ARQUITECTURA DE SISTEMAS DE GESTIÓN WORKFLOW EN SERVICIOS DE INFORMACIÓN PARA SALUD PÚBLICA

ANEXO B
MANUAL DE REFERENCIA JPDL

Camilo Ernesto Hoyos Calvache

Director: Ing. Diego Mauricio López Gutiérrez

Universidad del Cauca
Facultad de Ingeniería Electrónica y Telecomunicaciones
Departamento de Telemática
Línea de Investigación en Ingeniería de Sistemas Telemáticos
Popayán, Diciembre de 2005

1. ARCHIVO DE PROCESOS

El presente anexo describe el esquema del archivo de definición de procesos que interpreta el motor workflow JBPM: processdefinition.xml, escrito en el Lenguaje de Definición de Procesos JBPM (JPDL, JBPM Process Definition Languaje)

Un archivo de procesos es una descripción formal de un proceso de negocio. Se empaqueta como un archivo jar-estructurado, normalmente con una extensión .par. jBpm reconoce que la descripción de un proceso requiere tres tipos de datos:

1. Una descripción declaratoria del proceso comercial: en jPdl, se expresa en el archivo processdefinition.xml. El contenido de este documento explica el esquema de este archivo.
2. Programación de la lógica: adicionar la lógica de programación al proceso como se describe en processdefinition.xml, las clases java pueden ser incluidas en el archivo de procesos. Todas las clases en un archivo del procesos debe localizarse en el subdirectorio / clases.
3. Otros archivos de recursos: como un cliente del motor Workflow, se puede incluir una variedad de archivos de recursos del proceso para utilizar durante la ejecución. Por ejemplo las descripciones de formularios que están asociadas con las tareas a ser realizadas por una persona. jBpm no impone ninguna restricción en el tipo de archivos de recursos que usted desea incluir en una definición de procesos.

Mecanismo de versiones

Básicamente, el mecanismo de versiones de jBpm se resume en los siguientes principios:

- Cada vez que un archivo de procesos se despliega, en la base de datos de jBpm se crea una nueva definición de procesos.
- Al desplegarse, jBpm asigna un número de versión a la definición de procesos. Cuando el nombre del proceso es el mismo, los archivos se consideran del mismo proceso. jBpm asigna el número de versión de la siguiente manera: toma 1 + el número de la versión más alto de las definiciones del proceso actuales con el mismo nombre. Si es la primera versión, entonces asigna valor 1. La API de jBpm ofrece métodos para obtener la última definición de procesos con un nombre determinado.
- Una instancia de proceso (también conocida como ejecución del proceso) empieza en una definición dada, el proceso instanciado seguirá en ejecución dentro de la misma definición hasta quedar terminado.
- De esta manera cada proceso puede empezar en la última definición y mantenerse funcionando en la misma definición durante su tiempo de vida.
- Note que incluso en jBpm es posible para la versión que la lógica de la programación se asocie con un proceso. Por la inclusión de clases en el archivo de procesos, jBpm separará las clases para definición de procesos.

2. ESQUEMA DE 'processdefinition.xml'

2.1 Definición de tipo de documento

Definición del tipo de documento processdefinition.xml

```
<!DOCTYPE process-definition PUBLIC
  "-//jBpm/jBpm Mapping DTD 2.0 beta3//EN"
  "http://jbpm.org/dtd/processdefinition-2.0-beta3.dtd">
```

2.2 Definición de procesos

fragmentos DTD para definición de procesos

```
<!ELEMENT process-definition (  description?,
                        swimlane*,
                        type*,
                        start-state,
                        ( state |
                          milestone |
                          process-state |
                          decision |
                          fork |
                          join
                        )*,
                        end-state,
                        action*  ) >
<!ATTLIST process-definition name CDATA #REQUIRED >
```

2.3 Estado

fragmentos dtd para un estado

```
<!ELEMENT state ( description?, assignment?, action*, transition+ ) >
<!ATTLIST state name  CDATA #REQUIRED >
```

En jBpm el término estado tiene el mismo significado que en las máquinas de estados finitos (FSM, Finite State Machine) o los diagramas de estado UML.

El propósito del modelado de un proceso comercial en jBpm es crear un sistema software. Consideramos la definición del proceso como una parte del sistema software que es la estructura. Así que el término estado en jBpm será interpretado desde el punto de vista del sistema software del cual jBpm forma parte.

