**Modeling sub-processes**

**子流程建模**

Within a BPD, we can create a sub-process. A sub-process is simply a grouping together of a set of activities into an aggregate area. For example, consider the following process diagram:

在一个流程定义，我们能够创建一个子流程。一个子流程是简单地一分组聚集的一系列在集合领域的活动。举个例子， 参考下面的流程图表：



Here we see us take an order, package the order (getting more stock if we are out) and finally shipping the order. What we need to pay attention to is the center part of our diagram. The details of ordering stock (if needed) is perhaps not necessary to understanding the overall flow of our process. Instead, we might want the diagram to look as follows:

这里我们看见我们订了个订单， 打包订单（如果我们是空的，获得更多的进货）并且最终运送订单。我们需要注意的是我们图表的中心部分。订购进货的详情（如果必须）可能是不必要去理解整个我们流程。相反，我们可能想要图表看起来是下面这样：



Notice the marker on the Build Order step. This indicates that this is a sub-process. If we drill down into this step, we find:

注意在Build Order这一步的标记。这个表明这是一个子流程。如果我们进入这里面，我们发现：



What we have done is nested the steps for the Build Order activity as a grouped/hidden sequence of steps. This sub-process shares the same variables as the parent. In addition, the names of the steps defined in the sub-process must be distinct from those of the parent. It is as though the

steps were placed "in-line" in the parent process and have simply been hidden for readability. An important aspect to note is that there is **no** re-usability of these sub-process steps. Their hiding in a sub-process is for readability only.

我们已经做了的是对Build Order活动做嵌入的步骤，作为一聚合/隐藏顺序的步骤。 这个字流程和父流程共享相同的变量。另外，定义在子流程的名字必须是与父流程那些不同。

它好像是被‘内嵌的’放置在父流程并且简单地对可读性隐藏。一个重要方面要注意的是没有那些子流程步骤的可以重用性。它们隐藏在一个子流程是仅仅是为了可读性。

Experimentation has shown that if a Terminate Event is met within a sub-process then the sub process is terminated and control is returned to the step following the sub-process definition.

实验已经显示如果一个中止事件相遇在一个子里面，然后子流程被中止并且控制被返回到紧接着的子流程定义的步骤

See also:

• Terminate Event

参考：

。中止事件

**Modeling Linked Processes**

**链式流程建模**

An alternative style of creating sub-processes is the notion of the Linked Process. In this story, a separate BPD is created that contains re-usable BPD activities. Input and Output variables are defined which describe the expected inputs and outputs from the linked process. In the calling process, a Linked Process activity is defined and a reference added to the target BPD that should be invoked when the parent reaches it.

一个创建子流程替代的类型是链式流程。在这个故事， 一个单独的流程定义被创建，它包括可重用的流程定义活动。输入和输出变量被定义，它们描述了从链式流程里期待的输入和输出。在一个正在请求的流程，一个链式流程活动被定义，

Again we can see the marker that indicates that it contains additional steps. Notice the heavy border around the activity which marks it as distinct from a sub-process.

A step in the process can dynamically choose which linked-process to invoke without explicitly

having to define the name of the BPD to be called. To achieve this, create a variable of type

String and populate its value with the name of the linked-process to invoke. In the Advanced

section of the Implementation area, select that variable as the source of the name of the process. At

run-time, the variable will be consulted and a dynamic call to the process with that name will be

made.

If parameters are to be passed to a dynamically called process then each process that may be

potentially selected to be called must have the same set of parameters. Think of this as the

dynamically invoked process having a template.

The BPD that is named as the process to be started **must** have a Start Event contained within it

and this will be the starting point for the new sub-process.

When a parent BPD invokes a child BPD any variables passed in as parameters are passed by

reference. What this means is that if a child process changes the values of these passed in variables

then the changes will also occur in the parent process. Care must be taken here to watch out for

unexpected side effects.

The BPD Process ID of the parent is the same Process ID used for the child.

**Modeling Event Sub-processes**

For certain types of events, we can create a "sub-process" which will be invoked if such an event is

detected but is not otherwise handled elsewhere. For example, if we have a set of steps that we

wish to be executed whenever an exception is thrown we don't want to have to wire this code to

every activity in our process that may throw an exception. Instead, we create an Exception Event

Sub-process that will be invoked when **any** uncaught exception is thrown.

For example, looking at the following BPD fragment, we see an activity called "Do Something" that

presumably does something. Under normal circumstances, "Do Something" will end and that will

indicate the end of the process as a whole. But, what if "Do Something" throws an exception? That

is where the event sub-process that we called "Exception Handler" comes into play. It contains a

set of activities that will be executed whenever an exception event is thrown.

Page 206

If we double click to expand the Exception Handler, we will see that it itself contains steps:

In this case, it is simply a Script fragment that logs to the console. So, if the parent activity called

"Do Something" throws an exception, control will be given to the steps contained within the

"Exception Handler" which will log data.

Note that although Process Designer allows us to create multiple Sub-process handlers for the same

type of event, it is an error to do so. It isn't clear which of the two will be executed.

Because the Sub-process event handler doesn't have any inputs or outputs, there is no follow-on

work from this step.

