Understanding and enhancing the generated Java Server Faces client for human tasks

April 2010

Table of Content

1 Introduction

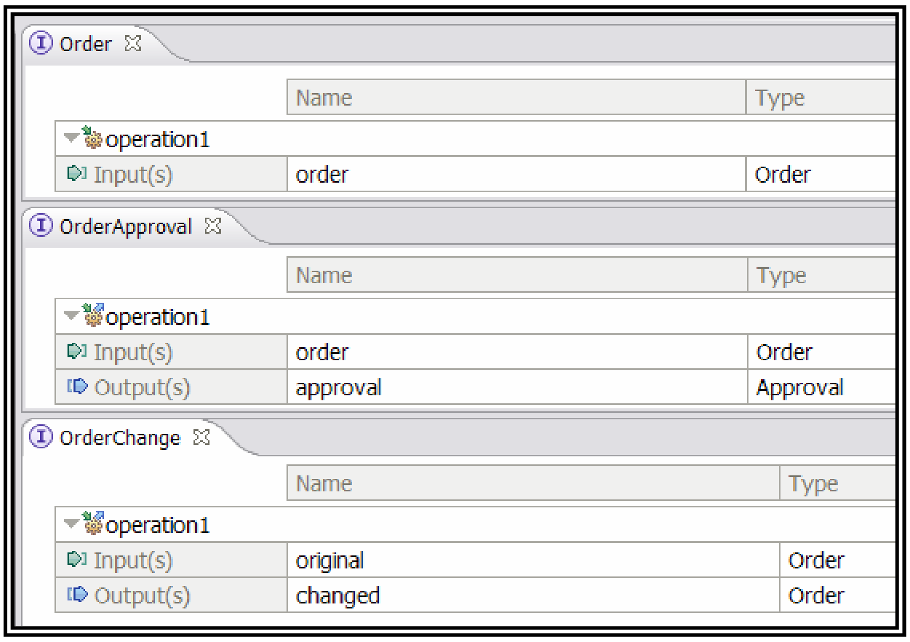
Starting with WebSphere Integration Developer 6.0.2 you can generate user interfaces for a set of human tasks. When you generate the user interface, a dynamic web project is generated in WebSphere Integration Developer. This project contains the generated artifact, which consist of several Java Server Faces (JSF) pages, Java code and required configuration files. These artifacts are automatically packaged into an EAR file, ready to be deployed and run on a WebSphere Application Server. However, such a generated JSF client can never meet all the business requirements that a business user might have, and hence the user needs to customize the generated JSF client to fulfill these requirements.

This document describes how the generated JSF client for human tasks works and how it can be customized and enhanced. The intent of this document is to familiarize the reader with the generated JSF client. Although the document steps into the details of the generated JSF client, it does not describe each and every single part of the generated JSF client. In the last chapter common enhancement scenarios are described in detail, all of them are based on the scenario - a good chance to go over again in one's mind.

1.1 Scenario

A fictional scenario is used in this document to show all the capabilities of the generated JSF client. The scenario is used as an example and is not a solution for a real business problem.

The scenario shows a simple order management process with a human approval step. The solution also supports a person asking another person to change an existing order. This document focuses on the technical aspects of JSF client generation.



Order

Name		Type
operation1		
Input(s)	order	Order

OrderApproval

Name		Type
operation1		
Input(s)	order	Order
Output(s)	approval	Approval

OrderChange

Name		Type
operation1		
Input(s)	original	Order
Output(s)	changed	Order

Figure 1: Scenario - Order Solution the interfaces

The scenario defines the interfaces *Order*, *OrderApproval* and *OrderChange* as shown in the figure above.

The figure below shows the assembled Order Solution.

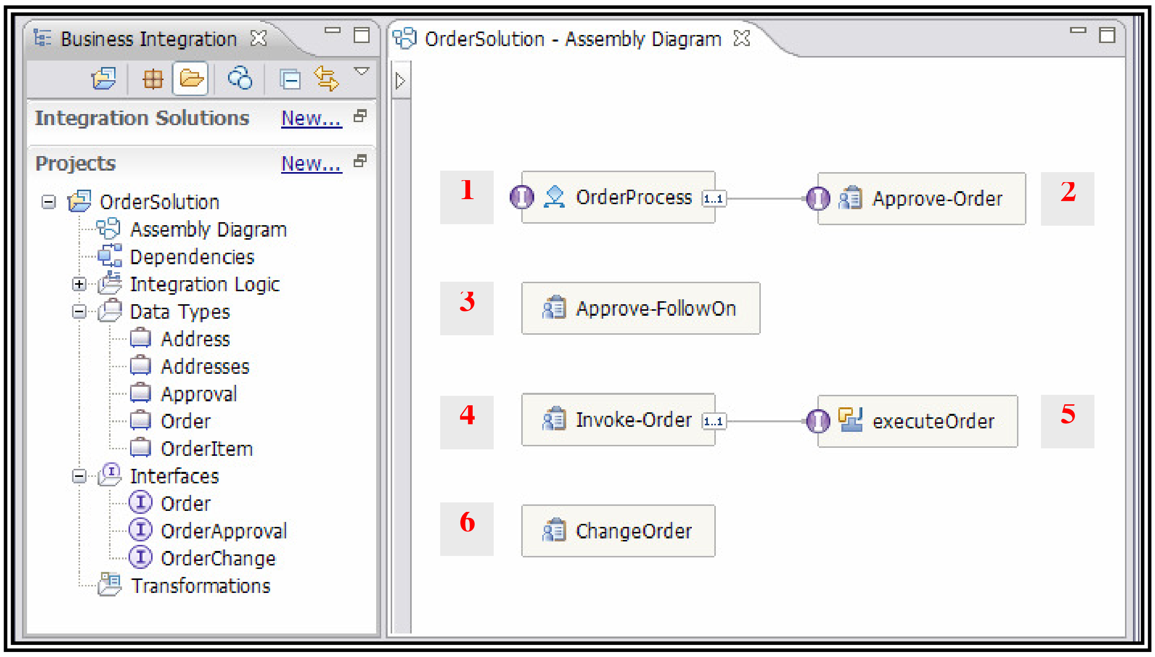


Figure 2: Scenario - Order Solution

1. The BPEL process *OrderProcess* contains all the business logic for processing an order; and implements interface *Order*. The process invokes a manual step to approve the order. The aim is to show how BPEL processes are started by the generated JSF client and how the custom properties (*OrderStatus* and *DepartmentNumber*) defined on the process template are used to search for process instances at runtime. The wired to-do task *Approve-Order,* which implements the interface *OrderApproval*, demonstrates how the generated JSF client uses a query to find task instances and correlates those instances with the previously generated input and output message JSF pages (the same mechanism is used for collaboration tasks).

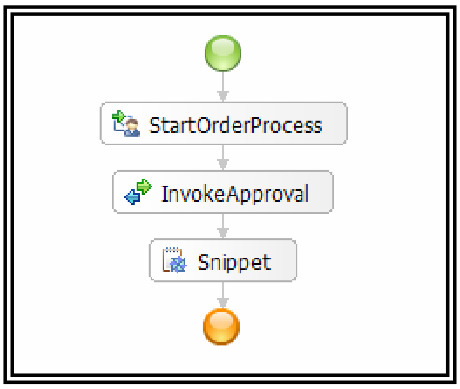
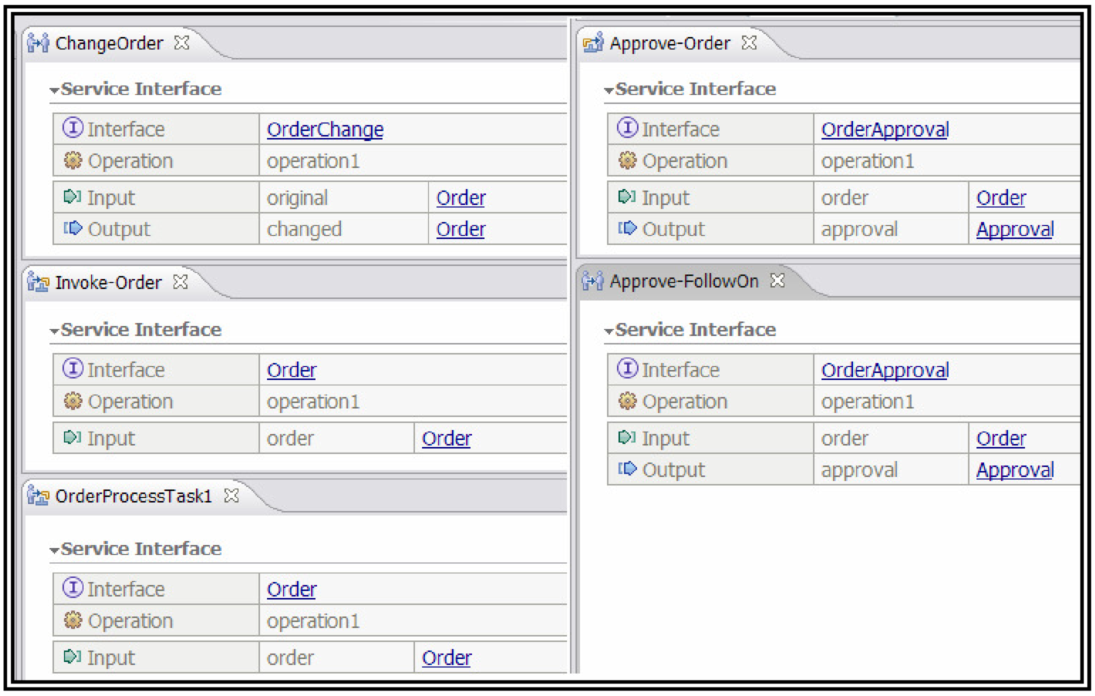


Figure 3: Scenario – BPEL Process OrderProcess

2. *Approve-Order* is the human task that implements the approval step. This task will also be used to show how the creation of follow-on tasks can be implemented.
3. Collaboration task *Approve-FollowOn* will be used to show an enhancement scenario (see '5.2 Add new functionality as a follow-on task') for the generated JSF client. Since it is the follow-on task for *Approve-Order* it also implements the interface *OrderApproval*.
4. The standalone invocation task *Invoke-Order* implements interface *Order*. A BPEL process can be started by a standalone or an inline invocation task. This standalone invocation task is used here to show the difference in the generated JSF code between the two types of invocation tasks. The wired component *executeOrder* is just to show the successful usage of *Invoke-Order*.
5. Component *executeOrder* simulates the order processing through a simple Java implementation. It just prints some parts of the data it receives to the console.
6. The collaboration task *ChangeOrder* implements interface *OrderChange*. The technical aspect of this task is to show how the generated JSF client handles pre-population of the output message with the values from input message. It also shows how the generated JSF client supports input and output messages with ar-

rays of flexible size at run time (refer also to chapter ‘4.2.2 Repetitive data structures - Arrays’), and how optional nested business objects are supported.

The figure below summarizes the implementation of the interfaces by human tasks.



ChangeOrder

Service Interface

Interface	OrderChange	
Operation	operation1	
Input	original	Order
Output	changed	Order

Invoke-Order

Service Interface

Interface	Order	
Operation	operation1	
Input	order	Order

OrderProcessTask1

Service Interface

Interface	Order	
Operation	operation1	
Input	order	Order

Approve-Order

Service Interface

Interface	OrderApproval	
Operation	operation1	
Input	order	Order
Output	approval	Approval

Approve-FollowOn

Service Interface

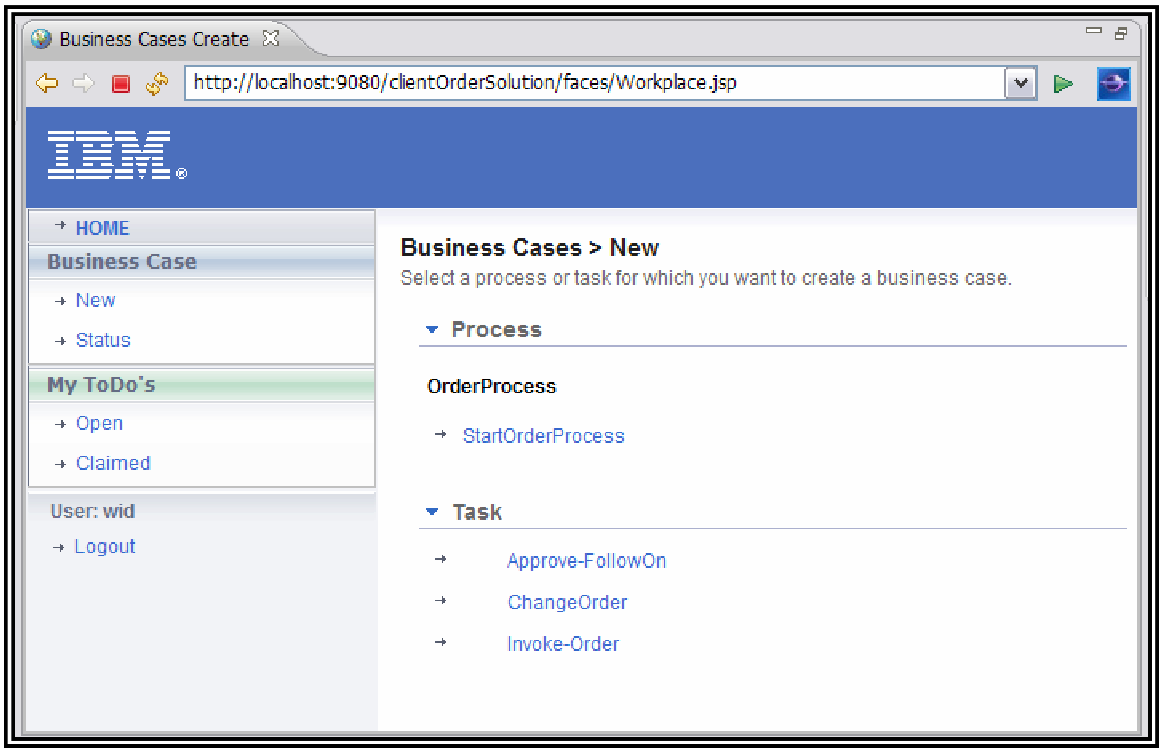
Interface	OrderApproval	
Operation	operation1	
Input	order	Order
Output	approval	Approval

Figure 4: Scenario – Implementation of interfaces by human tasks

2 Structure of the generated client

In this chapter we will discuss the JSF client that has been generated for the entire Order Solution sample. Before discussing the generated artifacts, a short preview of the running client is given here to see which part of the generated JSF client was generated from which human task in our scenario.

2.1 The generated client



Business Cases > New

Select a process or task for which you want to create a business case.

Process

OrderProcess

StartOrderProcess

Task

Approve-FollowOn

ChangeOrder

Invoke-Order

Figure 5: In action - generated JSF client for scenario

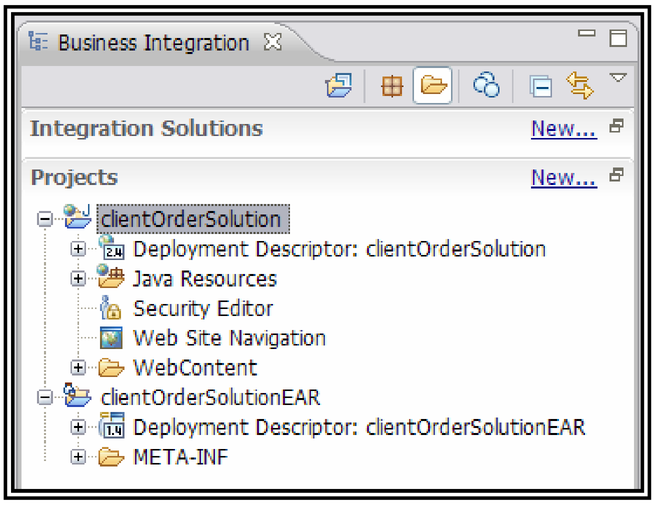
The figure above shows function block *Business Case* and *My ToDo's* on the left. To start a new *Business* Case there are several links:

- *StartOrderProcess* has been generated because of the inline invocation task (defined on the starting Receive activity of the process template)
- *Approve-FollowOn* and *ChangeOrder* have been generated for the two collaboration tasks
- *Invoke-Order* has been generated because of the standalone invocation task

To examine the status of previously started process instances the function *Status* is available in function block *Business Case*, see figure above. This function is only available because of the inline invocation task defined on process template.

Function block *My ToDo's* has been generated to work on instances of collaboration tasks *Approve-FollowOn* and *ChangeOrder*, and to work on instances of to-do task *Approve-Order*.

The *JSF custom client* generator creates a dynamic web project and an EAR project in WebSphere Integration Developer. Both projects together contain all the artifacts that have been generated based on the selected human tasks. The following figure shows the list of generated artifacts.



```
clientOrderSolution
    Deployment Descriptor: clientOrderSolution
    Java Resources
    Security Editor
    Web Site Navigation
    WebContent
clientOrderSolutionEAR
    Deployment Descriptor: clientOrderSolutionEAR
    META-INF
```

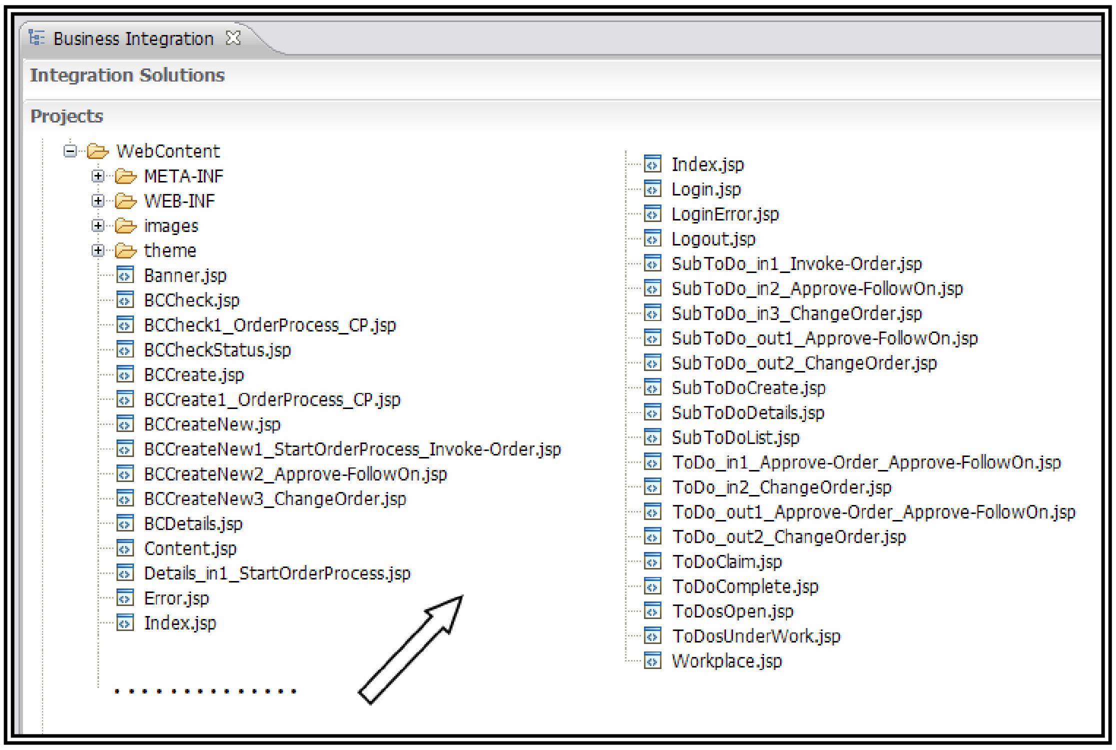
Figure 6: Generated JSF client for scenario

The EAR file is ready to be deployed, the defined security role *CustomClientUser* is mapped to *All authenticated users*, hence the generated JSF client works properly on the server with or without security settings. This security role has also been created and inserted during client generation.

