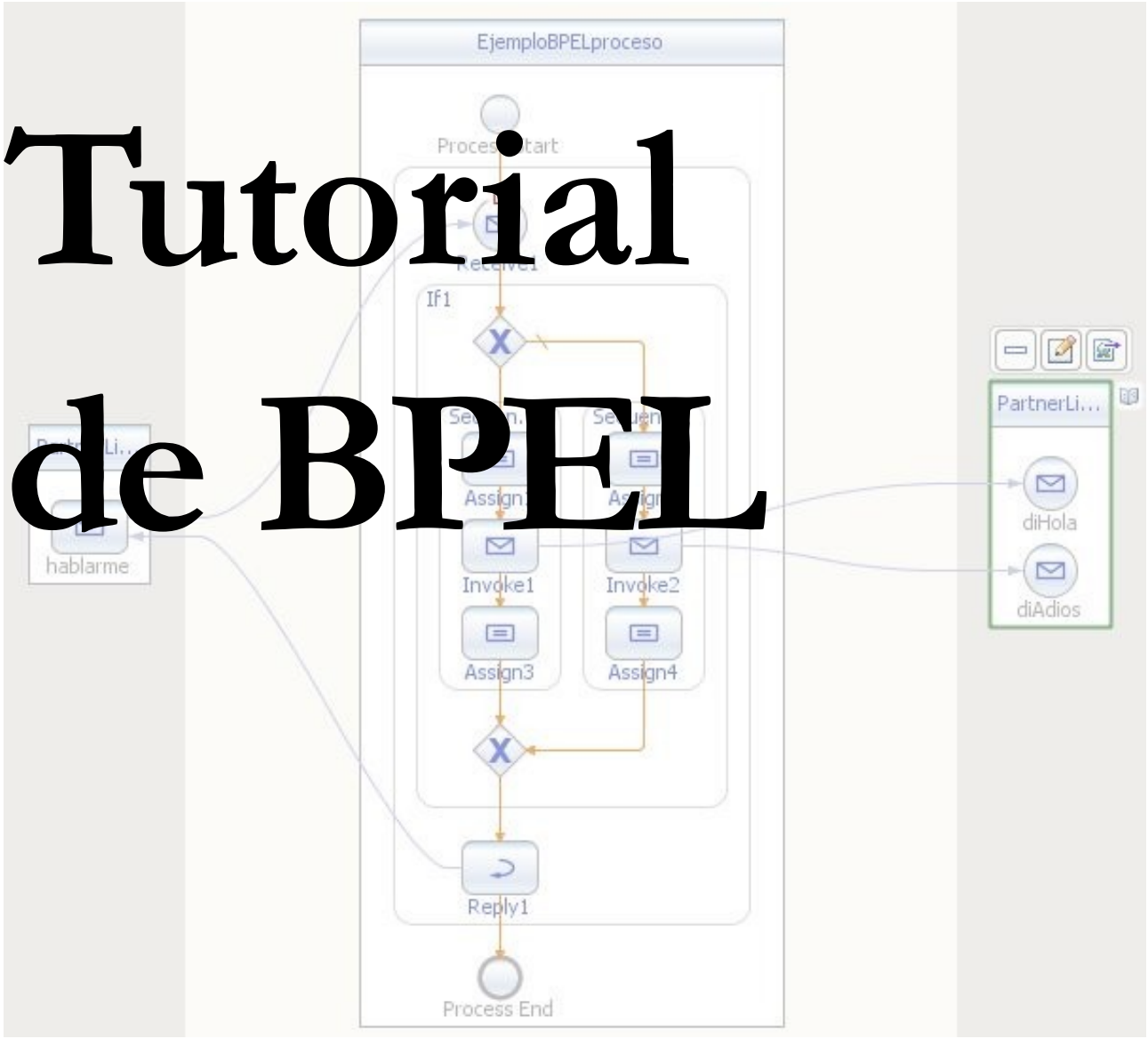
Tutorial de BPEL



Pablo García Sánchez
pgarcia@atc.ugr.es

Curso: Web 2.0. Arquitectura Orientada a Servicios en Java
Escuela de Posgrado
Febrero/Marzo de 2010

Introducción:

En este documento vamos a explicar pasito a pasito cómo crear un proceso BPEL simplón.
El resumen de los pasos sería:

1) Crear proceso BPEL
2) Crear Composite Application y meterle dentro el proceso
3) Desplegar y Testear

URLs de servicios a atacar:

http://evorq.ugr.es:8080/evorqServices/ejemploServiceService?wsdl
http://evorq.ugr.es:8080/evorqLogin/loginServiceService?wsdl

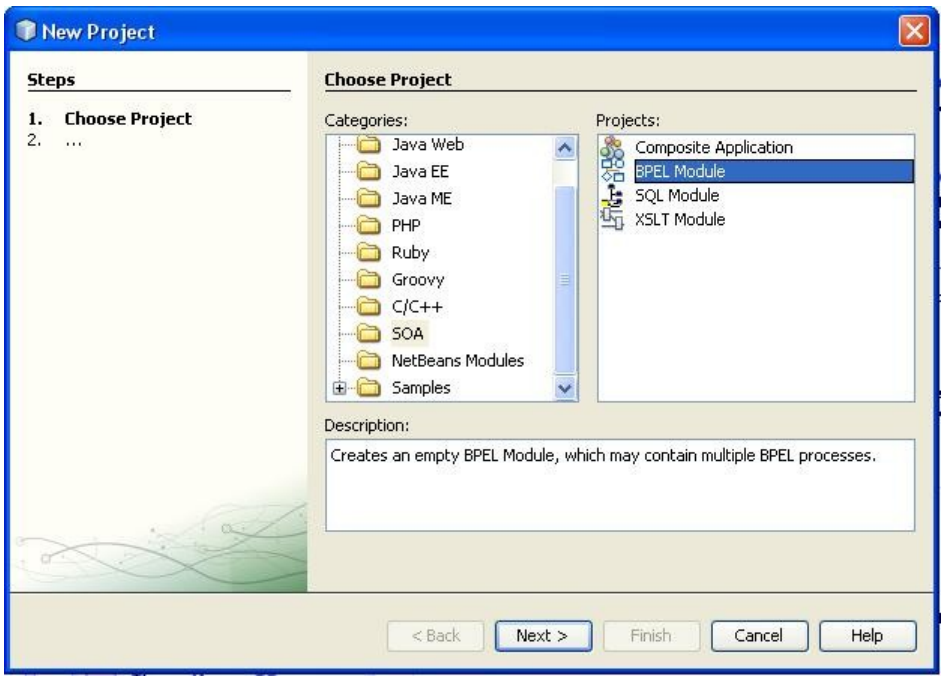
Ejemplos a realizar:

1) SynchronousSample
2) Ejemplo del tutorial (el que dice "Hola")
3) Ejemplo avanzado del tutorial (llama a los WS de la primera URL)
4) Ejercicio: cambiar el if por un *flow* y arreglar lo que pasa
5) Ejercicio: llamar a la operación *getTicket* de la segunda URL con datos correctos o incorrectos. Añadir manejador de faltas.
6) Llamar a la operación *diHora* con el ticket obtenido.

1. Crear Proceso BPEL

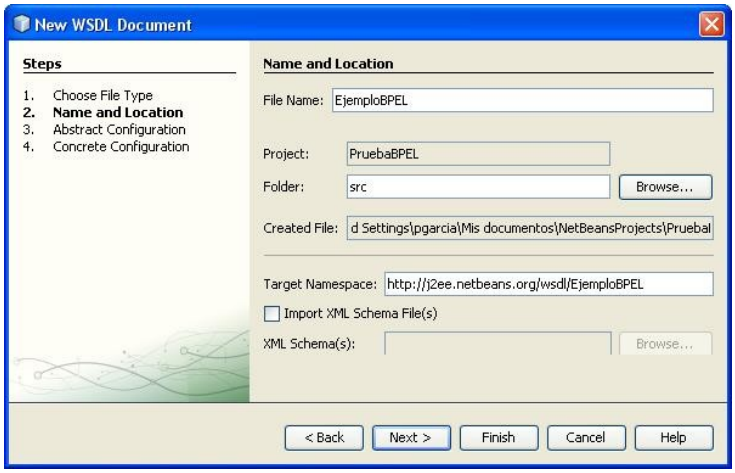
1) Crear Nuevo Proyecto:

Simplemente vamos a `New Project`, elegimos `SOA` y luego `BPEL Module`.



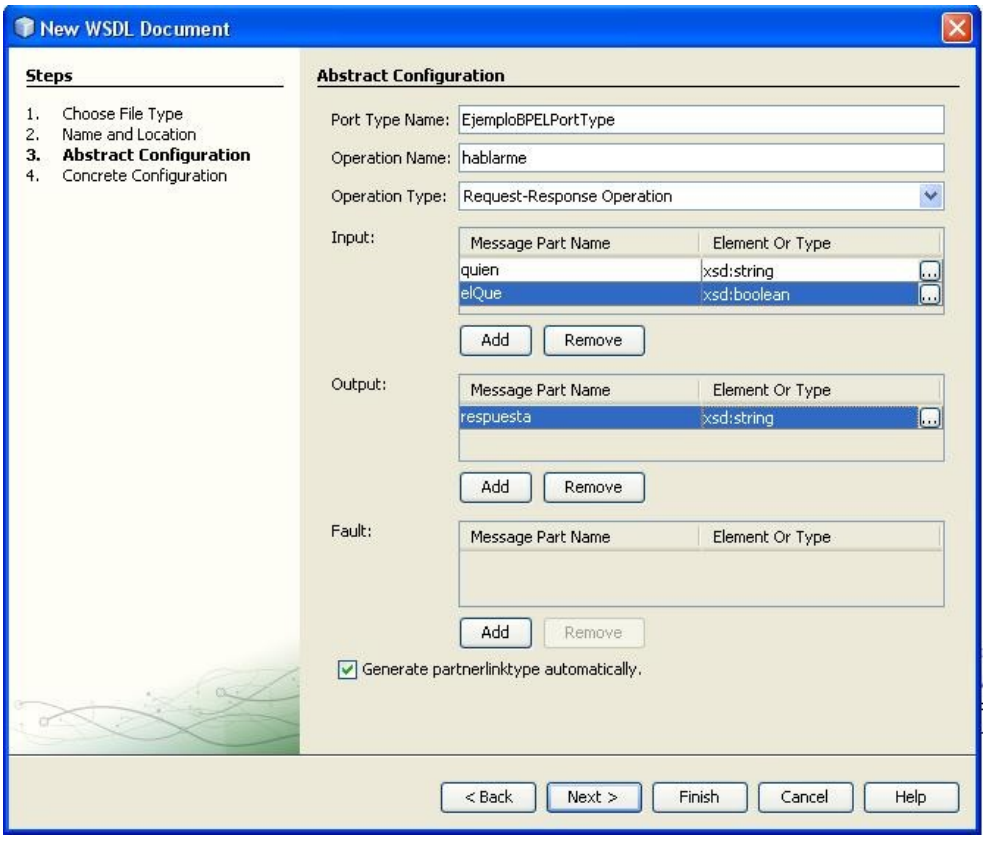
2) Crear WSDL

En el proyecto recién creado pulsamos en el menú contextual `New->WSDL Document`, apareciendo el siguiente asistente:

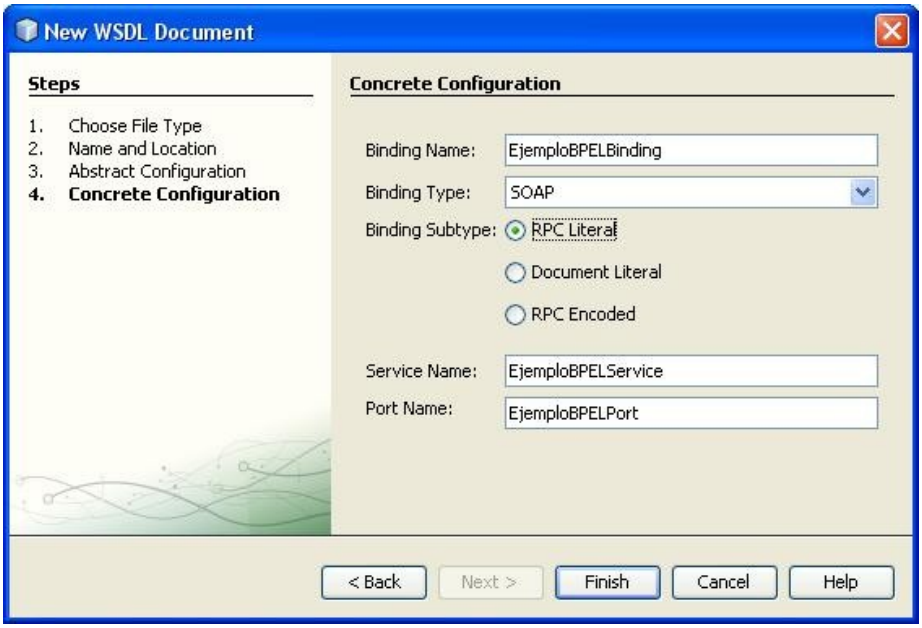


Elegimos un nombre para el WSDL y el *target namespace* (el que sale automático está bien, lo importante es que no se repita en otros proyectos para que no se líe la cosa).

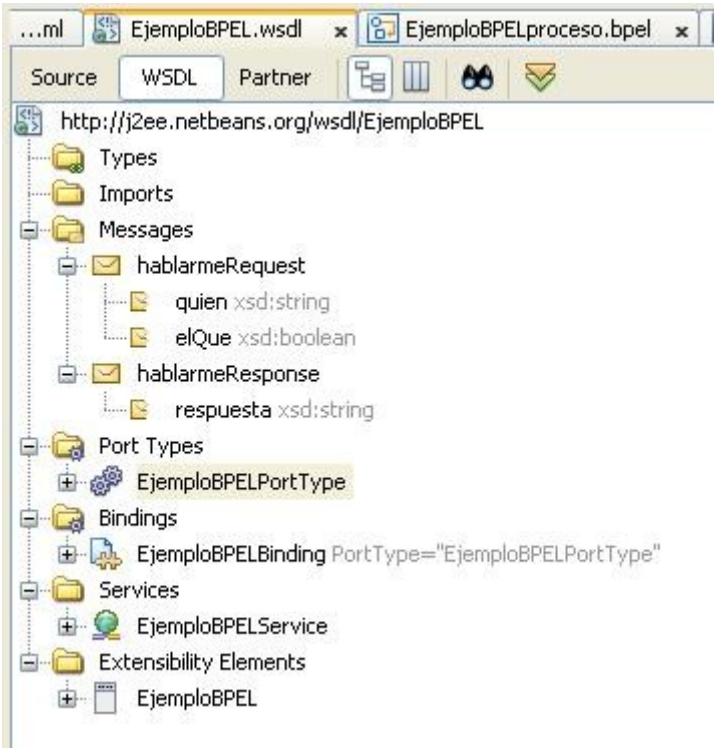
Creamos una operación de nombre "hablarme" con un *string* y *booleano* de parámetros de entrada y un string de salida, como vemos en la siguiente imagen:



En la siguiente ventana dejamos todo como está (ya depende el tema del proyecto en el que estemos trabajando!).

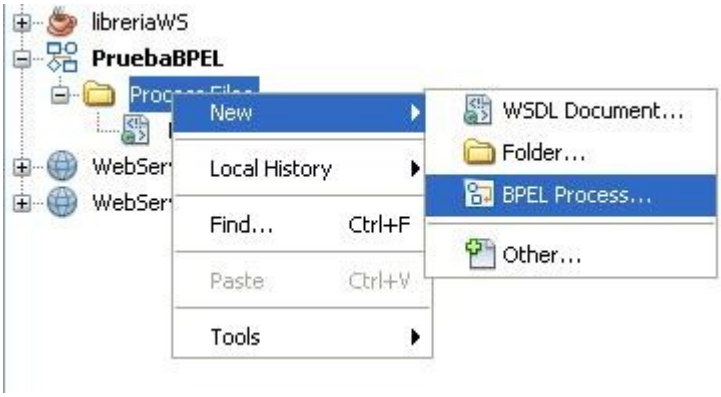


Podemos ver el resultado de nuestro WSDL recién generado, que tendrá esta pinta más o menos:



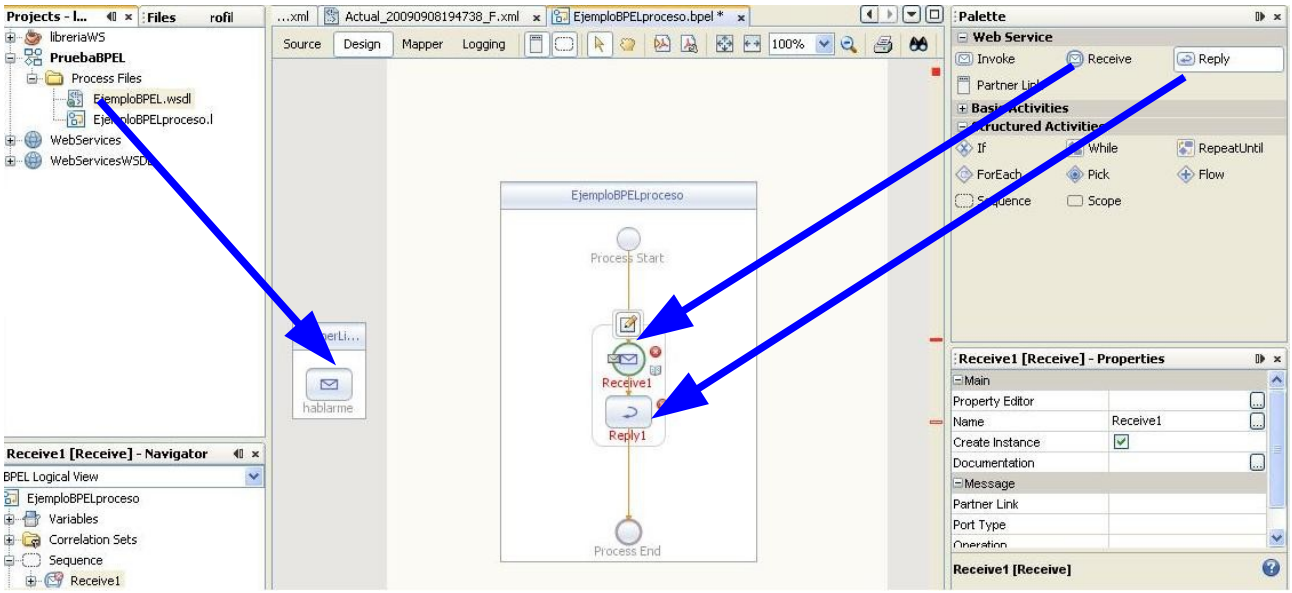
2) Crear proceso BPEL

Al igual que al crear el WSDL anterior seleccionamos en este caso `New-> BPEL Process` como se ve en la siguiente imagen:

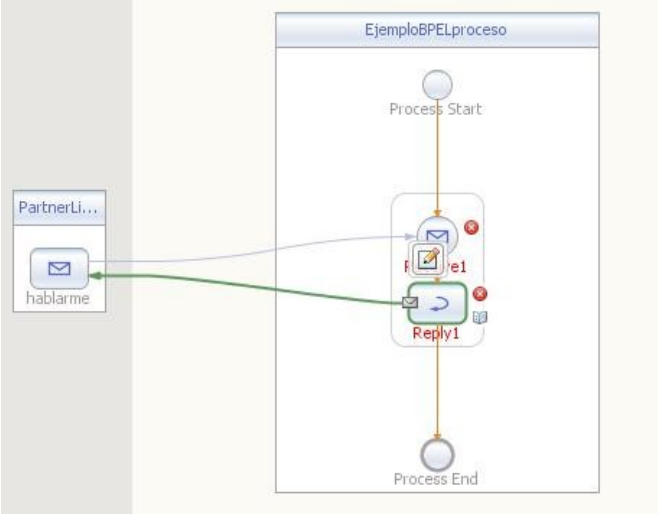


Ahora:

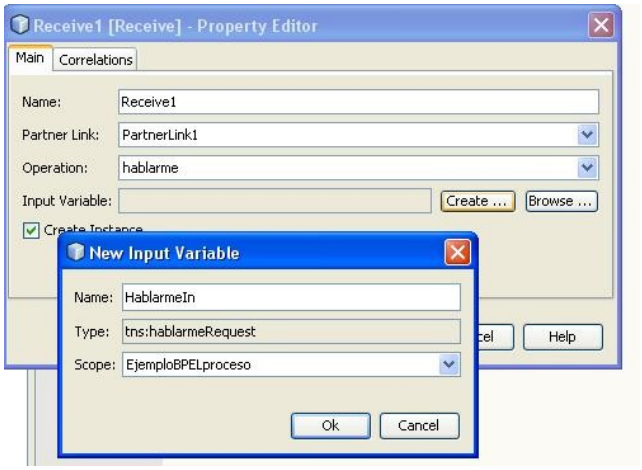
1) **Arrastramos** el WSDL anterior a la izquierda del proceso BPEL, será el WSDL que describe al proceso.
2) **Arrastramos** las actividades *Receive* y *Reply* al proceso. Serán la entrada y salida del mismo.



