Institución: Universidad Nacional de La Pampa (UNLPam), Facultad de Ingeniería de General Pico

Actividad curricular: Proyecto Final de Carrera

Título: Aplicando un Enfoque Dirigido por Modelos para construir Servicios Web a partir de Modelos de Procesos de Negocio

Autor: Lucas E. Pereyra

Grado académico alcanzado: Ingeniero en Sistemas

Director: Daniel Riesco, cátedra Análisis y Diseño de Sistemas II

Co-Directora: María Belén Rivera, cátedra Análisis y Diseño de Sistemas I

Fecha y lugar de presentación: 09/10/2020, General Pico, La Pampa

Fecha de aprobación: 09/10/2020

Jurados:

Pablo Becker, Facultad de Ingeniería de General Pico

Guillermo Lafuente, Facultad de Ingeniería de General Pico

María de los Ángeles Martín, Facultad de Ingeniería de General Pico

Resumen: El trabajo presenta un enfoque dirigido por modelos que asigna un rol central a los modelos de procesos de negocio durante la construcción de sistemas orientados a servicios y a microservicios. Se desarrolla una transformación dirigida por modelos que permite obtener de manera automática, la implementación parcial en código fuente de los servicios web representados en un modelo de procesos de negocio BPMN. La transformación se denomina BPMN2REST y convierte un diagrama de colaboraciones BPMN extendido en una implementación parcial de servicios web RESTful, utilizando el framework Spark para Java. La extensión de BPMN; denominada extBPMN, describe qué comunicaciones del modelo representan servicios REST, y contiene la información necesaria para obtener el código fuente, que será interpretada por BPMN2REST. El objetivo de la transformación BPMN2REST es acelerar el proceso de desarrollo de los servicios web RESTful de una empresa, por medio de la generación automática de código fuente. En adición, su uso permitirá que los modelos de procesos de negocio adquieran un rol protagonista, ya que resultarán necesarios para ejecutar la transformación. Esto a su vez, mejorará la consistencia entre los modelos y los servicios web implementados; acercando a los desarrolladores de software con los analistas de los procesos de negocio.

Palabras clave: Desarrollo Dirigido por Modelos, BPMN, REST, Eclipse IDE, Spark Framework

Abstract: This report presents a model-driven approach that assigns a central role to business process models during the construction of service-oriented and microservice-oriented systems. A model-driven transformation is developed. This transformation allows to automatically generate the partial implementation in source code of the web services represented in a BPMN based model. The transformation is called BPMN2REST and receives an extended BPMN collaboration diagram as input to produce a partial RESTful web service based project, using the Java Spark framework. A BPMN extension called extBPMN is provided. It describes which communications from the collaboration diagram represent REST service usages, and contains the necessary information that BPMN2REST will use to generate the source code. The aim of the BPMN2REST transformation is to accelerate the development process of a company's RESTful web services, through the automatic generation of source code. In addition, its use will emphasize the usage of business process models, since they will be required to execute the transformation. This, in turn, will improve the consistency between the models and the implemented web services; bringing software developers closer to business process analysts.

Key words: Model Driven Development, BPMN, REST, Eclipse IDE, Spark Framework

Universidad Nacional de La Pampa
Facultad de Ingeniería

Tesis presentada para obtener el grado de
Ingeniero en Sistemas

Aplicando un Enfoque Dirigido por Modelos para construir Servicios Web a partir de Modelos de Procesos de Negocio

Lucas Pereyra

Directora: Dra. María Belén Rivera
Co-Director: Dr. Daniel Riesco

General Pico, La Pampa
Junio, 2020

Agradecimientos

Agradezco profundamente a Belén Rivera, a quien aplaudo de pie por su rol de directora, demostrando total predisposición, dedicación y profesionalismo durante el transcurso de este trabajo, y dándome su aliento constantemente. También a Daniel Riesco, quien fue el arquitecto inicial de la tesis, brindándome su perspectiva y experiencia en el área; muy necesarias para posibilitar el trabajo.

Agradezco a mi mamá, mi abuela y mi hermano por su constante entusiasmo y apoyo, tanto durante la realización de esta tesis, como también durante el resto de la carrera. Su apoyo fue esencial para poder alcanzar mis metas y objetivos propuestos.

Agradezco a mi novia, Rocío, por apoyarme durante el transcurso del trabajo motivándome a superarme continuamente. Su entusiasmo y positivismo me ayudaron a mantenerme firme en cada etapa del mismo.

Agradezco a mis amigos, familiares, compañeros de estudio, docentes, y cualquier otra persona que en algún momento me haya brindado su apoyo en el estudio de esta hermosa profesión.

Para todos ustedes... ¡muchísimas gracias!

Resumen

En la actualidad, las organizaciones han migrado su estructura de negocios hacia un enfoque cooperativo, mostrando una mayor participación de terceros en sus procesos de negocio. Esta participación, suele presentarse como la solicitud o la prestación de un servicio. Muchos de estos servicios utilizan Internet como su principal canal de comunicación, basándose en el uso de tecnologías de servicios web, donde destaca la arquitectura REST como una de las alternativas más populares en el desarrollo de sistemas web y arquitecturas orientadas a servicios.

Comúnmente, los procesos de negocio de una organización son modelados utilizando el estándar BPMN. Un modelo BPMN puede representar los procesos de negocio junto con los intercambios producidos con terceros, en particular, las invocaciones o prestaciones de servicios web. Así, los modelos de negocios pueden ser utilizados como guía en el desarrollo de los servicios web que brinda una organización. Sin embargo, estos modelos no asumen un rol activo en tal proceso, de manera que pueden quedar obsoletos y no ser consistentes con los desarrollos. Asimismo, la construcción de los servicios debe ser realizada desde sus cimientos por los desarrolladores, dejando detalles de los modelos a libre interpretación de los mismos, lo que puede traer consigo la aparición de errores u omisiones por parte de estos profesionales.

En este trabajo de tesis se presenta un enfoque dirigido por modelos que pretende dar solución a esta problemática, asignando un rol central a los modelos de procesos de negocio. Se presenta una transformación dirigida por modelos que permite obtener de manera automática, la implementación parcial en código fuente de los servicios web representados en un modelo de procesos de negocio BPMN.

La transformación se denomina *BPMN2REST* y convierte un diagrama de colaboraciones BPMN extendido, en una estructura de directorios y clases Java, que conforman la implementación parcial de los servicios web RESTful, bajo el *framework* Spark. La extensión de BPMN; denominada *extBPMN*, describe qué comunicaciones del modelo representan servicios REST, y contiene la información necesaria para obtener el código fuente, que será interpretada por BPMN2REST.

El objetivo de la transformación BPMN2REST es acelerar el proceso de desarrollo de los servicios web RESTful de una empresa, por medio de la generación automática de código fuente. En adición, su uso permitirá que los modelos de procesos de negocio adquieran un rol protagonista, ya que resultarán necesarios para ejecutar la transformación. Esto a su vez, mejorará la consistencia entre los modelos y los servicios web implementados; acercando a los desarrolladores de software con los analistas de los procesos de negocio. Cabe destacar que la transformación podría ser adaptada para que genere la implementación de los servicios web en otras plataformas tecnológicas.

Índice general

1. Introducción **1**
1.1. Motivación . . . 2
1.2. Objetivos . . . 3
1.3. Contribuciones . . . 4
1.4. Organización del trabajo . . . 4

2. Marco teórico **5**
2.1. Procesos de Negocio . . . 6
2.1.1. Introducción al modelado de procesos de negocio . . . 6
2.1.2. Business Process Model and Notation (BPMN) . . . 7
2.2. Desarrollo de Software Dirigido por Modelos . . . 10
2.2.1. Tipos de modelos en MDD . . . 10
2.2.2. Transformaciones en MDD . . . 11
2.2.3. El metamodelado en MDD . . . 11
2.2.4. Arquitectura Dirigida por Modelos . . . 12
2.2.5. El ciclo de vida dirigido por modelos . . . 12
2.2.6. Ventajas de MDD . . . 13
2.3. Arquitecturas Orientadas a Servicios . . . 13
2.3.1. Arquitecturas de Microservicios . . . 14
2.4. Representational State Transfer (REST) . . . 15
2.5. Tecnologías y herramientas de soporte . . . 16
2.5.1. XML . . . 16
2.5.2. OCL . . . 17
2.5.3. Java Spark Framework . . . 18
2.5.4. Eclipse IDE . . . 20
2.5.5. Eclipse BPMN2 Modeler . . . 23
2.5.6. Acceleo . . . 25

3. Estado del arte **29**
3.1. Trabajos relacionados . . . 30
3.2. Herramientas de modelado BPMN 2.0 . . . 33
3.2.1. Evaluación y selección de las herramientas . . . 33
3.2.2. Justificación de la elección . . . 33

4. Desarrollo del trabajo **35**
4.1. Diseño de la transformación BPMN2REST . . . 36
4.1.1. Diseño de BPMN2REST: criterio de transformación . . . 37
4.1.2. Extensión de BPMN para dar soporte a BPMN2REST . . . 39
4.1.3. Definición formal de BPMN2REST . . . 41
4.2. Implementación de la transformación BPMN2REST . . . 42

4.2.1. Extensión de BPMN2 Modeler para la incorporación de extBPMN 42
4.2.2. Especificación de BPMN2REST mediante Acceleo 45
4.3. Definición e implementación de la validación BPMN2REST 50
4.3.1. Definición de las reglas de validación 50
4.3.2. Implementación de la validación empleando BPMN2 Modeler 51
4.4. Construcción del *plugin* BPMN2REST para Eclipse IDE 54
4.4.1. Estructura de la extensión para Eclipse IDE 54
4.4.2. Mejora de la extensión BPMN2REST utilizando la API de Eclipse IDE 55

5. Caso de estudio **61**
5.1. Presentación del caso de estudio 62
5.2. Desarrollo del caso de estudio 62

6. Conclusiones y consideraciones finales **72**
6.1. Cumplimiento de los objetivos 73
6.2. Conclusiones finales 73
6.3. Trabajo futuro 75

Anexos **78**
A1. Elementos gráficos de la notación BPMN 78
A2. Tipos de datos y operadores en OCL 82
A3. Proceso de creación de un proyecto en BPMN2 Modeler 83
A4. Proceso de creación de un proyecto Acceleo 85
A5. Generación automática de *plugins* desde proyectos Acceleo 87
A6. Listado de herramientas de modelado BPMN comunes a los sitios web de referencia . . 88
A7. Estructura de cada punto de extensión incorporado en los desarrollos 89
A7.1. Punto de extensión org.eclipse.bpmn2.modeler.runtime 89
A7.2. Punto de extensión org.eclipse.emf.validation.constraintProviders 90
A7.3. Punto de extensión org.eclipse.emf.validation.constraintBindings 91
A7.4. Punto de extensión org.eclipse.ui.menus 93
A7.5. Punto de extensión org.eclipse.ui.commands 93
A7.6. Punto de extensión org.eclipse.ui.handlers 94
A8. Clases servicio utilizadas en las consultas del módulo *queries.mtl* 94
A9. Manual de instalación del *plugin* BPMN2REST para Eclipse IDE 95

Referencias **96**

Índice de figuras

1.1. Diagrama BPMN que ilustra una doble colaboración entre cliente-empresa y empresa-empresa. En el mismo, se muestran los intercambios producidos para procesar el pago de un cliente. 2

2.1. Fragmento del diagrama de clases que muestra las clases *BaseElement*, *Definitions* y *RootElements*. Fragmento extraído de [2]. 8
2.2. Fragmento del diagrama de clases que muestra la clase *Collaboration*. Fragmento extraído de [2]. 8
2.3. Fragmento del diagrama de clases que muestra las clases *Participant*, *Interface* y *Operation*. Fragmento extraído de [2]. 9
2.4. Fragmento del diagrama de clases que muestra la relación entre *Process*, *FlowElement*, *FlowNode*, entre otras. Fragmento extraído de [2]. 9
2.5. Fragmento del diagrama de clases que muestra la clase *MessageFlow* y sus relaciones. Fragmento extraído de [2]. 10
2.6. Ciclo de vida del desarrollo de software dirigido por modelos. Imagen extraída de [6] . 12
2.7. Los tres pasos principales en el proceso de desarrollo MDD. Imagen extraída de [6]. . . 13
2.8. Resultado de la ejecución del servicio "hello". 19
2.9. Visualización de dos ejemplos de ejecución para los servicios REST definidos anteriormente. En estos ejemplos se ilustra cómo pueden utilizarse los tipos *Request* y *Response* de Spark. 19
2.10. Ejecución de servicio en Spark Framework que muestra el uso de parámetros en las URLs y el manejo del *query string*. 20
2.11. Visualización del IDE y sus distintas vistas. En el centro de la pantalla, se muestra la interfaz de edición de clases Java. En A, se muestra la vista de exploración de paquetes mientras que en B, se puede observar la vista de exploración de proyectos. C Muestra un panel que permite alternar entre la vista de conflictos, Javadoc, declaraciones y la consola de salida. En D se puede observar la vista *Outline* utilizada para mostrar los componentes del archivo que está actualmente siendo editado. 21
2.12. Representación gráfica del mecanismo de extensión de Eclipse IDE. Cada caja representa un desarrollo para la plataforma Eclipse. Los cuadrados en su interior, representan los puntos de extensión que define cada desarrollo, mientras que los círculos representan la utilización de un punto de extensión definido previamente. Las flechas señalan la utilización de puntos de extensión por parte de los *plugins*, y por lo tanto, la relación de dependencia entre los desarrollos. Como se observa, un *plugin* puede utilizar algún punto de extensión definido previamente en el *core* del IDE o en otro *plugin* para añadir su propio comportamiento; o puede definir nuevos puntos de extensión a ser utilizados por otros desarrollos. 22
2.13. Interfaz de configuración de puntos de extensión provista por Eclipse PDE. PDE automatiza la modificación del documento *plugin.xml*. 22
2.14. Visualización de mensajes de error ante una validación fallida de un modelo ejecutada por BPMN2 Modeler. 23
2.15. Fragmento de un diagrama de colaboraciones BPMN visualizado con el editor XML integrado en Eclipse IDE. 24

4.1. Arquitectura del marco de trabajo MDD desarrollado. 36
4.2. Representación gráfica de BPMN2REST. 37

4.3. Vistas de caja negra y de caja blanca de una misma comunicación BPMN. Notar la diferencia entre el nivel de detalle de ambas figuras: en el primer modelo, el *MessageFlow* conecta dos nodos *Participant*; mientras que en el segundo conecta dos nodos *Intermediate Event*. 38

4.4. Representación gráfica de BPMN2REST en base a la extensión introducida. 40

4.5. Representación gráfica de la composición de extBPMN. 41

4.6. Representación gráfica del mapeo realizado por BPMN2REST. Cada elemento de la izquierda genera una combinación de los elementos de la derecha. 42

4.7. Fragmento XML que define la extensión del elemento Operation según el punto de extensión *org.eclipse.bpmn2.modeler.runtime*. 43

4.8. Ventana de configuración del elemento *Message*. En la misma se pueden modificar los atributos nombre y tipo de dato para el elemento. Cada elemento de la notación tiene una ventana de configuración propia. 44

4.9. Fragmento XML que define los valores para el elemento *propertyTab*. 44

4.10. Ventana de configuración del elemento Operation luego de la extensión añadida. . . . 44

4.11. Selección de una extensión de BPMN2 Modeler para ser aplicada sobre un proyecto en Eclipse. Pueden existir varias extensiones en simultáneo, pero sólo una puede ser utilizada en un proyecto a la vez. 45

4.12. Fragmento XML que define los valores para el elemento runtime en el contexto del punto de extensión *org.eclipse.bpmn2.modeler.runtime*. 45

4.13. Ejemplo en código Java de la definición del *callback* asociado a un servicio. El mismo, indica que la implementación del servicio será dada por el método “myOperationImplementation” de la clase MyParticipantController. 47

4.14. Código del módulo “generateMainClass.mtl”. El mismo define cómo se genera la clase Main.java en una ejecución de la transformación. 47

4.15. Código del módulo “generateControllerFiles.mtl”. El mismo define cómo se generan las clases controlador en una ejecución de la transformación. 48

4.16. Código del módulo principal “main.mtl”. 49

4.17. Ventanas de configuración inicial para un proyecto en Acceleo. Entre otras configuraciones, debe indicarse el metamodelo con el que operará la transformación. . 49

4.18. Mecanismo de autompletado integrado en Acceleo. Conforme el usuario escribe el código de la transformación, se le presentan una serie de alternativas de autocompletado para nombres de funciones OCL. Opcionalmente puede mostrarse la documentación de dichas funciones a modo de guía rápida. 50

4.19. Configuración de ejecución generada por Acceleo. En la misma, se puede indicar el proyecto a ser ejecutado, el modelo BPMN de entrada, y el directorio donde deben generarse los archivos. 50

4.20. Ejemplo de notificación de los errores obtenidos al construir un modelo BPMN inválido utilizando BPMN2 Modeler. 52

4.21. Definición de la regla de validación (b) mediante OCL, dentro del punto de extensión *org.eclipse.emf.validation.constraintProviders* incorporado en el archivo *plugin.xml*. . . 53

4.22. Ejemplo de evaluación de las reglas incorporadas. El modelo que aparece en la figura, no contiene ningún mensaje entre participantes, y por lo tanto no cumple la regla (c). 53

4.23. Composición de la extensión BPMN2REST para Eclipse IDE. 55

4.24. Ventana de entrada del directorio de salida para los archivos generados por BPMN2REST. El *path* ingresado es validado en tiempo real, permitiendo que se ejecute la transformación sólo si el mismo referencia a una carpeta existente. 56

4.25. Diagrama de secuencia que ilustra la ejecución del método *run* del *wrapper*. 57

4.26. Diagrama de secuencia que ilustra el proceso de validación del modelo de entrada dentro del *wrapper*. 58

4.27. Diagrama de secuencia que ilustra el proceso ejecutado al dispararse el evento “aceptar”. 59

4.28. Ejecución de BPMN2REST desde la interfaz del usuario de Eclipse IDE. El menú se abre al hacer click derecho sobre un modelo extBPMN, haciéndose visible la opción “Run BPMN 2 REST”. 60

4.29. Diagrama de secuencia que ilustra las acciones realizadas por el *handler* asociado al comando “Run BPMN 2 REST”. 60

5.1. Diagrama de colaboraciones BPMN que detalla el proceso de reserva de viaje. El diagrama fue construido con BPMN2 Modeler. 63
5.2. Interfaz de selección de tipos de proyectos en Eclipse IDE. 64
5.3. Menú contextual abierto para un proyecto desde la vista de *Project Explorer*. Desde aquí se puede ingresar a la configuración de propiedades de un proyecto. 64
5.4. Ventana de configuración de propiedades de un proyecto en Eclipse IDE. 65
5.5. Ventana de configuración del elemento *Collaboration* en BPMN2 Modeler. 66
5.6. Configuración de elementos *Interface* en BPMN2 Modeler, dentro de la ventana de configuración de colaboraciones. 66
5.7. Vista de *Outline* que muestra los elementos BPMN del modelo. 67
5.8. Ventana de configuración de una interfaz BPMN en BPMN2 Modeler. 67
5.9. Configuración de elementos *Operation* en BPMN2 Modeler, dentro de la ventana de configuración de interfaces. 67
5.10. Vista de *Outline* mostrando las operaciones añadidas a una interfaz. 68
5.11. Configuración de las propiedades REST de un elemento *Operation*, conforme a las propiedades extBPMN. 68
5.12. Ventana de configuración de las propiedades de un participante en BPMN2 Modeler. Se muestra la pestaña *Participant*. 68
5.13. Opciones contextuales desplegadas desde la vista de exploración de proyectos al hacer click derecho sobre un diagrama BPMN. 69
5.14. Interfaz de selección del directorio de salida para la transformación BPMN2REST. . . 69
5.15. Código Java de la clase *Main* generado por BPMN2REST. 70
5.16. Fragmento de la clase Java *SistemaReservasController* generada por BPMN2REST. . 70
5.17. Resultado de ejecución del servicio *hacerReserva*. Se utilizó el complemento *RESTED Client* de Mozilla Firefox, para realizar la petición HTTP POST. 71

6.1. Esquema de dependencias utilizadas por los componentes de la característica BPMN2REST para Eclipse IDE. 75

A1.1. Representación gráfica de los tres tipos de evento básicos en BPMN. Imagen extraída de [2]. 78
A1.2. Representación gráfica de los subtipos de evento *message*, *timer* y *conditional*. Imagen adaptada de [2]. 78
A1.3. Representación gráfica de los distintos tipos de actividad en BPMN. Imagen adaptada de [2]. 79
A1.4. Representación gráfica de los distintos tipos de compuerta en BPMN. Imagen extraída de [2]. 80
A1.5. Representación gráfica de los elementos utilizados para el modelado de datos en BPMN. Imagen adaptada de [2]. 81
A1.6. Representación gráfica de los elementos de conexión en BPMN. Imagen adaptada de [2]. 81
A1.7. Representación gráfica de los elementos *pool* y *lane*, y visualización de un mismo ejemplo mediante las vistas de caja negra y de caja blanca. Imagen adaptada de [2]. 82
A3.1. Interfaz de creación de artefactos para un proyecto en Eclipse IDE. 83
A3.2. Interfaz de selección de tipos de diagrama BPMN soportados por BPMN2 Modeler. . 84
A3.3. Interfaz de configuración inicial del diagrama de colaboraciones BPMN. 84
A3.4. Editor gráfico para los diagramas BPMN embebido en Eclipse IDE. En A, se muestra el editor gráfico central. En B, se muestra la paleta de herramientas y elementos gráficos que pueden utilizarse para confeccionar el modelo. En C se muestra la vista de *Outline* con la lista de los elementos activos. 84
A3.5. Ventana de configuración de propiedades para el elemento *Message*. 85
A4.1. Interfaz de selección de tipos de proyecto desplegada al crear un nuevo proyecto en Eclipse IDE. 85
A4.2. Ventana de configuración inicial de un proyecto Acceleo. Desde aquí se pueden añadir los módulos que contendrá el proyecto. 86
A4.3. Interfaz de selección de metamodelos asociados al módulo nuevo añadido. 86

A4.4. Configuración de ejecución generada para un proyecto Acceleo. El campo A permite indicar el proyecto a ser ejecutado. El campo B, indica la clase principal del motor de ejecución del proyecto. Esta clase es generada por Acceleo. El campo C, permite introducir el modelo de entrada sobre el que se ejecutará la transformación. El campo D, permite configurar el directorio de salida donde se generarán los documentos de texto. Los campos A y B son completados por Acceleo automáticamente, mientras que el usuario debe indicar el modelo de entrada y el directorio de salida. 87

A5.1. Interfaz de configuración de un proyecto *plugin* generado desde un proyecto Acceleo. . 88

A5.2. Visualización de un *plugin* generado automáticamente desde un proyecto Acceleo. El botón “Generate Example” permite ejecutar la transformación sobre un modelo de entrada. 88

A7.1. Fragmento XML que muestra la incorporación del punto de extensión *org.eclipse.bpmn2modeler.runtime*. 90

A7.2. Fragmento XML que muestra la incorporación del punto de extensión *org.eclipse.emf.validation.constraintProviders*. 92

A7.3. Fragmento XML que muestra la incorporación del punto de extensión *org.eclipse.emf.validation.constraintBindings*. 92

A7.4. Fragmento XML que muestra la incorporación del punto de extensión *org.eclipse.ui.menus*. 93

A7.5. Fragmento XML que muestra la incorporación del punto de extensión *org.eclipse.ui.commands*. 94

A7.6. Fragmento XML que muestra la incorporación del punto de extensión *org.eclipse.ui.handlers*. 94

Índice de tablas

4.1. Ejemplos de interpretación que asigna BPMN2REST a los atributos extBPMN. 40

4.2. Definición formal de BPMN2REST. 42

4.3. Descripción de las consultas y funciones de usuario incorporadas manualmente a la librería nativa de Acceleo. 46

4.4. Definición formal de las reglas de validación empleando la notación OCL. 52

5.1. Colaboraciones BPMN que serán implementadas como servicios web RESTful. La columna Proveedor identifica al participante que provee el servicio y que, por lo tanto, recibirá una solicitud HTTP con los datos necesarios para poder brindarlo. La columna Cliente identifica al participante que requiere la ejecución del servicio. 65

5.2. Listado de valores para la URL y el método HTTP de acceso a los distintos servicios REST. 65

A2.1. Tipos de dato básicos en OCL. 82

A2.2. Tipos de dato utilizados para representar conjuntos. 82

A6.1. Listado de herramientas de modelado BPMN, presentado en función de la cantidad de menciones en los sitios de referencia. 88

A8.1. Clases servicio empleadas por las consultas del módulo *queries.mtl*. 95

Capítulo 1

Introducción

Este trabajo de tesis es presentado en el contexto del proyecto final requerido para la aprobación de la carrera Ingeniería en Sistemas (plan 2011), dictada en la Facultad de Ingeniería de General Pico, perteneciente a la Universidad Nacional de La Pampa, UNLPam. El capítulo aquí abordado hace de marco introductorio hacia el mismo, mencionando las fuentes que motivaron su realización, los objetivos a ser alcanzados y las contribuciones esperadas de su desarrollo.

1.1. Motivación

Un proceso de negocios define una secuencia de actividades que son realizadas en una organización, con o sin la ayuda de herramientas, con el propósito de obtener un resultado de negocios bien determinado ([1]). Los procesos de negocio de una empresa definen cómo se desempeña la misma en el logro de sus objetivos organizacionales, y constituyen su base operativa, denotando cómo son realizadas las distintas tareas que conforman su núcleo.

El modelado de procesos de negocio es una actividad cuyo propósito es definir el flujo de los procesos de negocio, a partir de la construcción de modelos que permitan representar de manera clara y concisa las tareas que son realizadas en ellos, los recursos involucrados en las mismas, los responsables de ejecutarlas, y los resultados obtenidos ([1]). Esta información es de vital importancia al momento de modificar los procesos, ya sea para adaptarlos a nuevos requerimientos del negocio en sí, o para optimizarlos buscando una mejora en su eficiencia.

En este contexto, el estándar *Business Process Model and Notation* (BPMN) es ampliamente utilizado en la especificación de procesos de negocio, permitiendo construir modelos precisos y formales a partir de la utilización de los elementos gráficos que provee el mismo ([1, 2]).

Con el avance de las tecnologías y el surgimiento de nuevos requerimientos del mercado, los modelos de negocio organizacionales han migrado su estructura hacia un enfoque colaborativo, dando lugar a las comunicaciones negocio a negocio. Este cambio se ha visto reflejado en la integración de los procesos de negocio de las distintas organizaciones, buscando la coparticipación de las mismas en el logro de los objetivos empresariales. En tal caso, es común que las actividades que se suceden "puertas adentro" de una organización requieran la interacción o participación de otros actores externos que actúen como proveedores de servicios de negocio, subsanando las limitaciones y extendiendo el alcance de la primera. Este esquema de colaboraciones permite dar soporte a necesidades de mayor envergadura a través de la composición de servicios de negocio, e impone un cambio de rol entre los actores involucrados que se vuelven prestadores y solicitantes de servicios de manera simultánea, como lo ilustra la figura 1.1.

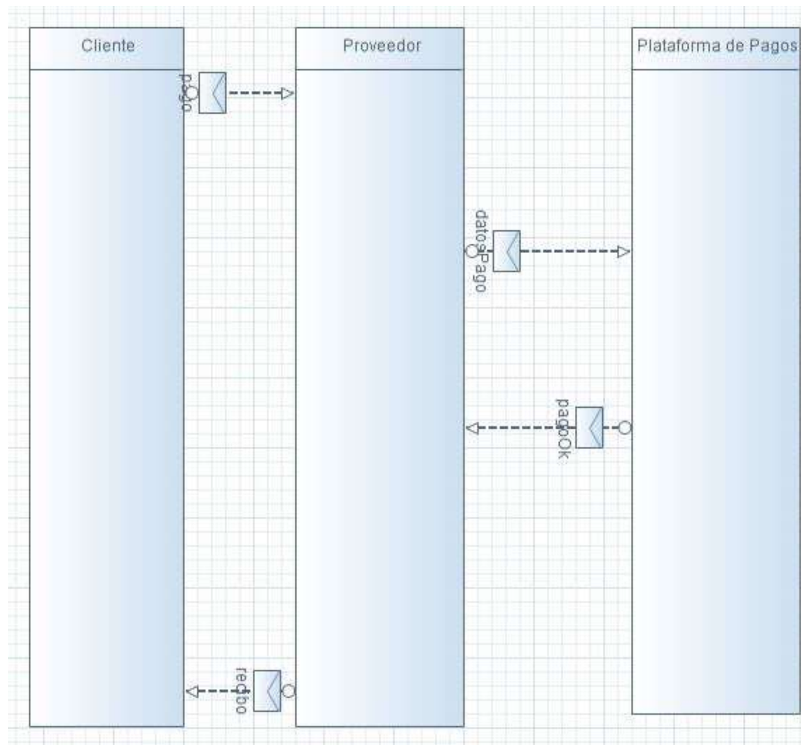


Figura 1.1: Diagrama BPMN que ilustra una doble colaboración entre cliente-empresa y empresa-empresa. En el mismo, se muestran los intercambios producidos para procesar el pago de un cliente.

La evolución de los principales canales y de las tecnologías de comunicación ha permitido dar lugar al uso de Internet como uno de los medios más empleados en el escenario de los intercambios inter-empresariales. Esto último se ha visto potenciado por la digitalización de los artefactos intercambiados, y ha sentado las bases para automatizar en forma total o parcial los servicios proveídos. De esta manera, es posible descomponer los procesos de negocio en servicios bien definidos, que son requeridos por sectores dentro o fuera de una organización, dando lugar a una arquitectura orientada a servicios o SOA ([3]).

Los servicios web o *Web Services* (WS) constituyen una de las tecnologías más utilizadas en los últimos años como mecanismo de invocación/prestación de servicios de negocio mediante Internet, empleando la Web para ello ([3]). En dicho contexto, *REpresentational State Transfer* (REST) ([4]) adquiere un rol protagonista al ser el estilo arquitectónico más empleado actualmente para definir las interfaces de cada servicio.

Para que una organización pueda proveer un servicio a través de la tecnología REST, es necesario diseñar e implementar la interfaz pública y la funcionalidad de dicho servicio, mediante la incorporación e integración de componentes de software que definan el comportamiento requerido. Esta tarea generalmente es llevada a cabo por desarrolladores de software que forman parte de la propia organización, o que son contratados para tal actividad.

El trabajo realizado por los desarrolladores de software, generalmente implica la traducción de modelos de software a código fuente que especifican el comportamiento denotado en los modelos, haciendo uso de las tecnologías concretas de implementación adoptadas para tal desarrollo. En dicho contexto, la utilización de herramientas CASE (*Computer Aided Software Engineering*) permite aumentar la productividad del personal, a partir de la obtención de artefactos de software (documentación, código fuente, archivos de configuración, entre otros) que son generados automáticamente desde los modelos ([5]). Este tipo de herramientas, comúnmente son desarrolladas a partir de técnicas de desarrollo dirigido por modelos o MDD (*Model Driven Development*) ([6]).

En un diagrama BPMN, es posible representar las comunicaciones entre las empresas (ver figura 1.1) indicando la colaboración entre los dos actores del negocio y el consecuente intercambio entre ellos. Estas colaboraciones pueden ser dotadas de cierto significado semántico, por ejemplo, la invocación de algún servicio web RESTful por parte de un actor. Este significado quizás resulta obvio para quien modela el proceso de negocios, pero no para los desarrolladores de software que deberán implementar tales servicios.

De esta manera, resulta de interés que los desarrolladores de software dispongan de una herramienta CASE que les permita obtener la implementación parcial de los servicios web RESTful presentes en un diagrama BPMN, que representa las colaboraciones de un proceso de negocios.

La motivación de esta tesis es la de contribuir con la temática de enfoques dirigidos por modelos para la construcción de servicios web RESTful a partir de colaboraciones en modelos de negocio. Para esto, se presenta la construcción de un complemento para el entorno de desarrollo Eclipse IDE que ejecuta la transformación de colaboraciones a servicios web implementados en base al *framework* Spark[1] para Java.

1.2. Objetivos

El objetivo general del trabajo es el desarrollo de una transformación MDD que permita obtener la implementación parcial en Java (bajo el *framework* Spark) de los servicios web RESTful presentes en los modelos de procesos de negocio BPMN; enfocando tal desarrollo en la construcción de una extensión para Eclipse IDE.

Los objetivos específicos que se desprenden del objetivo general son los siguientes:

- Comprender el Estado del Arte respecto a propuestas existentes para generar definiciones de servicios web RESTful, a partir de colaboraciones en los modelos de negocios.
- Estudiar el marco de trabajo Java Spark Framework para el desarrollo de aplicaciones web basadas en microservicios y el entorno de desarrollo propuesto por Eclipse IDE.
- Diseñar y desarrollar la transformación de BPMN a servicios web RESTful, denominada BPMN2REST.
- Validar los modelos de procesos de negocio BPMN de entrada conforme a los datos mínimos necesarios para generar la implementación de los servicios en Spark.
- Enriquecer la transformación, permitiéndole al usuario indicar datos auxiliares para cada uno de los servicios web tales como la URL o el método HTTP empleado, brindando mayor soporte en la configuración del código fuente generado.
- Desarrollar una extensión para Eclipse IDE que permita hacer uso de la transformación BPMN2REST desde dicha herramienta.
- Validar los desarrollos mediante su aplicación en un caso de estudio concreto.

[1] http://sparkjava.com

1.3. Contribuciones

Esta tesis pretende constituir un importante antecedente en la aplicación de la metodología dirigida por modelos para obtener código fuente a partir de modelos de procesos de negocio. Las principales contribuciones que aporta son:

1. El desarrollo de la transformación BPMN2REST para convertir colaboraciones de modelos de negocio BPMN en servicios web RESTful.
2. El desarrollo de una extensión o *plugin* para Eclipse IDE que implementa dicha transformación.

La confección de un complemento para Eclipse IDE que permita utilizar la transformación BPMN2REST, pone a disposición de los usuarios de la misma de una herramienta CASE que les permitirá aumentar su productividad durante la construcción de los servicios web RESTful de una empresa. Dicha herramienta, acercará de manera consistente al personal encargado de modelar los procesos de negocio, con los desarrolladores de software responsables de implementar los servicios web. Además, la herramienta será puesta a disposición de quienes estén interesados en extenderla ampliando su alcance y funcionalidades.

1.4. Organización del trabajo

El primer capítulo de este trabajo contextualiza al lector respecto al alcance del mismo y lo introduce a las temáticas abordadas.

El capítulo 2 introduce los contenidos teóricos que resultan indispensables para un buen entendimiento de los desarrollos realizados y de los procedimientos seguidos. Los temas son presentados desde un enfoque teórico a fin de ilustrar al lector las bases fundamentales necesarias.

En el capítulo 3 se realiza una breve revisión del estado del arte, haciendo mención a trabajos de investigación relacionados con las contribuciones de esta tesis. Además, se exponen los criterios que fundamentan la elección de Eclipse IDE y Eclipse BPMN2 Modeler como punto de partida de los desarrollos.

El capítulo 4 describe el desarrollo del trabajo a través de las distintas etapas de análisis, diseño e implementación llevadas a cabo en la construcción de la herramienta.

En el capítulo 5 se describe el desarrollo de un caso de estudio, mostrando la aplicabilidad del *plugin* desarrollado que implementa la transformación BPMN2REST en Eclipse IDE.

Por último, el capítulo 6 expone las principales conclusiones derivadas de la realización del trabajo, y las proyecciones que se desprenden del mismo.

Capítulo 2

Marco teórico

El presente capítulo introduce las bases teóricas que resultan necesarias para abordar los contenidos desarrollados a lo largo del trabajo. Se describen los conceptos básicos fundamentales que son mencionados durante el trabajo y que justifican la metodología empleada. Asimismo, se detallan las tecnologías y herramientas de implementación utilizadas, abordándolas desde un punto de vista técnico.

2.1. Procesos de Negocio

Según [1], un proceso de negocios consiste en un conjunto de actividades coordinadas, llevadas a cabo por humanos o herramientas, con el objetivo de lograr un determinado resultado de negocios. Los procesos de negocio definen el orden de realización del trabajo, de manera que están intrínsecamente relacionados con la eficiencia y efectividad de la empresa que los realiza: las empresas podrán operar de manera más eficiente si sus procesos son definidos adecuadamente. En la actualidad, las empresas eficientes destacan sobre sus competidores, dado que esta característica es un factor determinante en el alcance de sus objetivos.

Conocer y entender los detalles de los procesos de negocio resulta de gran importancia, dado que posibilita realizar optimizaciones en los mismos, resolviendo conflictos tales como la formación de cuellos de botella. La optimización de los procesos de negocio permite que sean más eficientes, acelera los flujos de trabajo, reduce la carga de los empleados, y mejora la utilización de los recursos ([1]). En base a lo anterior, la gestión de procesos de negocios es una actividad vital en las empresas. Cada organización debería conocer cómo se definen sus procesos de negocio, quién está involucrado en las distintas actividades, y cuánto demora la ejecución de cada uno de ellos.

2.1.1. Introducción al modelado de procesos de negocio

El modelado de procesos de negocio es una actividad fundamental en la gestión de procesos de negocio, cuyo objetivo es desarrollar modelos de los procesos, que definan en detalle el flujo seguido en cada uno de ellos ([1]). Un modelo de proceso de negocio consiste en una representación del proceso, que denota las actividades que son realizadas y el orden de su ejecución, los resultados obtenidos, los responsables involucrados, y los documentos intercambiados.

El modelado de procesos de negocio es una actividad importante, cuya realización permite satisfacer los siguientes objetivos ([1]):

- Especificar el resultado exacto de los procesos de negocio y entender el valor que tiene dicho resultado para el negocio en sí.
- Entender las actividades y las tareas realizadas en el proceso de negocios.
- Entender el orden de ejecución de las actividades, teniendo en cuenta paralelismos y sincronías entre las mismas.
- Identificar quién es responsable de ejecutar cada actividad y el nivel de responsabilidad que ello implica.
- Conocer el nivel de utilización de los recursos consumidos por el proceso de negocios.
- Conocer las interacciones entre el personal involucrado y el flujo de comunicación entre ellos.
- Conocer el flujo de la documentación.
- Identificar cuellos de botella y potenciales puntos de optimización.
- Introducir estándares de calidad exitosamente y aprobar las certificaciones.
- Guiar a los nuevos empleados, a fin de introducirlos a los procesos de negocio de manera rápida y eficiente.
- Entender a la organización como una totalidad a través de la comprensión de sus procesos.

El modelado de procesos de negocio como tal, es una actividad que debe ser realizada de manera eficiente. Para ello, es necesario utilizar una metodología y una notación que permitan especificar los modelos construidos. Existen múltiples metodologías empleadas en el modelado de procesos de negocio, este trabajo es transversal a cada una de ellas, aunque, por su parte, sólo será aplicable a la notación BPMN.

2.1.2. Business Process Model and Notation (BPMN)

BPMN es un estándar desarrollado por el *Object Management Group* (OMG) que define una notación gráfica con el fin de dar soporte al modelado de procesos de negocio. El objetivo principal de BPMN es proveer un lenguaje que sea fácil de entender por todos los interesados en el negocio, incluyendo a los analistas responsables de crear los bosquejos iniciales de los procesos; los desarrolladores encargados de implementar las tecnologías que serán utilizadas en ellos, y el personal a cargo de gestionar y monitorear dichos procesos. En adición, otra finalidad del estándar es proveer una notación orientada a los negocios para los lenguajes basados en XML que son diseñados para la ejecución de procesos de negocio, tales como *Web Services Business Process Execution Language* (WS-BPEL) ([2]).

La versión 1.0 de BPMN fue publicada en el año 2004, siendo ampliamente utilizada por usuarios y empresas en las actividades de modelado. Esta versión fue adoptada como un estándar por la OMG en el año 2006, y posteriormente, la versión 1.1 fue publicada en el año 2008. En el año 2009 se publicó la versión 1.2, y la 2.0 fue publicada en el año 2011 ([2, 7]). Esta última versión actualmente se encuentra vigente e incorpora múltiples cambios con respecto a las anteriores:

- Incorpora los modelos y diagramas de Conversación y de Coreografía.
- Define un mecanismo de extensibilidad para soportar extensiones en el modelo de la notación y en los elementos gráficos.
- Introduce nuevos subtipos para las actividades, eventos y gateways.
- Extiende la definición de las interacciones humanas.
- Refina la composición y correlación de eventos.
- Extiende los modelos de colaboraciones dando soporte a las interacciones entre múltiples participantes.
- Formaliza las reglas para la ejecución de diagramas.

BPMN es una notación formal que permite modelar procesos de negocio mediante el uso de elementos gráficos. El anexo A1 muestra los principales elementos gráficos de la notación, indicando el significado de cada uno de ellos y su correspondiente representación visual.

Tipos de modelos en BPMN 2.0

Un modelo BPMN puede ser especificado a través del uso de varios diagramas. Existen tres subtipos de modelos que pueden confeccionarse a partir de un modelo BPMN:

- Procesos (*processes*): permiten modelar un proceso de negocios, mostrando el trabajo realizado, los roles involucrados, y los recursos o documentos que intervienen. Puede especificarse con un diagrama de proceso BPMN.
- Colaboraciones (*collaborations*): permiten modelar las interacciones entre los distintos actores o participantes del negocio a través del envío de mensajes. Puede especificarse con un diagrama de colaboraciones BPMN, y puede contener diagramas de proceso y diagramas de coreografía.
- Coreografías (*choreographies*): permiten modelar las interacciones entre los distintos actores involucrados haciendo foco en las comunicaciones existentes y no en el flujo de trabajo realizado. Se modelan con diagramas de coreografías.

Metamodelo de BPMN 2.0

El metamodelo de BPMN ([2]) permite conocer con mayor detalle la manera en que deben relacionarse los elementos de la notación, indicando las restricciones que deben ser respetadas. En adición, ilustra el funcionamiento de los elementos que no tienen una representación gráfica, tales como las interfaces (*interface*) y las operaciones (*operation*). Cualquier modelo BPMN puede verse como un conjunto de instancias de las clases que componen su metamodelo. En base a esto, resulta conveniente realizar un breve acercamiento al mismo, mostrando las principales construcciones internas de la notación.

De acuerdo a la figura 2.1, *BaseElement* es la súper clase abstracta de la mayoría de los elementos de la notación. La misma provee a cada elemento el atributo *id*, utilizado para identificar de forma unívoca al elemento en el modelo BPMN; y de una documentación (*Documentation*) que permite asociarle anotaciones al mismo. La clase *Definitions* actúa como el contenedor principal de todos los elementos de un modelo BPMN, y define el *namespace* y la visibilidad de los mismos. Esta clase es un subtipo de *BaseElement* y juega un papel fundamental en el intercambio de modelos entre distintas herramientas

de modelado BPMN. Sólo existe un elemento *Definitions* por cada modelo BPMN. *RootElement* es el subtipo de *BaseElement* que modela a los elementos directamente contenidos en *Definitions*. Dicho de otra manera, una instancia de *Definitions* sólo puede contener instancias de *RootElement*. Las relaciones son ilustradas empleando el lenguaje estándar de modelado *Unified Modeling Language* (UML).

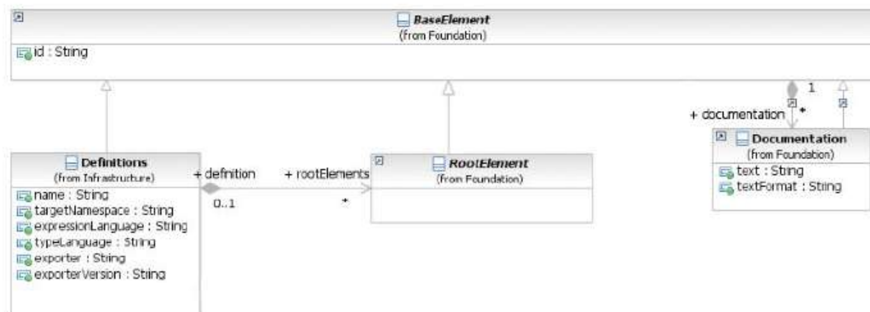


Figura 2.1: Fragmento del diagrama de clases que muestra las clases BaseElement, Definitions y RootElements. Fragmento extraído de [2].

Collaboration es un subtipo de *RootElement* utilizado en los modelos BPMN para definir las comunicaciones existentes en un proceso de negocios. Cada *Collaboration* actúa como el contenedor de los participantes del negocio (*Participant*) y de las interacciones entre los mismos (*MessageFlow*). Una colaboración BPMN es descripta a través de un elemento *Collaboration* que forma parte del modelo construido. La figura 2.2 muestra las principales relaciones de la clase *Collaboration*.

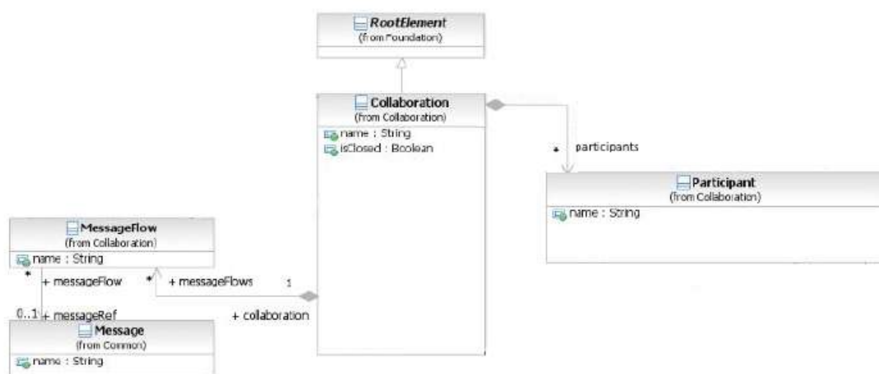


Figura 2.2: Fragmento del diagrama de clases que muestra la clase Collaboration. Fragmento extraído de [2].

La clase *Participant* es utilizada para representar a un actor del negocio. Cada participante, cuenta con su representación gráfica mediante el elemento *Pool*. Los participantes son utilizados para modelar la interacción entre distintos roles o actores del negocio, en el transcurso de los procesos. Es por ello que son contenidos en instancias de *Collaboration*. Se puede modelar el flujo de trabajo seguido dentro de cada participante, asociando un proceso (*Process*) al mismo.

BPMN permite especificar las operaciones que son realizadas dentro de los participantes luego de la recepción de un mensaje. Además, cada operación puede definir un retorno o mensaje de salida como resultado de su ejecución. Esto es soportado a través de la clase *Operation*, y sus relaciones con *Message* mediante los atributos *inMessageRef* (referencia al mensaje de entrada de la operación) y *outMessageRef* (referencia al mensaje de salida). Las operaciones deben ser agrupadas en interfaces con el fin de permitir su reutilización desde distintos participantes. La clase *Interface* brinda soporte para esta característica. Se puede acceder a las interfaces que brinda un participante a través de la relación *interfaceRefs*. Notar en la figura 2.3 que *Interface* es un subtipo de *RootElement*, con lo cual no está contenido en una instancia de *Collaboration*, esto permite su reutilización desde los otros subtipos de *RootElement* (por ejemplo, otras colaboraciones). La figura 2.3 ilustra este conjunto de relaciones entre los elementos de la notación.

La clase *Process* es utilizada para representar a los flujos de trabajo. Un flujo de trabajo en BPMN es construido mediante el uso de elementos de flujo (*FlowElement*). Estos elementos permiten expresar el flujo en términos de nodos (*FlowNode*) dispuestos en secuencia mediante conectores tales como los flujos de secuencia (*SequenceFlow*). Cada flujo de secuencia, indica en qué orden son atravesados dos nodos, identificando un origen (atributo *sourceRef*) y un destino (atributo *targetRef*). Existen tres tipos de nodos: las actividades (*Activity*), los eventos (*Event*) y las compuertas (*Gateway*). *Process* es un subtipo de *FlowElementsContainer*; la súper clase abstracta utilizada para definir contenedores de *FlowElement*. Notar que *Process* extiende a *RootElement* a través de *CallableElement*, que representa

a los elementos que pueden ser invocados (tales como procesos, subprocesos y tareas globales, entre otros). La figura 2.4 muestra la relación entre las clases mencionadas anteriormente.

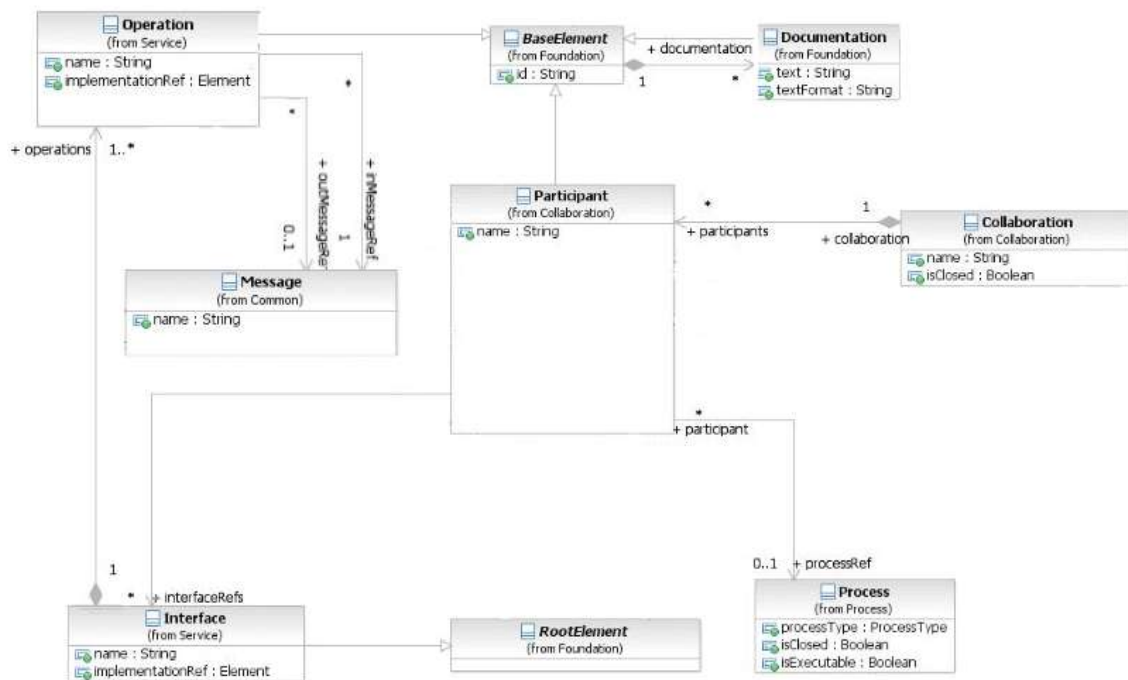


Figura 2.3: Fragmento del diagrama de clases que muestra las clases Participant, Interface y Operation. Fragmento extraído de [2].