You can implement your own client generator by implementing an Eclipse extension point. This extension point allows the contribution of wizard pages to the *User Interface Wizard for Human Tasks*. The selected human tasks will then be passed to your custom client generator, for more details refer to [4].

2.2 JSF pages

The figure below shows the JSF pages in the generated dynamic web project.



```
Business Integration
Integration Solutions
Projects
WebContent
    META-INF
    WEB-INF
    images
    theme
    Banner.jsp
    BCCheck.jsp
    BCCheck1_OrderProcess_CP.jsp
    BCCheckStatus.jsp
    BCCreate.jsp
    BCCreate1_OrderProcess_CP.jsp
    BCCreateNew.jsp
    BCCreateNew1_StartOrderProcess_Invoke-Order.jsp
    BCCreateNew2_Approve-FollowOn.jsp
    BCCreateNew3_ChangeOrder.jsp
    BCDetails.jsp
    Content.jsp
    Details_in1_StartOrderProcess.jsp
    Error.jsp
    Index.jsp
    ..............
    Index.jsp
    Login.jsp
    LoginError.jsp
    Logout.jsp
    SubToDo_in1_Invoke-Order.jsp
    SubToDo_in2_Approve-FollowOn.jsp
    SubToDo_in3_ChangeOrder.jsp
    SubToDo_out1_Approve-FollowOn.jsp
    SubToDo_out2_ChangeOrder.jsp
    SubToDoCreate.jsp
    SubToDoDetails.jsp
    SubToDoList.jsp
    ToDo_in1_Approve-Order_Approve-FollowOn.jsp
    ToDo_in2_ChangeOrder.jsp
    ToDo_out1_Approve-Order_Approve-FollowOn.jsp
    ToDo_out2_ChangeOrder.jsp
    ToDoClaim.jsp
    ToDoComplete.jsp
    ToDosOpen.jsp
    ToDosUnderWork.jsp
    Workplace.jsp
```

Figure 7: Generated JSF pages for scenario

The names of the JSF pages in the context of function block *Business Case* start with the letters *BC*, and the names of those which are in the context of function block *My ToDo's* start with *ToDo* or *SubToDo*. In addition to this the JSF pages can be categorized as follows:

1. JSF pages for the **infrastructure** of the generated JSF client. Most of these pages are the same, regardless of the human tasks selected when generating the JSF client. These page are:
    - Banner.jsp
      includes the image shown in the top of the JSF client
    - Error.jsp
      generic page to show an exception that is passed the user
    - Index.jsp
      the start page to invoke the JSF client, with or without server security

- Login.jsp, LoginError.jsp and Logout.jsp
  these pages are responsible to do a from-based login and logout (invalidate session), and show login error.

The content of the following **infrastructure** pages depends on the human tasks, selected. These pages are

- Content.jsp
  contains the function block *Business Case* if at least one collaboration or invocation task is among the human tasks selected for generation of the JSF client. The invocation task can be a standalone or an inline invocation task. The function *Status* of this function block is only available if there is at least one inline invocation task. The function block *My ToDo's* is only available if there is at least one collaboration task or one to-do task. The to-do task can be a standalone or an inline to-do task.
- Workplace.jsp
  contains a description for the functions that are available in Content.jsp.

2. JSF pages showing **overview** information. Their availability and content depends on the selection of human tasks for which the JSF client has been generated. The **overview** pages are the linked targets from the function block, they allow the user to step into more details.
   - BCCreate.jsp and BCCheck.jsp
     contain command links (`commandLink` from the JSF tag library) to initialize backing beans and to navigate into more details. More information on this is shown '5.1 Add support for a new human task'.
   - ToDosOpen.jsp and ToDosUnderWork.jsp
     both contain a list (`list` from the Business Process Choreographer tag library) to show instances of those collaboration and to-do tasks for which the JSF client has been generated. More information on this can be found below '2.3 Java packages' and in '2.4 Business Process Choreographer tag library'.
3. **Main pages** are JSF pages that include other JSF pages via `jsp:include`. The main pages contain buttons (`commandbar` from the Business Process Choreographer tag library) to allow user interactions in the JSF client. The including mechanism has been implemented to reduce the effort when changes apply to the `commandbar` or other common parts of the page, because the total number of **sub pages** (see below) depends on the number of human tasks for which to generate the JSF client. The total number of **main pages** does not depend on the number of human tasks for which to generate the JSF client, it depends on the type of tasks which have been selected.

- BCCreateNew.jsp
  is available if there is at least one collaboration or invocation task in the list of selected human tasks. The invocation task can be either a standalone or an inline task.
- BCCheckStatus.jsp and BCDetails.jsp
  are only available if the JSF client has been generated for at least one inline invocation task.
- ToDoClaim.jsp and ToDoComplete.jsp
  are only available if there is at least one collaboration or to-do task in the list of selected human tasks. The to-do task can be either a standalone or an inline to-do task.
- SubToDoCreate.jsp and SubToDoDetailsList.jsp
  are only available if the JSF client has been generated for at least one potential main task and one potential sub task. Potential main tasks are collaboration tasks or to-do tasks, potential sub tasks are collaboration tasks or standalone invocation tasks.

4. **Sub pages** are included in **main pages**. They contain proper JSF input-/output fields to render the message of a human task. There is one **sub page** for the input- and another one for the output message.

   Human tasks that implement the same interface will share the same message **sub pages**. The act of sharing is indicated by the last part of the **sub page** name, which consists of the template names of all human tasks that share the **sub page**. Since the length of a **sub page** name has an upper limit it could be the case that not all template names of human tasks are part of this name, even if they implement the same interface. To indicate in which **main page** a **sub page** is included at runtime the **sub page** name has the same prefix as the corresponding **main page** (it is the function or function block). Furthermore the name of the **sub page** contains the sub string ‘in’ or ‘out’ to indicate whether it is for an input or output message. Finally, to make the name of a **sub page** unique it also contains a number count. See the following example:

   ToDo_in1_Approve-Order_Approve-FollowOn.jsp

   The above shown **sub page** will be used in function block *My ToDo’s* to render the input message of the tasks *Approve-Order* and *Approve-FollowOn*.

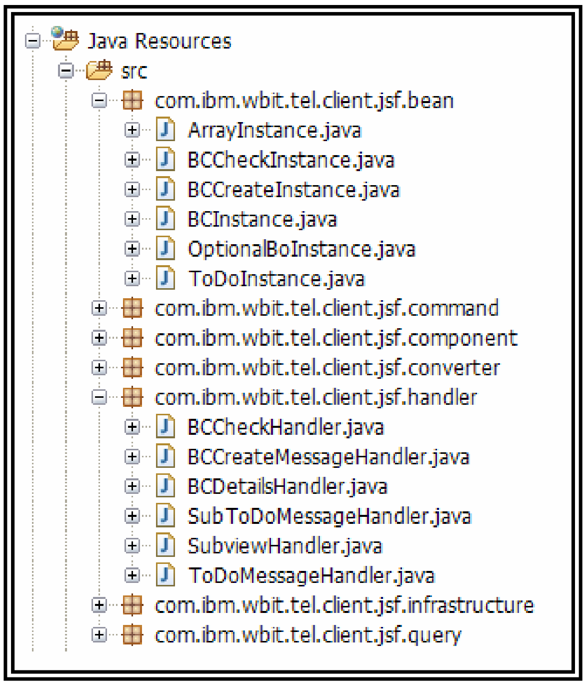
   Note:

   > For inline human tasks, the name of the Receive activity or the name of the human task activity is part of the **sub page** name rather then the task template name. This is because the task template names of inline human tasks are generated by the BPEL editor based on the process name and the human task activity name, which make them less readable.

For more details on how the inclusion of sub pages works refer to chapter ‘4.1 The sub page mechanism’.

2.3 Java packages

Besides the JSF pages, the generated dynamic web project also contains several java packages. These packages are discussed in this chapter.



```
Java Resources
  src
    com.ibm.wbit.tel.client.jsf.bean
      ArrayInstance.java
      BCCheckInstance.java
      BCCreateInstance.java
      BCInstance.java
      OptionalBoInstance.java
      ToDoInstance.java
    com.ibm.wbit.tel.client.jsf.command
    com.ibm.wbit.tel.client.jsf.component
    com.ibm.wbit.tel.client.jsf.converter
    com.ibm.wbit.tel.client.jsf.handler
      BCCheckHandler.java
      BCCreateMessageHandler.java
      BCDetailsHandler.java
      SubToDoMessageHandler.java
      SubviewHandler.java
      ToDoMessageHandler.java
    com.ibm.wbit.tel.client.jsf.infrastructure
    com.ibm.wbit.tel.client.jsf.query
```

*Figure 8: Generated JSF client – package *.handler and *. bean*

In the figure above, the classes in package com.ibm.wbit.tel.client.jsf.handler implement listeners.

The classes BCCheckHandler and BCCreateMessageHandler are action event listeners for JSF `commandLink` tags defined in pages BCCheck.jsp and BCCreate.jsp. They initialize the classes BCCheckInstance and BCCreateInstance of package com.ibm.wbit.tel.client.jsf.bean with the values extracted form the action events for further processing.

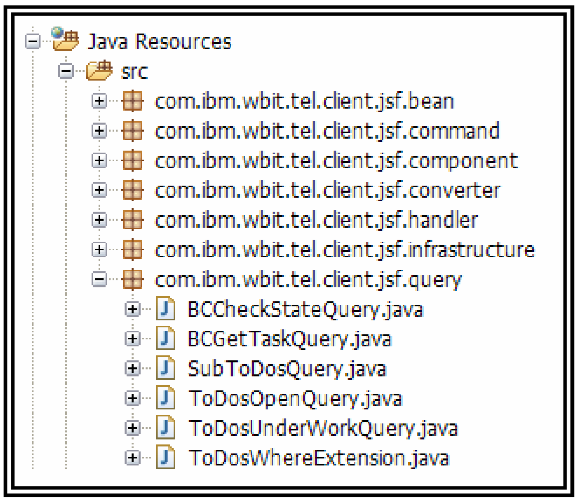
The classes BCDetailsHander, SubToDoMessageHandler and ToDoMessageHandler are item listeners for the Business Process Choreographer tag library `list` tags:

- BCDetailsHandler is a listener for the `list` tag in page BCCheckStatus.jsp
- SubToDoMessageHandler is a listener for the `list` tag in page SubToDoList.jsp
- ToDoMessageHandler is a listener for the `list` tags in page ToDosOpen.jsp and ToDosUnderWork.jsp.

An instance of class BCInstance or ToDoInstance is associated with each click on a `list` tag. These instances are used by the previously mentioned handlers to get values for further processing. A common functionality of each handler is to bring up a new **main page** which dynamically includes a **sub page** for rendering the input and output message of a human task. This common functionality is implemented by class SubviewHandler which is extended by all the handler classes described above. Refer to chapter '3.2 Configuration file faces-config.xml' to see how the SubviewHandler is initialized, and to chapter '4.1 The sub page mechanism' to see how this works at runtime.

The class ArrayInstance, supports input and output messages with arrays of flexible size; that is, a flexible number of array entries. This class is the data model for the JSF tag `dataTable` which is used in **sub page**s to visualize input and output messages with arrays, for example ToDo_in2_ChangeOrder.jsp and ToDo_out2_ChangeOrder.jsp. The class is implemented as a recursive data model to support an arbitrary nesting level of the arrays. For example an array containing an array, containing an array, containing an array and so on, for more details refer to '4.2.2 Repetitive data structures - Arrays'

The class OptionalBoInstance collapses and expands optional nested business objects in **sub pages** that render an input message, for example see BCCreateNew3_ChangeOrder.jsp.



```
Java Resources
  src
    com.ibm.wbit.tel.client.jsf.bean
    com.ibm.wbit.tel.client.jsf.command
    com.ibm.wbit.tel.client.jsf.component
    com.ibm.wbit.tel.client.jsf.converter
    com.ibm.wbit.tel.client.jsf.handler
    com.ibm.wbit.tel.client.jsf.infrastructure
    com.ibm.wbit.tel.client.jsf.query
      BCCheckStateQuery.java
      BCGetTaskQuery.java
      SubToDosQuery.java
      ToDosOpenQuery.java
      ToDosUnderWorkQuery.java
      ToDosWhereExtension.java
```

*Figure 9: Generated JSF client – package *.query*

To get information about human task instances and process instances, the data base of the business process engine must be queried. The classes of package com.ibm.wbit.tel.client.jsf.query are provided to perform such queries. All classes of this package extend class GenericBPCQuery of the Business Process Choreographer tag library, except class ToDosWhereExtension. For more information, see '2.4 Business Process Choreographer tag library'.

The class BCCheckStateQuery queries for process instances that have been started by the generated JSF client via an inline invocation task. Custom properties defined on the process template are used as search criteria for process instances. For more information, see the process template of the scenario.

A process template may have several receive activities, and each activity may have an inline invocation task that is able to start an instance of the process. The class BCGetTaskQuery finds out at runtime which inline invocation task has been used to start the process instance.

Since the generated JSF client is a specialized client, it only queries for instances of those human tasks which have been selected in the *User Interface Wizard*. This is ensured by the class ToDosWhereExtension. This class is used by the classes SubToDosQuery, ToDosOpenQuery and ToDosUnderWorkQuery to tailor the queries properly. If the queries would not be tailored to human tasks “known” by the generated JSF client, the client would fail if the input or output message of a human task instance would be shown for which no proper **sub page** has been generated.

The class ToDosOpenQuery is used to query for all “known” human task instances that are in state Ready and class ToDosUnderWorkQuery queries for those tasks in state Claimed according to the life cycle of a human task, see also [7]. The class SubToDosQuery is used to query for all sub task instances for a given main task instance.

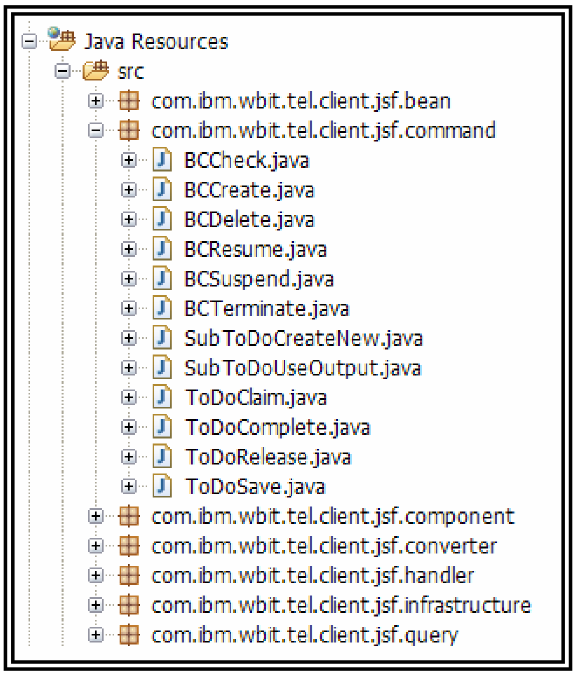
The following table shows the generated JSF client page in which the query result of the classes described above is shown.

Query class	JSF page	Remarks
BCCheckStateQuery	BCCheckStatus.jsp	Result is shown as a list with a maximum of three custom properties
BCGetTaskQuery	BCDetails.jsp	Result is shown as an input or output message of the inline invocation task
SubToDosQuery	SubToDoList.jsp	Result is shown as a list, only if the client has potential sub tasks
ToDosOpenQuery	ToDosOpen.jsp	Result is shown as a list
ToDosUnderWorkQuery	ToDosUnderWork.jsp	Result is shown as a list

Table 1: Query classes and JSF page where to show result

To see how the query classes are initialized refer to chapter ‘3.2 Configuration file faces-config.xml’

All of the classes in package com.ibm.wbit.tel.client.jsf.command implement the interface com.ibm.bpc.clientcore.Command, hence they are the listeners of `command` buttons from the Business Process Choreographer tag library.



```
Java Resources
  src
    com.ibm.wbit.tel.client.jsf.bean
    com.ibm.wbit.tel.client.jsf.command
      BCCheck.java
      BCCreate.java
      BCDelete.java
      BCResume.java
      BCSuspend.java
      BCTerminate.java
      SubToDoCreateNew.java
      SubToDoUseOutput.java
      ToDoClaim.java
      ToDoComplete.java
      ToDoRelease.java
      ToDoSave.java
    com.ibm.wbit.tel.client.jsf.component
    com.ibm.wbit.tel.client.jsf.converter
    com.ibm.wbit.tel.client.jsf.handler
    com.ibm.wbit.tel.client.jsf.infrastructure
    com.ibm.wbit.tel.client.jsf.query
```

*Figure 10: Generated JSF client – package *.command*

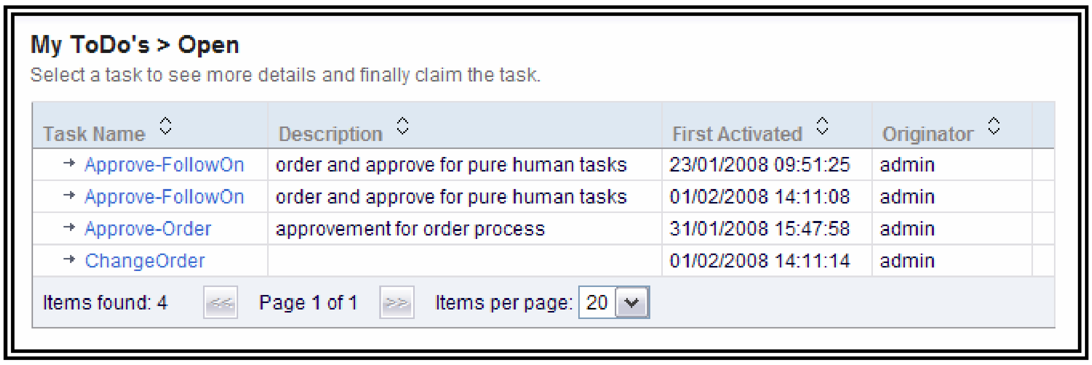
The classes implement calls to the Human Task Manager Service and the BusinessFlowManagerService to perform the actions associated with the `command` buttons. To allow this, proper initialized beans from package com.ibm.wbit.tel.client.jsf.bean are passed with each button event. For more details on the Business Process Choreographer tag library `command` buttons refer to chapter ‘2.4 Business Process Choreographer tag library’.

