Titulació:

Enginyeria en Organització Industrial

Alumne (nom i cognoms):

Jaume Nebot Martínez

Títol PFC:

Estudio de la arquitectura SOA (*Service Oriented Architecture*) como plataforma de desarrollo, administración y explotación de aplicaciones basadas en servicios (SaaS)

Director del PFC:

Juan Carlos Hernández Palacin

Convocatòria de lliurament del PFC

Setembre 2013

Contingut d'aquests volum: **-MEMÒRIA-**

ÍNDICE

ÍNDICE DE FIGURAS

ÍNDICE DE TABLAS

1. ACRÓNIMOS

SOA	Service Oriented Architecture
ESB	Enterprise Service Bus
POO	Programación Orientada a Objetos
RPC	Remote Procedure Call
RMI	Remote Method Invocation
CORBA	Common Object Request Object Architecture
OMG	Object Management Group
WS	Web Services
ODBC	Open DataBase Connectivity
SQL	Structured Query Language
COM	Component Object Model
DCOM	Distributed Component Object Model
EJB	Enterprise JavaBeans
JVM	Java Virtual Machine
W3C	World Wide Web Consortium
SLA	Service Level Agreement
QoS	Quality of Service
J2EE	Java 2 Platform Enterprise Edition
CIM	Computer Integrated Manufacturing
XML	Extensible Markup Language
SOAP	Simple Object Access Protocol
WSDL	Web Services Description Language
UDDI	Universal Description, Discovery and Integration
HTTP	Hypertext Transfer Protocol
OASIS	Organization for the Advancement of Structured Information Standards
SGML	Standard Generalized Markup Language
WS-I	Web Services Interoperability Organization
B2B	Business to Business
ODA	Open Document Architecture
UTF-8	8-bit Unicode Transformation Format
URI	Uniform Resource Identifier
URL	Uniform Resource Locator
WWW	World Wide Web
FTP	File Transfer Protocol
SMTP	Simple Mail Transfeotocor Prl
RTT	Round-Trip delay Time
TCP	Transmission Control Protocol
BPM	Business Process Management
JNDI	Java Naming and Directory Interface
BPEL	Business Process Execution Language
CRM	Customer Relationship Management
ROI	Return of Investment
REST	Representational State Transfer
XOP	XML-binary Optimized Packaging
MTOM	SOAP Message Transmission Optimization Mechanism
EAI	Enterprise Application Integration
HTTPS	Hypertext Transfer Protocol Secure
JMS	Java Message Service
ERP	Enterprise Resource Planning
XSLT	Extensible Stylesheet Language Transformations

MIT	Massachusetts Institute of Technology
SSH	Secure Shell
GUI	Graphical User Interface
ASF	Apache Software Foundation
JRE	Java Runtime Environment
NMR	Normalized Message Routing
OSGI	Open Services Gateway Initiative
API	Application Programming Interface
SO	Sistema Operativo
JAAS	Java Authentication and Authorization Service
JBI	Java Business Integration
SE	Service Engines
BC	Binding Components
EIP	Enterprise Integration Patterns
IEEE	Institute of Electrical and Electronics Engineer
SaaS	Software as a Service
PaaS	Platform as a Service
IaaS	Infraestructure as a Service
AWS	Amazon Web Services
RDS	Relational Database Service
JDK	Java Development Kit
IDE	Integrated Development Enviroment
RAM	Random-access memory
WADL	Web Application Description Language

2. OBJETO DEL PROYECTO

El propósito de este proyecto es el estudio de la arquitectura SOA (*Service Oriented Architecture)* en su nivel conceptual y también explorar qué herramientas existen actualmente dentro del campo de los sistemas de información que se basan en este paradigma. Dentro de la arquitectura SOA se prestará especial atención a una parte fundamental de ella, el ESB o *Enterprise Service Bus*, finalizando el proyecto con un ejemplo de aplicación sobre este componente. Un ESB es una herramienta software que ayuda en el cumplimiento de los requisitos que propone SOA. Entre estos requisitos encontramos la integración de las aplicaciones informáticas, debido a que habitualmente la información fluye a través de varias de ellas, el procesado y transformación de los datos que se intercambian en las aplicaciones, ya que es posible que cada aplicación trabaje con un formato diferente de los datos, el enrutamiento de las comunicaciones entre las diferentes aplicaciones conectadas al ESB, para tener centralizado e identificado el camino que sigue cada uno de los flujos de información, y la coordinación de los diferentes procesos que tienen lugar dentro del ESB.

3. JUSTIFICACIÓN

En las últimas décadas, la informatización de las organizaciones ha crecido de forma espectacular para poder ser competitivos en el mercado actual y ganar en eficiencia interna. Los sistemas de información son presentes en todas las áreas de la empresa, producción, logística, recursos humanos, finanzas, etc.

Esta gran variedad de áreas dentro de la empresa ha sido la base para la creación de un "ecosistema software", un conjunto de aplicaciones informáticas interdependientes que comparten un mismo hábitat que en muchos casos están basadas en tecnologías diferentes, ya sea porque a lo largo del tiempo la tecnología ha ido avanzando con la aparición de nuevos lenguajes de programación más usables y las nuevas aplicaciones se han basado en estas nuevas tecnologías, o porque las necesidades de negocio y/o infraestructuras propician la adaptación de una tecnología en concreto.

Como necesidades de negocio entendemos esas necesidades que surgen de las tareas relacionadas con la actividad de la empresa, como podrían ser procesos de compra, las ventas, controles de inventario, contabilidad, marketing, etc… Por ejemplo, antes de la aparición de Internet, las aplicaciones informáticas en una empresa eran programas que se instalaban en cada terminal, con la aparición de Internet se generó la oportunidad de realizar aplicaciones web que permitían el uso de navegadores web para acceder a las aplicaciones, solamente se requiere en cada ordenador pues, poseer un navegador para poder trabajar con las aplicaciones. Esta nueva forma de usar la tecnología implicó un nuevo abanico de oportunidades para las necesidades de negocio, por ejemplo, con Internet las ventas podían ser *online.* Es de esta forma, como a medida que la tecnología avanza, también avanzan los sistemas informáticos de la empresa, algunos lo hacen con diferentes tecnologías y otras se quedan tal y como están.

Estas aplicaciones pero, no están exentas de interactuar entre ellas y es común que la información deba fluir por varios sistemas para ser tratada o que una misma información sea requerida por varios sistemas a la vez. Por ejemplo, la información relacionada con una orden de pedido será tratada en los ámbitos de ventas, gestión de inventario, contabilidad... Por lo tanto, se requiere una adaptabilidad de las aplicaciones para que puedan interaccionar. Tal variabilidad de tecnologías dificulta mucho la tarea.

Es en este contexto pues, dónde la arquitectura SOA y un ESB permiten optimizar los sistemas de información y la tarea de comunicación entre sistemas.

4. ALCANCE

El alcance de este proyecto cubre el estudio de la arquitectura SOA, así como los conceptos necesarios para poder entenderla y los pasos previos en la historia de la arquitectura de software que han dado derivado en su creación. La memoria también incluye el estudio de los ESB, herramientas importantes dentro de la infraestructura SOA. Entre los ESB, se escogerá una herramienta en concreto sobre la cual se detallarán sus características, su instalación, y su puesta en marcha a partir de un ejemplo práctico debidamente explicado.

El alcance del proyecto no cubre la aportación de la infraestructura informática necesaria para llevar a cabo las tareas mencionadas anteriormente, es decir, ni los equipos informáticos necesarios, ni la conexión a Internet.

5. INTRODUCCIÓN HISTÓRICA A LA ARQUITECTURA DE SOFTWARE

La arquitectura de software es la parte de la informática que propone guías y patrones para desarrollar sistemas de información de forma estructurada. A lo largo de la historia se ha ido evolucionando en la forma en que se han desarrollado las aplicaciones informáticas, en la actualidad, la arquitectura SOA se ha convertido en el último eslabón de dicha evolución. Es por lo tanto interesante realizar una breve introducción para ver los pasos que han ido sucediendo a lo largo de la historia y que han llevado a la creación de SOA.

La primera persona que recomendó una estructuración correcta de los sistemas de software fue Edsger Dijkstra en 1968 [1]. Dijkstra introdujo la noción de sistemas operativos organizados en capas superpuestas dónde cada capa sólo se comunica con sus capas adyacentes. En el año 1969 P.I. Sharp introdujo el concepto de arquitectura de software basándose en las ideas de Dijkstra [1], diferenciando la ingeniería de software de la arquitectura de software tomando como ejemplo el sistema operativo de IBM OS/360. Sharp hace notar que aunque este sistema operativo está construido utilizando buenas prácticas de programación (debido a la ingeniería de software), en su conjunto constituye un amontonamiento amorfo de programas y eso es debido a que no había una figura de arquitecto de software que estructurara debidamente los subprogramas que constituían el sistema operativo.

5.1 *PROGRAMACIÓN ESTRUCTURADA*

En la década de los 70 aparece el concepto de diseño estructurado y los primeros modelos explícitos de desarrollo de software [1]. Esta forma de desarrollo solo contempla tres lógicas de control [2]:

- Secuencia: son bloques de sentencias que se ejecutan uno después del otro.
- Condicional: bloques de sentencias que se ejecutan solo si se cumple una condición.
- Interacción: repetición de una o varias sentencias mientras se cumpla una condición dada.

Con ello, se considera que un programa es estructurado si cumple las siguientes condiciones:

- Tiene un único punto de entrada y un único punto de salida.
- Todas y cada una de las sentencias del programa son ejecutables, es decir, no hay código que no se pueda ejecutar nunca debido a cierta casuística.
- No contiene bucles infinitos.
- Todos los posibles caminos llevan desde el punto de entrada al punto de salida.

Con la programación estructurada se imponía un orden a la programación, haciendo los programas más entendibles y mantenibles hasta cierto punto, ya que a medida que crezca la aplicación aunque esta esté estructurada será difícil su mantenimiento debido a que todas las instrucciones de código están juntas, constituyendo lo que se denomina una "aplicación monolítica" [2]. Este tipo de aplicaciones monolíticas tienen dos problemas, el primero es que no es posible la reutilización de código y el segundo es que se hace muy difícil la localización de errores, provocando pues un mantenimiento y evolución de la aplicación muy costoso. Para resolver estos problemas, apareció pues la programación modular.

5.2 *PROGRAMACIÓN MODULAR*

La programación modular se basa en la estrategia de "divide y vencerás" [2]. Ante un problema, es conveniente descomponerlo en partes más pequeñas para que cada parte sea abordada por separado y su resolución sea más sencilla. Un programa pues, se descompone en diferentes módulos que interactúan entre sí. Habrá un módulo principal que coordine las llamadas a otros módulos secundarios (mediante el uso de funciones) y la comunicación se realiza con el paso de parámetros. De esta manera conseguimos que cada módulo recibe unos parámetros de entrada y conforma un resultado de salida, convirtiéndose en una caja negra y que puede ser reutilizada, resolviendo así los problemas derivados de una programación estructurada. Un ejemplo de lenguaje modular es el lenguaje C. A partir de la programación modular surge la programación orientada a objetos, que utiliza objetos del mundo real para representar los módulos y las funciones.

5.3 *PROGRAMACIÓN ORIENTADA A OBJETOS (POO)*

La programación orientada a objetos se popularizó a principios de la década de los 90 y es el paradigma de programación más usado actualmente [2]. Con ella, se pretende resolver los problemas de la programación utilizando el trabajo colaborativo entre objetos. Un objeto es una entidad del mundo real trasladada a la programación. Estos objetos tienen atributos y comportamientos. Los atributos describen propiedades o características del objeto, y los comportamientos (o métodos) indican las operaciones que puede realizar el objeto o que se pueden realizar sobre el objeto. Los valores que tienen los atributos del objeto en un momento dado, se denomina el estado d un objeto. Tanto los atributos como los métodos de un objeto se representan conjuntamente en una clase (Figura 1). En esta clase se pueden apreciar sus atributos (DNI, numEmpleado, nombreEmpleado) y un método (altaEmpleado) que tiene como parámetros los tres atributos.

```
class Empleado
{
            string DNI;
            int numEmpleado;
            string NombreEmpleado;
            void AltaEmpleado(string DNI, int numEmpleado,
string NombreEmpleado)
            {…}
            …
}
```

Fig. 1. Una clase en programación orientada a objetos [2].

A partir de una clase se puede instanciar en memoria un objeto de dicha clase, tal y como se muestra en la Figura 2. Se instancia un objeto Empleado con la instrucción new Empleado() y posteriormente se asigna un valor a su atributo DNI.

```
Empleado objEmp = new Empleado();
objEmp.DNI="xxxxxxxxxx";
```

Fig. 2. Instanciación de un objeto a partir de una clase [2].

Las características principales de la programación a objetos son las siguientes:

- Abstracción: Con la abstracción se pretende obtener las características esenciales de un objeto. Por ejemplo, en el caso del empleado sería recoger sus atributos comunes (DNI, nombre, número de empleado, etc..) y las acciones posibles que puede hacer un empleado (alta, baja, modificación de datos, etc…).
- Encapsulamiento: establece el nivel de visibilidad de los atributos de una clase. Por ejemplo, podemos proteger el atributo DNI para que solamente se pueda utilizar desde dentro de la clase, o en caso contrario, podemos dar visibilidad a que se pueda usar desde fuera tal y como ocurre en el ejemplo de la Figura 2.
- Principio de ocultación: se utiliza conjuntamente con el encapsulamiento, de forma natural los atributos de un objeto están aislados del exterior, es mediante la creación de una interfaz (describiendo los métodos adecuados de la clase) con los que daremos acceso al exterior a los atributos de una clase. Por ejemplo, podemos dar acceso al atributo numEmpleado al exterior mediante un método getNumEmpleado() que devuelva el número de empleado. El principio de ocultación se utiliza para controlar el acceso al estado de un objeto y que éste no pueda cambiar inesperadamente de forma no controlada desde el exterior.
- Herencia: Se puede modular una jerarquía entre clases. Por ejemplo, supongamos que deseamos modelar una estructura organizativa en una empresa, donde tenemos muchos tipos de empleados, directores, gerentes, operarios, cada uno tiene sus características particulares, pero también tienen características en común. El nombre, el número de empleado, DNI, etc… con lo cual, podemos crear una super-clase Empleado con estos atributos en común, y luego clases hijas que hereden de ésta. La clase

Directivo heredará de Empleado sus atributos y además, tendrá los propios suyos, como por ejemplo, cual es su despacho. De esta manera, cada vez que creamos un tipo diferente de empleado no creamos otra vez su nombre, número de empleado ni DNI.

- Polimorfismo: Esta propiedad hace referencia a dos situaciones. La primera es que debido a la herencia podamos utilizar una clase padre como contendedor de una clase hija y usar los métodos de la clase padre. Por ejemplo, en la Figura 3 se muestra como declaramos un objeto Circulo como objeto Figura (su clase padre) y utilizamos sobre ella el método Dibujar().

```
Figura fig = new Figura();
Figura fig2 = new Circulo();
fig.Dibujar(); //Dibujará una figura
fig2.Dibujar(); //Dibujará un Circulo
```

Fig. 3. Uso de polimorfismo a nivel de herencia [2].

En la Figura 4 observamos el segundo tipo de polimorfismo, esto es poder crear métodos de una clase con el mismo nombre pero con diferentes parámetros de entrada.

```
double sumar (int op1, intop2) {...}
double sumar (double op1, double op2) {...}
```

Fig. 4. Uso del polimorfismo a nivel de métodos [2].

Como ejemplo de lenguajes de programación orientados a objetos encontramos C++, Java, C# y lenguajes de scripting que también soportan orientación a objetos como Phyton o Perl.

Hasta este momento la interacción entre objetos se produce a nivel de la misma aplicación y en la misma máquina. El siguiente paso evolutivo es que esta interacción se pueda realizar entre diferentes máquinas, dando lugar al software distribuido.

5.4 *SOFTWARE DISTRIBUIDO*

El software distribuido es aquel en que sus elementos pueden comunicarse entre sí aunque estén en máquinas diferentes a través de una red de comunicaciones mediante lo que se denomina protocolo RPC (*Remote Procedure Call).* El hecho de que el software sea distribuido es transparente al usuario, es decir, se trabaja con el software distribuido de la misma forma que se trabajaría en la programación orientada a objetos, sin estar pendientes de la gestión de las comunicaciones.

Estos sistemas son altamente acoplados. Por acoplados se entiende que hay una gran dependencia entre los elementos de cada capa que conforman la arquitectura distribuida, como

veremos a continuación, encontraremos diferentes capas como la de presentación, negocio y datos. Este acoplamiento entre capas se debe a que la interacción entre ellas se realiza con tecnologías muy específicas de cada vendedor, es decir, cada vendedor utiliza sus propios mecanismos de transmisión entre capas.

Dentro del software distribuido encontramos diferentes modelos de arquitectura [2]:

- Cliente-Servidor (Figura 5): Este sistema está conformado por un cliente donde reside la lógica de negocio y acceso a datos y el servidor suele ser únicamente un repositorio de información ya sea una base de datos, un repositorio de archivos,etc…

  Se entiende por **lógica de negocio** en el ámbito de la informática la parte de la aplicación que procesa los datos asociados a las tareas que se deban acometer. Por ejemplo, dentro de un proceso de venta se tendrán que consultar datos del comprador, del producto a vender, comprobar si hay stock suficiente, generar informes etc… Desde un punto de vista de la programación orientada a objetos, la lógica de negocio es la funcionalidad ofrecida por el software.

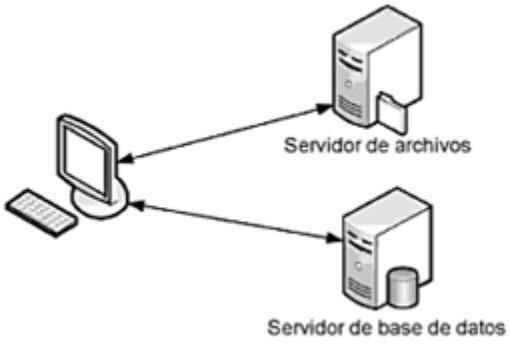

Fig. 5. Arquitectura cliente-servidor [2].

- Arquitectura 3-Tier (Figura 6): Dentro de las arquitecturas N-Tier (N capas) la más común es la de tres capas o 3-Tier. Esta arquitectura separa la lógica de negocio en una nueva capa. Con lo una aplicación queda divida en las siguientes capas:

  - Nivel de presentación: Se encarga de la presentación de la aplicación, es la parte de la aplicación por la que el usuario interactúa (la interfaz de usuario).
  - Nivel de negocio: En esta capa reside la lógica de negocio, donde se reciben las peticiones del usuario y se envían las respuestas después del procesado. Esta capa también se comunica con la capa de datos para obtener la información necesaria.
  - Nivel de datos: Es la capa donde están almacenados los datos, por ejemplo una base de datos.

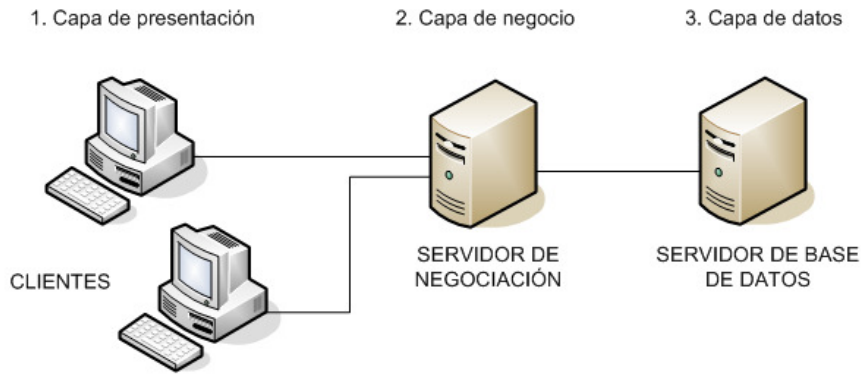

Fig. 6. Arqutiectura 3-Tier [3].

Como ejemplo de lenguajes que pueden operar en entornos distribuidos encontramos Java con su RMI (*Remote Method Invocation*), la tecnología DCOM de Microsoft y CORBA (*Common Object Request Object Architecture*) que es un estándar definido por la organización OMG (*Object Management Group*).

Con estas tecnologías mencionadas, se consigue traspasar la arquitectura de la programación orientada a objetos que se ejecuta en una misma máquina a un entorno distribuido compuesto de varias máquinas. Con lo cual, un objeto en una máquina es capaz de llamar al método de otro objeto que está alojado en una máquina diferente, la comunicación entre los cuales se realiza mediante la serialización, es decir, se convierten los parámetros enviados al método remoto a bytes antes de ser enviados. Posteriormente en el método remoto, se realiza el proceso inverso, los bytes se deserializan para obtener los parámetros.

Dentro de este entorno distribuido, aparecen posteriormente los componentes distribuidos, que son una evolución de las clases de la programación orientada a objetos a un nivel superior. Es decir, una clase representa un objeto del mundo real, con un componente se abarca un ámbito más grande, una funcionalidad en su conjunto.

5.5 *COMPONENTES DISTRIBUIDOS*

Dentro de los entornos distribuidos aparecen los componentes, que son módulos que encapsulan una funcionalidad [4]. Estos componentes proveen interfaces que especifican dicha funcionalidad para que otros componentes puedan utilizarlos. Los componentes pueden existir por sí mismos (es lo que se denomina auto-contenido) lo que les confiere la capacidad de ser modificados o substituidos por un nuevo componente sin afectar al funcionamiento general del sistema, mientras la interfaz del componente no se vea modificada. Una de las características más importantes de los componentes es su reusabilidad, se deben diseñar los componentes pensando en que deben conformar una unidad de software que puede llegar a ser usada por más de una aplicación o componente. Por lo tanto, al conformar una unidad de software, los componentes estarán implementados bajo una plataforma en concreto como por ejemplo los

EJB (*Enterprise Java Beans*) y éstos deberán ser desplegados y posteriormente activados para poder ser usados. En la Figura 7 se muestra un ejemplo de diseño basado en componentes en un hipotético caso de reservas de vacaciones donde se puede apreciar la diferencia entre componentes y las Clases propias de la programación orientada a objetos.

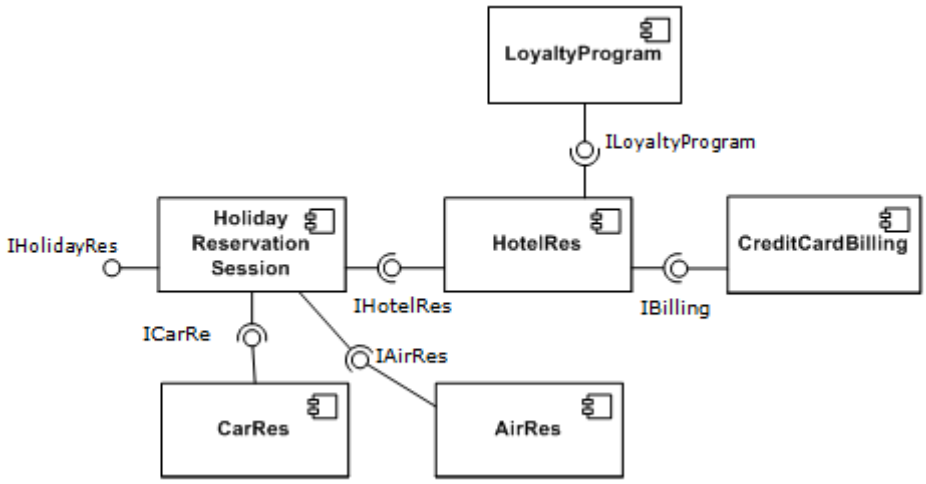

Fig. 7. Diseño de una aplicación basada en componentes [4].

Como clases tendríamos: Usuario, Reserva, Hotel, Coche, etc… En cambio en el ámbito de componentes tenemos entidades más globales, son funcionalidades. Como por ejemplo CarRes (Reserva de Coche). La interacción entre componentes se simboliza con la notación tipo *lollipop* (chupa-chups). La parte *lollipop* indica que se expone una funcionalidad, y la parte envolvente del *lollipop* indica que se utiliza dicha funcionalidad.

Como ya se ha mencionado anteriormente, la comunicación en entornos distribuidos se utiliza mediante protocolos RPC, lo que permite la comunicación entre componentes alojados en diferentes servidores. Esto se representa en la Figura 8, donde se aprecian una serie de componentes alojados en diferentes servidores. La comunicación entre ellos se realiza mediante RPC y los *proxy stubs*. Los *proxy stubs* son artefactos software que hacen de intermediario entre componentes alojados en diferentes servidores permitiendo la comunicación remota. Se observa como en el servidor A, los *proxy stubs* están coloreados con el mismo color que los componentes del servidor B, indicando así que proporcionan una interfaz de comunicación de los componentes del servidor B para los componentes del servidor A.

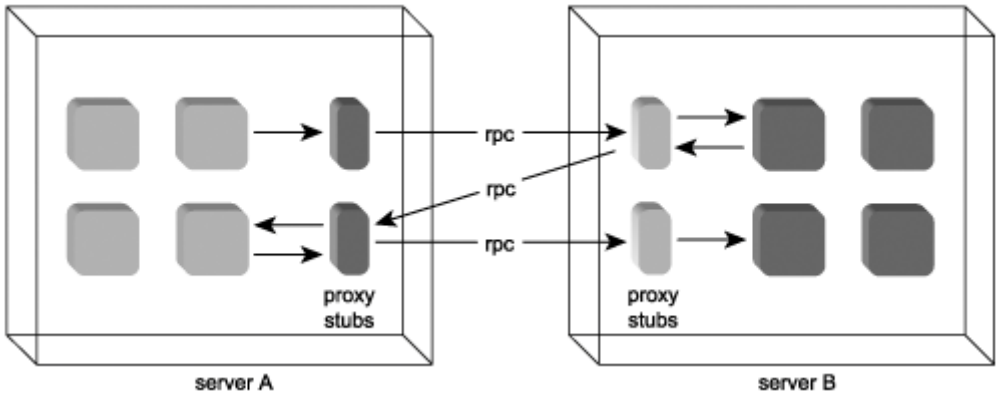

Fig. 8. Comunicación entre componentes alojados en diferentes servidores. [7]

5.6 *INTRODUCCIÓN A LOS SERVICIOS*

Con los servicios llegamos al final de esta evolución en la arquitectura de software. Un servicio es similar a un componente, es una unidad de trabajo bien definida, que acepta mensajes de entrada y produce mensajes de salida [2]. Así como un componente posee una interfaz que describe su funcionalidad para ser usado por otros componentes, un servicio también expone su funcionalidad al exterior mediante lo que se denomina un contrato [2]. Trataremos en profundidad los contratos de los servicios más adelante.

Los servicios se diferencian de los componentes en varios aspectos. Lo primero es que los componentes son usados entre aplicaciones en un ámbito local y los servicios pueden ser usados por otras empresas o instituciones. Los servicios pueden exponer al exterior una funcionalidad implementada a partir de componentes, esto confiere una nueva capa de abstracción dentro de la arquitectura de software, tal y como se representa en las Figura 9.

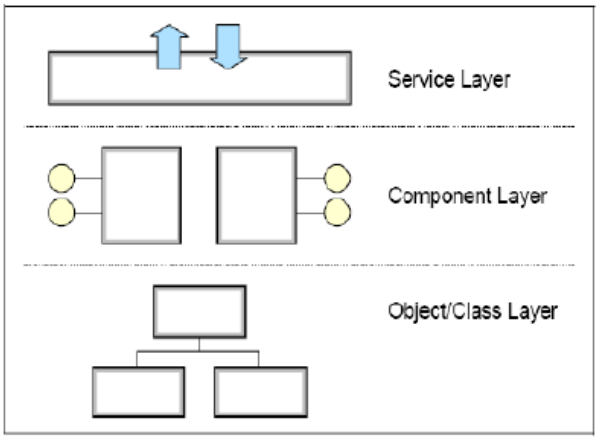

Fig. 9. Capas de aplicación: Servicios, componentes y objetos. [9]

Así como hemos descrito en la Figura 8 la comunicación remota entre componentes, podemos actualizar el mismo esquema con la nueva capa que aportan los servicios (Figura 10).

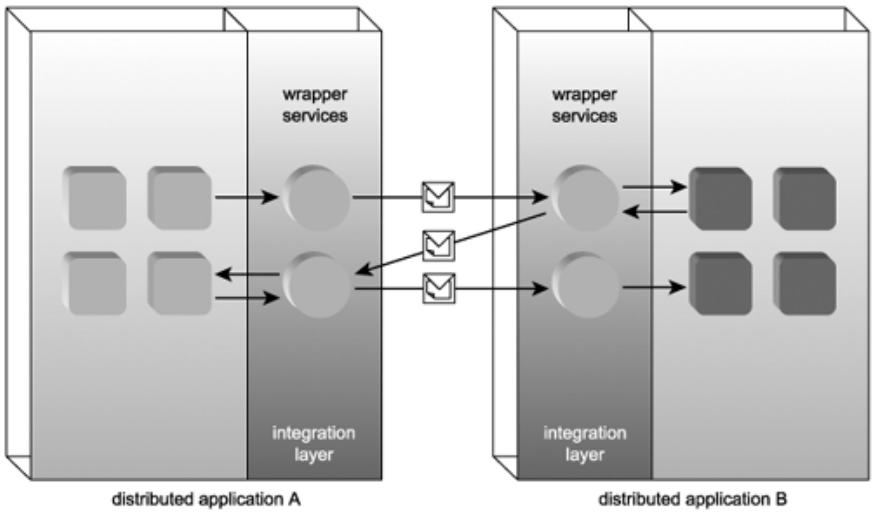

Fig. 10. Comunicación entre componentes alojados en diferentes servidores mediante la integración de servicios. [7]

La primera diferencia que se observa entre las Figuras 8 y 10 es que con los servicios se eliminan las conexiones RPC y la necesidad de implementar proxy stubs. Por el contrario, la comunicación se realiza mediante mensajes, donde un mensaje es una unidad de comunicación basada en texto, un ejemplo sencillo de mensaje sería por ejemplo un e-mail. El uso de mensajes a través de los servicios es muy importante debido a que confieren a los servicios la habilidad de integrar aplicaciones distribuidas que pueden estar implementadas con diferentes tecnologías ya que ahora no se depende de un protocolo concreto de comunicación. Los servicios construirán mensajes entendibles para los dos extremos de la comunicación e internamente se tratará la información y se adaptará a la tecnología concreta de cada aplicación, actuando así como envoltorio (*wrapper*) de los componentes. Este tipo de comunicación entre servicios que va directa del origen al destino sin pasar por ningún intermediario, se conoce como conexión punto a punto o en inglés *point-to-point*.

Otro diferencia remarcable es que al crear aplicaciones basadas en componentes se deben importar las referencias o librerías necesarias para usar los otros componentes, en cambio, en los servicios, no son necesarias debido a que las características de los servicios hacen que se requiera de otro servicio en el mismo instante que se ejecuta (lo que se dice en tiempo de ejecución o *runtime*), no cuando se construye el servicio.

Más adelante entraremos en detalle sobre los servicios debido a que son la base de la arquitectura SOA. De momento esta breve introducción a los servicios nos sirve para analizar las Figuras 11 y 12. En la figura 11 mostramos de forma gráfica y resumida la evolución de la

arquitectura de software analizada en todo este punto. Desde la arquitectura monolítica hasta los servicios. Se aprecia que antes de los servicios aparece *Web Services*. Los *Web Services* (o servicios web) son una forma de crear servicios que, como su nombre indica, operan en la Web (Internet). Más adelante se explicarán con más detenimiento debido a que el uso de servicios web es actualmente la forma más eficaz y extendida para implementar SOA [7]. Es importante remarcar el hecho de que el uso de los servicios web no implica de por sí que se está operando bajo el paradigma SOA, para que éstos se considere que operan bajo una arquitectura SOA, deben cumplir una serie de principios, como se verá más adelante.

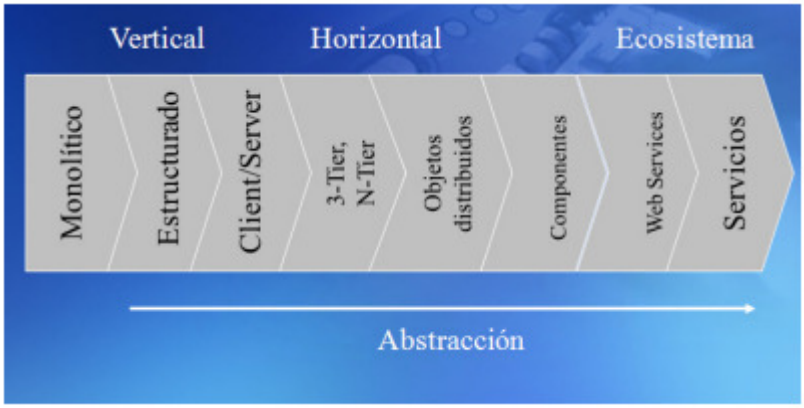

Fig. 11. Evolución de la arquitectura de software [5].

En la Tabla 1, se muestra en un cuadro comparativo los diferentes paradigmas de programación mencionados hasta ahora en base a los siguientes parámetros:

	Programación Estructurada	Objetos	Componentes	Servicios
Granularidad	Muy fina	Fina	Intermedia	Gruesa
Contrato	Definido	Privado/Publico	Publico	Publicado
Reusabilidad	Baja	Baja	Intermedia	Alta
Acoplamiento	Fuerte	Fuerte	Débil	Muy débil
Dependencias	Tiempo de Compilación	Tiempo de Compilación	Tiempo de Compilación	Run-Time
Ámbito de Comunicación	Intra-Aplicación	Intra-Aplicación	Inter-Aplicaciones	Inter-Empresas

Tabla 1. Propiedades de los diferentes tipos de arquitecturas. [5]

- **Granularidad:** la granularidad hace referencia a la composición de un elemento. Es decir, una mayor granularidad indica que un componente puede estar formado por un conjunto de subcomponentes, y a medida que descendemos en nivel de detalle vamos encontrando componentes de grano más fino. Por ejemplo, si nos fijamos en la figura 7, dentro del componente CarRes podríamos tener un objeto Car que nos describiría un coche. Este componente tendría una granularidad fina. A un nivel mayor tendríamos el componente CarRes que utiliza la entidad Car para desempeñar la funcionalidad de reserva de coche. Para finalizar, el componente HolidayReservation Session aun tiene una granularidad mayor a CarRes debido a que usa el componente CarRes y otros componentes.
  En la Tabla 1, se observa pues como a medida que ha ido evolucionado la arquitectura de software la tendencia ha sido ir creando artefactos software con una mayor granularidad. En efecto, si observamos la Figura 8 vemos que en la capa más inferior tenemos la programación orientada a objetos, luego la capa de componentes que se alimenta de la capa de POO y a su vez, la capa de servicios se puede alimentar de la capa de componentes.

- **Contrato:** Los contratos representan conceptos diferentes en función de la arquitectura de software, pero tienen en común el hecho de que representan la funcionalidad expuesta a modo de interfaz por un artefacto software. Una interfaz es una definición del "qué" puede hacer dicho artefacto, por el contrario, si nos referimos al "cómo" se realiza nos referimos a la implementación de dicha interfaz.

  En programación estructurada y en programación orientada a objetos podríamos tener una función o método donde la definición de la cabecera del método opera como un contrato, una posible cabecera sería la siguiente:

  ```
  int sumar(int num1, int num2)
  ```

  Esta cabecera nos indica que tenemos una método llamado "sumar" (marcado en azul) que devolverá un número del tipo entero (marcado en verde) y que recibe como parámetros dos números enteros (marcados en rojo). Como vemos, este contrato nos da información de lo qué podemos hacer, qué se necesita para hacerlo y qué obtenemos como resultado (podría no haber valor de retorno), pero no se muestra el cómo se hace. En la figura 12 se indica que en los Objetos el contrato puede ser público o privado, esto es debido a que una clase puede exponer la cabecera del método en el ámbito privado, solamente se puede usar este método desde la misma clase, o público, se puede llamar al método desde fuera de la clase, por ejemplo, desde otra clase diferente. En el caso de la programación distribuida, podríamos tener el siguiente ejemplo de contrato representado en la Figura 12 en base a una arquitectura cliente-servidor para un acceso a base de datos. En este caso, el contrato sería el protocolo de

comunicación usado con la base de datos (ODBC) y el lenguaje utilizado para realizar las consultas (SQL)

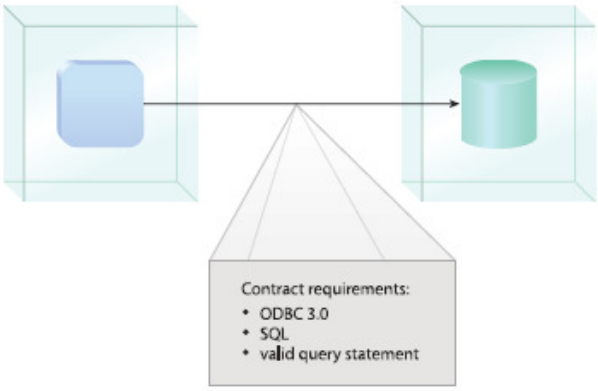

Fig. 12. Posible contrato en una arquitectura cliente-servidor. [8]

En el caso de los componentes que operan en una arquitectura distribuida podríamos tener el siguiente contrato (Figura 13). En esta figura se refleja que el contrato del componente refleja que para conectarse se debe usar la tecnología COM+ (evolución de la tecnología DCOM de Microsoft) y usar unos parámetros válidos de entrada y salida. En la figura 10 se indica que los componentes tienen un contrato público. A diferencia de los objetos que pueden tener métodos privados como hemos indicado, un contrato en un componente solamente se entiende si es público ya que define la funcionalidad del componente para ser usada por otros componentes.

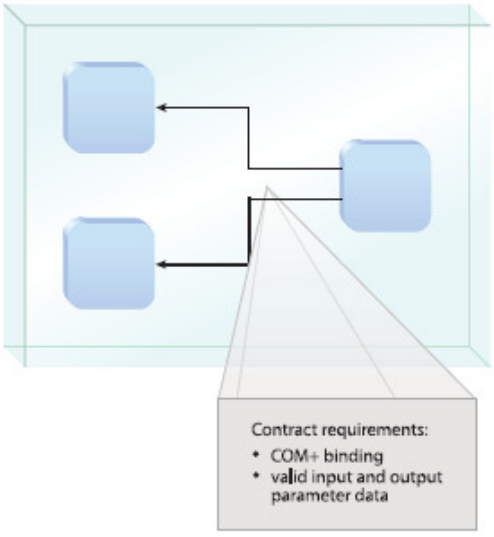

Fig. 13. Posible contrato para un componente en una arquitectura distribuida. [8]

Por último nos encontramos con el caso de los contratos para los servicios que trataremos con detalle en el siguiente capítulo. Podemos introducir que los contratos de los servicios son los más complejos ya que consta de una estructura bien definida con un alto nivel de detalle. Esta estructura consta de una cabecera con información relativa al servicio (nombre, versión, propietario, etc..) una sección con la definición de las operaciones que soporta el servicio y otros elementos, por ejemplo los relativos a seguridad. Se indica en la Tabla 1 que los contratos son publicados, a diferencia de los componentes que son públicos. Esta diferencia se entiende mejor si nos fijamos en el ámbito de comunicación, donde público hace referencia a un ámbito intra-aplicación (como los objetos) o inter-aplicación en el caso de los componentes. Es decir, en estos ámbitos podemos tener artefactos software que son visibles para los demás. El término publicado, da un paso más, se entiende como publicado a un nivel fuera de la aplicación o aplicaciones para tener un alcance mucho mayor y ser un elemento visible entre aplicaciones de diferentes ámbitos o empresas.

- **Reusabilidad:** Por reusabilidad se entiende la capacidad que tiene un elemento software de ser reutilizado en el futuro como parte de otro proceso diferente al que se creó en su origen. Es una buena práctica en la programación ya sea cuando creamos objetos, componentes o servicios idear el elemento con una visión puesta en los desarrollos posteriores que pueden aparecer en un futuro. Un ejemplo en programación orientada a objetos podría ser el uso de la herencia ya comentada anteriormente. Esta forma de diseñar se puede trasladar pues a elementos con una granularidad superior, como los componentes y los servicios, como el ámbito de comunicación es mayor en componentes y servicios, su reusabilidad por consiguiente es potencialmente mayor. Se debe indicar que la reusabilidad de los servicios es más alta porque los componentes están limitados a su entorno de ejecución, por ejemplo, los componentes EJB de Java se ejecutan en una o varias JVM (*Java Virtual Machine*). En cambio, los servicios pueden operar a nivel de entornos diferentes.

- **Acoplamiento:** El acoplamiento hace referencia al grado de dependencia que existe entre los elementos software. Mejor cuando el acoplamiento sea más débil debido a que se minimizan las dependencias haciendo que los componentes integrantes de la aplicación sean más fáciles de mantener ante posibles cambios. El ejemplo más simple de acoplamiento es cuando un elemento software accede directamente al dato de otro elemento software. El primer elemento dependerá pues del valor del dato del segundo elemento, con lo cual están acoplados.

  Al igual que un componente, el servicio ofrece reutilización, pero a diferencia del componente un servicio presenta mejoras debido a que tiene un menor grado de acoplamiento [5] gracias a su contrato como se verá posteriormente. Los componentes son más dependientes de la tecnología en las que operan, por ejemplo, los componentes EJB internamente usan RMI (también de Java) para sus comunicaciones.

Una de las características de los servicios será pues, su bajo acoplamiento y por lo tanto su alta independencia de la tecnología en la que se trabaja. Este es uno de los motivos principales por los que los componentes distribuidos han dado paso a los servicios. Otro motivo es el hecho de que las arquitecturas distribuidas (aparte de RMI de Java) han sido arquitecturas propietarias, es decir propias de un vendedor. Microsoft ha controlado DCOM, y aun CORBA, que ha sido un intento de de estandarización, la mayoría de sus implementaciones han sido propietarias obligando a quien las use a pagar por ellas [6].

- **Dependencias:** en esta tabla se indica el parámetro dependencias refiriéndose no al acoplamiento, sino en qué momento necesita un elemento software el conocer la existencia de otro elemento software con el que tiene una relación. Por tiempo de compilación se refiere a cuando se compila la aplicación, es decir, cuando se traduce a código máquina (el lenguaje que entiende el procesador) el código creado en lenguaje de alto nivel (C, Java, etc…). Para ser concisos, en el caso de Java se compila para que el código sea entendible para la máquina virtual de Java JVM no para el procesador. En este caso, este parámetro remarca que los servicios son más desacoplados que los componentes, ya que las relaciones con otros servicios se resuelven en tiempo de ejecución (*Run-Time*) y no en tiempo de compilación como los componentes.

- **Ámbito de comunicación:** El ámbito de comunicación es el entorno en que se desarrolla un elemento software. Como vemos, este ámbito se ha ido agrandando a medida que hemos evolucionado. Primero un ámbito intra-aplicación, con la programación estructurada y la programación orientada a objetos. Los componentes dan un paso más en el entorno distribuido y su ámbito es el de inter-aplicaciones, es decir entre aplicaciones pero dentro de una misma empresa y en un mismo entorno de ejecución en una tecnología concreta. Y con los servicios se agranda su alcance potencial a aplicaciones entre empresas y/o diferentes entornos de ejecución.

Hasta ahora, hemos visto la evolución de la arquitectura de software hasta los servicios y hemos introducido el concepto de servicio, haciendo una comparación con las arquitecturas predecesoras. Es el momento pues, de profundizar en el concepto de servicio dentro del marco SOA.

6. SERVICIOS

SOA es una arquitectura orientada a servicios, por lo tanto, la pieza fundamental sobre la cual se sustenta SOA es el servicio. Ya hemos definido previamente que un servicio es una unidad de trabajo que acepta mensajes de entrada y produce mensajes de salida. Otra definición posible es la elaborada por el organismo W3C (*World Wide Web Consortium*) que define un servicio como un recurso abstracto que representa una capacidad de realizar tareas que forman una funcionalidad coherente desde un punto de vista de los proveedores y de los consumidores [10]. Como vemos, esta descripción es bastante abstracta y no entra en detalle sobre la tecnología usada. Ahora bien, independientemente de la tecnología empleada, podemos determinar ciertas características de los servicios que serán comunes. Podemos determinar qué tipos de servicios hay en función de la lógica de negocio que implementan, como se relacionan y comunican los servicios entre sí, y cómo se diseñan. Estos puntos son los que detallaremos a continuación.

6.1 *TIPOS DE SERVICIOS*

Los servicios pues, encapsulan unidades con cierta lógica, estos servicios son autónomos, auto-contenidos (como los componentes) y no están aislados unos de otros ya que pueden comunicarse entre sí. El alcance de la lógica representada por el servicio puede variar [7]. Para ejemplificar esto, podemos tomar como referencia un proceso de negocio. Un **proceso de negocio** es un conjunto de tareas relacionadas lógicamente llevadas a cabo para lograr un resultado de negocio definido, donde el negocio son las actividades que producen valor para una organización, sus inversores o sus clientes [11]. La Figura 14 ejemplifica un posible proceso de negocio mediante un diagrama. En este diagrama podemos observar cómo se reflejan pasos dentro del proceso (*process step*) o también llamados actividades. Una actividad es una parte del proceso de negocio que no vale la pena descomponer y que no incluye ninguna toma de decisión. Un ejemplo podría ser realizar una factura. También se observan subprocesos. Un subproceso es una parte de un proceso de mayor nivel que tiene su propia meta, propietario, parámetros de entrada y de salida [11].

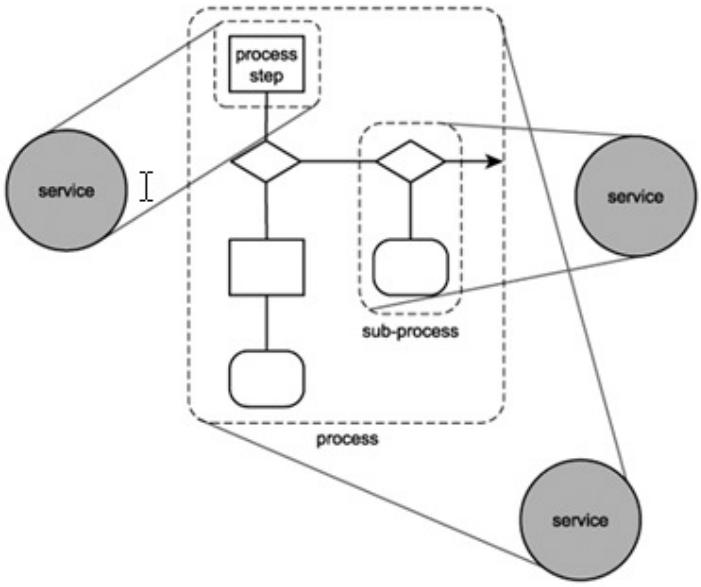

Fig. 14. Diferentes encapsulamientos de lógica por parte de un servicio. [7]

Un servicio pues, puede representar tanto actividades, como subprocesos o un proceso entero.

Existe una clasificación en tres tipos de servicio en función del tipo de lógica que encapsulan, su reusabilidad potencial, y a que dominio de la empresa pertenece la lógica representada por el servicio [8] (Figura 15):

- Servicios de entidad (*Entity Services*)
- Servicios de Tarea (*Task Services*)
- Servicios de utilidad (*Utilities Services*)

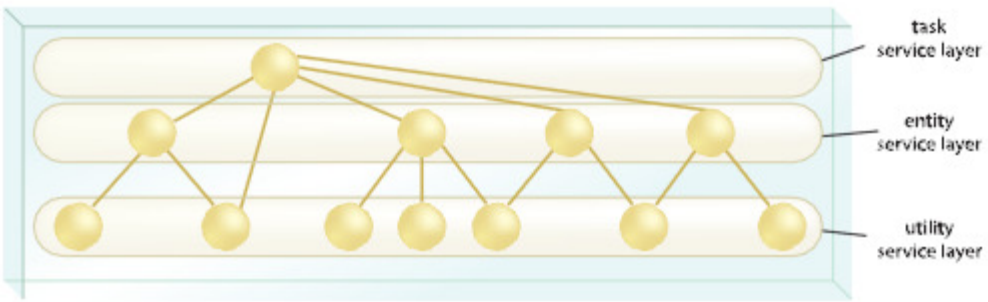

Fig. 15. Capas de servicios (representados como esferas) en función del tipo de servicio y la relación entre ellas. [8]

Veamos en detalle cada tipología:

Entity Services

Hacen referencia a las entidades de negocio usadas en la empresa. Por ejemplo, cliente, empleado (Figura 16) o pedido. Estas entidades serán usadas por los procesos de negocio de un nivel superior y dado que son entidades de negocio comunes en la empresa y por lo tanto involucradas en muchos procesos de negocio, son entidades con una alta reusabilidad.

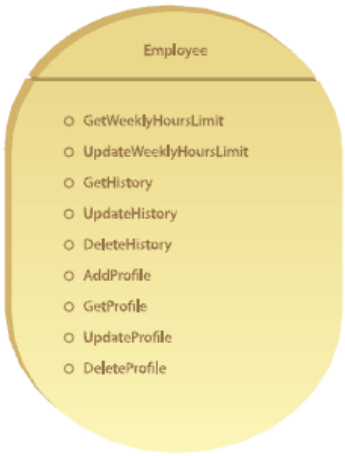

Fig. 16. Entidad empleado. [8]

En la Figura 16 se observan los métodos disponibles para la entidad Empleado. Se puede observar que siguen el patrón de acceso a datos CRUD (*Create, Read, Update, Delete*).

Task Services

Estos servicios (Figura 17) encapsulan todo un proceso de negocio, tal y como se ha visto en la Figura 15. Por lo tanto, estos servicios utilizarán los servicios de entidad en una secuencia de pasos para completar una tarea específica. Podríamos tener un proceso de negocio que trabajase con un historial de facturas, como por ejemplo un cálculo de stock disponible, este proceso de más complejidad requerirá el acceso a las entidades Factura entre otras entidades. Al ser el proceso más concreto que un servicio de entidad, lo servicios de tarea o de proceso de negocio tendrán una reusabilidad menor.

Fig. 17. *Un servicio de tarea o Task Service.* [8]

Utility Service

El tercer tipo de servicio (Figura 18) se refiere como su nombre indica a utilidades comunes a ser usadas tanto por *task services* como por *entity services*. A diferencia de los tipos explicados anteriormente, estos servicios no se centran en la lógica de negocio, más bien son servicios que ofrecen funcionalidades de apoyo, como por ejemplo sistemas de *logging* (autenticación en las aplicaciones), sistemas de notificaciones o de tratamiento de excecpciones.

Fig. 18. *Utility Service* que ofrece utilidades de transformación. [8]

6.2 *COMPOSICIÓN DE SERVICIOS*

Cuando hemos hablado de los *task service* hemos visto que estos usaban otros servicios para realizar su cometido. A este hecho se le conoce como composición de servicios y se simboliza mediante la unión de los servicios (Figura 19).

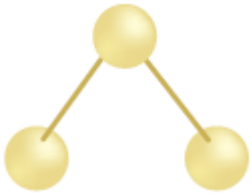

Fig. 19. Composición de servicios. [8]

Nótese que los servicios tienen la capacidad de participar en varias composiciones de servicios como se aprecia en la figura 15.

6.3 *INVENTARIO DE SERVICIOS*

Un inventario de servicios es una colección de servicios alojados en un repositorio controlado. A partir de él, se tendrá acceso a los servicios y poder realizar composiciones con ellos, tal y como se aprecia en la Figura 20. Una plataforma que soporte la arquitectura orientada a

servicios debe proveer los mecanismos necesarios para dar soporte tanto al inventario de servicios como a la posible composición de servicios.

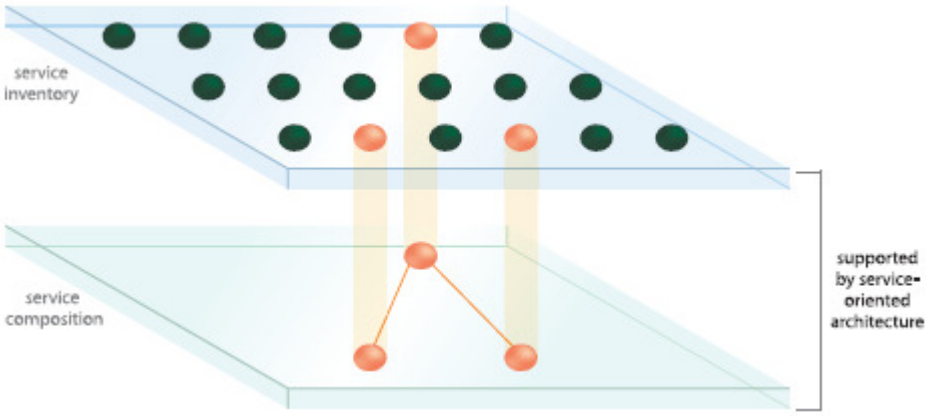

Figura 20. Inventario de servicios. [8]

6.4 *DESCRIPCIÓN DE SERVICIO*

Para que un servicio pueda ser reconocido (descubierto) para relacionarse con otro servicio o programa es necesario que tenga un documento que contenga información sobre él. A este documento se le llama descripción de servicio (Figura 21). Se puede observar como el servicio A tiene acceso a la descripción del servicio B cosa que le permitirá obtener toda la información necesaria para poder comunicarse con el servicio B.

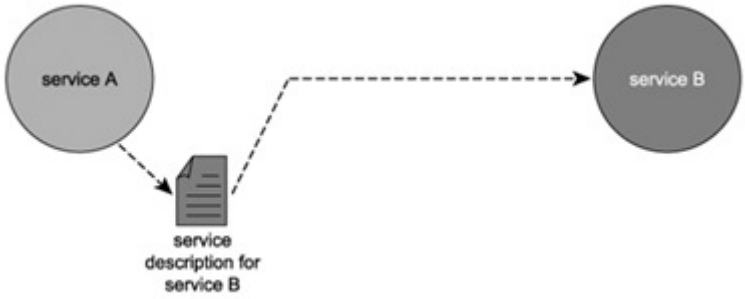

Fig. 21. Documento de descripción de servicio. [7]

La descripción de servicio contiene información relativa al servicio, como por ejemplo el nombre del servicio, las operaciones de las que consta y los datos de entrada y salida requeridos para dichas operaciones. Esta explicación puede recordar a la definición de contrato expuesta anteriormente, la diferencia entre un contrato y una descripción de servicio es que el contrato incluye una o varias descripciones de servicio, y a parte de ellas, puede contener otros documentos adicionales como se verá más adelante.

La descripción de servicio es importante debido a que facilita una relación débilmente acoplada entre los servicios, ya que solamente se expone del servicio la mínima información necesaria para establecer la comunicación, sin entrar en detalles técnicos de cómo está implementado el servicio.

6.5 *COMUNICACIÓN ENTRE SERVICIOS*

La comunicación entre servicios se realiza mediante mensajes, un mensaje es una unidad de comunicación autónoma. Se entiende por autónoma en que el mensaje mantiene el control de sí mismo (es auto-gobernado) y por lo tanto, el servicio que envía el mensaje pierde el control sobre él una vez enviado (Figura 22).

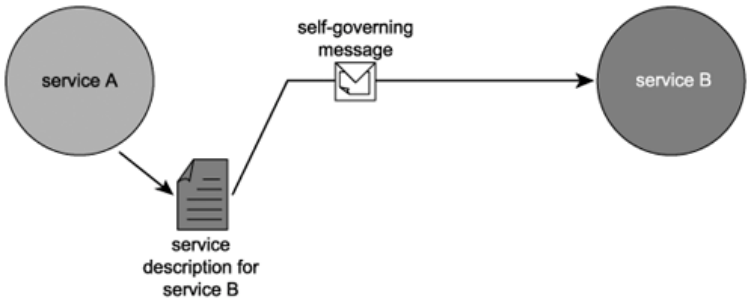

Fig. 22. Mensaje entre dos servicios. [7]

La característica de autonomía es importante tanto en servicios como en los mensajes. Se desarrollará este concepto en apartados posteriores.

6.6 *DISEÑO DE LOS SERVICIOS*

Hasta este punto hemos visto los servicios, las descripciones de servicio y los mensajes. Estos tres componentes son la base de la arquitectura orientada a servicios. Como hemos visto en la introducción histórica a la arquitectura de software en cada etapa de la evolución, los patrones de diseño se basaban en la idea de separación de conceptos. Esta idea lo que pretende es que para resolver un problema de relativa dificultad, lo mejor es dividir el problema en problemas más pequeños ya que así serán más fácil de resolver. Por ejemplo, en la programación orientada a objetos habían una serie de principios o patrones que la definían (encapsulación, abstracción, herencia, etc.), así, de la misma forma, a nivel de servicios también existe una serie de normas que rigen su diseño y que permiten resolver de forma separada las diferentes facetas de los servicios. Estos principios se agrupan en lo que se llama *Service-orientation* (Orientación a servicios) y forman la base en la que se sustenta SOA. En el próximo capítulo se entra en detalle en dichos principios.

7. SERVICE-ORIENTATION

Service-orientation es un paradigma de diseño de lógica basada en servicios que consta de una serie de principios. No existe una lista oficial, pero sí que hay unos principios que se asocian normalmente a la orientación a servicios [7]. Los enumeramos a continuación con una breve descripción y posteriormente se desarrollarán con más detalle.

1. **Contratos de servicio estandarizados**: un servicio debe expresar su propósito, sus capacidades, interfaz de entrada/salida y como están definidos sus tipos de datos para asegurar que el servicio definido está optimizado, es granular y estandarizado.

2. **Servicios con bajo acoplamiento**: hace referencia al nivel de dependencia entre servicios, el proveedor y el consumidor. Cuanto menos acoplamiento, mayor independencia del servicio y su posterior evolución.

