



IBM Software Group

WebSphere® Enterprise Service Bus V6.1
WebSphere Process Server V6.1
WebSphere Integration Developer V6.1

Custom mediation primitive



@business on demand.

© 2008 IBM Corporation
Updated June 13, 2008

This presentation provides a detailed look at the custom mediation primitive.

Goals

- Understand the custom mediation primitive details



Custom mediation

- ▶ Overview of function
- ▶ Use of terminals
- ▶ Definition of properties
- ▶ Basic code samples
- ▶ Changes since and migration from V6.0.2
- ▶ Error handling
- ▶ Examine custom code sample



The goal of this presentation is to provide you with a full understanding of the custom mediation primitive.

The presentation assumes that you are already familiar with the material presented in the **Mediation primitive common details** presentation and the **Common details – Promoted properties** presentation. These two presentations serve as a base for understanding mediation primitives in general.

In this presentation, an overview of the custom mediation primitive is provided along with information about the primitive's use of terminals and its properties. There are some small coding samples provided that are useful for helping you understand the basics about developing code for a custom mediation primitive. This primitive has changed considerably between version 6.0.2 and version 6.1. A review of those changes is provided along with details about how to migrate custom mediation primitives developed in version 6.0.2 to implementations that are compatible with version 6.1. Some error conditions you might encounter are described. Finally, a realistic example examining some of the new capabilities of custom mediation primitives is presented, including the complete code used to implement the example.

Overview of function

- Enables the use of custom mediation logic
 - ▶ Use when no built-in primitive provides a needed function
 - ▶ Logic might be simple tasks (such as printing to system log file)
 - ▶ Also enables complex routing and transformation logic
- Logic implemented in Java™ and defined as a:
 - ▶ Visual snippet
 - ▶ Java snippet
- Access to entire SMO using:
 - ▶ ServiceMessageObject APIs
 - ▶ DataObject APIs
- Variable number of user defined input and output terminals
- User defined properties that are promotable



The custom mediation primitive enables you to define your own custom mediation logic for use when the built-in primitives do not provide the needed functionality. In some cases this might be simple logic such as printing a message to the system log file. However, the custom mediation primitive is able to be used for complex transformation and routing logic, which enhances the overall possibilities for what you can do in a mediation flow.

The logic in a custom mediation primitive is implemented in Java. There are two approaches to doing this, using either a visual snippet or a Java snippet.

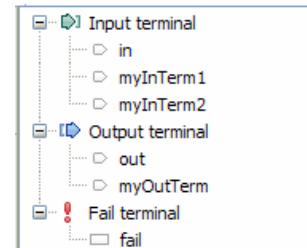
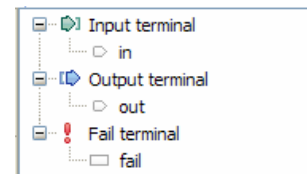
Within a custom mediation primitive, you have access to the entire SMO, which is both a ServiceMessageObject and a DataObject type. This enables you to use either the strongly typed ServiceMessageObject APIs or the loosely typed DataObject APIs for accessing and manipulating the contents of the SMO.

The custom mediation primitive allows you to define how many input and output terminals you require, each of which can support a different message type. This is one of the key elements that makes the custom mediation primitive capable of complex transformation and routing logic.

The ability to define user properties which are promotable enables you to expose aspects of your custom mediation primitive implementation so that its behavior can be administered at runtime.

Terminals

- Input terminals
 - ▶ One by default
 - ▶ From one to n allowed
- Output terminals
 - ▶ One by default
 - ▶ From zero to n allowed
- Fail terminal
- Message types:
 - ▶ All input and output terminals can be of different types
 - ▶ Fail terminal same type as the default in terminal



This slide examines the use of terminals for the custom mediation primitive. The upper two screen captures show the default terminal configuration. The bottom two screen captures show a custom mediation primitive with additional terminals configured.

The custom mediation primitive has one input terminal by default. It is required to have at least one input terminal but can be configured with any number of additional input terminals.

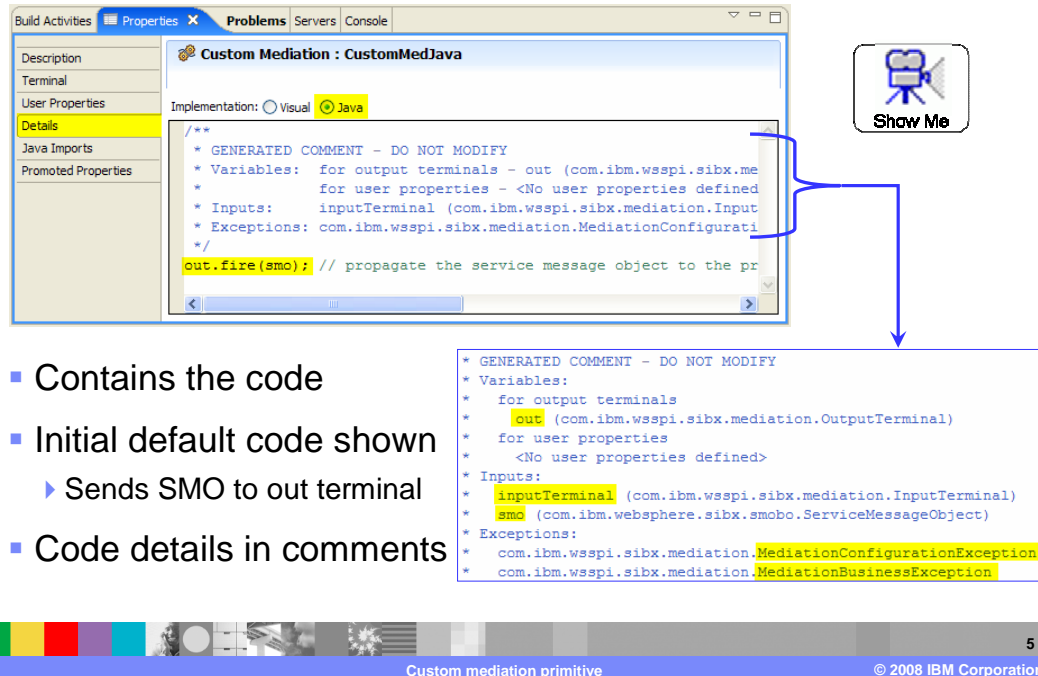
There is one output terminal configured by default. Like input terminals, there can be any number of output terminals configured. However, unlike input terminals, there is not a requirement for an output terminal. Therefore, you can delete the default output terminal if it is not needed by your implementation.