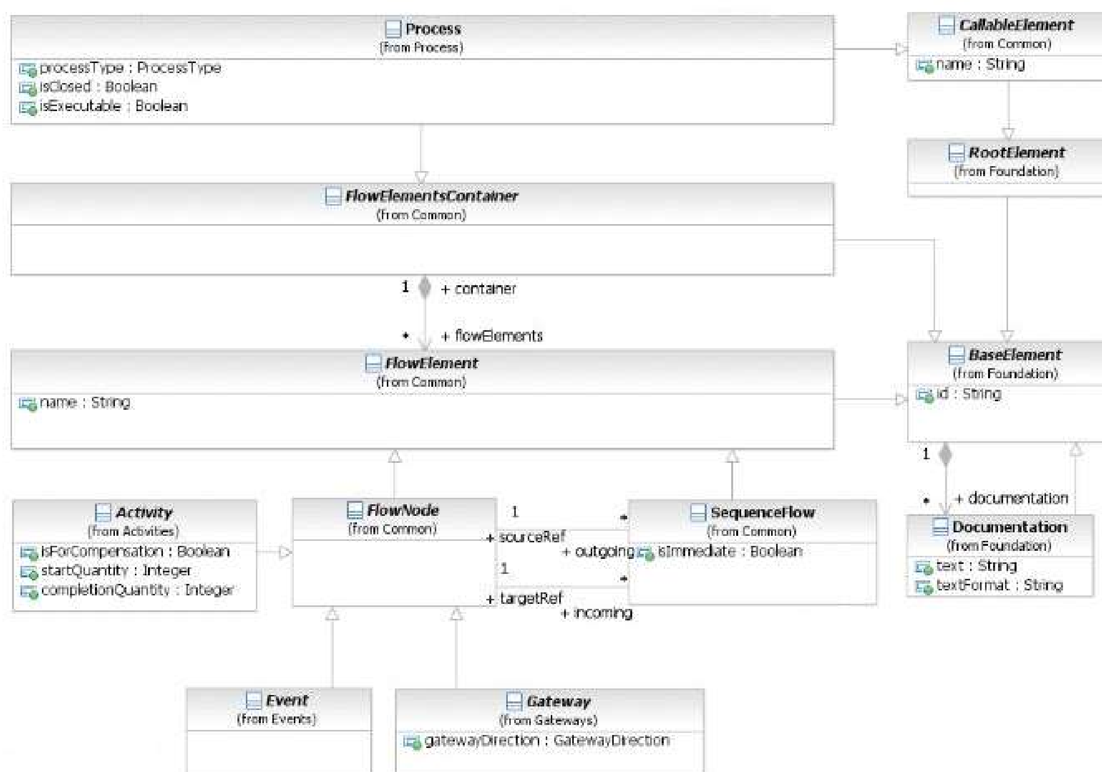


Figura 2.4: Fragmento del diagrama de clases que muestra la relación entre Process, FlowElement, FlowNode, entre otras. Fragmento extraído de [2].

MessageFlow es la clase que utiliza la notación para representar el envío de un mensaje entre participantes. Puede definirse el mensaje del flujo a partir de su atributo *messageRef*. En adición, el flujo debe ser definido en base a un origen (atributo *sourceRef*) y un destino (atributo *targetRef*). Cada flujo de mensaje conecta dos subtipos de *InteractionNode*; que es la clase utilizada para representar elementos que envían o reciben mensajes. Como se observa en la figura 2.5, las clases *Participant*, *Task* y *Event* pueden enviar/recibir mensajes.

Las clases descriptas anteriormente forman parte del núcleo o *core* de la notación BPMN 2.0. Estas clases son instanciadas por las herramientas de modelado BPMN al momento de confeccionar nuevos modelos, haciendo cumplir las restricciones definidas por la notación. Los modelos BPMN son creados a partir del uso de alguna herramienta que implemente el estándar [2]. Cada modelo es guardado en un archivo con la extensión *.bpmn*, que utiliza un lenguaje definido mediante XML para indicar las

relaciones entre las instancias de las clases. La estructura de dicho lenguaje también es definido en el estándar, y funciona de manera análoga a XMI para el caso de modelos UML. De esta manera, los modelos realizados con una herramienta de modelado pueden ser interpretados por otras distintas, permitiendo el intercambio de modelos BPMN.

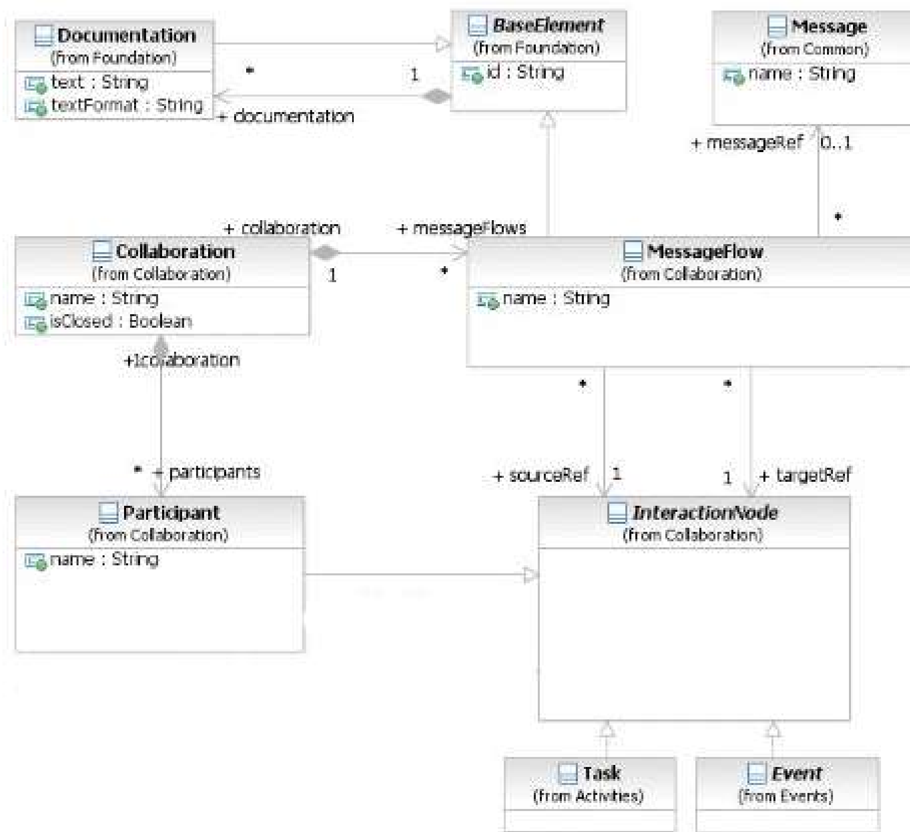


Figura 2.5: Fragmento del diagrama de clases que muestra la clase MessageFlow y sus relaciones. Fragmento extraído de [2].

2.2. Desarrollo de Software Dirigido por Modelos

El Desarrollo de Software Dirigido por Modelos o *Model Driven software Development* (MDD) presenta una iniciativa que se ha extendido en los últimos años, convirtiéndose en un nuevo paradigma de desarrollo de software ([6]). El objetivo de MDD es mejorar el proceso de construcción de software mediante la utilización de modelos y de potentes herramientas automatizadas. El adjetivo "dirigido", enfatiza que este paradigma asigna un rol central y activo a los modelos, siendo al menos tan importantes como el código fuente. Los puntos claves de MDD se identifican a continuación:

- El uso de un mayor nivel de abstracción en la especificación tanto del problema a resolver como de la solución correspondiente, en relación con los métodos tradicionales de desarrollo de software.
- El aumento de la confianza en la automatización asistida por computadora para soportar el análisis, el diseño y la ejecución.
- El uso de estándares industriales como medio para facilitar las comunicaciones, la interacción entre diferentes aplicaciones y productos, y la especialización tecnológica.

2.2.1. Tipos de modelos en MDD

Algunos modelos de software describen los componentes de manera independiente de los conceptos técnicos que involucran su implementación sobre una plataforma concreta; mientras que otros tienen como finalidad primaria definir tales conceptos técnicos ([6]). En base a esta diferencia, los principales tipos de modelos que define MDD son:

- Modelos independientes de la computación o *Computation Independent Models* (CIM): un CIM provee una vista del software independiente de la computación, sin mostrar detalles de la estructura del mismo. Usualmente, el CIM es utilizado para representar el dominio, permitiendo reducir la brecha entre los expertos del dominio y sus requisitos, y los expertos en diseñar y construir software.

- Modelos independientes de la plataforma o *Platform Independent Models* (PIM): un PIM es un modelo con un alto nivel de abstracción, independiente de cualquier tecnología o lenguaje de implementación. El PIM permite modelar el software desde el punto de vista de cómo se soporta de mejor manera al negocio, sin involucrar aspectos relacionados a su implementación. Esto permite que un PIM pueda luego ser implementado sobre distintas plataformas específicas.
- Modelos específicos de la plataforma o *Platform Specific Models* (PSM): un PSM representa una proyección de un PIM sobre una plataforma tecnológica específica. Un PIM puede generar múltiples PSM, cada uno para una tecnología en particular. Generalmente, los PSMs deben colaborar entre sí para brindar una solución completa y consistente.
- Modelos de implementación: un modelo de implementación es representado a través de código fuente que se obtiene de manera directa desde un PSM. El código será desplegado en la plataforma tecnológica concreta utilizada y generalmente, puede ser obtenido en forma total o parcial a partir de la transformación de los PSMs.

2.2.2. Transformaciones en MDD

La transformación entre modelos constituye el motor de MDD. Una transformación de modelos debe ser ejecutada sobre un modelo de entrada, produciendo un modelo de salida. De esta manera, los modelos se generan desde los más abstractos a los más concretos a través de la ejecución de transformaciones y/o refinamientos, hasta obtener el código fuente mediante la aplicación de una última transformación. Cada transformación subsecuente arroja como resultado un modelo más concreto y cercano a la plataforma de implementación. La intervención humana en cada transformación es prácticamente nula, denotando el uso de herramientas automatizadas en dicha tarea ([6]).

Cada transformación entre modelos, se especifica de acuerdo a una definición de transformación que indica cómo debe realizarse el mapeo del modelo fuente en el modelo destino. Generalmente, las transformaciones se especifican relacionando construcciones de un lenguaje fuente en construcciones de un lenguaje destino. Así, una definición de transformación consiste en una colección de reglas no ambiguas que especifican las formas en las que un modelo (o parte de él) puede ser utilizado para crear otro modelo (o parte de él).

Las transformaciones podrían describirse o implementarse utilizando cualquier lenguaje de programación. Sin embargo, para simplificar la tarea de codificar transformaciones se han desarrollado lenguajes de más alto nivel para tal fin. Generalmente, estos lenguajes son conocidos como lenguajes de definición de transformaciones, y permiten describir el mapeo entre los elementos del dominio de entrada y los del de salida.

Existen dos tipos de transformaciones en MDD: las transformaciones modelo a modelo o *Model to Model* (M2M) y las transformaciones modelo a texto o *Model to Text* (M2T). Las transformaciones M2M permiten obtener modelos destino como instancias de metamodelos específicos, mientras que las M2T arrojan como resultado uno o más archivos en algún formato textual. En base a ello, es común que las transformaciones M2M sean especificadas sólo en base a los metamodelos involucrados, mientras que las M2T requieran el uso de alguna plantilla textual auxiliar.

2.2.3. El metamodelado en MDD

A partir del uso de lenguajes de modelado, es posible crear modelos que especifican qué elementos pueden existir en un sistema. Si se define la clase Persona en un modelo, por ejemplo, se podrán tener instancias de Persona, como Juan, Pedro, entre otros. Por otro lado, la definición de un lenguaje de modelado establece qué elementos pueden existir en un modelo. Por ejemplo, el lenguaje UML establece que dentro de un modelo es posible utilizar los conceptos Clase, Atributo, Asociación, Paquete y demás. En base a esta similitud, es posible describir un lenguaje por medio de un modelo, usualmente llamado *metamodelo*. El metamodelo de un lenguaje describe qué elementos pueden ser utilizados en el lenguaje y cómo pueden ser conectados ([6]).

El metamodelado es un mecanismo que permite definir formalmente lenguajes de modelado, tales como UML, por ejemplo. El metamodelado juega un rol importante en MDD, fundado en los tres ejes siguientes:

1. El metamodelado brinda un mecanismo formal para definir lenguajes de modelado sin ambigüedades, permitiendo que herramientas automatizadas puedan interpretar y modificar los modelos.
2. Las reglas de transformación que constituyen la definición de una transformación describen cómo un modelo en un lenguaje fuente puede ser transformado en un modelo en un lenguaje destino. Estas reglas usan los metamodelos de los lenguajes fuente y destino para definir la transformación.

3. La sintaxis de los lenguajes en los cuales se expresan las reglas de transformación también debe estar formalmente definida para permitir su automatización. De manera que el metamodelado también es utilizado con el fin de especificar dicha sintaxis.

2.2.4. Arquitectura Dirigida por Modelos

La Arquitectura Dirigida por Modelos o *Model Driven Architecture* (MDA) es una de las iniciativas más conocida y extendida dentro del ámbito de MDD ([6]). MDA es un concepto promovido por la OMG a partir del año 2000, con el objetivo de afrontar los desafíos de integración de las aplicaciones y los continuos cambios tecnológicos.

En MDA, la funcionalidad del sistema es definida en primer lugar a través de un PIM, contemplando el dominio del que se trate. A partir del PIM, se obtienen los distintos PSM utilizados para la implementación correspondiente bajo los lenguajes específicos del dominio o lenguajes de propósito general como Java, C#, Python, entre otros. El proceso MDA completo se encuentra detallado en un documento que actualiza y mantiene la OMG, denominado la Guía MDA ([8]).

MDA está relacionada con múltiples estándares tales como el *Unified Modeling Language* (UML), el *Meta-Object Facility* (MOF), *XML Metadata Interchange* (XMI), *Enterprise Distributed Object Computing* (EDOC), el *Software Process Engineering Metamodel* (SPEM) y el *Common Warehouse Metamodel* (CWM).

Las dos principales motivaciones de MDA son la interoperabilidad y la portabilidad de los sistemas de software. Además, la OMG postula como objetivo de MDA separar el diseño del sistema tanto de la arquitectura como de las tecnologías de construcción, facilitando así que el diseño y la arquitectura puedan ser alterados independientemente. El diseño alberga los requisitos funcionales mientras que la arquitectura proporciona la infraestructura a través de la cual se hacen efectivos los requisitos no funcionales como la escalabilidad, fiabilidad o rendimiento. MDA impone que el PIM, que representa un diseño conceptual plasmando los requisitos funcionales, resista los cambios que se produzcan en las tecnologías de fabricación y en las arquitecturas de software. La transformación entre modelos es central en MDA, y normalmente es realizada utilizando herramientas automatizadas que permiten al usuario definir sus propias transformaciones.

2.2.5. El ciclo de vida dirigido por modelos

El ciclo de vida del desarrollo de software empleando MDD se muestra en la figura 2.6. Como puede observarse, este ciclo de vida no es muy distinto al ciclo de vida tradicional. Se identifican las mismas fases. Una de las mayores diferencias está en en el tipo de artefactos que se obtienen a lo largo del proceso: la mayoría de ellos consiste en modelos formales que pueden ser interpretados por herramientas automatizadas ([6]).

En el proceso de desarrollo de software tradicional, las transformaciones entre modelos son hechas normalmente mediante la intervención humana. En contraste, la propuesta de MDD persigue la automatización completa de dicha tarea, relegando el rol del personal humano al desarrollo de tales transformaciones. La figura 2.7 muestra las principales etapas del proceso de desarrollo MDD. Las líneas punteadas indican los pasos que son automatizados en el mismo. Como se observa, el proceso inicia luego de la construcción de un PIM que contiene los requerimientos del negocio y los conceptos del dominio. Una primera transformación permite obtener los modelos PSM en base a las tecnologías concretas de implementación. Una segunda transformación permite derivar automáticamente el código fuente del sistema en base a los PSM. Este código puede estar completo y ser totalmente funcional o puede requerir algún nivel de intervención humana final para alcanzar dicho punto.

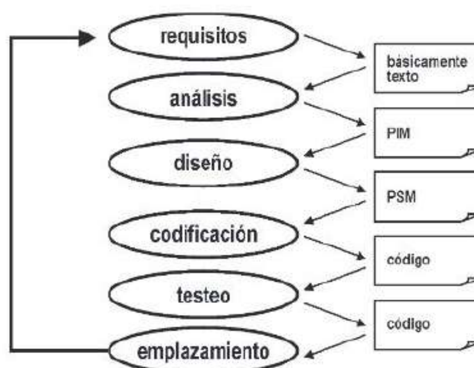


Figura 2.6: Ciclo de vida del desarrollo de software dirigido por modelos. Imagen extraída de [6]

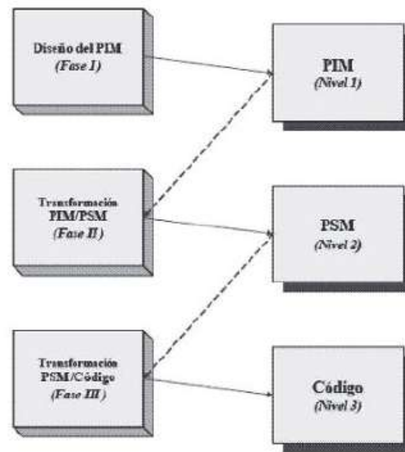


Figura 2.7: Los tres pasos principales en el proceso de desarrollo MDD. Imagen extraída de [6].

2.2.6. Ventajas de MDD

Entre las principales ventajas de la aplicación de MDD se encuentran las siguientes ([6]):

- *Incremento en la productividad*: MDD reduce los costos de desarrollo de software mediante la generación automática de código fuente y de otros artefactos a partir de los modelos.
- *Adaptación a los cambios tecnológicos*: el progreso de la tecnología hace que los componentes de software se vuelvan obsoletos rápidamente. MDD ayuda a solucionar este problema a través de una arquitectura fácil de mantener, donde los cambios se implementan de formar rápida y consistentemente, habilitando una migración eficiente hacia el uso de nuevas tecnologías.
- *Adaptación a los cambios en los requisitos*: al utilizar un proceso MDD, agregar o modificar funcionalidades de negocio se vuelve una tarea sencilla, ya que el trabajo de automatización ya está hecho. Sólo basta con desarrollar el modelo específico para la nueva función. El resto de la información necesaria ya ha sido capturada en las transformaciones y puede ser reutilizada.
- *Consistencia*: los artefactos generados de manera automática serán totalmente consistentes con los modelos de software utilizados.
- *Reutilización*: una vez desarrollados los modelos y las transformaciones, pueden ser reutilizados de manera recurrente amortizando los costos de desarrollo de software y agilizando el mantenimiento del mismo.

2.3. Arquitecturas Orientadas a Servicios

De acuerdo con [9], *Service Oriented Architecture* (SOA) es un término que ha recibido múltiples definiciones, aunque ninguna ha sido aceptada universalmente. Lo que sí tienen con común estas definiciones es la noción de *servicio*. Para SOA, un servicio debe cumplir con las siguientes características:

- Debe ser autocontenido. Cada servicio es un componente altamente modular que puede ser desplegado de manera independiente.
- Debe ser un componente distribuido; disponible en la red y que pueda ser accedido a través de un nombre o localizador.
- Debe tener una interfaz pública; que puede ser conocida por los usuarios sin saber cómo es implementada.
- Debe ser interoperable; de manera que los usuarios y proveedores de servicios pueden usar diferentes lenguajes y plataformas.
- Debe poder descubrirse por los usuarios desde un directorio de servicios en el cual puede estar registrado.
- Debe enlazarse dinámicamente; de manera que el usuario no necesita conocer la implementación del servicio en tiempo de compilación. El servicio es descubierto y enlazado en tiempo de ejecución.

Estas características describen un servicio ideal. En la práctica, algunas de ellas suelen no existir o existir de forma muy relajada, como es el caso de las últimas dos ([9]).

SOA es un estilo arquitectónico donde los sistemas consisten en usuarios y proveedores de servicios. Un estilo arquitectónico comúnmente consta de componentes, conectores y restricciones sobre cómo pueden ser combinados. Para SOA, los componentes básicos son los proveedores y usuarios de los servicios. Los conectores en SOA incluyen llamadas síncronas o asíncronas utilizando *Simple Object Access Protocol* (SOAP), HTTP plano, o alguna otra infraestructura de mensajes. Algunas de las restricciones del estilo arquitectónico SOA son:

- Los usuarios de los servicios envían solicitudes a los proveedores.
- Un proveedor de servicios puede también ser un usuario de servicios.
- Un usuario de servicios puede descubrir proveedores dinámicamente desde un directorio de servicios.

De acuerdo con [9], SOA es un estilo arquitectónico versátil que puede ser implementado siguiendo distintas alternativas:

(a) *SOAP*: esta tecnología es comúnmente la más utilizada para implementar SOA. Las interfaces de los servicios son definidos mediante *Web Services Definition Language* (WSDL), y los usuarios o proveedores de servicios se comunican mediante el protocolo SOAP. Los servicios pueden ser descubiertos desde un registro universal denominado *Universal Description, Discovery and Integration* (UDDI).

(b) *REST*: REST propone una alternativa de implementación mucho más liviana, basada en el uso del protocolo HTTP plano para la comunicación entre usuarios y proveedores. Este mecanismo es mucho más rápido e interoperable que SOAP dado que sólo se necesita soporte nativo para HTTP. Por otro lado, REST impone una interacción entre usuarios y proveedores basada en el acceso a recursos (ver sección 2.4).

(c) *Sistemas de mensajería*: la interacción entre usuarios de servicios y proveedores también puede basarse en el uso de sistemas de mensajería tales como IBM WebSphere MQ, Microsoft MSQM, Oracle AQ y SonicMQ. Estos productos ofrecen altas prestaciones con respecto a la fiabilidad en el envío de mensajes entre usuarios y proveedores. Sin embargo, presentan obstáculos con respecto a la interoperabilidad entre las plataformas que los utilicen.

2.3.1. Arquitecturas de Microservicios

El estilo arquitectónico basado en microservicios es un enfoque para desarrollar aplicaciones estructuradas como un conjunto de servicios de grano fino. Cada servicio es ejecutado en su propio proceso y se comunica utilizando mecanismos livianos tales como APIs REST. Los servicios son desarrollados en base a las funcionalidades del negocio y son desplegados de forma automatizada ([10]).

Los servicios que estructuran una aplicación basada en microservicios son altamente mantenibles y fáciles de testear; desacoplados entre si; desplegables de manera independiente; basados en las funciones del negocio y gestionados por pocas personas ([11]).

Según [10], las principales características de las arquitecturas de microservicios incluyen:

- *Componentes como servicios*: los componentes del sistema deben ser servicios de grano fino que pueden comunicarse entre sí mediante mecanismos que usen la red en lugar de librerías cargadas en memoria.
- *Organización en torno a las funciones del negocio*: el sistema se divide en servicios donde cada uno está organizado en torno a una funcionalidad del negocio. Cada servicio implementa toda la funcionalidad del negocio que agrupa desde la interfaz de usuario, la persistencia en el almacenamiento y cualquiera de las colaboraciones externas.
- *Productos, no proyectos*: en esta arquitectura normalmente se sigue la idea de que un equipo debe estar a cargo de un componente (servicio) durante todo el ciclo de vida del mismo, desde la etapa de diseño y construcción, la fase de producción y hasta la de mantenimiento. El sistema debe ser visto como un conjunto de relaciones entre servicios que evoluciona constantemente.
- *Gobierno descentralizado*: dado que el sistema se compone como un conjunto de múltiples servicios colaborativos, es posible elegir distintas tecnologías de implementación para cada uno de ellos, de acuerdo a las necesidades de la funcionalidad del negocio en cuestión.
- *Gestión de datos descentralizada*: cada servicio debería gestionar su propia base de datos, ya sea que se trate o no de la misma tecnología de base de datos global. La coordinación de los datos entre distintos servicios debería ser no transaccional, dando lugar a una consistencia eventual, aunque compensándolo operativamente.

- *Diseño evolutivo*: cada servicio debe poder ser reemplazado o actualizado de forma independiente permitiendo una fácil evolución. El diseño del servicio tiene que evitar en lo posible que su evolución afecte a sus consumidores.
- *Diseño tolerante a fallos*: las aplicaciones necesitan ser diseñadas de modo que puedan tolerar las fallas de los distintos servicios. Cualquier llamada de servicio puede fallar y el cliente tiene que ser capaz de responder a esto con la mayor facilidad y eficacia posible, evitando los habituales fallos en cascada de las arquitecturas distribuidas.

2.4. Representational State Transfer (REST)

En el año 2000, Roy Fielding identificó una serie de restricciones y principios de diseño que dieron lugar al estilo arquitectónico REST ([4]). REST está orientado a los sistemas hipermedia distribuidos, y define una serie de pautas para direccionar aspectos como la escalabilidad, visibilidad, confiabilidad, separación de tareas y performance.

El protocolo HTTP es adoptado en la mayoría de los sistemas web basados en REST dado que es altamente compatible con los principios que postula. Los principios REST son:

- *Recursos identificables*: un recurso en REST es una pieza de información que puede ser accedida y/o modificada. Este principio demanda que cada recurso debe tener un identificador de recurso que permita localizarlo unívocamente. Este principio es implementado en HTTP mediante el uso de URIs para identificar a cada recurso.
- *Manipulación a través de representaciones*: un recurso puede tener varias formas de representación asociadas. La interacción con los recursos en REST es realizada a través de sus representaciones. Esta restricción es implementada en HTTP mediante la definición de la representación en el *header* o encabezado de las solicitudes, haciendo uso de los tipos de dato MIME[1].
- *Mensajes autodescriptivos*: los mensajes deben contener la información necesaria para poder interpretar las representaciones de los recursos que contengan. Esta restricción es implementada en HTTP mediante el uso de parámetros en el *header* de las solicitudes.
- *Separación entre las capas cliente y servidor*: esta restricción define una separación necesaria entre cliente y servidor e impone una estructura basada en capas para los sistemas. Es implementada de manera directa mediante el uso de HTTP en los sistemas web.
- *Ausencia de estado*: los servidores no pueden mantener un estado con respecto a la sesión de un cliente. En su lugar, la información necesaria deberá enviarse en el cuerpo de los mensajes. Esta restricción debe implementarse por los desarrolladores de los sistemas.
- *Interfaz uniforme*: la interfaz de interacción con los recursos debe estar bien definida y ser uniforme entre los distintos recursos. Esta restricción es implementada en HTTP mediante el uso de los métodos HTTP que provee el protocolo:
    - GET es el método utilizado para consultar un recurso.
    - POST es utilizado para añadir un nuevo recurso al sistema.
    - PUT o PATCH son utilizados para actualizar un recurso existente.
    - DELETE es utilizado para eliminar un recurso.
    - HEAD funciona de manera similar a GET, pero sólo devuelve el código de respuesta y el *header* asociado.
    - OPTIONS es utilizado para obtener las opciones de comunicación configuradas para el recurso.

  En adición, HTTP define una serie de códigos de respuesta que son utilizados en conjunto con los métodos para definir de forma más precisa las interfaces de comunicación ([13]). Los más importantes son:

    - *200 Ok*: indica que la operación fue realizada con éxito. Puede utilizarse como respuesta a los métodos GET, PUT, DELETE y HEAD.
    - *201 Created*: indica que el recurso ha sido creado en el sistema. Puede utilizarse como respuesta a los métodos PUT y POST.
    - *301 Moved Permanently*: indica que los datos solicitados al servidor han sido movidos de forma permanente a otra localización.
    - *302 Moved Temporarily*: indica que los datos solicitados al servidor han sido movidos de forma temporal.

[1] *Multipurpose Internet Mail Extensions* (MIME) es un estándar que indica la naturaleza y formato de un documento, archivo o cadena de bytes ([12]). Algunos de los tipos más utilizados son: *text/html*, *text/plain*, *application/xml*, *application/json*, *multipart/form-data*.

- *403 Forbidden*: indica que no se tienen las credenciales suficientes para acceder al dato solicitado.
- *404 Not Found*: indica que la información solicitada no ha sido encontrada en el servidor.
- *500 Internal server Error*: indica que se ha producido un error en el servidor durante el procesamiento de la solicitud.
- *503 Service Unavailable*: indica que el servidor no puede procesar la solicitud debido a una sobrecarga.

- *Uso de caché*: las respuestas a las solicitudes pueden ser etiquetadas como *cacheables* o *no cacheables*, de manera que los componentes de caché ubicados entre el cliente y servidor puedan interceptar los mensajes y entregar las respuestas. Esta restricción es implementada en HTTP mediante el uso de los parámetros del *header* que permiten controlar el uso de caché.

Una *Application Programming Interface* o API provee una interfaz de funcionalidades que pueden ser utilizadas en la construcción de software, ocultando los detalles de otros componentes y/o sistemas involucrados. Una API REST o API RESTful se estructura como un conjunto de URIs o URLs que identifican a los distintos recursos disponibles en el servidor, y los métodos HTTP que pueden ser utilizados en cada uno de ellos. Mediante el uso de las APIs REST es posible acceder y modificar los recursos por medio de sus representaciones.

Un servicio RESTful se compone por una URI que identifica un recurso en el servidor, y un método HTTP que denota la operación sobre ese recurso. Una API REST se compone de servicios RESTful. Mediante el uso de APIs REST es posible comunicar sistemas del tipo cliente-servidor utilizando únicamente el protocolo HTTP como conector entre ambos. Es por ello que REST es ampliamente utilizado en las arquitecturas SOA y de microservicios como alternativa a los demás protocolos de servicios web.

2.5. Tecnologías y herramientas de soporte

La siguiente sección presenta las herramientas y tecnologías de implementación utilizadas en el desarrollo del trabajo.

2.5.1. XML

Extensible Markup Language (XML), es un estándar desarrollado por la W3C para el almacenamiento e intercambio de la información en base a documentos estructurados según una sintaxis de marcado ([14]). Un documento XML contiene datos y caracteres de marcado que son dispuestos siguiendo una sintaxis bien definida. Esta sintaxis, permite determinar cómo serán organizados los datos en el documento y la estructura lógica del mismo.

La primera versión de XML fue publicada en 1998 por la W3C y ha continuado en vigencia tras múltiples revisiones hasta el año 2004, donde la versión 1.1 surgió en su reemplazo. XML es derivado del estándar *Standard Generalized Markup Language* (SGML), definiéndose como un subconjunto del mismo.

XML fue desarrollado con los siguientes objetivos en consideración:

- Que pueda ser utilizado de manera directa sobre Internet.
- Que sea interoperable entre distintos tipos de aplicaciones.
- Que sea compatible con SGML.
- Que sea fácil escribir programas que procesen documentos XML.
- Que los documentos XML sean fáciles de entender por los humanos y que mantengan la claridad.
- Que sea fácil diseñar y crear documentos XML, y que sea una tarea formal y concisa.

Un documento XML consta de una estructura física y una lógica:

- Físicamente, cada documento se compone de unidades que reciben el nombre de entidades. Una entidad puede referenciar a otras, causando su inclusión en el documento. Un documento comienza con una raíz o entidad superior.
- Lógicamente, los documentos se componen de declaraciones, elementos, comentarios, referencias a caracteres e instrucciones de procesamiento; que son indicados a través del uso de marcado explícito.

El siguiente fragmento muestra un ejemplo de un breve documento XML. El elemento raíz es “personas”, y actúa como el contenedor de los demás. Cada elemento representa una entidad y actúa como un contenedor de datos que podrá ser interpretado por analizadores de XML.

```
<?xml version="1.0" encoding="UTF-8"?>
<personas>
    <persona profesion="cantante" sexo="mujer">
        <nombre>Jane</nombre>
        <apellido>Doe</apellido>
        <fecha_nacimiento>
            <dia>18</dia>
            <mes>06</mes>
            <año>1995</año>
        </fecha_nacimiento>
        <ciudad>La Plata</ciudad>
    </persona>
    <persona profesion="chef" sexo="hombre">
        <nombre>Jhon</nombre>
        <apellido>Doe</apellido>
        <fecha_nacimiento>
            <dia>03</dia>
            <mes>04</mes>
            <año>1993</año>
        </fecha_nacimiento>
        <ciudad>Mendoza</ciudad>
    </persona>
</personas>
```

2.5.2. OCL

Object Constraint Language (OCL) es un lenguaje formal utilizado para describir expresiones en modelos UML. Estas expresiones permiten especificar condiciones invariantes que deben cumplirse en el sistema siendo modelado, o consultas sobre los elementos de dicho modelo ([15]). Los usuarios de UML utilizan las expresiones OCL para especificar restricciones en sus modelos; para especificar operaciones cuya ejecución altera el estado del sistema, o para especificar consultas en el modelo UML de forma independiente a cualquier lenguaje de programación.

OCL es un lenguaje fácil de entender y de escribir que permanece libre de efectos adversos: la evaluación de una expresión OCL no altera los modelos UML ni cambian el estado del sistema, simplemente retorna un valor. OCL no es un lenguaje de programación, por lo cual no es posible invocar procesos o activar funcionalidades fuera de su contexto. OCL es un lenguaje de especificación tipado, con lo cual sus expresiones son evaluadas sobre tipos de dato concretos.

El anexo A2 describe los tipos de dato soportados de forma nativa por OCL, y los operadores que pueden utilizarse para cada uno de ellos.

Expresiones en OCL

Una expresión en OCL debe ser definida sobre un contexto del modelo UML. El contexto determina sobre qué elementos del modelo se aplica la restricción o condición. La palabra reservada *context* del lenguaje se utiliza para definir el contexto de las expresiones.

Una invariante en OCL es un tipo de expresión utilizado para especificar una condición que debe ser verdadera para todas las instancias del tipo definido en el contexto. La palabra reservada *inv* es utilizada para especificar invariantes:

```
context e: Empresa inv:
    e.numberOfEmployees() > 50
```

El ejemplo anterior especifica una invariante para todas las instancias de *Empresa*, donde la condición es tener más de cincuenta empleados.

La palabra reservada *self* puede ser utilizada en las expresiones OCL para referirse a la instancia actual sobre la que debe evaluarse la expresión. Además, es posible asignar un nombre a las invariantes construidas, colocándolo luego de *inv*. De esta manera, el ejemplo puede reescribirse como:

```
context Empresa inv enoughEmployees:
    self.numberOfEmployees() > 50
```

Las expresiones OCL pueden utilizarse para expresar pre- y post-condiciones que forman parte de una operación. Las pre-condiciones son condiciones que deben evaluar como verdaderas antes de la ejecución de la operación, mientras que las post-condiciones deben cumplirse luego de finalizada la misma. La operación debe ser indicada en el contexto, especificando el tipo de dato que la proporciona y la signatura de la misma. Las palabras reservadas *pre* y *post* son utilizadas para definir las pre- y post-condiciones. Dentro de las pre- y post-condiciones, puede utilizarse la palabra reservada *Self*:

```
context Sorteo::registrar(p: Persona): Boolean
    pre: not self.registrados->includes(p)
    post: self.registrados->includes(p)
```

En el ejemplo anterior, la operación denota el registro de una persona en un sorteo. Se define como pre-condición que la persona no haya sido registrada previamente, y como post-condición que el registro haya sido exitoso.

La palabra reservada *result* puede ser utilizada para referir al resultado de la operación. Además, es posible asignar nombres a las pre- y post-condiciones colocándolo luego de *pre* y *post*:

```
context Auto::viajar(a: Ubicacion, b: Ubicacion): Ubicacion
    pre tieneCombustible: self.nivelCombustible > 0
    post llegoADestino: result = b
```

En el ejemplo anterior, la operación describe un viaje en auto desde el punto a hacia el punto b. La pre-condición *tieneCombustible* garantiza que el auto puede partir, mientras que la post-condición *llegoADestino* verifica que haya alcanzado el punto b.

Es posible especificar las operaciones de consulta en OCL. Esto se hace utilizando la palabra reservada *body* para describir el cuerpo de la operación:

```
context Persona::estaCasada(): Boolean
    body: Matrimonio.allInstances()
        ->select(m | m.conyuges->includes(self))
        ->size() > 0
```

2.5.3. Java Spark Framework

Java Spark Framework es un micro-*framework* para el desarrollo web utilizando Java o Kotlin, basado en el *framework Sinatra*[2] para Ruby. Spark es utilizado ampliamente en el desarrollo de sitios web y APIs REST por su simplicidad mediante el uso de Java versión 8 ([16]).

Spark fue desarrollado para proveer una alternativa a los desarrolladores que utilizan Java o Kotlin para crear aplicaciones web de la manera más simple y expresiva posible, reduciendo la curva de aprendizaje en su uso. El *framework* permite el desarrollo y despliegue rápido de aplicaciones web haciendo foco en la productividad, mientras que mantiene la prolijidad y legibilidad del código mediante su sintaxis declarativa ([17]).

Estructura de un proyecto en Spark

Para crear un proyecto en Spark, se deben configurar las rutas web de la aplicación siguiento la sintaxis que propone el *framework*. Esta tarea se hace de forma declarativa dentro del método principal *main*. Una ruta web en Spark consiste de tres partes fundamentales: un verbo, un *path* y un *callback*. El verbo debe identificar al método HTTP utilizado para acceder a la ruta. Puede contener los valores *get*, *post*, *put*, *delete*, *head*, *trace*, *connect* y *options*. El *path* de la ruta identifica la URL mediante la cual puede ser accedida, y debe ser relativo al *host* de la aplicación web. Por último, el *callback* describe lo que sucede durante la invocación del servicio. El *callback* debe recibir dos parámetros (*request* y *response*), y normalmente se especifica utilizando una expresión *lambda* en Java 8.

El diseño de Spark permite construir de manera rápida aplicaciones basadas en servicios o micro-servicios. Cada ruta web descripta en torno a un verbo, *path* y *callback* identifica a un micro-servicio REST que compone la aplicación. Una ruta web en Spark se estructura de la siguiente manera:

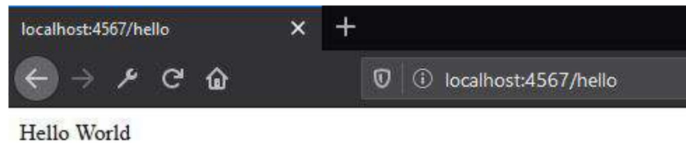
```
<verbo>(<path>, <callback>);
```

[2] http://sinatrarb.com/

El siguiente ejemplo ilustra la definición de una ruta o servicio REST en Spark:

```
public class HelloWorld {

    public static void main(String[] args) {
        get("/hello", (req, res) -> "Hello World");
    }
}
```

Spark utiliza el servidor *Jetty* y el puerto 4567 para alojar los servicios web. Al compilar el código anterior e iniciar el servidor en un entorno de desarrollo, es posible visualizar el funcionamiento del servicio accediendo a *localhost:4567/hello*. La figura 2.8 ilustra el resultado obtenido.

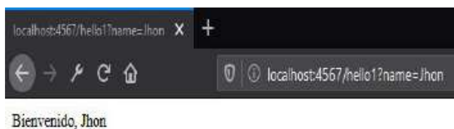


Hello World

Figura 2.8: Resultado de la ejecución del servicio "hello".

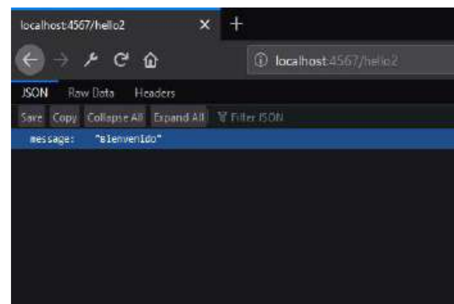
El *callback* asociado a cada ruta debe recibir dos parámetros que son inyectados automáticamente por Spark. El primero, es del tipo *spark.Request* y el segundo del tipo *spark.Response*. Estos parámetros pueden ser utilizados para trabajar con los datos de la solicitud HTTP, y para estructurar la respuesta brindada por el servicio REST, respectivamente. El siguiente ejemplo, ilustra la utilización de los tipos *Request* y *Response* en dos servicios REST. La figura 2.9 muestra los resultados de su ejecución.

```
public class HelloWorld {

    public static void main(String[] args) {

        get("/hello1", (request, response) -> {
            return "Bienvenido, " + request.queryParams("name");
        });

        get("/hello2", (request, response) -> {
            response.type("application/json");
            return "{\"message\":\"Bienvenido\"}";
        });
   }
}
```



Bienvenido, Jhon

(a) Resultado de ejecución del servicio "hello1". Este servicio usa el tipo Request para obtener el valor de un parámetro desde el query string y para utilizarlo en la respuesta.



(b) Resultado de ejecución del servicio "hello2". Este servicio usa el tipo Response para definir el tipo de respuesta del servicio como JSON.

Figura 2.9: Visualización de dos ejemplos de ejecución para los servicios REST definidos anteriormente. En estos ejemplos se ilustra cómo pueden utilizarse los tipos Request y Response de Spark.

Adicionalmente, es posible brindar la implementación de un servicio en Spark utilizando referencias a métodos en Java 8. Esto permite separar la declaración de las rutas de cada servicio, de su correspondiente implementación en Java, como se muestra a continuación:

```
public class HelloWorld {

    public static void main(String[] args) {

        get("/hello1", HelloServices::helloOne);

        get("/hello2", HelloServices::helloTwo);
   }
}

public class HelloServices {

    public static String helloOne(Request req, Response res) {
        return "Bienvenido, " + req.queryParams("name");
    }

    public static String helloTwo(Request req, Response res) {
        res.type("application/json");
        return "{\"message\":\"Bienvenido\"}";
    }
}
```

La URL de un servicio en Spark, debe permitir identificar a un recurso de manera unívoca, conforme al principio de identificación de la arquitectura REST. Para ello, Spark permite la definición de parámetros en los *paths* de cada servicio, que podrán tomar distintos valores de acuerdo a los recursos accesibles desde la API REST. Los parámetros se definen en Spark anteponiendo ":" al nombre del parámetro, dentro del *path* de cada servicio. Se puede acceder al valor del parámetro durante la invocación del servicio con el método *params* de la clase *Request*. El siguiente fragmento muestra un ejemplo de utilización de parámetros en los servicios, y la figura 2.10 muestra el resultado de su ejecución:

```
public class HelloWorld {

    public static void main(String[] args) {

        get("/hello/:id/saludo", (request, response) -> {
            return "Bienvenido, " + request.queryParams("name")
                + " usted tiene el id: " + request.params(":id");
        });
    }
}
```

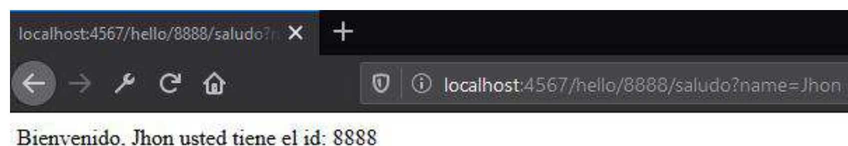


Figura 2.10: Ejecución de servicio en Spark Framework que muestra el uso de parámetros en las URLs y el manejo del query string.

Esta sección describió los fundamentos y nociones básicas de Java Spark Framework necesarias para abordar el trabajo. En [18] se presentan las características del *framework* en mayor profundidad y las distintas configuraciones que soporta según lo requiera cada proyecto.

2.5.4. Eclipse IDE

Eclipse IDE for Java Developers (de ahora en más, Eclipse IDE) es un entorno de desarrollo integrado para Java que forma parte del conjunto de herramientas provisto por la plataforma Eclipse. Eclipse IDE brinda soporte a los desarrolladores de software mediante sus múltiples herramientas y utilidades, y se presenta como una alternativa altamente extensible a partir de la posibilidad de incorporar *plugins* a la misma.

Eclipse IDE forma parte del abanico de herramientas disponibles en la plataforma Eclipse que son soportadas por la Eclipse Foundation. La Eclipse Foundation es responsable de proveer soporte a los usuarios de la plataforma Eclipse por medio de múltiples servicios como lo son la gestión de la propiedad intelectual, la regulación de normativas, la creación de infraestructuras y ecosistemas de desarrollo de software, entre otros ([19]).

Eclipse IDE facilita el desarrollo de software a partir de la automatización de tareas de chequeo y corrección de sintaxis, refactorización del código fuente, gestión de dependencias, gestión de artefactos de configuración, entre otros. El IDE proporciona a sus usuarios múltiples perspectivas y vistas que son utilizadas para acceder a distintas funcionalidades y herramientas según se necesite. Como lo ilustra la figura 2.11, es posible trabajar con múltiples vistas activas en simultáneo para un mismo proyecto.

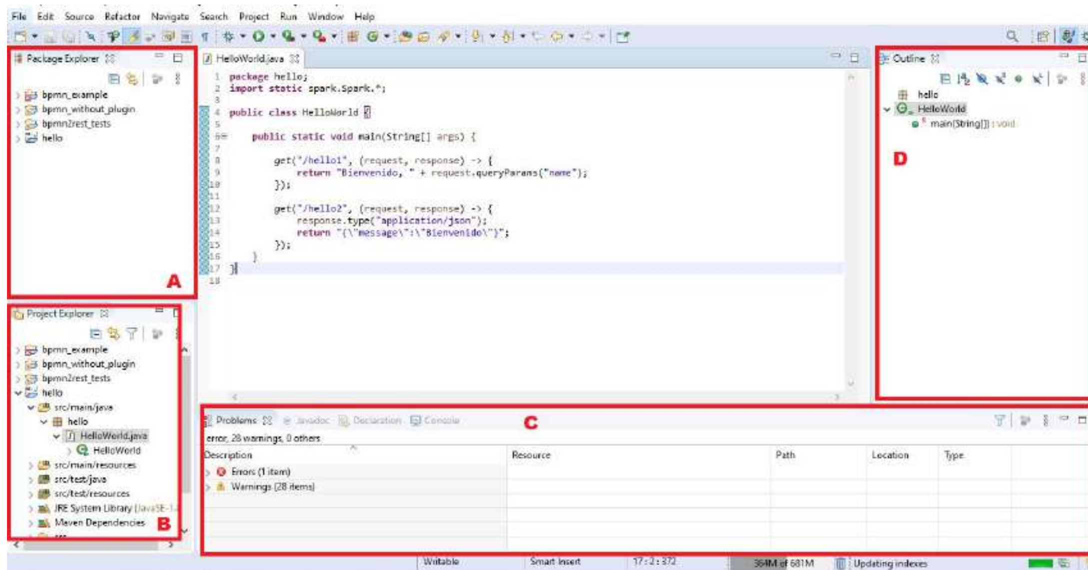


Figura 2.11: Visualización del IDE y sus distintas vistas. En el centro de la pantalla, se muestra la interfaz de edición de clases Java. En A, se muestra la vista de exploración de paquetes mientras que en B, se puede observar la vista de exploración de proyectos. C Muestra un panel que permite alternar entre la vista de conflictos, Javadoc, declaraciones y la consola de salida. En D se puede observar la vista Outline utilizada para mostrar los componentes del archivo que está actualmente siendo editado.

Cada vista permite visualizar y trabajar con distintos componentes o herramientas que provee el IDE. Por su parte, las perspectivas permiten agrupar conjuntos de vistas que pueden complementarse entre sí para brindar un servicio común. En el IDE, pueden coexistir múltiples vistas al mismo tiempo, pero sólo una perspectiva puede estar activa a la vez. Al activar una vista, la misma se muestra ocupando alguno de los distintos paneles del IDE, como se mostró en la figura anterior. Al activar una perspectiva, las vistas que componen dicha perspectiva son activadas en simultáneo. El usuario del IDE puede cerrar o desactivar las vistas que no utilice y abrir otras que no pertenezcan a la perspectiva actualmente activa. Además, el propio IDE define un conjunto de opciones que permiten configurar las vistas asociadas a una perspectiva determinada.

Desarrollo de extensiones para Eclipse IDE

Eclipse IDE puede ser extendido mediante la incorporación de extensiones que permitan hacer uso de nuevas funciones, utilidades o herramientas. Eclipse IDE junto con las demás herramientas que conforman la plataforma Eclipse, fueron desarrolladas prestando especial atención a su extensibilidad. De hecho, el propio IDE puede ser visto como un conjunto de *plugins* que interactúan entre sí.

Los complementos para Eclipse IDE son desarrollados haciendo uso del Eclipse SDK, que provee las utilidades del núcleo o *core* de Eclipse necesarias para integrar cada característica nueva al IDE. Un *plugin* puede encapsular una nueva característica en la forma de librerías de código Java, extensiones de la plataforma, o incluso documentación ([20]).

Un *plugin* para Eclipse IDE es desarrollado a partir de un proyecto del tipo *plugin*, cuya definición depende, en gran medida, de un archivo denominado *plugin.xml*. Este archivo es un documento XML que embebe la información estructural del *plugin* y que define su comportamiento. Este documento, normalmente contiene información sobre los puntos de extensión del *plugin*, ya sea que se trate de la definición de nuevos puntos de extensión, o de la utilización de alguno ya definido ([20]).

Un punto de extensión en Eclipse IDE es un conjunto de configuraciones que definen el comportamiento de una determinada característica añadida en un *plugin*. Los puntos de extensión son el principal mecanismo a partir del cual pueden incorporarse nuevas funcionalidades al IDE. Un mismo *plugin* puede utilizar múltiples puntos de extensión simultáneamente. Generalmente, cada punto de extensión permite añadir una funcionalidad o característica bien definida y granular. Como

lo ilustra la figura 2.12, un *plugin* para Eclipse IDE puede utilizar puntos de extensión previamente definidos para configurar un comportamiento o característica; o puede definir nuevos puntos de extensión que pueden ser utilizados posteriormente por otro desarrollo. Cada punto de extensión definido/utilizado por un *plugin* debe ser declarado en su documento *plugin.xml*. Cada punto de extensión debe indicar cómo puede ser utilizado o configurado. Esta información generalmente se muestra como parte de la documentación del *plugin*.

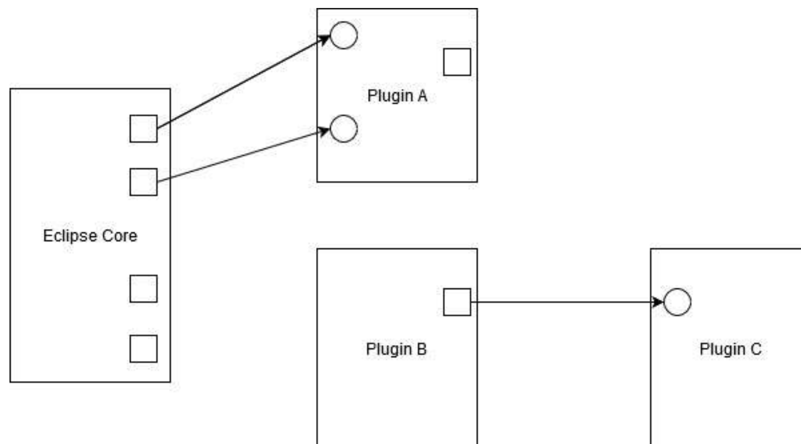
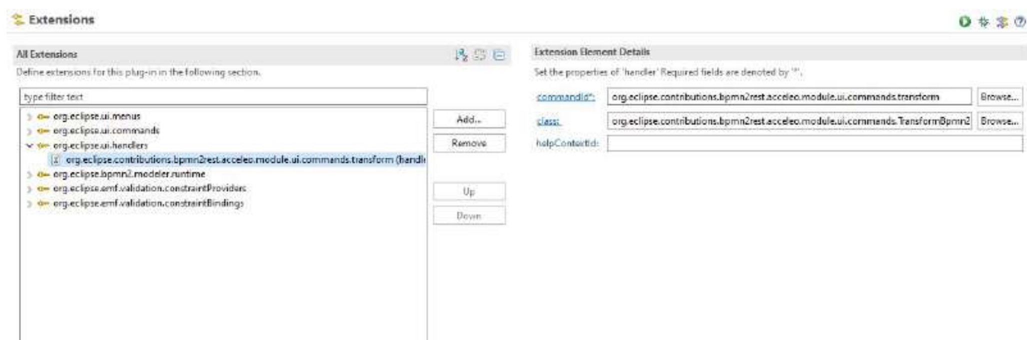


Figura 2.12: Representación gráfica del mecanismo de extensión de Eclipse IDE. Cada caja representa un desarrollo para la plataforma Eclipse. Los cuadrados en su interior, representan los puntos de extensión que define cada desarrollo, mientras que los círculos representan la utilización de un punto de extensión definido previamente. Las flechas señalan la utilización de puntos de extensión por parte de los plugins, y por lo tanto, la relación de dependencia entre los desarrollos. Como se observa, un plugin puede utilizar algún punto de extensión definido previamente en el core del IDE o en otro plugin para añadir su propio comportamiento; o puede definir nuevos puntos de extensión a ser utilizados por otros desarrollos.