3. **Abstracción**: se deben ocultar al máximo los elementos internos de un servicio. Un servicio debe verse como una caja negra, limitándose a lo definido en su contrato. Gracias a la abstracción, conseguiremos servicios adecuadamente desacoplados como se ha comentado en el punto anterior ya que se limita la parte expuesta de la lógica del servicio a las operaciones permitidas con sus parámetros y valores de retorno.

4. **Reusabilidad**: la encapsulación de la lógica de negocio en servicios permite que éstos sean reaprovechados por otros servicios y/o aplicaciones.

5. **Autonomía**: Un servicio tiene el control de su lógica de negocio y de su entorno de ejecución.

6. **Sin estado**: El servicio no puede depender de una gran información de estado, solamente necesita para operar sus parámetros de entrada para garantizar la disponibilidad y escalabilidad del servicio.

7. **Capacidad de ser descubierto**: gracias a la información relativa al servicio detallada en su descriptor de (sus metadatos), un servicio debe ser posible encontrarlo en un repositorio de servicios.

8. **Composición**: Define la capacidad de un servicio para formar parte de un servicio más complejo de más alto nivel. Esto provoca que la implementación de nuevos servicios se reducirá al mínimo ya que los nuevos servicios crecen a partir de otros ya existentes.

Estos son los ocho principios con los que se considera que si un servicio los cumple, éste se puede catalogar como *service-oriented*. Hay un aspecto que no se ha mencionado en los

principios pero que es intrínseco en todos ellos. Esta característica común es la de interoperabilidad. Se entiende por interoperabilidad la habilidad de interacción entre partes de un sistema que están construidos en diferentes tecnologías. Un ejemplo claro sería la comunicación entre sistemas que operan bajo tecnología Java y .NET. Los servicios *service-oriented*, es decir, los servicios que cumplen los principios anteriormente mencionados contribuyen de por sí a esta interoperabilidad. A continuación veremos en detalle dichos principios y una vez explicados, indagaremos con más detalle en el concepto de interoperabilidad y cómo ésta es alcanzada mediante los principios.

7.1 *CONTRATOS DE SERVICIOS ESTANDARIZADOS*

Los contratos (Figura 23) son la base de la orientación a servicios, ya que gracias a ellos se permite la relación entre servicios. Los contratos definen los servicios, las operaciones y los mensajes, expresando así su propósito y sus capacidades.

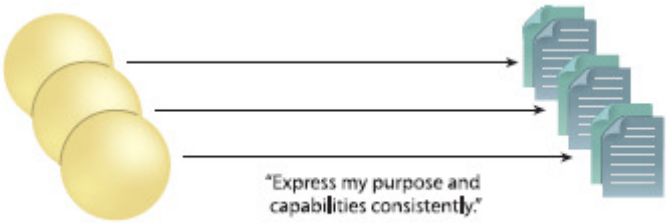

Fig. 23. Finalidad de un contrato de servicio. [8]

Como se ha mencionado anteriormente, un contrato incluye uno o varios documentos de descripción de servicio. Encontramos los documentos de descripción de servicio técnicos y los que no son técnicos [8]. Los documentos técnicos son los que son consumidos en tiempo de ejecución, es decir, aquellos que tienen la información necesaria para que se ejecute el proceso en que los servicios están involucrados, y los que no son técnicos, son documentos que aportan información de soporte adicional a la información técnica. Veamos en detalle cada tipo.

7.1.1 Documentos técnicos

En las descripciones técnicas encontramos las operaciones permitidas por el servicio (sus capacidades), los mensajes necesarios para intercambiar información a través de las operaciones y los modelos de datos que soportan los mensajes. Por ejemplo, si tenemos un servicio relativo a una factura, en su descripción técnica podríamos tener una operación que sea ObtenerFactura. Esta operación, tendrá dos mensajes, un mensaje de petición de la factura ObtenerFacturaPeticion y un mensaje que devolverá la respuesta ObtenerFacturaRespuesta. Ambos mensajes contendrán unos tipos de datos asociados, en

el mensaje de petición viajará el identificador de la factura para que el servicio sepa qué factura tiene que ir a buscar. Una vez la encuentre, construirá en el mensaje de respuesta otro tipo de dato que será la factura con toda su información. Es importante que estos tipos de datos estén estandarizados ya que es muy probable que la información de una factura esté presente en diferentes servicios. Por lo tanto, en nuestro ejemplo, en los datos usados en el mensaje de petición ObtenerFacturaPeticion definiríamos el identificador de la factura como NumeroFactura y no como ObtenerFacturaPeticionTipo. De esta forma, se estandariza el modelo de datos referente al número de factura y se puede reutilizar en otros servicios u operaciones que también lo usen.

Otra posible descripción técnica son las descripciones de políticas. Las políticas son un conjunto de condiciones que determinan si se debe proveer acceso a las operaciones o no, tal y como se muestra en la Figura 24.

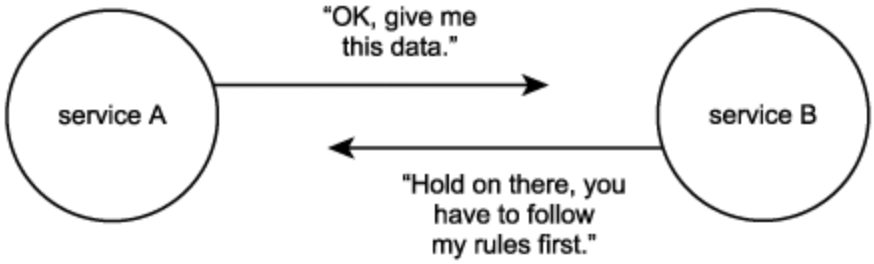

Fig. 24. Uso de políticas en los servicios. [7]

Como ejemplo de políticas encontramos reglas que hacen referencia al cifrado de mensajes con un determinado algoritmo, o que los mensajes deban viajar signados para así asegurar la autoría de los mensajes.

Las políticas también están sujetas a la estandarización, con lo cual podemos tener definiciones de políticas separadas de los servicios y por lo tanto susceptibles de ser reutilizadas.

7.1.2 Documentos no técnicos

Existen otros documentos a parte de los documentos técnicos que pueden incluirse en un contrato. El ejemplo más común es el uso del acuerdo de nivel de servicio o *service level agreement* (SLA). El SLA asocia características de calidad de servicio o *quality of service* (QoS) al contrato. Estas características pueden ser referentes a la disponibilidad, accesibilidad o rendimiento asociadas a los servicios, por ejemplo, proveer una garantía de tiempos de respuesta de los servicios y sus tiempos medios, estadísticas asociadas a los consumos (tipos de consumidores o número de accesos) y una programación horaria de garantía de disponibilidad del servicio.

7.1.3 Versionado

Cuando un servicio lleva ya un tiempo en uso en un entorno empresarial, lo más normal es que haya aumentado sus relaciones con otros servicios consumidores. Esto tiene una parte positiva y una posible parte negativa [8]. La parte positiva es que el uso intensivo del servicio demuestra que éste está teniendo un alto índice de reusabilidad. La parte negativa podría ser que en cierta manera se está generando un acoplamiento a nivel empresarial con el servicio, dado que cada vez más servicios dependen del contrato del servicio. Es aquí donde un buen diseño estandarizado del servicio minimiza el número de futuras versiones del contrato y por lo tanto, minimiza los cambios necesarios en los servicios que consumen dicho contrato.

7.1.4 Dependencias tecnológicas

Los contratos por naturaleza deben ser implementados con una tecnología en concreto, es por ello que dependen de dicha tecnología. Esto quiere decir que si la tecnología queda obsoleta o es actualizada, los contratos deberán modificarse acorde a los nuevos cambios, en el caso de que lleguen a influenciar en el diseño del contrato.

Una vez vistos los contratos en detalle, podemos analizar en qué manera influyen en otros principios. El uso de contratos por ejemplo, minimiza las dependencias con los demás servicios ya que éstas se concentran en el contrato evitando así dependencias entre las implementaciones de los respectivos servicios, por lo tanto, estamos favoreciendo el principio de desacoplamiento. Así mismo, limitando la información que aparece en el contrato a su propósito, capacidades y requisitos de interacción, dejando de banda la información no esencial, conseguimos que el servicio tenga el grado de abstracción requerido. En cuanto a la reusabilidad, el objetivo es conseguir servicios agnósticos, es decir, que sean lo más genéricos posibles para así aumentar su reusabilidad potencial. Esto se consigue mediante un buen diseño de los contratos, definiendo adecuadamente sus operaciones. Hemos visto con anterioridad que los servicios pueden alojarse en un repositorio de servicios. Aunque el servicio esté en el repositorio, es necesario que se defina claramente el propósito del servicio, incluyendo anotaciones descriptivas. De esta forma pues, se favorece así el principio de descubrimiento de los servicios.

7.2 *SERVICIOS CON BAJO ACOPLAMIENTO*

El acoplamiento, a nivel general, se refiere a las dependencias que se generan cuando cualquier cosa está conectada con otra [8]. Tendremos un alto acoplamiento o estrecho (en inglés *tight*) cuando exista un alto grado de dependencia, por el contrario tendremos bajo acomplamiento o flojo (en inglés *loose*) cuando la dependencia sea escasa. Como se puede observar, los términos alto o bajo tienen cierta ambigüedad y no es algo que se pueda cuantificar con exactitud.

En los servicios, existen dos vías principales de acoplamiento, tal y como se muestra en la Figura 25. Podemos hablar de acoplamiento en los servicios refiriéndonos a su relación con otros servicios, es decir, de forma externa entre el contrato del servicio y los consumidores del servicio, y también puede haber acoplamiento de forma interna, es decir, un acoplamiento entre el contrato del servicio y su implementación.

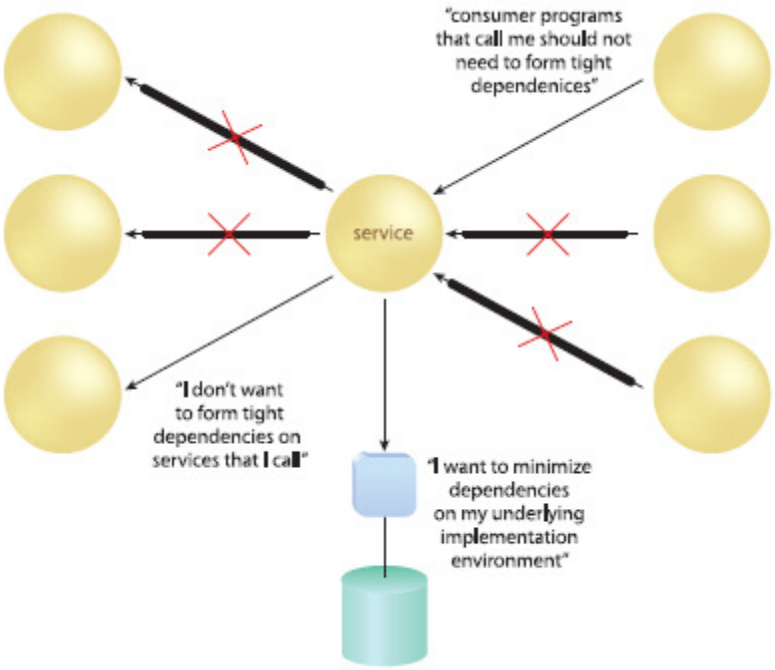

Fig. 25. Acoplamiento en los servicios. [8]

La implementación del servicio es la lógica de negocio del servicio (representado con un cuadrado) y el acceso a los datos necesarios por dicha lógica (representado con un cilindro). Veamos pues cada tipo de posible acoplamiento.

7.2.1 Acoplamiento entre los consumidores y el contrato (*consumer-to-contract coupling*)

Como no es posible obtener una independencia total, ya que esto implicaría que no hay relación entre los servicios, se obtiene la mínima dependencia generada entre servicios usando los contratos, tal y como se ha explicado en el principio de contrato estandarizado. El grado de acoplamiento vendrá determinado por el diseño del contrato, conseguiremos un bajo grado con operaciones bien definidas y estandarizadas que favorezcan servicios agnósticos como también se ha indicado en el principio de contrato.

7.2.2 Acoplamiento entre el contrato y la lógica del servicio

Otro tipo de acoplamiento es el generado entre el contrato y la lógica implementada por el servicio. Este tipo de acoplamiento puede ser recíproco, es decir que puede haber acoplamiento entre el contrato y la lógica y viceversa, entre la lógica y el contrato (Figura 26).

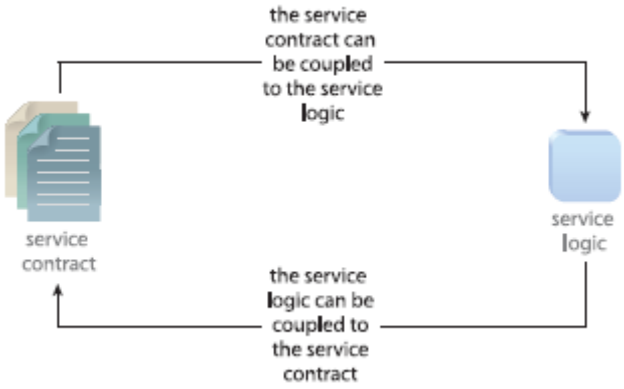

Fig. 26. Acoplamiento entre el contrato del servicio y la lógica del servicio. [8]

7.2.3 Acoplamiento lógica-contrato (*logic-to-contract coupling*)

Este tipo de acoplamiento (Figura 27) se produce cuando se construye el contrato del servicio antes de su lógica. Esta estrategia se denomina *contract-first*, es decir, primero el contrato y posteriormente se generará la lógica en función de dicho contrato. Esta estrategia es muy efectiva debido a que aseguramos que el diseño estándar definido en el contrato es incorporado a la lógica de negocio del servicio [8].

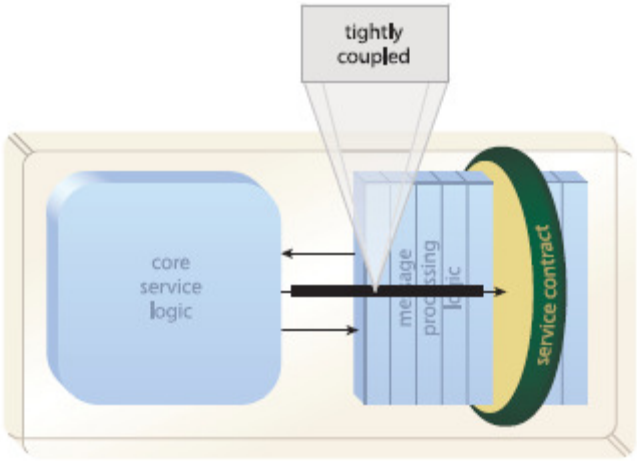

Figura 27. Acoplamiento *logic-to-contract*. [8].

Aunque en la Figura 27 se indica que el acoplamiento es alto, éste es considerado un buen acoplamiento, debido al hecho antes mencionado de que trasladamos el diseño elaborado en el contrato a la lógica de negocio. Por otra parte, la lógica no está completamente acoplada al

contrato debido a que se podría reemplazar en el futuro la lógica del servicio sin afectar a los consumidores que han establecido relaciones de dependencia con el contrato del servicio, ya que no haría falta cambiar el contrato [8].

7.2.4 Acoplamiento contrato-lógica (*contract-to-logic coupling*)

Este tipo de acoplamiento es el caso opuesto al anterior, es decir, habiendo una lógica de servicio ya existente, se genera el contrato a partir de ella (Figura 28). Podría ser el caso de la generación de una capa de servicios a partir de componentes distribuidos tal y como hemos estudiado antes en la Figura 10. El contrato se generaría a partir de la lógica ya implementada por los componentes.

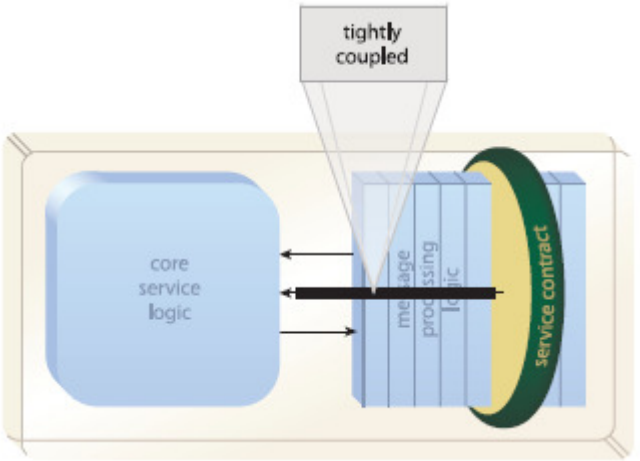

Fig. 28. Acoplamiento *contract-to-logic*. [8]

A diferencia del caso anterior, la lógica no depende del contrato, porque si ésta cambia, se debe generar un nuevo contrato. Como se observa pues, quien sí tiene un alto acoplamiento con la lógica es el contrato ya que su diseño depende completamente de la lógica. Esto implica que la estrategia *contract-to-logic* es un acoplamiento no deseado ya que el cambio de contrato puede causar problemas a los consumidores que dependen de dicho contrato, debiendo de realizar nuevos cambios cada vez que se realizan cambios en la lógica del servicio para adaptarlos a los nuevos cambios aparecidos en el contrato del servicio.

Retomando el ejemplo anteriormente mencionado sobre los servicios elaborados a partir de componentes, es posible que se genere un tercer tipo de acoplamiento en función de la tecnología utilizada en la implementación de estos componentes. Esto crea el acoplamiento contrato-tecnología que se explica a continuación.

7.2.5 Acoplamiento contrato-tecnología (*contract-to-tecnology coupling*)

Los contratos de los servicios elaborados a partir de componentes con tecnología propietaria (por ejemplo DCOM o CORBA) es posible que generen un alto acoplamiento con la tecnología en cuestión (Figura 29). Esto es debido a que estas tecnologías pueden llegar a imponer características específicas en los contratos [8].

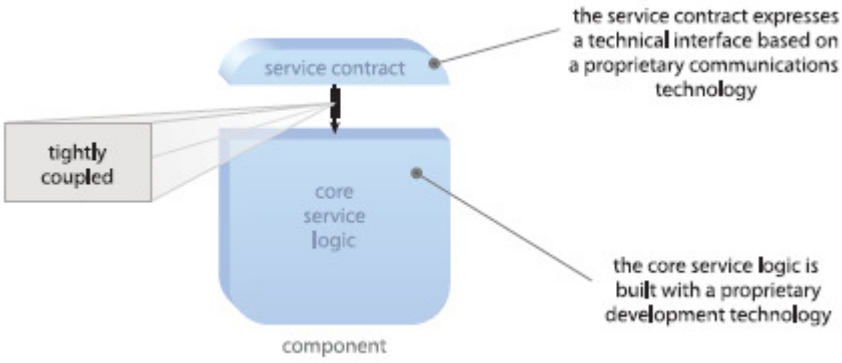

Fig. 29. Alto acoplamiento en componentes con tecnología propietaria. [8]

Un ejemplo de este acoplamiento sería la imposición por parte de la tecnología en todos los contratos de un determinado protocolo de comunicación que podría ser propietario o no considerado como un estándar. Esto implicaría que los potenciales consumidores deberían soportar dicho protocolo para poder usar el servicio [8].

El concepto de bajo acoplamiento va íntimamente relacionado con el de la abstracción, ya que el bajo acoplamiento se consigue con niveles adecuados de abstracción en el diseño del contrato. A continuación veremos en detalle dicho principio.

7.3 *ABSTRACCIÓN EN SERVICIOS*

La abstracción (Figura 30) se centra en hacer visible esas partes del servicio absolutamente necesarias para la comunicación con otros servicios, dejando de banda cualquier otro tipo de información.

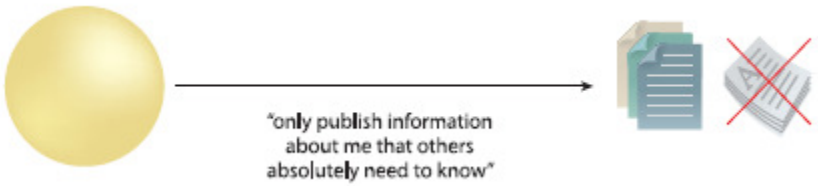

Fig. 30. Abstracción en los servicios. [8]

El propósito de ocultar la información no relevante del servicio, como su implementación y detalles de su diseño, es el de limitar el acoplamiento de los consumidores con el servicio, como hemos visto en el apartado anterior con el tipo de acoplamiento *consumer-to-contract*. De esta forma, un servicio se convierte en lo que comúnmente se llama una caja negra o *black box*, ya que no se ve como está constituida por dentro. El hecho de incluir información de más

hará que el acoplamiento vaya aumentando. Otra ventaja de la abstracción es que gracias a ella el propietario del servicio puede realizar la implementación como considere necesario, sin tener que pensar si dicha implementación puede afectar de alguna forma a los consumidores del servicio.

Cuando hablamos de abstraer información u ocultarla, nos referimos a información relativa al servicio, por lo tanto hablamos de meta-datos o meta-información. El próximo paso será detallar qué tipo de información es susceptible de ser abstraída en el servicio.

7.3.1 TIPOS DE META-INFORMACIÓN

En la Figura 31 se detallan los tipos de meta-información relativas a diferentes facetas de los servicios. A continuación explicaremos brevemente cada tipo.

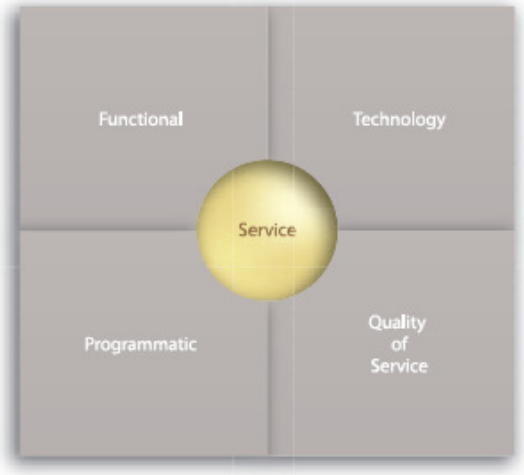

Fig. 31. Tipos de meta-información de un servicio. [8]

Información tecnológica

Es la información relativa a la implementación técnica del servicio. Conviene ocultarla ya que podemos tener un servicio implementado en una cierta tecnología y en un futuro se quiera implementar de nuevo dicha lógica en otra tecnología diferente. Esto se puede hacer sin problemas gracias a la abstracción aportada por el contrato. El cambio de tecnología será transparente al consumidor del servicio. Esto quiere decir que seguirá llamando al servicio sin darse cuenta de que ha habido un cambio interno en el servicio. Al consumidor se le proporciona la información tecnológica para hacer las llamadas al servicio (las operaciones de las que consta el servicio) pero no el cómo están realizadas.

Información funcional

La información funcional son los meta-datos que describen las capacidades de un programa. Hemos dicho que al consumidor del servicio se le proporciona la información para saber qué operaciones contiene el servicio, pero estas son solo un subconjunto de la funcionalidad que puede hacer internamente el servicio. Por lo tanto, se publica solamente la funcionalidad requerida mientras toda la información funcional restante permanece oculta al consumidor. Por ejemplo, en un servicio de consulta de facturas, podemos exponer al consumidor la funcionalidad de devolver una factura a partir de un identificador de factura. Sin embargo, internamente el servicio tendrá otras funcionalidades como por ejemplo obtener información de cada producto presente en la factura, pero esta funcionalidad no se quiere mostrar al consumidor y permanecerá oculta.

Información de lógica de programación

Asociada a la función tecnológica, la información de la lógica de programación es relativa a cómo la aplicación está construida, es decir, algoritmos utilizados, manejo de excepciones en el código (qué hacer cuando se dan ciertos tipos de errores) y lógica de las funciones utilizadas internamente por el servicio. Toda esta información es debidamente ocultada ya que no tiene sentido que el consumidor del servicio tenga acceso a ella.

Información de calidad de servicio

La información relativa a la calidad del servicio puede ser publicada al consumidor, por ejemplo, con el documento SLA explicado anteriormente en el principio de contrato estandarizado. Por ejemplo, podemos detallar en este documento que el servicio tiene cierta disponibilidad horaria, pero a la vez, se puede ocultar al consumidor no publicando los errores que han habido durante este periodo de disponibilidad.

Como hemos comentado, los principios de bajo acoplamiento y abstracción van estrechamente relacionados. Hay pero otros principios que reciben la influencia de la abstracción. Por ejemplo, los principios de reusabilidad, capacidad de ser descubierto y composición [8]. En ellos, la abstracción actúa como regulador y cuando más meta-información esté publicada más se potenciarán dichos principios. Un servicio con más información será potencialmente más descubierto y por lo tanto más reutilizado o incluido en una composición de servicios. Entraremos en detalle en el principio de reusabilidad en el próximo apartado.

7.4 *REUSABILIDAD DE LOS SERVICIOS*

La reusabilidad es la finalidad que persigue la computación orientada a servicios. Los servicios están para proporcionar una funcionalidad pero la idea está en que el servicio sea usado para el máximo número de propósitos (Figura 32).

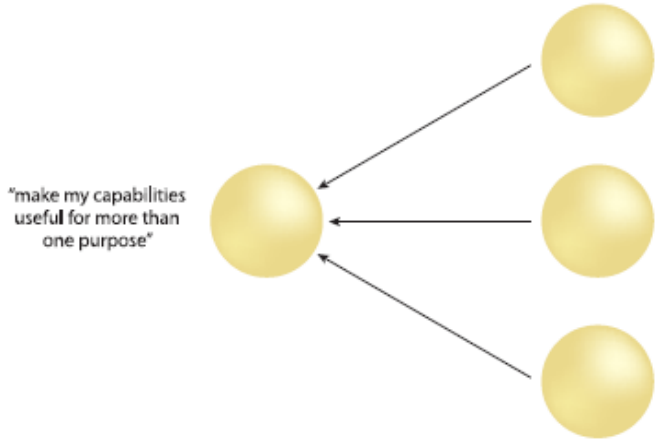

Fig. 32. Propósito de la reusabilidad. [8]

En los subsiguientes apartados vamos a entrar en detalle en el concepto de reusabilidad.

7.4.1 Programas de único propósito y multipropósito

En el apartado de la introducción a la historia de la arquitectura de software ya se introdujo el concepto de reutilización, indicando que era una buena práctica de programación construir los programas con la mente puesta en un futuro uso del programa diferente para el que se creó en un principio. Esto es lo que se llama programas multipropósito, en cambio, el desarrollo de programas limitados a un uso en concreto se les llama programas de único propósito.

Para ejemplificar todo esto, podemos utilizar el siguiente ejemplo básico [8]. Como programa de un único propósito podemos tener un contador de stock que será utilizado por el usuario encargado del almacén (Figura 33).

Fig. 33. Ejemplo de programa de un único propósito. [8]

Este contador de stock puede incrementar o decrementar el número de ítems del almacén, su interfaz gráfica está orientada a ítems de stock, pudiendo introducir una cantidad y añadirla o restarla, y visualizar el número de ítems total. También se ha construido la aplicación teniendo en mente solamente un ámbito y un tipo de usuario en concreto. Los programas multipropósito van un paso más allá, requiriendo más esfuerzo en la etapa de diseño, donde hay que pensar posibles escenarios diferentes en el que pueda llegarse a utilizar el

programa, y a qué diferentes tipos de usuarios puede dirigirse. Así, el ejemplo anterior en su versión multipropósito se convierte en el mostrado en la Figura 34.

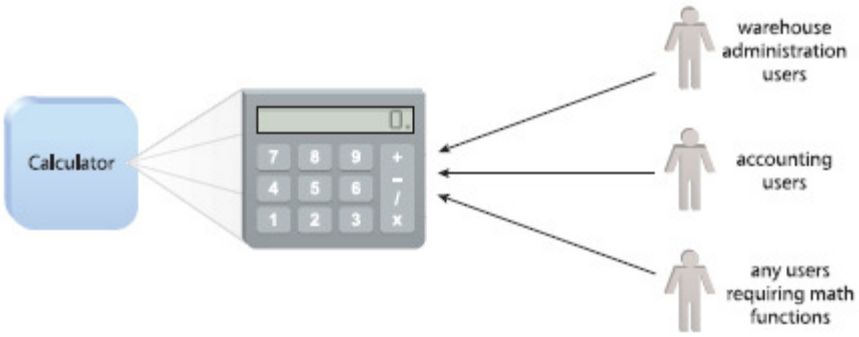

Fig. 34. Ejemplo de programa multipropósito. [8]

Ahora, el contador se ha convertido en una calculadora genérica, su interfaz ya no está enfocada solamente a los Ítems, con lo cual, a aumentado el tipo de usuarios potenciales del programa.

Es importante pues, tener en mente un escenario de programa multipropósito cuando se diseñen servicios, ya que es la finalidad de éstos. Hay otros aspectos en los que afecta la reutilización, vistos en el siguiente apartado.

7.4.2 Objetivos de la reusabilidad

La reusabilidad de los servicios implica que se persigan otros objetivos relacionados [8]. Por una parte, se espera un alto retorno de la inversión dedicada en la creación del servicio, ya que posteriormente solamente habrá que usarlo, y cuantas más veces, mejor. También se pretende minimizar el tiempo de futuros desarrollos de automatización de procesos mediante la composición de servicios ya existentes y la creación de un repositorio con un alto porcentaje de servicios agnósticos para que sean potencialmente reusables en el mayor grado posible. Anteriormente, nos hemos referido en varias ocasiones al agnosticismo de los servicios, indicando que los servicios deben de cumplir un propósito genérico. Más detalladamente, un servicio es agnóstico cuando su lógica es independiente de los procesos de negocio a los que pertenece y también es independiente de la tecnología o plataformas de aplicaciones usadas [8]. Por lo tanto, el agnosticismo de los servicios propicia una mayor reusabilidad, ya que podrán ser usados en un número mayor de procesos de negocio o en entornos tecnológicos diferentes.

La reusabilidad, como hemos visto, es un punto clave en la computación con servicios. El hecho de que un servicio tenga un alto índice de reusabilidad, y por lo tanto una gran demanda, implica que el servicio debe soportar un alto grado de concurrencia y proporcionar un buen rendimiento. Para conseguir esto, el servicio debe tener el control de su lógica y de su entorno, lo que es el principio de autonomía. Punto que veremos a continuación.

7.5 *AUTONOMÍA DE LOS SERVICIOS*

La autonomía representa la habilidad de auto-gobierno en un servicio [8]. Un servicio es autónomo si tiene el control y la libertad de hacer cambios en su lógica de servicio sin la necesidad de intervención de terceros (Figura 35).

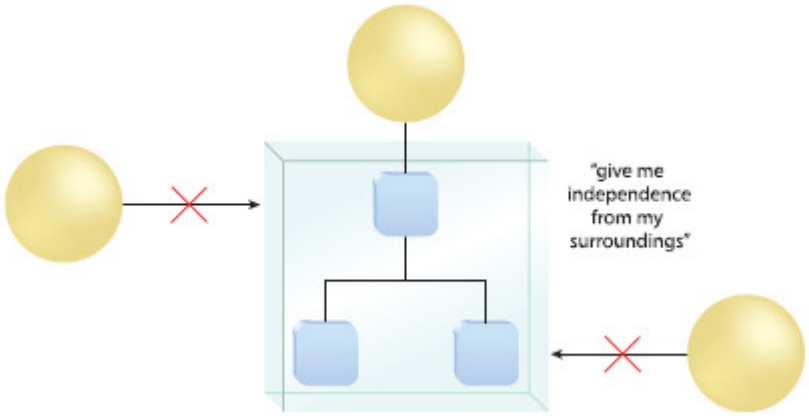

Fig. 35. Autonomía de un servicio. [8]

El aumento de autonomía significa también que las implementaciones de los servicios están más aisladas (*isolated*). Esto implica que aumenta tanto la seguridad del software como su predictibilidad, ya que se reducen las conexiones con otros elementos y se tiene un mayor control y conocimiento del comportamiento de la lógica.

Veamos a continuación los dos tipos de autonomía asociados a los servicios: autonomía en tiempo de ejecución y autonomía en tiempo de diseño.

7.5.1 Autonomía en tiempo de ejecución

Es el control del procesamiento de la lógica del servicio cuando éste es invocado y ejecutado. Los objetivos de incrementar la autonomía en tiempo de ejecución pasan por obtener un grado aceptable de rendimiento tanto a nivel de ejecución como de seguridad. Esto es importante cuando se hacen composiciones de servicios ya que el servicio que está compuesto de otros servicios, como por ejemplo un servicio de tarea ya visto, tiende a ser no autónomo ya que no tiene el control de la lógica de cada servicio y dependerá de los rendimientos individuales de cada servicio del cual está compuesto.

Otro objetivo del incremento de autonomía en tiempo de ejecución es aumentar el grado de predictibilidad. Es importante saber el comportamiento de la aplicación sobre todo cuando un servicio tiene alta reusabilidad y por lo tanto un alto acceso concurrente. La concurrencia es la situación que se da en un servicio cuando otros servicios acceden a él al mismo tiempo. Se

debe tener en cuenta en la lógica de negocio este fenómeno ya que se puede alterar sin quererlo el estado de las entidades utilizadas por la lógica de negocio. Para ejemplificar este problema, podemos tomar como ejemplo el servicio de calculadora detallado en la Figura 35. Supongamos que el usuario del almacén inicia un cálculo, y a la vez también lo hace el usuario cuentas. Podría darse el caso que el usuario de cuentas estuviera operando con el total producido por el usuario de almacén. Esta situación podría evitarse fácilmente cuando el primer usuario que accede, bloquea el uso de la calculadora y la libera cuando haya terminado los cálculos.

7.5.2 Autonomía en tiempo de diseño

La autonomía en tiempo de diseño es la capacidad de modificar el diseño del servicio por parte del poseedor del servicio. Los motivos que pueden llevar a ello son ajustar el servicio para dar soporte a una creciente demanda de su uso, es decir, escalar el servicio, hacer frente a nuevos requerimientos o incluso substituir la tecnología con la que está implementado el servicio si las nuevas necesidades lo requieren.

Debe recordarse el hecho que todas estas modificaciones de diseño están sujetas al contrato del servicio como bien se ha estudiado, es a partir de esta restricción y aceptando esta pérdida de control, en la que se tiene autonomía para realizar cambios sin que afecten a los consumidores que han establecido relaciones de dependencia con el servicio. Como también hemos visto, un buen contrato estandarizado, desacoplado y abstracto, que separe bien el contrato de su lógica dará pie a una libertad mayor para realizar cambios.

Hemos visto que la autonomía tiene relación con sus principios predecesores. A continuación entraremos en detalle al principio de sin estado. La autonomía favorece este principio también, ya que los servicios con un control sobre su lógica también tienen un mayor grado de control sobre su estado [8]. Hay que tener en cuenta pero, que solamente los servicios bien aislados y autónomos pueden favorecer un bajo grado de estado. Como ejemplo contrario a esto, podemos tener los servicios que actúan como *wrapper* de componentes legados. Ya hemos visto esta habilidad anteriormente, y también hemos visto que los componentes tienen un estado más rígido. Es por este motivo, que los servicios *wrapper* tendrán poco margen de maniobra sobre su estado, ya que internamente dependen de componentes.

7.6 SERVICIOS SIN ESTADO

Se entiende por estado una condición general sobre algo. En concreto, en el contexto del software, éste puede transitar por diferentes estados, normalmente porque está involucrado en la ejecución de una actividad [8]. Los servicios tienen dos estados primarios: activo y pasivo (o no activo). El estado activo indica que el servicio está operativo, en cambio, el estado pasivo indica que el servicio no está disponible. Dentro del estado activo, podemos identificar dos tipos de estado: con estado o *stateful* o sin estado o *stateless* (Figura 36).

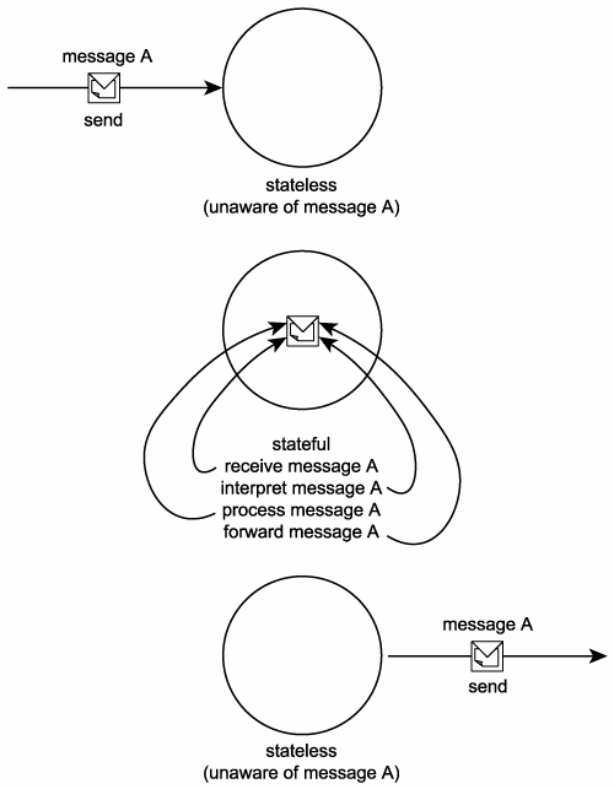

Fig. 36. Estados de un servicio. [7]

Por *stateless* se entiende que el servicio está activo pero no se encuentra procesando datos específicos de una tarea. A estos tipo de datos se denomina datos o información de estado. Por el contrario, stateful es cuando el servicio está procesando datos de estado. Siguiendo el proceso de la Figura 36, explicamos el proceso por los que el servicio cambia de estado. El servicio en un principio está *stateless* y activo. En un momento dado, el servicio recibe un mensaje. Entonces el servicio pasa a estado *stateful*, donde se trata el mensaje y se envía el mensaje de respuesta. Después del tratamiento de los datos de estado, el servicio vuelve a estado *stateless*.

El principio de sin estado pues, indica que debe reducirse al mínimo tanto la información de estado como el tiempo de procesamiento de dicha información. En efecto, a más cantidad y más tiempo de procesamiento, el servicio permanece más tiempo en estado *stateful* y esto provoca que no esté en estado *stateless* para poder recibir más peticiones.

Otra cuestión importante es que este principio promueve la reusabilidad y la escalabilidad ya que servicios sin estado tendrán un alto rendimiento y serán aptos para ser escalados en composiciones de servicios. Más adelante veremos el principio de composición, pero antes, veremos el principio de descubrimiento debido a que para realizar una composición, necesitaremos encontrar los servicios necesarios.

7.7 *DESCUBRIMIENTO DE SERVICIOS*

Anteriormente en el capítulo dedicado a los servicios, se ha introducido el concepto de inventario como un repositorio donde se alojan los servicios. Gracias a este repositorio podremos descubrir servicios.

La importancia del descubrimiento de servicios se refleja en el momento en que se quiere crear un nuevo servicio. Lo normal pues, es que antes de crearlo, investiguemos si ya existe un servicio que cumpla dicha funcionalidad y evitemos por una parte el desarrollo del nuevo servicio y por otra parte el duplicar funcionalidad en nuestro inventario de servicios. Para saber si la funcionalidad que necesitamos ya existe, deberá existir meta-información sobre los servicios que nos permitan averiguarlo. Esta meta-información está formada por el propósito del servicio, con una descripción detallada del servicio, las capacidades del servicio, es decir sus operaciones, y las limitaciones existentes en las operaciones ofrecidas por el servicio. Es importante que la meta-información sea de calidad, ya que en caso contrario no tendremos la información necesaria para determinar si un servicio nos sirve o no. Esta meta-información de cada servicio estará alojada en repositorio llamado registro de servicio o *service registry* y servirá como punto de entrada para la búsqueda de los servicios alojados en el inventario de servicios.

A continuación, detallamos el proceso de descubrimiento a partir de la Figura 37.

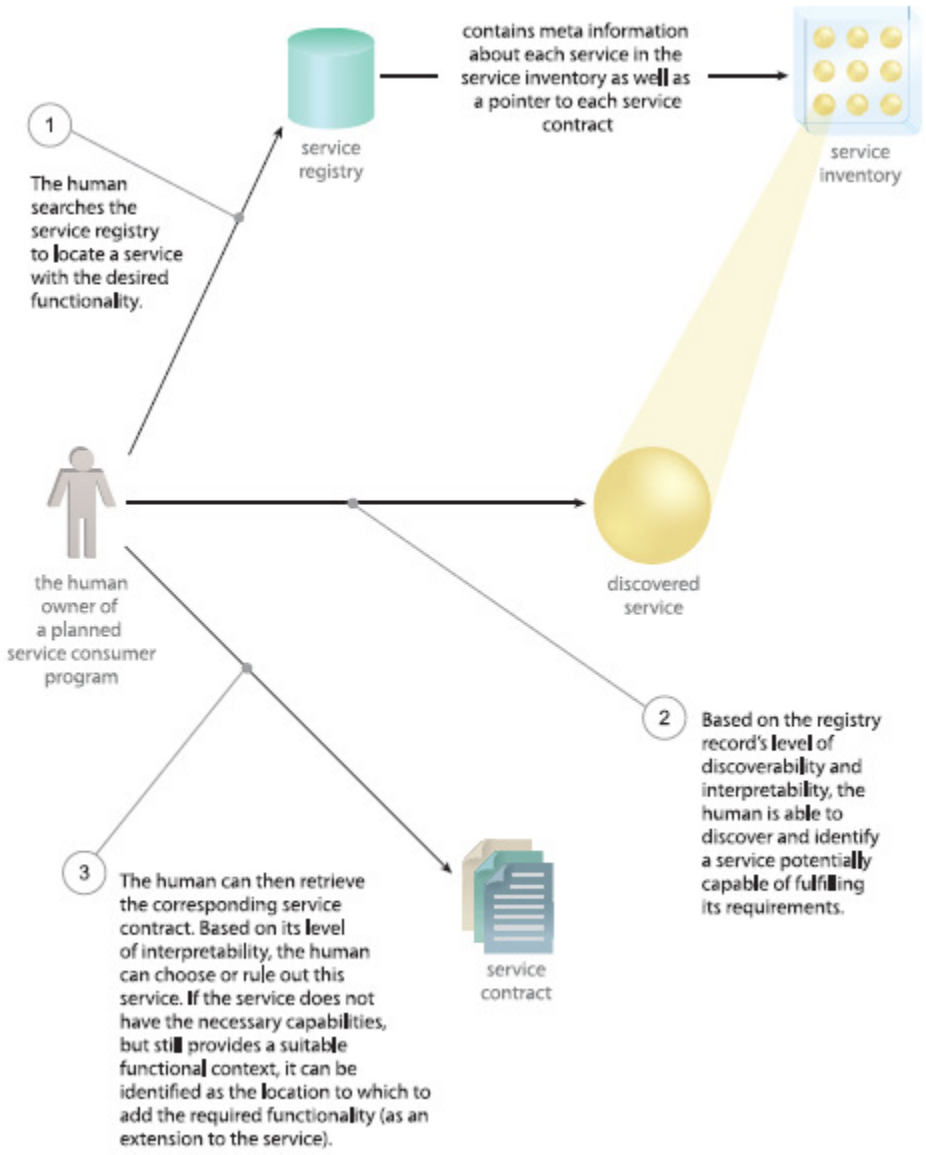

Fig. 37. Proceso de descubrimiento de servicios. [8]

En el primer paso, la persona que quiere crear un servicio busca en el registro de servicios si existe algún servicio con la funcionalidad deseada. En el paso 2, gracias a esta información y a la interpretación que hace la persona de dicha información, se descubre un posible servicio que podría servir para sus necesidades. En el paso 3, la persona obtiene el contrato de servicio y analizando el contrato con sus operaciones posibles, decide si el servicio es apto o lo rechaza. En el caso de que el servicio no ofrezca la funcionalidad requerida pero parece que ofrece una buena base, será el punto de partida para realizar el servicio necesitado.

A continuación detallaremos el último principio de *service-orientation*, el principio de composición. Gracias al descubrimiento, se facilitará la composición de servicios ya que podremos localizar con facilidad los servicios potenciales a utilizar.

7.8 *COMPOSICIÓN DE SERVICIOS*

La composición de servicios nos permite elaborar servicios complejos en combinación con otros servicios ya existentes (Figura 38). Hay que puntualizar, que un mismo servicio puede pertenecer a más de una composición de servicios.

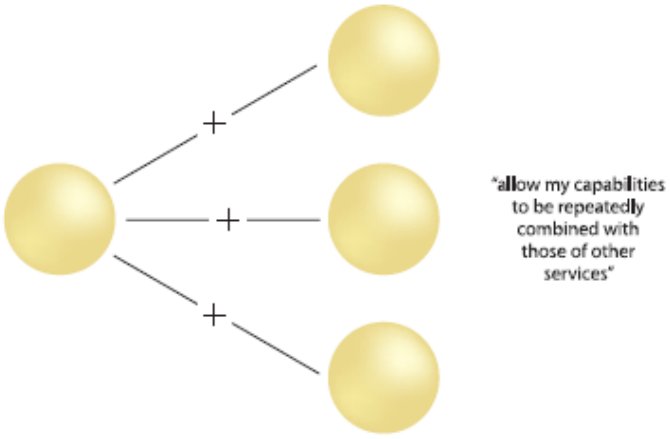

Fig. 38. Composición de servicios. [8]

La composición de servicios se basa en el mismo concepto introducido en la programación modular y orientada a objetos de la separación de cometidos o estrategia de divide y vencerás. Con ello, para la resolución de problemas complejos, una buena aproximación a su resolución es dividir el problema en otros de más pequeños que sean más fáciles de resolver. Por lo tanto, gracias a la composición de servicios se pueden utilizar o crear servicios más sencillos para luego componerlos y que den una solución al problema. Se puede observar como la composición de servicios tiene una relación estrecha con la reusabilidad, ya que promueve este principio como se ha indicado anteriormente.

A continuación entraremos en más detalle en la composición de servicios, introduciendo los conceptos de miembros de composición o controladores de composición. Los servicios adoptarán un rol u otro en función de cómo sus capacidades individuales participan dentro de la composición [8].

7.8.1 Controlador y miembro de composición

Se dice que un servicio es controlador de composición cuando se encuentra en el nivel más alto de la jerarquía construida en la composición. En este caso, la capacidad del servicio que se está ejecutando, internamente invoca las capacidades de los otros servicios involucrados en la composición, tal y como se muestra en la Figura 39.

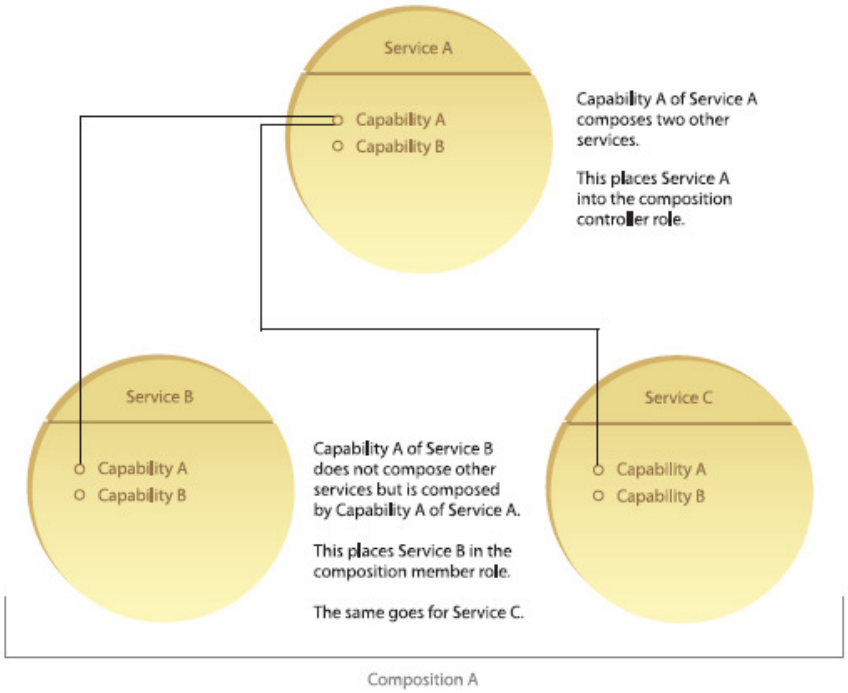

Fig. 39. Roles de composición. [8]

El servicio A actúa como controlador. Su capacidad A internamente utiliza las capacidades A de los servicios B y C. Como los servicios B y C no componen otros servicios, estos servicios juegan el rol de miembro de composición. En el caso que los servicios B y C a su vez ejercieran de controlador de otro servicios, entonces se podrían clasificar como sub-controladores.

Recordando los tipos de servicio explicados anteriormente, un servicio de tarea o un servicio de entidad podrían tener el rol de controlador, en cambio el rol de miembro de composición o de sub-controlador es más apto para servicios de entidad, debido a que los servicios de tarea suelen estar siempre a la cabeza jerárquica de la composición, con lo cual difícilmente adoptarán estos roles.

Una vez vistos todos los principios, podemos expandir el concepto de interoperabilidad como bien hemos introducido al principio de este capítulo.

7.9 *INTEROPERABILIDAD*

La interoperabilidad, como ya la hemos definido, es la habilidad de interacción entre partes de un sistema que están construidos en diferentes tecnologías. Por lo tanto, la interoperabilidad es un concepto que se aplica en el intercambio de datos. En el ámbito del software, si se requiere que dos plataformas tecnológicas diferentes intercambien datos, deberá haber un proceso de conversión para poder establecer la comunicación y que ambos extremos se entiendan. A este proceso se le llama integración. Con los principios anteriormente explicados de la orientación a

servicios, se busca una interoperabilidad intrínseca de los servicios que reduzca la necesidad de integración [8]. Estos principios pues, crean una plataforma de comunicación neutral independiente de las tecnologías usadas, como se muestra en la Figura 40. En este caso, gracias a la interoperabilidad se facilita la comunicación entre servicios implementados en dos tecnologías: .NET y J2EE.

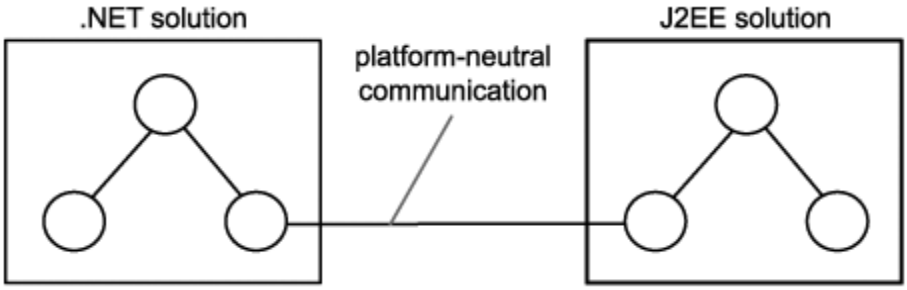

Fig. 40. La orientación a servicios facilita la interoperabilidad entre diferentes plataformas. [7]

Se debe decir que el concepto de interoperabilidad no es exclusivo del mundo del software. Como ejemplo de ello encontramos la pirámide CIM (*Computer Integrated Manufacturing*) *de* un proceso productivo (Figura 41).

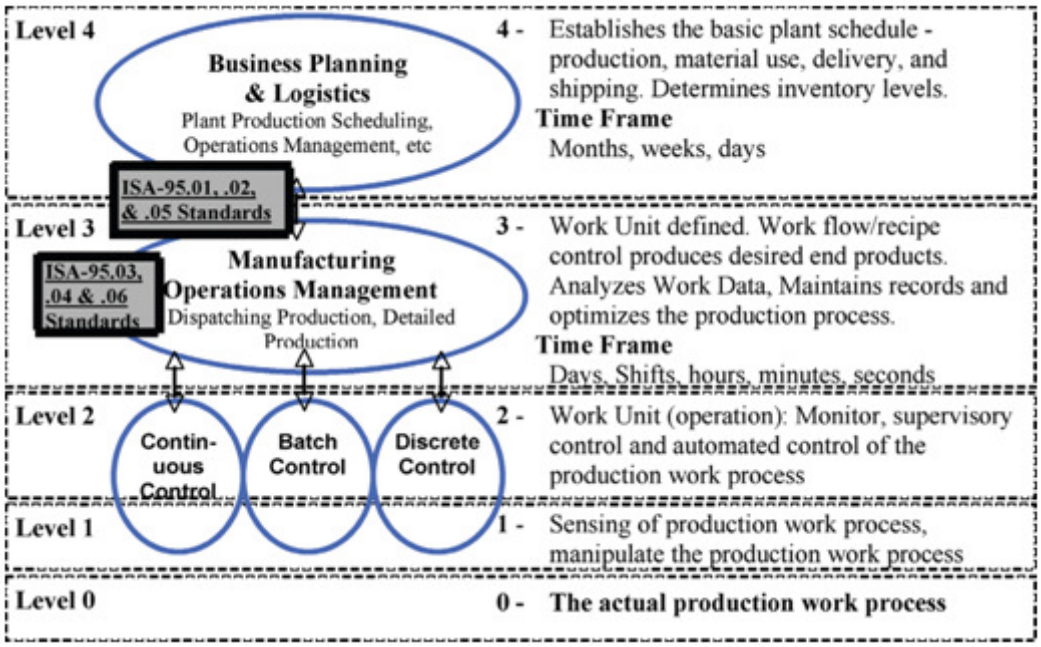

Fig. 41. Pirámide CIM.

Se observa como en entornos diferentes representados por los niveles, surge la necesidad de integración entre ellos para que el sistema completo sea interoperable y así poder comunicarse, a partir de ciertos estándares. La información viaja entre el nivel más alto donde

se elaboran los planes de producción a partir de programas informáticos especializados, al nivel más bajo que es el proceso productivo con su maquinaria respectiva.

Una vez introducido el concepto de interoperabilidad, podemos detallar de qué manera el diseño de servicios a partir de los principios de *service-orientation* establece una interoperabilidad por naturaleza.

7.9.1 Service-orientation e interoperabilidad

La creación de servicios a partir de los principios de la orientación a servicios, genera una nueva capa de comunicación en el entorno empresarial. Esta capa, como se refleja en la Figura 42, sirve de nexo de unión entre la capa de procesos de negocio y la capa de aplicaciones.

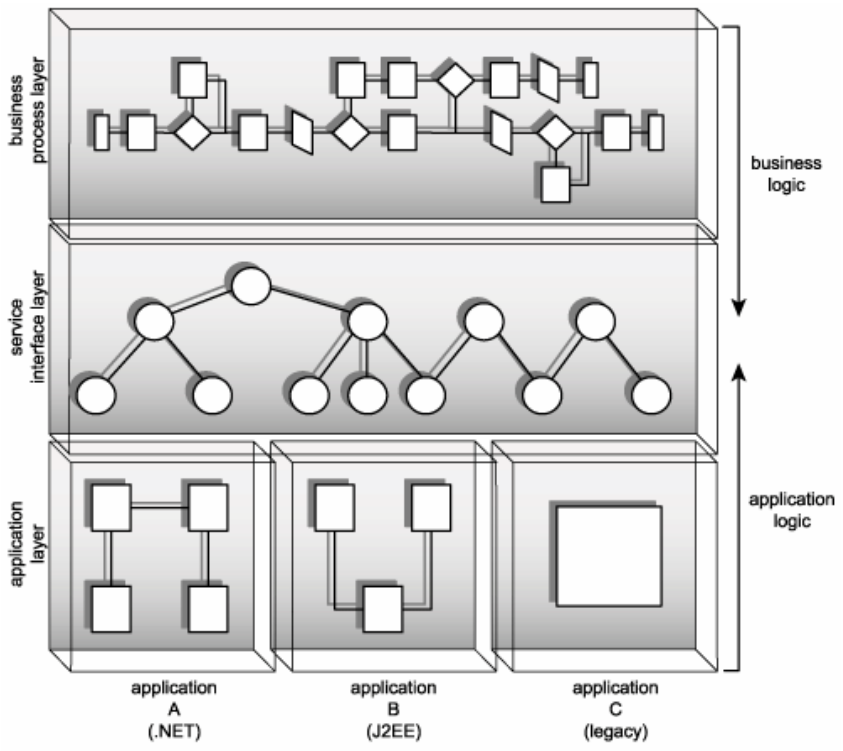

Fig. 42. La capa de servicios es un nexo de unión entre los procesos de negocio y las aplicaciones. [7]

Los servicios por una parte representan procesos de negocio empresariales y por la otra están construidos a partir de aplicaciones automatizadas que implementan la lógica necesaria de los procesos de negocio. Se puede apreciar que se diferencia entre *business logic* o lógica de negocio y *application logic* o lógica de aplicación. La diferencia entre ambos conceptos es pequeña y recae en el hecho de si la lógica es automatizada o no. Mientras que la lógica de negocio sería a nivel descriptivo y funcional, por ejemplo explicada en la normativa de la organización, la lógica de aplicación es la generada con la programación. En este estudio hemos usado y usaremos (si no se menciona explícitamente) indistintamente el

término lógica de negocio para los dos conceptos debido a que ya se sobreentiende a qué tipo nos referimos (automatizado o no) en función del contexto en que lo utilizamos.

La existencia de esta capa de servicios, facilita la interoperabilidad, como se refleja en la Figura 43. A continuación se enumeran los principios ya estudiados y de qué manera contribuyen a la ella [8] .

Contrato de servicio estandarizado

Como hemos visto, conviene estandarizar los datos en el contrato. De esta manera se sientan las bases para la compatibilidad entre contratos en un entorno interoperable, facilitando así la creación y reconocimiento de modelos de datos por cualquier sistema. Tomando el ejemplo explicado en el principio de contrato, un tipo de dato que se refiera al número de factura será más interoperable si se identifica como NumeroFactura en lugar de ObtenerFacturaPeticionTipo.

