Infor LX ION PI Builder User Guide

Contents

Contents

About this guide

This guide provides instructions and examples to use the LX ION PI Builder to create process instructions to use with the Infor LX Extension or the LX Connector.

Intended audience

This document is intended for programmers who intend to use the LX Extension or the LX Connector to program integrations between LX and other applications or third-party products. The tool automates the process of creating inbound and outbound process instructions required by the integrations.

Related documents

You can find the documents in the product documentation section of the Infor Support Portal, as described in "Contacting Infor" on page 13.

Contacting Infor

If you have questions about Infor products, go to Infor Concierge at https://concierge.infor.com/ and create a support incident.

The latest documentation is available from the Infor Support Portal. To access documentation on the Infor Support Portal, select **Search > Browse Documentation**. We recommend that you check this portal periodically for updated documentation.

If you have comments about Infor documentation, contact documentation@infor.com.

Chapter 1   Getting started

This chapter introduces the Infor LX ION PI Builder.

Description of the tool

The LX ION PI Builder is a Java tool that allows you to create process instructions used by either the LX Extension or the LX Connector to process business documents. The LX ION PI Builder is an Eclipse plug-in that allows you to create inbound process instructions, outbound process instructions, and exit point process instructions.

Inbound process instructions are used by the LX Extension and LX Connector runtime code to send data from a Business Object Document (BOD) message into LX. An inbound process instruction can consist of multiple steps. For example, you might require a process instruction that can navigate through screens and invoke an API. The LX ION PI Builder provides the ability to map BOD elements to screen fields, database fields, and APIs.

Outbound process instructions are used by the LX Extension and LX Connector runtime to create a BOD from data retrieved from LX. The process instruction can consist of multiple steps that are used to build a BOD message. The LX ION PI Builder allows you to map LX fields from the database to elements in an Infor BOD. Outbound messages are initiated by exit point programs or file triggers.

The LX ION PI Builder enables you to create exit point process instructions that provide a mapping of LX data to elements in an outbound process instruction that is used to build the outbound BOD.

Prerequisites

This software must be installed on your PC:

- Eclipse modeling tools

  Download the Eclipse Modeling Tools version that is compatible with your Windows OS version. We recommend that you use the 4.7 (Oxygen) version of Eclipse Modeling Tools.

  This software is available on the Eclipse download site:

  http://www.eclipse.org/downloads/

Installation instructions, prerequisites, requirements, and frequently asked questions for the Eclipse Modeling Tools are also available at the above site. Additional information about the specific Eclipse Modeling Tools release can be found in the product's readme folder.

Note: Ensure that the Java Runtime Environment type matches the Eclipse Modeling Tools type or Eclipse will not start. If the 64-bit Eclipse was installed, the 64 JRE should be installed.

Use the LX ION PI Builder 1.1.0 with these Infor products:

- LX Connector
- LX Extension 3.0

Installation

Download the latest version of the LX ION PI Builder User Guide for details on prerequisites and installation instructions.

We recommend that you start Eclipse with parameter -clean to clean the cache of any previous version registry data. The use of the -clean parameter is only required on the first start of Eclipse following the installation of a new version of LX ION PI Builder.

The LX ION PI Builder is delivered in the `PIBuilder.zip` file. The zip file contains these jar files:

- com.infor.lx.soa.adapter.developer_1.1.0_*vyyyymmddbnnnnn*.jar
- com.infor.lx.soa.adapter.developer.editor_1.1.0_*vyyyymmddbnnnnn*.jar
- com.infor.lx.soa.adapter.developer.edit_1.1.0_*vyyyymmddbnnnnn*.jar

In the JAR file names *vyyyymmdd* represents the build date and *bnnnnn* represents the build number that changes with each build.

Each JAR file is versioned according to the Eclipse versioning guidelines. Extract the contents of the zip file to the installed Eclipse directory. For example, if you have installed eclipse in a folder named eclipse4.70, then extract to the eclipse4.70 directory. The jar files are extracted into the plugins directory.
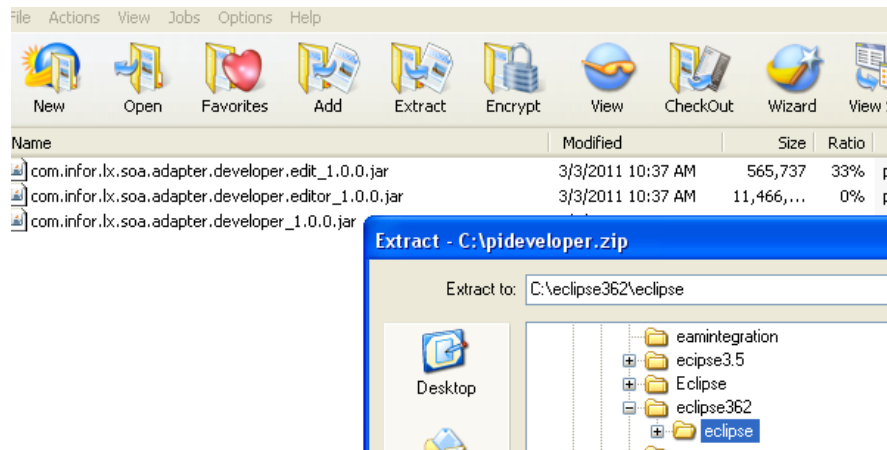
The tool also requires installation of the `pibuilder_tools.zip` file. The zip file includes files used by the pibuilder plugin. We recommend installing the zip file onto the `C:` root drive of your PC. The installation creates this `pibuilder_tools` directory:
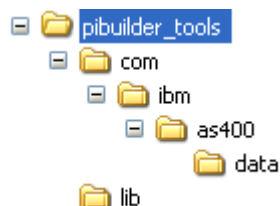
Creating a project and model object

To build a process instruction first create a project folder and then the model object. The model object allows you to add nodes to the tree in the Eclipse designer view. Each node added to the tree provides an instruction that ultimately is used by the runtime to process an inbound BOD message or to create an outbound BOD message.

Creating the project folder

Create a project folder for your integration project. If you will have both inbound and outbound process instructions, you can create a folder for each.

1   To switch to the Resource perspective, select **Window>Open Perspective>Other>Resource**.

2   Select **File>New>Project**.

3   Navigate to the General folder and select `Project`.

4   Click **Next**.

5   Specify the project a name, for example, the name of your integration project.
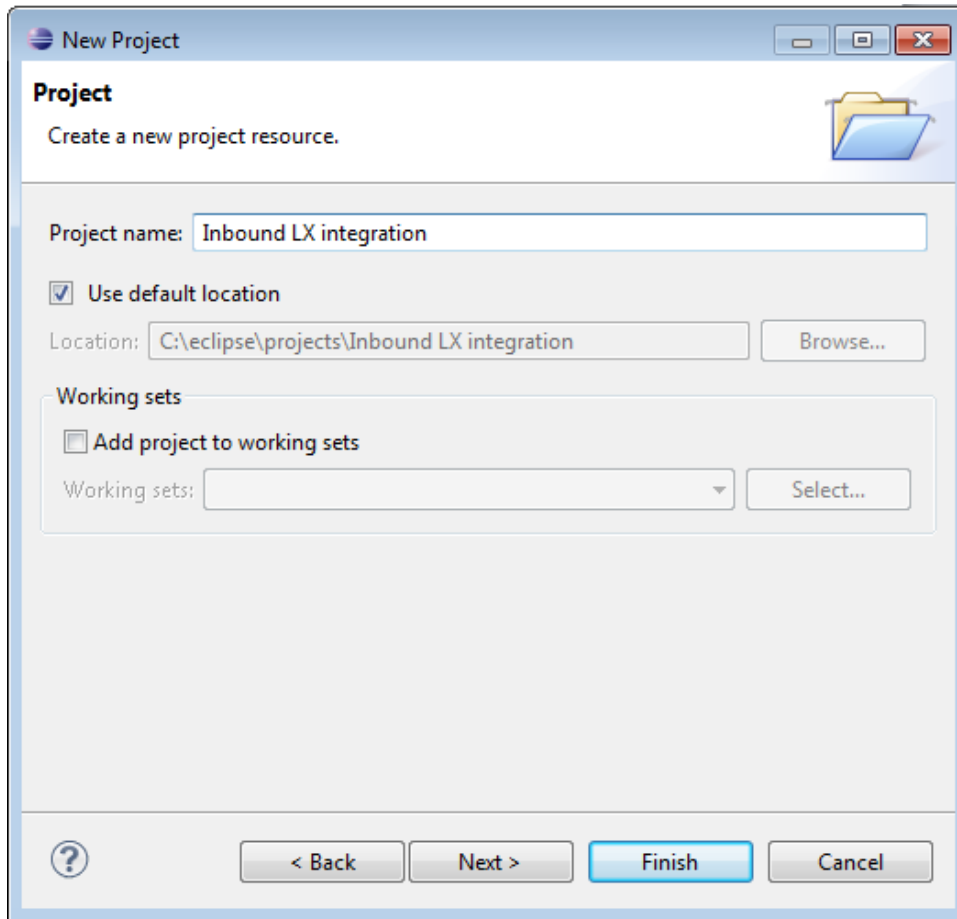
6 Click **Finish**.

Creating the model object

The steps to create a model object depend on whether you are creating an inbound or outbound process instruction. See Chapter 3 for a discussion of the techniques available to create an inbound model object. See Chapter 4 for instructions to create an outbound model object.

Using the Designer view

The designer view, also called the Tree view, is the repository for the instructions in the Model Object. You create the model object, add nodes to the model object, and define the node properties. Use the views for additional processing.

1 Select a node.

2 Right-click and select a Child or Sibling node. The list displays the child and sibling nodes that are valid for the selected node.
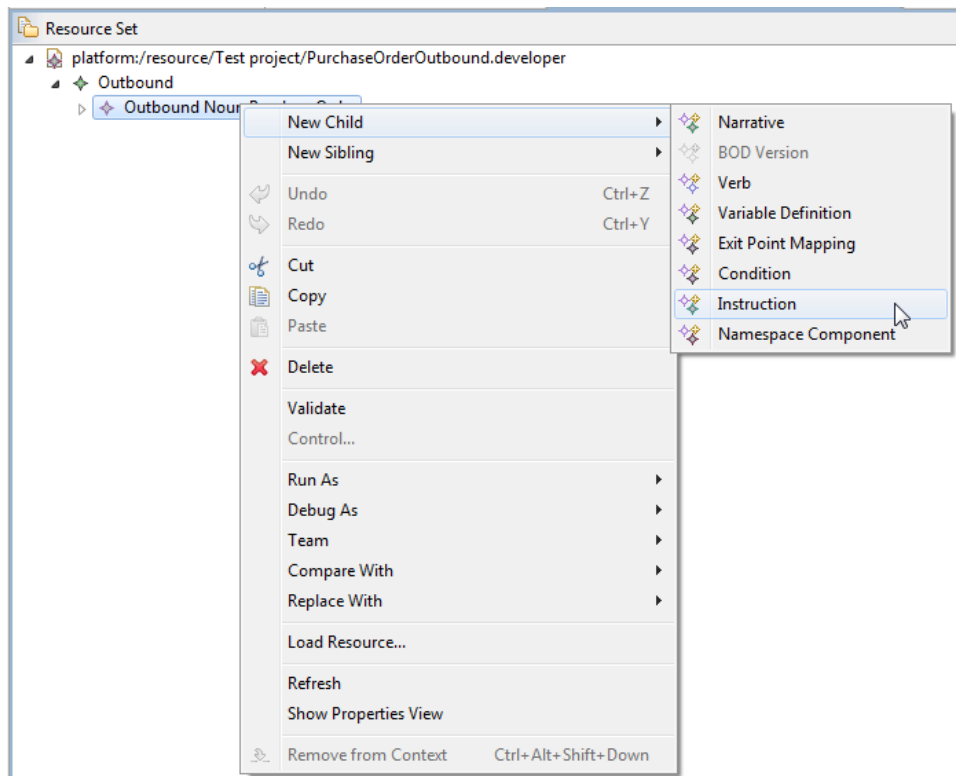
3 Right click the new node and select **Show Properties View**.
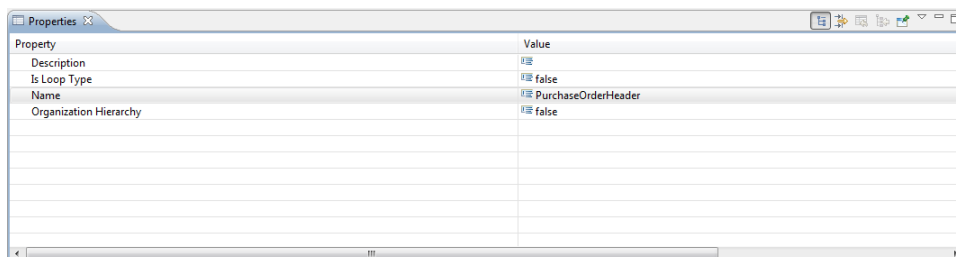
4 The list shows the properties that you can define for a node. Specify the properties as appropriate for your project.

5 Continue to add nodes to the tree. Use the views as described below to search for and add additional information.

Using LX ION PI Builder views

The LX ION PI Builder includes views that are provided by the eclipse plugin. Each view provides a specific function that you can use when you create a process instruction. The views provide this functionality:

- Map Elements from an inbound BOD message to LX legacy applications
- Map LX database fields to create an outbound BOD message.
- Add files to process instruction jar files
- Create SQL statements
- Create expressions
- Map fields from an exit point project into an outbound project.
- Edit comments in the tree

Within the pibuilder plugin, access the views from the **Window>Show View>Other>Infor ERP LX Views** menu.
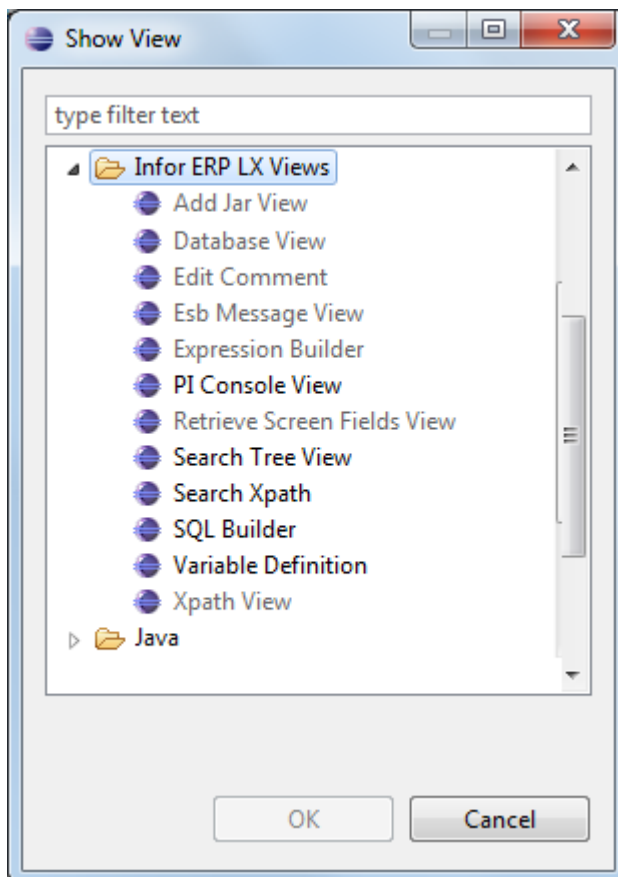
These views are available:

- **Add Jar**: Add files into process instruction jar files if a project uses IBM's Program Call Markup Language (PCML) to invoke LX APIs from the process instructions.
- **Database View**: Map an LX field to an Element that is added to an outbound BOD message.

- **Edit Comment**: Use this view to add comments into the tree to provide information about an instruction.
- **ESB Message View**: Test a message by sending it to the Inbox or Outbox.
- **Expression Builder**: Add logical and arithmetic expressions into the process instruction.
- **Retrieve Screen Fields View**: Retrieve data from LX that is used to build inbound and outbound process instructions.
- **Search Xpath**: Map elements in an inbound and outbound process instruction if a BOD template is available. BOD templates are available for LX Extension integrations that use ION for communications.
- **SQL Builder**: Add SQL statements into the process instruction. The statements are executed at runtime.
- **Xpath View**: Map element information into an instruction if a BOD template is available. BOD templates are available for LX Extension integrations that use ION for communications.

See the following topics for more information about the views.

Add Jar View

The LX Extension and LX Connector runtime uses PCML to process API calls to LX programs. If your project requires use of such APIs, create a process instruction that maps fields in the LX program to elements that can be used by either the inbound process instruction or outbound process instruction. These API process instructions produce PCML files that must be placed into the appropriate jar file. See Appendix A and Chapter 5 for information to create API projects.

Complete these tasks before you use this view:

- Set the `JAVA_HOME` environment variable in **System Properties**.
- Ensure the `pibuilder_tools` directory was installed to your PC.
- If building LX Extension process instructions, copy the `LXESBPI.jar` file to a directory on your PC. If building LX Connector process instructions copy the `LXCPI.jar` file to a directory on your PC.
- Generate PCML files from the API Project using the instructions in Appendix A.

1   Select **Window > Show View > Other > Infor ERP LX Views > Add Jar View**.
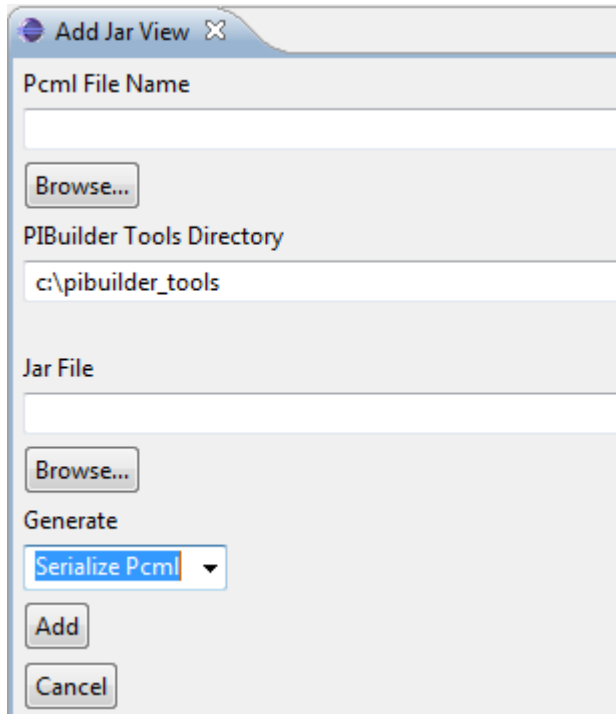
2   Specify this information:

Pcml File Name

Browse to the generated PCML file.

PIBuilder Tools Directory

Specify the name for the directory path to the `pibuilder_tools` directory.

Jar File

Browse to the jar file into which to add the serialized PCML file. If the API is added for an ION integration project, select the `LXESBPI jar` file in the directory into which you had placed the copy.

Generate

Select **`Serialize Pcml`**. All generated files with the PCML extension should be serialized for performance reasons.

3   Click **Add**.
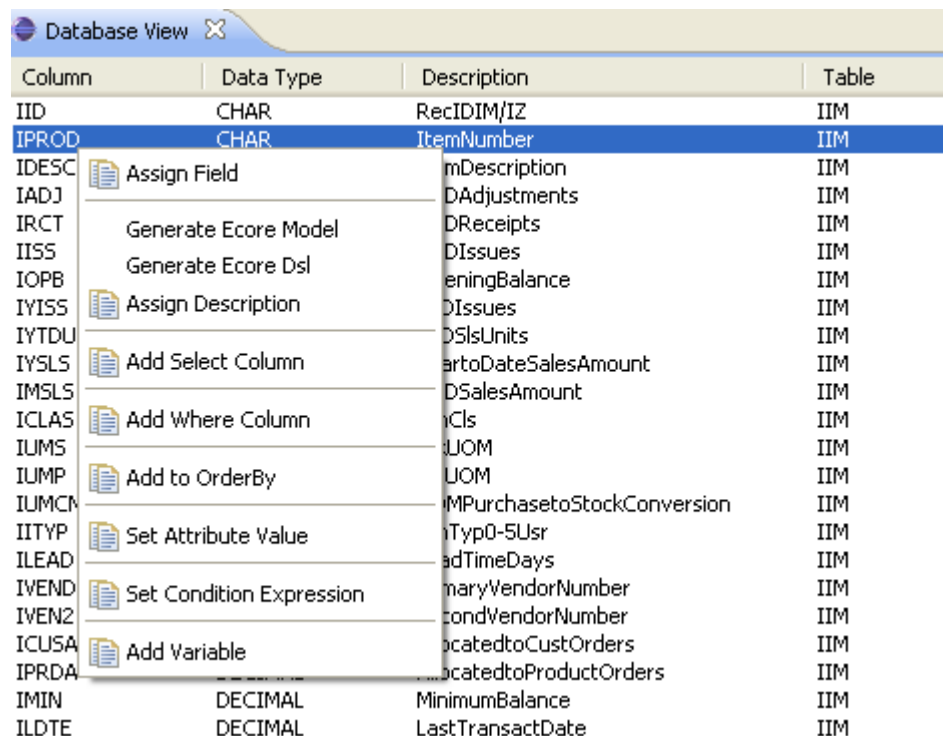
4   When the API process instruction is generated, it produces a PCML file and an XML file. Repeat Steps 2-3 to add the XML file to the same jar file but do not serialize the XML file. The jar file must include all files that are generated from the PCML project.

5   To test the API, copy the new jar file the appropriate directory: to the LX Extension installation directory or to the LxConnector IFS folder.

Database

Use this view to map fields, elements, and table names to create an outbound message. Add Mapping instructions to the tree and assign fields, element names, and table names to the Property Page for each instruction.

To access the Database view, use the Retrieve Screen Fields view to retrieve data. The Database view is opened automatically after the data is retrieved. This view enables you to map database columns to Mapping nodes, If Condition nodes, and Statement nodes in the tree.

In the Database View, right-click a row to display the Context menu. The Context menu includes these options:

Option	Description
Assign Field	Add the selected Column to the Database Field of the selected Mapping node Properties page. This option also sets the Table Name property in the Properties view for the selected Mapping node.
Assign Description	Add the selected Column to the Database Field Property and the Description field to the Element property of the selected Mapping node Properties page. This option also sets the Table Name property in the Properties view for the selected Mapping node.

Option	Description
Add Select Column	The SQL Builder view must be open to use this option. Select a Statement node to open the SQL Builder View. The Add Select Column option inserts the selected column into the Select box of the SQL Builder View and adds Table.Column into the selected statement.
Add Where Column	The SQL Builder view must be open to use this option. Select a Statement node to open the SQL Builder View. The Add Where Column inserts the selected column into the Where box of the SQL Builder View and adds Table.Column into the Where statement.
Add to Order By	The SQL Builder view must be open to use this option. Select a Statement node to open the SQL Builder View. The Add to Order By option inserts the selected column into the Order By box of the SQL Builder View and adds Table.Column into the Order By statement.
Set Attribute Value	This option sets the selected column to the Database Field property of the selected Attribute node.
Set Condition Expression	The Expression Builder View must be open to use this option. Select an If Condition node to open the Expression Builder View. This option adds the selected column into the Expression field in the Expression Builder View.
Add Variable	This option is not supported at this time.

Edit Comment

Use this view to add Comments into the tree. The comments are informational only and do not get generated in the process instruction. You can add a comment to each major node in the tree.

1   Select **Window > Show View > Other > Infor ERP LX Views > Edit Comment**.

2   Specify a comment in the edit box.

3   Click **OK** to update the Comment property of the selected Comment node.

ESB Message

Use this view to test messages by sending a message to the inbox or outbox.

1   Select **Window > Show View > Other > Infor ERP LX Views > ESB Message View**.

2   Specify this information:

File Name

Browse to select the BOD to be published.

Host

Specify the host where the inbox or outbox is stored.

Files Library

Specify the files library on the host server in which the inbox or outbox is stored.

Database User

Specify a user ID that is authorized to the host server.

Database Password

Specify the password for the user ID.

Select In/Outbox

Select the inbox or outbox to test:

- IONInbox
- IONOutbox
- LXCInbox

Expression Builder

You can use the Expression Builder view to add conditional logic into the process instruction, to allow decisions to be made at runtime. Assign expressions to If Condition instructions. The Expression Builder allows you to assign Xpath values from the Xpath View or database fields from the Database View as variables in an expression.

Note: To use the Xpath View you must have a BOD Template. BOD Templates are available for ION integrations.

Before you add an expression, review these rules:

- All expressions must be contained between parentheses ().
- An expression must not include a blank space before or after an operation.
- The AND and OR operators cannot be preceded or followed by a blank space.
- Each expression must be enclosed in parentheses. For example, a valid expression is `((x==y)||(a!=b))`.
- There must be an equal number of open and closing parentheses.
- Do not use quotation marks around constants. For example, `((x==BLANK)&&(y==2))`.

1   To open this view, select **Window > Show View > Other > Infor ERP LX Views > Expression Builder**.

Expression Builder

Select Condition

Select Operation

==	(EQUAL)
!=	(NOT EQUAL)
>	(GREATER THAN)
>=	(GREATER OR EQUAL)
<	(LESS THAN)
<=	(LESS OR EQUAL)
&&	(AND)
||	(OR)
*BLANK	(IsBlank)
!	(Not)
(	(Open Parenthesis)
)	(Close Parenthesis
+	(Addition)
-	(Subtraction)
/	(Division)
*	(Multiplication

Expression:

Validation Result:

OK   Cancel

Clear   Validate
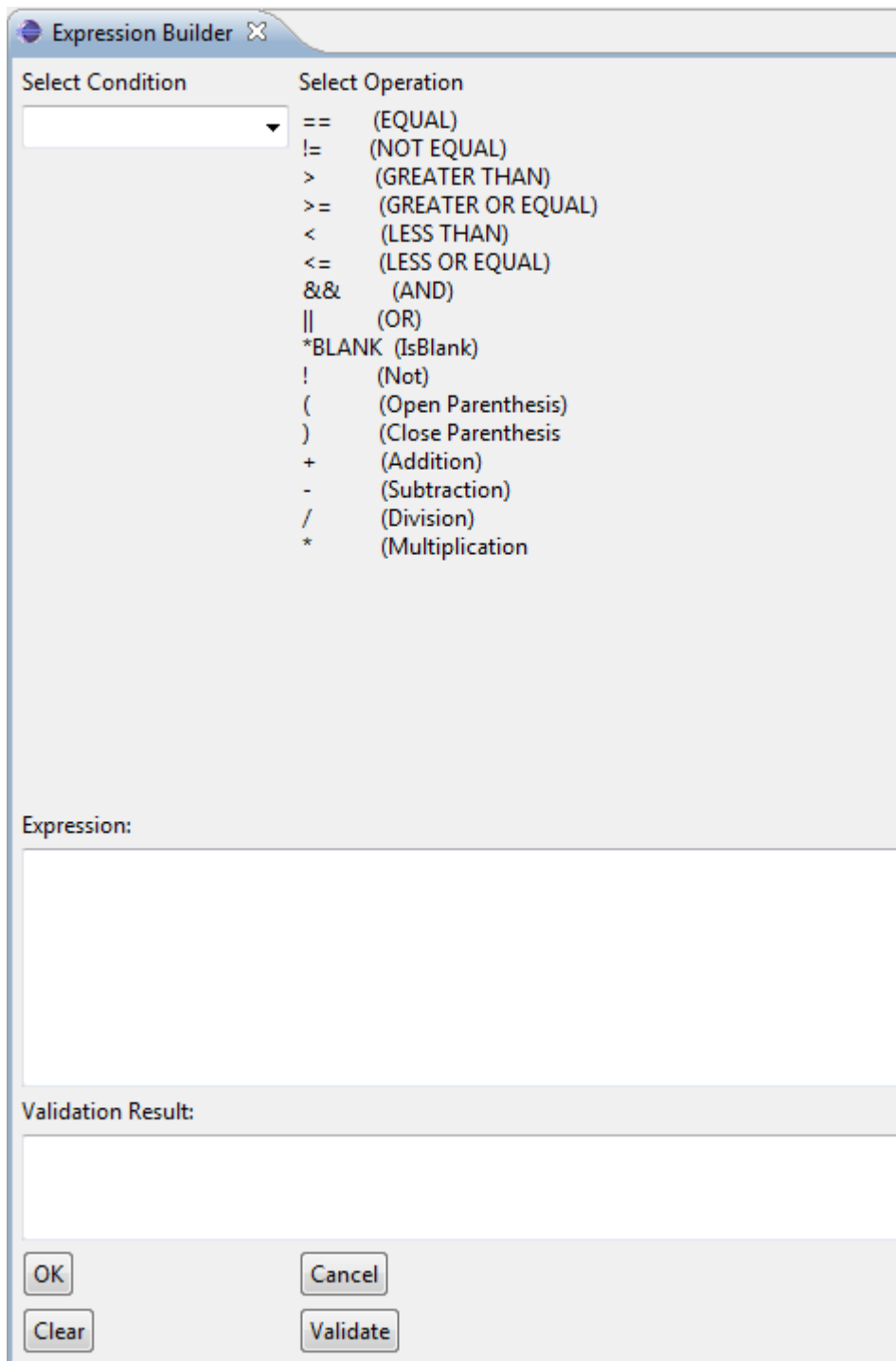
2   Specify this information:

Select Condition

Select an option:

- if

- elseif
- else
- Arithmetic Expression
- endif
- condif
- while

Select Operation

To build the expression, select an operation. Use the up and down arrows to move through the list.

3   To add columns, select a column from the Database View. To select an element, select an Xpath from the Xpath view.

4   Edit the expression.

5   Click **Validate** and review the validation results

Retrieve Screen Fields

Use the **Retrieve Screen Fields** view to extract data from the LX database. This view retrieves metadata that can be used to create inbound process instructions and data that can be used to create outbound process instructions. You can select a BOD Instance but BOD templates are only available for ION integrations.

When you click **OK**, relevant data is retrieved from the specified Host. If you specified display file names then a skeleton set of data creates an inbound process instruction in the designer view. If you entered Table data, the Database View opens with the metadata displayed. If you selected a BOD Template name, the Xpath View opens displaying the BOD data.

You can also use this view to retrieve data to use in an exit point trigger.

1   Select **Window > Show View > Other > Infor ERP LX Views > Retrieve Screen Fields**.

2 Specify this information:

Host

The name of the System i machine from which to extract data. For example, `MySystemi`.

Outfile

The name of the library where temporary metadata files are written; these files are used to build a skeleton inbound message. For example, `TEMPLIB`.

> **Note:** Library should not be in any LX environment *LIBL.

Library

The name of the files library. For example, `erplxf`.

Display File Names

The list is created when you use the display file field description command.

Table

A comma separated list of LX files. For example, `HPH,HPO,IIM,HPC`.

BOD Template Name

Use Browse to navigate to a directory containing an instance of a BOD. For example, `SyncRequisition.xml`.

Inbound/Outbound Attribute

Options:

- All
- Inbound only
- ExitPointfrom Spreadsheet

If you select Inbound only, then, when you retrieve screens fields, only inbound fields are added to the tree. Otherwise the Builder retrieves both inbound and outbound data.

User

A valid LX user ID to sign on to the System i.. For example, `User ID`.

Password

The password for the user ID. Note the password is masked. For example, Users password.

Connection Info

Error messages that indicate problems during retrieval of data. If there are no errors the box remains empty.

3   Click **OK** to retrieve data. To use this view to build an exit point trigger, see "Using SetExitPointData".

After you specify data retrieval information in the fields, click **OK** to populate the Database View and Xpath View with the requested data. To open the Xpath View, click **OK**, and select a template. For example, click **Browse** and select the `SyncRequisition` template. Specify an ERP LX Table name in the Table, such as `HPH`. Click **OK** to open the Database View. The Xpath and Database Views are shown below.

Using SetExitPointData

Use this option to build an exit point process instruction that is a trigger. First create a skeleton exit point project as shown in this screen:
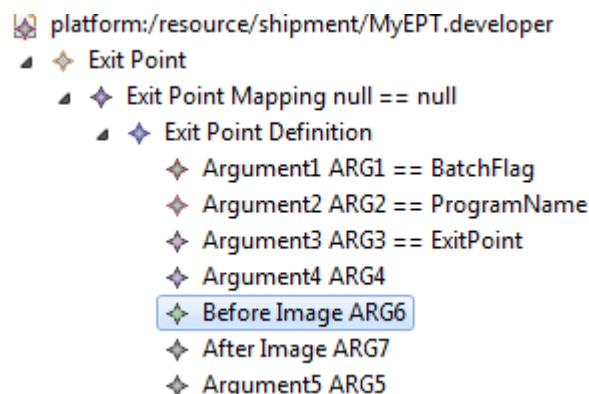
In the Retrieve Screen Fields view, enter the name of the files library and the name of the table that the exit point trigger represents. Select the **BeforeImage, Arg 6** in the trigger exit point that you are building, and then click **SetExitPointData**. Exit Point Data is created for each field in the table as shown in the screen below.
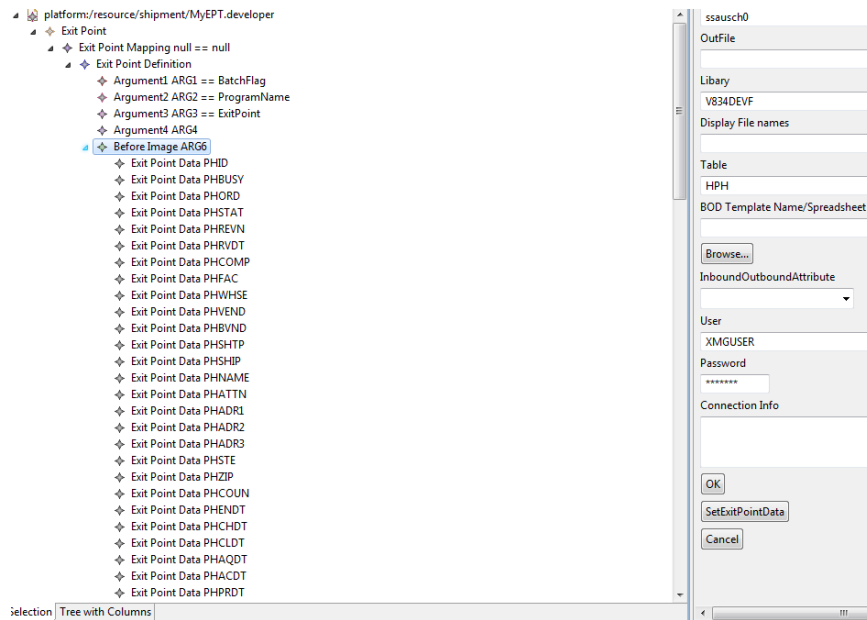
Search Tree

Use the Search Tree View to map variables defined in the exit point or trigger process instruction to an outbound process instruction. For example, an If Condition can check the Program Name. This view retrieves the Names that are defined in exit point and database trigger process instructions.

1   Select **Window > Show View > Other> Infor ERP LX Views > Search Tree**.
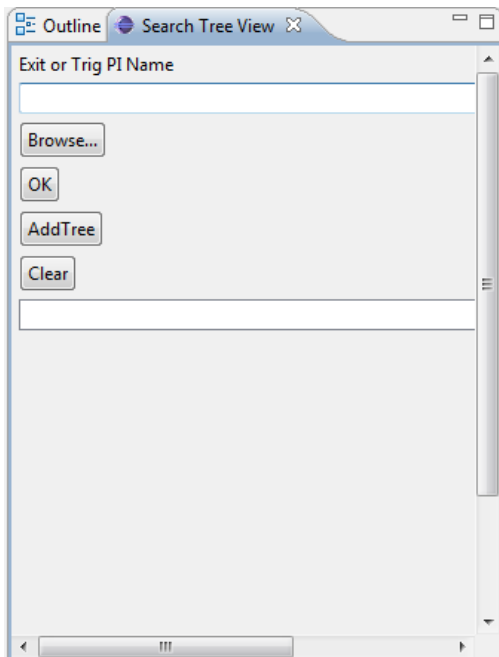
2   Click **Browse** to open a generated exit point or database trigger process instruction.

3   Click **OK**.

4   Click **AddTree** to populate a list box with the variables that were defined in the exit point or trigger process instruction.
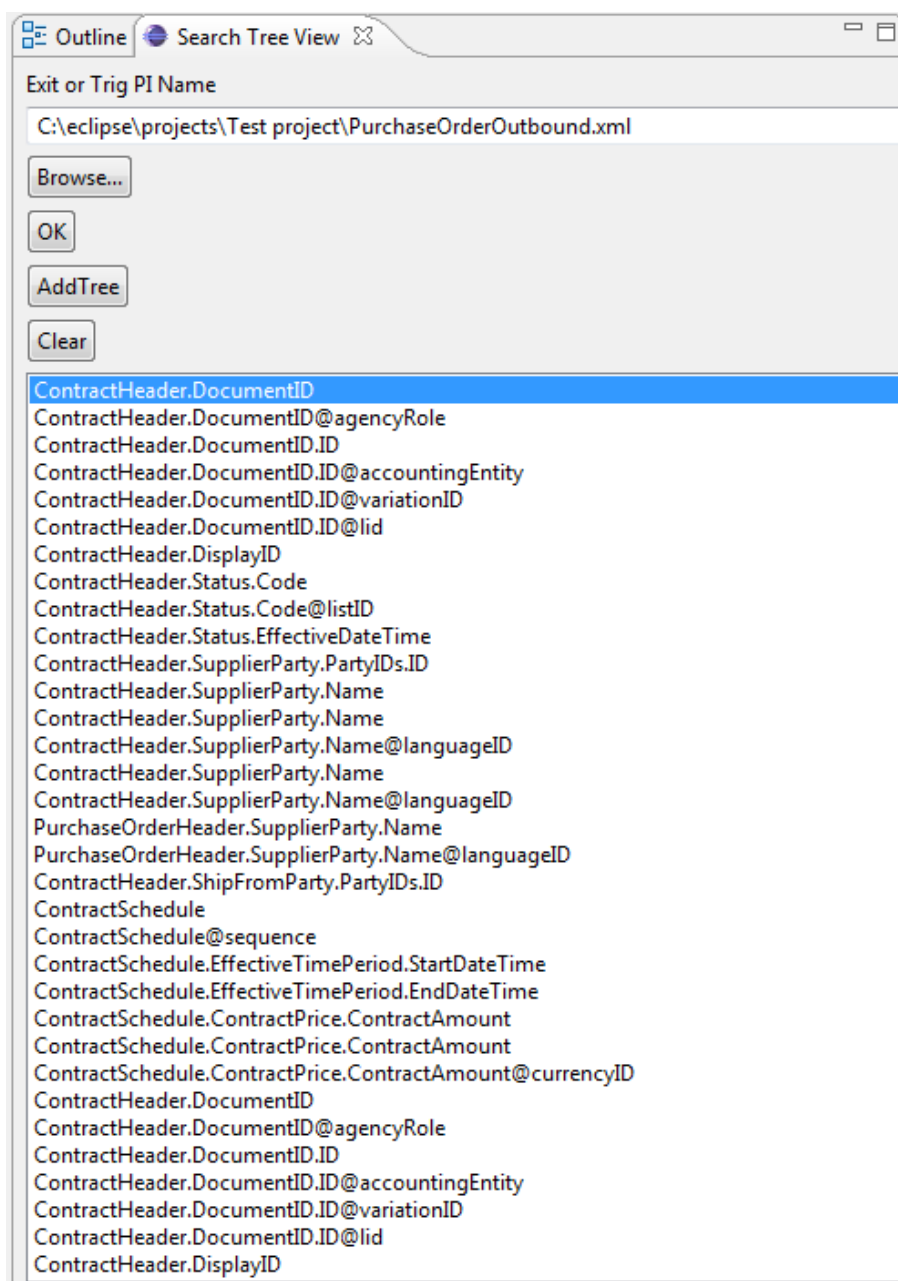
5   To add a variable to the property page:

    a   Select the property in the outbound instruction to which you want to add the variable.

    b   In the list box, double-click the variable to add it to the property page.

Search Xpath

Use the Search Xpath View to retrieve a subset of data from the Xpath View currently displayed. For example, search the Xpath View for all elements containing DocumentReference. You can map to the Element field on an inbound or outbound message.

Note: To use the Xpath View you must have a BOD Template. BOD Templates are available for ION integrations.

1   Select **Window>Show View>Other>Infor ERP LX Views>Search Xpath View**.
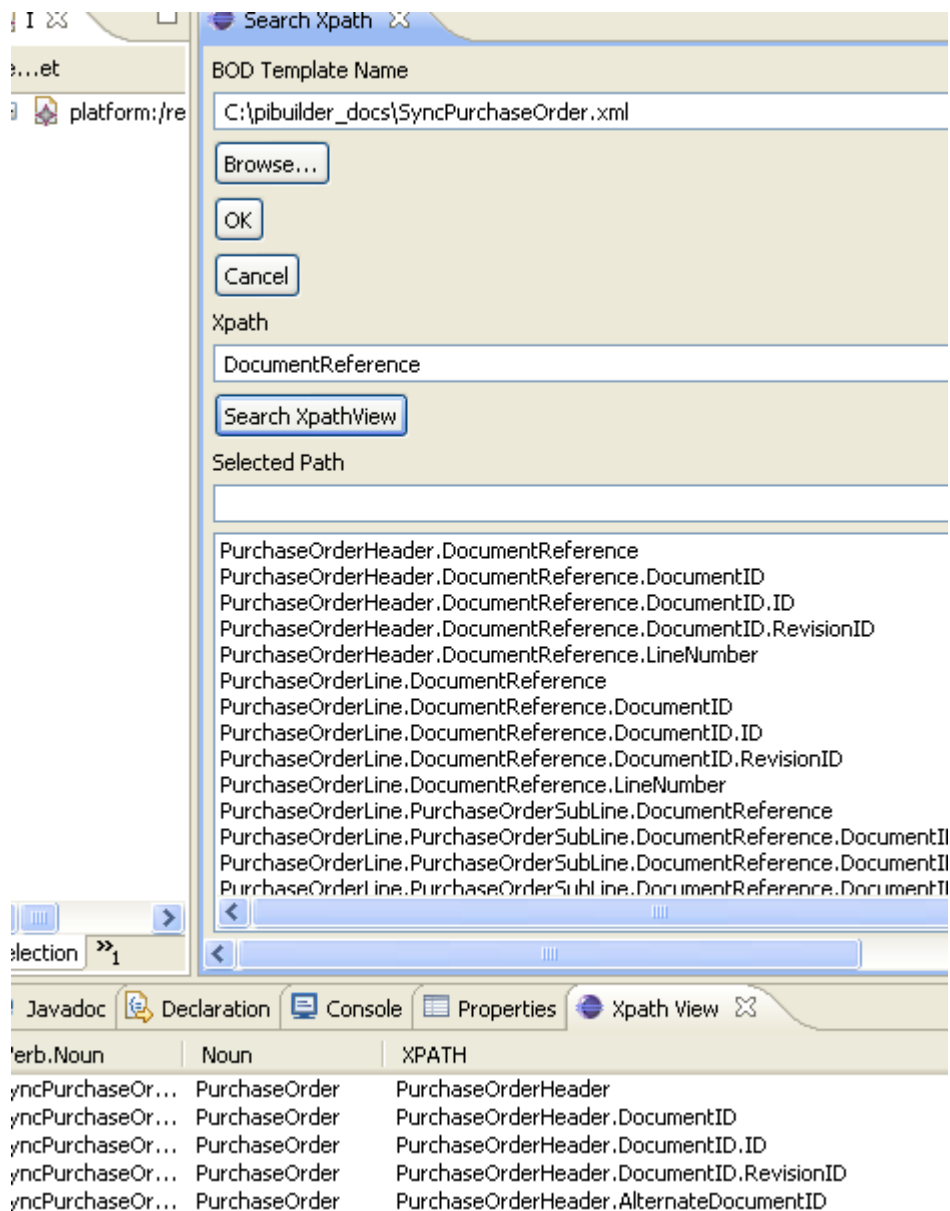
2   Click **Browse** and select a BOD template.

3   Click **OK**.

4 Enter an xpath, such as `DocumentID`, and click **Search XpathView** to retrieve a list of xpath elements that contain the word DocumentID.

5 To add an xpath to a selected Mapping node, double click an item from the list box. This sets the Element in the property page for the Mapping node.

SQL Builder

Use the SQL Builder view to add SQL statements into the process instruction. You can add Xpath variables and database fields to an SQL statement. The SQL builder contains edit boxes for the SELECT, WHERE and ORDER BY parts of an SQL statement.

1 Select **Window > Show View > Other > Infor ERP LX Views > SQL Builder**.

Select Functions	Select Test	OrderBy	Where-Functions	Where-Test
AVG	(	(	WHERE	(
COUNT	)	)	SUBSTR	)
MAX	<	,	TIME	<
MIN	<=	ASC	TIMESTAMP	<=
SUM	<>	DESC	YEAR	<>
||	=	AVG	SELECT	=
CASE	>	COUNT	WHERE	>
WHEN	>=	MAX	||	>=
AND	BETWEEN	MIN	END	BETWEEN
THEN	IN	SUM	NOT	IN
END	IS NOT NULL		AND	IS NOT NULL
SELECT	IS NULL		OR	IS NULL
UPDATE	LIKE			LIKE
FROM	AS			
SET				
DIGITS				

Select:

Where:

OrderBy:

Host

Files Library

User

Password

SQL Result

OK   Clear   Validate   Cancel

2 Click the appropriate values for these options:

- Select Functions
- Select Test
- OrderBy
- Where Functions
- Where Test

3 Edit the entries in the **Select**, **Where**, and **OrderBy** fields.

4 Specify this information:

Host

Specify the host where the inbox or outbox is stored.

Files Library

Specify the files library on the host server in which the inbox or outbox is stored.

Database User

Specify a user ID that is authorized to the host server.

Database Password

Specify the password for the user ID.

5 Click **Validate**.

6 If the SQL statement is valid, click **OK**. The SQL statement updates the Statement property of the selected Statement.

Variable Definition

This view is not currently supported.

Xpath View

Use the Xpath View to map element names to inbound and outbound process instructions for integrations that use ION communications. The Xpath View displays a BOD Instance as a flat file of Xpath values. The view has a Context menu that you use to map elements to fields and to create variables within the Expression Builder and SQL Builder.

Note: To use the Xpath View you must have a BOD Template. BOD Templates are available for ION integrations.

1 Open this view in the Retrieve Screen Fields View. Select a BOD Template Name.

2 Click **OK**. The view that opens contains the BOD instance.
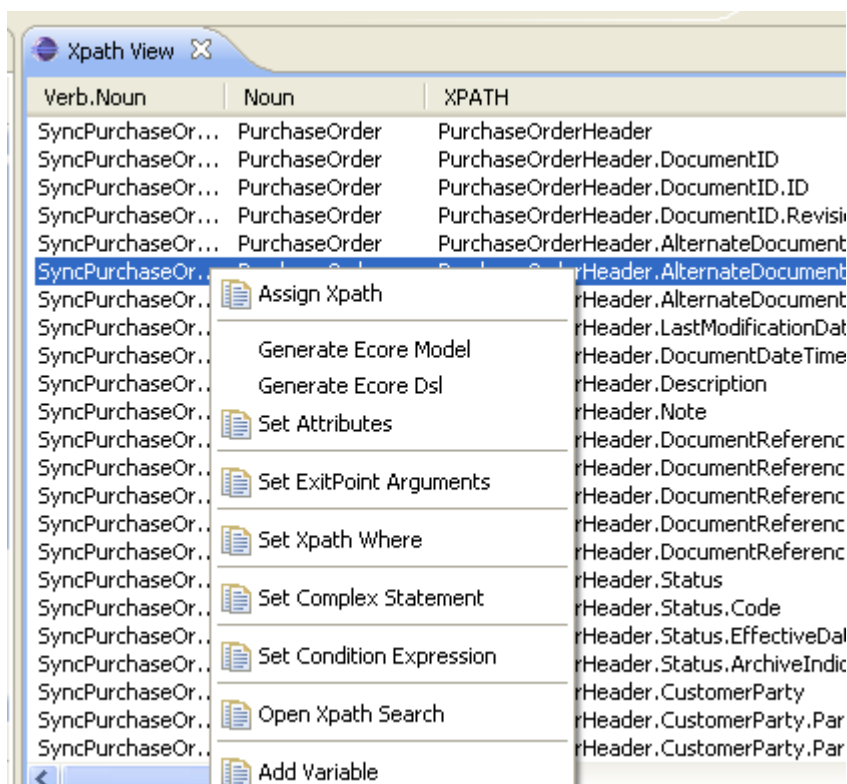
3   Use the context menu to select an Xpath from the XPATH column. To open the context menu, select a row and right click. The context menu has these options:

- **Assign Xpath**: assign the Xpath value to the Element property in a Mapping node.
- **Set Attributes**: add the Attributes in the Attributes column to a selected Mapping node.
- **Set Exit Point Arguments**: to add Argument child nodes to the Exit Point Definition Node, select both this option and the Exit Point Definition node in an Exit Point project.
- **Set Xpath Where**: set the Xpath into the where clause displayed in the SQL Builder View.
- **Set Complex Statement**: set the Xpath into the select clause in the SQL Builder View.
- **Set Condition Expression**: add the Xpath column to the Expression in the Expression Builder view.
- **Open Xpath Search**: open the Xpath Search View.
- **Add Variable**: this option is not currently supported.

Chapter 2   Node descriptions

This chapter describes the nodes that are added to the Model Object tree view. All nodes have a property page that provides an interface to enter properties used to define the node. The property information is used to add instructions into a process instruction when the Model Object developer project is generated.

Nodes of the tree

To build a tree of instructions, add nodes to the tree. See Chapters 3 and 4 for instructions to build the tree.

The process instructions are used to process a specific Business Object Document (BOD). For example, you may need to integrate item data from a third-party application into EPR LX. To integrate the data, use the tool to build a Model Object and generate a process instruction that can navigate through the LX item application. If you must produce an Item from an LX event that is used by a third-party application, use the tool to create process instructions that use an LX event to build Business Documents.

- Chapter 3 describes how to create an Inbound Model Object that produces a process instruction. Process instructions are used by the LX Extension and the LX Connector to process BOD data into LX.
- Chapter 4 describes how to create an Outbound Model Object that produces a process instruction. The process instruction is used by the LX Extension and the LX Connector to produce a BOD message.
- Chapter 5 contains several examples of how to, when, and which node to use when building a Model Object.

This chapter describes each node that can be added to a Model Object project and the properties that are used.

Action

Use the Action node to create an inbound Model Object that requires navigation through an LX legacy application. The property page for the node allows you to add information about an LX

screen, for example the program Name, the panel name of the screen and sequence of the flow through the application. This table shows properties for the node:

Property	Description
Action Include	Do not use in ION integration projects. This property is used by LX Connector process instructions. The Action Include value is the name of another process instruction to call. The called process instruction must be in the program flow to work. For example the LX Connector PurchaseOrder PI has an Action Include instruction that is ADPPUR01_POLine. In this case header mapping is contained in the PurchaseOrder process instructions and PurchaseOrderLine mapping is contained in the Action Include process instruction.
Allow Repeat	**Note:** Do not use in ION integration projects. This is used by some Version 1 LX Connector Process Instructions, but is not used by Connector version 2 process instructions. The default is set to false. Set this to true to loop through several lines if the same screen is used for data processing. **Caution:** This property may cause issues with screen looping.
Description	A short description of the Action. The description is not added into the generated process instruction.
Error Exit Return	Select the value from a drop down that lists LX function keys **F1** through **F24**, **Enter**, and **End**. Select the value that allows you to exit a screen if an error occurs that is not an override. Selecting **End** is not advised but is used by LX Connector process instructions to cache at a screen. Typically, this is used when multiple documents write to the spool file, such as INV500, to allow the reports to produce in a single file.
Panel Loop Begin Action	The default value is set to `0`. This property is used to allow looping over a sequence of LX screens when processing an inbound message that has subfile data. The value for the property is set to the Action that starts the loop. For example, when processing lines in a PurchaseOrder BOD, each line is added using a set of sequenced screens. The Model Object view contains several Action nodes that navigate in sequential order through the LX application. To add a line to LX requires looping through Action 5, Action 6, Action 7, Action 8, and Action 9 for each line. On Action 9 the Panel Loop Begin Action property is set to `5`; if there are more lines to process go back to Action 5.
Panel Name	The name of the panel, such as `PANEL01`. In some cases, to get the correct Panel Name, you must run the LX Extension or LX Connector with logging turned on. The Log contains the Panel Name to use.

Property	Description
Program Name	The name of the LX application. Generally, this displays in the upper left corner of the green screen, such as INV500D1.
Program Name Alias	Do not use in ION integration projects.
	Use this node with LX Connector projects to allow mapping of one Program Name to another. For example, if Webtop returns PUR500 when executing the first action but you need to add `PUR500D1` as the screen name, use an alias to map PUR500 to PUR500D1. Adding an alias creates an Alias Instruction in the PI. Most LX Connector process instructions do not require an alias.
	`<Alias><ProgramName Alias="PUR500">PUR500D1</ProgramName></Alias>`
Return	The function key that gets pushed to move to the next screen. Select from the list of Functions in the drop down list. If **End** is selected the LX Extension or LX Connector keeps this screen cached.
Sequence	Actions must be in the same sequence as the flow of the LX application screens. The first screen in the flow must have this value set to `1`. Each additional action should be increased by 1. For example, if you have four screens the sequence is 1, 2, 3, 4

Action Code

The Action Code node is used when building an inbound Model Object. It is the child of a Display Program node that is used to create an instruction that navigates through LX application screens. The node contains properties that set the method of the instruction. For example, to support an Add request from a BOD message, set the property Action Code Type to `Add`.

This table shows the Action Code properties:

Property	Description
Action Code Type	Select the Action Code Type. The Choices are `Create`, `Add`, `Replace`, `Change`, `Delete`, and `None`. This is the method requested by a BOD message.
Description	Short description of the action code. The description is not added into the generated process instruction.

Acknowledge

Add the Acknowledge node to the tree when you build a Model Object that navigates LX application screens. An Acknowledge node is a child node to an Action node.

Note: Do not use this node in ION integration projects. These integrations use an Acknowledge process instruction for building an Acknowledge BOD message.

An Acknowledge node is used to define the noun identifier. A noun identifier is the element in a BOD message that makes the message unique. For example, the noun identifier for an Item is ItemCode. Setting the Xpath property to ItemCode causes the `<ItemCode>` to be included in messages returned by LX to a client application. The Action Code node is a container of one or more Action nodes. Acknowledge is generally added as a child to the first Action in the Action Code container.

This table shows the properties for the Acknowledge node:

Property	Description
BOD Xpath Type	The BOD Xpath Type is not currently used. It is set to `NONE`
Xpath	This is the name of the Element that is returned to a sender application.
	For example if Xpath is set to ItemCode after execution of the application has completed a message is returned to the sender application. If the transaction was successful, the message contains this information:
	`<Envelope><ItemCode>MYITEM</ItemCode></Envelope>`

After Image

Add an After Image node to an Exit Point Model Tree view when an LX trigger program executes an LX event. The node requires that the entire data structure be mapped in the order of the trigger data structure. This node is a container of Exit Point Data nodes that provide the actual mapping capability.

The properties for the After Image node are shown in the table below. The Name must be ARG7.

Property	Description
Description	Short description about the mapping.
Name	ARG7

API Field Mapping

Use the API Field Mapping in a Model View project to map either fields from a database or elements in a BOD message to an LX API. The use of the node requires that a PCML Model Object project has been defined and generated to produce process instructions used at runtime. The PCML Model Object developer project maps fields from an RPG data structure to Elements that can be used in the API Field Mapping.

This table shows the properties available for API Field Mapping:

Property	Description
API Field	The API Field is the value given to the Name property of an Exit Point Data node defined in the PCML Model Object project. This is a parameter passed to the API program that is executed.
Description	The description of the field. The process instruction does not use the description.
Variable	The variable is the Value assigned to the API Field. Since the API Field is a parameter, this is the value assigned to the parameter. The variable can be a constant, a value from an element in the inbound message, or a value retrieved using an SQL statement.
Variable Type	See a description of the variable types in "Variable Type options" in this chapter.

API Instruction

Add the API Instruction node to a Model Object tree view when parameters in a Batch Program node must be updated before execution.

This table shows the properties for the API Instruction:

Property	Description
Description	This is a short description about the instruction. This is not generated into the process instruction.
Name	This is the name given to the API Instruction. This must be the name given to another Instruction in the Model Object project that defines a Batch Program.

Argument nodes

Add the Argument nodes to an Exit Point Model Object tree view when mapping Arguments for an exit point process instruction. Exit Point process instructions accept five arguments that include raw data. Each argument is mapped to a name that can be used later in an Outbound Model Object tree view.

Argument 1

This node is the first parameter passed by the data structure. Do not modify this node. This table shows all values for the properties:

Property	Description	Value
Data Name	The name available to an Outbound Model Tree view.	**BatchFlag**
Data Type	The LX data type	**char**
Data Usage	PCML usage used to invoke IBM PCML call.	**inherit**
Description	A short description of this argument	
Name	The name given to the argument	**ARG1**

Argument 2

This node is the second parameter passed by the data structure. Do not modify this node.

This table shows all values for the properties:

Property	Description	Value
Data Name	The name available to an Outbound Model Tree view.	**ProgramName**
Data Type	The LX data type	**char**
Data Usage	PCML usage used to invoke IBM PCML call.	**inherit**
Description	A short description of this argument	
Length	The length of the value	**10**

Property	Description	Value
Name	The name given to the argument	**ARG2**

Argument 3

This node is the third parameter passed by the data structure. Do not modify this node

This table shows all values for the properties:

Property	Description	Value
Data Name	The name available to an Outbound Model Tree view.	**ExitPoint**
Data Type	The LX data type	**char**
Data Usage	PCML usage used to invoke IBM PCML call.	**inherit**
Description	A short description of this argument	
Name	The name given to the argument	**ARG3**

Argument 4

Argument 4 must be mapped completely to a 256 byte data structure. This node requires Exit Point Data child nodes to map the raw data to names. Do not modify the Name property, ARG4.

This table shows all values for the properties:

Property	Description
Description	A short description of the argument
Name	ARG4

Argument 5

Argument 5 must be mapped completely to a 256 byte data structure. This node requires Exit Point Data child nodes to map the raw data to names. Do not modify the Name property, ARG5.

This table shows all values for the properties:

Property	Description
Description	A short description of the argument
Name	ARG5

Attribute

Use attribute nodes in inbound and outbound Model Objects. If you are building an outbound Model Object, the addition of an Attribute node allows you to add one or more attributes to an Element. If you are building an inbound Model Object, use an attribute node to map the value of an Elements attribute in a BOD message to a field in the LX legacy application.

This table shows the properties of the Attribute node:

Property	Description
Cross Reference	This property is supported only for ION integration projects. All other projects should use the default value. See "Cross Reference options" in this chapter for a list of options.
Database Field	The database field is a column in an LX file that is retrieved by Statement nodes. If the Database Field is specified, its value is given to the element attribute. For example, you want to add an attribute named currency to the current element. The value for this attribute is set using Database Field `HPH.PHCUR`. The value for `HPH.PHCUR` is retrieved by an SQL statement.
Date Field	Not used
Date Format	Not used
Date Separator	Not used
Date Time	Not used
Description	The description explains the attribute but is not generated into the process instruction.
Is Calculated Attribute	Not used
Is Time Stamp	Not used

Property	Description
Name	The Name given to the attribute. The Name cannot contain any spaces or XML special characters. For example, to add an attribute to element Status, add an Attribute node and set the Name to listID. This produces `<Status listID=""/>`
Qualifier Element Name	Not used
Region Type	Not used
Time Field	Not used
Time Format	Not used
Time Separator	Not used
Value	Set the Value property if the value for the attribute is a constant. For example, an attribute is added to element Code. The Value is set to Requisition Status and the Name is set to status. This produces `<Code status="Requisition Status">`

Batch Program

Add the Batch Program node if an LX API is required. Both inbound and outbound projects may require a Batch Program node.

For example, the value for an element is set using a parameter returned by executing an API program. This node is used to map either elements from a BOD message or data for an SQL result set to fields in the API. The Batch Program node requires a PCML Model Object that was created, generated, and placed in a process instruction jar file.

This table shows the properties for the Batch Program property page:

Property	Description
Action	Select one of these actions that is performed by the API: • None (Default) • Add • Replace • Create • Change • Delete
Description	A short description that explains the API. The description does not get generated into the PI.
Name	This property is the name of the LX API that is invoked, for example, SYS934B.
Struct Name	Not supported.

Before Image

Add the Before Image node to an Exit Point Model Tree view when an LX trigger program executes an LX event. This is the eighth argument passed by the LX event. The node requires that the entire data structure be mapped. If you have mapped the After Image you can use a single Exit Point Data node that sets the Name property to FILLER and define the structure to be blanks.

This table shows the properties for the Before Image property page:

Property	Description
Description	A short description of the mapping
Name	ARG6

BOD Element

A BOD element is added to an Exit Point Model Object tree view to define which process instruction is loaded by the event. This process instruction is used to build the Outbound BOD message.

This table shows the properties of the BOD Element property page:

Property	Description
Description	A short description of the node's function
Name	This is the name of an instruction in the project for the Outbound Model Object that is loaded when this LX event occurs. For example, if the Name is `IsCompReturn` and the Process Instruction Name is `ProductionReceiverOutbound` then the generated ProductionReceiverOutbound process instruction must contain a Condition that has its Name property set to `IsCompReturn`.
Process Instruction Name	This is the name of the process instruction used to build the BOD message. The process instruction Name must be Noun appended with Outbound

BOD Version

Add the BOD Version node to add version information into an Outbound BOD message. When the BOD message is produced, the information in the property page is added as attributes of the root element.

The BOD Version node is required to build outbound projects that use the LX Extension and ION connectivity.

This table shows the properties of the BOD Version node:

Property	Description
Bod Version ID	The version of the Infor business document. The BOD Version ID is added into the outbound message as the versionID attribute of the root element.
Description	Short description of the node's function. Description is not included in the generated process instruction.
Document Root Prefix	Not supported,
Release ID	The Release ID is the version of the OAGIS BOD. The BOD Release ID is added into the outbound message as the releaseID attribute of the root element.

Property	Description
Version ID	This is the html version that is written into the processing instruction of the generated document. For example: <? xml version="1.0" encoding="utf-8" ?>

Comment

Add a Comment node to the Model Tree to provide detailed explanations about an instruction that you added to the Mode Object. Only comment nodes included in the Narrative node are written to a generated process instruction. You can add Comments to both inbound and outbound Model Object trees.

