UNIVERSIDAD NACIONAL DEL CENTRO
DE LA PROVINCIA DE BUENOS AIRES
FACULTAD DE CIENCIAS EXACTAS

EasySOC: Una herramienta para facilitar el desarrollo y mantenimiento de aplicaciones orientadas a servicios

por
Mariano Fischer y Matías Martinez

Dr. Alejandro Zunino
Director

Ing. Marco Crasso
Co-Director

Tandil, Noviembre de 2008

Resumen

Computación Orientada a Servicios (SOC) es un paradigma de computación con gran auge en los últimos años y que está siendo ampliamente reconocido como la tendencia a seguir en cuanto al desarrollo de aplicaciones distribuidas. A pesar de los grandes beneficios proporcionados por este paradigma, el desarrollo de aplicaciones orientadas a servicios presenta una problemática en cuanto a tiempos y costos de desarrollo en dos fases del ciclo de vida de la aplicación. Durante la etapa de implementación los desarrolladores deben invertir mucho tiempo en descubrir los servicios que llevarán a cabo cierta funcionalidad a tercerizar. Además, como el código de la aplicación se encuentra ligado a algunos servicios en particular, se torna difícil la modificabilidad y testing durante la fase de mantenimiento, en caso de necesitar cambiar de proveedor de servicios.

Debido a esto, en el presente trabajo se propone EasySOC como un enfoque que permite el desarrollo de aplicaciones orientadas a servicios. Este enfoque propone técnicas de búsqueda e incorporación de servicios en las aplicaciones con altos niveles de modificabilidad, que conllevan a aliviar los costos en las etapas de desarrollo y mantenimiento de aplicaciones orientadas a servicios.

A su vez, EasySOCPlugin es una herramienta que materializa en enfoque EasySOC para la plataforma Java, la cual ha sido sujeta a distintas mediciones en escenarios de pruebas, a fin de verificar los beneficios del uso de EasySOC en el desarrollo de aplicaciones orientadas a servicios.

Palabras clave: Computación orientada a servicios, tercerización de servicios, inyección de dependecias, servicios Web, adaptación de interfaces.

Índice general

Índice de figuras

Índice de cuadros

Glosario

AOP Aspect Oriented Programing

API Application programmatic Interface

CORBA Common Object Request Broker Arquitecture

DB Database

DI Dependency Inyection

FTP File Transfer Protocol

HTML HyperText Markup Language

HTTP Hyper Text Transfer Protocol

IDE Integrated Development Environment

IR Information Retrieval

JEE Java Enterprise Edition

ORB Object Request Broker

POJO Plain Old Java Objec

RPC Remote Procedure Call

SMTP Simple Mail Transfer Protocol

SOA Service Oriented Architecture

SOAP Simple Object Access Protocol

SOC Service Oriented Computing

SW Servicio Web

SWs Servicios Web

TFIDF Term Frequency Inverse Document Frequency

UDDI Universal Description, Discovery and Integration

UML Unified Modeling Language

URL Uniform Resource Locator

VSM Vector Space Model

WSDL Web Service Definition Language

WSIF Web Service Invocation Framework

XML eXtensible Markup Language

XSD XML Schema Definition

CAPÍTULO 1

Introducción

El objetivo de este capítulo es dar una breve descripción del enfoque propuesto en este trabajo, utilizado para el desarrollo de aplicaciones orientadas a servicios. Además, se remarcan las principales falencias que poseen los enfoques existentes para el desarrollo de aplicaciones de este tipo, las cuales sirvieron como motivación para la creación del enfoque presentando en este trabajo.

Este capítulo está organizado como sigue: en la sección 1.1, se describen los problemas que motivaron la generación de esta tesis; en la sección 1.2, se describe la solución propuesta en este trabajo; por último, en la sección 1.3, se indica cómo ha sido organizado el resto del trabajo.

1.1. Motivación

Computación Orientada a Servicios (SOC) es un paradigma de computación cuyo principal objetivo es el desarrollo de aplicaciones distribuidas en ambientes heterogéneos. Bajo dicho paradigma, los sistemas distribuidos son construidos ensamblando, o componiendo, funcionalidad existente, la cual es publicada a través de una red y accedida mediante protocolos específicos. Esta funcionalidad se denomina servicio. Desde el punto de vista de la ingeniería de software, SOC es un paradigma interesante para el desarrollo de aplicaciones, ya que promueve fuertemente la reusabilidad del software de manera desacoplada.

Una arquitectura orientada a servicios, se encuentra compuesta por tres actores principales: un proveedor de servicios, un consumidor de servicios y un registro de servicios. Estos tres actores interactúan de la siguiente manera: el proveedor de servicios publica en un registro la descripción del servicio que él provee. Por otro lado, los consumidores buscan en un registro los servicios que cumplen con sus necesidades. Una vez que un servicio es seleccionado, éste será invocado desde la aplicación cliente. Mayoritariamente, las aplicaciones orientadas a servicios se materializan usando la tecnología de servicios Web. Un servicio Web es un

programa con una interfaz bien definida la cual puede ser localizada, publicada e invocada utilizando la infraestructura estándar de la Web.

A pesar de las ventajas que SOC provee, tales como el bajo grado de acople entre consumidor/proveedor de un determinado servicio y mayor reusabilidad de componentes, este paradigma posee la desventaja de aumentar los costos de dos etapas del ciclo de vida de un sistema de software: implementación y mantenimiento. En primer lugar, buscar servicios publicados en un registro UDDI (Universal Description, Discovery, and Integration) [26], la materialización de un registro según servicios Web, requiere invertir mucho tiempo. Esto, impacta directamente en los costos de la fase de implementación, porque SOC reemplaza el desarrollo de piezas específicas por el descubrimiento y contratación de las mismas. En segundo lugar, el resultado de introducir servicios externos a una aplicación es código fuente donde la lógica del negocio está "contaminada" con aspectos no funcionales, tales como localización, comunicación de datos sobre la red, etc. Esto no es un atributo de calidad deseable, dado que produce sistemas difíciles de entender, mantener y extender. Además, los *frameworks* actuales para invocar servicios, por ejemplo WSIF, producen código fuente subordinado a un determinado proveedor de servicios. Por ejemplo, si un proveedor utiliza un archivo de texto separado por comas para informar acerca de sus productos en oferta, mientras que otro proveedor informa de sus ofertas mediante un arreglo de número de productos, claramente el código necesario para consumir el servicio de uno u otro será distinto. En consecuencia, los cambios en las interfaces de los servicios externos o el reemplazo de los mismos por nuevos proveedores, requieren regenerar el código necesario para realizar la invocación a los mismos, lo cual propaga cambios por las partes "contaminadas" de la aplicación. En otras palabras, estos enfoques impactan directamente sobre los costos de la etapa del mantenimiento del software.

1.2. Solución propuesta

El objetivo del presente trabajo es presentar un enfoque que facilite el desarrollo de aplicaciones orientas a servicios, minimizando los costos de desarrollo y mantenimiento de este tipo de aplicaciones. Asistiendo a los desarrolladores, se busca mejorar los procesos de desarrollo de este tipo de aplicaciones. Además, se propone una herramienta, la cual materializa dicho enfoque.

El objetivo es asistir a los desarrolladores de aplicaciones orientadas a servicios mediante una serie de procesos semi-automáticos para llevar a cabo las siguientes tareas:

- Búsqueda de servicios que cumplan con las funcionalidades que se desean incorporar a la aplicación.
- Adaptación del servicio externo a la aplicación.
- Incorporación del servicio Web dentro de la aplicación.

En la Figura 1.1 se puede observar un diagrama general del enfoque propuesto, donde el flujo comienza por la búsqueda de servicios externos que cumplan con cierta funcionalidad requerida en la aplicación, continúa con la adaptación de los servicios encontrados y finalmente termina con la incorporación de éstos dentro de la aplicación en desarrollo.

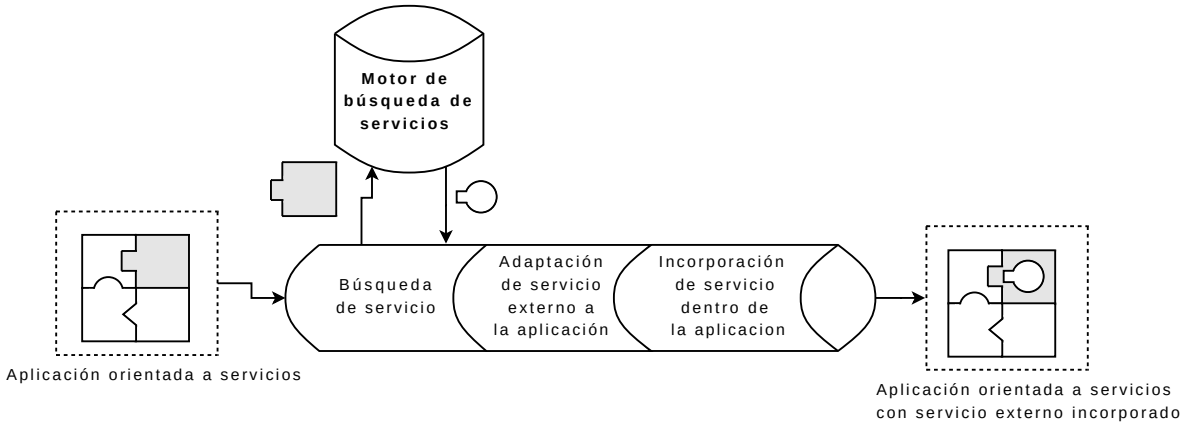


Figura 1.1: Vista global del enfoque EeasSOC.

1.2.1. Búsqueda de servicios

Durante el desarrollo de aplicaciones orientadas a servicios, el/la desarrollador/a decide cuáles componentes implementará y cuales de ellos serán tercerizados mediante servicios Web. Es decir, la implementación de un componente se delega a un servicio Web de terceros.

El enfoque propone asistir al desarrollador en el proceso de búsqueda de servicios Web de terceros, con el fin de incorporarlos dentro de la aplicación bajo desarrollo, y así proveer una implementación para los componentes no implementados por el/la desarrollador/a. Para ello, se define un proceso semi-automático de generación de *queries*[1] para la búsqueda de servicios Web. El proceso extrae información útil desde los distintos componentes de la aplicación, como así también desde la documentación contenida en sus códigos fuentes.

Luego, a partir del *query* generado, se realiza una búsqueda de servicios Web candidatos con el fin que el desarrollador/a pueda examinar la especificación de cada uno de ellos, para luego seleccionar e integrar uno de ellos dentro de la aplicación en desarrollo.

1.2.2. Adaptación del servicio externo a la aplicación

Una vez que el/la desarrollador/a selecciona el servicio Web de un tercero que implementará una funcionalidad determinada de la aplicación, el enfoque especifica cómo se debe llevar a cabo un proceso de adaptación entre las operaciones definidas por el/la desarrollador/a para su aplicación y aquellas que realmente provee el servicio Web. A este conjunto de operaciones que poseen tipos de datos de entrada y salida se lo denomina interfaz. Esta tarea es necesaria debido a que la interfaz del componente a tercerizar probablemente sea distinta a la interfaz del servicio Web seleccionado. El proceso de adaptación decide por cada método de la interfaz cuál es la operación más similar que provee el servicio Web. A su vez, decide cuál es la mejor correspondencia entre sus tipos de datos, tanto datos de entrada como de salida.

[1] *Un query es una consulta a un repositorio de servicios, debido a que es un término técnico dentro del contexto se utilizará la palabra "query" durante el resto del texto, omitiendo así su traducción directa a la palabra "consulta".*

1.2.3. Incorporación del servicio Web dentro de la aplicación

Una vez que se le presenta al desarrollador/a una manera de adaptar la interfaz del componente a tercerizar y la interfaz del servicio Web, el enfoque define un proceso automático de generación de código correspondiente a una implementación del componente tercerizado. Esta implementación contiene el código necesario para interactuar con el servicio Web seleccionado, además del código necesario para transformar, en caso de ser necesario, los datos de entrada y de salida de cada operación del servicio Web invocada. Luego, se debe ensamblar este componente creado con el resto de la aplicación de manera tal que la aplicación resultante posea altos niveles de mantenibilidad.

1.3. Organización

La organización del trabajo es como sigue: en el capítulo 2 se presenta el contexto en el cual se desea utilizar el enfoque propuesto en este trabajo, para suplir problemáticas existentes y mejorar los procesos de desarrollo actuales; en el capítulo 3 se presentan trabajos relacionados que intentan brindar alguna solución a las problemáticas planteadas que motivan el desarrollo de la tesis; en el capítulo 4 se describe el enfoque propuesto por este trabajo para facilitar el desarrollo y mantenimiento de aplicaciones orientadas a servicios.

En el capítulo 5 se detalla el diseño e implementación de una herramienta que materializa dicho enfoque. En el capítulo 6 se describe un escenario en el cual se puede observar cómo la herramienta aborda una problemática real y da solución a la misma, con el objetivo de medir empíricamente la efectividad del enfoque. En este capítulo se presentan también, el valor de distintas métricas calculadas a lo largo del proceso de desarrollo de la aplicación bajo el escenario planteado.

Finalmente, en el capítulo 7 se presentan las conclusiones del trabajo y se indican las limitaciones de la herramienta como así también las extensiones que se llevarán a cabo en el futuro.

CAPÍTULO 2

Contexto

El objetivo de este capítulo es presentar un paradigma denominado Computación Orientada a Servicios, el cual es utilizado para el desarrollo de aplicaciones distribuidas. Una de las características principales de este tipo de aplicaciones es que promueve la reusabilidad del software de manera desacoplada. Por otro lado, la materialización del paradigma puede lograrse utilizando una serie de protocolos y tecnologías existentes, ampliamente reconocidas y utilizadas en la actualidad. A esta materialización, se le pueden incorporar distintos patrones de diseño a fin de mejorar los requerimientos de calidad de los sistemas de software, como puede ser el patrón de Inyección de dependecia.

Este capítulo está organizado como sigue: en la sección 2.1 se describen nociones generales de Computación Orientada a Servicios (SOC), en la sección 2.2 se presentan las tecnologías con las cuales se materializa el paradigma SOC, en la sección 2.3 se presenta el patrón de diseño Inyección de dependencia cuyo objetivo es mejorar los requerimientos de calidad de un sistema de software, entre ellos el requerimiento de Mantenibilidad.

2.1. Introducción al paradigma Computación Orientada a Servicios

En la actualidad, para enfrentar los tiempos exigentes del mercado y los cambios constantes en los requerimientos, los procesos de desarrollo de software han incrementado su confianza en la reutilización de software [37]. Un paradigma de suma utilidad para el desarrollo de aplicaciones distribuidas, que promueve la reutilización de software de manera desacoplada, es denominado Computación Orientada a Servicios (SOC) [22]. Este paradigma tiene como principal objetivo brindar soporte durante el desarrollo de aplicaciones distribuidas en ambientes heterogéneos. Con SOC, sistemas distribuidos son construidos componiendo o ensamblando funcionalidad existente, la cual es publicada a través de la red y accedida mediante protocolos específicos. A esta funcionalidad se la denomina servicio [22].

Una aplicación orientada a servicios puede ser vista como una aplicación basada en componentes (componentes lógicos de software con una interfaz bien definida que se comunican entre ellos a través de mensajes), la cual es creada a partir de ensamblar dos tipos de componentes: *internos*, los cuales son embebidos dentro de la aplicación, y *externos*, los cuales se encuentran estática ó dinámicamente vinculados al servicio. Ambos exponen una clara interfaz de sus capacidades funcionales. Cuando se construye una nueva aplicación, el diseñador de software debe decidir de proveer una implementación para algún componente de la aplicación, ó bien, utilizar una implementación ya existente. A ésto lo llamaremos tercerizar. En este contexto, tercerizar un componente C significa llenar un agujero dejado por una funcionalidad faltante con una implementación de un servicio existente.

El paradigma SOC, reemplaza el desarrollo de componentes específicos de software con una combinación de distintas actividades: descubrimiento de servicios, selección de servicios e integración de servicios en aplicaciones.

La arquitectura básica de un sistema orientado a servicios incluye componentes capaces de:

- Intercambiar mensajes.
- Describir los servicios.
- Publicar y descubrir las descripciones de los servicios.

La arquitectura básica de un sistema con dichas características define la interacción entre componentes de software como un intercambio de mensajes entre componentes solicitantes de servicios y componentes proveedores de servicios. Un componente solicitante, es aquel que realiza la búsqueda de un servicio en una entidad de descubrimiento de acuerdo a sus necesidades y solicita la ejecución del mismo. Un componente proveedor es responsable de publicar la descripción del servicio que provee en una entidad de descubrimiento, así como aceptar y ejecutar las solicitudes a dichos servicios. Un componente de software puede tanto proveer como solicitar servicios. Una entidad de descubrimiento es un componente en el cual el servicio es publicado y puede ser descubierto, puede ser visto como un registro o directorio de servicios[16]. Por lo tanto, un componente de software definido en este tipo de arquitecturas puede tomar al menos uno de los siguientes roles:

- Solicitante de servicios.
- Proveedor de servicios.
- Entidad de descubrimiento.

En la Figura 2.1 se muestra la interacción entre los roles descriptos. Dicha interacción representa el paradigma denominado "*find, bind and execute*" (Buscar, ligar y ejecutar), el cual involucra descubrir un servicio publicado por algún proveedor que satisfaga las necesidades del solicitante, ligar las operaciones descritas en el servicio e invocarlo [16]. Un servicio es invocado luego que su descripción ha sido descubierta, con lo cual su descripción debe estar disponible en un directorio de servicios (entidad de descubrimiento), que permita a los proveedores registrar los servicios que proveen y a los solicitantes buscar y localizar éstos.

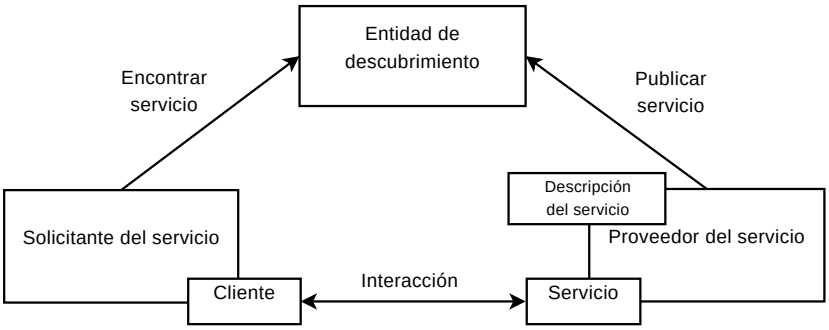


Figura 2.1: Arquitectura básica de un sistema orientado a servicios.

2.2. Materialización del paradigma Computación Orientada a Servicios

Mayoritariamente, la industria del software ha adoptado SOC utilizando la tecnología de servicios Web. Un servicio Web es un programa con una interfaz bien definida que puede ser localizada, publicada y accedida por medio de protocolos Web como HTTP o SMTP [39].

2.2.1. La tecnología de servicios Web

La arquitectura de servicios Web consta de una pila de protocolos, como se observa en la Figura 2.2. Mas allá que esta pila está en constante evolución, podemos indicar que actual-

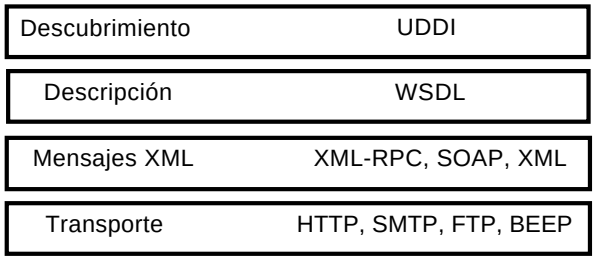


Figura 2.2: Pila de Protocolos de servicios Web.

mente contiene cuatro capas principales. La capa inferior de la pila, denominada capa de *Transporte*, es la responsable de transportar los mensajes entre los componentes de software. Actualmente, esta capa incluye los protocolos Hyper Text Transfer Protocol (HTTP), Simple Mail Transfer Protocol (SMTP), File Transfer Protocol (FTP), entre otros. La capa de *Mensajes* es la responsable de codificar los mensajes en un único formato XML, para lograr un común entendimiento. Esta capa incluye protocolos como XML-Remote Procedure Call (XML-RPC) y Simple Object Access Protocol (SOAP). La capa de *Descripción de servicios* es la responsable de describir la interfaz pública de un servicio Web específico. Actualmente, la descripción de la interfaz se logra a través del protocolo Web Service Description Language (WSDL). Por último, la capa de *Descubrimiento de servicios* es la responsable de centralizar servicios y proveer una interfaz para la búsqueda y publicación de servicios. Actualmente, el des-

cubrimiento de servicios se encuentra materializado por Universal Description, Discovery, and Integration (UDDI).

En la Figura 2.3 se muestra un sistema orientado a servicios, materializado utilizando las

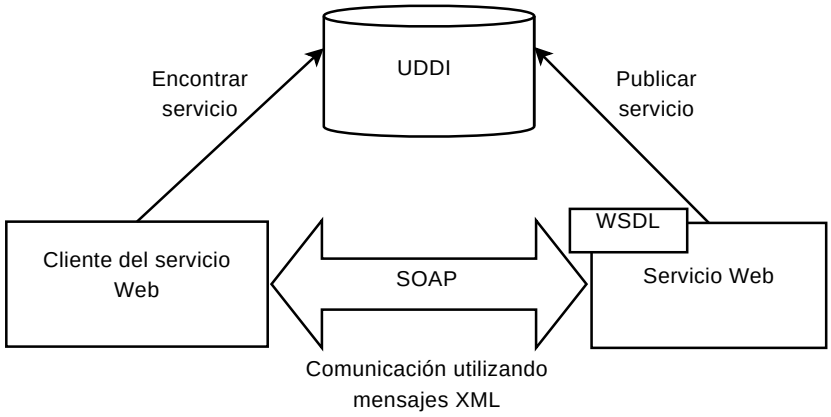


Figura 2.3: Infraestructura estándar de un Sistema orientado a servicios.

tecnologías de Servicios Web, UDDI, WSDL y SOAP.

Descubrimiento de servicios Web

El descubrimiento de servicios potencialmente relevantes, que satisfagan las necesidades del desarrollador, es un proceso que involucra la búsqueda de los éstos en entidades de descubrimiento materializadas en registros UDDI [26].

UDDI consiste en dos partes fundamentales. En primer lugar, es una especificación técnica para construir un registro distribuido de servicios Web. Los datos son almacenados completamente en formato XML y la especificación UDDI incluye detalles de la API para la búsqueda de información existente y la publicación de nueva información. Por otro lado, existen las entidades llamadas UDDI Business Registry que corresponden a una implementación completa de la especificación UDDI.

Estos registros son esencialmente catálogos en los que se publica la especificación de los servicios. Los servicios dentro de los catálogos se encuentran organizados en categorías de acuerdo a la "actividad del negocio" a la que corresponden. Los proveedores de servicios publican sus servicios en la categoría apropiada del registro UDDI. A través de una interfaz (API) bien definida, los desarrolladores de aplicaciones distribuidas pueden examinar el catálogo UDDI por categorías. Típicamente, cuando un usuario busca manualmente en un registro UDDI un servicio que satisface cierta funcionalidad deseada, éste busca primero la categoría relacionada a esta funcionalidad, y luego analiza exhaustivamente los servicios bajo dicha categoría.

Este método de descubrimiento de servicios basado en su categoría es claramente insuficiente. Es un método bastante informal y depende, en gran medida, del conocimiento en común entre el proveedor y el consumidor del servicio. Es responsabilidad del proveedor

publicar el servicio en la categoría apropiada. En cambio, el consumidor debe acertar la categoría para descubrir servicios de relevancia para él. Además, no se provee ningún mecanismo de soporte para la selección entre servicios alternativos [37]. Está claro que a medida que crece la cantidad de servicios publicados en un registro UDDI, más complicada se convierte la búsqueda manual del servicio dentro de una categoría.

Descripción de servicios Web

El lenguaje utilizado para especificar servicios Web se denomina WSDL y se encuentra basado en XML. Así, un servicio Web puede ser descripto en términos de la funcionalidad que él brinda (interfaz pública del mismo), definiendo las operaciones que provee, los mensajes de entrada y salida asociados a cada operación, así como los tipos de datos utilizados como parámetros. Dichas operaciones y mensajes se describen en abstracto y luego se ligan a un protocolo de comunicación en particular, como puede ser SOAP o POST sobre HTTP.

La especificación de servicios está dividida en seis elementos principales:

Definition: Elemento principal de un documento WSDL. Define el nombre del servicio e incluye todos los servicios descriptos en el documento.

Types: Describe los tipos de datos utilizados. WSDL no está asociado a un sistema de tipificación específico. Por defecto, se utiliza la especificación W3C XML Schema.

Message: Corresponde a mensajes "unidireccionales", ya sean de solicitud o de respuesta. Define el nombre del mensaje, además de cero o más elementos denominados Parts, que se refiere a los parámetros de los mensajes, ó bien a valores de retorno.

PortType: Estos elementos combinan múltiples elementos de tipo Message, con el fin de formar una operación, la cual puede ser de solicitud (compuesta por un mensaje de solicitud), de respuesta (compuesta por un mensaje de respuesta) o de solicitud/respuesta (compuesta por un mensaje de solicitud y uno de respuesta). Un PortType puede incluir una ó más operaciones.

Binding: Elemento que describe la especificación de cómo el servicio será implementado cuando se realiza el envío.Más precisamente, detalla como una operación PortType será transmitida. Se puede realizar el transporte vía HTTP GET, HTTP POST ó SOAP.

Service: Define la dirección para invocar el servicio específico. Comúnmente, incluye una URL para invocar al servicio SOAP.

En la Figura 2.4 se detalla la descripción de un servicio Web sencillo cuya funcionalidad es calcular la tasa de cambio (operación getRate) entre dos monedas.

Mensajes y Transporte de los servicios Web

El protocolo de mensaje de mayor renombre en el ámbito de los servicios Web es SOAP (Simple Object Access Protocol), y define cómo dos objetos en diferentes procesos pueden

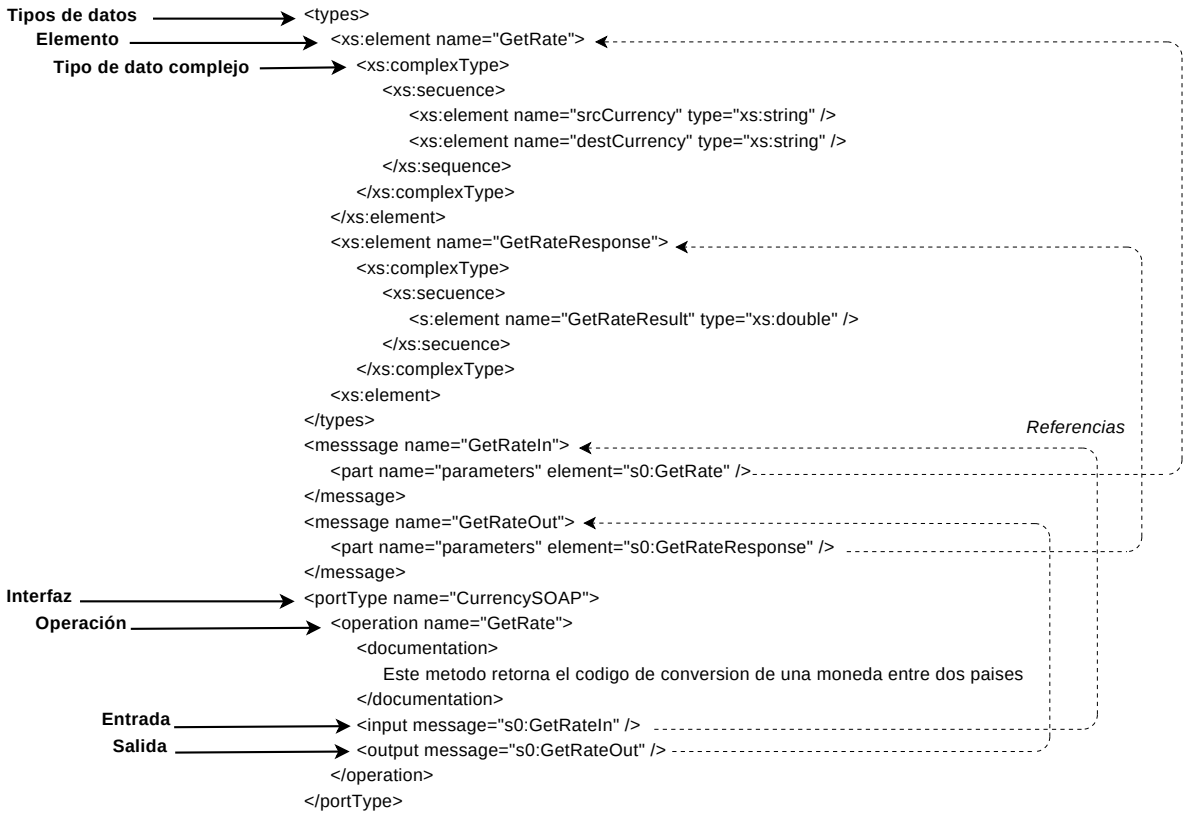


Figura 2.4: Descripción estándar de un servicio Web utilizando el lenguaje WSDL.

comunicarse por medio de intercambio de datos XML. Esto es una ventaja ya que la estructura XML facilita su lectura e interpretación (tanto por un sistema como por una persona), pero también es un inconveniente dado que los mensajes resultantes son más largos. SOAP funciona sobre diversos protocolos de red y en particular funciona sobre cualquier protocolo de Internet, generalmente HTTP. Un mensaje SOAP se encuentra compuesto de una cabecera opcional *Header*, la cual contiene meta-datos sobre enrutamiento, seguridad o transacciones; y una cabecera obligatoria *Body* que contiene la información principal, conocida como carga útil (*payload*). La Figura 2.5 muestra el formato de un mensaje SOAP para un ejemplo simple

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

(a) Solicitud

```
<soap:Envelope xmlsn: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>
```

(b) Respuesta

Figura 2.5: Ejemplo de mensaje SOAP.

en el cual un cliente solicita (a) información de un producto en particular (por medio de un identificador de producto) a un negocio proveedor del mismo, y obtiene como respuesta un detalle completo del producto (b).

2.2.2. Empleo de servicios Web en desarrollo de aplicaciones

Para poder invocar servicios Web desde el código de una aplicación, se debe interpretar la descripción WSDL asociada al servicio tercerizado. Esto se puede realizar de forma manual utilizando librerías de comunicación de bajo nivel, o de manera automática utilizando *frameworks* que proveen facilidades para tratar con el acceso e invocación de estos servicios, por lo que la interacción entre proveedor e invocador del servicio es implementada en un nivel más alto de abstracción. A estos *frameworks* se los denomina *Contract First*: basados en un documento WSDL como entrada, se genera el código cliente necesario para representar el

servicio remoto. A este código generado se lo denomina *Stub* y forma parte del patrón *Proxy* que se detalla en la Figura 2.6.

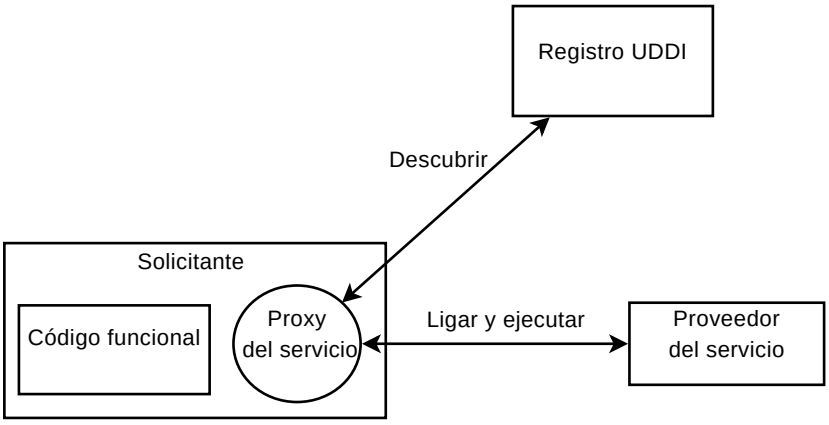


Figura 2.6: Patrón Proxy para la invocación de servicios Web.

Si bien esta práctica permite a los diseñadores separar el código lógico de la aplicación de aquel necesario para la comunicación con el servicio, la lógica de la aplicación estará mezclada con aquel código subordinado a un servicio en particular, como se visualiza en la Figura 2.7. Más precisamente, la lógica estará estrechamente ligada a la interfaz y a los tipos

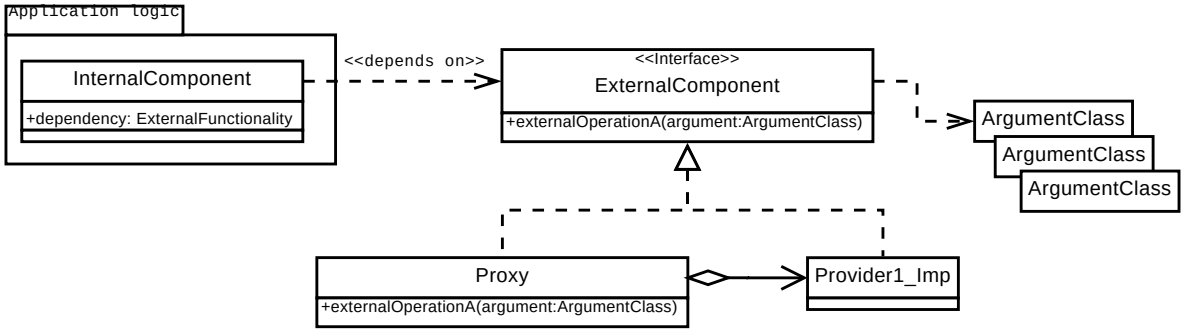


Figura 2.7: Subordinación de código a un servicio en particular.

de datos del servicio. Por lo tanto, un cambio en el proveedor del servicio implica además cambios en el código de la aplicación. En este caso, se debe reemplazar dentro de la lógica de la aplicación las referencias al *Stub* del servicio anterior por aquel del nuevo servicio. Esto reduce la calidad interna del software en desarrollo. De esta forma, lograr modificabilidad y portabilidad del software producido es bastante difícil. En particular, la mantenibilidad del software es extremadamente importante, porque el mantenimiento del software representa cerca el 60-80 % del ciclo de vida total del software [31].

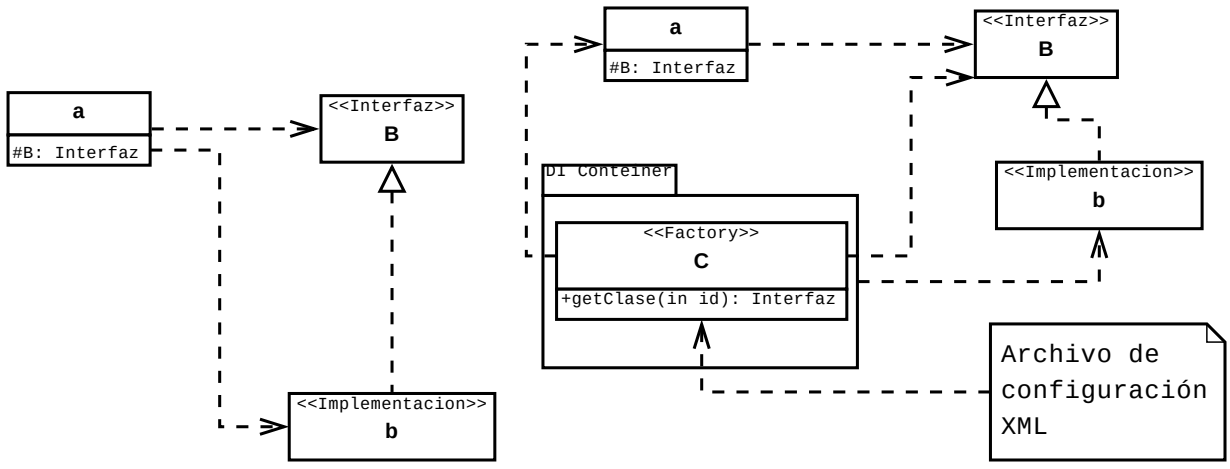
Más allá de los beneficios que proveen los servicios Web, como por ejemplo, el bajo acoplamiento entre el consumidor y el proveedor del servicio, y los altos niveles de interoperatibilidad, la tecnología de servicios Web no es compartida y reutilizada como se desea. La baja frecuencia de la utilización de servicios Web en el desarrollo de aplicaciones orientadas a servicios,

se debe principalmente a dos motivos: en primer lugar, los enfoques estándares de desarrollo de aplicaciones distribuidas requieren que los desarrolladores busquen de manera manual los servicios que se integrarán a la aplicación; y en segundo lugar, el desarrollador debe integrar el servicio previamente seleccionado, agregando código adicional a la aplicación.

2.3. Inyección de Dependencias

Inyección de dependencias (DI) es un patrón de diseño cuyo objetivo es mejorar los requerimientos de calidad de un sistema de software, entre ellos el requerimiento de Mantenibilidad. La mantenibilidad es la facilidad con que puede ser modificado un componente ó sistema de software. Este requerimiento de calidad es de suma importancia ya que el mantenimiento del software insume entre el 60-80 % del ciclo de vida del software [31]. Es difícil realizar una "medición" de la mantenibilidad de un sistema de software debido a que ésta recae en muchos factores, de los cuales algunos de ellos son muy subjetivos. Sin embargo, estudios han demostrado que módulos pequeños, desacoplados y altamente cohesivos tienden a incrementar la mantenibilidad de un sistema de software. Es por ello que el patrón de Inyección de Dependencias atenta a separar programas en componentes más pequeños e independientes, los cuales pueden ser configurados externamente. La suposición realizada es que el patrón de Inyección de Dependencias reduce significativamente dependencias entre/de módulos en una pieza de software, por lo tanto, esto hace que el software sea más mantenible [31].