El estado es el concepto central de jBpm. Al empezar a modelar un proceso en jBpm, lo

primero que se debe hacer es pensar en los estados del proceso, ya que forman la estructura del proceso.

Hay una razón muy específica por la cual jBpm se centra alrededor del concepto de estado. Es porque el estado no tiene un similar en lenguajes de programación. Un programa software o está corriendo o no. Típicamente un proceso comercial relaciona partes de lógica de programación que se ejecutan separadamente. jBpm permite modelar estos estados entre las partes relacionadas de la lógica de programación.

jPdl también define el flujo de control entre los estados con transiciones, decisiones, bifurcaciones, uniones y referencias. Note el control de flujo definido entre los estados. Los diseñadores del proceso pueden especificar una acción en un evento del proceso donde la lógica de programación sea más apropiada. La filosofía de jPdl es extender java con la administración de estados y tener la menor superposición con la programación. Esto diferencia a jPdl de otros lenguajes de procesos de negocio como BPEL, los cuales no hacen una clara separación.

2.4 Asignación

fragmentos dtd para una asignación

```
<ELEMENT assignment EMPTY >
<!ATTLIST assignment swimlane CDATA #IMPLIED
                assignment (optional|required) #IMPLIED
                authentication (optional|required|verify) #IMPLIED >
```

Cuando una ejecución de proceso llega a un estado, el motor Workflow esperará hasta que se dé un disparo externo (con el método de API de jBpm ExecutionService.endOfState (...)). Dentro de ese método, los jBpm calcularán el próximo estado de la instancia del proceso.

Así que un estado puede verse como una dependencia de un actor externo. El actor externo puede ser una persona, un grupo de personas o un sistema. Hay numerosas maneras de como los estados en un proceso comercial pueden relacionarse a tareas que básicamente pueden ser divididas en dos grupos:

2.4.1 Asignación basada en cliente

En esta estrategia, los usuarios de jBpm manejarán la lista de tareas para sus mismos usuarios. En este caso jBpm sólo se utiliza como un motor de ejecución para las máquinas de estados finitos. La responsabilidad de jBpm es calcular y guardar registro del estado de ejecución del proceso, mientras la responsabilidad del cliente es asignar la tarea de cada usuario. Típicamente los clientes quieren encontrar todas las instancias del proceso (o fichas) en un estado dado.

2.4.2 Asignación basada en proceso

En esta estrategia de asignación, jBpm es el responsable de asignar los estados a actores

y mantener registro de listas de tareas. En jBpm, el progreso durante la ejecución de un proceso se rastrea con una bandera (token). Un token tiene un apuntador hacia un estado y hacia un actor. En jBpm un actor es referenciado con una simple cadena de caracteres (es decir, con un objeto de tipo java.lang.String). Hay varios eventos relevantes en la asignación de token a los actores, a continuación se describe cada uno.

Siempre que un cliente inicie una nueva instancia del proceso o indique el fin de un estado, jBpm empieza a calcular el próximo estado para esa instancia del proceso.

Lo primero que se quiere mencionar acerca de las asignaciones es que el primer parámetro de las llamadas al método de la API es el actor. Ese parámetro se utiliza para especificar el responsable del método esta ejecutando.

En el caso de empezar una nueva instancia de proceso, en el estado inicial (start-state) se crea un token raíz. En el caso de invocar el fin de un estado (llamando al método endOfState), se necesita especificar un identificador de token como parámetro que está en algún estado. Entonces, JBPM empezará calculando el nuevo estado para el token. De momento, ignoraremos la concurrencia por simplicidad. El token viajará sobre transiciones y nodos hasta llegar a un estado (considerado también como un nodo). En ese momento, el token necesita ser asignado a un actor. Después de la selección del actor, jBpm asociará al actor seleccionado con el token y el método invocado (startProcessInstance o endOfState) retornará.

De este modo, cuando un token en un estado tiene una referencia a un actor, significa que la ejecución de esa instancia de proceso está esperando que un actor proporcione un disparador externo al motor jBpm. En este caso, un estado en el proceso corresponde a una tarea para un usuario. jBpm calcula la lista de tareas buscando todos los token asignados al actor dado para que selecciones un token de la lista y nuevamente señala el fin del estado.

