UNIVERSIDAD POLITÉCNICA DE CARTAGENA
E. T. S. Ingeniería de Telecomunicaciones

DISTRIBUCIÓN DE UN ENTORNO DE MODELADO UTILIZANDO SERVICIOS WEB

José Fermín Díaz Amado

Título del Proyecto

DISTRIBUCIÓN DE UN ENTORNO DE MODELADO UTILIZANDO SERVICIOS WEB

Autor

José Fermín Díaz Amado

Titulación

Grado en Ingeniería Telemática

Directores

Francisca Rosique Contreras
Juan Francisco Inglés Romero

Defensa

En Cartagena, Mayo de 2014

ÍNDICE

CAPÍTULO I

Introducción

Actualmente, el Desarrollo de Software Dirigido por Modelos (DSDM) [1] representa uno de los paradigmas de desarrollo software más en boga en el ámbito de la Ingeniería del Software. Las tecnologías entorno a este nuevo enfoque ofrecen una aproximación prometedora para superar las limitaciones expresivas de los lenguajes de programación de tercera generación, permitiendo a los diseñadores describir sistemas cada vez más complejos de manera más simple, gracias a la utilización de conceptos propios de sus dominios de aplicación. El DSDM busca, por lo tanto, elevar el nivel de abstracción utilizado durante las distintas etapas del ciclo de vida del software.

El entorno de modelado HuRoME (Humanoid Robot Modeling Environment) [2,3] integra un conjunto de herramientas diseñadas para facilitar el modelado de coreografías y la modernización del software existente para el robot humanoide Robonova [4]. HuRoME permite a los numerosos usuarios de Robonova: (1) modelar gráficamente y validar formalmente las secuencias de movimientos del robot (coreografías), (2) generar automáticamente la implementación asociada a cada coreografía en el lenguaje RoboBASIC, y (3) modernizar y reutilizar el software ya existente, permitiendo la obtención de los modelos equivalentes a cualquier programa RoboBASIC existente.

El entorno HuRoME ofrece una aproximación al desarrollo de software para robótica utilizando un enfoque de Desarrollo de Software Dirigido por Modelos. HuRoME no sólo facilita la generación automática de código a partir de modelos, sino también al proceso inverso, permitiendo la transformación de programas ya existentes (legacy code) en modelos, facilitando así su tratamiento (depuración, extensión, reutilización, análisis, simulación, etc.) con un nivel de abstracción mayor que el que proporciona el código fuente. No obstante, HuRoME, como la

mayoría de herramientas de DSDM, ofrece un despliegue tradicional, tratándose de una herramienta que se ejecuta de forma local en un ordenador. La distribución en servidores de las diferentes transformaciones de modelos que componen HuRoME (por ejemplo, para la generación automática de código RoboBASIC) facilitaría, entre otros aspectos, el mantenimiento y la extensión de estas transformaciones. La gestión, actualización, así como la incorporación de nuevas transformaciones de modelos serían procesos transparentes a los usuarios. Por otro lado, un esquema distribuido potencia el trabajo colaborativo entre usuarios. El despliegue de las trasformaciones de modelos como servicios distribuidos junto con un almacenamiento compartido, por ejemplo, en la "nube", compondrían los pilares de cualquier entorno colaborativo de DSDM.

El presente Proyecto trata de ilustrar las posibilidades de los Servicios Web para distribuir e integrar entornos de modelado basados en el DSDM. En concreto, este Proyecto utilizará como ejemplo la herramienta HuRoME dirigida al modelado y generación automática de código para el robot Robonova.

1.1 Objetivos

Los objetivos que se pretenden abordar en el presente Proyecto son los siguientes:

a) Aprender los fundamentos de los Servicios Web y su desarrollo con el lenguaje Java en Eclipse. El Proyecto presenta dos de las tecnologías más importantes que se utilizan para implementar Servicios Web, esto es, SOAP y RESTful. El Proyecto se desarrollará utilizando el lenguaje Java en el entorno Eclipse.

b) Conocer el entorno de modelado HuRoME.

c) Implementar una infraestructura software para distribuir transformaciones de modelos como Servicios Web. Se trata del objetivo principal del Proyecto. Este desarrollo será aplicado a la herramienta HuRoME.

d) Evaluar y extraer las conclusiones.

1.2 Fases de desarrollo del Proyecto

El desarrollo de este Proyecto se ha llevado a cabo siguiendo las etapas que se resumen a continuación:

a) Estudio de los fundamentos del desarrollo de Servicios Web en Java. En esta fase se abordan los fundamentos básicos del presente Proyecto, lo que incluye, entre otros aspectos, las APIs para la implementación de Servicios Web en Java, configuración del servidor de aplicaciones Tomcat y el manejo de las herramientas de desarrollo Eclipse***.***

b) Estudio del entorno de modelado HuRoME.

c) Diseño del Servicio Web para ejecutar transformaciones de modelos. En esta fase se desarrolla una interfaz independiente del dominio de aplicación para los Servicios Web.

d) Implementación de la infraestructura software genérica del Servicio Web. En esta fase se pretende implementar los procesos de serialización/deserialización de datos, así como el código para soportar la comunicación del Servicio Web.

e) Distribución de las transformaciones de modelos de la herramienta HuRoME usando Servicios Web. En esta fase, se aplica a la herramienta huRoME el desarrollo realizado en las fases anteriores. La sencillez de esta herramienta permite ilustrar y validar la propuesta presentada en el presente Proyecto. Por otro lado, en esta fase se abordará la implementación del Servicio Web utilizando dos tecnologías diferentes, SOAP y RESTful, lo que permite contrastar las ventajas y desventajas de ambas.

f) Evaluación, extracción de conclusiones y redacción de la memoria.

1.3 Organización de la memoria

El resto de la memoria se organiza como se indica a continuación:

Capítulo 2: Breve revisión del los Servicios Web SOAP y RESTful

Capítulo 3: Revisión de la herramienta HuRoME y su uso en el desarrollo de aplicaciones robóticas

Capítulo 4: Desarrollo de un Servicio Web para mejorar la distribución y operatividad de la herramienta HuRoME.

Capítulo 5: Pruebas y resultados.

Capítulo 6: Conclusiones y trabajos futuros.

Referencias

Anexo I: Código del Servicio Web SOAP implementado

Anexo II: Código del Servicio Web REST implementado

CAPÍTULO II

Fundamentos de los Servicios Web

En este capítulo se realiza una introducción a los fundamentos de los Servicios Web, reseñando históricamente sus orígenes, la arquitectura y las tecnologías que permiten su implementación en Java.

2.1 Definición de Servicio Web

Según el consorcio *World Wide Web* [5], los Servicios Web son sistemas software diseñados para soportar una interacción interoperable maquina a máquina sobre una red. Los Servicios Web suelen ser APIs Web que pueden ser accedidas dentro de una red (principalmente Internet) y son ejecutados en el sistema que los aloja. Así, un Servicio Web es una tecnología que utiliza un conjunto de protocolos y estándares que sirven para intercambiar datos entre aplicaciones. Estas aplicaciones pueden estar desarrolladas en lenguajes de programación diferentes y ser ejecutadas sobre plataformas diferentes. La interoperabilidad se consigue mediante la adopción de estándares abiertos.

2.2 Arquitectura básica de los Servicios Web

La definición de Servicio Web introducida anteriormente puede albergar muchos tipos diferentes de sistemas, no obstante, el caso común de uso se suele referir a clientes y servidores que se comunican mediante mensajes. En esta sección presentamos los estilos de uso más importantes de los Servicios Web.

2.2.1. Llamadas a Procedimientos Remotos

Los Servicios Web basados en RPC (Remote Procedure Calls) presentan una interfaz de llamada a procedimientos y funciones distribuidas. Así, la unidad básica de este tipo de servicios es la operación. Este estilo suele resultar sencillo a los desarrolladores de software dado que se trata de una visión ampliamente extendida en los lenguajes de programación tradicionales (el concepto de invocar a una función para realizar una operación está muy asumido). Además de su sencillez conceptual, se trata de un estilo muy extendido puesto que las primeras herramientas para Servicios Web (también llamadas de la primera generación) estaban centradas en esta visión. Sin embargo, ha sido algunas veces criticado por no ser débilmente acoplado, ya que suele ser implementado por medio del mapeo de servicios directamente a funciones específicas del lenguaje o llamadas a métodos.

Las RPC son muy utilizadas dentro del paradigma cliente-servidor. Siendo el cliente el que inicia el proceso solicitando al servidor que ejecute cierto procedimiento o función y enviando éste de vuelta el resultado de dicha operación al cliente. Hay distintos tipos de RPC, muchos de ellos estandarizados como pueden ser el RPC de Sun denominado ONC RPC [6], el RPC de OSF denominado DCE/RPC [7] y el Modelo de Objetos de Componentes Distribuidos de Microsoft DCOM [8], aunque ninguno de éstos es compatible entre sí. La mayoría de ellos utilizan un lenguaje de descripción de interfaz (*Interface Definition Language*, IDL) que define los métodos exportados por el servidor. Hoy en día se está utilizando el XML como lenguaje para definir el IDL y el HTTP como protocolo de red, dando lugar a lo que se conoce como Servicios Web. Ejemplos de éstos pueden ser XML-RPC [9].

2.2.2. REST

REST (Representational State Transfer) es un estilo de arquitectura de software para sistemas hipermedias distribuidos tales como la Web. El término fue introducido en la tesis doctoral de Roy Fielding [10], quien es uno de los principales autores de la especificación de HTTP.

En realidad, REST se refiere estrictamente a una colección de principios para el diseño de arquitecturas en red. Estos principios resumen cómo los recursos son definidos y diseccionados. El término frecuentemente es utilizado en el sentido de describir a cualquier interfaz que transmite datos específicos de un domino sobre HTTP sin una capa adicional, como hace SOAP. Estos dos significados pueden chocar o incluso solaparse. Es posible diseñar un sistema software de gran tamaño de acuerdo con la arquitectura propuesta por Fielding sin utilizar HTTP o sin interactuar con la Web. Así como también es posible diseñar una simple interfaz XML+HTTP que no sigue los principios REST, y en cambio seguir un modelo RPC.

Cabe destacar que REST no es un estándar, ya que es tan sólo un estilo de arquitectura. Aunque REST no es un estándar, está basado en estándares como HTTP, URL, representación de recursos (XML, HTML, GIF, JPEG…), tipos MIME (text/html, text/xml). La sección 2.4 abordará REST con mayor profundidad.

2.2.3. Arquitectura Orientada a Servicios

Los Servicios Web pueden ser implementados siguiendo los conceptos de la arquitectura SOA (Service-oriented Architecture). Este estilo se centra en torno al mensaje como unidad básica de comunicación, más que en la operación como pasaba con RPC. Así, típicamente se suele hablar

de servicios orientados a mensajes. Los Servicios Web basados en SOA reciben una gran aceptación por la mayor parte de desarrolladores de software y analistas. Al contrario que los Servicios Web basados en RPC, este estilo es débilmente acoplado, lo cual es preferible ya que se centra en el "contrato" proporcionado por la especificación de las interfaces, más que en los detalles de implementación subyacentes.

Las soluciones SOA han sido creadas para diseñar y desarrollar sistemas distribuidos que satisfagan objetivos de negocio, como facilidad y flexibilidad de integración con sistemas legados, alineación directa a los procesos de negocio reduciendo costes de implementación, innovación de servicios a clientes y una adaptación ágil ante cambios incluyendo reacción temprana ante la competitividad. Así, SOA permite la creación de sistemas de información altamente escalables que reflejan el negocio de la organización, a su vez brinda una forma bien definida de exposición e invocación de servicios (comúnmente pero no exclusivamente Servicios Web), lo cual facilita la interacción entre diferentes sistemas propios o de terceros.

SOA define las siguientes capas de software:

- **Aplicaciones básicas**. Sistemas desarrollados bajo cualquier arquitectura o tecnología, geográficamente dispersos y bajo cualquier figura de propiedad.

- **De exposición de funcionalidades**. Donde las funcionalidades de la capa aplicativa son expuestas en forma de servicios (generalmente como servicios web).

- **De integración de servicios.** Facilitan el intercambio de datos entre elementos de la capa aplicativa orientada a procesos empresariales internos o en colaboración.

- **De composición de procesos**. Que define el proceso en términos del negocio y sus necesidades, y que varía en función del negocio.

- **De entrega.** Donde los servicios son desplegados a los usuarios finales.

La metodología de modelado y diseño para aplicaciones SOA se conoce como análisis y diseño orientado a servicios. La arquitectura orientada a servicios no es una tecnología concreta sino, más bien, es un marco de trabajo para el desarrollo de software. Así pues, para que un proyecto SOA tenga éxito los desarrolladores deben seguir la mentalidad de crear servicios comunes que son orquestados por clientes o middleware para implementar los procesos de negocio. El desarrollo de sistemas usando SOA requiere un compromiso con este modelo en términos de planificación, herramientas e infraestructura.

Existen diversos estándares relacionados a los Servicios Web, como XML, HTTP, SOAP, REST, WSDL o UDDI. Hay que considerar, sin embargo, que un sistema SOA no necesita utilizar estos estándares para ser "Orientado a Servicios" pero normalmente es altamente recomendable su uso.

2.3 Desarrollo de Servicios Web con SOAP

SOAP (Simple Object Access Protocol) [11] es un protocolo estándar que define cómo dos objetos en diferentes procesos pueden comunicarse por medio de intercambio de datos XML. SOAP puede formar la capa base de una "pila de protocolos servicios web", ofreciendo un framework de mensajería básica en la cual los Servicios Web se puedan construir. Básicamente, este protocolo basado en XML consiste de tres partes: (1) un contenedor (*envelope*), el cual define qué hay en el mensaje y cómo procesarlo; (2) un conjunto de reglas de codificación para expresar instancias de tipos de datos; y (3) una convención para representar llamadas a procedimientos y respuestas. Por otro lado, las principales características del protocolo SOAP son:

- **Extensibilidad,** seguridad y WS-routing son extensiones aplicadas en el desarrollo.
- **Neutralidad,** SOAP puede ser utilizado sobre cualquier protocolo de transporte como HTTP, SMTP, TCP o JMS.
- **Independencia,** SOAP permite cualquier modelo de programación.

Como ejemplo de cómo los procedimientos SOAP pueden ser utilizados, un mensaje SOAP podría ser enviado a un sitio Web que tiene habilitado servicio web, para realizar la búsqueda de algún dato en una base de datos, indicando los parámetros necesitados en la consulta. El sitio podría retornar un documento formateado en XML con el resultado. Teniendo los datos de respuesta en un formato estandarizado "parseable", este puede ser integrado directamente en un sitio Web o aplicación externa.

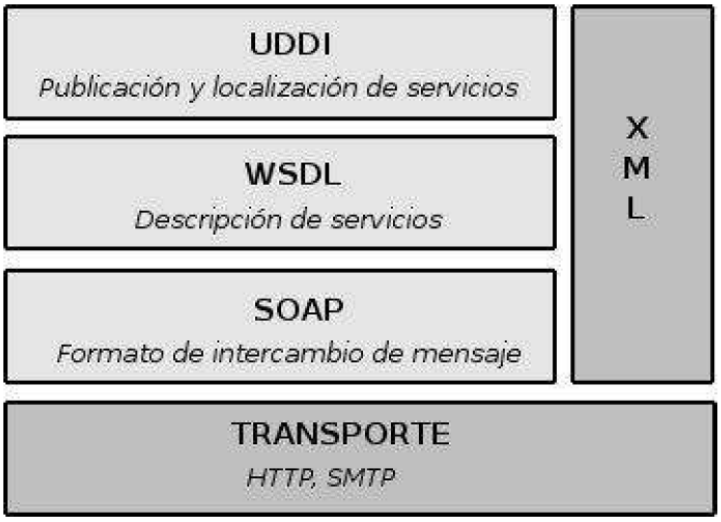

Figura 2.1.: Arquitectura Servicios Web SOAP (fuente [12])

La Figura 2.1 muestra los componentes típicos de un Servicio Web basado en SOAP, éstos son:

- **XML (eXtensible Markup Language)**, es un estándar para la definición de lenguajes de marcas, flexible y extensible, usado en los Servicios Web para especificar lenguajes y protocolos necesarios. Permite definir lenguajes para describir los servicios y representar los mensajes intercambiados.

- **SOAP (Simple Object Acces Protocol)**, Mecanismo de interacción entre extremos que surge a partir de la necesidad de un formato de mensajes neutro, abierto y extensible. La representación de los mensajes de invocación (argumentos) y respuesta como documentos XML. Especifica el modo de interacción, RCP (síncrono) o petición (asíncrono). Los mensajes se mapean en el protocolo de transporte (HTTP, SMTP).
- **WSDL (Web Service Description Language)**, lenguaje común para describir los servicios. La descripción de los servicios y sus interfaces se realiza de forma estándar mediante documentos XML, incluyendo toda la información necesaria para suplir un middleware común centralizado. Especifica las operaciones disponibles, con los parámetros de entrada y de salida. Puede usarse para generar los stubs/skeleton y las capas intermedias necesarias para escribir clientes que invoquen los Servicios Web y servidores que los implementen.
- **UDDI (Universal Description, Discovery and Integration)**, Publicación y localización de servicios. La descripción de los servicios (WDSDL) se almacena en un directorio de servicios, UDDI especifica cómo se publican y descubren los servicios y como trabajan los directorios de servicios web. El servidor da del alta los servicios (WSDL + descripción) y el cliente descubre servicios (WSDL).

2.3.1 Estructura de los mensajes SOAP

SOAP ofrece soporte para el envío de datos de aplicación arbitrarios, define un "contenedor" de mensajes XML, además no establece restricciones sobre el contenido del mensaje ni sobre el procesamiento a realizar sobre el.

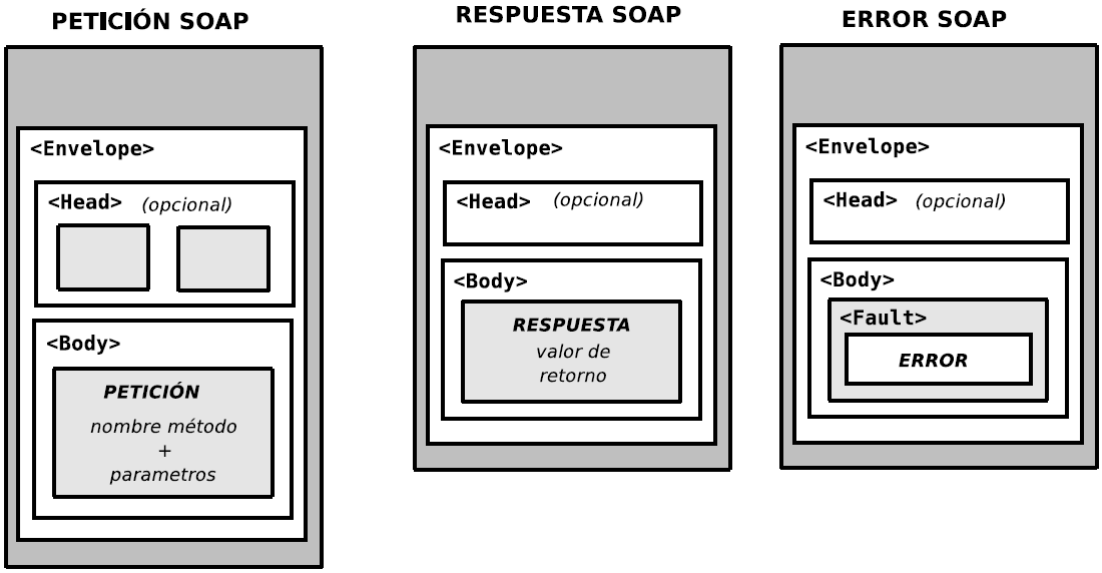

Figura 2.2.: Esquema básico de un mensaje SOAP (fuente [12])

La Figura 2.2 muestra las partes de un mensaje SOAP, éstas son:

- Cabecera (*head*), componente opcional que contiene información sobre el mensaje a usar por la infraestructura de Servicios Web, identificadores de transacciones, información de seguridad, etc.

- Cuerpo (*body*), componente obligatorio que contiene información específica a usar por las aplicaciones que usan o implementan el servicio web, los extremos son los encargados de acordar el formato de información intercambiada y de generar o procesar su contenido. Puede estructurarse en bloques, usualmente los bloques mapean una invocación o respuesta RCP en formato XML, un bloque especial *fault* usado para representar errores en el procesamiento de los mensajes SOAP.

La especificación SOAP es independiente del protocolo de transporte usado para transferir los mensajes, solo define un contenedor de mensajes y la forma de encapsularlos en el protocolo de transporte que se use. Así, la especificación del protocolo de transporte se realiza a través de un binding SOAP éste permite especificar cómo se envía un mensaje SOAP utilizando un protocolo de transporte determinado, incluyendo la definición de las direcciones origen y destino. Por ejemplo, como se puede ver en la Figura 2.3, los mensajes SOAP son enviados a través de peticiones HTTP POST, encapsulados en el cuerpo de la petición http. También se puede observar cómo la respuesta HTTP contiene un mensaje SOAP. Por último, se pueden definir bindings de SOAP para multitud de protocolos, incluyendo el protocolo SMTP (e-mail).

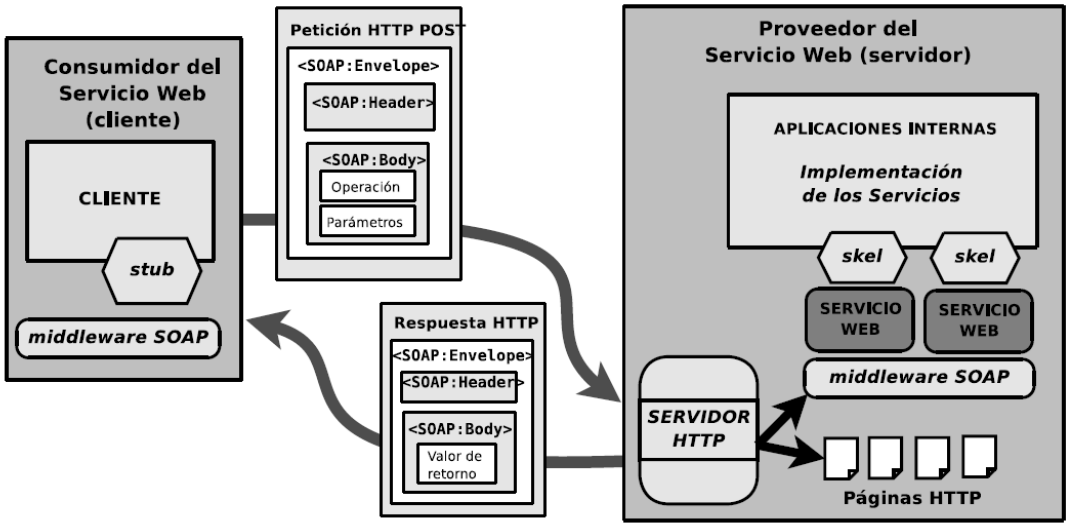

Figura 2.3.: Ejemplo Mensajes SOAP (fuente [12])

2.3.2 Protocolo WSDL, descripción de servicios

Los middlewares convencionales para el desarrollo de sistemas distribuidos (por ejemplo, CORBA) poseen lenguajes de definición de interfaces (Interface Definition Language, IDL), para, entre otros aspectos, especificar los argumentos de entrada y salida de los servicios. La definición de interfaces hace posible que los clientes sepan cómo interactuar con los servicios. A diferencia de estas tecnologías, los Servicios Web necesitan una descripción de interfaces más rica e independiente de la plataforma que, defina operaciones, los mecanismos de interacción y que especifique la localización del servicio (URI). WSDL (Web Service Description Language) [13] se usa como lenguaje de descripción de interfaz del servicio e indica como interactuar con el servicio, definiendo las operaciones a realizar, datos de envío y devolución más el formato de los mensajes con el protocolo de transferencia.

Las aplicaciones no escriben ni procesan directamente los mensajes XML SOAP, sino que a partir de la definición WSDL del Servicio Web se generan los elementos adicionales necesarios para que el servidor implemente el servicio y el cliente realice la llamada.

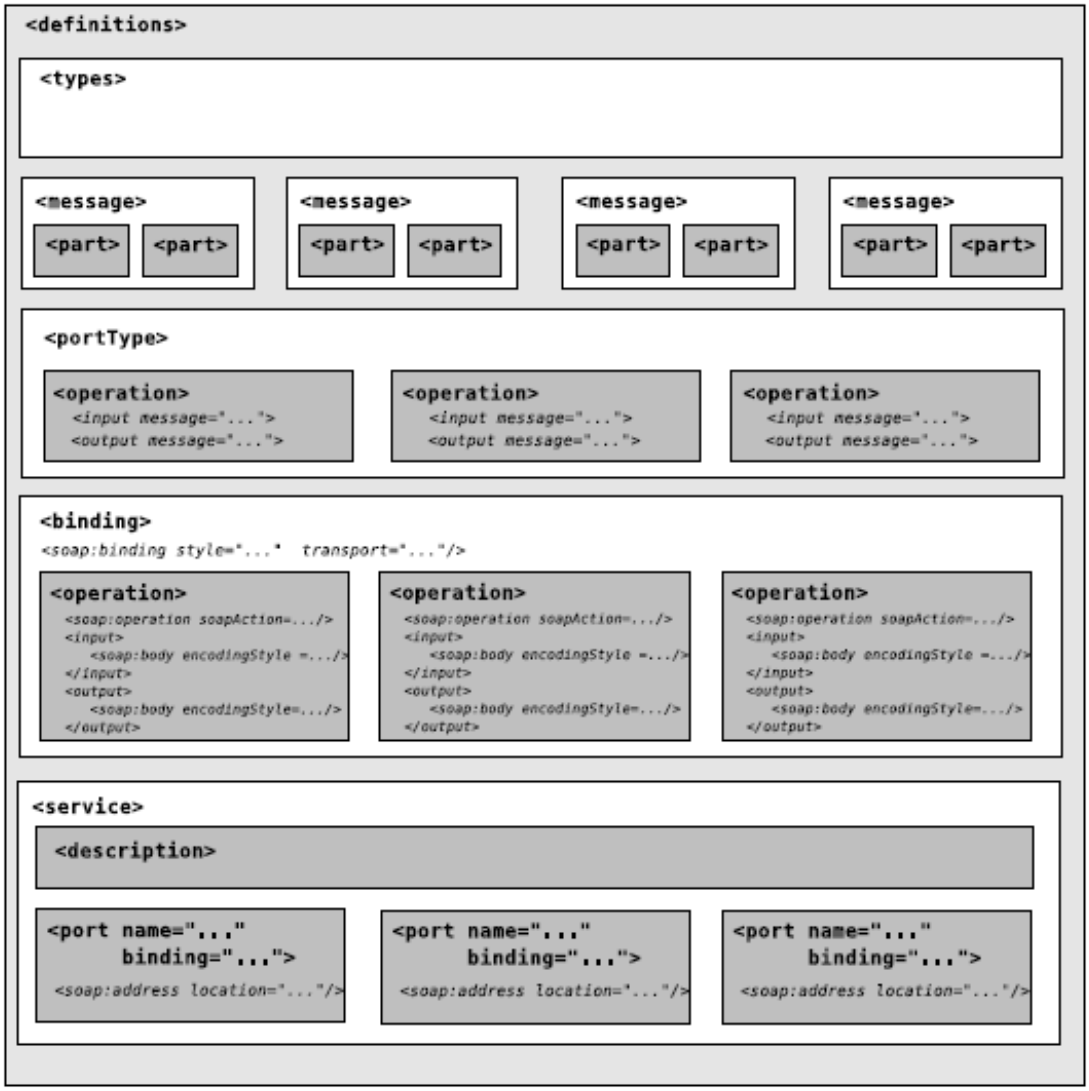

Figura 2.4.: Esquema básico de una discripción WSDL (fuente [12])

EL documento WSDL describe un Servicio Web como una colección de puertos capaces de intercambiar mensajes, en la que cada puerto tiene una definición abstracta (operaciones y mensajes) y una definición concreta (protocolos). A continuación se introducen los elementos más importantes que se pueden encontrar en una especificación WSDL. La Figura 2.4 muestra la disposición de estos elementos en WSDL.

