eInsight Correlations in CAPS 5.1
Correlation Examples, Example 2

ELS Static Counted Implementation

Michael.Czapski@sun.com
November 2007

Introduction

Correlations are probably the single least understood area of eInsight functionality. The example discussed here implements one of the “Event Linking and Sequencing” patterns, present in e*Gate 4.5 and eGate SRE, that is alleged to have been lost in ICAN and Java CAPS. In as much as implementing ELS in eInsight 5.1 using correlation requires some development, rather then just configuration, one could argue that it was lost. In as much as implementing ELS in eInsight 5.1 is possible and relatively simple, one could also argue the opposite.

This example implements a part of the ELS functionality dealing with linking a number of related messages, a counted correlation pattern, or an aggregator pattern.

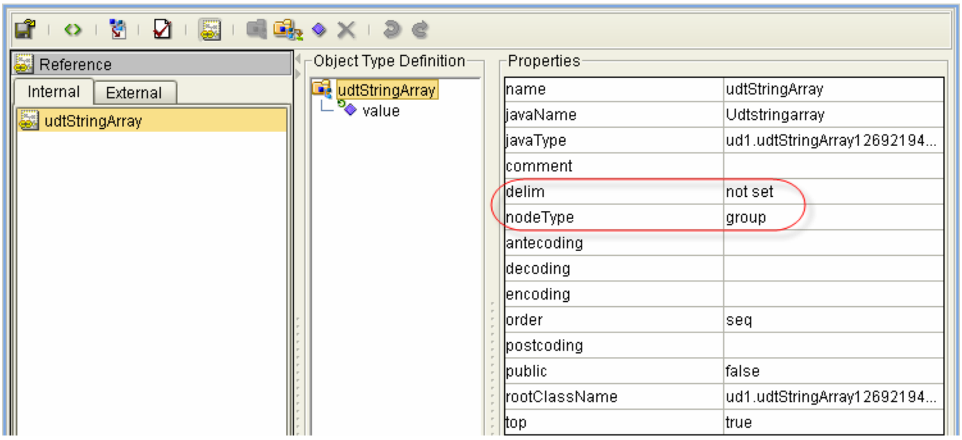
Unlike the simple implementation from Example 1, this implementation will correlate a varying number of messages, statically set at design time. Thus the same implementation can be used to correlate 2, 3, 10 or 30 messages, by modifying the value of a single business process attribute. By obtaining the value of the business process attribute, which controls the message count, from the environment or the initial message, one will change the static implementation into a dynamic counted correlation solution.

Key Points

Rather then developing the example step-by-step, as was done in example 1, only the key points will be illustrated and discussed. The entire solution is available as a Java CAPS project export and can be inspected.

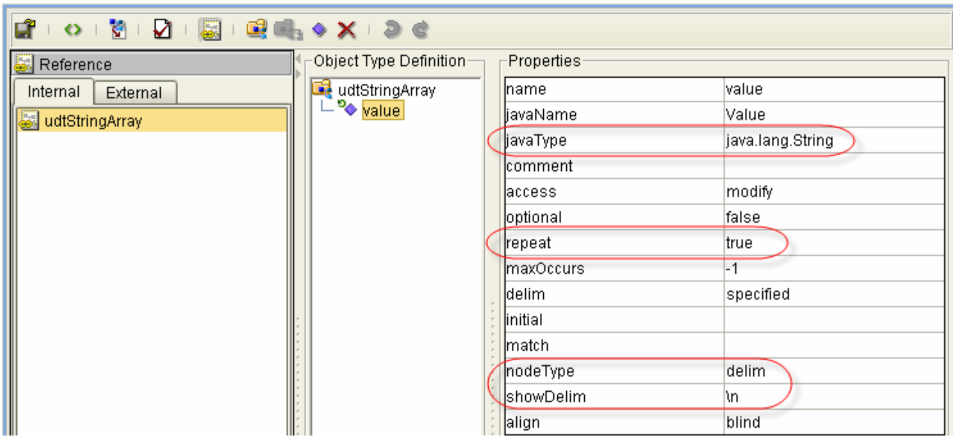
Message Container Array

In example 1 there were two JMS receive activities and only two messages were correlated. Each receive activity has an associated message container so both messages were available for aggregation once both were received. This example will also have two JMS Receive activities but it will be built to collect more then two messages. In order to collect a number of messages that is greater then the number of JMS Receive activities, hence greater then the number of containers available by default, we must implement an expandable array of containers to hold the messages as they are received. This we will do by creating a user-defined, delimited Object Type Definition (OTD) in which a single repeating string node will serve as a container for any number of messages, see Figures 1, 2 and 3. This is a serviceable simplification. More sophisticated solution is left to the reader.