A nivel conceptual, el patrón de Inyección de Dependencias define que los componentes de un sistema de software de alto nivel no deben depender de componentes de bajo nivel. Ambos deben depender de abstracciones [14]. Si esta definición es expresada en términos del paradigma de Programación Orientado a Objetos, una clase debe depender de una interfaz ó de tipos abstractos, y no de tipos concretos. En la Figura 2.8 se muestra un ejemplo desar-



(a) Ejemplo sin DI (b) Ejemplo con DI

Figura 2.8: Ejemplo del uso de inyección de dependencias.

rollado sin el patrón de DI (a) y el mismo ejemplo utilizando el patrón DI (b). En el primero

de ellos, se ha escrito la clase *a*, que contiene un objeto de clase *b*. La clase *b* es una implementación de la interfaz *B*. Esto hace que la clase *a* tenga una dependencia de la clase *b* así como también de la interfaz *B*. En el segundo ejemplo, se utiliza el patrón de Inyección de Dependencia, reemplazando en la clase *a* la dependencia a la clase *b* por una dependencia a la interfaz *B.* Esto permite que la clase *a* trabaje con cualquier implementación de la interfaz B, eliminando la dependencia previa de *a* a *b* [31].

El patrón de Inyección de dependencias requiere que, en algún lugar del sistema, se instancie una clase concreta (esto es, especificar una implementación en particular de la interfaz en cuestión) para su posterior uso. Por lo tanto, se necesita un mecanismo para que las clases concretas sean instanciadas y pasadas a los clientes de la misma, en lugar de delegar que cada clase instancie directamente los objetos que referencia. Llamaremos *Container* al componente responsable de realizar esta tarea. En el ejemplo mencionado anteriormente, el *Container* es el responsable de proveer y localizar los recursos necesarios para la clase *a*. Es decir, debe crear una instancia de la clase *b* e insertarla (de aquí el término "inyección") en la clase *a*.

Además, el *Container* es configurado externamente, comúnmente a través de un archivo XML. La utilización de un archivo XML para configurar (la dependencia entre) los objetos, permite definir sistemas de software en los cuales el acoplamiento es configurable. Es decir, las interdependencias entre las partes del sistema pueden ser configurados a través de este archivo sin la necesidad de modificar código fuente de dichas partes. Esto permite mantener la mayoría de los acoplamientos en un único lugar. Si se desea cambiar alguno de ellos, el desarrollador sólo deberá editar dicho archivo. Por lo tanto, el uso del patrón de Inyección de Dependencias permite escribir software más mantenible [31].

El patrón Inyección de Dependencias es parte del principio denominado Inversión de Control, término ampliamente usado para referirse a un modelo (de objetos POJOs) en el cual existe un *framework* que instancia los objetos de la aplicación y los configura para su uso en tiempo de ejecución [18]. Una característica propia del patrón de Inyección de Dependencias es que los componentes del sistema no dependen de APIs del *framework* en cuestión, y pueden ser aplicados a objetos que no "conocen" al *framework* [18]. Es decir, objetos que no han sido escritos con el conocimiento del *framework*, lo que implica que estos componentes tienen altos niveles de testeabilidad y reusabilidad.

En particular, aplicar el principio de Inyección de Dependencias en un sistema de software, afecta a ciertos atributos de calidad, contenidos dentro del requerimiento de calidad Mantenibilidad [14]:

Extensibilidad se define como la facilidad con que un sistema o componente puede ser modificado para incrementar su capacidad funcional [14]. A los términos del ejemplo, es más extensible porque se puede utilizar una implementación diferente de B sin modificar el código fuente de la clase *a*.

Testeabilidad se define como el grado en que un sistema o componente facilita el establecimiento de criterios de testeo y la *performance* de testeos para determinar cuando dichos criterios han sido logrados. Con respecto del ejemplo anteriormente mostrado, se puede realizar testeos de la clase *a* sin la necesidad de desarrollar una implementación "real" para la dependencia hacia la interfaz *B*. Es decir, se puede realizar una implementación de esta interfaz que se utilice sólo para los testeos de unidad.

Reusabilidad se define como el grado en que un módulo de software u otro producto puede ser usado en más de un sistema de software. DI puede mejorar reusabilidad de dos maneras: aumentando flexibilidad y rompiendo relaciones transitivas. DI puede aumentar la flexibilidad porque diferentes implementaciones pueden ser utilizadas. Por ejemplo, en la clase *a* del ejemplo previamente comentado, no se debe realizar ninguna modificación si se quiere cambiar la implementación de la interfaz B . Con respecto a romper las relaciones transitivas, para reusar efectivamente una clase es importante que ésta no debe estar ligada a grandes bloques de código innecesario. Si la clase que reutilizamos depende de una clase, transitivamente también depende de cualquier tipo que aparezca en una parte privada de aquella clase.

Spring[1] es el *framework* de Inversión de control más utilizado en la actualidad, existe tanto una implementación Java para el desarrollo de aplicaciones JEE como una implementación para el desarrollo de aplicaciones .NET.

2.4. Conclusión

La materialización de aplicaciones orientadas a servicios es posible utilizando las tecnologías existentes en la actualidad. Sin embargo, la construcción de tales tipos de aplicaciones utilizando enfoques tradicionales (p.ej *Contract First*) conlleva a altos costos de producción durante las etapas del ciclo de vida de tales aplicaciones. Además, el software producido no posee niveles de mantenibilidad deseables. A lo largo del capítulo 3 se presentan distintos trabajos que apuntan a brindar soluciones aplicadas a alguna etapa del proceso de desarrollo de aplicaciones orientadas a servicios, en particular en las tareas de búsqueda e integración de servicios.

[1] www.springframework.org / www.springframework.net

CAPÍTULO 3

Trabajos Relacionados

Como se mencionó con anterioridad, las arquitecturas orientadas a servicios (SOA) fomentan el desarrollo de aplicaciones distribuidas en ambientes heterogéneos, por ejemplo Internet. El desarrollo de dichas aplicaciones, conlleva tanto al descubrimiento de servicios Web, como a la incorporación de los mismos en las aplicaciones. Las técnicas actuales de búsqueda y publicación de servicios Web, presentan limitaciones que dificultan el descubrimiento de éstos. También existen limitaciones en la incorporación de los servicios Web en las aplicaciones, ya que el código necesario para invocar a los servicios externos "contamina" los aspectos funcionales de las mismas. Debido a ésto, han surgido distintos trabajos que intentan mejorar tanto el descubrimiento de servicios como la incorporación de los mismos a las aplicaciones.

Este capítulo está organizado como sigue: la sección 3.1 presenta trabajos para resolver los problemas que involucra el descubrimiento de servicios, la sección 3.2 hace énfasis en trabajos relacionados a la incorporación de servicios externos en una aplicación y en la sección 3.3 se presentan las conclusiones del capítulo.

3.1. Mejoramiento del descubrimiento de servicios

El descubrimiento de servicios es un aspecto crucial para el paradigma SOC. Debido a su importancia, desde la industria y la academia se han generado distintos enfoques de descubrimiento. Por un lado, los mecanismos de descubrimiento mayormente adoptados en la industria son los provistos por Common Object Request Broker Architecture (CORBA) y Universal Description, Discovery and Integration (UDDI). CORBA [29] permite buscar servicios, u "objetos" en la terminología de CORBA, por medio de identificadores alfanuméricos, y a través de uno o más pares <*clave, valor*>. Similarmente, UDDI [26] permite realizar búsquedas basadas en palabras claves y, además, navegar los servicios publicados según taxonomías predefinidas.

Por otro lado, desde la academia distintos investigadores han explorado dos direcciones. En una dirección se propone adaptar técnicas clásicas de *Information Retrieval* (IR) [1] para reducir el problema de encontrar servicios relevantes, al bien conocido problema de encontrar documentos relevantes. Dado que en la práctica las descripciones de servicios presentes en registros, por ejemplo en UDDI, contienen en su mayoría comentarios en idioma inglés, distintas técnicas de IR para tratar con documentos han demostrado que se adaptan con facilidad al dominio de los servicios [4]. La principal ventaja de los esfuerzos en esta dirección es que no agregan costo a los publicadores [4]. Su principal desventaja, es que dependen fuertemente de la capacidad de una descripción en lenguaje natural para comunicar cuál es la funcionalidad de su servicio correspondiente y, al mismo tiempo, de la calidad de una consulta para describir la necesidad de un descubridor, debido a que operan comparando las palabras que forman las descripciones de los servicios contra aquellas que constituyen las consultas.

En otra dirección, se sugiere que las descripciones de servicios puramente sintácticas no son suficientes y que es necesario enriquecerlas [35]. Para aumentar las descripciones tradicionales se promueven dos maneras con características opuestas según cómo impactan en los desarrolladores. Influenciada por la creciente adopción de la Web 2.0, la manera que requiere menos esfuerzo promueve que los servicios sean etiquetados [24] (eje., Seedka![1] anuncia 27.500 servicios etiquetados). En este contexto, una etiqueta o "tag" es una palabra cualquiera que se utiliza para indicar algún atributo relevante del servicio. Los tags no presentan mecanismos para desambiguar dos palabras sintácticamente parecidas ni la noción de sinónimos, pero son fáciles de crear, compartir y usar. En cambio, otra manera promueve el uso de definiciones conceptuales presentes en ontologías compartidas [35, 23, 27], que definen de forma precisa y no ambigua qué hace un servicio, a expensas de un aumento en el esfuerzo necesario para describir el servicio y los conceptos u ontologías relacionados con el primero [34].

En [12] se presenta un compendio de alternativas para descubrir servicios. El resto de la sección resume los trabajos relacionados con descubrimiento más relevantes y que siguen la línea de investigación menos intrusiva.

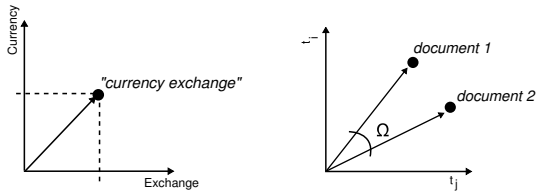
3.1.1. Comparación sintáctica de descripciones de servicios

Algunos métodos de recuperación de información tradicionales [33], originalmente concebidos para tratar con documentos, han mostrado que pueden ser adaptados para tratar con los estándares de descripción de servicios Web [28, 5, 4, 6]. En lugar de introducir un nuevo lenguaje de descripción de servicios o una nueva ontología para describir los mismos, se considera la información existente, y se la utiliza tanto como sea posible. Dicha información puede ser aquella contenida en los archivos de descripción WSDL, o bien, aquella información agregada por los usuarios en los registros UDDI.

Por ejemplo, el propósito del trabajo de [28] es crear un motor de búsqueda, donde toda la información es reunida y utilizada para encontrar la mejor funcionalidad provista por los servicios Web para un requerimiento especifico. Para llevar a cabo ésto, se utiliza un mecanismo de búsqueda que es muy frecuente en sistemas modernos de recuperación de información:

[1] Seedka! `http://seekda.com/`

VECTOR SPACE MODEL [41]. La idea central de este mecanismo es: un documento es dividido en n palabras claves, cada una de éstas constituye una dimensión en un espacio vectorial de n dimensiones. Gráficamente, en la Figura 3.1(a) se representa por medio de un vector,



(a) Representación. (b) Comparación.

Figura 3.1: Vector space model.

un documento que contiene los términos "currency" y "exchange". En consecuencia, documentos que contengan contenidos similares serán representados como vectores cercanos en el espacio, por lo que la búsqueda de documentos similares se traduce a la búsqueda de los vecinos más cercanos en el espacio. En la Figura 3.1(b) se puede observar como el coseno del ángulo Ω provee una estimación de cuan similares son dos documentos.

El núcleo del motor de búsqueda es conocido como TERM SPACE, la idea es crear un VECTOR SPACE donde cada espacio represente un término. Este concepto implica que el espacio crezca a medida que se agregan nuevas palabras claves. En un modelo de recuperación de información basado en VECTOR SPACE, cada documento es representado por un vector $\vec{d} = (d_1, d_2, ..., d_n)$, donde cada componente d_i es un número real que indica el grado de importancia del i-ésimo término en el documento d. La forma en que este peso se calcula es un factor que impacta sensiblemente al momento de medir cuan precisos son los resultados de un sistema de recuperación.

El método para pesar los términos de un documento más sencillo es el *booleano*, donde d_i es igual a 1 si el i-ésimo término pertenece a d y 0 en caso contrario. Este método es muy limitado, ya que ignora información importante como la frecuencia del los términos ó la longitud de un documento. Otro método para asignar peso a un término, que ha demostrado mejorar la precisión de los buscadores de documentos en general, es TERM FRECUENCY AND INVERSE DOCUMENT FRECUENCY (TF-IDF) [33], y en particular en el dominio de los servicios Web [28, 4]. TF-IDF considera la frecuencia de los términos tanto en un documento como en toda la colección, dando un peso alto a un término si es frecuente en un documento pero ocurre pocas veces en el resto de los documentos de la colección. Formalmente, para cada término t_i de un documento d, $tfidf_i = tf_i \bullet idf_i$, con:

$$tf_i = \frac{n_i}{\sum_{j=1}^{T_d} n_j} \tag{3.1}$$

donde el numerador (n_i) es el número de ocurrencias dentro de d del término a ser considerado, y el denominador es el número de ocurrencias de todos los términos dentro de d (T_d), y:

$$idf_i = \log \frac{|D|}{|\{d : t_i \epsilon\, d\}|} \tag{3.2}$$

donde $|D|$ es el número total de documentos en la colección y $|\{d : t_i \epsilon\, d\}|$ es el número de documentos donde el término t_i aparece.

Habiendo presentado los conceptos básicos del VSM, a continuación se explica cómo se adaptan estas ideas al dominio de los servicios. Como los servicios Web se describen por medio de documentos, sus descripciones pueden traducirse a vectores. Cada descripción de los servicios Web (archivos WSDL) incluidos en los registros UDDI es analizada con el fin de recuperar datos como son las definiciones, elementos y nombre del servicio. Luego, se extraen las palabras que conforman estos datos para crear el VECTOR SPACE MODEL. En la Figura 3.2 se puede observar la arquitectura básica del motor de búsqueda basado en VSM.

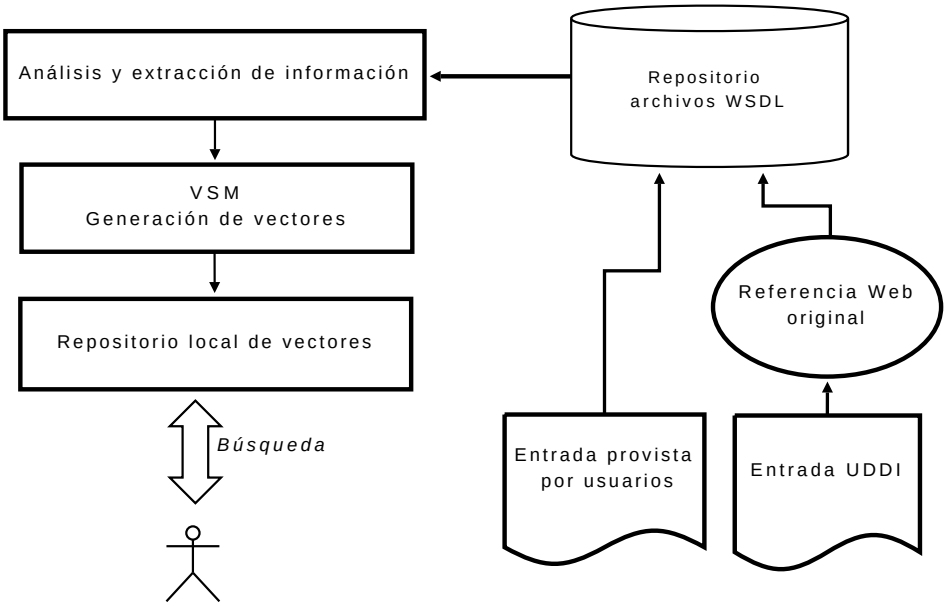


Figura 3.2: Arquitectura básica Vector Space Model.

Una vez que todos los documentos (descripciones WSDL de los servicios) son representados dentro de un mismo espacio de búsqueda, la relevancia entre ellos puede ser calculada de acuerdo a distintos métodos. La similitud con otro documento dentro del mismo espacio de búsqueda puede ser calculada mediante un algoritmo, con el fin de obtener un *ranking* final de resultados relevantes. El algoritmo más utilizado se denomina COSINE COEFFICIENT. Éste toma dos vectores y calcula el valor del coseno del ángulo que forman los mismos. Lo que implica que dos documentos con un ángulo pequeño entre sus representaciones vectoriales estarán relacionados uno con el otro, por lo que documentos con ningún término en común tendrán valor de coseno igual a cero, mientras que documentos idénticos tendrán valor 1. Formalmente, la similitud entre los documentos $\vec{q}$ y $\vec{s}$ se mide a través de la ecuación:

$$cosineSimilarity(\vec{q}, \vec{s}) = \frac{\vec{q} \bullet \vec{s}}{|\vec{q}||\vec{s}|} = \frac{\sum_{i=1}^{T} e_{\vec{q},i} e_{\vec{s},i}}{\sqrt{\sum_{i=1}^{T} e_{\vec{q},i}^2 \sum_{i=1}^{T} e_{\vec{s},i}^2}} \tag{3.3}$$

donde $e_{[\vec{q}|\vec{s}],i}$ es el valor TF-IDF correspondiente $\vec{q}$ y $\vec{s}$ en la i-ésima dimensión. El denominador en la ecuación 3.3 normaliza los vectores $\vec{q}$ y $\vec{s}$, lo que produce que éstos mantengan

su dirección original, pero su longitud/tamaño se convierta en 1. De esta forma, la comparación de documentos es independiente a la longitud de los mismos, evitando una tendencia favorable hacia aquellos con mayor tamaño. La complejidad de esta métrica de similitud es *O(t)*, donde *t* es el número de dimensiones que poseen los vectores, es decir los términos de los documentos.

Alternativamente, el enfoque WSQBE [5, 4] presenta un motor de búsqueda basado en un modelo de espacio vectorial, pero particionado. Dado que si el número de vectores en el espacio vectorial es grande, el proceso de *matching* puede ser muy costoso, pues implica comparar el vector del *query* con cada uno de los vectores que representan las descripciones de los servicios disponibles. Para solucionar este obstáculo en la *performance* del proceso *matching*, el enfoque WSQBE enriquece el motor de búsqueda utilizando un mecanismo de reducción del espacio vectorial basado en el algoritmo de clasificación *Rocchio* [17]. Este clasificador divide al espacio vectorial en subespacios, uno por cada categoría de servicios disponibles en un registro UDDI. Cada subespacio es representado por un vector promedio, llamado *Centroide*, calculado a partir de todos los vectores del subespacio. A su vez, los vectores pertenecientes a un mismo subespacio, representan descripciones de servicios que pertenecen a una misma categoría.

Una vez calculado el vector *centroide* para cada categoría, los pasos que desarrolla el proceso de descubrimiento, el cual se basa en un proceso de *matching*, son los siguientes:

1. Determinar la categoría más cercana al *query*: se compara el *query* con cada uno de los vectores *centroide*, con el fin de obtener a qué categoría se asemeja el *query* en cuestión. Las operaciones realizadas en este paso tienen un costo computacional del orden al número de categorías existentes en el repositorio.

2. Comparar el *query* contra los servicios pertenecientes a una categoría: se compara el *query* contra cada uno de los vectores que representan los servicios de la categoría seleccionada previamente. A partir de la medida mencionada anteriormente, se ordenan los servicios de forma decreciente de acuerdo al grado de similitud entre el *query* y cada servicios. Esta lista ordenada de servicios, será la salida del proceso de descubrimiento de servicios. Las operaciones realizadas en este paso tienen un costo computacional del orden del número máximo de servicios existentes en una categoría.

Se puede definir la complejidad del proceso de descubrimiento como la cantidad de comparaciones de vectores realizadas. Si el vector correspondiente al *query* se compara usando la métrica del coseno contra todos los vectores del espacio de búsqueda (es decir, sin aplicar reducción del espacio vectorial), tal como se plantea en 3.1.1, el proceso es del orden:

$$O(St) \tag{3.4}$$

donde S representa la cantidad de servicios disponibles (es decir, vectores en el espacio vectorial) y t es el número de términos diferentes en el espacio vectorial.

La complejidad del proceso de búsqueda propuesto por el enfoque WSQBE (el cual aplica reducción del espacio vectorial) es del orden:

$$O(max(Ct, S_C t_C)) \tag{3.5}$$

donde C es el número de categorías existente, S_C es el número máximo de servicios por categoría y t_C es el número máximo de términos de una categoría. El número de categorías C es, a lo sumo, igual a S, es decir $S \geqslant C$. El número máximo de servicios por una categoría S_C es, a lo sumo, igual a S, es decir, $S \geq S_C$. El número máximo de términos diferentes por categoría t_C es, a lo sumo, t, es decir, $t_C \leq t$. Claramente, la reducción del espacio vectorial se traduce a una menor cantidad de comparaciones entre vectores. Por ejemplo, si $C + S_C < S$, la reducción del espacio vectorial produce que se realicen menos comparaciones en relación si no se redujera el espacio vectorial. En cambio, si $C + S_C > S$ implica que exista un servicio dentro de cada categoría, por lo que el enfoque de reducción de espacio requiere sólo una comparación más. Por lo tanto, dependiendo de las características lingüísticas de cada sub-lenguaje usado en las categorías, el enfoque WSQBE requeriría menos operaciones para llevar a cabo la comparación de vectores[4].

Otra de los beneficios que brinda el enfoque WSQBE es que ofrece un lenguaje de consulta muy flexible. Básicamente, WSQBE acepta una, o más, palabras clave a modo de *query*. Adicionalmente, ofrece dos métodos de preprocesamiento para extraer palabras clave desde:

- Documentos WSDL: permite buscar un servicio semejante a aquel descripto en la mencionada especificación. Es útil para reemplazar un servicio por otro nuevo.
- Interfaz de Componente a tercerizar: permite buscar a partir de una interfaz que defina la funcionalidad del componente a tercerizar. Es útil para lo programadores, por están familiarizados con el lenguaje de consulta.

WSQBE ha demostrado, empíricamente, que retorna un servicio relevante en la posición 1 en el 83 % de las consultas que se analizaron utilizando un conjunto de servicios de 391 y 30 *queries* [4]. Vale aclarar que para ese experimento se utilizaron interfaces, escritas usando el lenguaje de programación Java, a modo de *queries*.

Otro trabajo que sigue la línea de no alterar ni la arquitectura clásica SOA ni los estándares de descripción de servicios para buscar servicios, basándose en comparaciones sintácticas, es [19]. Este trabajo presenta un *framework* de recuperación de servicios Web. Los autores también utilizan enfoques de recuperación de información para comparar los servicios. Este *framework* se ubica por encima de los registros UDDI existentes, y permite recuperar servicios en un orden de importancia tipo *ranking*. La Figura 3.3 detalla la arquitectura del *framework*.

Entre los lenguajes de consulta soportados por [19] se encuentran: consultas textuales y mediante el uso de plantillas. Un plantilla es una interfaz en la cual se define el conjunto de operaciones que el servicio debe proveer. Para realizar el *ranking* de los resultados, el *framework* utiliza el modelo de VECTOR SPACE y modos de cálculo TF-IDF.

El proceso de *matching* en [19] consiste de varios pasos. El primer paso consiste en el análisis de la descripción del servicio a partir de su especificación WSDL y la extracción de las descripciones desde el registro UDDI utilizando conexiones de bases de datos. El siguiente paso es separar las palabras concatenadas en términos individuales, y cada palabra es contraída a su raíz, ésto se aplica a cada elemento y atributo de la especificación WSDL. Luego, se etiqueta cada descripción del servicio y se construye un índice BITMAP para documentos XML conocido como BITCUBE, que representa los valores de los elementos en términos de bits de palabras. En el paso final se calcula la similitud entra cada consulta de entrada y

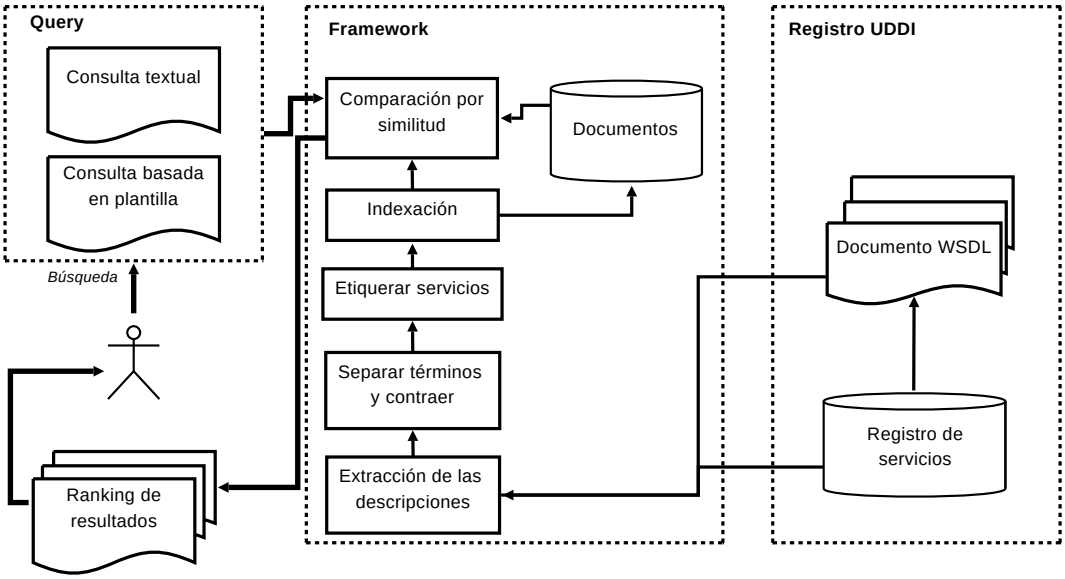


Figura 3.3: *Framework* de recuperación de servicios basado en *ranking.*

las descripciones de los servicios. Para realizar dicha tarea, se toma una variante del método TF-IDF, donde el peso de cada término es acorde a la posición donde se encuentra, ya que no sería lo mismo el valor para un término dentro del nombre del servicio que en la descripción textual del mismo, como se observa en la ecuación:

$$tf_i^{\bigstar} = 0{,}7 \cdot tf_i(nombre) + 0{,}3 \cdot tf_i(descripcin) \tag{3.6}$$

Una vez procesada la entrada como un vector de palabras, se procede a realizar el producto escalar entre ésta y cada descripción de servicio dentro del VECTOR SPACE. Este valor será el indicador de similitud entre cada servicio Web del registro y la consulta realizada.

3.1.2. Comparación semántica y estructural de descripciones de servicios

En [37, 38] los autores proponen un conjunto de métodos para calcular la similitud entre dos especificaciones WSDL de servicios Web. Como se explica a continuación, estas técnicas intentan aprovechar al extremo más información presente en un WSDL, en particular la estructura del documento.

Para evaluar la similitud entre dos especificaciones WSDL, en [37] se presenta un algoritmo que busca patrones XML que aparezcan en la estructura de ambas. Involucra la comparación del conjunto de operaciones que ofrecen los servicios, la cual se basa en la comparación de las estructura de los mensajes de entrada y salida de las operaciones, que a su vez, se basa en la comparación de los tipos de datos asociados a esos mensajes. Los tipos de datos de servicios Web son descriptos mediante lenguajes basados en XML. Si bien WSDL permite distintos enfoques de tipificación de datos, el más utilizado se denomina XML SCHEMA LANGUAGE, también conocido como XML SCHEMA DEFINITION (XSD) [7]. La especificación XML Schema incluye un sistema de tipos simples incorporados, en los que se incluye *strings, floats, doubles, integers, time* y *date.* Además, la especificación XML Schema provee la facilidad

de crear nuevos tipos de datos, con el fin de representar estructuras de datos más complejas. Es decir, los tipos pueden ser creados a partir de uno ó mas tipos de datos simples y/o tipos de datos complejos previamente creados . Es por ello, que la comparación entre tipos de datos se basa en las siguientes heurísticas:

- Dos tipos de datos simples son comparados en base a sus tipos de programación (con lo cual pueden ser compatibles, compatibles con pérdida de información o incompatibles).
- Dos tipos de datos complejos son comparados en base a los elementos que los constituyen y a la organización grupal entre ellos.
- Dos tipos de datos complejos, importados desde el mismo *namespace*, son considerados idénticos si tienen el mismo nombre.

En [38] se extiende el trabajo anterior incorporando un algoritmo para tener en cuenta la semántica de las palabras que se extraen de un documento WSDL. Se consideran los identificadores de los elementos del servicio, en adición a los tipos de programación y relaciones sintácticas. Este enfoque utiliza el diccionario de sinónimos WordNet [9] para calcular el grado de similitud entre cada identificador de cada par de elementos comparados en la especificación WSDL (servicios, operaciones, tipos de datos, etc). Para hallar el grado de similitud entre dos términos se debe determinar, mediante el uso de WordNet, si estos son sinónimos, hiperónimos, hipónimos ó si no existen ninguna relación entre ambos, este grado de similitud provisto por WordNet es un valor que se conoce como "distancia semántica". La hipótesis que sostiene este enfoque es que los nombres elegidos para los tipos de datos, operaciones y servicios usualmente reflejan la semántica de la capacidad esencial de un servicio. El proceso general es similar a la correspondencia estructural, con la única diferencia que en lugar de evaluar la similitud de dos tipos de datos, se evalúa la distancia semántica de los identificadores asociados a los tipos de datos.

Por medio de éstas técnicas, se pueden realizar búsquedas de nuevos servicios Web a partir de un servicio ya conocido, y comparar su estructura con el conjunto de servicios dentro de un registro, o también, pueden utilizarse como métodos adicionales de filtrado o refinamiento de resultados en un subconjunto de servicios descubiertos a partir de otros enfoques de descubrimiento.

3.1.3. Comparación combinada: sintaxis más *clustering*

En [6] se presenta un motor de búsqueda de servicios Web llamado *Woogle*, el cual permite, además de la búsqueda de servicios a partir de una palabra clave, la búsqueda de servicios Web similares a partir de la especificación de uno de ellos mediante un documento WSDL, completo o parcial. Para ello, se ha desarrollado un algoritmo de *clustering* que agrupa nombres de parámetros de operaciones de servicios Web en conceptos semánticos.

Clustering [1] es el proceso de particionar un conjunto de datos en subconjuntos, también denominado *Clursters*, de acuerdo a una característica que, idealmente, comparten los ejemplares de un mismo cluster. El objetivo de este algoritmo es brindar soporte semántico en la búsqueda de correspondencias entre parámetros de operaciones de servicios. El mencionado

algoritmo propone crear una serie de reglas de asociación que reflejen el grado de relación entre los distintos términos que conforman los identificadores de los parámetros de una operación. Luego, utilizando un algoritmo de *clustering*, y en base a estas reglas, se crean distintos conceptos semánticos. La heurística para formar estos conceptos es: "los parámetros tienden a expresar el mismo concepto si ocurren juntos" [6]. Estos conceptos, son utilizados posteriormente para determinar la similitud en los datos de entrada y salida de las operaciones de servicios Web.

El algoritmo de búsqueda de servicios presentado en [6], combina técnicas de recuperación de información tradicionales con el algoritmo de *clustering* presentado en este trabajo. Para calcular la similitud entre dos servicios Web se considera la similitud entre los siguientes elementos de las especificaciones de servicios Web:

- Nombre de parámetros de entrada y salida de una operación: Se consideran estos nombres como términos y se comparan con la medida TF/IDF. Para mejorar la precisión, al conjunto de términos se le aplica un preprocesamiento, como *Stemming*, *Removing Stop Words*, entre otras técnicas.

- Conceptos asociados a cada parámetro de entrada y salida de una operación: Para obtener la similitud de conceptos, se obtienen los términos resultantes del preprocesamiento de los nombres de parámetros, de manera similar al paso anterior. Luego, cada uno de éstos es reemplazado por su Concepto, los cuales fueron obtenidos a partir del algoritmo de *clustering* presentado anteriormente. Finalmente, se aplica la medición TF/IDF entre los conceptos involucrados.

- Descripción de Operación: Para obtener la similitud entre descripciones de operaciones, se comparan los conjuntos de palabras de cada descripción, utilizando la métrica TF/IDF.

- Descripción de servicio Web: Para obtener la similitud entre descripciones de servicios, se considera que cada descripción está formada por el siguiente conjunto de palabras: el nombre del servicio Web, la documentación contenida en el archivo WSDL, la descripción del servicio Web obtenida en la UDDI, el nombre de las operaciones y el de sus mensajes de entrada y salida. Luego, se comparan los conjuntos de términos obtenidos utilizando la métrica TF/IDF.

Por último, se obtiene la similitud entre dos servicios Web combinando linealmente los resultados obtenidos de manera individual, pudiendo asignar distintos pesos a cada uno de ellos.

Con respecto a la evaluación del algoritmo de cálculo de similitud, los resultados experimentales han demostrado que la técnica presentada en este trabajo, aumenta significativamente los valores de Precisión y Recall comparado con métodos más simples de comparación de servicios, basados en técnicas recuperación de información [6]. La precisión de Woogle ha sido evaluada utilizando un *data-set* con 411 servicios Web, de los cuales 25 de ellos se utilizaron como *queries*. El valor obtenido para esta métrica fue del 78 %. Por otro lado, el valor de Recall fue evaluado con 8 *queries*, obteniendo un valor del 88 %.

3.2. Incorporación de servicios externos en aplicaciones

En esta sección se presentan trabajos que intentan mejorar la incorporación de servicios externos en aplicaciones orientadas a servicios, ya que las herramientas existentes no proveen un soporte total en esta materia, resultando en un trabajo adicional del desarrollador/a. En 3.2.1 se presenta un trabajo para obtener una versión distribuida de una aplicación centralizada, en 3.2.2 se presenta una capa de software entre la aplicación cliente y los servicios Web que permite abstraer las características particulares de los distintos proveedores, en 3.2.3 se presenta un *framework* para el desarrollo de aplicaciones distribuidas, en 3.2.4 se presenta un enfoque de adaptadores de servicio para garantizar la interoperabilidad entre diferentes servicios Web.

3.2.1. Invocación dinámica de servicios Web utilizando Programación Orientada a Aspectos

En [32] se propone un método para obtener una versión distribuida de una aplicación centralizada, realizando selección e integración dinámica de servicios Web apropiados. El objetivo del mencionado trabajo es simplificar el proceso de *deployment* de aplicaciones Web, reduciendo su costo de producción y mantenimiento, obteniendo al mismo tiempo, una mejora considerable en cuanto a su flexibilidad y dinamismo.

La tarea de selección de servicios Web es llevada a cabo por medio de un componente llamado *Brokering Service*, mientras que la tarea de integración se realiza utilizando el paradigma de Programación Orientada a Aspectos (AOP) [21]. Mediante AOP se busca separar el código de invocación de servicios Web (el cual reemplaza al código de invocación de métodos locales) del código de la aplicación, y luego entrelazarlos en tiempo de ejecución. En la Figura 3.4 se puede observar cómo se realiza el enlazamiento de código utilizando el paradigma de AOP, donde cada aspecto contendrá el código de invocación a un servicio Web en particular. El hecho de separar aspectos de distribución del diseño conceptual, permite al enfoque propuesto simplificar considerablemente el proceso de desarrollo de aplicaciones Web [32]. Por lo tanto, el uso del paradigma AOP brinda la ventaja que el código correspondiente a los aspectos no se encuentra diseminado a lo largo de distintos componentes de la aplicación, con lo cual, estos componentes sólo contienen código funcional, obteniendo menores costos en la producción y mantenimiento de los mismos, y al mismo tiempo, permitiendo aumentar su reusabilidad y flexibilidad.

El método de desarrollo presentado en [32], en primer lugar, propone especificar cuál/es servicio/s se van a utilizar. Esto se puede hacer en forma manual, por ejemplo, indicando explícitamente la dirección URL del servicio Web, ó en forma automática. Para realizar esto automáticamente, se debe proveer una consulta (o *query*), la cual será utilizada para buscar servicios. Dicha consulta, se define mediante la especificación de los argumentos, entradas y salida, que debe soportar un potencial servicio. Esta información de configuración se plasma en un documento XML.

