02267: Software Development of Web Services

Week 7

Hubert Baumeister
huba@dtu.dk

Department of Applied Mathematics and Computer Science
Technical University of Denmark

Fall 2015

Recap

- BPEL: Automatisation of Business Processes
- Dialogs: Correlation sets
- Fault Handling
- Event Handling: Pick and Event Handler
- Order Process Example

Contents

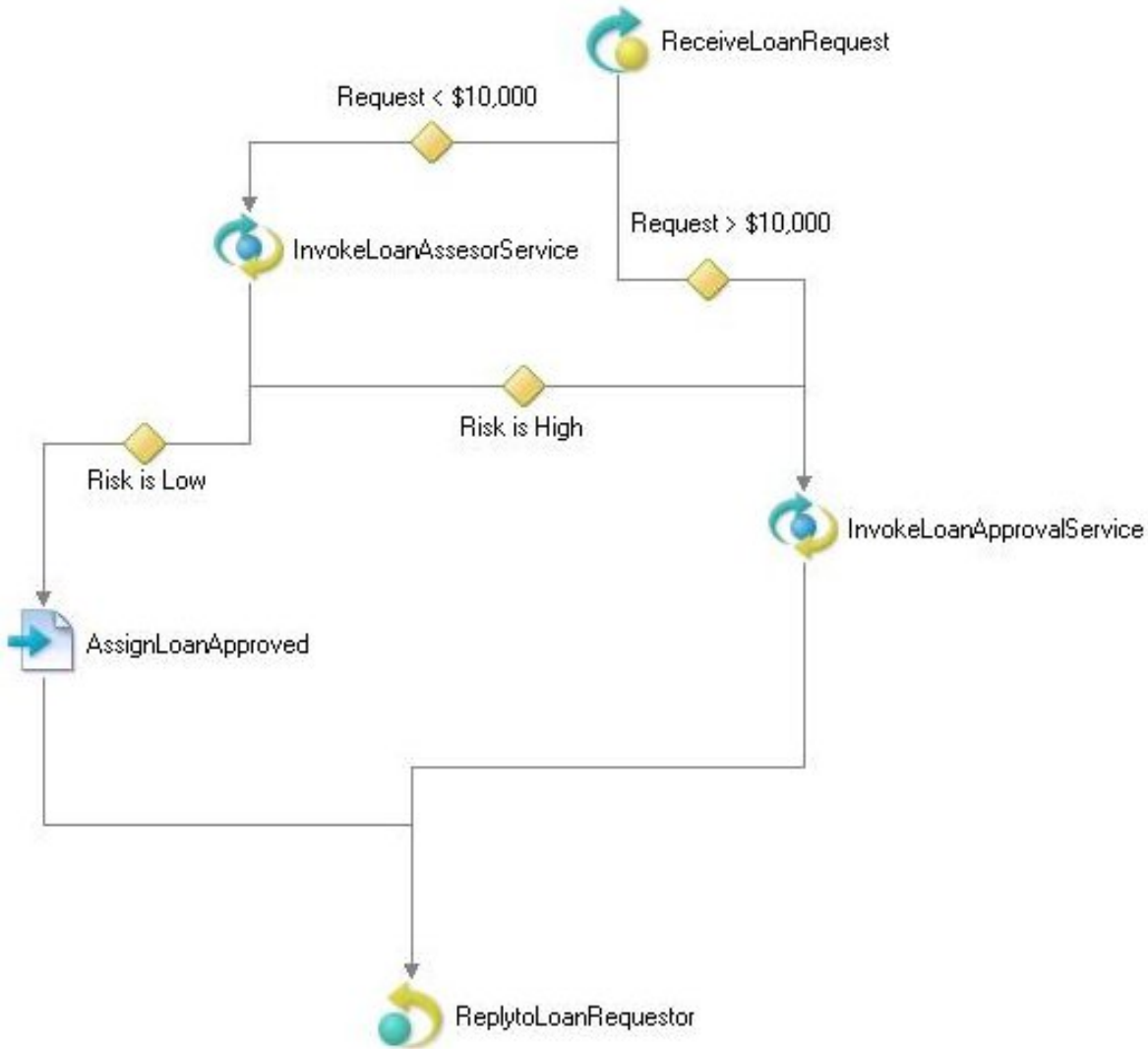
Properties of Business Processes

- Long running activities
  - Sequential execution would take to long for independent activities