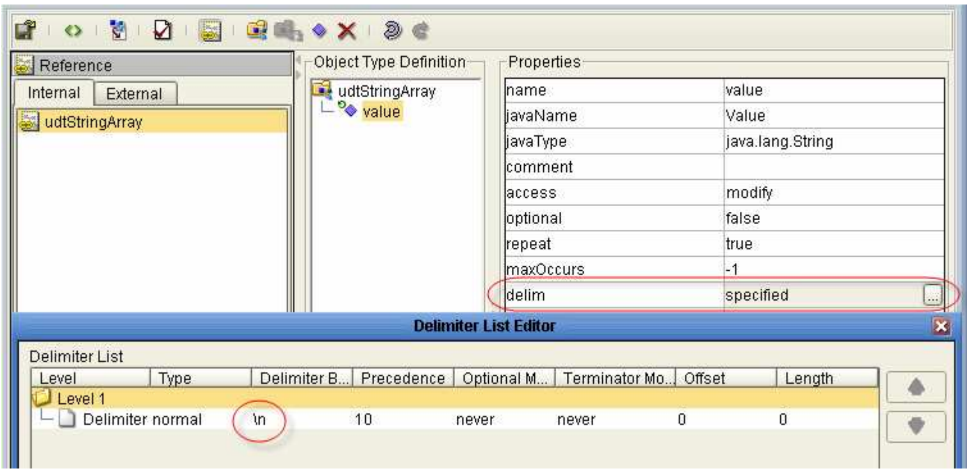
name	udtStringArray
javaName	Udtstringarray
javaType	ud1.udtStringArray12692194...
comment	
delim	not set
nodeType	group
antecoding	
decoding	
encoding	
order	seq
postcoding	
public	false
rootClassName	ud1.udtStringArray12692194...
top	true

Figure 1 User-defined OTD, root node properties



name	value
javaName	Value
javaType	java.lang.String
comment	
access	modify
optional	false
repeat	true
maxOccurs	-1
delim	specified
initial	
match	
nodeType	delim
showDelim	\n
align	blind

Figure 2 User-defined OTD, repeating element properties



name	value
javaName	Value
javaType	java.lang.String
comment	
access	modify
optional	false
repeat	true
maxOccurs	-1
delim	specified

Level	Type	Delimiter B...	Precedence	Optional M...	Terminator Mo...	Offset	Length
Level 1							
Delimiter normal		\n	10	never	never	0	0

Figure 3 User-defined OTD, delimiter definition

Business Process

The business process contains a loop, controlled by a couple of *int* business process attributes, in which all messages but the first one are received and added to the message buffer, Figure 4.

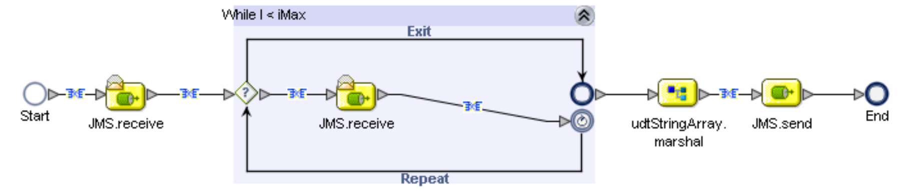
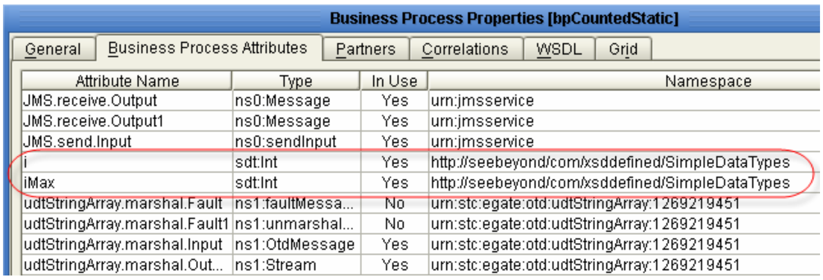


Figure 4 Business Process

Note that the User-defined OTD's marshal service is dragged onto the canvas and placed after the loop. This has two results. First, the buffer for messages becomes available to the process. Second, once the buffer is populated with messages we can marshal it into a JMS Text Message and send it on its way without any further ado.

