Arquitectura cliente-servidor con cliente J2ME e intercambio de datos entre capas basado en XML

Abstract

Se plantea aquí el uso de XML para transporte de los datos entre las capas de negocios y de presentación, en particular para clientes desarrollados en J2ME.
Se analizan las distintas posibilidades para el envío de datos y se expone una estructura para soportar la utilización de XML, así como una implementación. Se introduce además una herramienta creada por el autor para desarrollos de estas características.

Introducción

El patrón DTO (Data Transfer Objects)

Este patrón se ocupa de cómo se intercambian datos entre capas de una aplicación multi-capas o entre componentes de una aplicación distribuida.
Se debe tener en cuenta que el acceso a datos desde un componente remoto muchas veces implica extraer información de más de una entidad de información. Y, en esa multiplicidad de comunicaciones solicitando datos, la performance de la aplicación se vería resentida.
Formalmente entonces, el objetivo de este patrón es, solucionar la perdida de performance en componentes distribuidos que hacen llamadas remotas para acceder a datos de los objetos de negocios.
Aquí llegamos a la definición de DTO. Un DTO es básicamente un objeto creado exclusivamente para transportar datos. Datos que pueden tener su origen en una o más entidades de información. Estos datos son incorporados a una instancia de un JavaBean. Y esta instancia puede ser pasada a través de la aplicación, localmente o, lo que es más importante, puede ser serializada y enviada a través de la red, de forma que los clientes puedan acceder a información del modelo desde un solo objeto y mediante una sola llamada.
Un DTO normalmente no provee lógica de negocios o validaciones de ningún tipo. Solo provee acceso a las propiedades del bean. Algunos autores remarcan que el bean debe ser inmutable, dado que sus cambios no deben reflejarse en el sistema. Pero obviamente esto choca con la especificación de los JavaBean, que requiere que todos los atributos privados tengan sus métodos set y get.
Como siempre la última palabra la tiene el desarrollador, que determinará la mejor manera de manejar la mutabilidad de los DTO en su aplicación.
En cualquier caso y para terminar, los DTO deben ser considerados parte del modelo en tanto que son copias inmutables de los objetos de negocios.

Los objetos DTO y la API J2ME

La serialización es básicamente el proceso por el cual transformamos un objeto en un flujo de bytes para mandarlo a algún lado. Para una vez llegado allí, volver a re-armarlo como objeto y utilizarlo localmente.
Debido a la ausencia en J2ME tanto de soporte para serialización como de soporte para reflexión, (mecanismo mediante el cual los propios objetos son capaces de inspeccionarse a sí mismos, averiguando sus métodos, atributos, parámetros, constructores, etc.) la serialización

que se puede llevar a cabo debe ser desarrollada ad-hoc y adolecerá de ciertas limitaciones. Esto es, no podrá ser totalmente automatizada.
Para lograr una serialización en J2ME el desarrollador deberá crear una interface *serializable*, con métodos como writeObject y readObject. Y hacer luego que sus objetos DTO la implementen. Esto es, deberá proveer implementaciones "especializadas" de esos métodos, para cada objeto.
Si tuviéramos un objeto con los siguientes atributos:

```
Long puerto;
Long bb;
String ruta;
Int x;
```

Los métodos para su serialización serían:

```
public void writeObject(ObjectOutputStream out) throws IOException{
out.writeLong(puerto);
out.writeLong(bb);
out.writeInt(cc);
out.writeUTF(ruta);
out.writeInt(x);
}

public void readObject(ObjectInputStream in) throws IOException{
puerto = in.readLong();
bb = in.readLong();
cc = in.readInt();
ruta = in.readUTF();
x = in.readInt();
}
```

