BPEL Basics for Java Developers

Every day we hear about another IT group embarking on the long march toward the Service Oriented Architecture (SOA) utopia. The journey treks through a mind-numbing array of standards and technologies, including Business Process Execution Language (BPEL). These technologies are touted as indispensable, yet they seem remote, unfamiliar, and confusing. We look at our current applications and wonder, how do I get there from here?

The goal of this article is to act as a footbridge into the worlds of SOA and BPEL. It's not intended to reveal the entire BPEL language, but it should get you moving in the right direction from your starting point as a Java Developer.

Mapping the Unknown

As with any journey, it's probably best to start with a map of the terrain.

Starting the Journey

Most of us Java developers, faced with the complexities of SOA, are somewhere in stages one through four of the SOA and BPEL adoption continuum:

1. *Denial* - don't need it
2. *Coercion* - management says we have to use it
3. *Elation* - realization that it will solve all our enterprise application problems
4. *Depression* - realization that it will *not* solve all our enterprise application problems... yet

With a little luck, this article will help get you closer to stage five:

5. *Enlightenment* - understanding how - and when - to leverage the advantages of SOA, via BPEL (in this case, using your Java skills as a guide)

If you're familiar with an object-oriented language other than Java, you can read between the lines here to benefit as well.

Parsing the Language of SOA

One reason SOA seems unreachable is because of its unfamiliar terminology. The SOA universe is a jungle of terms that you need to hack your way through, but don't let the terminology throw you. Words like *service orchestration*, *portType*, and *partnerLinkType* can be parsed to make sense in your Java world.

We'll start with some basic SOA concepts, and you'll see that your journey has already begun.

Service Orchestration

The first thing to look at is the notion of *Service Orchestration* and how it relates to real, actual, you know – *code.*

Service Orchestration is nothing more than a fancy title for a program that calls web services during its execution. There's nothing magical or omnipotent about a "service orchestration". In fact we'll learn later that any service orchestration implemented using BPEL is *also* just a standard, WSDL-defined web service in its own right.

In the context of Java, a service orchestration might be a Java class that not only constructs and calls methods on other classes but which *also* invokes web services using special Java classes and interfaces designed specifically for that purpose. These specialized classes understand the way to send web service request messages. The

services are called – or *invoked* – based on the class' programmed logic. And the web service request messages – the ones that are sent when invoking those services – use data that's defined in the Java class implementing the orchestration.

From a Java design standpoint, a well-designed, service-oriented architecture is not unlike a well-designed set of Java classes, at least from a functionality and data *encapsulation* standpoint. Each web service in the architecture takes on the same sort of black box visibility we see in Java designs whose implementations and data are hidden from other client classes.

In fact, note that a service orchestration can be implemented in just about any language you choose – provided that the language supports mechanisms for the exchange of SOAP Messages (at a minimum).

The SOA Language Jungle

There are a few elements of SOA – specifically in the areas of WSDL and BPEL – that seem especially confusing. The first is the daunting collection of three- and four-letter acronyms that identify the technologies and standards associated with SOA. Just learning the differences between XML, XHTML, XSD, XSLT, XPath, and XQuery would be quite enough, but of course there's more.

In the context of WSDL – the XML-based descriptions that identify the definition of a web service that a client needs to know – there are a few terms in particular that can be distracting. And because BPEL depends directly on the WSDL standard (and also extends it), we need to get a handle on these terms. Their specific definitions are available in their standards documents, so here we'll just take a quick look at things that often cause confusion.

What's in a *portType?* Why aren't we *interfacing* instead?

For some reason when the WSDL standard was being created, the authors chose some unfamiliar terminology for some web service definition elements. One of these is the ***portType***. As Java developers, we might have wished for the term *interface*. If you're comfortable with the Java definition for *interface*, then you can skip a lot of frustration by just thinking *interface* whenever you see the term ***portType*** in a WSDL document, because that's what it is. (And as it turns out, one of the proposed changes for the WSDL 2.0 standard is to make this very change.) Where a Java *interface* specifies the methods an implementing class must provide, a ***portType*** specifies the ***operations*** a web service interaction partner must support.