To gain access to the exception details in an Exception Sub-Process handler, a variable of type

XMLElement must be created and assigned from the tw.system.step.error variable in the

Post Assignment of the Start node in the diagram:

Page 207

The Event Sub-process provides handling for event types other than exceptions. Message and

Timer events can also be modeled in a similar fashion.

For Event Sub-processes which have starts defined for Message, we have the opportunity to supply

a correlation id value to ensure that the correct instance of the process is woken by a corresponding

event.

Take extra special care when defining an Event Sub-process which is triggered by the arrival of a

message. The parameters called "Interrupt Parent Process?" and "Repeatable"

come into play. If "Interrupt Parent Process" is checked, the the arrival of the message

causes the container of the Sub-process to be terminated. If the "Repeatable" option is not

checked, then messages after the first one will be ignored.

See also

• Message Start Event

• DeveloperWorks - Designing event-driven business processes in IBM Business Process Manager - 2012-06-27

**Script Activities**

An activity can have an implementation type of Script.

Page 208

Script allows the programmer to include JavaScript code in-line within the BPD process. The

JavaScript can utilize the IBPM supplied JavaScript classes. Although this option is exposed to be

used it is unlikely to be a good long term strategy. Code entered here is not re-usable by other

activities or services. A better idea would be to build a General Service which includes a Server

Script element and invoke the General Service.

**External Implementation**

When an activity in a BPD is reached, it is associated with and implemented by one of the

implementation Service types associated with the IBPM product. These include the common

General Service, Integration Service and Human Service types. IBPM provides another type of

implementation that is called an "External Implementation". This can be subtle to understand so

we will take it slowly. Also note that in previous releases, the External Implementation used to be

called an External Activity.

The overall goal of the concept of the External Implementation is that some application or code

**outside** of the IBPM environment is going to perform some work on behalf of the overall execution

of the process. This is not an uncommon situation and IBPM provides a variety of ways in which

external applications can be **called** to perform work and return their results. This includes Web

Services, REST and other Integration mechanisms. The External Implementation concept though is

something different. It is much closer to the concept of a Human Service than it is to an Integration

Service.

When an External Implementation is reached in the BPD, a new IBPM Task is created and the BPD

process suspends itself until the task completes. Unlike a Human Service task, there are no sets of

Coaches or other UI components provided by IBPM associated with this Task. However, the Task

still exists and can be queried by the REST API or Web API. These APIs can be used by an

arbitrary external application to:

• Query for the existence of External Implementation tasks

• Obtain the parameters passed as input to an External Implementation

• Set the parameters to be returned from an External Implementation

• Utilize custom properties set on the External Implementation implementation

• Complete the External Implementation associated task

If we think about this for a while, we see that a BPD activity calling an external application through

an Integration Service is an explicit invocation of that application while an External Implementation

is much more focused on the External Implementation associated application polling and working

with the data.

To use external implementations, this capability must first be enabled in IBPM PD.

An External Implementation can be created from the Implementation Category:

Page 209

A dialog is presented that allows us to enter the name of the activity to be created:

Once done, the properties of the External Implementation can be entered:

Page 210

To use an External Implementation in a BPD, create an activity and change its implementation type

to be User Task. Next, the identity of the external implementation previously defined can be used.

**Completing an External Implementation – REST API**

When we obtain the data for a TaskInstance object of a task, we can get a property from that

object called the ExternalActivityID. If that value is null, then the task does **not** represent

an external activity. However, if it is not null, then its value represents an External Activity

identifier. A REST request called

GET /rest/bpm/wle/v1/externalActivity/{externalActivityID}/model[?

parts={string}]

can be used to return the model of the external activity. This model describes the possible input and

output data types for this activity. Here is an example of what is returned:

{

"status":"200",

"data":

{

"name":"EA1",

"customProperties":{},

"inputs":

{

"in1":

{

"isList":false,

Page 211

"type":"BO1"

}

},

"outputs":

{

"out1":

{

"isList":false,

"type":"BO1"

}

},

"validations":

{

"BO1":

{

"properties":

{

"f1":

{

"isList":false,

"type":"String"

},

"f2":

{

"isList":false,

"type":"String"

},

"f3":

{

"isList":false,

"type":"String"

}

},

"type":"object"

}

},

"ID":"60.0b902a52-8df2-44ed-85e2-04bbd980b01d"

}

}

Let us look at the inputs and outputs structures to begin with. These correspond to a list of

variables where each variable has a "type" attribute. The type attribute matches an entry in the

"validations" section.

When it comes time to obtain the input data for the External Activity, that data can be found in the

Task Instance details object. To return data as output to an External Activity, use the REST Task

Finish method (see: Finishing a Task).

When working with an External Activity, it is a good idea to build a notepad document that looks as

follows:

EA1

Inputs:

order

customerName String

orderType String

amount Decimal

Outputs:

Page 212

approval

approved Boolean

comments String

For a task input:

{

data:

{

data:

{

variables:

{

order:

{

customerName:

orderType:

amount:

}

}

}

}

}

For task completion:

{

approval:

{

approved:

comments:

}