Nótese que ambos métodos implican a las clases ObjectInputStream y ObjectOutputStream que también deben ser creadas y que extienden de DataInputStream y de DataOutputStream respectivamente. Y poseen, por lo tanto, los métodos de estas (para, agregar-al/extraer-del, flujo los distintos tipos de datos) y además métodos propios para leer objetos desde un flujo *readObject()* (la clase ObjectInputStream) y para escribirlos en él, *writeObject(Serializable object)* (de la clase ObjectOutputStream) que utilizan obviamente los métodos readObject y writeObject de la interface serializable .
Para más información sobre serialización de objetos en J2ME se puede consultar [1]. Y en [5] se puede encontrar un excelente artículo en español. Y también es muy interesante el planteo de [6] relativo a la performance de los distintos algoritmos de serialización (recursivos y no recursivos) en J2ME.

Objetos XML

Siendo que los objetos DTO por definición no llevan lógica y que J2ME no ofrece un mecanismo de serialización que nos haga transparente el intercambio de objetos aparece como una opción, muy razonable, la utilización de XML para codificar a los objetos de intercambio.

Y es que además de que la implementación de un mecanismo Objeto-XML-Objeto no diferiría demasiado de los visto para serialización de objetos, tendríamos la ventaja adicional de que el XML resulta legible, tanto por ordenadores, como por seres humanos. Y su utilización, como veremos luego, facilitaría enormemente las tareas de debugging, dado que para generar instancias distintas de un mismo objeto bastaría con editar a mano el archivo XML.

Cualquier browser, además, serviría para enviar un request al servidor y observar, en el mismo, la respuesta. Y es que si bien nuestra finalidad, en este caso puntual, es el desarrollo de un cliente en J2ME, poder chequear de un vistazo que la respuesta del servidor sea la apropiada es siempre una herramienta de debugging muy interesante. Y por último cabe destacar que la utilización de XML posibilita a nuestra aplicación dar respuesta tanto a clientes J2ME como J2SE (o páginas JSP) o .NET o lo que fuere.
Lógicamente el uso de XML conlleva actividades de parseo y análisis sintáctico cuyo desarrollo es mejor, si se puede, evitar. En J2ME si bien la API, por supuesto, tampoco trae facilidades para tratar con XML, se puede disponer de un framework que resuelve estos asuntos en muy pocos Kb. La librería en cuestión se llama KXML [2].

Implementación del intercambio de datos entre cliente-servidor basado en XML

El proceso de desarrollo

Una de las cosas importantes a resolver cuando se desarrolla un proyecto multi-capas es la definición de los objetos de intercambio de datos entre las capas de presentación y de negocios. Estos objetos de intercambio, lógicamente, estarán fuertemente condicionados por la "vista" que se desea obtener en el cliente.
Una vez acordados, los ingenieros de back-end construirán los servicios que devolverán esos objetos y los ingenieros de front-end se abocarán a construir el cliente que los consumirá.
Este es el punto exacto donde pueden empezar los problemas en el equipo. Esto es, si la capa de presentación, debe esperar por la de negocios para probar el cliente pueden surgir demoras indeseadas. Para esto la mejor solución es, luego de acordados los objetos de intercambio (DTO) encontrar la forma de generar objetos mock que nos provean a pedido los objetos que necesitamos para probar nuestro cliente.
Durante un desarrollo reciente desarrollé una herramienta, ***MockMe***, muy sencilla y adaptable que puede bajarse de [3] y en tanto que es un proyecto Open Source, puede ser modificado como se desee. Se trata básicamente de un servlet configurable a base de archivos .properties que busca XML's de un directorio local, en función de un parámetro que se le pasa en la request. Lo dicho, muy fácil y muy efectivo a la hora de despegarnos de los tiempos de desarrollo de la capa de negocios. *And last but no least*, de la disponibilidad de datos específicos también. En [3] además de la propia herramienta y algunos XML's de ejemplo puede obtenerse información sobre su instalación y configuración.