La asignación tiene más atributos que aún no se cubre: asignación y autenticación. El atributo asignación puede tener 2 valores: opcional y requerido. Requerido significa que después que la ejecución ha llegado a un estado, jBpm verificará si el token en ese momento está asignado a un actor. Si esto no se cumple, se lanza la respectiva excepción (AssignmentException). Si la asignación es opcional (= por defecto), jBpm permite dejar un token sin asignar cuando llega a un estado. El atributo autenticación especifica restricciones que permiten al actor señalar el fin de un estado. El actor es especificado con un parámetro de identificación, llamado ‘actorId’ en el método que indica el fin de un estado (endOfState). Opcional significa que no se requiere especificar el responsable de finalizar el estado. Requerido significa que se necesita especificar un actor y verificar que sea el quien señale el fin del estado.

2.5 Rol de los participantes

Por cuestiones técnicas de terminología del lenguaje de definición de procesos de jBpm, de ahora en adelante el rol de los participantes se denominará swimlane.

fragmentos dtd para swimlane

```
<!ELEMENT swimlane ( description?, delegation? ) >
<!ATTLIST swimlane name CDATA  #REQUIRED >
```

Típicamente, una persona es responsable de múltiples estados en un proceso. En jBpm esto se expresa creando un swimlane y asignando todos los estados de ese actor al swimlane. Un swimlane en un proceso de negocio puede verse como un nombre de rol para un actor dentro del proceso. La interpretación jBpm de swimlane corresponde al término swimlane utilizado en UML 1.5. El actor se asigna la primera vez que la ejecución llega a un estado para un swimlane dado.

Interfaz AssignmentHandler

```
public interface AssignmentHandler {
  String selectActor( AssignmentContext assignerContext );
}
```

La etiqueta de delegación dentro de un swimlane referencia una implementación de la interfaz AssignmentHandler.

El actor se almacena en una variable de proceso con el mismo nombre del swimlane. La próxima vez que el proceso llega a un estado para el swimlane, el motor jBpm identificará la variable y asignará el token al actor que allí se almacenó.

De este modo los swimlanes se definen en el nivel de proceso y los estados los referencian en el momento de la asignación.

El resultado de este cálculo se almacenará en una variable de proceso con el mismo nombre del swimlane. Entonces, cuando se llega al próximo estado del mismo swimlane, ese estado se asigna al mismo actor sin necesidad de utilizar otra vez el método AssignmentHandler. Ya que la relación entre un swimlane y el actor se guarda en una variable, es posible manipular esa relación actualizando la variable.

2.6 Variable y tipo

fragmentos dtd para tipo y variable

```
!ELEMENT type ( description?, (delegation | transient), variable* ) >
<!ATTLIST type java-type CDATA #IMPLIED >
<!ELEMENT transient EMPTY >

<!ELEMENT variable EMPTY >
<!ATTLIST variable name CDATA #REQUIRED >
```

2.6.1 Variables

Una variable es un par clave-valor asociado con una instancia de un proceso (= una ejecución de un proceso). La clave es un objeto java.lang.String y el valor puede ser de

cualquier tipo de objetos de java. Incluso los tipos de objetos de java que son desconocidos para jBpm se pueden utilizar en las variables de un proceso. Las variables pueden establecerse de las siguientes maneras:

- ExecutionService.startProcessInstance(String idActor, Long idDefinicion, **Map variables,** String nombreTransicion)
- ExecutionService.endOfState(String idActor, Long idToken, **Map variables,** String nombreTransicion)
- ExecutionService.setVariables(String idActor, Long idToken, **Map variables**)
- ExecutionContext.setVariable(String nombre, Object valor)
- ExecutionService.getVariables(String idActor, Long idToken)
- ExecutionContext.getVariable(String nombre)

Cuando se trabaja con variables se intenta imitar tanto como sea posible la semántica de java.util.Map a través de la API de jBpm. Esto significa que una variable es instanciada solamente cuando se inserta y que cualquier tipo de objeto java se puede utilizar como valor.