As with all mediation primitives, there is a fail terminal used when the custom mediation encounters an exception during its processing.

The message types of all the input and output terminals are allowed to be different. When the message type of the input terminal is different from the output terminal, the code in the custom mediation must modify the SMO structure. The message body of the SMO must conform to the output terminal's message type.

The fail terminal has the same message type as the default input terminal.

Properties – Details - Java implementation



The screenshot shows the 'Properties' view for a 'Custom Mediation : CustomMedJava' primitive. The 'Details' tab is active, displaying the Java implementation code. A 'Show Me' icon is located to the right of the code editor. A callout box provides a detailed view of the code comments.

- Contains the code
- Initial default code shown
 - ▶ Sends SMO to out terminal
- Code details in comments

```

/**
 * GENERATED COMMENT - DO NOT MODIFY
 * Variables:
 *   for output terminals
 *   out (com.ibm.wsspi.sibx.mediation.OutputTerminal)
 *   for user properties
 *   <No user properties defined>
 * Inputs:
 *   inputTerminal (com.ibm.wsspi.sibx.mediation.InputTerminal)
 *   smc (com.ibm.websphere.sibx.smobo.ServiceMessageObject)
 * Exceptions:
 *   com.ibm.wsspi.sibx.mediation.MediationConfigurationException
 *   com.ibm.wsspi.sibx.mediation.MediationBusinessException
 */
out.fire(smc); // propagate the service message object to the pr
  
```

Custom mediation primitive © 2008 IBM Corporation

The next few slides examine the various properties that are associated with a custom mediation primitive. The Details panel of the Properties view is where you select the implementation type and provide the code for the implementation. This slide looks at the Java snippet implementation. There is a text entry box which behaves like a mini Java editor, providing features such as content assist and context sensitive highlighting.

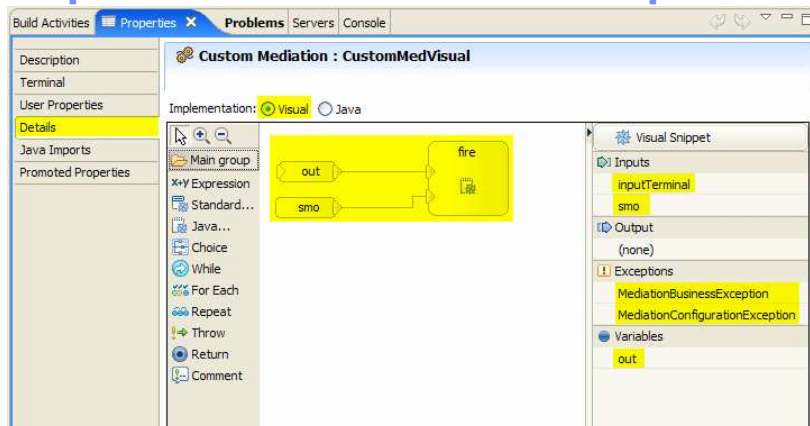
The code shown in the text entry box is the actual default implementation that appears when you first create a custom mediation with a Java implementation. There are several comments and one line of code provided. This discussion starts with the comments, because they provide valuable information useful to you when writing code. Notice in the text box, the comments start by stating "GENERATED COMMENT – DO NOT MODIFY". It is best to follow this advice, because when you modify the comments, the editor will place a new copy of these comments into the source before the copy that you modified. However, because the comments use long lines, you can see on this slide a copy that was extracted and edited so that some of the important items in the comments can be brought to your attention. First the variables for the output terminals and user properties are documented. In this case, there is one default output terminal named out, and no user defined properties. Next are the inputs, the inputTerminal and the SMO. The inputTerminal object enables you to determine information about the terminal that the incoming message flowed through. This is useful when you have multiple input terminals. The SMO input contains the entire SMO typed as a ServiceMessageObject. Finally, the MediationConfigurationException and the MediationBusinessException are listed, as these are the two exceptions that you can raise from your code.

These comments are dynamic. When configuring the custom mediation primitive, any new user properties or output terminals that you define are automatically added to the comments section.

The single line of code provided takes the smc and fires it on the output terminal. Typically, this code is needed as is, unless your implementation has modified the number of output terminals defined for the primitive.

Notice the Show Me icon in the slide. This is a link to a demonstration illustrating how to build a custom mediation primitive with a Java implementation.

Properties – Details - Visual implementation



- Initial default code shown - sends SMO to out terminal
- Same inputs, variables and exceptions as Java implementation
- Construct logic on canvas, dragging from palette and tray
- See Information Center - Customizing behavior with visual snippets:

<http://publib.boulder.ibm.com/infocenter/dmndhelp/v6r1mx/topic/com.ibm.wbit.610.help.activity.ui.doc/topics/cactint.html>



The visual snippet implementation provides a visual snippet editor in which you can create the logic for your custom mediation primitive. It consists of a palette, canvas and tray. In the center is the canvas, which is where you build up your logic through connecting visual constructs that provide various different capabilities. The shaded area to the left of the canvas is the palette. It contains icons representing the various constructs used to define the logic of your snippet. You click on these icons, drop them onto the canvas and wire them together to build your snippet. On the right is the tray, from which you can drag inputs, variables and exceptions onto the canvas. Notice that the tray contains the same items that are described in the comments for a Java implementation.