En segundo lugar, [32] propone la etapa de generación de código. Esta etapa comienza analizando el documento XML descrito, con el objetivo de obtener la especificación WSDL de cada servicio Web. Si en el documento se encuentra especificada la URL del WSDL, éste se recupera. Si no, se analiza la descripción básica del servicio a partir de la información que

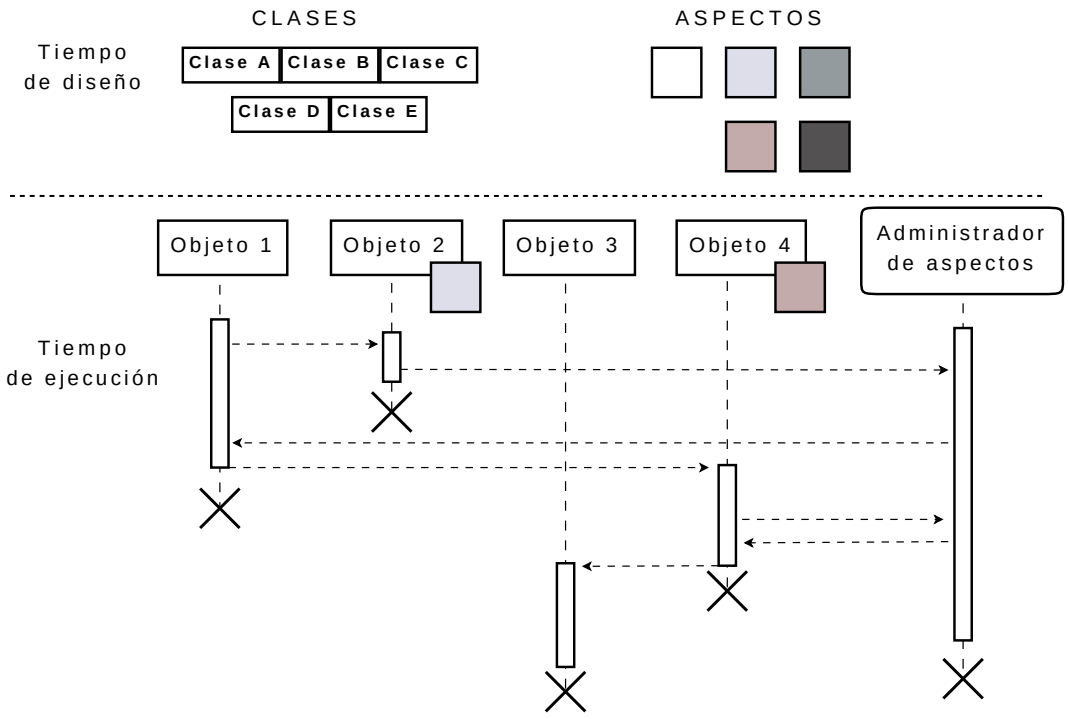


Figura 3.4: Enlace dinámico de código utilizando el paradigma Programación Orientada a Aspectos.

contiene en el documento XML. Luego, esta información es utilizada por el componente *Brokering Service* con el fin de descubrir un servicio Web apropiado, retornando la URL correspondiente al documento WSDL de dicho servicio.

Luego de recuperar la especificación WSDL del servicio Web a utilizar, se genera el código de los aspectos para invocar a los servicios Web. Cada aspecto generado, es estructurado en dos partes. La primera contiene todo lo necesario para inicializar los elementos que invocan a servicios Web (se utiliza un *Advise* de tipo *Before*). La segunda contiene el código de invocación del servicio Web y el análisis de su respuesta (se utiliza un *Advise* de tipo *Around*). Un *Advise* es un elemento del paradigma AOP que se utiliza para encapsular código que se debe ejecutar antes, después o mientras se invoca un método determinado [21]. El tipo de un *Advise* indica cuándo se ejecutará ese código, por ejemplo, un *Advise* de tipo *Before* significa que el código extra debe ejecutarse antes de invocar el método. Concretamente y retornando al enfoque descripto, al invocar un método que ha sido tercerizado, en realidad se activa un *Advise Before*. Éste, genera código que se usará para invocar el servicio Web, que ha sido explícitamente especificado o que se obtiene como respuesta desde el *Brokering Service*. El *Advise Around* encapsula el código que invoca remotamente al servicio y luego obtiene su respuesta.

El componente *Bokering Service* tiene como objetivo extender la funcionalidad provista por los registros UDDI. Este componente establece un canal de comunicación entre los clientes y los proveedores de servicios, y mantiene un registro de servicios para asegurarse que los servicios Web puedan ser descubiertos dinámicamente.

Como conclusión, en este trabajo se presenta una solución al problema de simplificar el desarrollo de aplicaciones Web, reemplazando invocaciones a métodos locales por invoca-

ciones a servicios Web sin modificar el código fuente de la aplicación.

3.2.2. Capa de administración de servicios Web

Para permitir el desarrollo de aplicaciones más flexibles y robustas, [3] propone la inserción de una nueva capa de software entre la aplicación cliente y los servicios Web, denominada CAPA DE ADMINISTRACIÓN DE SERVICIOS WEB (WSML). Dicha capa desacopla los servicios Web de las aplicaciones consumidoras, permitiendo cambiar de proveedor de servicio de manera transparente y dinámica. Estos cambios están basados en políticas de selección y cambios en las reglas del negocio. Estas políticas representan distintos requerimientos funcionales de la aplicación, siendo utilizadas durante el proceso de selección de servicios. Por ejemplo, WSML provee una librería de *templates* de políticas de selección que pueden ser utilizados para aprobar un conjunto específico de servicios Web.

La capa de administración de servicios Web, está constituida a partir del paradigma Programación Orientada a Aspectos (AOP). De esta manera, para realizar la comunicación con los servicios Web se evita utilizar el patrón *Proxy* cuya principal falencia es la baja mantenibilidad del código de la aplicación, como se detalló en 2.2.2. El uso del paradigma AOP contribuye a una clara modularización de la selección, integración y administración de servicios Web en la aplicación cliente, es decir, del lado del cliente. Esta modularización implica extraer de la aplicación cliente el código relacionado a los servicios Web, reduciendo las líneas de código de la aplicación, por lo tanto se facilita su mantenimiento.

WSML materializa el concepto de integración dinámica "*just in time*" de servicios: múltiples servicios pueden ser seleccionados, compuestos y utilizados en tiempo de ejecución para proveer la misma funcionalidad. Las ventajas de este enfoque son:

- Las aplicaciones se vuelven más flexibles, debido a que pueden adaptarse a los cambios en las reglas del negocio e interactuar con nuevos proveedores que en tiempo de desarrollo eran desconocidos.
- El cambio entre distintos proveedores en tiempo de ejecución es posible. De ésta forma, servicios que son inaccesibles debido a problemas en la red, congestionamiento, o problemas propios del servidor pueden ser reemplazados por servicios equivalentes.
- El reemplazo de código de invocación para servicios específicos por un modo genérico de invocación, encargado de la selección y administración del código de todos los servicios externos, facilita el mantenimiento y adaptabilidad de la aplicación.

La Figura 3.5 muestra la arquitectura general de la capa de administración de servicios. WSML es responsable de interceptar los pedidos a los proveedores externos y seleccionar el servicio apropiado para el mismo (redirección). La selección de un servicio Web se encuentra basada en disponibilidad y accesibilidad del mismo (selección). Puntos de monitoreo son introducidos en la aplicación cliente permitiendo un alto grado de flexibilidad (monitoreo).

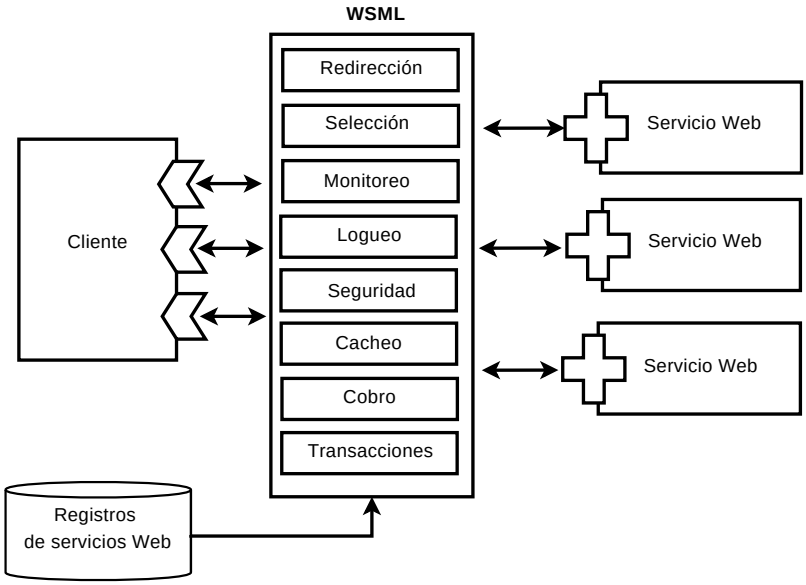


Figura 3.5: Arquitectura general del *framework* WSML.

3.2.3. Implementación de requerimientos de calidad en aplicaciones distribuidas:

En [8] presenta un *framework* denominado Object Infraestructure Framework (OIF) para el desarrollo de aplicaciones distribuidas, basado a partir de la extensión de la arquitectura Object Request Brokers (ORB) [36]. Ésta es una tecnología de *middleware* para administrar la comunicación y el intercambio de datos entre objetos distribuidos (los cuales representan distintos componentes funcionales). ORBs promueve la interoperabilidad de sistemas de objetos distribuidos debido a que permite a los desarrolladores construir sistemas ensamblando objetos -de diferentes proveedores- los cuales se comunican entre ellos a través de ORB [36]. Además, se oculta la forma en que estos objetos son localizados, como así también las características del medio de comunicación entre ellos. Esta arquitectura está conformada por objetos *proxy* tanto del lado de los cliente como del lado del servidor. El *proxy* del lado de cliente (también denominado *Stub*) es el responsable de realizar la conversión de la solicitud de un cliente (procedimiento conocido como *marshalling*) a un formato que pueda ser trasmitido por la red. El *proxy* del lado del servidor (también denominado *Skeleton*) realiza una conversión del requerimiento (procedimiento conocido como *desmarshall*) a estructuras locales nativa.

Este trabajo provee un lenguaje de alto nivel para la especificación de requerimientos no funcionales (también denominados requerimientos de calidad o *Ilities*) que se desean implementar en una aplicación desarrollada a partir del mencionado *framework*. Algunos *Ilities* considerados son Disponibilidad, Confiabilidad, Seguridad, *Performance* y Mantenibilidad. De esta forma, se mantiene una clara separación entre los componentes funcionales del sistema de aquellos no funcionales. La mencionada separación permite que los *ilities* puedan ser desarrollados, mantenidos y modificados con el menor impacto sobre las implementaciones funcionales.

La ejecución de estos requerimientos no funcionales *Ilities* se logra interceptando y manipu-

lando la comunicación entre los componentes funcionales. Los interceptores de esta comunicación son objetos denominados *Injectors*. En un sistema distribuido, una *Ility* puede ser inyectada por un *Injector* particular, tanto del lado del cliente como del servidor. Además, cada instancia y cada método de un objeto funcional pueden tener asociada una secuencia de *Injectors* distinta.

En la Figura 3.6 se puede observar una secuencia de *injectors* que manipulan la comunicación

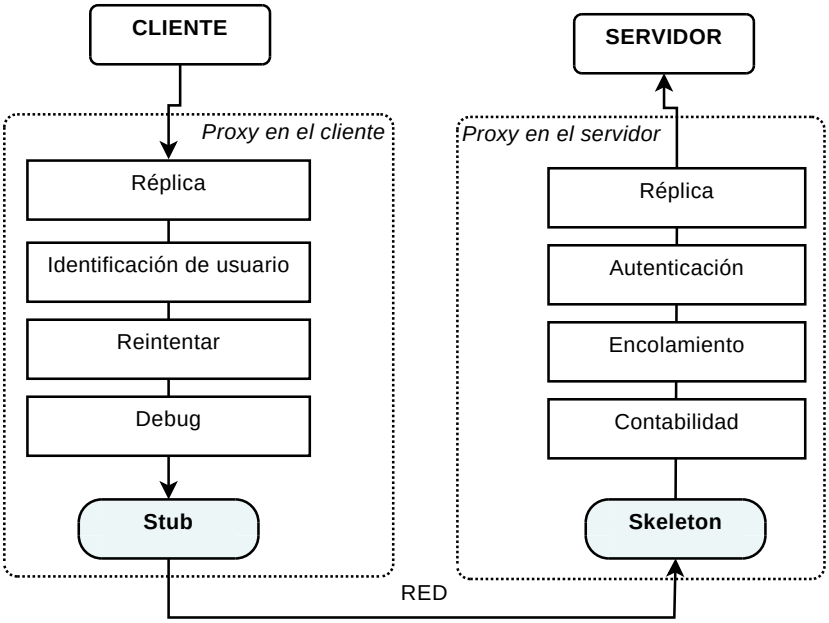


Figura 3.6: Ejemplo de inserción de *injectors* entre un cliente-servidor.

entre el cliente y el servidor para un método en particular.

La implementación del *framework* OIF se ha logrado modificando los *proxies* ORB. Cada uno de estos *proxies* contiene una estructura (de tipo Map) con las secuencia de *Injectors* correspondientes a cada método del *proxy*. Ante la invocación de un método específico de un *proxy*, se añade la ejecución de los *Inyectors* pertenecientes a la secuencia asociada al método en cuestión.

Existe una distancia conceptual bastante grande entre *Ilities* abstractas y una secuencia discreta de *Injectors*. Se denomina *Ility* abstracta, a aquel requerimiento de calidad que dado su nivel de abstracción, no depende de un problema en particular y su implementación puede ser reutilizada en distintas aplicaciones, un ejemplo de ésto puede ser el requerimiento "la aplicación debe ser segura". Para achicar este margen se ha creado PRAGMA, un compilador que toma una especificación de alto nivel de propiedades deseadas, mapeando dicha especificación con los apropiados conjuntos de *Injectors*. Específicamente, el compilador requiere una especificación en dicho lenguaje más una especificación de la interfaz de los componentes funcionales. La salida del compilador será un *proxy* ORB (correspondiente a la mencionada especificación de interfaz) que incluye la implementación de los requerimientos de calidad especificados.

3.2.4. Adaptación semiautomática de interacciones entre servicios

En [25] se presenta un nuevo enfoque para el desarrollo de clientes consumidores de servicios, el cual se basa en el desarrollo de adaptadores de servicios. La adaptación de servicios refiere al proceso de generar un servicio (adaptador) que actúe de mediador en la interacción entre dos servicios que poseen la misma funcionalidad pero diferentes interfaces (operaciones definidas formalmente) y protocolos (definen restricciones de orden para permitir la invocación de operaciones en una cierta secuencia), de ésta manera se puede garantizar un cambio transparente en el proveedor del servicio.

Para poder garantizar los cambios de proveedor se deben identificar las incompatibilidades de las interfaces y protocolos entre distintos servicios. Las incompatibilidades más comunes dado un proveedor SP y un consumidor SC a adaptar son:

- Message signature: un mensaje m en SP tiene diferente nombre o tipo de parámetro en la interfaz del adaptador SC.
- Message split/merge: un mensaje m en SP se corresponde con un conjunto de mensajes m_1,m_2,...,m_n en SC, o viceversa.
- Missing/extra messages: uno o más mensajes en SP no tienen correspondencia en SC, o viceversa.
- Message ordering: la definición del protocolo de SP puede esperar un mensaje m en un orden distinto respecto a lo enviado por SC, o viceversa.

Además existen dos subtipos de incompatibilidad de orden:

- *Unespecified reception*: en el que una parte envía un mensaje mientras la otra no espera recibir ninguno.
- *Deadlock*: es el caso cuando ambas partes esperan recibir un mensaje de la otra parte.

En la Figura 3.7 se observa un ejemplo de como un adaptador permite el cambio de un proveedor por otro con funcionalidades similares pero con interfaces y protocolos diferentes.

Un adaptador para el caso *Unespecified reception* puede automáticamente manejar dicha situación utilizando un *buffer* para los pedidos y solicitar los mismos una vez que se han detectado pedidos con menor orden de precedencia según los protocolos. Sin embargo la adaptación de *Deadlock* es una tarea que requiere mayor esfuerzo.

El problema de la adaptación de servicios es importante a la hora de asistir al desarrollador en la comparación de servicios, identificar que tipos de incompatibilidades poseen y como desarrollar el adaptador. Los trabajos sobre la adaptación de servicios existentes, asumen que no existen incompatibilidades en sus interfaces y el código de correspondencia entre las mismas debe ser provisto por el desarrollador, además, si existen interacciones que conlleven a *deadlocks*, éstas son consideradas no adaptables, pero muchos de estos casos pueden llegar a adaptarse.

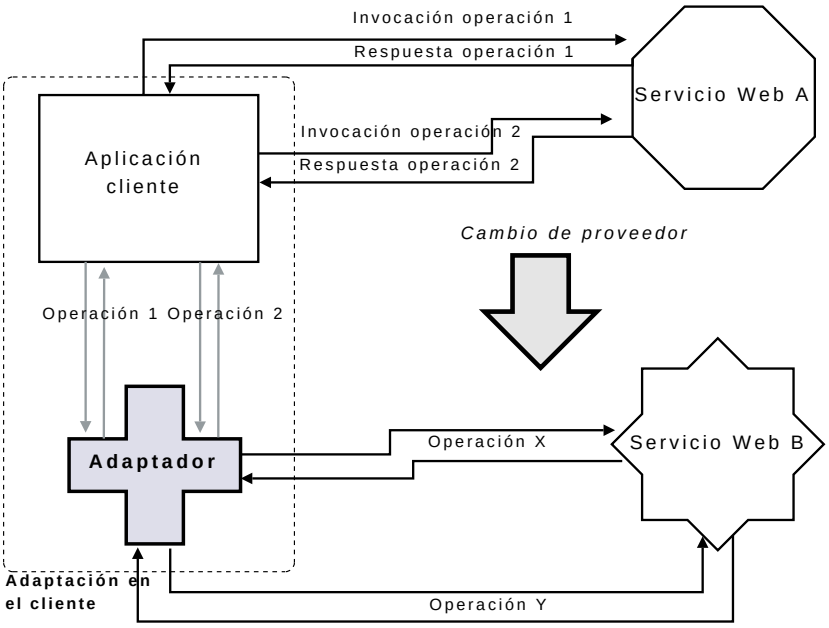


Figura 3.7: Ejemplo de adaptación entre servicios de similares funcionalidades y diferentes interfaces.

El modelo de adaptadores de servicios consiste en el mapeo de interfaces y un adaptador de protocolo. El adaptador de protocolo es un intermediario en el intercambio de mensajes entre los adaptadores de servicios, e instruye como se debe utilizar el mapeo de las interfaces antes y después de cada mensaje.

Además, el trabajo de [25] provee soporte semiautomático para la identificación de incompatibilidades y la generación de adaptadores. Haciendo las siguientes contribuciones:

- Provee un mecanismo semiautomático para identificar incompatibilidades a nivel interfaz e identificar correspondencias en los parámetros de entrada para resolver las mismas. Esto se logra a partir de enfoques sobre el mapeo de esquemas XML como se vio en la sección 3.1.2 , agregando información contextual al mismo.
- Provee un mecanismo automático de identificación de incompatibilidades a nivel protocolo, y generación de un adaptador si no existen *deadlocks*. Además, propone una forma de resolver las situaciones de *deadlock*. Para realizar esto genera un árbol llamado *Árbol de incompatibilidad* (Mismatch tree) para todas las incompatibilidades que resulten en *deadlock*. Este árbol provee una representación concisa de todos los *deadlock* y los mensajes incluidos en los mismos. La combinación de un árbol de representación de *deadlocks* y sugerencias de resolución ayudan al desarrollador al momento de generar el adaptador final.

3.3. Conclusión

En este capítulo se presentaron los trabajos, más relevantes, relacionados con mejorar tanto el descubrimiento de servicios Web como los mecanismos de incorporación de los mismos

en aplicaciones. Del análisis de éstos se observan un conjunto de características comunes, las cuales son:

- Los distintos trabajos de la sección 3.1 utilizan la tecnología existente de registros UDDI extendiéndola para suplir las problemáticas de descubrimiento que presenta la misma.
- El descubrimiento de servicios, debe contener algún tipo de mecanismo de comparación y puntuación, a fin de garantizar un *ranking* de servicios que permita obtener "el mejor candidato".
- La técnica VSM es ampliamente utilizada en el descubrimiento de servicios Web. La diferencia que radica en los distintos trabajos es, básicamente, qué información se recupera y cómo se compara.
- VSM asume que los nombres de los servicios, sus operaciones así también como sus comentarios, transportan palabras que son relevantes para entender qué ofrece un servicio. Esta particularidad ha sido investigada y demostrada con archivos de código fuente escritos en algún lenguaje de programación [40] y, dado que un documento WSDL refleja la funcionalidad de su implementación subyacente, intuitivamente parece válido asumir que se aplica para las descripciones de los servicios Web. Además, dado que un servicio es desarrollado para ser utilizado por terceros, generalmente se emplean buenas prácticas de programación y documentación para facilitar a un desarrollador externo entender cuál es la funcionalidad del servicio ofrecido [20].
- La calidad de los resultados obtenidos con un registro basado en VSM, depende también de cuan descriptivas sean las consultas utilizadas.
- Los distintos trabajos de la sección 3.2 descartan el patrón *proxy* como solución a la invocación de servicios. Cada uno de ellos propone soluciones aplicables en tiempo de desarrollo, ó bien, en tiempo de ejecución del software, que faciliten el mantenimiento de la aplicación. Para ello se utilizan técnicas que no "contaminen" el código funcional de las aplicaciones, como son el paradigma AOP, el uso de adaptadores o *brokers*.
- Las distintas técnicas de invocación de servicios, agregan tareas adicionales al desarrollador, cada una con un nivel de complejidad asociado, como la necesidad de aprender un nuevo lenguaje de programación orientado a aspectos, o generar archivos de configuración XML indicando cómo ensamblar los servicios a la aplicación.

Además de estas características, en el resto de este capítulo se puede observar que ningún trabajo aborda de manera completa y efectiva tanto las problemáticas asociadas al descubrimiento de servicios Web como las asociadas a la incorporación de los mismos en las aplicaciones, a excepción de [32], aunque el fuerte de este trabajo es la incorporación de los servicios y no ahonda en el descubrimiento de los mismos. No existe un enfoque que abarque las fases de desarrollo y mantenimiento del ciclo de vida para las aplicaciones orientadas a servicios. Con lo cual, una herramienta que facilite la implementación de aplicaciones orientadas a servicios, debe atacar los problemas individuales de la fase de desarrollo, sean descubrimiento e incorporación de servicios, para poder en conjunto abarcar dicha fase en su totalidad, sin dejar de lado los atributos de calidad de las aplicaciones desarrolladas.

CAPÍTULO 4

EasySOC

El enfoque EasySOC tiene como objetivo a largo plazo convertir a SOC en un paradigma establecido en lo referente a la programación distribuida. Para ello, a corto plazo es necesario simplificar el proceso de desarrollo y mantenimiento del software orientado a servicios, facilitando la tarea de tercerización de funcionalidad. En primer lugar, el descubrimiento y selección de servicios existentes no debe ser una tarea tediosa y que consuma tiempo a desarrolladores. En segundo lugar, el uso de servicios dentro de aplicaciones debería ser "lo menos intrusivo posible" a la lógica de la aplicación, con el fin de disminuir el esfuerzo de mantener la funcionalidad del lado del cliente una vez que ésta haya sido creada.

El enfoque EasySOC descripto en esta tesis se materializó en una solución orientada a la plataforma Java. Más precisamente, se provee una herramienta integrable a un entorno de desarrollo (también conocidos como IDE) con el fin de brindar facilidades a la hora de desarrollar aplicaciones orientadas a servicios. Los detalles de diseño e implementación de la misma se presentan en el capítulo 5.

Este capítulo está organizado como sigue: en la sección 4.1 se detallan las características generales del enfoque EasySOC; en la sección 4.2 se detalla el mecanismo de búsqueda de servicios propuesto por el enfoque EasySOC; mientras que en la sección 4.3 se detalla la incorporación de los servicios a aplicaciones orientadas a servicios utilizando el enfoque EasySOC.

4.1. Características del enfoque EasySOC

En la Figura 4.1 se puede observar el proceso general del enfoque EasySOC. Para ejemplificar, vamos a suponer que un/a desarrollador/a está construyendo una aplicación orientada a servicios. En un momento determinado, un asistente le acerca a él/ella una lista de

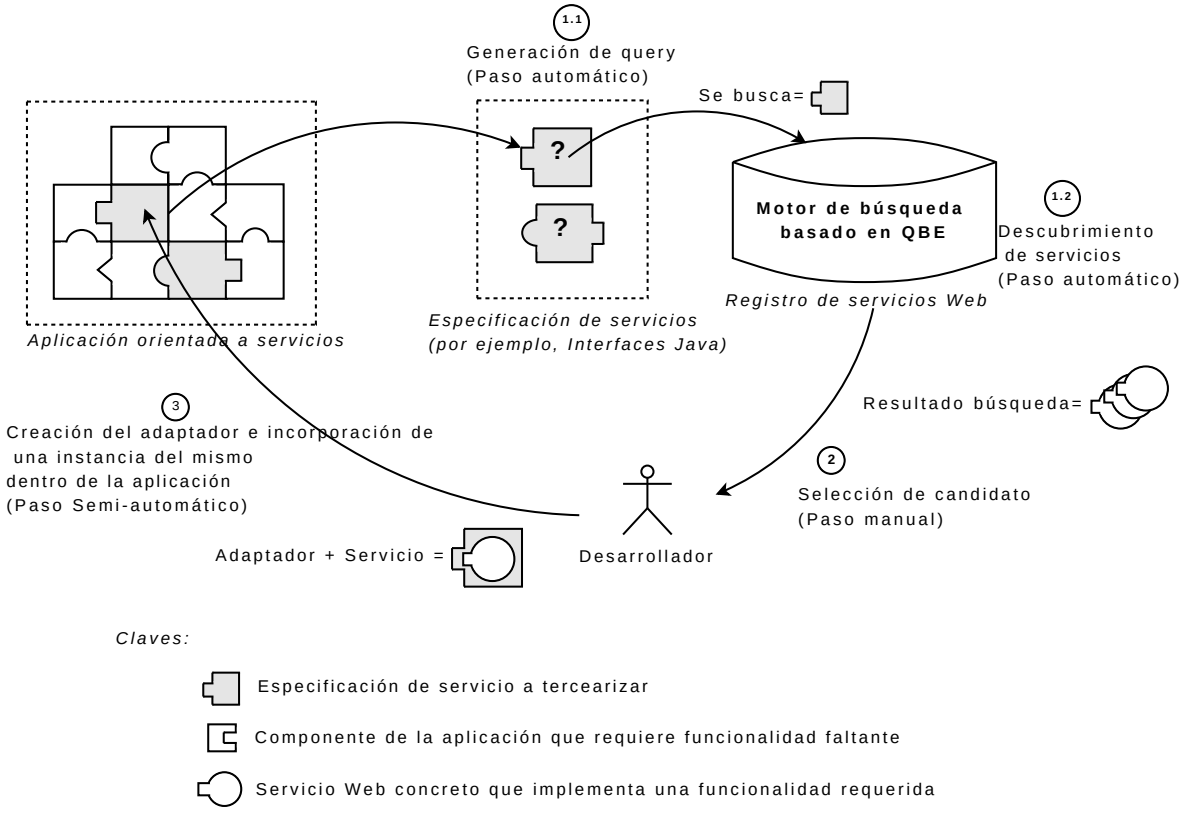


Figura 4.1: Vista general del enfoque EasySOC.

servicios que, potencialmente, podrían ser útiles para su aplicación. Además, una vez seleccionado uno de los servicios de la lista, el asistente genera el código para incorporar el candidato.

El proceso general propuesto por el enfoque EasySOC requiere que los diseñadores especifiquen la (potencial) interfaz del componente de la aplicación que se requiere tercerizar. Luego, EasySOC extrae información relevante del servicio deseado a partir del código fuente de la aplicación, incluyendo la mencionada interfaz y los comentarios de dicho código (Subpaso 1.1 en Figura 4.1), para buscar servicios similares utilizando un registro. EasySOC emplea WSQBE [4] para la búsqueda de servicios relevantes de terceros. Esencialmente,EasySOC utiliza WSQBE por los resultados positivos demostrados en [4] y porque trabaja con descripciones de servicios en WSDL, es decir, no requiere cambios. Lo novedoso de EasySOC con respecto al descubrimiento, son las técnicas de generación automáticas de consultas, las cuales "escarban" el código fuente de una aplicación que requiere servicios para mejorar la calidad de los resultados obtenidos al buscar en el registro de servicios.

La salida del proceso de descubrimiento será un conjunto de servicios cuyas operaciones, descriptas en sus documentos asociados (por ejemplo, un documento descripción WSDL), se asemejen a la signatura de las operaciones contenidas en la interfaz. En este sentido, el enfoque apunta a facilitar la tarea de descubrimiento de servicios, pero no se considera la selección automática de uno de ellos. Es decir, no se prescinde de la acción del/de la desarrollador/a durante el proceso de descubrimiento de servicios (Paso 2 en Figura 4.1).

Luego de la selección de los servicios candidatos realizada por el/la desarrollador/a, éstos son introducidos de manera no invasiva en la aplicación.El enfoque hace hincapié en la generación de código, con altos niveles de mantenibilidad, necesario para la invocación del servicio. Para ello, es central la idea de Inyección de Dependencias (DI) (ver sección 2.3). Con DI, servicios externos que proveen cierta funcionalidad pueden ser inyectados dentro de componentes de una aplicación que requieren de ese servicio, sin afectar la implementación de dichos componentes. Además, se combinan DI y el patrón de diseño Adapter [11] para establecer baja relación entre clientes y proveedores de servicios específicos. Es decir, que exista un bajo acoplamiento entre los componentes de la aplicación que invocan a componentes correspondientes a los servicios de terceros (Paso 3 en Figura 4.1). Una de las implicancias más importantes de ésto, es que permitirá cambiar un servicio por otro, que provee la misma funcionalidad, sin modificar el código del sistema que se está desarrollando.

En las siguientes secciones se detallan las dos etapas claves del proceso general del enfoque EasySOC: en la sección 4.2 se hace hincapié en el proceso de descubrimiento de servicios; mientras que en la sección 4.3 se trata la forma de introducir servicios de terceros dentro de una aplicación de una manera no invasiva y desacoplada.

4.2. Descubrimiento de Servicios

Como se explicó en la Sección 3.1.1, la precisión de las búsquedas usando registros basados en comparación sintáctica, tal es el caso de WSQBE, depende de cómo se describan los servicios y los *queries*. Por esto, el aporte central de EasySOC al descubrimiento de servicios consiste en generar *queries* descriptivos, de manera automática. La idea es aprovechar información que está tácitamente incluida en los componentes de un sistema orientado a

servicios, tanto los internos como los que se desean tercerizar. Por ejemplo, los nombres de las operaciones, las clases o los comentarios presentes en el código.

La anatomía de un sistema orientado a servicios consta de una serie de componentes internos que invocan a servicios externos. Durante el desarrollo de la aplicación cliente, el/la desarrollador/a diferencia los componentes que son implementados y aquellos que son tercerizados. Es decir, su implementación se delega a un servicio de un tercero. Cada componente a tercerizar es abstraído dentro de la aplicación cliente por medio de una interfaz que detalla las operaciones que brinda el componente, y los tipos de datos de sus argumentos. Se define al contexto como todo componente dependiente de la interfaz, es decir, aquellos componentes que invocan al componente tercerizado abstrayendo dicho comportamiento en la interfaz, y los argumentos contenidos en las operaciones definidas en la misma. En este sentido, EasySOC aprovecha la información descriptiva y programable contenida en la descripción "interna" del servicio y en el contexto en el que éste será utilizado, para presentarle al desarrollador una lista de servicios viables para implementar un componente tercerizado.

Uno de los mayores beneficios que brinda el enfoque EasySOC es la creación automática de *queries* a partir del código de la aplicación cliente. Por lo tanto, la tarea de descubrimiento de servicios utilizando el enfoque EasySOC es una tarea trivial ya que se provee generación de consultas de manera automática, por lo que el/la desarrollador/a sólo debe abocarse a detallar la interfaz del componente a tercerizar, y a partir de ella, el enfoque lo asistirá en búsqueda de servicios similares que, potencialmente, sirvan como implementación del componente en cuestión.

4.2.1. Generación automática de *queries* utilizando el enfoque EasySOC

El enfoque EasySOC apunta a la generación de *queries* a partir de la extracción de términos relevantes de la interfaz, que reflejen la funcionalidad del componente a tercerizar. Una interfaz comprende un nombre y una descripción de las operaciones que provee. Además, buenas prácticas de programación promueven a los desarrolladores a documentar el código fuente. Por lo tanto, una especificación de la interfaz consiste en una descripción textual de las partes que la constituyen (opcional) más la signatura de sus operaciones expuestas (obligatorio).

Las interfaces pueden contener términos que ayudan a representar la funcionalidad que estas definen. Definimos a estos términos como *relevantes* y al resto de los términos como *no relevantes* (como por ejemplo, palabras reservadas propias del lenguaje en que es escrita la interfaz). Extraer términos relevantes es una tarea importante ya que ayuda a crear *queries* más precisos, implicando el aumento en la precisión del mecanismo de descubrimiento. Por lo tanto, una tarea de preprocesamiento de la información es necesaria, a fin de descartar la mayor cantidad de términos no relevantes, o visto de otra forma, solamente considerar aquellos términos que sean relevantes para una buena descripción del servicio. El preprocesamiento propuesto por el enfoque EasySOC está compuesto por las siguientes cinco actividades:

1. Extracción de nombre y signatura de componentes.
2. Extracción de documentación.

3. Separación de nombres.
4. Eliminación de *stop words*.
5. *Stemming*.

Extracción de nombre y signatura de componentes

La primera actividad de preprocesamiento consta de obtener información relevante desde el código fuente de un componente de la aplicación cliente. Entre estos componentes, se incluye la interfaz de aquella funcionalidad a tercerizar.

La información útil que se extrae del código fuente es:

- Nombre del componente.
- Nombre de las operaciones definidas.
- Nombre de los tipos de datos de los parámetros de entrada y salida.
- Nombre de los parámetros de entrada.

Extracción de documentación

El código fuente de un componente puede incluir documentación que haya especificado el desarrollador. Por lo tanto, la segunda actividad del preprocesador es extraer los términos relevantes contenidos en la mencionada documentación.

Se puede distinguir dos tipos de documentación dentro de un código fuente, tal como se muestra en la Figura 4.2 :

- Documentación de cabecera: explica brevemente el fin del componente y detalla la estructura del mismo.
- Documentación funcional: relacionada a cada método declarado en la interfaz del componente, la cual explica la funcionalidad que brinda el método, además de detallar sobre los datos de entrada y de salida.

La estructura de la documentación mostrada en la Figura 4.2 se corresponde a la estructura propuesta por JavaDoc[1]. JavaDoc es una herramienta de Sun Microsystems[2] para generar APIs en formato HTML de un documento de código fuente Java. Además, es el estándar de la industria para documentar clases de Java. La mayoría de los IDEs los generan automáticamente, con lo cual, resulta muy sencillo para el desarrollador generar la documentación para la interfaz.

[1] http://java.sun.com/j2se/javadoc/
[2] http://www.sun.com/

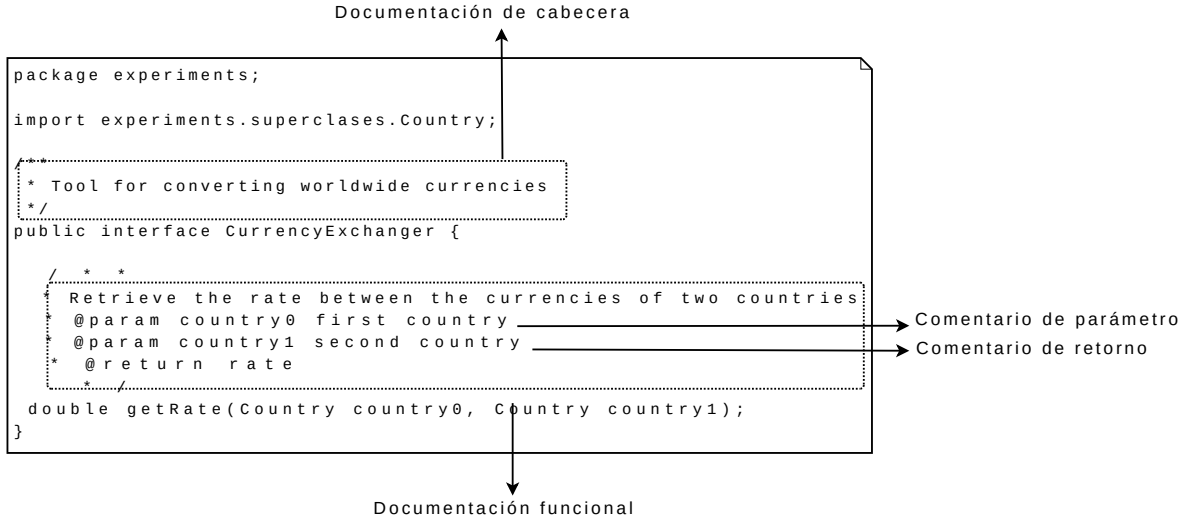


Figura 4.2: Ejemplo de documentación de una interfaz utilizando JavaDoc.

Una vez que se ha recuperado un conjunto de términos del código fuente de distintos componentes de la aplicación cliente, producto de haber aplicado las dos actividades presentadas previamente, el enfoque EasySOC propone tres actividades cuyo fin es refinar el conjunto de términos que pertenecerán al *query*. De está forma, se intenta crear un *query* exacto, cuyos términos reflejen la semántica del servicio deseado.