This table shows the Comment node properties:

Property	Description
Comment	The comment is a description that is used to enter information visible only in the tree view. Comments do not appear in the generated process instruction.
Print Comment	Not currently supported

Concatenation Field

Add the Concatenation Field node when the value assigned to an element requires the value to be a concatenation of data.

For example, when mapping an outbound element, the value assigned to the element requires concatenation of data that was retrieved using SQL Statements.

This table shows the Concatenation Field properties:

Property	Description
Add Leading Zeroes	Select **True** or **False**. Set this property to **True** if the value assigned to the Element must be a specific length. When the property is set to **True**, leading zeroes are inserted into the concatenated value if the length of the concatenated value is less than that set in the property Number of Characters.
Description	A short description of the concatenation. The description is not generated in the process instruction.
Field	If the Variable Type is database this is the database field that contains the value to be concatenated.
Identifier	Not used
Number of Characters	The maximum length of the concatenated value.
Pad With Blanks	Select **True** or **False**. Set this property to **True** if the value assigned to the Element must be a specific length. When the property is set to **True**, blank characters are appended to the concatenated value if the length of the concatenated value is less than that set in the property Number of Characters.
Variable Type	See section "Variable Type options" in this chapter for a list of options.
Xpath	Set this property if the Variable Type is inbound and concatenation requires a value from a BOD message. This is the xpath to the element in the inbound message.

Condition

Add the Condition node as a container of other instructions. All Model Object trees require at most one Condition node. This is the instruction used when the generated process instruction is loaded by the LX Extension or the LX Connector at runtime. A Condition node may contain several child nodes. This node can also be used as a looping condition used when processing an Inbound BOD Message.

This table shows the properties of the Condition node:

Node descriptions

Property	Description
Description	A short description of the condition. The description is not written into the generated process instruction.
Exit Instruction Name	This property is used when the Is Inbound Loop property is set to **True**. This is the Name of an Instruction node in the Model Object this is executed when the loop completes.
Is Acknowledge Instruction	Not currently supported.
Is Inbound Loop	Select true from the drop down if you need to loop through the Inbound message.
Name	This is the name given to the Condition node. A name allows this Condition to be called using an Instruction Name node.
Type	This is a constant set to Condition.

Conditional Instruction

Add a Conditional Instruction node when instructions need to be separated so that the process instruction that is generated executes the instructions in the required order. The node can be configured to allow looping through an inbound BOD message when creating an Inbound Model Object. The node can be used in both inbound and outbound Model Object trees.

This table shows the properties of the Conditional Instruction node:

Property	Description
Conditional Type	Define how the conditional Instruction is processed. These types are available: • Simple: Default. Most Conditional Instructions are Simple. • Inbound: Select this type to loop through child elements in a BOD message. If you select Inbound, you must enter the Element Name. The Element Name is the name of the element in the BOD message used for looping. The Conditional Instruction must be a child of an Instruction that has the Is Inbound Loop property set to true. • Sql: Not supported. • Inboundsql: Not supported.

Property	Description
Description	Short description of the instruction's function. This description is not added into the generated PI.
Element Name	Set this property if the Conditional Type is set to `Inbound`. The Name is the name of an element in the BOD that is used for looping. For example, if the Element Name is set to `ShipmentItem`, each ShipmentItem found in the message is processed separately.

Confirm Error Message

Use this node for LX Extension integrations that use ION.

The Confirm Error Message node is a child node of an Outbound Message Instruction. Use this node to process an inbound message that requires an error be produced based on information in the BOD message. The node allows definition of an LX message ID. Using the message ID, program SYS014C is called to retrieve the first level message text. The error causes a ConfirmBOD to be produced.

This table shows the properties of the Confirm Error Message property page:

Property	Description
Description	A short description of the instruction.
Message Id	The LX message ID to set in the ConfirmBOD.
Message Text	If this field is blank and the Message ID is a valid LX message ID, the message is extracted from SYS014C and added into the ConfirmBOD that is created.

Copyright

Add the copyright node to the Model Object tree to add copyright information into the generated process instruction. The node can be used by both inbound and outbound Model Object trees.

This node is required for ION integrations.

This table shows the properties of the Copyright node:

Property	Description
Copyright statement	Required for ION integrations

Data Area Field

Add the Data Area Field node to retrieve and map data from an LX Data Area object to an element in a BOD message or to an element in an outbound Mapping node.

This table shows the properties of the Data Area Field node:

Property	Description
Description	A short description of the instruction. This is not added into the generated process instruction.
Name	Variable that holds the value retrieved from the data area.
Number of Characters	Number of bytes to extract from the Data Area.
Precision	The precision of the value being extracted. If this is character data set this to zero.
Start Position	Start position when extracting data from the Data Area.
Type	Select the type from the drop down. These types are available: • char • packed • struct
Value	Value is filled at runtime and contains the variable name defined in the Name property

Data Area Instruction

Add the Data Area Instruction when data is required from an LX Data Area. The node allows entry of the name of the required Data Area.

This table shows the properties of the Data Area Instruction node:

Property	Description
Data Area Name	This is the name of the LX data area, for example `SSASYS`.

Database

Add the Database node to the tree view to provide the Mapping and SQL Statements that are used to build a BOD message. The parent node is the Instruction. The Name assigned to the Database node must be the same Name given to the parent Instruction node.

The node may contain two child nodes: Mapping Details and Database SQL Statements. Add the node to an outbound project.

This table shows the properties of the Database node:

Property	Description
Description	A short description that is not added to the process instruction
Locate Row Xpath Name	Not supported
Name	Set this property to the value of the Name assigned to the parent Instruction node.
Type	The type is a constant of SQL.

Database SQL Statements

Add the Database SQL Statements node as a child of the Database node. This node contains Statements that are used to build or process BOD messages.

This table shows the properties of the Database SQL Statements node:

Property	Description
Description	A short description of the node

Derive

Add the Derive node as a child of an Action node to extract data from that screen at runtime. The extracted data is inserted into the BOD message and used on subsequent screens, usually to continue processing the BOD message.

This table shows the properties for the Derive node:

Property	Description
Description	A short description of the node
Xpath	Set the property to an xpath element that maps to a field on the screen that data is extracted from.

Display Program

Add the Display Program node to map Elements from an inbound BOD message to fields defined in a sequence of LX application screens. The node is a child of an Instruction node. The name property of the Display program must match the Name property of the parent instruction node.

This table shows the properties of the Display Program node:

Property	Description
Description	A short description of the node
Name	The name must be set to have the same name as the parent Instruction node.
Type	The type is a constant with value `ScreenDef`.

Enumerated

Add the Enumerated node to map a value obtained from LX using result set data or PCML data that is required in a BOD message.

This table shows the properties of the Enumerated node:

Property	Description
BOD value	The Bod Value is the value written to the Outbound message.

Property	Description
Description	A short description of the node
LX Value	This is the value retrieved from LX

Exception

Add the Exception node as a child of an Action node. An Action is a container for Exception nodes. Use Exception nodes to allow processing override warning messages in inbound BOD messages. Exception processing allows processing to continue when warnings are returned.

This table shows the properties of the Exception node:

Property	Description
Description	A short description of the exception
Enable	Set this property to **True** to process a transaction when warnings are returned by an LX program. The message must have the defined Message ID. The message is processed using the value given in the Error Exit property. This allows the transaction to continue processing.
	If this property is set to **False**, when the Message Id is returned from LX, the runtime returns the error to the client application and the transaction will not complete.
Error Exit	Set this property to the value that allows the transaction to continue processing. For example, an **F6** or **F14** that is needed to override an exception.
Ignore	Set this property to **True** so that warning messages are not returned to a client application.
	• If the property is set to **False**, all warning messages are returned to the client application and marked a failure.
	• If the Enable property is **True** and the Ignore property is **False**, the transaction will continue to process. However, the warning message is returned.
	LX Extension Integration projects using ION should always set this property to **True**.

Property	Description
Message Id	This is the LX message id for the Exception, for example, `UMG0660`.
Number of Tries	This is the number of times to retry the Error Exit in case the override failed. For example, if a record is locked, you may resend the Error Exit again.
Wait Time	The time to wait before trying to resend the Error Exit value.

Exit Point Data

Add the Exit Point Data node to an Exit Point Model Object to map raw data to names that can be used in an Outbound Model Object tree view. Add the node a child of the Argument 4, Argument 5, Before Image, or After Image nodes.

This table shows the properties of the Exit Point Data node:

Property	Description
Description	A short description of the node
Is Event Field	Set this property to `True` if the raw data maps to an LX event, such as create.
Length	The number of bytes extracted from the data structure.
Name	Set this property to a character string of data that meets the W3C XML standard. A name cannot have XML special characters or blank spaces. The Name is passed in an XML message to the process instruction defined in the BODName node. This Name can be used by an Outbound Model Object project.
Precision	The default is `0`. This is the precision assigned when the type is packed.
Type	Select one of these options: • char • packed • struct

Property	Description
Usage	The default value is `inherit`. Use to process PCML.

Exit Point Definition

Add Exit Point Definition node to an Exit Point Model Object tree view to map arguments passed from an LX event to names that can be used in an Outbound Model Object Tree View. The Exit Point Definition is a child of an Exit Point Mapping node and requires child nodes for mapping purposes.

This table shows the properties of the Exit Point Definition:

Property	Description
Description	A short description of the node
Java Class Package	Deprecated
Name	The name given to this instruction. The name is currently not used.

Exit Point Mapping

Add the Exit Point Mapping node is added to an Exit Point Model Tree view to define the name of the exit point process instruction that is produced from the Model Object. The node requires child Exit Point Definition to define exit point mapping.

This table shows the properties of the Exit Point Mapping node:

Property	Description
Description	A short description of the node
Name	This name must follow the naming conventions for defining the name of an Exit Point project. The property should be the same as that given to the project. The name is a concatenation of the Program and Interface Point as defined in SYS635D1. For example, `PUR500BEXIT01` where PUR500B is the program name and EXIT01 is the interface point.

External Instruction

Use the External Instruction node if the project needs to load and execute a different Model Object. Each Model Object generates a process instruction. At runtime when this instruction executes, the current process instruction stops processing and loads the external process instruction and passes it the current BOD message.

This table shows the properties of the External Instruction node:

Property	Description
Description	A short description of the node
Entry Point Instruction Name	Set this property to the Name of the Entry Point in the external process instruction that is executed when the external process instruction is loaded.
Instruction Type	The type is a constant with value `ScreenDef`.
Process Instruction Name	Set this property to the name of the external process instruction that will be loaded

Expression

The Expression node is not supported. The node was used by early LX Extension integrations to build logical expressions evaluated at runtime. The node was replaced by the If Condition node which allows creation of expressions.

This table shows the properties of the Expression node:

Property	Description
Description	A short description of the node
Expression	Not used

Field

Add the Field node to reset a value for a parameter passed to a Batch Program. The API Field is the parameter and the value that is assigned to this parameter depends on the Variable Type. If the Variable Type is database then the value assigned to the parameter is that extracted from the Database Field. If the Variable type is inbound the value assigned to the parameter is extracted from the xpath to an element.

This table shows the properties for the Field node:

Property	Description
API Field	Set this property to the Name defined in the PCML Model Object. The Name maps to an API data structure field.
Database Field	Set this property if the value to assign to the API Field is extracted from a result set. The Variable Type must be set to database.
Description	A short description of the node.
Name	Set this property to an Xpath to the element whose value is assigned to the API Field.
Variable Type	See "Variable Type options" in this chapter.

Forced Value

Add the Forced Value node to an Action node if an inbound BOD message does not contain a mapping to a field that is required for an LX application. The node provides a property that sets a constant value and a Name that maps to a field on the application screen. At runtime the instruction adds a new element into the BOD message.

For example, to create a Purchase Order requires an action code of 01 but an action code is not provided in the BOD Message. Without an action in the BOD message, the LX program will not execute. Create a new element into the BOD message that maps to the action field. This instruction adds the new element at runtime.

This table shows the properties of the Forced Value node:

Property	Description
Description	A short description of the node
Forced Field Name	Set the Name for the element. This is an xpath name. This name is used to map a field to the Screen Field Mapping element. This is the name of the element that is added into the BOD message at runtime.

Property	Description
Forced Value Type	Select one of these options: • None • Create • Change • Delete • Add • Replace You can map the method that was sent with the message.
LX Value	Set this to the value required by the LX Application. This value is assigned to the Forced Field Name.

Huge Bod Entry

Note: The Huge Bod Entry node is supported only for processing instructions that are used by the LX Extension in ION integrations.

Add the Huge Bod Entry node to an Outbound Model Object tree when the size of the BOD message that is produced can be very large, for example, a PurchaseOrder with 9999 lines.

All outbound messages that are processed in batch must include batch information. The batch information is added into the BODID of a BOD message. The batch information includes these attributes:

• batchIdentifier
• batchSequence
• batchSize

For outbound projects the batchIdentifier is set by extracting the SOABATCH field from the LX ZPA file. Each BOD message produced in the batch is given a batchSequence that is a sequential number starting with 1. The last BOD message produced for the batch is assigned a batchSize that is the number of BODs in the batch.

Add the Node to an Inbound Model Object tree view if the Inbound Model Object must produce an Outbound BOD Message.

The LX Extension always stores inbound BOD messages that have batch information in the LX BATCH_ENTRY file. The Model Object project must include instructions to remove messages from the BATCH _ENTRY and write to the outbox. This node provides properties that provide this ability. See Chapter 5 for examples on defining Huge BODs.

A Model Object tree may use multiple Huge Bod Entry nodes.

This table shows the properties of the Huge Bod Entry node:

Property	Description
Batch ID	Always set this property to **True** to enable batch processing.
Batch Size	This value is currently not used.
Batch Size Field	Set this property when you are initializing the Batch Entry processing. Set this property to set the default value. This is maximum number of Child elements that can write into an outbound BOD message per batch. Set this property when the Bod Status property is set to **pending**.
Bod Id Verb	Set this property only when you need to produce an outbound BOD message from an inbound Model Object project. This verb is used to create the outbound BOD message. The BOD ID Verb may be Sync or Process.
Bod Status	Select one of these options: • Pending: Always initialize the Huge Bod Entry from the Entry point condition. This means define a Huge Bod Entry node and set the Bod Status to **Pending** and the Batch Size Field to a default value. The Pending status causes the BOD message to be written to the BATCH_ENTRY file. • Usable: Set the BODStatus to **Usable** to remove the BODmessage from the BATCH_ENTRY file and write to the Outbox. • None

Property	Description
Huge Bod Batch Mode	Select one of these modes: • None: Define outbound Model Object tree views that require no special processing. • UpdateHeader: Use the node for an Outbound Model Object that requires header information in the BOD to have additional processing. • Insert: Process an Inbound BOD message that contains a User Area element defining the batch information. This insertd the BOD message and the batch information into the LX Batch Entry file. • Extract:Use the node to extract BOD messages from the Batch Entry file if producing outbound BOD messages from an inbound Model Object project. • ExtractAll: Not currently supported. • SendOutbox: Send the current BOD message to the Outbox. • SendInbox: This Mode is not currently supported.
Huge Bod Message Type	Select one of these types: • None • Outbound: Select this option if to produce an outbound BODmessage. • Inbound: Not supported. • UserArea: Select this option if an inbound BOD message contains a UserArea that is used to define batch information.
Release All	This is a true or false option. Set this property when you are processing Huge BODs contained in an inbound BODmessage. Set this property to **True** to extract all BODs from the BATCH_ENTRY file.
Remove Infor Nid From Bodid	This is a true or false value and should be set to **True** if the BOD Id Verb is Process.
Sequence	This is currently not used.

If Condition

Add the If Condition node to the Model Object when decisions are required. The node provides an Expression property that is set to a logical expression evaluated at runtime. The node can also be used to evaluate an arithmetic expression.

This table shows the properties of the If Condition:

Property	Description
Available Methods	See "Available methods options" in this chapter
BOD Action Type	This is not used by the If Condition node.
Condition Type	Select the type of expression that is evaluated by the node. • If: Select this if the expression requires If logic. • Elseif: Select the elseif if the expression requires else if logic. • Else: Select the else if the expression requires else logic. • ArithmeticExpression: Select this if the expression is an arithmetic expression, for example (A/B). • Endif: Not supported. • Contif: Not supported. • While: Select this if you are processing Huge Bod information from the Outbound Message. This is used only when processing an outbound message from the inbound message.
Description	A short description of the function of the If Condition. This is not included in the generated process instruction.
Expression	Enter the logical or arithmetic expression. Use the Expression Builder view to create expressions. See Chapter 1 for the rules associated with valid Expressions. This property should be empty if the Condition Type property is set to else or if the Available Method is set to `SendConfirm`.
Loop Element Name	This is not supported for the If Condition node.

Instruction

Add the Instruction node as a container of other instructions. The instruction can be configured to loop through an element in an incoming BOD message.

This table shows the properties of the Instruction node:
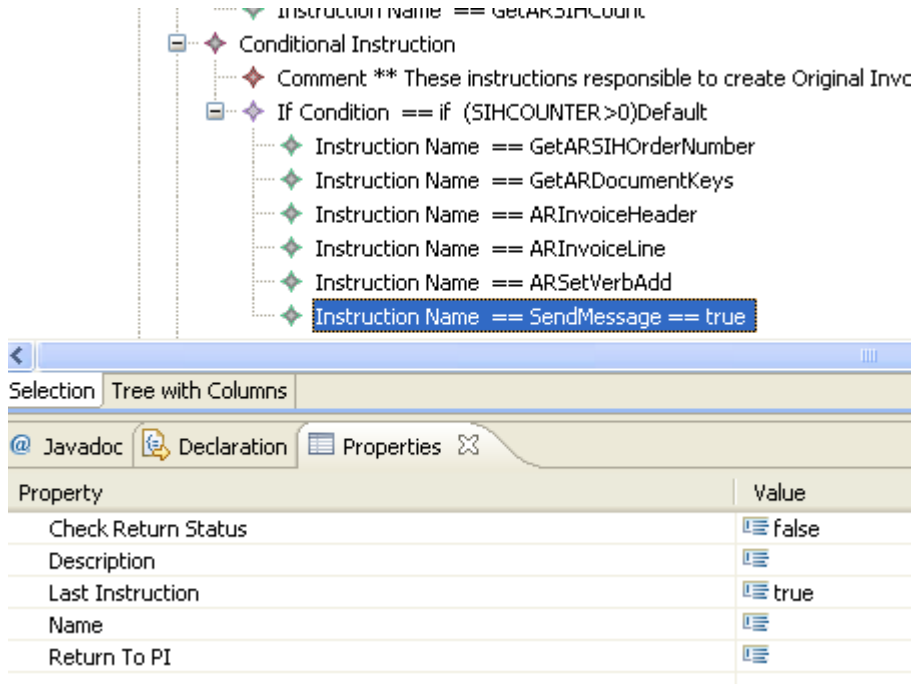
Property	Description
Description	A short description of the instruction. The description is not added into the generated process instruction.
Is Loop Type	Select **True** or **False**. Set this property to true if the instruction loops through an element in an inbound BOD Message.
	To define display programs, this property should be set to **False**.
Name	This property is required. It must contain character data that follows the W3C XML standards.
	Names cannot have blanks.
Organizational Hierarchy	Not used. The default is false.

Instruction Name

The Instruction Name node is added to the Object Model tree when an Instruction node having the same Name should be executed.

This table shows the properties for the Instruction Name node:

Property	Description
Check Return Status	Set this property to true if the Model Object will check if an error was returned by a previous LX application. If set to **True**, the Name must be set to the name of the Process instruction that was running when the error occurred.
Description	A short description of the node

Property	Description
Last Instruction	This is a true or false selection. Set the property to **True** when your process instruction contains instructions that produce additional BOD messages. For example, when producing an Invoice, an AR Invoice may also be created. The Condition node is used to invoke several Instruction nodes that are used to build an outbound BOD message as shown below. When this property is set to **True** it must be the last instruction that is run. This causes an Outbound message to be written to the outbox.

Name	Set the Name to the name of the Instruction node that executes.
Return To PI	This is set when an external process instruction has executed and you need to switch back before you can execute the next instruction. This is the name of the PI to switch control to. When this property is set, the Name is the name of the next Instruction to execute. The instruction must exist in the Return To PI.

Key Element

Add the Key Element if the Exit Point Model Object requires special processing based on action and key data, for example, if Replace messages are processed differently when LX triggers exit program PUR5502POUPDATE. A Key element allows inspection of data from the Exit Point that is triggered.

This table shows the properties available to the Key Element node:

Property	Description
Description	A short description of the priority requirements
Element	The name assigned to a field in the Exit Point Model Object

Locate Row

Add the Locate Row node if an Action used to map to a screen is mapping to a subfile. The node allows the runtime to update the correct row of data to the subfile.

This table shows the properties of the Locate Row node:

Property	Description
Description	A short description of the node
Locate Row Value	This value is normally not used as the Row Value is typically not known.
Note Processing	Not currently supported.
Row Value	This property is set when subfile data is being inserted into LX screens. The Row Value is the Xpath to the element that is used to locate the row. For example, `PurchaseOrderLine.LineNumber` means to fetch the value for the subfile row from the LineNumber in the message.
Xpath	Xpath is the parent of the Row Value. For example, if the Row Value is `PurchaseOrderLine.LineNumber`, the Xpath is PurchaseOrderLine.
Is Empty Row	Select `True` if you are creating a new row in the subfile. When this is set to true it looks for the first empty row and inserts the data. If set to `False`, the Inbound message must contain a valid Row Value.

Loop Element

Add the Loop Element to process child elements contained in an inbound BOD message.

For example, if you are creating a Purchase Order Model Object, the Loop Element adds an instruction in the generated process instruction that at runtime can process each Note in a PurchaseOrderHeader element. In this case the Model object uses a Loop Element node so that each Note element is processed. Processing a Note may include inserting the Note into an LX file.

> **Caution:** The following properties require that Loop Elements be contained within either an Instruction node that has the Is Inbound Loop property set to true or a Condition node that has the Is Inbound Loop property set to true.

- Available Methods
- Loop Element Reference
- Make Subfile Element
- Remove Loop Element.

If you use an Instruction you may add a Conditional Instruction to define the Looping.

If you do not define the looping in the Conditional Instruction, you must add a Loop Element as the first child of the Condition or Instruction. The Loop Element must have these properties:

- Search Loop Element property set to true
- Loop Element property set to the Xpath of the element in the BOD message to loop on.

This table shows the properties of the Loop Element node:

Property	Description
Available Methods	See "Available methods options" in this chapter for a description of the items.
	Only the Equal method is supported for the Loop Element. It is used for removing elements from a linked list of elements.
	If the method is selected the Remove Loop Element property must be set to **True**, and the Element Name must then be the name of an element used to compare values between a current node and a Next or Previous node.
	If a match is found the element is removed. The Loop Element Reference is either Next or Previous indicating the direction of comparison.
	The picture below shows how to configure a Loop Element that uses the Equal method.

For Each Element	Use this property to process individual child elements given the parent. For example you want to process each Components element contained in a ShipmentItem. In this case, the For Each Element is Components. If the For Each Element is set, the Loop Element property must be set to be the Xpath to the value, for example, **ShipmentItem.Components** as shown In the example, below.

Property	Description
Loop Element	This is the Xpath to the child element in the BOD message that is used for looping. Each occurrence of the value set is processed. For example, you may want to add information into the ShipmentItem child elements of a Shipment. Setting the value will process a single ShipmentItem at a time.
Loop Element Reference	The valid values for this property are Next or Previous. Use of the property requires the Available Methods to be set to Equal If set to `Previous` it compares the value of the Loop Element of the current element to that of the previous element and deletes the current if they are equal. This is used only when the Remove Loop Element is set to `True` and Available Method is set to `Equal`. See the **Caution** above to use of this property.
Make Subfile Element	Set the Make Subfile Element to `True` to allow creation of a new child element in a Parent. If this is set to `True` the Loop Element must be the Xpath to the element that is created. In the picture below a ConfirmDetail will be added into the BOD message. See the **Caution** above to use of this property.
Remove Loop Element	If the Remove Loop Element is set to `True` the current element will be removed from a BOD message. Set the Loop Element property to the Xpath of the element to remove as shown in the picture below. See the **Caution** above to use of this property.
Search Loop Element	Set this value to `True` if the Named Element is being used to search the inbound message. The Loop Element property must be set to the child element For which to search.

Mapping

Add the mapping node to map Elements to fields in an outbound process instruction. Add mapping elements in an Inbound Model Object tree if the Outbound Message Instruction is used.

This table shows the properties of the Mapping node:

Property	Description
Class Type	See section "Class Type options" in this chapter.
Conditioned Mapping	Not supported at this time.
Cross Reference	**Note:** This property is available only to LX Extension integrations that use ION connectivity. See "Cross Reference options" in this chapter for a list of options.

Property	Description
Database Field	Set this property to the column in an LX file that is fetched using an SQL Statement. The value for the column is stored in a result set. The value for the column is extracted from the result set and assigned to the Element.
Default Value	Set this property to a constant value that can be assigned to the current element in the event no value was assigned.
Description	A short description of the mapping.
Element	Set this property as an xpath value to an element. For example, a valid Xpath Element is **InvoiceHeader.CustomerParty.Name**. The Xpath uses the period character as a separator of elements contained in the path.
Format	This property requires that the DateTime Class Type is selected. The format that is supported is *YYYYMMDD*.
Is Sender Reference Identifier	Set this property to **True** if this Mapping node is mapping the noun identifier. The noun identifier is an element in the BOD message that uniquely identifies the message. For example, Element PurchaseOderHeader.DocumentID.ID. All outbound Model Object projects must define an element that is the noun identifier. There can be only one Mapping node that defines the noun identifier. For an integration that does not use ION connectivity, this is a Key value, for example, a Purchase Order Number. Set this to **True** for the element that is your noun identifier.
Organizational Hierarchy	Not supported.
Remove Element	This is currently not supported.
Repeating Element	Set the element to **True** if the mapping Element must contain child mapping Elements. This is required for multiple occurrences of an element. For example, **ShipmentItem.SerializedLot.Lot** elements may repeat in a Shipment Item. Each Lot contains child elements.
Separator	Set the Class Type to **DateTime**. The value should be a single character that is used to separate year, month, and the day.

Property	Description
Simple Expression Rule	Select a Simple Expression rule: • AlwaysAddElement: This is the default. The element is always produced into the outbound message. • AddElementIfTrue: Select this option if there is an If Condition logical expression that is associated with the Element. If the condition is successful, then the Element is added into the BOD message. • AddElementIfFalse: Select this option if there is an If Condition logical expression that is associated with the Element. If the condition fails, then the Element is added into the BOD message.
Table Name	This is the name of the table that contains the Database Field.

Mapping Detail

Add the Mapping Detail node as a container that holds Mapping node and Database SQL Statements. This required node allows you to define mapping. The mapping builds the BOD message.

This table shows the properties of the Mapping Detail node:

Property	Description
Description	A short description of the node
Name	This is not currently used but is a reference to this node.
Translation Required	Not supported.

Modification

Add the Modification node if you are using the Narrative node. Add the node to provide defect information into a generated process instruction.

This table shows the properties for the Modification node:

Property	Description
BMR Number	Use this property to set a defect number.
Date	Use this property to enter the date the defect was added.
Name	Use this property to enter information about the defect.

Namespace

Add the Namespace node to add namespace information into the BOD message produced by the process instruction.

Note: You must add this node to outbound Model Object projects developed for the LX Extension integrations that use ION.

This table shows the properties of the Namespace node:

Property	Description
Description	A short description of the namespace.
Is Default Namespace	Select true if you are setting the value for the Namespace URL to the default namespace. The default namespace is defined in the root element of the outbound message to have attribute `xmlns`.
Is Schema Location	Select true if you are setting the Namespace URL to contain as the schema location. The attribute for the schema location is `Prefix:schemaLocation`.
Namespace Url	The URL to add into the foot element of the BOD that is produced. All URLs except the default should have the `xsi` prefix.
Prefix	The prefix to use for the schema location attribute or for other attributes that are not the default.

Narrative

The Narrative Node has no properties but it does have children.

Note: This node is required for all integrations that use ION connectivity.

Noun

The Noun node is required if to build an Inbound Model Object. The Noun defines the name of the inbound process instruction.

This table shows the properties of the Noun node:

Property	Description
Java Package	Deprecated
Name	The name given to the process instruction that is generated. This is the BOD name.
Noun	Select a BOD name from the list. The list contains BOD names that are supported by integrations that use ION connectivity. If you do not find a noun, select **None** and set the Name property.
PI Entry Point Name	This is the name of the instruction in the Model Object that is the instruction that is loaded at runtime when the generated process instruction is loaded.

Outbound Message Instruction

Add the Outbound Message Instruction to an Inbound Model Object tree view if data from the BOD message is also used to produce an outbound message. The node is normally a child of the Instruction node, however, if the outbound message that is produced is determined to be a huge BOD then the Outbound Message Instruction node is added as a child of the Huge Bod Entry node.

The node generates as an Instruction node into the generated process instruction. At runtime, this instruction uses the properties to create Exit Point data that is required to produce the outbound BOD Message.

Note: If the Model Object tree generates a process instruction used by the LX Extension for an ION integration, these nodes may be required as children of the Outbound Message Instruction: Verb, Mapping, Namespace, and BOD Version.

This table shows the properties of the Outbound Message Instruction:

Property	Description
Available Methods	See "Available methods options" in this chapter for a description of the items available.
Entry Point To Process Instruction	This is the name of the Condition node in the Outbound Process Instruction that is to be produced.
Outbound Process Instruction Name	The Name of the process instruction to load.
Program Name	This is the name given to the `<ProgramName>` that is added to the exit point data passed to the Outbound Process Instruction.

Outbound Noun

The Outbound Noun is required to create an Outbound Model Object. The Outbound Noun defines the name of the outbound process instruction.

This table shows the properties for the Outbound Noun node:

Property	Description
BOD Action Code	The default is **Default** and is the only supported Action code.
Description	A short description about the node.
Entry Point Name	This is not required since this name is defined in the Exit Point process instruction and there could be multiple Exit Point/Trigger PIs that call the Outbound Model Object with different entry points. It might be preferred practice to match the Name property of the BOD Element defined in one of the Exit Point/Trigger PIs, but it is not necessary.
Name	The name given to the BOD message that is produced.
Noun	Select a BOD name. The list contains BOD names supported by integrations that use ION connectivity. If you do not find a noun, select None and set the Name

Pcml

The PCML node is the root element of a PCML Model Object tree view. This requires child nodes that are used for mapping API fields to an Element name used in either an Inbound or Outbound Model Object when calling the API.

This table shows the properties of the PCML property page:

Property	Description
Action	Select the method that is executed by the API, for example, **Add**.
Description	A short description of the node

Pcml Data

Add the Pcml Data node to map RPG fields in a data structure to an element that can be used by an Inbound or Outbound Batch Program node. This node is a child of the Pcml Data Entry node. Pcml Data must be defined for each parameter expected by the RPG program.

This table shows the properties of the Pcml Data node:

Property	Description
Description	A short description of the mapping
Init	Optional field that is an initial value.
Length	The number of bytes defined by the field.
Name	The field in the RPG data structure
PCML Parm Types	The RPG parameter type. Select one of these options: • None • Inbound • Outbound • Both Select **Both** if the parameter can be both Inbound and Outbound.
Precision	If the type is packed, specify the precision.

Property	Description
Size Validation Type	Select one of these options: • None • Reject • Truncate • Roundup • Rounddown • RoundHalfUp This is normally set to `None` in the project and set when executing the PCML from the Inbound or Outbound process instruction.
Type	Select one of these options: • char • packed
Usage	Select one of these options: • inputoutput: Default • inherit • input: The parameter is an input parameter only and the value is not returned by the RPG API. • output: The value is not sent as a parameter to the API.
Xpath	Enter only character data. The Xpath cannot contain any XML special characters. This is the name that can be used in the Batch Program that executes the PCML. For example, `warehouse`.

Pcml Entry Point

The Pcml Entry Point is the child of the root element PCML. This node is used when creating a PCML Model Object project. The node defines the name of the RPG program or service program that is executed.

This table shows the properties of the Pcml Entry Point node:

Property	Description
Description	A short description of the program

Property	Description
Entry Point	The name of the RPG program.
Is Service Pr0gram	If the RPG program is a service program set this to true.
Name	The name of the RPG program.
Path	Not used at this time.

Priority

Add the Priority node to an Exit Point Model Object tree view to update the priority of a message in the Safe Box. Add a Priority node as a child node of the BOD Element node. Add the node to improve performance when fetching from the Safe Box. For example, when an Exit Point is activated, the default priority is **4**, however, if the message is not important, resetting the priority to a lower value will fetch the message only after all higher priorities have been processed.

This table shows the properties of the Priority Node:

Property	Description
Action Code Type	Select an action. Setting the action applies the priority rule only if the LX event is of this action.
Is From Inbound	Set the value to **1** if the priority applies only to an Inbound message that requires producing an outbound message. The default is **0** (zero) which means that the LX Event was created from LX.
No Duplicates	This is a true or false value. When set to **True** the Inbound processor will check for existence of a row in the safe box based on key data.
Priority Level	Valid priorities range from **0** being lowest to **9** being highest. By default, all LX Event messages are processed at priority **4**.

Reset Element

Add the Reset Element node to an inbound model Object tree to reset the value of an element before the transaction is processed.

This table shows the properties of the Reset Element node:

Property	Description
Description	A short description of the node
Field	This property defines a field in the database to reset. This is set if the Variable Type is set to database.
Value	If the Variable Type is a constant this will hold the value to set.
Variable Type	See section "Variable Type options" for a list of the options available for this node.
Xpath Element	This is the complete path to the element in the BOD message whose value is to be reset.

Screen Field Mapping

Use the Screen Field Mapping node to map elements in an inbound message to fields on an LX application screen.

This table shows the properties of the Screen Field Mapping node:

Property	Description
Data Type	Define the field as a String or a Decimal. Set the type appropriately.
Date Type	True or false value. If the field is a date, set this to `True`.
Description	A short explanation of the Screen Field, if needed. The generated process instruction does not contain the description.
Element Name	The xpath to a BOD element. • If you are mapping a field in a LX Extension integration that uses ION connectivity and has a BOD template, set the element name by selecting a value from the Xpath View or double click an item in the selection widget on the Search Xpath View. • If you do not have a BOD template you must manually enter the value for the Element Name. Because the Element Name is an XPath, each element in the Xpath must be separated by a period. For example, `ItemMasterHeader.Description` defines the Description element in the BOD message.

Property	Description
Field Name	Enter the screen field name to map the Element to the field.
Length	The maximum length allowed for the field.
Line Type	True or false value. If the field on the green screen represents a field on a line, for example, a Requisition Line, set this value to **True**.
Precision	Number of decimal places. If the Data Type is Decimal, assign the precision.
Sequence	Not currently used
Subfile Type	True/False value. If the field is a subfile, for example Note, set this to True.
Available Action	**Note:** In an LX Connector process instruction always use the default ACFD. See section "Available action options" section for the options.
Class Type	Not used on inbound process instructions.
Default Value	If the Default Value has a value and the BOD message does not have a value for this element, the Default Value is assigned and sent into LX.
Size Validation Type	By default, neither the LX Extension nor the LX Connector runtime truncates values. If the value is too long, or if the precision of a number is too large, an error is returned. Select one of these options: • Truncate: Select this option if the element's value can be truncated. • Reject: Not used. If you are producing a PI for ION integrations, an error returned by LX produces a ConfirmBOD to the Outbox. If you are producing a PI for the LX Connector the error is returned to the client application. Version 2 of the LX Connector writes all errors into the LXCERRLOG.
Cross Reference	See "Cross Reference options" in this chapter for a list of options.
Display Column	Not supported
Display Row	Not supported
IO Attribute	Not supported

Simple Expression

The Simple Expression node is deprecated and is replaced by the If Condition node.

This table shows the properties page:

Property	Description
Condition Type	Select one of these types of condition: • Else • Arithmetic Expression • Contf • while
Default Value	The value to assign by default. This is not a required property.
Description	Short description of the Expression.
Logical Operator	Select the Logical Operator. The options are **AND**, **OR**, or **END**.
Operator	Select the operator. The options are: • None • = • - • + • !=
Variable1	The left operand. If this is a field from the database it must be preceded by a colon, for example **:PHLTM**.
Variable1 Type	See a description of the variable types in the "Variable Type options" in this chapter.
Variable2	The right operand
Variable2 Type	See a description of the variable types in the "Variable Type options" in this chapter

SQL Definition

Note: This node is available for outbound process instructions. LX Extension 2.0 and LX Connector 1.0 and earlier releases do not support this feature.

Use the SQL Definition to map the results returned from an SQL Statement to the value assigned to the element. An SQL Definition must include a child instruction Statement that defines the SQL Statement. To check that a row or rows were retrieved from the SQL Statement add child instructions SQL Success and SQL Failure to the SQL Statement.

This table shows the properties of the SQL Definition:

Property	Description
Expression Count	Not currently supported
Is Array Sql	True or false value. The default is `False`. Set the value to `True` if the SQL Statement that the definition processes retrieves a list of fields and each field must create a repeating child element in the BOD message. See Chapter 5 for an example of the Is Array Sql flag set to `True`.
Is Repeating Child	Not currently supported

SQL Failure/SQL Success

Note: These nodes are available for outbound process instructions. LX Extension 2.0 and LX Connector 1.0 and earlier releases do not support this feature.

Use the SQL Failure and SQL Success instructions to check if rows were returned when the SQL Statement is executed at runtime. The SQL Failure and SQL Success instructions must have one child instruction called SQL Result Set Variables. This instruction defines the variable that holds a value indicating where rows were returned.

This table shows the properties of the nodes:

Property	Description
Description	Short description of what constitutes a success or failure. The description is not written into the generated process instruction

SQL Result Set Variable

Note: This node is available for outbound process instructions. LX Extension 2.0 and LX Connector 1.0 and earlier releases do not support this feature.

The SQL Result Set Variable is a child of either an SQL Success instruction or an SQL Failure instruction. Use the SQL Result Set Variable to define a variable that holds a value indicating that rows were either returned or not returned. You can check this variable in a condition expression to determine a next set of instructions.

This table shows the properties of the SQL Result Set Variable:

Property	Description
Cross Reference	The default is **none**. To define an SQL Result set, you will probably not need any other value.
Description	Short description of the variable. Reference only.
Element Name	Not used in this instruction.
Name	Not used in this instruction.
Name Variable Type	Not used in this instruction.
Parent To Search For	Not used in this instruction.
Variable1 Type	See "Variable Type options" in this chapter.
Set Default	If this is a child of an SQL Success node, this value is assigned when a row returns successfully. If this is a child of an SQL Failure node, this value is assigned if no rows are returned by the SQL Statement.
Set SQL Expression Rule	Set to **Always Add Element**.
Substring	Not used in this instruction.
Value	Specify the name of the element that holds the value assigned to property Set Default. The SQL Success instruction and SQL Failure instruction must have the same Value. For example, a valid value could be **FILE.SQLERROR1** where FILE is the first table in the FROM defined in the SQL Statement (ECH.SQLERROR1)
Variable Type	Select Database. This instruction will store a new element called `<Value>SetDefault</Value>` into runtime temporary memory. It is removed from memory when the associated SQL Definition instruction is completed.

Statement

Add the Statement node if data must be retrieved from LX files to produce outbound BOD messages or to update LX files when processing inbound BOD messages.

This table shows the properties for the Statement node:

Property	Description
Description	A short description of the node
Field	Not currently used.
Looping Types	Select one of these types: • None: Default. Simple execution of the statement. • For loop: Retrieve rows of data, for example lines for a purchase order. An element is created for each row that is retrieved. • For Each Loop : Retrieve rows for a child of the For loop, for example get the notes for the line. • Iteraterows: Not supported.
Remove Previous Result	Set this to true if the Looping type is None and you need to remove the data retrieved using a previous execution of this statement.
Row Number	Not supported at this time.
Statement	This is an SQL statement.
Widget Type	Not supported at this time.

Substring Field

Add the Substring Field node to retrieve a portion of a value from an element in a BOD message.

This table shows the properties of the Substring Field node:

Property	Description
Database Name	If the value to substring is the result of an SQL statement set the Field as the Database Name.

Property	Description
Element	If the value to substring is a value in the inbound message set this as an xpath to the element to substring. If the value to substring is an attribute, prefix the name of the attribute with character @.
Parent To Search For	The Xpath to the parent element that contains the Element whose value will be used. For example, setting this to `ReceiveDeliveryHeader.DocumentReference` indicates that the Element is found in this parent.
Number Of Characters	This is the end index of a string. If this property is set to `0` then the value returns all characters from the Start Position. The string is 0 base that is the start index is 0.
Start Position	The beginning index of the string. The value is `0` base. For example if StartPosition is set to 0 and the Number Of Characters is `8`, it will get 8 characters starting at index 0 and ending at index 8.If the value is `thedograndown` this would return `thedogra`.
Value	If value in the Work Element has a constant value, then the value is sub-stringed.

Thread Rule

Add the Thread Rule node to improve performance in inbound processing. This node allows inspection of a target element at runtime. The target element's value is extracted from an Inbound BOD message and compared to the value of a currently running same-named BOD message. If the value is already processing, the runtime waits to process the new message.

Note: This node is used only by inbound process instructions that use ION connectivity.

This table shows the properties of the Thread Rule node:

Property	Description
Description	A short description of the rule

Variable

Add the Variable node to a Statement node to map elements or fields to a variable placed in a where or values clause. Add the Variables in the sequence in which they appear in the where statement. The Name for each variable must be unique.

This table shows the properties of the Variable node:

Property	Description
Cross Reference	See section "Cross Reference options" section for the list of options.
Description	A short description of the variable
Element Name	The complete xpath to the element in the BOD message to retrieve.
Name	The name that will be set in the where statement. The name must be unique.
Name variable type	
Parent to Search for	Not supported.
Set Default	Not supported.
Substring	Not supported.
Value	Not supported.
Variable type	See a description of the variable types in the "Variable Type options" in this chapter.

Verb

Add the Verb node to define the verb that is published when the outbound message is produced.

Caution: This node is used only by integrations the use ION connectivity. It is required for all outbound process instructions.

This table shows the properties of the Verb node:

Property	Description
Action Code	Select the method that writes to the outbound message.
Description	A short description about the verb.

Property	Description
Priority Level	Valid priorities are 0 – 9; 9 is the highest. This value sets the priority field in the outbox.
Verb	Select the verb. Only Sync and Process are supported at this time.

Verb Element

Add the Verb Element node to add Verb information. This node is a child of the Verb. Verbs can have one or more Verb Element nodes.

Note: The Verb Element is used only by ION Integrations.

This table shows the properties of the Verb Element:

Property	Description
Cross Reference	Select the type. If you are adding a TenantID to the verb, select `TenantID` to fetch the value from the integration cross reference. See the Cross Reference property defined for the Attribute node for a detailed description of options.
Database Field	Not required if child Work Elements are used to define the Field. This is the field retrieved by a previous result set.
Element Name	The name of the element that is added into the Data Area of the BOD message produced by this process instruction.
Value	If the Variable type is constant set the constant value in this property.
Variable Type	See a description of the variable types in the "Variable Type options" in this chapter.

Work Element

Add the Work Element node to add new elements into a BOD message that is used to process the inbound data.

This table shows the properties of the Work Element node:

Property	Description
Available Methods	See "Available methods options" in this chapter for a description of the items available.
Calculate Value	**Note:** LX Connector does not support this property. Set the value to false.
	Set the property to `True` if the Value set for the Variable Type inbound contains an attribute and you want to retrieve the value of the element instead of the value of the attribute.
	For example, if the Value is set to `ShipmentHeader.WarehouseLocation.ID@schemeName` and you want to retrieve the value assigned to element ID with attribute of schemeName in the inbound message then set Calculate Value to `True`.
	To get the attribute value instead, set Calculate Value to `False`.
Description	A short description of what the Work Element does. This is not written into the generated process instruction.
Set Message	A true or false value. Set the value to `True` if the Work Element must be added into the Inbound Message. Most Work Elements will have this value set to `True`.
SQL Statement	Not currently supported. Allows the developer to retrieve a value for the Work Element from the results set of an SQL statement.
Value	This is the value that is given to the XPath element that is added into the Inbound message.
Variable Type	See a description of the variable types in the "Variable Type options" in this chapter
XPath Element	This is the complete path to an element that is added into the inbound message. The Xpath Element must be prefixed by the name of the parent that the element is added to. Use the period to separate Elements. For example, to add a child element XLOC into parent ReceiveDeliveryItem, set the XPath Element value to: ReceiveDeliveryItem.XLOC

Available methods options

Available Methods provide special processing of an inbound BOD message. For example, there are methods to check if Elements in a BOD Message exist, and methods that define line processing used by the runtime. The Available Methods property is used on these property pages:

- If Condition
- Loop Element
- Outbound Message Instruction.
- Work Element

This table lists the available methods options:

Property	Description
None	
changeprocessreplace	Select the changeprocessreplace method to process a Replace inbound BOD message with child elements that require maintenance of an LX subfile; and your model object includes instructions that indicate the child element is changing an existing row in the subfile.
	For example, a SyncPurchaseOrder BOD message may contain several PurchaseOrderLiine child elements. Each child element contains data that maps to an LX subfile.
	The Model Object requires instructions that loop through each child element to determine if the PurchaseOrderLine already exists in the subfile.
	• If the PurchaseOrderLine already exists then the Model Object must check to see if the PurchaseOrderLine should be deleted.
	• If not, then a Work Element is added that has the Available Methods property set to **Changeprocessreplace** and an Xpath Element set to **PurchaseOrderLine**.

Property	Description
addprocessreplace	Select the addprocessreplace method when processing a Replace inbound BOD message that may contain child elements requiring maintenance of an LX subfile.
	The Model Object must contain instructions indicating the child element is a new row that must be added to the subfile.
	For example, when processing a SyncPurchaseOrder, the BOD message may contain several PurchaseOrderLiine child elements. Each child element contains data that maps to an LX subfile.
	The Model Object requires instructions that loop through each child element to determine if the PurchaseOrderLine already exists in the subfile.
	If the PurchaseOrderLine exists in the subfile a Work element is added as a child of the If Condition and this method is selected as the Available Method. The Xpath Element property of the Work Element is set to `PurchaseOrderLine`.
deleteprocessreplace	Process a Replace inbound BOD message that may contain child elements requiring maintenance of an LX subfile. The Model Object must contain instructions that indicate the child element exists in the subfile but should be deleted.
	For example, when processing a SyncPurchaseOrder, the BOD message may contain several PurchaseOrderLine child elements. Each child element contains data that maps to an LX subfile.
	The Model Object requires instructions that loop through each child element to determine if the PurchaseOrderLine already exists in the subfile.
	If the PurchaseOrderLine does exist in the subfile and the PurchaseOrderLine must be deleted then a work Element is added that sets the Available Method to `Deleteprocessreplace` and the XpathElement to `PurchaseOrderLine`.
SendConfirm	Select the SendConfirm method to set an If Condition node with a Condition Type of else to false. The If Condition node Expression property must be empty.
Empty	Not supported at this time.
SetTime	Not currently supported.
SetFirstBlankAddressToCity	Not currently supported.
SqlStatement	Not currently supported.

Property	Description
Exist	Check for the existence of an element in the BOD message.
	The If Condition node is required to check for existence of an element. After you add the If Condition node, set the Available Method to **Exist** and set the Expression property to the Xpath of the element, for example, **(PurchaseOrderHeader.Note).**
HasChildren	Check if an element in a BOD message has children.
	To check for the existence of children, add an If Condition node, select the Available Methods to HasChildren, and set the Expression property to the Xpath of the element you are checking. For example, Expression (PurchaseOrderLine) checks to see if there are children for a PurchaseOrderLine.
IsEmpty	Determine if an element in a BOD message is empty.
	For example, to check if a Note in a PurchaseOrderHeader is empty, add an If Condition node, select Available Method IsEmpty and set the Expression to (PurchaseOrderHeader.Note).
Count	Select the Count if the variable defined in an If Condition node is an AS in an SQL COUNT(1) statement defined in a Database SQL Statements container.
	For example a Statement in the inbound Model Object is set to **SELECT COUNT(1) AS SRVCOM FROM HPC WHERE HPC.PCCOM=':PurchaseOrderLine.Item.ItemID.ID' AND HPC.PCCTYP='1'.**
	Add an If Condition node to use this variable. Set the Available Methods in the If Condition node to Count and the Expression to **(SRVCOM!=1)**.
Has_infor-nid	Not currently supported
No_infor-nid	Not current supported
ExitProcessInstruction	Select this method when the runtime should exit execution of the process instruction. Generally, this is used to exit the PI when a BOD should not be processed.
SendOutboundMessage	Not currently supported.
SUM	Not currently supported.
InsertNonExistingXpathElement	Select this method to process Loop Elements. See the "Loop Element" section in this chapter for details.
Equal	Select Equal to process Loop Elements. See the "Loop Element" section in this chapter for details
SendInbound	This method is not currently supported.

Property	Description
NotEqual	This method is not currently supported.
IsUpper	Determine if an element value in a BOD message is upper case.
	For example, to determine if the value for inbound BOD element ReceiveDeliveryItem.SerializedLot.Lot.LotIDs.ID is upper case, add an If Condition node, set the method to IsUpper and set the Expression of the If Condition node to `(ReceiveDeliveryItem.SerializedLot.Lot.LotIDs.ID)`.
IsLower	Determine if an element value in a BOD message is lower case.
	For example to determine if the value for inbound BOD element ReceiveDeliveryItem.SerializedLot.Lot.LotIDs.ID is lower case, add an If Condition node, set the method to `IsLower` and set the Expression of the If Condition node to `(ReceiveDeliveryItem.SerializedLot.Lot.LotIDs.ID)`.

Available action options

Use these options to indicate on which method types the runtime should add the value to the inbound message. The default is all methods: ACRD.

This table shows the options that are available to the Available Action property in the Screen Field Mapping node:

Property	Description
ACRD	All properties are added for all methods (Default).
A	Value for the element is sent to LX if the method is Add.
C	Value for the element is sent to LX if the method is Change.
D	Value for the element is sent to LX if the method is Delete.
ACR	Value for the element is sent to LX if the method is Add, Change or Replace.
ARD	Value for the element is sent to LX if the method is Add, Replace or Delete.

Property	Description
ACD	Value for the element is sent to LX if the method is Add, Change or Delete.
AC	Value for the element is sent to LX if the method is Add or Change.
AR	Value for the element is sent to LX if the method is Add or Replace.
AD	Value for the element is sent to LX if the method is Add or Delete.
CRD	Value for the element is sent to LX if the method is Change, Replace or Delete.
CR	Value for the element is sent to LX if the method is Change or Replace.
CD	Value for the element is sent to LX if the method is Change or Delete.
RD	Value for the element is sent to LX if the method is Replace or Delete.
M	The value is assigned only if the value in the BOD message is the value expected by LX.
None	The value is never sent into LX.

Class Type options

Use the class type in outbound projects. These class types are available on the Mapping node property page. Several options have been deprecated.

Property	Description
None	Default. No special handling is required.
Enumerated	Select `Enumerated` when the Mapping requires enumeration.
DateTime	Select `DateTime` when the element maps to a date or time field in LX.
Quantity	Not supported at this time.
Amount	Not supported at this time.

Property	Description
Arithmetic	Replaced with an If Condition node that has a Conditional Type set to **ArithmeticExpression**. The expression must be a valid expression and be enclosed in parenthesis.
Concatenation	Not supported at this time. Use the Concatenation Field instructions as child elements of the Mapping instruction.
Default	Not supported at this time.
Attribute	Not supported at this time. Use child instruction Attribute.
Timestamp	Not supported at this time.
Normal Attribute	Not supported at this time.
firstNonBlank	Select this class if the result set contains multiple results. This class type searches for the current field and sets the value using the first child that is not empty in the list.
defaultIfBlank	Checks the current instruction for a value attribute that defines the default value. The value for the current element is retrieved using the field. If the value retrieved from the field is empty the default value is returned.
Choice	Not supported at this time.
default IfEqual	Not supported at this time.
simpleExpression	This has been deprecated and replaced by the If Condition node.
Condition	Not supported at this time.
BODIDUniqueId-	Not supported at this time.

Cross Reference options

Only integrations that use ION provide translation with this property. Use the default value None for other integration projects.

These nodes contain the Cross Reference property:

- Attribute

- Mapping
- Screen Field Mapping

To translate the value of an element, select one of these options to use a value from the LX Extension cross-reference file:

Property	Description
None	No translation occurs. (Default).
DataElement	Translate the value of an element using the SOA Cross Reference (SYS127) program.
AccountingEntity	Select this option if the value for the Accounting Entity is required from an inbound BOD message. There is no translation in the SOA Cross Reference.
Location	Select `Location` if the value for the Location is required from an inbound BOD message. There is no translation in the SOA Cross Reference.
lid	Add the lid attribute to a noun identifier element in an outbound message. This sets the value of the attribute to the value of the LXComponentLID property stored in the LX Extension configuration file.
TenantID	Map the value defined in the SOA Cross Reference file to the TenantID. The TenantID is written into the Verb portion of an ION BOD message.
variationID	Add the attribute variationID to an element that is the noun identifier and that produces a Sync outbound BOD.
SORLxXref	Select `SORLxXref` in an inbound project to translate the noun identifier into an LX Value. This uses the LX Extension cross reference file.
RevisionID	Add a RevisionID as a child element of a noun id. The RevisionID is added to the BODID.

Variable Type options

These nodes use the property Variable Type:

- API Field Mapping

- Concatenation Field
- Field
- Reset Element
- Simple Expression
- Variable
- Verb Element
- Work Element

This table lists the Variable Type options:

Property	Description
inbound	Select inbound to extract the value from an inbound message. When the Variable type is inbound and you are adding a new element into the Inbound message the Value should be the complete path to the element in the inbound message that contains the value you will assign to a new element defined by the Xpath property. Use this to map inbound projects
Database	Mapping to a value that is retrieved from a result set.
Constant	Assign a constant value to an element.
Constant Blank	Set a blank space as a value to an element.
API Field	Map the value to an API Field. API Field nodes are defined in a Batch Program instruction.
index	Select index when a sequenced attribute is required for a repeating element. The attribute must be sequenced.
Data Element	Reset an element in the inbound message if the value extracted from that element must be translated using the LX cross reference. Use in inbound process instructions. Typically, a reset requires a Work Element that contains a child Reset Element.
SorLxXref	Select SorLxXref in an inbound project to extract noun identifier attributes such as location or accountingEntity from a noun identifier element.
Arithmetic Expression	Select ArithmeticExpression when the value is an arithmetic expression that must be calculated. Enclose all values of this type in parentheses, for example, `(:ShipmentItem.Components.Quantity*:ShipmentItem.PlannedShipQuantity)`
TenantId	Select TenantId in an inbound process instruction to extract the value for the TenantID from the BODID.
Location	Select location in an inbound process instruction to extract the value for the location from the BODID.
AccountingEntity	Select `AccountingEntity` in an inbound process instruction to extract the value for the AccountingEntity from the BODID.

Property	Description
FromLogicalId	Select **FromLogicalId** in an inbound process instruction to extract the value for the logicalID of the sender.
Messageid	Select **MessageId** in an inbound project to retrieve the unique message ID given to the message currently being processed.
actiontype	Select **actiontype** if an inbound instruction must retrieve the value for the attribute actionType from the inbound message.
verb	Use this data type in inbound projects to check the value of the Verb that is received in the inbox. For example, Process or Sync.
Outbound	Select outbound to extract the value from an outbound message.
BatchIdentifier	Select **BatchIdentifier** if the inbound project must extract huge bod identifier information from a UserArea in the bod message.
CurrentElement	Use this data to build outbound projects when you use multiple instructions to set the value for the element. For example, the first instruction may retrieve a value from the current tree. The next instruction updates this value with another value, perhaps by using the Arithmetic Expression type. The Current Element indicates that this element has not yet been added into the outbound tree so retrieve the data from global memory. See Chapter 5.
BatchSequence	Select **BatchSequence** if the inbound project must extract huge bod sequence information from a UserArea in the bod message.
BatchSize	Select **BatchSize** if the inbound project must extract huge bod size information from a UserArea in the bod message.
BatchSORId	Select the **BatchSORId** in an inbound project to retrieve the value for the BODID.
BatchKeyData	Not a currently supported type.
ThreadRule	Not a currently supported type.
SQLErrorCode	Select **SQLErrorCode** if a Work Element is a child of the Statement Node and you want to check the value of a Work Element to determine the error. The Value property in the Work Element should be a variable that writes to the database.

Chapter 3   Creating inbound process instructions

This chapter provides instructions to create the model object project, to add nodes to the model object tree view, with examples. Appendix B contains a table describing the node Parent/Child relations. Chapter 2 contains a description of all nodes available to the LX ION PI Builder as well as the property page definitions for each.

Overview

The LX ION PI Builder provides a Developer Wizard to create an Inbound Model object. Creation of the Inbound Model Object opens a Tree in the Eclipse designer view. Add new Child nodes to the tree. Nodes added to the tree provide various types of instructions which are used by the LX Extension or LX Connector runtime to process BOD messages. After all nodes are added, the Model Object project is generated into a process instruction. The process instruction includes all of the instructions that were added into the tree, including these instructions:

- Map Elements in a BOD message to an LX legacy application
- Provide conditional logic when processing a BOD message
- Allow navigation through a BOD message
- Allow modification of an incoming BOD message
- Allow SQL Statements to process at runtime
- Allow communication with IDF Objects using IDF System-Link (see Appendix D).
- Support for LX 4.0 expanded fields (see Appendix E).
- New Instructions added for Extension 3.0 outbound (see Appendix F).

The LX ION PI Builder provides two techniques for creating Inbound Model Object projects.

Note: The special features referred to in this chapter are nodes added to the tree that generate instructions used by the LX Extension and LX Connector runtime when processing BOD messages into LX. See section 'Properties of the Action Node' for a description of all special features.

References in this chapter to PI refer to the Process Instruction generated from the Model Object project.

Note: References to Table A refer to Table A in Appendix B.

Technique 1

To create a model object, you can use the Retrieve Screen Fields view introduced in Chapter 1. This view allows you to import metadata that is created when you use the display file field description (DSPFFD) command. When you provide an Out File as well as a list of Display File Names, data is extracted from the output files and used to create a tree in the designer view. The tree that is constructed contains an Action Code node that allows navigation through an LX legacy program. You must modify the tree. This technique requires good knowledge of LX as well as the special features required by the LX Extension or LX Connector runtime. You are responsible as the developer for mapping all relevant fields to Element Names in the BOD. You are also responsible for adding all special features required by Actions (display screens) and all required conditional instructions and required database retrieval commands.

Technique 2

Another technique is to create an Inbound Model Object from an existing LX Connector process instruction. These process instructions are released with LX Connector 1.0 and 2.0. The advantage of this technique is that LX Connector process instructions already include the special features. This technique creates a Model Object by generating the object from the LX Connector PI. LX Connector process instructions generally contain at a minimum all of the screens required for navigating through an LX application. To modify the generated Model Object, open it in the designer view.

If you are building an LX Extension inbound process instruction that uses ION for connectivity you must remap the field mapping to an appropriate Element Name in the BOD. You are also responsible for adding required conditional instructions, required database retrieval instructions, and required API instructions. If you are creating a custom LX Connector inbound processing instruction you can rename Element properties as needed.

See the process instructions that are delivered with the LX Extension or with the LX Connector. If they are available to you, you can use them as templates for building inbound process instructions.

Manually create the model object

If neither Technique 1 or Technique 2 are used, you can manually add nodes to the Inbound Model Object. See "Using the Designer view" in Chapter 1.

We do not recommend this method. This method requires extended knowledge of the LX Extension or LX Connector runtime.

Nodes to add to the tree

See Appendix B for Table A for a description of the relationship between nodes on the Model Object tree view. See Chapter 2 for a complete description of the nodes available to the tree.

Using technique 1 to create an inbound model object

We recommend that you use the techniques outlined to create an inbound model object. Use Technique 1 if there is no LX Connector process instruction available. If you have access to an appropriate process instruction use Technique 2.

1   Select your resource project in the navigator view of the resource perspective.

2   Right click on your project and from the menu, select **File>New>Project**.

3   Navigate to the Example EMF Model Creation Wizards folder.
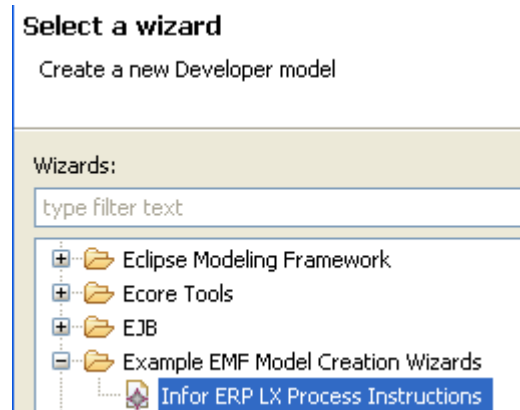
4   Select `Infor LX Process Instructions`.