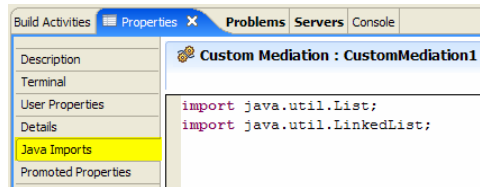
The default implementation provides the same functionality that the Java implementation does, code to fire the output terminal passing it the SMO.

Notice the Show Me icon in the slide. This is a link to a demonstration illustrating how to build a custom mediation primitive with a visual implementation.

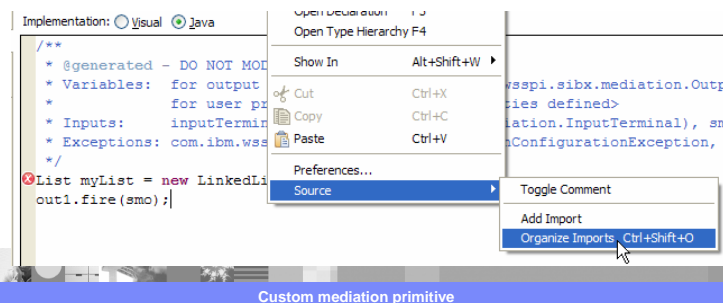
Usage of the visual snippet editor goes beyond the scope of this presentation. You should see the Information Center topic entitled “Customizing behavior with visual snippets” to learn how this editor is used.

Properties – Java Imports

- Contains imports needed to compile

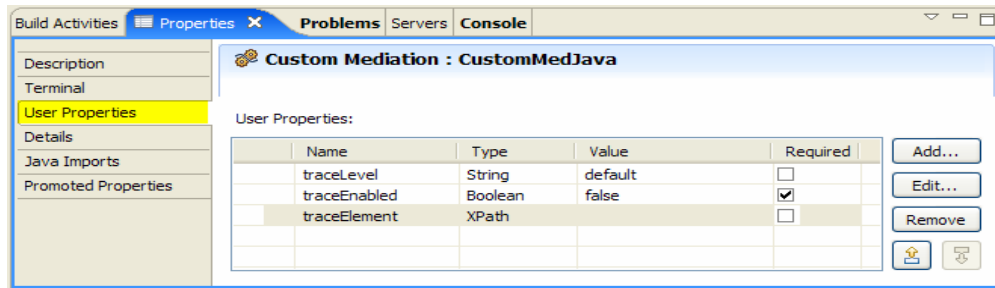


- Can be directly typed into field
- Can be set using Source -> Organize Imports



The Java Imports panel in the Properties view provides a text box where Java import statements are listed. These are the imports of classes needed by your implementation. These import statements can be typed directly into this field. However, it is generally easier to let the Java editor, on the Details panel, do this work for you. When writing Java code, any class referenced that the editor can't resolve is flagged as an error. From the editor pop-up menu, select Source and then Organize Imports. The imports needed will automatically be added. In the case where there are multiple classes with the same name, you are presented with a list of fully qualified class names from which you can select the required class to be imported.

Properties – User Properties

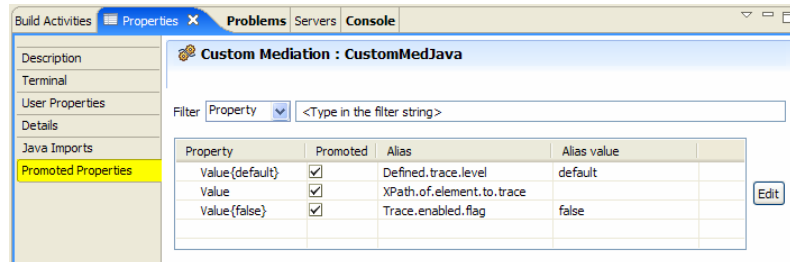


- User properties are accessible in code as Java variables
- Can provide initial value
- Can be required
- **Restriction:** Name must start with lower case character
- Types and Java types:
 - ▶ String String
 - ▶ Boolean boolean
 - ▶ Integer int
 - ▶ Float float
 - ▶ XPath String



The User Properties panel allows you to define user properties using a table based entry area. These user properties appear in your code as variables. The table provides you with columns to define the property name, its type, value and if it is a required property. The value specified in this table is used as the initial value for the property in your custom mediation code. Note the restriction that property names must begin with a lower case character. WebSphere Integration Developer does not enforce this restriction, but properties beginning with upper case letters cause runtime exceptions to occur when the mediation flow is initialized. The types that can be specified for a property are shown on the slide, along with the corresponding Java type that is used for the property in the code.

Promoted Properties



- User property values are promotable
- Provides ability to apply runtime administrative control of behavior
 - ▶ This is the key purpose of promoted properties
- No other properties are promotable



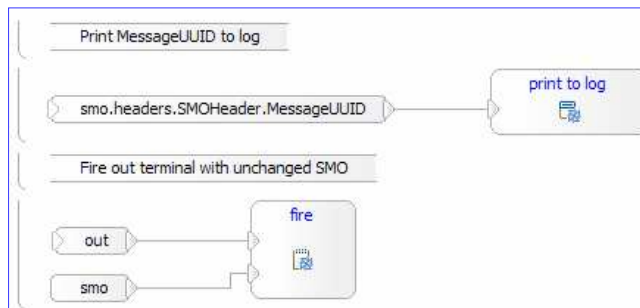
The Promoted Properties panel allows you to specify which of the custom mediation primitive's properties are promoted. The only properties that are promotable are the user properties. Consistent with the promotion of a table based property, only one column is actually promotable. In this case, it is the value column. Individual user properties can be selected to have their value promoted.

