Implementing Advanced Workflow Patterns in WebSphere Integration Developer and WebSphere Process Server, Part 2: Multiple instance patterns and iteration patterns

Ying Ji Sun
Yu Jie Gu
Jing Wen Cui

October 19, 2010

As highlighted in part 1 of the series, the Workflow Patterns Initiative is widely used or referenced when enterprises want to model their workflow. Its latest 43 workflow patterns post a big challenge to all kinds of BPEL engines.

WebSphere Process Server (WPS) is a powerful business process automation engine with high performance. Customers can implement their workflow patterns easily with WPS and WebSphere Integrated Developer (WID). This article highlights the latest WID v7.0 and WPS v7.0 can not only implement all 43 patterns, but also how to implement them easily.

View more content in this series

Introduction

As discusses in part 1, the Workflow Patterns Initiative (WPI) was established with the aim of delineating the fundamental requirements that arise during business process modeling on a recurring basis and describe them in an imperative way. It is widely used or referenced when enterprises want to model their workflow. In the first WPI version, 20 patterns were delivered in 2003. The number increased to 43 in the latest release. Those patterns post a big challenge to all kinds of BPEL engines. It is very difficult to support all of them. In another word, one BPEL engine which can support those patterns well should gain more customers as its power.

In part 2 of this article, we will introduce 2 categories patterns (Multiple Instance Patterns and Iteration Patterns) in detail.

Advanced branching and synchronization patterns

There are 9 patterns in this category, including:
1. Multiple Instances without Synchronization,
2. Multiple Instances with a Priori Design-Time Knowledge,
3. Multiple Instances with a Priori Run-Time Knowledge,
4. Multiple Instances without a Priori Run-Time Knowledge,
5. Static Partial Join for Multiple Instances,
6. Cancelling Partial Join for Multiple Instances,
7. Dynamic Partial Join for Multiple Instances,
8. Arbitrary Cycles,
9. Structured Loop

These patterns are related with branching and merging concept and widely used in real business process. Some of them can be implemented in WID 7 BPC editor easily, and some involve WID 7 SCA assembly structures.

**Multiple instances without synchronization**

Multiple instances of a task are created in a given process instance. And there is no need to synchronize them upon completion.

For example, suppose we have a task named approveVacationRequest. Multiple instances can be created and handled in the same time. No Synchronization is needed.

**Analysis**

The focus of this pattern is:

1. Multiple instances of a task can be created independently.
2. No need to synchronize upon completion.
3. Each of them must execute independently

**Implementation**
Figure 1. Multiple Instances without Synchronization

This pattern is Supported by <invoke> statement within <foreach>.

Listing 1. The operation1 in MultiInstancesInvokeeImpl

```java
addCounter();
System.out.println("[MultiInstancesInvokee] invoked time is: " + counter);
input1.setInt("field3", counter);
return input1;
```

Test Data

field2=2; Expected result: the SCA component will be invoked twice.

Multiple instances with a priori design-time knowledge

This pattern requires multiple instances which are created within a given process instance to be synchronized upon their completion. And the number of the instances can be known at the design time.

For example, the test plan should be approved by the test lead, component developer, the architect and the peer tester.

Analysis

The focus of this pattern is:

1. Multiple instances are created.
2. The required number is known at design time.
3. Synchronization is required at completion.

**Implementation**

**Figure 2. Multiple Instances with a Priori Design-Time Knowledge**

![Diagram of Multiple Instances with a Priori Design-Time Knowledge]

In the scope, we initialize a variable named `dataCollection` for collecting the data in `setTheResultIntoDataCollection` activity. We use `ForEach` activity to invoke other activity in parallel.

The next two listings below provide a snippet of the `InitDataCollection` and `setTheResultIntoDataCollection` activity.

