JBoss jBPM 3.1

Guía Práctica de Workflow y BPM

Tabla de Contenidos

Traducción al Español en el contexto del proyecto Met@logo y auspiciada con fondos de la Unión Europea
www.metalogo.org

19. Architectura Conectable

Capítulo 1. Introducción

JBoss jBPM es un sistema flexible y extensible de administración de flujo de trabajo. JBoss jBPM cuenta con un lenguaje de proceso intuitivo para expresar gráficamente procesos de negocio en términos de tareas, estados de espera para comunicación asíncrona, temporizadores, acciones automátizadas,... Para unir estas operaciones JBoss jBPM cuenta con el mecanismo más poderoso y extensible de control de flujo.

JBoss jBPM tiene mínimas dependencias y se puede utilizar con la misma simpleza que una biblioteca java. Pero también puede utilizarse en ambientes donde es esencial contar con un alto nivel de producción mediante la implementación en un servidor de aplicaciones J2EE en cluster.

JBoss jBPM se puede configurar con cualquier base de datos y se puede implementar en cualquier servidor de aplicación.

1.1. Descripción General

El flujo de trabajo central y la funcionalidad BPM tienen un simple formato de biblioteca java. Esta biblioteca incluye un servicio para almacenar, actualizar y recuperar información de proceso de la base de datos jBPM.

Figura 1.1. Descripción general de los componentes de JBoss jBPM

1.2. Kit de inicio de JBoss jBPM

El kit de inicio es una descarga que contiene todos los componentes de jBPM agrupados en una descarga simple. La descarga incluye:

- **jbpm-server**, un servidor de aplicación jboss preconfigurado.
- **jbpm-designer**, el plugin eclipse para crear procesos jBPM en forma gráfica.
- **jbpm-db**, el paquete de compatibilidad de la base de datos jBPM (consultar a continuación).
- **jbpm**, el componente central de jbpm incluidas las bibliotecas y la presente guía de usuario.
- **jbpm-bpel**, una referencia a la extensión BPEL de JBoss jBPM.

El servidor de aplicación jBoss pre-configurado tiene instalados los siguientes componentes:

- **El componente central jBPM**, organizado como archivo de servicio
- **Una base de datos integrada con las tablas jBPM**: la base de datos hipersónica predeterminada que contiene las tablas jBPM y ya contiene un proceso.
- **La aplicación web de la consola jBPM** puede ser utilizado por participantes del proceso así como por administradores jBPM.
- **El programador jBPM** para la ejecución de los temporizadores. El programador está configurado en el kit de inicio en forma de servlet. El servlet va a dar origen una secuencia para monitorear y ejecutar los temporizadores.
- **El ejecutor de comandos del jBPM** para la ejecución asíncrona de comandos. El ejecutor de comandos también se configura como servlet. El servlet da origen a

una secuencia para monitorear y ejecutar los comandos.
- **Un proceso de muestra** ya está implementado en la base de datos jBPM.

1.3. Diseñador gráfico de proceso JBoss jBPM

JBoss jBPM también incluye una herramienta gráfica de diseño. El diseñador es una herramienta gráfica para crear los procesos de negocio.

El diseñador gráfico de proceso JBoss jBPM es un plugin eclipse. Existe una instalación independiente de la herramienta de diseño en el roadmap.

La característca más importante de la herramienta gráfica de diseño es que incluye soporte tanto para las tareas del analista de negocios como para el desarrollador técnico. Esto permite una transición armónica desde la modelación de procesos de negocio a la implementación práctica.

El plugin está disponible como sitio de actualización local (archivo zip) para la instalación a través del mecanismo estándar eclipse de actualización de software. Y también existe un paquete que se puede descomprimir en su directorio local eclipse.

1.4. Componente central JBoss jBPM

El componente central JBoss jBPM está en simple software java (j2SE) para administrar definiciones de proceso y el ambiente de ejecución para la ejecución de instancias de proceso.

JBoss jBPM es una biblioteca java. Como consecuencia, se puede utilizar en cualquier ambiente java como por ejemplo una aplicación web, una aplicación swing, un EJB, un servicio web,...La biblioteca jBPM también se puede organizar y exponer en una sesión EJB sin estado. Esto permite la implementación y escalabilidad en cluster y para lograr productividades extremadamente altas. El EJB de sesión sin estado se escribe según las especificaciones J2EE 1.3 de modo que sea desplegable en cualquier servidor de aplicación.

El componente central JBoss jBPM está organizado como un simple archivo de bibliotecas java. Dependiendo de las funcionalidades que se utilicen, la biblioteca `jbpm-3.0.jar` tiene algunas dependencias respecto de otras bibliotecas de terceros tales como: hibernación, dom4j y otras. Dichas dependencias están cláramente documentadas cápitulo 5, Implementación

Para su persistencia jBPM utiliza hibernación en forma interna. A parte de la tradicional asignción O/R, hibernate también resuelve las diferencias de dialecto SQL entre las diferentes bases de datos, haciendo que jBPM pueda operar con todas las bases de datos actuales.

Es posible acceder al API de JBoss jBPM desde cualquier software java en su proyecto, por ejemplo, su aplicación web, sus EJB's, sus componentes de servicio web, sus beans accionados por mensajes o cualquier componente java.

1.5. Aplicación web de la consola de JBoss jBPM

La aplicación web de la consola jBPM tiene dos propósitos. Primero, sirve como interfaz central de usuario para interactuar con tareas de ejecución generadas por las

ejecuciones de proceso. Y en segundo lugar, es una consola de administración y monitoreo que permite inspeccionar y manipular instancias de ejecución.

1.6. Componente de identidad de JBoss jBPM

JBoss jBPM puede integrarse con cualquier directorio corporativo que contenga usuarios y otra información organizacional. Pero para proyectos no existe disponibilidad de componente de información organizacional JBoss jBPM que incluya dicho componente. El modelo utilizado en el componente de identidad tiene mayores beneficios que los tradicionales modelos servlet-, ejb- y portlet.

Para obtener más información, consulte la sección 11.11, "El componente de identidad"

1.7. Programador de JBoss jBPM

El programador de JBoss jBPM es un componente destinado a monitorear y ejecutar los temporizadores que estén programados durante ejecuciones de preceso.

El software del componente temporizador viene incluido en la biblioteca central jbpm, pero necesita implemntarse en uno de los siguientes entornos: ya sea si debe configurar el servlet del programador para dar origen a la secuencia de monitoreo o si tiene que iniciar un JVM por separado con el programador.

1.8. Paquete de compatibilidad de bases de datos de JBoss jBPM

El paquete de compatibilidad de bases de datos JBoss jBPM es un paquete de descarga que contiene toda la información los controladores y secuencias de comandos para que jBPM funcione con su base de datos de preferencia.

1.9. Extensión BPEL de JBoss jBPM

La extensión BPEL de JBoss jBPM es una extensión que viene por separado y que permite que jBPM soporte BPEL. La esencia de BPEL es un lenguaje de scripting xml para escribir servicios web en términos de otros servicios web.

Capítulo 2. Cómo partir

El presente capítulo permite revisar los primeros pasos para obtener JBoss jBPM y entrega las indicaciones iniciales para comerzar a operar de inmediato.

2.1. Revisión General de las Descargas

A continuación se encuentra una lista de los diferentes paquetes jBPM que están disponibles actualmente. Cada uno de estos paquetes contiene uno o más archivos descargables. En conjunto con cada uno de estos archivos, se encuentra una descripción de sus contenidos y un puntero sobre instrucciones de instalación pertinente, si es que están disponibles.

Todos los archivos descargables a continuación se pueden encontrar en la página sourceforge de descargas de jbpm.

2.1.1. jBPM 3

Descargue JBoss jBPM 3 desde sourceforge.net. Este es el paquete principal de distribución que contiene el core engine y una serie de módulos adicionales que pueden ser necesarios para trabajar con jBPM.

- **Kit de Inicio (jbpm-starters-kit-<version>.zip):** Si desea comenzar a usar jBPM rápidamente, este es el archivo que debe descargar. Contiene todos los demás módulos de este paquete, además del diseñador gráfico en una sola descarga. Extraiga el archivo comprimido en zip a una carpeta de su elección y lea el archivo denominado 'readme.html' para obtener más información e instrucciones adicionales de instalación. Con este kit de Inicio puede comenzar a trabajar inmediatamente con el capítulo 3, Tutorial.
- **Motor principal y Componente de Identidad (jbpm-<version>.zip):** La descarga contiene el core engine de jBPM, así como el componente de identidad para administración de actor y de grupo. Para comenzar a trabajar con él, extraiga el archivo en una carpeta de su elección. Encontrará punteros hacia la Guía de Usuario y otros importantes recursos de información en el archivo 'readme.html' en la carpeta 'jbpm-<version>' .
- **Extensiones de Base de Datos (jbpm-db-<version>.zip):** El paquete de extensión de base de datos contiene el motor principal de jBPM, así como el componente de identidad para administración de actor y de grupo. Para comenzar a trabajar con él, extraiga el archivo en una carpeta de su preferencia. Encontrará punteros hacia la Guía de Usuario y otros recursos con información importante en el archivo 'readme.html' en la carpeta 'jbpm-<version>'.

2.1.2. Diseñador de Procesos jBPM

Descargue el Diseñador de Procesos JBoss jBPM en sourceforge.net. El diseñador es un plugin eclipse y permite crear sus definiciones de proceso e implementarlas con facilidad. El plug-in está disponible para descargarlo ya sea como un elemento Eclipse comprimido en zip o como sitio de actualización de Eclipse comprimido. No existe diferencia en el contenido, la única diferencia es en la forma en que se debe hacer la

instalación.

- **Sitio de Actualización Eclipse (jbpm-gpd-site-<version>.zip):** Si desea asegurarse completamente de que la instalación del diseñador se desarrolle si contratiempos, le recomendamos que utilice el mecanismo del sitio de actualización junto con una nueva instalación de Eclipse. Por supuesto la versión de Eclipse debe corresponder al archivo de actualización descargado. Para iniciar el trabajo con el plugin del diseñador siga las instrucciones en el archivo 'readme.html' incluido en la carpeta de raíz para instalar el GPD en forma correcta.
- **Eclipse (jbpm-gpd-feature-<version>.zip):** si está cansado de tener que hacer cada vez una nueva instalación de Eclipse y está dispuesto a solucionar algunas posibles problemáticas, puede probar la descarga feature. En este caso la instalación es tan fácil como extraer el archivo en su instalación de Eclipse (asegúrese de que las carpetas de 'plugins' y 'features' queden en la misma ubicación de su instalación de Eclipse) reemplazando los archivos y carpetas con el mismo nombre que probablemente exista. Esta instalación es muy sencilla, pero se pueden presentar inconvenientes de incompatibilidad cuando se reemplazan los plugins ya existentes en su instalación debido a las otras características que instaló anteriormente. Aunque tienen el mismo nombre podría suceder que las versiones de estos plugins en conflicto no sean las mismas; de ello se desprenden las posibles incompatibilidades. Las instrucciones de instalación se repiten en el archivo 'readme.html'.

2.1.3. Extensión BPEL de jBPM

Descarge la extensión BPEL de JBoss desde sourceforge.net. Contiene sólo un archivo : `jbpm-bpel-<version>.zip`. Para comenzar a trabajar con las extensiones BPEL, busque en la Guía de Usuario en la subcarpeta 'doc' de la carpeta de nivel superior.

2.2. El directorio de proyectos de JBoss jBPM

- Soporte profesional: JBoss es la compañía que respalda este proyecto con soporte profesional, capacitación y servicios de consultoría.
- **Guía de usuario**: es el documento que está leyendo y sirve como el punto principal de ingreso a este proyecto.
- foros: establezca contacto con la comunidad, haga preguntas y discuta sobre jBPM
- wiki: información adicional principalmente proporcionada cpo la comunidad
- rastreador de problemas: para enviar errores y solicitudes específicas
- descargas: página sourceforge de descargas para jBPM
- listas de correo: las listas de correos se utilizan para enviar anuncios
- **javadocs**: parte de la descarga en el directorio doc/javadoc.

2.3. Acceso a CVS

2.3.1. Acceso anónimo a CVS

Como alternativa se puede obtener JBoss jBPM desde cvs con la siguiente información:

- Tipo de conexión: pserver

- Usuario: anonymous
- Host: anoncvs.forge.jboss.com
- Puerto: 2401 (which is the default)
- Repository path: /cvsroot/jbpm
- Etiqueta: :pserver:anonymous@anoncvs.forge.jboss.com:/cvsroot/jbpm

2.3.2. Acceso de desarrollador a CVS

Para tener acceso en calidad de desarrollador a cvs se debe firmar un acuerdo de colaboración y necesita una clave ssh. Es posible encontrar más información sobre ambos en the JBoss cvs repository wiki page

- Tipo de conexión: ext over ssh (extssh in eclipse)
- Usuario: sf.net username or jboss username
- Host: cvs.forge.jboss.com
- Puerto: 2401 (which is the default)
- Ruta de repositorio: /cvsroot/jbpm
- Etiqueta: :pserver:anonymous@cvs.forge.jboss.com:/cvsroot/jbpm

Capítulo 3. Tutorial

El presente tutorial muestra los aspectos básicos de proceso en jpdl y el uso de API para administrar las ejecuciones de tiempo de ejecución.

El tutorial tiene el formato de una serie de ejemplos. Los ejemplos se concentran en un tema en particular y contiene amplios comentarios. También es posible encontrar los comentarios en el paquete de descarga de jBPM en el directorio `src/java.examples`.

La mejor forma de aprender es crear un proyecto y experimentar creando variaciones sobre los ejemplos dados.

Cómo partir para los usuarios de eclipse: descargue jbpm-3.0-[version].zip y descomprímalo en su sistema. Luego haga "File" --> "Import..." --> "Existing Project into Workspace". Haga clic en "Next" Luego, busque el directorio raíz jBPM y haga clic en "Finish". Ahora ya tiene un proyecto jbpm.3 en su área de trabajo. Ahora puede encontrar los ejemplos del turorial en `src/java.examples/....` Cuando abra dichos ejemplos, se pueden ejecutar con "Run" --> "Run As..." --> "JUnit Test"

jBPM incluye una herramienta gráfica de diseño para crear el XML que se muestra en los ejemplos. Es posible encontrar instrucciones de descarga para la herramienta gráfica de diseño en la sección 2.1, "Revisión General de Descargables". No es necesario tener el diseñador gráfico para completar este tutorial.

Las máquinas de estado pueden ser

3.1. Ejemplo de Hello World

Una definición de proceso es un gráfico dirigido, compuesto por nodos y transiciones. El proceso hello world tiene 3 nodos. Para ver de qué forma se combinan estos componentes, vamos a comenzar con un simple proceso sin utilizar la herramienta gráfica de diseño. La siguiente imagen muestra la representación gráfica del proceso hello world:

Figura 3.1. Gráfico de proceso hello world

```
public void testHelloWorldProcess() {
  // Este método muestra una definición de proceso y una ejecución
  // de la definición de proceso.  La definición de proceso tiene
  // 3 nodos: un estado de inicio, un estado 's' y un
  // estado de término denominado 'end'.
  // La línea siguiente analiza un fragmento de texto xml en una
  // Definición de Proceso. Una Definición de Proceso es la
  // descripción formal de un proceso representado como un objeto java.
  ProcessDefinition processDefinition = ProcessDefinition.parseXmlString(
    "<process-definition>" +
    "  <start-state>" +
    "    <transition to='s' />" +
    "  </start-state>" +
    "  <state name='s'>" +
    "    <transition to='end' />" +
    "  </state>" +
    "  <end-state name='end' />" +
    "</process-definition>"
```

```
  );

  // La línea siguiente crea una ejecución de la definición de proceso.
  // Después de la construcción, la ejecución de proceso tiene una ruta
principal
  // de ejecución (=el token de raíz) que está posicionado en
  // start-state.
  ProcessInstance processInstance =
      new ProcessInstance(processDefinition);

  // Después de la construcción, la ejecución de proceso tiene una ruta
principal
  // de ejecución (=el token de raíz).
  Token token = processInstance.getRootToken();

  // También, después de la construcción, la ruta principal de ejecución está
posicionada
  // en el estado de inicio de la definición de proceso.
  assertSame(processDefinition.getStartState(), token.getNode());

  // Comencemos la ejecución de proceso, dejando el estado de inicio
  // en su transición predeterminada.
  token.signal();
  // El método de señal se bloquea hasta que la ejecución del proceso
  // entre a un estado de espera.

  // La ejecución de proceso habrá entrado al primer estado de espera
  // en el estado 's'. De modo que la ruta principal de ejecución está ahora
  // posicionada en estado 's'
  assertSame(processDefinition.getNode("s"), token.getNode());

  // Enviemos otra señal.  Esto reanuda la ejecución al
  // dejar el estado 's' en su transición predeterminada.
  token.signal();
  // Ahora el método de señal se devuelve porque la instancia de proceso
  // ha llegado a estado de término.

  assertSame(processDefinition.getNode("end"), token.getNode());
}
```

3.2. Ejemplo de base de datos

Una de las características básicas de jBPM es la capacidad de persistir en las ejecuciones de procesos en la base de datos cuando están en el estado de espera. El siguiente ejemplo muestra cómo almacenar una instancia de proceso en la base de datos jBPM. El ejemplo también sugiere un contexto en el cual esto se podría dar. Se crean métodos separados para diferentes partes de código de usuario. Por ejemplo, un trozo de código de usuario en una aplicación web inicia un proceso y persiste en la ejecución en la base de datos. Posteriormente, un bean accionado por mensajes carga la instancia de proceso desde la base de datos y reanuda su ejecución.

Es posible encontrar más información sobre la persistencia de jBPM en el capítulo 7, Persistencia.

```
public class HelloWorldDbTest extends TestCase {
```

```
  static JbpmConfiguration jbpmConfiguration = null;
```

```
  static {
    // Un archivo de configuración de muestra como este se puede encontrar en
    // 'src/config.files'.  Por lo general la información de configuración está
en
    // el archivo de recursos 'jbpm.cfg.xml', pero en este caso en la
configuración incluimos información
    // como una cadena XML.

    // Primero creamos una JbpmConfiguration en forma estática.  Una
JbpmConfiguration
    // se puede utilizar para todos los subprocesos en el sistema, por esa razón
podemos hacerla estática
    // con seguridad.

    jbpmConfiguration = JbpmConfiguration.parseXmlString(
      "<jbpm-configuration>" +

      // Un mecanismo de contexto jbpm separa el motor jbpm
      // principal de los servicios que jbpm utiliza en
      // el ambiente.

      "  <jbpm-context>" +
      "    <service name='persistence' " +
      "
factory='org.jbpm.persistence.db.DbPersistenceServiceFactory' />" +
      "  </jbpm-context>" +

      // También todos los archivos de recursos que son utilizados por jbpm
      // están referenciados desde jbpm.cfg.xml

      "  <string name='resource.hibernate.cfg.xml' " +
      "          value='hibernate.cfg.xml' />" +
      "  <string name='resource.business.calendar' " +
      "          value='org/jbpm/calendar/jbpm.business.calendar.properties' />"
+
      "  <string name='resource.default.modules' " +
      "          value='org/jbpm/graph/def/jbpm.default.modules.properties' />"
+
      "  <string name='resource.converter' " +
      "          value='org/jbpm/db/hibernate/jbpm.converter.properties' />" +
      "  <string name='resource.action.types' " +
      "          value='org/jbpm/graph/action/action.types.xml' />" +
      "  <string name='resource.node.types' " +
      "          value='org/jbpm/graph/node/node.types.xml' />" +
      "  <string name='resource.varmapping' " +
      "          value='org/jbpm/context/exe/jbpm.varmapping.xml' />" +
      "</jbpm-configuration>"
    );
  }

  public void setUp() {
    jbpmConfiguration.createSchema();
  }

  public void tearDown() {
    jbpmConfiguration.dropSchema();
  }
```


```
  public void testSimplePersistence() {
    // Entre las tres llamadas de método a continuación, todos los datos se
transfieren mediante la
    // base de datos.  Aquí, en esta prueba de unidad, se ejecutan estos tres
métodos
    // en forma consecutiva porque deseamos probar un un escenario completo de
    // proceso.  Pero en realidad, estos métodos representan
    // diferentes solicitudes a un servidor.

    // Dado que se parte con una base de datos limpia y vacía, primero debemos
    //  implementar el proceso.  En realidad, esto lo hace una vez el
    // desarrollador de proceso.
    deployProcessDefinition();

    // Supongamos que deseamos iniciar una instancia de proceso(=ejecución de
proceso)
    // cuando un usuario envía un formulario en una aplicación web...
    processInstanceIsCreatedWhenUserSubmitsWebappForm();

    // Entonces, posteriormente, a la llegada de un mensaje asincrónico la
    // ejecución debe continuar.
    theProcessInstanceContinuesWhenAnAsyncMessageIsReceived();
  }

  public void deployProcessDefinition() {
    // Esta prueba muestra una definición de proceso y una ejecución
    // de la definición de proceso.  La definición de proceso tiene
    // 3 nodos: un estado de inicio sin nombre, a estado 's' y un
    // estado de término denominado 'end'.
    ProcessDefinition processDefinition = ProcessDefinition.parseXmlString(
      "<process-definition name='hello world'>" +
      "  <start-state name='start'>" +
      "    <transition to='s' />" +
      "  </start-state>" +
      "  <state name='s'>" +
      "    <transition to='end' />" +
      "  </state>" +
      "  <end-state name='end' />" +
      "</process-definition>"
    );

    // Busque el generador de contexto de persistencia pojo que está configurado
a continuación
    JbpmContext jbpmContext = jbpmConfiguration.createJbpmContext();
    try {
      // Implemente la definición de proceso en la base de datos
      jbpmContext.deployProcessDefinition(processDefinition);

    } finally {
      // Separe el contexto de persistencia pojo.
      // Esto incluye limpiar el SQL para insertar la definición de proceso
      // en la base de datos.
      jbpmContext.close();
    }
  }

  public void processInstanceIsCreatedWhenUserSubmitsWebappForm() {
    // El código en este método podría estar dentro de una struts-action
```

```
    // o en un bean admdinistrado con JSF.

    // Busque el generador de contexto de persistencia pojo que está configurado
antes
    JbpmContext jbpmContext = jbpmConfiguration.createJbpmContext();
    try {

      GraphSession graphSession = jbpmContext.getGraphSession();

      ProcessDefinition processDefinition =
          graphSession.findLatestProcessDefinition("hello world");

      // Con la definición de proceso que recuperamos desde la base de datos,
podemos
      // crear una ejecución de la definición de proceso tal como en el
      // ejemplo de hello world (que fue sin persistencia).
      ProcessInstance processInstance =
          new ProcessInstance(processDefinition);

      Token token = processInstance.getRootToken();
      assertEquals("start", token.getNode().getName());
      // Iniciemos la ejecución de proceso
      token.signal();
      // Ahora el proceso está en el estado 's'.
      assertEquals("s", token.getNode().getName());

      // Ahora se guarda la processInstance en la base de datos.  De modo que
      // el estado actual de la ejecución del proceso se almacena en la
      // base de datos.
      jbpmContext.save(processInstance);
      // El siguiente método permite sacar la instancia de proceso
      // de la base de datos y reanuda la ejecución al entregar otra
      // señal externa.

    } finally {
      // Separe el contexto de persistencia pojo.
      jbpmContext.close();
    }
  }

  public void theProcessInstanceContinuesWhenAnAsyncMessageIsReceived() {
    // El código en este método podría ser el contenido de un bean accionado por
mensaje.

    // Busque el generador de contexto de persistencia pojo que está configurado
anteriormente
    JbpmContext jbpmContext = jbpmConfiguration.createJbpmContext();
    try {

      GraphSession graphSession = jbpmContext.getGraphSession();
      // Primero, necesitamos sacar la instancia de proceso desde la base de
datos.
      // Existen varias opciones para conocer la instancia de proceso con la que
estamos trabajando
      // en este caso.  Lo más sencillo en este simple caso de prueba es
simplemente buscar
      // la lista completa de instancias de proceso.  Eso debería darnos sólo un
      // resultado.  De modo que busquemos la definición de proceso.

      ProcessDefinition processDefinition =
```

```
          graphSession.findLatestProcessDefinition("hello world");

      // Ahora, busquemos todas las instancias de proceso de esta definición de
proceso.
      List processInstances =
          graphSession.findProcessInstances(processDefinition.getId());

      // Dado que sabemos que en el contexto de esta prueba de unidad existe
      // sólo una ejecución.  En la vida real, la processInstanceId se puede
      // extraer desde el contenido del mensaje que llegó o si el usuario toma
una opción.
      ProcessInstance processInstance =
          (ProcessInstance) processInstances.get(0);

      // Ahora podemos continuar con la ejecución. Oberve ue la instancia de
proceso
      // delega señales hacia la ruta principal de ejecución (=el token de raíz)
      processInstance.signal();

      // Luego de esta señal, sabemos que la ejecución de proceso debería haber
      // llegado en el estado de término.
      assertTrue(processInstance.hasEnded());

      // Ahora podemos actualizar el estado de la ejecución en la base de datos
      jbpmContext.save(processInstance);

    } finally {
      // Separe el contexto de persistencia pojo.
      jbpmContext.close();
    }
  }
}
```

3.3. Ejemplo de contexto: variables de proceso

Las variables de proceso contienen la información de contexto durante las ejecuciones de proceso. Las variables de proceso son similares a un `java.util.Map` que asocia nombres de variables a valores, los cuales son objetos java. Las variables de proceso se persisten como parte de la instancia de proceso. Con el fin de mantener la simpleza, en este ejemplo sólo mostramos el API trabajando con; sin persistencia.

Más información sobre variables se puede encontrar en el capítulo 10, Contexto

```
// Este ejemplo también parte desde el proceso hello world.
// En esta ocasión incluso sin modificación.
ProcessDefinition processDefinition = ProcessDefinition.parseXmlString(
  "<process-definition>" +
  "  <start-state>" +
  "    <transition to='s' />" +
  "  </start-state>" +
  "  <state name='s'>" +
  "    <transition to='end' />" +
  "  </state>" +
  "  <end-state name='end' />" +
  "</process-definition>"
);
```

```
ProcessInstance processInstance =
  new ProcessInstance(processDefinition);

// Obtenga la instancia de contexto desde la instancia de proceso
// para trabajar con las variables de proceso.
ContextInstance contextInstance =
  processInstance.getContextInstance();

// Antes de que el proceso haya salido del estado de inicio,
// vamos a establecer algunas variables de proceso en el
// contexto de la instancia de proceso.
contextInstance.setVariable("amount", new Integer(500));
contextInstance.setVariable("reason", "i met my deadline");

// Desde aquí en adelante, estas variables están asociadas con la
// instancia de proceso.  Ahora las variables de proceso están asequibles
// para el código de usuario mediante el API que se muestra, pero también en las
acciones
// e implementaciones de nodo.  Las variables de proceso también están
// almacenadas en la base de datos como parte de la instancia de proceso.

processInstance.signal();

// Las variables son accesibles mediante contextInstance.

assertEquals(new Integer(500),
             contextInstance.getVariable("amount"));
assertEquals("i met my deadline",
             contextInstance.getVariable("reason"));
```

3.4. Ejemplo de asignación de tarea

En el ejemplo siguiente mostraremos cómo se puede asignar una tarea a un usuario. Debido a la separación entre el motor de flujo de trabajo jBPM y el modelo organizacional, un lenguaje de expresión para calcular actores siempre sería demasiado limitado. En consecuencia, se debe especificar una implementación de AssignmentHandler para incluir el cálculo de actores para las tareas.

```
public void testTaskAssignment() {
  // el proceso que se muestra a continuación se basa en el proceso hello world.
  // El nodo de estado es reemplazado por un nodo de tarea.  El nodo de tarea
  // es un nodo en JPDL que representa un estado de espera y genera
  // tarea(s) que se deben llevar a cabo para que el proceso pueda continuar
  // ejecutándose.
  ProcessDefinition processDefinition = ProcessDefinition.parseXmlString(
    "<process-definition name='the baby process'>" +
    "  <start-state>" +
    "    <transition name='baby cries' to='t' />" +
    "  </start-state>" +
    "  <task-node name='t'>" +
    "    <task name='change nappy'>" +
    "      <assignment class='org.jbpm.tutorial.taskmgmt.NappyAssignmentHandler'
/>" +
    "    </task>" +
    "    <transition to='end' />" +
    "  </task-node>" +
```

```
    "  <end-state name='end' />" +
    "</process-definition>"
  );

  // Crear una ejecución de la definición de proceso.
  ProcessInstance processInstance =
      new ProcessInstance(processDefinition);
  Token token = processInstance.getRootToken();

  // Iniciemos la ejecución del proceso, dejando el estado de inicio
  // en su transición predeterminada.
  token.signal();
  // El método de señal se bloquea hasta que la ejecución de proceso
  // entra en el estado de espera.   En este caso, ese es el nodo de tarea.
  assertSame(processDefinition.getNode("t"), token.getNode());

  // Cuando la ejecución llegó al nodo de tarea, se creo una tarea 'change
nappy'
  // y se llamó al NappyAssignmentHandler para determinar
  // a quién se debe asignar la tarea.  El NappyAssignmentHandler
  // respondió 'papa'.

  // En un ambiente real, las tareas se obtendrían desde la
  // base de datos con los métodos en org.jbpm.db.TaskMgmtSession.
  // Dado que no queremos incluir la complejidad de persistencia en
  // el presente ejemplo, sólo tomamos la primera instancia de tarea de esta
  // instancia de proceso (sabemos que hay sólo una en este
  // escenario de prueba).
  TaskInstance taskInstance = (TaskInstance)
      processInstance
        .getTaskMgmtInstance()
        .getTaskInstances()
        .iterator().next();

  // Ahora, verificamos si taskInstance realmente se asignó a 'papa'.
  assertEquals("papa", taskInstance.getActorId() );

  // Ahora supongamos que 'papa' ha hecho sus deberes y marcamos la tarea task
  // como hecha.
  taskInstance.end();
  // Dado que esta era la última tarea por hacer, el término de esta
  // tarea desencadenó la continuación de la ejecución de la instancia de
proceso.

  assertSame(processDefinition.getNode("end"), token.getNode());
}
```

3.5. Ejemplo de acción personalizada

Las acciones son un mecanismo para unir su código java a un proceso jBPM. Las acciones pueden estar asociadas con sus propios nodos (si son pertinentes en la representación gráfica del proceso). O las acciones se pueden ubicar en eventos como: tomar una transición, salir o entrar a un nodo. En este caso, las acciones no son parte de la representación gráfica, pero se ejecutan cuando la ejecución activa los eventos en una ejecuión de proceso de tiempo de ejecución .

Empezaremos con un vistazo a la implementación de acción que vamos a utilizar en nuesto ejemplo: `MyActionHandler`. Esta implementación de manipulador de acciones no hace cosas espectaculares... sólo establece la variable buleana `isExecuted` en `true`. La variable `isExecuted` es estática, de modo que se puede acceder a ella desde el interior del manipulador de acciones así como desde la acción para verificar su valor.