Promotion of user properties provides a mechanism to enable administrative control over the behavior of a custom mediation primitive. The code in the custom mediation can make decisions based on user property values that are settable by administrators at runtime. In reality, it is this capability that makes user properties useful. A user property that is not promoted can just as easily be defined directly in the Java code rather than as a property.

Sample code accessing SMO

```
// Print MessageUUID to log
HeadersType headers = smo.getHeaders();
SMOHeaderType SMOHeader = headers.getSMOHeader();
String uuid = SMOHeader.getMessageUUID();
System.out.print(uuid);

// Fire out terminal with unchanged SMO
out.fire(smo);
```

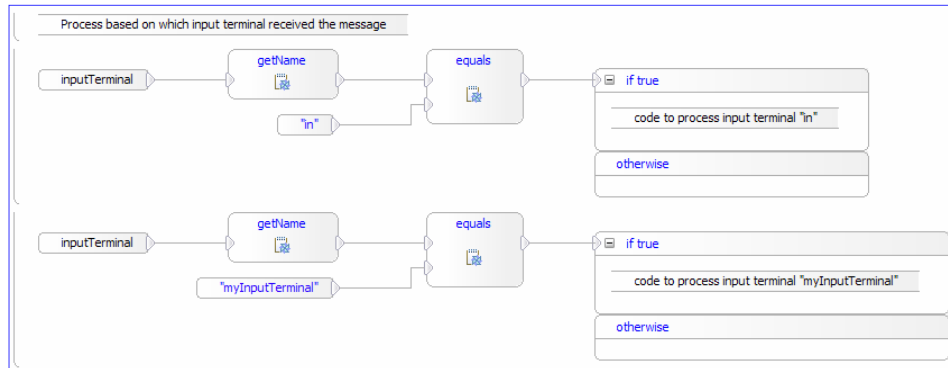


This slide shows some simple code illustrating both Java and visual implementations of a custom mediation primitive. These are not code segments, but show the entire implementation of the primitive. The code illustrates the use of the ServiceMessageObject APIs to access the UUID for the message, which is contained in the SMOHeader. It is then printed to the SystemOut.log file, which also appears in the Console view of WebSphere Integration Developer. The SMO is then fired on the output terminal named out, passing the unchanged SMO.

Sample code determining input terminal used



```
// Process based on which input terminal received the message
if (inputTerminal.getName().equals("in")) {
    // code to process input terminal "in"
} else if (inputTerminal.getName().equals("myInputTerminal")) {
    // code to process input terminal "myInputTerminal"
}
```



This is another code sample showing both the Java and visual implementations. In this case, only code segments are shown, rather than the entire implementation of the primitive.

Notice that the custom mediation primitive in the upper left has two input terminals. These code segments provide the code needed to determine which input terminal received the message, so that the appropriate processing can be done. The code uses the `inputTerminal` object to access the name of the terminal through which the message was received. This is then used in conditional statements checking if the terminal named "in" or "myInputTerminal" was used, thus determining what processing is required.

Sample code determining input message types and context types

```
// Get the ESBTerminalType for the input terminal
ESBTerminalType esbTT = (ESBTerminalType)inputTerminal.getType();

// Get the fully qualified message type of the input message body
QName qn = esbTT.getBodyType();
String msgType = qn.toString();
System.out.println("Message type of body    = " + qn.toString());

// Process based on message type
if (msgType.equals("{http://StoreLib/Ordering}submitOrderRequestMsg")) {
    // code to process submit order request message
} else if (msgType.equals("{http://StoreLib/Shipping}shipOrderRequestMsg")) {
    // code to process ship order request message
}

// Get the context types and print out
qn = esbTT.getTransientContextType();
if (qn != null)
    System.out.println("Transient context type = " + qn.toString());
qn = esbTT.getCorrelationContextType();
if (qn != null)
    System.out.println("Correlation context type = " + qn.toString());
qn = esbTT.getSharedContextType();
if (qn != null)
    System.out.println("Shared context type    = " + qn.toString());
```

```
Message type of body    = {http://PortfolioLibrary/CustomerService}getCustomerInformationRequestMsg
Transient context type  = {http://CustomerRoutingMediationModule}GetCustInfoTransientContext
Shared context type     = {http://StoreMediation}SharedCtxBO
```

12

Custom mediation primitive

© 2008 IBM Corporation

This sample is only shown in the Java implementation. It is used to illustrate code that can determine the message type of the incoming SMO and determine the existence and type of the transient, correlation and shared contexts.

Similar to the previous example, the `inputTerminal` is the starting point. This time it is used to obtain an `ESBTerminalType` object describing the input terminal through which the message arrived. From this the body type can be determined, which is the same as the message type of the SMO. In this example, the fully qualified type is printout out.

The next section of code is used to check the message type of the SMO and make a decision about what processing to do based on the type. This is very similar to the previous example, except the message type of the input terminal is used rather than the name of the input terminal.

The last section gets the type for each of the transient, correlation and shared contexts. Since these contexts are optional, the code needs to check to see if a null is returned from the `ESBTerminalType` object when getting the context type. If a type is returned, the code prints it out.

The bottom of the slide shows sample output from when this code segment is run. In this particular case, there was no correlation context, so you see the output for the message type, transient context and shared context, but not for the correlation context. Notice that in the printed output and in the code that is checking for the message type, the fully qualified type names are being used.

Understanding the firing of output terminals

- Code to fire a terminal can be anywhere in code logic
 - ▶ Contents of SMO at this time defines what is propagated
 - ▶ Terminal does not actually fire until code completes
- Multiple output terminals can be fired from the code
 - ▶ Actual fire of terminals happens in the same order

```
smo.setString("/body/getCustomerInformation/customerID","cust001");
outTerminal_A.fire(smo); // outTerminal_A is not really fired now

smo.setString("/body/getCustomerInformation/customerID","cust002");
outTerminal_B.fire(smo); // outTerminal_B is not really fired now

// outTerminal_A is now fired with customerID=cust001
// flow from outTerminal_A is followed until it is complete
// outTerminal_B is now fired with customerID=cust002
// flow from outTerminal_b is now followed
```



This slide is used to highlight an important concept that you need to understand regarding the firing of an output terminal. The code to fire an output terminal can appear at any point in your custom mediation primitive's logic. The contents of the SMO at the time the fire method is invoked on the output terminal determines the content of the SMO that is propagated through that terminal. However, the SMO is not actually propagated through the terminal at that time. It is propagated after all of the code in the custom mediation has completed.