5   Assign a name to the project. For example, if you are developing an inbound process instruction, use the noun as the name: `NounNameInbound.developer`. All File Name values must end with the `.developer` extension. Inbound in the name implies that the project is the inbound definition of the BOD.
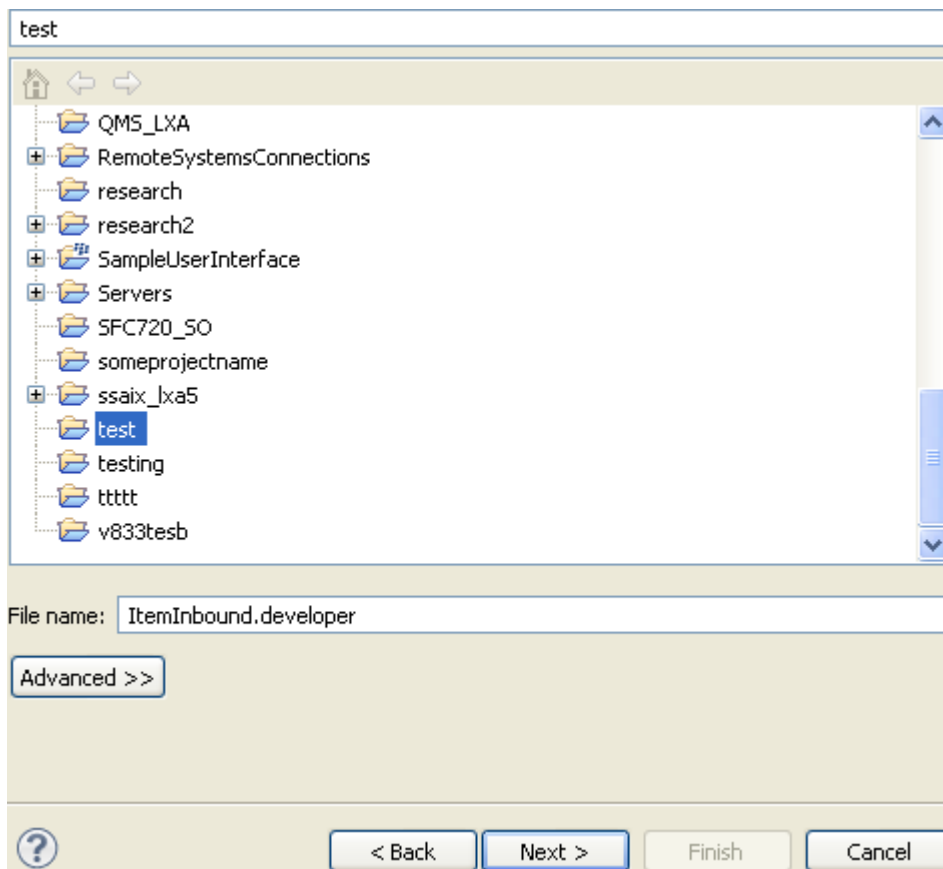
6   Click **Next**.

7   Select the Inbound model object. Accept the XML Encoding value and click **Finish**.
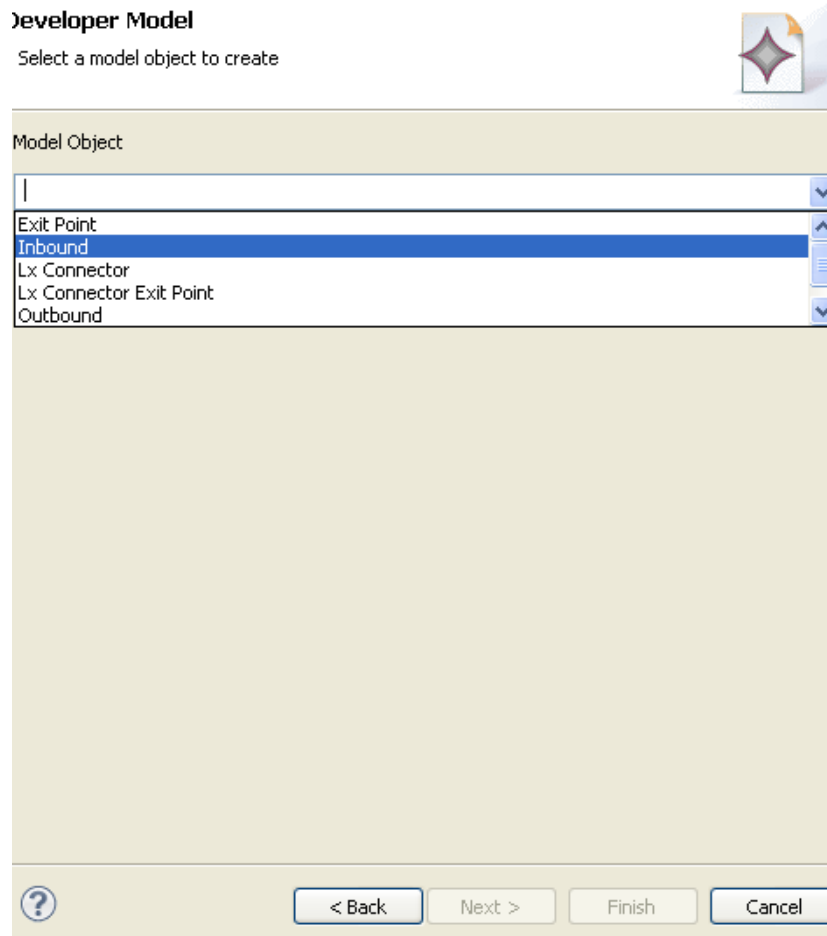
8   The designer view displays an inbound project tree. To create the inbound process instruction, add child nodes to the tree.
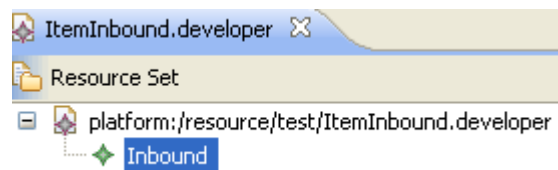
9   To create a PI that navigates through LX applications screens to process a BOD message into LX, see section "Add Display Program using Technique 1" in this Chapter.

Adding Display Program node

After you create the new Model Object, add nodes to the root element Inbound. In this section we will create a PI that allows navigation through screens of an LX application.

Table A shows that root element, Inbound, requires a child called Noun. To add a Noun node select the Inbound node, right click the node and select New Child Noun.

The Display Program node is used to map LX legacy applications to elements in the BOD message. Table A shows that the Instruction node is the parent of the Display Program. To produce a PI that navigates through the screens add a Display Program node to the tree. Select the Noun, right click and select New Child Instruction. Then select the Instruction node, right click and select New Child Display Program.

After adding the nodes to the tree your Model Object tree view should look like this:

Now you can add child nodes to the Display Program node that provide instructions to navigate
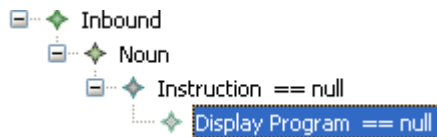
through LX application screens. Use the Retrieve Screen Fields View to retrieve meta data used to build the tree. This technique requires data to be retrieved from an out file location.

- Create the files in a library.
- Update the properties page for the Noun.
- Update the property page for the Instruction.
- Update the property page for the Display Program.
- Import data from the files into the display program.
- Generate the skeleton process instruction.
- Map screen fields to the Xpath value.

These topics include each of these steps in detail.

Creating the files in a library

Building a process instruction that can execute navigation of LX application screens requires gathering file field descriptions from one or more display files. The **DSPFFD** command retrieves field information for a display file. The process instruction is built over an entire LX application which may have one or more screens. You must include all required display files that must be navigated through when the application is invoked. Use the Retrieve Screen Fields View to enter the names of the display files and the name of the library to write output files used by the **DSPFFD** command. When you click OK on this screen, a CLP program is created that invokes the **DSPFFD** command over each required display file.

Updating the property page for the Noun

Before using the Retrieve Screen Fields View see Chapter 2 for a description of the properties of the Noun in the tree view.

In the tree view select the Noun node. If the Properties Page does not show, right click on the Noun node and select **Show Properties View** to open the property page.

See Chapter 2 for the properties available for the Noun node. For this example, set the Name of the noun to `ItemNote`.

Updating the property page for the Instruction

Select the Instruction node in the tree view to display the property page. The properties for the Instruction node are shown in Chapter 2. In this example, assign the Name property to `ItemNote`.

Updating the property page for the Display Program

Select the Display Program node in the tree view to set the properties for the Display Program. The properties for the Display program are defined in Chapter 2.

To continue with the example, assign the Display Program the name `ItemNote`, the same name you gave to the instruction.

If you double-click on the Display Program node, the Retrieve Screen Fields View displays in the Eclipse framework. This view is used to retrieve information from the files you created using the `DSPFFD` command.

At this point, you can create a skeleton Process Instruction by retrieving the data from the System i. See "Importing data into the display program."

Importing data into the display program

Note: BOD Templates are available only to LX Extension integrations using ION connectivity.

In this example, the display programs that are retrieved are INV190F1 and INV190F2.

1   Open the Retrieve Screens Field View if it is not already open. To open it, double click the Display Program node added in Updating the property page for Display Program section above or select **Window > Show View > Other > Retrieve Screens Fields View** from the Eclipse menu.

2   Specify the Host machine and the name of the Library that contains the INV190F1 and INV190F2 display programs.

The value for the Output File is the name of the LX library where the output files are placed after DSPFFD command is invoked.

> **Note:** LX library should not be in any LX environment *LIBL.

- If you are simply mapping screen fields to element names, you do not need to supply Table values. Typically, if you are mapping an inbound process instruction you will not map database fields. You may add multiple program names into the Display File Names edit box. If there is more than one file, separate the names with commas.

- If you are creating a process instruction to use with an LX Extension integration using ION connectivity and a BOD template is available, select the BOD template. The BOD template is supplied by the Development team. In our example, we are adding a note to a Requisition. For this example we have a BOD template named SyncRequisition.xml, so select it using the browse button. The template is used to map screen field values to BOD Element names.

- If you are creating a process instruction for use with the LX Connector there is no BOD template so leave this field empty.

3   Select either **All** or **Inbound Only** from the **Inbound/Outbound Attribute** selection box. If you select **Inbound Only**, only those fields that are enterable will populate into the inbound process instruction.

4   Specify the System i user name and password.

5   Click **OK** again to update the example project as shown below. If errors occur during the retrieval, then a message will display in the connection info widget.

> *RequisitionInbound.developer
>
> Resource Set
>
> ☐ platform:/resource/integration_folder/RequisitionInbound.developer
>   ☐ ◆ Inbound
>   ☐ ◆ Noun Requisition
>   ☐ ◆ Instruction == ItemNote
>   ☐ ◆ Display Program == ItemNote
>   ☐ ◆ Action Code Add
>   ☐ ◆ Action 1 == INV190D1 PANEL01== ENTER
>   ☐ ◆ Action 2 == INV190D1 MSG SFL RECORD== ENTER
>   ☐ ◆ Action 3 == INV190D1 MSG SFL CONTROL== ENTER
>   ☐ ◆ Action 4 == INV190D2 PANEL01== ENTER
>   ☐ ◆ Action 5 == INV190D2 MSG SFL RECORD== ENTER
>   ☐ ◆ Action 6 == INV190D2 MSG SFL CONTROL== ENTER

The skeleton contains a mapping of green screen fields. As the developer, you are responsible for mapping a screen field to an Element name in the BOD message.

6 Delete all Actions that are MSGS SFL RECORD or MSG SFL CONTROL. In the Actions which represent a screen in the navigation sequence delete any Screen Field Mappings that are not used as input fields on the screen. In this example, delete Action 2, Action 3, Action 5, and Action 6.

7 This leaves Action 1 and Action 4. This implies that to create a Note for the Requisition requires pushing the screen defined as Action 1 followed by pushing the screen defined by Action 4. To change the sequence from Action 4 to 2, expand Action 4.

8   When you change the sequence from 4 to 2, this screen is the second screen to navigate. Delete all Screen Field Mappings from the Action that are not input capable.

Mapping the screen field

If the BOD Template is available, map the screen field to an Element Name using the Search Xpath View or scroll through the list in the Xpath View.

Note: The XPath View and Search XPath View are only available if you are using a BOD template. If you do not have a BOD template, manually add mapping information into the property page for a node and can skip the section on mapping Using the Xpath View.

Using the Xpath view

Use the Xpath View to find an element and map the xpath value.

1   Open the **Properties** page of the Xpath View.

2   Select the **Screen Field Mapping** node on the tree that you want to map to.

3   Navigate to the Xpath View and scroll through the list to find the row to map. Click on the row to select it.

4   Right-click the row to bring up the context menu.

5   From the Context menu, select **Assign Xpath**.

In the example, below, the screen field X02PNDES is mapped to the RequisitionHeader.Note.
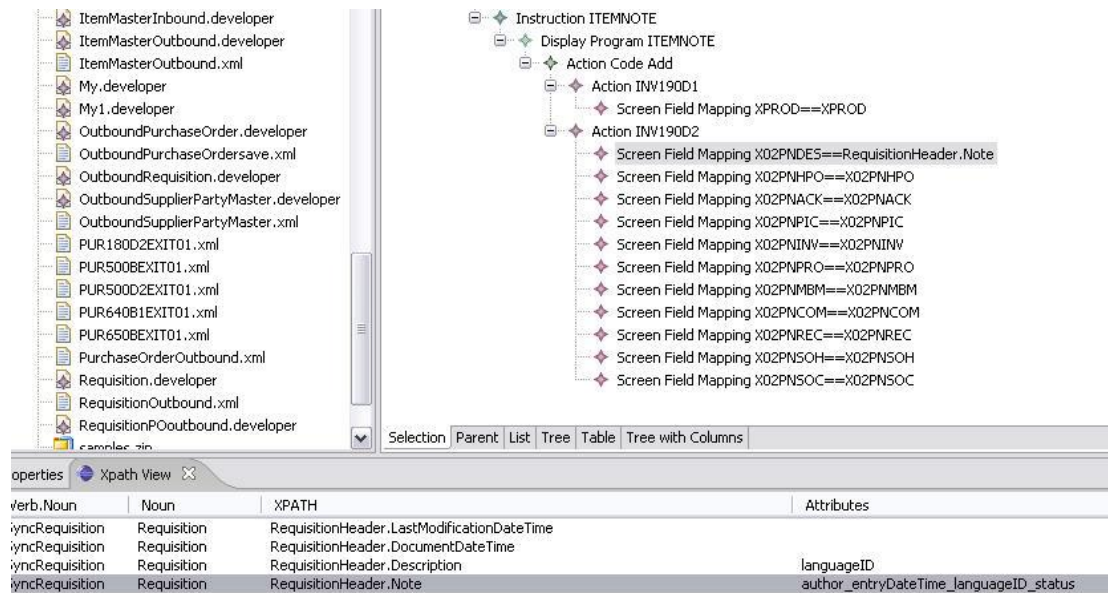
We recommend that you map only the fields required. However, do not delete any of the unmapped fields, because other integrations may require mapping of these fields. In this instance, you are mapping an element contained in a BOD to a field on a green screen.

Using the Search Xpath View

Use the Search Xpath View to find an element and map the Xpath value.

1   Double click on a Screen Field Mapping Node.

2   Select the BOD Template if the XPath view has not already been opened. If the XPath View is already open do not browse again.

3   This view searches for all occurrences of a given xpath in the Xpath View and presents a subset of the view in a selection box as shown below.
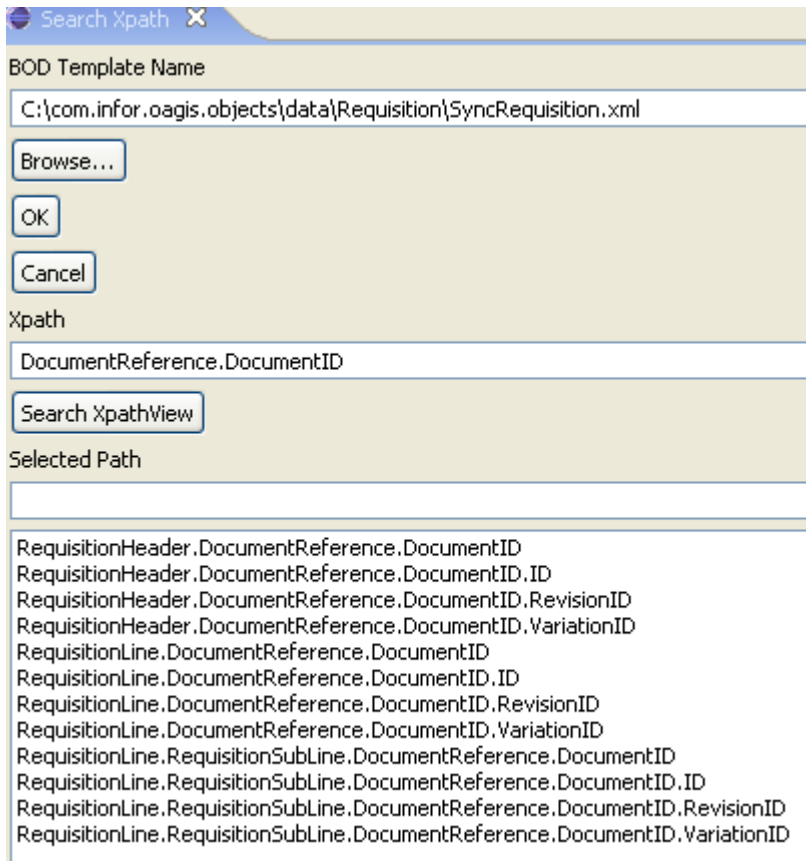
4   You can map a row in the selection box to a Screen Field Mapping by first selecting the Screen
    field Mapping and then double clicking on the item in the selection box to map as shown below.
    Double clicking an item maps to the Screen Field Mapping Element property in the property
    view.
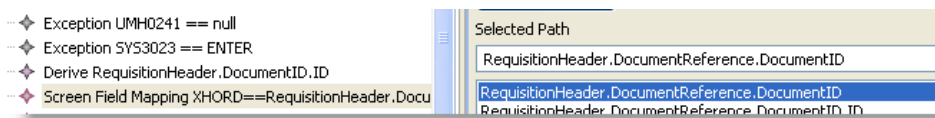
Using technique 2 to create a model object

You can use Technique 2 if the LX Connector Process Instructions are available. Internal Infor
Integration projects have access to these Process Instructions, but external projects do not unless
the developers have licenses for the LX Connector. This technique uses an existing LX Connector
process instruction to create a Model Object project. The Model Object created is opened into the
designer view by double clicking the Model Object project that is created from the PI. To use an
existing process instruction, you need to import the process instruction into the Resource project that

you created. Import the process instruction you want to modify into your project by executing
File/Import/FileSystem, as described below.

1   From the **Eclipse** menu, select **File > Import > General >FileSystem**.

2   Navigate to the directory that contains the process instruction.

3   Select the process instruction that you are importing.

4   After the file is imported, select the file in the project folder and right click to bring up the Context
menu.

5   Select **Infor LX Process Instruction**, then **Edit Inbound PI**. This creates a project in the
resource folder which is the name of the file that you imported.

For example, if you need to use display programs for INV500 for the BOD message that is
received, the Inventory.xml set of process instructions which are available with the LX Connector
project contains these instructions. In this example, the project would be called
Inventory.developer.
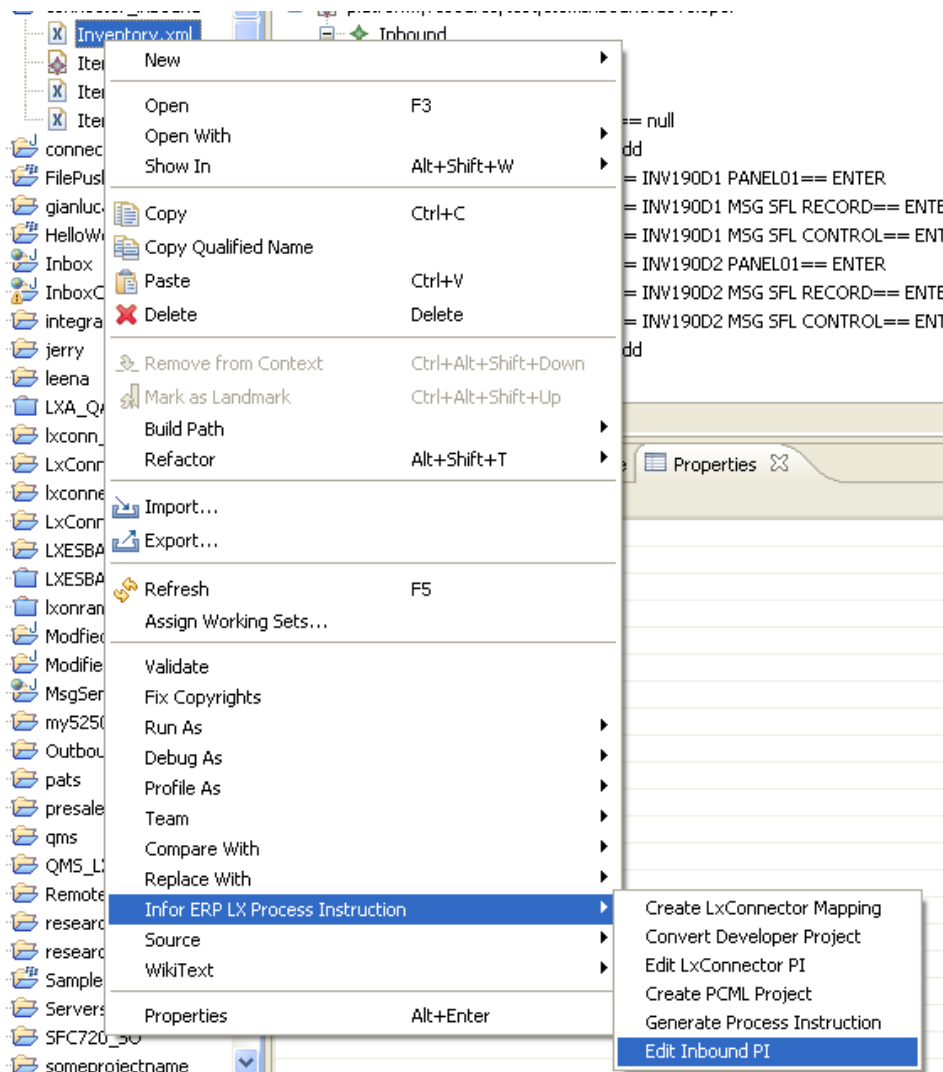
6   Rename the project to describe the new PI, such as
   `InventoryAdjustmentInbound.developer`. The example below continues with the original
   name, `Inventory.developer`.

7   Double-click the Inventory.developer to open the created project.

8   In the Design window, the open project looks like the following screen:
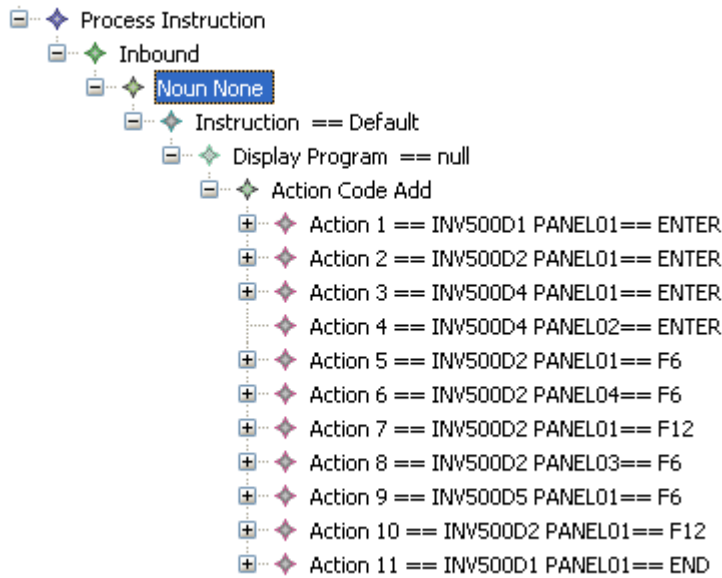
9   Select the **Noun** node

10 From the **Properties** page, select the correct BOD name, for example, `InventoryAdjustment`.

11 Assign names to the Instruction and the Display Program.

12 The imported structure is displayed when you expand the nodes.
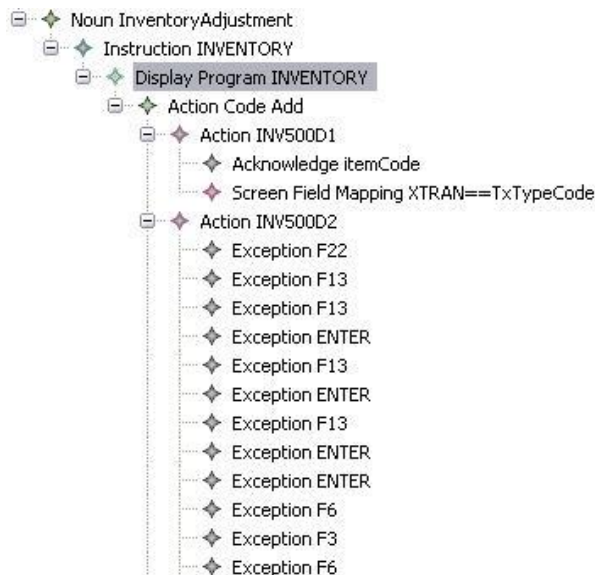
13 Map fields on the green screen to Xpath values in the Xpath View. This step applies only to LX Extension integrations using ION connectivity. If you are not using a BOD Template, you will have to use the property page to map Element and fields to the node.

a   To open the Xpath View, double click on either a Screen Field Mapping node or on a Display Program Node. Double click on the Screen Field Mapping to open the Search XPath View for which you can select a BOD Instance to retrieve. Double click on the Display Program to open the Retrieve Screen Fields View from which you can select an instance of the BOD.

b   Click **OK** to open the Xpath View with the instance information. Only one Xpath View can be opened at a time.

We recommend that you map only the fields required. However, do not delete any of the unmapped fields, because other integrations may require mapping of these fields. In this instance, you are mapping an element contained in a BOD to a field on a green screen.

Note: The project contains all of the functionality that the LX Connector supports. Exceptions are added automatically, as are Forced Values and Acknowledge elements. These features are discussed in "Features in display program process instructions."

Features in display program process instructions

You can add several types of features to a Display Program instruction node. The LX Extension and LX Connector use these features at runtime to perform special processing. The following features are available:

- Acknowledge
- Exception
- Forced Value
- Derive
- Locate Row

These features are explained in the included sections.

Acknowledge

The Acknowledge is a child that you can add to an Action node in the tree.

> **Caution:** Do not use an Acknowledge feature when building process instructions that are processed through the LX Extension using ION connectivity. ION publishes an Acknowledge message when a component receives a Process message.

The information in the Acknowledge node is passed back to a client application which if useful when using the EPR LX Connector. Add an Acknowledge node to an action that has a Screen Field Mapping Xpath that you want returned to the client application.

Exception

You can add Exception nodes to Actions. Exception nodes represent the Warning messages or Override messages that are presented in the display screens. The Exception nodes tell the LX Extension and the LX Connector how to handle a warning message that is returned from LX. If the warnings are not addressed, LX cannot complete the transaction. Add an Exception to an Action for each override error that can occur on the green screen. In other words, an Action can have many Exception nodes.

After you add an Exception node, you must set properties for the Exception. Use the Exception Properties page to enter the Error Exit value. The Error Exit value is the function key that allows processing of the exception so that the transaction can continue. For example, if a UM00054 error (Line will be added to delivery. Press **F22** to continue.) is encountered, you can override the error by sending an Error Exit value of F22 to allow processing to proceed. This means that at runtime, the LX Extension or LX Connector will send an **F22** message to LX so that the next screen sequence is returned.

An Enable property can be set to `True` or `False` for an exception. If you set it to `True`, the exception processing is enabled (processing can proceed). If you set it to `False`, exception processing is not enabled and the transaction fails.

If you do not enable a warning in the process instruction, and you are using the LX Extension using ION connectivity the warning message is returned in a ConfirmBod messages showing the error. When building a LX Extension integration that uses ION connectivity we recommend that you enable all override exceptions. If you are creating an LX Connector process instruction there is no ConfirmBOD message. Instead, the warning is returned to the client application. Version 2 of the LX Connector stores all messages returned from LX in the LXCERRLOG and the LX Connector Inbox.

The Ignore property can also be set to `True` or `False`. This property indicates whether the message returned from LX should be passed as a warning to a client application. If you set this value to `True`, the client application will not be informed of errors that have been received and overridden. If you set this value to `False`, the client application will receive notification of each error that was received and overridden.

In integration projects that are using the LX Extension using ION connectivity always set the Ignore property to `True`. Warning messages are handled as errors, and errors cause a ConfirmBOD message to be sent to ION. This is also recommended if you are handling LX Connector exceptions. The goal is to complete transactions without errors or warnings.
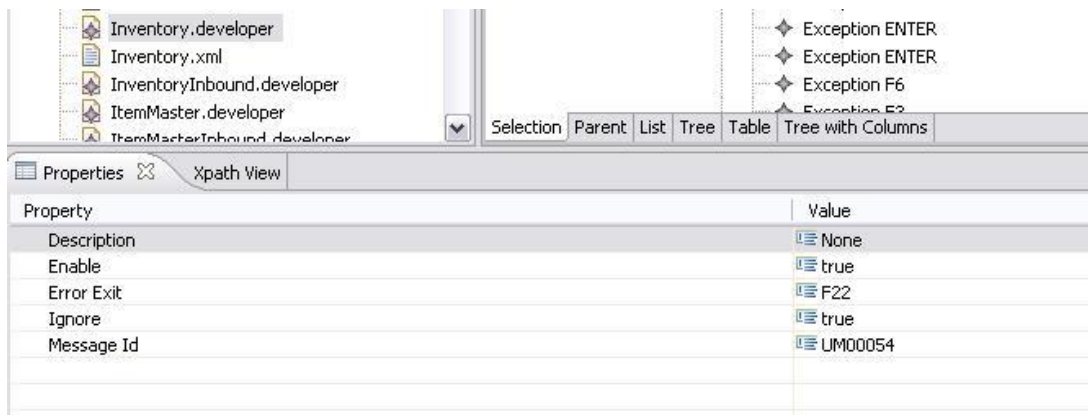
Forced Value

The Forced Value node can be added as a child of an Action node. This node allows mapping an action, such as create on an LX screen to an element in a BOD Message. In this case the Element is not required to be added in the original BOD message by the Sender but is added at runtime by the LX Extension or LX Connector. You can also use a Forced Value node when the inbound BOD message does not contain an element that maps to a required field on the green screen application. For example, to create an Item via the INV100D application green screen, users are required to enter **1** in the option field. The transaction cannot be completed unless **1** is entered in this field.

Look at the available Screen Field Mapping elements for the INV100D1 Action. If the required field, In this example, XACT, is listed, you select the XACT Screen Field Mapping and set the Element Name in the properties view to a unique Xpath value. In this case, you could map the Element Name to ItemMasterHeader.XACT. After you assign the Xpath to the field, you should add the Forced Value node to the Action. You must assign the Xpath value defined in the Screen Field Mapping to the Forced Field Name property of the Forced Value node.
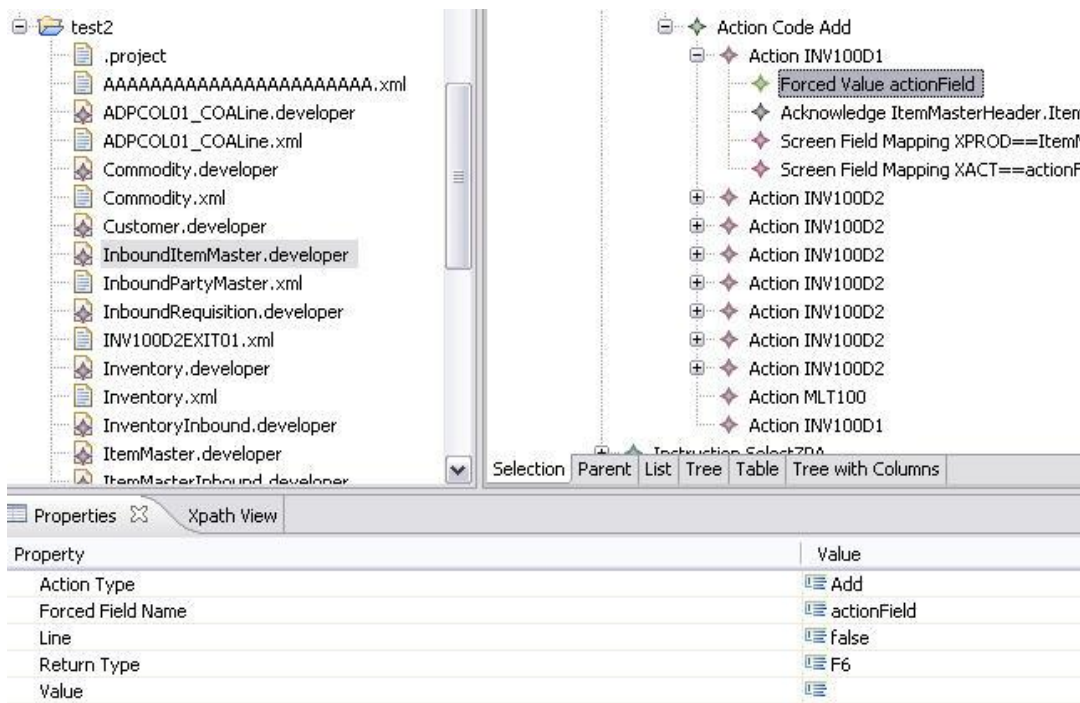
See chapter 2 for a description of the properties of the Force Value node.

Note that an Action can have one or more Forced Values. For example, an Action could contain a Forced Value for ActionType Add and another ForcedValue for Action Type Change.

Derive

Add the Derive node to the tree when the key value is a calculated value that must be returned when a message requires an AcknowledgeNoun message or when an Acknowledge node is used. When you add a Derive to an Action node, you are asking the runtime to derive the value from an LX screen and to return that value in an Acknowledge message if the incoming message received was a Process message.

The Property View for Derive contains an Xpath property that must match the Screen Field Mapping Element Name value. For example, transactional BODS such as Requisitions contain a DocumentID.ID Xpath whose value is determined during the creation of the Requisition. It is your responsibility as developer to determine what screen the Requisition number should be scraped from. The easiest way to determine this is to go through the screens from a System i server session. The screen that has the requisition number on it and also has a Screen Field Mapping is the Screen (Action) that the Derive node should be added to.

For example, PUR500D3 Panel02 contains a Screen Field Mapping for XHORD which contains the Requisition Number. Therefore, in the process instruction for the Action, add a Derive and then set the Xpath property for this node to be the same Element Name value that the Mapping has (RequistionHeader.DocumentID.ID). This causes the LX Extension to retrieve the requisition number from field XHORD on screen PUR500D3 Panel02. The derived value is added to the

AcknowledgeNoun message when a Process verb is received. In the case of an LX Connector process instruction this value will return to the client application.
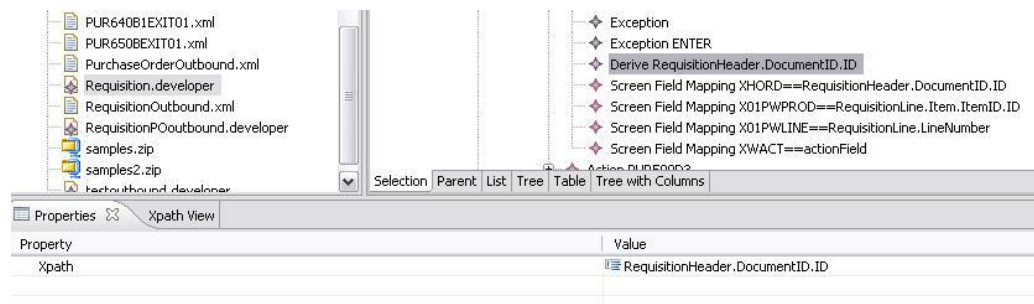
Locate Row

Add the Locate Row node when you are producing a process instruction that contains a screen that has subfiile data. The LX Extension and LX Connector process a single row of data at a time. For example, an inbound ProcessRequisition message may contain one or more lines. When adding subfile data (line information), the runtime adds each line of the inbound message by locating the first empty row of the subfile. The Locate Row feature is added to an action that contains subfile data. The Properties view for this feature contains the following properties:

- Note Processing: Select **True** or **False**. Set the property to **True** if the Action refers to a screen that allows for note entry. The LX runtime performs special handling of notes.

- Row Value: This property is the xpath to the element in the inbound message. The element's value is used to locate a particular row.

- Xpath: This property is an xpath to the element in the inbound message that contains subfile data.

When you add a Locate Row feature to an Action node, you are asking the runtime to use an Xpath value to locate a subfile row. For example, to change a line in a Requisition, the Row Value is the element used to locate the row, such as RequisitionLine.LineNumber. This means the value of the LineNumber in the parent element (RequisitionLine) is used to locate a subfile row.
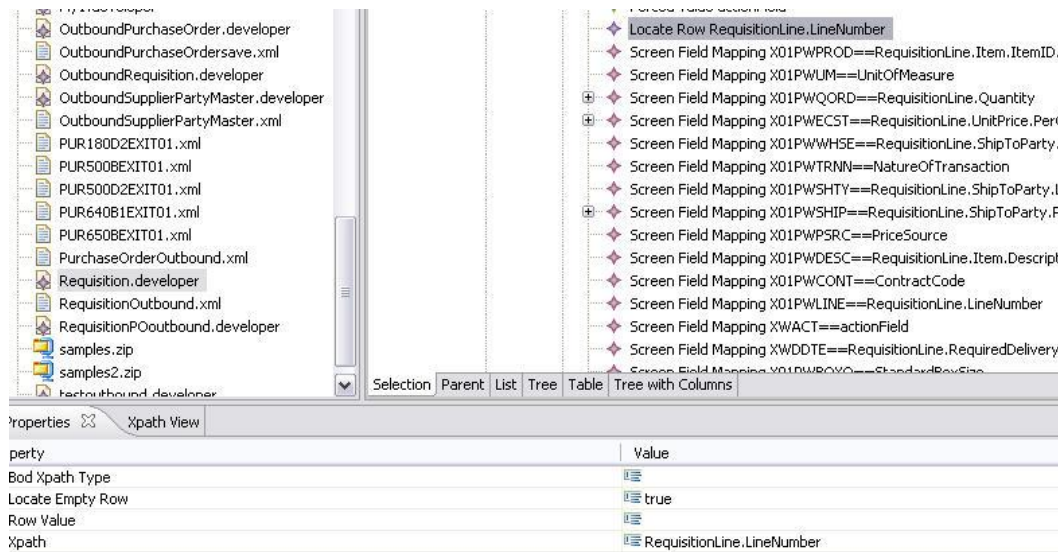
As shown below, the row is in the subfile.
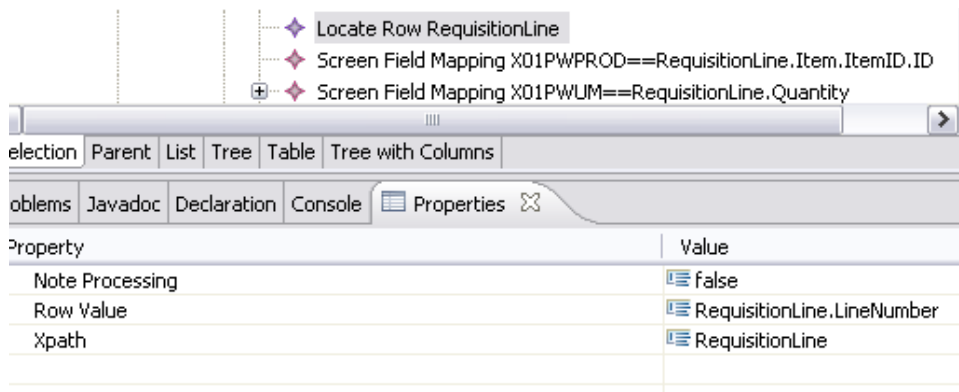
Setting the entry point condition

Inbound process instructions are invoked when the LX Extension or LX Connector retrieves a message from its Inbox. When the inbound process instruction is loaded there must be a start point where the runtime starts processing the instructions. For most inbound process instructions the entry point is a Condition Instruction. To create an entry point Condition follow these steps.

1   Add a new child Condition to the Noun node.

2   In the property view for the Condition, set the property values.

    a   Set the Inbound Loop to `false`.

    b   Set the Name to the PI Entry Point Name defined in the Noun property view as shown In the example, below. The Type is automatically set to Condition.
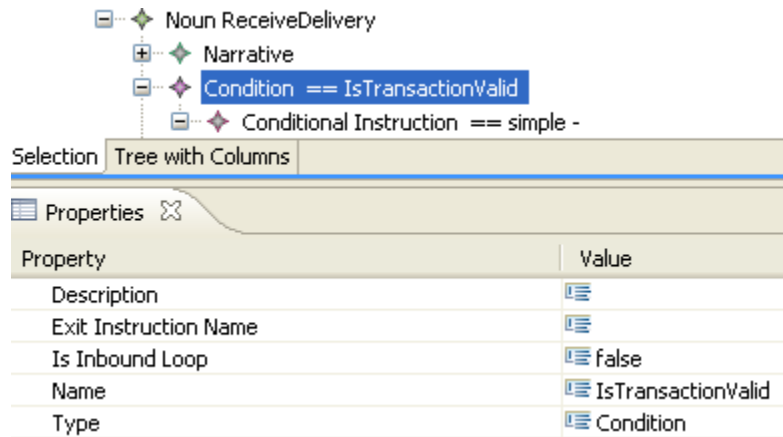
3   From the Condition node add a new child, Conditional Instruction. Adding this node allows you to
     add If Condition nodes that are used for expression evaluation. Use If Condition nodes to
     evaluate values in the Inbound message.

4   To add an expression for the If Condition, double click on the node to open the Expression
     Builder.

5   Set the ConditionType in the property view for the If Condition to `If`.

6   Build the Expression in the Expression Value and click **OK** to set the expression property in the
     property view for the If Condition node.

     In the example, shown below the expression is comparing the value of the type attribute
     assigned to xpath element ReceiveDeliveryHeader.DocumentReference@type to
     MaintenanceOrder. If the expression evaluates to `true` all chlld nodes assigned to this If
     Condition are executed.
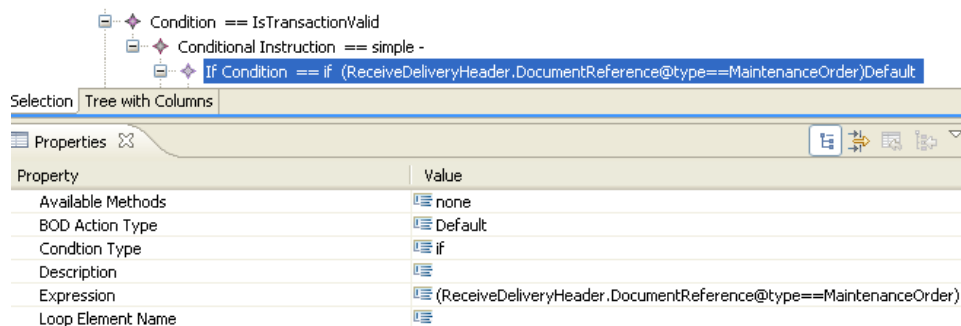
7   To add Instructions to the If Condition, create new Child Instruction Name nodes. The instruction
     names are used to invoke Instructions defined within the project. For example, three Instruction
     Name nodes have been added as child nodes of the If Condition. Each Instruction Name
     references a Database Instruction that has been previously defined. When the Instruction Name
     is processed the Instruction Name that is referenced is loaded and executed.

```
⊟ ◆ Condition  == IsTransactionValid
   ⊟ ◆ Conditional Instruction  == simple -
      ⊟ ◆ If Condition  == if  (ReceiveDeliveryHeader.DocumentReference@type==MaintenanceOrder)Default
         ├─ ◆ Instruction Name  == GetTransactionType
         ├─ ◆ Instruction Name  == GetWarehouseLocation
         └─ ◆ Instruction Name  == GetItemInfo
```
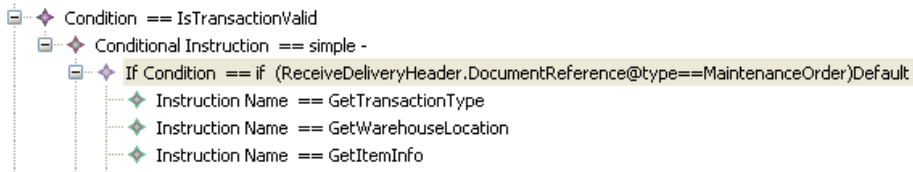
In this example, the Instruction Name points to the Instruction that was previously defined as shown below. This instruction executes an SQL statement:

```
⊟ ◆ Instruction  == GetTransactionType
   ⊟ ◆ Database GetTransactionType
      ⊟ ◆ Database SQL Statements
         └─ ◆ Statement SELECT SUBSTR(DATA,1,1) AS MROFLAG, SUBSTR(DATA,2,1) AS ITEMTY
```
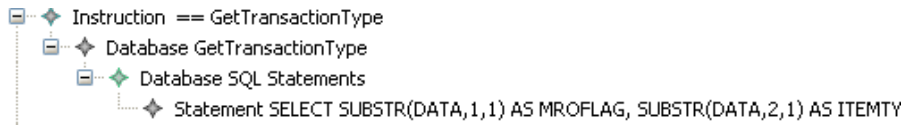
A Conditional Instruction may contain one or more If Conditions as shown below.

```
⊟ ◆ Condition  == IsTransactionValid
   ⊟ ◆ Conditional Instruction  == simple -
      ⊞ ◆ If Condition  == if  (ReceiveDeliveryHeader.DocumentReference@type==MaintenanceOrder)Default
      ⊞ ◆ If Condition  == elseif  (ReceiveDeliveryHeader.DocumentReference@type==PurchaseOrder)Default
      ⊞ ◆ If Condition  == elseif  (ReceiveDeliveryHeader.DocumentReference@type==ProductionOrder)Default
      ⊞ ◆ If Condition  == elseif  (ReceiveDeliveryHeader.DocumentReference@type==Transfer)Default
```

Each If Condition is evaluated and the first If Condition that evaluates true loads the instructions contained in that If Condition

If Condition nodes may contain child Work Element nodes. Work Elements are used to add information into the original inbound message. Work elements are generally required for a transaction to process successfully.

See Chapter 2 for the properties available for Work Element nodes.

Work element example

The following topics provide examples of how to use work elements.

See Chapter 2 for the description of the properties available for the Work Element node.

Example 1

In this example, the ReceiveDeliveryInbound project contains an entry point Condition instruction named IsTransactionValid. The Condition has a child Conditional Instruction that has many If Condition instructions. One of the If Condition expressions examines the value of the DocumentReference attribute. If the attribute type is set to ProductionOrder then a set of Instructions are executed. In this example, the If Condition shows the instructions that are performed if the else-if condition evaluates to true.
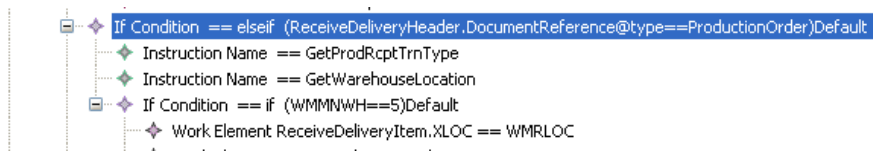
The If Condition type is set to **elseif** and the Expression compares the value of the Document Reference type attribute. When an expression is comparing a value to the attribute of an element the @ character must prefix the name of the attribute.

In this example, ReceiveDeliveryHeader.DocumentReference@type retrieves the value of the attribute from the inbound message. If the expression is true, then two Instruction Name instructions are executed. Both instructions are references to Database instructions that retrieve data from the database. After retrieving the data an If Condition compares the value retrieved in field WMMNWH to 5. If the expression evaluates to true a Work Element is added that inserts a value for location into the Inbound Message.

The Work Element property view is shown in Chapter 2. The property view indicates that the Value assigned to the Work Element will be the value retrieved in field WMRLOC from a database instruction. The Variable Type is set to database to indicate where to retrieve the value. The Xpath Element indicates the XLOC element is added as a child of element ReceiveDeliveryItem. In this example, the Set Message is true so XLOC is added as a child of element ReceiveDeliveryItem (`<ReceiveDeliveryItem><XLOC>WMRLOC</XLOC></ReceiveDeliveryItem>`)

To assign the value of the Work Element to a screen field, set the Element Name in the Screen Field Mapping property view to the XPathElement value set in the Work Element.

In this example, the value for Element Name is the same as the XpathElement, in this case, **ReceiveDeliveryItem.XLOC**.
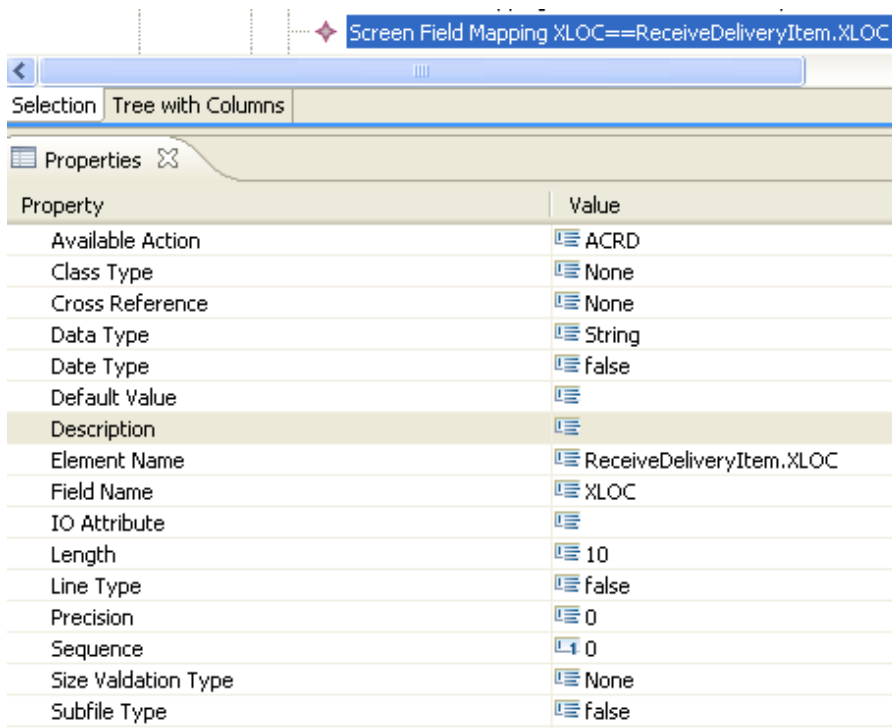
Example 2

In this example, a Substring Field instruction is added as a child of the Work Element. The value assigned to the work element will be the result of fetching a sub-stringed value from an element in the inbound message. This example uses the If Condition that was added in Example 1 and adds these elements:

Element	Description
If Condition	==if (WMMNWH==5_Default)
Work Element Parent	ReceiveDeliveryHeader
Work Element Child Element	PickNumber
Variable Type	Inbound
Value	

The Value for the work element is set by adding a Substring Field instruction as a child node. The following screen shows the setup for this example:
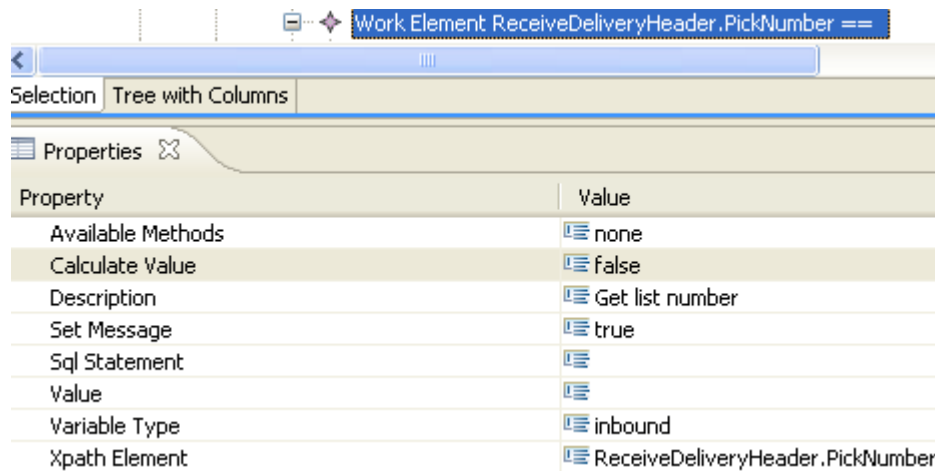
Select the Work Element, right click, and from the menu add new child Substring Field. The Substring Field properties are defined in the Properties for the Substring Field Node section of this chapter.

In the property view for the Substring Field set the complete xpath to the Element in the inbound message to substring. In this example, we want to use the value assigned to the element shown below. In this element we are specifically saying the value we are going to assign to Work Element PickNumber is the substringed value of element ID that has an attribute schemeName of ProductionOrder:

```
ReceiverDeliveryHeader.DocumentReference.DocumentID.ID@schemeName=ProductionO
rder
```

The Start Position in the Substring Field property view is set to 8. This is the position where the substring starts. The Number of Characters is 0. When the Number of Characters is 0 the instruction will include all characters following the StartPosition. For example, if the value is 12345678ABCDEFGH the value assigned to `<PickNumber>` is ABCDEFGH.
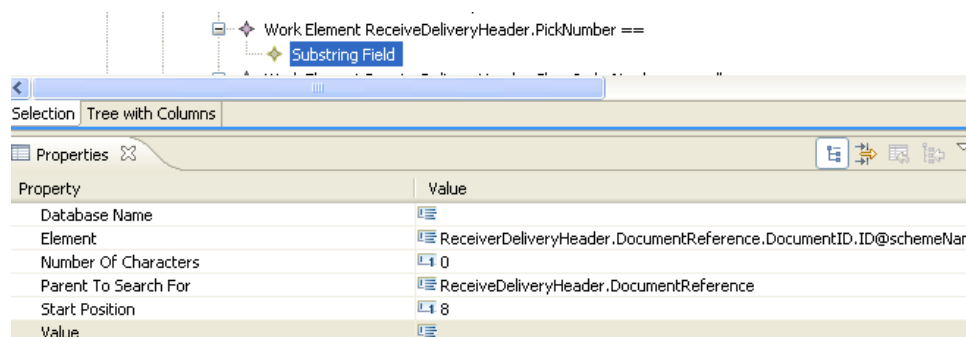
Because the Work Element adds the element into the inbound message the inbound message will contain `<ReceiveDeliveryHeader><PickNumber>` `ABCDEFGH</PickNumber></ReceiveDeliveryHeader>`

To map the Work Element to a Screen Mapping Field, set the Element property in the Screen Mapping Field to the Xpath Element property of the Work Element as shown below.

Substring handling for EX 2.2.023 and above

An example of using substring processing where an inbound message has an element that can only be 8 characters long and you need to validate whether a value having more than 8 characters was sent. For example:

1   The inbound message has element <Animal>Elephant</Animal> (index 0-7) and the database can only handle an 8 character animal type.

2   To validate the size of the data is 8, define a new work element and set the Xpath to be animaltype. Then add a child Substring Field to the work element, define the Element Animal in the substring property and set the start position to 8 and end position to 0. The end position of 0 is special processing by the runtime. When this is set to 0 it returns the substring from the start position and all characters that follow. For example, if you want to substring "unhappy" using start index of 2 and end of 0, the value would be "happy".

3   The runtime has been changed to automatically check to see if the element is in the inbound message.

4   If the element is found, it fetches the value from the inbound message (Elephant).

5   The runtime checks the length of the value returned, which is 8 in this example of <Animal>Elephant</Animal>.

6   In our example, the PI is expecting the start position to be at position 8 in the string, but there is no value at this position. The last character is at position 7. The runtime determines that the length of the string is less than the start + 1 (8 < 9).

7   The runtime previously returned a blank value if the data sent was greater than the length of the string. After this patch is installed, the runtime will return a null value in this scenario which means the animaltype work element is no longer added into the inbound message.

8   In order to map this value since it is not greater than the desired 8 characters, the PI developer must first check to see if the animaltype work element does not exist. If this is true, then a work element can be used to define the value from the inbound message. For example, add child work element using the value <Animal>, and Xpath animaltype, then map animaltype in the Screen Mapping field.

9   In the case where the element is not in the inbound message (either no <Animal> tag or an empty tag such as <Animal></Animal>, the runtime will not add the work element into the inbound message whereas in the past a blank value was assigned. Only elements that are in the inbound message are available to map to an LX field.

Batch program instruction

You can add instructions to the project to execute a legacy application at runtime. To use the legacy application:

1   In the Designer view tree, add a Batch Program node as a child of an Instruction node.

2   To map API fields to Variables, add API Field Mapping nodes as child nodes of the Batch Program.

This section defines the property page for all nodes needed to create a Batch Program instruction into a PI.

Mapping API fields to variables

To add a Batch Program Instruction:

1   Table A indicates the Batch Program node is a child to an Instruction node. To create a Batch Program instruction, add an Instruction node as a child of the Noun node.

2   Set the Name property for the Instruction to GetLocation.
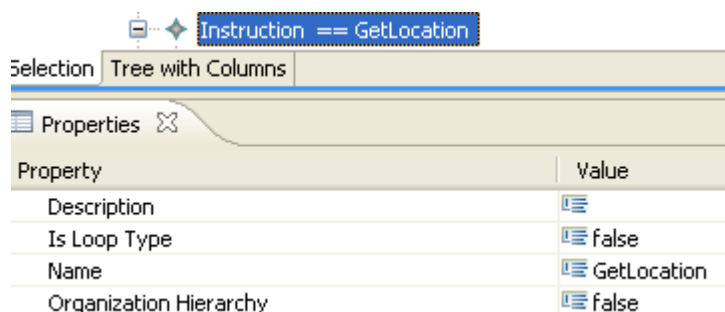
3 Select the Instruction, right click, and select new child Batch Program.

   a Set the **Name** to be the name of the legacy application. In this example, we are running `SYS830B2`.

   b Set the Action to the method defined in the definition of the API (see Appendix A). In this example, the Action is set to `Add` because when the API was defined that is the method that was defined.
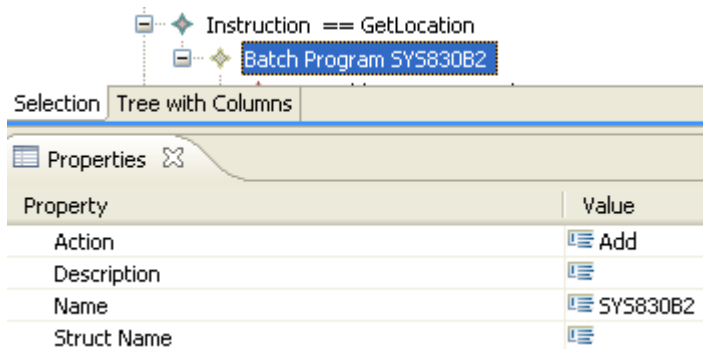
4 Select the Batch Program, right click, and choose **API Field Mapping**. The properties for the API Field Mapping are defined in Chapter 2.

5 In the property view for the API Field we want to pass a value from the Inbound message. Set the values for the API.

   a Set the **API Field** to the name given to the API field when the API process instruction was created. In this example, this field was named Warehouse.

   b Set the **Variable Type** to inbound because the value that is mapped to the API Field is from the inbound message.

   c Set the **Variable to the Xpath** of the element in the Inbound message whose value will be inserted into the Warehouse field. In this example, the value is extracted from element ReceiveDeliveryHeader.WarehouseLocation.ID.
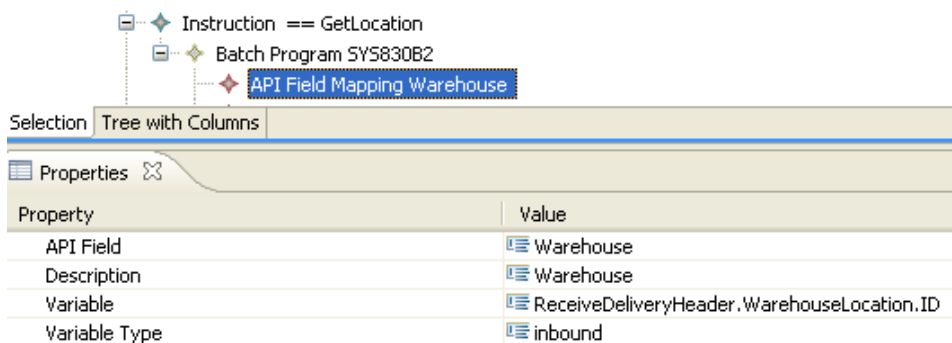
6 Add a second API Field Mapping child and set the properties in the property view.

   a In this example, WarehouseType was defined as the API Field in the API definition.

   b Map a variable 5.

   c Define the variable type as constant.
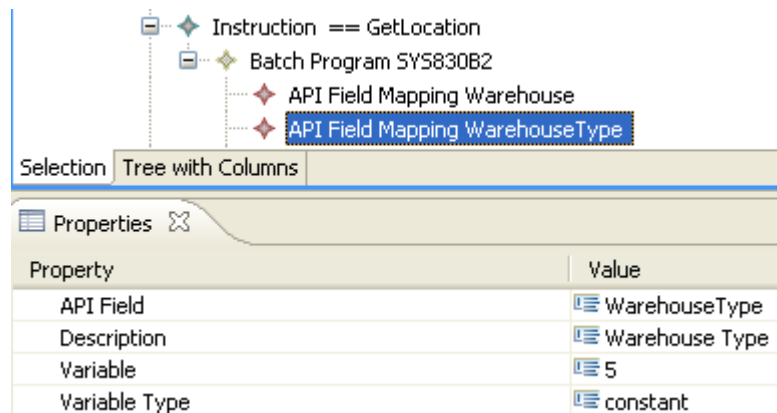
7   Add a third API Field Mapping and set only the API Field name. By setting no Variable and Variable Type of none means that this API Field (WarehouseLocation) is not mapped to a value.
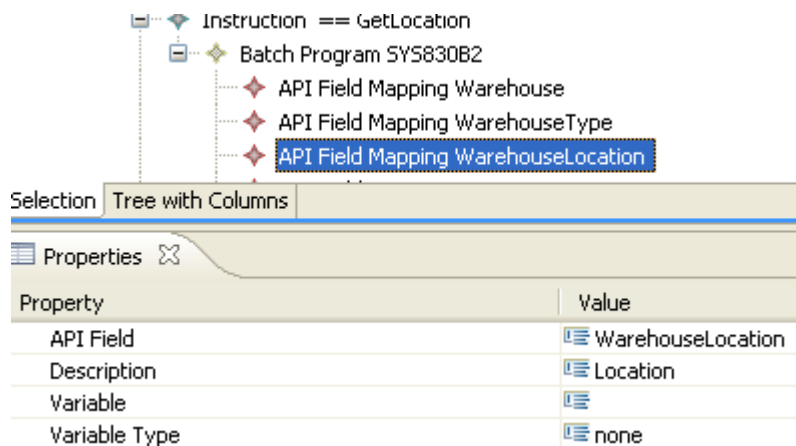
8   Add a fourth API Field Mapping for Return. Set the properties in the property view. Do not map the API Field to a variable.
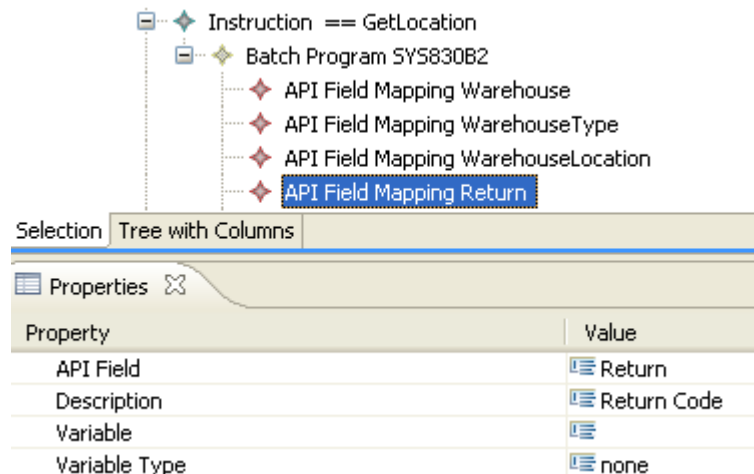
9   Add a fifth API Field Mapping for Location Type. Set the properties in the property view. Set the LocationType to Y.
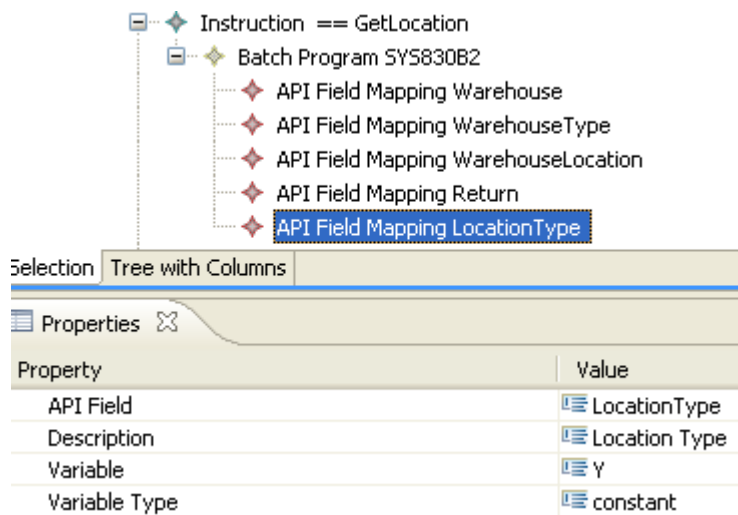
This completes mapping the API Instruction to Variables.