Separando objetos

En parte a partir del requisito de inmutabilidad deseable en los DTO y en parte por una cuestión de prolijidad y organización en nuestro desarrollo, es conveniente tener dos tipos de objetos para cada entidad de información, esto es: objetos locales y objetos DTO.
Por ejemplo, si utilizáramos internamente un objeto Persona y necesitáramos enviar sus datos, allende la JVM, deberíamos tener también PersonaDTO, que sería la versión de persona que circularía a través de la red (en XML). Es decir, si el servidor nos enviara un objeto Persona este nos llegaría en XML y al extraerlo obtendríamos un objeto PersonaDTO
En el caso puntual de nuestra arquitectura, al haber optado por utilizar XML como soporte, la conversión de nuestros datos quedaría:

ObjetoDTO-XML-ObjetoDTO

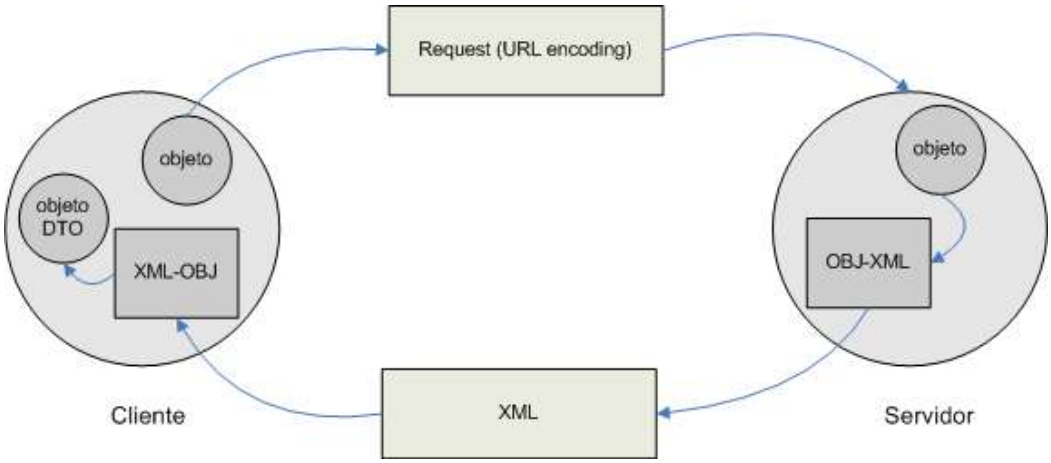

Figura 1. *Comunicación entre capas*

Arquitectura del cliente

La arquitectura del cliente es conveniente que esté basada en algunos patrones típicos. El MVC para, mediante un objeto Controller [4], generar las vistas obteniendo los datos del Modelo a partir de la invocación de métodos de un objeto Facade que a su vez encapsule las peticiones. Llamando tanto al Modelo Local (almacenamiento RMS) como al Modelo Remoto (http) según toque.