If there are multiple output terminals that should be fired, the fire method can be invoked on each of them. When the custom mediation code completes, the SMOs are propagated through these terminals in the order the fire methods were invoked in the code.

The sample code, along with the comments, illustrate the processing that occurs. In this code, the customerID field in the SMO body is set to a value of cust001 and then the terminal outTerminal_A is fired with the SMO. Then the customerID field in the SMO body is set to a value of cust002 and outTerminal_B is fired. In actuality, neither SMO has been propagated through either of these terminals. Now the custom mediation is complete, so the SMO containing a customerID of cust001 is propagated through terminal outTerminal_A. The mediation flow continues along the path wired from that terminal. Once that flow path completes, the SMO containing a customerID of cust002 is propagated through terminal outTerminal_B and the flow continues along that path.

Changes from V6.0.2

- Enhancements for custom mediation primitives

V6.0.2	V6.1
One input terminal	One to n input terminals
One output terminal	Zero to n output terminals
SMO as DataObject	SMO as ServiceMessageObject
Returned SMO propagated through out terminal	SMO to be propagated explicitly fired on any output terminal or terminals
Entire SMO or just the body	Always the entire SMO
Snippets – Java and visual	Snippets – Java and visual
Invoke option with fixed method signature	Invoke not supported – replaced with service invoke primitive usable with any method signature
No user defined properties	User defined properties that are promotable
No reuse	Reuse by copy/paste or configured primitive

14

Custom mediation primitive

© 2008 IBM Corporation

There were many significant changes that occurred to the capabilities of the custom mediation primitive between versions 6.0.2 and 6.1. This slide summarizes those differences and the subsequent slides look at migration considerations when moving a custom mediation primitive implementation from version 6.0.2 to version 6.1.

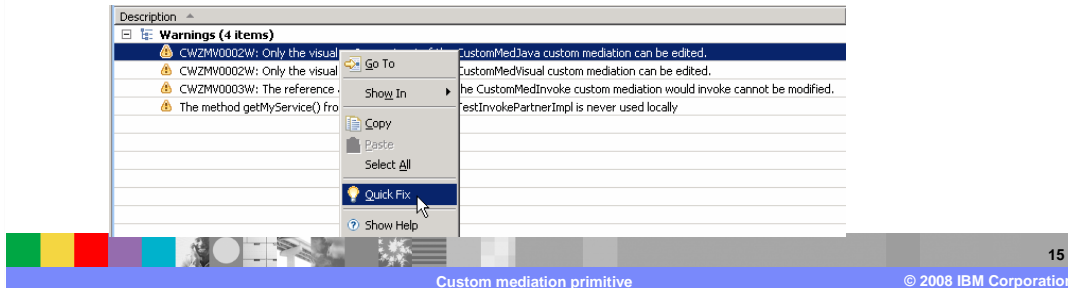
Rather than go through the table line by line, the key things to highlight are these. The use of terminals is much more flexible, allowing variable numbers of them and enabling control over the firing of the output terminals. The handling of the SMO now is consistent, with the entire SMO treated as a ServiceMessageObject, as opposed to a DataObject that might be the body or the entire SMO. The removal of the invoke option with a fixed signature from custom mediations has been replaced with the service invoke primitive enabling flexible method signatures. User defined properties that are promotable have enabled the administrative control of a custom mediation primitive's behavior. And finally, the ability to reuse a custom mediation primitive's implementation has been enabled through copy and paste.

Migration

- Migrating a V6.0.2 custom mediation to V6.1
 - ▶ Warning messages are produced
 - The warning messages are worded very poorly
 - Not clear what they are trying to tell you
 - Visual and Java have same message, Invoke has a different message

Description	Resource	Path	Location
Warnings (4 items)			
CWZMV0002W: Only the visual or Java snippet of the CustomMedJava custom mediation can be edited.	Mediation1.mfc	CustomMedExamples	CustomMedJava
CWZMV0002W: Only the visual or Java snippet of the CustomMedVisual custom mediation can be edited.	Mediation1.mfc	CustomMedExamples	CustomMedVisual
CWZMV0003W: The reference and the operation that the CustomMedInvoke custom mediation would invoke cannot be modified.	Mediation1.mfc	CustomMedExamples	CustomMedInvoke

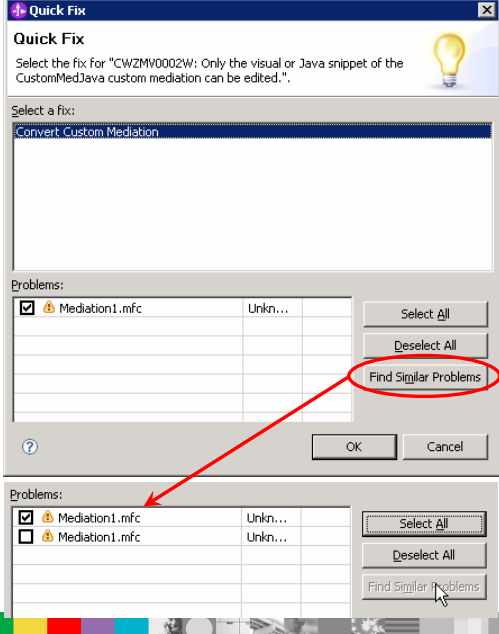
- ▶ Right click on message to get pop-up menu, select Quick Fix



The migration of a custom mediation primitive from version 6.0.2 to version 6.1 is quite easy. However, since the user interface is not at all intuitive, the next couple of slides explain the process.

