** or **.gif** file that you wish to use as a graph background picture.
- 6 In the UI Tab, check the **Picture** checkbox to apply the image to the canvas.
- 7 Optionally check the **Tile Picture** checkbox if you wish to tile the canvas with the image.
- 8 Next to **Picture Style** you will find a dropdown menu. Select one of the two choices available on this menu:
 - ◆ **World Offset**—Choosing this option means that the image will shrink or grow with the canvas as you choose different zoom settings and maintain its relative position to other objects on-canvas. This is the normal behavior for all Process Designer graphics.
 - ◆ **Device Offset**—Choosing this option means that the image will not shrink, grow, nor change position as you pan or zoom.
- 9 Optionally adjust **Picture X-Offset** and/or **Picture Y-Offset** values to place the picture exactly where you want it on the graph. (You may enter positive *or* negative values here as required.)

Autolayout Options

As explained earlier, Process Designer will reformat your graph according to various diagramming algorithms, if you desire. The auto-diagramming option you are most likely to use is the Hierarchical layout option. This option (**Layout > Hierarchical Layout**) will reformat a graph to a top-down (or left-to-right, or other) hierarchy view, with or without X/Y alignment of links, and with or without merging of parallel links.



Various constraint options are available for Hierarchical Layout (as for the other autolayout modes). To access the settings, use the **Layout** menu's **Properties** command, which brings up the Layout Properties dialog:



Take special note of the Edge Routing control group at the lower right. You must check the **Orthogonal Routing** checkbox if you want links to be X/Y-axis aligned. If you want stems of parallel links (coming into or out of a common node) to be depicted as a *single* stem, you should check the **Merge Edge Channels** checkbox.

5

Advanced Topics

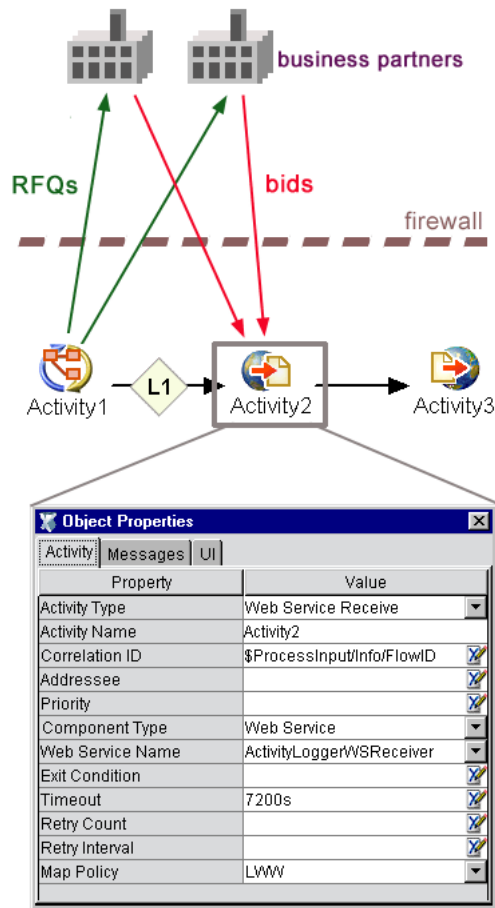
This chapter discusses concepts and scenarios that go beyond the simple “straight-through processing” use cases that have been discussed so far. In particular, we will examine the Web Service Receive activity and the Synchronize Subprocesses activity. The Web Service Receive activity is useful in implementing design patterns that rely on incoming notifications or requests as part of an ongoing process. The Synchronize Subprocesses activity, on the other hand, is useful for collecting and resynchronizing the results from a previous fan-out (or parallel division of workflow to multiple subprocess instances) by an upstream activity.

To get the most out of this chapter, you should be familiar with WSDL-based Web Services, Composer action models, and concepts involving message mapping, fault messages, and link logic.

Web Service Receive

The Web Service Receive activity type allows you to implement the WSDL Request-Response and One-Way port type patterns. These are patterns in which the “endpoint” activity (representing the Web Service that will fire) does nothing until triggered by an incoming request. The target activity’s implementation is, in this sense, passive—unlike the Notification and Solicit-Response scenarios, in which the underlying service is the *requestor* instead of the *requestee*.

The Web Service Receive activity must fulfill all the normal obligations of an activity in a process model. That means it has to be able to function as a link target, with timeout and retry behavior, fault behaviors, etc.



In this example, Activity 1 (a Subprocess activity) “fires” Activity2 (a Web Service Receive activity) via link L1. When and if Activity2 exits with a condition of *true*, its outgoing link (to Activity3) will be followed, but not until then. If Activity2 does not come back with an exit condition of *true* within the Timeout period (in this case, 7200 seconds: two hours), Activity2 will generate a `_TimeoutFault`.

A key concept to understand is that the runtime engine doesn’t *run* the service under Activity2. It merely provides appropriate input messages (as with any activity) at the proper time and collects the output message at the appropriate time. *Incoming requests to the server* cause the Web Service Receive’s *implementation* to be invoked or run through appropriate triggering mechanisms (involving servlets, JMS listeners, or whatever), independently of the process engine itself. In other words, the web-service app that underlies the WSR activity is just a web service on a server, like any other web service, and its URL might be hit at any time, but the process engine only cares about (and will only respond to) that web service *within the context of a given process*, with all its timeout constraints, etc. Should a business partner hit the URL when the WSR activity is not active, the partner will likely just get a SOAP fault message back.

A typical WSR usage might be one in which a process is designed to send requests to various vendors, collect the first valid response, and continue on to do some kind of processing. Using the pattern shown in the above diagram, the roundtrip scenario could look like this:

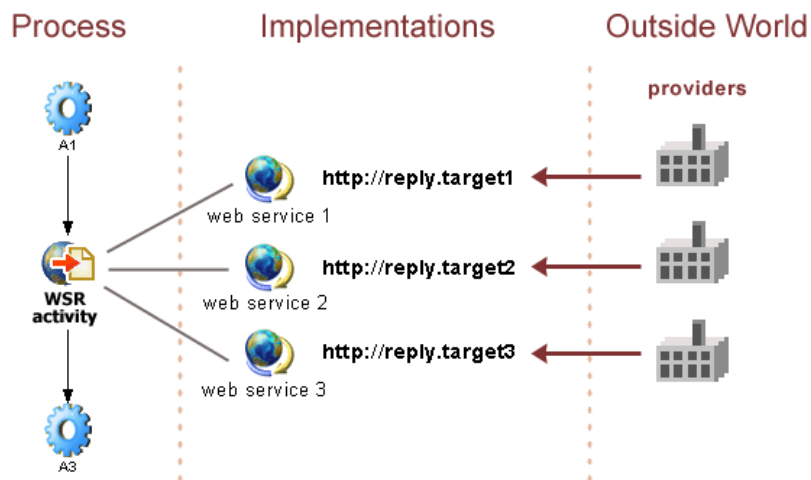
- ◆ The implementation for **Activity1** might be an app that issues a request for quote (RFQ) via *notifications* to one or more external business partners who have web services designed to handle such requests.
- ◆ **Activity2** might be configured in such a way that the notified business partners have two hours to reply with a quote. If no reply is received (from any partner) within two hours, the activity generates a Timeout Fault.

- ◆ **Activity2**'s underlying implementation might be a web service that wakes up the process engine immediately upon receiving a valid quote.
- ◆ **Activity2** exits immediately if a reply is handled (transferring control to the next activity); or else exits with a Timeout Fault after two hours. (We're disregarding the System Fault case for purposes of this example.)
- ◆ **Activity3** might notify a person or department (or another app, etc.) that a bid was received from so-and-so.

Note that this is not a fan-out/fan-in scenario, but a “first responder wins” type of scenario. If you were going to notify multiple partners *and collect multiple responses*, you would want to use the Synchronize Subprocesses Activity (described further below).

Multiple Implementations for a Single WSR Activity

It is possible to have multiple web services act as “the implementation” of a single Web Service Receive activity. This is because a Web Service Receive activity is built on top of a web service that waits to receive something—waits to be “hit.”



In the example shown above, a Web Service Receive activity (situated between two activities, A1 and A2) is able to respond to any of three different web services that have been deployed as implementations for the activity. When the WSR activity “fires,” the process simply waits for one of the three web services represented by URLs **reply.target1**, **reply.target2**, and **reply.target3** to receive input from a business partner. Each of the web services is a Composer application that contains a *Find Waiting Activity* action (as discussed in more detail in the next chapter). When one of the web services executes its Find Waiting Activity action—followed by a Release Waiting Activity action—the process continues on to the next activity, A3, assuming no fault conditions.

NOTE: If any business partner “hits” one of the three web services during a time when the WSR activity is not active (e.g., hasn’t been fired; or has fired and timed out), the partner will receive an error message of some kind. In most cases, this will be a SOAP fault.

Implementation Independence

The Process Manager imposes no restrictions on what the implementation for a Web Service Receive activity should look like. This is true for Web Services in general. The authors of WSDL put no limitations on how a Web Service should be implemented, and there are also no restrictions on the transport mechanism used. A Web Service needn’t use HTTP, for example, and payloads needn’t be passed via SOAP.

The Process Manager, likewise, allows your Web Service Receive activity’s implementation to take various forms: Composer web service, JMS service, or External (arbitrary implementation, not built in Composer). These choices are provided in a pull-down menu control on the Activity tab of the Object Properties dialog (see below).

Of course, for a Web Service to be a true Web Service, it should have a WSDL definition associated with it. Composer consults the WSDL when determining how to manage message maps for the Web Service Receive activity. In addition, since this is a Web Service *Receive* activity, the underlying service should implement either the WSDL One-Way or Request-Response port-type scenario. (The distinguishing characteristic of these two patterns is that the service implementing them is never the *initiator* of a transaction. The service is a *receiver*; you can think of the service as “listening” on a port.)

➤ **To use a Web Service Receive activity:**

- 1 Design and implement the Web Service that will serve as the activity implementation. It should have its own WSDL Resource. (For information on how to create services and WSDL Resources in Composer, consult the *eXtend Composer User’s Guide*).
- 2 If you created the Web Service inside another project, import the Web Service and its resources into the current project, which will contain your Process.
- 3 Create or open the Process in which you want to use the Web Service Receive activity.
- 4 Using the Web Service Receive variant of the Activity Tool (on the Process Designer toolbar), place a Web Service Receive activity icon on the process graph.
- 5 Draw links to and from the Web Service Receive activity the same way you would for any other kind of activity.
- 6 Bring the **Object Properties** pane into view (using **View > Object Properties**, if necessary).
- 7 In the **Object Properties** pane, click the **Activity** tab.
- 8 Next to **Component Type** in the properties list, use the pull-down menu to select one of **External**, **JMS Service**, or **Web Service**, as appropriate. See illustration.



- 9 Next to **Web Service Name**, use the pull-down menu to select the Web Service that you built in Step 1. (This list is prepopulated with the names of all Web Services in the current project.)
- 10 Set any other properties that you want to specify on the Activity tab.
- 11 Switch to the **Messages** tab.
- 12 Add any data mappings that you want to add, using the **Plus-sign** icon.
- 13 **Save** your work.

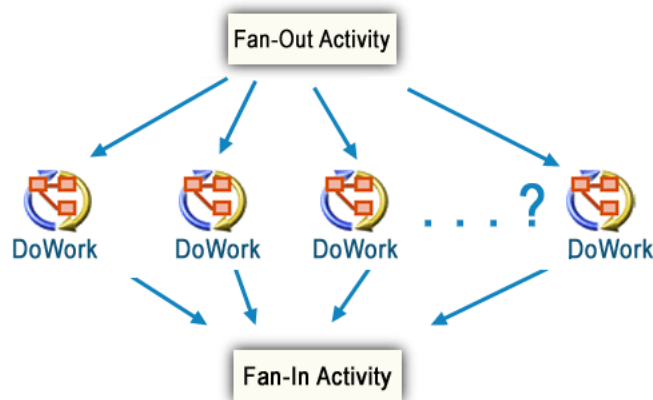
In order for a process to make use of the Web Service Receive activity, there must obviously be an underlying *implementation* consisting of a web service that communicates via the One-Way or Request-Response pattern(s) described in WSDL. This service must, in turn, be capable of communicating its “finished” status (and in most cases, some kind of XML data) back up to the process engine. Such communication requires the use of Find Waiting Activity and Release Waiting Activity actions in a service’s action model, as described in the next chapter.

NOTE: If you intend to use the Web Service Receive activity type, be sure to read about *Find Waiting Activity* and *Release Waiting Activity* actions in the next chapter.

Synchronize Subprocesses Activity

The Synchronize Subprocesses activity is similar to the Web Service Receive activity in that it, too, assumes an implementation that waits passively for incoming data and that may be invoked numerous times before it finally exits. Unlike the Web Service Receive activity, a Synchronize Subprocesses activity must use a Composer Component (an XML Map Component, for example) as its implementation.

The purpose of the Synchronize Subprocesses activity is to allow data from numerous input activities to be collected into a single activity, in situations where the number of inputs is not known until runtime. In other words, this represents a scenario that (due to an indeterminate number of links) can't be drawn on a process graph. It is sometimes called a “fan-out/fan-in” scenario.



The fan-out activity in this diagram might represent a start activity in a process that receives a batch of work items. The number of work items, however, is not known until runtime. Suppose a subprocess called *DoWork* can process exactly one work item, then pass it on to the next activity. Ideally, you'd want the start activity to be able to fan out N work items to N instances of *DoWork*, have those instances execute in parallel, then collect all the results of the various *DoWork* instances at a central Fan-In Activity, as shown.

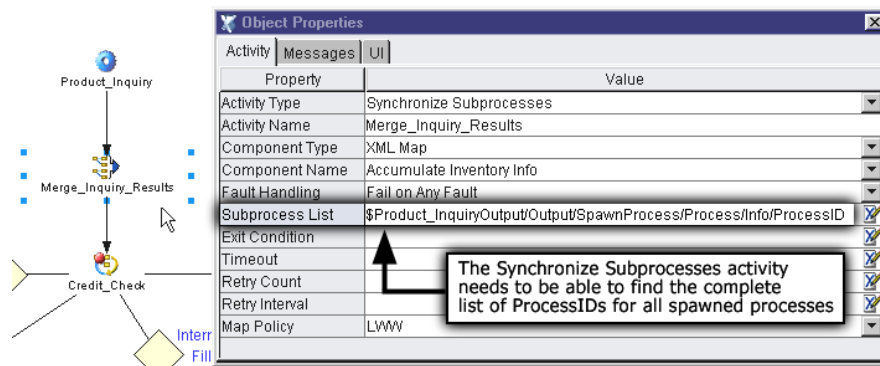
The problem is that this pattern can only be drawn if the maximum number of possible instances of *DoWork* (the maximum batch size) is known in advance. If it were possible to know, for instance, that a batch can never hold more than 12 work items, then you could place 12 activity icons on the graph, representing 12 launchable instances of *DoWork*, and connect links from the Fan-Out activity to each instance of *DoWork* (as well as outgoing links from each *DoWork* to a Fan-In Activity.) A simple XPath condition on each link could determine (by looking at the output from the Fan-Out activity) whether a given link should fire based on whether the appropriate source XPath contains data.

An explicit graph of the type just described will work. It wouldn't be pretty to look at, and the data mappings would be tedious to spell out, but it would work. The problem is that six months from now, someone could decide that the maximum batch size needs to be 200 instead of 12. Or, there may be no limit to the batch size. What then?

The Synchronize Subprocesses activity is designed to handle resynchronization of the results of a fan-out. The process engine performs certain services on behalf of the Synchronize Subprocesses activity, and the activity's implementation must be designed with certain runtime behaviors in mind. The salient points to bear in mind are:

- ◆ The Fan-Out Activity (which can be any of the standard Process Manager activity types) invokes N instances of a *Subprocess activity*. The instances are spawned from **Process Execute** actions inside the Fan-Out's action model, as part of a loop.
- ◆ Because the “work activities” are *subprocesses* and are *spawned* (rather than called synchronously), each subprocess returns a ProcessID to the Fan-Out activity immediately.