Separación de nombres

La separación de nombres permite lidiar con las distintas convenciones de nombres. En general, los desarrolladores combinar un verbo y un sustantivo para denotar los nombres de las operaciones, como puede ser *getQuote* o *get_quote*. Para poder lidiar con esta situación, esta técnica busca combinaciones de palabras y las divide en verbos y sustantivos, por ejemplo la palabra *getQuoteFor* será separada en las palabras *get*, *quote* y *for*. En la Tabla 4.1 se visualizan

Notación	Regla (Se divide ante)	Ejemplo Entrada	Ejemplo Salida
Java Beans	El cambio de “case”	getZipCode	get Zip Code
Hungarian	El cambio de “case”	ulAccountNum	ul Account Num
Símbolos especiales	La ocurrencia de “-” o “_”	get_Quote	get Quote

Cuadro 4.1: Reglas utilizadas para separar palabras combinadas.

las reglas utilizadas para realizar dicha división.

Eliminación de *Stop Words*

Stop Word es una palabra con un bajo nivel de utilidad en el contexto dado. La eliminación de símbolos y *Stop Words* tienden a limpiar el *query*. EasySOC utiliza una lista de 600 *Stop Word* para el idioma Inglés y una pequeña lista de *Stop Words* relacionadas al dominio de

servicios Web, como lo son *request*, *response*, *soap* y *post*. En general, la eliminación de símbolos especiales y *Stop Words* es una técnica simple pero de suma utilidad para filtrar términos considerados no relevantes. Por ejemplo, dado el conjunto de palabras *get*, *quote* y *for*, se descartan las *Stop Words get* y *for*, mientras que la restante *quote* se considera término relevante.

Stemming

EasySOC utiliza el algoritmo de *stemming* propuesto por *Porter [30],* que permite llevar cada palabra a su raíz, permitiendo eliminar de esta manera, las morfologías comunes presentes al final de cada palabra. El mencionado algoritmo reduce palabras del idioma Inglés. Por ejemplo, la raíz de los términos *stemmer*, *stemming*, *stemmed* es *stem*.

4.2.2. Expansión de *query*

El enfoque EasySOC no se limita a aplicar cada actividad de preprocesamiento sólo al código fuente correspondiente a la interfaz del componente a tercerizar, sino que aplica un concepto denominado "Expansión de *query*". La idea detrás del concepto es preprocesar los componentes de la aplicación cliente incluidos en el contexto de la mencionada interfaz. Con la "Expansión de *query*", se logra que el *query* incluya nuevos términos pertenecientes al dominio de la aplicación cliente. Para ello, se ha planteado dos procedimientos para la extracción de información útil del contexto.

El primero de ellos, se basa en obtener los argumentos de cada método de la interfaz, y recuperar la información útil asociada a cada uno de ellos, mediante las actividades de procesamiento previamente descriptas. Este procedimiento permite escalar en la jerarquía de super clases de estos nuevos componentes recuperados, buscando de manera recursiva más información relevante. En la Figura 4.3 se observa un ejemplo con sintaxis Java donde se muestran los términos extraídos producto del análisis de los argumentos del método *convertToFarhenheit* contenido en la interfaz *TemperatureUnitConverter*. Más precisamente, se muestra los términos recuperados desde la documentación del componente *TemperatureUnit* y de su superclase *Unit*.

El segundo procedimiento se basa en el análisis de los componentes que dependen del componente tercerizado. Esto implica que se aplique el preprocesamiento al código fuente de aquellos componentes que invocan a la interfaz (ejecutando algún método declarado en ella). Esto permite recuperar términos relevantes que reflejen el dominio donde se ejecutará el servicio Web a descubrir. De esta forma, el *query* posee información adicional que, potencialmente, sirve para refinar la búsqueda del servicio Web. Con el mismo criterio aplicado en el procedimiento anterior, se permite escalar tantos componentes dependientes como se desee, es decir, recuperar la documentación de un componente que invoca a otro componente, que a su vez, invoca al componente tercerizado, mediante la interfaz especificada por el desarrollador. En la Figura 4.4 se muestra un ejemplo con sintaxis Java que incluye una interfaz llamada *PDFCreator* y un componente llamado *TextProcessor* el cual depende de la interfaz mencionada. Es decir, el componente *TextProcessor* realiza una invocación al método *createPDF* de la interfaz. En el ejemplo, se muestra los términos recuperados a partir del análisis de la documentación de cabecera y funcional del componente dependiente.

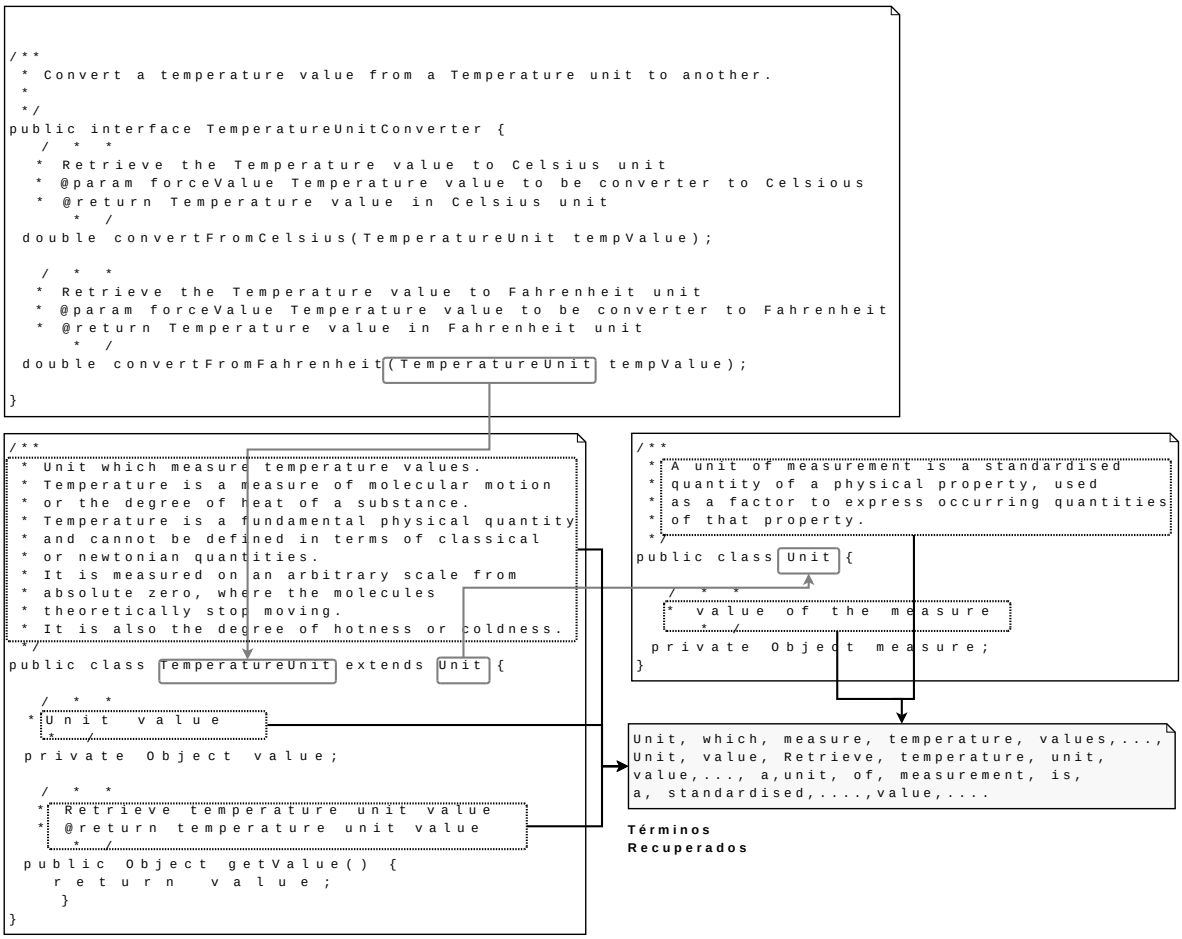


Figura 4.3: Ejemplo de extracción de documentación JavaDoc de los parámetros de una interfaz.

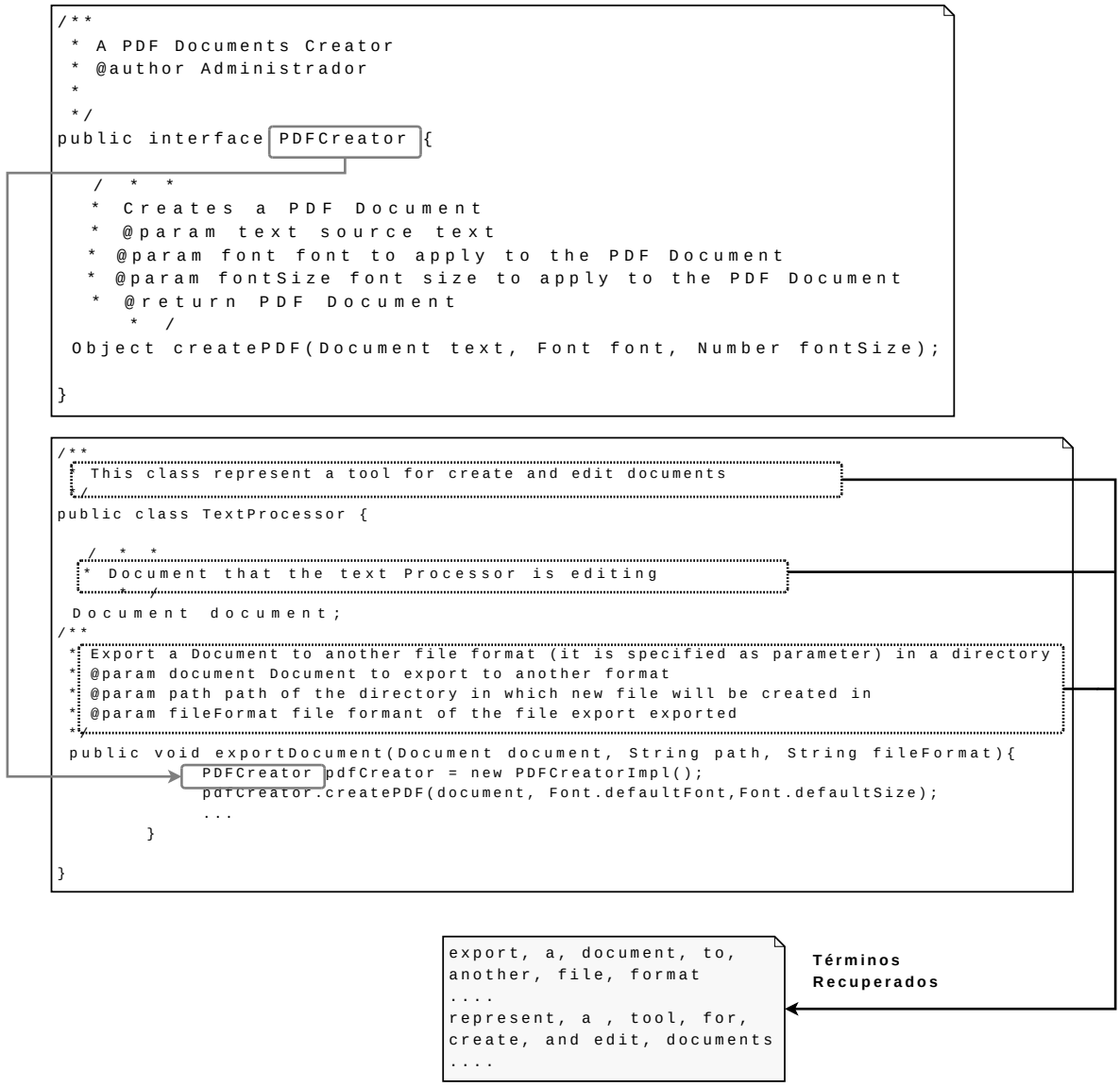


Figura 4.4: Ejemplo de extracción de documentación JavaDoc de un invocador de una interfaz.

A medida que se escala recursivamente en los componentes dependientes, puede aumentar la cantidad de términos recuperados que no pertenezcan al contexto del servicio Web a buscar (es decir, términos no relevantes), lo mismo puede suceder a medida que se escala en la jerarquía de super clases de los argumentos. Ante estas situaciones, EasySOC ofrece mecanismos para descartar aquellos componentes desarrollados fuera de la aplicación, es decir, provistos por el lenguaje de desarrollo ó por la utilización de distintos *frameworks*, y define topes a la cantidad de niveles a escalar. Además, EasySOC permite al desarrollador configurar la cantidad de niveles que se desean escalar en ambos casos, brindando la posibilidad de definir de manera manual el límite entre el contexto del servicio y el resto de la aplicación.

En la Figura 4.5 se observa el ejemplo de un buscador de libros y se detallan los resulta-

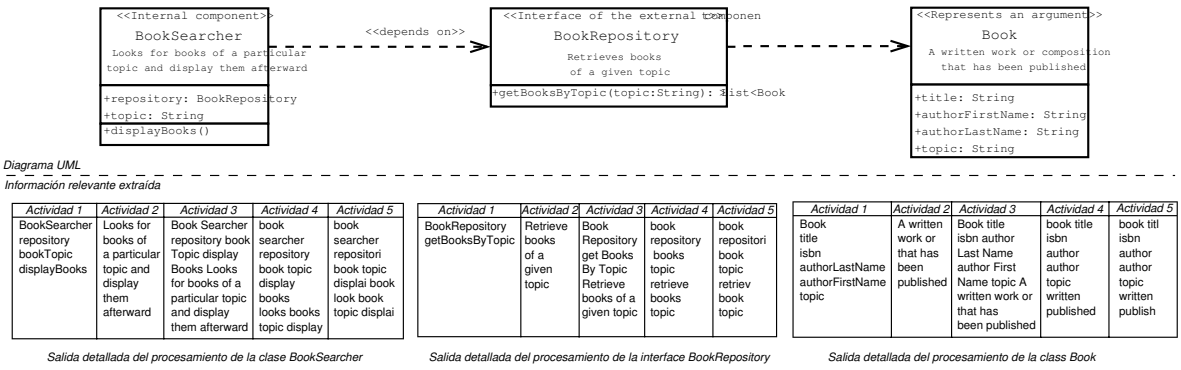


Figure 4.5: Resultado del preprocesamiento aplicado a un buscador de libros.

dos de cada actividad del preprocesamiento llevado a cabo por el enfoque EasySOC. En la mitad superior de la figura, se visualiza un diagrama UML con el diseño de (una parte de) la aplicación cliente desarrollada en Java. La clase *BookSearcher* depende de la interfaz *BookRepository*, la cual representa un servicio externo. Esta interfaz tienen un método llamado *getBooksByTopic*, el cual devuelve una lista de instancias de *Book*. El desarrollador posee una implementación del componente *BookSearcher* y *Book*, mientras que delega al enfoque EasySOC la implementación de la interfaz *BookRepository* mediante un servicio Web. Por lo tanto, el enfoque se encargará de generar el *query* de la búsqueda del servicio a partir de los términos relevantes que recupere desde los código fuentes de los tres componentes mencionados.

En la mitad inferior del diagrama, se detalla una tabla por cada componente contenido en el diagrama UML mencionado. Las celdas de cada tabla contienen la salida de cada actividad de preprocesamiento, respetando el orden de las actividades tal cual se han descrito previamente en este documento. El *query* final es conformado por la unión de los conjuntos de términos obtenidos durante la última actividad, correspondiente a cada componente procesado.

4.3. Incorporación de servicios candidatos

Luego que el/la desarrollador/a elije un servicio Web candidato, la tarea del enfoque EasySOC es incorporar el servicio seleccionado dentro de la aplicación cliente. Con este fin, EasySOC explota el patrón de Inyección de Dependencias (DI). La idea detrás de este concepto es establecer un nivel de abstracción entre los componentes de una aplicación (ya sean externos o internos) vía interfaces públicas, y lograr el desacople de componentes delegando responsabilidad de instanciación de componentes y vinculación al contenedor de una aplicación.

En una primer versión de una aplicación que consume servicios de terceros, (la cual no utiliza el principio de Inyección de dependencias) los componentes internos de la aplicación interactúan con las clases soporte del servicio Web. Estas clases soporte son necesarias para interactuar con el servicio en tiempo de ejecución. Estas clases son llamadas *Stub* o *Proxy* del servicio (ver sección 2.2.2). Es por ello que la lógica de la aplicación se encuentra mezclada con el código de configuración y uso de servicios Web. Por ejemplo, desde la lógica se realizan invocaciones explícitas al *Stub* de un servicio en particular. Como un *Stub* se encuentra asociado a un único servicio Web, el cambio de servicio Web involucra la generación de un nuevo *Stub* y, posteriormente, su incorporación dentro de la lógica de la aplicación. En general, ésto implica que se modifiquen aquellas partes del sistema que usaban al servicio original y, acto seguido, deban ser *testeadas* nuevamente. Por lo tanto, esta solución produce *software* que es complejo de mantener.

Por otra parte, se puede desarrollar un sistema orientado a servicios utilizando el principio de Inyección de dependencias. El código necesario para interactuar con el servicio es encapsulado dentro de un nuevo componente, y los parámetros de configuración correspondientes son ubicados en un archivo separado. Es responsabilidad del contenedor de Inyección de dependencias ensamblar en tiempo de ejecución los distintos componentes que forman la aplicación, ya sean componentes lógicos ó componentes "externos". El uso de Inyección de dependencias reduce el número de dependencias de clases concretas dentro de una aplicación, y produce un mejor diseño en términos de cohesión y extensibilidad. Intuitivamente, el código que implementa componentes lógicos es más fácil de reusar y testear mediante unidades de test, lo cual convierte a la aplicación más mantenible [4]. Empíricamente, se ha demostrado que software desarrollado utilizando el patrón DI tiende a tener menor acoplamiento que uno equivalente que no emplea DI [31], lo cual tiene un impacto directo en la mantenibilidad.

A pesar de los mencionados beneficios del uso de Inyección de dependencias en el desarrollo de aplicaciones orientadas a servicios, éste concepto no logra obtener software mantenible como se desea. Esto se debe a que los componentes lógicos de una aplicación cliente están "vinculados" a ciertos servicios específicos. El nexo de conexión entre ellos es la interfaz de cada servicio utilizado. De este modo, si se desea cambiar el proveedor de un servicio, se requiere adaptar la aplicación cliente al nuevo proveedor de servicios, ya que es altamente probable que ambos servicios posean distintas interfaces. A nivel de implementación, la mencionada adaptación implica reescribir manualmente la porción del código de aplicación que interactúa con la interfaz del servicio original, realizando los cambios necesarios para poder invocar al nuevo servicio.

Para superar este problema, el enfoque EasySOC redefine la idea de inyección de servicios Web introduciendo una capa intermedia que permite a los desarrolladores usar indistin-

tamente servicios diferentes (es decir, servicios de distintos proveedores y/ó servicios con distinta interfaz) en su aplicación. En lugar de inyectar directamente el *Stub* de un servicio Web dentro de la aplicación cliente, se inyecta un componente llamado *Service Adapter*.

4.3.1. Características generales del componente *Service Adapter*

Un componente *Service Adapter* es un adaptador de un servicio Web, inspirado en el patrón de diseño *Adapter* [11] ilustrado en la Figura 4.6 . Este patrón, también denominado *Wrapper*,

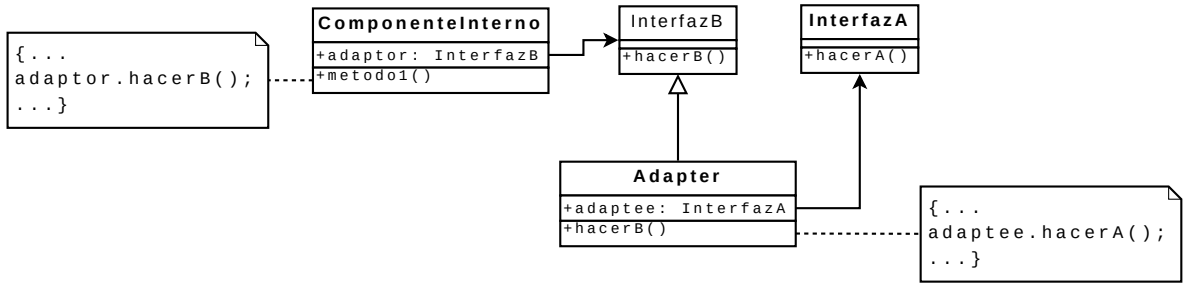


Figura 4.6: Patrón *Adapter.*

se encarga de adaptar la interfaz de una clase (por ejemplo, interfazA en Figura 4.6) a otra interfaz "esperada" por el usuario [11] (por ejemplo, interfazB en Figura 4.6).

En el enfoque propuesto, se propone la creación del componente *Service Adapter,* el cual es el encargado de adaptar la interfaz del servicio Web de un tercero (interfazA) a la interfaz del componente a tercerizar (interfazB). Esta última interfaz es aquella especificada por los desarrolladores en tiempo de diseño, la cual fue utilizada como entrada del proceso de descubrimiento de servicios, tal como se explicó en la sección 4.2. Además, el *Service Adapter* es quién interactúa directamente con el servicio Web, generalmente a través del *Stub* del servicio en cuestión, como se visualiza en la Figura 4.7. Los componentes internos de la aplicación

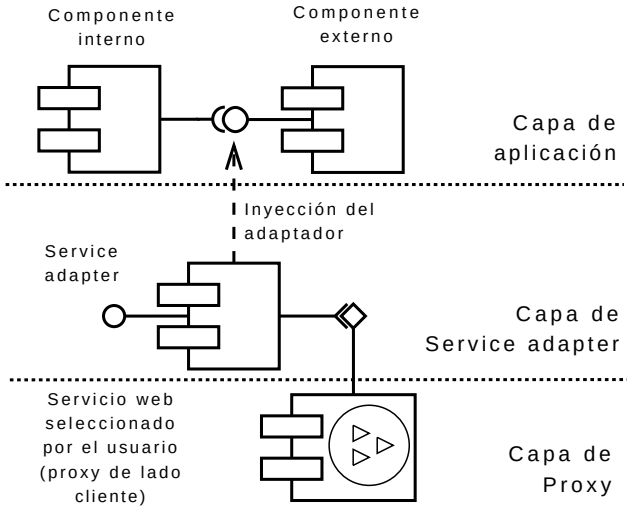


Figura 4.7: *Service Adapter* en EasySOC.

interactúan con los servicios tercearizados por medio de las interfaces situadas en la *Capa de aplicación*. La implementación de cada una de estas interfaces, es relegada a la *Capa de Service Adapter* encargada de lidiar con las características particulares de los proveedores, instanciados como *Stubs* de los servicios Web en la *Capa de Proxy*. Este diseño de capas, permite que las capas inferiores puedan ser cambiadas fácilmente sin modificar la funcionalidad de la capa de aplicación, al mismo que tiempo que, reemplazar una capa en el ámbito DI implica sólo modificar la capa inyectada por otra y definir esta nueva dependencia en un archivo de configuración externo.

En enfoques tradicionales, decidir cuál es la mejor operación dentro de una especificación de un servicio Web para un cierto requerimiento (por ejemplo, un método de la interfaz creada por el desarrollador) es una tarea que el/la desarrollador/a debe realizar observando exhaustivamente la especificación del servicio Web en cuestión, y de acuerdo a su criterio, generar el código de adaptación correspondiente. Es decir, la adaptación de interfaces es una tarea completamente manual.

Utilizando el enfoque EasySOC, el/la desarrollador/a delega al enfoque el proceso de creación de una capa intermedia, materializada mediante el componente *Service Adapter*. Dicho proceso de creación consta de dos etapas: en la primera de ellas se realiza una adaptación automática entre la interfaz cliente y la interfaz del servicio de terceros. En la segunda etapa, se genera el código del *Service Adapter* a partir de la adaptación acordada en el paso previo. Estas etapas se describen debajo.

Adaptación de interfaces

El proceso de adaptación de interfaces implica hallar correspondencias entre los elementos de dos interfaces de acuerdo a la similitud entre estos elementos. En el contexto del presente trabajo, la adaptación se realiza entre la interfaz cliente y la interfaz del servicio Web de terceros a integrar, y los elementos de las interfaces incluyen a métodos/operaciones y sus parámetros, además de los distintos tipos de datos de entrada y de salida de cada método/-operación.

Este proceso de adaptación consiste en la búsqueda, por cada método de la interfaz cliente, de la operación en la interfaz del servicio Web externo con mayor similitud. Se dispone de numerosos criterios para calcular el grado de similitud entre ellos. Por ejemplo, se puede comparar los tipos de datos de sus datos de entrada y salida, los nombres de las operaciones, documentación asociada, como así también una ponderación entre dos ó más de los criterios mencionados.

Una vez que se conoce la operación del servicio Web que implementará el método de la interfaz cliente, es necesario hallar una correspondencia entre los datos de entrada de la operación de la interfaz cliente y los datos de entrada de la operación del servicio Web. A su vez, también se requiere obtener una correspondencia entre los datos de salida del servicio Web y los datos de retorno del método de la interfaz.

EasySOC reutiliza y extiende los conceptos presentados en la sección 3.1.2 a fin de comparar semántica y estructuralmente dos interfaces de servicios. A diferencia de los algoritmos propuestos por [37, 38], EasySOC se basa en un enfoque híbrido donde se ponderan tanto las correspondencias estructurales de las operaciones y sus argumentos, como las

correspondencias semánticas de sus nombres. Además, EasySOC permite la sobrecarga de operaciones, es decir, una operación de un servicio puede corresponder a una o más operaciones en el segundo servicio. El algoritmo para hallar la mejor correspondencia es como sigue:

La entrada del proceso son dos interfaces de servicios a comparar, especificadas formalmente en algún lenguaje en común como puede ser WSDL. El primer paso consiste en obtener las operaciones expuestas en cada especificación. Luego, se compara cada una de las operaciones pertenecientes a uno de los servicios con cada una de las operaciones pertenecientes al segundo servicio. La comparación entre dos operaciones retorna un valor cuantitativo que representa la similitud entre ambas operaciones. Para obtener un valor cuantitativo de la similitud de dos servicios, se seleccionan aquellos pares de operaciones comparados previamente que maximicen la suma de sus valores de similitud, tal como se muestra en el Algoritmo 1 . La sumatoria del valor de similitud de estos pares seleccionados corresponde

procedure comparacionWebServices(*servicio*1, *servicio*2)

```
1: m ← número de operaciones en servicio1
2: n ← número de operaciones en servicio2
3: lista1 ← lista de operaciones en servicio1
4: lista2 ← lista de operaciones en servicio2
5: matrizOperacion ← contruirMatriz(m,n)
6: for i = 0 to m do
7:    for i = 0 to n do
8:       matrizOperacion[i][j] ← compararOperaciones(lista1[i],lista2[j])
9:    end for
10: end for
11: return pares de operaciones que maximizan la similitud de los servicios de acuerdo con
    matrizOperacion
```

Algorithm 1: Algoritmo de correspondencia de servicios.

al valor de similitud de los dos servicios comparados.

Para obtener un valor de similitud entre dos operaciones, se comparan los mensajes de entrada y de salida de ambas operaciones y sus nombres. La similitud entre dos operaciones (en términos cuantitativos) está dada por la sumatoria entre los valores de similitud de los mensajes de entrada, salida (estructuralmente) y el valor de similitud de sus nombres (semánticamente), tal como se muestra enel Algoritmo 2 . Cabe destacar que la sumatoria

procedure compararOperacioones($op1, op2$)

```
1: score ← P * (compararMensajes(op1.input, op2.input) + compararMensajes(op1.output, op2.output)) + ((1 - P) * compararNombres(op1.nombre, op2.nombre))
2: return score
```

Algorithm 2: Algoritmo de correspondencia *de operaciones.*

se encuentra ponderada, es decir, existe un valor P que define la importancia estructural y semántica de los términos de la sumatoria, en particular, asignando el valor 1 a P se define

una correspondencia puramente estructural y asignando el valor 0 se define una correspondencia puramente semántica, como se demuestra en el capítulo 6, el valor de *P* considerado por EasySOC es 0 al utilizarse como filtro de descubrimiento y cercano a 1 al utilizarse para detectar cual servicio es más fácil de adaptar.

Comparar dos nombres de operaciones implica comparar los términos relevantes que los conforman. Para ello, el nombre de la operación es preprocesado utilizando algunas de las actividades mencionadas en la sección 4.2.1, como son Separación de nombres y Eliminación de *StopWords*. Como resultado se obtiene dos conjuntos de términos relevantes. Luego, se procede a comparar cada término del primer conjunto con cada uno del segundo conjunto con el objetivo de determinar la similitud entre el par de términos. El valor cuantitativo de similitud de dos nombres de operaciones se calcula a partir de seleccionar aquellos pares de términos que maximicen el valor de similitud. Para poder obtener un valor cuantitativo de la similitud de dos términos, se debe hallar la relación semántica entre ellos. Es decir, determinar si los términos son iguales, sinónimos, hipónimos, hiperónimos, ó bien, si no existe relación semántica entre ellos. A partir de ello, se retorna un valor numérico que exprese esa relación, el cual varía desde el valor MAX_SCORE (los dos términos son iguales) hasta el valor MIN_SCORE (No existe relación semántica entre ellos). La Tabla 4.2 detalla los valores

Relación semántica	Valor numérico
Iguales	MAX_SCORE
Sinónimos	MAX_SCORE * 0.8
Hipónimos	MAX_SCORE * 0.6
Hiperónimos	MAX_SCORE * 0.6
Sin relación	MIN_SCORE

Cuadro 4.2: Valores númericos asignados a la relación semántica de dos términos.

númericos que definen la relación semántica entre dos términos.

Comparar dos mensajes, ya sea ambos de entrada ó de salida, implica comparar los tipos de datos asociados a los parámetro de cada mensajes, tal como se muestra en el Algoritmo 3

procedure -compararMensajes(*msg*1, *msg*2)

1: lista1 ← lista de tipos de datos asociados al mensaje *msg*1
2: lista2 ← lista de tipos de datos asociados al mensaje *msg*2
3: score ← compararTiposDeDatos(lista1,lista2)
4: **return** score

Algorithm 3: Algoritmo de correspondencia de mensajes.

. En primer lugar se compara cada tipo de datos de un mensaje con cada tipo de datos del restante mensaje, formando un conjunto de pares de tipos de datos. El valor de similitud de dos mensajes se calcula a partir de seleccionar aquellos pares que maximicen el valor de similitud.Por ejemplo, si se tienen dos mensajes: uno denominado *M1*cuyas partes referencian a dos tipos de datos *T1*y *T2*, y un segundo mensaje denominado *M2*cuyas partes referencian a tipos de datos *T3*y *T4*. Para obtener un valor cuantitativo de similitud entre los mensajes *M1*y *M2*, se deben obtener los valores de similitud entre los tipos de datos de *M1*y *M2*. Por lo tanto, se realizan cuatro comparaciones: (*T1*y *T3*con valor de similitud en-

tre las operaciones equivalente al valor *V1*), (*T1*y *T4*con valor *V2*), (*T2*y *T3*con valor *V3*) y (*T2*y *T4*con valor *V4*). Suponiendo que los valores poseen la propiedad V2>V1>V3>V4, el subconjunto de pares que maximiza el valor de similitud de los mensajes es (*T1*,*T4*con valor *V2*) y (*T2*,*T3*con valor *V3*). Por lo tanto, el valor de similitud de los mensajes *M1*y *M2*será la suma de los valores *V2*y *V3*. Es importante destacar que cada tipo de datos es incluido en algún par del mencionado subconjunto, a lo sumo, una sola vez.

El valor de similitud de dos mensajes se obtiene a partir de un algoritmo recursivo, cuya entrada son dos listas con los tipos de datos asociados a los respectivos mensajes. Se muestra un pseudocódigo en el Algoritmo 4 . Este algoritmo se basa en la siguiente heurística:

procedure compararTiposDeDatos($sourceList(m), targetList(n)$)

```
1: matriz ← contruirMatriz(m,n)
2: for i = 0 to m do
3:    for i = 0 to n do
4:       sourceType ← sourceList(i);
5:       targetType ← targetList(j);
6:       if sourceType y targetType son primitivos then
7:          matriz[i][j] ← -compararTiposPrimitivos(sourceType,targetType)
8:       end if
9:       if sourceType y targetType comparten el mismo nombre y namespace then
10:         matriz[i][j] ← -compararTiposIdenticos(sourceType,targetType)
11:      end if
12:      if sourceType ó targetType son complejos then
13:         nuevaSourceList ← obtenerElementosDeDatosCompuesto(sourceType)
14:         nuevaTargetList ← obtenerElementosDeDatosCompuesto(targetType)
15:         matriz[i][j] ← compararTiposDeDatos(nuevaSourceType,nuevaTargetType) + or-
            ganizationBonus(sourceType,targetType)
16:      end if
17:   end for
18: end for
19: return pares de tipos de datos con puntaje máximo de acuerdo con matriz
```

Algorithm 4: Algoritmo de correspondencia de tipo de datos.

- Dos tipos de datos simples son comparados en base a sus tipos de programación y a sus identificadores asociados: La medida cuantitativa de la similitud entre dos tipos de datos simples, se obtiene por medio de una suma ponderada entre el valor de similitud del tipo de programación y el valor similitud entre sus identificadores. El valor del primer término de la suma se determina de acuerdo al grado de compatibilidad entre los tipos de datos, si son plenamente compatibles (sin pérdida de información), semi-compatibles (con pérdida de información) ó incompatibles. De acuerdo a ello, la comparación de dos tipos de datos simples es valuada de acuerdo a la Tabla 4.3. El valor del segundo término, se obtiene a través de comparar los términos relevantes correspondiente a los identificadores, este procedimiento es idéntico a aquel utilizado en la comparación de nombres de operaciones descripto anteriormente.

Relación de compatibilidad de datos	Valor de correspondencia numérico
Compatibles sin pérdida	MAX_SCORE
Compatibles con pérdida	MAX_SCORE * 0.5
Incompatibles	MIN_SCORE

Cuadro 4.3: Valor de correspondencia numérico asociado a la relación entre dos tipos de datos.

- Dos tipos de datos complejos son comparados en base a los elementos que los constituyen y a la organización grupal entre ellos: el valor de similitud entre dos tipos de dato, de los cuales al menos uno de ellos es complejo, se corresponde al valor de similitud que existe entre los tipos de datos que componen cada tipo complejo involucrado en la comparación. Para ello, se realiza una invocación recursiva al algoritmo en cuestión, donde cada lista de entrada corresponde a los tipos de datos contenidos en cada complejo. Puede existir el caso en el que uno de ellos no sea complejo, y por lo tanto sea un tipo simple (primitivo). En este caso, la lista de entrada del llamado recursivo incluirá al mencionado tipo de dato simple. Esto implicará que este dato simple sea comparado con cada uno de los tipos de datos contenidos en el tipo de datos complejo.
- Dos tipos de datos complejos, importados desde el mismo *namespace*, son considerados idénticos si tienen el mismo nombre. En este caso el valor de correspondencia es aquel que representa una máxima compatibilidad entre los tipos de datos.

La salida del algoritmo, a su vez es utilizada como salida del proceso de adaptación de interfaces, está compuesta por un valor cuantitativo de similitud entre dos servicios más información que describe cómo se ha alcanzado dicho valor. Es decir, esta información incluye cuáles pares de operaciones han maximizado ese valor de similitud, y por cada par de operaciones, cuál es la correspondencia "ideal" entre los tipos de datos asociados a estas operaciones, que también está dado por un conjunto de pares de tipos de datos. Cabe destacar que se diferencian las correspondencias para los datos de entrada del par de operaciones de aquellas correspondiente a los datos de salida. Esta información de correspondencias es utilizada en la etapa de creación del componente *Service Adapter*.El enfoque EasySOC representa esta información mediante un lenguaje estructurado, por ejemplo XML. Esto implica que el desarrollador pueda observar el resultado del proceso de adaptación de interfaces y editarlo antes de ser consumido por la siguiente etapa. De esta forma, el desarrollador puede tener inferencia durante el proceso de adaptación, y transitivamente, durante la etapa de generación de código del *Service Adapter*(Paso 3 en Figura 4.1).

Generación de Código

El componente *Service Adapter* es una implementación de la interfaz especificada por el desarrollador, la cual detalla las operaciones a tercerizar. Esto permite que el componente pueda ser inyectado dentro de la aplicación cliente sin tener que realizar ningún cambio en ella. Por lo tanto, la materialización del componente *Service Adapter* se logra creando un componente que implemente la mencionada interfaz. Por otro lado, el *Service Adapter* posee una referencia a los componentes que interactúan con un servicio Web específico, es decir, el *Stub* del servicio. Esto permite que el *Service Adapter* actúe como una capa intermedia entre la

aplicación cliente y un servicio Web específico (representado por el componente *Stub*). De este modo, ante un cambio de servicio Web, sólo se tiene que reescribir el código del *Service Adapter*, mientras que la aplicación cliente no requiere ser modificada.

La generación del código del *Service Adapter* implica que el enfoque EasySOC genere código por cada método declarado en la interfaz cliente. Dentro de cada método del *Service Adapter* se identifican tres segmentos de códigos. En la Figura 4.8 se muestra un ejemplo de un com-

```
package matematicas;

public class Matematicas_Adapter implements IMatematicas {

 servicio.ICalculadoraPortType proxy = null;

   public java.lang.String[] getSumaConvertida(java.lang.String in0) {
   double param0 = Double.valueOf(in0).doubleValue();
     double param1 = 0.0;
     double return1 = 0.0;                                          A
    try {
        return1 = getProxy().sumarDoubles(param0, param1);
    } catch (Exception e) {
       e.printStackTrace();
       throw new RuntimeException(e);
    }                                                               B
     java.lang.String[] getSumaConvertidaReturn = new java.lang.String[1];
     getSumaConvertidaReturn[0] = Double.toString(return1);                 C
     return getSumaConvertidaReturn;
   }

   public void setProxy(servicio.ICalculadoraPortType proxy) {
      this.proxy = proxy;
   }

   public servicio.ICalculadoraPortType getProxy() {
      return this.proxy;
   }
}
```

Figura 4.8: Código ejemplo de un componente *Service Adapter.*

ponente *Service Adapter* escrito en el lenguaje Java, en donde se muestran los mencionados segmentos de código delimitados por los recuadros A, B y C.