**Listing 2. InitDataCollection**

```java
commonj.sdo.DataObject __result__1;
{// create DataCollection
    com.ibm.websphere.bo.BOFactory factory =
        (com.ibm.websphere.bo.BOFactory) new com.ibm.websphere.sca.ServiceManager().
            locateService("com/ibm/websphere/bo/BOFactory");
    __result__1 = factory.create("http://PatternsDemo","DataCollection");
}
dataCollection = __result__1;
```

**Listing 3. setTheResultIntoDataCollection**

```java
List list = dataCollection.getList("ProcessDataArray");
list.add(output1);
```
Test Data

field2=2; Expected result: the SCA component will be invoked twice, and the invoke response data are collected in an Array.

Multiple instances with a priori run-time knowledge

This pattern requires multiple instances which are created within a given process instance to be synchronized upon their completion. And the number of the instances can be affected by some run-time factors.

For example, the test plan should be approved by the required reviewers. The number of reviewers is different case by case.

Analysis

The focus of this pattern is:

1. Multiple instances are created.
2. The required number is affected by some runtime factor.
3. Synchronization is required at completion.

Implementation
The implementation is familiar with Multiple Instances with a Priori Design-Time Knowledge except that we have a human task named ManualSetIndex through which we can set the value to output1/field2 and in AssignIndex activity we transfer this value to a variable named indexnumber.

**Test Data**

1. Input any value to start the process,
2. Next go to My to-dos, and claim the ManualSetIndex task, and provide the field2=2,
3. Expected result: the SCA component will be invoked twice, and the invoke response data are collected in an Array.

**Multiple instances without a priori run-time knowledge**

This pattern requires multiple instances which are created within a given process instance to be synchronized upon their completion. And the number of the instances can be affected by some run time factors. More important, more instances can be added at run time.

For example, usually approving a team outing plan will only request half of the team members. However in some special case, more members will be involved to approve the case.
Analysis

The focus of this pattern is:

1. Multiple instances are created.
2. The required number is affected by some runtime factor.
3. More instances can be added at run time
4. Synchronization is required at completion.

Implementation

Figure 4. Multiple Instances without a Priori Run-Time Knowledge

We use subtask to implement this pattern. For detail on subtask, please refer to: http://publib.boulder.ibm.com/bpcsamp/v6r2/humanTaskFeatures/subtask.html. Notes: We should enable Enable subtask creation attribute of HumanTask2CreateSubtask activity.

Static partial join for multiple instances

This pattern requires multiple instances which are created within a given process instance to be synchronized upon their completion. The next task in the process is triggered when part of the instances complete.

For example, we can complete survey when we get feedback from 70 percent of the members. We will not care the result from the rest.

Analysis

The focus of this pattern is:

1. Multiple instances are created.
2. The required number is known when the first task commences.
3. The next task can be triggered when n instances of m completes.
4. The results of the rest instances are inconsequential.
**Implementation**

**Figure 5. Static Partial Join for Multiple Instances**

We use `EventHandler` to implement this pattern. In `Scope1` which contains `ForEach` activity, we control the numbers of the newly created task instances. In `HumanTask` activity, we initialize the `input1` variable. After the completion of `HumanTask`, an `Invoke` activity is triggered which will lead the process to the `OnEvent` scope.

In `OnEvent` Scope, we count the numbers of the tasks with variable `count`. If the count reaches `iteration`, the process goes to the next activity.

Note: Since there is more then one receiving activity be completed, the correlation set should be defined.

**Test Data**

For input data, we should input an array of type P34BO1 and an integer for iteration. It is recommended that the number of array is larger than the iteration. After claim and complete 2 human task activities, we can see in the log:

```
[10-10-3 10:38:42:921 CST] 0000004c SystemOut 0 Index: 1
[10-10-3 10:38:44:500 CST] 0000004c SystemOut 0 Index: 2
[10-10-3 10:38:44:718 CST] 0000004c SystemOut 0 Index: 3
[10-10-3 10:39:16:234 CST] 0000004c SystemOut 0 count: 1
[10-10-3 10:39:16:234 CST] 0000004c SystemOut 0 iteration > count
[10-10-3 10:39:23:218 CST] 0000004c SystemOut 0 into next activity
```
Cancelling partial join for multiple instances

This pattern requires multiple instances which are created within a given process instance to be synchronized upon their completion. The next task in the process is triggered when part of the instances complete. And in the same time, other instances are cancelled.