Note that the correlation key and the correlation set are created identically to the way they were created in Example 1, and the JMS Receive activities are configured to use correlations in exactly the same way as in Example 1. This is not discussed in this document but is critical to the operation of the example.

The loop control attribute, *i*, is set to 1 before entering the loop and incremented by 1 each time through. The loop termination attribute, *iMax*, is initialised to the number of messages to collect before the loop is entered, in this case 3. See Figures 5 and 6.


Business Process Properties [bpCountedStatic]
General
Business Process Attributes
Partners
Correlations
WSDL
Grid

Attribute Name	Type	In Use	Namespace
JMS.receive.Output	ns0:Message	Yes	urn:jmsservice
JMS.receive.Output1	ns0:Message	Yes	urn:jmsservice
JMS.send.Input	ns0:sendInput	Yes	urn:jmsservice
i	sdt:Int	Yes	http://seebeyond/com/xsddefined/SimpleDataTypes
iMax	sdt:Int	Yes	http://seebeyond/com/xsddefined/SimpleDataTypes
udtStringArray.marshal.Fault	ns1:faultMessa...	No	urn:stc:egate:otd:udtStringArray:1269219451
udtStringArray.marshal.Fault1	ns1:unmarshal...	No	urn:stc:egate:otd:udtStringArray:1269219451
udtStringArray.marshal.Input	ns1:OtdMessage	Yes	urn:stc:egate:otd:udtStringArray:1269219451
udtStringArray.marshal.Out...	ns1:Stream	Yes	urn:stc:egate:otd:udtStringArray:1269219451

Figure 5 Loop control attributes

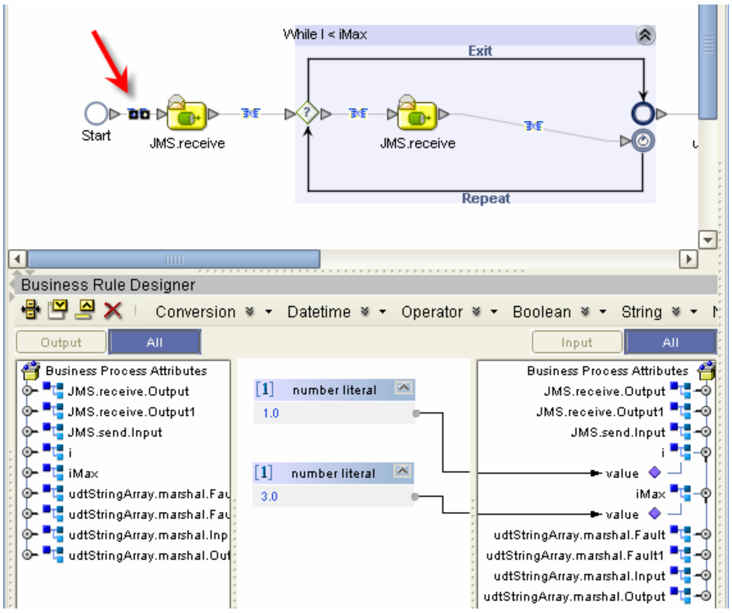


Figure 6 Initialisation of loop control variables

The Text Message from the first JMS Receive activity is assigned to iteration 1 of the OTD repeating element, see Figure 7. eInsight repeating structures are 1-based, not 0-based as would be the case in Java.