2.4 Business Process Choreographer tag library

The generated JSF client is based on the JSF components which primarily have been developed for the Business Process Choreographer Explorer. These components are also known as the Business Process Choreographer tag library, see [2] for detailed information. The Business Process Choreographer tag library is based on the client model of the Business Flow Manager Service and the Human Task Manager Service, for detailed information on that refer to [3]. This chapter shows only those parts of the Business Process Choreographer tag library which are mainly used in the generated JSF client.

2.4.1 List component

The `list` component is used to show a list or table on a JSF page. The figure below shows the list of open ToDo's.



My ToDo's > Open

Select a task to see more details and finally claim the task.

Task Name	Description	First Activated	Originator
→ Approve-FollowOn	order and approve for pure human tasks	23/01/2008 09:51:25	admin
→ Approve-FollowOn	order and approve for pure human tasks	01/02/2008 14:11:08	admin
→ Approve-Order	approvement for order process	31/01/2008 15:47:58	admin
→ ChangeOrder		01/02/2008 14:11:14	admin

Items found: 4   <<   Page 1 of 1   >>   Items per page: 20

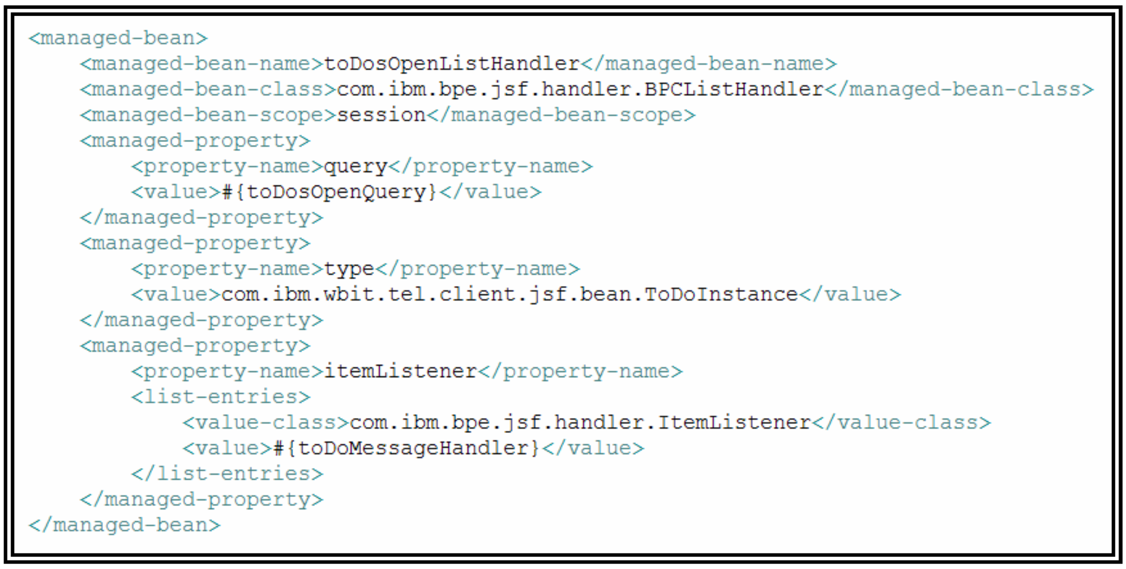
Figure 11: Business Process Choreographer tag library – list component in action

The source code for this list is shown below (it is a part of page ToDosOpen.jsp).

```
<bpe:list model="#{toDosOpenListHandler}" styleClass="list"
    rows="20" headerStyleClass="template-headers"
    buttonStyleClass="list-buttons" checkbox="false">

    <bpe:column name="name" action="#{toDoMessageHandler.getNextViewOpenList}" />
    <bpe:column name="description" />
    <bpe:column name="firstActivationTime"  />
    <bpe:column name="originator"  />
</bpe:list>
```

Figure 12: Business Process Choreographer tag library – list component in source code

The key for understanding the configuration of the `list` component shown above is to look at its `model`, the toDosOpenListHandler, which itself is defined in the faces-config.xml of the generated JSF client as follows.



```xml
<managed-bean>
    <managed-bean-name>toDosOpenListHandler</managed-bean-name>
    <managed-bean-class>com.ibm.bpe.jsf.handler.BPCListHandler</managed-bean-class>
    <managed-bean-scope>session</managed-bean-scope>
    <managed-property>
        <property-name>query</property-name>
        <value>#{toDosOpenQuery}</value>
    </managed-property>
    <managed-property>
        <property-name>type</property-name>
        <value>com.ibm.wbit.tel.client.jsf.bean.ToDoInstance</value>
    </managed-property>
    <managed-property>
        <property-name>itemListener</property-name>
        <list-entries>
            <value-class>com.ibm.bpe.jsf.handler.ItemListener</value-class>
            <value>#{toDoMessageHandler}</value>
        </list-entries>
    </managed-property>
</managed-bean>
```

Figure 13: Business Process Choreographer tag library – list component, configuration of its model in faces-config.xml

See figure above, the result set to be shown in the `list` is provided by the query toDosOpenQuery which is the already mentioned query class ToDosOpenQuery (also configured in faces-config.xml). Each query result is an instance of the already mentioned class ToDoInstance, see type configuration above. The `list` items can be clickable to drill down into more details, for this the configured itemListener is invoked. In the configuration above this is toDoMessageHandler which is the already mentioned class ToDoMessageHandler (also configured in faces-config.xml). This class must implement interface ItemListener so with each click in the `list` the corresponding instance of ToDoInstance is passed to method itemChanged(…).

The source code of the `list` component in figure 12 shows the definition of four `column` tags. Each `column` has a mandatory `name` attribute; the value of this `name` attribute must be a valid property of class ToDoInstance. The `action` attribute is used to make a `column` clickable; the value of the `action` attribute must evaluate to a string which is used as navigation target. This string is the logical name of the JSF page that is opened when a click in the `list` is performed.

For detailed information about `list` component refer to [2].

2.4.2 Commandbar component

The `commandbar` component contains `command` tags to allow user interactions. The figure below shows the source code of the `commandbar` component which defines the `command` claim; it is part of page ToDoClaim.jsp.

```
<bpe:commandbar model="#{toDoMessageHandler}" buttonStyleClass="button">
    <bpe:command label="#{bundle['BUTTON_CLAIM']}" commandID="ClaimToDoInstance"
        commandClass="com.ibm.wbit.tel.client.jsf.command.ToDoClaim"
        action="ToDoComplete" />
</bpe:commandbar>
```

Figure 14: Business Process Choreographer tag library – commandbar component in source code

The model class of the `commandbar` component must implement the interface Item-Provider to allow the `commandClass` to get the items associated with the `command`. In the generated JSF client the class ToDoMessageHandler also implements ItemProvider to provide the latest clicked ToDoInstance for `commandbar` components. The `command-Class` ToDoClaim implements interface Command, hence its method execute(…) is invoked once the command button is pressed. The navigation target defined by the `action` attribute is hard coded. Another option for the navigation target is to code the method execute(…) to return a computed string, rather than to defining the navigation target in the `action` attribute. To get the entire picture the following figure shows the implementation of method execute(…).

```
/**
 * Executes the claim command for the ToDo selected in open ToDos list.
 * Exceptions to be shown in commandbar will be thrown as CommandException.
 *
 * @param items the list of selected items
 * @return next navigation route as defined in faces-config, may be overwritten by action of commandbar
 * @exception CommandException to be shown in commandbar
 * @see com.ibm.bpc.clientcore.Command#execute(java.util.List)
 */
public String execute(List items) throws ClientException {

    if (items != null) {
        HumanTaskManagerService htm;
        //we have only one selected
        Object obj = items.get(0);
        if (obj instanceof ToDoInstance) {
            ToDoInstance toDoInstance = (ToDoInstance) obj;
            try {
                htm = ServiceFactory.getInstance().getHumanTaskManagerService();
                htm.claim(toDoInstance.getID());
            } catch (Exception e) {
                throw new CommandException(NLS_CATALOG, SERVICE_FAILED_TO_CLAIM_HUMAN_TASK, null, e);
            }
        }
    }
    return null;
}
```

Figure 15: Business Process Choreographer tag library – commandbar, implementation of execute method

For detailed information about the `commandbar` component refer to [2].

3 Configuration of the generated client

As described in chapter ‘2 Structure of the generated client’ some of the generated artifacts are the same, regardless of the types of human tasks for which the JSF client has been generated. All of the Java Code is invariant in this manner, even though at runtime it must work and act depending on the selection of human tasks. This is solved by the configuration of the web application, especially by the faces-config.xml file.

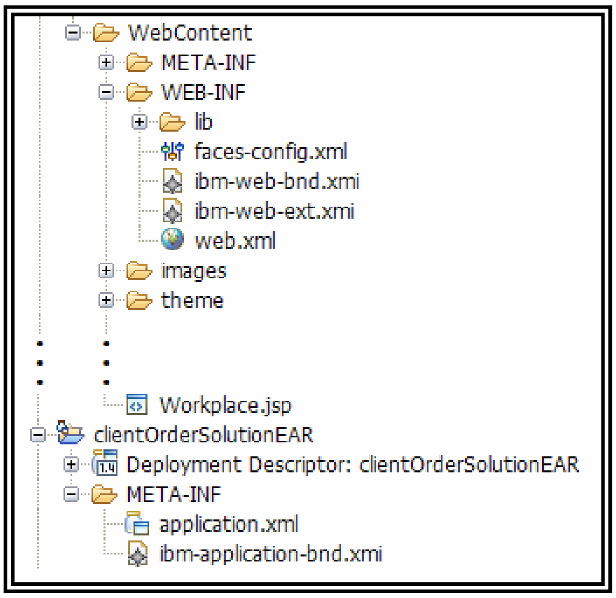


Figure 16: Configuration files of the generated JSF client

3.1 Configuration file web.xml

The content of web.xml does not contain any information about the selected human tasks for which the JSF client has been generated, but since it is the base configuration for the entire web application, the main points of it are described here.

The security role `CustomClientUser` is defined for a FORM based login, the infrastructure pages Login.jsp and LoginError.jsp are configured to be invoked at runtime for user authentication. The mapping of the security role to “All authenticated users” is done in the deployment descriptor file application.xml; the mapping can be seen as xml source in the file ibm-application-bnd.xmi. Mappings for the pages Login.jsp, LoginError.jsp and Logout.jsp servlet are defined in web.xml. This is required to invoke these JSPs for the FORM based login.

The page Index.jsp is configured to be the welcome file; it redirects requests properly to the overview page which shows all available functions of the entire web application.

Since the generated JSF client is required to work in a multi lingual environment it is important that all requests to it and responses from it have an UTF-8 character encoding. To ensure this the `UnicodeFilter` is defined.

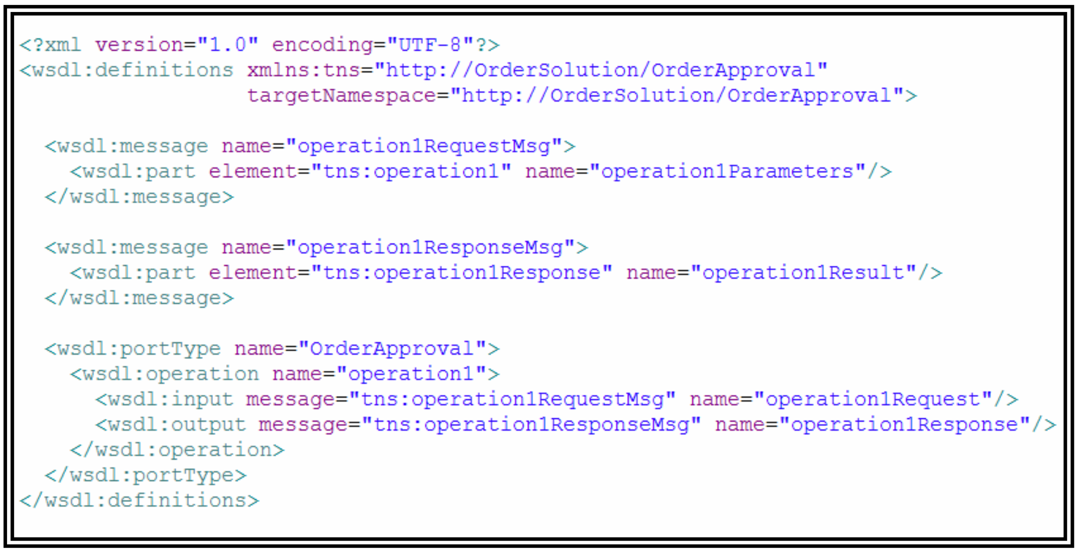
As mentioned previously the Business Process Choreographer tag library is based on the client model of the Business Flow Manager Service and the Human Task Manager Service. Both work on EJB interfaces provided by the business process engine. To make these EJB interfaces available, EJB references are defined in web.xml. These references are used in faces-config.xml to initiate the infrastructure of the generated JSF client. The EJB references are mapped to JNDI names in the file ibm-web-bnd.xmi.

3.2 Configuration file faces-config.xml

The dependency between the selection of human tasks for which the JSF client has been generated and its Java code is captured in the configuration file faces-config.xml. Before stepping into the details of faces-config.xml, the following definitions are introduced:

- The qualified name of a WSDL message, QNameOfWSDLMessage:

  This is the concatenation of the target namespace of a WSDL definition and the name of a message defined in that namespace. The target namespace is enclosed in brackets { }, furthermore it is different for input and output message, see the following example.



```
<?xml version="1.0" encoding="UTF-8"?>
<wsdl:definitions xmlns:tns="http://OrderSolution/OrderApproval"
                  targetNamespace="http://OrderSolution/OrderApproval">

  <wsdl:message name="operation1RequestMsg">
    <wsdl:part element="tns:operation1" name="operation1Parameters"/>
  </wsdl:message>

  <wsdl:message name="operation1ResponseMsg">
    <wsdl:part element="tns:operation1Response" name="operation1Result"/>
  </wsdl:message>

  <wsdl:portType name="OrderApproval">
    <wsdl:operation name="operation1">
      <wsdl:input message="tns:operation1RequestMsg" name="operation1Request"/>
      <wsdl:output message="tns:operation1ResponseMsg" name="operation1Response"/>
    </wsdl:operation>
  </wsdl:portType>
</wsdl:definitions>
```

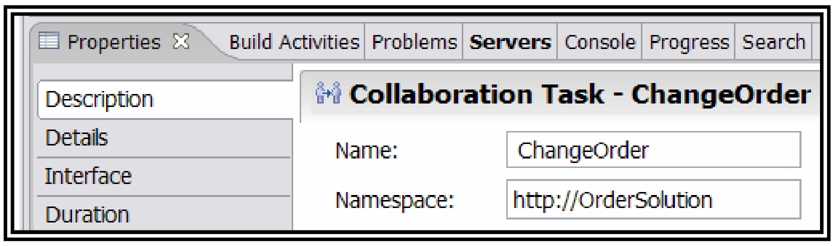
Figure 17: WSDL example of QNameOfWSDLMessage

According to the figure above:
QNameOfWSDLInputMessage
`{http://OrderSolution/OrderApproval}operation1RequestMsg`

QNameOfWSDLOutputMessage
`{http://OrderSolution/OrderApproval}operation1ResponseMsg`

- The HumanTaskID:

  The ID which makes a human task unique when a JSF client is generated for a template. The ID is the concatenation of the namespace of the human task and the template name available in the properties section of the human task editor, see example below:



Collaboration Task - ChangeOrder

Name: ChangeOrder

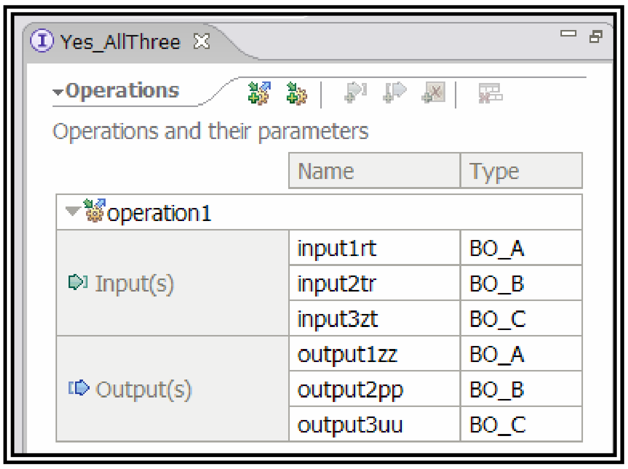
Namespace: http://OrderSolution

Figure 18: Example of HumanTaskID for template based approach

HumanTaskID according the figure above:
`http://OrderSolutionChangeOrder`

- The MessageElementNameOfBO:

  The element name used in the input or output message to wrap a business object, see the following example.



Operations and their parameters

	Name	Type
operation1		
Input(s)	input1rt	BO_A
	input2tr	BO_B
	input3zt	BO_C
Output(s)	output1zz	BO_A
	output2pp	BO_B
	output3uu	BO_C

Figure 19: Example of MessageElementNameOfBO

According to the figure above all values for:
MessageElementNameOfInputBO
`input1rt, input2tr, input3zt`

MessageElementNameOfOutputBO
`output1zz, output2pp, output3uu`

The following sub chapters explain the content of faces-config.xml.

