Universidad Nacional Autónoma de México

Facultad de Ciencias

# Un generador de documentos de formato Open Office XML

REPORTE DE TRABAJO PROFESIONAL

QUE PARA OBTENER EL TÍTULO DE:
LICENCIADO EN CIENCIAS DE LA COMPUTACIÓN

PRESENTA:
EDMUNDO ERNESTO BADILLO FERNÁNDEZ

DIRECTOR DE TRABAJO:
DRA. AMPARO LÓPEZ GAONA

2014

Universidad Nacional
Autónoma de México

Biblioteca Central

Dirección General de Bibliotecas de la UNAM

**UNAM – Dirección General de Bibliotecas**
**Tesis Digitales**
**Restricciones de uso**

**DERECHOS RESERVADOS ©**
**PROHIBIDA SU REPRODUCCIÓN TOTAL O PARCIAL**

Todo el material contenido en esta tesis esta protegido por la Ley Federal del Derecho de Autor (LFDA) de los Estados Unidos Mexicanos (México).

El uso de imágenes, fragmentos de videos, y demás material que sea objeto de protección de los derechos de autor, será exclusivamente para fines educativos e informativos y deberá citar la fuente donde la obtuvo mencionando el autor o autores. Cualquier uso distinto como el lucro, reproducción, edición o modificación, será perseguido y sancionado por el respectivo titular de los Derechos de Autor.

**Datos del Jurado**

1. Datos del alumno
   Badillo
   Fernández
   Edmundo Ernesto
   26 15 64 57
   Universidad Nacional Autónoma de México
   Facultad de Ciencias
   Ciencias de la Computación
   403001194

2. Datos del tutor
   Dra
   Amparo
   López
   Gaona

3. Datos del sinodal 1
   Dra
   Elisa
   Viso
   Gurovich

4. Datos del sinodal 2
   Dr
   José de Jesús
   Galaviz
   Casas

5. Datos del sinodal 3
   M en I
   Gerardo
   Avilés
   Rosas

6. Datos del sinodal 4
   Mat
   Salvador
   López
   Mendoza

7. Datos del trabajo escrito
   Un generador de documentos de formato Open Office XML
   42 p
   2014

# Un generador de documentos de formato Open Office XML

Edmundo Ernesto Badillo Fernández

Febrero de 2014

# Índice general

**Introducción** **III**

**1. Conceptos y Tecnologías Principales** **1**
- 1.1. XML . . . . . 1
  - 1.1.1. DOM . . . . . 2
  - 1.1.2. Esquema XML . . . . . 2
  - 1.1.3. XPath . . . . . 3
- 1.2. SOA . . . . . 4
- 1.3. SOAP . . . . . 4
- 1.4. Patrón de Diseño Visitor . . . . . 5

**2. Requerimientos** **6**
- 2.1. Requerimientos Funcionales . . . . . 7
- 2.2. Requerimientos No Funcionales . . . . . 7

**3. Solución Propuesta** **9**
- 3.1. Office Open XML (OOXML) . . . . . 10
  - 3.1.1. Controles de contenido . . . . . 11
  - 3.1.2. Cumplimiento de los requerimientos funcionales . . . . . 16

**4. Arquitectura** **17**

**5. Servicio Generador de Documentos** **21**
- 5.1. Servicio de Documentum . . . . . 25
- 5.2. DocumentoDocxDto . . . . . 25
- 5.3. Pool de Plantillas . . . . . 26
- 5.4. Manipulación de Documentos . . . . . 28
  - 5.4.1. La clase DocxUtils . . . . . 29

5.4.2. Uso del patrón de diseño Visitor . . . . . . . . . . . . . 31
5.4.3. Actualización de Controles de Contenido . . . . . . . . 32
5.5. Manipulación del formato ZIP . . . . . . . . . . . . . . . . . . . . 34
5.6. GenerarDocumentoService . . . . . . . . . . . . . . . . . . . . . . 39

**6. Conclusiones** **41**

# Introducción

El trabajo que aquí presento lo realicé como parte de mis servicios prestados a la empresa Top Personnel S. de R.L. de C.V. en un proyecto comisionado por el Servicio de Administración Tributaria, SAT, el cual fue nombrado SDMA *(Servicio de Desarrollo y Mantenimiento de Aplicaciones)*. El SAT se encontraba en una fase de modernización de su infraestructura tecnológica que tenía como uno de sus principales objetivos el simplificar y automatizar procesos existentes. Como parte de esta modernización, el SAT solicitó el desarrollo de un sistema para generar documentos de formato Office Open XML (formato usado por Microsoft Word en los archivos de extensión DOCX) y PDF a partir de plantillas creadas previamente. Este sistema sería usado en una gran variedad de casos de uso que requerían del llenado automático de documentos oficiales con un formato fijo. Muchos de estos documentos debían ser firmados digitalmente por lo que era necesario un sistema central encargado de generar documentos que soportara la inserción de firmas digitales. Dada la cantidad de tipos de documentos que se debían soportar, las plantillas tenían que ser fáciles de crear y editar. La información para llenar estas plantillas sería obtenida dinámicamente de bases de datos y servicios web existentes. Desde luego, la solución debía ser escalable y capaz de soportar a miles de usuarios de manera concurrente.

Debido a políticas internas del SAT, no era posible el uso de software libre que no hubiera sido aprobado con anterioridad y, por otro lado, las soluciones propietarias disponibles requerían de licencias con costo de millones de pesos a la vez de que el tiempo requerido para implementarlas dentro del proyecto era incierto. Cabe mencionar que la conclusión en tiempo del proyecto era algo crítico ya que en el contrato se había especificado que por cada día de retraso se descontaría un cierto porcentaje del pago final.

Dada esta situación se comisionó a un equipo, del cual formé parte, para investigar posibles soluciones que se pudieran desarrollar en tiempo. La

solución elegida se basó en desarrollar un sistema que leyera y manipulara directamente el formato OOXML sin usar bibliotecas de terceros. Esto haría posible que las plantillas en sí fueran documentos OOXML lo que facilitaría su creación y edición gracias a la posibilidad de usar Microsoft Word para ello. Esta solución no se había considerado previamente dado el reto técnico que manejar el formato OOXML directamente representaba, sobre todo considerando la complejidad que algunas plantillas requerían (por ejemplo, algunas plantillas deberían poder soportar la inserción de imágenes generadas dinámicamente). La solución fue implementada exitosamente y en tiempo.

En este reporte describo el desarrollo de esta solución el cual fue llevado a cabo por un equipo de tres personas y en el cual las partes fundamentales del mismo estuvieron a mi cargo. El primer capítulo contiene un breve repaso de las tecnologías principales usadas por el sistema. Posteriormente, el segundo capítulo, contiene una lista de los requerimientos, tanto funcionales como no funcionales, que se fijaron para este sistema. En el tercer capítulo, se describe la solución que fue elegida y explicaremos cómo ésta nos permitió cumplir con los principales requerimientos. En el cuarto capítulo se presenta la arquitectura del sistema incluyendo a los distintos servicios involucrados en la generación de documentos y mostraremos cómo ésta cumple con uno de los requerimientos no funcionales más importantes: la escalabilidad. Posteriormente, el quinto capítulo, contiene una descripción detallada de la implementación del *Servicio Generador de Documentos*, el cual es el servicio principal del sistema. Y finalmente, se presentarán las conclusiones de este trabajo.

# Capítulo 1

# Conceptos y Tecnologías Principales

En este capítulo se introducirán brevemente algunos conceptos y tecnologías que son necesarios para entender el resto de este reporte.

## 1.1. XML

XML [10] es un lenguaje de marcado estándar basado en texto diseñado para representar y almacenar datos de tal manera que estos puedan ser leídos tanto por humanos como por computadoras. Aunque su diseño se enfoca en documentos, es también usado para definir protocolos de intercambio de datos entre computadoras. Por ejemplo, SOAP (que se revisará más adelante) es un protocolo basado en XML cuyo objetivo es hacer posible la invocación de servicios en redes de computadoras. La información en un documento XML se almacena en una estructura de datos de árbol en donde cada nodo puede contener atributos y texto, u otros nodos como descendientes. Cada nodo es representado por una etiqueta como se muestra en el listado 1.1:

```xml
<?xml version="1.0"?>
<cuestionario>
  <pr sequencia="1">
    <pregunta>a?</pregunta>
    <respuesta>b</respuesta>
  </pr>
  <pr sequencia="2">
    <pregunta>c?</pregunta>
    <respuesta>d</respuesta>
  </pr>
</cuestionario>
```

Listado 1.1: XML

En este ejemplo, el nodo raíz es cuestionario. Este nodo tiene dos nodos hijo, ambos representados con la etiqueta pr, que a su vez contienen dos nodos hijos con distintas etiquetas. sequencia es el nombre de un atributo de la etiqueta pr.

### 1.1.1. DOM

Se le conoce como DOM *(Document Object Model)* a una representación de un documento XML con la cual se puede acceder al contenido del mismo usando un lenguaje de programación. La representación es independiente del lenguaje usado y es definida con clases y métodos que hacen posible acceder a toda la información contenida en un documento XML. Como es de esperarse, una de las principales clases es la clase Node que representa a una etiqueta o nodo en un documento XML. Esta clase tiene métodos como getChildNodes o getAttributes que hacen posible acceder a los nodos hijos y atributos del nodo respectivamente, lo que hace posible recorrer el árbol del documento.

### 1.1.2. Esquema XML

Aunque el estándar XML define que los datos de un XML se deben almacenar en una estructura de árbol, la estructura de este árbol en sí no está definida. Para definir a esta estructura se usa un esquema XML [10] el cual es a su vez un documento XML que contiene una especificación de la estructura de los documentos que cumplen con ese esquema. En un esquema XML se puede definir, por ejemplo, que la etiqueta o nodo cuestionario únicamente

puede tener como nodos hijos a cero o más nodos pr. En general, un esquema XML define a una clase de documentos XML con una estructura común. En el listado 1.2 se muestra un ejemplo de esquema XML que define una estructura compatible con el documento XML del listado 1.1.

```
<?xml version="1.0"?>
<xs:element name="cuestionario">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="pr">
        <xs:complexType>
          <xs:sequence>
            <xs:element name="pregunta" type="xs:string"/>
            <xs:element name="respuesta" type="xs:string"/>
          </xs:sequence>
          <xs:attribute name="secuencia" type="xs:positiveInteger"/>
        </xs:complexType>
      </xs:element>
    </xs:sequence>
  </xs:complexType>
</xs:element>
```

Listado 1.2: Esquema XML

### 1.1.3. XPath

XPath es un lenguaje usado para seleccionar nodos o valores en un documento XML. Una expresión XPath se evalúa sobre un documento XML dado y se obtiene un valor como resultado. Las expresiones XPath utilizan la estructura de árbol de los documentos XML para navegar hasta el nodo o valor que se desea seleccionar. A continuación se muestran algunos ejemplos sencillos de expresiones XPath junto con el valor que regresan al ser evaluados en el documento XML en el listado 1.1.