When you import a project into WebSphere Integration Developer that contains custom mediation primitives developed in version 6.0.2, warning messages are produced. The message text is not clear about what the warning is trying to tell you. For a custom mediation primitive with a Java or a visual implementation, the message states: "Only the visual or Java snippet of the XYZ custom mediation can be edited", where XYZ is the name of the custom mediation. For an invoke implementation, the message states: "The reference and the operation that the XYZ custom mediation would invoke cannot be modified", with XYZ again being the name of the custom mediation. Custom mediations from version 6.0.2 that produce these messages are able to run as is, but it is generally better to migrate them to the version 6.1 style. To do that, get the pop-up menu from the warning message and select the menu item Quick Fix, opening a dialog which is described on the next slide.

Migration – Quick fix dialog



Quick Fix

Select the fix for "CWZMV0002W: Only the visual or Java snippet of the CustomMedJava custom mediation can be edited."

Select a fix:
Convert Custom Mediation

Problems:

<input checked="" type="checkbox"/>	Mediation1.mfc	Unkn...
<input type="checkbox"/>		
<input type="checkbox"/>		

Buttons: Select All, Deselect All, Find Similar Problems, OK, Cancel

Problems:

<input checked="" type="checkbox"/>	Mediation1.mfc	Unkn...
<input type="checkbox"/>	Mediation1.mfc	Unkn...
<input type="checkbox"/>		

Buttons: Select All, Deselect All, Find Similar Problems

16
Custom mediation primitive © 2008 IBM Corporation

Quick Fix dialog

- ▶ Displays item for which Quick Fix was selected
- ▶ Find Similar Problems button used to locate others
- ▶ Dialog does not differentiate
 - Identifies the flow the custom mediation is in BUT DOES NOT identify which custom mediation
 - Therefore, you can only effectively do all of them because you don't know which individual ones to select

The quick fix dialog displays the item for which the quick fix was selected, which is the mediation primitive associated with the warning message whose pop-up menu was used. In the problems list, the mediation flow component containing that custom mediation is listed but the name of the custom mediation itself is not listed. There is a button you can use to find similar problems. The result is that the mediation flow component for each custom mediation with the same warning is listed, but again the actual custom mediation names are not listed. You then have to select which ones you want to apply the quick fix to. If you want to apply it to all of them you can select them all and proceed. However, if you want to selectively apply the quick fix, this dialog is not useful. The best approach in that case is to access this dialog for each individual custom mediation for which you want to apply the fix by using the pop-up menu from each individual warning message.

Migration – Java snippet implementation

- Results of a Quick Fix for a Java implementation
 - ▶ Basic example, showing minimum code
 - ▶ Method signatures are different
 - ▶ V6.1 always passes the entire SMO
 - Code operating on “input1” (the body) goes where shown

Implementation: Visual Java Invoke V6.0.2

Root: /body

Signature: commonj.sdo.DataObject execute(commonj.sdo.DataObject input1)

```
return (input1);
```

Implementation: Visual Java V6.1 after Quick Fix

```
/**
 * @generated - DO NOT MODIFY
 * Variables: for output terminals - out (com.ibm.wsspi.sibx.mediation.OutputTerminal)
 * for user properties - <No user properties defined>
 * Inputs: inputTerminal (com.ibm.wsspi.sibx.mediation.InputTerminal),
 * smob (com.ibm.websphere.sibx.smobo.ServiceMessageObject)
 * Exceptions: com.ibm.wsspi.sibx.mediation.MediationConfigurationException,
 * com.ibm.wsspi.sibx.mediation.MediationBusinessException
 */
DataObject input1 = (DataObject) smob.getBody();
smob.setBody(input1);
out.fire(smob);
```

Code goes here

17

Custom mediation primitive

© 2008 IBM Corporation

These next three slide look at the results of the quick fix processing for migrating a custom mediation primitive from version 6.0.2 to version 6.1. This slide starts by looking at the Java snippet implementation and is based on the minimum required code. Looking at the implementation in version 6.0.2, you can see the root property indicating that only the body of the SMO is being passed. The SMO is passed into the code as a DataObject named input1 which is returned unmodified. In the migrated version, it is the entire SMO that is passed in. The first thing that is done is to extract the body from the SMO, cast it to a DataObject and assign it to the DataObject named input1. Any code that was in the version 6.0.2 custom mediation before the return statement follows this. Because the code from version 6.0.2 might have modified input1 before returning it, the last thing done in the migrated code is to replace the SMO body with the contents of input1. The output terminal is fired with the SMO.

Migration – Visual snippet implementation

- Results of a Quick Fix for a Visual implementation
 - ▶ Basic example, showing minimum code
 - ▶ Method signatures are different
 - ▶ V6.1 always passes the entire SMO

The image displays two screenshots from the IBM Software Group migration tool. The top screenshot, titled 'V6.0.2', shows the 'Implement' dialog with 'Root: /body' and 'Signature: commonj.sdo.DataObject execute(commonj.sdo.DataObject input1)'. The bottom screenshot, titled 'V6.1 after Quick Fix', shows the visual implementation diagram. The diagram includes a 'Main group' with three components: 'getBody', 'setBody', and 'fire'. The 'getBody' component takes an 'smo' input and outputs 'input1'. The 'setBody' component takes 'input1' and 'smo' as inputs. The 'fire' component takes 'out' and 'smo' as inputs. A red arrow points to the 'setBody' component with the text 'Code goes here'. The right-hand side of the V6.1 screenshot shows a 'Visual...' panel with 'Inputs' and 'Output' sections.