El primer segmento, delimitado por el recuadro A de la Figura 4.8, incluye código correspondiente a la lógica necesaria para transformar los datos de entrada del método de la interfaz cliente a los datos de entrada de la operación del servicio Web correspondiente a ese método. Este código de trasformación debe incluirse antes de realizar la invocación, a través del *Stub*, de la operación del servicio Web. Para generar este segmento de código es necesaria la información de correspondencia entre datos obtenida durante el proceso de adaptación de interfaces.

El segundo segmento de código generado, delimitado por el recuadro B de la Figura 4.8, incluye código de invocación a una operación del servicio Web de terceros. Esto implica la inclusión de código para la invocación de un método del *Stub* referenciado por el *Service*

Adapter. La adaptación de interfaces realizada en la etapa previa indica cuál operación del servicio Web es la que se debe invocar en cada método de la interfaz cliente.

Por último, el tercer segmento de código, delimitado por el recuadro C de la Figura 4.8, incluye código para transformar los datos de salida de la operación del servicio Web en los datos de salida del método de la interfaz cliente luego de haber realizado la invocación a la operación del servicio Web. Este segmento de código también es generado a partir de la información obtenida durante el proceso de adaptación de interfaces (Paso 3 en Figura 4.1).

EasySOC provee mecanismos de generación automática del código del *Service Adapter*, como así también del *Stub* asociado al servicio Web. Además, genera automáticamente los artefactos de configuración necesarios, a fin que el contenedor de Inyección de dependencias pueda inyectar el *Stub* en el *Service Adapter* y el *Service Adapter* dentro de los componentes de aplicación dependientes. Otra gran ventaja que posee EasySOC respecto a los trabajos presentados en el capítulo 3, es la generación automática de test de unidad para las tres capas mencionadas con anterioridad (Capa de aplicación, capa de *Service Adapters* y capa de *Proxy*).

4.4. Conclusión

El enfoque EasySOC apunta a facilitar el desarrollo de aplicaciones orientadas a servicios. Es por ello que provee soluciones concretas durante etapas críticas del ciclo de vida de tales aplicaciones: descubrimiento de servicios, incorporación de los mismos a la aplicación en desarrollo y mantenimiento.

Utilizando el enfoque EasySOC, se reduce significativamente la complejidad de la tarea de descubrimiento de servicios. Esto se debe a que el proceso de descubrimiento propuesto por el enfoque es generar automáticamente consultas que incluyen palabras relevantes y no requiere de información adicional a la contendida dentro del código fuente de la aplicación. Este aspecto es una de las claves del enfoque, debido a que el desarrollador no debe realizar ningún esfuerzo adicional a la tarea de especificar e implementar los componentes de la aplicación, debiendo ser consiente que las buenas prácticas de programación, la documentación del código y la utilización de nombres representativos en las variables contribuirán a mejorar la precisión del proceso de descubrimiento. Esto implica una reducción notable en el tiempo de desarrollo de la aplicación orientada a servicios.

Ante la necesidad de tercerizar un componente de la aplicación en desarrollo, el enfoque acerca al desarrollador un listado con los servicios que, potencialmente, pueden ser incluidos dentro de la aplicación. Luego, el desarrollador puede seleccionar uno de estos servicios, con el fin de incorporarlo dentro de la aplicación. El enfoque EasySOC brinda asistencia en la tarea de incorporación, generando el código necesario para poder incluirlo. Es por ello que el enfoque crea una capa de adaptación de servicios. Esta capa contiene el código necesario para adaptar la interfaz servicio seleccionado a la interfaz del componente tercerizado. De este modo, la incorporación del servicio a la aplicación no requiere realizar cambios en ninguno de los componentes de la aplicación previamente elaborados.

El uso de la capa de adaptación de servicios en una aplicación orientada a servicios implica que el reemplazo de servicios tercerizados dentro de una aplicación cliente no afecta el código de la misma. En tal circunstancia, sólo se requiere generar un nuevo *Service Adapter*

correspondiente al nuevo servicio, por lo cual los componentes internos de la aplicación no deben ser modificados. Esta es la clave para lograr altos niveles de mantenibilidad, dado que agregar/cambiar un servicio sólo implica crear una clase (el adaptador) sin modificar las clases existentes de la aplicación como ocurre con otros enfoques.

Además de reducir el acoplamiento entre componentes internos de una aplicación y los servicios tercerizados, este enfoque permite a los desarrolladores diseñar, implementar, y testear código de los componentes de aplicación y luego, hacer foco en la incorporación de la funcionalidad tercerizada. Además, esta separación puede traer beneficios adicionales más allá de la calidad del software, y contribuye a mejorar el proceso de desarrollo en si mismo, porque estos dos grupos de tareas pueden desarrollarse independiente por dos grupos de desarrollo diferentes.

A modo de resumen, el enfoque EasySOC posee características que lo distinguen de los trabajos relacionados presentados en el capítulo anterior. En primer lugar, ninguno de los trabajos mencionados cubre plenamente proceso de desarrollo de aplicaciones orientadas a servicios. Cada uno de estos ofrecen soluciones que apunten al descubrimiento de servicios, ó bien, a la adaptación y/ó integración de servicios. En cambio, el enfoque EasySOC complementa el proceso de descubrimiento de servicio, sin demandar que el desarrollador deba adquirir nuevos conocimientos tecnológicos generando el código fuente del componente *Service Adapter*, de manera automática. Para ello, el desarrollador debe haber especificado la interfaz del componente a tercerizar (componente externo) y haber hallado el servicio Web que implementará la funcionalidad especificada en la mencionada interfaz, pudiendo haber utilizado o no el proceso de descubrimiento de servicios brindado por EasySOC.

En el siguiente capítulo se presenta el diseño y la implementación de una herramienta integrable al entorno de desarrollo Eclipse[3] que materializa el enfoque EasySOC. Por lo tanto, el desarrollo de aplicaciones distribuidas en el mencionado entorno implica la reducción en la complejidad en las tareas de desarrollo, produciendo software con altos niveles de modificabilidad y reusabilidad.

[3] http://www.eclipse.org/

CAPÍTULO 5

Diseño e implementación

En este capítulo se presenta una herramienta que implementa el enfoque EasySOC denominada EasySOCPlugin. La particularidad de la herramienta es que se encuentra desarrollada como una extensión del entorno de desarrollo Eclipse[1]. Aplicaciones con tal característica se las denomina *Plugin*[2]. La idea central del trabajo es proveer una herramienta embebida en el entorno Eclipse, que brinde asistencia a desarrolladores durante las etapas de desarrollo y mantenimiento de aplicaciones orientadas a servicios en la plataforma Java, implementando el enfoque EasySOC.

La herramienta hace hincapié en proveer interfaces gráficas amigables, cuya función sea asistir al desarrollador en la búsqueda e incorporación de servicios externos dentro de las aplicaciones. Además, la herramienta genera de manera automática casos de test de unidad para todo el código generado, facilitando la verificación y correctitud del mismo.

Este capítulo está organizado como sigue: en la sección 5.1 se detalla la arquitectura general de la herramienta; en la sección 5.2 se detalla la implementación del mecanismo de búsqueda de servicios; en la sección 5.3 se detalla la adaptación de un servicio externo a la interfaz definida por el desarrollador/a; en la sección 5.4 se detalla la incorporación de los servicios dentro de la aplicación cliente.

5.1. Arquitectura general

La herramienta se encuentra desarrollada bajo el concepto de Filtros (Pipe and Filter [2]), donde un filtro es un componente de software que dada una entrada le aplica un procesamiento y retorna una salida refinada. Bajo este concepto, diferentes filtros son ensamblados en forma de tubería conformando la aplicación en su totalidad. Este tipo de diseño permite que la herramienta pueda ser modificada, simplemente reemplazando o adicionando

[1] http://www.eclipse.org/
[2] http://www.eclipseplugincentral.com/

nuevos filtros a la misma. Los filtros se encuentran separados en tres módulos, con responsabilidades bien definidas. Las responsabilidades de cada módulo son:

- Generación de *queries* y búsqueda de servicios Web.
- Adaptación del servicio Web a la interfaz cliente.
- Incorporación del servicio adaptado a la aplicación cliente.

Sobre cada módulo, residen los componentes encargados de interactuar con el entorno de desarrollo Eclipse. Estos componentes no solo proveen las interfaces gráficas de la aplicación, sino también el motor de búsqueda de dependientes y argumentos de la interfaz a tercerizar.

En la Figura 5.1 se puede observar la arquitectura básica de la aplicación, y en la Figura 5.2

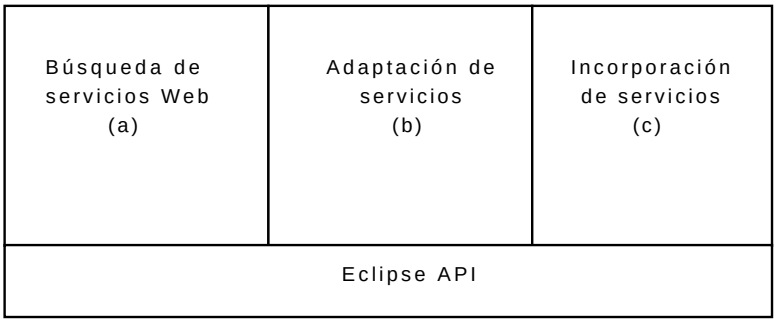


Figura 5.1: Arquitectura básica.

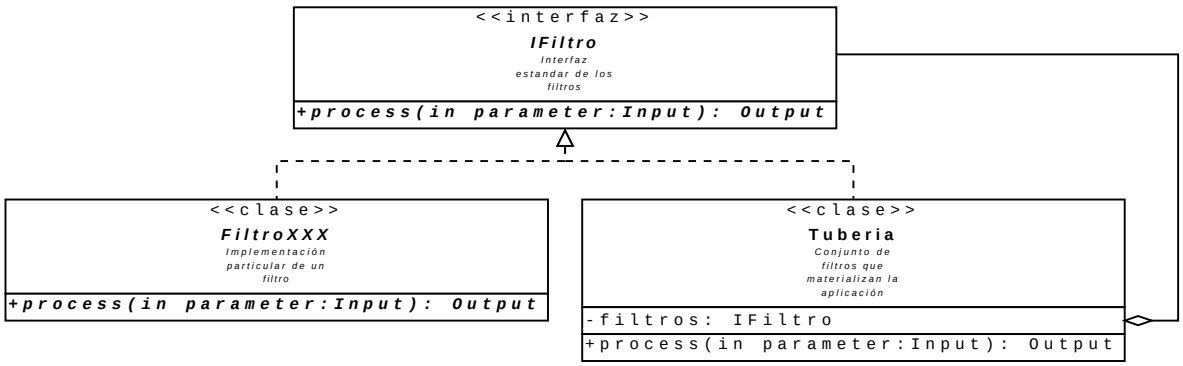


Figura 5.2: Implementación de la aplicación utilizando Filtros.

la implementación propuesta para los filtros.

La comunicación entre los módulos sigue el flujo de descubrimiento e incorporación de servicios presentado en la sección 4.1. Dada una aplicación cliente que defina en una interfaz la funcionalidad a tercerizar, el módulo (a) es el encargado de generar el *query* de búsqueda y retornar una lista de servicios candidatos. El desarrollador tiene la posibilidad de seleccionar uno de estos servicios con el fin de integrarlo a la aplicación bajo desarrollo. Para ello

se crea una capa intermedia entre la aplicación en desarrollo y el servicio seleccionado denominada *Service Adapter*. El módulo (b) toma como entrada la especificación de un servicio externo y la interfaz del componente interno a tercerizar, y devuelve la especificación formal de correspondencias entre ambas funcionalidades. Con dicha especificación, el módulo (c) procederá a la creación del código fuente del componente *Service Adapter*, como así también a la creación del archivo de configuración de Inyección de dependencias y los test de unidad correspondientes.

En la Figura 5.3 se detalla el deployment de la aplicación, el desarrollador desde su Pc in-

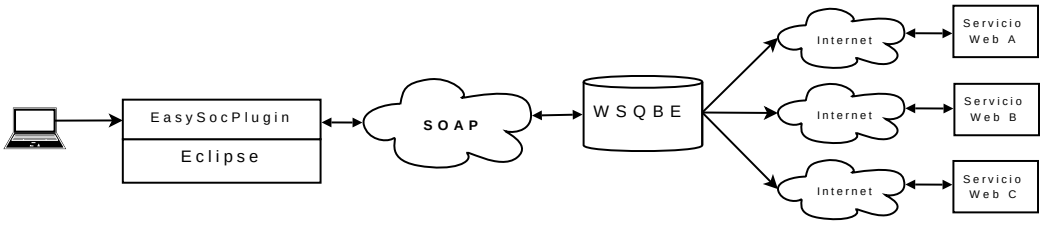


Figura 5.3: Vista de deployment.

teractúa con el entorno de desarrollo Eclipse con EasySOCPlugin incorporado, éste se comunica por medio de un servicio Web a un repositorio remoto WSQBE, el cual contiene las referencias a los distintos servicios Web publicados en Internet.

5.2. Módulo de generación de *queries* y búsqueda de servicios Web

El módulo de generación de *queries* y búsqueda de servicios Web es el encargado de realizar búsquedas de servicios de terceros mediante un proceso de descubrimiento de servicios que implementa el enfoque EasySOC. De esta forma, se automatiza la generación de *queries* y la búsqueda de servicios en un repositorio de servicios Web. Como se mencionó en 4.1, el enfoque implementado genera *queries* a partir de los componentes que forman la aplicación, recolectando información útil para llevar a cabo la búsqueda. Complementariamente, se provee una interfaz gráfica que permite al desarrollador acompañar el proceso de armado del *query*, permitiendo que éste sea un proceso sencillo.

En la Figura 5.4 se muestra un diagrama de actividades del proceso de búsqueda de servicios con las distintas etapas que lo conforman.

La primera etapa del proceso de búsqueda de servicios, es iniciada por medio de la selección de una interfaz Java (previamente definida dentro de la aplicación en desarrollo) que defina la funcionalidad del componente que se desea tercerizar, como se muestra en la Figura 5.5 . Esta actividad corresponde al estado rolutalo 1 en el diagrama mostrado en la Figura 5.4. Luego, la herramienta EasySOCPlugin comienza con la etapa de creación del *query*, a utilizarse en la búsqueda del servicio. Este proceso de armado puede ser guiado por el desarrollador, es decir, a través de interfaces gráficas de ayuda se le permite al mismo configurar distintas preferencias que definirán el contexto del servicio a buscar.

Para realizar la búsqueda de los componentes de software que pertenecen al contexto, se utiliza un motor de búsqueda creado a partir de componentes provistos por la API de Eclipse

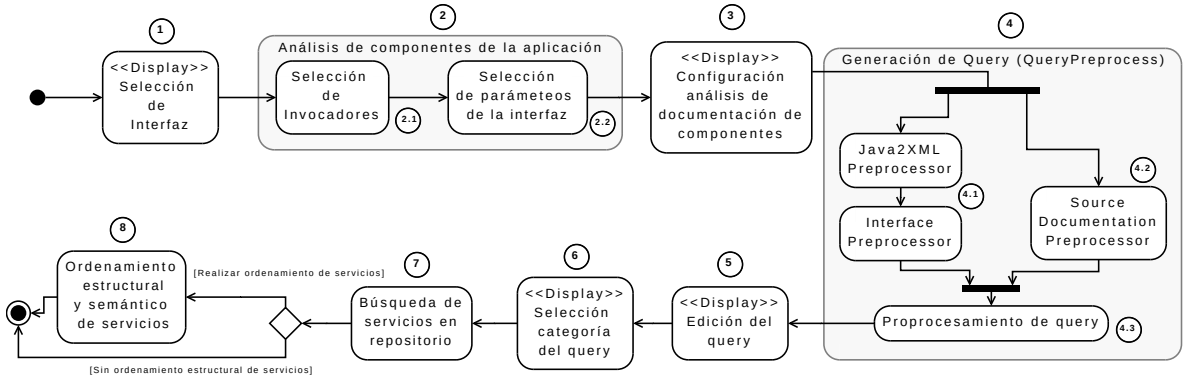


Figura 5.4: Proceso de descubrimiento de servicios

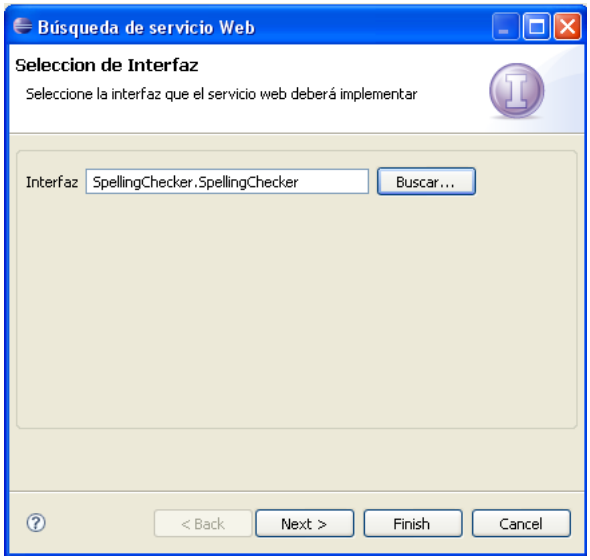


Figura 5.5: Pantalla selección de interfaz del componente a tercerizar.

más componentes customizados por la herramienta EasySOCPlugin, que permiten la inclusión en la búsqueda de los parámetros de configuración definidos por el desarrollador. En el Algoritmo 5 se puede observar cómo se realiza la búsqueda de dependientes, mientras

```
procedure buscarDependientes(interfaz, nivelActual)
1:  if nivelActual <maximoNivelDependencia then
2:    listaDependientes ← Eclipse.buscarDependientes(interfaz);
3:    for cada dependiente d de listaDependientes do
4:      if not(excluirDependienteDelContexto(d)) then
5:        agregarAResultadoDependientes(d);
6:        buscarDependientes(d,nivelActual+1);
7:      end if
8:    end for
9:  end if
10: return resultadoDependientes
```

Algorithm 5: Búsqueda de dependientes.

que en el Algoritmo 6 se observa la búsqueda de argumentos. Las configuraciones propias

```
procedure buscarArgumentos(interfaz, nivelActual)
1:  if nivelActual <maximoNivelSuperClase then
2:    listaArgumentos ← Eclipse.buscarArgumentos(interfaz);
3:    for cada argumento a de listaArgumentos do
4:      if not(excluirArgumentoDelContexto(a)) then
5:        agregarAResultadoArgumentos(a);
6:        buscarArgumentos(a,nivelActual+1);
7:      end if
8:    end for
9:  end if
10: return resultadoRependientes
```

Algorithm 6: Búsqueda de argumentos.

incluidas por la herramienta EasySOCPlugin al motor de búsqueda de Eclipse, son la configuración de niveles de búsqueda y la eliminación de clases que no pertenezcan al contexto como se detalla en la sección 4.2.2.

La recolección de información para la creación del *query* contempla desde la etapa 2 hasta la etapa 4 en la Figura 5.4. La primera de ellas (etapa 2) se encuentra estrechamente relacionada al concepto de Expansión de *query* detallado en 4.2.2. El objetivo es indicar cuáles componentes de la aplicación son considerados al momento de recuperar términos relevantes que conformen el *query*. De esta forma, la herramienta extrae información relevante sólo de los componentes seleccionados. Es decir, el desarrollador limita el contexto de la interfaz dentro de la aplicación, y de esta forma se "configura" la forma que se realiza la Expansión de *Query*. En la subetapa 2.1 se brinda la posibilidad de seleccionar cuáles componentes dependientes de la interfaz serán considerados durante el proceso de generación de *query*, mientras que en la subetapa 2.2 se da la posibilidad de indicar cuáles argumentos de los métodos de la interfaz serán considerados. En la Figura 5.6 se muestran las pantallas que brinda la herramienta

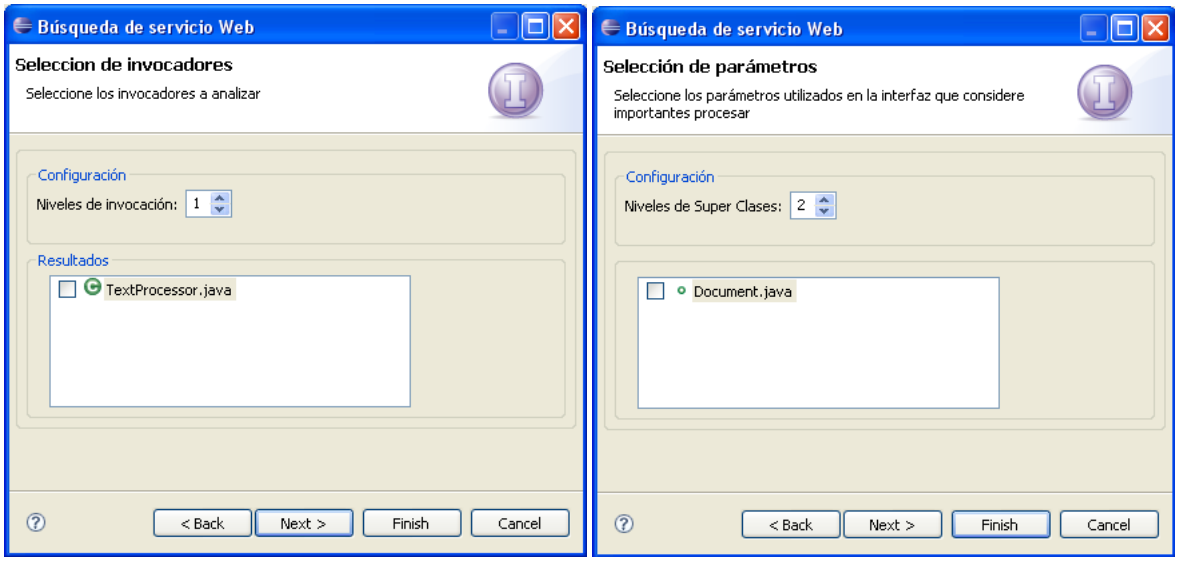


Figura 5.6: Selección de contexto.

para poder llevar a cabo la selección del contexto.

En la etapa 3 de la Figura 5.4 se brinda la posibilidad de seleccionar cuál tipo de documentación se inspecciona para cada tipo de componente, con el objetivo de extraer términos relevantes sólo de las partes de la documentación indicadas. Los distintos tipos de documentación, detallados en 4.2.1 son Documentación de cabecera, Documentación funcional ó Completo (considera tanto la documentación funcional como la de cabecera). Cabe destacar que la herramienta permite analizar distintos tipos de documentación, dependiendo del tipo de componente que se encuentre bajo análisis. Como se muestra en la Figura 5.7 , se permite

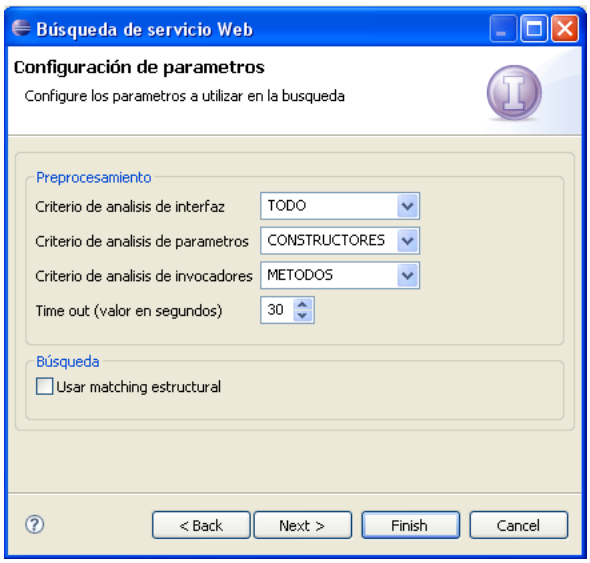


Figura 5.7: Selección de tipo de documentación.

indicar cuál tipo de documentación se extrae de la interfaz, cuál tipo se extrae de cada argu-

mento y cuál de cada dependiente.

Luego que el desarrollador ha indicado las preferencias para el proceso de generación del *query*, se procede a recuperar los términos que lo conforman a través de la etapa 4. En primer lugar, la herramienta aplica dos procedimientos a cada uno de los componentes de la aplicación bajo análisis (uno de ellos es la interfaz, los restantes son los argumentos y dependientes seleccionados en la etapa 2), con el objetivo de extraer los términos relevantes contenidos en sus nombres y de la signaturas de cada método. La diferencia entre los dos procedimientos radica en las técnicas utilizadas para llevar a cabo dichas tareas. El primero de estos procedimientos, denominado InterfacePreprocessor, extrae términos relevantes del código fuente de una interfaz utilizando Java Reflection API. El segundo procedimiento se denomina Java2XMLInterfacePreprocessor. Para llevar a cabo su tarea, en primer lugar, este preprocesador genera una representación XML del código fuente de un componente determinado. Para ello se utiliza la herramienta Java2XML[3]. Luego, se extraen términos relevantes desde la representación XML generada, en los que se incluye el nombre del componente, nombres de los métodos, nombre de los argumentos de cada método, etc. Ambas técnicas de preprocesamiento son alternativas, es decir, no necesariamente debe aplicarse ambos procedimientos.

En segundo término, la etapa 4 también se ocupa de recuperar términos relevantes desde la documentación contenida en los códigos fuentes de los componentes seleccionados anteriormente. Para ello se incluye un preprocesador denominado SourceDocumentationPreprocessor el cual recupera todos los términos contenidos dentro de la documentación, dependiendo del tipo de documentación a recuperar indicado en la etapa 3. Este preprocesador utiliza la herramiente Javadoc [4], la cual sirve para generar documentación de la APIs en formato HTML a partir del código fuente Java. Un pieza de suma importancia en la herramienta Javadoc son los Doclets. Estos proveen un mecanismo para inspeccionar, a nivel código fuente, la estructura de un programa, incluyendo los comentarios en formato Javadoc embebidos en el código. Si bien la herramienta Javadoc posee un Doclet por defecto, en este trabajo se ha creado un nuevo Doclet denominado SourceDocumentationDoclet, utilizando la Doclet API. Dicho Doclet no tiene como objetivo generar documentación de un componente (en HTML u en otro formato) a partir de su código fuente, sino que sólo se limita a inspeccionar dicho código, recuperando información potencialmente relevante para el preprocesador SourceDocumentationPreprocessor. Cabe destacar que el Doclet conoce el tipo de documentación que debe analizar, ingresado por el desarrollador en el paso 3.

Luego de haber aplicado los preprocesadores Java2XMLInterfacePreprocessor (y/ó InterfacePreprocessor) y SourceDocumentationPreprocessor, la herramienta ha recuperado un conjunto de términos que forman parte del *query*. En la Figura 5.8 se muestra la interacción entre los distintos elementos de la herramienta EasySOCPlugin que llevan a cabo la funcionalidad descripta anteriormente. Más precisamente, el componente PreprocessQuery es quién coordina la ejecución de cada uno de los preprocesadores para cada componente bajo análisis (estos componentes son la Interfaz, cero ó más Dependientes y cero ó más Argumentos).

Una vez que se ha obtenido toda la información necesaria a utilizar en la búsqueda, ésta es preprocesada de acuerdo a las técnicas de *text mining* propuestas en 4.2.1. El preprocesamien-

[3]http://java2xml.dev.java.net/
[4]http://java.sun.com/j2se/javadoc/

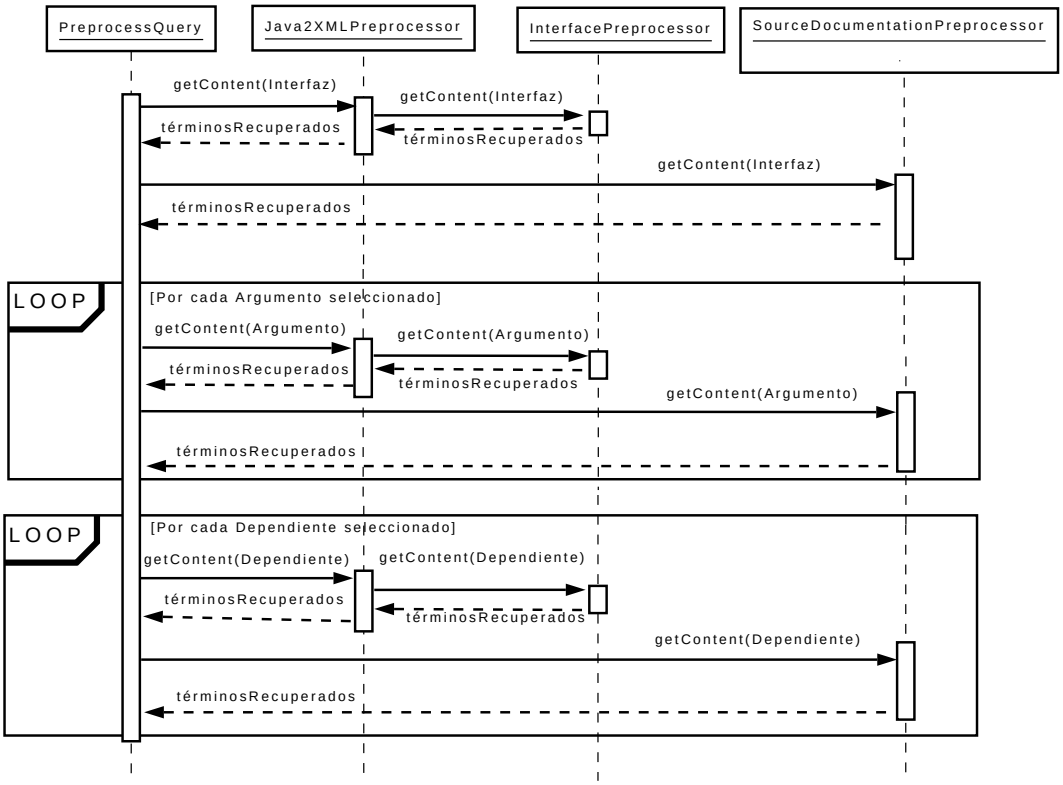


Figura 5.8: Generación de *Query*.

to es un conjunto de filtros, los cuales refinan el *query* aplicando las técnicas de separación de nombres, stemming y eliminación de *stop words* como se observa en la Figura 5.9.

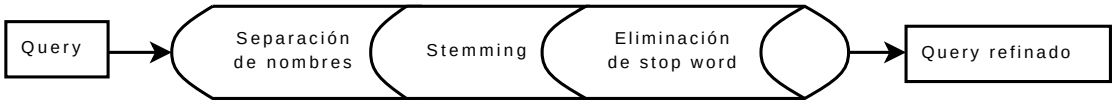


Figura 5.9: Preprocesamiento del query de búsqueda.

En la etapa 5, la herramienta EasySOCPlugin muestra los términos que conforman el *query* de búsqueda, con el objetivo que el desarrollador pueda editarlo. En la Figura 5.10 se mues-

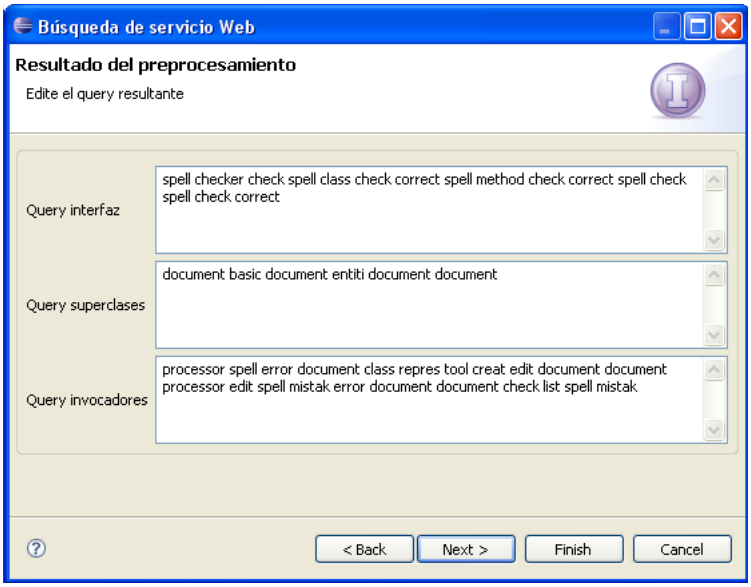


Figura 5.10: Pantalla de edición de *query*.

tra la pantalla que permite al desarrollador visualizar y editar el *query*. De esta forma, el desarrollador puede remover términos existentes, ó bien, agregar otros que él considere relevantes para la búsqueda del servicio en cuestión.

Debido a que los servicios Web se encuentran organizados en categorías dentro de un repositorio WSQBE, la herramienta posibilita al desarrollador seleccionar una categoría existente en el registro, con el objetivo de acotar la búsqueda de servicios considerando sólo aquellos que pertenezcan a la categoría seleccionada. En la etapa 6 de la Figura 5.4, la herramienta obtiene desde el registro WSQBE las categorías afines al *query* creado en las etapas previas. Posteriormente, como se muestra en la Figura 5.11, el desarrollador puede seleccionar una de estas categorías afines, ó bien, puede indicar que la búsqueda de servicios se realice considerando todas las categorías existentes en repositorio.

Ya con el query refinado, la herramienta EasySOCPlugin realiza la búsqueda de servicios similares a dicho *query*, mediante la invocación a un registro WSQBE remoto. Esta actividad corresponde a la etapa 7 de la Figura 5.4. Los servicios candidatos recuperados son ordenados descendientemente según la similitud con el *query* en cuestión, permitiendo al

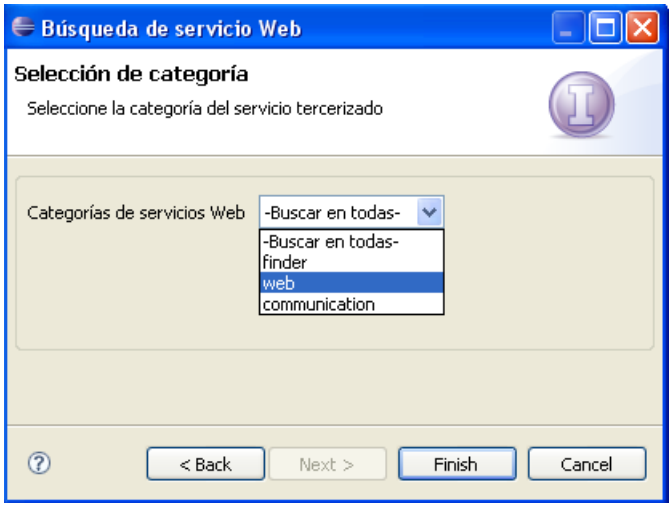


Figura 5.11: Selección de categoría del *query*.

desarrollador visualizar los servicios ordenados bajo este criterio, tal como se muestra en la Figura 5.12 . La herramienta brinda la posibilidad de visualizar la especificación de cada

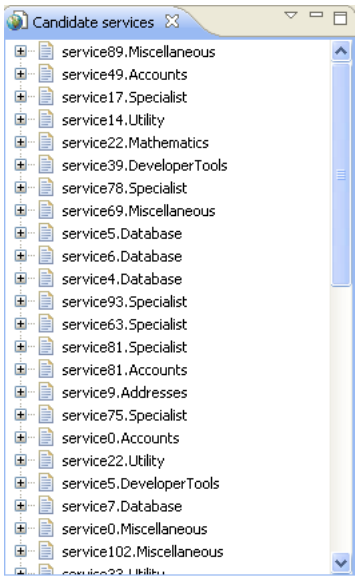


Figura 5.12: Lista de servicios candidatos.

servicio Web descubierto.

Complementariamente, la herramienta permite que la lista de servicios candidatos descubiertos pueda ser ordenada de acuerdo a la similitud estructural y semántica de cada uno de los servicios candidatos con respecto a las características deseadas del servicio a tercerizar, representado mediante la interfaz Java en cuestión. Este ordenamiento permite al desarrollador, intuir cuáles servicios serán más fáciles de adaptar dentro del conjunto de servicios candidatos. Para ello, la herramienta utiliza una implementación del algoritmo de cálculo de similitud estructural y semántica de dos servicios Web descripto en 4.3.1. Este algoritmo

insume dos especificaciones de servicio en lenguaje WSDL y retorna un valor que indica el grado de similitud entre las dos especificaciones. Cuanto mayor es este valor, mayor es la similitud estructural y semántica entre los servicios. El primer paso del ordenamiento consiste en calcular el valor de similitud entre la interfaz del componente a tercerizar y cada uno de los servicios candidatos. Como la interfaz del componente es especificada en lenguaje Java, se necesita la especificación WSDL correspondiente a ella para poder utilizar el algoritmo. Para realizar esta transformación, se utiliza la herramienta Java2WSDL, incluida en el *framework* Axis [5]. Una vez que se calcula el valor de similitud entre el WSDL de la interfaz y la especificación de cada servicio candidato, se ordena la lista de manera descendiente de acuerdo a estos valores. Esta actividad corresponde a la etapa 8 de la Figura 5.4.