- /cuestionario/pr[1]/pregunta regresa a?
- /cuestionario/pr[2]/pregunta regresa c?
- /cuestionario/pr[1]/@secuencia regresa 1

## 1.2. SOA

SOA *(Service Oriented Architecture)* [4] es un tipo de arquitectura de software que se basa en el uso de servicios desplegados independientemente que de manera colectiva proveen la funcionalidad del sistema. Cada servicio provee una funcionalidad específica a través de métodos que son invocados remotamente por otros servicios o aplicaciones. Esta funcionalidad se expone normalmente a través de protocolos implementados sobre HTTP.

Una de las ventajas de esta arquitectura es que los servicios pueden ser desarrollados y mantenidos de manera independiente logrando así un bajo acoplamiento en el sistema. Por ejemplo, cada servicio puede ser actualizado con mejoras o correcciones sin requerir de cambios en los demás servicios, o bien, es posible implementar un esquema de alta disponibilidad (ej. usando balanceo de cargas) en un servicio dado incrementando el número de usuarios concurrentes que puede soportar, sin afectar al resto del sistema.

Otra ventaja es la posibilidad de usar distintas tecnologías y lenguajes en los clientes de manera independiente a las tecnologías bajo las cuales está desarrollado el servicio en sí. Esto es posible ya que los protocolos que exponen la funcionalidad de los servicios se encuentran estandarizados por lo que existen bibliotecas para muchos lenguajes que son compatibles entre sí.

## 1.3. SOAP

SOAP [10] es un protocolo para servicios web basado en XML y HTTP en donde es posible definir objetos que pueden ser usados como parámetros al invocar métodos del servicio. De esta forma es posible hacer que el uso de SOAP sea transparente en lenguajes orientados a objetos: para crear un servicio SOAP en estos lenguajes basta con crear una interfaz con los métodos del servicio y una clase que implemente esta interfaz. Las herramientas para SOAP en estos lenguajes se encargan de crear el servicio a partir de la interfaz sin que el desarrollador tenga que crearlos manualmente. Las clases usadas en la interfaz son convertidas en objetos SOAP de manera transparente para el desarrollador.

## 1.4. Patrón de Diseño Visitor

El patrón de diseño Visitor [7] es usado para implementar algoritmos que trabajan sobre gráficas separando al algoritmo que actúa sobre los nodos del algoritmo que las recorre.

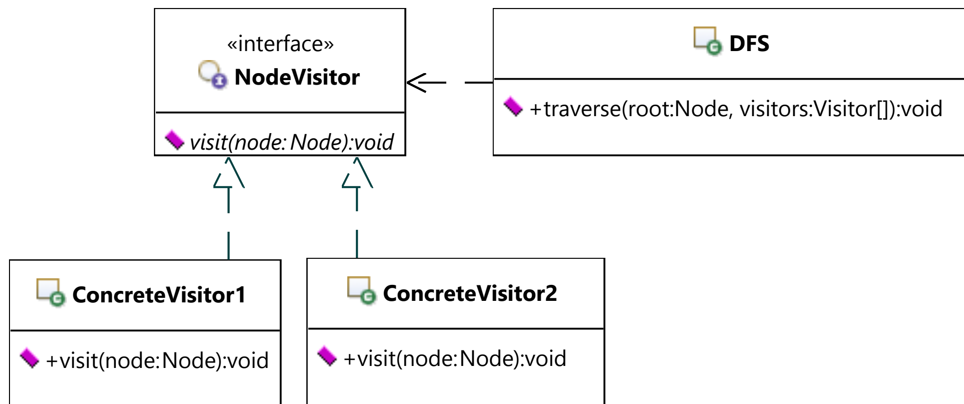


Figura 1.1: Patrón de diseño Visitor

En el diagrama de clases de la figura 1.1 se muestran las clases que conforman a dicho patrón. El algoritmo que actúa sobre los nodos es implementado en el método visit de las clases concretas. Por otro lado, otro algoritmo se encarga de recorrer la gráfica (por ejemplo, el algoritmo DFS) y de llamar al método visit de cada objeto por cada nodo visitado. De esta manera los algoritmos se separan de la estructura de la gráfica y de la manera de recorrerla. La principal ventaja de este patrón es que se vuelve fácil implementar nuevos algoritmos que actúan sobre la gráfica sin incrementar el número de recorridos que se hacen sobre ella.

# Capítulo 2

# Requerimientos

Como ya se mencionó, una gran variedad de casos de uso que el proyecto debía cubrir requerían de la generación de documentos a partir de datos obtenidos de bases de datos y servicios web. Aunado a esto, algunos de los casos de uso requerían que los documentos pudieran ser firmados digitalmente para certificar su autenticidad. Como un ejemplo de uno de estos casos de uso se encuentra la emisión de notificaciones: una notificación es el acto por el cual el SAT le da a conocer a una persona moral o física, de manera formal, una situación o hecho relacionado con sus obligaciones fiscales. Por ejemplo, el SAT puede emitir una notificación a una persona física por no presentar declaraciones. El documento que es entregado contiene una firma electrónica que autentifica al mismo. El destinatario puede entonces entrar al portal del SAT y verificar que la firma electrónica es auténtica y corresponde al documento en sus manos. Una vez concluido el proyecto, este tipo de documentos serían generados y firmados de manera automática.

Una presentación más detallada de los casos de uso que emplean la generación de documentos está fuera del alcance de este reporte ya que, el sólo describirlos, requiere de la introducción de conceptos de negocio particulares del SAT, los cuales en su mayoría son irrelevantes para el diseño e implementación de la solución.

Descrito el escenario anterior algunos de los requerimientos resultan evidentes, sin embargo, por claridad y completez se listan en las siguientes secciones.

## 2.1. Requerimientos Funcionales

- **Documentos generados de tipo DOCX y PDF**. Estos son los formatos soportados oficialmente por el SAT.
- **Plantillas fáciles de crear y editar**. La cantidad de documentos que se tenían que soportar y el tiempo que se contaba para ello hacían necesario que las plantillas pudieran ser creadas y mantenidas fácilmente.
- **Control preciso sobre el formato visual de los documentos generados**. Muchos documentos que se generarían serían documentos oficiales, por tal motivo era importante que el formato final fuera idéntico al formato de la plantilla.
- **Soporte de inserción de firma electrónica en los documentos generados**. La firma electrónica es una parte fundamental de la digitalización de los servicios del SAT por lo que era necesario soportarla.
- **Soporte de inserción de imágenes generadas dinámicamente**. Algunos casos de uso que requerían de inserción de códigos de barras los cuales son representados con imágenes.
- **Soporte de generación de tablas dentro del documento a partir de datos tabulares de longitud variable**. Esto era necesario para poder incluir listas de artículos de longitud variable en los documentos.

## 2.2. Requerimientos No Funcionales

- **Solución escalable y capaz de soportar miles de usuarios concurrentes**. El sistema debía soportar periodos de alta demanda en donde muchos usuarios acceden al servicio al mismo tiempo.
- **Tiempo de respuesta inmediato al generar un documento dado**. Existían casos de uso en los que el documento se generaba bajo demanda a través del sitio web del SAT por lo que una generación lenta o diferida resultaría en una mala experiencia de usuario.
- **Uso de software libre restringido a bibliotecas previamente aprobadas**. El SAT mantiene un control estricto del software que se ejecuta en sus sistemas, por tal motivo, cualquier biblioteca de software

que se deseara usar debía pasar primero por un proceso de aprobación. Dadas las dificultades que este proceso representaba, en el proyecto se había definido que únicamente se usarían bibliotecas previamente aprobadas.

- **Uso de arquitectura SOA (Service Oriented Architecture)**. Esta es una arquitectura muy usada actualmente en sistemas empresariales por lo que muchos proveedores de desarrollo están capacitados para implementarla. Teniendo el SAT una gran cantidad de proveedores, resulta lógico estandarizar esta arquitectura para lograr cierta compatibilidad entre los desarrollos de distintos proveedores.

- **Desarrollado en el lenguaje Java**. Aunque existen herramientas de Microsoft para generar documentos Word de manera automática, el uso de estas herramientas implicaría el uso de servidores Windows lo cual no se tenía contemplado en el proyecto. Esto habría significado un costo muy grande considerando licencias y el personal capacitado necesario para mantener el código y administrar servidores Windows.

# Capítulo 3

# Solución Propuesta

Una de las principales dificultades con las que nos enfrentamos al evaluar las opciones que teníamos fue el encontrar herramientas que pudieran controlar de manera precisa el formato visual de los documentos generados. Por ejemplo, Jasper Reports [8], una herramienta de software libre para la generación de reportes que se tenía aprobada en el proyecto, tenía la gran desventaja de que con él resultaba casi imposible lograr documentos (o reportes) con la apariencia que deseábamos, a la vez que la creación de las plantillas requería de una curva de aprendizaje elevada. Otra opción que se consideró, con las mismas dificultades, fue Apache FOP [6], una herramienta, también de software libre, que transforma documentos XSL-FO (un estándar basado en XML, lo que hace posible el uso de XSLT para convertir documentos XML a XSL-FO) en PDF, DOCX u otros formatos.

Ser capaces de generar documentos de formato OOXML a partir de plantillas también de formato OOXML era la solución ideal ya que se cumplirían perfectamente dos de los requerimientos que representaban el mayor reto gracias a la posibilidad de usar Microsoft Word:

- **Plantillas fáciles de crear y editar**.
- **Control preciso sobre el formato visual de los documentos generados**.

El formato OOXML fue diseñado considerando los casos de uso de captura de datos y generación automática de documentos. El único problema era que las herramientas para hacer esto se encuentran disponibles únicamente en sistemas Windows. Sin embargo, el formato OOXML [3], es un formato

estandarizado basado en XML y empaquetado en el formato ZIP por lo que potencialmente era posible implementar estas herramientas por nuestra cuenta. Cabe mencionar que esto es algo que normalmente no se hace en proyectos de desarrollo de componentes de negocio, como era el caso, debido al riesgo que ello implica. Riesgo justificado como lo descubrimos al encontrar un problema inesperado: ninguna biblioteca de manipulación de archivos ZIP disponible era compatible con Documentum [1] el cual es el sistema de almacenamiento de documentos usado por el SAT. Este problema fue solucionado desarrollando, de igual manera, las herramientas necesarias para manipular dicho formato directamente.