Eclipse PDE

Eclipse Plugin Development Environment (de ahora en más, Eclipse PDE) es un conjunto de características desarrolladas para Eclipse IDE que permiten agilizar el desarrollo de *plugins*. En sí, Eclipse PDE es una extensión del IDE que puede ser instalada de manera individual sobre el mismo. Eclipse PDE automatiza funcionalidades relacionadas con el desarrollo de cada *plugin* facilitando dicha tarea para sus usuarios. Mediante el uso de Eclipse PDE, es posible configurar los puntos de extensión de cada *plugin* desarrollado, empleando una interfaz interactiva que evita la modificación directa del documento XML. Esto último puede apreciarse en la figura 2.13.



(a) Ejemplo de configuración de un punto de extensión utilizando la interfaz que provee Eclipse PDE.

```xml
<extension
      point="org.eclipse.ui.handlers">
   <handler
         class="org.eclipse.contributions.bpmn2rest.acceleo.module.ui.commands.TransformBpmn2RestHandler"
         commandId="org.eclipse.contributions.bpmn2rest.acceleo.module.ui.commands.transform">
   </handler>
</extension>
```

(b) Fragmento XML generado automáticamente en el documento plugin.xml.

Figura 2.13: Interfaz de configuración de puntos de extensión provista por Eclipse PDE. PDE automatiza la modificación del documento plugin.xml.

Eclipse PDE también facilita la compresión y exportación de los *plugins* como archivos *JAR* o como características instalables en Eclipse IDE. Una característica instalable consiste en un grupo de *plugins* para el IDE que deben ser utilizados en conjunto. PDE permite exportar múltiples *plugins* como una única característica instalable sobre el IDE. Esto último resulta útil cuando se desea que dos o más *plugins* sean instalados en simultáneo, garantizando que uno de ellos podrá ser utilizado sólo si los demás también están disponibles en el IDE.

2.5.5. Eclipse BPMN2 Modeler

Eclipse BPMN2 Modeler es una herramienta de modelado de procesos de negocio BPMN basada en el uso de componentes gráficos. El objetivo principal de BPMN2 Modeler es proveer un *framework* gráfico de edición de flujos de trabajo que pueda ser fácilmente personalizado por cualquier motor de ejecución que utilice el estándar BPMN 2.0 ([21]).

BPMN2 Modeler extiende las herramientas nativas de Eclipse IDE incorporando un conjunto de utilidades para trabajar con modelos de procesos de negocio BPMN mediante el uso de un entorno de modelado gráfico. BPMN2 Modeler da soporte a los modelos del negocio en tres niveles:

1. Diagramas de procesos.
2. Diagramas de colaboraciones.
3. Coreografías.

Creación de un proyecto en BPMN2 Modeler

La creación de un modelo de procesos de negocio empleando BPMN2 Modeler es una tarea relativamente sencilla. El uso de esta herramienta resulta análogo a cualquier otra herramienta de modelado BPMN. El anexo A3 describe el proceso de creación de un nuevo proyecto utilizando BPMN2 Modeler.

Validación de modelos BPMN

BPMN2 Modeler incorpora un mecanismo de validación de modelos nativo haciendo uso del *plugin* EMF. Esta característica valida los modelos que están siendo confeccionados en tiempo real, mostrando una serie de notificaciones y mensajes al usuario en la vista *Problems*, como se puede observar en la figura 2.14.

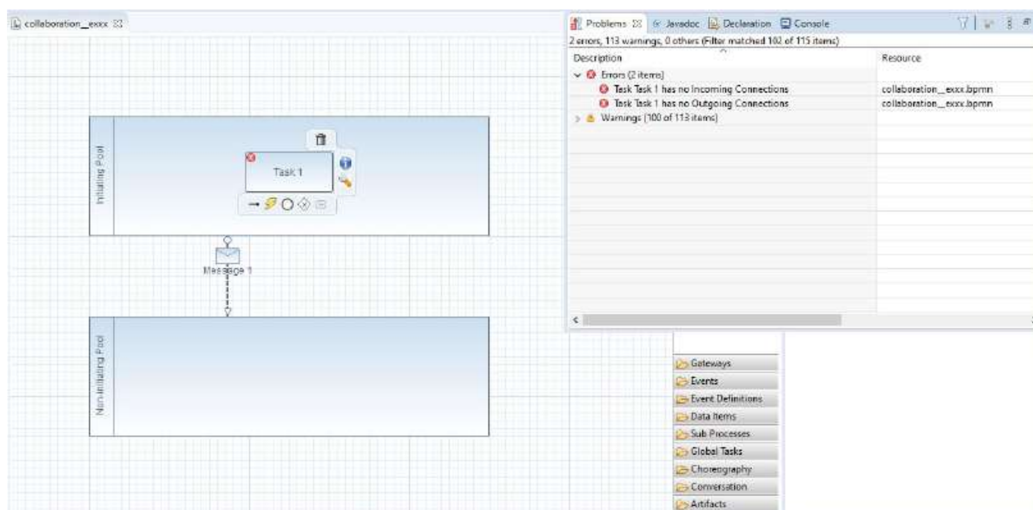


Figura 2.14: Visualización de mensajes de error ante una validación fallida de un modelo ejecutada por BPMN2 Modeler.

El mecanismo de validación que incorpora BPMN2 Modeler tiene como finalidad el cumplimiento de dos objetivos: el primero de ellos es notificar al usuario sobre la existencia de errores sintácticos en el uso de la notación (por ejemplo, un flujo de ejecución sin un evento de inicio asociado). Por su parte, el segundo objetivo es impedir que un modelo pueda ser guardado en un estado erróneo o incorrecto. Esta última situación puede darse ante la ocurrencia de algún error en tiempo de ejecución que genere un documento *.bpmn* sintácticamente inválido.

Documentos BPMN y extensibilidad de BPMN2 Modeler

Un modelo BPMN construido con BPMN2 Modeler es guardado como un documento XML con la extensión *.bpmn*. Este documento embebe la información necesaria para poder interpretar el modelo nuevamente y recrear su organización en base a los elementos gráficos. BPMN2 Modeler emplea la recomendación de la OMG para la representación de modelos BPMN como documentos XML. La figura 2.15 muestra un fragmento de un modelo BPMN visualizado como un documento XML.

BPMN2 Modeler es un conjunto de características que pueden ser instaladas sobre Eclipse IDE y que fueron desarrolladas siguiendo la arquitectura para el desarrollo de extensiones mencionada

anteriormente. BPMN2 Modeler fue desarrollado principalmente con el entorno de desarrollo de herramientas de modelado provisto por *Eclipse Modeling Framework* (EMF).

```xml
<?xml version="1.0" encoding="UTF-8"?>
<!-- origin at X=0.0 Y=0.0 -->
<bpmn2:definitions xmlns:tl="http://www.w3.org/2001/XMLSchema" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:bpmn2="http://www.o
  <bpmn2:message id="Message_2" name="Message 1"/>
  <bpmn2:collaboration id="Collaboration_1" name="Default Collaboration">
    <bpmn2:participant id="Participant_1" name="Initiating Pool" processRef="Process_1"/>
    <bpmn2:participant id="Participant_2" name="Non-initiating Pool" processRef="Process_2"/>
    <bpmn2:messageFlow id="MessageFlow_1" messageRef="Message_2" sourceRef="Participant_1" targetRef="Participant_2"/>
  </bpmn2:collaboration>
  <bpmn2:process id="Process_1" name="Initiating Process" definitionalCollaborationRef="Collaboration_1" isExecutable="false">
    <bpmn2:task id="Task_1" name="Task 1"/>
  </bpmn2:process>
  <bpmn2:process id="Process_2" name="Non-initiating Process" definitionalCollaborationRef="Collaboration_1" isExecutable="false"/>
  <bpmndi:BPMNDiagram id="BPMNDiagram_1" name="Default Collaboration Diagram">
    <bpmndi:BPMNPlane id="BPMNPlane_1" bpmnElement="Collaboration_1">
      <bpmndi:BPMNShape id="BPMNShape_1" bpmnElement="Participant_1" isHorizontal="true">
        <dc:Bounds height="150.0" width="500.0" x="100.0" y="100.0"/>
        <bpmndi:BPMNLabel id="BPMNLabel_1">
          <dc:Bounds height="74.0" width="15.0" x="106.0" y="138.0"/>
        </bpmndi:BPMNLabel>
      </bpmndi:BPMNShape>
      <bpmndi:BPMNShape id="BPMNShape_2" bpmnElement="Participant_2" isHorizontal="true">
        <dc:Bounds height="150.0" width="500.0" x="100.0" y="350.0"/>
        <bpmndi:BPMNLabel id="BPMNLabel_2">
          <dc:Bounds height="101.0" width="15.0" x="106.0" y="374.0"/>
        </bpmndi:BPMNLabel>
      </bpmndi:BPMNShape>
      <bpmndi:BPMNShape id="BPMNShape_Task_1" bpmnElement="Task_1" isExpanded="true">
        <dc:Bounds height="50.0" width="110.0" x="295.0" y="145.0"/>
        <bpmndi:BPMNLabel id="BPMNLabel_4">
          <dc:Bounds height="15.0" width="37.0" x="331.0" y="162.0"/>
        </bpmndi:BPMNLabel>
      </bpmndi:BPMNShape>
      <bpmndi:BPMNEdge id="BPMNEdge_MessageFlow_1" bpmnElement="MessageFlow_1" sourceElement="BPMNShape_1" targetElement="BPMNShape_2">
        <di:waypoint xsi:type="dc:Point" x="273.0" y="250.0"/>
        <di:waypoint xsi:type="dc:Point" x="273.0" y="300.0"/>
        <di:waypoint xsi:type="dc:Point" x="273.0" y="350.0"/>
        <bpmndi:BPMNLabel id="BPMNLabel_3"/>
      </bpmndi:BPMNEdge>
```

Figura 2.15: Fragmento de un diagrama de colaboraciones BPMN visualizado con el editor XML integrado en Eclipse IDE.

BPMN2 Modeler define una serie de puntos de extensión que permiten ampliar sus capacidades nativas mediante el desarrollo de extensiones sobre el mismo. Los puntos de extensión definidos por BPMN2 Modeler permiten:

- Extender el metamodelo de la notación añadiendo nuevos elementos.
- Extender el metamodelo de la notación a partir de un modelo EMF.
- Extender o modificar las pestañas de configuración de propiedades para los elementos de los modelos.
- Extender y añadir nuevas herramientas en el editor gráfico de modelos.
- Modificar la representación gráfica de cada elemento.
- Añadir nuevas reglas de validación para los modelos.

Eclipse Modeling Framework (EMF)

EMF es un *framework* de modelado y una utilidad de generación de código fuente que se utiliza para el desarrollo de herramientas y otras aplicaciones basadas en modelos de datos. EMF fue desarrollado bajo la plataforma Eclipse y permite obtener un conjunto de artefactos generados de manera automática desde un modelo, que pueden ser utilizados para su visualización y edición. Entre los artefactos que genera EMF se encuentran: clases Java para la representación de los modelos, adaptadores que permiten interactuar con el modelo en base a comandos, y una interfaz de edición básica ([22]).

EMF se estructura en base a tres componentes fundamentales:

- *EMF*: el *core* de EMF incluye un metamodelo (denominado *Ecore*) utilizado para describir modelos. Además, también provee soporte en tiempo de ejecución para los mismos, incluyendo algunas características como notificación de cambios, persistencia utilizando XMI, y una API para manipular objetos EMF de manera genérica.
- *EMF.Edit*: incluye clases genéricas y reutilizables para la construcción de editores para los modelos. Provee un conjunto de clases que permiten visualizar los modelos, sus elementos y las propiedades de cada uno empleando los modelos de interfaz de usuario estándares de Eclipse IDE. En adición, incluye un *framework* de comandos, junto con implementaciones genéricas que pueden utilizarse para construir editores para los modelos.
- *EMF.Codegen*: es un conjunto de utilidades que pueden utilizarse para generar de manera automática todos los componentes necesarios para construir un editor completo para un modelo EMF. Incluye una interfaz de configuración desde donde pueden especificarse opciones para la generación de artefactos. Codegen fue construido haciendo uso del Eclipse SDK.

EMF puede ser instalado sobre Eclipse IDE y ser utilizado como una ampliación del mismo. Normalmente, los modelos construidos con EMF representan metamodelos desde los cuales es posible generar automáticamente editores y otras utilidades. Estos artefactos pueden adjuntarse posteriormente en un *plugin* para Eclipse IDE permitiendo crear así una extensión para el IDE que actúe como una herramienta de modelado. Eclipse BPMN2 Modeler fue desarrollado siguiendo este último enfoque.

2.5.6. Acceleo

Acceleo es un entorno de generación de documentos textuales basado en plantillas desarrollado para la plataforma Eclipse. Acceleo es una característica gratuita y de código abierto que puede ser instalada sobre Eclipse IDE y extendida. Esta tecnología es ampliamente utilizada en tareas de generación automática de código fuente y transformaciones del tipo M2T ([23]).

Acceleo ha evolucionado con el desarrollo de nuevas versiones, ampliando sus características. Actualmente, cuenta con un poderoso editor de plantillas que asiste al usuario en el momento de construir la transformación M2T mostrando las distintas directivas y operaciones disponibles. Acceleo permite definir transformaciones M2T desde modelos EMF soportando la utilización de OCL para construir consultas sobre dichos modelos. Para ello, Acceleo cuenta con su propio lenguaje de definición de transformaciones, utilizado para describir el mapeo entre los modelos de entrada y los documentos textuales de salida.

El lenguaje Acceleo es una implementación de MTL que define un conjunto de directivas y construcciones básicas que pueden utilizarse para indicar cómo deben ser generados los documentos textuales de salida. Las directivas se utilizan para definir información general sobre los módulos en Acceleo, mientras que las construcciones permiten controlar el flujo de ejecución de una transformación y el resultado generado. La guía del usuario presentada en [24] expone las directivas y construcciones del lenguaje que pueden utilizarse dentro de un módulo *.mtl*. En este apartado se mostrarán las directivas y construcciones básicas junto a algunos ejemplos de uso.

Un módulo debe comenzar con una directiva *module*, declarando el nombre del módulo y los metamodelos con los que actúa. Cada metamodelo se indica con una URI que lo identifica según EMF. La sintaxis de la directiva *module* es la siguiente:

```
[module <module_name>('metamodel_URI_1', 'metamodel_URI_2') /]
```

Una plantilla en Acceleo se declara como una función que recibe un parámetro de entrada y genera un fragmento de texto. El parámetro de entrada normalmente se corresponde con alguno de los tipos definidos en los metamodelos del módulo. La plantilla entonces, indica cómo se mapea uno o más elementos del metamodelo en un fragmento del documento de texto de salida. Las plantillas se generan con la directiva *template* según la siguiente sintaxis:

```
[template public <template_name>(<param_name> : <param_type>)]
    ...template content
[/template]
```

La directiva *template* a diferencia de *module*, es una directiva que consta de una etiqueta de apertura y una de cierre. Dentro de estas etiquetas se define el cuerpo de la plantilla y pueden utilizarse consultas y construcciones del lenguaje para interactuar con el parámetro de entrada. El siguiente ejemplo muestra un módulo *.mtl* que genera un listado con todos los nombres de las clases contenidas en un modelo UML:

```
[comment encoding = UTF-8 /]
[module myModule('http://www.eclipse.org/uml2/2.1.0/UML') /]

[template public myTemplate(c: Class)]
    Nombre de la clase: [c.name /]
[/template]
```

El elemento *comment* puede ser una etiqueta abierta o no (como en el ejemplo) y se utiliza para introducir comentarios en las transformaciones. Algunos comentarios tienen un significado especial para el motor de transformación, tal como es el caso del comentario *encoding* utilizado en el ejemplo para indicar el conjunto de caracteres con los que se trabaja en el módulo.

Dentro de las plantillas, las construcciones en OCL denotadas entre "[" y "/]" serán interpretadas por el motor de ejecución de Acceleo, produciendo el reemplazo de esa construcción por el resultado de evaluarla. Dentro de estos marcadores pueden colocarse consultas e invocaciones de operaciones en OCL. El lenguaje Acceleo soporta un gran abanico de operaciones OCL definidas para cada tipo de dato particular. En [25] y [26] se listan las operaciones que pueden utilizarse.

Para construir transformaciones M2T es necesario conocer los detalles del metamodelo de entrada sobre el que se ejecutará la transformación. En el ejemplo anterior, el atributo *name* es una propiedad representada mediante un *String* que existe en el elemento *Class*. Esta información se detalla en el metamodelo de UML.

Dentro de una plantilla, es posible utilizar variables que han sido previamente inicializadas, reutilizando su valor a lo largo de las distintas construcciones del lenguaje. La inicialización de variables debe colocarse entre "{" y "}", luego del comienzo de la plantilla. Pueden inicializarse múltiples variables separadas por ";". El siguiente ejemplo muestra el uso de variables en las plantillas:

```
[comment encoding = UTF-8 /]
[module myModule('http://www.eclipse.org/uml2/2.1.0/UML') /]

[template public myTemplate(c: Class) {
    year : Integer = 2020;
    className = c.name;
}]
    Nombre de la clase: [className /]
    Año: [year /]
[/template]
```

Una consulta en un módulo es utilizada para extraer información del modelo de entrada. Las consultas utilizan la notación OCL para recorrer los elementos de los modelos y retornar un valor o una colección de valores. Cada consulta se define mediante la directiva *query* según la siguiente sintaxis:

```
[query public <query_name>({<param_name> : <param_type>}*) : <return_type> =
    <query_content>
/]
```

Una consulta define una función que puede ser invocada sobre uno o más elementos utilizando OCL dentro de una plantilla. Una consulta puede recibir cero o más parámetros y devolver un valor simple o un conjunto de ellos utilizando los tipos *Set*, *Bag* y *Sequence*. Normalmente, es una buena práctica separar los módulos *.mtl* que contienen plantillas de los que contienen consultas. El siguiente módulo ejemplifica el uso de la directiva *query* definiendo algunas consultas:

```
[comment encoding = UTF-8 /]
[module queries('http://www.eclipse.org/uml2/2.1.0/UML') /]

[query public isAbstract(c: Class) : Boolean =
    c.isAbstract()
/]

[query public getPublicAttributes(c : Class) : Set(Property) =
    c.attributes->select(visibility = VisibilityKind::Public)
/]

[query public getAttributeByName(c: Class, String : s) : Property =
    c.attributes->select(name = s)->asSequence()->first()
/]
```

Es posible invocar a un método Java dentro de una consulta en Acceleo. Esto es útil cuando resulta más adecuado resolver la consulta mediante construcciones en el lenguaje Java. Para invocar a un método Java, se utiliza la función *invoke*. El siguiente ejemplo muestra el uso de una consulta que delega su ejecución a una clase Java:

```
[query public getSupertypes(c : Class) : Set(Class) =
    invoke(
        'myPackage.services.UMLServices',
        'getSupertypes(org.eclipse.uml2.uml.Class)'
    )
/]
```

Al ejecutar esta consulta, Acceleo generará una instancia de *myPackage.services.UMLServices* e invocará el método *getSupertypes* de la misma, enviándole una instancia de *org.eclipse.uml2.uml.Class* que contiene los datos del elemento *c*. La clase *org.eclipse.uml2.uml.Class* es provista por EMF y actúa como una API para interactuar con datos de modelos UML utilizando clases Java. Es necesario

que el método *getSupertypes* retorne un tipo de dato *Set<Class>* para mantener la consistencia con la interfaz de la consulta declarada en el módulo. Las clases invocadas por consultas en Acceleo, normalmente reciben el nombre de clases "servicio".

Para poder utilizar las consultas y plantillas definidas en un módulo en cualquier otro, es necesario importarlo utilizando la directiva *import*. Pueden importarse varios módulos utilizando múltiples directivas *import*. Esta directiva debe ir al inicio, luego de *module* y su sintaxis se define como sigue:

```
[import <qualified_name_of_module> /]
```

La construcción *for* puede ser utilizada para iterar sobre una colección de elementos de un modelo y generar un fragmento de texto por cada iteración realizada. Su sintaxis se define como sigue:

```
[for (<var_name> : <var_type> | <collection>)]
    <for_content>
[/for]
```

La construcción *if* puede ser utilizada para generar un fragmento de texto de acuerdo al valor de un condicional. Su sintaxis se define como sigue:

```
[if (<condition>)]
    <if_content>
[/if]
```

Las construcciones *for* e *if* normalmente forman parte de las plantillas que define un módulo. El siguiente ejemplo muestra el uso de la directiva *import* para importar los módulos definidos previamente, junto con el uso de las construcciones *for* e *if*:

```
[comment encoding = UTF-8 /]
[module generate('http://www.eclipse.org/uml2/2.1.0/UML') /]
[import myPackage::acceleo::myModule /]
[import myPackage::acceleo::queries /]

[template public generate(Class : c)]

    [comment @main/]
    [if (not c.isAbstract())]
        [myTemplate(c) /]
        [for (p : Property | getPublicAttributes(c))]
        Atributo [p.name/] del tipo [p.type/]
        [/for]
    [/if]

[/template]
```

En el ejemplo anterior, los módulos *myModule* y *queries* son importados al módulo *generate*. Esta importación permite hacer uso de la plantilla *myTemplate* y de la consulta *getPublicAttributes*. Al momento de ejecutar una transformación generando el documento de texto, Acceleo no conoce desde dónde debe comenzar la interpretación de las plantillas. El comentario especial *@main* permite indicar cuál es la plantilla principal desde la cual debe comenzar el procesamiento. El marcador *@main* debe ir dentro de la plantilla principal, encerrado en las etiquetas *template*. En el ejemplo anterior, *generate* es la plantilla desde la cual comienza la interpretación durante una ejecución.

Suponiendo que la transformación se ejecuta sobre un diagrama de clases UML que tiene una única clase "Persona" con las siguientes propiedades:

- *public int edad*
- *protected Date fechaNacimiento*
- *public String nombre*

Entonces la ejecución de la transformación generará un documento de texto con la siguiente estructura:

```
Nombre de la clase: Persona
Atributo edad del tipo int
Atributo nombre del tipo String
```

Finalmente, la construcción *file* puede utilizarse en una plantilla para indicar el tipo de archivo que debe generar la transformación como resultado de su ejecución:

```
[comment encoding = UTF-8 /]
[module generate('http://www.eclipse.org/uml2/2.1.0/UML') /]
[import myPackage::acceleo::myModule /]
[import myPackage::acceleo::queries /]

[template public generate(Class : c)]
    [file('/files/descripcion_'.concat(c.name).concat('.txt'), false)]
    [comment @main/]
    [if (not c.isAbstract())]
        [myTemplate(c) /]
        [for (p : Property | getPublicAttributes(c))]
        Atributo [p.name/] del tipo [p.type/]
        [/for]
    [/if]
    [/file]
[/template]
```

La directiva *file* recibe como primer parámetro la URL destino relativa al proyecto donde se guardará el archivo generado. Esta URL puede finalizar con alguna extensión que indique el tipo de documento de salida generado. El segundo parámetro de *file* indica si el resultado de la transformación debe ser concatenado al contenido del archivo (en caso de que exista), o si debe sobreescribir su contenido nuevamente eliminando el contenido anterior. Al ejecutar nuevamente esta transformación sobre el modelo del ejemplo, el resultado es volcado sobre el archivo "files/descripcion_Persona.txt".

Los módulos Acceleo describen la estructura de la transformación M2T que se va a ejecutar. Esta transformación, debe ser encapsulada en un proyecto del tipo *Acceleo*. El anexo A4 describe el proceso de creación de un proyecto Acceleo, en el contexto de Eclipse IDE.

Finalmente, Acceleo brinda soporte a sus usuarios para desarrollar *plugins* para Eclipse IDE a partir de proyectos del tipo Acceleo. Estos *plugins* permiten ejecutar las transformaciones M2T desarrolladas desde el mismo entorno del IDE tal y como si fueran características nativas del mismo. El anexo A5 muestra con detalle el soporte que brinda la herramienta para llevar a cabo esta tarea.

Capítulo 3

Estado del arte

De acuerdo con el alcance planteado en el presente trabajo, es necesario realizar un estudio del estado del arte con respecto a las distintas temáticas abordadas, a fin de vislumbrar el punto de partida de los desarrollos. Así, el presente capítulo hace mención a trabajos de investigación que constituyen antecedentes importantes con respecto a los distintos contenidos tratados, haciendo foco en las similitudes y/o diferencias existentes en la forma de resolver la problemática presentada. Junto con ello, se muestra el proceso de selección de la herramienta de modelado BPMN adoptada como el núcleo de los desarrollos, mostrando la oferta disponible y los criterios aplicados en su selección. Por el alcance que implica esta tesis de grado, no se realizó una revisión sistemática de la literatura para seleccionar los trabajos relacionados. En su lugar, se estudiaron los aportes más relevantes al tema, encontrados en fuentes académicas.

3.1. Trabajos relacionados

Muchos trabajos de investigación tienen varios puntos en común con la propuesta presentada en esta tesis. En general, la aplicación de MDD a modelos abstractos a fin de obtener modelos y artefactos más concretos es un eje común recurrente. No obstante, la mayoría difiere en aspectos tales como la forma de aplicar MDD; los modelos involucrados; y los artefactos obtenidos mediante su aplicación. A continuación, se presentan una serie de trabajos relacionados, mostrando similitudes y diferencias con respecto a la propuesta de este trabajo de tesis.

En [7] se presenta una propuesta para generar automáticamente código Java desde un modelo BPMN 2.0 ejecutable[1]. La propuesta muestra que a partir de la información contenida en un diagrama BPMN ejecutable se puede obtener un prototipo de una aplicación *Enterprise Java EE*. En primer lugar, se aplica la metodología MDA y se convierte al modelo BPMN en un PIM denotado con un diagrama de clases UML. A partir del mismo, se obtiene un diagrama de clases UML específico para la plataforma Java EE, desde el cual es posible obtener el código Java a ser desplegado. Este trabajo utiliza el estándar *Query-View-Transformation* (QVT) para definir las transformaciones entre modelos.

En [27] se desarrollan transformaciones M2M para obtener modelos PSMs a partir de un PIM del sistema, modelado con UML. El trabajo procede sobre un caso de estudio concreto, obteniendo un PIM que denota el dominio del negocio mediante un diagrama de clases y un diagrama de casos de uso. En base al mismo, se obtienen tres PSMs distintos, cada uno para una plataforma tecnológica particular: un PSM para la plataforma Java, modelado con un diagrama de clases Java; un PSM para la plataforma WSDL, con los servicios y comunicaciones existentes en el negocio; y un PSM para la plataforma *Java Web Service Developer Pack*. Este trabajo ilustra cómo pueden obtenerse múltiples PSMs a partir de un único PIM. El lenguaje *ATLAS Transformation Language* (ATL) es utilizado para definir las transformaciones M2M.

En [28] se presenta una extensión desarrollada para dar soporte al modelado de arquitecturas SOA que utilicen WSDL. En primer lugar, se desarrolla una extensión sobre UML basada en el uso de estereotipos, etiquetas y restricciones que permite modelar interfaces WSDL mediante diagramas de clase UML. Luego, se define una transformación M2T que permite generar las definiciones de cada interfaz WSDL en XML, desde dicho modelo. El desarrollo se presenta como una extensión del *framework* MIDAS-CASE. La transformación MDD es desarrollada en Java, utilizando la API *Simple API for XML* (SAX) para trabajar con XML.

El trabajo presentado en [29] desarrolla un metamodelo y una transformación M2T que permite obtener la especificación en WSDL de los servicios de una organización. El metamodelo es desarrollado con la tecnología MOF y permite especificar los servicios del negocio en base a modelos de alto nivel. La transformación M2T es desarrollada sobre el metamodelo utilizando *MOFScript*, y permite generar la especificación en WSDL de las interfaces de cada servicio. Este trabajo presenta una derivación de documentos XML a partir de un modelo de negocios basado en servicios.

Este primer grupo de trabajos comparte un mismo eje común: la aplicación de MDD a modelos de negocio para obtener modelos y artefactos de software específicos para alguna plataforma tecnológica. Algunos de los puntos en común que presentan con este trabajo de tesis son: la aplicación de MDD a modelos de alto nivel para obtener artefactos Java o definiciones de servicios web; el desarrollo de transformaciones M2T; y la aplicación de MDD a modelos BPMN. Sin embargo, todos ellos difieren en el tipo de artefactos generados y en la plataforma tecnológica que utilizan: ninguno de ellos muestra cómo obtener servicios web RESTful desde modelos BPMN.

En [30] se presenta un marco de trabajo metodológico denominado *Business Process Service Oriented Methodology* (BPSOM) que organiza el proceso de desarrollo de sistemas basados en servicios, mediante la captura de información existente en los procesos de negocio. Esta metodología especifica cómo modelar arquitecturas SOA partiendo desde modelos de procesos de negocio, que denoten la existencia de servicios o comunicaciones entre participantes. Asimismo, el trabajo define una transformación M2M que permite obtener modelos independientes de la plataforma, en base a modelos BPMN. Los modelos obtenidos se expresan en el estándar *Service Oriented Architecture*

[1] Un modelo BPMN ejecutable puede ser interpretado y ejecutado por un motor de procesos de negocio, automatizando de manera parcial o total las actividades indicadas, siguiendo el flujo de trabajo modelado ([2]).

Modeling Language (SoaML) que define la OMG en [31]. La transformación entre modelos es definida mediante el estándar QVT.

En [32] se diseña una transformación M2M que permite obtener modelos SoaML desde procesos de negocio especificados en BPMN. El trabajo comprende el diseño de las reglas de conversión entre un dominio y otro, permitiendo obtener modelos SoaML parcialmente completos, desde modelos de procesos de negocio BPMN. La transformación M2M genera la definición de cada servicio en base a las colaboraciones del modelo BPMN de entrada. Dicha transformación no es implementada en el trabajo, ya que se especifica mediante lenguaje natural.

En [33] se propone una transformación M2M que permite obtener modelos PIM desde un CIM especificado con SoaML. En primer lugar, el trabajo muestra la utilización del estándar SoaML para construir modelos de negocio orientados a servicios. Los procesos de negocio son modelados mediante una arquitectura de servicios SoaML que define el modelo general, y una coreografía de servicios SoaML, utilizada para definir el modelo específico. La arquitectura de servicios se define con un diagrama de arquitectura de servicios, y representa al proceso de negocios como un conjunto de interacciones entre participantes y servicios. Por su parte, la coreografía de servicios es definida mediante un diagrama de actividades que detalla el funcionamiento interno de cada servicio. Una vez confeccionado el modelo CIM en SoaML, el trabajo define un conjunto de reglas que permiten obtener modelos de sistema mediante diagramas de caso de uso, diagramas de estado, diagramas de clase y diagramas de paquetes que modelan las distintas vistas del software empleando UML. En última instancia, se definen reglas de transformación que permiten derivar modelos de sistema orientados a la web, desde los modelos UML. Estos modelos se representan con diagramas de componentes SoaML y diagramas de artefactos UML. Las reglas de transformación entre los modelos son presentadas en lenguaje natural, sin ser implementadas en un lenguaje de definición de transformaciones.

En [34] se construye una transformación de modelos BPMN a modelos UML empleando la metodología MDD. El trabajo define una transformación M2M que permite convertir un modelo de procesos de negocio BPMN en un modelo de comportamiento UML. Dicha conversión se lleva a cabo transformando un diagrama de procesos BPMN en un diagrama de actividades UML, según las reglas formales definidas en el trabajo. La transformación es definida e implementada utilizando el lenguaje ATL y el entorno Eclipse Juno.

En [35] se presenta una transformación de modelos BPMN a modelos UML estáticos y dinámicos, mediante la aplicación de MDD. El trabajo define un conjunto de reglas de transformación que son implementadas en el lenguaje ATL, y que permiten convertir modelos de negocio en modelos de sistema. La transformación parte de diagramas de colaboración y diagramas de proceso BPMN, y aplica un conjunto de reglas de conversión para obtener diagramas de casos de uso, diagramas de estado y diagramas de clases UML, que modelan la vista funcional, dinámica y estática del sistema, respectivamente.

En [36] se propone el diseño y desarrollo de una transformación M2M que permite obtener de manera automática modelos PIM partiendo desde modelos CIM. El trabajo propone la utilización de BPMN y UML para modelar el negocio: se utilizan diagramas de proceso BPMN y diagramas de casos de uso UML para modelar el funcionamiento y los requisitos del negocio en cuestión. Por su parte, UML es utilizado para modelar el sistema, mediante diagramas de clases y diagramas de secuencia, que modelan el aspecto estático y dinámico del mismo. Finalmente, el trabajo presenta una transformación entre el CIM y el PIM aplicando MDD. La transformación es definida mediante el estándar QVT.

Los trabajos [30, 32, 33, 34, 35, 36] comparten la aplicación de técnicas de MDD a modelos CIM para obtener modelos PIM. En la mayoría de ellos, los modelos CIM se expresan mediante la notación BPMN, utilizando diagramas de procesos o diagramas de colaboraciones. Mientras que, por otro lado, para los modelos PIM se utiliza UML o SoaML. Todos estos trabajos presentan una característica en común con este trabajo de tesis: la aplicación de técnicas de MDD a modelos BPMN, asumiendo que el negocio es soportado por el intercambio de servicios entre distintos actores. No obstante, difieren en los artefactos de software finales que logran obtener: todos ellos aplican MDD para obtener modelos más concretos desde modelos abstractos, sin generar implementaciones en código fuente u otros artefactos desplegables.

En [37] se construyen dos perfiles UML para el modelado de APIs REST y se desarrolla una transformación M2T que permite obtener la especificación de cada servicio RESTful bajo el estándar *RESTful API Modeling Language* (RAML). El trabajo lleva a cabo la construcción de un perfil UML destinado al modelado estructural de APIs REST empleando la notación UML. En adición, se diseña un segundo perfil UML que puede ser utilizado en conjunto con el anterior para añadir información adicional sobre cada servicio REST, de acuerdo al estándar RAML. El trabajo concluye con la definición de una transformación M2T que, partiendo de un modelo UML extendido, permite obtener la especificación formal (parcialmente completa) de cada servicio REST, empleando RAML. La transformación es definida utilizando el lenguaje MTL y el entorno Acceleo.

En [38] se desarrolla una herramienta basada en la web que permite generar APIs RESTful a

partir de instancias de modelos EMF (modelos *ecore*) para realizar el CRUD[2] de los elementos de dicho modelo. Partiendo de un modelo definido en EMF, la herramienta genera una implementación de API REST utilizando distintas tecnologías Java, que permiten manipular dicho modelo utilizando las representaciones XML y JSON del mismo. La API es generada de manera automática mediante la ejecución de una transformación M2T que genera el código fuente de la aplicación web. Los servicios REST generados permiten operar con el modelo, actualizando sus elementos, de manera que el modelo puede verse como un recurso REST. La transformación es definida empleando el lenguaje *Epsilon Transformation Language* (ETL) implementado para la plataforma Eclipse.

En [39] se desarrollan dos transformaciones M2T que permiten generar implementaciones en Java de APIs REST funcionales. Por un lado, se define un DSL que permite representar las relaciones entre los distintos recursos REST en un modelo. Por otra parte, se define un segundo DSL que permite representar una API REST modelando los elementos que utiliza el estándar *OpenAPI* para la especificación de servicios web RESTful. Cada DSL permite representar una API REST de una manera distinta: haciendo foco en la estructura de los recursos REST, o centrándose en la especificación técnica de la API. El trabajo concluye con el desarrollo de una transformación M2T para cada DSL que permite generar de manera automática implementaciones de APIs REST parcialmente completas en Java. Las transformaciones son definidas utilizando la tecnología *Xtend*[3].

En [41] se desarrolla una herramienta que permite generar implementaciones de APIs REST parcialmente completas, partiendo desde la especificación textual de los requerimientos. La herramienta utiliza un componente que interpreta los requerimientos y crea un PIM incompleto, modelando los principales recursos REST de la API. A partir de la intervención del usuario para completar el PIM, la herramienta genera el PSM de la API REST para la plataforma Java, utilizando un segundo componente basado en MDD. El PSM puede ser modificado por el usuario según sea necesario. Finalmente, una transformación M2T es ejecutada para obtener el código Java de la API REST. El código obtenido también puede ser refinado por el usuario según lo necesite. El trabajo utiliza ATL para el desarrollo de las transformaciones M2M y de Acceleo para la transformación M2T.

En [42] se desarrollan transformaciones M2M que permiten mapear una definición de servicios en WSDL, a la definición equivalente en REST. Se construyen tres metamodelos distintos: el metamodelo de WSDL, el de REST, y un tercer metamodelo que representa los servicios de una arquitectura SOA. A continuación, el trabajo define las transformaciones M2M WSDL a SOA, REST a SOA, SOA a REST y WSDL a REST. De esta manera, mediante un documento WSDL, las transformaciones permiten generar su equivalencia en REST y viceversa. Cada transformación es definida utilizando el estándar QVT.

En [43] se construye un perfil UML específico para el modelado de servicios web RESTful bajo el entorno *Java Spring*. El perfil introduce extensiones para modelar APIs RESTful empleando UML, y es construido utilizando la herramienta *Enterprise Architect*. Adicionalmente, se define una transformación M2T utilizando las funcionalidades que provee dicha herramienta para generar documentos textuales desde modelos. La transformación genera documentos de configuración desde modelos UML, que pueden ser utilizados posteriormente para generar aplicaciones RESTful funcionales en *Spring*, por medio de la utilidad *Spring Roo*.

En [44] se propone el desarrollo de un *Domain Specific Language* (DSL) cuyo objetivo es modelar los recursos RESTful de un dominio para luego obtener el código fuente de la API REST aplicando MDD. El trabajo muestra la construcción del DSL y el desarrollo de la transformación M2T que permite convertir una instancia del DSL en código Java funcional para implementar la API REST bajo el *framework Jersey* . La transformación es implementada utilizando las herramientas *Xtext* y *Xtend*.

Los trabajos [37, 38, 39, 41, 42, 43, 44] presentan antecedentes en la obtención de implementaciones de APIs REST, mediante la aplicación de MDD a modelos de software. En la mayoría de ellos, los modelos transformados son del tipo PSM, y la transformación requiere escasa o nula intervención humana, generando código fuente y artefactos prácticamente desplegables en un servidor. Sin embargo, todos estos trabajos inician los procesos de transformación con modelos basados en el uso de tecnologías de implementación concretas, tales como Java. En adición, la mayoría de los modelos iniciales son especificados con UML (con el uso o no de perfiles) o con algún DSL específico. Estos dos últimos puntos muestran las diferencias existentes con este trabajo de tesis, que presenta la transformación de modelos BPMN a servicios web RESTful en Java.

El trabajo presentado en [45] muestra el desarrollo de un *plugin* para Eclipse IDE que permite el modelado de arquitecturas SOA desde dicho entorno, utilizando el estándar SoaML. Se desarrolla la extensión empleando las utilidades que EMF provee para la construcción del metamodelo, la generación de la API en Java manipular los elementos de los modelos, y el desarrollo del entorno gráfico de modelado. Dicho trabajo, al igual que esta tesis, son antecedentes que ilustran el proceso de desarrollo

[2]CRUD es el acrónimo en inglés de *Create-Read-Update-Delete*, su equivalente en español es ABMC (Alta-Baja-Modificación-Consulta): las cuatro operaciones que permiten administrar un recurso persistente.

[3]Xtend es un lenguaje de programación de alto nivel que puede ser compilado para generar código Java ([40]). Su sintaxis sintética permite especificar funcionalidades complejas en pocas líneas de código. Normalmente, se lo utiliza en conjunto con *Xtext*, que es un lenguaje de especificación de DSLs. Ambas tecnologías se complementan y permiten generar clases *Java Beans* automáticamente desde un DSL.

de extensiones sobre la plataforma Eclipse.

3.2. Herramientas de modelado BPMN 2.0

El desarrollo principal de esta tesis se centra en la construcción de una extensión que permita obtener código fuente, a partir de un modelo de negocios de alto nivel, expresado mediante BPMN. Dado que la construcción de una herramienta de modelado no forma parte de los objetivos propuestos, resulta necesario realizar un estudio de las distintas herramientas de modelado BPMN disponibles, a fin de seleccionar la que más adecuada resulte para ser adoptada como el centro de los desarrollos. En esta sección, se presenta la investigación realizada al respecto, en conjunto con la herramienta seleccionada.

3.2.1. Evaluación y selección de las herramientas

El proceso de selección llevado a cabo, consistió en una serie de etapas, donde en cada una se evaluaron determinadas características y/o restricciones que debían cumplir las herramientas, con el objetivo de acotar la cantidad de resultados y facilitar la selección.

En primer lugar, se tomaron como referencia los sitios [46, 47, 48, 49, 50] y se listaron cada una de las herramientas de modelado presentadas en cada uno de ellos. Para cada una de las herramientas obtenidas, se hizo un conteo del número de veces que la misma era mencionada en las distintas fuentes de referencia, dejando en la lista aquellas que aparecían tres o más veces en total. En el anexo A6 se muestran todas las herramientas mencionadas en los sitios, y el número de veces que son referenciadas.

El resultado parcial obtenido, se filtró nuevamente, dejando únicamente las herramientas gratuitas y de código abierto disponibles. En este punto, las herramientas candidatas eran:

- *Modelio* [4]
- *Camunda* [5]
- *Bonitasoft* [6]

A continuación, se indagó sobre las características que provee cada herramienta, con el objetivo de obtener aquellas que permitan ser utilizadas en un contexto de desarrollo dirigido por modelos, y que faciliten la extensión de los modelos BPMN generados. En este punto, Modelio pareció destacar por sobre las otras dos, anticipando la selección de dicha herramienta para encarar los desarrollos. Sin embargo, se incorporó la herramienta *Eclipse BPMN2 Modeler* al listado, generando un cambio en la decisión final, optando por la adopción de esta última para la realización del trabajo.

3.2.2. Justificación de la elección

Eclipse BPMN2 Modeler no es una herramienta de modelado independiente en sí, sino que se trata de una extensión de Eclipse IDE, que posibilita la construcción de procesos de negocio mediante la notación BPMN. Esta herramienta resulta ser la más adecuada para llevar a cabo los desarrollos, ya que por su carácter de plugin, puede ser complementada con otras extensiones de Eclipse tales como Acceleo, permitiendo crear una herramienta mucho más potente. Junto con ello, las siguientes características de Eclipse BPMN2 Modeler también contribuyeron de forma positiva en su elección:

- Extensibilidad de los modelos: la herramienta proporciona un mecanismo de extensión ([51]) que permite ampliar los elementos básicos de la notación BPMN añadiendo datos y atributos, con el fin de satisfacer alguna necesidad específica de un dominio (como en el caso de este trabajo) de manera fácil y ágil, manteniendo la portabilidad de los modelos. Junto con ello, es posible también extender las reglas de validación de cada uno de los elementos del modelo generado.
- Extensibilidad de la interfaz del usuario: en conjunto con el punto anterior, el plugin también permite personalizar la interfaz del usuario integrada en la herramienta de modelado, esta característica, resulta ser vital para una implementación satisfactoria del punto anteriormente mencionado, pues al extender el modelo, dicha extensión generalmente es acompañada por una adaptación de la interfaz que facilite interactuar con el cambio introducido ([52]).
- Integración con MDD: como bien se anticipó, es posible integrar al plugin otras herramientas desarrolladas para Eclipse tales como Acceleo y EMF; facilitando la construcción de un entorno de desarrollo dirigido por modelos muy potente.

[4] https://www.modelio.org/
[5] https://camunda.com/
[6] https://www.bonitasoft.com/

- Editor de código fuente: el producto final que se obtiene en la aplicación de la herramienta propuesta son fragmentos de código fuente incompleto, comúnmente denominados *stubs*. Con lo cual es ideal proveer al usuario de la misma de un editor de código, a fin de que pueda trabajar con los *stubs* sin que deba configurar otro entorno de desarrollo aparte. Así, integrar el desarrollo en Eclipse IDE permite construir modelos de alto nivel y trabajar con el código fuente obtenido, empleando la misma herramienta.

- Herramienta gratuita, libre y de código abierto: por ser un desarrollo de la comunidad de Eclipse, la herramienta cumple con todas las restricciones impuestas por el proceso de selección anteriormente abordado.

- Comunidad amplia y activa: Eclipse BPMN2 Modeler es soportada por toda una comunidad de desarrolladores y usuarios que se encuentra en permanente actividad, mediante la creación de nuevas funcionalidades, la solución de errores o conflictos, el aporte de nuevas ideas y la construcción de ejemplos de uso.

Capítulo 4

Desarrollo del trabajo

El siguiente capítulo presenta la descripción del desarrollo del trabajo. La figura 4.1 muestra la arquitectura de componentes del marco de trabajo MDD desarrollado. Como se puede observar, la transformación BPMN2REST permite obtener código Java en base a un modelo BPMN que actúa como entrada. BPMN2REST es especificada utilizando el lenguaje Acceleo, y define la conversión M2T mapeando elementos del metamodelo de BPMN a elementos de la sintaxis de Java. El *plugin* BPMN2REST es desarrollado para la plataforma Eclipse IDE, y permite ejecutar la transformación sobre modelos BPMN, generando la implementación parcial de los servicios REST en el *framework* Spark. En adición, este componente valida el modelo de entrada, garantizando que la transformación sea ejecutada sobre modelos correctos.

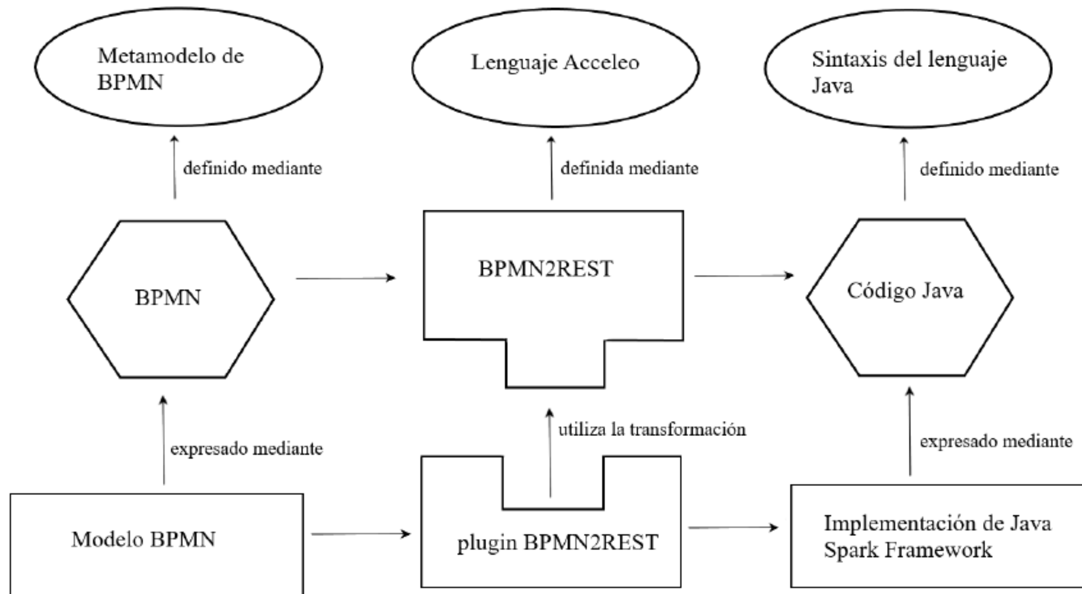


Figura 4.1: Arquitectura del marco de trabajo MDD desarrollado.

De esta manera, los temas abordados incluyen la extensión de la herramienta de modelado BPMN para dar soporte al dominio REST, el diseño de la transformación BPMN2REST, su implementación mediante las herramientas seleccionadas, el desarrollo del *plugin* sobre Eclipse IDE, y la implementación de la validación de los modelos BPMN de entrada.

4.1. Diseño de la transformación BPMN2REST

Una transformación entre modelos, ya sea M2T o M2M, es definida mediante un conjunto de reglas que especifican el mapeo entre los elementos del primer dominio, a los elementos del segundo dominio (ver sección 2.2.2). Dado que la transformación BPMN2REST permite obtener código fuente en base a un modelo BPMN, el dominio origen de la transformación estará conformado por los elementos del metamodelo de la notación BPMN 2.0; mientras que el dominio destino se conformará por elementos de la sintaxis que define el lenguaje de programación Java, en el contexto del *framework* Spark, como lo ilustra la figura 4.2.

Dado que BPMN2REST debe generar código Java en Spark, desde un modelo BPMN; es necesario definir cómo será la correlación entre elementos de BPMN y elementos del *framework* Spark para Java. En términos generales, BPMN2REST deberá:

- Identificar las invocaciones a servicios REST presentes en el modelo.
- Identificar el proveedor de cada servicio REST.
- Generar el esqueleto o *stub* de cada servicio REST proveído.
- Generar la definición de cada servicio REST utilizando la URL de acceso y el método HTTP definidos.

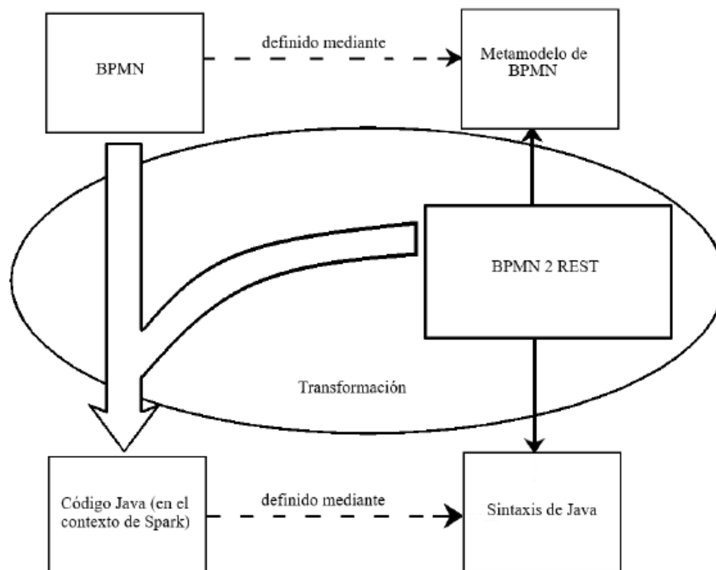


Figura 4.2: Representación gráfica de BPMN2REST.

4.1.1. Diseño de BPMN2REST: criterio de transformación

Diseñar la transformación BPMN2REST implica definir algún criterio que permita determinar cómo deben ser interpretados los modelos de entrada BPMN a fin de identificar elementos tales como los servicios involucrados, sus proveedores, el nombre de cada servicio, entre otros. Este criterio juega un rol imprescindible para definir de manera formal la transformación abordada. Durante el diseño de BPMN2REST, se confeccionaron distintos criterios para definir la transformación. Como regla general, el criterio *n + 1* resuelve algún conflicto que presenta el criterio *n*, ampliando las consideraciones que se tienen en cuenta en el mismo.