Elemento <definitions>, raíz del documento WSDL, usado para declarar los espacios de nombres

Elemento <types>, define tipos y estructuras de datos intercambiados

Elemento <message>, define los datos que van a ser intercambiados indicando su nombre y contenido, un mensaje para un servicio concreto contendrá un elemento <part> y cada parte estará caracterizada por un nombre y un tipo.

Elemento <portType>, define grupos de operaciones (puertos), para cada operación (elemento <operation>) se le asigna el nombre y se especifica el intercambio de mensajes. El intercambio de mensajes puede ser one-way (cliente invoca servicio con un único mensaje), notifications (servidor envía un mensaje), request-response (servidor recibe petición y responde) y solicit-response (servidor invoca y espera respuesta)

Elemento <bindings>, asocia a un grupo de operaciones (porType) una especificación de la codificación de mensajes y el protocolo de transporte a utilizar. Informa a los usuarios del Servicio Web de los protocolos a usar, de como estructurar los mensajes XML y de lo que se espera recibir al enviar un mensaje. WSDL permite bindings para SOAP, HTTP GET, HTTP POST y MIME (SMTP), en el caso de SOAP como intercambio de mensajes el binding contiene toda la información necesaria para construir y procesar automáticamente los mensajes SOAP.

Elemento <service>, especifica una agrupación lógica de puertos, cada puerto especifica un punto final (endpoint).

2.3.3 Protocolo UDDI, publicación de servicios

Protocolo UDDI (Universal Description, Discovery and Integration) [14] se emplea para desarrollar sistemas de registro de servicios, de manera que se proporcionan operaciones (vía SOAP) para registrar y buscar Servicios Web. Cada servicio se registra con su nombre, una descripción del servicio (URL del WSDL, descripción textual…). El API de UDDI está especificada con WSDL, los servidores publican Servicios Web que son localizados por parte de los clientes. El tipo de información ofrecida es:

Páginas blancas, identificador y dirección de contacto de la organización que publica el Servicio Web.

Páginas amarillas, Descripción de los Servicios Web ofrecidos usando diferentes tipos de categorizaciones.

Páginas verdes, información técnica sobre los Servicios Web (URL de descarga del WSDL).

La Figura 2.5 muestra de manera esquematizada el funcionamiento de un sistema de registro de servicios basado en UDDI. Puede observarse cómo los tres roles principales (registro UDDI, consumidor y proveedor) interactúan para que finalmente el consumidor pueda contactar con los servicios ofrecidos por el proveedor.

2.3.4 Servicios Web SOAP con Java

La creación de Servicios Web basados en SOAP en Java se realiza a través de JAX-WS [15] (Java API for XML Web Services). Se trata de una API que forma parte de la plataforma Java EE de Sun Microsystems. Al igual que las otras APIs de Java EE, JAX-WS utiliza anotaciones, introducidas en Java SE 5, para simplificar el desarrollo y despliegue de los clientes y puntos finales de los Servicios Web. No obstante, algunos framework incorporan algunas herramientas que permiten asistir al programador durante el proceso de desarrollo del

Servicio Web. Estas herramientas permiten la implementación del servicio desde una perspectiva top-down o down-top, abstrayendo en cierta medida los detalles de la API.

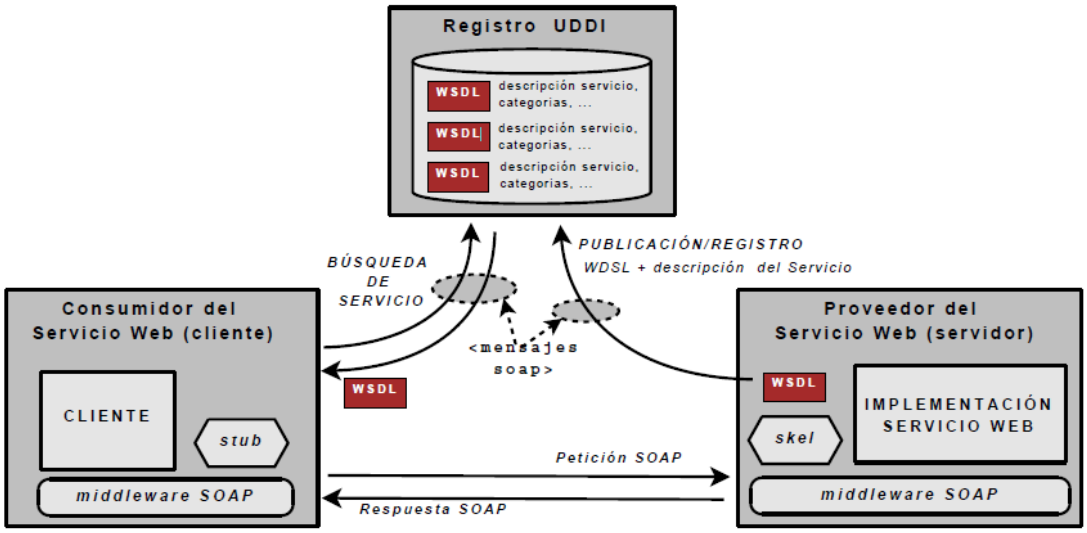

Figura 2.5.: Esquema básico de un sistema de registro UDDI (fuente [12])

La implementación de referencia de JAX-WS se desarrolla como un proyecto de código abierto y forma parte del proyecto GlassFish [16], un servidor de aplicaciones Java EE de código abierto. Se llama JAX-WS RI (por Reference Implementation) y se dice que es la implementación de calidad de producción (contrariamente a la implementación de referencia antigua que era una prueba de concepto). Esta implementación de referencia es ahora parte de la distribución Metro [17].

2.4 Desarrollo de Servicios Web con RESTful

REST (Representational State Transfer) [10] es estilo arquitectural de software para sistemas distribuidos en la World Wide Web. Describe cualquier interfaz Web simple que utiliza XML y HTTP, sin las abstracciones adicionales de los protocolos basados en patrones de intercambio de mensajes como el protocolo de servicios web SOAP. Un concepto importante en REST es la existencia de recursos (elementos de información), que pueden ser accedidos utilizando un identificador global (URI. Uniform Resource Identifier). Para manipular estos recursos, los componentes de la red (clientes y servidores) se comunican a través de una interfaz estándar (HTTP) e intercambian representaciones de estos recursos (ficheros que se descargan y se envían).

La petición puede ser transmitida por cualquier número de conectores (por ejemplo clientes, servidores, cachés, túneles, etc.) pero cada uno lo hace sin "ver más allá" de su propia petición (lo que se conoce stateless (sin estado), otra restricción de REST, que es un principio común con muchas otras partes de la arquitectura de redes y de la información). Así, una aplicación puede interactuar con un recurso conociendo el identificador del recurso y la acción requerida, no necesitando conocer si existen cachés, proxys, cortafuegos, túneles o cualquier otra cosa entre ella y el servidor que guarda la información. La aplicación, sin embargo, debe comprender el

formato de la información devuelta (la representación), que es por lo general un documento HTML o XML, aunque también puede ser una imagen o cualquier otro contenido.

Los principios de REST son los mismos en los que se basa el éxito de la Web como resultado de una serie de diseños fundamentales clave:

- **Un protocolo cliente/servidor sin estado**. Cada mensaje HTTP contiene toda la información necesaria para comprender la petición. Como resultado, ni el cliente ni el servidor necesitan recordar ningún estado de las comunicaciones entre mensajes. Sin embargo, en la práctica, muchas aplicaciones basadas en HTTP utilizan cookies y otros mecanismos para mantener el estado de la sesión (algunas de estas prácticas, como la reescritura de URLs, no son permitidas por REST)

- **Un conjunto de operaciones bien definidas que se aplican a todos los recursos de información.** HTTP en sí define un conjunto pequeño de operaciones, las más importantes son POST, GET, PUT y DELETE.

- **Una sintaxis universal para identificar los recursos**. En un sistema REST, cada recurso es direccionable únicamente a través de su URI.

- **El uso de hipermedios**. tanto para la información de la aplicación como para las transiciones de estado de la aplicación: la representación de este estado en un sistema REST son típicamente HTML o XML

2.4.1 Servicios Web RESTful con Java

Java define el soporte a la tecnología REST a través de la especificación Java Specification Request 311 (JSR 311) [18]. Esta especificación se la conoce con el nombre JAX-RS (*The Java API for RESTful Web Services*). JAX-RS usa anotaciones Java para definir los elementos REST en el código. Las anotaciones más importantes son las siguientes:

- **@PATH(path).** Establece la ruta a la base URL + / your_path. La URL base se basa en el nombre de la aplicación, el servlet y el patrón de URL del archivo de configuración "web.xml ".
- **@POST**. Indica que el siguiente método responderá a una solicitud HTTP POST
- **@GET.** Indica que el siguiente método responderá a una solicitud HTTP GET
- **@PUT.** Indica que el siguiente método responderá a una solicitud HTTP PUT
- **@DELETE.** Indica que el siguiente método responderá a una solicitud HTTP DELETE
- **@Produces(MediaType.TEXT_PLAIN [, MAS TIPOS]).** @ Produce define qué tipo MIME se entrega mediante un método anotado con @ GET
- **@Consumes(type [, mas tipos])**. @Consume define qué tipo MIME se consume por este método.
- **@PathParam.** Se utiliza para inyectar valores de la URL en un parámetro de método. De esta manera se inyecta por ejemplo, el ID de un recurso en el método para obtener el objeto correcto.

Jersey [19] es la implementación de referencia de JAX-RS. Jersey contiene básicamente un servidor y un cliente REST. Así, la librería ofrece la implementación esencial para comunicar clientes y servidores. Además, el servidor (implementado como *servlet* Java) es capaz de localizar los recursos que se sirven a través de REST en el código del programador. Este servlet analiza las peticiones HTTP entrantes y selecciona la clase y método Java correcto para responder a la petición. Esta selección está basada en las anotaciones anteriores dispuestas en las clases y los métodos. Por último, comentar que JAX-RS soporta la creación de mensajes XML y JSON (véase el apartado 2.4.2) a través de la API JAXB [20] (*Java Architecture for XML Binding*).

2.4.2 Formato de mensaje JSON

En los Servicios Web basados en REST, el formato de los mensajes a intercambiar suele ser XML (tal y como se ha comentado anteriormente) o en formato JSON. JSON, acrónimo de *JavaScript Object Notation*, es un formato ligero para el intercambio de datos. JSON es un subconjunto de la notación literal de objetos de JavaScript que no requiere el uso de XML. Una de las ventajas de JSON sobre XML como formato de intercambio de datos en este contexto es que es mucho más sencillo escribir un analizador sintáctico (parser) de JSON. Además, los mensajes codificados con JSON suelen ser más pequeños (en número de bytes) que su equivalente en XML. Por otro lado, si bien es frecuente ver JSON posicionado contra XML, también es frecuente el uso de JSON y XML en la misma aplicación. Las Figuras 2.6 y 2.7 muestran un ejemplo de mensaje en formato JSON y XML respectivamente.

```
{"menu": {
  "id": "file",
  "value": "File",
  "popup": {
    "menuitem": [
      {"value": "New", "onclick": "CreateNewDoc()"},
      {"value": "Open", "onclick": "OpenDoc()"},
      {"value": "Close", "onclick": "CloseDoc()"}
    ]
  }
}}
```

Figura 2.6.: Ejemplo de mensaje en formato JSON

```
<menu id="file" value="File">
  <popup>
    <menuitem value="New" onclick="CreateNewDoc()" />
    <menuitem value="Open" onclick="OpenDoc()" />
    <menuitem value="Close" onclick="CloseDoc()" />
  </popup>
</menu>
```

Figura 2.7.: Ejemplo de mensaje en formato XML

2.5 RESTful vs. SOAP

El principal beneficio de SOAP recae en que las interfaces de los servicios están bien definidas (a través de WSDL), lo que permite poder ser testado y depurado antes de poner en marcha la aplicación. En cambio, las ventajas de la aproximación basada en REST recaen en la potencial escalabilidad de este tipo de sistemas, así como el acceso con escaso consumo de recursos a sus operaciones debido al limitado número de operaciones y el esquema de direccionamiento unificado.

En la Tabla 2.1 se comparan algunas características de SOAP y RESTful. Como se ha comentado, REST utiliza el protocolo de comunicación HTTP y, normalmente, el formato XML o JSON para el intercambio de datos. Cada URL representa un objeto sobre el que se pueden utilizar los métodos POST, GET, PUT y DELETE. Sin embargo en SOAP, toda la infraestructura está basada en XML, cada objeto puede tener métodos definidos por el programador con los parámetros que sean necesarios. Por lo tanto, podemos concluir que REST es más ligero, con poca configuración, se lee fácilmente (URL) y no hace falta nada especial para implementarlo. Por otro lado, como ya hemos comentado, SOAP tiene un tipado más fuerte lo que le proporciona una cierta capacidad para verificar la corrección de las operaciones.

	REST	*SOAP*
Características	*Las operaciones se definen en los mensajes.* *Una dirección única para cada instancia del proceso.* *Cada objeto soporta las operaciones estándares definidas.* *Componentes débilmente acoplados.*	*Las operaciones son definidas como puertos WSDL.* *Dirección única para todas las operaciones.* *Múltiple instancias del proceso comparten la misma operación.* *Componentes fuertemente acoplados*
Ventajas declaradas	*Bajo consumo de recursos.* *Las instancias del proceso son creadas explícitamente.* *El cliente no necesita información de enrutamiento a partir de la URI inicial.* *Los clientes pueden tener una interfaz "listener" (escuchadora) genérica para las notificaciones.* *Generalmente fácil de construir y adoptar.*	*Fácil (generalmente) de utilizar.* *La depuración es posible.* *Las operaciones complejas pueden ser escondidas detrás de una fachada.* *Envolver APIs existentes es sencillo* *Incrementa la privacidad.* *Herramientas de desarrollo.*
Posibles desventajas	*Gran número de objetos.* *Manejar el espacio de nombres (URIs) puede ser engorroso.* *La descripción sintáctica/semántica muy informal (orientada al usuario).* *Pocas herramientas de desarrollo.*	*Los clientes necesitan saber las operaciones y su semántica antes del uso.* *Los clientes necesitan puertos dedicados para diferentes tipos de notificaciones.* *Las instancias del proceso son creadas implícitamente.*

Tabla 2.1.: REST vs. SOAP

La Tabla 2.2 muestra algunas diferencias entre la tecnología REST y SOAP. A modo de conclusión, podemos decir que, junto a otras ventajas, la sencillez a la hora de implementar Servicios Web con REST hace que exista una gran aceptación y apoyo a esta tecnología. Así pues, hoy en día, a pesar de que ha habido muchas empresas que han apostado por SOAP, el estilo REST se abre camino en el mundo empresarial a gran velocidad. Parece que REST podría tener una mayor aceptación en el futuro, no obstante, ello dependerá de cómo evolucionen los frameworks que permiten desarrollar esta tecnología. Por otro lado, actualmente se han presentado propuestas donde se enlazan los principios de REST con la nueva versión de SOAP. En lo referente al presente proyecto y a título personal, REST es una forma de desarrollo más ágil.

	REST	*SOAP*
TECNOLOGIA	• *Pocas operaciones con muchos recursos* • *Se centra en la escalabilidad y rendimiento a gran escala para sistemas distribuidos hipermedia.*	• *Muchas operaciones con pocos recursos* • *Se centra en el diseño de aplicaciones distribuidas.*
PROTOCOLO	• *HTTP GET, HTTP POST, HTTP PUT, HTTP DEL* • *XML auto descriptivo* • *Sincrono*	• *SMTP, HTTP POST, MQ* • *Tipado fuerte, XMl Schema* • *Sincrono y Asincrono*
SEGURIDAD	• *HTTPS* • *Comunicacion punto a punto y segura.*	• *WS SECURITY* • *Comunicacion origen a destino segura.*

Tabla 2.2.: REST vs. SOAP

CAPÍTULO III

Humanoid Robot Modeling Environment (HuRoME)

En el presente capítulo presentamos una breve revisión de las características y funcionalidades de la herramienta HuRoME [2-3].

3.1 Descripción general de la herramienta

El entorno de modelado HuRoME (*Humanoid Robot Modeling Environment*) integra un conjunto de herramientas diseñadas para facilitar el modelado de coreografías y la modernización del software existente para el robot humanoide Robonova [4]. HuRoME permite a los numerosos usuarios de Robonova: (1) modelar gráficamente y validar formalmente las secuencias de movimientos del robot (coreografías), (2) generar automáticamente la implementación asociada a cada coreografía en el lenguaje RoboBASIC, y (3) modernizar y reutilizar el software ya existente, permitiendo la obtención de los modelos equivalentes a cualquier programa RoboBASIC existente.

El entorno HuRoME ofrece una aproximación al desarrollo de software para robótica utilizando un enfoque de Desarrollo de Software Dirigido por Modelos. HuRoME no sólo facilita la generación automática de código a partir de modelos, sino también al proceso inverso, permitiendo la transformación de programas ya existentes (legacy code) en modelos, facilitando así su tratamiento (depuración, extensión, reutilización, análisis, simulación, etc.) con un nivel de abstracción mayor que el que proporciona el código fuente. En el siguiente apartado describimos brevemente los fundamentos del *Desarrollo de Software Dirigido por Modelos*.

3.2 Breve revisión del Desarrollo de Software Dirigido por Modelos

El *Desarrollo de Software Dirigido por Modelos* (DSDM) [1] comprende un conjunto de técnicas y herramientas que permiten modelar formalmente los sistemas que se quieren desarrollar para, posteriormente, aplicando una serie de transformaciones automáticas, obtener el código final de las aplicaciones. Así, el DSDM gira entorno a la definición y el uso sistemático de modelos y de transformaciones de modelos a lo largo de todo el ciclo de vida del desarrollo de software.

La noción de modelo es muy antigua y puede definirse como "una abstracción o simplificación de la realidad con un cierto propósito" [21]. Más concretamente, de acuerdo a la definición proporcionada por Bézivin y Gerbé en [22], "un modelo es una simplificación de un sistema construido con un objetivo definido. El modelo ha de ser capaz de responder a las preguntas que se le formulen como si se tratara del sistema real. Las respuestas que proporciona el modelo deberán ser exactamente las mismas que si respondiera el sistema, con la condición de que las preguntas se formulen en los mismos términos en los que se definió el modelo". Así, se puede decir que un modelo es una simplificación de un sistema que proporciona información sobre el mismo en el contexto de unos ciertos objetivos.

Un sistema puede estar representado por uno o más modelos, cada uno de ellos centrado en representar un determinado aspecto de interés con un cierto grado de detalle (nivel de abstracción). Así, es posible crear modelos de análisis, de diseño, e incluso modelos de implementación, muy próximos a la plataforma de desarrollo que se empleará para codificar el sistema final. Los modelos de más alto nivel pueden evolucionar a otros de más bajo nivel mediante la aplicación de transformaciones automáticas de modelos, hasta obtenerse un modelo lo suficientemente detallado como para poder generar, a partir de él, el código final del sistema. Así, el uso de modelos y de transformaciones modelo-a-modelo y modelo-a-código permiten automatizar, en gran medida, el proceso de desarrollo de software.

Para definir cualquier modelo es necesario contar con un lenguaje de modelado. Los meta-modelos definen formalmente la sintaxis abstracta de los lenguajes de modelado [23], recogiendo sus conceptos (palabras del lenguaje), así como las reglas que indican cómo se pueden combinar dichos conceptos para formar modelos válidos. De este modo, como se indica en [22], "un modelo solamente será válido si es conforme a su meta-modelo". Por otra parte, la representación (ya sea gráfica o textual) asociada a cada uno de los elementos del meta-modelo (conceptos y relaciones) constituye la sintaxis concreta del lenguaje.

Por último, la semántica asociada a los elementos de cada meta-modelo, esto es, su significado o interpretación, tiene impacto fundamentalmente en las transformaciones de modelos, en las que se define cómo transformar (interpretar) cada concepto del modelo de partida, en términos otro lenguaje (ya sea de modelado o de programación).

Actualmente, el DSDM es uno de los paradigmas de desarrollo de software más en boga en el ámbito de la Ingeniería del Software. Los fundamentos sobre los que asienta el DSDM fueron establecidos hace ya un par de décadas. Sin embargo, el auge de este enfoque sólo ha sido posible en los últimos años gracias, por una parte, a la encomiable labor de estandarización que, en el ámbito del DSDM, ha llevado a cabo la OMG (*Object Management Group*) con su iniciativa MDA (*Model-Driven Architectura*) [24], y por otra, a la aparición en el mercado de las primeras herramientas que dan soporte a este nuevo enfoque, permitiendo explotar todo su potencial.

Entre las herramientas que actualmente dan soporte al DSDM, cabe mencionar las siguientes: DSL Tools (de Microsoft) [25], Meta-Edit+ (de la empresa *Meta-Case*) [26] y Eclipse. En los últimos años, Eclipse se ha convertido en el estándar de facto para la comunidad de DSDM, ya que implementa las principales tecnologías estandarizadas por el OMG para dar soporte a este enfoque.

3.3 Descripción de las funcionalidades de HuRoME

A continuación se describen las tres herramientas que conforman HuRoME (ver Figura 3.1), todas ellas desarrolladas utilizando las facilidades para DSDM que ofrece la plataforma Eclipse. En primer lugar, la Sección 2.1 describe la herramienta gráfica de modelado y validación de coreografías, implementada con GMF (*Graphical Modeling Framework*) [27]. A continuación, la Sección 2.2, describe la transformación modelo-a-texto (M2T), implementada con JET (*Java Emitter Templates*) [28], para generar código RoboBASIC y/o RoboScript [29] a partir de los modelos anteriores. Por último, la Sección 2.3, describe la transformación texto-a-modelo (T2M), implementada con Xtext [30] y ATL (*ATLAS Transformación Lenguaje*) [31], para obtener modelos a partir de código RoboScript.

3.3.1 Editor gráfico de modelos

HuRoME ofrece una herramienta gráfica de modelado que permite diseñar, en un entorno amigable e intuitivo, coreografías de movimientos para robots Robonova. Así, cualquier usuario puede especificar, de manera totalmente 'visual', las coreografías del robot sin necesidad de conocer su lenguaje de programación.

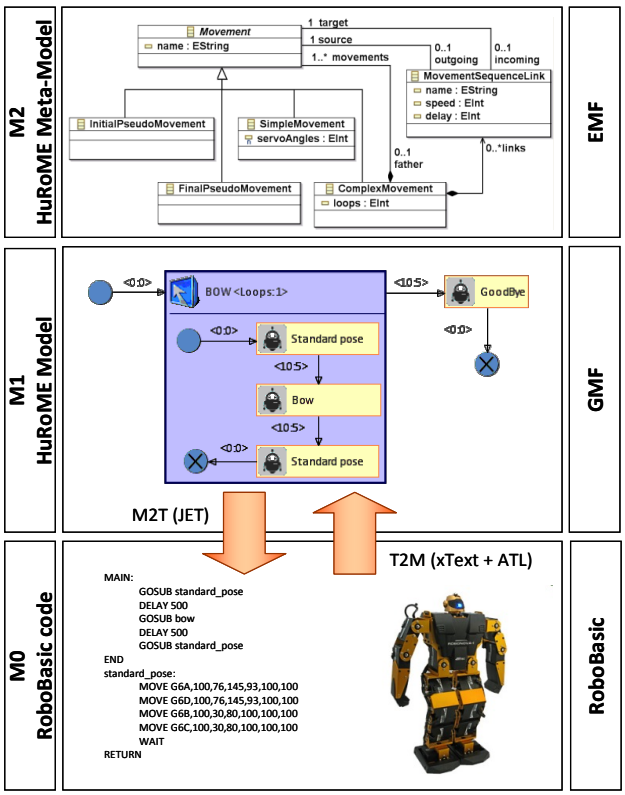

Figura 3.1.: Herramientas integradas en HuRoME (fuente [3])

El meta-modelo sobre el que se ha construido esta herramienta, incluye los conceptos necesarios para modelar las coreografías de movimientos de los robots Roblonaba, en particular: (1) *SimpleMovement*, modela los cambios de postura logrados mediante el accionamiento de uno o más de sus actuadores mecánicos; (2) *ComplexMovement*, modela la composición jerárquica de movimientos; (3) *PseudoMovement*, modela los puntos de control que establecen el inicio y el final de cada secuencia; y (4) *MovementSequenceLink*, enlaza parejas de movimientos indicando el orden en que éstos se ejecutan. Entre los atributos asociados a cada uno de estos conceptos, encontramos los valores angulares de las articulaciones en cada SimpleMovement, el número de veces que se repite cada *ComplexMovement*, o el retardo y la velocidad asociada a los *MovementSequenceLink*.