## 3.1. Office Open XML (OOXML)

Un documento OOXML o, lo que es lo mismo, archivo de extensión DOCX es simplemente un archivo de formato ZIP que contiene directorios, archivos XML y otros recursos como imágenes.

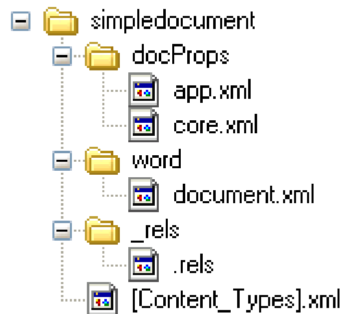


Figura 3.1: Estructura de archivos del formato Office Open XML.

La estructura de los directorios es en realidad arbitraria. La Figura 3.1 muestra la estructura de directorios que usa Microsoft Word por omisión. En realidad, lo importante en un paquete de Office Open XML, son las relaciones que los archivos tienen entre sí. Mientras se conserven estas relaciones la estructura de los directorios puede modificarse arbitrariamente. Las relaciones están especificadas en los archivos con extensión .rels. En la Figura 3.1 se muestra un archivo sin nombre con esta extensión el cual, al no tener nombre, es considerado la especificación de las relaciones del documento en sí. En el listado 3.1 se encuentra un ejemplo del contenido de este archivo.

```xml
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<Relationships xmlns="http://schemas.openxmlformats.org/package/2006/
    relationships">
  <Relationship Id="rId3"
    Type="http://schemas.openxmlformats.org/officeDocument/2006/
        relationships/extended-properties"
    Target="docProps/app.xml"/>

  <Relationship Id="rId2"
    Type="http://schemas.openxmlformats.org/package/2006/relationships/
        metadata/core-properties"
    Target="docProps/core.xml"/>

  <Relationship Id="rId1"
    Type="http://schemas.openxmlformats.org/officeDocument/2006/
        relationships/officeDocument"
    Target="word/document.xml"/>
</Relationships>
```

Listado 3.1: Contenido del archivo .rels

En este archivo se indican las rutas de los archivos principales. Por ejemplo, el contenido del documento se encuentra en el archivo word/document.xml.

### 3.1.1. Controles de contenido

Como ya se mencionó, el formato OOXML fue diseñado para soportar la creación y edición de plantillas. Esto se hace posible mediante el uso de *controles de contenido* los cuales son campos especiales que se definen en el contenido del documento. Microsoft Word soporta estos campos por lo que es posible crear y editar plantillas con él. En la figura 3.2 se observa cómo Microsoft Word muestra los controles de contenido. En el listado 3.2 se muestra el fragmento XML de este control como se encuentra en el archivo word/document.xml.

nombreORazonSocialContribuyente_97
**Nombre o Razón Social nombreORazonSocialContribuyente_97**
**RFC del Contribuyente rfcContribuyente_132**
**Domicilio Fiscal del Contribuyente domicilioFiscalContribuyente_17**

Figura 3.2: Control de Contenido como aparece en Microsoft Word.

```xml
<w:sdt>
  <w:sdtPr>
    <w:rPr>
      <w:rStyle w:val="NOSPELLCHECKING" />
      <w:b w:val="0" />
      <w:lang w:val="es-MX" />
    </w:rPr>
    <w:alias w:val="nombreORazonSocialContribuyente_97" />
    <w:tag w:val="2" />
    <w:id w:val="275206824" />
    <w:lock w:val="sdtContentLocked" />
    <w:placeholder>
      <w:docPart w:val="325EA29F5D1D4A73900AA03DFFCFA8B3" />
    </w:placeholder>
    <w:text />
  </w:sdtPr>
  <w:sdtContent>
    <w:r w:rsidR="00390150" w:rsidRPr="00390150">
      <w:rPr>
        <w:rStyle w:val="NOSPELLCHECKING" />
        <w:b w:val="0" />
        <w:lang w:val="es-MX" />
      </w:rPr>
      <w:t>nombreORazonSocialContribuyente_97</w:t>
    </w:r>
  </w:sdtContent>
</w:sdt>
```

Listado 3.2: Fragmento XML de un control de contenido.

Estos controles, al igual que cualquier elemento en un documento OOXML, pueden tener datos arbitrarios asociados mediante una relación definida en el archivo document.xml.rels a otro archivo dentro del paquete. En particular es posible asociar un control de contenido a una expresión de XPath el cual es un

lenguaje que permite seleccionar nodos o atributos contenidos en documentos XML. Por ejemplo, la expresión /A/B/Persona/@nombre, al ser evaluada en el XML de la figura 3.3, selecciona el texto Yazmin Gonzalez.

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<A>
  <B>
    <Persona nombre="'Yazmin Gonzalez"'/>
  </B>
</A>
```

Listado 3.3: XML de Datos

Por lo tanto, si contamos con documentos XML con una estructura definida, podemos generar documentos a partir de ellos evaluando las expresiones de XPath asociadas a cada control de contenido en la plantilla. Esta es una técnica común para generar documentos OOXML por lo que, aunque Microsoft Word no soporta este tipo de asociaciones entre controles de contenido y expresiones de XPath, existen herramientas disponibles que sí lo hacen. En particular, el SAT cuenta con una licencia de la herramienta xPresso para Microsoft Word [2] la cual sirve para dicho propósito. Cabe notar que el uso de esta herramienta es posible ya que se usa únicamente para crear plantillas y no para generar los documentos. Por ello no pertenece propiamente al Generador de Documentos y no necesita cumplir el requerimiento de estar desarrollado en Java. Recordemos que uno de los requerimientos para el Generador de Documentos era la capacidad de formar parte de una arquitectura de servicios web desarrollada con Java.

La herramienta xPresso para Microsoft Word hace uso del archivo word/_rels/document.xml.rels para asociar un nuevo archivo XML, customXml/item1.xml, en donde se hacen persistentes las expresiones XPath asociadas a los controles de contenido extendiendo, así, la funcionalidad del formato OOXML. En el listado 3.4 se muestra cómo se indica la relación con este archivo.

```
<?xml version="1.0" encoding="UTF-8"?>
<Relationships xmlns="http://schemas.openxmlformats.org/package/2006/
    relationships">
  ...
  <Relationship Id="rId2"
    Target="../customXml/item1.xml"
    Type="http://schemas.openxmlformats.org/officeDocument/2006/
        relationships/customXml"/>
  ...
</Relationships>
```

Listado 3.4: word/_rels/document.xml.rels

Finalmente en el archivo customXml/item1.xml se definen una serie de mapeos dentro del nodo XMLMapping contenido a su vez en el nodo VarDefTree. Esta es la expresión XPath que debe evaluarse en el XML de Datos recibido. En el listado 3.5 se puede ver un ejemplo en donde se muestra la expresión que corresponde al control de contenido que vimos anteriormente.

```
<xPressoModel xmlns="http://www.docscience.com/xPressoWord3.0">
  <VarDefTree>
    <Id>2</Id>
    <Name>nombreORazonSocialContribuyente_97</Name>
    <IsValid>true</IsValid>
    <ParentId>1</ParentId>
    <VarDefType>Simple</VarDefType>
    <XMLMapping>/A/B/Persona/@nombre</XMLMapping>
    <IsUndefined>false</IsUndefined>
    <IsExpanded>True</IsExpanded>
  </VarDefTree>
</xPressoModel>
```

Listado 3.5: customXml/item1.xml

Una vez que evaluamos esta expresión XPath en el XML de Datos del listado 3.3 obtenemos el valor final que se deberá reemplazar en el control de contenido. En la figura 3.6 se muestra el fragmento XML de un control de contenido en un documento generado.