- ◆ The Fan-Out activity implementation should collect the ProcessIDs under a known XPath in Output. That XPath must, in turn, be specified in the property sheet for the Synchronize Subprocesses activity as shown here:



- ◆ The component that provides the underlying implementation of the Synchronize Subprocesses activity need not know about the list of ProcessIDs. The runtime engine will call the implementing component the appropriate number of times, based on this list; then it will pass control (when every subprocess has finished) to the next link or links in the chain, barring a fault condition. Thus, *the implementing component does not need to know that it is being used as part of a loop.*
- ◆ Each time the fan-in implementation is fired, the Input message part will contain the output from a subprocess that just finished. It is up to the Synchronize Subprocesses implementation (the fan-in component) to process the newly acquired data as needed. Usually, this will mean accumulating it onto Output, for reasons explained below.
- ◆ When all subprocesses have returned, the activity returns (barring a fault condition) and the parent process continues down the normal control chain.

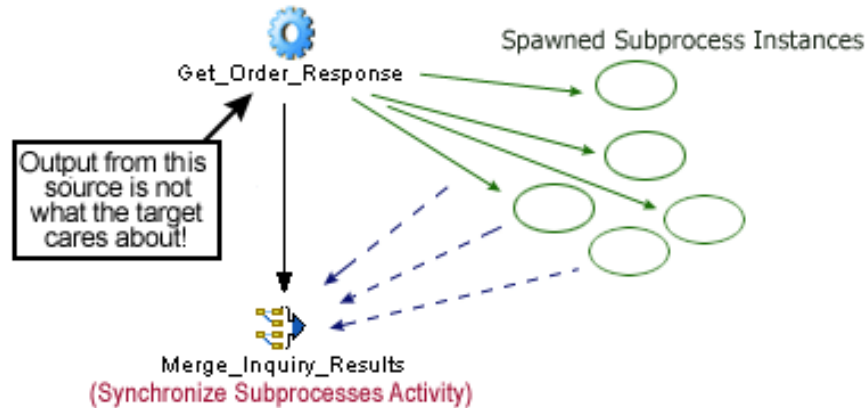
Data Mapping in the Synchronize Subprocesses Activity

The Synchronize Subprocesses activity will always have at least three message parts: Input, Input1, and Output. The activity *implementation* will have DOMs corresponding to these part names as well, but the parts have unique roles and an implementation should be designed with those roles clearly in mind.

Input

From the implementation's point of view, the *Input* message part is where *subprocess output* will be received. Each time a spawned subprocess returns, its output gets passed to the merge component's Input. (The "merge component" here means the Synchronize Subprocesses activity *implementation*: an XML Map component, JDBC component, or whatever.)

In the case of most other activity types, data from the previous activity's Output is passed into the target implementation's Input DOM. In the Synchronize Subprocesses case, however, this is not true, because the activity that fires the Synchronize Subprocesses activity is not really the data source of interest. See below.



The Synchronize Subprocesses activity implementation (or *merge* component) is interested in data provided by the *subprocess instances* that were spawned. It looks to `Input` to find that data. Each time the merge component is fired, it sees a single work-item's worth of data in `Input`.

Input1

The Synchronize Subprocesses activity implementation will typically map the `Input1` DOM straight to the `Output` DOM before doing anything else. That is to say, there will usually be an XML Map action at the top of the implementation's action model that looks like:

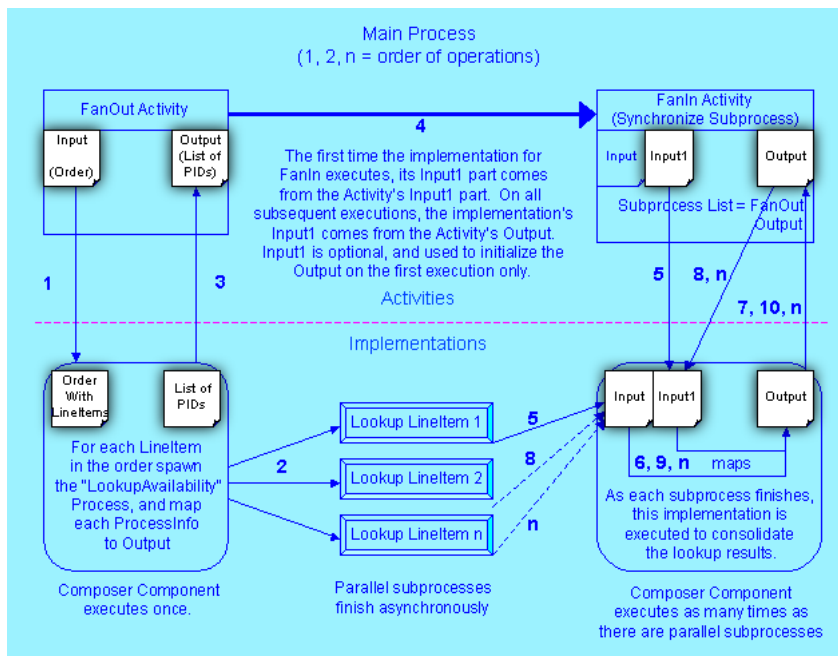


This is because the merge component's `Output` part will be *fed back into* `Input1` on every subsequent invocation of the component. See discussion below.

Output

In order to allow the Synchronize Subprocesses activity implementation to accumulate or consolidate "work items" into a single document, by adding subprocess returns one at a time to an incrementally built DOM, the Synchronize Subprocesses activity recycles its implementation's `Output` back to `Input1`. In other words, on invocation N , the implementation receives, in `Input1`, the `Output` from invocation $N-1$. (On invocation zero, `Input1` is empty.)

See diagram below.



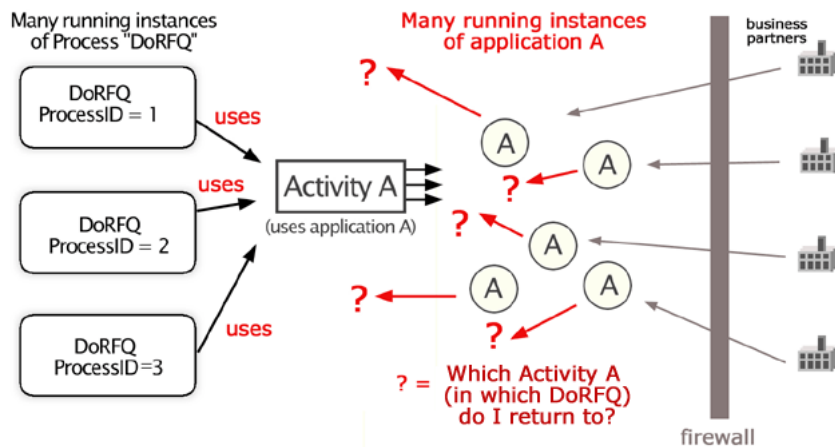
Fault Handling

You can choose to have the Synchronize Subprocesses activity raise a Fault message according to one of two policies: *Fail on Any Fault*, or *Fail if All Fail*. In the first instance, the activity faults out as soon as any one of the feeder activities (the data-producing subprocesses) gives a fault. In the second case, all of the spawned subprocesses must return before a fault is generated. In either case, if the Synchronize Subprocesses activity results in a fault, the process of which it is a part will terminate unless the fault is handled (just as it normally would). Therefore, as a safeguard against a single fanned-out subprocess instance failing your whole process, you should take time to “think through” a robust fault-handling scheme.

Waiting Activities

Any time an activity (such as a subprocess or Web Service Receive activity) is in a *wait state*, waiting to receive a response to some request that was made asynchronously by another activity, it is said to be a *waiting activity*. In the wait state, the activity is not “running” in the normal sense of the word; it is not in memory. The activity implementation might be a Web Service that operates according to the Request-Response or One-Way port types of WSDL. It gets fired when a request comes in via HTTP to the server, or via a message sent to a JMS message listener, etc. After the service is finished, the activity for which it is the implementation (the waiting activity) needs to “wake up” and notify the process engine so that the proper *process instance* can continue to execute the appropriate flow pattern.

But an activity implementation, being merely an application or service of some sort, doesn’t necessarily know (nor *should* it know) that it is being used in a stateful process. The application (the activity implementation) might be a generic, reusable, multi-role application or component that gets invoked by external clients as well as by local applications. It may be a part of several different process models. At any one time, there might be dozens of process instances using the component as an activity implementation. When an instance of the component fires, it has no idea who called it or why; it doesn’t magically know if it is being used as an activity implementation in a running process. See below.



If an activity is waiting for its underlying implementation to produce output, the underlying service or component has to have some way of hooking back into the correct process instance, because numerous process instances (possibly belonging to different process models) might be using the same implementation. A correlation value of some kind must be passed into the waiting activity's implementation so that the implementation can get the activity out of the wait state and let the proper process instance resume navigation.

NOTE: The particulars of how and when to specify a correlation value will be discussed in the next chapter.

The scenario, then, is this:

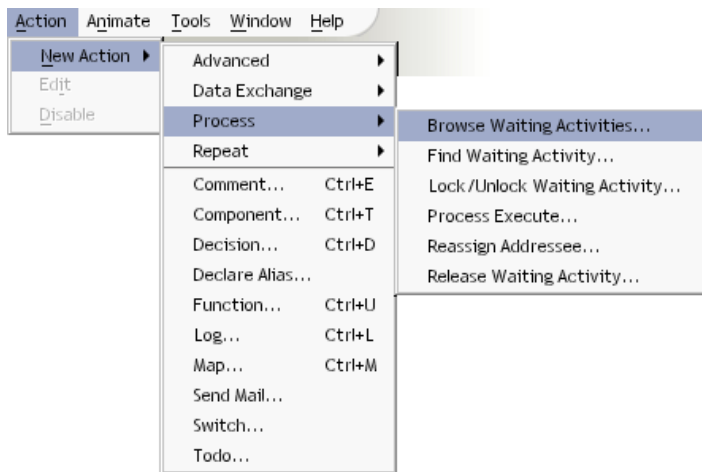
- ◆ The activity that makes the original outbound notification to an external service or business partner must pass a *correlation value* to the service. This can be a custom CorrelationID in conjunction with the Process name, or it can be a ProcessID in conjunction with the Activity name. (See the next chapter for details.)
- ◆ The web service that serves as the implementation for the Web Service Receive Activity must get the correlation value back from the external service (business partner).
- ◆ The web service (WSR implementation) must be a Composer service with an action model that contains a *Find Waiting Activity* action. (**New Action > Process > Find Waiting Activity**.) The correlation value(s) will be used in this action as a means of looking up the appropriate waiting activity in the appropriate process.
- ◆ Once the Find Waiting Activity action has successfully executed, it must be followed by a *Release Waiting Activity* action. (**New Action > Process > Release Waiting Activity**.)

“Waiting Activity” Actions

When the Process Manager has been installed as part of a Composer installation, all component editors for all component types (JDBC, XML Map, JMS, Telnet, etc.) have six Process-related actions available for use in any action model:

- ◆ Browse Waiting Activities
- ◆ Find Waiting Activity
- ◆ Lock/Unlock Waiting Activity
- ◆ Process Execute
- ◆ Reassign Addressee
- ◆ Release Waiting Activity

These actions are available off the Process submenu in the New Action menu. You can use them in the action model for any type of Composer component or service (XML Map component, JDBC component, etc.), but if they're used in a component, the component should be wrapped in a Composer *web service*.



Five of the six activities are related to Waiting Activity functionality. All such functionality assumes the presence of an activity whose implementation follows a *One-Way* or *Request-Response* type of communication pattern. These are patterns in which the web service waits, passively, for an external request.

Find Waiting Activity and Release Waiting Activity actions will be used together in most scenarios that involve waiting activities, regardless of the nature of the associated business tasks. That's because both are needed in order to "wake up" a Web Service Receive activity once it has been enabled.

When process flow reaches a WSR activity at runtime, the process goes to sleep and only wakes up again when:

- ◆ a Composer web service executes a Release Waiting Activity action targeting the WSR activity, *or*
- ◆ the WSR activity times out

In other words, the coupling between a WSR activity and its underlying implementation is quite loose. A Web Service Release activity can be thought of as simply a place in the process flow where the process goes to sleep until it is woken up either by an alarm clock (i.e., the activity times out) or by a web service that knows how to wake the process up again.

For an in-depth discussion of Waiting Activity actions and their usage, see the next chapter.

Waiting Activities and Human Interaction

The actions called *Browse Waiting Activities*, *Lock/Unlock Waiting Activity*, and *Reassign Addressee* add optional functionality designed to make it possible to use waiting activities in a human-intervention type of workflow, where human operators perform tasks in response to notification by activities. It is common in this type of flow for notifications to be sent to human operators, who will ultimately post work back to the process via waiting activities. This type of scenario is discussed in greater depth in the following chapter.

The concept of an *Addressee* is exposed in some of the "waiting activity" action dialogs. This allows work items (that is, message parts, or node branches within parts) to be assigned to specific individuals according to their roles, as part of a running process. The individuals in question can be notified of arriving work via an activity designed for that purpose; and the process instance can call on a Web Service Receive activity (or other "waiting activity") to receive various individuals' work back into the system.

The notion of work-item *Priority* is also exposed in this system.

NOTE: Addressee and Priority are initially specified in the Object Properties panel of the Web Service Receive activity. (The Addressee and Priority properties will not be visible in other activity types. A Web Service Receive activity must be selected in order to see these properties.)

Work items can be marked as *locked* for exclusive use by one individual, programmatically, through the Lock/Unlock Waiting Activity action.

Work items can be *reassigned* to different individuals via the Reassign Addressee action.

In addition, waiting activities representing the work queues of specific individuals can be *browsed* or tallied using the Browse Waiting Activity action.

Through the creative use of these actions, you can develop sophisticated (yet robust and easy to test) workflow systems involving work queues, work items with varying priorities, human operators with roles, and so on.

6

Waiting Activities and Addressees

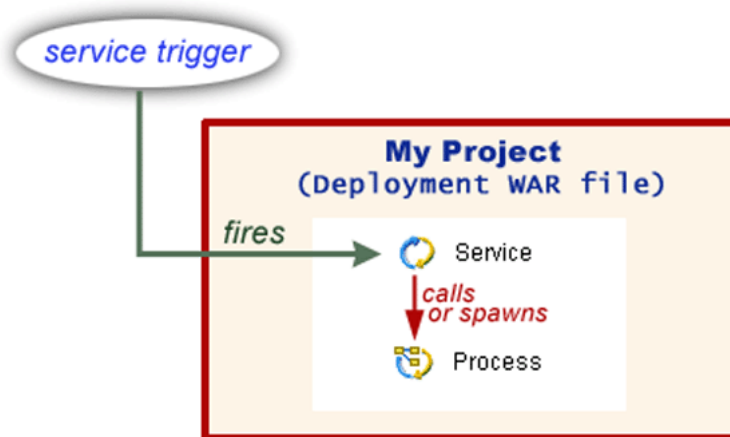
This chapter discusses a wide variety of issues relevant to invocation and control of processes and activities, including the various actions that can be used inside components to implement “human intervention” scenarios involving work lists. To get the most out of this chapter, you should already be familiar with Composer project deployments and standard J2EE packaging and deployment constructs, such as EAR/WAR files, *web.inf* files, contexts, servlets, and so on; and you should be familiar with the basic Composer service trigger types. For more information on the latter subject, be sure to consult your Composer Server User’s Guide for the app server environment (WebSphere, Weblogic, SilverStream) into which you will be deploying.