Bajo acoplamiento

El bajo acoplamiento promueve la interoperabilidad al crear servicios menos dependientes entre ellos, ya que esto facilitará también que otros consumidores de otros entornos compartan datos con los servicios. Esto es consecuencia directa de la interoperabilidad aportada por el contrato, ya que como se ha visto, el bajo acoplamiento entre consumidor y contrato de servicio se deriva de un buen diseño de contrato.

Abstracción

También como consecuencia directa del diseño del contrato, aplicando la abstracción se limita la interacción de los consumidores con el servicio, por lo tanto, si se focaliza la interacción a determinadas capacidades y modelos de datos, se puede asegurar en mayor grado la interoperabilidad. Obviamente, en caso contrario, más información aportada por el servicio, más capacidades y más modelos de datos, implicaría menos control sobre la información y por lo tanto más posibilidades de encontrar conflictos entre consumidores y el servicio.

Reusabilidad

La reusabilidad incrementa la interoperabilidad como consecuencia directa de un mayor consumo de servicios que ya se presuponen interoperables a partir de su contrato estandarizado.

Autonomía

El control que ejerce un servicio sobre su entorno de ejecución genera servicios más seguros y con un comportamiento más predecible, esta estabilidad promueve la interacción con diferentes tipos de consumidores.

Sin estado

En la línea del principio de autonomía, servicios sin estado con alto índice de disponibilidad y escalabilidad fomentarán la interoperabilidad debido al grado de seguridad aportado por este principio.

Descubrimiento

Con el descubrimiento de servicios se maximiza el potencial de interoperabilidad del servicio debido a que hay más consumidores potenciales que lo pueden encontrar.

Composición

En consonancia con la reusabilidad ya que la composición es un modo de reutilización, estos deben ser altamente interoperables ya que su posible pertinencia a diferentes composiciones, incrementa su habilidad de inter-operar con otros servicios.

En el punto en que nos encontramos, ya conocemos con detalle los servicios, los principios de la orientación a servicios y la característica de interoperabilidad que potencian. Éstos son pilares fundamentales en la arquitectura orientada a servicios, pero aun nos falta otro elemento a considerar para entender plenamente el concepto SOA.
El elemento del que hablamos son los servicios web o *web services*. Como se dijo anteriormente, los servicios web son un tipo de implementación de servicios, en concreto, un conjunto de tecnologías que se han convertido en el estándar *de facto* en la creación de servicios. El hecho de que se hayan convertido en estándar es debido a que no existe actualmente una tecnología que reproduzca con más fidelidad los principios de la orientación a servicios (principios que son agnósticos como se ha visto) ni el concepto de interoperabilidad [7]. Esto no implica que la arquitectura SOA sea obligatoriamente con servicios web, lo que sí implica es que actualmente se promueve o recomienda la construcción de SOA con servicios web [7] y se deja la puerta abierta a que si un día en el futuro aparece una tecnología superior a los servicios web, entonces se dejarán de utilizar y la nueva tecnología se convertirá en el estándar.

Otro punto a tener en cuenta, es que tanto los servicios web como el concepto SOA existen por sí mismos. Se puede operar con servicios web sin que esto implique que se ha construido una arquitectura SOA como veremos más adelante. En definitiva, la relación entre *web services* y SOA es sinérgica (ambas tecnologías se potencian mutuamente), no de obligatoriedad.

Una vez aclarado este punto, indagaremos en profundidad sobre los servicios web. Una vez estudiados, podremos entrar finalmente en el concepto SOA.

8. SERVICIOS WEB (*WEB SERVICES*)

La plataforma de los w*eb services* se define a través de un conjunto de estándares de la industria que han sido aceptados por la comunidad. Para entender el concepto de estándar primero hay que introducir el concepto de especificación. Una especificación es un documento que propone un estándar. Esta especificación para convertirse en un estándar debe enviarse a una organización reconocida de estándares. Esta organización analiza y aprueba dicha especificación y luego la divulga como un estándar de la industria.

Para explicar los servicios web, describiremos brevemente de qué estándares consta, analizaremos brevemente la historia sobre ellos y qué organizaciones han tomado parte y por último, explicaremos en detalle sus estándares.

8.1 *INTRODUCCIÓN A LOS SERVICIOS WEB*

Primeramente hay que indicar que los servicios web constan de dos generaciones diferenciadas, cada una de ellas consta de una colección diferente de estándares. La primera generación de servicios web está formada por los siguientes estándares:

- XML (*Extensible Markup Language*): Es el formato estándar para los datos que se vayan a intercambiar. Son ficheros de texto con una estructura determinada y se usará como forma de estructurar los datos en los estándares SOAP y WSDL.
- SOAP (*Simple Object Access Protocol*): Protocolo sobre los que se establece el intercambio, es decir, la forma en que se construirán los mensajes para la comunicación entre servicios.
- WSDL (*Web Services Description Language*): Es el documento de la interfaz pública para los servicios Web. Es una descripción de los requisitos funcionales necesarios para establecer una comunicación con los servicios web, por lo tanto, con WSDL damos soporte al principio de contrato estandarizado de la orientación a servicios. Para la definición y estandarización de sus datos, WSDL utiliza a su vez el estándar XML Schema.
- UDDI (*Universal Description, Discovery and Integration*): Registro para publicar la información de los servicios web. Permite comprobar qué servicios web están disponibles. UDDI es por lo tanto la implementación de un registro de servicio, concepto ya estudiado en el principio de descubrimiento de la orientación a servicios.
- WS-I Basic Profile: Son un conjunto de recomendaciones sobre el uso de los protocolos mencionados anteriormente elaboradas por el organismo WS-I.

En la Figura 43 se representa el uso de estos estándares. A la izquierda, se observa un servidor que ha implementado un WS, es decir, el proveedor del servicio. El proveedor registrará en UDDI el servicio publicando el descriptor WSDL. A la derecha, tenemos un cliente

que quiere consumir este servicio. Para ello, podrá buscarlo en UDDI o simplemente obtener directamente el WSDL si se le facilita. Una vez se tiene el WSDL, el consumidor podrá enviar un mensaje SOAP hacia el proveedor a través de Internet (HTTP). El proveedor del servicio recibirá la petición, realizará las operaciones pertinentes y elaborará un mensaje de respuesta que enviará al consumidor otra vez a través de HTTP.

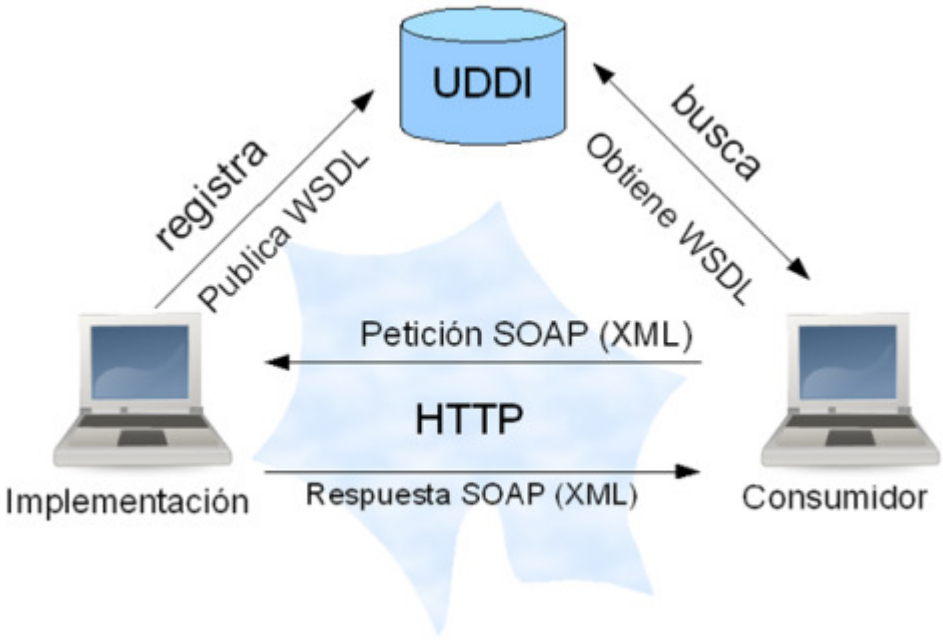

Fig. 43. Uso de estándares en la primera generación de servicios web. [15]

La segunda generación de servicios web, también conocida como WS-*, surgió para ampliar y resolver problemas existentes en la primera generación, sobretodo en áreas como la calidad de servicio, transacciones y mensajería segura, entre otras. Por ejemplo, existe el estándar WS-Policy para dar soporte a las políticas sobre servicios web, concepto explicado en los documentos técnicos del principio de contrato estandarizado.

Ahora que tenemos la idea principal de lo que son los servicios web, pasaremos a explicar brevemente qué organizaciones y cómo han contribuido a la creación de sus estándares, a partir de la Tabla 2.

	W3C	OASIS	WS-I
Established	1994	1993 as the SGML Open, 1998 as OASIS	2002
Approximate membership	400	600	200
Overall goal (as it relates to SOA)	To further the evolution of the Web, by providing fundamental standards that improve online business and information sharing.	To promote online trade and commerce via specialized Web services standards.	To foster standardized interoperability using Web services standards.
Prominent deliverables (related to SOA)	XML, XML Schema, XQuery, XML Encryption, XML Signature, XPath, XSLT, WSDL, SOAP, WS-CDL, WS-Addressing, Web Services Architecture	UDDI, ebXML, SAML, XACML, WS-BPEL, WS-Security	Basic Profile, Basic Security Profile

Tabla 2. Comparativa entre organizaciones de estándares. [7]

El paso de arquitecturas distribuidas propietarias vistas anteriormente con su propia tecnología a arquitecturas interoperables, implica el uso de estándares neutrales en referencia a su propietario o como se dice en inglés, *vendor-neutral*. En la promoción de estos estándares se han visto involucradas tanto las mismas empresas vendedoras como Microsoft, IBM o Sun Microsystems, como organizaciones de estándares. Entre estas organizaciones de estándares las más importantes son W3C, OASIS y WS-I [7]. Veamos a continuación cada una de ellas.

***World Wide Web Consortium* (W3C)**

Esta organización fue fundada por Tim Berners-Lee en 1994 y se ha centrado en la promoción de la World Wide Web como un medio global de intercambio de información. Su primer proyecto fue la creación del lenguaje HTML y con la aparición de los negocios electrónicos, W3C respondió a las nuevas necesidades surgidas con la fundación de estándares como XML o XML Schema. En W3C se han desarrollado estándares importantes para los servicios web, entre ellos los que constituyen su base, como SOAP y WSDL. También han producido una de las pocas documentaciones de referencia sobre servicios web en plataformas neutrales. W3C es una organización rigurosa con sus desarrollos, los proyectos son sujetos a numerosas revisiones y esto provoca que los estándares puedan llegar a tardar dos o tres años en ser completados.

***Organization for the Advancement of Structured Information Standards* (OASIS)**

Originalmente fue fundada en 1993 como SGML Open y en 1998 se refundó a OASIS para evidenciar el cambio de SGML (*Standard Generalized Markup Language*) a XML, punto que desarrollaremos al explicar XML. Sus desarrollos más importantes son el servicio de registros UDDI de la primera generación de servicios web y la especificación WS-Security sobre

seguridad, en la segunda generación. También han intentado promover un estándar para el intercambio de datos en el comercio electrónico B2B (*Business to Business*) con la especificación ebXML.

***Web Services Interoperability Organization* (WS-I)**

Esta organización fue fundada en 2002 y su objetivo principal no es crear estándares, sino supervisar que los estándares cumplen el objetivo de interoperabilidad. WS-I publicó el WS-I Basic Profile, un documento con recomendaciones que indica cómo deben usarse los estándares correctamente para que colectivamente se maximice la interoperabilidad. Los estándares mencionados en este documento son WSDL, SOAP, UDDI, XML y XML Schema.

Estas organizaciones, aunque sean independientes, cuentan con bastantes miembros compuestos por las mismas empresas vendedoras, evidenciando así la influencia comercial en la creación de estándares. A parte de las mencionadas Microsoft, IBM y Sun Microsystems, existen otras empresas que han participado en la creación de estándares [7], como por ejemplo BEA Systems, Oracle, Tibco, Hewlett-Packard, Canon, Commerce One, Fujitsu, Software AG, Nortel, Verisign y WebMethods.

En el próximo punto, vamos a entrar en detalle en los estándares de los servicios web.

8.2 *SERVICIOS WEB: ESTÁNDARES*

Para nuestro estudio es importante estudiar con detenimiento los principales estándares de los servicios web por dos motivos. El primer motivo es su estrecha relación con SOA y por lo tanto atañe a la parte teórica de nuestro estudio. El segundo motivo y más importante es que debemos tener unos conocimientos básicos de estos estándares ya que en la segunda parte del proyecto nos serán de utilidad para realizar un ejemplo del uso de servicios dentro de un ESB. Los estándares que veremos a continuación son los siguientes: XML, SOA, WSDL y XML Schema. Nos centramos en estos estándares porque constituyen la base de la implementación de los servicios web, creados como hemos visto, por la organización W3C. XML es el lenguaje con el que se construyen los demás estándares, SOAP es la implementación de los mensajes en los servicios web y WSDL servirá para construir contratos estandarizados con la ayuda de XML Schema.

La descripción completa de los siguientes estándares se puede encontrar en la web de W3C [12].

8.2.1 XML (*Extensible Markup Language*)

El XML es un lenguaje simple de marcas (*tags*) desarrollado por W3C como ya hemos visto, cuya función principal es la de describir datos. Juega un papel fundamental en el intercambio de información pudiendo definir gran variedad de datos. Es un lenguaje que deriva del estándar

SGML (Figura 44). Este estándar se creó con el objetivo de que los documentos electrónicos fuesen independientes de los formatos generados por los procesadores de texto y los sistemas operativos [13]. Posteriormente apareció el estándar ODA (*Open Document Architecture*), que siguiendo con la filosofía de crear documentos para estructurar la información, incluía la posibilidad de representar imágenes a partir de vectores. Esto supuso una limitación debido a que las aplicaciones debían de tener un software especial para interpretar estos datos gráficos. De todas formas, ODA sirvió de influencia para los estándares posteriores como HTML y XML. Tanto HTML como XML son lenguajes de marcas, la diferencia entre el lenguaje HTML y XML es que HTML está orientado al entorno web, en qué y cómo se debe visualizar la información en un navegador, en cambio, el lenguaje XML nos dice cómo se estructura la información en un ámbito general.

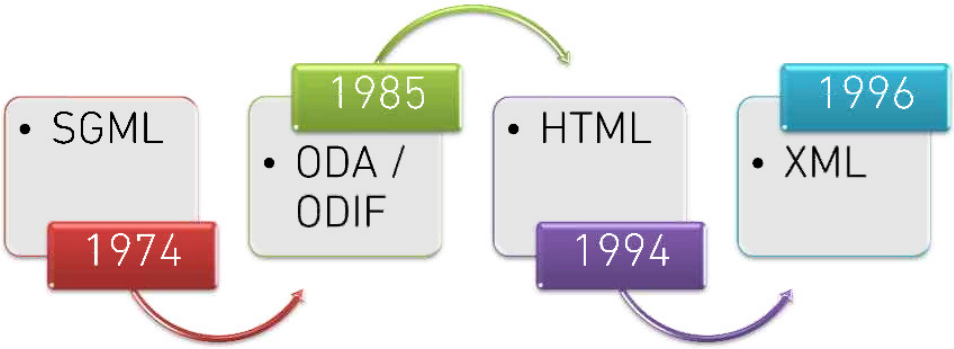

Fig. 44. Evolución histórica de los lenguajes de marcas o etiquetas.

El lenguaje XML se popularizó en la década de los 90 con la llegada de los negocios electrónicos o *eBusiness* [7], ya que los desarrolladores podían enriquecer la información enviada a través de Internet añadiéndole significado y contexto. En el contexto SOA, XML se convierte en un estándar de vital importancia debido a que establece el formato y la estructura de los mensajes enviados entre los servicios, es por ello que vale la pena conocer más sobre este estándar, como haremos en las secciones que vienen a continuación.

Diseño de un documento XML

Como se ha explicado, con el lenguaje XML podemos representar modelos de información de forma estructurada mediante el uso de etiquetas. Un ejemplo de ello es la Figura 45.

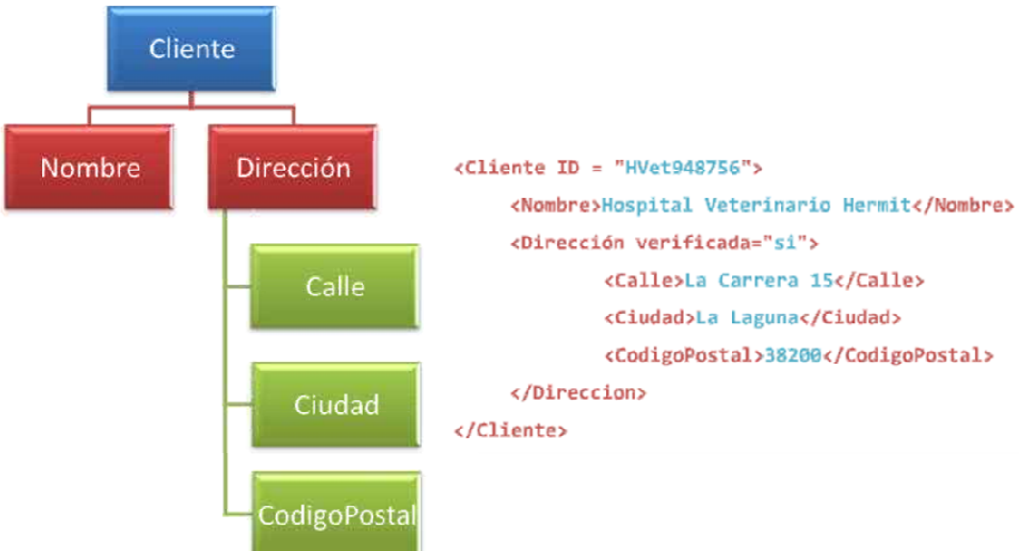

Fig. 45. Documento XML con su representación gráfica.

Se puede apreciar como a partir de una estructura jerárquica que representa a un cliente con su nombre y dirección, podemos trasladar dicha estructura a un documento de texto. Para detallar como se crea dicha estructura, vamos a relatar cómo se construye paso a paso.

El elemento a la cabeza de la estructura, en este caso Cliente, contiene todos los subelementos (Nombre y Dirección). Por lo tanto, deberá abrir la primera etiqueta y cerrar la última etiqueta. Una etiqueta se construye con el nombre de la etiqueta entre los corchetes angulares “<” y “>”. La etiqueta de cierre, para indicarla como tal, precederá al nombre de la etiqueta con el símbolo “/”. El resultado de la representación del elemento Cliente será el siguiente:

```
<cliente>
</cliente>
```

El segundo paso será representar sus subelementos (Nombre y Dirección). Para indicar esta relación de pertenencia, incluiremos los subelementos entre las etiquetas de su entidad padre (Cliente) de la siguiente forma:

```
<cliente>
        <nombre></nombre>
        <Direccion></Direccion>
</cliente>
```

A su vez, la Dirección también tiene subelementos (Calle, Ciudad y CodigoPostal). Haremos lo mismo y anidaremos estos elementos dentro de la etiqueta Dirección:

```
<cliente>
<nombre></nombre>
<Direccion>
        <Calle></Calle>
```

```
        <Ciudad></Ciudad>
        <CodigoPostal></CodigoPostal>
</Direccion>
</cliente>
```

Así de esta forma, hemos conseguido representar en formato xml la estructura jerárquica. Después de esto, podemos añadir contenido para asignar un valor a cada etiqueta, para realizar esto hay que escribir el valor entre las etiquetas de apertura y cierre del elemento. Muestra de ello es por ejemplo el elemento Calle:

```
<Calle>La carrera 15</Calle>
```

Una cosa a tener en cuenta es que los elementos pueden tener atributos que añadan información sobre ellos. El atributo se escribe dentro de la etiqueta de apertura del el elemento seguido de un símbolo de igualdad "=" y entre comillas se indica el valor del atributo. Un ejemplo de ello es el elemento Dirección:

```
<Direccion verificada="si">
```

En el ejemplo descrito en la Figura 45 no se aprecia una sentencia que se suele utilizar en el principio del documento y no tiene carácter obligaotorio. Esta sentencia es el prólogo, como ejemplo de ello podríamos tener:

```
<?xml version="1.0" encoding="UTF-8"?>
```

El prólogo indica que el documento es del tipo XML mediante la etiqueta:

```
<? xml …?>
```

Entre los puntos suspensivos se pueden incluir uno o más comentarios o instrucciones de procesamiento. En el prólogo de ejemplo se muestran como atributos las instrucciones de procesamiento indicando la versión de XML y el tipo de *encoding* del documento, es decir, la codificación de los caracteres del documento, en este caso, UTF-8. Indicar la codificación de los caracteres es útil debido a que los mensajes XML pueden enviarse entre entornos con condiciones diferentes y éstos pueden trabajar con codificaciones de caracteres distintas. Esto puede causar problemas con caracteres especiales o acentos, si se indica la codificación, el procesador del mensaje XML en la parte del consumidor sabrá en qué codificación debe tratarlo.

Ya hemos descrito la construcción de mensajes XML, pero existen unas normas que se deben cumplir para que la estructura de un documento XML se interprete como tal. Los documentos que cumplen dichas normas se catalogan de documentos bien formados. Veamos a continuación dichas directrices.

Documentos XML bien formados

Los documentos denominados como bien formados o construidos (del inglés *well formed*) son aquellos que cumplen con todas las definiciones básicas de formato y pueden, por lo tanto,

analizarse correctamente por cualquier analizador sintáctico (*parser*) que cumpla con las normas. Estas normas son las siguiente :

- Los documentos deben seguir una estructura estrictamente jerárquica con lo que respecta a las etiquetas que delimitan sus elementos. Una etiqueta debe estar correctamente incluida en otra, es decir, las etiquetas deben estar correctamente anidadas. Los elementos con contenido deben estar correctamente cerrados.
- Los documentos XML sólo permiten un elemento raíz del que todos los demás sean parte, es decir, solo pueden tener un elemento inicial. Nuestro ejemplo de la Figura 45 cumpliría con esta norma, pues solamente existe un elemento raíz Cliente.
- Los valores atributos en XML siempre deben estar encerrados entre comillas simples o dobles.
- El XML es sensible a mayúsculas y minúsculas. Existen también un conjunto de caracteres llamados espacios en blanco (espacios, tabuladores, retornos de carro, saltos de línea) que los procesadores XML tratan de forma diferente en el marcado XML.
- Es necesario asignar nombres a las estructuras, tipos de elementos y entidades.
- Las construcciones como etiquetas, referencias de entidad y declaraciones se denominan marcas; son partes del documento que el procesador XML espera entender. El resto del documento entre marcas son los datos “entendibles” por las personas.

Comentarios

Por último, indicar que es posible incluir comentarios en un fichero XML. Un comentario es un texto aclarativo que no se tiene en cuenta cuando se analiza el fichero. Con estos comentarios haremos más entendible la información contenida en un fichero XML. Los comentarios se deben escribir entre los símbolos “<! --“ y “ -->” como el ejemplo siguiente:

```
<!-- Esto es un comentario: a continuación tenemos una etiqueta -->
<etiqueta>Contenido de la etiqueta</etiqueta>
```

Una vez visto el estándar XML, estamos en posición de explicar tanto los estándares SOAP como WSDL, debido a que utilizan el lenguaje XML para ser implementados. Antes de de todo, introduciremos el estándar XML Namespaces, debido a que es de uso frecuente dentro de los mensajes SOAP y de los descriptores de fichero WSDL.

8.2.2 XML Namespaces (xmlns)

Los espacios de nombres o namespaces son una recomendación de W3C que proveen una forma para evitar conflictos de nombres entre los elementos de un documento XML [15]. Esta herramienta se utiliza mediante un atributo con nombre xmlns con la siguiente sintaxis:

xmlns:<prefijo>=<URI>

Dónde prefijo y URI serán:

- Prefijo: Puede ser el nombre que queramos. Cuando se define un espacio de nombres para un elemento, todos sus elementos hijo con el mismo prefijo son asociados al mismo namespace.
- URI (*Uniform Resource Identifier)*: Es una cadena de caracteres que identifica un recurso de Internet. Éste tendrá el formato de una URL (*Uniform Resource Locator*) convencional que identifica un dominio de Internet. Este URI se usará para dar al espacio de nombres un nombre único, aunque a veces se usa como un puntero a una página web que contiene información sobre el *namespace.*

Para ejemplificar el uso de XML Namespaces, usaremos el siguiente ejemplo:

```
<table>
  <tr>
    <td>Apples</td>
    <td>Bananas</td>
  </tr>
</table>

<table>
  <name>African Coffee Table</name>
  <width>80</width>
  <length>120</length>
</table>
```

En este fragmento XML, el elemento <table> entra en conflicto debido a que se usa el mismo nombre de elemento pero ambos tienen un contenido y significado diferente. El primer fragmento corresponde a la sintaxis de html mientras el siguiente fragmento corresponde a una sintaxis personalizada por el desarrollador.

Para evitar este conflicto entre elementos, resolveremos el conflicto usando los namespaces, definiremo2s dos espacios de nombre diferentes marcados en verde. Por un lado usaremos la etiqueta “h” para el primer fragmento table, y la etiqueta “f” para el segundo fragmento. Así nuestro anterior documento XML quedaría de la siguiente manera:

```
<root xmlns:h="http://www.w3.org/TR/html4/"
xmlns:f="http://www.w3schools.com/furniture">

<h:table>
  <h:tr>
    <h:td>Apples</h:td>
    <h:td>Bananas</h:td>
  </h:tr>
</h:table>

<f:table>
   <f:name>African Coffee Table</f:name>
```

```
    <f:width>80</f:width>
  <f:length>120</f:length>
</f:table>
```

</root>

Ahora ya podemos identificar mediante sus respectivas etiquetas los dos tipos de table dentro del mismo documento. Una vez introducido el concepto de espacios de nombres, ya podemos entrar en detalle en el estándar SOAP.

8.2.3 SOAP (*Simple Object Access Protocol*)

Como hemos visto con la orientación a servicios, todas las comunicaciones entre servicios son basadas en mensajes, esto implica la creación de un formato de mensaje estandarizado que sea usado por todos los servicios. Esto se consigue mediante SOAP. SOAP es un protocolo que define un estándar para intercambiar información estructurada utilizando el lenguaje XML, tanto en los mensajes de peticiones como en las respuestas .La comunicación es por lo tanto, una faceta fundamental en SOA. Esta importancia asociada a los mensajes implica que éstos deban ser altamente flexibles y proporcionar una alta extensibilidad, de lo contrario, si hubieran cambios en la implementación de mensajes la integridad de todo el sistema quedaría en entredicho. Un ejemplo de esta extensibilidad es la posibilidad de incorporar funcionalidades adicionales como por ejemplo los estándares de la segunda generación de servicios web. Las características de flexibilidad y extensibilidad dotan a los mensajes de robustez e independencia, de aquí que describan, como hemos visto cuando hemos introducido el concepto de mensaje, como auto-gobernados. A continuación veremos cómo se obtienen estas características gracias a la estructura del mensaje SOAP.

Estructura de un mensaje SOAP

Un mensaje SOAP consta de la siguiente estructura tal y como se observa en la Figura 46:

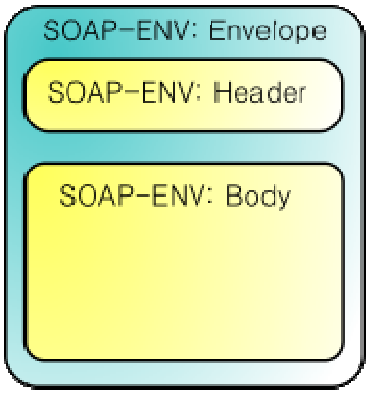

Figura 46. Estructura de un mensaje SOAP. [16]

Un mensaje SOAP se compone primeramente de una sección que englobará todo el mensaje y actuará como un sobre, esta sección será el Envelope y se representará con una etiqueta <soap:Envelope>. Dentro de el sobre, tendremos la cabecera (Header) y el cuerpo del mensaje (Body).

La cabecera del mensaje consta de una serie de bloques o *header blocks* y estará representada con la etiqueta <soap:Header>. La cabecera contendrá meta-información del mensaje, y aunque es opcional, normalmente se incluye debido a que es gracias a ella que se puede extender el mensaje SOAP a partir de sus bloques. Estos bloques son paquetes de meta-información suplementarios que aportan información relativa sobre el procesamiento, el enrutamiento, propiedades e instrucciones del mensaje a los servicios que interaccionarán con estos mensajes. Esto es importante porque gracias a los bloques de la cabecera conseguimos la autonomía de los servicios, ya que no deberán guardar ni mantener lógica asociada a la información específica del mensaje [7], el mensaje será suficientemente inteligente llevando con él la lógica necesaria.

La siguiente sección dentro del sobre del mensaje SOAP es el cuerpo, representado con la etiqueta <soap:Body>, que es donde habrá la información propiamente del mensaje. Normalmente será información en formato XML. En la Figura 47 se muestra un ejemplo sencillo de mensaje SOAP donde se puede apreciar una cabecera y un cuerpo.

```
<?xml version="1.0"?>
<soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
  <soap:Header>
    <!--Optional header information goes here. -->
    <To>Scott</To>
    <From>Suzanne</From>
  </soap:Header>

  <soap:Body>
    <!--Message goes here. -->
    Please pick up some milk on your way home from work.
  </soap:Body>
</soap:Envelope>
```

Fig. 47. Ejemplo de un mensaje SOAP. [17]

Lo primero que podemos observar en la Figura 47 es el uso de Namespaces dentro de la etiqueta soap:Envelope, para así evitar problemas con los nombres dentro del documento usando el prefijo **soap**. A continuación se muestra la cabecera, que contiene dos elementos que describen a quien compuso el mensaje, y posible receptor del mismo. El último elemento es el cuerpo, que contiene un mensaje simple.

Hemos dicho que SOAP se usa tanto para mensajes de llamada, como de respuesta, veamos un ejemplo de cada tipo.

Mensaje de llamada

Como ejemplo de mensaje de una petición SOAP, tomaremos una posible mensaje realizado a un servicio al que le pasaremos un identificador de producto y el servicio nos devolverá información relativa a dicho producto. En el ejemplo mostrado a continuación observamos que la petición se realiza llamando a la operación o método getProductDetails. Este nombre debe coincidir con la operación ofrecida por el servicio, en caso contrario no se reconocerá. Como parámetro de dicha operación, observamos el parámetro productId que contendrá el valor del identificador de producto sobre el cual queremos consultar la información. A continuación se detalla el posible código de dicha llamada:

```
<soap:Envelope
xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
   <soap:Body>
     <getProductDetails xmlns="http://warehouse.example.com/ws">
       <productId>827635</productId>
     </getProductDetails>
   </soap:Body>
</soap:Envelope>
```

Una vez el servicio procese esta petición y se ejecute la lógica de negocio que hace posible la recuperación de la información del producto, por ejemplo, mediante un acceso a una base de datos, el servicio elaborará otro mensaje de respuesta, como veremos a continuación.

Mensaje de respuesta

El mensaje de respuesta puede contener los resultados de la llamada al método o una estructura de fallo bien definida en el caso de que haya habido algún error en la ejecución de la lógica del serivicio. Por convenio, la estructura de respuesta tiene el mismo nombre que el método original al que se le añade *result*, por lo tanto el contenido de la respuesta empezará con la etiqueta <getProductDetailsResponse>. Si se encuentra un error durante la ejecución, el mensaje de respuesta devolverá una estructura de datos con información relativa a este error, por ejemplo con un código de error y un mensaje descriptivo. De esta forma, el consumidor del servicio puede tratar de forma adecuada el caso de error. A continuación mostramos una posible respuesta a la petición anterior:

```
<soap:Envelope
xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
   <soap:Body>
     <getProductDetailsResponse
xmlns="http://warehouse.example.com/ws">
       <getProductDetailsResult>
         <productName>Toptimate 3-Piece Set</productName>
         <productId>827635</productId>
```

```
        <description>3-Piece luggage set.  Black
Polyester.</description>
        <price>96.50</price>
        <inStock>true</inStock>
      </getProductDetailsResult>
    </getProductDetailsResponse>
  </soap:Body>
 </soap:Envelope>
```

Una vez visto como se comunican los servicios entre ellos, pasaremos a ver la creación de la descripción del sevicio del fichero mediante el documento WSDL. Previamente, hablaremos del estándar XML Schema. Este estándar se usa dentro de los documentos WSDL y para entender su importancia debemos recordar el principio de contrato estandarizado. Cuando se ha explicado dicho principio se ha hecho hincapié en la necesidad de tener los datos estandarizados. Pues bien, para conseguir la definición de datos y por consiguiente poder estandarizarlos, se provee dentro del marco de trabajo de los servicios web el estándar XML Schema, cuyo funcionamiento veremos en el siguiente apartado.

8.2.4 XML Schema

El XML Schema es un lenguaje de esquema para escribir la estructura y las restricciones del contenido de un documento XML [18]. Como se ha visto anteriormente, fue desarrollado por W3C.

Veamos el siguiente ejemplo:

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
     <xsd:element name="Libro">
         <xsd:complexType>
            <xsd:sequence>
                 <xsd:element name="Título" type="xsd:string"/>
                  <xsd:element name="Autores" type="xsd:string"
                maxOccurs="10"/>
                 <xsd:element name="Editorial" type="xsd:string"/>
            </xsd:sequence>
            <xsd:attribute name="precio" type="xsd:double"/>
        </xsd:complexType>
   </xsd:element>
</xsd:schema>
```

En este documento se crea el elemento **xsd:schema** indicando así que vamos a crear una estructura de datos. Con **<xsd:element** name="Libro"**>** estamos indicando que queremos una elemento XML de nombre libro:

```
<Libro></Libro>
```

Con **<xsd:complexType>** indicamos que este elemento Libro tendrá subelementos: Titulo, Autores, Editorial, Precio. Título, Autores y Editorial están dentro de un elemento **xsd:sequence** y esto implica que deberán estar en este mismo orden. Por lo tanto, una estructura que cumple esta definición es la siguiente:

```
<Libro>
    <Titulo></Titulo>
    <Autores></Autores>
    <Editorial></Editorial>
    <precio></precio>
</Libro>
```

Observamos que en los **xsd:element** aparecen dos atributos a parte del atributo name. Estos atributos son type y maxOccurs. El atributo type indica el tipo de dato de un elemento. A continuación se detallan los tipos de datos más habituales y una breve descripción sobre ellos [19].

- String: Representa una cadena de caracteres. Por ejemplo: “Esto es una cadena de caracteres”.
- Boolean: Representa los valores lógicos cierto y falso o true y false en inglés.
- Decimal: Números decimales que se pueden obtener de la división de un entero y potencias de 10^n de la forma: $i/10^n$ donde $n \geq 0$.
- Float: Decimales con una precisión de 32 bits.
- Double: Decimales con una precisión de 64 bits.
- Integer: Números enteros.
- DateTime: Representa un instante de tiempo a partir del año, mes, dia, hora, minuto y segundo.

El segundo atributo observado es maxOccurs, que indica cuantos elementos de ese tipo pueden haber. En este caso, se podrían definir como máximo diez autores. Es decir, pueden haber entre uno y diez elementos <Autores></Autores>.

Ahora que ya sabemos la forma de definir tipos de datos, podemos entrar en detalle en los ficheros de descripción de servicio WSDL.

8.2.5 WSDL (*Web Services Description Language*)

Cuando se ha realizado la introducción a los servicios se ha hablado del documento de descripción de servicio, que permitía identificar un servicio y así actuar como interfaz de comunicación con otros servicios que quisieran usarlo. Así también, gracias a este documento un servicio puede ser descubierto. Posteriormente se ha introducido el concepto de la orientación a servicios de bajo acoplamiento y cómo se conseguía gracias al contrato estandarizado y por lo tanto, a la descripción de servicio incluida en él. En el ámbito de los

servicios web, la implementación de un descripción de servicio se realiza con el documento WSDL que es una combinación de SOAP y XML Schema.

Un programa cliente que pretenda usar un servicio web puede leer el WSDL para determinar qué funciones están disponibles en el servicio, esto se realiza a partir del punto de contacto que permite la comunicación, a este punto de contacto se llama *service endpoint* y estará indicado en el WSDL. Los tipos de datos usados por las operaciones del servicio se incluyen en el archivo WSDL en forma de XML Schema. El cliente puede usar SOAP para hacer la llamada a una de las funciones listadas en el WSDL, que como hemos visto cuando se ha explicado SOAP, el nombre usado de la operación en el mensaje SOAP debe coincidir con la operación definida en el WSDL.

Una vez introducido el concepto de WSDL y su finalidad, entraremos en detalle en cómo está estructurado.

Estructura del WSDL

La estructura de un WSDL se puede diferenciar en dos categorías separadas tal y como se refleja en la Figura 48.

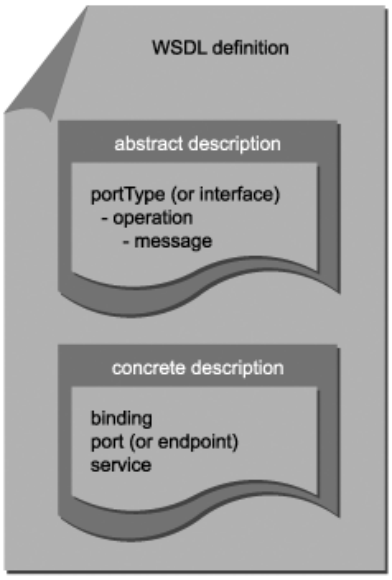

Fig. 48. Estructura de un WSDL. [7]

Estas dos categorías son la descripción abstracta y la descripción concreta. La descripción abstracta es la parte del WSDL que establece las interfaces del servicio, es decir, todas aquellas partes del servicio que se pueden definir sin tener en cuenta ningún tipo de tecnología en concreto. De esta forma se preserva la integridad del servicio y si en un futuro se cambia la tecnología estas partes no habrá que cambiarlas. Las partes que sí que deberían de cambiar son las de la descripción concreta, debido que en esta parte del WSDL es dónde se asocia la descripción abstracta a un tipo de tecnología en concreto, por ejemplo,

el protocolo de transporte usado en la comunicación. Veamos en detalle cada parte del WSDL.

Descripción abstracta

La descripción abstracta está compuesta de las siguientes secciones:

- Tipos de datos: a esta sección le corresponde la etiqueta <type>. Es donde se describen los tipos de datos usados en los mensajes. Se utiliza XML Schema para definirlos.

- Mensajes: a esta sección le corresponde la etiqueta <message>. Es la definición abstracta de los elementos del mensaje. Con los mensajes definimos los datos intercambiados en las peticiones y respuestas establecidas entre el solicitante del servicio y el servicio. En los mensajes por lo tanto, estarán definidos los parámetros y valores de retorno usados. Estos parámetros pueden definirse a partir de las estructuras creadas en la sección de tipos de datos mediante el atributo element, o se puede indicar directamente un tipo si el dato es de tipo básico, como por ejemplo un String, mediante el atributo type.

- PortType o Interface: a esta sección le corresponde la etiqueta <portType>. Es la definición abstracta de las operaciones permitidas, dentro de cada operación se definen qué mensajes se recibirán (mensajes input) o se enviarán (mensajes output). Estos mensajes serán los definidos previamente en la sección de mensajes. En función de si se puede o no recibir mensajes dentro de las operaciones, hay cuatro tipos de transmisiones permitidas, tal y como se muestra en la Figura 49.

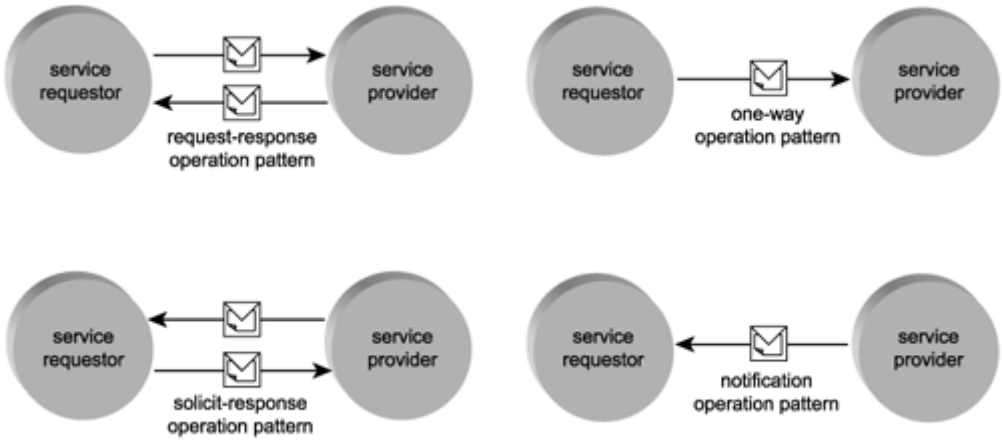

Fig. 49. Tipos de transmisiones permitidas en el WSDL. [7]

- One-way: El servicio solamente recibe mensajes, es decir solamente tiene mensajes del tipo input.
- Request-Response: El servicio recibe un mensaje y devuelve otro mensaje de respuesta, por lo tanto tendrá un mensaje input y otro output.

- Solicit-Response: El proceso inverso a Request-Response, el servicio envía un mensaje y recibe una respuesta, con lo que tendrá un mensaje output y otro input.
- Notification: El proceso inverso a One-way, el servicio solamente envía mensajes, por consiguiente, tendrá solamente mensajes del tipo output.

Una vez vistas las secciones de la descripción abstracta, detallaremos las secciones de la descripción concreta.

Descripción concreta

La descripción concreta está compuesta de las siguientes secciones:

- Binding: a esta sección le corresponde la etiqueta <binding>. En esta sección se especifica un protocolo de comunicación concreto (SOAP/HTTP, HTTP GET/POST, etc…) y un formato de datos para una operación en particular. Es decir, asociamos una implementación tecnológica concreta, a las operaciones del servicio.
- Servicios: a esta sección le corresponde la etiqueta <service>. En ella se definen los puertos que especifican los endpoints. Harán referencia a un Binding. Definen pues, dónde se estará preparado para recibir/enviar mensajes.

Para ejemplificar todos estos conceptos detallados tanto en la descripción abstracta como concreta, analizaremos un WSDL de ejemplo y comentaremos cada sección:

```
<definitions name="StockQuote"
       targetNamespace="http://example.com/stockquote.wsdl"
       xmlns:tns="http://example.com/stockquote.wsdl"
       xmlns:xsd1="http://example.com/stockquote.xsd"
       xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
       xmlns="http://schemas.xmlsoap.org/wsdl/">

 <types>
    <schema targetNamespace="http://example.com/stockquote.xsd"
            xmlns="http://www.w3.org/2000/10/XMLSchema">
        <element name="TradePriceRequest">
           <complexType>
               <all>
                   <element name="tickerSymbol" type="string"/>
               </all>
           </complexType>
        </element>
        <element name="TradePrice">
           <complexType>
               <all>
                   <element name="price" type="float"/>
               </all>
```

```
            </complexType>
          </element>
        </schema>
    </types>

    <message name="GetLastTradePriceInput">
        <part name="body" element="xsd1:TradePriceRequest"/>
    </message>

    <message name="GetLastTradePriceOutput">
        <part name="body" element="xsd1:TradePrice"/>
  </message>

  <portType name="StockQuotePortType">
      <operation name="GetLastTradePrice">
        <input message="tns:GetLastTradePriceInput"/>
        <output message="tns:GetLastTradePriceOutput"/>
      </operation>
  </portType>

  <binding name="StockQuoteSoapBinding"
type="tns:StockQuotePortType">
      <soap:binding style="document"
transport="http://schemas.xmlsoap.org/soap/http"/>
      <operation name="GetLastTradePrice">
        <soap:operation
soapAction="http://example.com/GetLastTradePrice"/>
        <input>
            <soap:body use="literal"/>
        </input>
        <output>
            <soap:body use="literal"/>
        </output>
      </operation>
  </binding>

  <service name="StockQuoteService">
      <documentation>My first service</documentation>
      <port name="StockQuotePort"
binding="tns:StockQuoteSoapBinding">
        <soap:address location="http://example.com/stockquote"/>
      </port>
  </service>