```xml
<w:sdt>
  <w:sdtPr>
    <w:rPr>
      <w:rStyle w:val="NOSPELLCHECKING" />
      <w:b w:val="0" />
      <w:lang w:val="es-MX" />
    </w:rPr>
    <w:alias w:val="nombreORazonSocialContribuyente_97" />
    <w:tag w:val="2" />
    <w:id w:val="275206824" />
    <w:lock w:val="sdtContentLocked" />
    <w:placeholder>
      <w:docPart w:val="325EA29F5D1D4A73900AA03DFFCFA8B3" />
    </w:placeholder>
    <w:text />
  </w:sdtPr>
  <w:sdtContent>
    <w:r w:rsidR="00390150" w:rsidRPr="00390150">
      <w:rPr>
        <w:rStyle w:val="NOSPELLCHECKING" />
        <w:b w:val="0" />
        <w:lang w:val="es-MX" />
      </w:rPr>
      <w:t>Yazmin Gonzalez Bayardo</w:t>
    </w:r>
  </w:sdtContent>
</w:sdt>
```

Listado 3.6: Fragmento XML de un control de contenido.

Como se puede ver, el proceso es relativamente sencillo. Sin embargo, existen también controles de contenido compuestos en donde la expresión de XPath asociada selecciona a más de un valor. En este caso el control de contenido se debe encontrar dentro de una tabla. Los valores generados se insertan en nuevas filas creadas dentro de la tabla. La implementación de esta funcionalidad no se cubre en este reporte ya que representa únicamente detalles técnicos una vez cubierto el caso simple.

### 3.1.2. Cumplimiento de los requerimientos funcionales

A continuación se describe cómo esta solución cumple con los requerimientos funcionales.

- **Documentos generados de tipo DOCX y PDF**. Evidentemente la solución soporta el formato DOCX. Para generar documentos PDF se utilizó un sistema ya existente en el SAT. En el quinto capítulo se describe la integración del Servicio Generador de Documentos con este sistema.

- **Plantillas fáciles de crear y editar**. En esta solución las plantillas son en sí documentos DOCX que se pueden editar usando Microsoft Word como cualquier otro documento.

- **Control preciso sobre el formato visual de los documentos generados**. Suponiendo un diseño correcto de la plantilla, los documentos generados mantendrían el aspecto visual de la plantilla ya que nuestra solución únicamente inserta texto o imágenes en lugares predefinidos de una forma predecible. Aunque existe el riesgo de que una cadena de texto muy larga desalinee el documento, este es un problema que se puede manejar al momento de diseñar la plantilla. Cada plantilla soportaría un rango de longitud en los controles de contenido de acuerdo a las reglas de negocio propias del caso de uso correspondiente.

- **Soporte de inserción de firma electrónica en los documentos generados**. Este requerimiento en realidad se obtiene automáticamente ya que la firma electrónica es únicamente una cadena de texto que se debe incluir en el documento generado.

- **Soporte de inserción de imágenes generadas dinámicamente**. Los controles de contenido también pueden contener imágenes por lo que es posible insertar imágenes de la misma manera en que se inserta texto. Únicamente es necesario agregar la imagen al documento DOCX que, como se verá más adelante, es simplemente un archivo de formato ZIP en el que se pueden agregar nuevos archivos.

- **Soporte de generación de tablas dentro del documento a partir de datos tabulares de longitud variable**. Esto se cumple soportando controles de contenido compuestos los cuáles no se cubren en este reporte.

# Capítulo 4

# Arquitectura

El proyecto SDMA fue desarrollado sobre una arquitectura tipo SOA. Esto quiere decir que la mayoría de los componentes del sistema que encapsulaban lógica de negocio se desarrollaron como servicios que exponían su funcionalidad a través de SOAP sobre HTTP. El Generador de Documentos no fue la excepción por lo que se desarrolló como un servicio web. Para entender cómo fue diseñado este servicio recordemos que, por un lado, la generación de documentos se basa en controles de contenido que tienen asociadas expresiones XPath, las cuales son evaluadas sobre un documento XML con los datos que se van a insertar y que, por otro lado, los datos a insertar se habrían de extraer de distintas fuentes como bases de datos u otros servicios web. Por lo tanto, si contábamos con un servicio que generara documentos a partir de una plantilla y un documento XML, lo único que haría falta sería entonces un servicio que obtuviera estos datos y generara un documento XML a partir de ellos. Este componente fue a su vez diseñado como un servicio web y fue llamado *Servicio Integrador*. Al documento XML con los datos que este servicio integra se le nombró, sin mucha creatividad, *XML de Datos*.

En realidad existen muchos *Servicios Integradores*. Cada uno se especializa en un tipo de XML de Datos con una estructura definida por un esquema XML y encapsula una lógica de acceso a datos particular a un caso de uso. Por ejemplo, supongamos que los datos necesarios para generar una notificación de incumplimiento de obligaciones fiscales de una persona física se encuentran disponibles en servicio web, A, y de una base de datos, B. El servicio integrador encargado de este caso de uso sería entonces responsable de conectarse al servicio web A y la base de datos B, y obtener los datos necesarios de ellos. Los datos serían entonces integrados en un *XML de Datos* el cual se pasaría

al *Servicio Generador* junto con la plantilla correspondiente.

Podemos ahora describir el proceso general para generar un documento:

1. La aplicación que requiere generar un documento invoca al Servicio Integrador pasando parámetros específicos del integrador de datos en cuestión. Por ejemplo, un servicio integrador asociado a una plantilla que contiene datos de una persona, recibiría como parámetro un identificador de dicha persona, por ejemplo CURP, con el cual se buscaría a la persona en una base de datos.

2. El Servicio Integrador realiza las consultas a otros servicios o bases de datos necesarias para obtener los datos que necesita y, con ellos, genera un documento XML, el XML de Datos, el cual le es regresado a la aplicación.

3. La aplicación invoca al Servicio Generador de Documentos pasando como parámetros el identificador de la plantilla y el XML de Datos.

4. El Servicio Generador de Documentos obtiene la plantilla del repositorio de documentos, extrae las expresiones XPath asociadas a los controles de contenido dentro de la plantilla, evalúa estas expresiones contra el XML de Datos y los valores resultantes se insertan en los controles de contenido generando así un nuevo documento el cual se guarda en el repositorio de documentos.

5. La aplicación solicita el documento generado al repositorio de documentos.

Esta interacción se muestra en el diagrama de secuencia de la figura 4.1. Cabe notar que las llamadas en el diagrama son invocaciones remotas ya que cada aplicación se encuentra albergada en un servidor distinto.

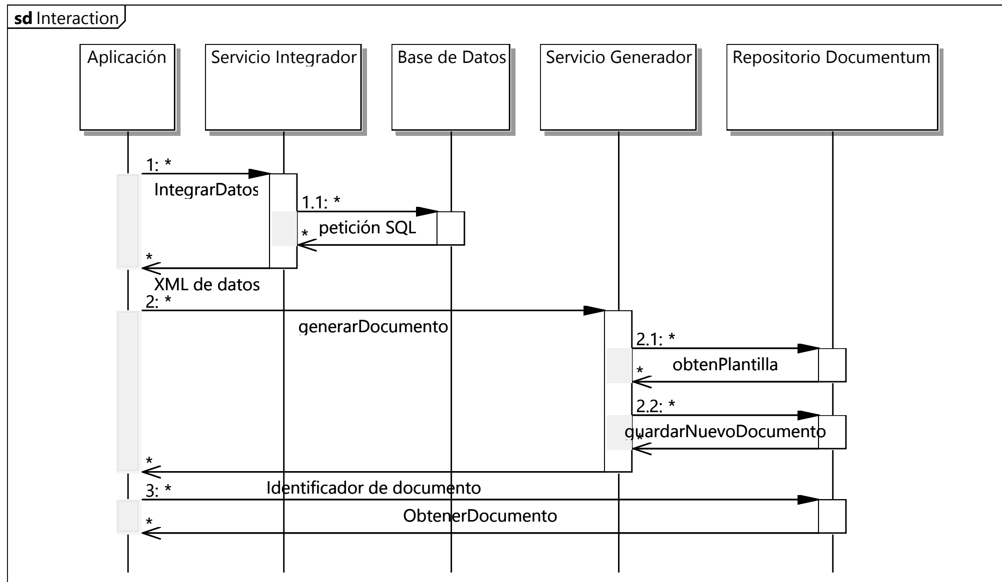


Figura 4.1: Generación de Documento

Cada Servicio Integrador fue desarrollado por un equipo distinto, cuyos integrantes estaban familiarizados con los datos que cada plantilla necesitaba pero que ignoraban completamente la manera interna de funcionar del Servicio Generador de Documentos (nótese que esto indica que existe un bajo acoplamiento). El Servicio Generador de Documentos, en cambio, sí es único. Si este hecho hiciera pensar al lector que pudiera existir un efecto de cuello de botella, hay que recordar que otra ventaja de esta arquitectura es que podemos escalar la capacidad de cada servicio fácilmente, por ejemplo, creando varias instancias del Servicio Generador de Documentos detrás de un balanceador de cargas como se muestra en la figura 4.2.