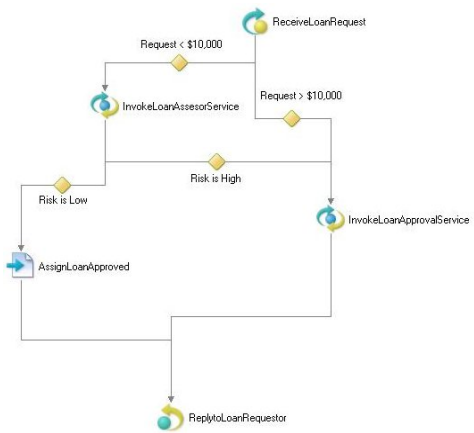
→ Concurrent activities
  - Execute as many activities concurrently as possible
  - Make dependencies of activities explicit

→ Flow construct
  - Activities in principle executed concurrently
    - → set of activities
  - Only link activities if they depend on each other
    - → set of links

Flow construct



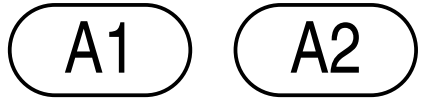
Flow construct



- Activities are executed in principle in parallel
- Activities are linked
  - source of a link is performed first
  - target of the link is performed after source is completed
- Links can only be traversed if their transition condition evaluates to true
- `http://www2.imm.dtu.dk/courses/02267/examples/week07/bpel/loanapproval.bpel`
- `http://www2.imm.dtu.dk/courses/02267/examples/week07/bpel/loanapproval.wsdl.xml`

Concurrent execution of activities

- Activities A1 and A2 are executed concurrently

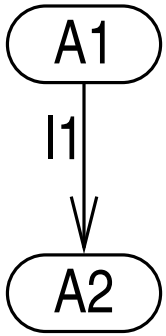


Concurrent Execution

```
<flow>
  <invoke name="A1" partnerLink="Seller" .../>
  <invoke name="A2" partnerLink="Shipper" .../>
</flow>
```

Links between activities

- Activity A1 is performed
- Only after A1 has finished A2 is started



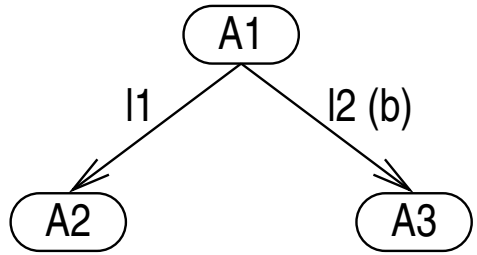
Linking two activities

```
<flow>
  <links>
    <link name="l1"/>
  </links>
  <invoke name="A1" partnerLink="pl1" .. >
    <sources><source linkName="l1"/></sources>
  </invoke>
  <invoke name="A2" partnerLink="pl2" .. >
    <targets><target linkName="l1"/></targets>
  </invoke>
</flow>
```

Two links from one activity

After *A*1 has been completed

- if *b* is true then *A*2 and *A*3 are executed in parallel
- if *b* is not true, only *A*2 is executed

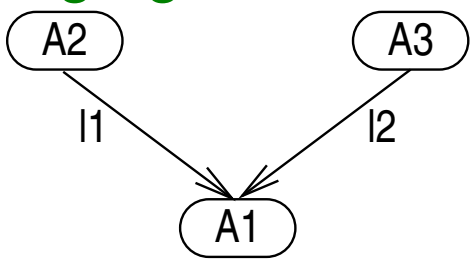


Splitting control flow

```
<flow>
  <links>
    <link name="l1"/><link name="l2"/>
  </links>
  <invoke name="A1" partnerLink="pl1" .. >
    <sources><source linkName="l1"/>
      <source linkName="l2">
        <transitionCondition>b</transitionCondition>
      </source>
    </sources>
  </invoke>
  <invoke name="A2" partnerLink="pl2" .. >
    <targets><target linkName="l1"/></targets>
  </invoke>
  <invoke name="A3" partnerLink="pl3" .. >
    <targets><target linkName="l2"/></targets>
  </invoke>
</flow>
```