La herramienta de modelado proporciona una sintaxis concreta (en este caso gráfica), para cada uno de los conceptos del meta-modelo (sintaxis abstracta). En la Figura 3.2 se muestra el editor gráfico de modelos desarrollado como parte de HuRoME. Esta editor consta de tres partes: (1) una paleta de herramientas (panel derecho) desde la que el usuario puede seleccionar los conceptos que desea incorporar a sus modelos, (2) un área de trabajo (panel central) donde podrá modelar las coreografías de movimientos del robot, y (3) una vista de propiedades (panel inferior) para añadir y completar la información asociada a los distintos elementos del modelo (por ejemplo, el valor angular de cada uno de los servomotores del robot en cada *SimpleMovement*).

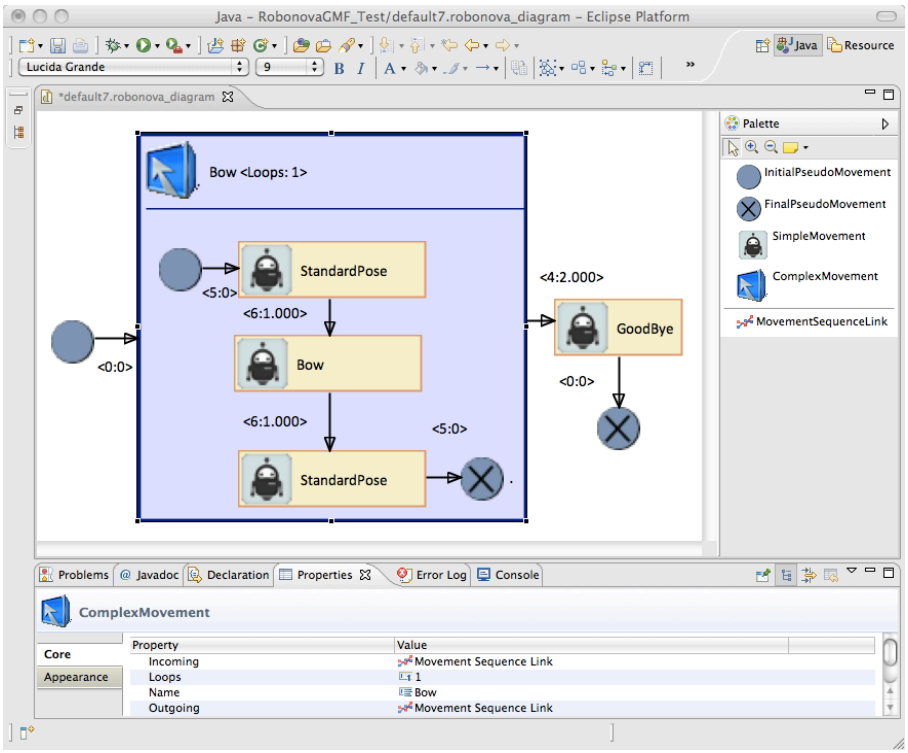

Figura 3.2.: Apariencia del editor gráfico de modelos de HuRoME

La herramienta de modelado, implementada como parte de HuRoME, incorpora facilidades de validación, que permiten comprobar formalmente la corrección sintáctica de los modelos, comprobando si éstos son conformes al meta- modelo y a una serie de restricciones adicionales, escritas en lenguaje OCL y añadidas al editor. Entre las restricciones incluidas en este caso, cabe destacar las siguientes: (1) no se permiten las conexiones reflexivas (conectar un

movimiento consigo mismo); (2) sólo se pueden enlazar movimientos del mismo nivel de composición; (3) el elemento inicial de una secuencia tiene sólo un link de salida y el final sólo un link de entrada; (4) todos los movimientos, simples o compuestos, tienen un único link de entrada y otro de salida, salvo la excepción descrita en 3.

Cuando se detecta una violación de estas reglas o de las recogidas explícitamente en el meta-modelo, los elementos afectados son marcados con un círculo rojo y una cruz (ver Figura 3.2). Sólo cuando los modelos están totalmente libres de errores de validación, puede aplicárseles la transformación que se describe a continuación para generar el código correspondiente para el robot.

3.3.2 Generación de código a partir de modelos

El objetivo de esta herramienta es convertir, mediante una transformación automática modelo- a-texto (M2T), los modelos diseñados y validados con el editor gráfico descrito en la Sección 2.1, en código RoboBASIC que pueda ser ejecutado en un robot Robonova. La Tabla 1 refleja las estructuras del lenguaje RoboBASIC a las que se traducen los distintos conceptos definidos en el meta-modelo descrito en la sección anterior.

El procedimiento de transformación lo dirige el motor de ejecución de JET. Éste recorre el modelo especificando qué plantilla de código se debe invocar para cada tipo de elemento encontrado.

Así pues, según se indica en la Tabla 1, las instancias de la clase *ComplexMovement* son traducidas en bucles que se repiten el número de veces indicado por el atributo *loops*. El código que se incluirá en cada bucle vendrá determinado por el resultado de procesar los elementos contenidos en cada *ComplexMovement*. Por otro lado, los elementos SimpleMovement se transforman en una o más sentencias de movimiento simultáneo de los servomotores, delimitadas con la clave WAIT. De este modo, el robot transiciona a una pose determinada por el valor de la propiedad *servoAngles*. Por último, las instancias de la clase *MovementSequenceLink* se traducen a sentencias que controlan la velocidad y el retardo de los movimientos, de acuerdo a los valores establecidos en las propiedades *speed* y *delay*.

Conceptos del dominio	Código RoboScript
ComplexMovement	*Código elementos contenidos repetido loops veces*
SimpleMovement	PTP [ALLON | ALLOFF] MOVE24 *servoAngles*
MovementSequenceLink	SPEED *speed* DELAY *delay*

Tabla 1.: Correspondencia entre los conceptos del meta-modelo y las estructuras de código RoboScript (fuente [3])

3.3.3 Generación de modelos a partir de código

Con el objetivo de soportar el proceso inverso al descrito en el apartado anterior, la versión actual de HuRoME incorpora una transformación texto-a-modelo (T2M) que permite obtener modelos de coreografías a partir de código RoboBASIC v1.0 ya existente (bien generado automáticamente por la herramienta anterior o codificado manualmente por un desarrollador). En

una primera fase, partiendo del código RoboBASIC v1.0, se realiza una transformación T2M para obtener un modelo conforme al meta-modelo de Xtext. Asociado a esta fase se ha desarrollado un editor textual para el lenguaje RoboBASIC, que permite el formateo léxico y la validación sintáctica del código (ver Figura 3.3). En una segunda fase, se ha implementado en ATL una transformación M2M con el fin de obtener modelos conformes al meta- modelo de HuRoME a partir de los modelos Xtext obtenidos en la etapa anterior.

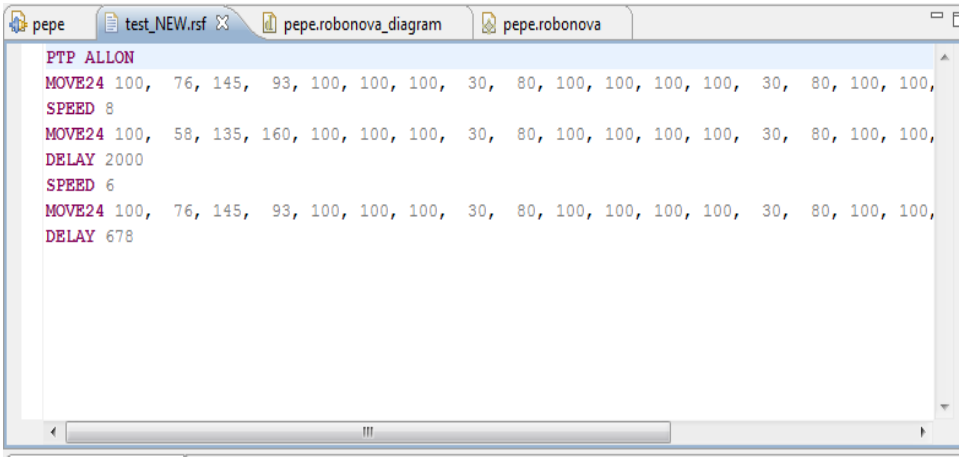

Figura 3.3.: Editor textual XTEXT

3.4 Ejemplo I: obteniendo código desde un modelo

En este apartado ilustramos el uso de la herramienta HuRoME para obtener código RoboBASIC y RoboScript a partir de un modelo de ejemplo de secuencia de movimientos del robot.

3.4.1 Diseño de un modelo

A modo de ejemplo, se propone realizar el modelo correspondiente a una coreografía sencilla. En la secuencia que planteamos, el robot parte de una posición estándar, se encuentra alzado sobre sus piernas y con los brazos relajados, tras ello, se inclinará a modo de reverencia y volverá a su posición, por último, termina levantando un brazo como despedida. Para ello, creamos un nuevo Proyecto Java en Eclipse (File→New→Java Project) y tras ello, un nuevo modelo HuRoME (New→Other→Robonova Diagram). Tras esta operación se han creado dos ficheros, uno con la extensión *.robonova (continente de la información del modelo) y otro con *.robonova_diagram (continente de la información gráfica del diagrama). La Figura 3.2 muestra el ejemplo realizado utilizando el editor gráfico de modelos HuRoME.

3.4.2 Validación del modelo

Validamos el modelo representado en la Figura 3.2 seleccionando la opción *Validate* en el menú principal. Obtendremos un diálogo que nos indica la correcta validación del modelo, en caso de erros obtendremos la información detallada en la vista *Problems*.

3.4.3 Generación de código

Para lanzar la generación automática de código RoboBASIC o RoboScript desde el modelo creado y validado, se han seguido los siguientes pasos:

1. Copiamos el modelo (archivo *.robonova) y lo movemos a la carpeta llamada model localizada en el Proyecto JET (este Proyecto forma parte de la herramienta HuRoME y contiene la implementación de la transformación M2T).

2. Abrimos la ventana Run Configurations (seleccionando Run→Run Configurations). Véase la Figura 3.4. En esta ventana se crea una nueva configuración haciendo doble clic en JET Transformation de la lista (sólo en el caso de que no exista ya una instancia creada) y escribimos la ruta al modelo de entrada de la transformación. Finalmente pulsamos Run.

3. Aparecerá dos ficheros de salida, uno con código roboBASIC (*.bas) y otro con RoboScript (*.rsf), en la raíz del Proyecto (Figura 3.5).

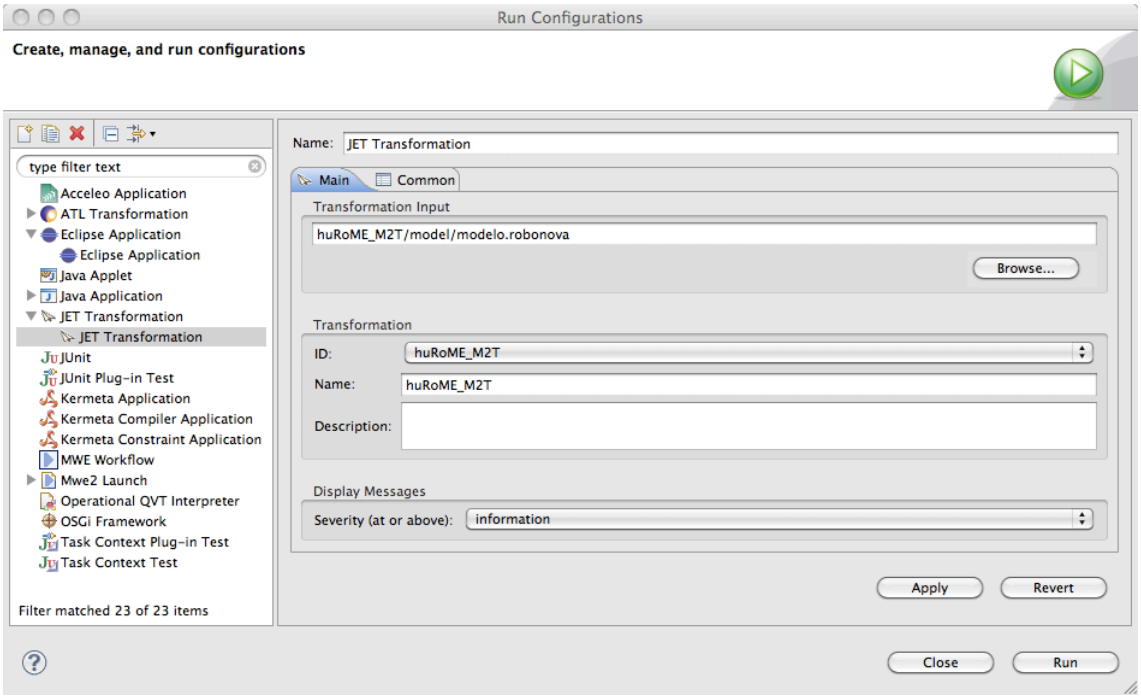

Figura 3.4.: Ventana Run Configurations

3.5 Ejemplo II: obteniendo el modelo desde código RoboScript

En este apartado partimos del código RoboScript generado anteriormente para ilustrar el procedimiento para obtener el correspondiente modelo gráfico, que deberá ser equivalente al de la Figura 3.2.

3.5.1 Serialización del código RoboScript

En primer lugar, tras abrir el código con el editor textual Xtext de HuRoME, ejecutamos el proceso de serialización para obtener la representación del código en formato XMI (extensión *.xmi). En la figura 3.7 puede verse el modelo obtenido desde código RoboScript.

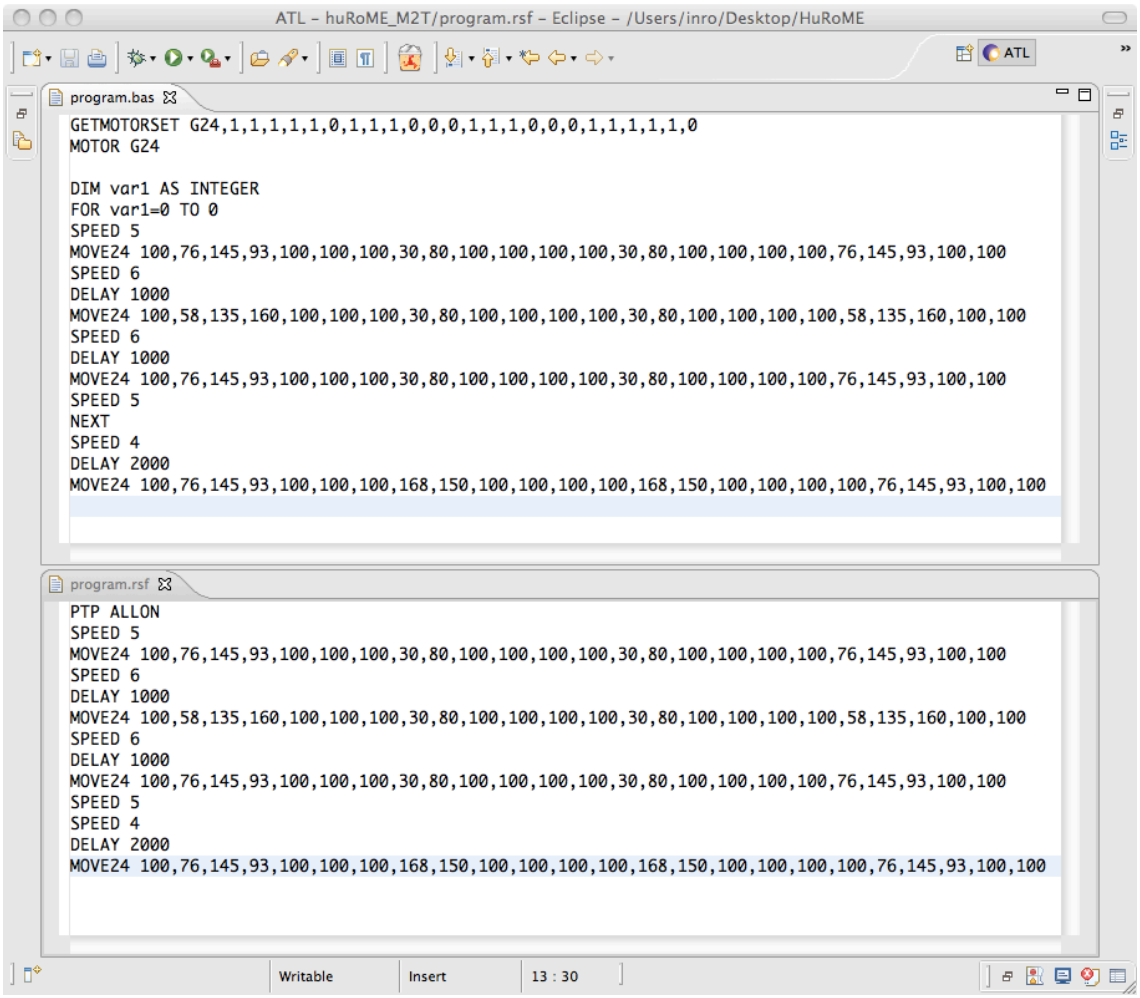

Figura 3.5.: Código generado. La parte superior corresponde a RoboBASIC, la parte inferior a RoboScript

3.5.2 Ejecución de la transformación

Para la obtención del modelo a partir de la representación XMI de código, se siguen los siguientes pasos:

1. Abrimos la ventana Run Configurations (seleccionando Run→Run Configurations).En esta ventana se crea una nueva configuración haciendo doble clic en ATL Transformation de la lista (sólo en el caso de que no exista ya una instancia creada) y revisamos o escribimos la ruta a los meta-modelos y modelos de entrada y salida. Finalmente pulsamos Run.

2. Se creará un nuevo modelo con extensión *.robonova que podrá abrirse con el Tree Editor.

3. Para conseguir una representación gráfica del modelo y poder abrirlo utilizando el editor GMF, debemos copiar el fichero *.robonova en el Proyecto Java que fue creado para albergar el plug-in del editor gráfico. Tras ello, seleccionar el fichero y desplegar su menú contextual haciendo clic sobre la opción “Inicialize robonova_diagram diagram file”, ésta generará forma automática el fichero *.robonova_diagram. En la Figura 3.8 puede verse el modelo del ejemplo en sus distintas representaciones.

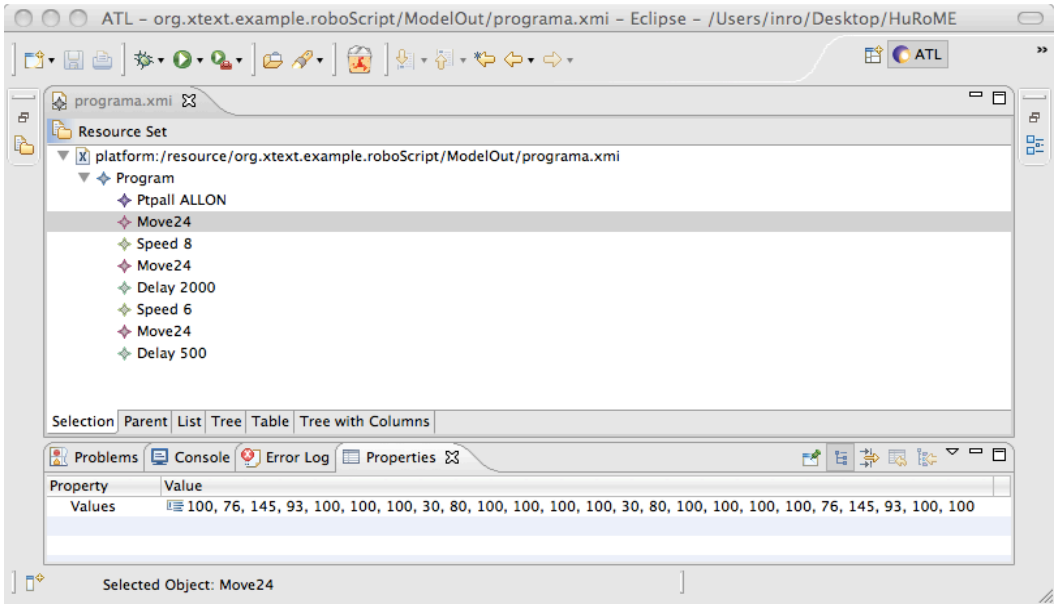

Figura 3.7.: Modelo tras el proceso de serialización

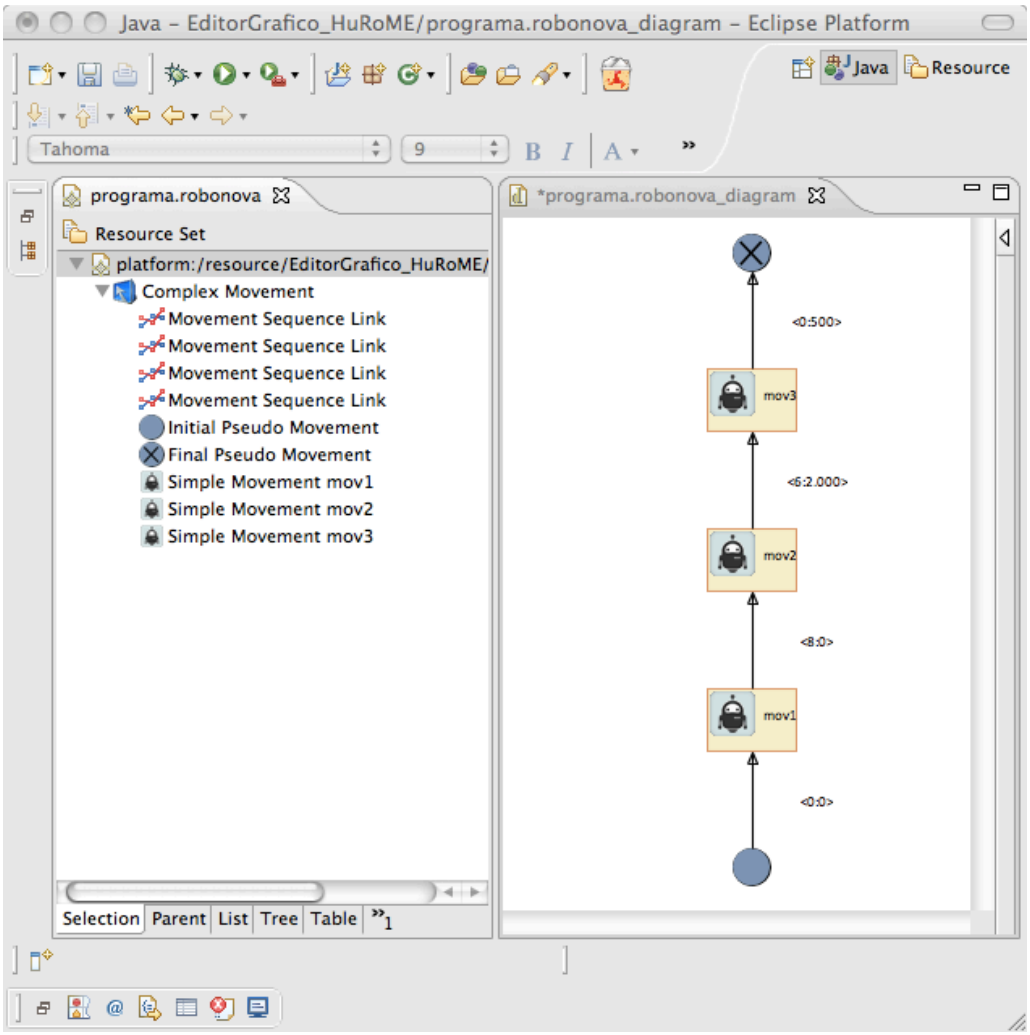

Figura 3.8.: Modelo final tras la transformación ATL

CAPÍTULO IV

Distribución del entorno HuRoME a través de Servicios Web

En este capítulo utilizamos la herramienta HuRoME para ilustrar cómo podrían utilizarse Servicios Web para distribuir entornos de Desarrollo de Software Dirigido por Modelos (DSDM). Esta tecnología permitiría desplegar transformaciones de modelos en servidores remotos mejorando, entre otros aspectos, la flexibilidad, extensibilidad y mantenibilidad del entorno de modelado.

4.1 Consideraciones iniciales

En el capítulo anterior presentamos una revisión de la herramienta de modelado HuRoME. Así pues, dado que HuRoME es una herramienta sencilla que aglutina características comunes y extendidas en otros entornos de Desarrollo de Software Dirigido por Modelos, consideramos que esta herramienta representa un ejemplo adecuado para ilustrar el desarrollo y aplicación de Servicios Web en este tipo de entornos de modelado.

Se pretende realizar un Servicio Web para ejecutar las transformaciones de modelos de la herramienta HuRoME. Para ello se implementarán dos versiones del mismo servicio, que permitirá comparar dos de las tecnologías más en voga en el desarrollo de Servicios Web: SOAP y REST. Cabe señalar que, a pesar de utilizar SOAP y REST para implementar el mismo Servicio Web, la diferencia entre ambos casos residirá únicamente en los detalles de la implementación, pero no en la función o la arquitectura de la aplicación. Así pues, se intentará reutilizar la misma definición de mensajes e infraestructura de clases.

Una vez finalizado el desarrollo del Servicio Web con SOAP y REST, se realizaran algunas pruebas que permitirán exponer las particularidades de ambas tecnologías. Se comparará la diferencia entre los formatos de mensaje XML y JSON y entre serializar los modelos utilizando XMI o el formato binario de EMF. Los resultados serán presentados en el siguiente capítulo.

4.2 El entorno de desarrollo

A continuación se introducen los elementos principales empleados para el desarrollo de los Servicios Web.

1. **Eclipse**

   Es el IDE donde se crearan los Servicios Web. El presente proyecto utiliza Eclipse IDE for Java EE Developers [32] versión Indigo. Se ha elegido esta distribución de Eclipse porque permite crear Dymanic Web Project, que son el tipo de proyectos donde se pueden crear aplicaciones web. Java EE (Enterprise Edition), es una plataforma de programación para desarrollar y ejecutar software de aplicaciones en el lenguaje de programación Java. Java EE tiene varias especificaciones de API, tales como JDBC, RMI, e-mail, JMS, Servicios Web, XML, etc y define cómo coordinarlos. Java EE también configura algunas especificaciones únicas para Java EE para componentes. Estas incluyen Enterprise JavaBeans, servlets, portlets (siguiendo la especificación de Portlets Java), Java Server Pages y varias tecnologías de servicios web. Para que Eclipse funcione en el ordenador deberá tener instalado previamente la máquina virtual de java (JVM), que es la encargada de traducir el código Java a un código intermedio y portátil denominado bytecode, para su correspondiente paso a código máquina.