3.2.1 Message handlers and sub pages

The message handlers are responsible for including the correct **sub page** that renders the input or output message of a human task into a **main page**. These message handlers (`bcCreateMessageHandler`, `bcCheckHandler`, `bcDetailsHandler`, `toDoMessage-Handler`, `subToDoMessageHandler`) can be found in the package com.ibm.wbit.tel.client.jsf.handler, see '2.3 Java packages'. Each message handler has the property `subviewMap` which must be initialized to enable the mentioned inclusion of **sub pages**.

```
<managed-property>
    <property-name>subviewMap</property-name>
    <map-entries>
        <map-entry>
            <key>{http://OrderSolution/OrderChange}operation1RequestMsginputJSF</key>
            <value>ToDo_in2_ChangeOrder</value>
        </map-entry>

        . . . .

    </map-entries>
</managed-property>
```

Figure 20: Example of subviewMap initialization

The `value` of a `subviewMap` property is always the file name of the **sub page** to include without the file extension. The `key` is slightly different; its value depends on the message handler for which the `subviewMap` is defined:

- Valid `key` values for `bcCreateMessageHandler`:
    - QNameOfWSDLInputMessage + "inputJSF"
    - Component name of process (as shown in the assembly diagram)

- Valid `key` values for `bcCheckHandler`:
    - Component name of process (as shown in the assembly diagram)
- Valid `key` values for `bcDetailsHandler`, `toDoMessageHandler`, and `subToDoMessageHandler`
    - QNameOfWSDLInputMessage + “inputJSF”
    - QNameOfWSDLOutputMessage + “outputJSF”

Note:

The component name of the process is used as `key` if the **sub page** to include will render input fields of custom properties that are defined on process template.

Refer to ‘4.1 The sub page mechanism’ to learn how the configuration described above is used at runtime.

3.2.2 Lists and queries

The `list` component is described in ‘2.4 Business Process Choreographer tag library’, so in this topic we show how the `list` tags are configured.

The generated JSF client has up to four different `list` tags, and each tag has its own model which means its own list handler (`bcCheckListHandler`, `toDosOpenListHandler`, `toDosUnderWorkListHandler`, `subToDosListHandler`). Each list handler defines which query that should be used to fill the list, the type of the query result, and finally which item listener to invoke regarding click events. The table below shows this information.

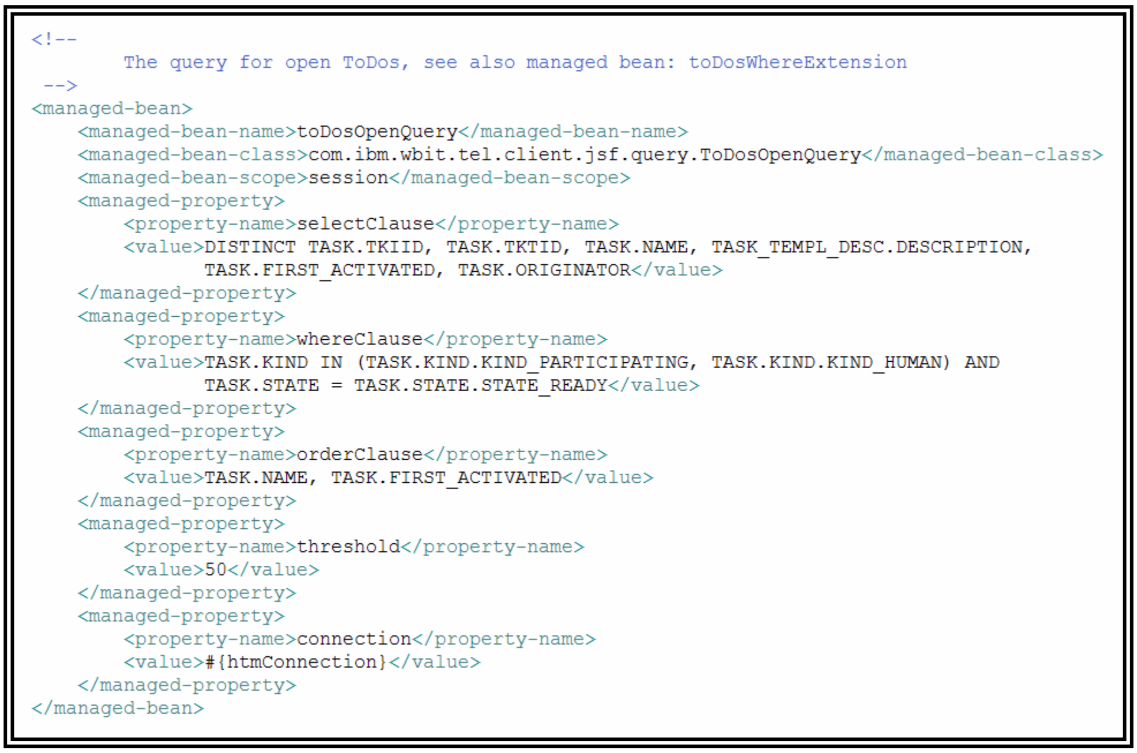
list handler	**query (query class)**	**type (bean class)**	**item listener/s (handler class)**
bcCheckListHandler	bcCheckQuery	BCInstance	bcDetailsHandler BCInstanceDetails
toDosOpenListHandler	toDosOpenQuery	ToDoInstance	toDoMessageHandler
toDosUnderWorkListHandler	toDosUnderWorkQuery	ToDoInstance	toDoMessageHandler
subToDosListHandler	subToDosQuery	ToDoInstance	subToDoMessageHandler

Table 2: Configuration of the list handlers

Remarks on the table above:

- Since each list should show different information, each list hander has its own query (query class, compare to figure 9 in ‘2.3 Java packages’)
- `bcCheckListHandler` has two item listeners
    - o `bcDetailsHandler` known from chapter ‘3.2.1 Message handlers and sub pages’
    - o `BCInstanceDetails` which is defined in faces-config.xml and used in the page BCDetails.jsp as model for a Business Process Choreographer details component, another component of the Business Process Choreographer tag library, refer to [2].
- `toDosOpenListHandler` and `toDosUnderWorkListHandler` use the same item listener since the further processing is similar.

The following figure shows how the query classes are initialized. Although the classes have different queries, they are all initialized in the same way. The following figure shows the initialization of query class `toDosOpenQuery`.



```
<!--
        The query for open ToDos, see also managed bean: toDosWhereExtension
 -->
<managed-bean>
    <managed-bean-name>toDosOpenQuery</managed-bean-name>
    <managed-bean-class>com.ibm.wbit.tel.client.jsf.query.ToDosOpenQuery</managed-bean-class>
    <managed-bean-scope>session</managed-bean-scope>
    <managed-property>
        <property-name>selectClause</property-name>
        <value>DISTINCT TASK.TKIID, TASK.TKTID, TASK.NAME, TASK_TEMPL_DESC.DESCRIPTION,
               TASK.FIRST_ACTIVATED, TASK.ORIGINATOR</value>
    </managed-property>
    <managed-property>
        <property-name>whereClause</property-name>
        <value>TASK.KIND IN (TASK.KIND.KIND_PARTICIPATING, TASK.KIND.KIND_HUMAN) AND
               TASK.STATE = TASK.STATE.STATE_READY</value>
    </managed-property>
    <managed-property>
        <property-name>orderClause</property-name>
        <value>TASK.NAME, TASK.FIRST_ACTIVATED</value>
    </managed-property>
    <managed-property>
        <property-name>threshold</property-name>
        <value>50</value>
    </managed-property>
    <managed-property>
        <property-name>connection</property-name>
        <value>#{htmConnection}</value>
    </managed-property>
</managed-bean>
```

Figure 21: Initialization of query class toDosOpenQuery

As shown in chapter ‘2.3 Java packages’ the class ToDosOpenQuery extends GenericBPCQuery, which is part of the client model. The initialization shown in the figure above set the properties of class GenericBPCQuery.

The `selectClause`, `whereClause` and `orderClause` have the same behaviour as the corresponding SQL queries against a rational data base. To learn about predefined views for queries on human task instances (which columns can be used in the single clauses) refer to [8].

The `threshold` limits the number of query results per request.

In order to execute queries, the class ToDosOpenQuery needs a `connection` to the Human Task Manager Service, which is also configured in faces-config.xml.

See the `whereClause` again; because of `TASK.STATE.STATE_READY` the `list` shows all open ToDos.

The `whereClause` is enhanced by `toDosWhereExtension` to ensure that the only human task instances that are shown in the `list` are the ones for which the JSF client has been generated. The enhancement is done in the query class ToDosOpenQuery, see method getWhereClause(). The initialization of `toDosWhereExtension` is shown below.

```
<!--
    Where clause extension for open and under work ToDos
-->
<managed-bean>
    <managed-bean-name>toDosWhereExtension</managed-bean-name>
    <managed-bean-class>com.ibm.wbit.tel.client.jsf.query.ToDosWhereExtension</managed-bean-class>
    <managed-bean-scope>session</managed-bean-scope>
    <managed-property>
        <property-name>taskNames</property-name>
        <list-entries>
            <value-class>java.lang.String</value-class>
            <value>Approve-Order</value>
            <value>ChangeOrder</value>
            <value>Approve-FollowOn</value>
        </list-entries>
    </managed-property>
    <managed-property>
        <property-name>taskNamespaces</property-name>
        <list-entries>
            <value-class>java.lang.String</value-class>
            <value>http://OrderSolution</value>
            <value>http://OrderSolution</value>
            <value>http://OrderSolution</value>
        </list-entries>
    </managed-property>
</managed-bean>
```

Figure 22: Initialization of class toDosWhereExtension

The figure above shows the initialization of `toDosWhereExtension` with the template name and namespace of all collaboration and to-do human tasks “known” by the generated JSF client, so the query is tailored to them. Since the template names and namespaces are `<list-entries>` the sequence matters. The first list entry of `taskNames` belongs to the first list entry of `taskNamespaces`, second list entry of `taskNames` belongs to second list entry of `taskNamespaces` and so on.

The main points of the other queries that are also defined in faces-config.xml are:

- `toDosUnderWorkQuery`
  - queries for collaboration and to-do human tasks in `TASK.STATE.STATE_CLAIMED`
  - also enhanced by `toDosWhereExtension`
- `subToDosQuery`
  - queries for collaboration and standalone invocation human tasks (these are potential sub tasks)
  - enhanced by `subToDosWhereExtension`
  - correlation to main task is implemented in class SubToDosQuery
- `bcCheckQuery`
  - queries for process instances started via an inline invocation human task
  - search criteria (using custom properties) is implemented in class BCCheckStateQuery

3.2.3 Application meta-info

At runtime the generated JSF client needs certain information which is not directly related to a query against the data base of the business process engine or to a single class of the Java package, it is more likely to be needed in several places. Such information is provided to the generated JSF client by java.util.HashMap classes which are initialized in faces-config.xml below the following comment:

```
================ Application Meta Info ==================
```

Meta-Info HashMap 1:

Name:	`toDosClientTypes`
Purpose:	The generated client could be a hybrid one, which means that in the *User Interface Wizard for Human Tasks* some human tasks have been selected to render their input and output message using JSF and others have been selected to render their input and output message using Lotus Forms. This HashMap is used to distinguish between them.
Key:	HumanTaskID is determined according to template base approach, see '3.2 Configuration file faces-config.xml'

Value:	"JSF" or "FORMS"

Meta-Info HashMap 2:

Name:	`toDosInEqualsOut`
Purpose:	To pre populate the output message with the values from the input message, it is necessary to have a mapping that determines which part of the input message can be used to pre populate which part of the output message. Pre population is only possible between business objects that have the same name and target namespace.
Key:	HumanTaskID + MessageElementNameOfInputBO
Value:	MessageElementNameOfOutputBO

Meta-Info HashMap 3:

Name:	`toDosMainInEqualsSubIn`
Purpose:	To pre populate the input message of a sub task (just before starting it) with the values from the input message of the main task (already running), it is necessary to have a mapping that determines which part of the main task's input message can be used to pre populate which part of the sub task's input message. Pre population is only possible between business objects that have the same name and target namespace.
Key:	HumanTaskID of main task + HumanTaskID of sub task + MessageElementNameOfInputBO of main task
Value:	MessageElementNameOfInputBO of sub task

Meta-Info HashMap 4:

Name:	`toDosSubOutEqualsMainOut`
Purpose:	To pre populate the output message of a main task (just before completing it) with the values from the output message of the sub task (already finished), it is necessary to have a mapping that determines which part of the sub task's output message can be used to pre populate which part of the main task's output message. Pre population is only possible between business objects that have the same name and target namespace.
Key:	HumanTaskID of sub task + HumanTaskID of main task + MessageElementNameOfOutputBO of sub task
Value:	MessageElementNameOfOutputBO of main task

Meta-Info Class:

Name:	`arrayMetaInfo`
Purpose:	An instance of this class contains meta-info about arrays; which is the minimum and maximum number of array entries and the id of nested arrays. The meta-info is set to the instance in **sub page** which renders an input message that contains an array, for example BCCreateNew3_ChangeOrder.jsp. The meta-info is used by the model class ArrayInstance to ensure that the minimum number of entries is shown, the maximum number of entries is not exceeded and nested arrays are also rendered. The initialization of `arrayMetaInfo` is done in the **sub page**s rather than in faces-config.xml to make the sub pages more self contained.

3.2.4 Miscellaneous

The singleton ServiceFactory is provided in the infrastructure Java package of the generated JSF client to make the Business Flow Manager Service and the Human Task Manager Service available. The properties bfmConnection and htmConnections of the ServiceFactory are initialized for local client view or remote client view depending on the selection in *User Interface Wizard for Human Tasks.*

For **main pages** and **infrastructure** pages navigation rules are defined to map the logical name of the JSF page to its physical name. Navigation rules for **sub pages** are not necessary, because their name is computed at runtime via `jsp:include`.

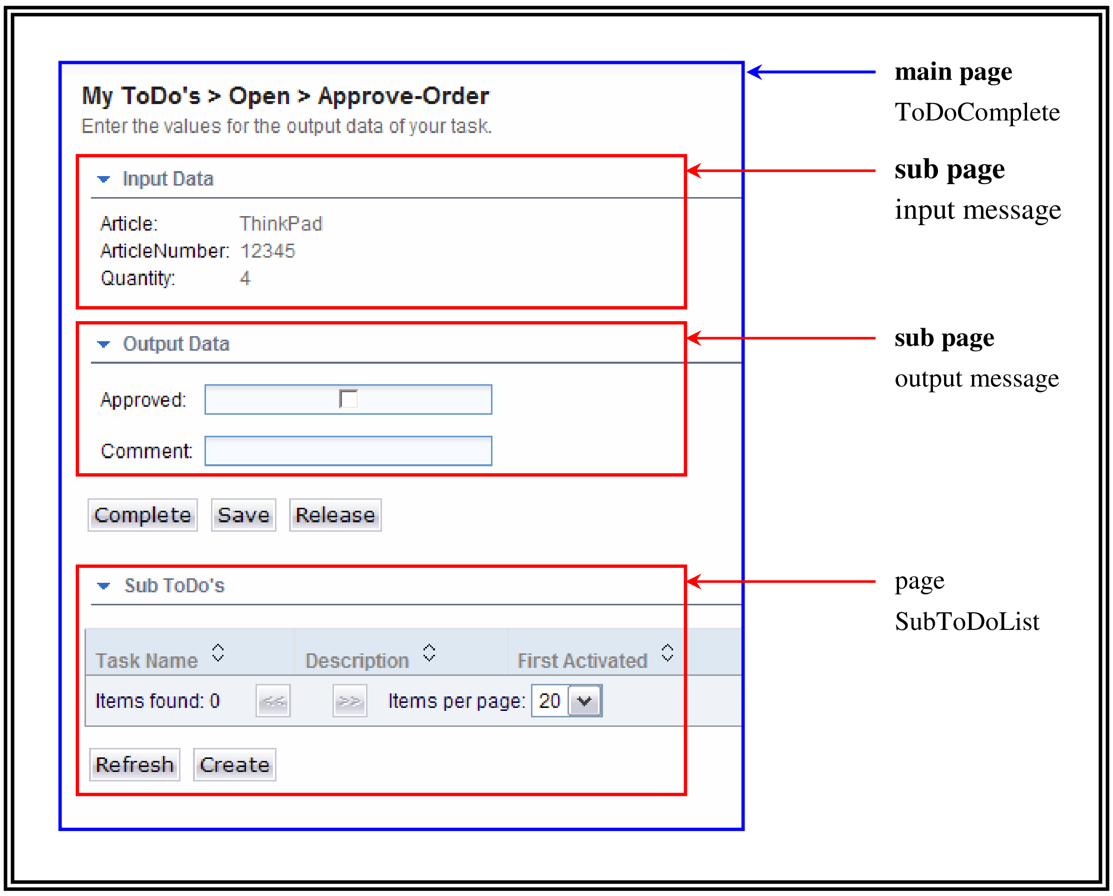
All converters that are used in the generated JSF client are mapped to short converter ids. So each time a converter is necessary for a JSF input or output field it can be referenced by its short id, rather then by its long fully qualified name.

4 The UI for input and output messages

This chapter shows how the **sub page** mechanism works in the generated JSF client and how to set and get values to the JSF input and output fields of sub pages. How to customize the look and feel of the user interface is also shown.

4.1 The sub page mechanism

The **sub pages** contain JSF input and output fields (or more exactly the JSF tags `outputText`, `inputText`, `selectBooleanCheckbox` ...) to render the message of a human task. There is one **sub page** for the input message and another one for the output message. To see how these **sub pages** are included into a **main page**, see the following figure of page ToDoComplete.


My ToDo's > Open > Approve-Order

Enter the values for the output data of your task.

Input Data

Article:	ThinkPad
ArticleNumber:	12345
Quantity:	4

Output Data

Approved:	☐
Comment:	

Sub ToDo's

Task Name	Description	First Activated

Items found: 0 Items per page: 20

Figure 23: Main page ToDoComplete with included sub pages

The page SubToDoList is included if potential sub tasks are available. The inclusion works as follows:

```
<jsp:include page=" SubToDoList.jsp" flush="true" />
```

This example is straight forward since its name is already known at development time, but this is not the case for **sub pages** rendering input and output messages, because:

1. The names of the **sub pages** are generated
2. The **sub page** to be shown depends on the human task instance to complete
3. As mentioned in ‘2.2 JSF pages’ it could be the case that several human task templates share the same message **sub pages**.

