UNIVERSIDADE FEDERAL DE CAMPINA GRANDE
CENTRO DE CIÊNCIAS E TECNOLOGIA
COORDENAÇÃO DE PÓS-GRADUAÇÃO EM INFORMÁTICA

Uma ferramenta de apoio ao desenvolvimento de *Web Services*

Andrés Ignácio Martinez Menéndez

Área de concentração: Ciência da Computação

Linha de Pesquisa: Redes de Computadores e Sistemas Distribuídos

Campina Grande - PB
Agosto / 2002

UNIVERSIDADE FEDERAL DE CAMPINA GRANDE
CENTRO DE CIÊNCIAS E TECNOLOGIA
COORDENAÇÃO DE PÓS-GRADUAÇÃO EM INFORMÁTICA

Uma ferramenta de apoio ao desenvolvimento de *Web Services*

Andrés Ignácio Martínez Menéndez

Dissertação submetida à Coordenação de Pós-Graduação em Informática do Centro de Ciências e Tecnologia da Universidade Federal de Campina Grande como requisito para obtenção do grau de mestre em Ciências (MSc).

Orientador: Jacques Philippe Sauvé

Campina Grande - PB
Agosto / 2002

FICHA CATALOGRÁFICA

MENÉNDEZ, Andrés Ignácio Martínez

M538W

Uma ferramenta de apoio ao desenvolvimento de Web Services

Dissertação (mestrado), Universidade Federal de Campina Grande, Centro de Ciências e Tecnologia, Coordenação de Pós-Graduação em Informática, Campina Grande, Agosto de 2002-08-30

97 p. Il.

Orientador: Jacques Philippe Sauvé

Palavras-chaves:

1. Engenharia de Software
2. Desenvolvimento web
3. Web Services

CDU – 519.683

"UMA FERRAMENTA DE APOIO AO DESENVOLVIMENTO DE WEB SERVICES"

ANDRÉS IGNÁCIO MARTÍNEZ MENÉNDEZ

DISSERTAÇÃO APROVADA EM 26.08.2002

PROF. JACQUES PHILIPPE SAUVÉ, Ph.D
Orientador

PROF. MARCUS COSTA SAMPAIO, Dr.
Examinador

PROF. ÁLVARO FRANCISCO DE C. MEDEIROS, D.Sc
Examinador

CAMPINA GRANDE – PB

A minha avô Elisa Menéndez que sempre serviu como exemplo de dedicação para toda a família.

Agradecimentos especiais vão para meus pais Angel e Graça, minha irmã Elisa e meu filho Andrés por todos momentos que os privei da minha presença nesses dois anos e aos colegas do DTI, os quais considero a minha segunda família.
São raras as oportunidades que temos para prestar homenagem às pessoas queridas e neste pequeno espaço é impossível colocar todos aqueles que, de alguma forma, contribuíram para que este trabalho esteja concluído. A todos vocês as minhas mais sinceras desculpas.

Sumário

Lista de siglas

API	Application Programming Interface
B2B	Business to business
CEP	Código de Endereçamento Postal
CORBA	Commom Object Request Broker Arquitecture
CPF	Cadastro de Pessoa Física
DII	Dynamic Invocation Interface
DOM	Document Object Model
DTD	Document Type Definition
EJB	Enterprise Java Beans
HTML	Hyper Text Markup Language
HTTP	HyperText Transport Protocol
IDE	Integrated Development Enviroment
J2EE	Java 2 Enterprise Edition
J2SE	Java 2 Standard Edition
JAXM	Java API for XML Messaging
JAXP	Java API for XML Processing
JAXR	Java API for XML Registries
JAX-RPC	Java API for XML-based RPC
JDBC	Java Database Connectivity
JSP	JavaServer Pages
JWSDP	Java Web Service Developer Pack
NAICS	North American Industry Classification System
RMI	Remote Method Invocation
RPC	Remote Procedure Call
SAX	Simple API for XML Parsing
SOAP	Simple Object Access Protocol
UDDI	Universal Description Discovery and Integration
UML	Unified Modeling Language
URI	Uniform Resource Identifiers
URL	Uniform Resource Locator
WASP	Web Application and Service Plataform

WSDL	Web Service Definition Language
XML	Extensive Markup Language
WSTK	Web Service Toolkit

Lista de figuras

Lista de Tabelas

Resumo

Web services são componentes de software que oferecem uma funcionalidade específica. A grande vantagem é que estes componentes podem ser acessados por diferentes sistemas, através de padrões da Internet, como HTTP, XML e SOAP. A utilização desses padrões é que permite criar aplicações distribuídas com o uso de *web services*. Este trabalho explica em detalhes as tecnologias envolvidas na criação de *web services* e mostra a implementação de uma ferramenta de gerenciamento para os mesmos.

Abstract

Web Services are software components that provide an especific funtionality. The great profit is that this components can be access by other systems, over Internet standards, like HTTP, XML and SOAP. The utilization of this standards allow to create distributed applications with web services. This work explain in details the tecnologies used in web services creation and show an implementation of a management tool for them.

Capítulo 1

Introdução

1.1 Introdução

De tempos em tempos o mundo da tecnologia de informação passa por mudanças radicais. Foi assim com o paradigma da orientação a objetos, a abordagem cliente/servidor, a Internet, entre outros. Estamos novamente à beira de mais uma mudança e o responsável por esta transformação é chamado de *web service*.

As tecnologias atuais permitem a busca de informações na Web através de links HTML; porém essas consultas devem ser realizadas com a intervenção humana por meio de software cliente, a exemplo do *browser*. Os *web services* têm como proposta permitir que as aplicações possam procurar informação sem a intervenção de pessoas e interagir uma com as outras. O advento dos *web services* preencherá uma lacuna existente atualmente: uma interface *machine-friendly* para a web [1], permitindo que aplicações possam trocar informações.

O uso de *web services* pelas empresas vai permitir que sistemas informatizados possam se comunicar de maneira a agilizar transações comerciais. Um cenário típico do uso de *web services* é o envio de uma solicitação para um serviço publicado em uma URL via HTTP, usando o protocolo SOAP (*Simple Object Access Protocol*). O *web service* vai receber a solicitação e processá-la; no final ele deverá retornar uma resposta também usando SOAP.

A definição da interface de um *web service* é feita através de WSDL (*Web Service Definition Language*). Além disso, existem serviços web especializados em publicar e descobrir *web services*, chamados de *service provider*. Eles podem ser considerados como as

páginas amarelas dos *web services*. Um *service provider* deve implementar o padrão UDDI (*Universal Description Discovery and Integration*) para a publicação do serviço. Na Figura 1 podemos uma aplicação cliente acessando um web service que pode ser encontrado em um *service provider.*

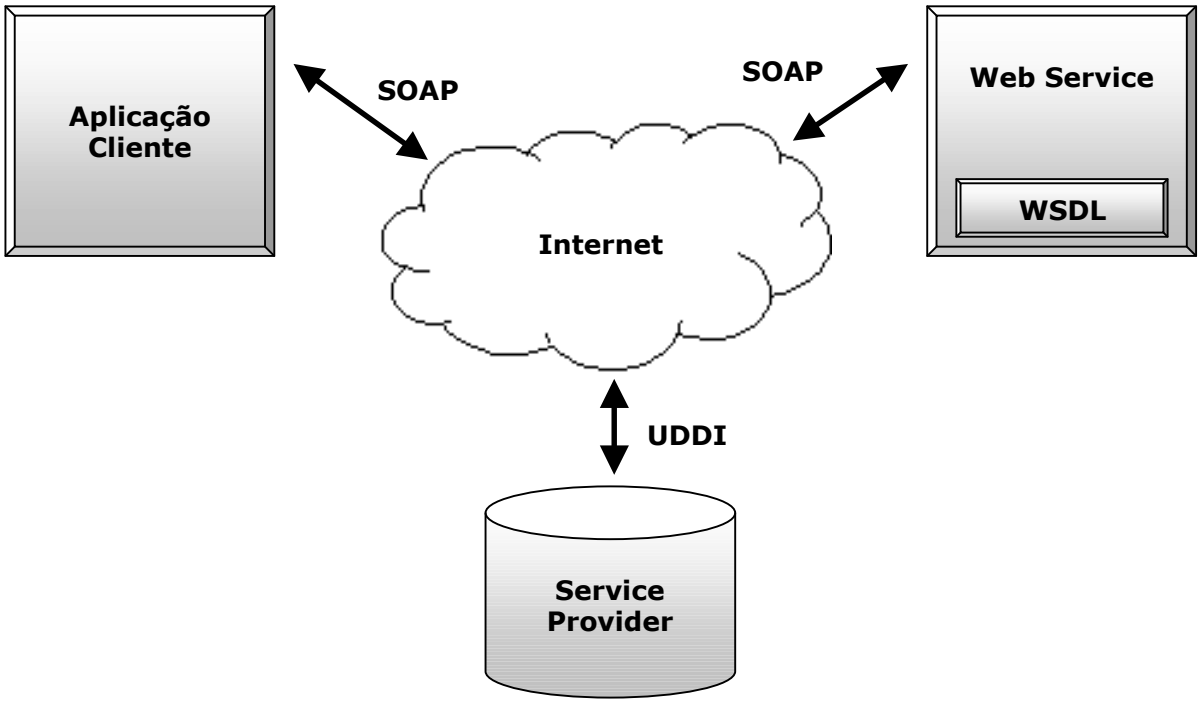

Figura 1 – Esquema conceitual de um web service

1.2 Objetivo da dissertação

Fazendo uma análise no nível de informatização nas empresas, podemos perceber que a maioria dos sistemas atende as necessidades internas, isto é, eles funcionam da empresa para dentro. O movimento de globalização obrigou as empresas a se tornarem mais competitivas e um dos pontos que medem essa competitividade é o seu nível de informatização. Porém, os sistemas não devem ser fechados e nem voltados exclusivamente para dentro da empresa. Interoperabilidade é a palavra da vez e a preocupação em colocar os sistemas para conversar entre si será um dos pontos mais fortes no desenvolvimento de sistemas nos próximos anos [1].

Atualmente, a interoperabilidade está em um estágio pouco avançado. Poucos são os sistemas que conseguem conversar com outros para a realização de transações. Os sistemas que se enquadram neste caso o conseguiram a muito custo, com soluções proprietárias que os deixou com pouca ou nenhuma flexibilidade.

A tecnologia de *web services* é baseada em padrões da indústria de informática e a sua abordagem simplifica a colaboração em transações B2B (*business to business*). Dessa forma, o uso de *web services* vai permitir a comunicação entre sistemas de uma forma simples e, o que é melhor, seguindo padrões [2].

O uso de *web services* permitirá que os sistemas possam se comunicar entre si. Porém, colocar um *web service* funcionando para aceitar requisições não é uma tarefa simples. É necessário definir e criar as interfaces usando WSDL, configurar o servidor de aplicação, publicar o serviço em um registro UDDI, preparar-se para receber e responder solicitações usando SOAP, entre outras tarefas.

Para facilitar a geração de *web services* é necessário ter ferramentas que automatizem os processos que foram comentados. Este trabalho tem como objetivo criar uma ferramenta de gerenciamento de *web services*. Ela permitirá definir e publicar web services de uma forma simples e rápida, deixando os complicados detalhes de implementação escondidos do desenvolvedor.

1.3 Escopo e relevância

Embora seja possível desenvolver *web services* em várias linguagens de programação, neste trabalho a criação é feita com a linguagem de programação Java, por meio das APIs disponibilizadas no pacote Java Web Services Developer Pack (JWSDP) da Sun Microsystems. Isto se deve ao fato das tecnologias baseadas Java estarem em um estágio mais desenvolvido para a criação de *web services* do que os seus concorrentes, como por exemplo, as tecnologias .NET da Microsoft.

Embora os *web services* tenham surgido para resolver a necessidade de interoperabilidade entre as aplicações, a tecnologia envolvida não é simples de ser absorvida.

A ferramenta construída neste trabalho servirá para esconder os complicados detalhes de criação *web services* além de melhorar a produtividade do desenvolvedor.

1.4 Descrição da estrutura do trabalho

Este trabalho é divido em sete capítulos. Veremos, a seguir, uma breve descrição de cada um deles.

O Capítulo 1 faz uma breve introdução da tecnologia de *web services*. Descreve a necessidade de comunicação entre aplicações de maneira a agilizar as transações *business-to-business*. Cita as dificuldades de trabalhar com *web services* sem o uso de ferramentas automatizadas e mostra que o objetivo da dissertação é a criação de interfaces que auxiliem em todo o processo de desenvolvimento.