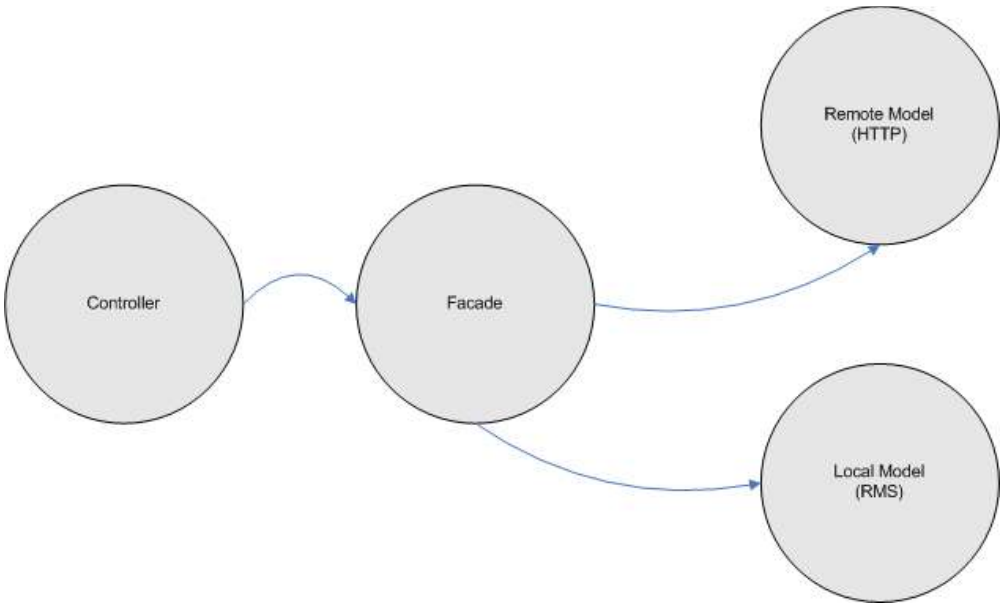

Figura 2. *Arquitectura de la aplicación*

Implementación

Para implementar la transferencia de datos vía XML es útil hacer que nuestros objetos implementen ciertas interfaces. En el caso de los objetos de negocios es útil hacerlos extender de una interface como la siguiente:

```
<IBusinessObject>
      public String toShow();
      public String toRequestString();
```

El método *toShow()* es útil para la visualización del objeto en la interfaz de usuario (así como para tareas de debugging). Y el método *toRequestString()* es necesario para que desarrolle el URL encoding que genere el request para ese objeto. A continuación ejemplos de ambos:

```
public String toShow() {
      Calendar cal = Calendar.getInstance();
      cal.setTime(getFecha());

      String txt = "Censo = " + getCenso()
            + " \nFecha fichaje = " + FechaUtil.formatDate(getFecha())
            + " \nJornada de comienzo = " + getJornadaComienzo()
            + " \nJornadas = " + getTurnos()
            + " \nSustituye = " + (isReemplaza()? "SI":"NO");

      return txt;
}
```

```
public String toRequestString() {
      StringBuffer buff = new StringBuffer();
      buff.append("codoper=" + getCodoper()
                  + "&jornadacomienzo=" + getJornadaComienzo()
                  + "&turnos=" + getTurnos()
                  + "&reemplaza=" + isReemplaza()
                  + "&movil=" + getMovil());
return buff.toString();
}
```

En la siguiente Figura se grafica su utilización:

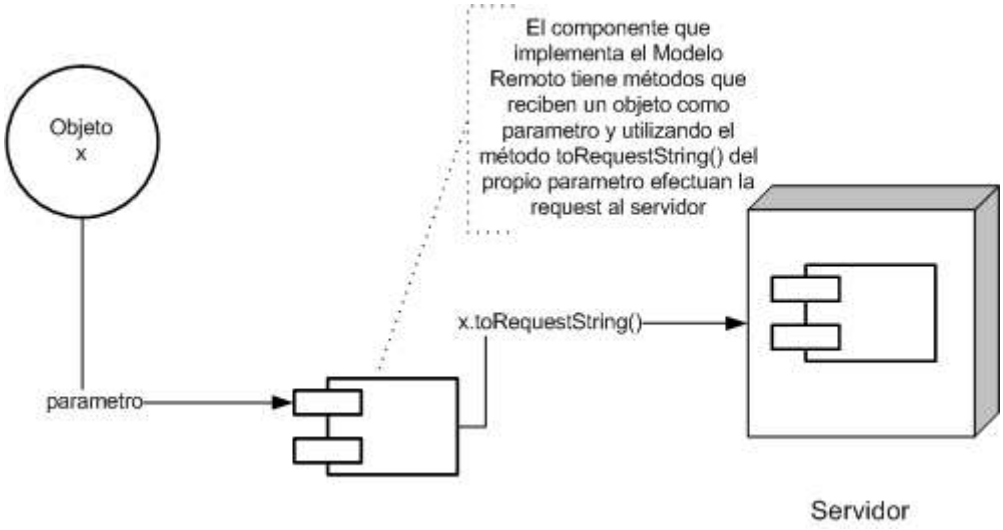

Figura 3. *Pasos para el envío de un requerimiento por URL encoding*

El proceso de recibir la respuesta del servidor y transformarla en un objeto se ve en la Fig. 4. Y consta de una serie de pasos intermedios, implementados por métodos del Modelo Remoto.