Los elementos *MessageFlow* y *Message* en BPMN permiten indicar el envío de un mensaje entre dos participantes o actores del negocio. Estos elementos resultan de gran utilidad para señalar la invocación de un servicio RESTful desde un cliente hacia un proveedor, y dan origen al primer criterio utilizado:

Criterio 1. Dado un flujo de mensaje *M* que contiene un mensaje *m*, con origen en el participante A y destino en el participante B; interpretar que el participante B provee un servicio REST hacia el participante A.

Si bien este enfoque resuelve muy bien cómo identificar cada servicio que provee un participante a los demás, el siguiente conflicto se hace presente: cada flujo de mensaje existente en el modelo BPMN de entrada indicará necesariamente una invocación a un servicio RESTful. Está claro que esto no siempre es así, pues existe la posibilidad de que no todos los envíos de mensajes entre los actores del negocio se encuentren automatizados, o que lo estén, pero empleando otro tipo de tecnologías distintas.

De esta manera, se hace necesario complementar el enfoque propuesto mediante el uso de otros elementos BPMN, o de otros criterios para evitar tal problemática, manteniendo un alto nivel de representación. En este contexto, el elemento *Interface* de BPMN complementa en gran parte al enfoque anteriormente propuesto.

Una interfaz (*Interface*) en BPMN, permite agrupar un conjunto de operaciones (*Operation*) que pueden ser proveídas por uno o varios participantes. A su vez, cada *Operation* representa un servicio que brinda un participante a los demás, y soporta de manera nativa la definición de atributos tales como parámetros de entrada/salida, tecnología de implementación, puertos, entre otros.

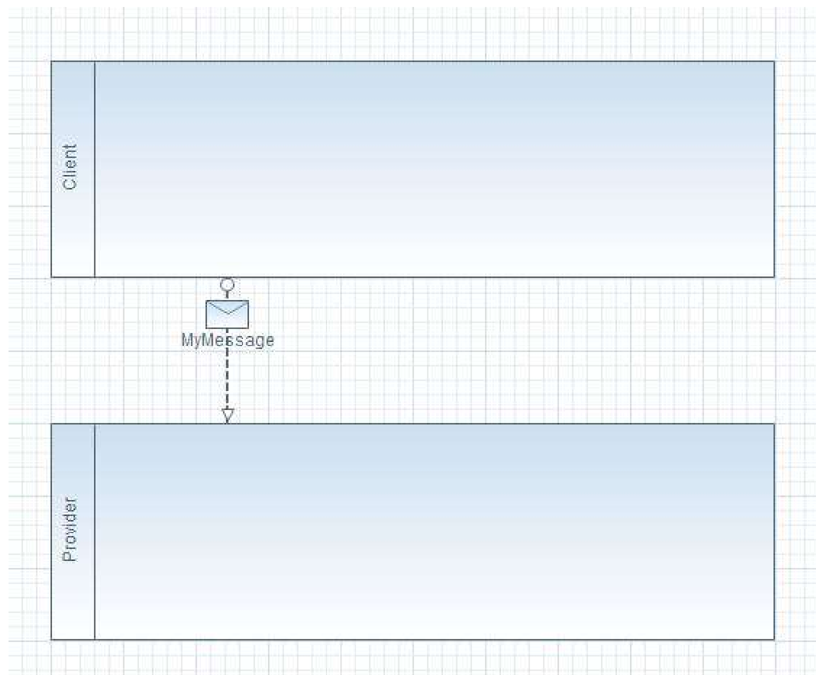
Una operación existente en una interfaz, admite la asociación de un mensaje bajo el rol de *inMessageRef*, indicando que dicho mensaje actúa como la entrada de la operación. A su vez, cada participante admite la asociación de varias interfaces (o ninguna), permitiendo definir las operaciones que brinda dicho participante hacia los demás. Haciendo uso de estas utilidades que provee de manera nativa BPMN, es posible confeccionar un criterio más robusto para identificar los servicios REST que proporcionan los participantes:

Criterio 2. Dado un flujo de mensaje *M* que contiene un mensaje *m*, con origen en el participante A y destino en el participante B, interpretar que el participante B provee un servicio REST al participante A si y sólo si B está asociado a alguna interfaz que contiene una operación cuyo mensaje de entrada sea *m*.

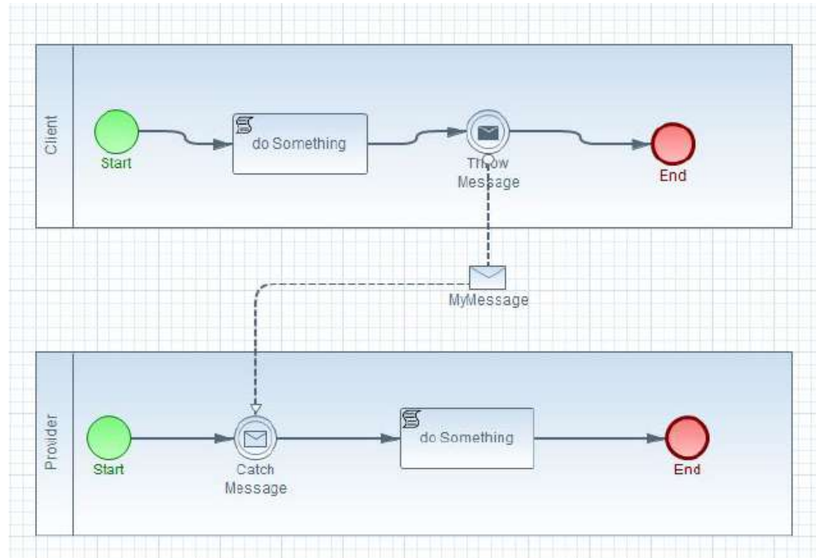
Este criterio interpreta que las operaciones definidas en las interfaces, representan servicios REST

que son proveídos por participantes. Con este enfoque, se hace una comparación entre los mensajes enviados entre participantes, y los mensajes que son la entrada de alguna operación. Así, cuando existe un *match* entre ambos, el mismo se interpreta como la invocación de un servicio. Dado que el elemento *Operation* permite que se le defina un nombre, es conveniente adoptar tal atributo como el nombre representativo del servicio REST.

El criterio 2 amplía al criterio 1 y resuelve el conflicto anteriormente mencionado, aunque consta de una gran limitación: dado que los flujos de mensajes deben ser exclusivamente entre participantes (elementos *Participant)* del modelo, este criterio no es aplicable a modelos presentados mediante una vista de *caja blanca*. Esta limitación es ilustrada de mejor manera en la figura 4.3. Allí, se comparan la vista de *caja negra* con la vista de *caja blanca* de un mismo ejemplo, y es posible observar que la aplicación de tal criterio al modelo 4.3b, arrojará que no existen servicios solicitados.



(a) Vista de caja negra de la comunicación BPMN.



(b) Vista de caja blanca de la comunicación BPMN.

Figura 4.3: Vistas de caja negra y de caja blanca de una misma comunicación BPMN. Notar la diferencia entre el nivel de detalle de ambas figuras: en el primer modelo, el MessageFlow conecta dos nodos Participant; mientras que en el segundo conecta dos nodos Intermediate Event.

En la vista de caja blanca, se detalla el proceso que ocurre dentro de cada participante, de manera que es posible modelar con mayor precisión las actividades que ocurren en un determinado proceso de negocios. En este contexto, se utilizan elementos BPMN que describen el envío o recepción de un mensaje de manera síncrona o asíncrona, junto con las actividades que se llevan a cabo en cada participante al ocurrir tal evento.

En esta vista, es común que un flujo de mensajes conecte nodos del tipo *Service Task*, *Send Task*, *Receive Task*, *Message Start Event*, *Message End Event* o *Message Intermediate Event* (ver sección 2.1.2) para indicar cómo ocurren las interacciones de manera detallada. Dado que este tipo de nodos existen en el contexto de un participante, es posible conocer qué participante actúa como el receptor del mensaje en cuestión, y así determinar el proveedor del servicio. Este razonamiento da lugar a una ampliación del criterio 2:

Criterio 3. Dado un flujo de mensaje M que contiene un mensaje m, cuyo origen es el participante A, o algún elemento ubicado dentro del participante A, y destino es el participante B, o algún elemento ubicado dentro del participante B; interpretar que el participante B provee un servicio REST al participante A si y sólo si B está asociado a alguna interfaz que contiene una operación cuyo mensaje de entrada sea m.

El nuevo criterio confeccionado, permite que BPMN2REST pueda identificar quién es el proveedor de un servicio REST, independientemente de si se trata de una vista de caja blanca o de caja negra.

Si bien el criterio 3 resuelve de manera satisfactoria los conflictos presentados y constituye un buen punto de partida para definir la transformación BPMN2REST, todavía no resulta suficiente para que pueda ser especificada de manera formal. Ello se debe a que la naturaleza abstracta de BPMN, no permite definir dentro de la misma notación, información de vital importancia para BPMN2REST. Este problema surge debido a que tal información, resulta dependiente de la tecnología de implementación utilizada (REST, en este caso).

4.1.2. Extensión de BPMN para dar soporte a BPMN2REST

En general, un modelo de procesos de negocios presenta información con un alto nivel de abstracción, representando las cualidades más importantes de los negocios involucrados, y tomando distancia de las tecnologías, arquitecturas o herramientas computacionales que se utilicen a lo largo de las distintas actividades. De esta manera, es válido afirmar que un modelo de este tipo puede ser clasificado como CIM, como es el caso particular de los modelos BPMN.

La notación BPMN permite especificar y detallar satisfactoriamente los procesos de negocio de una organización. En dicho aspecto, BPMN cumple con sus objetivos de diseño completamente. No obstante, la notación no proporciona mecanismos directos que permitan detallar las tecnologías de implementación involucradas en los procesos. Esto constituye un obstáculo en la especificación de los servicios RESTful involucrados en las comunicaciones de un negocio; ya que, según lo planteado en la sección anterior, BPMN2REST deberá ser precisa hasta el punto de codificar las implementaciones de los servicios REST en código Java empleando Spark.

En tal caso, y, de acuerdo con la definición de los servicios REST conforme al *framework* Spark (ver sección 2.5.3), existen dos datos de gran importancia para cada servicio que deben ser definidos de manera precisa en dicho entorno:

- La URL o *path* relativa del servicio.
- El método HTTP con el que se accede al servicio.

De acuerdo a la sección anterior, cada operación en una interfaz implementada por un participante, es interpretada por BPMN2REST como un servicio REST que provee dicho participante a los demás. Ahora bien, para que BPMN2REST genere el código Java de cada servicio detectado al aplicar el criterio de la transformación, deberá obtener de alguna manera la URL y el método HTTP de acceso al mismo. Es necesario que esta información, por lo tanto, exista en el modelo BPMN de entrada.

Si bien el elemento *Operation* admite la definición de algunos atributos que podrían ser utilizados para definir los datos anteriores (mensaje/s de entrada y/o salida y nombre de la operación, por ejemplo), los mismos forman parte del *core* de la notación BPMN y no fueron diseñados para ser utilizados con dicho fin; pues en tal caso, su uso podría conducir a la confección de archivos *.bpmn* corruptos; los modelos podrían dejar de ser portables; o la definición formal de BPMN2REST podría tornarse compleja.

En base a lo anterior, resulta necesario el desarrollo de una extensión sobre BPMN que permita introducir información específica del dominio REST a los modelos. Dicha información, le permitirá a BPMN2REST determinar la URL y el método de acceso HTTP para cada servicio, de manera precisa y directa.

Así, se define la extensión *extBPMN* (*extended* BPMN) como el super-conjunto de BPMN compuesto por todos los elementos de BPMN nativos, adicionando los siguientes atributos al elemento *Operation*:

- *Operation.rest_method*.
- *Operation.rest_url*.
- *Operation.rest_prepend_controller*.

La extensión extBPMN añade tres atributos que permitirán a BPMN2REST identificar los datos necesarios para la definición de cada servicio. Estos atributos se añaden al elemento *Operation* utilizado para representar un servicio proveído por un participante. El atributo *rest_method* contiene una cadena de caracteres que identifica al método HTTP utilizado para invocar al servicio. Los valores que admite son "GET", "POST", "PUT", "DELETE", "PATCH" y "OPTIONS" (y sus

equivalentes en minúscula). Por su parte, el atributo *rest_url* contiene una cadena de caracteres que identifica la URL de acceso al servicio. Esta URL deberá ser relativa al host de alojamiento, respetando el formato que acepta Spark para la definición de la ruta de invocación de los servicios REST (por ejemplo, "esta/es/mi/url" podría ser un valor válido en este atributo).

Dada la especificación de una interfaz con sus operaciones en un modelo BPMN, es probable que existan varios participantes actuando como proveedores de los mismos servicios, definidos en dicha interfaz. En este caso, es posible reutilizar este elemento BPMN, asociando a más de un proveedor la misma interfaz (y por lo tanto, los mismos servicios). Ahora bien, si se define una combinación para los valores *rest_url* y *rest_method* sobre una operación, la configuración resultante será la misma para todos los participantes que provean dicha operación (a través de la misma interfaz). En tal caso, BPMN2REST codificará el *binding* de cada servicio REST, sin tener en cuenta que puedan existir varias definiciones de servicios asociadas a la misma URL de acceso y al mismo método HTTP. A fin de evitar este inconveniente sin quitar la posibilidad de reutilización de las interfaces BPMN, se añade el atributo *rest_prepend_controller* a las operaciones. Este campo tendrá un valor booleano, que al ser activado (valor *true*), le indicará a BPMN2REST que debe incorporar el nombre del participante a la URL del servicio REST, generando un *binding* único para cada servicio. La tabla 4.1 ilustra la interpretación que asigna BPMN2REST a los atributos introducidos, en base a cuatro ejemplos prácticos.

Nombre del elemento Operation	**rest method**	**rest url**	**rest prepend controller**	**Nombre del participante proveedor del servicio**	**Interpretación de BPMN2REST**
servicioA	GET	/servicioA	false	proveedor	Generar el servicio "servicioA" disponible en la URL /servicioA, mediante HTTP GET
servicioB	GET	/servicioB	true	proveedor	Generar el servicio "servicioB" disponible en la URL /proveedor/servicioB, mediante HTTP GET
servicioB	GET	/servicioB	true	afip	Generar el servicio "servicioB" disponible en la URL /afip/servicioB, mediante HTTP GET
servicioC	POST	/servicioC	true	jhon_doe	Generar el servicio "servicioC" disponible en la URL /jhon_doe/servicioC, mediante HTTP POST

Tabla 4.1: Ejemplos de interpretación que asigna BPMN2REST a los atributos extBPMN.

Si bien extBPMN se constituye como una versión enriquecida de BPMN, donde los cambios introducidos resultan muy leves y prácticamente imperceptibles para el *core* de la notación, debe notarse que el dominio origen de la transformación BPMN2REST será ahora un modelo extBPMN y no BPMN, como lo muestra la figura 4.4. Cabe mencionar que extBPMN se estructura como un super-conjunto de BPMN, es decir, una ampliación de la notación original, como lo muestra la figura 4.5.

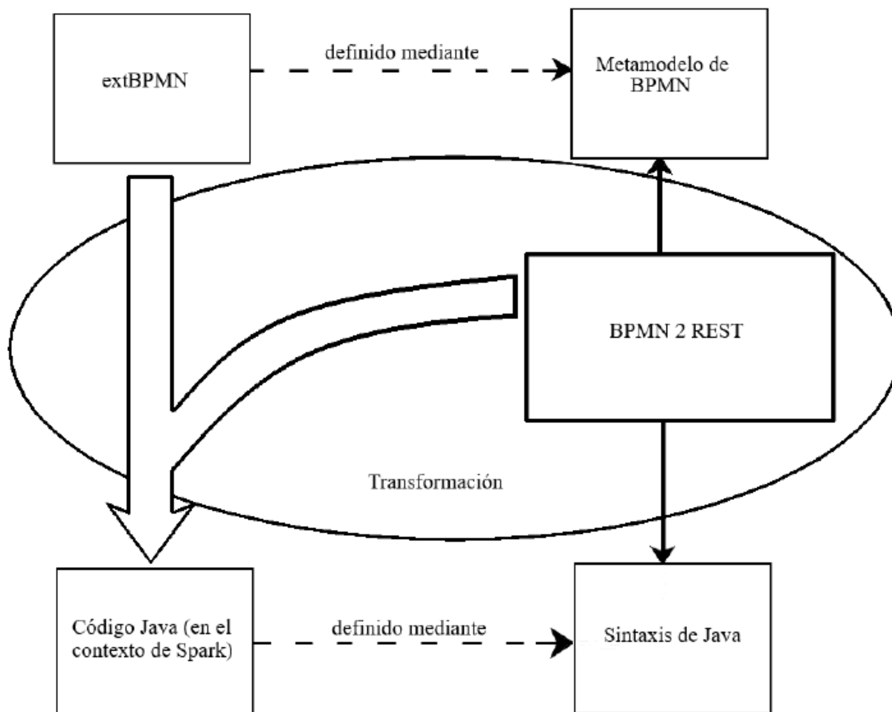


Figura 4.4: Representación gráfica de BPMN2REST en base a la extensión introducida.

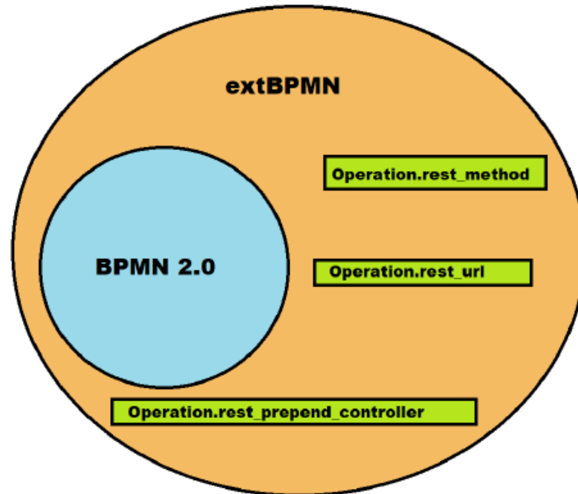


Figura 4.5: Representación gráfica de la composición de extBPMN.

4.1.3. Definición formal de BPMN2REST

La transformación BPMN2REST genera código Java a partir de un modelo BPMN (o extBPMN, según lo dicho anteriormente). Éste código Java se contextualiza en el *framework* para el desarrollo de aplicaciones basadas en microservicios Spark. Es así, que el código fuente obtenido a partir de la aplicación de BPMN2REST a un modelo, deberá contener el *binding* de cada servicio junto con la implementación de cada uno de ellos.

De acuerdo a la estructura que define Spark para la definición de cada servicio que compone una aplicación web basada en Java (ver sección 2.5.3); está claro que deben especificarse tres elementos para cada uno de ellos:

1. Método HTTP por el cual se accede al servicio.
2. URL a través de la cual se solicita el servicio.
3. *Callback* que provee la implementación del servicio.

Spark interpreta estos tres elementos y genera un servicio REST que al ser accedido mediante el método HTTP indicado, a través de la URL definida, devuelve el resultado de ejecutar el *callback* utilizando los datos de la solicitud o *request* como los parámetros de entrada.

La extensión extBPMN brinda el soporte necesario para que BPMN2REST determine con facilidad los datos 1 y 2 para cada servicio REST. En el caso del tercer dato, Spark permite indicar la implementación del correspondiente servicio a partir de una función anónima en Java. De esta manera, es posible hacer uso de esta utilidad para delegar la implementación del servicio a una clase Java.

Es posible que existan participantes asociados a varias operaciones (a través de interfaces BPMN), que no *matcheen* a ningún *MessageFlow* con su mensaje de entrada. De acuerdo con el criterio 3 definido anteriormente (ver sección 4.1.1), BPMN2REST sólo contemplará aquellas operaciones cuyo mensaje de entrada sea referenciado por un *MessageFlow*, interpretando que ese envío de mensaje señala la invocación de un servicio REST. En tal caso, las operaciones de un participante que no hagan *match*, con algún *MessageFlow*, serán ignoradas. De esta manera, el término "proveedor activo" hará referencia a los participantes BPMN que proveen al menos un servicio de acuerdo al criterio 3.

Cada clase Java responsable de implementar los servicios REST, será generada a partir de un participante que actúe como el proveedor activo de uno o más servicios. Para cada uno de ellos, BPMN2REST generará una clase con el nombre *<Participant.name>Controller.java* (es decir, el nombre de la clase será generada concatenando el sufijo "Controller" al nombre del participante BPMN que provee los servicios). Todos los controladores serán agrupados en una carpeta *controllers*.

Cada servicio REST identificado por la transformación, será proveído por uno o más participantes. Cada *match* entre participante, servicio y flujo de mensaje, generará un código esqueleto con la implementación (vacía) del método Java en la clase controlador correspondiente. Este método brindará la implementación del servicio. El método recibirá dos parámetros: *req* (del tipo spark.Request) y *resp* (del tipo spark.Response), y su nombre se conformará en base a la versión *lower camel case* del atributo *Operation.name*.

Por último, el *binding* para cada servicio será definido en el método *main* de la clase *Main.java*. Allí, se definirá la relación entre el método HTTP, la URL y el *callback* de acuerdo al entorno Spark. En base a ello, es claro que sólo debe generarse un archivo *Main.java* por cada ejecución de BPMN2REST. Este archivo estará en la misma ubicación que la carpeta *controllers*.

La tabla 4.2 muestra la definición formal de BPMN2REST, indicando el mapeo entre elementos del dominio origen (extBPMN) y destino (código Java en Spark). Se muestra la especificación formal en OCL de las consultas para los elementos del metamodelo BPMN, su especificación en lenguaje natural, y los correspondientes elementos textuales a ser generados. Los elementos del modelo de entrada que escapen a los selectores, no serán tenidos en cuenta en dicho proceso. La figura 4.6 muestra el mapeo uno a uno realizado por BPMN2REST: cada elemento de la izquierda obtenido por los selectores, genera los elementos indicados a la derecha. Así, por ejemplo, desde un proceso de negocios (representado por *Collaboration*) se genera una clase Main.java y un directorio “controllers”.

<table>
<tr><th>Especificación formal del selector (OCL)</th><th>Especificación en lenguaje natural del selector</th><th>Significado del selector</th><th>Elemento/s del dominio destino generados</th></tr>
<tr><td>Collaboration.allInstances->first()</td><td>Primer instancia de colaboración definida en el modelo.</td><td>Colaboración (representa el modelo de entrada).</td><td>1) Archivo Main.java.

2) Código esqueleto de la clase Main.

3) Método main(String[] args) en la clase Main con el cuerpo vacío.</td></tr>
<tr><td>Participant.allInstances->select(p | p.interfaceRefs->select(i |
i.operations->select(o |
MessageFlow.allInstances->exists(mf |
mf.messageRef == o.inMessageRef
)
)->notEmpty()
)->notEmpty()
)</td><td>Participantes asociados al menos a una interfaz tal que la misma contiene al menos una operación cuyo mensaje de entrada es referenciado por un flujo de mensajes.</td><td>Participantes que proveen algún servicio según el criterio 3.</td><td>1) Archivo <p.name.toUpperCamelCase()>Controller.java dentro del directorio /controllers.

2) Código de la clase <p.name.toUpperCamelCase()>Controller (esqueleto).</td></tr>
<tr><td>Participant.allInstances->collect(p | p.interfaceRefs->collect(i |
i.operations->select(o |
MessageFlow.allInstances->select(mf |
mf.messageRef == o.inMessageRef
)->select(mf | mf.targetRef == p ||
p.processRef.artifacts->includes(mf.targetRef)
)->count() == 1
)
)->flatten()
)->flatten()</td><td>Operaciones asociadas al menos a una interfaz implementada por al menos un participante, cuyo mensaje de entrada es referenciado por un flujo de mensajes entrante al participante que las provee.</td><td>Operaciones que son proveídas por al menos un participante, representando un servicio, de acuerdo al criterio 3.</td><td>1) Para cada participante proveedor del servicio:

a) Generar el esqueleto del método <o.name.toLowerCamelCase()>(Request req, Response, res) en la clase controlador del participante.

b) Generar la definición del binding en el método Main.main utilizando los valores o.rest_method y o.rest_url. Bindear el servicio al método generado en la clase controlador.</td></tr>
</table>

Tabla 4.2: Definición formal de BPMN2REST.

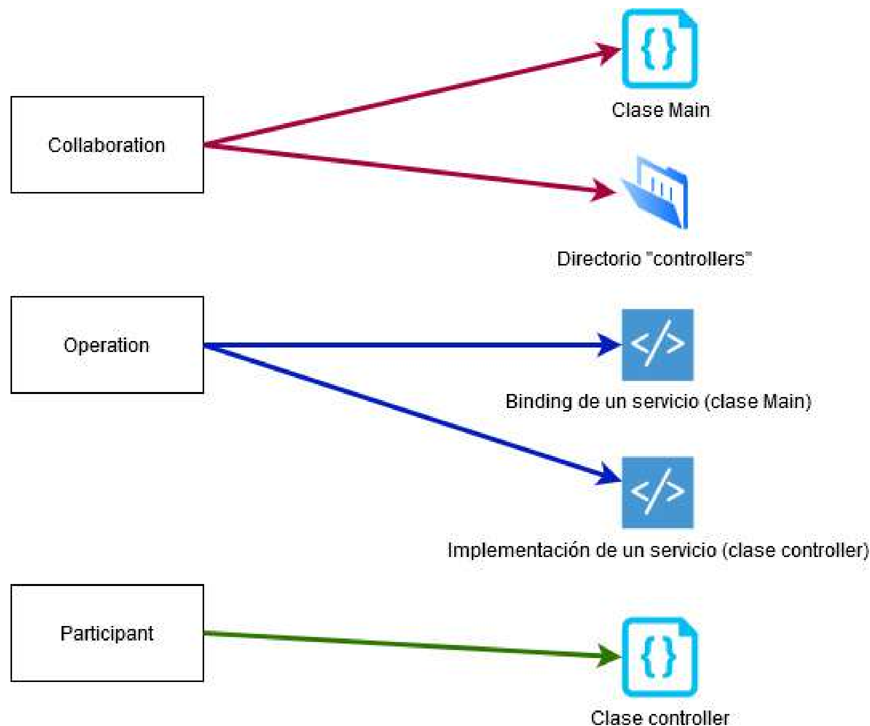


Figura 4.6: Representación gráfica del mapeo realizado por BPMN2REST. Cada elemento de la izquierda genera una combinación de los elementos de la derecha.

4.2. Implementación de la transformación BPMN2REST

Partiendo de la definición formal de BPMN2REST y de extBPMN, es posible en este punto construir una implementación para ambos diseños haciendo uso de plataformas específicas de desarrollo. Como se anticipó anteriormente, en el presente trabajo se utiliza el entorno de desarrollo provisto por Eclipse para implementar la solución propuesta. De esta manera, en la siguiente sección se abordará la definición de extBPMN empleando el *plugin* Eclipse BPMN2 Modeler, y la especificación de BPMN2REST a partir del uso del *plugin* Acceleo para Eclipse.

4.2.1. Extensión de BPMN2 Modeler para la incorporación de extBPMN

Eclipse BPMN2 Modeler ([21]) es un *plugin* desarrollado para el ecosistema de desarrollo Eclipse, que brinda soporte al modelado de procesos de negocio bajo la notación BPMN 2.0. A partir del uso de este *plugin*, es posible modelar procesos de negocio de manera fácil e intuitiva, utilizando para ello la interfaz gráfica que provee el mismo.

Eclipse BPMN2 Modeler puede ser extendido a fin de dar soporte a distintas necesidades de dominios más específicos, o de incorporar nuevas funcionalidades al mismo. El *plugin* define una serie de puntos de extensión ([53]) en base a los cuales, es posible añadir código Java que es ejecutado ante determinados eventos o situaciones, o definir características y comportamientos adicionales sobre la notación (por ejemplo, un atributo nuevo en un elemento BPMN). Una extensión de BPMN2 Modeler, debe ser encapsulada en un *plugin* para Eclipse, cuya definición será dada por su documento *plugin.xml*. De esta manera, extBPMN será implementada a través de un *plugin* que extienda a Eclipse BPMN2 Modeler.

Implementar la extensión extBPMN básicamente implica extender el elemento BPMN *Operation* añadiéndole tres atributos personalizados. Resulta necesario entonces utilizar algún mecanismo que permita extender los elementos BPMN soportados por defecto por BPMN2 Modeler. El punto de extensión *org.eclipse.bpmn2.modeler.runtime* (ver anexo A7) es definido por BPMN2 Modeler y puede ser incorporado en un *plugin* para añadir nuevas características al entorno gráfico de la herramienta y al modelo básico de la notación BPMN empleado por la misma ([54]).

El punto de extensión debe ser añadido al documento *plugin.xml*, junto con los datos que resultan necesarios para la correcta interpretación de los atributos por parte del motor de BPMN2 Modeler. Dentro del punto de extensión, el elemento *modelExtension* puede ser utilizado para añadir propiedades a los elementos de la notación estándar BPMN. La figura 4.7 muestra el fragmento XML utilizado para incorporar la extensión del elemento *Operation* sobre BPMN2 Modeler. En el mismo, se especifica el elemento BPMN a ser extendido, los atributos en cuestión y el tipo de dato adoptado por cada uno de ellos.

```
<modelExtension
      description="BPMN2REST Operation extension for supporting REST features"
      id="org.eclipse.contributions.bpmn2rest.modeler.extensions.Operation"
      name="BPMN2REST Operation Extension"
      runtimeId="org.eclipse.contributions.bpmn2rest.modeler.runtime"
      type="Operation">
   <property
         description="URL or path to the endpoint of this operation"
         label="REST Service URL"
         name="rest_url">
   </property>
   <property
         description="HTTP method to use on this RESTful service"
         label="REST Service HTTP Method"
         name="rest_method"
         value="GET">
   </property>
   <property
         description="Prepend the controller&apos;s name to the URL. It avoids hav
         label="Add Controller Preffix to URL"
         name="rest_prepend_controller"
         type="EBoolean"
         value="false">
   </property>
</modelExtension>
```

Figura 4.7: Fragmento XML que define la extensión del elemento Operation según el punto de extensión org.eclipse.bpmn2.modeler.runtime.

La incorporación del punto de extensión anteriormente abordado, permite dar soporte a extBPMN a través del *plugin* BPMN2 Modeler. Como consecuencia directa, un modelo BPMN generado por esta extensión, contendrá elementos *Operation* con los tres atributos definidos, aunque el valor de cada uno de ellos será vacío. Resulta de gran interés que el valor de los atributos añadidos pueda ser definido por el usuario, haciendo uso de la interfaz gráfica de configuración de los elementos BPMN que provee BPMN2 Modeler; tal como sucede para los demás atributos estándares de cada elemento de la notación (ver figura 4.8). Así, es necesario dar soporte también en este nivel a la extensión desarrollada.

Por defecto, BPMN2 Modeler permite definir los valores de los atributos para los distintos elementos BPMN a través de interfaces gráficas de configuración. Dentro del punto de extensión utilizado anteriormente, el elemento *propertyTab* permite añadir una pestaña en la ventana de configuración de un elemento BPMN, para la asignación de valores a determinados atributos de dicho elemento; es decir, permite extender la interfaz de configuración por defecto utilizada por los elementos BPMN ([52]). De esta manera, la figura 4.9 muestra el fragmento XML que define el elemento *propertyTab* utilizado para agregar una nueva pestaña en la ventana de configuración del elemento *Operation*. La pestaña nueva, denominada "BPMN2REST Extension", permitirá definir los valores para los tres atributos añadidos anteriormente desde la perspectiva del usuario que crea el modelo BPMN, como muestra la figura 4.10.

Cualquier extensión sobre BPMN2 Modeler debe ser encapsulada en un proyecto del tipo *plugin* para Eclipse que incorpore el punto de extensión *org.eclipse.bpmn2.modeler.runtime*. Tanto la extensión añadida sobre el elemento *Operation* como la nueva pestaña de configuración, deben ser definidas a partir de un fragmento XML embebido en el documento *plugin.xml*. En otras palabras, el desarrollo de la extensión sobre BPMN2 Modeler para implementar extBPMN, es encapsulado en su propio *plugin* para Eclipse. Para la incorporación del punto de extensión, así como la definición del

archivo *plugin.xml* y el manejo de las dependencias, se utilizó el entorno provisto por Eclipse PDE.

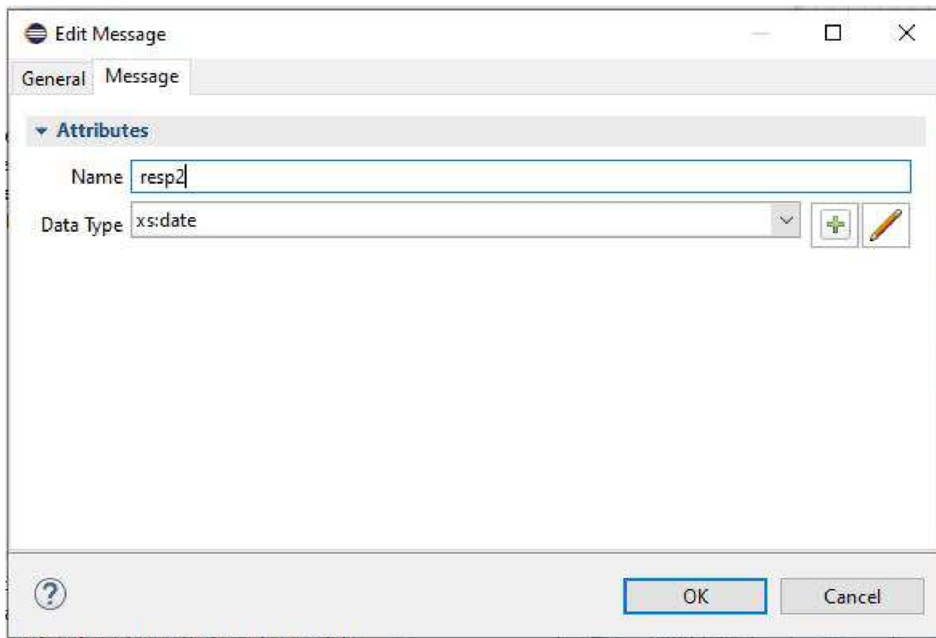


Figura 4.8: Ventana de configuración del elemento Message. En la misma se pueden modificar los atributos nombre y tipo de dato para el elemento. Cada elemento de la notación tiene una ventana de configuración propia.

```
<propertyTab
      class="default"
      features="rest_url rest_method rest_prepend_controller"
      id="org.eclipse.contributions.bpmn2rest.modeler.propertyTabs.Operation"
      label="BPMN2REST Extension"
      popup="true"
      runtimeId="org.eclipse.contributions.bpmn2rest.modeler.runtime"
      type="Operation">
</propertyTab>
```

Figura 4.9: Fragmento XML que define los valores para el elemento propertyTab.

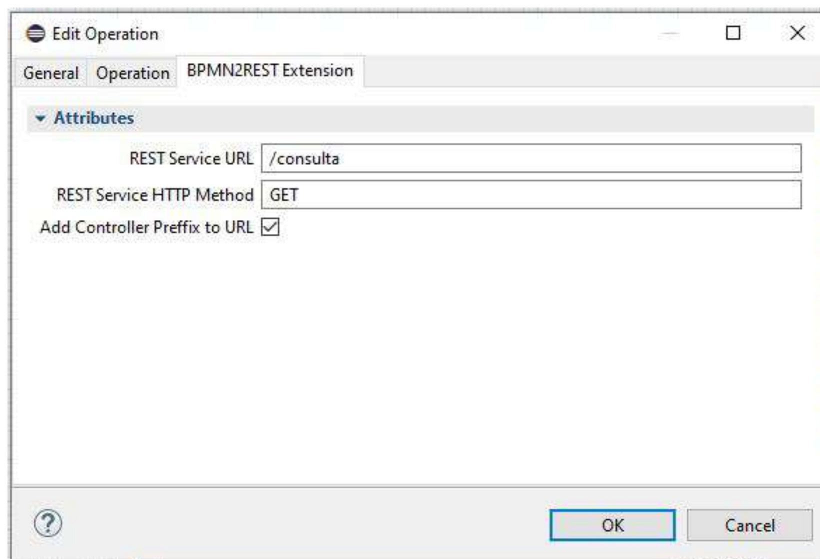


Figura 4.10: Ventana de configuración del elemento Operation luego de la extensión añadida.

Eclipse BPMN2 Modeler permite la extensión de sus funcionalidades a través de la construcción de *plugins* que incorporen el punto de extensión *org.eclipse.bpmn2.modeler.runtime*. Una vez desarrollados, estos *plugins* pueden ser instalados sobre Eclipse IDE y reconocidos por BPMN2 Modeler como extensiones funcionales del mismo ([54]). Una extensión podrá ser seleccionada para ser utilizada en un proyecto creado en Eclipse, esto activará las características añadidas por la misma, dentro del proyecto. Sin embargo, sólo una extensión de BPMN2 Modeler podrá ser habilitada para trabajar en un proyecto simultáneamente. La extensión a utilizar puede seleccionarse desde la ventana de configuración de un proyecto, bajo la pestaña “BPMN2”, como se muestra en la figura 4.11.

Cada extensión de BPMN2 Modeler debe definir su propio *namespace*. Este valor es utilizado como el *namespace* principal para cada modelo BPMN creado en un proyecto que utilice dicha extensión.

De esta manera, BPMN2 Modeler puede distinguir entre los modelos creados con una extensión del *plugin* y los modelos creados con otra distinta. Aun así, el desarrollador de las extensiones debe explicitar un mecanismo de evaluación que permita saber si un modelo BPMN fue desarrollado o no con una extensión determinada. Esto se hace incorporando el elemento *runtime* al punto de extensión *org.eclipse.bpmn2.modeler.runtime*.

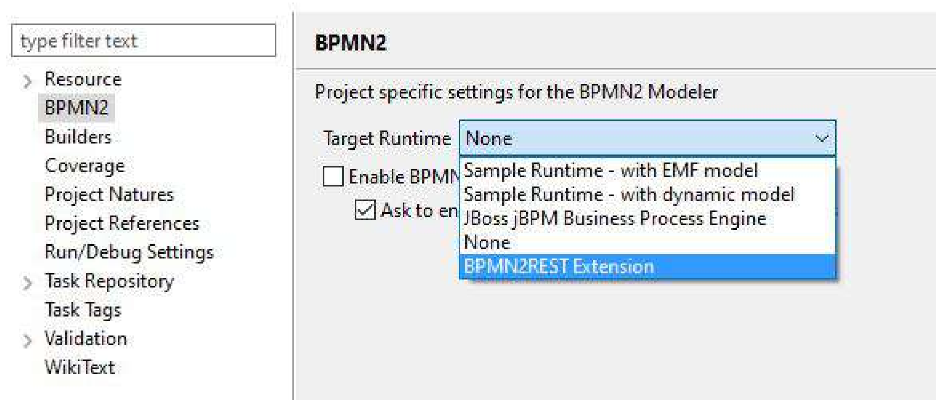


Figura 4.11: Selección de una extensión de BPMN2 Modeler para ser aplicada sobre un proyecto en Eclipse. Pueden existir varias extensiones en simultáneo, pero sólo una puede ser utilizada en un proyecto a la vez.

El elemento *runtime* debe existir de forma mandatoria en un *plugin* que extienda a BPMN2 Modeler. En el mismo, se indica un ID que identifica a la extensión, y una clase Java que implemente la interfaz *org.eclipse.bpmn2.modeler.core.IBpmn2RuntimeExtension*. Dicha interfaz, impone la definición de los métodos *getTargetNamespace* y *isContentForRuntime*. El primero debe retornar un *String* con el valor a ser utilizado en el *namespace* de los modelos, mientras que el segundo, recibe como parámetro el archivo *.bpmn* que representa un modelo, y debe devolver *true* si dicho modelo fue generado por la extensión en cuestión, o *false* si no fue así.

La figura 4.12 muestra el fragmento que define el elemento *runtime* requerido por el punto de extensión. En el mismo, se indica el ID *org.eclipse.contributions.bpmn2rest.modeler.runtime*, y la clase *org.eclipse.contributions.bpmn2rest.modeler.Bpmn2RestRuntimeExtension* como la implementación de la interfaz *IBpmn2RuntimeExtension*.

```
<runtime
      class="org.eclipse.contributions.bpmn2rest.modeler.Bpmn2RestRuntimeExtension"
      description="BPMN2REST BPMN2 Modeler Plugin Extension"
      id="org.eclipse.contributions.bpmn2rest.modeler.runtime"
      name="BPMN2REST Extension">
</runtime>
```

Figura 4.12: Fragmento XML que define los valores para el elemento runtime en el contexto del punto de extensión org.eclipse.bpmn2.modeler.runtime.

4.2.2. Especificación de BPMN2REST mediante Acceleo

Acceleo ([23]) es un *plugin* para Eclipse IDE que define un entorno de desarrollo, encuadrado en el marco de MDD, que permite implementar transformaciones del tipo M2T. Acceleo brinda un conjunto de herramientas que da soporte a la creación de transformaciones bajo el lenguaje de definición MTL, a la vez que asiste a los desarrolladores en dicha tarea.

La definición formal de BPMN2REST abordada anteriormente, especifica cómo es el mapeo entre el dominio extBPMN y el código Java a ser generado. Dado que este código se encuadra bajo el *framework* Spark para Java, cada servicio REST debe ser definido siguiendo las prácticas que impone dicho *framework*. Así, la siguiente estructura de directorios y archivos, define el resultado de una ejecución de BPMN2REST tomando como entrada un modelo extBPMN:

- Una clase Java “Main”, que define las distintas rutas de la API REST y el *callback* de cada una.
- Un directorio “controllers” que alojará las clases del tipo controlador. Este directorio existirá si existe al menos una clase controlador que implemente un servicio REST.
- Una o más clases del tipo controlador que tendrán la implementación de cada servicio REST provisto por un participante.

De acuerdo al entorno que provee Acceleo, una transformación se define en base a una serie de módulos o archivos *.mtl*, que contienen directivas, plantillas e instrucciones que son interpretadas por el motor de la herramienta (ver sección 2.5.6). Según la definición formal de BPMN2REST y, conforme a la estructura que impone el *framework* Spark para definir microservicios en Java, la transformación M2T es definida en Acceleo a partir de los siguientes módulos:

- *main.mtl:* es el módulo principal. Define el orden de invocación de los otros módulos y constituye el punto de inicio de la transformación.
- *queries.mtl:* contiene la definición de las distintas consultas utilizadas y las funciones de usuario definidas para construir la transformación.
- *generateMainClass.mtl:* define cómo es el mapeo de los elementos de extBPMN hacia el código Java de la clase "Main.java".
- *generateControllerFiles.mtl:* define el mapeo de los elementos de extBPMN hacia el código Java de una clase controlador.

Consultas incorporadas a la transformación

En términos generales, al definir una transformación del tipo M2T es necesario hacer explícito el mapeo entre los elementos del dominio de origen y los del dominio de salida. Dicho mapeo debe describirse en base al uso de un lenguaje de definición de transformaciones (en este caso, MTL); definiendo la correspondencia entre los elementos del modelo de entrada y la estructura textual de los documentos que deben generarse (ver sección 2.2.2). En este contexto, el lenguaje debe proveer al usuario de algún mecanismo que le permita definir consultas sobre el modelo sobre el cual se ejecuta la transformación. Cabe destacar que estas consultas deben ser especificadas sobre los elementos del metamodelo asociado al modelo de entrada, permitiendo que las mismas sean aplicables a distintas instancias de dicho metamodelo (es decir, otros modelos de entrada del mismo tipo).

El lenguaje de definición de transformaciones que utiliza Acceleo basa fuertemente su sintaxis en el estándar OCL ([15]) para la definición de sus consultas. Acceleo pone a disposición del usuario una librería de funciones OCL entre las que se pueden encontrar consultas nativas soportadas por el estándar, y consultas adicionales definidas por el propio *plugin* con el objetivo de extender el soporte brindado al usuario (ver librerías en [25] y [26]). Acceleo permite la definición de nuevas consultas definidas por el usuario, a fines de extender el soporte nativo. Las mismas pueden ser especificadas a través del uso de OCL, o mediante la especificación en código Java de las mismas, empleando una clase servicio para ello (ver sección 2.5.6).

Durante el desarrollo de la transformación fue necesario incorporar consultas definidas por el usuario, a fin de evitar la repetición de código (tanto OCL como Java). Algunas consultas fueron implementadas haciendo uso del estándar OCL, mientras que otras se definieron mediante código Java, aprovechando el uso de las utilidades nativas que provee el lenguaje de programación (por ejemplo, para el tratamiento de cadenas de texto). La tabla 4.3 muestra las consultas incorporadas a la transformación, cuál fue su propósito, y en qué tecnología fue desarrollada. Las consultas implementadas en Java fueron construidas a partir de la invocación de clases servicio que encapsulan el código fuente ejecutado. En el anexo A8 se muestran las distintas clases servicio incorporadas a la transformación, y las consultas que las utilizan. La librería de consultas de usuario confeccionada fue incorporada en el archivo *queries.mtl* y es incluída en los módulos *generateMainClass.mtl* y *generateControllerFiles.mtl*.

Signatura de la consulta	**Descripción**	**Tecnología de implementación**
Set<Participant>getServiceProviders(Definitions d)	Devuelve los participantes de la colaboración contenida en d que proveen al menos un servicio a los demás.	OCL
Set<Operation>getProvidedOperations(Participant p)	Devuelve los servicios que provee el participante p a través de los elementos *Operation* que los representan.	OCL
String getServiceHttpMethod(Operation o)	Devuelve el método HTTP (en minúsculas) definido en la propiedad *o.rest_method*. Si ningún método es definido, devuelve "get".	Java
String getServiceUrl(Operation o, Participant p)	Devuelve la URL de invocación del servicio o, que provee el participante p. La URL se tomará del atributo *o.rest_url*, utilizando el atributo *p.name* como prefijo a la misma, de acuerdo al valor de *o.rest_prepend_controller*.	Java
String toUpperCamelCase(String s)	Devuelve la versión *upperCamelCase* de una cadena de caracteres.	Java
String toJavaMethod(String s)	Limpia la cadena s, devolviendo una versión sin los caracteres que no puedan ser utilizados como el nombre de un método Java. Remueve los espacios en blanco, los caracteres "/" y "\", y agrega un caracter "_" en el caso de que s comience con un dígito numérico.	Java

Tabla 4.3: Descripción de las consultas y funciones de usuario incorporadas manualmente a la librería nativa de Acceleo.

Mapeo de extBPMN a Main.java

En base a la estructura definida por el *framework* Spark (ver sección 2.5.3), la clase Main debe contener la definición de las rutas que componen la aplicación basada en microservicios en Java. Cada ruta se compone de tres elementos fundamentales:

- El método HTTP con el cual se accede al servicio.
- La URL relativa al servidor para acceder al servicio.
- El *callback* que provee la implementación del servicio.

El *callback* que brinda la implementación del servicio, es especificado generalmente a través de una expresión *lambda* en Java. Esta expresión contiene la especificación del servicio web RESTful, e

indica las acciones que se realizarán con los parámetros de la solicitud HTTP, y la construcción de la respuesta hacia dicha solicitud. Para ello, el *callback* debe recibir un parámetro del tipo *spark.Request* y otro del tipo *spark.Response*, que serán inyectados por Spark al momento de requerir la ejecución del servicio.

Definir la implementación de cada servicio en la clase Main, supone un obstáculo al momento de mantener el código fuente legible y entendible: si los servicios definidos son complejos y contienen muchas líneas de código, el mantenimiento de los mismos puede tornarse una tarea dificultosa y la clase Main puede volverse muy grande. Además, resulta de interés poder aislar determinados servicios de otros a fin de encapsular el funcionamiento de los mismos, o agruparlos bajo determinados criterios ocultando su visibilidad entre los distintos grupos de desarrolladores. Esto último es útil, por ejemplo, si se quiere que sólo un grupo de desarrolladores esté a cargo de un determinado conjunto de servicios. En dicho caso, los demás desarrolladores podrán utilizar estos servicios sin conocer cómo están implementados (es decir, tendrán una vista de caja negra de los mismos).

De acuerdo a lo dicho anteriormente, puede especificarse el *callback* o la implementación del servicio en Spark por medio de una referencia a un método Java. El método debe ser estático y devolver un *String* que representa la respuesta del servicio. Para cada servicio, se tendrá un método en una clase controlador que actúe como su manejador, brindando la implementación del mismo. La clase controlador será definida en base al participante que brinda dicho servicio REST, en tanto que el método a ser invocado será derivado del elemento *Operation* que representa al servicio. La figura 4.13 muestra cómo se define el *callback* para un ejemplo concreto.

```
get("/myoperation", MyParticipantController::myOperationImplementation);
```

Figura 4.13: Ejemplo en código Java de la definición del callback asociado a un servicio. El mismo, indica que la implementación del servicio será dada por el método "myOperationImplementation" de la clase MyParticipantController.

Por último, el método HTTP y la URL del servicio, son datos que existen en los elementos *Operation* extendidos en el dominio extBPMN, y que por lo tanto, deben ser consultados por la transformación. Estos datos serán utilizados en la definición de las rutas a los servicios REST, en conjunto con el *callback*.

La figura 4.14 muestra la plantilla ejecutada para generar el archivo Main.java, definida en *generateMainClass.mtl*. En la misma, se utiliza la consulta *getServiceProviders* para obtener los participantes que proveen al menos algún servicio REST, y la consulta *getProvidedOperations* para obtener los servicios proveídos por un determinado participante. Dado que los servicios se obtienen como elementos *Operation*, se hace uso de las consultas *getServiceUrl* y *getServiceHttpMethod* para recuperar la URL y el método HTTP de los mismos. También se utilizan las funciones *toUpperCamelCase* y *toJavaMethod* para manipular y formatear los atributos del tipo *String*. Notar la directiva utilizada al principio del módulo para incluir las consultas definidas en *queries.mtl*.

```
[comment encoding = ISO-8859-1 /]
[module generateMainClass('http://www.omg.org/spec/BPMN/20100524/MODEL')]
[import org::eclipse::contributions::bpmn2rest::acceleo::module::main::queries /]

[template public generateMainClass(root : Definitions) { serviceProviders : Set(Participant) = getServiceProviders(root); }]
[file ('Main.java', false, 'Cp1252')]
import static spark.Spark.*;
[if (not serviceProviders->isEmpty())]
import controllers.*;
[/if]

public class Main {

    /**
     * service route definitions using Spark
     *
     * @param  String['[]'/] args
     * @return void
     */
    public static void main(String['[]'/] args) {
        [for (p : Participant| serviceProviders)]

        /* [p.name/] exposed services */
        [for (op : Operation | getProvidedOperations(p))]

        [getServiceHttpMethod(op)/]("[getServiceUrl(op, p)/]", [toUpperCamelCase(p.name)/]Controller::[toJavaMethod(op.name)/]);
        [/for]
        [/for]
    }

}
[/file]
[/template]
```

Figura 4.14: Código del módulo "generateMainClass.mtl". El mismo define cómo se genera la clase Main.java en una ejecución de la transformación.

Generación automática de las clases controlador

Si bien las clases controlador no son un requisito del *framework* Spark para estructurar el código Java, fueron adoptadas como un mecanismo adecuado para aislar los servicios que provee cada participante de los demás, de manera ordenada, manteniendo la legibilidad del código fuente y adhiriendo al patrón MVC[1]. Como se mencionó antes, cada servicio REST que compone una aplicación es implementado mediante un *callback* que, comúnmente, se especifica con una expresión *lambda* en Java. Esta expresión invocará al método X de la clase Y, donde X es la versión *lowerCamelCase* del nombre del elemento *Operation*, mientras que Y es la versión *UpperCamelCase* del nombre del participante, con el sufijo "Controller".