Para obtener más información sobre las acciones, consulte la sección 9.5, "Acciones"

```
// MyActionHandler representa una clase que podría ejecutar
// algún código de usuario durante la ejecución de un proceso jBPM.
public class MyActionHandler implements ActionHandler {

  // Antes de cada test (en el setUp), el miembro isExecuted
  // se dejará en falso.
  public static boolean isExecuted = false;

  // La acción establecerá isExecuted en verdadero de modo que
  // se pueda mostrar la prueba de unidad cuando se está ejecutando
  // la acción.
  public void execute(ExecutionContext executionContext) {
    isExecuted = true;
  }
}
```

Como se menciona anteriormente, antes de cada test, estableceremos el campo estático `MyActionHandler.isExecuted` en falso;

```
  // Cada test se inicia estableciendo el miembro estático isExecuted
  // de MyActionHandler en falso.
  public void setUp() {
    MyActionHandler.isExecuted = false;
  }
```

Comenzaremos con una acción en una transición.

```
public void testTransitionAction() {
    // El proceso siguiente es una variante del proceso hello world.
    // Hemos agregado una acción en la transición desde el estado 's'
    // al estado de términio.  El propósito de este test es mortrar
    // lo fácil que es integrar código java en un proceso jBPM.
    ProcessDefinition processDefinition = ProcessDefinition.parseXmlString(
      "<process-definition>" +
      "  <start-state>" +
      "    <transition to='s' />" +
      "  </start-state>" +
      "  <state name='s'>" +
      "    <transition to='end'>" +
      "      <action class='org.jbpm.tutorial.action.MyActionHandler' />" +
      "    </transition>" +
      "  </state>" +
      "  <end-state name='end' />" +
      "</process-definition>"
    );

    // Comencemos con una nueva ejecución para la definición de proceso.
    ProcessInstance processInstance =
      new ProcessInstance(processDefinition);
```

```
    // La siguiente señal hace que la ejecución salga del estado
    // start e ingrese al estado 's'
    processInstance.signal();

    // Aquí mostramos que MyActionHandler todavía no se ha ejecutado.
    assertFalse(MyActionHandler.isExecuted);
    // ... y que la ruta principal de ejecución está posicionada en
    // el estado 's'
    assertSame(processDefinition.getNode("s"),
               processInstance.getRootToken().getNode());

    // La siguiente señal desencadena la ejecución del token de raíz.  La llave
toma la transición con la
    // acción y la acción se ejecuta durante el
    // llamado al método de señal.
    processInstance.signal();

    // Aquí podemos ver cuál MyActionHandler se ejecutó durante
    // la llamada al método de señal.
    assertTrue(MyActionHandler.isExecuted);
  }
```

El siguiente ejemplo muestra la misma acción, pero ahora las acciones se ubican en los eventos `enter-node` y `leave-node` respectivamente. Cabe mencionar que un nodo tiene más de un tipo de evento en contraste con una transición, la cual correspondía a un solo evento. En consecuencia las acciones ubicadas en un nodo se deben poner en un elemento de evento.

```
ProcessDefinition processDefinition = ProcessDefinition.parseXmlString(
  "<process-definition>" +
  "  <start-state>" +
  "    <transition to='s' />" +
  "  </start-state>" +
  "  <state name='s'>" +
  "    <event type='node-enter'>" +
  "      <action class='org.jbpm.tutorial.action.MyActionHandler' />" +
  "    </event>" +
  "    <event type='node-leave'>" +
  "      <action class='org.jbpm.tutorial.action.MyActionHandler' />" +
  "    </event>" +
  "    <transition to='end'/>" +
  "  </state>" +
  "  <end-state name='end' />" +
  "</process-definition>"
);

ProcessInstance processInstance =
  new ProcessInstance(processDefinition);

assertFalse(MyActionHandler.isExecuted);
// La siguiente señal hace que la ejecución salga del modo
// inicio e ingrese al estado 's'.  De modo que se ingrese al estado 's'
// y por ende se ejecute la acción.
processInstance.signal();
assertTrue(MyActionHandler.isExecuted);
```

```
// Restablezcamos el MyActionHandler.isExecuted
MyActionHandler.isExecuted = false;

// La siguiente señal hace que la ejecución salga del
// estado 's'.  De modo que la acción se ejecute nuevamente.
processInstance.signal();
// Voila.
assertTrue(MyActionHandler.isExecuted);
```

Capítulo 4. Programación Orientada a Objetos

4.1. Introducción

El presente capítulo se puede considerar como el manifiesto de JBoss jBPM. Entrega una reseña general de la visión y de las ideas subyacentes de la actual estrategia y direcciones futuras del proyecto JBoss jBPM. Esta visión difiere significativamente del enfoque tradicional.

Primero que nada, creemos en lenguajes de proceso múltiples. Existen diferentes ambientes y diferentes propósitos que exigen su propio lenguaje de proceso específico.

En segundo lugar, la Programación Orientada a Objetos es una nueva técnica de implementación que sirve como base para todos los lenguajes de proceso basados en gráficos.

El principal beneficio de nuestro enfoque es que éste define una tecnología de base para todos los tipos de lenguajes de proceso.

El desarrollo actual de software depende más y más de lenguajes específicos de los dominios. Un programador Java típico utiliza unos cuantos lenguajes específicos para dominios. Los archivos XML en un proyecto que se usan como entradas para varios marcos se pueden considerar como lenguajes específicos para el dominio.

Figura 4.1. Posicionamiento de lenguajes basados en objetos

Los lenguajes específicos de dominio para flujo de trabajo, BPM, orquestación y pageflow se basan en la ejecución de un gráfico dirigido. Pero no en el caso de otros como archivos de asociación de hibernación y archivos de configuración ioc. La Programación Orientada a Objetos es la base de todos los lenguajes específicos de dominio que se basan en la ejecución de un gráfico.

La Programación Orientada a Objetos es una técnica muy simple que describe la forma en que los gráficos se pueden definir y ejecutar en un sencillo lenguaje de programación OO.

En la Sección 4.5, "Dominios de Aplicación", analizaremos los lenguajes de proceso de uso más frecuente los cuales se pueden implementar utilizando Programación Orientada a Objetos como flujo de trabajo, BPM, orquestación y pageflow.

4.1.1. Lenguajes específicos de dominio

Cada lenguaje de proceso se puede considerar como un Lenguaje Específico de Dominio (DSL). La perspectiva DSL le da a los programadores una clara idea de como se relacionan los lenguajes de proceso con la simple programación OO.

Esta sección podría dar la impresión de que nos concentramos exclusivamente en los ambientes de programación. Nada podría ser menos cierto. La Programación Orientada a Objetos incluye todo el continuo desde el producto BPM de las bibliotecas API hasta los productos completos de la suite BMP. Los productos de la suite BPM son ambientes de desarrollo de software completos que se centran en procesos de negocio. En ese tipo de productos se evita, en la medida de lo posible, la codificación en lenguajes de

programación.

Un aspecto importante de los lenguajes específicos de dominio es que cada lenguaje tiene una gramática determinada. Dicha gramática se puede expresar como un modelo de dominio. En el caso de java esto es Clase, Método, Campo, Constructor,... En jPDL esto es Nodo, Transición, Acción,... En reglas, esto es condición, consecuencia,...

La idea principal de DSL es que los desarrolladores piensen en esas gramáticas en el momento de crear artefactos para un lenguaje específico. El IDE se construye en torno a la gramática de un lenguaje. En consecuencia, puede haber diferentes editores para crear los artefactos. Por ejemplo, un proceso jPDL tiene un editor gráfico y un editor de fuente XML. También puede haber diferentes formas de almacenar el mismo artefacto: para jPDL, esto podría ser un archivo XML de proceso o el gráfico de objeto serializado de nodos y objetos de transición. Otro ejemplo (teórico) es java: se puede utilizar el formato de archivo de clase de java en el sistema. Cuando un usuario inicia el editor, se generan las fuentes. Cuando un usuario guarda se guarda la clase compilada....

Si bien hace diez años la mayor parte del tiempo de un programador se destinaba a escribir código, ahora se ha producido un cambio hacia el aprendizaje y uso de lenguajes específicos de dominio. Dicha tendencia se mantendrá y el resultado es que los programadores tendrán una opción entre marcos y escribir software en la plataforma host. JBoss SEAM es un gran paso en esa dirección.

Algunos de esos lenguajes se basan en la ejecución de un gráfico. Por ejemplo, jPDL para flujo de trabajo en Java, BPEL para orquestación de servicios, SEAM pageflow,... la Programación Orientada a Objetos es una base común para todos estos tipos de lenguajes específicos de dominio.

En el futuro, un desarrollador podrá elegir para cada lenguaje un editor de su preferencia. Por ejemplo, un programador experimentado probablemente prefiera editar java en el formato fuente del archivo porque eso funciona muy rápido. Pero un desarrollador java con menos experiencia podría elegir un editor tipo "point and click" para componer una funcionalidad que de como resultado una clase java. La edición de la fuente java es mucho más flexible.

Otra forma de analizar estos lenguajes específicos de dominio (incluidos los lenguajes de programación) es desde la perspectiva de la estructuración del software. La Programación Orientada a los Objetos (OOP) agrega estructura mediante métodos de agrupamiento con sus datos. La Programación Orientada al Aspecto (AOP) agrega una forma de extraer inquietudes transversales. Los marcos de Dependency Injection (DI) y Inversion of Control (IoC) agregan una conexión fácil de gráficos de objetos. También los lenguajes de ejecución basados en gráficos (según se analiza en el presente documento) pueden ser útiles para abordar la complejidad mediante la estructuración de parte de su proyecto de software en torno a la ejecución de un gráfico.

Para obtener más información sobre Lenguajes Específicos de Dominio (DSL), consulte Martin Fowler's bliki.

4.1.2. Características de los lenguajes basados en gráficos

Existen numerosos lenguajes de proceso basados en gráficos. Existen grandes diferencias en el ambiente y el enfoque. Por ejemplo, BPEL está pensado como un componente de orquestación de servicios basado en XML sobre una arquitectura de Enterprise Service

Bus (ESB). Y un lenguaje de proceso pageflow podría definir cómo se pueden navegar las páginas de una aplicación web. Estos son dos ambientes completamente diferentes.

A pesar de todas estas diferencias, existen dos características que se pueden encontrar en casi todos los lenguajes de proceso: soporte para estados de espera y una representación gráfica. Esto no es una coincidencia, porque son precisamente esas dos características las que no cuentan con un soporte suficiente en los lenguajes de programación simple Orientada a Objetos (OO) como Java.

La Programación Orientada a Objetos es una técnica para implementar estas dos características en un lenguaje de programación OO. La dependencia de Programación Orientada a Objetos de la programación OO implica que todos los lenguajes de proceso implementados sobre la Programación Orientada a Objetos deberán desarrollarse en OOP. Pero esto no significa que los lenguajes de proceso propiamente tales expongan algo de esta naturaleza OOP . Por ejemplo, BPEL no tiene ninguna relación con la programación OO y se puede implementar sobre la Programación Orientada a Objetos.

4.1.2.1. Soporte para estados de espera

Un lenguaje de programación imperativo como Java se utiliza para expresar una secuencia de instrucciones que un sistema debe ejecutar. No hay instrucción de espera. Un lenguaje imperativo es perfecto para describir, por ejemplo, un ciclo de respuesta de solicitud en un servidor. El sistema está ejecutando en forma continua la secuencia de instrucciones hasta que se procesa la solicitud y se termina la respuesta.

Pero una solicitud de ese tipo, por lo general, es parte de un escenario mayor. Por ejemplo, un cliente envía una orden de compra, la cual debe ser validada por un administrador de órdenes de compra. Luego de la aprobación la información se debe ingresar en el sistema ERP. Muchas solicitudes al servidor son parte del mismo escenario mayor.

Entonces, los lenguajes de proceso son lenguajes para describir el escenario mayor. Una distinción muy importante que debemos hacer aquí son los escenarios que son ejecutables en un sistema (orquestación) y escenarios que describen el protocolo entre sistemas múltiples (coreografía). La técnica de implementación de Programación Orientada a Objetos sólo se concentra en lenguajes de proceso que sean ejecutables en una máquina (orquestación).

De modo que un proceso de orquestación describe el escenario general en términos de un sistema. Por ejemplo: se inicia un proceso cuando un cliente envía una orden de compra. El paso siguiente en el proceso es la aprobación del administrador de órdenes de compra. Entonces el sistema debe agregar una entrada en la lista de tareas del administrador de órdenes de compra y **esperar** hasta que el administrador de órdenes entrega el input esperado. Cuando se recibe el input, el proceso continua con la ejecución. Ahora se envía un mensaje al sistema ERP y nuevamente este sistema **espera** hasta que se recibe la respuesta.

De modo que para describir el escenario para un sistema, necesitamos un mecanismo para manejar los estados de espera.

En la mayoría de los dominios de aplicación, la ejecución se debe persistir durante los estados de espera. Por esa razón no basta con subprocesos de bloqueo. Los programadores en Java más hábiles podrían pensar en los métodos Object.wait() y

Object.notify();. Estos se podrían utilizar para estimular los estados de espera, pero el problema es que no se puede persistir en los subprocesos.

Las continuaciones son una técnica para hacer persistir el subproceso (y las variables de contexto). Esto podría ser suficiente para resolver el problema de estado de espera. Pero como veremos en la sección siguiente, también es importante contar con una representación gráfica para muchos de los dominios de aplicación. Y las continuaciones son una técnica que se basa en la programación imperativa, de modo que no es adecuada para la representación gráfica.

Entonces un aspecto importante del soporte para los estados de espera es que las ejecuciones necesitan ser persistibles. Distintos dominios de aplicación podrían tener diferentes requerimientos para persistir ese tipo de ejecución. Para la mayoría de las aplicaciones de flujo de trabajo, BPM y orquestación, la ejecución necesita persistir en una base de datos relacional. Por lo general, una transición de estado en la ejecución del proceso corresponde a la transacción en la base de datos.

4.1.2.2. Representación Gráfica

Algunos aspectos del desarrollo de software pueden beneficiarse muy bien de un enfoque basado en gráficos. La Administración de Procesos de Negocio BPM es uno de los dominios de aplicación más obvios de los lenguajes basados en gráficos. En ese ejemplo, la comunicación entre un analista de negocios y el desarrollador mejora al utilizar el diagrama basado en gráficos del proceso de negocios como lenguaje en común. Consulte también la sección 4.5.1, "Administración de Proceso de Negocios (BPM)".

Otro aspecto que se puede ver beneficiado por una representación gráfica es el pageflow. En este caso, las páginas, la navegación y los comandos de acción se muestran y aparecen juntos en la representación gráfica.

En la Programación Orientada a Objetos nos enfocamos a diagramas de gráficos que representan alguna forma de ejecución. Eso es una clara diferenciación con los diagramas de clase UML, los cuales representan un modelo estático de la estructura de datos OO.

También la representación gráfica se puede visualizar como una característica faltante en la programación OO. No existe una forma adecuada en la cual se pueda representar en forma gráfica la ejecución de un programa OO. De modo que no hay relación directa entre un programa OO y la vista gráfica.

En la Programación Orientada a Objetos, la descripción del gráfico es esencial y es un artefacto real de software como un archivo XML que describe el gráfico de proceso. Dado que la vista gráfica es una parte intrinseca del software, siempre está sincronizada. No hay necesidad de una traducción manual desde los requerimientos técnicos a un diseño de software. El software está estructurado en torno al gráfico.

4.2. Programación Orientada a Objetos

Lo que presentamos aquí es una técnica de implementación para lenguajes de ejecución basados en gráficos. La técnica que se presenta aquí se basa en la interpretación de un gráfico. Otras técnicas para la ejecución de un gráfico se basan en colas de mensajes o generación de código.

Esta sección explica la estrategia de como se puede implementar la ejecución de gráficos sobre un lenguaje de programación OO. Para quienes están familiarizados con los patrones de diseño, es una combinación del patrón de comando y la cadena de patrón de responsabilidad.

Comenzaremos con el modelo más simple y luego los extenderemos gradualmente.

4.2.1. La estructura del gráfico

Primero, la estructura del gráfico se representa con las clases `Node` y `Transition`. Una transición tiene una dirección de modo que los nodos tienen transacciones salientes y entrantes.

Figura 4.2. Clases de Nodo y Transición

Un nodo es un comando y tiene un método `execute` . Se supone que las Subclases de Nodos sobreescriben el método de ejecución para implementar algún comportamiento específico para ese tipo de nodo.

4.2.2. Una ejecución

El modelo de ejecución que definimos en esta estructura de gráfico podría parecer similar a máquinas de estado finito o diagramas de estado UML. De hecho, la Programación Orientada a Objetos se puede utilizar para implementar ese tipo de comportamientos, pero también sirve para mucho más.

Una ejecución (también conocida como token o testigo) se representa con una clase denominada `Execution`. Una ejecución tiene una referencia al nodo vigente.

Figura 4.3. Clase de la Ejecución

Las transiciones pueden hacer pasar la ejecución desde un nodo fuente a un nodo de destino con el método `take`.

Figura 4.4. El método take de Transición

Cuando la ejecuión llega a un nodo, ese nodo se ejecuta. El método de ejecución del nodo también es responsable de propagar la ejecución. Propagar la ejecución significa que un nodo puede traspasar la ejecución que llegó al nodo a través de una de sus transiciones salientes al próximo nodo.

Figura 4.5. El método execute del Nodo

Cuando un método ejecutar de un nodo no logra propagar la ejecución, se comporta como un estado de espera. También cuando se crea una nueva ejecución, se inicializa en un nodo de inicio y luego espera un evento.

Un evento se entrega a una ejecución y puede desencadenar el inicio de una ejecución. Si el evento entregado a una ejecuión se relaciona con una transición saliente del nodo vigente, la ejecuión toma dicha transición. Luego la ejecución continua propagándose

hasta que entra en otro nodo que se comporta como un estado de espera.

Figura 4.6. El método de evento de Ejecución

4.2.3. Un lenguaje de proceso

Entonces ya podemos ver que hay soporte para las dos principales características: los estados de espera y la representación gráfica. Durante los estados de espera una Ejecución sólo apunta a un nodo en el gráfico. Tanto el gráfico de proceso como la Ejecución se pueden persistir: por ejemplo, a una base de datos relacional con una hibernación tipo asignador O/R o serializando el gráfico del objeto en un archivo. También es posible ver que los nodos y las transiciones forman un gráfico y, por ende, existe un acople directo con una representación gráfica.

Un lenguaje de proceso no es más que un conjunto de implementaciones de Nodo. Cada implementación de Nodo corresponde con un constructo de proceso. El comportamiento exacto del constructo de proceso se implementa sobreescribiendo el metodo de ejecución.

Aquí mostramos un ejemplo de lenguaje de proceso con 4 constructores de proceso: un estado de inicio, una decisión, una tarea y un estado de término. Este ejemplo no está relacionado con el lenguaje de proceso jPDL.

Figura 4.7. Un ejemplo de lenguaje de proceso

Los objetos de nodo concretos ahora se pueden utilizar para crear gráficos de proceso en nuestro ejemplo de lenguaje de proceso.

Figura 4.8. Un ejemplo de proceso

Al crear una nueva ejecución para este proceso, comenzamos posicionando la ejecución en el nodo de inicio. De modo que siempre y cuando la ejecución no reciba un evento, la ejecución permanece posicionada en el estado de inicio.

Figura 4.9. Una nueva ejecución

Ahora veamos lo que sucede cuando se activa un evento. En esta situación inicial, activamos el evento predeterminado que corresponde con la transición predeterminada.

Esto se logra invocando el método de evento en el objeto de ejecución. El método de evento propaga la búsqueda de la transición saliente predeterminada y traspasa la ejecución por medio de la transición al invocar el método take en la transición y transfiriéndose como un parámetro.

La transición traspasa la ejecución al nodo de decisión e invoca el método execute. Supongamos que la implementación execute de la decisión realiza un cálculo y decide propagar la ejecución enviando el evento 'yes'- a la ejecución. Esto hace que la ejecuión continue con la transición 'yes' y la ejecución llega en el 'doubleCheck' de la tarea.

Supongamos que la implementación del nodo de tarea doubleCheck agrega una entrada

a la lista de tareas del verificador y luego espera el input del verificador al no seguir propagando la ejecución.

Ahora, la ejecución permanece posicionada en el nodo de tarea doubleCheck. Todas las invocaciones anidadas comienzan a devolverse hasta que vuelve el método original de evento.

Figura 4.10. Una ejecución en el estado de espera 'doubleCheck'

4.2.4. Acciones

En algunos dominios de aplicación debe existir una forma de incluir la ejecución de lógica de programación sin necesidad de introducir un nodo para ello. En BPM por ejemplo este es un aspecto muy importante. El analista de negocios está a cargo de la representación gráfica y el desarrollador es reponsable de que sea ejecutable. No es aceptable si el desarrollador debe cambiar el diagrama gráfico para incluir un detalle técnico en el cual no está interesado el analista de negocios.

Una `Action` también es un comando con un método execute. Las acciones pueden estar asociadas con eventos.

Existen 2 eventos básicos activados por la clase Nodo mientras se ejecuta una ejecución: `node-leave` y `node-enter`. Junto con los eventos que hacen que se tomen las transiciones, esto otorga libertad para inyectar lógica de programación en la ejecución de un gráfico.

Figura 4.11. Acciones que están ocultas en la vista gráfica

Cada evento puede estar asociado con una lista de acciones. Todas las acciones se ejecutan con las activaciones de eventos.

4.2.5. Ejemplo de código

Con el fin de que la gente se familiarice con los principios de la Programación Orientada a Gráficos hemos desarrollado estas 4 clases en menos de 130 líneas de código. Basta con sólo leer el código para tener una idea o puede comenzar a jugar con ellas e implementar sus propios tipos de nodo.

Aquí está el ejemplo de código:

- Execution.java
- Node.java
- Transition.java
- Action.java

También es posible descargar todo el proyecto fuente (297KB) y comenzar a jugar con el directamente. Incluye el proyecto eclipse, así que basta con importarlo a su eclipse como proyecto para comenzar a usarlo. Asimismo, existe una serie de pruebas que muestran ejecuciones básicas de proceso y los conceptos de ejecución avanzada de gráficos en la sección siguiente.

4.3. Extensiones de Programación Avanzada Orientada a

Objetos

En la sección anterior se presentó el sencillo modelo de Programación Orientada a Objetos en su forma más simple. Esta sección analiza varios aspectos de los lenguajes basados en gráficos y cómo se puede utilizar o extender la Programación Orientada a Objetos para satisfacer estos requerimientos.

4.3.1. Variables de proceso

Las variables de proceso mantienen los datos contextuales de una ejecución de proceso. En un proceso de siniestro de seguro, el 'monto solicitado', 'monto aprobado' y 'a pagar' podrían ser buenos ejemplos de variables de proceso. En muchos sentidos, son similares a los campos miembros de una clase.

La Programación Orientada a Objetos se puede extender fácilmente con soporte para variables de proceso mediante la asociación de pares de valores claves que están asociados con una ejecución. Las rutas concurrentes de ejecución y composición de proceso complican un poco las cosas. Las reglas de scoping definen la visibilidad del proceso en caso de que haya rutas concurrentes de ejecución o subprocesos.

'Patrones de Datos de Flujo de trabajo' es un amplio resultado de investigación sobre los tipos de scoping que se pueden aplicar a las variables de proceso en el contexto de subprocesamiento y ejecuciones concurrentes.

4.3.2. Ejecuciones concurrentes

Supongamos que está desarrollando un proceso de 'venta' con un lenguaje de proceso basado en gráficos para flujo de trabajo. Luego de que el cliente envía la orden o pedido existen actividades posteriores para facturarle al cliente y también hay una secuencia de actividades para despachar los artículos al cliente. Como puede imaginarse, las actividades de facturación y despacho se pueden hacer en paralelo.

En ese caso una ejecución es suficiente para mantener un registro del estado de todo el proceso. Revisemos los pasos para extender el modelo de Programación Orientada a Objetos y agreguemos soporte a las ejecuciones concurrentes.

Primero, transformemos el nombre de la ejecución en una ruta de ejecución. Luego podemos introducir un concepto nuevo denominado ejecución de proceso. Una ejecución de proceso representa una ejecución completa de un proceso y contiene muchas rutas de ejecución.

Las rutas de ejecución se pueden ordenar en forma jerárquica. Es decir, una ruta de ejecución raíz se crea cuando se instancia nueva ejecución de proceso. Cuando la ruta de ejecución raíz se bifurca en múltiples rutas de ejecución concurrentes, la raíz es la principal y las rutas de ejecución recientemente creadas son las secundarias de la raíz. De esta forma la implementación se hace muy sencilla: la implementación de una unión tiene que verificar si todas las rutas de ejecución secundarias están ya posicionadas en su nodo combinado. Si así es, la ruta de ejecución primaria puede reanudar la ejecución saliendo del nodo combinado.

Mientras las rutas de ejecución jerárquicas y la implementación combinada basada en rutas de ejecución secundarias abarcan una gran parte de los casos de uso, quizá sean aconsejables otros comportamientos de concurrencia en circunstancias específicas. Por

ejemplo, cuando existen múltiples combinaciones se relaciones con una sola división. En tal situación, se necesitan otras combinaciones de datos de tiempo de ejecuión e implementaciones combinadas.

Figura 4.12. Accciones que están normalmente ocultas de la vista gráfica

Las rutas de ejecución múltiples concurrentes a menudo se mezclan con una programación de subproceso múltiple. Especialmente en el contexto de flujo de trabajo y BPM, estas son bastante diferentes. Un proceso especifica una máquina de estado. Consideremos por un momento que una máquina de estado está siempre en un estado estable y que las transiciones de estado son instantáneas. Luego es posible interpretar rutas concurrentes de ejecución consultando los eventos que causan las transiciones de estado. La ejecución concurrente entonces significa que los eventos que se pueden manipular no están relacionados entre las rutas de ejecución concurrentes. Ahora supongamos que las transiciones de estado en la ejecución de proceso se relaciona con una transición de base de datos (según se explica en la sección 4.3.6, "Persistencia y Transacciones"), luego uno ve que la programación de subproceso múltiple ni siquiera se necesita realmente para soportar las rutas concurrentes de ejecución.

4.3.3. Composición del proceso

La idea de ejecuciones de proceso

4.3.4. Ejecución síncrona

4.3.5. Continuaciones asíncronas

4.3.6. Persistencia y Transacciones

4.3.7. Servicios y entorno

4.4. Arquitectura

4.5. Dominios de aplicación

4.5.1. Business Process Management (BPM)

El objetivo de BPM es que una organización funcione en forma más eficiente. El primer paso es analizar y describir cómo se hace el trabajo dentro de la organización. La definición de un proceso de negocios es una descripción de la forma en que las personas y los sistemas trabajan en conjunto para llevar a cabo un trabajo específico. Una vez que se han descrito los procesos de negocios puede comenzar la búsqueda de optimizaciones.

En ocasiones los procesos de negocio han evolucionado en forma orgánica y basta con sólo mirar el proceso general de negocios para notar algunas ineficiencias evidentes. La búsqueda de modificaciones que hagan más eficiente un proceso de negocios se denomina Reingeniería de Proceso de Negocios (BPR). Una vez que se automatiza una

buena parte del proceso de negocios las estadísticas y los análisis retrospectivos pueden ayudar a encontrar e identificar estas ineficiencias.

Otra forma de mejorar la eficiencia puede ser mediante la automatización de la totalidad o parte del proceso de negocios utilizando tecnologías de información.

La automatización y modificación de los procesos de negocio son las formas más comunes de lograr que una organización funcione en forma más eficiente.

Los gerentes continuamente desglosan trabajos en pasos que deben ser ejecutados por los miembros de sus equipos de trabajo. Por ejemplo un gerente de desarrollo de software que organiza un evento de formación de equipo. En ese caso la descripción del proceso de negocios podría existir sólo en la cabeza del gerente. Otras situaciones como el manejo de un siniestro de seguro en el caso de una gran compañía de seguros requiere un enfoque BPM más formal.

La ganancia total que se puede obtener al administrar así los procesos de negocio son los mejoramientos de eficiencia multiplicados por la cantidad de ejecuciones del proceso. El costo de administrar procesos de negocio de manera formal es el esfuerzo adicional que se destina a analizar, describir, mejorar y automatizar el proceso de negocios. De modo que el costo debe tomarse en consideración al determinar qué procesos se seleccionarán para una administración y/o automatización formal.

4.5.1.1. Objetivos de los sistemas BPM

4.5.1.2. Proceso de desarrollo de procesos

4.5.2. Orquestación de servicios

La orquestación en ocasiones se considera en el contexto de la orquestación de servicios.

Figura 4.13. Servicio

4.5.2.1. Orquestación comparado con Coreografía

4.5.3. Pageflow

4.5.4. Programación visual

...orientada a desarrolladores menos experimentados.

En mi opinion la comunidad java actual es una comunidad relativamente pequeña, formada por unos cuantos usuarios aventajados. Por supuesto "point and click" (*apuntar y hacer clic)* tiene una serie de restricciones que no permiten manejar todos los constructos de Java. Pero le abre la puerta a una nueva generación de desarrolladores de software (menos experimentados). De modo que los diferentes editores para un sólo lenguaje puede enfocarse a diferentes tipos/niveles de desarrolladores.

4.6. Incorporación de lenguajes basados en gráficos

Cuando el motor del BPM se puede integrar completamente al proyecto de desarrollo de software y cuando incluso las tablas de la base de datos del motor BPM se pueden integrar a la base de datos del proyecto estamos hablando de motor BMP Incorporable. Ese es nuestro objetivo con la Programación Orientada a Objetos: una base común para implementar lenguajes basados en gráficos.

4.7. Mercado

4.7.1. El mejor lenguaje de proceso

Tradicionalmente los proveedores han buscado desarrollar el mejor lenguaje de proceso. El enfoque es especificar un lenguaje de proceso como un conjunto de constructos. Cada constructo tiene una representación gráfica y un comportamiento de tiempo de ejecución. En otras palabras, cada constructo es un tipo de nodo en el gráfico de proceso y un lenguaje de proceso es sólo un conjunto de constructos de nodo.

La idea fue que los proveedores buscaran el mejor conjunto de constructos de proceso para formar un lenguaje de proceso de aplicación universal. Esta visión todavía está presente hoy y la denominamos como la búsqueda del mejor lenguaje de proceso.

Creemos que el enfoque no debería centrarse en buscar el mejor lenguaje de proceso, sino más bien en encontrar una base común que se puede utilizar como una base para lenguajes de proceso en diferentes escenarios y diferentes ambientes. La Programación Orientada a Objetos como la presentamos a continuación se debe considerar como dicha base.

4.7.2. Fragmentación

El paisaje actual del flujo de trabajo, las soluciones de BPM y de orquestación está completamente fragmentado. En esta sección describimos dos dimensiones de esta fragmentación. La primera dimensión se denomina continuo de producto BPM y se muestra en la siguiente figura. El término fue acuñado originalmente por Derek Miers y Paul Harmon en 'The 2005 BPM Suites Report'.