2.6.2 Tipo

Un tipo especifica la forma en que jBpm debería almacenar el valor de una variable en la base de datos. jBpm tiene un campo de texto para almacenamiento de valores para que la conversión entre el texto y el objeto se realice mediante serialización:

La interfaz serializable

```
public interface Serializer {
  String serialize( Object object );
  Object deserialize( String text );
}
```

Un tipo puede verse como una referencia a un serializador. jBpm incluye las aplicaciones de Serialización predefinidas para los siguientes tipos de java:

```
java.lang.String
java.lang.Long
java.lang.Double
```

2.6.3 Comparación Variable-Tipo

Las variables que son de un tipo que Java soporta por defecto no tienen que ser declaradas en el archivo processdefinition.xml. El motor jBpm tiene un mecanismo semi-automático para asignación de tipo: cuando se instancia una variable, jBpm intenta calcular su tipo examinando su valor. Si el valor de la variable es un tipo que Java soporta por defecto, se utiliza ese tipo. De lo contrario, jBpm verifica si corresponde a un tipo declarado en el archivo processdefinition.xml. En esta comparación jBpm también considera los súper tipos de los valores de las variables. Cuando el tipo no se encuentra, jBpm trata la variable como transitoria. Para evitar que jBpm tenga que pasar por este

proceso de comparación, las variables se pueden especificar en cada tipo.

2.6.4 Variables transitorias

Algunas variables no necesitan ser persistentes en la base de datos. Una variable para correo electrónico (to.email.address) se provee en una llamada del método endOfState() a la API jBpm. El estado tiene una transición saliente con una acción que envía un correo a una dirección especificada. Si esa fue la única utilización de la variable 'to.email.address' no debería ser persistente. Para este propósito jBpm soporta variables transitorias que no se guardan en la base de datos y solamente se puede utilizar dentro de una llamada al método de la API jBpm que las proporciona. En otras palabras, el alcance de las variables transitorias es una llamada a un método de la API jBpm.

2.7 Estado inicial

fragmentos dtd para el estado inicial

```
<!ELEMENT start-state ( description?, transition+ ) >
<!ATTLIST start-state name  CDATA #REQUIRED
                swimlane CDATA #IMPLIED >
```

En un proceso el estado inicial es el único desde el cual inician todas las instancias del proceso. Note que al tiempo que inicia la instancia de un proceso, sus variables ya se pueden alimentar. Otro concepto importante es que se pueden tener múltiples transiciones salientes en el estado inicial. En este caso es necesario especificar cual transición se debería tomar cuando se inicia una instancia de un proceso.

2.8 Anuncio

Por cuestiones técnicas de terminología del lenguaje de definición de procesos de jBpm, de ahora en adelante el anuncio se denominará milestone.

Fragmento dtd para un milestone

```
<!ELEMENT milestone ( description?, action*, transition ) >
<!ATTLIST milestone name  CDATA #REQUIRED>
```

Un milestone es un tipo especial de estado que se puede utilizar para sincronización entre dos rutas de ejecución concurrentes. Se puede utilizar en una situación en que una ruta de ejecución necesita esperar hasta que ocurra un evento en otra ruta de ejecución. Si el milestone no fue alcanzado, la ejecución espera en el estado milestone hasta que la otra ruta de ejecución concurrente alcance el milestone. Si el milestone ya fue alcanzado, la ejecución pasa solo a través de ese estado.

Un estado milestone se relaciona con una o más acciones que señalan el alcance de un milestone. Esas acciones se pueden modelar en el proceso con el ActionHandler por defecto (`org.jbpm.delegation.action.MilestoneReachedActionHandler`). La acción

que indica al motor jBpm que se alcanzó un milestone se podría programar de la siguiente manera en el archivo processdefinition.xml:

```
...
  <milestone name="theMilestone">
    <transition to="stateAfterMilestone" />
  </milestone>
  ...
  <state name="stateBeforeReachingMilestone" swimlane="initiator">
    <transition to="stateAfterReachingMilestone">
      <action>
          <delegation
class="org.jbpm.delegation.action.MilestoneReachedActionHandler">theMilestone</delegation>
      </action>
    </transition>
  </state>
  ...
```

2.9 Estado de proceso

fragmento dtd para un estado de proceso

```
<!ELEMENT process-state ( description?, delegation, action*, transition+ ) >
<!ATTLIST process-state name CDATA #REQUIRED>
```

Un estado de proceso corresponde a la invocación de un súper-proceso. Los procesos padre inician un subproceso cuando la ejecución llega al estado de proceso. El proceso permanece en el estado de proceso mientras dura el sub-proceso. Cuando se termina el sub-proceso, se abandona el estado de proceso.

2.10 Decisión

fragmento dtd para una decisión

```
<!ELEMENT decision ( description?, delegation, action*, transition+ ) >
<!ATTLIST decision name CDATA #REQUIRED>
```

Una decisión decide entre múltiples caminos de ejecución excluyentes. Si usted es un programador, véalo como una estructura si-entonces (if-else), pero por supuesto, una decisión puede tener tantas transiciones salientes como se desee.