Referencing the Instruction for execution

The Batch Program instruction must be referenced so that it can be executed. In this example, the Instruction GetLocation will be referenced in an If Condition defined in the entry point Condition. In the screen below property Available Methods is set to Exist and the Expression is set to ReceiveDeliveryItem.StatusReasonCode. If the element StatusReasonCode exists in the inbound message, then reference the GetLocation API instruction.
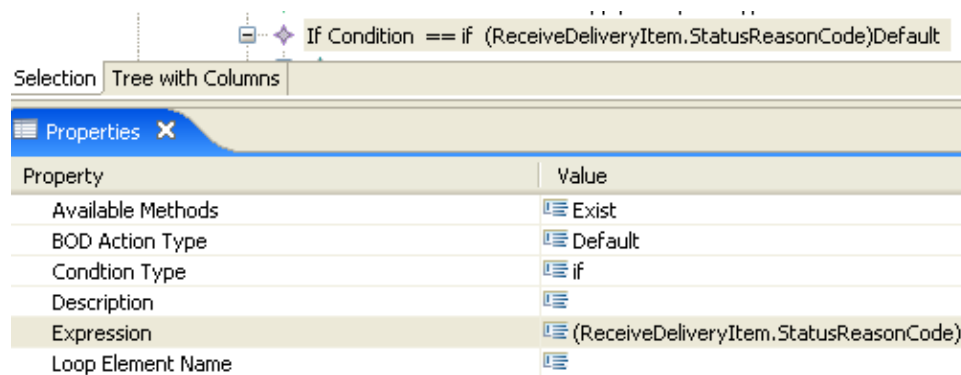
1   Select the If Condition, right click, and select new child API Instruction. In the property view for the API Instruction set the Name to **GetLocation**. Setting this name loads the Get Location Instruction defined earlier.
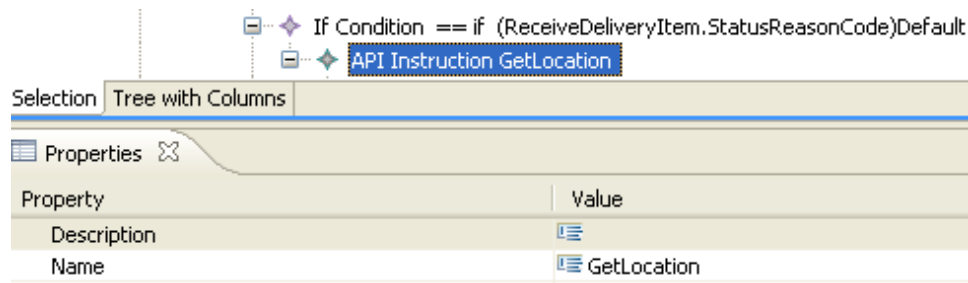
For this example, API Fields LocationType and Warehouse defined in the GetLocation Batch Program Instruction will be overridden under certain conditions. To override a variable defined in the Batch Program Instruction GetLocation Instruction, select the API Instruction, right click, and select new child Field. The properties for the Field are defined in Chapter 2.

2 Open the property view for the Field. Set the Name for the Field to **N** and set the Variable Type to **constant**.
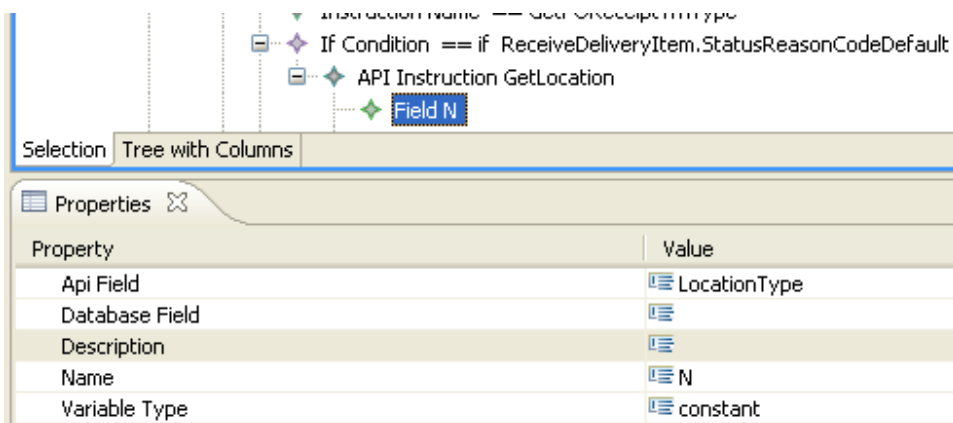
3 Add a second Field to the API Instruction to overlay the Warehouse.

    a In the Field property view set the **API Field** to **Warehouse**.

    b Set the **Name** to **ReceiveDeliveryHeader.WarehouseLocation.ID**.

    c This value for **Name** is the xpath value from the Inbound message so set the **Variable Type** to **Inbound**.
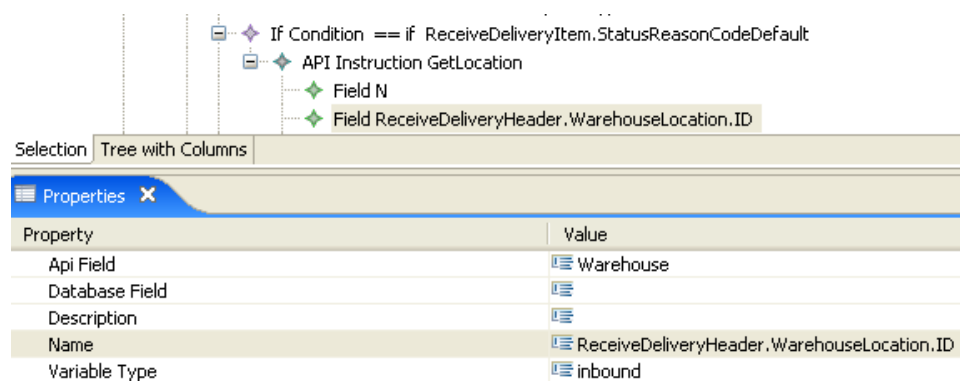
4   When the If Condition is true the API Instruction loads the Batch Program Instruction named GoodLocation. The value for Warehouse and Location Type defined in the Batch Program instruction are overridden by the values in the Field Instructions.

Retrieving a value from the API call

For this example, we want to retrieve a value returned from the API call. To retrieve values, use the Work Element.

1   For this example, add a Work Element as a child of the API Instruction.

2   In the Work Element property view, set Value to be the Batch Program field whose value we want to retrieve.

3   Set the Variable Type property to **API Field**.

4   To map this value to a field on the green screen, set the XPath Element to the element that will be added into the inbound message.

5   Set the Set Message property to **True**.

This instruction reads field WarehouseLocation from the API result set and adds a new Element named WarehouseLocation, sets the value to the API value retrieved from the result set, and then adds the new Element as a child of the ReceiveDeliveryItem. Setting the Work Element into the inbound message allows the developer to map a Screen Mapping.

Loop elements

Table A indicates that a Loop Element node is a child of the Conditional Instruction Node. Add a Loop Element node to process child elements of a BOD message. For example your Model Object may need to process a PurchaseOrder that contains a PurchaseOrderHeader containing many

Notes. Use the Loop Element to process each Note contained in the PurchaseOrderHeader. The actual processing of the note may include an instruction that adds the Note into an LX file. See Chapter 2 for properties available to the Loop Element node.

These sections provide examples of using the Loop Element node:

- Using the For Each Property to process children
- Using Loop Element in a Conditional Instruction
- Using Loop Element in a Condition Instruction

Using the For Each property to process children

In this example, assume we are creating a Model Object tree view for the PurchaseOrder Noun. These are the requirements for this project:

- The generated PI must include instructions that process all Notes that are contained within a PurchaseOrderHeader element of an incoming PurchaseOrder BOD message.
- Process each note using an RPG API.
- Do not insert empty Notes using the API.
- A PurchaseOrderHeader may contain many Note elements.

This example shows how to add nodes into the PurchaseOrder Model Object tree view that meets the requirements listed above.

To determine which nodes need to be added to our tree, look at each requirement.

- We need to add a node that can process all Note elements contained in a PurchaseOrderHeader. Reviewing the node descriptions in Chapter 2 we see that the Loop Element note will allow us to meet this requirement.
- Since empty Note elements cannot be inserted we need to evaluate the value of each Note. Chapter 2 indicates that the If Condition node lets us evaluate a value.
- We need to call an RPG API to process a Note. Chapter 2 shows that a Batch Program node allows us this.
- Since we will invoke the processing of nodes from an Instruction Name defined in another instruction we need an Instruction node that will process the Note.

Creating the instruction to process the Note

To create the instruction to process the Note:

1 Select the Noun node and add New Child Instruction.

2 Select the Instruction node just added and in the property page set the Name to **DOHEADERNOTE**.

3 Add a Loop Element node to process each Note in the PurchaseOrderHeader. Table A shows that the Loop Element is a child of the Conditional Instruction but not the Instruction. You need to create a New Child Conditional Instruction of the Instruction node.

4 Select the Instruction, right-click,and select New Child **Conditional Instruction**.
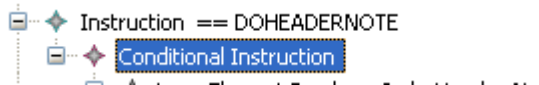
5 Select the Conditional Instruction and add New Child **Loop Element**.

6 Select the Loop Element Child and set two of the properties.

a Set the **For Each Element** to the name of the Element to process. In this example, it is **Note**.

b Set the Loop Element to be the Xpath to the Note. In this example, that value is **PurchaseOrderHeader.Note**.

7 Do not set the other properties. The For Each Element instruction will process each Note. This screen shows the property page for the Loop Element:

Property	Value
Available Methods	none
For Each Element	Note
Loop Element	PurchaseOrderHeader.Note
Loop Element Reference	none
Make Subfile Element	false
Remove Loop Element	false
Search Loop Element	false

Because processing a Note requires conditional logic and a Batch API, create another Instruction that contains nodes that do this processing. Add a new Instruction node and a new Instruction Name node. The Instruction Name will be added as a child in the DOHEADERNOTE instruction.

8 Select the Loop Element, right click and select New Child **Instruction Name**.

9 Select the Instruction Name to set the property Name in the property page. The name is the name given to the Instruction node that we will add next. This new instruction will contain nodes that evaluate the Note and invoke an API call. In this example, we will set the Name to be **SetNote**. The property page for the Instruction Name that was added is shown below.
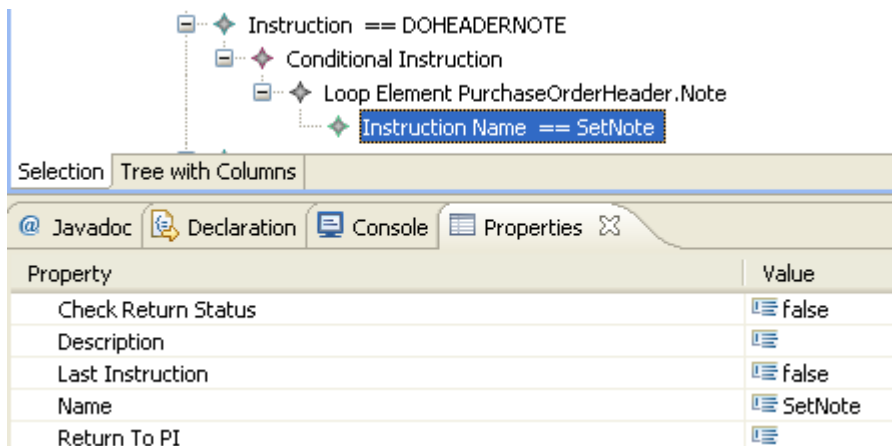
10 We added an Instruction Name node with name `SetNote`; we must create an Instruction node having a Name of SetNote.

Evaluating the Note and executing the API

To create the Instruction that evaluates the Note and executes the API:

1 Select the Noun, right click and select **New Child Instruction**.

2 Select the Instruction node that was just added and set the Name to `SetNote`. The picture below shows the new Instruction.
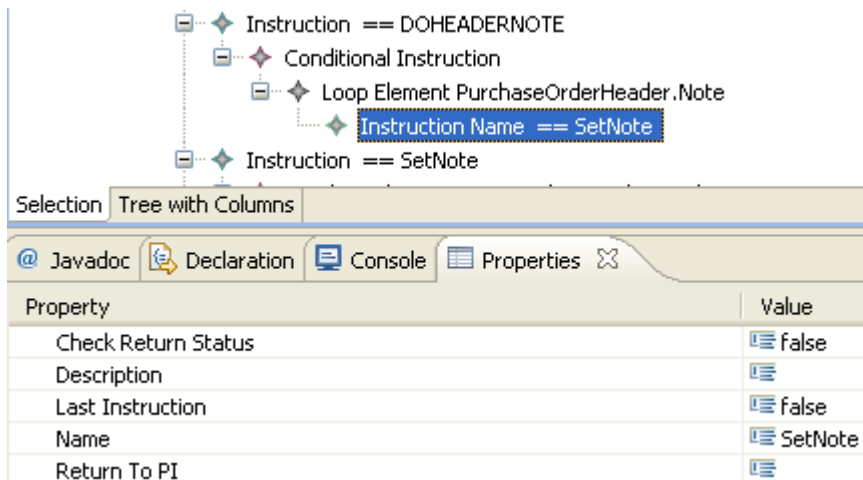
The SetNote instruction is used to process a Note using an API. The instruction should process only non-empty Notes from the BOD message. For this we look in Chapter 2 and see that the If Condition node can be added to evaluate a value for Note and can make a decision based on its value. However, Table A shows that an Instruction node does not have an If Condition as a child. Looking through Table A we find that adding a child Conditional Instruction to the Instruction node allows us to add an If Condition child node. So in this case we need to add a Conditional Instruction node so that we can add our If Condition node.

3   Select the SetNote Instruction, right click, and select New Child **Conditional Instruction**. Select the Conditional Instruction node to open the property page. You may set the property Element Name to PurchaseOrderHeader for clarity but since we are executing this instruction from the DOHEADERNOTE using the For Each Element property of the Loop Element, it is not used. The picture shown below shows the property page setting for our Conditional Instruction. See Chapter 2 for the properties available to this node.
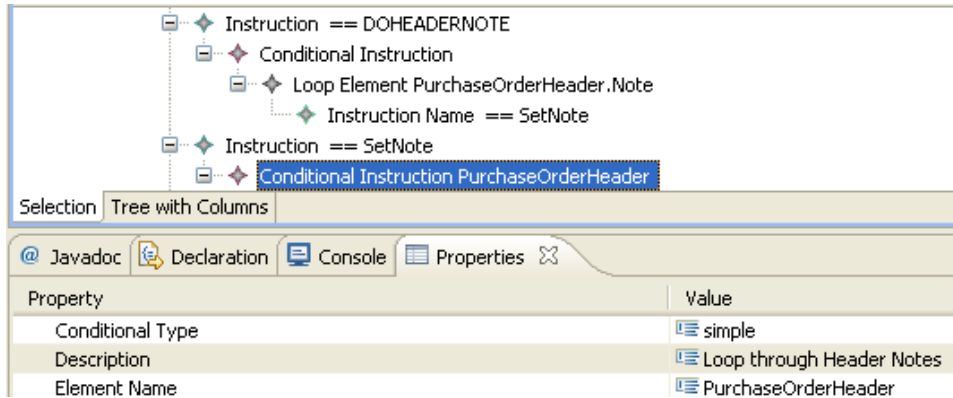
4   Evaluate the Note. Select the Conditional Instruction node, right click then select New Child **If Condition**.

See Chapter 2 for the properties available to the If Condition node. Select the If Condition node to open the property page and set the Expression. In this example, we want to check the value of the Note. We do this by setting the Expression in the property page to (PurchaseOrder.Note). See Chapter 1 on how to add an Expression using the Expression Builder view. Set the Available Method property by selecting `IsEmpty` from the list. This checks to see if the Note has a value. Set the Condition Type to `if` because we want to execute a java if condition using the generated PI at runtime. The picture below shows the property page for the If Condition that was added.
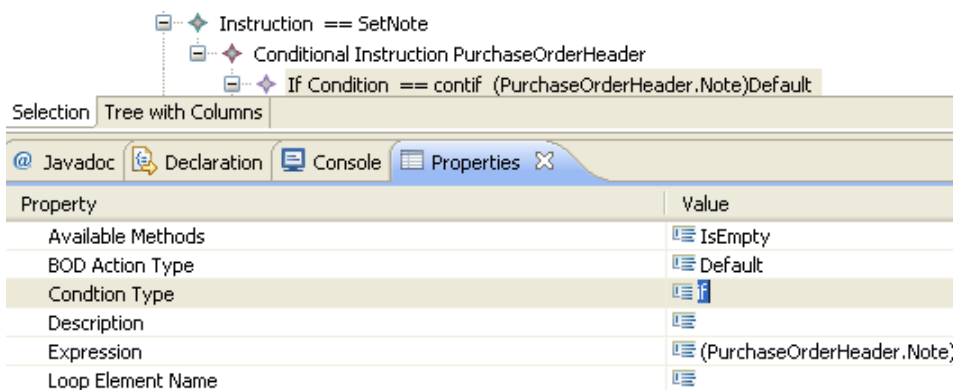
At runtime the PI that is generated will return true if the Note is empty or false if not. To handle the false case requires adding another If Condition but this time set the Condition Type in the property page to else. When an if Condition sets the Condition Type to else there can be no Expression set.

5   Select the Conditional Instruction node and add New Child **If Condition**. Select the node to open the property page. Set the Condition Type to `else`. The picture below shows the property page for our second If Condition.
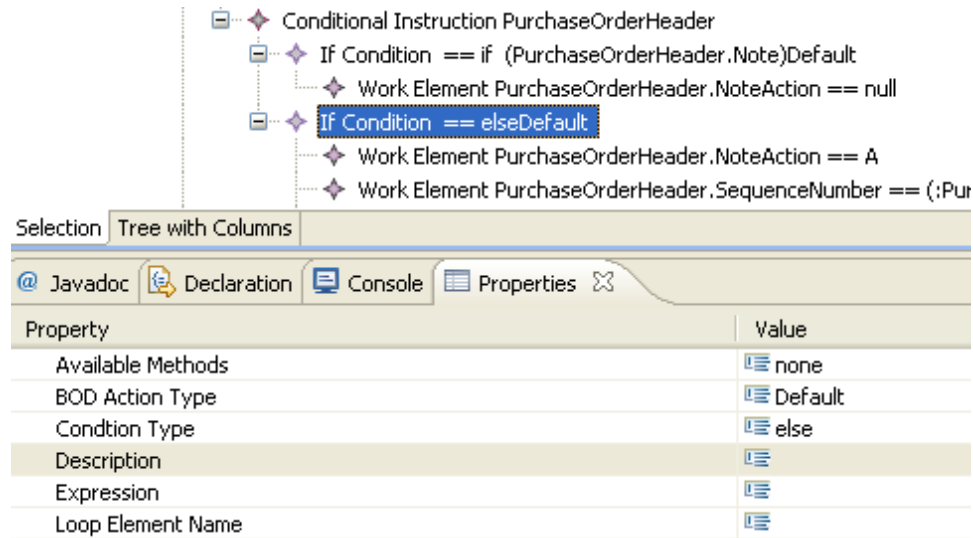
Each If Condition may contain nodes that get executed depending on the results. If the evaluated expression is true then any child nodes added to that If Condition will be executed, otherwise the child nodes of the else If Condition node are executed. In this example, Work Element nodes are added as children of both If Condition nodes.

6   In the case the expression evaluates to true, a Work Element is added that does nothing. Select the first If Condition, right click and select New Child **Work Element**. See Chapter 2 for the properties available to the Work Element. Select the Work Element added to open the property page. The property page is shown below and shows that a new element called NoteAction is set but it is never added into the BOD Message since Set Message is false.
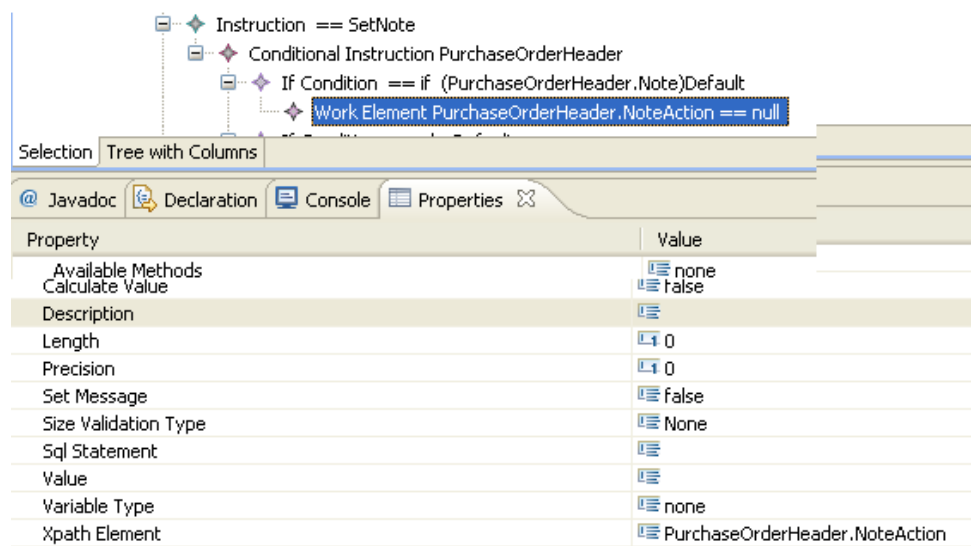
7   Create Work Elements for the case that the expression evaluates to false. This is the case that will process the note so Work Elements are added into the BOD message and used by the API definition.

8   Select the **else If** Condition node and add Work Element child nodes. The first Work Element will set a constant value to an element added to the message. The element that is added is called **NoteAction** and it is added into the PurchaseOrderHeader of the BOD message. The properties for this Work Element are shown in the picture below.

9   The second Work Element property page is shown below. It uses an Arithmetic expression to set a value for the SequenceNumber element that is added into the BOD message.

10 In our example, there are other elements in the BOD message that need to be evaluated, therefore, we need another set of If Condition Nodes added as children of the If Condition == else node.

11 Select the `else If` Condition node, right click and select New Child **If Condition**. Select the If Condition node just added to open the property page and set the Conditional Type to `if`. Set the expression. In this example, we want to evaluate the value of an elements attribute. To evaluate an attribute of an element requires use of the @ sign. The Expression shown below evaluates the value assigned to the Note attribute named use. If the (PurchaseOrderHeader.Note@use==External) means that if the use attribute value evaluates to External we will process a set of Work Element instructions. In this example, we add a New Child Work Element to the If Condition node.
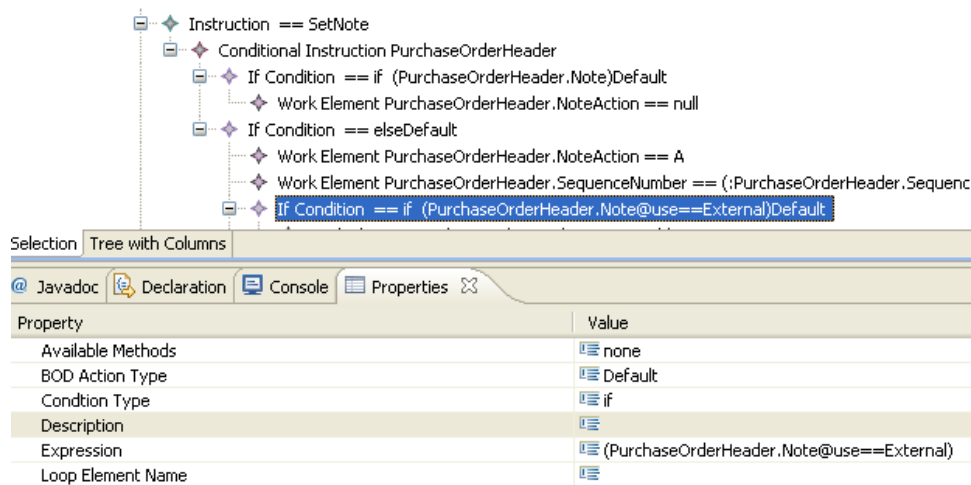
12 If the expression evaluates to true add the New Child Work Element. This adds a new element into the PurchaseOrderHeader called PrivatePublic and sets it to a constant value of `E`.
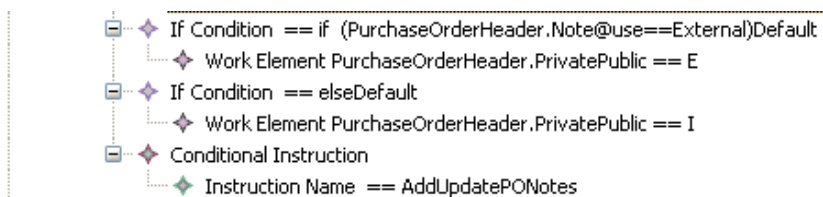
13 Add another If Condition node to process when the expression fails. Select the `else If` Condition to add another If Condition. In this case the Condition Type property is set to else therefore the expression is not set. Add a Work Element as shown in the picture above that sets the PrivatePublic to a constant of `I`.

14 At this point the SetNote instruction contains the evaluation of the Note, now we need to execute an API using the Batch Program node. Select the `else If` Condition instruction that occurs when the Note is not empty and add a New Child Conditional Instruction that is used to call our Batch Program.

15 Select the Conditional Instruction just added, right click and select new Child **Instruction Name**. Select the Instruction Name to open the property page and set the Name to `AddUpdatePONote`.

This means there must be another Instruction named **AddUpdatePONote** which executes a Batch Program node. The entire SetNote Instruction is shown below.
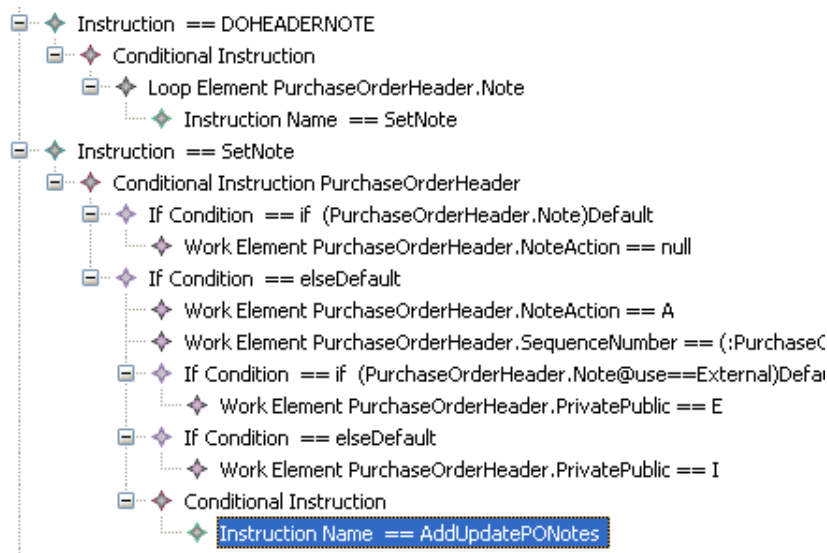
Summary

At this point we have added two instructions into the PurchaseOrder Model Object tree view. At runtime the PI generated from the Model Objects performs these functions:

- Executes the DOHEADERNOTE instruction to process each Note contained in the PurchaseOrderHeader.
- Executes the SetNote Instruction for each Note. The SetNote instruction evaluates the value of the Note. If the note is not empty new elements are added into the BOD message that are used by the API that is invoked using the Instruction Name.

Mapping BOD elements to the API

To add an instruction that maps elements contained in the BOD Message to the API:

1   Select the Noun node, right click and select New Child **Instruction**. Select the Instruction node to open the property page. Set the Name to **AddUpdatePONotes**. See Chapter 2 for the nodes available to add an API; the Batch Program node allows you to map an API.

2   Select the AddUpdatePONotes Instruction node, right click, and select New Child **Batch Program**. See Chapter 2 for a discussion of the properties of the Batch Program node.

3   Select the Batch Program node to open the property page.

4   Set the **Name** property to the name of the RPG API program. In this example, we will invoke SYS934B.

5   Select Action **Add** to add a Note into LX using SYS934B.
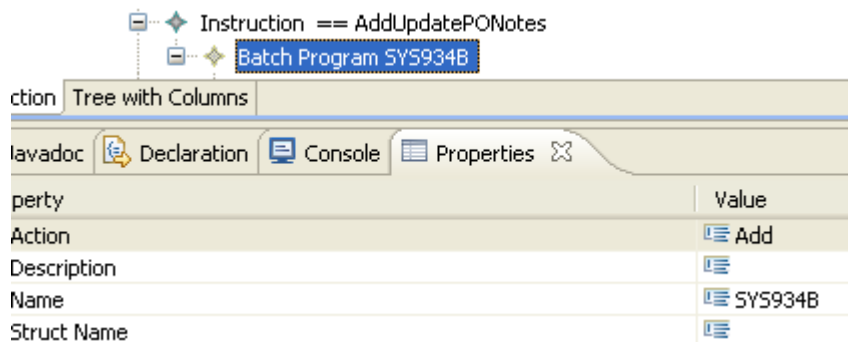
6 Map elements defined in the SYS934B generated PI to elements in the BOD message. Add an API Field Mapping for each parameter that is passed to the API. See the Properties for the API Field Mapping for a description of the available properties. This screen shows that eight parameters are mapped to the PI. Note that some of Work Elements that were added are mapped to the API.
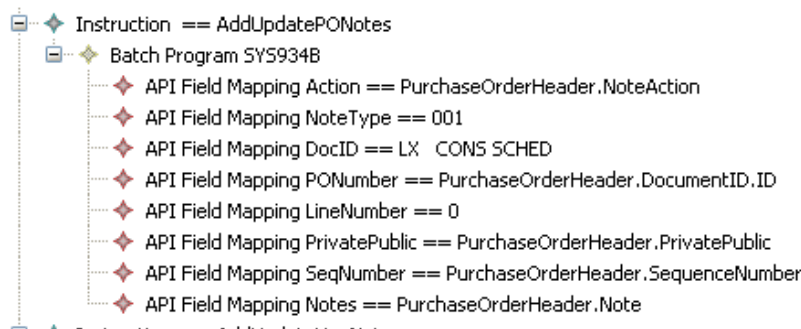
7 The picture below shows the last API Field Mapping property page. In this case the API Field Notes is defined to map to an API field in the SYS934BAdd PI that is generated from the SYS934B Model Object. The Variable Type indicates the value is retrieved from the BOD message at runtime and the Variable is the Xpath used to retrieve the value, in this case the value for the Note element.
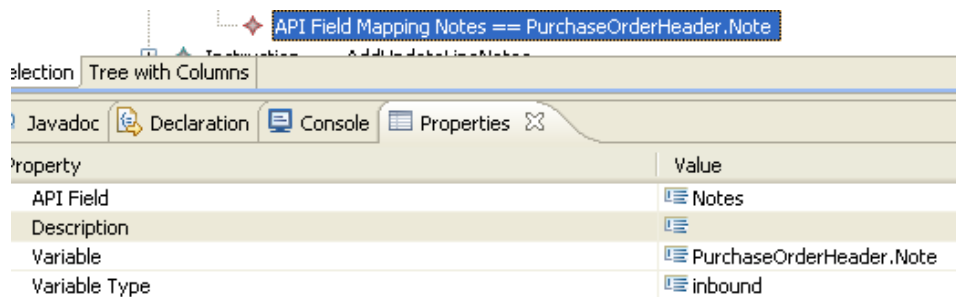
Using the Loop Element in a Conditional Instruction

In this example, we are adding a new Instruction in an existing Model Object project. The new instruction provides this functionality:

- Processes each ShipmentItem contained in a Shipment. This requires an Instruction Node that has the Is Inbound Loop property set to true.

- Requires evaluation of message data. If Condition nodes are needed. Since an Instruction node does not have an If Condition as a child a Conditional Instruction is needed to contain the If Condition nodes.

- Need to create a new child element that will be mapped to a legacy LX application. Need a Loop element to add the new element. The name of the element will be ConfirmDetail.

- Create a ConfirmDetail for each ShipmentItem. We need to set the Conditional Instruction to be a looping conditional.

See chapter 2 for a discussion of the properties available for the Loop Element.

To create an instruction that sums quantities:

1 Select the Noun and add New Child Instruction.

2 Select the Instruction node to set the properties.

3 Set the Name to `SumTheQuantities`.

4 Set the Is Inbound Loop to `True`. This screen shows the Instruction node and properties:
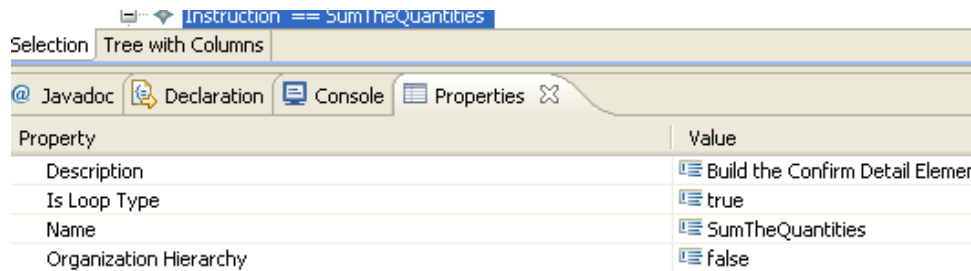
5 Add a Conditional Instruction that will be the parent of the If Condition nodes.

6 To create new elements for each ShipmentItem, set the Conditional Instruction properties for looping. Set the Conditional Type inbound and the Element Name to ShipmentItem. This instruction loops through each ShipmentItem in the BOD message. This screen shows the properties and values for the Conditional Instruction:
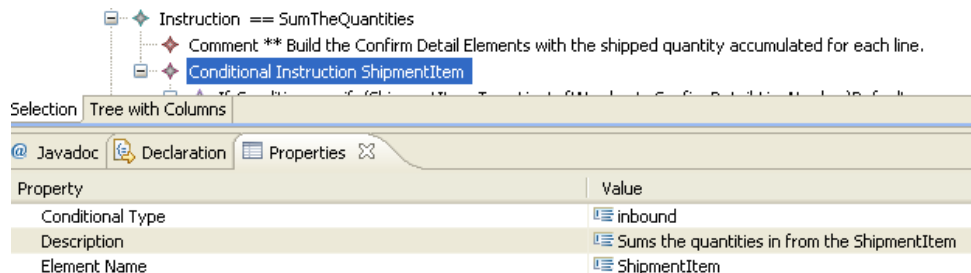
7 Add an `If` Condition to check to see if a new element is required. If the expression evaluates to true, add a Loop Element that has the Make Subfile Element set to true and set the Loop Element to the name of the element to add into the BOD message.
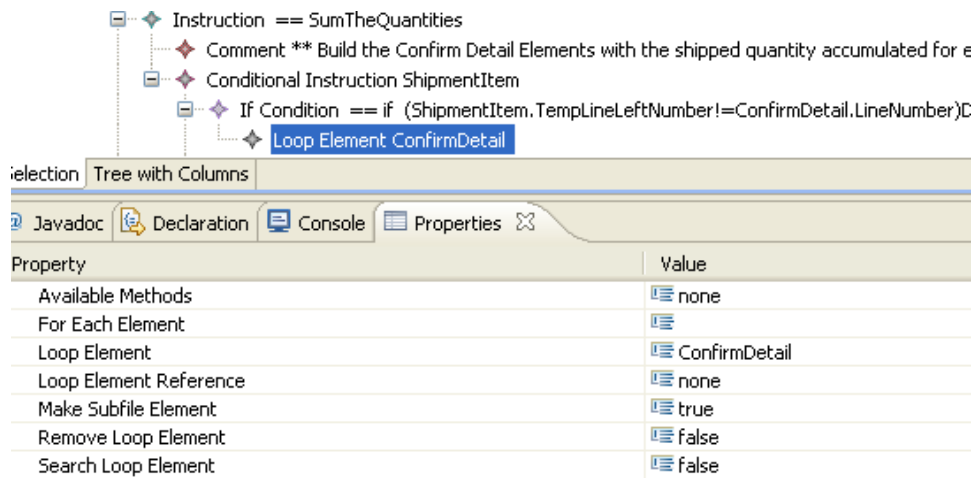
8   Add another If Condition as a child of the Conditional Instruction that evaluates the value of an Element. If it evaluates to true add child elements into the ConfirmDetail child using Work Elements. The picture below shows that Work Elements having the Set Message property set to true are added to the ConfirmDetails using values from the ShipmenItem that is currently being processed.
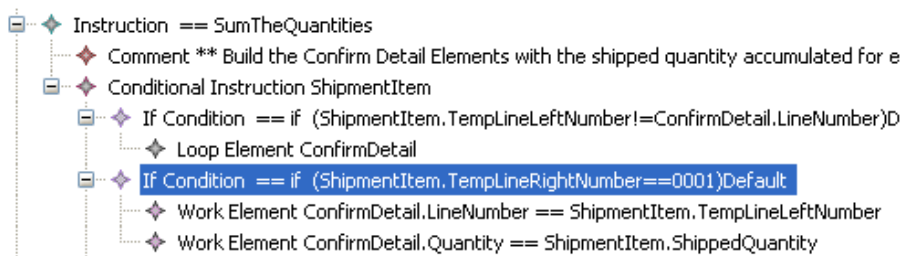
9   Add additional expressions that check data from the BOD message and use Work Element nodes to set the child element into the ConfirmDetail using data from the ShipmentItem that is being processed. In the picture below the quantity is updated depending upon the value of the TemplLineRightNumber of the current ShipmentItem.

```
☐ ◆ Instruction  == SumTheQuantities
    ◆ Comment ** Build the Confirm Detail Elements with the shipped quantity accumulated for each line.
    ☐ ◆ Conditional Instruction ShipmentItem
        ☐ ◆ If Condition  == if  (ShipmentItem.TempLineLeftNumber!=ConfirmDetail.LineNumber)Default
              ◆ Loop Element ConfirmDetail
        ☐ ◆ If Condition  == if  (ShipmentItem.TempLineRightNumber==0001)Default
              ◆ Work Element ConfirmDetail.LineNumber == ShipmentItem.TempLineLeftNumber
              ◆ Work Element ConfirmDetail.Quantity == ShipmentItem.ShippedQuantity
        ☐ ◆ If Condition  == elseDefault
              ◆ Work Element ConfirmDetail.Quantity == (:ConfirmDetail.Quantity+:ShipmentItem.ShippedQuantity)
```

Selection | Tree with Columns

@ Javadoc | Declaration | Properties ⊠

Property	Value
Length	0
Precision	0
Set Message	true
Size Validation Type	None
Sql Statement	
Value	(:ConfirmDetail.Quantity+:ShipmentItem.ShippedQuantity)
Variable Type	ArithmeticExpression
Xpath Element	ConfirmDetail.Quantity

10 This example shows how to create an Instruction that creates new elements named ConfirmDetaii.

Two child elements are added to the ConfirmDetail: Quantity and LineNumber.

For each ShipmentItem contained in the Shipment message a new ConfirmDetail is created and updated with data from the current ShipmentItem.

The new elements are mapped to an LX application in another instruction defined in the project. The picture below shows the mapping to the LX application.


```
☐ ◆ Instruction  == ProcessShipmentPickConfirmation
    ☐ ◆ Display Program  == ProcessShipmentPickConfirmation
        ☐ ◆ Action Code Replace
            ⊞ ◆ Action 1 == ORD570D1 PANEL01== ENTER
            ⊞ ◆ Action 2 == ORD570D1 PANEL02== F19
            ☐ ◆ Action 3 == ORD570D7 PANEL01== ENTER
                  ◆ Exception UM00911 == ENTER
                  ◆ Forced Value ConfirmDetail.ActionField == 11
                  ◆ Locate Row ConfirmDetail
                  ◆ Screen Field Mapping XXACT==ConfirmDetail.ActionField
                  ◆ Screen Field Mapping XXLIN==ConfirmDetail.LineNumber
                  ◆ Screen Field Mapping XXCQTY==ConfirmDetail.Quantity
```

Using a Loop Element in a Condition Instruction

This example makes the following assumptions.

- A Condition node is added that has the Is Inbound Loop property set to `True`.
- The Condition node has the Exit Instruction Name property set and has a name of LoopingCondition.

- We will create an Instruction that allows summing the quantities. The Instruction requires evaluating an expression and adding new subfile data.

- We will use the ShipmentItem as the name of the element to loop over.

- After processing is complete the Exit Instruction is executed.

To use a loop element in a Condition Instruction:

1 Select the noun and add a new Condition node.

2 Select the node to open the property page.

3 Set the **Name** to `Conditional Loop`.

4 Set the Is Inbound Loop to `True`.

5 Set the Name of the Instruction that is executed after all data is processed (LoopExitInstruction).
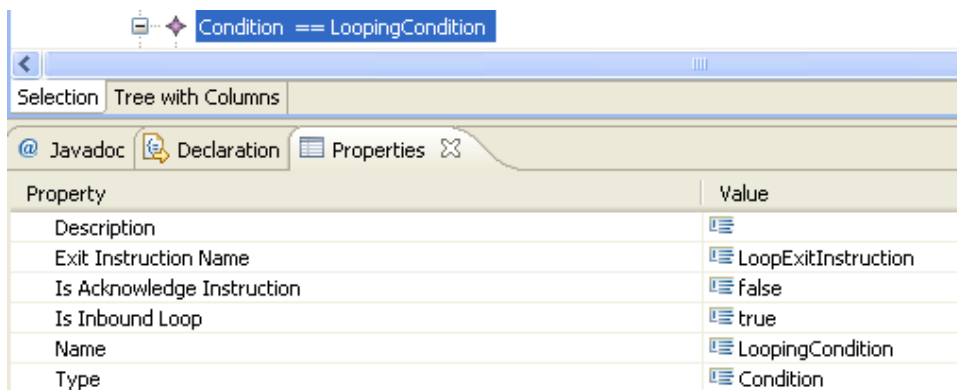
6 The caution in Chapter 2 in the Loop Element properties indicates that we must add a Loop Element as the first child of the Condition. This is used to search for our loop element, ShipmentItem as shown in this screen:
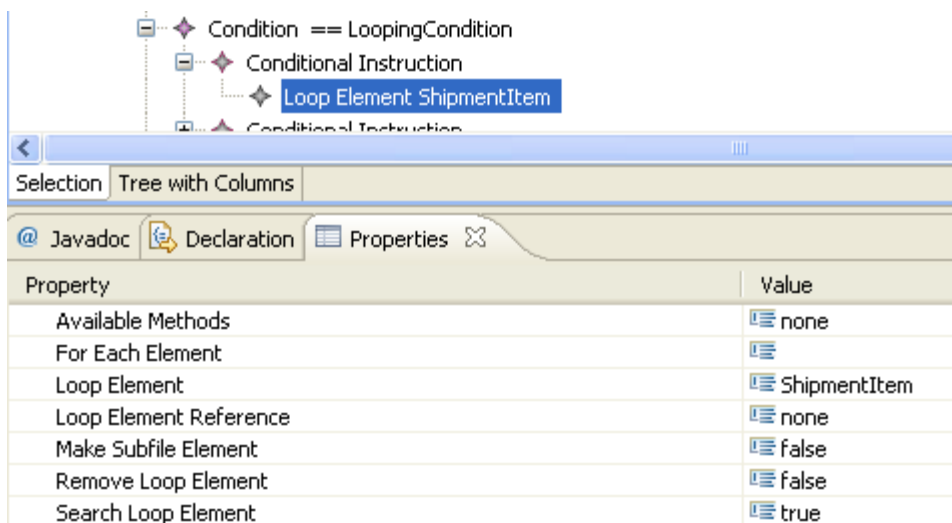
7 The Conditional node may contain many Conditional Instructions that perform various instructions. One of these Conditional Instruction nodes is used to add a loop element into our

BOD message and to remove elements from the BOD message. The instruction is shown below. Notice several Work Elements are used to update new elements into the BOD Message. An expression checks the StatusCode of the ShipmentHeaderand executes instruction SumTheQuantities if it is true. If not true, Loop Element nodes are added.
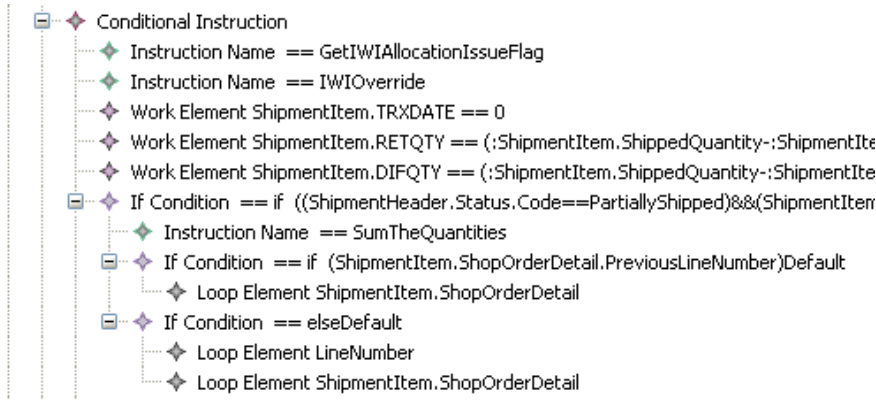
8   The picture below shows the SumTheQuantities Instruction defined as a looping Instruction by setting the Is Inbound Loop property to **True**.
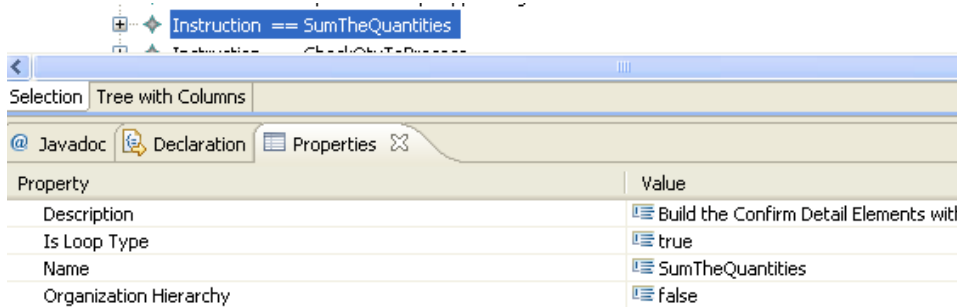
9   This instruction is similar to the SumTheQuantities in the previous example. Add a Conditional Instruction that is defined as a conditional loop by setting the Condition type property to inbound and the Element Name to Shipment Item. We want to sum the quantities for all of the Items.
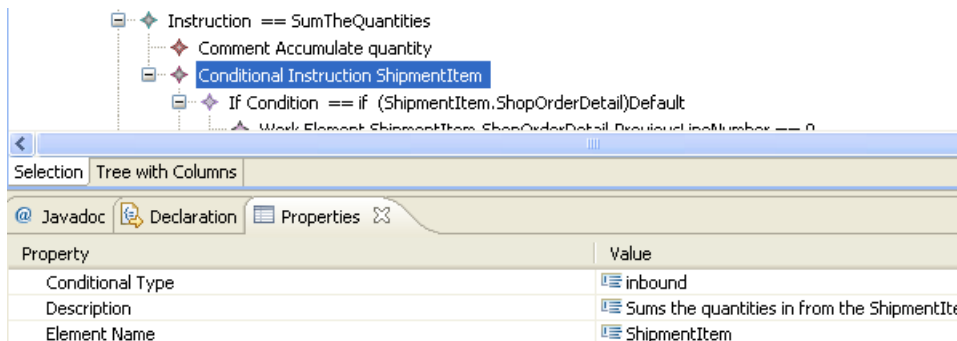
10 The picture below shows the SumTheQuantities instruction. We are updating ShipmentItem data with ShopOrderDetail by using Work Elements.
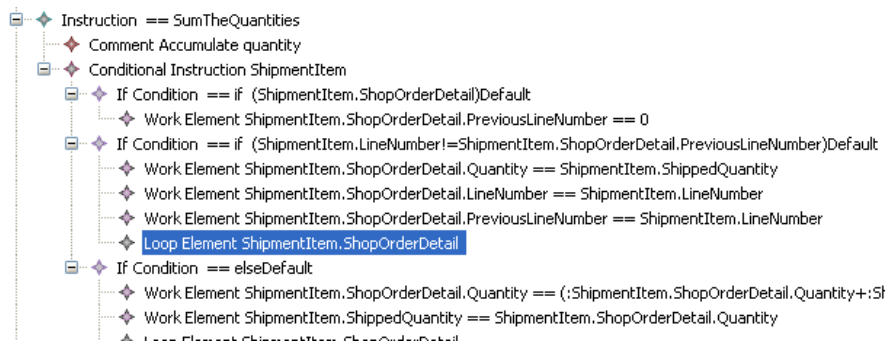
11 After executing the SumTheQuantities instruction an expression is evaluated to determine if the current ShipmentItem has a ShopOrderDetail. If not, one is created using a Loop Element by setting the Loop Element to the Xpath of the element to create (ShopOrderDetail) and setting the Make Subfile Element to `True`.

12 Looking at the else condition in the picture shown below we see that the first Loop Element is used to delete a ShipmentItem element from the BOD if the LineNumber of the current ShipmentItem is Equal to the Previous ShipmentItem LineNumber. In this case the previous ShipmentItem is deleted from the BOD message.
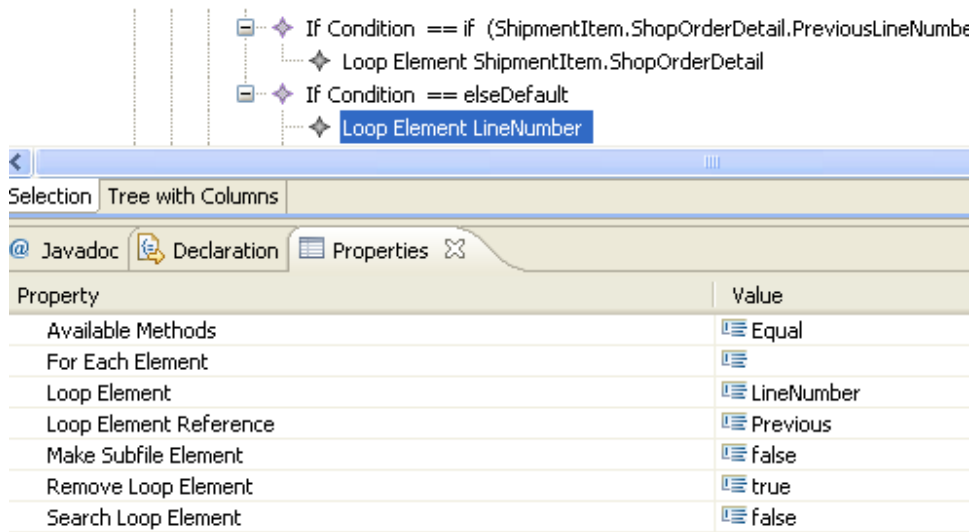
13 The next Loop Element sets the Search Loop Element to be a new element in the ShipmentItem called ShopOrderDetail.
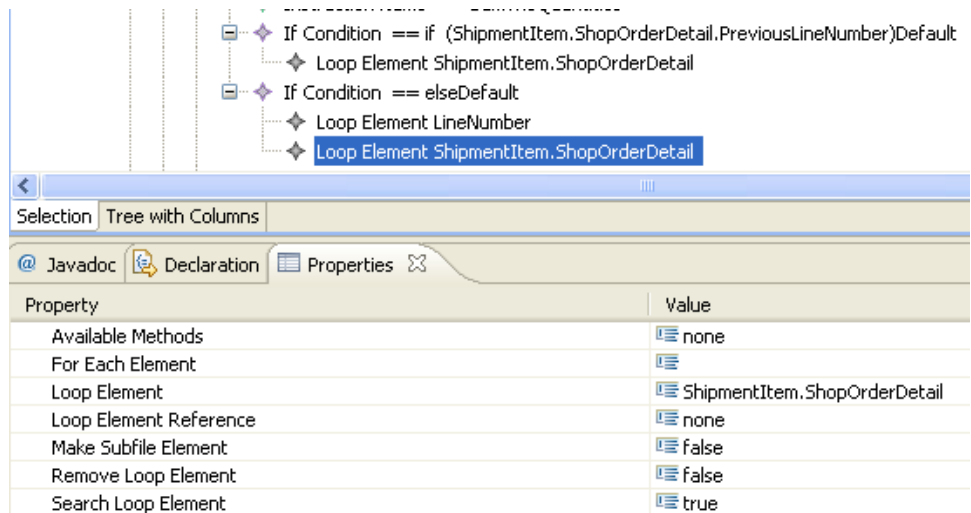
14 When all ShipmentItem nodes have been processed the Exit Instruction defined in the Condition node is executed. The exit instruction is shown in the Exit Instruction below.

Exit Instruction

The looping Condition has a property Exit Instruction Name that is a reference to an Instruction to process after all occurrences of the Loop Element have been processed. It is not a required property. The exit instruction can be used to continue processing another display program. If needed, it can process additional database retrievals or any other instruction that may have already been defined. In the screen shown below the exit instruction uses the Instruction Name to invoke other instructions.
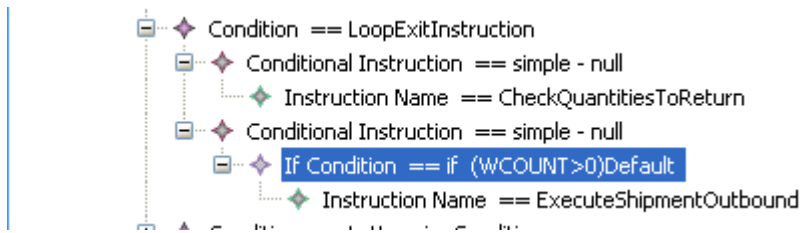
Additional inbound capabilities

This section describes additional capabilities of an inbound process instruction.

Concatenation Field

You can use Work Elements to concatenate data from an inbound message. See Chapter 2 for the properties available for the Concatenaton Field node.

The Xpath property is used to define the element that is created in the message. The Set Message property must be set to true to add the element into the current inbound message. The Variable Type must be set to Inbound when setting the inbound message. For example the Shipment contains a ShipmentItem with child LineNumber. The screen below indicates that this element will be updated.

To continue this example, add a Concatenation Field to the Work Element. Set the properties in the **Concatenation** Field. In the picture shown below the value is extracted from element ShipmentItem.TempLineLeftNumber. After extracting the value the value for the Work Element is updated by concatenating the value currently extracted from LineNumber with that extracted from the **Concatenation** Field. For example if LineNumber was 0001 and TempLineLeftNumber was _0001, then the new value assigned to LineNumber is `0001_0001`.
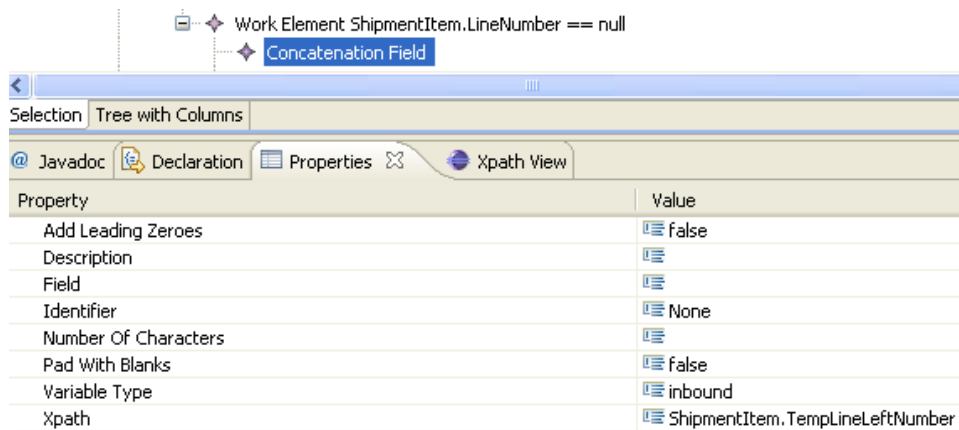
Substring Field

You can use Substring Fields to update a value set by a Work Element. See Chapter 2 for the properties available for the Substring Field node.

To set the Work Element value with a substring of another value, add a Work Element to an Instruction and then select the Work Element and add new child Substring Field. Set the properties for the Work Element. The screen below indicates that element TempLineRightNumber will be assigned the value extracted from a Substring Field.
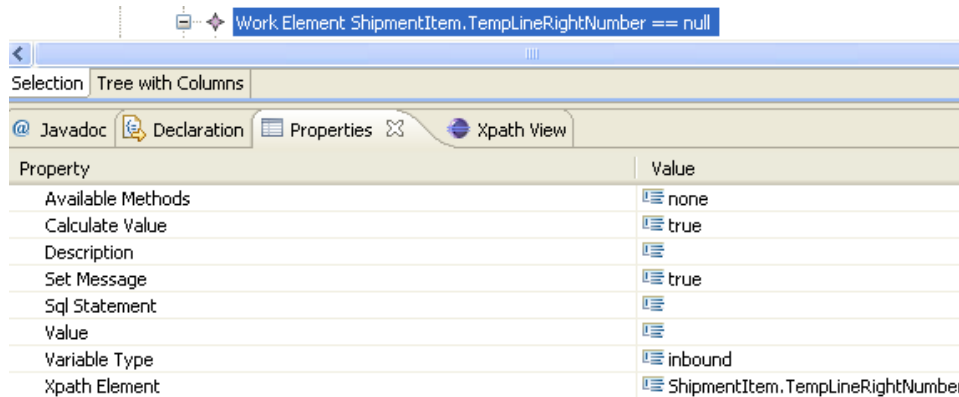
To continue this example, add a new child Substring Field to define the value that will be assigned to the Work Element. In the picture below the value is extracted from ShipmentItem.LineNumber starting in position 5. The Start Position is zero-based so if the value in Line Number is `12345_6789123`, then the extracted starts with the character after the fifth position. Because the Number of Characters is `0` the extracted value is everything after the fifth position (`_6789123`). If the Number of Characters is not `0` then this is the end position. For example, if Number of Characters is `8` then the extracted value is (`_67`). See the following screen:
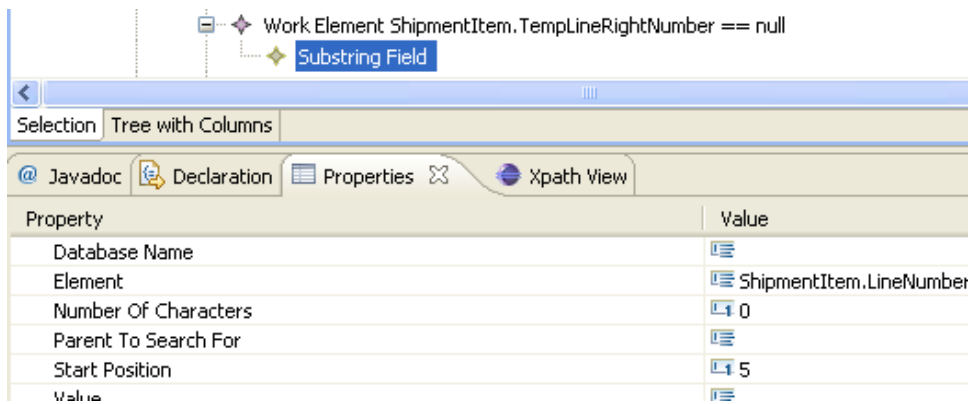
Outbound Message

An instruction can be created that allows the Inbound process instruction to create an outbound message. The Outbound Message instruction contains properties that are used to load an outbound process instruction and uses Mapping instructions to create the message passed to the outbound process instruction.

1   To create such an instruction, select the Noun node, right click, and choose new child **Instruction**.

2   Select the Instruction node, right click, and choose new child **Outbound Message Instruction**. This node allows you to add children that set the Verb for the outbound message and map elements that are the parameters used by the outbound process instruction.

See Chapter 2 for a description of the properties of the Outbound Message Instruction node.