 </definitions>
```

A continuación se da una descripción en detalle de cada apartado:

Types

Se definen dos tipos de datos, TradePriceRequest que tendrá un elemento tickerSympol del tipo string (cadena de caracteres) que solo podrá aparecer una vez. Esto se define con la etiqueta <all>.

El segundo elemento definido es TradePrice que tendrá un subelemento Price del tipo float (decimal).

Message

Se definen dos mensajes el primero GetLasTradePriceInput hace referencia al elemento TradePriceRequest definido en la sección types, y GetLastTradePriceOutput que hará referencia al elemento TradePrice también definido en la sección types.

PortType

Se define la operación GetLastTradePrice dónde el mensaje de la petición será el mensaje GetLastTradePriceInput y la respuesta será el mensaje GetLastTradePriceOutput.

Por lo tanto, se está definiendo una operación dónde se enviará un identificador de valor bursátil (tickerSymbol) y se recibirá la respuesta de su valor actual en el mercado (Price).

Binding

En esta sección se define el binding StockQuoteSoapBinding donde se indica que la operación GetLastTradePrice se hará sobre el protocolo HTTP/SOAP a partir del atributo transport=http://schemas.xmlsoap.org/soap/http

Hasta este punto todo se había definido de forma abstracta, las operaciones y los mensajes. Es con el binding dónde se le da unas características concretas de protocolo de transporte.

Service

Se define el endpoint donde llegaran las peticiones para el binding. El endpoint se define en:

<soap:address location="http://example.com/stockquote"**/>**

Esto significa que la petición SOAP realizada por el consumidor del servicio se deberá enviar a la URL definida en el atributo location.

Una vez vistos los puntos más importantes de los servicios web realizaremos una comparativa entre los principios de *service-orientation* y los servicios web, para ver como éstos proveen una implementación de dichos principios, y por lo tanto, para construir soluciones SOA.

8.3 *SERVICE-ORIENTATION Y SERVICIOS WEB*

Los servicios web ofrecen de forma inherente ciertos aspectos reflejados en los principios de la orientación a servicios [7]. Estos principios son los siguientes: contrato estandarizado, bajo acoplamiento, abstracción y composición. El uso de WSDL cumple a la vez varios de ellos, primeramente el de contrato estandarizado, ya que gracias a la descripción de servicio construimos la interfaz de comunicación entre servicios. Hemos visto también como dentro del WSDL se pueden definir tipos de datos que promueven la estandarización de la información mediante el uso de XML Schema. El uso de WSDL implica a su vez un desacoplamiento de la lógica del servicio y sus capacidades, ya que los consumidores no hace falta que tengan detalle de la implementación de las operaciones ofrecidas por el servicio, simplemente deben conocer las operaciones disponibles. Ya hemos visto que la forma de conseguir un bajo acoplamiento es gracias a la abstracción aportada por el servicio y su contrato, por lo tanto de forma natural, los servicios web promoverán este principio también, actuando como una caja negra. Gracias a ello, el propietario del servicio podrá asumir decisiones sobre la implementación sin tener en cuenta su efecto en los consumidores. Otro principio que los servicios web aportan de forma natural es el de la composición, el grado en que esta se consiga dependerá tanto del diseño de los servicios como de la reusabilidad intrínseca que el servicio ofrezca a partir de su lógica de negocio. Dicho de otra manera, un servicio web es naturalmente apto para una composición, pero en qué medida lo sea dependerá de la funcionalidad para la que se haya creado.

Los otros principios de los que consta la orientación a servicios, reusabilidad, autonomía, sin estado y descubrimiento, no se puede decir que sean conseguidos de forma natural por los servicios web. Debe haber un diseño a conciencia sobre ellos para que se puedan conseguir dichos principios. La reusabilidad está relacionada con la composición, como ya hemos dicho, a una reusabilidad mayor los servicios son potencialmente más aptos para las composiciones, pero la reusabilidad deberá ser conseguida mediante la funcionalidad y su lógica subyacente. Por lo tanto, no depende directamente del uso de por sí de los servicios. El principio de autonomía, tanto aplicado a autonomía en tiempo de ejecución como en tiempo de diseño, no dependerá del servicio de forma intrínseca, sino que será gracias a un diseño estudiado que conseguiremos los grados de seguridad y predictibilidad del comportamiento del servicio de forma adecuada. En cuanto el estado del servicio, hemos visto que este viene determinado en gran medida por la información contenida en los mensajes, a más información de estado a procesar el servicio estará más tiempo activo y por lo tanto menos disponible. Por lo tanto, en este aspecto será más importante el diseño de los mensajes SOAP que el propio servicio. Para finalizar, el descubrimiento de los servicios es una cuestión externa al servicio, el contrato del

servicio se usa en su descubrimiento, pero sin la infraestructura adecuada como UDDI o un inventario de servicios, no será posible el descubrimiento.

Una vez vista la relación con los principios de la orientación a servicios, concluiremos nuestro estudio dedicado a los servicios web indicando sus principales ventajas e inconvenientes.

8.4 *VENTAJAS E INCONVENIENTES DE LOS SERVICIOS WEB*

Los servicios web como ya se ha mencionado anteriormente, son una herramienta de referencia en la construcción de una arquitectura orientada a servicios. Esto es así porque las ventajas de su uso son numerosas y tienen pocas desventajas. A continuación veremos estas ventajas e inconvenientes.

8.4.1 Ventajas de los servicios web

El hecho de que los servicios web estén basados en estándares, aporta una serie de beneficios. Entre ellos encontramos la interoperabilidad entre aplicaciones de software independientemente de las plataformas sobre las que se instalen, como por ejemplo, aplicaciones basadas en Java o .NET. El uso de estos estándares a través de servicios web hace qua ambas plataformas se comuniquen en el mismo idioma. Una consecuencia directa de la interoperabilidad es que los servicios web promueven que servicios de diferentes compañías ubicadas en diferentes lugares geográficos puedan ser combinados fácilmente para proveer servicios integrados, gracias a que los servicios web operan por Internet y la información basada en XML puede viajar a través de diferentes protocolos de red, como por ejemplo los protocolos: HTTP (*Hypertext Transfer Protocol*), protocolo usado en la WWW (*World Wide Web*) , FTP (*File Transfer Protocol*), el protocolo de transferencia de ficheros, o SMTP (*Simple Mail Transfer Protocol*), el procotolo de correo electrónico. Otro beneficio aportado es que estos estándares al estar basados en texto gracias a XML, esto hace más fácil el acceso a su contenido y entender su funcionamiento, como hemos comprobado con los ejemplos mostrados de los mensajes SOAP y del descriptor de servicio WSDL. Otra ventaja a destacar ya estudiada, es que gracias al cumplimiento de principios como el contrato estandarizado, bajo acoplamiento y abstracción, conseguimos separar el diseño del servicio representado en su contrato, de su implementación, consiguiendo así que si hay decisiones que implican un cambio de tecnología en la implementación del servicio esta se puede realizar sin afectar a las dependencias generadas del servicio con sus consumidores.

8.4.2 Inconvenientes de los servicios web

Como inconveniente, podemos indicar que el rendimiento de los servicios web es bajo si se compara con otros modelos de computación distribuida ya vistos en la introducción a la arquitectura del software, tales como EJB, CORBA o DCOM. Es uno de los inconvenientes derivados de adoptar un formato basado en texto. La diferencia radica en el hecho que los

textos se deben procesar y esto requiere un tiempo de cómputo y recursos superiores a los necesitados para protocolos RPC, que como ya hemos visto, serializan en bytes los datos a transmitir. En los servicios web se antepone la interoperabilidad entre plataformas aunque esto suponga un descenso del rendimiento. Se intenta minimizar esta problemática mediante el principio de servicios sin estado o *stateless* como hemos visto con anterioridad. Para ejemplificar gráficamente esta diferencia de rendimiento podemos observar la Figura 50.

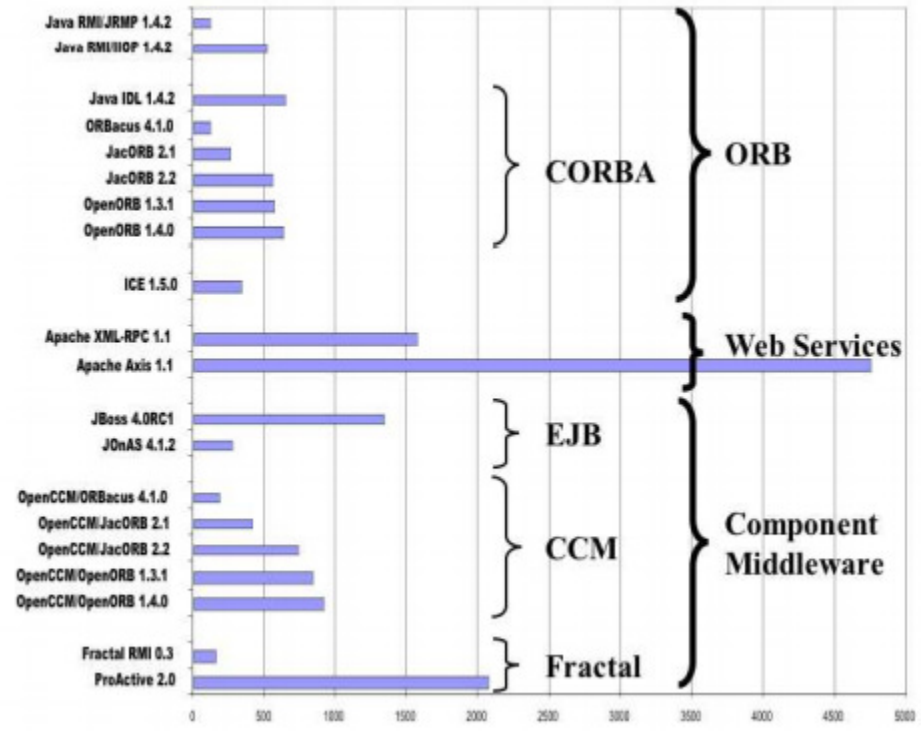

Fig. 50. Comparativa de *round-trip latency* de servicios web y componentes distribuidos. [20]

En la Figura 50 se realiza una comparativa de diferentes tecnologías de comunicación distribuida basadas en Java. Vemos la presencia de una implementación de CORBA a partir de Java, y de los EJB de Java también mencionados con anterioridad. Nos fijaremos en estas tecnologías como ejemplo de arquitecturas distribuidas ya comentadas anteriormente, y también por supuesto, en las implementaciones Java de los servicios web Java XML-RPC y Java Axis como ejemplo de servicios web. Las mediciones observadas en el gráfico son a partir de los tiempos medios en microsegundos de *round-trip latency* o *round-trip delay time* (RTT). Este tiempo es el que tarda un paquete de datos enviado desde un emisor en volver a éste mismo habiendo pasado por el receptor de destino. De esta forma se puede tener una idea de lo que tarda un componente o servicio web en recibir, procesar y enviar la información. Del gráfico se observa como efectivamente, los servicios web son los que presentan un rendimiento menor o que es lo mismo, tienen unos tiempos RTT mayores.

Después de mencionar las ventajas e inconvenientes de los servicios web, podemos mencionar una característica que depende de cómo se mire es una ventaja o un inconveniente. Al utilizar

el protocolo HTTP sobre TCP (*Transmission Control Protocol*) en el puerto 80, la comunicación entre servicios puede esquivar medidas de seguridad basadas en *firewall* (cortafuegos) cuyas reglas tratan de bloquear o auditar la comunicación entre programas a ambos lados de la barrera. Para entender mejor este concepto simplemente nos tenemos que fijar que cuando accedemos por un navegador de Internet (Internet Explorer, Google Chrome, Firefox, etc…) e introducimos una URL, por ejemplo, http://www.google.es lo que se está realizando es pues una petición HTTP y por defecto éstas son por el puerto 80. Sería equivalente a escribir http://www.google.es:80 . Por lo tanto, en organizaciones con fuertes medidas de seguridad bloquearan con el uso del *firewall* muchos puertos exceptuando el puerto 80, porque entonces sus usuarios no podrían navegar. Es por este motivo que puede resultar una ventaja o un inconveniente el uso de servicios web por HTTP. Por un lado es ventajoso porque no se depende del departamento de sistemas quien debe autorizar/denegar el uso de determinados puertos. Por otro lado, se puede interpretar como un uso inseguro ya que se permite un tipo de comunicación de negocio no controlada.

Llegado a este punto de nuestro estudio, ya tenemos un conocimiento firme de la evolución de la arquitectura de software que ha dado lugar a los servicios, conocemos el funcionamiento de los servicios, los principios que rigen su diseño y la forma actual de implementarlos mediante los servicios web. Cada uno de estos puntos mencionados nos servirán de base para poder comprender con mayor facilidad el concepto de arquitectura orientada a servicios, que explicaremos a continuación en el siguiente capítulo.

9. ARQUITECTURA ORIENTADA A SERVICIOS (SOA)

Lo primero que se debe decir referente a SOA es que no existe ningún organismo o institución oficial que la defina. Por lo que una buena forma de introducir su concepto será exponer diferentes definiciones ya que cada una de ellas puede aportar puntos de vista diferentes. La definición propuesta por el organismo OASIS [21] es:

"SOA es un paradigma para organizar y utilizar capacidades distribuidas que pueden estar bajo el control de diferentes dominios. Provee una manera uniforme de ofrecer, descubrir, interactuar con ellos y sus capacidades de uso para producir el efecto deseado consistente en precondiciones y expectativas medibles."

Esta definición de OASIS cataloga SOA primeramente como un paradigma, es decir, es un marco conceptual y que no está constituido por una tecnología en concreto. También indica que es un tipo de arquitectura distribuida, pero no entra en detalle sobre si ésta se consigue mediante servicios, simplemente utiliza el concepto abstracto de capacidades. Sí que hace hincapié en el hecho de que la comunicación debe ser interoperable (entre diferentes dominios), de que la forma de comunicación debe ser uniforme, o dicho de otra manera, estandarizada, y que las capacidades pueden de ser descubiertas.

Otra definición posible según Mary E. Shacklett, una conocida analista tecnológica y presidenta de Transworld Data, es la siguiente [22]:

"Técnica que ha surgido del desarrollo de software orientado a objetos que fue evolucionando hacia la creación de servicios web que encapsulan piezas de software de negocio usadas en la web para coordinar procesos de negocio de empresa."

En esta definición hay cuatro conceptos importantes a tener en cuenta. El primero es que remarca el hecho de que la orientación a servicios es una evolución de sus arquitecturas predecesoras, ciertamente tiene puntos en común con la POO, como la encapsulación, ya que los servicios también encapsulan su lógica mediante la abstracción. Esta definición junto con la de OASIS, reflejan lo visto en el capítulo de la introducción histórica, como se ha evolucionado de POO a arquitecturas distribuidas y finalmente a servicios.

El segundo punto importante es que en este caso, y a diferencia de la definición de OASIS, se menciona explícitamente la relación de SOA con servicios y aun más, con los servicios web. De alguna manera está indicando lo que se ha comentado a lo largo del estudio, y es que la evolución ha llevado a la actualidad al uso específico de servicios web dentro de SOA, siempre teniendo en cuenta que SOA es un paradigma y por lo tanto es agnóstica en cuanto a tecnología.

El tercer punto importante de la definición es el concepto de coordinación de procesos de negocio. La coordinación de procesos de negocio es similar al concepto de composición de servicios, pero implica un nivel superior de abstracción por encima de los servicios. Los servicios se coordinan para formar procesos de negocio más complejos, es lo que se llama de servicios, se entrará en detalle sobre este concepto más adelante.

El último punto interesante es la mención del entorno empresarial en la definición, en este caso, la definición es más concreta que la vista anteriormente, donde la descripción es más abstracta en cuanto su ámbito de uso.

Otra definición posible es la ofrecida por IBM [23]:

"Una arquitectura orientada a servicios es un estilo de arquitectura para crear una infraestructura de IT empresarial que explota los principios de *service-orientation* para conseguir una estrecha relación entre los sistemas de información y el negocio".

En esta definición de IBM, con una clara orientación al negocio empresarial, se hace una referencia explícita a los principios de la orientación a servicios ya estudiados y se remarca que gracias a SOA se obtiene el alineamiento entre negocio y tecnología. Este concepto de alineamiento es un tema recurrente en el ámbito de las TI. Se considera que tanto el negocio como la tecnología deben estar alineados, es decir, la tecnología no existe como fin sino que es un medio para conseguir los objetivos marcados por el negocio, por lo tanto, debe haber una estrecha relación entre ellos y la forma de conseguirlo es mediante SOA, modelando los servicios como funciones de negocio y coordinándolos de forma adecuada para definir los procesos de negocio. Otro concepto derivado de este alineamiento es el concepto de BPM (*Business Process Management*), BPM son un conjunto de actividades y conocimientos cuyo objetivo es mejorar la eficiencia de la organización a través de la gestión sistemática de los procesos de negocio, mediante su modelado, automatizado, integración y optimización de forma continua. BPM por lo tanto, involucra tanto a procesos de negocio, como tecnologías de información y a personas. Antes hemos introducido el concepto de orquestación de servicios, podríamos decir que esta orquestación sería una parte a realizar dentro de la gestión global que representa BPM.

Por último, mencionaremos la definición de SOA realizada por Thomas Erl, una eminencia en el ámbito SOA y autor de varios libros, dos de los cuales se encuentran en la bibliografía de nuestro estudio. Según Erl pues [7]:

"SOA es una forma de arquitectura tecnológica que se adhiere a los principios de *service-orientation*. Cuando SOA se realiza a través de la plataforma tecnológica de *Web services,* SOA establece el potencial para dar soporte y promover estos principios a lo largo de los procesos de negocio y los dominios automatizados de una empresa".

Thomas Erl también da una orientación empresarial a SOA, y dice que ésta se basa en los principios de la orientación a servicios. Se debe destacar la frase “Cuando SOA se realiza a través de servicios web” ya que con esto indica que SOA no solamente se realiza a través de servicios web, pero sí que a través de ellos se obtiene el mayor potencial. Esta definición es un resumen de una definición más extensa dónde también se menciona SOA como una evolución de las arquitecturas anteriores y como ésta estandariza la comunicación a lo largo de la empresa.

Gracias a estas definiciones hemos visto dos posibles vertientes sobre SOA. La primera es una definición abstracta de ella, sin mencionar un entorno empresarial ni la utilización de servicios ni ningún tipo de tecnología en concreto de servicios. La otra posibilidad es una SOA, más concisa, basada en servicios que se rigen por unos principios concretos, y además, estos servicios están implementados en una tecnología en especial, los servicios web.

A partir de esta reflexión podemos elaborar el siguiente esquema (Figura 51) que nos dará una visión global del concepto SOA y posteriormente iremos desenvolviendo cada entidad reflejada.

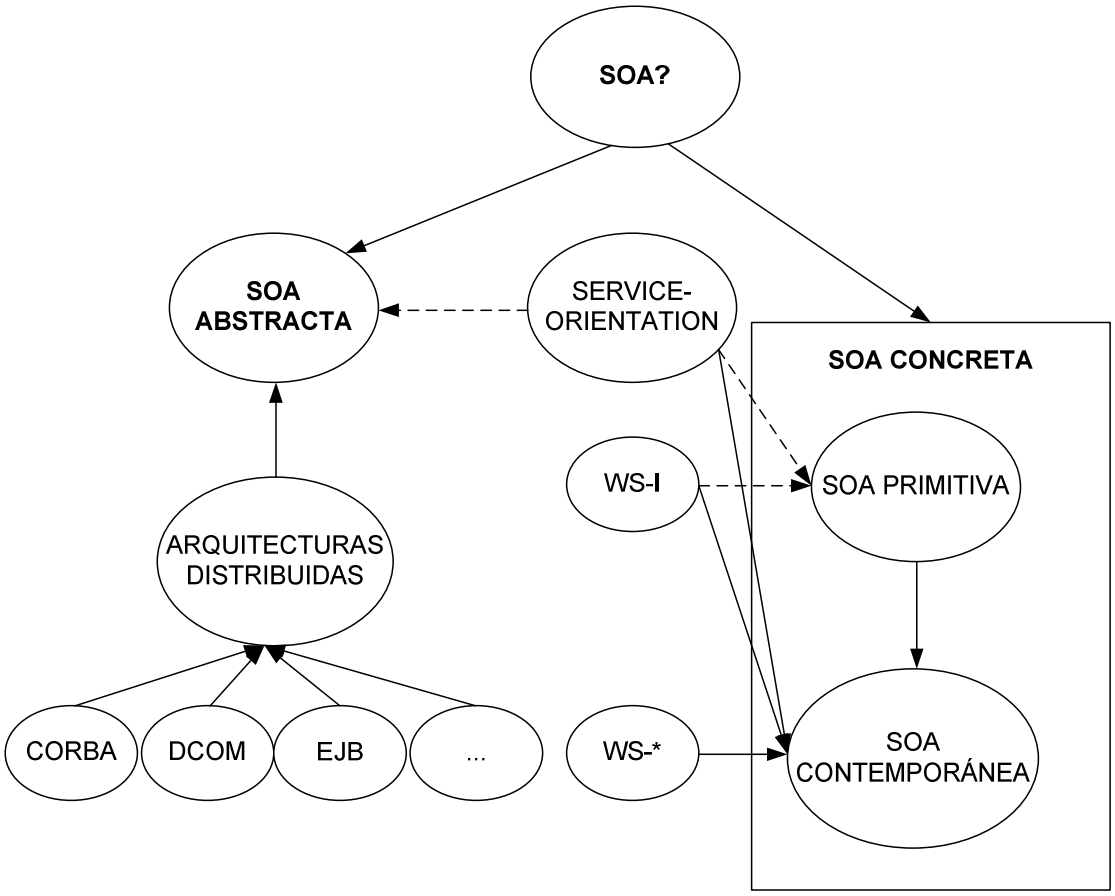

Fig. 51. Diferentes interpretaciones de SOA.

La primera división que nos encontramos reflejada en el gráfico es entre SOA Abstracta y SOA Concreta. Con estos nombres se pretende diferenciar las dos concepciones sobre SOA

extraídas de las definiciones respectivas. Empezaremos estudiando el concepto de SOA abstracta.

9.1 *SOA ABSTRACTA*

Con SOA abstracta (diferenciando abstracta del concepto de abstracción visto en los principios de la orientación a servicios) se pretende dar cobertura a aquellas interpretaciones de SOA que parten de definiciones como la de OASIS. En estas definiciones, se da un concepto abstracto de servicio, un servicio puede ser cualquier entidad que recibe información, la procesa, y devuelve una respuesta, sin especificar tampoco como viaja dicha información. Desde esta perspectiva, cualquier elemento software mínimamente definido puede actuar como servicio, y por lo tanto, los componentes de las arquitecturas distribuidas pueden actuar como tal.

Otra cosa a tener en cuenta es si dichas implementaciones de SOA cumplen los principios de la orientación a servicio y hasta qué grado lo hacen, por eso se ha representado la flecha de relación de SOA abstracta con s*ervice-orientation* con una línea discontinua. Por ejemplo, los componentes distribuidos tienen una interfaz que puede actuar a modo de contrato, pero serán unas implementaciones muy básicas que se limitarán a la definición de las operaciones del servicio, en cambio, como hemos estudiado, el concepto de contrato estandarizado es más amplio. Otro ejemplo es el principio de descubrimiento, los EJB por ejemplo pueden ser descubiertos remotamente mediante el uso de JNDI (*Java Naming and Directory Interface*) pero su ámbito de operación es intra-aplicación como hemos visto, es decir, se limita al abarcado por la arquitectura distribuida a la que pertenece. Otro principio que no se consigue completamente es el de acoplamiento, por ejemplo, cuando se busca un componente en JNDI se necesita tener referencias de dicho componente en tiempo de compilación a diferencia de los servicios implementados como servicios web. Hay pero, algunos principios que podrían ser soportados por una SOA abstracta, como el principio de abstracción, ya que se puede esconder la lógica del componente de forma adecuada y publicar en su interfaz solamente la información requerida por otros componentes. Otro principio que también se puede conseguir es el de reusabilidad de los componentes, siempre y cuando se tenga en cuenta que su ámbito de actuación será reducido al de la propia arquitectura distribuida, sin posibilidad de interaccionar con otros sistemas construidos con otras tecnologías.

En definitiva, una SOA abstracta sólo podrá cumplir parcialmente los principios de orientación a servicios. Esto tiene un inconveniente crucial que la diferencia de la SOA concreta, y éste es el concepto de interoperabilidad potenciado por cada uno de los ocho principios de la orientación a servicios. Aún cuando consigamos una SOA abstracta que cumpla de forma aceptable los principios de la orientación a servicios, nunca podrá ser interoperable con otras plataformas tecnológicas. A no ser, como también hemos visto, que dicho componente se encapsule en un servicio que haga de *wrapper* y que permita exponer al exterior la funcionalidad del componente, pero en este caso, ya se consideraría que la comunicación es mediante servicios y mensajes, y el componente queda relegado a un segundo plano como lógica de servicio.

Una vez vista la SOA abstracta, pasaremos a explicar el concepto de SOA concreta y sus diferenciaciones entre SOA primitiva y SOA contemporánea, términos acuñados por Thomas Erl [7].

9.2 *SOA CONCRETA*

A diferencia de la SOA abstracta, consideramos como SOA concreta aquel concepto de SOA definido a partir de unas características concretas. Estas características que hemos identificado a partir de las definiciones de SOA son el cumplimiento de los principios de *service-orientation* y el uso de los servicios web, que como hemos visto, son la implementación de servicios que cumplen con más rigor dichos principios y prueba de ello es que son la plataforma más usada en la actualidad.

Thomas Erl identifica dos tipos de SOA, dentro de la categoría que nosotros hemos definido como SOA concreta. Estos dos tipos son SOA Primitiva y SOA Contemporánea. SOA Primitiva son las primeras versiones que aparecen de SOA, y posteriormente ésta da lugar a la SOA Contemporánea, que presenta extensiones sobre SOA Primitiva. A continuación describiremos ambos tipos de SOA.

9.3 *SOA PRIMITIVA*

Entendemos por SOA Primitiva como las primeras versiones de esta arquitectura basadas en la mayoría de los principios de orientación a servicios y de la primera generación de servicios web, como se observa en la Figura 51. Tanto la relaciones representadas de los principios de orientación a servicios como WS-I están indicadas con línea discontinua, esto es debido a que no se considera como parte de ella la parte relacionada con el descubrimiento de los servicios, esto es, el uso de UDDI y por lo tanto el principio de descubrimiento. En estas primeras construcciones SOA los servicios se implementan como servicios web, los mensajes se construyen con SOAP, y la descripción de servicio se realiza mediante WSDL. La característica de descubrimiento se asocia con una SOA Contemporánea, como veremos a continuación.

9.4 *SOA CONTEMPORÁNEA*

En la Figura 51 se refleja la SOA Contemporánea como una evolución de SOA primitiva, que cumple los principios de la orientación a servicios y está basada en servicios web, con la primera y segunda generación. SOA Contemporánea es la extensión de su versión primitiva a partir de los nuevos desarrollos y especificaciones que han ido apareciendo sobre la plataforma de los servicios web. Gracias a estas extensiones, se puede concretar una definición propia para SOA Contemporánea:

“SOA Contemporánea representa una arquitectura abierta, ágil, extensible, federada y modular, compuesta por servicios autónomos, con soporte para la calidad del servicio (QoS), que

proveen de diferentes proveedores, interoperables, descubribles y potencialmente reutilizables, implementados como *Web services*." [7]

A continuación, veremos los elementos detallados en esta definición que hacen de SOA Contemporánea el marco SOA de referencia actual y una evolución de la SOA Primitiva.

Abierta

SOA Contemporánea se define como abierta debido al uso de estándares ya vistos, gracias a ellos los servicios son auto-contenidos y promueven un bajo acoplamiento debido a que los servicios solamente deben centrarse en el descriptor de servicio para conocerse entre ellos.

Ágil

Gracias a la capa de servicios creada con el modelado de procesos de negocio a partir de servicios y a la abstracción y bajo acoplamiento que aportan estos servicios, se dota a SOA con la debida agilidad para hacer frente a los cambios que puedan venir tanto de la lógica de negocio, ya sea automatizada o no, como de la infraestructura tecnológica.

Extensible

Con el uso de descripciones de servicio bien diseñadas, es decir, con un buen nivel de abstracción y estandarización, se promueve la reutilización de los servicios extendiendo así la funcionalidad del servicio a tareas diferentes.

Federada

Con federada se entiende que gracias a la función de *wrapper* de los servicios, se unifica la tecnología de las aplicaciones legadas (ya existentes) sin tener que reemplazar toda la lógica existente ya implementada en la organización por nuevas implementaciones. En resumen, se estandarizan las aplicaciones construidas con diferentes tecnologías existentes en la empresa, a partir de los servicios y sus estándares.

Modular

El uso de los estándares provistos con WS-* aportan flexibilidad a SOA, se puede conseguir el grado de extensión requerido en SOA. Es decir, estos estándares actúan como módulos, podemos incluir los que se requieran en cada momento. Más adelante entraremos en detalle en este aspecto.

QoS

Otra diferencia que aporta una SOA Contemporánea es el concepto de calidad de servicio o QoS. Ya se ha estudiado anteriormente que en los contratos de servicios pueden haber documentos adicionales, y que el más representativo es el documento de calidad del servicio.

Gracias a QoS podemos añadir características inexistentes en la versión primitiva de SOA, como por ejemplo, la protección del contenido de los mensajes mediante encriptación, añadiendo notificaciones de entrega en los envíos de mensajes o poder hacer que las tareas de negocio llevadas a cabo por los servicios sean transaccionales.

Descubribles

En la versión primitiva de SOA, aún cuando en la primera generación de servicios web existía ya el registro UDDI, raramente se utilizaba. Esto es debido a que las primeras SOA venían de las tradicionales arquitecturas distribuidas, donde los componentes se encapsulaban en servicios y por lo tanto las conexiones eran de componente a componente. Al haber este tipo de conexiones, el uso de un WSDL basta para establecer la comunicación. Es con SOA Contemporánea, cuando se fomenta un registro de servicios a nivel de empresa, a partir del cual se puedan descubrir servicios para realizar composiciones.

En la Figura 51, se ha marcado con línea continua la relación de los principios de orientación a servicio y WS-I con SOA Contemporánea, para remarcar que incluye el principio de descubrimiento y el registro UDDI.

Vemos pues, que la SOA Contemporánea se ha convertido en una extensión de la SOA primitiva, aportando conceptos nuevos, sobre todo con el uso de los estándares provistos en la segunda generación de los servicios web de forma modular (Figura 52).

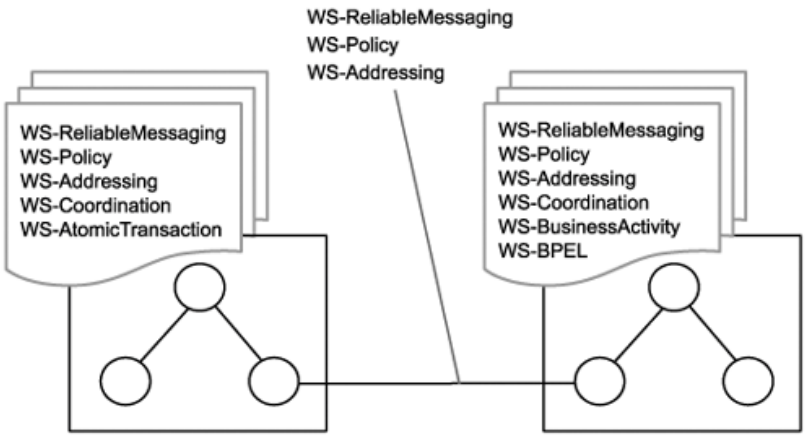

Fig. 52. Estándares aportados por WS-* en una SOA Contemporánea. [7]

Entre todos estos estándares, destacaremos por su importancia el estándar WS-Policy, que da soporte al concepto de políticas y que dotan de inteligencia a los mensajes como se ha explicado anteriormente. Otros estándares de interés son WS-BPEL y WS-CDL. El estándar WS-BPEL hace referencia a la orquestación de servicios (Figura 53).

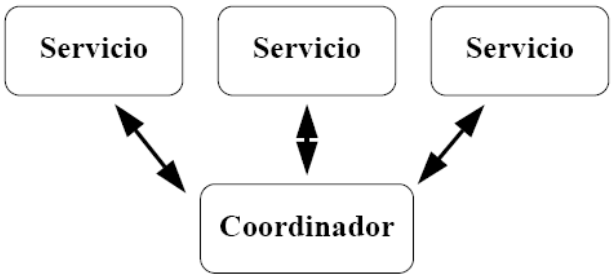

Fig. 53. Orquestación de servicios. [28]

La orquestación de servicios sirve para conectar diferentes procesos encapsulados en servicios para generar un proceso de negocio más complejo mediante lo que se denomina un *workflow* o flujo de trabajo. La lógica que coordina los procesos se separa de los servicios, con esta abstracción, se consigue un mantenimiento más sencillo debido a que ahora esta lógica no reside dentro del servicio. WS-BPEL utiliza internamente el estándar WS-Coordination, que permite gestionar meta-datos relativos a la coordinación de servicios, como el número de servicios involucrados, la duración del proceso, o el número de instancias concurrentes que pueden existir a la vez, es decir, si el mismo proceso se puede ejecutar n veces en paralelo.

El otro estándar, WS-CDL, es la implementación del concepto de coreografía. La coreografía (Figura 54) establece unas reglas de colaboración entre servicios, la diferencia entre coreografía y orquestación es que la orquestación determina un flujo de trabajo en concreto y tiene un propietario dentro de una organización en concreto. En cambio, la coreografía tiene un ámbito que puede ser inter-empresas, cuando múltiples servicios de diferentes organizaciones necesitan trabajar juntos para conseguir una finalidad conjunta [7].

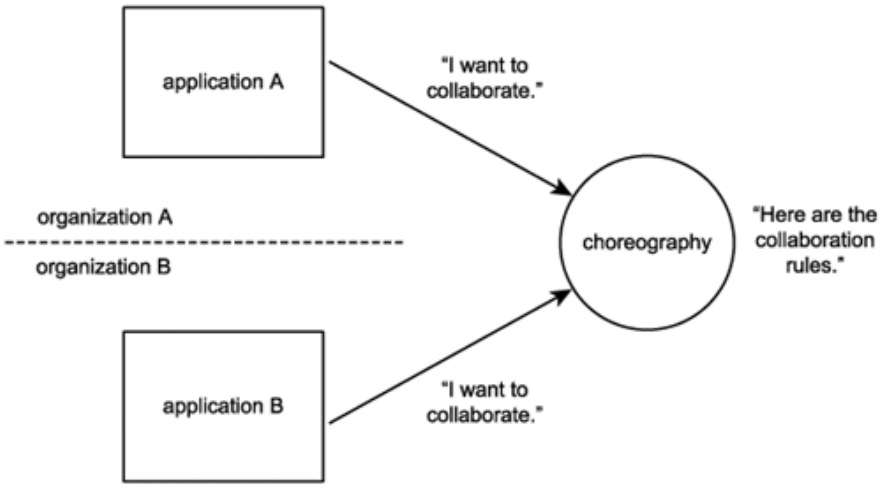

Fig. 54. Coreografía de servicios. [7]

Hasta ahora, hemos visto las dos concepciones principales sobre SOA, desde un punto de vista de una definición abstracta que da lugar a varias interpretaciones y desde el punto de vista opuesto, teniendo en cuenta que SOA está basada en unos principios y éstos se cumplen

a través de la utilización de servicios web. Desde este último punto de vista y tomando como referencia una SOA Contemporánea, podemos realizar una taxonomía de ella para observar gráficamente su composición, donde entrarán elementos ya estudiados y otros de nuevos que detallaremos. Esta taxonomía se refleja en la Figura 55.

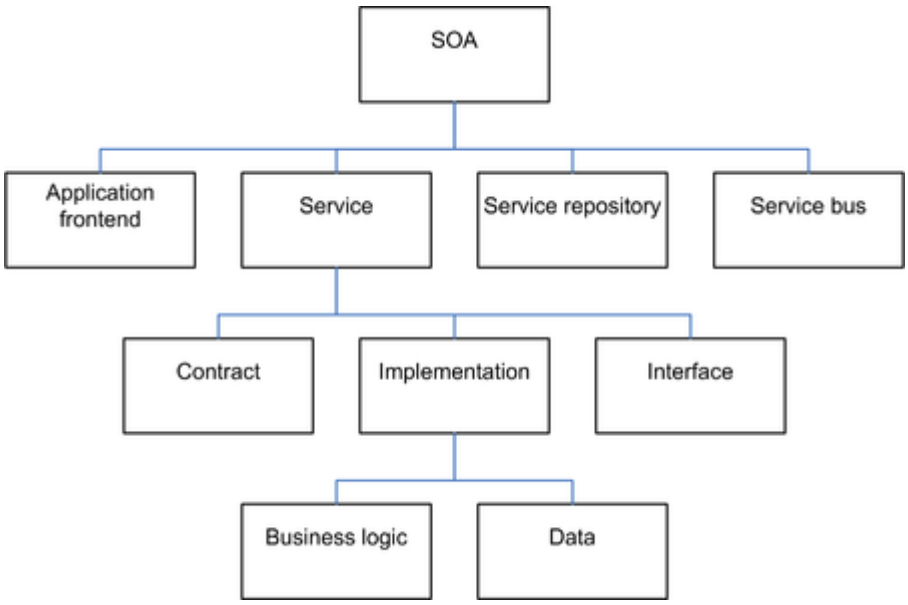

Fig. 55. Taxonomía de una SOA Contemporánea . [29]

Dentro de esta taxonomía encontramos elementos ya estudiados, como el concepto de servicio y sus elementos asociados: el concepto de contrato e interfaz, aunque interfaz podría estar asociada dentro del contrato, ya que es en la descripción de servicio donde se crea la interfaz de operaciones disponibles del servicio. Dentro del servicio también encontramos la representación de su lógica de negocio mediante el elemento *Implementation*, y éste puede estar compuesto tanto por la lógica de negocio propiamente dicha y sus fuentes de datos (*Data*) de donde se obtiene la información necesaria. Un ejemplo de datos sería una base de datos donde se guarda la información relativa a una factura, y sus líneas de factura asociadas.
El otro elemento estudiado con anterioridad es el de repositorio de servicios, donde tendremos el inventario con los servicios y un registro que nos permitirá descubrir los servicios. A parte de estos elementos, encontramos dos que aún no hemos visto. Estos son el concepto de *Application frontend y Service Bus*.

Por *Application frontend* se entiende como la parte que está al frente, es decir, la parte visual de las aplicaciones que nos permitirán interaccionar con el sistema. Siguiendo el ejemplo de las facturas, nuestro *frontend* podría ser una aplicación web simple, con un formulario que nos permita introducir en un campo de texto el número de factura a consultar y un botón para enviar

la información. Una vez pulsado el botón, se llama al servicio relacionado con las facturas, éste internamente a partir de su lógica de negocio consulta en la base de datos la información relativa a la factura, y el servicio devuelve al *frontend* la información, que se muestra por pantalla al usuario que ha realizado la consulta.

El último elemento detallado es el de *Service Bus* o *Enterprise Service Bus (*ESB). Haciendo una analogía con un coche, el ESB sería el motor de SOA. Un ESB es una herramienta software que hace de intermediario entre consumidores y proveedores de servicios. Su razón de ser es la misma que nos encontramos en el mercado de consumo. Existen fabricantes de productos, consumidores, y entre medio están los mayoristas que hacen de intermediarios, reduciendo así el número de transacciones comerciales necesarias entre unos y otros. De la misma forma pues, actúa un ESB, centralizando las comunicaciones de los servicios. ESB es una pieza fundamental en una arquitectura SOA, y será tratada por separado más adelante.

Una vez visto el concepto de SOA Contemporánea y de qué partes se compone, estudiaremos como se debe construir una arquitectura SOA dentro de un entorno empresarial.

9.5 *METODOLOGÍA DE IMPLEMENTACIÓN DE SOA*

La implementación de soluciones orientadas a servicios es un proceso complejo que requiere la participación y cooperación de partes interesadas (*stakeholders*) dentro de una empresa (Figura 56).

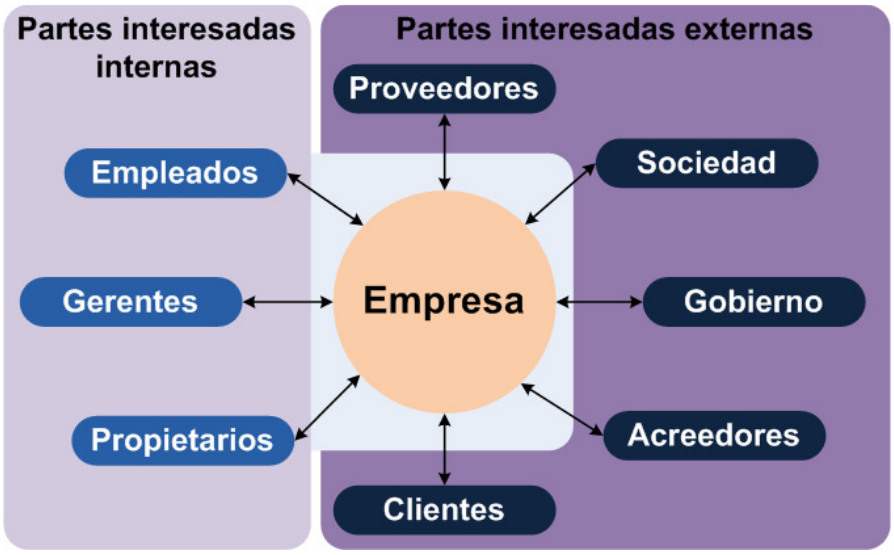

Fig. 56. *Stakeholders* de una empresa. [24]

El término *stakeholder* fue utilizado por primera vez por R. E. Freeman [25], y define como *stakeholder* como “quienes pueden afectar o son afectados por las actividades de una empresa”. Dentro de estos, como se observa en la Figura 56, podemos tener partes

interesadas de dentro de la empresa, como los empleados, gerentes o propietarios, y partes externas, como proveedores, clientes, accionistas, etc…

Debido a que SOA constituye un alineamiento entre el negocio y la tecnología, tal y como la definió IBM, debería de haber un trabajo conjunto entre partes interesadas, sobre todo las internas, en la construcción de una arquitectura orientada a servicios. Esto es debido a que los servicios representan procesos de negocio de la empresa, con lo cual, responsables de la gerencia con un fuerte conocimiento del negocio deberán relacionarse con otros empleados de la empresa cuya función esté relacionada con los sistemas de información. Si desde el departamento de sistemas se construyen servicios que no representan adecuadamente las funciones de negocio, los servicios no serán de utilidad o lo serán en menor grado. También se debería de tener en cuenta en este diseño de SOA a las partes interesadas externas, ya que es muy posible que proveedores o clientes hagan uso de los servicios publicados por la empresa, con lo cual, estos servicios deberían estar construidos para satisfacer las necesidades de sus consumidores.

Dentro de este marco colaborativo a nivel empresarial, se requiere de una metodología bien definida para llevar a la realidad una arquitectura orientada a servicios, describiendo los pasos a realizar, los roles y responsabilidades de cada grupo involucrado. La parte más importante de cualquier metodología es, por supuesto, el proceso de la creación de servicios. Existen dos tipos de estrategias de aproximación en base a las capas de lógica de negocio y lógica de aplicación.

La primera estrategia es la *top-down*. Esta estrategia simboliza un proceso de definición de arriba a abajo, es decir, de la capa de lógica de negocio hacia la capa de lógica de aplicación. Esta estrategia se considera que tiene un componente más analítico, debido a que al empezar por la lógica de negocio permite no solamente crear servicios que cumplan los principios de *service-orientation* sino que también permite una reorganización de los procesos de negocio actuales.

La segunda estrategia es la *bottom-up* o de abajo a arriba. Será el proceso inverso a la estrategia *top-down*, con lo que el diseño de los servicios se empieza a partir de la lógica de aplicación, es decir, de las aplicaciones ya existentes. Esta estrategia es más rápida que la *top-down* debido a que el tiempo dedicado al diseño de los servicios es menor, estos se generan a partir de las aplicaciones ya existentes. El inconveniente de esta estrategia es que no se conseguirá un cumplimento de los principios de *service-orientation* tan adecuado como la estrategia *top-down.* Esto es debido a que se parte de lógica existente y lo único que se hace es incluir servicios web como *wrappers*, con lo que, por ejemplo, no se puede crear un buen diseño de contrato estandarizado.

En resumen, la estrategia *top-down* es una estrategia que requiere más tiempo (y por lo tanto es más costosa) que la estrategia *bottom-up.* Pero por otro lado, es la estrategia deseada para conseguir un buen nivel de cumplimiento de una SOA Contemporánea [7].

Un ejemplo de una posible metodología *top-down* podría ser la representada en la Figura 57.

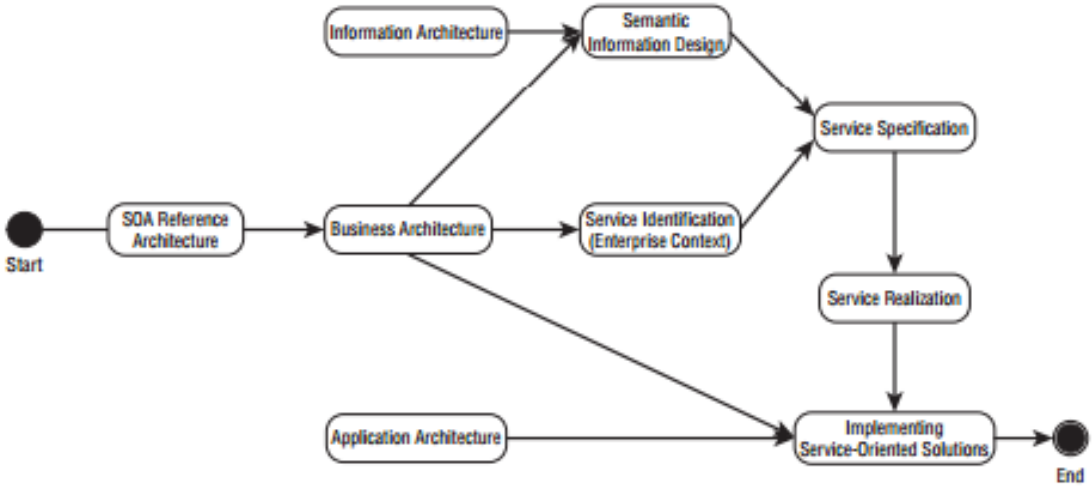

Fig. 57. Metodología *top-down* para la implementación de SOA. [26]

Las actividades principales de esta metodología son las siguientes:

- **Referencia de la arquitectura SOA.**

  Define los aspectos principales de la arquitectura SOA, como por ejemplo, la definición de servicio y los conceptos estudiados relacionados con la orientación a servicio: contrato, mensaje, inventario de servicios, etc.. También otros conceptos que forman parte del lenguaje relacionado con SOA y los servicios, como acoplamiento, granularidad, estandarización, en general, términos usados dentro de SOA y que las personas involucradas en la implementación de SOA deben conocer.

- **Definición de la arquitectura de negocio**

  Definición de los procesos de negocio, servicios e información involucrados en el conjunto de la empresa a nivel operacional y estratégico. Un ejemplo de ello podría ser un proceso de compra o de venta, detallando los pasos que se suceden en proceso y que entidades se ven involucradas, como por ejemplo, las facturas.

- **Identificación de servicios**

  Definición del conjunto de servicios en el contexto empresarial. Esto formará el inventario de servicios. A partir de los procesos de negocio identificados en el paso anterior, se analizan dichos procesos y se identifican los posibles servicios que puedan

haber en ellos, tal y como hemos estudiado, podemos determinar servicios de tarea, de entidad y de utilidad.

- **Definición del modelo de información semántica**

  Creación de un modelo de información empresarial que define las semánticas compartidas de los procesos y servicios. Esta actividad se lleva en paralelo con la identificación de los servicios. Este modelo semántico es influenciado tanto por la arquitectura de información como la de negocio. Gracias a este paso, conseguiremos elaborar un vocabulario unificado que nos será muy útil para la creación de contratos de servicio estandarizados como se ha visto.

- **Especificación de servicios**

  Creación de los contratos de servicio en tiempo de diseño. Crearemos las estructuras de datos estandarizadas necesarias a partir del modelo de información semántica elaborado en el paso anterior y también se crearán las interfaces que determinan las operaciones que permitirán los servicios. Por ejemplo, si hemos determinado un servicio Factura, una posible operación sería getFactura con parámetro de entrada el identificador de la factura.

- **Implementación de servicios**

  En esta fase se implementan los servicios, teniendo en cuenta la arquitectura de las aplicaciones y la arquitectura de negocio. Lo importante de este proceso es que no hay que tener un inventario completo de servicios identificados, se puede empezar por el nivel de servicio de más alto nivel y luego ir descendiendo en cascada para cada vez ir definiendo más servicios, actualizando nuestro inventario de servicios. Siguiendo el caso de ejemplo con facturas, una vez definido el servicio Factura y estando éste operativo, puede ser que sea un nuevo requisito más adelante la creación de un servicio que sea LiniaFactura, podremos entonces crear este nuevo servicio y modificar la implementación del servicio Factura para que utilice el nuevo servicio LiniaFactura. Con esto, la funcionalidad de Factura no se ha visto alterada de cara a sus consumidores ya que el cambio ha sido transparente, y por otro lado tenemos en nuestro inventario un nuevo servicio LiniaFactura que podrá ser reutilizado por otros servicios si hace falta.

Para finalizar nuestro estudio sobre SOA, en el siguiente punto vamos a hablar de las ventajas e inconvenientes del uso de este tipo de arquitectura.

9.6 *VENTAJAS DE LA ARQUITECTURA SOA*

Como el ámbito de actuación de SOA está entre la tecnología y el negocio, encontraremos ventajas desde las dos perspectivas empresariales.

Desde un punto de vista tecnológico, como primer beneficio encontramos la integración de aplicaciones y en consecuencia, una mayor interoperabilidad. Ya hemos introducido anteriormente estos conceptos, indicando que la integración es habilitar un marco común para que sistemas heterogéneos puedan intercambiar información, gracias a la interoperabilidad intrínseca que aporta el diseño de la orientación a servicios y el uso de estándares aportados por los servicios web, se reduce la necesidad de integración debido a que todos los sistemas usan el mismo lenguaje, evitando así desarrollos dedicados a la conversión y adaptación de datos de unas aplicaciones a otras. La herramienta software dentro de SOA que sirve de plataforma para la integración de aplicaciones es el ESB, que como hemos dicho, estudiaremos más en detalle en el siguiente capítulo. Otra ventaja relacionada con la interoperabilidad es que gracias a SOA y al uso de servicios web, se federan los sistemas legados con poco esfuerzo, o sea, solamente hace falta que el servicio web actúe como *wrapper* de estos sistemas.

Otro beneficio aportado por SOA es el de la reusabilidad de los servicios, esto repercute reduciendo el tiempo de duración de futuros desarrollos debido a que se aprovechan funcionalidades ya implementadas por los servicios existentes. Pero no solo nos beneficiamos en desarrollos futuros, el uso de servicios permite reducir también los tiempos de realización en modificaciones sobre la lógica actual, debido a que tenemos las funcionalidades centralizadas en servicios sin tener duplicidades de la misma funcionalidad en diferentes aplicaciones.

Otra ventaja aportada por SOA es que es componible, no solamente refiriéndonos a la composición de servicios, sino también al uso de los estándares WS-*. Como hemos visto, su naturaleza modular nos permiten conseguir diferentes grados de realización dentro de SOA. Por ejemplo, podemos añadir políticas al funcionamiento de los servicios con WS-Policy y dentro de ellas, posteriormente especificar políticas relativas a seguridad en la comunicación con WS-Security.

En resumen, el entorno tecnológico con bajo acoplamiento, componible, interoperable y potencialmente reusable proporcionado por SOA, dota a una organización de la agilidad necesaria para responder adecuadamente ante posibles cambios.

Para ejemplificar de forma gráfica estos beneficios, podemos usar la Figura 58. En esta figura se representa un escenario empresarial a partir de tres aplicaciones (*Service Scheduling, Order Processing, Account Management*), antes de desarrollar SOA y después de SOA. En el escenario anterior a SOA, nos encontramos con varios problemas. El primero es que las tres aplicaciones son monolíticas, es decir, contienen ellas mismas toda la lógica necesaria para su

funcionamiento, lo que repercute en la existencia de funcionalidad replicada. Tanto *Order Processing* como *Account Management* son procesos que implementan cada uno la misma funcionalidad *Order Status.* Otro problema es que esta misma funcionalidad, representa procesos de negocio diferentes, esto se observa en el diagrama de color blanco adjunto al nombre de la funcionalidad. Otro problema que podemos detectar es que cada aplicación mantiene las conexiones con los repositorios de datos, por ejemplo, si nos fijamos en la base de datos de CRM (*Customer Relationship Management*) donde encontraremos los datos relativos a los clientes, vemos que las tres aplicaciones tienen una conexión con esta base de datos. Si esta base de datos sufre algún cambio, deberemos modificar las tres aplicaciones para ajustarlas a la modificación, repercutiendo así en un mayor tiempo dedicado y por consiguiente, un coste mayor.

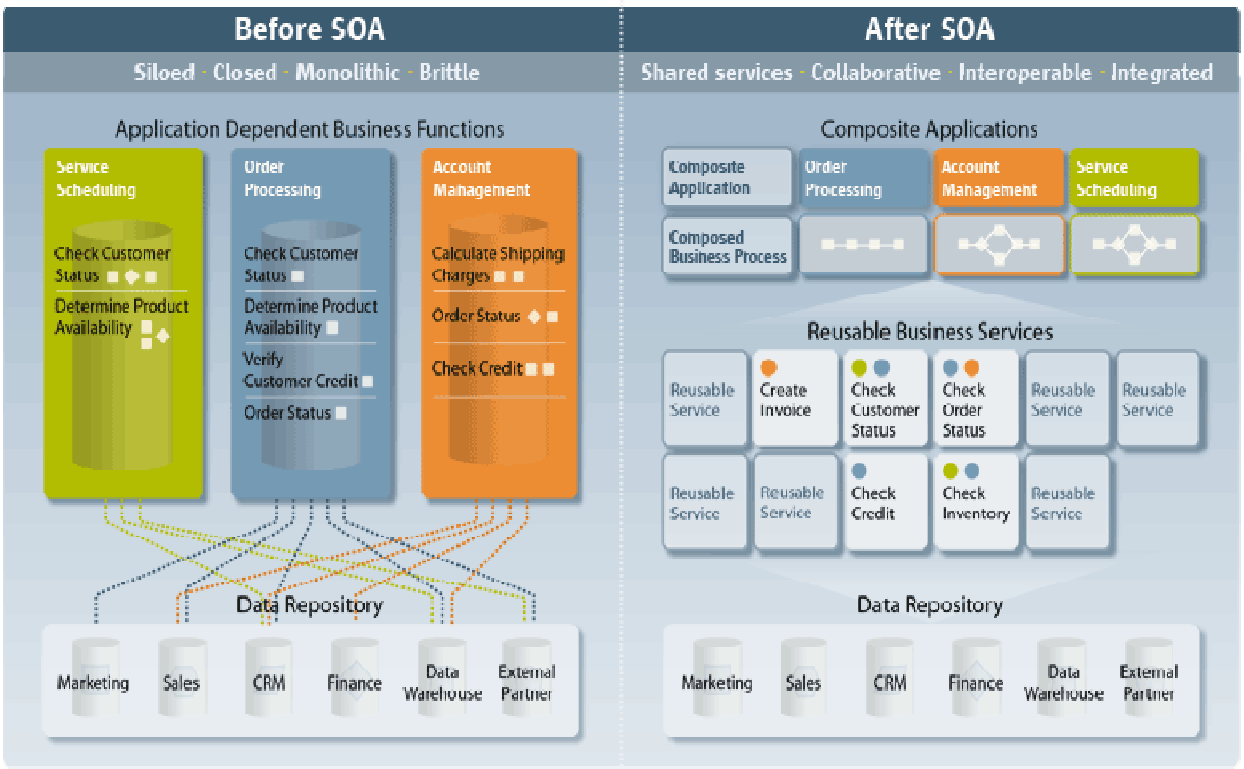

Fig. 58. Posible sistema de información antes y después de aplicarse SOA. [27]

Una vez explicados los problemas, podemos ver como se han solucionado a partir del escenario representado después de SOA. Con SOA, lo primero que observamos es que se han estandarizado los procesos de negocio, y éstos se han construido a partir de una composición de servicios, por ejemplo, lo que antes era una aplicación monolítica *Order Processing* se ha convertido en una composición de servicios, compuesta por los servicios *Check Customer Status, Check Inventory, Check Credit* y *Check Order Status*. También observamos que ahora ya no existe duplicidad en la funcionalidad, gracias a la reutilización de servicios, observamos que ahora el servicio *Check Order Status* se utiliza tanto en *Order Processing* como en *Account Management.* Si ahora se realiza un cambio en la lógica de negocio de *Check Order Status*,

solamente se deberá modificar este servicio, y no como antes, que deberíamos de haber modificado todas las aplicaciones que tuvieran implementada esta funcionalidad. Otra cosa que podemos observar es que también se han estandarizado los servicios, *Order Status* ha pasado a ser *Check Order Status*, ahora hay uniformidad en todos los servicios de comprobación, *Check Customer Status, Check Order Status* y *Check Inventory*. Por último, vemos que ya no existe un enmarañado de conexiones entre las aplicaciones y los repositorios de datos. Esto es debido al uso de un ESB, que como hemos dicho, haciendo de intermediario reduce el número de conexiones necesarias. Ahora las aplicaciones se conectan al ESB y las bases de datos también. De esta forma, un cambio en la base de datos CRM solamente implica realizar un cambio dentro del ESB, y automáticamente todas las aplicaciones que se conectan al ESB para acceder a la base de datos CRM ya verán la modificación, sin tener que ir aplicación por aplicación a realizarla.

Ya hemos visto las ventajas desde un punto de vista tecnológico, también las podemos encontrar des de un punto de vista estratégico, como se refleja en la Figura 59.

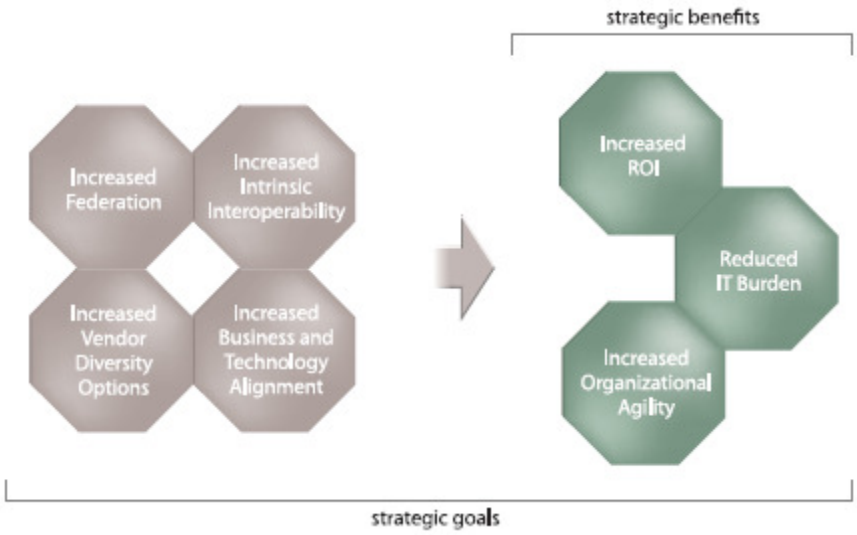

Fig. 59. Beneficios estratégicos de la arquitectura SOA. [8]

Gracias a una SOA Contemporánea, podemos conseguir varios objetivos estratégicos. El primero es un incremento de la federación de los sistemas de información. Con esto queremos decir que conseguimos un entorno unificado gracias al uso de los servicios, donde cada uno de ellos encapsula un segmento de la empresa a partir de los procesos de negocio que representa. Gracias a este entorno federado se consigue unidad a nivel tecnológico. El uso de estándares *vendor-neutral* que proveen los servicios web, por un lado como hemos visto, incrementan de forma intrínseca la interoperabilidad, promoviendo nuevos modelos de negocios que involucren a stakeholders externos, como por ejemplo clientes o proveedores, fomentando así los negocios B2B (*Business to Business*) como podrían ser las colaboraciones que se dan en agencias de viajes con tour-operadores, donde se ofrecen servicios web para reservas de hoteles y vuelos. Por otro lado, el uso de estándares incrementa la diversidad de

posibles proveedores tecnológicos. En efecto, el uso de estándares elimina el hecho de centrarse en un proveedor específico con su tecnología particular, ahora hay la posibilidad de contratar cualquier proveedor que trabaje con dichos estándares. Otro objetivo estratégico habitual es el alineamiento entre negocio y tecnología, este alineamiento es necesario debido a que si cambia la naturaleza o la dirección del negocio, cuanto más alineada esté la tecnología al negocio, menos cambios se deberán acometer en los sistemas de información. Este alineamiento se consigue con el alto grado de abstracción que se consigue con la capa de servicios, sobre todo si se aplica una estrategia de despliegue de SOA *top-down*, consiguiendo que el diseño de los servicios provenga directamente de la lógica de negocio de la organización. Es por este motivo que se aconseja la estrategia *top-down* y no la *bottom-up*, ya que con *bottom-up* partimos de la lógica de aplicación y no tendremos un alto alineamiento con el negocio.

Estos objetivos estratégicos se convierten en una serie de beneficios. El más destacado es el incremento del ROI (*Return of Investment*) o retorno de la inversión. El ROI mide los beneficios obtenidos en un futuro a partir de una inversión inicial. Un ejemplo de cálculo de ROI sería el mostrado en la Figura 60. En esta figura se comparan dos soluciones, una basada en sistemas tradicionales, como podrían ser las arquitecturas distribuidas ya estudiadas, y la otra es una solución SOA. Se puede observar que en la etapa inicial cuando se realiza la inversión, la solución SOA requiere una mayor inversión, debido a que se necesita todo un proceso de análisis de los procesos de negocio para la construcción de servicios. Pero como se observa, a medida que pasa el tiempo se recupera la inversión inicial en mayor medida que con las soluciones tradicionales. Esto es debido principalmente a la reusabilidad que aportan los

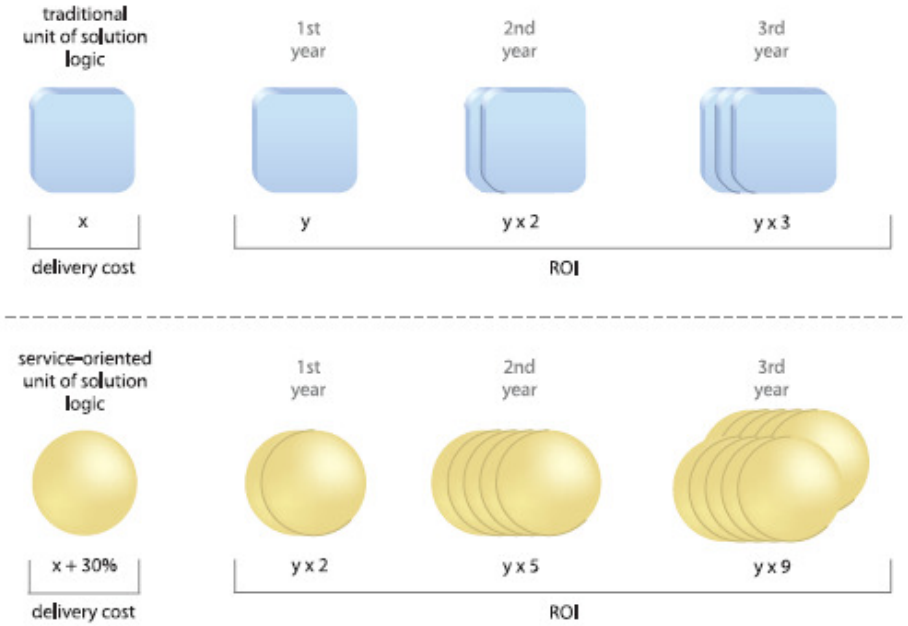

Fig. 60. Cálculo del ROI en proyectos SOA. [8]

servicios ya que son potencialmente más reusables que las soluciones tradicionales debido a que son también interoperables, aumentando así su alcance potencial. Otro factor que hace que el ROI sea mayor es que SOA, al estar compuesta de un entorno homogéneo formado por servicios, no depende de una gran variedad de sistemas heterogéneos que pueden hacer incrementar el presupuesto del departamento de sistemas de información destinado al mantenimiento de dichos sistemas. Por consiguiente, se reduce también la carga de trabajo en este departamento.

9.7 *INCONVENIENTES DE LA ARQUITECTURA SOA*

SOA también tiene sus desventajas, no siempre va a ser aconsejable operar bajo una arquitectura de software como la que se propone. A continuación detallamos estos aspectos donde una SOA Contemporánea no es tan recomendable.

De la misma manera que se ha dicho que la arquitectura SOA es útil en situaciones con gran variedad de sistemas con tecnologías diferentes, se da el caso contrario, cuando tenemos que todas las aplicaciones y sistemas están constituidas con una misma tecnología, o también, que no haya necesidad de relacionarse con entidades ajenas a la empresa que justifiquen la utilización de servicios. En estos casos seguramente resultará menos costoso y más provechoso el uso de sistemas de comunicación de la propia tecnología con la que se trabaja. En el caso de Java por ejemplo, el uso de componentes EJB es una buena solución.

Otro inconveniente importante es el uso de contratos de servicio estandarizados debido a que estos hacen que de por sí SOA tenga una base “pesada” en cuanto a transferencia de datos, como hemos visto en el estudio de rendimiento de los servicios web. Si encima se le añade a los mensajes intercambiados gran información de datos, como podrían ser datos geográficos para mapas, se penaliza en exceso el rendimiento. Existe un protocolo con un rendimiento mayor, el protocolo REST (*Representational State Transfer*). REST es un protocolo similar a SOAP, basado en XML y HTTP, que simplifica la estructura del mensaje. El problema que tiene este protocolo es que con la simplificación del mensaje se pierde la inteligencia y autogobierno que proporciona SOAP, también hay que tener en cuenta que en entornos empresariales los mensajes no suelen contener elevados volúmenes de datos. Por lo tanto, SOAP es más útil en entornos empresariales y REST se utiliza en servicios con alta frecuencia de uso y alto volumen de datos, como podría ser el servicio de Google Maps. Si queremos mejorar el rendimiento de SOAP dentro de SOA, deberemos de usar herramientas que nos ayuden a optimizar el rendimiento, como por ejemplo el *XML-binary Optimized Packaging* (XOP) o el *SOAP Message Transmission Optimization Mechanism* (MTOM), ambas especificaciones elaboradas por W3C.

Una vez finalizado el capítulo dedicado a SOA, nos centraremos en el estudio del ESB, concepto que hemos ido introduciendo y del que hemos visto su importancia a la hora de construir una arquitectura orientada a servicios. En el próximo capítulo explicaremos con más

detalle el concepto de integración y cómo han evolucionado las herramientas software para dar cobertura a este aspecto hasta la llegada de los ESB.

10. *ENTERPRISE SERVICE BUS* (ESB)

Un ESB (*Enterprise Service Bus*) es un combinado de tecnologías software que media entre las aplicaciones empresariales y permite la comunicación entre todos los actores de la arquitectura SOA. Como se ha comentado en el capítulo anterior, un ESB actúa como intermediario reduciendo el número de conexiones necesarias entre publicadores de servicios y consumidores y también como elemento de integración. A continuación, entraremos en detalle en el concepto de integración y posteriormente explicaremos como hace de intermediario un ESB.

La necesidad de integración a nivel empresarial surge al disponer de un escenario formado por múltiples aplicaciones construidas a partir de tecnologías diferentes. La evolución hasta un escenario heterogéneo puede tener varios orígenes. Una posibilidad es el propio crecimiento de la empresa, que implica la expansión o creación de nuevas áreas de negocio, que necesitarán nuevas aplicaciones automatizadas que den soporte a los nuevos procesos de negocio. Debido a este cambio constante, no siempre se realizarán los desarrollos software con la misma tecnología, puede ser que con el paso del tiempo la tecnología con la que se construían las aplicaciones haya quedado obsoleta, o los nuevos procesos de negocio tienen unos requerimientos que no se pueden cubrir con la tecnología actual y se deben buscar nuevas tecnologías que sí soporten las nuevas necesidades. Otras causas que pueden provocar la necesidad de integración son las adquisiciones y fusiones de empresas, debido a que se incorporan o se deben rediseñar los procesos de negocio de la organización. En estos casos, también habrán problemas de modelos de datos entre las aplicaciones propias de la empresa y las nuevas, ya que cada empresa tiene su idiosincrasia y su forma de trabajar, mientras en una empresa se trabaja con ciertos conceptos de negocio, puede ser que en otra empresa estos conceptos se llamen de otra manera o directamente no existan. En cualquier caso pues, ya sea por aplicaciones con diferentes tecnologías y/o con aplicaciones que trabajan con modelos de datos dispares, se deberá de realizar un trabajo de integración.

Como respuesta a esta necesidad, surgieron las plataformas EAI (*Enterprise Application Integration*). Un EAI es un componente software que actúa como punto central de comunicación de todas las aplicaciones, siguiendo una distribución llamada *Hub & Spoke*.

Una tipología de integración *Hub & Spoke* (Figura 61) usa un gestor (bróker) centralizado llamado *Hub* y una serie de adaptadores (*Spoke*) que conectan las aplicaciones al *Hub*. Los adaptadores transforman la información recibida de las aplicaciones y la convierten a un tipo de datos que el *Hub* reconoce. Después el *Hub* realiza las transformaciones necesarias para adaptarlas a las aplicaciones destino y por último, el *Hub* direcciona (o como se dice comúnmente, rutea) los mensajes del sistema origen al sistema destino.

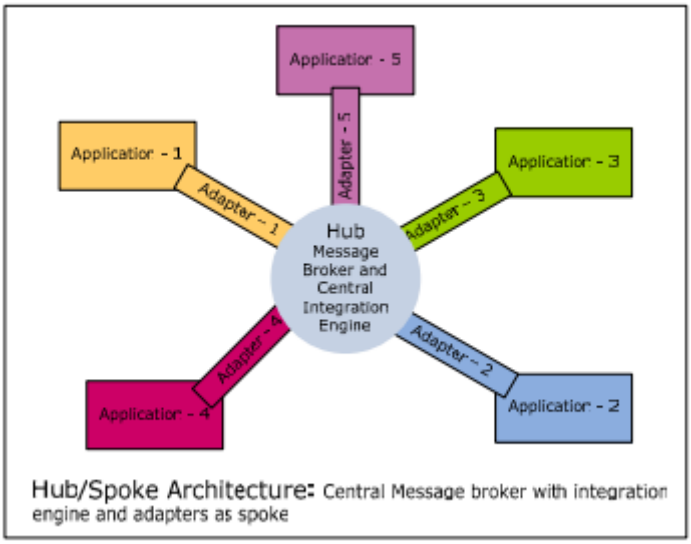

Fig. 61. Arquitectura Hub/Spoke. [30]

En la Figura 61 podemos observar como tenemos cinco aplicaciones diferentes y en medio de todas ellas se encuentra el *Hub* o *Broker*, que actúa como motor central para la integración. El *Hub* debe disponer de los adaptadores especiales para cada aplicación.

Esta arquitectura presenta una serie de problemas. El primer problema es que se deben disponer tantos adaptadores como aplicaciones con diferentes tecnologías tengamos. Y si hay cambios en dichas tecnologías, con mucha probabilidad se deberán realizar cambios en los adaptadores, esto se traduce en un alto acoplamiento entre aplicaciones y conectores. El otro problema que acarrea esta tipología es el hecho de tener el *Hub* como elemento central. Al estar todo centralizado en el *Hub*, se pueden generar problemas de cuello de botella a nivel de comunicaciones penalizando así el rendimiento o directamente, si hay un fallo grave se compromete la integridad de todo el sistema.

Debido a estos problemas, surgió el ESB para reemplazar la antigua EAI, y cambiar la topología *Hub & Spoke* por una topología del tipo *Bus*. Esta tipología se representa de forma similar a los buses de datos existentes en el *hardware* informático o a la topología bus de redes informáticas, como un canal donde se conectan todos los dispositivos y por el cual circulará la información (Figura 62).

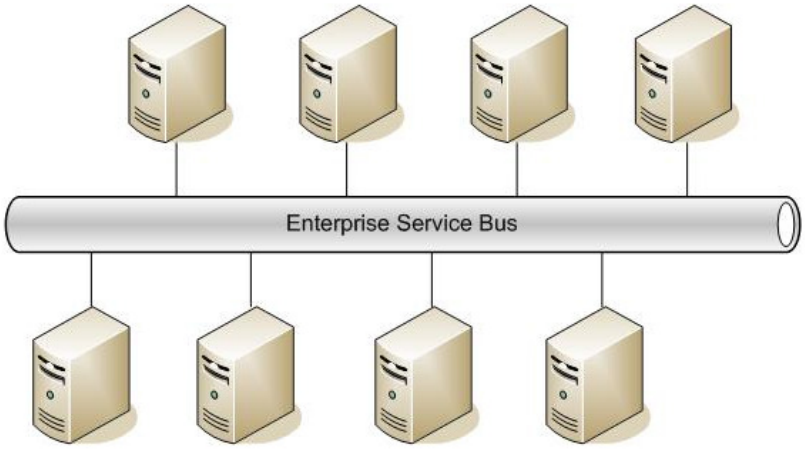

Fig. 62. Topología tipo Bus. [31]

Hay varias ventajas aportadas por un ESB. La primera ventaja aportada es que un bus de datos puede estar distribuido en varias máquinas físicas, eliminando así el problema subyacente de la arquitectura *Hub & Spoke*. Ahora, si hay un problema en un servidor no se compromete toda la infraestructura. Otra ventaja del ESB es que está orientado a servicios y por lo tanto está basado en estándares, en cambio, EAI con su topología *Hub & Spoke* solían ser soluciones propietarias [32], eso implica que los conectores también lo eran. Ahora con ESB, los servicios se podrán conectar al Bus con conectores estandarizados, reduciéndose así el acoplamiento, normalmente como servicios web pero también habrán otros tipos. Antes de ver estos tipos de conectores y otras funcionalidades aportadas por el ESB que lo diferencian de su predecesora EAI, explicaremos cómo un ESB actúa de intermediario.

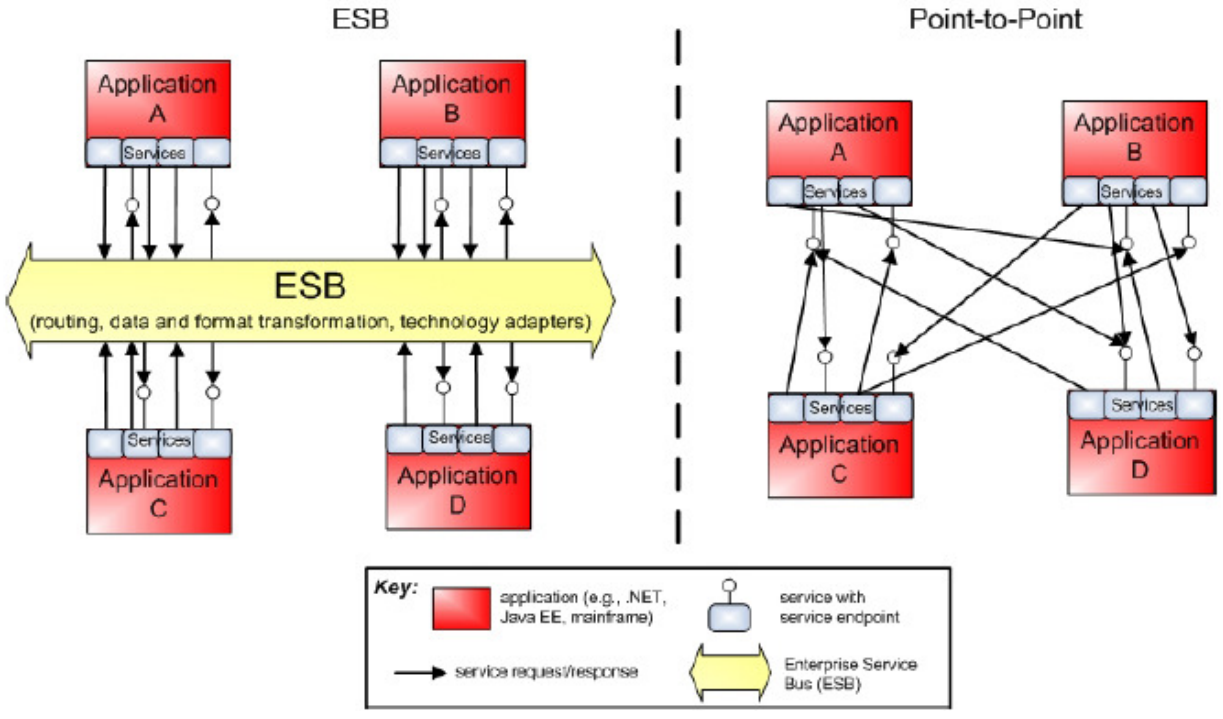

Fig. 63. Diferencia entre conexiones mediante ESB y conexiones punto a punto. [33]

Una consecuencia directa del uso de un ESB como plataforma intermedia de integración entre servicios, o como se suele llamar en inglés, *middleware*, es que se consigue evitar las conexiones directas o conexiones punto a punto. En el escenario representado a la derecha en la Figura 63, se muestra un conjunto de aplicaciones que interaccionan entre ellas mediante este tipo de conexiones, creando un escenario enmarañado fuertemente acoplado conocido como conexiones tipo espagueti. En esta situación, teniendo n aplicaciones y considerando solamente una conexión posible entre dos aplicaciones, habrá un total de $n \cdot (n-1)/2$ conexiones. En cambio, como se refleja en el escenario de la izquierda de la Figura 63, y teniendo en cuenta igualmente una conexión posible por aplicación, habrá n conexiones. Con lo cual ya no hacen falta tantas conexiones, cada aplicación se conectará al Bus y a partir de él se direccionará a la aplicación necesaria, gracias a la función de *routing* que realiza el bus.

Como se refleja dentro del elemento Bus de la Figura 63, un ESB aporta una serie de funcionalidades que se suman a la característica primaria de integración y que hacen de él una evolución de las plataformas EAI. En el siguiente punto veremos en detalle qué otras características constituyen un ESB.

10.1 *Características de un ESB*

A continuación, enumeraremos las características que posee un ESB, comenzando por dos características relacionadas con la integración. La primera hace referencia a los tipos de comunicación que se pueden establecer dentro de un ESB y la siguiente característica detallará qué tipos de conectores existen para facilitar la integración.

Comunicaciones síncronas y asíncronas.

Las comunicaciones entre dos partes, por ejemplo entre dos servicios, pueden ser de dos tipos: síncronas o asíncronas. Las comunicaciones síncronas son aquellas en que el responsable del envío de la información permanece bloqueado esperando a que llegue una respuesta del receptor antes de realizar cualquier otra tarea, habiendo un tiempo de espera limitado para recibir respuesta. Como ejemplo de comunicación síncrona podemos tener un sistema de *login* de usuario en un sistema. El usuario introduce su *login* y *password* y el sistema verifica estas credenciales, si el sistema no está disponible por algún motivo, se mostrará un mensaje de error advirtiendo al usuario del problema. El otro tipo de comunicación es la comunicación asíncrona. En este tipo de comunicaciones, la parte que envía la información o mensaje continúa haciendo otras tareas inmediatamente después de enviar el mensaje al receptor, no habiendo un tiempo limitado de espera de la respuesta. Una vez se ha enviado la información, ésta se guarda en una cola de mensajes para que el receptor pueda tratarla cuando pueda. Un ejemplo de comunicación asíncrona podría ser el envío de un pedido. El emisor del mensaje envía la información relativa al pedido y seguidamente puede dedicarse a otras tareas.

Un ESB pues, permite estos dos tipos de comunicación entre los servicios conectados a él.

Integración entre diferentes lenguajes y tecnologías.

A diferencia de un EAI, que utiliza conectores propietarios específicos de cada aplicación, un ESB debe permitir el máximo número de estándares de comunicación tal y como se muestra en la Figura 64: HTTP, HTTPS, JMS (*Java Message Service*), SOAP, REST, XML, RPC, etc…

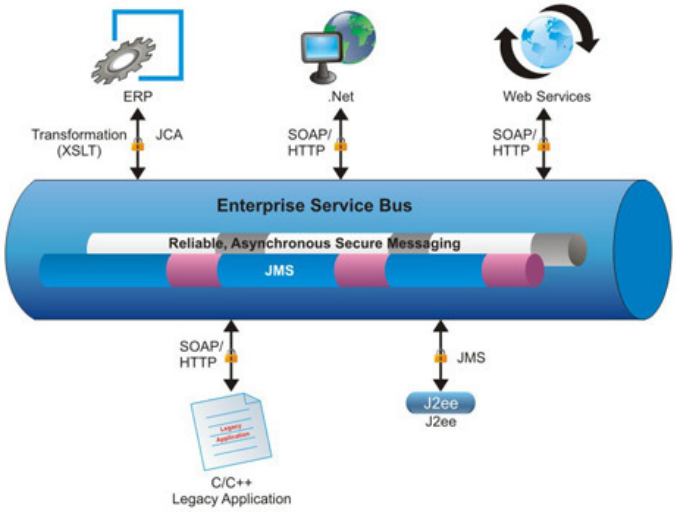

Fig. 64. Un ESB es un componente que permite integrar gran variedad de tecnologías. [34]

Como se observa en la Figura 64, gracias a los conectores como SOAP/HTTP o JMS (un servicio de mensajería Java que trataremos con detalle más adelante), podemos integrar diferentes aplicaciones y/o servicios, como servicios web, aplicaciones .NET o aplicaciones J2EE. En el conector asociado a los sistemas de gestión empresariales (ERP), se observa la notación *Transformation* (XSLT), que se explicará a continuación.

Conversión y mapeo de datos.

Se pueden realizar conversiones y transformaciones del mensaje origen a partir del estándar XSLT desarrollado por W3C. XSLT (*Extensible Stylesheet Language Transformations*) son unas hojas de estilo que realizan transformaciones de documentos XML o transformaciones a otros formatos como por ejemplo HTML. XSLT es muy usado para separar el contenido de la capa de presentación en el entorno web. Podemos tener un XML con el contenido a presentar

en una página web, se le aplica una hoja de estilo XSLT que realiza una transformación añadiendo la capa de presentación en HTML al contenido, por ejemplo creando tablas HTML para presentar datos, dando como resultado la página web.

Otra utilización del XSLT es el mapeo de datos, esto nos permite relacionar información entre dos estructuras diferentes, existen herramientas visuales que facilitan este proceso tal y como se muestra en la Figura 65, con un ejemplo de una utilidad proporcionada por el entorno de desarrollo JDeveloper de Oracle. En ambos extremos de la pantalla hay dos estructuras con diferentes tipos de datos y con algunos datos en común. En el centro de la pantalla se realizan conexiones de forma gráfica entre los elementos comunes y así realizar el *mapping*.

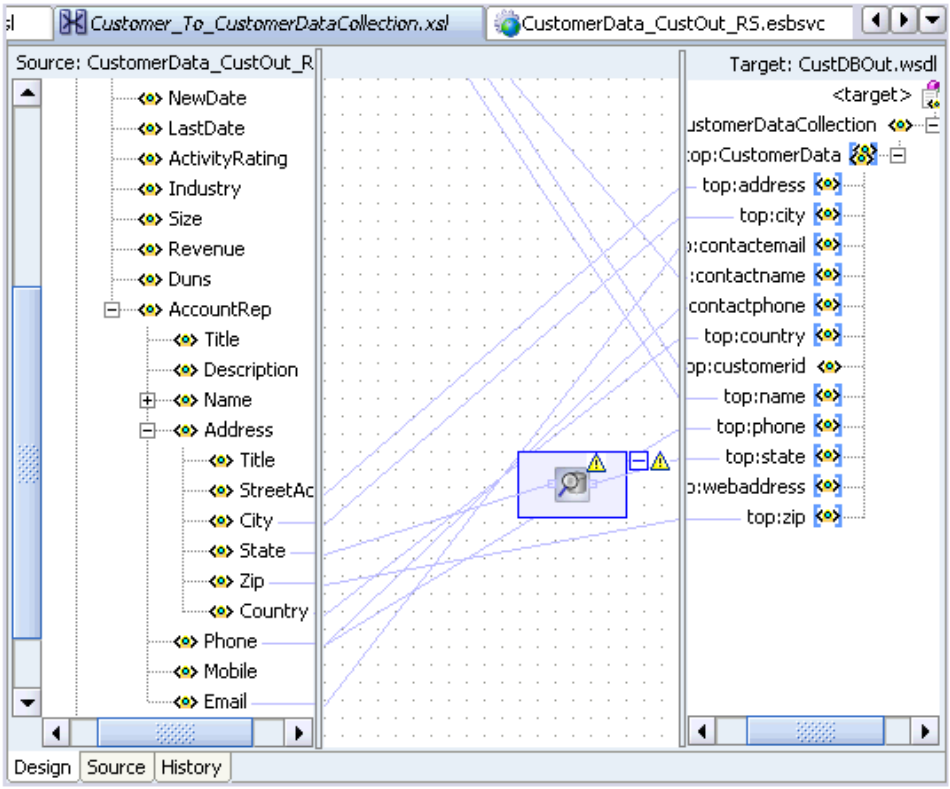

Fig. 65. Herramienta visual para el *mapping* de datos. [35]

Enrutamiento y direccionamiento de mensajes.

Los mensajes que llegan al ESB se pueden enrutar, es decir, se pueden dirigir a un servicio determinado en función de unas reglas. Estas reglas pueden estar descritas en el mismo mensaje, por ejemplo con el uso del estándar WS-Addressing en la cabecera SOAP, o se pueden establecer reglas en el ESB que analicen el contenido del mensaje, lo que se llama *Content-Based Router*. Estas reglas se pueden configurar para analizar la existencia de campos o valores determinados dentro del mensaje, y en función de ellos, direccionar el servicio a un sitio o a otro. También se pueden establecer reglas que determinen que los mensajes deban seguir una ruta predeterminada o crear una lista de recipientes, y cuando un mensaje llegue al ESB, este se propague a todos los miembros de esta lista. También existen

ESB que contienen herramientas que permiten el ruteo y la mediación, como por ejemplo el motor de ruteo y mediación Apache Camel.

Trazabilidad de las operaciones.

Debe poseer un sistema de trazabilidad (*logging)* para poder a posteriori identificar los pasos que han seguido las operaciones involucradas y así observar si ha habido algún problema y en qué momento se ha producido.

Mecanismos de seguridad y garantía de la calidad del servicio (QoS).

Nos referimos a la autenticación (saber quién es quién), autorización (si se está autorizado o no para acceder a un recurso), encriptación y transaccionalidad. Por ejemplo, para la autenticación se puede utilizar Kerberos [36]. Kerberos es un protocolo de autenticación creado por el MIT que permite a dos computadores en una red insegura demostrar su identidad mutuamente de forma segura. Para otras características QoS, como las relacionadas con la seguridad, se puede hacer uso por ejemplo del estándar WS-Security para proporcionar encriptación a los mensajes, y del estándar WS-ReliableMessaging que asegura que los mensajes son enviados correctamente a través de un sistema incluso con presencia de fallos en la red.

Monitorización y control centralizado.

Un ESB debe proveer una interfaz con el usuario para que se pueda administrar debidamente el componente. Hay varias formas de acceder al ESB, puede ser mediante consolas como el cmd de Windows (Procesador de comandos de Windows) o si nos conectamos a una máquina remota y queremos hacerlo de forma segura, existen las consolas SSH (*Secure Shell).* También existe la posibilidad de operar con el ESB a través de interfaces web.

Orquestación de servicios y diseño de workflows de negocio (BPEL).

Ya hemos definido con anterioridad el concepto de *workflow* y como se pueden orquestar los servicios añadiendo una capa de abstracción por encima de los servicios. Dentro de un ESB pues, se pueden generar orquestaciones de servicios a partir del estándar ya mencionado con anterioridad WS-BPEL o también con el uso de Apache Camel, que aparte de ser un motor de ruteo también permite la creación de orquestaciones.

El lenguaje usado para la orquestación es BPEL (*Business Process Execution Language*) , este lenguaje está basado en XML y permite invocar servicios con cierta lógica de negocio añadida, pudiendo crear así flujos de negocio a partir de servicios. También existen herramientas que permiten de forma visual crear estas orquestaciones, como por ejemplo con el editor

JDeveloper tal y como se muestra en la Figura 66, donde se puede observar el modelado de un flujo de negocio.

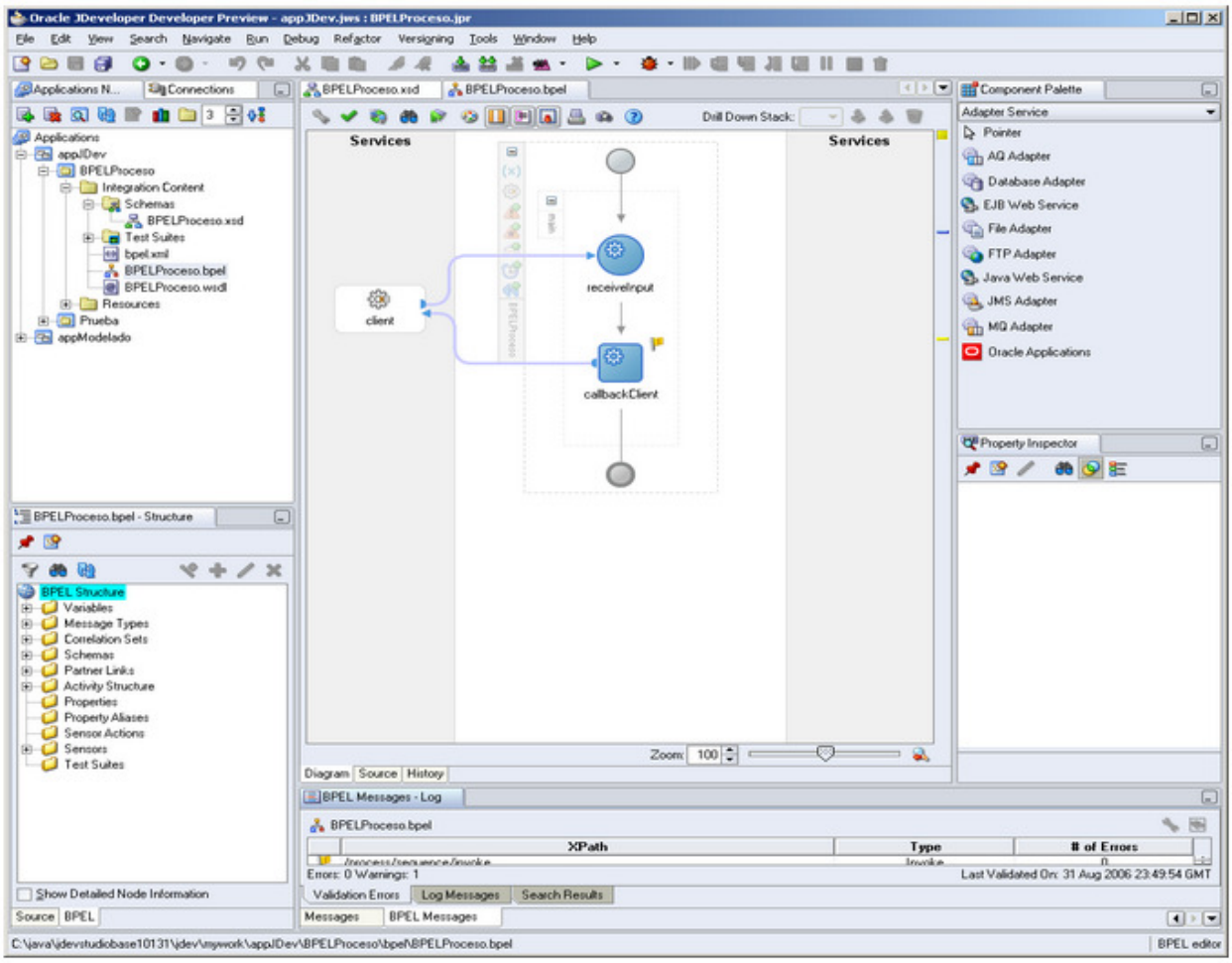

Fig. 66. Creación de un proceso de negocio con BPEL. [37]

Funcionalidad distribuida.

Se puede distribuir un ESB en diferentes servidores, gracias al uso del Bus y su mensajería. No hace falta que toda la infraestructura esté centralizada como en otro tipo de arquitecturas, por ejemplo Hub/Spoke, que tiene el bróker centralizado y un fallo en el mismo puede dejar sin conectividad toda la infraestructura.

10.2 *APLICACIONES ESB*

En la actualidad existe en el mercado una amplia oferta de productos ESB tanto comerciales como de código abierto. Si bien en todos los casos la finalidad de estos productos es básicamente la misma, existen ciertas diferencias sustanciales entre ellos. Los ESB comerciales suelen disponer de ciertas herramientas tipo GUI (*Graphical User Interface*) que facilitan la administración de los distintos componentes, tal y como hemos visto en el punto

anterior con las herramientas visuales para mapeo de datos o creación de procesos de negocio. Otras ventajas aportadas por los ESB comerciales son el soporte técnico y documentación de referencia exhaustiva. Los ESB de código abierto (*opensource*) suelen carecer de interfaces ricas de administración y en general la documentación suele ser menos detallada, pero por el contrario, normalmente cuentan con el respaldo de una comunidad de desarrolladores potente detrás, como suele ser habitual en los proyectos de código abierto. Otra ventaja sin lugar a duda es que no tienen coste de adquisición y los gastos se centrarán en su implantación y mantenimiento. Por último, remarcar el hecho de que al no disponer de tantas herramientas adicionales como los editores visuales, hace que los ESB de código abierto sean normalmente más ligeros y requieran de infraestructuras menos robustas, lo que puede ser una ventaja cuando no se dispone de los recursos necesarios, como por ejemplo, unos servidores lo suficientemente potentes.

Un primer criterio de decisión sobre qué ESB es más conveniente en cada situación será entonces optar por productos comerciales de pago o de código abierto. Si ya se dispone de un proveedor de pago como Oracle o IBM y se dispone del presupuesto suficiente, posiblemente permanecer con estos proveedores sea una buena decisión.

Pero a parte de este criterio, hay otros que hay que tener en cuenta ante una posible toma de decisiones. Existe una firma comercial de referencia que se dedica a realizar este tipo de estudios de mercado: Forrester Research [38]. Esta empresa, entre otros tipos de estudios, elabora periódicamente informes comparativos a partir de múltiples criterios entre los principales proveedores de ESB, ya sean comerciales o de código abierto, como veremos a continuación. Este informe en su versión más reciente del 2011 es el informe *"The Forrester Wave: Enterprise Service Bus, Q2 2011"* [39], realiza una comparativa del mercado actual, donde todos los productos presentados cubren las funcionalidades básicas de integración, como por ejemplo, enrutamiento, transformación de datos, mediación o seguridad. En concreto, se evalúan 109 criterios divididos en 5 áreas funcionales principales, representadas en la Figura 67.

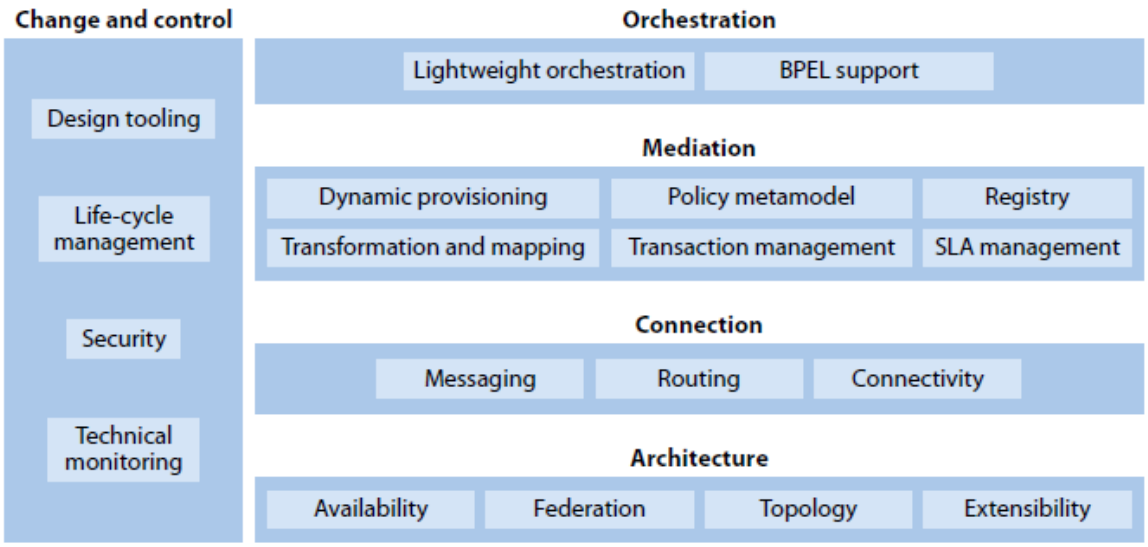

Fig. 67. Áreas de evaluación del estudio sobre soluciones ESB de Forrester Research. [39]

El área de arquitectura se centra en aspectos como la tolerancia a fallos, rendimiento, escalabilidad del sistema, es decir, la capacidad de la plataforma en ofrecer una estabilidad considerable aún cuando el sistema vaya creciendo, las topologías que soporta el ESB y la capacidad de ser extendido a partir de características complementarias. El área de conexión tiene en cuenta factores relativos a estándares en los mensajes, protocolos de comunicación y conectividad. Con el área de mediación se pretende analizar aspectos como la transformación y mapeo de datos, gestión de transacciones o la coordinación de los acuerdos de nivel de servicio (SLA). El área de Orquestación se centrará en esas características relativas a la orquestación de servicios por un lado y por el otro en la gestión de procesos de negocio mediante el lenguaje BPEL. Por último, tenemos el área de Cambio y control que se centrará en factores como las posibles herramientas de diseño disponibles, seguridad, monitoreo y control del ciclo de vida de los componentes asociados al ESB.

Forrester, en su estudio, analiza estas áreas funcionales desde tres perspectivas diferentes de alto nivel: oferta actual, estrategia y presencia en el mercado. De los 109 criterios, 79 son asociados a las características ofrecidas actualmente por cada proveedor. Otros 15 criterios se centran en la perspectiva de la estrategia del producto: el coste de las soluciones, alianzas estratégicas y verificación de referencias de clientes. Los 15 criterios restantes evalúan la penetración en el mercado de cada proveedor, a partir de instalaciones realizadas, nuevos clientes e ingresos anuales obtenidos de la venta de los productos ESB.

Forrester se centra en 9 proveedores y 11 referencias individuales ESB tal y como se muestra en la Tabla 3.

Vendor	Product evaluated	Product version evaluated	Version release date
FuseSource	Fuse ESB	4.0	October 2008
IBM	WebSphere Enterprise Service Bus (WESB)	7	December 2009
	WebSphere Enterprise Service Bus Registry Edition (WESBRE)	7	October 2010
	WebSphere Message Broker (WMB)	7	October 2009
MuleSoft	Mule ESB	3	September 2010
Oracle	Oracle Service Bus	11 g R1	April 2010
Progress Software	Sonic ESB	8	March 2010
Red Hat	JBoss ESB	5.0.2	July 2010
Software AG	webMethods ESB Platform	8.0	December 2009
Tibco Software	ActiveMatrix Service Bus	3.0.1	August 2010
WSO2	WSO2 ESB	3.0	May 2010

Tabla 3. Proveedores ESB evaluados por el estudio de Forrester Research. [39]

El resultado de su estudio es el detallado en la Figura 68, a partir de las tres perspectivas de alto nivel comentadas anteriormente: Oferta actual en el eje Y, Estrategia en el eje X y el área de la circunferencia que representa a cada ESB, indica su presencia en el mercado.

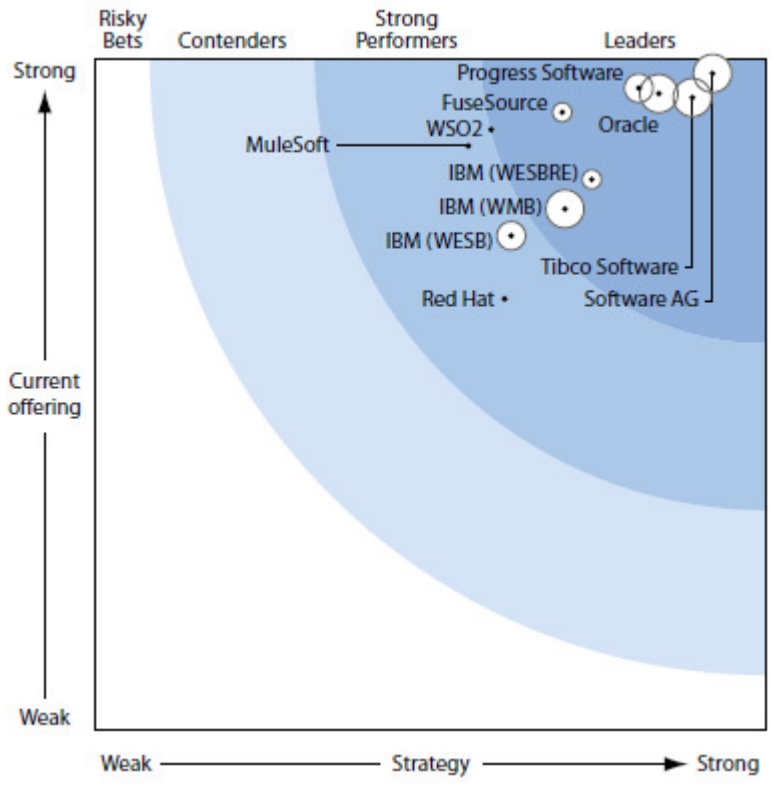

Fig. 68. Resultado del estudio de soluciones ESB de Forrester Research. [39]

Se puede observar como las compañías Software AG, Tibco, Oracle, Progress Software e IBM lideran la clasificación, todas estas organizaciones ofrecen productos comerciales de pago. Se debe remarcar que dentro de la categoría de líderes de mercado pero con menor puntuación, se encuentran dos proveedores de código abierto: FuseSource y WSO2. Teniendo entre estos dos, más presencia de mercado FuseSource con su producto Fuse ESB [40].

Como en el presente estudio se debe crear un ejemplo de uso de un ESB, deberemos escoger un producto en concreto. Partiendo de la base de que no disponemos de presupuesto ni de un entorno empresarial que justifique la adquisición de un producto propietario, nos deberemos centrar en productos de código abierto. Los proveedores *opensource* presentes en el estudio de Forrester son Fuse ESB, MuleSoft y Red Hat. Entre ellos, es Fuse ESB quien lidera la clasificación, con unos valores superiores a los demás tanto en oferta, como en estrategia y presencia en el mercado. Fuse ESB es fundamentalmente un producto basado en el ESB Apache ServiceMix [41], al que opcionalmente se puede optar por una versión de pago si se desea para obtener beneficios adicionales, entre ellos, disponer de un soporte técnico.

En base a este estudio, parece que Fuse ESB es un buen candidato. Podemos utilizar la herramienta de Google de tendencias de búsqueda para comparar y medir el interés que suscitan los siguientes ESB: ServiceMix, Mule ESB [42], WSO2 ESB [43] y Jboss ESB [44]. El resultado es de esta comparativa es el mostrado en la Figura 69.

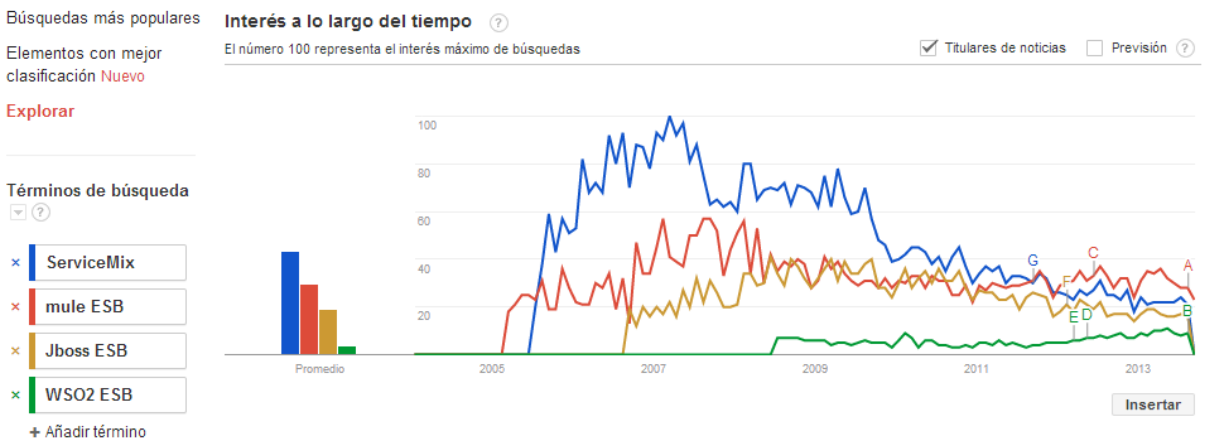

Fig. 69. Tendencias de búsqueda en Google de los principales ESB *opensource*. [45]

Observamos que en la actualidad Mule ESB tiene ligeramente un interés de búsqueda superior a ServiceMix, pero el promedio de interés máximo de búsquedas en los últimos 5 años es para ServiceMix.

Así pues, en base al estudio de Forrester Research, escogeremos para nuestro proyecto la herramienta Fuse ESB basado en Apache ServiceMix, sobre el que entraremos en detalle en el siguiente capítulo.

11. FUSE ESB (APACHE SERVICEMIX)

Como se ha mencionado anteriormente, Fuse ESB es un producto de código abierto basado en Apache ServiceMix, que a su vez, está basado en estándares Java, como iremos introduciendo en adelante. El hecho de que FuseSource pueda crear un producto a partir de otro existente es debido a que ServiceMix está publicado bajo licencia Apache, que es una licencia de software libre creada por la *Apache Software Foundation* (ASF) [46] que permite al usuario del software la libertad de usarlo, distribuirlo, modificarlo y distribuir versiones modificadas de ese software. A parte de la versión libre de coste, o como se suele decir edición comunitaria (*communtity* edition), con Fuse ESB se puede optar a una versión de pago (*enterprise edition*) donde se puede pagar por unos servicios adicionales como por ejemplo, instaladores, herramientas de test, optimización, actualizaciones incrementales, etc.

En el siguiente punto veremos cuáles son las características de un ServiceMix.

11.1 *SERVICEMIX: ARQUITECTURA*

En este apartado se detallan los componentes que conforman la arquitectura del ESB ServiceMix. Introduciremos los conceptos que conforman dicha arquitectura a partir de la Figura 70, describiéndolos brevemente para tener un concepto global de la arquitectura y posteriormente realizando una explicación más detallada de cada parte.

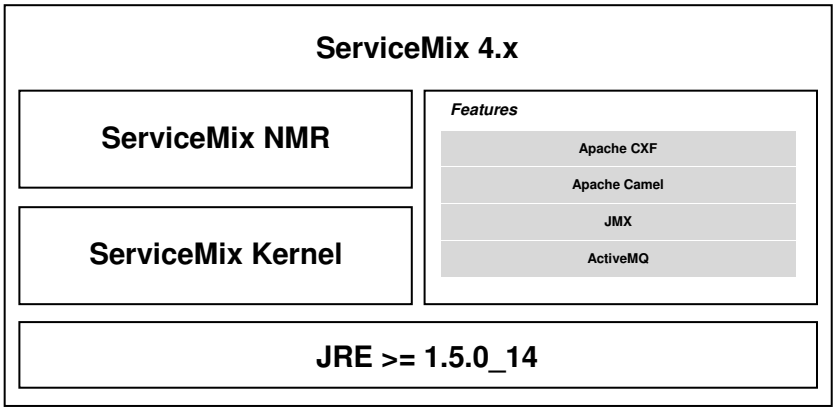

Fig. 70. Arquitectura ServiceMix.

- **JRE (Java Runtime Environment)**

  ServiceMix se ejecuta en un entorno JAVA con versión superior o igual a 1.50_14. El servidor sobre el cual se ejecuta ServiceMix debe tener instalada la JRE. JRE es el entorno que permite que aplicaciones escritas en Java puedan ejecutarse en una máquina. Dentro de JRE, la máquina virtual de Java JVM (*Java Virtual Machine*) es la

encargada de traducir las clases Java compiladas a lenguaje de más bajo nivel entendible por el procesador del ordenador.

- **ServiceMix Kernel**

  Es el núcleo del ServiceMix, el contenedor principal.

- **ServiceMix NMR (*Normalized Message Routing*)**

  Es el bus propiamente dicho, por donde circulan los mensajes que intercambian las aplicaciones conectadas al bus.

- **Prestaciones tecnológicas (Features)**

  Son un conjunto de utilidades desplegadas en ServiceMix con diferentes funciones, como veremos más adelante.

11.1.1 ServiceMix Kernel

Antes de hablar del Kernel, se debe introducir el concepto de contenedor OSGi debido a que éste constituye el motor principal del Kernel y determina la forma en que opera internamente el ESB.

Contenedor OSGi

OSGI (*Open Services Gateway Initiative*) es una especifiación Java que propone un entorno de programación basada en módulos, a diferencia de los entornos Java convencionales donde no existe el concepto de componente modular. En un entorno Java estándar, existen las clases Java, éstas se engloban en paquetes y éstos paquetes pueden constituir aplicaciones o librerías para ser utilizadas por otras aplicaciones. Sea de la forma que sea, se crean ficheros de aplicación monolíticos. Por ejemplo, una aplicación web creada en java se constituye a partir de un único fichero con extensión “.war”, o si elaboramos un conjunto de librerías Java para proveer cierta funcionalidad, éstas constituyen un archivo “.jar”. En este escenario, si queremos realizar un cambio en un sub-módulo interno de la aplicación, deberemos recompilar toda la aplicación para crear de nuevo el fichero war o jar. Un entorno modular da un paso más y aporta ventajas respecto las aplicaciones monolíticas, entre ellas, encontramos la facilidad al cambio, podemos cambiar un módulo por otro sin afectar al funcionamiento global de la aplicación, también facilita el desarrollo paralelo de módulos pudiendo trabajar diferentes equipos en diferentes módulos a la vez, se fomenta la reutilización y resulta más fácil de realizar test ya que se pueden testear los módulos por separado. De esta forma se construye un entorno donde los módulos están débilmente acoplados. Tal y como se ve pues, podríamos decir que un contenedor OSGi tiene varias

similitudes a un entorno SOA, y no es difícil encontrarse con analogías del tipo “OSGi equivale a SOA dentro de la JVM” [47] , donde los módulos (llamados *bundles* en inglés), serían análogos a los servicios.

Otra ventaja aportada por la plataforma OSGi, es que podemos tener en el contenedor unas librerías iguales pero de diferentes versiones, ya que puede ser que las diferentes aplicaciones que las usan deban usar versiones distintas de una mismo conjunto de funcionalidades. Para entender bien este concepto, se debe explicar primero la gestión de las librerías dentro de la JVM. Por defecto, una JVM tiene un cargador de clases (*classloader*) en común para todas las aplicaciones. El cargador de clases carga dinámicamente las clases Java contenidas en las librerías alojadas en el servidor y en las aplicaciones, para que éstas puedan ser utilizadas. Esto puede generar problemas, debido a que si en el servidor hay una librería en concreto y una aplicación contiene la misma librería, el servidor no sabrá qué clases se deben cargar, si las de la librería alojada en el servidor, o las de la librería de la aplicación. Este error es frecuente en programación, dando lugar a las clásicas excepciones Java ClassCastException o NoClassDefFoundError. OSGi soluciona este problema, haciendo que las dependencias entre librerías y módulos sean declarativas. Esto significa que se indica explícitamente qué librerías va usar cada modulo, evitando así conflictos y pudiendo tener librerías iguales con versiones diferentes.

A continuación explicaremos como están constituidos los *bundles* y como se realizan las declaraciones de librerías dentro de ellos. Dentro de OSGi, cada *bundle* se construye a partir de un fichero jar. Un fichero jar, es como un fichero zip, dentro de él habrá una estructura de archivos y carpetas. Dentro de esta estructura, encontraremos las clases Java de la funcionalidad aportada por el *bundle* y un fichero llamado MANIFEST.MF, que contendrá la información relativa al *bundle*, es decir, sus metadatos. Por lo tanto:

Bundle OSGi = JAR + MANIFEST.MF con METADATOS OSGi

Los metadatos más significativos que podemos encontrar dentro del fichero manifest están marcados en cuadrados rojos en la Figura 71, donde se muestra un ejemplo de fichero manifest del *bundle* Apache Commons Logging, componente que proporciona unas librerías para hacer trazas o *logs* dentro de las aplicaciones Java . Veamos cuáles son dichos metadatos:

- **Bundle-Name:** El nombre del *bundle*, debe ser lo bastante claro para ser legible por las personas.

- **Bundle-SymbolicName:** Es el identificador unívoco del *bundle* dentro del contendor OSGi.

- **Bundle-version:** Versión del *bundle*. El par formado por Bundle-SymbolycName y Bundle-Version permite referenciar una versión de un *bundle*. Es de esta forma que podremos tener identificados dentro de un mismo contenedor OSGi dos *bundles* iguales pero de versiones diferentes.

- **Export-package:** Declara los paquetes que se exponen fuera del *bundle*. Esto significa que otros *bundles* podrán usar los paquetes expuestos en esta lista, los que no estén en esta lista solamente serán de uso interno del *bundle*.

- **Import-Package:** Declara las dependencias de paquetes externos que el *bundle* necesita para su funcionamikento. Podrían ser paquetes que otro *bundle* ha puesto en su export-package. También se pueden especificar una versión o rango de versiones concretos que se requieren. En la Figura 71 podemos observar por ejemplo, que para el paquete javax.servlet se requiere una versión comprendida entre la 2.1.0 (incluida) y la versión 3.0.0 (no incluída).

