Universidad Politécnica de Madrid
Escuela Técnica Superior de Ingenieros Informáticos

Proyecto Fin de Carrera

Diseño de lenguajes de programación basados en procesos de negocio ejecutables

AUTOR: Álvaro Pantoja Casero

TUTOR: Loïc Martínez Normand

Julio 2018

A mis padres.

A los que me vieron empezar, pero no acabar.

A los que me adelantaron por la derecha.

Agradecimientos

Desde estas líneas quiero en primer lugar agradecer su confianza, su tiempo y su esfuerzo a mis padres, sin los cuales hubiera sido difícil optar a estudiar y – al fin – terminar esta carrera. No hay suficientes palabras para agradecer tantos viajes, *tuppers*, llamadas, paciencia, dinero y preocupaciones.

Por otro lado quiero agradecer a todos aquellos con los que me codeé en algún momento u otro haciendo practicas, compartiendo apuntes, ejercicios, cafés, partidas o bostezos en clase, y cuyos nombres han caído en el mas inintencionado olvido.

Además, claro, un recuerdo especial para resto de la tropa que permaneció unida hasta que el tiempo y la vida nos ubicó a cada uno en su sitio: Nacho, Sanfrux, Juanche, Raúl, Pipe, Miguel y Charly.

Por último, a Loïc por su tiempo y sus gestiones en este ultimo esfuerzo.

Índice de contenido

Índice de figuras

Índice de tablas

1 Introducción

Hoy en día, cualquier negocio que tenga informatizada en parte o totalmente la gestión de su información o de los procesos que la manejan, sabe de las dificultades que surgen a la hora de realizar cualquier mínimo cambio –por nimio que sea– en cualquiera de sus componentes, ya sea la lógica interna, las bases de datos o las interfaces gráficas usadas por los usuarios del mismo. Algunos ejemplos de las dificultades que surgen son: desarrollos largos y/o complejos, usuarios insatisfechos, perdidas de información, propensión a errores... Puede ocurrir que este hecho impida afrontar en muchos casos los cambios deseados.

Otro de los problemas relacionados con la gestión de la información, es la mala gestión de la *información del negocio* contenida en dichos sistemas. La lógica o procesos de negocio implementados por el sistema suelen ser desconocidos para gran parte de los implicados en el negocio, y con frecuencia suelen estar mal documentados y diseñados. En consecuencia, se dificulta en gran medida su mantenimiento y posibilidad de adaptación y mejora.

Como resultado de ambos escenarios suele ocurrir que aparecen sistemas informáticos gigantescos, monolíticos, inamovibles y que solo se reemplazan cuando son demasiado obsoletos y hay que migrar todo el sistema a un nuevo desarrollo, que normalmente tiene poco o nada tiene que ver con lo anterior.

Podría pensarse que este problema solo atañe a negocios y sistemas de un tamaño considerablemente grande, pero también sucumben a esta problemática sistemas medianos (aplicación de red de una empresa) o pequeños (programa de contabilidad en un único ordenador). Baste con pensar en cómo explicar a un pequeño empresario que *hoy* no puede disponer de su sistema por una actualización crítica, cuando mañana hay que enviar sin dilación a la administración de turno aquel documento tan urgente. Sirva esto último, además, como ejemplo del tipo de información que debiera formar parte de la lógica del sistema: cuándo se pueden realizar tareas de mantenimiento del mismo, y cuándo no.

El Proyecto Fin de Carrera (PFC) aquí expuesto pretende dar una solución genérica a este problema para un subconjunto concreto de estos sistemas, en los que la lógica está muy bien definida y puede representarse **gráficamente** mediante uno o varios diagramas de flujo, según el detalle y las dimensiones que se precisen. No obstante, y bajo un enfoque optimista, idealmente cualquier otro sistema podría migrarse y desarrollarse con el lenguaje y las herramientas aquí propuestas.

La solución aportada pretende ser genérica para permitir a cada negocio implementar sobre ella las partes específicas que requiera para cubrir sus necesidades.

La idea que subyace en la solución adoptada es la de poder proporcionar un entorno mínimo que permita diseñar sistemas cuya lógica quede lo mas definida, encapsulada y desacoplada posible, de modo que cualquier cambio en el sistema sea muy sencillo de realizar y minimice el impacto en el mismo.

Además, el sistema diseñado usará una notación propia de los *procesos de negocio*, que facilite la legibilidad, comprensión y modificación del sistema por actores de perfil no técnico. Lo cual permite también implicar en el diseño y mantenimiento del mismo a otros actores antes no presentes, como dirección, negocio, etc.

Por lo tanto, este PFC consiste en el diseño de un entorno de desarrollo base, que permite diseñar y ejecutar programas vistos como *procesos de negocio*, programas basados en la notación BPMN. El entorno se ha diseñado para que sea extensible y modificable para adaptarse a las necesidades de cada negocio.

El trabajo se divide en:

- El desarrollo de un lenguaje de programación basado en un subconjunto del lenguaje de diseño de procesos de negocio BPMN, que permita diseñar programas a partir de una representación gráfica de los mismos.
- El diseño y desarrollo de un compilador que permita crear programas ejecutables definidos mediante dicho lenguaje.
- El diseño y desarrollo de un entorno de ejecución, compuesto por un motor de procesos base (librería con clases base e interfaces), donde poder ejecutar dichos programas.

2 Estado de la cuestión

2.1 Introducción

Como se comentó en la introducción, es un problema conocido el hecho de que no se gestione convenientemente el *conocimiento* o *lógica* de un negocio cualquiera, ya sea una industria de acero, una universidad o la consulta de un dentista, por poner algunos ejemplos dispares. Es más, frecuentemente ese conocimiento o lógica del negocio [White and Miers, 2008]:

- Está al alcance solamente de algunos (pocos) participantes en el negocio.
- No se modela.
- No se documenta ni comunica convenientemente.
- No se amolda a las necesidades de los distintos perfiles.
- Es rígido, difícil de modificar y adaptar.

En las últimas décadas varios grupos de interés han realizado diferentes esfuerzos para intentar solventar estos problemas, esfuerzos que han dado como fruto la aparición de distintas soluciones que intentan suplir, total o parcialmente, las carencias antes descritas. Son los llamados *modelos de negocio.*

Como *modelo de negocio* se define al conjunto de conceptos, objectos, reglas y relaciones que permite modelar y expresar la lógica de un negocio cualquiera, de modo mas o menos abstracto [Recker,2005]. Conceptos como *tarea*, *responsable*, *proceso* o *esfuerzo,* o relaciones como "*pertenece a", "usado por", "asignado a*", tienen que quedar definidos con la suficiente amplitud y flexibilidad como para cubrir las necesidades de cualquier negocio, de modo que el modelo sea muy genérico, pero muy potente a la vez.

Algunos modelos de negocio son muy abstractos, como SPEM [OMG,2008a] o BPDM [OMG,2008b], y definen un meta-modelo o modelo-plantilla de elementos y conceptos de negocio que sirve de base para definir otros modelos mas específicos. No definen notaciones o metodologías, por lo que no se detallaran aquí.

Otros modelos bajan un escalón y ya proponen notaciones, diagramas y metodologías, que permiten entre otras cosas, modelar y representar gráficamente los elementos del negocio y las relaciones entre ellos.

Por ultimo, los modelos más detallados, proporcionan incluso soluciones en las que se incluyen elementos de procesamiento o ejecución de los procesos diseñados.

No obstante, una característica común a todos estos modelos de negocio es que son *especificaciones*, no implementaciones; esto es, proponen, y no imponen, un modelo sobre el que trabajar y desarrollar una implementación particular.

Pueden verse con claridad los distintos niveles en los que puede dividirse la actividad del Modelo de Negocio en el ejemplo de división piramidal propuesto por *BPTrends Associates* [BPTrends,2018] mostrado en la Figura 1. Un buen modelo de negocio debería cubrir lo máximo posible los dos primeros niveles, el empresarial y el de procesos, pero sin entrar en el último, el de implementación. No obstante, el modelo debe ser consciente de la existencia de este último nivel, y estar orientado a facilitar las tareas que han de realizarse dentro de él.

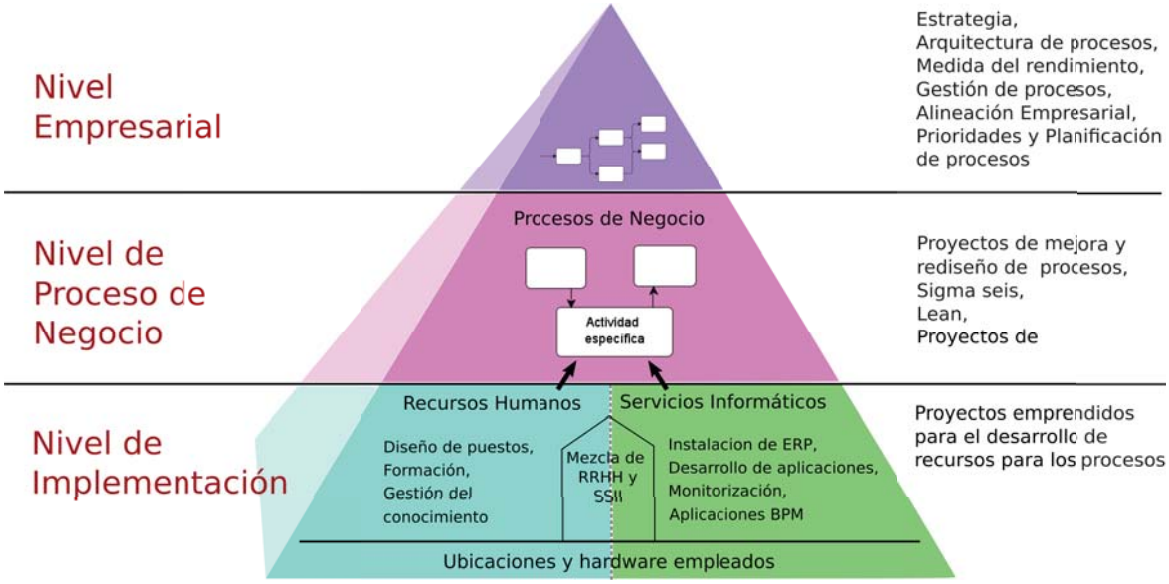


Figura 1: Niveles de actividad empresarial según los procesos de negocios

2.2 Procesos de negocio

Por proceso de negocio, dentro del ámbito BPM, se entiende como aquel conjunto de tareas relacionadas y estructuradas de modo que ejecutándose en una determinada secuencia permiten lograr un objetivo dentro del negocio [Dumas et al.,2013][von Scheel,2014]. Ha de ser por tanto un proceso estructurado y compresible, y suficientemente detallado como para favorece la comprensión del mismo por la audiencia, los actores, a la que vaya dirigido.

Actualmente hablar de *procesos de negocio* está a la orden del día en el vocabulario de muchos entornos empresariales –y no solo empresariales. El término se ha hecho *manido* debido a su generalidad, ya que permite englobar procedimientos de muy diversa índole, y representa el factor común que se extrae de los diversos procesos existentes.

Para manejar estos procesos, esto es, para definirlos, diseñarlos, comunicarlos, modificarlos, implementarlos, ejecutarlos... han surgido infinidad de notaciones, nomenclaturas y estándares.

Los retos a los que estos distintos enfoques intentan dar solución es:

- Negocios cambiantes.
- Perfiles muy diversos implicados en el negocio.
- Optimización de procesos.

Para ello, cualquier mecanismo de modelado y gestión de procesos debería permitir al menos:

- Modelado flexible.
- Diversas granularidades de definición.
- Reutilizabilidad.
- Automatización
- Trazabilidad
- Monitorización

2.3 Modelos de procesos y Lenguajes de notación

La mayoría de los modelos de procesos de negocio comentados anteriormente proporcionan un lenguaje o notación estándar para modelar procesos de negocio. Cada uno tiene sus peculiaridades y son mas o menos complejos y extensos. Se analizarán tres en concreto: XPDL, BPEL y BPMN.

La Tabla 1 incluye una referencia rápida a distintos estándares de lenguajes, modelos y metamodelos existentes y las organizaciones responsables de los mismos, entre los que se pueden localizar los 3 lenguajes analizados aquí.

Object Management Group OMG - (www.omg.org)	Business Process Modeling Notation (BPMN) Business Process Definition Metamodel (BPDM) SPEM 2.0
Workflow Management Coalition WfMC - (www.wfmc.org)	XML Process Definition Language (XPDL) Workflow API (WAPI) Workflow XML (WfXML)
OASIS (www.oasis-open.org)	Business Process Execution Language (BPEL)
W3C (www.w3c.org)	Open, SOAP, WSDL, core XML specifications Web Services Choreography Description Language (WS-CDL)
WS-I (www.ws-i.org)	Interoperability of WS technologies and standards WS-I Basic Profile

Tabla 1: Organizaciones y sus estándares

2.3.1 XPDL

Lenguaje propuesto por la WfMC, XPDL [WfMC,2005] nació con el propósito de facilitar el intercambio de definiciones de procesos de negocio entre herramientas de *workflow*. El objetivo del XPDL es proponer un lenguaje común para el nivel de procesos de negocio, permitiendo exportar e importar definiciones de procesos entre diversas aplicaciones, desde sistemas de gestión de *workflow* hasta herramientas de modelado y simulación.

Los principales elementos de este lenguaje son:

- **Paquete**: contenedor de otros elementos.
- **Aplicación**: elemento empleado para identificar otras aplicaciones o herramientas invocadas por los procesos.
- **Proceso o *workflow***: el proceso de negocio propiamente dicho. Compuesto de actividades y transiciones.
- **Actividad**: el elemento básico de un proceso. Es el encargado de definir alguna actividad o tarea a realizar. Se enlazan entre si mediante transiciones. Pueden ser actividades de enrutamiento (decisiones), de implementación (tarea, llamada a otro proceso, llamada a una aplicación, etc.) o de bloques (agrupador de subactividades).
- **Transición**: elemento para unir actividades, define además un sentido o transición de una a otra.
- **Participante**: elemento utilizado para identificar a los participantes en un proceso, esto es, aquellas entidades capaces de llevar a cabo un trabajo, como son: un rol, un humano, un sistema o un recurso.

- **Datos**: Elementos usados para especificar los datos relevantes para los procesos y actividades de los mismos. Estos datos se usan luego para tomar decisiones, o para compartir información entre actividades y/o subflujos de proceso.

El motor de procesos jBPM (ver 2.4.3.1) usó en un principio una modificación de esta notación para definir sus procesos, llamada jPDL, si bien luego migró a BPMN en versiones posteriores.

La ultima especificación XPDL es la 2.2.

2.3.2 BPEL

El Lenguaje de Ejecución de Procesos de Negocio BPEL [OASIS,2007], también conocido como WS-BPEL, es un lenguaje basado en XML que permite la descripción de procesos de negocio.

El lenguaje está enfocado a la orquestación de servicios web dentro de Arquitecturas Orientada a Servicios (SOA), permitiendo la composición de los servicios web con un enfoque orientado a procesos. Esto implica que el papel de BPEL está centrado principalmente en la ejecución de servicios web en el orden correcto, de manera que la implementación tecnológica de los procesos de negocio sea flexible y alineada con los objetivos del negocio.

BPEL define múltiples elementos de diseño, entre los que se destacan:

- **Participantes**: Los actores implicados los procesos de negocio.
- **Contenedores**: Los mensajes (datos) que se transmiten.
- **Operaciones**: Los servicios web y métodos invocados durante para el proceso.
- **Puertos**: Elementos que permiten la gestión de las conexiones con los servicios web, así como la transformación y combinación de los datos, ya sean los enviados o los recibidos.

La última especificación BPEL es la 2.0, publicada en 2007.

Aunque, como su propio nombre indica, parte de BPEL está pensado para diseñar procesos *ejecutables,* tiene como gran contra que es un lenguaje demasiado específico, orientado a orquestar llamadas a servicios web: carece de algunos aspectos de bajo nivel que son necesarios para poder adoptarlo como el lenguaje buscado en este PFC, sobre todo por la ausencia de suficientes elementos gráficos para representar procesos mediante diagramas.

2.3.3 BPMN

La *Notación para la Gestión de Procesos de Negocio* BPMN [OMG,2010a], fue propuesto inicialmente por el grupo de trabajo BPMI, perteneciente a la organización OMG, en 2004. El principal objetivo de BPMN es proporcionar una notación que sea fácilmente comprensible por todas las partes implicadas en el negocio. De acuerdo con su propia definición "(BPMN) persigue crear un puente que cubra el vacío existente entre el diseño de procesos de negocio y su implementación".

Otro objetivo de BPMN es el de asegurar que el lenguaje XML empleado para la ejecución de los procesos, permita visualizar mediante unos diagramas de proceso orientados al negocio, de modo que permita una rápida implementación en lenguajes ejecutables y/o servicios web.

Es este último aspecto el que lo hace idóneo para el trabajo desarrollado, ya que los procesos que se pueden diseñar mediante BPMN tienen una transición muy fácil a código ejecutable. Esto es así debido a que las estructuras y elementos que proporciona son similares a los que se pueden encontrar en un programa informático (decisiones, bucles, recursividad, paralelismo, subprocesos, etc.).

La ultima especificación BPMN es la 2.0.2, publicada en 2013.

Se ha seleccionado esta notación concreta por cubrir todas las necesidades requeridas para poder implementar un lenguaje y motor propios a partir de sus elementos, ser sencilla de manejar, disponer de múltiples editores y tener la suficiente potencia y flexibilidad como para adaptarse a las necesidades del presente Proyecto Fin de Carrera. No en vano pasa por ser una de las más empleadas actualmente [Harmond, 2018].

En el siguiente apartado se detallan los distintos diagramas y elementos de diseño que ofrece el lenguaje BPMN.

2.4 BPMN

2.4.1 Diagramas

El lenguaje de notación BPMN define diversos modelos o diagramas para cubrir todas las facetas de diseño de los procesos de un negocio. Estos diagramas pueden combinarse, de modo que hay ciertos elementos que pueden formar parte de cualquier diagrama, en función de las necesidades del proceso.

A continuación se repasa brevemente cada tipo de diagrama, junto con un ejemplo de cada uno, y por último se mostrarán algunos ejemplos de diagramas mixtos, mas complejos.

2.4.1.1 Diagrama de proceso

Se trata de un diagrama que describe los pasos para realizar un proceso de negocio. Se define como un grafo dirigido o diagrama de flujo en el que quedan conectados diversos elementos funcionales, que aportan la semántica necesaria para la ejecución de un proceso.

Una de las cualidades de este diagrama es su granularidad, ya que permite definir desde un proceso muy amplio, a nivel de empresa, hasta un proceso muy concreto, que realiza una única persona.

En estos diagramas se usan los siguientes tipos de elementos:

- Actividades o Tareas, que indican una acción o trabajo a realizar.
- Compuertas, que permiten encauzar la ejecución del proceso en función de datos o eventos u otro tipo de lógica, o dividir la ejecución.
- Subprocesos: permiten abstraer parte del proceso y simplificar el diagrama.
- Eventos: modelan sucesos que ocurren en el proceso o que se reciben externamente.

Puede verse un ejemplo en la Figura 2, que representa la gestión de un pedido. El proceso comienza con un evento inicial, con el que se recibe un pedido (Actividad). El pedido luego se acepta por un empleado (Tarea humana), que puede decidir rechazarlo (Evento de cancelación). Si se acepta, se ejecutan dos tareas en paralelo, una para gestionar los elementos a enviar, otra para gestionar el envío propiamente dicho (paquetería, transporte). Cuando ambas acaban, un empleado debe revisar que todo esté bien, y el proceso termina.

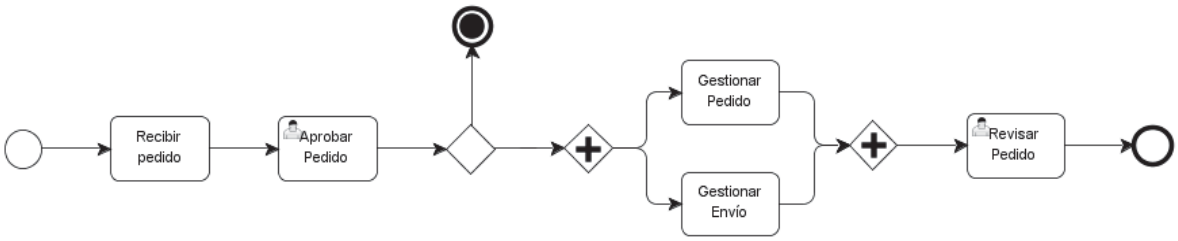


Figura 2: Ejemplo de proceso para la gestión de un pedido

Estos procesos pueden marcarse como ejecutables, o no ejecutables, en función de la granularidad o el propósito del mismo. Un proceso de muy alto nivel puede ser simplemente descriptivo y no tener el suficiente detalle como para poder ejecutarse tal cual.

Este tipo de diagrama es el seleccionado para representar los programas que se podrán diseñar, compilar y ejecutar en el entorno presentado en el presente PFC. Serán de grano muy

fino, de modo que todos los detalles necesarios para la ejecución del mismo puedan detallarse en el mismo.

Para facilitar el diseño de proceso complejos, estos diagramas pueden dividirse en subprocesos, los cuales se definen también con este tipo de diagrama. La notación además proporciona elementos específicos para invocar a los subprocesos. Puede verse un ejemplo en la Figura 3, similar al proceso de gestión de pedidos de la Figura 2, pero en la que la validación del pedido se ha decidido realizar mediante un subproceso, compuesto por dos tareas secuenciales.

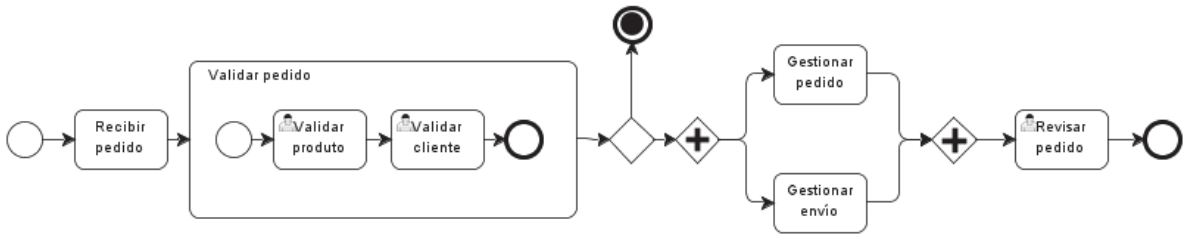


Figura 3: Mismo proceso anterior, invocando a un subproceso para validar el pedido

Por ultimo, los procesos pueden presentarse en estos diagramas compartimentados o divididos en calles, que permiten asociar grupos de actividades a entidades del negocio. De este modo parte del proceso puede asignarse a cualquier división jerárquica u organizacional del negocio, que asumirán la realización de dicha parte. Las calles pueden a su vez subdividirse en mas calles. La Figura 4 muestra un proceso con 2 calles principales, y a su vez una de ellas dividida en otras dos calles.