2. **Framework CXF**

   CXF es un framework completo, de código abierto, para Servicios Web. JAX-WS (Java API for XML Web Services) para el desarrollo de Servicios Web tipo SOAP y JAX-RS (Java API for RESTful Web Services) para el desarrollo de Servicios Web tipo RESTful. En los próximos pasos, veremos como a través del asistente podemos seleccionar cxf para utilizar en el proyecto y como es añadido en el tipo de creación del Servicio Web a bajo nivel (sin asistente).

   Anotaciones. Es una forma de añadir metadatos al código fuente Java que están disponibles para la aplicación en tiempo de ejecución. CXF permite crear anotaciones en las clases Java que deseamos publicar en el Servicio Web.

   Instalación. Debemos descargar CXF en el ordenador [33], una vez realizamos esto se descomprime el archivo. Nos dirigimos a Window->Preferences, donde seleccionamos Web Services->CXF 2.x preferences y presionamos sobre add. Buscamos, a través del árbol de directorios, la ubicación de CXF y presionamos en finish. Lo activamos y exportamos la librería para poder ser utilizada.

3. **Servidor de aplicaciones Tomcat**

   Tomcat es un servidor Web con soporte de servlets y JSP (Java Server Pages). Como está escrito en Java, funciona en cualquier sistema operativo que disponga de la maquina virtual de Java.

   Instalación. Se descarga Apache Tomcat [34] en el ordenador, a continuación se descomprime el archivo. Se crea un nuevo proyecto Eclipse, File->New->Dinamic Web

Project. En la viñeta New Runtime seleccionamos Apache Tomcat v7.0, seleccionamos next y agregamos el directorio donde se encuentra Apache Tomcat.

4.3 Creación de un Servicio Web SOAP

En esta sección abordaremos la creación de un Servicio Web con SOAP. Se describirán los pasos seguidos y la implementación realizada.

4.3.1 Uso del asistente para la creación de un Servicio Web

Dado que la forma más habitual para crear un Servicio Web es utilizar el asistente que proporciona Eclipse, a continuación se describe de forma básica. No obstante, comentar que en este proyecto no se ha hecho uso de éste, sino que se ha optado por implementar el código del servicio para tener un mayor control sobre el mismo.

1. Se accede mediante el botón derecho a Web Services->Create Web Service, mostrándose la ventana representada en la Figura 4.1.
2. Se selecciona el tipo de desarrollo que se pretende seguir. El tipo “bottom up” indica que el código del servicio será generado automáticamente. Para ello, el usuario debe proporcionar la clase Java que implementa la funcionalidad del servicio con las correspondientes anotaciones definidas por CXF. Por otro lado, si se selecciona el tipo “top down” el Servicio Web se generaría automáticamente a partir de la definición de interfaz WSDL creada por el usuario.
3. También deberá especificarse el servidor sobre el se ejecuta la aplicación. En nuestro caso, se selecciona Apache CXF 2.X y Tomcat v7.0 Server. Véase Figura 4.2.
4. Por último, el asistente nos permite diferentes opciones, como generar el servicio, desplegarlo y ejecutarlo. Por ejemplo, la Figura 4.3 muestra la instalación del servicio.

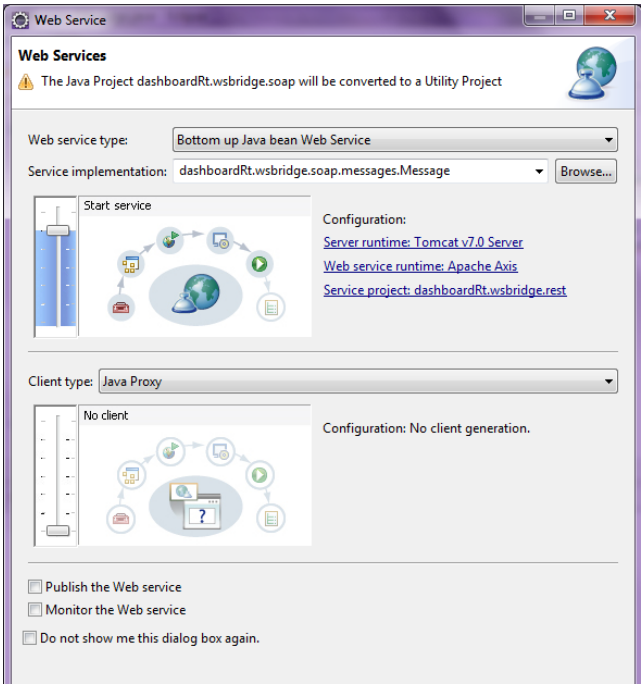

Figura 4.1.: Creación servicio web.

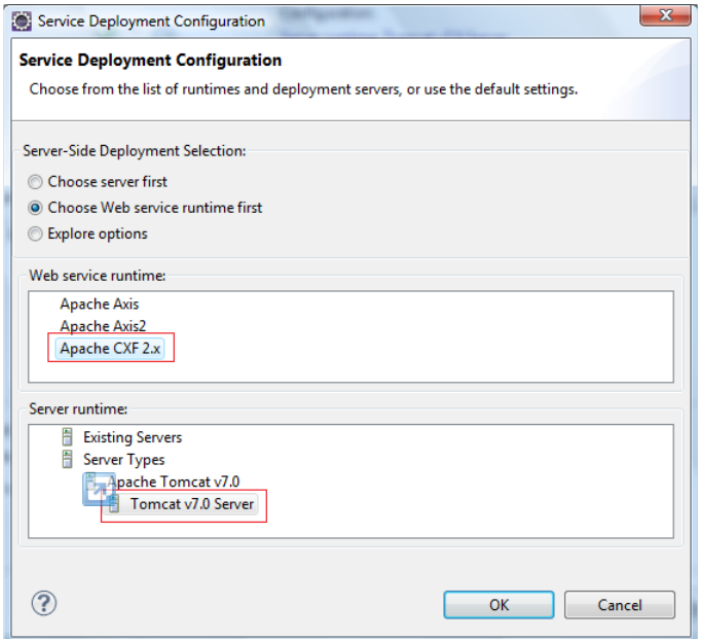

Figura 4.2.: Configuración del servicio web.

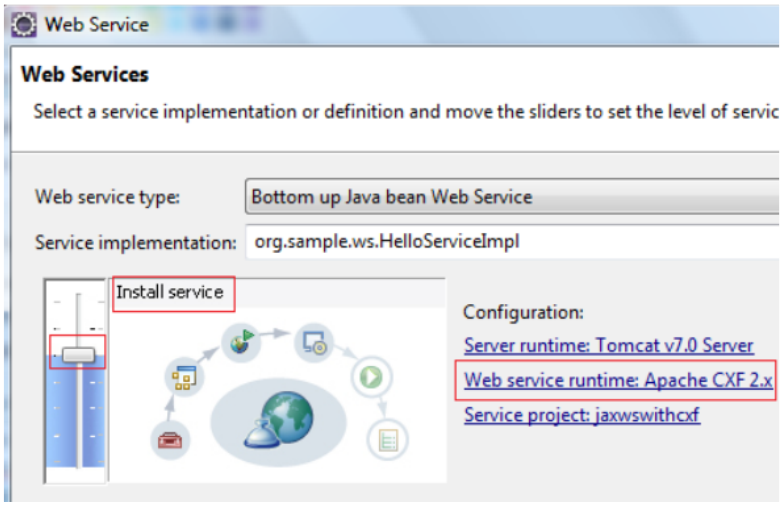

Figura 4.3.: Instalación del servicio web

4.3.2 Detalles del desarrollo realizado

El diagrama UML de clases diseñado para la implementación del Servicio Web SOAP para el entorno HuRoME queda representado en la Figura 4.4. Comentar que este diagrama de clases representa una infraestructura cuyos elementos son generales e independientes del entorno HuRoME. Por ello, se mantendrá esta organización de clases realizadas para la implementación del Servicio Web con REST. A continuación veremos los detalles de cada clase.

La clase ***Message***, como su nombre indica es el mensaje que se intercambia entre el cliente y el servidor. Éste contendrá la información necesaria para que el Servicio Web ejecute las operaciones pertienentes. Además, el servicio enviará el resultado al cliente instanciando también esta clase. Dado que esta clase es el elemento primordial para la comunicación SOAP, debemos poder serializar cualquier objeto Message a código XML, para poder enviarlo y una vez

recibido, deserializar dicho XML para obtener una instancia Message. Así pues, es necesario indicar la correspondencia entre los distintos atributos de la clase Message y el formato XML. Para dicha tarea, se utiliza JAXB (Java Architecture for XML Binding) [20] que es una API que permite a los desarrolladores Java asignar clases de Java a representaciones XML. JAXB proporciona dos características principales: la capacidad de serializar las referencias de objetos Java a XML y la inversa, es decir, deserializar XML en objetos Java. En otras palabras, JAXB permite almacenar y recuperar datos en memoria en cualquier formato XML. Para ello utiliza anotaciones que indican cómo será tratado el código que va a continuación de ellas, es decir, como irá representado en el XML.

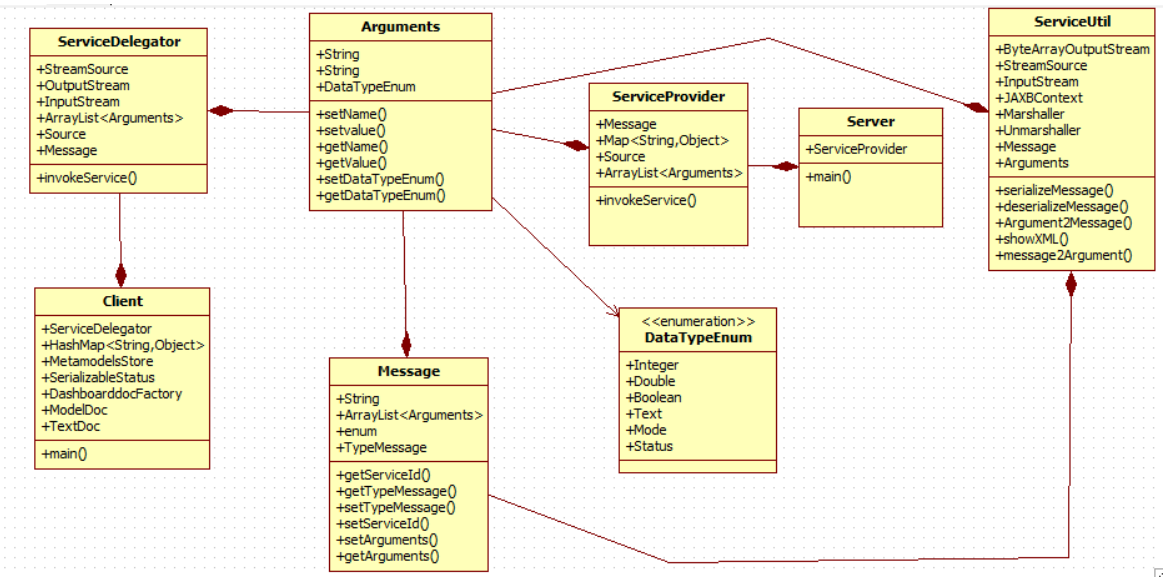

Figura 4.4.: Diagrama de clases del Servicio Web SOAP implementado.

Las anotaciones principales de JAXB son las siguientes:

- **@XmlRootElement**, indica el elemento raíz del XML, cuyo nombre queda definido mediante el atributo “name”. Este elemento se define a nivel de clase.

```
@XmlRootElement(name = "message")
public class Message {

    private String serviceId;
    private static ArrayList<Arguments> argu = new ArrayList<Arguments>();
    public enum typeMessage{Request, response}
    public typeMessage type;

    public String getServiceId() {
        return serviceId;
    }

    public typeMessage getTypeMessage(){

        return type;
    }
```

Figura 4.5.: Anotación @XmlRootElement en la implementación de la clase Message

- **@XmlElement**, indica que el método en el que está ubicada será un elemento dentro del XML, en la jerarquía marcada por XmlRootElement, y llevará el nombre definido por "name". Este elemento está asociado a un atributo de la clase, para establecer este vínculo, puede colocarse la anotación en el método modificador de dicho atributo. Comentar que JAXB requiere que todos los atributos que vayan aparecer como campos en el XML tengan métodos get/set para acceder al valor. Además, la signatura de dichos métodos debe estar compuesta por el nombre del atributo en cuestión.

```
@XmlElement(name = "type")
public void setTypeMessage(typeMessage type){

    this.type=type;
}
/**
 *
 * @param serviceId, establece el valor de la etiqueta en el XML generado (idString)
 */
@XmlElement(name = "idString")
public void setServiceId(String serviceId) {
    this.serviceId = serviceId;
}
```

Figura 4.6.: Anotación @XmlElement en la implementación de la clase Message

- **@XmlElementWrapper**, permite traducir vectores o listas como conjuntos de datos en XML. Por ejemplo, en la Figura 4.7 puede verse cómo los argumentos serán dispuestos como hijos del elemento "data" en el XML.

```
@XmlElementWrapper(name="data")
@XmlElement(name = "parameter")
public void setArguments(ArrayList<Arguments> argu){

    this.argu = argu;
}

public ArrayList<Arguments> getArguments() {
    return argu;
}
```

Figura 4.7.: Anotación @XmlElementWrapper en la implementación de la clase Message

La clase Message contiene el identificador del servicio, que permite distinguir un servicio de otro y el tipo de mensage, que puede ser una petición o una respuesta. En el mensaje se añaden un conjunto de argumentos, que son los datos de entrada/salida que se intercambian con el Servicio Web. Cada argumento, definido por la clase ***Argument***, debe tener un nombre, un valor y el tipo de argumento. Los tipos de los argumentos están especificados por el enumerador ***DataTypeEnum***, pueden ser enteros, booleanos, números reales, modelos (se refiere a modelos EMF), textos (para indicar documentos de texto, por ejemplo, generados tras una transformación modelo-a-texto) o status (para indicar errores ocurridos durante la ejecución del servicio).

La clase ***ServiceDelegator***, es llamada internamente por la clase Client, se trata de un elemento que actúa como enlace local del servicio que se quiere ejecutar. Dado que en el presente proyecto la implementación del cliente y el servidor no se ha realizado por medio de asistente, esta clase contiene la implementación de la conexión SOAP con el servidor programada "a más bajo nivel", un enfoque que ofrece mayor flexibilidad a la hora de manejar la información enviada y recibida. Tal y cómo se observa en la Figura 4.8, ServiceDelegator indica el servicio que

queremos ejecutar y el puerto del Servicio Web donde el cliente ha de conectarse para poder realizar peticiones. Esto es, SERVICE_NAME y PORT_NAME respectivamente, más adelante se verá como se especifica este puerto SOAP en el servidor.

```
public class ServiceDelegator {

    private static final QName SERVICE_NAME = new QName("http://dashboardRt.wsbridge.soap/", "Dashboard");
    private static final QName PORT_NAME    = new QName("http://dashboardRt.wsbridge.soap/", "DashboardPort");

    StreamSource stream,stream2;
    OutputStream os;
    InputStream bytes;
    ArrayList<Arguments> outputsArray = new ArrayList<Arguments>();


    public void invokeService(String url, String service,
            Map<String, Object> inputs,
            Map<String, Object> outputs) throws JAXBException
    {


        Source request = null;
        Source response = null;

        // (1) C—digo de inicializaci—n para crear un objeto Dispatch<Source> que se
        // utiliza para realizar la petici—n de servicio

        Service srv = Service.create(SERVICE_NAME);
        srv.addPort(PORT_NAME, SOAPBinding.SOAP11HTTP_BINDING, url);
        Dispatch<Source> sourceDispatch = srv.createDispatch(PORT_NAME, Source.class, Service.Mode.PAYLOAD);
```

Figura 4.8.: Fragmento de código de la clase ServiceDelegator.

El cliente ejecuta el Servicio Web de manera transparente, como si fuera un servicio local, mediante el método invokeService(). Este método (véase Figura 4.8) tiene como argumentos el mapa "inputs" que se rellenará con la información que se envía al servicio y el mapa "outputs" con la infomación resultante recibida (el cliente pasa incialmente este mapa vacío). Notar que además invokeService tiene un argumento para indicar el nombre del proceso que queremos invocar. Ello se debe a que consideramos que un mismo Servicio Web podría ejecutar varios procesos, por ejemplo, varias transformaciones de modelos.

En general el método invokeService crea el Message, lo envía al Servicio Web, obtiene la respuesta y devuelve su contenido rellenando el mapa outputs. Para que el mensaje pueda ser procesado por el Servicio Web, éste ha de tener un formato "entendible" por dicho servicio, para dicha tarea el mensaje ha de ser serializado. Serializar el mensage es pasarlo a formato XML, como vimos disponemos de la API JAXB para dicha tarea. Puede verse un esquema de las acciones que lleva a cabo la clase ServiceDelegator (y la clase ServiceProvider) en la Figura 4.12.

La clase ***Client*** inicia la llamada al servicio con los datos de entrada que se enviarán a través del mensaje. A pesar de que esta clase contiene un método main para facilitar las pruebas, podría integrarse el cliente en la herramienta HuRoME.

```
inputs.put("parametro_ejemplo_1", new Integer(23));
inputs.put("parametro_ejemplo_2", new Double(10.5));
inputs.put("parametro_ejemplo_3", new Boolean(true));
inputs.put("parametro_ejemplo_4", new Boolean(false));
inputs.put("Ejemplo STATUS", status);
inputs.put("Ejemplo MODEL", model);
inputs.put("Ejemplo TEXT", text);


// Invocaci—n del servicio
delegator.invokeService(ENDPOINT_ADDR, "servicio_ejemplo", inputs, outputs);
```

Figura 4.9.: Datos de envío.

La clase ***Server*** especifica mediante ENDPOINT_ADDR la URL donde atenderá peticiones a los clientes y se establece un tiempo de cinco minutos en el que el servidor estará atendiento peticiones (Vease que se crea un *ServiceProvider*(), es el encargado de atender las peticiones a través del ENDPOINT_ADDR), una vez pasado ese tiempo se deberá reiniciar el servidor o cambiar el código para aumentar el tiempo de escucha mediante el hilo Thread.

```
public class Server {

    /*
     * Direcci—n del servicio
     */
    public final String ENDPOINT_ADDR = "http://localhost:9000/dashboard";

    protected Server() {

        System.out.println("Starting Server");

        // Publicaci—n del servicio web
        ServiceProvider implementor = new ServiceProvider();
        Endpoint.publish(ENDPOINT_ADDR, implementor);
    }

    public static void main(String[] args) throws JAXBException, FileNotFoundException, InterruptedException {

        new Server();
        System.out.println("Server ready...");

        // El servidor permanece activo durante un periodo de 5 min
        Thread.sleep(5 * 60 * 1000);

        System.out.println("Server exiting");
        System.exit(0);
    }

}
```

Figura 4.10.: Clase Server.

La clase ***ServiceProvider*** es la clase que se encarga de ejecutar la operación que el cliente demanda. En la Figura 4.11 puede observarse una serie de anotaciones en la implementación de esta clase. Estas anotaciones pertenecen a la API JAX-WS, a continuación las describimos.

- **@WebServiceProvider.** Mediante esta anotación de indica el puerto donde escucha el ServicioWeb (atributo portName), el nombre del servicio (atributo serviceName) y el espacio de nombres (atributo targetNameSpace)
- **@BindingType.** En esta anotación describimos que el protocolo de transporte de los mensajes será Http, como hemos visto SOAP no se encuentra fuertemente asociado a ningún protocolo de transporte: La especificación de SOAP no describe como se deberían asociar los mensajes de SOAP con HTTP. Un mensaje de SOAP no es más que un documento XML, por lo que puede transportarse utilizando cualquier protocolo capaz de transmitir texto.
- **@ServiceMode,** se ha seleccionado el tipo PAYLOAD, lo que quiere decir que el proveedor va a trabajar con la carga de los mensajes (PAYLOAD).

En la Figura 4.12 se representa el esquema básico de acciones que se llevan a cabo en ServiceProvider. Además de la deserialización y serialización de mensajes, esta clase invoca el método invokeService(), éste implementa en el servidor las transformaciones de modelos de la herramienta HuRoME que han de ejecutarse. Comentar que la implementación de estas transformaciones en el servidor queda fuera del ámbito del presente proyecto, para este cometido se ha utilizado el framework Dashboard Runtime [35].

Para concluir y poder reutilizar métodos en varias clases, se crea la clase ***ServiceUtil*** para reunir el conjunto de métodos que resultan de utilidad general. Estos métodos son: (1) *Argument2Message*(), que dado un mapa con datos de entrada (inputs) crea un objeto Message; (2) *mesage2Argument*(), que realiza la operación inversa; (3) *serializeMessage*(), que serializa un objeto Message a código XML; (4) *deserializeMessage*(), para realizar la deserialización; (5) *showXML*(), dado un Message podemos ver por pantalla su representación en XML.

```
@WebServiceProvider(portName = "DashboardPort",
    serviceName = "Dashboard",
    targetNamespace = "http://example.com/request/wsdl")
@BindingType(value = "http://schemas.xmlsoap.org/wsdl/soap/http")
@ServiceMode(value = javax.xml.ws.Service.Mode.PAYLOAD)
public class ServiceProvider implements Provider<Source> {

    Message ms;
    Map<String, Object> inputs,outputs;
    Source d = null;
    ArrayList<Arguments> inputsArray = new ArrayList<Arguments>();
```

Figura 4.11.: Anotaciones servidor.

4.4 Creación de un Servicio Web RESTful

En este punto se aborda la creación de un Servicio Web basado en REST. El diagrama UML de clases utilizado para implementar el Servicio Web RESTful está representado en la Figura 4.13. Se puede ver cómo éste es muy similar al empleado en SOAP se debe a que hemos intentado mantener y reutilizar la infraestructura básica del Servicio Web. Este apartado comparterá muchos otros conceptos y explicaciones dadas anteriormente para SOAP.

La clase ***Servidor*** ya no forma parte del diagrama UML, esto es porque para el tipo REST, las clases ServiceDelagator y ServiceProvider utilizan la implementación del cliente y servidor (respectivamente) proporcionada por la API JAX-RS. Por lo tanto, el Servicio Web se configura y despliega directamente en el servidor Tomcat sin requerir de un método main para dichas tareas.

La clase ***ServiceProvider*** es la que se montará en Tomcat para que escuche peticiones de los clientes y ejecute el servicio. Esta clase contiene un conjunto de anotaciones, definidas por la API JAX-RS, para especificar los recursos REST del servicio y sus rutas de acceso. Estas anotaciones son:

- **@Path** indica que el elemento asociado es accesible a través de la ruta establecida por los clientes para obtener el servicio. Esta anotación puede preceder a una clase y/o a una método, teniendo en cuenta que cada vez que aparece significa que el servidor resuelve cada una de las rutas indicadas asociándolas a operaciones HTTP a recursos.
- **@Get,** como se ha visto en la revisión de REST en el capítulo 2, esta tecnología utiliza un conjunto de operaciones establecido en estándar http, de las cuales las más importantes son GET, PUT, POST y DELETE. Esta anotación indica que un recurso concreto en el servidor (que puede ser un modelo resultado de aplicar una transformación) puede ser solicitado a través de una operación GET. La diferencia con la operación POST (cuya etiqueta se presentará más adelante) es que GET pasa el valor de los argumentos de entrada a través de la URL, es decir, éstos pueden verse concatenados al final de la dirección. Sin embargo, POST los oculta en la petición http.

El hecho de que GET exponga los argumentos podría ocasionar problemas de privacidad.

- **@Consumes** y **@Produces** definen el tipo de MIME consume y produce una operación, de forma predeterminada puede consumir y producir cualquier tipo de MIME. En el caso del presente proyecto, se ha utilizado el formato JSON.