Una clase controlador encapsula todas las implementaciones de los servicios que provee un participante en un modelo extBPMN. Cada implementación toma la forma de un método estático (*static*) en Java que recibe los argumentos Request y Response. Una clase controlador nombrada "XController" existirá sólo si existe un participante con el nombre "X" que brinda al menos un servicio REST a través de una interfaz BPMN (ver criterio 3 en sección 4.1.1). A su a vez, dicha clase tendrá tantos métodos como servicios REST provea el participante asociado a la misma. Se tomó la decisión de emplear métodos estáticos para definir el código de cada servicio, con el fin de simplificar el manejo de dependencias en el *framework* Spark. Dado que no es posible conocer la implementación en código de un determinado servicio REST, la transformación define un esqueleto o *stub* para cada uno de ellos, con una implementación estándar que devuelve la respuesta "Example Response". De esta manera, escribir el código Java que define qué es lo que hace el servicio, queda a cargo de los desarrolladores finales que utilicen la herramienta.

La figura 4.15 muestra la plantilla que genera las clases controlador, definida en *generateControllerFiles.mtl*. Para obtener los participantes que proveen servicios (es decir, los candidatos a ser clases controlador), se utiliza la consulta *getServiceProviders*, mientras que cada servicio que provee un participante se obtiene mediante la consulta *getProvidedOperations*. Notar que cada clase controlador se genera bajo la ruta "/controllers/NombreParticipanteController.java", lo cual la ubica dentro del directorio "controllers".

```
[comment encoding = ISO-8859-1 /]
[module generateControllerFiles('http://www.omg.org/spec/BPMN/20100524/MODEL')]
[import org::eclipse::contributions::bpmn2rest::acceleo::module::main::queries /]

[template public generateControllers(root : Definitions)]
[for (p : Participant| getServiceProviders(root))]
[file ('controllers/'.concat(toUpperCamelCase(p.name).concat('Controller.java')), false, 'Cp1252')]
package controllers;

import spark.Request;
import spark.Response;

public class [toUpperCamelCase(p.name)/]Controller {
    [for (op : Operation | getProvidedOperations(p))]

    /**
     * controller method for handling [op.name/] operation
     *
     * @param  req Request
     * @param  res Response
     * @return     String
     */
    public static String [toJavaMethod(op.name)/](Request req, Response res) {
        return "Example Response"; // replace this with the service implementation
    }
    [/for]
}

[/file]
[/for]
[/template]
```

Figura 4.15: Código del módulo "generateControllerFiles.mtl". El mismo define cómo se generan las clases controlador en una ejecución de la transformación.

Definición del módulo principal

Una transformación M2T en Acceleo siempre se compone de un módulo principal y, opcionalmente, otros módulos secundarios. Generalmente, el módulo principal se denomina *main.mtl*, aunque no es un requisito mandatorio. El módulo principal debe contener la plantilla desde la cual se inicia la ejecución de la transformación. Esta plantilla es señalada con el comentario especial *@main* y es la primera en ser procesada por el motor de Acceleo. Dado que el módulo principal es el que invoca a las consultas y plantillas definidas en los otros módulos, es común que contenga directivas *import* y configuraciones a ser utilizadas durante la transformación M2T.

[1]MVC son las siglas de Modelo-Vista-Controlador o *Model-View-Controller*. Según [55], MVC es un patrón arquitectónico ampliamente adoptado para el desarrollo de sistemas web, que se rige por la división del software en tres capas: el modelo, que encapsula la lógica del negocio y el manejo de los datos; la vista, que encapsula la lógica de la presentación de la información; y el controlador, que encapsula la interacción con el usuario.

La figura 4.16 muestra el módulo principal definido en *main.mtl*. En el mismo, se observan las directivas necesarias para importar los módulos *generateMainClass.mtl* y *generateControllerFiles.mtl*, y la invocación de cada plantilla en el orden correspondiente, dentro de la plantilla principal *main* que se ejecuta sobre el elemento *Definitions* del modelo de entrada.

```
[comment encoding = UTF-8 /]
[module main('http://www.omg.org/spec/BPMN/20100524/MODEL')]
[import org::eclipse::contributions::bpmn2rest::acceleo::module::main::generateMainClass /]
[import org::eclipse::contributions::bpmn2rest::acceleo::module::main::generateControllerFiles /]

[template public main(root : Definitions)]
[comment @main /]
[root.generateMainClass()/]
[root.generateControllers()/]
[/template]
```

Figura 4.16: Código del módulo principal "main.mtl".

Construcción del plugin para ejecutar la transformación

El desarrollo de la transformación BPMN2REST empleando Acceleo se realizó siguiendo una serie de pasos bien definidos en la documentación de dicha herramienta (ver [24]).

En primera instancia, se creó un proyecto del tipo *Acceleo Project*. Allí, se configuró el metamodelo de BPMN para ser utilizado en la transformación (ver figura 4.17). Al iniciar el proyecto, Acceleo genera de manera automática un módulo principal *main.mtl*, y una clase Java que permite ejecutar la transformación empleando el motor de ejecución de la herramienta.

Durante el desarrollo de los módulos que componen la transformación, fue indispensable el uso de la documentación de Acceleo ([24]) en la construcción de las consultas, y en la definición de las plantillas. Aquí, el mecanismo de autocompletado integrado en la interfaz de edición de texto de la herramienta aceleró notablemente el desarrollo (ver figura 4.18).

Una vez desarrollada la transformación, fue posible ejecutarla haciendo uso de la configuración de ejecución que, nuevamente, Acceleo generó de manera automática al momento de iniciar el proyecto (ver figura 4.19). Utilizando este artefacto, es posible indicar un modelo que actuará como la entrada de la transformación (en este caso, un archivo *.bpmn*), y un directorio de salida donde se podrán ver los documentos generados. Esta configuración permitió probar la transformación con distintos modelos BPMN y evaluar el resultado obtenido con cada uno de ellos.

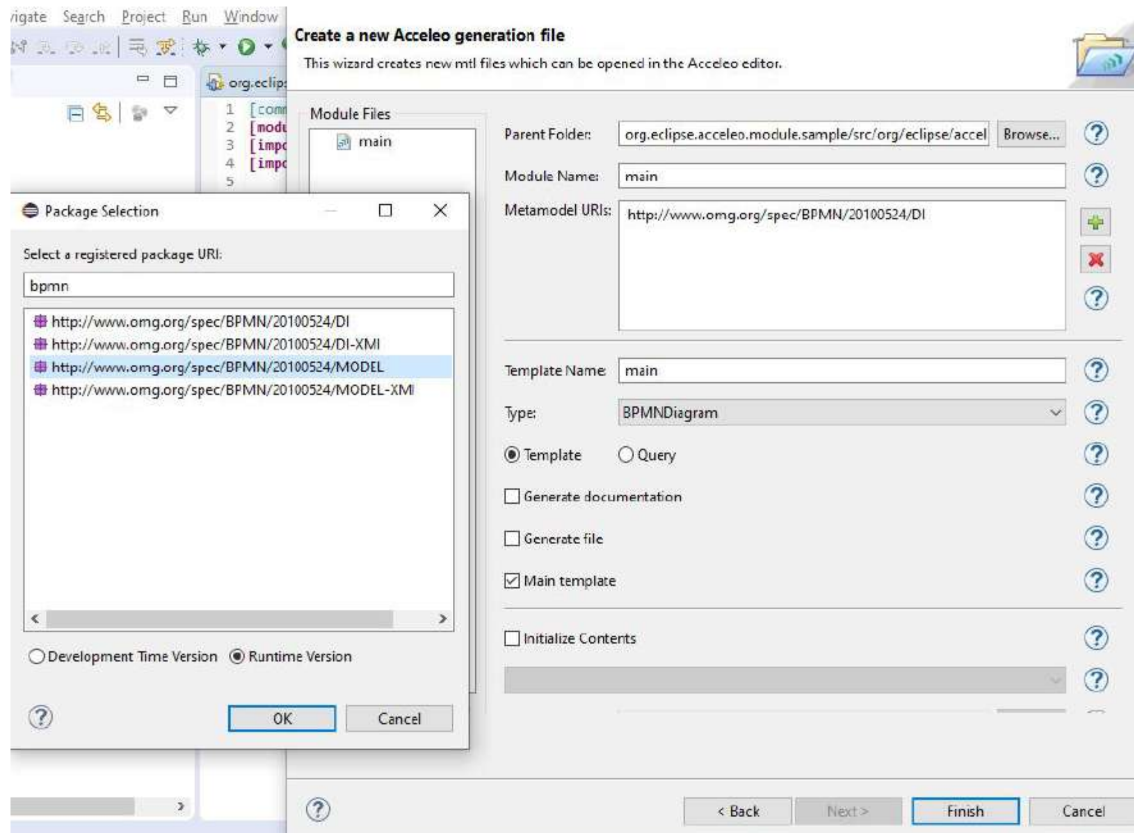


Figura 4.17: Ventanas de configuración inicial para un proyecto en Acceleo. Entre otras configuraciones, debe indicarse el metamodelo con el que operará la transformación.

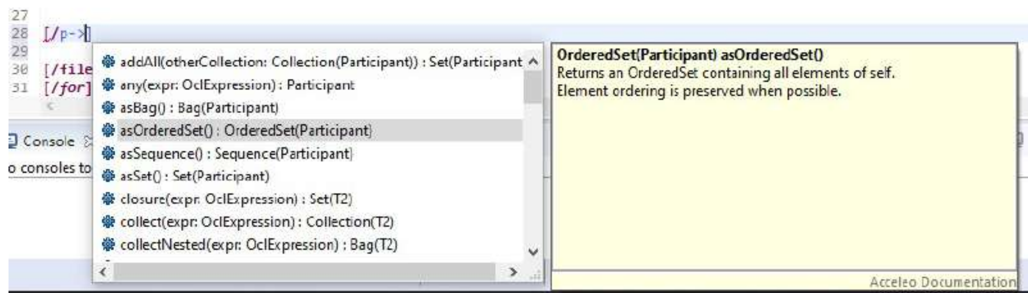


Figura 4.18: Mecanismo de autompletado integrado en Acceleo. Conforme el usuario escribe el código de la transformación, se le presentan una serie de alternativas de autocompletado para nombres de funciones OCL. Opcionalmente puede mostrarse la documentación de dichas funciones a modo de guía rápida.

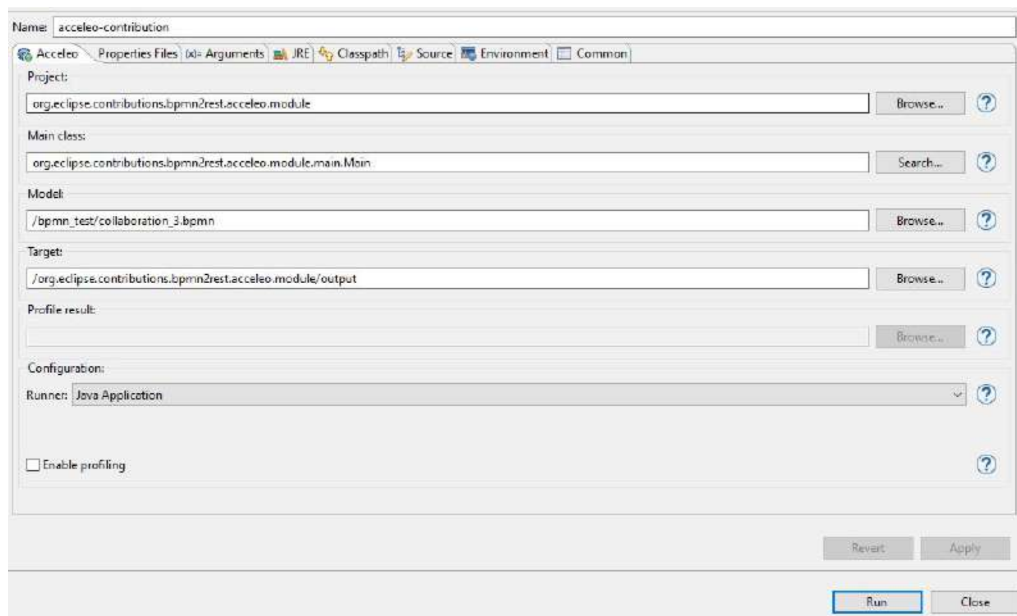


Figura 4.19: Configuración de ejecución generada por Acceleo. En la misma, se puede indicar el proyecto a ser ejecutado, el modelo BPMN de entrada, y el directorio donde deben generarse los archivos.

Finalmente, Acceleo permite exportar de manera automática las transformaciones desarrolladas en el contexto de un proyecto Acceleo, generando un *plugin* para Eclipse IDE. Esta utilidad permitió usar la transformación desarrollada dentro del entorno de Eclipse, como si se tratara de una característica nativa del mismo. Los *plugins* generados por Acceleo contienen los artefactos Java necesarios para ejecutar la transformación desarrollada. En este caso, la clase generada se denomina *GenerateAll* y la invocación del método *doGenerate* sobre una instancia de la misma, ejecuta la transformación BPMN2REST en un modelo extBPMN de entrada. La clase *GenerateAll* constituye el motor de ejecución de la transformación, provisto por Acceleo.

4.3. Definición e implementación de la validación BPMN2REST

Durante su ejecución, la transformación BPMN2REST efectúa una serie de consultas sobre el modelo extBPMN de entrada, a fin de obtener los datos necesarios para generar el código Java bajo el *framework* Spark. Las consultas aplicadas sobre el modelo de entrada, se definen en términos del metamodelo de la notación BPMN, en base a expresiones OCL que son evaluadas durante la ejecución de la transformación, como se abordó anteriormente. Estas consultas operan con la estructura interna del modelo extBPMN asumiendo que el mismo es adecuado para ejecutar la transformación. Dado que esto no siempre es así, resulta necesario incorporar un mecanismo que permita validar dicha estructura, a fin de evitar posibles evaluaciones erróneas que puedan ser arrojadas como resultado de la transformación, generando código Java no funcional.

En la presente sección se abordará la construcción del mecanismo de validación BPMN2REST, destinado a garantizar que los modelos extBPMN sobre los cuales se ejecute la transformación, cumplan con los requisitos mínimos necesarios para que las consultas sean ejecutadas correctamente.

4.3.1. Definición de las reglas de validación

El mecanismo de validación adoptado, deberá evaluar un conjunto de condiciones o reglas sobre el modelo extBPMN de entrada, arrojando resultados positivos o negativos de acuerdo al

cumplimiento o no de las mismas. Conforme a la observación de las consultas que definen la transformación BPMN2REST, se definieron cinco reglas de validación a ser evaluadas sobre los modelos:

(a) *Debe existir un diagrama de colaboraciones y sólo uno.*

El elemento *Definitions* es el contenedor principal de un modelo BPMN (ver sección 2.1.2). Este elemento contiene cada *Collaboration* que compone al documento BPMN final. Dado que un elemento *Collaboration* representa un diagrama de colaboraciones BPMN y, la transformación BPMN2REST fue diseñada para ser aplicada a un único diagrama; esta condición garantiza que la misma no sea ejecutada sobre un documento BPMN que contenga varios diagramas de este tipo.

(b) *El modelo debe contener al menos dos participantes.*

El elemento *Collaboration* es el contenedor de los elementos *Participant* que representan a los distintos actores a lo largo de un modelo de procesos de negocio. En tal caso, si dicho modelo consta de menos de dos participantes, entonces es claro que no existirán comunicaciones que puedan ser implementadas mediante servicios REST, y por lo tanto no tendrá sentido alguno aplicar la transformación.

(c) *El modelo debe contener al menos una comunicación entre los participantes del negocio.*

De manera similar al punto anterior, si no existe al menos un flujo de mensajes (*MessageFlow*) entre dos participantes del proceso de negocios, entonces no existirá ningún servicio REST que pueda implementarse.

(d) *Al menos un participante debe implementar una interfaz en el modelo.*

Dado que cada servicio REST es obtenido por la transformación a través de un *match* entre las operaciones que provee un participante y los mensajes que recibe el mismo, es necesario que dicho participante implemente al menos una interfaz (*Interface*) BPMN, donde se definan las operaciones proveídas a los demás actores del negocio. Si no existiese ningún participante implementando una interfaz, nuevamente, ejecutar la transformación sobre el modelo será totalmente en vano.

(e) *No pueden existir dos operaciones distintas que referencien al mismo mensaje dentro de un participante proveedor de servicios.*

Por último, dado que BPMN2REST identifica los servicios REST proveídos por un participante haciendo un match entre una operación (*Operation*) que brinde el mismo, y un mensaje que recibe (mediante un *MessageFlow* entrante); no pueden existir dos operaciones distintas que referencien al mismo mensaje bajo el rol de *input*. Si ese es el caso, entonces el mensaje referenciado no debe asociarse con ningún flujo de mensajes que entre al participante en cuestión. Esta condición busca cuidar que BPMN2REST genere el código en Spark sólo para los servicios que son explícitamente solicitados en el modelo de procesos del negocio.

Las reglas de validación definidas garantizan la ejecución sin errores de BPMN2REST, permitiendo obtener código Java funcional, en base al modelo de entrada utilizado. La tabla 4.4 muestra la definición formal de cada regla de validación mediante la notación OCL.

4.3.2. Implementación de la validación empleando BPMN2 Modeler

Anteriormente, se abordó la extensión de Eclipse BPMN2 Modeler a partir del desarrollo de un *plugin* cuyo objetivo fue definir la estructura de los modelos extBPMN. Este mismo enfoque fue adoptado para implementar las reglas de validación previamente definidas, empleando las utilidades que provee la herramienta.

El *plugin* BPMN2 Modeler contiene un conjunto de reglas de validación establecidas por defecto, cuya ejecución sobre un modelo BPMN construido con la herramienta, permite conocer si la estructura interna del modelo es correcta o si es necesario corregir elementos del mismo. Una regla de validación que no se cumple en un modelo BPMN, arroja una notificación que es capturada por Eclipse IDE y mostrada al usuario, permitiéndole conocer cuál es el error que ha cometido. La figura 4.20 muestra la notificación al usuario emitida durante el incumplimiento de dos reglas de validación que incorpora BPMN2 Modeler por defecto: el evento de finalización "End Event 1" no se conecta con el resto de los elementos mediante un flujo de secuencia, y la tarea "Task 1" no posee una conexión de salida hacia otro elemento BPMN.

Como se indica en [56], BPMN2 Modeler utiliza el *framework EMF Validation Framework* para definir las reglas de validación a ser aplicadas sobre los modelos construidos con dicha herramienta. A partir del uso de esta utilidad que proporciona el *core* de EMF (ver sección 2.5.5), las reglas de validación son declaradas y asociadas a los elementos de la notación BPMN, para luego ser evaluadas en tiempo de ejecución sobre cada uno de los modelos, notificando el incumplimiento de alguna de ellas.

Descripción de la regla de validación	Especificación en OCL
Debe existir sólo un diagrama de colaboraciones	context Definitions inv: self ->rootElements ->select(e	e.oclIsKind(Collaboration)) ->size() = 1
Deben existir al menos 2 participantes en el modelo	context Collaboration inv: self ->participants ->size() >= 2
Debe existir al menos una comunicación en el modelo	context Collaboration inv: self ->messageFlows ->select(mf	mf->messageRef.oclIsKind(Message)) ->size() >= 1
Debe existir al menos un participante que implemente una interfaz	context Collaboration inv: self ->participants ->select(p	p->interfaceRefs->notEmpty()) ->size() >= 1
No pueden existir dos operaciones distintas que referencien al mismo mensaje dentro de un participante proveedor de servicios	context Collaboration inv: self ->participants ->forAll(p | p.interfaceRefs->collect(i | i.operations)->flatten() ->forAll(op1, op2 | op1.inMessageRef <>op2.inMessageRef or op1 = op2 or self.messageFlows->select(mf | mf.messageRef = op1.inMessageRef and (mf.targetRef = p or (not p.processRef.oclIsUndefined() and p.processRef.flowElements->includes(mf.targetRef))))->size() = 0))

Tabla 4.4: Definición formal de las reglas de validación empleando la notación OCL.

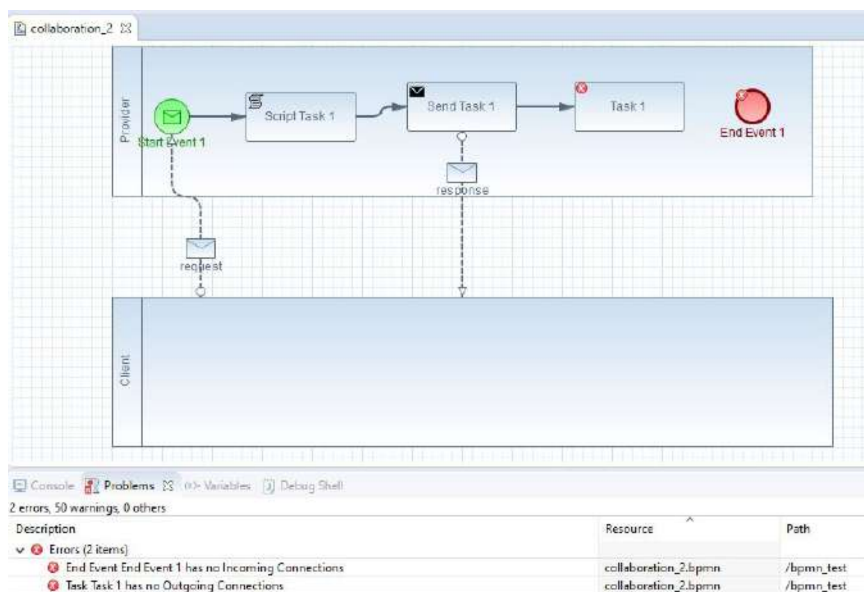


Figura 4.20: Ejemplo de notificación de los errores obtenidos al construir un modelo BPMN inválido utilizando BPMN2 Modeler.

BPMN2 Modeler incorpora el punto de extensión *org.eclipse.emf.validation.constraintProviders* (que a su vez, es definido por el *core* de EMF), definiendo las distintas reglas de validación que deben ser evaluadas por defecto en cualquier modelo construido con esta herramienta. Incorporando este mismo punto de extensión, es posible añadir y/o sobreescribir las reglas definidas para un elemento de la notación, extendiendo el mecanismo de validación de BPMN2 Modeler. Cada regla de validación se define empleando el elemento *constraint* dentro del punto de extensión; a su vez, todas las reglas definidas deben agruparse dentro de una “categoría de reglas”, definida mediante el elemento *category*. Esto último permite realizar operaciones sobre conjuntos de reglas o filtrarlas de acuerdo a la categoría que corresponden. El anexo A7 detalla la estructura del punto de extensión *org.eclipse.emf.validation.constraintProviders*.

Las reglas incorporadas serán evaluadas automáticamente por BPMN2 Modeler, mostrando las notificaciones correspondientes, según sea necesario. Esto último implica que la acción de implementar una regla de validación personalizada sobre el motor de BPMN2 Modeler resulte prácticamente declarativa: basta con indicar cómo se estructuran las reglas, y el *plugin* se encargará de ejecutarlas cuando sea necesario, y de notificar al usuario el incumplimiento de cada una de ellas.

Una regla de validación incorporada mediante BPMN2 Modeler, puede ser evaluada de dos maneras: la evaluación *batch* o por lotes, es ejecutada luego de que los cambios en el modelo son guardados, y generalmente es utilizada para realizar controles semánticos que pueden ser corregidos fácilmente. Por su parte, la evaluación *live* o en tiempo real, se ejecuta en el mismo momento en el que se ha realizado un cambio en el modelo (aun sin haber sido guardado), impidiendo que éste pase a un estado corrupto donde no pueda ser interpretado nuevamente por la herramienta. Debe notarse entonces que las validaciones en tiempo real comúnmente abarcan reglas que son más críticas para el correcto funcionamiento del *plugin*. Esto último determina la elección de la modalidad *batch* para implementar las reglas definidas anteriormente.

Por último, una regla de validación puede ser especificada en código Java, indicando una clase que implemente la interfaz *org.eclipse.emf.validation.AbstractModelConstraint* y que provea la evaluación de la condición; o mediante la utilización de la notación formal OCL. Para definir la evaluación de las reglas, se optó por la última alternativa. La figura 4.21 muestra la incorporación de la regla de validación (b), mediante la definición del elemento *constraint* dentro del punto de extensión en cuestión. En la misma, se observa el elemento BPMN sobre el cual debe ser evaluada (etiqueta *target*), el modo de evaluación (atributo *mode*), y la condición a ser evaluada, expresada en OCL.

```
<constraint
      id="org.eclipse.contributions.bpmn2rest.validation.twoParticipants"
      isEnabledByDefault="true"
      lang="OCL"
      mode="Batch"
      name="HasAtLeastTwoParticipantsConstraint"
      severity="ERROR"
      statusCode="1">
   <message>
      Collaborations must have at least 2 participants
   </message>
   <target
         class="Collaboration">
   </target>
   <![CDATA[
        self.participants->size() > 1
   ]]>
</constraint>
```

Figura 4.21: Definición de la regla de validación (b) mediante OCL, dentro del punto de extensión org.eclipse.emf.validation.constraintProviders incorporado en el archivo plugin.xml.

Si bien las reglas de validación declaradas al incorporar el punto de extensión *org.eclipse.emf.validation.constraintProviders* pueden ser interpretadas por BPMN2 Modeler, las mismas no serán evaluadas si no se asocian a una extensión de dicha herramienta. Esta tarea resulta necesaria, nuevamente, por la posibilidad de que coexistan múltiples extensiones de BPMN2 Modeler a lo largo de varios proyectos, donde cada uno implemente sus propias reglas de validación. Para asociar un conjunto de reglas de validación a una extensión de BPMN2 Modeler, se debe incorporar el punto de extensión *org.eclipse.emf.validation.constraintBindings*, como se muestra en el anexo A7. A partir de aquí, BPMN2 Modeler podrá interpretar correctamente las reglas definidas y evaluarlas sobre los modelos construidos con la extensión desarrollada anteriormente.

La figura 4.22 muestra la notificación emitida por BPMN2 Modeler al advertir el incumplimiento de la regla de validación (c) sobre un modelo de procesos de negocio donde existen dos participantes (A y B), sin ninguna comunicación entre ambos.

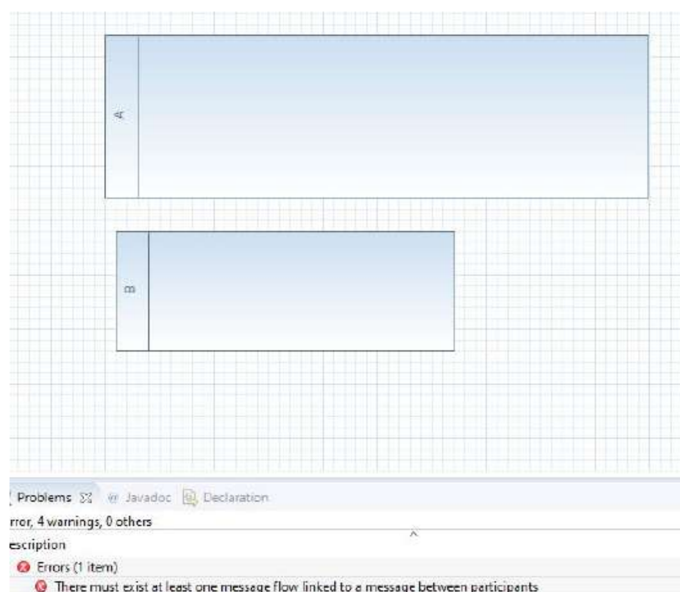


Figura 4.22: Ejemplo de evaluación de las reglas incorporadas. El modelo que aparece en la figura, no contiene ningún mensaje entre participantes, y por lo tanto no cumple la regla (c).

4.4. Construcción del *plugin* BPMN2REST para Eclipse IDE

Cada uno de los desarrollos abordados hasta aquí, puede ser integrado en la confección de un *plugin* para Eclipse IDE que facilite el uso de dichas herramientas desde este entorno de desarrollo. La construcción de la extensión para Eclipse, permite poner a disposición de los usuarios del IDE cada una de las herramientas confeccionadas, tal como si se tratase de alguna característica nativa del mismo.

En la presente sección, se exponen los detalles de la construcción del *plugin* para Eclipse IDE que permite a los usuarios utilizar la transformación BPMN2REST en modelos extBPMN, desde la comodidad del propio entorno de desarrollo.

4.4.1. Estructura de la extensión para Eclipse IDE

Según lo abordado en las secciones anteriores, en primer lugar fue necesario extender la notación BPMN adicionando algunas características que permitan dar soporte al dominio REST, dando origen a los modelos del tipo extBPMN. Esta extensión, implementada mediante BPMN2 Modeler, fue llevada a cabo a partir de la construcción de un *plugin* que incorpore el punto de extensión *org.eclipse.bpmn2.modeler.runtime*.

A continuación, la definición de la transformación BPMN2REST empleando el lenguaje MTL dio origen a un proyecto del tipo "Acceleo" (que encapsula los módulos *.mtl*). Adicionalmente, haciendo uso de las utilidades que provee el entorno de desarrollo de Acceleo, se construyó un *plugin* para Eclipse IDE que permite ejecutar dicha transformación en un modelo extBPMN de entrada.

Por último, la implementación de las reglas de validación para los modelos de entrada, se realizó definiendo un nuevo *plugin* sobre BPMN2 Modeler, utilizando los puntos de extensión *org.eclipse.emf.validation.constraintProviders* y *org.eclipse.emf.validation.constraintBindings*.

De esta manera, los desarrollos realizados dieron origen a las siguientes extensiones:

(a) Extensión de BPMN2 Modeler para dar soporte a extBPMN.

(b) Plugin que define la transformación BPMN2REST (Proyecto del tipo "Acceleo" que contiene los módulos *.mtl*).

(c) Extensión sobre Eclipse IDE que permite ejecutar BPMN2REST en un modelo de entrada (generado automáticamente por Acceleo).

(d) Extensión de BPMN2 Modeler para implementar las reglas de validación.

Cada una de estas extensiones, se materializa en un proyecto del tipo *plugin*, que extiende características de Eclipse IDE, o de BPMN2 Modeler (notar que al extender el *plugin* BPMN2 Modeler, se extiende también Eclipse IDE, pues el primero es a su vez una extensión del segundo). Es así, entonces, que cada uno de ellos está definido por un archivo *plugin.xml* que describe los puntos de extensión incorporados (ver sección 2.5.4). Dado que cada punto de extensión de cada uno de estos proyectos no entra en conflicto con los demás, es posible agruparlos a todos bajo un único *plugin*, definiendo un único archivo *plugin.xml* que incorpore cada punto de extensión, y colocando de manera ordenada los demás artefactos que utiliza cada uno.

Existe una única limitante en la idea planteada anteriormente: Acceleo impone que el *plugin* para ejecutar la transformación (c), y el *plugin* que la define (b), no se agrupen bajo una misma extensión. Esto implica, que ambos desarrollos no pueden ser integrados definiendo un archivo *plugin.xml* común para los dos. Teniendo en cuenta esta restricción y aplicando el planteo abordado anteriormente, se estructura la extensión BPMN2REST para Eclipse IDE como el conjunto formado por los dos *plugins*:

- *org.eclipse.contributions.bpmn2rest.acceleo.module*: define la transformación BPMN2REST en base a los módulos *.mtl*. Contiene el desarrollo (b).
- *org.eclipse.contributions.bpmn2rest.acceleo.module.ui*: agrupa los desarrollos (a), (c) y (d). Posibilita ejecutar BPMN2REST desde el entorno del IDE.

La figura 4.23 ilustra la composición de la extensión BPMN2REST para Eclipse IDE, y la relación entre sus componentes. El *plugin* *org.eclipse.contributions.bpmn2rest.acceleo.module.ui* depende de *org.eclipse.contributions.bpmn2rest.acceleo.module* ya que el primero contiene las utilidades para poder ejecutar la transformación BPMN2REST, definida en el segundo.

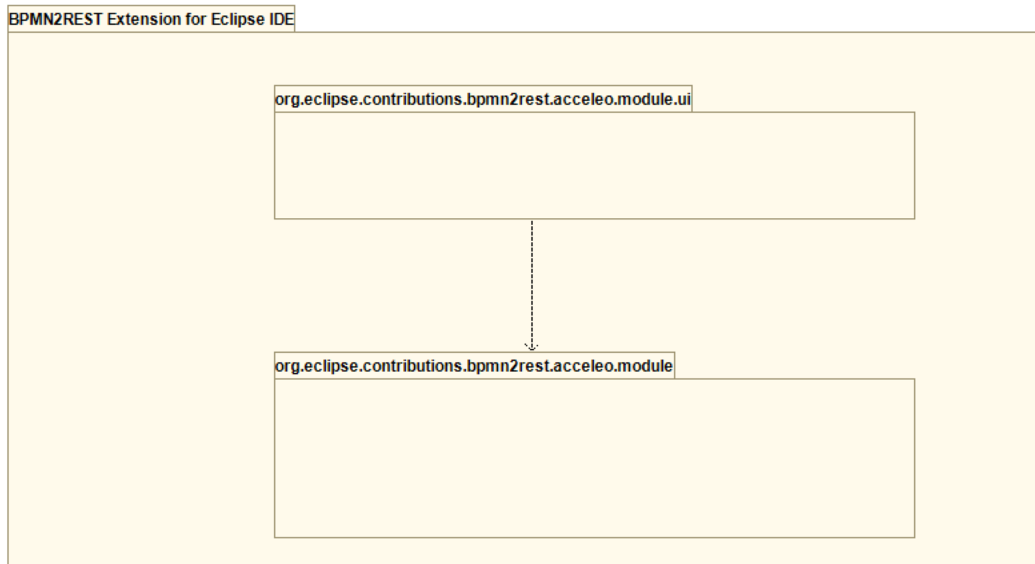


Figura 4.23: Composición de la extensión BPMN2REST para Eclipse IDE.

El *plugin org.eclipse.contributions.bpmn2rest.acceleo.module.ui* encapsula los desarrollos que deben ser adicionados a la definición de la transformación BPMN2REST, con el objetivo de evitar o corregir conflictos que puedan existir durante la ejecución de la misma: dicho desarrollo se compone de la definición de la extensión extBPMN, la incorporación de la reglas de validación BPMN2REST, y el motor de ejecución necesario para poder llevar a cabo la transformación en un modelo de entrada.

Si bien la extensión BPMN2REST para Eclipse IDE se estructura en dos *plugins* distintos que trabajan en conjunto, es posible agruparlos a ambos bajo una única característica o *feature* instalable en Eclipse IDE utilizando la herramienta PDE. Esto garantiza que ambos *plugins* son instalados en simultáneo sobre el IDE. Además, una característica instalable en Eclipse se muestra al usuario como tal, sin entrar en detalles sobre los *plugins* y artefactos que la componen. Esto permite ocultar información acerca de las dependencias y relaciones entre los componentes de la característica, brindando transparencia hacia el usuario final.

4.4.2. Mejora de la extensión BPMN2REST utilizando la API de Eclipse IDE

Si bien el *plugin org.eclipse.contributions.bpmn2rest.acceleo.module.ui* permite ejecutar la transformación BPMN2REST exitosamente para un modelo de entrada, el mismo presenta algunas limitantes que hacen que no resulte adecuado para sus usuarios finales. En esta sección se exponen las dificultades que presentó el *plugin* al momento de ser utilizado para trabajar, y las soluciones propuestas para cada una de ellas. Cada problemática fue abordada haciendo uso de las utilidades que provee la API de Eclipse IDE para acceder a las funcionalidades del *core* de dicho entorno.

Selección del directorio de salida de la transformación

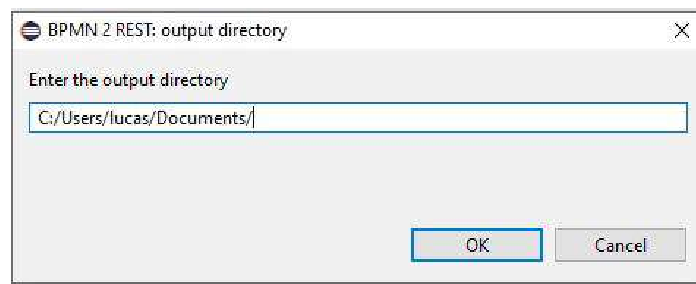
La ejecución de BPMN2REST sobre un determinado modelo extBPMN de entrada, arroja como resultado un conjunto de archivos conformado por la clase *Main*, el directorio *controllers* y las clases *Controller*. De acuerdo a lo anterior, resulta conveniente que para cada ejecución de BPMN2REST, el usuario pueda señalar el directorio de salida donde deben ser ubicados estos archivos.

La clase *org.eclipse.jface.dialogs.InputDialog* provista por el *core* de Eclipse IDE, permite generar una ventana en el contexto del IDE, que contiene un campo de entrada de texto, un botón de "aceptar" y otro de "cancelar". El comportamiento por defecto del botón de cancelación cierra la ventana mostrada, mientras que, la acción ejecutada al pulsar el botón de aceptar requiere ser programada extendiendo dicha clase. En adición, una instancia de *InputDialog* puede emplear a un objeto del tipo *org.eclipse.jface.dialogs.IInputValidator* cuyo objetivo es asegurar que el texto introducido por el usuario sea válido de acuerdo a algún criterio definido. De esta manera, las siguientes clases Java fueron añadidas al *plugin* a fin de permitir la selección del directorio de salida:

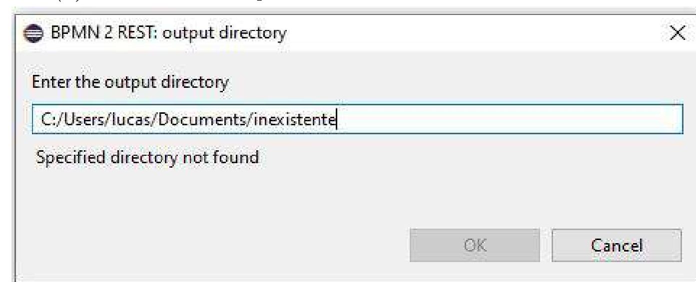
1. OutputDirectoryInputDialog: extiende a *InputDialog*, e implementa el comportamiento de la ventana al presionar el botón de aceptar.
2. IsDirectoryValidator: implementa la interfaz *IInputValidator* validando el texto introducido en la ventana. Valida que el texto sea un *path* hacia un directorio válido.

Al crear una instancia de *OutputDirectoryInputDialog*, el IDE renderiza una ventana en la interfaz del usuario, permitiéndole introducir la ruta hacia el directorio de salida (donde se generarán los

archivos al ejecutar la transformación). La entrada del usuario será validada por *IsDirectoryValidator* asegurando que sólo pueda presionar el botón de aceptar si la ruta introducida es válida y conduce a un directorio existente, como lo muestra la figura 4.24. En dicho caso, el botón de aceptar dará inicio a la ejecución de BPMN2REST para un modelo de entrada, utilizando el directorio indicado como el contenedor de los archivos a ser generados.



(a) El botón de "aceptar" se habilita si el directorio es válido.



(b) El botón de "aceptar" se deshabilita y se muestra un mensaje de error si el directorio ingresado no existe.

Figura 4.24: Ventana de entrada del directorio de salida para los archivos generados por BPMN2REST. El path ingresado es validado en tiempo real, permitiendo que se ejecute la transformación sólo si el mismo referencia a una carpeta existente.

Validación de los modelos antes de la ejecución de BPMN2REST

En secciones anteriores se abordó la validación de los modelos extBPMN a fin de garantizar la ejecución sobre BPMN2REST en modelos que sean correctos. Las reglas de validación definidas e incorporadas en el *plugin* *org.eclipse.contributions.bpmn2rest.acceleo.module.ui* son ejecutadas por el *core* de BPMN2 Modeler sobre cada modelo extBPMN confeccionado, notificando el incumplimiento de alguna de ellas en la vista *Problems*.

No obstante, la validación de los modelos es prácticamente pasiva: si un modelo no cumple alguna de las reglas, se notifica al usuario, pero aun así es posible ejecutar la transformación sobre el mismo. Esto se debe a que el *plugin* de validación y el *plugin* para ejecutar BPMN2REST son completamente ajenos entre sí. Por consiguiente, resulta necesario crear una relación entre ambos que permita interrumpir o evitar la ejecución de BPMN2REST ante el incumplimiento de alguna de las reglas de validación sobre el modelo.

La ejecución de BPMN2REST es llevada a cabo por la clase *org.eclipse.contributions.bpmn2rest.acceleo.module.ui.common.GenerateAll*, generada automáticamente por Acceleo, como se abordó en la sección 4.2.2. Añadiendo un adaptador o *wrapper*[2] a la misma, es posible incorporar un paso más al proceso de ejecutar la transformación. En este nuevo paso, se validará el modelo de entrada según las reglas definidas anteriormente. Ante una evaluación negativa, se interrumpirá la ejecución de BPMN2REST, notificando el error detectado; mientras que ante una validación positiva, se proseguirá con la ejecución de la transformación sobre el modelo, utilizando una instancia de *GenerateAll*.

La clase *GenerateAll* constituye el motor de ejecución de la transformación BPMN2REST, y es generada automáticamente por Acceleo. Para llevar a cabo una ejecución de la transformación, primero se debe crear una instancia de la clase indicándole el *path* hacia el modelo extBPMN de entrada. Luego, la invocación del método *doGenerate* ejecutará la transformación generando los archivos de salida. Este método recibe como parámetro una instancia del tipo *org.eclipse.core.runtime.IProgressMonitor*, que permitirá visualizar el progreso de la transformación mientras esté en curso. La interfaz *IProgressMonitor* es provista por Eclipse y su implementación permite monitorear el progreso de una actividad mientras la misma se está ejecutando.

El *wrapper* definido, se denomina *ExecuteCommandRunnable* y se ubica en el paquete *org.eclipse.contributions.bpmn2rest.acceleo.module.ui.components*. Esta clase implementa la interfaz *org.eclipse.jface.operation.IRunnableWithProgress* provista por el *core* de Eclipse IDE. Un objeto del

[2] Una clase *wrapper* actúa como una envoltura para algún otro objeto, comportándose como una capa más de indirección hacia el mismo, según [57]. De esta manera, pueden añadirse comportamientos adicionales a la funcionalidad básica del objeto envuelto. Esta funcionalidad adicional estará presente en el *wrapper*.

tipo *IRunnableWithProgress* representa una acción o un proceso ejecutable, y por lo tanto, define su propio método *run* denotando allí el cuerpo de dicho proceso. Nuevamente, el método *run* recibe un objeto del tipo *IProgressMonitor* responsable de monitorear la ejecución del proceso en cuestión. Dentro del método *run* definido en el *wrapper*, se realiza la siguiente secuencia de acciones:

1. Se ejecuta la validación BPMN2REST sobre el modelo extBPMN.
2. Si la validación es negativa, se muestra un mensaje en la interfaz del usuario utilizando la clase *org.eclipse.jface.dialogs.MessageDialog* del *core* de Eclipse. A continuación, se finaliza la ejecución del método *run*.
3. Si la validación es positiva, se crea una instancia de *GenerateAll* y se ejecuta la transformación sobre el modelo de entrada invocando al método *doGenerate*, utilizando el mismo objeto *IProgressMonitor* recibido en el método *run*.
4. Si algún error ocurre durante la ejecución de BPMN2REST, se interrumpe la ejecución del método *run* y se notifica al usuario mediante un mensaje de error.
5. Si la ejecución de BPMN2REST finaliza exitosamente, se notifica al usuario mediante un mensaje informativo.

La figura 4.25 ilustra la secuencia descripta anteriormente en el contexto del método *run* del *wrapper*.

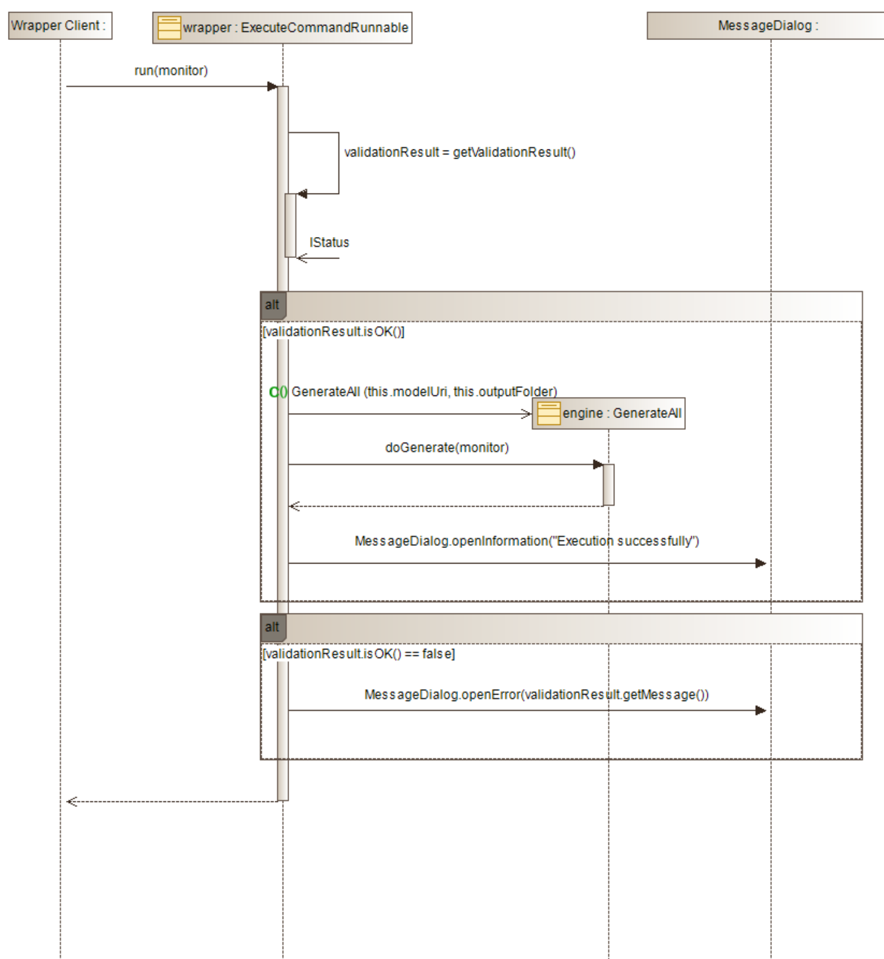


Figura 4.25: Diagrama de secuencia que ilustra la ejecución del método run del wrapper.

La validación del modelo extBPMN de entrada es llevada a cabo en el contexto de ejecución del *wrapper*. Allí, primeramente se obtiene la representación en memoria del modelo, parseando el archivo *.bpmn* ubicado en el *path* del mismo. Esta tarea es realizada por las clases *Resource*, *Resource.Factory*, *Resource.Factory.Registry*, y *ResourceSet* ubicadas en el paquete *org.eclipse.emf.ecore.resource* del *core* de EMF; en conjunto con *Bpmn2ResourceFactoryImpl* del paquete *org.eclipse.bpmn2.util*.

Utilizando el modelo en memoria, es posible ejecutar todas las reglas de validación (es decir, las definidas para los modelos extBPMN más las que define BPMN2 Modeler por defecto) sobre cada uno de los elementos del modelo. Sin embargo, sólo resulta de interés validar las reglas añadidas anteriormente, ya que si alguna de ellas no evalúa correctamente, no tendrá sentido continuar con la evaluación de las demás dado que la transformación no será ejecutada. En base a este contexto, es oportuno implementar la interfaz *org.eclipse.emf.validation.service.IConstraintFilter* provista por el *core* de Eclipse EMF. La misma, permite definir un filtro para las distintas reglas de validación que se aplicarán al modelo a ser evaluado. En base a ello, se definió la clase *CustomConstraintFilter* del tipo

IConstraintFilter cuya implementación, remueve todas las reglas de validación que provee BPMN2 Modeler por defecto, dejando sólo las que fueron definidas para los modelos extBPMN. El filtro definido para las reglas simplemente compara la categoría a la que pertenecen las mismas, descartando las que no correspondan al conjunto añadido en la sección 4.3.

La ejecución de la validación es llevada a cabo por una instancia de la clase *org.eclipse.emf.validation.service.IBatchValidator* (nuevamente, proporcionada por el *core* de EMF). Esta instancia es obtenida mediante la invocación del método *newValidator* de la clase *org.eclipse.emf.validation.service.ModelValidationService*. Esta última, se estructura como un *singleton* [3] que contiene métodos generales para validar modelos originados mediante el entorno EMF. Así, a la instancia de *IBatchValidator* obtenida, se le asocia el filtro de reglas definido anteriormente mediante la invocación de *validator.addConstraintFilter(new CustomConstraintFilter())* (*validator* es una instancia de *IBatchValidator*). En este punto, es posible ejecutar la validación mediante la invocación de *validator.validate(inMemoryModel)* sobre el modelo en memoria. El resultado de la ejecución estará encapsulado en una instancia de *org.eclipse.core.runtime.IStatus*. *IStatus* es una clase del *core* de Eclipse que permite representar el estado final de la ejecución de una tarea. Todo el proceso de validación, ocurre al invocar al método privado *getValidationResult* del *wrapper*. El diagrama de secuencia de la figura 4.26 ilustra la secuencia seguida en dicho proceso.

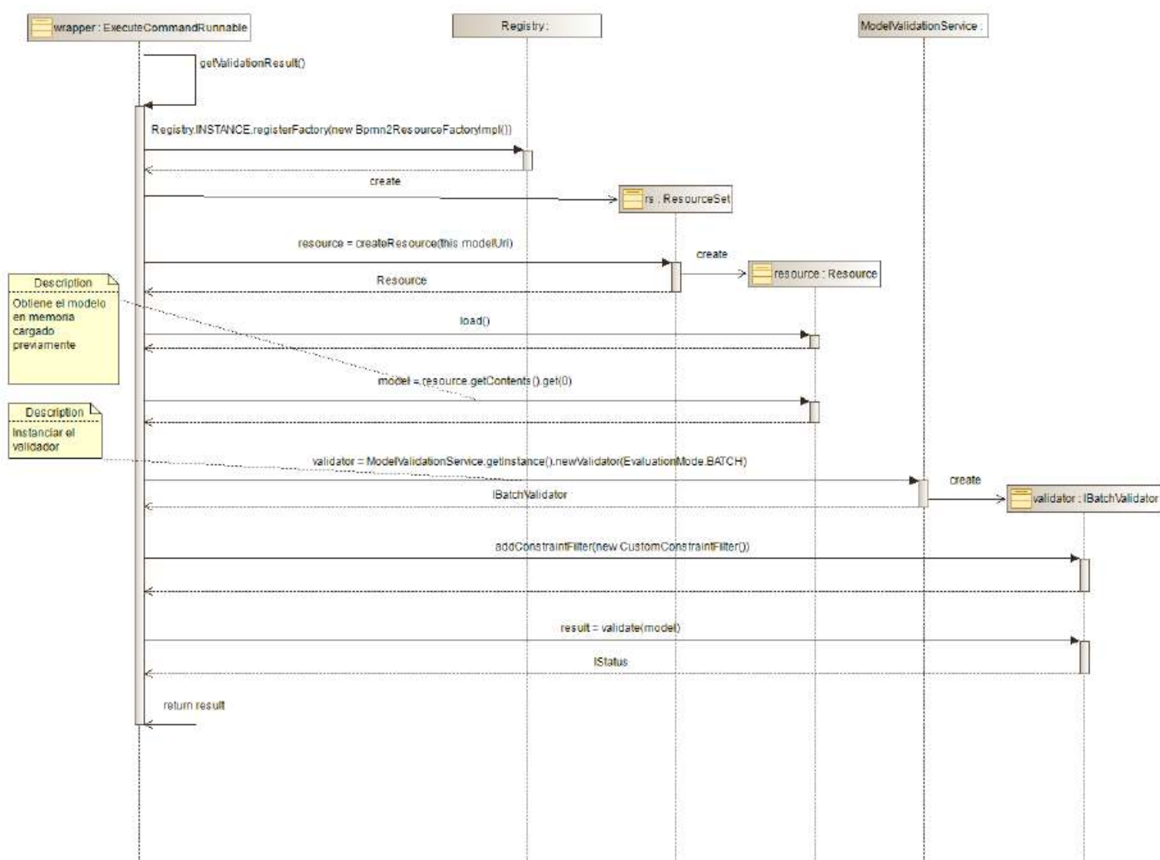


Figura 4.26: Diagrama de secuencia que ilustra el proceso de validación del modelo de entrada dentro del wrapper.

