# **Conversational state** management in **Web Service Technologies**

Homework for

Seminars in Software Engineering

*Author*:
Claudio Di Ciccio
(dc.claudio@gmail.com)

## The service

- "**The service** is a software artifact characterized by its **behavior**
  - the potential evolutions resulting from its interaction with some external systems, such as a client service"
- Given this description, we are focusing on the **behavioral aspect** of services, while we use to consider ***Web Services*** as collections of **remote procedures** to call, widely spread around the network, based on XML communications
- This project focuses on **dynamic self-evolving services**, concentrating the work on
  - the execution of processes behind the service
  - the interaction with clients developing step after step
  - the separation between *application logic* and *process-state logic*

## Model-driven design

- “A **conversation** is a sequence of message exchanges that can occur between a client and a service as part of the invocation of a Web Service”
  - Commonly, Web Services are a collection of request-response operations
    - they do not keep any state information
    - they are completely defined by the communication protocol
      - WSDL
- With model-driven design, we **focus on actions**, not messages
  - for example, only after a login (a web method to call) we can access another functionality (another web method to call)

# Model-driven design

- Every service, once its schema is defined, can be part of a **community**
  - on that community, we can create **composite Web Services**
- **Composition** can be done in **EXPTIME**
  - in the **size** of the **available services**
    - usually, services have not huge defining schemata, then this cost is not impossible to afford

## Service state and behavior management

- Given the service schema, we may need to be supported by a **state manager**
- We may want it to delegate the control over processes' flow
  - It has to act like a referee for addressing
    - concurrency
    - evolution
    - correctness

    of the transitions
  - Thus, the Web Service operations' business logic is separated from the control check
  - On the other hand, the process manager does not mind neither the message exchange, nor the application variables to update

## Conversational Web Services

- Web Services are **stateless** *by default*
  - Based on stateless protocols (SOAP over HTTP, or SMTP...)
  - No information about the caller identity, neither about the client's interaction history
- “A single web service may communicate with multiple clients at the same time, and it may communicate with each client multiple times.
- In order for the web service to track data for the client during **asynchronous communication**, it must have a way to remember which data belongs to which client and to keep track of where each client is in the process of operations.”

## Our goal

- Twofold:
  - **Keep the state**-related information (*stateful* Web Services)
  - Integrate the Web Service with an **automatic behavioral manager**, besides the service implementation
- Solutions must
  - respect standards (for portability reasons) the most as possible
  - use already developed technologies the most as possible
    - so that we can use them in real-world applications

## Technologies

- We choose **Java** as the main coding language
    - J2EE, EJB 3.0
- and Java-based software infrastructures (**JBoss family**)
    - JBoss AS, JBoss WS
        - Application servers, WS-plugin
    - jBPM
        - “Process containers”

## Web Services in J2EE v.5

- Every
  - *POJO* (Plain Old Java Object)
  - ***Stateless** Session Bean*
  - *Servlet*
- can be a **Web Service Endpoint**, given that it respects a defined interface
  - this means that the abstract interface can be independent from the concrete service implementation

## Web Services in J2EE v.5

- The software architect/programmer can define the protocols
  - through classes and interfaces definitions
- The container decides the policy of pooling instances
  - the same concept already seen for EJBs
    - users are unaware of which instance is actually serving the requests
- As told before, no mention about the **business protocol**
  - we call “business protocol” the specification of which messages exchange sequences are supported by the service
    - for example, expressed in terms of constraints on the order that service operations would be invoked in

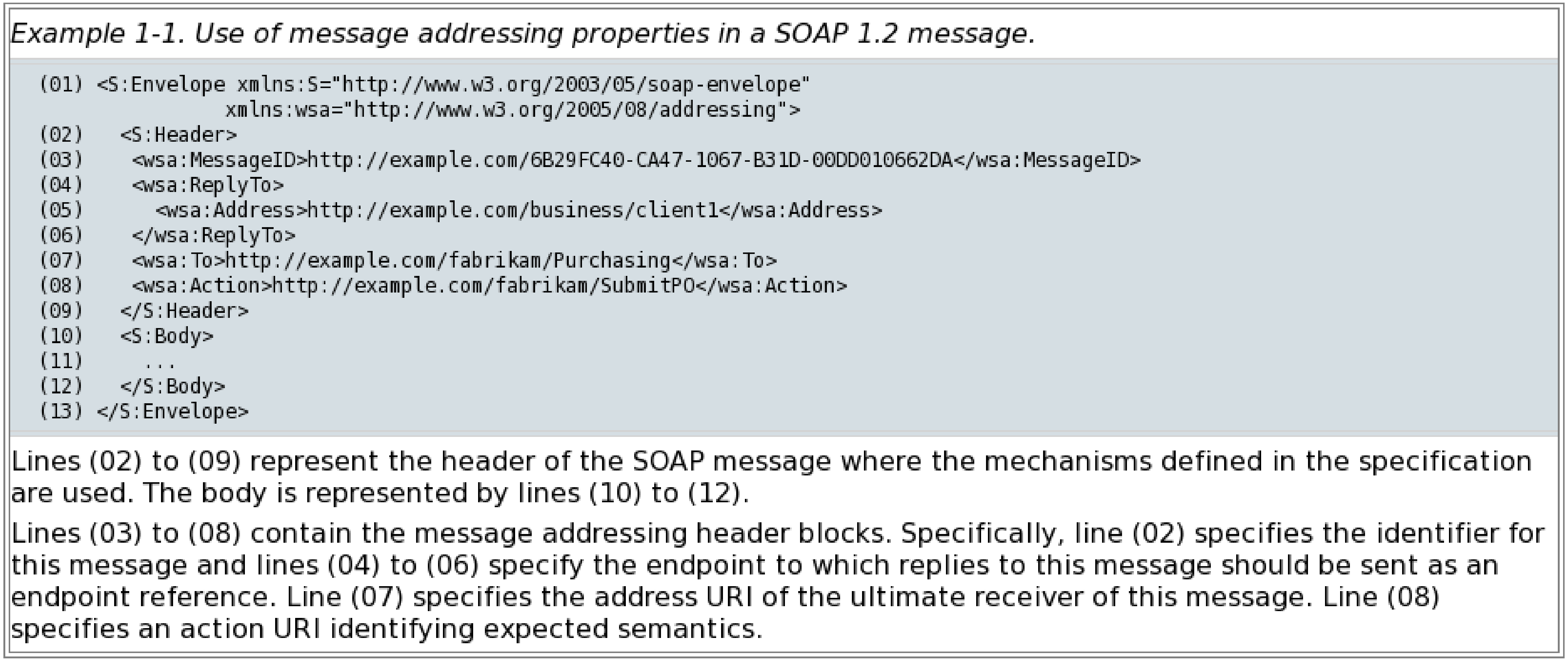
## Stateful Web Services in J2EE v.5

- Almost any kind of Bean can stay behind a WS, so why don't we use ***Stateful*** *Session Beans* to implement it?
  - The compiler does not warn...
  - ... the container does: it's not admitted, as now
- Why don't we associate *Stateless* Session Beans to *Stateful* Session Beans as inner properties?
  - Given you can not decide which instance actually serves you, injecting Stateful Beans inside Stateless ones does not work, either
    - Typically, the container overflows the memory stack after a few calls, as new Stateful objects are created at any WS-call

## Stateful WSs: WS-Addressing

- Solution: **WS-Addressing**
  - http://www.w3.org/TR/ws-addr-core/
- WS-Addressing is a **W3C standard** based on **SOAP**
  - it “defines a family of message addressing properties that convey end-to-end message characteristics including **references** for **source** and **destination endpoints** and **message identity** that allows uniform addressing of messages independent of the underlying transport”
- Essentially, it defines some extra headers to put into the SOAP envelope, so that clients can have a “pointer” to the service endpoint, and identify the caller
  - it suits with our goal!

# Stateful WSs: WS-Addressing



*Example 1-1. Use of message addressing properties in a SOAP 1.2 message.*

```
(01) <S:Envelope xmlns:S="http://www.w3.org/2003/05/soap-envelope"
                 xmlns:wsa="http://www.w3.org/2005/08/addressing">
(02)   <S:Header>
(03)    <wsa:MessageID>http://example.com/6B29FC40-CA47-1067-B31D-00DD010662DA</wsa:MessageID>
(04)    <wsa:ReplyTo>
(05)      <wsa:Address>http://example.com/business/client1</wsa:Address>
(06)    </wsa:ReplyTo>
(07)    <wsa:To>http://example.com/fabrikam/Purchasing</wsa:To>
(08)    <wsa:Action>http://example.com/fabrikam/SubmitPO</wsa:Action>
(09)   </S:Header>
(10)   <S:Body>
(11)     ...
(12)   </S:Body>
(13) </S:Envelope>
```

Lines (02) to (09) represent the header of the SOAP message where the mechanisms defined in the specification are used. The body is represented by lines (10) to (12).