```
Manifest-Version: 1.0
Export-Package: org.apache.commons.logging;version="1.1.1",org.apa
 commons.logging.impl;version="1.1.1";uses:="javax.servlet,org.apa
 avalon.framework.logger,org.apache.commons.logging,org.apache.log
 .apache.log4j"
Implementation-Title: Jakarta Commons Logging
Implementation-Version: 1.1.1
Bundle-Classpath: .
Built-By: dlg01
Specification-Vendor: Apache Software Foundation
Bundle-Name: Apache Commons Logging
Created-By: Apache Maven
X-Compile-Source-JDK: 1.2
Implementation-Vendor: Apache Software Foundation
Bundle-Vendor: SpringSource
Implementation-Vendor-Id: org.apache
Build-Jdk: 1.4.2_16
Bundle-Version: 1.1.1
Specification-Title: Jakarta Commons Logging
Bundle-ManifestVersion: 2
Import-Package: javax.servlet;version="[2.1.0, 3.0.0)";resolution:
 ional,org.apache.avalon.framework.logger;version="[4.1.3, 4.1.3]"
 olution:=optional,org.apache.log;version="[1.0.1, 1.0.1]";resolut
 =optional,org.apache.log4j;version="[1.2.15, 2.0.0)";resolution:=
 onal
Bundle-SymbolicName: com.springsource.org.apache.commons.logging
Specification-Version: 1.0
Extension-Name: org.apache.commons.logging
Archiver-Version: Plexus Archiver
X-Compile-Target-JDK: 1.2
```

Fig. 71. Ejemplo de fichero MANIFEST.MF adaptado a un contenedor OSGi. [47]

Kernel

Ahora que ya tenemos una idea del funcionamiento de un contendor OSGi, podemos pasar a explicar el Kernel del ServiceMix. El Kernel, representado en la Figura 72, también llamado Apache Karaf, es una implementación básica de un contenedor OSGi al que se le han añadido ciertas funcionalidades para hacerlo más versátil y que permitan gestionar con más

facilidad los *bundles* instalados. El Kernel, está en la base de la arquitectura ServiceMix, y constituye el núcleo de este ESB.

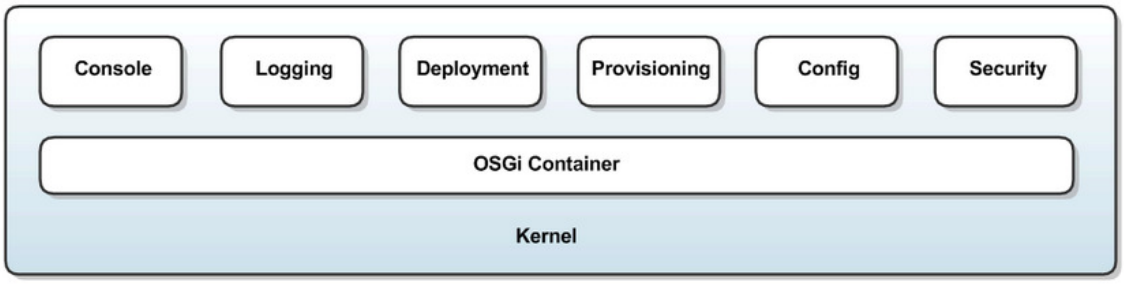

Fig. 72. El Kernel o núcleo del ServiceMix. [48]

Las funcionalidades asociadas al Kernel extienden el contenedor OSGi y lo dotan de ciertas características ya identificadas anteriormente en el estudio de los ESB. Por ejemplo, la trazabilidad (Logging), mecanismos de seguirdad (Security) y monitorización para proveer un entorno de control centralizado (Console). Todas estas características, al estar a nivel de Kernel, se centran en la parte más arquitectónica del ESB, es decir, para proveer una infraestructura adecuada para los servicios. Veamos a continuación estas utilidades:

Hot deployment

Despliegue en caliente de los *bundles* OSGi. Se podrán administrar, iniciar, parar, actualizar y borrar. Por despliegue en caliente se entiende que no hace falta parar el servidor para realizar las modificaciones.

Dynamic configuration

Los *bundles* existentes en el ServiceMix, suelen tener parámetros para sus configuraciones en ficheros de propiedades. Un fichero de propiedades consta de una lista de duplas nombre=valor, un ejemplo podría ser una propiedad que diga si se quiere o no tener activados los *logs* en un servicio. Esta propiedad estaría determinada por la siguiente propiedad: “logging=true”, con esto, estaremos diciendo que que queremos que la propiedad de *logging* esté activada. Dentro de ServiceMix, se habilita un directorio de ficheros de propiedades que se monitorizan, y cuando se detecta que ha habido un cambio en un fichero, se propaga esta propiedad al servicio. El directorio donde se depositan estos ficheros está en la carpeta <HOME>/etc. Donde <HOME> es el directorio principal de la instalación del servidor.

Logging System

Diferentes API (*Application Programming Interface*) de logging, por ejemplo, Log4J, JCL, o SLF4J. Una API, o como se dice en castellano, interfaz de programación de aplicaciones, es una agrupación de funcionalidades para ser utilizadas por las aplicaciones que las necesiten. En este caso, cuando una aplicación quiere poner trazas en su código para ver la evolución de la información tratada, no hace falta que implemente de cero toda la funcionalidad de *logging*, sino que ya exiten librerías que lo hacen, solamente hay que incorporarlas para utilizarlas.

Provisioning

Se podrán instalar los *bundles* desde diferentes vías: dejando los ficheros de instalación en una carpeta determinada para su despliegue (*deploy*), o mediante medios externos, por ejemplo, a través de una URL o incluso también con la herramienta Maven. Maven es un *framework,* o entorno de trabajo, de construcción de aplicaciones Java. En este estudio, cuando realicemos nuestro ejemplo de uso del ServiceMix, utilizaremos Maven y por lo tanto veremos con más detalle su funcionamiento.

Native OS integration

Integración en SO (Sistema Operativo) como servicio. Esto es útil por si se quiere realizar una instalación del ESB que tenga un funcionamiento continuo en un servidor, de esta forma, se puede crear por ejemplo un servicio Windows para no tener que arrancar a mano el ESB cada vez que se reinicia el servidor.

Extensible Shell console y Remote Access

Consola para gestionar *bundles*, ver los *logs*, instalar *bundles*, reiniciar el servidor, etc…

Security framework JAAS

JAAS (*Java Authentication and Authorization Service*) es una API que permite a las aplicaciones Java accceder a servicios de control de autenticación y acceso.

Managing instances

Gestión de diferentes instancias de Karaf a través de la consola.

Una vez visto el Kernel, pasaremos a detallar las prestaciones tecnológicas incorporadas en el ServiceMix. Las *features* constituyen una capa de más alto nivel y ésta se sitúa por encima del Kernel, es decir, la capa de prestaciones tecnológicas se provee de la infraestructura creada por el Kernel.

11.1.2 Prestaciones tecnológicas

En el contexto del ServiceMix, se han desarrollado un conjunto de *bundles* (Figura 73) especialmente pensados para funcionar sobre el Kernel. En esta capa también encontramos ciertas características detalladas en el estudio del ESB. Entre ellas, encontramos por ejemplo, BPEL que cubre la funcionalidad de orquestación de servicios, JBI para la integración de componentes creados con diferentes tecnologías, JMS para permitir comunicaciones síncronas o asíncronas mediante el uso de mensajes, Apache CXF que es un *framework* para la creación de servicios web compuesto por las API JAX-WS y JAX-RS, y para finalizar, tenemos NMR y Camel que cubren la necesidad de enrutamiento.

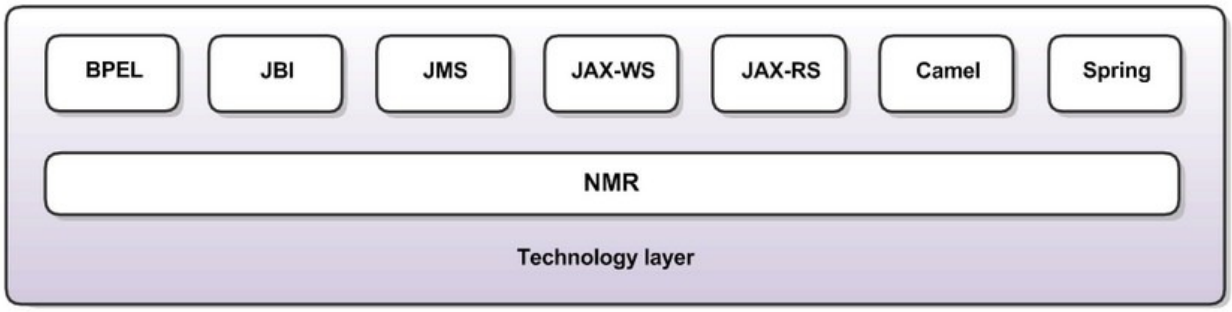

Fig. 73. Capa de características y tecnologías que operan en el ServiceMix. [48]

A continuación, veremos en detalle cada funcionalidad tecnológica provista dentro de esta capa.

BPEL (*Business Process Execution Language*)

Anteriormente ya se ha hablado sobre este estándar que proporciona un lenguaje para la orquestación de servicios. Esta funcionalidad también es presente en ServiceMix.

JBI (*Java Business Integration*)

JBI es una especificación Java que proporciona una arquitectura donde cada componente (*plug-in*) se conectan a contenedores JBI y proveen o consumen servicios. JBI está orientado al uso de servicios web pero no son estrictamente necesarios para su funcionamiento.

Hay dos tipos de componentes:

- ***Service Engines* (SE):** Motores de servicio. Dónde se encuentra la lógica de negocio. Pueden consumir otros servicios, consumir servicios web, realizar transformaciones de datos, etc…
- ***Binding components* (BC):** Son los componentes de vinculación. Sirven para enviar y recibir mensajes. Cuando reciben mensajes del exterior los normalizan y lo envían al

NMR (*Normalized Message Router*). Y para exportar mensajes al exterior hacen el proceso inverso, reciben mensajes normalizados del NMR y los desnormalizan adaptando el mensaje al consumidor. Por normalizar se entiende el uso exclusivo del estándar XML.

JBI pretende ser un estándar a nivel de ESB, es decir, que todos los ESB usen JBI de tal manera que cualquier componente desplegado en un tipo de ESB también sirva para otro.

Junto con el *bundle* JBI provisto en el el ServiceMix, también se proporciona la herramienta JAVA JMX (*Java Management eXtensions*). JMX proporciona una plataforma para gestionar el entorno JBI, como puede ser la instalación, despliegue, monitorización y control de ciclo de vida de los componentes JBI.

Apache ActiveMQ (JMS)

Apache ActiveMQ es una implementación del estándar Java JMS. JMS es un estándar Java de mensajería para el uso de colas de mensajes. Permite a los componentes de las aplicaciones crear, enviar, recibir y leer mensajes de forma síncrona o asíncrona. Aunque el uso de colas está más asociado al concepto de comunicación asíncrona, también es factible usarlo de forma síncrona [49]. Para comunicación síncrona, JMS ofrece el método *recieve*, donde el consumidor de los mensajes se conecta a la cola y se espera indefinidamente hasta que llegue un mensaje, también existe la posibilidad de determinar un tiempo de espera, por ejemplo, *recieve*(5000) esperaría 5000 ms (5 segundos) la recepción de un mensaje, una vez transcurrido este tiempo si no se ha recibido ningún mensaje, el programa sigue su flujo de ejecución. Para comunicaciones asíncronas, existe el método *onMessage*, este método configurado en el consumidor, se ejecuta solamente cuando se detecta la llegada de un mensaje.

Apache CXF (JAX-WS y JAX-RS)

Apache CXF es una implementación de las especificaciones Java JAX-WS y JAX-RS. JAX-WS es la especificación Java para la creación de servicios web y JAX-RS es también una especificación Java para servicios web pero en este caso del tipo REST, término ya introducido cuando hemos hablado de los inconvenientes de los servicios web. REST pues, son mensajes que utilizan XML y HTTP parecidos a los mensajes SOAP pero son más sencillos de construir y más ligeros, ya que no poseen las abstracciones adicionales de los mensajes SOAP que hacen de él un estándar complejo y pesado. Para observar la diferencia y teniendo en cuenta lo ya visto anteriormente sobre el estándar SOAP, mostramos un ejemplo de REST:

```
<?xml version="1.0" encoding="UTF-8"?>
<messages>
  <message>
```

```
    <date_created>Mon May 15 02:45:00 +0000 2008</date_created>
    <user>webmonkey</user>
    <text>Eating a banana and writing some code</text>
  </message>
  <message>
    <date_created>Mon May 15 02:44:00 +0000 2008</date_created>
    <user>anotheruser</user>
    <text>I think it might be time to go to bed</text>
  </message>
  <message>
   <date_created>Mon May 15 02:43:00 +0000 2008</date_created>
   <user>someone</user>
   <text>What the heck is REST?</text>
 </message>
</messages>
```

Como se observa, un mensaje REST es un XML mucho más sencillo, no contiene definiciones de tipos de datos, ni cabeceras ni descripciones concretas, conteniendo así menos texto y haciéndolo más ligero, pero por el contrario, como ya se ha estudiado anteriormente, esta falta de información adicional hacen que sean menos configurables y menos aptos para entornos empresariales.

Apache Camel

Apache Camel es un motor de ruteo y mediación basado en los patrones EIP (*Enterprise Integration Patterns*) [50]. Dentro de ServiceMix, es una alternativa al uso del NMR ya que ambos elementos se utilizan para el enrutamiento, con la diferencia que Camel puede trabajar con mensajes de cualquier tipo, en cambio, los mensajes en NMR deben ser del tipo XML.

Spring

Spring es un *framework* de desarrollo de aplicaciones para la plataforma Java. Gracias a Spring se pueden construir aplicaciones de alto rendimiento, fácilmente testeables y con un elemento muy importante, un alto grado de reutilización de código. Spring permite mediante la configuración en ficheros XML la creación de objetos Java y éstos luego se pueden recuperar desde cualquier clase Java. Por lo tanto, no hace falta que todas las clases u otros objetos Java que usen ese objeto lo instancien en memoria, simplemente lo recuperan de Spring y todos usan el mismo, estando el objeto instanciado un memoria una sola vez. A este proceso de habilitar un mismo objeto a diferentes clases/objetos se llama *Injection* (Inyección).

Por lo tanto, en ServiceMix habrán desplegados los *bundles* de Spring que ayudarán en la implementación de nuestras aplicaciones BC (*Binding Components*) y SE (*Service Engines*).

11.1.3 ServiceMix NMR (Normalized Message Routing)

Como ya se ha introducido, es el bus por donde pasan los mensajes en la comunicación entre componentes, forma parte de la especificación JBI. El NMR (Figura 74) no es más que un conjunto de *bundles* desplegados sobre un contenedor OSGi, especialmente sobre el propio ServiceMix Kernel (Apache Karaf).

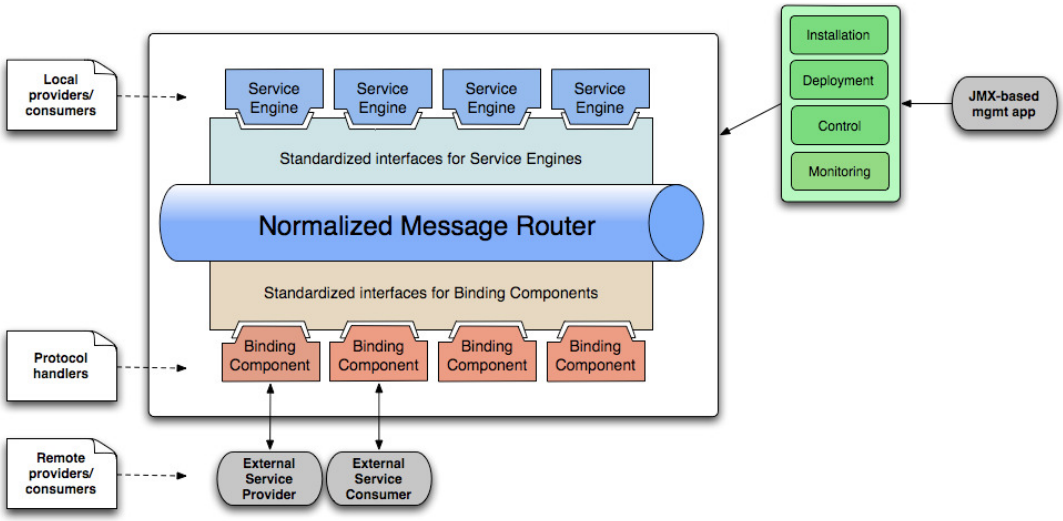

Fig. 74. El bus del ServiceMix (NMR). [51]

NMR se encarga de la recepción, transformación y ruteo de mensajes entre los diferentes *bundles*. Cada BC procesa mensajes input, NMR los enruta, procesa y los encamina a los SE.

Por ejemplo, en un posible servicio que sirva para dar de alta clientes, podríamos tener diferentes BC en función del protocolo a utilizar hacia el exterior (altaClienteSOAP, altaClienteFileXML, altaClienteJMS) y todos estos BC se enrutarían al mismo SE interno (altaCliente).

Una vez visto a nivel teórico el funcionamiento y las características ofrecidas por el ESB ServiceMix, ya estamos en disposición de realizar un ejemplo práctico sobre él. Antes de dar paso a la construcción de nuestra aplicación de ejemplo, cerraremos la parte teórica de este estudio dedicado a SOA y los servicios, con una breve introducción a la computación en la nube o *Cloud Computing*. Incluimos esta parte en nuestro estudio porque si bien el *Cloud* está basado en el concepto de servicios, no se puede clasificar como SOA. Ambos conceptos son diferentes tal y como veremos en el siguiente capítulo.

12. CLOUD COMPUTING

Cloud Computing o computación en la nube es un paradigma que permite ofrecer servicios a través de Internet. En general, se pretende ofrecer como servicio todo lo que pueda ofrecer un sistema informático, redes, servidores, almacenamiento, aplicaciones, servicios). Según el IEEE *Computer Society* [52] es un paradigma en que la información se almacena de manera permanente en servidores de Internet y ésta es accesible por cualquier dispositivo como pueden ser los equipos de escritorio convencionales (*Desktops*) o portátiles (*Laptops*), dispositivos móviles, etc. Representamos en la Figura 75 el *Cloud Computing* dónde la nube es el elemento donde residen todos nuestros datos y aplicaciones.

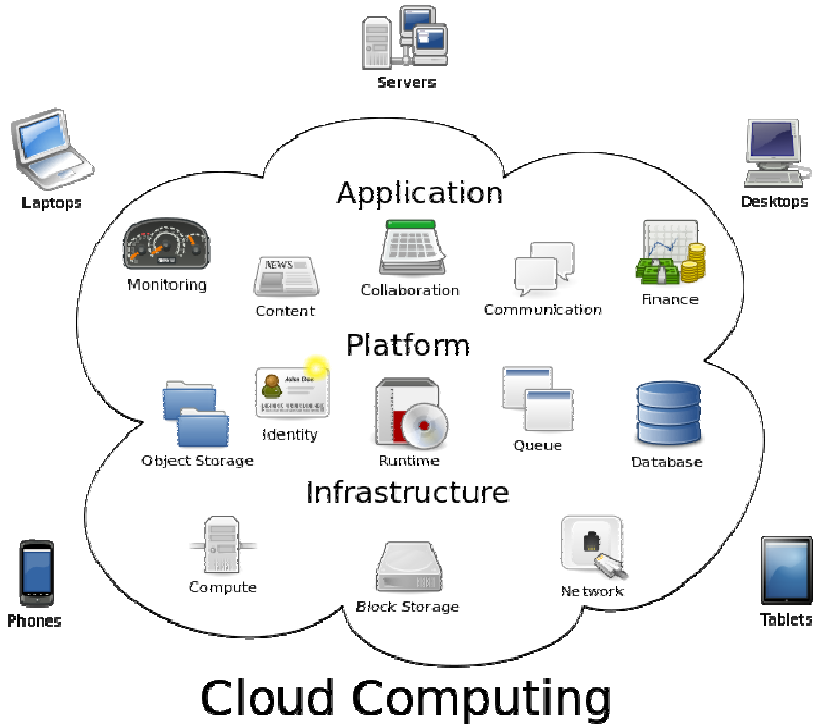

Fig. 75. Cloud Computing. [53]

Estos servicios alojados en la nube permiten que un cliente pueda consumirlos y estar operativos rápidamente con el mínimo esfuerzo de gestión o interacción con el proveedor de estos servicios. Hay tres modelos de servicio en la nube (Figura 76), los cuales describiremos brevemente y posteriormente desarrollaremos por separado.

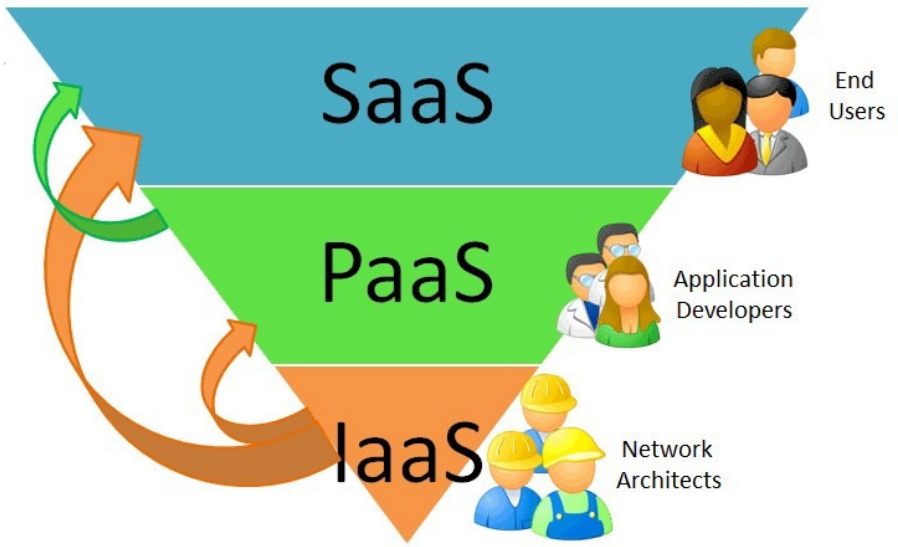

Fig. 76. Modelos de servicio en la nube. [54]

- **SaaS (*Software as a Service*) :** En castellano, software como servicio. Este modelo de servicio implica la creación de aplicaciones cuyos usuarios son los usuarios finales (*end users*), es decir, aquellos usuarios que interaccionan la aplicación en última instancia, por ello están en el nivel más superior de la Figura 76.

- **PaaS (*Platform as a Service)*:** En castellano, plataforma como servicio. Este modelo de uso de la nube proporciona una plataforma de programación sin necesidad de un *hardware* ni *software* interno en los ordenadores personales. PaaS está destinada a los desarrolladores de aplicaciones (*application developers*), cuyas aplicaciones construidas serán usadas como SaaS por los usuarios finales.

- **IaaS (*Infraestructure as a Service*):** En castellano, infraestructura como servicio. IaaS proporciona los servicios de almacenamiento, computación y ancho de banda según demanda. Este modelo será usado tanto por desarrolladores de aplicaciones como por usuarios finales.

A continuación detallaremos cada modelo.

12.1 *SAAS: SOFTWARE COMO SERVICIO*

La capacidad proporcionada al consumidor es utilizar las aplicaciones del proveedor, que se ejecutan en una infraestructura en la nube. Se puede acceder a las aplicaciones desde diversos dispositivos del cliente, a través de una interfaz de cliente simple (*thin client*), como un navegador web. El consumidor no gestiona ni controla la infraestructura subyacente en la nube, que incluye la red, los servidores, los sistemas operativos, el almacenamiento, ni tampoco las

funcionalidades de aplicación individuales, con la posible excepción de configuraciones de la aplicación limitadas específicamente al usuario.

Como ejemplos encontramos: SalesForce, Siebel on demand, MS Office, Web NotSuite y Google Apps [55], que proporciona varios servicios bien conocidos, como Gmail para el correo electrónico, Google Calendar para calendarios o Google Drive para almacenar documentos.

12.2 *PAAS: PLATAFORMA COMO SERVICIO*

En este caso se pretende desproveer al consumidor de su infraestructura de desarrollo y trasladarla a la nube, ésta puede ser creada por el consumidor, o las aplicaciones adquiridas, creadas utilizando lenguajes de programación y herramientas de apoyo del proveedor. Se crea un entorno que contiene los componentes y APIs necesarios para operar sobre una tecnología concreta. PaaS puede ofrecer servicios para todas las fases del ciclo de desarrollo de software y sus pruebas asociadas o puede ofrecer servicios sobre un área en concreto, como puede ser la gestión documental.

Como ejemplos encontramos: Engine, Force.com, MS Azure, Yahoo! Pipes y Google APP [56]. Google App proporciona una plataforma para desarrolladores de aplicaciones que da soporte a múltiples lenguajes de programación, entre ellos, Java, Python o PHP. También provee varios *frameworks* que ayuden a los desarrollos, como por ejemplo, el *framework* para Java Spring que como hemos visto, también está presente en el ESB ServiceMix.

12.3 *IAAS: INFRAESTRUCURA COMO SERVICIO*

La infraestructura como servicio o también *Hardware* como servicio permite subministrar al cliente la capacidad de procesamiento, almacenamiento, redes y otros recursos informáticos fundamentales en los que el cliente puede implantar y ejecutar cualquier software, que incluye sistemas operativos y aplicaciones. El cliente no gestiona ni controla la infraestructura, pero controla los sistemas operativos, el almacenamiento, las aplicaciones implantadas y posiblemente tenga un control limitado de algunos componentes de la red, como los *firewalls*.

Entre las empresas que ofrecen servicios IaaS encontramos Amazon, IBM, HP, RackSpace o Telefónica. Si tomamos como ejemplo Amazon, vemos que esta empresa ofrece entre sus múltiples servicios el Amazon AWS (*Amazon Web Services*) [57] y dentro de éste, por ejemplo, existe el servicio Amazon RDS (Amazon *Relational Database Service*). Este servicio permite la configuración y gestión de bases de datos relacionales de los tipos MySQL, Oracle y SQL Server. De esta forma, si una empresa quiere implantar un sistema de bases de datos no es necesario que invierta en la infraestructura de servidores y en la instalación de programas físicamente en la misma empresa, todo esto ya está en la nube. Simplemente deberá usar el servicio y ver cómo puede integrarlo con las aplicaciones que requieran de su uso.

12.4 *BENEFICIOS DEL CLOUD COMPUTING*

Los beneficios que aportan la computación en la nube son varios, por ejemplo, la prestación de servicios a nivel mundial: las infraestructuras de la nube tienen mayor capacidad de adaptación, recuperación de datos y reducción al mínimo de tiempos de inactividad. La computación en la nube también permite al cliente prescindir de la infraestructura *hardware* con lo que si se inicia un proyecto empresarial nuevo permite reducir la inversión inicial. De la misma forma pasa con el *software*, éste está disponible de forma más rápida y sin requerir una gran inversión. Otro beneficio aportado es que la nube ofrece actualizaciones automáticas, sin que el usuario deba intervenir en ellas. Por último, gracias al *Cloud,* se promueve el uso eficiente de la energía, en los centros de datos tradicionales los servidores consumen más energía de la necesaria, en cambio, en la nube, la energía consumida es la necesaria reduciendo el desperdicio.

12.5 *INCONVENIENTES DEL CLOUD COMPUTING*

Podemos citar varios inconvenientes del *cloud computing*. Por ejemplo, la centralización de las aplicaciones y el almacenamiento de los datos produce una dependencia de los proveedores de los servicios y la disponibilidad de las aplicaciones está ligada a la disponibilidad de acceso a Internet. Otro inconveniente importante es que los datos de negocio de la organización no están físicamente en la empresa con lo que se pierde el control referido a temas de seguridad. Por último, cabe destacar la escalabilidad a largo plazo. Un incremento del uso de la infraestructura de la nube puede provocar reducciones en la calidad del servicio si los proveedores de los servicios no poseen la capacidad de computación necesaria para soportar dicha demanda.

12.6 *CLOUD VERSUS SOA*

Cloud y SOA tiene puntos en común como el hecho de estar basados en servicios y que éstos son independientes de su localización. Ahora bien, no podemos afirmar que tanto la computación *Cloud* como SOA sean lo mismo o que *Cloud* es una forma de SOA. Es posible que en los proveedores de servicios en Cloud operen internamente como una implementación del paradigma SOA, pero esto no se puede saber sin conocer este funcionamiento. Otro punto en que difieren es el ámbito de actuación. SOA tiene un enfoque empresarial y *Cloud* se refiere a cómo se ofrecen recursos al cliente final, su entrega, almacenamiento y gestión.

Por lo tanto, se puede decir que tanto *Cloud* como SOA pueden ser complementarios pero a nivel conceptual son cosas diferentes. Por ejemplo, un sistema empresarial que opera bajo SOA, se podría trasladar a la nube para ofrecer las ventajas que ésta ofrece: agilidad en la provisión de la misma, escalabilidad, prescindir de la infraestructura hardware, gestión y monitorización, etc…

Aquí concluye la parte teórica de nuestro estudio. En el siguiente capítulo realizaremos un ejemplo práctico de uso del Fuse ESB, mediante la creación de una aplicación basada en servicios web.

13. IMPLEMENTACIÓN DE UNA APLICACIÓN EN FUSE ESB - SERVICEMIX

En este apartado desarrollaremos una aplicación de ejemplo en un ESB Fuse ServiceMix versión 4.3.1. En un primer paso detallaremos los requerimientos de instalación y el proceso de instalación del Fuse. Después, cuando tengamos el ESB operativo, desarrollaremos en él una aplicación.

13.1 *REQUERIMIENTOS PARA LA INSTALACIÓN DEL FUSE ESB Y DEL ENTORNO DE NUESTRA APLICACIÓN*

Como ya hemos visto en nuestro estudio anteriormente, Fuse ESB se trata de una versión *opensource* y entre otras cosas, se diferenciaba de los ESB propietarios en que eran más ligeros. Con lo cual, no serán necesarios grandes requerimientos como veremos a continuación. A continuación citaremos los requerimientos necesarios para llevar a cabo la instalación, teniendo en cuenta que nos centramos en el sistema operativo Windows.

Sistema operativo

Windows - Windows 2000, XP, 2003, Vista, 2008 y Windows 7. 32-bit y 64-bit. La instalación detallada en este documento se realizará sobre Windows 7.

Versión de Java

Necesitamos un JDK (*Java Development Kit*) para poder compilar nuestras aplicaciones Java. Necesitamos una versión 1.6.0_18 o superior. Usaremos la versión jdk-6u43-windows-i586.

Versión de Fuse ESB.

Usaremos la versión apache-servicemix-4.3.1-fuse-01-15.

Versión de Apache Maven

Utilizaremos la herramienta Maven para compilar nuestras aplicaciones. En concreto, la versión Apache Maven 2.2.1.

Versión de Eclipse

Para el desarrollo de este proyecto usaremos el IDE (*Integrated Development Enviroment*) *opensource* Eclipse JEE Helios, Esta programa nos permitirá editar nuestras aplicaciones. La versión que utilizaremos será eclipse-jee-helios-SR2-win32.

Versión SOAP UI

Utilizaremos este programa para testear nuestra aplicación y ver que funciona correctamente. SOAP UI permite lanzar peticiones SOAP a un servicio web y posteriormente ver la respuesta. Utilizaremos el programa SOAP UI 4.5.1.

Espacio requerido en el disco duro

A continuación mostramos en una tabla los requerimientos de espacio a partir de las aplicaciones listadas anteriormente. En el caso de Maven, también se hace una estimación de lo que ocuparán las librerías que se instalarán automáticamente cuando se intente compilar las aplicaciones.

Componente	Espacio Requerido (aprox.)
JDK Java	300 MB
Fuse ESB	250 MB
Apache Maven	1.5 GB
Eclipse Helios	300 MB
SOAP UI	200 MB
TOTAL	2.6 GB

Tabla 4. Espacio requerido en el disco duro para nuestra aplicación.

Se estima que como mínimo debe haber 2.6 GB de espacio para nuestro ejemplo. A partir de aquí, si se crean más aplicaciones se requerirá más espacio, tanto para las aplicaciones en sí como por las librerías que se descarguen para que sea necesario la ejecución de dichas aplicaciones.

Espacio de memoria RAM

Para una ejecución correcta debe haber un mínimo de 2 GB de RAM. Al estar en un entorno Windows, sería recomendable entre 2 y 4 GB.

Internet

Se debe estar conectado a Internet para poder descargar todas las aplicaciones que hacen falta.

13.2 *INSTALACIÓN DEL SERVIDOR FUSE ESB*

Los pasos para instalar el servidor son los siguientes. Nuestra unidad de trabajo será la unidad F:\.

1. Nos descargamos el fichero apache-servicemix-4.3.1-fuse-01-15.zip[1].

2. Descomprimimos en la unidad F:\ el fichero descargado tal y como se muestra en la Figura 77.

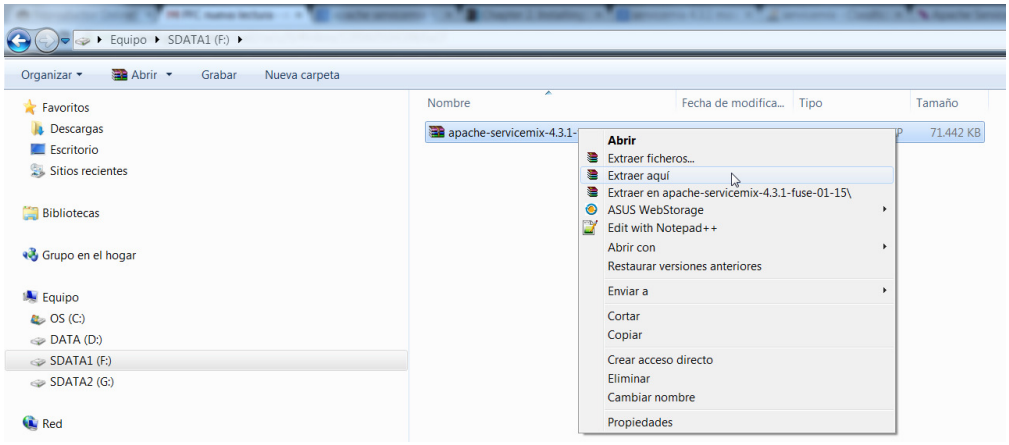

Fig. 77. Extracción del archivo de instalación del Fuse ESB.

El resultado de descompresión nos habrá creado una carpeta de nombre apache-servicemix-4.3.1-fuse-01-15 tal y como se muestra en la Figura 78.

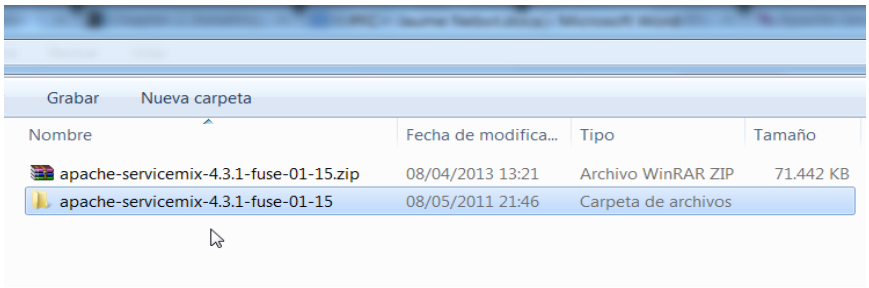

Fig. 78. Directorio del Fuse ESB instalado.

3. Descargamos la JDK de Java jdk-6u43-windows-i586.exe[2] . Una vez descargado, ejecutamos el instalador en modo administrador tal y como se observa en la Figura 79.

[1] http://repo.fusesource.com/nexus/content/groups/public/org/apache/servicemix/apache-servicemix/4.3.1-fuse-01-15/apache-servicemix-4.3.1-fuse-01-15.zip

[2] http://www.oracle.com/technetwork/java/javasebusiness/downloads/java-archive-downloads-javase6-419409.html#jdk-6u43-oth-JPR

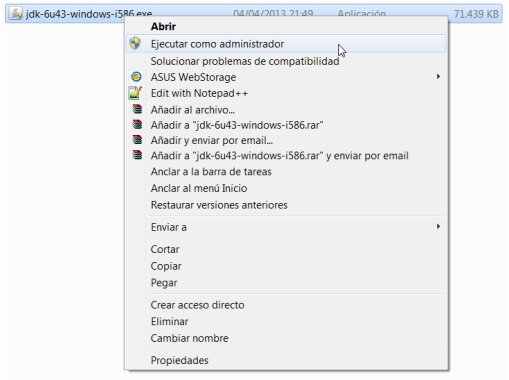

Fig. 79. Ejecución del fichero de instalación de la JDK.

Nos aparecerá la pantalla de inicio de instalación. Hacemos clic en el botón Next (Figura 80).

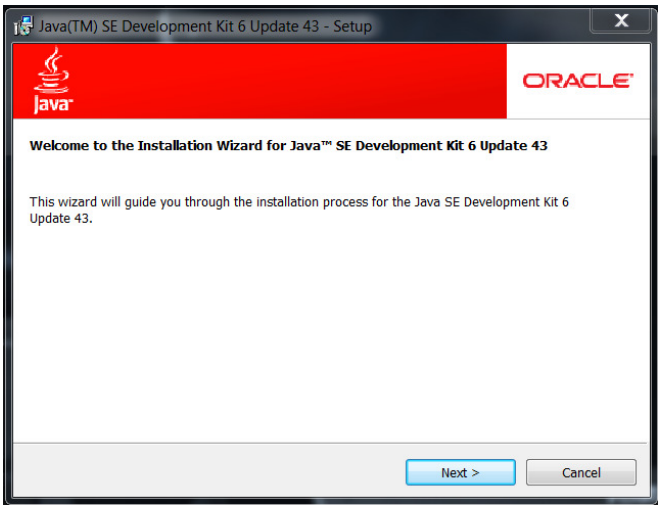

Fig. 80. Instalación de la JDK, paso 1.

En el siguiente paso (Figura 81) nos fijamos en el directorio por defecto donde se realizará la instalación (Install to), este directorio será nuestra variable del sistema <<RUTA RAÍZ JAVA>> que utilizaremos más adelante. En nuestro caso la ruta es C:\Program Files (x86)\Java\jdk1.6.0_43

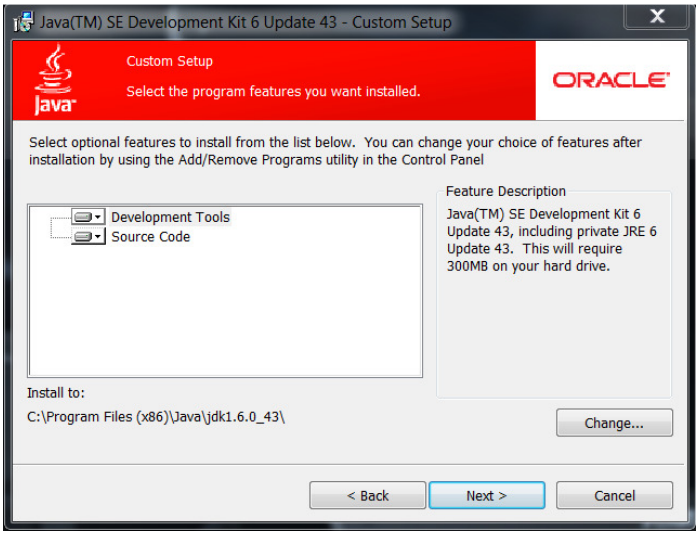

Fig. 81. Instalación de la JDK, paso 2.

Dejamos las opciones como están y hacemos clic en Next.

Se iniciará el proceso de instalación, y una vez finalizado nos saldrá la siguiente pantalla (Figura 82) informándonos que la instalación se ha completado satisfactoriamente. Hacemos clic en Finish.

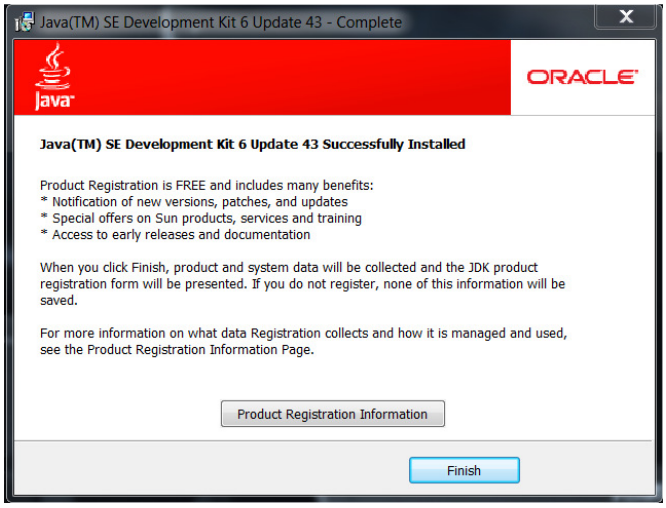

Fig. 82. Instalación de la JDK, paso 3.

4. Nos descargamos el *framework* Maven[3] que nos servirá para compilar la aplicación y ponerla en el repositorio Maven a disposición del ServiceMix.

   Lo instalamos descomprimiéndolo en F:\ (Figura 83).

[3] http://www.apache.org/dyn/closer.cgi/maven/maven-3/3.0.4/binaries/apache-maven-3.0.4-bin.zip

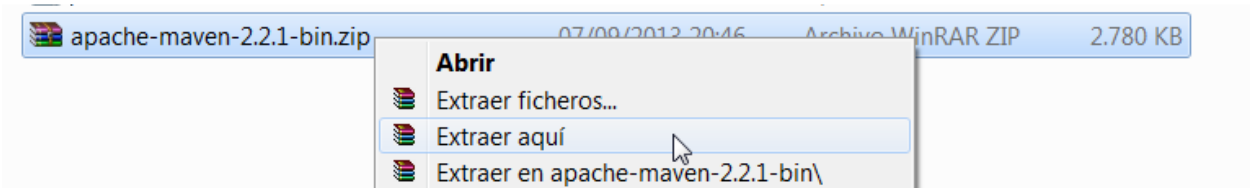

Fig. 83. Descompresión del fichero de instalación de Apache Maven.

El resultado será una carpeta de nombre apache-maven-2.2.1 (Figura 84).

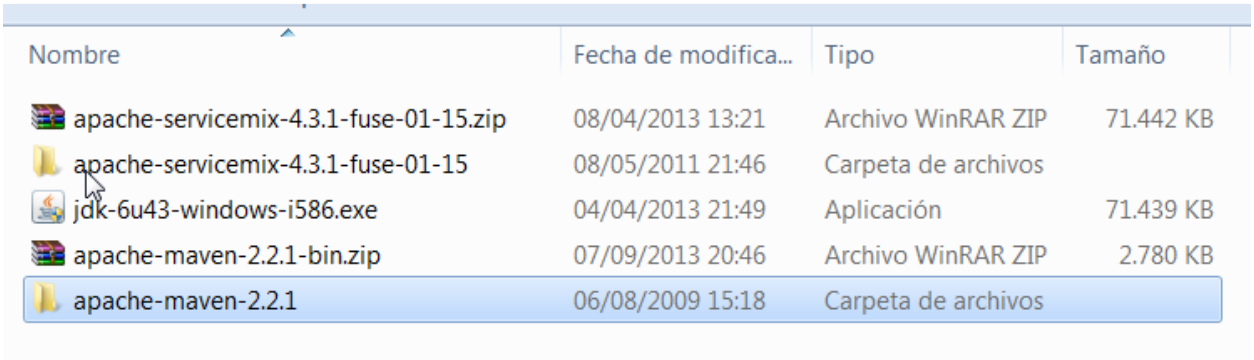

Fig. 84. Directorio del Apache Maven.

La ruta “F:\apache-maven-2.2.1” será nuestra variable M2_HOME.

5. Definimos les variables de entorno en Windows

Para ello nos digimos a la configuración del sistema haciendo clic en el botón Inicio de Windows, botón derecho sobre Equipo y seleccionamos Propiedades (Figura 85).

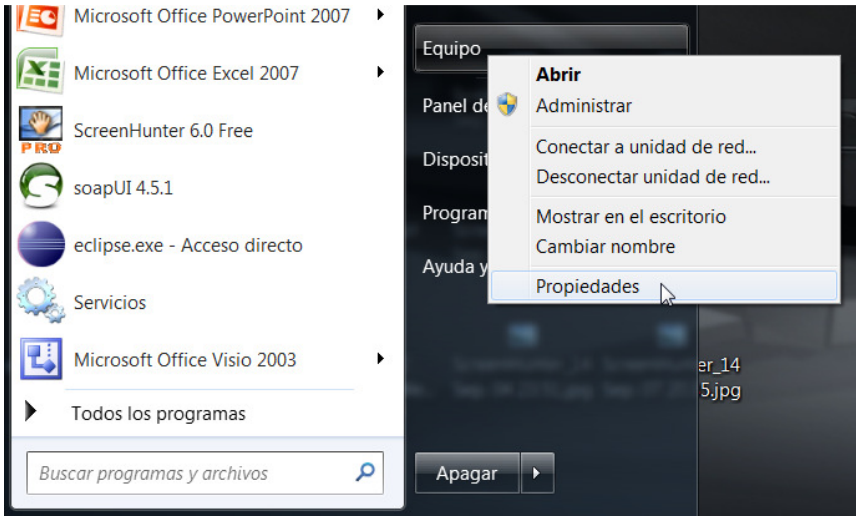

Fig. 85. Propiedades de Equipo.

Se nos abrirá una ventana y hacemos clic en configuración avanzada del sistema (Figura 86).

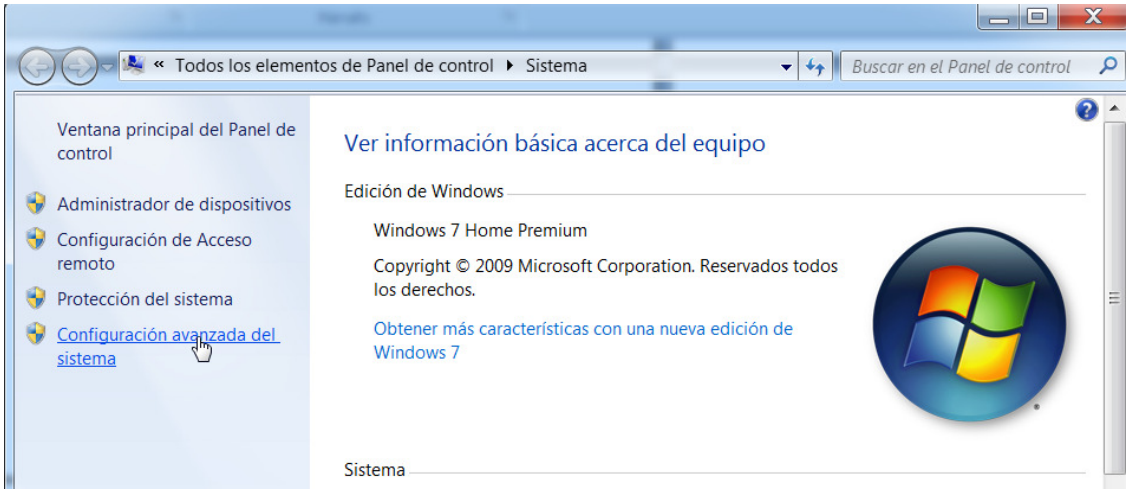

Fig. 86. Configuración avanzada del sistema.

Una vez dentro de la configuración avanzada del sistema, hacemos clic en variables de entorno (Figura 87).

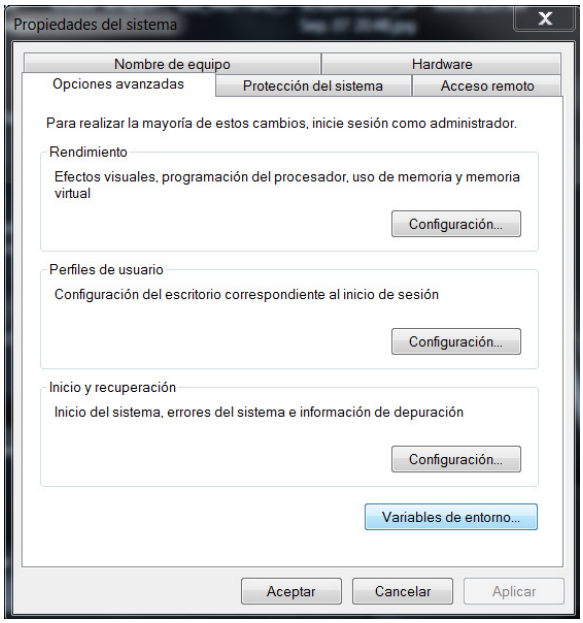

Fig. 87. Variables de entorno.

Ahora añadimos las variables haciendo clic en el botón Nueva… de la sección Variables del sistema (Figura 88).

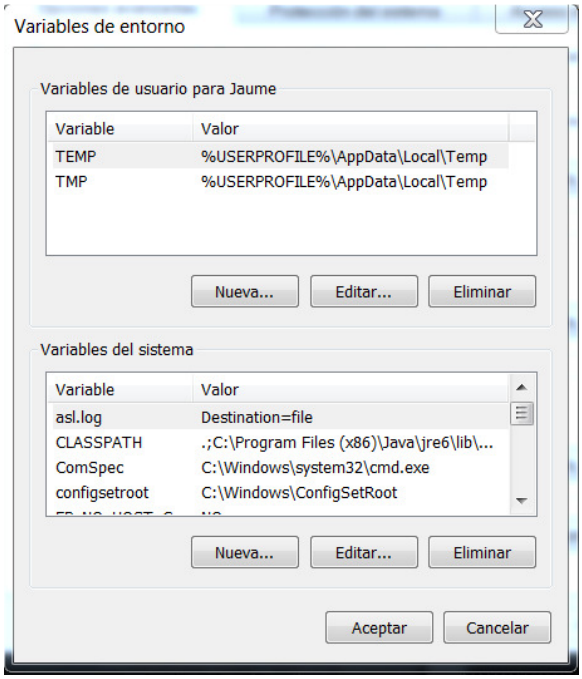

Fig. 88. Variables del sistema.

Se nos abrirá una ventana para incluir las variables (Figura 89),

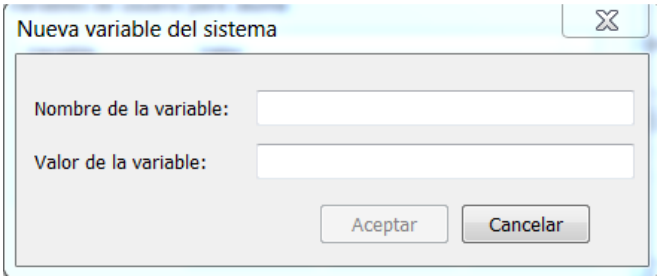

Fig. 89. Nueva variable del sistema.

Las variables a añadir en formato “Nombre de la variable = Valor de la variable” son las siguientes: M2_HOME para indicar el directorio raíz de Maven, JAVA_HOME para el directorio raíz de Java, JAVA_TOOL_OPTIONS para añadir la codificación por defecto UTF-8, KARAF_HOME para nuestro directorio raíz del Fuse, y por último modificaremos la variable path, para poner los directorios donde se encuentran los ejecutables tanto de Java, como de Maven.

A modo de ejemplo (Figura 90) se muestra como se debería incluir la primera variable.

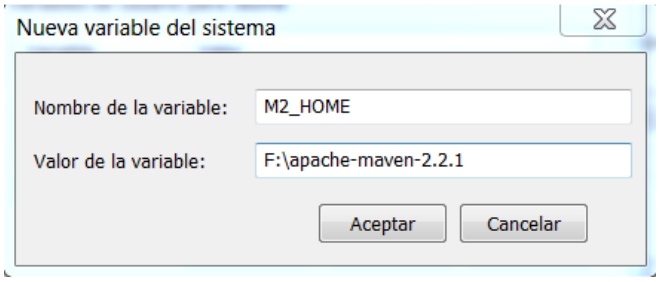

Fig. 90. Ejemplo de creación de una nueva variable del sistema.

Repetimos el proceso para las siguientes variables:

```
JAVA_HOME=<<RUTA RAIZ JAVA>>
JAVA_TOOL_OPTIONS =-Dfile.encoding=UTF-8
KARAF_HOME = F:\apache-servicemix-4.3.1-fuse-01-15
```

Para la variable path, editaremos la variable path ya existente y añadiremos al final del valor de la variable el siguiente texto:

```
;%JAVA_HOME%\bin;%M2_HOME%\bin
```

En el caso de que ya exista la variable %JAVA_HOME%\bin porque la instalación del Java ya la ha puesto, pondremos el siguiente texto:

```
;%M2_HOME%\bin
```

La sintaxis dentro del valor de la variable path del estilo %nombre_de_la_variable% indica que se hace referencia a una variable del sistema ya existente. El valor de la variable path debe quedar de la siguiente manera (Figura 91):

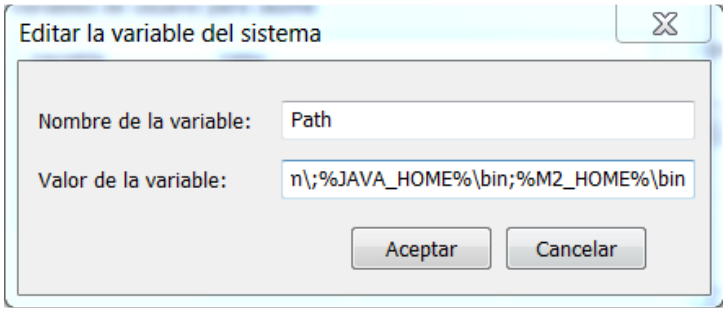

Fig. 91. Edición de la variable del sistema path.

6. Modificamos el fichero " F:\apache-maven-2.2.1\conf\settings.xml" para incluir un repositorio en Internet por defecto, en el que Maven irá a buscar las librerías. En la sección <mirrors></mirrors> añadimos el código marcado en rojo:

<mirrors>
<!-- mirror
| Specifies a repository mirror site to use instead of a given repository. The repository that

```
 | this mirror serves has an ID that matches the mirrorOf element of this mirror. IDs are
used
| for inheritance and direct lookup purposes, and must be unique across the set of
mirrors.
  |
<mirror>
        <id>mirrorId</id>
        <mirrorOf>repositoryId</mirrorOf>
        <name>Human Readable Name for this Mirror.</name>
        <url>http://my.repository.com/repo/path</url>
</mirror>
 -->
         <mirror>
                <id>Central</id>
                <url>http://repo1.maven.org/maven2</url>
                <mirrorOf>central</mirrorOf>
        </mirror>
 </mirrors>
```

También realizaremos la siguiente modificación que consta de los siguientes dos pasos:

a. Creación del directorio “F:\repository”

b. Mofidificación del fichero “F:\apache-maven-2.2.1\conf\settings.xml” donde cambiamos el siguiente código:

```
<!-- localRepository
  | The path to the local repository maven will use to store artifacts.
  |
  | Default: ~/.m2/repository
 <localRepository>/path/to/local/repo</localRepository>
 -->
```

Por el siguiente código:

```
<!-- localRepository
  | The path to the local repository maven will use to store artifacts.
  |
  | Default: ~/.m2/repository
  -->
 <localRepository>F:\repository</localRepository>
```

Con esta modificación estamos indicando que queremos especificar el directorio donde queremos situar el repositorio donde Maven se descargará las librerías necesarias.

Realizamos esta modificación porque se ha detectado en instalaciones sobre Windows XP, que el directorio del repositorio por defecto se sitúa bajo el directorio “C:\Documents

And Setttings\....” y los espacios presentes en los nombres de los directorios son fuente de errores. De esta forma, evitaremos posibles problemas a raíz de esta casuística.

7. Ejecutamos la consola, en el Botón Inicio buscamos la palabra “cmd” y nos aparecerá la consola. Hacemos clic en ella (Figura 92).

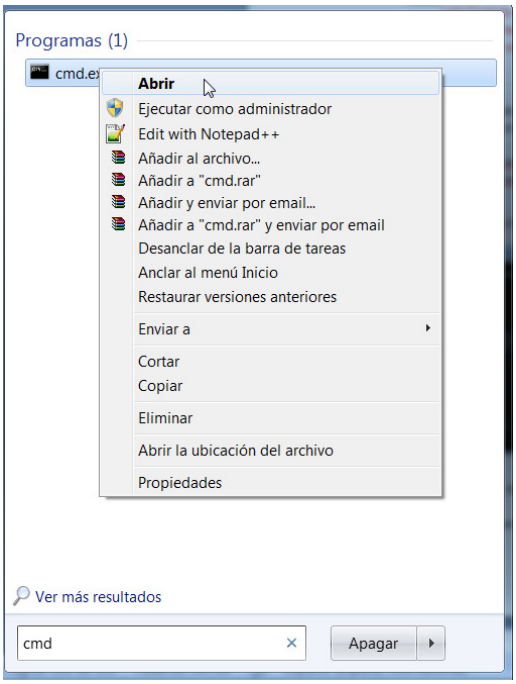

Fig. 92. Ejecución de la consola de Windows o línea de comandos.

Una vez abierta, ejecutamos la instrucción “F:\apache-servicemix-4.3.1-fuse-01-15\bin\karaf.bat” para arrancar el ServiceMix, tal y como muestra en la Figura 93.

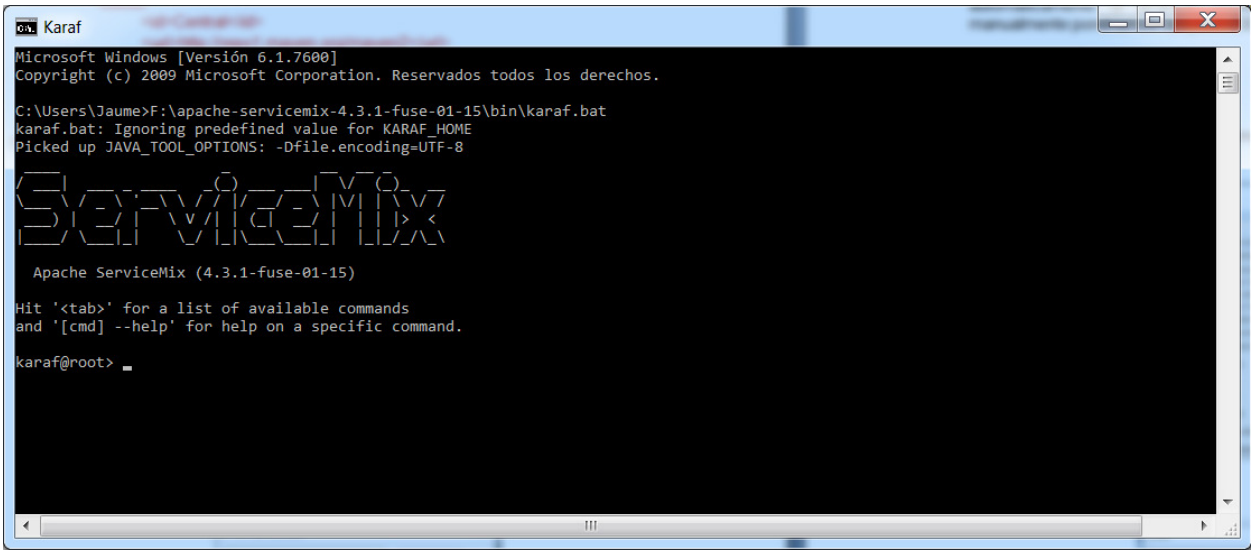

Fig. 93. Consola del ServiceMix.

Si aparece la línea karaf@root> es que se ha arrancado correctamente el ServiceMix, tal y como se aprecia en la Figura 93.

8. Instalamos ServiceMix como servicio de Windows.

- Ejecutamos desde la consola karaf (Figura 94) la siguiente instrucción para instalar el *bundle* wrapper que nos permitirá crear los archivos necesarios para instalar el ServiceMix como servicio:

```
karaf@root> features:install wrapper
```

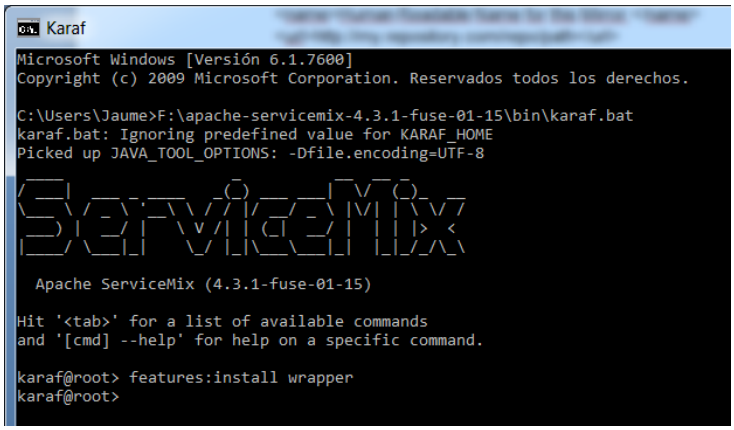

Fig. 94. Instalación del *bundle* wrapper.

```
karaf@root> wrapper:install -s AUTO_START -n SERVICEMIX -d SERVICEMIX
-D "ServiceMix Service"
```

Con el parámetro AUTO_START indicamos que el servicio se iniciará automáticamente al arrancar Windows. Si quisiéramos arrancar el servicio manualmente pondríamos la opción DEMAND_START (Figura 95).

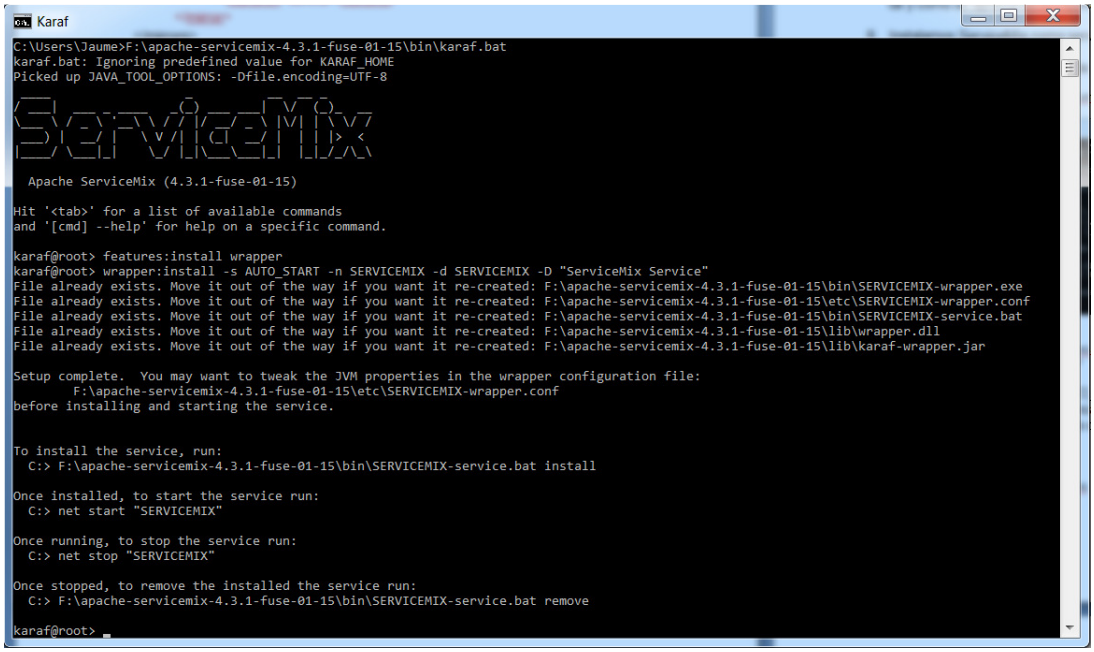

Fig. 95. Instalación de los archivos que sirven para instalar ServiceMix como servicio de Windows.

Los primeros mensajes de *File already exists* son normales porque estos archivos ya estaban creados. Se ha realizado de nuevo la instalación a modo de ejemplo para mostrar el mensaje de ejecución completada.

- Paramos Karaf tal y como se muestra en la Figura 96:

```
karaf@root> osgi:shutdown
```

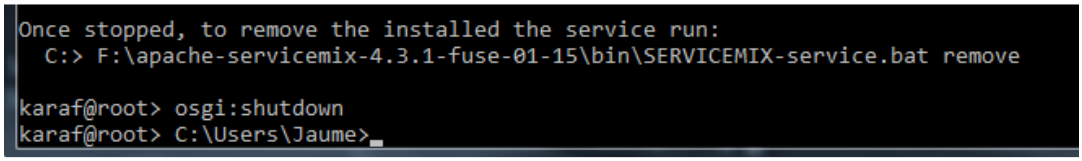

Fig. 96. Comando para salir de la consola ServiceMix.

- Cerramos la consola actual y abrimos otra en modo administrador tal y como muestra la Figura 97. Se ejecuta en modo administrador para poder tener los permisos de usuario que permitan instalar servicios Windows.

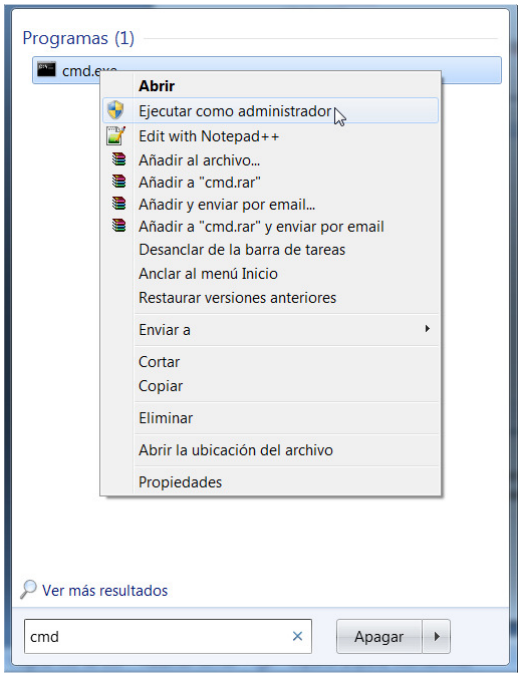

Fig. 97. Consola de Windows ejecutada como usuario administrador.

Luego, ejecutamos la siguiente instrucción para instalar el ServiceMix como servicio de Windows: "F:\apache-servicemix-4.3.1-fuse-01-15\bin\SERVICEMIX-service.bat install". (Figura 98).

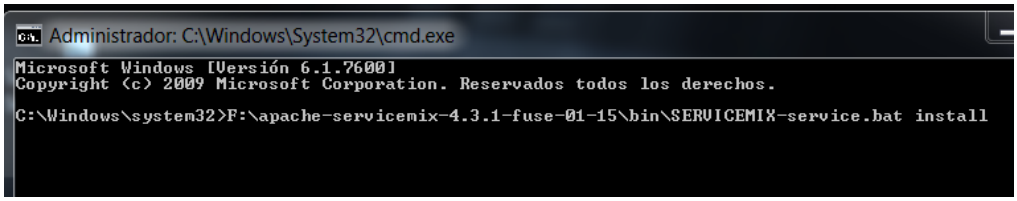

Fig. 98. Instalación de ServiceMix como servicio de Windows.

Si se ha instalado correctamente, aparecerá el mensaje que se muestra en la Figura 99.

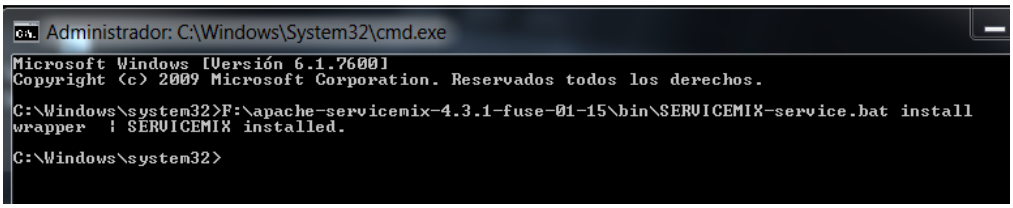

Fig. 99. Resultado satisfactorio del servicio de Windows de ServiceMix.

- El último paso es comprobar que se inicia y se para correctamente el servicio. el Desde la línea de comandos ejecutamos: net start SERVICEMIX (Figura 100).

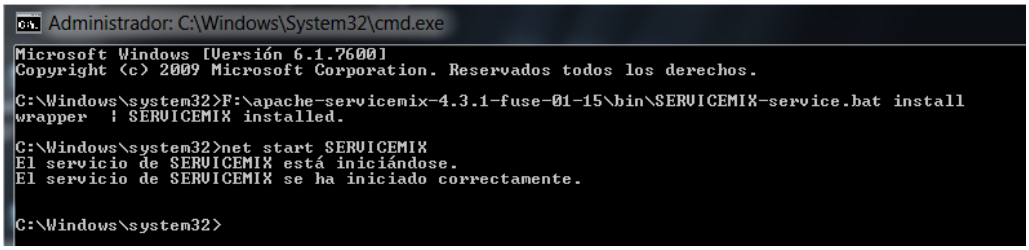

Fig. 100. Inicio del servicio de Windows de ServiceMix.

Y a continuación ejecutamos: net stop SERVICEMIX (Figura 101).

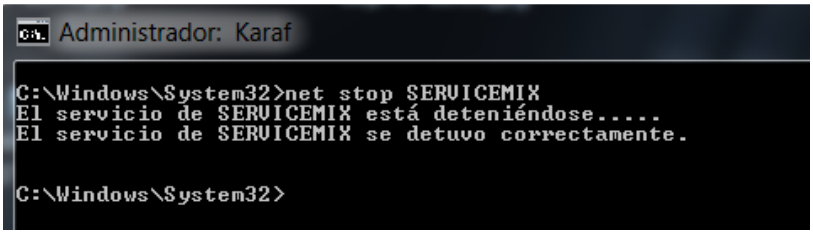

Fig. 101. Parada del servicio de Windows de ServiceMix.

9. Instalamos la consola web de Karaf, desde la consola Karaf para poder acceder también por interfaz web a ServiceMix.