Two links merging in one activity

Merging control flow



- *A*1 starts
  1. If both *A*2 and *A*3 have been completed and
  2. If the transition condition of at least one of the incoming transitions is true
- If *A*2 and *A*3 are completed but all the incoming transition conditions are false, then a joinFailure exception is thrown
- Note that this behaviour can be changed
  - with a joinCondition in targets
  - with a suppressJoinFailure attribute on the whole process or single scopes

Flow Activity General Remarks

- The link structure is not allowed to have cycles
- The links can go in and out of nested flow constructs (e.g. if one activity in the flow construct itself contains flow constructs)
- OpenEsb supports the flow construct but does not support links

Contents

Transactions

- Transaction: A transaction represents a set of activities accessing or changing resources that should be performed either as a whole or none of them.
- Transactions should have the ACID property
  - Atomicity: Either the effect of all activites of a transactions is achieved or none
    - Needs to undo the effect of successful activities if later activities in a transaction fail
  - Consistency: Resources are all in a consistent state before and after a transaction
  - Isolation: A transaction can be understood in isolation
    - Several transactions acting on the same resources can be thought of a to run one after the other (serialization; order is undefined)
  - Duration: The effects of a transaction are permanent

Strategies to ensure ACID property

- pessimistic: assume resources are changed by other processes
  - → lock the resources on start of the transaction
  - → Disadvantage: locked resources are not accessible
- optimistic: assume resources are not changed by other processes
  - → don't lock resources, but check if the resources have been changed by other transactions during commit

Variants of transactions

- Atomic Transactions (Short-lived transactions)
  - Short time span: seconds, minutes
  - Can ensure ACID properties
  - Is able to lock resources for the duration of the transaction
- Distributed Resources
  - With distributed resources: The resources can reside on different network nodes
    - E.g. Transfer money from bank *A* to bank *B*

    $\rightarrow$ 2 Phase Commit Protocol (2PC) can be used commit or roll-back distributed resources
- Long-living Transactions
  - Long time span: hours, weeks, months
  - E.g. Atomicy difficult to achieve:
    - Some of the transaction's activity will have a permanent effect

    $\rightarrow$ Use of compensation activities to cancel an activity (e.g. pay by credit card, refund credit card)

Web Services and Transactions

- Transactions in the context of Web services
  - have complex behaviour (more than just database updates) and therefore have application specific compensation actions
  - involve multiple parties and span many organisations (represented by Web services) (distributed)
  - can have long duration
  - are nested

Transcations in BPEL

- Nested transactions: Transactions and atomic actions (e.g. invoking a Web service) are represented by scopes
- Problems with transactions and Web services:
  - Long running and parallel transactions
    - $\rightarrow$ no locking possible
  - Application specific undo actions (e.g. with credit card payment)
    - $\rightarrow$ compensation activities are defined by process designer

$\rightarrow$ Scopes have compensation handlers

- Compensation handlers contain (a sequence of) activities to undo the actions of the scope
- Compensation handler are executed if an error occurs after a successful completion of the scope

Compensation handler execution: Case 1

```
process FH = {default_fault_handler}
        CH = {default_compensation_handler}
   scope s1: FH = {def._fault_hndlr} CH = {ca1}
      .... Some activities ....
   scope s2: FH = {def._fault_hndlr}
             CH = {Invoke refund credit card}
      Invoke send payment to customer
      Receive credit card information from customer
      Invoke bill credit card with bank
   a1
```

- Default Fault Handler:
  a) Call the compensation handlers of the subscopes
  b) Rethrows the exception
- Default Compensation Handler
  a) Call the compensation handlers of the subscopes