Note que una decisión modela una situación dónde el motor Workflow decide qué ruta elegir basada en el contexto (= variables) y quizás algunos recursos externos. Como una alternativa, se puede modelar múltiples transiciones salientes en un estado. En ese caso, el cliente jBpm debe decidir cual de las transiciones salientes debe tomar incluyendo el nombre de la transición seleccionada como parámetro en la invocación del método de finalización de estado (endOfState).

2.11 Tridentes

fragmento dtd para un tridente

```
<!ELEMENT fork ( description?, delegation?, action*, transition+ ) >
<!ATTLIST fork name              CDATA #REQUIRED
          corresponding-join CDATA #IMPLIED>
```

Un tridente genera múltiples rutas concurrentes de ejecución. La interfaz ForkHandler permite especificar un comportamiento personalizado para el tridente. Pero el comportamiento predefinido (cuando no se especifica alguna delegación) es que se produce un token hijo por cada transición saliente del tridente. De este modo, solo será necesario implementar la interfaz ForkHandler para la concurrencia avanzada.

Normalmente, un tridente tiene relacionada una agrupación que define un bloque de concurrencia. Con el comportamiento predefinido del tridente y de la agrupación, solo se soportan características estrictas, no se soportan transiciones fuera del bloque de concurrencia.

interfaz ForkHandler

```
public interface ForkHandler {
  void fork( ForkContext forkContext ) throws ExecutionException;
}
```

2.12 Agrupaciones

fragmento dtd para una agrupación

```
<!ELEMENT join ( description?, delegation?, action*, transition ) >
<!ATTLIST join name              CDATA #REQUIRED
          corresponding-fork CDATA #IMPLIED>
```

Una agrupación une múltiples rutas concurrentes de ejecución. La interfaz JoinHandler permite personalizar su comportamiento pero el comportamiento predefinido (cuando no se especifica alguna delegación) es que une todos los token que se han producido en el tridente correspondiente. El último token en llegar a la agrupación dispara al token padre para proceder con la transición saliente de la agrupación. De este modo, solo se necesita implementar la interfaz JoinHandler para concurrencia avanzada.

Restricción: una agrupación solo puede tener una transición saliente.

interfaz JoinHandler

```
public interface JoinHandler {
  void join( JoinContext joinContext ) throws ExecutionException;
```

```
}
```

2.13 Estado final (end-state)

fragmento dtd para estado final

```
<!ELEMENT end-state EMPTY >
<!ATTLIST end-state name CDATA #REQUIRED>
```

Una definición de proceso tiene exactamente un estado final. Cuando la ejecución de una instancia de proceso llega al estado final, se termina la instancia del proceso.

2.14 Transición

fragmento dtd para una transición

```
<!ELEMENT transition ( action* )>
<!ATTLIST transition name CDATA #IMPLIED
                to   CDATA #REQUIRED>
```

Las transiciones especifican conexiones dirigidas entre nodos. El elemento de la transición debería colocarse dentro del nodo desde el cual sale la transición.

2.15 Acción

fragmento dtd para una acción

```
<!ELEMENT action ( delegation ) >
<!ATTLIST action event-type (process-start|process-end|
                    state-enter|state-leave|state-after-assignment|
                    milestone-enter|milestone-leave|
                    decision-enter|decision-leave|
                    fork-enter|fork-every-leave|
                    join-every-enter|join-leave|
                    transition) #IMPLIED>
```

Una acción es un código Java que puede ser ejecutado por el motor workflow en un evento durante la ejecución del proceso.

La `acción` es siempre definida como el hijo de un elemento-de-definición-de-proceso (ejemplos de elemento: `definición del proceso`, `estado`, `transición`, `decisión...`). El elemento del padre más el tipo de evento define el momento exacto en se ejecuta la acción durante la ejecución del proceso. Los posibles tipos de evento de una acción dependen del elemento que contiene la acción. Los nombres de tipo de evento sugieren ya en qué elemento son aplicables.

interfaz ActionHandler

```
public interface ActionHandler {
  void execute( ExecutionContext executionContext );
}
```

2.16 Delegación

Fragmento dtd para una delegación

```
<!ELEMENT delegation ( #PCDATA ) >
<!ATTLIST delegation class CDATA #REQUIRED>
```

Explican la relación entre el elemento contenedor y la interfaz que la clase de la delegación tiene que implementar.