A la izquierda se pueden ver los lenguajes de programación. Este lado del continuo está enfocado hacia los desarrolladores de IT. Los lenguajes de programación son las más flexibles y se integran completamente con los demás softwares desarrollados para un proyecto particular, pero se necesita bastante programación para implementar un proceso de negocios.

A la derecha están las suites BPM. Dichas suites BPM son ambientes completos para el desarrollo de software destinados a analistas de negocio. El software se desarrolla en torno a los procesos de negocio. No se necesita programación para crear software ejecutable en estas suites BPM.

Figura 4.14. El continuo de producto BPM.

Los productos tradicionales marcan 1 lugar en el continuo de producto de BPM. Para estar completos estos productos tienden a apuntar hacia el extremo derecho del

continuo. Esto es problemático porque da como resultado un systema monolítico que es muy difícil de integrar a un proyecto, ya que combina software OOP con procesos de negocio.

La Programación Orientada a Objetos se puede construir como una simple biblioteca que se integra perfectamente con un ambiente sencillo de programación. Por otra parte, esta biblioteca se puede organizar y pre-implementar en un servidor para transformarse en un servidor BPM. Luego se agregan otros productos y se organizan con el servidor BPM para transformarse en una suite BPM completa.

El resultado neto es que las soluciones basadas en Programación Orientada a Objetos puede enfocarse en todo el continuo. Dependiendo de los requerimientos en un proyecto en particular, la suite BPM se puede modificar y personalizar de acuerdo al nivel correcto de integración con el ambiente de desarrollo de software.

La otra dimensión de la fragmentación es el dominio de aplicación. Como se muestra anteriormente, un dominio de aplicación BPM es completamente diferente a orquestación o pageflow. También en esta dimensión, los productos tradicionales se concentran en un solo dominio de aplicación, en circunstancias que la Programación Orientada a Objetos abarca la gama completa.

Si esto lo presentamos en un gráfico entrega una clara idea de la actual fragmentación del mercado. En el mercado de lenguajes basados en gráficos los precios son altos y los volúmenes son bajos. Esta comenzando la consolidación y esta tecnología está destinada a transformarse en una base común para lo que hoy es un paisaje de mercado fragmentado y confuso.

Figura 4.15. Dos dimensiones de fragmentación.

4.7.3. Otras técnicas de implementación

Basadas en colas de mensajes.

Basadas en generación de código.

Capítulo 5. Implementación

jBPM es un motor BPM incorporable, lo que significa que se puede tomar jBPM e incorporarlo en su propio proyecto java en lugar de instalar un producto distinto e integrarlos. Uno de los aspectos clave que posibilita esto es la minimización de las dependencias. En este capítulo se analizan las bibliotecas jbpm y sus dependencias.

5.1. Ambiente tiempo de ejecución Java

jBPM 3 necesita J2SE 1.4.2+

5.2. Bibliotecas jBPM

jbpm-[version].jar es la biblioteca que contiene la funcionalidad esencial jbpm.

jbpm-identity-[version].jar es la biblioteca (opcional) que contiene un componente de identidad como se describe en la sección 11.11, "El componente de identidad".

5.3. Bibliotecas de terceros

En una implementación mínimo es posible crear y ejecutar procesos con jBPM solamente poniendo el commons-logging y la biblioteca dom4j en su classpath. Considere que no se soporta la persistencia de procesos a una base de datos. La biblioteca dom4j se puede eliminar si no utiliza el análisis xml de proceso, sino que construye su gráfico de objeto en forma programática.

Tabla 5.1.

Biblioteca	Uso	Descripción
commons-logging.jar	logging en jbpm y hibernate	El código jBPM se registra con commons logging. La biblioteca configurar para despachar los registros para, por ejemplo, regi ... Consulte la la guía de usuario apache commons para obtener r configurar el registro commons. Si está acostrumbrado a log4 poner la biblioteca log4j y log4j.properties en el classp automáticamente detecta esto y utiliza dicha configuración.
dom4j-1.6.1.jar	Definiciones de proceso y persistencia hibernate	Análisis xml

Una implementación típica para jBPM incluye el almacenamiento persistente de definiciones de proceso y las ejecuciones de proceso. En ese caso, jBPM no tiene ninguna dependencia fuera de hibernate y sus bibliotecas dependientes.

Por supuesto, las bibliotecas necesarias para hibernar dependen del ambiente y de qué características utilizar. Para obtener detalles consulte en la documentación hibernate. La tabla siguiente entrega una indicación para un ambiente de desarrollo POJO independiente.

jBPM se distribuye con hibernate 3.1 final. Pero también puede funcionar con 3.0.x. En ese caso quizá tenga que actualizar algunas consultas de hibernación en el archivo de

configuración hibernate.queries.hbm.xml. Para obtener más información sobre la personalización de consultas, consulte la sección 7.6, "Personalización de Consultas".

Tabla 5.2.

Biblioteca	Uso	Descripción
hibernate3.jar	Persistencia de hibernación	El mejor asignador O/R
antlr-2.7.5H3.jar	Utilizado en análisis mediante persistencia de hibernación	Biblioteca del analizador
cglib-2.1_2jboss.jar	Persistencia de hibernación	Biblioteca utilizada para proxies de hibernació
commons-collections.jar	Persistencia de hibernación	
ehcache-1.1.jar	Persistencia de hibernación (en la configuración predeterminada)	Implementación de cache de segundo nivel proveedor de caché diferente para hiberna necesita esta biblioteca
jaxen-1.1-beta-4.jar	Definiciones de proceso y persistencia de hibernación	Biblioteca XPath (utilizada por dom4j)
jdbc2_0-stdext.jar	Persistencia de hibernación	
asm.jar	Persistencia de hibernación	Biblioteca de código asm byte
asm-attrs.jar	Persistencia de hibernación	Biblioteca de código asm byte

La biblioteca beanshell es opcional. Si no se incluye no es posible utilizar la integración beanshell en el lenguaje de proceso jbpm y se recibe un mensaje de registro, el cual indica que jbpm no pudo cargar la clase Script y, en consecuencia, el elemento de secuencia de comando no estará disponible.

Tabla 5.3.

Biblioteca	Uso	Descripción
bsh-1.3.0.jar	Intérprete de script beanshell	Sólo se utiliza en secuencias de comandos y decisiones. Cuando de proceso, la biblioteca beanshell se puede eliminar, pero lue asignación Script.hbm.xml en `hibernate.cfg.xml`

Capítulo 6. Configuración

La forma más sencilla de configurar jBPM es colocando el archivo de configuración `jbpm.cfg.xml` en la raíz de ruta de clase. Si dicho archivo no se encuentra como recurso, se debe utilizar la configuración mínima predeterminada, la cual se incluye en la biblioteca jbpm. Cabe mencionar que la configuración mínima no tiene ninguna configuración para persistencia.

La configuración jBPM se representa con la clase java `org.jbpm.JbpmConfiguration`. La forma más sencilla para obtener la JbpmConfiguration es hacer uso del método único de instancia `JbpmContext.getInstance()`.

Si se desea cargar una configuración desde otra fuente es posible utilizar los métodos `JbpmConfiguration.parseXxxx`.

```
static JbpmConfinguration jbpmConfiguration = JbpmConfinguration.getInstance();
```

JbpmConfiguration es threadsafe y en consecuencia se puede mantener en un miembro estático. En todos los subprocesos se puede utilizar la JbpmConfiguration como una fábrica de objetos JbpmContext. Un JbpmContext por lo general representa una transacción. El JbpmContext deja servicios disponibles dentro de un bloque de contexto. Un bloque de contexto se ve de la siguiente forma:

```
JbpmContext jbpmContext = jbpmConfiguration.createJbpmContext();
try {
  // Esto es lo que denominamos un bloque de contexto.
  // Aquí se pueden realizar operaciones de flujo de trabajo

} finally {
  jbpmContext.close();
}
```

`JbpmContext` deja disponibles un conjunto de servicios y la configuración para jBPM. Estos servicios están configurados en el archivo de configuración `jbpm.cfg.xml` y permite que jBPM se ejecute en cualquier ambiente Java y utilice los servicios que haya disponible en ese ambiente.

Aquí se muestra una configuración típica para the JbpmContext tal como se puede encontrar en `src/config.files/jbpm.cfg.xml`:

```
<jbpm-configuration>

  <jbpm-context>
    <service name='persistence'
factory='org.jbpm.persistence.db.DbPersistenceServiceFactory' />
    <service name='message' factory='org.jbpm.msg.db.DbMessageServiceFactory' />
    <service name='scheduler'
factory='org.jbpm.scheduler.db.DbSchedulerServiceFactory' />
    <service name='logging'
factory='org.jbpm.logging.db.DbLoggingServiceFactory' />
    <service name='authentication'
factory='org.jbpm.security.authentication.DefaultAuthenticationServiceFactory'
```

```
/>
  </jbpm-context>

  <!-- configuration resource files pointing to default configuration files in
jbpm-{version}.jar -->
  <string name='resource.hibernate.cfg.xml' value='hibernate.cfg.xml' />
  <!-- <string name='resource.hibernate.properties' value='hibernate.properties'
/> -->
  <string name='resource.business.calendar'
value='org/jbpm/calendar/jbpm.business.calendar.properties' />
  <string name='resource.default.modules'
value='org/jbpm/graph/def/jbpm.default.modules.properties' />
  <string name='resource.converter'
value='org/jbpm/db/hibernate/jbpm.converter.properties' />
  <string name='resource.action.types'
value='org/jbpm/graph/action/action.types.xml' />
  <string name='resource.node.types' value='org/jbpm/graph/node/node.types.xml'
/>
  <string name='resource.parsers' value='org/jbpm/jpdl/par/jbpm.parsers.xml' />
  <string name='resource.varmapping'
value='org/jbpm/context/exe/jbpm.varmapping.xml' />

  <bean name='jbpm.task.instance.factory'
class='org.jbpm.taskmgmt.impl.DefaultTaskInstanceFactoryImpl' singleton='true'
/>
  <bean name='jbpm.variable.resolver'
class='org.jbpm.jpdl.el.impl.JbpmVariableResolver' singleton='true' />
  <long name='jbpm.msg.wait.timout' value='5000' singleton='true' />

</jbpm-configuration>
```

En este archivo de configuración se pueden ver tres partes:

- El primer análisis configura el contexto jbpm con un conjunto de implementaciones de servicio. Las opciones de configuración posibles están cubiertas en los capítulos que analizan las implementaciones específicas de servicio.
- La segunda parte corresponde a asignaciones de referencias a recursos de configuración. Estas referencias de recursos se pueden actualizar si se desea personalizar uno de estos archivos de configuración. Por lo general, se hace una copia de la configuración predeterminada, la cual está en `jbpm-3.x.jar` y se pone en algún lugar de classpath. Luego se debe actualizar la referencia en este archívo y jbpm utiliza su versión personalizada de un archivo de configuración.
- La tercera parte son algunas configuraciones miscelaneas utilizadas en jbpm. Estas opciones de configuración se describen en los capítulos que analizan los temas específicos.

El conjunto de servicios configurados en forma predeterminada está enfocado en un simple ambiente webapp y dependencias mínimas. El servicio de persistencia obtiene una conexión jdbc y todos los demás servicios van a utilizar la misma conexión para llevar a cabo sus servicios. De modo que todas sus operaciones de flujo de trabajo están centralizadas en 1 en la conexión A JDBC sin necesidad de un administrador de transacciones.

`JbpmContext` contiene métodos de conveniencia para la mayoría de las operaciones de procesos comunes:

```
  public void deployProcessDefinition(ProcessDefinition processDefinition) {...}
  public List getTaskList() {...}
  public List getTaskList(String actorId) {...}
  public List getGroupTaskList(List actorIds) {...}
  public TaskInstance loadTaskInstance(long taskInstanceId) {...}
  public TaskInstance loadTaskInstanceForUpdate(long taskInstanceId) {...}
  public Token loadToken(long tokenId) {...}
  public Token loadTokenForUpdate(long tokenId) {...}
  public ProcessInstance loadProcessInstance(long processInstanceId) {...}
  public ProcessInstance loadProcessInstanceForUpdate(long processInstanceId)
{...}
  public ProcessInstance newProcessInstance(String processDefinitionName) {...}
  public void save(ProcessInstance processInstance) {...}
  public void save(Token token) {...}
  public void save(TaskInstance taskInstance) {...}
  public void setRollbackOnly() {...}
```

Cabe mencionar que los métodos `XxxForUpdate` registran el objeto cargado para auto-guardado de modo que no sea necesario llamar uno de los métodos de guardado en forma explícita.

Es posible especificar varios `jbpm-context`s, pero luego es necesario asegurarse de que cada `jbpm-context` reciba un atributo `name` único. Los contextos con nombre se pueden recuperar con `JbpmConfiguration.createContext(String name);`

Un elemento de `service` especifica el nombre de un servicio y la fábrica de servicios para dicho servicio. El servicio sólo se crea en caso que se solicite con `JbpmContext.getServices().getService(String name).`

Las fabricas también se pueden especificar como un elemento en lugar de un attributo. Eso podría ser necesario para inyectar alguna información de configuración en los objetos. El componente responsable del análisis de XML, crear y conectar los objetos se denomina fábrica de objetos.

6.1. Archivos de configuración

A continuación se encuentra una breve descripción de todos los archivos de configuración que se pueden personalizar en jBPM.

6.1.1. Archivo Hibernate cfg xml

Este archivo contiene configuraciones de hibernación y referencias hacia los archivos de recursos de asignación de hibernación.

Ubicación: `hibernate.cfg.xml` a menos que se especifique algo distinto en la propiedades jbpm.hibernate.cfg.xml en el archivo jbpm.properties. En el proyecto jbpm el archivo predeterminado hibernate cfg xml está ubicado en el directorio `src/config.files/hibernate.cfg.xml`

6.1.2. Archivo de configuración de consultas de hibernación

Este archivo contiene consultas de hibernación que se utilizan en las sesiones jBPM `org.jbpm.db.*Session`.

Ubicación: `org/jbpm/db/hibernate.queries.hbm.xml`

6.1.3. Archivo de configuración de tipos de nodo

Este archivo contiene la asignación de elementos de nodo XML a clases de implementación de nodo.

Ubicación: `org/jbpm/graph/node/node.types.xml`

6.1.4. Archivo de configuración de tipos de acción

Este archivo contiene la asignación de elementos de acción XML a clases de implementación de Acción.

Ubicación: `org/jbpm/graph/action/action.types.xml`

6.1.5. Archivo de configuración de calendario de negocios

Contiene la definición de horas hábiles y tiempo libre.

Ubicación: `org/jbpm/calendar/jbpm.business.calendar.properties`

6.1.6. Archivo de configuración de asignación de variables

Especifica de qué forma se convierten los valores de las variables de proceso (objetos java) en instancias de variables para almacenarlas en la base de datos jbpm.

Ubicación: `org/jbpm/context/exe/jbpm.varmapping.xml`

6.1.7. Archivo de configuración del convertidor

Especifica las asignaciones id-to-classname. Las id's están almacenadas en la base de datos. El org.jbpm.db.hibernate.ConverterEnumType se utiliza para asignar las ids a objetos únicos.

Ubicación: `org/jbpm/db/hibernate/jbpm.converter.properties`

6.1.8. Archivo de configuración de módulos predeterminados

Especifica cuáles módulos se agregan a una nueva ProcessDefinition en forma predeterminada.

Ubicación: `org/jbpm/graph/def/jbpm.default.modules.properties`

6.1.9. Archivo de configuración de analizadores de archivo de proceso

Ubicación: `org/jbpm/jpdl/par/jbpm.parsers.xml`

6.2. Fábrica de objetos

La fábrica de objetos puede crear objetos de acuerdo con un archivo de configuración tipo beans. El archivo de configuración especifica la forma como se deben crear, configurar y conectar los objetos para formar un gráfico de objetos completo. La fábrica de objetos puede inyectar las configuraciones y otros beans en un solo bean.

En su forma más simple, la fábrica de objetos puede crear tipos básicos y beans java a partir de dicha configuración:

```
<beans>
  <bean name="task" class="org.jbpm.taskmgmt.exe.TaskInstance"/>
  <string name="greeting">hello world</string>
  <int name="answer">42</int>
  <boolean name="javaisold">true</boolean>
  <float name="percentage">10.2</float>
  <double name="salary">100000000.32</double>
  <char name="java">j</char>
  <null name="dusttodust" />
</beans>

-------------------------------------------------------

ObjectFactory of = ObjectFactory.parseXmlFromAbove();
assertEquals(TaskInstance.class, of.getNewObject("task").getClass());
assertEquals("hello world", of.getNewObject("greeting"));
assertEquals(new Integer(42), of.getNewObject("answer"));
assertEquals(Boolean.TRUE, of.getNewObject("javaisold"));
assertEquals(new Float(10.2), of.getNewObject("percentage"));
assertEquals(new Double(100000000.32), of.getNewObject("salary"));
assertEquals(new Character('j'), of.getNewObject("java"));
assertNull(of.getNewObject("dusttodust"));
```

También se pueden configurar listas:

```
<beans>
  <list name="numbers">
    <string>one</string>
    <string>two</string>
    <string>three</string>
  </list>
</beans>
```

y mapas

```
<beans>
  <map name="numbers">
    <entry><key><int>1</int></key><value><string>one</string></value></entry>
    <entry><key><int>2</int></key><value><string>two</string></value></entry>
    <entry><key><int>3</int></key><value><string>three</string></value></entry>
  </map>
</beans>
```

Los beans se pueden configurar con inyección directa en campo y establecedores de propiedades.

```
<beans>
  <bean name="task" class="org.jbpm.taskmgmt.exe.TaskInstance" >
    <field name="name"><string>do dishes</string></field>
    <property name="actorId"><string>theotherguy</string></property>
  </bean>
</beans>
```

Los beans se pueden referenciar. El objeto referenciado no tiene que ser un bean, puede ser una secuencia, un entero o cualquier otro objeto.

```
<beans>
  <bean name="a" class="org.jbpm.A" />
  <ref name="b" bean="a" />
</beans>
```

Los beans se pueden construir con cualquier constructor

```
<beans>
  <bean name="task" class="org.jbpm.taskmgmt.exe.TaskInstance" >
    <constructor>
      <parameter class="java.lang.String">
        <string>do dishes</string>
      </parameter>
      <parameter class="java.lang.String">
        <string>theotherguy</string>
      </parameter>
    </constructor>
  </bean>
</beans>
```

... o con un método de fábrica en un bean ...

```
<beans>
  <bean name="taskFactory"
         class="org.jbpm.UnexistingTaskInstanceFactory"
         singleton="true"/>

  <bean name="task" class="org.jbpm.taskmgmt.exe.TaskInstance" >
    <constructor factory="taskFactory" method="createTask" >
      <parameter class="java.lang.String">
        <string>do dishes</string>
      </parameter>
      <parameter class="java.lang.String">
        <string>theotherguy</string>
      </parameter>
    </constructor>
  </bean>
</beans>
```

... o con un método estático de fábrica en una clase ...

```
<beans>
  <bean name="task" class="org.jbpm.taskmgmt.exe.TaskInstance" >
    <constructor factory-class="org.jbpm.UnexistingTaskInstanceFactory"
method="createTask" >
```

```
      <parameter class="java.lang.String">
        <string>do dishes</string>
      </parameter>
      <parameter class="java.lang.String">
        <string>theotherguy</string>
      </parameter>
    </constructor>
  </bean>
</beans>
```

Cada objeto nombrado se puede marcar como único con el atributo `singleton="true"`. Eso significa que una fábrica de objetos siempre devuelve el mismo objeto para cada solicitud. Cabe mencionar que los objetos singleton no se comparten entre diferentes fábricas de objetos.

Esta característica provoca la diferenciación entre los métodos `getObject` y `getNewObject`. Los usuarios típicos de la fábrica de objetos utilizan el `getNewObject`. Esto significa que, en primer lugar, la caché de objetos de la fábrica de objetos se debe vaciar antes de construir el nuevo gráfico de objeto. Durante la construcción del gráfico de objeto, los objetos no-singleton se almacenan en el caché de objetos de la fábrica para permitir referencias compartidas a un objeto. El caché de objetos singleton es diferente al caché sencillo de objetos. El caché de singleton nunca se vacía, mientras que el caché simple de objetos se vacía al inicio de cada método `getNewObject`.

Una extensión hacia el esquema tradicional de beans es context-builder tag. Un elemento context-builder contiene una lista de objetos ContextInterceptor. La propiedad singleton es importante aquí: los ContextInterceptors se pueden marcar como singleton, mientras que los interceptores de contexto que mantienen el estado en las variables de miembro se deben crear para cada solicitud. Los ContextBuilders no se deben declarar como singleton dado que tienen que mantener el estado (el índice del interceptor de contexto en ejecución en caso que se produzca una excepción).

Capítulo 7. Persistencia

IEn la mayoría de las situaciones, jBPM se utiliza para actualizar la ejecución de procesos que involucran largo tiempo. En este contexto, "un largo tiempo" significa que incluye varias transacciones. El objetivo principal de la persistencia es almacenar ejecuciones de proceso durante los estados de espera. Por este motivo, es conveniente pensar en ejecuciones de proceso como máquinas de estado. En una transacción deseamos mover la máquina de estado de ejecución de proceso de un estado al siguiente.

Existen tres (3) formas diferentes de representar una definición de proceso: como xml, como objetos de java y como registros en la base de datos jBPM. La información de ejecución (=tiempo de ejecución) y la información de registro se puede representar de dos (2) formas: como objetos de java y como registros en la base de datos jBPM.

Figure 7.1. Las transformaciones y las diferentes formas

Para obtener más información sobre la representación xml de las definiciones y archivos de proceso, consulte capítulo 16, Lenguaje de Definición de Proceso jBPM (JPDL).

se puede encontrar mayor información sobre la forma de implementar un archivo de proceso en la base de datos sección 16.1.1, "Implementación de un archivo de proceso"

7.1. La API de Persistencia

7.1.1.Relación con el marco de configuración

La API de persistencia es un integrado con el marco de configuración configuration framework que expone ciertos métodos de persistencia convenientes en JbpmContext. Por lo tanto, las operaciones de API de Persistencia se pueden llamar en un bloque de contexto jBPM como el siguiente:

```
JbpmContext jbpmContext = jbpmConfiguration.createJbpmContext();
try {

  // Invoque las operaciones de persistencia aquí

} finally {
  jbpmContext.close();
}
```

En lo siguiente, se da por supuesto que la configuración incluye un servicio de persistencia similar a este (como en el archivo de configuración de ejemplo `src/config.files/jbpm.cfg.xml`):

```
<jbpm-configuration>

  <jbpm-context>
    <service name='persistence'
factory='org.jbpm.persistence.db.DbPersistenceServiceFactory' />
    ...
  </jbpm-context>
```

```
  ...
</jbpm-configuration>
```

7.1.2. Métodos de conveniencia en JbpmContext

Las tres operaciones más comunes de persistencia son:

- Implementación de unproceso
- Inicio de una nueva ejecución de un proceso
- Continuación de una ejecución

Primero la implementación de una definición de proceso. Normalmente, esto se hace directamente desde el diseñador de proceso gráfico o desde la tarea deployprocess ant. Pero acá se puede ver la manera de hacerlo de forma programática:

```
JbpmContext jbpmContext = jbpmConfiguration.createJbpmContext();
try {
  ProcessDefinition processDefinition = ...;
  jbpmContext.deployProcessDefinition(processDefinition);
} finally {
  jbpmContext.close();
}
```

ara la creación de una ejecución de proceso necesitamos especificar la definición de proceso para la que esta ejecución será una instancia. La manera más común de especificar esto es consultando el nombre del proceso y dejando que jBPM busque la versión más reciente del proceso en la base de datos:

```
JbpmContext jbpmContext = jbpmConfiguration.createJbpmContext();
try {
  String processName = ...;
  ProcessInstance processInstance =
      jbpmContext.newProcessInstance(processName);
} finally {
  jbpmContext.close();
}
```

Para una ejecución de proceso continua, necesitamos buscar la instancia del proceso, el testigo o la taskInstance en la base de datos, llamar algunos métodos en los objetos POJO jBPM y, posteriormente, volver a guardar las actualizaciones hechas a processInstance en la base de datos.

```
JbpmContext jbpmContext = jbpmConfiguration.createJbpmContext();
try {
  long processInstanceId = ...;
  ProcessInstance processInstance =
      jbpmContext.loadProcessInstance(processInstanceId);
  processInstance.signal();
  jbpmContext.save(processInstance);
} finally {
  jbpmContext.close();
}
```

Recuerde que si usa los métodos xxx**ForUpdate** en JbpmContext, ya no es necesario llamar de manera explícita a jbpmContext.save porque se produce automáticamente durante el cierre de jbpmContext. Por ejemplo, supongamos que deseamos informar a jBPM sobre una taskInstance que se ha finalizado. Observe que el término de instancia de tarea puede generar que la ejecución continúe por lo que el processInstance relacionado con taskInstance se debe guardar. La manera más conveniente de hacer esto es usar el método loadTaskInstanceForUpdate:

```
JbpmContext jbpmContext = jbpmConfiguration.createJbpmContext();
try {
  long taskInstanceId = ...;
  TaskInstance taskInstance =
      jbpmContext.loadTaskInstanceForUpdate(taskInstanceId);
  taskInstance.end();
} finally {
  jbpmContext.close();
}
```

Como información adicional, en la próxima parte se explica la forma cómo jBPM administra la persistencia y usa la hibernación.

La `JbpmConfiguration` actualiza un conjunto de `ServiceFactory`s. Las fábricas de servicio están configuradas en `jbpm.cfg.xml` como se mostró anteriormente y no funcionan para instancias. El `DbPersistenceServiceFactory` solamente se instancia la primera vez que se necesita. Posteriormente, las fábricas de servicio (service factories) se actualizan en `JbpmConfiguration`. Un `DbPersistenceServiceFactory` gestiona un `SessionFactory` de hibernación. Pero de igual manera se crea una fábrica de servicio de hibernación no funcional cuando se solicita por primera vez.

Figura 7.2.Las clases relacionadas con la persistencia

Durante la invocación de `jbpmConfiguration.createJbpmContext()`, sólo se crea `JbpmContext`. En ese momento no se realizan más inicializaciones relacionadas con la persistencia. La secuencia de comando `JbpmContext` gestiona un `DbPersistenceService`, que se instancia con la primera solicitud. El `DbPersistenceService` gestiona la sesión de hibernación. Asimismo, la sesión de hibernación en `DbPersistenceService` también se crea no funcional. Como resultado, únicamente se abrirá una sesión de hibernación cuando se invoque la primera operación que requiera de persistencia y no antes.

7.1.3. Uso avanzado de API

El DbPersistenceService actualiza una sesión de hibernación inicializada no funcional. Todo acceso a la base de datos se realiza mediante esta sesión de hibernación. Todas las consultas y actualizaciones hechas por jBPM son expuestas por las clases XxxSession como por ejemplo, GraphSession, SchedulerSession, LoggingSession,... Estas clases de sesiones hacen referencia a las consultas de hibernación y todas usan la misma sesión de hibernación que se muestra a continuación.

Las clases XxxxSession también son accesibles mediante el JbpmContext.

7.2. Configuración del servicio de persistencia

7.2.1. La fábrica de sesiones de hibernación

De manera predeterminada, el DbPersistenceServiceFactory utilizará el recurso hibernate.cfg.xml en la raíz de la ruta de clase para crear una fábrica de sesión de hibernación. Recuerde que el recurso de archivo de configuración de hibernación se asocia en la propiedad 'jbpm.hibernate.cfg.xml' y que se puede adaptar en el jbpm.cfg.xml. La siguiente es la configuración predeterminada:

```
<jbpm-configuration>
  ...
  <!-- configuration resource files pointing to default configuration files in
jbpm-{version}.jar -->
  <string name='resource.hibernate.cfg.xml'
          value='hibernate.cfg.xml' />
  <!-- <string name='resource.hibernate.properties'
                  value='hibernate.properties' /> -->
  ...
</jbpm-configuration>
```

Cuando se especifica el **resource.hibernate.properties** de propiedad, las propiedades en dicho archivo de recurso **sobrescriben todas** las propiedades en el hibernate.cfg.xml. En vez de actuailzar el hibernate.cfg.xml para apuntar a su base de datos, hibernate.properties se puede usar para manejar actualizaciones jbpm de manera conveniente: es posible copiar el hibernate.cfg.xml sin necesidad de volver a aplicar los cambios.

7.2.2. DbPersistenceServiceFactory

El DbPersistenceServiceFactory en sí tiene tres propiedades de configuración adicionales: isTransactionEnabled, sessionFactoryJndiName y dataSourceJndiName. Para especificar cualquiera de estas propiedades en el jbpm.cfg.xml, es necesario especificar la fábrica de servicio como un bean en el elemento de fábrica como el siguiente:

```
  <jbpm-context>
    <service name="persistence">
      <factory>
        <bean factory="org.jbpm.persistence.db.DbPersistenceServiceFactory">
          <field name="isTransactionEnabled"><false /></field>
          <field name="sessionFactoryJndiName">
            <string value="java:/myHibSessFactJndiName" />
          </field>
          <field name="dataSourceJndiName">
            <string value="java:/myDataSourceJndiName" />
          </field>
        </bean>
      </factory>
    </service>
    ...
  </jbpm-context>
```

- **isTransactionEnabled**: de manera predeterminada, jBPM comenzará y terminará las transacciones de hibernación. Para desactivar las transacciones y prohibir que jBPM administre las transacciones con hibernación, configure la propiedad isTransactionEnabled em falso como en el ejemplo anterior. Consulte la sección 7.3, "Transacciones de hibernación" para obtener mayor información sobre las transacciones.
- **sessionFactoryJndiName**: es nula, de manera predeterminada, es decir que la fábrica de sesiones no se busca en JNDI. Si está configurada y se necesita una fábrica de sesiones para crear una sesión de hibernación, se busca la fábrica de sesiones en jndi utilizando el nombre JNDI proporcionado.
- **dataSourceJndiName**: es nula de manera predeterminada y la creación de conexiones JDBC se delega para hibernación. Al especificar una fuente de datos, jBPM busca una conexión JDBC en la fuente de datos y la proporciona a hibernación mientras abre una nueva sesión, consulte la sección 7.5, "Elementos ingresados por el usuario".

7.3. Transacciones de hibernación

De manera predeterminada, jBPM delega la transacción a hibernación y usa la sesión según el patrón de transacción. jBPM inicia una transacción de hibernación cuando se abre una sesión de hibernación. Esto sucede la primera vez que se invoca una operación persistente en el jbpmContext. La transacción se efectúa justo antes de que se cierre la sesión de hibernación. Esto sucede en jbpmContext.close().

Use `jbpmContext.setRollbackOnly()` para marcar una transacción para realizar la anulación. En dicho caso, la transacción se anula y se devuelve al momento justo antes de que se cerrara la sesión en el jbpmContext.close().

Para prohibir que jBPM invoque cualquier tipo de método de transacción en la API de hibernación, configure la propiedad isTransactionEnabled en falso como se explicó en la sección 7.2.2, "DbPersistenceServiceFactory".