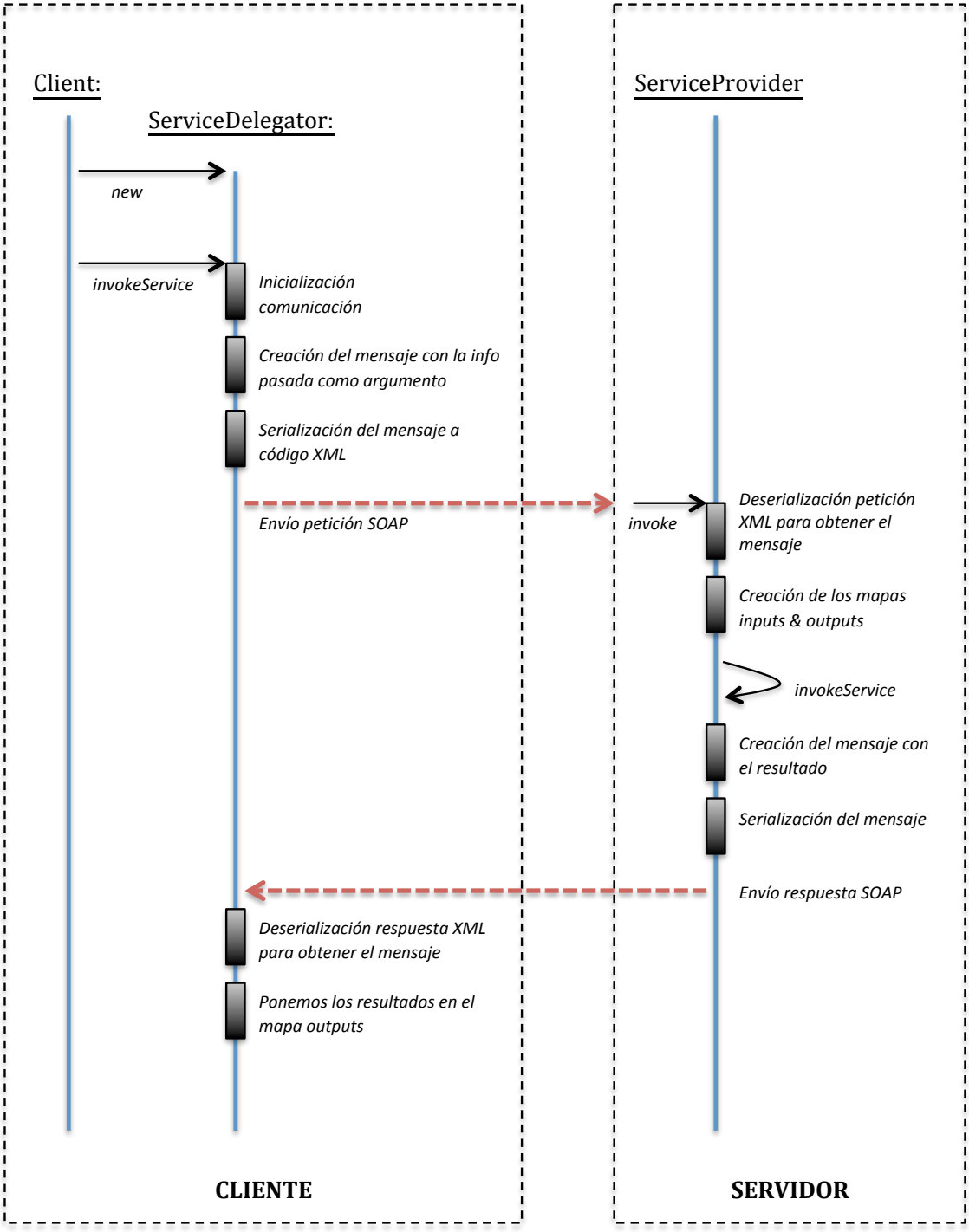

Figura 4.12.: Esquema general de acciones llevadas a cabo en ServiceDelagator y ServiceProvider

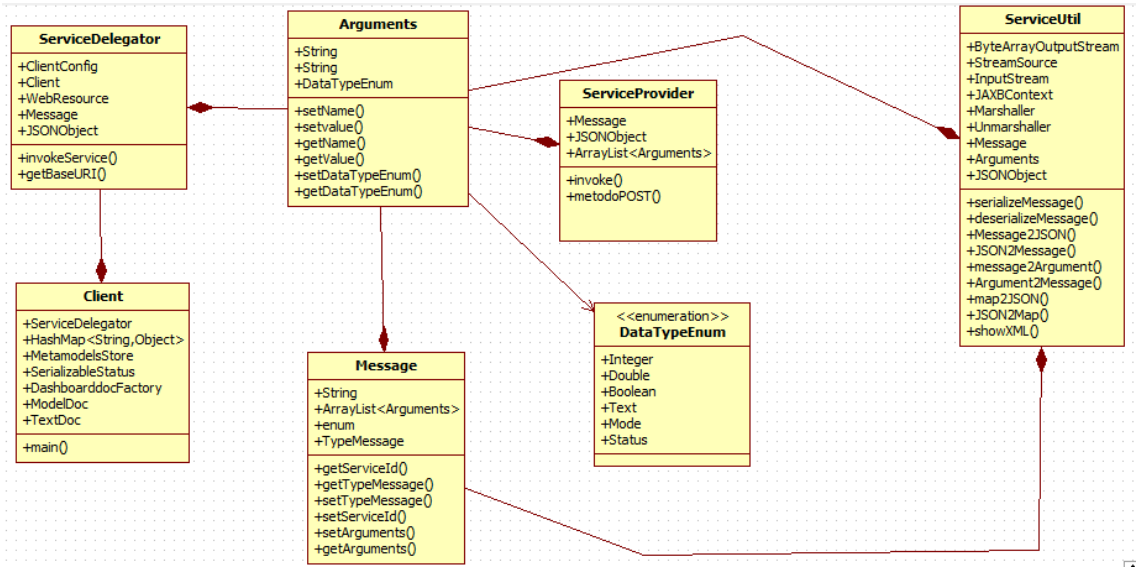

Figura 4.13.: Diagrama de clases del Servicio Web implementado con REST

- **@QueryParam**, esta anotación indica el identificador del argumento que será pasado por la URL.
- **@Post**, como se ha dicho en la definición de @Get, disponemos de las operaciones que el protocolo http proporciona. Esta anotación equivale a la operación http POST, que se traduce por actualizar un recurso solicitado. Como en el caso de sus homólogas, la anotación precede al método que se ejecutara cuando se invoca POST.

Para comparar el rendimiento entre un servicio implementado con GET o POST, hemos desarrollado el mismo servicio con ambas operaciones. En el siguiente capítulo se muestran los resultados obtenidos de esta comparativa. Las Figuras 4.14 y 4.15 muestran un extracto del código de la clase ServiceProvider, podemos observar el uso de las etiquetas anteriores para implementar el servicio con la operación GET (Figura 4.14) como con POST (Figura 4.15).

La clase ***Cliente*** permanece inalterada respecto del caso SOAP, rellena los datos del mapa inputs con los que se creará la petición, hace la llamada al servicio con estos datos mediante invokeService() de ServiceDelegator y muestra el resultado de la operación ejecutada en el Servicio Web.

En la clase ***ServiceDelegator,*** haciendo uso de la API JAX-RS se crea el objeto de tipo WebResource llamado serviceClient, que será el encargado de realizar la petición al Servicio Web. Para ello, utilizando los datos obtenidos a través de invokeService() se crea el mensaje con la información. Como hemos visto en la deficicion de REST, esta arquitectura soporta el intercambio de mensajes del tipo XML y JSON, para esta implementación se ha elegido el intercambio de mensaje JSON. Haciendo uso del método de conversión, implementado en ServiceUtil y llamado Message2JSON, se formatea el mensaje a tipo JSON. Como ya se adelantó en la creación del Servicio Web, para el caso GET, el JSON irá contenido en la URL para realización del servicio. ServiceClient resuelve la URL utilizando los métodos path() y añade el archivo JSON en el método queryParam(). Para finalizar recibe la respuesta en formato JSON, lo pasa a tipo Message mediante JSON2Message de serviceUtil y pasa los datos a la clase Client para que los muestre. Para el caso POST, la información va contenida en el cuerpo del

mensaje, no como en el caso GET que va en la URL. El objeto encargado de realizar la petición nos lo proporciona la API JAX-RS y es del tipo ClientResponse, este resuelve la URL donde se ubica el Servicio Web mediante los métodos path() y especifica que el método que ha de invocar va seguido de la anotación @Post mediante el método post() en el cual se introduce como argumento de entrada la información que recibirá el método del Servicio Web. En la Figura 4.16 puede verse un extracto de código de la clase ServiceDelegator.

```
@Path("/servidor")
public class ServiceProvider  {

    JSONObject back;
    Map<String, Object> inputs,outputs;
    Message msg,ms;

    /**
     * @category invoke es parte de la ruta de acceso al metodo, GET el tipo de operacion HTTP resultante
     * , y QueryParam quiere decir que los datos para que se ejecute la operación son pasados a través de
     * @param request
     *
     */
    @Path("invoke")
    @GET
    @Produces(MediaType.APPLICATION_JSON)
    public String invoke(@QueryParam("request") String request) throws JAXBException, JSONException {

        JSONObject json = new JSONObject(request);
        //Crear los mapas a partir de la info contenida en el mensaje
        msg = ServiceUtil.JSON2Message(json);

        inputs = new HashMap<String, Object>();
        outputs = new HashMap<String, Object>();
        invokeService(msg.getServiceId(),inputs,outputs);
        ms = new Message();
        ms.setServiceId(msg.getServiceId());
        ms.setTypeMessage(Message.typeMessage.response);
        back = ServiceUtil.Message2JSON(ms);
        return back.toString();

    }
```

Figura 4.14.: Servicio Web utilizando la operación GET

```
@Path("invoke")
@POST
@Produces(MediaType.TEXT_PLAIN)
@Consumes(MediaType.TEXT_PLAIN)
public String metodoPOST(String request) throws IOException, JSONException {

    json = new JSONObject(request);
    //Crear los mapas a partir de la info contenida en el mensaje
    System.out.println("El JSON DE POST TIENE"+request);

    msg = ServiceUtil.JSON2Message(json);

    inputs = new HashMap<String, Object>();
    outputs = new HashMap<String, Object>();
    inputs= ServiceUtil.JSON2Map(json);

    invokeService(msg.getServiceId(),inputs,outputs);
    ms = new Message();
    ServiceUtil.Argument2Message(outputs, ms);
    ms.setServiceId(msg.getServiceId());
    ms.setTypeMessage(Message.typeMessage.response);

    back = ServiceUtil.Message2JSON(msg);
    return back.toString();
```

Figura 4.15.: Servicio Web utilizando la operación POST

Las clases ***Arguments***, ***Message*** y el enumerador ***DataTypeEnum*** permanecen igual que en el caso SOAP.

La clase ***ServiceUtil*** conserva los métodos que se crearon para el caso SOAP más otros específicos que se han añadido para el caso REST. Estos métodos son: (1) *JSON2Map*(), necesario para pasos intermedios de conversión entre tipos; (2) *Map2JSON*(), realiza la operación inversa que el método anterior; (3) *JSON2Message*(), realiza la conversión entre el tipo Message y JSON, uno es el contenedor de datos para mostrar y otro para intercambio entre cliente-servidor; (4) *Message2JSON*(), realiza la operación inversa al método anterior, nos proporcióna el archivo JSON; (5) *showJSON*(), este método ha sido creado para poder visualizar como es una archivo JSON, es decir como está formateado; (6) *showMessage*(), este método ha sido creado para ver el contenido de un mensaje.

Por último, comentar que a diferencia del caso anterior que el servidor se arracancaba ejecutando la clase Server, en este caso hay que seleccionar con el botón derecho la opción de menú Run As->Run On Server. Haciendo esto el Servicio Web REST arrancará y estará listo para atender peticiones.

```
public class ServiceDelegator {

    public void invokeService(String url, String service,
            Map<String, Object> inputs,
            Map<String, Object> outputs) throws JAXBException, JSONException
    {
        /**
         * Creamos el Servicio Web
         */
        ClientConfig config = new DefaultClientConfig();
        Client client = Client.create(config);
        WebResource serviceClient = client.resource(getBaseURI());
        /**
         * Creación del mensaje con los datos
         */
        Message msg = new Message();
        Message msg2;
        msg.setServiceId(service);
        msg.setTypeMessage(typeMessage.Request);
        ServiceUtil.Argument2Message(inputs, msg);
        /**
         * Una de las diferencias de este Servicio Web respecto del SOAP es el tipo de mensaje que usamos para el intercambio, en este caso es JSON
         */
        JSONObject json = ServiceUtil.Message2JSON(msg);
        /**
         * Sistema de realizar la petición, accedemos a través del Path (Podemos visualiarlo también en el navegador), y en la Query (petición) enviamos el archivo JSON
         */
        String response = serviceClient.path("rest").path("servidor").path("invoke").queryParam("request", json.toString()).accept(MediaType.APPLICATION_JSON).get(String.class);

        /**
```

Figura 4.16.: Clase ServiceDelegator.

CAPÍTULO V

Pruebas y resultados

En este capítulo se han realizado una serie de pruebas para evaluar los Servicios Web creados. Básicamente, como detallaremos más adelante, las pruebas han consistido en ejecutar dos transformación, comprobar la corrección de los resultados y tomar medidas relativas al tamaño de los paquetes intercambiados durante la comunicación.

5.1 Consideraciones iniciales

Para la realización de las pruebas se han utilizado los siguientes componentes hardware y software.

5.1.1 Monitor de red WireShark

Es un analizador de protocolos que se utiliza para realizar análisis y solucionar problemas en redes de comunicaciones. Permite monitorizar redes inalámbricas Wi-Fi. Se presenta con una interfaz gráfica en la que podemos seleccionar diferentes opciones de filtrado y en la que se representa la información capturada en forma de tabla. Cada fila identifica cada uno de los paquetes capturados. Las capturas realizadas pueden ser guardadas en el disco, pudiendo así, ser analizada posteriormente. La Figura 5.3 muestra la interfaz principal de WireShark.

5.1.2 Ordenadores utilizados

Para la realización de las pruebas se han utilizado dos ordenadores portátiles (en adelante PC1 y PC2). El PC1 ha hecho las funciones de Servidor Web y el PC2 las de cliente del Servicio.

Para interconectar ambos ordenadores, se ha creado una red inalámbrica con dirección privada 192.168.1.0 de clase C con máscara de red 255.255.255.0, el PC1 atiende peticiones a través de la interfaz 192.168.1.2 y el PC2 realiza las peticiones a través de la interfaz 192.168.1.23. Las características de estos ordenadores se mencionan a continuación.

- **PC1:** Este ordenador es un Toshiba Satellite L750/L755, con un procesador Intel Core i5-2450M, con RAM de 8GB a 1333Mhz DDR3 de dos Slots, disco duro de 640GB – 5400RPM SATA, puerto Ethernet 10/100 y Wi-Fi Wireless networking (802.11b/g/n). El sistema operativo que utiliza es Windows 7.
- **PC2:** Este ordenador es un Macbook Pro 2011, con un procesador Intel Core i7 a 2.2GHz, con RAM de 4GB a 1333Mhz DDR3 de dos Slots, disco duro de 500GB – 5400RPM SATA, puerto Ethernet 10/100/1000 y Wi-Fi Wireless networking (802.11b/g/n). El sistema operativo que utiliza es Mac OS X Lion.

5.2 Descripción de las pruebas

En el presente proyecto se ha desarrollado un Servicio Web con dos tecnologías diferentes, SOAP y REST. En el PC1 ejecutará el Servicio Web y en el PC2 el cliente del Servicio. El Servicio Web simula la implementación de dos transformaciones del entorno HuRoME, la primera, modelo-a-texto donde el cliente envía un modelo Hurome y recibe como respuesta el código RoboBasic generado. La segunda, una transformación texto-a-modelo donde el cliente envía un documento con código RoboBasic y recibe el modelo Hurome generado. Para más información sobre estas transformaciones véase el capítulo 3, en concreto, las secciones 3.3.2 y 3.3.3. Comentar que para facilitar el manejo de documentos de código RoboBasic, éstos son representados como modelos EMF que contienen el código como un atributo de tipo string. Los detalles sobre el meta-modelo de estos documentos así como la implementación de las transformaciones queda fuera del ámbito del presente proyecto. Para más información véase el framework Dashboard Runtime [35].

Se realizarán las pruebas considerando dos formatos distintos para los documentos (modelos y textos) intercambiados en la comunicación: (1) un formato XML (XMI) y (2) un formato binario (implementado por el framework EMF [36]). Las Figuras 5.1 y 5.2 muestran el contenido en formato XMI del documento RoboBasic y el modelo Hurome utilizados en las pruebas. En la Tabla 5.1 se especifican los tamaños de ambos documentos para los formatos considerados.

	Hurome	**RoboBasic**
Formato XMI	4307 bytes	1413 bytes
Formato Binario	705 bytes	1215 bytes

Tabla 5.1.: Tamaño documentos intercambiados en la comunicación

Las pruebas a realizar son las siguientes:

- **Prueba SOAP M2T / T2M:** Transformación modelo-a-texto (Hurome a RoboBasic) y texto-a-modelo (RoboBasic a Hurome) invocando el Servicio Web SOAP. Los documentos se codifican en formato XMI. Notar que en el caso de SOAP los

documentos sólo pueden ser transmitidos en formato XMI, ello es debido a que los mensajes SOAP que contienen estos documentos son serializados a XML y, según el estándar, este lenguaje considera inválidos una serie de caracteres especiales utilizados en el formato binario. Por lo tanto, el formato binario que utiliza EMF para serializar modelos es incompatible con los mensajes XML de SOAP.

- **Prueba REST GET M2T/T2M – Documentos en XMI:** Transformación modelo-a-texto (Hurome a RoboBasic) y texto-a-modelo (RoboBasic a Hurome) invocando la operación GET del Servicio Web REST. Los documentos se codifican en formato XMI.
- **Prueba REST GET M2T/T2M – Documentos en Binario:** Transformación modelo-a-texto (Hurome a RoboBasic) y texto-a-modelo (RoboBasic a Hurome) invocando la operación GET del Servicio Web REST. Los documentos se codifican en formato binario.
- **Prueba REST POST M2T/T2M – Documentos en XMI:** Transformación modelo-a-texto (Hurome a RoboBasic) y texto-a-modelo (RoboBasic a Hurome) invocando la operación POST del Servicio Web REST. Los documentos se codifican en formato XMI.
- **Prueba REST POST M2T/T2M – Documentos en Binario:** Transformación modelo-a-texto (Hurome a RoboBasic) y texto-a-modelo (RoboBasic a Hurome) invocando la operación POST del Servicio Web REST. Los documentos se codifican en binario.

```
<?xml version="1.0" encoding="ASCII"?>
<dashboarddoc:TextDoc xmi:version="2.0" xmlns:xmi="http://www.omg.org/XMI" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:dashboar
xsi:schemaLocation="http://dashboarddoc/1.0 ../metamodels/dashboardDoc.ecore" name="SimpleDance" userpath="./SimpleDance">
  <resources xsi:type="dashboarddoc:Archive" name="SimpleDance.rsf" codification="" content="SPEED 6&#xA;MOVE24  85,  71, 152,  91, 112,
  60, 100,  40,  80, 100, 100, 100, 100,  40,  80, 100, 100, 100, 112,  76, 145,  93,  92,  60,&#xA;MOVE24  76,  72, 160,  82, 128,  70, 100,
   100, 100, 100, 100,  35,  90, 100, 100, 100, 110,  92, 124,  97,  93,  70,&#xA;MOVE24  93,  76, 145,  94, 109, 100, 100,  35,  90, 100, 100,
    35,  90, 100, 100, 100,  93,  76, 145,  94, 109, 100,&#xA;&#xA;MOVE24  93,  76, 145,  94, 109, 100, 100, 105, 100, 100, 100, 100, 100, 105,
    100, 100,  93,  76, 145,  94, 109, 100,&#xA;delay 2000&#xA;&#xA;MOVE24  93,  76, 145,  94, 109, 100, 100,  35,  90,100, 100, 100, 100,  35,
     100,  93,  76, 145,  94, 109, 100,&#xA;MOVE24  76,  72, 160,  82, 128,  70, 100,  35,  90,100, 100, 100,100,  35,  90,  100, 100, 100,110,
     7,  93,  70,&#xA;MOVE24  85,  71, 152,  91, 112,  60, 100,  40,  80,100, 100, 100, 100,  40,  80,  100, 100, 100, 112,  76, 145,  93,  92,
     &#xA;SPEED 6&#xA;MOVE24 100,  76, 145,  93, 100, 100, 100,  30,  80, 100, 100, 100, 100,  30,  80, 100, 100, 100, 100,  76, 145,  93, 100,
</dashboarddoc:TextDoc>
```

Figura 5.1.: robobasic.xmi

```
<?xml version="1.0" encoding="UTF-8"?>
<hurome:ComplexMovement xmi:version="2.0" xmlns:xmi="http://www.omg.org/XMI" xmlns:xsi="http://www.w3.org/2001/XMLSchem
xmlns:hurome="http://hurome/1.0" name="modelo1">
  <links speed="8" delay="1" source="//@movements.0" target="//@movements.1"/>
  <links speed="7" delay="2" source="//@movements.1" target="//@movements.2"/>
  <links speed="6" delay="3" source="//@movements.2" target="//@movements.3"/>
  <links speed="1" delay="8" source="//@movements.4" target="//@movements.5"/>
  <movements xsi:type="hurome:InitialPseudoMovement" name="Inicial" outgoing="//@links.0"/>
  <movements xsi:type="hurome:SimpleMovement" name="simple1" incoming="//@links.0" outgoing="//@links.1"/>
  <movements xsi:type="hurome:SimpleMovement" name="simple2" incoming="//@links.1" outgoing="//@links.2"/>
  <movements xsi:type="hurome:ComplexMovement" incoming="//@links.2" outgoing="//@movements.3/@links.3">
    <links speed="4" delay="5" source="//@movements.3/@movements.0" target="//@movements.3/@movements.1"/>
    <links speed="5" delay="4" source="//@movements.3/@movements.2" target="//@movements.3/@movements.0"/>
    <links speed="3" delay="6" source="//@movements.3/@movements.1" target="//@movements.3/@movements.3"/>
    <links speed="2" delay="7" source="//@movements.3" target="//@movements.4"/>
    <movements xsi:type="hurome:SimpleMovement" name="simple3" incoming="//@movements.3/@links.1" outgoing="//@movement
    <movements xsi:type="hurome:SimpleMovement" name="simple4" incoming="//@movements.3/@links.0" outgoing="//@movement
    <movements xsi:type="hurome:InitialPseudoMovement" name="Inicial" outgoing="//@movements.3/@links.1"/>
    <movements xsi:type="hurome:FinalPseudoMovement" name="Final" incoming="//@movements.3/@links.2"/>
  </movements>
  <movements xsi:type="hurome:SimpleMovement" name="simple5" incoming="//@movements.3/@links.3" outgoing="//@links.3"/>
  <movements xsi:type="hurome:FinalPseudoMovement" name="Final" incoming="//@links.3"/>
</hurome:ComplexMovement>
```

Figura 5.2.: hurome.xmi

5.3 Realización de las pruebas

5.3.1 Prueba SOAP M2T / T2M

Se han realizado la ejecución del Servicio Web para cada una de las dos transformaciones. Cabe recordar que SOAP utiliza mensajes XML, por lo tanto, dada la incompatibilidad de algunos de los caracteres del formato binario, los documentos son codificados sólo en XMI.

- **Hurome2RoboBasic (M2T),** en este caso, el cliente envía el modelo hurome y el servidor le contesta con el documento roboBasic. Así pues, al analizar las tramas capturadas por WireShark, observamos que se establece un esquema de comunicación cliente-servidor, en el que el cliente envía una petición http post de tamaño 5127 bytes fragmentada en 5 paquetes y el servidor envía una respuesta http de tamaño 2037 bytes fragmentada en 4 paquetes.
- **RoboBasic2Hurome (T2M),** en este caso, el cliente envía el documento roboBasic y el servidor le contesta con el modelo hurome. Observamos en el análisis de las tramas que el cliente envía una petición http post de 2183 bytes fragmentada en 3 paquetes y recibe una respuesta http de 5006 bytes fragmentada en 3 paquetes.

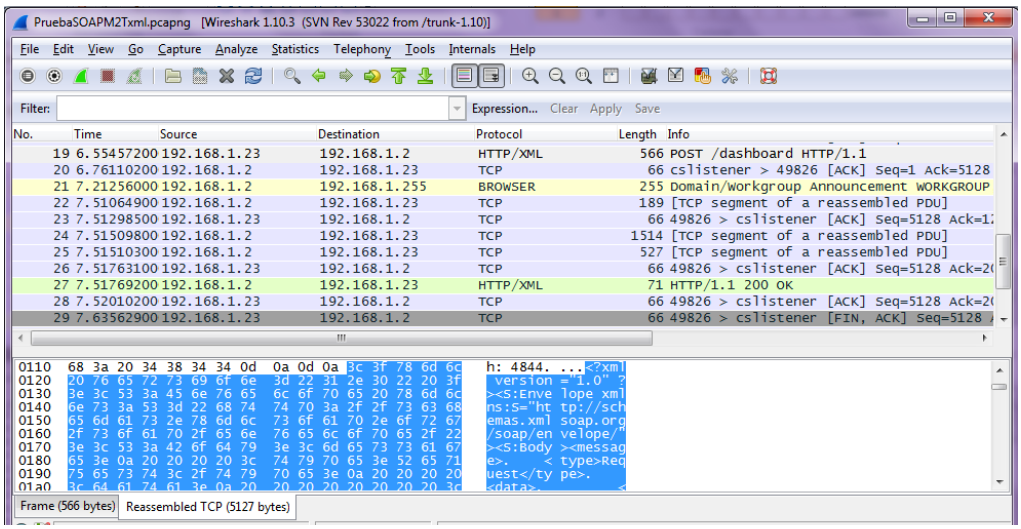

Figura 5.3.: Prueba SOAP hurome2RoboBasic

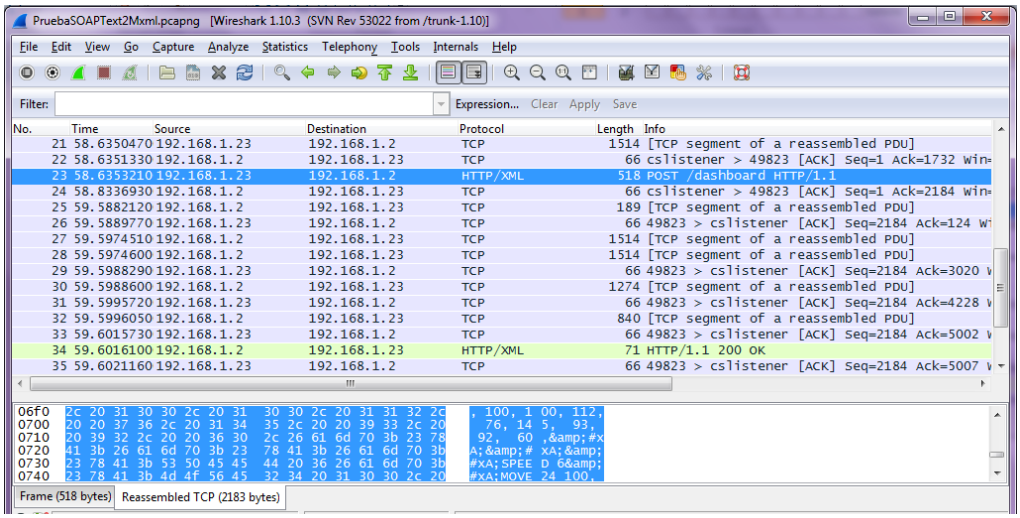

Figura 5.4.: Prueba SOAP RoboBasic2hurome

Se aprecia que los tamaños de los mensajes enviados son mayores respecto al tamaño de los documentos mostrados en la Tabla 5.1. Esto es debido a que estos documentos para ser enviados se encapsulan en el mensaje XML (ver descripción de la clase Message en el capítulo anterior). Este mensaje no sólo contiene el documento en cuestión, sino que también alberga otra información, como el nombre y tipo de los argumentos enviados, el identificador del servicio a ejecutar, etc.

5.3.2 Prueba REST GET M2T/T2M – Documentos en XMI

Para esta prueba, se ha invocado el Servicio Web REST a través de la operación GET. El formato de los mensajes que se intercambian es JSON. Los documentos aparecen codificados dentro del mensaje en formato XMI.

- **Hurome2RoboBasic (M2T),** en este caso, el cliente envía el modelo hurome y el servidor le contesta con el documento roboBasic. Así pues, al analizar las tramas capturadas por WireShark, observamos que se establece un esquema de comunicación cliente-servidor, en el que el cliente envía una petición http get de tamaño 7509 bytes fragmentada en 5 paquetes y el servidor envía una respuesta http de tamaño 1722 bytes fragmentada en 3 paquetes.
- **RoboBasic2Hurome (T2M),** en este caso, el cliente envía el documento roboBasic y el servidor le contesta con el modelo hurome. Observamos en el análisis de las tramas que el cliente envía una petición http get de 1985 bytes fragmentada en 2 paquetes y recibe una respuesta http de 2609 bytes fragmentada en 3 paquetes.