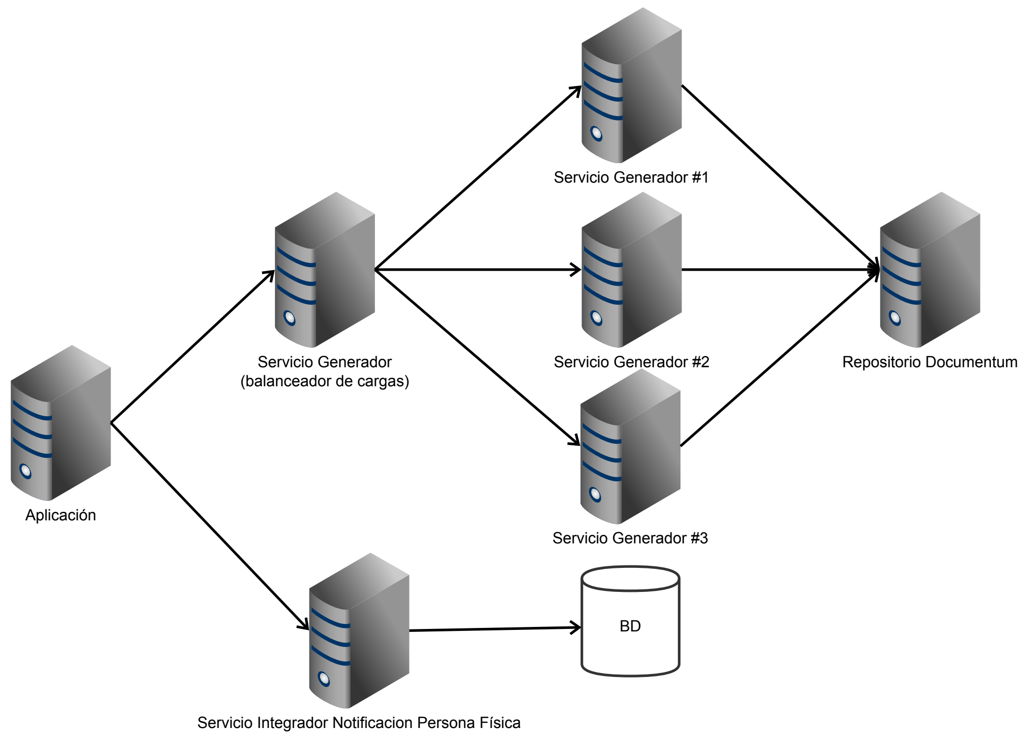


Figura 4.2: Balanceo de cargas

Dado que los Servicios Integradores son en realidad muy simples, en este reporte nos enfocaremos en detallar al Servicio Generador de Documentos.

# Capítulo 5

# Servicio Generador de Documentos

Como ya se mencionó, uno de los requerimientos para el Generador de Documentos era que este debía apegarse a una arquitectura SOA y debía estar desarrollado en el lenguaje Java. El protocolo elegido para ello, y el cual es la elección por omisión en la arquitectura SOA, fue SOAP.

El servicio se implementó en una sola aplicación web de Java. Esto quiere decir que todas las clases pertenecientes al Generador de Documentos, y las cuales son descritas en esta sección, se encuentran contenidas en un solo archivo WAR. Un archivo WAR es simplemente un archivo de formato ZIP que empaqueta todas las clases de una aplicación web. Algunas clases, conocidas como *servlets*, se encargan de procesar solicitudes HTTP. El soporte para el protocolo SOAP se obtuvo a través de la biblioteca Apache Axis [5], la cual implementa justamente un *servlet* capaz de procesar solicitudes SOAP sobre HTTP.

Esta aplicación web se despliega en un contenedor web de Java (IBM WebSphere en este caso) el cual se encarga de albergar a una o más aplicaciones web como lo es el Generador de Documentos. El contenedor es responsable de recibir la petición HTTP y redirigirla al *servlet* que ha sido configurado para responder a ella. Detrás del Servlet de Axis que procesa en sí la petición SOAP se encuentran, finalmente, las clases propias del Generador de Documentos. En primera instancia se encuentra la clase que expone los métodos del servicio, GenerarDocumentoService.

A continuación se lista la secuencia de pasos que se llevan a cabo para procesar una solicitud de generación de documentos.

1. La aplicación que desea generar un documento crea una solicitud HTTP al servidor donde se encuentra el contenedor de aplicaciones. Una vez establecida la sesión HTTP, la aplicación envía un documento XML usando el protocolo SOAP con una invocación a un método del servicio, por ejemplo, crearDocumentoDOCX.

2. El contenedor web recibe la petición HTTP y la redirige al Servlet de Axis.

3. El Servlet de Axis interpreta el XML enviado por la aplicación usando el protocolo SOAP y redirige la llamada al método crearDocumentoDOCX de la clase TransformarDocumentoWebService.

Esta interacción se muestra en la figura 5.1.

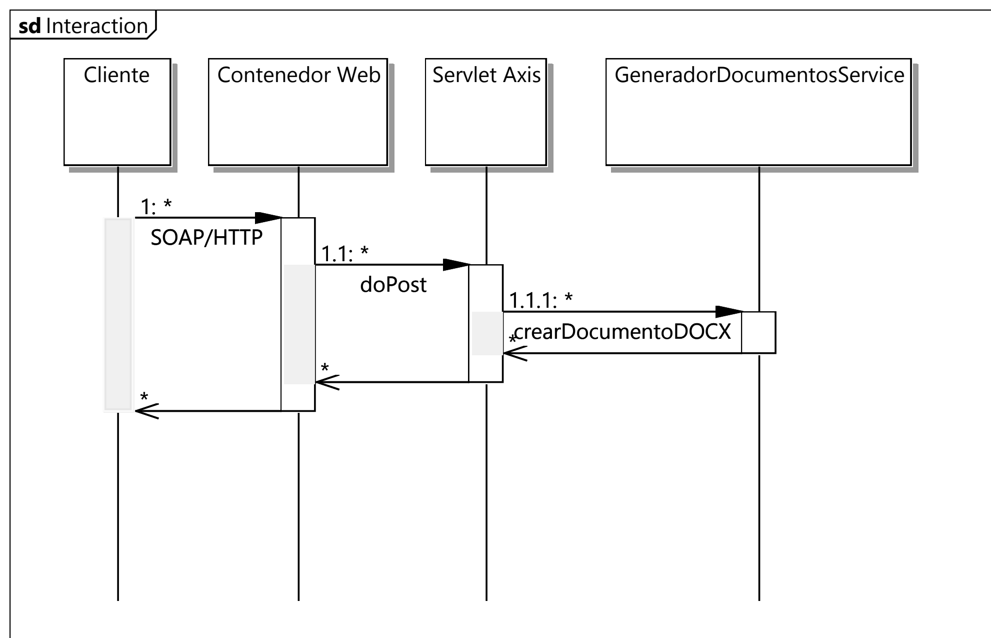


Figura 5.1: Invocación al método crearDocumentoDOCX

Habiendo descrito, a grandes rasgos, la arquitectura del sistema y cómo se procesan las peticiones al servicio web, nos enfocaremos ahora en revisar las clases que realizan en sí la generación de documentos. En las siguientes

secciones mostraremos las clases principales del servicio y las técnicas usadas para hacer eficiente la generación de documentos. Estas clases se muestran en la figura 5.2.

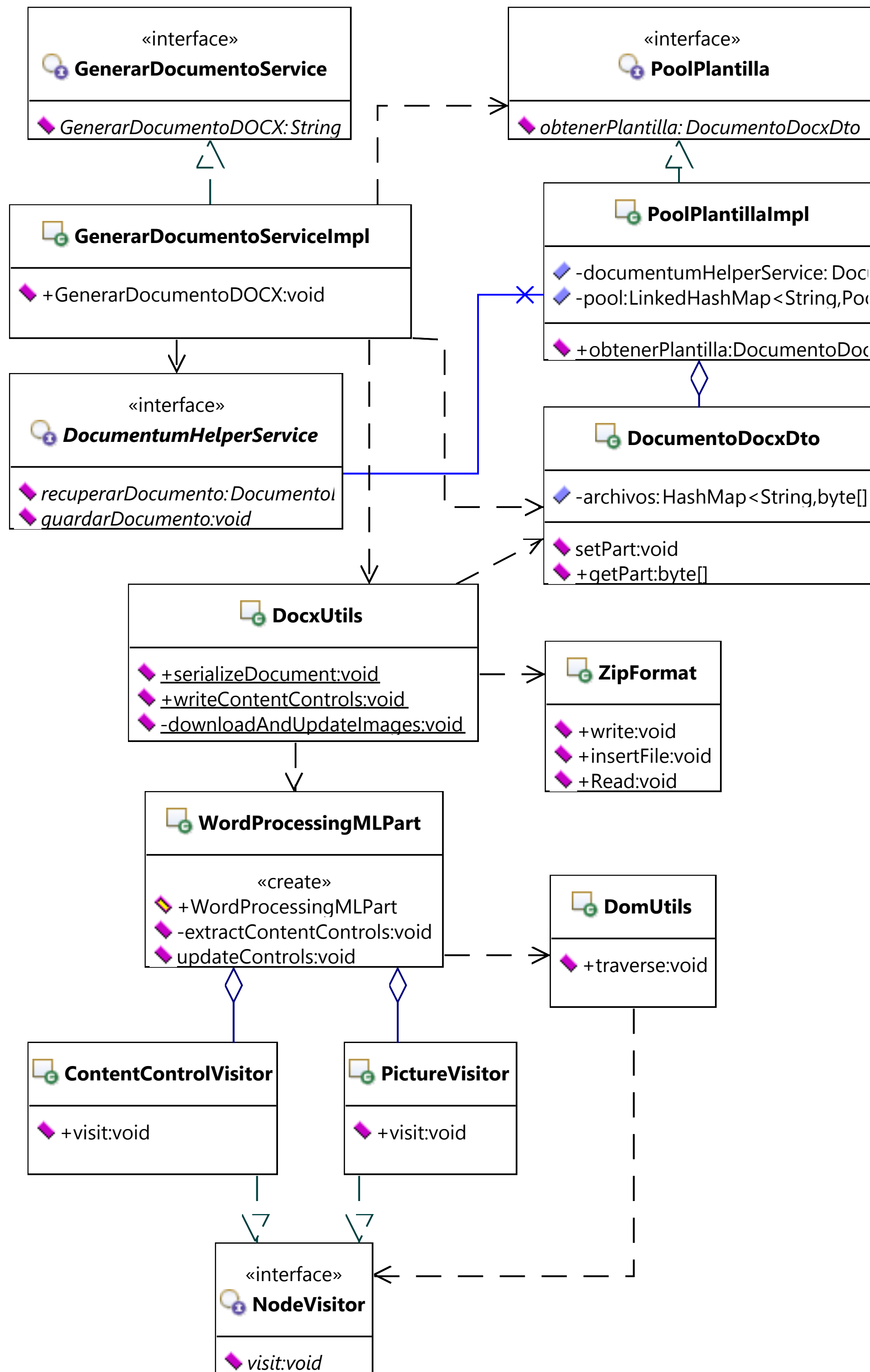


Figura 5.2: Clases Principales

## 5.1. Servicio de Documentum

DocumentumHelperService es una interfaz que representa al servicio de Documentum [1]. Documentum es una plataforma utilizada para almacenar y gestionar contenidos. Aunque tiene muchas funcionalidades, el Generador de Documentos únicamente lo usa para almacenar y obtener documentos. Al llamar métodos de esta interfaz realmente se están invocando los métodos de un servicio web a través de SOAP. Al tratarse de invocaciones remotas, las llamadas a los métodos en esta interfaz tienen un alto costo por lo que es importante restringir su frecuencia.

Tanto las plantillas como los documentos generados persisten mediante este servicio. Al recibir una petición, el Generador de Documentos recibe el identificador de la plantilla del documento que se desea generar. Este identificador se usa para obtener la plantilla del servicio de Documentum. Una vez que el nuevo documento ha sido generado, éste persiste usando el mismo servicio. Finalmente, el Generador de Documentos regresa este identificador al cliente.

Figura 5.3: DocumentumHelperService

## 5.2. DocumentoDocxDto

DocumentoDocxDto es una clase cuya única función es almacenar a un documento DOCX, ya sea una plantilla o un documento generado. Recordemos que un archivo DOCX es en realidad un archivo de formato ZIP con varios archivos contenidos. Por este motivo la clase DocumentoDocxDto realmente almacena archivos representados con un arreglo de bytes y una ruta que corresponde a la ruta del archivo en el paquete (incluyendo su nombre).

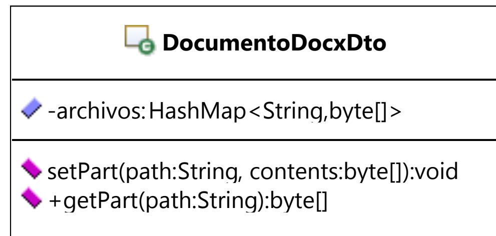


Figura 5.4: DocumentoDocxDto

Como puede verse en la figura 5.4 los archivos se guardan en una estructura Hash Map con lo que se asegura un tiempo constante al guardar y obtener archivos o partes (el término "parte" es usado en la especificación del formato OOXML para referirse a los documentos XML contenidos).

## 5.3. Pool de Plantillas

La clase PoolPlantilla resulta de una optimización que fue necesaria hacer. Como ya se mencionó, las llamadas al servicio de Documentum tienen un alto costo al tratarse de un servicio remoto. Por otro lado, es un requerimiento que la generación de documentos sea inmediata (menor a un segundo). Por tal motivo, se decidió mantener un número definido de plantillas localmente para evitar llamadas innecesarias al servicio de Documentum. A este tipo de patrón se le conoce comúnmente como *cache*. Este término posiblemente era más apropiado que el término *pool*, el cual se usa para referirse simplemente a un conjunto de recursos disponibles.

Esta optimización supone que la probabilidad de que se requiera una plantilla usada recientemente es mayor que la probabilidad de que se requiera una plantilla arbitraria. Aunque las ventajas son obvias, existen dos complicaciones que se deben atender:

1. Las plantillas pueden quedar desactualizadas si se modifican en el servidor mientras éstas existen en el *pool*. La solución a este problema es mantener simplemente un tiempo de expiración tras del cual, aunque la plantilla se encuentre en el *pool*, se forzará una petición al servidor de Documentum.

2. El número final de plantillas que existirían se desconocía por lo que era preferible limitar el número de plantillas que podían existir localmente en un momento dado, dando prioridad a las plantillas solicitadas más recientemente.

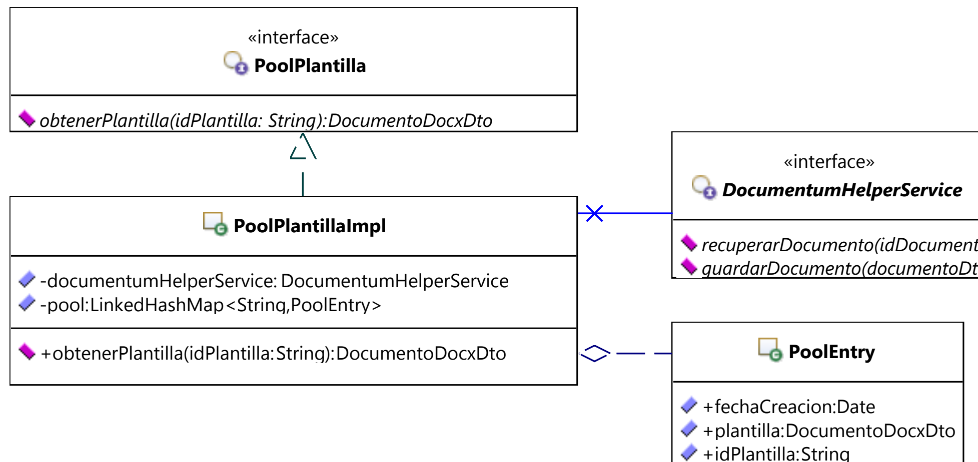


Figura 5.5: PoolPlantilla

Notemos en la figura 5.5 que la clase PoolPlantilla tiene una referencia al servicio de Documentum de donde se obtienen las plantillas.

Notemos también el uso de la estructura de datos LinkedHashMap. Esta estructura es un Hash Map en donde las entradas se encuentran ligadas como en la estructura de datos Lista Ligada. Esto nos da las ventajas de ambas estructuras en una sola: es posible acceder, eliminar e insertar valores en tiempo constante dada su llave (que en este caso es un identificador de plantilla) y, al mismo tiempo, se cuenta con un orden definido sobre las entradas. Este orden es usado para mantener una lista con las plantillas a las que se ha accedido más recientemente. Cada vez que una plantilla es solicitada, esta se reinserta al final de la lista. De esta forma, cuando se ha alcanzado la capacidad máxima permitida de la lista y necesitamos guardar una nueva plantilla, simplemente eliminamos la entrada en la cabeza de la lista la cual tenemos garantizada que es la plantilla a la que se accedió menos recientemente. Esto se observa en el algoritmo 1.

**Algoritmo 1** Obtención de la plantilla