No segundo capítulo, o enfoque é voltado para as tecnologias e os padrões que são utilizados nos *web services*: XML, UDDI, SOAP e WSDL. O capítulo faz uma explanação sobre cada tecnologia individualmente e como elas podem ser combinadas para o desenvolvimento de *web services*.

O desenvolvimento de *web services* é o assunto abordado no Capítulo 3. Serão mostrados os problemas encontrados quando são criados *web services* sem o uso de ferramentas. Veremos também alguns pacotes existentes atualmente, os quais auxiliam os desenvolvedores na criação de serviços. O capítulo é finalizado com um levantamento dos problemas existentes nas ferramentas de modo a fazer um levantamento de requisitos para a criação de uma nova ferramenta que atenda às necessidades encontradas.

O Capítulo 4 tem como objetivo mostrar o pacote integrado desenvolvido pela Sun Microsystems – JWSDP (Java Web Service Developer Pack) e as suas principais APIs. Será mostrado como utilizar os principais serviços disponibilizados pela ferramenta e quais são os artefatos que é capaz de gerar.

No quinto capítulo veremos em detalhes a ferramenta de gerenciamento de *web services* que este trabalho propõe como solução. Será mostrado a sua arquitetura, os seus

principais componentes e a sua forma de trabalho. Será mostrado, com exemplos, como implantar um serviço no servidor web e como publicar registros UDDI referentes ao *web service*.

O uso prático da ferramenta desenvolvida é visto no Capítulo 6. Serão mostrados os passos necessários para a criação e publicação de um *web service* fictício de reserva de hotéis. Será mostrada também uma pequena aplicação de uma agência de turismo, que faz uso do *web service* publicado.

O Capítulo 7 discute os resultados da dissertação. Fala sobre as conclusões alcançadas, dos problemas que foram enfrentados durante todo o trabalho e finaliza com várias propostas de trabalhos futuros envolvendo o uso de *web services*.

Capítulo 2

Tecnologias para a construção de *Web Services*

Este capítulo tem dois objetivos: primeiro, mostrar as vantagens que o uso de *web services* proporciona nas transações realizadas entre aplicações; em segundo lugar, ele vai explicar individualmente cada tecnologia e como elas podem ser combinadas para a construção de *web services*.

2.1 O que é um *web service*

Para poder explicar o que é um *web service*, vamos analisar alguns exemplos de transações que ocorrem atualmente.

No primeiro exemplo tomemos o ponto de vista de uma loja virtual que atende seus clientes através do envio de produtos pelo correio. Em algum momento da compra o cliente terá que informar o endereço da entrega. Nesse momento o Código de Endereçamento Postal (CEP) exerce um papel fundamental, já que ele nos indica o estado, a cidade, o bairro e a rua do destinatário. Somente com a posse desses elementos é que a loja virtual pode calcular o valor do frete a ser pago para o envio do produto. Uma solução para a loja seria armazenar na sua base de dados todos os CEPs brasileiros, isto é, ter uma cópia da base utilizada pelos Correios. Evidentemente que esta solução trará muitos problemas de atualização, já que uma mudança na base dos Correios deverá ser refletida na loja virtual.

Qual seria o papel dos *web services* neste caso? Os Correios poderiam criar um *web service* que recebesse uma mensagem contendo o valor de um CEP e retornasse uma outra mensagem com os dados referentes ao CEP indicado. A aplicação da loja virtual teria que enviar uma solicitação com o CEP do cliente e esperar pela resposta dos Correios para poder calcular o frete. Neste caso todos sairiam ganhando: os correios receberão um endereço

correto, facilitando a entrega; a loja vai agilizar sua aplicação e o cliente receberá o seu pedido muito mais rapidamente.

Atualmente o site dos correios na Internet, encontrado em www.correios.com.br, disponibiliza uma consulta através de CEP. Porém, esta consulta é feita para pessoas e não para aplicações.

Um outro provável cenário seria uma aplicação de cadastro de conta corrente em um banco. Para abrir a conta, o banco exigiria que o cliente tivesse um Cadastro de Pessoa Física (CPF). As aplicações atuais conseguem apenas calcular o dígito verificador do CPF, que é uma informação de pouca utilidade para a instituição bancária. O ideal seria verificar se aquele CPF é realmente válido, quem é o portador, onde ele reside, se está em dia com a Receita Federal, entre outros. A Receita Federal poderia preparar um *web service* que respondesse não apenas uma, mas várias solicitações: se vai receber ou pagar imposto de renda, quais são os bens declarados, qual a principal fonte pagadora, sem contar toda a parte referente aos dados cadastrais. A aplicação bancária poderia se valer do *web service* para agilizar a abertura da conta, uma vez que os dados cadastrais seriam preenchidos pela resposta do *web service*. Uma opção mais radical permitiria que o banco armazenasse unicamente o CPF do cliente, sendo que o cadastro seria acessado através do banco de dados da Receita Federal.

Como último exemplo poderíamos pensar em uma aplicação de controle de estoque. Atualmente, quando um dos itens atinge o estoque mínimo, o usuário é informado para que providencie uma ordem de compra para aquele produto. O usuário entra em contato com os seus fornecedores e faz a tomada de preços para finalmente fechar o negócio. Em uma aplicação de controle de estoque usando *web services* as tarefas poderiam ser automatizadas. Quando algum dos itens ficar abaixo do estoque mínimo, diversas mensagens seriam enviadas para os fornecedores solicitando preço, disponibilidade, tempo de entrega, entre outros. As respostas seriam analisadas pela aplicação e o fechamento do negócio com um dos fornecedores seria feito baseado em critérios previamente estabelecidos, ressaltando que nenhuma intervenção humana seria necessária.

Como vimos por meio dos exemplos, o uso de *web services* vai permitir a interoperabilidade entre aplicações de maneira a agilizar transações. Na verdade, a consulta de

informações pela web está largamente difundida e é realizada facilmente por pessoas. O que se quer é permitir que aplicações possam realizar essas mesmas consultas sem o auxilio de seres humanos.

2.2 Tecnologias usadas pelos *web services*

Como foi dito anteriormente, um *web service* é uma aplicação que aceita solicitações de outros sistemas através da Internet [3], sempre baseado em padrões para facilitar a comunicação entre as máquinas. Esses padrões são baseados em XML (*eXtensible Markup Language*) que, como será visto posteriormente, vai permitir uma grande portabilidade de dados. A presente seção discute as tecnologias recentemente padronizadas que permitem o desenvolvimento de *web services*.

2.2.1. XML

XML é um padrão da indústria de informática que tem como maior objetivo o intercâmbio de dados. Ele permite que sistemas possam trocar informações de uma forma mais abrangente que arquivos texto, já que podemos dar semântica aos dados que estão sendo manipulados.

2.2.1.1. Documento XML

Da mesma forma que HTML (*Hyper Text Markup Language*), o XML se utilizam etiquetas[1] para representar os dados; porém existem várias diferenças entre esses dois padrões. Enquanto HTML especifica *como* os dados devem ser exibidos, XML se preocupa com o *significado* dos dados armazenados [4]. A seguir temos um exemplo de um documento XML que representa um curso e dois alunos com a sua média.

```
<curso>
  <aluno>
    <nome>Danielle</nome>
    <MGP>7.9</MGP>
```

[1] Tags

```
  </aluno>
  <aluno>
    <nome>Cinthia</nome>
    <MGP>6.5</MGP>
  </aluno>
</curso>
```

Pelo exemplo podemos notar que as etiquetas que foram usadas indicam a estrutura e o conteúdo armazenado. Uma outra grande diferença entre XML e HTML é que as etiquetas XML são extensíveis, permitindo que documentos XML possam representar fielmente os dados por ele armazenados. Já o HTML possui um conjunto de etiquetas pré-definidas e fixas, não permitindo adicionar novas.

Cada etiqueta do documento XML é chamado de *elemento*. Dessa forma, temos vários elementos no XML mostrado no exemplo: curso, aluno, nome e média. Como existem elementos que estão localizados dentro de outros elementos, podemos dizer que são sub-elementos. `Aluno` é um sub-elemento de `curso`, já `nome` e `média` são sub-elementos de `aluno`.

2.2.1.2. Linguagem para definição de esquemas

A característica do XML de permitir que etiquetas possam ser criadas de acordo com as necessidades do usuário faz com que exista uma preocupação com a estrutura que será utilizada pelo documento XML; isto é, o documento deve seguir um esquema pré-definido para que ele seja válido. A linguagem para a definição desse esquema é chamada de *Document Type Definition* (DTD). O DTD define as etiquetas que serão utilizadas num documento XML e a sua estrutura hierárquica, incluindo também a ordem que elas devem aparecer. Um DTD para o documento XML visto anteriormente poderia ser definido da seguinte forma:

```
<!ELEMENT curso (aluno)+>
<!ELEMENT aluno (nome, MGP)>
<!ELEMENT nome (#PCDATA)>
<!ELEMENT MGP (#PCDATA)>
```

O DTD anterior indica que `curso` será o elemento de mais alto nível. Mostra também que entre as etiquetas `<curso>` e `</curso>` deve aparecer pelo menos um `aluno`. A segunda

linha indica que cada elemento `aluno` deve conter os elementos `nome` e `MGP`. Tanto a terceira quanto a quarta linha indicam que os elementos `nome` e `MGP` irão armazenar dados.

2.2.1.3. Portabilidade do XML

O DTD, como o que foi mostrado na seção anterior, é que torna o documento XML portável [2]. Se uma aplicação recebe um documento XML baseado no DTD `aluno`, ela pode processar o documento de acordo com as regras especificadas no DTD. Caso um dos elementos não apareça, esteja fora de ordem ou seja encontrado um elemento que não está na estrutura do DTD, o documento torna-se inválido e a aplicação pode ficar ciente que um erro aconteceu.

Uma outra característica que torna o XML portável é o fato do documento não incluir instruções de como os dados serão mostrados. Manter os dados separados das instruções de formatação faz com que os dados possam ser exibidos em diferentes mídias [4]. O último ponto a ser destacado é que os documentos XML estão no formato texto, o que permite que seja lido tanto por pessoas quanto por máquinas, o que facilita a interpretação.

2.2.2. SOAP

Embora XML seja feito para a troca de dados, ele sozinho não é suficiente para o intercâmbio de informações através da web. É necessário ter um protocolo que possa enviar e receber mensagens contendo documentos XML: é neste ponto que entra o SOAP (*Simple Object Access Protocol*).

O propósito do SOAP é o de fazer chamadas a procedimentos remotos em cima do protocolo HTTP (ou outro protocolo padronizado da web), com a grande vantagem de não impor restrições de algum tipo de implementação para os pontos de acesso, como fazem atualmente RMI, CORBA e DCOM.

SOAP é um protocolo para troca de informações em um ambiente distribuído e é composto de três partes: envelope, header e body [8], conforme descrevemos a seguir:

2.2.2.1. SOAP Envelope

O elemento envelope está localizado no topo da hierarquia do protocolo SOAP. Ele é composto de dois elementos: `XMLnamespace` e `encodingStyle`. O `XMLnamespace` é um conjunto de nomes para tipos de elementos XML e nomes de atributos, isto é, um esquema XML [2]. Um dos esquemas mais usados nas mensagens SOAP encontra-se em `http://schemas.xmlsoap.org/soap/envelope`.

O atributo `encodingStyle` identifica o tipo de dados reconhecido pelas mensagens SOAP, além de especificar como os dados devem ser serializados para o transporte através da web. É possível indicar mais de um URI (*Uniform Resource Identifiers*) para identificar as regras de serialização, onde a ordem de colocação é da mais específica para a menos específica.

2.2.2.2. SOAP *Header*

O atributo `header` é opcional, porém, caso exista, deve ser colocado logo após o elemento envelope. Para uma mensagem SOAP sair da sua origem e chegar no seu destino, é provável que passe por diversos nós enquanto viaja pela web, sendo que cada nó pode receber e enviar mensagens SOAP. O `header` pode ser usado para que, ao passar pelos nós, seja feito um processamento com a mensagem. Alguns desses tipos de processamento podem ser autorização, gerenciamento, entre outros.