Here's an example of a ***portType*** definition that has one ***operation*** – '***request***':

```
<wsdl:portType name="loanServicePT">
       <wsdl:operation name="request">
              <wsdl:input message="tns:creditInformationMessage" />
              <wsdl:output message="tns:approvalMessage" />
              <wsdl:fault name="unableToHandleRequest"
```

```
                            message="tns:errorMessage" />
                </wsdl:operation>
        </wsdl:portType>
```

Must my *binding* have *use* AND *style?*

There are some exceptions, but for the most part BPEL processes deal almost exclusively with what we refer to as the *abstract* portion of a WSDL document, that is, the stuff that defines the ***types***, ***messages***, ***operations*** and (ugh) ***portTypes***. So you'll only really need to be comfortable with the WSDL element ***binding*** and the attributes ***use*** and ***style*** when you get to the deployment phase of your implementation. Deployment of your BPEL process is associated with the *concrete* portion of a WSDL definition, specifically, "how and where do I physically connect to a service and what does my request message need to look like?" The ***binding*** and ***service*** elements and their subordinates are used to define the "how" and "where" of connecting to a service, and the "what" to send. Below is an example of related ***service*** and ***binding*** definitions:

```
<wsdl:binding name="SOAPBinding1" type="tns:riskAssessmentPT">
  <soap:binding style="rpc" transport="http://schemas.xmlsoap.org/soap/http" />
  <wsdl:operation name="check">
    <soap:operation soapAction="" style="rpc" />
      <wsdl:input>
        <soap:body use="encoded" namespace="urn:loanassessor"
                   encodingStyle="http://schemas.xmlsoap.org/soap/encoding/" />
      </wsdl:input>
      <wsdl:output>
        <soap:body use="encoded" namespace="urn:loanassessor"
                   encodingStyle="http://schemas.xmlsoap.org/soap/encoding/" />
      </wsdl:output>
    </wsdl:operation>
</wsdl:binding>

<wsdl:service name="LoanAssessor">
  <wsdl:documentation>Loan Assessor Service</wsdl:documentation>
    <wsdl:port name="SOAPPort1" binding="tns:SOAPBinding1">
      <soap:address
        location="http://localhost:8080/active-bpel/services/AssessorWebService" />
    </wsdl:port>
</wsdl:service>
```

Be sure to note that the ***binding*** element is found in the namespaces referenced by both the ***soap:*** and ***wsdl:*** prefixes. See the individual standards for these for information on the differences and dependencies between the two.

Do partners really *link*?

The BPEL standard defines several extensions to WSDL. One of these is an element named ***partnerLinkType***. This one takes a bit of cogitating to understand, but the trick for a Java developer is in noticing the '*Type'* portion of the term.

Just to skip ahead for a moment, in BPEL we'll be defining ***partnerLinks***. These ***partnerLinks*** identify how the process communicates with the web services it's

orchestrating (as well as the clients that invoke it, since every deployed BPEL process is a WSDL-defined web service in its own right).

As you might surmise, a ***partnerLink*** in a BPEL process is an instance of a ***partnerLinkType***. The obscurely-named ***partnerLinkType*** is just a kind of abstract class definition that identifies the *interface(s)* (i.e., ***portTypes***) which must be *implemented* so that two web service *partners* may communicate with each other using web service technology. Here's an example of a ***partnerLinkType*** definition in action:

```
<plnk:partnerLinkType name="riskAssessmentLT">
       <plnk:role name="assessor" portType="tns:riskAssessmentPT" />
</plnk:partnerLinkType>
```

Keep cutting through the jungle; we're getting there.

Why do I need WSDL when I *already* have an *RPC*?

Before web service / SOA technology got off the ground, one of the mechanisms used by a program running on a computer in Dallas to call another program running on a system in New York was something called a *remote procedure call*. As we know, this form of remote communication was available prior to the WSDL standard.

So a Java programmer who has used RPC might naturally ask: why bother with WSDL if there's already a remote calling mechanism available?

The answer is *standardization*.

WSDL is completely platform, language and operating system agnostic. It provides a framework to specify not only a remote procedure call, but an entire communications interface, including data type and message definitions, service endpoints, faults and different data transmission protocols (i.e., ***bindings***). The message exchange details specified in a WSDL definition can be implemented, again, using any language and on any platform / operating system with facilities to support SOAP messaging (at least).