```
karaf@root> features:install webconsole
```

Podemos comprobar que se ha instalado correctamente (Figura 102):

```
karaf@root> features:list | grep webconsole
```

Fig. 102. Instalación del *bundle* de la consola web.

10. Ahora podemos ejecutar la consola del servidor karaf mediante interfaz web en la URL http://localhost:8181/system/console/ (Figura 103).

Usuario: smx
Contraseña: smx

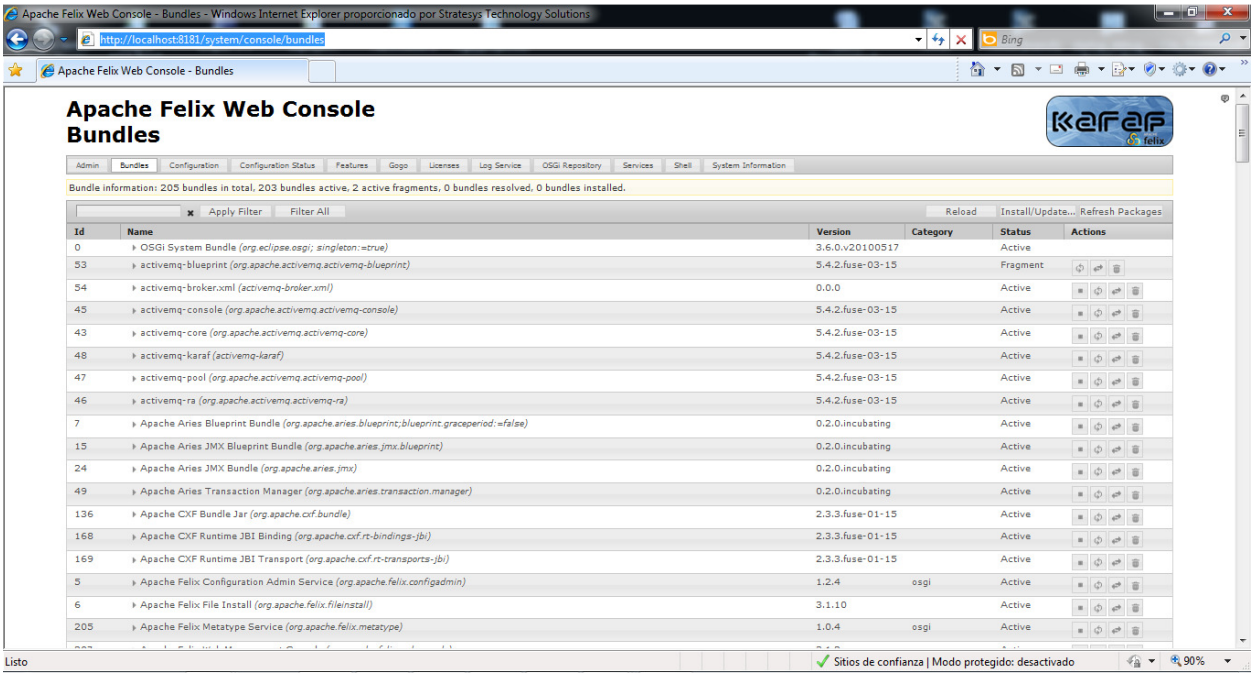

Fig. 103. Consola Karaf accesible por web.

Principalmente la consola web se utiliza para administrar los *bundles* instalados en el ServiceMix. Podemos instalar o desinstalar *bundles*, o parar, refrescar, reinstalar y eliminar *bundles* ya existentes.

13.2.1 Comandos habituales de la consola (línea de comandos) KARAF

Los comandos más habituales serán los siguientes:

`osgi:list` → Listar qué *bundles* hay instalados

Ejemplo (Figura 104):

```
[ 194] [Active     ] [Created     ] [       ] [   60] Apache ServiceMix :: Components :: Saxon Service Engine (2011.01.0.
[ 195] [Active     ] [Created     ] [       ] [   60] Apache ServiceMix :: Components :: WS-Notification Service Engine (2
[ 196] [Active     ] [Created     ] [       ] [   60] Apache Karaf :: Shell Service Wrapper (2.1.4.fuse-00-15)
[ 197] [Active     ] [            ] [       ] [   60] Apache Felix Metatype Service (1.0.4)
[ 198] [Resolved   ] [            ] [       ] [   60] Apache Karaf :: Web Console :: Branding (2.1.4.fuse-00-15)
                                   Hosts: 199
[ 199] [Active     ] [Created     ] [       ] [   60] Apache Felix Web Management Console (3.1.2)
                                   Fragments: 198
[ 200] [Active     ] [Created     ] [       ] [   60] Apache Karaf :: Web Console :: Admin Plugin (2.1.4.fuse-00-15)
[ 201] [Active     ] [Created     ] [       ] [   60] Apache Karaf :: Web Console :: Features Plugin (2.1.4.fuse-00-15)
[ 202] [Active     ] [Created     ] [       ] [   60] Apache Karaf :: Web Console :: Gogo Plugin (2.1.4.fuse-00-15)
karaf@root>
```

Fig. 104. Ejemplo de listado de *bundles*.

Al ejecutar este comando, se nos listan los *bundles* instalados por consola. El primer número indica el identificador (ID) del *bundle*. Por ejemplo, el *bundle* de la consola web Apache Felix Web Management Console (3.1.2) tiene como ID el 199.

`osgi:install` → Instalar un bundle desde un repositorio externo.

Ejemplo:

```
osgi:install -s file:ProjectDir/camel-bundle/target/camel-bundle-1.0-SNAPSHOT.jar
```

`osgi:uninstall <<ID_BUNDLE>>` → Esta instrucción desinstala un *bundle* ya instalado en ServiceMix.

Ejemplo: Si queremos desinstalar el *bundle* de la consola web, ejecutaríamos la instrucción osgi:uninstall 199

`osgi:stop <<ID_BUNDLE>>` → Esta instrucción para un *bundle* ya instalado en ServiceMix.

Ejemplo: Si queremos parar el *bundle* de la consola web, ejecutaríamos la instrucción `osgi:stop 199`

`osgi:start <<ID_BUNDLE>>` → Arrancar un *bundle* ya instalado y que actualmente está parado.

Ejemplo: Si queremos iniciar el *bundle* de la consola web, ejecutaríamos la instrucción `osgi:start 199`

`nmr:list` → Listar los *endpoints* publicados en el bus NMR. Cuando se instalan *bundles* que publican *endpoints* de acceso, con este comando se listan dichos *endpoints*, como se observa en la Figura 105.

Fig. 105. Listado de los endpoints disponibles en el NMR.

`logout` → Con este comando salimos de la consola y paramos el servidor (Figura 106).

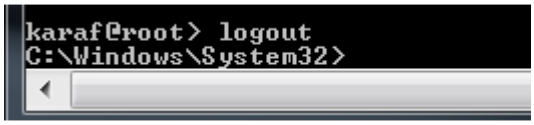

Fig. 106. Comando para salir y parar el servidor ServiceMix.

13.3 *DESARROLLO DE LA APLICACIÓN UPC-WHEATHER*

Una vez tenemos operativo nuestro ServiceMix, ya podemos crear nuestra aplicación, desplegarla al servidor y testearla.

Vamos a crear una aplicación llamada UPC-WEATHER. Esta aplicación consultará un *web service* existente en Internet sobre condiciones climatológicas de una ciudad. Se llamará al servicio mediante dos parámetros, la ciudad y el país y el servicio nos devolverá un texto en formato XML todo en un String (cadena de caracteres) .

Nuestra aplicación leerá estos resultados y los transformará a una nueva estructura de datos. Por otra parte la aplicación publicará otro *web service* que posibilitará la consulta del tiempo de la ciudad en el nuevo formato que le hemos dado. En resumen (Figura 107), el cliente realizará una llamada a un servicio web publicado en ServiceMix, la implementación interna de este webservice llamará al otro *web service* GlobalWeather ya publicado en Internet y realizará la conversión de datos de su formato a nuestro formato.

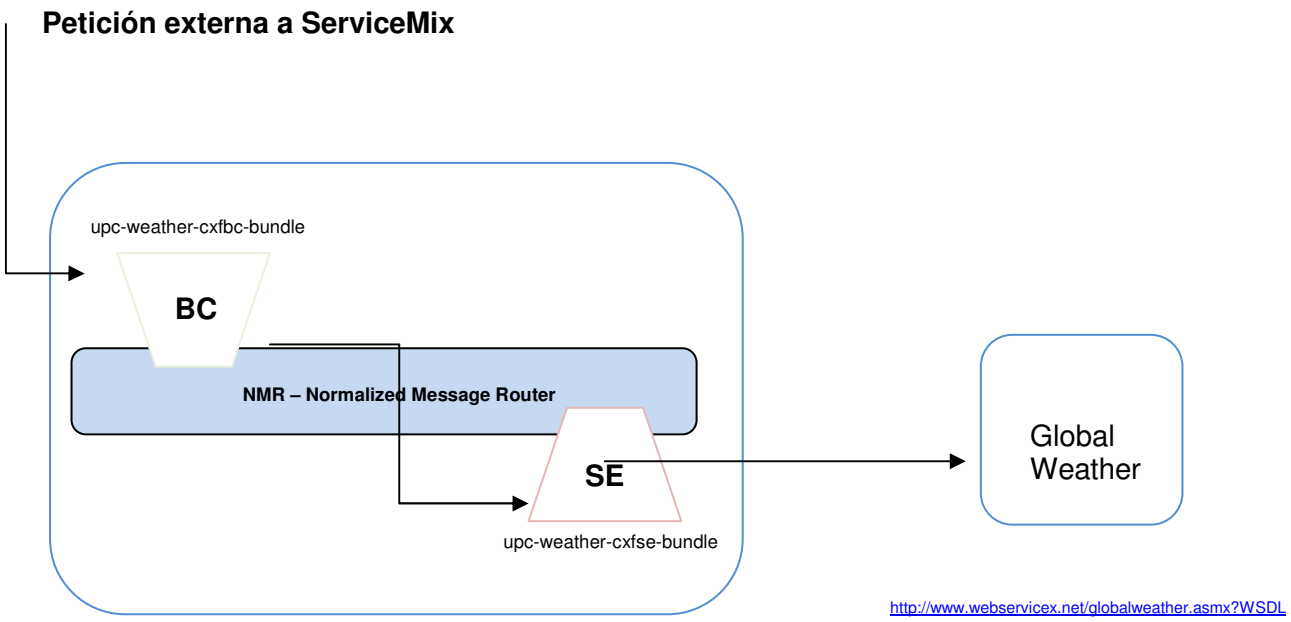

Fig. 107. Funcionamiento de la aplación UPC-WEAHTER

Descripción de componentes/bundles:

- **BC (*Binding Component*) upc-weather-cxfbc-bundle**

  Bundle que publica una interfaz de búsqueda propia vía HTTP/SOAP.

- **SE (*Service Engine*) upc-weather-cxfse-bundle**

  Bundle con la lógica de negocio del servicio. Realizará la conversión de datos entre nuestra interfaz y la interfaz de GlobalWeather.

13.3.1 Creación e importación del proyecto al IDE Eclipse

Para el desarrollo de este proyecto usaremos el IDE (*Integrated Development Enviroment*) *opensource* Eclipse, en concreto la versión Helios Java EE sr2[4] (Figura 108).

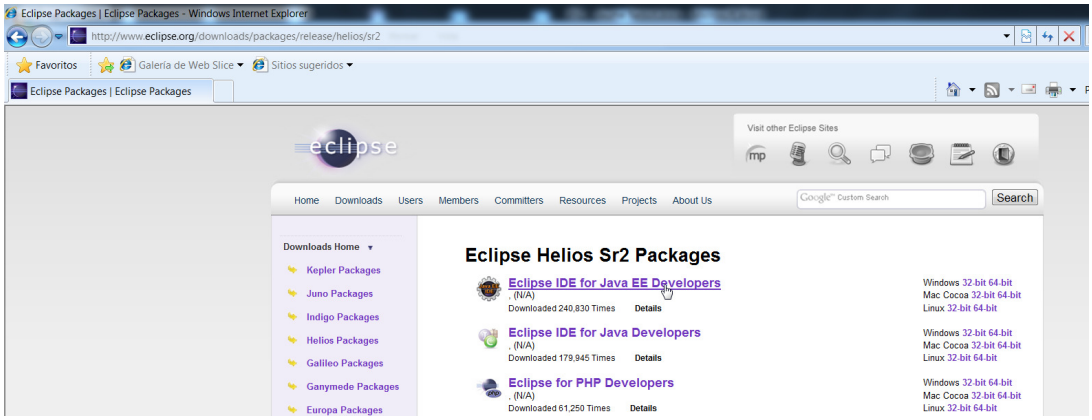

Fig. 108. Link del instalador Eclipse IDE for Java EE Developers.

Empezaremos instalando el IDE Eclipse, aunque en nuestro caso no es necesario empezar por este punto ya que las aplicaciones las crearemos con Maven como veremos más adelante. Pero sí que necesitaremos Eclipse para editar las aplicaciones. Para instalarlo, solamente debemos descomprimir el .zip en nuestra unidad F:\ (Figura 109).

[4] http://www.eclipse.org/downloads/packages/release/helios/sr2

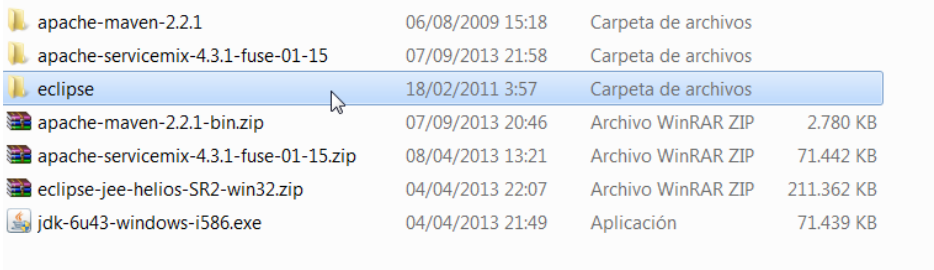

Fig. 109. Directorio de la aplicación Eclipse.

Ahora vamos a crear un directorio de trabajo llamado workspace en la unidad F:\, donde crearemos nuestros proyectos Eclipse (Figura 110).

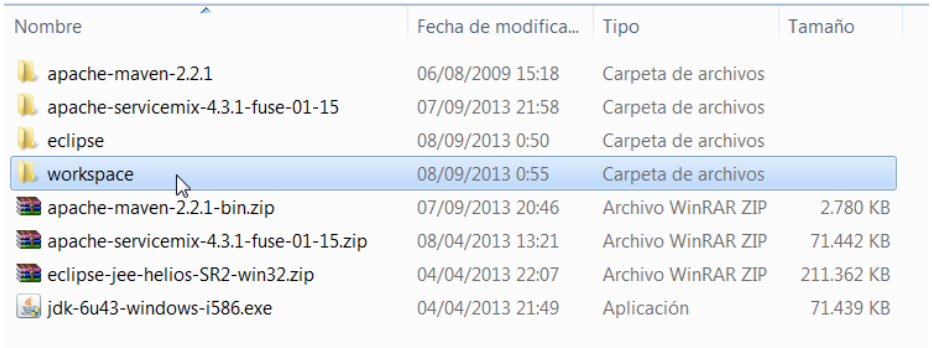

Fig. 110. Creación del directorio de trabajo de Eclipse.

Ahora abrimos el Eclipse lo abrimos mediante el fichero F:\eclipse\eclipse.exe (Fig. 111).

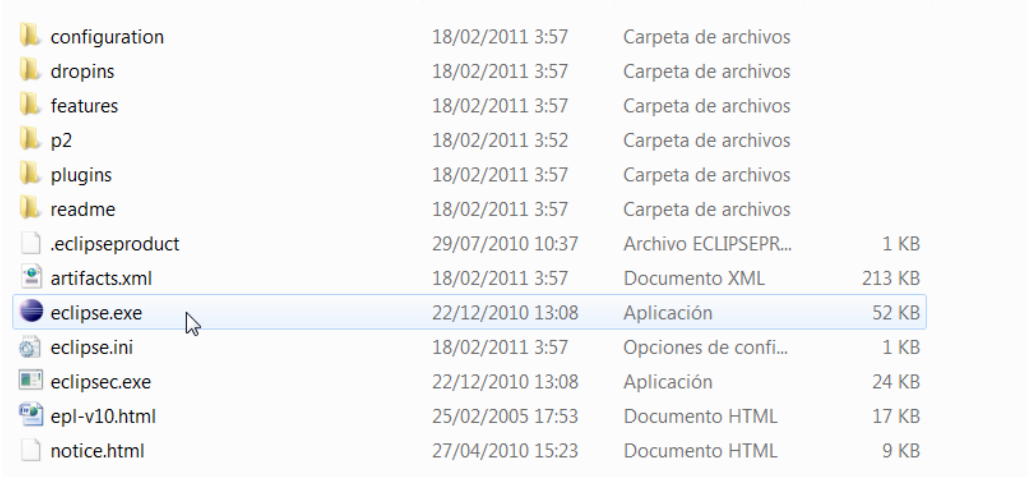

Fig. 111. Archivo para iniciar la aplicación Eclipse.

Cuando lo abrimos, nos pide que seleccionemos un directorio de trabajo. Indicamos nuestro directorio “F:\workspace” y hacemos clic a OK (Fig. 112).

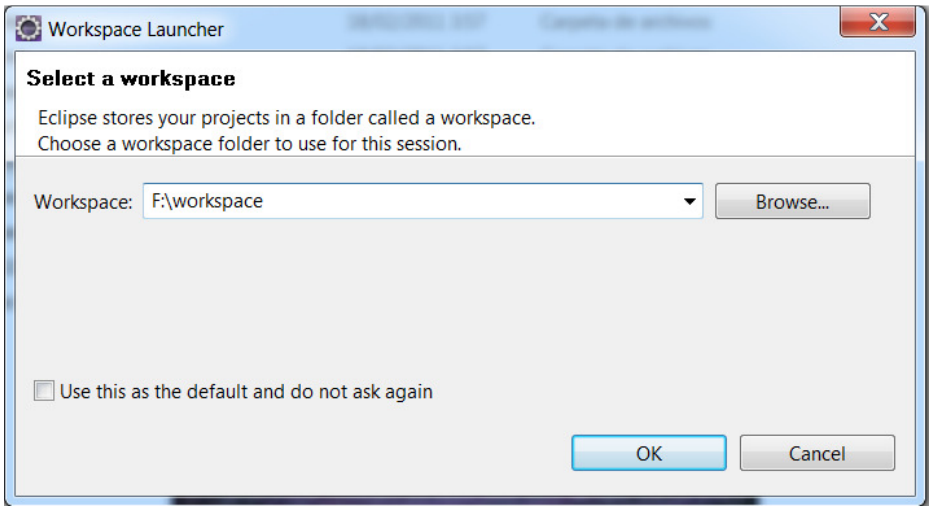

Fig. 112. Selección de directorio de trabajo de nuestro Eclipse.

En este punto, ya tenemos nuestro Eclipse abierto tal y como se muestra en la Figura 113.

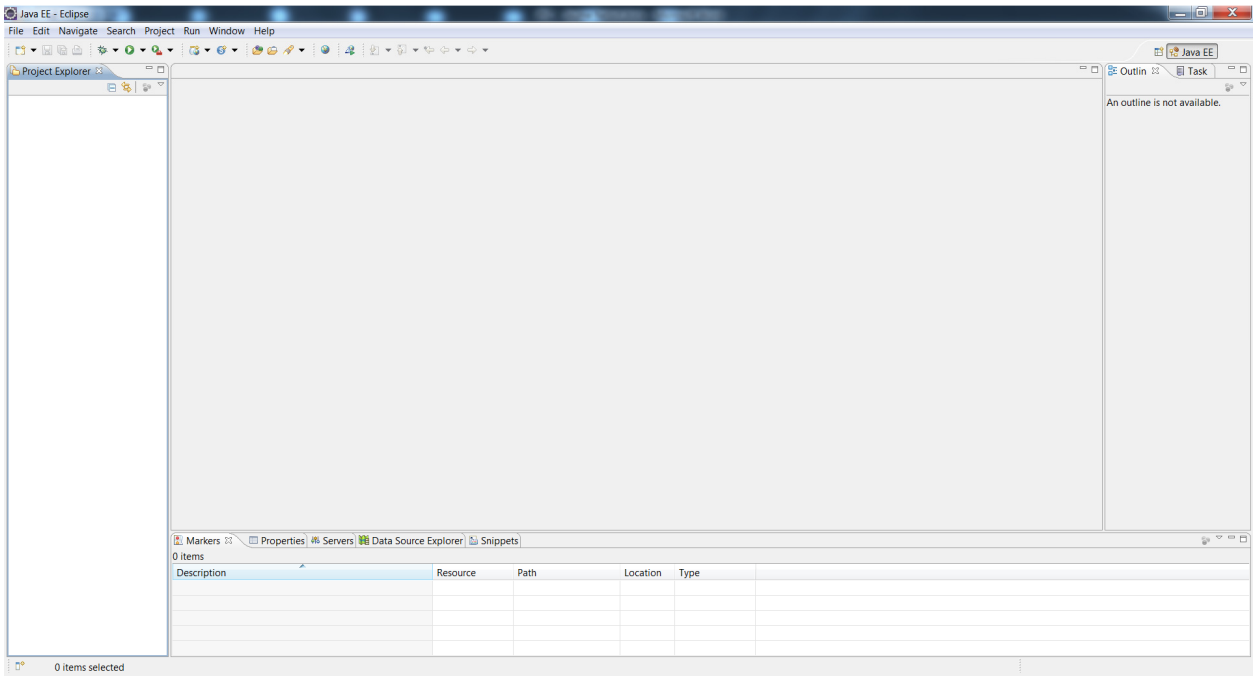

Fig. 113. Pantalla principal de la aplicación Eclipse.

Una vez abierto el Eclipse, instalaremos un *plug-in* (una extensión) de Eclipse para que nos permita trabajar con proyectos Maven. Para ello Iremos a la opción Help → Install New Software (Figura 114).

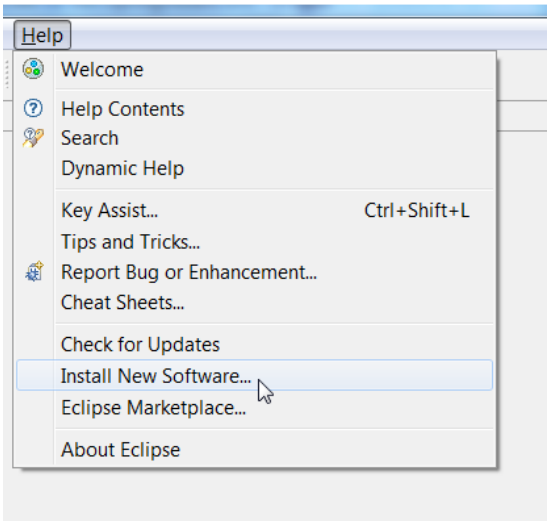

Fig. 114. Opción de menú de Eclipse Install New Software.

Se nos abrirá una pantalla de búsqueda de componentes .Ahora escribimos la URL[5] en el campo de búsqueda Work with y aparecerá en la lista el componente Maven Integration for Eclipse, marcamos el *checkbox* para seleccionarlo y hacemos clic en el botón Next (Figura 115)

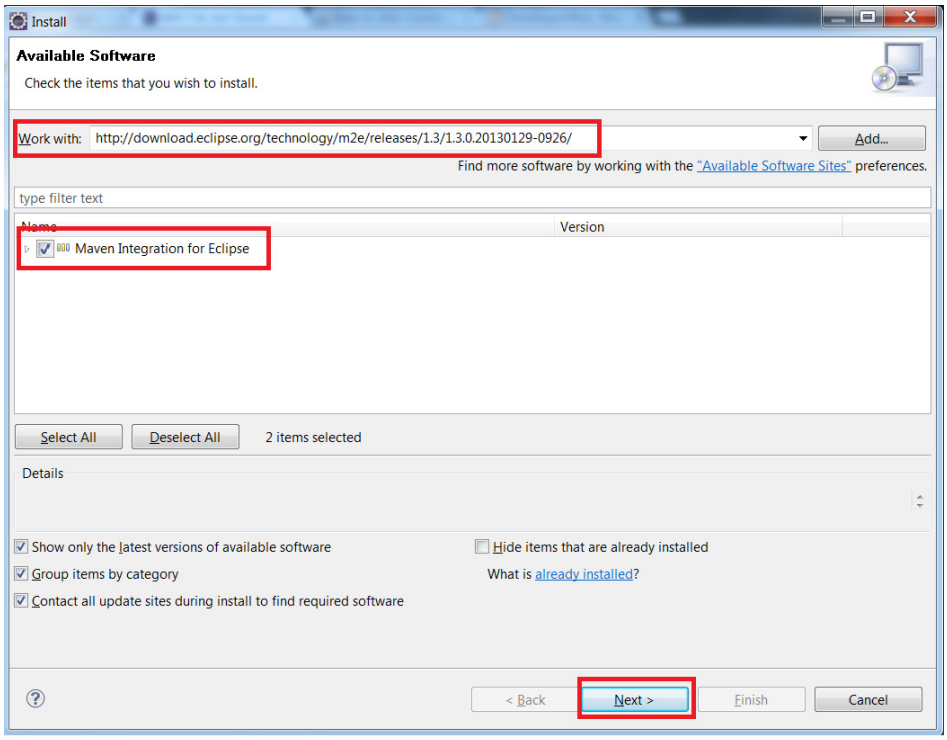

Fig. 115. Eclipse Install New Software.

[5] http://download.eclipse.org/technology/m2e/releases/1.3/1.3.0.20130129-0926/

En el siguiente paso simplemente hacemos clic en el botón Next (Figura 116).

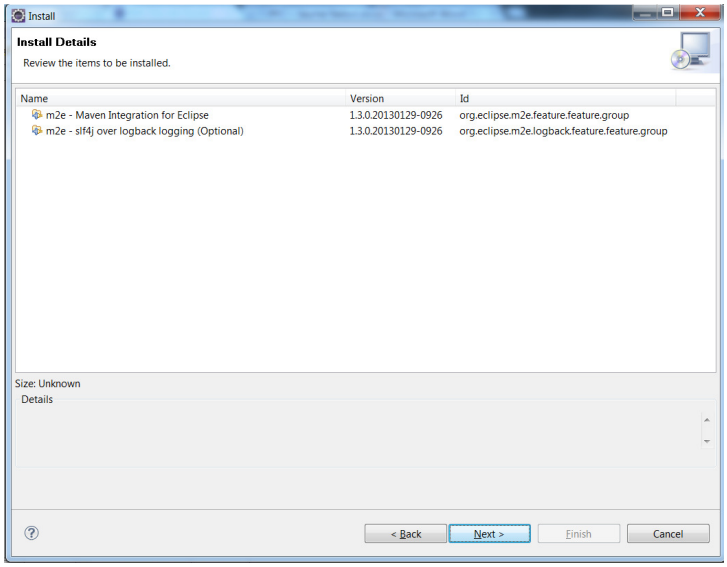

Fig. 116. *Plug-in* Maven Integration for Eclipse.

En el siguiente paso seleccionamos que aceptamos la licencia y hacemos clic en Finish (Figura 117).

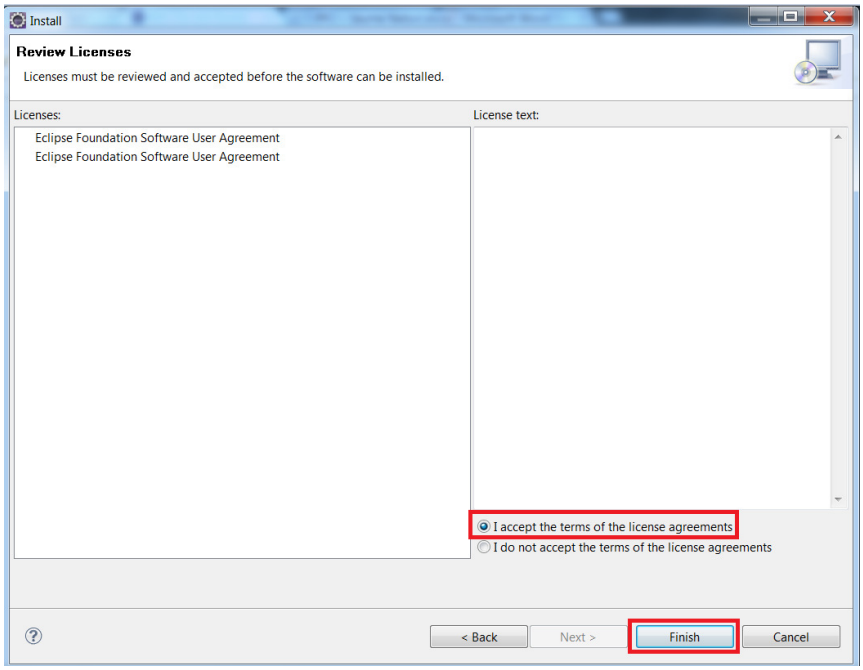

Fig. 117. Aceptamos la licencia del componente Maven for Ecilpse.

Una vez completada la instalación, nos pedirá que reiniciamos el Eclipse, marcamos la opción Restart Now (Figura 118).

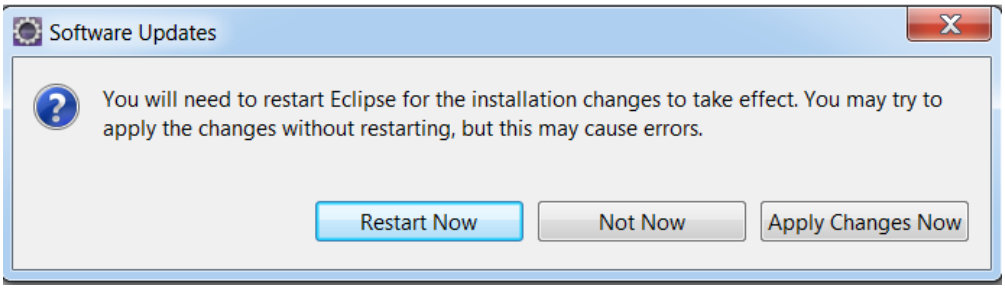

Fig. 118. Ventana emergente para reinicar el Eclipse.

Una vez tenemos el Eclipse con el plugin Maven instalado, vamos a crear nuestros proyectos Eclipse que constituirán nuestra aplicación. Pero para ello, haremos uso de la herramienta Maven a través de sus arquetipos. Un arquetipo Maven es una configuración preestablecida de proyectos, y esto nos permite crear automáticamente la estructura de los proyectos sin tener que hacerla a mano. Con Eclipse también se pueden crear los proyectos, pero Maven ya tiene arquetipos para proyectos de ServiceMix, con lo cual nos ahorraremos trabajo si los usamos. En concreto, como queremos usar servicios web, haremos uso de un arquetipo del tipo CXF. Como ya hemos estudiado, ServiceMix trabaja con Apache CXF para la creación de servicios web. Dentro de los arquetipos CXF, existen los del tipo *wsdl* first. Si recordamos el punto donde hemos estudiado la creación de *wsdl*, vimos que la mejor estrategia para crear contratos de servicio estandarizados era primero crear un wsdl y posteriormente generar las clases Java a partir de él, y no viceversa. Como es un muy comuna la realización de este tipo de proyectos, Maven ya proporciona dicho arquetipo.

Usaremos el mismo arquetipo para los dos proyectos a crear (upc-weather-cxfbc-bundle y upc-weather-cxfse-bundle). Empezaremos por el componente SE. Para ello, iniciamos la consola de comandos de Windows y nos posicionamos en el directorio “F:\workspace”. Ejecutaremos el siguiente comando Maven:

“mvn archetype:generate -DarchetypeGroupId=org.apache.servicemix.tooling -DarchetypeArtifactId=servicemix-cxf-wsdl-first-osgi-bundle -DgroupId=edu.upc.weather_client.impl -DartifactId=upc-weather-cxfse-bundle -Dversion=1.0.0-SNAPSHOT”

Antes de ejecutarlo, explicaremos qué significa cada parámetro:

-DarchetypeGroupId=org.apache.servicemix.tooling → Es el *package* (paquete) de los arquetipos ServiceMix.

-DarchetypeArtifactId=servicemix-cxf-wsdl-first-osgi-bundle → Es el nombre del artefacto maven correspondiente a un proyecto ServiceMix CXF del tipo wsdl first.
-DgroupId=edu.upc.weather_client.impl → Será el paquete Java de nuestro proyecto, en este paquete habrán las clases Java que implementarán el servicio web.

-DartifactId=upc-weather-cxfse-bundle → Será el nombre de nuestro proyecto o artefacto.

-Dversion=1.0.0-SNAPSHOT → Indicamos que es la primera versión de nuestro proyecto. Con SNAPSHOT indicamos que esta versión es evolutiva, es decir, que estamos trabajando para obtener la versión 1.0.0.

Ahora ya podemos ejecutar el comando (Figura 119).

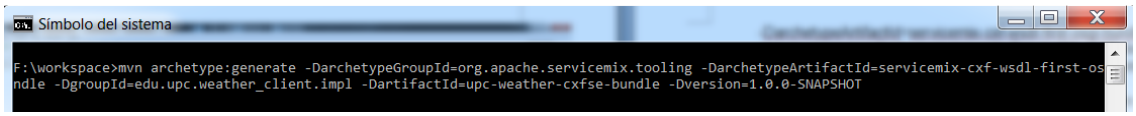

Fig. 119. Ejecución del comando Maven para la creación de nuestro componente SE.

Durante la instalación nos pide que demos confirmación a los parámetros de creación de nuestro artefacto (Figura 120). Simplemente hacemos Intro para continuar.

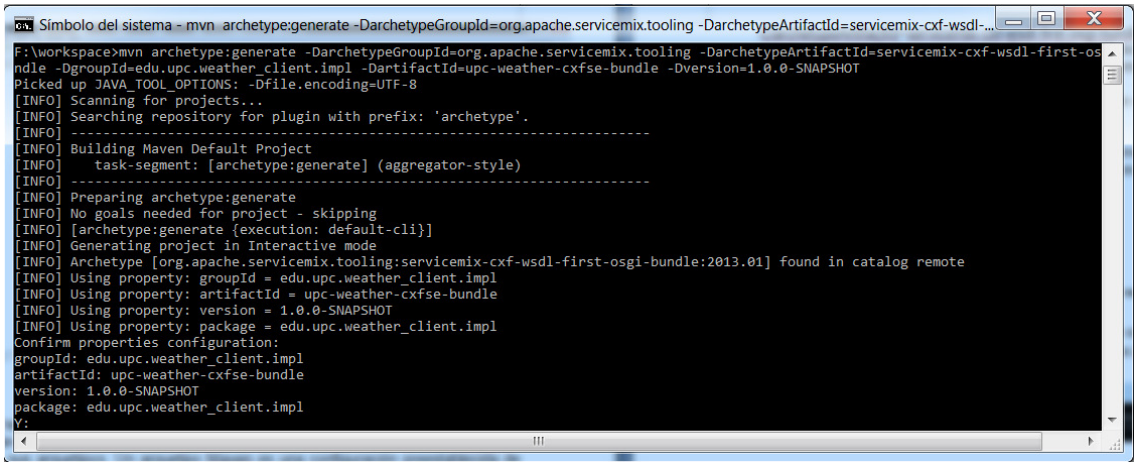

Fig. 120. Confirmación de las propiedades de configuración de nuestro componente SE.

Al confirmar, se finaliza la construcción del proyecto. Maven nos informa con el mensaje "BUILD SUCCESSFUL" para indicarnos que el proceso ha finalizado correctamente (Fig. 121).

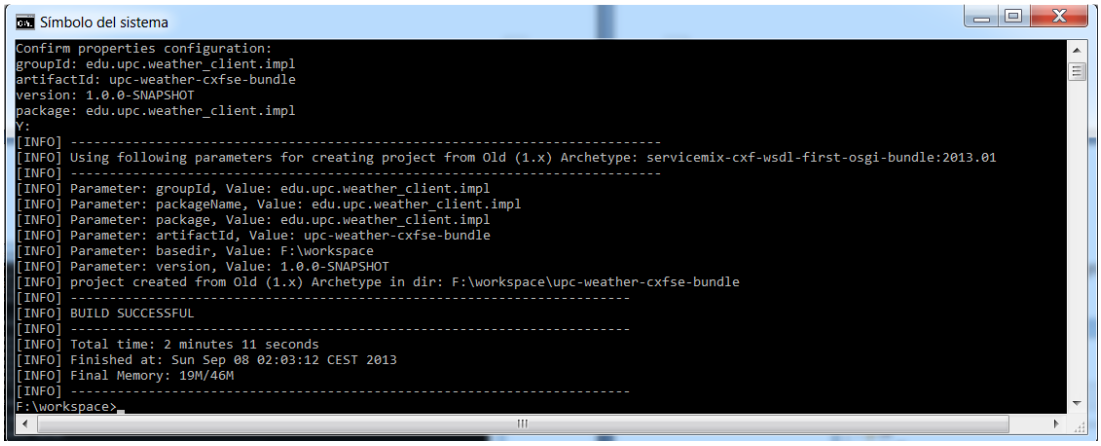

Fig. 121. Proceso de creación del proyecto SE finalizado con éxito.

Ahora repetimos el proceso para el componente BC. Con el comando Maven (Figura 122):

"mvn archetype:generate -DarchetypeGroupId=org.apache.servicemix.tooling -DarchetypeArtifactId=servicemix-cxf-wsdl-first-osgi-bundle -DgroupId=edu.upc.weather_client.impl -DartifactId=upc-weather-cxfbc-bundle -Dversion=1.0.0-SNAPSHOT"

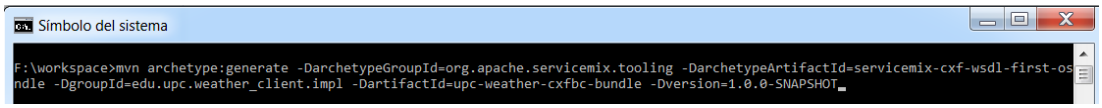

Fig. 122. Ejecución del comando Maven para la creación de nuestro componente BC.

Una vez ejecutado y aceptado las propiedades de configuración, obtenemos el mensaje de creación satisfactoria (Figura 123).

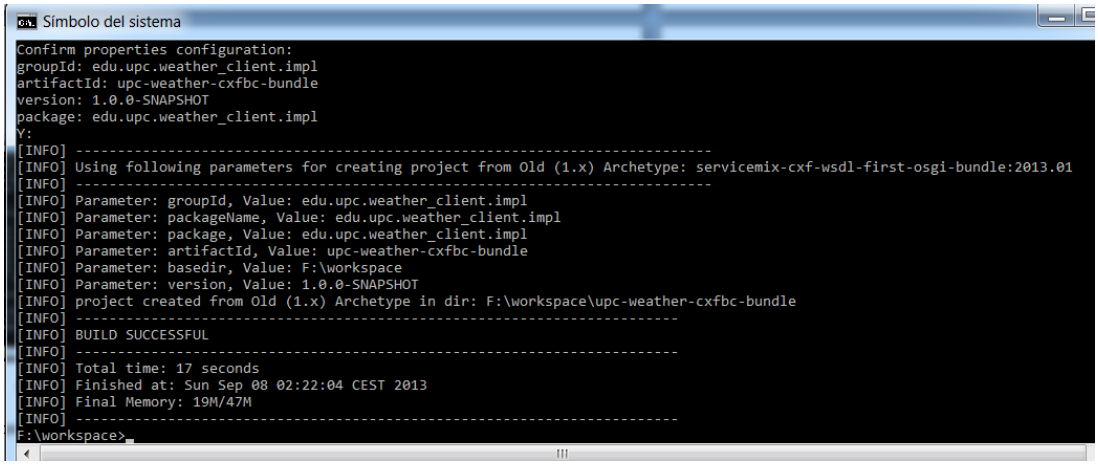

Fig. 123. Proceso de creación del proyecto BC finalizado con éxito.

Una vez creados los proyectos, vamos a importar desde el Eclipse los dos componentes upc-weather-cxfbc-bundle y upc-weather-cxfse-bundle. Para ello vamos a seleccionar la opción File ->Import (Figura 124).

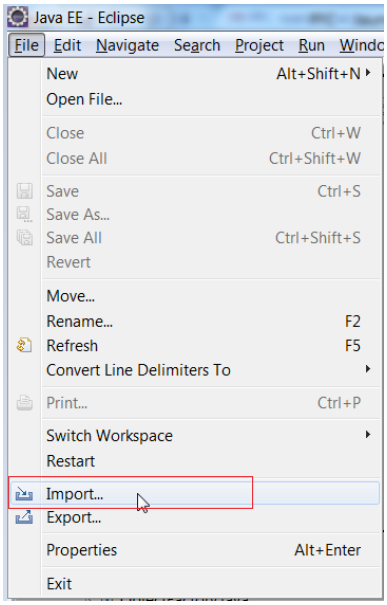

Fig. 124. Importación de proyectos en Eclipse.

A continuación seleccionamos Existing Maven Projects (Figura 125), ya que nuestros *bundles* están construidos como una aplicación Maven.

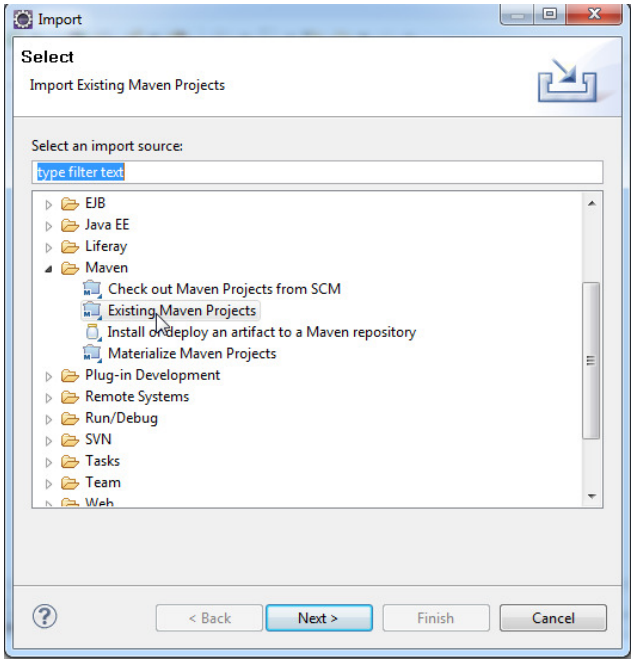

Fig. 125. Selección de proyectos Maven.

Hacemos clic en Next y en la siguiente pantalla (Figura 126) Seleccionamos como Root Directory la ubicación de nuestro componente upc -weather-cxfbc-bundle. Automáticamente se seleccionará el fichero pom.xml que es el fichero que describe el proyecto Maven. Hacemos clic en Finish.

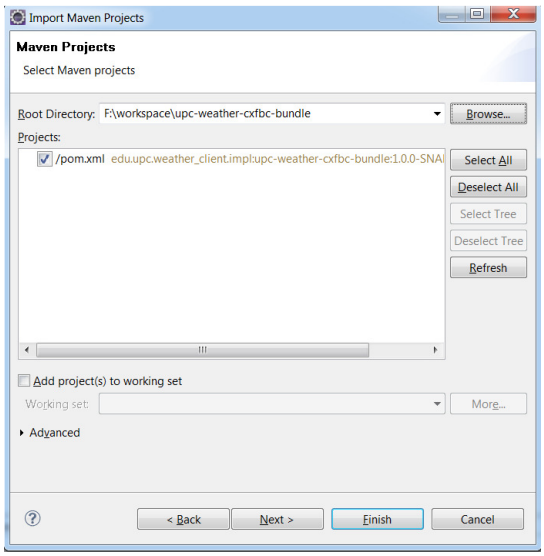

Fig. 126. Selección del fichero pom.xml del componente.

Debemos repetir este proceso para el otro componente upc -weather-cxfse-bundle. Una vez concluido, tendremos en nuestro Project Explorer de Eclipse las dos aplicaciones importadas correctamente, tal y como se muestra en la Figura 127.

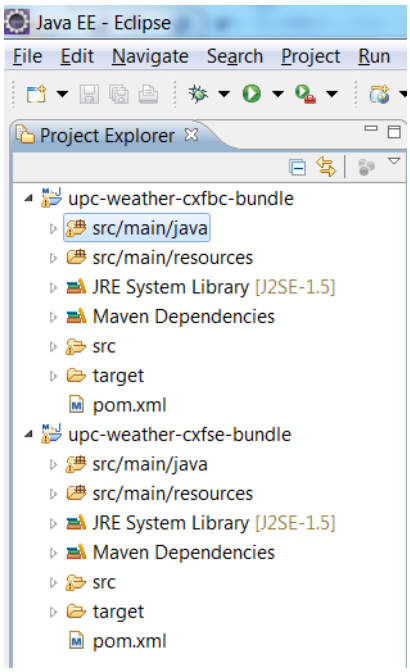

Fig. 127. Project Explorer del Eclipse con nuestros proyectos.

En este punto ya tenemos nuestras aplicaciones listas para ser modificadas a través del Ecipse. A continuación crearemos el proyecto upc-weather desde el Eclipse, que servirá para hacer de proyecto padre de los dos sub-proyectos y crear una *feature* de ServiceMix. Una *feature* es una agrupación de *bundles* para así desplegarlos a la vez y no tener que desplegarlos por separado.

13.3.2 Feautre upc-weather

El primer paso será crear un proyecto Maven. Abrimos la opción File → New Project → Other (Figura 128).

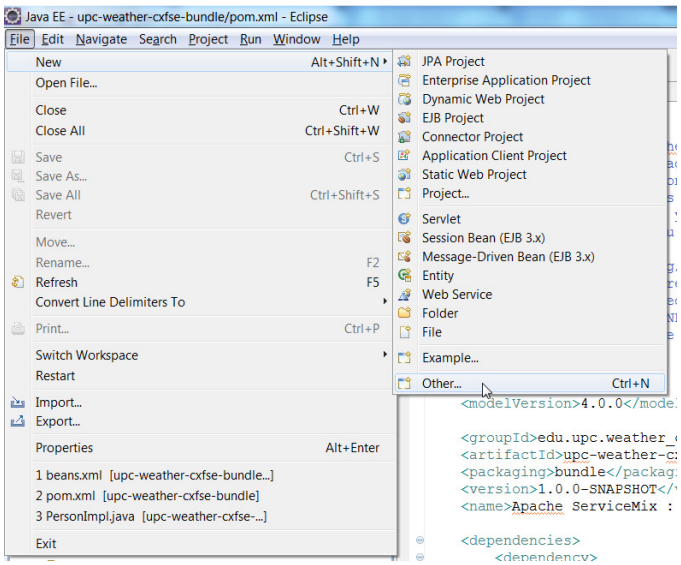

Fig. 128. Creación de proyecto Maven. Paso 1.

Seleccionamos Maven Project y clic en Next (Fig. 129).

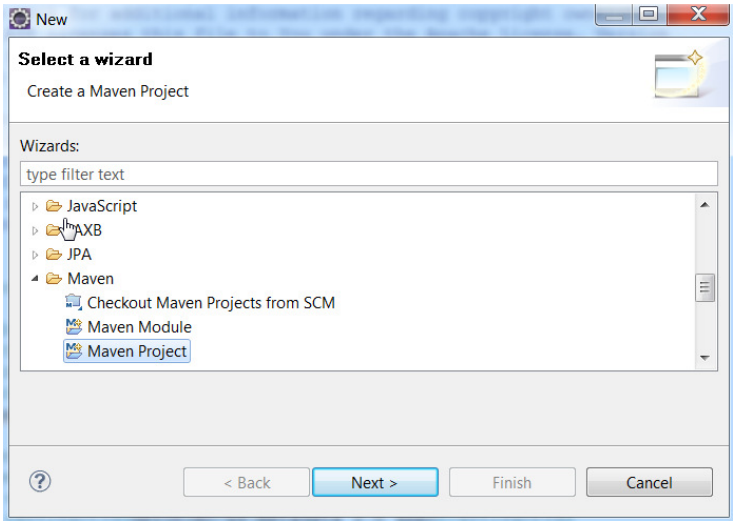

Fig. 129. Creación de proyecto MAven. Paso 2.

Marcamos el *checkbox* de la opción Create a simple Project y hacemos clic en Next (Fig. 130).

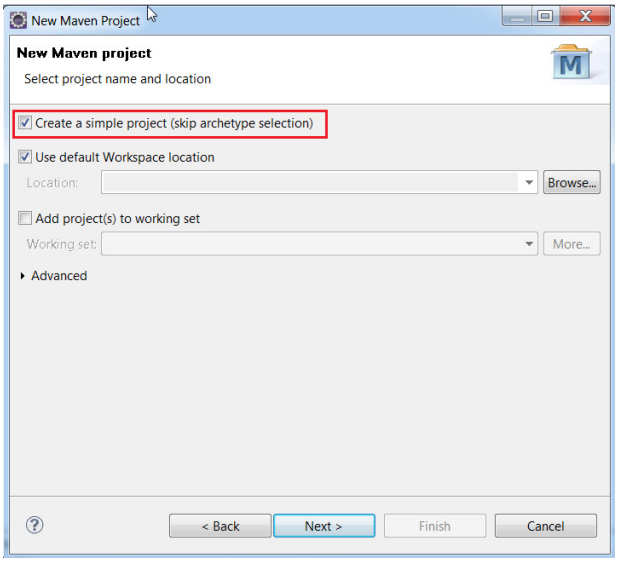

Fig. 130. Creación de proyecto MAven. Paso 3.

Ahora rellenamos la siguiente información y hacemos clic en Finish (Figura 131).

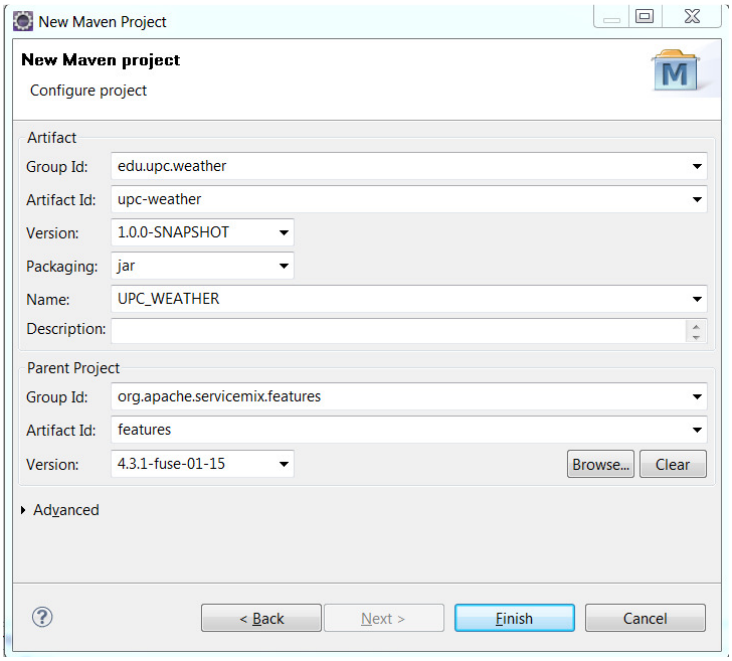

Fig. 131. Creación de proyecto MAven. Paso 4.

El proyecto queda de la siguiente manera (Figura 132).

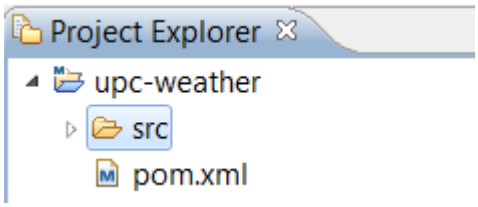

Fig. 132. Proyecto upc-weather creado.

Primero borramos el paquete src como muestra la Figura 133 ya que no habrá código Java en este proyecto.

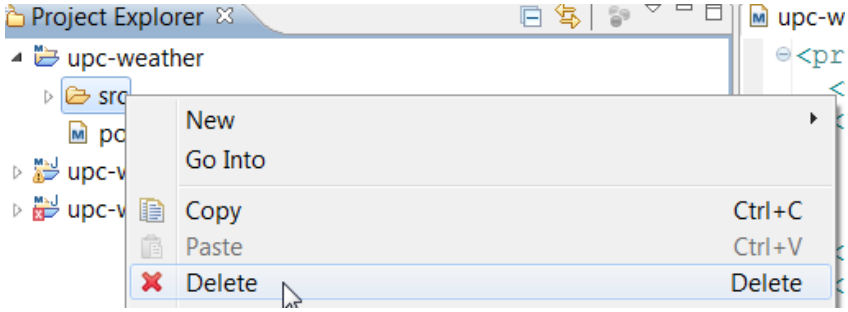

Fig. 133. Borrado de una carpeta/archivo del proyecto.

Ahora modificamos el fichero pom.xml. Este fichero es el fichero que lee Maven para compilar, resolver dependencias de librerías, etc. Escribiremos las siguientes propiedades, marcadas rojo, para que Maven sepa encontrar las referencias de los componentes *feature*.