Unimos las actividades *Assign* y *Reply* a la operación WSDL que habíamos creado antes arrastrando los sobrecillos:

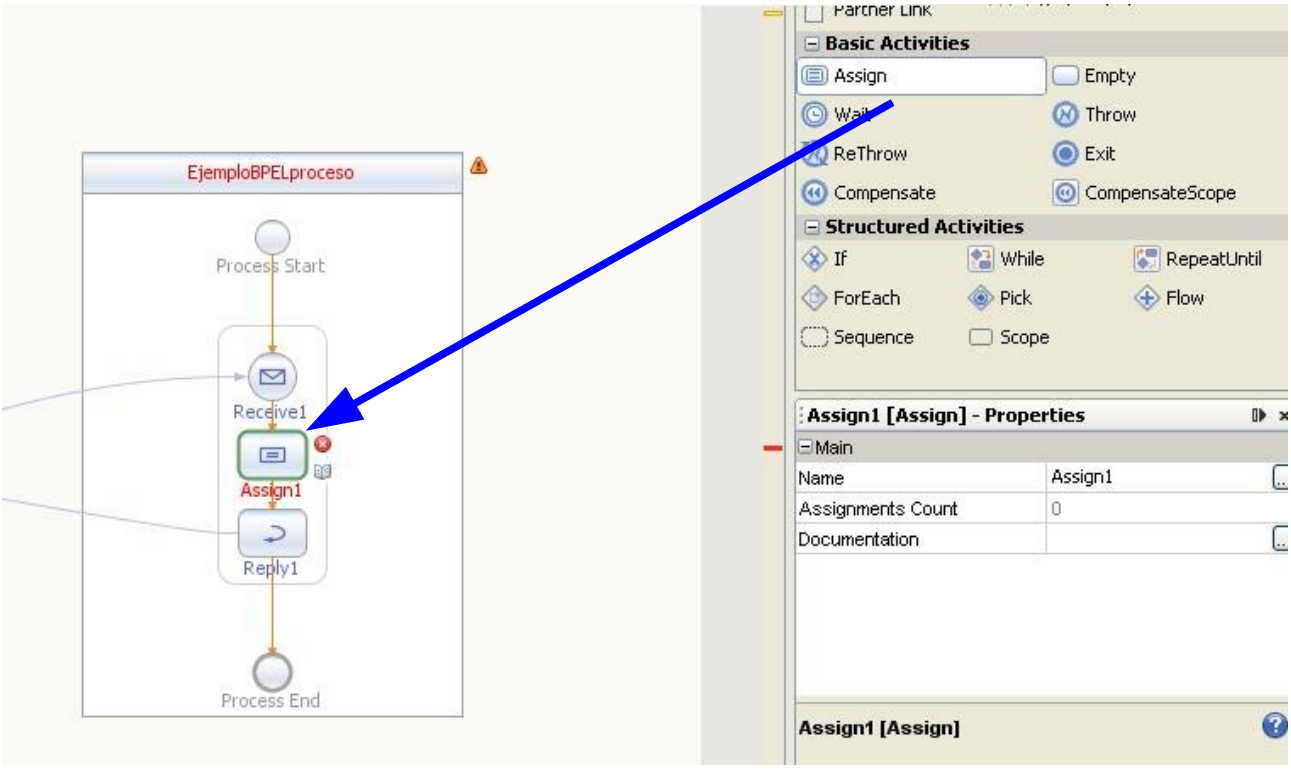


Como podemos ver, sigue dando errores (los *receives* y *replies* necesitan variables que recibir o enviar, claro). Hacemos doble clic en cada uno de ellos y le damos a "`Create`":



Como puede verse la nomenclatura por defecto es *OperacionIn* u *OperacionOut*. Es recomendable seguirla. Hay que hacer lo mismo en el reply, ojo!

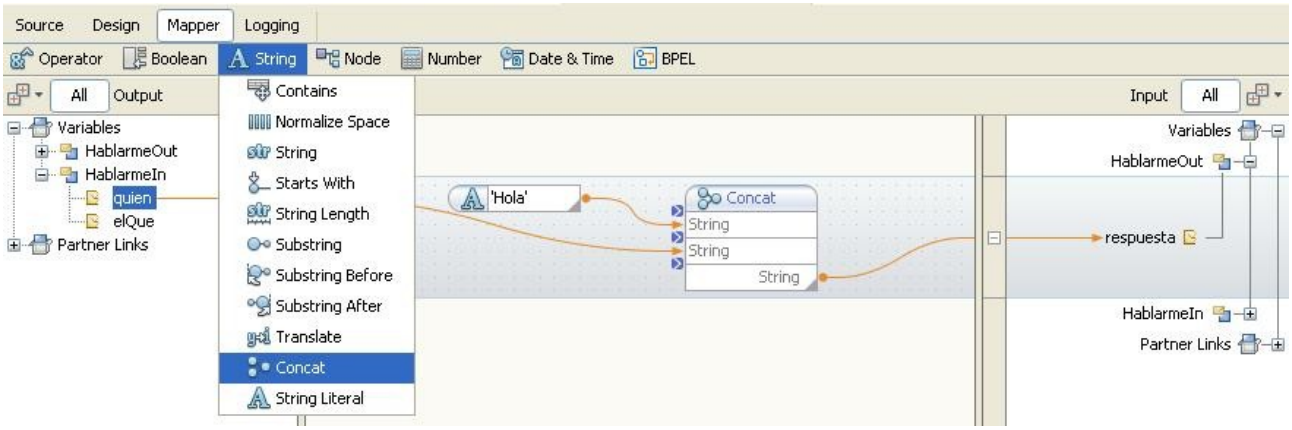
Ahora tenemos que hacer algo con las variables que hemos creado, que si no menuda gracia tendría BPEL. Por ejemplo vamos a probar la actividad *Assign*, arrastrándola al proceso.



En este instante uno empieza a darse cuenta de que las aspas rojas son errores, por lo que tenemos que deshacernos de ellas. Así hacemos doble clic en la actividad "*Assign*" para hacer la asignación, apareciendo la siguiente ventana: El mapeador (*mapper*).

Vamos a añadir un "Hola" a la variable que recibamos de entrada y devolver la cadena resultante. Para ello seguimos los siguientes pasos:

1) Crear una operación *Concat* del menú String (como muestra la imagen siguiente).
2) De ese menú creamos un String Literal con la palabra *"Hola"*.
3) Arrastrar la flecha del "Hola" al primer hueco del *Concat* y la parte "*quien*" de la variable HablarmeIn a la segunda parte del *Concat* (el resto de huecos se pueden rellenar con más *strings* si quisieramos).
4) Arrastramos la salida a la parte "*respuesta*" de la variable "*HablarmeOut*"



OJO! Antes de nada tenemos que fijarnos que la variable que queremos asignar tiene que estar seleccionada (en este caso es *"respuesta"*).

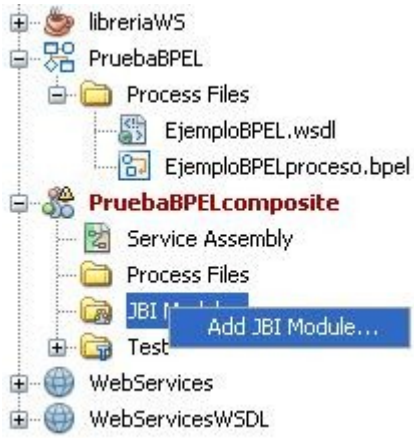
Y listo, tenemos un proceso BPEL que implementa una operación de un WSDL recibe un String y un Booleano y devuelve "Hola"+String. No es muy impresionante pero por algo se empieza.

2) Crear Composite Application

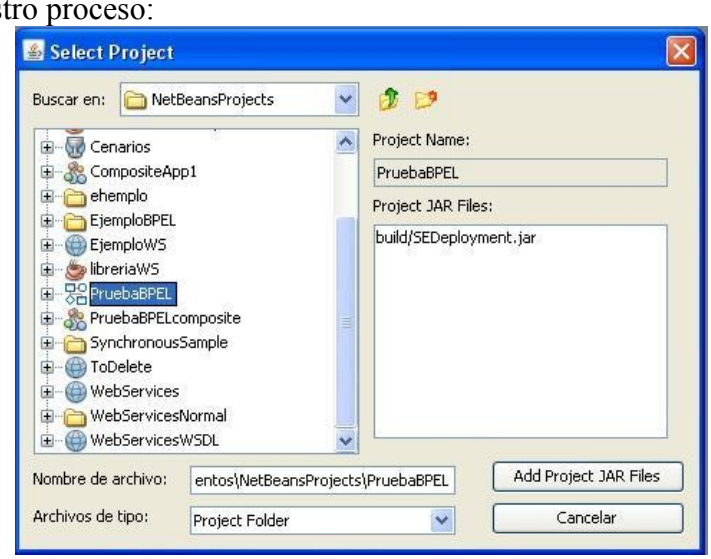
Como hemos visto anteriormente en el curso JBI se basa en las *Composite Applications* para desplegar componentes en Glassfish, así que tenemos que crear una para poder desplegarla en nuestro servidor.

Vamos a `New->Project->SOA->Composite Application` y la llamamos en un alarde de originalidad *PruebaBPELcomposite.*

Después añadimos el proceso BPEL con "`Add JBI Module`":

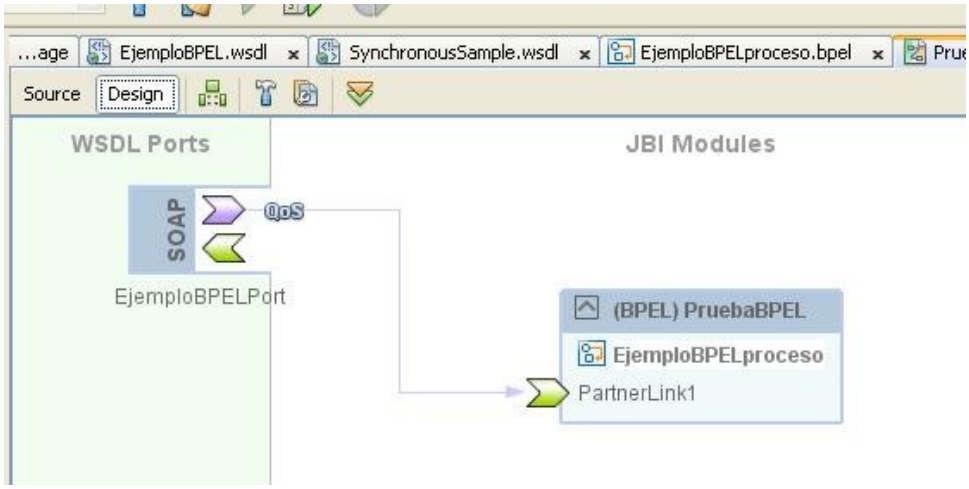


Y seleccionamos nuestro proceso:



A continuación hacemos "*Clean and Build*" en el menú contextual del proyecto y la aplicación se

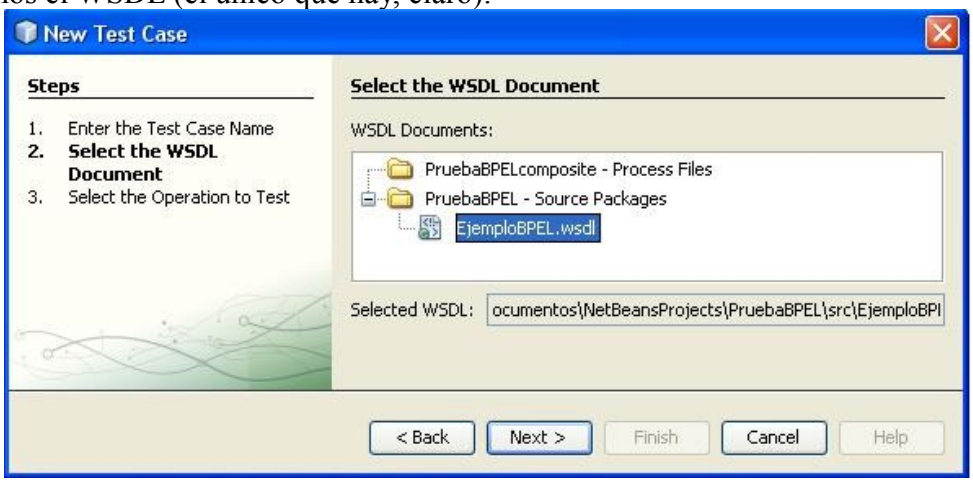
refrescará:



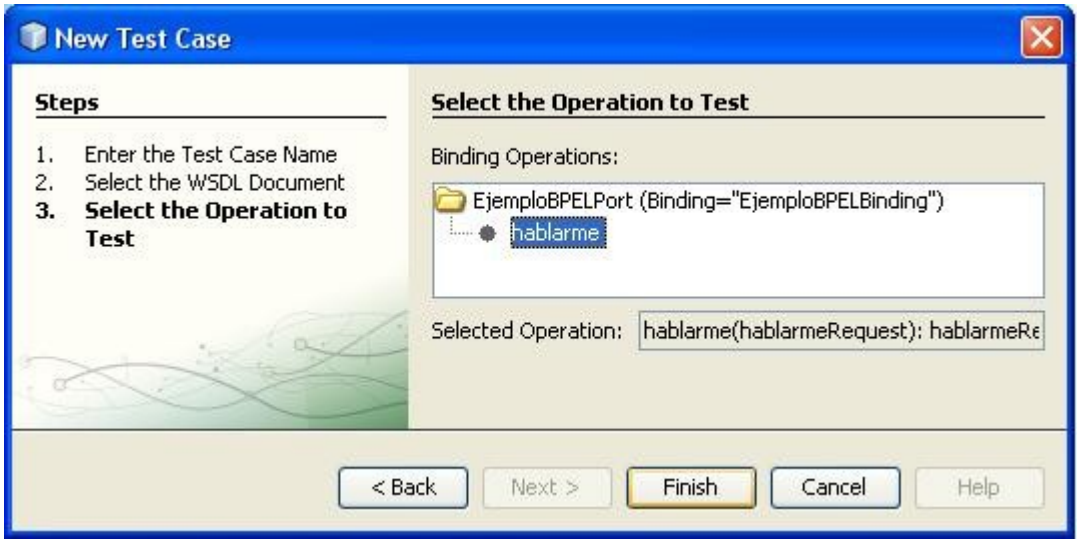
3) Desplegar y testear la aplicación

Dentro de la carpeta "Test" añadimos un nuevo *Test Case*.

Seleccionamos el WSDL (el único que hay, claro):

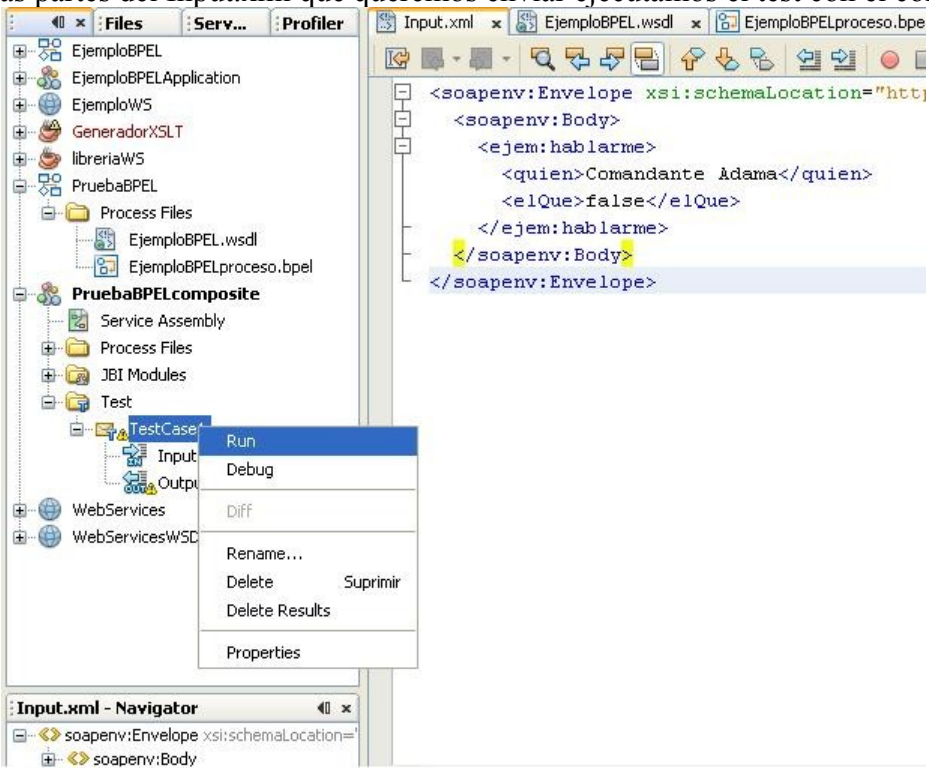


Y seleccionamos la operación (la única que hay, claro):



Se nos ha creado el test con un Input.xml y un Output.xml. El primero es la llamada SOAP al WSDL y el segundo está vacío. Un test se considera correcto si la salida del test coincide con el fichero Output.xml. Como está vacío, aunque la cosa funcione no van a ser iguales, por lo que el test fallará a propósito.

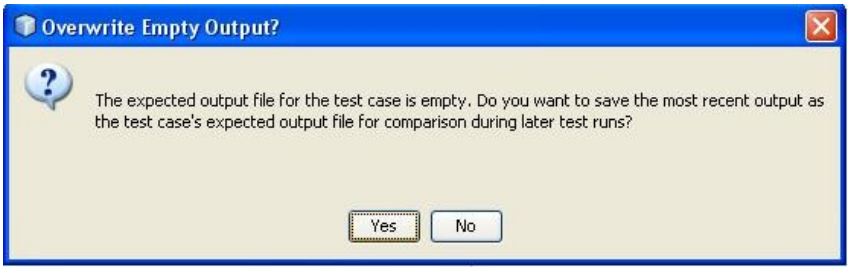
Tras rellenar las partes del Input.xml que queremos enviar ejecutamos el test con el comando *Run*:



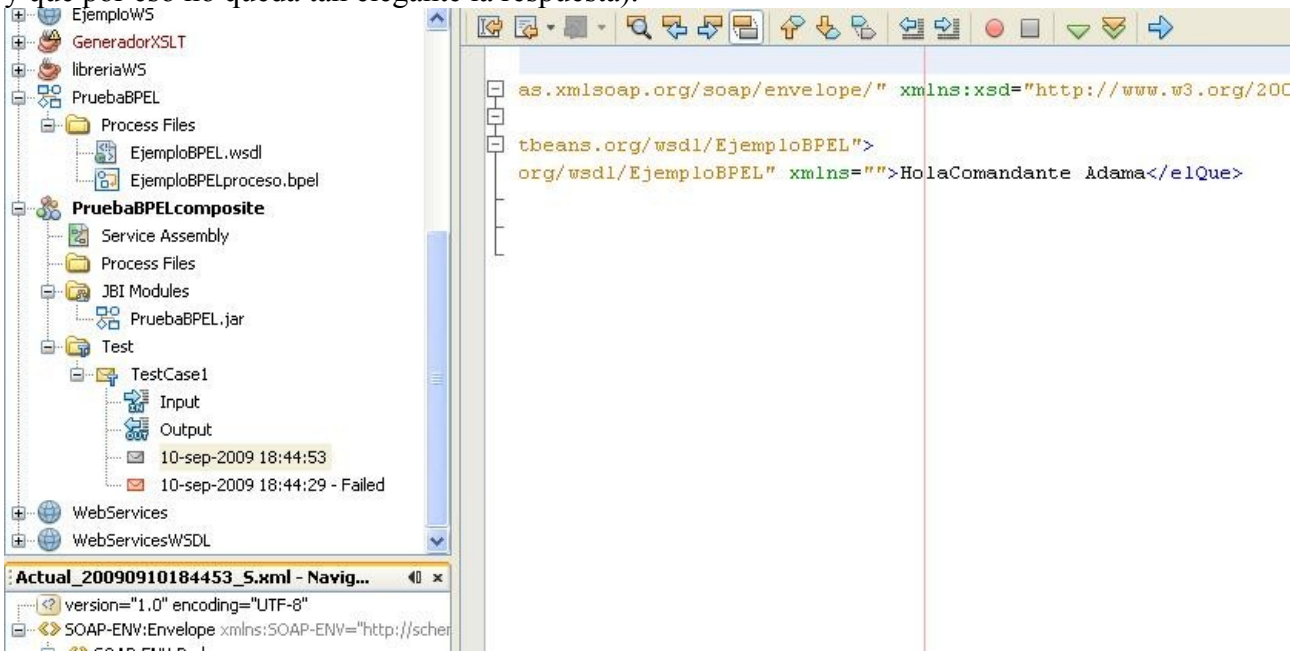
Si no se ha hecho antes la aplicación se desplegará en Glassfish y tras un rato hará la llamada SOAP al servicio del WSDL del BPEL.

Como hemos dicho la primera vez que hagamos tests NetBeans nos indicará que se está comparando con un Output.xml vacío, así que nos propone que el que acabamos de recibir sea el

bueno. Le decimos que sí, que vale.



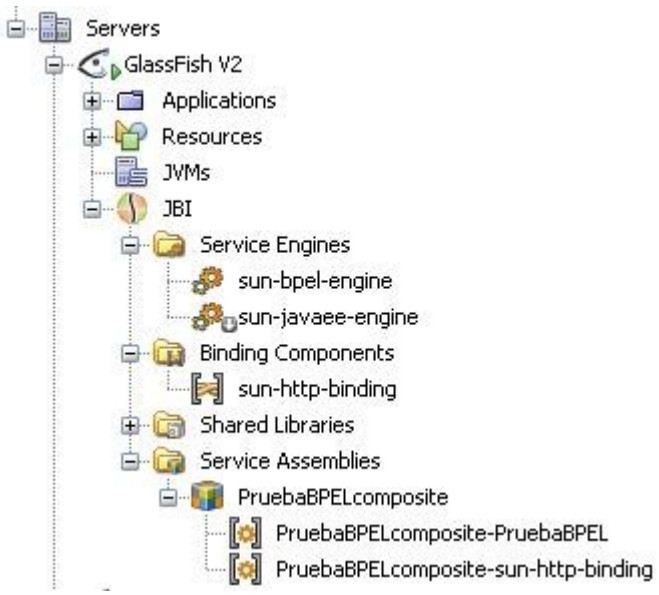
Para ver el resultado de los tests hacemos doble clic en el sobre que los representa. Vemos que la cosa ha ido bien (obviando que no he puesto un espacio al lado del "Hola" de la asignación anterior, y que por eso no queda tan elegante la respuesta).