To explain how the correct input message **sub page** is found at runtime, lets assume that a click on `toDosUnderWorkListHandler` has occurred. The proper `toDoInstance` is passed to `toDoMessageHandler` which then uses the inherited functionality of the `sub-viewHandler` to determine the **sub page** for the input message, as shown in the figure below.

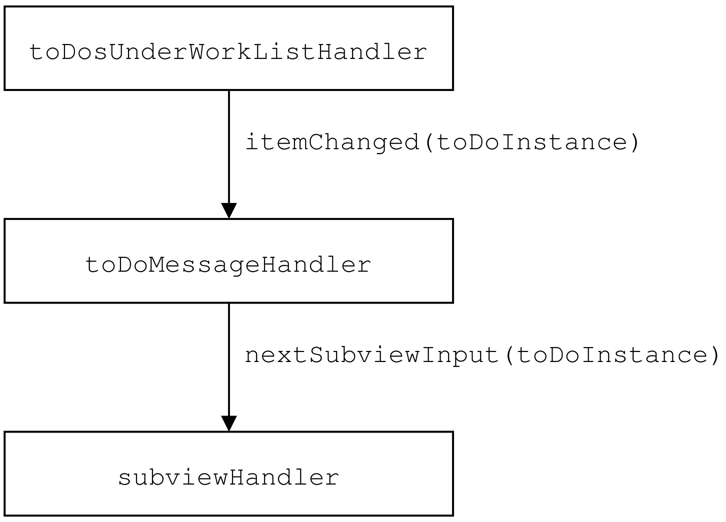

toDosUnderWorkListHandler
itemChanged(toDoInstance)
toDoMessageHandler
nextSubviewInput(toDoInstance)
subviewHandler

Figure 24: Click on list ToDosUnderWork and subsequent flow

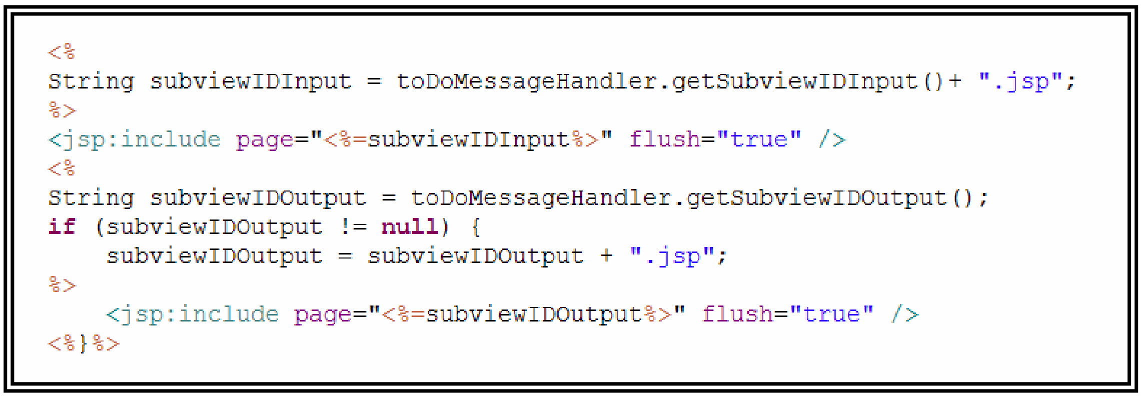
The `subviewHandler` uses the passed `toDoInstance` to get the QNameOfWSDLIn-putMessage via method `getInputMessageTypeName;` this is the first part of the **sub page** key. The middle part of the **sub page** key is obtained form the called method `nex-tSubviewInput`, (see figure above). It is the fixed value “input”. The last part of the **sub page** key is the client type; it is obtained from Meta-Info HashMap `toDosClient-Types,` where the key for this HashMap is the HumanTaskID from `toDoInstance.` For the generated JSF client (no hybrid one, no Lotus Forms) the client type is always

"JSF". Now that the entire **sub page** key has been built it can be used in subviewMap to get the name of the input message **sub page**. The figure below shows this in pseudo code.

```
                        subviewHandler


p1= toDoInstance.getInputMessageTypeName() //QNameOfWSDLInputMessage

p2= "method nextSubviewInput called"       //"input"

p3= toDosClientTypes.get(toDoInstance.getHumanTaskID()) //"JSF"


subviewIDInput = subviewMap.get(p1+p2+p3)
```

Figure 25: Pseudo code to determine the input message sub page

The algorithm for output message **sub pages** is pretty much the same as for input message **sub pages**. Inupt and output message **sub page** are included in **main page** ToDoComplete as follows:



```
<%
String subviewIDInput = toDoMessageHandler.getSubviewIDInput()+ ".jsp";
%>
<jsp:include page="<%=subviewIDInput%>" flush="true" />
<%
String subviewIDOutput = toDoMessageHandler.getSubviewIDOutput();
if (subviewIDOutput != null) {
    subviewIDOutput = subviewIDOutput + ".jsp";
%>
    <jsp:include page="<%=subviewIDOutput%>" flush="true" />
<%}%>
```

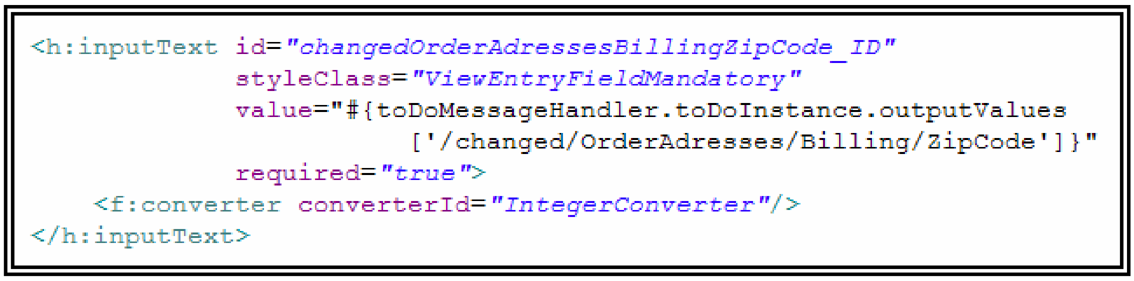
Figure 26: Include statements in main page ToDoComplete

4.2 Set and Get values in sub pages

To show input and output fields, the message **sub pages** contain the JSF tags `outputText`, `inputText`, `dataTable`, `selectBooleanCheckbox` and others. All of these tags have the attribute `value` to set or get a value to them. The values are set to or get from backing beans of the generated JSF client. These beans are in package com.ibm.wbit.tel.client.jsf.bean (see chapter '2.3 Java packages'). Since setting and getting values to and from these beans are done in the same manner for all **sub pages**, the only example shown in this section is for class ToDoInstance. However, setting and getting values to and from these beans is different for non repetitive data structures and for repetitive data structures (arrays), hence it is explained in different chapters.

4.2.1 Non repetitive data structures

As shown in chapter '4.1 The sub page mechanism', the ToDoMessageHandler is involved each time message **sub pages** are shown in the context of ToDo's, therefore it has the property toDoInstance. The class ToDoInstance has the properties inputValues and outputValues, both of type java.util.HashMap. In the case of non repetitive data structures, values to these HashMaps can be set or get directly by the JSF fields by providing a HashMap key in the above mentioned attribute `value`. See the following example for the HashMap outputValues, it is from page ToDo_out2_ChangeOrder.jsp.

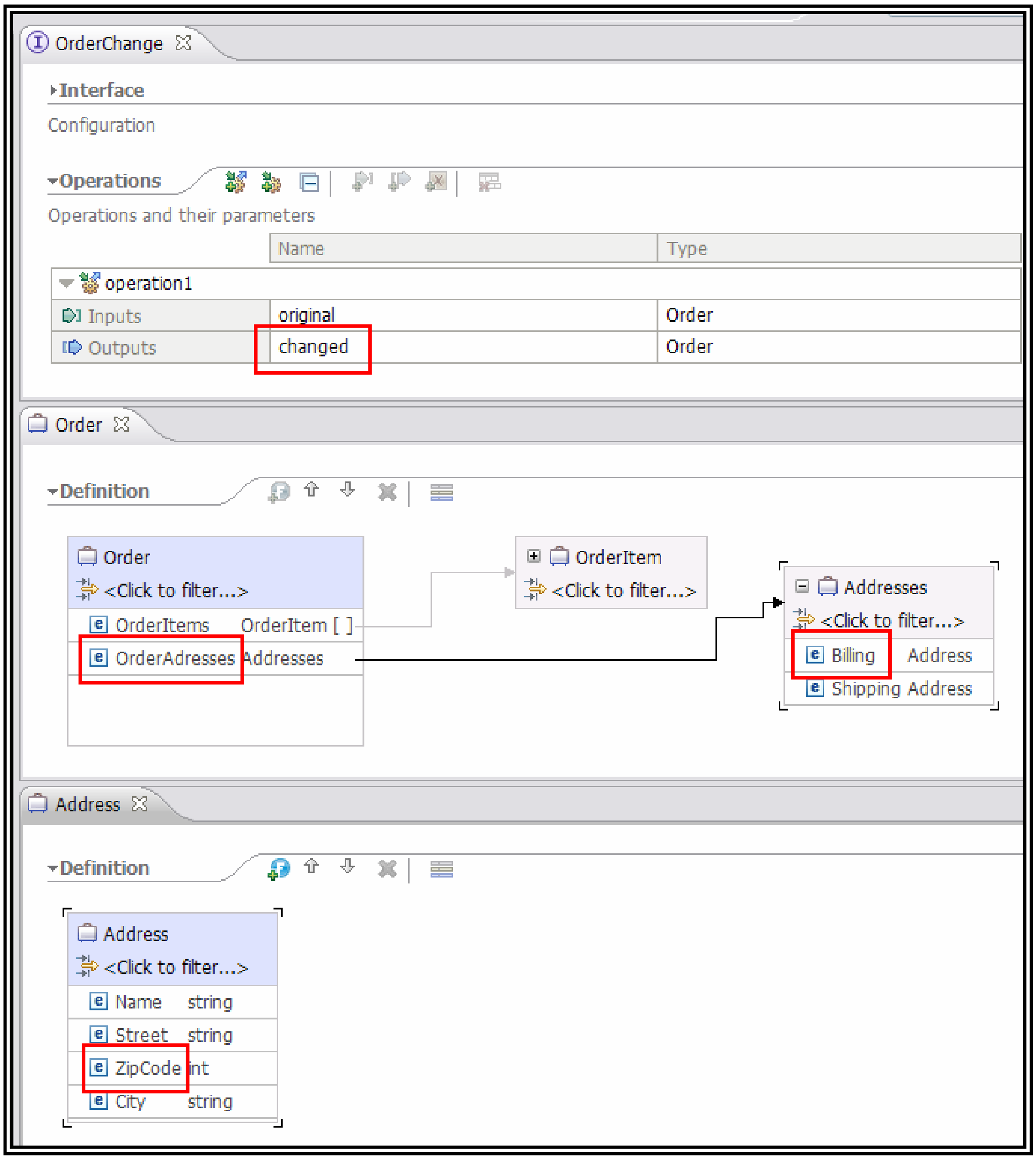


```
<h:inputText id="changedOrderAdressesBillingZipCode_ID"
             styleClass="ViewEntryFieldMandatory"
             value="#{toDoMessageHandler.toDoInstance.outputValues
                         ['/changed/OrderAdresses/Billing/ZipCode']}"
             required="true">
    <f:converter converterId="IntegerConverter"/>
</h:inputText>
```

Figure 27: Get and set value to inputText for non repetitive data structure

In the figure above, the HashMap key is shown in brackets starting with a slash. If the page is rendered, the value from HashMap outputValues mapping to this key will be shown. If the page is submitted, the current value of the `inputText` will be set to HashMap outputValues using the key shown above.

The HashMap key is an Xpath expression. It is the Xpath from the root of the business object tree to the leaf data primitive to be shown by the JSF field. It always starts with a slash. For messages using the document literal wrapped style, the first part of the Xpath (see figure above, the "`changed`" in attribute `value`) comes from the wrapper element, hence it can be found in the interface WSDL. The figure below shows the interface and the business objects for which the Xpath shown in the figure above has been generated.



	Name	Type
operation1		
Inputs	original	Order
Outputs	changed	Order

Order	
<Click to filter...>	
OrderItems	OrderItem [ ]
OrderAdresses	Addresses

OrderItem
<Click to filter...>

Addresses	
<Click to filter...>	
Billing	Address
Shipping	Address

Address	
<Click to filter...>	
Name	string
Street	string
ZipCode	int
City	string

Figure 28: The parts for the Xpath expression for non repetitive data structure

To see further examples of Xpath expressions just generate a JSF client for human tasks that use the business objects you are interested in and look into the appropriate message **sub page**.

As shown above the values of the JSF fields will be set to or get from a HashMap. This HashMap is not required to be type safe according to J2SE 5, but the HashMap values must be of the right type, the right Java primitive. The converter tag is used to ensure compliance with this rule. For more information on converters, see figure 27 and also the faces-config chapter ‘3.2.4 Miscellaneous’. The conversion of the Java primitives to proper Service Data Objects (SDOs) which are applicable for the business process engine and vice versa is done in the class ToDoInstance; it uses the class DataObjectUtils from the Business Process Choreographer tag library to do the above mentioned conversions.

4.2.2 Repetitive data structures - Arrays

The way to set and get values is similar for repetitive and non repetitive data structures. The values are set or get to a java.util.HashMap, but for each repetition a separate HashMap is necessary. Hence the data model for a repetitive data structure is a java.util.ArrayList of HashMaps. This ArrayList is available as a property in class com.ibm.wbit.tel.client.jsf.bean.ArrayInstance, see method getValues. Instances of ArrayInstance are available in property inputValuesArray and outputValuesArray of class ToDoInstance. These two properties are HashMaps since a message **sub page** might have several arrays defined on it.

The figure below shows in pseudo code how to access array output values.

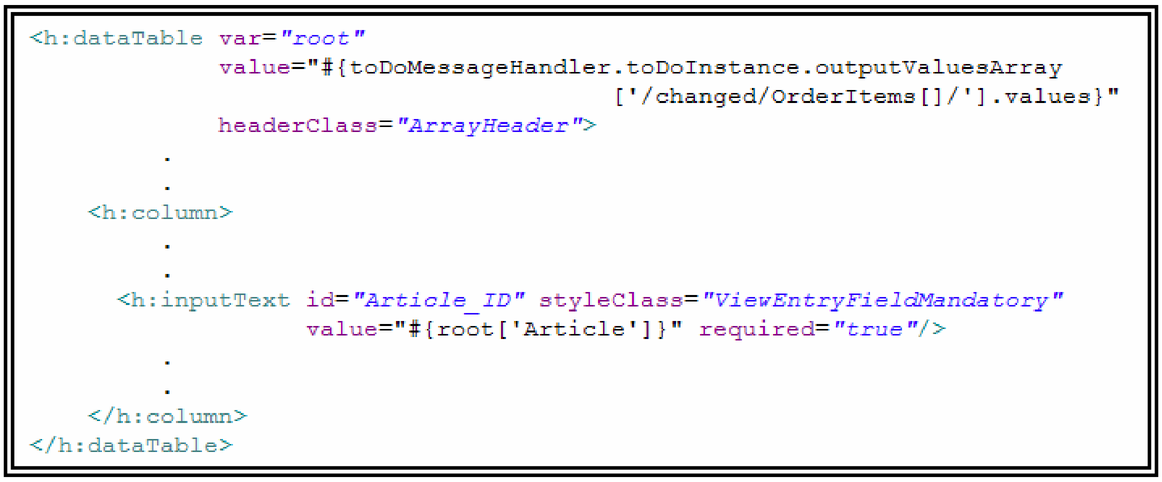
```
1.) toDoInstance.getOutputValuesArray()

      → HashMap<String, ArrayInstance>

      all array instances for a message sub page (output values)


2.) anArrayInstance.getValues()

      → ArrayList<HashMap<String, Object>>

      all output values for an array instance, the list contains a
      HashMap for each data repetition.
```

Figure 29: Pseudo code to set and get output values of a repetitive data structure

In step 1, see figure above, a HashMap is returned with all array instances for a message **sub page**. To get a specific array instance a key is passed to this HashMap in the message **sub page**, see figure below.

In step 2 an ArrayList is returned. It contains a HashMap for each data repetition, so each HashMap contains the values for the data fields of a repetiton.

The following figure shows how the values are set or get for a repetitive data structure in the generated JSF client, it is from page ToDo_out2_ChangeOrder.jsp.