2.2.2.3. SOAP *Body*

O elemento `body` é a parte principal de mensagem SOAP. É onde estão armazenadas as informações necessárias para o que destinatário possa processar o pedido e retornar uma resposta. O exemplo abaixo mostra uma mensagem SOAP para a solicitação de informações para o site de pesquisa Google. Podemos notar que é no elemento `body` que são passados os parâmetros (key, maxResult, filter, safeSearch, entre outros) para que a aplicação possa responder a mensagem de forma adequada.

```
<SOAP-ENV:Envelope
          xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
          xmlns:xsi="http://www.w3.org/1999/XMLSchema-instance"
          xmlns:xsd="http://www.w3.org/1999/XMLSchema">
    <SOAP-ENV:Body>
       <ns1: doGoogleSearch xmlns:ns1="urn:GoogleSearch"
             SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
             <key xsi:type="xsd:string">CcnEHaJ3SGmaphcqn0ALb8GbJ0azC2fz</key>
             <q xsi:type="xsd:string">"web service"</q>
             <start xsi:type="xsd:int">0</start>
             <maxResults xsi:type="xsd:int">10</maxResults>
             <filter xsi:type="xsd:boolean">true</filter>
             <restrict xsi:type="xsd:string"></restrict>
             <safeSearch xsi:type="xsd:boolean">false</safeSearch>
             <lr xsi:type="xsd:string"></lr>
             <ie xsi:type="xsd:string">latin1</ie>
             <oe xsi:type="xsd:string">latin1</oe>
       </ns1: doGoogleSearch >
    </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

2.2.2.4. SOAP sobre HTTP

SOAP segue o modelo usado no HTTP de request/response. Dessa forma, o SOAP fornece os parâmetros de uma solicitação para o HTTP e recebe os parâmetros de uma resposta HTTP [8]. A única restrição é que aplicações HTTP devem usar o tipo `"text/xml"` de acordo com a RFC 2376 [12] quando SOAP inclui o elemento `body` na mensagem HTTP.

Seguem dois exemplos do uso de SOAP sobre HTTP. A primeira delas é uma solicitação usando HTTP POST. Já no segundo temos uma possível resposta HTTP. Vale ressaltar que SOAP segue a mesma semântica usada nos códigos de retorno do HTTP [8], que pode ser vista na resposta recebida.

```
POST /PrecoBom HTTP/1.1
Host: www.PrecoBom.br
Content-Type: text/xml; charset="utf-8"
Content-Length: nnnn
SOAPAction: "http://www.precobom:8080/webservice"

<SOAP-ENV:Envelope
    xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
    SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
    <SOAP-ENV:Body>
        <m:GetPreco xmlns:m="http://www.precobom:8080/webservice">
            <symbolo>Caneta</symbolo>
        </m:GetPreco>
    </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

```
HTTP/1.1 200 OK
Content-Type: text/xml; charset="utf-8"
Content-Length: nnnn

<SOAP-ENV:Envelope
    xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
    SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"/>
    <SOAP-ENV:Body>
        <m:GetPrecoResposta xmlns:m="http://www.precobom:8080/webservice">
            <Preco>1.5</Preco
       </m:GetPrecoResposta>
    </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

A primeira mensagem SOAP é uma solicitação para que o *web service* publicado na loja `precobom` retorne o valor de um determinado item. A segunda mensagem é a resposta do web service indicando o valor do item solicitado.

2.2.3. WSDL

WSDL (*Web Services Definition Language*) é usada para responder três perguntas sobre um *web service*: o que é o serviço, onde encontrá-lo e como chamá-lo. WSDL define os serviços como um conjunto de *endpoints*, isto é, pontos de acesso na rede [10]. Os elementos contidos em um documento WSDL estão listados a seguir:

- *types*: um container para definição de tipos;
- *message*: definição dos tipos dos dados que são passados nas operações;
- *operation*: descrição de uma ação disponibilizada pelo serviço;
- *port Type*: conjunto de operações suportadas nos *endpoints*;
- *Binding*: especificação do formato dos dados para um determinado portType;
- *Port*: é um *endpoint*, definido como a combinação de um *binding* com um endereço de rede;
- *Service*: um conjunto de *endpoints*.

Segue um exemplo de um documento WSDL, onde podem ser vistos todos os elementos, com exceção do `type`. O documento nos mostra que o serviço possui duas mensagens: `olaMundo` e `OlaMundoResponse`. O elemento `portType` indica quais são as

operações disponibilizadas pelo serviço, enquanto que o elemento `operation` especifica se as mensagens são de entrada ou saída. Podemos ver que no elemento `binding` são indicados os esquemas que devem ser usados para as operações que foram especificadas no `portType`, além do tipo e da maneira como devem ser transportados os dados. O elemento `port` indica onde o serviço pode ser localizado na rede.

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

<definitions name="WServiceService"
    targetNamespace="http://hello.org/wsdl"
        xmlns:tns="http://hello.org/wsdl"
        xmlns="http://schemas.xmlsoap.org/wsdl/"
        xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
        xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <types/>
  <message name="olaMundo">
    <part name="String_1" type="xsd:string"/>
  </message>
  <message name="olaMundoResponse">
    <part name="result" type="xsd:string"/>
  </message>
  <portType name="olaIF">
    <operation name="olaMundo">
      <input message="tns:olaMundo"/>
      <output message="tns:olaMundoResponse"/>
    </operation>
  </portType>
  <binding name="olaIFBinding" type="tns:olaIF">
    <operation name="olaMundo">
      <input>
        <soap:body encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
                   use="encoded" namespace="http://ola.org/wsdl"/>
      </input>
      <output>
        <soap:body encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
                   use="encoded" namespace="http://ola.org/wsdl"/>
      </output>
      <soap:operation soapAction=""/>
    </operation>
    <soap:binding
          transport="http://schemas.xmlsoap.org/soap/http"
          style="rpc"/>
  </binding>
  <service name="WService">
    <port name="olaIFPort" binding="tns:olaIFBinding">
      <soap:address location="http://localhost:8080/wservice"/>
    </port>
  </service>
</definitions>
```

2.2.4. UDDI (*Universal Description Discovery and Integration*)

Como foi dito anteriormente, os *web services* estão se tornando a base do comércio eletrônico. Empresas publicam seus *web services* que são chamados por outras empresas de maneira a executar uma transação. O gerenciamento de um ambiente que tenha algumas

poucas chamadas a *web services* pode ser uma tarefa simples [5]. Porém, com o crescimento e popularização de *web services*, as empresas terão muito mais opções de serviços que poderão ser utilizados nas suas aplicações. Como fazer para que novos serviços, necessários para as aplicações, sejam descobertos? Se for realizado manualmente, como podemos garantir que todos os *web services* existentes foram descobertos? O padrão UDDI tem como papel resolver o problema de descoberta de *web services* e ajudar na sua publicação.

A Figura 2 mostra onde está localizado o UDDI em uma pilha que contém outras tecnologias e padrões.

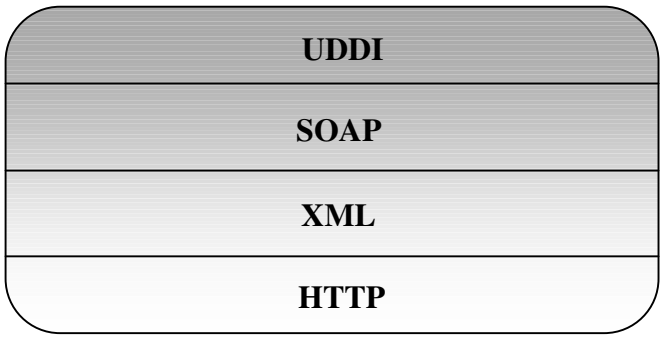

Figura 2 – Representação da pilha com UDDI e outros padrões / tecnologias

2.2.4.1. Estrutura de dados do UDDI

Como podemos ver na Figura 3, existem quatro grandes estruturas que armazenam os dados no UDDI: *businessEntity*, *businessService*, *bindingTemplate* e *tModel*. Veremos, a seguir, detalhes de cada uma delas.

A estrutura businessEntity representa a informação armazenada sobre a empresa ou entidade que publica o serviço. Segundo UDDI *Data Structure Reference* V1.0, a estrutura do *businessEntity* é constituída de:

```
<element name = "businessEntity">
    <type content = "elementOnly">
        <group order = "seq">
            <element ref = "discoveryURLs" minOccurs = "0" maxOccurs = "1" />
            <element ref = "name" />
            <element ref = "description" minOccurs = "0" maxOccurs = "*" />
            <element ref = "contacts" minOccurs = "0" maxOccurs = "1" />
            <element ref = "businessServices" minOccurs = "0" maxOccurs = "1" />
            <element ref = "identifiedBag" minOccurs = "0" maxOccurs = "1" />
            <element ref = "categoryBag" minOccurs = "0" maxOccurs = "1" />
        </group>
```

```
        <attribute name = "businessKey" minOccurs = "1" type = "string" />
        <attribute name = "operator" type = "string" />
        <attribute name = "authorizedName" type = "string" />
    </type>
</element>
```

Da estrutura acima podemos destacar `businessKey` como sendo a chave do registro UDDI. O campo `name` é onde deve ser armazenado o nome da entidade ou empresa que está publicando o registro. No campo `description` pode ser colocada uma breve descrição dos negócios da empresa.

A estrutura *businessService* contém informações sobre cada um dos serviços oferecidos pela empresa. Para cada *businessEntity* podemos ter vários *businessService*; este relacionamento é feito através do campo `businessKey`.

```
<element name = "businessService">
    <type content = "elementOnly">
        <group order = "seq">
            <element ref = "name" />
            <element ref = "description" minOccurs = "0" maxOccurs = "*" />
            <element ref = "contacts" minOccurs = "0" maxOccurs = "1" />
            <element ref = "bindingTemplates" />
            <element ref = "categoryBag" minOccurs = "0" maxOccurs = "1" />
        </group>
        <attribute name = "serviceKey" minOccurs = "1" type = "string" />
        <attribute name = "businessKey" type = "string" />
    </type>
</element>
```

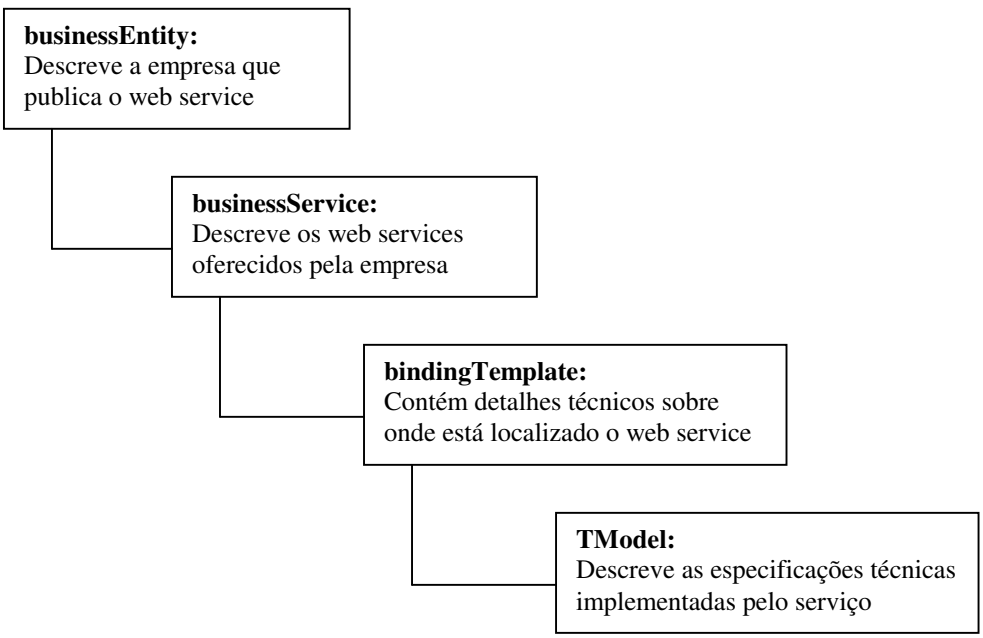

Figura 3 – Estruturas UDDI e seus relacionamentos

Na especificação de estrutura de *businessService* temos o campo `serviceKey`, que é a chave do registro. O campo `name` indica o nome do serviço e no campo `description` podemos descrever o serviço. Cada *businessService* pode conter um ou mais *bindingTemplates*, como pode ser visto na própria estrutura.