```
<name>UPC_WEATHER</name>
  <packaging>pom</packaging>

<modules>
 <module>upc-weather-cxfse-bundle</module>
 <module>upc-weather-cxfbc-bundle</module>
</modules>

  <repositories>
        <!-- FuseSource maven repositories -->
        <repository>
            <id>fusesource.releases</id>
            <name>FuseSoure releases repository</name>
            <url>http://repo.fusesource.com/maven2/</url>
            <releases>
                <enabled>true</enabled>
            </releases>
            <snapshots>
                <enabled>false</enabled>
            </snapshots>
        </repository>
        <repository>
            <id>fusesource.snapshots</id>
            <name>FuseSource Snapshot Repository</name>
            <url>http://repo.fusesource.com/maven2-snapshot</url>
            <snapshots>
                <enabled>true</enabled>
            </snapshots>
            <releases>
                <enabled>false</enabled>
            </releases>
```

```
    </repository>
</repositories>

<pluginRepositories>
    <!-- FuseSource maven repositories -->
    <pluginRepository>
        <id>fusesource.releases</id>
        <name>FuseSoure releases repository</name>
        <url>http://repo.fusesource.com/maven2/</url>
        <releases>
            <enabled>true</enabled>
        </releases>
        <snapshots>
            <enabled>false</enabled>
        </snapshots>
    </pluginRepository>
    <pluginRepository>
        <id>fusesource.snapshots</id>
        <name>FuseSource Snapshot Repository</name>
        <url>http://repo.fusesource.com/maven2-snapshot</url>
        <snapshots>
            <enabled>true</enabled>
        </snapshots>
        <releases>
            <enabled>false</enabled>
        </releases>
    </pluginRepository>
 </pluginRepositories>
```

Ahora, crearemos el fichero XML de la *feature*, que será el que se despliegue al ServiceMix. Para ello creamos un nuevo fichero , botón derecho sobre el proyecto y opción New → File (Figura 134).

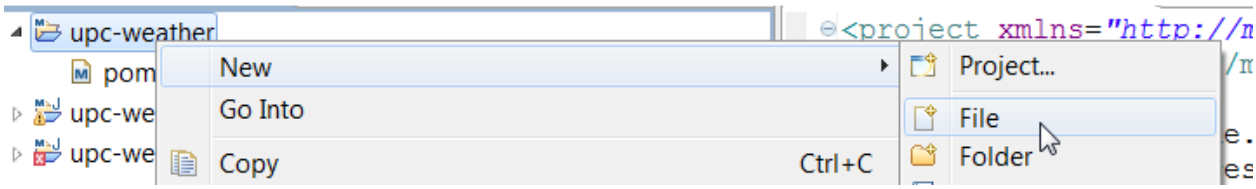

Fig. 134. Creación de un nuevo fichero. Paso 1.

Ponemos el nombre del fichero upc-weather-feature-1.0.0-SNAPSHOT.xml y hacemos clic en el botón Finish (Figura 135).

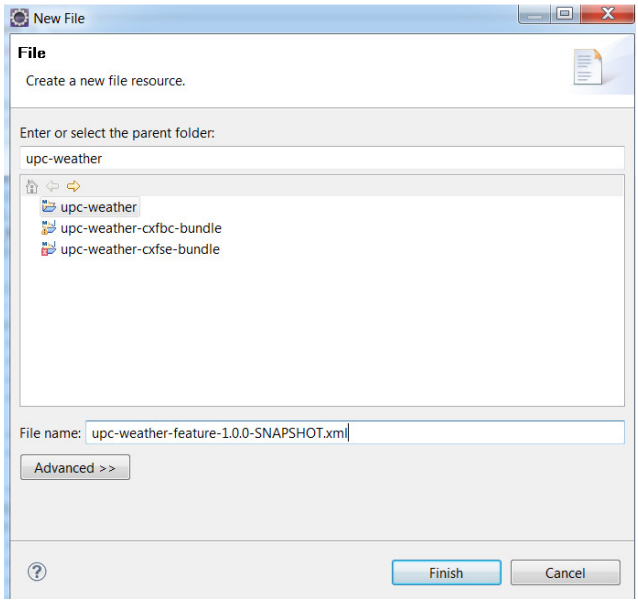

Fig. 135. Creación de un nuevo fichero. Paso 2.

Ahora modificamos el fichero, abriéndolo (doble-clic sobre el fichero), posicionándonos en la pestaña Source (Figura 136) y añadimos la información que crea la *feature* en base a nuestros dos proyectos:

```
<?xml version="1.0" encoding="UTF-8"?>
<features name="upc-pfc-features">
  <feature name="upc-pfc-connect-feature" version="${versionFUSE}">
    <bundle>mvn:edu.upc.weather/upc-weather-cxfse-bundle/1.0.0-SNAPSHOT</bundle>
    <bundle>mvn:edu.upc.weather/upc-weather-cxfbc-bundle/1.0.0-SNAPSHOT</bundle>
  </feature>
</features>
```

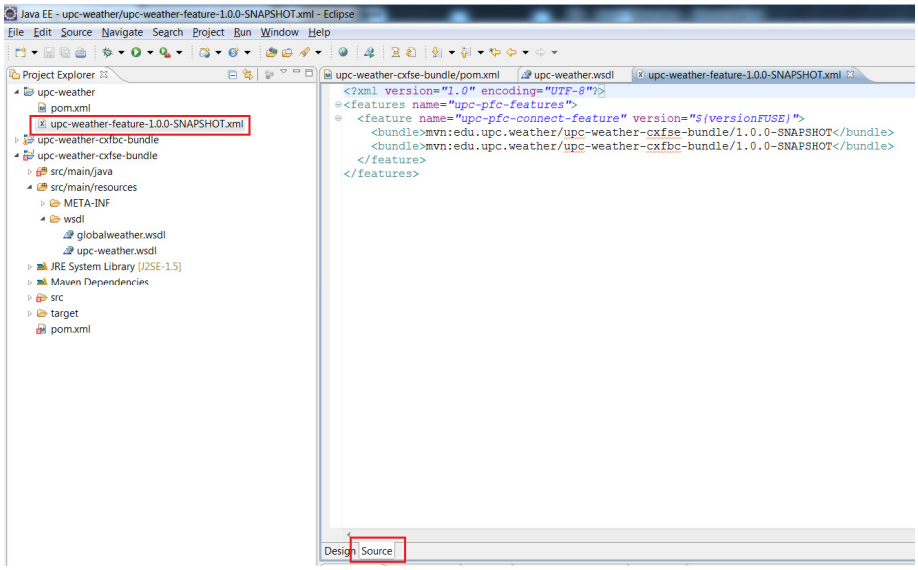

Fig. 136. Edición de un fichero.

Se puede apreciar mediante el prefijo mvn: que los *bundles* estarán instalados en el repositorio local del Maven, es decir, cuando hayamos creado los componentes BC y SE, los instalaremos en el repositorio Maven para que puedan estar disponibles al desplegar la *feature.* A continuación, modificaremos el proyecto SE.

Importante: Se debe copiar el fichero pom.xml al directorio "F:\workspace". Porque al ser el .pom del proyecto padre, debe estar en la raíz de todos los proyectos hijos.

13.3.3 Bundle upc-weather-cxfse-bundle

Como ya se ha introducido, en este componente reside la lógica de negocio. Se llamará al servicio web GlobalWeather publicado en Internet y publicaremos al NMR este servicio de forma interna en el Bus, para que pueda ser llamado por el componente BC. A este proceso, se le llama crear un endpoint interno. También realizaremos la conversión de datos de GlobalWeather a upc-weather. La estructura de carpetas del componente es la siguiente (Figura 137):

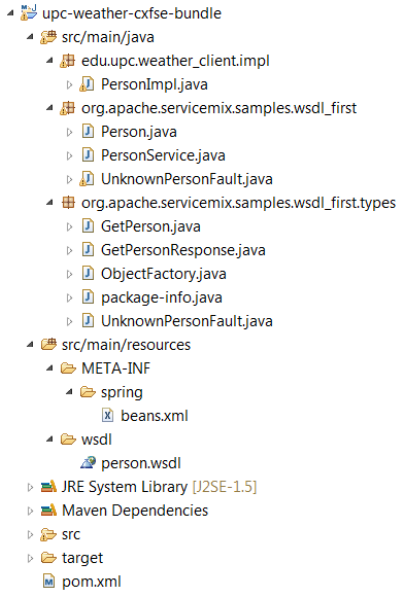

Fig. 137. Estructura de directorios del componente SE.

La primera tarea será limpiar el proyecto ya que al utilizar el arquetipo Maven, se ha creado el proyecto con un ejemplo de prueba. Vamos a eliminar los siguientes paquetes (mirar en la Figura 136 su localización):

- org.apache.servicemix.samples.wsdl_first
- org.apache.servicemix.samples.wsdl_first.types

También eliminamos el *wsdl* de prueba y su carpeta en la que está (wsdl):

- Person.wsdl

Para eliminar un paquete o un archivo nos situamos sobre el elemento, clic con botón derecho y seleccionamos la opción Delete, tal y como se muestra en la Figura 138.

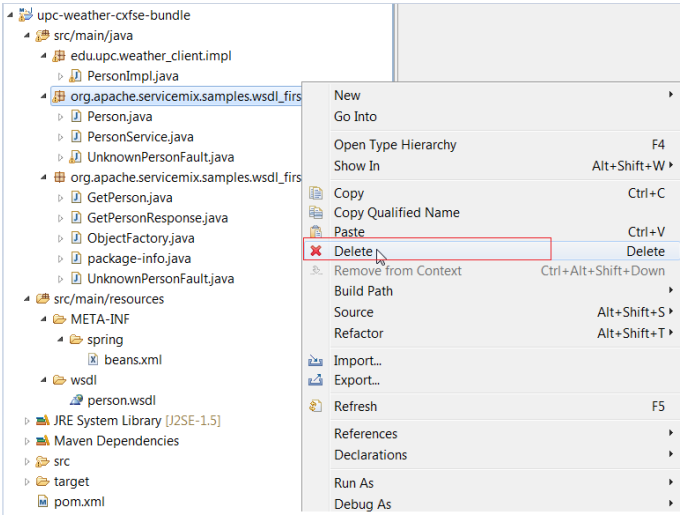

Fig. 138. Eliminación de un paquete o archivo.

Al eliminar estos archivos han salido errores como se aprecia en la Figura 139, porque se hacía referencia a ellos. Los iremos arreglando a medida que vayamos haciendo nuestra aplicación. Es posible, a modo general, que Eclipse presente errores pero no sean errores de verdad, ya que a veces el Eclipse no se refresca bien y tiene comportamientos un poco extraños. En nuestro caso, como compilaremos las aplicaciones con Maven, no debemos fijarnos en exceso en los errores que muestra Eclipse.

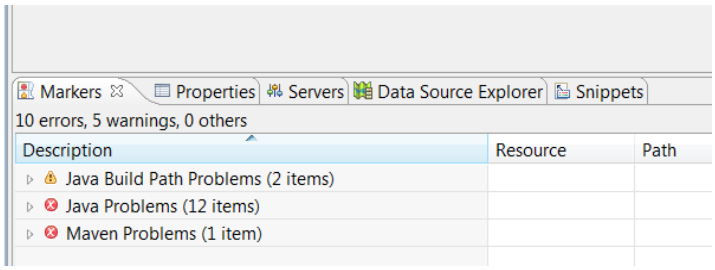

Fig. 139. Problemas generados al haber eliminado los archivos de ejemplo.

El primer paso será descargarnos el *wsdl* del servicio GlobalWeather publicado en Internet ya que deberemos generar las clases Java a partir de él para poder llamar a este servicio. Introducimos en el navegador la URL[6] y se nos mostrará el archivo WSDL.

[6] http://www.webservicex.net/globalweather.asmx?WSDL

Guardamos la página con el nombre **internet-weather.wsdl** y posteriormente guardamos el archivo en la ruta de nuestro proyecto (Figura 140):

"F:\workspace\upc-weather-cxfse-bundle\src\main\resources\"

Ahora ya hemos sustituido el wsdl de ejemplo que venía por el que utilizaremos nosotros para llamar al servicio web.

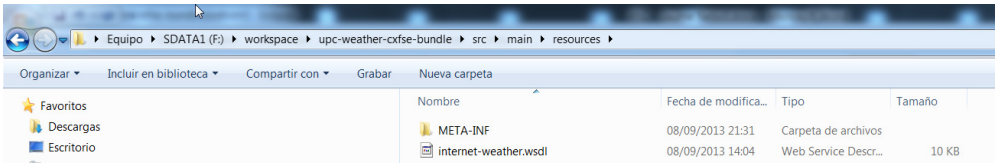

Fig. 140. Directorio donde guardamos el fichero internet-weather.wsdl descargado.

Para que Eclipse vea que se ha puesto este archivo hay que refrescar el proyecto, opción Refresh en el menú contextual del proyecto (Figura 141).

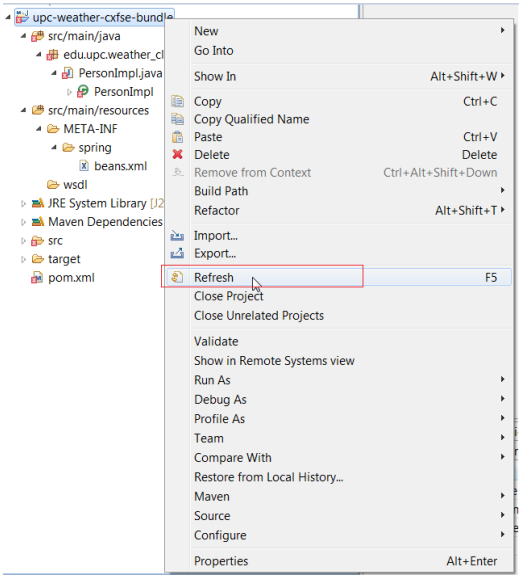

Fig. 141. Opción de refresco del proyecto.

upc-weather.wsdl

El siguiente paso será incluir en nuestro proyecto nuestro wsdl de upc-weather. Para ello, creamos un nuevo archivo wsdl en la capeta src/main/resources/ desde el Eclipse. Editamos este fichero e incluimos el siguiente texto:

```
<?xml version="1.0" encoding="UTF-8"?>
<wsdl:definitions name="upc-weather" targetNamespace="http://weather.upc.edu/upc-weather-client/1.0" xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
xmlns:tns="http://weather.upc.edu/upc-weather-client/1.0"
```

```
xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/">
  <wsdl:types>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
xmlns:tns="http://weather.upc.edu/upc-weather-client/1.0" elementFormDefault="unqualified"
targetNamespace="http://weather.upc.edu/upc-weather-client/1.0" version="1.0">
<xs:element name="getWeather" type="tns:getWeather"/>
<xs:element name="getWeatherResponse" type="tns:getWeatherResponse"/>
<xs:complexType name="getWeather">
    <xs:sequence>
      <xs:element minOccurs="0" name="city" type="xs:string"/>
      <xs:element minOccurs="0" name="country" type="xs:string"/>
    </xs:sequence>
  </xs:complexType>
<xs:complexType name="getWeatherResponse">
    <xs:sequence>
      <xs:element minOccurs="0" name="return" type="tns:weather"/>
    </xs:sequence>
  </xs:complexType>
<xs:complexType name="weather">
    <xs:sequence>
      <xs:element minOccurs="0" name="dewPoint" type="xs:string"/>
      <xs:element minOccurs="0" name="location" type="xs:string"/>
      <xs:element minOccurs="0" name="pressure" type="xs:string"/>
      <xs:element minOccurs="0" name="relativeHumidity" type="xs:string"/>
      <xs:element minOccurs="0" name="skyConditions" type="xs:string"/>
      <xs:element minOccurs="0" name="status" type="xs:string"/>
      <xs:element minOccurs="0" name="temperature" type="xs:string"/>
      <xs:element minOccurs="0" name="time" type="xs:string"/>
      <xs:element minOccurs="0" name="visibility" type="xs:string"/>
      <xs:element minOccurs="0" name="wind" type="xs:string"/>
    </xs:sequence>
  </xs:complexType>
</xs:schema>
  </wsdl:types>
  <wsdl:message name="getWeatherResponse">
    <wsdl:part name="parameters" element="tns:getWeatherResponse">
    </wsdl:part>
  </wsdl:message>
  <wsdl:message name="getWeather">
    <wsdl:part name="parameters" element="tns:getWeather">
    </wsdl:part>
  </wsdl:message>
  <wsdl:portType name="upc-weather">
    <wsdl:operation name="getWeather">
      <wsdl:input name="getWeather" message="tns:getWeather">
    </wsdl:input>
      <wsdl:output name="getWeatherResponse" message="tns:getWeatherResponse">
    </wsdl:output>
    </wsdl:operation>
  </wsdl:portType>
  <wsdl:binding name="WeatherWSServiceSoapBinding" type="tns:upc-weather">
    <soap:binding style="document" transport="http://schemas.xmlsoap.org/soap/http"/>
    <wsdl:operation name="getWeather">
      <soap:operation soapAction="" style="document"/>
      <wsdl:input name="getWeather">
        <soap:body use="literal"/>
      </wsdl:input>
      <wsdl:output name="getWeatherResponse">
        <soap:body use="literal"/>
      </wsdl:output>
    </wsdl:operation>
  </wsdl:binding>
  <wsdl:service name="upc-weather">
    <wsdl:port name="upc-weatherPort" binding="tns:WeatherWSServiceSoapBinding">
      <soap:address location="http://localhost:9090/upc-weatherPort/"/>
    </wsdl:port>
  </wsdl:service>
</wsdl:definitions>
```

Con este wsdl, hemos definido una operación de llamada al servicio web getWeather con parámetros ciudad y país. Y la operación del mensaje de retorno nos devolverá una

estructura con toda la información del tiempo, tal y como se muestra en el tipo de datos GetWeatherResponse.

Así de esta forma, ya tenemos los dos *wsdl* en nuestro proyecto (Figura 142).

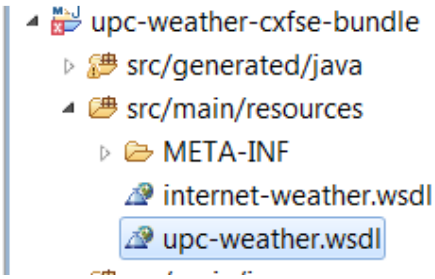

Fig. 142. Archivos wsdl usados en el componente SE.

Ahora realizaremos modificaciones en el fichero pom.xml.

Pom.xml

Primero añadiremos la referencia de su proyecto padre, el proyecto upc-weather ya creado. Incluimos el siguiente texto marcado en rojo:

```
<modelVersion>4.0.0</modelVersion>

<parent>
 <groupId>edu.upc.weather</groupId>
 <artifactId>upc-weather</artifactId>
 <version>1.0.0-SNAPSHOT</version>
 </parent>
```

Después realizaremos unas ligeras modificaciones en la siguiente información para cambiar algunos nombres. El texto siguiente:

```
<groupId>edu.upc.weather_client.impl</groupId>
<artifactId>upc-weather-cxfse-bundle</artifactId>
<packaging>bundle</packaging>
<version>1.0.0-SNAPSHOT</version>
<name>Apache ServiceMix :: CXF WSDL First OSGi Bundle</name>
```

Lo cambiaremos por el siguiente (marcado en rojo):

```
<groupId>edu.upc.weather</groupId>
<artifactId>upc-weather-cxfse-bundle</artifactId>
<packaging>bundle</packaging>
<version>1.0.0-SNAPSHOT</version>
<name>UPC_WEATHER_CXFSE_BUNDLE</name>
```

El siguiente cambio en este fichero es eliminar todo lo que está dentro del *tag* <build></build> y sustituirlo por lo que está en rojo. En esta sección, indicaremos que vamos a generar los proxy Java para los dos *wsdl* que tenemos. El wsdl conseguido de GlobalWeather (marcado en amarillo) y el *wsdl* creado para upc-weather (marcado en

verde). Un proxy Java son un conjunto de clases Java normalmente autogeneradas por un motor de servicios web que sirven para invocar un *web service*. Para generar estas clases, se utiliza la utilidad Apache CXF. Esto se realiza con la siguiente parte del código. En este código, también modificamos las librerías que se requieren en el contenedor OSGi (plugin org.apache.felix) debido a que con la creación del arquetipo no se han generado todas las librerías necesarias.

```
<build>
        <plugins>
            <plugin>
                <groupId>org.apache.maven.plugins</groupId>
                <artifactId>maven-compiler-plugin</artifactId>
                <version>2.0.2</version>
                <configuration>
                    <source>1.5</source>
                    <target>1.5</target>
                    <encoding>UTF-8</encoding>
                </configuration>
            </plugin>
            <plugin>
                <groupId>org.apache.cxf</groupId>
                <artifactId>cxf-codegen-plugin</artifactId>
                <version>2.6.3</version>
                <executions>
                    <execution>
                        <phase>generate-sources</phase>
                        <configuration>
                            <!--
                               Update source root to specify where generated source code
                  should
                                be placed
                            -->
                            <sourceRoot>${basedir}/src/generated/java</sourceRoot>
                            <defaultOptions>
                                <noAddressBinding>true</noAddressBinding>
                            </defaultOptions>
                            <wsdlOptions>
                                <wsdlOption>
                                    <!-- Update wsdl tag with appropriate wsdl file -->

                  <wsdl>${basedir}/src/main/resources/internet-weather.wsdl</wsdl>
                                </wsdlOption>
                            </wsdlOptions>
                        </configuration>
                        <goals>
                            <goal>wsdl2java</goal>
                        </goals>
                    </execution>
                </executions>
            </plugin>
            <plugin>
                <groupId>org.apache.cxf</groupId>
                <artifactId>cxf-codegen-plugin</artifactId>
                <version>2.6.3</version>
                <executions>
                    <execution>
                        <phase>generate-sources</phase>
                        <configuration>
                            <!--
                              Update source root to specify where generated source code
                              should be placed
                            -->
                            <sourceRoot>${basedir}/src/generated/java</sourceRoot>
                            <wsdlOptions>
                                <wsdlOption>
                                    <!-- Update wsdl tag with appropriate wsdl file -->

                                <wsdl>${basedir}/src/main/resources/wpc-weather.wsdl</wsdl>
                                    <extraargs>
                                        <extraarg>-verbose</extraarg>
                                    </extraargs>
```

```
                                </wsdlOption>
                            </wsdlOptions>
                        </configuration>
                        <goals>
                            <goal>wsdl2java</goal>
                        </goals>
                    </execution>
                </executions>
            </plugin>
            <plugin>
                <groupId>org.apache.felix</groupId>
                <artifactId>maven-bundle-plugin</artifactId>
                <configuration>
                    <instructions>
                        <Bundle-SymbolicName>${project.artifactId}</Bundle-SymbolicName>
                        <Import-Package>
                            javax.jws,
                            javax.jws.soap,
                            javax.wsdl,
                            javax.xml.bind,
                            javax.xml.bind.annotation,
                            javax.xml.namespace,
                            javax.xml.soap,
                            javax.xml.ws,
                            javax.xml.parsers,
                            org.w3c.dom,
                            META-INF.cxf,
                            org.apache.cxf.bus,
                            org.apache.cxf.bus.spring,
                            org.apache.cxf.bus.resource,
                            org.apache.cxf.configuration.spring,
                            org.apache.cxf.resource,
                            org.apache.cxf.feature,
                            org.apache.servicemix.cxfse,
                            org.springframework.beans.factory.config,
                            org.apache.servicemix.common.osgi
                        </Import-Package>
                        <Require-Bundle>org.apache.cxf.bundle</Require-Bundle>
                        <Export-Package>
                            edu.upc.weather.upc_weather_client._1_0,
                            edu.upc.weather.upc_weather_client._1
                        </Export-Package>
                        <!--
                            <DynamicImport-Package>*</DynamicImport-Package>
                        -->
                    </instructions>
                </configuration>
            </plugin>
        </plugins>
</build>
```

Ahora generaremos las clases java a partir de los WSDL. Para generar el código la primera vez o cada vez que se realicen cambios en los WSDL, se debe usar la instrucción Maven “mvn clean generate-sources” desde el directorio donde esté el fichero pom.xml.

Entonces, nos posicionamos en la línea de comandos de Windows en el directorio “F:\workspace\upc-weather-cxfse-bundle” que es donde se encuentra el pom.xml. Y ejecutamos dicha instrucción (Figura 143).

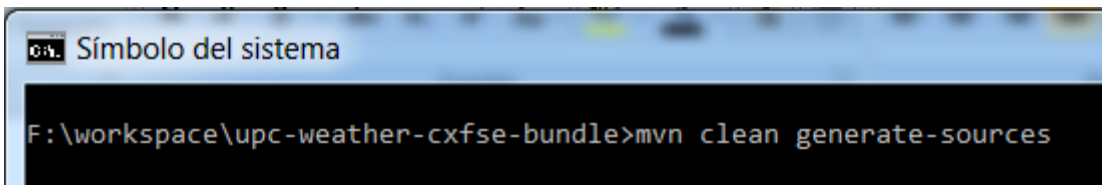

Fig. 143. Instrucción para generar los ficheros Java a partir de los wsdl.

Al finalizar debemos recibir el siguiente mensaje de que la creación de ficheros ha sido un éxito (Figura 144).

```
Loading FrontEnd jaxws ...
Loading DataBinding jaxb ...
wsdl2java -encoding UTF-8 -d F:\workspace\upc-weather-cxfse-bundle\src\generated\java -verbose
rc/main/resources/wsdl/upc-weather.wsdl
wsdl2java - Apache CXF 2.6.3

[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESSFUL
[INFO] ------------------------------------------------------------------------
[INFO] Total time: 1 minute 26 seconds
[INFO] Finished at: Sun Sep 08 18:33:32 CEST 2013
[INFO] Final Memory: 30M/73M
[INFO] ------------------------------------------------------------------------
F:\workspace\upc-weather-cxfse-bundle>
```

Fig. 144. Resultado exitoso de la generación de ficheros.

Y si observamos nuestro proyecto, veremos las clases generadas (Figura 145).

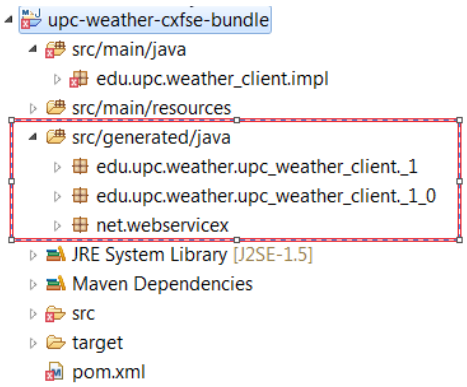

Fig. 145. Clases Java generadas por Maven a partir de los wsdl.

A continuación, crearemos los objetos en Spring (Beans) , que nos permitirán interaccionar con los servicios web a partir de las clases que acabamos de generar.

Beans.xml

Primero de todo borraremos todo lo contenido en este fichero entre el *tag* de más alto nivel <beans> … </beans>. Para ello editaremos el fichero que se encuentra en la ruta src/main/resources/META-INF/spring/beans.xml

El paso siguiente es incluir en el fichero el siguiente texto marcado en rojo:

```
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:cxfse="http://servicemix.apache.org/cxfse/1.0"
    xmlns:jaxws="http://cxf.apache.org/jaxws"
    xsi:schemaLocation="
```

```
        http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd
        http://servicemix.apache.org/cxfse/1.0
http://servicemix.apache.org/cxfse/1.0/servicemix-cxf-se.xsd
        http://cxf.apache.org/jaxws http://cxf.apache.org/schemas/jaxws.xsd
        http://www.springframework.org/schema/osgi
http://www.springframework.org/schema/osgi/spring-osgi.xsd">


<jaxws:client id="weatherProxy"
                  serviceClass="net.webservicex.GlobalWeatherSoap"
                  address="http://www.webservicex.net/globalweather.asmx" >
        <jaxws:features>
          <bean class="org.apache.cxf.feature.LoggingFeature"/>
        </jaxws:features>
    </jaxws:client>

    <cxfse:endpoint>
        <cxfse:pojo>
          <bean class="edu.upc.upc_weather_client.impl.UpcWeatherImpl">
              <property name="stub" ref="weatherProxy" />
          </bean>
        </cxfse:pojo>
    </cxfse:endpoint>

    <bean class="org.apache.servicemix.common.osgi.EndpointExporter" />
</beans>
```

A continuación explicamos cada parte. Con la siguiente línea estamos incluyendo el espacio de nombres para poder usar los tags <cxfse..> más adelante.

```
xmlns:cxfse="http://servicemix.apache.org/cxfse/1.0"
```

Con el siguiente código declararemos el bean con identificador “ weatherProxy” para consumir el Proxy Java GlobalWeather:

```
<jaxws:client id="weatherProxy"
          serviceClass="net.webservicex.GlobalWeatherSoap"
          address="http://www.webservicex.net/globalweather.asmx" >
 <jaxws:features>
   <bean class="org.apache.cxf.feature.LoggingFeature"/>
 </jaxws:features>
</jaxws:client>
```

Donde GlobalWeatherSoap es una de las clases generadas en el paso anterior que proviene del wsdl internet-weather.wsdl.
También crearemos el bean de nuestro endpoint interno expuesto al NMR para que desde el componente BC se pueda acceder a él:

```
<cxfse:endpoint>
        <cxfse:pojo>
          <bean class="edu.upc.weather_client.impl.UpcWeatherImpl">
              <property name="stub" ref="weatherProxy" />
          </bean>
        </cxfse:pojo>
</cxfse:endpoint>
```

Con el código anterior estamos dicienciendo que nuestra clase UpcWeatherImpl, será la implementación de nuestro servicio web upc-weather. Con “property name=”stub””, estamos diciendo que UpcWeatherImpl tendrá inyectado el bean de nombre “stub” que referenciará a “weatherProxy”.

Clase UpcWeatherImpl

La clase UpcWeatherImpl está en el paquete edu.upc.weather_client.impl, que como vemos contiene la clase de ejemplo que se había creado al crear el arquetipo Maven. Por lo tanto, borraremos esta clase como ya hemos explicado antes, y crearemos una nueva clase Java de nombre UpcWeatherImpl de la siguiente forma: botón derecho sobre el paquete edu.upc.weather_client.impl → New → Class (Figura 146).

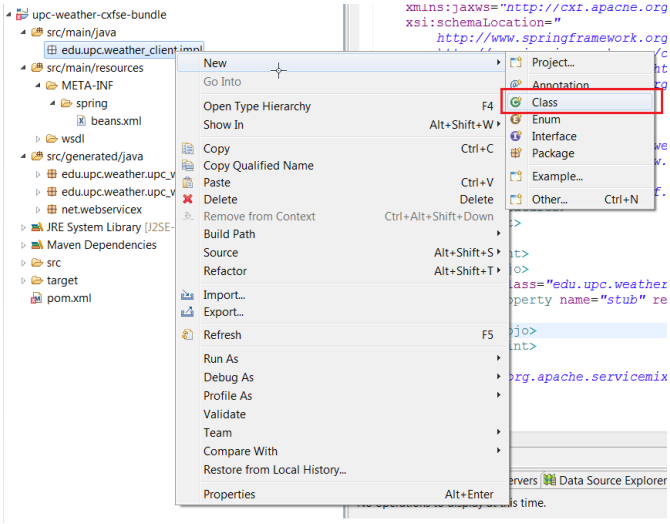

Fig. 146. Creación de una nueva clase Java. Paso 1.

En el siguiente paso introducimos la siguiente información tal y como muestra la Figura 147. Aceptamos haciendo clic en el botón Finish.

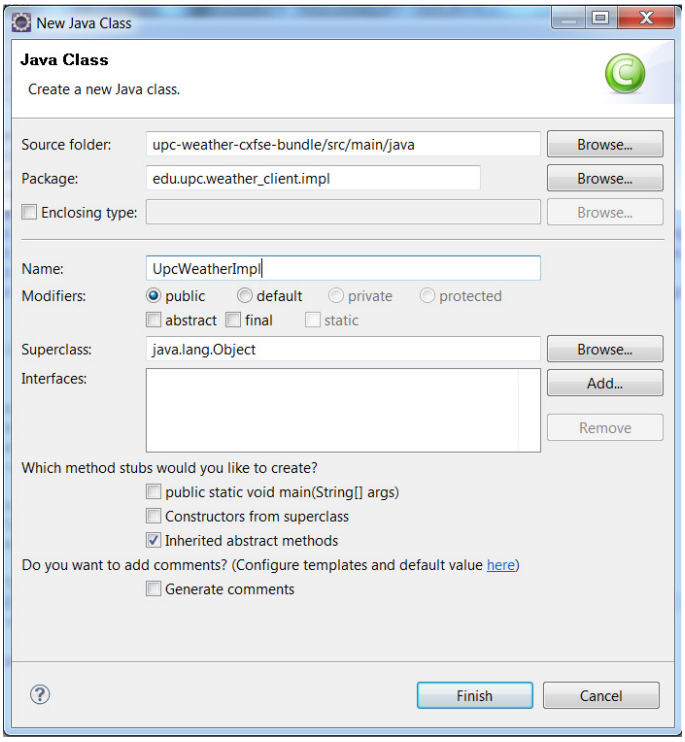

Fig. 147. Creación de una nueva clase Java. Paso 2.

Abrimos la clase, borramos su contenido y copiamos el siguiente código:

```
package edu.upc.weather_client.impl;

import javax.jws.WebParam;
import javax.jws.WebService;
import net.webservicex.GlobalWeatherSoap;
import edu.upc.weather.upc_weather_client._1.Weather;
import edu.upc.weather.upc_weather_client._1_0.UpcWeather;

//Declaracion del metodo de esta clase como implementación de un webservice
@WebService(serviceName = "upc-weather", targetNamespace = "http://weather.upc.edu/upc-
weather-client/1.0", endpointInterface =
"edu.upc.weather.upc_weather_client._1_0.UpcWeather")
public class UpcWeatherImpl implements UpcWeather {

//stub declarado en Spring que nos permitira acceder al servicio de tiempo publicado en
Internet GlobalWheater
    GlobalWeatherSoap stub;

    //metodo que se ejucutara cuando alguien llame al webservice
    //implementa el metodo getWeather definido en el upc-weather.wsdl
public Weather getWeather(@WebParam(name = "city", targetNamespace = "") String city,
@WebParam(name = "country", targetNamespace = "") String country) {

        //llamamos al servicio web de GlobalWeather con los parametros de ciudad y pais
        //el resultado es un String con todos los resultados
        String weather = stub.getWeather(city, country);

        //creamos un nuevo objeto Weather que sera nuestra respuesta
         Weather result = new Weather();

         //aqui recuperamos los valores que nos interesan de la respuesta enviada por
GlobalWeather
                String location =
        weather.substring(weather.indexOf("<Location>")+"<Location>".length(),
        weather.indexOf("</Location>"));
```

```
                String wind =
            weather.substring(weather.indexOf("<Wind>")+"<Wind>".length(),
            weather.indexOf("</Wind>"));
                String visibility =
            weather.substring(weather.indexOf("<Visibility>")+"<Visibility>".length(),
            weather.indexOf("</Visibility>"));
                String skyConditions =
            weather.substring(weather.indexOf("<SkyConditions>")+"<SkyConditions>".length(),
            weather.indexOf("</SkyConditions>"));
                String temperature =
            weather.substring(weather.indexOf("<Temperature>")+"<Temperature>".length(),
            weather.indexOf("</Temperature>"));
                String relativeHumidity =
            weather.substring(weather.indexOf("<RelativeHumidity>")+"<RelativeHumidity>".lengt
            h(), weather.indexOf("</RelativeHumidity>"));
                String pressure =
            weather.substring(weather.indexOf("<Pressure>")+"<Pressure>".length(),
            weather.indexOf("</Pressure>"));
                String status =
            weather.substring(weather.indexOf("<Status>")+"<Status>".length(),
            weather.indexOf("</Status>"));

            //Añadimos a nuestro objeto de respuesta la informacion anterior
            result.setLocation(location);
            result.setWind(wind);
            result.setVisibility(visibility);
            result.setSkyConditions(skyConditions);
            result.setTemperature(temperature);
            result.setRelativeHumidity(relativeHumidity);
            result.setPressure(pressure);
            result.setStatus(status);

            //el metodo devuelve el objeto con el resultado
            return result;
        }

        public GlobalWeatherSoap getStub() {
            return stub;
        }

        public void setStub(GlobalWeatherSoap stub) {
            this.stub = stub;
        }
    }
```

A continuación detallamos cada paso:

En la clase UpcWeatherImpl indicamos mediante Annotations (anotaciones Java, van precedidas del símbolo @) que nuestra clase hará referencia a un servicio web cuya interfaz es la que hemos generado en el fichero pom.xml y nuestro nombre de servicio para ser referenciado será upc-weather. En la variable stub tendremos la referencia al bean de GlobalWeatherSoap creado en el fichero beans.xml.

```
//Declaracion del metodo de esta clase como implementación de un webservice

@WebService(serviceName = "upc-weather", targetNamespace = "http://weather.upc.edu/upc-
weather-client/1.0", endpointInterface =
"edu.upc.weather.upc_weather_client._1_0.UpcWeather")
public class UpcWeatherImpl implements UpcWeather {

//stub declarado en Spring que nos permitira acceder al servicio de tiempo publicado en
Internet GlobalWheater

    GlobalWeatherSoap stub;

...
```

Implementamos el método getWeather que estará definido en la interfaz UpcWeather, con parámetros de entrada la ciudad y páis:

```
//metodo que se ejucutara cuando alguien llame al webservice
//implementa el metodo getWeather definido en el upc-weather.wsdl

public Weather getWeather(@WebParam(name = "city", targetNamespace = "") String city,
@WebParam(name = "country", targetNamespace = "") String country) throws ExceptionFault{
```

Ahora podemos llamar al webservice GlobalWeather a partir de la variable stub y recuperar en un objeto String el resultado.

```
//llamamos al servicio web de GlobalWeather con los parametros de ciudad y pais
//el resultado es un String con todos los resultados

String weather = stub.getWeather(city, country);
```

Y realizar nuestras conversiones:

```
//aqui recuperamos los valores que nos interesan de la respuesta enviada por GlobalWeather

Weather result = new Weather();
String location = weather.substring(weather.indexOf("<Location>")+"<Location>".length(),
weather.indexOf("</Location>"));
String wind = weather.substring(weather.indexOf("<Wind>")+"<Wind>".length(),
weather.indexOf("</Wind>"));
…
```

Por último, añadimos a nuestro objeto de respuesta la información que hemos e laborado en el paso anterior y devolvemos el objeto de resultado:

```
//Añadimos a nuestro objeto de respuesta la informacion anterior

result.setLocation(location);
result.setWind(wind);
result.setVisibility(visibility);
result.setSkyConditions(skyConditions);
result.setTemperature(temperature);
result.setRelativeHumidity(relativeHumidity);
result.setPressure(pressure);
result.setStatus(status);

//el metodo devuelve el objeto con el resultado
        return result;
```

Una vez visto el componente SE al completo, vamos a crear el componente BC.

13.3.4 Bundle upc-weather-cxfbc-bundle

El primer paso, será eliminar los paquetes src/main/java y src/main/java/resources/wsdl debido a que este componente no usará lógica de negocio, lo que haremos será publicar al exterior del ESB nuestro servicio web. Una vez eliminados, debemos incluir nuestro fichero upc-weather.wsdl en el directorio src/main/resources/. Más adelante veremos el porqué. Así después de estos dos pasos, nuestra estructura de proyecto será la siguiente (Figura 148):

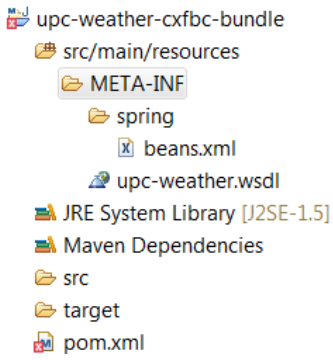

Fig. 148. Estructura de directorios del componente BC.

Editamos el fichero beans.xml y borramos el contenido que viene por defecto entre los *tags* <beans …..> </beans>

Ahora declaramos entre los tags <beans ….></beans> el siguiente componente:

```
<bean id="logging" class="org.apache.cxf.interceptor.LoggingInInterceptor"/>

<cxfbc:consumer wsdl="classpath:upc-weather.wsdl"
             targetService="upc-weather:upc-weather"
             targetInterface="upc-weather:upc-weather">
 <cxfbc:features>
     <bean class="org.apache.cxf.feature.LoggingFeature"/>
 </cxfbc:features>
 </cxfbc:consumer>

<bean class="org.apache.servicemix.common.osgi.EndpointExporter" />
```

Este *endpoint* "consumer" es consumidor pero respecto del NMR, es decir consumirá un servicio disponible en el bus, el que nosotros hemos publicado con identificador upc-weather y referenciamos con las propiedades targetService y targetInterface. También hay que referenciar qué wsdl estamos utilizando (con la propiedad wsdl) para que conozca los datos definidos, por este motivo hemos incluido el wsdl en este proyecto. Este *endpoint* publicará al exterior el servicio upc-weather mediante el protocolo SOAP/HTTP.

Nótese que referenciamos el servicio como upc-weather:upc-weather. El primer upc-weather hace referencia al identificador de *namespace* y el segundo upc-weather es el nombre del servicio. Los *namespace* están declarados en la cabecera del fichero beans.xml, se marca en rojo los añadidos:

```
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:jaxws="http://cxf.apache.org/jaxws"
    xmlns:cxfbc="http://servicemix.apache.org/cxfbc/1.0"
xmlns:upc-weather="http://weather.upc.edu/upc-weather-client/1.0"
    xsi:schemaLocation="
        http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd
        http://cxf.apache.org/jaxws http://cxf.apache.org/schemas/jaxws.xsd
        http://www.springframework.org/schema/osgi
http://www.springframework.org/schema/osgi/spring-osgi.xsd
```

```
        http://servicemix.apache.org/cxfbc/1.0
http://servicemix.apache.org/cxfbc/1.0/servicemix-cxf-bc.xsd">
```

Este Namespace (xmlns:upc-weather) debe coincidir con el Namespace declarado en las *annotations* de la clase UpcWeatherImpl definidas en el componente SE y que también está definido en el WSDL upc-weather.wsdl.

Pom.xml

Editamos este fichero para realizar ciertas modificaciones, en primer lugar, como hemos hecho con el componente SE, añadiremos el siguiente texto marcado en rojo:

```
<modelVersion>4.0.0</modelVersion>
<parent>
 <groupId>edu.upc.weather</groupId>
 <artifactId>upc-weather</artifactId>
 <version>1.0.0-SNAPSHOT</version>
</parent>
```

Cambiaremos algunos textos de la descripción del componente marcados en rojo:

```
<groupId>edu.upc.weather</groupId>
<artifactId>upc-weather-cxfbc-bundle</artifactId>
<packaging>bundle</packaging>
<version>1.0.0-SNAPSHOT</version>
<name>UPC_WEATHER_CXFBC_BUNDLE</name>
```

Por último, reemplazamos la sección <plugins> dentro de <build> completamente, para dejar solamente el plugin referente a las librerías del contenedor OSGi (marcado en rojo):

```
<build>
 <plugins>
     <plugin>
         <groupId>org.apache.felix</groupId>
         <artifactId>maven-bundle-plugin</artifactId>
         <configuration>
             <instructions>
                 <Bundle-SymbolicName>${project.artifactId}</Bundle-SymbolicName>
                 <Import-Package>
                     javax.jws,
                     javax.wsdl,
                     javax.xml.bind,
                     javax.xml.bind.annotation,
                     javax.xml.namespace,
                     javax.xml.soap,
                     javax.xml.ws,
                     META-INF.cxf,
                     org.apache.cxf.bus,
                     org.apache.cxf.bus.spring,
                     org.apache.cxf.bus.resource,
                     org.apache.cxf.configuration.spring,
                     org.apache.cxf.resource,
                     org.apache.cxf.feature,
                     org.apache.cxf.interceptor,
                     org.apache.servicemix.cxfbc,
                     org.apache.servicemix.http,
                     org.apache.servicemix.http.endpoints,
                     org.springframework.beans.factory.config,
                     org.apache.servicemix.common.osgi
                 </Import-Package>
                 <Require-Bundle>org.apache.cxf.bundle</Require-Bundle>
             </instructions>
         </configuration>
```

```
    </plugin>
 </plugins>
</build>
```

Ya hemos terminado nuestro desarrollo de la aplicación. En los próximos puntos veremos cómo desplegarla en el ServiceMix.

13.3.5 Compilación de la apliación

Para compilar la aplicación ejecutamos la instrucción "mvn clean install" desde la línea de comandos y estando dentro del directorio padre (F:\workspace) donde están los componentes BC y SE (Figura 149).

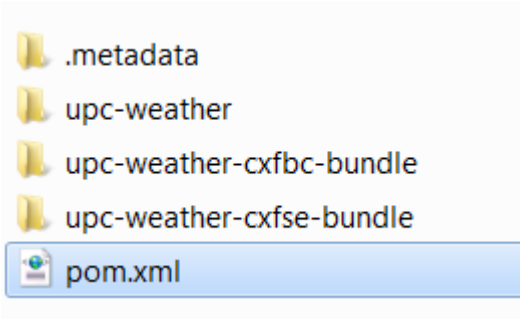

Fig. 149. Directorio padre del proyecto

En este directorio tendremos el fichero pom.xml de la aplicación padre, que es el que se leerá cuando se realice la instrucción clean install (internamente install realiza la tarea compile para compilar la aplicación). La tarea install instalará en el repositorio local Maven de nuestro ordenador los componentes BC y SE.

Para compilar e instalar pues, ejecutamos la instrucción de la Figura 150.

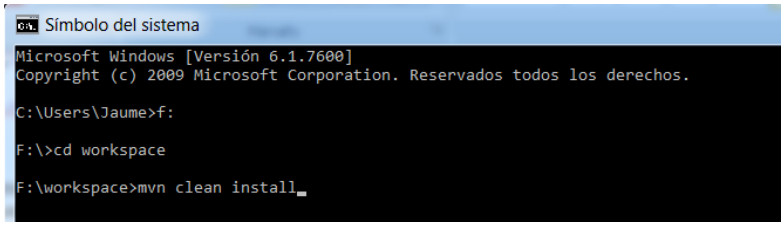

Fig. 150. Compilación de la aplicación con Maven.

Una vez ejecutada la instrucción, debemos observar que el proceso se ha realizado correctamente como se muestra en Figura 151. En este momento, ya tenemos la aplicación instalada en el repositorio Maven ya disponible para ser desplegada al ServiceMix.

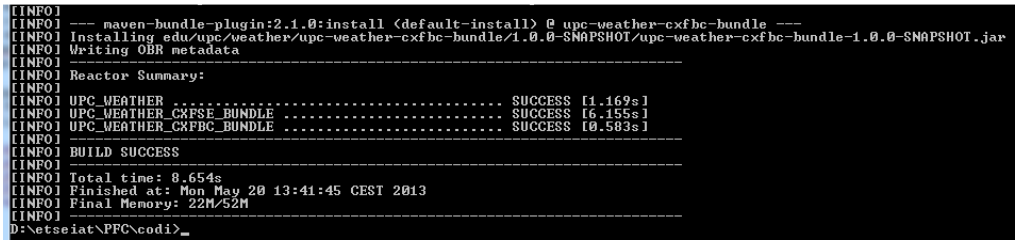

Fig. 151. Mensaje de Maven indicando que se ha compilado la aplicación correctamente.

13.3.6 Despliegue de la aplicación al ServiceMix

Una vez tenemos los *bundles* de la aplicación ya instalados en el repositorio Maven, podemos desplegarlos en el ServiceMix. Para ello utilizaremos la conectividad con Maven y el uso de *features*. Ya hemos creado la *feature*, que se encuentra en:

F:\workspace\upc-weather\upc-weather-feature-1.0.0-SNAPSHOT.xml

Para instalar la feature, utilizaremos el sistema hot deploy. Aunque esto significa que se pueden instalar en caliente los *bundles*, es decir sin parar el servidor**, se recomienda reniciar Karaf ya que se ha observado que este sistema no termina de funcionar bien**. Para ello, una vez arrancado el Karaf, copiaremos el fichero upc-weather-feature-1.0.0-SNAPSHOT.xml al directorio “F:\ apache-servicemix-4.3.1-fuse-01-15\deploy”.

Para eliminar la feature simplemente debemos eliminar este fichero de dentro de la carpeta deploy.

Una vez copiado el fichero de la *feature*, podemos comprobar que se han instalado correctamente los bundles des de la consola Karaf (o desde la consola web).
Ejecutamos:

```
karaf@root> osgi:list
```

Debemos observar como en la Figura 152 que el estado de los bundles es de STARTED.

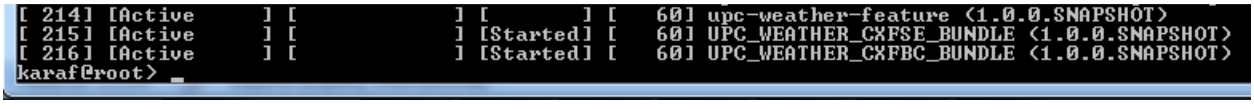

Fig. 152. Bundles de la aplicación levantados correctamente.

13.3.7 Test de la aplicación

Para hacer el test de nuestro proyecto ServiceMix utilizaremos la aplicación SOAP UI[7]. Esta aplicación nos permite realizar peticiones SOAP/HTTP para probar *web services*.

Una vez instalada la aplicación, un nuevo proyecto SOAPUI a partir de nuestro wsdl upc-weather.wsdl. Para ello desde la aplicación SOAP UI hacemos File -> New soapUI Project y cargamos en Initial WSDL/WADL nuestro str-weather.wsdl que se encuentra en F:\workspace\upc-weather-cxfse-bundle\src\main\resources\upc-weather.wsdl tal y como se muestra en la Figura 153.

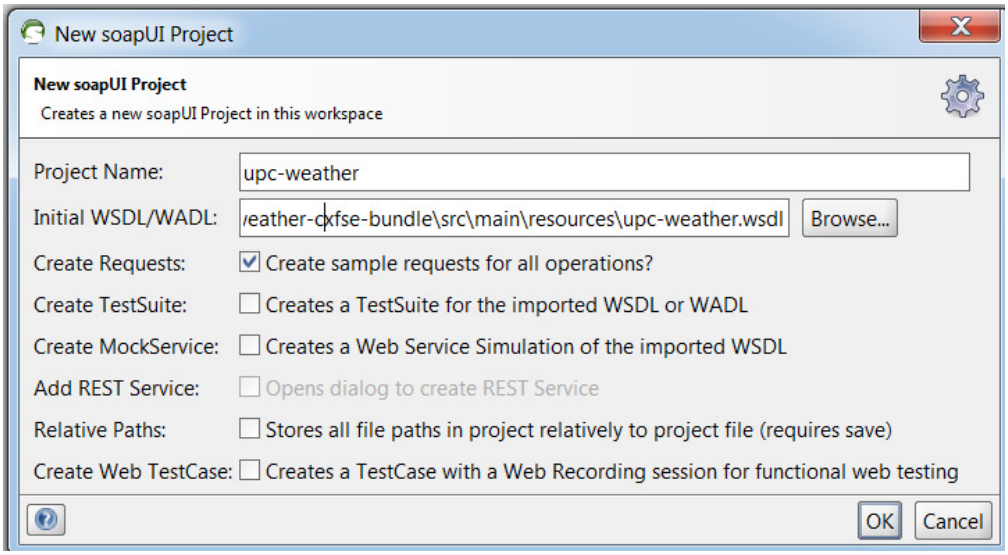

Fig. 153. Nuevo proyecto soapUI.

Hacemos clic en Ok. Una vez cargado el proyecto, en la parte de la izquierda abrimos el nodo upc-weather->WeatherWSServiceSoapBinding->getWeather->Request 1 de la misma forma que se muestra en la Figura 154.

[7] http://sourceforge.net/projects/soapui/files/soapui/4.5.1/soapUI-x32-4.5.1.exe/download

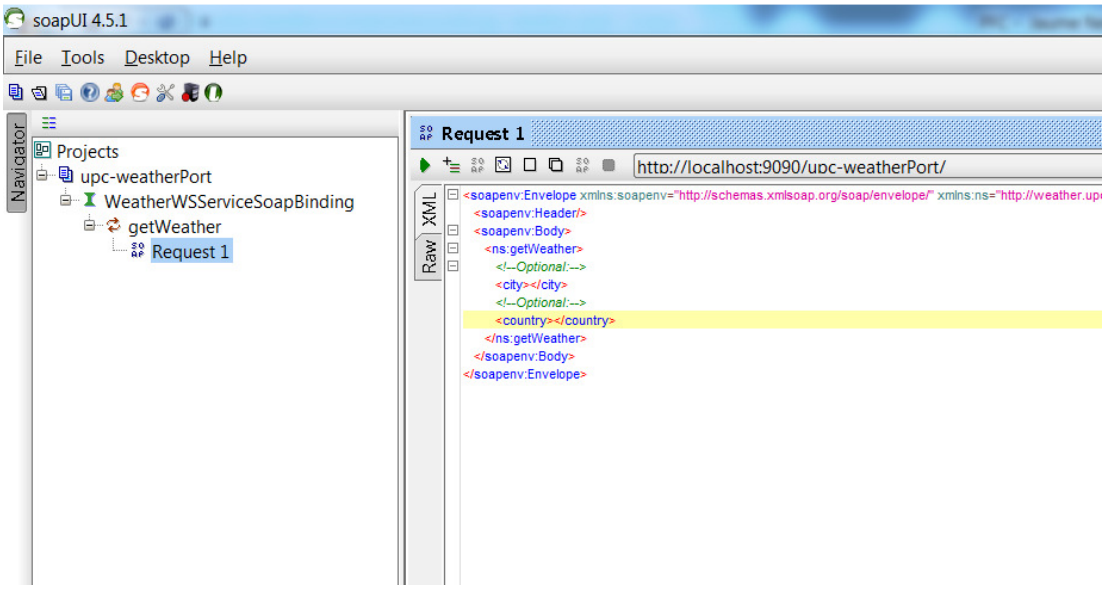

Fig. 154. Petición SOAP a nuestro endpoint del ServiceMix.

Ahora rellenamos la petición con un valor para ciudad y país, por ejemplo Barcelona/Spain (Figura 155).

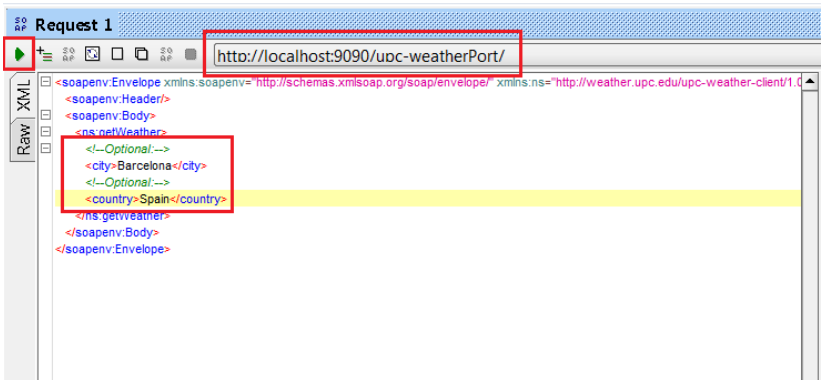

Fig. 155. Petición SOAP preparada para ejecutarse.

En la parte superior de la Figura 155 se observa el *endpoint* que habíamos especificado en el wsdl http://localhost:9090/upc-weatherPort/ . Para ejecutar la petición debemos hacer clic en el botón Run que se observa en la parte superior izquierda de la Figura 155.

Una vez ejecutamos la petición, obtenemos la respuesta (Figura 156):

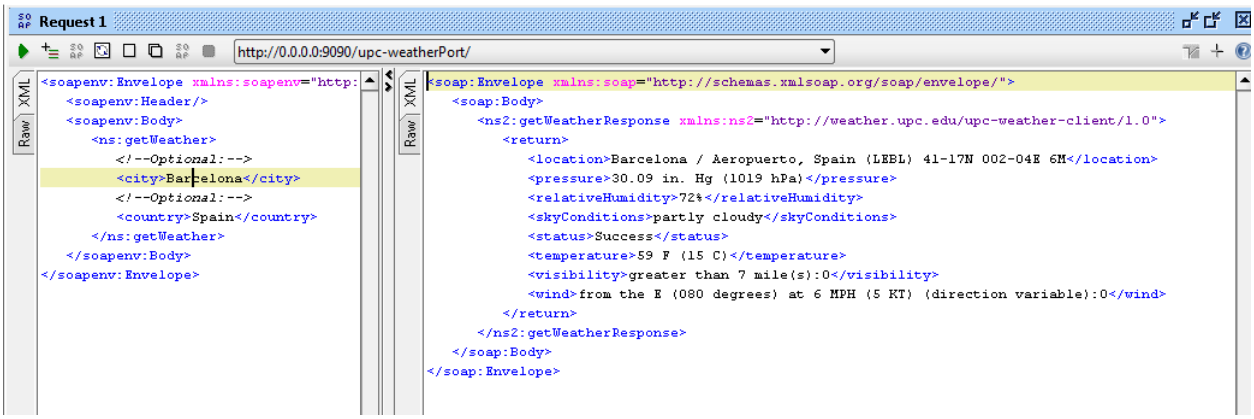

Fig. 156. Resultado de la ejecución del servicio web str-weather.

Podemos observar las trazas del fichero log del ServiceMix en “F:\apache-servicemix-4.3.1-fuse-01-15\data\log\servicemix.log” para observar las peticiones que se han realizado. En caso de error es útil analizar este log para ver dónde ha habido algún problema.

14. CONLUSIONES

SOA es una arquitectura de software que aporta muchos beneficios, permite un desacoplamiento entre aplicaciones y una centralización de servicios que hace que los sistemas de información sean mantenibles y escalables. El uso de servicios implementados como *web services* dentro de SOA permiten la interoperabilidad con otros sistemas, un menor esfuerzo en los siguientes desarrollos y un mayor reaprovechamiento de las funcionalidades ya ofrecidas por los servicios existentes gracias a la reutilización.

Ahora bien, SOA, tiene sus fuerzas y sus debilidades y debemos saber cuándo nos conviene desarrollar sistemas bajo sus directrices. SOA es muy útil en entornos empresariales, donde la comunicación a priori no implica un alto volumen de datos y dónde se requiere un gran control y auditoría de las comunicaciones. Se debe tener en cuenta también, la complejidad de la implantación de un sistema completo bajo el paradigma de SOA. Se debe conocer bien el negocio para una correcta identificación de los posibles servicios, y como separar bien las funcionalidades para no incurrir en redundancias.

Una buena (incluso básica) estrategia para implementar SOA pasa por el uso de un ESB, que nos permita mantener un control centralizado de las comunicaciones y la mediación entre todos los sistemas conectados a él. También hemos visto como existen en el mercado componentes ESB *opensource* que proporcionan una arquitectura SOA basada en estándares abiertos que carecen de coste de adquisición, al no ser software propietario de pago.

15. BIBLIOGRAFÍA

[1] Carlos Billy Reynoso, "Introducción a la arquitectura de software", 2004. http://carlosreynoso.com.ar/archivos/arquitectura/Introduccion.PDF

[2] César de la Torre, Roberto González, "Arquitectura SOA con tecnología Microsoft", Krassis Press, 2008.

[3] Wikipedia, http://es.wikipedia.org/wiki/Programaci%C3%B3n_por_capas

[4] Wikipedia, http://es.wikipedia.org/wiki/Ingenier%C3%ADa_de_software_basada_en_componentes

[5] Carlos Billy Reynoso, "Arquitectura Orientada a Servicios".

[6] Jason Bloomerg, "The role of the service-oriented architect", The Rational edge e-zine, 2003.

[7] Thomas Erl, "Service-Oriented Architecture (SOA): Concepts, Technology, and Design", Prentice Hall, 2005.

[8] Thomas Erl, "SOA Principles of Service Design", Prentice Hall, 2007. http://serviceorientation.com/index.php/serviceorientation/index

[9] "ARQ-RFC-01-Pautas y recomendaciones para SOA v.091", BPS, 2006.

[10] W3C, http://www.w3.org/TR/ws-gloss/

[11] Wikipedia, http://es.wikipedia.org/wiki/Proceso_de_negocio

[12] W3C, http://www.w3.org/

[13] Observatiorio tecnológico (Gobierno de España), http://recursostic.educacion.es/observatorio/web/ca/software/programacion/675-xml

[14] Wikpedia, http://en.wikipedia.org/wiki/Open_Document_Architecture

[15] W3C, http://www.w3schools.com/xml/xml_namespaces.asp

[16] Blog Gramáticas formales, http://gramaticasformales.wordpress.com/category/web-services/

[17] Desarrollo web, http://www.desarrolloweb.com/articulos/1557.php

[18] Wikipedia, http://es.wikipedia.org/wiki/XML_Schema

[19] W3C, http://www.w3.org/TR/xmlschema11-2/#built-in-primitive-datatypes

[20] Christophe Demarey, Gael Harbonnier, Romain Rouvoy, Philippe Merle, "Benchmarking the round-trip latency of various Java-Based middelware platforms", 2005. http://studia.complexica.net/Art/RI040102.pdf

[21] Wikipedia, http://en.wikipedia.org/wiki/Service-oriented_architecture

[22] Blog Pensando en SOA, http://pensandoensoa.com/2010/02/

[23] IBM, http://public.dhe.ibm.com/software/dw/webservices/ws-soa-whitepaper.pdf

[24] Wikipedia, http://es.wikipedia.org/wiki/Stakeholder

[25] R. Edward Freeman, "Strategic Management: A Stakeholder Approach", Cambridge University Press, 1984.

[26] Michael Rosen, Boris Lublinsky, Kevin T. Smith, Marc J. Balcer, "Applied SOA: Service-Oriented Architecture and Design Strategies",Wiley, 2008.
http://www.infoq.com/resource/articles/applied-soa/en/resources/Applied-SOA.pdf

[27] Slideshare, http://es.slideshare.net/raghamadabhushi/soa-session

[28] Blog Oposiciones TIC http://oposicionestic.blogspot.com.es/2012/08/arquitectura-soa-orientada-servicios.html

[29] Dirk Krafzig, Karl Banke, Dirk Slama, "Enterprise SOA", Pertinence Hall, 2005.

[30] Anurag Goel, "Enterprise Integration EAI vs. SOA vs. ESB".
http://ggatz.com/images/Enterprise_20Integration_20-_20SOA_20vs_20EAI_20vs_20ESB.pdf

[31] Rob Brooks-Bilson, http://rob.brooks-bilson.com/index.cfm/ESB

[32] Universidad Carlos III, http://ocw.uc3m.es/ingenieria-telematica/tecnologias-de-distribucion-de-contenidos/transparencias_tdc/enterprise-service-bus-esb.pdf

[33] Phil Bianco, Rick Kotermanski, Paulo Merson, "Evaluating a Service-Oriented Architecture", 2007. http://www.sei.cmu.edu/reports/07tr015.pdf

[34] Fiorano, http://www.fiorano.com/products/ESB-enterprise-service-bus/Fiorano-ESB-enterprise-service-bus.php

[35] Oracle, http://www.oracle.com/technetwork/developer-tools/jdev/index-084237.html

[36] Lucia Kapová, "Introduction to Enterprise Service Bus", Charles University Prague.
http://d3s.mff.cuni.cz/research/seminar/download/2005-11-01-Kapova-ESB.pdf

[37] Adictos al Trabajo, http://www.adictosaltrabajo.com/tutoriales/tutoriales.php?pagina=BPEL

[38] Forrester, http://www.forrester.com/home/

[39] Ken Vollmer, "The Forrester Wave™: Enterprise Service Bus, Q2 2011".
http://objectstore.ru/pdf/2011_forrester-esb-wave.pdf

[40] FuseSource, http://fusesource.com/products/enterprise-servicemix/

[41] Apache ServiceMix, http://servicemix.apache.org/

[42] Mulesoft, http://www.mulesoft.org/

[43] WSO2, http://wso2.com/products/enterprise-service-bus

[44] JBoss, http://www.jboss.org/jbossesb

[45] Google, http://www.google.es/trends/explore#q=ServiceMix%2C%20mule%20ESB%2C%20Jboss%20ESB%2C%20WSO2%20ESB&cmpt=q

[46] Apache Software Fundation, http://www.apache.org/

[47] Blog Justo Aguilar, http://blog.justoaguilar.com/2009/08/que-es-osgi-y-para-que-sirve/

[48] FuseSource, http://fusesource.com/docs/esb/4.4/esb_prod_intro/ESBGetStartedArchIntro.html

[49] Novell, http://www.novell.com/documentation/extend52/Docs/help/MP/jms/tutorial/pointToPoint-1.htm

[50] Apache Camel, http://camel.apache.org/eip.html

[51] Wiki Apache, https://cwiki.apache.org/confluence/display/SM/5.+JBI

[52] Centro de difusión de tecnologías ETSIT-UPM, http://www.ceditec.etsit.upm.es/index.php?option=com_content&view=article&id=21808&Itemid=1439&lang=es

[53] Wikipedia, http://es.wikipedia.org/wiki/Computaci%C3%B3n_en_la_nube

[54] Blog BCN Binary. Tecnología e Informática, http://bcnbinaryblog.com/cloud-computing-infraestructura-como-servicio-iaas/

[55] Google, http://www.google.com/intl/es_es/enterprise/apps/business/

[56] Google Cloud, https://cloud.google.com/products/app-engine

[57] Amazon, http://aws.amazon.com/es/