Java Enterprise — budowa aplikacji

Business Process Execution Language

Adam Maciejewski Sylwia Zając

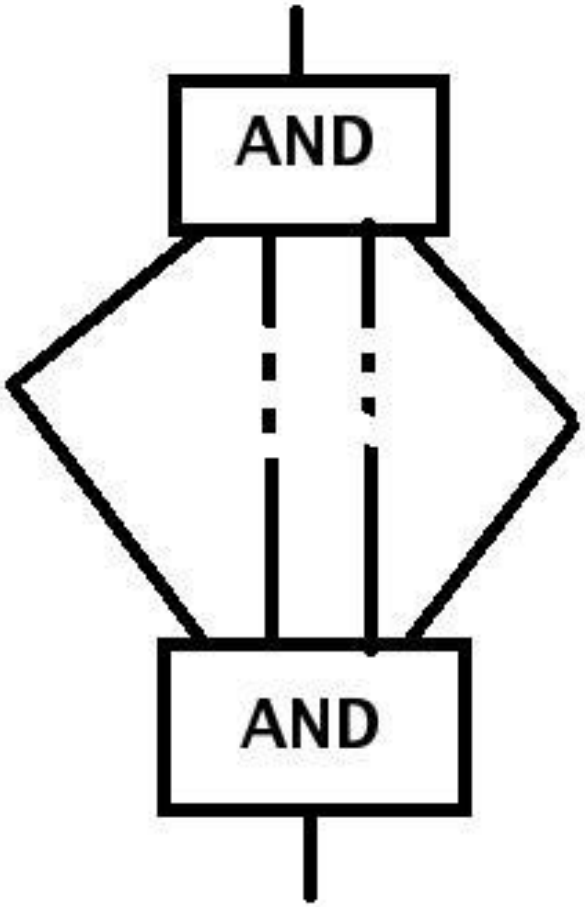
1 czerwca 2007

Business Process Execution Language

Co to jest przepływ pracy…

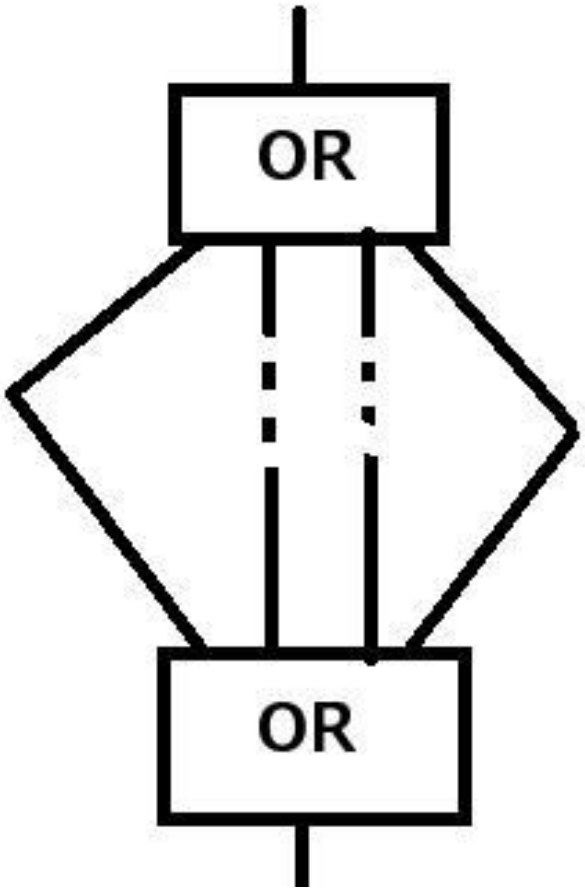
Przepływ pracy to sposób zarządzania zadaniami, przepływem informacji i pracą, w szczególności zaś kolejnością (niekoniecznie sekwencyjną) jej wykonywania.

Kiedy chcemy wszystko . . .