La ejecución del método *run* del *wrapper* construido, lleva a cabo la validación y la ejecución de la transformación BPMN2REST sobre un modelo de entrada. Aunque se trate de un proceso complejo y existan muchas dependencias entre objetos alrededor de una ejecución del *wrapper*, la tarea de iniciarlo es relativamente sencilla gracias a las utilidades del *core* de Eclipse. La invocación del método *run* de la interfaz *org.eclipse.ui.progress.IProgressService* recibe un parámetro del tipo *IRunnableWithProgress* (es decir, un proceso ejecutable), y se encarga de ejecutarlo, mostrando en la interfaz del usuario el progreso del mismo. El uso de este método es altamente conveniente ya que delega en el *core* de Eclipse IDE la inyección de la dependencia *IProgressMonitor* utilizada tanto por *GenerateAll* como por *ExecuteCommandRunnable*. La instancia principal de *IProgressService* es responsable de ejecutar los procesos que recibe por parámetro, mostrando el progreso de cada uno de ellos en la interfaz principal del IDE. Esta instancia puede ser obtenida invocando al método *getProgressService* de la interfaz *org.eclipse.ui.IWorkbench*. Esta interfaz, representa al componente principal de la interfaz del usuario en el contexto del IDE y permite acceder a servicios globales que provee la misma, con lo cual, raramente es implementada dado su alto nivel de complejidad. Generalmente, existe una única instancia de *IWorkbench* en tiempo de ejecución, y dicha instancia es accesible a través de varios

[3]El patrón de diseño *Singleton*, permite asegurar que sólo una instancia de una determinada clase es generada en tiempo de ejecución ([58]). Generalmente esta instancia es accesible mediante métodos estáticos que aseguran el cumplimiento de este comportamiento. La instancia de una clase que implementa este patrón de diseño se conoce como instancia *singleton*.

servicios que se encuentran en ejecución durante el ciclo de vida del IDE. La invocación del método *getWorkbench* de la clase *org.eclipse.ui.PlatformUI* es una manera sencilla de obtener una instancia de *IWorkbench*. La clase estática *PlatformUI* representa a la interfaz del usuario global que provee el IDE y es el punto de entrada principal para interactuar con la misma. En resumen, la invocación de *PlatformUI.getWorkbench().getProgressService().run* permitirá ejecutar una instancia del *wrapper* en cualquier momento. En base al desarrollo abordado en el apartado anterior, la ejecución del *wrapper* se llevará a cabo al presionar el botón de aceptar de la ventana generada por *OutputDirectoryInputDialog*. El diagrama de la figura 4.27 muestra la secuencia seguida luego de que se pulsa el botón de "aceptar".

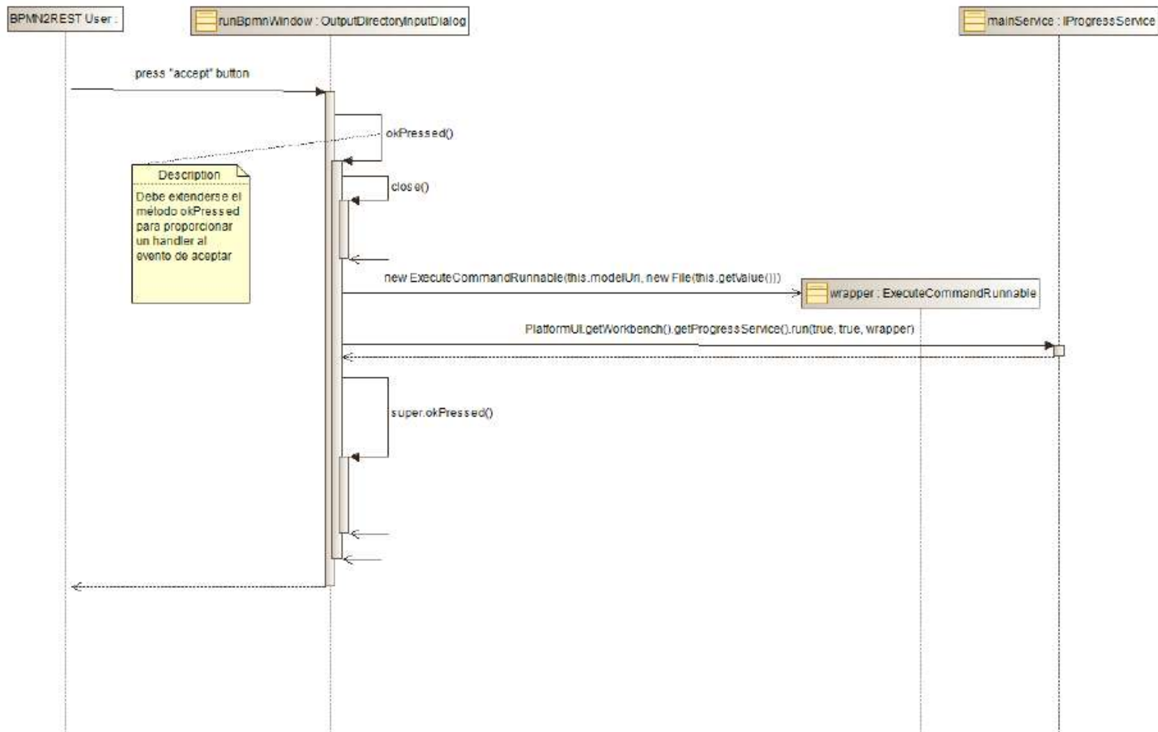


Figura 4.27: Diagrama de secuencia que ilustra el proceso ejecutado al dispararse el evento "aceptar".

Integración del desarrollo en la GUI

La clase *ExecuteCommandRunnable* permite ejecutar la validación del modelo de entrada, y la transformación BPMN2REST sobre el mismo, generando los artefactos de salida. Es conveniente que la ejecución de la transformación sobre el modelo sea llevada a cabo como el resultado de una acción realizada por el usuario del IDE. En dicho caso, la solución más simple e intuitiva indica que BPMN2REST debe ser ejecutada al presionar algún botón visible en la interfaz del usuario.

El punto de extensión *org.eclipse.ui.menus* definido por el *core* de Eclipse IDE, permite añadir opciones adicionales a los distintos menúes disponibles en la interfaz del IDE. Cada opción añadida, se representa por un botón que al ser seleccionado, crea y dispara un evento a ser manipulado por un *handler* o manejador de eventos. Cada *handler* debe implementarse como una clase que extiende a *org.eclipse.core.commands.AbstractHandler*. Para asociar el *handler* definido con el evento que lanza el botón del menú, deben incorporarse adicionalmente los puntos de extensión *org.eclipse.ui.commands* y *org.eclipse.ui.handlers*. El primero, permite asociar un "comando" con el botón del menú, mientras que el segundo, se utiliza para relacionar una clase *handler* con el comando en cuestión (de manera que un mismo *handler* puede ser utilizado por varios comandos distintos). En el anexo A7 se describe la estructura de los puntos de extensión *org.eclipse.ui.menus*, *org.eclipse.ui.commands* y *org.eclipse.ui.handlers*.

El botón se añadió como una opción adicional dentro del menú contextual desplegable existente para un archivo con la extensión *.bpmn*. De esta manera, se puede acceder a la opción (y por lo tanto, ejecutar BPMN2REST) haciendo click derecho sobre el modelo extBPMN listado en la vista de exploración de proyectos (*Project Explorer*) tal como lo ilustra la figura 4.28. La utilización de un menú contextual facilita la obtención del *path* del modelo extBPMN haciendo uso de las utilidades del *core* del IDE, a la vez que impone al usuario la restricción de que sólo se puede ejecutar una transformación en simultáneo, y sólo podrá ser ejecutada sobre un archivo *.bpmn*.

El *handler* para el evento lanzado por la selección de la opción, será definido por el método *execute* de la clase *TransformBpmn2RestHandler* incorporada en el *plugin*. Dentro del mismo, se obtiene el *path* del modelo extBPMN sobre el cual se seleccionó la opción, y se crea una instancia de *OutputDirectoryInputDialog*, permitiendo al usuario indicar el directorio de salida de la transformación BPMN2REST. A partir de la selección del botón de aceptar, iniciará la ejecución de BPMN2REST sobre el modelo de entrada, de acuerdo al uso de la clase *ExecuteCommandRunnable*. El progreso de la transformación será visualizado en la interfaz del usuario, y al finalizar la misma, se notificarán

los resultados obtenidos. La figura 4.29 ilustra la ejecución del *handler* mediante un diagrama de secuencia.

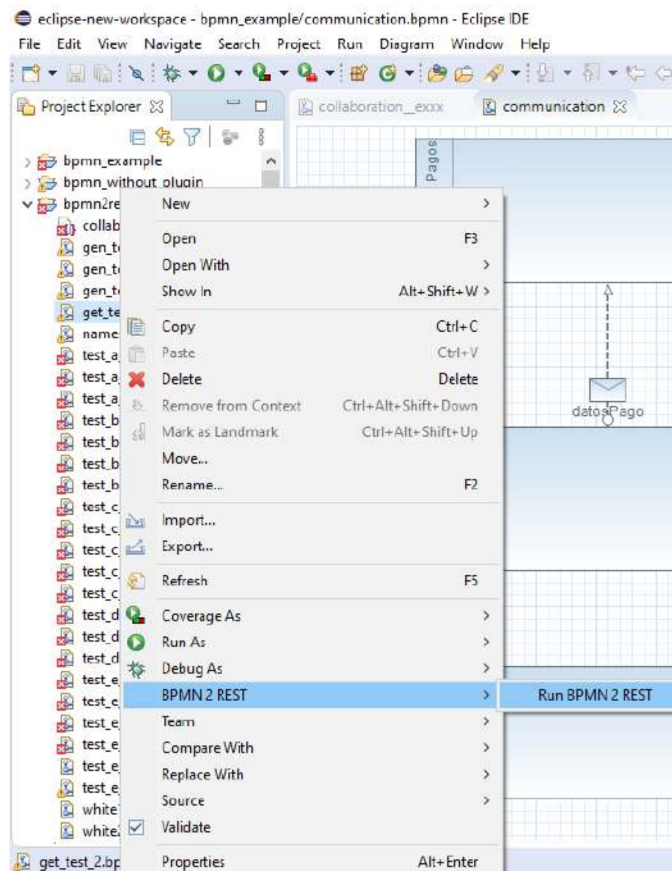


Figura 4.28: Ejecución de BPMN2REST desde la interfaz del usuario de Eclipse IDE. El menú se abre al hacer click derecho sobre un modelo extBPMN, haciéndose visible la opción “Run BPMN 2 REST”.

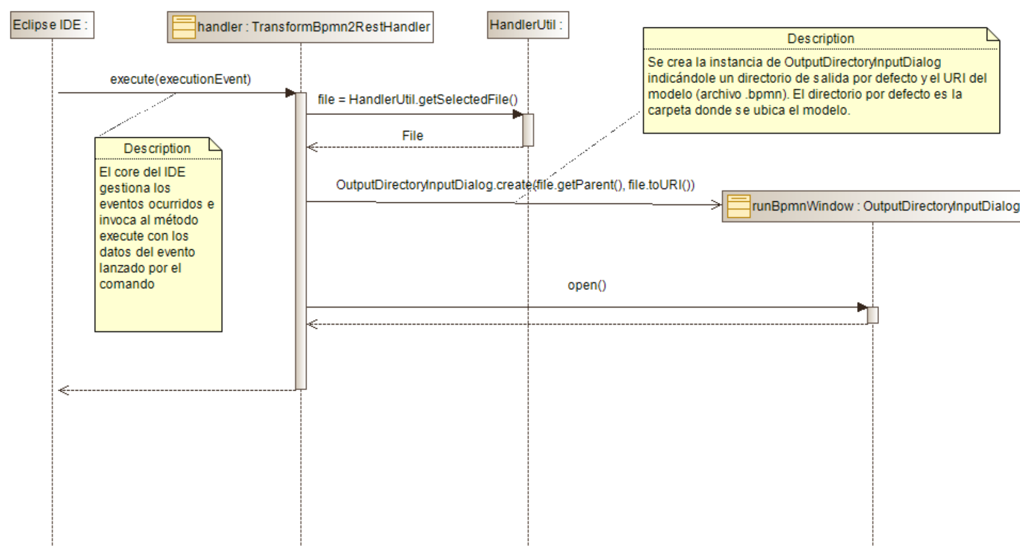


Figura 4.29: Diagrama de secuencia que ilustra las acciones realizadas por el handler asociado al comando “Run BPMN 2 REST”.

Capítulo 5

Caso de estudio

El siguiente capítulo aborda el desarrollo de un caso de estudio cuyo fin es validar la correctitud de los desarrollos e ilustrar los conceptos fundamentales detallados en secciones anteriores. Con el mismo, se comprenderá cómo es el proceso de implementación de arquitecturas SOA bajo la plataforma REST, conforme al uso de la herramienta desarrollada. Además, se resaltarán los beneficios obtenidos a partir de su uso.

5.1. Presentación del caso de estudio

El caso de estudio fue adaptado de [59] y describe el proceso de reserva de un viaje empleando un sistema de reservas. Tres participantes intervienen en el proceso: el sistema cliente, el sistema de reservas y el banco. El sistema cliente es operado directamente por el cliente interesado y generalmente, se compone de un conjunto de interfaces de usuario que brindan transparencia, ocultando la comunicación con los demás sistemas y componentes distribuidos. El sistema de reservas puede a su vez, ser un sub-sistema que forma parte de una arquitectura mayor, o un sistema independiente. La comunicación con el sistema de reservas es totalmente distribuida, con lo cual puede ser visto como un conjunto de interfaces de servicios web. El banco, representa a la entidad bancaria que maneja los fondos del cliente. Puede asumirse que la interacción con este participante es a través de un sistema bancario que provee una API de servicios automatizados. La figura 5.1 muestra un diagrama de colaboraciones BPMN con los detalles del proceso.

Para poder hacer una reserva, el cliente debe identificarse en el sistema de reservas enviando sus credenciales de acceso. Una vez identificado, el cliente solicita algún tipo de información sobre viajes disponibles y recibe una propuesta de itinerario. El cliente decide si aceptar o rechazar dicha propuesta. De acuerdo a su decisión, el sistema de reservas aguarda la respuesta y actúa de acuerdo a la misma. En el caso de un rechazo, el proceso termina. En el caso de confirmar el itinerario, el cliente envía la confirmación y el sistema confirma la reserva, emitiendo una notificación. A partir de allí, el sistema cliente se comunica con el banco para enviar el pago de la reserva. El banco procesa el pago y notifica al sistema de reservas. El proceso finaliza con la emisión y el envío del ticket del viaje al cliente.

A lo largo del proceso, varias interacciones toman lugar entre los distintos participantes. La mayoría de ellas pueden ser resueltas con alguna tecnología de comunicación distribuida, normalmente servicios web empleando WSDL o REST. Para ilustrar de mejor manera el funcionamiento de la transformación BPMN2REST, se asumirá que no todas las comunicaciones entre participantes se encuentran totalmente automatizadas, o que lo hacen, empleando una tecnología distinta a REST. Adicionalmente, el banco fue modelado con una vista de caja negra para mostrar la aplicación de la transformación en vistas de caja negra y de caja blanca.

5.2. Desarrollo del caso de estudio

Como requisito para el correcto desarrollo del caso de estudio, se asume que el entorno Eclipse BPMN2 Modeler y el complemento BPMN2REST para Eclipse IDE han sido instalados correctamente. El anexo A9 presenta una guía con los pasos de instalación que deben seguirse.

Creación del proyecto en Eclipse

El caso de estudio debe ser expresado por medio de un modelo de procesos de negocio en BPMN2 Modeler. Este modelo, debe existir dentro de un proyecto en Eclipse IDE, a fin de que pueda ser aplicado el complemento BPMN2REST. Primero, debe crearse un proyecto general en Eclipse. Para ello, se seleccionará la opción *Project* dentro del tipo de proyecto *General*, mostrado en la interfaz de selección de tipos de proyectos (ver figura 5.2). Para acceder a esta interfaz, debe seleccionarse la opción *File > New > Project*.

Luego de creado el nuevo proyecto, se debe asociar el complemento BPMN2REST al mismo. Esto permitirá que los diagramas construidos con BPMN2 Modeler dentro del proyecto, apliquen las características de la extensión BPMN2REST. Para hacer esto, debe abrirse la vista de exploración de proyectos (*Project Explorer*) y desde allí, ingresar a la ventana de configuración de propiedades del proyecto, haciendo click derecho (sobre el proyecto), y luego seleccionando *Properties*, como se muestra en la figura 5.3.

Desde la ventana de propiedades, seleccionando la pestaña *BPMN2*, se puede configurar al proyecto para que utilice el complemento BPMN2REST. Para ello, debe seleccionarse la opción *BPMN2REST Extension* en el desplegable que se muestra para la configuración del *Target Runtime*. La figura 5.4 muestra la ventana de propiedades de un proyecto, y la selección del complemento BPMN2REST.

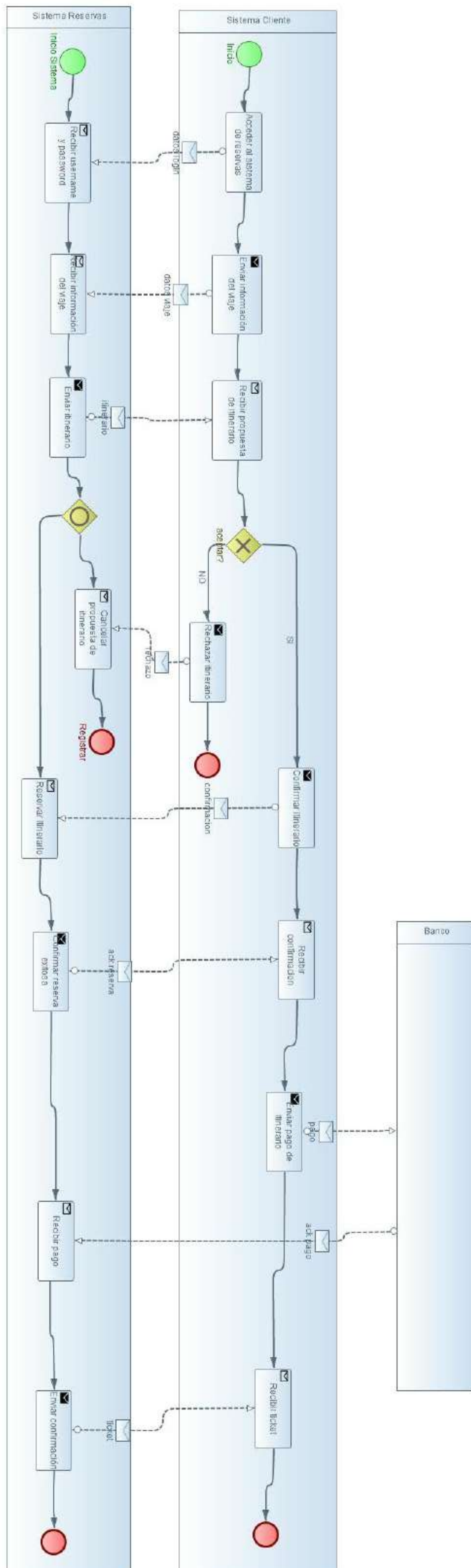


Figura 5.1: Diagrama de colaboraciones BPMN que detalla el proceso de reserva de viaje. El diagrama fue construido con BPMN2 Modeler.

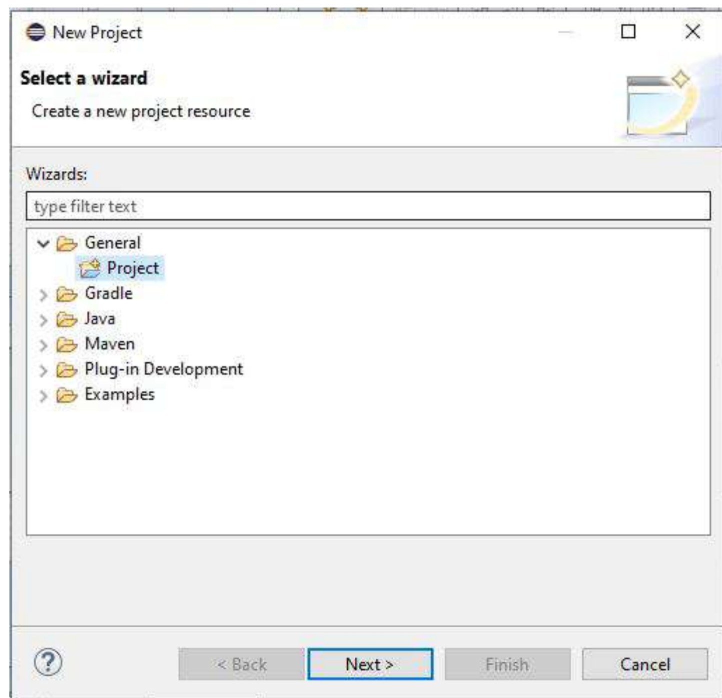


Figura 5.2: Interfaz de selección de tipos de proyectos en Eclipse IDE.

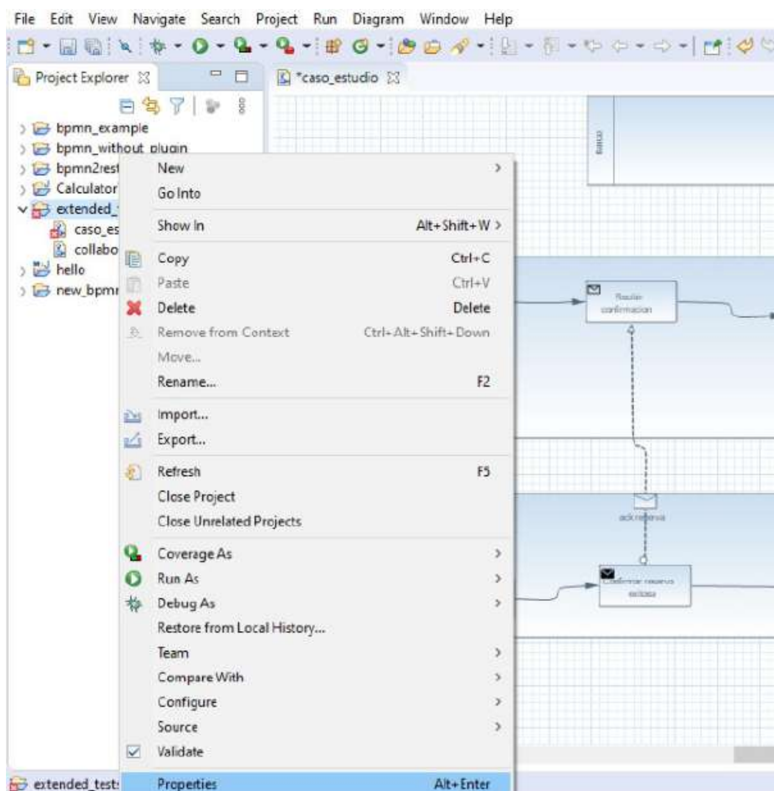


Figura 5.3: Menú contextual abierto para un proyecto desde la vista de Project Explorer. Desde aquí se puede ingresar a la configuración de propiedades de un proyecto.

Modelado del proceso de negocios

Luego de creado el proyecto en Eclipse, podrá modelarse el proceso de negocios creando un nuevo diagrama de colaboraciones BPMN con Eclipse BPMN2 Modeler, tal como se detalla en la sección 2.5.5. El proceso de negocios debe ser modelado siguiendo las pautas para la construcción de modelos BPMN. Se utiliza un diagrama de colaboraciones BPMN, dado que los diagramas de proceso no permiten indicar las comunicaciones entre los participantes. La figura 5.1 presentada anteriormente, muestra el diagrama de colaboraciones BPMN con el detalle del proceso de negocios y las interacciones entre los participantes.

Diseño de los servicios REST

A partir del modelo de procesos de negocio (plasmado mediante un diagrama de colaboraciones BPMN), la siguiente actividad comprende el diseño de las comunicaciones REST. En primera instancia, deben determinarse qué interacciones entre participantes van a ser soportadas mediante servicios web RESTful. La tabla 5.1 muestra los *MessageFlows* del diagrama que serán convertidos a servicios REST. Para cada uno de ellos, se detalla su proveedor y cliente, el elemento *Message* involucrado, y el nombre

representativo elegido para el servicio asociado.

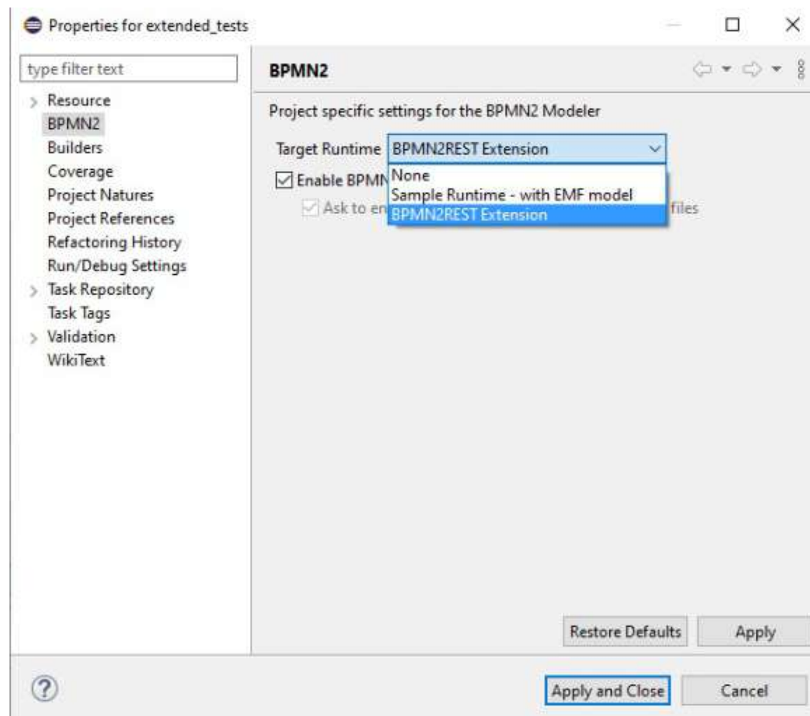


Figura 5.4: Ventana de configuración de propiedades de un proyecto en Eclipse IDE.

Nombre del servicio	Proveedor	Cliente	Mensaje	Detalle del servicio
loginCliente	Sistema Reservas	Sistema Cliente	datos login	Identifica al usuario en el sistema de reservas
obtenerPropuesta	Sistema Reservas	Sistema Cliente	datos viaje	Solicita una propuesta de viaje al sistema de reservas
rechazarPropuesta	Sistema Reservas	Sistema Cliente	rechazo	Cancela la propuesta de itinerario y la elimina
hacerReserva	Sistema Reservas	Sistema Cliente	confirmación	Confirma la reserva para el itinerario
transferir	Banco	Sistema Cliente	pago	Recibe los datos del pago del cliente y realiza una transferencia al sistema de reservas
notificarPago	Sistema Reservas	Banco	ack pago	Notifica al sistema de reservas la confirmación del pago del itinerario
recibirTicket	Sistema Cliente	Sistema Reservas	ticket	Envía los datos del ticket para el viaje del cliente

Tabla 5.1: Colaboraciones BPMN que serán implementadas como servicios web RESTful. La columna Proveedor identifica al participante que provee el servicio y que, por lo tanto, recibirá una solicitud HTTP con los datos necesarios para poder brindarlo. La columna Cliente identifica al participante que requiere la ejecución del servicio.

Cada servicio mencionado en la tabla 5.1 será implementado como un servicio RESTful que provee alguno de los participantes del modelo BPMN. Para cada uno de ellos, es necesario definir la URL y el método HTTP de acceso, que darán lugar a su interfaz REST. La tabla 5.2 muestra los datos de la interfaz de cada uno de los servicios.

Nombre del servicio	URL	Método HTTP
loginCliente	/login	POST
obtenerPropuesta	/itinerarios	GET
rechazarPropuesta	/itinerarios/:id	DELETE
hacerReserva	/itinerarios/:id	POST
transferir	/transferencias/enviar	POST
notificarPago	/itinerarios/:id/pagar	POST
recibirTicket	/viajes	POST

Tabla 5.2: Listado de valores para la URL y el método HTTP de acceso a los distintos servicios REST.

Diseño del modelo extBPMN

En base al diagrama de colaboraciones BPMN y a la definición de los servicios REST, la siguiente etapa comprende la construcción del modelo extBPMN utilizando la herramienta BPMN2 Modeler. En síntesis, las tareas abordadas implican definir las interfaces de cada participante BPMN en términos de los servicios REST que provee, utilizando los elementos nativos de la notación BPMN y los que introduce extBPMN.

Dado que los participantes presentan interfaces muy distintas unas de otras, los servicios definidos no son reutilizados a lo largo del modelo. Esto puede apreciarse en la definición de un *Message* distinto para cada flujo, y un servicio distinto para cada comunicación BPMN. En base a esta cualidad, la definición de las interfaces de cada participante resulta una actividad directa que puede derivarse de observar las tablas 5.1 y 5.2. Así, las interfaces identificadas para cada participante son:

- Interfaz *ServiciosCliente*

- Servicio: *recibirTicket*. Entrada: *ticket*. Método: *POST*. URL: */viajes*.

- Interfaz *ServiciosReserva*
  - Servicio: *loginCliente*. Entrada: *datos login*. Método: *POST*. URL: */login*.
  - Servicio: *obtenerPropuesta*. Entrada: *datos viaje*. Método: *GET*. URL: */itinerarios*.
  - Servicio: *rechazarPropuesta*. Entrada: *rechazo*. Método: *DELETE*. URL: */itinerarios/:id*.
  - Servicio: *hacerReserva*. Entrada: *confirmación*. Método: *POST*. URL: */itinerarios/:id*.
  - Servicio: *notificarPago*. Entrada: *ack pago*. Método: *POST*. URL: */itinerarios/:id/pagar*.

- Interfaz *ServiciosBanco*
  - Servicio: *transferir*. Entrada: *pago*. Método: *POST*. URL: */transferencias/enviar*.

Para cada una de estas interfaces, debe construirse una interfaz BPMN utilizando el elemento *Interface*. Para crear una nueva interfaz en el modelo, se debe acceder a la ventana de configuración del elemento *Collaboration* haciendo doble click sobre el mismo, en la vista de *Outline*. La ventana de configuración de una colaboración presenta distintas pestañas con opciones. La pestaña *Interfaces* permitirá añadir nuevas interfaces al modelo. Haciendo click en el botón *Add* (señalado con un símbolo de suma), la interfaz cambiará su apariencia, mostrando nuevos campos de configuración para los elementos *Interface* añadidos. La figura 5.5 muestra la ventana de configuración de colaboraciones, mientras que la figura 5.6 muestra la ventana de configuración de una nueva interfaz BPMN. El campo *Interface Name* es utilizado para asignar un nombre a la interfaz de servicios.

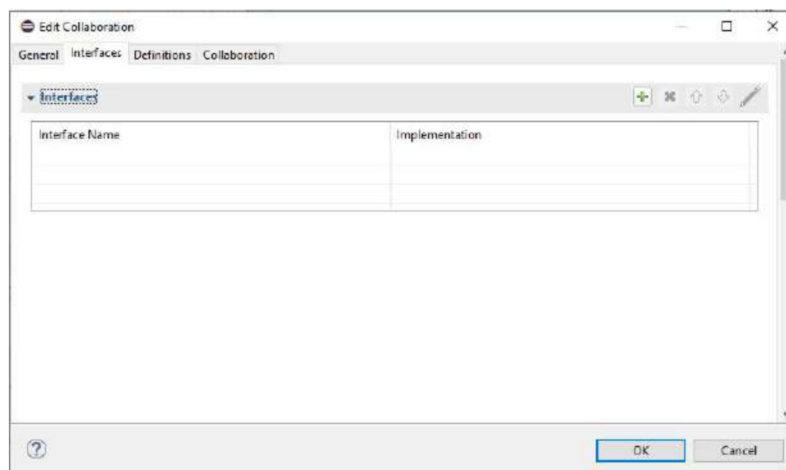


Figura 5.5: Ventana de configuración del elemento Collaboration en BPMN2 Modeler.

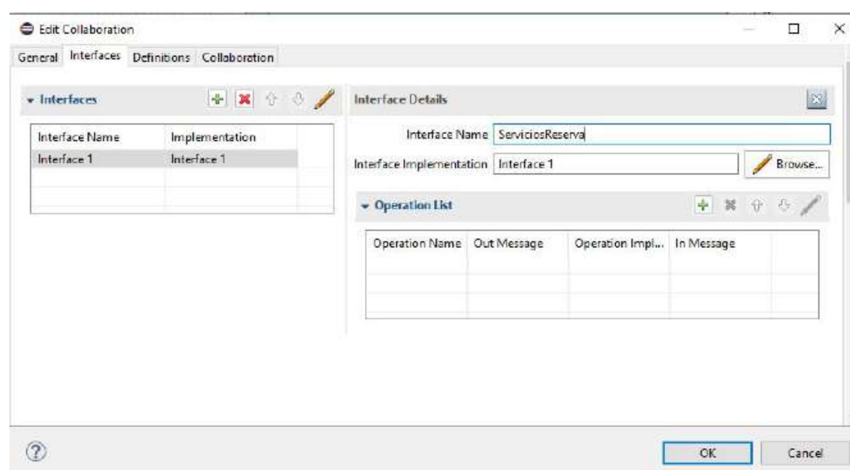


Figura 5.6: Configuración de elementos Interface en BPMN2 Modeler, dentro de la ventana de configuración de colaboraciones.

Una vez añadidas las interfaces, podrán ser seleccionadas desde la vista de *Outline*, como lo muestra la figura 5.7. Haciendo doble click sobre una interfaz en esta vista, se desplegará la ventana de configuración de elementos *Interface*. Desde aquí, se podrán configurar los servicios de una interfaz, añadiendo operaciones BPMN, desde la pestaña *Interface*. La figura 5.8 muestra la ventana de configuración de una interfaz. Se puede añadir una nueva operación haciendo click en *Add* (señalado con un símbolo de suma). Esta acción cambiará la apariencia de la ventana, añadiendo campos de configuración para las operaciones agregadas a la interfaz. La figura 5.9 muestra la interfaz de configuración de una nueva operación añadida. Desde esta interfaz, se puede asignar un nombre a la operación utilizando el campo *Operation Name*, y asociar el mensaje de entrada correspondiente, desde el campo *In Message*.

Figura 5.7: Vista de Outline que muestra los elementos BPMN del modelo.

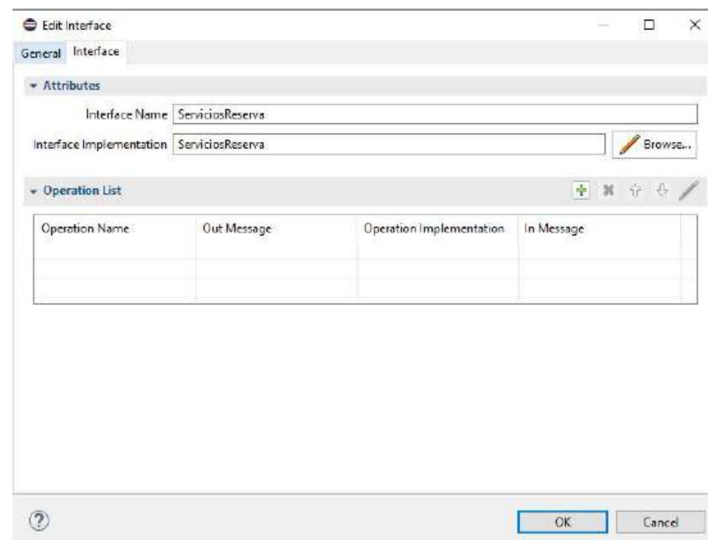


Figura 5.8: Ventana de configuración de una interfaz BPMN en BPMN2 Modeler.

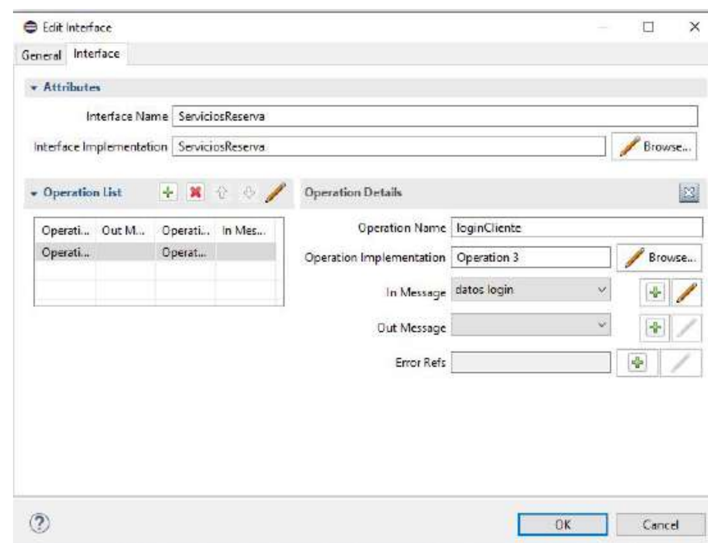


Figura 5.9: Configuración de elementos Operation en BPMN2 Modeler, dentro de la ventana de configuración de interfaces.

Una operación representa un servicio REST que brinda un participante a los demás, según su interfaz de servicios. Para definir el servicio en la arquitectura REST, deben definirse el método HTTP y la URL del mismo. Luego de añadir una nueva operación a una interfaz, la misma es listada en la vista de *Outline* junto a la interfaz que la engloba, como muestra la figura 5.10. Haciendo doble click sobre el elemento *Operation*, se abrirá la ventana de configuración de la operación. Desde aquí, podrán configurarse los valores REST de la operación, ingresando a la pestaña denominada *BPMN2REST Extension*. La figura 5.11 muestra la pestaña de configuración de las propiedades REST, dentro de la ventana de configuración de la operación.

Para el presente caso de estudio, no es necesario tildar el campo *Add Controller Preffix to URL* en ninguno de los servicios, ya que, como se mencionó antes, cada operación es única de un participante y por ello las interfaces no son compartidas.

Para cada uno de los servicios REST, deberán seguirse estos pasos para definir la interfaz que los engloba y su representación, utilizando elementos *Operation*. Una vez estén definidas todas las interfaces y sus respectivos servicios, debe asociarse cada interfaz al participante que la provee. Para ello, es necesario acceder a la ventana de configuración de las propiedades de un participante

(*Participant*). La ventana puede abrirse haciendo doble click sobre el elemento *Participant* listado en la vista de *Outline*, o en el *Pool* que lo representa, dentro de la interfaz de edición gráfica. La ventana de configuración se muestra en la figura 5.12. Desde la pestaña *Participant*, es posible asociarle una interfaz existente al participante. Esto se hace seleccionando el botón *Add* (representado con un símbolo de suma), en la sección *Interfaces Provided by Participant*, y luego eligiendo la interfaz de servicios que brinda el participante en cuestión.

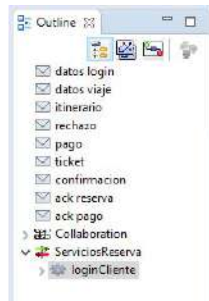


Figura 5.10: Vista de Outline mostrando las operaciones añadidas a una interfaz.

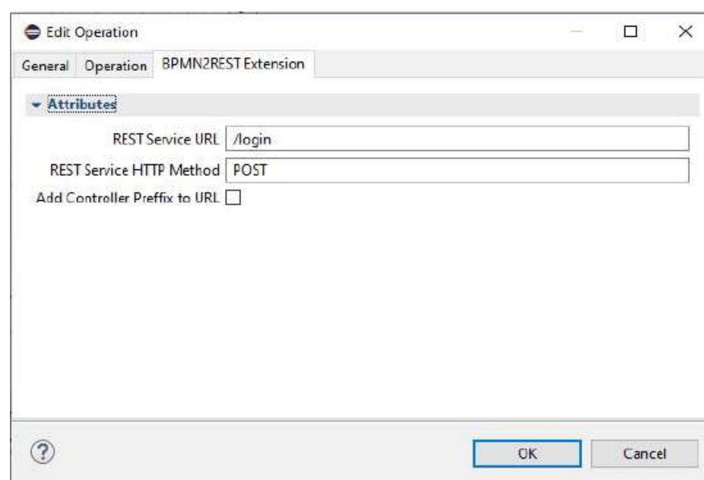


Figura 5.11: Configuración de las propiedades REST de un elemento Operation, conforme a las propiedades extBPMN.

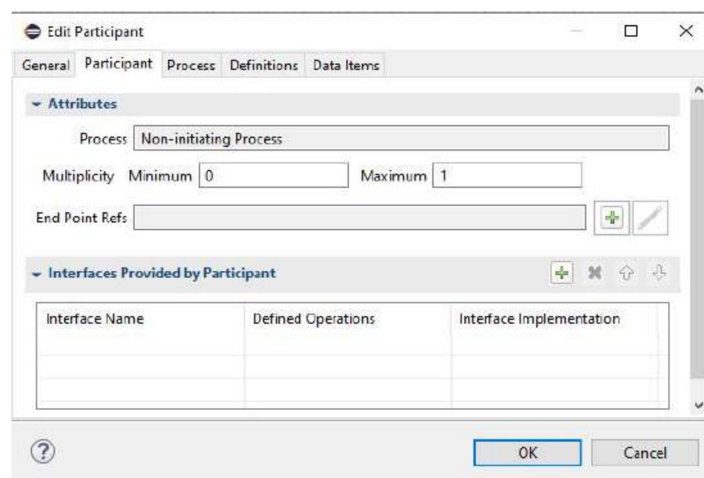


Figura 5.12: Ventana de configuración de las propiedades de un participante en BPMN2 Modeler. Se muestra la pestaña Participant.

Ejecución de BPMN2REST

Una vez finalizada la confección del modelo extBPMN; la siguiente etapa comprende la ejecución de BPMN2REST desde el entorno Eclipse IDE para obtener los *stubs* en código fuente de los servicios. Antes de ejecutar la transformación, todos los cambios realizados al modelo deben ser guardados.

Desde la vista de exploración de proyectos (*Project Explorer*), se muestran los distintos artefactos del proyecto iniciado con Eclipse. Haciendo click derecho sobre la colaboración BPMN, se listarán las opciones contextuales de la vista, como muestra la figura 5.13. Desde allí, seleccionando *BPMN 2 REST* > *Run BPMN 2 REST*, se abrirá la interfaz de selección del directorio de salida, mostrado en la figura 5.14. Al introducir un *path* hacia un directorio válido de salida y seleccionar *OK*, la transformación BPMN2REST será ejecutada, notificando su correcta finalización, o un mensaje de error en caso contrario.

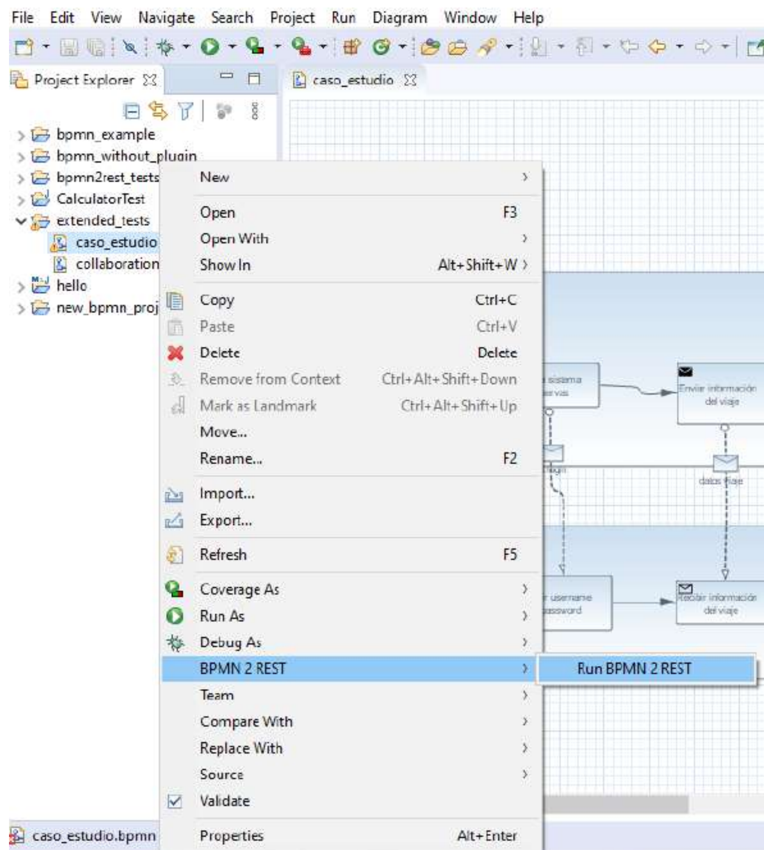


Figura 5.13: Opciones contextuales desplegadas desde la vista de exploración de proyectos al hacer click derecho sobre un diagrama BPMN.

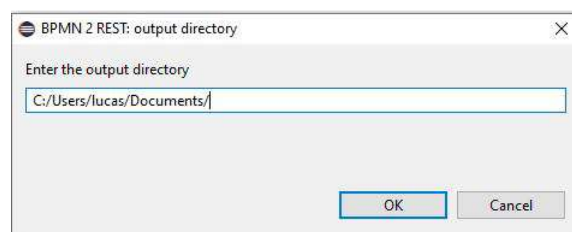


Figura 5.14: Interfaz de selección del directorio de salida para la transformación BPMN2REST.

Como resultado de la ejecución de BPMN2REST, la siguiente estructura de archivos es generada en el directorio de salida especificado:

- *Main.java* (clase Main)
- *controllers* (directorio)
  - *BancoController.java* (clase controlador)
  - *SistemaReservasController.java* (clase controlador)
  - *SistemaClienteController.java* (clase controlador)

La figura 5.15 muestra el código de la clase *Main.java*, generado por BPMN2REST, mientras que la figura 5.16 muestra un fragmento de la clase *SistemaReservasController.java* también generada por la transformación.

Normalmente, los desarrolladores de los servicios REST del banco, estarán interesados en implementar sólo los servicios de su empresa, por lo que modelarán sólo la interfaz *ServiciosBanco* sin darle importancia a las demás. Esto les permitirá obtener el código Java de los servicios que les interesa construir. Este mismo razonamiento puede aplicarse con los demás participantes del modelo, de manera que un mismo modelo BPMN puede implicar la construcción de distintos modelos extBPMN, según sea la organización que desee desarrollar los servicios REST.

Refinamiento y despliegue de los servicios

Los archivos obtenidos con la ejecución de BPMN2REST, pueden ser utilizados como la base de un proyecto en Java Spark Framework. Una vez iniciado un nuevo proyecto en Spark con las dependencias correspondientes, las clases generadas previamente pueden ser importadas accediendo a la opción *File > Import* en Eclipse IDE.

```java
import static spark.Spark.*;
import controllers.*;

public class Main {

    /**
     * service route definitions using Spark
     *
     * @param  String[] args
     * @return void
     */
    public static void main(String[] args) {

        /* Sistema Cliente exposed services */

        post("/viajes", SistemaClienteController::recibirTicket);

        /* Banco exposed services */

        post("/transferencias/enviar", BancoController::transferir);

        /* Sistema Reservas exposed services */

        post("/itinerarios/:id/pagar", SistemaReservasController::notificarPago);

        post("/itinerarios/:id", SistemaReservasController::hacerReserva);

        post("/login", SistemaReservasController::loginCliente);

        get("/itinerarios", SistemaReservasController::obtenerPropuesta);

        delete("/itinerarios/:id", SistemaReservasController::rechazarPropuesta);
    }

}
```

Figura 5.15: Código Java de la clase Main generado por BPMN2REST.

```java
package controllers;

import spark.Request;
import spark.Response;

public class SistemaReservasController {

    /**
     * controller method for handling obtenerPropuesta operation
     *
     * @param  req Request
     * @param  res Response
     * @return     String
     */
    public static String obtenerPropuesta(Request req, Response res) {
        return "Example Response"; // replace this with the service implementation
    }

    /**
     * controller method for handling hacerReserva operation
     *
     * @param  req Request
     * @param  res Response
     * @return     String
     */
    public static String hacerReserva(Request req, Response res) {
        return "Example Response"; // replace this with the service implementation
    }

    /**
     * controller method for handling notificarPago operation
     *
     * @param  req Request
     * @param  res Response
     * @return     String
     */
    public static String notificarPago(Request req, Response res) {
        return "Example Response"; // replace this with the service implementation
    }
```

Figura 5.16: Fragmento de la clase Java SistemaReservasController generada por BPMN2REST.

La importación de la clase *Main* y de cada uno de los controladores al proyecto en Spark, reduce el trabajo de los desarrolladores a la implementación de cada servicio REST utilizando las clases *Request* y *Response*. Esta tarea es realizada manualmente por los desarrolladores de los servicios y recibe el nombre de refinamiento. A partir de la implementación de los servicios REST, el despliegue de los mismos es una actividad sencilla llevada a cabo por Spark. El código a continuación muestra un ejemplo de refinamiento para el servicio *SistemaReservasController.hacerReserva*, junto al resultado

de su ejecución, mostrado en la figura 5.17.

```
/**
 * controller method for handling hacerReserva operation
 *
 * @param  req Request
 * @param  res Response
 * @return     String
 */
public static String hacerReserva(Request req, Response res)
{
    return "Ud. ha reservado el itinerario + req.params(":id");
}
```

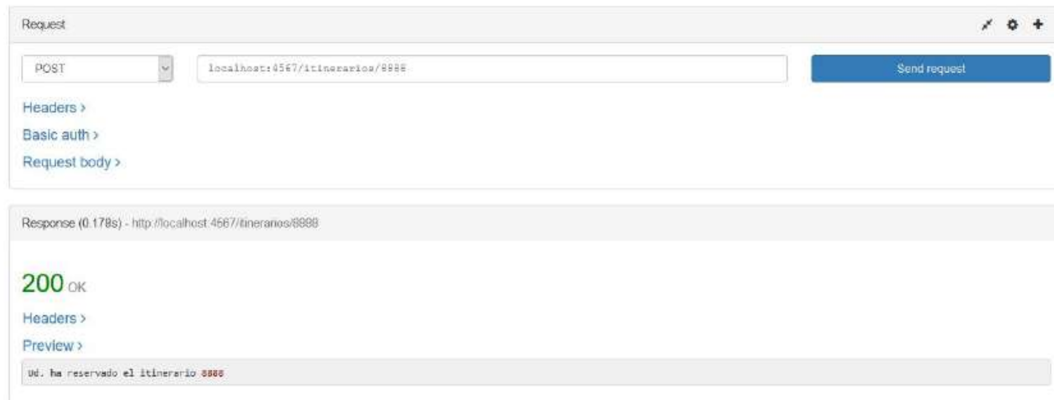


Figura 5.17: Resultado de ejecución del servicio hacerReserva. Se utilizó el complemento RESTED Client de Mozilla Firefox, para realizar la petición HTTP POST.