5.3. Módulo de adaptación del servicio Web a la interfaz cliente

El módulo de adaptación del servicio Web implementa la primera actividad del proceso de creación del *Service Adapter*, denominada Adaptación de Interfaces. El objetivo es encontrar una manera de adaptar la interfaz del componente interno que se desea tercerizar y la interfaz del servicio candidato seleccionado.

El conjunto de filtros, mostrado en la Figura 5.13 detalla la implementación del módulo.

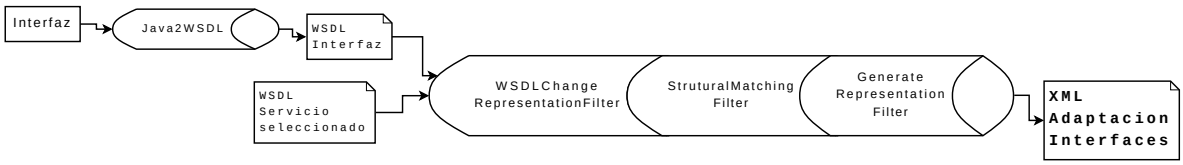


Figura 5.13: Filtros de Adaptación de Interfaces.

El primero de estos filtros, WSDLChangeRepresentationFilter insume dos documentos WSDL. Uno de ellos corresponde a la especificación de servicio candidato seleccionado, el restante corresponde a la especificación del componente interno que se desea tercerizar. Como se mencionó en el capítulo 4, el desarrollador especifica el componente interno, cuya implementación desea tercerizar, mediante una interfaz Java. Por lo tanto, se utiliza la herramienta Java2WSDL incluida en el *framework* Axis para obtener una especificación WSDL equivalente a la interfaz del componente interno. De esta manera, ambas especificaciones de interfaces están descriptas en el mismo lenguaje. Este filtro se encarga de llevar ambas definiciones a una representación de interfaces orientada a objetos común, con el objetivo de manipular los distintos elementos de cada especificación de manera trivial durante los filtros venideros. En la Figura 5.14 se muestra un diagrama de clases que muestra los elementos que conforman la representación. La utilización de una representación interna permite independizar al proceso de adaptación de interfaces del estándar utilizado para especificar la entrada (en este caso WSDL), ya que si se desea utilizar otro tipo de especificación en las interfaces sólo se debe reemplazar el filtro WSDLChangeRepresentationFilter por uno nuevo que genere la representación interna de objetos a partir del estándar de especificación deseado.

[5] http://ws.apache.org/axis/

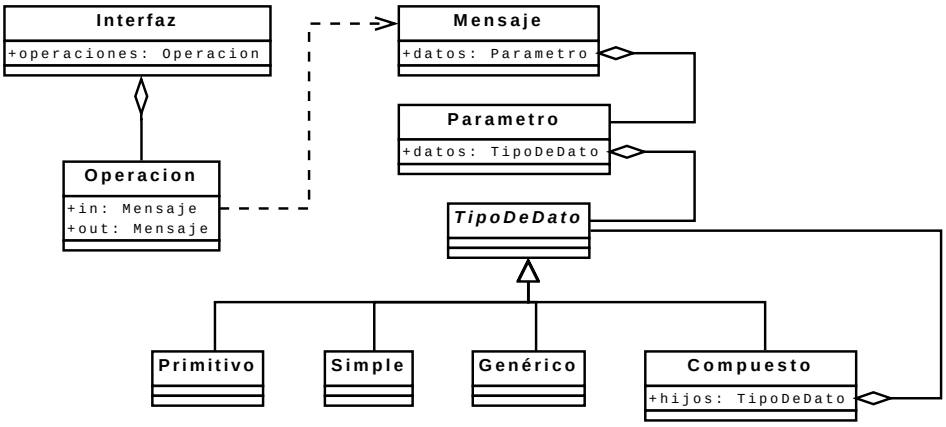


Figura 5.14: Representación orientada a objetos de una interfaz a adaptar.

El segundo filtro de la Figura 5.13, denominado StructuralMatchingFilter, ejecuta una implementación del algoritmo de similitud semántica y estructural de interfaces de servicios presentado en 4.3.1. Como se mencionó en dicha sección, este algoritmo no sólo obtiene el valor de similitud de las interfaces, sino que además registra cómo se obtuvo dicho valor. De este modo, se obtiene una vía (cercana a la ideal) de adaptar ambas interfaces.

Aprovechando la estructura jerárquica que conforman los elementos de cada representación de interfaces, la implementación del algoritmo se basa en la técnica de *Backtracking*. Además, utiliza estructuras matriciales para mantener la información de similitud de dos elementos, ya sean operaciones, mensajes ó tipos de datos. Cada celda de la estructura matricial contiene un valor numérico que indica la similitud entre dos elementos de cada especificación del servicio y una referencia a una matriz que indica cómo se obtuvo ese valor a partir de sus subelementos. Los elementos que pertenecen a la representación de la especificación del componente interno se denominan elementos *Sources* y aquellos que correspondan a la representación de la especificación de servicio externo se denominan elementos *Targets*. En la Figura 5.15 se observa un ejemplo de la estructura matricial mencionada.

La entrada tanto del filtro como del algoritmo de similitud son las dos representaciones de los servicios generadas en el filtro anterior. La salida del algoritmo, y consecuentemente del filtro, es una estructura de datos que vincula los distintos elementos de las representaciones.

El primer paso del algoritmo de similitud, consiste en construir una matriz donde las filas corresponden a las operaciones de la especificación del componente interno y las columnas corresponden a cada una de las operaciones de la especificación del servicio. Cada celda S_i, T_j de la denominada *Matriz de matching de operaciones* contiene un valor de similitud de la operación i-ésima del *source* (S_i) y la operación j-ésima del *target*(T_j). Además, posee referencias a dos matrices denominadas *Matrices de matching de mensajes*. Una relaciona los mensajes de entrada de la operación S_i con los mensajes de entrada de T_j, la restante relaciona los mensajes de salida de dichas operaciones. A partir de análisis de estas matrices se calcula el mencionado valor de similitud de las operaciones S_i y T_j.

Cada *Matriz matching de mensajes*, posee filas que corresponden a los mensajes (de entrada ó salida según corresponda a la matriz) de S_i y columnas que corresponden a los mensajes de T_j. De igual forma, cada celda Ms_i, Mt_j posee un valor que indica la similitud del i-ésimo

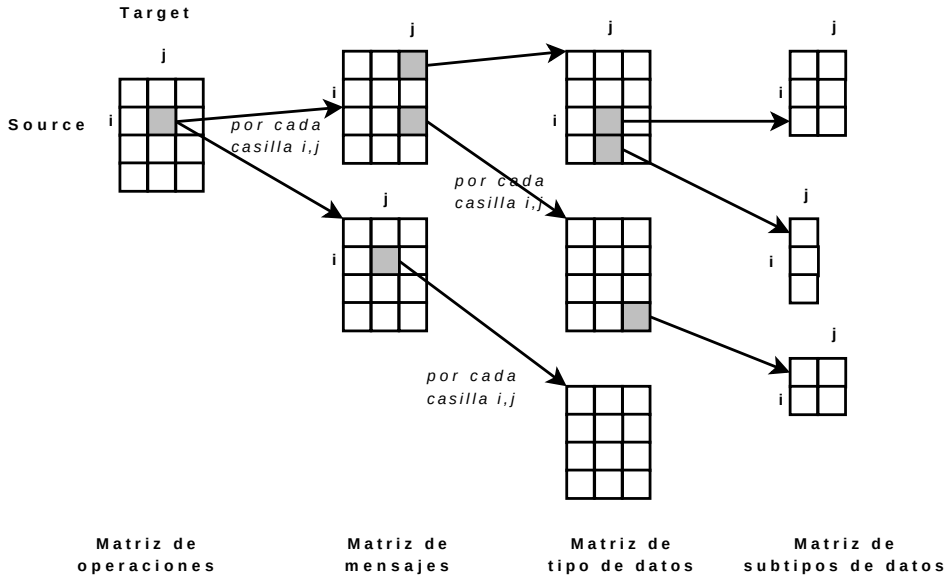


Figura 5.15: Estructura matricial contenedora de los valores de correspondencia estructural.

mensaje *source* (Ms_i) con el j-ésimo mensaje *target* (Mt_j). También, se posee una referencia a una matriz denominada *Matriz de matching de tipo de datos*, donde cada fila corresponde a los tipos de datos relacionado al mensaje Ms_i mencionado y las columnas corresponden a los tipos de datos del mensaje Mt_j.

Cada celda Dts_i, Dtt_j de la matriz de tipo *Matriz de matching de tipo de datos* posee un valor de similitud entre el tipo de dato i-ésimo (Dts_i) del mensaje Ms_i, y el tipo de dato j-ésimo (Dtt_j) del mensaje Mt_j. Si ambos tipos de datos son primitivos, el valor de similitud se obtiene determinando la compatibilidad entre ambos, y a partir de ello, se determina un valor numérico de similitud tal como se explicó en 4.3.1. Por otro lado, si alguno de los datos no es primitivo (es decir, al menos uno es complejo), el problema se trata de manera recursiva. Nuevamente se tiene una matriz de tipos de datos, pero ahora las filas y columnas corresponden a los tipos de datos contenidos en Dts_i y Dtt_j respectivamente. Vale aclarar que si se tiene un tipo de datos complejo y otro primitivo, la matriz posee una única fila/columna (dependiendo si el primitivo es *source* ó *target* respectivamente) que representa al tipo de datos primitivos. De este modo, se compara cada tipo de datos "hijo" del tipo complejo, con aquel tipo de dato primitivo.

El valor de correspondencia entre dos tipos de datos primitivos es aquel mencionado en la sección 4.3.1. A fin de mejorar la performance del algoritmo de matching se han utilizado técnicas de cache almacenando valores previamente calculados de correspondencia entre tipos de datos y el cálculo entre la distancia semántica de dos términos.

Una vez que el algortimo encuentra el valor de similitud de dos tipos de datos primitivos, empieza un "retroceso" hasta hallar el valor de similitud entre las dos representaciones de interfaces. Es decir, se calcula secuencialmente los valores de similitud de los distintos elementos de la representaciones (tipos de datos, mensajes, operaciones) cuyo cálculo ha sido dejado pendiente. El primer paso del retroceso consiste en calcular la similitud de los tipos de datos complejos (que contienen a los datos primitivos) a partir de la información hallada.

De esta forma, se tienen todos los valores de similitud entre los tipos de datos que comprenden los complejos contenidos en una *Matriz de matching de tipo de datos*. El siguiente paso consiste en obtener los pares entre tipos de datos "hijos" del *source* y del *target* que maximizan el valor de similitud (se remarca que si uno de los tipos de datos es primitivo, el tipo de dato "hijo" es él mismo). El conjunto de pares se obtienen mediante el método Kuhn's Hungarian [10], dada la matriz devuelve una lista de pares de elementos (en este caso el par de elementos esta formado por un tipo de datos *source* y otro tipo de datos *target*) que maximiza el valor de similitud.

Una vez que se obtiene el valor de similitud entre todas las combinaciones de tipos de datos de un mensaje *source* y uno *target*, es posible calcular la similitud entre estos dos mensajes obteniendo los pares de tipos de datos de estos mensajes que maximizan la similitud. Para ello, se utiliza el método Kuhn´s Hungarian con la matriz de tipo *Matriz de matching de tipo de datos* correspondiente al par de mensajes bajo análisis. A partir de haber calculado el valor de similitud de cada uno de los pares de mensajes de dos operaciones, se completa cada celda de la *Matrices de matching de mensajes.* Luego, se puede obtener la similitud entre dos operaciones, hallando los pares de mensajes que maximicen la similitud de las operaciones, tanto para los mensajes de entrada como de salida. El conjunto de pares se determina también a partir del método Kuhn's Hungarian, pero considerando las matrices *Matrices de matching de mensajes.* Por último, a partir de los valores de similitud de las operaciones, se busca aquellos pares de operaciones que maximicen el valor de similitud de servicio.

De esta forma, la salida del filtro estará constituida por todos los pares de correspondencias entre elementos de las representaciones de interfaces obtenidas por el algoritmo. Es decir, cada par de operaciones junto con una referencia a la lista de pares de mensajes correspondientes, y a su vez, cada par de mensajes junto con una referencia a la lista de pares de tipos de datos. Eventualmente, cada par de tipos de datos puede tener asociada una lista de pares de tipos de datos, correspondientes a la relación entre los subtipos de datos de, al menos, uno de ellos.

El tercer fitro de la Figura 5.13, denominado GenerateRepresentationFilter, traduce la información de correspondencias generada por el filtro anterior a un documento escrito en un lenguaje formal comprensible por el desarrollador. Esta información indica cómo se debe realizar la adaptación entre las dos interfaces (aquella correspondiente al componente interno y la asociada al servicio Web). El desarrollador puede editar/modificar el documento con el fin de tener participación en la etapa de creación del *Service Adapter*.

Para ello, en el presente trabajo se ha creado un lenguaje estructurado para representar documentos que contengan la información de correspondencias entre los elementos de la representación de la interfaz. Un aspecto importante a remarcar, es que el lenguaje establece las relaciones entre elementos de la especificaciones WSDL de los servicios. Esto se debe a que la entrada del algoritmo de *matching* insume dos documentos WSDL: uno correspondiente a la interfaz del cliente y otro a la interfaz del servicio externo.

Este lenguaje está definido a partir del metalenguaje estructurado XML. Se define una serie de etiquetas, cuya misión es relacionar a dos elementos de cada una de las especificaciones WSDL. La particularidad de estas etiquetas es que cada una de ellas poseen dos atributos denominados *Source* y *Target*. Estos atributos se los puede considerar como referencias a elementos de las especificaciones WSDL. El atributo *Source* referencia a un elemento de la especificación WSDL equivalente a la interfaz del componente a implementar mientras que

el atributo *Target* lo hace a uno de la especificación del servicio externo. Cabe remarcar que el tipo de elemento referenciado por estos atributos depende de la etiqueta en cuestión. La Figura 5.16 muestra un ejemplo de un documento especificación de mapeo de interfaces.

```
<?xml version="1.0" encoding="UTF-8"?>
<definition>
  <schema>
    <source path="D:/desarrollos/mappings/ICalculator.wsdl" ref="edu.isistan.client.ICalculator" />
    <target path="D:/desarrollos/mappings/CalculatorService.wsdl" />
  </schema>
  <mapping>
    <operationMapping source="getRadians" target="degreesToRadians">
      <in>
        <partMapping source="parameters" target="part1">
          <typeMapping source="in0" target="value" />
        </partMapping>
      </in>
      <out>
        <partMapping source="parameters" target="part1">
          <typeMapping source="getRadiansReturn" target="return.fromValue" />
        </partMapping>
      </out>
    </operationMapping>
  </mapping>
</definition>
```

Figura 5.16: Ejemplo de especificación de *matching* de interfaces.

Por la naturaleza jerárquica propia del lenguaje estructurado XML, una etiqueta puede tener otras etiquetas hijas. La relación jerárquica entre estas etiquetas *Mappings*propuestas es idéntica a la relación jerárquica que existe en un documento WSDL. Es decir, la etiqueta *operationMapping*puede contener una etiqueta *in*y otra *out*. A su vez, cada una de estas contiene contiene una ó más etiquetas *partMapping*, las cuales contienen al menos una etiqueta hijo *typeMapping*.

Las etiquetas *operationMapping* se utilizan para relacionar sólo dos operaciones pertenecientes a dos especificaciones de servicios. La primera de las operaciones relacionada pertenece a la especificación WSDL que describe la interfaz cliente. A esta operación se la denomina operación *Source*. La segunda operación, denominada operación *Target*, pertenece a la especificación WSDL de un servicio externo. En el contexto de adaptación de interfaces, relacionar dos operaciones de dos servicios indica que la implementación de la operación *Source* se delega a la operación *Target*. Es decir, toda invocación a la operación *Source* de la interfaz cliente desde algún componente interno de la aplicación, implica una invocación remota (implícitamente mediante el componente *Service Adapter*) a la operación *Target* del servicio externo. La etiqueta *operationMapping* posee dos atributos, *Source* y *Target*, que contienen los nombres de las operaciones anteriormente mencionadas.

Las etiquetas *in*y*out*, relacionan dos mensajes de dos operaciones (*in*los mensajes de entrada y *out*los de salida). El nombre de ambas operaciones se indica en los atributos *source*y *target*de la etiqueta *operationMapping*padre. Cada etiqueta contiene una ó más etiquetas*partMapping*, las cuales representan relaciones entre los parámetros en cuestión. Si bien el filtro StructuralMatchingFilter no establece relaciones entre los *parts* de un mensaje, estas relaciones existen implícitamente cuando se relaciona los tipos de datos de dos mensajes. Por lo tanto, las relaciones entre las partes de dos mensajes se hacen explicitas mediante las etiquetas*partMapping*.

Por último, la etiqueta *typeMapping*tiene como objetivo relacionar dos tipos de datos: uno de ellos pertenecientes a un mensaje de una operación *Source*y el restante perteneciente a un mensaje de una operación *Target*. Ambos tipos de datos relacionados deben pertenecer a mensajes del mismo tipo, ya sean mensajes de entrada de la operación, ó bien, mensajes de salida de la operación.

La tarea del filtro GenerateRepresentationFilter consiste en la creación del documento en el lenguaje estructurado, utilizando la información de mapeos de elementos generada por el filtro predecesor. Como se muestra en el Algoritmo 7, el procedimiento que lleva a cabo el filtro es recorrer los pares de elementos, y según el tipo de elementos que se trate, genera una etiqueta específica. Es decir, recorre cada uno de los pares de operaciones, generando la etiqueta *operationMapping*cuyos atributos *Source*y *Target*son los nombres de las operaciones. Luego, se recorre las dos lista de pares de mensajes de entrada y salidas asociadas a cada par de operación, generando las etiquetas hijas *in*y *out*.Luego, a partir de cada par de tipo de datos asociado a un determinado par de mensajes, se crea un *typeMapping* con los nombres de los tipos de datos del par en los atributos *Source*y *Target*.

En la Figura 5.17 se puede observar el editor que provee la herramienta, en la cual el desarrollador/a puede modificar el documento que contiene la especificación de correspondencias resultante. Este editor provee una pestaña para edición XML, Figura 5.17 (a), y otra pestaña en la cual se realiza una previsualización de las correspondencias, Figura 5.17 (b).

5.4. Módulo de incorporación del servicio adaptado a la aplicación cliente

Como se mencionó en 5.1, la salida del proceso de adaptación de interfaces se utiliza durante el proceso de generación del *Service Adapter*.

Ante la necesidad de crear una serie de artefactos, entre ellos la clase Java correspondiente al *Service Adapter*, la herramienta utiliza un *framework* de generación de código y documentos denominado *CodeGen*. Además del componente *Service Adapter*, la herramienta genera automáticamente el archivo de configuración del Container de Inyección de dependencias y los casos de test de unidad del código generado.

Los detalles del *framework* de generación de código se mencionan en 5.4.1. La creación del código del *Service Adapter* se detalla en la sección 5.4.2. Por otro lado, la creación de archivos de configuración del Container de inyección de dependencias se detalla en sección 5.4.3. Por último, el proceso de creación de artefactos correspondientes a unidades de testeos de distintos componentes de la aplicación cliente se detalla en la sección 5.4.4.

5.4.1. *Framework* CodeGen

El objetivo del *framework* es proveer un motor extensible de generación de artefactos. El concepto básico de este motor de generación se muestra en la Figura 5.18. La generación de los artefactos está compuesto por una una serie de Filtros que deben implementar la interfaz IFilter<ARTIFACT,OUTPUT>. Los filtros pueden ser divididos en dos clases: aquellos que

procedure preprocesar(*listaParesOperaciones*)

```
1:  documento ← crearNuevoDocumento;
2:  for cada par p de listaParesOperaciones do
3:     operacionSource ← p.source;
4:     operacionTarget ← p.target;
5:     etiquetaOperacion ← crearEtiquetaOperationMapping;
6:     etiquetaOperacion ← crearAtributo('source'operacionSource.nombreOperacion);
7:     etiquetaOperacion ← crearAtributo('target',operacionTarget.nombreOperacion);
8:     documento ← agregar(etiquetaOp);
9:     listaParesMensajes ← p.listaParesMensajes;
10:    preprocesarMensajes(listaParesMensajes,etiquetaOperacion);
11: end for
12: return documento
```

procedure preprocesarMensajes(*listaParesMensajes*, *etiquetaPadre*)

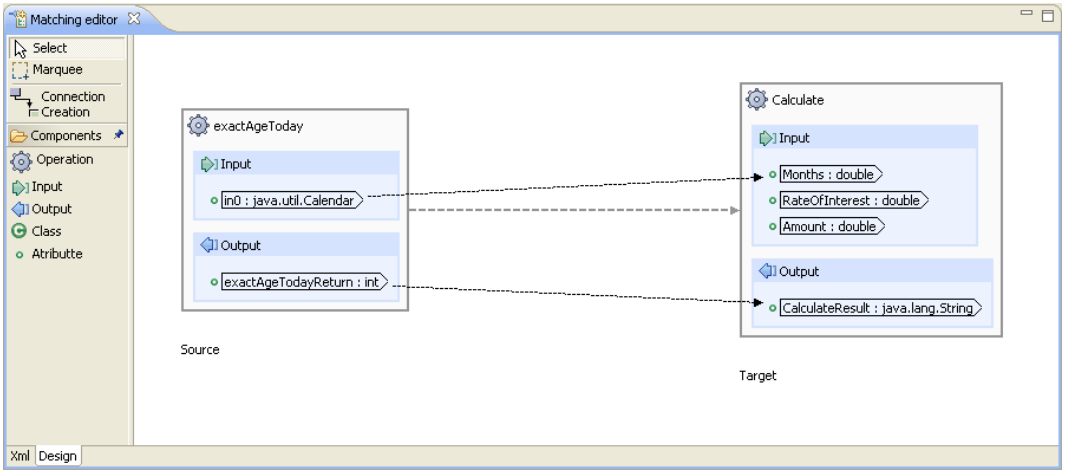
```
1:  for cada par p de listaParesMensajes do
2:     mensajeSource ← p.source;
3:     mensajeTarget ← p.target;
4:     etiquetaMensaje ← crearEtiquetaMessageMapping;
5:     etiquetaMensaje ← crearAtributo('source',mensajeSource.nombreMensaje);
6:     etiquetaMensaje ← crearAtributo('target',mensajeTarget.nombreMensaje);
7:     etiquetaPadre ← agregar(etiquetaMensaje);
8:     listaParesTiposdeDatos ← p.listaParesTiposdeDatos;
9:     preprocesarTiposdeDatos(listaParesTiposdeDatos,etiquetaMensaje);
10: end for
```

procedure preprocesarTiposdeDatos(*listaParesTiposdeDatos*, *etiquetaPadre*)

```
1:  for cada par p de listaParesTiposdeDatos do
2:     tipoDeDatoSource ← p.source;
3:     tipoDeDatoTarget ← p.target;
4:     if tipoDeDatoSource y tipoDeDatoTarget son primitivos then
5:        etiquetaType ← crearEtiquetaTypeMapping;
6:        etiquetaType ← crearAtributo('source',tipoDeDatoSource.nombreMensaje);
7:        etiquetaType ← crearAtributo('target',tipoDeDatoTarget.nombreMensaje);
8:        etiquetaPadre ← agregar(etiquetaType);
9:     end if
10:    if tipoDeDatoSource y tipoDeDatoTarget son complejos then
11:       listaParesSubTiposdeDatos ← p.listaParesSubTiposdeDatos;
12:       preprocesarTiposdeDatos(listaParesSubTiposdeDatos,etiquetapadre);
13:    end if
14: end for
```

Algorithm 7: Algoritmo de correspondencia de mensajes.

(a) Edición XML



(b) Previsualización correspondencias

Figura 5.17: Editor de correspondencías entre servicios Web.

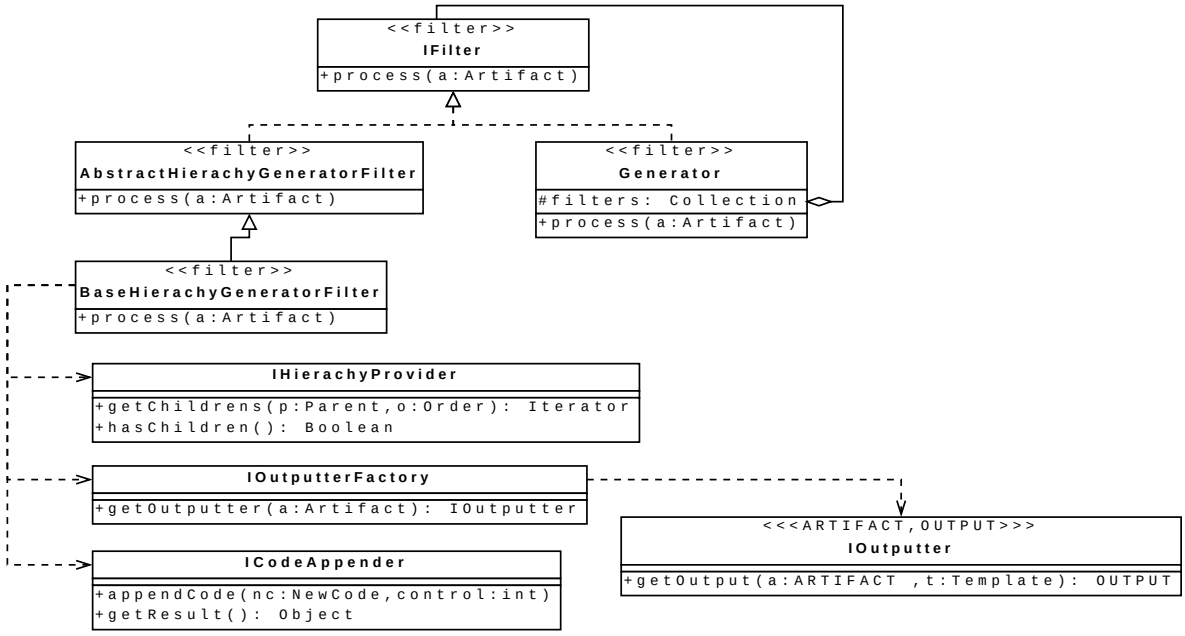


Figura 5.18: *Framework* de generación de código.

se encargan de la generación de código, ó bien, aquellos que se encargan de la persistencia del mismo.

El componente CodeGenerator es el encargado de administrar y coordinar la ejecución secuencial de los filtros. La entrada del motor (y por lo tanto del primer filtro) es un artefacto que incluye la información necesaria durante la generación de un nuevo artefacto generar. Este artefacto está compuesto por distintos tipos de elementos organizados de manera jerárquica. A su vez, cada tipo de elemento tiene asociado un representador. Cada uno de estos representadores genera una representación para su tipo de elemento asociado. Los representadores implementan la interfaz IOutputter<ARTIFACT,OUTPUT>, la cual es parametrizada con el artefacto que representa y por el tipo de salida que genera, respectivamente. Como los elementos que conforman un artefacto están organizados jerárquicamente, los representadores también se encuentran organizados de la misma manera.

El *framework* provee un filtro, denominado AbstractHierachyGeneratorFilter, encargado de la generación de código asociado a artefactos que poseen una organización jerárquica. El objetivo es coordinar el proceso de generación de un artefacto, con el fin de generar una representación para cada uno de los elementos hijos en la jerarquía que lo conforman. Además se provee una extensión a dicho filtro denominada BaseHierachyGeneratorFilter la cual interactúa con otros tres componentes del *framework*. El primero de ellos es un *factory* de representadores (IOutputter), siendo una implementación de la interfaz IOutputterFactory. El segundo de ellos, es una implementación de la interfaz IHierachyProvider encargada de determinar cuáles son los elementos hijos dentro de la jerarquía para cada tipo de elemento. Por último, una implementación del ICodeAppender es quién se encarga de organizar la salida que genera los representadores asociados a cada elemento.

El procedimiento que lleva a cabo el filtro BaseHierachyGeneratorFilter es recorrer los el-

ementos que forman un artefacto, siguiendo su organización jerárquica. El orden entre los tipos de elementos los determina el componente IHierachyProvider. Para cada uno de los elementos, el filtro obtiene el representador (IOutputter) asociado para generar la representación del elemento por medio del componente IOutputterFactory. Por último, el componente ICodeAppender recibe la representación del elemento generada, con el fin componer las distintas representaciones y armar así el nuevo artefacto de salida. Este procedimiento de generación se detalla en el Algoritmo 8.

procedure process(ARTIFACT *artifact*)

```
1: outputter ← outputterFactory.getOutputter(artifact);
2: salida ← outputter.getOutput(artifact);
3: codeAppender.appendCode(salida);
4: artefactosHijosIterator ← hierachyProvider.getChildren(artifact);
5: for cada artefactoHijo de artefactosHijosIterator do
6:    process(artefactoHijo);
7: end for
8: return codeAppender.getResultCode();
```

Algorithm 8: Procedimiento general del filtro BaseHierachyGeneratorFilter.

Por otro lado, se provee filtros encargados de la persistencia del artefacto generado. Estos filtros toman el código generado del artefacto, el cual se encuentra almacenado en un componente ICodeAppender, con el fin de almacenarlo en el sistema de archivos.

5.4.2. Generación de Código del *Service Adapter*

El proceso de generación del *Service Adapter* se encarga de generar el código del componente *Service Adapter*. Este componente es una clase Java que implementa la interfaz del componente interno de la aplicación que se desea tercerizar. De esta forma, el componente *Service Adapter* puede ser inyectado dentro de la aplicación cliente sin tener que realizar ningún cambio en ella.

El proceso recibe como entrada la interfaz del componente interno a tercerizar, la especificación de dicha interfaz en lenguaje WSDL, la especificación del servicio Web a integrar en la aplicación en lenguaje WSDL y el documento formal que define cómo se realizará la adaptación de interfaces generado por el proceso de adaptación explicado en 4.3.1. En la Figura 5.19 se observa la acción disparadora del proceso en la herramienta EasySOCPlugin al seleccionar el documento de adaptación asociado a una interfaz y un servicio externo.

Conceptualmente, el proceso de generación del componente *Service Adapter* materializa el código fuente del *Service Adapter* a partir del documento de adaptación de interfaces, a través de una serie de pasos que se detallan a continuación:

Por cada método definido en la interfaz del componente a tercerizar, el proceso de generación realiza los siguientes pasos:

1. Buscar el elemento *operationMapping* dentro del documento de adaptación que corresponda a dicho método. Para ello, se busca la etiqueta tipo cuyo atributo *Source* coincida con el nombre del método en cuestión.

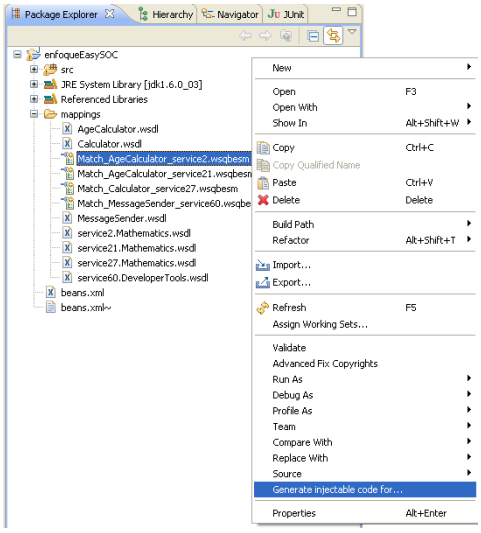


Figura 5.19: Acción disparadora del proceso de generación de código.

2. Generar código correspondiente al encabezado del método. Esto incluye: identificador que indique el alcance del método (privado, público, estático, etc), tipo de dato de retorno, nombre del método, parámetros de entrada del método (tipo de datos y su nombre).

3. Analizar los datos de entrada de la operación relacionada, la cual se obtiene a través del atributo *Target* de la etiqueta *operationMapping* obtenida en el primer paso. Se genera código de declaración de variables por cada dato de entrada de dicha operación. Cada uno de estos datos de entrada puede estar relacionado a un dato de entrada de la operación *Source*. Esta información se obtiene mediante las etiquetas *typeMapping* incluidas dentro del elemento *in*. En tal caso se genera código para transformar el valor del tipo de datos *source*, a un valor equivalente correspondiente al tipo de datos de entrada de la operación *Target*. El código generado depende de los dos tipo de datos que se estén manipulando, si es necesario código de conversión implícito o no por ejemplo.

4. Generar código correspondiente a la invocación de un método del servicio Web externo. Esto implica invocar a la operación *Target* de dicho servicio, mediante un *Stub* del servicio Web externo.

5. Generar código para transformar los datos de salida de la operación *Target*. Esto involucra generar código de declaración de la variable del tipo de dato de retorno del método *Source*. La información necesaria para realizar el código de transformación entre los datos de salida del método *Target* al método *Source* se obtiene a partir de los elementos *typeMapping* contenidos dentro del elemento *out*. Por último, se crea la sentencia de retorno de la variable creada.

A nivel de implementación, el proceso de generación del *Service Adapter* es una extensión del *framework* CodeGen. Como muestra la Figura 5.20 , el proceso consta de seis filtros. Algunos de ellos son extensiones de filtros provistos el *framework*, el resto son filtros provistos por el *framework*, siendo sus propiedades establecidas de acuerdo a las necesidades el proceso.

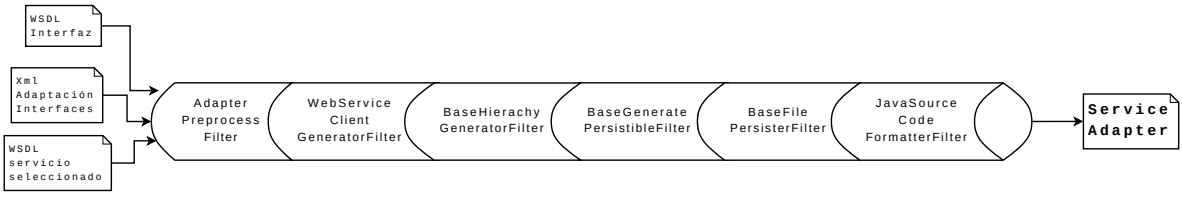


Figura 5.20: Filtros del proceso de generación de *Service Adapter.*

El primero de los filtros, denominado AdapterPreprocessFilter, tiene como objetivo principal interpretar el documento de adaptación y las especificaciones de las interfaces del componente interno y del servicio externo seleccionado. De esta forma, se crea una representación orientada a objetos que representa los elementos de ambas estructuras, y se mantienen estructuras que relacionan los distintos elementos de la representaciones, de acuerdo al documento de adaptación de interfaces.

El segundo filtro denominado WebServiceClientGeneratorFilter se encarga de crear el *Stub* del servicio Web seleccionado para implementar el componente interno. Para ello, se utiliza la herramienta WSDL2Java provista en el *framework* Axis. La clase *Stub* será una variable de instancia del componente *Service Adapter*, y sus métodos serán invocados desde éste para satisfacer las necesidades de cada uno de los métodos definidos en la interfaz cliente. En la Figura 5.25 se muestra la pantalla asociada a la generación del *Stub*, en la cual se debe

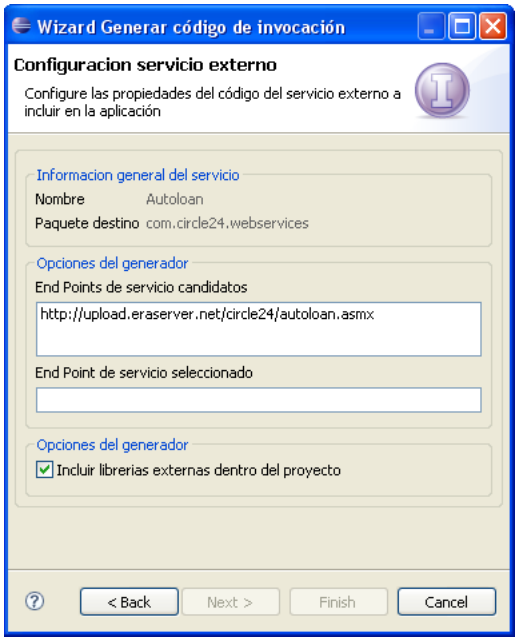


Figura 5.21: Pantalla de configuración de generación de *Service adapter*.

seleccionar la dirección del servicio Web.

El tercer filtro, denominado BaseHierachyGeneratorFilter es el encargado de generar el código fuente en lenguaje Java del componente *Service Adapter*. Como este filtro es nativo del *framework*, sólo demandó generar implementaciones para las tres propiedades del filtro men-

cionadas en la sección anterior.

El cuarto filtro BaseGeneratePersistibleFilter y el quinto filtro BaseFilePersisterFilter son los encargados de la persistencia del código fuente generado en el sistema de archivos, ambos filtros son provistos por el *framework*.

Por último, el filtro JavaSourceCodeFormatterFilter es el encargado de formatear el código fuente generado. Para ello, este filtro utiliza la librería Jalopy[6]. La entrada del filtro es la ubicación del archivo que contiene el código fuente. El código fuente formateado es almacenado en el mismo archivo de entrada.

5.4.3. Generación de archivos de configuración

Como se mencionó en 4.3.1, el enfoque EasySOC utiliza Inyección de Dependencias para mantener una correcta separación entre la aplicación y un servicio Web especifico. En particular, utilizando DI se prentende que el componente *Service Adapter* sea inyectado dentro de la aplicación sin tener que realizar ningún cambio en la misma.