A estrutura *bindingTemplate* é composta por informações técnicas sobre *web services*. O campo `bindingKey` é a chave do registro e o campo `serviceKey` é o relacionamento com a estrutura *businessService*. Em `accessPoint` deve ser colocado o ponto de entrada para o *web service*, que normalmente é preenchido com uma URL.

```
<element name = "bindingTemplate">
    <type content = "elementOnly">
        <group order = "seq">
            <element ref = "description" minOccurs = "0" maxOccurs = "*" />
                <group order = "choice">
                    <element ref = "accessPoint" minOccurs = "0" maxOccurs = "1" />
                    <element ref = "hostingRedirector" minOccurs = "0" maxOccurs = "1" />
                </group>
                <element ref = "tModelInstanceDetails" />
        </group>
        <attribute name = "bindingKey" minOccurs = "1" type = "string" />
        <attribute name = "serviceKey" type = "string" />
</type>
</element>
```

A estrutura *tModel* tem como objetivo principal o armazenamento de metadados sobre o serviço [7]. É uma estrutura relativamente simples, tendo os campos `tModelKey`, `name`, `description` e `URL`, esta última é normalmente utilizada para encontrar mais informações sobre o serviço.

```
<element name = "tModel">
    <type content = "elementOnly">
        <group order = "seq">
            <element ref = "name" />
            <element ref = "description" minOccurs = "0" maxOccurs = "*" />
            <element ref = "overviewDoc" minOccurs = "0" maxOccurs = "1" />
            <element ref = "identifierBag" minOccurs = "0" maxOccurs = "1" />
            <element ref = "categoryBag" minOccurs = "0" maxOccurs = "1" />
        </group>
        <attribute name = "tModelKey" minOccurs = "1" type = "string" />
        <attribute name = "operator" type = "string" />
        <attribute name = "autorizedName" type = "string" />
    </type>
</element>
```

Capítulo 3

Desenvolvimento de web services

O Capítulo 2 mostrou as vantagens de uso de *web services* para melhorar a interoperabilidade dos sistemas e também quais são as tecnologias utilizadas para a sua criação. Neste capítulo veremos como desenvolver *web services* e quais os problemas que podem ser encontrados neste desenvolvimento.

3.1. Utilização das tecnologias sem ajuda de ferramentas

Como foi dito na Seção 2.2, as tecnologias envolvidas na criação de *web services* estão baseadas no uso de documentos XML. Embora seja possível manter arquivos XML manualmente, o mais indicado é contar com alguma ferramenta que possa lidar com leitura, gravação e navegação dos elementos dentro de um documento.

Um documento WSDL indica quais são as operações de um serviço e onde elas podem ser localizadas [10]. Supondo que um *web service* seja a porta de entrada para um sistema na linguagem Java, a geração do documento WSDL depende dos métodos, definidos nas classes, para poder especificar as operações que o serviço deve estar preparado para responder. Além dos métodos, devem ser indicados os esquemas que serão usados pelas operações, a maneira como os dados devem ser transportados, os *endpoints* e os pontos de acesso ao serviço. Realizar estas tarefas de forma manual requer tempo e acima de tudo precisão, já que um pequeno erro fará com que o serviço não funcione de acordo com o esperado.

Para enviar uma mensagem e receber a resposta de um *web service* usamos o protocolo SOAP. Para a montagem de um documento SOAP, temos os mesmos problemas encontrados para a criação de documentos WSDL. Além disso, temos o problema do envio e do recebimento de mensagens sobre HTTP, que se for feito sem a ajuda de ferramentas torna a vida do desenvolvedor bastante complicada. A mensagem recebida é um documento XML

que deve ser processado, com ajuda de *analisadores sintáticos*, para que a aplicação possa continuar seu fluxo.

A publicação do serviço é feita nos *service providers* utilizando registros UDDI. Os provedores de serviço nada mais são do que *web services* que estão prontos para receber mensagens SOAP e publicar os dados do serviço. Este é mais um exemplo onde podemos realizar todas as operações manualmente ou utilizar ferramentas que encapsulem a criação e o envio de mensagens SOAP contendo dados a serem publicados.

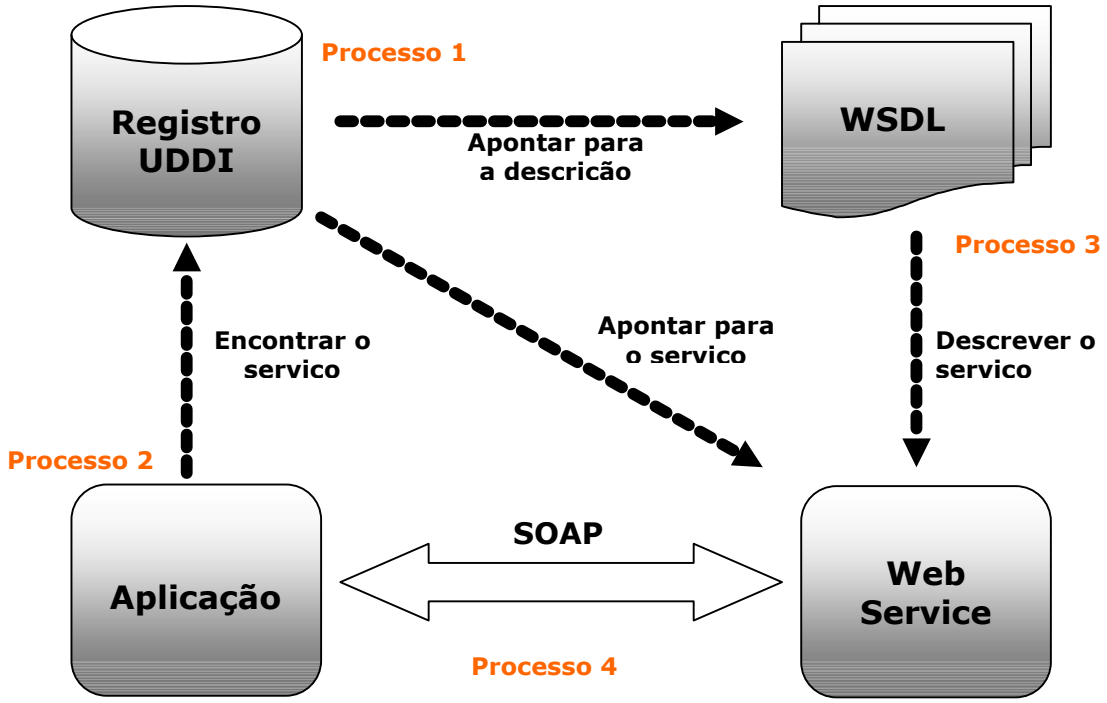

Figura 4 – Tecnologias de web services que precisam ser automatizadas

Na Figura 4 são mostrados os processos onde é necessário ter ferramentas que ajudem na criação de *web services*. O processo 1 está representando os serviços da entidade provedor de serviço que publicam *web services*; este processo vai ajudar o desenvolvedor a fazer com que o registro UDDI aponte para o serviço e para a descrição do serviço (o documento WSDL). A automatização do processo 2 vai permitir que uma aplicação possa encontrar registros UDDI que apontem para *web services* que atendam as necessidades do usuário. O processo 3 representa a geração de documentos WSDL, seja por meio de classes Java, *Enterprise Java Beans* (EJB) ou ainda outras linguagens de programação, este processo é um dos mais críticos da criação de *web services*. No processo 4 temos a troca de mensagens entre

a aplicação e o *web service* através do protocolo SOAP. Possuir uma ferramenta de auxilio permitirá que as aplicações possam criar e transmitir requisições de uma forma simples.

Pelo exposto, podemos notar que a criação de *web services* sem o uso de ferramentas adequadas fará com que a sua utilização seja apenas mais um modismo, ao invés de se tornar a grande solução para a comunicação entre aplicações que se propõe.

3.2. Ferramentas de desenvolvimento de web services

Atualmente existem algumas ferramentas para a criação de *web services* disponíveis no mercado. Entre as grandes empresas temos a Sun Microsystems e a IBM; porém, várias empresas de menor porte também lançaram produtos para o promissor futuro de desenvolvimento de *web services*. A seguir, algumas dessas ferramentas são brevemente descritas.

3.2.1. IBM Web Service Toolkit

O pacote Web Service Toolkit (WSTK) da IBM foi uma das primeiras ferramentas disponíveis no mercado com a tentativa de tornar mais fácil a criação de *web services*. Quando a IBM e a Microsoft definiram a especificação do padrão WSDL, a IBM prontamente colocou o WSTK para permitir o uso de documentos WSDL em registros UDDI [13]. O WSTK foi pioneiro em oferecer um registro UDDI privado que espelha as principais operações dos *service providers*; isto facilita o desenvolvedor, pois os testes com registros UDDI não precisam ser feitos com uma conexão para a Internet.

O WSTK oferece os seguintes componentes e funções:

- UDDI Java API (UDDI4J) - permite manutenções de registros UDDI, sejam as públicas (*service providers*) ou privados através do uso do banco de dados DB2;
- Java2WSDL – é um utilitário que permite a geração de documentos WSDL para código Java;
- Apache AXIS – infra-estrutura para o envio e recebimento de mensagens SOAP;
- Xerces – um parser XML para Java;

- WebSphere Application Server.

O WSTK usa uma interface gráfica, criada com Java Swing para permitir a execução em várias plataformas, de fácil uso e que torna simples a utilização dos recursos disponíveis. Porém, a ferramenta não tem uma boa integração entre os componentes e nem possui um gerenciamento dos *web services* desenvolvidos pelo usuário.

A tecnologia de *web services* está amadurecendo e a IBM aponta para vários novos requisitos solicitados pelo mercado: segurança, identificação, gerenciamento, notificação [13], entre outros. Estes itens devem estar presentes em uma boa ferramenta de desenvolvimento de *web services*.

3.2.2. Java Web Service Developer Pack (JWSDP)

É um pacote desenvolvido pela Sun Microsystems, que tem como objetivo reunir as tecnologias usadas pelos *web services*, abrangendo todo o ciclo de desenvolvimento. As tecnologias disponíveis no JWSDP são [11]:

- Java Servlets
- JavaServer Pages (JSP)
- JSP Standard Tag Library (JSTL)
- Java XML Pack:
  - Java API for XML Messaging (JAXM)
  - Java API for XML Processing (JAXP)
  - Java API for XML Registries (JAXR)
  - Java API for XML-based RPC (JAX-RPC)

Do pacote JWSDP podemos destacar a API Java para processamento das tecnologias de *web services*. Este conjunto de APIs facilita o trabalho do desenvolvedor, como: geração de documentos WSDL, chamadas de métodos em serviços disponíveis usando RPC, envio e recebimento de mensagens com o uso de SOAP e facilidade de publicação do serviço nos *service providers* existentes.

Um dos problemas encontrados no JWSDP é que as configurações do *web service* são feitas manualmente, por meio da criação de arquivos XML. Estes arquivos são processados pela ferramenta Ant, que permite a chamada de comandos do sistema operacional baseado em arquivos XML. Dessa forma, é possível compilar classes Java, copiar arquivos, criar arquivos JAR, entre outros, com a leitura de um arquivo XML. Além disso, devem ser editados vários outros documentos XML, como o config.xml e o web.xml, para que o *web service* possa ser publicado no servidor web Tomcat.

3.2.3. Outras ferramentas disponíveis

Nos últimos meses foram lançados vários pacotes para criação de *web services*. A seguir serão mostrados alguns desses produtos.

Web Application and Service Plataform (WASP) *Suite* é um conjunto de ferramentas criadas pela empresa Systinet para o desenvolvimento de *web services*. WASP *Developer*, WASP *Server for Java* e WASP UDDI são os três módulos do WASP *Suite*. WASP *Developer* é uma ferramenta que permite a geração de *web services* através de aplicações já existentes em Java ou C++. A grande vantagem é que o WASP *Developer* pode ser colocado como um *plug-in* em várias IDEs, como por exemplo Forte for Java e JBuilder.

WASP UDDI fornece um registro UDDI privado, da mesma forma que as ferramentas da Sun e da IBM, com a vantagem de usar a especificação UDDI V2. WASP *Server* é um ambiente de desenvolvimento que permite trabalhar com vários banco de dados, servidores de aplicação e servidores web. Fornece uma boa interoperabilidade com várias implementações SOAP, incluindo o WSTK da IBM [17].