Capítulo 6

Conclusiones y consideraciones finales

El siguiente capítulo introduce las conclusiones y los resultados obtenidos tras la realización de este trabajo de tesis. Primero, se exponen los resultados obtenidos considerando los objetivos propuestos para la tesis. Luego, se presentan las conclusiones finales derivadas del desarrollo del trabajo. Finalmente, se listan las líneas de trabajo futuro que se prevén tras la realización de este trabajo.

6.1. Cumplimiento de los objetivos

Este trabajo de tesis presentó los siguientes objetivos específicos a ser alcanzados:

1. *Comprender el Estado del Arte respecto a propuestas existentes para generar definiciones de servicios web RESTful, a partir de colaboraciones en los modelos de negocios.*
2. *Estudiar el marco de trabajo Java Spark Framework para el desarrollo de aplicaciones web basadas en microservicios y el entorno de desarrollo propuesto por Eclipse IDE*
3. *Diseñar y desarrollar la transformación de BPMN a servicios web RESTful, denominada BPMN2REST.*
4. *Validar los modelos de procesos de negocio BPMN de entrada conforme a los datos mínimos necesarios para generar la implementación de los servicios en Spark.*
5. *Enriquecer la transformación, permitiéndole al usuario indicar datos auxiliares para cada uno de los servicios web tales como la URL o el método HTTP empleado, brindando mayor soporte en la configuración del código fuente generado.*
6. *Desarrollar una extensión para Eclipse IDE que permita hacer uso de la transformación BPMN2REST desde dicha herramienta.*
7. *Validar los desarrollos mediante su aplicación en un caso de estudio concreto.*

El objetivo 1 fue alcanzado totalmente y documentado en el capítulo 3 de la tesis. Se indagaron trabajos de investigación e informes técnicos que mostraban la aplicación de técnicas de MDD sobre el conjunto tecnológico conformado por BPMN, SOA y REST, y que estaban alineados con este trabajo. La revisión del estado del arte permitió tener un panorama general de las contribuciones existentes con respecto a los conceptos abordados, y facilitó visualizar el alcance de esta tesis ilustrando las tareas comunes que deben realizarse para aplicar MDD efectivamente.

El estudio de Java Spark Framework para el desarrollo de microservicios REST fue necesario para conocer la estructura de las clases Java que impone el marco de trabajo, y así poder definir la transformación BPMN2REST correctamente. Por su parte, Eclipse IDE fue estudiado en profundidad no sólo para ser utilizado como entorno de desarrollo, sino también para ser extendido mediante la construcción de *plugins*. De esta manera, el objetivo 2 fue alcanzado satisfactoriamente. El capítulo 2 presenta las bases fundamentales sobre Java Spark Framework y el entorno Eclipse IDE necesarias para comprender el desarrollo técnico de la tesis.

El capítulo 4 comienza con el diseño de la transformación BPMN2REST, primero, empleando lenguaje natural y luego, formalizando la transformación en base a la extensión extBPMN. Posteriormente, se describió la implementación de BPMN2REST utilizando Acceleo y la herramienta de modelado BPMN2 Modeler. Así, los objetivos 3 y 5 se lograron efectivamente. El mecanismo de validación de BPMN2 Modeler fue adoptado para validar los modelos de entrada a la transformación, garantizando que sólo se ejecuta sobre modelos válidos, satisfaciendo así, el objetivo 4. Por último, los desarrollos fueron integrados en la construcción de un *plugin* que permitiera utilizarlos desde Eclipse IDE mediante un único proceso de instalación; lo cual permitió alcanzar el objetivo 6.

Finalmente, el capítulo 5 mostró el uso del *plugin* y de la transformación en un caso de estudio concreto, mostrando los beneficios de su uso e ilustrando el aporte de la herramienta en un contexto de aplicación real; cumpliendo con el objetivo 7.

El cumplimiento de cada uno de los objetivos específicos del trabajo, muestra el alcance del objetivo general propuesto; a saber: *desarrollar una transformación MDD que permita obtener la implementación parcial en Java de los servicios web RESTful presentes en los modelos de procesos de negocio BPMN; enfocando tal desarrollo en la construcción de una extensión para Eclipse IDE.*

6.2. Conclusiones finales

La metodología dirigida por modelos se presenta como un área temática muy prometedora, que comprende prácticas de ingeniería de software de gran interés. Cada vez más herramientas de desarrollo de software proveen características que brindan algún tipo de asistencia en la construcción

de modelos u otros artefactos, de manera automática o semi-automática. Estas herramientas, comúnmente conocidas como CASE, emplean prácticas de desarrollo dirigido por modelos para dar soporte a los desarrolladores, absorbiendo tareas repetitivas e incrementando la productividad. MDD comprende un área de ingeniería en pleno desarrollo, que avanza continuamente y que, actualmente, puede verse en aplicación en los entornos y herramientas más comunes utilizadas en la producción de software.

El metamodelado constituye uno de los contenidos básicos para el correcto desarrollo de transformaciones MDD. El presente trabajo permitió conocer en profundidad el metamodelo de BPMN, necesario para poder desarrollar la transformación BPMN2REST y para especificar las reglas de validación. Asimismo, la construcción de extBPMN puede ser considerada como una extensión del metamodelo de BPMN para los elementos *Operation*.

La extensión extBPMN permite especificar los valores REST de cada servicio que una organización provee a otra/s. Un modelo BPMN se ubica a nivel CIM y presenta un mayor nivel de abstracción que un modelo extBPMN. Desde un modelo BPMN pueden derivarse múltiples modelos extBPMN. Dado que extBPMN detalla que los servicios serán implementados con la arquitectura REST, puede afirmarse que un modelo extBPMN pertenece al nivel PSM. BPMN2REST interpreta a los modelos extBPMN como la especificación de una arquitectura SOA implementada en REST, y genera el código Java de cada uno de los servicios definidos en la misma.

El trabajo estimuló el aprendizaje de nuevos contenidos no abordados previamente. Entre ellos se destacan MDD; el desarrollo de software para Eclipse IDE; el uso de los *plugins* BPMN2 Modeler y Acceleo; y la arquitectura de Java Spark Framework. En adición, resultó necesario repasar temáticas que sí fueron dictadas en la carrera, de manera mucho más profunda y detallada. Este segundo grupo comprende contenidos como BPMN y procesos de negocio; SOA y REST; OCL; XML; y el lenguaje de programación Java.

La utilización de la plataforma Eclipse como entorno de desarrollo, y el uso de las características BPMN2 Modeler y Acceleo; permitieron comprender y experimentar el nivel de soporte que proveen dichas herramientas a sus usuarios, y los beneficios de su uso. En este aspecto, cabe destacar que las herramientas son altamente extensibles y proveen varias utilidades a sus usuarios. Adicionalmente, Eclipse cuenta con una comunidad activa que continuamente se encuentra reportando problemas o dando soluciones a los mismos. Los aportes de la comunidad mediante la publicación de blogs, así como la resolución de conflictos en *Stack Overflow*[1] fueron de gran ayuda en el uso de las herramientas.

En contraste, el desarrollo de complementos para Eclipse resultó ser una tarea compleja, dado que los errores y problemas surgidos al utilizar el *core* de Eclipse o el de alguno de sus *plugins*; no contaban con un nivel adecuado de documentación y soporte. En este caso, las soluciones fueron encontradas luego comparar el código fuente con otros desarrollos similares para Eclipse, o a partir de una profunda indagación en blogs que reportaban problemas semejantes. En tal aspecto, debe mencionarse que la documentación para los usuarios desarrolladores de *plugins* no cuenta con un buen nivel de detalle, y presenta pocos ejemplos de uso.

En base a las dependencias entre los *plugins* y los componentes utilizados en el desarrollo, puede estructurarse la arquitectura del complemento BPMN2REST para Eclipse IDE en tres capas, como se muestra en la figura 6.1. Una primera capa se conforma por el *core* de Eclipse, que provee las clases Java, los puntos de extensión y demás utilidades necesarias para integrar un desarrollo en el IDE. En segundo lugar, pueden ubicarse los *plugins* base utilizados para el desarrollo de BPMN2REST: Eclipse BPMN2 Modeler y Acceleo. Ambos aportan las clases Java y los puntos de extensión necesarios para dar soporte a extBPMN y definir la transformación. Por último, la tercera capa contiene a la característica BPMN2REST desarrollada para Eclipse. La misma, se estructura como una característica compuesta por dos *plugins*, como se mencionó en el capítulo 4. El *plugin* *org.eclipse.contributions.bpmn2rest.acceleo.module* define la transformación en base a módulos *.mtl* y la encapsula generando el motor de la transformación. Este motor, es utilizado por el *plugin* *org.eclipse.contributions.bpmn2rest.acceleo.module.ui*, que permite integrar la transformación al IDE, utilizando el *core* de Eclipse y extendiendo el complemento BPMN2 Modeler.

El código fuente generado por BPMN2REST se corresponde con la implementación de los servicios RESTful en Java Spark Framework. Ésta arquitectura fue elegida en base a su alta afinidad con las aplicaciones basadas en microservicios REST. No obstante, es posible adaptar o extender la transformación para que genere el código fuente utilizando cualquier otra plataforma de implementación.

La utilización de BPMN2REST implica una ganancia en términos de productividad por parte de los desarrolladores de software: el uso de la herramienta permite acelerar la codificación de los servicios, y ayuda a evitar los errores por fatiga, absorbiendo las tareas repetitivas. Por otro lado, la transformación asigna un rol central a los modelos de procesos de negocio, donde la interfaz de cada servicio es obtenida desde un modelo extBPMN que la define. Esto brinda una vista consistente entre la implementación de cada servicio y los modelos del negocio; acercando a los desarrolladores de software con los analistas del negocio.

Finalmente, este trabajo destaca la importancia para las organizaciones de modelar sus procesos

[1] https://stackoverflow.com

de negocio, identificando las colaboraciones existentes entre ellas y otras entidades externas. Esta actividad, no sólo permite que los procesos organizacionales estén debidamente documentados, sino que también ayuda a identificar vulnerabilidades en los flujos de trabajo, permitiendo mejorar la eficiencia con la que se realizan las tareas y aumentar la calidad de los resultados obtenidos. En este sentido, la contribución principal del trabajo fue la generación automática de servicios web que den soporte a las colaboraciones identificadas en un proceso de negocio. Así, el desarrollo presentado en esta tesis pretende sumar una utilidad adicional en la tarea de automatización de procesos de negocio.

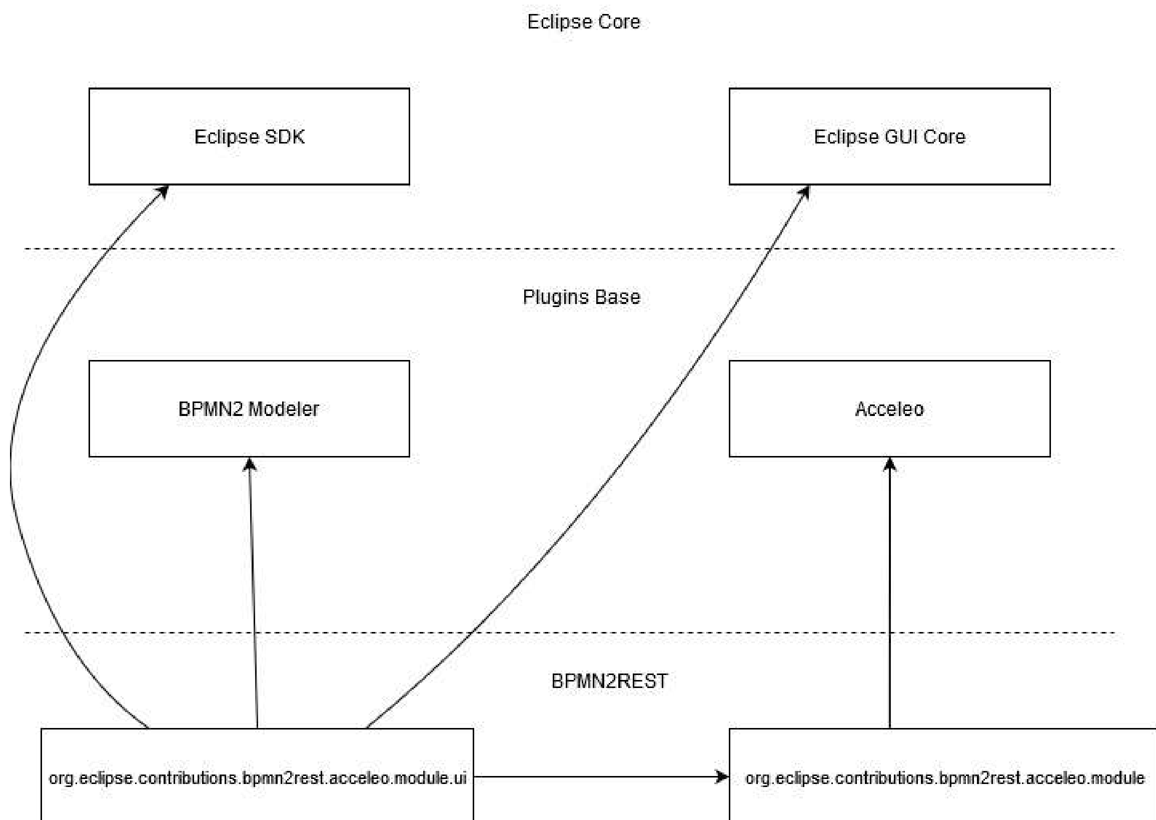


Figura 6.1: Esquema de dependencias utilizadas por los componentes de la característica BPMN2REST para Eclipse IDE.

El trabajo fue puesto a disposición de los interesados en un repositorio de GitHub[2], permitiendo así su libre utilización y modificación según se necesite.

6.3. Trabajo futuro

A continuación, se listan algunas de las líneas de trabajo futuro y proyecciones que surgen considerando este trabajo como base de sus desarrollos:

- Extender el trabajo permitiendo indicar datos adicionales para cada servicio REST en los modelos extBPMN, con el fin de brindar un mayor soporte en el código fuente generado. Algunos datos que podrían indicarse son: el tipo MIME de la respuesta HTTP, las restricciones en los valores de la solicitud HTTP, el recurso REST que será accedido por el servicio, el código de la respuesta HTTP que devolverá, entre otros.
- Extender la transformación brindando soporte para generar los servicios REST en otros *frameworks* o lenguajes de programación distintos, que se utilicen en el desarrollo de APIs REST.
- Extender el trabajo permitiendo generar artefactos o componentes adicionales de manera automática, que puedan ser utilizados en el desarrollo de los microservicios. Se podría generar la documentación formal de la API, la especificación de la representación de cada recurso REST (utilizando JSONSchema o XMLSchema, por ejemplo), el modelo de persistencia, el POM del proyecto en Spark, entre otros.
- Extender el trabajo permitiendo generar la implementación en SOAP de los servicios web denotados en los modelos extBPMN.
- Ampliar la transformación BPMN2REST para que pueda generar el código Java de los servicios REST a partir de coreografías en BPMN, o para que pueda utilizar otros tipos de modelos como punto de partida (UML, SoaML, RAML, por ejemplo).

[2]https://github.com/perlucas/bpmn2rest

Para finalizar, se está trabajando en la redacción de un *paper* con los desarrollos abordados en esta tesis, a ser enviado para su revisión al Congreso Nacional de Ingeniería Informática - Sistemas de Información (CONAIISI), edición 2020.

Anexos

A1. Elementos gráficos de la notación BPMN

BPMN 2.0 define múltiples tipos de elementos gráficos que pueden ser utilizados en los modelos y que permiten representar distintos escenarios. A continuación, se exponen los elementos comúnmente más utilizados en los modelos de procesos de negocio. Una referencia mucho más completa y detallada de BPMN 2.0 se presenta en [2].

EVENTOS

Un evento (*event*) representa un hecho que sucede durante la ejecución de un proceso o una coreografía. Los eventos afectan los flujos de ejecución y normalmente, tienen una causa que los origina o un resultado que arrojan. Existen tres tipos básicos de evento, que se diferencian de acuerdo al momento en el que ocurren dentro del flujo: Inicio, Intermedio y Finalización.

Un evento de inicio (*start*) indica el comienzo de un flujo de ejecución, mientras que un evento de finalización (*end*) dispara la terminación del mismo. Los eventos intermedios (*intermediate*) ocurren entre el inicio y la finalización de un flujo de ejecución, de manera que pueden afectar el curso de acción seguido, pero no pueden finalizarlo directamente. La figura A1.1 muestra la representación gráfica de cada uno de estos eventos. Puede indicarse una causa de origen o un disparador para los eventos de inicio. Por otro lado, para los eventos de finalización se puede indicar un resultado esperado o arrojado como consecuencia de la ejecución del flujo. Por su parte, para los eventos intermedios puede indicarse una causa o un resultado. Al asociarle una causa de origen a un evento de inicio, se da lugar al tipo *start catch event*. Análogamente, al asociarle un resultado de ejecución a un evento de finalización, se da origen al tipo *end throw event*. Para los eventos intermedios, existen los tipos *intermediate catch event* e *intermediate throw event*.

Figura A1.1: Representación gráfica de los tres tipos de evento básicos en BPMN. Imagen extraída de [2].

Para cada uno de estos eventos, existen subtipos que permiten complementar su semántica, indicando la causa o el efecto de su ocurrencia. En [2] se expone un listado completo de cada uno de ellos, denotando cómo funcionan y en qué momento pueden utilizarse. A continuación, se detallan tres de los subtipos comúnmente más utilizados:

- Mensaje (*message*): si el evento es del tipo *start*, entonces indica el comienzo de un flujo luego de la recepción de un mensaje desde un participante. Si el evento es del tipo *end*, entonces indica el envío de un mensaje al finalizar el flujo. Si el evento es del tipo *intermediate*, entonces indica el envío de un mensaje (*intermediate throw message*) o la recepción de uno (*intermediate catch message*), causando la reanudación del flujo en este último caso.
- Temporizador (*timer*): si el evento es del tipo *start*, se usa para indicar el comienzo de un flujo al llegar determinada fecha o cumplirse determinado lapso de tiempo. Un evento del tipo *intermediate catch timer*, se usa para frenar la ejecución del flujo y reanudarlo luego de un determinado intervalo.
- Condicional (*conditional*): si el evento es del tipo *start*, se usa para indicar el comienzo de un flujo al cumplirse una condición. Si el evento es del tipo *intermediate catch*, entonces dispara un evento que indica que una condición se ha vuelto verdadera.

La figura A1.2 muestra la representación gráfica de cada uno de estos eventos. Como puede observarse, la representación de los eventos sigue un mismo patrón denotado por una figura central que indica el subtipo del evento, encerrada en un contorno que indica el tipo básico de evento (inicio, intermedio o finalización). Cuando el evento es del tipo *throw*, entonces la figura central muestra un relleno sólido, mientras que en los eventos del tipo *catch* no sucede esto.

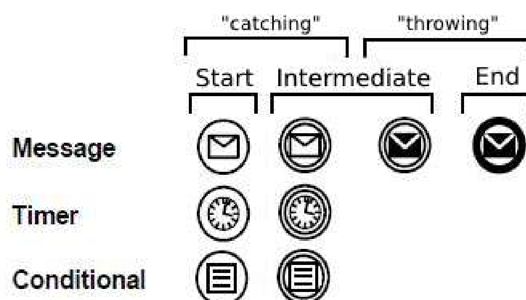


Figura A1.2: Representación gráfica de los subtipos de evento message, timer y conditional. Imagen adaptada de [2].

ACTIVIDADES

Una actividad (*activity*) se utiliza para representar trabajo que es realizado durante un proceso de negocios. Las actividades pueden ser atómicas o compuestas. Existen tres tipos fundamentales de actividades:

- Tarea (*task*): representa una actividad atómica dentro de un flujo de ejecución. Normalmente, una tarea es específica de un determinado flujo o proceso, aunque también existen las tareas globales que pueden repetirse a lo largo de distintos escenarios. Existen varios tipos de tarea:
  - Tarea de servicio (*service task*): representa una tarea que utiliza un servicio web o una aplicación automatizada.
  - Tarea de envío (*send task*): representa la tarea de enviar un mensaje.
  - Tarea de recepción (*receive task*): representa la tarea de esperar la recepción de un mensaje enviado por un participante.
  - Tarea de usuario (*user task*): representa una tarea que es realizada por una persona con la asistencia de una aplicación de software.
  - Tarea manual (*manual task*): representa una tarea que es realizada sin utilizar ningún tipo de soporte automatizado.
  - Regla del negocio (*business rule task*): permite señalar la ejecución de una regla del negocio a través de un motor de reglas del negocio, y obtener los resultados de la misma.
  - Script (*script task*): permite especificar un fragmento de código que será ejecutado por un motor de procesos de negocio.
- Subproceso (*subprocess*): consiste en una actividad compuesta, cuyos detalles internos han sido modelados utilizando actividades, compuertas, eventos y flujos de secuencia. Un subproceso puede ser representado ocultando sus detalles internos (vista colapsada) o exponiéndolos (vista expandida), como se muestra en la figura A1.3.
- Actividad de llamada (*call activity*): esta actividad referencia a una tarea global o a un proceso, y su ejecución implica una transferencia del control hacia el elemento referenciado. Existen tres tipos de representación gráfica para esta actividad, como lo ilustra la figura A1.3.

La figura A1.3 muestra la representación gráfica de los distintos tipos de actividad en BPMN.

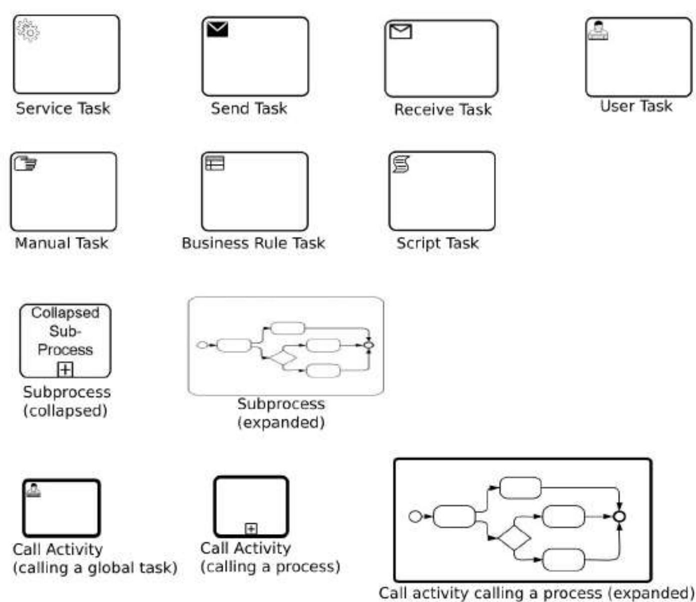


Figura A1.3: Representación gráfica de los distintos tipos de actividad en BPMN. Imagen adaptada de [2].

COMPUERTAS

Las compuertas (*gateways*) son utilizadas para mostrar cómo interactúan los flujos de secuencia a medida que convergen o divergen dentro de un proceso. Estos elementos permiten controlar la forma en la que se ejecuta el trabajo. Existen varios tipos de compuertas:

- De exclusión (*exclusive*): permiten definir una serie de caminos alternos de ejecución, donde sólo uno es ejecutado en una determinada situación. El camino que se ejecutará dependerá de la evaluación de una condición asociada al mismo. Ante la primera condición evaluada como verdadera, los demás caminos son descartados y se ejecutará el camino determinado por dicha condición.

- De inclusión (*inclusive*): permiten definir una serie de caminos alternos de ejecución, donde es posible que se ejecuten más de uno a la vez. La primera condición evaluada como verdadera no descarta la evaluación de las demás condiciones, de manera que pueden evaluarse y ejecutarse varios caminos en simultáneo.
- Basadas en eventos (*event-based*): permiten definir una serie de caminos alternos cuya ejecución depende de la evaluación de condicionales sobre los datos de un evento.
- Basadas en eventos y paralelas (*parallel event-based*): permiten definir una serie de caminos alternos y paralelos, donde la ejecución de cada uno depende de la evaluación de una condición sobre los datos de un evento. En este caso, la ejecución de un camino no descarta la posible ejecución de los demás, que quedan a la espera de los eventos asociados a las compuertas.
- De paralelismo (*parallel*): son utilizadas para crear y sincronizar flujos de ejecución paralelos entre sí.
- Complejas (*complex*): son utilizadas para modelar comportamientos de sincronización que resultan complejos y que escapan del alcance de las compuertas anteriormente mencionadas.

La figura A1.4 muestra la representación gráfica de cada compuerta en BPMN.

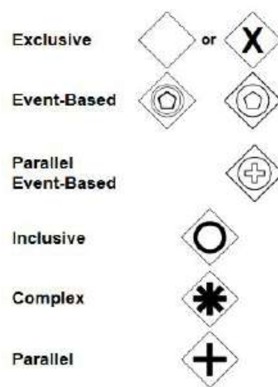


Figura A1.4: Representación gráfica de los distintos tipos de compuerta en BPMN. Imagen extraída de [2].

MODELADO DE DATOS

BPMN brinda soporte al modelado de los datos utilizados a lo largo de los procesos de negocio a través de varios elementos, cada uno con su significado semántico y su propia representación gráfica:

- Mensaje (*message*): representa el contenido de una comunicación entre dos participantes. Los mensajes pueden asociarse a flujos de mensaje, representando el contenido de dicha interacción.
- Objeto de dato (*data object*): provee información sobre las actividades que deben ejecutarse, o sobre lo que produce cada una de ellas. Este elemento puede representar a un objeto de datos singular, o a una colección de ellos (*data object collection*).
- Dato de entrada (*data input*): permite representar la información necesaria para la ejecución de un determinado proceso, subproceso, o de una actividad.
- Dato de salida (*date output*): permite representar el resultado que debe arrojar la ejecución exitosa de un proceso, subproceso o de una actividad.
- Repositorio de datos (*data store*): representa un almacén o repositorio que actúa como proveedor de datos almacenados de forma persistente.

La figura A1.5 muestra la representación gráfica de cada uno de estos elementos.

OBJETOS DE CONEXIÓN

Los objetos de conexión que provee BPMN son utilizados para relacionar los elementos gráficos de los modelos y para indicar el orden de ejecución del trabajo seguido en los procesos de negocio. Los elementos de conexión disponibles son los siguientes:

- Flujo de secuencia (*sequence flow*): este elemento permite indicar el orden de ejecución del trabajo. La ejecución se indicará de acuerdo al sentido en el que apunta el flujo.
- Flujo de mensaje (*message flow*): este elemento permite señalar la existencia de una comunicación entre participantes. La comunicación identifica dos roles: el emisor y el receptor del mensaje; cada uno identificado de acuerdo al sentido que muestre el flujo.

- Asociación de dato (*data association*): este elemento es un conector que permite indicar una relación entre un proceso, subproceso o actividad y un elemento de datos (*data object*, *data input* o *data output*); señalando que el mismo actúa como la entrada necesaria para realizar el trabajo, o el resultado que debe obtenerse del mismo.

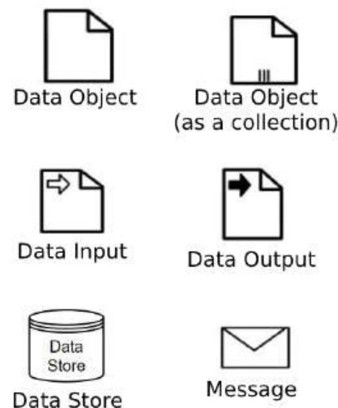


Figura A1.5: Representación gráfica de los elementos utilizados para el modelado de datos en BPMN. Imagen adaptada de [2].

La figura A1.6 muestra la representación gráfica de los elementos de conexión.

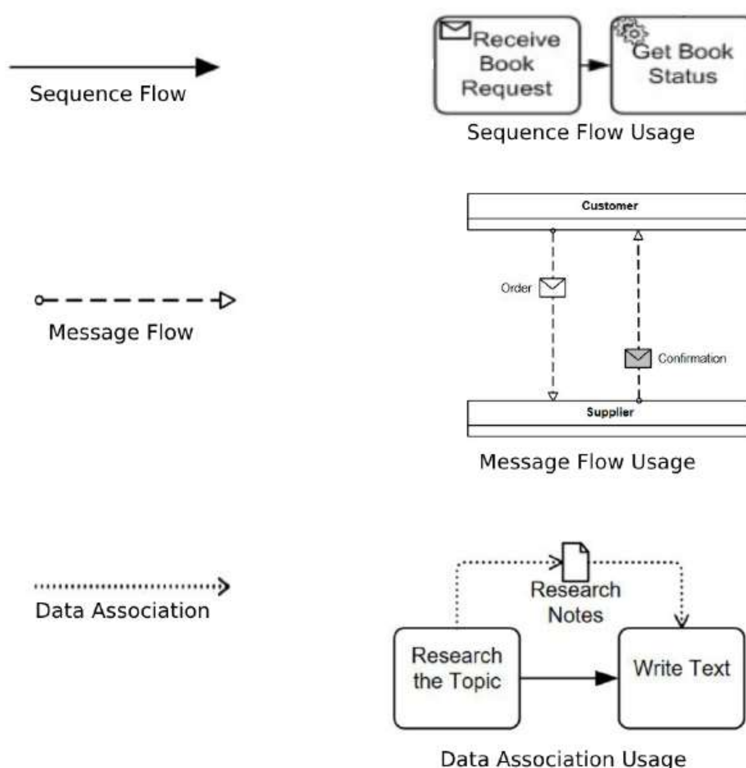


Figura A1.6: Representación gráfica de los elementos de conexión en BPMN. Imagen adaptada de [2].

CONTENEDORES Y SUBDIVISIONES

Un *pool* es la representación gráfica de un participante en BPMN. Cada participante en un modelo, representa a un actor externo (por ejemplo, otra compañía) o un determinado rol que interviene en el proceso (por ejemplo, vendedor, proveedor o usuario). Los *pools* pueden ser utilizados para ilustrar las actividades del proceso que son desarrolladas en una organización u otra, mostrando una vista de *caja blanca* con respecto a lo que sucede en los actores externos.

Un *lane* es una subdivisión de un *pool* que permite representar los distintos roles involucrados que a su vez forman parte de un participante. Otro uso de los *lanes* es el de clasificar las actividades del proceso contenido en su *pool*.

La figura A1.7 muestra los elementos *pool* y *lane*, respectivamente. Para el primero, se presenta la versión alternativa que permite la vista de *caja blanca*.

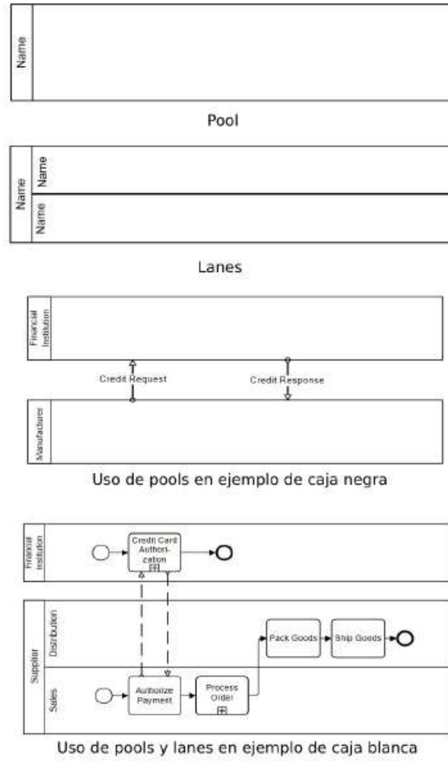


Figura A1.7: Representación gráfica de los elementos pool y lane, y visualización de un mismo ejemplo mediante las vistas de caja negra y de caja blanca. Imagen adaptada de [2].

A2. Tipos de datos y operadores en OCL

Los tipos de dato básicos definidos en OCL se detallan a continuación en la tabla A2.1.

Tipo de dato	**Descripción**	**Ejemplos**
Integer	Representa un número entero.	-1, 30, 3444
Real	Representa un número decimal. Incluye a los enteros.	1.5, 3.14
Boolean	Representa un valor lógico. Los valores que admite son *true* y *false*.	*true*
String	Representa una cadena de texto.	‘este es un string’
OclInvalid	Representa un valor inválido. Admite un único valor *invalid*.	*invalid*
OclVoid	Representa un valor vacío. Admite los valores *invalid* y *null*.	*invalid*, *null*

Tabla A2.1: Tipos de dato básicos en OCL.

En adición a los anteriores, OCL define algunos tipos de dato para la representación de conjuntos de valores: *Collection*, *Set*, *Sequence*, y *Bag*. Estos tipos de dato son detallados en la tabla A2.2.

Tipo	**Descripción**
Collection	*Collection* es el súper tipo abstracto sobre el que se definen todas las operaciones de conjuntos que pueden ser utilizadas por los usuarios de OCL.
Set	Representa un conjunto de elementos bajo la definición matemática de conjuntos. No admite elementos repetidos.
Bag	Equivale a un *Set* aunque admite elementos repetidos.
Sequence	Equivale a un *Bag* aunque sus elementos se encuentran ordenados (*Set* y *Bag* no admiten ningún tipo de orden en sus elementos).

Tabla A2.2: Tipos de dato utilizados para representar conjuntos.

Por último, dado que las expresiones OCL son escritas en el contexto de un modelo UML, todos los tipos especificados en el mismo (clases, asociaciones, tipos de dato) pueden ser utilizados en la construcción de dichas expresiones.

OCL soporta un conjunto amplio de operadores nativos que son aplicables sobre determinados tipos de dato. A continuación, se mencionan los operadores de uso más común.

Los siguientes operadores son aplicables a todos los tipos de dato soportados por OCL:

- = (Igual a)
- <> (Distinto a)

Los siguientes operadores son aplicables a los tipos de dato *Integer* y *Real*:

- + (Suma)
- - (Resta)
- * (Producto)
- / (División)
- > (Mayor a)
- >= (Mayor o igual a)
- < (Menor a)
- <= (Menor o igual a)

Los siguientes operadores son aplicables sobre el tipo de dato *String*:

- + (Concatenación)
- > (Mayor a)
- >= (Mayor o igual a)
- < (Menor a)
- <= (Menor o igual a)

Los siguientes operadores son aplicables sobre el tipo de dato *Boolean*:

- *or* (Disyunción)
- *and* (Conjunción)
- *xor* (Disyunción exclusiva)
- *not* (Negación)
- *implies* (Implicación)

OCL define dos operadores de navegabilidad que permiten recorrer las relaciones entre los objetos de los modelos UML:

- . (punto): este operador es utilizado para navegar desde un objeto utilizando una propiedad u operación. Ejemplos: *persona.nombre*; *persona.toString()*.
- $->$ (flecha): este operador es utilizado para navegar desde una colección utilizando una propiedad u operación. Ejemplo: $persona.padres->size()$.

Por último, OCL pone a disposición de sus usuarios una extensa librería de funciones aplicables a los distintos tipos de dato que simplifican la construcción de las expresiones. Las funciones pueden consultarse en [15].

A3. Proceso de creación de un proyecto en BPMN2 Modeler

Luego de la instalación del *plugin* BPMN2 Modeler en Eclipse IDE, dos nuevos tipos de artefactos son añadidos bajo la categoría *BPMN2* en la interfaz de creación de artefactos: *BPMN2 Model* y *Generic BPMN 2.0 Diagram*, como lo muestra la figura A3.1. Seleccionando el segundo, se abrirá la ventana mostrada en la figura A3.2. La interfaz de creación de artefactos puede ser accedida seleccionando la opción *File > New > Other* en el IDE.

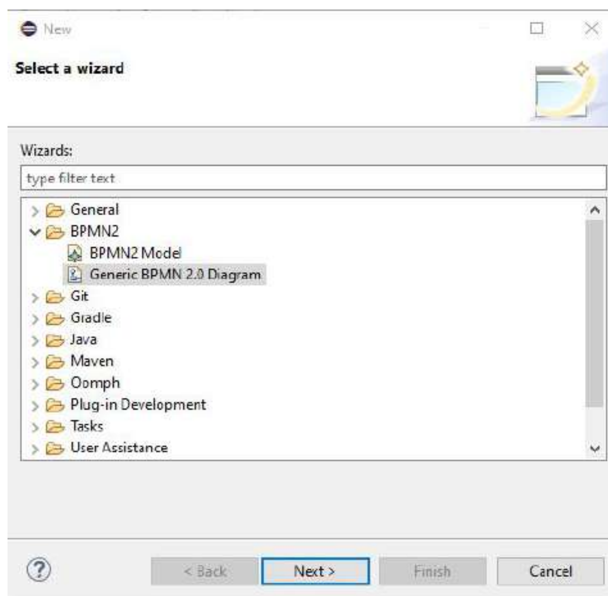


Figura A3.1: Interfaz de creación de artefactos para un proyecto en Eclipse IDE.

La figura A3.2 muestra la ventana de selección del tipo de diagrama BPMN. Seleccionando la segunda opción se creará un nuevo diagrama de colaboraciones BPMN. Esta selección abrirá la ventana mostrada en la figura A3.3 que permite colocarle un nombre al diagrama y un *namespace* (espacio de nombres) alternativo al que viene por defecto. Presionando *Finish* se inicializará el diagrama y se abrirá el editor gráfico del modelo y la vista de *Outline*, como lo muestra la figura A3.4.

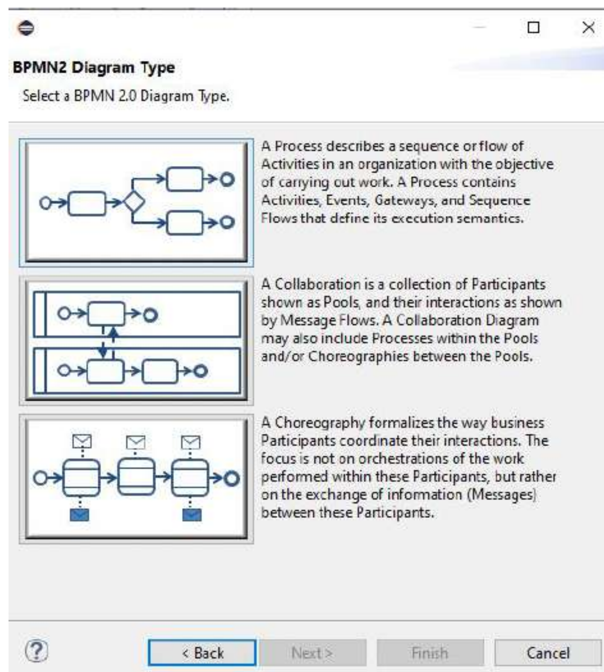


Figura A3.2: Interfaz de selección de tipos de diagrama BPMN soportados por BPMN2 Modeler.

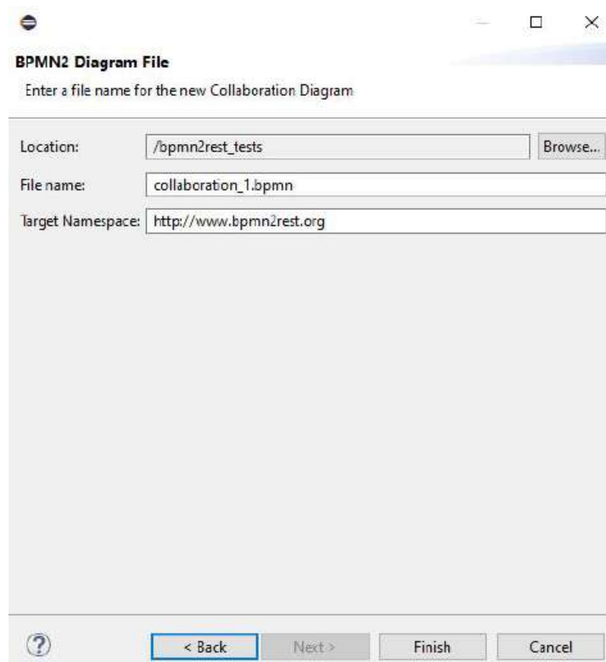


Figura A3.3: Interfaz de configuración inicial del diagrama de colaboraciones BPMN.

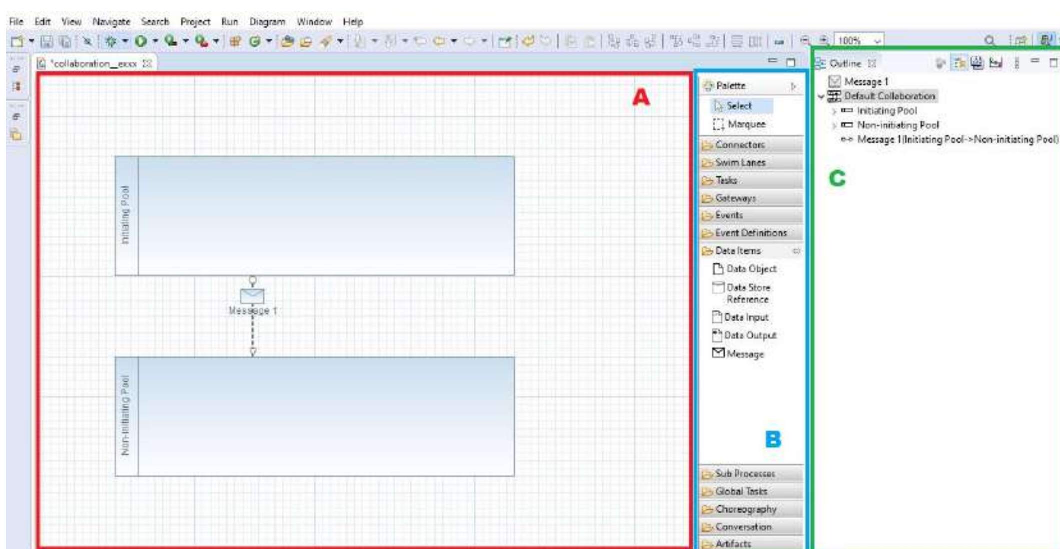


Figura A3.4: Editor gráfico para los diagramas BPMN embebido en Eclipse IDE. En A, se muestra el editor gráfico central. En B, se muestra la paleta de herramientas y elementos gráficos que pueden utilizarse para confeccionar el modelo. En C se muestra la vista de Outline con la lista de los elementos activos.

El editor gráfico muestra la representación del modelo utilizando los elementos de la notación. Pueden añadirse más elementos con la herramienta *Palette* embebida en el editor (ver B en figura A3.4). En la vista de *Outline* se listan los elementos del modelo actualmente en uso (ver C en figura A3.4). Haciendo doble click en un elemento listado en la vista de *Outline* o en su representación gráfica en el editor, se podrá acceder a la ventana de configuración de propiedades del elemento. Esta ventana permite cambiar los atributos de los distintos elementos y normalmente, se estructura en distintas pestañas de configuración destinadas a cubrir distintos aspectos del mismo. La figura A3.5 muestra la ventana de configuración del elemento *Message*.

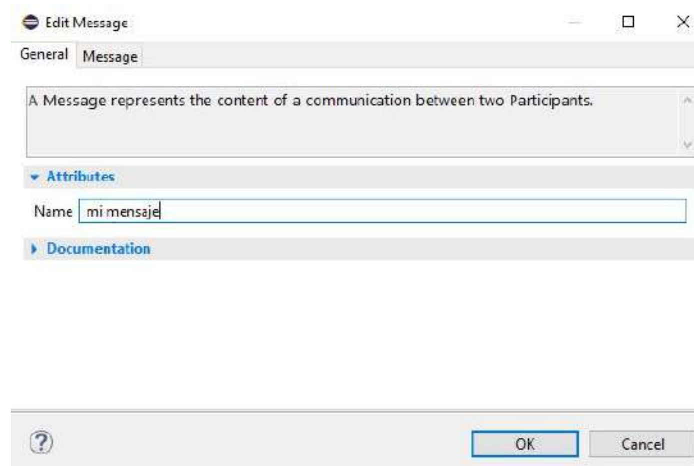


Figura A3.5: Ventana de configuración de propiedades para el elemento Message.

A4. Proceso de creación de un proyecto Acceleo

Una vez instalado Acceleo sobre Eclipse IDE, tres nuevos tipos de proyectos son añadidos a la interfaz de selección de tipos de proyectos, bajo la categoría *Acceleo Model to Text*: *Acceleo Project*, *Acceleo UI Launcher Project* y *Convert to an Acceleo Project* tal como lo muestra la figura A4.1. La opción *Acceleo Project* permite la creación de un nuevo proyecto Acceleo. Se puede acceder a la interfaz de selección de tipos de proyectos mediante *File > New > Project*.

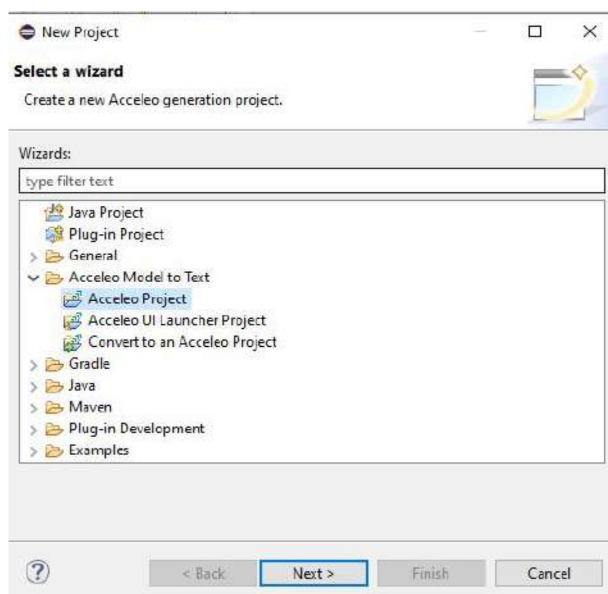


Figura A4.1: Interfaz de selección de tipos de proyecto desplegada al crear un nuevo proyecto en Eclipse IDE.

Al seleccionar *Acceleo Project*, se desplegará la ventana de configuración inicial mostrada en la figura A4.2. Un proyecto Acceleo se compone de un conjunto de módulos. Un módulo es un archivo con la extensión *.mtl* que contiene plantillas o consultas. Un proyecto Acceleo debe tener un módulo principal denominado “main” desde el cual inicia el flujo de ejecución de la transformación M2T. La ventana de configuración mostrada en la figura A4.2 permite añadir tantos módulos como sean necesarios al proyecto. Para cada módulo, es posible definir un nombre, el directorio donde se alojará el módulo, el metamodelo con el que trabajará el módulo, el tipo de dato sobre el cual se ejecuta el módulo, y si se trata de una plantilla (*template*) o de una consulta.

Al añadir un metamodelo a un módulo, se despliega una lista de URIs que referencian a los distintos metamodelos estándar desarrollados con la tecnología EMF. Un módulo en Acceleo puede trabajar con uno o varios metamodelos EMF para confeccionar una transformación del tipo M2T. Es posible

que los módulos trabajen con metamodelos personalizados que hayan sido generados con EMF y que sean distintos a los estándares que provee esta tecnología. La figura A4.3 muestra el listado desplegado al añadir un nuevo metamodelo al módulo seleccionado.

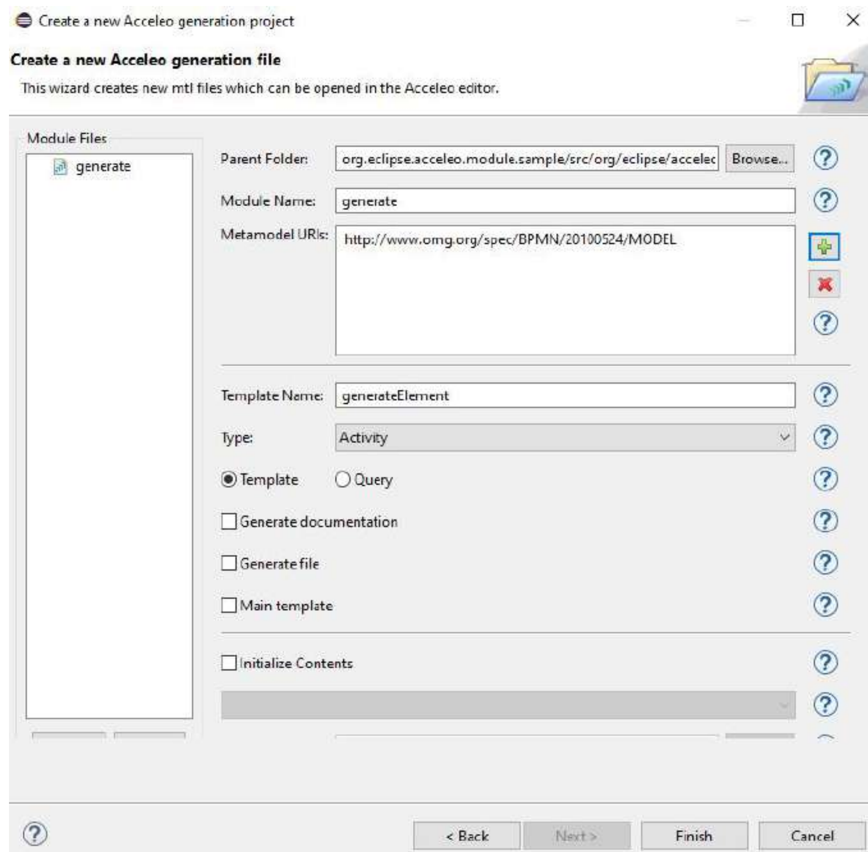


Figura A4.2: Ventana de configuración inicial de un proyecto Acceleo. Desde aquí se pueden añadir los módulos que contendrá el proyecto.

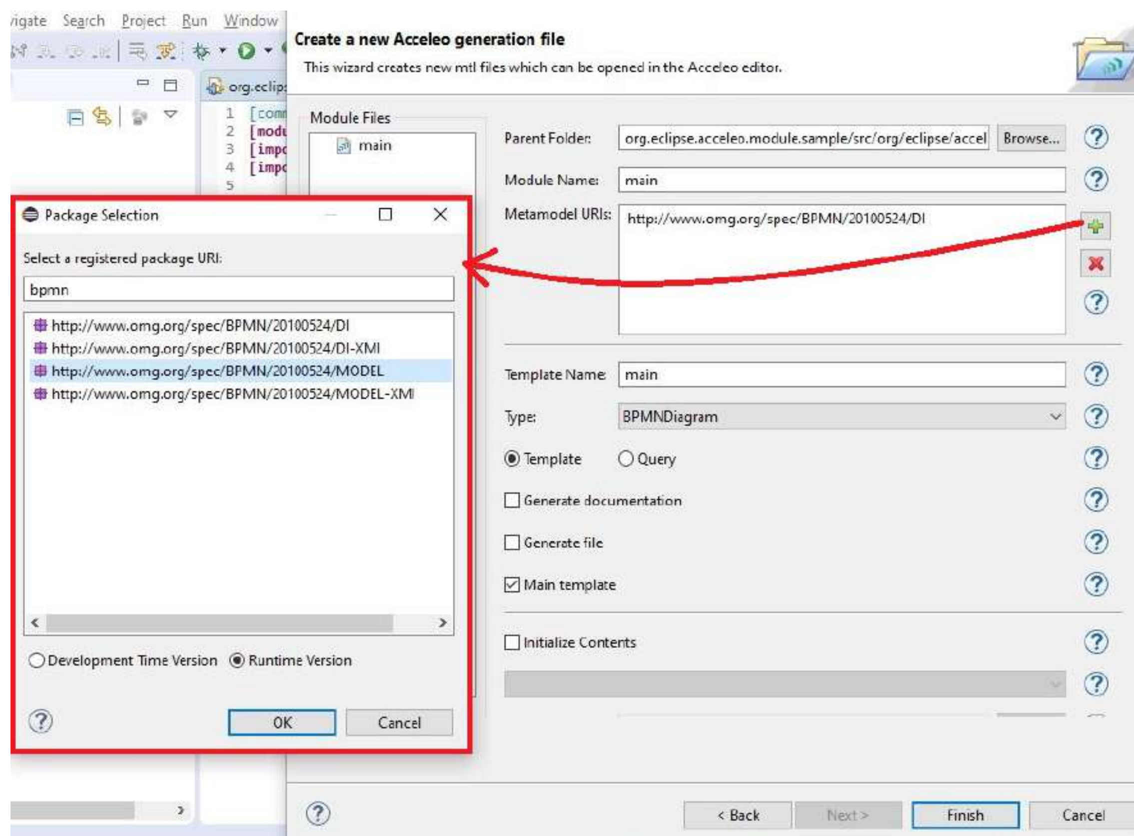


Figura A4.3: Interfaz de selección de metamodelos asociados al módulo nuevo añadido.