De este modo, la herramienta EasySOCPlugin posibilita que las aplicaciones implementen esta característica utilizando el *framework* de Inverción de Control Spring. Este provee un componente llamado Container encargado de la instanciación y ensamblado de los distintos elementos que forman la aplicación. La configuración del Container (y por lo tanto de la aplicación) se logra mediante un archivo de configuración XML. De este modo, la incorporación del *Service Adapter* (transitivamente del servicio externo) dentro de un componente de la aplicación que insume al componente interno tercerizado (implementado por el componente *Service Adapter*) se lleva a cabo en tiempo de ejecución.

Uno de los objetivos de la herramienta es la creación (ó modificación en caso de existir) del archivo de configuración del Container que contenga la declaración de los distintos componentes de la aplicación y los vínculos existentes entre ellos.

Luego de la creación del componente *Service Adapter*, la herramienta crea este archivo de configuración en caso que no exista. Caso contrario, utiliza el archivo ya existente. En segundo lugar, declara el componente *Service Adapter* recientemente creado. Debido a que es el nexo entre la aplicación y un servicio Web específico, el componente *Service Adapter* tiene un vinculo a clases especificas del servicio en cuestión (por ejemplo, al *Stub* del servicio). Por un lado, la herramienta logra, mediante el Container, que el vínculo entre estos componente se resuelva en tiempo de ejecución. Para ello declara en el archivo las clases específicas del servicio, por ejemplo su *Stub*. Luego, se declara un vínculo entre estos componentes declarados, es decir, entre el *Service Adapter* y el *Stub* del servicio. En la Figura 5.22 se observa la definición de un localizador del servicio Web (a), el *Stub* (b), el componente *Service Adapter* (c) y un dependiente que utiliza la funcionalidad tercerizada (d).

Por otro lado, existen componentes en la aplicación que dependen del componente tercerizado, cuya implementación es el *Service Adapter*. Por lo tanto, el vínculo entre la aplicación y el *Service Adapter* también se logra en tiempo de ejecución. La herramienta declara en el archivo de configuración los componentes dependientes, y además declara un vínculo entre cada uno de ellos con el *Service Adapter*. En la Figura 5.26 se muestra la pantalla de selección

[6] http://jalopy.sourceforge.net/

```
   <beans>
(a) <bean id="exampleLocator" class="com.example.ServiceLocator"/>

(b) <bean id="exampleStub" class="com.example.Stub">
      <constructor-arg index="0" value="http://example.com:80/serviceEndPoint" />
      <constructor-arg index="1" ref="exampleLocator" />
    </bean>

(c) <bean id="service_Adapter" class="example.Service_Adapter">
      <property name="proxy" ref="exampleStub" />
    </bean>

(d) <bean id="dependant" class="example.Dependant">
      <property name="dependant" ref="service_Adapter" />
    </bean>
   </beans>
```

Figura 5.22: Ejemplo de archivo de configuración del Container de inyección de dependencias.

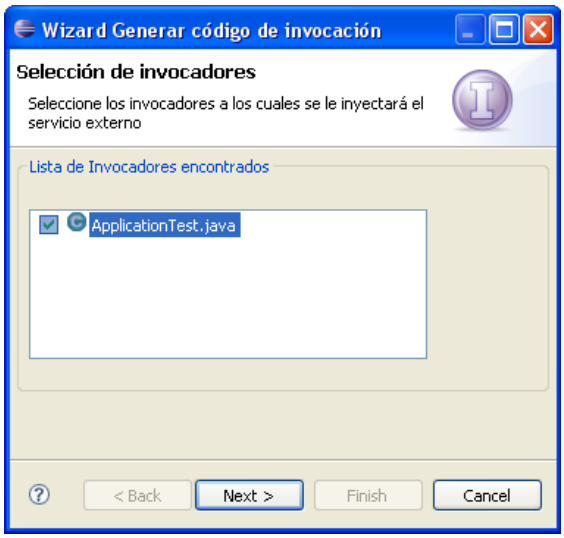


Figura 5.23: Pantalla de selección de dependientes a inyectar.

de componentes dependientes a inyectar dentro del archivo de configuración.

El proceso de generación del *Service Adapter* se implementa utilizando el *framework* CodeGen mediante una serie de filtros, tal como se muestra en la Figura 5.24 .

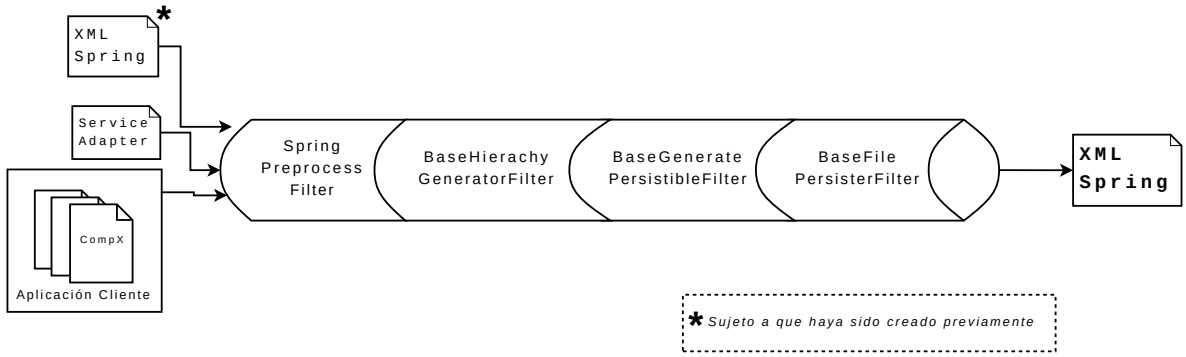


Figura 5.24: Filtros para la generación de archivo de configuración.

El primer filtro, denominado SpringPreprocessFilter, se encarga de generar una representación orientada a objetos del archivo de configuración en caso que esta exista. Caso contrario, crea una nueva representación.

El segundo filtro, denominado BaseHierachyGeneratorFilter, es el encargado de incluir en la representación interna de la configuración la nueva información pertinente al nuevo servicio a incorporar a la aplicación. Es decir, agregar declaraciones del nuevo *Service Adapter*, *Stub* de servicio Web, componentes internos que utilicen/dependan del componente a tercerizar y especificar los vínculos entre ellos. Al igual que el proceso de generación del *Service Adapter*, se modela los distintos tipos de elementos que conforman dicha representación y una serie de representadores asociados a cada uno de estos tipos. Del mismo modo que en la generación del *Service Adapter*, este filtro es nativo del *framework*, lo que demanda generar implementaciones para las tres propiedades mencionadas con anterioridad.

Los dos últimos filtros, denominados BaseGeneratePersistibleFilter y XmlFilerPersisterFilter, son los encargados de llevar la representación interna al formato de texto (quedando conformado el XML) y luego almacenarlo en el sistema de archivos.

En la Figura 5.25 se muestra la pantalla asociada a la generación del archivo de configuración, donde se debe detallar el nombre del *Service Adapter*, la ubicación del archivo de configuración de dependencias y en la cual se valida si cada dependiente a inyectar posee una correcta estructura de inyección (debe existir un mótodo Java conocido como “método setter” para establecer el valor del *Service Adapter* dentro del dependiente).

5.4.4. Generación de casos de test de unidad

Una extensión interesante que propone la herramienta al enfoque EasySOC, es la generación automática de casos de test de unidad, a fin de poder testear los servicios tercerizados, como son los adaptadores generados. Esto es de suma utilidad no sólo al momento de desarrollar la aplicación, sino como una herramienta de validación y simplificación ante la necesidad

Figura 5.25: Pantalla de configuración de inyección de código.

de cambiar de proveedor, ya que la nueva funcionalidad provista puede ser verificada por medio de estos casos de test de una manera muy sencilla y efectiva.

Por un lado, la herramienta genera de manera automática un test de unidad correspondiente al componente *Service Adapter*, con el objetivo de verificar que la tercerización de operaciones no produzca fallos. De este modo, se valida la transformación de los datos de entradas o de salida de cada una de las operaciones del componente como así también la invocación al servicio externo.

Por otro lado, la herramienta brinda la posibilidad de crear un test de unidad por cada componente de la aplicación que depende de un componente tercerizado. Cabe remarcar que cada uno de estos componentes a testear dependen de la interfaz del componente tercerizado. La implementación de este componente tercerizado, delegada a un *Service Adapter* previamente creado, se resuelve en tiempo de ejecución. Con este tipo de test se intenta verificar la correctitud de los distintos componentes de la aplicación luego de haber delegado la implementación de ciertas partes a terceros. Cada uno de estos test de unidad de un componente de la aplicación tiene el objetivo de validar el funcionamiento de cada operación que brinda el componente. A su vez, como algunas de estas operaciones utilizan funcionalidad provista por el componente tercerizado, indirectamente se realiza un testeo del componente *Service Adapter* y del servicio Web de terceros. En la Figura 5.26 se muestra la pantalla asociada a la generación de casos de test, en la cual se eligen los componentes a testear.

La implementación del proceso de generación de casos de test es similar al proceso de generación del componente *Service Adapter*, también habiendo utilizado el *framework* CodeGen presentado en 5.4.2. Como muestra la Figura 5.27 , la implementación está conformada de cuatro filtros, todos ellos provistos por el mencionado *framework*.

El primero de los filtros se denomina BaseHierachyGeneratorFilter. Para la implementación del proceso, sólo se creó dos nuevos componentes requeridos por el filtro. El primero de

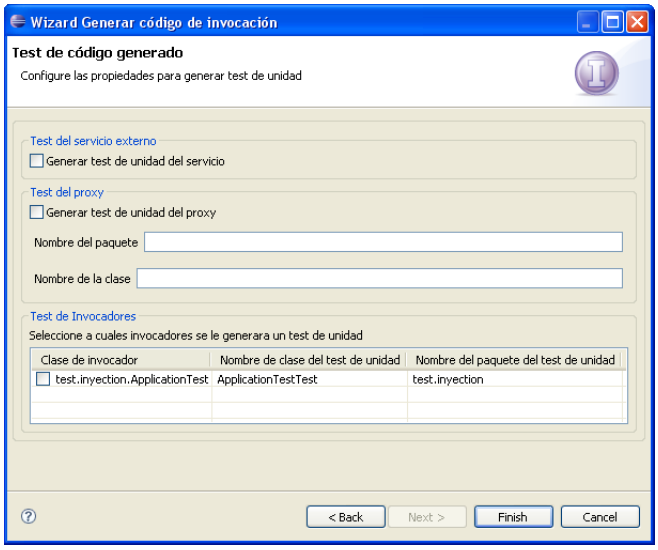


Figura 5.26: Pantalla de selección de casos de test a generar.

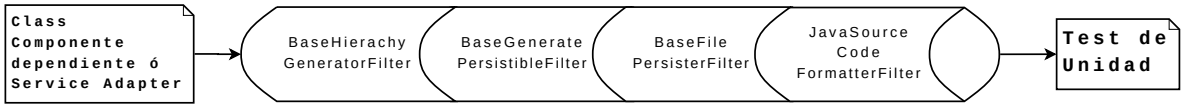


Figura 5.27: Filtros del proceso de generación de test de unidad

ellos, denominado TestCaseHierachyProvider es una implementación de la interfaz IHierachyProvider, mientras que el componente TestCaseOutputterFactory es una implementación de IOutputterFactory. La entrada del filtro es una clase Java que corresponde al componente que se desea crearle un test de unidad. Por ello, el filtro utiliza Java Reflection API para inspeccionar las distintas operaciones del componente y los correspondientes tipos de datos, con el fin de crear código de testeo para cada uno de ellos.

Los 3 filtros restantes, denominados BaseGeneratePersistibleFilte, BaseFilePersisterFilter y JavaSourceCodeFormatterFilter, también son utilizados en el proceso de creación del *Service Adapter* explicado en la sección 5.4.2.

5.5. Conclusión

EasySOCPlugin brinda a los desarrolladores un conjunto de herramientas integradas al entorno de desarrollo Eclipse con el fin que el desarrollo de aplicaciones orientadas a servicios sea una tarea trivial y sencilla. El hecho que Eclipse sea uno de los entornos de desarrollos más populares y utilizados en la actualidad, permite que la mayoría de los desarrolladores ya estén familiarizados con el mismo, por lo que la incursión de nueva funcionalidad al desarrollo de aplicaciones orientadas a servicios utilizando EasySOCPlugin no implica esfuerzos adicionales. Esto se debe a que las interfaces gráficas que provee la EasySOCPlugin facilitan la búsqueda de servicios y la adaptación de los mismos dentro de la aplicación bajo desarrollo. Una de las características a destacar de la herramienta EasySOCPlugin es que

produce software con altos niveles de calidad. En el capítulo 6 se muestra un ejemplo de la aplicación desarrollada con la aplicación, la cual tiene mayores niveles de mantenibilidad que una equivalente desarrollada.

Por otro lado, la arquitectura de la herramienta se encuentra claramente dividida en distintos módulos independientes. Esto permite que la extensión y/o modificación de cada uno de estos módulos sea directa, ya que el uso de filtros así lo facilita.

CAPÍTULO 6

Resultados experimentales

El objetivo de este capítulo es presentar las evaluaciones experimentales de la implementación del enfoque EasySOC presentado en el capítulo 5.

Este capítulo está organizado como sigue: en la sección 6.1 se evalúa el impacto de las técnicas de Expansión de *Query* en la precisión de un sistema para el descubrimiento de servicios, comparando los resultados obtenidos usando consultas expandidas contra consultas ordinarias; en la sección 6.2 se evalúa la incorporación del servicio externo dentro de la aplicación cliente, comparando la mantenibilidad de una aplicación desarrollada bajo el enfoque EasySOC contra la de la misma aplicación, pero desarrollada bajo otro enfoque.

6.1. Evaluación del mecanismo de descubrimiento

El enfoque EasySOC facilita el proceso de descubrimiento de servicios de terceros ofreciendo un proceso automático de generación de *queries*. Medir la *performance* del proceso de descubrimiento del enfoque implica que se realice un análisis de la eficiencia del proceso de generación automática de *queries* presentado en 4.2.1. Es decir, se busca determinar cómo varía la *performance* del proceso de descubrimiento de acuerdo a distintas contemplaciones durante el armado del *query*.

Existen distintos métodos para evaluar la *performance* del proceso de descubrimiento. En particular, la medición de la *performance* de este trabajo se calcula en términos de la porción de servicios relevantes incluidos en la lista de servicios candidatos devuelta por el registro de servicios y la posición relativa respecto a aquellos servicios no relevantes devueltos. Un servicio relevante es aquel cuya funcionalidad satisface la funcionalidad especificada en el código de la interfaz definida por el/la desarrollador/a. En particular, en este trabajo se utiliza las mediciones *R-Precision*, *Recall* y *Precision-at-n*.

Para calcular las mediciones, se crearon 30 experimentos. Cada uno de ellos representa (parte de) una aplicación cliente, escrita con el lenguaje de programación Java, la cual contiene un

componente cuya implementación será delegada a un servicio de terceros. Cada experimento está conformado por el código fuente de la interfaz del componente a tercerizar, el código fuente de los componentes no primitivos que actúan como argumentos de los métodos declarados en dicha interfaz y, al menos, un componente interno de la aplicación, el cual interactúa con el componente a tercerizar (mediante la mencionada interfaz). El objetivo es evaluar para cada experimento, la precisión del mecanismo de descubrimiento del enfoque EasySOC. Es decir, se evalúa la lista de servicios retornada, determinando cuales de estos son relevantes al experimento en cuestión, un servicio relevante es denominado *hit*.

El éxito del proceso de descubrimiento (la capacidad del proceso para devolver *hits* en las primeras posiciones de la lista de servicios candidatos) está estrechamente ligado a las características del *query* de búsqueda. Como el enfoque EasySOC genera *queries* a partir de distintos componentes de la aplicación cliente mediante el concepto de *Expansión de Query*, en el presente trabajo se mide la efectividad del proceso de descubrimiento según el tipo de componentes fuentes utilizados para extraer términos relevantes que conformen el *query*. A continuación, se presentan cinco tipos de *queries*:

Interfaz: El *query* se forma a partir de los términos correspondientes al nombre de la interfaz que define el servicio a tercerizar, como así también de los nombres de las operaciones que la componen.

Documentación: Además de los términos del caso Interfaz, se agregan los términos relevantes incluidos en la documentación provista por el/la desarrollador/a en dicha interfaz, es decir el Javadoc.

Argumentos: Además de los términos del caso Documentación, se agregan términos obtenidos desde el nombre y operaciones, como así también de aquellos incluidos en la documentación, de las clases correspondientes a los argumentos de cada operación de la interfaz.

Dependientes: Además de los términos del caso Documentación, se agregan términos obtenidos desde el nombre, las operaciones y la documentación de las clases correspondientes a los componentes internos de la aplicación, los cuales interactúan con el componente externo.

Completo: El *query* se forma a partir de todos los términos relevantes recuperados desde el código fuente de la interfaz, desde sus argumentos y desde los componentes dependientes. Además, se agregan los términos relevantes extraídos de la documentación de cada componente mencionado.

La Tabla 6.1 detalla la cantidad de términos recuperados en promedio para cada uno de los distintos *queries*, aplicados a los 30 experimentos definidos. Como se puede observar, la expansión de consultas genera *queries* con mayor cantidad de palabras, ampliando el dominio de búsqueda al considerar nuevos términos y reforzar aquellos más significativos al aumentar la cantidad de ocurrencias de los mismos.

El escenario de testeos del proceso de descubrimiento de la implementación de EasySOC incluye una colección de servicios Web categorizados, presentados en [13]. Dicha colección, también llamada *dataset*, se encuentra formada por 393 servicios divididos en 11 categorías.

Query	Palabras extraídas (promedio)
Interfaz	4.57
Documentación	9.77
Argumentos	14.43
Dependientes	16.07
Completo	22.87

Cuadro 6.1: Términos recuperados en promedio por cada mecanismo de Expansión de query.

Para cada experimento, se tienen identificados cuales servicios del repositorio son *hits*. Es decir, cuales servicios son aptos para implementar la funcionalidad faltante del experimento. Cada experimento tiene al menos un servicio relevante.

Para calcular las métricas del mecanismo de descubrimiento, este trabajo plantea realizar los 5 *queries* definidos anteriormente para cada experimento planteado. A partir de cada uno de estos *queries*, el proceso de descubrimiento de servicios resulta en una lista de servicios candidatos a los cuales se aplicarán métricas, con el fin de determinar cual tipo de query optimiza la performance del proceso de descubrimiento para el escenario planteado.

A continuación, se presentan las métricas utilizadas junto los valores asociados.

R-Precision

Una de las métricas utilizadas para calcular la *performance* del proceso de descubrimiento de servicios es *R-precision*. Básicamente, dado un query con R documentos relevantes, la medida calcula la precisión en la R-ésima posición en la lista de servicios. Por ejemplo, dados 10 documentos relevantes dentro del *dataset* para un cierto experimento, y todos ellos son devueltos en los primeras 10 posiciones, la precisión es del 100 %. En cambio, si sólo 5 de ellos son devueltos en las primeras 10 posiciones, se tiene una precisión del 50 %.

Formalmente, se define la medida *R-precision* como $\frac{RetRel_R}{R}$, donde R es la cantidad de documentos relevantes y $RetRel_R$ es la precisión en las R primeras posiciones de la lista de candidatos.

Para calcular el valor de *R-Precision* asociado a cada tipo de *query*, en primer lugar se creó un *query* de cada tipo asociado a cada uno de los 30 experimentos. Como se tiene 5 tipos de *query*, se crearon un total de 150 *queries*. Luego, se calculó el valor de la métrica para cada uno de los *queries*. Por lo tanto, por cada experimento, se obtuvieron 5 valores de la métrica, uno por cada tipo de *query*. En la Figura 6.1 se muestra el valor de la métrica correspondiente a cada uno de los experimentos. Por último, se calculó el valor de la métrica *R-Precision* por cada tipo de *query* específico, promediando los valores de la métrica (correspondientes al tipo de *query* en cuestión) de los 30 experimentos. La Tabla 6.2 detalla los valores promedio de los tipos de *query* utilizados.

Los valores promedios de *R-Precision* obtenidos demuestran que el tipo de query Interfaz maximiza el valor de la métrica. En este caso, el enfoque EasySOC incluye, en promedio, 76.1 % de los servicios relevantes al inicio de la lista. Es decir, se incluye el 76.1 % de los servicios relevantes antes de aquellos no relevantes.

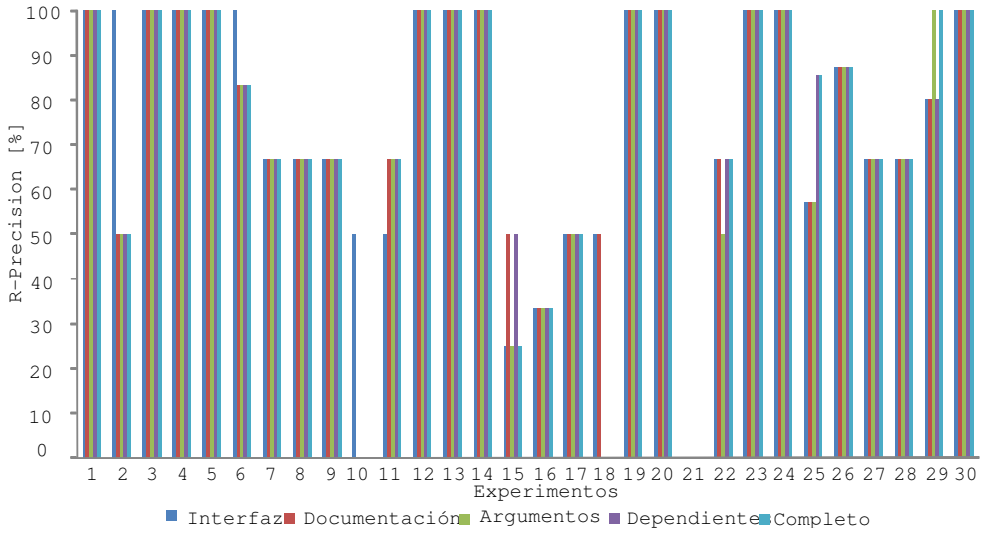


Figura 6.1: Valor de *R-Precision* de cada experimento

Query	*R-Precision*
Interfaz	76.1 %
Documentación	73.6 %
Argumentos	71.21 %
Dependientes	72,88 %
Completo	72,72 %

Cuadro 6.2: Valor *R-Precision* de cada tipo de *query*.

Recall

La métrica *Recall* mide la *performance* del proceso de descubrimiento a partir de los servicios relevantes que recupera. La medida *Recall* es 100 % cuando cada documento relevante es devuelto en la lista de candidatos. Formalmente, se define *Recall* como $\frac{RetRel}{R}$ donde $RetRel$ es el número total de servicios relevantes incluido en la lista de candidatos y R el número total de documentos relevantes. De más esta decir, que si el proceso de descubrimiento devuelve todos los documentos WSDL, se alcanzaría el máximo valor de *Recall*. Sin embargo, aumenta el trabajo del desarrollador, ya que debe buscar los servicios relevantes en la totalidad de la lista de servicios. Por ello, la lista de candidatos se limita a incluir sólo 10 servicios. De esta forma, se busca obtener un buen valor de *Recall* balanceando la cantidad de servicios candidatos y la cantidad de servicios relevantes recuperados, de tal forma que el/la desarrollador/a sólo examine 10 especificaciones de servicios. De este modo, se cumple $RetRel = RetRel_{10}$.

Al igual que en la métrica anterior, el valor de la métrica *Recall* asociado a cada tipo de *query*, se calcula promediando los valores de la métrica obtenido por cada experimento. La Tabla 6.3 detalla los valores promedio de los queries. En la Figura 6.2 se muestra el valor de

Query	*Recall*
Interfaz	93.13 %
Documentación	95.36 %
Argumentos	95.36 %
Dependientes	95.36 %
Completo	95.91 %

Cuadro 6.3: Valor *Recall* de cada tipo de *query*.

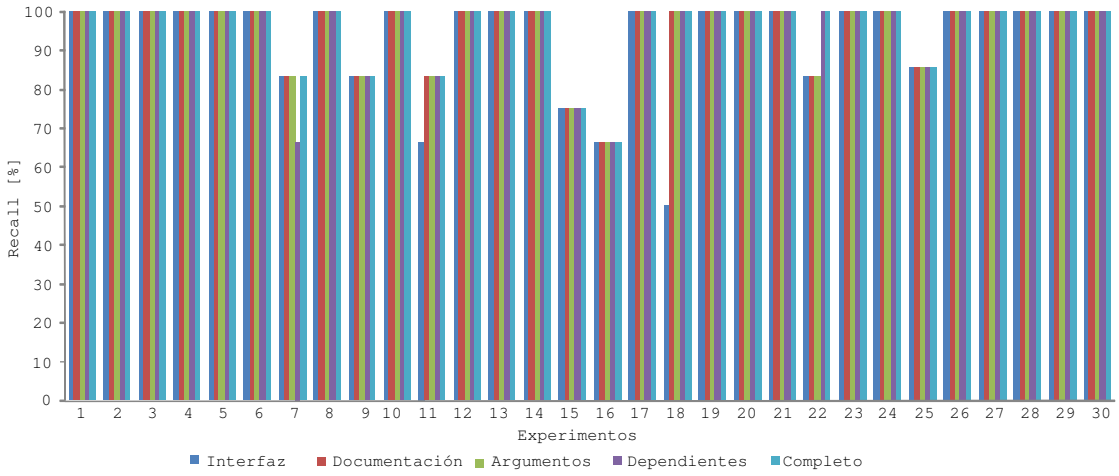


Figura 6.2: Valor de *Recall* de cada experimento

la métrica correspondiente a cada uno de los experimentos. Estos valores demuestran que el enfoque EasySOC incluye la mayoría de los servicios relevantes dentro de la lista de 10 servicios candidatos.

Precision-at-n

La métrica *Precision-at-n* calcula la precisión en diferentes puntos de la lista de candidatos. Por ejemplo, si los 10 primeros documentos de la lista de candidatos son todos relevantes y los siguientes 10 no relevantes, se tiene una precisión del 100 % en la décima posición de la lista, pero una precisión del 50 % en la vigésima.

Formalmente, *Precision-at-n* se define como $\frac{RetRel_n}{n}$ donde $RetRel_n$ es el número de servicios recuperados en las primeras n posiciones.

Al igual que en las métricas anteriores, el valor *Precision-at-n* para cada tipo de query se calcula a partir del promedio de los valores de la métrica de los 30 experimentos. Se ha calculado la métrica para los valores $n = 1, 2, 4, 6, 8, 10$, los cuales se muestran en la Figura 6.3

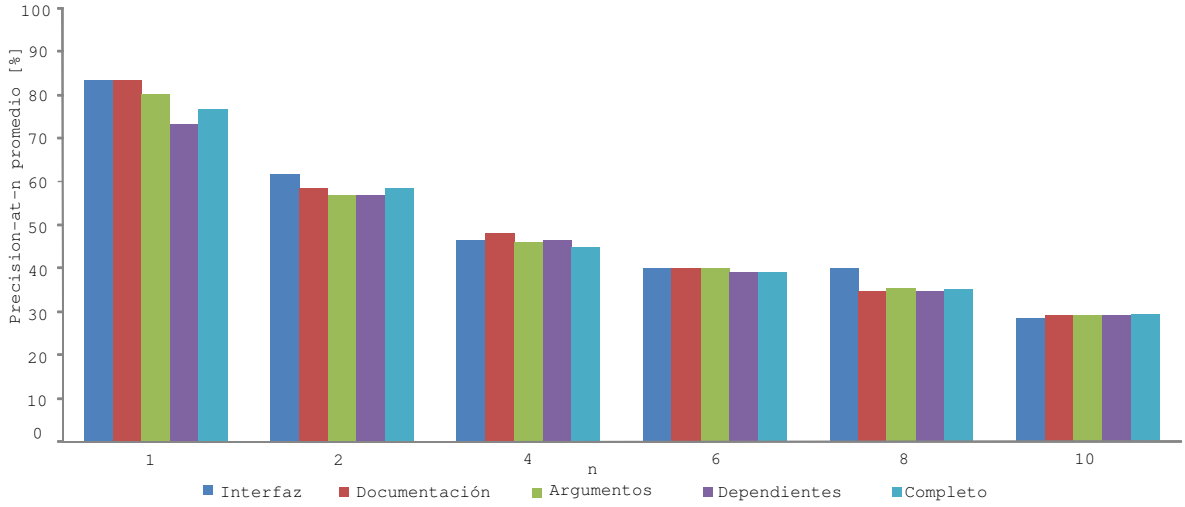


Figura 6.3: Valores de *Precision-at-n* para distintos valores n

.

La Tabla 6.4 detalla los valores promedio de *Precision-at-1* para los queries. Estos valores

Query	*Precision-at-1*
Interfaz	83.33 %
Documentación	83.33 %
Argumentos	80 %
Dependientes	73.33 %
Completo	76.67 %

Cuadro 6.4: Valor Precision-at-1 de cada tipo de *query*.

implican que, por ejemplo, cuando se utiliza el tipo de *query* Documentación ó Interfaz el 83.34 % de los servicios que aparecen al tope de la lista de servicios candidatos son servicios relevantes.

6.1.1. Reordenamiento Estructural y Semántico

EasySOC brinda la posibilidad de realizar un ordenamiento de la lista de servicios candidatos devuelta por el repositorio. Para ello, se utiliza el algoritmo para el cálculo de similitud estructural y semántica entre dos servicios presentado en el capítulo 4.

Una alternativa al análisis planteado anteriormente consiste en volver a calcular las métricas *R-Precision*, *Recall* y *Precision-at-n* para cada uno de los experimentos mencionados anteriormente, habiendo ordenado previamente cada lista de servicios candidatos. El criterio de ordenamiento comprende en ordenar la lista de acuerdo al valor de similitud de cada servicio candidato con respecto a la especificación del componente a tercerizar. Los servicios son ordenados, en forma descendiente, de acuerdo al valor de similitud devuelto por el algoritmo. En otras palabras, el servicio con mayor similitud semántica y estructural quedará al tope de la lista. Cabe destacar que el algoritmo posee un parámetro P que define la importancia estructural y semántica, donde P acepta un valor entre cero y uno. En la Tabla 6.5

P	R-Precision (%)	Recall (%)	Precision-at-1 (%)
Sin ordenamiento estructural	76.1	93.13	83.33
0	47,43	61,01	56,67
0.1	46,32	61,46	56,67
0.2	46,32	61,46	56,67
0.3	43,98	59,65	56,67
0.4	43,43	57,98	53,33
0.5	43,85	56,32	50,00
0.6	41,40	55,68	46,67
0.7	38,43	55,57	43,33
0.8	31,59	55,10	40,00
0.9	26,39	49,03	26,67
1	31,12	52,64	26,67

Cuadro 6.5: Métricas obtenidas luego del reordenamiento de lista de servicios candidatos.

se muestra el valor de las métricas para distintos valores de P. La primera fila de datos de la tablas incluye el valores de las métricas sin haber realizado el ordenamiento. Por otro lado, la búsqueda de servicios se realizó con *queries* de tipo Interfaz.

El análisis de las métricas indica que realizar un ordenamiento de los servicios candidatos según el valor de similitud estructural y semántica con la especificación del componente tercerizado, no aumenta el valor de ninguna de las métricas presentadas en este trabajo. Las métricas tienden a mejorar (sin llegar a equiparar los valores previamente presentados) a medida que el valor de P tiende a cero. Es decir, se le da mayor relevancia a las correspondencia semántica. Sin embargo, la lista de servicios ordenada bajo este criterio indica cuáles servicios son más fáciles de adaptar. Es decir, aquellos servicios al tope de la lista son aquellos servicios que pueden ser adaptados más fácilmente a la interfaz del componente interno. Por lo tanto, este ordenamiento es un complemento al proceso de búsqueda de servicio, brindando al desarrollador una herramienta adicional al momento de seleccionar el servicio externo.

Como se mencionó en 4.3, el algoritmo de similitud estructural y semántica de servicios

es utilizado durante la etapa de generación del componente *Service Adapter*. En ese caso, el valor de la propiedad P debe ser cercano a 1. Es decir, se le da mayor relevancia a las correspondencias estructurales, debido a que el proceso de adaptación de dos servicios está gobernado por los tipos de datos contenidos en las operaciones de los mismos, siendo poco relevante para el proceso la información semántica contenida en la especificación.

6.2. Evaluación de incorporación de servicio candidato

Como se mencionó en 4.4, desarrollar aplicaciones orientadas a servicios utilizando el enfoque EasySOC produce, entre otras ventajas, software con altos niveles de mantenibilidad. El objetivo de esta sección es demostrar empíricamente la afirmación mencionada comparando una aplicación desarrollada con dicho enfoque y una aplicación equivalente desarrollada mediante el enfoque *Contract-First*, descripto en 2.2.2. Los servicios incluidos en la aplicación son algunos de los servicios existentes en la colección de servicios mencionada en la sección anterior. Es importante destacar que ambas aplicaciones utilizan los mismos componentes internos y delegan la misma funcionalidad a tercerizar a los mismos servicios Web en ambos casos.

El objetivo es comparar distintas métricas obtenidas desde el código fuente de la aplicación desarrollada con el enfoque EasySOC con aquellas métricas obtenidas desde el código de una aplicación desarrollada con el enfoque *Contract-First*, con el fin de medir los beneficios del enfoque EasySOC en el mantenimiento del software. Por otro lado, se busca determinar el impacto del cambio de proveedor de alguno de los servicios para cada una de las métricas.

Ambas aplicaciones tienen como requerimientos funcionales calcular un índice a partir de la edad de una persona y enviar esa información a dicha persona. A modo de resumen, la funcionalidad provista por esta aplicación es:

1. Calcular dicho índice numérico para una persona de acuerdo a su edad.
   - *a*) Hallar la edad de una persona.
   - *b*) Dado la edad de una persona y una constante, el índice numérico de la persona es la multiplicación de ambos valores mencionados.
2. Enviar un mensaje a una persona que incluya su índice, calculado en el inciso anterior.

La implementación orientada a servicios de esta aplicación implica que se deban seleccionar e incluir dentro de la aplicación servicios Web de terceros que implementen las siguientes funcionalidades:

- Calcular los años que transcurrieron hasta la fecha actual. Con esta funcionalidad se implementa el inciso 1*a*.
- Calcular la multiplicación entre dos valores. De este modo, se implementa el inciso 1*b*.
- Enviar un mensaje a una persona. Con esta funcionalidad se implementa el inciso 2.

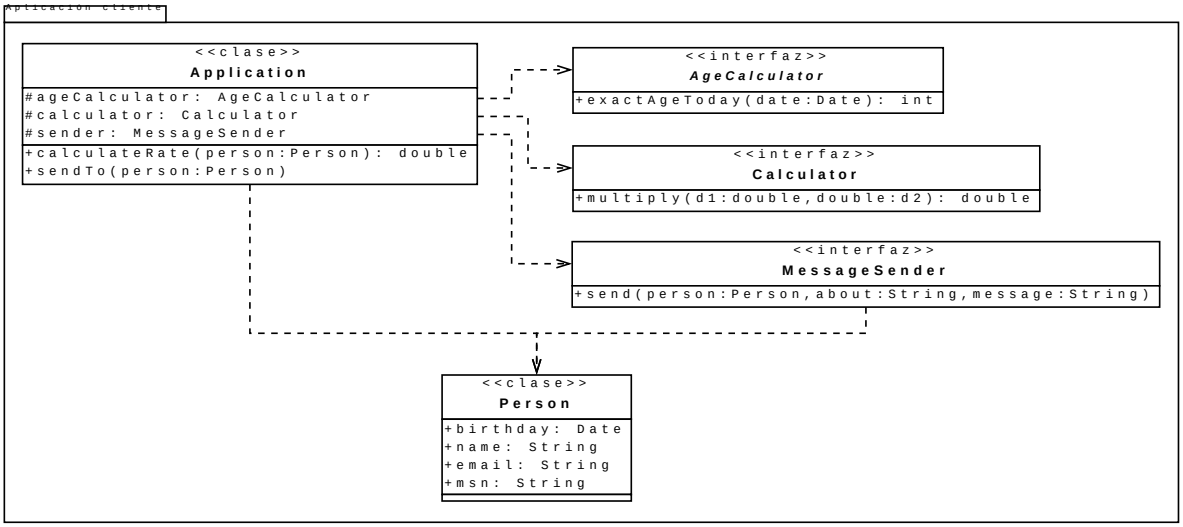


Figura 6.4: Aplicación cliente a tercerizar.