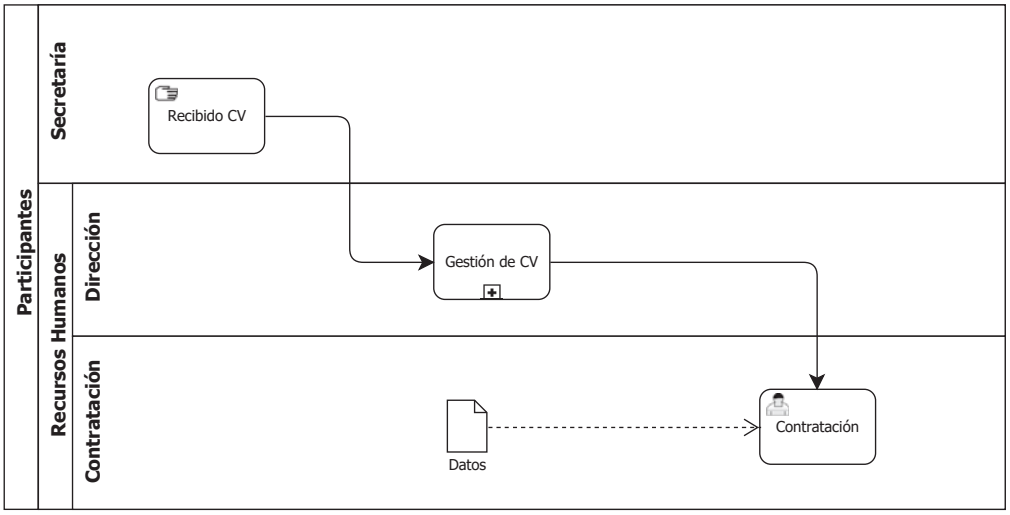


Figura 4: Proceso con diversas calles

2.4.1.2 Diagrama de colaboración

Es un diagrama que permite representa la colaboración o comunicación entre distintos procesos o actores del negocio.

Engloba a los diagrama de proceso anteriores y emplea conceptos como:

- Piscina: contenedor de procesos. Cada proceso independiente se representa en una piscina.
- Calle: cada uno de los compartimientos de las piscinas, en los que puede subdividirse un proceso, ya comentados en el apartado anterior.
- Mensajes: Los mensajes que se envían entre procesos distintos para compartir información y/o sincronizase.

Puede mostrarse detallado, con el contenido de los procesos y calles de cada piscina, como en la Figura 5, o bien simplificado, en el que solo se muestran las piscinas y las comunicaciones (paso de mensajes) entre ella, como en la Figura 6.

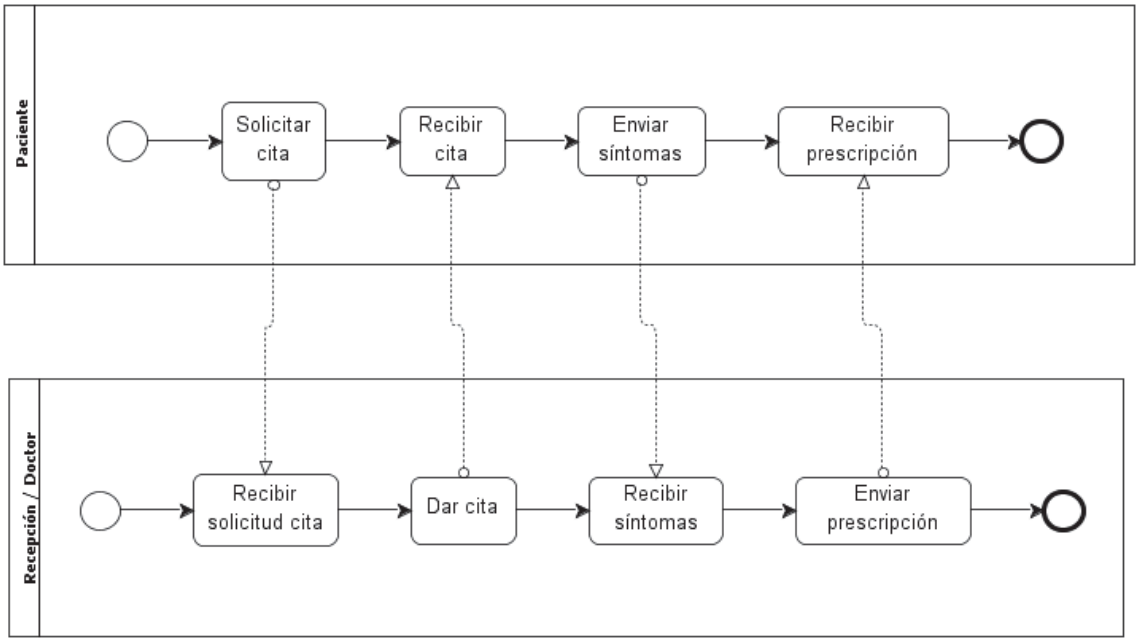


Figura 5: Diagrama de colaboración expandido

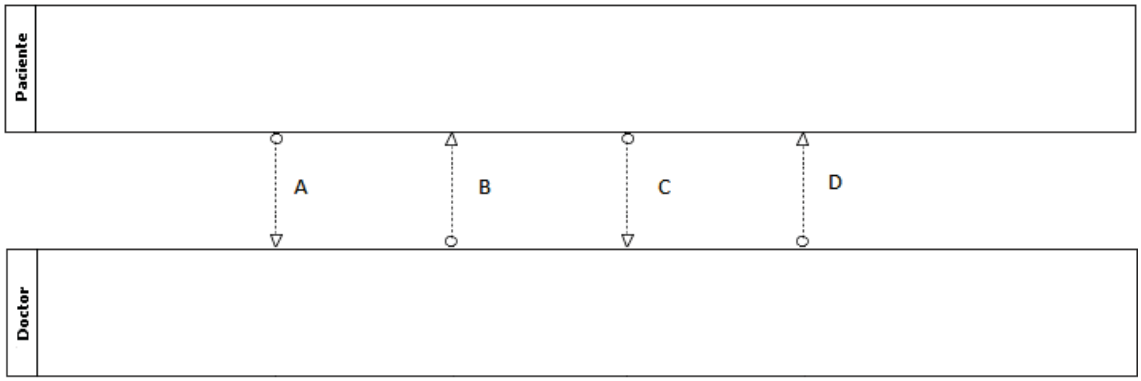


Figura 6: Diagrama de colaboración simplificado

2.4.1.3 Diagrama de coreografía

Estos diagramas representan las tareas realizadas por uno o varios participantes en el proceso, así como la secuencia de interacciones entre los mismos.

Son útiles para mostrar el intercambios de información y la definición de mensajes entre distintos procesos, sin entrar en el detalle de la implementación de las tareas ni de los procesos. Se intenta solo mostrar qué participantes intervienen, quién inicia la comunicación, y qué mensajes se pasan.

No se emplearán en el presente PFC.

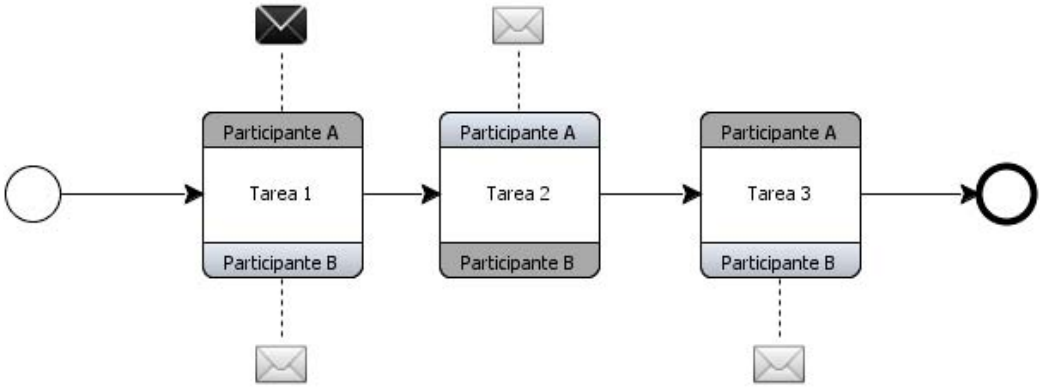


Figura 7: Ejemplo de diagrama de coreografía

2.4.1.4 Diagramas compuestos

Los diagramas presentados hasta ahora pueden combinarse en función de las necesidades del diseño, de modo que pueda detallarse adecuadamente la información a representar del

negocio. El modelo BPMN es un modelo flexible en ese sentido. En la Figura 8 se muestra un diagrama que combina elementos de los diagramas de procesos, colaboración y coreografía.

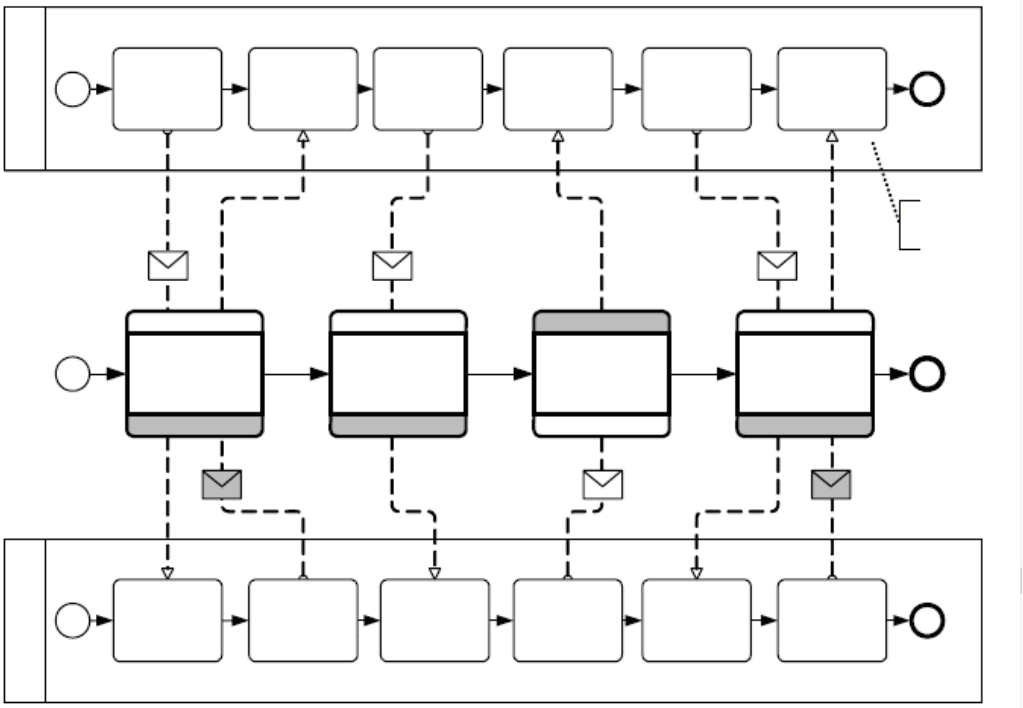


Figura 8: Ejemplo de diagrama compuesto

2.4.2 Elementos de diseño

A continuación se detallan todos los elementos de diseño que define la notación BPMN. Se muestran catalogados por tipo, mostrándose un resumen de ellos en la Tabla 2. La mayoría de ellos pueden usarse en cualquiera de los diagramas vistos anteriormente, pero sobre todo se emplean en los diagramas de Proceso, que son los son mas interesantes para el presente PFC.

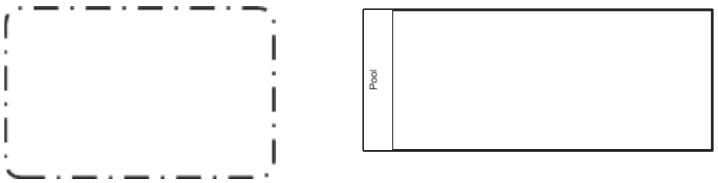
Actividades	
Eventos	
Compuertas	
Conectores	
Datos	
Otros	

Tabla 2: Algunos elementos BPMN

2.4.2.1 Actividades

Es la unidad básica que representar un **trabajo** o **tarea** a realizar dentro del proceso. Bajo esta categoría se engloban diversos tipos de actividades, teniendo en común que se representan por un rectángulo con las esquinas redondeadas.

Las actividades pueden dividirse en tres grupos: *Tareas*, *Subprocesos* y *Llamadas*. La Tabla 3 muestra las características comunes representativas de cada uno de ellos.

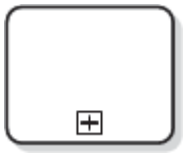
Tarea		• Borde normal
Subproceso		• Símbolo [+]
Llamada		• Borde grueso

Tabla 3: Tipos de Actividad

2.4.2.1.1 Tareas

Son actividades atómicas realizadas dentro del proceso. Se usan para indicar que no se puede descomponer la actividad en otras de grano mas fino.

La Tabla 4 muestra todos los tipos de Tareas que se definen, su representación gráfica y alguna de sus características.

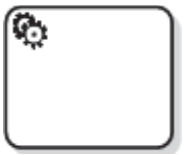
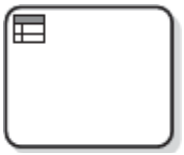
Abstracta		• No define ningún tipo de tarea específica
Llamada a servicio		• Realiza una llamada a un servicio. • Tiene datos de entrada y salida.
Envío de mensaje		• Define el envío de un mensaje a un participante del proceso. • Cuando el mensaje se envía, la tarea acaba.
Recepción de mensaje		• Define la recepción de un mensaje de un participante del proceso. • El proceso espera hasta que llega el mensaje. • Cuando el mensaje se recibe, la tarea acaba y el proceso continúa.
Tarea de usuario		• Define una tarea a realizar por un usuario dentro sistema. • Controlada por un gestor de tareas.
Tarea manual		• Define una tarea manual realizada fuera del sistema. • No hay seguimiento de la misma.
Regla de negocio		• Define la interacción con un motor de reglas de negocio. • Tiene datos de entrada y de salida.
Script		• Define la ejecución de cierto código dentro del sistema. • Tiene datos de entrada y de salida.

Tabla 4: Tipos de Tarea

2.4.2.1.2 Subprocesos

Son actividades complejas que pueden ser definidas por otras subactividades, conectores y compuertas mas específicas. Son útiles para agrupar funcionalidad de cara a ocultar detalles

no descriptivos, añadir transaccionalidad o gestionar eventos. Tienen como inconveniente que no se puede reutilizar por otros procesos.

Pueden encontrarse contraídos o expandidos. Si están contraídos, se representan por un rectángulo redondeando y un símbolo [+] en la parte inferior. Si están expandidos, puede verse el proceso que contienen. La Figura 9 muestra ambas representaciones.

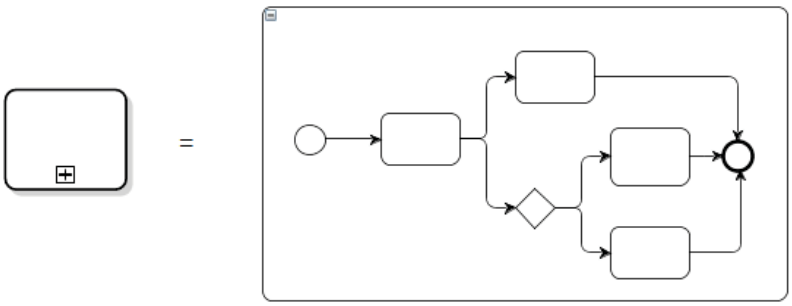


Figura 9: Un mismo subproceso contraído y expandido

La Tabla 5 muestra los tres tipos de subproceso que pueden encontrarse.

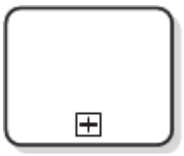
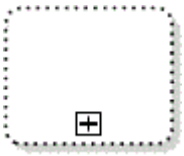
Subproceso		• Subproceso básico, con línea continua
Transacción		• Indica que el subproceso ha de ejecutarse dentro de una transacción • Línea doble continua
De evento		• Indica que el subproceso se ejecuta necesariamente a partir de un evento • Línea simple discontinua

Tabla 5: Tipos de Subproceso

2.4.2.1.3 Llamada

Son invocaciones a otras actividades globales, definidas fuera del proceso. Pueden invocarse tareas o procesos. Se representa mediante por un rectángulo redondeando con borde **grueso**.

Hay tantos tipos como tipos de tarea y subprocesos hay. Se representan igual que la actividad invocada, pero con el borde con línea gruesa. La Figura 10 recoge algunos ejemplos.

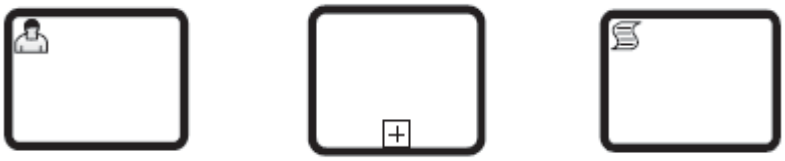


Figura 10: Ejemplos de llamadas

2.4.2.1.4 Marcadores

Son indicaciones adicionales para definir algún aspecto o funcionalidad añadida, y se pueden añadir únicamente a tareas y subprocesos. Se representa con símbolos adicionales que "decoran" la actividad marcada. Estos marcadores en algunos casos pueden combinarse.

2.4.2.1.4.1 Bucles

Indica que la actividad se repetirá mientras una condición asociada se evalúe como cierta. La Figura 11 muestra dos ejemplos de actividades realizadas en bucle, uno de una tarea abstracta, otro de un subproceso.

Figura 11: Actividades en bucle

2.4.2.1.4.2 Instanciación múltiple

Indica que se crearán múltiples instancias de la actividad. Existen dos tipos:

- **Secuencial**: La ejecución de las actividades se realizará de forma secuencial, una detrás de otra, y en un orden determinado. Se indica con tres lineas paralelas horizontales, como se puede ver en la Figura 12.

Figura 12: Actividades secuenciales

- **Paralela:** La ejecución de las actividades se realizará de forma paralela, todas a la vez. Se indica con tres lineas paralelas verticales, como se puede ver en la Figura 13.

Figura 13: Actividades paralelas

2.4.2.1.4.3 Compensación

Sirve para indicar aquellas actividades que son específicas para deshacer otras acciones ya realizadas en otras actividades previas. Se representa con dos triángulos blancos junto, con borde negro, apuntando a la izquierda, como puede verse en la Figura 14.

Figura 14: Actividades de compensación

Se activan con eventos específicos para este propósito, de modo que siempre se encontrarán asociados a ellos, como puede verse en la Figura 15. Se hablará de los eventos con mas detalle en los siguientes apartados.

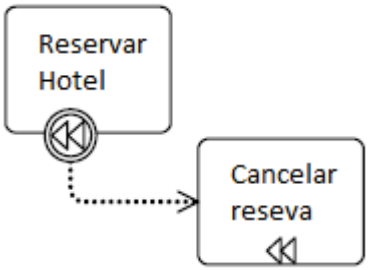


Figura 15: Ejemplo de compensación

2.4.2.1.4.4 Ad hoc

Indican que el proceso que contienen es un proceso no estándar. No tiene por que tener conectores ni eventos, pero sí actividades. Son en última instancia los responsables de las actividades los que deciden el orden y la lógica de la ejecución de las mismas.

Solo aplican a los *subprocesos.* Puede verse su representación gráfica en la Figura 16.

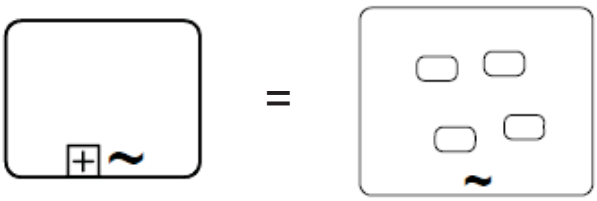


Figura 16: Subproceso ad hoc, contraído y expandido

2.4.2.2 Eventos

Un evento es algo que ocurre durante la ejecución de un proceso, e influye en el desarrollo del mismo. Debe tener un origen o destino bien definido, ya que el flujo del proceso dependerá de ellos, de donde estén ubicados, de cuando se reciben, de quien los emita, de los datos que lleven asociados, etc.

Se representan gráficamente mediante círculos, y la semántica asociada a los mismos es un poco mas compleja que las vistas hasta ahora, ya que es un elemento muy versátil y proporciona una funcionalidad muy variada.

A continuación se detallarán las características de los eventos divididas en varias clasificaciones, para facilitar su comprensión. Al final se mostrarán unas tablas resumen con todos ellos.

2.4.2.2.1 Características y clasificación

Los eventos tienen una serie de características que permiten clasificarlos de diversas maneras:

- Por su ubicación en el proceso: indicado por la linea exterior del círculo.

Inicial		• Indica un punto de entrada en el proceso. • Tiene que existir al menos uno. • Borde sencillo
Final		• Indica un punto de salida en una de las ramas de un proceso. • Tiene que existir al menos uno. • Borde grueso
Intermedio		• Indica algo ocurre entre el principio y el final de un proceso. • Borde doble

Tabla 6: Eventos clasificados por su ubicación en el proceso

- Por su tipo: indicado por el contenido del círculo.

Mensaje		• Asociado a un mensaje entrante o saliente.
Temporizador		• Asociado al vencimiento de un temporizador.
Condicional		• Asociado a una condición lógica.
Señal		• Asociado a la difusión o notificación de un suceso.
Múltiple		• Indica que hay varias maneras de activar este evento, pero solo una de ellas es necesaria (OR).
Múltiple paralelo		• Indica que hay varias maneras de activar este evento, y hacen falta todas ellas para activarlo (AND).
Escalado		• Para comunicar sucesos fuera de los límites de un subproceso.
Compensación		• Para tomar acciones encaminadas a deshacer actividades previamente realizadas.
Enlace		• Evento especial no semántico • Sirve para enlazar dos diagramas fraccionados.
Cancelación		• Especifico para comunicar la cancelación de subprocesos transaccionales.
Fin		• Evento especial para terminar la ejecución de <u>todas</u> las ramas del proceso.

Tabla 7: Eventos clasificados por su tipo

- Por su comportamiento síncrono: indicado también por la linea exterior.