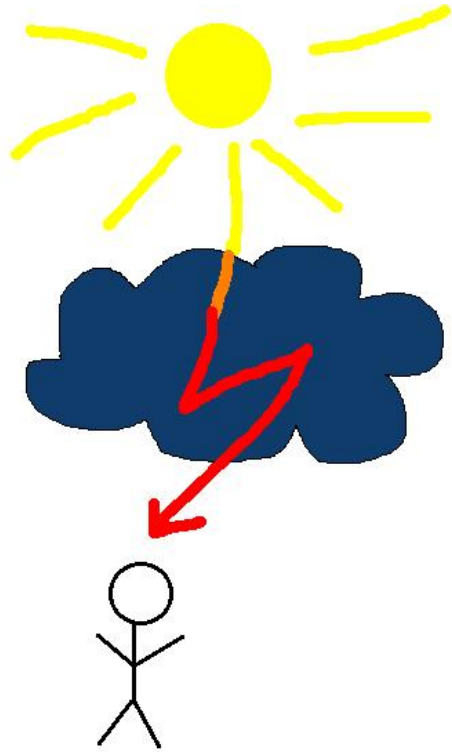
(czyli Flow)

Kiedy chcemy dokładnie jeden . . .



(czyli Pick)

A to dostajemy po złożeniu wszystkich klocków...



czyli proces biznesowy ;-)

Czym jest BPEL

- XMLowe narzędzie do modelowania przepływów pracy
- Business Process Execution Language for Web Services
- Z procesów BPEL mogą korzystać zarówno wolnostojące aplikacje, jak też usługi sieciowe
- Aby używać BPELa nie trzeba wcale znać się na programowaniu, nawet analityk biznesowy może łatwo tworzyć i modyfikować BPELowe przepływy pracy, używając do tego wizualnych narzędzi

Jak zbudować (ręcznie) prosty proces

Procesy biznesowe to tak naprawdę usługi sieciowe realizujące interfejs opisany w języku WSDL. BPEL pozwala nam definiować dwa rodzaje procesów biznesowych:

Jak zbudować (ręcznie) prosty proces

Procesy biznesowe to tak naprawdę usługi sieciowe realizujące interfejs opisany w języku WSDL. BPEL pozwala nam definiować dwa rodzaje procesów biznesowych:

abstrakcyjne określają tylko, jakie wiadomości będą wymieniane z usługami składowymi i z klientem; nie zawierają informacji, co z tymi wiadomościami robić

Jak zbudować (ręcznie) prosty proces

Procesy biznesowe to tak naprawdę usługi sieciowe realizujące interfejs opisany w języku WSDL. BPEL pozwala nam definiować dwa rodzaje procesów biznesowych:

abstrakcyjne określają tylko, jakie wiadomości będą wymieniane z usługami składowymi i z klientem; nie zawierają informacji, co z tymi wiadomościami robić

wykonywalne oprócz wymienianych wiadomości definiują także szczegóły implementacji logiki biznesowej procesu; mogą być uruchamiane, stanowiąc pełnoprawne usługi sieciowe

Deskryptor WSDL

- Wygląda niemal dokładnie tak samo, jak w przypadku zwyczajnej usługi sieciowej, z tym że . . .
- . . . tutaj musimy zadeklarować także porty (*port types*) udostępniane przez usługi składowe oraz typy połączeń z partnerami (*partner link types* — służy do tego specjalne BPELowe rozszerzenie do WSDL)
- W idealnym przypadku każda z usług definiowałaby takie rzeczy za siebie, a nam wystarczyłoby tylko zaimportować potrzebne pliki . . . ale w rzeczywistości przynajmniej typy połączeń musimy określić sami

Deskryptor WSDL — przykład

```
<definitions targetNamespace="http://manufacturing.org/wsdl/purchase"
      xmlns:sns="http://manufacturing.org/xsd/purchase"
      xmlns:pos="http://manufacturing.org/wsdl/purchase"
      xmlns:plnk="http://schemas.xmlsoap.org/ws/2003/05/partner-link/">

<import namespace="http://manufacturing.org/xsd/purchase"
        location="http://manufacturing.org/xsd/purchase.xsd"/>
...
<plnk:partnerLinkType name="purchasingLT">
   <plnk:role name="purchaseService">
       <plnk:portType name="pos:purchaseOrderPT"/>
   </plnk:role>
</plnk:partnerLinkType>
<plnk:partnerLinkType name="invoicingLT">
   <plnk:role name="invoiceService">
       <plnk:portType name="pos:computePricePT"/>
   </plnk:role>
</plnk:partnerLinkType>
</definitions>
```

Elementy `partnerLinkType`

- Każdy element `partnerLinkType` opisuje komunikację pomiędzy parą usług sieciowych; musi on mieć unikatową nazwę oraz zawierać jeden lub dwa elementy `role`

Elementy `partnerLinkType`

- Każdy element `partnerLinkType` opisuje komunikację pomiędzy parą usług sieciowych; musi on mieć unikatową nazwę oraz zawierać jeden lub dwa elementy `role`
- Każdy element `role` opisuje rolę, jaką będzie pełnić usługa po jednej ze stron łącza (jeśli określimy tylko jedną z ról, po drugiej stronie będzie mogła się znaleźć dowolna usługa)