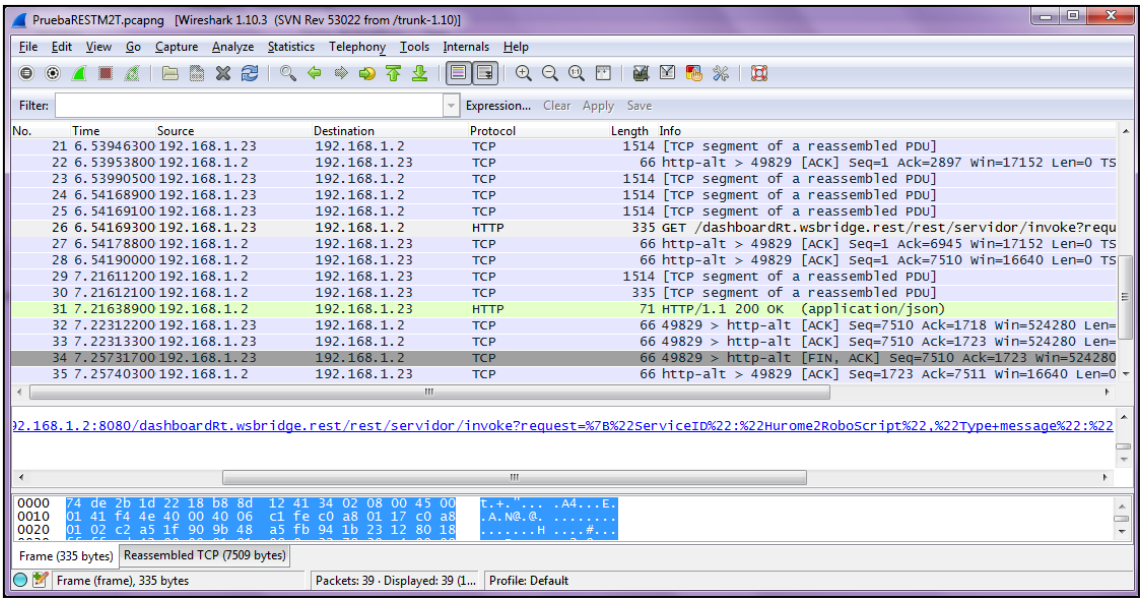

Figura 5.5.: Prueba REST GET hurome2RoboBasic – documentos en XMI

Se aprecia que el tamaño de la petición asociado a la transformación M2T sufre un aumento significativo en su tamaño. Dado que JSON es un formato más compacto que XML, en principio todos los mensajes intercambiados deberían haber sido menores que en el caso de SOAP. No obstante, debemos considerar que, por un lado, los documentos (que ocupan el grueso del mensaje) están codificados de la misma forma en ambos casos y, por otro, en GET los datos de las peticiones se transmiten a través de la URL. Al codificar los datos en la URL, se insertan caracteres extra para preceder espacios y otros caracteres especiales, por lo tanto, el tamaño

del paquete puede crecer a causa de esta codificación, tal y como se observa en la petición en la transformación M2T.

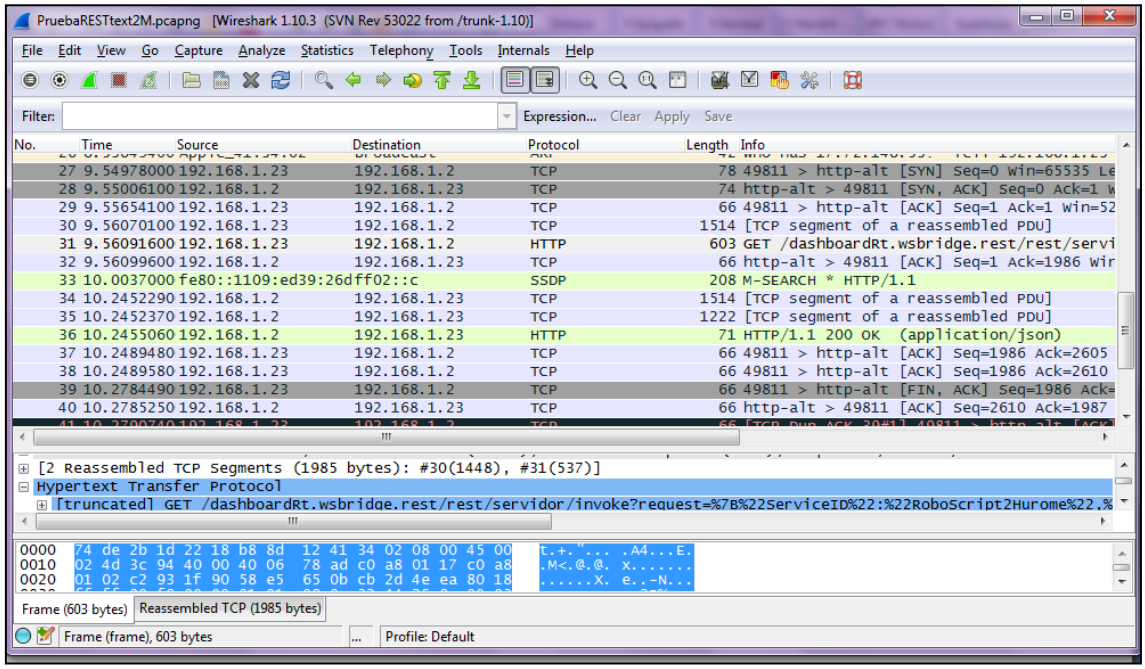

Figura 5.6.: Prueba REST GET RoboBasic2hurome – documentos en XMI

5.3.3 Prueba REST GET M2T/T2M – Documentos en binario

Para esta prueba, se ha invocado el Servicio Web REST a través de la operación GET. El formato de los mensajes que se intercambian es JSON. Los documentos aparecen codificados dentro del mensaje en formato binario.

- **Hurome2RoboBasic (M2T),** en este caso, el cliente envía el modelo hurome y el servidor le contesta con el documento roboBasic. Así pues, al analizar las tramas capturadas por WireShark, observamos que se establece un esquema de comunicación cliente-servidor, en el que el cliente envía una petición http get de tamaño 6132 bytes fragmentada en 5 paquetes y el servidor envía una respuesta http de tamaño 1648 bytes fragmentada en 3 paquetes.
- **RoboBasic2Hurome (T2M),** en este caso, el cliente envía el documento roboBasic y el servidor le contesta con el modelo hurome. Observamos en el análisis de las tramas que el cliente envía una petición http get de 1847 bytes fragmentada en 2 paquetes y recibe una respuesta http de 1112 bytes fragmentada en 2 paquetes.

En esta prueba puede observarse que el tamaño de la información intercambiada disminuye gracias al menor tamaño de los documentos codificados en binario. No obstante, puede verse que las peticiones no se han reducido demasiado, incluso, podemos apreciar que la petición asociada a la transformación M2T es sustancialmente mayor que la correspondiente petición en SOAP. Tal y como se ha explicado anteriormente, esto es debido a la re-codificación que sufren los documentos para su envío por URL, donde se añaden caracteres extra para introducir espacios y otros símbolos especiales.

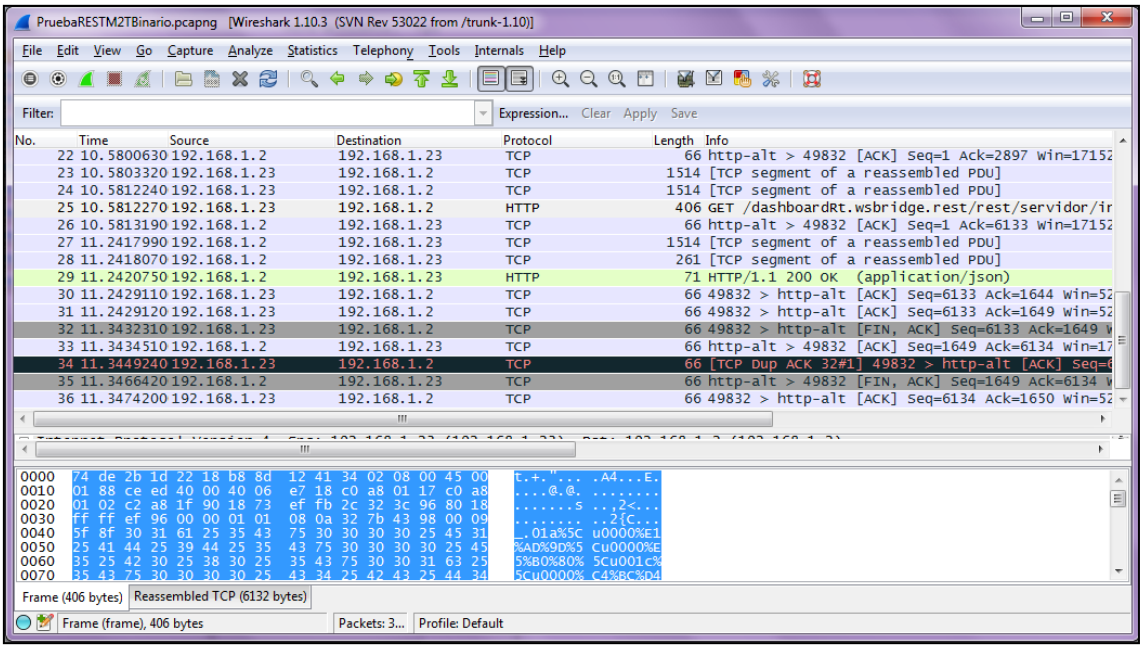

Figura 5.7.: Prueba REST GET hurome2RoboBasic – documentos en binario

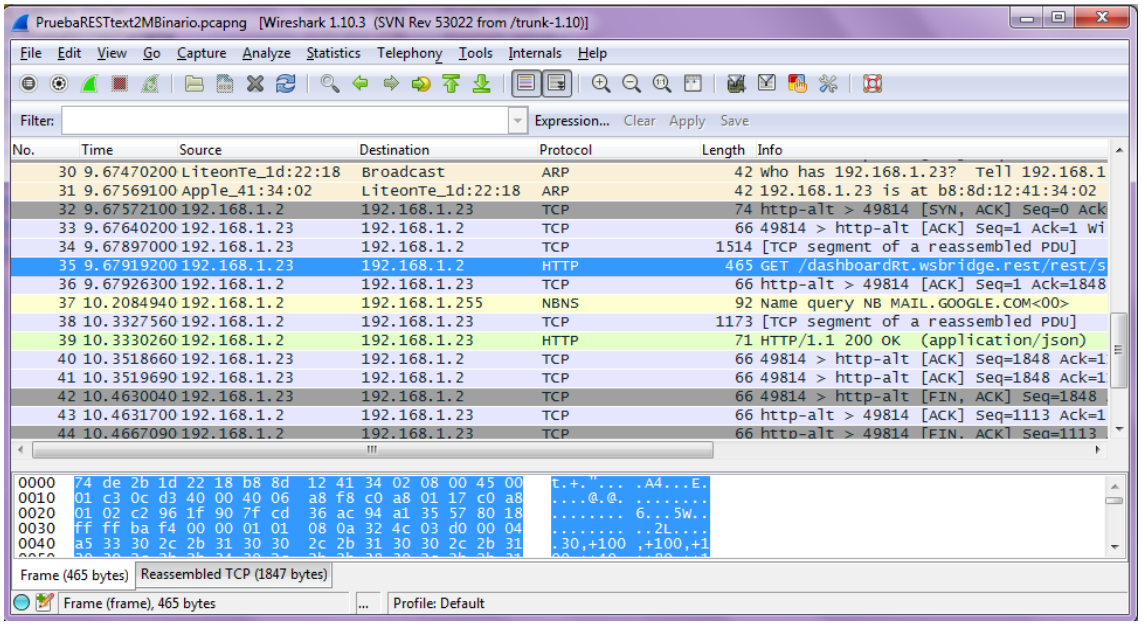

Figura 5.8.: Prueba REST GET RoboBasic2hurome – documentos en binario

5.3.4 Prueba REST POST M2T/T2M – Documentos en XMI

Para esta prueba, se ha invocado el Servicio Web REST a través de la operación POST. El formato de los mensajes que se intercambian es JSON. Los documentos aparecen codificados dentro del mensaje en formato XMI.

- **Hurome2RoboBasic (M2T),** en este caso, el cliente envía el modelo hurome y el servidor le contesta con el documento roboBasic. Así pues, al analizar las tramas capturadas por WireShark, observamos que se establece un esquema de comunicación cliente-servidor, en el que el cliente envía una petición http post de tamaño 5157 bytes

fragmentada en 5 paquetes y el servidor envía una respuesta http de tamaño 1642 bytes fragmentada en 3 paquetes.

- **RoboBasic2Hurome (T2M),** en este caso, el cliente envía el documento roboBasic y el servidor le contesta con el modelo hurome. Observamos en el análisis de las tramas que el cliente envía una petición http post de 1771 bytes fragmentada en 3 paquetes y recibe una respuesta http de 2603 bytes fragmentada en 3 paquetes.

A modo de conclusión, se ve que el tamaño de paquete en la petición es menor que en el caso GET, esto es debido a que los datos van en el cuerpo del mensaje y no en la URL. Podemos decir que, en general, la comunicación sería más eficiente que los casos anteriormente analizados (SOAP y REST GET).

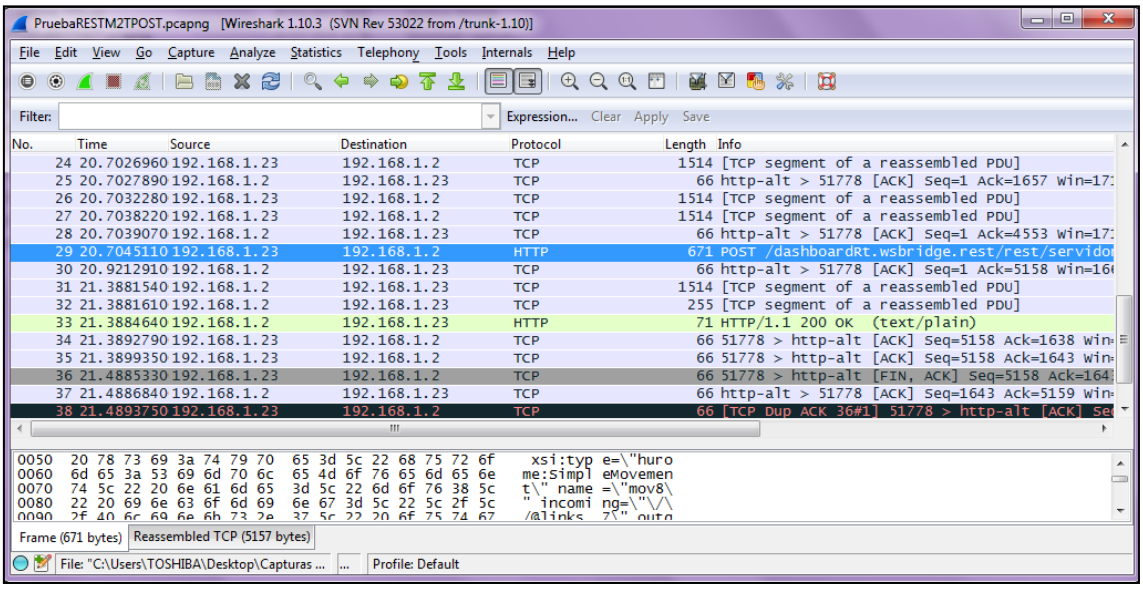

Figura 5.9.: Prueba REST POST hurome2RoboBasic – documentos en XMI

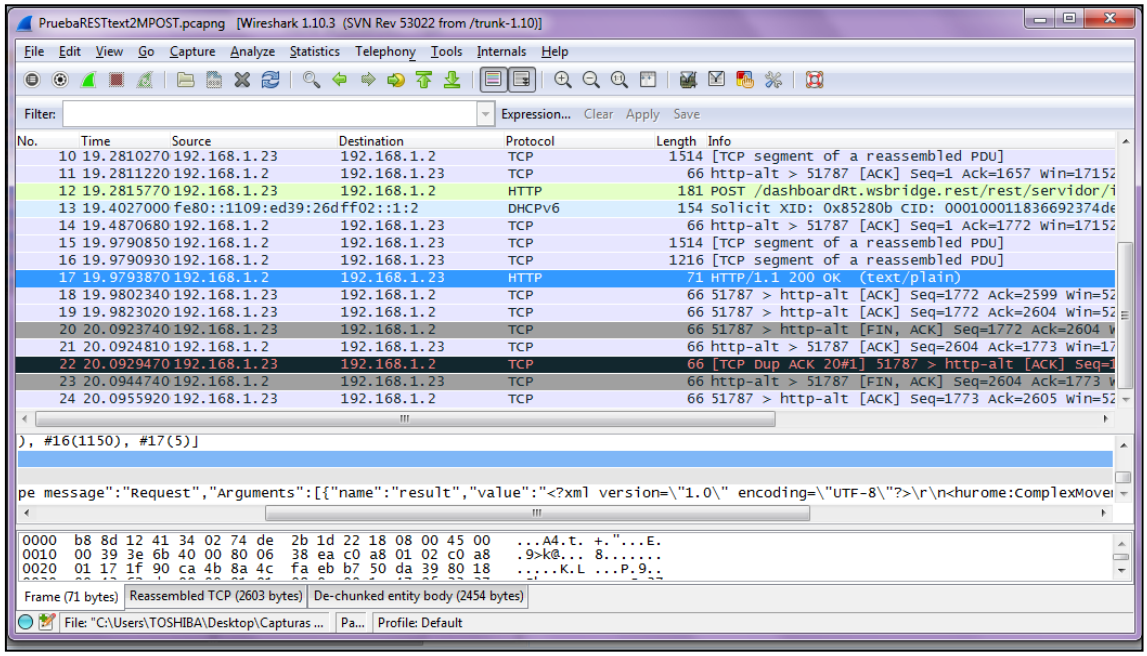

Figura 5.10.: Prueba REST POST RoboBasic2hurome – documentos en XMI

5.3.5 Prueba REST POST M2T/T2M – Documentos en binario

Para esta prueba, se ha invocado el Servicio Web REST a través de la operación POST. El formato de los mensajes que se intercambian es JSON. Los documentos aparecen codificados dentro del mensaje en formato binario.

- **Hurome2RoboBasic (M2T),** en este caso, el cliente envía el modelo hurome y el servidor le contesta con el documento roboBasic. Así pues, al analizar las tramas capturadas por WireShark, observamos que se establece un esquema de comunicación cliente-servidor, en el que el cliente envía una petición http post de tamaño 2910 bytes fragmentada en 3 paquetes y el servidor envía una respuesta http de tamaño 1642 bytes fragmentada en 3 paquetes.
- **RoboBasic2Hurome (T2M),** en este caso, el cliente envía el documento roboBasic y el servidor le contesta con el modelo hurome. Observamos en el análisis de las tramas que el cliente envía una petición http post de 1705 bytes fragmentada en 3 paquetes y recibe una respuesta http de 1106 bytes fragmentada en 2 paquetes.

Este caso representa el tipo de comunicación más eficiente entre todos los casos evaluados. Analizando el tamaño de paquete se observa que es el más ligero, debido a que el tipo de codificación de los documentos es binario (en la Tabla 5.1 podemos ver que su tamaño es menor en relación con el formato XMI) y que se utilizan mensajes JSON (más compacto que el XML).

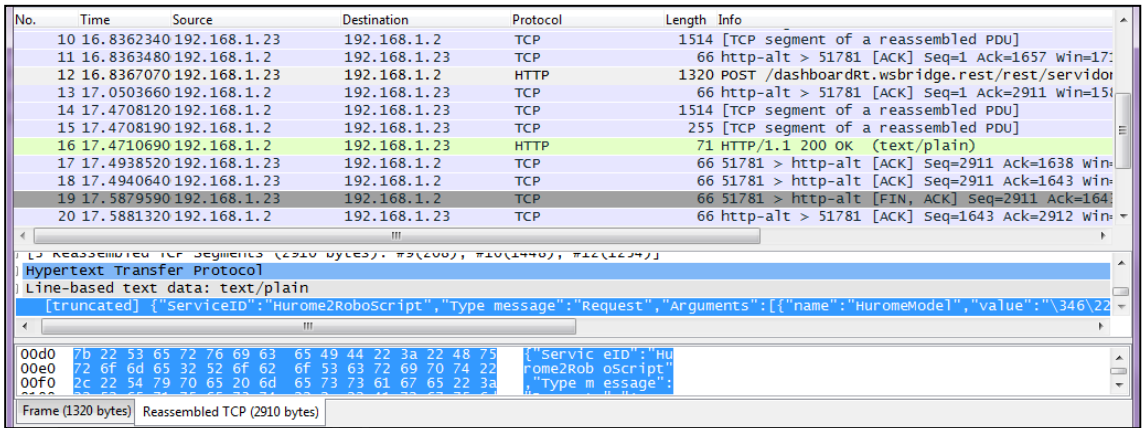

Figura 5.11.: Prueba REST POST hurome2RoboBasic – documentos en binario

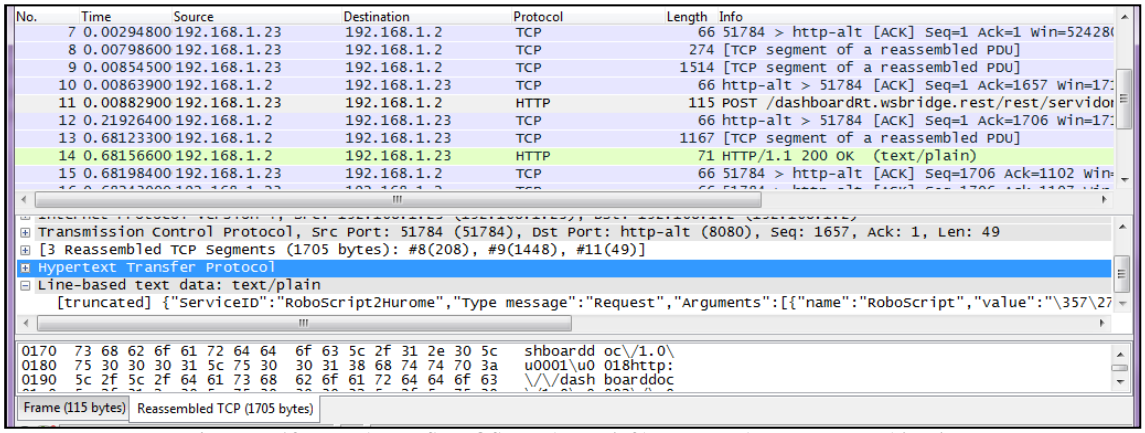

Figura 5.12.: Prueba REST POST RoboBasic2hurome – documentos en binario

5.4 Resumen de resultados

En la Tabla 5.2 se pueden observar el total de bytes intercambiados para cada una de las pruebas realizadas. Si bien en todos los casos la ejecución de las transformaciones ha sido correcta y efectiva, en términos de eficiencia de la comunicación (considerando sólo el trasiego de información), la mejor implementación ha sido REST con operación POST, formato de mensaje JSON y documentos codificados en binario. Por el contrario, la peor implementación del Servicio Web es la realizada con SOAP, con mensajes en formato XML y documentos codificados en XMI. Entre estas dos implementaciones existe una diferencia del 51,29% en el total de bytes intercambiados.

Servicio Web SOAP – Documentos en XMI

M2T	Petición: 5127 bytes (5 fragmentos)	Respuesta: 2037 bytes (4 fragmentos)
T2M	Petición: 2183 bytes (3 fragmentos)	Respuesta: 5006 bytes (6 fragmentos)

TOTAL: 14353 bytes

Servicio Web REST con GET – Documentos en XMI

M2T	Petición: 7509 bytes (6 fragmentos)	Respuesta: 1722 bytes (3 fragmentos)
T2M	Petición: 1985 bytes (2 fragmentos)	Respuesta: 2609 bytes (3 fragmentos)

TOTAL: 13825 bytes

Servicio Web REST con GET – Documentos en binario

M2T	Petición: 6132 bytes (5 fragmentos)	Respuesta: 1648 bytes (3 fragmentos)
T2M	Petición: 1847 bytes (2 fragmentos)	Respuesta: 1112 bytes (2 fragmentos)

TOTAL: 10739 bytes

Servicio Web REST con POST – Documentos en XMI

M2T	Petición: 5157 bytes (5 fragmentos)	Respuesta: 1642 bytes (3 fragmentos)
T2M	Petición: 1771 bytes (3 fragmentos)	Respuesta: 2603 bytes (3 fragmentos)

TOTAL: 11273 bytes

Servicio Web REST con POST – Documentos en binario

M2T	Petición: 2910 bytes (3 fragmentos)	Respuesta: 1642 bytes (3 fragmentos)
T2M	Petición: 1705 bytes (3 fragmentos)	Respuesta: 1106 bytes (2 fragmentos)

TOTAL: 7363 bytes

CAPÍTULO VI

Conclusiones y Líneas de Trabajos Futuras

En este capítulo se exponen las conclusiones más relevantes del Proyecto alcanzadas durante su realización. Para terminar, se introducen algunas posibles líneas de trabajo futuro.

6.1. Conclusiones

A lo largo del desarrollo de este Proyecto se han alcanzado varias conclusiones, entre las que cabe destacar las siguientes:

- ✓ El presente Proyecto proporciona una guía sobre el desarrollo de Servicios Web.
- ✓ Se ha mostrado una posible forma de desarrollar Servicios Web para distribuir las transformaciones de los entornos de modelado de Desarrollo de Software Dirigido por Modelos. En concreto, se han distribuido dos transformaciones de la herramienta HuRoME. Una transformación modelo-a-texto que permite generar desde un modelo Hurome código RoboBasic y otra inversa, texto-a-modelo, que genera un modelo Hurome desde código RoboBasic.
- ✓ Se han considerado dos formatos diferentes para serializar los modelos y documentos de texto generados o enviados como entrada a la transformación. Estos formatos han sido XMI y un formato binario, ambos soportados por el framework EMF. Dado que XMI es un lenguaje basado en XML, en general, los documentos codificados en formato binario tienen menor tamaño que los codificados en XMI.