Example Outbound Message Instruction

To create a message that is passed to an outbound process instruction:

1   Select the Instruction, right click and choose new child **Outbound Message Instruction**. Set the property Name in the property view as shown below.
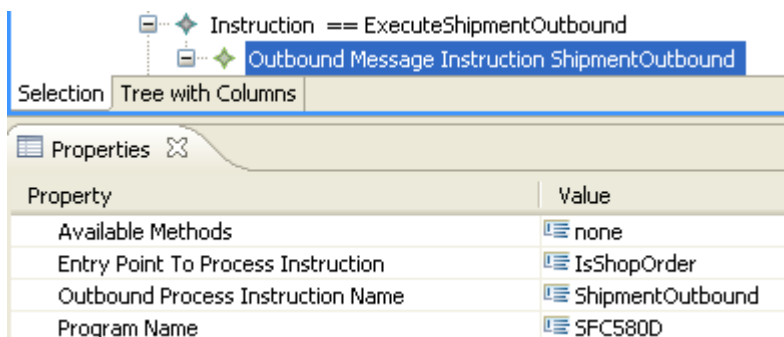
2   To define the verb for the message that is produced, select the Instruction, right click, and choose new child **Verb**

3   Select the Action Code to the event you are producing. For example, if you are Adding, select **Add**.

4   Set the Verb from the selection box to Process if LX is not the SOR for the message being produced. If LX is the SOR, set this to `Sync`.
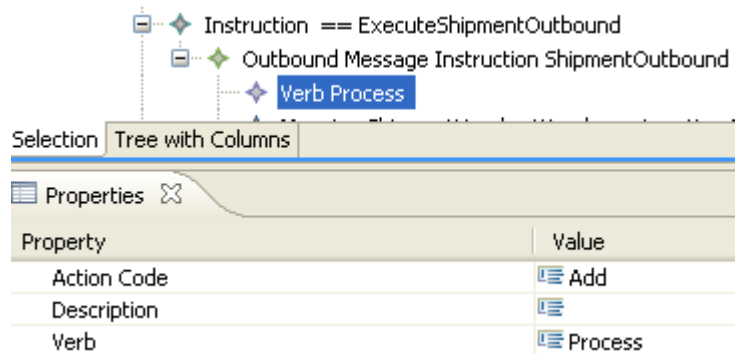
5   Select the instruction, right click, and choose new child Mapping for each value that is passed to the outbound process instruction. Map names that have been defined in the exit point process instruction that invokes this process instruction.

In this example, the exit point process instruction is `SFC580DEXIT02`. The parameters passed by the exit point are shown below. Create a mapping for each parameter.

```
<data name="EPWarehouse" type="char" length="3" usage="inherit" eventField="false" />
<data name="EPShopList" type="packed" length="5" precision="0" usage="inherit" eventField="false" />
<data name="EPOrderNbr" type="packed" length="5" precision="0" usage="inherit" eventField="false" />
<data name="FILLER" type="char" length="241" usage="inherit" eventField="false" />
```

6   Select a Mapping node and set the properties for the node. We are mapping the value from the inbound message to the name assigned in the exit point. The name assigned in the exit point is used as an element when the process instruction receives an exit point message. Set the element name to point to the value that will be assigned to the parameter
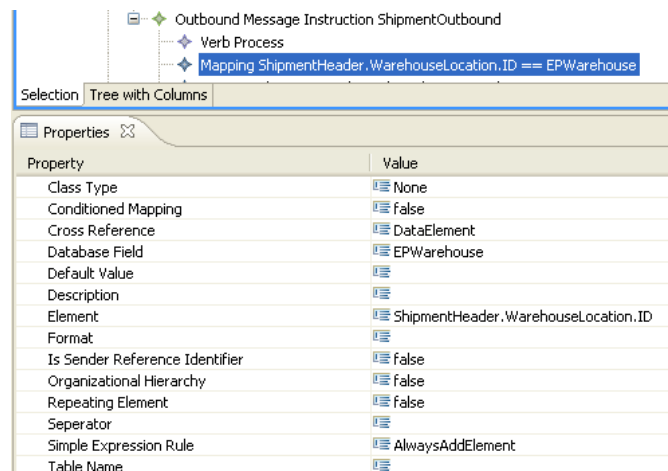
7 Add two additional Mapping child nodes and set the **Element** and **Database** Field in the property view.
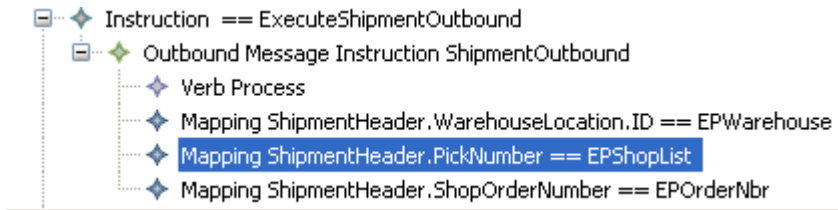
The message passed to the ShipmentOutbound contains these parameters:

<EPWarehouse>xpath value</EPWarehouse.>

<EPShopList.>xpath value</EPShopList>

<EPOrderNbr>xpath value</EPOrderNbr>

This instruction named ExecuteShipmentOutbound produces a ProcessShipment message when this instruction is referenced with an Instruction Name node.

Chapter 4   Creating outbound process instructions

This chapter describes how to create outbound process instructions with the Infor LX ION PI Builder. Information contained in this chapter describes how to create process instructions used either by the LX Extension or the LX Connector runtime code. Both the LX Extension and the LX Connector contain an Outbound Processor that is used to process the instructions.

Overview

LX uses exit points or triggers to produce outbound messages. This process is used to build the outbound BOD message:

- The exit point or trigger passes arguments to the LX event handler. The Event Handler for the LX Extension is SYS070C and that for the LX Connector is SYS071C.
- The event handler passes the arguments in the form of an xml message to the Outbound Processor.
- The Outbound processor uses information in the event to invoke an exit point process instruction which can interpret the event message.
- The exit point process instruction passes the name of the BOD process instruction to the Outbound Processor.
- The process instruction is used to build the BOD message for the event.

This chapter provides instructions to create these projects and process instructions:

- Exit point projects that produce exit point process instructions.
- Outbound projects that produce outbound process instructions.
- Exit point process instructions that interpret an event message and determine the BOD message to use for the event. The exit point process instruction passes the name of the BOD to the Outbound Processor.
- Outbound process instructions that build the BOD for the event.

In the following sections an exit point project and an outbound project are created that produce the set of process instructions that build a Purchase Order BOD message.

Creating exit point and outbound projects

Use the LX ION PI Builder to create exit point, pcml, inbound, and outbound projects. Follow naming standards for all projects and use the file `extension .developer`. All projects are used to build a process instruction. See Chapter 1 for instructions to create a project folder.

Creating an exit point project

Use an exit point project to map element names to fields in an LX data structure that is defined in an LX Application. Map the entire data structure. This section explains how to create the project and how to name the project. To develop the exit point project, see the "Developing exit point and outbound projects" section.

1   Select the project folder, right click, and select **New Project**.

2   Navigate to the Infor Global Solutions folder and select LX Process Instructions. Click **Next**.
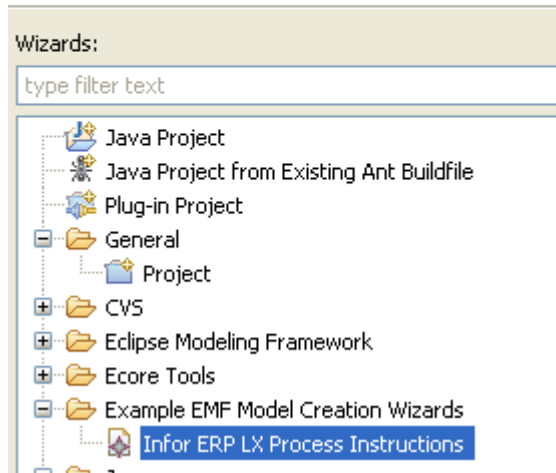
3   Select your project folder.

4   Specify the name of your project. When creating exit point projects use the following conventions to name the project:

- Use the parameters that were set when the exit point definition was created using program SYS635D1.

- The project name must be the value assigned to the Program concatenated with the value assigned to the interface point. See the screens below.

- All projects must end with the developer extension.

5 The Interface Point is set to EXIT01 and the program is PUR500B. Following the rules stated above, the project is named PUR500BEXIT01.developer.

```
Act Program     Interface Point      Sequence Call Program     Mode Status
     PUR500B     EXIT01                   2      SYS070C          0   Active
```
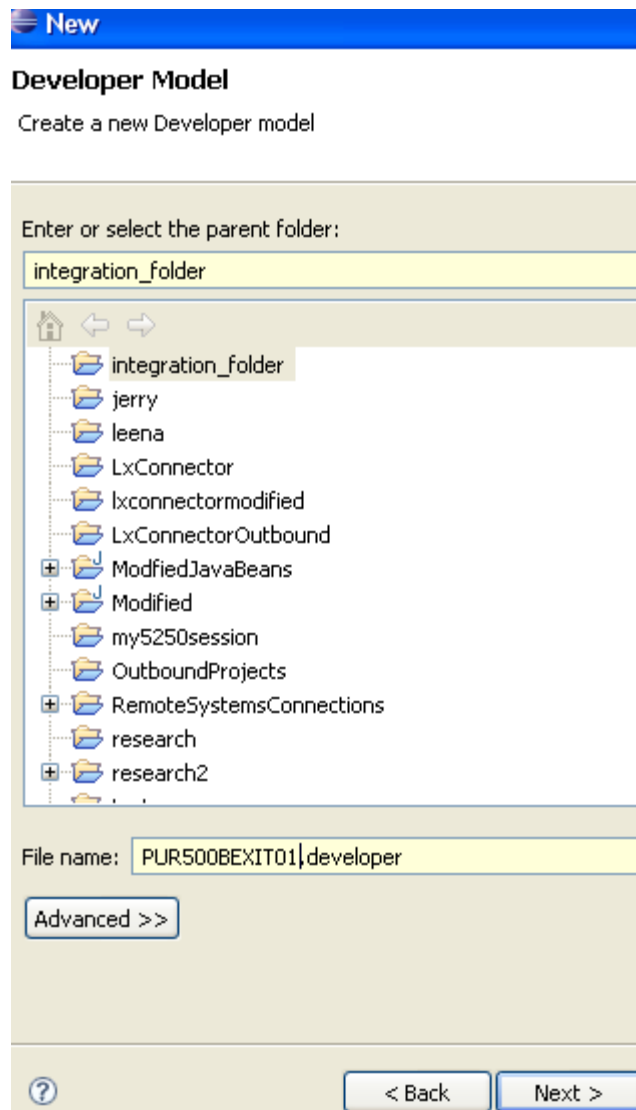

If a trigger is used to produce the LX event message, the name of the exit point project must be the value assigned to the **Call Program** field in SYS637D1. As shown in this screen, the name of the exit point project that would be assigned over this trigger is required to be INVIIMT02.developer:

6 Select Exit Point as the Model Object.

```
Triggered File    Trigger Time Trigger Event Sequence Call Program Status
IIM               AFTER        INSERT        1        INVIIMT02 Active
```

7 Click **Finish** to create the project PUR500BEXIT01.developer in the project folder in the Navigator pane.
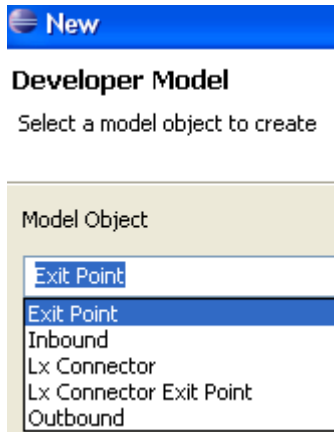
Creating an outbound project

When you create an outbound project the name follows this convention: NounName concatenated with the word Outbound.

For example, if you are creating a project that produces a PurchaseOrder BOD then the name of the project is `PurchaseOrderOutbound.developer`. All projects must end with the `.developer` extension.

To create an Outbound project:

1 Select the Outbound Model object.
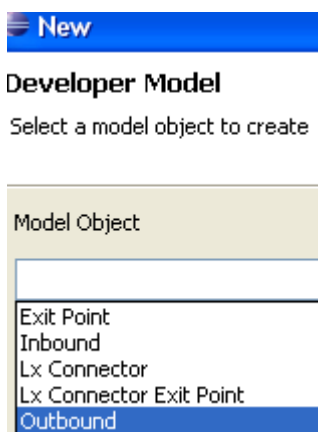
2 Click **Finish** to create the BodNounOutbound.developer project in the Navigator Pane.

Developing exit point and outbound projects

Exit Point and Outbound projects are used to build process instructions. The generated process instruction will contain a set of instructions that are used by the Outbound Processor to build a BOD message.

When building Exit Point projects the root node is the Exit Point node. When building outbound projects, the root node is the Outbound Node. To build a process instruction, add child nodes to the root node. See chapter 2 for definitions of all nodes available to the PI builder.

Both an Exit Point project and an Outbound project are required to produce a BOD message. An Exit Point process instruction may invoke many Outbound process instructions.

Create an exit point project for each exit point or trigger used to build the BOD message. Developers add child nodes into the project. The child nodes provide the ability to map elements to a 256-byte data structure that is defined in the business application. To determine the structure, manually inspect the business application.

Note: References within this chapter to BOD template assume development of LX Extension process instructions that use ION connectivity. If you are developing a process instruction that does not use ION connectivity a BOD template is not available so the XPath view is not supported. Manual mapping of data using the properties page is required.

Perform these tasks:

- Populate the XPath view with data for the appropriate BOD template.
- Create the exit point process instruction.
- Create the outbound process instruction.

These tasks are detailed in the following sections.

Populating the Xpath View

Note: Element Names must be added manually if you are building an exit point process instruction that does not use ION connectivity. The Xpath view requires a BOD template. If you are not using a BOD template you can skip this section.

The Xpath View is a view in the LX ION PI Builder used for mapping an xpath value to a Name property of an Exit Point Data node. To populate data into the Xpath view use the Search XPath View.

1   To open this view, select **Window > Show > View > Other**.

2   Navigate to the Infor LX View and select Search Xpath as shown below. After the Xpath View is filled with a BOD template you can select elements to map to the Exit Point.
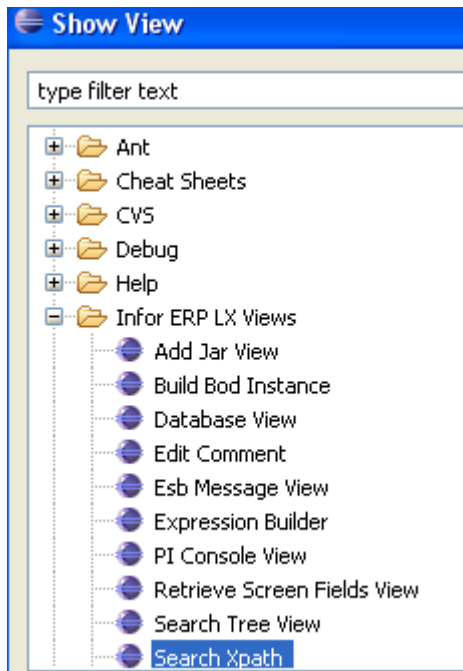
3   After you open the view, use Browse to navigate to the BOD template. Developers provide the BOD template. In this example, we selected the SyncPurchaseOrder.xml template to build an exit point process instruction for a Purchase Order BOD.
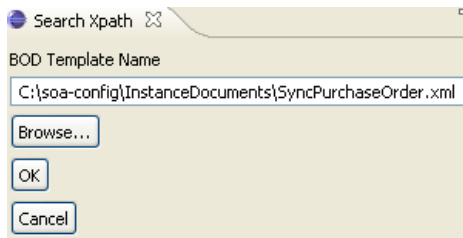
4   Click **OK** to open the XPath View with data as shown below.

When developing an exit point process instruction, you may need to map an Xpath value from the Xpath View to a field name in the exit point structure.

Developing the exit point process instruction

Note: If you do not have a BOD template you can skip this section and go to the "Developing the exit point process instruction without BOD template" section.

The nodes commonly required for building exit point Model Object are listed below. The property page for each is defined in Chapter 2.

- Exit Point
- Exit Point Definition
- Argument 1
- Argument 2
- Argument 3
- Argument 4
- Argument 5
- After Image – Triggers only
- Before Image – Triggers only
- BOD Element
- Priority

To develop the exit point process instruction using a BOD template:

1   Double click on the exit point.developer project that was previously created and saved to the project folder. This opens the project in design view. Initially the project contains the Exit Point root node.
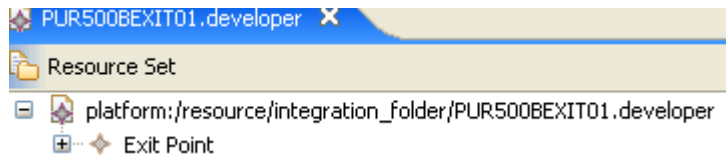

2   Enable the Properties page. The Properties page must be active when you are using the PI builder to build process instructions. The Properties page is updated programmatically when you make certain selections from Context menu items.

3   Select the Exit Point Node, right click, and select **Show Properties View** to display the Properties page. Chapter 2 defines the properties available via the property page for each node.
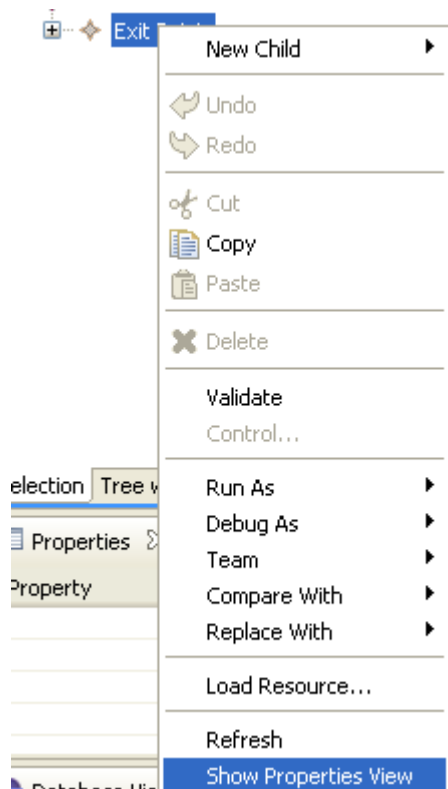
4   To develop exit point process instructions, add child nodes to the Exit Point node. You will map Property Names to fields in a data structure. Select the Exit Point node, right click, and select **New Child>Exit Point Mapping**.


5   Exit Point Mapping is a child of the Exit Point node. The name of the exit point must have the same name as the project. The name must follow the naming conventions discussed in "Create an Exit Point project". In this example, the project name is **PUR500BEXIT01**. Set the Property Name as shown in the following screen.
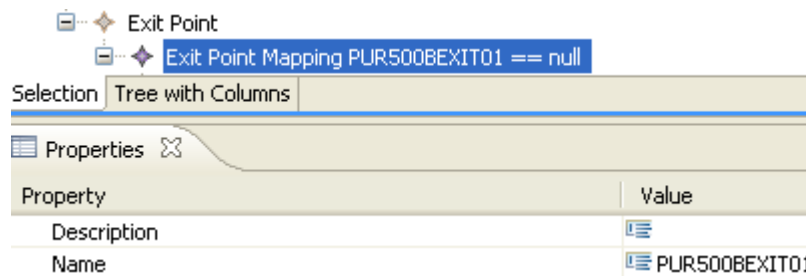
6   Add child nodes to the root to create the data structure that is defined in the LX Application. This requires addition of an Exit Point Definition node to the tree.

7   Select the Exit Point Mapping node, right click, choose **New Child>Exit Point Definition**. This creates a child node called Exit Point Definition. See the following screen.
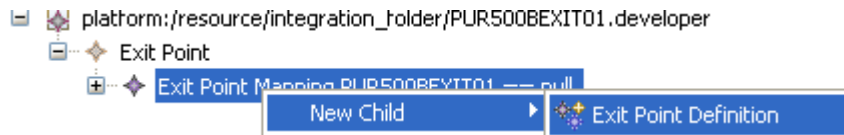
8   Do not set any of the properties in the Property View for the Exit Point Definition Node. This node is a parent node that contains the arguments required for mapping to an LX Application data structure. When mapping to an exit point, five arguments are added to the parent node. To add these arguments automatically, select a menu option in the XPath View.
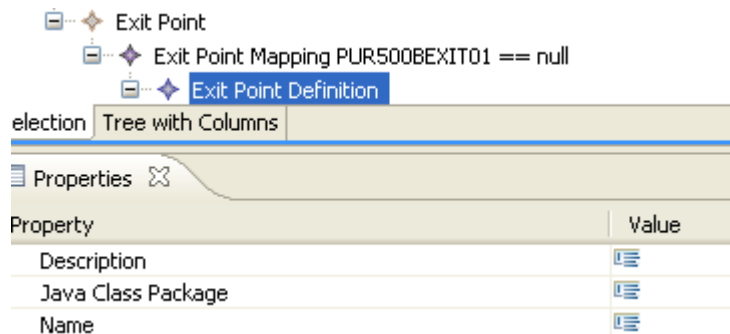
9   Navigate to the Xpath View and right-click in the view to display the context menu.

10 Select **Set Exit Point Arguments** from the context menu, as shown in the following screen.
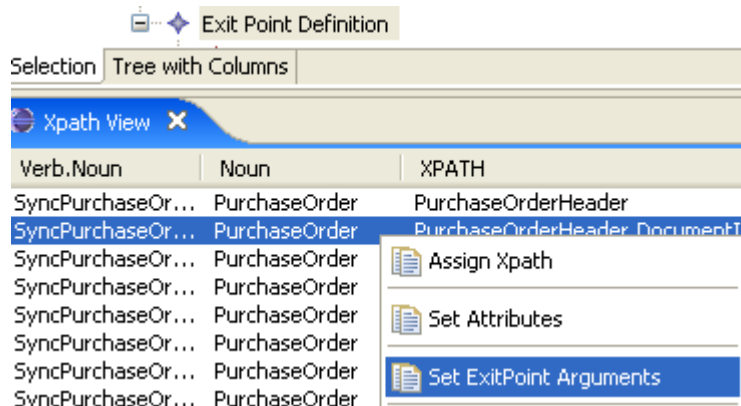
The Set Exit Point Argument action automatically creates the skeleton shown below. ARG4 and ARG5 are of interest to you as the developer. ARG1, ARG2, and ARG3 require no change as the default is correct for all exit point projects. Triggers require seven arguments.
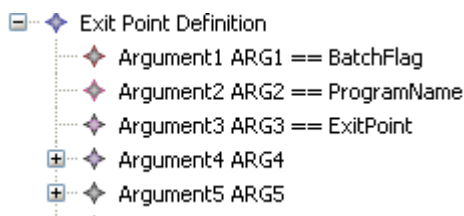
Developing the exit point process instruction without BOD template

To develop the exit point process instruction without use of a BOD template:

1   Double click on the exit point.developer project that was previously created and saved to the project folder. This opens the project in design view. Initially the project contains the Exit Point root node.
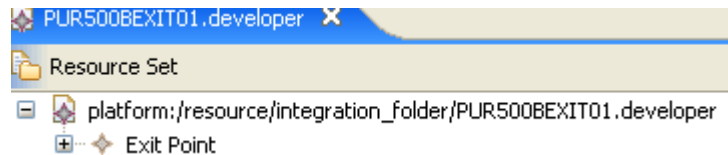
2   Enable the Properties page. The Properties page must be active when you are using the PI builder to build process instructions. The Properties page is updated programmatically when you make certain selections from Context menu items.

3   Select the Exit Point Node, right click, and choose **Show Properties View** to display the Properties page. Chapter 2 contains the property page definitions for all nodes.
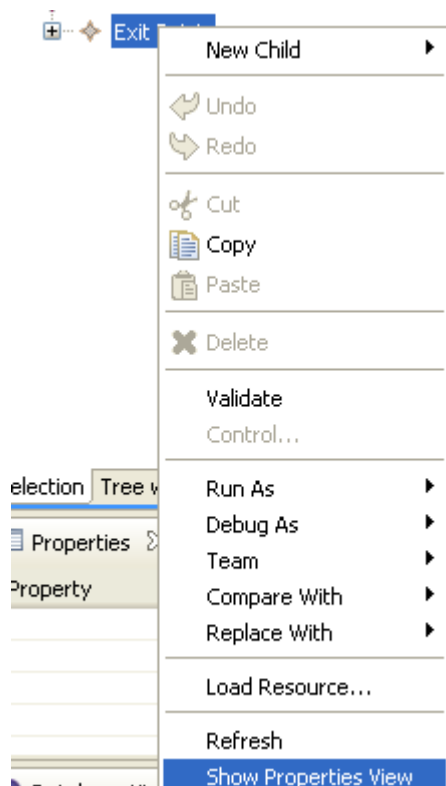
4   Open the property page for the node. To develop exit point process instructions, add child nodes to the Exit Point node. You will map Property Names to fields in a data structure.

5   Select the Exit Point node, right click, and select **New Child>Exit Point Mapping**.

6   The Exit Point Mapping is a child of the Exit Point node. The name of the exit point must have the same name as the project. The name must follow the naming conventions discussed in "Creating an exit point project". In this example, the project name is `PUR500BEXIT01`. Set the Property Name as shown in the following screen.
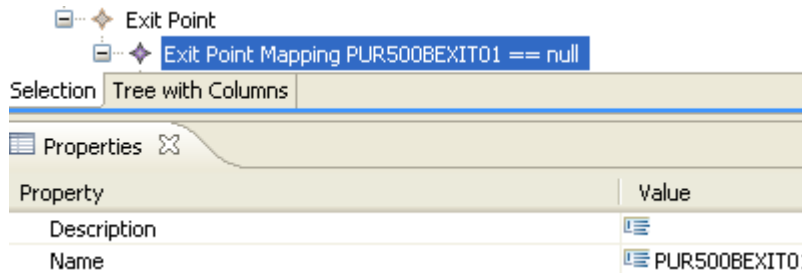
7   Select `Exit Point Mapping`, right click and set **New Child Argument1**. Repeat these steps adding Argurment 2, Argument3, Argument4 and Argument 5.
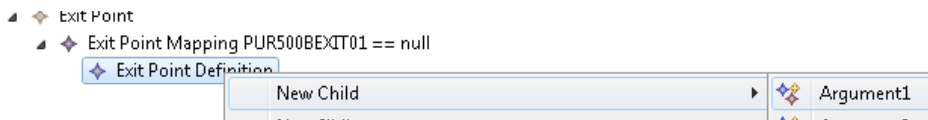
8   Do not modify any properties for Argument1, Argument2 or Argument3. These are preconfigured with constant data.
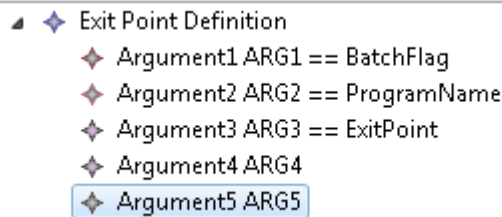
See "Mapping exit point arguments" section.

Mapping exit point arguments

Map ARG4 and ARG5. For ARG4, the entire 256-byte array must be mapped. The array must be filled exactly as defined in the LX Application data structure.

To map the arguments:

1   Mapping elements to the data structure requires you to add new nodes to the tree. Select node ARG4, right click, and select **New Child > Exit Point Data**.

2   Add as many Exit Point Data nodes as are needed to map the entire 256 byte data structure.

3   Use the Properties page for each Exit Point Data node. Set these properties:

a   Set the length and precision if the Type property for the node is packed. Set the Length property to the number of bytes and not the length of a string.

b   Set the precision to 0 if the type is String.

c   Set the Name. The Name property can have any string value but it cannot contain blanks. You can map an XPATH value into this field from the XPath View if you have a BOD instance otherwise manually add the Name. The value set in the Name property can be used in the Outbound project as a variable when mapping occurs.
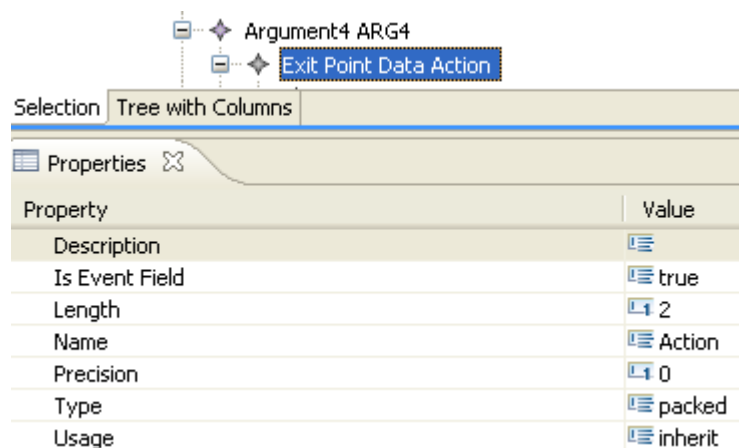
In the example, shown in Figure 4-27, the first 2 bytes in the 256 byte array define the event that occurred, that is, created, changed, or deleted. If the data maps to an event, the Is Event Field property must be set to true as shown. In this example, the length is set to 2 bytes and the precision is 0.

All event fields must be defined as an Enumerated type. Since the first 2 bytes In this example, map to an event, you must add Enumerated child nodes to the Exit Point Data Node.

d   Add an Enumerated child node for each event that is supported by a BOD message. For example, if the BOD message supports Add and Replace actions, add two Enumerated nodes as children of the Exit Point Data node.

The BOD Value property is the value assigned to the actionCode attribute in a BOD Message. In this example, Add is the BOD Value and it maps to an LX Value, In this example, the LX Value of `01`. This means that if the first 2 bytes are 01 the action Code in the BOD is set to `Add`.
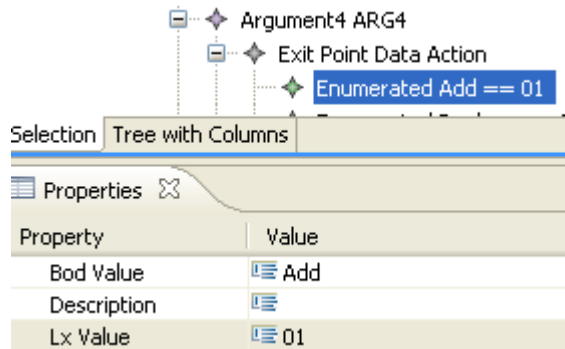
4   If the first 2 bytes of the data structure is 02 then the actionCode in the BOD message is set to
     Replace as shown below.
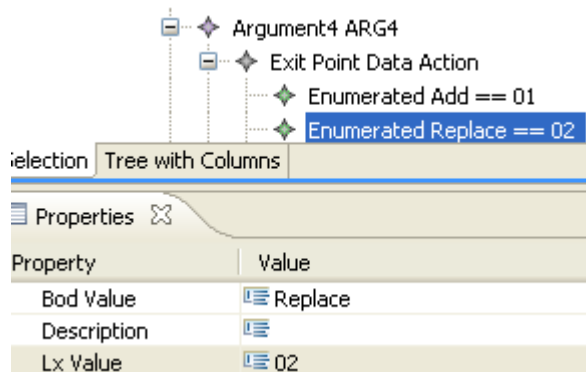
5   In this example, the first two bytes of the data structure do not map to a BOD element, but
     represent an event, so the Name property can have any value. In this example, it is set to
     Action. This property can be inspected and used by the outbound project when building an
     outbound process instruction. The complete definition for the PUR500B data structure is shown
     in the screen below. In this example, a Name assigned to a node, DocumentID, was selected
     from the XPath View. However, in the event the BOD template does not exist one would
     manually add the Name as DocumentID.

     The value assigned to Name properties cannot contain blanks. For example, Status Code is an
     invalid Name. The value assigned to the Name is converted to an xml element which does not
     allow blanks. For example these are valid values to use for the Name: Action, StatusCode,
     DocumentID, Program, User, and FILLER. All values assigned to the Name property are
     converted to an element in an xml message and the value associated with that name is assigned
     to the element.

     Example: At runtime, if the DocumentID has a value of 12345 in the raw data, then the converted
     xml message will contain `<DocumentID>12345</DocumentID>`.

     This converted xml message, which consists of the data from the exit point, is passed to the
     outbound process instruction. The outbound process instruction uses the converted xml
     message to build the outbound BOD message. The elements in this xml message can be used
     when mapping Element Names to fields when building an Outbound Process Instruction. For
     example the `<DocumentID>12345</DocumentID>` can be assigned to

PurchaseOrderHeader.DocumentID.ID by setting the field associated with this element to DocumentID (the name assigned to the property in the exit point).

To map blank data, set the Name property to `FILLER`. For example, if there are 183 blanks, set the Name as `FILLER` with a length of `183`. You may have multiple FILLER Names defined in the data structure.
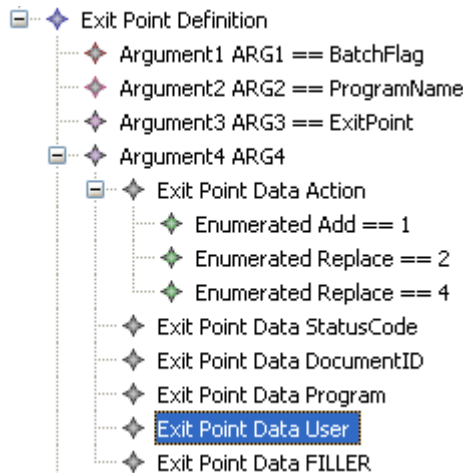
6   If you are using a BOD template you can map the appropriate Xpath to the Name. Select the Xpath from the Xpath View and then select Assign Xpath from the Context menu, as shown below. The values assigned to the Name property do not have to be Elements in the Xpath; these values are used for mapping purposes only.
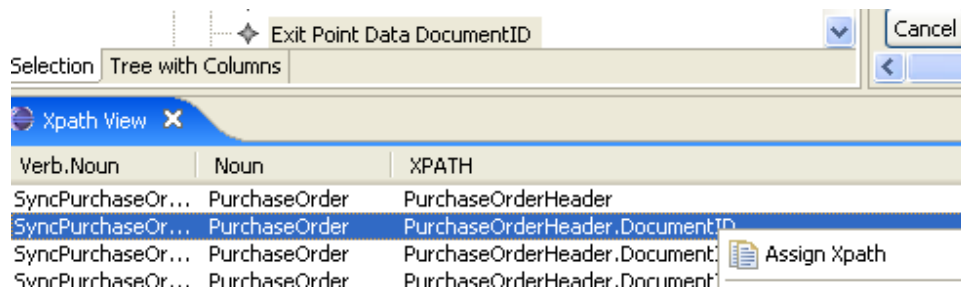
7   Map ARG5. The same mapping strategy as detailed previously applies to node ARG5. In this example, ARG5 is not used for mapping and is defined as FILLER as shown below.
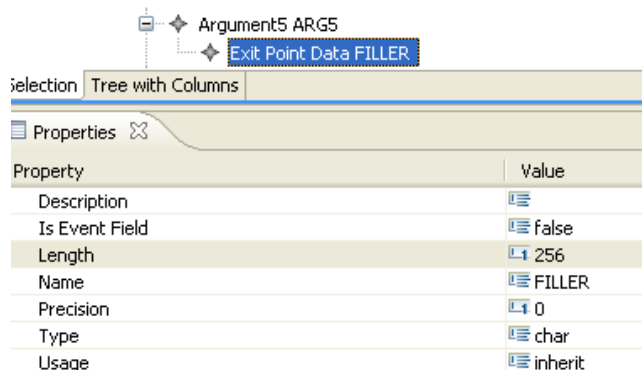
Adding a BOD element

The Exit Point Process instruction that is produced from the exit point project must contain information to indicate which outbound process instruction is used. All exit point process instructions will invoke an outbound process instruction. The outbound process instruction builds the BOD message. To provide this information, add a BOD element as a child of the Exit Point Definition node.

To add the BOD element to the exit point project:

1   Select the Exit Point Definition node, right click, and select **New Child>BOD Element**.
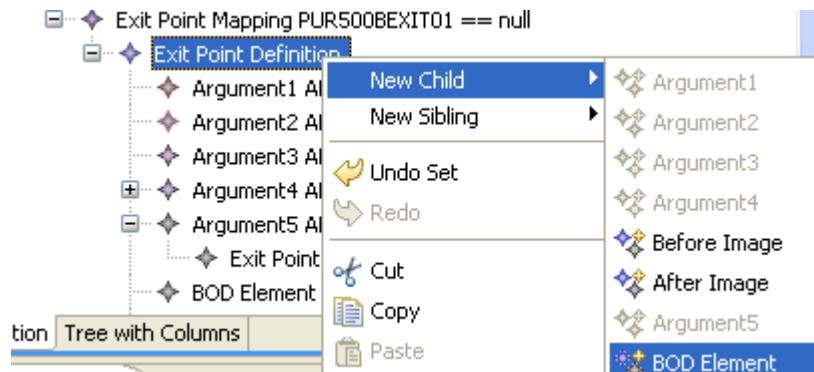
2   Select the new BOD Element and view the Properties page. See Chapter 2 for a description of the properties available for the BOD Element node.

3   Set the property, Process Instruction Name, to be the name of the process instruction that is used to build the BOD message when this exit point process instruction is called by an event. The value given to the Process Instruction Name must follow the naming convention, NounOutbound, as shown below. An exit point program may invoke many outbound process instructions. A new BOD Element is added for each process instruction that can be invoked.
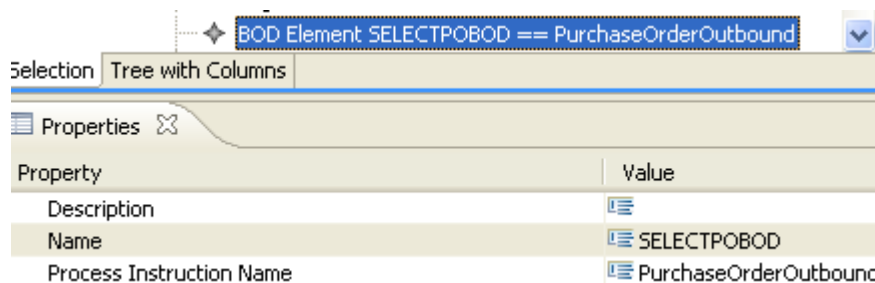
Example: If you are creating a PurchaseOrder outbound process instruction, set the property, Process Instruction Name, to `PurchaseOrderOutbound`. This name must match the name of the outbound process instruction that will create the PurchaseOrder BOD message.

The Properties page also has a Name. The Name is the entry point defined in the outbound process instruction. The Entry Point is the Name of an instruction that is defined in the generated outbound process instruction. This is the instruction that is executed when the process instruction is loaded. It is the starting point for building the BOD. Typically, this Name points to a Condition node that contains additional instructions used by the LX Extension or LX Connector

runtime. Figure 4-35 shows the Name **SELECTPOBOD** which is the name of an instruction defined in the PurchaseOrderOutbound process instruction. When you define the outbound process instruction, you must create an instruction with this name. See "Creating an outbound process instruction."

4   After you complete the mapping for ARG4 and ARG5 and add the BOD Elements, save the project.

Generating the exit point process instruction

To generate the exit point process instruction:

1   Select the PUR500BEXITO1.developer project from the Navigator Pane.

2   Right click and choose **Generate Process Instruction** to create a `PUR500BEXIT01.xml` file in the project folder. This file is your exit point process instruction.

3   If creating a project to be used in a LX Extension integration move this process instruction to the LX Extension installation directory in the IFS to test it. If you are creating this for the LX Connector move the process instruction to the IFS directory where the LX Connector is installed.
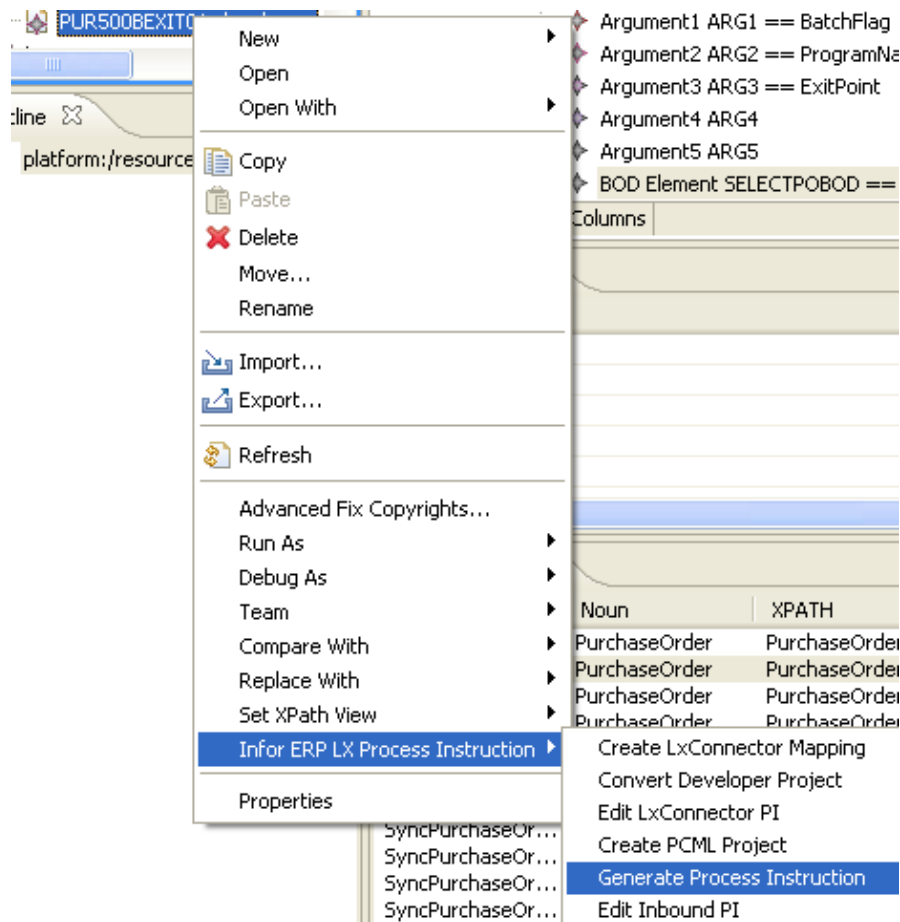
Add a Priority to the BOD Element in an Exit Point Model Object

Appendix B indicates that the BOD Element node may have a child Priority node. A priority node is used for performance reasons. For example, suppose the Inbox was flooded with ReceiveDelivery BOD messages that caused LX event PUR550D2POUPDATE to fire for each. Furthermore, assume that the ReceiveDelivery could be for the same PurchaseOrder.

This could cause an abundance of these events to fire into the Safe Box. To prevent a bottle neck in the outbox a Priority node can be added that will check the LX Event in the Safe Box and filter based on key data. For example, the screen below shows a Priority node added to the BOD Element node that produces a PurchaseOrderOutbound message. The properties for the Priority are defined in Chapter 2.

In this example, the properties are set to indicate that when this event fires the runtime will check to see if the Action is a Replace. If it is a Replace and was fired as a result of an Inbound message (property Is From Inbound is true) the runtime will inspect the value of the Key Element and go through the Safe Box removing any duplicates (property No Duplicates is true). Those that remain will be processed at a priority of 0 (property Priority Level is 0). Note in the picture below that the Key Element must be mapped to a name within the Exit Point definition, In this example, the mapping is in ARG4.

Creating an outbound process instruction

See Appendix B for a table of Parent/Child relations when building a Model Object tree. See Chapter 2 for a description of the Nodes and the properties that are defined on the property page.

The outbound process instruction consists of several instructions that are referenced by a Name property. Outbound process instructions are loaded when the generated Exit Point process instruction is loaded. The Exit Point process instruction is used to pass LX event data to the Outbound Model Object. The Name of the outbound process instruction to load is defined in the Exit Point Process Instructions as is the reference to the Instruction in the outbound process instruction to run.

The following example describes the creation of a very basic outbound process instruction.

This example describes how to create an outbound process instruction that produces a PurchaseOrder BOD. In the basic case, you would have an instruction that is a mapping between an LX database file and elements in the BOD.

In the Exit Point Program, previously described in this document and named **PUR500BEXIT01**, the BOD Element node contains a Name property that is a reference to an instruction. In the Exit Point example, the Name was set to **SELECTPOBOD**. **SELECTPOBOD** is a reference to an instruction which must be defined in the outbound Model Object project that produces a PurchaseOrder BOD.

In this example, we are creating the PurchaseOrder BOD that must contain the SELECTPOBOD instruction. This is the entry point instruction that is loaded when the exit program is executed. The sections that follow describe how to create a PurchaseOrder outbound model object that produces a process instruction. The example instructs the developer to use a Database node that maps Elements to database field values and to define the entry point instruction using a Condition node. The Condition node is used as a container of other instructions and in this example, contains the Database node that provides the mapping. The process instruction is produced after all nodes have been added to the Outbound Mode Object tree view.
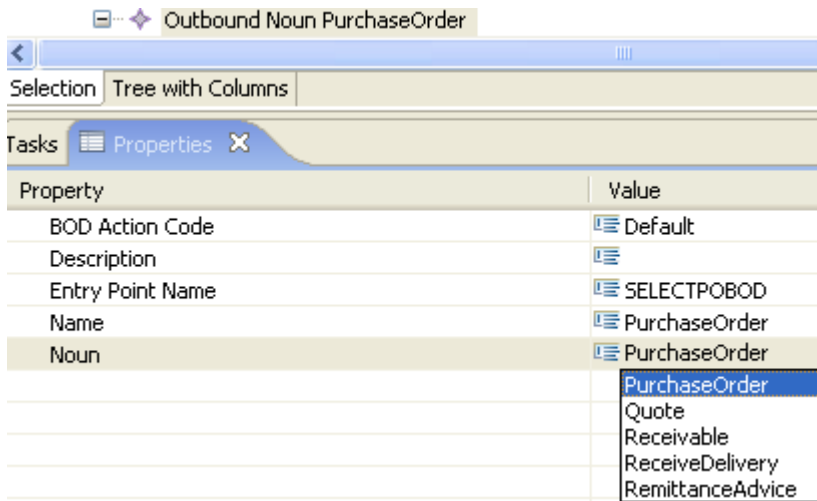
Adding the Outbound Noun

The Outbound Noun node is a required node that is used to identify the BOD. It assigns the BOD name and the name of the instruction that is the point of entry into the process instruction loaded at runtime. The purpose of this example is to demonstrate how to create a PurchaseOrder BOD. To build the PurchaseOrder process instruction requires adding new child nodes to the tree view. Each child node has a set or properties defined in the property page for the node. All property pages are defined in Chapter 2.

To add the outbound noun:

1   Open the outbound developer project, PurchaseOrderOutbound.developer, by double clicking the project in the Navigator pane.

2   The root node for all outbound process instructions is Outbound. The root node is defined when the project is created. The first requirement for building an outbound process instruction is to add the Outbound Noun node. To add this node, select the root node Outbound, right click, and select **New Child>Outbound Noun**.
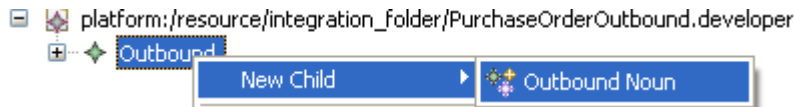
3   Open the property view and set the properties for the Outbound Noun node. Define the Noun
    and Entry Point Name properties. Review these properties:

    - Do not change the BOD Action property; this is the default property.

    - The Entry Point Name property is not required as there could be multiple Exit Point/Trigger
      Projects that call the Outbound Model Object with different entry points. It might be preferred
      practice to have the Entry Point Name property be set to have the same value as the Name
      property of the BOD Element defined in one of the Exit Point/Trigger Projects. For the
      example shown earlier in this chapter the Name **SELECTPOBOD** references the Instruction
      that is the entry point in the Outbound Model Object.

4   You may select the name of the noun using the Noun drop down. If it is not listed, set the
    selection to **none** and specify the noun into the Name property, such as **PurchaseOrder**.

5   To set the BOD name, select the drop down widget for the Noun property. For this example, a
    PurchaseOrder BOD is being produced so the Noun is selected as PurchaseOrder.

Adding Child Nodes to the Noun node

Adding child nodes in the designer view of the outbound project produces a set of instructions called
process instructions used at runtime to generate a BOD message. To add child nodes to the Noun
node, right click on the Noun node and select New Child. The menu displays a list of choices. Most
outbound projects require the addition of the child nodes listed below. Each of these nodes is
explained using the development of a PurchaseOrder BOD. All properties for the nodes are
described in Chapter 2.

- Narrative
- Instruction

- Condition
- Mapping Detail
- Mapping
- Database

If you are creating an Outbound Model Object tree view that will generate a process instruction for an ION integration the following nodes are required.

- Verb
- Namespace
- BOD Version

Adding a BOD Version node

Note: If you are creating an outbound process instruction for the LX Connector, the BOD Version is not supported, you may skip this section. The LX Connector does not use ION connectivity.

The BOD Version node is required for outbound projects that use the LX Extension and use ION to route messages. This node is used to add version information as attributes of the BOD that is produced by the generated process instruction. The properties of the node are shown in Chapter 2.

1   To add a node, select parent node, OutboundNoun, right click and select **New Child BOD Version**.

2   The property view for the BOD Version node contains properties that set attributes when the BOD message is created at runtime.

- Set the Release ID property to the OAGIS release, for example, `9.2`.
- Set the BOD Version ID to be the release of the Infor BOD, for example, `2.5.1`.
- Set the Bod Version ID to the version of the Infor BOD, for example `2.5.1`.
- Set the Version ID property. This property is the version of xml which is `1.0`. If the Version ID is not set in the property view, it will default to `1.0`.
- The property Document Root Prefix is deprecated, do not set it.
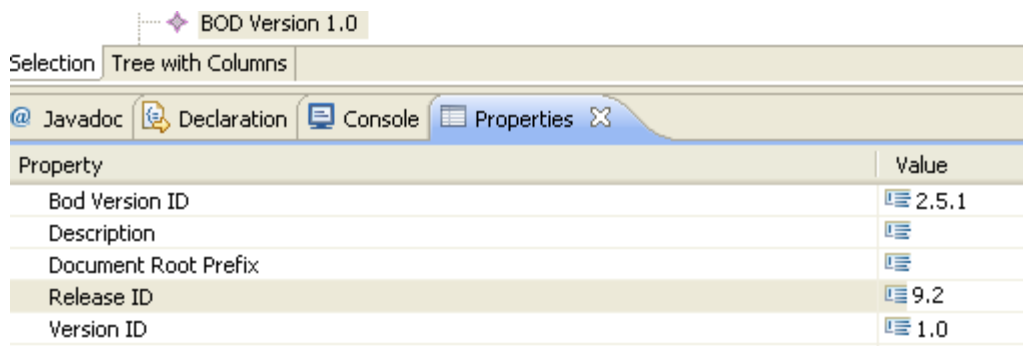
Adding a Narrative Node

All outbound projects may contain a Narrative node but this is not a required node. This node provides copyright and historical information about the process instruction. All information provided in the Narrative is added into the process instruction produced from the Model Object. The node provides historical information about the process instruction.

To add a Narrative node:

1   Select the Outbound Noun node, right click, and then select **NewChild > Narrative**.

2   After you add a Narrative node, add child nodes that provide the instructions. To add child nodes, select the Narrative node, double click and the select **New Child > Copyright**, **New Child > Comment** or **New Child > Modification**.

3   To add copyright information, select Narrative, right click, and select **New Child > Copyright**. The Copyright node has a single property Copy Right Statement that has a default value. The default value is set to the Infor Copyright statement and is shown below.

THIS SOFTWARE IS THE PROPERTY OF AND CONTAINS CONFIDENTIAL INFORMATION OF INFOR AND/OR ITS AFFILIATES OR SUBSIDIARIES AND SHALL NOT BE DISCLOSED WITHOUT PRIOR WRITTEN PERMISSION. LICENSED CUSTOMERS MAY COPY AND ADAPT THIS SOFTWARE FOR THEIR OWN USE IN ACCORDANCE WITH THE TERMS OF THEIR SOFTWARE LICENSE AGREEMENT ALL OTHER RIGHTS RESERVED.

(c) COPYRIGHT 2009 INFOR. ALL RIGHTS RESERVED. THE WORD AND DESIGN MARKS SET FORTH HEREIN ARE TRADEMARKS AND/OR REGISTERED TRADEMARKS OF INFOR AND/OR ITS AFFILIATES AND SUBSIDIARIES. ALL RIGHTS RESERVED. ALL OTHER TRADEMARKS LISTED HEREIN ARE THE PROPERTY OF THEIR RESPECTIVE OWNERS.

- To add Comment nodes, select Narrative, right click, and select **New Child > Comment**. The properties of the Comment node are shown.
- To edit the Comment property, use the Comment View provided with the LX ION PI Builder.
- To open the Comment View, double click the Comment node.
- To save the change to the Comment node, click **OK**.
- The property Print Comment is not a supported property.
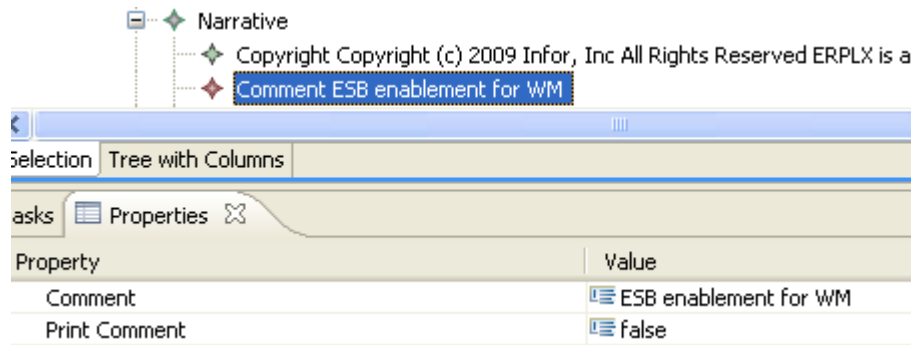
4   Modifications provide historical information about the process instruction. To add a Modification node, select the Narrative node, right click, and select **New Child > Modification**.

5   The properties of the Modification node are shownbelow. These nodes provide the ability to add a defect Number, Date and Name information. A Modification can contain Comment nodes.

6   To add a Comment to a Modification node, select the Modification node, right click, and select **New Child > Comment**. The comment provides information about the modification.
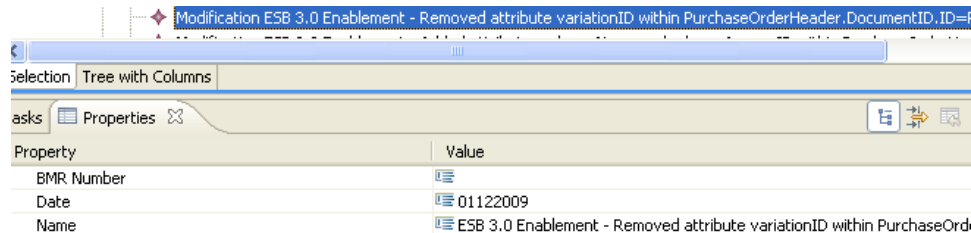
Adding an Instruction node

Every outbound project must have at least one Instruction node which is the instruction that gets executed when the process instruction is loaded. All Instructions are referenced by adding child node Instruction Name. The Name property of the Instruction must have the same Name as that assigned to the parent Instruction node Name property. Most projects will contain many Instruction Nodes.

1   To add an Instruction node, select the Noun node, right click, and choose **New Child > Instruction**. Appendix B shows the child nodes available to the Instruction.

2   To create a Database type, Instruction that provides the mapping between an Element in a BOD message and a database field in a result set.

3   To add an Instruction node, select the Noun node, right click, and choose **New Child > Instruction**. All Instruction nodes must have the property Name defined. To define the name, open the property view for the Instruction node. Set the Name property to an alpha string. This is the name of the instruction that can be invoked from a Condition node or by an Instruction Name node. If the instruction does not have this property set, the process instruction will not produce a BOD message.

The screen below shows that the Instruction node is named PurchaseOrderHeader. The Is Loop Type is set to false because it is not a looping instruction. Looping instructions are used when processing Inbound messages. The Organization Hierarchy is set to `false`, this is not used.

For this example we are creating a Purchase Order and need to map the BOD elements that will generate into the BOD message to the header fields in a Purchase Order.

Description	
Is Loop Type	false
Name	PurchaseOrderHeader
Organization Hierarchy	false

4   The purpose of the PurchaseOrderHeader instruction is to map BOD Elements to database fields. This requires addition of new child nodes. To map BOD elements to database fields requires addition of the Database node.

5   To add a Database Node as a child of the PurchaseOrderHeader Instruction, select the PurchaseOrderHeader Instruction node and choose **New Child > Database**. Set the properties for the Database node in the property view. See Chapter 2 for the property descriptions.

6   All Database nodes require that the Name property is set. The value for this property must be the same value that was given to the parent Instruction node. In this example, the Instruction node property Name was set to **PurchaseOrderHeader**. Therefore, the Name property for the Database node must be the same, **PurchaseOrderHeader**, as shown below.

Property	Value
Description	
Locate Row Xpath Name	
Name	PurchaseOrderHeader
Type	SQL

7   The Database node may have these child nodes shown below. Each is described in Chapter 2. For this example, we will use Mapping Detail and Database SQL Statements.

- Comment.
- Mapping Detail
- Database SQL Statements.

Adding a Mapping Detail

In this example, we are mapping element names to database fields. This requires use of the Mapping Detail node.

1   To add a node, select the Database node, right click, and choose **New Child > Mapping Detail**.

A Mapping Detail Node is required for mapping BOD elements to database fields. This node is used to hold child nodes named Mapping. Mapping nodes are used to define the Element name as well as attributes required by the Element. The Mapping nodes are used to map an element that is added to a BOD message to a database field. The name assigned to the element must be

the complete path to the element, in this document the Name assigned to the Element is known as the Xpath.

2   A Mapping Detail node contains many Mapping nodes. The Mapping node property view contains properties that provide a name for the Element and the name of the Database Field that contains the value for the Element. To add a Mapping detail, select the Mapping Detail node, right click, and choose **New Child > Mapping**.
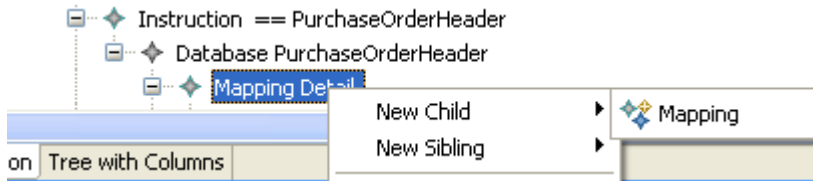

3   After you add a Mapping node, open the property view for the node. The properties for the Mapping node are defined in Chapter 2. In this example, we are setting the Element and the Database Field properties. This example instructs how to use the designer view to set the value for the Element and the value for the Database Field.

4   If you are not using a BOD template proceed to step 4. On the property view set the value for the Element property as shown below. Both the Xpath View and the Property View must be open to set the Element name.

5   Select the Mapping node. Navigate to the Xpath View, and scroll through the XPATH column until you find the Xpath to map.

6   After finding the correct XPATH value, select the row.

7   Right click in the XPath View to display the menu and select **Assign Xpath**. When the Assign Xpath is selected, it sets the Element property in the Mapping node that was selected. See the screens below.
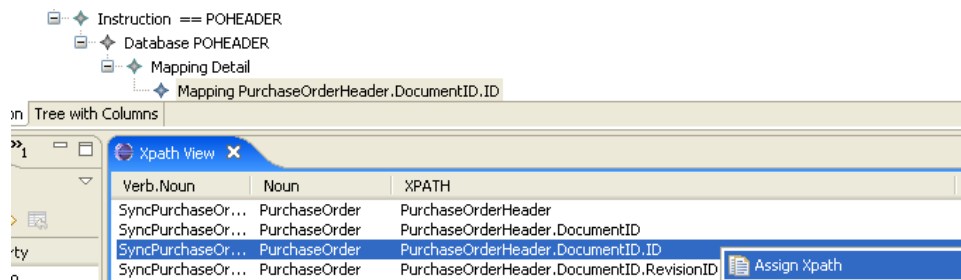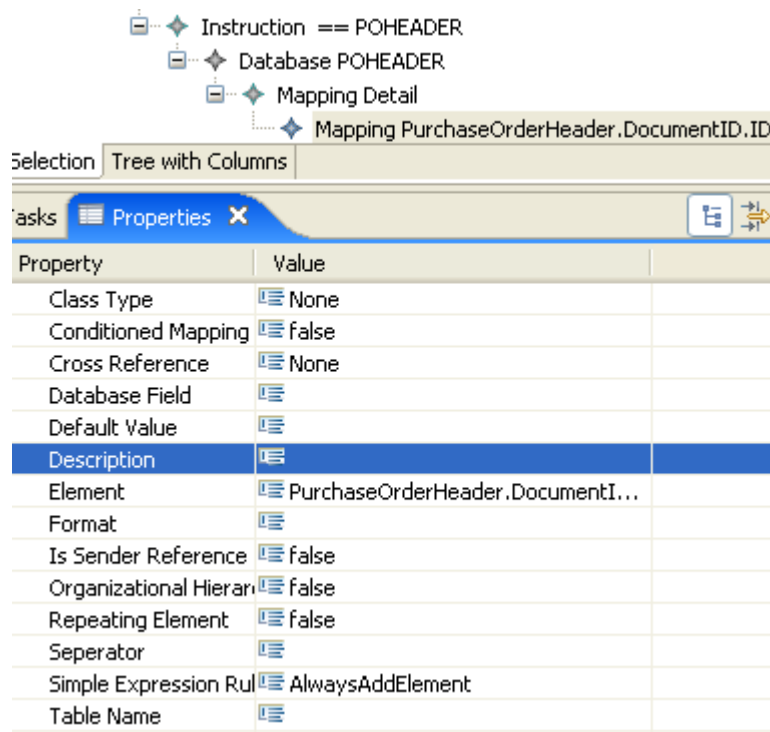
8 You may use the Database View to set both the Element and Database Fields. To use the designer view to set the Database Field property for the Mapping node, the Property view and Database View must be open. To set the Database Field:

  a Select the Mapping node, navigate to the Database View, and scroll through the view to find the column to map.

  b After selecting the row right click to display the menu.

  c You can elect to use the Description field as the Element. In this case select the **Add Description** from the menu. This will set the **Datatabase** field to the value in the Column and the Element to the Description and the Table, otherwise from the menu select **Assign Field**. This sets both the Database Field and Table properties for the Mapping node selected. See the screens below.
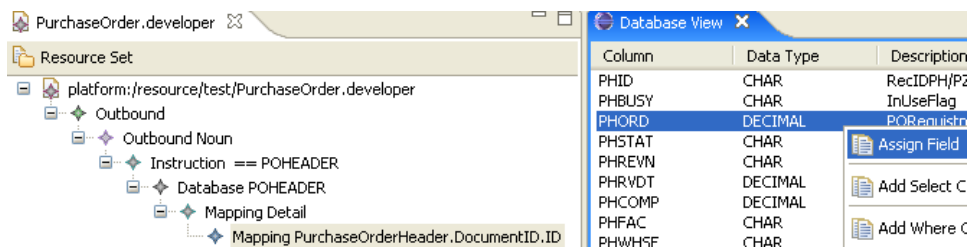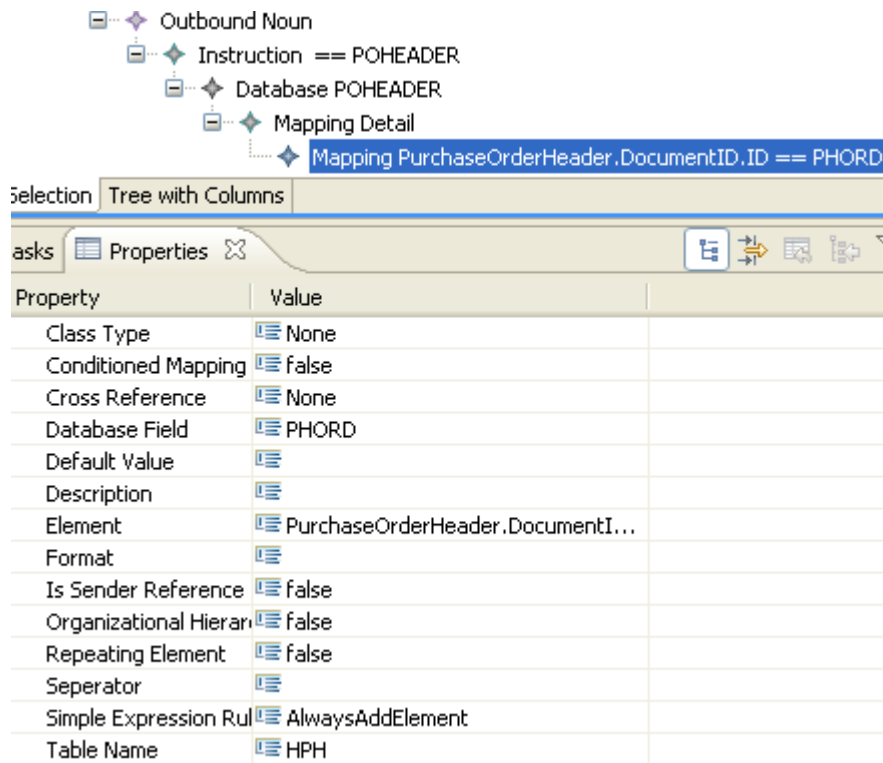
9   Repeat steps 2, 3, and 4 to add all required mappings.