This is exactly the same example applied to a visual snippet implementation, showing only the minimum possible code. Again, looking at the implementation in version 6.0.2, you can see the root property indicating that only the body of the SMO is being passed. The SMO that is passed into the code is named `input1`, which is then returned unmodified. In the migrated version, it is the entire SMO that is passed in. The first thing that is done is to extract the body from the SMO and assign it to `input1`. Any code that was in the version 6.0.2 custom mediation before the return statement follows this. Similar to the previous example, code from version 6.0.2 might have modified `input1` before returning it. Therefore, the last thing done in the migrated code is to replace the SMO body with the contents of `input1`, and then fire the SMO on the output terminal.

Migration – Invoke implementation

- Results of a Quick Fix for an Invoke implementation
 - ▶ Converted to Java implementation that calls the service defined by the reference

V6.0.2

Implementation: <input type="radio"/> Visual <input type="radio"/> Java <input checked="" type="radio"/> Invoke	
Reference:	CustomMedTestInvokePartner
Operation:	mediate
Root:	/body

V6.1 after Quick Fix

```

Implementation:  Visual  Java
/**
 * @generated - DO NOT MODIFY
 * Variables: for output terminals - out (com.ibm.wsspi.sibx.mediation.OutputTerminal)
 *            for user properties - <no user properties defined>
 * Inputs:    inputTerminal (com.ibm.wsspi.sibx.mediation.InputTerminal), smo (com.ibm.websphere.
 *            Exceptions: com.ibm.wsspi.sibx.mediation.MediationConfigurationException, com.ibm.wsspi.sibx.me
 */
Service service = (Service) ServiceManager.INSTANCE.locateService("CustomMedTestInvokePartner");
OperationType operation = service.getReference().getOperationType("mediate");
Object returnType = service.invoke(operation, smo.getBody());
smo.setBody(returnType);
out.fire(smo);
  
```

19

Custom mediation primitive

© 2008 IBM Corporation

Since there is no equivalent implementation type to the invoke option from version 6.0.2, the quick fix converts a custom mediation invoke into a Java snippet. The properties for the invoke option include the name of the reference and the operation to invoke on that reference. Similar to the other examples, only the body was being passed in version 6.0.2. The migrated code uses the ServiceManager to locate the service for the reference, and uses the service to obtain the operation type. The operation is then invoked on the service, passing it the body extracted from the SMO. The returned body from the operation is then used to update the SMO and the output terminal is then fired.

Error processing

- MediationRuntimeException thrown for
 - ▶ User property name starting with an upper case character
 - ▶ User property value inconsistent with property type

User Properties:

Name	Type	Value	Required	
BadPropertyName	String	name should start with lower case	<input type="checkbox"/>	Add...
badPropertyValue	Integer	value needs to be an integer	<input type="checkbox"/>	Edit...

- Exceptions thrown by your custom mediation code
 - ▶ You can explicitly raise a MediationBusinessException or MediationConfigurationException
 - ▶ Any other exception occurring is wrapped by a MediationBusinessException
 - ▶ Fail terminal flow taken if wired



20

Custom mediation primitive

© 2008 IBM Corporation

The error processing details and considerations are examined on the next couple of slides. A MediationRuntimeException is thrown when a property name starts with an upper case character. It is also thrown when the value for a property is inconsistent with the type specified for the property. Examples of each of these are shown in the screen capture of the user properties.

Exceptions can occur during the processing of the custom mediation primitive. In the case where you have written code that raises an exception, the exception must be either a MediationBusinessException or a MediationConfigurationException. When some other exception occurs, such as a null pointer exception, it is wrapped in a MediationBusinessException. For any of these exceptions, if the fail terminal is wired, the flow will continue from the fail terminal.

Error processing

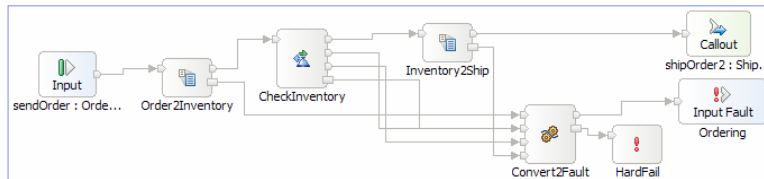
- Null or incorrectly typed SMO fired from output terminal
 - ▶ Does not produce an error at the custom mediation
 - ▶ Errors occurring downstream in the flow depend upon flow definition
- Fail terminal message type
 - ▶ Fail terminal's message type defined by default input terminal's message type
 - ▶ There can be multiple input terminals with different message types
 - ▶ The actual SMO input message is fired on the fail terminal
 - ▶ Therefore, the flow following the fail terminal cannot be dependent upon the message type of the SMO



There is no checking of the message type of the SMO that is fired on an output terminal to ensure it is compatible with the defined message type for the terminal. Therefore, when this occurs, there is no error produced at the custom mediation itself. It is likely that this will cause a problem somewhere downstream in the flow from the custom mediation. If and how that problem is manifested depends upon your flow definition.

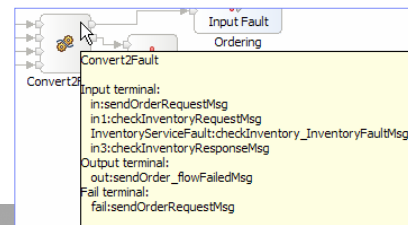
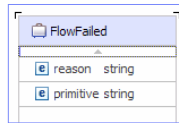
One of the error processing considerations for a custom mediation is the message type of the fail terminal. Similar to other primitives, the message type of the fail terminal is defined by the message type of the input terminal, in this case the default input terminal. However, for a custom mediation primitive there can be multiple input terminals with different message types. The message propagated through the fail terminal when an exception occurs is the same type as the input terminal through which it was received. When you have a custom mediation primitive which has multiple input terminals of different message types, the definition of the flow from the fail terminal should not have any dependencies on the message type of the SMO.