- ✓ SOAP no permite utilizar modelos y documentos de texto serializados en binario. Ello se debe a que esta tecnología utiliza mensajes XML para establecer la comunicación y contener los datos de entrada y salida del servicio. El lenguaje XML considera inválidos una serie de caracteres especiales utilizados en el formato binario de los documentos.

- ✓ Se han implementado cinco versiones del mismo Servicio Web utilizando dos de las tecnologías más relevantes: SOAP y REST. Estas versiones han sido: SOAP considerando XMI como formato de los documentos, REST GET con documentos en XMI o binario y REST POST también con XMI o serialización binaria.

- ✓ Se han realizado una serie de pruebas para determinar la implementación más eficiente en términos de cantidad de bytes intercambiados (la más eficiente es aquella que emplea menos bytes en la comunicación). Se concluye que la implementación más eficiente es el servicio REST POST con serialización binaria de los documentos. Para la ejecución de este servicio se requiere un intercambio de bytes un 51,29% menor que el servicio implementado con SOAP.

- ✓ En el Servicio Web implementado con REST GET (a diferencia del método POST), los datos de la petición (los argumentos de entrada de la transformación) se incluyen en la URL. Ello no sólo podría tener problemas de privacidad y seguridad sino que las peticiones GET resultan ser de mayor tamaño que las peticiones POST. Se debe a que los datos se codifican en la URL añadiéndose caracteres adicionales para preceder espacios y otros caracteres especiales al contenido de los documentos que se envían en la petición.

6.2. Líneas de Trabajo Futuras

En este apartado se incluyen algunas posibles extensiones y mejoras que consideramos que sería interesante abordar de cara al futuro.

- 👍 Realizar una evaluación más exhaustiva de las distintas tecnologías utilizadas para desarrollar los Servicios Web aplicadas a las transformaciones de modelos.

- 👍 Generalizar la implementación realizada para poder ser aplicada a cualquier herramienta de Desarrollo de Software Dirigido por Modelos.

- 👍 Crear herramientas para generar de forma automática el código del Servicio Web dada una transformación concreta.

- 👍 Investigar alternativas a los Servicios Web que permitan distribuir las transformaciones de un entorno de modelado.

Referencias

[1] T. Stahl, M. Voelter, and K. Czarnecki, *Model-Driven Software Development: Technology, Engineering*, Management, ed. Wiley, 2006, ISBN: 978-0-470-02570-3.

[2] J.F. Inglés-Romero, C. Vicente-Chicote, D. Alonso, *HuRoME: Entorno para Modelado de Coreografías y Modernización de Código para un Robot Humanoide*, XV Jornadas de Ingeniería del Software y Bases de Datos (JISBD'10), Valencia (Spain), September 2010. ISBN 978-84-92812-51-6.

[3] J.F. Inglés-Romero*: Humanoid Robot Modeling Environment (HuRoME)*, Trabajo Final de Máster, Máster en Tecnologías de la Información y Comunicaciones, Universidad Politécnica de Cartagena, September 2010.

[4] Robonova 1, HITEC Robotics, http://www.robonova.com/

[5] World Wide Web, http://www.w3.org/

[6] Open Network Computing Remote Procedure Call, http://tools.ietf.org/html/rfc5531

[7] Luke Kenneth Casson Leighton (1999). *DCE/RPC over SMB: Samba and Windows NT Domain Internals*. Sams. ISBN 1-57870-150-3

[8] Distributed Component Object Model, http://msdn.microsoft.com/library/cc201989.aspx

[9] Simon St. Laurent, Joe Johnston, Edd Dumbill. (June 2001) *Programming Web Services with XML-RPC* O'Reilly. First Edition. Foreword by Dave Winer.

[10] Roy Thomas Fielding, http://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm

[11] SOAP, http://www.w3.org/TR/soap/

[12] Rest VS SOAP, Rafael Navarro Marset. ELP-DSIC-UPV Modelado, Diseño e Implementación de Servicios Web 2006-07

[13] Web Services Description Language, http://www.w3.org/TR/wsdl

[14] Universal Description, Discovery and Integration, https://www.oasis-open.org/committees/uddi-spec/doc/tcspecs.htm

[15] Java API for XML Web Services, https://jax-ws.java.net/nonav/jaxws-api/2.2/index.html

[16] GlassFish, https://glassfish.java.net/

[17] Metro, https://metro.java.net/

[18] JAX-RS spec., https://jax-rs-spec.java.net/

[19] Jersey, https://jersey.java.net/

[20] JAXB, https://jaxb.java.net/

[21] Booch, G., Rumbaugh, J. and Jacobson, I.: Unified Modeling Language User Guide. 2nd Edition. Addison-Wesley, 2005.

[22] Bézivin, J. and Gerbé, O.: "Towards a Precise Definition of the OMG/MDA Framework." En Proceedings of 16th IEEE International Conference on Automated Software Engineering (ASE'01), Noviembre 2001.

[23] Mellor, S.J., Scott, K., Uhl, A., Weise, D.: MDA Distilled. Principles of Model-Driven Architecture. Addison-Wesley, 2004.

[24] OMG. "Model Driven Architecture." Document ormsc/2001-07-01, Julio 2001. Accesible desde http://www.omg.org/

[25] DSL Tools. http://msdn.microsoft.com/en-us/library/bb126235.aspx

[26] Meta Edit. http://www.metacase.com/

[27] The Eclipse Graphical Modeling Framework (GMF), http://www.eclipse.org/modeling/gmf/

[28] JET, http://www.eclipse.org/modeling/m2t/?project=jet#jet

[29] RoboBASIC Command Instruction Manual v2.10, http://www.hitecrobotics.com

[30] Xtext, http://www.eclipse.org/Xtext/

[31] ATLAS Transformation Language (ATL), http://www.eclipse.org/atl/

[32] Eclipse IDE for Java Developers, http://www.eclipse.org

[33] Apache CXF, http://cxf.apache.org/

[34] Tomcat, http://tomcat.apache.org/download-70.cgi

[35] D.J. Pérez-Sánchez, J.F. Inglés-Romero, C. Vicente-Chicote, *Generación Automática de Vistas de Control para Herramientas de Desarrollo de Software Dirigido por Modelos en Eclipse,* XVIII Jornadas en Ingeniería del Software y Bases de Datos (JISBD '13), Madrid (Spain), 17-20 September 2013

[36] Eclipse Modeling Framework, https://www.eclipse.org/modeling/emf/

Anexo I. Código del Servicio Web SOAP

Este anexo muestra el código desarrollado para la implementación del Servicio Web SOAP.

A1.1 Clase ServiceDelegator

```
package dashboardRt.wsbridge.soap;

import dashboardRt.wsbridge.soap.utils.ServiceUtil;
import java.io.InputStream;
import java.io.OutputStream;
import java.util.ArrayList;
import java.util.Map;
import javax.xml.bind.JAXBException;
import javax.xml.namespace.QName;
import javax.xml.transform.Source;
import javax.xml.transform.stream.StreamSource;
import javax.xml.ws.Dispatch;
import javax.xml.ws.Service;
import javax.xml.ws.soap.SOAPBinding;
import dashboardRt.wsbridge.soap.messages.Arguments;
import dashboardRt.wsbridge.soap.messages.Message;
import dashboardRt.wsbridge.soap.messages.Message.typeMessage;

/**
 *
 * @author Fermin
 * @category Clase que configura el mensaje, lo serializa, envía y obtiene la respuesta
 *
 */
public class ServiceDelegator {

        private static final QName SERVICE_NAME = new QName("http://dashboardRt.wsbridge.soap/",
"Dashboard");
        private static final QName PORT_NAME     = new QName("http://dashboardRt.wsbridge.soap/",
"DashboardPort");

        StreamSource stream,stream2;
        OutputStream os;
        InputStream bytes;
        ArrayList<Arguments> outputsArray = new ArrayList<Arguments>();


        public void invokeService(String url, String service,
                        Map<String, Object> inputs,
                        Map<String, Object> outputs) throws JAXBException
        {


                Source request = null;
                Source response = null;

                // (1) C–digo de inicializaci–n para crear un objeto Dispatch<Source> que se
                // utiliza para realizar la petici–n de servicio

                Service srv = Service.create(SERVICE_NAME);
                srv.addPort(PORT_NAME, SOAPBinding.SOAP11HTTP_BINDING, url);
                Dispatch<Source> sourceDispatch = srv.createDispatch(PORT_NAME, Source.class,
Service.Mode.PAYLOAD);


                // (2) Crear del mensaje a partir de la informaci–n pasada como argumento
                // vemos si inputs lleva info
                //Debemos recorrer el mapa input y por cada entrada del mapa creo un Arguments
con name y value (Key y valor!!!) y lo meto al array

                Message msg = new Message();
                msg.setServiceId(service);
```

```
                msg.setTypeMessage(typeMessage.Request);
                ServiceUtil.Argument2Message(inputs, msg);

                // (3) Serializar el mensaje para obtener el correspondiente c—digo XML de la
petici—n

                request =  ServiceUtil.serializeMessage(msg);

                // (4) Env'ar la petici—n de servicio

                response = sourceDispatch.invoke(request);

                // (5) Deserializar la respuesta XML para obtener el mensaje como objeto Java

                msg = ServiceUtil.deserializeMessage(response);


                // (6) Rellenar el mapa outputs con la informaci—n contenida en el mensaje
                ServiceUtil.message2Argument(msg, outputs);
                outputs.put("Id del servicio en string", msg.getServiceId());
        }
}
```

A1.2 Clase ServiceProvider

```
package dashboardRt.wsbridge.soap;

import java.util.ArrayList;
import java.util.HashMap;
import java.util.Iterator;
import java.util.Map;
import javax.xml.bind.JAXBException;
import javax.xml.transform.Source;
import javax.xml.ws.BindingType;
import javax.xml.ws.Provider;
import javax.xml.ws.ServiceMode;
import javax.xml.ws.WebServiceProvider;
import dashboardRt.wsbridge.soap.messages.Message;
import dashboardRt.wsbridge.soap.messages.Arguments;
import dashboardRt.wsbridge.soap.utils.ServiceUtil;

/**
 *
 * @author Fermin
 * @category Clase que recibe el mensaje, obtiene la ejectuta el servicio y envia la respuesta
 */
@WebServiceProvider(portName = "DashboardPort",
        serviceName = "Dashboard",
        targetNamespace = "http://example.com/request/wsdl")
@BindingType(value = "http://schemas.xmlsoap.org/wsdl/soap/http")
@ServiceMode(value = javax.xml.ws.Service.Mode.PAYLOAD)
public class ServiceProvider implements Provider<Source> {

        Message ms;
        Map<String, Object> inputs,outputs;
        Source d = null;
        ArrayList<Arguments> inputsArray = new ArrayList<Arguments>();

        /*
         * El servidor llamar‡ a este mŽtodo cada vez que se recibe una petici—n de servicio.
         * Se pasa un objeto request que contine el cuerpo de la petici—n SOAP, es decir, el
```

```
         * mensaje XML que el cliente ha enviado. El mŽtodo debe devolver el resultado del
         * servicio como otro mensaje XML.
         */

        @Override
        public Source invoke(Source request) {

                // (1) Deserializar la petici—n XML para obtener el mensaje como objeto Java

                try {

                        ms = ServiceUtil.deserializeMessage(request);

                } catch (JAXBException e) {
                        System.out.println("Se ha producido excepción en deserializar:" +
e.toString());
                        e.printStackTrace();

                }
                // (2) Crear los mapas inputs y outputs para llamar a la funci—n invokeService,
                // utilizando para ello la informaci—n contenida en el mensaje

                inputs = new HashMap<String, Object>();
                outputs = new HashMap<String, Object>();

                // (3) Llamar a la funci—n invokeService

                invokeService(ms.getServiceId(),inputs,outputs);

                // (4) Crear el mensaje que ha de enviarse en respuesta del servicio

                Message n = new Message();
                n.setServiceId(ms.getServiceId());
                n.setTypeMessage(Message.typeMessage.response);
                // (5) Serializar el mensaje para obtener la correspondiente respuesta XML, esta
                // respuesta se retorna como un objeto Source

                try {

                        d = ServiceUtil.serializeMessage(n);

                } catch (JAXBException e) {

                        e.printStackTrace();
                }

                return d;

        }

        /*
         * Este mŽtodo invocar‡ el servicio en el componente (toolbox) correspondiente
         */
        public void invokeService(String service,
                        Map<String, Object> inputs,
                        Map<String, Object> outputs)
        {

                // La conexi—n con el componente que implementa el servicio no se tendr‡ en
cuenta
                // al principio. As’, para realizar pruebas utilizaremos la siguiente
implementaci—n
                System.out.println("Ejecutando servicio con los siguientes inputs:");

                Iterator<String> keyIterator = inputs.keySet().iterator();
                String key;
                while(keyIterator.hasNext()) {
                        key = keyIterator.next();
                        System.out.println("\t[Name:" + key + "; value:" + inputs.get(key) +
"]");
                }

                outputs.put("result", new Boolean("true"));
```

```
        }

}
```

A1.3 Clase Arguments

```
package dashboardRt.wsbridge.soap.messages;

import dashboardRt.wsbridge.soap.utils.DataTypeEnum;
/**
 * @author Fermin
 *
 * @category Clase para establecer datos
 */
public class Arguments {

        private String name;
        private String value;
        private DataTypeEnum type;

        /**
         *
         * @param name, establece el nombre del argumento
         */
        public void setName(String name){

                this.name = name;
        }

        /**
         *
         * @param value, establece el valor del argumento
         */
        public void setValue(String value){

                this.value = value;
        }

        public String getName(){

                return name;
        }

        public String getValue(){

                return value;
        }

        /**
         *
         * @param type, indica de que tipo de dato se trata, INTEGER,DOUBLE,BOOLEAN,MODEL,TEXT ó
STATUS
         */
        public void setDataTypeEnum(DataTypeEnum type){

                this.type=type;
        }

        public DataTypeEnum getDataTypeEnum(){

                return type;
        }



}
```

A1.4 Clase Message

```
package dashboardRt.wsbridge.soap.messages;

import java.util.ArrayList;
import javax.xml.bind.annotation.XmlElement;
import javax.xml.bind.annotation.XmlElementWrapper;
import javax.xml.bind.annotation.XmlRootElement;

// Esta clase representa el mensaje que el cliente env’a y recibe como respuesta
// una vez ejecutado el servicio web. Este mensaje deber‡ contener el identificador del
// servicio que deseamos ejecutar en el componente, los inputs y los outputs (el mensaje
// no tendr‡ outputs cuando corresponda con una petici—n de servicio, por lo
// contrario, s’ aparecer‡n Žstos cuando el mensaje sea creado para transmitir el resultado
// del servicio web). Como este mensaje debe ser serializable a c—digo XML debe utilizarse
// la API JAXB
/**
 *
 * @author FERMIN
 *
 * @category La clase Message contiene un conjunto de Arguments que forman el contenido del
paquete XML
 *
 */
@XmlRootElement(name = "message")
public class Message {

        private String serviceId;
        private static ArrayList<Arguments> argu = new ArrayList<Arguments>();
    public enum typeMessage{Request, response}
        public typeMessage type;

        public String getServiceId() {
                return serviceId;
        }

        public typeMessage getTypeMessage(){

                return type;
        }
        /**
         *
         * @param type, establece el valor de la etiqueta en el XML generado, (type)
         */
        @XmlElement(name = "type")
        public void setTypeMessage(typeMessage type){

                this.type=type;
        }
        /**
         *
         * @param serviceId, establece el valor de la etiqueta en el XML generado (idString)
         */
        @XmlElement(name = "idString")
        public void setServiceId(String serviceId) {
                this.serviceId = serviceId;
        }
        /**
         *
         * @param argu, mediante XmlElementWrapper cambiamos el valor a la etiqueta XML por el
de data
         */
        @XmlElementWrapper(name="data")
        @XmlElement(name = "parameter")
        public void setArguments(ArrayList<Arguments> argu){

                this.argu = argu;
        }

        public ArrayList<Arguments> getArguments() {
                return argu;
        }
```

```
}
```

A1.5 Clase Client

```
package dashboardRt.wsbridge.soap.tests;

import java.io.FileNotFoundException;
import java.util.HashMap;
import java.util.Iterator;
import javax.xml.bind.JAXBException;
import dashboard.runtime.MetamodelsStore;
import dashboard.runtime.dashboarddoc.DashboarddocFactory;
import dashboard.runtime.dashboarddoc.ModelDoc;
import dashboard.runtime.dashboarddoc.TextDoc;
import dashboard.runtime.diagnosis.SerializableStatus;
import dashboard.runtime.diagnosis.StatusSeverityEnum;
import dashboardRt.wsbridge.soap.ServiceDelegator;

/**
 *
 * @author Fermin
 * @category Client rellena las entradas con los datos e invoca el servicio
 */
public class Client {

	/*
	 * Direcci—n del servicio
	 */
	public final static String ENDPOINT_ADDR = "http://localhost:9000/dashboard";


	public static void main(String[] args) throws JAXBException, FileNotFoundException {

		ServiceDelegator delegator = new ServiceDelegator();

		// Creaci—n de los mapas vacios
		HashMap<String, Object> inputs = new HashMap<String, Object>();
		HashMap<String, Object> outputs = new HashMap<String, Object>();

		//Registramos los modelos para poder utilizarlos
		MetamodelsStore mmstore = MetamodelsStore.getInstance();
		mmstore.initialize("../dashboard.runtime/metamodels");

		//Vamos con un tipo STATUS
		SerializableStatus status = new SerializableStatus();
		status.setSeverity(StatusSeverityEnum.OK);


		//Vamos con un tipo MODEL
		DashboarddocFactory factory = DashboarddocFactory.eINSTANCE;
		ModelDoc model = factory.createModelDoc();
		model.load("../dashboard.runtime/models/huromeModel.xmi");

		//Vamos con un tipo TEXT

		TextDoc text = factory.createTextDoc();
		text.load("../dashboard.runtime/models/TextDoc.xmi");


		// Creaci—n de las entradas en el mapa inputs
		inputs.put("parametro_ejemplo_1", new Integer(23));
		inputs.put("parametro_ejemplo_2", new Double(10.5));
		inputs.put("parametro_ejemplo_3", new Boolean(true));
		inputs.put("parametro_ejemplo_4", new Boolean(false));
		inputs.put("Ejemplo STATUS", status);
		inputs.put("Ejemplo MODEL", model);
		inputs.put("Ejemplo TEXT", text);
```

```
                // Invocaci—n del servicio
                delegator.invokeService(ENDPOINT_ADDR, "servicio_ejemplo", inputs, outputs);

                // Recorremos el mapa outputs para mostrar su contenido
                Iterator<String> keyIterator = outputs.keySet().iterator();
                System.out.println("Estamos recorriendo outputs para mostrarlo:");
                String key;
                while(keyIterator.hasNext()) {
                        key = keyIterator.next();
                        System.out.println("\t[Name:" + key + "; value:" + outputs.get(key) +
"]");
                }
        }
}
```

A1.6 Clase Server

```
package dashboardRt.wsbridge.soap.tests;

import java.io.FileNotFoundException;
import javax.xml.bind.JAXBException;
import javax.xml.ws.Endpoint;
import dashboardRt.wsbridge.soap.ServiceProvider;


public class Server {

        /*
         * Direcci—n del servicio
         */
        public final String ENDPOINT_ADDR = "http://localhost:9000/dashboard";

        protected Server() {

                System.out.println("Starting Server");

                // Publicaci—n del servicio web
                ServiceProvider implementor = new ServiceProvider();
                Endpoint.publish(ENDPOINT_ADDR, implementor);
        }

        public static void main(String[] args) throws JAXBException, FileNotFoundException,
InterruptedException {

                new Server();
        System.out.println("Server ready...");

        // El servidor permanece activo durante un periodo de 5 min
        Thread.sleep(5 * 60 * 1000);

        System.out.println("Server exiting");
        System.exit(0);
        }
}
```

A1.7 Clase DataTypeEnum

```
package dashboardRt.wsbridge.soap.utils;

/**
 *
 * @author Fermin
 * @category Enumerador para identificar el tipo de dato
 *
 */
public enum DataTypeEnum {
        INTEGER,DOUBLE,BOOLEAN,MODEL,TEXT,STATUS; }
```

A1.8 Clase ServiceUtil

```
package dashboardRt.wsbridge.soap.utils;

import java.io.ByteArrayOutputStream;
import java.io.InputStream;
import java.io.StringReader;
import java.util.ArrayList;
import java.util.Map;
import javax.xml.bind.JAXBContext;
import javax.xml.bind.JAXBException;
import javax.xml.bind.Marshaller;
import javax.xml.bind.Unmarshaller;
import javax.xml.transform.Source;
import javax.xml.transform.stream.StreamSource;
import dashboard.runtime.dashboarddoc.DashboarddocFactory;
import dashboard.runtime.dashboarddoc.ModelDoc;
import dashboard.runtime.dashboarddoc.TextDoc;
import dashboard.runtime.dashboarddoc.impl.ModelDocImpl;
import dashboard.runtime.dashboarddoc.impl.TextDocImpl;
import dashboard.runtime.diagnosis.SerializableStatus;
import dashboard.runtime.diagnosis.StatusSeverityEnum;
import dashboardRt.wsbridge.soap.messages.Message;
import dashboardRt.wsbridge.soap.messages.Arguments;

/**
 *
 * @author Fermin
 * @category Clase que proporciona métodos para la conversión entre formatos, Para ver
descripciones de métodos y la clase en su totalidad ver el util de REST, esta contiene los
métodos de aqui mas los especificos de REST
 */
public class ServiceUtil {

        public static ByteArrayOutputStream os,osShow;
        public static StreamSource stream,stream2;
        public static InputStream bytes;

        // Esta clase puede utilizarse para implementar algunos mŽtodos comunes que se
        // utilicen en el c—digo del cliente y en el servidor, por ejemplo, mŽtodos
        // para serializar y deserializar los mensajes:

        public static Source serializeMessage(Message msg) throws JAXBException{


                 JAXBContext context = JAXBContext.newInstance(Message.class);
                 Marshaller m = context.createMarshaller();
                 m.setProperty(Marshaller.JAXB_FORMATTED_OUTPUT, Boolean.TRUE);
                 os = new ByteArrayOutputStream();
                 m.marshal(msg, os);
                 stream = new StreamSource(new StringReader(os.toString()));
                 return stream;
        }

        public static Message deserializeMessage(Source request) throws JAXBException{

                JAXBContext context = JAXBContext.newInstance(Message.class);
                Unmarshaller u = context.createUnmarshaller();
                Message msg2 = (Message) u.unmarshal(request);
                return msg2;

        }

        public static void Argument2Message(Map<String, Object> map, Message msg){

                Object[] keys = map.keySet().toArray();
            Object[] values = map.values().toArray();
            Arguments in;
                ArrayList<Arguments> array = new ArrayList<Arguments>();

            int i=0;

            while(i<map.size()){
```

```
                if(values[i].getClass()==Integer.class){

                        in = new Arguments();
                        in.setName(keys[i].toString());
                        in.setValue(values[i].toString());
                        in.setDataTypeEnum(DataTypeEnum.INTEGER);
                        array.add(in);


                }else if(values[i].getClass()==Double.class){

                        in = new Arguments();
                        in.setName(keys[i].toString());
                        in.setValue(values[i].toString());
                        in.setDataTypeEnum(DataTypeEnum.DOUBLE);
                        array.add(in);

                }else if(values[i].getClass()==Boolean.class){

                        in = new Arguments();
                        in.setName(keys[i].toString());
                        in.setValue(values[i].toString());
                        in.setDataTypeEnum(DataTypeEnum.BOOLEAN);
                        array.add(in);

                }else if(values[i].getClass()==SerializableStatus.class){


                        in = new Arguments();
                        in.setName(keys[i].toString());
                        SerializableStatus status = (SerializableStatus) values[i];
                        String statusValue = status.serialize();
                        in.setValue(statusValue);
                        in.setDataTypeEnum(DataTypeEnum.STATUS);
                        array.add(in);

                }else if(values[i].getClass()==ModelDocImpl.class){

                        in = new Arguments();
                        in.setName(keys[i].toString());
                        ModelDoc model = (ModelDoc) values[i];
                        String modelValue = model.serialize();
                        in.setValue(modelValue);
                        in.setDataTypeEnum(DataTypeEnum.MODEL);
                        array.add(in);

                }else if(values[i].getClass()==TextDocImpl.class){

                        in = new Arguments();
                        in.setName(keys[i].toString());
                        TextDoc text = (TextDoc) values[i];
                        String textValue = text.serialize();
                        in.setValue(textValue);
                        in.setDataTypeEnum(DataTypeEnum.TEXT);
                        array.add(in);

                }else{

                        System.out.println("No es ningun tipo de los establecidos" +
keys[i].toString() + "\n" +values[i].getClass());
                }

                i++;
            }

            msg.setArguments(array);


        }

        public static void showXML(Message msg) throws JAXBException{
```

```
                JAXBContext context = JAXBContext.newInstance(Message.class);
                 Marshaller m = context.createMarshaller();
                 m.setProperty(Marshaller.JAXB_FORMATTED_OUTPUT, Boolean.TRUE);
                 osShow = new ByteArrayOutputStream();
                 m.marshal(msg, osShow);
                 System.out.println("Mostramos mensaje serializado a XML : " +
osShow.toString());

        }

        public static void message2Argument(Message msg, Map<String, Object> map){

                ArrayList<Arguments> mapsArray = msg.getArguments();

                for(int i=0; i<mapsArray.size();i++){

                        Arguments m = mapsArray.get(i);

                        if(m.getDataTypeEnum().toString()=="DOUBLE"){

                                map.put(m.getName(),Double.parseDouble(m.getValue()));

                        }else if(m.getDataTypeEnum().toString()=="INTEGER"){

                                map.put(m.getName(), Integer.parseInt(m.getValue()));

                        }else if(m.getDataTypeEnum().toString()=="BOOLEAN"){


                                map.put(m.getName(), Boolean.parseBoolean(m.getValue()));

                        }else if(m.getDataTypeEnum().toString()=="STATUS"){

                                SerializableStatus status2 = new SerializableStatus();
                                status2.deserialize(m.getValue());
                                map.put(m.getName(), status2);

                        }else if(m.getDataTypeEnum().toString()=="MODEL"){

                                DashboarddocFactory factory = DashboarddocFactory.eINSTANCE;
                                ModelDoc model = factory.createModelDoc();
                                model.deserialize(m.getValue());
                                map.put(m.getName(), model);

                        }else if(m.getDataTypeEnum().toString()=="TEXT"){

                                DashboarddocFactory factory = DashboarddocFactory.eINSTANCE;
                                TextDoc text2 = factory.createTextDoc();
                                text2.deserialize(m.getValue());
                                map.put(m.getName(),text2);

                        }else{

                                System.out.println("ERROR, tipo de dato no contemplado" +
m.getValue().toString() +"\n" + m.getName());
                        }


                }

        }

}
```

Anexo II. Código del Servicio Web RESTful

Este Anexo muestra el código utilizado para desarrollar el Servicio Web REST.