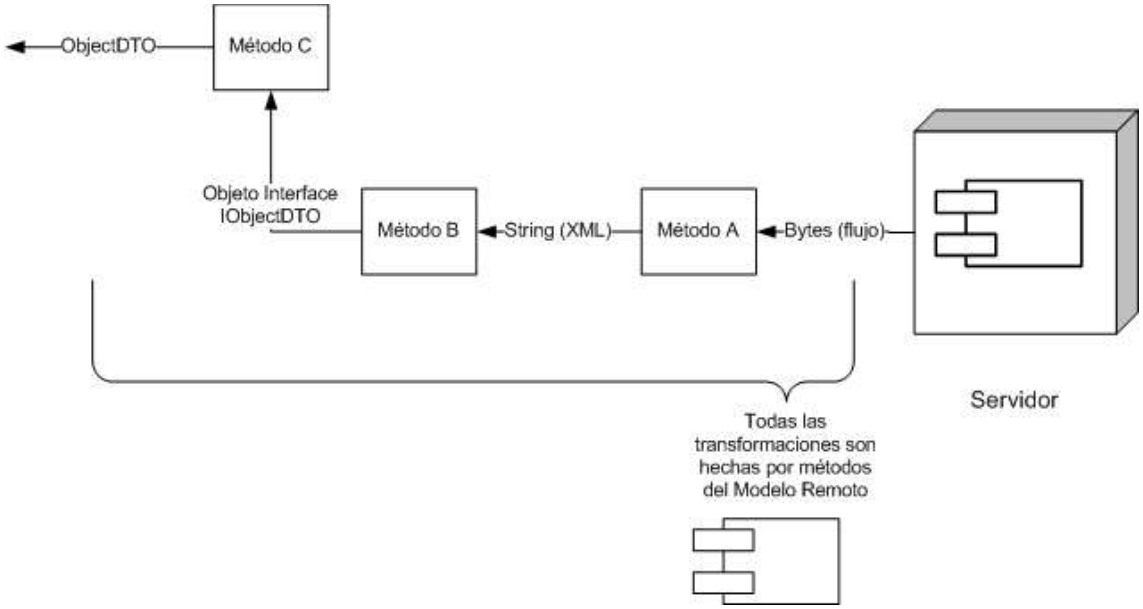

Figura 4. *Pasos para la transformación de la respuesta del servidor en un objeto DTO*

Los objetos DTO

Siendo que los objetos circularan entre capas como flujos XML, estos objetos DTO deberán tener métodos tanto para extraer los datos del objeto como, a la inversa, para darle estado al objeto a partir de un conjunto de valores encapsulados en una Hashtable. Nos resultará necesario definir un contrato con ellos. A continuación la interface para los objetos DTO:

```
<IObjectDTO>

public void populateFields(Hashtable fields)
Hashtable extractFields()
```

El hecho de que implementen esta interface nos será útil también, para simplificar la recuperación de los objetos utilizando el polimorfismo.
Si, por ejemplo, tratáramos con un objeto Persona que tuviera los siguientes atributos:

- id
- fechaNacimiento
- estado
- experiencia
- direccion
- pais

Una implementación factible para los métodos anteriores sería la siguiente:

```
public void populateFields(Hashtable fields) {

this.id = (String) fields.get("id");
```

```
      this.fechaNacimiento = new Date(Long.parseLong((String)fields.get
("fechaNacimiento")));
      this.estado = Integer.parseInt((String)fields.get("estado"));
      this.experiencia = Integer.parseInt((String)fields.get("experiencia"));
      this.direccion = (String) fields.get("direccion");
      this.pais = (String) fields.get("pais");
      }

public Hashtable extractFields() {
      Hashtable fields = new Hashtable();
      fields.put("id", this.censo);
      fields.put("fechaNacimiento", "" + this.fechaNacimiento.getTime());
      fields.put("estado", "" + this.estado);
      fields.put("experiencia", "" + this.experiencia);
      fields.put("direccion", this.direccion);
      fields.put("pais", this.pais);
      return fields;
      }
```