7.4. Transacciones administradas

La situación más común para las transacciones administradas es cuando se usa jBPM en un servidor de aplicaciones JEE como JBoss. A continuación se presenta la situación más común:

- Configurar una fuente de datos en el servidor de aplicaciones
- Configurar hibernación para usar dicha fuente de datos para sus conexiones
- Utilizar transacciones administradas por container
- Desactivar transacciones en jBPM

7.5. Elementos ingresados por el usuario

También es posible proporcionar conexiones JDBC, sesiones de hibernación y fábricas de sesiones de hibernación de manera programática a jBPM.

Cuando se proporciona dicho recurso a jBPM, éste usa los recursos proporcionados y no

los configurados.

La clase `JbpmContext` contiene algunos métodos de conveniencia para introducir recursos de manera programática. Por ejemplo, para proporcionar una conexión JDBC a jBPM, use el siguiente código:

```
JbpmContext jbpmContext = jbpmConfiguration.createJbpmContext();
try {
  Connection connection = ...;
  jbpmContext.setConnection(connection);

  // Invoque una o más operaciones de persistencia

} finally {
  jbpmContext.close();
}
```

La clase JbpmContext tiene los siguientes métodos de conveniencia para proporcionar recursos de manera programática:

```
JbpmContext.setConnection(Connection);
JbpmContext.setSession(Session);
JbpmContext.setSessionFactory(SessionFactory);
```

7.6. Personalización de consultas

Todas las consultas HQL que jBPM utiliza están centralizadas en un archivo de configuración. El archivo de configuración hibernate.cfg.xml hace referencia a dicho archivo de recursos como se muestra a continuación:

```
<hibernate-configuration>
  ...
    <!-- hql queries and type defs -->
    <mapping resource="org/jbpm/db/hibernate.queries.hbm.xml" />
  ...
</hibernate-configuration>
```

Para personalizar una o más de estas consultas, haga una copia del archivo original y ponga su versión adaptada en cualquier parte de la ruta de clase. A continuación actualice el 'org/jbpm/db/hibernate.queries.hbm.xml' de referencia en el hibernate.cfg.xml para apuntar a su versión adaptada.

7.7. Compatibilidad de base de datos

jBPM se ejecuta en cualquier base de datos que admita hibernación.

Los archivos de configuración de ejemplo en jBPM (`src/config.files`) especifican el uso de la base de datos hipersónica en memoria. Esa base de datos resulta ideal durante el desarrollo y en las pruebas de funcionamiento. La base de datos hipersónica en memoria conserva todos sus datos en memoria y no los almacena en discos.

7.7.1. Chambio de la base de datos jBPM

A continuación se muestra una lista indicativa de pasos a seguir cuando se cambia jBPM para que use una base de datos diferente:

- Ponga el archivo de biblioteca de controlador jdbc en la ruta de clase
- Actualice la configuración de hibernación usada por jBPM
- Cree el diagrama en la nueva base de datos

7.7.2. El diagrama de base de datos jBPM

El subproyecto jbpm.db contiene un número de controladores, instrucciones y secuencias de comando que facilitan el uso de la base de datos de su preferencia. Para obtener mayor información, consulte el archivo readme.html en la raíz del pryecto jbpm.db.

Si bien jBPM puede generar secuencias de comando DDL para todas las bases de datos, estos diagramas no siempre están optimizados. Por esta razón es conveniente hacer que su DBA revise el DDL que se genera para optimizar los tipos de columna y el uso de índices.

Durante el desarrollo, es posible que le interese la siguiente configuración de hibernación: Si configura la propiedad de configuración de hibernación 'hibernate.hbm2ddl.auto' para 'create-drop' (es decir, en el hibernate.cfg.xml), el diagrama se crea automáticamente en la base de datos la primera vez que se usa en una aplicación. Cuando se cierra la aplicación, se deja el diagrama.

También es posible invocar la generación de diagrama por programa utilizando `jbpmConfiguration.createSchema()` y `jbpmConfiguration.dropSchema()`.

7.8. Combinación de clases de hibernación

En su proyecto, puede usar la hibernación para su persistencia. La combinación de sus clases persistentes con las clases persistentes de jBPM es opcional. Combinar su persistencia de hibernación con la persistencia de hibernación de jBPM presenta dos grandes beneficios:

Primero: iniciar la sesión, la conexión y la administración de transacciones se hace más fácil. Al combinar jBPM y su persistencia en una fábrica de sesiones de hibernación, existe una sesión de hibernación, una conexión jdbc que maneja tanto su persistencia como la de jBPM. De esta manera, las actualizaciones jBPM se encuentran automáticamente en las mismas transacciones que las actualizaciones de su propio modelo de dominio. Esto puede eliminar la necesidad de uso de un administrador de transacciones.

En segundo lugar, esto permite colocar su objeto persistente de hibernación en las variables de proceso sin mayores complicaciones.
La manera más fácil de integrar sus clases persistentes con las clases persistentes de jBPM es mediante la creación de un hibernate.cfg.xml central. Puede tomar el `src/config.files/hibernate.cfg.xml` de jBPM como punto de partida y agregar referencias a sus archivos de asociación de hibernación ahí.

7.9. Adaptación de archivos de asociación de hibernación jBPM

Para adaptar cualquiera de los archivos de asociación de hibernación jBPM, puede efectuar los siguientes pasos:

- Copie el archivo(s) de asociación de hibernación jBPM que desea copiar de las fuentes (`src/java.jbpm/...`) o desde jbpm jar.
- Ponga la copia en cualquier parte de la ruta de clase.
- Actualice las referencias a los archivos de asociación adaptados en el archivo de configuración hibernate.cfg.xml

7.10. Caché de segundo nivel

BPM usa el caché de segundo nivel de la hibernación para conservar definiciones de proceso en memoria una vez que los ha cargado. Las colecciones y las clases de definición de proceso se configuran en los archivos de asociación de hibernación jBPM con el elemento de caché como se muestra:

```
<cache usage="nonstrict-read-write"/>
```

Ya que las definiciones de proceso nunca (deben) cambian, es aceptable tenerlas en la caché de segundo nivel. Consulte también la sección 16.1.3, "Cambio de definiciones de proceso implementadas".

La caché de segundo nivel es un aspecto importante en la implementación JBoss jBPM. Si no fuera por esta caché, JBoss jBPM presentaría una desventaja muy importante en comparación con otras técnicas para implementar un motor BPM.

La estrategia de almacenamiento en caché se configura en `nonstrict-read-write`. En el momento de la ejecución, la estrategia de almacenamiento en caché se puede configurar en `read-only`. Pero en dicho caso sería necesario un conjunto separado de archivos de asociación de hibernación para implementar un proceso. Por este motivo hemos preferido la capacidad de escritura-lectura-no estricta.

Capítulo 8. La base de datos jBPM

8.1. Cambio de Backend de la base de datos

El cambio del backend de la base de datos JBoss jBPM es razonablemente sencillo. Veremos este proceso paso a paso usando PostgreSQL como ejemplo. El proceso resulta idéntico para todas las demás bases de datos admitidas. Para varias de estas bases de datos compatibles, existen varios controladores JDBC, archivos de configuración de hibernación y archivos de versión Ant para la generación de secuencia de comandos de creación de base de datos en la distribución jBPM, en el subproyecto DB. Si no puede encontrar estos archivos para la base de datos que desea usar, debe primero asegurarse de que Hibernación sea compatible con su base de datos. Si este fuera el caso, puede revisar los archivos de una de las bases de datos que se encuentran en la base de datos de proyecto y replicarlo usando su propia base de datos.

Para este documento usaremos la distribución del Kit de inicio de JBoss jBPM. Supondremos que este kit de inicio se extrajo a una ubicación en su máquina llamada ${JBPM_SDK_HOME}. Encontrará el subproyecto de base de datos de jBPM en el ${JBPM_SDK_HOME}/jbpm-db.

Una vez instalada la base de datos es necesario que ejecute la secuencia de comandos de creación de la base de datos. Estos crean las tablas jBPM en la base de datos. Para hacer la webapp predeterminada que se ejecuta con esta nueva base de datos es necesario actualizar algunos archivos de configuración del servidor incluidos en el Kit de inicio. No entraremos en mucho detalle para estos cambios de configuración. Si desea obtener mayor información sobre los diferentes parámetros de configuración en el servidor, recomendamos que consulte la documentación JBoss.

8.1.1. Instalación del Administrador de base de datos PostgreSQL

El cambio del backend de la base de datos JBoss jBPM es razonablemente sencillo. Veremos este proceso paso a paso usando PostgreSQL como ejemplo. El proceso resulta idéntico para todas las demás bases de datos admitidas. Para varias de estas bases de datos compatibles, existen varios controladores JDBC, archivos de configuración de hibernación y archivos de versión Ant para la generación de secuencia de comandos de creación de base de datos en la distribución jBPM, en el subproyecto DB. Si no puede encontrar estos archivos para la base de datos que desea usar, debe primero asegurarse de que Hibernación sea compatible con su base de datos. Si este fuera el caso, puede revisar los archivos de una de las bases de datos que se encuentran en la base de datos de proyecto y replicarlo usando su propia base de datos.

Para este documento usaremos la distribución del Kit de inicio de JBoss jBPM. Supondremos que este kit de inicio se extrajo a una ubicación en su máquina llamada ${JBPM_SDK_HOME}. Encontrará el subproyecto de base de datos de jBPM en el ${JBPM_SDK_HOME}/jbpm-db.

Una vez instalada la base de datos es necesario que ejecute la secuencia de comandos de creación de la base de datos. Estos crean las tablas jBPM en la base de datos. Para hacer la webapp predeterminada que se ejecuta con esta nueva base de datos es necesario actualizar algunos archivos de configuración del servidor incluidos en el Kit de inicio. No entraremos en mucho detalle para estos cambios de configuración. Si

desea obtener mayor información sobre los diferentes parámetros de configuración en el servidor, recomendamos que consulte la documentación JBoss.

.

Figura 8.1. Herramienta PostgreSQL pgAdmin III después de la creación de la base de datos JbpmDB

Después de la instalación de la base de datos, podemos utilizar una herramienta de visualización de base de datos como DBVisualizer par ver el contenido de la base de datos. Antes de definir una conexión de base de datos con el DBVisualizer, es posible que necesite agregar el controlador PostgreSQL JDBC al administrador de controladores. Seleccione 'Tools->Driver Manager...' (Herramientas -> Administrador de controladores) para abrir la ventana del administrador de controladores. En la figura siguiente verá un ejemplo de cómo agregar el controlador PostgreSQL JDBC driver.

Figura 8.2. Adición del controlador JDBC al administrador de controladores

Ahora todo está configurado para definir una conexión de base de datos en DBVisualizer a la base de datos recién creada. Utilizaremos esta herramienta más adelante en este documento para asegurarnos de que la secuencia de comandos de creación y la implementación de proceso funcionen según lo previsto. Consultaremos la siguiente figura como ejemplo en la creación de la conexión en DBVisualizer. Como se puede ver, la base de datos aún no tiene tablas. Las crearemos en la siguiente sección.

Figura 8.3. Creación de la conexión a la base de datos jBPM

Otro punto que vale la pena mencionar es la dirección URL de la base de datos anterior: 'jdbc:postgresql://localhost:5432/JbpmDB'. Si creó la base de datos JbpmDB con otro nombre o si PostgreSQL no se está ejecutando en la máquina localhost o en otro puerto, tendrá que adaptar la dirección URL de su base de datos de acuerdo con esto.

8.1.2. Creación de la base de datos JBoss jBPM

Tal como ya se ha mencionado, encontrará las secuencias de comandos de la base de datos para una gran cantidad de bases de datos admitidas en el subproyecto de base de datos. Las secuencias de comando de la base de datos de PostgreSQL se encuentran en la carpeta '${JBPM_SDK_HOME}/jbpm-db/build/postgresql/scripts. La secuencia de comando de creación se llama 'postgresql.create.sql'. Usando el DBVisualizer, puede cargar esta secuencia de comando cambiando a la ficha 'SQL Commander' y seleccionando 'File->Load...' (Archivo, Cargar). En el siguiente cuadro de diálogo, navegue al archivo de la secuencia de comando de creación. La siguiente figura muestra el resultado de esta acción.

Figura 8.4. Carga de la secuencia de comandos de creación de base de datos

Para ejecutar esta secuencia de comando con DBVisualizer, seleccione 'Database-

>Execute' (Base de datos -> Ejecutar). Todas las tablas JBoss jBPM se crean después de este paso. En la siguiente figura se muestra la situación.

Figura 8.5. Ejecución de la secuencia de comandos de creación de base de datos

Después de estos pasos ya no quedan datos en las tablas. Para que funcione jBPM webapp, debe crear por lo menos algunos registros en la tabla jbpm_id_user. Para tener exactamente las mismas entradas en esta tabla que la distribución predeterminada del kit de inicio que se ejecuta en HSQLDB, recomendamos ejecutar la siguiente secuencia de comando.

```
insert into JBPM_ID_USER (ID_, CLASS_, NAME_, EMAIL_, PASSWORD_)
       values ('1', 'U', 'cookie monster', 'cookie.monster@sesamestreet.tv',
'crunchcrunch');
insert into JBPM_ID_USER (ID_,CLASS_, NAME_, EMAIL_, PASSWORD_)
       values ('2', 'U', 'ernie', 'ernie@sesamestreet.tv',
'canthereyoubert,theresabananainmyear');
insert into JBPM_ID_USER (ID_,CLASS_, NAME_, EMAIL_, PASSWORD_)
       values ('3', 'U', 'bert', 'bert@sesamestreet.tv',
'ernie,theresabananainyourear');
insert into JBPM_ID_USER (ID_,CLASS_, NAME_, EMAIL_, PASSWORD_)
       values ('4', 'U', 'grover', 'grover@sesamestreet.tv', 'mayday mayday');
```

8.1.3. Actualización de configuración de servidor JBoss jBPM

Antes de que realmente podamos usar la base de datos creada recientemente con la webapp predeterminada de JBoss jBPM es necesario efectuar algunas actualizaciones a la configuración JBoss jBPM. La ubicación del servidor es '${JBPM_SDK_HOME}/jbpm-server'. Lo primero que haremos para actualizar esta configuración es crear una fuente de datos que apunte a nuestra base de datos JbpmDB. En un segundo paso nos aseguraremos de que la webapp predeterminada se comunica con esta fuente de datos y ya no con la fuente de datos HSQLDB.

```
<?xml version="1.0" encoding="UTF-8"?>

<datasources>
  <local-tx-datasource>
    <jndi-name>JbpmDS</jndi-name>
    <connection-url>jdbc:postgresql://localhost:5432/JbpmDB</connection-url>
    <driver-class>org.postgresql.Driver</driver-class>
    <user-name>user</user-name>
    <password>password</password>
    <metadata>
      <type-mapping>PostgreSQL 8.1</type-mapping>
    </metadata>
  </local-tx-datasource>
</datasources>
```

Para crear una fuente de datos debemos crear un archivo llamado, por ejemplo, jbpm-ds.xml con el contenido indicado en la lista de programa anterior. Naturalmente que es posible que sea necesario cambiar algunos de los valores en este archivo para ajustarlo a su situación en particular. Sólo tiene que guardar este archivo en la carpeta ${JBPM_SDK_HOME}/jbpm-server/server/jbpm/deploy. Felicitaciones: acaba de crear

una fuente de datos para su servidor JBoss jBPM. Bueno, casi... Para que todo realmente funcione tiene que copiar el controlador JDBC correcto en la carpeta ${JBPM_SDK_HOME}/jbpm-server/server/jbpm/lib. Ya hemos utilizado antes este controlador mientras lo instalábamos en DBVisualizer para poder examinar la base de datos recién creada. El nombre del archivo es 'postgresql-8.1-*.jdbc3.jar' y se encuentra en la subcarpeta jdbc de la carpeta de instalación PostgreSQL.

Si no está utilizando PostgreSQL, es posible que se pregunte dónde encontrará los parámetros para efectuar este paso. Para una gran cantidad de bases de datos admitidas por el servidor de aplicaciones JBoss, debe descargar una distribución JBoss AS y revisar la carpeta 'docs/examples/jca'.

Hacer que la webapp predeterminada se comunique con la fuente de datos correcta tampoco es muy difícil. El primer paso es simplemente buscar el archivo 'jboss-service.xml' en la carpeta '${JBPM_SDK_HOME}/jbpm-server/server/jbpm/deploy/jbpm.sar/META-INF'. Cambie el contenido de este archivo con el contenido de la lista que se muestra a continuación. Un lector atento se dará cuenta de que la única diferencia es un intercambio del token 'DefaultDS' by 'JbpmDS'.

```
<?xml version="1.0" encoding="UTF-8"?>

<server>
  <mbean code="org.jbpm.db.jmx.JbpmService"
         name="jboss.jbpm:name=DefaultJbpm,service=JbpmService"
         description="Default jBPM Service">
    <attribute name="JndiName">java:/jbpm/JbpmConfiguration</attribute>
    <depends>jboss.jca:service=DataSourceBinding,name=JbpmDS</depends>
  </mbean>
</server>
```

Lo último que tenemos que hacer es lograr que todo lo que se ejecute sea una manipulación del archivo 'jbpm.sar.cfg.jar' en la carpeta '${JBPM_SDK_HOME}/jbpm-server/server/jbpm/deploy/jbpm.sar'. Es necesario extraer este archivo en alguna ubicación y abrir el archivo llamado 'hibernate.cfg.xml'. Luego cambie la sección que contiene las propiedades de conexión jdbc. Esta sección es similar a la siguiente lista. Este archivo tiene dos cambios: la propiedad hibernate.connection.datasource no debe apuntar a la fuente de datos JbpmDS creada en el primer paso de esta sección y la propiedad hibernate.dialect debe coincidir con el dialecto PostgreSQL.

```
<?xml version='1.0' encoding='utf-8'?>

<!DOCTYPE hibernate-configuration PUBLIC
          "-//Hibernate/Hibernate Configuration DTD 3.0//EN"
          "http://hibernate.sourceforge.net/hibernate-configuration-3.0.dtd">

<hibernate-configuration>
  <session-factory>

    <!-- jdbc connection properties -->
    <property
name="hibernate.dialect">org.hibernate.dialect.PostgreSQLDialect</property>
    <property name="hibernate.connection.datasource">java:/JbpmDS</property>

    <!-- other hibernate properties
    <property name="hibernate.show_sql">true</property>
```

```
    <property name="hibernate.format_sql">true</property>
    -->

    <!-- ########################################### -->
    <!-- # mapping files with external dependencies # -->
    <!-- ########################################### -->

    ...

  </session-factory>
</hibernate-configuration>
```

Ya estamos listos para activar el servidor y ver si la webapp funciona. Todavía no es posible iniciar ningún proceso ya que no se ha implementado ninguno. Para hacer esto consultaremos el documento en la implementación de definición de proceso.

8.2. Actualizacioens de base de datos

En el subproyecto jbpm.db, encontraremos lo siguiente:

- Una secuencia de comando SQL para crear el diagrama jBPM 3.0.2 (para Hypersonic)
- Una secuencia de comando SQL para crear el diagrama jBPM 3.1 (para Hypersonic)
- Una secuencia de comando SQL para ctualizar un diagrama jBPM 3.0.2 a un diagrama jBPM 3.1 (para Hypersonic)
- Las secuencias de comando ant para crear la actualización del diagrama

Las secuencias de comando SQL de diagrama se pueden encontrar en el directorio `hsqldb/upgrade.scripts`. Para ejecutar la herramienta de actualización de su base de datos, siga estas pautas:

- Requisito previo: Asegúrese de que el proyecto jbpm.db está instalado junto al proyecto jbpm. En el starters-kit, este es el caso automáticamente. Si jbpm está instalado en una ubicación diferente, actualice la dir jbpm.3.location en propiedades de versión de como corresponde.
- Requisito previo: Debe tener el jar de controlador JDBC adecuado para su base de datos.
- Actualice las propiedades en el build.properties en la raíz del proyecto jbpm.db:
  - **upgrade.hibernate.properties**: un archivo de propiedades que contiene las propiedades de conexión a su bse de datos en estilo de hibernación.
  - **upgrade.libdir**: el directorio que contiene los jars de controlador jdbc de la base de datos.
  - **upgrade.old.schema.script**: la secuencia de comando de generación de diagrama para crear el diagrama antiguo de la base de datos. (Si ya existe, no necesita esta propiedad.)
- Para la creación del diagrama antiguo y luego el cálculo de las diferencias, ejecute la secuencia de comando **`'ant upgrade.db.script'`**
- Para sólo calcular la secuencia de comando de actualización sin cargar primero el diagrama de base de datos antiguo, ejecute la secuencia de comando **`'ant upgrade.hibernate.schema.update'`**
- Después de un término correcto, encontrará la secuencia de comando de actualización en `build/database.upgrade.sql`

Para actualizar de jBPM 3.0.2 a jBPM 3.1, la secuencia de comando SQL de actualización generada (para HSQLDB) se muestra en la siguiente lista:

```
# New JBPM_MESSAGE table
create table JBPM_MESSAGE (
  ID_ bigint generated by default as identity (start with 1),
  CLASS_ char(1) not null,
  DESTINATION_ varchar(255),
  EXCEPTION_ varchar(255),
  ISSUSPENDED_ bit,
  TOKEN_ bigint,
  TEXT_ varchar(255),
  ACTION_ bigint,
  NODE_ bigint,
  TRANSITIONNAME_ varchar(255),
  TASKINSTANCE_ bigint,
  primary key (ID_)
);

# Added columns
alter table JBPM_ACTION add column ACTIONEXPRESSION_ varchar(255);
alter table JBPM_ACTION add column ISASYNC_ bit;
alter table JBPM_COMMENT add column VERSION_ integer;
alter table JBPM_ID_GROUP add column PARENT_ bigint;
alter table JBPM_NODE add column ISASYNC_ bit;
alter table JBPM_NODE add column DECISIONEXPRESSION_ varchar(255);
alter table JBPM_NODE add column ENDtareas_ bit;
alter table JBPM_PROCESSINSTANCE add column VERSION_ integer;
alter table JBPM_PROCESSINSTANCE add column ISSUSPENDED_ bit;
alter table JBPM_RUNTIMEACTION add column VERSION_ integer;
alter table JBPM_SWIMLANE add column ACTORIDEXPRESSION_ varchar(255);
alter table JBPM_SWIMLANE add column POOLEDACTORSEXPRESSION_ varchar(255);
alter table JBPM_TASK add column ISSIGNALLING_ bit;
alter table JBPM_TASK add column ACTORIDEXPRESSION_ varchar(255);
alter table JBPM_TASK add column POOLEDACTORSEXPRESSION_ varchar(255);
alter table JBPM_TASKINSTANCE add column CLASS_ char(1);
alter table JBPM_TASKINSTANCE add column ISSUSPENDED_ bit;
alter table JBPM_TASKINSTANCE add column ISOPEN_ bit;
alter table JBPM_TIMER add column ISSUSPENDED_ bit;
alter table JBPM_TOKEN add column VERSION_ integer;
alter table JBPM_TOKEN add column ISSUSPENDED_ bit;
alter table JBPM_TOKEN add column SUBPROCESSINSTANCE_ bigint;
alter table JBPM_VARIABLEINSTANCE add column TASKINSTANCE_ bigint;

# Added constraints
alter table JBPM_ID_GROUP add constraint FK_ID_GRP_PARENT foreign key (PARENT_)
references JBPM_ID_GROUP;
alter table JBPM_MESSAGE add constraint FK_MSG_TOKEN foreign key (TOKEN_)
references JBPM_TOKEN;
alter table JBPM_MESSAGE add constraint FK_CMD_NODE foreign key (NODE_)
references JBPM_NODE;
alter table JBPM_MESSAGE add constraint FK_CMD_ACTION foreign key (ACTION_)
references JBPM_ACTION;
alter table JBPM_MESSAGE add constraint FK_CMD_TASKINST foreign key
(TASKINSTANCE_) references JBPM_TASKINSTANCE;
alter table JBPM_TOKEN add constraint FK_TOKEN_SUBPI foreign key
(SUBPROCESSINSTANCE_) references JBPM_PROCESSINSTANCE;
alter table JBPM_VARIABLEINSTANCE add constraint FK_VAR_TSKINST foreign key
(TASKINSTANCE_) references JBPM_TASKINSTANCE;
```

8.3. Inicio del administrador hsqldb en JBoss

No es realmente crucial para jBPM, pero en algunas situaciones durante el desarrollo resulta conveniente abrir el administrador de base de datos hypersonic que permite acceder a los datos de la base de datos JBoss hypersonic.

Comience por abrir el explorador y navegue a la consola JMX del servidor jBPM. La dirección URL que debe usar en el explorador para hacer esto es: http://localhost:8080/jmx-console. Naturalmente que esto se verá ligeramente diferente si ejecuta jBPM en otra máquina o en otro puerto diferente al predeterminado. En la siguiente captura de pantalla muestra la página resultante:

Figura 8.6. Consola JMX JBoss jBPM

Si hace clic en el vínculo 'database=localDB,service=Hypersonic' en las entradas JBoss , verá la pantalla JMX MBean del administrador de base de datos HSQLDB. Si se desplaza por esta página, en la sección de operaciones, verá la operación 'startDatabaseManager()'. Esto se muestra en la siguiente captura de pantalla.

Figura 8.7. La HSQLDB MBean

Al hacer clic en el botón para invocar se inicia la aplicación HSQLDB Database Manager. Esta es una herramienta cliente de base de datos más bien difícil, pero funciona de manera aceptable para el propósito de ejecutar esta secuencia de comando generada. Puede alternar las teclas ALT-TAB para ver esta aplicación ya que puede estar oculta por otra ventana. En la siguiente figura se muestra esta aplicación con la secuencia de comando anterior cargada y lista para su ejecución. Al presionar el botón 'Execute SQL' (Ejecutar SQL) se ejecuta el cmando y se actualiza de manera eficaz la base de datos.

Figura 8.8. El administrador de base de datos HSQLDB

Capítulo 9. Modelamiento de procesos

9.1. Descripción general

Una definición de proceso representa una especificación formal de un proceso de negocios y se base en un gráfico dirigido. El gráfico está compuesto de nodos y transiciones. Cada nodo del gráfico es de un tipo específico. El tipo de nodo define el comportamiento del tiempo de ejecución. Una definición de proceso tiene exactamente una fecha de inicio.

Un testigo es una ruta de ejecución. Un testigo es el concepto de tiempo de ejecución que mantiene un apuntador a un nodo en el gráfico.

Una instancia de proceso es una ejecución de una definición de proceso. Cuando se crea una instancia de proceso, se crea un testigo para la ruta principal de ejecución. A este testigo se le conoce como testigo raíz del proceso y se coloca en el estado inicial de la definición de proceso.

Una señal instruye a un testigo a que continué con la ejecución del gráfico. Al recibir una señal sin nombre, el testigo deja su nodo actual sobre la transición de salida predeterminada. Cuando se especifica un nombre de transición en la señal, el testigo deja su nodo sobre la transición especificada. Se delega una señal entregada a la instancia de proceso del testigo raíz.

Después de que el testigo ingresa a un nodo, el nodo se ejecuta. Los nodos son responsables de la continuación de la ejecución del gráfico. La continuación de la ejecución del gráfico se realiza al hacer que el testigo deje el nodo. Cada tipo de nodo puede implementar un comportamiento diferente para la continuación de la ejecución del gráfico. Un nodo que no propaga la ejecución se comporta como un estado.

Las acciones son porciones de código java que se ejecutan ante eventos en la ejecución de proceso. El gráfico es un instrumento importante en la comunicación sobre los requerimientos del software. Sin embargo el gráfico es sólo una vista (proyección) del software que se produce. Oculta muchos detalles técnicos. Las acciones son un mecanismo para agregar detalles técnicos fuera de la representación gráfica. Una vez que el gráfico está en su lugar, se puede decorar con acciones. Los tipos de evento principal ingresan un nodo, dejan un nodo y toman una transición.

9.2. Gráfico de proceso

La base de una definición de proceso es un gráfico formado por nodos y transiciones. Esta información se expresa en un archivo xml llamado `processdefinition.xml`. Cada nodo tiene un tipo, como por ejemplo, estado, decisión, bifurcación, unión. Cada nodo tiene un conjunto de transiciones de salida. Es posible asignar un nombre a las transiciones que salen de un nodo para diferenciarlas. Por ejemplo: En el siguiente diagrama se muestra un gráfico de proceso del proceso de jBAY auction.

Figura 9.1. Gráfico del proceso auction

A continuación se muestra el gráfico del proceso de auction jBAY representado como xml:

```
<process-definition>

  <start-state>
    <transition to="auction" />
  </start-state>

  <state name="auction">
    <transition name="auction ends" to="salefork" />
    <transition name="cancel" to="end" />
  </state>

  <fork name="salefork">
    <transition name="shipping" to="send item" />
    <transition name="billing" to="receive money" />
  </fork>

  <state name="send item">
    <transition to="receive item" />
  </state>

  <state name="receive item">
    <transition to="salejoin" />
  </state>

  <state name="receive money">
    <transition to="send money" />
  </state>

  <state name="send money">
    <transition to="salejoin" />
  </state>

  <join name="salejoin">
    <transition to="end" />
  </join>

  <end-state name="end" />

</process-definition>
```

9.3. Nodos

Un gráfico de proceso está formado por nodos y transiciones. Para obtener mayor información sobre el gráfico y su modelo de ejecución, consulte el capítulo 4, Programación Orientada a Objetos.

Cada nodo tiene un tipo específico. El tipo de nodo determina lo que sucede cuando una ejecución llega en el nodo en el tiempo de ejecución. jBPM tiene un conjunto de tipos de nodo implementados previamente que se pueden usar. De manera alternativa, puede crear códigos personalizados para implementar su propio comportamiento específico de nodo.

9.3.1. Responsabilidades de nodos

Cada nodo tiene dos responsabilidades principales: primero, puede ejecutar en código java simple. Normalmente, el código java simple se refiere a la función del nodo, es

decir, creación de unas pocas instancias de tareas, envío de una notificación, actualización de una base de datos. En segundo lugar, un nodo es responsable de la propagación de la ejecución del proceso. Básicamente, cada nodo tiene las siguientes opciones para la propagación de la ejecución del proceso:

- **1. No propagar la ejecución.** En ese caso el nodo se comporta como un estado de espera.
- **2. Propagar la ejecución por una de las transiciones de salida del nodo.** Esto significa que el testigo que llegó originalmente en el nodo se transfiere por una de las transiciones de salida con el executionContext.leaveNode(String) de llamada de API. El nodo ahora actuará como un nodo automático en cuanto a que puede ejecutar alguna lógica de programación personalizada y luego continuar con la ejecución de proceso automáticamente sin esperar.
- **3. Crear rutas de ejecución.** Un nodo puede decidir crear testigos. Cada nuevo testigo representa una nueva ruta de ejecución y cada nuevo testigo se puede iniciar mediante las transiciones de salida del nodo. Un buen ejemplo de este tipo de comportamiento el nodo de bifurcación.
- **4. Finalizar rutas de ejecución.** Un nodo puede decidir finalizar una ruta de ejecución. Esto significa que el testigo se finaliza y se termina la ruta de ejecución.
- **5. De manera más general, un nodo puede modificar toda la estructura del tiempo de ejecución de la instancia del proceso.** La estructura del tiempo de ejecución es una instancia de proceso que contiene un árbol de testigos. Cada testigo representa una ruta de ejecución. Un nodo puede crear y finalizar testigos, poner cada testigo en un nodo del gráfico e iniciar testigos en transiciones.

jBPM contiene, como cualquier flujo de trabajo y motor BPM, un conjunto de tipos de nodo implementados previamente que tienen una configuración documentada y un comportamiento específicos. Pero la característica única de jBPM y de la fundación Programación Orientada a Objetos es que hemos abierto el modelo para los desarrolladores.Los desarrolladores pueden escribir su propio comportamiento de nodo y usarlo en un proceso.