Elementy `partnerLinkType`

- Każdy element `partnerLinkType` opisuje komunikację pomiędzy parą usług sieciowych; musi on mieć unikatową nazwę oraz zawierać jeden lub dwa elementy `role`
- Każdy element `role` opisuje rolę, jaką będzie pełnić usługa po jednej ze stron łącza (jeśli określimy tylko jedną z ról, po drugiej stronie będzie mogła się znaleźć dowolna usługa)
- Wewnątrz elementu `role` musi się znaleźć dokładnie jeden element `portType`, będący odniesieniem do definicji *portu* poprzez który będziemy się komunikować z daną usługą

Definicja procesu — szkielet

```
<process name="ncname" targetNamespace="uri"
      xmlns="http://schemas.xmlsoap.org/ws/2003/03/business-process/">

  <partnerLinks>?
    <partnerLink name="ncname" partnerLinkType="qname"
             myRole="ncname"? partnerRole="ncname"?>+
    </partnerLink>
  </partnerLinks>

  <variables>?
    <variable name="ncname" messageType="qname"?
                type="qname"? element="qname"?/>+
  </variables>

  activity
</process>
```

Definicja procesu — objaśnienia

partnerLinks

Elementy `partnerLinkType` w pliku WSDL deklarowały typy portów i role w komunikacji naszego procesu z innymi usługami; teraz dla każdego łącza jedną z ról przypisujemy definiowanemu procesowi (`myRole`), a drugą jego partnerowi (`partnerRole`)

Definicja procesu — objaśnienia

partnerLinks

Elementy `partnerLinkType` w pliku WSDL deklarowały typy portów i role w komunikacji naszego procesu z innymi usługami; teraz dla każdego łącza jedną z ról przypisujemy definiowanemu procesowi (`myRole`), a drugą jego partnerowi (`partnerRole`)

variables

Odbierane wiadomości będziemy zapisywali na zmiennych — deklarujemy je, określając ich nazwy i typy (podajemy tylko jeden z atrybutów `messageType`, `type` i `element`)

Jak określić przebieg procesu

- Co właściwie nasz proces ma robić?
- Podajemy to w końcowej części pliku z definicją procesu, po wszystkich deklaracjach
- Ogólny schemat jest zwykle taki: odbieramy zlecenie od klienta, wywołujem jakieś zewnętrzne usługi, coś robimy z ich wynikami, by na końcu zwrócić odpowiedź. . .

Najprostszy przykład

```
...
<sequence>
   <receive partnerLink="purchasing"
            portType="lns:purchaseOrderPT"
            operation="sendPurchaseOrder"
            variable="Order"/>

   <invoke partnerLink="invoicing"
           portType="lns:computePricePT"
           operation="performPriceCalculation"
           inputVariable="Order"
           outputVariable="Invoice"/>

   <reply partnerLink="purchasing"
          portType="lns:purchaseOrderPT"
          operation="sendPurchaseOrder"
          variable="Invoice"/>
</sequence>
...
```

Komentarz do przykładu

sequence grupuje elementy, które mają być wykonane kolejno, jeden po drugim

receive czeka na wiadomość z danego łącza i zapisuje ją na podanej zmiennej

invoke wywołuje (synchronicznie) zewnętrzną usługę, wiadomość–zlecenie pobierając z `outputVariable`, a wynik zapisując na `inputVariable`

reply wysyła odpowiedź na zapytanie odebrane przez `receive`, pobierając ją ze wskazanej zmiennej

Atrybut `partnerLink` identyfikuje łącze, które ma zostać użyte do komunikacji z usługą wskazaną przez `portType` i `operation`

Elementy sterujące

- Sekwencyjne
  - sequence
  - switch
  - while
- Współbieżne
  - flow
  - pick
  - wait

Switch — składnia

```
<switch standard-attributes>
 standard-elements
 <case condition="bool-expr">+
   activity
 </case>
 <otherwise>?
   activity
 </otherwise>
</switch>
```

Switch — przykład

```
<switch xmlns:inventory="http://supply-chain.org/inventory"
        xmlns:FLT="http://example.com/faults">
 <case condition= "bpws:getVariableProperty(stockResult,level) > 100">
   <flow>
      <!-- perform fulfillment work -->
   </flow>
 </case>
 <case condition="bpws:getVariableProperty(stockResult,level) >= 0">
   <throw faultName="FLT:OutOfStock"
          variable="RestockEstimate"/>
 </case>
 <otherwise>
   <throw faultName="FLT:ItemDiscontinued"/>
 </otherwise>
</switch>
```