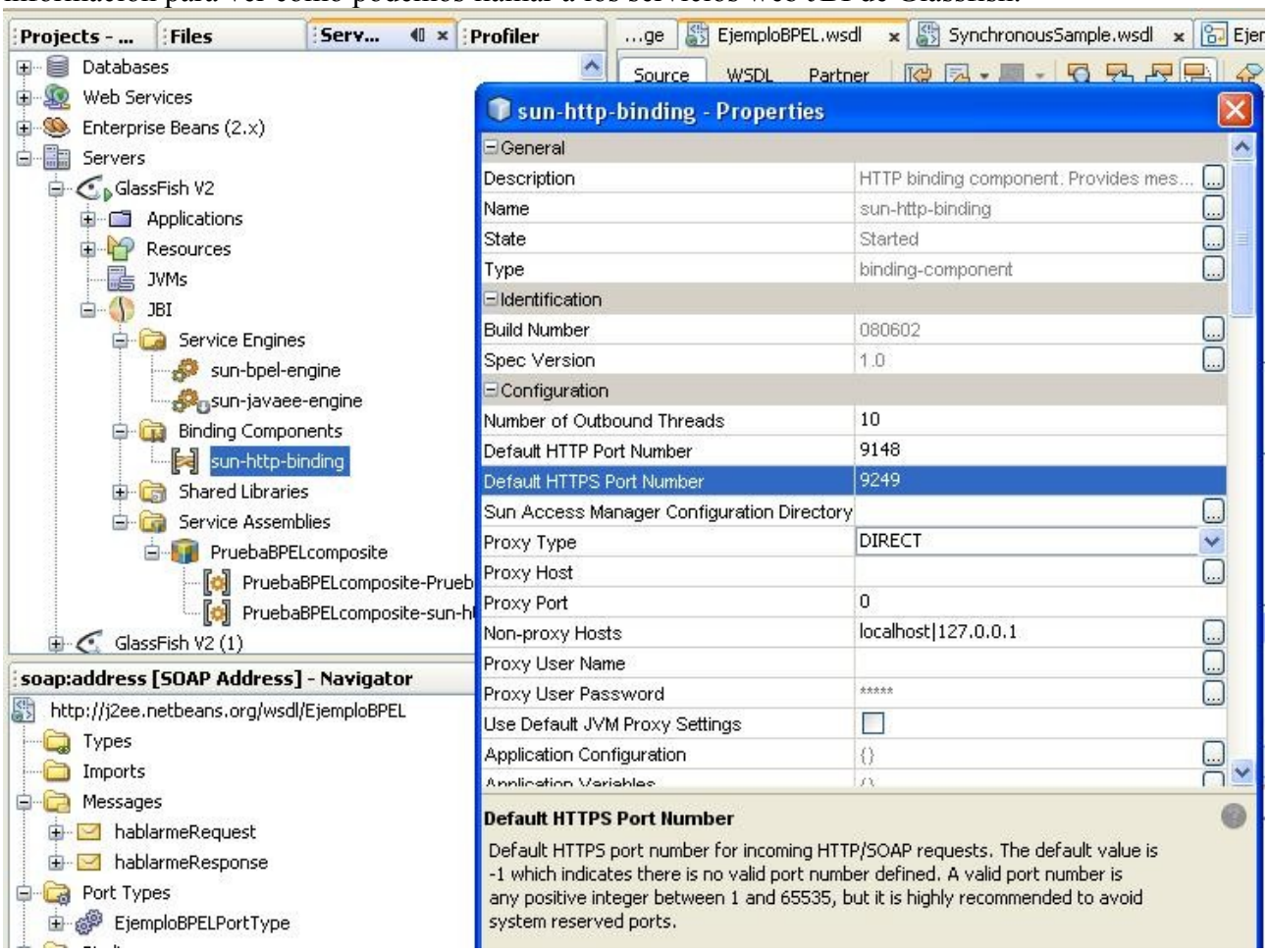
Gestión de la Composite Application

Hemos dicho que se ha desplegado la aplicación en Glassfish y que hemos llamado a un WS a una dirección que no sabemos aún. ¿Cómo podemos quitar la aplicación de Glassfish y saber más cosas sobre ella?

Simplemente nos vamos a la pestaña *Services* (al lado de la de *Projects*). Vemos que el servidor GlassFish está arrancado (tiene un play al lado del pescao). Si desplegamos el nodo JBI podemos ver los *Services Engines* que tenemos instalados (en este caso el de BPEL y el de JavaEE, pero podemos añadir muchos más) y que tenemos desplegada una *Composite Application* llamada *PruebaBPELcomposite*. ¡La nuestra!



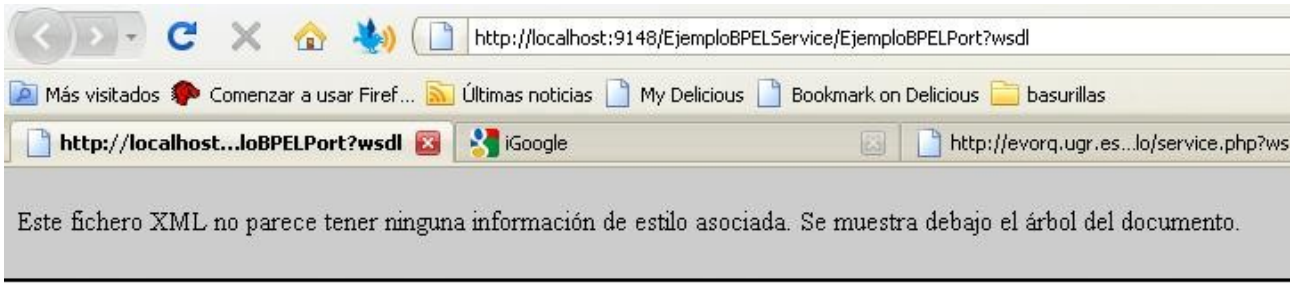
Vemos que tenemos un Binding Component para comunicarse por HTTP (podríamos instalar otro para SMTP, JMS o incluso Bluetooth si quisieramos desarrollarlo!), pero vamos a ver su información para ver cómo podemos llamar a los servicios web JBI de Glassfish.



Vemos que el puerto SOAP que usa es el 9148, así que ya sabemos donde mirar. Comprobamos la url del servicio en el propio archivo WSDL del proyecto:

```
<service name="EjemploBPELService">
        <port name="EjemploBPELPort" binding="tns:EjemploBPELBinding">
            <soap:address location="http://localhost:$
{HttpDefaultPort}/EjemploBPELService/EjemploBPELPort"/>
        </port>
    </service>
```

Y sustituimos el puerto por el que hemos visto en la imagen anterior, el 9148.




```
- <wsdl:definitions name="EjemploBPEL" targetNamespace="http://j2ee.netbeans.org/wsdl/EjemploBPEL">
    <wsdl:types> </wsdl:types>
  - <wsdl:message name="hablarmeResponse">
      <wsdl:part name="respuesta" type="xsd:string"> </wsdl:part>
    </wsdl:message>
  - <wsdl:message name="hablarmeRequest">
      <wsdl:part name="quien" type="xsd:string"> </wsdl:part>
      <wsdl:part name="elQue" type="xsd:boolean"> </wsdl:part>
    </wsdl:message>
  - <wsdl:portType name="EjemploBPELPortType">
    - <wsdl:operation name="hablarme">
        <wsdl:input name="input1" message="tns:hablarmeRequest"> </wsdl:input>
        <wsdl:output name="output1" message="tns:hablarmeResponse"> </wsdl:output>
      </wsdl:operation>
    </wsdl:portType>
  - <wsdl:binding name="EjemploBPELBinding" type="tns:EjemploBPELPortType">
      <soap:binding style="rpc" transport="http://schemas.xmlsoap.org/soap/http"/>
    - <wsdl:operation name="hablarme">
        <soap:operation/>
      - <wsdl:input name="input1">
          <soap:body use="literal" namespace="http://j2ee.netbeans.org/wsdl/EjemploBPEL"/>
        </wsdl:input>
      - <wsdl:output name="output1">
          <soap:body use="literal" namespace="http://j2ee.netbeans.org/wsdl/EjemploBPEL"/>
        </wsdl:output>
      </wsdl:operation>
    </wsdl:binding>
  - <wsdl:service name="EjemploBPELService">
    - <wsdl:port name="EjemploBPELPort" binding="tns:EjemploBPELBinding">
        <soap:address location="http://150.214.221.124:9148/EjemploBPELService/EjemploBPELPort"/>
      </wsdl:port>
    </wsdl:service>
  - <plnk:partnerLinkType name="EjemploBPEL">
    - <!--
        A partner link type is automatically generated when a new port type is added. Partner link typ
        In a BPEL process, a partner link represents the interaction between the BPEL process and a par
        A partner link type characterizes the conversational relationship between two services. The par
      -->
      <plnk:role name="EjemploBPELPortTypeRole" portType="tns:EjemploBPELPortType"/>
    </plnk:partnerLinkType>
  </wsdl:definitions>
```

Fíjate como actualiza automáticamente la *location* del servicio con mi IP (150.214.221.124:9148)

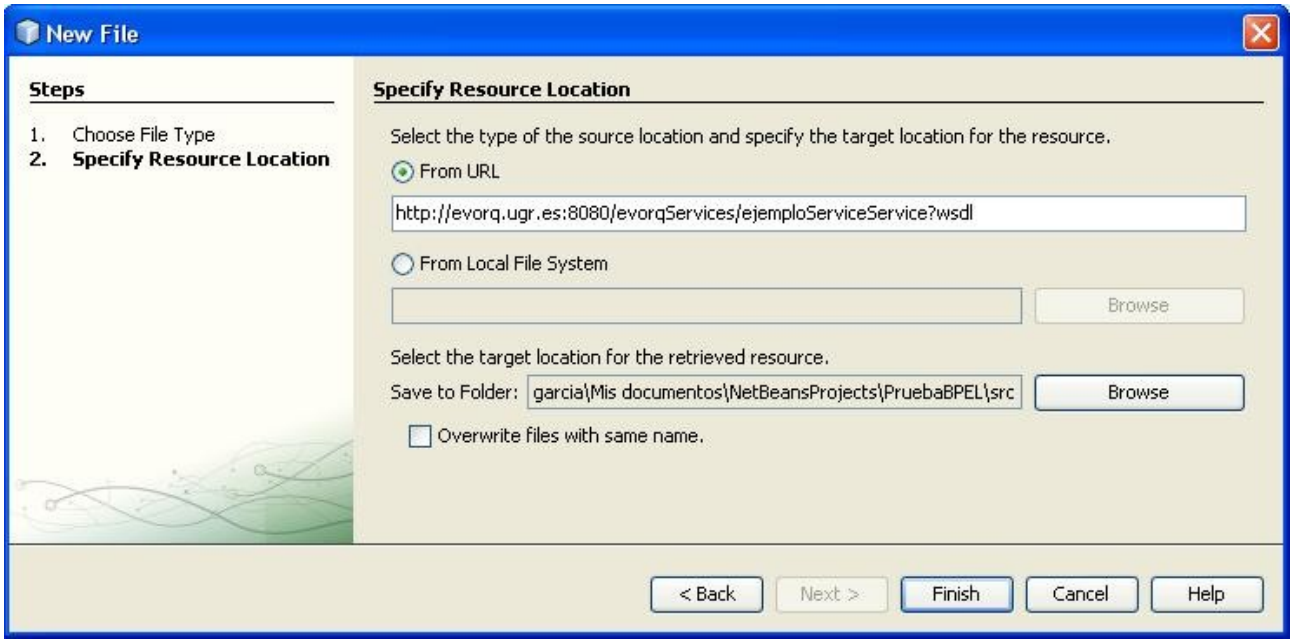
Podemos testearlo con SoapUI utilizando este WSDL:

También funciona!