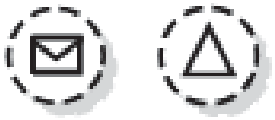
Bloqueante		• Cualquier evento con el borde continuo. • En subprocesos los eventos iniciales de este tipo bloquean al padre hasta que el proceso hijo termina
No bloqueante		• Cualquier evento con el borde continuo. • En subprocesos los eventos iniciales de este tipo no bloquean al padre, que continua sin esperar al proceso hijo.

Tabla 8: Eventos clasificados por su comportamiento síncrono

- Por el sentido de la comunicación: indicado por el color de relleno del círculo.

Recepción		• Para eventos iniciales o intermedios. • Implica esperar la llegada de dicho evento. • El color de relleno del contenido del evento es *blanco.*
Emisión		• Eventos intermedios o finales • Implica emitir el evento correspondiente • El color de relleno del contenido del evento es *negro*

Tabla 9: Eventos clasificados por el sentido de la comunicación

En la Figura 17 pueden verse todas las combinaciones posibles de las características comentadas, dando muestra de todo los posibles eventos que pueden encontrarse en los diagramas BPMN. Allá donde no hay evento en la celda correspondiente es porque no tiene sentido la combinación.

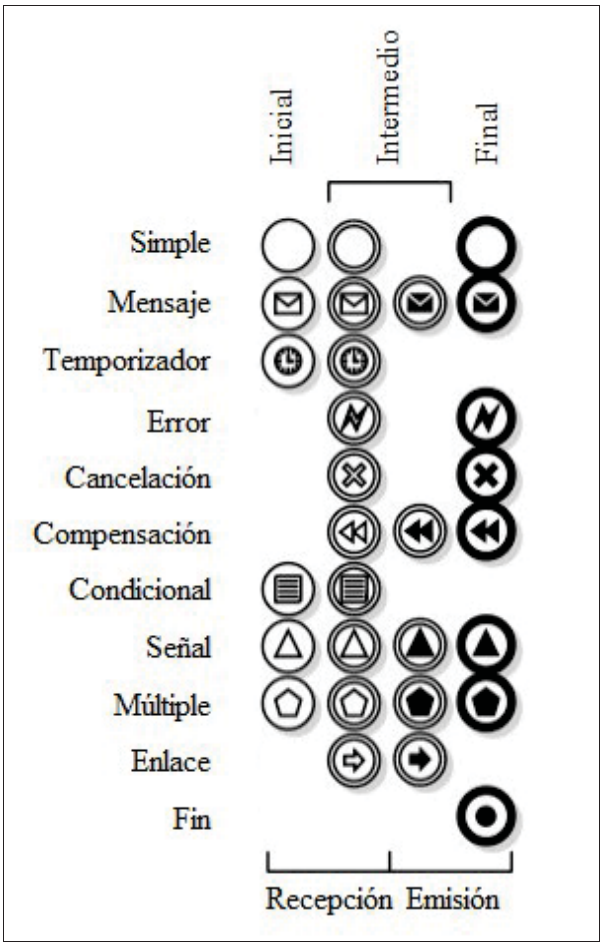


Figura 17: Resumen de eventos

2.4.2.2.2 Eventos asociados a actividades

Todos los eventos anteriores pueden aparece o bien aislados, indicando que el proceso espera por o envía un evento en ese punto, o bien asociado a una tarea concreta, en cuyo caso aparecerán en el borde de la misma, como puede verse en la Figura 18. Sirve para poder gestionar los eventos lanzados desde dentro de la tarea, como puedan ser errores, vencimiento de algún temporizador o algún tipo de escalado. Ya que una actividad puede generar distintos tipos de evento, pueden asociarse uno varios eventos al borde de la misma.

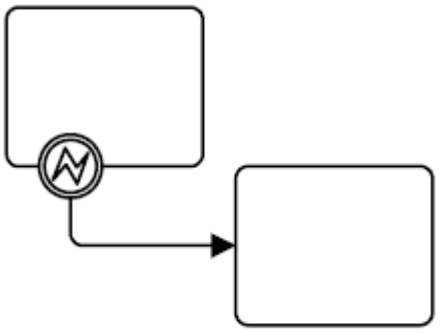


Figura 18: Evento de error asociado a una tarea

Los eventos también pueden aparecer en el borde de un subproceso, ya sea expandido o contraído, como se muestra en el ejemplo de la Figura 19, donde se gestionan dos eventos distintos que puede lanzar el subproceso de Envío.

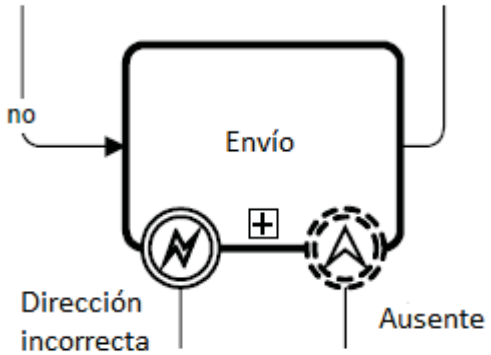


Figura 19: Dos eventos asociados a un subproceso

Si el proceso esta contraído y comienza con un evento, se indica con linea discontinua y el evento que lo lanza incrustado dentro, no en el borde, como se muestra en la Figura 20.

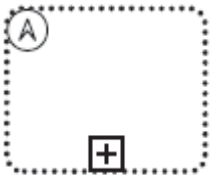


Figura 20: Subproceso iniciado por un evento

2.4.2.3 Compuertas

Se usan para controlar los distintos caminos o ramas de ejecución de un proceso, que pueden incluso paralelas. En función de su tipo servirán para dividir, dirigir, contraer o seleccionar los distintos flujos de ejecución.

Se representan mediante un rombo y su contenido determina la funcionalidad de la compuerta.

2.4.2.3.1 Exclusiva

Sirve para tomar una decisión en función de la condición que contengan sus distintas transiciones de salida. Solo se tomará una de las transiciones de salida, por lo que las decisiones deben ser excluyentes entre si.

Puede representarse con un rombo sin contenido o con una X. Ambas representaciones son equivalentes. Se muestran sendos ejemplos en las figuras 21 y 22.

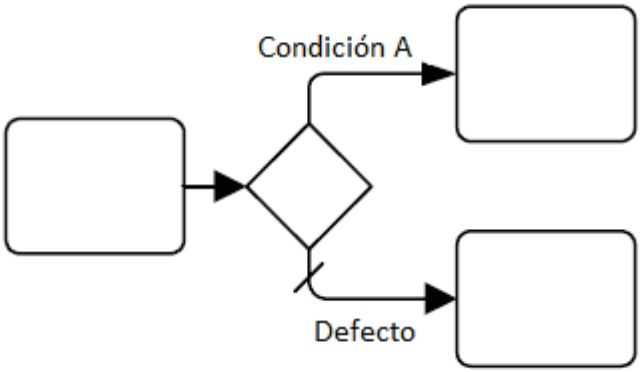


Figura 21: Compuerta exclusiva sin contenido

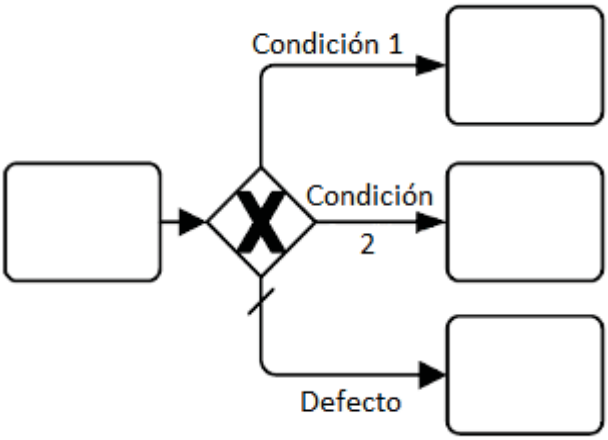


Figura 22: Compuerta exclusiva con X

2.4.2.3.2 Inclusiva

Sirve también para tomar una decisión en función de la condición que contengan las distintas transiciones de salida. A diferencia de la compuerta exclusiva, se tomarán *todas* las transiciones de salida cuya condición se cumpla, por lo que esta compuerta puede dividir la ejecución en varias ramas que paralelas.

Se denota con un rombo cuyo contenido es una circunferencia, como puede apreciarse en la Figura 23.

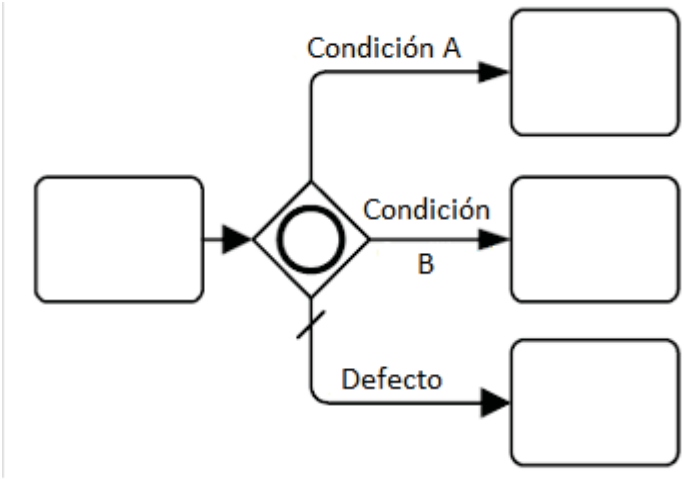


Figura 23: Compuerta inclusiva

2.4.2.3.3 Paralela

Esta compuerta se usa para crear o terminar ejecuciones en paralelo dentro del proceso. Se equipararía a una estructura *fork* / *join* de programación. Ambas funciones se identifican con la misma notación, un rombo con un símbolo + en el interior. No define ninguna condición.

Opcionalmente puede omitirse el rombo y simplemente mostrar dos flujos salientes de una actividad, como puede verse en el ejemplo la Figura 24, que muestra el uso de esta compuerta para dividir la ejecución en sus dos formas posibles.

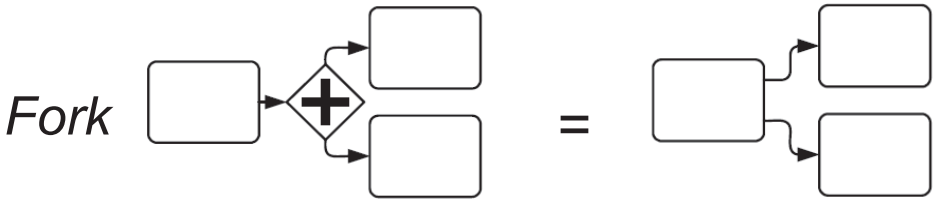


Figura 24: Compuerta paralela dividiendo la ejecución

Cuando funciona como *join* la ejecución del proceso no continua hasta que **todas** las ramas de ejecución esperadas confluyan hacia la compuerta, de modo asíncrono. Es decir, no hace falta que lleguen a la vez. Para usar esta compuerta requiere que previamente se haya pasado un *fork* o algún otro elemento que divida la ejecución (por ejemplo, una compuerta inclusiva).

En el Figura 25 pude verse un ejemplo de uso, en el que la actividad de la derecha no se ejecutará hasta que las dos actividades de la izquierda se no hayan completado (cada una por su) y llegado a la compuerta paralela.

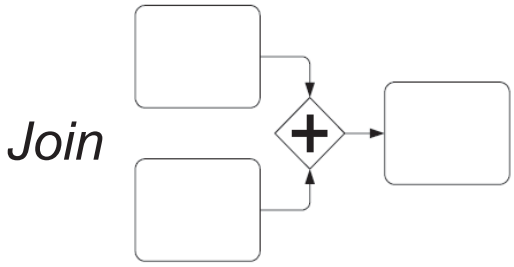


Figura 25: Compuerta paralela uniendo dos ejecuciones

2.4.2.3.4 Basada en eventos

Estas compuerta sirven para tomar decisiones en función de la recepción de eventos. Las transiciones de salida deben llegar a eventos o tareas relacionadas con eventos. En la Figura 26 se muestra un ejemplo de una compuerta que espera por 3 eventos distintos, o bien el vencimiento de un temporizador de 1 día, o bien que se reciba el mensaje 1 o el 2.

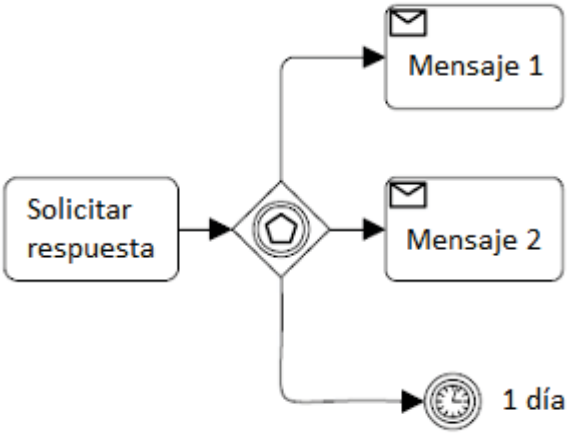


Figura 26: Compuerta basada en eventos

En función del contenido del rombo, puede tratarse de una compuerta exclusiva o paralela.

Si es exclusiva, cuando alguno de los eventos se recibe, la ejecución continua por la rama

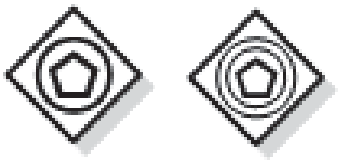


Figura 27: Compuerta exclusiva basada en eventos

correspondiente, y el resto de ramas ya no pueden activarse. Se representa mediante un circulo (evento) con un pentágono, como puede verse en la Figura 27.

Si es paralela, el primer evento no impide seguir escuchando el resto, de modo que sirve para poder dividir la ejecución en varias ramas paralelas. Se representa mediante un circulo (evento) con un símbolo + , como puede verse en la Figura 28.

Figura 28: Compuerta paralela basada en eventos

2.4.2.3.5 Compleja

Cuando ninguna de las compuertas anteriores cubre las necesidades del negocio, se usa una compuerta compleja. Esta compuerta contiene una expresión compleja que ayuda decidir que hacer, además puede tomar decisiones basadas en eventos y/o crear ejecuciones paralelas

Se representa con un rombo en cuyo interior hay con una cruz de ocho puntas, similar a un asterisco. Puede verse un ejemplo en la Figura 29.

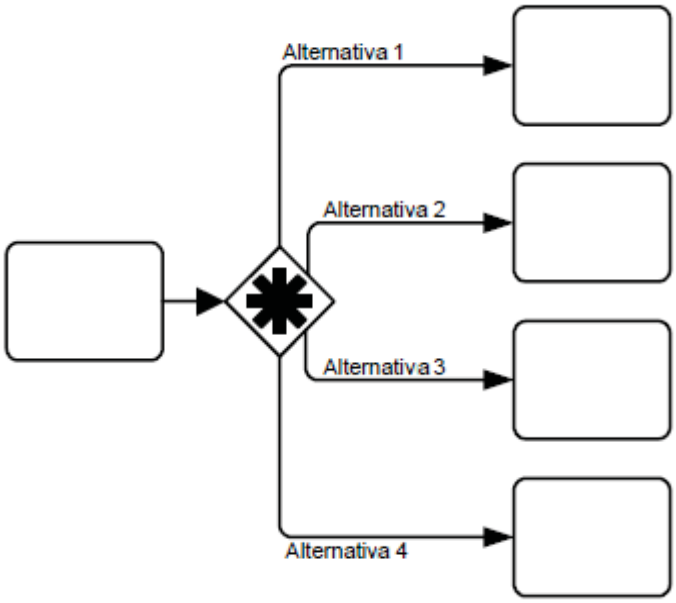


Figura 29: Compuerta compleja

2.4.2.4 Conectores

Son las líneas que sirven para conectar dos elemento distintos de un proceso BPMN. Cada tipo de conector tiene una grafía, una semántica y unas reglas de conexión distintas.

Se detallan a continuación los distintos tipos.

2.4.2.4.1 Transición

Si la línea es solida y en un extremo hay una flecha sólida, se trata de una transición. La punta de la flecha indica el destino. El otro extremo indica el origen. Hay tres tipos.

Normal

Figura 30: Transición normal

Tiene necesariamente un origen y un destino.

Condicional

Para realizar la transición, tienen que cumplirse una condición.

Figura 31: Transición condicional

Por defecto

Usada exclusiva y obligatoriamente como salida de compuertas de decisión que definen mas de una condición de salida. Si no se cumple ninguna, se toma esta transición por defecto.

Figura 32: Transición por defecto

2.4.2.4.2 Mensaje

Son conectores para indicar el envío de un mensaje. Se representa mediante una linea discontinua con un pequeño círculo en el extremo origen y un flecha sin relleno en el extremo destino.

Figura 33: Conector para envío de mensajes

Se usan exclusivamente dentro de los diagramas de colaboración (ver Figura 34). Tanto el origen como el destino tienen que estar preparados para enviar o recibir menajes.

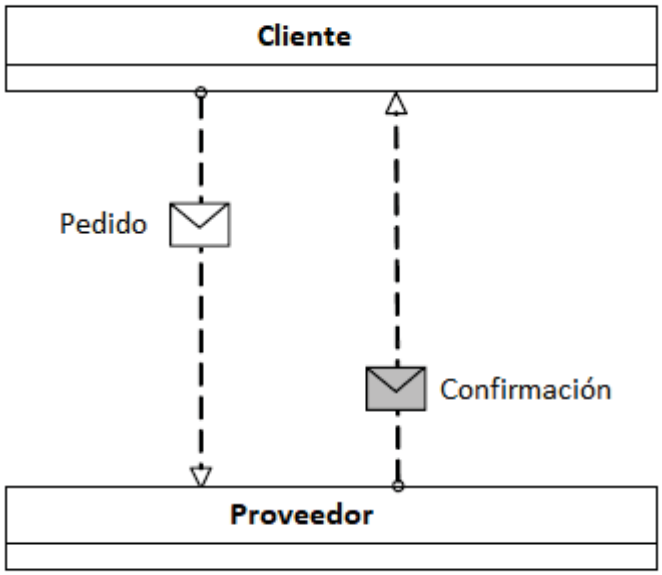


Figura 34: Diagrama de colaboración con envío de mensajes

2.4.2.4.3 Asociación

Sirven para conectar artefactos con elementos del proceso. Se representan mediante una linea punteada y una flecha en el extremo destino, opcional.

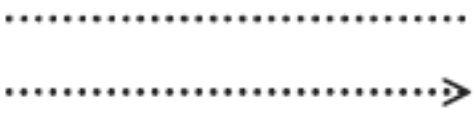


Figura 35: Conectores de asociación

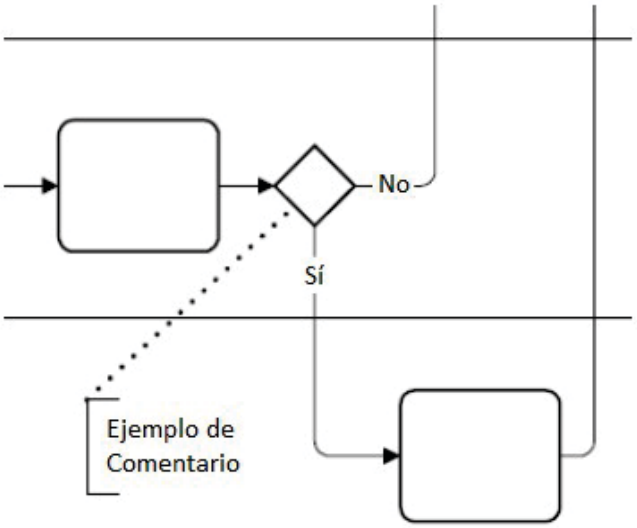


Figura 36: Una asociación entre un comentario y una compuerta

2.4.2.4.4 Datos

Son asociaciones que conectan elementos de datos con elementos del proceso. Se usan los mismos tipos de lineas que en las asociaciones.

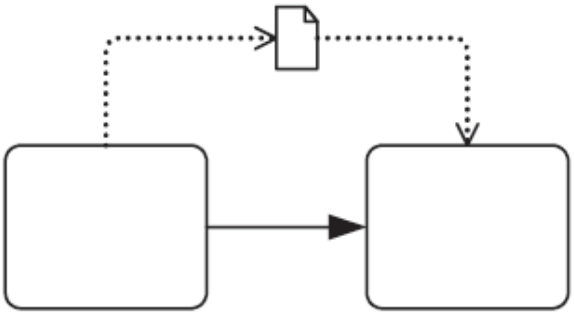


Figura 37: Ejemplo de conector de datos

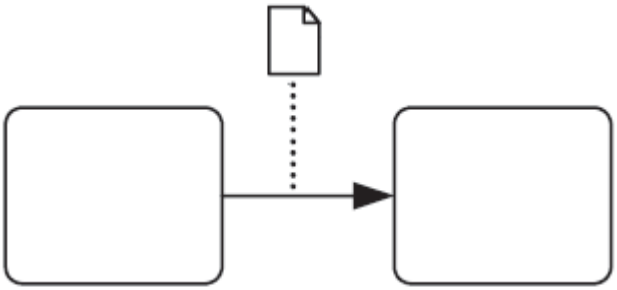


Figura 38: Otro ejemplo de uso de conector de datos

2.4.2.5 Datos

Son componentes que representan elementos genéricos de datos.

2.4.2.5.1 Objetos

Sirven para definir los datos que se usarán dentro de un proceso. Tienen un estado interno. Son compartidos por todos los elementos de un diagrama concreto. Pueden contener colecciones.

Figura 39: Objeto simple y colección, respectivamente

2.4.2.5.2 Entrada / Salida

Son datos que son usados específicamente como parámetros de entrada o salida en algunas actividades. También pueden definirse como colecciones.

Figura 40: Datos de entrada y salida, respectivamente

2.4.2.5.3 Almacenes

Representan el acceso a datos persistentes definidos fuera del alcance del propio proceso. Pueden ser accesos tanto de lectura como de escritura. Se representan con un cilindro.

En el proceso de la Figura 41 se pueden ver todos los tipos de elemento de datos descritos en los anteriores apartados.