Lines (03) to (08) contain the message addressing header blocks. Specifically, line (02) specifies the identifier for this message and lines (04) to (06) specify the endpoint to which replies to this message should be sent as an endpoint reference. Line (07) specifies the address URI of the ultimate receiver of this message. Line (08) specifies an action URI identifying expected semantics.

- “A reference may contain a number of individual parameters that are associated with the endpoint to facilitate a particular interaction (...) **Reference parameters** are provided by the issuer of the endpoint reference and are assumed to be opaque to other users of an endpoint reference.”

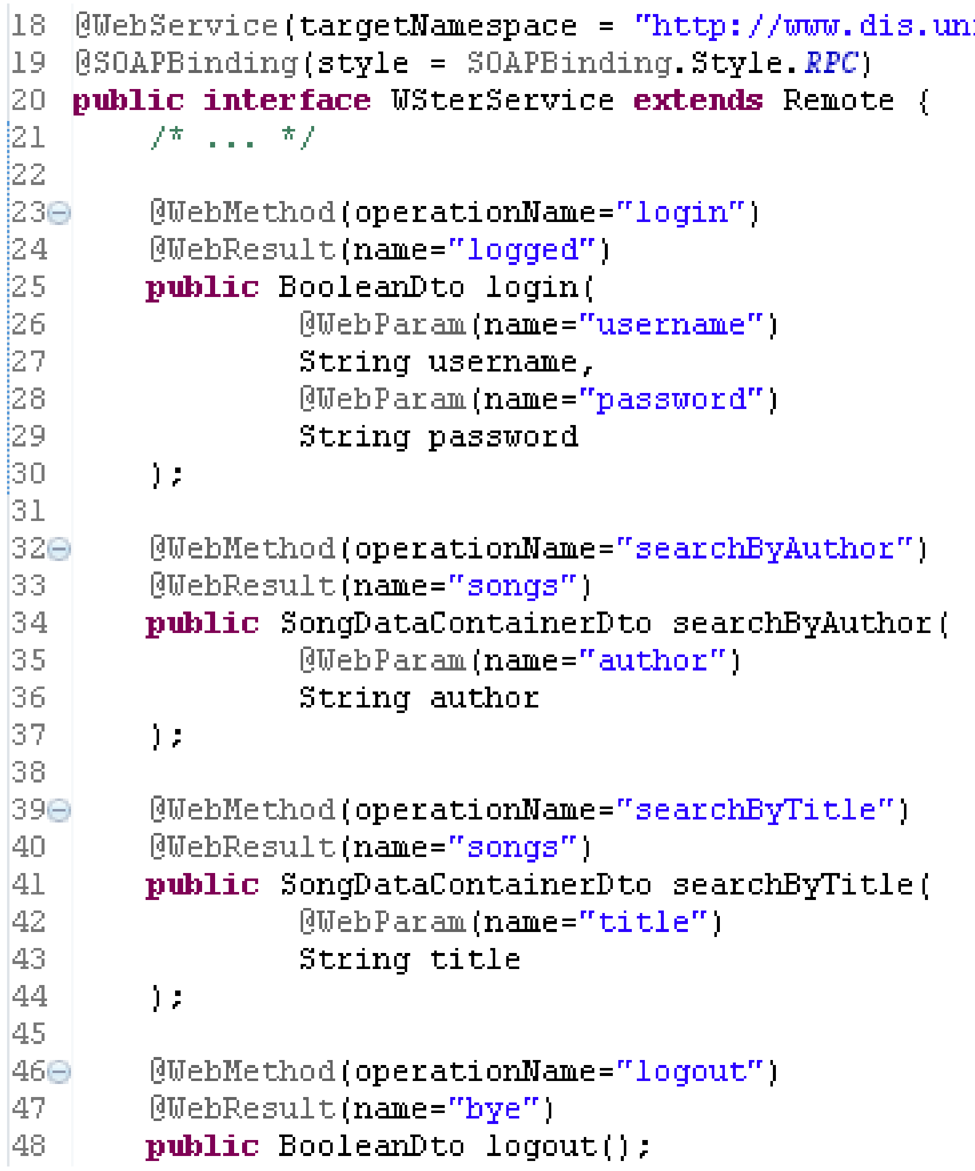
## Stateful WSs: WS-Addressing

- We can insert the client (conversational) id as a `replyTo`'s **reference parameter**
- Every call will be identified by this unique id
- The client will take care of saving it
  - It recalls the SESSION-header for HTTP protocols
  - And like the SESSION-header for HTTP protocols, it can support the maintenance of client-server conversation state

## Resuming the ideas

- Services are processes
  - whose defined behavior is hold by a process manager
    - application layer related
  - whose conversational identity is saved on an identifier
    - a kind of piggy-backing on the payload
      - transport layer related

# Web Services: implementation in J2EE v.5



```java
@WebService(targetNamespace = "http://www.dis.uniroma1.it/sis/conws/wster")
@SOAPBinding(style = SOAPBinding.Style.RPC)
public interface WSterService extends Remote {
    /* ... */

    @WebMethod(operationName="login")
    @WebResult(name="logged")
    public BooleanDto login(
            @WebParam(name="username")
            String username,
            @WebParam(name="password")
            String password
    );

    @WebMethod(operationName="searchByAuthor")
    @WebResult(name="songs")
    public SongDataContainerDto searchByAuthor(
            @WebParam(name="author")
            String author
    );

    @WebMethod(operationName="searchByTitle")
    @WebResult(name="songs")
    public SongDataContainerDto searchByTitle(
            @WebParam(name="title")
            String title
    );

    @WebMethod(operationName="logout")
    @WebResult(name="bye")
    public BooleanDto logout();
```

- The interesting point in **J2EE5** is the use of annotations, linking interface definitions to classes modules

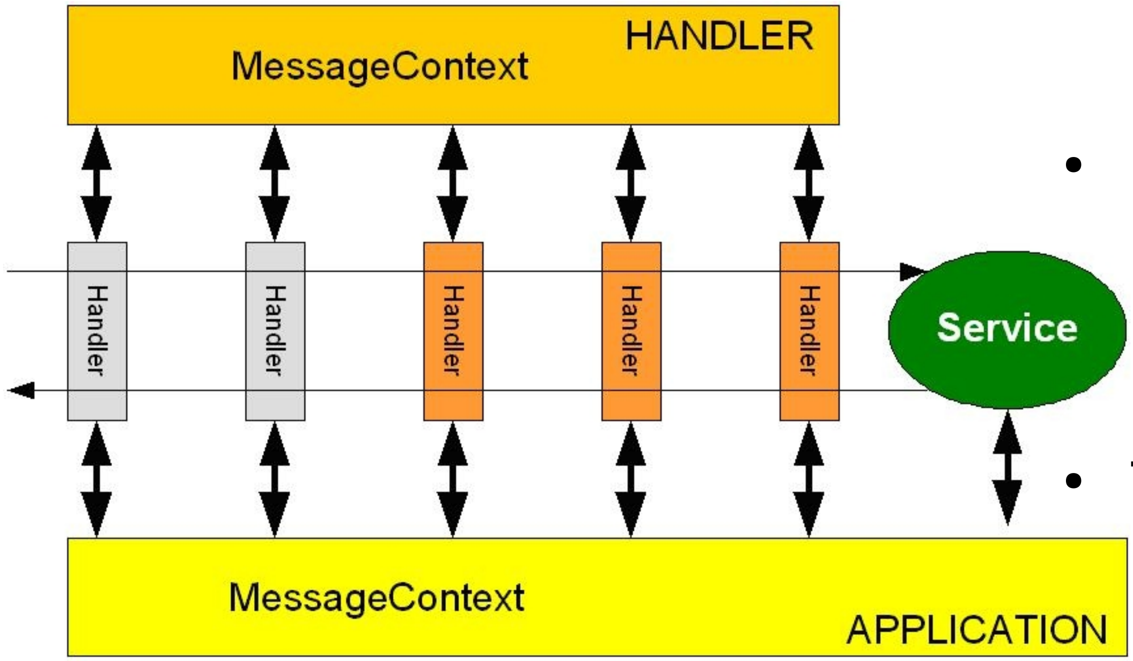
## Web Services: implementation in J2EE v.5