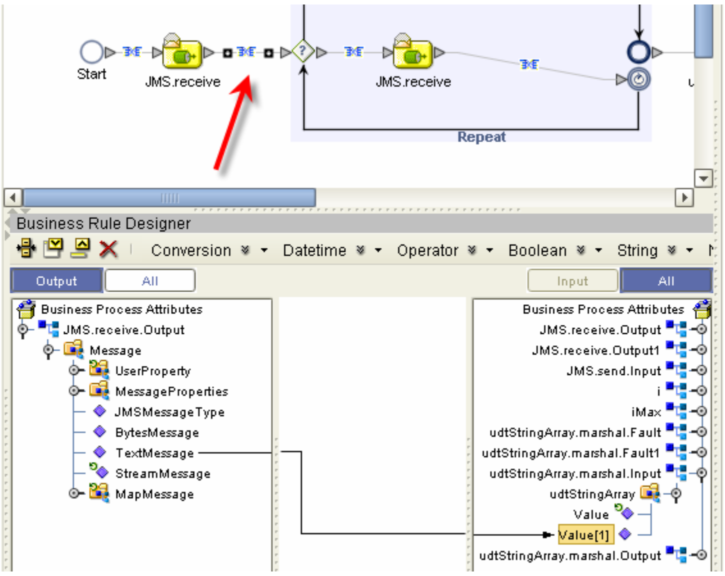


Figure 7 Add the first message to the buffer

The loop conditional is simple, *i* < *iMax*, see Figure 8.

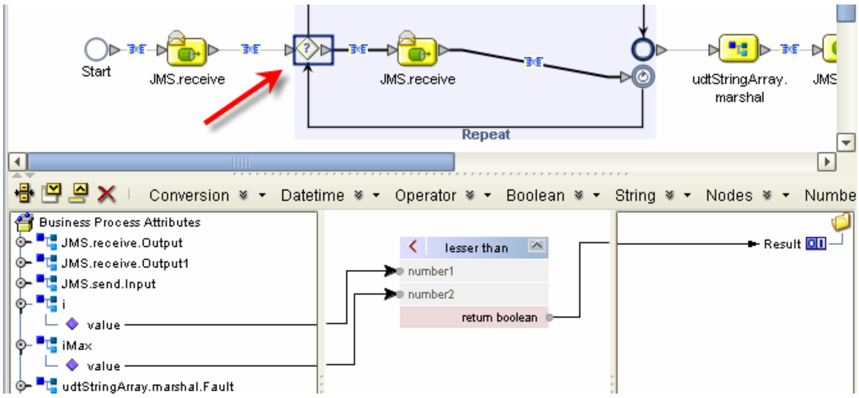


Figure 8 Loop conditional

By incrementing *i*, the loop control attribute, before the next JMS Receive we have its value ready to use for adding the next message to the buffer, see Figure 9.

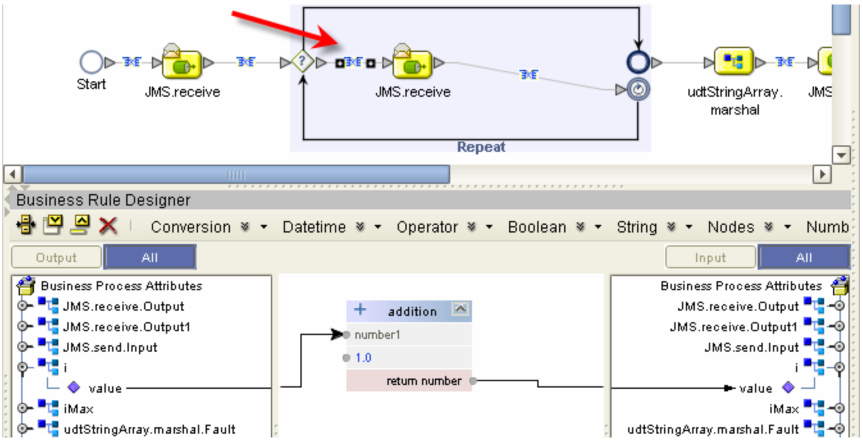


Figure 9 Incrementing loop counter

The Text Message from the JMS Receive activity inside the loop is assigned to the 'current' element of the buffer. First time through the loop it will be 2. The predicate looks a bit complex but it is really quite simple. We merely assign *i*, the loop counter, to the result of 'New Predicate', see Figure 10.