While

Składnia

```
<while condition="bool-expr" standard-attributes>
  standard-elements
  activity
</while>
```

Przykład

```
...
<variable name="orderDetails" type="xsd:integer"/>
   ...
<while condition=
          "bpws:getVariableData(orderDetails) > 100">
 <scope>
   ...
 </scope>
</while>
```

Flow — składnia

```
<flow standard-attributes>
 standard-elements?
 <links>?
   <link name="ncname">+
 </links>
 activity+
</flow>
```

Flow — przykład

```
<flow>
 <links>
   <link name="XtoY"/>
   <link name="CtoD"/>
 </links>
 <sequence name="X">
    <source linkName="XtoY"/>
    <invoke name="A" .../>
    <invoke name="B" .../>
 </sequence>
 ...
```

```
 <sequence name"Y">
    <target linkName="XtoY"/>
    <receive name="C" ...>
      <source linkName="CtoD"/>
    </receive>
    <invoke name="E" .../>
 </sequence>
 <invoke partnerLink="D" ...>
   <target linkName="CtoD"/>
 </invoke>
</flow>
```

Pick – składnia

```
<pick createInstance="yes|no"? standard-attributes>
 standard-elements
 <onMessage partnerLink="ncname" portType="qname"
            operation="ncname" variable="ncname"?>+
   <correlations>?
      <correlation set="ncname" initiate="yes|no"?>+
   </correlations>
   activity
 </onMessage>
 <onAlarm (for="duration-expr" | until="deadline-expr")>*
   activity
 </onAlarm>
</pick>
```

Pick — przykład

```
<pick>
 <onMessage partnerLink="buyer"
                portType="orderEntry"
                operation="inputLineItem"
                variable="lineItem">
       <!-- activity to add line item to order -->
 </onMessage>
 <onMessage partnerLink="buyer"
                portType="orderEntry"
                operation="orderComplete"
                variable="completionDetail">
       <!-- activity to perform order completion -->
  </onMessage>
  <!-- set an alarm to go after 3 days and 10 hours -->
  <onAlarm for="'P3DT10H'">
       <!-- handle timeout for order completion  -->
  </onAlarm>
</pick>
```

Wait

Składnia

```
<wait (for="duration-expr" | until="deadline-expr") standard-attributes>
 standard-elements
</wait>
```

Przykład

```
<sequence>
<wait until="'2002-12-24T18:00+01:00'"/>
 <invoke partnerLink="CallServer" portType="AutomaticPhoneCall"
         operation="TextToSpeech"
         inputVariable="seasonalGreeting">
 </invoke>
</sequence>
```

„Standardowe” . . .

. . . atrybuty

```
name="ncname"?
joinCondition="bool-expr"?      <!-- domyslnie alternatywa -->
                <!-- stanow wszystkich wchodzacych link-ow -->
suppressJoinFailure="yes|no"?   <!-- domyslnie "no" -->
```

. . . elementy

```
<source linkName="ncname" transitionCondition="bool-expr"?/>*
<target linkName="ncname"/>*
```

Przypisania na zmienne

```
...
<assign>
   <copy>
      <from variable="PO" part="customerInfo"/>
      <to variable="shippingRequest"
          part="customerInfo"/>
   </copy>
</assign>
...
```

Przypisywanie wartości na zmienne realizujemy za pomocą elementu `assign`, w którym możemy umieścić jeden lub więcej elementów `copy`. W każdym z nich musimy podać, skąd chcemy skopiować wartość (`from`) i gdzie ma ona zostać zapisana (`to`)

Przypisania — warianty

Przypisywana wartość może być po podana dosłownie:

```
<from> ... literal value ... </from>
```

. . . albo być bardziej złożonym wyrażeniem w języku XPath:

```
<from expression="general-expr"/>
```

. . . możemy ją też pobrać z innej zmiennej lub jej części (jeśli była typu złożonego):

```
<from variable="ncname" part="ncname"?/>
```

Podobnie ze zmienną, na która zapisujemy tę wartość:

```
<to variable="ncname" part="ncname"?/>
```

Przypisania — warianty c.d.