- **Annotations**
  - separate *runtime* instructions from *a priori* definitions
    - separate semantics
      - e.g. communication protocol and application logic
  - can be used instead of external configuration files
  - make the code look cleaner
  - make the code easier to update
    - the amount of cross-references over configuration files decrease

# Implementation in JBossAS / JBossWS

- *JBoss WS* does not support natively annotations like "@Addressing"
  - Sun Glassfish does
- We can use custom **SOAP Message Handlers** in order to support WS-Addressing
  - JBoss WS libraries provide `WSAddressingServerHandler` and `WSAddressingClientHandler` classes
  - we can use them in order to properly set up SOAP headers, putting the conversational id into them

## Message Handlers stack



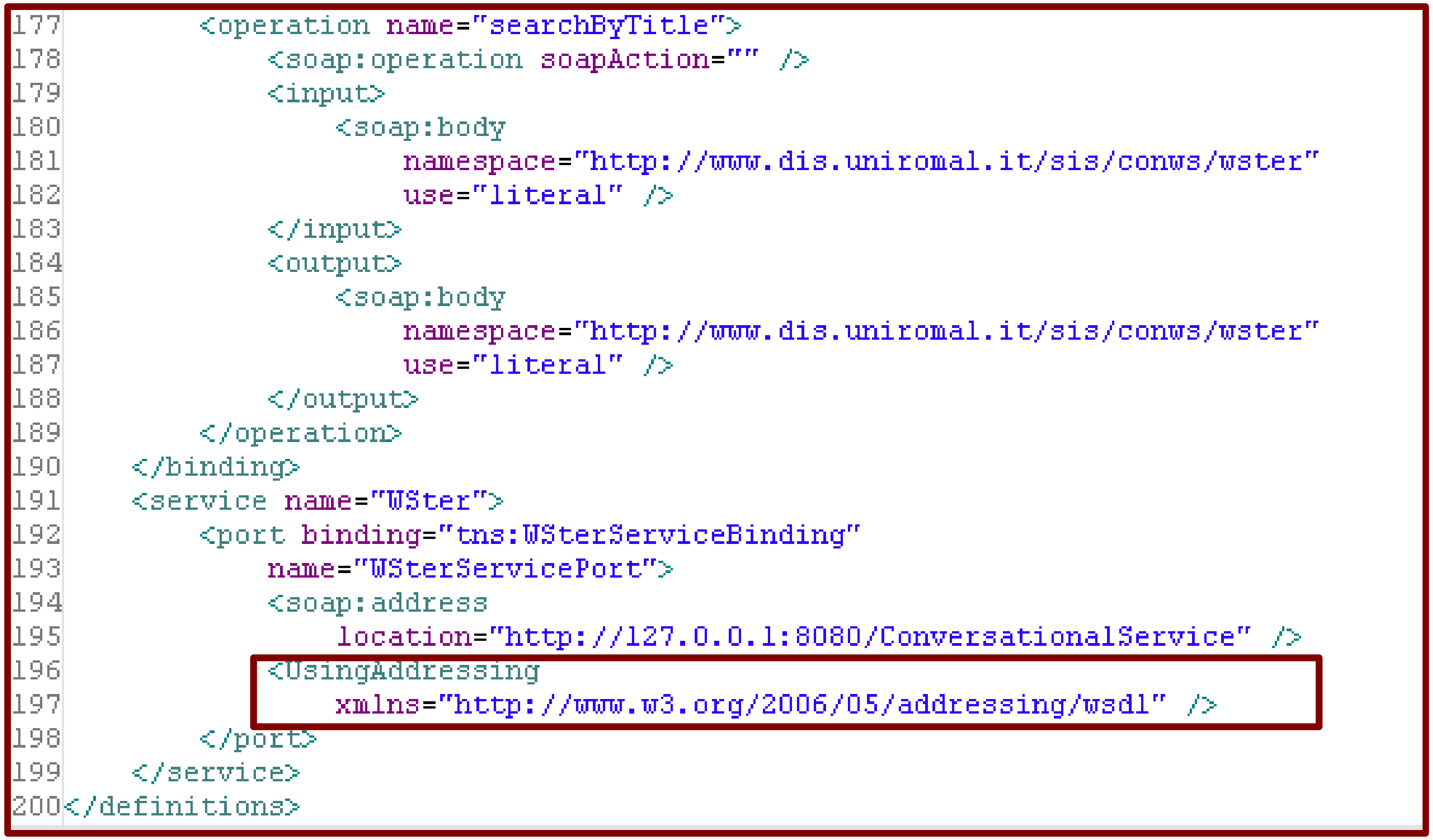
- The link among **Transport** and **Application Layer** is created by **Message Handlers**
- **Client-side** and **server-side** handlers are organized into an ordered list known as "**Handler Chain**".
- The handlers within a handler chain are invoked each time a message is sent or received.
  - Our classes, enriching SOAP headers, lay down there

## Some code: server side

```java
@WebService(
        name = "WSterService",
        targetNamespace = "http://www.dis.uniroma1.it/sis/conws/wster",
        serviceName = "WSter",
        wsdlLocation = "WEB-INF/wsdl/WSter.wsdl",
        endpointInterface = "it.uniroma1.dis.sis.wster.service.WSterService"
)
@EndpointConfig(configName = "Standard WSAddressing Endpoint")
@HandlerChain(file = "WEB-INF/jaxws-handlers.xml")
@SOAPBinding(style = SOAPBinding.Style.RPC)
public class WSterServiceEndpoint implements WSterService {
```

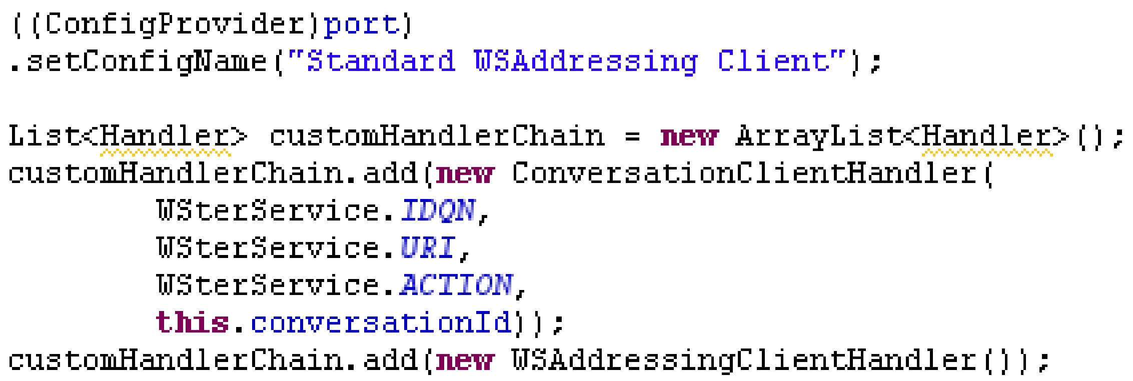
```xml
<handler-chains xmlns="http://java.sun.com/xml/ns/javaee"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://java.sun.com/xml/ns/javaee javaee_web_services_1_2.xsd">

    <handler-chain>
        <protocol-bindings>##SOAP11_HTTP</protocol-bindings>
        <handler>
            <handler-name>WS-Addressing Server Handler</handler-name>
            <handler-class>
                org.jboss.ws.extensions.addressing.jaxws.WSAddressingServerHandler
            </handler-class>
        </handler>
        <handler>
            <handler-name>Conversational Server Handler</handler-name>
            <handler-class>
                it.uniroma1.dis.sis.wster.service.handler.ServerHandler
            </handler-class>
        </handler>
    </handler-chain>

</handler-chains>
```

# Some code: server side



```
        <operation name="searchByTitle">
            <soap:operation soapAction="" />
            <input>
                <soap:body
                    namespace="http://www.dis.uniroma1.it/sis/conws/wster"
                    use="literal" />
            </input>
            <output>
                <soap:body
                    namespace="http://www.dis.uniroma1.it/sis/conws/wster"
                    use="literal" />
            </output>
        </operation>
    </binding>
    <service name="WSter">
        <port binding="tns:WSterServiceBinding"
            name="WSterServicePort">
            <soap:address
                location="http://127.0.0.1:8080/ConversationalService" />
            <UsingAddressing
                xmlns="http://www.w3.org/2006/05/addressing/wsdl" />
        </port>
    </service>
</definitions>
```