A2.1 Clase ServiceDelegator

```
package dashboardRt.wsbridge.rest;

import dashboardRt.wsbridge.rest.utils.ServiceUtil;
import java.util.Map;
import javax.xml.bind.JAXBException;
import java.net.URI;
import javax.ws.rs.core.MediaType;
import javax.ws.rs.core.UriBuilder;
import org.codehaus.jettison.json.JSONException;
import org.codehaus.jettison.json.JSONObject;
import com.sun.jersey.api.client.Client;
import com.sun.jersey.api.client.WebResource;
import com.sun.jersey.api.client.config.ClientConfig;
import com.sun.jersey.api.client.config.DefaultClientConfig;
import dashboardRt.wsbridge.rest.messages.Message;
import dashboardRt.wsbridge.rest.messages.Message.typeMessage;


public class ServiceDelegator {


        public void invokeService(String url, String service,
                        Map<String, Object> inputs,
                        Map<String, Object> outputs) throws JAXBException, JSONException
        {
                /**
                 * Creamos el Servicio Web
                 */
                ClientConfig config = new DefaultClientConfig();
            Client client = Client.create(config);
            WebResource serviceClient = client.resource(getBaseURI());

            /**
             * Creación del mensaje con los datos
             */
                Message msg = new Message();
                Message msg2;
                msg.setServiceId(service);
                msg.setTypeMessage(typeMessage.Request);
                ServiceUtil.Argument2Message(inputs, msg);

                /**
                 * Una de las diferencias de este Servicio Web respecto del SOAP es el tipo de
mensaje que usamos para el intercambio, en este caso es JSON
                 */
                JSONObject json = ServiceUtil.Message2JSON(msg);



                /**
                 * Sistema de realizar la petición, accedemos a través del Path (Podemos
visualiarlo también en el navegador), y en la Query (petición) enviamos el archivo JSON
                 */
                String response =
serviceClient.path("rest").path("servidor").path("invoke").queryParam("request",
json.toString()).accept(MediaType.APPLICATION_JSON).get(String.class);

                /**
                 * Extraemos resultados del mensaje JSON que nos proporciona el Servicio Web
                 */
                JSONObject o = new JSONObject(response);
                msg2 = ServiceUtil.JSON2Message(o);
```

```
                ServiceUtil.message2Argument(msg2, outputs);

                outputs.put("Id del servicio en string", msg.getServiceId());

        }

        private URI getBaseURI() {
                return
UriBuilder.fromUri("http://localhost:8080/dashboardRt.wsbridge.rest").build();
        }

}
```

A2.2 Clase ServiceProvider

```
package dashboardRt.wsbridge.rest;

import java.util.HashMap;
import java.util.Iterator;
import java.util.Map;
import javax.ws.rs.GET;
import javax.ws.rs.Path;
import javax.ws.rs.QueryParam;
import javax.ws.rs.core.MediaType;
import javax.ws.rs.Produces;
import javax.xml.bind.JAXBException;
import org.codehaus.jettison.json.JSONException;
import org.codehaus.jettison.json.JSONObject;
import dashboardRt.wsbridge.rest.messages.Message;
import dashboardRt.wsbridge.rest.utils.ServiceUtil;

/**
 *
 * @author Fermin
 * @category Clase principal del Servicio Web, mediante el formateo con anotaciones se
especifica que
 * y a través de donde obtenemos el Servicio. Servidor es parte de la ruta que hay que
introducir para obtener la respuesta
 */
@Path("/servidor")
public class ServiceProvider  {

        JSONObject back;
        Map<String, Object> inputs,outputs;
        Message msg,ms;

        /**
         * @category invoke es parte de la ruta de acceso al metodo, GET el tipo de operacion
HTTP resultante, produces es lo que obtenemos (que en el caso que nos ocupa es la repuesta en
JSON)
         * , y QueryParam quiere decir que los datos para que se ejecute la operación son
pasados a través del URL, Formateo de paquete JSON
         * @param request
         *
         */
        @Path("invoke")
        @GET
        @Produces(MediaType.APPLICATION_JSON)
        public String invoke(@QueryParam("request") String request) throws JAXBException,
JSONException {

                JSONObject json = new JSONObject(request);
                //Crear los mapas a partir de la info contenida en el mensaje
                msg = ServiceUtil.JSON2Message(json);

                inputs = new HashMap<String, Object>();
                outputs = new HashMap<String, Object>();
                invokeService(msg.getServiceId(),inputs,outputs);
                ms = new Message();
                ms.setServiceId(msg.getServiceId());
                ms.setTypeMessage(Message.typeMessage.response);
```

```
                back = ServiceUtil.Message2JSON(ms);
                return back.toString();

        }

/**
         * Metodo que se invoca a partir de una petición POST
         * @param request
         * @return Respuesta de Servicio Web método POST en formato JSON
         * @throws IOException
         * @throws JSONException
         */

        @Path("invoke")
        @POST
        @Produces(MediaType.TEXT_PLAIN)
        @Consumes(MediaType.TEXT_PLAIN)
        public String metodoPOST(String request) throws IOException, JSONException {

                json = new JSONObject(request);
                //Crear los mapas a partir de la info contenida en el mensaje
                System.out.println("El JSON DE POST TIENE"+request);

                msg = ServiceUtil.JSON2Message(json);

                inputs = new HashMap<String, Object>();
                outputs = new HashMap<String, Object>();
                inputs= ServiceUtil.JSON2Map(json);

                invokeService(msg.getServiceId(),inputs,outputs);
                ms = new Message();
                ServiceUtil.Argument2Message(outputs, ms);
                ms.setServiceId(msg.getServiceId());
                ms.setTypeMessage(Message.typeMessage.response);

                back = ServiceUtil.Message2JSON(msg);
                return back.toString();



          }

        /*
         * Este mŽtodo invocar‡ el servicio en el componente (toolbox) correspondiente
         */
        public void invokeService(String service,
                        Map<String, Object> inputs,
                        Map<String, Object> outputs)
        {

                // La conexi—n con el componente que implementa el servicio no se tendr‡ en
cuenta
                // al principio. As’, para realizar pruebas utilizaremos la siguiente
implementaci—n
                System.out.println("Ejecutando servicio con los siguientes inputs:");

                Iterator<String> keyIterator = inputs.keySet().iterator();
                String key;
                while(keyIterator.hasNext()) {
                        key = keyIterator.next();
                        System.out.println("\t[Name:" + key + "; value:" + inputs.get(key) +
"]");
                }

                outputs.put("result", new Boolean("true"));
        }

}
```

A2.3 Clase Arguments

```
package dashboardRt.wsbridge.rest.messages;
```

```
import dashboardRt.wsbridge.rest.utils.DataTypeEnum;

public class Arguments {

        private String name;
        private String value;
        private DataTypeEnum type;

        public void setName(String name){

                this.name = name;
        }

        public void setValue(String value){

                this.value = value;
        }

        public String getName(){

                return name;
        }

        public String getValue(){

                return value;
        }

        public void setDataTypeEnum(DataTypeEnum type){

                this.type=type;
        }

        public DataTypeEnum getDataTypeEnum(){

                return type;
        }



}
```

A2.4 Clase Message

```
package dashboardRt.wsbridge.rest.messages;

import java.util.ArrayList;
import javax.xml.bind.JAXBException;
import javax.xml.bind.annotation.XmlElement;
import javax.xml.bind.annotation.XmlElementWrapper;
import javax.xml.bind.annotation.XmlRootElement;
import dashboardRt.wsbridge.rest.utils.ServiceUtil;

// Esta clase representa el mensaje que el cliente env’a y recibe como respuesta
// una vez ejecutado el servicio web. Este mensaje deber‡ contener el identificador del
// servicio que deseamos ejecutar en el componente, los inputs y los outputs (el mensaje
// no tendr‡ outputs cuando corresponda con una petici—n de servicio, por lo
// contrario, s’ aparecer‡n Žstos cuando el mensaje sea creado para transmitir el resultado
// del servicio web). Como este mensaje debe ser serializable a c—digo XML debe utilizarse
// la API JAXB

@XmlRootElement(name = "message")
public class Message {

        private String serviceId;
        private static ArrayList<Arguments> argu = new ArrayList<Arguments>();
    public enum typeMessage{Request, response}
        public typeMessage type;

        public String getServiceId() {
```

```
		return serviceId;
	}

	public typeMessage getTypeMessage(){

		return type;
	}

	@XmlElement(name = "type")
	public void setTypeMessage(typeMessage type){

		this.type=type;
	}


	@XmlElement(name = "idString")
	public void setServiceId(String serviceId) {
		this.serviceId = serviceId;
	}

	@XmlElementWrapper(name="data")
	@XmlElement(name = "parameter")
	public void setArguments(ArrayList<Arguments> argu){

		this.argu = argu;
	}

	public ArrayList<Arguments> getArguments() {
		return argu;
	}



	public Message valueOf(String src) {
		Message result = null;
		try {
			result = ServiceUtil.deserializeMessage(src);
		} catch (JAXBException e) {
			// TODO Auto-generated catch block
			e.printStackTrace();
		}
		return result;
	}

}
```

A2.5 Clase Client

```
package dashboardRt.wsbridge.rest.tests;

import java.io.FileNotFoundException;
import java.util.HashMap;
import java.util.Iterator;
import javax.xml.bind.JAXBException;
import org.codehaus.jettison.json.JSONException;
import dashboard.runtime.MetamodelsStore;
import dashboard.runtime.dashboarddoc.DashboarddocFactory;
import dashboard.runtime.dashboarddoc.ModelDoc;
import dashboard.runtime.dashboarddoc.TextDoc;
import dashboard.runtime.diagnosis.SerializableStatus;
import dashboard.runtime.diagnosis.StatusSeverityEnum;
import dashboardRt.wsbridge.rest.ServiceDelegator;


public class Client {

	/*
	 * Direcci—n del servicio
	 */
	public final static String ENDPOINT_ADDR = "http://localhost:9000/dashboard";
```

```
	public static void main(String[] args) throws JAXBException, FileNotFoundException,
JSONException {

		ServiceDelegator delegator = new ServiceDelegator();

		// Creaci—n de los mapas vacios
		HashMap<String, Object> inputs = new HashMap<String, Object>();
		HashMap<String, Object> outputs = new HashMap<String, Object>();

		//Registramos los modelos para poder utilizarlos
		MetamodelsStore mmstore = MetamodelsStore.getInstance();

	mmstore.initialize("/C:/Users/TOSHIBA/ProjectRest/dashboard.runtime/metamodels");

		//Vamos con un tipo STATUS
		SerializableStatus status = new SerializableStatus();
		status.setSeverity(StatusSeverityEnum.OK);


		//Vamos con un tipo MODEL
		DashboarddocFactory factory = DashboarddocFactory.eINSTANCE;
		ModelDoc model = factory.createModelDoc();

	model.load("/C:/Users/TOSHIBA/ProjectRest/dashboard.runtime/models/huromeModel.xmi");

		//Vamos con un tipo TEXT

		TextDoc text = factory.createTextDoc();
		text.load("/C:/Users/TOSHIBA/ProjectRest/dashboard.runtime/models/TextDoc.xmi");


		// Creaci—n de las entradas en el mapa inputs
		inputs.put("parametro_ejemplo_1", new Integer(23));
		inputs.put("parametro_ejemplo_2", new Double(10.5));
		inputs.put("parametro_ejemplo_3", new Boolean(true));
		inputs.put("parametro_ejemplo_4", new Boolean(false));
		inputs.put("Ejemplo STATUS", status);
		inputs.put("Ejemplo MODEL", model);
		inputs.put("Ejemplo TEXT", text);

			// Invocaci—n del servicio
		delegator.invokeService(ENDPOINT_ADDR, "servicio_ejemplo", inputs, outputs);

		// Recorremos el mapa outputs para mostrar su contenido
		Iterator<String> keyIterator = outputs.keySet().iterator();
		System.out.println("Estamos recorriendo outputs para mostrarlo:");
		String key;
		while(keyIterator.hasNext()) {
			key = keyIterator.next();
			System.out.println("\t[Name:" + key + "; value:" + outputs.get(key) +
"]");
		}

	}

}
```

A2.6 Clase DataTypeEnum

```
package dashboardRt.wsbridge.rest.utils;

public enum DataTypeEnum {

	INTEGER,DOUBLE,BOOLEAN,MODEL,TEXT,STATUS;

}
```

A2.7 Clase ServiceUtil

```
package dashboardRt.wsbridge.rest.utils;

import java.io.ByteArrayOutputStream;
import java.io.InputStream;
import java.io.StringReader;
import java.util.ArrayList;
import java.util.HashMap;
import java.util.Map;
import javax.xml.bind.JAXBContext;
import javax.xml.bind.JAXBException;
import javax.xml.bind.Marshaller;
import javax.xml.bind.Unmarshaller;
import javax.xml.transform.stream.StreamSource;
import org.codehaus.jettison.json.JSONArray;
import org.codehaus.jettison.json.JSONException;
import org.codehaus.jettison.json.JSONObject;
import dashboard.runtime.dashboarddoc.DashboarddocFactory;
import dashboard.runtime.dashboarddoc.ModelDoc;
import dashboard.runtime.dashboarddoc.TextDoc;
import dashboard.runtime.dashboarddoc.impl.ModelDocImpl;
import dashboard.runtime.dashboarddoc.impl.TextDocImpl;
import dashboard.runtime.diagnosis.SerializableStatus;
import dashboardRt.wsbridge.rest.messages.Message;
import dashboardRt.wsbridge.rest.messages.Arguments;

/**
 *
 * @author Fermin
 * @category Clase que nos proporciona métodos para conversión entre formatos, esta clase posee
los mismos método que el util para SOAP pero añadiendo varios mas para REST
 *
 */
public class ServiceUtil {

        public static ByteArrayOutputStream os,osShow;
        public static StreamSource stream,stream2;
        public static InputStream bytes;

        /**
         * @category Método utilizado para serializar el mesnsaje que será enviado al Servicio
Web, esta serialización de basa en aplicarle un formateo XML
         * @param msg, es el mensaje a serializar
         * @return La cadena XML a enviar
         * @throws JAXBException Indica si ha habido algún error serializando
         */
        public static String serializeMessage(Message msg) throws JAXBException{

                JAXBContext context = JAXBContext.newInstance(Message.class);
                Marshaller m = context.createMarshaller();
                m.setProperty(Marshaller.JAXB_FORMATTED_OUTPUT, Boolean.TRUE);
                os = new ByteArrayOutputStream();
                m.marshal(msg, os);
                return os.toString();
        }
        /**
         * @category Clase que transforma la repsuesta XML a formato Message
         * @param request, XML recibido contenedor de la respuesta
         * @return Respuesta en formato Message
         * @throws JAXBException Indica la posibilidad de errores en el proceso
         */
        public static Message deserializeMessage(String request) throws JAXBException{

                JAXBContext context = JAXBContext.newInstance(Message.class);
                Unmarshaller u = context.createUnmarshaller();
                Message msg2 = (Message) u.unmarshal(new StreamSource(new
StringReader(request)));
                return msg2;

        }
        /**
```

```
         * @category Método para serializar el mensaje a formato JSON, que es el tipo que
intercambiaran los servicios REST
         * @param msg, mensaje a serializar
         * @return Devuelve el objeto JSON listo para intercambio
         */
        public static JSONObject Message2JSON(Message msg){

                JSONObject myObject = new JSONObject();
                JSONObject aux;
                ArrayList<Arguments> argu;
                argu = msg.getArguments();
                JSONArray list = new JSONArray();
                try{
                        myObject.put("ServiceID", msg.getServiceId());
                        myObject.put("Type message", msg.getTypeMessage());
                        for (int i=0; i<argu.size();i++){
                                aux = new JSONObject();
                                aux.put("name", argu.get(i).getName());
                                aux.put("value", argu.get(i).getValue());
                                aux.put("type", argu.get(i).getDataTypeEnum());
                                list.put(aux);
                        }
                        myObject.put("Arguments",list);

                    }catch(JSONException ex){
                        System.out.println("Excepción creando JSON" + ex.getLocalizedMessage());
                }

                return myObject;

        }
        /**
         * @category Método para convertir el tipo JSON a tipo Message
         * @param myObject, Objeto JSON que será convertido
         * @return, El tipo Message
         * @throws JSONException, indica posibles errores en la conversión
         */
        public static Message JSON2Message(JSONObject myObject) throws JSONException{

                Message msg = new Message();
                ArrayList<Arguments> argu = new ArrayList<Arguments>();
                Arguments a;
                JSONObject aux;
                JSONArray jsonArray;
                msg.setServiceId((String) myObject.get("ServiceID"));
                if(myObject.get("Type message").toString().contains("n")){
                        msg.setTypeMessage(Message.typeMessage.response);
                }else{
                        msg.setTypeMessage(Message.typeMessage.Request);
                }
                jsonArray = myObject.getJSONArray("Arguments");
                for(int i=0; i<jsonArray.length();i++){
                        a = new Arguments();
                        aux = new JSONObject();
                        aux = (JSONObject) jsonArray.get(i);
                        if(aux == null){
                                System.out.println("NULL el object");
                        }
                        if(aux.get("type").toString().contains("TEXT")){
                                a.setDataTypeEnum(DataTypeEnum.TEXT);
                        }else if(aux.get("type").toString().contains("STATUS")){
                                a.setDataTypeEnum(DataTypeEnum.STATUS);
                        }else if(aux.get("type").toString().contains("BOOLEAN")){
                                a.setDataTypeEnum(DataTypeEnum.BOOLEAN);
                        }else if(aux.get("type").toString().contains("MODEL")){
                                a.setDataTypeEnum(DataTypeEnum.MODEL);
                        }else if(aux.get("type").toString().contains("INTEGER")){
                                a.setDataTypeEnum(DataTypeEnum.INTEGER);
                        }else if(aux.get("type").toString().contains("DOUBLE")){
                                a.setDataTypeEnum(DataTypeEnum.DOUBLE);
                        }
                        a.setName(aux.getString("name"));
                        a.setValue(aux.getString("value"));
```

```
                argu.add(a);
        }
        msg.setArguments(argu);
        return msg;
}

/**
 * @category Método para configurar los Argumentos en un Message
 * @param map Map contenedor de los datos
 * @param msg Los datos en formato Message, es el resultado del método
 */
public static void Argument2Message(Map<String, Object> map, Message msg){

        Object[] keys = map.keySet().toArray();
    Object[] values = map.values().toArray();
    Arguments in;
        ArrayList<Arguments> array = new ArrayList<Arguments>();

    int i=0;

    while(i<map.size()){

        if(values[i].getClass()==Integer.class){


                in = new Arguments();
                in.setName(keys[i].toString());
                in.setValue(values[i].toString());
                in.setDataTypeEnum(DataTypeEnum.INTEGER);
                array.add(in);


        }else if(values[i].getClass()==Double.class){

                in = new Arguments();
                in.setName(keys[i].toString());
                in.setValue(values[i].toString());
                in.setDataTypeEnum(DataTypeEnum.DOUBLE);
                array.add(in);

        }else if(values[i].getClass()==Boolean.class){

                in = new Arguments();
                in.setName(keys[i].toString());
                in.setValue(values[i].toString());
                in.setDataTypeEnum(DataTypeEnum.BOOLEAN);
                array.add(in);

        }else if(values[i].getClass()==SerializableStatus.class){

                in = new Arguments();
                in.setName(keys[i].toString());
                SerializableStatus status = (SerializableStatus) values[i];
                String statusValue = status.serialize();
                in.setValue(statusValue);
                in.setDataTypeEnum(DataTypeEnum.STATUS);
                array.add(in);

        }else if(values[i].getClass()==ModelDocImpl.class){

                in = new Arguments();
                in.setName(keys[i].toString());
                ModelDoc model = (ModelDoc) values[i];
                String modelValue = model.serialize();
                in.setValue(modelValue);
                in.setDataTypeEnum(DataTypeEnum.MODEL);
                array.add(in);

        }else if(values[i].getClass()==TextDocImpl.class){

                in = new Arguments();
                in.setName(keys[i].toString());
                TextDoc text = (TextDoc) values[i];
```

```
                        String textValue = text.serialize();
                        in.setValue(textValue);
                        in.setDataTypeEnum(DataTypeEnum.TEXT);
                        array.add(in);

                }else{

                        System.out.println("No es ningun tipo de los establecidos" +
keys[i].toString() + "\n" +values[i].getClass());
                }

                i++;
            }

            msg.setArguments(array);


        }
        /**
         * @category Conversión de Map a JSON
         * @param Mapa de entrada
         * @return Json con la información de Map
         * @throws JSONException indica si ha habido errores en la conversión
         */
        public static JSONObject map2JSON(Map<String, Object> inputs) throws JSONException{

                JSONObject json = new JSONObject();
                json.put("parametro_ejemplo_1", inputs.get("parametro_ejemplo_1"));
                json.put("parametro_ejemplo_2", inputs.get("parametro_ejemplo_2"));
                json.put("parametro_ejemplo_3", inputs.get("parametro_ejemplo_3"));
                json.put("parametro_ejemplo_4", inputs.get("parametro_ejemplo_4"));
                json.put("Ejemplo STATUS", inputs.get("Ejemplo STATUS"));
                json.put("Ejemplo MODEL", inputs.get("Ejemplo MODEL"));
                json.put("Ejemplo TEXT", inputs.get("Ejemplo TEXT"));
                return json;

        }
        /**
         * @category Método no relevante para la realización del proyecto, pero considero que
ver como queda el archivo formateado a JSON ayuda a entender el proceso de eintercambio de
mensajes entre WS y clientes
         * @param json, Json a ser mostrado
         */
        public void showJSON( JSONObject json){

                System.out.println("Vemos lo que tiene JSON :"  + json.toString());
        }

        /**
         * @category Método para ver el contenido de un Message
         * @param msg, mensaje a evaluar
         */
        public void showMessage(Message msg){
                ArrayList<Arguments> inputsArray = new ArrayList<Arguments>();
                if(msg == null){
                        System.out.println("mensaje vacio!!");
                }else{
                        System.out.println("Contenido del sms :" + msg.getServiceId());
                        inputsArray = msg.getArguments();
                        for (int i=0 ; i<inputsArray.size(); i++){

                                Arguments n = inputsArray.get(i);
                                System.out.println("Request inputs tiene : " + n.getName() +
"\n"+ "Valor :" + n.getValue() + "\n" + "Tipo :" + n.getDataTypeEnum());

                        }
                }
        }
        /**
         * @category Método que pasa la información de JSON a Map
         * @param json, parametro contenedor de la información en JSON
         * @return Map con la información
         * @throws JSONException indica la posibilidad de errores en el proceso
```

```
         */
        public static HashMap<String, Object> JSON2Map(JSONObject json) throws JSONException{

                HashMap<String, Object> map = new HashMap<String, Object>();
                map.put("parametro_ejemplo_1", json.get("parametro_ejemplo_1"));
                map.put("parametro_ejemplo_2", json.get("parametro_ejemplo_2"));
                map.put("parametro_ejemplo_3", json.get("parametro_ejemplo_3"));
                map.put("parametro_ejemplo_4", json.get("parametro_ejemplo_4"));
                map.put("Ejemplo STATUS", json.get("Ejemplo STATUS"));
                map.put("Ejemplo MODEL", json.get("Ejemplo MODEL"));
                map.put("Ejemplo TEXT", json.get("Ejemplo TEXT"));
                return map;


        }
        /**
         * Método no necesario para la realización del proyecto, pero creo que su utilidad puede
ser de ayuda a comprender como son los tipos
         * de archivo que se intercambian y a comprender la serialización. El método muestra un
XML sobre un mensaje de entrada
         * @param msg, mensaje con el que se forma el XML
         * @throws JAXBException indica la posibilidad de errores en el proceso
         */
        public static void showXML(Message msg) throws JAXBException{

                JAXBContext context = JAXBContext.newInstance(Message.class);
                 Marshaller m = context.createMarshaller();
                 m.setProperty(Marshaller.JAXB_FORMATTED_OUTPUT, Boolean.TRUE);
                 osShow = new ByteArrayOutputStream();
                 m.marshal(msg, osShow);
                 System.out.println("Mostramos mensaje serializado a XML : " +
osShow.toString());

        }
        /**
         * @category Metodo que extrae los datos de Message y los pasa a Arguments, paso
necesario para su desmenuce y futura muestra
         * @param msg Mensaje contenedor de los datos
         * @param map resultado
         */
        public static void message2Argument(Message msg, Map<String, Object> map){

                ArrayList<Arguments> mapsArray = msg.getArguments();

                for(int i=0; i<mapsArray.size();i++){

                        Arguments m = mapsArray.get(i);

                        if(m.getDataTypeEnum().toString()=="DOUBLE"){

                                map.put(m.getName(),Double.parseDouble(m.getValue()));

                        }else if(m.getDataTypeEnum().toString()=="INTEGER"){

                                map.put(m.getName(), Integer.parseInt(m.getValue()));

                        }else if(m.getDataTypeEnum().toString()=="BOOLEAN"){

                                map.put(m.getName(), Boolean.parseBoolean(m.getValue()));

                        }else if(m.getDataTypeEnum().toString()=="STATUS"){

                                SerializableStatus status2 = new SerializableStatus();
                                status2.deserialize(m.getValue());
                                map.put(m.getName(), status2);

                        }else if(m.getDataTypeEnum().toString()=="MODEL"){

                                DashboarddocFactory factory = DashboarddocFactory.eINSTANCE;
                                ModelDoc model = factory.createModelDoc();
                                model.deserialize(m.getValue());
                                map.put(m.getName(), model);
```

```
                        }else if(m.getDataTypeEnum().toString()=="TEXT"){

                                DashboarddocFactory factory = DashboarddocFactory.eINSTANCE;
                                TextDoc text2 = factory.createTextDoc();
                                text2.deserialize(m.getValue());
                                map.put(m.getName(),text2);

                        }else{

                                System.out.println("ERROR, tipo de dato no contemplado" +
m.getValue().toString() +"\n" + m.getName());
                        }


                }

        }
```