Understanding How Processes Are Triggered

In order for a process to be invoked, it needs to be associated with a *Process Execute* action (see “The Process Execute Action” further below) in a calling component. The calling component can be any valid Composer component type (XML Map, JDBC, HTML, Telnet, or whatever). At some point, however, the component must itself be called by a service, and the service must be triggered by one of the standard Composer service trigger types. From the top down, then, the activation sequence is:

- ◆ HTTP/SOAP request fires servlet (typically)
- ◆ Servlet (service trigger) fires Composer service (web service)
- ◆ Composer service fires Composer component(s)
- ◆ Composer component fires Process
- ◆ Activities start and stop within the Process instance’s lifetime

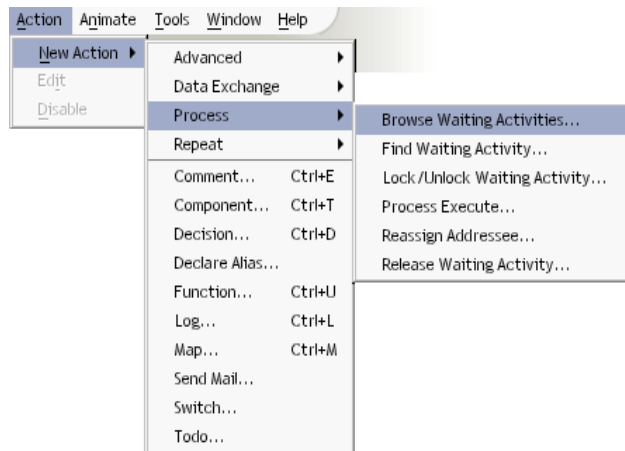
In its simplest form, the activation chain looks something like:



Here, a Composer service is shown calling/spawning a Process directly. But as mentioned earlier, any component type (XML Map, JDBC, etc.) can also call or spawn a Process. The way this is done is via a Process Execute action (discussed in further detail below).

Process-Related Actions

The Process Manager adds six process-related actions to the component editor menus.



The actions can be thought of as supporting three basic types of functionality:

- ◆ **Process invocation via an action:** This is accomplished by the *Process Execute* action. Within a given project, any Composer component or service (whether inside or outside of a Process) can launch any process in that project via this action.
- ◆ **Reentry into a Process:** The *Find Waiting Activity* and *Release Waiting Activity* actions make it possible for a service that implements a Web Service Receive activity to get the attention of the process engine after it has finished running.
- ◆ **Human-accessible work queues:** The *Browse Waiting Activities*, *Lock/Unlock Waiting Activities*, and *Reassign Addressee* actions provide support for scenarios involving delegation of tasks to individuals in an organization.

The Process Execute Action

The Process Execute action allows you to launch a Process using runtime inputs and outputs that you specify. By using this action in a component's action model, you can invoke any Process in the current project.

The Process Execute action is similar to Composer's regular Component action (which fires components), except that it covers two possible methods of execution (namely, Call and Spawn) and optionally allows you to register the called/spawned process as a subprocess of a parent process.

If the process is started via a *Call*, then the action model containing the Process Execute action (the source component) will block any further action processing until the called process returns. This is the same behavior as for the regular Component action.

If the process is started via *Spawn*, then the process is executed in a "fire and forget" mode wherein the spawner does not wait for the spawnee to return. Instead, the spawned process returns (immediately) a "receipt" consisting of a unique identifier for the process and a timestamp. (See below.) This information can be used by other process actions such as the Browse Waiting Activities or Find Waiting Activity actions.

Data Returned by a Spawned Process

When you spawn a process, you are invoking the process asynchronously, in “fire and forget” manner. The spawned process will hand back a message containing certain information about the process instance that was spawned (which can be useful later). The “return receipt” information handed back by a spawned process looks like:

Output	Data
MESSAGE	
Process	
Info	
ProcessID	1
CreationDate	Fri Feb 15 15:42:49 EST 2

The information returned includes the `ProcessID` associated with the particular process instance that was just started, and the date of birth of the process instance, in `Process/Info/CreationDate`.

How to Create a Process Execute Action

➤ To create a Process Execute action:

- 1 Open the Composer component or service from which you wish to invoke a process. Click inside its action model at the point where you want to insert the action.
- 2 In the **Action** menu, choose **New Action > Process > Process Execute**, as shown above. (You can also reach this command from the context menu, available by right-clicking in the action pane.) A dialog appears.

- 3 Using the pulldown menu in the top left corner of the dialog, choose the Execute Method: **Spawn** (fire and forget) or **Call** (block until results come back).
- 4 From the pull down menu under **Process Component**, select the process that you want to invoke. The menu will be prepopulated with the names of all process xObjects in the current project.
- 5 Under **Passed Part**, select (from the dropdown list that appears when you click in this field) the name of the component DOM that will be the data source for the process.
- 6 Under **Returned Part**, select (from the dropdown list that appears) the name of the DOM that will receive information back from the process. For a spawn action, see discussion below.
- 7 If you are spawning this process as part of a fan-out (in anticipation of later using a Synchronize Subprocesses activity to sync back up), check the **Spawn as Subprocess...** checkbox and indicate (via XPath expression) where the parent process’s `ProcessID` can be found.

NOTE: This is an advanced option that is useful primarily when you are working with the Synchronize Subprocesses activity type. Leave this checkbox unchecked unless the current component is an activity implementation for a process and you are using the Synchronize Subprocesses activity type somewhere else in the same process.

- 8 Click **OK** to dismiss the dialog.

More about the Process Execute Dialog

Passed Part represents the runtime name(s) of the source component parts that will be passed into the target process as its ProcessInput message. When you select a process to execute from the drop down list box, the parts it is expecting as defined by its input XML Template will appear. You then simply match up the current component parts to pass with their process counterparts. The Passed Parts do not have to match the template parts in name. However, to insure that the process receives all the data it needs, the number of parts passed should equal the number of parts expected.

The output from a Process Execute action is returned to a Part in the current component that you specify. If the process is executed via a Call, then the process output will be placed in the Returned Part. If the process is executed via Spawn, then a Process Info receipt is placed in the Returned Part.

Spawn as Subprocess of Parent ID

The checkbox called “Spawn as a subprocess of Parent ID,” underneath the Returned Part section of the Process Execute dialog, is visible only when Spawn is selected as the Execute method. The controls just below the checkbox appear, also, when Spawn mode has been chosen. These controls allow you to correlate a spawned process with a given parent process, so that the process engine can keep track of subprocess returns. This is important only in the context of a Synchronize Subprocesses activity implementation.

NOTE: If you are not implementing a “fan-out” type of scenario culminating in a fan-in via a Synchronize Subprocesses activity, you do not need to concern yourself with this discussion.

The parent Process ID that is attached to each process when it is spawned by the Process Execute action allows the engine to return each spawned process’s results to the correct parent process. Since there may be many instances of the parent process running at one time, this mechanism prevents one instance of a parent process from receiving the results of a different instance.

For a more detailed discussion of a fan-in/fan-out scenario using the Synchronize Subprocess activity, see “Synchronize Subprocesses Activity” in the preceding chapter.

Deployment and the Process Execute Action

The unit of deployment in all Composer projects is the Web Service xObject. Thus, as with all other Composer components (e.g. JDBC, EDI, XML Map, 3270, etc.), any Process you wish to expose to a business partner must be executed from inside a Web Service component.

In the simplest case, you can deploy a Process by placing a single Process Execute action inside a Web Service, which you then deploy as you normally would. You merely need to make sure that the Web Service’s input message (and its constituent parts) matches the input(s) for the Process. Then it is a simple matter to call the process via the Process Execute action and pass in the parts.

In a more complex deployment, the Process Execute action may be part of a larger action model that either prepares the initial process message or runs multiple other components and/or processes as well.

Find Waiting Activity Action

You will typically use the *Find Waiting Activity* action inside the implementation for a Web Service Receive activity. (It is usually followed by a *Release Waiting Activity* action. See discussion further below.) The *Find Waiting Activity* action allows you to retrieve runtime information from the process engine for a Web Service Receive activity that is waiting to be fired by (for example) a business partner. The retrieved information, along with the business partner's message, is then used to generate an output message for the activity.

A *Release Waiting Activity* action generally follows every *Find Waiting Activity* action. The *Release Waiting Activity* action causes output to be passed to the Web Service Receive activity and signals its "exit readiness" to the process engine, thus allowing the process flow to continue.

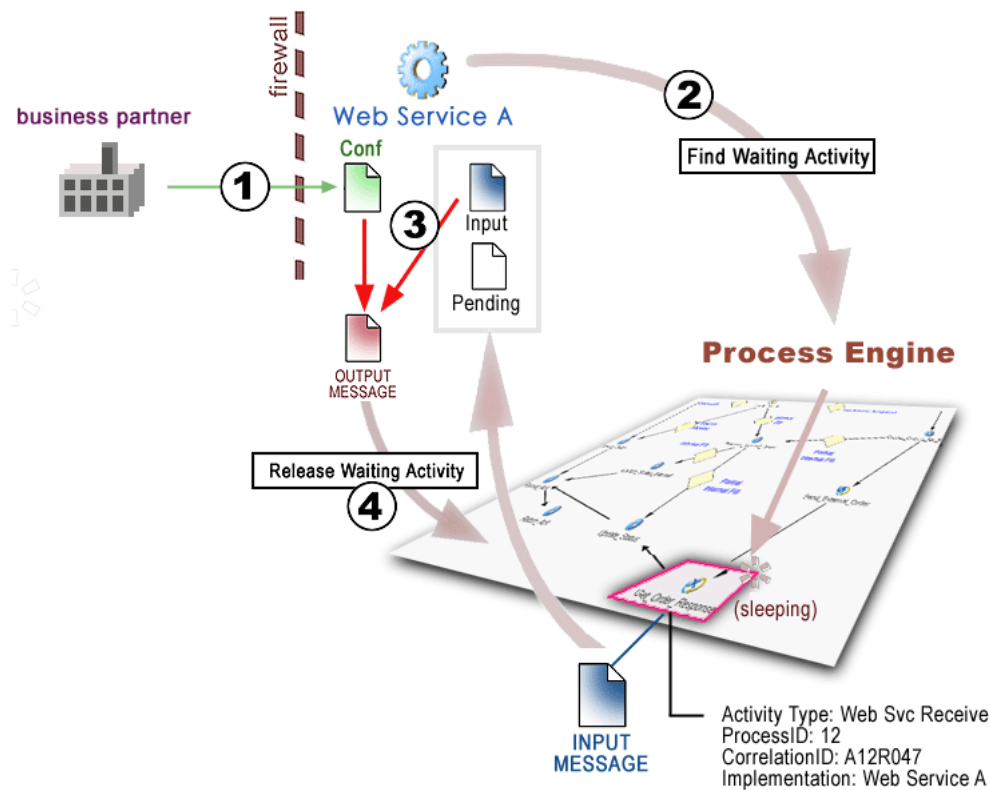
Recall that in the Process Manager, the *Web Service Receive* activity allows a process (or branch of a process) to halt the flow of control at that activity in order to wait for a Web Service to receive information that is necessary to continue the process flow.

The implementation for the Web Service Receive will typically be an exteNd Composer web service with a published WSDL endpoint for the business partner to contact. After being contacted, this Web Service needs a way to find its associated Web Service Receive activity in the correct process instance, pass it the business partner's message, and signal the process engine that the activity is complete (see *Release Waiting Activity* described later). The *Find Waiting Activity* action fills the need of locating the proper Web Service Receive activity.

NOTE: It's important to keep in mind throughout this discussion that activities and activity implementations are not the same thing. *Activities* are abstract entities that have certain attributes and states meaningful only to the process environment. An activity *implementation* is the business application that carries out some task in software. An activity has certain properties associated with it—these are shown in the property sheets which make up the tabs in the Object Properties panel. But in general, an activity doesn't know anything about the implementation, or underlying app, that carries out the actual work required to accomplish a given business task. Conversely, an implementation doesn't know that it is being used in a process.

A Scenario

Consider the following scenario: You have defined a process that places an order, sends a confirmation to a business partner asking for final approval to execute the order, and then waits to hear back from the partner. The partner sends a message referencing the order number back to you, at which point the order process continues. A *Web Service Receive activity* is used for the part of the process that waits to be contacted by the business partner (see No. 1 in the diagram below). This activity's implementation is usually a standard Web Service. That Web Service, in turn, uses a *Correlation ID* to keep everything instance-bound. The business partner will have been given this ID by an upstream activity in the process (the activity that queried the partner). The following diagram shows what happens when the partner finally answers back.



When the business partner sends a confirmation message to Web Service A (which implements the Web Service Receive activity in the above diagram), Web Service A needs to find and “wake up” its associated activity and process. Fortunately, Web Service A contains a Find Waiting Activity action that does precisely this (see 2, above). Using the Find Waiting Activity action, the web service finds the *Web Service Receive activity* that has been waiting for the business partner’s response. That activity’s input message and PendingActivity document can be utilized in creating an output message for the activity (see 3 above). Using the PendingActivity document, the Web Service executes a *Release Waiting Activity* action and passes output back to the Web Service Receive activity (see 4 above), which exits and allows the process to continue.

Finding a Waiting Activity

A waiting activity can be found using one of two methods. One method uses a combination of Process Name and Correlation ID; the other uses Activity Name and ProcessID.

The Correlation ID method is most common for business interactions with business partners on opposite sides of a firewall (i.e. two separate companies). The Correlation ID is simply any unique value—such as a timestamp, work order number, confirmation number, etc.—created earlier in the process and communicated to the business partner. The *Find Waiting Activity* action will extract the CorrelationID from a location in the Input document that you specify, then submit this ID to the process engine to find.

The second lookup method bases its inquiry on a unique key constructed by combining an Activity Name (i.e. the name of the Web Service Receive activity of which the component is a part) and the ProcessID of the process instance in question. The Find Waiting Activity dialog (below) allows you to enter this information.

The second lookup method is more common in cases where requesting and responding parties are both inside a common firewall.

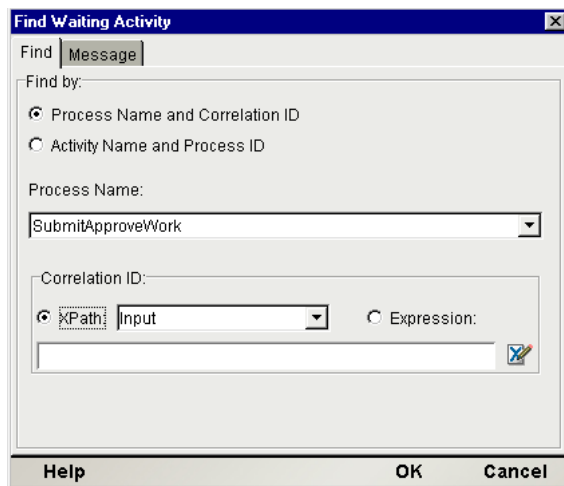
With either lookup method, it is essential that the business partner provide the needed ID information in the input message to the Web Service. That ID information will consist either of a Correlation ID, or a combination of Process ID and Activity Name.

The Find Waiting Activity Dialog

The Find Waiting Activity dialog consists of two tabs of controls. The Find tab is where you specify criteria used to find a waiting activity. The Message tab allows you to specify where to place the information returned from the process engine regarding the activity you find.