## Some code: client side

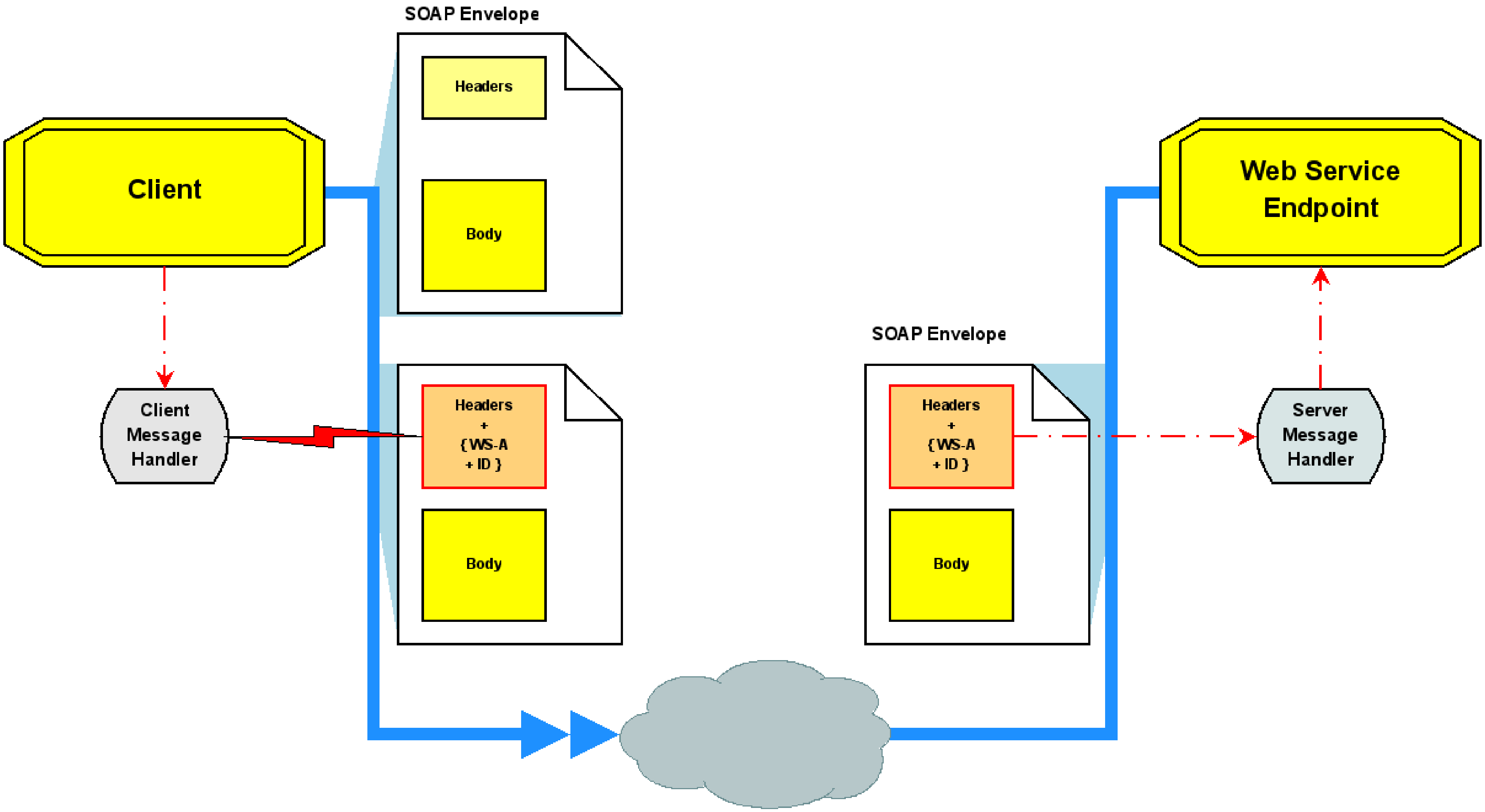


```
((ConfigProvider)port)
.setConfigName("Standard WSAddressing Client");

List<Handler> customHandlerChain = new ArrayList<Handler>();
customHandlerChain.add(new ConversationClientHandler(
        WSterService.IDQN,
        WSterService.URI,
        WSterService.ACTION,
        this.conversationId));
customHandlerChain.add(new WSAddressingClientHandler());
```

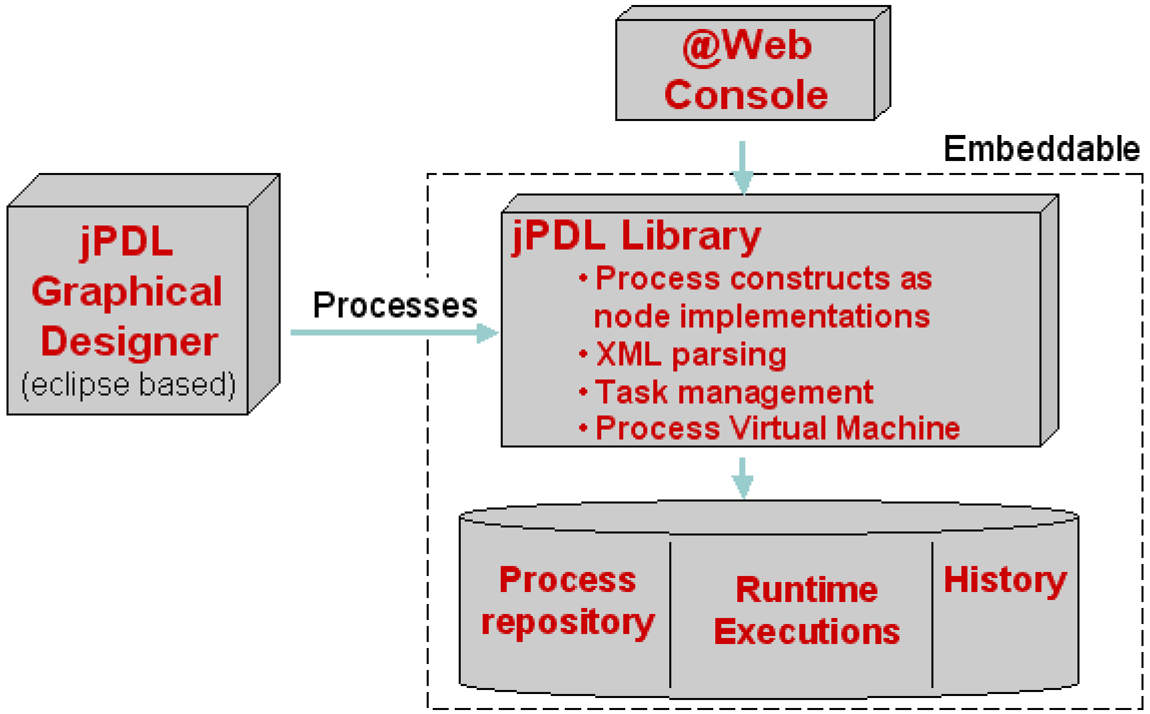
## Resuming ideas

- In order to implement a conversational Web Service framework we have to
  - Implement the WS-Addressing framework
    - Implement Message Handlers, both for client and server
      - able to insert WS-Addressing tags into the SOAP headers
    - Put those Message Handlers into the Handlers Stack
    - Declare the WS-Addressing standard compliance into the WSDL file
  - Insert the conversational id as a parameter of WS-Addressing headers
    - updating properly the message handling methods into the previously defined Message Handler classes

# Resuming ideas



# Services as Processes: jBPM



- **jBPM** (**JBoss Business Process Manager**) is a framework for advanced processes management
- jBPM is designed to be a real *workflow-management* tool
  - We use a subset of its facilities

## Services as Processes: jBPM

- jBPM supports native **jPDL** (**jBPM Process Definition Language**) language
  - **XML based**
  - *BPEL-extensible*, in case we want to integrate it with a pre-existent orchestrator
  - *Graph-Oriented*