Orchestration Using Java

We've seen that the language of SOA can be understood in Java terms. The next step is to find out why it might be advantageous to create service orchestrations using SOA techniques rather than Java. To prove the point, we can start by understanding how orchestration might be done using Java.

The components needed for orchestration, namely executable code, message exchange, and business logic, are inherent characteristics of Java. Java is, of course, executable, and

it's a given that we can create a Java class that executes the logic for a service orchestration.

In terms of message exchange, the WSDL messages exchanged by web services are defined using XML, and as luck would have it, XML is supported extremely well by Java. The Java API for XML-based Remote Procedure Calls (JAX-RPC) gives us almost everything we need for web service message exchanges using SOAP. So Java can certainly handle the *lingua franca* of web service orchestration.

In terms of business logic, that is, the logic representing a business process – the "BP" in BPEL, there are certainly logical constructs that a Java class can implement.

With the understanding of the components needed for an orchestration, we can look at how an orchestration might be done in Java. Doing so lets us understand the difference between where we are now (Java) and why we might take the road to enlightenment.

Loan Approval Process: a Service Orchestration in… Java?

Going through the exercise of creating a fancy *service orchestration* in Java takes about 15 minutes. All we need to do is create a web service client with the right sort of logic.

WSDL2Java

For this exercise we'll use the ActiveBPEL Designer Loan Approval BPEL Process. This process is described in the WS-BPEL 2.0 Standard document available from OASIS. When you install the free ActiveBPEL Designer package, *you can find the WSDL that is the basis for the BPEL process,* ***loanServicePT.wsdl*** in the Samples folder.

We'll be using this WSDL file to generate the source files for a *Java* web service client / service orchestration. Note that our Java implementation will be invoking the *same* deployed services we use to demonstrate the Loan Approval process when it's implemented using BPEL.

One Java-based package that is available for web service development is *Axis*, an open source implementation of the SOAP standard from Apache (http://ws.apache.org/axis/). Included in the Axis package is a handy widget called *WSDL2Java*. We can use it to create the Java artifacts we need to construct a service orchestration.

What WSDL2Java creates for us, ultimately, is the special set of Java classes that know how to use JAX-RPC to exchange WSDL messages using SOAP. When you take a look at these generated files later, you'll see why you don't want to have to code them by hand. Once you have these classes generated, you still have some work to do. You'll need to create a *ServiceOrchestration* Java class that not only invokes these newly-generated special classes, but contains the logic for the orchestration as well.

The finished collection of 11 files (see the links at the end of this article), is a Java-based implementation of the Loan Approval Process that is described in the WS-BPEL 2.0

Standard document. In all, it is about 850 lines of code, most of which were generated by WSDL2Java.

Basically, the business logic for this process accepts an incoming request for a loan, examines the amount and the name, and follows a different set of approval steps depending on the name and the amount requested. Here's the logic that makes this Java class a *business process*:

```
/**
 * Execute the business logic of this service orchestration.
 *
 * @param first
 * @param last
 * @param loanAmount
 * @param appLimit
 */
public String executeProcess( String first, String last,
                              BigInteger loanAmount, BigInteger appLimit )
   throws RemoteException
{
   String response ;
   if ( loanAmount.compareTo( appLimit ) >= 0 )
   {
      // Amount is at or above the approval limit - send directly to the approver.
      //
      response = mApprover.approve( first, last, loanAmount );
   }
   else
   {
      // Below the approval limit - have the Assessor check the applicant's risk.
      //
      response = mAssessor.check( first, last, loanAmount );

      // Check risk level - submit request to the Approver if not 'low'.
      //
      if ( response.equalsIgnoreCase( "low" ))
      {
         response = "yes";
      }
      else
      {
         response = mApprover.approve( first, last, loanAmount );
      }
   }

   return response ;
}
```

If you deploy the sample web services that are within the Samples folder of the Loan Approval sample, you will see that the runtime behavior for them is the same as the runtime behavior for the *ServiceOrchestration* Java class.

Take some time to examine the rest of the *ServiceOrchestration* class, as well as the generated classes – *Loan*.java*, *Risk*.java* and *SOAP*.java*. Note that these generated classes are not the *implementations* of the web services that the orchestration uses to perform its business logic. These generated classes are the "glue" that allows our *ServiceOrchestration* class to *invoke* those web services.