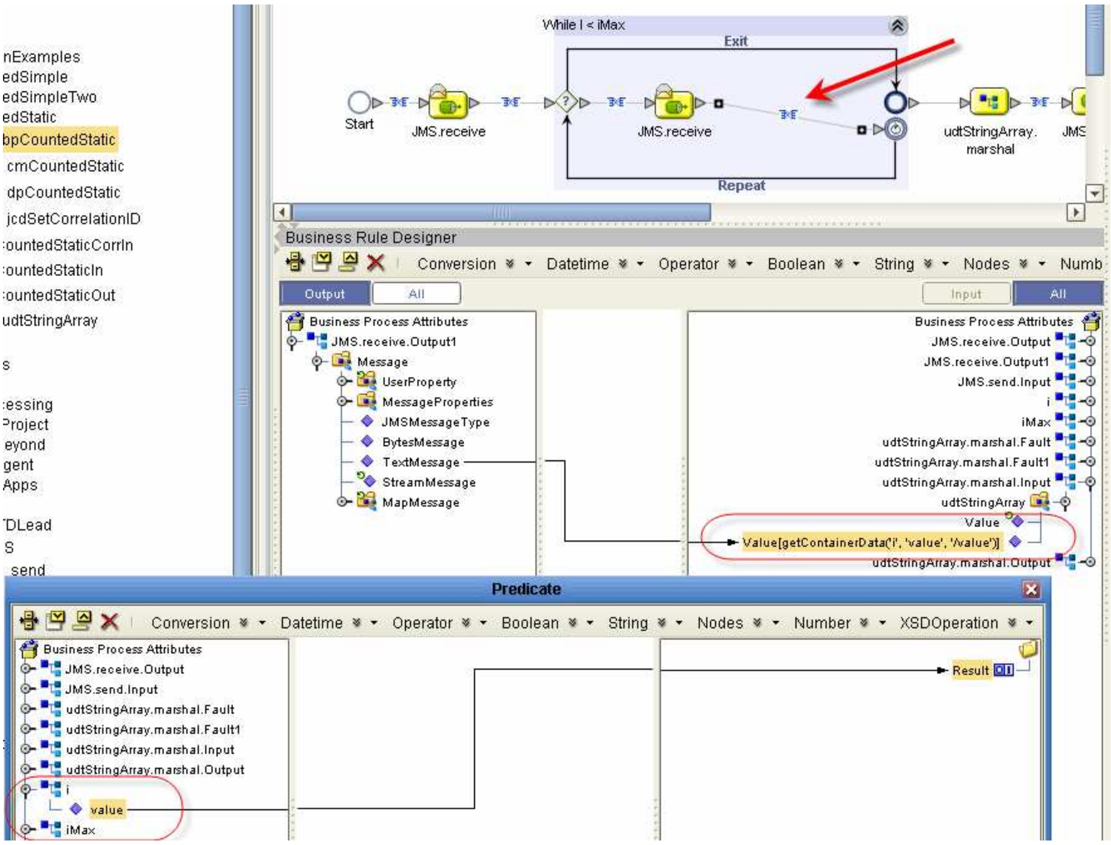


Figure 10 Populate "*i*'th" buffer.

Finally, we copy the output of the marshal service of the User-define OTD into the JMS Text Message and send it on its way.

Create a connectivity map, which will look exactly the same as in Example 1, create a deployment profile, build and deploy.

Note that the solution uses the very same Java Collaboration as that used in Exercise 1 to populate the JMS Header Property Correlation Id, see Figure 11.

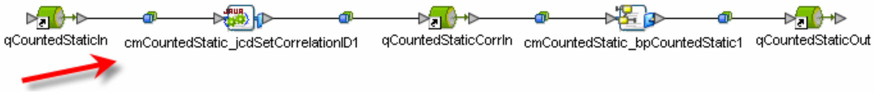

qCountedStaticIn
cmCountedStatic_jcdSetCorrelationID1
qCountedStaticCorrIn
cmCountedStatic_bpCountedStatic1
qCountedStaticOut

Figure 11 Connectivity Map

Exercise the solution

Exercise the solution by submitting sample messages to the initial queue as follows: aa, bb, cc, bb, aa, cc, cc, bb, aa.

In the final queue you should see three messages, one containing three lines of cc, one containing three lines of bb and one containing three lines of aa.

Summary

This solution is an example that contains all essential elements that any static counted correlation will have to have. By setting the value of the *iMax* variable one can collect any number of messages with the same correlation id. By modifying the way *iMax* is set, for example from an external property or from a property in the initial message, one can develop a dynamic counted solution.