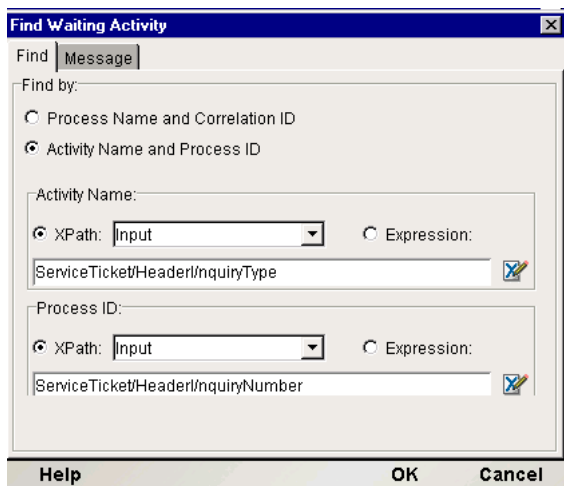
Find tab

The Find tab will take on a different appearance depending on which of the two radio buttons is selected in the top part of the dialog. When “Process Name and Correlation ID” is selected, the dialog takes on the following appearance:



The screenshot shows the 'Find Waiting Activity' dialog box with the 'Message' tab selected. Under 'Find by:', the radio button for 'Process Name and Correlation ID' is selected. The 'Process Name:' dropdown menu is set to 'SubmitApproveWork'. Under 'Correlation ID:', the radio button for 'XPath' is selected, and the dropdown menu is set to 'Input'. There is an empty text field for the XPath expression and a small icon to its right. At the bottom, there are 'Help', 'OK', and 'Cancel' buttons.

If you are finding an Activity by **Process Name and Correlation ID**, select a process name from the dropdown list. Then specify an XPath expression identifying where, in the message received from the business partner, Composer will find the CorrelationID. (Alternatively, click the Expression radio button and specify an ECMAScript expression that will evaluate to the needed ID.) The message part containing the business partner’s message will normally be Input, but others are allowed. Note that the element containing the CorrelationID does *not* need to be named “CorrelationID.” A valid XPath expression might be: PurchaseOrder/Header/POID.



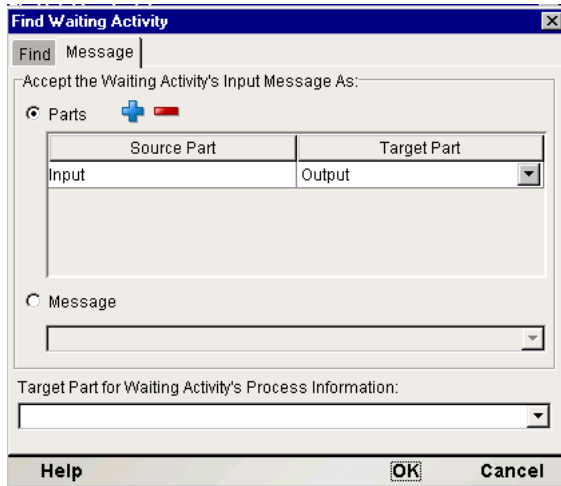
The screenshot shows the 'Find Waiting Activity' dialog box with the 'Message' tab selected. Under 'Find by:', the radio button for 'Activity Name and Process ID' is selected. The 'Activity Name:' dropdown menu is set to 'Input'. The text field for the XPath expression is 'ServiceTicket/Header/InquiryType'. Below that, the 'Process ID:' dropdown menu is set to 'Input'. The text field for the XPath expression is 'ServiceTicket/Header/InquiryNumber'. At the bottom, there are 'Help', 'OK', and 'Cancel' buttons.

When finding an activity by **Activity Name** and **Process ID** (using the radio button labelled “Activity Name and Process ID”), specify an XPath expression identifying where, in the message received from the business partner, Composer can expect to find the Activity Name (i.e. the name of the Web Service Receive activity). The message part containing the business partner’s message will ordinarily be Input, but others are allowed. Note that the element containing the Activity Name does not need to be named “Activity Name.” A valid XPath expression might be: `ServiceTicket/Header/InquiryType`. Also, in similar fashion, specify the location of the ProcessID.

Message tab

When an activity is found, the process engine will return two XML documents to the Web Service that issued the *Find Waiting Activity* action. The first document is the original input message (consisting of 1 or more parts) to the Web Service Receive activity before it began waiting for a contact from a business partner. This allows the activity’s implementation to work on the activity message or use it as a reference with the message received from the business partner.

The second document returned by the process engine is runtime information about the waiting activity (see details below).



The first section of the Message tab allows you to map the activity’s original input message into the Web Service so you can work on it. Two radio buttons control the options available to you:

- ◆ The **Parts** radio button allows you to map each part of the activity’s input message to a part in the Web Service. This will be the choice for most applications.
- ◆ The **Message** radio button allows you to map the entire activity input message (including all its parts) to a single part in the Web Service. If no parts are available for use, you will need to add Temp documents to the Web Service.

The second section of the Messages tab allows you to specify what part in the Web Service will receive the waiting activity’s process information from the process engine. Select a part from the dropdown list. If no part is available for use, you will need to add a Temp document to the Web Service.

The PendingActivity document

The second document returned by the *Find Waiting Activity* action is used by a *Release Waiting Activity* action to signal the completion of the Web Service Receive activity and allow the process flow to continue to execute. The document returned by the process engine that describes a waiting activity contains a root element named *PendingActivity*.

The *PendingActivity* document contains the following child elements:

- ◆ **ProcessID**—This is the unique number associated with the *process instance* in which the found Web Service Receive activity exists. This data is used by the Release Waiting Activity action to restart the waiting activity.
- ◆ **QueueDate**—This is a date/time stamp indicating when the Web Service Receive activity starting waiting for contact from a business partner.
- ◆ **ActivityName**—The name of the Web Service Receive activity that is waiting.
- ◆ **ProcessName**—The name of the process to which the Web Service Receive activity belongs.
- ◆ **CorrelationID**—The unique key used to identify and find the Web Service Receive activity. The value is specified as a property of the Web Service Receive activity and is set by the process when the activity executes and begins waiting.
- ◆ **Addressee**—The name of a user who is supposed to contact this Web Service Receive activity. This data is usually used in assigning work to people in work queue applications that involve user intervention/interaction with a long-running process. The value is specified as a property of the Web Service Receive activity and is set by the process when the activity executes and begins waiting. This data is usually used in processes that run completely behind the firewall.
- ◆ **Priority**—This data is used in assigning work to people in work queues and allows an application querying the process engine to sort waiting activities by relative importance. The value is specified as a property of the Web Service Receive activity and is set by the process when the activity executes and begins waiting. This data is usually used in processes that run completely behind the firewall.
- ◆ **LockedBy**—This label is typically the name of a user in a work queue who has flagged this activity instance as being locked for exclusive use by one individual. No actual lock is created; rather, it is a semaphore or flag value for work-queue applications querying the process. The value is set by a *Lock/Unlock Waiting Activity* action.
- ◆ **LockedUntil**—A date value indicating when the lock will be removed. The value is set by a *Lock/Unlock Waiting Activity* action.

Release Waiting Activity Action

The *Release Waiting Activity* action is used inside the implementation (usually a Web Service) for a Web Service Receive activity. The action is usually preceded, at some point in the implementation's action model, by a *Find Waiting Activity* action. The action passes data to the waiting Web Service Receive activity; and that data becomes the output message, signalling the activity's completion to the process engine. The passed data is usually information from a business partner that contacted the Web Service. Thus, the *Release Waiting Activity* action is the callback mechanism used by a Web Service to produce an output message for, and signal the completion of, a Web Service Receive activity inside a process.

Before an Activity can be released, you must produce a message part that contains a *PendingActivity* document indicating the Process and Web Service Receive activity you wish to release. This can be accomplished via a *Find Waiting Activity* action (see the previous section). In addition, you will need one or more parts that will be passed back to the waiting Web Service Receive activity as its output message. Once you have a *PendingActivity* document and one or more parts to serve as activity output, you can release an activity.

The Release Waiting Activity Dialog

The Release Waiting Activity dialog has three sections to it.

The first section allows you to specify a part in the Web Service that contains a PendingActivity document describing the process and activity you wish to release. This part must have been populated previously by a *Find Waiting Activity* action.

The second section is where you map parts in the Web Service to output message parts in the waiting Web Service Receive activity.

The third section is optional and allows you to return output to the activity but flag it as a fault message causing the process flow out of the Web Service Receive activity to follow fault links.

Source Part	Target Part
Input	Output

Under “Part containing Waiting Activity’s Process Information,” just select a part name from the drop down list.

NOTE: In order for a part to show in the list, it must contain a *PendingActivity* document.

The second section of the dialog allows you to specify data from the Web Service that will become the waiting activity’s output message. The **Parts** option allows you to map one or more parts in the Web Service to one or more parts in the Web Service Receive activity’s output message. This will be the choice for most applications. The Target Part name you enter will be created in the activity’s output message for you. The **Message** option allows you to map a single part in the Web Service as the entire activity output message (including all its parts).

The third section in the dialog allows you to specify (optionally) that the returned data is to be flagged as a fault inside the process. Use the expression builder to specify a fault message name corresponding to one that was defined for the Web Service Receive activity on its Messages tab.

Human Participation in Processes

Most processes require *some* kind of human interaction, if for no other purpose than the initial triggering of the process. In some scenarios, human involvement in *all* phases of a particular business task can be essential. Purchase orders may require personal approval; product inquiries may require personal e-mail responses or phone calls; large transactions may require escalation to a particular individual; and so on.

You can use Composer’s Process Manager to implement a wide variety of sophisticated human-centric workflows. The Process Manager has features that make it easy to:

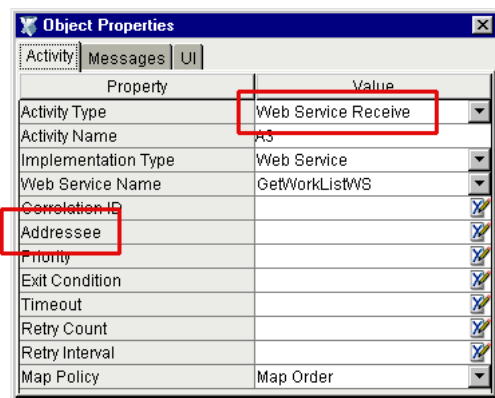
- ◆ Assign (and reassign or reroute) work to individuals
- ◆ Assign priorities to work items
- ◆ Mark work items as locked or unlocked for exclusive use by an individual
- ◆ Browse a process for worklists, filtered by individual(s)
- ◆ Retrieve individual work items
- ◆ Integrate back-end systems into the workflow
- ◆ Integrate easy front-end access to work lists via JSP or HTML

To create human entry-points into a process, you will generally use Web Service Receive activities to expose outward-facing web-service applications (i.e., the implementations of the Web Service Receive activities). The user-facing services could be exposed via JSP or HTML pages; or they might be exposed by other means.

Your user-facing services can be designed to allow users to browse work queues, find and lock work items, unlock work items, reassign work, and/or push work back into the system. The actions that make these operations possible include the Find Waiting Activity and Release Waiting Activity actions already discussed above, as well as the Browse Waiting Activities, Lock/Unlock Waiting Activity, and Reassign Addressee activities discussed below.

Addressees

Human participants in a process are known, in Process Manager, as *Addressees*. The runtime engine associates an Addressee with a particular document or work item via the Addressee property on the Object Properties panel for the Web Service Receive activity type.



NOTE: Addressee is a property of Web Service Receive activities only. You will not see this field in the Object Properties panel for other activity types.

The Addressee value is specified as an *XPath expression*. This affords a great deal of flexibility, since the Addressee can be in a passed-in message part, or it can be determined dynamically at runtime, or it can be hard-coded to a particular string value. Thus, you can accommodate any of the following common scenarios:

- ◆ The sales person to whom the order should go is determined by a JSP scriptlet or EJB at the time the order is submitted online. The ProcessInput message already contains the necessary Addressee name at process invocation.
- ◆ An order arrives via the Web and kicks off a process. The Addressee is determined dynamically—“just in time”—by business logic in a preprocessing component, and it appears in the output message of an activity.
- ◆ All orders must eventually be approved by John Smith. Therefore, a Web Service Receive activity is hard-wired to an Addressee value of “John Smith.”

The Role of the Web Service Receive Activity

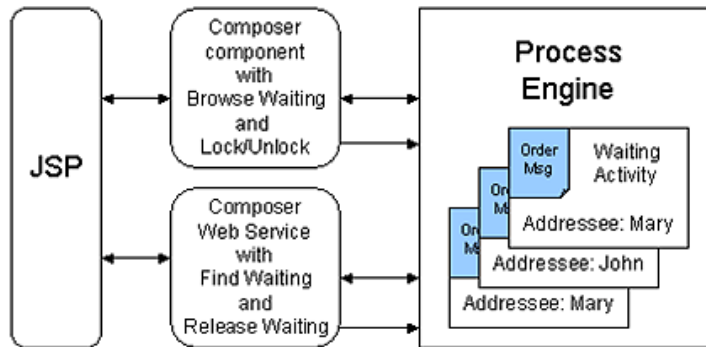
The Web Service Receive activity is the main touchpoint for human input into automated processes built with the Process Manager.

When a Web Service Receive activity “fires,” three things happen:

- ◆ Its Addressee property becomes associated with a string value (usually, although not necessarily, representing a real person’s name)
- ◆ The underlying implementation for that activity (namely, a user-facing Web Service that follows the WSDL One-Way or Request-Response pattern) becomes operational
- ◆ The Process Engine puts the Web Service Receive *activity* (not its implementation!) into a waiting state

A person (i.e., a worker or an administrator; the Addressee) can then use a work-group application implemented in JSPs to execute Composer components or services to:

- 1 Browse all the waiting activities addressed to a particular person (using a Browse Waiting Activities action),
- 2 Lock waiting activities to prevent other users from approving an order (using a Lock/Unlock Waiting Activity action) while they are being reviewed by an Administrator,
- 3 Unlock an order, allowing it to be approved (using a Lock/Unlock Waiting Activity action),
- 4 Retrieve orders and mark them as approved (using a Find Waiting Activity action), and/or
- 5 Complete the approval and allow the process to proceed to the next activity (by finally calling a Release Waiting Activity action).



Notice that only the Release Waiting Activity action can actually *complete* a Web Service Receive activity (that is, cause it to exit). So by definition, the implementation to a Web Service Receive activity (i.e. a Web Service) *must* contain a Release Waiting Activity action. The other process actions (Browse Waiting Activities, Lock/Unlock Waiting Activity, Reassign Addressee, and even Find Waiting Activity) can be used in a variety of components not directly connected to a process. Such components might use these actions to add oversight and “see-into” functionality to external applications, giving users a means to view and manage waiting activities.

Browse Waiting Activities Action

The *Browse Waiting Activities* action can be used in any service or component (even one outside the process), as long as the process and the service/component are deployed as part of the same Composer project. The sole purpose of the Browse Waiting Activities action is to allow an application to obtain a list (or lists) of pending activities, filtered by Addressee.

When you execute a Browse Waiting Activities action, you are merely supplying the Process Server with a name, or a list of names. The Process Server, in turn, examines all Web Service Receive activities in all instances of all processes, and hands back a list of activities waiting to be acted on by the individuals (or departments, etc.) in question.

The list that the Process Server returns in response to a browse is a *PendingActivity* document. This document contains such information as the ProcessID and Activity Name for the pending activity, as well as other information that can be used, if desired, to find and do work on behalf of a waiting activity. (See “The PendingActivity document” further above for additional discussion of the PendingActivity document structure.)

Where to Use the Browse Waiting Activities Action

The Browse Waiting Activities action is usually utilized in scenarios where a long-running process requires the intervention of, or interaction with, a live person through another application such as a JSP. (See the eXtend Composer Silverstream Server Guide section titled “Creating a JSP that calls a Composer Service” for more detail.) For example, you may have a heavily trafficked process that processes orders one-at-a-time. The process contains an activity that accepts a single order and passes it to a Web Service Receive activity, where the process stops and waits for the order to be approved by a particular person. That person, in turn, will use a JSP (Java Server Page) to input his approval. In this kind of scenario, the Addressee (the person who approves the order) needs to be able to find out about (or discover) work waiting to be done, and also push work into the system as it gets done. The discovery part can be accomplished via a Browse Waiting Activities action in a service (not necessarily internal to the process) that can be triggered off a JSP. The “data push” part can be done via a service that implements the Web Service Receive activity type. This service would use a Find Waiting Activities action to look up individual work items, and a Release Waiting Activity action to execute the “push.”