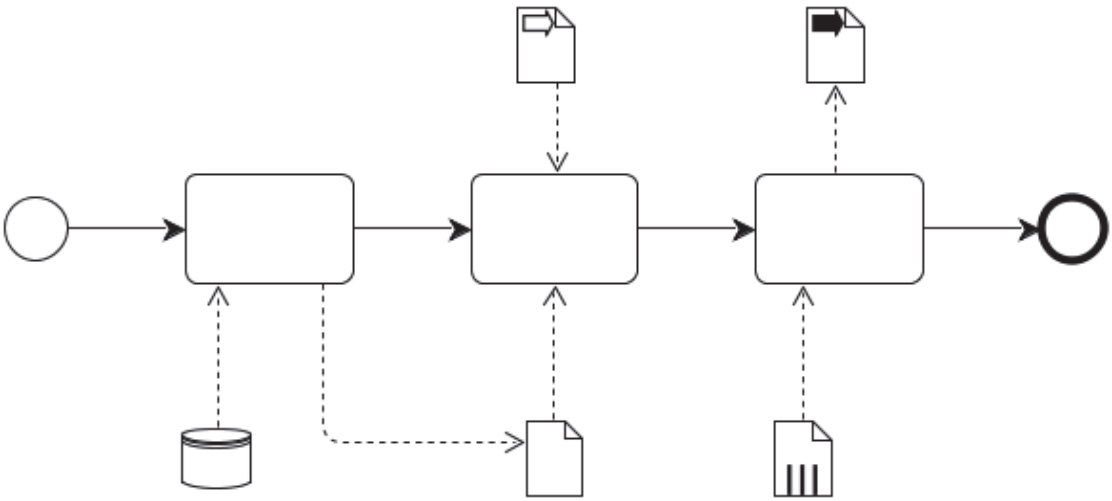


Figura 41: Proceso con elementos de datos

2.4.2.6 Otros elementos

Ademas de los elementos anteriores, que proporcionan un catálogo muy extenso que permite definir casi cualquier lógica imaginable, existen otros elementos que no entran dentro de las categorías descritas. Son piscinas, calles, anotaciones y grupos.

2.4.2.6.1 Piscinas y calles

Sirven para repartir las tareas a los distintos participantes en un proceso.

Las piscinas y las calles son compartimientos que representan a las entidades responsables de las actividades en un proceso (p.e. una organización, un rol o un sistema). Cada piscina y calle tendrá asignado un responsable y los elementos que estén dentro de su piscina o calle serán responsabilidad de dicho participante.

Las calles son un nivel inferior en la jerarquía, de modo que puede haber procesos solo con piscina sin calles o bien puede haber una piscina con varias calles, de modo que la piscina actúa como agrupador jerárquico de los dos responsables asociados a las calles.. La Figura 42 muestra un diagrama de colaboración con dos procesos definidos mediante piscinas, una sin calles (Cliente) y otra (Comercio) con dos (Mostrador y Almacén).

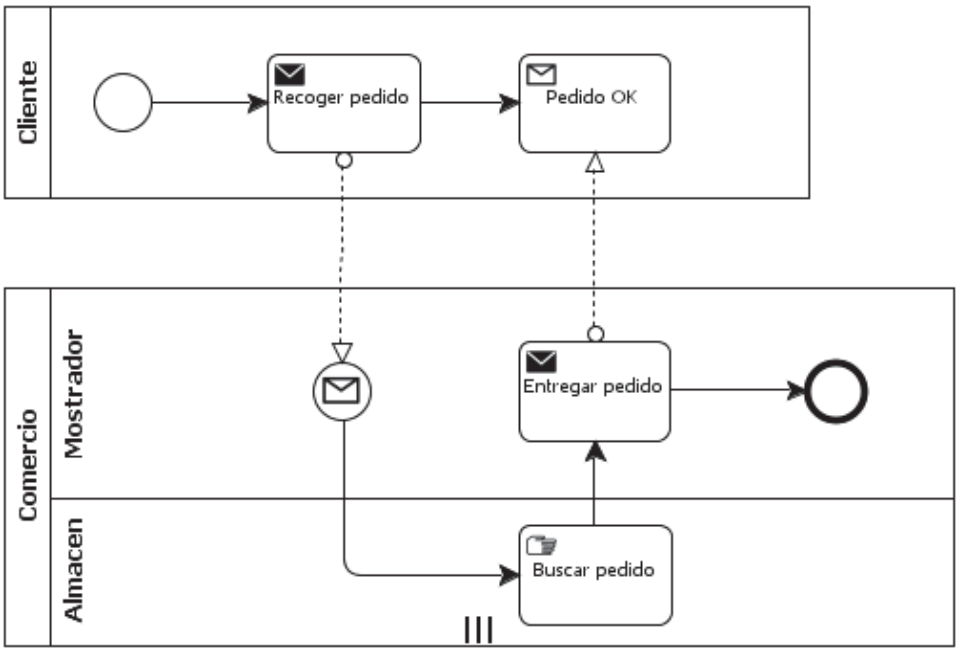


Figura 42: Procesos con piscinas y calles

2.4.2.6.2 Artefactos

Son elementos no semánticos que permiten dotar de información adicional a los diagramas para facilitar su comprensión.

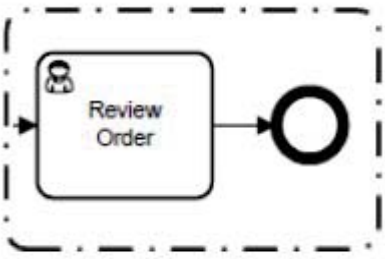


Figura 43: Grupo

Grupos

Permiten agrupar los elementos. Se usa una línea discontinua con puntos y rayas.

Anotaciones de texto

Son acotaciones textuales que se pueden asociar cualquier elemento.

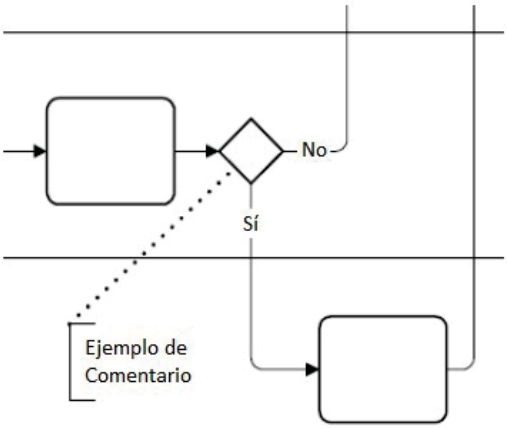


Figura 44: Un comentario a una compuerta

2.4.3 Implementaciones conocidas

Desde su primera versión 1.0 hasta la actualidad, han sido múltiples las implementaciones comerciales de la especificación BPMN, con diverso grado de relevancia y éxito. Con la especificación 2.0 las soluciones se multiplicaron, y ahora el mercado parece haber explotado, no existiendo quien no aporte su propia herramienta de modelo y ejecución de procesos BPMN. Algunos de las mas conocidas pueden encontrarse en [Wikipedia,2018a], aunque un simple rastreo más pormenorizado permite encontrar alguna más que no aparece en la lista, como [Bosch,2018] o [Joget,2018].

De las implementaciones que aparecen en [Wikipedia,2018] se destacan a continuación 4 de ellas, por ser las que tienen un grado de madurez mayor que el resto, y por tanto, son casos de éxitos mas probados. Se han descartado aquellas con desarrollos estancados o menos activos, o aquellas que simplemente son complementos de otras soluciones mas extensas y con otro propósito. La Tabla 10 muestra un resumen actualizado de [Wikipedia,2018] con los detalles de las mismas.

Nombre	Versión	Fecha	Lenguaje	Plataforma	Licencias
jBPM	7.8.0	07/2018	BPMN 2.0	Java EE	Apache Software License 2.0
Activiti	6.0.0	05/2017	BPMN 2.0	Java	Apache Software License 2.0
Bonita BPM	7.7.2	07/2018	BPMN 2.0	Java	LGPL v2, GPL v2, Proprietary
Bizagi	11.1	06/2017	BPMN 2.0	Java EE y .NET	Propietaria

Tabla 10: Algunas de las implementaciones conocidas de BPMN

2.4.3.1 jBPM

jBPM, de Red Hat JBoss [RedHat,2018], es una plataforma que cubre la especificación BPMN 2.0 y proporciona además otros elementos adicionales que hacen de ella una solución muy usada actualmente. Incluye entre otras cosas, un editor de procesos, un motor BPMN, modeladores de datos y formularios de usuario, un gestor de instancia de procesos y monitorización. Además permite la integración con otros sistemas, como motores de reglas de negocio (sistemas de toma de decisiones) o *scripting* (automatización).

Se trata de una plataforma muy potente y de por sí ya incluye todo lo necesario para realizar proyectos tanto de pequeña como de mediana o gran escala. No obstante, al ser un producto de código abierto, muy bien documentado y con APIs muy completas y extensas, y con actualizaciones frecuentes (mantenido en parte por la comunidad de desarrolladores), se convierte en un candidato ideal para proyectos en los que se requiera extender de modo particular la funcionalidad básica proporcionada por el motor. Es un motor BPMN 2.0 completo, y a la vez una plataforma de desarrollo.

La Figura 45 muestra un esquema general de la arquitectura modular de la plataforma jBPMN, compuesta por herramientas web, de desarrollo, servicios y el motor de procesos.

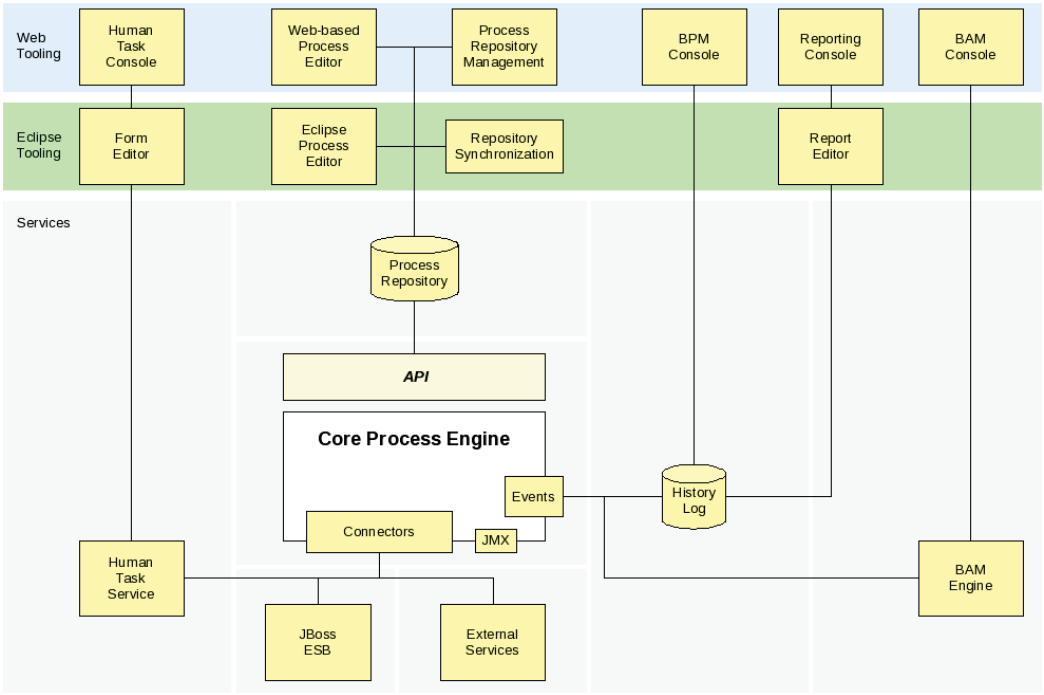


Figura 45: Arquitectura de la plataforma jBPM

2.4.3.2 Activiti BPMN

Una solución también de código abierto con gran aceptación es Activiti BPMN 2.0 Platform [Activiti, 2018], desarrollado por Alfresco Software, Inc, e iniciada por antiguos miembros del proyecto jBPM – aunque no es una rama de dicho proyecto, comenzaron de cero.

Posee un editor de procesos integrado dentro de la plataforma de desarrollo Eclipse, y otro disponible en plataformas web (Signavio), un motor de procesos y una herramienta de gestión procesos y usuarios.

Al igual de jBPM es una solución muy flexible y puede utilizarse como núcleo para otros desarrollos, gracias a las APIs que ofrece, utilizarse solo como gestión de conocimiento de procesos de negocio mediante BPMN, o bien como solución completa de servidor, integrada dentro de otros sistemas o en la nube.

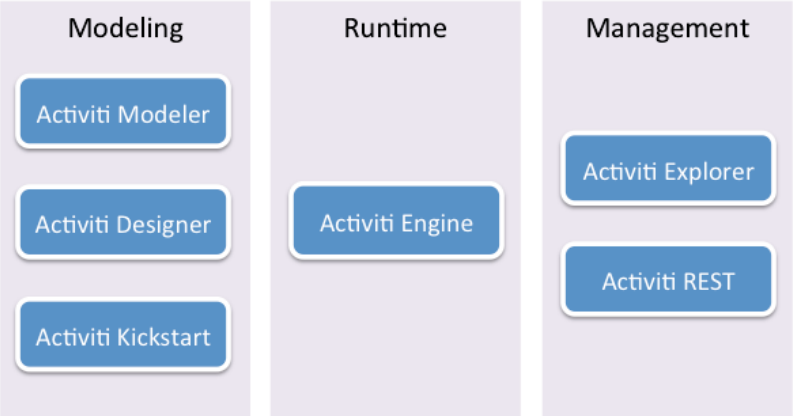


Figura 46: Componentes de la plataforma BPMN Activiti

2.4.3.3 Bonita BPM

Bonita BPM [Bonitasoft,2018], desarrollado por Bonitasoft, es otra solución muy completa y con gran implantación en entornos comerciales. Tiene dos versiones, Community, de código abierto y licencia Apache, y otra comercial.

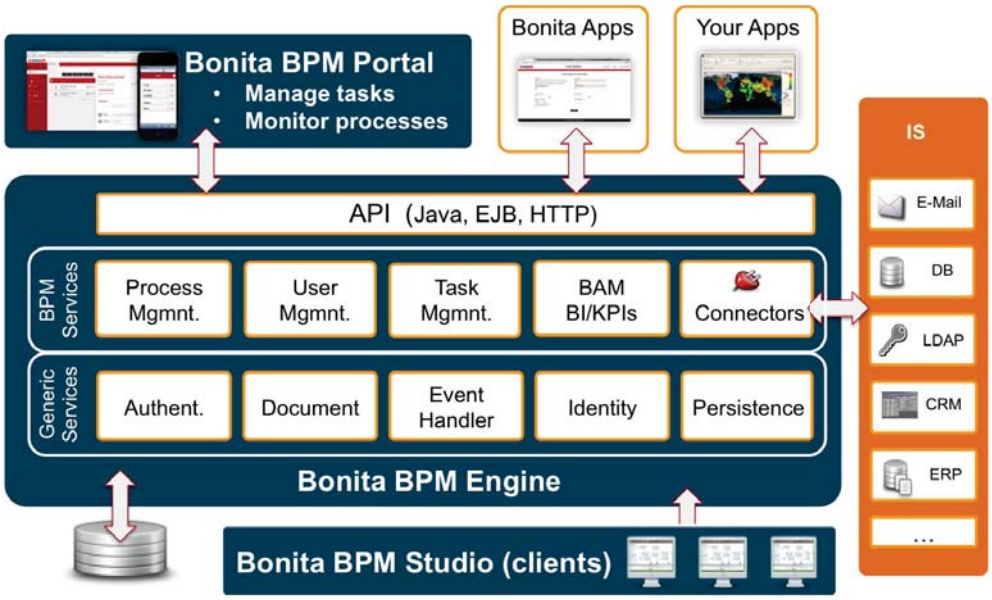


Figura 47: Arquitectura de la plataforma Bonita BPM 6

Consta como el resto de un editor completo BPMN y de un motor de procesos donde ejecutarlos, además de un portal de administración donde gestionar usuarios, instancias de proceso, etc.

2.4.3.4 Bizagi BPM

Por ultimo se muestra una solución puramente comercial, Bizagi [Bizagi, 2018]. aunque ofrece dos herramientas de modo gratuito, no siendo ninguna de ellas de código abierto. El motor de procesos es propietario.

Las herramientas gratuitas son *Bizagi BPMN Modeler* (editor de procesos BPMN), y *Bizagi Studio*, donde se pueden diseñar y construir aplicaciones basadas en los procesos, añadiendo funcionalidad como interfaces de usuario, formularios, reglas de negocio, pero sin ser ejecutables. *Bizagi Engine*, producto comercial, es el que se encarga de poner en funcionamiento los programas-procesos diseñados y construidos con las otras dos aplicaciones.

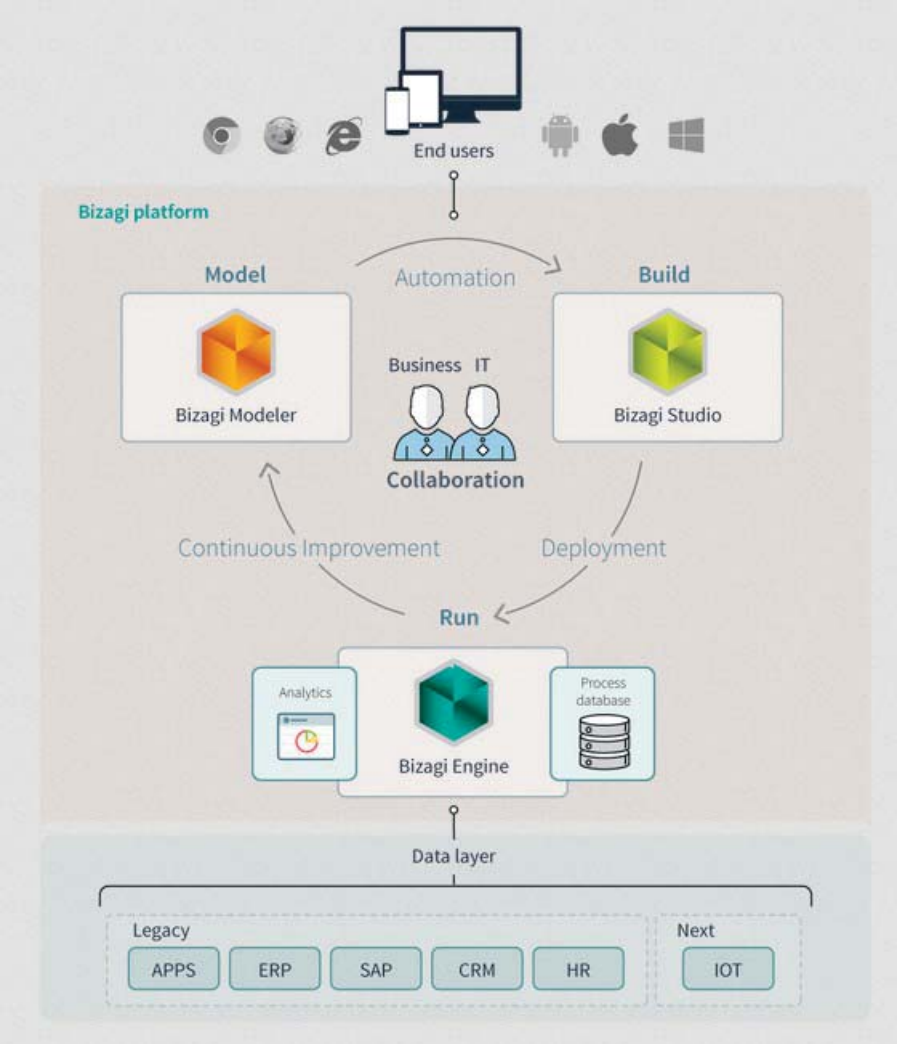


Figura 48: Arquitectura de la plataforma Bizagi

2.4.4 Editores BPMN

El catálogo disponible de editores BPMN es amplio. Hay editores más o menos logrados, que cubren más o menos la especificación completa de BPMN 2.0, o que añaden más o menos contenido especifico.

Ademas de las plataformas completas BPMN vistas en el apartado anterior, las cuales ya traen incorporado un editor BPMN propio (normalmente adaptado a la solución particular de dicha plataforma), existen en el mercado soluciones únicamente pensadas para diseñar diagramas BPMN.

Se han probado varias de ellas de cara estudiar cual sería un candidato ideal para poder editar los procesos que se diseñen en el sistema propuesto en el presente PFC.

2.4.4.1 Yaoqiang BPMN Editor

La aplicación elegida ha sido Yaoqiang BPMN Editor, porque es de los que mejor cubre la especificación BPMN 2.0 (en cuanto a diagramas de proceso se refiere). Además, no añade ningún contenido extra no compatible, y es de código abierto, lo cual la hace ideal para

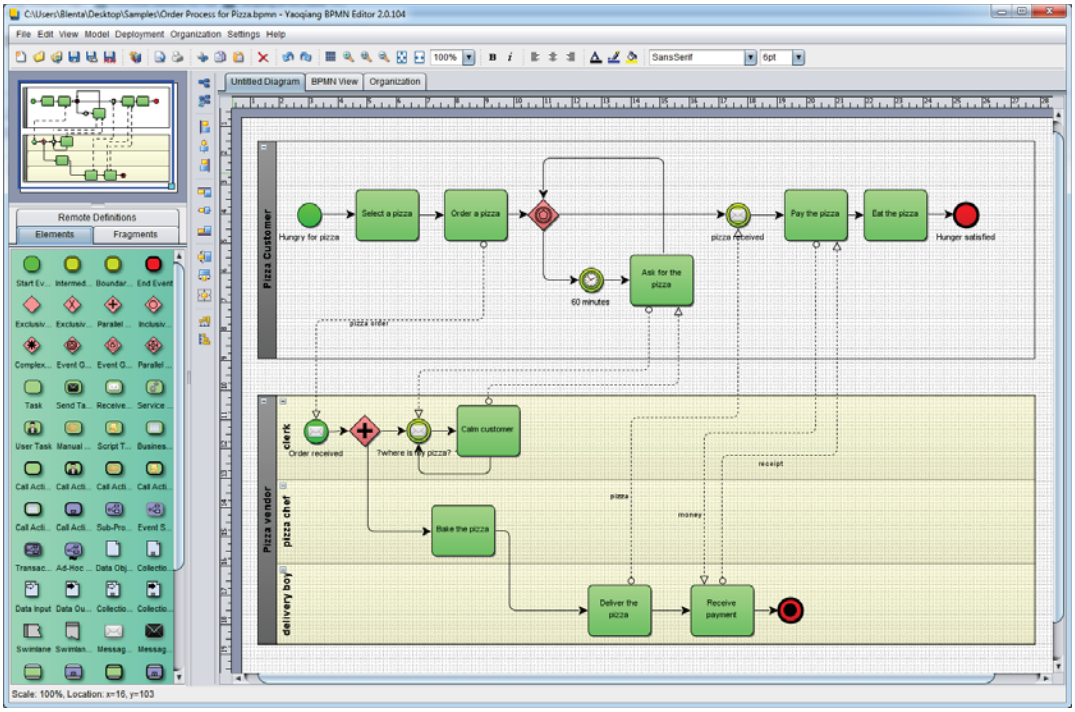


Figura 49: Vista general de Yaoqian BPMN 2.0 Editor

futuras ampliaciones. Pero sobre todo, se ha elegido porque permite editar directamente el código fuente que genera, lo cual es ideal para el propósito del sistema diseñado.