Dado que en general las operaciones aritméticas se realizarían en forma local, vale aclarar que la funcionalidad del ejemplo se eligió con fines orientativos y está condicionada por los servicios presentes en el data-set. La Figura 6.4 detalla la estructura de la aplicación cliente a tercerizar; en la Figura 6.5 se observa la implementación de dicha aplicación bajo el enfoque

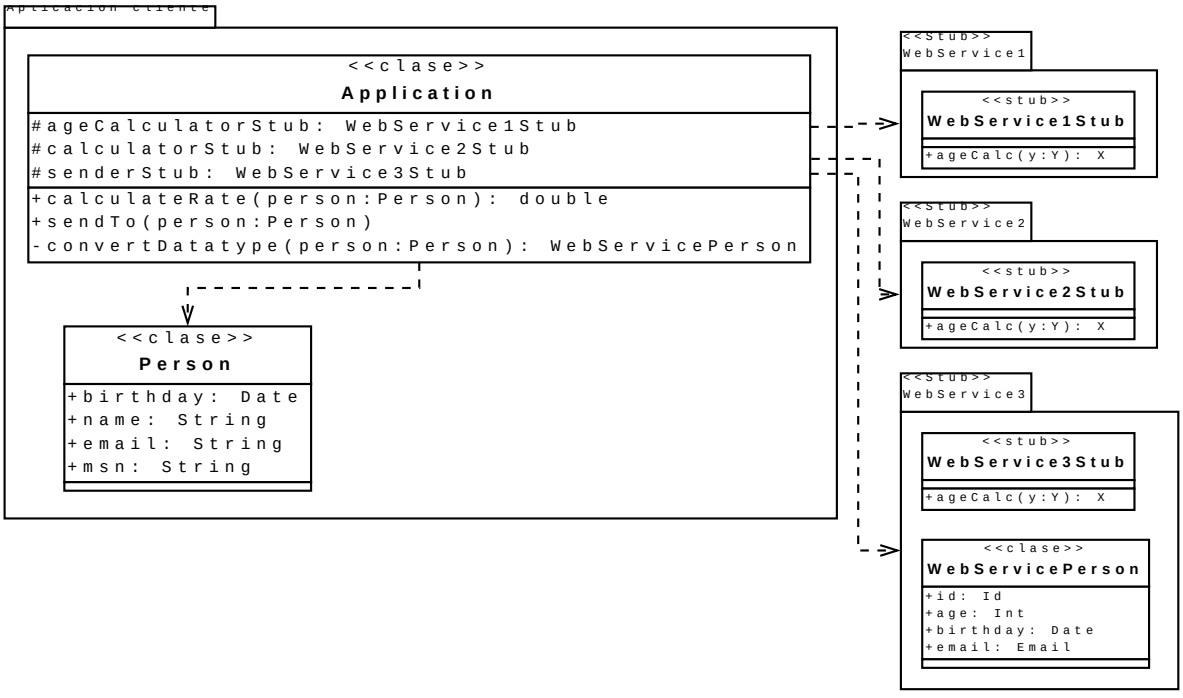


Figura 6.5: Implementación de aplicación utilizando *Contract-First*.

Contract-First, el cual es basado en el enfoque *Proxy* presentado en la sección 2.2.2; en la Figura 6.6 la misma aplicación implementada bajo el enfoque EasySOC, en la cual se puede

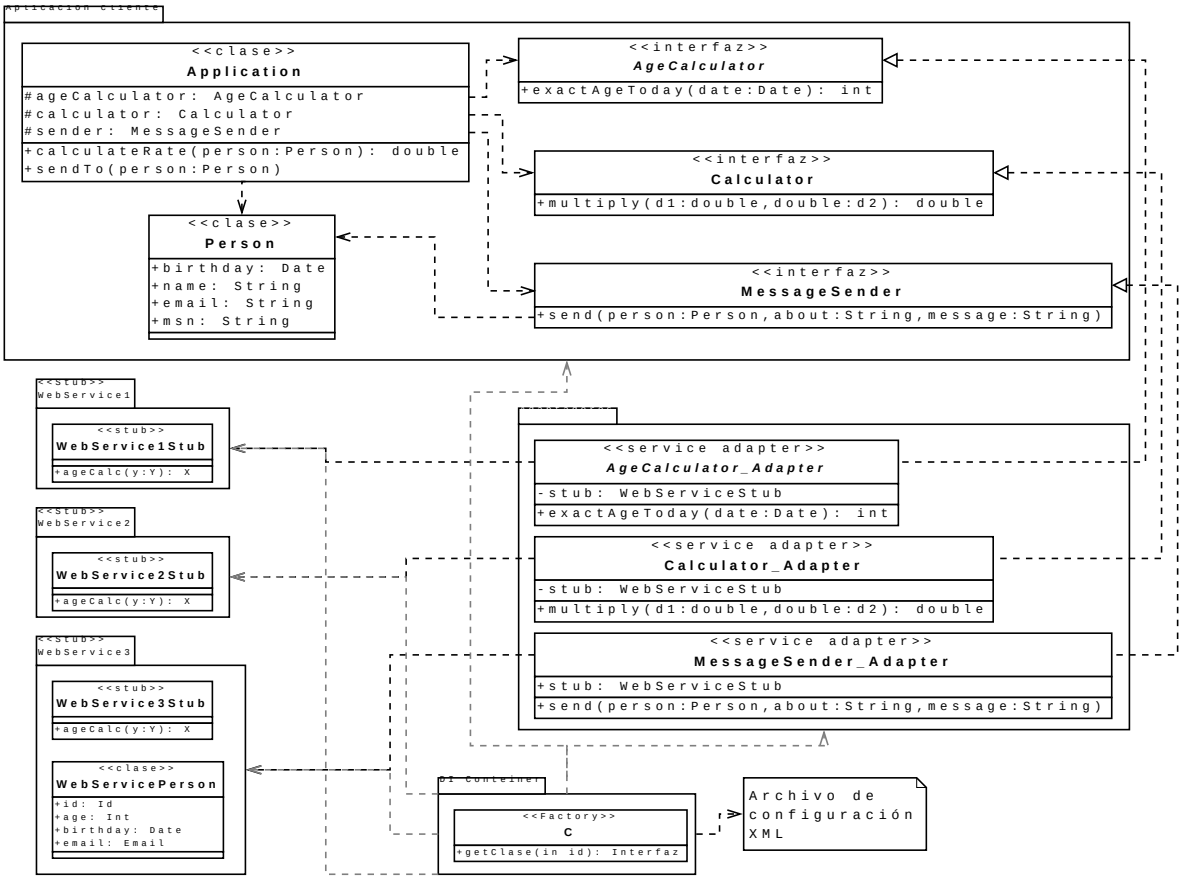


Figura 6.6: Implementación de aplicación utilizando EasySOC.

observar la inclusión de un conteiner responsable de inyectar las dependencias como se explicó en la sección 2.3. Ambas implementaciones han sido desarrolladas bajo la plataforma Java y se ha utilizado la herramienta *Stan4J*[1] para obtener las métricas.

Las métricas obtenidas a partir del análisis del código de las dos aplicaciones orientadas a servicios son:

- Líneas de código fuente (SLOC): cuenta la cantidad de líneas de código de la aplicación, excluyendo líneas en blanco y comentarios. Se considera el código que implementa lógica de la aplicación más el código para interactuar con los servicios Web. Cuanto menor es el valor de la métrica, menor es la cantidad de código necesario para mantener una vez que la aplicación ha sido implementada.
- *Efferent Coupling* (Ce): indica cuánto depende las clases e interfaces dentro de un paquete de las clases e interfaces de otro paquete. En este contexto, como el código de un *Stub* de un servicio en la implementación *Contract-First* no depende del código que implementa la lógica de la aplicación, Ce sólo se refiere al numero de clases/interfaces acopladas que dependen del *Stub* de los servicios de la aplicación.
- Acoplamiento entre Objetos (CBO): es la cantidad de clases en que una clase/interfaz particular está acoplada. Por lo tanto, cuando una clase este menos acoplada a otras clases, hay mas factibilidad de reutilizarla. Como reusabilidad forma parte de mantenibilidad, esta métrica puede ser usada como indicador de cuánto mantenible es el software.
- Repuesta de clase (RFC): cuenta el número de diferentes métodos que pueden ser potencialmente ejecutados cuando un objeto de una determinada clase recibe un mensaje, incluyendo métodos de su jerarquía de clases como así también métodos que pueden invocar a otros objetos. Se puede ver que si la cantidad de métodos que son invocados en respuesta a un mensaje recibido, la actividad de testeo es engorrosa y difícil ya que se necesita un gran nivel de entendimiento del código. Como testeabilidad es también uno de los componentes de mantenibilidad [15], es deseable lograr bajos valores de esta métrica para las distintas clases que forman la aplicación bajo análisis.

En la tabla 6.6 se muestran las métricas detalladas anteriormente para cada una de las apli-

Enfoque utilizado	Ce	SLOC	SLOC/Unidad	CBO	RFC
Contract-First	9	92	46	2.5	14.5
EasySOC	3	146	18.25	1.6	5.12

Cuadro 6.6: Métricas del código fuente.

caciones. Vale aclarar durante el análisis de las métricas no se contempla el código fuente de cada *Stub* de los servicios Web utilizados. Esto se debe a que los servicios utilizados en ambas implementaciones son los mismos, por lo tanto, el código para interactuar con dichos servicios no varía según el enfoque utilizado para implementar la aplicación orientada a servicios.

[1] www.stan4j.com

Análisis de Métricas

En primer lugar, los valores de la métrica Ce para la aplicaciones *Contract-First* y EasySOC son 9 y 3, respectivamente. La principal razón de semejante diferencia entre los valores es que el enfoque EasySOC maneja el paradigma de Inyección de dependencias. El componente *Service Adapter* contiene sólo una referencia a la interfaz del servicio Web, delegando la instanciación del *Stub* al container de la aplicación. Por otro lado, la aplicación *Contract-First*, desarrollada sin en paradigma DI, interactúa no sólo la interfaz del servicio Web sino que también debe instanciar una clase que implemente dicha interfaz, involucrando además interactuar con alguna otra clase particular del servicio en cuestión. Independientemente de la diferencia de valores ó del uso de inyección de dependencias en la aplicación *Contract-First*, si se excluye del análisis de la métrica el componente *Service Adapter*, el valor de la misma para la variante EasySOC se convierte en cero. Esto significa que los restantes componentes de la aplicación no dependen de un servicio Web en particular.

Con respecto a métrica SLOC, los valores de las aplicaciones *Contract-First* y EasySOC son 92 y 146, respectivamente. En la Figura 6.7 se puede observar la cantidad de líneas de códi-

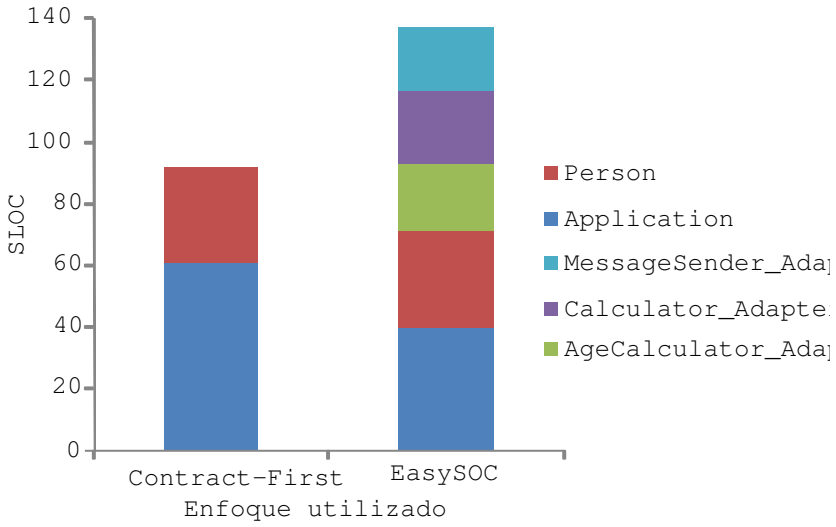


Figura 6.7: Métrica SLOC

gos fuente de las aplicaciones implementadas y, a su vez, se muestra la proporción de líneas de código asociada a cada componente. La diferencia entre la cantidad de líneas de código entre ambas aplicaciones se debe, básicamente, a la implementación de un componente *Service Adapter* por cada servicio Web utilizado en la aplicación EasySOC. Sin embargo, a pesar de la cantidad redundante de líneas de código de la mencionada aplicación, existen muchos aspecto positivos en la utilización del *Service Adapter* que justifican la diferencia entre los valores. Por un lado, gran porcentaje del código del *Service Adapter* es generado automáticamente, lo que significa que el desarrollador no debe invertir esfuerzos en la tarea de programación del componente. Por otro lado, cambiar el proveedor de un servicio en la aplicación EasySOC implica generar un componente *Service Adapter* correspondiente al nuevo servicio Web, mientras que el resto de la lógica de la aplicación se mantiene sin modificaciones. En contraste, el cambio de un servicio Web en la aplicación *Contract-First* involucra modificaciones en las clases en las que el servicio es invocado (por ejemplo, sentencias de

invocación de los métodos del servicio ó conversión de tipos contenidos en la interfaz del servicio). Por lo tanto, esta actividad requiere que el/la desarrollador/a tenga que inspeccionar y modificar gran cantidad de código, pudiendo introducir *bugs* dentro de una aplicación previamente testeada. Cambiar un servicio en la aplicación *Contract-First* implicó la modificación de 19 líneas de código correspondiente al componente interno de la aplicación, las cuales estaban ligadas al servicio reemplazado. Es decir, el 31 % del código fuente del componente sufrió modificaciones. Luego de agregar al código del componente interno el código correspondiente al nuevo servicio, el número de líneas del código fuente se mantuvo igual, debido a que tanto el servicio reemplazado como el reemplazante poseían signaturas similares, por lo que no hubo que agregar código adicional para la adaptación de los tipos de datos.

Una métrica que se deduce de la métrica anterior se denomina SLOC/Unidad, la cual indica la cantidad de lineas de código promedio en cada módulo de la aplicación (clases, interfaces, etc). Los valores correspondientes a la aplicaciones *Contract-First* y EasySOC son 46 y 18.25, respectivamente. El enfoque EasySOC incluye todo el código que interactúa con un servicio dentro del componente *Service Adapter*. Esto implica una correcta modularización de la aplicación, ya que existe un *Service Adapter* por cada servicio Web utilizado. Es decir, existen módulos (eje. *Service Adapter*) que incluyen pequeñas cantidades de código. En la Figura 6.7 se muestra la correcta modularización de la aplicación EasySOC, teniendo 5 clases con altos niveles de cohesión y bajo acoplamiento entre ellas. Dos de estas clases *Application* y *Person* implementan la lógica de la aplicación, mientras que las 3 restantes son clases de adaptación de servicios. Por otro lado, una aplicación implementada con el enfoque *Contract-First* puede no tener una correcta modularización, pudiendo así encontrarse el código para la invocación de uno ó mas servicios mezclado con el código de la lógica de la aplicación dentro de un sólo módulo. Esto afecta claramente la modificabilidad y comprensión del código de la aplicación con tales características. En la Figura 6.7 se muestra que la aplicación está formada por sólo dos clases. Una de ellas, *Application* contiene el 66 % del código total de la aplicación, incluyendo el código para interactuar con los servicios externos.

La métrica CBO ha sido computada a partir de las aplicaciones *Contract-First* y EasySOC obteniendo los valores 2.6 y 1.6 respectivamente. Un alto valor de CBO es extremadamente poco deseable, ya que afecta negativamente la modularidad de la aplicación e impide la reutilización. Más allá del deseo de mantener un bajo acoplamiento entre las clases de la aplicación, se desea llevar al mínimo valor posible el acoplamiento con las clases asociadas a un servicio especifico, debido a que éstas son propensas a cambios. El valor de la métrica para la aplicación EasySOC es menor que aquel asociado a la aplicación *Contract-First*, debido a que todo el código relacionado a un servicio especifico se encuentra incluido dentro de un único módulo (por ejemplo, el *Service Adapter*), el cual tiene altos niveles de cohesión. Esto implica que el enfoque EasySOC aumente la testeabilidad y reusabilidad, debido a que los componentes internos de la aplicación (clases que implementan la lógica) no dependen directamente de los servicios.

Por último, los valores de la métrica RFC asociados a las aplicaciones *Contract-First* y EasySOC son 14.5 y 5.12, respectivamente. Un bajo valor de esta métrica implica mejor testeabilidad y trazabilidad. Además, indica que las clases que implementa la lógica de la aplicación EasySOC suelen ser más fáciles de entender que aquellas de la aplicación *Contract-First*.

Además de las métricas mencionadas, el tiempo de desarrollo de ambas aplicaciones es un

valor a tener en cuenta. Bajo el enfoque EasySOC, la generación de código a partir de servicios Web previamente seleccionados se tradujo a una tarea automática, reduciendo así los tiempos de desarrollo y testeo (ya que también se incorporó de manera automática códigos para realizar test de unidad). Ante un cambio de proveedor de algún servicio, sólo se requirió generar automáticamente un nuevo *Service Adapter* con su correspondiente caso de test. Utilizando este enfoque no se requirió modificar ninguna línea de código existente de la aplicación cliente. Bajo el enfoque *Contract-First*, se debió adaptar los códigos fuente de los componentes internos para poder incorporar el código que interactúa con el servicio externo. El cambio de proveedor de un servicio, requirió volver adaptar los componentes internos para poder interactuar con el nuevo servicio externo, y regenerar código de test para verificar la correctitud del cambio. Esto implicó reemplazar en el código de la lógica de la aplicación, toda referencia a clases específicas del servicio utilizado por aquellas clases del nuevo servicio externo. Además, se debe adaptar la interfaz del nuevo servicio, requiriendo reescribir código de conversión de tipos de datos.

6.3. Conclusiones

En primer lugar, la generación automática de query, mediante la técnica Expansión de query, mejora la precisión de las búsquedas utilizando WSQBE. Precisamente, los valores de la métrica *Recall* varían entre 93.1 % y 95.9 % en todos los casos de Expansión de *query* planteados. Esto implica que se recupere un alto porcentaje de los servicios relevantes para un determinado *query*. Los valores de la métrica *Precision-at-1* nos indica que entre el 73.33 % y 83.33 % de la búsquedas realizadas retornan un servicio relevante al tope de la lista de servicios candidatos.

En segundo lugar, la automatización del proceso de generación de *query* implica la disminución del tiempo de desarrollo de la aplicación. Sin embargo, la técnica Expansión de *query* depende en gran medida de las buenas prácticas que adopten los/as desarrolladores/as (utilización de nombres significativos en variables, documentación del código fuente, etc.), permitiendo generar un *query* preciso.

En tercer lugar, el enfoque EasySOC promueve la generación de código con bajo acoplamiento, alta cohesión y altos niveles de mantenibilidad. Ante un cambio de proveedor de un servicio no se requiere cambios en la lógica de la aplicación ni tampoco en las unidades de testeos asociadas. Esto implica una reducción considerable en los tiempos de desarrollo y mantenimiento, brindando soluciones aplicables en la etapa de implementación de la aplicación como así también en la etapa de testeo de la misma.

CAPÍTULO 7

Conclusiones

En este trabajo se presentó un enfoque denominado EasySOC para el desarrollo de aplicaciones orientadas a servicios. Además, se presentó una herramienta denominada EasySOC-Plugin que implementa dicho enfoque para la plataforma Java.

Considerando las características generales del paradigma Computación Orientada a Servicios (SOC) y una serie de tecnologías utilizadas en la actualidad en la materialización del mismo, presentadas en el capítulo 2, se puede concluir que el uso típico de *frameworks* para consumir servicios externos produce que las aplicaciones orientadas a servicios desarrolladas posean bajos niveles de mantenimiento. El reemplazo de un servicio Web en aplicaciones desarrolladas con estos enfoques es una tarea ardua, ya que requiere la generación de nuevos componentes (denominados *Stubs*) asociados al nuevo servicio de terceros, además de integrarlos al resto de la aplicación, debiendo reescribir código dentro de los componentes que incluyen la lógica de la aplicación. Como consecuencia de esto, se deben volver a testear los distintos componentes de la aplicación para validar que el cambio de proveedores de servicio no ha producido anomalías en el resto de la aplicación. Por lo tanto, el reemplazo de servicios de terceros dentro de este tipo de aplicaciones impacta en distintos componentes de la aplicación (entre ellos componentes lógicos), además de ser una tarea manual costosa y propensa a errores.

El análisis de una serie de trabajos presentado en el capítulo 3 que apuntan a brindar soluciones durante las etapas del ciclo de vida de la aplicaciones orientadas a servicios, determina que dos etapas relevantes en el desarrollo de aplicaciones orientadas a servicios son el descubrimiento de servicios de terceros y la integración de un determinado servicio de tercero dentro de la aplicación en desarrollo. Con el enfoque *Contract-First*, ambas etapas deben ser realizadas de manera manual por el desarrollador. Cada trabajo presentado hace foco en una de estas dos etapas, utilizando la infraestructura actual, sin requerir modificaciones en la misma, lo cual denota la importancia de extender las soluciones existentes. En cuanto a los trabajos de descubrimiento de servicios, la mayoría de ellos se basa en la técnica de recuperación de información denominada Vector Space Model. La diferencia entre

los distintos trabajos que lo adoptan es qué información se recupera y cómo se compara. Por otro lado, los trabajos de incorporación de servicios estudiados utilizan el patrón *proxy* para la invocación de servicio y proponen técnicas complementarias como son DI para lograr bajo acoplamiento entre los distintos componentes de software.

Estos trabajos no brindan soluciones durante la totalidad del ciclo de desarrollo, solo se concentran en alguna de las etapas de este ciclo. A raíz de esto, se presenta el enfoque denominado EasySOC, con el objetivo de brindar asistencia durante el ciclo de desarrollo de aplicaciones orientadas a servicios. En primer lugar, se brinda un proceso semi-automático para la búsqueda de servicios Web de terceros. Uno de los objetivos planteados fue la automatización del proceso de descubrimiento de servicios Web de terceros. Este proceso genera un *query* de búsqueda a partir de componentes propios de la aplicación en desarrollo, lo cual permite al desarrollador/a descubrir con gran precisión servicios que satisfagan la funcionalidad deseada, sin la necesidad de aprender una nueva sintaxis para llevar a cabo dicha búsqueda. Además, la herramienta EasySOCPlugin permite al desarrollador acompañar el proceso de generación de *query* con el fin de obtener un *feedback* continuo, y así generar un *query* que produzca búsquedas de servicios Web aún más precisas. De esta forma, el desarrollador no debe ocuparse de elaborar manualmente *queries* de búsqueda ni realizar búsquedas exhaustivas en los registros UDDI, tarea que suele frustrar a los desarrolladores debido a que los servicios están organizados en categorías que agrupan grandes cantidades de servicios. Este proceso conlleva a un ahorro considerable en tiempo de desarrollo de las aplicaciones, minimizando el esfuerzo realizado por el desarrollador. Los experimentos realizados en este trabajo, han demostrado que el 80 % de las búsquedas de servicios realizadas utilizando el proceso de generación de *query* de EasySOC, descubren un servicio relevante al tope de la lista de servicios Web recuperados.

Por otro lado, el enfoque EasySOC facilita la tarea de integración de un servicio Web dentro de la aplicación en desarrollo. Uno de los objetivos del enfoque es que la aplicación resultante posea altos niveles de mantenibilidad. Más precisamente, el cambio de proveedor de un servicio Web no debe repercutir sobre los componentes que implementan la lógica de la aplicación. Para ello, ante la necesidad de incorporar un servicio dentro de la aplicación, el enfoque genera el código de un componente específico a un servicio Web denominado *Service Adapter*. Este componente, encargado de adaptar la interfaz de un servicio Web a incorporar a la interfaz del componente que el desarrollador decide tercerizar su implementación, permite encapsular las invocaciones al servicio Web; mientras que los componentes que implementan la lógica consumen los servicios Web de manera implícita, mediante invocaciones a esta capa intermedia que proveen los *Service Adapter*. Por otra parte, el uso del patrón de Inyección de dependencias permite al enfoque EasySOC ensamblar, en tiempo de ejecución, los distintos componentes internos de la aplicación en desarrollo (los cuales implementan la lógica del negocio) con aquellos componentes *Service Adapter,* los cuales representan a los distintos servicios externos. De esta forma, los componentes lógicos poseen una dependencia dinámica con los servicios. La separación entre los componentes que implementan la lógica y aquellos que interactúan con servicios Web particulares (materializados por los *Service Adapter*) propuesta por el enfoque, logra altos niveles de mantenibilidad. Esto se debe a que el reemplazo de un servicio Web de un tercero implica sólo la generación de un nuevo componente *Service Adapter* (manteniendo la misma interfaz pero su código interactúa con el nuevo servicio Web) y modificar la dependencia existente por este nuevo componente. Al no alterar la interfaz del *Service Adapter*, no hay necesidad de modificar ningún componente

lógico.

Para comprobar que las aplicaciones desarrolladas con el enfoque EasySOC tienen mayor mantenibilidad que aquellas desarrolladas con enfoques tradicionales, se han implementado dos aplicaciones equivalentes presentadas en el capítulo 6, una de ellas utilizando el enfoque *Contract-First* y la restante utilizando el enfoque EasySOC. Luego, los valores brindados por distintas métricas relacionadas al mantenimiento de software para cada una de las aplicaciones demostraron que la aplicación EasySOC posee módulos con bajo acoplamiento, alta cohesión y altos niveles de mantenibilidad en comparación al otro enfoque.

El enfoque EasySOC presenta un algoritmo de adaptación de interfaces (también denominado algoritmo de *matching*) para encontrar una manera de adaptar la interfaz del componente a tercerizar y la interfaz de un servicio Web. La salida del algoritmo es un documento formal que indica cómo se debe realizar la adaptación entre dos interfaces. Este documento de adaptación es utilizado en la construcción del componente *Service Adapter*. La herramienta EasySOCPlugin provee un proceso automático para la creación de un componente *Service Adapter* de un servicio Web, además de los archivos de configuración propios de un Container de Inyección de dependencias. Por otra parte, EasySOCPlugin ofrece la posibilidad que el usuario infiera en el proceso de creación del componente *Service Adapter*. También, ofrece la posibilidad de visualizar cómo se realiza la adaptación de interfaces y editar el documento de adaptación de interfaces creado por la herramienta a partir de aplicar el algoritmo de *matching*.

Además, EasySOCPlugin ofrece la posibilidad de crear test de unidad por cada componente lógico que interactúe con servicios Web externos, como así también la creación de un test de unidad relacionado al componente *Service Adapter*. De esta forma, se puede verificar la consistencia de la aplicación ante un cambio de proveedor de un servicio Web, ejecutando los test de unidad creados por la herramienta.

En conclusión, la ventaja del la utilización del enfoque EasySOC y la correspondiente herramienta EasySOCPlugin en el desarrollo de aplicaciones orientadas a servicios, es que el desarrollador concentra la mayor parte del tiempo de desarrollo en la creación de los distintos componentes lógicos de la aplicación, minimizando extremadamente los tiempos y complejidad de tareas típicas de desarrollo de aplicaciones de este tipo como son la búsqueda e integración de servicios de terceros. Estas tareas eran unas de las grandes falencias de los enfoques tradicionales de desarrollo como *Contract-First*. Por otra parte, el alto nivel de mantenimiento que poseen las aplicaciones generadas con el enfoque, permite simplificar la etapa de mantenimiento de una aplicación orientada a servicios, la cual puede insumir entre un 60-80 % del ciclo de vida del software.

7.1. Limitaciones

Una de las mayores limitaciones de la herramienta se presenta al encontrar inconsistencias en la sintaxis de los documentos WSDL, debido a las diferencias de criterios que existen entre las distintas herramientas que generan una especificación WSDL correspondiente a una pieza de software. Aquellos servicios cuyo documento de especificación WSDL posean características propias que no formen parte del estándar recomendado por parte del W3C[1],

[1] http://www.w3.org/

pueden no ser debidamente adaptados por EasySOCPlugin, pudiendo producir inconvenientes al momento de generar el código del componente *Service Adapter*.

Por otra parte, existen casos en que la adaptación entre la interfaz del componente tercerizado y la interfaz del servicio Web a incluir dentro de la aplicación no puede llevarse a cabo plenamente. Es decir, no se logra realizar una transformación entre un tipo de datos de la interfaz del componente y otro del servicio, debido a que existen incompatibilidades entre ellos. Debido a esto, el componente *Service Adapter* generado, no incluye código de transformación entre los tipos de datos incompatibles, pero si añade sentencias de comentarios dentro del código del componente para alertar del inconveniente. Ante esta situación, el desarrollador debe continuar la adaptación de las interfaces manualmente proveyendo código de transformación de datos adecuado dentro del código de componente *Service Adapter* generado.

Otra limitación presente en la herramienta es la presencia de listas con tipos de datos parametrizados en la interfaz del componente a tercerizar. Este déficit se debe a que la herramienta utilizada para obtener la especificación WSDL correspondiente a una interfaz no contempla esa característica. Por lo tanto, se acarrean problemas al momento de realizar la adaptación entre dicha interfaz y aquella del servicio Web seleccionado, como así también al momento de generar código correspondiente al *Service Adapter*.

Por otro lado, la adaptación entre dos interfaces, si bien puede ser correcta, y ajena a errores y anomalías, puede ser distinta a una adaptación que espera el desarrollador. Por lo tanto, el desarrollador puede customizar la adaptación a través del documento de adaptación de interfaces, o bien, reescribir código correspondiente al *Service Adapter* generado.

Con respecto a la búsqueda de servicios Web, la creación de *queries* precisos por parte del proceso de generación de *queries* depende en gran medida de las buenas prácticas que adopten los desarrolladores para especificar los distintos componentes del sistema. En particular, los nombres de los componentes, como así también sus atributos y métodos, deben tener nombres significativos a la funcionalidad que estos ofrecen. Además, documentar el código fuente ayuda a mejorar la precisión del mecanismo de búsqueda.

EasySOCPlugin requiere que las aplicaciones orientadas a servicios desarrolladas con EasySOC utilicen el *framework* Spring, como Container de inyección de dependencias. Esto se debe a que se requiere que la aplicación se encuentre en ejecución bajo un contexto de inyección de dependencias, para poder agregar los nuevos componentes generados (*Service Adapters* y *Stubs*) dentro de dicho contexto.

7.2. Trabajos futuros

El enfoque EasySOC propuesto y la herramienta EasySOCPlugin en particular, permite que cada módulo de la arquitectura sea mejorado a futuro de manera independiente. Con lo cual, por ejemplo, se podrían agregar nuevos métodos de expansión de queries, o modificaciones al algoritmo de adaptación de interfaces.

Una de las limitantes que presenta la herramienta en la actualidad, es no permitir el uso de listas parametrizadas dentro del código de la interfaz del componente a tercerizar. A futuro, con la maduración de las herramientas generadoras de especificaciones WSDL se incorporará esta característica.

La herramienta EasySOCPlugin permite previsualizar la adaptación de interfaces, se agregará la posibilidad de editar la adaptación desde la misma previsualización . De este modo, el desarrollador podrá indicar la manera de adaptar la interfaz del componente interno a tercerizar y la del servicio Web a integrar de una manera gráfica sin la necesidad de editar el documento de adaptación de interfaces.

Bibliografía

[1] Ricardo Baeza-Yates and Berthier Ribeiro-Neto. *Modern Information Retrieval*. Addison-Wesley, 1999.

[2] Len Bass, Paul Clements, and Rick Kazman. *Software Architecture in Practice*. Addison-Wesley Professional, December 1997.

[3] María Agustina Cibrán, Bart Verheecke, Wim Vanderperren, Davy Suvée, and Viviane Jonckers. Aspect-oriented programming for dynamic Web Service selection, integration and management. *World Wide Web*, 10(3):211–242, 2007.

[4] M. Crasso, A. Zunino, and M. Campo. Easy Web Service discovery: a Query-by-Example approach. *Science of Computer Programming*, To appear, 2008.

[5] M. Crasso, A. Zunino, and M. Campo. Query by example for web services. In *Proceedings of the* 23^{rd} *Annual ACM Symposium on Applied Computing, Special Track on Web Technologies*, 2008.

[6] Xin Dong, Alon Halevy, Jayant Madhavan, Ema Nemes, and Jun Zhang. Similarity search for web services. 2004.

[7] David C. Fallside and Priscilla Walmsley. XML Schema part 0: Primer second edition. W3C recommendation, World Wide Web Consortium, October 2004.

[8] Robert E. Filman, Stuart Barrett, Diana D. Lee, , and Ted Linden. Inserting ilities by controlling communications. 2005.

[9] R. Tangi G. A. Miller, C. Fellbaum. Wordnet.

[10] Zvi Galil. Efficient algorithms for finding maximum matching in graphs. *ACM Comput. Surv.*, 18(1):23–38, 1986.

[11] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. Design patterns: Abstraction and reuse in object-oriented designs. pages 406–431, 1993.

[12] John Garofalakis, Yannis Panagis, Evangelos Sakkopoulos, and Athanasios Tsakalidis. Contemporary Web Service Discovery Mechanisms. *Journal of Web Engineering*, 5(3):265–290, 2006.

[13] Andreas Heß, Eddie Johnston, and Nicholas Kushmerick. Assam: A tool for semi-automatically annotating semantic web services. In *In Intl. Semantic Web Conf. (ISWC*, pages 320–334, 2004.

[14] E. Tempero y H.Melton H.Y. Yang. An empirical study into use of dependency injection in java. *19th Australian Conference on Software Engineering*, 2008.

[15] International Organization for Standardization. Software Engineering - Product Quality - Part 1: Quality Model. 2001.

[16] M. Stevens J. McGovern, S. Tyagi and S. Matthew. *Java Web Service Architecture*. Morgan Kaufmann Publishers, 2003.

[17] T. Joachims. A probabilistic analysis of the rocchio algorithm with tdidf for text categorization. *International Conference on Machine Learning, Morgan Kaufman, Nashville, Tennesee,USA*, pages 143–151, 1997.

[18] R. Johnson. J2ee development frameworks. *ITSystem Perspectives*, 2005.

[19] Y. Hwang K. Lee K. Lee, M. Lee. A framework for xml web services retrieval with ranking. *International Conference on Multimedia and Ubiquitous Engineering (MUE)*, 2007.

[20] Daniel R. Kahan, Michael F. Nowlan, and M. Brian Blake. Taming web services in the wild. *IEEE International Conference on Web Services (ICWS'06)*, pages 957–958.

[21] Gregor Kiczales. Aspect-oriented programming. *ACM Comput. Surv.*, 28(4es):154, 1996.

[22] M. P. Singh M. N. Huhns. Service-oriented computing: Key concepts and principles. *IEEE Intenet Computing*, (1):75–81, 2005.

[23] Cristian Mateos, Marco Crasso, Alejandro Zunino, and Marcelo Campo. Supporting ontology-based semantic matching of web services in movilog. In Jaime SimÃ£o Sichman, Helder Coelho, and Solange Oliveira Rezende, editors, *IBERAMIA-SBIA*, volume 4140 of *Lecture Notes in Computer Science*, pages 390–399. Springer, 2006.

[24] Rob McCool. Rethinking the Semantic Web, part II. *IEEE Internet Computing*, 10(1):96, 93–95, 2006.

[25] Hamid Reza Motahari Nezhad, Boualem Benatallah, Axel Martens, Francisco Curbera, and Fabio Casati. Semi-automated adaptation of service interactions. In *WWW '07: Proceedings of the 16th international conference on World Wide Web*, pages 993–1002, New York, NY, USA, 2007. ACM.

[26] OASIS Consortium. UDDI Version 3.0.2. UDDI Spec Technical Committee Draft, October 2004.

[27] Massimo Paolucci and Katia Sycara. Autonomous semantic Web services. *IEEE Internet Computing*, 7(5):34–41, 2003.

[28] C. Platzer and S. Dustdar. Vector space search engine for web services. *Proceedings of the Third European Conference on Web Services (ECOWS)*, 2005.

[29] Alan LaMont Pope. *The CORBA Reference Guide: Understanding the Common Object Request Broker Architecture*. Addison-Wesley, Boston, MA, USA, 1998.

[30] M. Porter. An algorithm for suffix stripping. *Program*, 14:130–137, 1980.

[31] E. Razina and D.Janzen. Efects of dependency injection on maintainability.

[32] Marisol Pérez Reséndiz and José Oscar Olmedo Aguirre. Dynamic invocation of web services by using aspect-oriented programming. 2005.

[33] Fabrizio Sebastiani. Machine learning in automated text categorization. *ACM Computing Surveys*, 34(1):1–47, 2002.

[34] Mehrnoush Shamsfard and Ahmad Abdollahzadeh Barforoush. Learning ontologies from natural language texts. *International Journal of Human-Computer Studies*, 60:17–63, 2004.

[35] Kaarthik Sivashanmugam, Kunal Verma, Amit P. Sheth, and John A. Miller. Adding semantics to Web Services standards. In Liang-Jie Zhang, editor, *The 2003 International Conference on Web Services*, pages 395–401, Las Vegas, NV, USA, September 2003. CSREA Press.

[36] Andrew E Wade. Distributed client-server databases. *Object Magazine 4*, 1:47–52, April 1994.

[37] Y. Wang and E. Stroulia. Flexible interface matching for web-service discovery. *Fourth International Conference on Web Information Systems Engineering (WISE)*, 2003.

[38] Y. Wang and E. Stroulia. Structural and semantic matching for assessing web-service similarity. *International Journal of Cooperative Information Systems*, 2004.

[39] Sanjiva Weerawarana, Francisco Curbera, Frank Leymann, Tony Storey, and Donald F. Ferguson. *Web Services Platform Architecture: SOAP, WSDL, WS-Policy, WS-Addressing, WS-BPEL, WS-Reliable Messaging, and More*. Prentice Hall PTR, 2005.

[40] René Witte, Qiangqiang Li, Yonggang Zhang, and Juergen Rilling. Text mining and software engineering: An integrated source code and document analysis approach. *IET Software Journal*, 2:3–16.

[41] W. Z. SKM Wong and patrick Wong. Generlized vector space model in information retrieval. 2005.