Es ahí donde el flujo de trabajo tradicional y los sistemas BPM son mucho más cerrados. Normalmente proporcionan un conjunto fijo de tipos de nodo (llamado el lenguaje del proceso). Su lenguaje de proceso es cerrado y el modelo de ejecución está oculto en el ambiente de tiempo de ejecución. Resultados de investigaciones en patrones de flujo de trabajo demuestran que cualquier lenguaje no es suficientemente poderoso. Hemos optado por un modelo simple y permitir a los desarrolladores crear sus propios tipos de nodo. De esa manera el lenguaje de proceso JPDL es abierto .

A continuación, analizaremos los tipos de nodo más importantes de JPDL.

9.3.2. Nodo de tarea Nodetype

Un nodo de tarea representa una o más tareas que efectúan los usuarios. De este modo, cuando la ejecución llega en un nodo de tarea, las instancias de tarea se crean en las listas de tara de los participantes del flujo de trabajo. Posteriormente, el nodo se comporta como un estado de espera. Así cuando los usuarios realizan sus tareas, el término de la tarea genera la continuación de la ejecución. En otras palabras, conduce a una nueva señal que es llamada en el testigo.

9.3.3. Estado de Nodetype

Un estado es un estado de espera bare-bones. La diferencia con un nodo de tarea es que no se crea ninguna instancia de tarea en ninguna lista de tareas. Esto puede resultar útil si el proceso debe esperar por un sistema externo. Es decir, con la entrada del nodo (mediante una acción del evento de ingreso de nodo), se puede enviar un mensaje al sistema externo. Posteriormente, el nodo pasa a un estado de espera. Cuando el sistema externo envía un mensaje de respuesta, esto puede conducir a una token.signal(), que genera la continuación de la ejecución de proceso.

9.3.4. Decisión de Nodetype

En realidad existen dos maneras de modelar una decisión. La diferencia entre las dos se basa en *quién* toma la decisión. Si la decisión la toma el proceso (es decir: especificado en la definición de proceso). O si una entidad externa proporciona el resultado de la decisión.

Cuando los procesos toman la decisión, se debe usar un nodo de decisión. Existen básicamente dos maneras de especificar los criterios de decisión. La más fácil es mediante la adición de elementos de condición en las transiciones. Las condiciones son expresiones de secuencia de comando beanshell que arrojan como resultado un boleano. En el tiempo de ejecución, el nodo de decisión hace un bucle en sus transiciones de salida (en el orden especificado en el xml) y evalúa cada condición. Se toma la primera transición para la que las condiciones resuelven a 'true'. De manera alternativa, es posible especificar una implementación del DecisionHandler. La decisión se calcula en una clase java y el método de decisión de la implementación DecisionHandler devuelve como resultado la transición de salida seleccionada.

Cuando la decisión es tomada por una parte externa (es decir: no forma parte de la definición de proceso), se deben usar varias transiciones de salida de un estado o de un nodo de estado de espera. La transición de salida se puede entonces proporciona en la generación externa que reanuda la ejecución una vez que termina el estado de espera. Es decir

`Token.signal(String transitionName)` y `TaskInstance.end(String transitionName)`.

9.3.5. Bifurcación de Nodetype

Una bifurcación divide una ruta de ejecución en varias rutas de ejecución concurrentes. El comportamiento de bifurcación predeterminado es crear un testigo secundario para cada transición que sale de la bifurcación, creando una relación principal-secundario entre el testigo que llega a la bifurcación.

9.3.6. Unión de Nodetype

La unión predeterminada supone que todos los testigos que llegan a una unión son secundarios de un mismo principal. Esta situación se crea al utilizar la bifurcación como se mencionó anteriormente y cuando todos los testigos creados por una bifurcación llegan a la misma unión. Una unión termina cada testigo que ingresa a ella. A continuación, la unión examina la relación principal-secundario del testigo que ingresa a

la unión. Cuando han llegado todos los testigos relacionados, el testigo principal se propaga por la transición de salida (la única). Cuando aún existen testigos relacionados, la unión se comporta como un estado de espera.

9.3.7. Nodo de Nodetype

El nodo de tipo funciona en la situación en que se desea crear un código propio en un nodo. El nodo nodetype espera una acción de subelemento. La acción se ejecuta cuando la ejecución llega al nodo. El código que se escribe en el actionhandler puede hacer cualquier cosa que se desee pero también es responsable de la propagación de la ejecución.

Este nodo se puede usar si desea utilizar una JavaAPI para implementar alguna lógica funciona que sea importante para el analista de negocios. Al usar un nodo, éste se hace visible en la representación gráfica del proceso. Para fines de comparación, las acciones (que se cubren a continuación) permiten agregar un código que es invisible ien la representación gráfica del proceso, en caso de que la lógica no sea importante para el analista de negocios.

9.4. Transiciones

Las transiciones tienen un nodo fuente y un nodo destino. El nodo fuente se representa mediante la propiedad from y el nodo destino mediante la propiedad to.

Un nodo puede tener, de manera opcional, un nombre. Recuerde que la mayoría de las funciones jBPM dependen de la unicidad del nombre de transición. Si existe más de una transición con el mismo nombre, se toma la primera transición con el nombre dado. En caso de que se produzcan nombres duplicados de transición en un nodo, el método `Map getLeavingTransitionsMap()` entrega como resultado menos elementos que `List getLeavingTransitions()`.

La transición predeterminada es la primera transición en la lista.

9.5. Acciones

as acciones son porciones de código java que se ejecutan ante eventos en la ejecución de proceso. El gráfico es un instrumento importante en la comunicación sobre los requerimientos del software. Sin embargo el gráfico es sólo una vista (proyección) del software que se produce. Oculta muchos detalles técnicos. Las acciones son un mecanismo para agregar detalles técnicos fuera de la representación gráfica. Una vez que el gráfico está en su lugar, se puede decorar con acciones. Esto significa que se puede asociar el código java con el gráfico sin cambiar la estructura del gráfico. Los tipos de evento principal ingresan un nodo, dejan un nodo y toman una transición.

Observe la diferencia entre una acción que se coloca en un evento a diferencia de una acción que se coloca en un nodo. Las acciones que se colocan en un evento se ejecutan cuando se genera el evento. Las acciones o eventos no tienen manera de impactar sobre

el flujo de control del proceso. Es similar al patrón observer. Por otra parte, una acción que se coloca en un nodo tiene la responsabilidad de propagar la ejecución.

Veamos un ejemplo de una acción en un evento. Supongamos que deseamos hacer una actualización de una base de datos en una transición en particular. La actualización de la base de datos es técnicamente esencial, pero no es importante para el analista de negocios.

Figura 9.2. Una acción de actualización de base de datos

```
public class RemoveEmployeeUpdate implements ActionHandler {
  public void execute(ExecutionContext ctx) throws Exception {
    // Obtenga el empleado activado de las variables del proceso.
    String firedEmployee = (String) ctx.getContextInstance().getVariable("fired
employee");

    // al tomar la misma conexión de base de datos que la usada para las
actualizaciones jbpm, hemos
    // vuelto a usar la transacción jbpm de la actualización de nuestra base de
datos.
    Connection connection =
ctx.getProcessInstance().getJbpmSession().getSession().getConnection();
    Statement statement = connection.createStatement();
    statement.execute("DELETE FROM EMPLOYEE WHERE ...");
    statement.execute();
    statement.close();
  }
}

<process-definition name="yearly evaluation">

  ...
  <state name="fire employee">
    <transition to="collect badge">
      <action class="com.nomercy.hr.RemoveEmployeeUpdate" />
    </transition>
  </state>

  <state name="collect badge">
  ...

</process-definition>
```

9.5.1. Configuración de acciones

Para obtener mayor información sobre la adición de configuraciones a sus acciones personalizadas y sobre la manera de especificar la configuración en `processdefinition.xml`, consulte la sección 16.2.3, "Configuración de delegaciones"

9.5.2. Referencias de acción

Las acciones pueden tener un nombre dado. Es posible hacer referencia a las acciones con nombre desde otras ubicaciones donde se pueden especificar las acciones.

Asimismo, es posible colocar acciones con nombre como elementos secundarios en la definición de proceso.

Esta función es interesante si se desea limitar la duplicación de configuraciones de acción (es decir, cuando la acción tiene configuraciones complicadas). Otro caso de uso es la ejecución o programación de acciones de tiempo de ejecución.

9.5.3. Eventos

Los eventos especifican momentos en la ejecución del proceso. El motor jBPM genera eventos durante la ejecución del gráfico. Esto se produce cuando jbpm calcula el próximo estado (es decir: procesamiento de una señal). Un evento siempre es relativo con un elemento en la definición de proceso como por ejemplo, la definición de proceso, un nodo o una transición. La mayoría de los elementos de proceso pueden generar diferentes tipos de eventos. Por ejemplo un nodo puede generar un evento `node-enter` y un evento `node-leave`. Los eventos son generadores de acciones. Cada evento tiene una lista de acciones. Cuando el motor jBPM genera un evento, se ejecuta la lista de acciones.

9.5.4. Propagación de eventos

Los superestados crean una relación principal-secundario en los elementos de una definición de proceso. Los nodos y las transiciones presentes en un superestado tienen dicho superestado como un principal. Los elementos de nivel superior tienen las definiciones de proceso como principal. La definición de proceso no tiene un principal. Cuando se genera un evento, éste se propaga hasta la jerarquía principal. Esto permite, por ejemplo, capturar todos los eventos de transición en un proceso y asocia acciones con dichos eventos en una ubicación centralizada.

9.5.5. Secuencia de comando

Una secuencia de comando es una acción que ejecuta una secuencia de comando beanshell. Consulte el sitio Web the beanshell, para obtener más información sobre beanshell. De manera predeterminada, todas las variables de proceso se encuentran disponibles como variables de secuencia de comando y las variables que no son secuencias de comando se escriben en las variables del proceso. Las siguientes variables de secuencia de comando también se encuentran disponibles:

- executionContext
- token
- node
- task
- taskInstance

```
<process-definition>
  <event type="node-enter">
    <script>
      System.out.println("this script is entering node "+node);
    </script>
  </event>
  ...
</process-definition>
```

Para personalizar el comportamiento predeterminado de carga y almacenamiento de variables en la secuencia de comando, el elemento `variable` se puede usar como un subelemento de secuencia de comando. En ese caso, la expresión de secuencia de comando tiene también que colocarse en un subelemento de secuencia de comando: `expression`.

```
<process-definition>
  <event type="process-end">
    <script>
      <expression>
        a = b + c;
      </expression>
      <variable name='XXX' access='write' mapped-name='a' />
      <variable name='YYY' access='read' mapped-name='b' />
      <variable name='ZZZ' access='read' mapped-name='c' />
    </script>
  </event>
  ...
</process-definition>
```

Antes de que comience la secuencia de comando, las variables de proceso variables `YYY` y `ZZZ` se hacen disponibles a la secuencia de comando como variables de secuencia de comando b y crespectivamente. Cuando termina la secuencia de comando, el valor de la variable de secuencia de comando `a` se almacena en la variable de proceso almacenada `XXX`.

Si el atributo `access` de `variable` contiene '`read`', la variable de proceso se carga como una variable de secuencia de comando antes de la evaluación de la secuencia de comando. Si el atributo `access` contiene '`write`', la variable de comando se almacena como variable de proceso después de la evaluación. El atributo `mapped-name` puede hacer que la variable de proceso esté disponible con otro nombre en la secuencia de comando. Esto puede resultar útil cuando los nombres de la variable de proceso contienen espacios u otros caracteres literales de secuencia de comando no válidos.

9.5.6. Eventos personalizados

Observe que es posible generar sus propios eventos de cliente durante la ejecución de un proceso. Los eventos están definidos de manera única por la combinación de un elemento de gráfico (nodos, transiciones, definiciones de proceso y superestados son elementos de gráfico) y un tipo de evento (java.lang.String). jBPM define un conjunto de eventos que se generan para los nodos, las transiciones y otros elementos de gráfico. Sin embargo como usuario, puede generar sus propios eventos. En las acciones, en sus implementaciones de nodo personalizado o aun fuera de la ejecución de una instancia de proceso, puede llamar a `GraphElement.fireEvent(String eventType, ExecutionContext executionContext);`. No existen restricciones para seleccionar los nombres de los tipos de evento.

9.6. Superestados

Un superestado es un grupo nodos. Los superestados pueden estar anidados de manera

recursiva. Los superestados se pueden usar para incorporar alguna jerarquía en la definición de proceso. Por ejemplo, una aplicación se puede usar para agrupa todos los nodos de un proceso en fases. Las acciones se pueden asociar con eventos de superestado. Una consecuencia es que un testigo puede estar en varios nodos anidados en un momento en particular. Esto puede resultar conveniente para verificar si una ejecución de proceso se encuentra, por ejemplo, en la fase de inicio. En el model jBPM, el usuario puede agrupar cualquier conjunto de nodos en un superestado.

9.6.1. Transiciones de superestados

Los testigos de los nodos presentes en el superestado pueden tomar todas las transiciones de salida de un superestado. Las transiciones también pueden llegar en superestados. En dicho caso, el testigo se redirige al primer nodo del superestado. Los nodos fuera del superestados pueden tener transiciones directamente a los nodos dentro del superestado. Asimismo, de manera contraria, los nodos de los superestados pueden tener transiciones a los nodos fuera del superestado o al superestado mismo. Los superestados pueden también tener diferentes autoreferencias.

9.6.2. Eventos de superestado

Existen dos eventos exclusivos de los superestados: `superstate-enter` y `superstate-leave`. Estos eventos se generan al margen de las transiciones donde ingresen o salgan los nodos respectivamente. Estos eventos no se generan siempre y cuando un testigo toma las transiciones en el superestados.

Recuerde que hemos creado tipos de evento separados para los estados y los superestados. Esto facilita la diferenciación entre eventos de superestado y eventos de nodo que se propagan desde el superestado.

9.6.3. Nombres jerárquicos

Los nombres de nodo tienen que ser únicos en su alcance. El alcance de un nudo es su colección de nodos. Tanto la definición de proceso como el superestado son colecciones de nodo. Para consultar los nodos de los superestados, tiene que especificar el relativo, barra diagonal (/) nombre separado. La barra diagonal separa los nombres de nodo. Use las comillas ('..') para referirse a un nivel superior. En el siguiente ejemplo se indica la manera de hacer referencia a un nodo en un superestado:

```
<process-definition>
  ...
  <state name="preparation">
    <transition to="phase one/invite murphy"/>
  </state>
  <super-state name="phase one">
    <state name="invite murphy"/>
  </super-state>
  ...
</process-definition>
```

En el siguiente ejemplo se muestra cómo subir por la jerarquía del superestado

```
<process-definition>
```

```
  ...
  <super-state name="phase one">
    <state name="preparation">
      <transition to="../phase two/invite murphy"/>
    </state>
  </super-state>
  <super-state name="phase two">
    <state name="invite murphy"/>
  </super-state>
  ...
</process-definition>
```

9.7. Manejo de excepciones

El mecanismo de manejo de excepciones de jBPM únicamente es válido para excepciones java. La ejecución del gráfico en sí no genera problemas. Únicamente la ejecución de las clases de delegación puede conducir a excepciones.

En `process-definition`s, `node`s y `transition`s, se puede especificar una lista de `exception-handlers`. Cada `exception-handler` tiene una lista de acciones. Cuando se produce una excepción en una clase de delegación, se busca una `exception-handler` adecuada en la jerarquía principal del elemento de proceso. Una vez que se encuentra, se ejecutan las acciones de `exception-handler`..

Observe que el mecanismo de manejo de excepciones de jBPM no es completamente similar al manejo de excepciones java. En java, una excepción atrapada puede influir sobre el flujo de control. En el caso de jBPM, el mecanismo de manejo de excepciones de jBPM no puede cambiar el flujo de control. La excepción se atrapa o no se atrapa. Las excepciones no atrapadas se colocan en el cliente (es decir, el cliente que llama a `token.signal()`) o la excepción es atrapada por una jBPM `exception-handler`. Para excepciones de atrapada, la ejecución del gráfico continúa como si no se hubiera producido ninguna excepción.

Observe que en una `action` que maneja una excepción es posible colocar el testigo en un nodo arbitrario en el gráfico con `Token.setNode(Node node.`

9.8. Composición de proceso

jBPM admite la composición de proceso mediante `process-state`. El estado de proceso es un estado asociado con otra definición de proceso. Cuando llega una ejecución de gráfico al estado de proceso, se crea una nueva instancia de proceso del subproceso y se asocia con la ruta de la ejecución que llegó al estado de proceso. Lar ruta de la ejecución del superproceso espera hasta que finalice la instancia del subproceso. Cuando finaliza la instancia del subproceso, la ruta de ejecución del superproceso sale del estado de proceso y continúa la ejecución del gráfico en el superproceso.

```
<process-definition name="hire">
  <start-state>
    <transition to="initial interview" />
  </start-state>
```

```
  <process-state name="initial interview">
    <sub-process name="interview" />
    <variable name="a" access="read,write" mapped-name="aa" />
    <variable name="b" access="read" mapped-name="bb" />
    <transition to="..." />
  </process-state>
  ...
</process-definition>
```

Este proceso 'hire' contiene un `process-state` que genera un proceso 'interview'. Cuando la ejecución llega en la primera 'interview', se crea una nueva ejecución (=instancia de proceso) para la versión más reciente del proceso 'interview'. A continuación la variable 'a' del proceso hire se copia en la variable 'aa' del proceso interview. De igual manera, la variable hire variable 'b' se copia en la variable interview 'bb'. Una vez que termina el proceso interview, sólo la variable 'aa' del proceso interview se vuelve a copiar en la variable 'a' del proceso hire.

En general, cuando se inicia un subproceso, todas las `variable`s con acceso `read` se leen en el superproceso y se alimentan al subproceso recién creado antes de que se dé la señal de salida del estado de inicio. Cuando finalizan las instancias de subproceso, todas las `variable`s con acceso de `write` se copian del subproceso al superproceso. El atributo `mapped-name` del elemento variable permite especificar el nombre de variable que se debe usar en el subproceso.

9.9. Comportamiento de nodo personalizado

En jBPM es bastante fácil crear nodos personalizados. Para crear nodos personalizados es necesario escribir una implementación del ActionHandler. La implementación puede ejecutar cualquier lógica de negocio y además tiene la responsabilidad de propagar la ejecución de gráfico. Veamos un ejemplo que actualiza un sistema ERP. Leeremos un monto del sistema ERP, agregaremos un monto almacenado en las variables de proceso y almacenaremos el resultado de vuelta en el sistema ERP. Según la magnitud del monto, tendremos que salir del nodo mediante la transición 'small amounts' (montos pequeños) o 'large amounts' (montos grandes).

Figura 9.3. Fragmento de ejemplo de proceso de actualización de ERP

```
public class AmountUpdate implements ActionHandler {
  public void execute(ExecutionContext ctx) throws Exception {
    // lógica de negocios
    Float erpAmount = ...get amount from erp-system...;
    Float processAmount = (Float)
ctx.getContextInstance().getVariable("amount");
    float result = erpAmount.floatValue() + processAmount.floatValue();
    ...update erp-system with the result...;

    // propagación de ejecución de gráfico
    if (result > 5000) {
      ctx.leaveNode(ctx, "big amounts");
    } else {
      ctx.leaveNode(ctx, "small amounts");
```

```
      }
    }
}
```

También es posible crear y unir testigos en implementaciones personalizadas de nodos. Para ver un ejemplo de cómo hacer esto, revise la implementación de nodo de bifurcación y de unión en el código fuente jbpm :-).

9.10. Ejecución de gráfico

El modelo de ejecución de gráficos de jBPM se basa en la interpretación de la definición de proceso y en el patrón de la cadena de comandos.

Interpretación de definición de proceso significa que los datos de la definición de proceso se almacenan en la base de datos. La información de definición de proceso se usa durante la ejecución de proceso en el tiempo de ejecución. En caso de que haya alguna preocupación: utilizamos el caché de segundo nivel de hibernación para evitar cargar la información de definición en el tiempo de ejecución. Ya que las definiciones de proceso no cambian (consulte la creación de versión de proceso), la hibernación puede almacenar en caché las definiciones de proceso en memoria.

El patrón de la cadena de comando significa que cada nodo en el gráfico es responsable de la propagación de la ejecución del proceso. Si un nodo no propaga la ejecución, se comporta como si estuviera en estado de espera.

El objetivo es iniciar la ejecución en las instancias de proceso y que la ejecución continúe hasta que entre en un estado de espera.

Un testigo representa una ruta de ejecución. Un testigo tiene un apuntador a un nodo en el gráfico de proceso. Durante los estados de espera, los testigos pueden persistir en la base de datos. Ahora veremos el algoritmo para calcular la ejecución de un testigo. La ejecución se inicia cuando se envía una señal al testigo. La ejecución luego se transfiere por las transiciones y nodos mediante el patrón de la cadena de comandos. Estos son los métodos pertinentes en un diagrama de clase.

Figura 9.4. Métodos relacionados de ejecución de gráfico

Cuando un testigo está en un nodo, se pueden usar señales para enviarlas al testigo. El envío de una señal es una instrucción para comenzar la ejecución. Por lo tanto una señal debe especificar una transición de salida del nodo actual del testigo. La primera transición es la predeterminada. En una señal enviada a un testigo, éste toma su nodo actual y llama al método `Node.leave(ExecutionContext,Transition)`. Pensemos en ExecutionContext como un testigo ya que el principal objeto en ExecutionContext es un testigo. El método `Node.leave(ExecutionContext,Transition)` genera el evento `node-leave` y llama a `Transition.take(ExecutionContext)`. Dicho método genera el evento `transition` y llama a `Node.enter(ExecutionContext)` en el nodo de destino de la transición. Dicho método genera el evento `node-enter` y llama a `Node.execute(ExecutionContext)`. Cada tipo de nodo tiene su propio

comportamiento que se implementa en el método de ejecución. Cada nodo es responsable de la propagación de la ejecución de gráfico llamando nuevamente a `Node.leave(ExecutionContext,Transition)`. En resumen:

- Token.signal(Transition)
- --> Node.leave(ExecutionContext,Transition)
- --> Transition.take(ExecutionContext)
- --> Node.enter(ExecutionContext)
- --> Node.execute(ExecutionContext)

Observe que el cálculo completo del siguiente estado, incluida la invocación de las accione, se hace en el subproceso del cliente. Existe la creencia errónea de que todos los cálculos se *deben* hacer en el subproceso del cliente. Tal como sucede con cualquier invocación asíncrona, es posible usar mensajería asíncrona (JMS) para eso. Cuando se envía el mensaje en la misma transacción que la actualización de instancia de proceso, se solucionan todos los problemas de sincronización. Algunos sistemas de flujo de trabajo usan mensajería asíncrona entre todos los nodos y el gráfico. Sin embargo, en ambientes de alta productividad, este algoritmo proporciona mucho más control y flexibilidad para afectar el rendimiento de un proceso de negocios.

9.11. Demarcación de transacción

Tal como se explicó en la sección 9.10, "Ejecución de gráfico" y en el capítulo 4, Programación Orientada a Objetos, jBPM ejecuta el proceso en el subproceso del cliente y es asíncrono por naturaleza. Esto significa que `token.signal()` o `taskInstance.end()` sólo devolverán un resultado cuando el proceso haya ingresado en un nuevo estado de espera.

La función jPDL que se describe aquí es desde una perspectiva de modelamiento corresponde al capítulo 13, Continuaciones asíncronas.

En la mayoría de las situaciones, este es el enfoque más directo ya que la ejecución de proceso se puede vincular fácilmente a las transacciones del servidor: el proceso se mueve de un estado al siguiente en una transacción.

En algunas situaciones en que los cálculos del proceso toman mucho tiempo, este comportamiento puede resultar no deseable. Para manejar esto, jBPM incluye un sistema de mensajería asíncrona que permite continuar un proceso de manera asíncrona. Naturalmente, en un ambiente empresarial java es posible configurar jBPM para que use un sistema de distribución de mensajes JMS en vez de la versión en el sistema de mensajería.

En cualquier nodo, jPDL admite el atributo `async="true"`. Los nodos asíncronos no se ejecutan en el subproceso del cliente. En cambio, se envía un mensaje por el sistema de mensajería síncrono y el subproceso se devuelve al cliente (es decir, se devuelve `token.signal()` o `taskInstance.end()`).

Recuerde que el código de cliente jbpm ahora puede realizar la transacción. El envío del mensaje se debe realizar en la misma transacción que las actualizaciones de proceso. De modo que el resultado neto de la transacción es que el indicador se ha movido al siguiente nodo (que aún no se ha ejecutado) y un mensaje `org.jbpm.command.ExecuteNodeCommand` se ha enviado por el sistema de mensajería asíncrona al Ejecutor de comandos jBPM.

El Ejecutor de comandos jBPM lee los comandos de la cola de espera y los ejecuta. En el caso de `org.jbpm.command.ExecuteNodeCommand`, el proceso continúa con la ejecución del nodo. Cada comando se ejecuta en una transacción separada.

De modo que para continúen los procesos asíncronos, es necesario que un Ejecutor de comandos jBPM esté funcionando. La manera más sencilla de hacer esto es configurar `CommandExecutionServlet` en su aplicación Web. De manera alternativa, se debe asegurar que el subproceso CommandExecutor esté funcionando de cualquier otra manera.

Como modelador de procesos, en realidad no debe preocuparle la mensajería asíncrona. El punto principal a tener en cuenta es la demarcación de la transacción: jBPM funcionará de manera predeterminada en la transacción del cliente y efectuará todo el cálculo hasta que el proceso entra en un estado de espera. Use `async="true"`para demarcar una transacción en el proceso.

Veamos un ejemplo:

```
...
<start-state>
  <transition to="one" />
</start-state>
<node async="true" name="one">
  <action class="com...MyAutomaticAction" />
  <transition to="two" />
</node>
<node async="true" name="two">
  <action class="com...MyAutomaticAction" />
  <transition to="three" />
</node>
<node async="true" name="three">
  <action class="com...MyAutomaticAction" />
  <transition to="end" />
</node>
<end-state name="end" />
...
```

El código de cliente para interactuar con las ejecuciones de proceso (inicio y continuación) es exactamente el iso que con los procesos normales (síncronos):

```
...start a transaction...
JbpmContext jbpmContext = jbpmConfiguration.createContext();
try {
  ProcessInstance processInstance = jbpmContext.newProcessInstance("my async
process");
  processInstance.signal();
  jbpmContext.save(processInstance);
} finally {
  jbpmContext.close();
}
```

Después de esta primera transacción, el testigo de raíz de la instancia de proceso apunta al nodo one y se habrá enviado un mensaje `ExecuteNodeCommand` al ejecutor de comandos.

En una transacción siguiente, el ejecutor de comandos lee el mensaje de la cola de espera y ejecuta el nodo `one`. La acción puede decidir propagar la ejecución o entrar en un estado de espera. Si la acción decides propagar la ejecución, la transacción termina cuando la ejecución llega al nodo dos. Y así sucesivamente...

Capítulo 10. Context0

El contexto se refiere a las variables de proceso. Las variables de proceso son pares de valores clave que actualizan la información relacionada con la instancia del proceso. Ya que el contexto se debe poder almacenar en una base de datos, existen algunas limitaciones menores.

10.1. Acceso a variables

La org.jbpm.context.exe.ContextInstance sirve como la interfaz central para trabajar con las variables de proceso. Es posible obtener la ContextInstance de una ProcessInstance como la siguiente:

```
ProcessInstance processInstance = ...;
ContextInstance contextInstance = (ContextInstance)
processInstance.getInstance(ContextInstance.class);
```

Las operaciones más básicas son

```
void ContextInstance.setVariable(String variableName, Object value);
void ContextInstance.setVariable(String variableName, Object value, Token
token);
Object ContextInstance.getVariable(String variableName);
Object ContextInstance.getVariable(String variableName, Token token);
```

Los nombres de variable son java.lang.String. jBPM admite de manera predeterminada los siguientes tipos de valor:

- `java.lang.String`
- `java.lang.Boolean`
- `java.lang.Character`
- `java.lang.Float`
- `java.lang.Double`
- `java.lang.Long`
- `java.lang.Byte`
- `java.lang.Short`
- `java.lang.Integer`
- `java.util.Date`
- `byte[]`
- `java.io.Serializable`
- `classes that are persistable with hibernate`

Asimismo, es posible almacenar un valor nulo sin tipo con persistencia.

Todos los demás tipos se pueden almacenar en las variables de proceso sin ningún problema. Pero causará una excepción cuando se intenta guardar la instancia de proceso.

Para configurar jBPM para almacenar objetos persistentes de hibernación en las variables, consulte Almacenamiento de objetos persistentes de hibernación.

10.2. Duración de variables

No es necesario declarar las variables en el archivo de proceso. En el tiempo de ejecución basta con colocar cualquier objeto java en las variables. Si dicha variable no está presente, se creará. Tal como sucede con una `java.util.Map` simple.

Las variables se pueden eliminar con

```
ContextInstance.deleteVariable(String variableName);
ContextInstance.deleteVariable(String variableName, Token token);
```

La capacidad de cambio automático de los tipos ahora es admitida. Esto significa que está permitido sobrescribir una variable con un valor de un tipo diferente. Naturalmente, se debe intentar limitar el número de cambios de tipo ya que esto crea una comunicación db mayor que una simple actualización de una columna.

10.3. Persistencia de variable

Las variables forman parte de la instancia de proceso. Al guardar la instancia de proceso en la base de datos hace que la base de datos esté sincronizada con la instancia de proceso. Las variables se crean, actualizan y elimina de la base de datos como resultado de guardar (=actualizar) la instancia de proceso en la base de datos. Consulte el capítulo 7, Persistencia.

10.4. Alcances de variables

Cada ruta de ejecución (es decir testigo) tiene su propio conjunto de variables de proceso. La solicitud de una variable siempre se hace en un testigo. Las instancias de proceso tienen un árbol de testigos (consulte programación orientada a objetos). Al solicitar una variable sin especificar un testigo, el testigo predeterminado es el testigo raíz.