2.4.4.2 Camuda BPMN Online editor

Editor web, permite crear y guardar diagramas BPMN con facilidad desde cualquier navegador, sin mas requisitos [Camuda, 2018a]. De fácil manejo e intuitivo (si se conocen los elementos BPMN), es un buen punto donde diseñar y compartir rápidamente diagramas BPMN. Ofrece ademas una librería Javascript muy completa para desarrollos web.

Tiene como inconveniente que no permite editar las propiedades de cada componente ni editar el código BPMN (XML) generado. Hay que guardarlo y editarlo posteriormente.

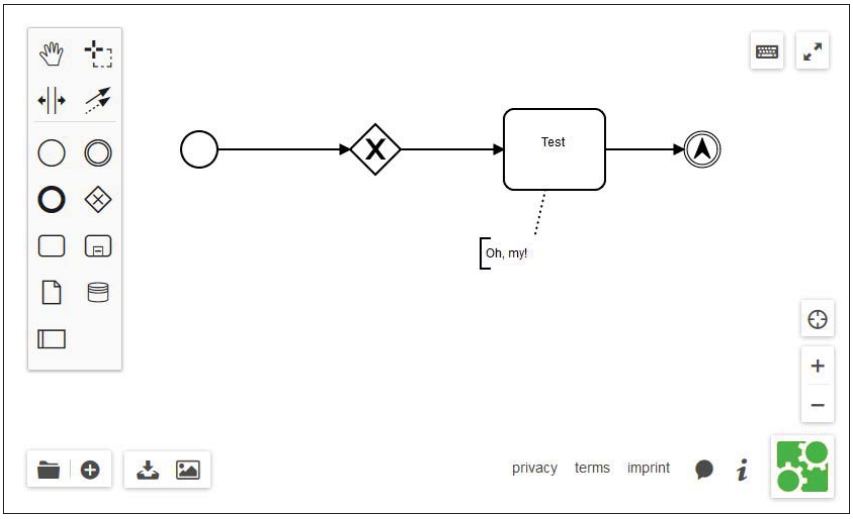


Figura 50: Camuda BPMN Online Editor

2.4.4.3 Eclipse BPMN2 Modeler

Se trata de un plug-in del entorno de desarrollo Eclipse [Eclipse,2018], que añade nuevas vistas y herramientas para poder crear diagramas BPMN. Algo mas limitado a la hora de diseñar los procesos, ya que no es demasiado intuitivo. Sí permite editar el código fuente y diseñar subprocesos fácilmente.

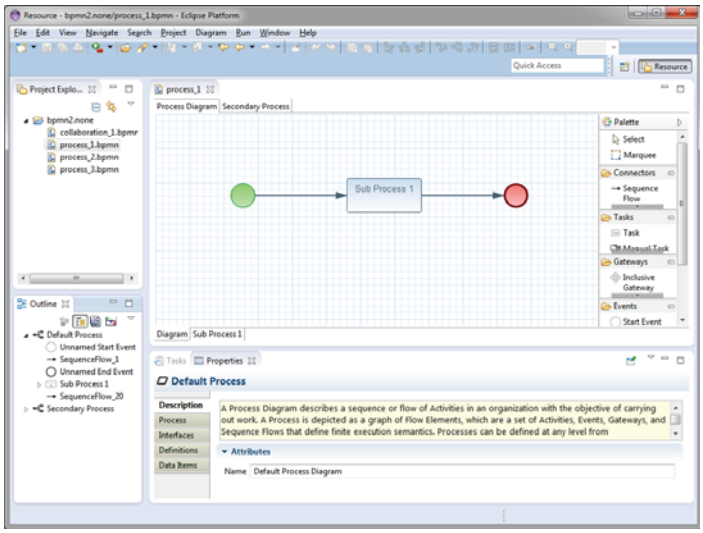


Figura 51: Vista general de BPMN2 Modeler

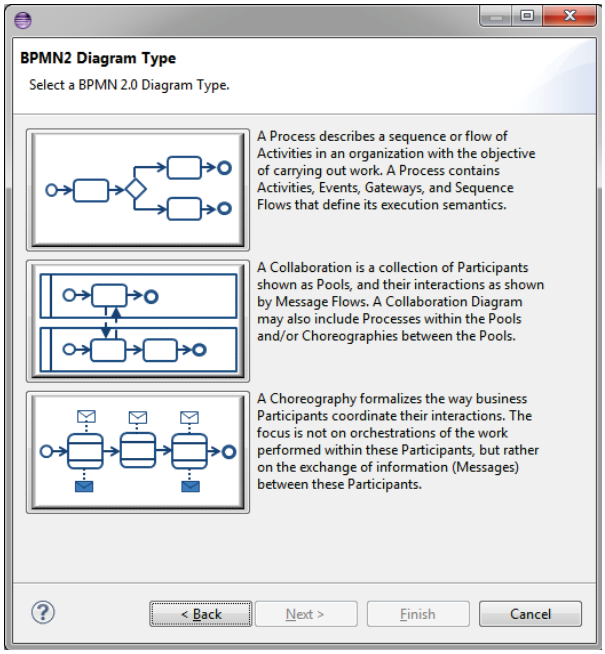
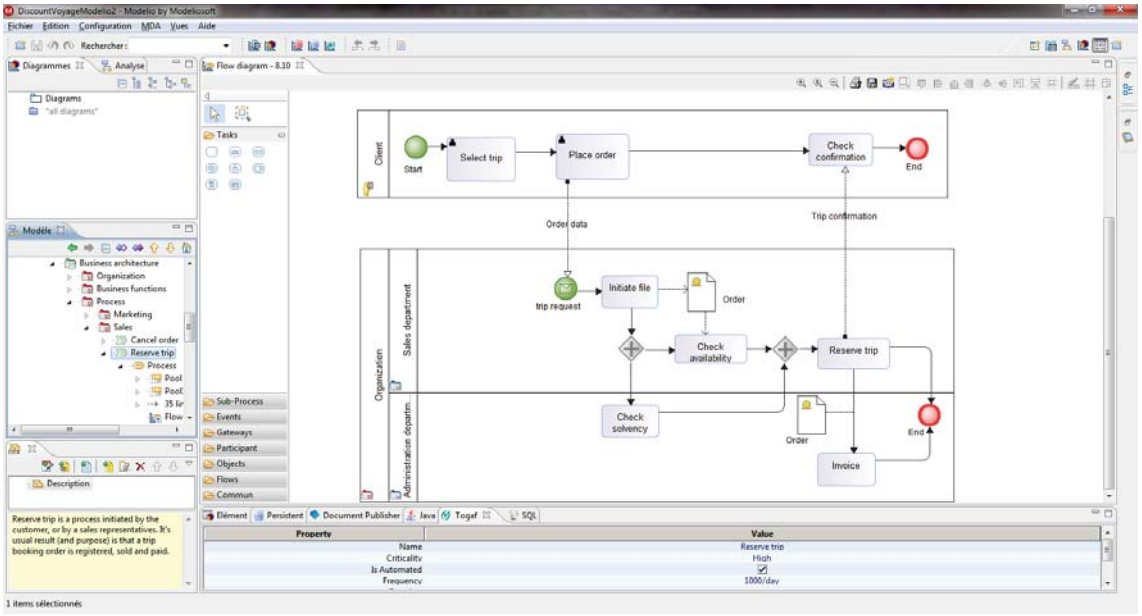


Figura 52: Selección de tipo de diagrama en BPMN2 Modeler

2.4.4.4 Modelio

Solución de carácter más general, ya que permite realizar más tipos de diagramas, no sólo BPMN, como UML o MDA. Basado en la plataforma Eclipse, Modelio [Modeliosoft,2018] es una solución muy completa. Su único inconveniente es que el código que genera da algún problema de formato al realizar el análisis sintáctico, ya que añade atributos y elementos adicionales, por lo cual se ha descartado su uso.



2.5 Tecnologías, Lenguajes y Estándares Empleados

2.5.1 BPMN

Como se ha visto con detalle en el apartado anterior, se ha decidido emplear este lenguaje de notación BPMN como base para diseñar los programas que se usarán como código fuente para resto del sistema. En concreto, se emplean los **diagramas de proceso** BPMN de la versión 2.0.

Los componentes BPMN utilizados, permitidos para desarrollar programas se detallan y justifican en el apartado ??.

2.5.2 Java

Desarrollado en origen por Sun Microsystems, y ahora bajo el auspicio de Oracle, por Java [Oracle,2018] se conoce tanto como a un lenguaje de programación orientado a objetos, como a la maquina virtual que permite ejecutar programas ejecutados en dicho lenguaje. Una de sus principales ventajas es que se trata de un lenguaje multiplataforma, lo que permite, dado un código objeto ya compilado, poder ejecutarlo en cualquier entorno en el que pueda correr una maquina virtual Java, ya sea Windows, Linux, UX, etc.

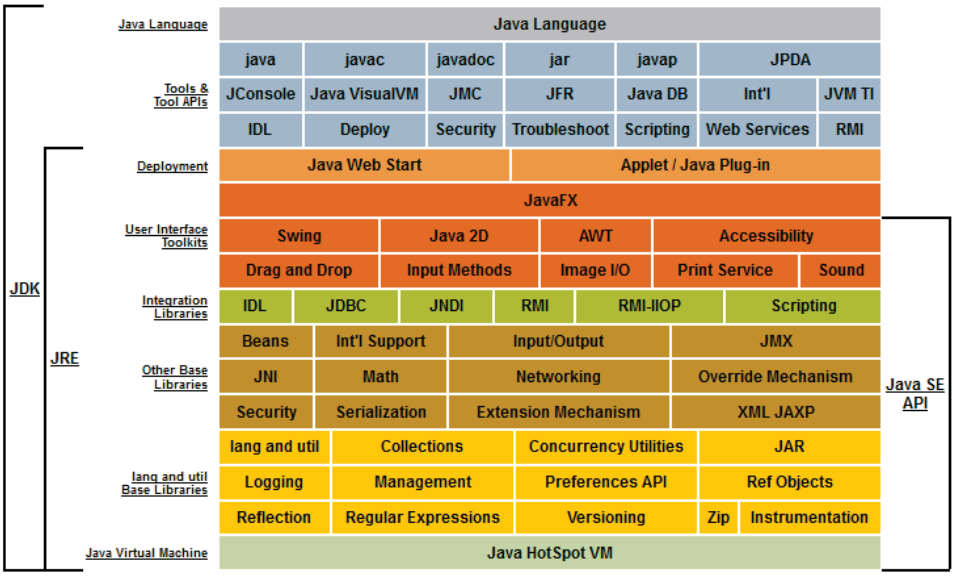


Figura 53: Componentes de las plataforma Java SE

Otra de las ventajas de usar Java es la de partir de un núcleo base muy completo, con mucha funcionalidad ya disponible *out-of-the-box,* sin más que instalar el *software*.

En la Figura 53 se muestra un diagrama con los distintos componentes de la arquitectura de Java, desde el lenguaje hasta la máquina virtual.

Por ultimo, cabe destacar la gran aceptación de Java tanto en la comunidad de desarrollo *software* como en el mercado, lo que permite disponer con facilidad de una miríada de código ya disponible en forma de bibliotecas o *software*.

La ultima versión de Java disponible a fecha hoy es la 10, en concreto la 10.0.1, si bien para la realización de este proyecto la especificación Java empleada ha sido la 7. La migración a Java

8 se descartó ya que ciertas librerías son compatibles con esa versión Java. La versión java 10 es demasiado reciente. Otra de las ventajas de hacerlo en Java 7 es mayor compatibilidad hacia atrás: Java 8 incluye nuevas sintaxis que impedirían ejecutar los programas en sistemas con Java 7.

2.5.2.1 Tipos de archivo manejados

- `Java`: Son los ficheros de código fuente, son su sintaxis particular. Puede haber mas de una clase definida dentro de un mismo proceso.
- `Class`: Son los ficheros de código objeto generados al compilar. Cada uno contiene una única clase.
- `Jar`: Son archivos comprimidos que contienen varias clases Java y/o recursos (imágenes, texto, etc.). Tiene una estructura propia, en la que las clases están distribuidas en carpetas o paquetes. Ademas puede tener cierta meta-información acerca del contenido del mismo, clase por defecto, versión de Java utilizada, etc.

2.5.3 Groovy

Groovy [Apache Groovy, 2018a] es un lenguaje de programación orientado a objetos definido sobre la plataforma Java. Esto implica que todos los programas y librerías de Java pueden usarse con Groovy. Groovy añade funcionalidad y facilidad de uso, ya que aporta características del *scripting* a Java, permitiendo el uso de código interpretado (léase semi-interpretado: se compila al vuelo).

Algunas de sus características mas interesante para el presente PFC son:

- Metaprogramación: proporciona métodos para manipular las clases y sus métodos desde dentro, y en tiempo de ejecución. Ademas se puede trabajar con prototipos.
- Soporte para arboles sintácticos abstractos: posee una API especifica para ello.
- Clases base Java extendidas: amplia la funcionalidad a las clases base Java para manejar cadenas de texto, listas, etc.
- *Closures*: permite construir fragmentos de código que funcionan como entidades propias ejecutables de modo independiente.
- Sintaxis simplificada: las reglas sintácticas Java se relajan (control de tipos, paréntesis, puntos y comas)y permiten escribir código mas rápidamente.

Por contra, hay que tener presente las siguientes contraindicaciones:

- Código adicional: alguna de las extensiones comentadas antes hacen que se añada mucho código adicional a los objetos que no siempre se utiliza, y que puede ralentizar los tiempos de ejecución drásticamente en algunos casos. Como solución posible se recomienda migrar las clases implicadas a Java.
- Propensión a errores: Al tener una sintaxis muy abierta y dinámica, suelen aparecer problemas en *tiempo de ejecución* que no siempre son fáciles de detectar y/o resolver.

La versión Groovy utiliza en el proyecto ha sido la 1.8, si bien la última disponible a día de hoy es la 3.0.

2.5.4 Otros

2.5.4.1 Maven

Apache Maven [Apache,2018] es una herramienta muy útil para la gestión de proyectos *software* con control de dependencias. Permite definir fácilmente la estructura del *software* a construir, identificando código fuentes, recursos, pudiendo ademas controlar las fases de construcción del mismo, gestionar las pruebas automáticas y realizar publicaciones en repositorios de *software*, entre otras cosas.

Ademas existen múltiples componentes que se pueden añadir a la construcción para, por ejemplo, comprobar la validez del código con respecto a unas reglas de diseño, comprobar la calidad de las pruebas en cuanto a la cantidad de código que cubren, etc.

Existen múltiples repositorios de *software* disponibles en Internet a los cuales se puede acceder mediante Maven gracias un mecanismo estándar de identificación de dependencias, en el que se indican el grupo, el artefacto, y la versión. Con esos tres elementos y un repositorio donde buscar, Maven se encarga de descargar los archivos necesarios, custodiarlos y asignarlos como bibliotecas de dependencias de cualquier proyecto, ya sea para pruebas, compilación y/o distribución del *software*.

Se ha utilizado la versión 3.5.4

2.5.4.2 JUnit

Herramienta de automatización de pruebas *software* para Java [Junit,2018]. Se he empleado la versión 4, aunque la última versión disponible es la 5. Permite escribir clases de prueba muy fácilmente, pruebas que a su vez se integran dentro del ciclo de desarrollo gracias a herramientas como Maven o Eclipse. Proporciona tanto mecanismos de validación como

anotaciones o extensiones para realizar pruebas complejas como integración de sistemas o acceso a bases de datos de prueba.

La unidad básica de ejecución de JUnit es el Test. Basta con escribir una clase y anotar uno de sus métodos con `@Test`, y ya automáticamente JUnit reconoce ese método como un test. Por lo tanto, durante la fase de ejecución de pruebas, se invoca a JUnit para que busque todas las clases que contengan métodos anotados con `@Test`, y los ejecute, secuencialmente (sin orden prefijado, por defecto).

En el código del método anotado, hay que realizar la prueba que se desee. Si el método llega al final sin haber lanzado ningún error, por convenio, el test se ha superado con éxito. Para indicar que el test falló, hay que invocar el método `fail(mensaje_de_error)`. Se recomienda que el nombre del método/test sea descriptivo.

Al final, JUnit mostrará un resumen con todos los resultados de los tests lanzados. Si alguno falla, el ciclo de construcción del software para, y hasta que no se resuelvan los errores no se puede continuar (si se usa Maven, por ejemplo).

Se muestra a continuación un test real de los usados en el presente PFC:

```
/**
 * Simple process test with intermediate interrupting events with no signal event definition
 */
@Test
public void simpleProcessTestWithIntermediateInterruptingEventsWithNoSignalEventDefinition() {

  try {
    InputStream input = this.getClass().getClassLoader()
        .getResourceAsStream("procesoG.xml");
    if (input != null) {
      // Se lee el proceso
      BpmnTree ast = parser.doParse(input);

      List<BaseElement> il = ast.getElementsByType(Interruption.class);
      assertTrue(il.size() == 1);
      Interruption interruptNode = (Interruption) il.get(0);
      assertTrue(interruptNode.getSignalEventDefinitions() != null);

      List<SignalEventDefinition> signals = interruptNode.getSignalEventDefinitions();

      assertTrue(signals != null);
      assertTrue(signals.size() == 0);

    } else {
      fail(filenotfoundmessage);
    }

  } catch (ParseException e) {
    if (e.getCause() instanceof IOException)
      fail(filenotfoundmessage);
    else if (e.getCause() instanceof SAXParseException)
      fail(documentonovalidomsg);
    else if (e.getCause() instanceof SAXException)
      fail("Documento mal formado");
```

```
    else if (e.getCause() instanceof MalformedURLException)
      fail("URL mal formada");
    else
      fail(e.getMessage());
  }}
```

3 Planteamiento del problema

El problema al que intenta dar solución este PFC es: ¿podría usarse alguno de estos lenguajes de modelado de procesos de negocio para poder diseñar y ejecutar programas informáticos? O planteado de otro modo: ¿Podrían diseñarse programas informáticos de modo que su lógica interna, el negocio, esté representada mediante una notación estándar de procesos de negocio? De este modo:

- Se facilitaría la *compresión* del programa por perfiles no técnicos.
- Se facilitaría el *diseño* y *modificación* de dichos programas.
- Se contaría con una descripción visual de la lógica implementada por el programa.

Se ha optado por el lenguaje de notación BPMN por diversos motivos expuestos anteriormente:

- Tiene una notación con representación gráfica.
- Cubre las necesidades mínimas requeridas para realizar programas simples.
- Es extensible.
- Existen editores gráficos de código libre.
- Tiene un amplio soporte en la industria, considerándose un estándar de facto.

3.1 Objetivos

El objetivo final de este Proyecto Fin de Carrera es poder ejecutar programas diseñados mediante el lenguaje de notación de procesos BPMN.

La idea es proporcionar un entorno base, genérico, que pueda ser extendido y adaptado *a posteriori* a las necesidades de cualquier *negocio*.

Para llevar a cabo dicho cometido, se plantean dos objetivos:

3.1.1 Construir un compilador de procesos de negocio

Para poder ejecutar un programa, hay que transformar el código fuente, en este caso en formato BPMN, en código objeto, ejecutable en un entorno concreto.

El compilador es el elemento encargado de esta tarea, para lo cual:

- Valida el contenido del código fuente, sintáctica y semánticamente.
- Crea una representación interna del programa.
- Genera un programa ejecutable a partir de la representación anterior y del entorno de ejecución base (ver siguiente punto).

Derivados de este objetivo, se detectan otros dos objetivos secundarios, pero necesarios para abordar con éxito esta tarea:

3.1.1.1 Selección de un subconjunto BPMN

El carácter extensivo del lenguaje BPMN, que trata de dar cubrir un amplio espectro procesos de negocio, lo convierte en un lenguaje muy rico, pero muy complejo, con algunas de sus elementos (diagramas, componentes, etc.) apenas sí usados en la práctica [Freund and Rücker, 2012]. Implementar un compilador que reconozca toda su sintaxis queda fuera del alcance del presente PFC, ya que, aparte ser mas costoso, no aporta funcionalidad adicional a los programas que se pretenden compilar y ejecutar.

Se plantea, por tanto, el objetivo de seleccionar un subconjunto mínimo de la notación BPMN que cubra las necesidades mínimas y suficientes que permitan realizar programas completamente funcionales de acuerdo a unos requisitos mínimos de complejidad y utilidad.

3.1.1.2 Selección de un editor BPMN

Hay que seleccionar un editor BPMN que permita crear y editar los programas/procesos creados mediante el compilador, para lo cual:

- tiene que cubrir perfectamente la especificación BPMN 2.0.
- no debe añadir elementos ajenos a la especificación.
- ha permitir editar el código fuente XML en caso de necesidad.
- opcionalmente, se valorará positivamente que sea un editor de código abierto, de cara a poder ampliarlo en un futuro.

3.1.2 Crear un entorno de ejecución base para la ejecución de los procesos

Para que un proceso se ejecute, hace falta una serie de elementos base o *clases*, predefinidos, extensibles, y conocidos por el compilador a la hora de generar el código objeto. En tiempo de ejecución, estos paquetes serán los que conformen el entorno de ejecución de los procesos,

proporcionando los servicios y métodos genéricos básicos disponibles para cualquier proceso que quiera ejecutarse.

El entorno de ejecución ha de ser fácilmente modificable y extensible para poder adecuarse a los distintas necesidades especificas de cada *negocio*.

Entre otras cosas, el entorno de ejecución ha de proporcionar:

- Un proceso Base: será el proceso del que extiendan todos los demás.
- Un contexto: el contexto de ejecución de los procesos, donde puedan almacenar la información que necesiten durante su ciclo de vida.
- Eventos: los eventos generados o recibidos por los procesos.
- Repositorio de procesos: almacén de procesos disponibles. Necesario para que un proceso pueda invocar a otros procesos o subprocesos.

4 Diseño e implementación del sistema

Se presentan a continuación todos los detalles relativos al diseño e implementación del sistema realizado para alcanzar los objetivos planteados en el apartado anterior.

4.1 Lenguajes de programación

Se ha optado por implementar la solución con los lenguajes Java y Groovy, ya que son lenguajes con gran soporte en la comunidad y existen muchas librerías con funcionalidad ya disponible. Groovy es una extensión de Java, que incorpora aún mas funcionalidad y permite, entre otras cosas, generar y ejecutar scripts *al vuelo*, acceder y manipular en tiempo de ejecución las características de un objeto (reflectividad y metaclases) y manipular mas fácilmente árboles sintácticos para generar código.