If you're curious, why not try this implementation now? When you're done, we'll continue on to the next big clearing – where we'll discover why we probably don't want to use this implementation.

On to BPEL Basics

What we're really trying to do here is learn more about BPEL basics with our Java knowledge as a guide. So let's hack our way into that clearing over there, sit down and do some comparisons between our Java service orchestration and the sample BPEL implementation – Loan Approval – found in the WS-BPEL 2.0 Standard document.

Getting the Message – the Receive Activity and `main()`

WSDL-defined web services respond to WSDL messages. That is their purpose. They aren't typically executed any other way. This goes double for a BPEL process, which can *only* be started when it receives an incoming WSDL message.

In our Java implementation, the incoming "message" is actually comprised of the command line arguments we pass to `main()`. That's what kicks off an instance of our *ServiceOrchestration* class. The `main()` method then takes these arguments (data), constructs some objects, and then places the incoming data into those objects so they can be used by other parts of our class. Then the `executeProcess()` method of the class is called to execute the business logic defined there. Here's the `main()` for *ServiceOrchestration*:

```
/**
 * Construct a service orchestration object that calls the Approver and
 *   Assessor services as needed to approve or reject a loan.
 *
 * @param args  Command-line application arguments.
 */
public static void main( String[] args )
{
   if ( args.length < 3 )
   {
      usage();
   }
   else
   {
      try
      {
         BigInteger amount = BigInteger.valueOf( Long.parseLong( args[2] ));
         BigInteger appLimit = BigInteger.valueOf( 10000L );
         String firstName = args[0],
                lastName  = args[1];
         String response ;

         try
         {
            RiskAssessmentPTProxy assessor = new RiskAssessmentPTProxy();
            LoanApprovalPTProxy approver = new LoanApprovalPTProxy();
            ServiceOrchestration so = new ServiceOrchestration( assessor, approver );
            response = so.executeProcess(firstName, lastName, amount, appLimit);
            System.out.println( "Loan approval result: " + response );
```

```
            }
            catch ( Exception e )
            {
               reportError( e );
            }
         }
         catch( NumberFormatException nfe )
         {
            usage();
         }
      }
   }
```

BPEL provides us with a close analog for `main()` – the *Receive* activity. A Receive activity is typically the first construct that appears in a BPEL process; it accepts the data from the incoming WSDL message and places it into a variable that is accessible to the rest of the process. Here's the XML that forms the Receive activity in the Loan Approval sample process:

```
<bpel:receive createInstance="yes" name="ReceiveCustomerRequestforLoanAmt"
              operation="request" partnerLink="customer"
              portType="lns:loanServicePT" variable="request">
   <bpel:sources>
      <bpel:source linkName="receive-to-assess">
         <bpel:transitionCondition>$request.amount &lt; 10000
         </bpel:transitionCondition>
      </bpel:source>
      <bpel:source linkName="receive-to-approve">
         <bpel:transitionCondition>$request.amount &gt;= 10000
         </bpel:transitionCondition>
      </bpel:source>
   </bpel:sources>
</bpel:receive>
```

Notice that the ***operation*** attribute of the ***receive*** element is analogous to the `main()` function in our Java class. Here, the operation name is ***request***, but we could just as easily have called it ***main***.

The ***variable*** attribute of the ***receive*** element specifies where the incoming data is to be placed for use by the remainder of the process. Our Java class splits the incoming data into simple variables – `amount`, `firstName` and `lastName`. Here we instead simply copy the data into a *complex* variable, ***request***, which can hold all three values.

Just like our Java implementation, the BPEL process could be designed to receive *additional* incoming messages while it is executing. The Receive activity can be used to do this if we know exactly where in our logic sequence those messages will arrive.

Calling All Web Services – the Invoke Activity and `invoke()`

Once a service orchestration has begun executing, its primary goal is to interact with one or more WSDL-defined web services in such a way as to fulfill the request it has

received. In order to do this, the orchestration needs some generic way to *invoke* a web service – that is, to place data values into a WSDL message and then send the message to the web service.

In Java, this message transmission can be performed using any variant of the remote procedure call mechanism, and in the WSDL-defined world, we use the JAX-RPC library to do this. The function provided in this case is the `invoke()` method.