```
function ObtenerPlantilla(idPlantilla)
    if pool.Contains(idPlantilla) then
        entry ← pool.Remove(idPlantilla)
        if entry.FechaCreacion > NOW − tiempoExpiracion then
            pool.Put(idPlantilla, entry)                ▷ Insertamos al final
            return entry.plantilla
        end if
    end if
    entry ← new()
    entry.Plantilla ← dhs.RecuperarDocumento(idPlantilla)
    entry.FechaCreacion ← NOW
    if pool está lleno then
        pool.RemoveFirst()
    end if
    pool.Put(idPlantilla, entry)                        ▷ Insertamos al final
    return entry.plantilla
end function
```

Todas las llamadas a los métodos de LinkedHashMap que usamos aquí (Contains, Remove, RemoveFirst [1] y Put) son de tiempo constante, por lo que la complejidad de este algoritmo es constante. Como un hecho interesante, observemos que el uso de LinkedList y HashMap por separado no nos daría el mismo resultado ya que, al momento de remover un elemento de la lista necesitaríamos forzosamente recorrer la lista para encontrar el elemento.

## 5.4. Manipulación de Documentos

En la figura 5.6 se muestra un diagrama de clases con las clases utilizadas para manipular el formato OOXML y generar documentos.

[1] Este método en realidad no existe en la clase LinkedHashMap, sin embargo, su implementación es trivial ya que podemos obtener el primer elemento iterando sobre la colección y deteniéndonos en la primera iteración. Una vez hecho esto solo es necesario llamar al método remove.

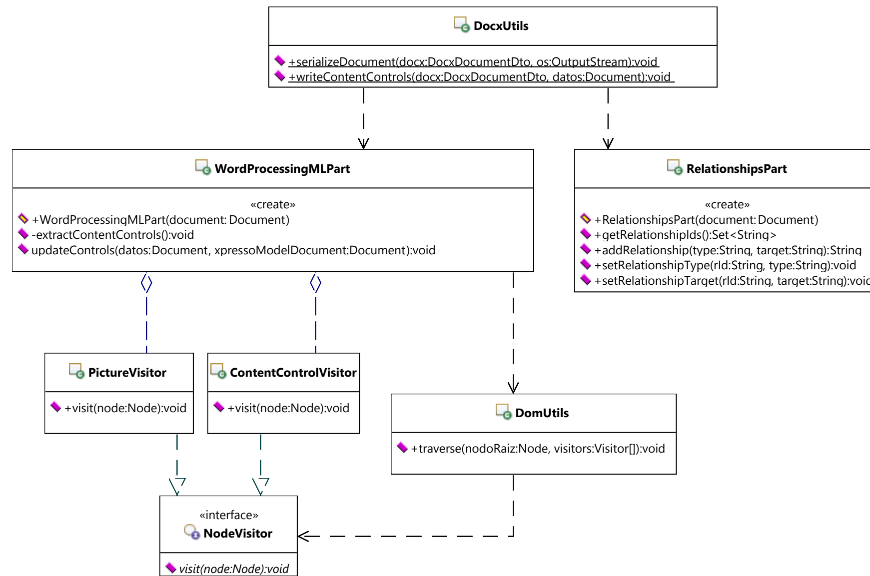


Figura 5.6: Manipulacion del formato OOXML

### 5.4.1. La clase DocxUtils

DocxUtils expone el método writeContentControls que recibe un documento DOCX (la plantilla) representado con una instancia de la clase DocxDocumentoDto y el DOM del XML de Datos representado por la clase org.w3c.dom.Document. El DOM de un XML es simplemente una representación con objetos Java de todos los nodos (representados por la clase org.w3c.dom.Node) que existen en el XML. Esta representación es navegable desde el código ya que los nodos exponen métodos como getChildNodes y getParentNode que regresan los nodos hijos y el nodo padre, respectivamente. La construcción del DOM a partir del XML se hace previamente con un *parser* de XML. Cabe notar que la construcción de un DOM implica necesariamente cargar en memoria una representación completa del XML en cuestión, por ello es necesario tener cuidado con el tamaño de los documentos que se leen. Esto representa un posible problema de seguridad en el sistema ya que no se implementaron este tipo de verificaciones. Sin embargo, de acuerdo a las reglas de negocio conocidas por el equipo, ninguno de los documentos XML procesados podían provenir de usuarios.

Las funciones del método writeContentControls son las siguientes:

1. Extraer los documentos word/document.xml, word/rels/document.xml.rel y customXml/item1.xml, y encapsularlos en instancias de las clases WordProcessingMLPart, RelationshipsPart y org.w3c.dom.Document, respectivamente.

2. Descargar imágenes referenciadas por los controles de contenido [2].

3. **Invocar la actualización de los controles de contenido llamando al método updateControls de WordProcessingMLPart.**

4. Actualizar la instancia de la clase DocxDocumentoDto recibida como parámetro con las nuevas imágenes y el documento word/document.xml actualizado.

La interacción completa se muestra en la figura 5.7.

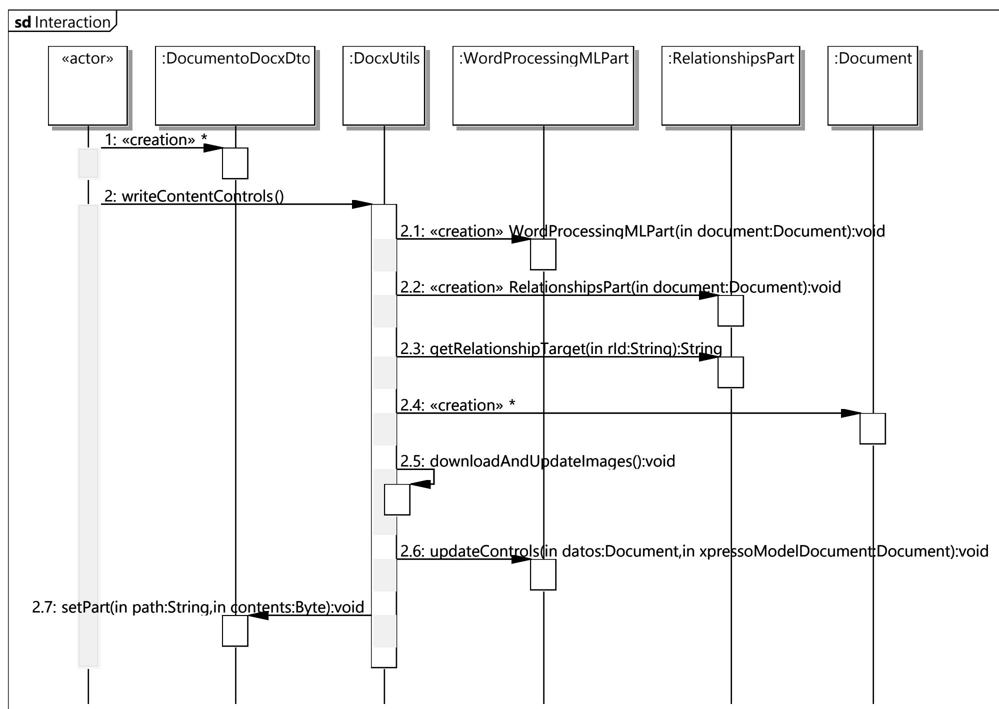


Figura 5.7: Interacciones de DocxUtils

[2]La inserción de imágenes no se cubre en este reporte.

### 5.4.2. Uso del patrón de diseño Visitor

La clase WordProcessingMLPart se usa para representar al documento word/document.xml y es la principal clase en la generación de documentos. Esta clase tiene conocimiento de WordprocessingML el cual es el dialecto de XML usado para representar el contenido de un documento OOXML. Para manipular este formato, esta clase mantiene una variable con el DOM del documento word/document.xml sobre el cual se hace un recorrido en profundidad (DFS) para extraer los nodos que se van actualizar (ej. controles de contenido e imágenes) usando el patrón de diseño Visitor. En nuestra implementación del patrón Visitor, la gráfica es un árbol de nodos, en donde cada nodo representa a un elemento XML del documento word/document.xml y el algoritmo que actúa sobre los nodos identifica a los nodos y realiza transformaciones en ellos para insertar texto o imágenes.

Para entender la utilidad del patrón Visitor en nuestro problema, consideremos el caso en el que se desean extraer controles de contenido del árbol que representa al documento word/document.xml. Si implementamos un algoritmo, extraerControles, que recibe al nodo raíz, estaríamos combinando al algoritmo de extracción con la lógica de acceso a la gráfica. El problema de esto es que si posteriormente requerimos de un nuevo algoritmo para extraer, por ejemplo, imágenes, tendríamos que crear un nuevo método, extraerImagenes, que recorrería el árbol de manera similar y por separado, resultando en dos recorridos sobre la gráfica. Si usamos el patrón Visitor, en cambio, podemos asegurar que siempre haremos un solo recorrido en la gráfica sin importar el número de algoritmos que trabajen sobre ella.

En la figura 5.8 se muestra nuestra implementación del patrón Visitor el cual es usado para extraer los controles de contenido e imágenes que se van a modificar en el documento.

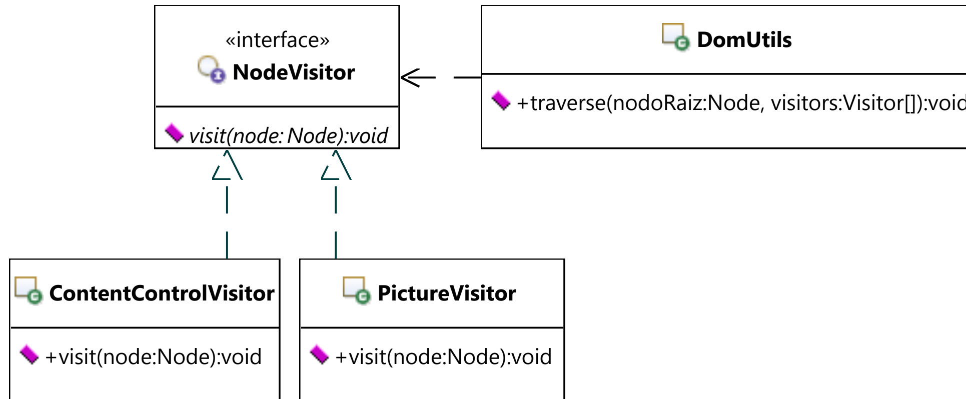


Figura 5.8: Patrón de diseño Visitor

### 5.4.3. Actualización de Controles de Contenido

Una vez que se han extraído los controles de contenido del documento, el siguiente paso es actualizar sus valores. Para ello, como ya se ha explicado, se usa el XML de Datos y la expresión de XPath asociada a cada control de contenido.

El DOM del XML de Datos se pasa como parámetro al método updateContentControls al igual que el DOM del documento XML que contiene las expresiones XPath. Cada control de contenido tiene un alias. Para encontrar la expresión XPath asociada, únicamente tenemos que buscar este alias en el documento XML (ver listado 5.1).