Return modeled fault for flow errors



- Input operation has a FlowFailed fault defined
 - ▶ Any problems in the flow are returned to caller using this fault
- Flow has several different message types
 - ▶ Two XSL transformations and a service invoke including a modeled fault
- Single custom mediation used to construct the fault message
 - ▶ All fail, service invoke timeout and modeled fault terminals wired to the custom mediation
 - ▶ Multiple input terminals for the various types

	Name	Type
sendOrder		
Input(s)	order	Order
Output(s)	status	string
Fault	flowFailed	FlowFailed



22

Custom mediation primitive

© 2008 IBM Corporation

These next two slides take a look at the use of a custom mediation primitive, transforming any errors that occur within the flow to a modeled fault. The modeled fault contains information about the error that occurred and is returned to the caller. This slide sets up and explains the scenario and the next slide provides a complete Java snippet implementation.

This flow is for an operation called sendOrder, which is defined in the screen capture on the lower left of the slide. Notice that the operation defines a fault called flowFailed which passes back a FlowFailed business object, which is shown on the bottom center of the slide. It contains two strings, a reason and the name of the primitive in which the error occurred.

Now focus on the flow shown at the top of the slide. The normal processing of this flow involves two XSL transformation primitives and a service invoke primitive. Each of the XSL transformations has a fail terminal as does the service invoke primitive. In addition, the service invoke has a timeout terminal and a terminal for a modeled fault that can be returned by the service. Overall, there are four different message types defined across these five terminals, all of which are wired to the custom mediation named Convert2Fault. The bottom right screen capture shows the terminal names and message types for all the terminals, including the four input terminals. The input terminals which are wired from the fail and timeout terminals are named in, in1 and in3. The input terminal that is wired from the modeled fault of the service invoke is named InventoryServiceFault.

Return modeled fault for flow errors

```

----- Create FlowFailed object

// Create the FlowFailed object that gets returned for the fault
ServiceManager serviceManager = new ServiceManager();
BOFactory bof = (BOFactory) serviceManager.locateService("com/ibm/websphere/bo/BOFactory");
DataObject flowFailed = bof.create("http://StoreLib",
    "FlowFailed");

----- Initialize the FlowFailed with reason

// Initialize the FlowFailed object with the reason for the failure
// The input terminal InventoryServiceFault is for a modeled fault, all others are unmodeled
if (inputTerminal.getName().equals("InventoryServiceFault")) {
    // Get the failure information from the body
    flowFailed.setString("reason", smo.getString("/body/InventoryFault_element/description"));
    flowFailed.setString("primitive", "ServiceInvoke=CheckInventory");
} else {
    // Get the failure information from failInfo in the context
    flowFailed.setString("reason", smo.getString("context/failInfo/failureString"));
    flowFailed.setString("primitive", smo.getString("context/failInfo/origin"));
}

----- Create body wrapper for FlowFailed object

// Create the wrapper for the SMO body and add the FlowFailed object to it
ServiceMessageObjectFactory smoFactory = ServiceMessageObjectFactory.eINSTANCE;
DataObject faultBody = smoFactory.createServiceMessageObjectBody(
    "http://StoreLib/Ordering",
    "sendOrder_flowFailedMsg");
faultBody.setDataObject("flowFailed", flowFailed);

----- Update SMO body and fire output terminal

// Update the SMO with the new body and fire the out terminal
smo.setBody(faultBody);
out.fire(smo);

```

23

Custom mediation primitive

© 2008 IBM Corporation

This slide provides a complete implementation of the custom mediation primitive functionality described on the previous slide.

The fault that is to be returned contains a FlowFailed business object, so the first thing that is done is to create this object. This is done the way any business object is created in Java code, by using the BOFactory.

The next thing to do is to initialize the FlowFailed object so that the reason and primitive information is populated. At this point there is need to differentiate between an input message resulting from the modeled fault versus the other sources of input messages, because the location of the data in the SMO is different. The code checks to see if the input terminal was named InventoryServiceFault, in which case it obtains the reason from the body of the fault message. The name of the primitive does not exist in the SMO, so the primitive name is set from a hard coded value. If the input message came from any of the other input terminals, the reason and primitive values are obtained from the failInfo section of the SMO context.

In order to place the FlowFailed business object into the SMO, it needs to be contained in a body wrapper for the sendOrder operation's flowFailed message. The body wrapper is created using the ServiceMessageObjectFactory and the FlowFailed object is placed into it.

Finally, the body wrapper for the fault is used to replace the existing body of the SMO, and the result is fired on the output terminal.

Summary

- Examined the custom mediation primitive details



Custom mediation

- ▶ Overview of function
- ▶ Use of terminals
- ▶ Definition of properties
- ▶ Basic code samples
- ▶ Changes since and migration from V6.0.2
- ▶ Error handling
- ▶ Examine custom code sample

24

Custom mediation primitive

© 2008 IBM Corporation

In summary, this presentation provided details regarding the custom mediation primitive, along with an overview of its function and information about the primitive's use of terminals and its properties. Some small coding samples were provided that are useful for helping you understand the basics about developing code for a custom mediation primitive. Because this primitive has changed considerably between version 6.0.2 and version 6.1, a review of those changes was provided. Also provided were details about how to migrate custom mediation primitives developed in version 6.0.2 to implementations that are compatible with version 6.1. Some error conditions you might encounter were described. Finally, a realistic example examining some of the new capabilities of custom mediation primitives was presented, including the complete code used to implement the example.

Feedback

Your feedback is valuable

You can help improve the quality of IBM Education Assistant content to better meet your needs by providing feedback.

- Did you find this module useful?
- Did it help you solve a problem or answer a question?
- Do you have suggestions for improvements?

Click to send e-mail feedback:

mailto:iea@us.ibm.com?subject=Feedback_about_WBIV61_CustomPrimitive.ppt

This module is also available in PDF format at: [../WBIV61_CustomPrimitive.pdf](..WBIV61_CustomPrimitive.pdf)



You can help improve the quality of IBM Education Assistant content by providing feedback.