## Why jPDL?

- It is XML-based
  - XML is widely supported
  - XML is structured
    - far looser than relational schemata
  - Frameworks, languages, technologies for interacting, transforming, querying, updating XML data are already defined
- You can make queries over contents and structures
  - as we may use *HennessyMilner* formulas for asserting properties about FSMs, we can use *XPath* for expressing them about XML-based jPDL descriptors
- jPDL is very **close to WS-TSL**

# Finite State Machines

- FSMs (**Finite State Machines**) are the construct that we use to describe services' behavior (*Roman approach*).
- We recall here that a deterministic FSM (TS, standing for Transition System) is defined as follows:

$$TS = \langle \Sigma, S, s_0, \delta, F \rangle$$

- Where
  - $\Sigma$ is the finite alphabet of actions
  - $S$ is the finite set of states
  - $s_0$ is the initial state, in $S$
  - $\delta$ is the transition function
  - $F$ is the set of finite states
    - $F \subseteq S$

# Non-deterministic FSMs

- It is quite common for Web Services to have transitions that do not directly depend from clients' invocations.
  - For example, the server, not the client, decides whether user-names and passwords provided are correct for the user to login
    - and then access protected functionalities
      - that is, make the service evolve to states that, otherwise, would be not accessible
  - Thus, we have ***devilish*** **non-determinism**
  - Consequently, we have to consider this new formal object

$$TS_N = \langle \Sigma, S, s_0, \delta_N, F \rangle$$

  - Where every symbol maintains the previous meaning, except

$$\delta_N \subseteq S \times \Sigma \times S$$

    - that is a **relation**, not a function anymore (the same action can make the service proceed through different states)

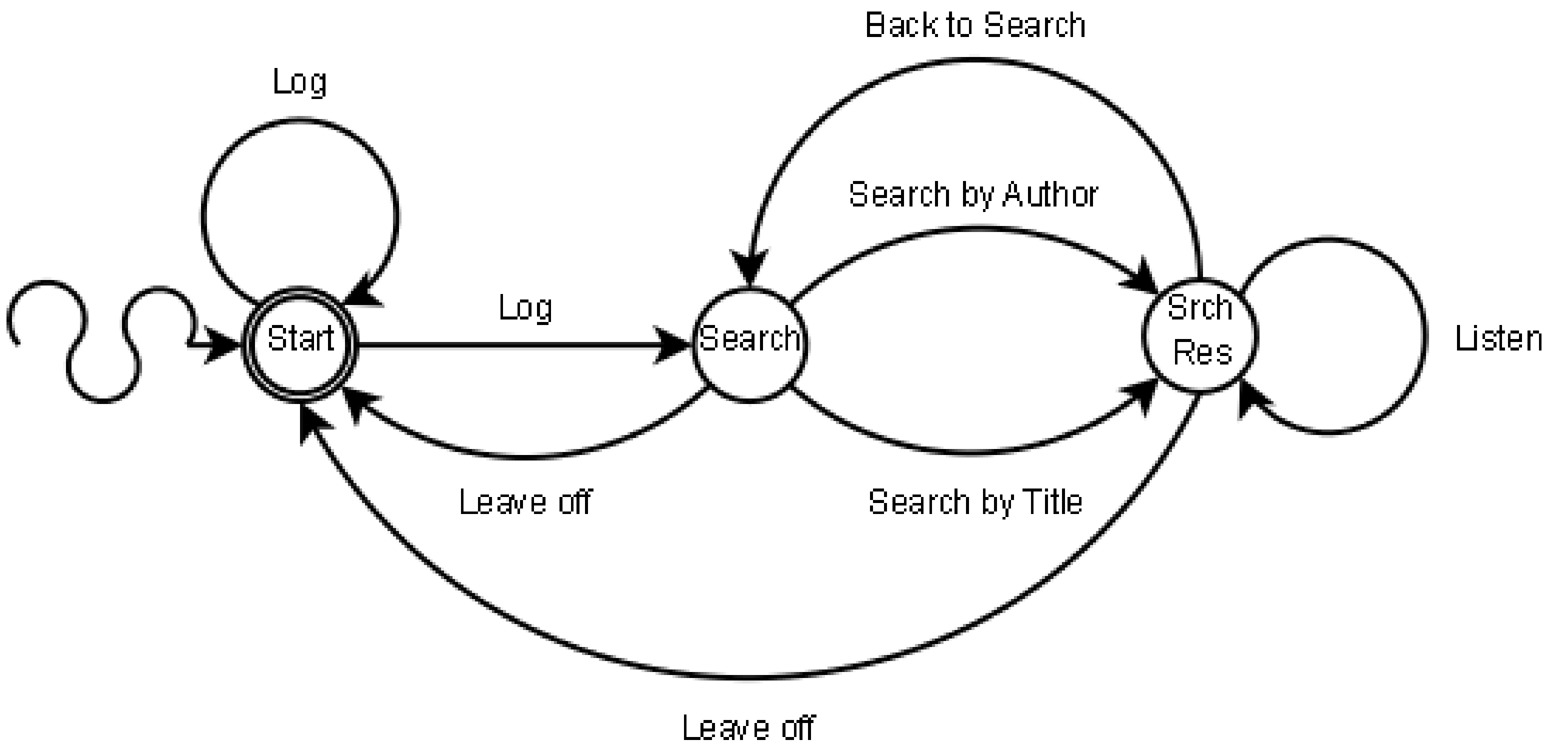
# From TSs to jPDL documents

- As you can see, mapping deterministic FSMs to jPDL (XML) documents is quite straightforward

| | |
|---|---|
| $s_0$ | <start-state name="$s_0$"> |
| $s \in S$ | <state name="$s$"> |
| $f \in F$ | <end-state name="$f$"/> |
| $\delta(s, a) = s'$ | <state name="$s$"><br>  <transition to="$s'$" name="$a$"><br></state> |

# From TSs to jPDL documents

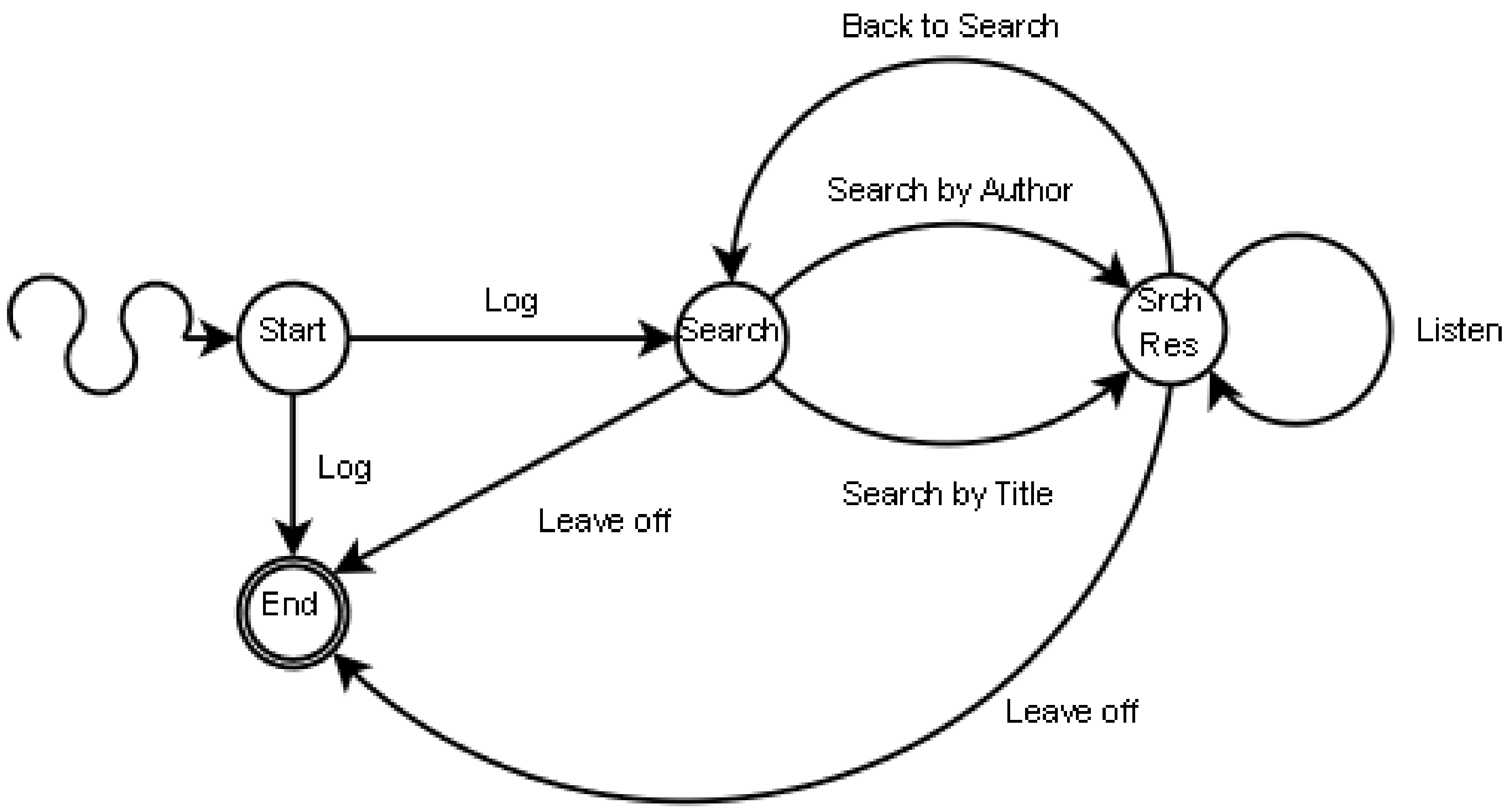
- Two conditions are imposed by jPDL, regarding det. FSMs:
  - start-states and end-states cannot be the coincident
  - transitions cannot end to start-states, nor start from end-states
- Thus, a FSM like the following have to be changed...