Możemy również manipulować właściwościami wiadomości przechowywanej na wybranej zmiennej:

```
<to variable="ncname" property="qname"/>
<from variable="ncname" property="qname"/>
```

. . . a nawet w trakcie wykonania procesu zmieniać usługi, które będą stać po drugiej stronie łącza:

```
<from partnerLink="ncname" endpointReference="myRole|partnerRole"/>
<to partnerLink="ncname"/>
```

(w części `from` podajemy, czy przypisywana wartość to *endpoint reference* dla naszej roli czy roli partnera; celem przypisania jest zawsze `partnerRole`)

Wywołania asynchroniczne

Nasz proces może pracować zbyt długo, by klienci chcieli zawieszać się w oczekiwaniu na jego zakończenie. Możemy zatem, zamiast korzystać ze standardowego mechanizmu *request–reply*, pozwolić na wywoływanie procesu w sposób asynchroniczny:

1. odbieramy zlecenie przy użyciu elementu `receive`, ale tym razem po łączu *jednokierunkowym*
2. idziemy robić swoje, klient na nas nie czeka
3. gdy skończymy, wywołujemy u klienta specjalnie do tego celu zdefiniowaną operację zwrotną (*callback*), obliczony wynik przesyłając jako jej parametr

Wywołanie asynchroniczne — deskryptor WSDL

```
<portType name="purchaseOrderPT">
   <operation name="sendPurchaseOrder">
      <input message="pos:POMessage"/>
   </operation>
</portType>
<portType name="customerCallbackPT">
   <operation name="sendPurchaseOrder">
      <input message="pos:POMessage"/>
   </operation>
</portType>
...
<plnk:partnerLinkType name="purchasingLT">
   <plnk:role name="purchaseService">
       <plnk:portType name="pos:purchaseOrderPT"/>
   </plnk:role>
   <plnk:role name="customer">
       <plnk:portType name="pos:customerCallbackPT"/>
   </plnk:role>
</plnk:partnerLinkType>
```

Wywołanie asynchroniczne — definicja procesu

```
<partnerLinks>
   <partnerLink name="purchasing"
            partnerLinkType="lns:purchasingLT"
            myRole="purchaseService"
            partnerRole="customer"/>
</partnerLinks>
...
<sequence>
   <receive partnerLink="purchasing"
            portType="lns:purchaseOrderPT"
            operation="sendPurchaseOrder"
            variable="Order">
   </receive>
   ...
   <invoke partnerLink="purchasing"
          portType="lns:customerCallbackPT"
          operation="sendPurchaseOrder"
          variable="Invoice"/>
</sequence>
```

Wywołania asynchroniczne, c.d.

- Również nasz proces może asynchronicznie wywoływać inne usługi, jeśli ich twórcy przewidzieli taką możliwość.
- Musimy tylko zdefiniować odpowiednie operacje zwrotne (*callbacks*), przy użyciu których odbierzemy wyniki
- Potem tworzymy osobną gałąź wykonania procesu, w której najpierw wywołamy (`invoke`) zdalną usługę, po czym zawiesimy się (`receive`) w oczekiwaniu na wynik

Wywołania asynchroniczne — przykład

```
...
<flow>
   ...
   <sequence>
      <invoke  partnerLink="shipping"
               portType="lns:shippingPT"
               operation="requestShipping"
               inputVariable="shippingRequest"/>
      <receive partnerLink="shipping"
               portType="lns:shippingCallbackPT"
               operation="sendSchedule"
               variable="shippingSchedule"/>
   </sequence>
   ...
</flow>
...
```

Obsługa błędów: wyjątki

Rzucanie wyjątku (wewnątrz procesu):

```
<throw faultName="qname" faultVariable="ncname"?
```

. . . i jego łapanie (w deklaracjach)

```
<faultHandlers>?
  <!-- there must be at least one fault handler -->
  <catch faultName="qname"? faultVariable="ncname"?>*
    activity
  </catch>
  <catchAll>?
    activity
  </catchAll>
</faultHandlers>
```

Kryteria łapania wyjątków

- Jeśli wyjątek zawiera dane, to może go złapać `catch`, w którym zgadza się zarówno nazwa, jak i typ; gdy nie ma takiego — ten, który ma zgodny typ, a nie podaje nazwy
- Jeśli danych nie ma, musi się zgadzać nazwa wyjątku
- W ostateczności wszystko łapie `catchAll`

Jak zareagować na błąd?