Compensation handler execution: Case 2

```
process FH = {default_fault_handler}
        CH = {default_compensation_handler}
   scope s1: FH = {def._fault_hndlr} CH = {ca1}
      .... Some activities ....
   scope s2: FH = {def._fault_hndlr}
             CH = {Invoke refund credit card}
      Invoke send payment to customer
      Receive credit card information from customer
      Invoke bill credit card with bank
      a1
   ...
```

Nested Scopes

```
process FH = {compensate}
    scope s0: CH = {ca0}
       .... Some activities ....
    scope s1: CH = {compensate}
       Receive Order
       scope s2: CH = {invoke request goods back}
          Invoke send goods
          Receive confirm goods have arrived
       scope s3:: CH = {refund payment}
          Invoke send invoice
          Receive payment
          Invoke bill credit card
    a1
```

Compensation

- Who can have compensation handlers?
  - Scopes (including the process itself)
  - Invoke (not supported by NetBeans/GlassFish)
    - → Wrap the invoke in a scope
- Compensation needs to be initiated using the compensate activity
  - Only in fault handlers and compensation handlers
  - Note: the default fault handler executes compensate
  - → own fault handlers need to include the compensate activity

A common situation

```
process
    ...
    foreach purchase
       scope: CH = {invoke cancel_purchase}
          invoke purchase
    ...
```

Compensation Handlers

Compensation Handler

```
<scope>
   <compensationHandler>
      <invoke partnerLink="Seller"
         portType="SP:Purchasing"
         operation="CancelPurchase"
         inputVariable="getResponse"
         outputVariable="getConfirmation">
      </invoke>
   </compensationHandler>
   <invoke partnerLink="Seller"
      portType="SP:Purchasing"
      operation="Purchase"
      inputVariable="sendPO"
      outputVariable="getResponse">
   </invoke>
</scope>
```

- Important: CancelPurchase is only executed if Purchase was successful and e.g. an activity in the super-scope had an error

Compensation Handler

Shortcut for Invoke

```
<invoke partnerLink="Seller"
   portType="SP:Purchasing"
   operation="Purchase"
   inputVariable="sendPO"
   outputVariable="getResponse">
   <compensationHandler>
      <invoke partnerLink="Seller"
         portType="SP:Purchasing"
         operation="CancelPurchase"
         inputVariable="getResponse"
         outputVariable="getConfirmation">
      </invoke>
   </compensationHandler>
</invoke>
```

- Important: Compensation handler with Invoke does not work in OpenEsb. The execution of an invoke with a compensation handler, gives a NullPointerException in the BPEL execution engine

→ Wrap a scope with a compensation handler around the invoke activity

Compensation Handlers default behaviour

1. A fault occurs in a scope or in an invoke
2. The default fault handler of the parent scope
   1. calls compensate activity
   2. rethrows the exception
3. The compensation handler of each successful terminated sub-scope or invoke is executed
   - The default compensation handler for scopes executes the compensation handlers of their sub-scopes

$\rightarrow$ Note if you have your own faultHandlers and if you have compensation handlers, then your fault handler needs to have a compensate activity

Contents

Examination Project

1. Examination Project: 5 weeks
   - Start: Monday 26.10 (next Monday)
   - End: Monday 30.11
   - Teams of 4—6
     - Team building: next Monday

     $\rightarrow$ participation mandatory
   - Implementing Web services (simple, composite, and RESTful)
   - Writing a report
2. Project presentation by the project teams
   - Project presentation ($\approx$ 10min) + questions: total 45 min
   - Dates Tuesday—Friday week 51 (15—18.12)

Contents

RESTful Web Services

- RESTful Web services: based on the structure of the Web as defined through HTTP
  1) based on the concepts of
     - resources indentified by URI's

  $\rightarrow$ Media types (e.g. plain text, XML, HTML, ..., other MIME types) define the representation of the resources
  2) HTTP Methods (GET, POST, PUT, DELETE) act on the resources
     - RESTful Web services define what these operations mean for a particular resource
  3) uses HTTP status codes (e.g. 200 ok; 201 created; . . . , 500 internal server error)