10 After all Mapping nodes have been added, create SQL statements that retrieve the values for the
database fields defined in the Mapping nodes. To add SQL statements requires the addition of
the Database SQL Statements node.

11 Select the Database node, right click, and choose **New Child > Database SQL Statements**.
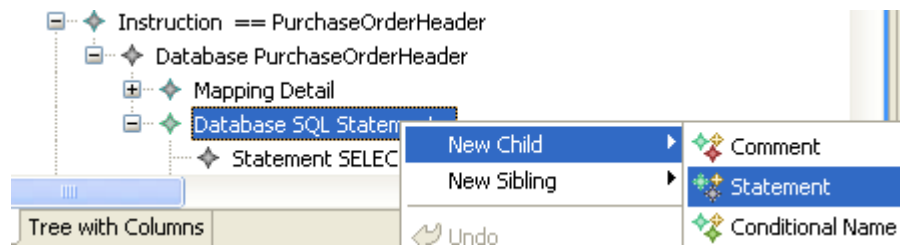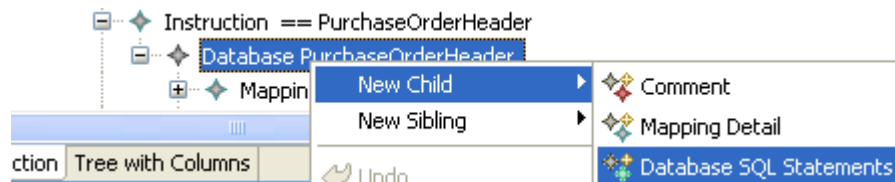
12 The Database SQL Statements node may contain many SQL statements. To add a statement,
select the Database SQL Statements node, right click, and select **New Child > Statement**.

13 Selecting the Statement opens the property view for the node.

a   After adding the Statement node, double click the node to open the SQL Builder View. Use this view to create an SQL statement.

b   Click **OK** to set the Statement property in the property view page. A Database SQL Statement may contain multiple Statement nodes. Each SQL statement is executed in order of the appearance in the designer view. All SQL statements in the Instruction>Database node are executed before mapping occurs.

```
Instruction == PurchaseOrderHeader
  Database PurchaseOrderHeader
    Mapping Detail
    Database SQL Statements
        Statement SELECT HPH.PHORD,HPH.PHLDT,HPH.PHLTM,HPH.PHENDT,HPH.PHCMT,HPH.PHID,H
        Statement SELECT ESN.SNDESC AS PRTNOTE  FROM ESN WHERE ESN.SNTYPE='P' and ESN.SNC
        Statement SELECT ESN.SNDESC AS PRVNOTE FROM ESN WHERE ESN.SNTYPE='P' and ESN.SNC
```
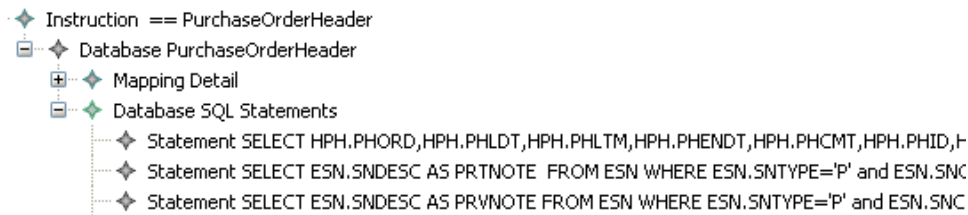
Adding a Condition node

Condition nodes are used to add additional instructions. Appendix B indicates that the Condition Node has 2 child nodes available, Comment and Conditional Instruction. In this example, we are interested in the Conditional Instruction. A Condition node can have many child Conditional Instruction nodes each of which contains a set of child nodes. It is common in Outbound Model Object projects to use the Condition node as the instruction that is the point of entry.

1   Add a Condition node to the Noun Node. This Condition node will be the point of entry into the process instruction. To add a Condition Node, select the Outbound Noun node, right click, and select **New Child > Condition**.


2   Set the Condition properties. Set the Name to be the entry point Name assigned in the Exit Point project. In this example, that name is **SELECTPOBOD**. Leave the default values for all other properties. When the Exit Point program invokes this outbound process instruction it will start executing instructions at the Condition named **SELECTPOBOD**.
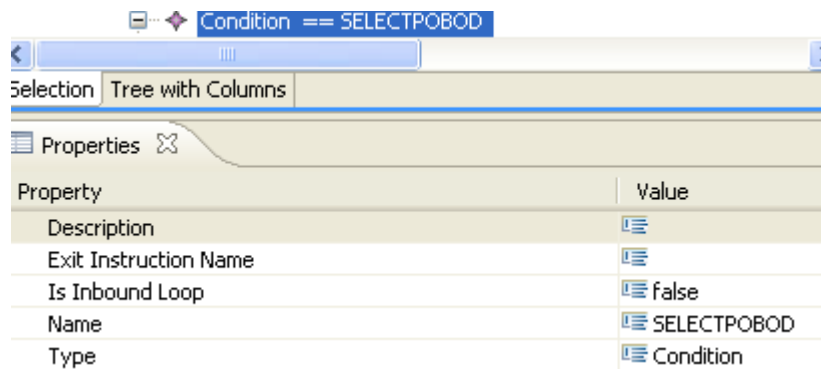
3 The entry point for this process instruction is a Condition node, which may contain many Conditional Instruction nodes. It is common that developers add a Conditional Instruction and then add child nodes to the Conditional Instruction. The children of the Conditional Instruction provide varying capabilities, such as providing evaluation of expressions and execution of other Instruction nodes added to the tree.

4 In this example, we add a Conditional Instruction to the tree so that we can add a child node Instruction Name. Adding this child node will reference the Instruction node that maps the Purchase Order header information. We can also add If Condition child nodes to the Conditional Instruction to provide the ability to make decisions. Because other instructions are required to create this process instruction, add new Child Conditional Instruction as a child of the Condition. Do not set any properties of the Conditional Instruction node.

5 In this example, it is assumed that a Database Instruction has already been created as described in the "Adding a Mapping Detail" section and the Name assigned to the Instruction is **PurchaseOrderHeader**. The purpose of this example is to create a process instruction that publishes a PurchaseOrder BOD. To produce the BOD message requires adding child nodes into the Conditional Instruction. The PurchaseOrderHeader Database Instruction contains the mapping and SQL statements that build the header portion of the PurchaseOrder BOD message; therefore, we need to execute this instruction from the Conditional Instruction node. To execute the PurchaseOrderHeader Instruction add **New Child > Instruction Name** as a child of the Conditional Instruction node.
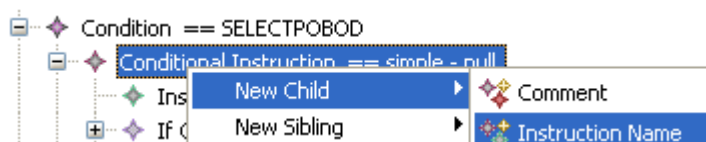
6 Open the property page for the Instruction Name node and set the properties. See Chapter 2 for a description of the Instruction Name node.

a Set the Name to **PurchaseOrderHeader**, which is the name given to the Database Instruction created earlier (PurchaseOrderHeader).

b Use the default for Check Return Status (false). This is not applicable for outbound messages. It is used by inbound process instructions for error handling.
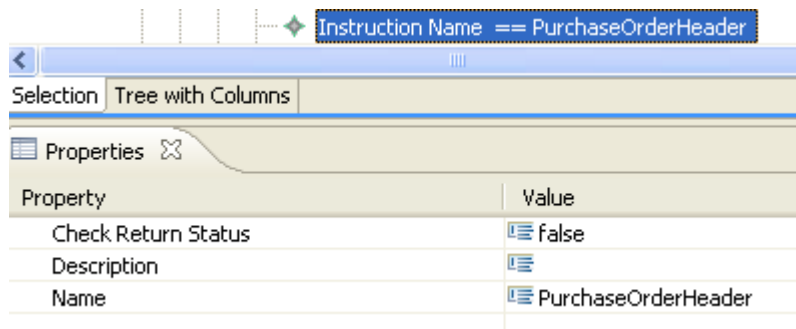
7 In this example, the Condition node is the instruction that is executed when the process instruction is loaded. The Condition node has a single Conditional Instruction node. The Conditional Instruction node contains a single Instruction Name child node that has a name of **PurchaseOrderHeader**. The Name is a reference to the Database Instruction. The Instruction has the same name. In this example, when the Instruction Name is executed, the Database Instruction named **PurchaseOrderHeader** produces a BOD message that has the elements defined within the Database Instruction node. You would also need instructions that map purchase order lines and instructions that write a verb into the BOD message. See Chapter 5 for instructions to process multiple rows of data.

Mapping elements to database fields example

In this example, we will map Elements in the Purchase Order to fields in the HPH file.

To map elements to database fields:

1 Access the **Retrieve Screen Fields View** window.

2   Set the relevant properties:

Property	Description
Host machine	Enter the name of the host machine where the database or files library exists.
Library	Enter the name of the library where the files exist
Display file names	Leave this field blank. Use this field when you build inbound process instructions.
Table	Enter the metadata tables to retrieve. If multiple tables are required, separate each by a comma, for example, HPH,HPO.

Property	Description
BodTemplateName	Select a BOD instance name to display in the Xpath view. This view allows mapping from an instance document to a Mapping node property.
Inbound/Outbound attribute	Leave this field blank. Use this field when you build inbound process instructions.
User/Password	Enter a user ID that has access to the host as well as to the database.

3   Click **OK** to populate the Database View.

4   Add a Mapping Detail node as a child node to the Database. This node will contain many child Mapping nodes. Each Mapping node defines the mapping between the BOD element and a field in a database. Add a Mapping node for each element in the BOD. If you are creating LX Extension messages that are routed using ION, as you assign XPath values to the Element property of the Mapping node you must ensure that the XPath assigned to the Element is in the correct sequence that the BOD xml schema requires. Messages passed via ION must be valid BOD messages, if not, errors occur during routing. When a BOD message is produced by the Outbound Processor the elements are inserted into the message in the order they appear in the designer view.

5   The screen below shows an example of a mapped Database Instruction named **PurchaseOrderHeader**. The Element property in the Mapping node was populated using the XPath View and the Database Field was populated using the Database View. See Xpath/Database mapping section.

```
□──◆ Database PurchaseOrderHeader
   □──◆ Mapping Detail
      ⊞──◆ Mapping PurchaseOrderHeader.DocumentID
      ⊞──◆ Mapping PurchaseOrderHeader.DocumentID.ID
      ⊞──◆ Mapping PurchaseOrderHeader.LastModificationDateTime == PHLDT
      ⊞──◆ Mapping PurchaseOrderHeader.DocumentDateTime == PHCDT
         ──◆ Mapping PurchaseOrderHeader.Description == PHCMT
      ⊞──◆ Mapping PurchaseOrderHeader.Note == PRTNOTE
      ⊞──◆ Mapping PurchaseOrderHeader.Note == PRVNOTE
      ⊞──◆ Mapping PurchaseOrderHeader.Status.Code == PHID
      ⊞──◆ Mapping PurchaseOrderHeader.Status.Description == PHID
      ⊞──◆ Mapping PurchaseOrderHeader.CustomerParty.Location.ID == PHWHSE
      ⊞──◆ Mapping PurchaseOrderHeader.CustomerParty.Location.Name == LDESC
      ⊞──◆ Mapping PurchaseOrderHeader.CustomerParty.Location.Name
      ⊞──◆ Mapping PurchaseOrderHeader.CustomerParty.BuyerContact.ID == PHBUYC
      ⊞──◆ Mapping PurchaseOrderHeader.CustomerParty.BuyerContact.Name
      ⊞──◆ Mapping PurchaseOrderHeader.SupplierParty.PartyIDs.ID == PHVEND
      ⊞──◆ Mapping PurchaseOrderHeader.SupplierParty.Name == VNDNAM
      ⊞──◆ Mapping PurchaseOrderHeader.SupplierParty.Name
      ⊞──◆ Mapping PurchaseOrderHeader.SupplierParty.Location.Address
      ⊞──◆ Mapping PurchaseOrderHeader.SupplierParty.Contact.Name == VCON
      ⊞──◆ Mapping PurchaseOrderHeader.SupplierParty.Contact.Name
      ⊞──◆ Mapping PurchaseOrderHeader.SupplierParty.Contact.Communication.UseCode
      ⊞──◆ Mapping PurchaseOrderHeader.SupplierParty.Contact.Communication.DialNumbe
      ⊞──◆ Mapping PurchaseOrderHeader.SupplierParty.Contact.Communication.UseCode
      ⊞──◆ Mapping PurchaseOrderHeader.SupplierParty.Contact.Communication.DialNumbe
      ⊞──◆ Mapping PurchaseOrderHeader.SupplierParty.Contact.Communication.UseCode
      ⊞──◆ Mapping PurchaseOrderHeader.SupplierParty.Contact.Communication.URI == V
      ⊞──◆ Mapping PurchaseOrderHeader.ShipToParty.PartyIDs.ID
      ⊞──◆ Mapping PurchaseOrderHeader.ShipToParty.Name == PHNAME
      ⊞──◆ Mapping PurchaseOrderHeader.ShipToParty.Name
      ⊞──◆ Mapping PurchaseOrderHeader.ShipToParty.Location
         ──◆ Mapping PurchaseOrderHeader.ShipToParty.Contact.Name == PHDEST
         ──◆ Mapping PurchaseOrderHeader.ShipToParty.Contact.Communication.CountryDia
      ⊞──◆ Mapping PurchaseOrderHeader.ExtendedAmount == EXTAMT == 0
      ⊞──◆ Mapping PurchaseOrderHeader.CarrierParty
         ──◆ Mapping PurchaseOrderHeader.CarrierParty.PartyIDs.ID == PHCARR
```

Mapping an element Xpath

Note: If you are not using a BOD template you may skip this section.

To map an element Xpath name to a database mapping:

1   Select the Database Mapping node and then navigate to the Xpath View.

2   Select the row from the view that you want to map, then right-click and select **Context > Assign Xpath**.
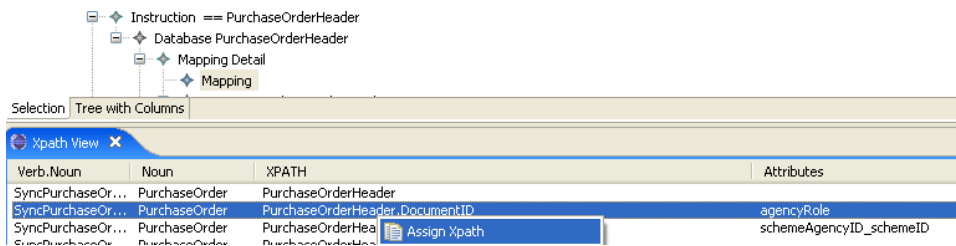
3 The mapping node was selected by clicking on it and then the PurchaseOrderHeader.DocumentReference was selected from the Xpath View. Choosing Assign Xpath from the menu sets the Element property on the Mapping node property page to the XPATH value. If the mapping does not work make sure the property page is open, this allows updating the Element. After the Element is set the Mapping node on the designer view displays the Xpath to the element as shown below.
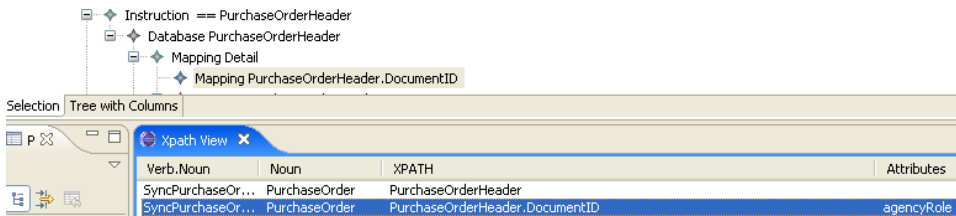
4 In this example, the Mapping node PurchaseOrderHeader.DocumentReference requires an attribute called **agencyRole**. When an attribute is added as a child of the Mapping this adds an attribute into the element at runtime when the BOD message is produced. Navigate to the same row in the Xpath View, right click, and select **Set Attributes**.
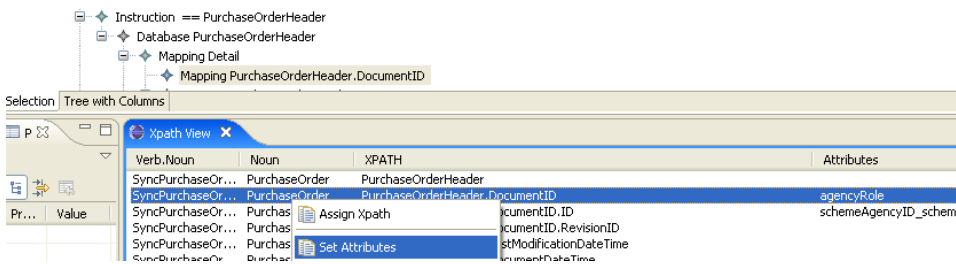
5 Selecting Set Attributes adds new child nodes called Attribute nodes as children of the Mapping node.
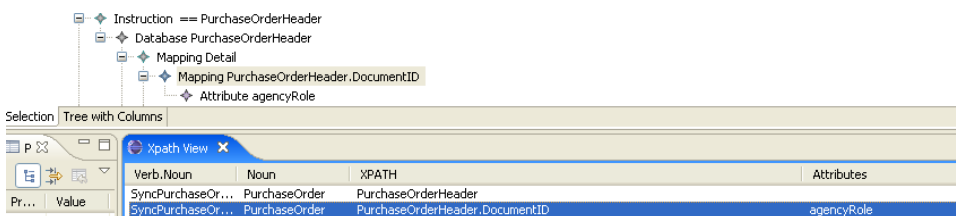
6 Open the property view for the Attribute node that was just added. The Name was set to **agencyRole**. Set the relevant properties for the Attribute. If the Attribute has a constant value set the property Value in the property view to the constant, for example, Customer. If the

attribute is a database field set the Database Field property. If cross referencing is required select the Cross Reference type. See "Adding attribute values" for more mapping information.

Note: Cross referencing is not supported by LX Connector process instructions.
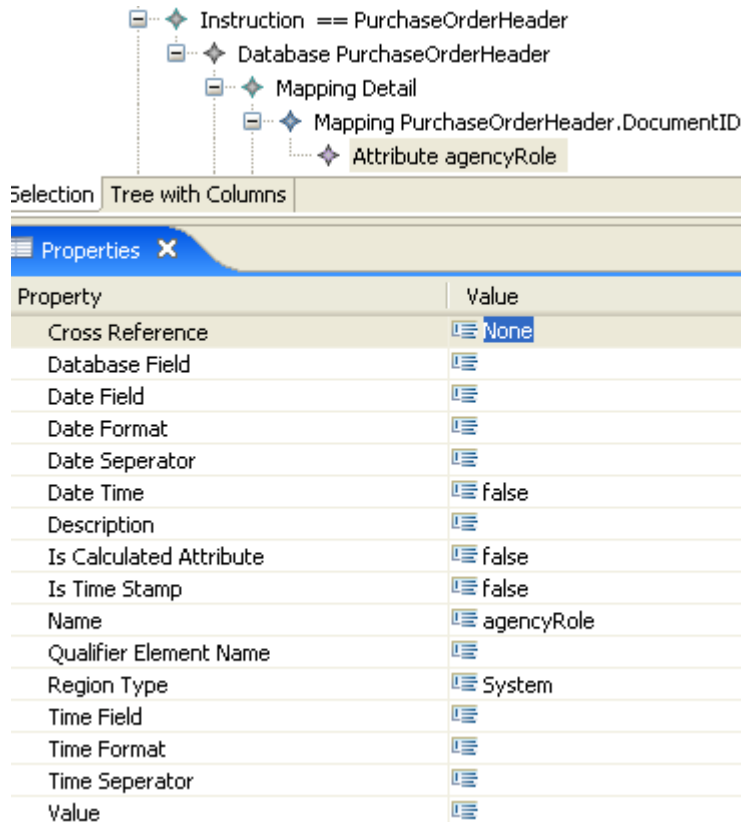
7   Set the **Database** Field in the property view for the Element. In this example, the Element (PurchaseOrder.DocumentReference) is a parent element that has an attribute. Because it is a parent element do not set a value in the property page for the Database Field. When the element is added into the BOD message it will be `<DocumentReference agencyRole="">` and will contain child elements if they exist, otherwise the element will be empty `<DocumentReference agencyRole=""/>`.

8   Add a **New Child > Mapping** node and then click it to select the node.

9   Navigate to the Xpath View and select XPATH PurchaseOrderHeader.DocumentID.ID.

10 To set the Element property in the property view, select **Assign Xpath** from the menu.

11 Navigate to the Database View to set the Database Field in the property page.

12 In the Database View select row PHORD, right click, and choose **Assign Field** from the menu. Assign Field sets the Database Field in the property view. At runtime the value assigned to the element ID is retrieved from an SQL statement which fetches the value for field HPH.PHORD. By mapping the Field to the Element, the ID is assigned the value retrieved from the field. For example, if HPH.PHORD was **1234**, at runtime the Element is added in the BOD message as shown below.

```
<PurchaseOrderHeader><DocumentID><ID>1234</ID></DocumentID></PurchaseOr
derHeader>.
```
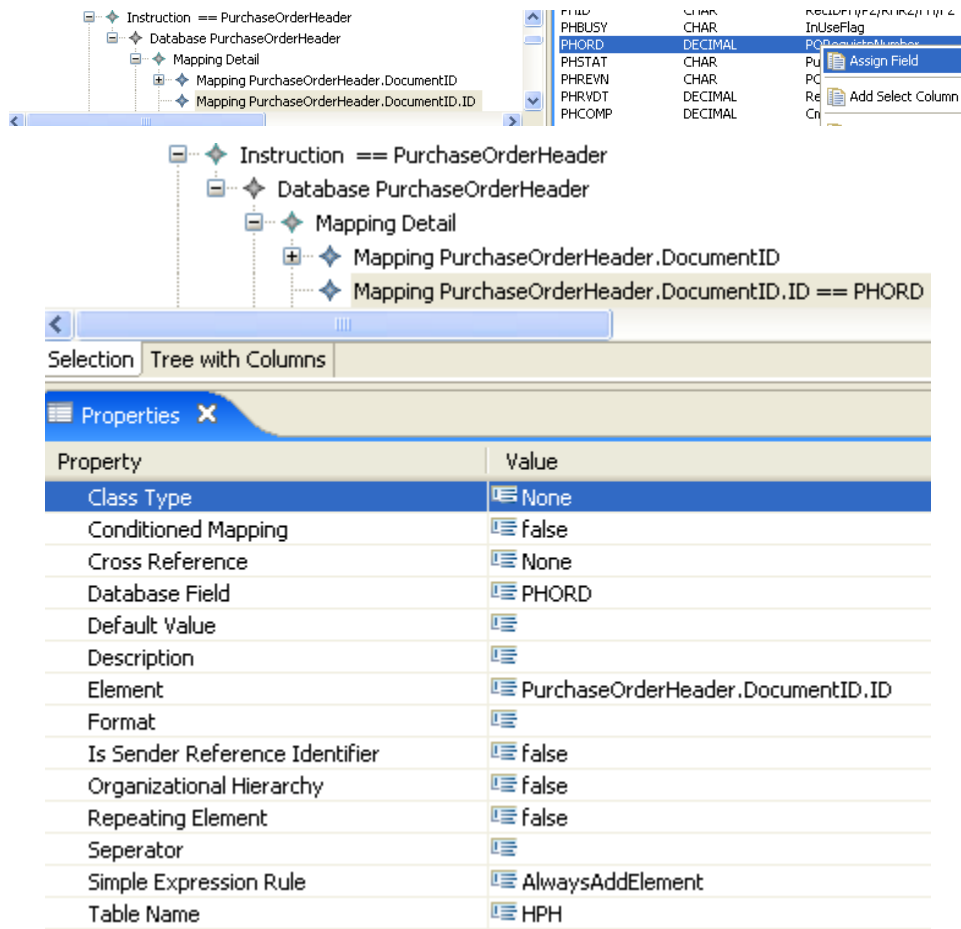

13 Instead of mapping a Database Field using the Database View explained in Step 6, you could map the value passed from the Exit Point message. In this example, the exit point process instruction defined earlier in this document, mapped name DocumentID to the purchase order number defined in the data structure. In the property view set the Database Field to DocumentID. At runtime the value will use the value passed in the exit point instead of the value from an SQL statement.
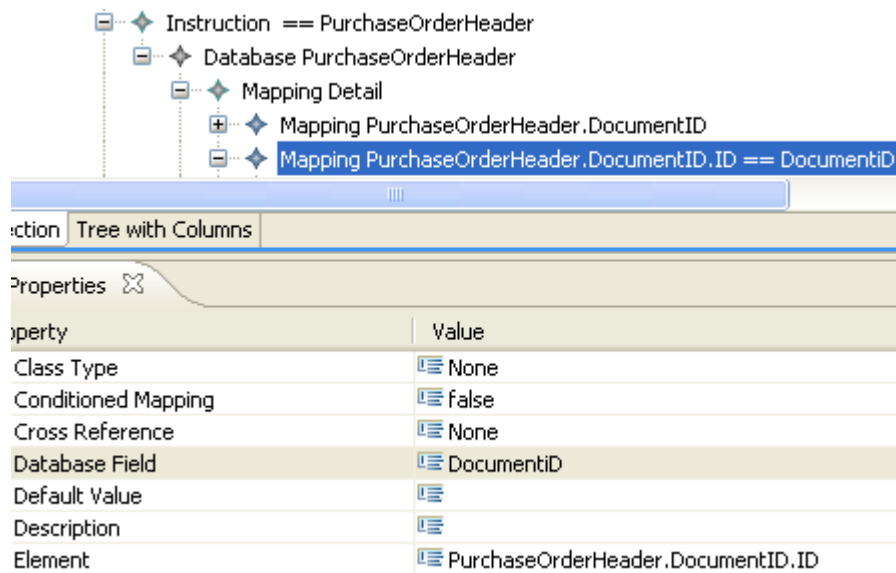
Adding attribute values

Note: If you do not have a BOD template you must add the value for the Name in the attribute property page manually. The Xpath View is only available if you are using a BOD template.

1   An element in a BOD message may require attributes. To add attributes to a Mapping node that defines an element select the Xpath row from the Xpath view, right click, and choose **Set Attributes**.

2   In this example, it is assumed that we are mapping a noun identifier that requires attributes accountingEntity, location, variationID and lid. All outbound messages must define a noun identifier but they are not required to have attributes. Add a new mapping node by selecting the Mapping Detail node and choosing **New Child > Mapping**.

    Note: Noun identifier attributes are used when ION routing is used. The noun identifier attributes are location, accountingEntity, lid and variationID. If you are producing a Sync BOD message, the variationID is a required attribute.

3   If you are usng a BOD template select from the XPath view the row having PurchaseOrderHeader.DocumentID.ID and choose **Assign Xpath** to set the element in the Property view. If not using a BOD template, manually add the Name for the Element in the property page.
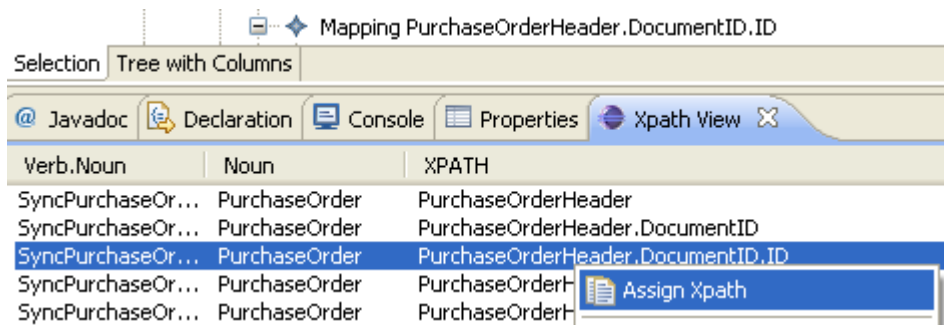
4 Select the element that was just added and add a new Child **Concatenation** field. We need to map the element to the field in the database but we also want to make sure that the number is always 8 digits. The **Concatenation** Field allows us to define this.

5 Select the **Concatenation** Field that was just added and then select the Field in the Database View using the **Assign Field**. This sets the Field and the Variable Type in the Concatenation Field property page.

6 Navigate to the property page and set property Add Leading Zeroes to true and Number of Characters to **8** as shown below.
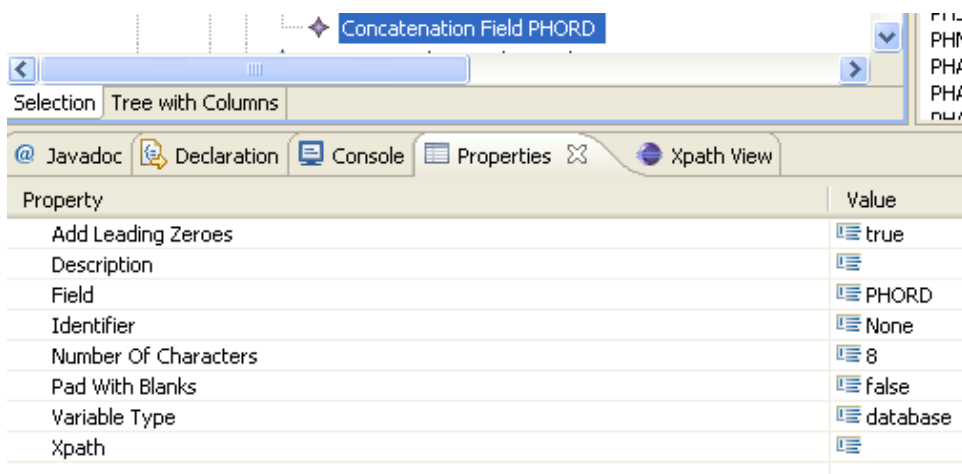
7 If you are using a BOD template add all attributes listed for this Xpath row to the PurchaseOrderHeader.DocumentID.ID node. Right click the row in the Xpath View and select **Set Attributes**.
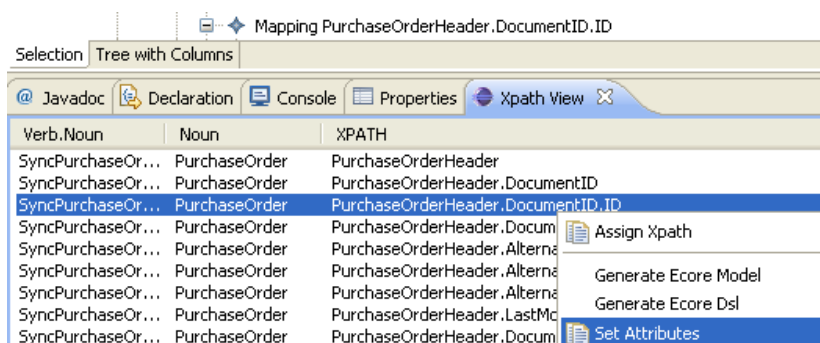
8 Because this Element is a noun identifier, open the property page for the Element PurchaseOrderHeader.DocumentID.ID and set the Property Is Sender Reference Identifier to **true**.
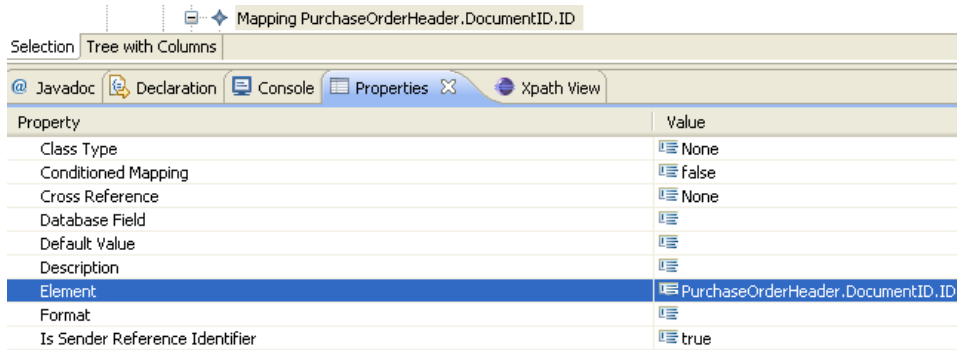
9 Open the property view for the Attribute node named accountingEntity that was added when you selected all attributes. Add child element Concatenation Field because there must be two characters and if not add leading zeroes.

10 Select the Accounting Entity attribute then select the field from the Database View to map.

11 Select Assign Field to set the field and variable type in the property page and then navigate to the property page to set the property Number of Characters to **2** and property Add Leading Zeroes to **true**.
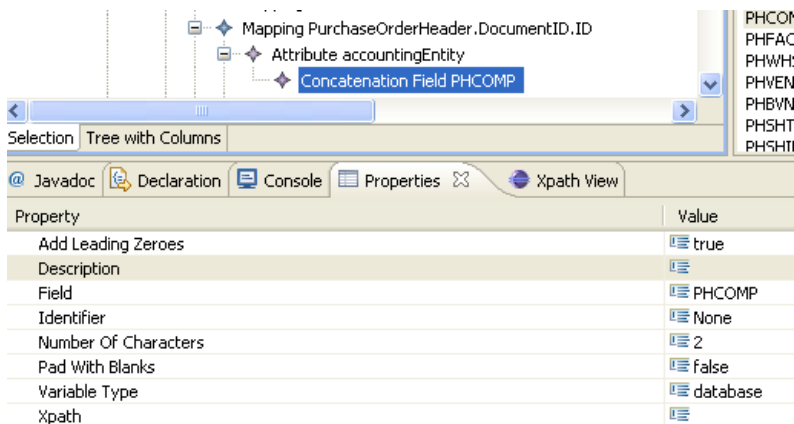
12 Next set the value for the location attribute. In this example, we need to add a prefix to the location. We want to concatenate the prefix to the value in PHCOMP. Select the Location Attribute node and add two new child nodes named Concatenation Field.

13 In the first Concatenation Field define the Variable Type as constant and set the Field to **1-**.
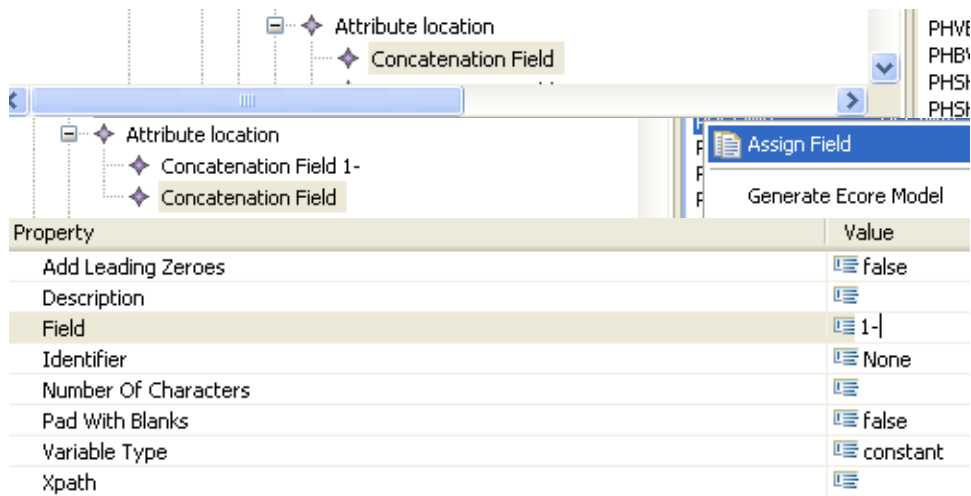
14 Select the second Concatenation Field and then from the Database View select the field to map the attribute to, and then select **Assign Field**. This sets the Field and the Variable Type in the properties page.
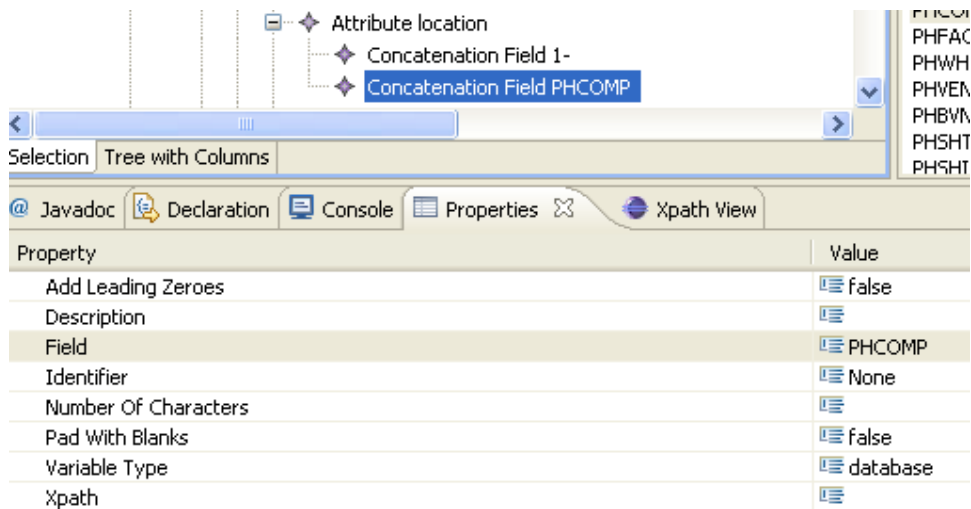
15 When producing LX Extension process instructions that use ION communications the noun identifier Element that produces a Sync message must have attribute variationID. To add the attribute select the attribute and navigate to the property page.

16 Select `variationID` from the Cross Reference property dropdown list.
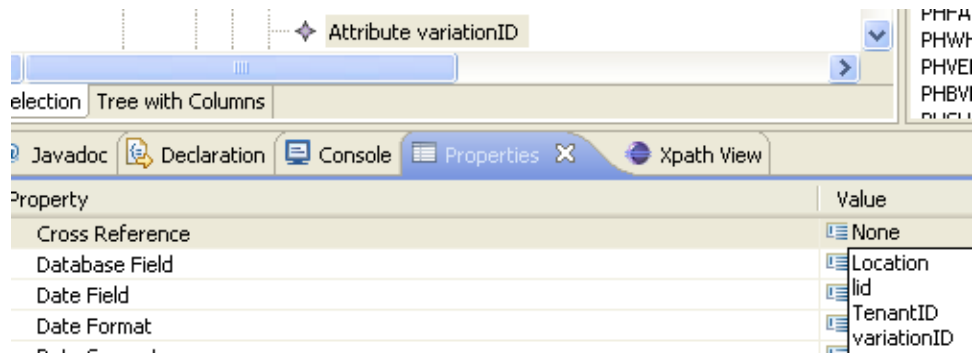
At runtime the variationID is calculated using the LX Extension cross reference file. The Cross Reference file is supported only when using the LX Extension using ION routing.
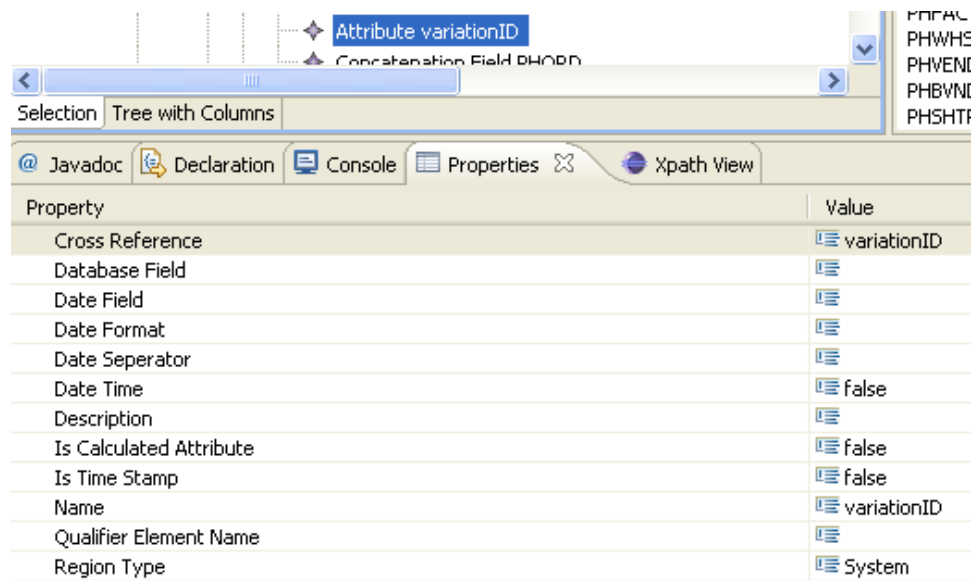
17 Since the element is a noun identifier it may also define the attribute lid. This also requires setting the CrossReference property. Select the lid attribute and set the Cross Reference to lid. This will set the URL at runtime.

The completed definition of the element that identifies the noun is shown below.
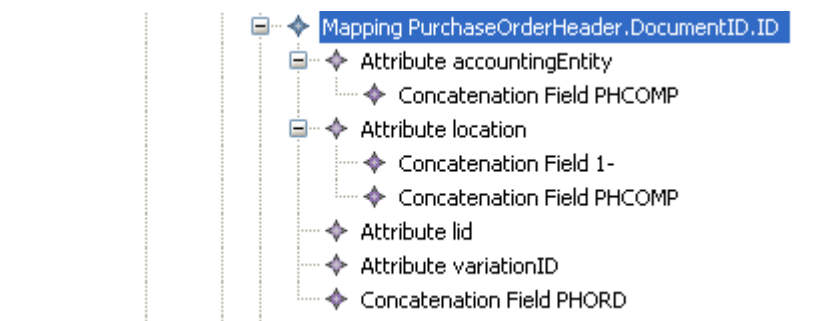
Adding an SQL instruction

Add an SQL instruction to retrieve the data required to build the BOD from LX.

1   Select the Database Instruction PurchaseOrderHeader node and add a child called Database SQL Statements. This allows you to create one or more SQL statements.

2   Use the SQL Builder View to build a SELECT statement.

3   To create a Statement element, select **Database SQL Statements** and add **New Child > Statement**.

4   Double click on the Statement to open the SQL Builder View.

5   In the SQL Builder view add columns from the Database View by selecting the column, right clicking, and selecting **Add Select Column**. The select list must be separated by commas. The column is prefixed by the table name when the Add Select Column is selected.
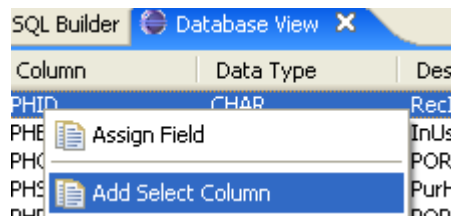
6   Select the function from the Select Functions list and use commas to separate the fields.
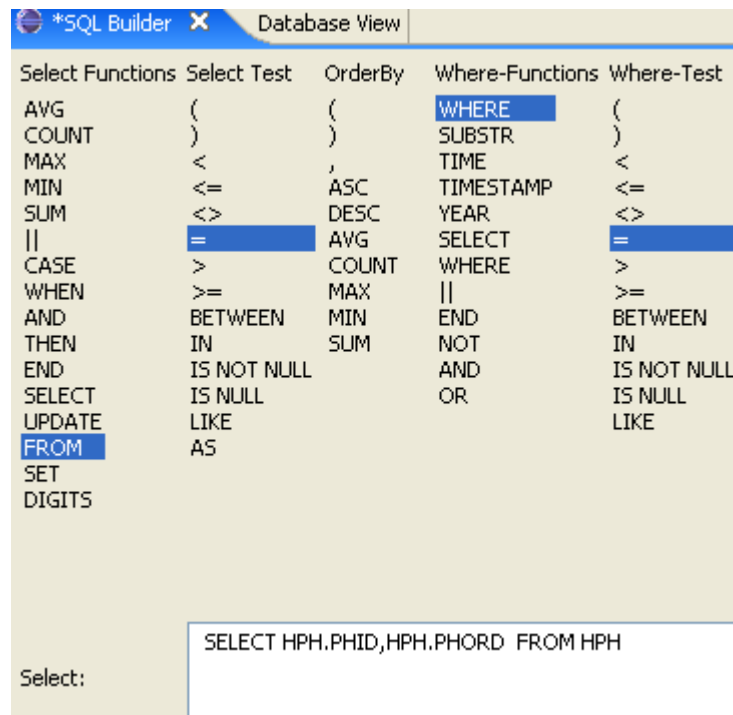
7   In the SQL Builder View set the where clause by selecting from the Where-Functions, Where-Test lists. You may also select a column in the Database View by right clicking and selecting **Add Where Column** into the where statement. To define a variable, prefix the value to the left of

the operation with a colon, for example, `:DocumentID.` In this example, DocumentID is defined by the exit point message.

8   Click **OK** to update the Statement property in the property view. Make sure the Statement node has been selected before clicking **OK**.
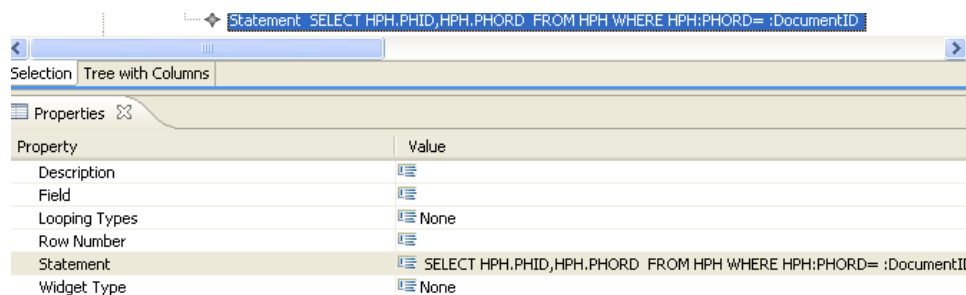
Generating the process instruction

To generate and save the PurchaseOrderOutbound process instruction and review the xml file:

1   Select the PurchaseOrderOutbound.developer project.

2   Right-click on the developer project and select **Infor LX Process Instruction** from the context menu.

3   Select **Generate Process Instruction** to create a `PurchaseOrderOutbound.xml` file.

4   Open the xml file with the System Editor. The process instruction contains all the instructions required to produce a BOD message.

Creating an outbound process instruction with conditions

The outbound process instruction that was created in the preceding section was simple. Process instructions may be quite complex and require several Condition nodes or several child Conditional Instruction nodes. The Conditional Instruction node may require child nodes that allow decisions based on the BOD message produced, it may require invoking several Instruction nodes, or it may require invoking API type instructions.

In this example, we will add complexity. We will add a requirement that before processing the header we must check that the purchase order is an active order.

In this example, we will use the same Condition node SELECTPOBOD to expand upon. The BOD message is produced only for active purchase orders so an If Condition node is needed.

1   Select the Conditional Instruction node and add **New Child > If Condition**.

2   Chapter 2 defines the properties of the If Condition node. To check if the order number is active, select the `If` Condition node that was added and set property Condition Type to `If`.

    a   Set the expression using the Expression Builder view introduced in Chapter 1. We want to check a name that is defined by the Exit Program that loaded this process instruction. Open the PUR500BEXIT01 Model Object to find a name that allows us to check the purchase order status.

    b   In ARG 4, a value for StatusCode is passed to the generated process instruction. We can check the value of this by setting this in the expression as shown in screens below.

Select Condition   Select Operation

if   ==   (EQUAL)
                            !=   (NOT EQUAL)
                            >   (GREATER THAN)
                            >=   (GREATER OR EQUAL)
                            <   (LESS THAN)
                            <=   (LESS OR EQUAL)
                            &&   (AND)
                            || (OR)
                            *BLANK (IsBlank)
                            !   (Not)
                            ((Open Parenthesis)
)   (Close Parenthesis
                            +   (Addition)
                            -       (Subtraction)
                            /   (Division)
                            *   (Multiplication

Expression:

((StatusCode==PH))

Validation Result:

OK   Cancel

Clear   Validate

3 Add child elements to the If Condition node that are executed if the expression evaluates to true. In this example, we will add the Instance Name node that invokes the execution of building the PurchaseOrderHeader. The screen below shows the If Condition node with the Instruction Name node added as a child.
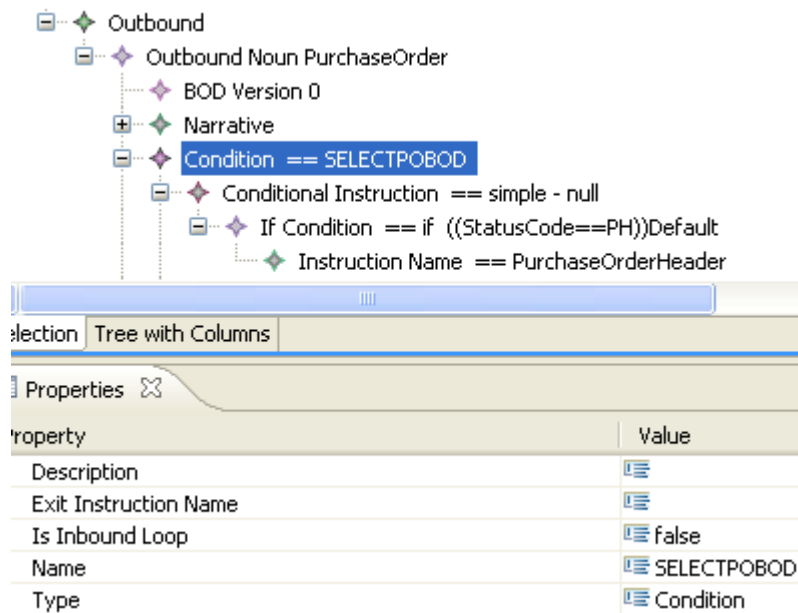
4 This is the last step in a project that contains a single Condition Instruction used to produce a PurchaseOrder BOD. Use **CTRL + S** to save the project.

Checking the process instructions

To review the process instructions, open the xml file with the System Editor. The process instruction includes all instructions that were defined in the designer view.

```
<Instruction name="SELECTPOBOD" type="Condition">
– <Condition executeMethod="none" expression="((StatusCode==PH))" type="if">
      <ExecuteInstruction name="PurchaseOrderHeader" />
  </Condition>
```

```
– <Elements looptype="false" name="PurchaseOrderHeader">
  – <Element classtype="NormalAttribute" expressionRule="AlwaysAddElement" name="DocumentID" organizationHierarchy="false"
      srid="false" translation="None" widget="NormalAttribute" xpath="PurchaseOrderHeader.DocumentID">
    – <NormalAttribute>
        <Attribute name="agencyRole" value="Customer" xpath="PurchaseOrderHeader.DocumentID@agencyRole" />
      </NormalAttribute>
    </Element>
  – <Element expressionRule="AlwaysAddElement" field="HPH.PHORD" name="ID" organizationHierarchy="false" srid="true"
      table="HPH" translation="None" xpath="PurchaseOrderHeader.DocumentID.ID">
    – <NormalAttribute>
        <Attribute name="lid" translation="lid" xpath="PurchaseOrderHeader.DocumentID.ID@lid" />
        <Attribute field="HPH.PHCOMP" name="accountingEntity" translation="AccountingEntity"
          xpath="PurchaseOrderHeader.DocumentID.ID@accountingEntity" />
        <Attribute field="HPH.PHCOMP" name="location" translation="AccountingEntity"
          xpath="PurchaseOrderHeader.DocumentID.ID@location" />
      </NormalAttribute>
    </Element>
```

Chapter 5 Additional capabilities

This chapter contains several examples of how to use the LX ION PI Builder to add nodes to the Model Object Tree view.

Introduction

Outbound Model Object projects are created to produce outbound Bod messages that are stored in an outbox. When building an Outbound Model Object an Outbound Noun is added as the first child of the root node named Outbound. In most outbound model object projects the Outbound Noun will have one or more child Condition nodes, one or more Instruction nodes, a single Narrative node and if the model object is being created for a SOA Integration that uses ION routing a single BOD Version and three or four Namespace nodes.

This screen shows a project that when generated produced an ItemMaster BOD.
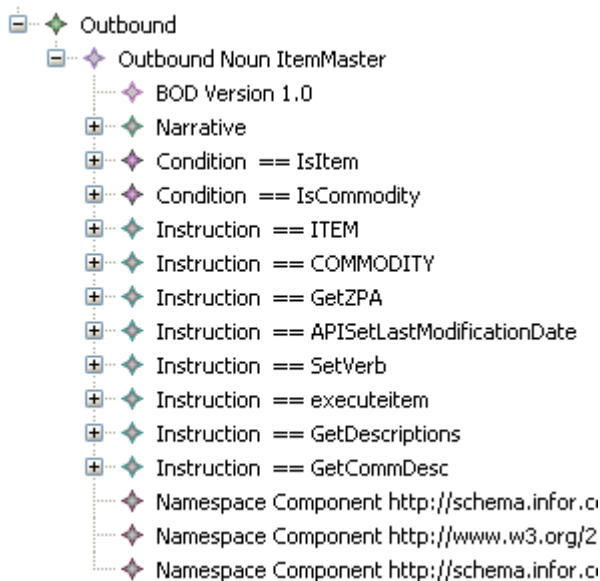
When building an Inbound Model Object a Noun node is added as the first child of the root element Inbound. In most inbound model object tree views the Noun node has a single Narrative node, one or more Condition nodes and one or more Instruction nodes.

This screen shows an inbound Model object that when generated produces a process instruction that processes Shipments into LX.
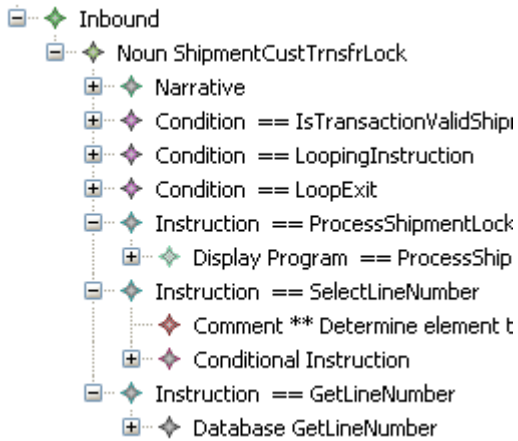
Sample Model Object tree view of entry point

All Outbound and Inbound Model Objects produce a process instruction. Each project must include a node that is defined as the entry point. When the process instruction is generated this is the instruction that is invoked when the process instruction is loaded.

Exit Points and Trigger Model Object projects produce exit point process instructions which have BOD Element nodes. The BOD Element node defines the Entry Point instruction Name as well as the name of the process instruction loaded at runtime.

The outbound Model Object tree view shown below has two entry points both defined using a Condition Node. The first Condition is named IsCustomerOrder and the second Condition is named IsResupplyOrder. Regardless of which entry point is invoked each child contained by the Condition node is executed as an instruction that in the end builds a BOD.

```
⊟ ◆ Condition  == IsCustomerOrder
    └── ◆ Comment ** From Pick Release, step can be executed from the last
    ⊟ ◆ Conditional Instruction
        ├── ◆ Instruction Name  == CheckPickWarehouseType
        └── ◆ Instruction Name  == SetBatchInfo
    ⊟ ◆ Conditional Instruction
        ├── ◆ Comment ** This order type BOD will only be sent if warehouse
        ├── ◆ Comment **** "0" means SYS500X1 was not found in call stack
        ⊟ ◆ If Condition  == if  ((IWM.WMMNWH==5)&&(SYS500X1==0))D
            ├── ◆ Comment ** Read ZPA to get Picking Quantity Basis.
            ├── ◆ Comment ** Read ZPA to get Generate OLM loads for WM
            ⊞ ◆ Conditional Instruction
            ⊞ ◆ Conditional Instruction
            ⊞ ◆ Conditional Instruction
        ⊞ ◆ If Condition  == elseDefault
⊟ ◆ Condition  == IsResupplyOrder
    └── ◆ Comment ** From Pick Confirm, step can be executed from the last
    ⊟ ◆ Conditional Instruction
        ├── ◆ Comment **SQL to retrieve order type and warehouse type.
        ├── ◆ Instruction Name  == CheckOrderType
        └── ◆ Instruction Name  == SetBatchInfo
    ⊟ ◆ Conditional Instruction
        ⊞ ◆ If Condition  == if  ((ECH.HDTYP==9)&&(IWM.WMMNWH==5)
        ⊞ ◆ If Condition  == elseDefault
```

Inbound Model Objects produce inbound process instructions. In this case the inbound process instruction is loaded when a BOD message is read from an inbox. The entry point to the inbound process instruction is defined in the Noun property Pi Entry Point Name. If this is not set the process instruction will not work.

In the screen below the property page for the Noun shows the PI Entry Point Name IsTransactionValid. This is a Condition node in the tree view that contains several Conditional Instruction nodes used to define other nodes. The screen shows that the Instruction Name loads the Instruction in the Tree View having this same name. In the example, we see this is a Display Program that uses screen navigation.

Samples of outbound element mappings

This section contains several samples of how to map an Element name to a Database Field used to retrieve a value.

All of the samples in this section assume that an Instruction node has been added as a child node of the Outbound Noun Node. This Instruction node contains a single Database node which contains two child nodes, a Mapping Detail node and a Database SQL Statements node.

The Mapping Detail node is a container of Mapping nodes. Each Mapping node is used to map Element names to Database Field values. In these examples, all database fields are retrieved using Sql Statements.

The Database SQL Statements node is a container of Statement nodes. Each Statement retrieves data from LX files into a result set. The result set contains the values that are used to map to the Database Field of the Mapping node.

Sample 1

This example shows how to define a Mapping node that produces an element in a BOD. In this example, the node will use the Class Type property set to Enumerated as well as the Database Field to map a value to the element.

The Mapping node property page used to produce the element into the BOD is shown below.

- The Element that is added into the BOD message is an xpath to the name Code.

- The Database Field is set to column **PHID**.

- The Table Name is set to **HPH**.

- The value for this field (**HPH.PHID**) is contained in a result set that was executed using a Statement contained in the Database SQL Statements node.

- The Class Type is set to **Enumerated**.
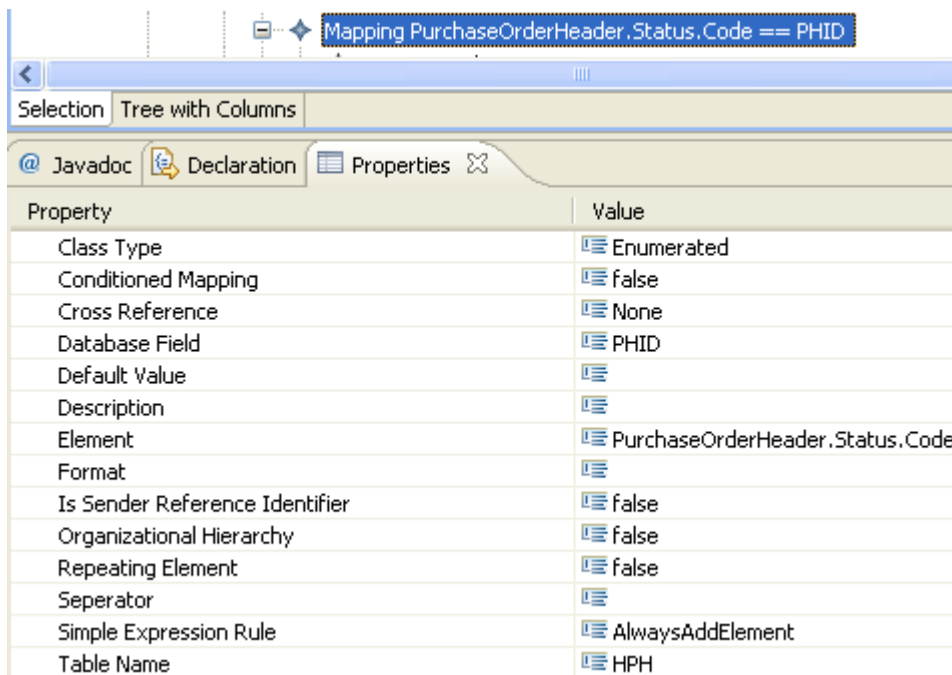
Since the Class Type is Enumerated the mapping requires child Enumeration nodes that map an LX value to a BOD value.