CapeStudio é um ambiente integrado para a criação de *web services*. Desenvolvido pela empresa Cape Clear Software, o pacote tem como objetivo facilitar o desenvolvimento, dando recursos para projetar e executar a publicação dos *web services*. O CapuStudio foi projetado para ajudar os desenvolvedores nos seguintes aspectos [18]:

- projeto e criação de novos *web services*;
- conversão de sistemas em Java, EJB e CORBA para *web services*;

- criação de clientes em Java, Visual Basic e JSP;
- integração de documentos XML com *web services*;
- teste e publicação de *web services*.

O CapeStudio possui um editor de documentos WSDL, chamado de WSDL Assistant que pode gerar *session beans*, interface *home* e interface remota de modo a permitir criar *web services* baseados em EJB. Além disso, fornece suporte para tipos complexos, vetores, objetos e manipulação de erros.

Existem ainda outras ferramentas disponíveis e a cada dia surgem novidades a respeito, seja na criação de um novo pacote ou atualizações das versões existentes. A Borland Inc. lançou sua ferramenta, chamada de Borland Web Services Kit, como um *plug-in* para o JBuilder 7. O pacote reúne um conjunto de APIs, arquiteturas e padrões para Java. A BEA, empresa fabricante do servidor de aplicação WebLogic, também criou uma ferramenta para *web service*s que possui suporte para as principais tecnologias: UDDI, SOAP e WSDL, totalmente integrado com o próprio WebLogic.

3.3. Problemas existentes nas ferramentas atuais

Um ano em tecnologia de informática não tem equivalência para outras áreas de conhecimento se a análise for feita em termos de novos paradigmas ou aprimoramento dos existentes. Com *web services* não poderia ser diferente, o que temos hoje é um cenário bem diferente do que tínhamos no início de 2002, quando foi feita nossa proposta de criação de uma ferramenta de gerenciamento. As ferramentas Java Web Service Toolkit (WSTK) da IBM e Java Web Service Develop Pack (JWSDP) da Sun Microsystems sofreram grandes mudanças e melhoramentos nas suas últimas versões, porém ainda não possuem um gerenciamento dos *web services* criados.

Embora as ferramentas tenham colocado à disposição do usuário vários pontos-chave: geração de documentos WSDL, envio e recebimento de mensagens SOAP e API ou interfaces para publicação de web services usando UDDI, existem algumas arestas inacabadas, que tornam a criação de *web services* uma tarefa senão complicada, ao menos manualmente trabalhosa.

Em um artigo publicado no final de 2001 [15], chamado *Creating a Web Service in 30 Minutes*, percebemos que o autor tenta mostrar que criar um *web service* é simples e pode ser feito rapidamente. Na verdade, criar um *web service* em apenas meia hora não é tão verdadeiro, precisa que o usuário conheça as tecnologias envolvidas e tenha as ferramentas certas. Podemos dizer que: (a) os trinta minutos transformam-se em alguns dias; (b) as operações são realizadas mecanicamente, sem entender o verdadeiro funcionamento das tarefas envolvidas.

Pelo que podemos ver, as ferramentas atuais fornecem a infra-estrutura básica para a criação de *web services*. Entretanto, elas deixam a desejar em pelo menos três pontos:

a) gerenciamento dos serviços publicados;
b) edição de arquivos para realizar as operações;
c) pouca integração entre as tecnologias.

Os problemas comentados merecem atenção especial por parte dos desenvolvedores. Uma ferramenta que ajude a resolver os pontos citados terá grandes chances de se impor sobre as outras.

3.4. Requisitos de uma ferramenta para gerenciamento de *web services*

As versões dos pacotes que foram analisados mais profundamente, o WSTK e o JWSDP, não estão providas de ferramentas de gerenciamento para os *web services* que foram publicados, seja do lado cliente ou do lado servidor. Dessa forma, os processos que foram automatizados por essas ferramentas ajudam somente no desenvolvimento.

Depois que um *web service* está funcionando, isto é, foi criado e publicado passando a receber e responder requisições, ele torna-se um sistema como outro qualquer que necessitará eventualmente de manutenção, seja para a correção de problemas ou para a adição de novos recursos.

Sendo um pouco mais específico na criação de *web services* usando a linguagem Java, qualquer manutenção deverá mudar um arquivo com extensão .WAR que é publicado no servidor web para tornar o *web service* disponível na Internet. Este arquivo contém um documento WSDL, as classes necessárias e outros arquivos para a publicação do serviço. Como foi dito na Seção 2.2.3, o documento WDSL descreve as interfaces do *web service* e indica onde o serviço pode ser encontrado. Realizar uma mudança em qualquer um desses pontos vai trazer problemas para os usuários do *web service* se não for tomado o devido cuidado.

O que queremos é que tanto a criação quanto a manutenção de *web services* sejam feitos de uma maneira organizada. Mudanças realizadas nas interfaces ou nos *endpoints* devem ser refletidos no registro UDDI do serviço. Modificações feitas nas regras de negócio, implementadas nas classes Java, devem ser facilmente incorporadas ao *web service* sem que em nada afete a sua estrutura.

O que foi citado no parágrafo anterior só torna-se possível quando, além das APIs e interfaces, é disponibilizada uma ferramenta para gerenciamento de *web services*. Esta ferramenta deve armazenar os dados referentes a cada serviço, como por exemplo:

- Quais são as classes utilizadas;
- Onde o serviço foi publicado (*service provider*);
- Onde está localizado o documento WSDL que descreve o serviço;
- Qual é o ponto de acesso ao serviço na rede (*endpoint*);
- Quais são os parâmetros das mensagens SOAP para cada um dos serviços chamados.

De posse dessas informações é possível fazer alterações nos *web services* existentes sem ter que se preocupar de que forma eles foram colocados em funcionamento. Conhecendo os detalhes do *web service*, as chances de ocorrer um erro quando uma atualização for realizada serão menores. Uma classe Java faltando, o nome do *endpoint* diferente ou ainda a publicação do arquivo WSDL em outro diretório do que foi definido, vai fazer com que os usuários do serviço tenham problemas para se comunicar com o *web service*.

Vimos que é necessário ter uma ferramenta que além de ajudar no desenvolvimento de *web services* faça um gerenciamento dos artefatos necessários para todas as etapas do ciclo de

vida de um serviço. Fazendo uma analogia, podemos comparar o que se deseja com o que fazem as interfaces de desenvolvimento integrado (IDEs), JBuilder ou Delphi por exemplo, com os projetos mantidos por elas. Cada projeto contém as configurações que permitem o gerenciamento dos programas e fazem com que o produto final, que pode ser um arquivo executável ou arquivos JAR, possa ser gerado com muita simplicidade.

Dessa forma, a ferramenta deve atender os seguintes requisitos:

a) Armazenar dados do *service provider* para permitir uma rápida publicação do serviço;
b) Gerenciar os *web services* desenvolvidos para permitir uma fácil manutenção;
c) Enviar mensagens SOAP sem a necessidade de criação de programas específicos;
d) Gerenciar serviços cliente que fazem acesso a *web services* publicados;
e) Esconder do desenvolvedor todos os detalhes técnicos da criação e publicação de *web services*.

Capítulo 4

O pacote Java Web Service Developer Pack

O pacote da Sun Microsystems para desenvolvimento de *web services* foi chamado de *Java Web Service Developer Pack.* Ele concentra as principais tecnologias para processamento de documentos XML, que servem como base para a criação de *web services.* Fornece um conjunto de APIs na linguagem de programação Java e diversas ferramentas para a geração dos artefatos necessários tanto para a geração quanto para a publicação de serviços. Neste capítulo veremos tanto como utilizar estas APIs quanto o funcionamento das ferramentas do pacote.

4.1 Java API for XML Processing (JAXP)

A API JAXP permite processar documentos XML dentro de aplicações escritas na linguagem Java. JAXP usa os padrões SAX (*Simple API for XML Parsing*) e DOM (*Document Object Model*) para o processamento do documento, dando ao usuário a possibilidade de escolha entre a representação do XML como um objeto ou a manipulação dos dados de forma seqüencial.

Quando um documento XML é processado com a API SAX, temos uma leitura ou gravação seqüencial dos elementos. Por outro lado, a API DOM fornece uma estrutura em árvore para a representação dos elementos. A construção do DOM precisa ler todo o documento XML e montar a hierarquia na memória, isto faz com que o tempo e o uso da memória para o DOM sejam maiores do que para o SAX.

O principal pacote da API JAXP é o `javax.xml.parser`. Dentro do pacote podemos encontrar duas implementações que são representadas pelas classes: `SAXParserFactory` e `DocumentBuilderFactory`. Esta última é a que permite lidar com o documento XML como um objeto no padrão DOM.

4.1.1 Simple API for XML (SAX)

O esquema representativo de manipulação de documentos XML usando SAX pode ser visto na Figura 5. O processo é iniciado com a criação da instância do objeto `SAXParser` por meio da classe `SAXParserFactory`. Para fazer a leitura é chamado o método `parser()`, que por sua vez pode chamar um ou mais métodos definidos na aplicação. Estes métodos são definidos pelas seguintes interfaces: `ContentHandler`, `ErrorHandler`, `DTDHandler`, e `EntityResolver`.

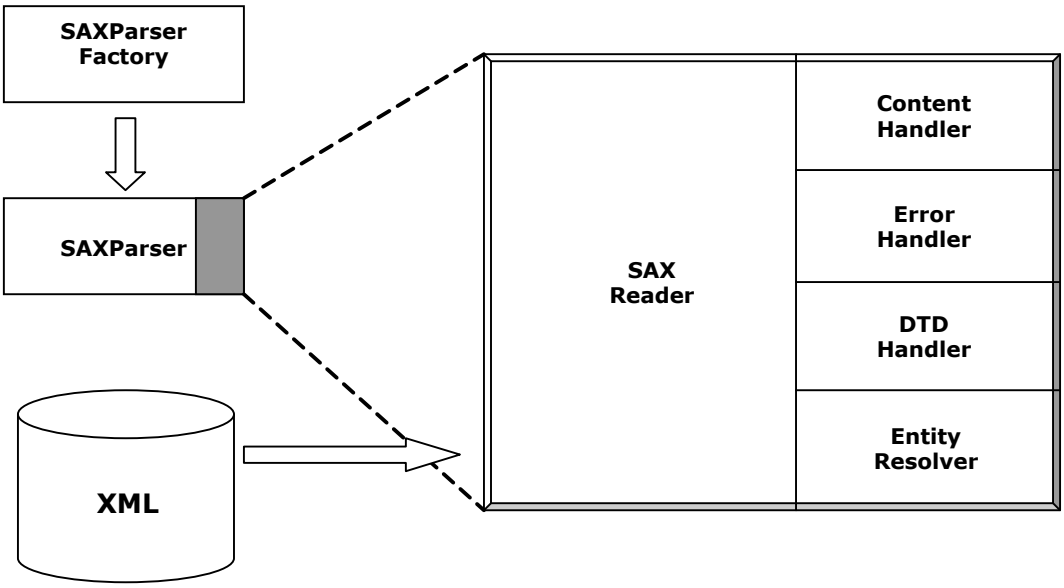

Figura 5 – Esquema de manipulação de documentos usando SAX

A interface `ContentHandler` define os métodos `startDocument`, `endDocument`, `startElement` e `endDocument` que são chamados quando é encontrada uma etiqueta XML. A interface também define os métodos `characters` e `processingInstruction`, para quando for encontrado um elemento texto no documento XML.

A interface `ErrorHandler` define os métodos `error`, `fatalError` e `warning`, que são chamados quando um erro é encontrado. O método `resolveEntity`, da interface `EntityResolver`, é chamado quando a localização dos dados deve ser feita por meio de uma URI.

A Tabela 1 mostra os pacotes existentes na API SAX.

Pacote	Descrição
`org.xml.sax`	É a definição da interface SAX
`org.xml.sax.ext`	É a definição estendida da interface SAX, usada no processamento de documentos XML que usam DTDs
`org.xml.sax.helpers`	É composta por classes de ajuda para a interface SAX, como por exemplo a definição de métodos nulos para as interfaces do `SAXReader`.
`javax.xml.parsers`	Definição da classe `SAXParserFactory` que retorna o `SAXReader`, além da definição de classes para manipulação de erros.

Tabela 1 – Pacotes SAX

4.1.2 A API Document Object Model (DOM)

Na Figura 6 podemos ver o esquema para manipulação de documentos XML usando DOM.