```xml
<xPressoModel xmlns="http://www.docscience.com/xPressoWord3.0">
  <VarDefTree>
    <Id>2</Id>
    <Name>nombreORazonSocialContribuyente_97</Name>
    <IsValid>true</IsValid>
    <ParentId>1</ParentId>
    <VarDefType>Simple</VarDefType>
    <XMLMapping>/A/B/Persona/@nombre</XMLMapping>
    <IsUndefined>false</IsUndefined>
    <IsExpanded>True</IsExpanded>
  </VarDefTree>
</xPressoModel>
```

Listado 5.1: customXml/item1.xml

Una vez obtenida la expresión XPath en el nodo XMLMapping, únicamente tenemos que evaluar la expresión en el DOM del XML de Datos y reemplazar el texto del control de contenido con el valor obtenido.

La generación de documentos también soporta controles de contenido compuestos. Estos controles de contenido representan un conjunto de datos que se despliegan en una tabla. Aunque las transformaciones en el árbol son más complejas, como en el caso de los controles de contenido simples, los principios son los mismos por lo que no detallaremos este proceso en el reporte.

## 5.5. Manipulación del formato ZIP

Un problema inesperado que encontramos fue el hecho de que los documentos DOCX generados por nuestro sistema no podían ser convertidos a PDF por Documentum. La conversión de formato no presentaba ningún problema si los documentos eran creados por Microsoft Word y, por otro lado, los documentos generados podían abrirse y editarse con Microsoft Word sin problemas. Después de una larga investigación descubrimos que la causa eran pequeñas diferencias en el formato ZIP con el que se empaquetaban. Documentum solo podía descomprimir un paquete ZIP si era creado por Microsoft Word o Windows. Cualquier otra biblioteca o herramienta para manejar el formato ZIP que probamos producía archivos incompatibles con Documentum. Este problema fue reportado al equipo de soporte de Documentum pero no se nos dio ninguna solución. Siendo la generación de documentos PDF un requerimiento esencial, era necesario encontrar una solución a este problema. Por tal motivo, tuvimos que desarrollar clases que manipularan el formato ZIP directamente.

El formato ZIP es un formato muy bien documentado. Sin embargo, existen campos que cada aplicación puede usar de manera arbitraria. La única diferencia que encontramos con los archivos ZIP, generados por nuestro sistema, se encontraba en estos campos. Desafortunadamente, no logramos encontrar información de cómo Microsoft Word los utiliza. Documentum, al parecer, esperaba cierta información en estos campos que no encontraba en nuestros archivos. La solución fue preservar en general todos los campos del archivo ZIP leído y agregar soporte para insertar archivos.

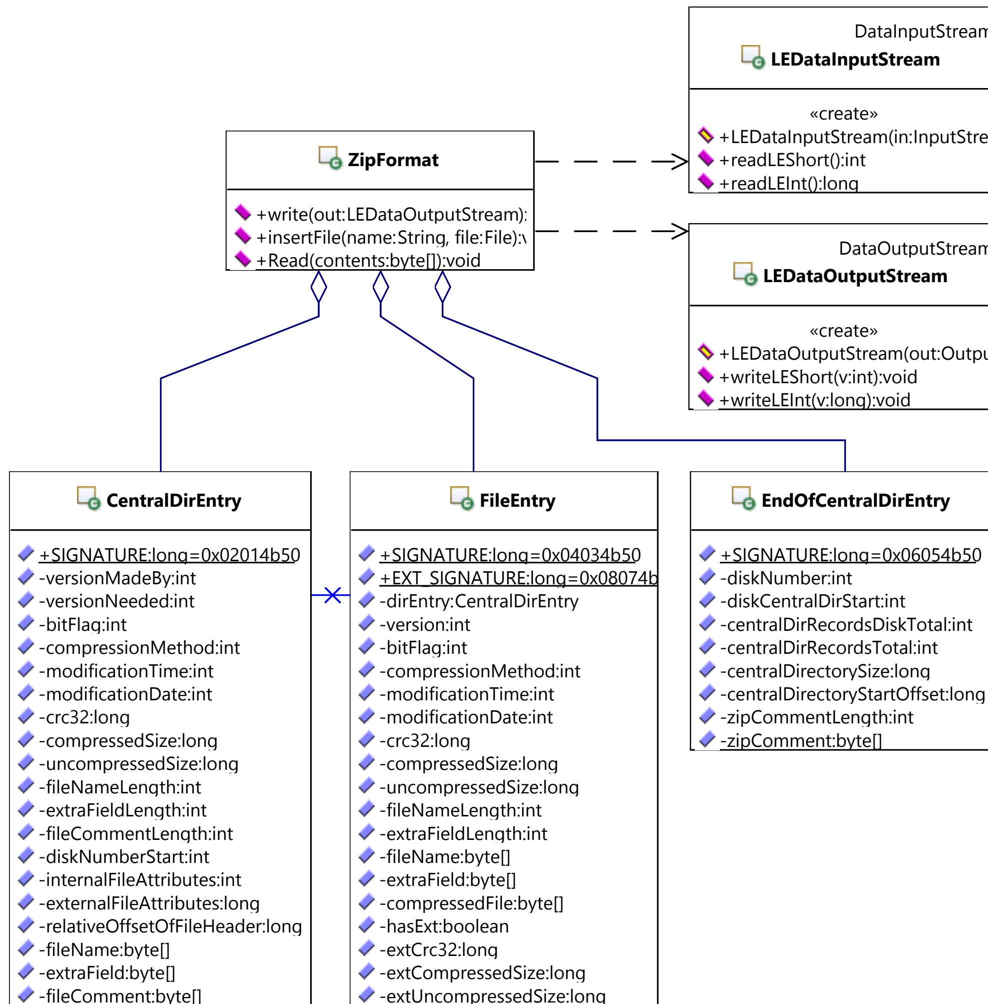


Figura 5.9: Clases para manipular el formato ZIP

Nuestra estrategia se basó en crear clases que representaran los elementos del formato ZIP como se encuentran en la especificación [9]. Tales clases únicamente mantienen la información que se encuentra en el formato de manera que se pueda acceder a ella desde Java haciendo posible actualizar al paquete ZIP (por ejemplo, para insertar archivos).

Esencialmente, existen dos secciones en un archivo ZIP: el directorio central, el cual mantiene información de la estructura de los directorios

y archivos contenidos, y la lista de los archivos en sí. Para insertar un archivo en el paquete ZIP, únicamente debemos crear una nueva entrada al directorio central, usando la clase CentralDirEntry, y otra en la lista de archivos, usando la clase FileEntry. El archivo se puede comprimir usando la clase java.util.zip.Deflater la cual usa la biblioteca de compresión de archivos ZLIB que es la implementación de compresión usada en los archivos ZIP. El código se muestra en el listado 5.2.