# From TSs to jPDL documents

– ... into the following one

- Final states have to turn into nodes without further transitions admitted
- Start states must not have reentrant arcs



## Non-deterministic FSMs

- In this example, “Log” is the non-deterministic devilish transition
  - jPDL does not admit transitions to have multiple reachable states
    - jPDL documents are used to describe internal processes that are going to be automatically managed by the jBPM engine
    - it is obvious, then, that jBPM is not designed for managing non-deterministic processes: **non-determinism** in services' Transition Systems **is used to hide some internal details**
    - every concrete program is deterministic, and there is no behavioral information to hide to the process manager
  - In these cases, we can use special nodes called ***decision-state***s

## Process manager's capabilities usage

- We may want to use the process manager engine as
  1. a *passive* flow controller
     - It checks that every transition is legally invoked
  2. an *active* flow controller
     - It decides the state to evolve to, when the FSM (execution graph) has a multi-arc
       - decides, in case of *devilish* (server-dependent) non-determinism, the transition to move through
  3. an *active* flow controller, able to call *user-defined actions*
     - When the client asks for a transition to proceed, and it's linked to an action, the process manager calls the proper action handler

## Process manager's capabilities usage

- We choose the **third option** because:
  - it includes the previous ones;
  - jBPM can directly manage exceptions thrown during the action;
    - the exception recovery policy and rollback is due to it, not to the calling method
  - it enriches the focus on actions, rather than using the framework as a simple flow-control instrument.
- jPDL does not require the programmer to write any code line into the definition file
  - It delegates the action, or the decisions, to proper Java classes defined somewhere else by the programmer
    - The definition remains abstract: it's like putting another labeling
      - Encoding classes is like giving an interpretation to the schema

## From TSs to jPDL: update

- We have to enlarge the algebra representing Transition Systems for internal representation, by adding three new labeling functions and three new sets

$$PTS = \langle \Sigma, S, s_0, \delta, F, A, \alpha, H, \eta, D, \omega \rangle$$

  - As you can see, we are considering $\delta$ as a function, and not as a relation
- Where
  - $A$ is the set of action-handlers (labels)
  - $D$ is the set of decision-states
    - $D \subseteq S$
  - $H$ is the set of decision-handlers (labels)

# From TSs to jPDL: update

$$PTS = \langle \Sigma, S, s_0, \delta, F, A, \alpha, H, \eta, D, \omega \rangle$$

- $\alpha$ is the partial labeling function that, given a state $s$ and an action $a$, returns an action-handler $e$ (element of $A$); we assume that only if $\delta$ is defined on $s$ and $a$, then $\alpha$ is defined as well
- $\eta$ is the labeling function that, given a decision-state $d$ (in $D$) returns a decision-handler (element of $H$)
- $\omega$ is the transition function that, given a decision-state $d$ (in $D$) and an action $a$, returns a state $s$

# jPDL syntax: update

$d \in D$

```
<decision name="d"/>
```

$\alpha(s,a)=e$

```
<state name="s">
  <transition to="δ(s, a)" name="a">
    <action class="e"/>
  </transition>
</state>
```

$\eta(d)=h$

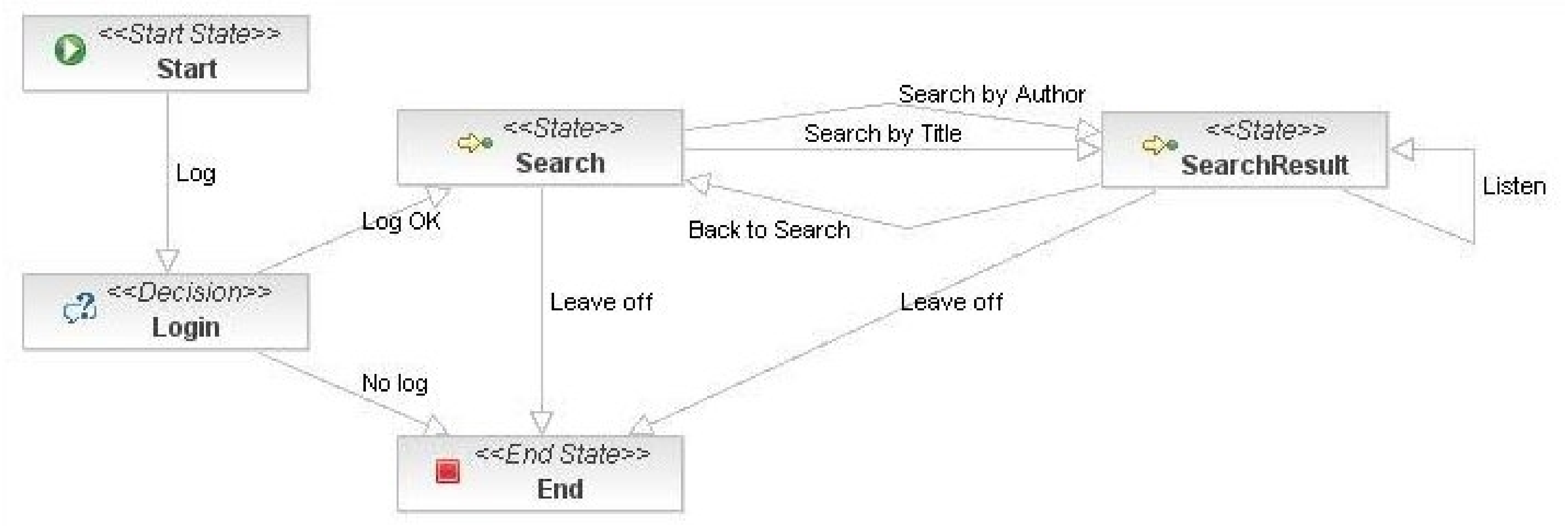
```
<decision name="d"><handler class="h"/></decision>
```