1   Select the Code Mapping node and add New Child Enumeration.

2   Select the Enumeration node and set the properties.

3   In the property view set the LX Value to **PH** and set the Bod Value to **Open**.

4   Add another new child Enumeration and in the property view of the Enumeration set the LX Value to PZ and the Bod Value to Closed.

     The screen below shows two Enumerated Child nodes added to the Mapping Node. When the Element is added to the BOD the value for HPH.PHID is extracted from the result set. If that

value is **PH** then the value for the Element added into the BOD message is Open. If that value is **PZ** the value is set to `Closed`.
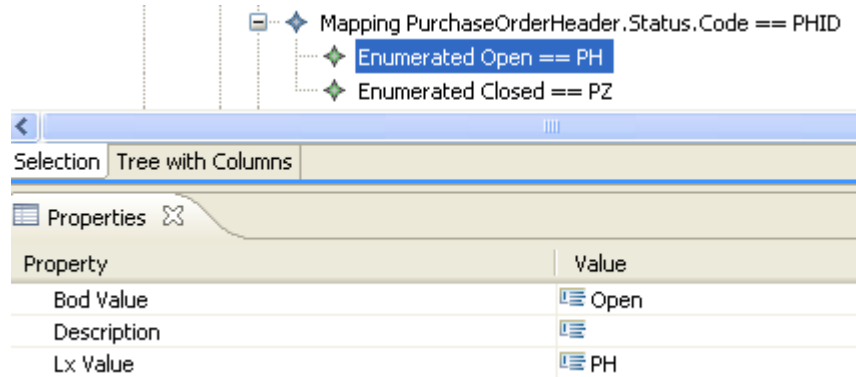
Sample 2

This sample shows how to add an attribute into an Element using the Mapping node.

- Add a New Child Mapping to the Mapping Details node
- Set the Element property in the Mapping node
- Do not set the Database Field
- Do not set the Table Name
- Set the Class Type to the default none

The property page for this Mapping node is shown below.
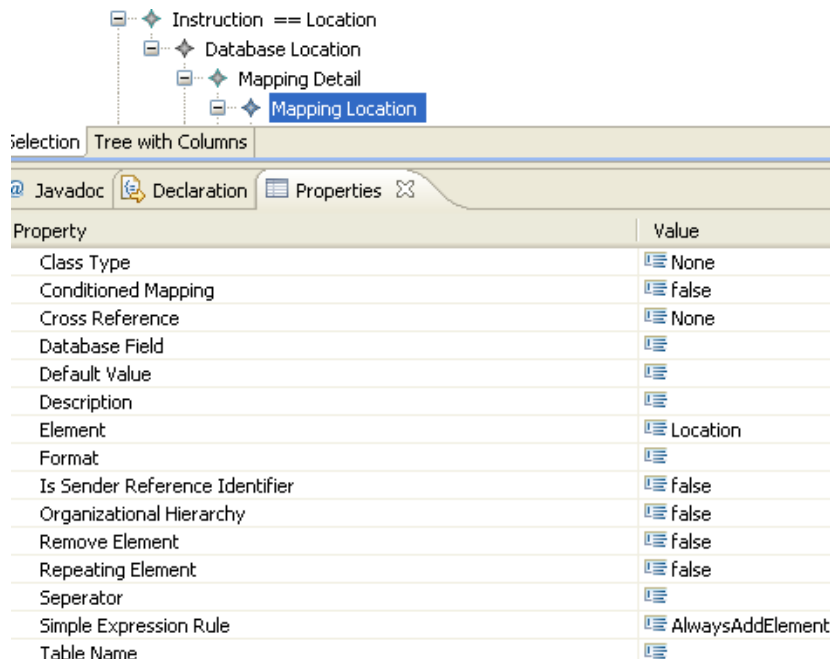
Select the Mapping node, right click and select **New Child Attribute**. Set the properties for the Attribute. In this example, the value for the attribute is constant.

- Set the Name property to the name of the attribute.
- Set the Value to the value to assign the attribute.
- The property page for the Attribute is shown below. The Name of the attribute add to the Location element defined above is type and the value for the attribute is "Warehouse".

In this example, since the Element has no value but does have an attribute at runtime the BOD message will contain `<Location type="Warehouse">`.

Sample 3

This example shows how to add an attribute to an element that is assigned a sequential value. The example also shows how to define the element as one that may repeat in a BOD message. Select the Mapping Detail node and add New Child Mapping

- Set property Element to the name of the element that repeats in the BOD.
- Set the Repeating Element to **true**.

The property page for the Mapping node is shown below. The Element is ContractSchedule and the Repeating Element property is set. This implies that our BOD message will have one more child elements called `<ContractSchedule>`.

Select the Mapping node and add New Child Attribute. Open the property page and set it to be a sequential attribute.

- Set the name to sequence
- Set the value for the attribute using a Work Element.

Select the Attribute node and add New Child Conditional. Then select the **Conditional Instruction** and add **New Child Work Element**.
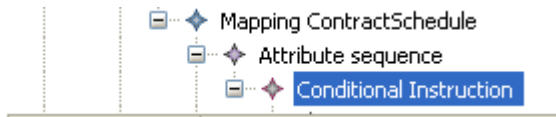
- Set the properties for the Work Element to increment the value for the sequence by one for each ContractSchedule added to the BOD.
- Set the Calculate Value to `true`.
- Set the Variable Type to `Index`.
- Set the Value to `1`.

The property page settings are shown below. The value assigned to the sequence will start with 1 and increment by 1 for each child added into the BOD.

After generating the process instruction the runtime adds an incremented value for each occurrence of ContractSchedule, For example:

    <ContractSchedule sequence="1"></ContractSchedule>

    <ContractSchedule sequence="2"></ContractSchedule>

Sample 4

The example shows use of variables in an If Condition node. The variables are set using a SELECT COUNT(1) AS variable Statement.

The Statement shown below is defined in the Database SQL Statements and the variable called COUNTLOTS is stored.

    (SELECT COUNT(1) FROM ELA B WHERE B.AORD=:EPOrderNbr AND B.ALOT <> ' ' AND
    B.ALINE=IPP.PPORLN) as WK1COUNTLOTS FROM IPP

Select the Mapping detail and right click to add a New Child Mapping. Set the properties of the Mapping node.

- Set the Element to an xpath that is written into the BOD Message ShipmentItem.SerializedLot.Lot.
- Set the Repeating Element to **true**.
- Set the Simple Expression rule to **AddElementifTrue**.

In this sample, we have added Mapping node Lot into the Mapping Detail. The property page for the Mapping is shown below.
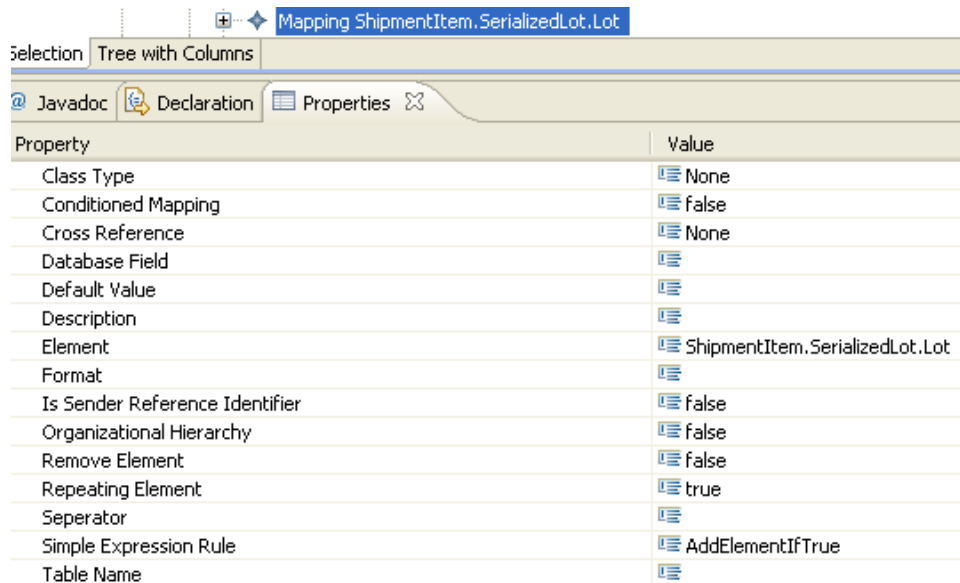
Since the Single Expression Rule is set to **AddElementIfTrue** select the Mapping node and add New Child If Condition. Set the properties for the If Condition.

- Set the Condition Type to **if**.
- Set the Expression to check the value of the variable. **(WK1COUNTLOTS>=1)**
- Since the variable is from a SELECT COUNT Statement set the **Available Methods** to **Count**.
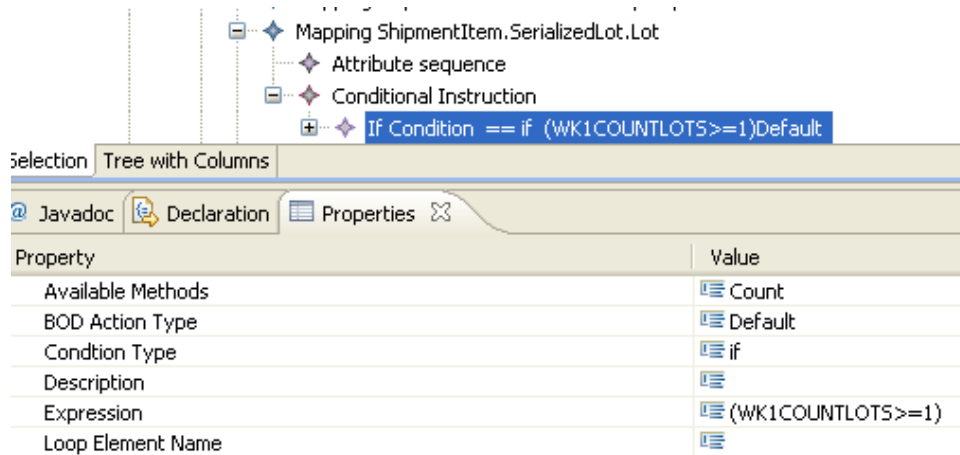
Since the Simple Expression Rule is **AddElementIfTrue**, if the expression evaluates to true then the Lot is added into the BOD.

    <ShipmentItem><SerializedLot><Lot sequence=""></Lot></SerializedLot></Shipment>

If the expression is false the BOD message will have:

    <ShipmentItem><SerializedLot/></ShipmentItem>.

Note: In some cases, the Expression contains both a comparison to a count and a comparison to a non-count variable. In this case, do not set the **Available Methods** to **Count**. Instead, set the **Available Methods** to **none** and prefix the variable that is retrieved with the COUNT(*) function with a colon (**:**). For example, set the expression to:

    ((PHID==RZ)&&(:PXCOUNT==0))

Sample 5

This sample shows how to use the Repeating Element property. The screen in Sample 4 shows that the Lot is a repeating element in a ShipmentItem parent. The screen below shows that if WK1COUNTLOTS, a variable defined in a SELECT COUNT Statement node, is one or greater than any nodes that are child nodes of the If Condition executed.

Child nodes added to the If Condition are child elements of the <Lot>. To add children into a Repeating Element <Lot>:

1   Select the If Condition node added in sample 4 and add New Child Mapping.

2   Set the Element in the property view of the child Mapping node to **ShipmentItem.SerializedLot.Lot.LotIDs.ID**.

3   Set the Database Field to ALOT and the Table to ELA.

4   To add a second New Child Mapping to the **If** Condition, select the mapping node and set the Element in the property view to **ShipmentItem.SerializedLot.Lot.Quantity**.

5   Set the Database Field to **LQALL**.

6   Set the Table Name to **ELA**.

At runtime if the If condition evaluates to true then Lot will be added to the BOD message and it will contain an ID and a Quantity for the lot as shown below.

    <SerializedLot><Lot><LotIDS><ID></ID></LotIDS><Quantity></Quantity></Lot></SerializedLot>

Sample 6

This sample shows how to add an element into the Exit Point message at runtime. The Exit Point message is created using data from the LX Event data.

- When adding into the Exit Point data a Work Element is used and the xpath is always Noun.Criteria.Equal.Element, where Noun is the name of the Outbound message, and Element is the name that will be used in the Model View object.

- Since the exit point data is used to build a BOD message you need to add it before starting to build the BOD.

- Add a Work Element into the entry point condition before any instructions that build the BOD.

Select the exit point condition and add a Conditional node.

In this case data will be added to the Exit point data only for a specific condition. Add an If Condition node as a child of the Conditional Instruction.

- Set the Condition Type to `if`.
- Set the Expression `((ExitPoint==EXITGEN)||(ExitPoint==EXIT01))`.
- Select the If Condition node and add a Work Element that sets data into the Exit Point message if the expression is `true`.

The property page for the Work Element is shown below.

- Set the Xpath Element to `ReceiveDelivery.Criteria.Equal.InventoryFlag`.
- Set the Variable type to `constant`.
- Set the Value to a constant value of `0`.

At runtime the InventoryFlag is added into the message passed by the exit point as shown below.

> <ReceiveDelivery><Criteria><Equal><InventoryFlag>0</InventoryFlag></Equal></Criteria></Receive
> Delivery>

Once the InventoryFlag is added into the exit point it can be used elsewhere in the project such as a variable in SQL or expression statements as shown in the following SQL statement. Note that because it is used in an SQL statement the variable must be prefixed with the variable indicator character which is the colon (:).

> SELECT ITH.THNII, ITH.TWHS, ITH.THRNO, ITH.THCTM, ITH.THCDT, ITH.TREF, ITH.THWS, ITH.TTYPE,
> ITH.THTIME, ITH.TVEND, ITH.TPROD, ITH.TLOT, ITH.THADVN, ITH.THLIN, ITH.THMRB, ITH.TQTY,
> ITH.THTUM,ITH.THTOTW, ITH.THCNTR, ITH.THADVN, ITH.TSEQ, ITH.TTDTE FROM ITH WHERE ITH.TPROD
> =':ItemID' AND ITH.THNII =':InventoryFlag' AND ITH.TSEQ = :TransactionHistorySequence

Defining database Statements that loop

Certain BODs require header and detail information be generated by a process instruction. For example, a Purchase Order contains header and line information, a Shipment contains header and ShipmentItems. For these types of BODs, SQL statements must be executed to retrieve data for each PurchaseOrderLine or Shipmentitem. To write the BOD correctly requires creating Model Objects that use Statements that can loop through each returned row, write all data for that row into a BOD message, and then move on to the next row. To provide this capability the LX ION PI Builder allows developers to set looping information on a Statement node. There are Looping Types defined in the property view for a Statement node. These are the Looping Types:

- None
- Forloop
- Foreachloop
- Iteraterows (not supported)

Use the forloop type to create a child element named PurchaseOrderLine for each line retrieved by the Statement. For example, use this on the SQL statement that retrieves all lines. The forloop must be the first Statement contained in the Database SQL Statements node.

In this example, all purchase order lines are added to the PurchaseOrder BOD message. This requires creation of a Statement node that retrieves all fields required to define a single line and setting the Looping Type property for this Statement to **forloop** as shown in this screen:
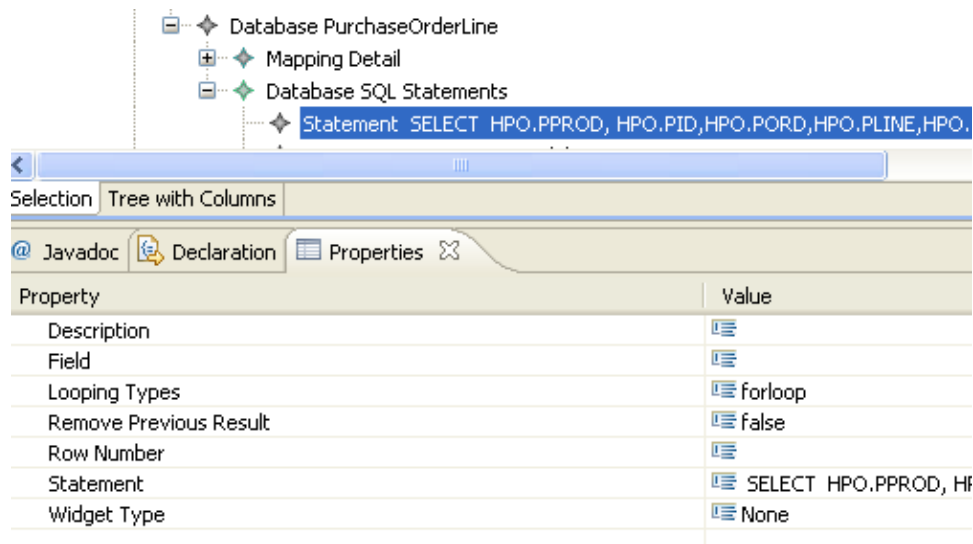
Subsequent Statements contained in the Database SQL Statements node should set the looping type to **foreachloop**.

For this example, we add a new Instruction to define a PuchaseOrderLine. Since a PurchaseOrder can have many PurchaseOrderLine elements in a BOD message the Instruction node property Is Loop Type must be set to true. The property page for the PurchaseOrderLine instruction is shown below. The Instruction contains child node Database. The figure below shows that the Database node Name property is the same as the Instruction Name which is a requirement.
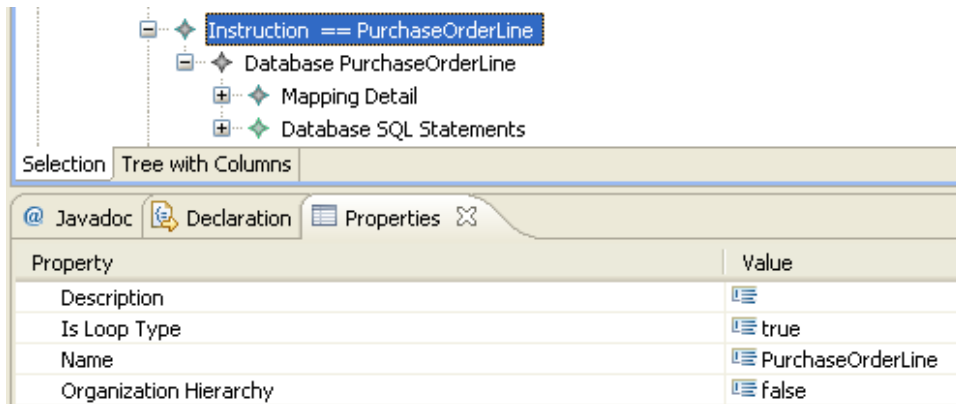
The Database node contains Mapping Detail and Database SQL Statements. Mapping Detail is a container that holds Mapping nodes that are used for Mapping an Element to a value. Database SQL Statements is a container of Statement nodes. The container has all of the SQL statements needed to successfully build a PurchaseOrderLine.

To add all lines into the BOD message:

1   Select the Database PurchaseOrderLine and add new child Database SQL Statements.

2   Add new child Statement. Double click on the statement node to open the SQL builder. Use this to build an SQL statement.

The SQL statement shown below was built for this sample. Note that the SQL statement includes a variable shown in bold print. All variables in an SQL statement must be prefixed by the variable indicator (:).

SELECT CASE HPO.PONIIT when '1' then HPO.PPROD CONCAT HPO.PONIIT ELSE HPO.PPROD END AS PPROD,
HPO.PID,HPO.PORD,HPO.PLINE,HPO.PCLAS,HPO.PVITM,HPO.PODESC,HPO.PQORD,HPO.PQREC,HPO.PUM,HPO.POCUR,HPO.PWHSE,HPO.PECST,HPO.POSRCE,HPO.POATTN,HPO.POADR1,HPO.POADR2,HPO.POADR3,HPO.POADR4,HPO.POADR5,HPO.POADR6,HPO.POSTE,HPO.POCOUN,HPO.POZIP,HPO.PSHIP,HPO.PONAME,HPO.PONIIT,HPO.PBUYC,HPO.POSHTY,HPO.PDDTE,HPO.PODTME,HPO.PODEST,HPO.PUMCN,HPO.POFAC,HPO.PGLNO,HPO.POCONT, HPO.POCWUM FROM HPO   WHERE HPO.PORD = :**DocumentID** and PID like 'P%' ORDER BY HPO.PLINE

3   The SQL statement above retrieves all lines for the PurchaseOrder specified by variable DocumentID. To add each line into the BOD Message, set the Looping Type in the Statement to forloop. The forloop will create a new `<PurchaseOrderLine>` element for each row returned from the SQL statement. All elements are added into the PurchaseOrderLine a row at a time until there are no more rows. If ten rows are retrieved from the SQL statement, then there will be ten `<PurchaseOrderLine>` child elements contained in the BOD message that gets produced.
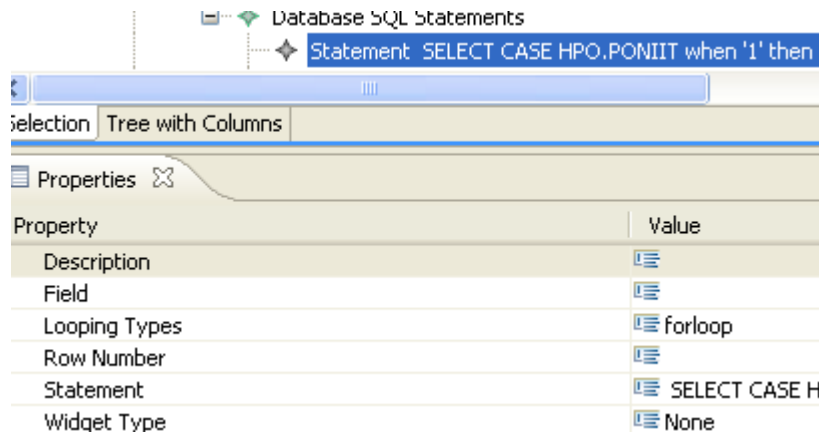
4   Add additional SQL statements to retrieve information about a line. Each additional statement must have the Looping Type foreachloop set in the property view for the statement. For example, to add notes to a purchase order line you could add the SQL statement shown below. The variables in the SQL statement are shown in bold. The foreachloop indicates the SQL statement is getting Notes for the current line being processed.

SELECT ESN.SNDESC AS LPRTNOTE FROM ESN WHERE ESN.SNTYPE='P' and ESN.SNCUST = **:HPO.PORD** and ESN.SNSHIP=**:HPO.PLINE** and ESN.SNPRT = 'Y' ORDER BY ESN.SNSEQ
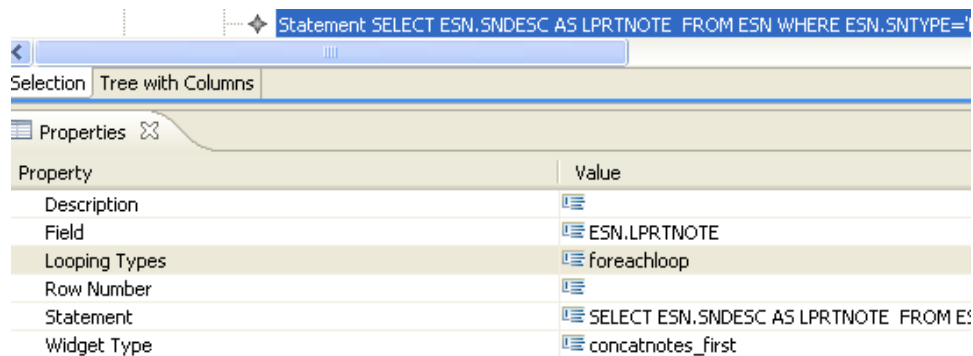
5 Use the Widget Type called `concatnotes_first` to concatenate the notes to return a single <Note> in the BOD message. See "Using widgets."

Using widgets

The property view for Statements node has a Widget Type property. The property is set using a drop down list. The widget types are very specific and were created for very special features of current integration projects. The types are explained below.

Widget Type	Description
None	Default
Concatnotes	This widget is specifically for adding notes into a BOD message. The widget concatenates the value of the field retrieved from the SQL statement. It can concatenate one field. For example, SELECT ESN.SNDESC as LPRNOTE from ESN would concatenate the value retrieved from field SNDESC. A single Note with the concatenated value is written into the BOD message.
Concatnotes_last	This widget is specifically for adding notes into a BOD message. When selected, the widget concatenates the value of the field retrieved from the SQL statement result set using fields returned from the last row only. It can concatenate one field. For example, SELECT ESN.SNDESC as LPRNOTE from ESN would concatenate the value retrieved from field SNDESC. A single Note with the concatenated value is written into the BOD message

Widget Type	Description
Concatnotes_first	This widget is specifically for adding notes into a BOD message. When selected, the widget concatenates the value of the field retrieved from the SQL statement result set using fields returned from the first row only. It can concatenate one field. For example, SELECT ESN.SNDESC as LPRNOTE from ESN would concatenate the value retrieved from field SNDESC. A single Note with the concatenated value is written into the BOD message
simpleexpression	The Simple Expression is deprecated and should not be used. In earlier integration projects this widget was used to evaluate the SQL statement using a simple expression routine. This widget has been deprecated and replaced by the If Condition node.

Defining the verb

Caution: The Verb node is required by LX Extension integrations that use ION connectivity.

Every outbound message that uses ION connectivity must define a verb instruction. The outbound project can have multiple instructions that define verbs for different conditions. Currently the LX Extension supports only the Sync and Process verbs. ION Integrations require that the Verb have a child element named TenantID. Optionally the verb may contain properties AccountingEntityID or LocationID.

A Verb instruction node is used to set the verb properties into the BOD message. See Chapter 2 for the properties available for the Verb.

Adding verb information

1   To add verb information, create a new child Instruction:

2   Select the Outbound Noun node. Add a new Instruction node.

3   Set the Instruction Name property to **SetVerb**.

4   Select the Instruction node and add new child Verb.

5   Select the Verb node and in the property view set the Verb property by selecting from the drop down list. The choices are Sync, Process, Acknowledge, and Show. If LX is the SOR for this BOD select Sync; if LX is not the SOR select Process. Acknowledge and Show are not supported at this time.
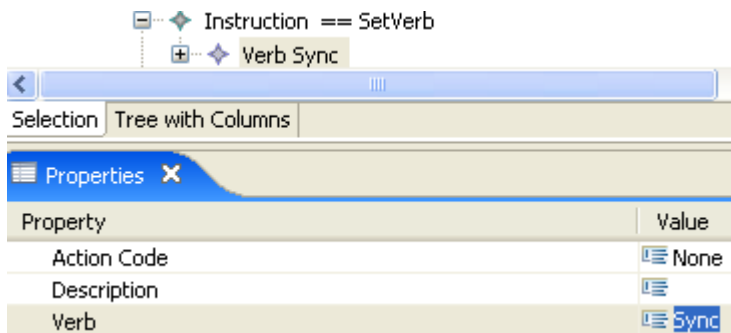
The Verb is added at runtime into the BOD message DataArea. For example, if the Verb selected is Sync the message will contain `<DataArea><Sync/>`. The Verb node allows you to add properties of the verb into the BOD message using Verb Element nodes.

Adding verb properties to the BOD message

To add properties of the verb into the BOD message:

1   Select the Verb node and add new child Verb Element.

2   The TenantID is a required property for all BOD messages. To set this property:

   a   Select the Verb Element.

   b   In the property view, select **TenantID** from the **Element Name** drop down list in the Value column.

   c   The TenantID must be defined in the SOA Cross Reference program (SYS127) as part of the Integration setup. Values defined in the Cross Reference are retrieved if the Verb Element Cross Reference property is set to a value other than none. To retrieve the BOD Value given to the TenantID, select **TenantID** as the value from the **Cross Reference** drop down list.
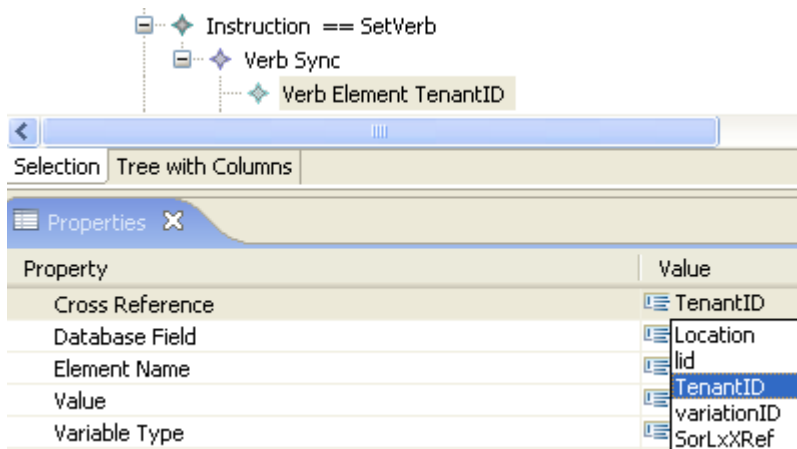
3   If the BOD message must include the AccountingEntityID create a second child Verb Element.

   a   In the property view for this node set the Element Name to **AccountingEntityID**.

    b   If this accounting entity is defined in the SOA Cross Reference program (SYS127) then set the **Cross Reference** value from the drop down list to `Accounting Entity`.

4   If the BOD message must include the LocationID create a new child Verb Element.

    a   In the property view set the Element Name to `LocationID`.

    b   If this location is defined in the SOA Cross Reference set the Cross Reference to Location. If the LocationID should have the same value as the Accounting EntityID, set the Cross Reference to `AccountingEntity`.

5   If you use an SQL statement to retrieve the values for the accounting entity and location, set the Database Field in the Verb Element property view to the SQL field that contains the value for the database field and the prefix for the Table (for example, `HPH.PHCOMP`). If the BOD message requires a BOD value and not the LX value, then the LX value must be cross referenced with a BOD value in the SOA Cross Reference application. If the BOD value must be added to the BOD message, then the Verb Element must have the Cross Reference set to `Accounting Entity` or `Location`, depending on which property is being set.

In this sample, three Verb Elements were added to the SetVerb Instruction shown below for the PurchaseOrder BOD. The LocationID has the same value as the `AccountingEntityID`. Retrieve the LX value for the LocationID from field **HPH.PHCOMP** and use this value to retrieve the BOD value from the Cross Reference. The Cross Reference is set to `AccountingEntity` so the BOD value that has been set up for the LX value is retrieved from the Cross Reference and is the value assigned to the property in the BOD message.
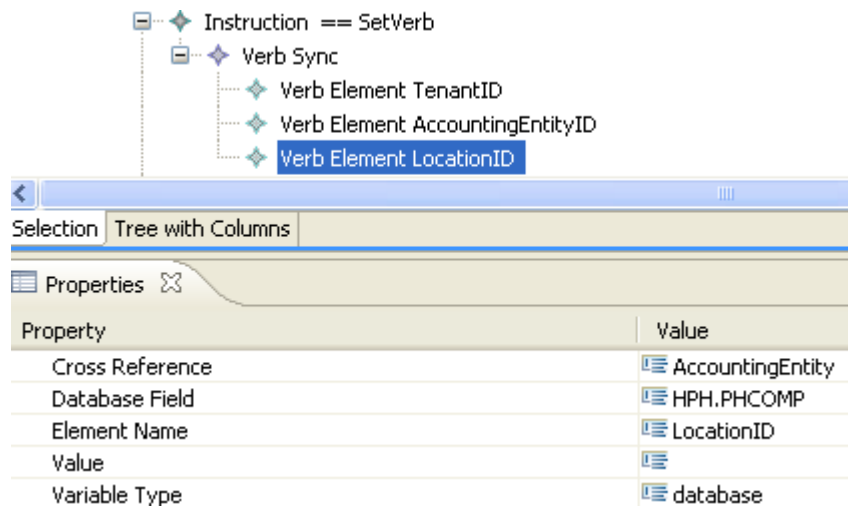

Adding the verb instruction

To invoke the Set Verb Instruction from the entry point Condition, add a new child Instruction Name as a child of the Conditional Instruction node or as a child of the If Condition node.

In the property view, set the Name for the Instruction Name to the same value set for the Instruction node. In this example, the Instruction Name is `SetVerb`. In the following screen, the Verb is added

into the BOD message after the header and all lines have been added. Always add the Verb Instruction as the last instruction: it is added after the BOD message has been created.
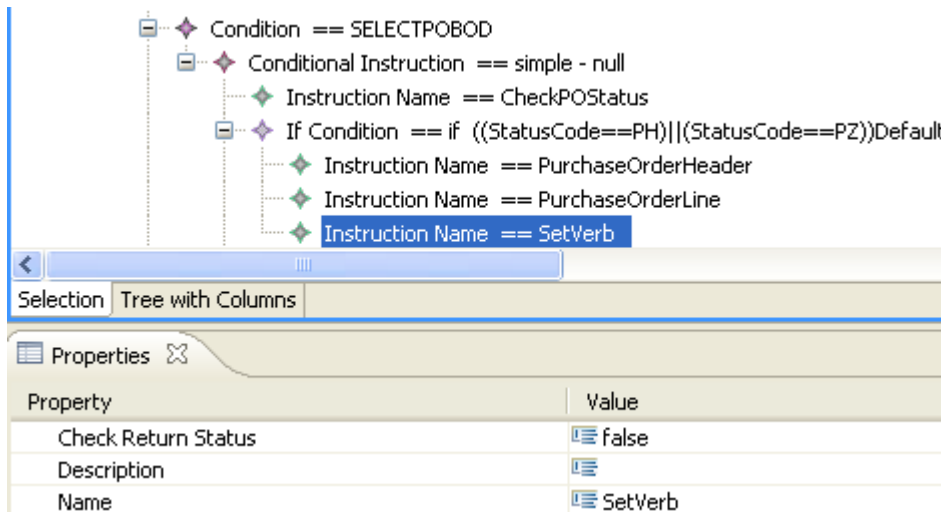
At runtime the verb that was defined earlier is written into the BOD message as shown below.

```
<DataArea>

    <Sync>

        <TenantID>INFOR</TenantID>

        <AccountingEntityID>WMS</AccountingEntityID>

        <LocationID>WMS</LocationID>
```

Defining Data Areas in the process instruction

To retrieve LX information from a data area, map positions in a data area to a Name property. For example, you could map Name FetchArea for a character length of 3 starting at position 10 to extract the data from a named data area starting at position 10 for a length of 3. In the example, shown below the requirement is to retrieve the value from start position 245 for a length of 1 from data area SSASYS.

To create a Data Area instruction that retrieves the value:

1   Select the Outbound Noun node and add new child Instruction.

2   In the property view for the Instruction, set the Name.

3   Select the Instruction and add new child Data Area Instruction.
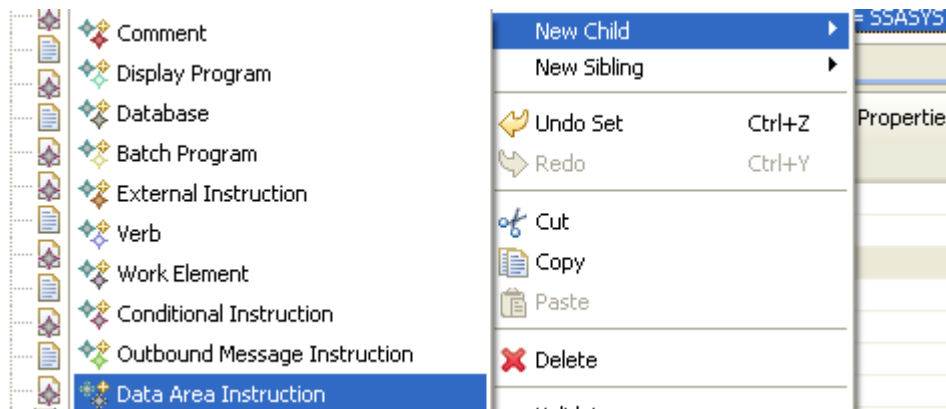
4   Use the property view to set Data Area Name. This is a required property.
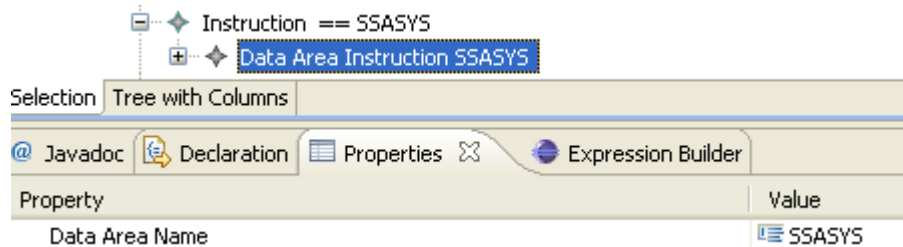
5   Select the Data Area Instruction and a new child Data Area Field for each position that will be retrieved from the data area. Open the Data Area Field property view. See Chapter 2 for a list of properties.

6   Specify a Name, Start Position and Number of Characters. The default type for a Data Area Field is `char`.

7   In the following screen, the Instruction is extracting one character starting at position 245 and setting the Name.
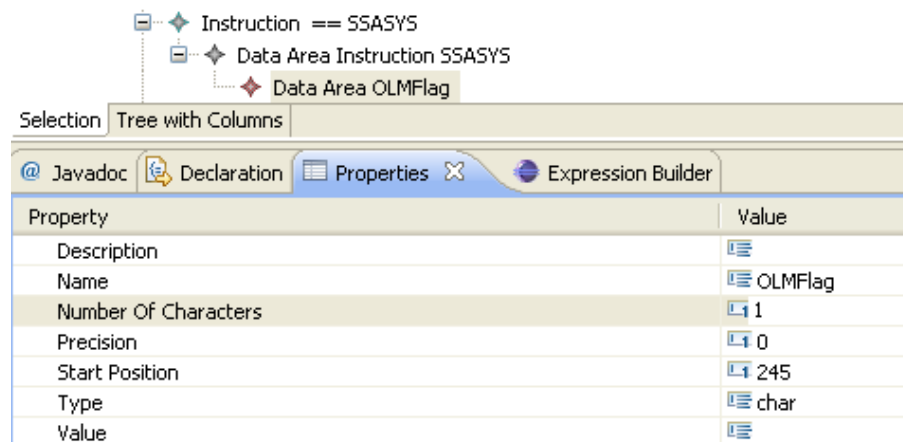
Example of invoking a data area instruction

A data area instruction must be invoked from an Instruction Name defined elsewhere in the process instruction. This example uses a Condition and Work Element.

1   Add a new Conditional Instruction in the appropriate place in the process instruction.

2   Add a new child Instruction Name and assign the Name as that given to the Instruction in Step 2 above.

3   Select the Instruction Name and add a new child Work Element.

4   Open the property view for the Work Element. The work element is used to add element OLMFlag into the Xpath element. In this example, it is added into the PurchaseOrder.Criteria.Equal element. This element is defined by the exit point process instruction. (See Sample 6.).

5   In Sample 6 above the Name OLMFlag was used to extract a value from the data area. If you assign the Name of the Data Area object to the Value of the WorkElement, the value that was retrieved is assigned to the Xpath Element OLMFlag.
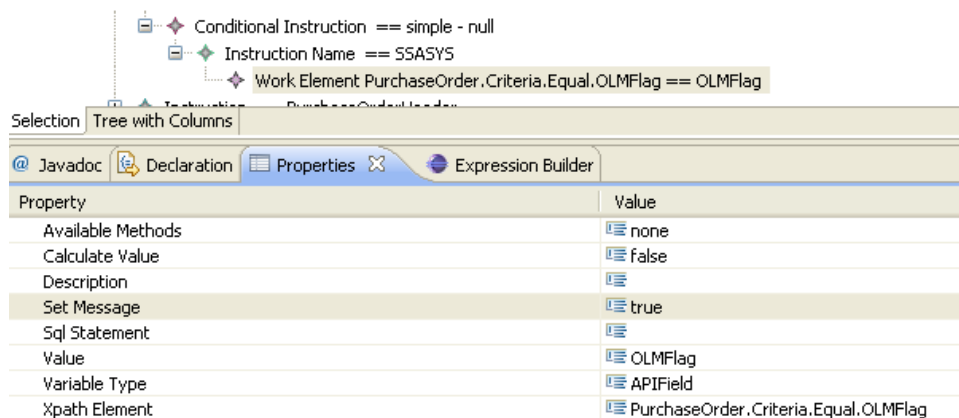
6   By setting the Value to the Name that was given in the Data Area Instruction, at runtime the value retrieved for this Name is assigned as the value given to the element (OLMFlag) we are adding into the exit point message.

Creating multiple BODs from a single transaction

You can create process instructions that produce multiple BOD messages for a single transaction. The exit point could provide all of the information needed to map data to different instructions. Each instruction could produce a BOD message. You can also use data retrieved from an SQL statement to produce multiple BODs for the transaction.

This section provides instructions to create a process instruction that produces multiple messages, for example, multiple invoices.

1   Create a Database instruction that maps fields defined in the exit point to element names that are in the BOD message.

2   Create a second Database Instruction that maps fields to elements.

3   Create a Condition that contains multiple Conditional Instructions. The Conditional Instructions will use Instruction Name objects to execute the Database instructions defined in step 1 and step 2.

4   Normally a process instruction produces a single BOD message. Use an Instruction Name object to produce multiple BOD messages. The Instruction Name object contains a property called Last Instruction. When this property is set to true the Send Message name is added to it. This instruction must be added as the last instruction in the Conditional Instruction. When the LX Extension executes a Send Message instruction the message is produced and written to the Outbox. After putting the message in the Outbox the LX Extension continues to the next instruction which may produce another BOD message.

5   The screen below shows a Conditional Instruction that contains a Send Message instruction. After executing the Send Message the LX Extension proceeds to the next Conditional Instruction to continue processing. The additional conditional instructions could invoke the second database instruction created in Step 2.

```
Conditional Instruction  == simple - null
    Comment ** These instructions responsible to create Original Invoice or Credit Memo or Debit Memo from each SIH
    If Condition  == if  (SIHCOUNTER>0)Default
        Instruction Name  == GetARSIHOrderNumber
        Instruction Name  == GetARDocumentKeys
        Instruction Name  == ARInvoiceHeader
        Instruction Name  == ARInvoiceLine
        Instruction Name  == ARSetVerbAdd
        Instruction Name  == SendMessage == true
```

A second method that produces multiple BODs from a single transaction requires using the If Condition object and an SQL Instruction.

6   To create a PI that uses the If Condition and an SQL Instruction, set the Properties in the If Condition to produce multiple BOD messages.

7   Set the Condition Type to `while`.

8   Set the expression to a while type expression, for example (ORIGDOCSIHCOUNTER>0). In this case ORIGDOCSIHCOUNTER was defined in a previously executed instruction.

9   Set the Loop Element Name to the name of the field to extract from a result set. This value should hold the number of times to execute the while statement.

10 The screen shows the If Condition which contains an Instruction Name Send Message. This instruction sends the message to the Outbox. All instructions contained in the while condition are executed until the count is 0.

11 An SQL statement named GetAROrgSIHOrderNumber is used to retrieve the orders in the invoice. A BOD message is created for each order number in the invoice. The SQL Instruction requires configuration as shown in this screen:

12 Set these properties:

a   Set Field to the field in the SQL statement that contains the order number.

b   Set Looping type to `forloop`. The data returned from the SQL instruction is passed for each iteration of the while loop.

Defining an arithmetic summation

The summation feature traverses an outbound message at runtime after the BOD message has been completed and before the message is passed to the outbox. You can, for example, use this

method to sum all the lines in a purchase order. A completed BOD can be traversed given a parent element to search and the name of the child within the search element to sum. For example, PurchaseOrderHeader.TotalAmount is the summation of all PurchaseOrderLine.TotalAmount values.

To accomplish the summation:

1   Create a Conditional Instruction and add an If Condition as a child.

2   Set these properties on the If Condition:

    a   Condition Type: set to **Arithmetic Expression**.

    b   Available Methods: set to **SUM**.

    c   Loop Element Name: specify the name of the element to search for in the BOD outbound message.

    d   Expression: specify the complete Xpath to the child element in the Loop Element. The expression must be prefixed with a colon.
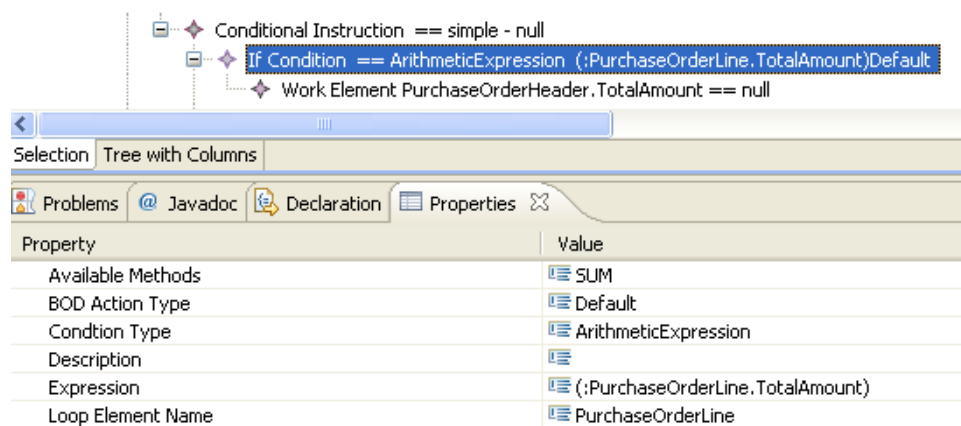
3   Add a Work Element as a child of the If Condition to set an element in the BOD with this calculated sum.

4   Set these properties on the Work Element:

    •   **Xpath Element**: set to the element in the outbound message for which the value will be set.

    •   **Set Message**: set to true so that the value of the Xpath Element is reset to the summed value.

Defining a Work Element to use rounding and truncation rules

In some cases, the values assigned to an element are defined as an arithmetic expression. The result returned from a calculation may need to be formatted to use rounding or truncation rules.

To define the rules:

1   Add a Work Element as a child to a parent node.

2   Set these properties:

- **Variable Type**: select `Arithmetic Expression`.
- **Value**: specify the expression.
- **SizeValidationType**: select the validation type, in this example, `RoundHalfUp`.
- **Length**: specify the total number of digits for the value.
- **Precision**: specify the number of digits to the right of the decimal.

The validation rule is applied after the expression is calculated.
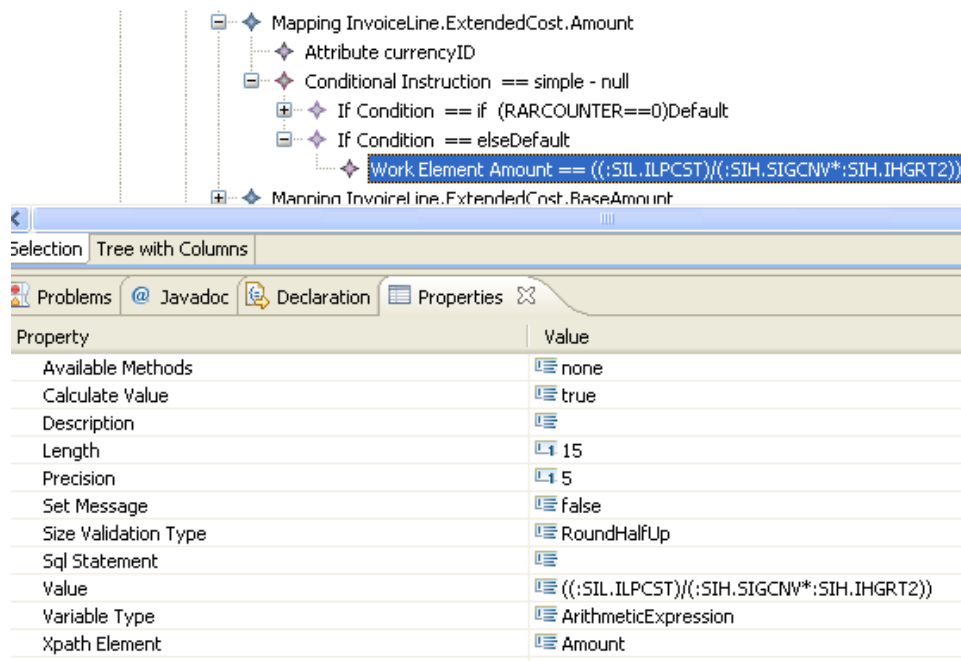
Defining a Huge BOD

Note: Huge BODs are supported only for ION Integrations.

Huge BODs can impact performance because of size. Add Huge Bod Entry nodes to a Model Object to provide the ability to produce multiple BOD messages for the same transaction. Each message that is produced is assigned a batchIdentifier as well as a batchSequence. The batchidentifier is the same for each BOD that is created. See Chapter 2 for a discussion of the property page for Huge Bod Entry nodes.

This sample shows how do add support into the outbound Model Object project that can enable batch processing. Creating a process instruction than can process a huge BOD includes these steps:

1   Create an instruction that initializes Batch information

2   To execute this initialization instruction, use an Instruction Name from the entry point instruction.

3   Create an If Conditional Instruction that uses a Conditional type of `while` that will process all lines and create new BODs as needed.

4   Write each BOD to the outbox.

5   In your Outbound Model Object tree, select the Outbound Noun node, right click, and select **Instruction**.

6 In the Properties view, set the Instruction name to `SetBatchInfo` to initialize batch information for the BOD that is being produced. This screen shows the property page for the new instruction node:
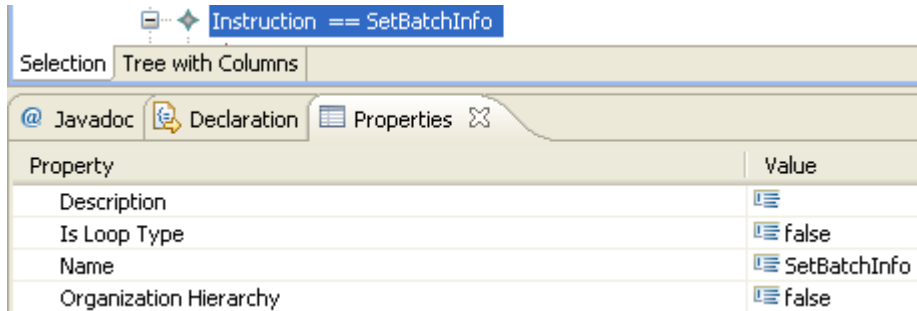

7 Appendix B indicates that the Huge Bod Entry node is a child of a Conditional Instruction node. Add the nodes.

8 Select the Instruction node, right click, and select **New Child > Conditional Instruction**.

a Select the Conditional Instruction node, right click, and select **New Child > Huge Bod Entry**. This screen shows the property page for the Huge Bod Entry node: Set the properties as shown below.
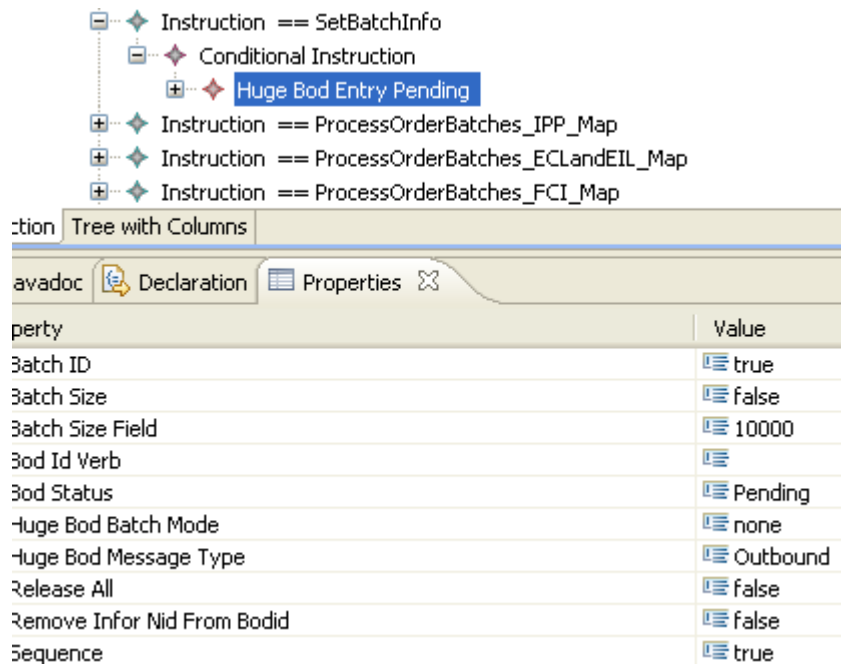
9 Set these properties:

- **Batch ID**: select `true` to enable batch processing.

- **Batch Size Field**: set this property to the default value `10000`. You can override this value in the LX Extension topology file. This property is the maximum number of child elements than can write into a BOD Message. If this number is exceeded, a new BOD is created to include the additional line information.

- **Bod Status**: select `Pending` to temporarily store each BOD message in the LX Extension BATCH_ENTRY file.

- **Huge Bod Message Type**: select `outbound`.

- **Sequence**: select `true` to indicate that each BOD message has a sequence. For example, the first message is batchSequence 1.

10 Add the child elements that are used in the batch size field count. You must add a new Huge Bod Child Entry for each child to include in the number of elements to add.

11 Select the Huge Bod Entry node, right click, and select **Huge Bod Child Element**.

12 In the Properties view, define the Name of a child element to include in the count. In this example, we are counting the number of ShipmentItem elements that are added into each batched BOD message.

13 Set the Huge Bod Child Element Type to ChildElement. The screen shows the property page for the node:
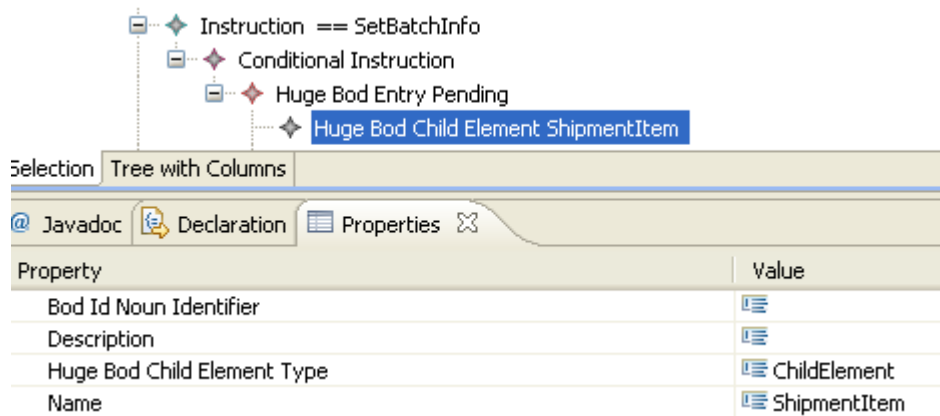
14 The SetBatchInfo instruction is used to set up batch processing. This instruction should be invoked before any BOD message is built. Add an Instruction Name node into the entry point of the process instruction.

15 From the entry point condition node invoke the SetBatchInfo Instruction. Add the Instruction Name node to execute the SetBatchInfo Instruction.
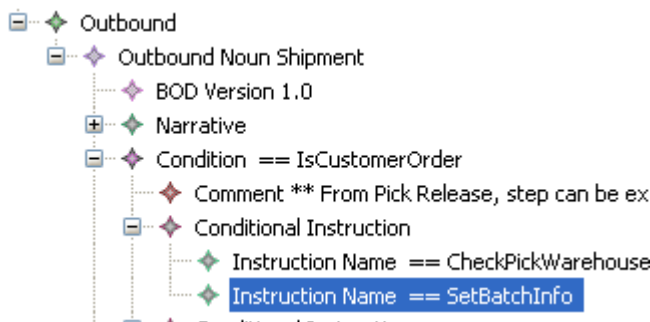
16 Add a new Instruction to process each ShipmentItem and make sure that each BOD that is produced does not exceed the BatchSize. The default BatchSize was defined in the SetBatchInfo instruction to `10000`, however this can be overwritten in the topology file. The value

for the BatchSize is stored in memory using variable BatchInfo. Use this variable to process Huge Bod instructions.

- Add a Conditional Instruction node.
- Add an If Condition to the Conditional Instruction node.
- In the Properties view, specify these properties to process all instructions contained in the If Condition:
- Condition Type: select **while**.
- Expression: specify **(BatchSize)**. Enclose the expression in parentheses.

17 Add child nodes to the If Condition node. The child nodes process ShipmentItems and provide instructions on what to do with the BOD message after it is built. This screen shows that two Conditional Instructions have been added to the If Condition.

18 The first Conditional Instruction has an Instruction Name child node. At runtime, this instruction invokes the Instruction ReadIPP_Map which is an Instruction that builds a ShipmentItem. This screen shows the ReadIPP_Map instruction:

19 After the BOD Message is built, the second Conditional Instruction is executed. This instruction contains a Huge Bod Entry node. The property page for this node is shown below. Notice in this node the BatchSizeFIeld is not set and the Bod Status has been changed from Pending to Usable. This indicates that the BOD message that was created and temporarily stored in the BATCH_ENTRY can be removed from the BATCH_ENTRY and written to the LX Extension outbox.
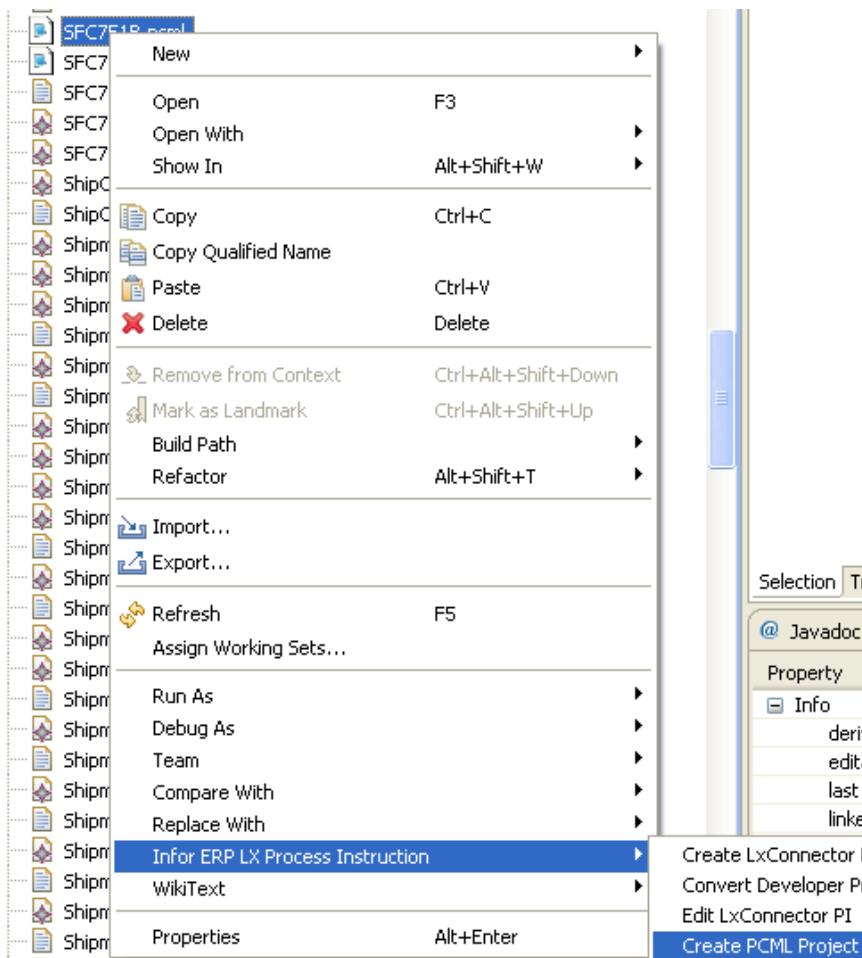

Sample PCML Model Object tree view

Appendix A describes how to create a PCML Model Object. To create the project you must have a PCML file that is produced from the RPG program. The generated PCMLfile should be Imported into a generic project folder. In this example, we have generated SFC751B and imported it to a project folder.

1 To create a pcml project, select the SFC751B.pcml file, right click and select **Create PCML Project**.

2 This produces a SFC751B.developer project. Double click the project to open in the tree view. The Model Object tree view is shown below. A Pcml Data node is added for each Parameter that is passed to the API.
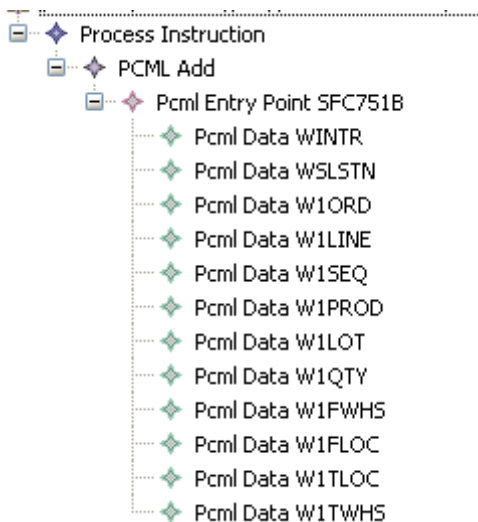
3   To modify the Pcml Data, assign an Xpath to each node. The Xpath is a name that can be used in the BatchProgram instruction that is executed from the generated inbound or outbound process instruction. In the screen below W1PROD is the field in the RPG data structure and Item is the name used in when defining the Batch Program.