```
public class ZipFormat {

  ...

  // Inserta un archivo en el paquete ZIP
  public void insertFile(String fileName, byte[] data) throws IOException {

    FileEntry fileEntry = fileEntryMap.get(fileName);
    CentralDirEntry dirEntry = centralDirEntryMap.get(fileName);

    if ((fileEntry == null) || (dirEntry == null)) {
      throw new RuntimeException("El archivo debe existir en el paquete");
    }

    // Calculamos el Cycle Redundancy Check
    // Este es un campo de 32 bits que sirve para
    // verificar que el archivo no este corrupto
    CRC32 crc = new CRC32();
    for (int i = 0; i < data.length; i++)
    {
      crc.update(data[i]);
    }

    byte[] uncompressedFile = data;
    byte[] compressedFile;

    if (fileEntry.getCompressionMethod() == METHOD_STORED)
    {
      compressedFile = uncompressedFile;
    }
    else if (fileEntry.getCompressionMethod() == METHOD_DEFLATED)
    {
      compressedFile = compress(uncompressedFile);
    }
    else
    {
      throw new RuntimeException("Metodo de compresion desconocido.");
    }

    fileEntry.setCompressedFile(compressedFile);

    long compressedSize = compressedFile.length;
    long uncompressedSize = uncompressedFile.length;
    long crc32 = crc.getValue();
```

```
    fileEntry.setCompressedSize(compressedSize);
    fileEntry.setUncompressedSize(uncompressedSize);
    fileEntry.setCrc32(crc32);

    if (fileEntry.hasExt()) {
      fileEntry.setExtCompressedSize(compressedSize);
      fileEntry.setExtUncompressedSize(uncompressedSize);
      fileEntry.setExtCrc32(crc32);
    }

    dirEntry.setCompressedSize(compressedSize);
    dirEntry.setUncompressedSize(uncompressedSize);
    dirEntry.setCrc32(crc32);

    endOfCentralDirectoryEntry.setCentralDirectoryStartOffset(getStartOfCentralDir());
    endOfCentralDirectoryEntry.setCentralDirectorySize(getSizeOfCentralDir());

    // Calculamos las nuevas posiciones de los archivos en el paquete
    calcFileEntriesOffsets();
  }

  // Comprime un arreglo de bytes usando la biblioteca ZLIB
  static private byte[] compress(byte[] input) {
    Deflater deflater = new Deflater(Deflater.BEST_SPEED, true);
    deflater.setInput(input, 0, input.length);
    deflater.finish();
    byte[] buff = new byte[input.length + 1024];
    int wsize = deflater.deflate(buff);

    int compressedSize = deflater.getTotalOut();

    if (wsize == 0)
      throw new RuntimeException();
    else if (wsize != compressedSize)
      throw new RuntimeException();

    if (deflater.getTotalIn() != input.length)
      throw new RuntimeException();
    if (compressedSize >= input.length - 4)
      throw new RuntimeException();

    byte[] output = new byte[compressedSize];
    System.arraycopy(buff, 0, output, 0, compressedSize);
    return output;
  }

  ...
}
```

Listado 5.2: Inserción de Archivos

Un problema que se encontró fue que los números enteros del formato ZIP (usados, por ejemplo, para indicar el tamaño de los archivos) usan el ordenamiento de bytes conocido como *little endian* y no tienen signo mientras

que en Java se usa el ordenamiento *big endian* y los enteros tienen signo. Recordemos que el ordenamiento de bytes es simplemente la posición en que cada byte se almacena en el segmento de 4 bytes que representan al entero (en el caso de un entero de 32 bits). La solución a este problema es únicamente revertir el orden de los bytes al momento en el que se leen o escriben los números enteros. Este soporte se agregó en las clases LEDataInputStream (listado 5.3) y LEDataOutputStream (listado 5.4).

```
public class LEDataInputStream extends DataInputStream {

  public LEDataInputStream(InputStream in) {
    super(in);
  }

  // Lee un entero de 16 bits
  public final int readLEShort() throws IOException {
    int ch1 = in.read(); // Lee un byte
    int ch2 = in.read(); // Lee un byte
    // Combinamos los bytes recorriendo el segundo 8 bits a la derecha
    return (ch1 & 0xff) | ((ch2 & 0xff) << 8);
  }

  // Lee un entero de 32 bits
  public final long readLEInt() throws IOException {
    // Una forma elegante de revertir todos los bytes...
    return readLEShort() | ((long) readLEShort() << 16);
  }
}
```

Listado 5.3: LEDataInputStream

Notemos que en el código en el listado 5.3 los enteros de 16 bits se almacenan, en Java, usando el tipo de dato `int` el cual tiene 32 bits, mientras que los enteros de 32 bits se almacenan usando el tipo de dato `long` de 64 bits. La razón de esto es que los enteros en Java tienen signo por lo que el primer bit está reservado. Por lo tanto, si usáramos, por ejemplo, el tipo de dato `short` para almacenar un entero de 16 bits sin signo, habría algunos valores que en Java representarían un número completamente distinto y negativo.

```
public class LEDataOutputStream extends DataOutputStream {

  public LEDataOutputStream(OutputStream out) {
    super(out);
  }

  // Escribe un entero de 16 bits
  public void writeLEShort(int v) throws IOException {
    super.write((v >>> 0) & 0xff);
```

```
    super.write((v >>> 8) & 0xff);
  }

  // Lee un entero de 32 buts
  public void writeLEInt(long v) throws IOException {
    super.write((int)((v >>> 0) & 0xff));
    super.write((int)((v >>> 8) & 0xff));
    super.write((int)((v >>> 16) & 0xff));
    super.write((int)((v >>> 24) & 0xff));
  }
}
```

Listado 5.4: LEDataOutputStream

Para escribir se sigue exactamente la misma estrategia de revertir el orden de los bytes.

## 5.6. GenerarDocumentoService

Una vez descritos los principales componentes del servicio, veamos cómo el servicio responde a una solicitud de generación de documento:

1. Se reciben el identificador de la plantilla y el archivo XML de Datos.
2. Se obtiene la plantilla del Pool de Plantillas.
3. Se crea un clon de la plantilla el cual será el nuevo documento generado.
4. Se llama el método writeContentControls de DocxUtils pasando el clon de la plantilla y el DOM del XML de Datos.
5. Una vez que writeContentControls regresa, el nuevo documento ya tiene los controles de contenido actualizados.
6. El nuevo documento se guarda en el servicio de Documentum.
7. Se regresa el identificador del nuevo documento en Documentum.

La interacción completa se muestra en la figura 5.10.

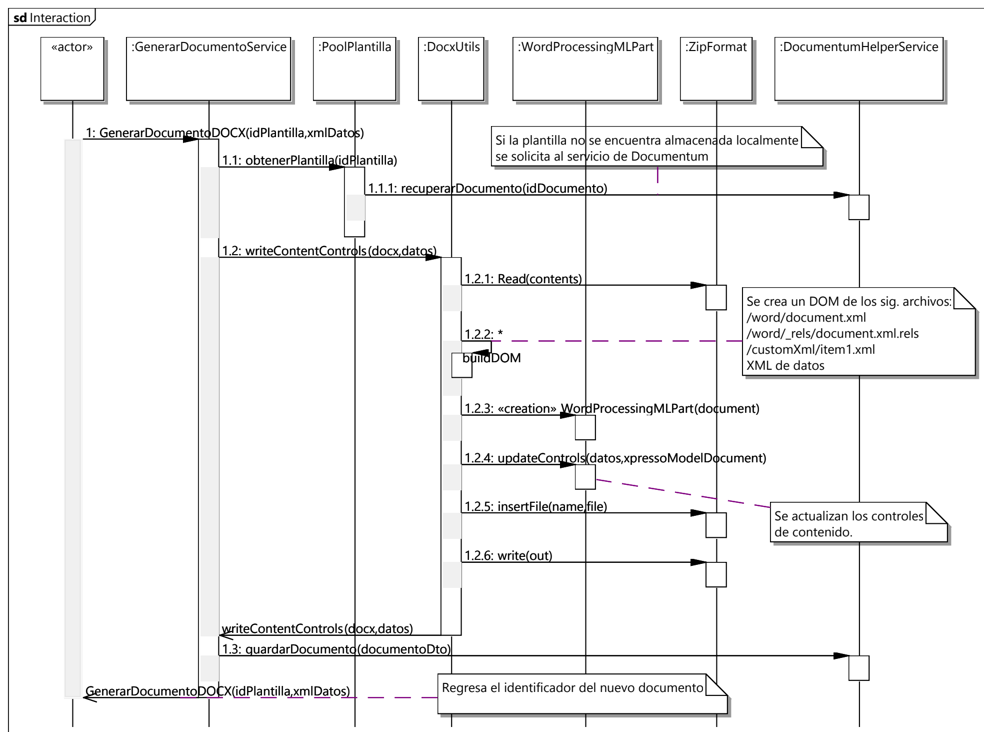


Figura 5.10: Generación del Documento

# Capítulo 6

# Conclusiones

El Generador de Documentos fue un proyecto concluido exitosamente y en tiempo a pesar de su complejidad. En este reporte se han omitido una enorme cantidad de detalles debido a que resultaría imposible describir todas las decisiones, tecnologías y algoritmos que se emplearon para resolver la amplia cantidad de problemas que surgen de manera natural al desarrollar un sistema como éste. Habiendo trabajado ya durante más de 7 años en la industria de desarrollo de software, puedo decir que el éxito de este proyecto ha sido algo excepcional en una industria en donde los retrasos, los defectos, los problemas de seguridad o simplemente los proyectos cancelados son la norma y no la excepción. La industria de desarrollo emplea a muchas personas, desafortunadamente muy pocos son los que tienen la formación necesaria para poder enfrentar la gran variedad de problemas que se presentan en casi cualquier proyecto no trivial.

El objetivo de este trabajo ha sido mostrar un pequeño ejemplo de cómo he aplicado los conocimientos adquiridos en la carrera durante mi trayectoria laboral: desde el entendimiento de los protocolos usados en los servicios web (visto en Redes de Computadoras), hasta la comprensión a nivel de byte del almacenamiento de los números enteros (visto en Arquitectura de Computadoras). El entendimiento de la complejidad computacional (visto en Análisis de Algoritmos) lo utilizo para no perder el tiempo haciendo optimizaciones triviales con las que sólo se ganarían unas milésimas de segundo. Mi conocimiento acerca de las estructuras de datos (visto en Introducción a las Ciencias de la Computación) lo uso para mantener una complejidad baja en las funciones que implemento. En general, el pensamiento analítico y el gusto por la formalidad que desarrollé durante toda la carrera lo uso en todo momento.

# Bibliografía


[1] EMC Corporation. Documentum. http://www.emc.com/domains/documentum/index.htm.

[2] EMC Corporation. xpresso for word. http://www.docscience.com/products/xpressionde.htm.

[3] ECMA. Standard ecma-376. http://www.ecma-international.org/publications/standards/Ecma-376.htm.

[4] Thomas Erl. *SOA Design Patterns*. Prentice Hall, 2008.

[5] The Apache Software Foundation. Axis. http://axis.apache.org/.

[6] The Apache Software Foundation. Formatting objects processor. http://xmlgraphics.apache.org/fop/.

[7] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, 1994.

[8] Jaspersoft. Jasper reports. http://community.jaspersoft.com/project/jasperreports-library.

[9] PKWARE. Formato zip. http://www.pkware.com/documents/APPNOTE/APPNOTE-6.3.2.TXT.

[10] Aaron Skonnard and Martin Gudgin. *Essential XML Quick Reference: A Programmer's Reference to XML, XPath, XSLT, XML Schema, SOAP, and More*. Addison-Wesley Professional, 2001.