En concreto se usa la versión 1.7 de Java [Oracle,2018] y la 1.8 de Groovy [Apache Groovy,2018a].

Además, para facilitar la compilación y construcción de los distintos componentes, se ha usado Apache Maven, en su versión 3.5.0 [Apache,2018].

4.2 Subconjunto BPMN utilizado

Se ha seleccionado un subconjunto de la notación BPMN que permite realizar los programas con unas mínimas funcionalidades deseables.

En concreto, se usarán únicamente Diagramas de Proceso, uno de los diagramas definidos por la notación BPMN, serializados en formato XML. Cada proceso que se defina de este modo se corresponderá con un único programa ejecutable.

Con estos diagramas de proceso se puede:

- Ejecutar código arbitrario (scripts Groovy en este caso)
- Invocar a otros procesos de forma síncrona o asíncrona.
- Realizar decisiones condicionales, tipo *if - else*
- Realizar decisiones por casos, tipo *switch - case*
- Realizar bucles
- Enviar y esperar eventos

Este conjunto mínimo de elementos permite realizar cualquier lógica de proceso que se requiera, gracias a los scripts Groovy genéricos. Se puede hacer que el proceso abra una ventana de consulta al usuario, realice una llamada a un servicio web, se conecte con un dispositivo Bluetooth, etc. Todo dependerá de la implementación deseada, para lo cual deberán desarrollarse los scripts genéricos necesarios, programados en en Groovy o Java. Se expondrán ejemplos más adelante.

4.2.1.1 Extensiones

El lenguaje BPMN permite añadir elementos adicionales a las definiciones estándar de cada elemento. Son los llamados elementos de extensión. Cada elemento puede tener uno o varios, y son opcionales. Las extensiones no son más que propiedades adicionales que se añaden a las ya definidas por la notación. Tienen un nombre y un valor.

Estas extensiones se añaden anidando dentro del elemento seleccionado un elemento denominado `<extensionElements>`, que puede contener una lista arbitraria de elementos hijos a añadir, con sus atributos particulares.

En el presente PFC se han utilizado para añadir funcionalidad extendida a aquellos elementos, o bien muy genéricos, o bien que tuvieran alguna carencia de cara a cubrir las necesidades detectadas

Por ejemplo, en elemento `<callActivity>`, o Llamada, se reconoce un atributo extra, `async`, que permite indicar si la llamada es asíncrona o no:

```
<callActivity calledElement="subprocess1" id="_3" name="Nodo1">
      <extensionElements>
            <property name="async">true</property>
      </extensionElements>
</callActivity>
```

4.2.1.2 Elementos BPMN empleados

Se detallan a continuación los elementos BPMN que pueden emplearse para diseñar programas, seleccionados de entre todos los listados en el apartado 2.4.2.

De cada uno se muestra una breve descripción junto con los atributos y elementos anidados que pueden tener, tanto obligatorios (por defecto) como opcionales (indicado).

4.2.1.2.1 Atributos y elementos comunes

- Atributos
  - `id`: Identificador único del elemento en el ámbito del proceso. Sirve para hacer referencia al elemento desde otros elementos de diseño. Puede usarse cualquier secuencia de caracteres alfanuméricos, aunque el editor los genera automáticamente con la sintaxis "_<número>": _1, _234, _12...
  - `name`: Nombre del elemento. Se representa gráficamente y se puede modificar con el editor directamente.
- Elementos
  - `extensionElements`: Opcional. Elemento complejo, que contiene otros elementos anidados, no pertenecientes al dominio BPMN, exclusivos para este trabajo. Si bien pueden asignarse a cualquier elemento para extender su funcionalidad, aquí se usan única para las actividades de tipo `Task`, ya que son las actividades genéricas que se han especializado para realizar tareas específicas de este trabajo.

4.2.1.2.2 Actividades

- `Task` (Tarea): Actividad base. El resto de actividades extienden de esta, de modo que los atributos y características aquí comentados también aplican al resto.
  - Elementos (todos opcionales):
    - `standardLoopCharacteristics`: indica que la actividad ha de repetirse tantas veces como indique el contenido del elemento anidado `loopCondition:`

```
<standardLoopCharacteristics loopMaximum="2">
      <loopCondition><![CDATA[n<3]]></loopCondition>
</standardLoopCharacteristics>
```

    - `incoming / outgoing`: elementos que indican que transiciones de salida o de entrada tiene la actividad. Cada uno contiene un identificador de transición, elemento de tipo `SequenceFlow.`

```
      <incoming>_4</incoming>
      <outgoing>_5</outgoing>
      <outgoing>_6</outgoing>
```

- `CallActivity` (Llamada): Se usan para invocar a otros procesos.
  - Atributos:

        - `CalledElement`: identificador del proceso a llamar
    - Elementos extendidos:
        - `Async`: Si es cierto, la llamada se realiza y el proceso llamante continua. Si es falso, la llamada se realiza y el proceso *llamante* se interrumpe hasta que el proceso *llamado* termina. Opcional. Por defecto, es falso.
- `ScriptTask`: Se usan para ejecutar un script Groovy
    - Atributos:
        - `Script`: contiene en un literal de texto el código Groovy a ejecutar. Puede contener varias lineas de código

```
<scriptTask id="_8" name="Task" >
      <incoming>_9</incoming>
      <outgoing>_10</outgoing>
      <script><![CDATA[context.afterCatch = true]]>
      </script>
</scriptTask>
```

4.2.1.2.3 Conectores

Se reconocen dos conectores:

- `SequenceFlow` (Transición): Se emplean conectar otros elementos, determinando un orden secuencial de recorrido de los mismos.
    - Atributos
        - `sourceRef`: identificador del elemento origen.
        - `targetRef`: identificador del elemento destino.
        - `ConditionExpression`: condición lógica que debe cumplirse para que se realice la transición. Opcional. Usado para transiciones de salida de nodos `ExclusiveGateway.`

4.2.1.2.4 Eventos

Se reconocen los siguiente eventos:

- `StartEvent`: evento inicial. Tiene que existir uno y solo uno.
- `EndEvent:` evento final. Puede haber varios.

- `IntermediateCatchEvent`: Evento intermedio (captura)
- `IntermediateThrowEvent`: Evento intermedio (envío)
- `boundaryEvent`: Evento asociado a una actividad, se puede emitir durante su ejecución.

Elementos comunes:

- `errorEventDefinition`:
  - Atributos:
    - `errorRef`: Contiene el nombre completo de la clase que implementa el error capturado o emitido (solo para eventos de tipo Error)
- `signalEventDefinition`:
  - Atributos:
    - `signalRef`: Contiene el nombre de la señal enviada o esperada.

4.2.1.2.5 Compuertas

Se reconocen los siguiente compuertas:

- `EventBasedGateway`: compuerta basada en eventos.
  - Atributos:
    - `eventGatewayType`. Siempre será "`Exclusive`", ya que solo se activará una salida, la correspondiente al primer evento esperado que llegue.
- `ExclusiveGateway`: compuerta tipo O lógico.

4.2.1.3 Particularidades de los procesos BPMN

Los procesos BPMN que se pueden compilar han de tener las siguientes características:

- Tiene que haber un y sólo un proceso por fichero fuente.
- El identificador del proceso ha de ser único.
- El proceso debe comenzar por un nodo de Evento de comienzo (*StartEvent*).
- El proceso debe tener al menos un Evento de fin (*EndEvent*).
- No puede haber conectores sin enlazar.
- Se pueden usar elementos de extensión `<extensionElements>`

- Solo se pueden emplear los elementos descritos en el apartado 4.2.1.2 . Cualquier elemento no reconocido hará que falle el proceso en fase de compilación.

4.3 Diseño

En este apartado se describen los detalles relativos al diseño e implementación del sistema realizado, sistema que puede dividirse en dos: el compilador de procesos y el motor o entorno de ejecución de procesos. El editor de procesos queda fuera del alcance: como se vio en apartados anteriores, se ha utilizado el editor de código abierto Yaoqiang BPMN 2.0 Editor [Yaoqiang, 2017].

Cada uno de las partes se describe mediante uno o varios diagramas de clases UML [OMG,2017][OMG, 2018]. Este tipo de diagramas permiten definir las entidades *software* del sistema por medio de clases, con sus atributos y métodos, y relaciones entre ellas (asociación, implementación, etc.).

4.3.1 Compilador

El compilador se he ha realizado mediante los lenguajes de programación Java y Groovy.

Es el encargado de recibir un fichero fuente en formato BPMN (puede tener extensión BPMN o XML), y generar un fichero JAR que contiene uno o vario ficheros `.class`, con el código objeto Java obtenido a partir del fichero recibido.

Para realizar su cometido, el compilador depende de las clases base e interfaces en los que se apoya todo proceso para su posterior ejecución. Se detallan más adelante.

El compilador está compuesto por un analizador sintáctico, un validador semántico y un generador de código. Internamente construye y maneja dos estructuras: un árbol sintáctico y una tabla de símbolos.

El proceso de compilado es como sigue:

- Se recibe un fichero XML en formato BPMN.
- Se procesa y se crean el árbol sintáctico y se rellena la tabla de símbolos.
- Se valida semánticamente el árbol, apoyándose en la tabla de símbolos.
- Se genera el código objeto a partir del árbol, válido sintáctica y semánticamente.
- En caso de cualquier error, el proceso se para y se comunica al usuario

La Figura 54 resume estos pasos de modo gráfico.

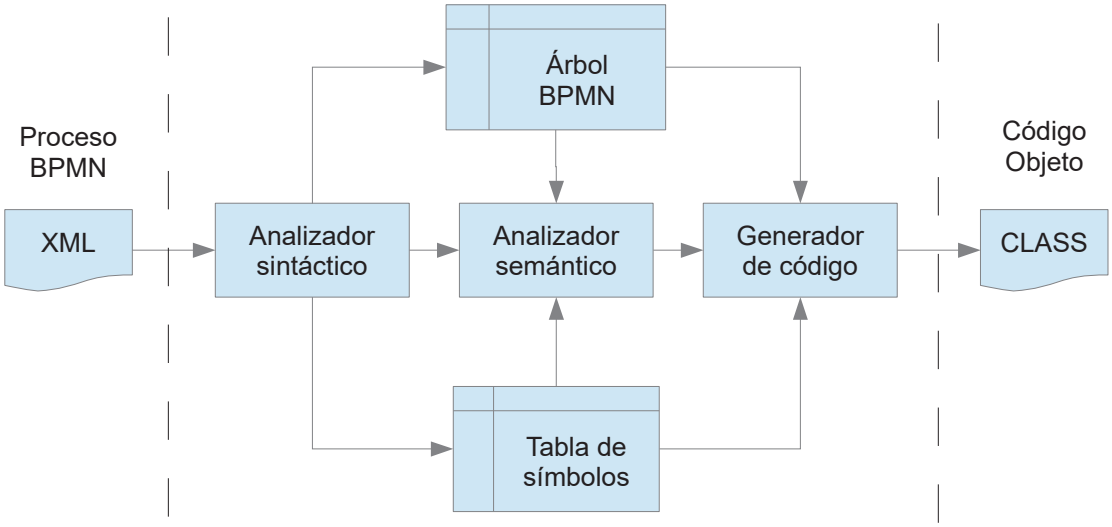


Figura 54: Pasos del proceso de compilación

4.3.1.1 Parseador – Analizador sintáctico

El parseador lee un documento BPMN en formato XML y construye un árbol sintáctico con él.

Usa para ello la herramienta de *Apache Commons Digester* [Apache,2013], que permite configurar qué hacer con cada elemento de un documento XML recibido. Esta herramienta valida el documento contra el esquema BPMN 2.0 (`BPMN20.xsd`)

En este caso, se buscarán solo los elementos reconocidos por el compilador, pertenecientes al subconjunto BPMN seleccionado, y se obtendrá un objeto **`BpmnTree`**, con toda la información recopilada durante el parseo.

El parseador lanzará un error si el XML está mal formado, si hay mas de un proceso definido en el XML, si no se encuentra alguno de los atributos obligatorios, etc. Todos errores relacionados con esta primera lectura del documento XML.

4.3.1.2 Árbol sintáctico BPMN

Como resultado del parseo del XML tomado cómo código fuente, se obtiene una estructura similar a un Árbol de Sintaxis Abstracta, pero ampliado, al combinarse con las particularidades de los procesos BPMN.

Se ha optado por esta solución intermedia para facilitar los análisis sintáctico y semántico en detrimento de la generación de código. En un compilador clásico el árbol sintáctico generado

es una representación abstracta del código y basta con recorrerlo del modo adecuado para obtener el código objeto deseado.

En este caso, el árbol sintáctico BPMN es una estructura intermedia que permite crear a posteriori un Árbol de Sintaxis Abstracta Groovy, del que ya directamente se pude obtener el código objeto Java ejecutable en una Máquina Virtual Java (JVM).

Una vez obtenido este árbol BPMN, se recorre con dos *visitadores* distintos: el validador semántico, y el generador de código.

4.3.1.3 Tabla de símbolos

Otra estructura necesaria obtenida tras los análisis sintáctico y semántico es la Tabla de símbolos. Internamente es un mapa –conjunto de pares (clave, valor)– y es dónde se guarda referencia de todos los elementos sintácticos encontrados, junto con su identificación.

4.3.1.4 Validador semántico

Es un visitador del Árbol BPMN que se encarga de validar semánticamente el contenido del mismo.

La validación se realiza siguiendo el patrón de diseño *Visitor* [Wikipedia,2018b], de modo que el árbol BPMN determina el recorrido que se hace por los distintos elementos del mismo, teniendo cada uno de ellos un método `accept(Visitor)`, y es el validador (Visitor) el que realiza el análisis de cada elemento, con el método `visit(Elemento).`

En la validación particular de cada elemento se realizan las comprobaciones pertinentes para comprobar que el código puede generarse (en la siguiente fase), y se usa la Tabla de símbolos para guardar o acceder a la información necesaria en cada momento.

4.3.1.5 Generador de código

La última fase del compilador es la generación de código, en la que, una vez comprobado que el código es valido sintáctica y semánticamente, se ha construido la tabla de símbolos y se ha recopilado toda la información necesaria, se genera el código objeto ejecutable deseado, en este caso Java.

Para ello, se ha implementado otro *Visitor* que va construyendo una clase Java dinámicamente usando el paquete AST de Groovy [Codehaus, 2014].

La nueva clase Java construida usa las clases e interfaces definidos en el entorno de ejecución, que se verá a continuación.

Por último, la clase construida se compila con un compilador Java estándar, se obtiene uno varios ficheros `.class`, y se empaquetan en un fichero JAR.

4.3.1.6 Diagramas de clases

La Figura 55 muestra el diagrama de clases y entidades implicadas en el proceso de compilación. Solo se muestran las clases principales, sus relaciones y algunos de sus métodos y atributos principales. En la siguiente sección 4.3.1.7 se listan y detallan todas las clases, junto con los los atributos y métodos que proporciona cada una.

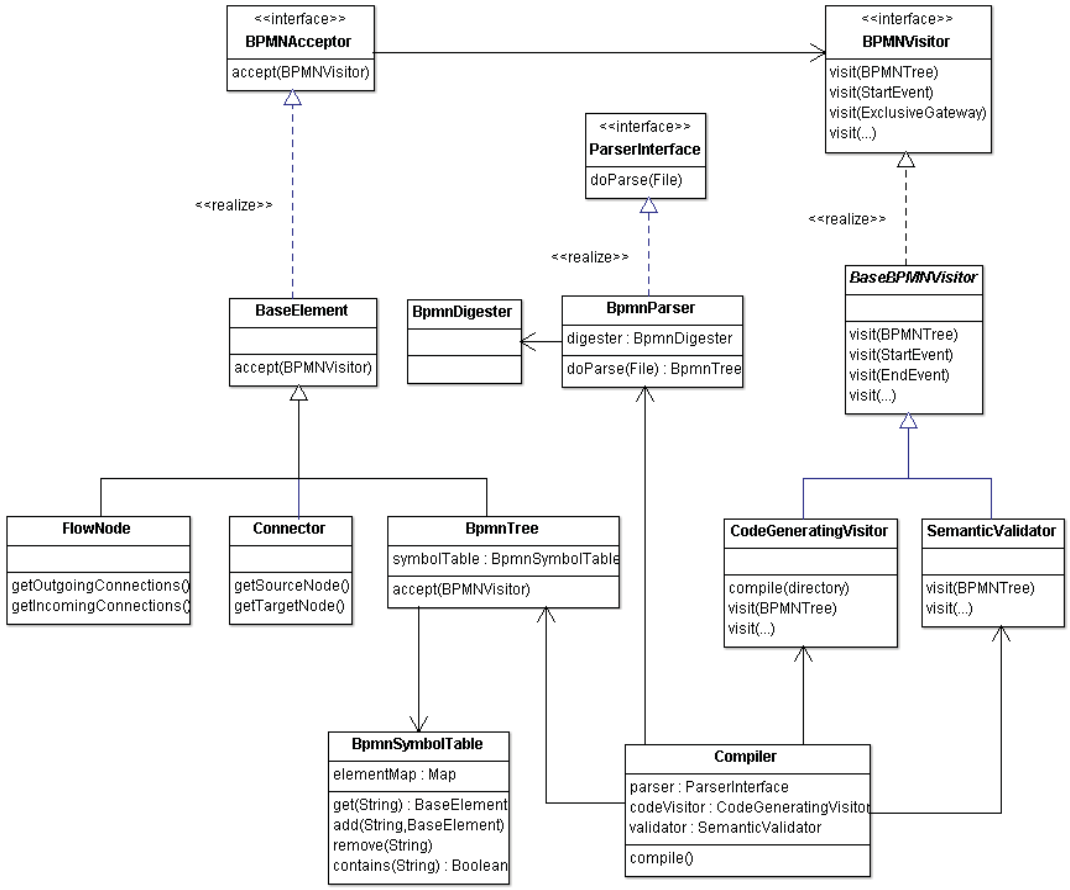


Figura 55: Diagrama de clases del compilador

En el siguiente diagrama, Figura 56, se muestra toda la jerarquía completa de elementos BPMN que derivan de la clase *BaseElement*. Son todos los elementos BPMN reconocidos por el analizador sintáctico.

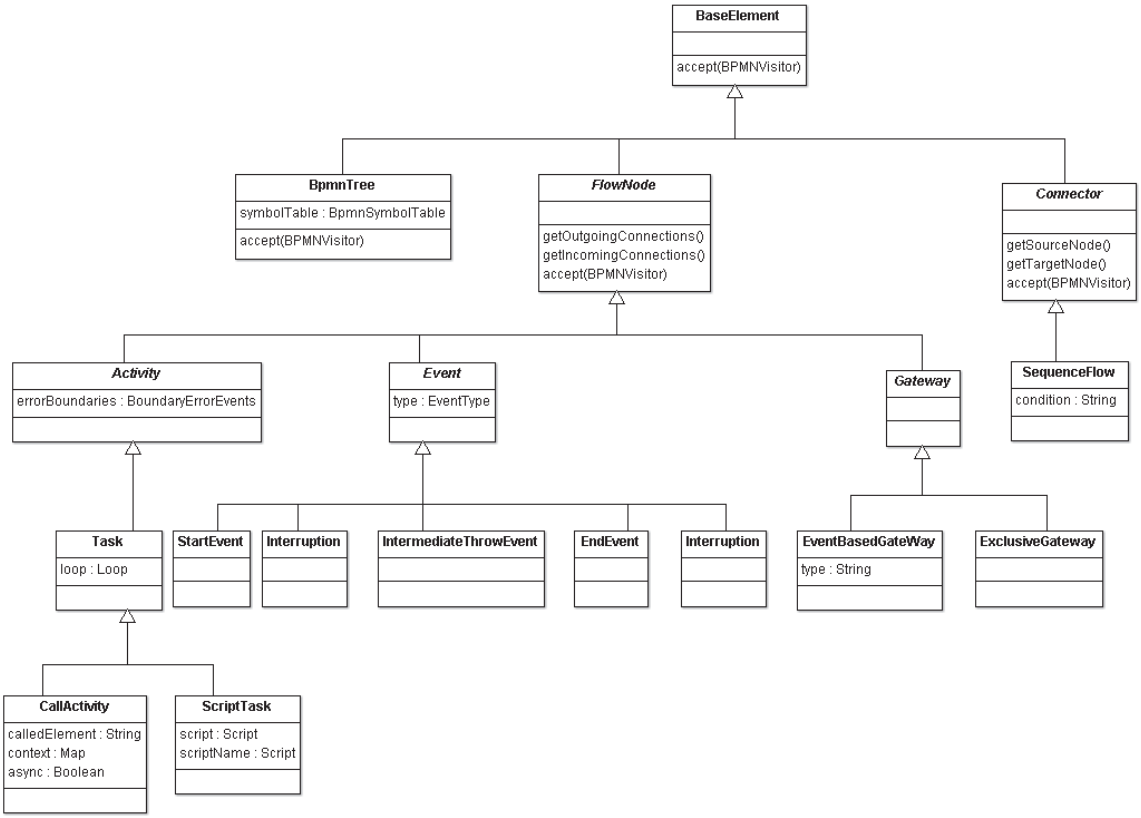


Figura 56: Clases del modelo BPMN

4.3.1.7 Descripción de las clases

4.3.1.7.1 Compilador

Se describen a continuación todas las clases implicadas en el proceso de compilación. Es importante destacar el uso de Groovy y su paquete AST, ya que facilitan mucho el proceso. Groovy es un lenguaje que aporta las ventajas dinámicas de un lenguaje interpretado (como cualquier lenguaje de *scripting*), además de incluir reflexibilidad y meta-clases. El paquete AST proporciona objetos y utilidades para ir construyendo objetos al vuelo, así como para compilarlos. Su modelo es muy completo, y contiene clases para cualquier concepto que pueda encontrarse en un programa informático: Clase, Constructor, Método, Parámetro, Sentencia, Bloque, Asignación, etc.

El proceso general es simple: el compilador recibe un proceso BPMN en formato XML, construye un árbol sintáctico con él, lo valida semánticamente, y genera código a partir de su

contenido, construyendo el meta-código con los elementos de Groovy AST. El resultado final es un objeto de tipo Clase (`classNode`), que AST permite compilar y generar un fichero `.class` a partir de él.

4.3.1.7.1.1 Compiler

Es la clase encargada de centralizar todo el proceso de generación de código objeto ejecutable a partir del código fuente XML recibido como entrada. Además, gestiona y comunica los posibles errores de compilación que pueden darse durante el proceso.