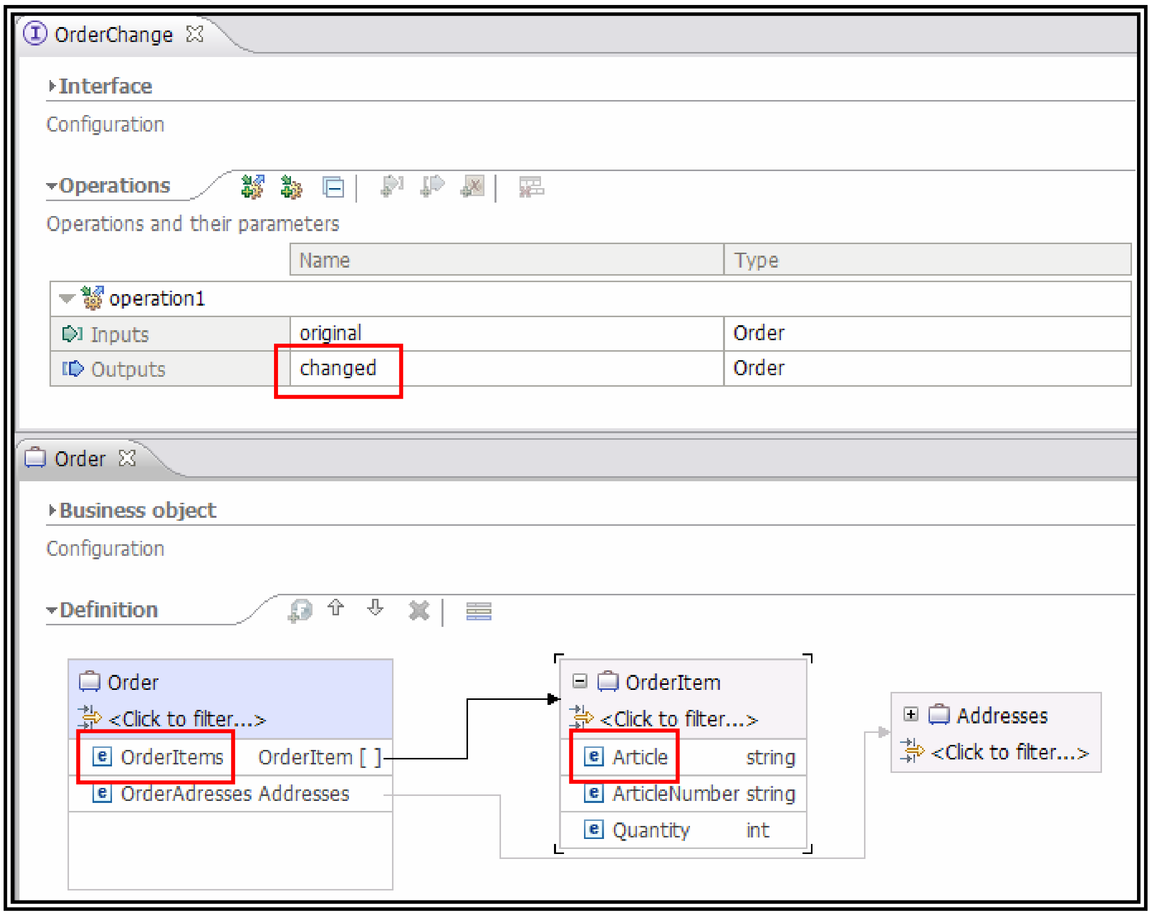


```
<h:dataTable var="root"
             value="#{toDoMessageHandler.toDoInstance.outputValuesArray
                                         ['/changed/OrderItems[]/'].values}"
             headerClass="ArrayHeader">
          .
          .
    <h:column>
          .
          .
      <h:inputText id="Article_ID" styleClass="ViewEntryFieldMandatory"
                   value="#{root['Article']}" required="true"/>
          .
          .
    </h:column>
</h:dataTable>
```

Figure 30: Get and set value to inputText for repetitive data structure

The JSF tag `dataTable` is suitable to handle a repetitive data structure. For this purpose the repetitive data structure must be available to the attribute `value` to enable the `dataTable` to iterate over it. Each repetition is then available to the attribute `var`. See figure above, the array key “`/changed/OrderItems[]/`” is used to get the proper array instance. To get all values of this array instance the “`.values`” is used. A data repetition is available to the local variable “`root`”, and the data primitives of this repetition can be got using a primitive key, e.g. “`Article`”.

The figure below shows the interface and the business objects for which the Xpath expressions shown in the figure above have been generated.



	Name	Type
operation1		
Inputs	original	Order
Outputs	changed	Order

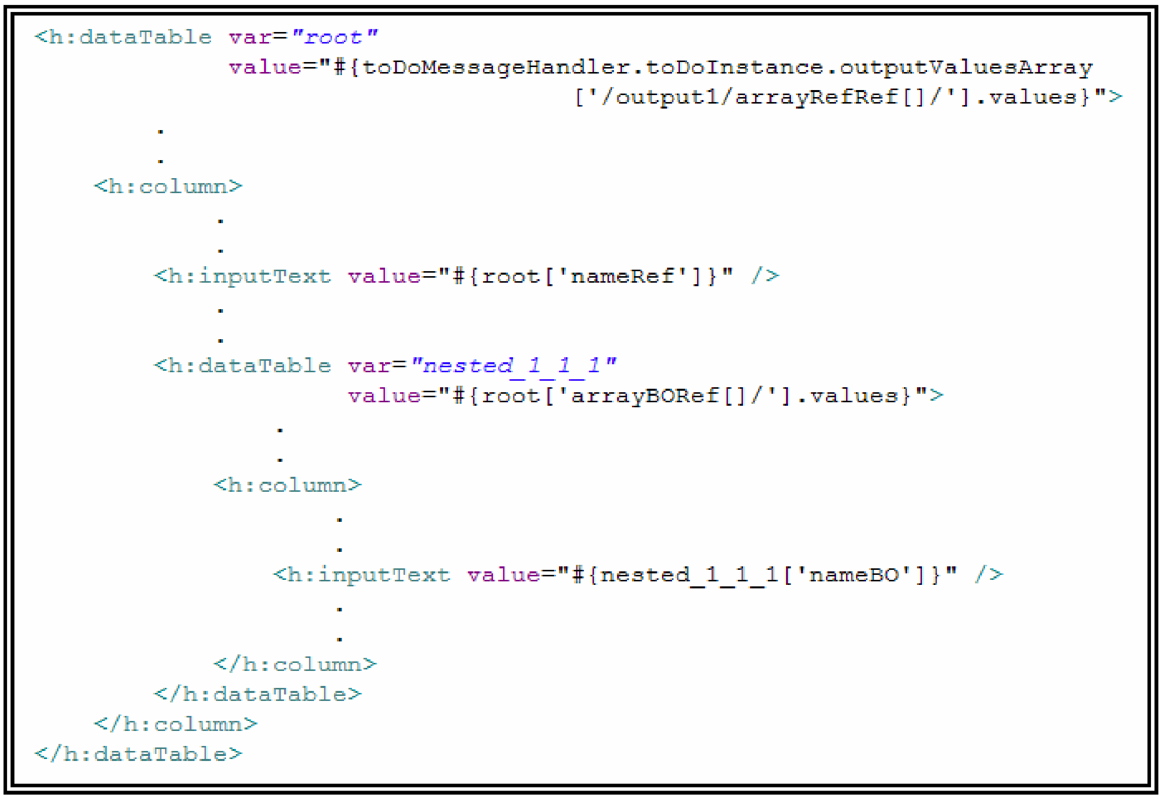
Figure 31: The parts for the Xpath expression for repetitive data structure

The above mentioned array key "`/changed/OrderItems[]/`" and the primitive key "`Article`" are relative keys; they identify the data at a certain level of nesting (nesting regarding arrays). The opposite of relative keys are absolute keys; they identify the data from the root of the business object tree to the leaf data primitive, e.g. "`/changed/OrderAdresses/Billing/ZipCode`". Absolute keys must be used To exchange data with the business process engine. The conversion from relative to absolute keys and vice versa is implemented in class ArrayInstance, see methods transformAbsoluteToRelative and transformRelativeToAbsolute.

The class ArrayInstance can do the following:

- add and remove repetitions (the message **sub page** provides buttons for these)
- ensure that the number of minimum occurrences for a repetition is met
- ensure that the number of maximum occurrences for a repetition is not exceeded
- handle repetitive data structures within repetitive data structures, or arrays of arrays.

In a message **sub page,** nested arrays are defined as nested `dataTable`s. The figure below shows the definition for an array of arrays.



```
<h:dataTable var="root"
             value="#{toDoMessageHandler.toDoInstance.outputValuesArray
                                   ['/output1/arrayRefRef[]/'].values}">
        .
        .
    <h:column>
            .
            .
        <h:inputText value="#{root['nameRef']}" />
            .
            .
        <h:dataTable var="nested_1_1_1"
                     value="#{root['arrayBORef[]/'].values}">
                .
                .
            <h:column>
                    .
                    .
                <h:inputText value="#{nested_1_1_1['nameBO']}" />
                    .
                    .
            </h:column>
        </h:dataTable>
    </h:column>
</h:dataTable>
```

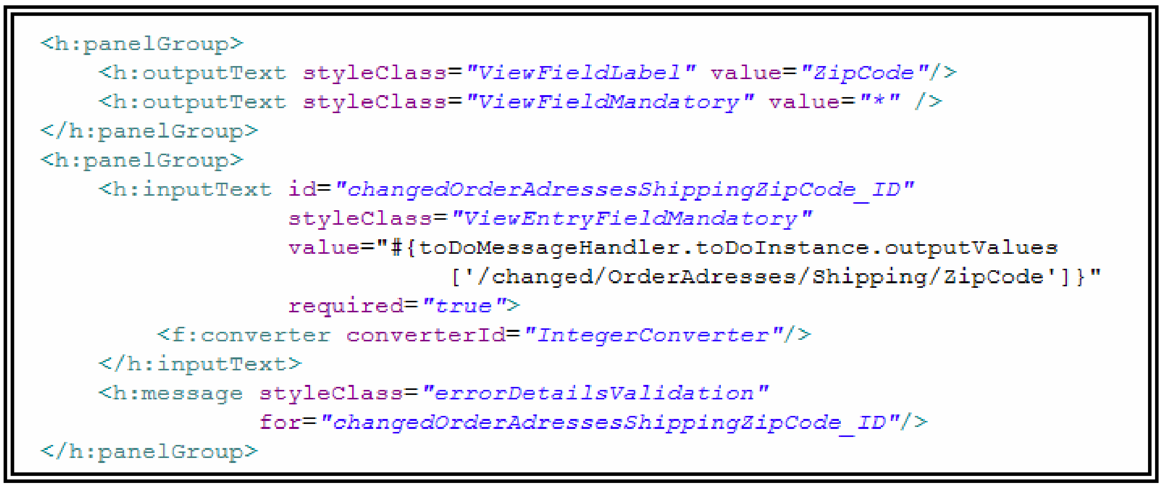
Figure 32: Get and set value to inputText for nested repetitive data structure

The figure above shows how the `dataTable`s are nested in the case of nested repetitive data structures. To get all values of a nested array instance, its array key is used on a repetition of the outer array, see the attribute `value` of nested `dataTable` which it has the value “`#{root['arrayBORef[]/'].values}`”.

Setting and getting values for nested repetitive data structures works in the same way as for non nested repetitive data structures, refer to figure 30.

4.3 Changing the look and feel of the user interfaces

The sequence in which the JSF fields occur on the page, their labels, their grouping, their colours, how much they are indented and so on does not affect the function of the generated JSF client from a technical point of view. They can be customized according to the business users needs.



```
<h:panelGroup>
    <h:outputText styleClass="ViewFieldLabel" value="ZipCode"/>
    <h:outputText styleClass="ViewFieldMandatory" value="*" />
</h:panelGroup>
<h:panelGroup>
    <h:inputText id="changedOrderAdressesShippingZipCode_ID"
                 styleClass="ViewEntryFieldMandatory"
                 value="#{toDoMessageHandler.toDoInstance.outputValues
                            ['/changed/OrderAdresses/Shipping/ZipCode']}"
                 required="true">
        <f:converter converterId="IntegerConverter"/>
    </h:inputText>
    <h:message styleClass="errorDetailsValidation"
               for="changedOrderAdressesShippingZipCode_ID"/>
</h:panelGroup>
```

Figure 33: JSF input field with description

In the figure above, the value of the attributes `id`, `value` (of tag `inputText`) and `for` should not be changed, since they are used for reference purposes.

To change the label of the JSF field, change the values of the values attributes of the tag `outputText`.

The value of all the attributes of `styleClass` attributes can be changed, however the referenced style classes must exist in the cascading style sheet.

If a different converter is to be used it must be available for the generated JSF client, and the value of attribute `converterId` must be changed appropriately.

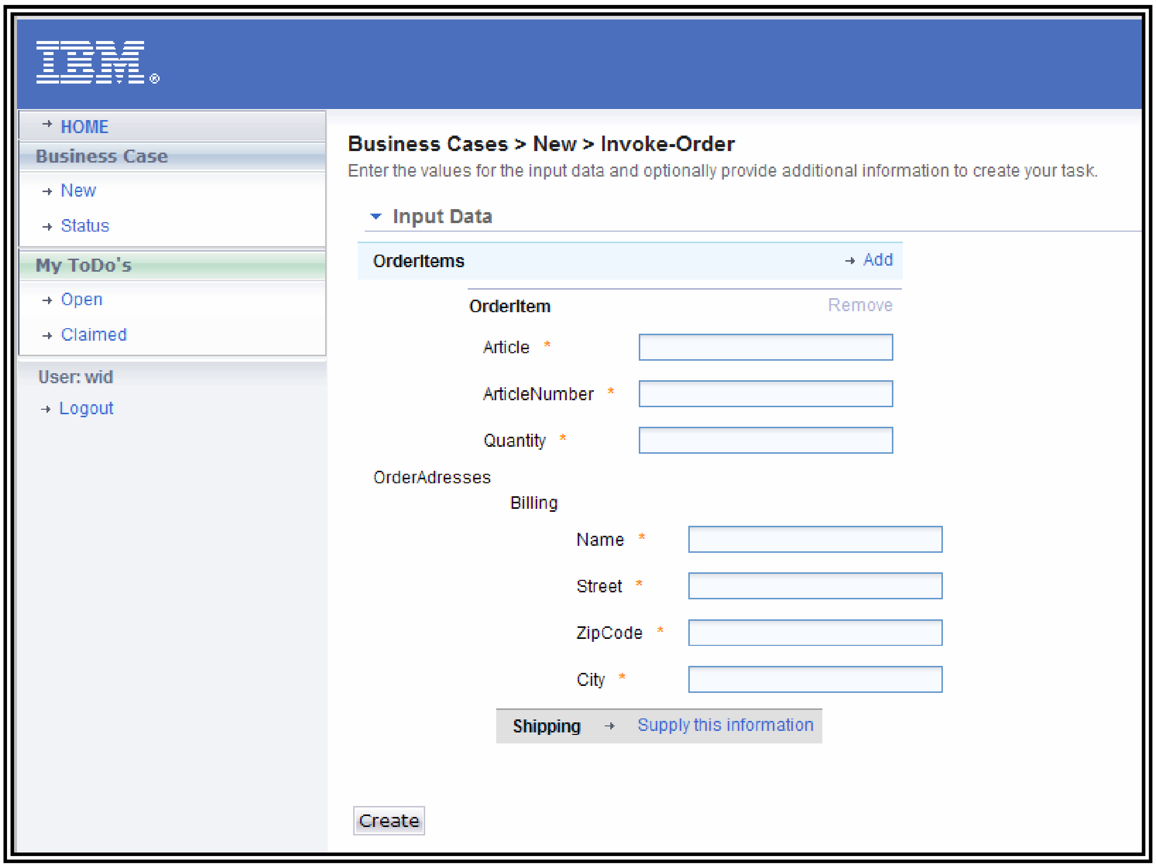
The value of attribute `required` is directly derived from the business object's leaf data primitive. If the value of this attribute is changed from `true` to `false`, but the data primitive is required according to its definition, a default value will be set at runtime by the Business Process Choreographer tag library for this data primitive.

If the position of the tag `inputText` is changed, the position of the corresponding tag `message` (see value of attributes `id` and `for`) should also be changed. The reason for this is that the error message should occur near the `inputText` that caused the error.

The message **sub page** may be modified in the Design view of WebSphere Integration Developer's Page Designer. Be aware that sometimes Page Designer automatically surrounds JSF tags with a further JSF `form` tag. This will cause problems at runtime, since the **main page** already surrounds the entire **sub page** with a JSF `form` tag. Hence the

JSF `form` tag automatically added by Page Designer should be removed, see [5] for further information on this.

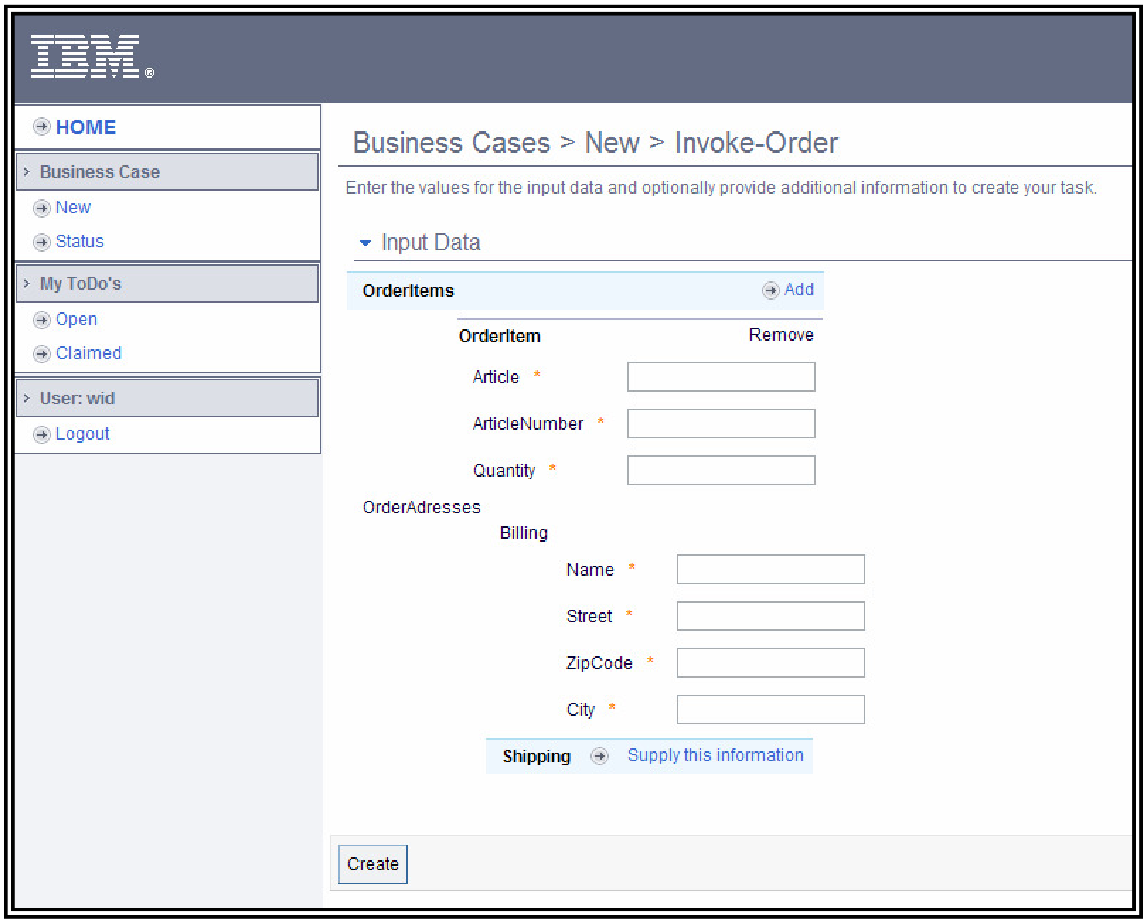
Major changes to the look and feel of the user interface can be achieved by changing the cascading style sheet of the generated JSF client. In the figures below, the changes are due to different cascading style sheets.