El método A de la figura, podría implementarse como sigue (*backendComms()*). Será el encargado de abrir la conexión con el servidor (sus parámetros de entrada son la URL del servidor y la request) y de devolver un objeto String con el XML.

```
public static String backendComms(String requestURL, String requeststring) throws
ApplicationException
{
      hc = null;
      dis = null;
      StringBuffer messagebuffer = new StringBuffer();
      String requestString = requestURL + requeststring;

            try {
                  hc = (HttpConnection) Connector.open(requestString,
                              Connector.READ_WRITE, true);
                  hc.setRequestMethod(HttpConnection.GET);
                  hc.setRequestProperty("User-Agent",
                              "Profile/MIDP-2.0 Configuration/CLDC-1.1");
                  hc.setRequestProperty("Content-Language", "es-ES");
                  dis = new DataInputStream(hc.openInputStream());
                  int ch;
                  long len = hc.getLength();
                  if (len != -1) {
                        for (int i = 0; i < len; i++) {
                                    if ((ch = dis.read()) != -1) {
                                    messagebuffer.append((char) ch);
                              }
                        }
                  } else {
                        while ((ch = dis.read()) != -1) {
                              messagebuffer.append((char) ch);
                        }
                  }
                  dis.close();

            } catch (InterruptedIOException iioe) {
                  messagebuffer = new StringBuffer("Time-out en la red.");
            throw new ApplicationException(messagebuffer.toString());

            } catch (IOException ioe) {
                  messagebuffer = new StringBuffer("Problema de servidor caído.
Intente luego");
            throw new ApplicationException(messagebuffer.toString());

            } catch (ApplicationException ae) {
                  messagebuffer = new StringBuffer("Proceso interrumpido.");
            throw new ApplicationException(messagebuffer.toString());
```

```
            } catch (Exception e) {
                  messagebuffer = new StringBuffer("Problema de red. Intente
luego");
            throw new ApplicationException(messagebuffer.toString());

            }finally {
                  cerrarConexion();
            }
            return messagebuffer.toString();
      }
```

Luego de obtenido el String con el XML debemos extraer de él, el *IObjectDTO* que corresponda. Es claro que además de los datos pertinentes, el XML deberá identificar la implementación de *IObjectDTO* a la que correspondan esos datos.
En nuestro ejemplo lo haremos en dos pasos. Primero se parseará el XML en forma genérica para obtener un objeto usando la interface *IObjectDTO*. Y luego por casting se lo llevará al objeto esperado.
El parseo del XML, como veremos más adelante, requerirá de una clase *XMLUtil* que utilizará la API provista por KXML.

Supongamos por ejemplo que deseamos obtener un objeto del tipo PersonaDTO. En el modelo, se tendrán distintos métodos para obtener los distintos objetos DTO. Para este caso utilizaremos uno llamado *getPersona()*.

```
public PersonaDTO getPersona(Persona _persona) throws ApplicationException
{
      StringBuffer requestString = new StringBuffer();
      String tempStr = QUESTION + _persona.toRequestString();
      requestString.append(tempStr);
      String xml = backendComms(LOAD_URL, requestString.toString());
       PersonaDTO personaDTO = convertXMLToPersonaDTO(xml);
return personaDTO;
}
```

En la línea en negrita vemos la llamada al método C de la Figura 4. Que hace el casting, luego de llamar al método que extrae el objeto del XML (*convertXMLToIObjectDTO(String xml)*).

```
private static PersonaDTO convertXMLToPersonaDTO(String _xml) {
   PersonaDTO persona = (PersonaDTO)convertXMLToIObjectDTO(_xml);
return persona;
}
```