How It Works

A waiting activity (no matter which process it is in) can be located by Addressee alone, using the Browse Waiting Activities action. But for this to work, the associated Web Service Receive activity *must* have a non-empty Addressee property. To specify a value for Addressee, just open the Process graph in Process Designer, click on the Web Service Receive activity in question, bring the Object Properties panel into view, and enter a legal XPath value next to Addressee. (See screen shot under “Addressees” above.) The XPath should point either to an input message part that contains an Addressee string, or a hard-coded string value.

A second Web Service Receive activity property named *Priority* can also be set in the Object Properties panel. Priority is an arbitrary number that allows the application to sort or filter retrieved work items before displaying them to the user. You can assign any value(s) you want here, or leave the value empty.

In most applications, a Browse Waiting Activities action will be followed by other process actions like Lock/Unlock Waiting Activities, Reassign Addressee, and/or Find (or Release) Waiting Activity. For instance, one possible scenario might be as follows. An administrator for a work group selects multiple waiting activities for a group of users (using the Browse Waiting Activities action). The administrator places a lock on all the selected activities to prevent users from working on the work items (using a Lock/Unlock Waiting Activities action) while they are under review. The administrator reassigns some work items among the users (using the Reassign Addressee action), finds and works on the high priority work items (using the Find Waiting Activities action) and completes them (using a Release Waiting Activity action), and then unlocks the activities not worked on (again using a Lock/Unlock Waiting Activities action).

Comparing Browse and Find

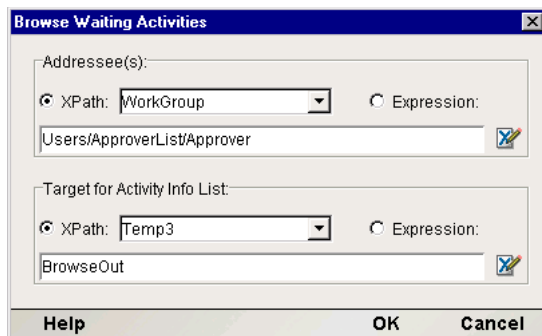
The Browse Waiting Activities action differs from the Find Waiting Activities action in the following characteristics:

- ◆ Browse can search for waiting activities by Addressee only, whereas Find can search only by Process Name/CorrelationID or Activity Name/ProcessID.
- ◆ Browse can return information on multiple activities, whereas Find returns information on just a *single* waiting activity.
- ◆ Browse maps its results to a message part or XPath location, whereas Find maps its results to a part only.
- ◆ Browse does not return the input messages to the found activities, whereas Find does return the waiting activity's input message. So by using a Browse coupled with a Find, an administrator can look into the details of a waiting activity such as looking at the actual order.

NOTE: Both Browse and Find are nondestructive. No waiting activity is marked as finished until a Release Waiting Activity action has been called on it.

Creating a Browse Waiting Activities Action

To create a Browse Waiting Activities action, go into a component and right-click in the action model; then select **New Action > Process > Browse Waiting Activities** from the context menu. A dialog will appear.



The Browse Waiting Activities dialog contains two basic control groups. The first control group offers a way to point to a list of Addressees in a message part in the component. The value(s) contained in the nodelist will be used as the search key(s) for finding waiting activities. For example, in the dialog above, the Addressee XPath points to the list of Approvers shown below.

WorkGroup	Data
Users	
ApproverList	
Approver	Mary
Approver	Susan
ReviewerList	
Reviewer	John
Reviewer	John

The second control group allows you to specify *where to place the results* of the browse. Specify a part name (such as Temp) and an XPath location within the part. The results of the browse will be placed as child elements of the XPath you specify.

Temp3	Data
BrowseOut	
PendingActivity	
ProcessID	55813
QueueDate	Mon Mar 11 12:46:51 EST 2002
ActivityName	ApproveOrder
ProcessName	AcceptApproveOrders
CorrelationID	1029384756
Addressee	Mary
Priority	1
LockedBy	
LockedUntil	
PendingActivity	
ProcessID	55711
QueueDate	Mon Mar 11 11:24:55 EST 2002
ActivityName	ApproveOrder
ProcessName	AcceptApproveOrders

A successful browse will return one or more *PendingActivity* documents (as shown above), each containing child elements describing the waiting activity. If the browse finds no waiting activities, then only the XPath you specify will be created and there will be no *PendingActivity* children elements beneath it.

NOTE: Unlike the Find Waiting Activity action, the Browse Waiting Activities action does *not* return the *input message* for the found activity. In order to retrieve the input message, you must loop through each *PendingActivity* element using a Repeat for Element action, and perform a Find Waiting Activity action on the activities of interest. (The *PendingActivity* branches contain all the information required by a Find Waiting Activity action to retrieve any given activity.)

Lock/Unlock Waiting Activity

The Lock/Unlock Waiting Activity action flags a waiting Web Service Receive activity as being in use or clears that flag, indicating the waiting activity is available to be worked on. The flag consists of the two elements `/LockedBy` and `/LockedUntil` in the *PendingActivity* document associated with a waiting Web Service Receive activity. A non-null value in the `LockedBy` element indicates the waiting activity is unavailable for use. When a lock is flagged, the `LockedUntil` element contains a date/time stamp indicating when the lock or flag will be cleared automatically by the process engine.

It is important to note that the Lock/Unlock Waiting Activity action does not physically lock the waiting activity. It simply marks the activity as being in use. Even when flagged as being in use, a Find Waiting Activity/Release Waiting Activity set of actions can work on and complete the activity. It is up to the designer of the work group application to honor the lock.

NOTE: Locked activities are not excluded from Browse results. The *PendingActivity* information returned by a Browse will show all applicable activities, including both locked and unlocked ones.

The Lock/Unlock Waiting Activity action is typically utilized in applications where a long-running process requires the intervention of, or interaction with, a live person through another application such as a JSP. (See the Composer Silverstream Server Guide section titled "Creating a JSP that calls a Composer Service" for more detail.) These work-group applications commonly use *work queues* filled with *work items* assigned to addressees.

Prerequisites for Locking/Unlocking an Activity

Before you can lock or unlock a waiting activity, you must have a *ProcessID* and *Activity Name* with which to locate the waiting activity. A Lock/Unlock Waiting Activity action will generally be preceded in an action model by a successful Browse Waiting Activities or Find Waiting Activity action. The result of either action is a *PendingActivity* document from which you can reference the necessary *ProcessID* and *Activity Name*.

If the activity you are trying to lock or unlock is no longer present in the process engine, Composer will throw an exception, so it is good practice to anticipate this (for example by placing Lock/Unlock Waiting Activity actions inside a Try/On Error action).

If the Lock/Unlock Waiting Activity action is successful (i.e., no exception is thrown), nothing is returned and the next action in the action model executes.

Creating a Lock/Unlock Waiting Activity Action

To create a Lock/Unlock Waiting Activities action, go into a component and right-click in the action model; then select **New Action > Process > Lock/Unlock Waiting Activities** from the context menu. A dialog will appear.

The dialog box is titled "Lock/Unlock Waiting Activity". It features a close button in the top right corner. The main content is organized into five sections, each with a radio button to select between "XPath" and "Expression" methods. The first section, "Lock Waiting Activity", is selected. The "Process ID" section has "XPath" selected with a dropdown menu showing "Temp1" and a text field containing "PendingActivity/ProcessID". The "Activity Name" section has "Expression" selected with a text field containing ""RecCreateChangeWI"". The "Locked By" section has "XPath" selected with a dropdown menu showing "Input" and a text field containing "WORKITEMREQUEST/ADDRESSEE". The "Lock Duration" section has "Expression" selected with a dropdown menu showing "Input" and a text field containing "120". At the bottom of the dialog are three buttons: "Help", "OK", and "Cancel".

The Lock/Unlock Waiting Activity dialog has 5 sets of controls:

- ◆ The first control group contains two radio buttons that determine whether the action is to set a lock or clear an existing one.
- ◆ The second control specifies the **ProcessID** you wish to target. Specify an XPath within a *PendingActivity* document down to the ProcessID element.
- ◆ The third control specifies the **Activity Name** you wish to target. Specify an XPath within a *PendingActivity* document down to the Activity Name element.
- ◆ The fourth control is the **LockedBy** flag. Specify a meaningful value for the people or processes who might inspect it by executing a Browse or Find against the waiting activity while you have it flagged as locked.
- ◆ The fifth control is the **Lock Duration**. Specify a time interval that will be used to calculate a date/time stamp to place in the LockedUntil element of the *PendingActivity* document associated with the waiting activity. The time interval default unit of measurement is *seconds*, so entering the text 60 will leave the lock flag in place for 60 seconds, after which the flag will be cleared automatically. Other units of measure include minutes (specified inside single quotes as: '60m'), hours (specified inside single quotes as: '60h'), and days (specified inside single quotes as: '60d').

NOTE: If you select the Lock Waiting Activity radio button, values for all controls are required. If you select the Unlock Waiting Activity radio button, values for the Process ID and Activity Name controls only are required.

The Lock dialog settings shown in the screen shot further above might give the following result when another user (say, Mary) browses waiting activities:

Temp3	Data
<ul style="list-style-type: none"> <ul style="list-style-type: none"> <ul style="list-style-type: none"> ProcessID QueueDate ActivityName ProcessName CorrelationID Addressee Priority LockedBy LockedUntil 	<ul style="list-style-type: none"> 55813 Mon Mar 11 12:47:22 EST 2002 ApproveOrder AcceptApproveOrders 1029384756 Mary 1 Administrator Mon Mar 11 01:10:22 EST 2002
<ul style="list-style-type: none"> <ul style="list-style-type: none"> <ul style="list-style-type: none"> ProcessID QueueDate ActivityName 	<ul style="list-style-type: none"> 55711 Mon Mar 11 12:49:12 EST 2002 ApproveOrder

The Reassign Addressee Action

The Reassign Addressee action allows you to change the value of the Addressee attribute assigned to a waiting Web Service Receive activity. In most cases, the original value of the Addressee will be set by the Process Manager when a Web Service Receive activity enters its waiting state. Once in its wait state, the current Addressee can be changed to another value by the Reassign Addressee action. You also have the option of reassigning the the current Addressee for *one* Web Service Receive activity, or *all* of that person's Web Service Receive activities. (For example, you might want to reassign all of Mary's work to Joe while Mary is out sick.)

Remember that the Addressee is an optional attribute that can be assigned to a Web Service Receive activity. The presence or absence of a value does not inherently affect the processing of the Web Service Receive activity except as a flag or tag to an external work group application.

The Reassign Addressee action is typically used in applications where a long-running Composer process requires the intervention of, or interaction with, a live person through another application such as a JSP-driven application or form. Work group applications allowing human interaction commonly use *work queues* filled with *work items* assigned to addressees.

Reassigning an Addressee

Before you reassign the addressee to a waiting activity, you must decide if the action will reassign *all* or just *one* of the activities associated with a particular Addressee. If you want the action to reassign *all* activities, then you need only define two parameters for the action: an XPath or ECMAScript expression identifying the current Addressee, and an XPath or ECMAScript expression identifying the new Addressee.

If you want to reassign a single specific activity of the current Addressee, then you will also need to supply a ProcessID and Activity Name. To do this, the Reassign Addressee action must be preceded in an action model by a successful Browse Waiting Activities or Find Waiting Activity action. The result of the Browse or Find will be a PendingActivity document from which you can reference the necessary ProcessID and Activity Name.

After the Reassign Addressee action executes, whether successful or unsuccessful, nothing is returned. To verify the success of the action, perform another Browse.

Creating a Reassign Addressee Action

To create a Reassign Addressee action, go into a component and right-click in the action model; then select **New Action > Process > Reassign Addressee** from the context menu. A dialog will appear.

The screenshot shows a dialog box titled "Reassign Addressee". It has a standard Windows-style title bar with a close button. The dialog is organized into several sections:

- Original Addressee:** Contains radio buttons for "XPath:" (selected) and "Expression:". Below is a text input field with a search icon.
- New Addressee:** Contains radio buttons for "XPath:" (selected) and "Expression:". Below is a text input field with a search icon.
- Specified Activity:** Contains radio buttons for "All Activities" and "Specified Activity" (selected). Below is an "Activity Name:" label, radio buttons for "XPath:" (selected) and "Expression:", and a text input field with a search icon.
- Process ID:** Contains radio buttons for "XPath:" (selected) and "Expression:". Below is a text input field with a search icon.

At the bottom of the dialog are three buttons: "Help", "OK", and "Cancel".

The Reassign Addressee dialog has several groups of controls. The first control identifies the current Addressee (the one whose work will be reassigned) while the second control identifies the new Addressee. For each one, enter an XPath location from a part in the current action model or an ECMAScript expression that will resolve to the correct Addressee name. (The value for each will typically be passed into the component in which the Reassign Addressee action is used.)

The **All Activities** and **Specified Activity** radio buttons in the middle of the dialog determine whether *all* activities for the current Addressee will be reassigned (as in the case where all of Mary's work needs to be reassigned to Joe), or just one specific activity. If Specified Activity is chosen, the control groups called **Activity Name** and **Process ID** become enabled and you must enter an XPath or ECMAScript expression identifying the specific **Activity** to reassign, along with the specific **ProcessID** containing that activity. In order to supply these values, you will generally have performed a Browse Waiting Activities or Find Waiting Activity action.

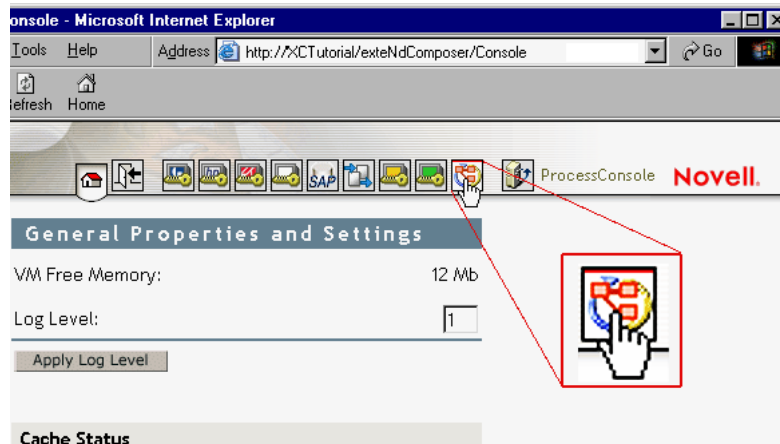
7

Runtime Administration of Processes

This chapter discusses the use of the Process Server Console to manage deployed processes.

Server Console Usage

From the Composer Enterprise Server main console page (shown below), click the “Process Console” icon in the top row of buttons:



NOTE: Composer Enterprise Server and the Process Server should be installed and running on your application server prior to attempting to access these consoles.