Business Cases > New > Invoke-Order

Enter the values for the input data and optionally provide additional information to create your task.

Figure 34: Generated JSF client with the default look and feel



Business Cases > New > Invoke-Order

Enter the values for the input data and optionally provide additional information to create your task.

Input Data

OrderItems Add

OrderItem Remove

Article *

ArticleNumber *

Quantity *

OrderAdresses

Billing

Name *

Street *

ZipCode *

City *

Shipping Supply this information

Create

Figure 35: Generated JSF client with a changed look and feel

The cascading style sheet used by the generated JSF client is the file styles.css; it is placed in the dynamic web project in WebContent\theme. It contains style class definitions that are referenced by JSF tags `outputText`, `inputText`, `selectBooleanCheckbox` and others. The properties of these style classes (colour, width, margin . . .) can be changed to design a new look and feel for the generated JSF client. The most common style classes and to which parts of the generated JSF client they apply is shown in the next chapter. Additional style classes may be introduced and referenced by the previously mentioned JSF tags in order to fine tune the design.

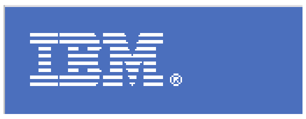
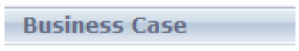
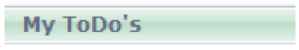
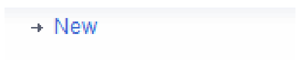
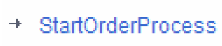
The file styles.css also contains style classes which are referenced by the Business Process Choreographer tag library to define the look and feel of components, such as:

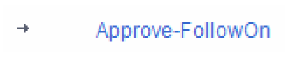
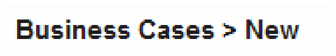
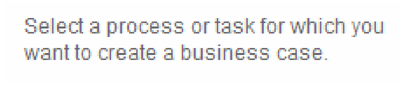
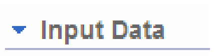
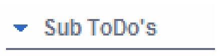
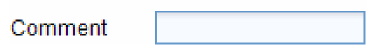
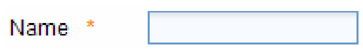
- `commandbar`
  see comment `/* Styles for Table Lists */` in file styles.css
- `list`
  see comment `/* Styles for Command Bar */` in file styles.css

To change the look and feel of the Business Process Choreographer tag library components, the properties of the style classes should be changed; introducting new style classes will not work. To get more information about the style classes available for the Business Process Choreographer tag library components refer to [1].

4.3.1 Most common style classes

The following table lists the most common style classes of the generated JSF client and the UI parts to which they apply.

UI part	Description	Style class
	Sets the background color of the banner (the image can be selected in the client generation wizard)	`Banner`
	Sets the background color and font style of the function block "Business Case"	background: `BackgroundBC` label: `NavigationLabelBC`
	Sets the background color and font style of the function block "My ToDo's"	background: `BackgroundMTD` label: `NavigationLabelMTD`
	Sets the link style in • function blocks • array header • array entry • optional nested business objects	`NavigationLink`
	Sets the link style to start a process	`NavigationViewLink`

	Sets the link style to start a task	`Indentend`
	Sets the style for the navigation information on the page	`ViewHeadingBC`
 Select a process or task for which you want to create a business case.	Sets the style for the description on a page	`SelectStatement`
	Sets the head line style in the "Business Case" section for • input message • processes • tasks	`ViewLabelBC`
	Sets the head line style in the "My ToDo's" section for • input message • output message • sub todos	`ViewLabelMTD`
	Sets the label and input field for optional data	label: `ViewFieldLabel` input field: `ViewEntryField`
	Sets the label, asterisk and input field for mandatory data	label: `ViewFieldLabel` asterisk: `ViewFieldMandatory` input field: `ViewEntryFieldMandatory`

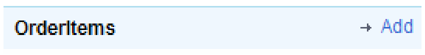
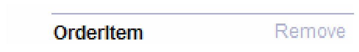
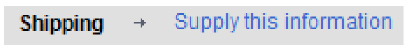
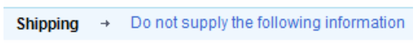
 OrderItems → Add	Sets the the array header style	background: `ArrayHeader` label: `ArrayLabel`
 OrderItem Remove	Sets the array entry style	background: `ArrayEntryHeader` label: `ArrayLabel`
 Shipping → Supply this information	Sets the header for optional nested business object when collapsed	background: `ArrayHeader.Passive` label: `ArrayLabel`
 Shipping → Do not supply the following information	Sets the header for optional nested business object when expanded	background: `ArrayHeader` label: `ArrayLabel`

Table3: UI parts of generated JSF client and their style classes

5 Enhancement scenarios for the generated client

If a generated JSF client has already been customized for the business user needs and afterwards the scope of work for this client work is increased, then it might be better to enhance the already customized client to support this increased scope rather than to generate a new client for the increased scope and do the same customization again. This chapter shows some enhancement scenarios.

5.1 Add support for a new human task

A common step to be done each time when support for a new human task needs to be added is to update some parts of the faces-config.xml of the client to be enhanced, see also '3.2 Configuration file faces-config.xml'. This should be done as follows:

1. Generate a new JSF client for all the human tasks for which the enhanced client should work; the existing tasks, and the new task
2. Open the newly generated faces-config.xml and the faces-config.xml of the client to be enhanced
3. In the faces-config.xml of the client to be enhanced, replace the following sections completely with the corresponding sections from the newly generated faces-config.xml
    a. `Where clause extension for open and under work ToDos`
    it contains the configuration of `toDosWhereExtension`
    b. `Where clause extension for sub ToDos`
    it contains the configuration of `subToDosWhereExtension`
    c. `Application Meta Info`
    it contains the configuration of `toDosClientTypes, toDosInEqualsOut, toDosMainInEqualsSubIn` and `toDosSubOutEqualsMainOut`
    d. `Navigation rules`
    it contains the mapping of logical name to file name of all **infrastructure-, overview-** and **main pages**.
4. Save the faces-config.xml of the client to be enhanced
5. The last step is to answer the question whether all required message **sub pages** to support the new human task are already available in the existing client (available due to the initial generation of it). Specifically:
    a. If the new human task implements the same interface as a human task for which the existing client has been generated, and both human tasks are of the same type (invocation, collaboration or to-do), then consider sub chapter '5.1.1 Sub pages for input- / output message already available'.

b. If none or not all of the required message **sub pages** are available in the existing client, consider sub chapter '5.1.2 Sub page for input- / output message not available'.

5.1.1 Sub pages for input- / output message already available

In these enhancement scenarios the existing message **sub pages** can be re-used, some **overview** and **main pages** as well as parts of faces-config.xml must be enhanced, depending on the human task type to be added.

5.1.1.1 Invocation task (standalone)

In the following enhancement scenario we assume that the standalone invocation task *Invoke-Order* from the scenario (see '1.1 Scenario') is being added to an existing client. Lets assume that the client to be enhanced has already another standalone invocation task (named *ExistingSI*) using the same WSDL interface.

Enhancement in pages:

In BCCreate.jsp we need a further `commnadLink:`

```
<h:commandLink action="BCCreateNew"
               actionListener="#{bcCreateMessageHandler.itemSelected}">
        <h:outputText styleClass="Indentend" value="Invoke-Order"/>
        <f:attribute name="componentName" value="Invoke-Order"/>
        <f:attribute name="inputMessage"
                     value="{http://OrderSolution/Order}operation1RequestMsginputJSF"/>
</h:commandLink>
```

The `componentName` is the name of the *Invoke-Order* as shown in the assembly editor. The `inputMessage` is the QNameOfWSDLInputMessage + "inputJSF" of *Invoke-Order;* it is the same `inputMessage` as that of *ExistingSI.*

Since *Invoke-Order* is a potential sub task we need to enhance SubToDoCreate.jsp, this page is only available if the client to be enhanced has at least one main task. Enhance the `<h:selectOneMenu` in page SubToDoCreate.jsp with the following:

```
<f:selectItem itemValue="2"  itemLabel="Invoke-Order"/>
<f:attribute name="componentName2" value="Invoke-Order"/>
<f:attribute name="inputMessage2"
value="{http://OrderSolution/Order}operation1RequestMsginputJSF"/>
```

All above shown occurrences of number `2` must be replaced by the correct number count for `<f:selectItem` according to the actual `<h:selectOneMenu` of the client to be enhanced.

Enhancements in faces-config.xml:

1. Common steps as described in '5.1 Add support for a new human task'.
2. Managed bean `bcCreateMessageHandler`, property `componentNamespaces` needs a further entry, see below.

```
<map-entry>
      <key>Invoke-Order</key>
      <value>http://OrderSolution</value>
</map-entry>
```

The `key` is the name of the *Invoke-Order* as shown in the assembly editor; the `value` is its namespace as defined in human task editor (see figure 18).

5.1.1.2 Invocation task (inline)

In the following enhancement scenario lets assume that support for a further business process should be added to an existing client. As shown in '2 Structure of the generated client' business processes are started by the generated JSF client via an inline invocation task defined on the starting Receive activity. So for the following enhancement scenario lets assume that we are adding support to start and check the state of business process instances of the *OrderProcess*. Hence we need to add support for the inline invocation task defined on the starting Receive activity *StartOrderProcess* (see '1.1 Scenario'). Lets also assume that the client to be enhanced has already another business process also started via an inline invocation task (named *ExistingII*), using the same WSDL interface.

New pages necessary:

If custom properties are defined on the process template and they should be used in the enhanced client as search criteria for process instances, then proper **sub pages** must be generated for those custom properties and copied to the WebContent folder of the client to be enhanced. To get the **sub pages** for the custom properties generate a new JSF client for *OrderProcess* (for inline invocation task *StartOrderProcess*); in the new generated JSF client in folder WebContent you will find the pages BCCheck1_OrderProcess_CP.jsp and BCCreate1_OrderProcess_CP.jsp.

Enhancements in pages:

In BCCheck.jsp we need a further `commnadLink`:

```
<h:commandLink action="BCCheckStatus"
               actionListener="#{bcCheckHandler.itemSelected}">
   <h:outputText styleClass="NavigationViewLinkLeft" value="OrderProcess"/>
   <f:attribute name="componentName" value="OrderProcess"/>
   <f:attribute name="searchSubview" value="BCCheck1_OrderProcess_CP"/>
</h:commandLink>
```

The `componentName` is the name of the *OrderProcess* as shown in the assembly editor. The `searchSubview` is the name of the previously generated and copied **sub page** for the custom properties.

In BCCreate.jsp we need a further `commandLink`:

```
<h:commandLink action="BCCreateNew"
               actionListener="#{bcCreateMessageHandler.itemSelected}">
 <h:outputText styleClass="NavigationViewLink" value="StartOrderProcess"/>
 <f:attribute name="componentName" value="OrderProcess"/>
 <f:attribute name="inputMessage"
            value="{http://OrderSolution/Order}operation1RequestMsginputJSF"/>
 <f:attribute name="startName" value="StartOrderProcess"/>
</h:commandLink>
```

The `componentName` is the name of the *OrderProcess* as shown in the assembly editor. The `inputMessage` is the QNameOfWSDLInputMessage + "inputJSF" of *OrderProcess;* it is the same `inputMessage` as that of *ExistingII.* The `startName` is the name of the starting Receive activity of *OrderProcess.*

Enhancements in faces-config.xml:

1. Common steps as described in '5.1 Add support for a new human task'.
2. Managed bean `bcCreateMessageHandler`, property `subviewMap` needs a further entry, see below.

   ```
   <map-entry>
         <key>OrderProcess</key>
         <value>BCCreate1_OrderProcess_CP</value>
   </map-entry>
   ```

   The `key` is the name of the *OrderProcess* as shown in the assembly editor; the `value` is the name of the **sub page** for the custom properties.
3. Managed bean `bcCreateMessageHandler`, property `componentNamespaces` needs a further entry, see below.

```
<map-entry>
      <key>OrderProcess</key>
      <value>http://OrderSolution</value>
</map-entry>
```

The `key` is the name of the *OrderProcess* as shown in the assembly editor; the `value` is the namespace as defined in the properties section of BPEL editor.

4. Managed bean `bcCheckHandler`, property `subviewMap` needs a further entry, see below.

```
<map-entry>
      <key>OrderProcess</key>
      <value>BCCheck1_OrderProcess_CP</value>
</map-entry>
```

The `key` is the name of the *OrderProcess* as shown in the assembly editor; the `value` is the name of the **sub page** for the custom properties.

5.1.1.3 Collaboration task

In the following enhancement scenario we assume that the collaboration task *ChangeOrder* from the scenario (see ‘1.1 Scenario’) should be added to an existing client. Lets also assume that the client to be enhanced already has another collaboration task (named *ExistingC*) using the same WSDL interface.

Enhancement in pages:

In BCCreate.jsp we need a further `commnadLink:`

```
<h:commandLink action="BCCreateNew"
               actionListener="#{bcCreateMessageHandler.itemSelected}">
        <h:outputText styleClass="Indentend" value="ChangeOrder"/>
        <f:attribute name="componentName" value="ChangeOrder"/>
        <f:attribute name="inputMessage"
                value="{http://OrderSolution/OrderChange}operation1RequestMsginputJSF"/>
</h:commandLink>
```

The `componentName` is the name of *ChangeOrder* as shown in the assembly editor. The `inputMessage` is the QNameOfWSDLInputMessage + “inputJSF” of *ChangeOrder;* it is the same `inputMessage` as that of *ExistingC.*

Since *ChangeOrder* is a potential sub task we need to enhance SubToDoCreate.jsp. This page is available in the client to be enhanced because of *ExistingC* which is a potential main task (and potential sub task as well). Enhance the `<h:selectOneMenu` in page SubToDoCreate.jsp with the following:

```
<f:selectItem itemValue="1"  itemLabel="ChangeOrder"/>
<f:attribute name="componentName1" value="ChangeOrder"/>
<f:attribute name="inputMessage1"
value="{http://OrderSolution/OrderChange}operation1RequestMsginputJSF"/>
```

All above shown occurrences of number `1` must be replaced by the correct number count for `<f:selectItem` according to the actual `<h:selectOneMenu` of the client to be enhanced.

<u>Enhancements in faces-config.xml:</u>

1. Common steps as described in ‘5.1 Add support for a new human task’.
2. Managed bean `bcCreateMessageHandler`, property `componentNamespaces` needs a further entry, see below.

   ```
   <map-entry>
         <key>ChangeOrder</key>
         <value>http://OrderSolution</value>
   </map-entry>
   ```

   The `key` is the name of *ChangeOrder* as shown in the assembly editor; the `value` is the namespace as defined in human task editor (see figure 18).

5.1.1.4 To-do task (standalone and inline)

If a standalone to-do task, e.g. *ApproveOrder* from the scenario (see ‘1.1 Scenario’), is added to an existing client which already has a standalone or an inline to-do task using the same WSDL interface, then the enhancement steps to be done are just the common steps described in ‘5.1 Add support for a new human task’. The same enhancement steps apply if an inline to-do task should be added to an existing client.

5.1.2 Sub page for input- / output message not available

In these enhancement scenarios no or not all required message **sub pages** exist in the client to be enhanced to support a new human task. So the **sub page** must be generated and added as well as configured in faces-config.xml. To do this it is necessary to determine the QNameOfWSDLInputMessage and the QNameOfWSDLOutputMessage of the human task to be added. For more information, see ‘3.2 Configuration file faces-config.xml’.

It could be the case that adding a new human task also adds a new task type to the existing client. For example, the existing client has only invocation tasks and now a collaboration task needs to be added. This means consequently a new function (new capability) is introduced and some **overview** and **main pages** must also be added.

5.1.2.1 Invocation task (standalone)

In the following enhancement scenario we assume that the standalone invocation task *Invoke-Order* from scenario (see ‘1.1 Scenario’) should be added to an existing client. First, we have to execute the common steps as described in ‘5.1 Add support for a new human task’, keep both faces-config.xml files open since they are needed in the following steps.

New pages necessary:

Due to this enhancement it may be necessary to add further **overview** or **main pages**, which pages to add can be seen in the faces-config.xml of the client to be enhanced. If the section `Navigation rules` shows any warnings (broken links) then the corresponding page is missing. Missing **overview** or **main pages** can be copied from the new generated JSF client into the client to be enhanced.

Enhancement of function:

If the client to be enhanced does not already have an invocation task (standalone or inline) or a collaboration task, then it is also necessary to enhance the page Content.jsp with the following, which can be obtained from Content.jsp of the newly generated JSF client.

```
<h:panelGrid styleClass="BackgroundBC" columns="1">
      <h:outputText styleClass="NavigationLabelBC"
                    value="#{bundle['NAVIGATION_BUSINESS_CASES']}" />
      <h:commandLink styleClass="NavigationLink" action="BCCreate">
      <h:outputText value="#{bundle['NAVIGATION_CREATE_NEW']}" />
      </h:commandLink>
</h:panelGrid>
```

A further function which could be available due to the introduction of a standalone invocation task is the support of sub tasks, but only if the client to be enhanced has also a collaboration task or a to-do task (standalone or inline). To find out if sub tasks can be supported, refer to the new faces-config.xml and see property `subTaskTemplatesAvailable` of `toDoMessageHandler`. The value of this property in the new faces-config.xml is also valid for the client to be enhanced, so set this value to the client to be enhanced accordingly.

Enhancement for new sub pages:

Determine the QNameOfWSDLInputMessage + "inputJSF" of *Invoke-Order*; for the example here it is {http://OrderSolution/Order}operation1RequestMsginputJSF. Find all subviewMap entries in the new faces-config.xml that have this as key, e.g.

```
<map-entry>
  <key>{http://OrderSolution/Order}operation1RequestMsginputJSF</key>
  <value>BCCreateNew1_StartOrderProcess_Invoke-Order</value>
</map-entry>
```

The map entries will be found for the managed beans `bcCreateMessageHandler`, `bcDetailsHandler` and `subToDoMessageHandler`. All found map entries should be copied to the proper subviewMap in the faces-config.xml of the client to be enhanced, but only if such a key is not already present there.