REST = Representational State Transfer

- REST is an architecture style defined by Roy Fielding, one of the authors of HTTP 1.0 and 1.1
- Characteristics
  - Client-server
  - Stateless
    - The server does not hold any application state (e.g. state regarding a session); all the information needs to be provided by the client

    $\rightarrow$ improves scalability
  - Cachebale
  - Uniform Interface
    - use of standard methods, e.g. GET, POST, ...
    - use of standard representation, e.g. MIME types

Uniform Interface

- GET: Get a resource
  - Does not change the state of the resource
  - Is idempotent $\rightarrow$ can be cached
  - Only simple parameters
- PUT: Update a resource
  - Should replace the old representation by a new one
  - Complex content in the body of a HTTP request
- DELETE: Delete a resource
- POST: Do something with the resource (general purpose method)
  - Complex content in the body of a HTTP request
  - URL encoded form parameter

Student Registration

- Classical Web Services focus on services (= operations)
- RESTful Web Services focus on resources

URI Path	Resource Class	HTTP Methods
/institute	InstituteResource	GET, PUT
/institute/address	InstituteResource	GET, PUT
/students	StudentsResource	GET, POST
/students/{id}	StudentResource	GET, PUT, DELETE

- CRUD: Create, Read, Update, Delete

Student Registration: Institute Resource

Institute resource

- When accessing `http://localhost:8080/sr/webresources/intitute` using GET, ”DTU” is returned
- Accessing the same URL using PUT with a string, the name of the institution is changed

```
URI Path                          | Resource Class    | HTTP Methods
----------------------------------------------------------------------
/institute                        | InstituteResource| GET, PUT
```

- Bottom up development:
  1. JAX-RS annotated client
  2. WADL (Web Application Description Language)
  3. Client using Jersey

Using GET on institute resource

Request

```
GET /sr/webresources/institute HTTP/1.1
Accept: text/plain
Host: 127.0.0.1:8070
```

Response

```
HTTP/1.1 200 OK
Content-Type: text/plain

DTU
```

Using PUT on institute resource

Request

```
PUT /sr/webresources/institute HTTP/1.1
Accept: text/plain
Content-Type: text/plain
Host: localhost:8070

Technical University of Denmark
```

Response

```
HTTP/1.1 204 No Content
```

Java implementation: JAX-RS annotations

```
package ws.dtu;

import javax.ws.rs.GET;
import javax.ws.rs.PUT;
import javax.ws.rs.Path;
import javax.ws.rs.core.MediaType;

@Path("institute")
public class InstituteResource {

    private static String name = "DTU";

    @GET
    @Produces(MediaType.TEXT_PLAIN)
    public String getInstituteName() {
        return name;
    }

    @PUT
    @Consumes(MediaType.TEXT_PLAIN)
    public void setInstituteName(String input) {
        name = input;
    }

    @Path("reset") // Resets the name for testing purposes
    @PUT
    public void reset() {
       name = "DTU";
    }
}
```

RESTful Client using Jersey

```
import javax.ws.rs.client.*;
import javax.ws.rs.core.MediaType;
...

public class StudentRegistrationTest {

 Client client = ClientBuilder.newClient();
 WebTarget r =
  client.target("http://localhost:8070/sr/webresources/institute");

 @Test
 public void testGetInstituteName() {
  String result = r.request().get(String.class);
  assertEquals("DTU",result);
 }

 @Test
 public void testPutInstituteName() {
  String expected = "Technical University of Denmark";
  r.request().put(Entity.entity(expected, MediaType.TEXT_PLAIN));
  assertEquals(expected,r.request().get(String.class));
 }

 @Before
 public void resetInstituteName() {
  r.path("reset")
   .request()
   .put(Entity.entity("", MediaType.TEXT_PLAIN));
 }
}
```

Next Week

- Continuing with REST services
  - Student registration example as REST service
  - Representations, Mime-types, JSON
  - Error handling
- Forming of project groups (4–6) and start of examination project
  - Participation next week is mandatory