É por meio da classe `javax.xml.parsers.DocumentBuilderFactory` que conseguimos uma instância para DocumentBuilder que é o responsável pela criação do documento em forma de hierarquia, seguindo a especificação DOM.

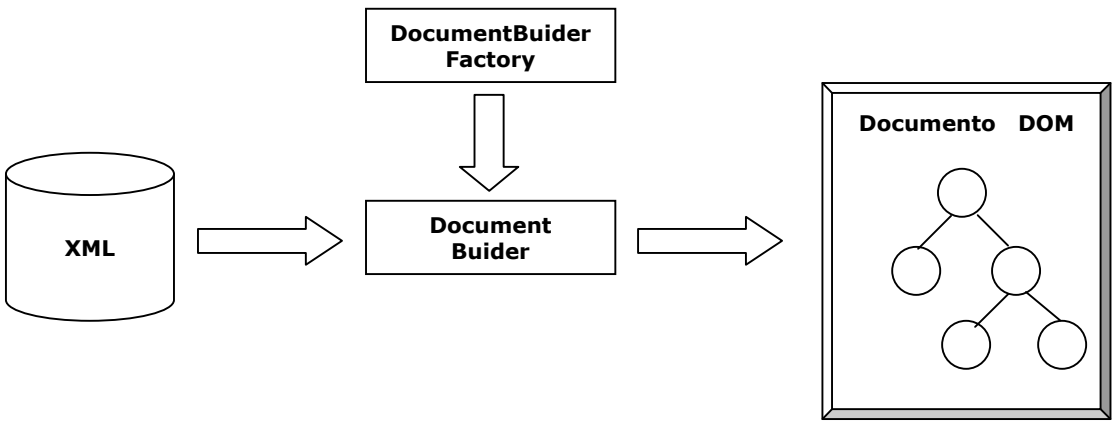

Figura 6 – Esquema de manipulação de documentos XML com DOM

A seguir é mostrada uma tabela contendo os pacotes do DOM.

Pacote	Descrição
`org.w3c.dom`	É a definição da interface DOM para os documentos XML, definido pela W3C.
`javax.xml.parser`	É a definição das classes `DocumentBuilderFactory` e `DocumentBuilder`, que retornam um objeto que implementa a interface DOM da W3C. Este pacote também define a classe `ParserConfigurationException` para a manipulação de erros.

Tabela 2 – Pacotes DOM

4.2 API JAXR

Os registros de negócios estão se tornando, cada vez mais, um componente importante na tecnologia de *web services*. Eles permitem que empresas possam colaborar entre si de uma maneira dinâmica [4]. *Web services* publicados em registros UDDI permitem que aplicações possam descobrir os seus pontos de acesso na Internet e possam fazer chamadas as suas interfaces.

Java API for XML Registries (JAXR) fornece uma maneira de acessar registros de negócios pela Internet. Estes registros são descritos como "páginas amarelas eletrônicas" pois contém uma lista de empresas e dos produtos que elas oferecem. JAXR permite que os desenvolvedores possam acessar esses registros UDDI com o uso da linguagem Java.

4.2.1 A arquitetura da API JAXR

A arquitetura da JAXR pode ser dividida em duas partes:

- JAXR cliente, que permite que usuários possam acessar os registros armazenados;
- JAXR *provider*, que implementa diversas interfaces definidas pela especificação UDDI, com o objetivo de permitir que os clientes possam acessar os registros.

São dois os principais pacotes implementados pela JAXR *provider*: `javax.xml.registry` e `javax.xml.registry.infomodel`.

O pacote `javax.xml.registry.infomodel` é quem define quais serão os tipos dos registros armazenados e como será o seu relacionamento. A interface básica é `RegistryObject`, mas também estão presentes `Organization`, `Service` e `ServiceBinding`, que são as principais definições dos registros UDDI.

O pacote `javax.xml.registry` implementa as seguintes interfaces:

- `Connection`, permite a conexão de um cliente com o registry provider;
- `RegistryService`, permite ao cliente obter e usar as interfaces para acessar os dados armazenados;
- `BusinessQueryManager`, permite ao cliente procurar um registro;
- `BusinessLifeCycleManager`, permite ao cliente modificar as informações do registro.

Uma grande parte dos métodos da API JAXR usa a classe `Collection` como valor de retorno pois ele permite trazer vários registros de uma só vez. A exceção `JAXRException` é lançada pela maioria dos métodos sempre que um erro ocorre.

4.2.2 Gerenciamento dos registros UDDI

A seguir serão mostrados os passos para a criação, manutenção e consulta de registros UDDI usando um provedor de serviços. Para alguns exemplos é necessário fazer um cadastro para obter os dados do usuário e senha. Dois sites que podem ser usados para teste e que implementam a definição UDDI são: `http://uddi.rte.microsoft.com` e `http://www-3.ibm.com/services/uddi/v2beta/protect/registry.html`.

4.2.2.1 Criando uma conexão

Inicialmente deve ser criada uma instância da classe `ConnectionFactory`. Depois devemos especificar as URL, que serão usadas para criar a conexão com o provedor de serviços como um conjunto de propriedades. Essas URLs devem ser atribuídas à instância `ConnectionFactory` para finalmente criar a conexão. Quando a conexão for feita com o uso de um firewall, devemos especificar nas propriedades o host e a porta.

```
...
// criação da instância de ConnectionFactory
ConnectionFactory connFactory = ConnectionFactory.newInstance();

// definição das propriedades
Properties props = new Properties();
props.setProperty("javax.xml.registry.queryManagerURL",
                  "http://www-3.ibm.com/services/uddi/v2beta/inquiryapi");
props.setProperty("javax.xml.registry.lifeCycleManagerURL",
                  "http://www-3.ibm.com/services/uddi/v2beta/protect/publishapi");
props.setProperty("javax.xml.registry.factoryClass",
                  "com.sum.xml.registry.uddi.ConnectionFactoryImpl");

// definição quando o cliente está atras de um firewall
props.setProperty("javax.xml.registry.http.proxyHost","host.dominio");
props.setProperty("javax.xml.registry.http.proxyPort","8080");

//atribuição das propriedades
connFactory.setProperties(props);

// criação da conexão
Connection connection = connFactory.createConnection();
...
```

4.2.2.2 Obtendo o objeto *RegistryService*

Depois da criação da conexão o cliente pode obter o objeto `RegistryService` que dará acesso aos objetos `BusinessQueryManager` e `BusinessLifeCycleManager` para permitir a consulta e a manutenção dos registros, respectivamente.

```
...
RegistryService rs = connection.getRegistryService();

// objeto para consulta no registro
BusinessQueryManager bqm = rs.getBusinessQueryManager();

// objeto para manutenção dos dados do registro
BusinessLifeCycleManager blcm = rs.getBusinessLifeCycleManager();
...
```

4.2.2.3 Consultando registros

A interface `BusinessQueryManager` permite a consulta de registros baseados em métodos que retornam um objeto `BulkResponse`, que é na verdade uma coleção de objetos. Os métodos mais utilizados são:

- `findOrganization`, retorna um conjunto de organizações que preenchem um determinado critério, como por exemplo o nome;
- `findServices`, retorna um conjunto de serviços oferecidos pela organização;
- `findServiceBinding`, retorna as informação de acesso para um determinado serviço.

A seguir temos o exemplo de uma consulta de organizações usando como critério de pesquisa o nome:

```
...
// definir os critérios de pesquisa
Collection findQualifiers = new ArrayList();
FindQualifiers.add(FindQualifier.SORT_BY_NAME_DESC);
Collection namePatterns = new ArrayList();

// procurar empresas que tenham a frase "web service" em qualquer posição
NamePatterns.add("%web service%");

// aplicar os critérios ao BusinessQueryManager
BulkResponse response = bqm.findOrganization(namePatterns, null, null, null, null);

// a resposta é uma coleção de objetos
Collection orgs = response.getCollection();
...
```

Depois de encontrar as organizações é possível obter os serviços publicados e a sua forma de acesso. A seguir temos o trecho de um programa que permite a consulta de `service` e de `serviceBinding`.

```
...
// obter os registros das organizações
Iterator orgIter = orgs.iterator();
while ( orgIter.hasNext() ) {
  Organization org = (Organization) orgIter.next();
  Collection services = org.getServices;

  // obter os serviços de cada organização
  Iterator svcIter = services.iterator();
  while ( svc.hasNext() ) {
    Service svc = (Service) svcIter.next();
    Collection serviceBinding = svc.getServiceBinding();

    // obter os serviceBinding de cada serviço
    Iterator sbIter = serviceBinding.iterator();
    while ( sbIter.hasNext() ) {
      ServiceBinding sb = (ServiceBinding) sbIter.next();
    }
```

```
  }
}
...
```

4.2.2.4 Manutenção de registros

Para que um usuário possa criar, editar ou remover registros, ele deve possuir uma autorização que é obtida mediante um cadastro no provedor de serviços. No ato do cadastramento, o usuário informará o nome do usuário (*username*) e a senha (*password*) que permitirá ter acesso a manutenção dos registros. Estes dois parâmetros devem ser enviados para o provedor de serviços para conseguir uma autorização. Segue o trecho de um programa que tenta abrir uma conexão com autorização:

```
...
// definição dos dados da autorização
String username = "nome_usuario";
String password = "senha_usuario";

// atribuindo os dados da autenticação
PasswordAuthentication passwdAuth = new PasswordAuthentication( username,
                                                                password.toCharArray() );

// abertura da conexão autenticada
Set creds = new HashSet();
creds.add(passwdAuth);
connection.setCredentials(creds);
...
```

A estrutura de dados mais complexa do registro UDDI é a organização. Ela é geralmente composta de: nome (`name`), descrição (`description`), chave (`key`), pessoa de contato (`PrimaryContact`), classificação (`classification`), serviços (`service`) e pontos de acesso (`serviceBindings`). A seguir temos um exemplo de como adicionar dados a uma estrutura do tipo `Organization`.

```
...
// Adicionando o nome e a descrição
Organization org = blcm.createOrganization("Andrés Web Service");
InternationalString s = blcm.createInternationalString("criação de web services");
org.setDescription(s);

// Adicionando o contato
User primaryContact = blcm.createUser();
PersonName pName = blcm.createPersonName("Andrés Menéndez");
primaryContact.setPersonName(pName);

TelephoneNumber tNum = blcm.createTelephoneNumber();
tNum.setNumber("(79) 234-5678");
Collection phoneNums = new ArrayList();
phoneNums.add(tNum);
primaryContact.setTelephoneNumbers(phoneNums);

EmailAddress emailAddress = blcm.createEmailAddress("andres@unit.br");
```

```
Collection emailAddresses = new ArrayList();
emailAddresses.add(emailAddress);
primaryContact.setEmailAddresses(emailAddresses);

// Adicionando o contato à organização
org.setPrimaryContact(primaryContact);
...
```

As organizações normalmente pertencem a uma ou mais classificações dentro de uma ou mais esquemas de classificação (taxonomia) [11]. O exemplo abaixo adiciona uma classificação usando a taxonomia NAICS (North American Industry Classification System).

```
...
// Adicionar o esquema de classificação NAICS
ClassificationScheme cScheme = bqm.findClassificationSchemeByName("ntis-gov:naics");

// criar a classificação
Classification classification = (Classification)
         blcm.createClassification(cScheme, "Desenvolvimento de Software", "123456");
Collection classifications = new ArrayList();
classifications.add(classification);

//adicionar a classificação
org.addClassifications(classifications);
...
```

Da mesma forma que o objeto `Organization`, o objeto `Service` possui um nome e uma descrição, além disso possui o campo `key` que é gerado automaticamente quando o serviço é inserido. Cada serviço pode ter um `serviceBinding` para informar como ele deve ser acessado, estas informações são compostas de uma descrição, uma URI e um link para maiores informações sobre o serviço. O seguinte exemplo mostra como adicionar um `service` e um `serviceBinding` para uma organização.

```
...
// Criar o serviço
Collection services = new ArrayList();
Service service = blcm.createService("Reserva");
InternationalString is = blcm.createInternationalString("Reserva de Hotel");
service.setDescription(is);

// Criar o serviceBindings
Collection serviceBindings = new ArrayList();
ServiceBinding binding = blcm.createServiceBinding();
is = blcm.createInternationalString("Descrição da reserva do hotel");
binding.setDescription(is);
binding.setAccessURI("http://hotel.com.br:8080/webservice");
serviceBindings.add(binding);

// Adicionar o serviceBinding
service.addServiceBindings(serviceBindings);

// adicionar o serviço
services.add(service);
org.addServices(services);
...
```