$\omega(d,a)=s$

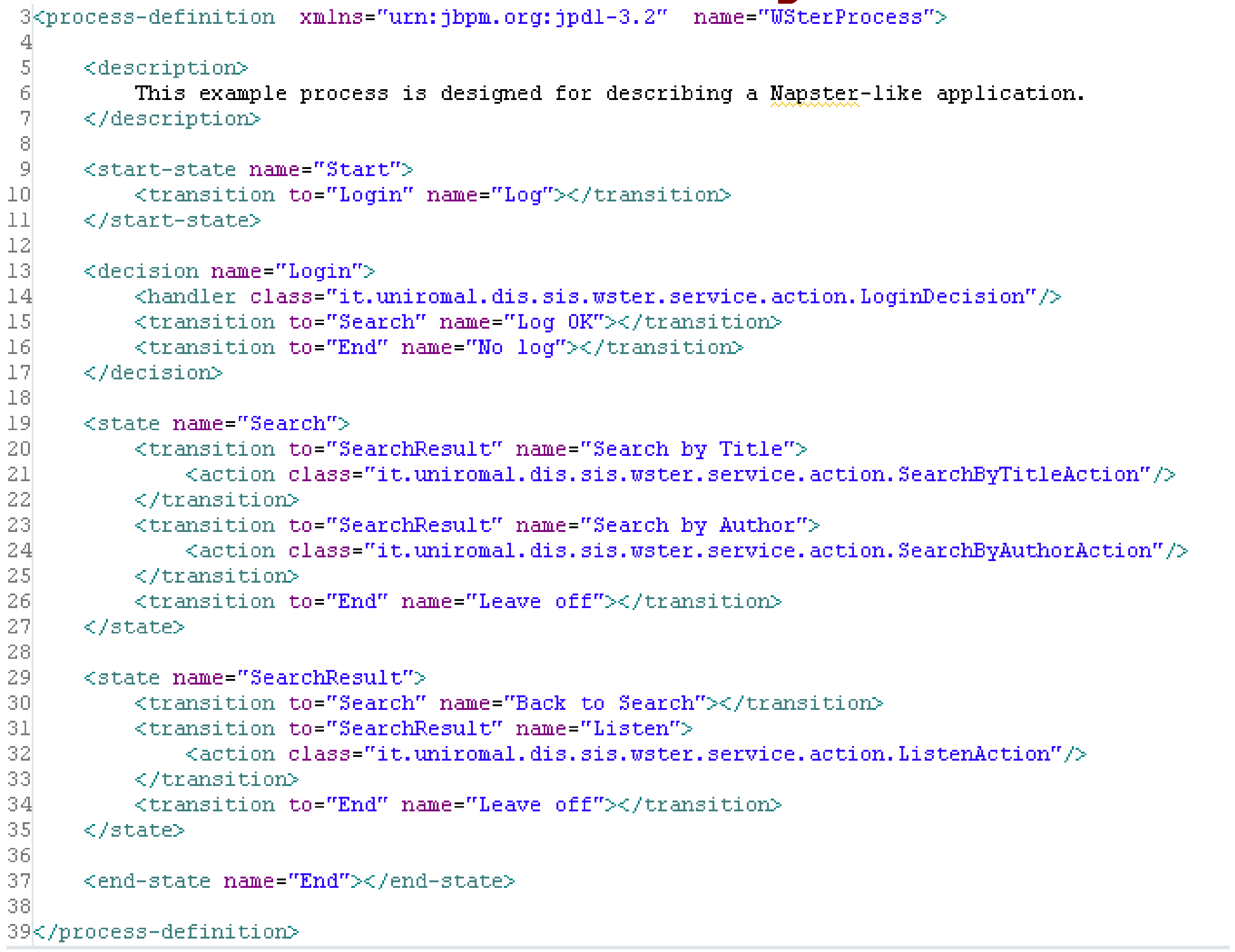
```
<decision name="d">
  <handler class="α(d, a)"/>
  <transition to="s" name="a"/>
</decision>
```

# jBPM process

- Here is the example service behavior, expressed as a jBPM process
  - The next slide will show the same process, XML-encoded



# Some code: jPDL



```xml
<process-definition  xmlns="urn:jbpm.org:jpdl-3.2"  name="WSterProcess">

    <description>
        This example process is designed for describing a Napster-like application.
    </description>

    <start-state name="Start">
        <transition to="Login" name="Log"></transition>
    </start-state>

    <decision name="Login">
        <handler class="it.uniroma1.dis.sis.wster.service.action.LoginDecision"/>
        <transition to="Search" name="Log OK"></transition>
        <transition to="End" name="No log"></transition>
    </decision>

    <state name="Search">
        <transition to="SearchResult" name="Search by Title">
            <action class="it.uniroma1.dis.sis.wster.service.action.SearchByTitleAction"/>
        </transition>
        <transition to="SearchResult" name="Search by Author">
            <action class="it.uniroma1.dis.sis.wster.service.action.SearchByAuthorAction"/>
        </transition>
        <transition to="End" name="Leave off"></transition>
    </state>

    <state name="SearchResult">
        <transition to="Search" name="Back to Search"></transition>
        <transition to="SearchResult" name="Listen">
            <action class="it.uniroma1.dis.sis.wster.service.action.ListenAction"/>
        </transition>
        <transition to="End" name="Leave off"></transition>
    </state>

    <end-state name="End"></end-state>

</process-definition>
```

# HennessyMilner / XPath

- Some examples about how to express HM formulas over TSs as XPath queries over jPDL descriptors
  - HM

    $Search \;\rightarrow\; \langle Search{\cdot}by{\cdot}Title\rangle T \;\wedge\; [Search{\cdot}by{\cdot}Title]SearchResult$
  - XPath

```
boolean(
      not(//state[@name="Search"])
      or (
            //transition[
                  parent::state[@name="Search"]
                  and @name="Search by Title"
                  and @to="SearchResult"]
      )
)
```

  - TRUE

# HennessyMilner / XPath

- Some examples about how to express HM formulas over TSs as XPath queries over jPDL descriptors
    - HM

      $Search \rightarrow \langle any \rangle T \wedge [any - Search{\cdot}by{\cdot}Title] F$
    - XPath

```
boolean(
     not(//state[@name="Search"])
     or (
          //transition[parent::state[@name="Search"]]
          and not(
               //transition[parent::state[@name="Search"]
               and not(@name="Search by Title")
               ]
          )
     )
)
```

    - FALSE

# HennessyMilner / XPath

- Some examples about how to express HM formulas over TSs as XPath queries over jPDL descriptors
  - HM

    $Start \rightarrow \langle any \rangle T \wedge [any - Log]F$

  - XPath

```
boolean(
      not(//start-state[@name="Start"])
      or (
            //transition[parent::start-state[@name="Start"]]
            and not(
                  //transition[
                        parent::start-state[@name="Start"]
                        and not(@name="Log")]
            )
      )
)
```

  - TRUE

## Web Service / jBPM interaction

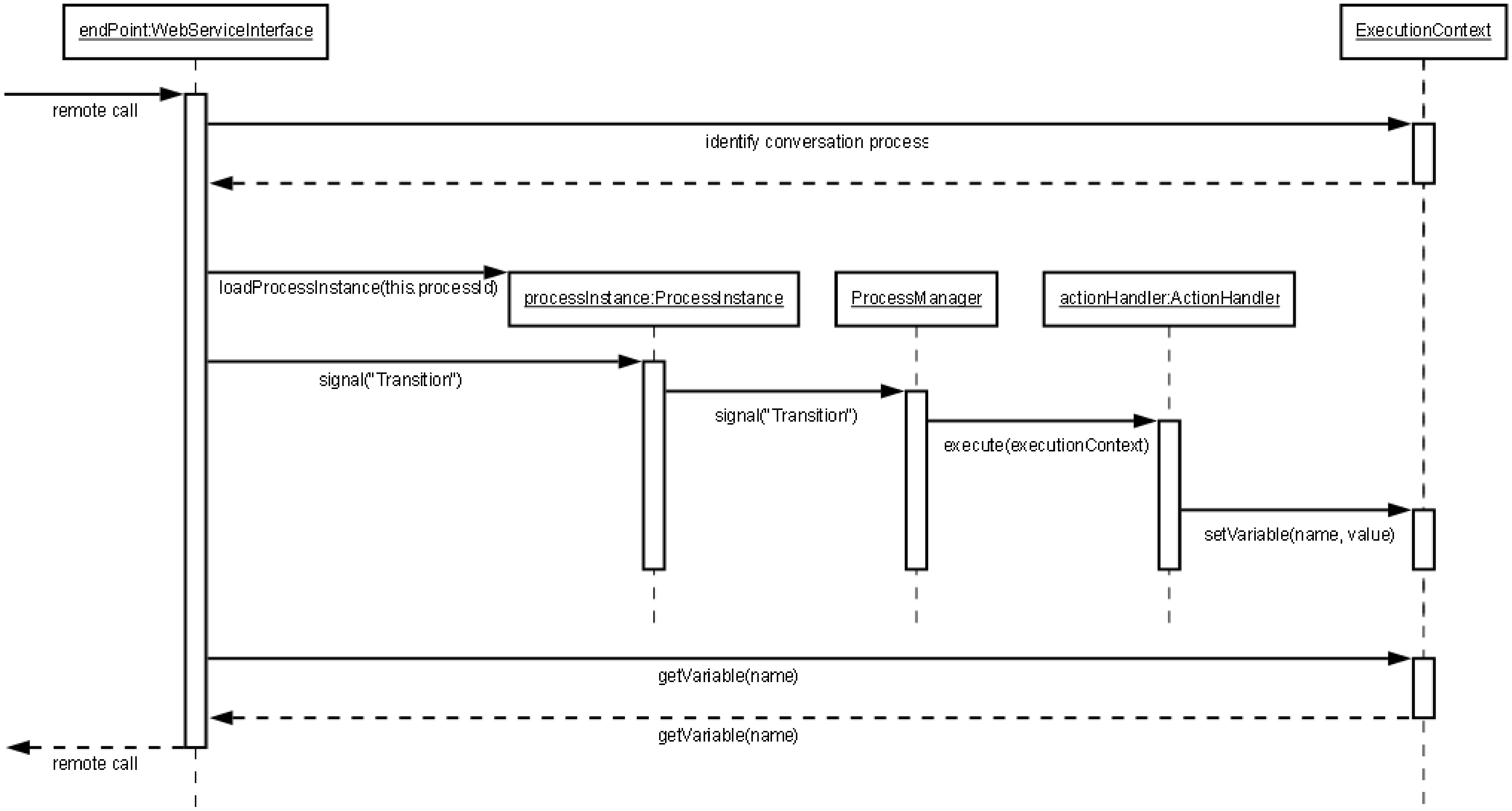
- In the framework we are describing now, Web Services act like a ***façade*** for the real service core
- The service core is spread over
  - jBPM
    - the process manager
  - Action-handlers and decision-handlers
    - Java beans implementing the