Let's look at *ServiceOrchestration.executeProcess()*, where several lines down we see the following method call:

```
response = mApprover.approve( first, last, loanAmount );
```

This is the point where we are invoking a web service to approve a loan using the data values provided as method arguments. If we trace this call through the various classes generated by WSDL2Java, we eventually arrive at *SOAPBindingStub.approve()*:

```
public java.lang.String approve(java.lang.String firstName, java.lang.String name,
      java.math.BigInteger amount) throws java.rmi.RemoteException
{
   if ( super.cachedEndpoint == null )
   {
      throw new org.apache.axis.NoEndPointException();
   }
   org.apache.axis.client.Call _call = createCall();
   _call.setOperation(_operations[0]);
   _call.setUseSOAPAction(true);
   _call.setSOAPActionURI("");
   _call.setSOAPVersion(org.apache.axis.soap.SOAPConstants.SOAP11_CONSTANTS);
   _call.setOperationName(new
       javax.xml.namespace.QName("urn:loanapprover", "approve"));

   setRequestHeaders(_call);
   setAttachments(_call);
   try
   {
      java.lang.Object _resp =
            _call.invoke(new java.lang.Object[] { firstName, name, amount });

      if ( _resp instanceof java.rmi.RemoteException )
      {
         throw (java.rmi.RemoteException)_resp;
      }
      else
      {
         extractAttachments(_call);
         try
         {
            return (java.lang.String)_resp;
         }
         catch (java.lang.Exception _exception)
         {
            return (java.lang.String)org.apache.axis.utils.JavaUtils
                  .convert(_resp, java.lang.String.class);
         }
      }
   }
   catch (org.apache.axis.AxisFault axisFaultException)
   {
```

```
        if ( axisFaultException.detail != null )
        {
           if ( axisFaultException.detail instanceof java.rmi.RemoteException )
           {
              throw (java.rmi.RemoteException)axisFaultException.detail;
           }
        }
        throw axisFaultException;
     }
  }
```

Examining the contents of this method, we find that it constructs an object of type *org.apache.axis.client.Call*, which is one of the specialized classes that "know" how to send WSDL messages using the SOAP protocol. That knowledge is implemented in the *Call* class by the `invoke()` method, which is responsible for sending a WSDL message to the web service and, in this case, accepting a response – `_resp`.

If you examine the rest of this Java file, you'll see that it has been hard-coded (by WSDL2Java), along with the sequence of method calls that brought us to this `approve()` method, to do one thing: to invoke the ***approve*** operation on the ***ApproverWebService*** service, which is located at the endpoint reference designated by ***http://localhost:8080/activebpel/services/ApproverWebService***.

If that seems like a lot of code to do one specific thing, just wait...

In BPEL, this invocation functionality is facilitated by the Invoke activity. Here's the XML that basically does everything we were just looking at, and more:

```
<bpel:invoke inputVariable="request" name="InvokeLoanApprover"
             operation="approve" outputVariable="approval" partnerLink="approver"
             portType="lns:loanApprovalPT">
   <bpel:targets>
      <bpel:target linkName="receive-to-approve"/>
      <bpel:target linkName="assess-to-approve"/>
   </bpel:targets>
   <bpel:sources>
      <bpel:source linkName="L2"/>
   </bpel:sources>
</bpel:invoke>
```

This might be the first place Java developers will notice how BPEL is specifically *tuned* for service orchestration.

Here, the data for the outgoing WSDL message is specified by the ***inputVariable*** attribute. Think of this as the "input" to the web service being invoked. In this case the data will be found in the ***request*** variable, which is a complex value similar to the ***request*** variable discussed earlier (in fact, it's the same variable). The operation to invoke is identified by the ***operation*** attribute, in this case, ***approve***. Finally, we see that the response data – the *output* of the web service invocation - should be placed into the variable designated by the ***outputVariable*** attribute, ***approval***.

The ***partnerLink*** attribute provides one of the huge advantages of using BPEL. This attribute's value – ***approver*** – specifies a reference to information that is stored by the BPEL server when the process is deployed. Unlike our Java implementation, where all of our endpoint reference information is hard-coded, in BPEL the concrete endpoint reference to be used by the process is specified during the *deployment* phase. This has advantages in that the process itself doesn't have to be modified if, for instance, the service endpoint reference is moved to a different server in another network or another country. In such a case our "hard-coded" Java implementation would either have to be completely regenerated using WSDL2Java, or we'd have to modify those generated files to parameterize this value. And modifying generated files is always a dicey proposition.