Para poder adicionar uma organização no provedor de serviços devemos enviar uma solicitação usando o método `saveOrganizations`. A resposta é a chave do registro que foi inserido ou uma exceção indicando o tipo do erro. No exemplo a seguir podemos ver como publicar os dados de uma organização.

```
...
// Publicar a organização
Collection orgs = new ArrayList();
orgs.add(org);
BulkResponse response = blcm.saveOrganization(orgs);

// obter as possíveis exceções
Collection exceptions = response.getException();

// verificar se ocorreu alguma exceção
if ( exceptions == null ) {

  // obter a chave do registro
  Collections keys = response.getCollection();
  Iterator keyIter = keys.iterator();
  if ( keyIter.hasNext() ) {
    javax.xml.registry.infomodel.Key orgKey =
               (javax.xml.registry.infomodel.Key) keyIter.next();

    // obter a chave da organização
    Org.setKey(orgKey);
  }
}
...
```

Para remover os dados do registro é necessário passar como parâmetro a chave que foi retornada pelo método `saveOrganization`, para um dos métodos de remoção do `BusinessLifeCycleManager`: `deleteOrganization`, `deleteService` e `deleteServiceBinding`. No seguinte trecho de programa podemos ver a remoção de todos os dados de uma organização.

```
...
// obter a chave da organização
Collection keys = new ArrayList();
keys.add(key);

//remover a organização
BulkResponse response = blcm.deleteOrganization(keys);

// obter as possíveis exceções
Collection exceptions = response.getException();

// verificar se ocorreu alguma exceção
if ( exceptions == null ) {
  ...
}
...
```

4.3 API JAXM

Java API for XML Messaging (JAXM) fornece um meio padronizado de envio de documentos XML pela Internet usando a plataforma Java. A API está baseada nas especificações SOAP 1.1 e SOAP *with Attachments* [4].

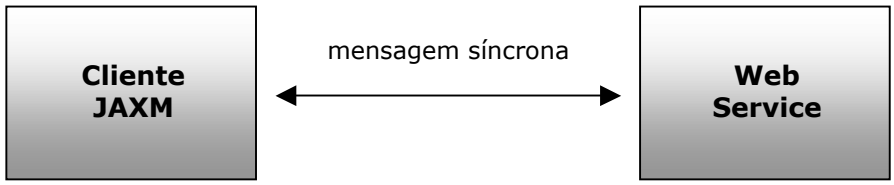

Figura 7 – Envio de uma mensagem *standalone*

São duas as maneiras de enviar mensagens através da JAXM: usando *messaging provider* e envio de mensagens ponto a ponto, chamadas de *standalone message*. Mensagens sem o uso do *messaging provider* obriga ao cliente JAXM a enviar o documento XML diretamente para o web service que implementa solicitações do tipo *request/response*. Este tipo de solicitação é uma mensagem síncrona, isto é, o envio e o recebimento da mensagem são realizados na mesma operação.

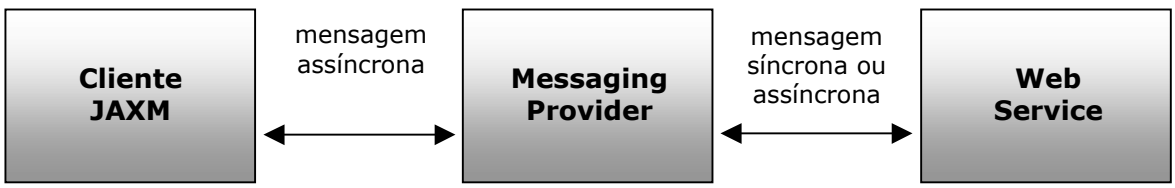

Figura 8 – Envio de uma mensagem usando *messaging provider*

O uso do *messaging provider* no envio de mensagens permite mandar mensagens assíncronas. A mensagem é enviada para o *messaging provider* que executa o encaminhamento para o destino, encarregando-se do reenvio caso aconteça algum problema na entrega. É possível que a mensagem seja enviada para *messaging providers* intermediários antes de chegar ao seu destino final, este roteamento é também uma tarefa do *messaging provider*.

4.3.1 Tipos de mensagens

A API JAXM segue o padrão SOAP que especifica o formato das mensagens que podem ser enviadas. Existem dois tipos de mensagem SOAP: mensagens com e sem anexos.

JAXM fornece um conjunto de classes que representam cada um dos elementos da mensagem SOAP. Dessa forma, temos: `SOAPMessage`, `SOAPPart`, `SOAPEnvelope`, `SOAPHeader` e `SOAPBody`. As mensagens sem anexos seguem a estrutura hierárquica mostrada a seguir:

```
I – SOAP Message
  A – SOAP Part
    1 – SOAP Envelope
      a – SOAP header (opcional)
      b – SOAP body
```

Quando um objeto `SOAPMessage` é criado, automaticamente são gerados todos os elementos necessários na mensagem SOAP. O objeto `SOAPMessage` contém um objeto `SOAPPart` que por sua vez possui um objeto `SOAPEnvelope`. Da mesma forma, o objeto `SOAPEnvelope` contém os objetos `SOAPHeader` e `SOAPBody`.

Quando desejamos enviar em uma mensagem SOAP conteúdo que não seja um documento XML, uma imagem ou um arquivo texto por exemplo, deve ser feito através de anexos, chamados de *attachment part*. A estrutura de mensagens contendo partes anexadas segue a seguinte hierarquia:

```
I – SOAP Message
  A – SOAP Part
    1 – SOAP Envelope
      a – SOAP header (opcional)
      b – SOAP body
  B – Attachment part
  C - Attachment part
```

Como pode ser visto na hierarquia, é permitido enviar um ou mais anexos na mensagem SOAP. JAXM fornece a classe `AttachmentPart` que servirá para representar o anexo da mensagem que deseja ser enviada.

4.3.2 Conexões

Todas as mensagens SOAP enviadas com JAXM devem usar uma conexão. É ela que se encarrega de executar a entrega da mensagem para um *web service* específico ou para um *messaging provider*. JAXM fornece duas classes distintas para representar cada tipo de conexão: `SOAPConnection` e `ProviderConnection`.

O primeiro passo para conseguir uma conexão *standalone* é obter a instância de um objeto `SOAPConnectionFactory`. Com essa instância é que pode ser criada a conexão. No exemplo abaixo está o trecho de um programa que obtém uma conexão:

```
...
// obter uma conexão standalone
SOAPConnectionFactory conFactory = SOAPConnectionFactory.newInstance();
SOAPConnection con = conFactory.createConnection();
...
```

Para conseguir uma conexão usando um *messaging provider* devemos instanciar um objeto `ProviderConnectionFactory`. Porém, diferentemente das conexões *standalone*, para abrir uma conexão com um *messaging provider* é necessário conhecer o seu nome, que deve ser baseado na Java Name and Directory Interface (JNDI). O seguinte trecho de código mostra como obter uma conexão para um *messaging provider* fictício, de nome "WService".

```
...
// obter uma conexão messaging provider
Context ctx = new InitialContext();
ProviderConnectionFactory pcFactory = (ProviderConnectionFactory) ctx.lookup("WService");

ProviderConnection con = pcFactory.createConnection();
...
```

4.3.3 Criação de mensagens

Se a mensagem a ser enviada é do tipo *standalone*, a criação é através da classe `messageFactory`. A instância dessa classe permite acessar cada uma das partes da mensagem; como a parte do *header* é opcional é possível remove-lo chamando o método `detachNode`.

```
...
// criação da mensagem
MessageFactory factory = MessageFactory.newInstance();
SOAPMessage message = factory.createMessage();
```

```
// obter cada um dos elementos da mensagem
SOAPPart soapPart = message.getSOAPPart();
SOAPEnvelope envelope = soapPart.getEnvelope();
SOAPHeader header = envelope.getHeader();
SOAPBody body = envelope.getBody();

// remover o header da mensagem
header.detachNode();
...
```

A criação de mensagem para *messaging provider* é um pouco mais complicada. Para criar a mensagem é necessário especificar o perfil (*profile*) que se deseja usar. Cada *message provider* indica quais os perfis que ele disponibilizada, isto é feito através dos métodos `getMetadata` e `getSupportedProfiles`.

```
...
// obter os profiles a partir dos metadados
ProviderMetaData metaData = pcCon.getMetaData();
String[] supportedProfiles = metaData.getSupportedProfiles();

// procurar o profile desejado
String profile = null;
for (int i=0; i < supportedProfiles.length; i++) {
  if (supportedProfiles[i].equals("ebxml")) {
    profile = supportedProfiles[i];
    break;
  }
}

// criação da mensagem
MessageFactory factory = pcCon.createMessageFactory(profile);
...
```

4.3.4 Adicionando conteúdo à mensagem

Para adicionar conteúdo à mensagem é necessário criar um objeto do tipo `SOAPBodyElement`. Uma instância do objeto `Name` deve ser passada como parâmetro no método `addBodyElement`. A seguir é mostrado um trecho de código que adiciona um objeto `SOAPBodyElement` na mensagem:

```
...
// adicionando o nome e o elemento na parte body da mensagem
SOAPBody body = envelope.getBody();
Name bodyName = envelope.createName("RetornarPreco", "m", "http://www.BoaCompra.com.br");
SOAPBodyElement element = body.addBodyElement(bodyName);

// criação de um elemento filho
Name name = envelope.createName("item");
SOAPElement item = element.addChildElement(name);
symbol.addTextNode("Mouse");
...
```

A mensagem SOAP criada usando o código Java do trecho anterior deverá ter o seguinte aspecto:

```
<SOAP-ENV:Envelope
   xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
     <SOAP-ENV:Body>
        <m:RetornarPreco xmlns:m= "http://www.BoaCompra.com.br">
           <item>Mouse</item>
        </m:RetornarPreco>
     </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

4.3.5 Enviando a mensagem

Para enviar uma mensagem *standalone* é necessário usar o objeto `SOAPConnection`, chamando o método `call`. Este método recebe dois parâmetros: a mensagem e o endereço de destino. O seguinte trecho de código mostra o envio de uma mensagem *standalone*.

```
...
// enviando a mensagem
URLEndpoint endpoint = new URLEndpoint( "http://www.BoaCompra.com.br/consultaPreco");
SOAPMessage response = con.call(message, endpoint);

// fechando a conexão
con.close();
...
```

O envio de uma mensagem para um *messaging provider* é mais simples, já que o destino foi especificado na criação da mesma. A única tarefa que deve realizada é invocar o método `send` do objeto `ProviderConnection`.