SEGUNDA PARTE: Orquestando Servicios

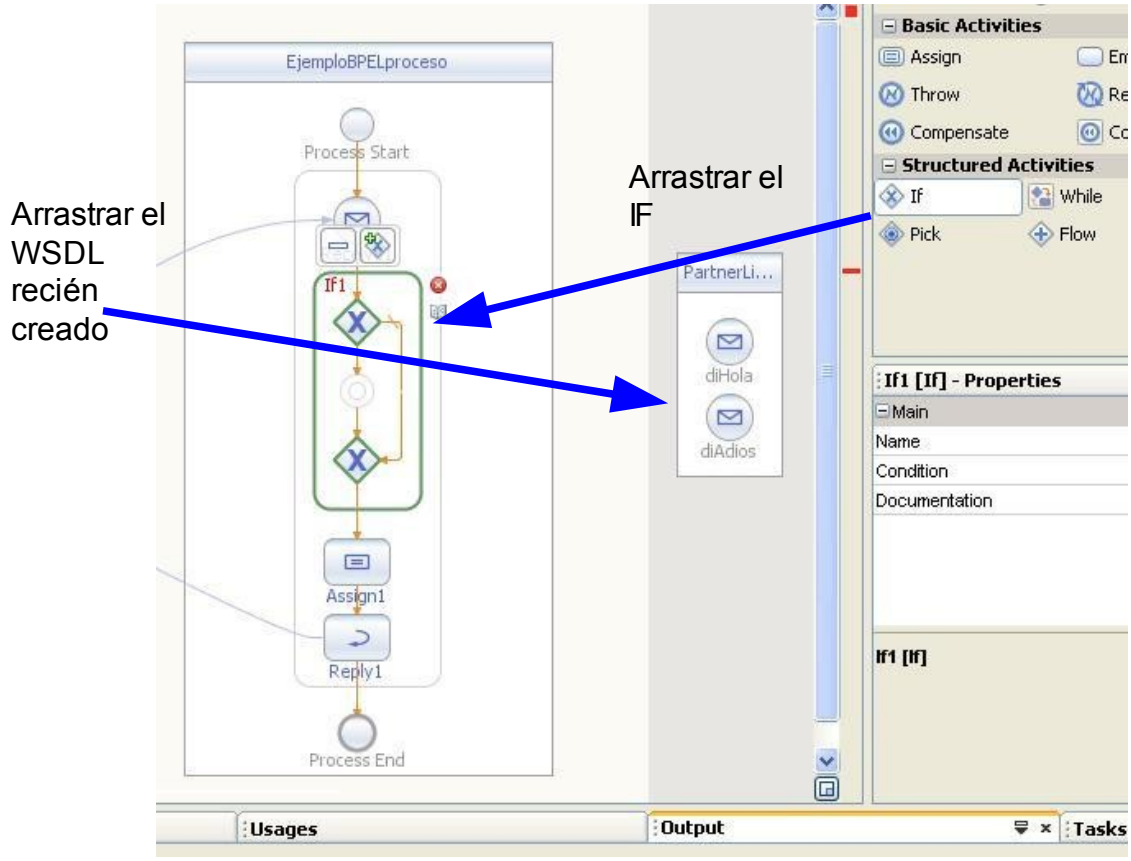
Bueno, pero la gracia de BPEL es orquestar servicios, ¿no? Vamos a añadirle por tanto alguna funcionalidad interesante a nuestra aplicación.

Añadimos un WSDL externo con los servicios que vamos a utilizar, desde *New File->External WSDL Document*:

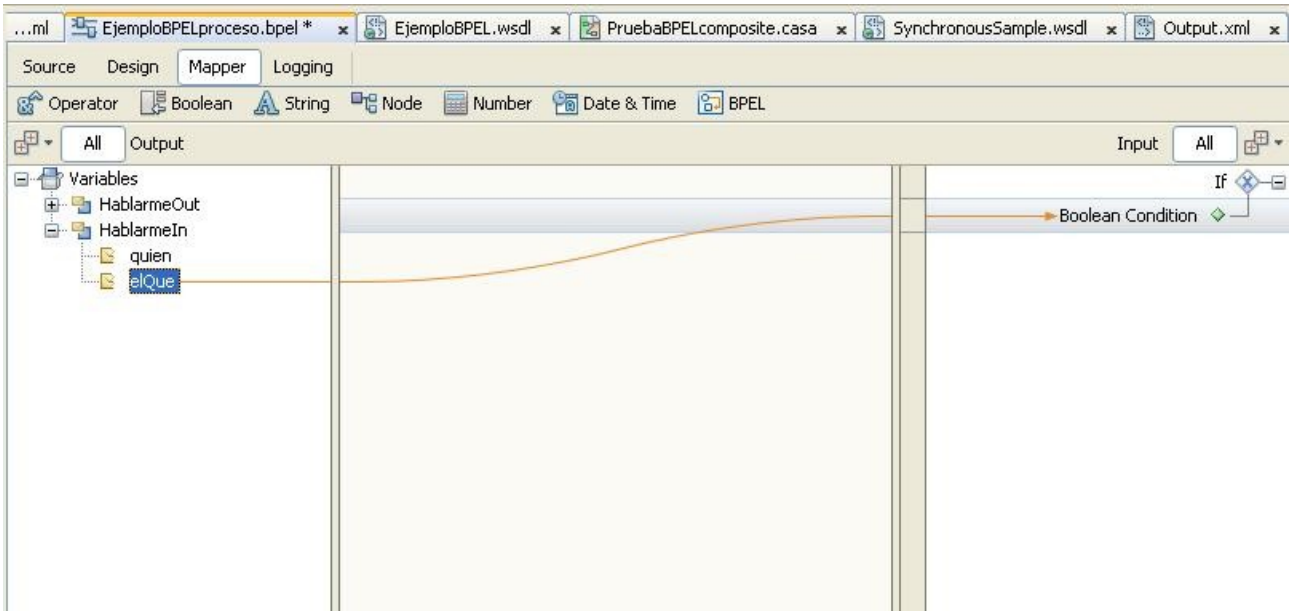


Ahora **arrastramos** el WSDL desde el proyecto a la parte **derecha** del proceso. Ya podemos ver las operaciones a las que el BPEL puede llamar.

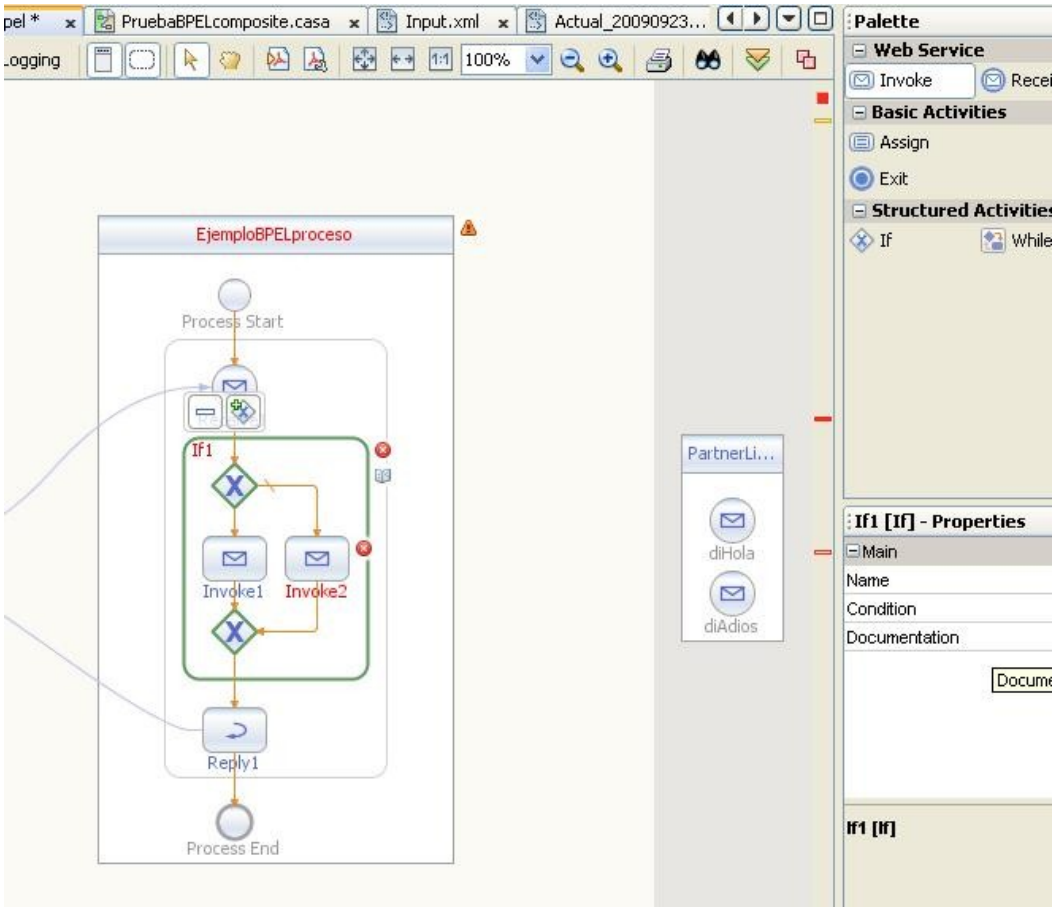
Vamos a probar también a llamar a una de las dos operaciones dependiendo del booleano de la entrada, así que añadimos un *if* arrastrándolo:



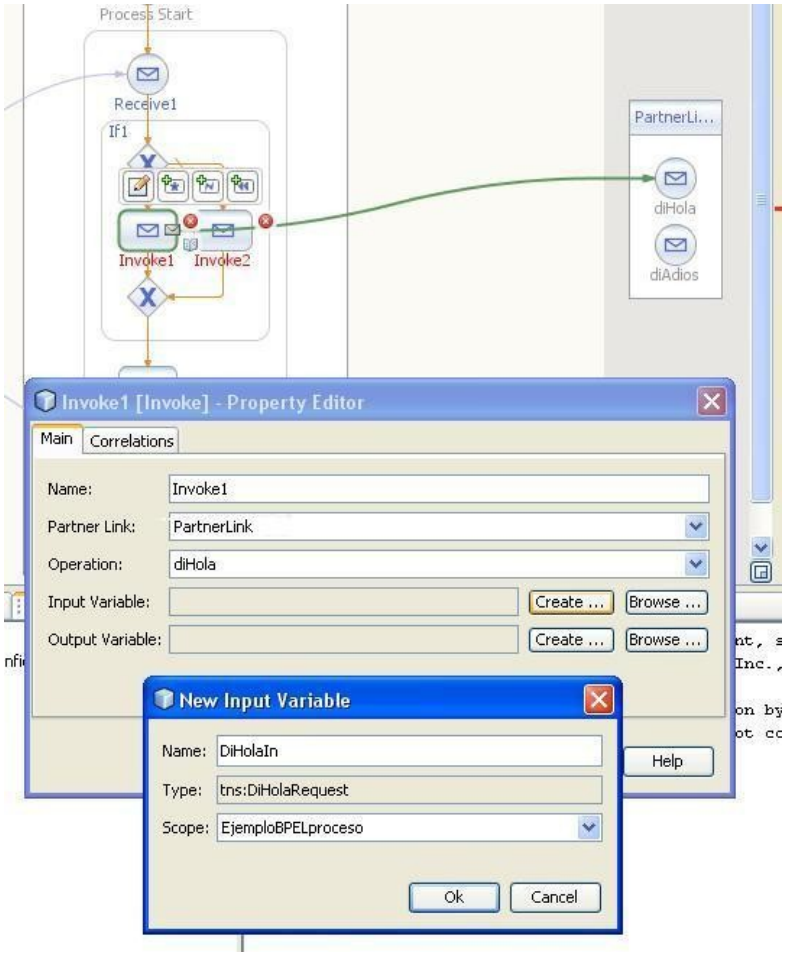
Y asignamos la variable de entrada del *if* a la condición en el *mapper* que se muestra al hacer doble clic en él:



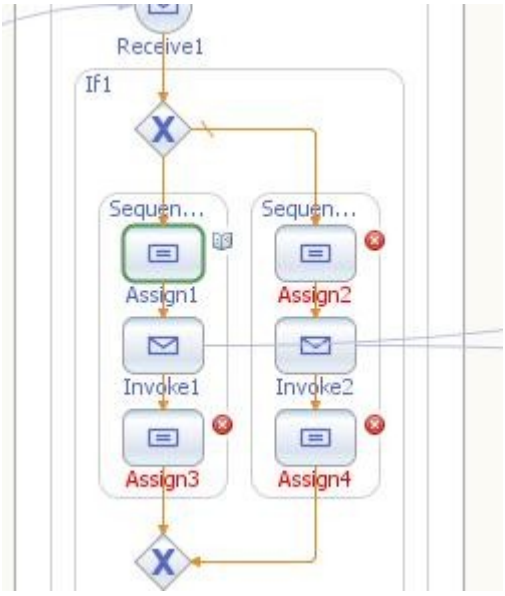
Después añadimos los *invokes*:



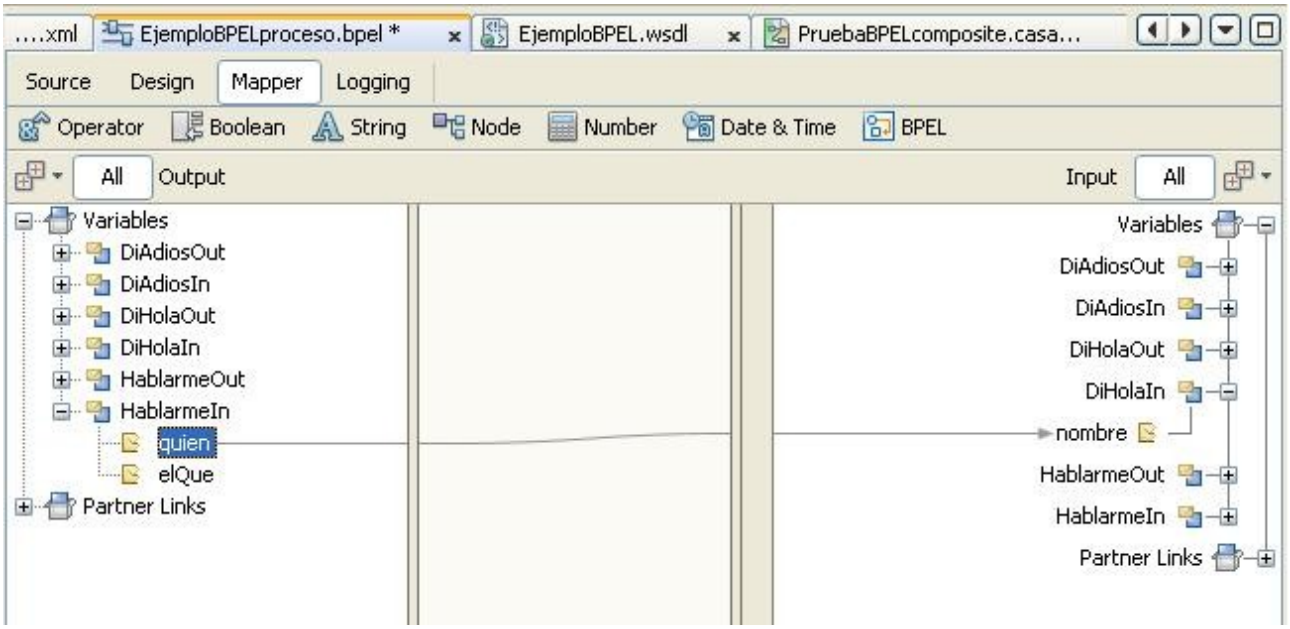
Y los unimos con las operaciones con los sobrecillos, creando las variables:



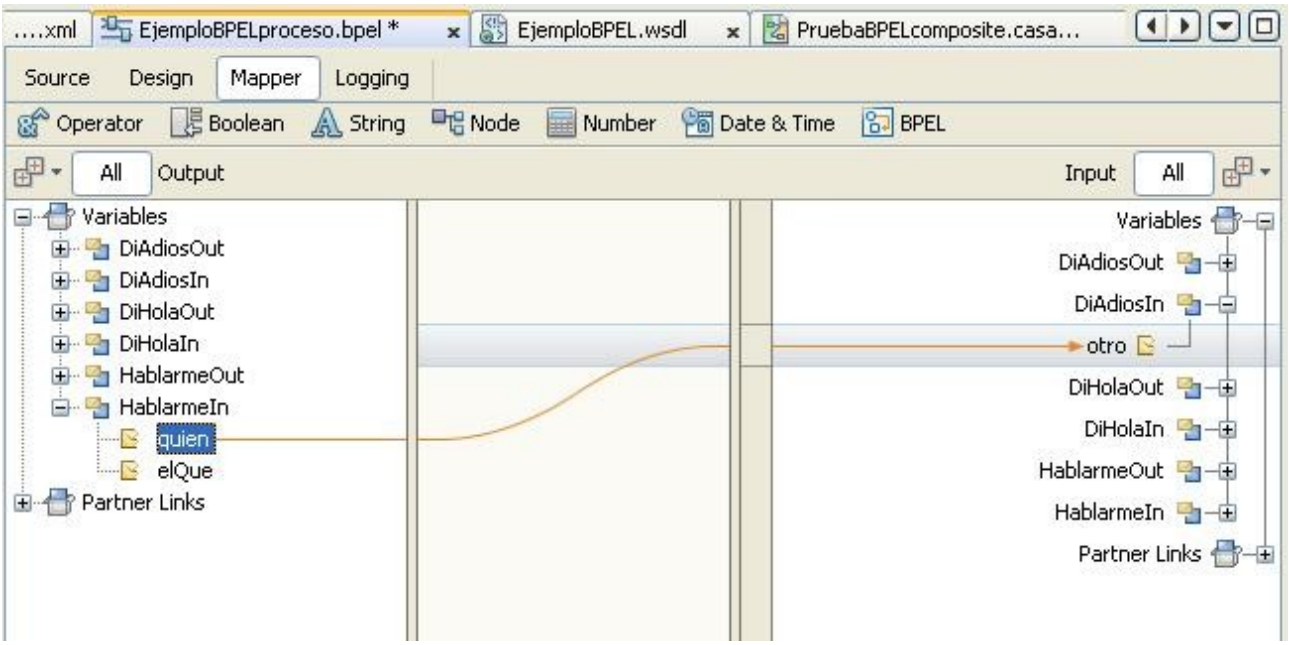
Añadimos assigns antes y después de los invokes



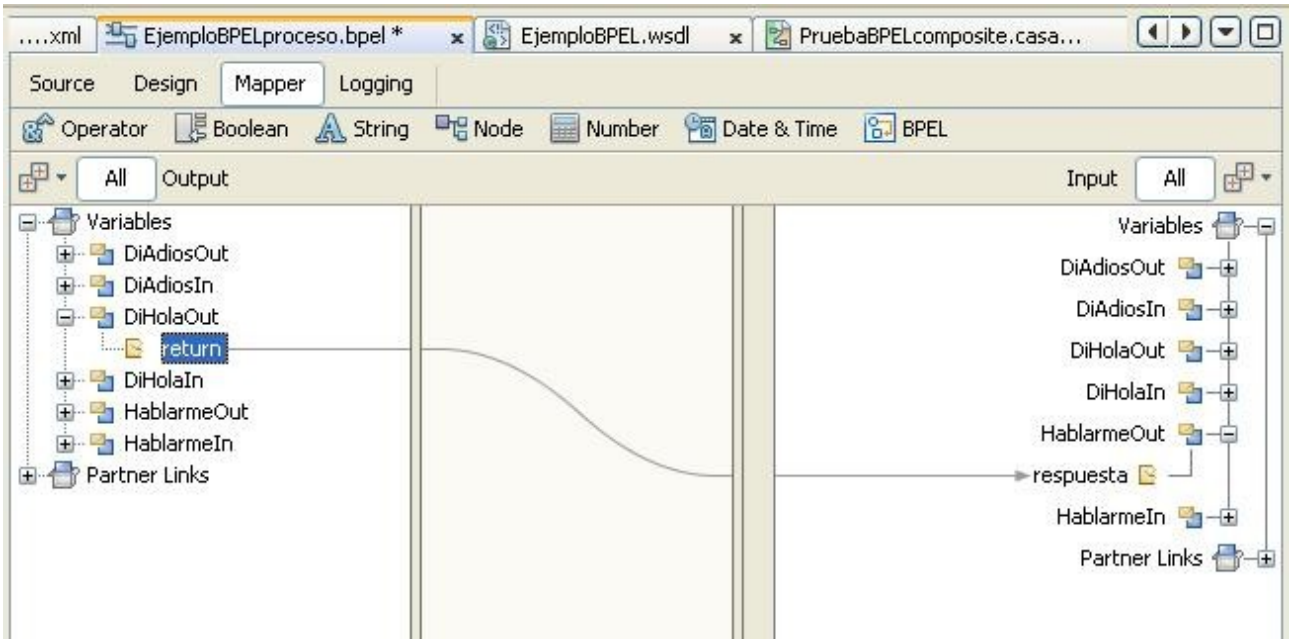
Y en cada uno de ellos metemos el nombre del HablarmeIn en la parte que corresponda:
Para decir Hola:



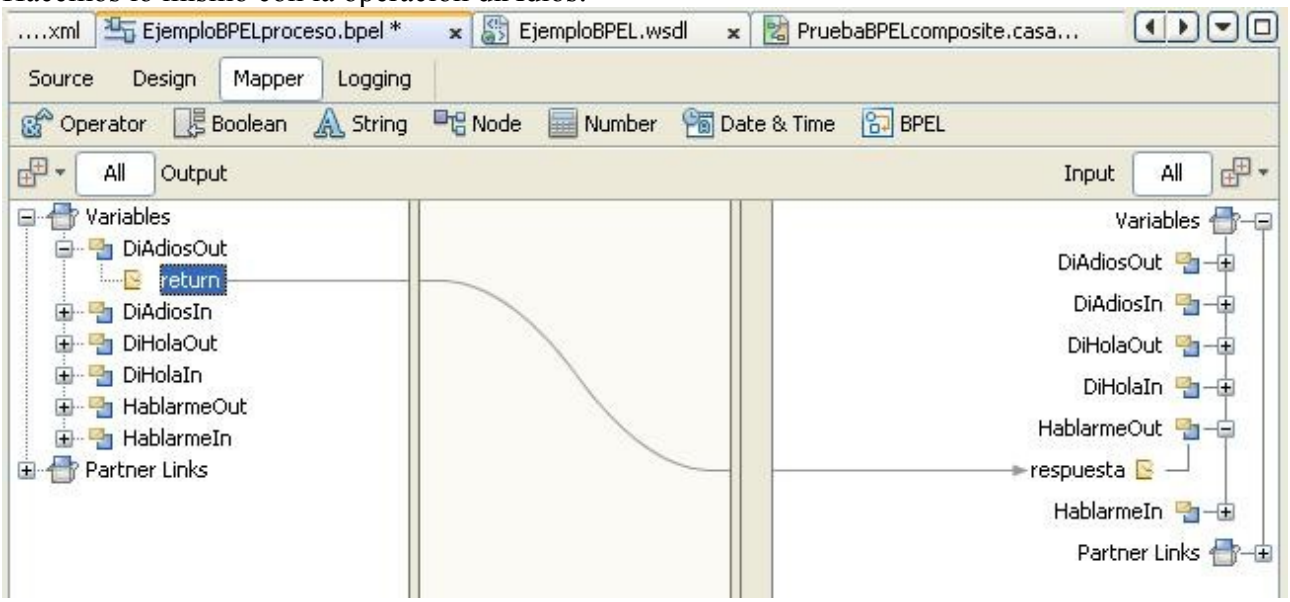
Para decir adios.



Y luego en los otros *assigns* la salida:



Hacemos lo mismo con la operación diAdios:



Y listo, podemos volver a ejecutar tests pasándole un booleano para que nos diga “Hola” o “Adios”. No es muy impresionante pero para empezar no está mal.

Nota: ¿Os dice siempre “Hola”? Eso es que el flujo tiene algo mal que no habéis quitado... mirad el recorrido que hace el proceso BPEL, a ver si hay algo que sobra...

Ah! Los ejemplos resueltos están en http://evorq.ugr.es/cursows/bpel/

Ejercicio: Servicio Web de Ticket

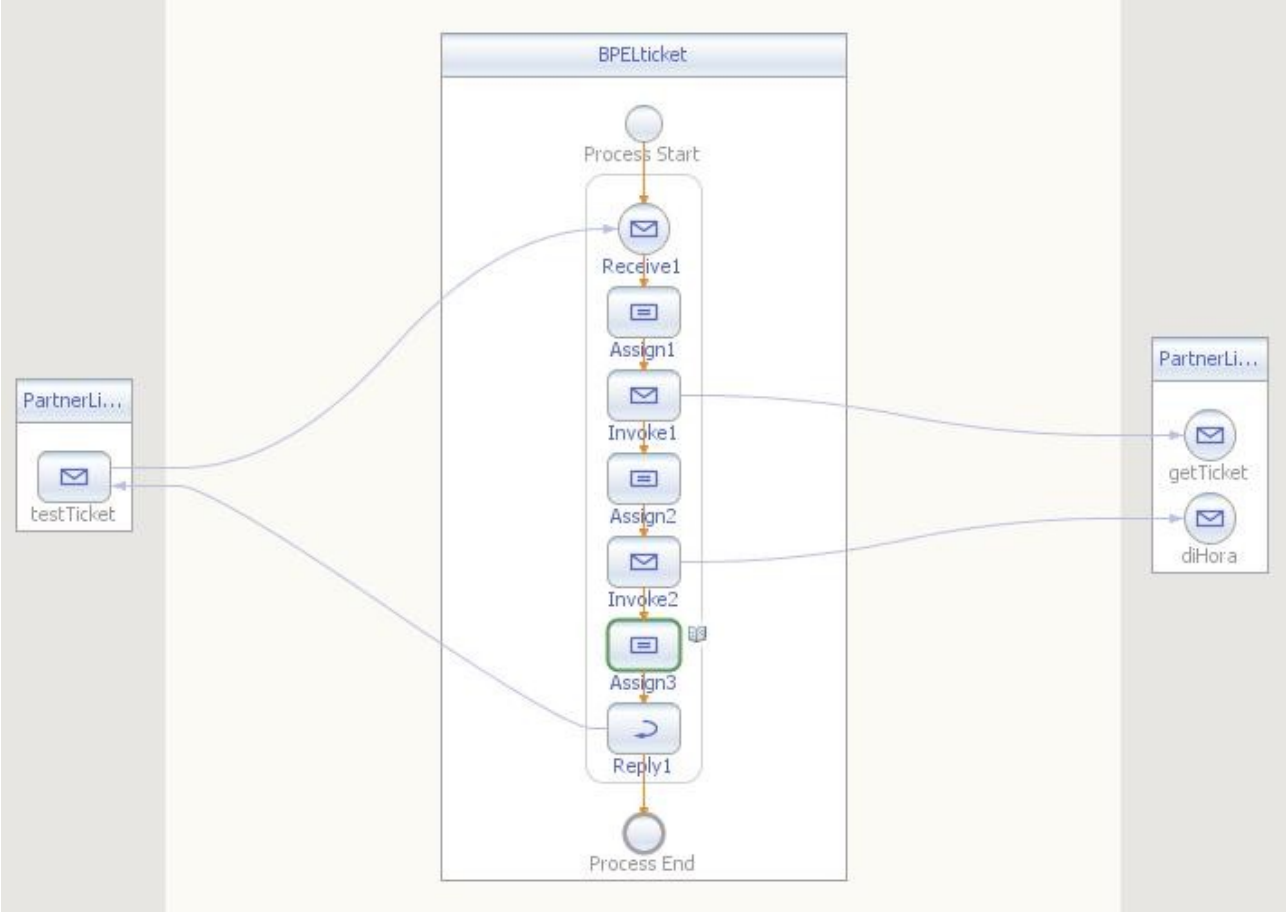
En este ejercicio vamos utilizar gestión de tickets. El uso de tickets sirve para centralizar el acceso a servicios web que requieran autentificación: en vez de usar usuario y contraseña en cada operación segura utilizamos un ticket autogenerado por el sistema.

Vamos a crear un nuevo proyecto. El WSDL de entrada del proceso BPEL tendrá una operación con dos strings de entrada (ususario/contraseña) y uno de salida que devuelve la hora.

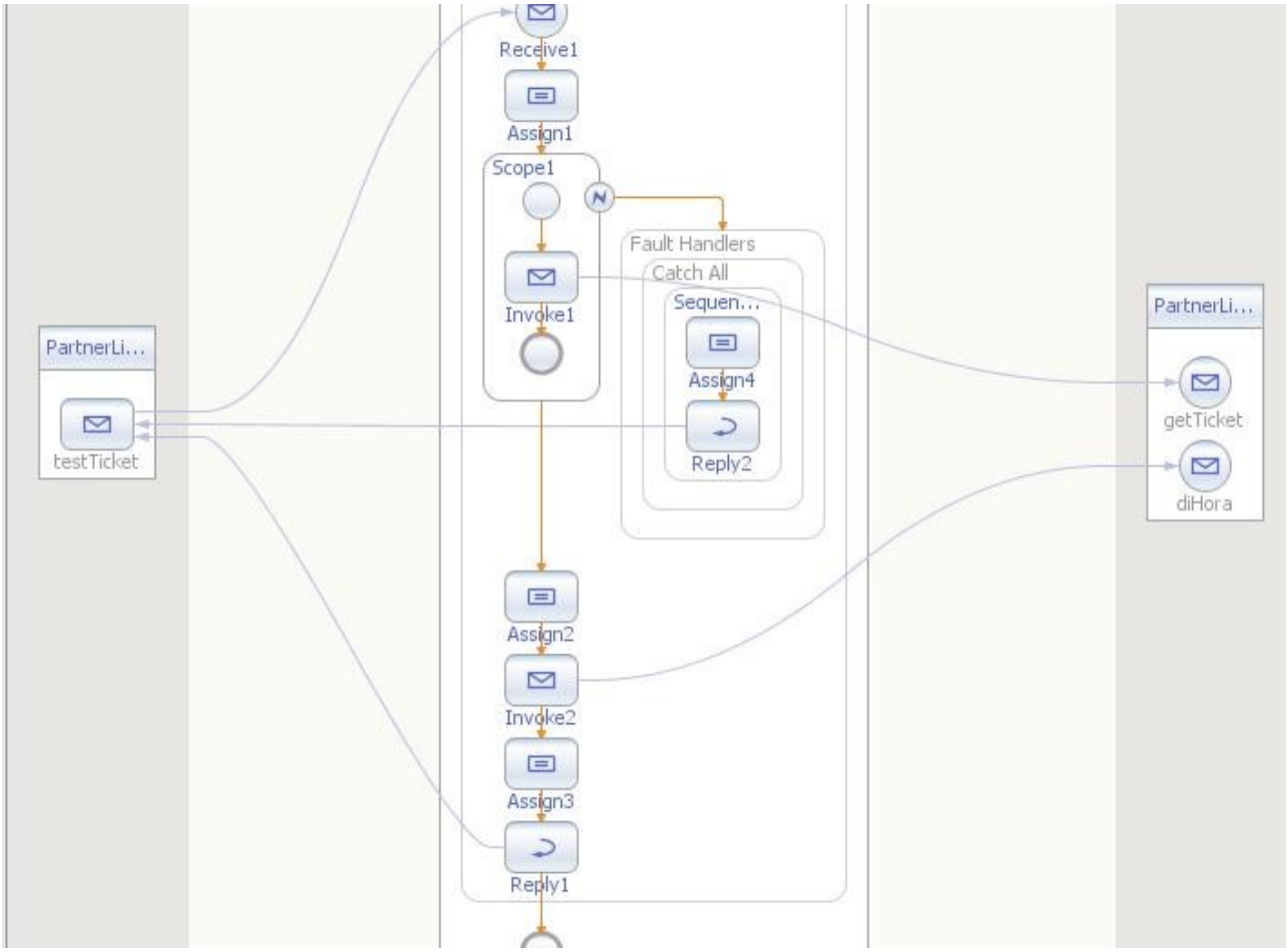
El WSDL a llamar está en http://evorq.ugr.es:8080/evorqLogin/loginServiceService?wsdl

Hay que llamar primero a la operación "getTicket" que devuelve un ticket que será utilizado en la operación "diHora".

El proceso tiene que quedar así:



Comprobar que funciona con el usuario **batman** y la contraseña **joker.** Comprobad qué pasa si metemos algún parámetro mal. Utilizar el manejador de faltas para corregir el error: añadir un *scope,* pulsar en él, añadir el *fault handler,* añadir un *catch all*, y asignar un error al *reply.*


Receive1
Assign1
Scope1
Fault Handlers
Catch All
Sequen...
Invoke1
Assign4
Reply2
PartnerLi...
testTicket
PartnerLi...
getTicket
diHora
Assign2
Invoke2
Assign3
Reply1