La consulta de variable se hace de manera recursiva por los principales del testigo dado. El comportamiento es similar al alcance de las variables en los lenguajes de programación.

Cuando una variable no existente se configura en un testigo, la variable se crea en el testigo raíz. Esto significa que cada variable tiene un alcance de proceso de manera predeterminada. Para hacer que un testigo de variable sea local, es necesario crearlo explícitamente con:

```
ContextInstance.createVariable(String name, Object value, Token token);
```

10.4.1. Sobrecarga de variables

La sobrecarga de variables significa que cada ruta de ejecución puede tener su propia copia de una variable con el mismo nombre. Se consideran como independientes por lo que pueden ser de diferentes tipos. La sobrecarga de variables puede ser interesante si se inician varias rutas de ejecución concurrentes por la misma transición. Lo único que diferenciaría esas rutas de ejecución es los respectivos conjuntos de variables.

10.4.2. Anulación de variables

La anulación de variables significa que las variables de rutas de ejecución anidadas anulan las variables en rutas de ejecución más globales. Por lo general, las rutas de ejecución anidadas se relacionan con la concurrencia: las rutas de ejecución entre una bifurcación y una unión son secundarias (anidadas) de la ruta de ejecución que llegó a la bifurcación. Por ejemplo, si tiene una variable 'contact' (contacto) en el alcance de la instancia de proceso, puede anular esta variable en las rutas de ejecución anidadas 'shipping' (envío) y 'billing' (facturación).

10.4.3. Alcance de variable de instancia

Para obtener mayor información sobre las variables de instancia de tarea, consulte la sección 11.4, "Variables de instancia de tarea".

10.5. Variables transitorias

Cuando una instancia de proceso persiste en la base de datos, las variables normales también persisten como parte de la instancia de proceso. En algunas situaciones es posible usar una variable en una clase de delegación, pero es recomendable no almacenarla en la base de datos. Un ejemplo podría ser una conexión de base de datos que desea transferir externamente a jBPM a una clase de delegación. Esto se pude efectuar mediante las variables transitorias.

La duración de las variables transitorias es igual que la duración del objeto java ProcessInstance.

Debido a su naturaleza, las variables transitorias no se relacionan con un testigo. Por lo que solamente existe una asociación de variables transitorias para un objeto de instancia de proceso.

Es posible acceder a las variables transitorias mediante sus conjuntos de métodos en la instancia de contexto y no es necesario declarar en el processdefinition.xml.

```
Object ContextInstance.getTransientVariable(String name);
void ContextInstance.setTransientVariable(String name, Object value);
```

10.6. Personalización de persistencia de variables

Las variables se almacenan en la base de datos en un enfoque de dos pasos:

```
user-java-object <---> converter <---> variable instance
```

Las variables se almacenan en `VariableInstance`s. Los miembros de `VariableInstance`s se asocian a los campos de la base de datos con hibernación. La configuración predeterminada de jBPM utiliza 6 tipos de VariableInstances:

- `DateInstance` (con un campo java.lang.Date que se asocia a `Types.TIMESTAMP` en la base de datos))

- `DoubleInstance` (con un campo java.lang.Double que se asocia a `Types.DOUBLE` en la base de datos)
- `StringInstance` (con un campo java.lang.String que se asocia a `Types.VARCHAR` en la base de datos)
- `LongInstance` (con un campo java.lang.Long que se asocia a `Types.BIGINY` en la base de datos)
- `HibernateLongInstance` (que se usa para los tipos que se pueden hibernar con un campo de id largo Se asocia un campo java.lang.Object como referencia a una entidad de hibernación en la base de datos)
- `HibernateStringInstance` (que se usa para los tipos que se pueden hibernar con un campo de id de secuencia de comando. Se asocia un campo java.lang.Object como referencia a una entidad de hibernación en la base de datos)

Los `Converter`s convierten objetos de usuario java y los objetos java pueden ser almacenados por las `VariableInstance`s. De modo que cuando se configura una variable de proceso con, por ejemplo `ContextInstance.setVariable(String variableName, Object value)`, el valor se convierte opcionalmente con un convertidor. A continuación el objeto convertido se almacena en una `VariableInstance`. Los `Converter`s son implementaciones de la siguiente interfaz:

```
public interface Converter extends Serializable {
  boolean supports(Object value);
  Object convert(Object o);
  Object revert(Object o);
}
```

Los convertidores son opcionales. Los convertidores deben estar disponibles para el cargador de clase jBPM

La forma como se convierten y almacenan los objetos java de usuario en las instancias de variable se configura en el archivo `org/jbpm/context/exe/jbpm.varmapping.properties`. Para personalizar este archivo de propiedad, coloque una versión modificada en la raíz de la ruta de clase, como se explica en la sección 6.1, "Configuración de archivos". Cada línea del archivo de propiedades especifica 2 ó 3 nombres de clase separados por espacios: el nombre de clase del objeto java de usuario, opcionalmente el nombre de clase del convertidor y el nombre de clase de la instancia de variable. Cuando utilice sus convertidores personalizados, asegúrese de que estén en la ruta de clase jBPM. Cuando utilice las instancias de variable personalizadas, también deben estar en la ruta de clase jBPM y el archivo de asociación de hibernación de `org/jbpm/context/exe/VariableInstance.hbm.xml` se debe actualizar para incluir la subclase personalizada de VariableInstance.

Por ejemplo, veamos el siguiente fragmento xml en el archivo `org/jbpm/context/exe/jbpm.varmapping.xml`

```
    <jbpm-type>
      <matcher>
```

```
        <bean class="org.jbpm.context.exe.matcher.ClassNameMatcher">
          <field name="className"><string value="java.lang.Boolean" /></field>
        </bean>
      </matcher>
      <converter class="org.jbpm.context.exe.converter.BooleanToStringConverter"
/>
      <variable-instance
class="org.jbpm.context.exe.variableinstance.StringInstance" />
    </jbpm-type>
```

Este fragmento especifica que todos los objetos de tipo `java.lang.Boolean` se deben convertir usando el convertidor `BooleanToStringConverter` y que el objeto resultante (una secuencia de comandos) se almacena en un objeto de instancia de variable de tipo StringInstance.

Si no se ha especificado un convertidor como en

```
    <jbpm-type>
      <matcher>
        <bean class="org.jbpm.context.exe.matcher.ClassNameMatcher">
          <field name="className"><string value="java.lang.Long" /></field>
        </bean>
      </matcher>
      <variable-instance
class="org.jbpm.context.exe.variableinstance.LongInstance" />
    </jbpm-type>
```

Eso significa que los objetos Long que se colocan en las variables únicamente están almacenados en una instancia de variable de tipo LongInstance sin ser convertida.

Capítulo 11. Administración de tareas

La principal función del jBPM es la habilidad de ejecutar un proceso con persistencia. Esta característica es extremadamente útil en la administración de tareas y listas de tareas para las personas. El jBPM permite especificar una proción de software que describe un proceso general que puede contener estados de espera para tareas humanas.

11.1. Tareas

Las tareas forman parte de la definición de proceso y definen la manera en que deben crearse y asignarse las instancias del proceso durante las ejecuciones del proceso.

Las tareas pueden ser definidas en `task-node`s y en `process-definition`. La forma más usual es definir una o más `tasks` en un `task-node`. En ese caso, el `task-node` representa una tarea a ser realizada por el usuario y la ejecución del proceso debe esperar hasta que el actor finalice la tarea. Cuando el actor finaliza la tarea, el proceso de ejecución debe continuar. Cuando se especifican más tareas en un `task-node`, el comportamiento predeterminado o por defecto es esperar a que todas las tareas sean finalizadas.

Las tareas también se pueden especificar en `process-definition`. Las tareas especificadas en la definición del proceso se pueden buscar por nombre y por referencia desde los `task-node`s o usadas desde acciones internas. De hecho, es posible buscar por nombre todas las tareas (también en task-nodes) a las que se asigna un nombre en la definición del proceso.

Los nombres de las tareas deben ser únicos dentro de toda la definición del proceso. Puede asignarse una `priority` a las tareas. Esta prioridad se usa como la prioridad de cada instancia de tareas creada para esta tarea. Posteriormente, las instancias de las tareas pueden cambiar esta prioridad inicial.

11.2. Instancias de las tareas

Es posible asignar una instancia de tarea a un actorId (java.lang.String). Todas las instancias de las tareas se almacenan en un cuadro de la base de datos (JBPM_TASKINSTANCE). Consultando este cuadro para revisar las instancias de las tareas para un actorId determinado, se obtiene la lista de tareas de ese usuario en particular.

El mecanismo de la lista de tareas jBPM puede combinar tareas jBPM con otras tareas, aun con aquellas tareas que no estén relacionadas con la ejecución de un proceso. De esta forma, los desarrolladores de jBPM pueden combinar fácilmente tareas de proceso jBPM con tareas de otras aplicaciones, en un almacén centralizado de listas de tareas.

11.2.1. Ciclo de vida de una instancia de tareas

El ciclo de vida de una instancia de tareas es directo: después de ser creadas, las instancias de tareas pueden ser iniciadas de forma opcional. Posteriormente las instancias de tareas pueden ser finalizadas, lo que significa que la instancia de tarea

queda marcada como terminada.

Recuerde que para mayor flexibilidad, la asignación no forma parte de la duración, por lo que las instancias de tareas pueden ser asignadas o no asignadas. La asignación de instancias de tareas no influye en la duración de la instancia de tareas.

Las instancias de tareas son creadas comúnmente por la ejecución del proceso, ingresando un `task-node` (con el método `TaskMgmtInstance.createTaskInstance(...)`). Luego, un componente interfaz de usuario requerirá a la base de datos las listas de tareas usando el `TaskMgmtSession.findTaskInstancesByActorId(...)`. Luego, después de recoger la información del usuario, el componente IU solicita `TaskInstance.assign(String)`, `TaskInstance.start()` o `TaskInstance.end(...)`.

Una instancia de tareas mantiene su estado por medio de propiedades de fecha : `create`, `start` y `end`. Se puede acceder a estas propiedades a través de sus respectivos buscadores en `TaskInstance`.

Actualmente, las instancias de tareas terminadas son marcadas con una fecha de término para que no sean buscadas a través de consultas posteriores a las listas de tareas. Permanecen en la tabla JBPM_TASKINSTANCE.

11.2.2. Instancias de tareas y ejecución de gráficos

Las instancias de tareas son los elementos en una lista de tareas de un actor. Una instancia de tareas puede ser señalizar. Una instancia de tarea de señalización es una instancia de tarea que, cuando se finaliza, puede enviar una señal a su testigo para continuar la ejecución del proceso. Las instancias de tareas pueden ser de bloqueo, por lo no se permite que el testigo relacionado (=ruta de ejecución) deje el nódulo de la tarea antes que la instancia de tarea esté terminada. Por defecto, las instancias de tareas son de señalización y no-bloqueo.

En el caso que más de una instancia de tarea sea asociada a un nodo de tarea, el desarrollador del proceso puede especificar de qué manera la terminación de las instancias de tareas puede afectar la continuación del proceso. A continuación se encuentra una lista de los valores que pueden otorgarse a la propiedad de señalización de un nodo de tareas:

- **última**: Esto es por defecto. Procede a la ejecución cuando se completa la última instancia de tarea. Cuando no se crean tareas a la entrada de este nodo, continúa la ejecución.
- **Última espera**: Procede a la ejecución cuando se completa la última instancia de tarea. Cuando no se crean tareas a la entrada de este nodo, la ejecución espera en el nodo de tareas hasta que se creen tareas.
- **Primera**: Procede a la ejecución cuando se completa la primera instancia de tarea. Cuando no se crean tareas a la entrada de este nodo, continúa la ejecución.
- **Primera espera**: Procede a la ejecución cuando se completa la primera instancia de tarea. Cuando no se crean tareas a la entrada de este nodo, la ejecución espera en el nodo de tareas hasta que se creen tareas.
- **No sincronizado**: La ejecución siempre continúa, sin importar si se crean tareas o

si éstas aún no han terminado.
- **Nunca**: La ejecución nunca continúa, sin importar si se crean tareas o si éstas aún no han terminado.

La creación de una instancia de tarea se puede basar en un cálculo de tiempo de ejecución. En ese caso, agregue un `ActionHandler` en el evento `node-enter` del `task-node` y configure el atributo `create-tareas="false"`. A continuación hay un ejemplo del manejo de una acción de implementación:

```
public class Createtasks implements ActionHandler {
  public void execute(ExecutionContext executionContext) throws Exception {
    Token token = executionContext.getToken();
    TaskMgmtInstance tmi = executionContext.getTaskMgmtInstance();

    TaskNode taskNode = (TaskNode) executionContext.getNode();
    Task changeNappy = taskNode.getTask("change nappy");

    // Ahora, se han creado 2 instancias de tarea para la misma tarea.
    tmi.createTaskInstance(changeNappy, token);
    tmi.createTaskInstance(changeNappy, token);
  }
}
```

Tal como se muestra en el ejemplo, las tareas a crear se pueden especificar en el nodo de tareas. También se pueden especificar en `process-definition` y tomar de `TaskMgmtDefinition`. `TaskMgmtDefinition` lo que extiende el ProcessDefinition con la información sobre el manejo de tareas.

El métido API para marcar las instancias de tareas cuando son completadas es `TaskInstance.end()`. Opcionalmente, puede especificar una transición en el método de finalización. En caso que la finalización de esta instancia de tarea cause la continuación de la ejecución, el nodo de tareas se mantiene sobre la transición especificada.

11.3. Asignación

Una definición de proceso contiene nodos de tareas. Un `task-node` contiene cero o más tareas. Las tareas son una descripción estática que forma parte de la definición de proceso. Durante el tiempo de ejecución, las tareas ocasionan la creación de instancias de tarea. Una instancia de tarea corresponde a una entrada en la lista de tareas de una persona.

Con jBPM, el modelo de inserción y extracción (ver a continuación) de asignación de tareas se puede aplicar en combinación. El proceso puede calcular el responsable de una tarea e insertarla en su lista de tareas. O, si no, se puede asignar una tarea a un grupo de actores, en cuyo caso cada uno de los actores del grupo puede extraer la tarea y ponerla en la lista personal de tareas del actor.

11.3.1. Interfaces de asignación

La asignación de instancias de tarea se realiza a través de la interfaz `AssignmentHandler`:

```
public interface AssignmentHandler extends Serializable {
  void assign( Assignable assignable, ExecutionContext executionContext );
}
```

Al crear una instancia de tarea se llama a una implementación de controlador de asignación. En ese momento, la instancia de tarea se puede asignar a uno o más actores. La implementación de `AssignmentHandler` debe llamar a los métodos `Assignable` (`setActorId` o `setPooledActors`) para asignar una tarea. El `Assignable` es una `TaskInstance` o una `SwimlaneInstance` (=función de proceso).

```
public interface Assignable {
  public void setActorId(String actorId);
  public void setPooledActors(String[] pooledActors);
}
```

Se puede asignar tanto `TaskInstances` como `SwimlaneInstances` a un usuario específico o a un grupo de actores. Para asignar una `TaskInstance` a un usuario, llame a Assignable.setActorId(String actorId). Para asignar una `TaskInstance` a un grupo de actores candidatos, llame a `Assignable.setPooledActors(String[] actorIds)`.

Cada tarea de la definición de proceso se puede asociar con una implementación de controlador de asignación para realizar la asignación en el tiempo de ejecución.

Cuando se debe asignar más de una tarea de un proceso a la misma persona o grupo de actores, considere el uso de un carril.

Para permitir la creación de `AssignmentHandler` reutilizables, cada uso de `AssignmentHandler` se puede configurar en `processdefinition.xml`. Consulte la sección 16.2, "Delegación" para obtener mayor información acerca de como agregar configuración a los controladores de asignación.

11.3.2. El modelo de datos de asignación

El modelo de datos para administrar asignaciones de instancias de tarea e instancias de carril a actores es el siguiente. Cada `TaskInstance` tiene un actorId y un conjunto de actores.

Figura 11.1. Diagrama del modelo de asignación

El actorId es el responsable de llevar a cabo la tarea, mientras que el conjunto del pool de actores representa un conjunto de candidatos que podrían ser los responsables si toman la tarea. Tanto el actorId como los actores del pool son opcionales y también pueden combinarse.

11.3.3. Modelo de inserción

El actorId de una instancia de tarea indica al responsable de dicha tarea. Los actores

agrupados para una `TaskInstance` son los actores candidatos para dicha tarea. Por lo general, el actorId de una `TaskInstance` hará referencia a un usuario. Los actores agrupados pueden hacer referencia a usuarios y/o grupos.

La lista personal de tareas de un usuario son todas las `TaskInstance` que tiene el usuario especifico como a `actorId`. Esta lista se puede obtener con `TaskMgmtSession.findTaskInstances(String actorId)`.

11.3.4. Modelo de extracción

Por otra parte, las tareas de las tareas de grupo para un usuario determinado son las tareas para las cuales hay una referencia a ese usuario en los actores del pool. Buscar la lista de tareas de grupo consiste comúnmente en una operación que tiene dos pasos: 1) obtener todos los grupos para el usuario determinado desde un componente de identidad y 2) obtener la lista de todas las tareas grupales para el conjunto combinado del actorId del usuario y del actorId que hace referencia a los grupos de usuarios. La obtención de la lista de las tareas grupales que son ofrecidas a un usuario dado puede ser realizada con los métodos `TaskMgmtSession.findPooledTaskInstances(String actorId)` o `TaskMgmtSession.findPooledTaskInstances(List actorIds)`. Estos métodos sólo encontrarán las instancias de tareas para las cuales el actorId es `nulo` y uno de los actorId dados se corresponde con uno de los actores del grupo.

Para evitar que haya múltiples usuarios trabajando en la misma tarea agrupada, basta con actualizar el actorId de la `TaskInstance` con el actorId del usuario. Luego de hacerlo, la instancia de tarea no aparecerá en la lista de tareas agrupadas, sino que sólo lo hará en la lista personal de tareas del usuario. Configurar el actorId de una `taskInstance` en `null` volverá a poner la instancia de tarea en las tareas agrupadas.

11.4. Instancia de tareas variables

Una instancia de tarea puede tener su propio conjunto de variables. Además, una instancia de tarea puede 'ver' las variables de proceso. Las instancias de tarea generalmente se crean en una ruta de ejecución (=testigo). Esto crea una relación primario-secundario entre el testigo y las instancias de tarea similar a la relación primario-secundario entre los propios testigos. Las reglas normales de ámbito se aplican entre las variables de una instancia de tarea y las variables del testigo asociado. Se puede encontrar más información acerca del ámbito en la sección 10.4, "Alcances de las variables".

Esto significa que una instancia de tarea puede 'ver' sus propias variables más todas las variables de su testigo asociado.

El controlador se puede usar para crear, llenar y enviar variables entre el ámbito de instancias de tarea y las variables con ámbito de proceso.

11.5. Controladores de tareas

Durante la creación de una instancia de tarea, los controladores de tarea pueden llenar las variables de instancia de tarea y cuando la instancia de tarea finaliza, el controlador

de tarea puede enviar los datos de la instancia de tarea a las variables de proceso.

Se debe tener en cuenta que no es obligación usar controladores de tarea. Las instancias de tarea también pueden 'ver' las variables de proceso asociadas con su testigo. Se debe usar controladores de tarea cuando se desea:

- a) Crear copias de variables en las instancias de tarea de manera que las actualizaciones intermedias de las variables de instancia de tarea no afecten a las variables de proceso hasta que el proceso haya finalizado y las copias se envíen nuevamente a las variables de proceso.
- b) Que las variables de instancia de tarea no se asocien una a una con las variables de proceso. Por ejemplo, si el proceso tiene variables 'ventas en enero' 'ventas en febrero' y 'ventas en marzo'. Entonces, podría ser necesario que la forma para la instancia de tarea muestre el promedio de ventas de los 3 meses.

El propósito de las tareas es el recolectar información de los usuarios. Pero existen muchas interfaces de usuario que podrían usarse para presentar las tareas a los usuarios. Por ejemplo, una aplicación en la red, una aplicación swing, un mensaje instantáneo, un formato de correo electrónico.... De esa forma, los controladores de tareas hacen de puente entre las variables del proceso (=contexto del proceso) y la aplicación de la interfaz del usuario. Los controladores de tareas ofrecen una visión de las variables del proceso a la aplicación de la interfaz del usuario.

El controlador de tarea realiza la traducción (si la hay) de las variables de proceso a las variables de tarea. Cuando se crea una instancia de tarea, el controlador de tarea es responsable de extraer la información de las variables de proceso y de crear las variables de tarea. Las variables de tarea sirven como la entrada de la forma de la interfaz de usuario. Y la entrada del usuario se puede almacenar en las variables de tarea. Cuando el usuario finaliza la tarea, el controlador de tarea es responsable de actualizar las variables de proceso sobre la base de los datos de instancia de tarea.

Figura 11.2.Los controladores de tareas

En un escenario simple, existe una asignación uno a uno entre las variables de proceso y los parámetros de forma. Los controladores de tarea se especifican en un elemento de tarea. En este caso, se puede usar el controlador de tarea predeterminado de jBPM y asume una lista de elementos variable en su interior. Los elementos variable expresan la forma en que se copian las variables de proceso en las variables de tarea.

Enl ejemplo siguiente se muestra la forma en que se puede crear copias independientes de variables de instancia de tarea sobre la base de variables de proceso:

```
<task name="clean ceiling">
  <controller>
    <variable name="a" access="read" mapped-name="x" />
    <variable name="b" access="read,write,required" mapped-name="y" />
    <variable name="c" access="read,write" />
  </controller>
</task>
```

El atributo `name` hace referencia al nombre de la variable de proceso. El atributo `mapped-name` es opcional y hace referencia al nombre de la variable de instancia de

tarea. Si se omite el atributo mapped-name, éste asume el nombre en forma predeterminada. Se debe tener en cuenta que mapped-name también se usa como la etiqueta para los campos en la forma de instancia de tarea de la aplicación web.

El atributo `access` especifica que si la variable se copia durante la creación de la instancia de tarea, se escribirá nuevamente en las variables de proceso al finalizar la tarea, si se requiere. La interfaz de usuario puede usar esta información para generar los controles de forma adecuados. El atributo access es opcional y en forma predeterminada es 'read,write'.

Un `task-node` puede tener muchas tareas y un `start-state` puede tener una tarea.

Si la asignación simple uno a uno entre variables de proceso y parámetros de forma es demasiado limitante, también se puede escribir una implementación TaskControllerHandler propia. He aquí la interfaz TaskControllerHandler:

```
public interface TaskControllerHandler extends Serializable {
  void initializeTaskVariables(TaskInstance taskInstance, ContextInstance
contextInstance, Token token);
  void submitTaskVariables(TaskInstance taskInstance, ContextInstance
contextInstance, Token token);
}
```

Y ésta es la forma de configurar en una tarea la implementación personalizada de controlador de tarea:

```
<task name="clean ceiling">
  <controller class="com.yourcom.CleanCeilingTaskControllerHandler">
    -- here goes your task controller handler configuration --
  </controller>
</task>
```

11.6. Carriles

Un carril es una función de proceso. Se trata de un mecanismo para especificar que el mismo actor debe realizar múltiples tareas del proceso. Así, luego de crear la primera instancia de tarea para un carril dado, el actor debe recordarse en el proceso para todas las tareas posteriores que estén en el mismo carril. Por lo tanto, un carril tiene una assignment y todas las tareas que hacen referencia a un carril no deben especificar una assignment.

Cuando se crea la primera tarea en un carril dado, se llama al `AssignmentHandler` del carril. El `Assignable` que se pase al `AssignmentHandler` será la `SwimlaneInstance`. Es importante saber que todas las asignaciones que se realizan en las instancias de tarea de un carril dado se propagarán a la instancia de carril. Este comportamiento se implemente como valor predeterminado debido a que la persona que asume una tarea para cumplir una determinada función de proceso tendrá conocimiento de dicho proceso particular. De esta manera, todas las asignaciones posteriores de instancias de tarea a dicho carril se realizan automáticamente a ese usuario.

Carril (Swimlane, en inglés) es una terminología adoptada de los diagramas de actividad UML.

11.7. Carril en una tarea de inicio

Un carril se puede asociar con la tarea de inicio para capturar al iniciador del proceso.

Cuando se crea una nueva instancia de tarea para dicha tarea, el actor autenticado actual será capturado con Authentication.getAuthenticatedActorId() y ese actor se almacenará en el carril de la tarea de inicio.

Por ejemplo:

```
<process-definition>
  <swimlane name='initiator' />
  <start-state>
    <task swimlane='initiator' />
    <transition to='...' />
  </start-state>
  ...
</process-definition>
```

Además, se puede agregar variables a la tarea inicial como con cualquier otra tarea para definir la forma asociada con la tarea. Consulte la sección 11.5, "Controladores de tareas"

11.8. Eventos de las tareas

as tareas pueden tener acciones asociadas. Hay 4 tipos estándar de eventos definidos para tareas: `task-create`, `task-assign`, `task-start` y `task-end`.

`task-create` se activa cuando se crea una instancia de tarea.

`task-assign` se activa cuando se está asignando una instancia de tarea. Se debe tener en cuenta que en las acciones ejecutadas en este evento se puede acceder al actor previo con `executionContext.getTaskInstance().getPreviousActorId();`

`task-start` se activa cuando se llama `TaskInstance.start()`. Esto se puede usar para indicar que el usuario está realmente comenzando a trabajar en esta instancia de tarea. Iniciar una tarea es opcional.

`task-end` se activa cuando se llama `TaskInstance.end(...)`. Esto marca la finalización de la tarea. Si la tarea se asocia con la ejecución de un proceso, esta llamada podría desencadenar la reanudación de la ejecución del proceso.

Debido a que las tareas pueden tener asociados eventos y acciones, también se puede especificar controladores de excepción en una tarea. Para obtener más información acerca de controladores de excepción, consulte la sección 9.7, "Manejo de excepciones".

11.9. Temporizadores de tareas

Al igual que con los nodos, se puede especificar temporizadores en tareas. Consutle la sección 12.1, "Temporizadores".

Lo que hay de especial acerca de los temporizadores para tareas es que se puede

personalizar el `cancel-event` para temporizadores de tarea. En forma predeterminada, un temporizador en una tarea se cancelará cuando la tarea finalice (=finalizada). Pero con el atributo `cancel-event` en el temporizador, los desarrolladores de procesos pueden personalizar, por ejemplo, `task-assign` o `task-start`. El `cancel-event` es compatible con varios eventos. Los tipos de `cancel-event` se pueden combinar, especificándolos en el atributo con una lista separada por comas.

11.10. Personalización de instancias de tareas

Las instancias de tareas se pueden personalizar. La forma más sencilla de realizarlo es crear una subclase de `TaskInstance`. Luego, se crea una implementación `org.jbpm.taskmgmt.TaskInstanceFactory` y se configura estableciendo la propiedad de configuración `jbpm.task.instance.factory` enel nombre de clase completo en el jbpm.cfg.xml. Si se usa una subclase de TaskInstance, se debe crear también un archivo de asignación de hibernación para la subclase (usando la hibernación `extends="org.jbpm.taskmgmt.exe.TaskInstance"`). Luego agregue ese archivo de asignación a la lista de archivos de asignaciones en `hibernate.cfg.xml`

11.11. El componente de identidad

La administración de usuarios, grupos y permisos se conoce comúnmente como administración de identidades. jBPM incluye un componente opcional de identidad que se puede reemplazar fácilmente con un almacén de datos de identidad de la propia compañía.

El componente de administración de identidades de jBPM incluye conocimiento del modelo organizacional. La asignación de tarea se realiza por lo general con conocimiento organizacional. Así que esto implica conocimiento de un modelo organizacional, la descripción de los usuarios, grupos y sistemas, y las relaciones entre ellos. Opcionalmente, también se puede incluir permisos y funciones en un modelo organizacional. Diversos intentos de investigación académica han fallado, lo que demuestra que no se puede crear un modelo organizacional genérico que sirva para toda organización.

La forma en que jBPM maneja esto es definiendo un actor como un participante real en un proceso. Un actor se identifica mediante su ID llamado actorId. jBPM sólo tiene conocimiento acerca de actorIds y estos se representan como `java.lang.String` para flexibilidad máxima. De manera que cualquier conocimiento acerca del modelo organizacional y la estructura de esos datos está fuera del ámbito del motor del núcleo de jBPM.

Como extensión de jBPM se proveerá (en el futuro) un componente para administrar dicho modelo simple de usuario-funciones. Esta relación de muchos a muchos entre usuarios y funciones es el mismo modelo definido en las especificaciones J2EE y servlet, y puede servir como un punto de partida para nuevos desarrollos. Las personas interesadas en contribuir deben inspeccionar el rastreador de temas jboss jbpm jira para obtener más detalles.

Tenga en cuenta que el modelo de usuario-funciones, tal como se usa en las especificaciones de servlet, ejb y portlet, no es suficientemente potente para manejar asignaciones de tarea. Dicho modelo es una relación de muchos a muchos entre usuarios y funciones. Esto no incluye información acerca de equipos y la estructura organizacional de usuarios que participan en un proceso.

11.11.1. El modelo de identidad

Figura 11.3. El diagrama de clases de modelo de identidad

Las clases en amarillo son las clases relevantes para el controlador de asignación de expresión que se analiza a continuación.

Un `User` representa un usuario o un servicio. Un `Group` es cualquier tipo de grupo de usuarios. Los grupos pueden anidarse para modelar la relación entre un equipo, una unidad de negocios y la compañía completa. Los grupos tienen un tipo para diferenciar entre los grupos jerárquicos y, por ejemplo, los grupos de color de cabello. `Memberships` representan la relación de muchos a muchos entre usuarios y grupos. Se puede representar una pertenencia para representar un puesto en una compañía. El nombre de la pertenencia se puede usar para indicar la función que el usuario cumple en el grupo.

11.11.2. Expresiones de asignación

El componente de identidad viene con una implementación que evalúa una expresión para el cálculo de actores durante la asignación de tareas. He aquí un ejemplo de cómo usar la expresión de asignación en una definición de proceso:

```
<process-definition>
  ...
  <task-node name='a'>
    <task name='laundry'>
      <assignment expression='previous --> group(hierarchy) --> member(boss)' />
    </task>
    <transition to='b' />
  </task-node>
  ...
```

La sintaxis de la expresión de asignación se define:

```
first-term --> next-term --> next-term --> ... --> next-term

where

first-term ::= previous |
               swimlane(swimlane-name) |
               variable(variable-name) |
               user(user-name) |
               group(group-name)

and

next-term ::= group(group-type) |
```

```
member(role-name)
```

11.11.2.1. Primeros términos

Una expresión se resuelve de izquierda a derecha. El primer término representa un `User` o `Group` en el modelo de identidad. Los términos posteriores calculan el siguiente término del usuario o del grupo intermedio.

`previous` significa que la tarea se asigna al actor autenticado actual. Esto significa, el actor que realizó el paso previo en el proceso.

`swimlane(swimlane-name)` significa que el usuario o el grupo se toma de la instancia de carril especificada.