Es muy simple, pues delega toda las funcionalidades en las clases asociadas, que se encargan en última instancia de realizar las tareas de análisis sintáctico y semántico, y generación de código.

4.3.1.7.1.2 BpmnParser

Es la clase encargada de crear el árbol sintáctico de compilación. Recibe un archivo y se apoya en la clase `BpmnDigester` para realizar el análisis sintáctico.

4.3.1.7.1.3 BpmnDigester

Es la clase encargada de ir construyendo el árbol sintáctico a medida que consume el documento XML en el que está definido el proceso en formato BPMN.

Se trata de una extensión de la clase `org.apache.commons.digester3` que emplea dos conjuntos de reglas distintos que controlan la generación del árbol sintáctico. Las reglas se van ejecutando a medida que se encuentra el patrón que las dispara dentro del documento XML. Hay dos conjuntos de reglas por cuestión de legibilidad. El primero, `BpmnModelRuleSet`, contiene todas las reglas que reconocen los elementos estándar de la notación BPMN 2.0 empleados en el presente trabajo. Así, hay al menos una regla por cada elemento del modelo implementado en el paquete `compiler.ast.model`. El segundo conjunto, `ExtendedRuleSet`, contiene aquellas reglas específicas que extienden la notación BPMN y que son necesarias para dotar de la funcionalidad adicional a los procesos. En concreto, es una única regla que permite recoger y tratar la información contenida dentro de los elementos `extensionElements`, ideados con estre propósito en el estándar BPMN.

Al finalizar el procesamiento del documento XML, se habrá generado un árbol sintáctico, objeto de la clase BpmnTree, que contendrá toda la información estática contenida en dicho documento, junto con el diccionario de datos, recopilado a la par que se ha generado el árbol.

Está información incluye los atributos y datos asociados a cada nodo del proceso, las relaciones entre los nodos, las condiciones, los bucles, eventos, etc.

4.3.1.7.1.4 BpmnTree

Es la clase que implementa el árbol sintáctico abstracto que contiene toda la información necesaria para compilar un proceso BPMN.

El árbol representa fielmente las relaciones entre nodos existentes en el proceso BPMN, así como el contenido descriptivo de cada nodo. Cada proceso de negocio empieza necesariamente con un evento inicial, de modo que el nodo raíz del árbol será precisamente este evento inicial, de tipo `StartEvent`. A partir de él se pueden encontrar el resto de nodos del proceso, bien como nuevos nodos intermedios, bien como hojas, nodos que no tienen hijos. Recorriéndose el árbol en profundidad se puede reconstruir el diagrama BPMN original del que partió el mismo.

Como se comentó en el punto anterior, junto con el árbol semántico, también se construye una tabla de símbolos, por comodidad contenida dentro del árbol, de tipo `BpmnSymbolTable`. Internamente es una tabla *hash* indexada por nombre, y que guarda elementos de tipo `BaseElement`. La tabla, junto con los habituales métodos para insertar, borrar o recuperar elementos, también proporciona métodos para realizar búsquedas de elementos por tipo.

4.3.1.7.1.5 SemanticValidator

Es la clase encargada de realizar la validación semántica del árbol generado, apoyándose para ello en la tabla de símbolos. Realiza diversas comprobaciones para asegurar que el código a generar es valido. Entre ellas:

- Qué solo existe un proceso definido.
- Que el proceso sólo tiene un evento de comienzo.
- Que existe al menos un nodo de fin.
- Que todos los nodos son alcanzables. Esto es, no hay ningún nodo huérfano.
- Que no hay bucles (estáticos)
- Que todas las referencias entre elementos es válida. Los elementos referenciados existen y son del tipo esperado.
- Que los nodos tienen el número de conectores de entrada y/o salida esperados. Varían en función del tipo de nodo, su ubicación y/o nodos anteriores o posteriores.

- Que los elemento tienen los atributos esperados, y que su contenido es válido. (nombres, referencias, condiciones o tipos, por ejemplo)

4.3.1.7.1.6 CodeGeneratingVisitor

Esta clase se encarga de generar el código a partir del árbol sintáctico, una vez ha superado el análisis semántico del punto anterior. Es la que soporta mas complejidad, ya que es la encargada de interpretar toda la información recopilada hasta el momento.

Se apoya en el lenguaje Groovy y concretamente en el paquete `org.codehaus.groovy.ast`, que proporciona las clases que definen el propio lenguaje Groovy, así como utilidades que permiten ir construyendo clases Groovy dinámicamente.

La clase implementa el patrón *Visitor* y por tanto, se dedica a visitar en profundidad los nodos del árbol sintáctico, e ir generando las estructuras necesarias para completar un objeto Groovy a partir del cual generar ya un código ejecutable.

Se apoya internamente en dos objetos mas, `ClassNode` y una pila de objetos `Statement`, ambos objetos procedentes del paquete AST de Groovy. El primero es la meta-clase que se va construyendo durante el recorrido del árbol (por cada proceso BPMN se genera una clase). El segundo es una estructura auxiliar que permite llevar control de los anidamientos de llamadas que se dan dentro de los nodos del proceso.

Al final del proceso, la clase `ClassBuilder` genera un fichero `.class` en el mismo directorio donde se leyó el XML del proceso.

Todas las clases generadas extienden de la clase base `ProcessBase`.

4.3.1.7.2 Elementos BPMN

Se describen en este apartado todas las clases que definen el modelo que representa los distintos elementos BPMN reconocidos por el compilador. Se corresponden con los distintos objetos que se van creando y completando a medida que se crea el árbol BPMN visto en el apartado anterior.

4.3.1.7.2.1 BaseElement

Es la clase raíz, de la que heredan todas las demás. Define por tanto todas las propiedades y métodos comunes que usan todos los elementos BPMN implementados. En concreto define un nombre y un identificador único, un enlace a su elemento padre (si corresponde), y métodos para acceder al árbol semántico, la tabla de símbolos, añadir elementos hijo. Declará

un método abstracto, `accept()`, que deben implementar las clases hijas, y que es consecuencia de ser la clase base de los elementos del árbol sintáctico: es el método que invocan las clases que visitan el árbol, el validador semántico y constructor de código.

4.3.1.7.2.2 Connector

Es la clase que representa el enlace o relación entre dos (y solo dos) elementos del proceso. El enlace es direccional, de modo que hay un elemento origen (del que parte la relación) y un elemento destino (al que llega la relación). Los elementos conectado han de ser necesariamente de tipo `FlowNode`.

4.3.1.7.2.3 SequenceFlow

Clase que extiende la clase `Connector` para añadir un atributo `condition` que permite definir una condición a evaluar para realizar la transición o no (se empleará en las clases `Gateway`)

4.3.1.7.2.4 FlowNode

Clase abstracta, base de la que extienden todas las definidas a continuación: son las que representan elementos concretos de un flujo dentro del proceso, unidos mediante los conectores antes descritos.

Define dos listas, una de conectores de entrada y otra de conectores de salida, así como los métodos para manipularlos, conectar el nodo con otro, etc. También define un atributo particular de esta implementación, llamado `checkpoint`, y que guarda la referencia a un nodo al que volver en caso de una excepción de vuelta atrás (`rollback`). Se explicará mas adelante cuando se describan los procesos, las llamadas a subprocesos y la gestión de excepciones.

4.3.1.7.2.5 Activity

Clase abstracta que define las actividades BPPM, los nodos encargados de realizar las tareas. Define un atributo que contiene una lista de `BoundaryErrorEvents,` que son los eventos que pueden ocurrir dentro de una actividad y que se comunican fuera, para que el proceso contenedor los escuche y pueda actuar convenientemente.

4.3.1.7.2.6 Task

Son las actividades de mas bajo nivel. Tiene un atributo opcional `loop`, que permite definir las condiciones de repetición de la tarea en el caso de ser necesario.

Aunque la clase no es abstracta, no se crean objetos de este tipo directamente. Serán todos de los dos tipos definidos a continuación.

4.3.1.7.2.7 CallActivity

Clase que implementa una actividad de llamada a un subproceso. La llamada puede ser síncrona o asíncrona. Si es síncrona, el proceso padre (el que contiene este nodo), espera a que el subproceso llamado acabe. Si es asíncrona, se limita a llamar al subproceso y continuar la ejecución del siguiente nodo.

4.3.1.7.2.8 ScriptTask

Clase que implementa una actividad que ejecuta un script Groovy contenido en la misma. Está clase es la que puede usarse para ejecutar código arbitrario dentro de los procesos.

4.3.1.7.2.9 Event

Clase abstracta que permite definir un evento genérico. Contiene una lista de objetos `SignalEventDefinition` que representan los distintos tipos de señales esperadas o enviadas por el evento, en función de su tipo. De esta clase derivan todos los eventos definidos a continuación.

4.3.1.7.2.10 StartEvent

Clase que representa el evento de inicio de un proceso BPMN.

4.3.1.7.2.11 InterruptionEvent

Clase que representa el evento interrupción de un proceso BPMN. Usado para detener el flujo de un proceso cuando se realizan llamadas síncronas a subprocesos.

4.3.1.7.2.12 IntermediateThrowEvent

Clase que representa un evento lanzado asíncrona mente dentro de un proceso. El evento se lanza y se continua la ejecución del proceso.

4.3.1.7.2.13 EndEvent

Clase que representa el evento de fin de un proceso BPMN. Cuando se alcanza uno de estos eventos el proceso acaba. Si hubiera un proceso padre esperando por él, deberá escuchar este evento.

4.3.1.7.2.14 Gateway

Clase abstracta que representa a una compuerta genérica BPMN.

4.3.1.7.2.15 EventBasedGateway

Clase que representa una compuerta basada en eventos. Los eventos por los que se espera están definidos por los nodos a los que se llega por las transiciones de salida.

4.3.1.7.2.16 ExclusiveGateway

Clase que representa una compuerta exclusiva. Todas sus transiciones de salida han de tener una condición. La primera condición que se cumpla, hará que se tome ese flujo de ejecución en el proceso, ignorándose el resto de opciones.

4.3.2 Entorno de ejecución

Para poder ejecutar los procesos compilados en la fase anterior, se ha creado un entorno de ejecución base, compuesto de varios interfaces, clases base y servicios, que proporcionan la funcionalidad básica para ejecutar los procesos de negocio definidos mediante el lenguaje diseñado para este trabajo.

4.3.2.1 Interfaces y clases base

Los procesos ejecutables obtenidos mediante el compilador son clases Java/Groovy que extienden de un mismo proceso base, y cumple una determinada interfaz.

Cada proceso BPMN se genera una una única clase Java/Groovy, y su punto de entrada será siempre el método `doExecuteProcess()`.

4.3.2.2 Contexto

Se ha implementado un mecanismo de contexto de modo que los procesos puedan guardar y acceder en tiempo de ejecución a la información que necesiten para llevar a cabo su cometido. El contexto se hereda de procesos padres a hijos, es jerárquico, de modo que primero se accede al contexto propio, y luego se busca en la jerarquía superior. Al acabar un proceso, el contexto se pasa al proceso padre (si lo hay) para permitirle acceder a la información del hijo *a posteriori*.

La clase `ProcessContext` es la que contiene la implementación de este contexto. Es un mapa con pares (clave, valor), que además permite guardar información de eventos, y jerarquizar los contextos.

Desde los procesos puede accederse a los elementos del contexto (a la pila de contextos, para se exactos). Se pueden evaluar expresiones directamene desde elementos BMPN (XML) usando la notación siguiente:

```
${<expresión>}
```

Donde la expresión puede ser:

- una variable, en cuyo caso se accederá al valor de la variable en ese momento
  - `${variable}`
- una lista, usando la notación de corchetes para acceder al elemento concreto:
  - `${mi_lista[3]}`
- un mapa, accediendo con el operador punto o con corchetes:
  - `${mapa.clave}`
  - `${mapa[“clave”]}`
- combinaciones de los elementos anteriores:
  - `${lista[1].valor}`
  - `${mapa.lista[indice]}`

También dentro de los scripts Groovy se puede acceder al contexto usando la variable predefinida `context`. Aplican los mismos casos anteriores. Ejemplos:

- `context.variable`
- `context.lista[indice++]`
- `context.mapa.clave`
- `context.mapa[“clave”].lista[0]`

4.3.2.3 Eventos

Se ha implementado un sistema de eventos necesario para las comunicaciones entre procesos. Los procesos pueden enviar eventos y esperar por eventos. El envío de eventos es siempre asíncrono, de modo que el proceso lo lanza y continúa su ejecución. La espera de eventos es asíncrona, y puede llegar a ser múltiple: un proceso puede esperar por N eventos y el primero que llegue es el que se toma como válido y se continúa la ejecución del proceso en función de este evento (su tipo o su contenido)

Cada evento tienen un contexto propio asociado, de modo que cuando un evento llega a un proceso, el contexto del evento se pasa al contexto del proceso de modo que sea accesible desde ese momento en adelante.

4.3.2.4 Repositorio de procesos

Para organizar la ejecución de los procesos, dado que puede haber varios procesos en ejecución en un determinado momentos, algunos terminados y otros interrumpidos, esperando por algún evento, se ha implementado un repositorio de procesos simple que se encarga de todo esto.

Internamente contiene:

- Un gestor de eventos, encargado de recibir eventos, identificar que eventos esperan los procesos, y enviarlos convenientemente.
- Un ejecutor de procesos, que gestiona los distintos hilos de ejecución, que en ultima instancia son los que soportan la ejecución de los procesos.
- Un área de procesos, donde los procesos que están en ejecución se registran.

4.3.2.5 Ejecución de procesos

El motor o entorno de ejecución soporta dos modos de funcionamiento distinto, para dar soporte a dos casos de uso distintos: ejecutar un único proceso, independiente de otros, o ejecutar una serie de procesos interdependientes. Se comentan a continuación ambos modos, remarcándose las semejanzas y diferencias esenciales.

En todo caso, el ciclo de vida de cualquier proceso se puede representar mediante diagrama de estados mostrado en la Figura 57, en el que se pueden ver los distintos estados por los que puede pasar un proceso, y las transiciones posibles entre los mismos.

4.3.2.5.1.1 Un único proceso

Es el modo mas sencillo de ejecución. El motor recibe una única clase que contiene el código compilado del proceso, la registra, busca su evento inicial (todo proceso tiene uno), y lo lanza. El proceso en ese momento, escucha el evento y pasa de interrumpido a estar en ejecución. El proceso seguirá en ejecución (transitando entre nodos, ejecutando tareas) hasta que llegue a alguno de los eventos de fin (todo proceso tiene que tener al menos uno). En ese momento, el motor detecta este evento de fin y finaliza.

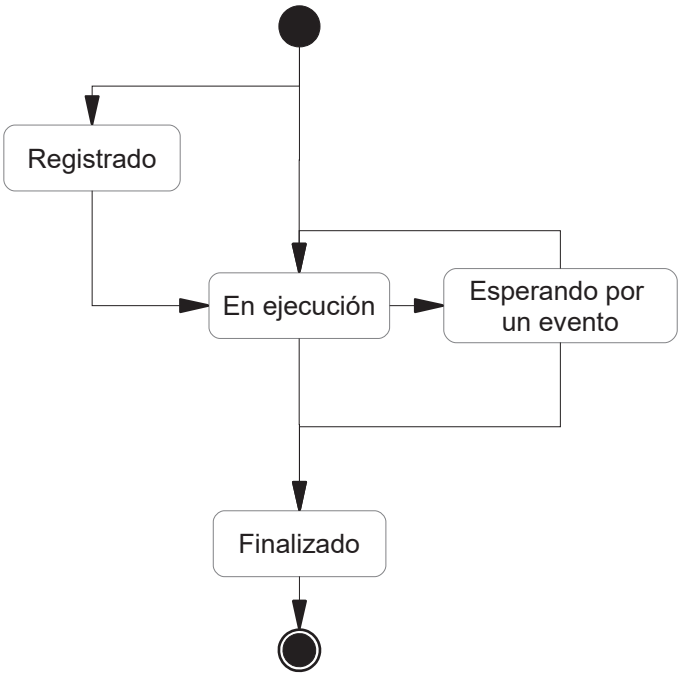


Figura 57: Diagrama de estados de un proceso

Es responsabilidad del programador del proceso de diseñarlo de manera que no aparezcan bucles ni se quede a la espera de eventos que no vayan a lanzarse. Si ocurre algún error no controlado durante la ejecución del mismo, el motor lo detectará y se parará notificando de ello.

4.3.2.5.1.2 Varios procesos

Es el modo que da más versatilidad y potencia al entorno de ejecución desarrollado. Exige un diseño de procesos mas cuidadoso, ya que implica sincronizar varios procesos. Se parte de un directorio o varios (denominado `classpath`) en los cuales se encuentran varios ficheros `.class`, cada uno con un proceso distinto. El motor registra todos los procesos que encuentra (controlando que no haya repetidos, etc.) y lanza un evento concreto, que puede ser configurable, aunque por defecto se lanza uno preestablecido ENGINE_START. Todos los procesos que tengan ese evento como inicial, se arrancarán **en paralelo**. Hasta que todos los procesos no acaben (no haya ninguno en ejecución) el motor seguirá levantado, gestionando los procesos y los eventos que vayan esperando y lanzando.

4.3.2.6 Diagrama de clases

La Figura 58 muestra el diagrama de clases y entidades implicadas en el proceso de compilación. Solo se muestran las clases principales, sus relaciones y algunos de sus métodos y atributos principales. En la siguiente sección 4.3.1.7 se listan y detallan todas las clases, junto con los los atributos y métodos que proporciona cada una.

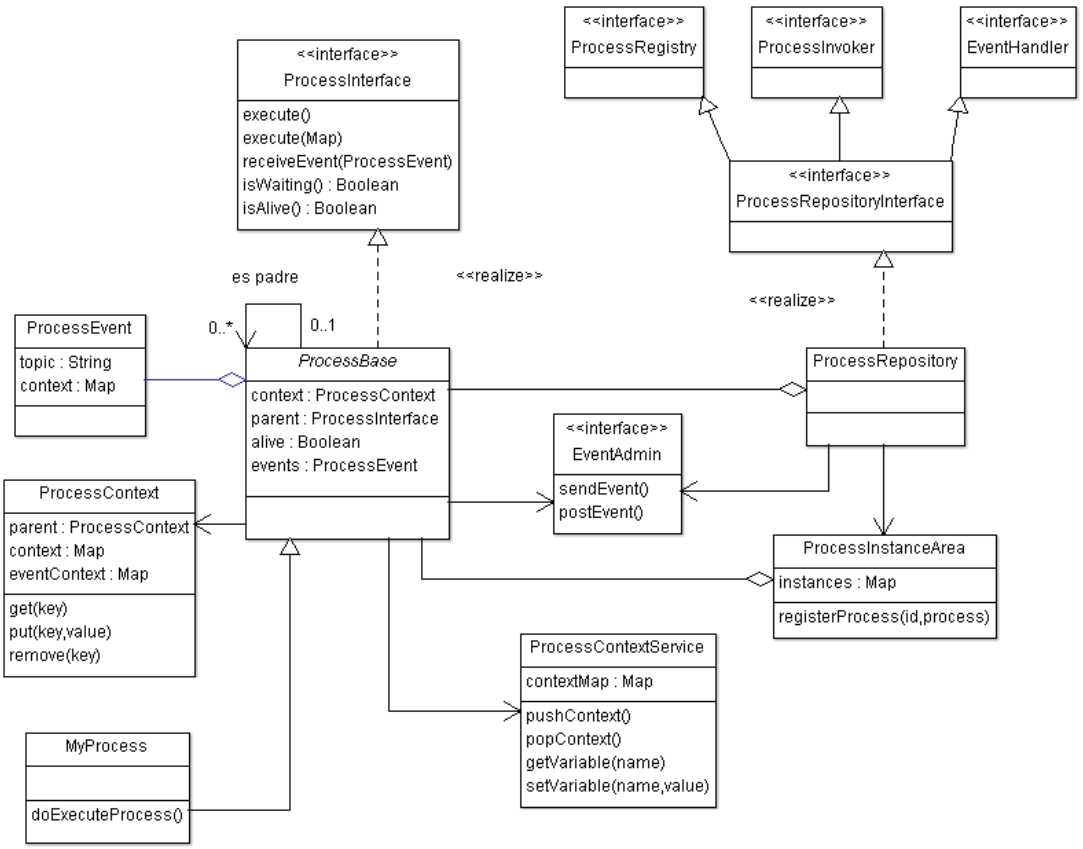


Figura 58: Clases del motor de procesos

4.3.2.7 Descripción de las clases

Se describen en esta sección los detalles relevantes de las clases implicadas en el entorno de ejecución, esto es, las encargadas de llevar a cabo todas las tareas necesarias para gestionar la ejecución del ciclo de vida de los procesos, comunicación de eventos, gestión de contextos, coordinación de hilos de ejecución, etc.

Se pueden distinguir dos conjuntos. Uno es las clases implicadas en la ejecución de un proceso (`ProcessBase`, `ProcessEvent` y `ProcessContext`), el otro sería el formado por las

clases encargadas de gestionar la comunicación y la administración de procesos (`ProcessInstanceArea`, `ProcessRepository`, `EventAdmin` y `ProcessContextService`)

4.3.2.7.1 ***ProcessInterface***

Todos los procesos compilados implementan este interfaz, el cual define ciertos métodos públicos conocidos, que permiten:

- Ejecutar el proceso (con o sin parámetros)
- Ejecutar el proceso como un subproceso hijo de otro.
- Obtener el contexto del proceso.
- Conocer el estado del proceso. Puede estar:
  - En ejecución
  - Esperando por un evento
  - Finalizado
- Enviar un evento al proceso.

4.3.2.7.2 ***ProcessBase***

Clase abstracta de la que extienden todos los procesos compilados. Contiene implementaciones básicas de los métodos anteriormente expuestos. Define además el método abstracto `doExecuteProcess()` que es el que se implementa al compilar los procesos BPMN, y se invoca al ejecutarlos.

4.3.2.7.3 ***ProcessEvent***

Es la clase empleada en tiempo de ejecución para comunicar eventos entre procesos. Para distinguirlos se usa el atributo `topic`, que representa el asunto del evento. Todo evento tiene asociado un contexto, esto es, un mapa de pares clave, valor, que se comunican junto con el evento. Si un proceso admite un evento, es decir, está esperando por él, el contexto del evento se copia en el contexto del proceso, para poder acceder a él.

4.3.2.7.4 ***ProcessContext***

Esta clase implementa el contexto de ejecución de un proceso. Tiene dos contextos diferenciados, el de eventos y el de proceso propiamente dicho. Se hace así para que el

contexto de los eventos no sobrescriba el contexto del proceso, y pueda accederse a ambos por separado. El proceso en su ejecución puede gestionar unicamente el contexto de proceso.