Una vez finalizada la configuración inicial, el proyecto Acceleo tendrá un conjunto de módulos *.mtl*. El módulo principal *main* es el que se ejecuta inicialmente durante una transformación. Comúnmente, desde *main* se incluyen o utilizan los demás módulos añadidos al proyecto. Cada módulo consiste en un archivo con la extensión *.mtl* que contiene instrucciones y texto plano. Las instrucciones son interpretadas por el motor de ejecución de Acceleo y son descriptas en el lenguaje Acceleo, que

se corresponde con una implementación del estándar *MOF Model to Text Transformation Language* (MOFM2T) o simplemente *Model Transformation Language* (MTL) recomendado en [60] por la OMG.

Luego de la definición de cada módulo utilizando las instrucciones del lenguaje Acceleo, es posible ejecutar la transformación sobre un modelo de entrada. Este modelo debe estar desarrollado sobre el/los metamodelo/s que utiliza cada módulo. Acceleo genera automáticamente una configuración de ejecución para Eclipse IDE cuando se trata de un proyecto del tipo Acceleo. Una configuración de ejecución permite indicar las distintas opciones y parámetros de entrada que deben utilizarse para ejecutar un determinado proyecto en Eclipse. El usuario puede editar los valores definidos por defecto en las configuraciones de ejecución para cambiar la forma en que son ejecutados los proyectos. La figura A4.4 muestra un ejemplo de una configuración de ejecución generada automáticamente para un proyecto en Acceleo. Como se observa, es posible cambiar el modelo de entrada a la transformación, el directorio de salida de los documentos y otros valores para obtener distintos resultados en la ejecución. Para ejecutar un proyecto Acceleo utilizando una configuración de ejecución, debe seleccionarse el proyecto y luego las opciones *Run* > *Run As* > *Launch Acceleo Application*.

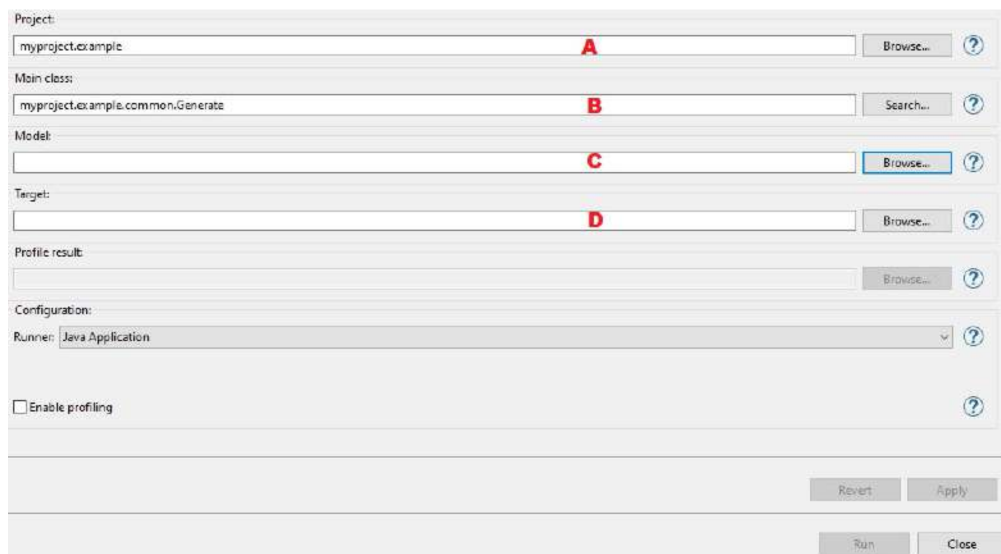


Figura A4.4: Configuración de ejecución generada para un proyecto Acceleo. El campo A permite indicar el proyecto a ser ejecutado. El campo B, indica la clase principal del motor de ejecución del proyecto. Esta clase es generada por Acceleo. El campo C, permite introducir el modelo de entrada sobre el que se ejecutará la transformación. El campo D, permite configurar el directorio de salida donde se generarán los documentos de texto. Los campos A y B son completados por Acceleo automáticamente, mientras que el usuario debe indicar el modelo de entrada y el directorio de salida.

A5. Generación automática de *plugins* desde proyectos Acceleo

Desde la interfaz de selección de tipos de proyectos (ver figura A4.1), el proyecto del tipo *Acceleo UI Launcher Project* permite construir un *plugin* para Eclipse IDE a partir de un proyecto Acceleo. Al seleccionarlo, se debe indicar cuál es el proyecto Acceleo desde el cual se construirá el *plugin*, como lo muestra la figura A5.1. Una vez finalizada la configuración inicial, Acceleo genera de manera automática un *plugin* genérico para Eclipse IDE que añade un nuevo botón a la interfaz del usuario. Este botón permite ejecutar la transformación M2T sobre un modelo EMF de entrada. El botón se muestra en el menú contextual de las vistas de exploración de paquetes o de exploración de recursos, desplegado al hacer click izquierdo sobre un elemento en particular, como se muestra en la figura A5.2.

Durante la construcción de un proyecto del tipo Acceleo, un conjunto de clases Java son adicionadas a los archivos *.mtl* que definen la transformación M2T. Estas clases corresponden al motor de ejecución de Acceleo generado automáticamente por dicha tecnología. El motor de ejecución es utilizado para ejecutar la transformación cuando se ejecuta el proyecto utilizando una configuración de ejecución. Este motor de ejecución consiste en un conjunto de clases Java que utilizan el *core* de Acceleo para generar los documentos de texto indicados en los módulos *.mtl*. Acceleo permite generar un *plugin* para Eclipse IDE a partir de un proyecto del tipo Acceleo. Dicho *plugin*, básicamente hace uso de estas clases Java que constituyen el motor de ejecución de la transformación. El *plugin* generado utiliza los puntos de extensión que provee el IDE de forma nativa, para incorporar nuevos botones a los menúes existentes en el mismo. Así, se incorpora un nuevo botón cuya selección invoca al motor de ejecución del proyecto Acceleo enviándole el modelo de entrada seleccionado.

Resulta adecuado utilizar el *plugin* generado automáticamente por Acceleo cuando se trata de proyectos simples y de corto alcance. Sin embargo, cuando se trata de proyectos de mayor envergadura, es conveniente personalizar dicho desarrollo haciendo uso de las clases Java que forman parte del motor de ejecución de Acceleo según sea necesario.

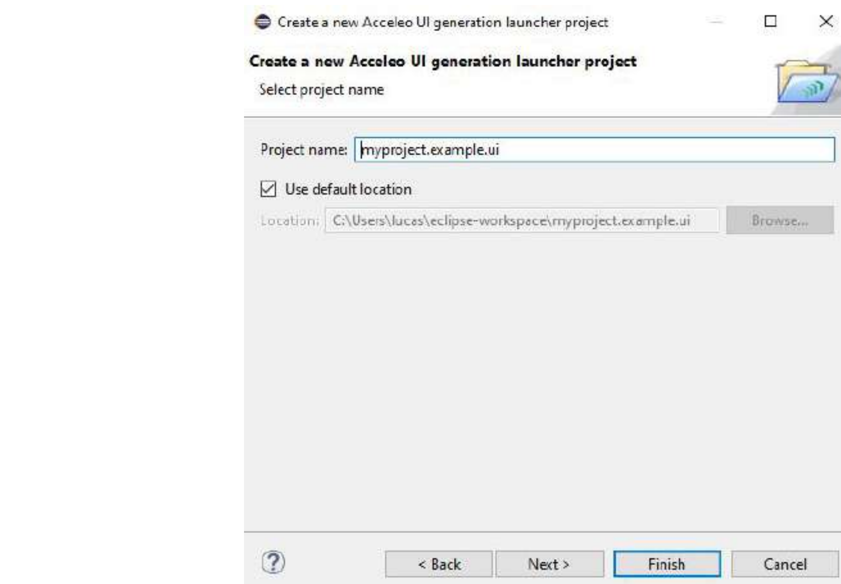


Figura A5.1: Interfaz de configuración de un proyecto plugin generado desde un proyecto Acceleo.

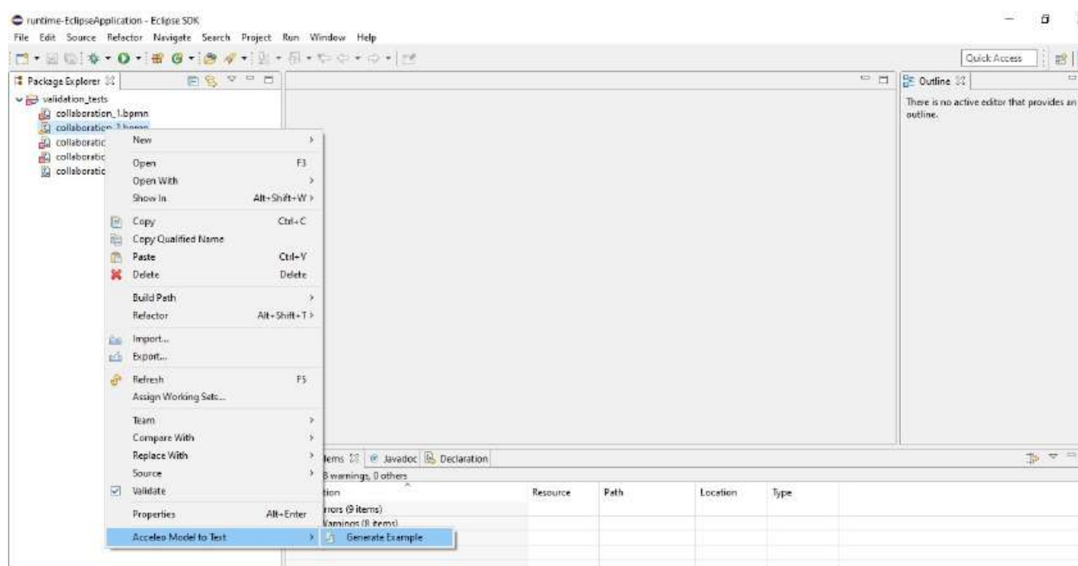


Figura A5.2: Visualización de un plugin generado automáticamente desde un proyecto Acceleo. El botón “Generate Example” permite ejecutar la transformación sobre un modelo de entrada.

A6. Listado de herramientas de modelado BPMN comunes a los sitios web de referencia

El listado de la tabla A6.1 muestra las herramientas de modelado BPMN mencionadas en alguno de los sitios web tomados como referencia durante el proceso de selección de la herramienta principal. Los resultados se ordenan por número de apariciones en todos los sitios.

Número de menciones en los sitios de referencia	**Nombre de las herramientas**
4	Modelio, Camunda Modeler
3	Bonitasoft, ARIS Express, Alfresco, ProcessMaker
2	Bizagi Modeler, RedHat Jboss BPM Suite, Adobe LiveCycle, RunaWFE, jBPM, Joget, jSonic BPM
1	Enterprise Architect, MagicDraw, Umodel, Visual Paradigm, Rational Software Architect, ADONIS: Community Edition, BeePMN, BPMN.io, MID Innovator for Business Analysts, Yaoqiang BPMN Editor, Orchestra, Eclipse BPMN2 Modeler

Tabla A6.1: Listado de herramientas de modelado BPMN, presentado en función de la cantidad de menciones en los sitios de referencia.

A7. Estructura de cada punto de extensión incorporado en los desarrollos

A continuación, se muestra la especificación de cada punto de extensión utilizado a lo largo de los desarrollos abordados en la construcción de la extensión para Eclipse IDE. Los ítems anidados indican que se trata de un elemento o atributo XML contenido en otro elemento. Al final de cada especificación, se muestra su uso real en la extensión, mostrando el fragmento XML que contiene el punto de extensión.

A7.1. Punto de extensión org.eclipse.bpmn2.modeler.runtime

Este punto de extensión es definido por BPMN2 Modeler ([61]) y puede ser incorporado en una extensión de la herramienta para añadir nuevas características a la misma, cambiando las utilidades del entorno de modelado gráfico, la representación gráfica de los elementos de la notación, las pestañas de configuración de propiedades de los distintos elementos, o el modelo de la notación utilizado por la herramienta.

El punto de extensión define los siguientes elementos:

- **runtime:** este elemento es utilizado de manera obligatoria al incorporar el punto de extensión. El mismo define un *namespace* para los modelos BPMN confeccionados con esta extensión. Definir este elemento permite asignarle una identidad a la extensión que se está desarrollando. Sólo se define una vez.
  - **id (atributo):** define el ID utilizado para identificar de manera unívoca a la extensión.
  - **name (atributo):** asigna un nombre a la extensión que será visualizada en la pantalla de selección de extensiones de BPMN2 Modeler (ver figura 4.11 en sección 4.2.1).
  - **description (atributo):** asigna una descripción breve a la extensión que será visualizada junto con su nombre.
  - **class (atributo):** clase Java que implementa la interfaz org.eclipse.bpmn2.modeler.core.IBpmn2RuntimeExtension. La misma debe proveer datos tales como el *namespace* y las URIs de los tipos de dato y/o lenguajes utilizados en los modelos, e implementar los *hooks* que serán invocados en las distintas etapas del ciclo de vida del plugin.
- **modelExtension:** este elemento permite extender un elemento de la notación estándar utilizada por BPMN2 Modeler, añadiéndole nuevas propiedades. Puede definirse varias veces.
  - **id (atributo):** asigna un ID único a la extensión del elemento.
  - **name (atributo):** asigna un nombre representativo a la extensión del elemento.
  - **despription (atributo):** asigna una descripción breve de la extensión del elemento.
  - **runtimeId (atributo):** el ID del elemento *runtime* de la extensión de BPMN2 Modeler sobre la cual debe aplicarse esta extensión en el modelo.
  - **type (atributo):** el elemento de la notación BPMN que se va a extender. Ejemplos: Operation, Message, MessageFlow, entre otros.
  - **property:** este elemento permite incorporar una propiedad al elemento de la notación, dando lugar a la extensión de la misma. Puede definirse varias veces para incorporar varias propiedades.
    - **label (atributo):** leyenda que se mostrará en las pestañas de configuración señalando qué representa esta propiedad.
    - **description (atributo):** breve descripción de la propiedad.
    - **name (atributo):** este valor será utilizado como el nombre de la propiedad, y por lo tanto, indicará cómo se puede obtener su valor para un elemento de la notación que la utilice.
    - **type (atributo):** tipo de dato que utilizará la propiedad. Si no se define, se asume que será un *string*.
    - **value (atributo):** valor por defecto para la propiedad.
- **propertyTab:** este elemento permite definir una nueva pestaña de configuraciones para un elemento de la notación BPMN. Sus valores serán interpretados por BPMN2 Modeler, dando lugar a la nueva pestaña de configuraciones. Generalmente, se utiliza para poder habilitar las propiedades añadidas a un elemento mediante *modelExtension*, para que puedan ser asignadas desde la ventana de configuración del mismo. Pueden definirse varios elementos de este tipo.
  - **id (atributo):** ID que identifica de manera unívoca a esta pestaña de configuración.
  - **runtimeId (atributo):** referencia al elemento *runtime*. Señala cuál es la extensión sobre la que se aplicará esta pestaña de configuración.

- **label (atributo):** leyenda que se mostrará como el título de la pestaña de configuración.
- **class (atributo):** este atributo permite indicar una clase responsable de renderizar la pestaña de configuración. Si se pone el valor “default”, entonces BPMN2 Modeler se encargará de tal tarea.
- **popup (atributo):** si se activa, permite que la pestaña de configuración sea visible para las ventanas de configuración flotantes.
- **type (atributo):** indica el elemento de la notación BPMN sobre el cual se añadirá esta pestaña, dentro de su ventana de configuración.
- **features (atributo):** indica qué propiedades del elemento deben mostrarse en la pestaña. Su valor es un *string* conteniendo los nombres de las propiedades separadas por un espacio en blanco.

La figura A7.1 muestra la incorporación de este punto de extensión en el desarrollo abordado.

```xml
<extension
      point="org.eclipse.bpmn2.modeler.runtime">
   <runtime
         class="org.eclipse.contributions.bpmn2rest.modeler.Bpmn2RestRuntimeExtension"
         description="BPMN2REST BPMN2 Modeler Plugin Extension"
         id="org.eclipse.contributions.bpmn2rest.modeler.runtime"
         name="BPMN2REST Extension">
   </runtime>
   <modelExtension
         description="BPMN2REST Operation extension for supporting REST features"
         id="org.eclipse.contributions.bpmn2rest.modeler.extensions.Operation"
         name="BPMN2REST Operation Extension"
         runtimeId="org.eclipse.contributions.bpmn2rest.modeler.runtime"
         type="Operation">
      <property
            description="URL or path to the endpoint of this operation"
            label="REST Service URL"
            name="rest_url">
      </property>
      <property
            description="HTTP method to use on this RESTful service"
            label="REST Service HTTP Method"
            name="rest_method"
            value="GET">
      </property>
      <property
            description="Prepend the controller&apos;s name to the URL. It avoids having the same REST URL for the same
            label="Add Controller Preffix to URL"
            name="rest_prepend_controller"
            type="EBoolean"
            value="false">
      </property>
   </modelExtension>
   <propertyTab
         class="default"
         features="rest_url rest_method rest_prepend_controller"
         id="org.eclipse.contributions.bpmn2rest.modeler.propertyTabs.Operation"
         label="BPMN2REST Extension"
         popup="true"
         runtimeId="org.eclipse.contributions.bpmn2rest.modeler.runtime"
         type="Operation">
   </propertyTab>
</extension>
```

Figura A7.1: Fragmento XML que muestra la incorporación del punto de extensión org.eclipse.bpmn2modeler.runtime.

A7.2. Punto de extensión org.eclipse.emf.validation.constraintProviders

Este punto de extensión es definido por una extensión de Eclipse EMF, denominada EMF Validation Framework, y es utilizada para definir reglas de validación sobre los modelos construidos con la tecnología EMF ([62]). Dado que BPMN2 Modeler fue desarrollado empleando esta tecnología, puede incorporarse este punto de extensión para añadir nuevas reglas de validación a los modelos construidos con esta herramienta. Las reglas de validación deben ser definidas y encapsuladas en un plugin que extienda a BPMN2 Modeler.

El punto de extensión define los siguientes elementos:

- **category:** este elemento permite definir una categoría para agrupar las reglas de validación. Puede definirse varias veces, para definir varias categorías a la vez.
  - **id (atributo):** ID que identifica de forma unívoca a la categoría.
  - **name (atributo):** nombre asignado a la categoría.
- **constraintProvider**: este elemento contiene la definición de las reglas de validación. Normalmente se define una sola vez.
  - **cache (atributo):** es utilizado únicamente por el *core* de validación. Normalmente no se cambia su valor por defecto (true).
  - **mode (atributo):** define el modo de validación utilizado, puede ser “Batch” o “Live”.
  - **constraints:** permite agrupar un conjunto de reglas de validación empleando un conjunto de categorías. Actúa como un contenedor para las reglas de validación. Puede definirse varias veces para delimitar varios grupos de reglas.

- categories (atributo): lista de ID's de categorías separadas por coma. Asigna al conjunto de reglas de validación las categorías definidas en este atributo.
  - **constraint:** este elemento define una regla de validación. Puede ser definido varias veces para incorporar varias reglas. Si el lenguaje de especificación de la regla es OCL, puede incorporarse un nodo de texto a este elemento indicando el código a ser evaluado.
    - **id (atributo):** ID que identifica de manera unívoca a la regla.
    - **name (atributo):** nombre asignado a la regla de validación.
    - **isEnabledByDefault (atributo):** define si la regla de validación estará activa o no por defecto.
    - **lang (atributo):** define en qué lenguaje se especificará la regla de validación. Puede ser "OCL" o "Java".
    - **mode (atributo):** modo de evaluación de la regla de validación. Puede ser "Batch" o "Live".
    - **severity (atributo):** nivel de importancia asignada al error que genera el incumplimiento de la regla. Puede ser "INFO", "WARNING", "ERROR" o "CANCEL".
    - **statusCode (atributo):** código de error que genera el incumplimiento de la regla.
    - **message:** contiene el mensaje de error generado ante el incumplimiento de la regla.
    - **target:** especifica el elemento (o los elementos) sobre el cual debe evaluarse la regla. El atributo **class** define el elemento BPMN sobre el cual se aplica la regla.
- **package:** define el metamodelo sobre el cual deben ser ejecutadas las reglas de validación.
  - **namespaceUri (atributo):** señala la URI hacia el metamodelo (en este caso, el metamodelo de BPMN).

La figura A7.2 muestra la incorporación de este punto de extensión en el desarrollo abordado.

A7.3. Punto de extensión org.eclipse.emf.validation.constraintBindings

Este punto de extensión se incorpora en conjunto con el anterior. Es utilizado para definir un criterio a partir del cual BPMN2 Modeler determina si debe aplicar o no las reglas de validación definidas en el punto de extensión anterior ([63]).

El punto de extensión define los siguientes elementos:

- **clientContext:** este elemento permite definir el criterio por el cual BPMN2 Modeler selecciona los elementos a los cuales aplicar un conjunto de reglas de validación. Un elemento "contexto" define, entonces, el contexto de aplicación de un conjunto de reglas de validación.
  - **id (atributo):** ID que identifica al contexto de manera unívoca.
  - **default (atributo):** indica si este contexto debe ser utilizado como el contexto por defecto. El contexto por defecto es utilizado por todas aquellas reglas de validación que no están asociadas a un contexto actualmente.
  - **enablement:** este elemento contiene el criterio de evaluación que utilizará BPMN2 REST sobre los elementos de un modelo BPMN para determinar si debe aplicar o no la regla.
    - **test:** permite definir una condición que formará parte del criterio establecido. La condición se determinará comparando el valor de una propiedad del modelo BPMN con un valor dado.
      - **property (atributo):** indica el ID de la propiedad que se evaluará en el modelo.
      - **value (atributo):** indica el valor con el cual se comparará la propiedad del modelo.
- **binding:** permite adjuntar un conjunto de reglas de validación a un *clientContext*, indicando que para los elementos que cumplan con el criterio, se deben aplicar las reglas contenidas en la categoría indicada.
  - **category (atributo):** indica la categoría de reglas a asociarse.
  - **context (atributo):** indica el *clientContext* que se asociará a la categoría.

La figura A7.3 muestra la incorporación de este punto de extensión al desarrollo abordado. En el mismo, el contexto se determina como los modelos BPMN cuya propiedad *runtime* sea el definido en los puntos de extensión anteriores. Esto aplicará las reglas de validación sólo a los modelos construidos con la extensión desarrollada.

```xml
<extension
      point="org.eclipse.emf.validation.constraintProviders">
   <category
         id="org.eclipse.contributions.bpmn2rest.validation.category"
         name="BPMN2REST Validation">
   </category>
   <constraintProvider
         cache="true"
         mode="Batch">
      <constraints
            categories="org.eclipse.contributions.bpmn2rest.validation.category">
         <constraint
               id="org.eclipse.contributions.bpmn2rest.validation.oneCollaboration"
               isEnabledByDefault="true"
               lang="OCL"
               mode="Batch"
               name="HasOneCollaboration"
               severity="ERROR"
               statusCode="1">
            <message>
               The model must contain only one collaboration
            </message>
            <target
                  class="Definitions">
            </target>
            <![CDATA[
                self.rootElements->select(e | e.oclIsKindOf(Collaboration))->size() = 1
            ]]>
         </constraint>
        <constraint
              id="org.eclipse.contributions.bpmn2rest.validation.twoParticipants"
              isEnabledByDefault="true"
              lang="OCL"
              mode="Batch"
              name="HasAtLeastTwoParticipantsConstraint"
              severity="ERROR"
              statusCode="1">
           <message>
              Collaborations must have at least 2 participants
           </message>
           <target
                 class="Collaboration">
           </target>
           <![CDATA[
               self.participants->size() > 1
           ]]>
        </constraint>
        <constraint
              id="org.eclipse.contributions.bpmn2rest.validation.oneValidInterface"
              isEnabledByDefault="true"
              lang="OCL"
              mode="Batch"
              name="HasAtLeastOneValidInterfaceConstraint"
              severity="ERROR"
              statusCode="1">
           <message>
              There must exists at least one participant implementing one interface
           </message>
           <target
                 class="Collaboration">
           </target>
           <![CDATA[
               self.participants->select(p | p.interfaceRefs->size() > 0)->size() > 0
           ]]>
        </constraint>
       <constraint
             id="org.eclipse.contributions.bpmn2rest.validation.oneCommunication"
             isEnabledByDefault="true"
             lang="OCL"
             mode="Batch"
             name="HasAtLeastOneCommunicationConstraint"
             severity="ERROR"
             statusCode="1">
          <message>
           There must exist at least one message flow linked to a message between particip
          </message>
          <target
                class="Collaboration">
          </target>
          <![CDATA[
              self.messageFlows->select(mf | mf.messageRef.oclIsKindOf(Message))->size() :
          ]]>
        </constraint>
        <constraint
              id="org.eclipse.contributions.bpmn2rest.validation.distinctServiceMessages"
              isEnabledByDefault="true"
              lang="OCL"
              mode="Batch"
              name="HasDistinctServiceMessages"
              severity="ERROR"
              statusCode="1">
           <message>
              Service providers can&apos;t have two operations with the same input message
           </message>
           <target
                 class="Collaboration">
           </target>
           <![CDATA[
               self.participants->forAll(p |
                    p.interfaceRefs
                    ->collect(i | i.operations)->flatten()
                    ->forAll(op1, op2 |
                            op1.inMessageRef <> op2.inMessageRef or
                            op1 = op2 or
                            self.messageFlows->select(mf |
                                mf.messageRef = op1.inMessageRef and
                                (
                                    mf.targetRef = p or
                                    (not p.processRef.oclIsUndefined() and p.processRef.fl
                                )
                            )->size() = 0
                    )
               )
           ]]>
        </constraint>
      </constraints>
      <package
            namespaceUri="http://www.omg.org/spec/BPMN/20100524/MODEL-XMI">
      </package>
   </constraintProvider>
</extension>
```

Figura A7.2: Fragmento XML que muestra la incorporación del punto de extensión org.eclipse.emf.validation.constraintProviders.

```xml
<extension
      point="org.eclipse.emf.validation.constraintBindings">
   <clientContext
         default="false"
         id="org.eclipse.contributions.bpmn2rest.acceleo.module.ui.clientContext">
      <enablement>
         <test
               property="org.eclipse.bpmn2.modeler.property.targetRuntimeId"
               value="org.eclipse.contributions.bpmn2rest.modeler.runtime">
         </test>
      </enablement>
   </clientContext>
   <binding
         category="org.eclipse.contributions.bpmn2rest.validation.category"
         context="org.eclipse.contributions.bpmn2rest.acceleo.module.ui.clientContext">
   </binding>
</extension>
```

Figura A7.3: Fragmento XML que muestra la incorporación del punto de extensión org.eclipse.emf.validation.constraintBindings.

A7.4. Punto de extensión org.eclipse.ui.menus

Este punto de extensión es definido por Eclipse IDE para permitir añadir componentes a la interfaz del usuario básica que provee la herramienta ([64]). Entre los componentes que se pueden añadir se encuentran: vistas, barras de herramientas, menúes principales, opciones en los menúes principales, menúes contextuales, entre otros.

El punto de extensión define los siguientes elementos:

- **menuContribution:** define un conjunto de componentes a ser añadidos en esta extensión del IDE. Puede definirse varias veces, indicando que se incorporarán varios grupos de componentes.
  - **locationURI (atributo):** contiene una URI cuyo formato indica la ubicación de los componentes. En esta URI se especifica en qué vista estarán los componentes, y en qué posición de la misma, indicando el orden relativo a otros botones o componentes.
  - **menu:** este elemento indica la incorporación de un menú dentro del conjunto de componentes agrupados en *menuContribution*. Puede definirse varias veces.
    - **id (atributo):** utilizado para asignar un ID que identifique de forma unívoca al menú.
    - **label (atributo):** define la leyenda o etiqueta con el que se mostrará el botón de apertura del menú.
    - **command:** este elemento se utiliza para insertar una referencia a un comando dentro de este menú. Esto dará como resultado una nueva opción en el menú, que se corresponde con el comando definido. Puede definirse varias veces.
      - **id (atributo):** define un ID que identifica este elemento de manera unívoca.
      - **commandId (atributo):** ID del comando que se asociará a esta opción.
      - **mnemonic (atributo):** cadena de caracteres sin espacios utilizada internamente por el IDE para identificar a este elemento.
      - **style (atributo):** define el estilo del botón del menú. Puede ser "push" (botón normal, por defecto); "radio" (estilo *radio button*); "toggle" (estilo *checkbox*); o "pulldown" (estilo desplegable, embebiendo otro menú).
    - **visibleWhen:** permite definir un criterio o un conjunto de ellos que serán utilizados por el IDE para determinar si debe mostrar o no el componente visual en ese momento.

La figura A7.4 muestra la incorporación de este punto de extensión en el desarrollo. Se definió un menú denominado "BPMN2 REST" dentro de la vista de exploración de proyectos (*Project Explorer*), que sólo será visible para los archivos con la extensión *.bpmn*.

```xml
<extension
      point="org.eclipse.ui.menus">
   <menuContribution
         allPopups="true"
         locationURI="popup:org.eclipse.ui.navigator.ProjectExplorer#PopupMenu">
      <menu
            id="org.eclipse.contributions.bpmn2rest.acceleo.module.ui.mainMenu"
            label="BPMN 2 REST">
         <command
               commandId="org.eclipse.contributions.bpmn2rest.acceleo.module.ui.commands.transform"
               id="org.eclipse.contributions.bpmn2rest.acceleo.module.ui.transformOption"
               mnemonic="transform_cmd"
               style="push">
         </command>
         <visibleWhen>
            <with
                  variable="activeMenuSelection">
               <iterate
                     ifEmpty="false">
                  <adapt
                        type="org.eclipse.core.resources.IResource">
                     <test
                           property="org.eclipse.core.resources.extension"
                           value="bpmn">
                     </test>
                  </adapt>
               </iterate>
            </with>
         </visibleWhen>
      </menu>
   </menuContribution>
</extension>
```

Figura A7.4: Fragmento XML que muestra la incorporación del punto de extensión org.eclipse.ui.menus.

A7.5. Punto de extensión org.eclipse.ui.commands

Este punto de extensión es definido por Eclipse IDE y su incorporación permite agregar comandos que serán utilizados en el manejo del ciclo de vida de la interfaz del usuario por parte del IDE ([65]). Un comando, representa una acción que realiza un usuario en algún momento durante la utilización del IDE. Normalmente, un comando es definido primero, y luego es asociado a un evento de la interfaz del usuario. Un mismo comando puede estar asociado a varios botones de la interfaz del usuario, aunque un botón sólo se asocia a uno.

Los elementos que define este punto de extensión son los siguientes:

- **command:** representa la definición de un comando. Pueden definirse varios comandos a la vez.
  - **id (atributo):** define un ID que identifica de manera unívoca al comando.
  - **categoryId (atributo):** referencia a una categoría de comandos definida. Permite ubicar un comando dentro de una categoría.
  - **name:** asigna una etiqueta al comando que será visualizada en los menúes que lo utilicen como una opción.
- **category:** define una nueva categoría de comandos. Su uso general es el de agrupar comandos.
  - **id (atributo):** define un ID que identifica de manera unívoca a la categoría.
  - **name:** asigna un nombre para la categoría.

La figura A7.5 muestra la incorporación de este punto de extensión para la definición del comando "Run BPMN 2 REST". La selección de esta opción desde el menú contextual, permite ejecutar la transformación BPMN2REST sobre un modelo extBPMN.

```
<extension
      point="org.eclipse.ui.commands">
   <command
         categoryId="bpmn2rest.commands.category"
         id="org.eclipse.contributions.bpmn2rest.acceleo.module.ui.commands.transform"
         name="Run BPMN 2 REST">
   </command>
   <category
         id="bpmn2rest.commands.category"
         name="BPMN2REST commands">
   </category>
</extension>
```

Figura A7.5: Fragmento XML que muestra la incorporación del punto de extensión org.eclipse.ui.commands.

A7.6. Punto de extensión org.eclipse.ui.handlers

Este punto de extensión es definido por Eclipse IDE y generalmente se utiliza con el anterior. Permite asociar un manejador o *handler* a cada comando definido, que será responsable de la ejecución del mismo ([66]). El *handler* es invocado por el *core* del IDE al dispararse su comando asociado ante la selección de alguna opción en la interfaz del usuario.

Los elementos que define este punto de extensión son:

- **handler:** define un *handler* y lo asocia a un comando (o varios).
  - **class (atributo):** clase Java que implementa el *handler*. Debe extender a *org.eclipse.core.commands.AbstractHandler.*
  - **commandId (atributo):** referencia al comando definido previamente en el punto de extensión *org.eclipse.ui.commands.*

La figura A7.6 muestra la incorporación de este punto de extensión en el desarrollo, para definir un *handler* para el comando "Run BPMN 2 REST".

```
<extension
      point="org.eclipse.ui.handlers">
   <handler
         class="org.eclipse.contributions.bpmn2rest.acceleo.module.ui.commands.TransformBpmn2RestHandler"
         commandId="org.eclipse.contributions.bpmn2rest.acceleo.module.ui.commands.transform">
   </handler>
</extension>
```

Figura A7.6: Fragmento XML que muestra la incorporación del punto de extensión org.eclipse.ui.handlers.

A8. Clases servicio utilizadas en las consultas del módulo *queries.mtl*

Para implementar consultas en Acceleo utilizando la tecnología Java, es necesario definir clases del tipo servicio (ver sección 2.5.6). Estas clases serán instanciadas e inyectadas en las consultas definidas a fin de ejecutar el código Java que contienen. Para ello, las consultas deben invocar a dichas clases. Durante el desarrollo de las consultas, se definieron dos clases servicio:

- BPMN2ServiceFacade: clase utilizada para obtener los valores de los atributos añadidos en extBPMN, mediante el uso de la API de EMF para navegar instancias de metamodelos.
- CommonHelper: clase utilizada para realizar operaciones con cadenas de texto (strings) a lo largo de los distintos módulos de la transformación.

En la tabla A8.1 se presentan las clases servicio empleadas por las consultas definidas en el módulo *queries.mtl*. Se muestran los métodos invocados en las consultas implementadas en la tecnología Java.

Signatura del método	Descripción	Clase servicio que lo define	Consulta o función que lo invoca
String getServiceHttpMethod(Operation op)	Devuelve el método HTTP definido en *op.rest_method*. Si el valor del atributo es inválido, se utiliza el valor "get". El string devuelto estará en minúsculas.	BPMN2ServiceFacade	getServiceHttpMethod
String getServiceUrl(Operation op, Participant p)	Devuelve la URL de acceso al servicio representado por *op*. Si el valor del atributo *op.rest_prepend_controller* es *true*, entonces se utiliza el atributo *p.name* como prefijo de la URL.	BPMN2ServiceFacade	getServiceUrl
String toUpperCamelCase(String s)	Devuelve la versión *upperCamelCase* del string *s*.	CommonHelper	toUpperCamelCase
String toJavaMethod(String s)	Formatea el string *s* para que pueda ser utilizado como un nombre de método Java válido.	CommonHelper	toJavaMethod

Tabla A8.1: Clases servicio empleadas por las consultas del módulo queries.mtl.

A9. Manual de instalación del *plugin* BPMN2REST para Eclipse IDE

Los siguientes pasos deben ser seguidos para instalar correctamente el complemento BPMN2REST en Eclipse IDE:

1. El *plugin* debe ser instalado sobre una versión de Eclipse IDE igual o posterior a la 4.13. El nombre del producto es *Eclipse IDE for Java Developers* y la versión 4.13 se conoce como 2019/06.
2. Instalar el *plugin* denominado *Eclipse BPMN2 Modeler* para el IDE. La versión del mismo debe ser mayor o igual a la 1.5.0. Adicionalmente, se recomienda instalar la característica de ejemplos, incorporada en el *plugin* de BPMN2 Modeler.
3. Instalar el *plugin* BPMN2REST. Para ello, descargar el archivo *install-bpmn2rest.zip* desde el repositorio alojado en la URL *https://github.com/perlucas/bpmn2rest*. Extraer el comprimido en un directorio, e instalar el *plugin* desde Eclipse IDE utilizando la herramienta de instalación desde repositorios locales, accesible en *Help > Install New Software*.
4. Luego de instaladas las características, debe reiniciarse el IDE para que se apliquen los cambios. Para ello, seleccionar la opción *File > Restart*.

Referencias

[1] M. B. Juric y K. Pant, *Business Process Driven SOA using BPMN and BPEL*. Packt Publishing, 2008.

[2] Object Management Group, «Business Process Model and Notation, version 2.0», inf. téc., ene. de 2011. Disponible en: http://www.omg.org/spec/BPMN/2.0.

[3] M. P. Papazoglou, «Service-oriented computing: Concepts, characteristics and directions», *Proceedings -4th International Conference on Web Information Systems Engineering, WISE 2003*, 2003.

[4] R. Fielding, «Architectural Styles and the Design of Network-based Software Architectures», 2000. Disponible en: https://www.ics.uci.edu/~fielding/pubs/dissertation/fielding_dissertation.pdf.

[5] M. Chicaiza, *Herramientas CASE*. Disponible en: http://marcochicaiza72.blogspot.com/p/herramientas-case.html (Accedido el 19/03/2020).

[6] C. Pons, R. Giandini y G. Pérez, *Desarrollo de Software dirigido por modelos: Conceptos teóricos y su aplicación práctica*. mar. de 2010.

[7] C. A. Martínez, *Transformación de Modelos de Procesos del Negocio BPMN 2.0 a Componentes de la Capa del Negocio Java*, Universidad Nacional de San Luis, 2015.

[8] OMG, «MDA Guide (rev. 2.0)», inf. téc., 2014. Disponible en: http://www.omg.org/docs.

[9] P. Bianco, R. Kotermanski y P. Merson, «Evaluating a Service-Oriented Architecture», 2007. Disponible en: https://resources.sei.cmu.edu/library/asset-view.cfm?assetid=8443.

[10] M. Fowler y J. Lewis, «Microservices», 2014. Disponible en: https://martinfowler.com/articles/microservices.html (Accedido el 03/04/2020).

[11] C. Richardson, *What are microservices?* Disponible en: https://microservices.io/ (Accedido el 03/04/2020).

[12] MDN, *MIME types (IANA media types)*. Disponible en: https://developer.mozilla.org/en-US/docs/Web/HTTP/Basics_of_HTTP/MIME_types (Accedido el 03/04/2020).

[13] MDN, *Códigos de estado de respuesta HTTP*. Disponible en: https://developer.mozilla.org/es/docs/Web/HTTP/Status (Accedido el 03/04/2020).

[14] W3C, «Extensible Markup Language (XML) 1.1 (Second Edition)», inf. téc. Disponible en: https://www.w3.org/TR/2004/REC-xml11-20040204/ (Accedido el 26/03/2020).

[15] Object Management Group, «Object Constraint Language, version 2.4», inf. téc., feb. de 2014. Disponible en: http://www.omg.org/spec/OCL/2.4.

[16] P. Live, *Cómo crear una API con Spark Java*. Disponible en: https://platzi.com/blog/api-spark-java/ (Accedido el 09/04/2020).

[17] P. Wendel, D. Ase y L. Lofdahl, *Spark Framework: An expressive web framework for Kotlin and Java*. Disponible en: http://sparkjava.com/ (Accedido el 09/04/2020).

[18] P. Wendel, D. Ase y L. Lofdahl, *Documentation - Spark*. Disponible en: http://sparkjava.com/documentation (Accedido el 09/04/2020).

[19] E. Foundation, *About the Eclipse Foundation*. Disponible en: https://www.eclipse.org/org/ (Accedido el 10/04/2020).

[20] E. Foundation, *Help - Eclipse Platform - Platform Plug-in Developer Guide*. Disponible en: https://help.eclipse.org/2020-03/nav/2_0 (Accedido el 10/04/2020).

[21] E. Foundation, *Eclipse BPMN2 Modeler*. Disponible en: https://www.eclipse.org/bpmn2-modeler/ (Accedido el 16/04/2020).

[22] E. Foundation, *Eclipse Modeling Project*. Disponible en: https://www.eclipse.org/modeling/emf/ (Accedido el 15/04/2020).

[23] E. Foundation, *Acceleo*. Disponible en: https://www.eclipse.org/acceleo/ (Accedido el 20/11/2019).

[24] E. Foundation, *Acceleo/User Guide*. Disponible en: https://wiki.eclipse.org/Acceleo/User_Guide (Accedido el 20/11/2019).

[25] *Acceleo/OCL Operations Reference.* Disponible en: https://wiki.eclipse.org/Acceleo/OCL_Operations_Reference (Accedido el 20/11/2019).

[26] *Acceleo/Acceleo Operations Reference.* Disponible en: https://wiki.eclipse.org/Acceleo/Acceleo_Operations_Reference (Accedido el 20/11/2019).

[27] J. Bézivin, S. Hammoudi, D. Lopes y F. Jouault, «Applying MDA Approach for Web Service Platform», *Proceedings - IEEE International Enterprise Distributed Object Computing Workshop, EDOC*, 2004, ISSN: 15417719. DOI: 10.1109/EDOC.2004.1342505.

[28] J. M. Vara, V. D. Castro y E. Marcos, «WSDL automatic generation from UML models in a MDA framework», *Proceedings - International Conference on Next Generation Web Services Practices, NWeSP 2005*, 2005. DOI: 10.1109/NWESP.2005.87.

[29] H. Estrada, I. Moralez-Ramírez, A. Martínez y O. Pastor, «From business services to Web services: An MDA approach», *CEUR Workshop Proceedings*, 2010, ISSN: 16130073.

[30] A. Delgado, I. Guzmán y F. Ruiz, «Desarrollo de servicios con SoaML desde procesos de negocio en BPMN : metodología y automatización», *VII Jornadas de Ciencia e Ingeniería de Servicios (JCIS'11)*, 2011.

[31] O. M. Group, «Service Oriented Architecture Modeling Language (SoaML) Specification, version 1.0.1», inf. téc., mayo de 2012. Disponible en: http://www.omg.org/spec/SoaML/1.0.1.

[32] Y. Lemrabet, J. Touzi, D. Clin, M. Bigand y J.-P. Bourey, «Mapping of BPMN models into UML models using SoaML profile», *8th International Conference of Modeling and Simulation - MOSIM'10*, 2010. Disponible en: http://www.enim.fr/mosim2010/articles/312.pdf.

[33] Y. Rhazali, Y. Hadi y A. Mouloudi, «CIM to PIM transformation in MDA: From service-oriented business models to web-based design models», *International Journal of Software Engineering and its Applications*, 2016, ISSN: 17389984. DOI: 10.14257/ijseia.2016.10.4.13.

[34] C. Ariste, J. Ponisio, L. Nahuel y R. S. Giandini, «Diseñando Transformaciones de Modelos CIM/PIM: desde un enfoque de negocio hacia un enfoque de sistema», *Simposio Argentino de Ingeniería de Software (ASSE 2015)-JAIIO 44 (Rosario, 2015)*, 2015. Disponible en: http://sedici.unlp.edu.ar/bitstream/handle/10915/52403/Documento_completo.pdf-PDFA.pdf?sequence=1&isAllowed=y.

[35] Y. Rhazali, Y. Hadi y A. Mouloudi, «A new methodology CIM to PIM transformation resulting from an analytical survey», *MODELSWARD 2016 - Proceedings of the 4th International Conference on Model-Driven Engineering and Software Development*, 2016. DOI: 10.5220/0005690102660273.

[36] A. Kriouile, «An MDA Method for Automatic Transformation of Models from CIM to PIM», *American Journal of Software Engineering and Applications*, 2015, ISSN: 2327-2473. DOI: 10.11648/j.ajsea.20150401.11.

[37] D. Rossi, «UML-based model-driven REST API development», *WEBIST 2016 - Proceedings of the 12th International Conference on Web Information Systems and Technologies*, 2016. DOI: 10.5220/0005906001940201.

[38] H. Ed-Douibi, J. L. C. Izquierdo, A. Gómez, M. Tisi y J. Cabot, «EMF-REST: Generation of RESTful APIs from models», *Proceedings of the ACM Symposium on Applied Computing*, 2016. DOI: 10.1145/2851613.2851782.

[39] R. C. d. C. Gonçalves e I. Azevedo, «RESTful Web Services Development With a Model-Driven Engineering Approach», 2018. DOI: 10.4018/978-1-5225-7455-2.ch009.

[40] E. Foundation, *Xtend - Modernized Java.* Disponible en: https://www.eclipse.org/xtend/ (Accedido el 23/05/2020).

[41] C. Zolotas, T. Diamantopoulos, K. C. Chatzidimitriou y A. L. Symeonidis, «From requirements to source code: a Model-Driven Engineering approach for RESTful web services», *Automated Software Engineering*, 2017, ISSN: 15737535. DOI: 10.1007/s10515-016-0206-x.

[42] A. da Silva de la Cruz, *Una aproximación MDA para la conversión entre servicios web SOAP y RESTful*, Universidad Complutense, Madrid, España, 2013.

[43] E. Ormeño, M. Lund, L. Aballay y S. Aciar, «An UML profile for modeling RESTful services», *13th Argentine Symposium on Software Engineering, ASSE 2012*, 2012, ISSN: 18502792.

[44] V. Schreibmann, «Design and Implementation of a Model Driven Approach for RESTful APIs», 2014. Disponible en: https://www.semanticscholar.org/paper/Design-and-Implementation-of-a-Model-Driven-for-Schreibmann/38852c6290ace48ff4e3ee3e7646e6e4058ca16b.

[45] A. Delgado, L. González, S. Larroca, A. Pastorini, F. Ruiz e I. G.-R. D. Guzmán, «SoaML Eclipse plug-in para modelado de servicios», *XVI Jornadas de Ingeniería del Software y Bases de Datos*, 2011.

[46] *Free & Commercial BPMN 2 Tools for Agile BPM.* Disponible en: https://bpmntools.com/ (Accedido el 16/08/2019).

[47] *Free BPMN modelling tools - 2018 edition.* Disponible en: https://bpmtips.com/free-bpmn-modelling-tools-2018-edition/ (Accedido el 16/08/2019).

[48] *Best free and open source Business process management tools*. Disponible en: https://www.saasworthy.com/blog/free-and-open-source-bpm-tools-to-effectively-manage-your-business-processes (Accedido el 16/08/2019).

[49] *The Top 15 Free and Open Source BPM Software*. Disponible en: https://solutionsreview.com/business-process-management/the-top-15-free-and-open-source-bpm-software/ (Accedido el 16/08/2019).

[50] *Top 18 Free And Open Source BPM Software for Businesses*. Disponible en: https://www.softwaresuggest.com/blog/top-free-open-source-bpm-software/ (Accedido el 16/08/2019).

[51] *BPMN2-Modeler/DeveloperTutorials/ModelExtension*. Disponible en: https://wiki.eclipse.org/BPMN2-Modeler/DeveloperTutorials/ModelExtension (Accedido el 07/11/2019).

[52] *BPMN2-Modeler/DeveloperTutorials/CustomPropertyTabs*. Disponible en: https://wiki.eclipse.org/BPMN2-Modeler/DeveloperTutorials/CustomPropertyTabs (Accedido el 07/11/2019).

[53] *BPMN2-Modeler/DeveloperTutorials*. Disponible en: https://wiki.eclipse.org/BPMN2-Modeler/DeveloperTutorials (Accedido el 07/11/2019).

[54] *BPMN2-Modeler/DeveloperTutorials/ExtendingRuntime*. Disponible en: https://wiki.eclipse.org/BPMN2-Modeler/DeveloperTutorials/ExtendingRuntime (Accedido el 07/11/2019).

[55] Codecamy, *MVC: Model, View, Controller*. Disponible en: https://www.codecademy.com/articles/mvc (Accedido el 29/05/2020).

[56] *BPMN2-Modeler/DeveloperTutorials/Validation*. Disponible en: https://wiki.eclipse.org/BPMN2-Modeler/DeveloperTutorials/Validation (Accedido el 07/11/2019).

[57] *Wrapper Definition*. Disponible en: https://techterms.com/definition/wrapper (Accedido el 02/03/2020).

[58] *Singleton Design Pattern*. Disponible en: https://sourcemaking.com/design_patterns/singleton (Accedido el 02/03/2020).

[59] F. Corradini, A. Morichetta, A. Polini, B. Re y F. Tiezzi, «Collaboration vs Choreography Conformance in BPMN 2.0: from Theory to Practice», *22nd IEEE Enterprise Distributed Object Computing Conference (EDOC 2018)*, 2018.

[60] OMG, «MOF Model to Text Transformation Language, v1.0», inf. téc., ene. de 2008. Disponible en: http://www.omg.org/spec/MOFM2T/1.0/PDF.

[61] *BPMN2 Modeler Runtime Specialization*. Disponible en: https://download.eclipse.org/bpmn2-modeler/doc/api/core.html (Accedido el 05/12/2019).

[62] *Eclipse EMF Validation Framework - EMF Model Validation Constraint Providers*. Disponible en: https://www.linuxtopia.org/online_books/eclipse_documentation/eclipse_emf_validation_framework_developer_guide/topic/org.eclipse.emf.validation.doc/references/extension-points/eclipse_emf_validation_framework_org_eclipse_emf_validation_constraintProviders.html (Accedido el 10/12/2019).

[63] *Eclipse EMF Validation Framework - EMF Validation Constraint Bindings*. Disponible en: https://www.linuxtopia.org/online_books/eclipse_documentation/eclipse_emf_validation_framework_developer_guide/topic/org.eclipse.emf.validation.doc/references/extension-points/eclipse_emf_validation_framework_org_eclipse_emf_validation_constraintBindings.html (Accedido el 10/12/2019).

[64] *Help - Eclipse Platform (org.eclipse.ui.menus)*. Disponible en: https://help.eclipse.org/2019-12/index.jsp?topic=%2Forg.eclipse.platform.doc.isv%2Fguide%2Fworkbench_cmd_menus.htm (Accedido el 10/01/2020).

[65] *Help - Eclipse Platform (org.eclipse.ui.commands)*. Disponible en: https://help.eclipse.org/2019-12/index.jsp?topic=%2Forg.eclipse.platform.doc.isv%2Freference%2Fextension-points%2Forg_eclipse_ui_commands.html (Accedido el 10/01/2020).

[66] *Help - Eclipse Platform (org.eclipse.ui.handlers)*. Disponible en: https://help.eclipse.org/2019-12/index.jsp?topic=%2Forg.eclipse.platform.doc.isv%2Fguide%2Fworkbench_cmd_handlers.htm (Accedido el 10/01/2020).