`variable(variable-name)` significa que el usuario o grupo se toma de la instancia de variable especificada. La instancia de variable puede contener un `java.lang.String`, en cuyo caso dicho usuario o grupo se busca en el componente de identidad. O la instancia de variable contiene un objeto de `User` o `Group`.

`user(user-name)` significa que el usuario dado se toma del componente de identidad.

`group(group-name)` significa que el grupo dado se toma del componente de identidad.

11.11.2.2. Términos siguientes

`group(group-type)` obtiene el grupo para un usuario. Significa que los términos previos deben dar como resultado un User. Busca el grupo con el tipo de grupo dado en todas las pertenencias del usuario.

`member(role-name)` obtiene el usuario que realiza una función dada para un grupo. Significa que los términos previos deben dar como resultado un `Group`. Este término busca el usuario con una pertenencia al grupo para el cual el nombre de la pertenencia coincide con el role-name dado.

11.11.3. Eliminación del componente de identidad

Si se quiere usar un origen propio de datos para información organizacional, como la base de datos de usuarios o el sistema ldap de la compañía, simplemente se quita el componente de identidad de jBPM. Lo único que se necesita hacer es asegurarse que se borre la línea ...

```
<mapping resource="org/jbpm/identity/User.hbm.xml"/>
<mapping resource="org/jbpm/identity/Group.hbm.xml"/>
<mapping resource="org/jbpm/identity/Membership.hbm.xml"/>
```

de `hibernate.cfg.xml`

El `ExpressionAssignmentHandler` depende del componente de identidad, por lo que no puede usarse tal como está. En el caso que quiera usar nuevamente el `ExpressionAssignmentHandler` y unirlo al almacén de datos de los usuarios, se puede dampliar del `ExpressionAssignmentHandler` y anular el método `getExpressionSession`.

```
protected ExpressionSession getExpressionSession(AssignmentContext
assignmentContext);
```

Capítulo 12. Planificador

En este capítulo se describe cómo trabajar con temporizadores en jBPM.

Tras eventos del proceso, se puede crear temporizadores. Cuando un temporizador vence, se puede ejecutar una acción o se puede realizar una transición.

12.1. Temporizadores

La forma más sencilla de especificar un temporizador es agregar un elemento temporizador al nodo.

```
<state name='catch crooks'>
  <timer name='reminder'
         duedate='3 business hours'
         repeat='10 business minutes'
         transition='time-out-transition' >
    <action class='the-remainder-action-class-name' />
  </timer>
</state>
```

Si se especifica un temporizador en un nodo, éste no se ejecuta después de que se deja el nodo. Tanto la transición como la acción son opcionales. Cuando se ejecuta un temporizador, se producen los siguientes eventos en secuencia:

- se activa un evento de tipo `timer`
- si se especifica una acción, esa acción se ejecuta.
- Si se especifica una transición, se envia una señal para reanudar la acción durante la transición especificada.

Cada temporizador debe tener un nombre único. Si no se especifica un nombre en el elemento `timer`, se asume el nombre del nodo como nombre del temporizador.

La acción del temporizador puede ser cualquier elemento de acción compatible, como por ejemplo `action` o `script`.

Las acciones crean y cancelan los temporizadores. Los dos elementos de acción son `create-timer` y `cancel-timer`. En realidad, el elemento temporizador que aparece anteriormente es sólo una notación corta de una acción crear-temporizador en `node-enter` y una acción cancelar-temporizador en `node-leave`.

12.2. Implementación del programador

Las ejecuciones de proceso crean y cancelan temporizadores. Los temporizadores se almacenan en un almacén de temporizadores. Un ejecutor de temporizadores independiente debe comprobar el almacén de temporizadores y ejecutar los temporizadores cuando corresponda.

Figura 12.1. Componentes generales del plrogramador

En el siguiente diagrama de clases se muestran las clases que participan en la

implementación del programador. Las interfaces `SchedulerService` y `TimerExecutor` se especifican para que el mecanismo de ejecución de temporizadores sea conectable.

Figura 12.2. Información general de componentes de programador

Capítulo 13. Continuaciones asíncronas

13.1. El concepto

jBPM se basa en Programación Orientada a Objetos (GOP). Básicamente, GOP especifica una máquina de estado simple que puede manejar rutas de ejecución concurrentes. Pero en el algoritmo de ejecución especificado en GOP, todas las transiciones de estado se realizan en una sola operación en el subproceso del cliente. Si no está familiarizado con el algoritmo de ejecución definido en el capítulo 4, Programación Orientada a Objetos, léalo primero. En forma predeterminada, esta realización de transiciones de estado en el subproceso del cliente es un buen enfoque puestos que se ajusta naturalmente a las transacciones del lado del servidor. La ejecución de proceso se mueve de un estado de espera a otro en una transacción.

Pero en algunas situaciones, un programador podría querer afinar la demarcación de transacción en la definición de proceso. En jPDL, es posible especificar que la ejecución de proceso debe continuar en forma asíncrona con el atributo `async="true"`. `async="true"` se puede especificar en todos los tipos de nodo y todos los tipos de acción.

13.2. Un ejemplo

Normalmente, un nodo siempre se ejecuta después de que un testigo entra en él. Así, el nodo se ejecuta en el subproceso del cliente. Los dos ejemplos siguientes exploran las continuaciones asíncronas. El primer ejemplo es parte de un proceso con tres nodos. El nodo 'a' es un estado de espera, el nodo 'b' es un paso automatizado y el nodo 'c' es nuevamente un estado espera. Este proceso no contiene ningún comportamiento asíncrono y se representa en la imagen a continuación.

En el primer cuadro se muestra la situación inicial. El testigo apunta al nodo 'a', lo que significa que la ruta de ejecución espera un desencadenador externo. Dicho desencadenador debe darse mediante el envío de una señal al testigo. Al llegar la señal, el testigo pasará del nodo 'a' a través de la transición al nodo 'b'. Después de que el testigo llega al nodo 'b', el nodo 'b' se ejecuta. Se debe recordar que el nodo 'b' es un paso automatizado que no se comporta como un estado de espera (por ejemplo, enviando un correo electrónico). Así que el segundo cuadro es una instantánea obtenida cuando el nodo 'b' se ejecuta. Puesto que el nodo 'b' es un paso automatizado en el proceso, la ejecución del nodo 'b' incluirá la propagación del testigo a través de la transición al nodo 'c'. El nodo 'c' es un estado de espera, de manera que el tercer cuadro muestra la situación final después de que vuelve el método de señal.

Figura 13.1. Ejemplo 1: Proceso sin continuación asíncrona

Si bien la persistencia no es obligatoria en jBPM, el escenario más común es que se llame a una señal dentro de una transacción. Ésta es una mirada a las actualizaciones de dicha transacción. En primer lugar, el testigo se actualiza para apuntar al nodo 'c'. Estas actualizaciones se generan por hibernación como resultado de la

`GraphSession.saveProcessInstance` en una conexión de JDBC. En segundo lugar, en caso de que la acción automatizada accediera y actualizara algunos recursos transaccionales, dichas actualizaciones transaccionales deben combinarse con la misma transacción o ser parte de ella.

Ahora, el segundo ejemplo que es una variante del primero y presenta una continuación asíncrona en el nodo 'b'. Los nodos 'a' y 'c' se comportan igual que en el primer ejemplo, es decir que se comportan como estados de espera. En jPDL, un nodo se marca como asíncrono configurando el atributo `async="true"`.

El resultado de agregar `async="true"` al nodo 'b' es que la ejecución de proceso se dividirá en dos partes. La primera parte ejecutará el proceso hasta el punto donde se ejecutará el nodo 'b'. La segunda parte ejecutará el nodo 'b' y dicha ejecución se detendrá en el estado de espera 'c'.

Por lo tanto, la transacción se dividirá en dos transacciones independientes. Una transacción para cada parte. Aunque se requiere un desencadenador externo (la invocación del método `Token.signal`) para dejar el nodo 'a' en la primera transacción, jBPM desencadenará y realizará automáticamente la segunda transacción.

Figura 13.2. Ejemplo 2: Un proceso con continuaciones asíncronas

Para acciones, el principio es similar. Las acciones marcadas con el atributo `async="true"` se ejecutan fuera del subproceso que ejecuta el proceso. Si se configura la persistencia (lo está en forma predeterminada), las acciones ejecutarán en una transacción independiente.

En jBPM, las continuaciones asíncronas se realizan usando un sistema de mensajería asíncrona. Cuando la ejecución de proceso llega al punto en que se debe ejecutar en forma asíncrona, jBPM suspenderá la ejecución, producirá un mensaje de comando y lo enviará al ejecutor de comando. El ejecutor de comando es un componente independiente que, al recibir el mensaje, reanudará la ejecución del proceso donde se suspendió.

jBPM se puede configurar para usar un proveedor de JMS o su sistema de mensajería asíncrona incorporada. El sistema de mensajería incorporada es de funcionalidad bastante limitada, pero permite la compatibilidad con esta característica en entornos donde JMS no está disponible.

13.3. El ejecutor de comandos

El ejecutor de comandos es el componente que reanuda las ejecuciones de proceso en forma asíncrona. Éste espera la llegada de mensajes de comando a través de un sistema de mensajería asíncrona y los ejecuta. Los dos comandos que se usan para continuaciones asíncronas son `ExecuteNodeCommand` y `ExecuteActionCommand`.

El proceso de ejecución produce estos comandos. Durante la ejecución de proceso se creará un `ExecuteNodeCommand` (POJO) en la `MessageInstance`, para cada nodo que deba ejecutarse en forma asíncrona. La instancia de mensaje es una extensión no persistente de la ProcessInstance y simplemente recopila todos los mensajes que deben enviarse.

Los mensajes se enviarán como parte de la `GraphSession.saveProcessInstance`. La implementación de dicho método incluye un generador de contexto que actúa como un aspecto en el método saveProcessInstance. Los interceptores reales se pueden configurar en el `jbpm.cfg.xml`. Uno de los interceptores, `SendMessagesInterceptor`, se configura en forma predeterminada y leerá los mensajes desde la `MessageInstance` y los enviará a través del sistema de mensajería asíncrona configurable.

`SendMessagesInterceptor` usa las interfaces `MessageServiceFactory` y `MessageService` para enviar mensajes. Esto es para que la implementación de la mensajería asíncrona sea configurable (también en `jbpm.cfg.xml`).

13.4. Mensajería asíncrona incorporada de jBPM

Al usar la mensajería asíncrona incorporada de jBPM, los mensajes se enviarán haciéndolos persistir en la base de datos. Esta persistencia de mensaje se puede realizar en la misma transacción/conexión de jdbc que las actualizaciones de proceso de jBPM.

Los mensajes de comando se almacenarán en la tabla JBPM_MESSAGE.

El ejecutor de comandos POJO (`org.jbpm.msg.command.CommandExecutor`) leerá los mensajes desde la base de datos y los ejecutará. Así, la típica transacción del ejecutor de comando POJO tiene el siguiente aspecto: 1) leer siguiente mensaje de comando 2) ejecutar mensaje de comando 3) borrar mensaje de comando.

Si la ejecución de un mensaje de comando falla, la transacción volverá a su estado anterior. Luego, se iniciará una nueva transacción que agrega el mensaje de error al mensaje en la base de datos. El ejecutor de comandos filtra todos los mensajes que contienen una excepción.

Figura 13.3. Transacciones del ejecutor de comandos POJO

Si por una u otra razón, fallare la transacción que agrega la excepción al mensaje de comando, ésta también vuelve a su estado anterior. En ese caso, el mensaje permanece en la cola sin una excepción de manera que se reintente después.

Limitación: Se debe tener cuidado pues el sistema de mensajería asíncrona incorporada de jBPM no es compatible con el bloqueo multinodo. Así que no se puede simplemente implementar el ejecutor de comando POJO en múltiples instancias y tenerlas configuradas para usar la misma base de datos.

13.5. JMS para arquitecturas asíncronas

La característica de continuaciones asíncronas abre un nuevo mundo de escenarios de uso de jBPM. Allí donde jBPM se usa habitualmente para modelar procesos empresariales, ahora puede usarse desde una perspectiva más técnica.

Basta con imaginar que se tiene una aplicación con unos cuantos procesamientos asíncronos. Eso por lo general requiere bastante esfuerzo si se dificulta configurar el enlace de todo el software de producción y consumo de mensajes en forma conjunta.

Gracias a jBPM ahora es posible crear una imagen de la arquitectura asíncrona general, tener todo el código en POJO y agregar demarcación de transacción en el archivo de proceso general. jBPM se encargará de enlazar los emisores a los receptores sin necesidad de escribir todo el código JMS o MDB por cuenta propia.

13.6. JMS para mensajería asíncrona

Tarea pendiente (se se ha implementado aún)

13.7. Direcciones futuras

Tareas pendientes: Agregar compatibilidad para múltiples colas. Así se hará posible especificar una cola para cada nodo o acción que esté marcado como asíncrono. Además, sería ideal producir mensajes para un conjunto de colas en una operación por rondas. Puesto que todo esto debe ser configurable tanto para JMS como para los sistemas de mensajería incorporada, se requerirá algo de reflexión acerca de cómo realizar todas estas configuraciones. Las definiciones de proceso no deben depender de ninguna de las dos implementaciones posibles.

Capítulo 14. Calendario hábil

Este capítulo describe el calendario hábil de jBPM. El calendario hábil conoce las horas hábiles y se usa en el cálculo de fechas de vencimiento para tareas y temporizadores.

El calendario hábil puede calcular una fecha mediante la adición de una duración y una fecha.

14.1. Duración

Una duración se especifica en horas absolutas o hábiles. He aquí la sintaxis:

```
<quantity> [business] <unit>
```

Donde <quantity> es un texto analizable con Double.parseDouble(quantity). <unit> es uno de {second, seconds, minute, minutes, hour, hours, day, days, week, weeks, month, months, year, years}. Y agregando la indicación opcional business significa que sólo se considerarán las horas hábiles para esta duración. Sin la indicación business, la duración será interpretada como un período de tiempo absoluto.

14.2. Configuración del calendario

El archivo org/jbpm/calendar/jbpm.business.calendar.properties especifica lo que son las horas hábiles. El archivo de configuración se puede personalizar y se puede poner una copia modificada en la raíz de la ruta de clases.

Este es el ejemplo de especificación de horas hábiles incluido en forma predeterminada en jbpm.business.calendar.properties:

```
hour.format=HH:mm
#weekday ::= [<daypart> [& <daypart>]*]
#daypart ::= <start-hour>-<to-hour>
#start-hour and to-hour must be in the hour.format
#dayparts have to be ordered
weekday.monday=    9:00-12:00 & 12:30-17:00
weekday.thuesday=  9:00-12:00 & 12:30-17:00
weekday.wednesday= 9:00-12:00 & 12:30-17:00
weekday.thursday=  9:00-12:00 & 12:30-17:00
weekday.friday=    9:00-12:00 & 12:30-17:00
weekday.saturday=
weekday.sunday=

day.format=dd/MM/yyyy
# holiday syntax: <holiday>
# holiday period syntax: <start-day>-<end-day>
# below are the belgian official holidays
holiday.1=  01/01/2005 # nieuwjaar
holiday.2=  27/3/2005  # pasen
holiday.3=  28/3/2005  # paasmaandag
holiday.4=  1/5/2005   # feest van de arbeid
holiday.5=  5/5/2005   # hemelvaart
holiday.6=  15/5/2005  # pinksteren
holiday.7=  16/5/2005  # pinkstermaandag
holiday.8=  21/7/2005  # my birthday
holiday.9=  15/8/2005  # moederkesdag
holiday.10= 1/11/2005  # allerheiligen
```

```
holiday.11= 11/11/2005 # wapenstilstand
holiday.12= 25/12/2005 # kerstmis

business.day.expressed.in.hours=              8
business.week.expressed.in.hours=            40
business.month.expressed.in.business.days=   21
business.year.expressed.in.business.days=   220
```


Capítulo 15. Registros

El propósito de los registros es seguir la pista del historial de una ejecución de proceso. Al cambiar los datos de tiempo de ejecución de una ejecución de proceso, todas las diferencias se almacenan en los registros.

Los registros de proceso, abarcados en este capítulo, no deben confundirse con los registros de software. Los registros de software siguen la pista de la ejecución de un programa de software (usualmente para fines de depuración). Los registros de proceso siguen la pista de la ejecución de instancias de proceso.

Existen diversos casos de uso para información de registros de proceso. El más obvio es la consulta del historial de proceso por parte de participantes de una ejecución de proceso.

Otro caso de uso es la Supervisión de Actividad Empresarial (BAM). La BAM consultará o analizará los registros de ejecuciones de proceso para encontrar información estadística útil acerca del proceso empresarial. Por ejemplo, ¿cuánto tiempo promedio se destina a cada paso del proceso?, ¿cuáles con los cuellos de botella del proceso?, etc. Esta información es clave para implementar una verdadera administración de proceso empresarial en una organización. La verdadera administración de proceso empresarial consiste en la forma en que una organización administra sus procesos, la forma en que estos están respaldados por tecnología de la información *y* la forma en que éstas se mejoran mutuamente en un proceso iterativo.

El siguiente caso de uso es la funcionalidad deshacer. Los registros de proceso se pueden usar para implementar la característica deshacer. Puesto que los registros contienen las diferencias de la información de tiempo de ejecución, estos pueden reproducirse en orden inverso para llevar el proceso nuevamente a un estado previo.

15.1. Creación de registros

Los módulos de jBPM producen registros durante las ejecuciones de proceso. Pero los usuarios también pueden insertar registros de proceso. Una entrada de registro es un objeto java que se hereda de `org.jbpm.logging.log.ProcessLog`. Las entradas de registro de proceso se agregan a la `LoggingInstance`. La `LoggingInstance` es una extensión opcional de `ProcessInstance`.

jBPM genera diversos tipos de registros: Registros de ejecución de gráficos, registros de contexto y registros de administración de tareas. Para obtener más información acerca de los datos específicos contenidos en dichos registros, se debe consultar los javadocs. Un buen punto de partida es la clase `org.jbpm.logging.log.ProcessLog` ya que desde esa clase se puede desplazar hacia abajo por el árbol de herencia.

La `LoggingInstance` recopila todas las entradas de registro. Cuando se guarda la `ProcessInstance`, todos los registros en la `LoggingInstance` se vaciarán en la base de datos. El campo `logs` de una `ProcessInstance` no tiene asignado hibernación para evitar que los registros se recuperen desde la base de datos en cada transacción. Cada `ProcessLog` se realiza en el contexto de una ruta de ejecución (`Token`) y, por lo tanto, el `ProcessLog` hace referencia a dicho testigo. El Token sirve también como un generador de secuencia de índice para el índice del `ProcessLog` en el `Token`. Esto

será importante para la recuperación de registro. De esta manera, los registros que se producen en transacciones posteriores tendrán números de secuencia secuenciales. (vaya, muchas 'sec' :-s).

Para implementaciones donde los registros no son importantes, basta con eliminar la `LoggingDefinition` opcional en la `ProcessDefinition`. Eso evitará que se capturen las `LoggingInstance`s instancias y no se actualizará ningún registro. Más adelante, se agregará un control de configuración más detallado sobre los registros. Consulte el tema jira 'configuración a nivel de registro'.

El método API para agregar registros de proceso es el siguiente.

```
public class LoggingInstance extends ModuleInstance {
  ...
  public void addLog(ProcessLog processLog) {...}
  ...
}
```

El diagrama UML para información de registros tiene este aspecto:

Figura 15.1. Diagrama de clases de información de registros de jBPM

Un `CompositeLog` es un tipo especial de entrada de registro. Sirve como un registro principal para una serie de registros secundarios, creando así los medios para una estructura jerárquica en los registros. La API para insertar un registro es la siguiente.

```
public class LoggingInstance extends ModuleInstance {
  ...
  public void startCompositeLog(CompositeLog compositeLog) {...}
  public void endCompositeLog() {...}
  ...
}
```

Los `CompositeLog`s siempre deben llamarse en un bloque `try-finally` para asegurar que la estructura jerárquica de registros sea coherente. Por ejemplo:

```
startCompositeLog(new MyCompositeLog());
try {
  ...
} finally {
  endCompositeLog();
}
```

15.2. Recuperacaión de registros

Como ya se mencionó, los registros no se pueden recuperar desde la base de datos desplazando la LoggingInstance a sus registros. En cambio, los registros de una instancia de proceso siempre deben consultarse desde la base de datos. La `LoggingSession` tiene 2 métodos que se pueden utilizar para esto.

El primer método recupera todos los registros de una instancia de proceso. Estos registros se agrupan por testigo en una asignación. La asignación asociará una lista de

ProcessLogs con cada testigo de la instancia de proceso. La lista contendrá ProcessLogs en el mismo orden que se crearon.

```
public class LoggingSession {
  ...
  public Map findLogsByProcessInstance(long processInstanceId) {...}
  ...
}
```

El segundo método recupera los registros de un testigo específico. La lista devuelta contiene ProcessLogs en el mismo orden que se crearon.

```
public class LoggingSession {
  public List findLogsByToken(long tokenId) {...}
  ...
}
```

15.3. Almacenamiento de base de datos

En ocasiones se quiere aplicar técnicas de almacenamiento de datos a los registros de proceso de jbpm. El almacenamiento de datos significa que se crea una base datos independiente que contiene los registros de proceso que se usarán para diversos propósitos.

Es posible que haya muchas razones para crear un almacén de datos con la información de registro de proceso. A veces `podría ser para librarse de consultas pesadas desde la base de datos de producción 'en vivo'. En otras situaciones podría servir para realizar algunos análisis exhaustivos. El almacenamiento de datos incluso podría realizarse en un diagrama de base de datos modificada que se optimiza para este propósito.

Esta sección tenía como intención proponer la técnica de almacenamiento en el contexto de jBPM. Los propósitos son tan diversos que impiden que se incluya una solución genérica en jBPM capaz de abarcar todos esos requisitos.

Capítulo 16. Lenguaje de Definición de Proceso de jBPM (JPDL)

JPDL especifica un esquema xml y el mecanismo para empaquetar todos los archivos asociados con la definición de proceso en un archivo de proceso.

16.1. El archivo de proceso

Un archivo de proceso es un archivo zip. El archivo central en el archivo de proceso es `processdefinition.xml`. La información principal en dicho archivo es el gráfico de proceso. El `processdefinition.xml` contiene además información acerca de acciones y tareas. Un archivo de proceso puede contener también archivos asociados como clases, formas de ui para tareas, etc, ...

16.1.1. Implementación de un archivo de proceso

La implementación de archivos de proceso se puede realizar de tres formas: Con la herramienta de diseñador de proceso, con una tarea task o en forma programada.

La implementación de un archivo de proceso con la herramienta de diseñador aún está en construcción.

La implementación de un archivo de proceso con una tarea ant se puede realizar de la siguiente forma:

```
<target name="deploy.par">
  <taskdef name="deploypar" classname="org.jbpm.ant.DeployParTask">
    <classpath --make sure the jbpm-[version].jar is in this classpath--/>
  </taskdef>
  <deploypar par="build/myprocess.par" />
</target>
```

Para implementar más archivos de proceso simultáneamente, use los elementos de conjunto de archivos anidados. El atributo de archivo en sí es opcional. Otros atributos de la tarea ant son:

- **cfg**: es opcional, el valor predeterminado es 'hibernate.cfg.xml'. El archivo de configuración de hibernación que contiene las propiedades de conexión jd-bc a la base de datos y los archivos de asignación.
- **properties**: es opcional y sobrescribe *todas* las propiedades de hibernación tal como se encuentran en el hibernate.cfg.xml.
- **createschema**: si se configura en verdadero, el esquema de base de datos de jbpm se crea antes de que se implementen los procesos.

Los archivos de proceso también se pueden implementar en forma programada con la clase `org.jbpm.jpdl.par.ProcessArchiveDeployer`

16.1.2. Versiones de proceso

Las definiciones de proceso nunca deben cambiar debido a que es extremadamente difícil (si no, imposible) predecir todos los posibles efectos secundarios de los cambios de definición de proceso.

Para sortear este problema, jBPM tiene un sofisticado mecanismo de versiones de proceso. El mecanismo de versiones permite que coexistan en la base de datos múltiples definiciones de proceso con el mismo nombre. Una instancia de proceso se puede iniciar en la versión más reciente disponible en ese momento y se mantendrá en ejecución en esa misma definición de proceso durante su duración completa. Cuando se implementa una versión más nueva, las instancias de creación reciente se iniciarán en la versión más reciente, en tanto que las instancias de proceso más antiguas se mantendrán en ejecución en las definiciones de proceso más antiguas.

Las definiciones de proceso son una combinación de un gráfico de proceso especificado en forma declarativa y opcionalmente, un conjunto de clases java asociadas. Las clases java pueden ponerse a disposición del entorno de tiempo de ejecución de jBPM de dos formas: asegurando que estas clases sean visibles para el cargador de clase de jBPM. Normalmente, esto significa que se puede poner las clases de delegación en un archivo `.jar` junto al `jbpm-[version].jar`. Las clases java deben incluirse además en el archivo de proceso. Cuando se incluye las clases de delegación en el archivo de proceso (y éstas no son visibles para el cargador de clase de jbpm), jBPM también aplicará versiones en estas clases. Se puede encontrar más información acerca de la carga de clase de proceso en la sección 16.2, "Delegación"

Cuando se implementa un archivo de proceso, éste crea una definición de proceso en la base de datos de jBPM. Las definiciones de proceso se pueden versionar sobre la base del nombre de definición de proceso. Cuando se implementa un archivo de proceso con nombre, el implementador asignará un número de versión. Para asignar este número, el implementador buscará el número de versión más alto para definiciones de proceso con el mismo nombre y le sumará 1. Las definiciones de proceso sin nombre siempre tendrán el número de versión -1.

16.1.3. Cambio de definiciones de proceso implementadas

El cambio de definiciones de proceso después de que se han implementado en la base de datos de jBPM reviste muchas posibles dificultades. Por lo tanto, se recomienda enfáticamente no hacerlo.

En realidad, existe una gama completa de cambios posibles que se pueden realizar en una definición de proceso. Algunas de esas definiciones de proceso son inocuas, pero otros cambios tienen consecuencias que van más allá de lo esperado y deseable.

Se debe considerar las instancias de proceso de migración El cambio de definiciones de proceso después de que se han implementado en la base de datos de jBPM reviste muchas posibles dificultades. Por lo tanto, se recomienda enfáticamente no hacerlo.

En realidad, existe una gama completa de cambios posibles que se pueden realizar en una definición de proceso. Algunas de esas definiciones de proceso son inocuas, pero otros cambios tienen consecuencias que van más allá de lo esperado y deseable.

Se debe considerar la `JbpmContext.getSession()`.

La caché de segundo nivel: Sería necesario eliminar una definición de proceso desde la caché de segundo nivel después de actualizar una definición de proceso existente.

Consulte también la sección 7.10, "Caché de segundo nivel"

16.1.4. Migración de instancias de proceso

Un enfoque alternativo al cambio de definiciones de proceso sería convertir las ejecuciones en una nueva definición de proceso. Se debe tomar en cuenta que esto no es trivial debido a la naturaleza de larga vida de los procesos empresariales. Actualmente, se trata de un área experimental para la que no hay mucha compatibilidad inmediata aún.

Como es sabido, existe una clara diferencia entre datos de definición de proceso, datos de instancia de proceso (los datos de tiempo de ejecución) y los datos de registros. Con este enfoque, se crea una nueva e independiente definición de proceso en la base de datos de jBPM (mediante, por ejemplo, la implementación de una nueva versión del mismo proceso). Luego, la información de tiempo de ejecución se convierte a la nueva definición de proceso. Esto podría implicar una traducción debido a que los testigos del proceso antiguo podrían estar apuntando a nodos que se han eliminado en la nueva versión. Así, sólo se crean datos nuevos en la base de datos. Pero una ejecución de un proceso se extiende sobre dos objetos de instancia de proceso. Esto podría resultar un poco complicado para las herramientas y los cálculos estadísticos. Cuando los recursos lo permitan, se agregará compatibilidad para esto en el futuro. Por ejemplo, se puede agregar un indicador de una instancia de proceso a su predecesora.

16.1.5. Conversión de proceso

Hay disponible una clase de conversión para ayudar a convertir los archivos de proceso de jBPM 2.0 a archivos de proceso compatibles con jBPM 3.0. Se debe crear un directorio de salida para guardar los archivos de proceso convertidos. Escriba la siguiente línea de comandos desde el directorio de compilación de la distribución de jBPM 3.0:

java -jar converter.jar indirectory outdirectory

Sustituya "indirectory" por el directorio de entrada donde residen los archivos de proceso de jBPM 2.0. Sustituya "outdirectory" por el directorio de salida creado para guardar los archivos de proceso recién convertidos.

16.2. Delegación

La delegación es el mecanismo utilizado para incluir el código personalizado de los usuarios en la ejecución de procesos.

16.2.1. Cargador de clases jBPM

Es el cargador de clase que carga las clases de jBPM. Esto significa, el cargador de clase que tiene la biblioteca jbpm-3.x.jaren su ruta de clase. Para hacer visibles las clases para el cargador de clase de jBPM, es necesario ponerlas en un archivo jar y poner el archivo jar junto al jbpm-3.x.jar. Por ejemplo, en la carpeta WEB-INF/lib en el caso de aplicaciones web.

16.2.2. El cargador de clase de proceso

Las clases de delegación se cargan con el cargador de clase de proceso de su

respectiva definición de proceso. El cargador de clase de proceso es un cargador de clase que tiene el cargador de clase de jBPM como un primario. El cargador de clase de proceso agrega todas las clases de una determinada definición de proceso. Se puede agregar clases a una definición de proceso poniéndolas en la carpeta `/classes` del archivo de proceso. Se debe tener en cuenta que esto es útil sólo cuando se quiere versionar las clases que se agregan a la definición de proceso. Si las versiones no son necesarias, es mucho más eficiente poner las clases a disposición del cargador de clase de jBPM.

16.2.3. Configuración de delegaciones

Las clases de delegación contienen código de usuario que se llama desde dentro de la ejecución de un proceso. El ejemplo más común es una acción. En el caso de la acción, se puede llamar a una implementación de la interfaz `ActionHandler` en un evento del proceso. Las delegaciones se especifican en el `processdefinition.xml`. Se puede suministrar tres datos al especificar una delegación:

- 1) el nombre de clase (se requiere): el nombre de clase completo de la clase de delegación.
- 2) el tipo de configuración (opcional): especifica la forma de instanciar y configurar el objeto de delegación. En forma predeterminada se usar el constructor predeterminado y se omite la información de configuración.
- 3) la configuración (opcional): la configuración del objeto de delegación en el formato requerido por el tipo de configuración.

La siguiente es una descripción de todos los tipos de configuración:

16.2.3.1. config-type field

Este es el tipo de configuración predeterminado. El `config-type field` instanciará primero un objeto de la clase de delegación y luego configurará valores en los campos del objeto según se especifica en la configuración. La configuración es xml, donde los elementnames deben coincidir con los nombres de campo de la clase. El texto de contenido del elemento se pone en el campo correspondiente. Si es necesario y factible, el texto de contenido del elemento se convierte al tipo de campo.