For all copied map entries the corresponding message **sub page** must also be copied from the newly generated JSF client to the client to be enhanced. The name of the message **sub page** to be copied is the value of the map entry. If there is a name clash, which means that the key already exists in the map, with the **sub page** names in the client to be enhanced then rename the **sub page** to be copied and change the value of the map entry accordingly.

Note:

If the interface of *Invoke-Order* has an output message defined then the above described enhancement must also be done for QNameOfWSDLOutputMessage + "outputJSF".

Enhancement in other pages:

Follow the two enhancement steps described in '5.1.1.1 Invocation task (standalone)'; of course it is not necessary to do the common steps referenced there again.

5.1.2.2 Invocation task (inline)

In the following enhancement scenario we assume that support for a further business process should be added to an existing client. As shown in ‘2 Structure of the generated client’ business processes are started by the generated JSF client via an inline invocation task defined on the starting Receive activity. So for the following enhancement scenario we assume to add support to start and check the state of business process instances of the *OrderProcess*. Hence we need to add support for the inline invocation task defined on the starting Receive activity *StartOrderProcess* (see ‘1.1 Scenario’). First we have to execute the common steps as described in ‘5.1 Add support for a new human task’. Keep both faces-config.xml files open since they are needed in the following steps.

New pages necessary:

Due to this enhancement it may be necessary to add further **overview** or **main pages**, which pages to add can be seen in the faces-config.xml of the client to be enhanced. If section `Navigation rules` shows any warnings (broken links) then the corresponding page is missing. Missing **overview** or **main pages** can be copied from the newly generated JSF client into the client to be enhanced.

Enhancement of function:

If the client to be enhanced does not already has an invocation task (standalone or inline) or a collaboration task, then it is also necessary to enhance the page Content.jsp with the following, which can be obtained from Content.jsp of the newly generated JSF client.

```
<h:panelGrid styleClass="BackgroundBC" columns="1">
       <h:outputText styleClass="NavigationLabelBC"
                     value="#{bundle['NAVIGATION_BUSINESS_CASES']}" />
       <h:commandLink styleClass="NavigationLink" action="BCCreate">
       <h:outputText value="#{bundle['NAVIGATION_CREATE_NEW']}" />
       </h:commandLink>
</h:panelGrid>
```

If the client to be enhanced does not already have a business process with custom properties and support for this must be added now, then a further enhancement of Content.jsp is necessary. It can be obtained from Content.jsp of the newly generated JSF client and put to the `<h:panelGrid` shown above.

```
<h:commandLink styleClass="NavigationLink" action="BCCheck" >
       <h:outputText value="#{bundle['NAVIGATION_CHECK_STATUS']}" />
</h:commandLink>
```

Note:

Inline invocation tasks are not applicable for sub tasks.

Enhancement for new sub pages:

Determine the QNameOfWSDLInputMessage + “inputJSF” of *OrderProcess*; for the example here it is {http://OrderSolution/Order}operation1RequestMsginputJSF. Find all subviewMap entries in the new faces-config.xml that have this as key, e.g.

```
<map-entry>
  <key>{http://OrderSolution/Order}operation1RequestMsginputJSF</key>
  <value>BCCreateNew1_StartOrderProcess_Invoke-Order</value>
</map-entry>
```

The map entries will be found for the managed beans `bcCreateMessageHandler`, `bcDetailsHandler` and `subToDoMessageHandler`. All the map entries that are found should be copied to the proper subviewMap in the faces-config.xml of the client to be enhanced, but only if such a key is not already present there.

For all the map entries that are copied, the corresponding message **sub page** must also be copied from the newly generated JSF client to the client to be enhanced. The name of the message **sub page** to be copied is the value of the map entry. If there is a name clash with the **sub page** names, which means that the key already exists in the map, then rename the **sub page** to be copied and change the value of the map entry accordingly.

Note:

If the interface of *OrderProcess* would also have an output message defined then the above described enhancement must also be done for QNameOfWSDLOutputMessage + “outputJSF”

Enhancement in other pages:

Follow the three enhancement steps described in ‘5.1.1.2 Invocation task (inline)’; of course it is not necessary to do the common steps referenced there again.

5.1.2.3 Collaboration task

In the following enhancement scenario we assume that the collaboration task *ChangeOrder* from scenario (see ‘1.1 Scenario’) is to be added to an existing client. First we have to execute the common steps as described in ‘5.1 Add support for a new human task'. Keep both faces-config.xml files open since they are needed in the following steps.

New pages necessary:

Due to this enhancement it may be necessary to add further **overview** or **main pages**, which pages to add can be seen in the faces-config.xml of the client to be enhanced. If section `Navigation rules` shows any warnings (broken links) then the corresponding page is missing. Missing **overview** or **main pages** can be copied from the newly generated JSF client into the client to be enhanced.

Enhancement of function:

If the client to be enhanced does not already have an invocation task (standalone or inline) or a collaboration task, then it is also necessary to enhance the page Content.jsp with the following, which can be obtained from Content.jsp of the newly generated JSF client.

```
<h:panelGrid styleClass="BackgroundBC" columns="1">
       <h:outputText styleClass="NavigationLabelBC"
                     value="#{bundle['NAVIGATION_BUSINESS_CASES']}" />
       <h:commandLink styleClass="NavigationLink" action="BCCreate">
       <h:outputText value="#{bundle['NAVIGATION_CREATE_NEW']}" />
       </h:commandLink>
</h:panelGrid>
```

If the client to be enhanced does not already have a to-do task (standalone or inline) or a collaboration task, then it is also necessary to enhance the page Content.jsp with the following, which can be obtained from Content.jsp of the newly generated JSF client.

```
<h:panelGrid styleClass="BackgroundMTD" columns="1">
       <h:outputText styleClass="NavigationLabelMTD"
                     value="#{bundle['NAVIGATION_MY_TODOS']}" />
       <h:commandLink styleClass="NavigationLink"
                      action="#{toDoMessageHandler.refreshOpenList}">
              <h:outputText value="#{bundle['NAVIGATION_OPEN']}" />
       </h:commandLink>
       <h:commandLink styleClass="NavigationLink"
                     action="#{toDoMessageHandler.refreshUnderWorkList}">
              <h:outputText value="#{bundle['NAVIGATION_CLAIMED']}" />
       </h:commandLink>
       <h:outputText rendered="#{subToDoMessageHandler.init}" />
</h:panelGrid>
```

A further function which will be available due to the introduction of the collaboration task is the support of sub tasks, hence the property `subTaskTemplatesAvailable` of `toDoMessageHandler` should be set to `true` in the faces-config.xml of the client to be enhanced.

Note:

> If the property `subTaskTemplatesAvailable` originally has a value of `false` and the client to be enhanced already has a standalone invocation task, then the **sub page** necessary to create a new instance of this standalone task as a sub task is missing. This **sub page** can be found in the newly generated JSF client and copied into the client to be enhanced.

Enhancement for new sub pages:

Determine the QNameOfWSDLInputMessage + “inputJSF” of *ChangeOrder*; for the example here it is {http://OrderSolution/OrderChange}operation1RequestMsginputJSF. Find all subviewMap entries in the new faces-config.xml that have this as key, e.g.

```
<map-entry>
   <key>{http://OrderSolution/OrderChange}operation1RequestMsginputJSF</key>
   <value>BCCreateNew2_ChangeOrder</value>
</map-entry>
```

The map entries will be found for the managed beans `bcCreateMessageHandler`, `toDoMessageHandler` and `subToDoMessageHandler`. All map entries that are found should be copied to the proper subviewMap in the faces-config.xml of the client to be enhanced, but only if such a key is not already present there.

For all the map entries that are copied the corresponding message **sub page** must also be copied from the newly generated JSF client to the client to be enhanced. The name of the message **sub page** to be copied is the value of the map entry. If there is a name clash with the **sub page** names, which means that the key already exists in the map, then rename the **sub page** to be copied and change the value of the map entry accordingly.

Note:

> Since the interface of *ChangeOrder* has an output message defined, the above described enhancement must also be done for QNameOfWSDLOutputMessage + “outputJSF”; for the example this is {http://OrderSolution/OrderChange}operation1ResponseMsgoutputJSF

Enhancement in other pages:

Follow the two enhancement steps described in ‘5.1.1.3 Collaboration task’; of course it is not necessary to do the common steps referenced there again.

5.1.2.4 To-do task (standalone and inline)

In the following enhancement scenario we assume that the standalone to-do task *ApproveOrder* from scenario (see ‘1.1 Scenario’) should be added to an existing client.

The same enhancement steps apply if an inline to-do task should be added to an existing client. First we have to execute the common steps as described in ‘5.1 Add support for a new human task’. Keep both faces-config.xml files open since they are needed in the following steps.

New pages necessary:

Due to this enhancement it may be necessary to add further **overview** or **main pages**, which pages to add can be seen in the faces-config.xml of the client to be enhanced. If section `Navigation rules` shows any warnings (broken links) then the corresponding page is missing. Missing **overview** or **main pages** can be copied from the newly generated JSF client into the client to be enhanced.

Enhancement of function:

If the client to be enhanced does not already have a to-do task (standalone or inline) or a collaboration task, then it is also necessary to enhance the page Content.jsp with the following, which can be obtained from Content.jsp of the newly generated JSF client.

```
<h:panelGrid styleClass="BackgroundMTD" columns="1">
      <h:outputText styleClass="NavigationLabelMTD"
                    value="#{bundle['NAVIGATION_MY_TODOS']}" />
      <h:commandLink styleClass="NavigationLink"
                     action="#{toDoMessageHandler.refreshOpenList}">
            <h:outputText value="#{bundle['NAVIGATION_OPEN']}" />
      </h:commandLink>
      <h:commandLink styleClass="NavigationLink"
                    action="#{toDoMessageHandler.refreshUnderWorkList}">
            <h:outputText value="#{bundle['NAVIGATION_CLAIMED']}" />
      </h:commandLink>
      <h:outputText rendered="#{subToDoMessageHandler.init}" />
</h:panelGrid>
```

A further function which could be available due to the introduction of the standalone to-do task is the support of sub tasks, but only if the client to be enhanced also has a collaboration task or a standalone invocation task. To find out if sub tasks can be supported refer to the new faces-config.xml and see property `subTaskTemplatesAvailable` of `toDoMessageHandler`. The value of this property is also valid for the client to be enhanced, so set the property `subTaskTemplatesAvailable` in the client accordingly.

Note:

> If the property `subTaskTemplatesAvailable` originally has a value of `false` and the client to be enhanced already has a collaboration task or a standalone invocation task, then the **sub pages** necessary to create a new instance of this collaboration task or standalone task as a sub task is missing. These **sub pages** can be found in the newly generated JSF client and copied into the client to be enhanced.

<u>Enhancement for new sub pages:</u>

Determine the QNameOfWSDLInputMessage + "inputJSF" of *ApproveOrder*; here it is {http://OrderSolution/OrderApproval}operation1RequestMsginputJSF. Find all subviewMap entries in the new faces-config.xml that have this as key, e.g.

```
<map-entry>
   <key>{http://OrderSolution/OrderApproval}operation1RequestMsginputJSF</key>
   <value>ToDo_in1_Approve-Order</value>
</map-entry>
```

The map entries will be found for the managed beans `toDoMessageHandler` and `subToDoMessageHandler`. All the map entries that are found should be copied to the proper subviewMap in the faces-config.xml of the client to be enhanced, but only if such a key is not already present there.

For all the map entries that are copied, the corresponding message **sub page** must also be copied from the newly generated JSF client to the client to be enhanced. The name of the message **sub page** to be copied is the value of the map entry. If there is a name clash with the **sub page** names, which means that the key already exists in the map, then rename the **sub page** to be copied and change the value of the map entry accordingly.

Note:

> Since the interface of *ApproveOrder* has an output message defined, the above described enhancement must also be done for QNameOfWSDLOutputMessage + "outputJSF"; for the example this is {http://OrderSolution/OrderApproval}operation1ResponseMsgoutputJSF

5.2 Add new functionality as a follow-on task

In this enhancement scenario support for a follow-on task is added to an existing client. A follow-on task is a direct successor of a completed task. It can be of type collaboration task or standalone invocation task, where the output message of the task to complete and the follow-on task must be of the same type; the input messages may differ.

The client to be enhanced in this scenario has been generated previously for *OrderProcess* and *Approve-Order;* see '1.1 Scenario'. It should be enhanced with collaboration task *Approve-FollowOn* in a manner that *Approve-Order* can be completed and *Approve-FollowOn* is directly started with the input message *Approve-Order*. As well, instances of *Approve-FollowOn* should be shown in both lists of function block My ToDo's and it should be possible to work on those instances (claim, complete, save…).

Enhancement of main page:

To complete an instance of *Approve-Order* and directly start an instance of *Approve-FollowOn* a further button is added to the `<bpe:commandbar` on page ToDoComplete.jsp. The added part is shown below:

```
<bpe:commandbar model="#{toDoMessageHandler}" buttonStyleClass="button">
       . . .

<bpe:command label="FollowOn" commandID="FollowOnToDoInstance"
       commandClass="com.ibm.wbit.tel.client.jsf.command.FollowOnSupport"
       context="#{toDoMessageHandler}"/>

       . . .

</bpe:commandbar>
```

In the enhancement above, the added `<bpe:command` is associated with class FollowOnSupport, which is a new class which implements the enhancement using the Human Task Manager Service, see below.

Enhancement of command package:

To get the new class FollowOnSupport the already existing class ToDoComplete has been copied and renamed into package com.ibm.wbit.tel.client.jsf.command. Just the implementation of method execute has been changed as follows:

```
public String execute(List items) throws ClientException {

if (items != null) {
  HumanTaskManagerService htm;
  //get the toDoInstance to be completet
  Object obj = items.get(0);
  if (obj instanceof ToDoInstance) {
    ToDoInstance toDoInstance = (ToDoInstance) obj;
    try {
      htm = ServiceFactory.getInstance().getHumanTaskManagerService();

      String compNameFollowOnTask = "Approve-FollowOn";
      String nameSpaceFollowOnTask = "http://OrderSolution";
      String toDoCompleteID = toDoInstance.getID().toString();

      //get inputmessage of the toDoInstance to be completed
      ClientObjectWrapper cow =  htm.getInputMessage(toDoInstance.getID());
      Object inputForFollowOn = cow.getObject();


      //complete toDoInstance and start instance of follow on ToDo
      htm.completeWithNewFollowOnTask(toDoCompleteID, compNameFollowOnTask,
      nameSpaceFollowOnTask, new ClientObjectWrapper(inputForFollowOn));

      } catch (Exception e) {
          throw new CommandException(NLS_CATALOG,
                              SERVICE_FAILED_TO_COMPLETE_HUMAN_TASK, null, e);
      }
    }
  }
  //to reflect complete and follow on action in that list from which we came
  toDoMessageHandler.refreshLastList();

  //we always return to that list from which we came
  return toDoMessageHandler.getLastList();

}
```

In the java code above, the key point for this enhancement is the method `complete-WithNewFollowOnTask` provided by the Human Task Manager Service, see also [3]. It also needs the component name and name space of the follow-on task (as shown in the assembly diagram).

Enhancement of application meta info:

To show instances of *Approve-FollowOn* in both lists of the function block My ToDo's it is necessary to enhance the properties `taskNames` and `taskNamespaces` of managed bean `toDosWhereExtension` (see '3.2.2 Lists and queries') in faces-config.xml. The added entries are shown below:

```
<managed-property>
       <property-name>taskNames</property-name>
       <list-entries>
              <value-class>java.lang.String</value-class>
              <value>Approve-FollowOn</value>
               . . .
               . . .
       </list-entries>
</managed-property>

<managed-property>
       <property-name>taskNamespaces</property-name>
       <list-entries>
              <value-class>java.lang.String</value-class>
              <value>http://OrderSolution</value>
               . . .
               . . .
       </list-entries>
</managed-property>
```

In the enhancements above, since the properties `taskNames` and `taskNamespaces` are lists the sequence in which their values occur is relevant. This means the position of an entry in `taskNames` must match the position of the corresponding entry in `taskNamespaces`. The values are according to template based approach, compare with figure 18.

Since the *Approve-Order* and *Approve-FollowOn* implement the same interface they can share the message **sub pages** (see '2.2 JSF pages'). To enable this, managed bean `toDosClientTypes` must get a further map entry:

```
<map-entry>
     <key>http://OrderSolutionApprove-FollowOn</key>
     <value>JSF</value>
</map-entry>
```

The key is the HumanTaskID (see '3.2 Configuration file faces-config.xml') according to template based approach, the value is fixed to 'JSF'.

References

[1] Styles used in the Business Process Choreographer Explorer interface

http://publib.boulder.ibm.com/infocenter/dmndhelp/v7r0mx/index.jsp?topic=/com.ibm.websphere.bpc.doc/doc/bpc/r7styles.html

[2] Developing Web applications for business processes and human tasks, using JSF components

http://publib.boulder.ibm.com/infocenter/dmndhelp/v7r0mx/index.jsp?topic=/com.ibm.websphere.bpc.doc/doc/bpc/t7jsf_application.html

[3] IBM WebSphere Application Server™, Release 7 API Specification

http://publib.boulder.ibm.com/infocenter/dmndhelp/v7r0mx/index.jsp?topic=/com.ibm.websphere.wbpmcore.javadoc.doc/web/apidocs/index.html

[4] Embedding a client generator for Business Process Choreographer into WebSphere Integration Developer

http://www-1.ibm.com/support/docview.wss?rs=2308&context=SSQQFK&q1=client&uid=swg27009471&loc=en_US&cs=utf-8&lang=en

[5] Customization of a message JSP in a generated JSF client

http://www-1.ibm.com/support/docview.wss?rs=2308&context=SSQQFK&q1=client&uid=swg21254323&loc=en_US&cs=utf-8&lang=en

[6] Initialize the output message with the input message in a generated JSF client

http://www-1.ibm.com/support/docview.wss?rs=2308&context=SSQQFK&q1=client&uid=swg21254704&loc=en_US&cs=utf-8&lang=en

[7] The life cycle of a Human Task

http://publib.boulder.ibm.com/infocenter/dmndhelp/v7r0mx/index.jsp?topic=/com.ibm.websphere.bpc.doc/doc/bpc/ctasklifecycle.html

[8] Database views for Business Process Choreographer

http://publib.boulder.ibm.com/infocenter/dmndhelp/v7r0mx/index.jsp?topic=/com.ibm.websphere.bpc.doc/doc/bpc/r6bpc_dbviews.html