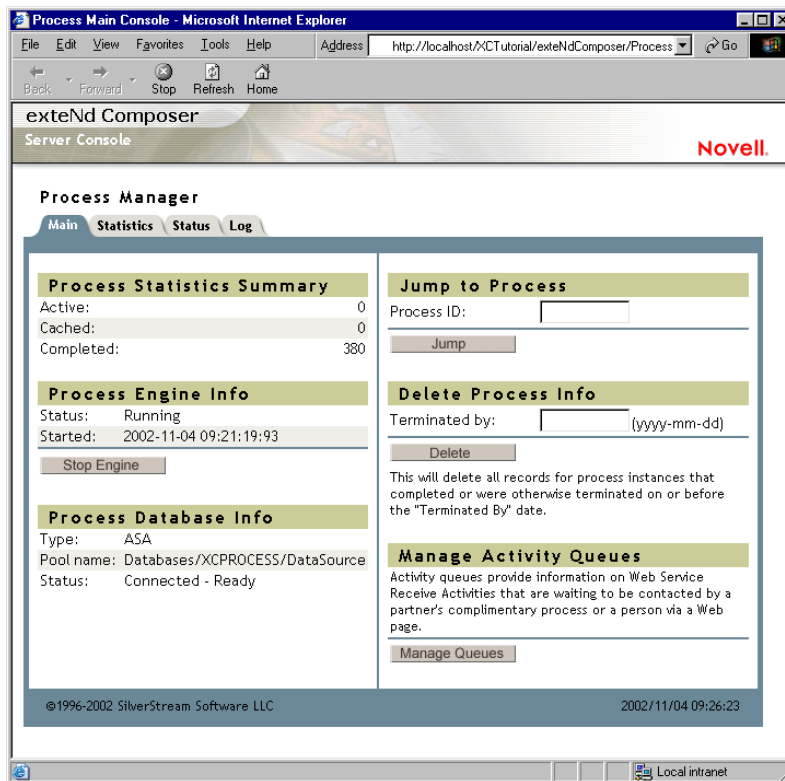
After clicking the Process Console button, a new screen should appear in a new browser window, as shown in the section below. Notice the presence of four tabs (**Main**, **Statistics**, **Status**, and **Log**). These tabs are discussed in the sections to follow.

Process Manager Console: Main Tab

The console’s Main tab reveals a screen comprised of the following sections:

- ◆ Process Statistics Summary
- ◆ Process Engine Info
- ◆ Process Database Info
- ◆ Jump to Process
- ◆ Delete Process Info
- ◆ Manage Activity Queue

Each of these sections is described below:



Process Statistics Summary

The Process Statistics Summary section displays the count of Active, Cached, and Completed processes. The latter refers to the number of process instances that have run since the Process Server was started (i.e., the Start date given in the next section), whether they ended in success or a fault of some kind.

Process Engine Info

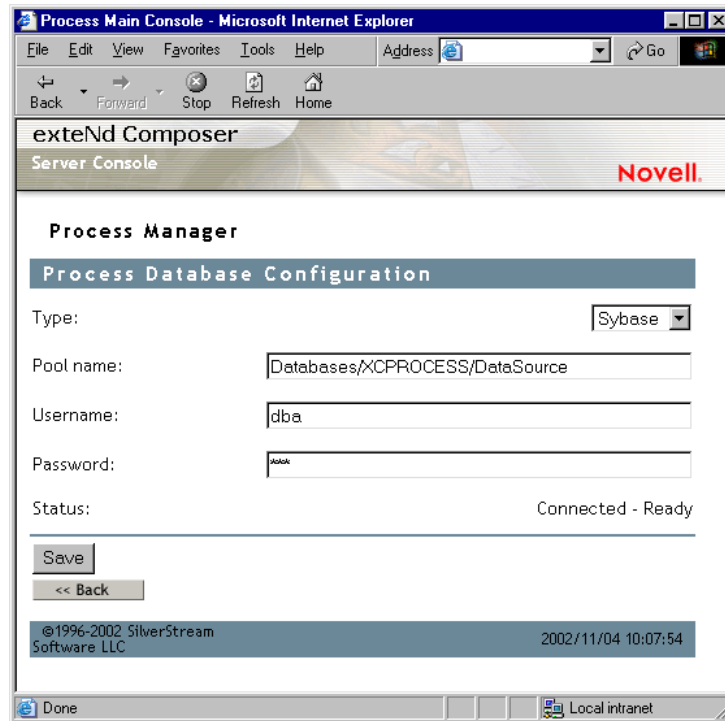
The Process Engine Info section shows whether the process engine is running and, if it is running, the date and time at which it was started. If the process engine is running, the Process Engine Status is “Running” and the button below is labeled “Stop Engine.” If the process engine is *not* running, the Process Engine Status is “Suspended” and the button below is labeled “Start Engine.”

Process Database Info

The Process Database Info section displays general information regarding the *process database*. (See the first few pages of this guide, as well as the product Release Notes, for information on setting up this database.) This is the database Process Manager uses to persist “state data” for long-running processes.

- ◆ **Type**—the type of database (e.g. Oracle, DB2, ASA, etc.)
- ◆ **Pool Name**—the name of the connection pool
- ◆ **Status**—the status may be:
 - ◆ Not Connected to Database
 - ◆ Can’t Connect to Database
 - ◆ Connected—Not Initialized
 - ◆ Connected—Ready

- ◆ **Configure**—the Configure button will be displayed only when the Process Engine is stopped. Pressing the **Configure** button in the Process Database Info section will display the Process Database Configuration page, from which you can configure the database (see below).



- ◆ To configure the database, select a database type from the dropdown list (e.g. Oracle, DB2, ASA, etc.) and enter a pool name. You may save the configuration by pressing the **Save** button. Once the Configuration is saved, you can initialize the database by pressing the **Initialize Database** button. The Initialize Database button is displayed only when the status is *Connected—Not Initialized*.

Jump to Process

The Jump to Process section of the main console enables you to display the status of a specific process by entering the ID of the process and pressing the Go button.

Delete Process Info

Process records can be completely deleted via the Delete Process Info section. You may delete all information for process instances that were terminated (e.g. completed or otherwise terminated) by a specified date. To do this, enter the “Terminated By” date and press the **Delete** button. For example, if you enter 2002-02-01 and press the **Delete** button, all records for process instances completed or otherwise terminated on or February 1, 2002 will be permanently deleted.

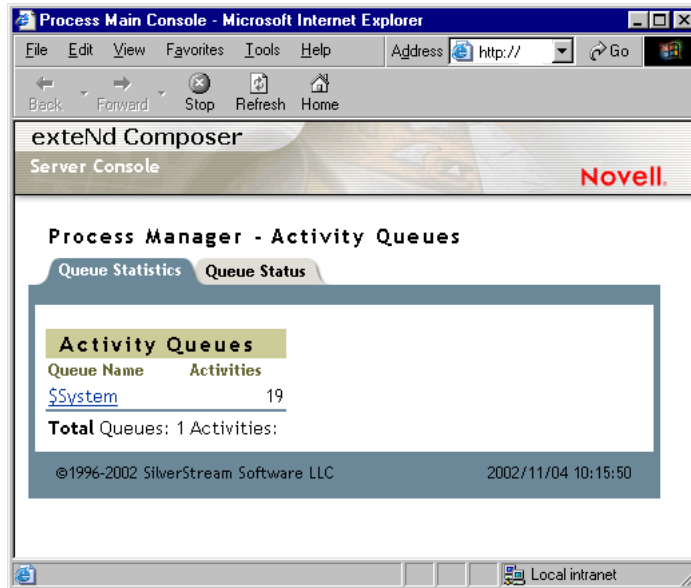
NOTE: When a process finishes running or is manually terminated, only the process instance's input documents and output documents will be maintained. Any interim documents created by the process instance will be purged, automatically, upon termination of the process instance.

Manage Activity Queues

You may administer activity queues by pressing **Manage Activity Queues** button on the Main tab. Doing so will display a page with two tabs that provide queue statistics and queue status.

Queue Statistics

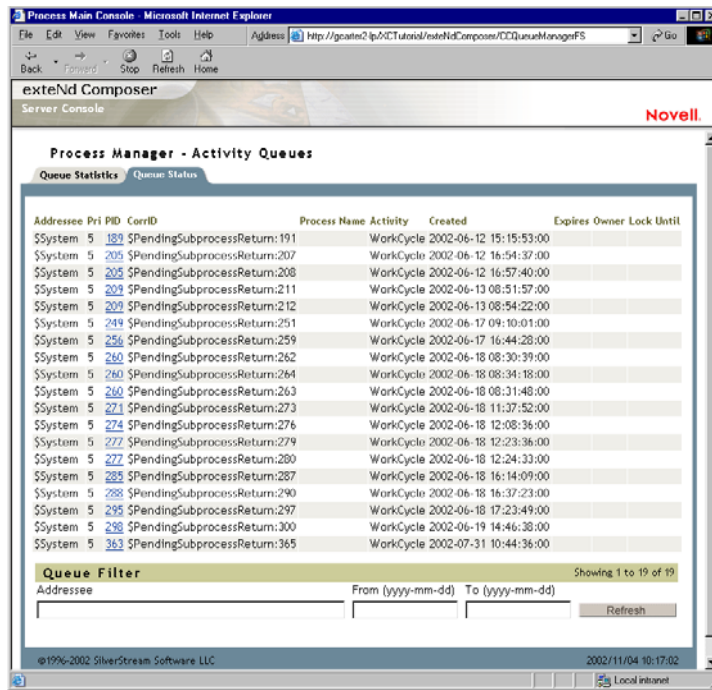
The Queue Statistics tab displays a table that contains a sorted list of addresses in the activity queue and a count of the work items assigned to that addressee. These statistics are automatically refreshed every 60 seconds.



Queue Status

The Queue Status tab (see illustration below) displays a table with the following columns:

- ◆ **Addressee**—the Addressee name
- ◆ **Priority**—the priority
- ◆ **PID**—the Process ID
- ◆ **Corr ID**—the correlation ID
- ◆ **Process Name**—the name of the process
- ◆ **Activity**—the name of the Activity
- ◆ **Created**—the creation date of the Activity instance
- ◆ **Expires**—the expiration date of the Activity instance
- ◆ **Owner**—Owner of the lock
- ◆ **Lock Until**—the date which the lock expires



The PID column contains hot links to the *Process Detail* info for the process instances. (This will open in a new browser window.) The Process Detail windows are discussed in a later section.

If the activity does not have a timeout, the Expires column is blank.

If the Activity has been locked via the Lock Waiting Activity Action, the Owner column displays the name of the Owner of the Locked Activity and the Lock Until column displays the date to which the activity has been locked. If the Activity does not have a Lock, the Owner and Lock Until fields are blank.

Optionally, you may filter Queue Status by Addressee by entering an Addressee name. You may also choose to display only those activities that were queued within a specific range of dates by entering the range of dates in the From and To fields.

Navigation

The Queue Status tab, like other Composer Process Manager Console pages, displays up to twenty (20) records per page. If there are more than 20 records, a controls at the bottom the page allow you to move to the first page, the previous page, the next page, or the last page.

Process Manager Console: Statistics Tab

The Process Manager Console's Statistics tab provides a list of all processes and a count of Running and Completed process instances for each process. (See below.) On the Totals line at the bottom, you will see (from left to right) the total number of **Processes** (not process *instances*, but different process models), **Running** process instances, and **Completed** process instances. The processes are listed alphabetically by name on the left. Each name is a hot link that will take you to a Status page listing a status table filtered by the process name.

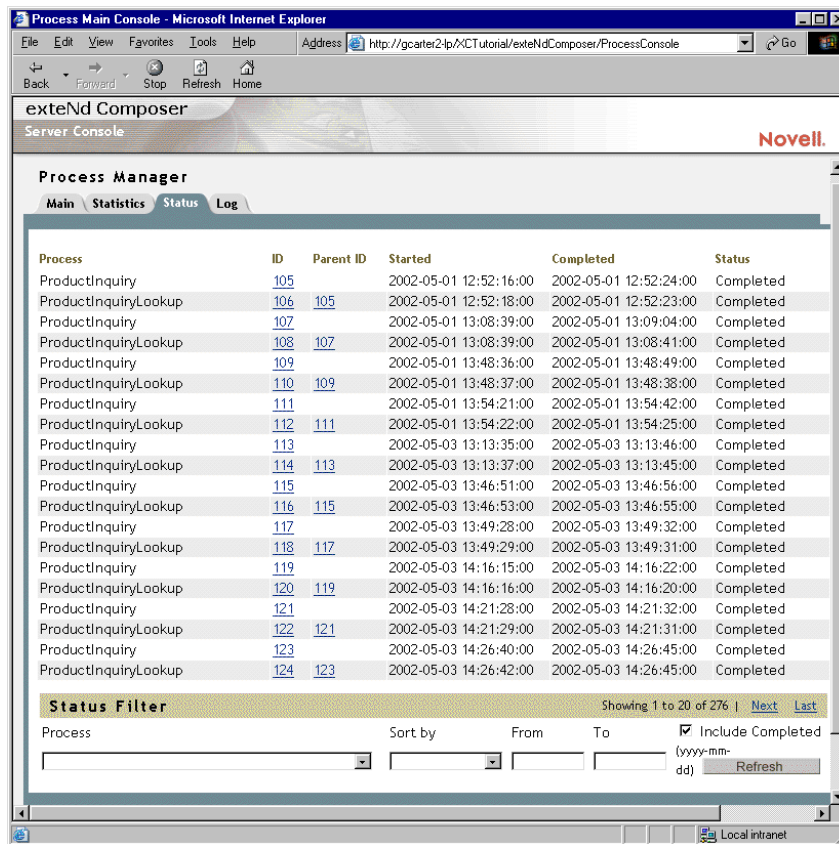
Process	Running	Completed
G SubProcess - Call WSR by ProcessName and CorrelationID	0	5
G WS Receive called by ProcessName and CorrelationID	0	3
G1 SubProcess - Call WSR by ProcessName and CorrelationID	0	5
G1 WS Receive called by ProcessName and CorrelationID	0	7
G2 SubProcess - Call WSR by Activity Name and ProcessID	0	9
G2 WS Receive called by Activity Name and ProcessID	0	9
ProductInquiry	0	15
ProductInquiryLookup	0	15
SubmitApprove	0	25
SubmitApproveWork	0	57
V Fan Out then Synch Subprocess In	0	2
V Fan Out Worker	0	14
WorkCycle	0	105
Z1	0	1
Z2	0	1
Z3	0	1
Z4	0	1
Z5	0	1
Totals:	18	276

Process Manager Console: Status Tab

The Process Manager Console's Status tab gives you a view of the overall execution status of all processes, filterable by process name and date range, with control over which field to sort by. The filter controls are at the bottom of the page.

As always, 20 result lines are displayed at a time. To page through the available results, click the **First**, **Prev**, **Next**, or **Last** links at the lower right corner of the page.

This view is not updated in real time. Therefore, a **Refresh** button is provided near the bottom right corner.



Process ID and **Parent ID** numbers are clickable links. You can “drill down” on a specific process instance by clicking the appropriate Process ID link. Clicking a link will open a new browser window in which the Process Detail page, described later, is presented.

Status Filter

The **Status Filter** control group (bottom of page) allows you to control how processes are displayed in the Status tab view. You can choose, for example, to display process instances for a specific named process using the **Process** drop-down control.

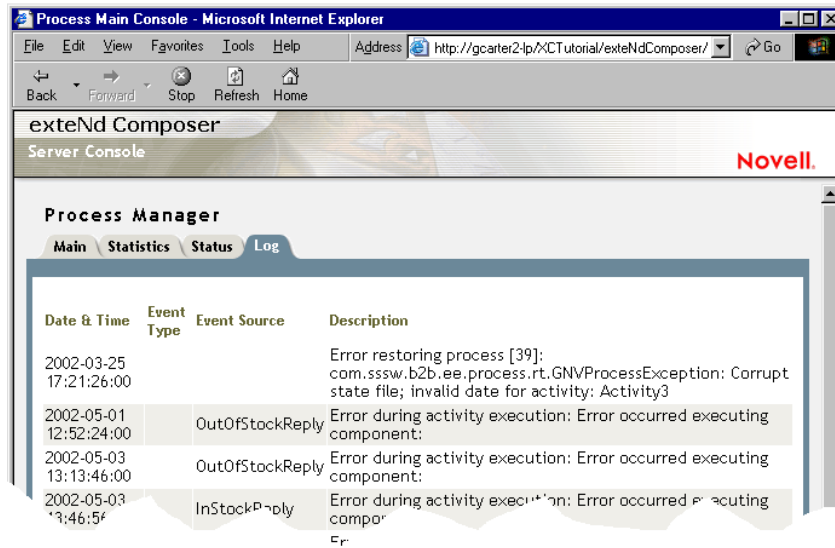
Using the **Sort By** control, you can sort the list of displayed processes by Process Name, Process ID, the Create Date/Time, the Modify Date/Time or the Status.