Este que sigue sería entonces el método B de la Figura 4, que es el que utiliza la clase XMLutil para parsear el XML:

```
private static IObjectDTO convertXMLToIObjectDTO(String xml) {
      InputStreamReader isr = new InputStreamReader(new ByteArrayInputStream
(
                        xml.getBytes()));
      IObjectDTO ioDto = null;
      try {
            ioDto = XMLUtil.getObjectFromXML(isr);
      } catch (IOException ioe) {
            System.out.println("Problemas generando el objeto IObjectDTO");
            ioe.printStackTrace();
            }
```

```
return ioDto;
}
```

Aquí una implementación posible para la clase XMLUtil

```
public class XMLUtil {

public static IObjectDTO getObjectFromXML(InputStreamReader insr) throws IOException
{

	IObjectDTO obj = null;
	XmlParser parser = new XmlParser(insr);
	Document document = new Document();
	document.parse(parser);

		// Para mostrar el contenido del documento XML en la consola
		// usar document.write( new XmlWriter( new OutputStreamWriter(
		// System.out) ) );

	Element objectElement = document.getElement("object");
	obj = getObjectFromXMLElement(objectElement);

return obj;
}

public static IObjectDTO getObjectFromXMLElement(Element objectElement)
{

	String className = getTextFromElement(objectElement, "classname");
	Element fieldsElement = objectElement.getElement("fields");
	int childCount = fieldsElement.getChildCount();
	Hashtable fields = new Hashtable();
	Element fieldElement = null;

	for (int i = 0; i < childCount; i++) {
	if (fieldsElement.getType(i) == Xml.ELEMENT) {
		fieldElement = fieldsElement.getElement(i);

		String fieldName = getTextFromElement(fieldElement, "name");

		// El campo es un objeto
		if (fieldElement.getElement("value").getAttributeCount() > 0) {
			IObjectDTO object = getObjectFromXMLElement
(fieldElement.getElement("value").getElement("object"));
			fields.put(fieldName, object);
		}
		// El campo es un String
		else {
		String fieldValue = stripCData(getTextFromElement(fieldElement,
"value"));
		fields.put(fieldName, fieldValue);
				}
			}
		}

		// Crear el objeto especificado en classname
		IObjectDTO obj = null;
		try {
			Class clase = Class.forName(className);
			obj = (IObjectDTO) clase.newInstance();
			obj.populateFields(fields);
		} catch (Exception e) {
		System.out.println("Excepción al crear la clase: " + e.getMessage());
		e.printStackTrace();
		}

		return obj;
	}
```

```
public static StringBuffer getXMLFromObject(IObjectDTO object, int level) throws
IOException {
	StringBuffer objectToXML = new StringBuffer();
	String tabs = "";
	Class clase = object.getClass();
	String className = clase.getName();

	// Obtener número de tabs
	for (int i = 0; i < level; i++)
		tabs += "\t\t\t\t";

	// Si es el inicio de un documento, mostrar la cabecera xml
	if (level == 0)
	objectToXML.append("<?xml version=\"1.0\" encoding=\"ISO-8859-1\"?>\n\n");

	// Mostrar los datos del objeto
	objectToXML.append(tabs + "<object>\n");
	objectToXML.append(tabs + "\t<classname>" + className + "</classname>\n");
	objectToXML.append(tabs + "\t<fields>\n");

	Hashtable campos = object.extractFields();
	Enumeration e = campos.keys();

	while (e.hasMoreElements()) {
		Object key = e.nextElement();
		objectToXML.append(tabs + "\t\t<field>\n");

		objectToXML.append(tabs + "\t\t\t<name>" + key + "</name>\n");
		if (campos.get(key) instanceof String) {
		objectToXML.append(tabs + "\t\t\t<value><![CDATA[" + campos.get(key) +
"]]></value>\n");
		} else if (campos.get(key) instanceof IObjectDTO) {
		objectToXML.append(tabs + "\t\t\t<value nested=\"true\">\n");
		objectToXML.append(getXMLFromObject((IObjectDTO) campos.get(key), level
+ 1));
		objectToXML.append(tabs + "\t\t\t</value>\n");
			}

		objectToXML.append(tabs + "\t\t</field>\n");
		}

	objectToXML.append(tabs + "\t</fields>\n");
	objectToXML.append(tabs + "</object>\n");

	return objectToXML;

	}

private static String getTextFromElement(Element elementRoot, String elementName) {
	String returnText = elementRoot.getElement(elementName).getText();

	return returnText;
	}

private static String stripCData(String value) {
	String prefix = "<![CDATA[";
	String sufix = "]]>";

	if (value.indexOf(prefix) >= 0)
		value = value.substring(value.indexOf(prefix) + prefix.length());
	if (value.indexOf(sufix) >= 0)
		value = value.substring(value.indexOf(sufix));

	return value;
	}

}
```