4   After you set the Xpath and PCML Parm Types for each Pcml Data node, generate process instructions. Selecting the SFC751B.developer file, right click, and select **Generate Process Instruction**.
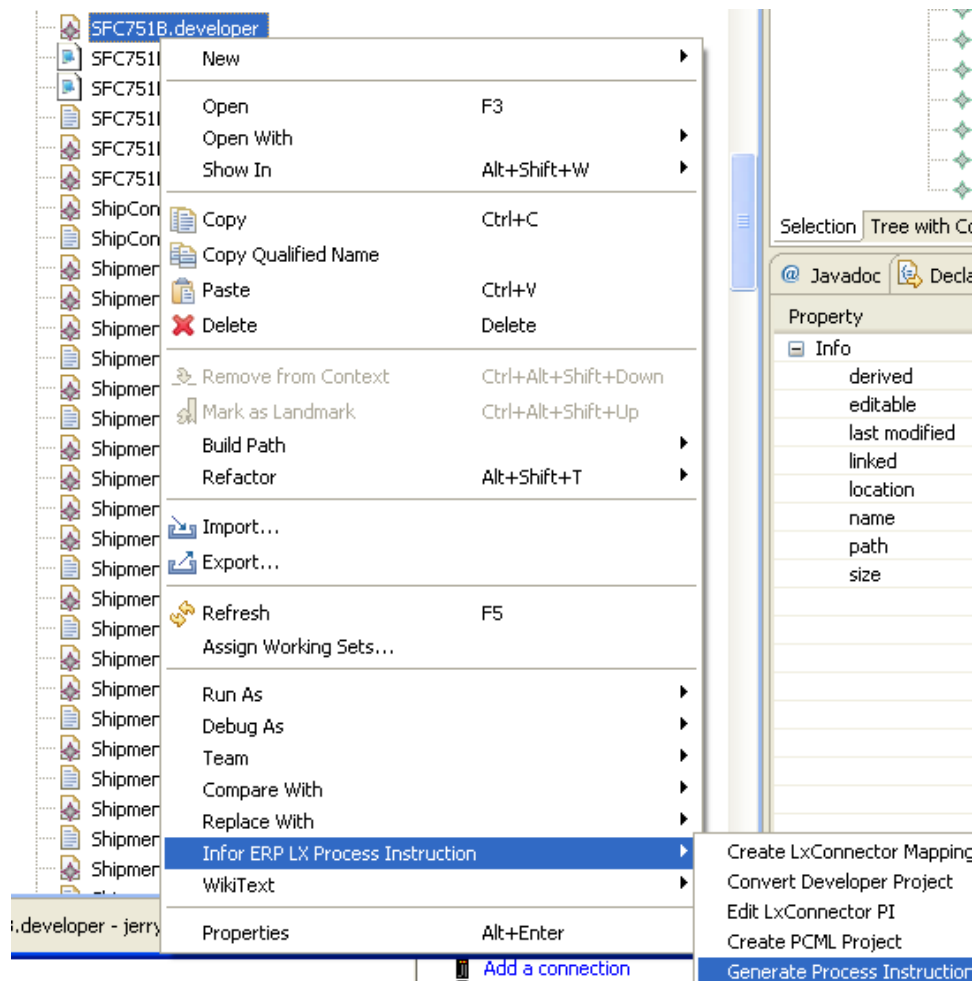
5   This generates two files into the project folder, `SFC751BAdd.pcml` and `SFC751BAdd.xml`. Both of these files must be added into the `LXESBPI.jar` file if the process instructions are used by the LX Extension. If used by the LX Connector they are added into the `LXCPI.jar`.
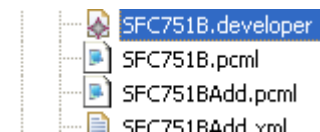
6   Use the Add Jar File view introduced in Chapter 1 to add the files to the jar files. Be sure to select the serialize option when adding the `SFC751BAdd.pcml` file. Do not serialize the `SFC751BAdd.xml` file. The `SFC751BAdd.xml` file is the process instruction when executing the `SFC751BAdd.pcml.ser` at runtime.

The screen below shows the `SFC751BAdd.xml` generated process instruction. Each element is the xpath defined in the Pcml Data node. Each value is the field defined in the API data structure.

```
<?xml version="1.0" encoding="UTF-8" ?>
- <SFC751B>
  - <Add Pcml="SFC751BAdd" ServicePgm="F">
      <RunTime Input="T" length="1">WINTR</RunTime>
      <ListNumber Input="T" Output="T" length="8" precision="0">WSLSTN</ListNumber>
      <OrderNumber Input="T" Output="T" length="8" precision="0">W1ORD</OrderNumber>
      <LineNumber Input="T" Output="T" length="4" precision="0">W1LINE</LineNumber>
      <ELASequence Input="T" Output="T" length="3" precision="0">W1SEQ</ELASequence>
      <Item Input="T" Output="T" length="35">W1PROD</Item>
      <Lot Input="T" Output="T" length="25">W1LOT</Lot>
      <Quantity Input="T" Output="T" length="11" precision="3">W1QTY</Quantity>
      <FromWarehouse Input="T" Output="T" length="3">W1FWHS</FromWarehouse>
      <FromLocation Input="T" Output="T" length="10">W1FLOC</FromLocation>
      <ToLocation Input="T" Output="T" length="10">W1TLOC</ToLocation>
      <ToWarehouse Input="T" Output="T" length="3">W1TWHS</ToWarehouse>
    </Add>
</SFC751B>
```

# Sample API defined in the process instruction

See Appendix A for instructions to create and generate an API process.

This is a continuation of the Sample Pcml Model Object tree view. In this example, we will execute the SFC751B program by using the Batch Program node in our Inbound Model Object tree view.

1   In the Inbound Model Object tree view select the Noun, right click and select **Add Child Instruction**.

2   Set the Name to `TransferAllocations`. The property page for the Instruction node is shown below.

| Property	Value
Description	
Is Loop Type	false
Name	TransferAllocations
Organization Hierarchy	false
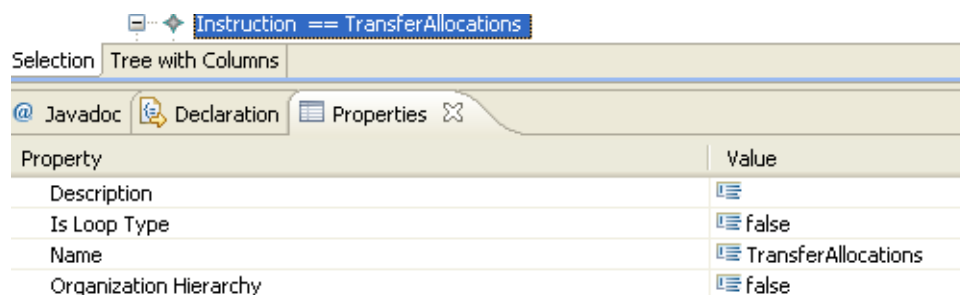
3   We want to create this instruction so that when it is invoked using an Instruction Name node from another instruction it will execute a Batch Program. Select the Instruction, right click and select **New Child Batch Program**. The property page for the node is shown below.

4   Set the Name to be the name of the API that is executed at runtime.

5   Set the Action to that defined in the PCML Model Object, Add for this sample.
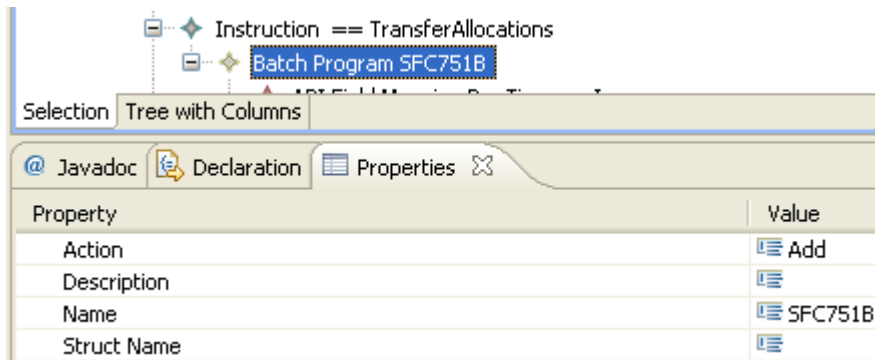
6   Select the Batch Program node and add an API Field Mapping for each parameter that is being passed to the SFC751B program. The API Field Mapping is used to map a value from a BOD message to an API Field. The API Field is the value set in the Xpath when creating the PCML Model Object project.

The screen below shows the mapping for all parameters. The property page for the first mapping shows that we are setting name RunTime to a constant Variable Type having a value of **1**. The variable property holds the value. If you look in the SFC751B.xml process instruction, you see that when the runtime executes the SFC751B program using PCML it assigns **1** to field **WINTR**. The screen also shows that the Item (W1PROD) is set to the value in the ShipmentItem.ItemID.ID element in the current ShipmentItem that is being processed.
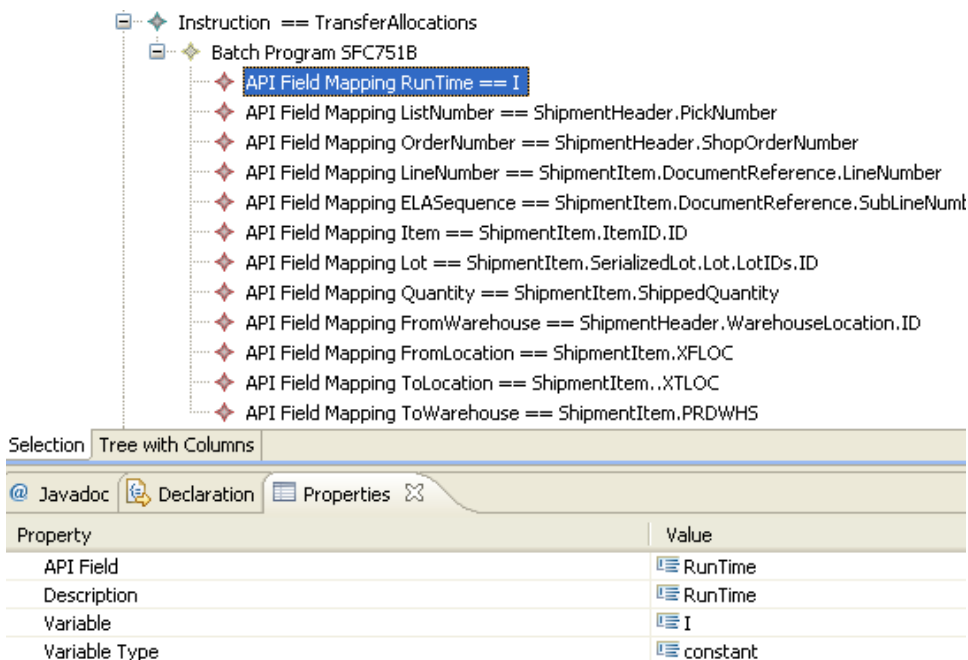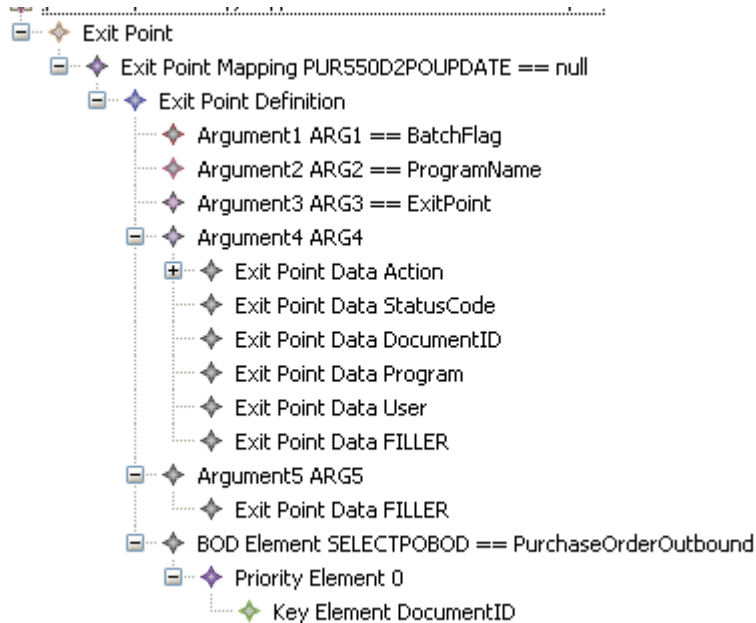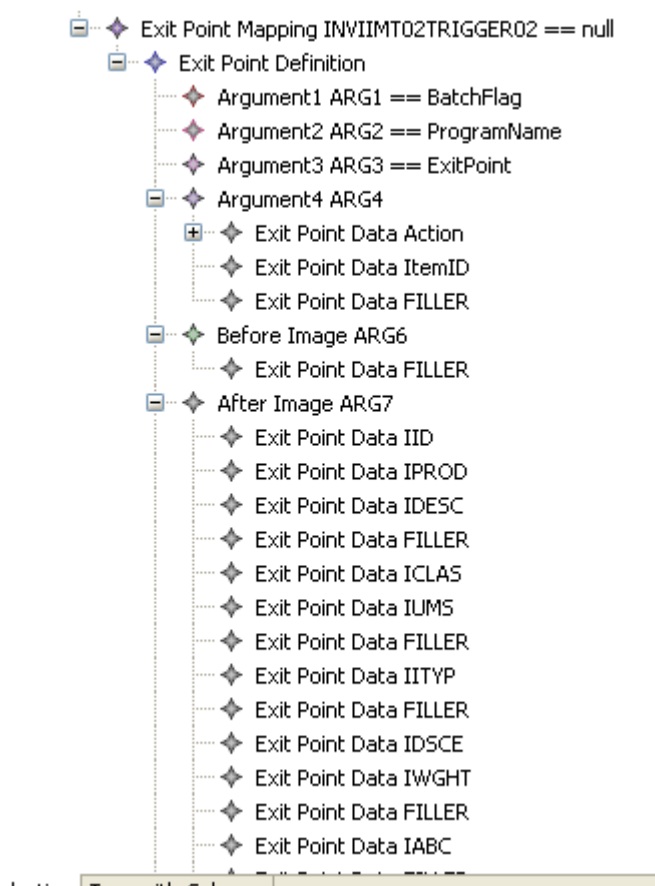
Sample exit point Model Object tree view

Create exit point Model Object tree views to map exit point data defined in an RPG data structure to elements that can be used when building an inbound or outbound process instruction. When creating an Exit Point Model Object the nodes that are typically used are listed below. When mapping an exit point 5 arguments are required. Argument 4 and Argument 5 are 256 byte data structures. The entire data structure must be mapped in the order defined in the RPG data structure.

- Exit Point Mapping
- Exit Point Definition
- Argument 1
- Argument 2
- Argument 3
- Argument 4
- Argument 5
- BOD Element
- Exit Point Data
- Priority
- Key Element

This screen shows a sample model object of an exit point:

Triggers can also be mapped using an Exit Point. The difference between mapping an Exit Point and a trigger is that the Exit Point Model Object tree requires 2 more nodes. The Before Image and After Image map to a 9999 byte data structure. All fields in the data structures must be added and must be in the order defined by the file. The screen below shows an Exit Point Model Object tree view for a trigger.

```
☐ ◆ Exit Point Mapping INVIIMT02TRIGGER02 == null
    ☐ ◆ Exit Point Definition
        ◆ Argument1 ARG1 == BatchFlag
        ◆ Argument2 ARG2 == ProgramName
        ◆ Argument3 ARG3 == ExitPoint
    ☐ ◆ Argument4 ARG4
        ⊞ ◆ Exit Point Data Action
            ◆ Exit Point Data ItemID
            ◆ Exit Point Data FILLER
    ☐ ◆ Before Image ARG6
            ◆ Exit Point Data FILLER
    ☐ ◆ After Image ARG7
            ◆ Exit Point Data IID
            ◆ Exit Point Data IPROD
            ◆ Exit Point Data IDESC
            ◆ Exit Point Data FILLER
            ◆ Exit Point Data ICLAS
            ◆ Exit Point Data IUMS
            ◆ Exit Point Data FILLER
            ◆ Exit Point Data IITYP
            ◆ Exit Point Data FILLER
            ◆ Exit Point Data IDSCE
            ◆ Exit Point Data IWGHT
            ◆ Exit Point Data FILLER
            ◆ Exit Point Data IABC
```

Sample use of Acknowledge

An Acknowledge node is used by LX Connector inbound integration projects. Adding an Acknowledge into the project causes key information to be returned to a client application. In this sample a project is created using the LX ION PI Builder. The screen below shows an Inbound tree view for an Item. The Acknowledge is added as a child of the first Action.

When a Create BOD request is received by the Lx Connector runtime the Item process instruction is loaded and executes the Create Instruction shown above. The Acknowledge in the instruction causes the ItemCode to be added in the message returned after execution to a client application. The screen shown below shows a message returned to a client application after the Item has executed. Notice the message contains the `<ItemCode>`.

```
<Envelope>
 <Item identifier="2011-03-3008:16:26_Item_Create13014
  <itemCode>SMG-STANDARD80</itemCode>
  <MessageDetails>
   <Error>UMI9101
     <ErrMsgId>UMI9101</ErrMsgId>
     <ErrMsg>Cannot Add, entry already exists. Cause . . .
   </Error>
  </MessageDetails>
 </Item>
</Envelope>
```

Using Available Methods

Available Methods is a property used by these nodes.

- If Condition
- Loop Element
- Outbound Message Instruction.
- Work Element

Chapter 2, section "Available methods options" describes the methods that are available. This section shows some examples on how to use the Available Methods options.

Addprocessreplace

Use this method in your Model Object project to process an inbound BOD message that Replaces rows in an LX subfile and the BOD message contains information that adds a new row, updates a row, or deletes a row from the subfile.

This example shows how to use the Addprocessreplace method.

The example assumes the Model Object project will process inbound PurchaseOrder messages that replace lines in an LX legacy application. The Model Object project must be able to process a replace method if the BOD message contains several PurchaseOrderLines. The Model Object project must be able to add, update, and delete lines using an LX legacy application.

Each line of the message is processed differently by the LX Extension or LX Connector runtime. To add support for this into the Model Object

- To update Elements in a Bod message Work Elements are used.
- Since the project must support adding, changing or deleting lines in a subfile the Available Method property must be set.
- For those lines that insert into the subfile the Available Method is Addprocessreplace
- For those lines that are changed in the subfile set the Available Method to Changeprocessreplace
- For those lines that are deleted in the subfile set the Available Method to Deleteprocessreplace.
- Add If Condition nodes to process the correct Work Element
- The property page shown below shows the Work Element that is executed to update a PurchaseOrderLine to include information that will Add a new row into the subfile.

ExitProcessInstruction

Use this Method if the Model Object needs to inspect the contents of an inbound Bod message and based on context exit the process instruction.

To exit processing an inbound BOD message include the following instructions into the Model Object.

• Use a Work Element to check the value of an Element in the BOD message.

• Add an If Condition that if true invokes an Instruction that exits the process instruction.

• Use an Outbound Message Instruction to set the Available Methods to ExitProcessInstruction.

• Add a Confirm Error Message to set an LX message id.

The screen shown below shows the Instruction that is called from an Instruction Name node. The Outbound Message Instruction is used to retrieve an error message using the message id in the Confirm Error Message. This creates an error message that is returned as a ConfirmBOD and posted to the outbox.

InsertNonExistingXpathElement

Some inbound projects need to loop through elements in the Bod message and add new child elements into a child. This is usually needed for very complex processing such as partial shipments. For example, when processing a ShipmentItem in a Shipment Bod a Work Element can be used to create a new child element into the current ShipmentItem.

The property page for the Work Element is shown below.

1   To create a new Element set the Available Methods to InsertNonExistingXpathElement. This adds the ConfirmDetail into the ShipmentItem.

2   Set the property Xpath Element to the element that is added into the ConfirmDetail element that was added. Since the Variable Type is set to Inbound the Value for the new element is extracted from the value of Shipment.Item.TemLineLeftNumber.

IsEmpty

Use the IsEmpty Available Method in an inbound Model View if your project needs to inspect an element and decide if the Element is empty.

Use an If Condition node to check a node. In the If Condition node select the Available Method to IsEmpty and the Expression property to the Element to check in the Bod Message.

IsLower

If you are creating a Model View that needs to make a decision based on the case of an Elements value when processing an inbound Bod Message add an If Condition node, select the Available Methods to IsLower if you are checking if the value is all lower case and then set the Expression to the node to check. There is also an isUpper that is set the same way.


Sample use of SQL Definition

Note: This feature is not supported by LX Extension 2.0 and LX Connector 1.0 and earlier releases. This feature will be available in future LX Extension and LX Connector releases.

Use this instruction to assign a value from an SQL Result set to an element. In this example, we will use an SQL Definition to build an outbound Quote BOD for an ION integration. We will use an SQL Definition instruction to add salesperson information to an element. The element that contains the SalesPerson data is QuoteHeader.OrderCommission. This example assumes that a project was created for the QuoteOutbound and that you are in the process of adding Mapping elements into the Mapping Detail section of the Database Instruction.

To add and use an SQL Definition instruction:

1   In Mapping Detail, create a Mapping Element for QuoteHeader.OrderCommision.

2   In the property page set these properties:

- Element: QuoteHeader.OrderCommission

- Repeating Element: **True**. Multiple OrderCommission elements may exist in the BOD message.

- Do not change any of the other properties.
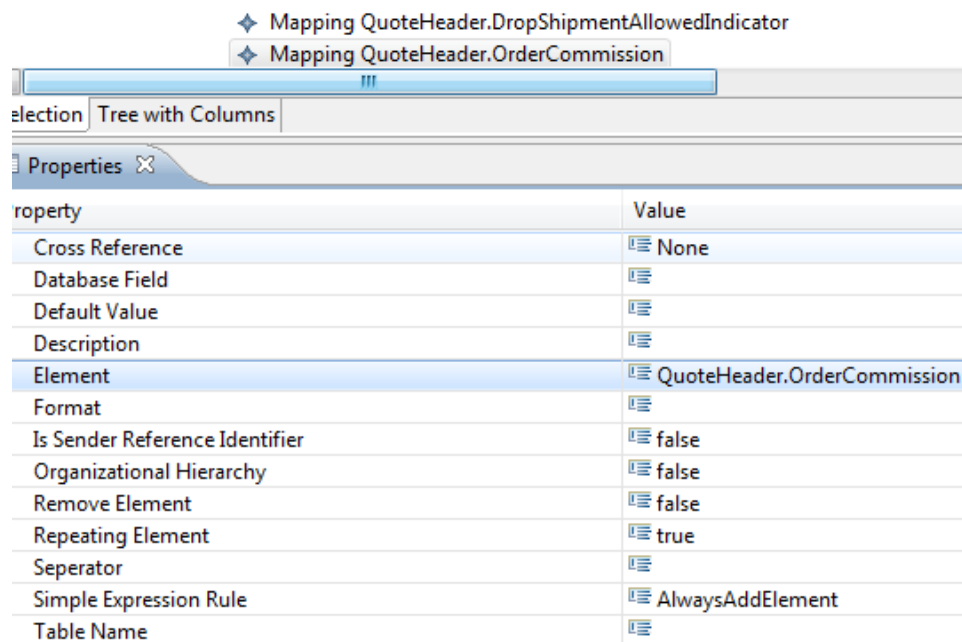
3   Add a Conditional instruction as a child of the QuoteHeader.OrderCommission.

4   Add an Sql Definition as a child of the Conditional instruction.

5   On the Property page for the SQL Definition accept the default values.
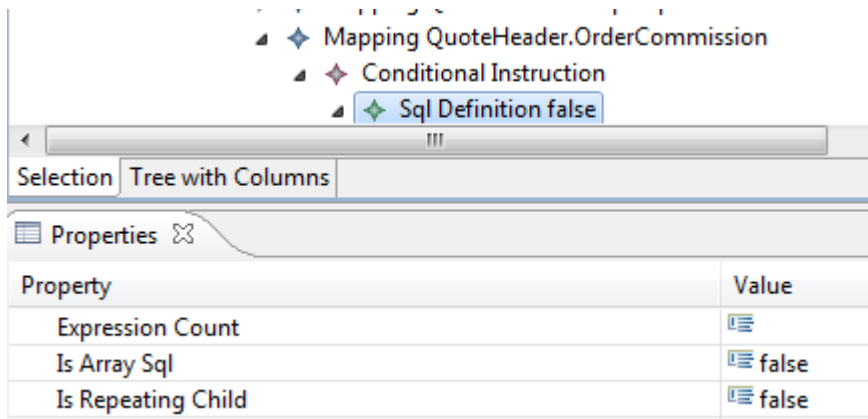
6　Select the SQL Definition and add new child Statement.

7　Use the SQL Expression Builder view to build an SQL statement that retrieves sales person data. The SQL statement is shown in the property page below:
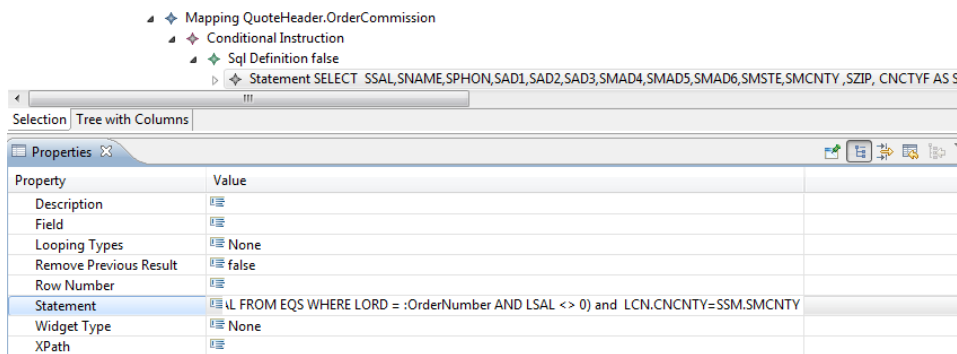
The complete SQL statement is a union and is shown in the Expression Builder view below.
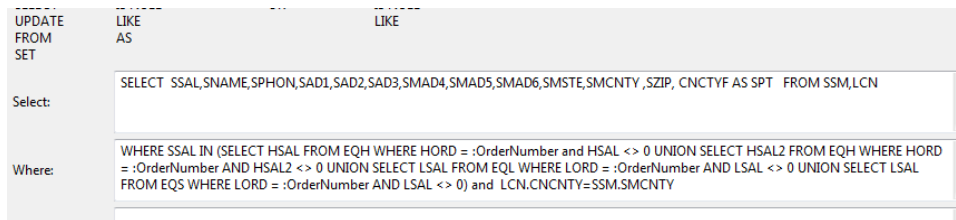
8　The SQL statement retrieves several fields and each field is used to add an element into the QuoteHeader.OrderCommission.

9　To avoid issues at runtime, add the SQL Success and SQL Failure instructions to check if rows were returned. To add these instructions, select the Statement, add new child Conditional Instruction, and then add new child SQL Failure.

10　Select the SQL Failure instruction and add new child Sql Result Set Variable.

11　For this example, set these properties with these values:

- Set Default: **failed**
- Value: **SSM.SQLERROR1**

12 The SSM file is the first file in the FROM of the SQL statement. If no rows are retrieved this
    element is saved in temporary memory: `<SSM><SQLERROR1>failed</SQLERROR1></SSM>`.

    The property page for the SQL Failure instruction is shown below.

13 Add a child instruction to see it the SQL Statement returned rows. Select the Conditional
    Instruction, add new child SQL Success, and then add new child Sql Result Set Variable.

14 On the property page, set the Set Default property to success. The other properties have the
    same values as those defined for the SQL Failure instruction.

15 Select the SQL Statement, add a child Conditional Instruction, and then add a child If Condition.

16 Use the variable defined in the SQL Success instruction to define the next instructions that are
    executed. On the If Condition property page, set these properties to determine if rows were
    retrieved:

    •   Available Methods: `none`

    •   Condition Type: `if`

    •   Expresson: `(SSM.SQLERROR1==success)`

17 If rows are returned, add Mapping instructions for relevant fields in the SQL statement. These
    instructions are child elements that are added into the Order Commission. In this example, five
    child elements are added as children in the OrderCommission:

In this example, the SalesPersonID is assigned the value from the SSAL field retrieved in the Statement.

At runtime the Sql Definition uses information from the SQL statement to add element OrderCommission and its child elements to the BOD as shown in this sample:

```
- <OrderCommission>
  - <SalesPerson>
      <ID>1-600670</ID>
      <Name languageID="en-US">Rick's test SALESPERSON name90</Name>
    - <Communication sequence="1">
        <ChannelCode listID="Communication Channels">Phone</ChannelCode>
        <UseCode listID="Communcations Use Codes">Office</UseCode>
        <AreaDialing>PHONE**********</AreaDialing>
      </Communication>
    - <Communication sequence="2">
        <ChannelCode listID="Communication Channels">Mail</ChannelCode>
        <UseCode listID="Communcations Use Codes">Office</UseCode>
      - <Address>
          <AddressLine languageID="en-US" sequence="1">SSM add 1
            xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx50</AddressLine>
          <AddressLine languageID="en-US" sequence="2">SSM add 2
            xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx50</AddressLine>
          <AddressLine languageID="en-US" sequence="3">SSM add 3
            xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx50</AddressLine>
          <AddressLine languageID="en-US" sequence="4">SSM add 4
            xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx50</AddressLine>
          <AddressLine languageID="en-US" sequence="5">SSM add 5
            xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx50</AddressLine>
          <AddressLine languageID="en-US" sequence="6">SSM add 6
            xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx50</AddressLine>
          <CityName>SSM add 3 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx50</City
          <CountrySubDivisionCode>SMSTE</CountrySubDivisionCode>
          <PostalCode>66601</PostalCode>
        </Address>
```

Sample of SQL Definition with Is Array SQL

Note: This feature is not supported by LX Extension 2.0 and LX Connector 1.0 and earlier releases. This feature will be available in future LX Extension and LX Connector releases.

This example shows how to create an SQL Statement that retrieves a set of fields. Each field retrieved must be a child element in the BOD message. In this example, it is assumed that you are mapping elements to a PurchaseOrder BOD that uses ION. In this example, the BOD must display all of the Tax Codes for a Purchase Order item that do not have all blanks as the returned value. The element that provides the tax information is PurchaseOrderLine.DistributedTax.TaxJurisdicationCodes.Code. The example will create a new Code for each field listed in the SQL statement that is non-blank.

This is the SQL statement that is executed:

    SELECT RTRC01, RTRC02, RTRC03, RTRC04, RTRC05, RTRC06, RTRC07, RTRC08, RTRC09, RTRC10 FROM
    ZRT WHERE RTCVCD=':HPO.POVTXC' AND RTICDE = ':HPO.POITXC' AND RTID='RT' AND
    RTWHSE=':HPO.PWHSE'

To use the SQL Definition with property Is Array SQL set to `true`:

1   On the Mapping instruction property page, set these properties:

   •   Element: the parent is TaxJurisdictionCodes; add a code element to retrieve each tax code. The complete element nameis
       `PurchaseOrderLine.DistributedTax.TaxJurisdicationCodes.Code.`

- Repeating Element: set to **`true`** to retrieve each tax code in the SQL statement and add a Code element for each tax code.

- Simple Expression Rule: set to **`AddElementIfTrue`**. The Code is only added into the BOD message if it passes a condition.
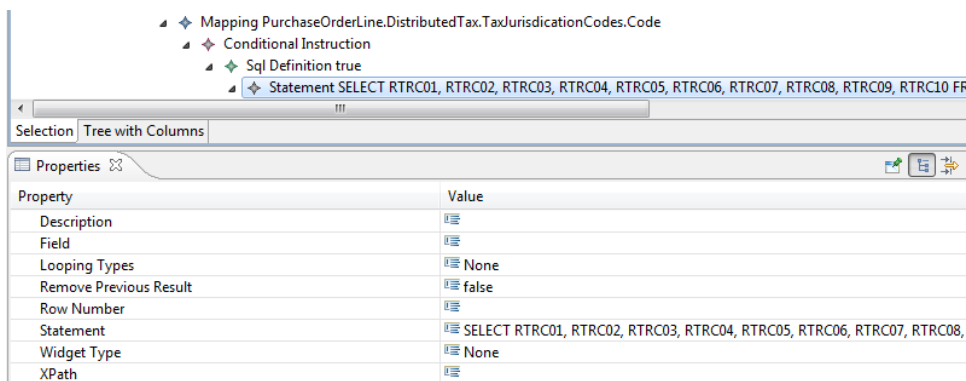

2   Add a Conditional instruction and the Sql Definition instruction.

3   On the Sql Definition property page, set **Is Array** to **`true`**.

4   Add a Statement instruction and use the Expression Builder view to define the SQL statement.

5   Add SQL Success and SQL Failure instructions.

6   Add the Sql Result Set Variable instructions to both the SQL Failure and SQL Success instructions.

```
⊿ ✦ Sql Definition true
    ⊿ ✦ Statement SELECT RTRC01, RTRC02, RTRC03, RTRC04, RTRC05, RTRC
        ⊿ ✦ Conditional Instruction
            ⊿   SQL Failure
                  ✦ Sql Result Set Variable
            ⊿ ✦ SQL Success
                  ✦ Sql Result Set Variable
```

election | Tree with Columns

Properties ⊠

Property	Value
Cross Reference	None
Description	
Element Name	
Name	
Name Variable Type	none
Parent To Search For	
Set Default	Success
Sql Def Expression Rule	AlwaysAddElement
Substring	
Value	ZRT.SQLErrorCode
Variable Type	database

7   On the Sql Result Set Variable property page, set the value to `ZRT.SQLErrorCode` to return rows when the SQL statement is executed.

8   Add a Mapping instruction for the Code and map it to each field in the Statement. The screen below shows that each field is examined and if the value of the field is not BLANK a new repeating element is added into the BOD Message.

At runtime the TaxJurisdictionCodes have one or more codes as shown in this sample BOD:

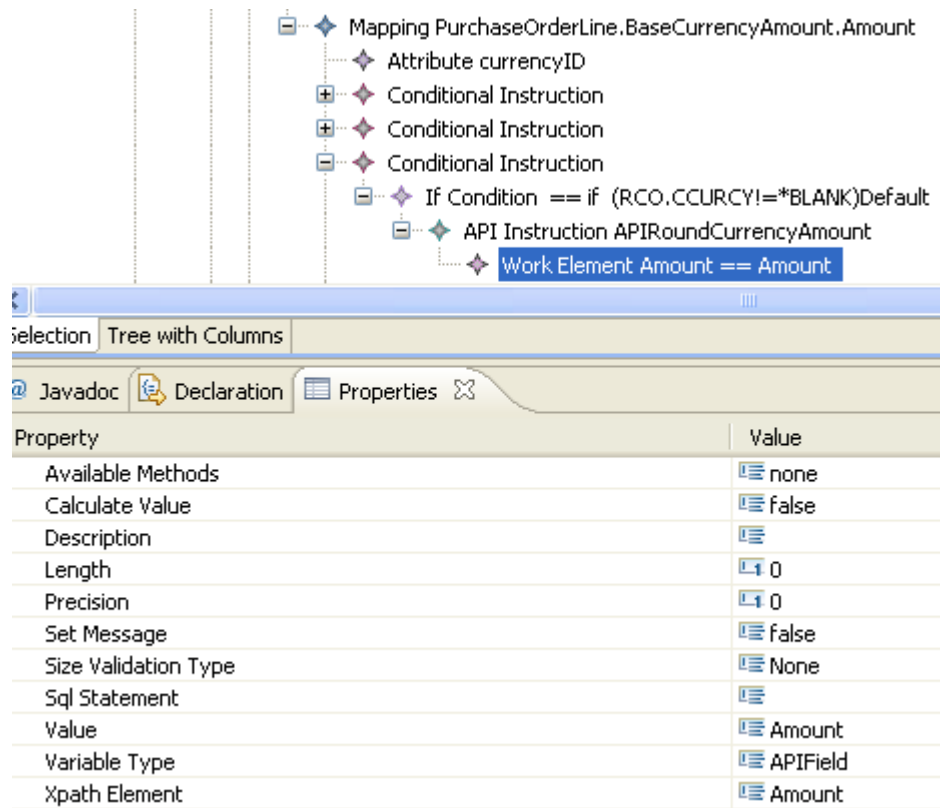# Samples using Variable Type options

Chapter 2 section "Variable Type options" contains the list of all options available to the property called Variable Type. The following nodes contain this property. This section includes a few samples on how to use some of these.

- API Field Mapping
- Concatenation Field
- Field
- Reset Element
- Simple Expression
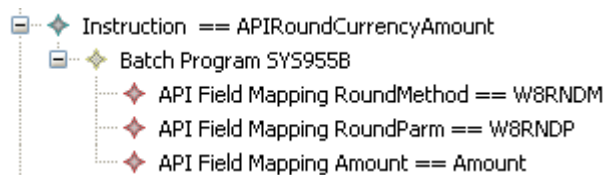- Variable
- Verb Element
- Work Element

APIField

This sample shows use of the variable type APIField in an outbound project. In this example, we need to set a value to an element using an API.

The screen below shows a PurchaseOrder outbound Model Object mapping of element PurchaseOrderLine.BaseCurrencyAmount. The screen also shows there are several instructions that are executed to set the value for this element. The instruction of interest In this example, is the API Instruction Name APIRoundCurrencyAmount. The API Instruction has a child Work Element that has Variable Type APIField. The Xpath property in the property page indicates the parameter in the API that is invoked that is set to the Value of an element called Amount.

When the purchase order process instruction executes the API Instruction it loads the Batch Program instruction shown below. Before execution of the API, parameters must be set. The Work Element Value property is the element whose value will be assigned to API Field Mapping Amount.

For example, at runtime before execution of the SYS955B program the element value in memory is assigned to parameter Amount and then the API is executed.

```
<Amount currencyID="US$">2.000000000</Amount>
```

Inbound

This sample shows use of the inbound variable type when mapping an API instruction. The screen below is an inbound Model Object view. The property page for an API Field mapping is also shown in the screen. At runtime the generated Shipment process instruction is executed when a Shipment BOD request is loaded. When the TransferAllocations Batch Program is executed the value for API Field ELASequence, a parameter passed to the API, is set by retrieving the Variable from the inbound message. In this sample it extracts the value from Variable ShipmentItem.DocumentReference.SubLineNumber

```
□···◆ Instruction  == TransferAllocations
   □···◆ Batch Program SFC751B
         ···◆ API Field Mapping RunTime == I
         ···◆ API Field Mapping ListNumber == ShipmentHeader.PickNumber
         ···◆ API Field Mapping OrderNumber == ShipmentHeader.ShopOrderNumber
         ···◆ API Field Mapping LineNumber == ShipmentItem.DocumentReference.LineNumber
         ···◆ API Field Mapping ELASequence == ShipmentItem.DocumentReference.SubLineNumber
         ···◆ API Field Mapping Item == ShipmentItem.ItemID.ID
         ···◆ API Field Mapping Lot == ShipmentItem.SerializedLot.Lot.LotIDs.ID
         ···◆ API Field Mapping Quantity == ShipmentItem.ShippedQuantity
         ···◆ API Field Mapping FromWarehouse == ShipmentHeader.WarehouseLocation.ID
         ···◆ API Field Mapping FromLocation == ShipmentItem.XFLOC
         ···◆ API Field Mapping ToLocation == ShipmentItem..XTLOC
         ···◆ API Field Mapping ToWarehouse == ShipmentItem.PRDWHS
Selection | Tree with Columns
```

```
@ Javadoc | 🖹 Declaration | 🔲 Properties ✕

Property                          Value
  API Field                         ⬚ ELASequence
  Description                       ⬚ ELASequence
  Variable                          ⬚ ShipmentItem.DocumentReference.SubLineNumber
  Variable Type                     ⬚ inbound
```

CurrentElement

In this sample a Mapping node is added into the Mapping details. The sample shows how to used Variable Type CurrentElement in a Work Element

- Set the Element to **ExtendedAmount**
- The Value for this Element is executed using API's.

```
□···◆ Mapping RequisitionLine.ExtendedAmount
    ···◆ Attribute currencyID
  □···◆ Conditional Instruction
     □···◆ API Instruction APIGetExtendedAmount
          ···◆ Work Element Tax == Tax
          ···◆ Work Element ExtendedAmount == ExtendedAmount
```

The value for the Extended Amount is set after the APIGetExtendedAmount completes. However, there are more instructions that have been added to this Element. The screen below shows an Arithmetic Condition is used that contains a child Work Element node. The property page shows:

- Variable Type is set to CurrentElement which means we are going to assign the current value assigned to the ExtendedAmount

- The Xpath Element is the path to the Value.

- This instruction sums the value currently in the RequisitionHeader.ExtendedAmount with the current value of the ExtendedAmount for this line.

Appendix A API process instructions

This appendix describes how to create an API process instruction. An API is an interface to an LX application, typically an RPG program. The LX Extension and LX Connector use PCML so you must generate a PCML file when you compile the RPG program. Use this PCML file to create an API Process instruction.

Define the mapping

1   On the System i, compile the RPG program and generate the PCML file for the program.

2   Copy the generated PCML file to the project folder directory in the LX ION PI Builder.

3   Right click on the PCML file to display the context menu.

4   Select **Create PCML Project** to create a .developer project with the same name as the PCML file. For example, if the PCML file is named `SYS830B2.pcml,` then the project that is created is named **SYS830B2.developer**.

5   Double click the developer project to open the developer project that was created in the Step 4.

6   Select the PCML node and set the Action in the property view. From the drop down, accept the default, **Add**, or select **Change**, **Delete**, **Replace**, or **Create**.

7   Select the PCML Entry Point node. If the program is a service program set the **is Service Program** property to `true`. Otherwise leave all default values.

8   Select each node in the project and define the properties. The properties are listed below.

    Description

    Optional. Specify a simple description of the field. The process instruction does not use this field.

    Init

    Optional. Define an initial value for the field.

    Length

    This is the field length as defined in the RPG program.

    Name

    The RPG column. Do not change this name.

PCML Parm Types

This is not used.

Precision

The precision is set when you create the project.

Type

The type is set when you create the project.

Usage

Specify the field usage. Valid entries are: **inputoutput**, **input**, **output**, or **inherited**.

Xpath

The Xpath is a string that cannot contain any blanks. It can be a complete Xpath of an element or it can be a simple name such as Warehouse. This is the value that is used when using the API in the outbound or inbound process instruction.

9   Save the API developer project.

Generating the process instruction

Perform these setup tasks before you generate the process instructions:

1   Install the `pibuilder_tools.zip` file to the computer.

2   If you are building an API to be used by the LX Extension copy the `LXESBPI.jar` file to the PC. If you are building an API to be used by the LX Connector copy the `LXCPI.jar` file to the PC.

3   Set the **JAVA_HOME** environment variable in System Settings to point to your java SDK environment.

To generate the process instruction:

1   Right click on the API developer project and select **Generate Process Instruction** from the menu. This creates a new PCML file in the navigator pane that has the Action appended to the name. For example, `SYS830B2.pcml` is now `SYS830B2Add.pcml`. A mapping xml file having the same name but having xml as the extension is also created. Both files must be added to the API jar file. The generated PCML file must be serialized for performance reasons. To add the files to the API jar file use the Add Jar File View.

2   Select **Window > Show View > Other** to open the Add Jar File View.

3   Select Infor ERP LX Views/Add Jar File view.

4   Select the browse button in the view and navigate to the `apiAdd.pcml` file that was generated in Step 1.

5   Set the PIBuilder tools directory to point to the location that the pibuilder_tools were installed to.

6 Select the Jar file to add the serialized file into. Select the `LXESBAPI jar` file from the project folder that you copied to your PC.

7 In the Generate drop down, select **Serialize Pcml** and click **Add**.

8 Select the xml file that was generated and put it in the same JAR file. Set the Generate drop down to `blank` and click **Add**.

After you add the API files to the JAR file, copy the updated JAR file to the LX Extension installation directory for testing. If you are using LX Connector copy the updated `LXCPI.jar` file to the LX Connector IFS directory.

The API has to be defined as an Instruction in your outbound or inbound process instruction. See Chapter 5, Additional Capabilities, for directions to add an API instruction.

Appendix B Inbound tree view

Table A describes the nodes that can be added into the designer view tree when building the model object. All Parent Nodes have Child nodes some of which are Parents of children. A complex process instruction is created by adding child nodes to a Parent. All child nodes are added to the tree by selecting the Parent, right clicking and selecting **New Child**. This presents a menu of choices from which the developer selects.

Parent node	Child nodes
Inbound	Noun
Noun	Narrative
	Condition
	Instruction
	Thread Rule
Narrative	Copyright
	Comment
	Modification
Modification	Comment
Condition	Comment
	Conditional Instruction
Conditional Instruction	Comment
	Instruction Name
	Work Element
	Verb
	Loop Element
	If Condition
	API Instruction
	Mapping
	Huge Bod Entry

Parent node	Child nodes
Instruction	Comment
	Display Program
	Database
	Batch Program
	External Instruction
	Work Element
	Verb
	Conditional Instruction
	Outbound Message Instruction
	Data Area Instruction
	Huge Bod Batch Instruction
Thread Rule	Comment
	Work Element
Instruction Name	Comment
	Work Element
Work Element	Comment
	Concatenation Field
	Substring Field
	Reset
	Attribute
Verb (not supported for Lx Connector process instructions).	Verb Element
If Condition	Comment
	API Instruction
	Verb
	Work Element
	Loop Element
	Instruction Name
	Mapping
	If Condition
	Conditional Instruction
	Concatenation Field

Parent node	Child nodes
API Instruction	Comment
	Field
	Work Element
Mapping	Comment
	Date Time
	Attribute
	Enumerated
	Concatenation Field
	Expression
	Field
	Variable
	Simple Expression
	Instruction
	Conditional Instruction
	Inbound Path
	Mapping
Huge Bod Entry Instruction (Not supported by Lx Connector process Instrucitons)	Outbound Message
Display Program	Action Code
Action Code	Action
Action	Exception
	Forced Value
	Automated Locator
	Derive
	Locate Row
	Acknowledge
	Screen Field Mapping
	Validate Element
Locate Row	Row
Database	Database SQL Statements
Database SQL Statements	Comment
	Statement
	Conditional Name
	Instruction Name

Parent node	Child nodes
Batch Program	API Field Mapping
Data Area Instruction	Data Area Field
Outbound Message Instruction (Not supported by Lx Connector process instructions)	Verb Mapping Name Space BOD Version
Exit Point	Exit Point Mapping
Exit Point Mapping	Exit Point Definition
Exit Point Definition	Argument 1 Argument 2 Argument 3 Argument 4 Argument 5 Before Image After Image BOD Element
Argument 4	Exit Point Data
Argument 5	Exit Point Data
Before Image	Exit Point Data
After Image	Exit Point Data
BOD Element	Priority
Priority	Key Element
Outbound Message Instruction	Confirm Error Message
PCML	Pcml Entry Point
Pcml Entry Point	Pcml Data

Appendix C Inbound and outbound logging

When you generate a process instruction the LX ION PI Builder also generates an error log.

Generating an error log

Logging validates syntax in both inbound and outbound process instructions. The error logs are in XML format and are named similarly to the process instruction, for example: ContractDebugOutbound.xml. A log is generated each time you generate a process instruction.

To generate an error log:

1   In Package Explorer, right click a process instruction and select I**nfor ERP LX Process Instruction > Generate Process Instruction**.
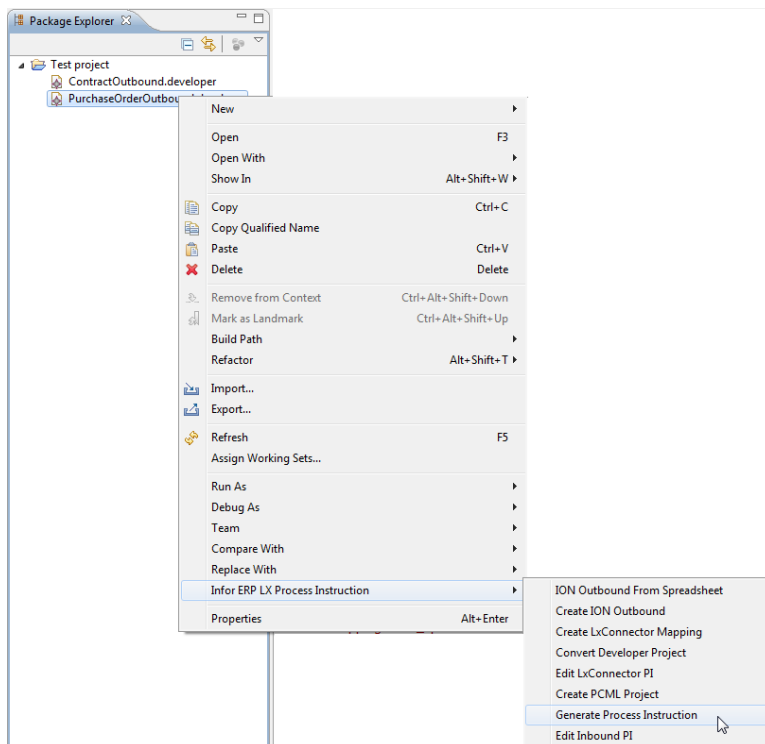
2   The LX ION PI Builder creates the process instruction and a log. Open the log in a Web browser.

3 Review the errors and, if necessary, make corrections in the LX ION PI Builder to satisfy the error condition.

Debug log

The debug log contains information about the instructions added in the process instruction. The log contains the structure of the instructions that are contained in the tree. The log attempts to determine if the user made a mistake when building the instructions. The log is in XML format and can be opened with a browser.

To find possible errors, search for the word <ERROR. Read the text of the error and then verify if it is truly an error.

These are some of the errors reported in the log:

- SQL statement may be missing a leading colon (:) that is required for a translation to occur. For example, the <ERROR> is indicating that the where clause is missing a prefix. In this case the <ERROR> is correct because we need to substitute the value for ECH.OCOLS. Change this to **:EOC.OCOCLS** in the Model Object and regenerate the process instruction otherwise the SQL will fail at runtime.

  <Statement>SELECT OCOCDS FROM EOC WHERE EOC.OCOCLS = EQH.CHOCLS</Statement>

  <ERROR>Where clause missing prefix : on variables</ERROR>

  <ERROR>Where clause missing prefix : on variables</ERROR>

- Bad expressions in the If Condition. For example the error below indicates that you have created an invalid If expression that may fail at runtime. An Expression cannot contain blanks.

  <ifcondition>

  <ERROR>No blank characters, single or double quotes allowed in expression</ERROR><expression>

  ((EST.SQLErrorCode==Success) && (DUN9==*BLANKS))</expression>

  Change the expression and remove the blanks in front of the &&. It should be
  `((EST.SQLErrorCode==Success)&&(DUN9==*BLANKS))`

- SQL Definition <ERROR> messages can be ignored with version 1.0.0 of the LX ION PI Builder.

- Element not found. Check the Mapping property page to confirm that the Element has a complete path.

Appendix D IDF System PI

Note: This Appendix is not applicable if developing process instructions for versions of LX Extension or LX Connector prior to version 3.0.

Both LX Connector 3.0 and LX Extension 3.0 support communications with Infor IDF Development Framework using the IDF System-Link web service. The process instructions for this type of message does not map to green screen panels but instead map to properties of the IDF object. This Appendix provides instruction on how to modify a existing process instruction that maps to green screen panel fields to a IDF System-Link process instruction that maps to IDF object properties. Important: An IDF object must exist.

The ItemMaster is an Extension 3.0 process instruction that was modified to use the Infor IDF object Enterprise Item. As a general starting point when modifying an existing process instruction

- Try to change only those instructions that map the panel screens
- Change only those mappings instruction that Create, Change, Delete or Replace an ItemMaster.

To demonstrate how to modify the ItemMaster developer project open the Ext 2.2 version of the ItemMaster developer project if it is available. The instructions below demonstrate how the EX 2.2 ItemMaster was modified.

First step is to navigate to the Instruction PROCESSITEM. Open the properties page for the Instruction and set the Is System Lin property to true. This is important because if this is not set the IDF System-Link message will not work.

Modification process
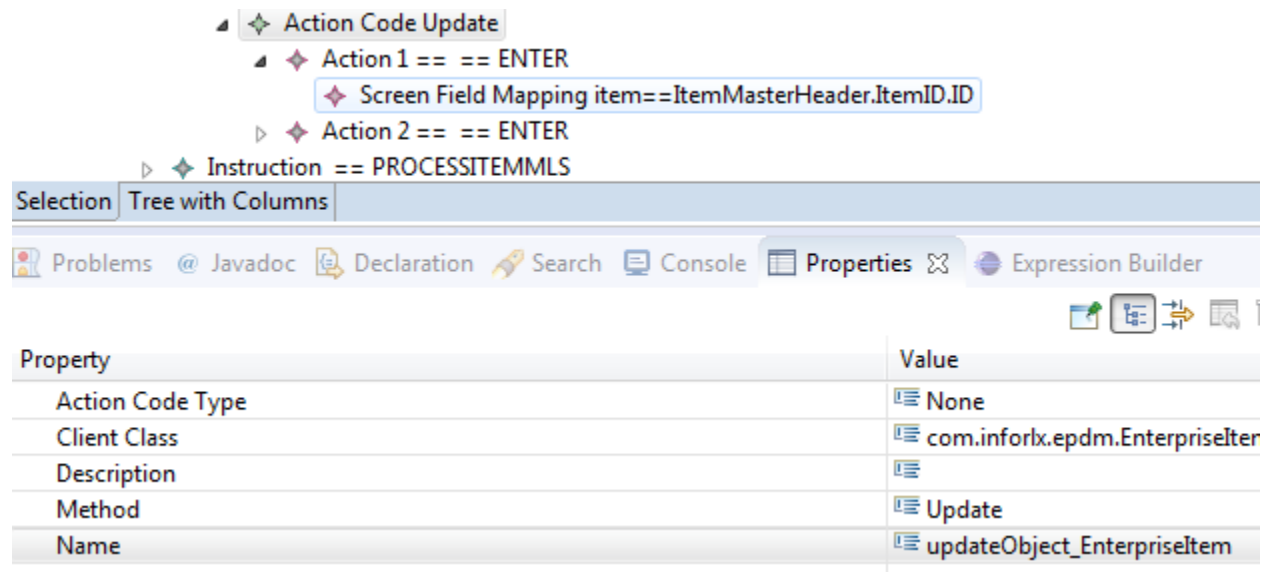
Expand the PROCESSITEM Instruction and then expand the Display Program node. This exposes several Action nodes that have child Screen Field Mapping nodes.

The modified IDF System-Link ItemMaster needs only 2 Actions. Copy all of the Screen Field Mapping nodes contained in Actions 3 through 13 into action 2. Then delete Actions 3 through 13. After having copied all of the Screen Mapping fields into Action 2 delete and Exception or Foreced Value nodes from Action 2. When you have finished there are only 2 Actions. The first Action will be used for passing the Key data and the second Action will map all of the elements that are available in the integration message to properties of the IDF object Enterprise Item. The picture below shows the modification.

After creating the two Action Instructions, modify the parent node (Action Code) of the two Actions Open the properties page for Action Code. Notice that the properties are all empty. Since the Action Code Type is a Replace set the Method property to Update. The Client Class is the complete name of the IDF object EnterpriseItem. Set the Client Class to the complete path to (com.inforlx.epdm.EnterpriseItem). The name is name of the method that is called by the EnterpriseItem. So in this case it is updateObject_EnterpriseItem.

Next step is to map the key to the IDF ItemEnterpise property in Action 1. Select Action 1 and open the property page for it. Since this is the EnterpriseItem key property set the property Domain Entity Key to true. This is an important step, because if the Domain Entity Key is not set to true the request message sent to IDF System-Link will not work.

Now select the Screen Field Mapping in Action 1. If an IDF object has more than 1 key property all of them must be defined in Action 1 as a Screen Field Mapping. For the EnterpriseItem there is a single key hence only 1 Screen Field Mapping instruction. Select the Screen Field Mapping and open the property page for it. Modify the Field Name property which is currently mapped to a green screen panel field. Change this Field Name to the name of the key property in EnterpsieItem. That Field Name should be changed to item (check with IDF developer as to the property names). In the screen shot below the Size Validation property is set to reject which means if the value for item is longer than 35 characters the message is rejected and an error is returned.

Now modify all of the Field Name properties for each Scree Field Mapping under Action 2. Change the value to the IDF EnterpriseItem property. It is suggested to map only those elements that are required for the integration project you are working on. This modified ItemMaster developer project is released with the install of the LX Extension 3.0 and is delivered in the PI_Mapping folder of the installed IFS directory.

After all of the Fields are changed to the appropriate EnterpriseItem property you are done with the mapping. Generate the process instruction and copy to the IFS directory where LX Extension has been installed to test.

Appendix E Field expansion support

Note: This Appendix is not applicable if developing process instructions for versions of LX Extension or LX Connector prior to version 3.0. Important Note: This Appendix is not applicable if developing process instructions for versions of LX Extension or LX Connector prior to version 3.0.

To support expanded fields in LX 4.0 more properties where added to some of the instructions.

The instructions below are used when building outbound process instructions.

This appendix does not apply to Lx Connector outbound process instructions.

Modified instructions.

The new properties shown below should be used only for those elements that refer to document reference elements such as DocumentID.ID. The document referenced is stored in the LX Extension SOA Cross Reference (XID) file. This file is used to determine the correct value to assign to an attribute or element value in version LX EX 3.0.

Class Type XidReference is an indicator to the extension runtime that data in the XID file will be used in determining the values assigned to an element or its attributes. For example the document reference for a PurchaserOrder is defined in element DocumentID.ID. This element has an accountingEntity attribute that's value is mapped to the LX company. The company is a field that has been expanded from length 2 to 3. In addition the value of the DocumentID.ID is mapped to a field that has expanded from 8 to 9. Because of the expansion the developer should define the Class Type property in the Mapping for this element to XidReference and set the Noun to PurchaseOrder. Because of the expansion the developer uses a Concatenation field to hold the value for the accountingEntity attribute and a ConcatenationField to hold the value of the DocumentID.ID. Since both the accountingEntity attribute and the DocumentID.ID should have leading zeroes the developer sets the Add Leading Zeros and the Add Leading Zeroes New Field Size to true in both the accountingEntity ConcatenationField and the DocumentID.ID ConcatenationField. Since the accountingEntity has expanded from 2 to 3 characters the developer sets the Number Of Characters to 2 and the Number Of Characters New Field Size to 3 in the ConcatenationField for the DocumentID.ID element.

Mapping Instruction – The new properties are

- Class Type – added type XidReference that is an instruction to the runtime that when building this element it will need to check the XID file for data. This should be used on all elements in the bod being produced that are references to other BODs.

- Noun – This is a drop down list that contains all of the current Infor supported nouns. This should only be set if the Class Type is set to XieReference.
- Noun Name – If the Class Type is XidReference and the noun does not exist in the drop down list enter the name of the Noun.
- ConcatenationField – The new properties are
- Add Leading Zeroes – Set this to true if the previous length should include leading 0's.
- Add Leading Zerios New Field Size – Set this to true if the new length should include leading 0's.
- Number Of Characters – the previous length of the value. For example 2 if this is mapping the accounting Entity.
- Number Of Characters New Field Size – The expanded length for the element. For example if mapping accountingEntity it is 3.
- Pad With Blanks – Set this to true if the previous length was padded with trailing blanks
- Pad With Blanks New Field Size – Set this to true if the new length should be padded with trailing blanks

Xid Reference Rules

The Extension runtime processes Class Type XIdReference marked elements using these rules:

- Check existence in the XID file for the given noun having a value a Number Of Characters in length.
- If found the length of the elements value will be Number Of Characters as will the attributes assigned to the element.
- If a row is not found for the Noun having a length of Number Of Characters the value assigned to the element is the Number Of Characters New Field Size. If the element has an accountingEntity attribute the runtime checks the existence of a row having noun AccountingEntity and a value of lenght Number Of Characters. If the row is found the element attribute accountingEntity is assigned a length of Number Of Characters. If the row was not found the length of the value assigned to the accountingEntity is Number Of Characters New Field Size.
- If the Element has both an accountingEntity and location attribute the length of the value assigned to the location will be the same.

Example of the rules

- XID has accountingEntity = 02
- XID has location 1-02
- XID noun is PurchaseOrder
- DocumentiD.ID = 34438333 Number Of Characters = 8 Number Of Character New Field Size = 9
- AccountingEntity Number Of Characters =2 Number Of Characters New Field Size = 3

- location Number Of Characters = 2 Number Of Characters New Field Size = 3
- Element DocumentID.ID has Class Type XIdReference
- Element DocumentID.ID has Class Type XIdReference
- The runtime checks for existence of a row in the xid for Purchase order having a value of 34438333.
- No row is found so the runtime sets the DocumentID.ID value to 034438333 (Number Of Characters New Field Size)

Runtime checks the XID for an AccountingEntity havina a value of 02. Since the XID does have an AccountngEnttity of 02 the attribute accountingEntity is set to 02 and the location is set to 1-02.

Appendix F New instructions

Two new instructions have been added for developer use.

Comparison Work Element

The Comparison Work Element is added as an instruction when a decision based on the length of the Elements mapped field value determines whether an element should be published in the BOD being produced.

The Comparison Work Element has the following properties:

- Allow Blanks – If this is set to true it allows a blank value for the Element to publish
- Comparison Operator – drop down list of possible operators. The available operators are Equal, NotEqual, Greater, GreaterEqual, Less, and LessEqual.
- Description – This is not published in the process instruction but add information in the project.
- Length – The number of characters that the ComparisonOperator is checking.
- Value – The field that is being compared

For example say an element named IMUPC is mapped to field IIM,IMUPC and the developer only wants this element published to the Item BOD if the Value for field IIM.IMUPC retrieved from an sql of the IIM is LessEqual to Length of 12. Adding this instruction causes the runtime to examine the Value of IIM.IMUPC and determines the value is LN7181601UPC, then checks if the Length of the Value is <- 12. In this ccase the lenghtof LN7181601UPC is 12 so the element is published to the Item bod as `<IMUPC> LN7181601UPC</IMUPC>`. If this comparison operator had failed then the `<IMUPC>` element would not be published in the bod.

For an example of a process instruction using this instruction see the ItemMasterOutbound delivered in the PI_Mapping folder of EX 3.0 install folder.

Array Instruction

The Array Instruction is used by developers when a database field returns an array of characters where each character in the array needs to be inspected. This was required in the Shop Calendar defined in the FinancialCalendar outbound BOD.

The properties of the instruction are:

- Array Element – This is the parent name of the Element that is outputting the array data.
- Array Field – This is the name of the field that holds the array of data
- Array Index Value – this holds the value of the current index.
- Array Index Variable – this holds the current index
- Array Size – This holds the field length.
- Increment Array Index – this holds the number of positions to increment the index by

An example of use is when an sql returns a field FSC.SCTYPE that holds an array of information. In this case this field holds Calendar information. The SCTYPE has a field length of 366 characters. In this example, each index represents a day in a year. In this case the developer would add an ArrayInstruction so they can iterate over each indexed value of the field. In this example, the properties are set as follows:

Array Size is 366, the Increment Array Index is 1 so the runtime increments this index by 1, the Array Field is FSC.SCTYPE, the Array Element will be set to the Name of the Parent element that holds child elements representing Calendar information, For this example the Parent Element will be set to Period. The Array Index value is work element that holds the Current value stored in the current index. This is defined by the developer and this example it is set to WKPRDVALUE. This will be populated by the runtime with the value in the current index, for example if the fist character has a W in it the WKPRDVALUE is set to W. This field is updated with every iteration of the array. The same applies for Array Index Variable this is defined by the developer as a work element that holds the current index for example 1. In this example, this is set to WKPRDINDEX. So every iteration will increase this by 1. The instruction is inspecting each index value and from that value determines the information that should be displayed for the Period. Please seen the FinancialCalendar Instruction ShopCalendar for an example of using this instruction. This process instruction is located in the PI_Mapping folder of the EX 3.0 install IFS directory.