Conversiones de tipo compatible:

- La cadena no necesita conversión, desde luego. Pero se recorta.
- Tipos primitivos como int, long, float, double, ...
- Y las clases de envoltura básica para los tipos primitivos.
- Listas, conjuntos y colecciones. En ese caso cada elemento del contenido xml se considera un elemento de la colección y se analiza, en forma recursiva, aplicando las conversiones. Si el tipo de los elementos es diferente de `java.lang.String` esto se puede indicar especificando un atributo type con el nombre de tipo completo.
- Asignaciones. En este caso, se espera que cada elemento del campo-elemento tenga un subelemento `key` y un elemento `value`. Ambos se analizan usando las reglas de conversión en forma recursiva. Tal como ocurre con las colecciones, se supone una conversión a `java.lang.String` si no se especifica ningún atributo `type`.

- org.dom4j.Element
- Para cualquier otro tipo, se usa el constructor de cadena.

Por ejemplo en la siguiente clase...

```
public class MyAction implements ActionHandler {
  // los especificadores de acceso pueden ser privados, predeterminados,
protegidos o públicos
  private String city;
  Integer rounds;
  ...
}
```

...esta es una configuración válida:

```
...
<action class="org.test.MyAction">
  <city>Atlanta</city>
  <rounds>5</rounds>
</action>
...
```

16.2.3.2. config-type bean

Igual a config-type field pero luego las propiedades se configuran a través de métodos configuradores, en lugar de directamente en los campos. Se aplican las mismas conversiones.

16.2.3.3. config-type constructor

Este instanciador tomará el contenido completo del elemento xml de delegación y lo pasará como texto en el constructor de clase de delegación.

16.2.3.4. config-type configuration-property

En primer lugar, se usa el constructor predeterminado, luego este instanciador tomará el contenido completo del elemento xml de delegación y lo pasará como texto en el método `void configure(String);`. (como en jBPM 2)

16.3. Expresiones

Para algunas de las delegaciones, existe compatibilidad para un JSP/JSF EL como lenguaje de expresión. En acciones, asignaciones y condiciones de decisión, se puede escribir una expresión como, por ejemplo, `expression="#{myVar.handler[assignments].assign}"`

Los fundamentos de este lenguaje de expresión se pueden encontrar en el tutorial J2EE.

El lenguaje de expresión de jPDL es similar al lenguaje de expresión de JSF. Esto significa que jPDL EL se basa en JSP EL, pero usa la notación #{...}e incluye compatibilidad para enlace de método.

Según el contexto, se puede usar variables de proceso o variables de instancia de

proceso como variables de inicio junto con los siguientes objetos implícitos:

- taskInstance (org.jbpm.taskmgmt.exe.TaskInstance)
- processInstance (org.jbpm.graph.exe.ProcessInstance)
- processDefinition (org.jbpm.graph.def.ProcessDefinition)
- testigo (org.jbpm.graph.exe.Token)
- taskMgmtInstance (org.jbpm.taskmgmt.exe.TaskMgmtInstance)
- contextInstance (org.jbpm.context.exe.ContextInstance)

Esta característica se vuelve realmente potente en un entorno JBoss SEAM. Debido a la integración entre jBPM y JBoss SEAM, odos los beans respaldados, EJB's y demás `one-kind-of-stuff` quedan disponibles directamente dentro de la definición de proceso. ¡Gracias, Gavin! ¡Absolutamente impresionante! :-)

16.4. Diagrama xml de jPDL

El esquema de jPDL es aquél utilizado en el archivo processdefinition.xmlen el archivo de proceso.

16.4.1. Validación

Al analizar un documento XML de jPDL, jBPM validará el documento contra el esquema de jPDL cuando se cumplan dos condiciones: primero, se debe hacer referencia al esquema en el documento XML de la siguiente manera

```
<process-definition xmlns="urn:jbpm.org:jpdl-3.1">
  ...
</process-definition>
```

Y segundo, el analizador xerces debe estar en la ruta de clase.

The jPDL schema can be found in `${jbpm.home}/src/java.jbpm/org/jbpm/jpdl/xml/jpdl-3.1.xsd` or at http://jbpm.org/jpdl-3.1.xsd.

16.4.2. process-definition

Cuadro 16.1.

Nombre	Tipo	Multiplicidad	
name	atributo	opcional	El nombre del proceso
swimlane	elemento	[0..*]	Los carriles que se usan en es de proceso y se usan para asig
start-state	elemento	[0..1]	El estado de inicio del proces estado de inicio es válido, per
{end-state|state|node|task-node|process-state|super-state|fork|join|decision}	elemento	[0..*]	Los nodos de la definición de sin nodos es válido, pero no se
event	elemento	[0..*]	Los eventos de proceso que s
{action|script|create-timer|cancel-timer}	elemento	[0..*]	Las acciones definidas globaln eventos y transiciones. Se deb

Nombre	Tipo	Multiplicidad	
			especificar un nombre para qu
task	elemento	[0..*]	Las tareas definidas globalme acciones.
exception-manejador	elemento	[0..*]	Una lista de controladores de descartadas por clases de dele

16.4.3. el nodo

cuadro 16.2.

Nombre	Tipo	Multiplicidad	De
{action|script|create-timer|cancel-timer}	elemento	1	Una acción personalizada para este nodo
common el nodo elements			Consulte common node elem

16.4.4. common el nodo elements

cuadro 16.3.

Nombre	Tipo	Multiplicidad	Descripc
name	atributo	requerido	El nombre del nodo
async	atributo	{ true | false }, false is the default	Si se configura en true, este nodo se ejecutará en forn Continuationes asíncronas
transition	elemento	[0..*]	Las transiciones salientes. Cada transición que deja permite que un máximo de una de las transiciones sa que se especifica se llama la transición predeterm cuando el nodo se deja sin especificar una transición.
event	elemento	[0..*]	Tipos de evento compatibles: {node-enter|node-leave
exception-manejador	elemento	[0..*]	Una lista de controladores de excepción que se aplica de delegación incluidas en esta definición de nodo.
temporizador	elemento	[0..*]	Especifica un temporizador que supervisa la duración

16.4.5. start-state

cuadro 16.4.

Nombre	Tipo	Multiplicidad	Descripci
name	atributo	opcional	el nombre del nodo
task	elemento	[0..1]	La tarea de iniciar una nueva instancia para este proceso o pa sección 11.7, "Carril en tarea de inicio"
event	elemento	[0..*]	tipos de evento compatible: {node-leave}
transition	elemento	[0..*]	Las transiciones salientes. Cada transición que deja u
exception-manejador	elemento	[0..*]	Una lista de controladores de excepción que se aplica de delegación incluidas en esta definición de nodo.

16.4.6. end-state

cuadro 16.5.

Nombre	Tipo	Multiplicidad	Descripci
name	atributo	requerido	El nombre del estado final.
event	elemento	[0..*]	Tipos de evento compatibles: {node-enter}.
exception-manejador	elemento	[0..*]	Una lista de controladores de excepción que se aplica de delegación incluidas en esta definición de nodo.

16.4.7. state

cuadro 16.6.

Nombre	Tipo	Multiplicidad	Descripción
common el nodo elements			Consulte common node elements

16.4.8. nodo de tareas

cuadro 16.7.

Nombre	Tipo	Multiplicidad	Descripció
signal	atributo	opcional	{unsynchronized|never|first|first-wait|last|last-wait}, especifica el efecto de la finalización de tarea en la cont
create-tareas	atributo	opcional	{yes|no|true|false}, el valor predeterminado es `true`. S de tiempo de ejecución debe determinar cuáles de las ta agregar una acción en `node-enter`, crear las tareas en `false`.
end-tareas	atributo	opcional	{yes|no|true|false}, el valor predeterminado es `false`. En casc `leave`, todas las tareas que aún están abiertas se finalizan.
task	elemento	[0..*]	{yes|no|true|false}, el valor predeterminado es `false`. En casc `leave`, todas las tareas que aún están abiertas se finalizan.
common el nodo elements			Consulte common node elements

16.4.9. process-state

cuadro 16.8.

Nombre	Tipo	Multiplicidad	Descripci
sub-process	elemento	1	El subproceso que se asocia con este nodo.
variable	elemento	[0..*]	Especifica la forma en que los datos debe copiarse d inicio y desde el subproceso al superproceso tras finali
common el nodo elements			Consulte common node elements

16.4.10. super-state

cuadro 16.9.

Nombre	Tipo	Multiplicidad	
{end-state | state | el nodo | task-node | process-state | super-state | fork | join | decision}	elemento	[0..*]	Los n anida
common el nodo elements			Cons

16.4.11. fork

cuadro 16.10.

Nombre	Tipo	Multiplicidad	Descripción
common el nodo elements			Consulte common node elements

16.4.12. join

cuadro 16.11.

Nombre	Tipo	Multiplicidad	Descripción
common el nodo elements			Consulte common node elements

16.4.13. decision

cuadro 16.12.

Nombre	Tipo	Multiplicidad	Descr
manejador	elemento	se debe especificar un elemento 'manejador' o las condiciones en las transiciones	el nombre de una implementación de `org.jbp`
transition	elemento	[0..*]	Las transiciones salientes. Las transiciones salientes condición. La decisión buscará la primera transición transición sin una condición se considera que evalúa Consulte el elemento condición
common el nodo elements			Consulte common el node elements

16.4.14. event

cuadro 16.13.

Nombre	Tipo	Multiplicidad	
type	atributo	requerido	El tipo de evento que se expresa evento.
{action | script | create-timer | cancel-timer}	elemento	[0..*]	La lista de acciones que deben ej

16.4.15. transition

cuadro 16.14.

Nombre	Tipo	Multiplicidad	Desc
name	atributo	opcional	El nombre de la transición. Se debe tener `nodo` *debe* tener un nombre distinto
to	atributo	requerido	El nombre jerárquico del nodo de destino. Pa jerárquicos, consulte la sección 9.6.3, "Nomb
{action|script|create-timer|cancel-timer}	elemento	[0..*]	Las acciones que se ejecutarán tras asumir e acciones de una transición no necesitan pone
exception-manejador	elemento	[0..*]	Una lista de controladores de excepción que por clases de delegación incluidas en esta de

16.4.16. action

cuadro 16.15.

Nombre	Tipo	Multiplicidad	Descri
name	atributo	opcional	El nombre de la acción. Cuando las acciones rec definición de proceso. Esto puede ser útil para a acciones sólo una vez.
class	attibute	ya sea, a ref-name o una expresión	El nombre de clase completo de la clase que imp `org.jbpm.graph.def.ActionHandler`.
ref-name	attibute	ya sea este o clase	El nombre de la acción a la que se hace referen procesando si se especifica una acción a la que s
expression	attibute	ya sea este, una clase o un ref-name	Una expresión jPDL que resuelve un método. Cons
accept-propagated-events	atributo	opcional	{yes|no|true|false}. El valor predeterminado es y se ejecutará en eventos que fueron activados en información, consulte la sección 9.5.4, "Propagac
config-type	atributo	opcional	{field|bean|constructor|configuration-property}. object y cómo el contenido de este elemento deb para ese action-object.
	{content}	opcional	El contenido de la acción se puede usar como info implementaciones de acción personalizadas. Esto reutilizables. Para obtener más información acerc sección 16.2.3, "Configuración de delegaciones".

16.4.17. script

cuadro 16.16.

Nombre	Tipo	Multiplicidad	Descripc
name	atributo	opcional	El nombre de la secuencia de comandos-acción. Cuando la

Nombre	Tipo	Multiplicidad	Descripc
			desde la definición de proceso. Esto puede ser útil para acciones sólo una vez.
accept-propagated-events	atributo	opcional [0..*]	{yes|no|true|false}. El valor predeterminado es yes|t ejecutará en eventos que fueron activados en el elem información, consulte la sección 9.5.4, "Propagación
expression	elemento	[0..1]	la secuencia de comando beanshell. Si no especifica e expresión como el contenido del elemento de secuenc elemento de expresión).
variable	elemento	[0..*]	Variable in para la secuencia de comandos. Si no se e testigo actual se cargarán en la evaluación de la secu in si se quiere limitar el número de variables cargadas

16.4.18. expression

cuadro 16.17.

Nombre	Tipo	Multiplicidad	Descripción
	{content}		una secuencia de comando bean shell.

16.4.19. variable

cuadro 16.18.

Nombre	Tipo	Multiplicidad	Descripción
name	atributo	requerido	El nombre de variable de proceso.
access	atributo	opcional	El valor predeterminados es `read,write`. Es una lista sepa Los únicos especificadores de acceso utilizados hasta ahora so
mapped-name	atributo	opcional	En forma predeterminada toma el nombre de variable. Espec variable. El significado del mapped-name depende del conte secuencia de comandos, será el nombre de variable de sec tarea, será la etiqueta del parámetro de forma de tarea y pa utilizado en el subproceso.

16.4.20. manejador

cuadro 16.19.

Nombre	Tipo	Multiplicidad	Descripció
expression	attibute	ya sea este o una clase	Una expresión de jPDL. El resultado devuelto se transform cadena resultante debe coincidir con una de las transicion sección 16.3, "Expresiones".
class	attibute	ya sea este o un ref-name	El nombre de clase completo de la clase que implementa `org.jbpm.graph.node.DecisionHandler`.
config-type	atributo	opcional	{field|bean|constructor|configuration-property}. Especifi

Nombre	Tipo	Multiplicidad	Descripció
			cómo el contenido de este elemento debe usarse como in object.
	{content}	opcional	El contenido del controlador se puede usar como informa implementaciones de controlador personalizadas. Esto pe reutilizables. Para obtener más información acerca de co sección 16.2.3, "Configuración de delegaciones".

16.4.21. temporizador

cuadro 16.20.

Nombre	Tipo	Multiplicidad	Descripc
name	atributo	opcional	El nombre del temporizador. Si no se especifica ni cierre. Se debe tener en cuenta que cada temporizad
duedate	atributo	requerido	La duración (expresada opcionalmente en horas hábile creación del temporizador y la ejecución del temporiz obtener más información de la sintaxis.
repeat	atributo	opcional	{duration | 'yes' | 'true'} Después que un temporizador se h opcionalmente la duración entre ejecuciones reiteradas de especifica `yes` o `true`, se toma la misma duración de la fec sección 14.1, "Duración" para obtener más informació
transition	atributo	opcional	Un nombre de transición que se toma cuando se e evento de temporizador y ejecutar la acción (si la ha
cancel-event	atributo	opcional	Este atributo sólo debe usarse en temporizadores temporizador debe cancelarse. En forma predeterm puede configurar, por ejemplo, en `task-assign` o pueden combinar especificándolos en una lista separa
{action|script|create-timer|cancel-timer}	elemento	[0..1]	Una acción que se debe ejecutar cuando este tempor

16.4.22. create-temporizador

cuadro 16.21.

Nombre	Tipo	Multiplicidad	Descripción
name	atributo	opcional	El nombre del temporizador. El nombre se puede usar para ca timer.
duedate	atributo	requerido	La duración (expresada opcionalmente en horas hábiles) que es del temporizador y la ejecución del temporizador. Consulte la s información sobre la sintaxis.
repeat	atributo	opcional	{duration | 'yes' | 'true'}Después que un temporizador se ha ejec opcionalmente la duración entre ejecuciones reiteradas del tem especifica `yes` o `true`, se toma la misma duración de la fecha la sección 14.1, "Duración" para obtener más información de la

Nombre	Tipo	Multiplicidad	Descripción
transition	atributo	opcional	Un nombre de transición que se toma cuando se ejecuta el tem temporizador y ejecutar la acción (si la hay).

16.4.23. cancel-temporizador

cuadro 16.22.

Nombre	Tipo	Multiplicidad	Descripción
name	atributo	opcional	El nombre del temporizador que se cancelará

16.4.24. task

cuadro 16.23.

Nombre	Tipo	Multiplicidad	Descripción
name	atributo	opcional	`El nombre de la tarea. Se puede hacer refer a través de la TaskMgmtDefinition`
blocking	atributo	opcional	{yes|no|true|false}, el valor predeterminado es false. S puede dejar cuando la tarea no ha finalizado. Si se confi testigo permite continuar la ejecución y dejar el nodo. porque normalmente la interfaz de usuario fuerza el block
signalling	atributo	opcional	{yes|no|true|false}, el valor predeterminado es true. Si si podrá desencadenar la continuación del testigo.
duedate	atributo	opcional	Es una duración expresada en horas absolutas o hábiles, ta hábil
swimlane	atributo	opcional	referencia a un carril. Si se especifica un carril en una tare
priority	atributo	opcional	Una de {highest, high, normal, low, lowest}. Si no, se pue prioridad. A modo de información: (máximo=1, mínimo=5)
assignment	elemento	opcional	describe una delegación que asigna la tarea a un actor cua
event	elemento	[0..*]	Tipos de evento compatibles: {task-create|task-start|task-assign|task-end}. Especialm propiedad no persistida `previousActorId` a la `TaskIn`
exception-manejador	elemento	[0..*]	Una lista de controladores de excepción que se aplica a t delegación incluidas en esta definición de nodo.
temporizador	elemento	[0..*]	Especifica un temporizador que supervisa la duración d temporizadores de tarea, se puede especificar el `ca` `cancel-event` es `task-end`, pero se puede personali `start`.
controller	elemento	[0..1]	Especifica cómo las variables de proceso se transforman e usuario usa los parámetros de forma de tarea para presen

16.4.25. swimlane

cuadro 16.24.

Nombre	Tipo	Multiplicidad	Descripción
name	atributo	requerido	El nombre del carril. Se puede hacer referencia `TaskMgmtDefinition`.
assignment	elemento	[1..1]	Especifica la asignación de este carril. La asignación se rea cree en este carril.

16.4.26. assignment

cuadro 16.25.

Nombre	Tipo	Multiplicidad	Descripción
expression	atributo	opcional	Por motivos históricos, esta expresión de atributo **no** hace r es una expresión de asignación para el componente de ident acerca de cómo escribir expresiones de componente de iden "expresiones de asignación". Se debe tener en cuenta que e componente de identidad de jbpm.
actor-id	atributo	opcional	Un actorId. Se puede usar junto con pooled-actors. El actor- puede hacer referencia a un actorId fijo de esta manera `ac` referencia a una propiedad o método que devuelva una cade `actorid="myVar.actorId",` la cual invocará el método getActorId en la variable de insta
pooled-actors	atributo	opcional	Una lista separada por comas de actorIds. Se puede usar en actores agrupados se puede especificar de la siguiente man `pointersisters".` Los pooled-actors se resolverán como una expresión. Así qu propiedad o método que debe devolver una cadena[], una c actores agrupados
class	atributo	opcional	El nombre de clase completo de una implementación de `or` `.def.AssignmentHandler`
config-type	atributo	opcional	{field | bean | constructor | configuration-property}. Especifica object y cómo el contenido de este elemento debe usarse c assignment-handler-object.
	{content}	opcional	El contenido del assignment-element se puede usar como in implementaciones de AssignmentHandler. Esto permite la cr Para obtener más información acerca de configuración de de "Configuración de delegaciones".

16.4.27. controller

cuadro 16.26.

Nombre	Tipo	Multiplicidad	Descripción
class	atributo	opcional	`El nombre de clase completo de una implementaci`

Nombre	Tipo	Multiplicidad	Descripción
			`org.jbpm.taskmgmt.def.TaskControllerHandler.`
config-type	atributo	opcional	{field | bean | constructor | configuration-property}. Especifica có object y cómo el contenido de este elemento debe usarse como assignment-handler-object.
	{content}		El contenido del controlador es la configuración del controla atributo class). Si no se especifica ningún controlador de tarea variables.
variable	elemento	[0..*]	En caso de que el atributo class no especifique ningún contr controlador debe ser una lista de variables.

16.4.28. sub-process

cuadro 16.27.

Nombre	Tipo	Multiplicidad	Descripción
name	atributo	requerido	El nombre del subproceso. Para saber cómo someter a prueba sub subprocesos"
version	atributo	opcional	La versión del subproceso. Si no se especifica ninguna versión, dado.

16.4.29. condition

cuadro 16.28.

Nombre	Tipo	Multiplicidad	Descripció
	{content} or atributo expression	requerido	El contenido del elemento de condición es una expresión j decisión toma la primera transición (según lo ordenado en expresión se resuelve en `true`. Si ninguna de las condicion saliente predetermina

16.4.30. exception-manejador

cuadro 16.29.

Nombre	Tipo	Multiplicidad	Descripción
exception-class	atributo	opcional	Especifica el nombre completo de la clase java descartable excepción. Si este atributo no se especifica, coincide con t (`ble`).
action	elemento	[1..*]	Una lista de acciones que se ejecutará cuando una excepci excepción.

Capítulo 17. Seguridad

Las características de seguridad de jBPM aún se encuentran en etapa alfa. En este capítulo se documenta la autenticación y la autorización conectables. Además, se identifican las partes de la estructura que están terminadas y las que no.

17.1. Tareas pendientes

Por parte de la estructura, aún falta definir un conjunto de permisos que el motor de jbpm verifica mientras se ejecuta un proceso. Actualmente, cada usuario puede comprobar sus propios permisos, pero aún no existe un conjunto de permisos predeterminado en jbpm.

Existe sólo una implementación de autenticación predeterminada en estado finalizado. Hay otras implementaciones de autenticación previstas, pero todavía no se implementan. La autorización es opcional, y aún no existe implementación de autorización. En el caso de la autorización también, hay una serie de implementaciones de autorización previstas, pero éstas aún no se han resuelto.

Pero tanto para autenticación como para autorización, la estructura está allí para conectar un mecanismo propio de autenticación y autorización.

17.2. Autenticación

La autenticación es el proceso de saber en nombre de quién se está ejecutando el código. En el caso de jBPM, el entorno debe poner esta información a disposición de jBPM. Debido a que jBPM siempre se ejecuta en un entorno específico como una aplicación web, un EJB, una aplicación swing o algún otro entorno, es siempre el entorno circundante el que debe realizar la autenticación.

En unas cuantas situaciones, jBPM necesita saber quién ejecuta el código. Por ejemplo, para agregar información de autenticación en los registros de proceso a fin de saber quién hizo qué y cuándo. Otro ejemplo es el cálculo de un actor sobre la base del actor autenticado actual.

En cada situación que jBPM necesita saber quién ejecuta el código, se llama al método central `org.jbpm.security.Authentication.getAuthenticatedActorId()`. Dicho método se delegará a una implementación de `org.jbpm.security.authenticator.Authenticator`. Al especificar una implementación del autenticador, se puede configurar la forma en que jBPM recupera al actor actualmente autenticado desde el entorno.

El autenticador predeterminado es `org.jbpm.security.authenticator.JbpmDefaultAutenticator`. Dicha implementación mantendrá una pila `ThreadLocal` de actorId autenticados. Los bloques autenticados se pueden marcar con los métodos `JbpmDefaultAutenticator.pushAuthenticatedActorId(String)` y `JbpmDefaultAutenticator.popAuthenticatedActorId()`. Asegúrese de poner siempre estar demarcaciones en un bloque try-finally. Para los métodos de inserción y extracción de la implementación de este autenticador, existen métodos preparados provistos en la clase de Autenticación base. El motivo por el cual

JbpmDefaultAutenticator mantiene una pila de actorIds en lugar de sólo un actorId es simple: permite que el código de jBPM diferencie entre código que se ejecuta en nombre del usuario y código que se ejecuta en nombre del motor de jbpm

Consulte los javadocs para obtener más información.

17.3. Autorización

La autorización valida si un usuario autenticado tiene permitido realizar una operación segura.

El motor de jBPM y el código de usuario pueden verificar si un usuario tiene permitido realizar una operación dada con el método `org.jbpm.security.Authorization.checkPermission(Permission)`.

La clase de Autorización delegará además dicha llamada a una implementación configurable. La interfaz para conectarse en distintas estrategias de autorización es `org.jbpm.security.authorizer.Authorizer`.

En el paquete org.jbpm.security.authorizer hay algunos ejemplos que muestran proyectos de implementaciones de autorizador. La mayoría no se ha implementado completamente y ninguno se ha sometido a prueba.

También es una tarea pendiente la definición de un conjunto de permisos de jBPM y la verificación de dichos permisos por parte del motor de jBPM. Un ejemplo podría ser verificar que el usuario autenticado actual tenga suficientes privilegios para finalizar una tarea mediante llamada a `Authorization.checkPermission(new TaskPermission("end", Long.toString(id)))` en el método TaskInstance.end().

Capítulo 18. TDD para flujo de trabajo

18.1. Introducción de TDD para flujo de trabajo

Puesto que el desarrollo de software orientado a procesos no se diferencia del desarrollo de cualquier otro software, creemos que las definiciones de proceso deben ser sometibles a prueba fácilmente. Este capítulo muestra la forma de usar JUnit simple sin ninguna extensión para realizar pruebas unitarias de las definiciones de proceso que se crean.

El ciclo de desarrollo de mantenerse tan corto como sea posible. Debe ser posible verificar inmediatamente los cambios realizados en los orígenes del software. De preferencia, sin ningún paso intermedio de versión compilada. Los ejemplos que se presentan a continuación mostrarán la forma de desarrollar y someter a prueba procesos de jBPM sin pasos intermedios.

En su mayoría, las pruebas unitarias de definiciones de proceso son escenarios de ejecución. Cada escenario se ejecuta en un método de prueba JUnit e introduce los desencadenadores externos (leer: señales) en la ejecución de un proceso y después de cada señal verifica si el proceso está en el estado esperado.

He aquí un ejemplo de dicha prueba. Se toma una versión simplificada del proceso de subasta con la siguiente representación gráfica:

Figura 18.1. El proceso de prueba de subasta

Ésta es la versión escrita de una prueba que ejecuta el escenario principa:

```
public class AuctionTest extends TestCase {

  // analiza la definición de proceso
  static ProcessDefinition auctionProcess =
      ProcessDefinition.parseParResource("org/jbpm/tdd/auction.par");

  // obtiene los nodos para fácil aserción
  static StartState start = auctionProcess.getStartState();
  static State auction = (State) auctionProcess.getNode("auction");
  static EndState end = (EndState) auctionProcess.getNode("end");

  // la instancia de proceso
  ProcessInstance processInstance;

  // la ruta principal de ejecución
  Token token;

  public void setUp() {
    // crea una instancia de proceso para la definición de proceso dada
    processInstance = new ProcessInstance(auctionProcess);

    // la ruta principal de ejecución es el testigo raíz
    token = processInstance.getRootToken();
  }

  public void testMainScenario() {
    // después de crear la instancia de proceso, la ruta principal de
```

```
    // ejecución se posiciona en el estado de inicio.
    assertSame(start, token.getNode());

    token.signal();

    // después de la señal, la ruta principal de ejecución se ha
    // movido al estado de subasta
    assertSame(auction, token.getNode());

    token.signal();

    // después de la señal, la ruta principal de ejecución se ha
    // movido al estado final y proceso ha terminado
    assertSame(end, token.getNode());
    assertTrue(processInstance.hasEnded());
  }
}
```

18.2. Fuentes XML

Antes de comenzar a escribir escenarios de ejecución, es necesaria una `ProcessDefinition`. La forma más fácil de obtener un objeto `ProcessDefinition` es analizando xml. Si se tiene finalización de código, escriba `ProcessDefinition.parse`y active la finalización de código. Luego se obtienen los diversos métodos de análisis. Existen básicamente 3 formas de escribir xml que se pueda analizar a un objeto `ProcessDefinition`:

18.2.1. Análisis de un archivo de proceso

Un archivo de proceso es un archivo zip que contiene el xml de proceso en un archivo llamado `processdefinition`.xml. El diseñador de proceso de jBPM lee y escribe archivos de proceso. Por ejemplo:

```
...
static ProcessDefinition auctionProcess =
    ProcessDefinition.parseParResource("org/jbpm/tdd/auction.par");
...
```

18.2.2. Análisis de un archivo xml

En otras situaciones, es posible que se quiera escribir el archivo processdefinition.xml a mano y luego empaquetar el archivo zip con, por ejemplo, una secuencia de comandos ant. En tal caso, se puede usar `JpdlXmlReader`

```
...
static ProcessDefinition auctionProcess =
    ProcessDefinition.parseXmlResource("org/jbpm/tdd/auction.xml");
...
```

18.2.3. Análisis de una secuencia de comandos xml

La opción más simple es analizar el xml en la prueba unitaria en línea desde una secuencia de comandos simple.

```
...
static ProcessDefinition auctionProcess =
    ProcessDefinition.parseXmlString(
  "<process-definition>" +
  "  <start-state name='start'>" +
  "    <transition to='auction'/>" +
  "  </start-state>" +
  "  <state name='auction'>" +
  "    <transition to='end'/>" +
  "  </state>" +
  "  <end-state name='end'/>" +
  "</process-definition>");
...
```

18.3. Prueba de subprocesos

Tarea pendiente (consulte test/java/org/jbpm/graph/exe/ProcessStateTest.java)

Capítulo 19. Arquitectura contectable

La funcionalidad de jBPM se divide en módulos. Cada módulo tiene una parte de definición y una de ejecución (o tiempo de ejecución). El módulo central es el módulo de gráficos, compuesto de la `ProcessDefinition` y la `ProcessInstance`. La definición de proceso contiene un gráfico y la instancia de proceso representa una ejecución del gráfico. Todas las demás funcionalidades de jBPM están agrupadas en módulos opcionales. Los módulos opcionales pueden ampliar el módulo de gráficos con características adicionales como contexto (variables de proceso), administración de tareas, temporizadores, etc.

Figura 19.1. Arquitenctura conectable

Además, la arquitectura conectable de jBPM es un mecanismo excepcional para agregar recursos personalizados al motor de jBPM. La información de definición de proceso personalizada se puede agregar mediante la adición de una implementación de `ModuleDefinition` a la definición de proceso. Cuando se cree la `ProcessInstance`, se crea una instancia para cada `ModuleDefinition` en la `ProcessDefinition`. La `ModuleDefinition` se usa como una fábrica de `ModuleInstances`.

La forma más integrada de ampliar la información de definición de proceso es agregarla al archivo de proceso e implementar un `ProcessArchiveParser`. El `ProcessArchiveParser` puede analizar la información agregada al archivo de proceso, crear la `ModuleDefinition` personalizada y agregarla a la `ProcessDefinition`.

```
public interface ProcessArchiveParser {

  void writeToArchive(ProcessDefinition processDefinition, ProcessArchive
archive);
  ProcessDefinition readFromArchive(ProcessArchive archive, ProcessDefinition
processDefinition);

}
```

Para realizar este trabajo, la `ModuleInstance` personalizada debe recibir notificación de eventos relevantes durante la ejecución del proceso. La `ModuleDefinition` personalizada podría agregar implementaciones de `ActionHandler` tras eventos del proceso que sirvan como controladores de devolución de llamada para estos eventos del proceso.

Si no, un módulo personalizado podría usar AOP para enlazar la instancia personalizada con la ejecución del proceso. JBoss AOP se adapta muy bien a este trabajo gracias a que es óptima, fácil de aprender y también forma parte de la pila de JBoss.