Como vemos, la clase XMLUtil utiliza ambos métodos de la interface IObjectsDTO (*populateFields(Hashtable fields)* y *extractFields()*) para las conversiones

(XML a Objeto) y (Objeto a XML) respectivamente. Y se utiliza consecuentemente tanto del lado del cliente como del lado del servidor.
Finalmente vemos un ejemplo de XML (en este caso enviado por el servidor como resultado de una operación de login):

```
<?xml version="1.0" encoding="ISO-8859-1"?>

<object>
	<classname>org.lm2a.bebay.dtoobjects.LoginDTO</classname>
	<fields>
		<field>
			<name>version</name>
			<value><![CDATA[1.1.5]]></value>
		</field>
		<field>
			<name>numMensajes</name>
			<value><![CDATA[0]]></value>
		</field>
		<field>
			<name>mensaje</name>
			<value><![CDATA[Autentificado correctamente]]></value>
		</field>
		<field>
			<name>fechaAccion</name>
			<value><![CDATA[1161772420625]]></value>
		</field>
		<field>
			<name>codigo</name>
			<value><![CDATA[00]]></value>
		</field>
	</fields>
</object>
```

Una vez construidas las clases (XMLUtil) y los métodos de soporte genéricos (`backendComms`), para el agregado de nuevos objetos solo deben ser creadas sus clases (que implementarán IobjectDTO) y los métodos para invocar los servicios que los devuelvan (tanto en Facade como en el Modelo Remoto).

Referencias

[1] Object Serialization in CLDC-Based Profiles Available: http://java.sun.com/developer/J2METechTips/2002/tt0226.html#tip2

[2] KXML website, available: http://kxml.objectweb.org/

[3] MockMe project, available: http://mario.lamenza.googlepages.com/home

[4] Patterns Roadmap. Available:

http://java.sun.com/developer/technicalArticles/J2EE/patterns/J2EEPatternsRoadmap.html

[5] Celeste Campo Vazquez, Rosa Ma Garcıa Rioja, Guillermo Diez-Andino Sancho, "Desarrollo de un mecanismo de serializacion en J2ME". Available: www.it.uc3m.es/celeste/papers/Serializacion.pdf

[6] Antonio J. Sierra, Recursive and non-recursive method to serialize object at J2ME . Available: http://www.w3.org/2006/02/Sierra10022006.pdf

Sobre el autor:

Mario La Menza es líder de Proyectos para Tecnologías Java en Inerza SA. Y actualmente está terminando su doctorado en Informática por la ULPGC. Recientemente un artículo de su autoría, sobre agentes Jade y videostreaming en situaciones de movilidad, fue premiado con el "Best Paper Award" en la IMECS 2006 (Hong Kong).

e-mail: *mario.lamenza@gmail.com*