I'll get back to you – the Reply Activity and `return()`

When a service orchestration completes, it will typically – though not always – respond with some piece of data, the result of responding to the incoming request. In this phase, a WSDL message is populated with data, and the message is then sent as a response.

In our Java implementation, we run our *ServiceOrchestration* from the command line, so we simulate this response by simply outputting the loan application result to the console. If we were to design a service orchestration that was *also* a web service, we would provide the response to the loan application using `return()`, just as the *SOAPBindingStub.approve()* method provides the result of the web service invocation using `return()`. The data in this return value would be copied to a WSDL message and sent as the response.

In BPEL, there is a specific activity used to generate a response: the *Reply* activity. A Reply is responsible for extracting data from a process variable and placing it into a WSDL message that is then sent back in response to the one accepted by the preceding Receive. In fact syntactically, a Reply activity requires that the process have a Receive (or equivalent) somewhere earlier in the flow of the process.

Here's the XML syntax our BPEL Process uses to reply to the loan approval request:

```
<bpel:reply name="AcceptMessageToCustomer" operation="request"
            partnerLink="customer" portType="lns:loanServicePT"
            variable="approval">
   <bpel:targets>
      <bpel:target linkName="L2"/>
      <bpel:target linkName="L1"/>
   </bpel:targets>
</bpel:reply>
```

The ***variable*** attribute – here set to the value ***approval*** – designates the process variable whose data will be used for the outgoing WSDL message. The rest of the Reply's attributes are used to match the activity up with a preceding Receive (or equivalent). Seeing this, you might guess that there can be multiple Receive/Reply pairs active at one time in a running business process. If so, you'd be right.

Exceptional Fault Tolerance – BPEL Faults and Java Exceptions

Both Java and BPEL have a standardized means of dealing with error conditions. In Java we have exceptions. In BPEL, the equivalent is called a fault. BPEL faults work surprisingly similarly to Java exceptions.

In Java, we know that once an exception is thrown, if it is not handled by a `try/catch` block, it is passed up the stack's call chain or into the enclosing `try/catch` block for handling there, until – if it's never handled at all – it eventually results in the Java program's abnormal termination. The exception is then typically reported via a stack dump output to the console or a log.

BPEL faults, and BPEL's fault handling, are very similar to this. Like Java, BPEL supports the construct of a *scope*, by defining an activity named *Scope*. Other than the Invoke activity, there is no concept in BPEL that is analogous to the method call, so there is no call stack, *per se*. But Scope activities *can* be nested, much like `try/catch` blocks can be nested. BPEL faults can be handled at each Scope level, and unhandled faults are passed to the next highest enclosing scope, until they appear at the top, "process level", where the BPEL server's fault handling takes over – just like Java's stack dump handler.

At this point the disposition of the BPEL fault depends on the type of operation the process is implementing. For instance, the fault might be sent back to the invoking client as a WSDL response. Most of the fault handling you'll do in BPEL will feel very similar to what we do in Java, once you're familiar with the terminology and the rules involved.

"Undo" – BPEL Compensation and ... *err*...

The last area of comparison we want to look at briefly relates to *transactions*. Transactions implemented in our Java programs typically involve locking records of a database or other resource, executing some logic, and then *committing* the resulting changes in such a way that we get a kind of "all-or-nothing" behavior. That is, either the logic sequence is completed – including any associate data modifications to various resources – or *none* of it is. We typically refer to such transactions as *ACID* because they preserve the ***a***tomicity, ***c***onsistency, ***i***solation, and ***d***urability of the operation(s) encompassed by the transaction. This helps preserve the integrity of our programs' logic execution as well as the integrity of the data managed by those programs. Java transaction managers are specifically designed to provide this functionality.

One issue that always arises when using Java to implement business logic is that many business operations typically occur over the span of hours, days, months and even years in some extreme cases. These periods far exceed the amount of time that we can *practically* hold a lock on a network or data resource. As such, the mechanisms that typically provide ACID reliability require workarounds when implementing business logic.