By entering dates in the **From** and **To** fields, you can display processes that were started and, optionally, completed within a specified range of dates. Check the **Include Completed** check box if you want to display records of finished processes.

After selecting your filter options, press the **Refresh** button to display a new list of processes based on your current Filter and Sort settings.

Process Manager Console: Log Tab

The Process Manager Console's Log tab displays log messages in the following format, sorted by Date and Time:



You may filter the view of the log by clicking on the checkboxes at the bottom of the page:



After making your desired choices, press the Refresh button.

Detail View for a Process Instance

When you click a link for a process instance (such as any of the links in the **ID** column of the **Status Tab** in the main process console), you will see an Activity Detail view for that process instance pop open in a new browser window. (Using multiple browser windows, you can monitor multiple process instances simultaneously.)

The detail view for a process instance has three tabs: **Activities Detail**, **Messages** and **Log**.



Process Detail: Activities Detail Tab

Activities Detail is the first tab on the detail page and is the default view when the window first opens.

In this tab, you'll see the Name, ID, Start Date/Time, Completed Date/Time, and Status of the individual activities that comprise the process instance. The process instance will be listed as either Running or Complete. If the process is running, buttons will be present enabling you to Suspend or Terminate the running process instance.

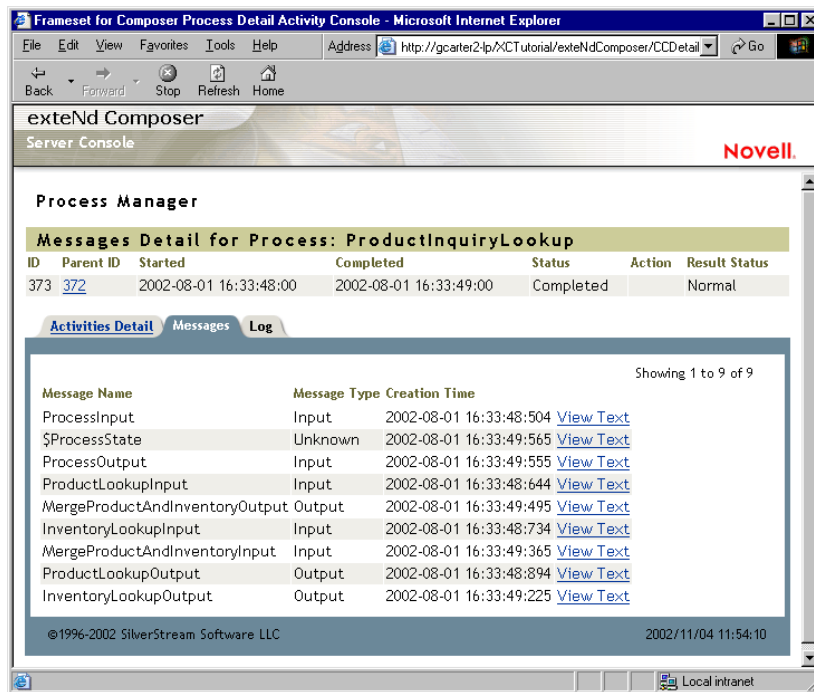
The columns in the Activities Detail tab view have the following meanings.:

Activity	This is the name of the Activity. The Activity names are hyperlinked to the Activity Data Monitor. The Activity Data Monitor displays Input and Output documents and their data values.	
Activity Type	This is the type of activity – Web Service, Subprocess, Composer Component or End Point.	
Started	This is the date and time when the activity started.	
Completed	This is the date and time when the activity was completed.	
Status	This column displays the Status of the activity.	
	Completed	After the associated operation has completed, continuation of the activity depends on its exit condition. If this evaluates to false, the activity is iterated, by either continuing with 'enabled' or 'running' depending on the associated operation. If the exit conditions evaluates to true, the activity reaches the 'Completed' state.
	Running	The Activity's state once it is started.
	Terminated	The Activity's status when the process completes before the Activity.

	Enabled	The flow engine decides that this activity instance could now possibly be executed and puts it into the 'Enabled' state. Depending on the nature of the activity and its associated operation, it might remain in that state until it is started through an explicit requests (e.g. for <i>in</i> or <i>in-out</i> operations), or the flow engine will start it right away (e.g. for <i>out</i> or <i>out-in</i> requests).
--	---------	--

Process Detail: Messages

The Messages tab gives you a view of the process instance's messages (i.e., input and output documents). The messages are sorted by name.



The **Messages** tab displays the following information:

Message Name	The name of the message.
Message Type	The message type may be either Input or Output
Creation Time	The time the message was created.
View Text	The View Text link displays the message in a new browser window, as shown below.

```

http://ckeller-lp/Process/extendComposerProcess/getMessageContent?pid=82&mesa...
<?xml version="1.0" encoding="UTF-8" ?>
- <EngineResponse>
- <Message>
- <![CDATA[
  <?xml version="1.0" encoding="UTF-8" ?>
  <MESSAGE>
  <Output>
  <PRODUCTRESPONSE>
  <SKU>DAD7777</SKU>
  <CATEGORY>Kitchen</CATEGORY>
  <NAME>Oak Butcher Block</NAME>
  <DESCRIPTION>Professional grade butcher block with kniv
  <MANUFACTURER>Dadio</MANUFACTURER>
  <LISTPRICE>295</LISTPRICE>
  <IMAGEFILE>Bblock.jpg</IMAGEFILE>
  <IMAGEWIDTH>411</IMAGEWIDTH>
  <IMAGEHEIGHT>271</IMAGEHEIGHT>
  <INVENTORYSTATUS>In Stock</INVENTORYSTATUS>
  </PRODUCTRESPONSE>
  </Output>
  </MESSAGE>
]]>
</Message>
</EngineResponse>

```

Process Detail: Log

The Log tab summarizes logged events of various types. See below.

The screenshot shows the 'Log' tab in the 'Process Manager' section of the 'exteNd Composer' server console. The main heading is 'Log Detail for Process: ProductInquiryLookup'. Below this is a table with columns: ID, Parent ID, Started, Completed, Status, Action, and Result Status. The log entry shows ID 375, Parent ID 374, Started at 2002-08-01 16:59:53:00, Completed at 2002-08-01 16:59:57:00, Status 'Completed', Action 'Completed', and Result Status 'Normal'. Below this table are tabs for 'Activities Detail', 'Messages', and 'Log'. The 'Log' tab is active, showing a detailed log table with columns: Date & Time, Event Type, Event Source, and Description. The log entries are as follows:

Date & Time	Event Type	Event Source	Description
2002-08-01 16:59:54:00	Process Started	ProductInquiryLookup	The process has started
2002-08-01 16:59:54:00		DL	executing data link: DL
2002-08-01 16:59:54:00	Activity Started	ProductLookup	The activity has started
2002-08-01 16:59:54:00		DL1	executing data link: DL1
2002-08-01 16:59:54:00	Activity Started	InventoryLookup	The activity has started
2002-08-01 16:59:54:00			activityReturn(375, ProductLookup)

At the bottom of the window (scroll down as necessary), you'll find checkboxes that you can use to control the types of events summarized in the table.

Log Filter Showing 1 to 19 of 19

Notice
 Debug
 Error
 Process
 Activity
 Refresh

Select the checkboxes of interest, then click the **Refresh** button to bring up a listing of logged events (appropriately filtered).

A Testing

Environmental Differences between Design-Time Testing and Server Testing

There are significant environmental differences between animation-based “step through” testing in Composer and server-side (deployment) testing. Both types of testing are needed, obviously, to verify the processes and services you build. Some environmental differences that you should be aware of are detailed in the table below.

Requirement	Testing in Composer	Testing on the Server
Console views and administrative monitoring	Not available in a non-deployed, design-time environment	Administrative consoles are available
Logging	Messages go to Output pane of main Composer window	Messages visible in Log tab of console
Testing of long-running processes	Not practicable in a design-time setting (some processes may take days or weeks)	Can and should be done here
Data persistence	No database required	Database must be configured for Process Server's use
Visual depiction of running process's state	Available at animation time (canvas view updates as process runs)	No canvas (graph) views in this release
Process instance info	Process IDs start at one at the beginning of each design session, then increment as new process instances are executed. With each launch of Process Designer, the Process ID numbering is reset to begin again at one.	Process IDs are generated continuously and never reset to one.
Runtime variables for: * Connection names * Client credentials * Log File Paths * DTD URIs * XSL URIs * Send Mail Server * XML Interchange URIs	Often point to locations on local machine, for design and test purposes	Should be set to point to locations on production Servers and Web
Triggering	Processes can be executed either from a Process Execute action inside an animating component, or directly from one of the animation toolbar buttons	Every process must be deployed with a service that can kick it off

B Performance Tuning

Configuration Options

Process Server performance can be tuned in various ways. The necessary adjustments are accomplished by editing the **xc_process_config.xml** file. For a SilverStream app server install, this file would be located in (for example) **D:\Silverstream3.7\extendComposer\lib**.

Cache

The Process Server cache is managed by changing the values of `<PROCESS_CACHE>`.

Sleep Time

The value of the `<SLEEP>` element controls the number of seconds the Process Server waits in a delay loop before checking to see if any in-memory processes have exceeded their `<CUTOFF>` period. (See below.)

Cutoff Period

The value of the `<CUTOFF>` element controls the maximum number of seconds a process is allowed in memory without any activity. If the `<CUTOFF>` for any in-memory process is exceeded at the end of a `<SLEEP>` period, then the process will be purged from memory. It is, however, still in persistence in the database and can still reenter an executable state, albeit more slowly than if in memory.

Total In-Memory Process Instances

The value of the `<SIZE>` element controls the the maximum number of processes that will be allowed in memory before swapping occurs.

C

Process Management Glossary

Activity An activity is a unit of work within a process model, representing a business task. On an operational level, an activity is a named operation with a signature that specifies the inputs, outputs, and possible faults associated with the operation. The activity is separate from its implementation. The implementation (which can be any Composer component type, or any Component service) performs a task on behalf of the activity.

Addressee The Addressee property (which exists only on the Web Service Receive activity type) provides a way to tag activity instances with a label, typically corresponding to the name of an individual in the organization.

Asynchronous A mode of operation in which work is done *independently and in parallel* with other work. (That is, there are no time-order dependencies between parties.) In software terms, an asynchronous task executes in its own thread. The term “fire and forget” is often used when referring to a process that has been spawned asynchronously. See also *Thread* and *Spawn*, below.

Business Process Management (BPM) *Business Process Management* is the study of ways to model business functions in terms of their component activities and participant roles.

BPML Business Process Modeling Language: an XML grammar for describing workflow, created and managed by the Business Process Management Initiative (<http://www.bpmi.org>). It is roughly comparable in scope to WSFL. Process Manager follows WSFL closely; it does not adhere to BPML.

Call A *call* event is one of two lifecycle events that can invoke an instance of a process. (The other such event is *spawn*; see below.) Unlike a spawned process, which returns an instance ID immediately, a called process does not return until the process flow has completed. A call operation implies synchronous processing, whereas a spawn operation is equivalent to “fire and forget.”

Choreography A particular set of sequenced operations is often colloquially referred to (in a business-process context) as a choreography. See also *PIP*, below.

Control Link A control link is the WSFL construct that defines a single step in the flow of control from one activity to another. It specifies the “activity traversal order” so that the workflow engine knows how to get from a given activity to the next one in sequence.

Correlation ID In Process Manager, a Correlation ID is an arbitrary user-specified string or number that can be used to associate data in a given message part with a transaction context. Correlation ID is a common term for this kind of user-defined label, but it is not a formal WSDL or WSFL concept.

Cyclic Graph A *cyclic graph* is a graph that permits links from downstream nodes back to upstream nodes, forming a loop. Such graph patterns are not allowed in Process Manager.

Data Link A data link is an atomic unit of data flow, specifying one or more data sources along with one or more data targets. The sources and targets are activities within the running process. While in most cases data flow will mirror control flow, it is possible that data can bypass certain activities in a flow or arrive at a target by a more direct path than might be specified in the control flow. Hence, data links do not always follow control links.

Dead-Path Elimination Dead-path elimination refers to the special lookahead operation that the Process Server conducts every time a conditional expression (a link condition, for example) evaluates to false. When flow along a given path is no longer possible due to a false link condition, all downstream links must be marked as false so that joins can be evaluated in the course of operation. (The path goes from being dead to being known-false.) If this were not done, downstream joins could hang indefinitely.

Exit Condition An exit condition is a boolean value (determined by runtime evaluation of user-supplied XPath logic) that indicates whether a given activity executed normally. Outgoing control links cannot be followed until and unless the exit condition is true. If the exit condition is false, the activity will execute again (if allowed by the timeout and retry settings).

Factoring In programming, factoring is the attempt to split code into smaller, more generic (and thus reusable) units of work.

Fan-Out A type of execution pattern in which a collection of N discrete works items gives rise to an asynchronous invocation of N instances of a particular process designed to work on the work items.

FlowInstanceID Every WSFL process that is invoked via a *spawn* operation is required to return a unique FlowInstanceID to the caller immediately. This ID can be a timestamp or can be an arbitrary string, but it must uniquely identify a particular instance of a running process. This value is used as the input value of other lifecycle operations (such as *enquire*; see Lifecycle Interface, below).

Flow Model The flow model is the XML representation of the directed graph that models the business process. In other words, it is the all-encompassing set of activities, control links, and data links that comprise a given process. A flow model makes the choreography of a process explicit, such that an execution engine can instantiate the process at runtime and understand how to manage the flow of control over the process's lifecycle.

Graph An abstract visual representation of a system of nodes. In Process Manager terms, a *graph* of a process is what you draw on the Process Designer canvas.

Implementation The concrete realization in software of an activity. Every activity must have an implementation.

Join Condition When two or more activities target the same successor activity, the decision of whether the successor activity can be invoked may depend on factors that can be evaluated only when upstream activities have finished executing. The runtime engine makes this decision based on user-supplied logic in a *join condition*. The join condition takes as input the respective boolean values (or "truth values") of incoming links. It performs some user-specified set of logical operations on the link values and returns true or false. A true condition means that the join target will be invoked. False means that control flow ends at the join. Note that unlike link and exit logic (which both use XPath), join logic is expressed in a simple pseudocode-like boolean logic. The join condition, in other words, has no knowledge of messages or message parts (nor any data whatever). It only knows about link boolean values.

Lifecycle Interface The Lifecycle Interface is the WSDL-defined web service interface that describes the basic set of operations that all WSFL processes must support. These operations include *spawn*, *call*, *suspend*, *resume*, *enquire*, and *terminate*. These operations are global in scope (they apply to the process-as-a-whole) and can be managed administratively.

Link Condition A link condition is an XPath expression that resolves to a boolean value. Its value determines whether a given link can be traversed by the process engine at runtime. The XPath expression typically utilizes data from an upstream activity's output.

Long Running Some processes can take days or weeks to run to completion. Such processes are called *long-running*.

Map Policy A map policy specifies how data should be mapped in the special case where two or more data links target the same message part(s). A policy of Last Writer Wins (LWW) means that newly arriving data will overwrite older data. A policy of First Writer Wins (FWW) means that once any data have been written, data arriving later will be ignored. Map Order means that for any given incoming message, XPath-to-XPath mappings will occur in the order listed in the Messages tab of the activity's property sheet, completely ignoring timestamps.

Message In WSFL and WSDL, a *message* is an abstract definition of a bundled set of data. The logical parts that are bundled together as part of the message structure are known as *message parts* (see below). Activities operate on messages; hence, the interface to an activity can be specified in terms of its input and output messages.

Message Part In WSFL and WSDL, a *message part* is a logical unit of a message. In Process Manager terms, the parts correspond to XML documents that activity implementations can inspect, modify, and transform into new parts of new messages.