For example, suppose we are voting in a decision. If we have reached the threshold, there is no need for the rest to submit. So the rest submit task can be cancelled.

Analysis

The focus of this pattern is:

1. Multiple instances are created.
2. The required number is known when the first task commences.
3. The next task can be triggered when n instances of m completes.
4. The results of the rest instances are cancelled.

Implementation

Figure 6. Cancelling Partial Join for Multiple Instances

We use the new feature from both WPS v7.0 and WID v7.0 named parallel task to implement this pattern.

Test Data

When a process instance is initialized, 3 human tasks are created. When 2 of them complete, the process goes to the next activity. We can see the left task is cancelled.

Dynamic partial join for multiple instances

This pattern requires multiple instances which are created within a given process instance to be synchronized upon their completion. And the number of the instances can be affected by some run time factors. More important, more instances can be added at run time. The completion condition is specified at each time when an instance of the task completes.
For example, suppose we are voting in a decision in a group. The team lead can decide if it is necessary to involve members from other team or disable involving others.

**Analysis**

The focus of this pattern is:

1. Multiple instances are created.
2. The required number is affected by some runtime factor.
3. More instances can be added at run time.
4. Adding switch can be turned off.
5. The completion condition is specified at each time when an instance of the task completes.

**Implementation**

**Figure 6. Dynamic Partial Join for Multiple Instance**

We implement this pattern with `EventHandler` and `FaultHandler` feature of WPS. In the `FaultHandler` (see `Catch` activity), we simulate the completion of the task instances. If the user wants to move to the activity, we throw an exception named `UserChooseToFinishFault` and then the process will go to `Snippet7` to continue.

We have got 3 scopes here. In `ForEach` scope, we initialize the input parameter and other variables. We create task instances in `ForEach` activity. In `Scope1`, we wait for the command to
add or disable adding activity. \textit{WhetherToAddTask} is the activity to receive such command. The \textit{Invoke} activity will invoke the same human task as the one in \textit{Foreach} scope. The \textit{onEvent} scope performs the familiar function as in pattern \textit{Static Partial Join for Multiple Instances} in which we will not synchronize when the task completes.

Note: Since there is more than one receiving activity, the correlation set must be defined.

\textbf{Test Data}

For input parameter of the process, we provide a BO array with 3 element and the processId is set to 11111. We can see 4 human tasks are created. 3 Human tasks and 1 \textit{WhetherToAddTask}.

\begin{verbatim}
[10-10-3 11:00:15:484 CST] 0000004c SystemOut 0 Initialize
[10-10-3 11:00:15:500 CST] 0000004c SystemOut 0 AddNewTask
[10-10-3 11:00:17:906 CST] 0000006d SystemOut 0 Entry HumanTask
[10-10-3 11:00:19:687 CST] 0000006d SystemOut 0 Entry HumanTask
[10-10-3 11:00:25:656 CST] 0000004b SystemOut 0 into while loop
\end{verbatim}

We can attempt to complete a human task. Note that we should set processId to 11111 and field3 to false.

\begin{verbatim}
[10-10-3 11:05:14:546 CST] 0000003f SystemOut 0 Exit HumanTask
[10-10-3 11:05:15:750 CST] 0000006d SystemOut 0 User choose NOT to finish
\end{verbatim}

We are still in waiting status for adding new task. Now we can attempt to add a new task. We should set field1 of output1 to true.