One area of the WS-BPEL Standard specifically addresses this: *Compensation*. Compensation provides a standardized way to create something referred to in BPEL parlance as a "long-running transaction". There is no direct analog in the Java world for this. Compensation in BPEL is similar in some ways to what most GUI applications provide as an "Undo" feature, but there are some differences. Firstly, the action(s) taken to undo a particular segment of a BPEL process must take place in response to a BPEL fault. Secondly, the activities that perform the compensation must be specified (at design time) by the BPEL Process developer, that is, compensation behavior does not happen *automatically*.

Going *Around* the Jungle

SOA and service orchestration development requires that we overcome a number of different obstacles. Some of these are language-related and some are conceptual. One additional obstacle you'll encounter as you move into WSDL-defined web service orchestration development is working with XML.

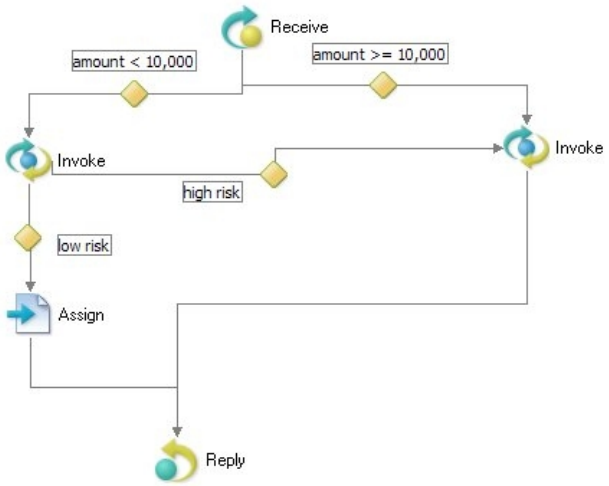
XML is designed to be human-readable. Technically, that goal was accomplished – for very simple schemas and XML instances. Out there in the SOA jungle, however, WSDL definitions get lengthy and often derive from numerous Schema sources. A single BPEL process might define an orchestration involving 25 different services and 50 WSDL definitions, each with its own set of schema definitions. Eventually, all this XML becomes daunting; especially when, say, two separate, 43-character namespace URLs differ by a single letter. Suffice it to say that during the debug phase of your service orchestration development, finding errors in your XML will often make looking for a needle in a haystack look like a vacation.

ActiveBPEL Designer to the Rescue!

Each new generation of languages – and XML-based BPEL is a domain-specific *language* – tends to spawn a collection of tools designed to make working with those languages easier. Because BPEL is XML-intensive and because the service orchestrations developed with it can be virtually incomprehensible once they grow beyond the "Hello, World!" level of complexity, numerous tools have sprung up to aid in creating BPEL processes *graphically*, rather than textually.

The first fully-functional tool of this type was Active-Endpoints' *ActiveBPEL® Designer* (formerly known as *ActiveWebflow Professional*), which was first introduced in 2004. ActiveBPEL Designer is an Eclipse-based, SOA design tool that facilitates BPEL process design and development by allowing the developer to concentrate on the logic and data of the process rather than looking for missing XML end-tags and/or tracking down obscure Java library functions.

As a simple example, Designer supports development using a presentation like the one shown below, rather than the XML-based BPEL syntax shown above. Here is the Loan Application process, as graphically depicted in ActiveBPEL Designer.



ActiveBPEL Designer can take us *around* most of the SOA jungle, so we can get to where we need to be more quickly and easily. A good next step at this point would be to get a copy of the free, fully functional Designer application and go through the Tutorial found in the online Help. This tutorial will take you through the basics of creating a functional BPEL process – and also demonstrate how to simulate, deploy and remotely debug it as well.

Summary

Hopefully this little journey has helped clear away some of the remoteness and confusion in BPEL and SOA. Probably the most important guidance to take away from this trek is:

- Web services can be implemented in any language that supports the required messaging paradigms
- Web service orchestrations are basically just service-enabled programs and can be implemented using anything that supports web service development
- BPEL has a number of constructs that can be easily grasped by a Java developer, simply by looking at their analogs in Java

For additional information:

Get in touch with us at Active Endpoints – http://www.active-endpoints.com – BPEL is what we do!

Full source and documentation for this article can be found here.

OASIS is the standards committee that manages the BPEL standard.