Metadata Data about data. In Process Manager terms, the *metadata* representation of a process is a non-visual XML representation of a given process's actual structure and attributes. The metadata blueprint of a process is used by Process Server to construct process instances at runtime.

Notification A one-way operation is a web service execution pattern in which the service proactively sends a message, with no expectation of a response. It is "fire and forget" pattern. However, it is often used in conjunction with the One-Way pattern (see below) in order to complete an asynchronous roundtrip communication with a partner. In such a case, the web service that implements the Notification pattern will typically embed correlation information in the outgoing message, so that information received in a later One-Way operation can be "matched up" with the transaction context of the Notification. See also *One-Way*, below.

One-Way A one-way operation is a web service execution pattern in which the endpoint receives a message (but does not send one back to the initiator). The one-way web service is a passive receiver. See also *Notification*.

Operation In WSDL, an operation is a specified sequence of message transfers (described in terms of named input messages and output messages). See also *Port Type*, below.

PIP[®] RosettaNet Partner Interface Processes: a set of *de facto* industry standards that define business interaction patterns between trading partners. The interaction patterns include sequencing and timeout rules for various kinds of common business transactions. The patterns, because of their intricate sequencing (time domain) requirements, are often called *choreographies*.

Port Type In WSDL, a *port type* is a named set of operations. (An operation, in turn, is a specification of a particular time-order sequence of particular messages.) Four port types are supported by WSDL: One-Way, Request-Response, Solicit-Response, and Notification. (See individual definitions of these items.)

Process A description of the activities, control-flow patterns, and data-flow relationships involved in performing a particular business task. WSFL (see below) describes processes as *web services compositions*. It is assumed, in WSFL, that processes (or workflows) are automated.

ProcessID A number that uniquely identifies a process instance (a running process) within the Process Server at runtime.

Request-Response A request-response operation is a web service execution pattern in which the service receives a message, then sends a (correlated) message back to the initiator. The request-response web service is a passive receiver. It responds with an output message.

RosettaNet A non-profit industry organization dedicated to “the adoption and promotion of open content and open transaction standards in electronic commerce across the Information Technology (IT), Electronic Components (EC) and Semiconductor Manufacturing (SM) supply chains.” See <http://www.rosettanel.org/> for details.

Semaphore A flag value meant to signal the availability or unavailability of (typically) a function or file, in the context of the file’s lock status.

Service Provider A service provider is the party responsible for performing a particular activity within a business process.

Service Provider Type In order to maintain separation between the definition of a business process and its implementation, WSFL defines activities as being implemented by abstract service provider *types* rather than by specific service providers (which can later be mapped to the types). The service provider type and its associated interface are defined by a WSDL document. Service providers must properly implement a given web service interface in order to handle a particular activity in the business process.

SOAP Simplified Object Access Protocol: a lightweight XML-based protocol for exchange of information in a distributed environment. The protocol definition consists of three parts: an envelope that defines a framework for describing what is in a message and how to process it, a grammar for specifying application-defined datatypes, and a grammar for representing remote procedure calls and responses.

Solicit-Response A solicit-response operation is a web service execution pattern in which the service *sends* a message proactively, then receives a response. In this scenario, the web service is an *initiator* of a transaction. Since a response from a participant is required as part of the pattern, this type of web service is assumed to execute synchronously. (That is, on sending its message, it blocks until the reply message comes back.) See also *Request-Response*, above.

Spawn Spawn is a WSFL-defined lifecycle operation that allows one-way (asynchronous) invocation of a process. (The corresponding *synchronous* launch event is the *call* event. See further above.) When a process is spawned, it returns a result (the Process ID) immediately.

Subprocess A process that has been called by another process.

Synchronize Subprocesses Activity The Synchronize Subprocesses Activity is one of Process Manager’s core activity types. It is a special-purpose activity type designed to aid in the collection and collation of data from multiple spawned instances of a component. The implementation to the activity is often called a “merge component,” because it typically merges incoming data. The Synchronize Subprocesses Activity thus constitutes the “fan-in” piece in a fan-out/fan-in scenario.

System Fault The runtime engine raises a *System fault* when an activity implementation generates an unhandled exception; *or* a subprocess activity returns a fault message; or the runtime engine encounters a message or message type that it doesn’t know how to handle; or a Timeout fault occurred and was not handled by an activity designed for that purpose. (In this case, two faults are actually generated: one Timeout and one System.) When a System fault occurs, the process instance produces a message called *_SystemFault*, with a part name called (also) *_SystemFault*.

Thread An execution context with no time-order dependencies on other operations occurring in other contexts.

Transition Condition (Link Logic) As a process is run, the execution engine must be able to recognize when a particular activity is finished, identify the next activity in the flow, and make a decision as to whether the next activity should be invoked or not invoked, based on user-specified transition logic. A *transition condition* determines whether flow should continue along the current path. The transition condition is specified in XPath and always evaluates to true or false.

UDDI Universal Description, Discovery and Integration specification (maintained by <http://www.uddi.org>). A scheme for exposing business services via web-based registries.

Waiting Activity Any time an activity (such as a subprocess or Web Service Receive activity) is in a *wait state*, waiting to receive a response to some request that was made asynchronously by another activity, it is said to be a *waiting activity*.

Web Service Receive The Web Service Receive Activity is one of the core Process Manager activity types. It is a passive, “listening” activity type meant to implement the Request-Response or One-Way transaction patterns described by WSDL.

Web Services Composition A process model based on web services. Essentially, any WSFL process.

Workflow In the context of BPM, a workflow is a process. WSFL favors the term *process* because its authors anticipate that most automated workflows will rely, ultimately, on Web Services. (In more traditional workflow systems, activities tend to center around human-mediated activities.)

WSDL Web Services Description Language: An XML format for describing web services as a set of endpoints operating on messages. The operations and messages are described abstractly, then bound to a concrete network protocol and message format to define an endpoint. Related concrete endpoints are combined into abstract endpoints (services). Services are thus defined using six major elements: *types*, *message*, *portType*, *binding*, *port*, and *service*.

WSFL Web Services Flow Language: An XML format for describing workflow processes as linked activities. The activities may be web services, or other workflow processes.

Index

Symbols

_TimeoutFault 77, 116

A

action

- Browse Waiting Activities 139
- Lock/Unlock Waiting Activity 141
- Process Execute 129
- Reassign Addressee 143
- Release Waiting Activity 131, 135

activities

- creating 65
- finding 132

activity

- end 26
- fan-out 50, 119
- fault handler 76
- lookup 132
- renaming 66
- source vs. target 41
- start 26
- Synchronize Subprocesses 119
- types 26
- waiting 124
- Web Service Receive 101, 118
- Web Service Send 99

activity detail 155

activity icons 87

activity implementation 131

Activity Tool 65

Addressee 124, 135, 137, 151

administration 147

- Addressee views 151
- lock info 151
- Process Database Info 148
- Process Engine Info 148
- queue 150
- statistics 148

algorithm, execution 39

algorithm, process execution 39

alignment to grid 111

Alt key and grid alignment 111

AND Split 44

animation 78

animation and deployment testing 159

architectural summary 53

asynchronous fan-out 49

autodiagramming 113

autolayout 113

B

background image 112

batch processing 119

best practices 54

bombsight view 85, 111

BPM, justification for 20

branch logic 43

breakpoints 79

Browse Waiting Activities 124

Browse Waiting Activities action 139, 140

Business Process Management (BPM) 19

C

call 33, 129

Call vs. Spawn (Process Execute) 128

canvas

background images 112

customization 112

choreography, retry 73

circular layout 110

collisions, data-mapping 74

Comparing Browse and Find 140

COMPONENT_FAULT_SUBCODE 75

compound branch logic 44

concurrent processing 49

condition

join 72

conditional branching 43

conditions

exclusive-OR 45

exit 40, 71

configuration of database 11

Configure Database 149

configuring the engine's database 12

connection pools 11

Consoles, Process Manager 147

CORR ID 150

Correlation ID 132

custom grid size 111

customization 112

Cutoff Period 161

cyclic graph 68

D

- data links 53, 68
- data mapping 68
- data merging 32
- database
 - configuring Process Engine 148
 - synchronization 12
 - WebLogic setup 12
 - WebSphere setup 12
- DB2 11
- dead links 31
- dead path elimination 31
- debugging 78, 81
- Deferred Mode 30, 31, 40, 45, 46, 72
- Delete Process Info 147, 149
- deployment 130
- Device Offset 112
- DoBatch 51
- document purging 149
- DOM view 83
- dynamic fan-out 49

E

- edge routing 113
- End Activities 26
- engine 148
- engine (see also Process Server) 148
- engine database 11
- environmental differences 159
- error, link creation 46
- Exit Condition
 - specifying 71
- exit condition 29, 40
 - specifying 72
- Expression Builder 70, 72
- external data store 48

F

- factoring 21
- Fail on First Fault 122
- fan-out 119, 130
- fan-out component 50
- Fan-Out/Fan-In 49
 - recursive 51
- FAQ 35
- fault 82
- Fault Codes 75
- Fault Handling 76, 122
- fault handling 75
- Fault Messages 75
- filter 153
- filter criteria (admin) 153
- Find Waiting Activity. 123
- First Writer Wins 74
- First writer wins (FWW) 32
- FWW 74

G

- grid behavior 111
- GVXMLProperties_process 59

H

- Hierarchical Layout 113
- hierarchical modelling 53
- human interaction scenarios 124
- human participation in processes 136

I

- icons, activity 87
- images, background 112
- Immediate Mode 30, 40, 72
- Immediate Mode, 45
- implementation, activity 66
 - tasks vs. 27
- Initialize Database 149
- Initialize database 12
- initialize database 149
- input message
 - named same as output 47
- Input1 DOM 121
- inquire (lifecycle event) 33
- installation, databases and 12
- Internal Revenue Service 45
- Invalid Configuration message 11
- iterating on an external data store 48

J

- JMS Components 48
- JMS Receive action 48
- JMS Service 50
- JMSDestination 48
- JMSMessageID 48
- JNDI Name, connection pools and 11
- join condition 29, 45
- Join Logic 45
- jpeg or .gif image on canvas 112
- JSP 137, 138, 139, 142
- Jump to Process 149

L

- Last Writer Wins 74
- Last writer wins (LWW) 32
- layout modes 113
- lifecycle events 33
- link
 - conditions, specifying 67
 - creation 67
- link condition 67
- Link tool 88

- links 28
 - auto-alignment 113
 - backward-facing 68
 - creating 67
 - data 68
 - triangle shape 77
 - XPath 67
- Lock Duration 143
- lock until 150
- Lock Waiting Activity 151
- Lock/Unlock Waiting Activity 125
- Lock/Unlock Waiting Activity action 141
- LockedBy 135, 141, 143
- LockedUntil 135, 141
- log 155, 157
- log messages 82
- Log Tab 154
- logged events 157
- logic, link 67
- lookup methods, activity 132
- looping 46, 47
 - asynchronous 49
 - reentrancy and 68
- LWW 74

M

- MainCode 75
- Manage Activity Queue button 150
- Map Order 32, 74
- Map Policy 32, 74
- mapping 68
 - start activity 71
- mapping an activity to itself 47
- merge component 50, 120
- Merge Edge Channels 113
- message
 - fault, contents of 82
 - parts 28
 - show/hide 83
 - Timeout fault 76
- message maps 68
- message naming 68
- message parts 28
- messages 27, 155
- Messages Tab 97
- metadata description 33
- multiple Undo 111

N

- naming conventions
 - message 68
- navigation of consoles 151
- New Process 59
- Non-Exclusive OR Split 44

O

- Object Properties 42
 - Addressee 124
- Object Properties panel 43, 67, 85, 94
- ODBC data source 11
- Oracle 11
- ORSplit 44
- Orthogonal Layout 110
- orthogonal routing 113
- overview of Process Manager 53
- overview pane 85
- Overview Window 111
- overwrite policy 74
- owner 150

P

- panning 111
- parallel processing 49
- Parent ID 130
- Passed Part, 129
- pending processes (admin) 152
- PendingActivity document 135, 139, 141
- performance 36
- picture, adding to canvas 112
- policy, overwrite 74
- portType 42
- Priority 124, 135
- process
 - create 59
 - human interaction with 136
 - input template 71
 - invoking via action 128
 - new 59
 - triggers 127
- process architecture summary 53
- Process Database Configuration 12
- process database info 148
- Process Designer GUI 85
- process engine database 148
- Process Execute action 128, 129
- Process Manager Architectural Layers 33
- Process Model Pane 85
- Process Properties 91
- Process Server
 - database 11
- Process Server Execution Model 39
- Process Statistics Summary 148
- ProcessID 39
- ProcessInput, mapping to 71
- ProcessOutput, mapping to 71
- ProductInquiryProcess 57
- property sheets 41
- purging of documents 149

Q

- queue 150
- Queue Status Tab 150
- QueueDate 135
- queues 128
- QuickFilter 153

R

- Reassign Addressee 124
- Reassign Addressee action 143
- recursive process graph 51
- reentrancy 46
- reentrant loops 68
- Release Waiting Activity 123
- Release Waiting Activity action 131, 135
- rename activity 66
- Request-Response pattern 115
- resume 33
- resynchronization 119
- resynchronizing database 11
- Retry Count 73
- Retry Interval 73, 76
- Retry Interval, 73
- Returned Part 129
- Run to Breakpoint 79
- runtime execution algorithm 39

S

- scenarios, work-group 138
- sequencing 28
- server start/stop 36
- Service Providers
 - adding types 64
- Set Breakpoint 79
- setup, database 12
- Shapes tool 89
- SilverStream app server 12
- Sleep Time 161
- snap behavior 111
- SOAP trigger 127
- spawn 33, 41, 129
- Spawn (Process Execute) 128
- Spawn as Subprocess of Parent ID 130
- Spawn as Subprocess... 129
- split-or-work strategy 52
- Start Activities 26
- start/end activity mapping 71
- statistics 148, 150, 152
 - filter/sort 153
- Status Tab, ,admin console 152
- status, queue 150
- Step Into/Over 79
- Sticky Tools 111
- SubCode 75
- Subprocess 88
- summary of WSFL workings 53
- summary statistics (admin) 148

- suspend 33
- Sybase 12
- Synchronize Subprocesses Activity 52
- Symmetric Layout 110
- synchronization 50
- synchronization failure 31
- synchronization logic 28
- synchronize database 12
- Synchronize Subprocesses Activity 88, 119, 130
- System Faults 75
- System log 82
- SYSTEM_FAULT_MAINCODE 75

T

- Tasks vs activities 27
- templates
 - process input 71
- terminate 33
- Terminated By 149
- testing and debugging 78
- Text tool 88
- threaded subprocesses 50
- Tile Picture 112
- Timeout 32, 73, 76
- Timeout Faults 76
- TIMEOUT_FAULT_MAINCODE 75
- toolbar 86
- tools
 - link 88
 - sticky mode 111
 - text 88
- tools,shape 89
- transition condition 29
- Tree Layout 110
- triangle link icon 77
- trigger types 127
- troubleshooting
 - database synchronization 12

U

- UNHANDLED_MESSAGE_SUBCODE 75
- user access to queued work 136

V

- viewport rect 111

W

- Waiting Activities 122
- Waiting Activity 124
- Web Service Receive 88, 101, 115, 118, 131, 132, 137, 138
- Web Service Send 88, 99
- Web Services Flow Language 25
- WebLogic-specific setup info 12
- WebSphere-specific setup info 12

- work items 125, 150
- work queues 128
- workflow 19
- workflow models, human-centric 136
- workflow, human 124
- workgroups 136
- workhorse process 50
- World Offset 112
- WSDL 27, 42, 118
- WSFL 19
 - best practices 54
 - looping and 46
 - summary of key points 53

X

- XML Template 71
- XOR join 29
- XOR-Split 43
- XPath 29, 70
 - in links 67
- XSL 159
- x-y alignment of links 113

Z

- zooming, interactive 111