\begin{verbatim}
[10-10-3 11:06:43:468 CST] 0000003e SystemOut 0 User choose to add more tasks
[10-10-3 11:06:43:468 CST] 0000003e SystemOut 0 processId:11111
[10-10-3 11:06:43:859 CST] 0000004c SystemOut 0 Event triggered
[10-10-3 11:06:46:781 CST] 0000004c SystemOut 0 into while loop
[10-10-3 11:06:47:265 CST] 0000006d SystemOut 0 Adding new HumanTask
[10-10-3 11:06:47:796 CST] 0000004c SystemOut 0 Entry HumanTask
\end{verbatim}

We can see that we have 3 human tasks again and \textit{WhetherToAddTask} is still there for the following up command. If we set field1 in \textit{WhetherToAddTask} to false, we can see \textit{WhetherToAddTask} is no longer existing.

\begin{verbatim}
[10-10-3 11:06:47:796 CST] 0000004c SystemOut 0 Entry HumanTask
[10-10-3 11:09:51:046 CST] 0000003c SystemOut 0 User choose NOT add any task
\end{verbatim}

Now we can move to next activity by working on a human task and set field3 to true. Now the process goes to Snippet7.

\begin{verbatim}
[10-10-3 11:11:49:171 CST] 0000003c SystemOut 0 Exit HumanTask
[10-10-3 11:11:49:828 CST] 0000004c SystemOut 0 User choose to finish
\end{verbatim}

\textbf{Arbitrary cycles}

This pattern is about the cycles in a process. Some activities can be entered and exited several time according to the cycle.
Implementation

Figure 8. Arbitrary Cycles

We implement this pattern with the Generalized Flow feature in WPS v7.0 and WID v7.0

Test data

Test Data1:

1. Input any value to start the process,
2. then go to My to-dos, and claim the ArbitraryCyclesTask1, input any value for field1
3. then go to My to-dos, and claim the ArbitraryCyclesTask2, input any value for field1
4. Expected result: the following activity should be invoked sequentially
   - Snippet1, Snippet2, Snippet3, Snippet4, Snippet5

Test Data2:
1. Input any value to start the process,
2. then go to My to-dos, and claim the ArbitraryCyclesTask1, input "redo1" for field1
3. then go to My to-dos, and claim the new ArbitraryCyclesTask1, input any value for field1
4. then go to My to-dos, and claim the ArbitraryCyclesTask2, input any value for field1
5. Expected result: the following activity should be invoked sequentially:
   - Snippet1, Snippet2, Snippet3, Snippet2, Snippet3, Snippet4, Snippet5

Test Data3:

1. Input any value to start the process,
2. then go to My to-dos, and claim the ArbitraryCyclesTask1, input any value for field1.
3. then go to My to-dos, and claim the ArbitraryCyclesTask2, input any "redo2" for field1.
4. then go to My to-dos, and claim the new ArbitraryCyclesTask1, input any value for field1.
5. then go to My to-dos, and claim the new ArbitraryCyclesTask2, input any value for field1.
6. Expected result: the following activity should be invoked sequentially:
   - Snippet1, Snippet2, Snippet3, Snippet2, Snippet3, Snippet4, Snippet3, Snippet4, Snippet5

Structured loop

This pattern is about the loops in a process. We can have the re-test or post-test condition in the process. That is, the for loop and the while loop

Implementation

We use ForEach feature to implement the Structured Loop as below:

Figure 9. Structured Loop

![Diagram of Structured Loop](image)

We use While Loop feature to implement the Structured Loop (Pre-Test) as below:
Figure 10. Structured Loop Pre Test

We use Repeat Until Loop feature to implement the Structured Loop (Post-Test) as below:

Figure 11. Structured Loop Post Test

Conclusion

In this article, we introduce 7 multiple instance patterns and 2 iteration patterns with related WID implementation. With those workflow control patterns, we can construct complex business process gracefully.
Downloadable resources

<table>
<thead>
<tr>
<th>Description</th>
<th>Name</th>
<th>Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>PI files for this article</td>
<td>Part2Pl.zip</td>
<td>178KB</td>
</tr>
</tbody>
</table>

© Copyright IBM Corporation 2010

Trademarks
(www.ibm.com/developerworks/ibm/trademarks/)