Najprościej wysłać wiadomość o błędzie:

```
<catch faultName="lns:cannotCompleteOrder"
       faultVariable="POFault">
   <reply partnerLink="purchasing"
          portType="lns:purchaseOrderPT"
          operation="sendPurchaseOrder"
          variable="POFault"
          faultName="cannotCompleteOrder"/>
</catch>
```

trzeba tylko wcześniej zadeklarować to w WSDL–u:

```
<operation name="sendPurchaseOrder">
   <input message="pos:POMessage"/>
   <output message="pos:InvMessage"/>
   <fault name="cannotCompleteOrder"
          message="pos:orderFaultType"/>
</operation>
```

Jak zareagować na błąd, c.d.?

Możemy też spróbować wycofać dotychczas wykonane czynności — służą do tego tzw. *compensation handlers*

Deklaracja:

```
<compensationHandler>
    <invoke partnerLink="Seller" portType="SP:Purchasing"
            operation="CancelPurchase"
            inputVariable="getResponse"
            outputVariable="getConfirmation">
      <correlations>
          <correlation set="PurchaseOrder" pattern="out"/>
      </correlations>
    </invoke>
</compensationHandler>
```

. . . i wywołanie:

```
<compensate scope="RecordPayment"/>
```

Korelacje — co to jest?

- Jeśli prowadzimy naraz kilka asynchronicznych konwersacji z tą samą usługą, to potrzebujemy jakiegoś mechanizmu, by odróżniać od siebie wiadomości należące do każdej z nich
- BPEL daje nam do tego mechanizm tzw. *correlation sets*, czyli zbiorów własności, które będą identyczne we wszystkich wiadomościach składających się na jedną konwersację
- Dzięki temu krążące po sieci wiadomości będą mogły trafić do właściwych egzemplarzy usług czy procesów uruchomionych pod danym adresem

Korelacje w praktyce — definiowanie

W deskryptorze WSDL:

```
<definitions name="correlatedMessages" ...
    xmlns:tns="http://example.com/supplyMessages.wsdl"
    xmlns:bpws="http://schemas.xmlsoap.org/ws/2003/03/business-process/">
    ...
    <bpws:property name="orderNumber" type="xsd:int"/>
    <bpws:propertyAlias propertyName="tns:orderNumber"
              messageType="tns:POMessage" part="PO"
              query="/PO/Order"/>
    ...
```

W definicji procesu:

```
<correlationSets
        xmlns:cor="http://example.com/supplyMessages.wsdl">
    <correlationSet name="PurchaseOrder"
            properties="cor:customerID cor:orderNumber"/>
</correlationSets>
```

Korelacje w praktyce — wykorzystanie

Z korelacji możemy korzystać w elementach `receive`, `reply`, `invoke` i `onMessage`:

```
<invoke partnerLink="Buyer" portType="SP:BuyerPT"
        operation="AsyncPurchaseResponse" inputVariable="POResponse">
   <correlations>
     <correlation set="PurchaseOrder" initiate="no" pattern="out">
     <correlation set="Invoice" initiate="yes" pattern="out">
   </correlations>
</invoke>
```

(atrybut `initiate="yes"` stosujemy, gdy wiadomość ma zapoczątkować konwersację; `pattern` określa, czy nasz proces wysyła, czy odbiera daną wiadomość)

Gdzie umieszczać te wszystkie deklaracje?

- Można globalnie . . . ale można też tworzyć lokalne zakresy:

```
<scope variableAccessSerializable="yes|no">
    <variables>...</variables>?
    <correlationSets>...</correlationSets>?
    <faultHandlers>...</faultHandlers>?
    <compensationHandler>...</compensationHandler>?
    <eventHandlers>...</eventHandlers>?
    activity
</scope>
```

- Niektóre deklaracje mogą się również pojawiać bezpośrednio wewnątrz pewnych elementów (np. `invoke`)

Więcej informacji

Specyfikacja BPEL 1.1
http://www-106.ibm.com/developerworks/webservices/library/ws-bpel/

Business Process Execution Language, Part 1: An Introduction
http://developers.sun.com/jsenterprise/nb_enterprise_pack/reference/techart/bpel2.html

IBM developerWorks: Learning BPEL4WS
http://www.ibm.com/developerworks/views/webservices/libraryview.jsp?search_by=BPEL4WS:

Więcej informacji

A Hands-on Introduction to BPEL
http://www.oracle.com/technology/pub/articles/matjaz_bpel1.html

WS-BPEL Correlation
http://www.computing.dcu.ie/~rbarrett/wsbpel.html

NetBeans Knowledge Base
http://www.netbeans.org/kb/trails/soa.html