      `org.jbpm.graph.def.ActionHandler`

      `org.jbpm.graph.node.DecisionHandler`

    interfaces
- Programmers must implement *ActionHandlers* and *DecisionHandlers*
  - Thus, they provide the semantics behind process descriptors

# Web Service / jBPM interaction


endPoint:WebServiceInterface
ExecutionContext
remote call
identify conversation process
loadProcessInstance(this.processId)
processInstance:ProcessInstance
ProcessManager
actionHandler:ActionHandler
signal("Transition")
signal("Transition")
execute(executionContext)
setVariable(name, value)
getVariable(name)
getVariable(name)
remote call

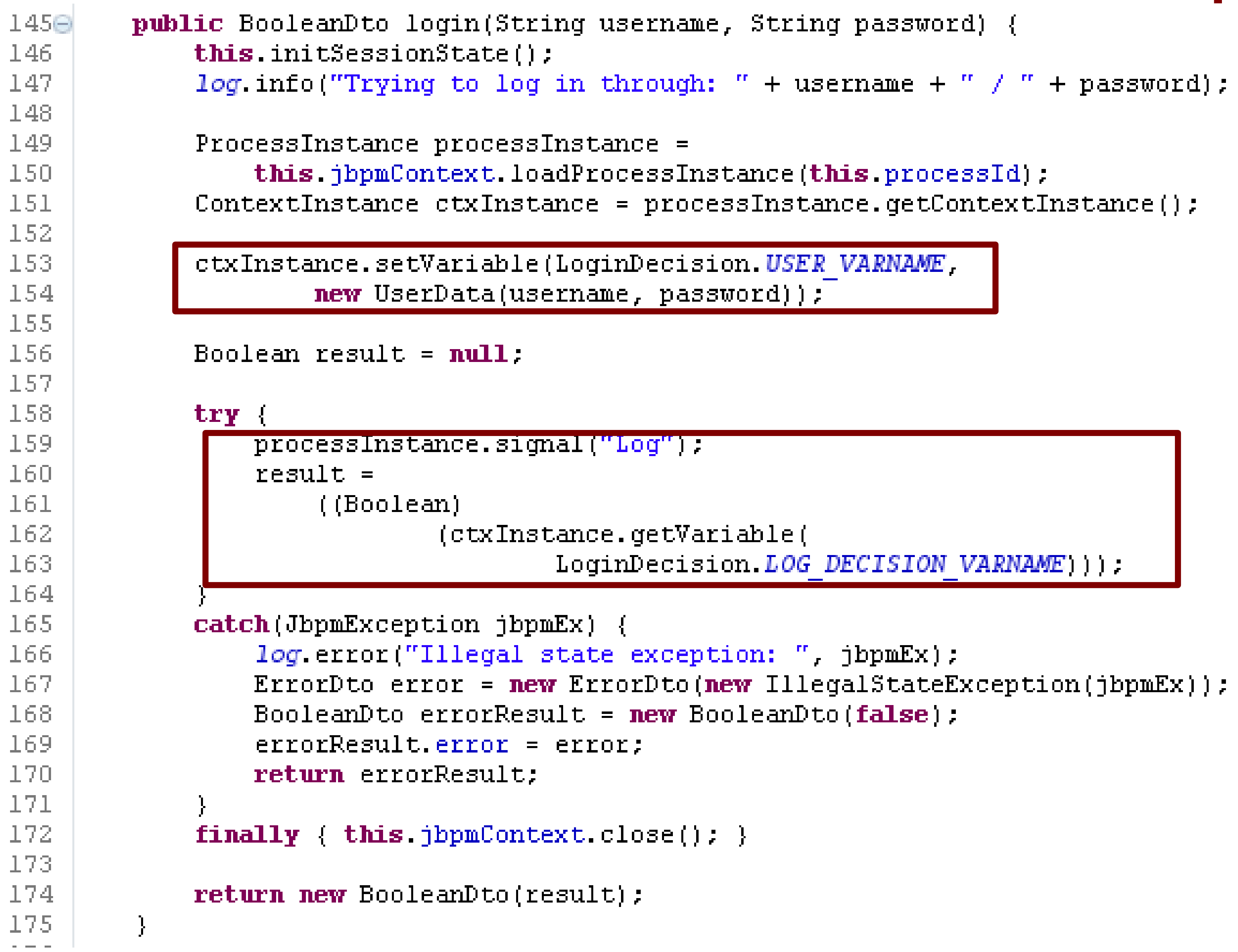
## Web Service / jBPM interaction

- Differently from BPEL, there is no information about variables to update and parameters to call inside the process descriptors
  - **Information hiding**
- Actions and Web Service Endpoints don't share any reference, but the so called ***jBPM Context***
  - **Encapsulation**
- They access temporary runtime variables provided by the *jBPM Context*
  - jBPM framework guarantees persistence, coherence, concurrency over process runtime variables

# Web Service / jBPM interaction

- Thus, Web Service operations' implementing code
  1. identifies the process related to the conversation
  2. invokes the *Jbpm Context* methods for activating proper transitions
  3. waits for the *Jbpm Context* to complete its business
     - registering the transition for updating the current state on the specified process
     - activating eventual decisions and actions
  4. returns the computations' results back to the client

# Some code: Web Methods impl.



```java
public BooleanDto login(String username, String password) {
    this.initSessionState();
    log.info("Trying to log in through: " + username + " / " + password);

    ProcessInstance processInstance =
        this.jbpmContext.loadProcessInstance(this.processId);
    ContextInstance ctxInstance = processInstance.getContextInstance();

    ctxInstance.setVariable(LoginDecision.USER_VARNAME,
            new UserData(username, password));

    Boolean result = null;

    try {
        processInstance.signal("Log");
        result =
            ((Boolean)
                    (ctxInstance.getVariable(
                            LoginDecision.LOG_DECISION_VARNAME)));
    }
    catch(JbpmException jbpmEx) {
        log.error("Illegal state exception: ", jbpmEx);
        ErrorDto error = new ErrorDto(new IllegalStateException(jbpmEx));
        BooleanDto errorResult = new BooleanDto(false);
        errorResult.error = error;
        return errorResult;
    }
    finally { this.jbpmContext.close(); }

    return new BooleanDto(result);
}
```

## An example: WSter

- A **simple application** using the framework applied to the example showed before
  - Divided into three logical units
    - physically deployed in three different archives, following the **MVC** architecture
      - A WAR with presentation modules (View)
        - JSP files, presentation logic, clients of the main Web Service
        - Conversational WS-Addressing *MessageHandler*s
      - A WAR with control modules (Control)
        - Service definition files
          - jPDL descriptor, message handlers stack descriptor, WSDL
        - Service implementation classes
          - Servlet, *ActionHandler*s, *DecisionHandler*s, *MessageHandler*s
        - Conversational WS-Addressing *MessageHandler*s
      - A JAR with persistence modules (Model)
        - EJB 3.0 - based Web Service (for remote access)
        - JPA-connected Database