4.3.2.7.5 ***ProcessContextService***

Es el servicio de contextos global. Es necesario para gestionar llamadas anidadas a subprocesos. Funciona como una pila de contextos de proceso, en la que siempre se escribe por defecto en el contexto de la cima de la pila. Proporciona métodos par buscar tanto en el contexto de la cima, como en cualquiera de los otros contextos apilados. También para encontrar contextos por nombre, así como para apilarlos y desapilarlos.

4.3.2.7.6 ***ProcessRepository***

Es la clase que implementa el repositorio estático de procesos. Permite mantener una lista de procesos conocidos y expone para ello una serie de métodos para registrarlos, eliminarlos del registro, buscarlos y consultarlos por nombre o por eventos de comienzo.

Al iniciarse el entorno de ejecución se buscan en el directorio de clases (`classpath`) todos los procesos que existan, y se registran en este repositorio, para luego poder acceder a ellos en el momento que se necesite.

4.3.2.7.7 ***ProcessInstanceArea***

Es la clase que implementa el are de procesos que están en interrupción a la espera de algún evento. Provee métodos para registrar procesos que pasan de ejecución a interrumpidos, para ver que procesos están esperando por un determinado evento, y para despertar procesos y pasarlos a ejecución de nuevo.

4.3.2.7.8 ***EventAdmin***

Es la clase encargada de gestionar la comunicación de eventos entre procesos. Tiene dos operaciones principales: `postEvent()` y `sendEvent()`, para enviar eventos asícronamente y síncronamente, respectivamente.

4.4 Editor

Para editar los procesos, se ha utilizado el *software* de código abierto Yaoqiang BPMN Editor [Yaoqiang,2017], que cubre la especificación BPMN 2.0.

Es el editor más completo de los revisados. No obstante, no permite modificar las extensiones especificas utilizadas para los procesos aquí definidos, extensiones que si están dentro del estándar BPMN 2.0. Como consecuencia de esto, los procesos pueden abrirse sin problemas con este editor, pero para modificarlos convenientemente hay que hacerlo manualmente. Se recomienda usar el editor en modo solo-lectura, para evitar posible perdida de código que no reconozca el editor.

Dado que el editor es de código abierto, puede plantearse un trabajo futuro que implique modificar el editor para que que cubra las extensiones específicas necesarias para los procesos de este trabajo.

5 Evaluación del sistema

5.1 Introducción

En este capítulo se describen las pruebas realizadas para comprobar el buen funcionamiento del sistema implementado en este proyecto. El objetivo es comprobar que tanto el compilador como el entorno de ejecución desarrollados realizan su cometido correctamente.

En concreto, el compilador:

- Ha de generar código objeto correctamente a partir de archivos XML.
- Deber detectar y mostrar los errores sintácticos que encuentre.
- Debe detectar y mostrar los errores semánticos que encuentre.

Por su parte el entorno de ejecución::

- Debe ejecutar los procesos generados con el compilador.
- Ha de gestionar correctamente los eventos que se emiten y se capturan dentro de los procesos.
- Debe ser capaz de mantener un registro de los procesos y subprocesos en ejecución.

En concreto, se han diseñado y realizado diversas baterías de pruebas automatizadas, tanto unitarias como de integración, es decir, pruebas a componentes individuales y pruebas al sistema completo o subsistemas detectados.

Además, se ha comprobado el nivel de cobertura de dichas pruebas, entendiendo por cobertura el porcentaje de lineas de código que han sido comprobadas.

En concreto se han marcados unos objetivos a verificar:

- un porcentaje superior al 80% de las líneas del código desarrollado tiene al menos una prueba que lo ejecuta.
- un porcentaje superior al 50% de todas las ramas lógicas de ejecución quedan cubiertas.

Tanto para realizar las pruebas unitarias como las de integración, se ha usado la herramienta JUnit, en su versión 4.8.2 [Junit,2018].

Para realizar las pruebas de cobertura de código, se ha empleado la herramienta Cobertura [Codehaus,2015], un complemento de la herramienta Maven.

Una de las ventajas de usar estas herramientas es que tanto JUnit como Cobertura se integran dentro de Maven, por lo el proceso de codificación, compilación, despliegue, pruebas y verificación queda simplificado, al ser gestionado por Maven, una vez configurado convenientemente.

5.2 Evaluación

5.2.1 Pruebas unitarias

Por cada clase implementada, se ha creado un test unitario par probar el correcto funcionamiento de sus métodos. A veces se ha creado una clase solo para probar el funcionamiento conjunto de varias clases. Con esto se persigue comprobar que cada pieza individual del sistema funciona por si misma como se espera. En algunos casos se ha usado el paradigma *Test_Driven Delevopement* (TDD), en el que las pruebas se desarrollan *antes* el propio código. De este modo el funcionamiento (el *qué*), queda establecido antes que la implementación (el *cómo*).

Entran dentro de esta categoría pruebas realizadas a cada uno de los componentes del compilador, o cada uno de los elementos BPMN implementados.

5.2.2 Pruebas de integración

Las pruebas de integración se han realizado sobre bloques mas grandes de código, de modo que las pruebas se realizan a partir de objetos mas altos en la jerarquía del sistemas y engloban varias clases. De este modo se comprueba el correcto engranaje de las mismas. Son mas complejas, pero permiten realizar pruebas mas reales al sistema, pues son pruebas cercanas al concepto de *caso de uso*, mas global.

Entran dentro de esta categoría pruebas realizadas sobre el compilador entero o sobre el entorno de ejecución.

5.2.3 Cobertura de código

Una vez realizadas tanto las pruebas unitarias como las de integración, se han evaluado dichas pruebas con la herramienta de verificación de cobertura de código. La herramienta utilizada, Cobertura, sirve para evaluar los porcentajes tanto de líneas cubiertos por las pruebas, como de *flujos lógicos de ejecución* cubiertos, entendiéndose como esto último cada uno de los flujos distintos que puede tomar la ejecución de un fragmento de código en un momento dado, en función de los datos.

Si se tiene el fragmento de código siguiente (pseudocódigo):

```
if (a > 100 AND b < 90) { … } else { … }
```

La herramienta Cobertura permite detectar, además de que en qué casos se pasa por la rama IF y en qué casos se pasa por la rama ELSE, cuantas de las combinaciones que se evalúan en la rama IF se han comprobado. En el ejemplo puesto, puede haber 4 casos posibles, esto es:

- **a** mayor que 100 y **b** menor que 90 → Cierto
- **a** mayor que 100 y **b** mayor o igual que 90 → Falso
- **a** menor o igual que 100 y **b** mayor o igual que 90 → Falso
- **a** menor o igual que 100 y **b** menor que 90 → Falso

Gracias a esta herramienta se han desarrollado más y mejores pruebas unitarias y de integración, ya que además de los porcentajes comentados, permite identificar las clases o paquetes de clases con menor cobertura, qué líneas concretas no se han probado, ramas no cubiertas, etc.

En la Tabla 11 se muestran los resultados obtenidos con esta herramienta a nivel global para el paquete `org.openfidelia.process`, que es el que contiene todas las clases tanto del

Coverage Report - org.openfidelia.process

Package	# Classes	Line Coverage		Branch Coverage		Complexity
org.openfidelia.process	9	84%	79/94	34%	66/191	1,346
org.openfidelia.process.compiler.ast	4	91%	68/74	72%	16/22	1,277
org.openfidelia.process.compiler.ast.factories	4	71%	27/38	29%	7/24	1
org.openfidelia.process.compiler.ast.model	26	82%	430/523	78%	100/128	1,366
org.openfidelia.process.compiler.parser	2	0%	0/2	N/A	N/A	1
org.openfidelia.process.compiler.parser.impl	5	80%	93/116	N/A	N/A	2,364
org.openfidelia.process.compiler.visitors.code	4	86%	415/482	65%	98/150	3
org.openfidelia.process.compiler.visitors.exceptions	2	100%	4/4	N/A	N/A	1
org.openfidelia.process.compiler.visitors.semantic	1	75%	9/12	100%	2/2	1,2
org.openfidelia.process.instrumentation	1	N/A	N/A	N/A	N/A	1
org.openfidelia.process.repository	5	0%	0/1	N/A	N/A	1
org.openfidelia.process.repository.impl	2	95%	19/20	100%	4/4	2,5
org.openfidelia.process.runtime	1	0%	0/2	0%	0/8	0
org.openfidelia.process.script	1	100%	24/24	50%	13/26	0

Tabla 11: Resumen generado por la herramienta Cobertura

compilado como del entorno de ejecución. La columna *Line Coverage* contiene los porcentajes de cobertura de linea de cada paquete. La columna *Branch Coverage* contiene los porcentajes de cobertura de ramas de ejecución, por paquete. Por último, la columna

Complexity, contiene un indice de complejidad de cada paquete. Cuanto mas cerca de 1 este el valor, menos complejidad.

Se puede comprobar que se han cubierto 1168 de 1392 lineas de código, que suponen un 84% de cobertura, y 306 de 555 de ramas de ejecución, un 55% de las mismas.

6 Resultados y conclusiones

6.1 Resumen

Se ha desarrollado satisfactoriamente un sistema sencillo, pero potente, que permite desarrollar y ejecutar programas diseñados de modo gráfico, apoyados en la notación BPMN, Java y scripts Groovy. Se ha comprobado la validez del mismo gracias a suficientes pruebas unitarias y de integración. Además, se han desarrollado varios procesos de prueba más o menos complejos para comprobar que el desarrollo de los programas desde cero es factible.

Unas de las ventajas del sistema radica en que es fácilmente extensible debido al uso de componentes genéricos (nodos), que han de implementarse en función de las necesidades concretas del producto a realizar. Junto con un entorno de ejecución con servicios básicos como gestión de eventos, contextos o un repositorio de procesos, se proporciona un sistema completo y funcional, pero pensado para ser ampliado.

El principal inconveniente detectado es que el desarrollo aún es laborioso porque hay que modificar el código manualmente, ya que el editor, aún siendo el más adecuado disponible, no da soporte para permitir introducir el contenido extendido (si soportado por BPMN) necesario para los procesos aquí descritos.

Además, a la hora de depurar los procesos, falta información de lo que ocurre en tiempo de ejecución, por lo que hay que introducir trazas en los scripts para ir siguiendo la ejecución del mismo.

En conclusión, el sistema permite diseñar y ejecutar desde programas muy sencillos hasta programas arbitrariamente complejos, basados únicamente en la notación BPMN y ciertas normas de diseño impuestas por la implementación desarrollada.

6.2 Líneas futuras

Se detectan múltiples puntos que admiten mejoras, principalmente orientadas a cubrir mas funcionalidad ya soportada por BPMN, o bien a facilitar el desarrollo con el sistema actual, ampliando y adaptando el editor, puesto que es de código abierto.

6.2.1 Compilador / Ejecución

Se identifican las siguientes lineas posibles de trabajo:

- Soporte para la dividir la ejecución y realizar procesamiento paralelo. Supondría dar soporte en el compilador y en el entorno de ejecución a los nodos FORK / JOIN descritos en el apartado 2.4.2.3.3.
- Añadir gestión de piscinas y/o calles, para poder repartir la ejecución de un proceso en varias instancias del motor de procesos (para procesadores multinúcleo o *clusters*).
- Implementar los nodos y eventos de mensajería, necesarios para comunicar nodos en distintas calles o piscinas del punto anterior.
- Implementar nodo de llamada a servicio web. Supondría dar soporte a comunicaciones web para recibir o enviar datos a otros sistemas.
- Implementar un sistema de vistas de usuario, de modo que puedan hacerse programas con interfaz gráfica.
- Implementar los nodos de acceso a datos, para poder persistir o acceder a datos en soportes como ficheros, base de datos, etc. Añadir gestión de transaccionalidad transparente.
- Desarrollar nodos con scripts predefinidos que puedan invocarse desde cualquier proceso.
- Implementar un sistema de trazas para que sea sencillo loguear información desde cualquier nodo.
- Implementar un sistema de monitorización de procesos, de modo que pueda conocerse en cualquier momento qué procesos están en ejecución, en que nodo, cuanto tardan en ejecutarse, etc.
- Dar soporte a temporizadores, tanto eventos como nodos específicos.
- Desarrollar un Plugin para Maven que encuentre y compile directamente ficheros BPMN como si fuera un fichero fuente mas, como Java o Groovy. Serviría para hacer proyectos de cualquier índole con procesos incluidos dentro como parte de su lógica interna.

6.2.2 Editor

El editor de procesos empleado en el presente trabajo, Yaoqiang BPMN Editor, pese a ser un desarrollo muy activo y liberar versiones cada poco tiempo, tiene algunas carencias que impiden que se adapte al 100% a los procesos diseñados.

Se proponen las siguientes mejoras que se podrían incorporar al mismos para poder facilitar el diseño e implementación de los procesos BPMN:

- Dar soporte a los campos extendidos BPMN. El lenguaje diseñado hace uso de los campos `<extensionElements>` definidos en el estándar BPMN, para incluir funcionalidad extendida, como se vio en el apartado . De este modo no habría que editar el XML fuente manualmente.
- Mejoras para poder incluir nodos prediseñados, específicos del negocio, con código predefinido, configurable mediante parámetros.
- Crear un repositorio de procesos ya disponibles para poder incluirlos o invocarlos desde el diseño de un proceso.

7 Referencias y bibliografía

[Activiti, 2018] *Activiti Open Source Business Automation*. Alfresco Software, 2018.
Disponible en https://www.activiti.org

[Apache, 2013] *Apache Commons Digester, The Apache Software Foundation*, 2013.
Disponible en https://commons.apache.org/proper/commons-digester

[Apache, 2018] Apache Maven Project, *The Apache Software Foundation*, 2018.
Disponible en https://maven.apache.org/

[Apache Groovy, 2018a] Apache Groovy, *The Apache Groovy Project*, 2018.
Disponible en http://groovy-lang.org/

[Apache Groovy, 2018b] *Runtime and compile-time metaprogramming, Section 2.1 The Apache Groovy Project, 2018.*
Disponible en http://groovy-lang.org/metaprogramming.html, Sección 2.1

[Bizagi, 2018] Bizagi BPM *Platform*, Bizagi, 2018.
Disponible en https://www.bizagi.com/es/bpm

[Bonitasoft, 2018] Bonita BPM, Bonitasoft, 2018.
Disponible en https://www.bonitasoft.com/products#about-bonita-bpm

[Bosch, 2018] Bosch inubit BPM, Bosch, 2018.
Disponible en https://www.bosch-si.com/bpm-and-brm/inubit-bpm/business-process-management.html

[BPTrends, 2018] BPTrends Associates, 2018.
Disponible en http://www.bptrendsassociates.com

[Camuda, 2018] *BPMN Viewer and Editor*, Camuda y otros, 2018.
Disponible en https://demo.bpmn.io/

[Chinosi and Trombetta 2012] Chinosi, M. & Trombetta, A. *BPMN: An introduction to the standard. Computer Standards & Interfaces*, 34(1):124–134, 2012.

[Codehaus, 2014] Paquete Groovy AST, The Codehaus, 2014.

Disponible en http://docs.groovy-lang.org/2.3.5/html/api/org/codehaus/groovy/ast/package-summary.html

[Codehaus, 2015] Cobertura Maven Plugin, Codehaus, 2015.
Disponible en http://www.mojohaus.org/cobertura-maven-plugin/

[Dumas et al., 2013] Dumas, M., La Rosa, M., Mendling, J., Reijers, H. *Fundamentals of Business Process Management*, Springer, 2013.

[Eclipse, 2018] *BPMN2 Modeler, Eclipse Foundation, Inc.*, 2018.
Disponible en https://www.eclipse.org/bpmn2-modeler/

[Freund and Rücker, 2012] J. Freund and B. Rücker, *Real Life BPMN*, Camunda, 2012.

[Harmond, 2018] *The State of Business Process Management 2018*, Paul Harmon, BPTrends.com, 2018.
Disponible en https://www.bptrends.com/2018-state-of-business-process-management-lp

[Havey 2005] Havey, M. *Essential Business Process Modeling*, 2015, O'Reilly Media, Inc.

[Hesse, 2017] BPMN Tool Matrix, Moritz Hesse, 2017.
Disponible en https://bpmnmatrix.github.io/

[Joget, 2018] Joget Workflow, Joget, 2018.
Disponible en https://www.joget.org/

[Junit, 2018] Junit, The JUnit Team 2018.
Disponible en https://junit.org

[Modeliosoft, 2018] *Modelio Open Source Modeling Environmen,* Modeliosoft, 2018.
Disponible en https://www.modelio.org/

[OASIS, 2007] Web Services Business Process Execution Language Version 2.0, OASIS, 2007.
Disponible en http://docs.oasis-open.org/wsbpel/2.0/wsbpel-v2.0.html

[OMG, 2008a] *Software & Systems Process Engineering Metamodel Specification Version 2.0*, OMG, 2008.

Disponible en https://www.omg.org/spec/SPEM/2.0

[OMG, 2008b] *Business Process Definition Metamodel Specification Version* 1.0, OMG 2008.

Disponible en https://www.omg.org/spec/BPDM/1.0/

[OMG, 2010a] *Business Process Model and Notation (BPMN) Version 2.0, 2010, Object Management Group, OMG Doc. Number: formal/2011-01-0. OMG, 2010.*

Disponible en https://www.omg.org/spec/BPMN/2.0

[OMG, 2010b*] BPMN 2.0 by Example, Version 1.0, OMG, OMG Doc. Number: dtc/2010-06-02*

[OMG, 2017] *Unified Modeling Language Specification Version* 2.5.1

Disponible en https://www.omg.org/spec/UML

[OMG, 2018] *Unified Modeling Language*, OMG, 2018.

Disponible en http://www.uml.org

[Oracle, 2018] Java, Oracle, 2018.

Disponible en https://www.java.com/es

[Recker, 2005] Recker, Jan C. *Process Modelling in the 21st Century*. 2005, BPTrends.

[Recker, 2008] Recker, Jan C. *BPMN Modeling – Who, Where, How and Why*. 2008, BPTrends.

[RedHat, 2018] jBPM, Red Hat, 2018.

Disponible en http://docs.jboss.org/jbpm/release/7.8.0.Final/jbpm-docs/html_single

[von Scheel, 2014] Mark von Rosing, Henrik von Scheel, August-Wilhelm Scheer, *The Complete Business Process Handbook: Body of Knowledge from Process Modeling to BPM, Volume 1*. 2014, Morgan Kaufmann.

[White, 2006] White, Stephen A. *Introduction to BPMN*, 2006, IBM Software Group.

[White, 2012] White, Stephen A. *BPMN: Past, Present, and Future*, 2012, IBM Software Group.

[White and Miers, 2008] White, Stephen A. & Miers, D. *BPMN Modeling and Reference Guide: Understanding and Using BPMN*. 2008, Future Strategies.

[Wikipedia, 2018a] *List of BPMN 2.0 Engines*, Wikipedia, 2018.
Disponible en https://en.wikipedia.org/wiki/List_of_BPMN_2.0_engines

[Wikipedia, 2018b] *Visitor Pattern,* Wikipedia, 2018.
Disponible en https://en.wikipedia.org/wiki/Visitor_pattern

[WfMC, 2005] *XML Process Definition Language Specification*, Version 2.2.
Disponible en http://www.xpdl.org/standards/xpdl-2.2

[Yaoqiang, 2017] Yaoqiang BPMN Editor, Yaoqiang, Inc. 2017.
Disponible en https://sourceforge.net/projects/bpmn

Apéndice A: Glosario

Cluster: Conjunto de sistemas similares que funcionan sincronizados y pueden funcionar como uno solo, repartiendo el trabajo a realizar entre los distintos nodos que lo componen.

Grupo de interés: Conjunto de organizaciones, empresas o personas organizadas en torno a un interés común.

Out-of-the-box: Toda aquella funcionalidad que se puede encontrar disponible tan s olo con instalar un *software*, sin mas configuración o ampliaciones.

Plugin: componente *software* que extiende o modifica la funcionalidad de otro componente *software* mayor, el cual, a su vez, permite estas extensiones.

Servicio Web (o *Web Service*): Interfaz web que expone un sistema para proporcionar un servicio concreto a clientes externos.

Script: Programa escrito para un entorno de ejecución especial que permite automatizar tareas que podrían ser realizadas una a una de modo manual. Normalmente son programas interpretados, no compilados.

Visitor: Patrón de diseño que permite recorre una jerarquía de clases y realizar operaciones sobre los elementos de la misma, sin modificar ni los elementos ni la jerarquía.

Workflow: Flujo de trabajo. Por extensión se conoce así a todos los sistemas que permiten gestionar procesos, y también a los diagramas de flujo que representan dichos procesos.

Apéndice B: Acrónimos

API
Application Programming Interface, Interfaz de Programación de Aplicaciones.

AST
Abstract syntax tree, Árbol de sintaxis abstracta. Estructura en forma de árbol empleada para compilar programas.

BPDM
Business Process Definition Metamodel, Metamodelo para la definición de procesos de Negocio. Es la definición de conceptos estándar empleados para definir modelos de procesos negocio, adoptado por la OMG.

BPEL
Business Process Execution Language, Lenguaje para la ejecución de procesos de negocio.

BPM
Business Process Management, Gestión de procesos de negocio.

BPMN
Business Process Management Notation. Notación BPM desarrollada por la OMG.

jBPM
Java Business Process Model. Modelo de Procesos de Negocio Java.

jPDL
jBPM Process Definition Language, Lenguaje de Definición de Procesos jBPM.

MDA
Model Driven Architecture: Arquitectura dirigida por modelos, propuesta de diseño *software* propuesta por la OMG.

OASIS
Organization for the Advancement of Structured Information Standards, Organización para el desarrollo de estándares para manejo de información estructurada.

OMG
Object Management Group, consorcio empresarial, responsable entre otros de UML, y BPMN.

PFC
Proyecto Fin de Carrera.

SOA
Service Oriented Architecture, Arquitectura orientada a servicios.

SPEM
Software Process Engineering Metamodel, Metamodelo para la Ingeniería de procesos *software*.

TDD
Test Driven Developement. Desarrollo Dirigido por las Pruebas. Paradigma de programación.

UML
Unified Modeling Language, lenguaje unificado de diseño. Estándar de facto para diseño de *software*.

WfMC
Workflow Management Coalition. Coalición para la Gestión de Flujos de Trabajo.

XML
eXtended Mark-up Language: Lenguaje de Marcas Extendido.

XPDL
XML Process Definition Language. Lenguaje de definición de procesos mediante XML.