```
...
// enviando a mensagem
con.send(message);

// fechando a conexão
con.close();
...
```

4.4 API JAX-RPC

O mecanismo de RPC (*Remote Procedure Call*) possibilita que um cliente execute chamadas a procedimentos localizados em servidores remotos. Um exemplo típico é o modelo cliente/servidor. O servidor define um serviço como um conjunto de procedimentos que o cliente pode executar [21]. No RPC baseado em XML, uma chamada a um procedimento

remoto é usada tendo como base o protocolo SOAP. A especificação SOAP 1.1 define o padrão para troca de mensagens em um ambiente distribuído.

Java API for XML-based Remote Procedure Calls (JAX-RPC) tem como objetivo permitir a chamada de procedimentos remotos usando a plataforma Java. As chamadas e o seu retorno são transmitidos como mensagens SOAP sobre o HTTP.

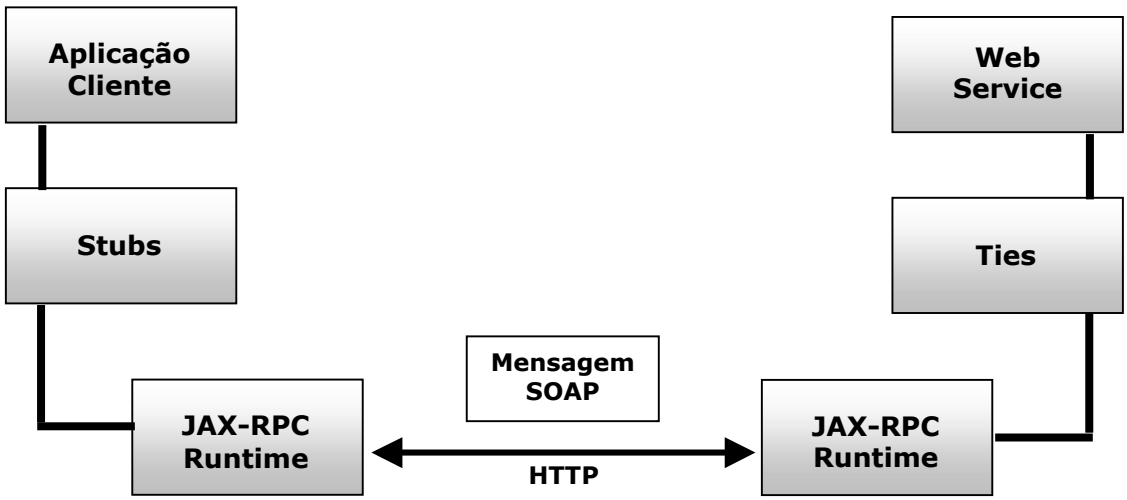

Figura 9 – Arquitetura da JAX-RPC

Embora JAX-RPC dependa de protocolos complexos, a API esconde esta complexidade do desenvolvedor de aplicações [11]. A API combina XML com RPC, o que permite que clientes possam executar procedimentos tanto em ambientes distribuídos quanto remotos. O uso da JAX-RPC tem a vantagem de permitir a independência de plataforma devido a linguagem de programação Java. Porém, o mais importante da JAX-RPC é que ela não é restritiva, isto é, um cliente pode acessar um *web service* que não esteja rodando na plataforma Java e vice-versa. Isto se deve ao fato da JAX-RPC ser baseada em tecnologias definidas pela W3C: HTTP, SOAP e WSDL.

Usando JAX-RPC, o desenvolvedor do lado servidor especifica os procedimentos remotos pela definição dos métodos e de uma interface em Java, além de criar as classes para os métodos. Os programas clientes chamam os métodos com um objeto local, chamado de *stub*, que representa o serviço remoto. *Stubs* e *ties* são classes de baixo nível que

permitem a comunicação entre o cliente e o servidor, sendo essenciais para o funcionamento de um *web service*.

4.4.1 Definição do serviço

Para a definição do web service é necessário criar uma interface que declara os métodos que podem ser chamados pelos clientes remotos [11]. A interface deve seguir as seguintes regras:

- Deve estender a classe `java.rmi.Remote`;
- Não são permitidas declarações de constantes: `public`, `final`, `static`;
- Os métodos devem lançar a exceção `java.rmi.RemoteException`;
- Os tipos de retorno dos métodos devem ser suportados pelos tipos JAX-RPC.

No exemplo abaixo temos a definição da interface, chamada `wserviceIF`, com dois métodos que poderão ser chamados pelos clientes remotos. Além da interface é necessário especificar uma classe que a implementa, que no exemplo foi chamada de `wserviceImpl`.

```
//interface wserviceIF
package wservice;

import java.rmi.Remote;
import java.rmi.RemoteException;

public interface wserviceIF extends Remote {
    public String olaMundo(String s) throws RemoteException;
    public int adicionar(int x) throws RemoteException;
}


// classe wserviceImpl
package wservice;

public class wserviceImpl implements wserviceIF {

    public String mensagem = new String("Olá ");

    public String olaMundo(String s) {
        return new String(mensagem + s);
    }

    public int adicionar(int x) {
        return x++;
    }
}
```

4.4.2 A ferramenta `xrpcc`

Parte integrante da implementação da JAX-RPC, a ferramenta `xrpcc` gera os artefatos necessários para a comunicação com RPC – *stubs* e *ties* – além de outros arquivos de configuração. A ferramenta também gera interfaces Remote Method Invocation (RMI) e documentos WSDL.

4.4.2.1 Arquivo de configuração

A ferramenta `xrpcc` faz a leitura de um documento XML que contém as configurações dos arquivos que devem ser gerados. O arquivo de configuração pode ter dois diferentes formatos: interface RMI ou documento WSDL. Se o arquivo de configuração for no formato RMI ele poderá gerar *stubs*, *ties*, arquivo de configuração do servidor e documentos WSDL; seguindo o seguinte formato:

```
<?xml version="1.0" encoding="UTF-8"?>
<configuration
  xmlns="http://java.sun.com/jax-rpc-ri/xrpcc-config">
    <rmi name=" "
         targetNamespace=" "
         typeNamespace=" ">
      <service name=" "
               packageName=" ">
        <interface name=" "
                   servantName=" "
                   soapAction=" "
                   soapActionBase=" "/>
      </service>
      <typeMappingRegistry>
      </typeMappingRegistry>
    </rmi>
</configuration>
```

Se o arquivo de configuração estiver no formato WSDL, ele também poderá gerar *stubs*, *ties*, arquivo de configuração do servidor e interfaces RMI. O documento XML deve ter o seguinte formato:

```
<?xml version="1.0" encoding="UTF-8"?>
<configuration
  xmlns="http://java.sun.com/jax-rpc-ri/xrpcc-config">
    <wsdl name=" "
         location=" "
         packageName=" ">
      <typeMappingRegistry>
      </typeMappingRegistry>
    </rmi>
</configuration>
```

O arquivo de configuração do servidor gerado pela ferramenta `xrpcc` tem os elementos abaixo mostrados:

```
port0.tie=wservice.wserviceIF_Tie
port0.servant=wservice.wserviceImpl
port0.name=wserviceIF
port0.wsdl.targetNamespace=http://wservice.org/wsdl
port0.wsdl.serviceName=wserviceTeste
port0.wsdl.portName=wserviseIFPort
portcount=1
wsdl.location=/WEB-INF/wserviceTeste_Service.wsdl
```

4.4.3 Criação do *deployment descriptor*

O *deployment descriptor* é um arquivo que fornece informações de configuração para o servidor web sobre os seus componentes (JSP ou servlets) [11]. Como o serviço é publicado como um servlet, o *deployment descriptor* deve fornecer alguns elementos para o servidor web.

```
<?xml version="1.0" encoding="UTF-8" ?>
<web-app>
  <display-name>wserviceApplication</display-name>
  <description>aplicação de teste Web Service</description>
 <servlet>
    <servlet-name>JAXRPCEndpoint</servlet-name>
    <display-name>JAXRPCEndpoint</display-name>
    <description>Ponto de acesso para a aplicação Web Service</description>
    <servlet-class>com.sun.xml.rpc.server.http.JAXRPCServlet</servlet-class>
  <init-param>
      <param-name>configuration.file</param-name>
      <param-value>/WEB-INF/wservice_Config.properties</param-value>
    </init-param>
    <load-on-startup>0</load-on-startup>
  </servlet>
 <servlet-mapping>
    <servlet-name>JAXRPCEndpoint</servlet-name>
    <url-pattern>/jaxrpc/*</url-pattern>
  </servlet-mapping>
 <session-config>
    <session-timeout>60</session-timeout>
  </session-config>
</web-app>
```

Vale ressaltar que o arquivo `wservice_Config.properties` foi gerado pela ferramenta `xrpcc` e o valor do elemento `<url-pattern>` – `jaxrpc` – deverá ser usado na URL que indica o ponto de acesso na rede do web service.

4.4.4 Definição do cliente

Para que um programa cliente possa fazer chamadas a um procedimento remoto, é necessário criar as classes *stubs* com a ferramenta `xrpcc`. No exemplo mostrado abaixo podemos notar o uso das classes `wserviceIF_Stub` e `wservice_Impl`; elas foram geradas com o auxílio da ferramenta `xrpcc`. O prefixo `wserviceIF` foi retirado da definição da interface do serviço e o prefixo `wservice` corresponde ao nome do serviço especificado no arquivo de configuração.

A seguir é mostrado um programa que faz a chamada do procedimento remoto `olaMundo` definido pelo web service que pode ser encontrado na URL especificada na propriedade do *stub*.

```
package cliente;

public class TesteCliente {

  public static void main() {
    try {
      wserviceIF_Stub stub = (HelloIF_Stub) (new wservice_Impl().getwserviceIF());
      stub._setProperty( javax.xml.rpc.Stub.ENDPOINT_ADDRESS_PROPERTY,
                         "http://localhost:8080/wservice/jaxrpc");
      System.out.println(stub.olaMundo("Andrés"));
    } catch (Exception ex) {
      ex.printStackTrace();
    }
  }

}
```

Podemos notar que inicialmente deve ser criada uma instância da classe `wserviceIF_stub` que representa localmente o método remoto. A seguir devemos informar uma propriedade que especifica qual será a URL onde se encontra o *web service* que será chamado. Finalmente temos uma chamada ao objeto remoto por meio do *stub*.

4.4.5 Tipos suportados pela JAX-RPC

A implementação da JAX-RPC faz o mapeamento de alguns tipos da linguagem de programação Java para definições em XML/WSDL. Podemos tomar como exemplo a classe `java.lang.String` para o tipo `xsd:string` do XML. Porém, somente algumas classes estão mapeadas, segue a relação de classes que foram mapeadas na implementação da JAX-RPC:

- `java.lang.Boolean`
- `java.lang.Byte`
- `java.lang.Double`
- `java.lang.Float`
- `java.lang.Integer`
- `java.lang.Long`
- `java.lang.Short`
- `java.lang.String`
- `java.math.BigDecimal`
- `java.math.BigInteger`
- `java.util.Calendar`
- `java.util.Date`

Além das classes acima, JAX-RPC suporta os tipos primitivos: `boolean`, `byte`, `double`, `float`, `int`, `long` e `short`. Vetores dos tipos suportados pela JAX-RPC também são permitidos. Dessa forma, podemos ter `int[]` ou `String[]`, mas vetores multidimensionais, como `float[][]`, não são permitidos.

4.4.5.1 Classes da aplicação

JAX-RPC também suporta classes que tenham sido escritas na aplicação. A especificação da JAX-RPC as define como *value types*, pois seus valores podem ser passados entre os clientes e o serviço remoto [11]. Para que o JAX-RPC possa suportar as classes da aplicação, elas devem ser as seguintes regras:

- Devem ter um construtor público default;
- Não devem implementar a interface `java.rmi.Remote`;
- Os seus campos devem ser suportados pela JAX-RPC.

4.4.6 Dynamic Invocation Interface

Quando uma aplicação cliente utiliza *stubs* estáticos, como foi visto na seção 4.4.4, devem ser especificados, em tempo de projeto, os nomes dos métodos e a URL onde está publicado o *web service*. Com o uso da *Dynamic Invocation Interface* (DII), um cliente pode executar procedimentos remotos mesmo que a assinatura do método ou o nome do serviço sejam conhecidos somente em tempo de execução.

Comparados com clientes que usam *stubs* estáticos, clientes que utilizam DII são mais difíceis de codificar e testar. Entretanto, clientes DII têm mais flexibilidade já que podem: descobrir serviços dinamicamente, configurar e executar chamadas remotas.

O seguinte exemplo mostra um programa que acessa um *web service* usando DII:

```
package wservice.dynamic;

import javax.xml.rpc.Call;
import javax.xml.rpc.Service;
import javax.xml.rpc.JAXRPCException;
import javax.xml.rpc.namespace.QName;
import javax.xml.rpc.ServiceFactory;
import javax.xml.rpc.ParameterMode;

public class ClienteDII {
  private static String qnameService = "wservice";
  private static String qnamePort = "wserviceIF";
  private static String BODY_NAMESPACE_VALUE = "http://wservice.org/wsdl";
  private static String ENCODING_STYLE_PROPERTY="javax.xml.rpc.encodingstyle.namespace.uri";
  private static String NS_XSD = "http://www.w3.org/2001/XMLSchema";
  private static String URI_ENCODING = "http://schemas.xmlsoap.org/soap/encoding/";

  public static void main() {
    try {
      String endpoint = "http://localhost:8080/wservice/jaxrpc";
      ServiceFactory factory = ServiceFactory.newInstance();
      Service service = factory.createService(new QName(qnameService));

      QName port = new QName(qnamePort);

      Call call = service.createCall();
      call.setPortTypeName(port);
      call.setTargetEndpointAddress(endpoint);
      call.setProperty(Call.SOAPACTION_USE_PROPERTY, new Boolean(true));
      call.setProperty(Call.SOAPACTION_URI_PROPERTY, "");
      call.setProperty(ENCODING_STYLE_PROPERTY, URI_ENCODING);

      QName QNAME_TYPE_STRING = new QName(NS_XSD, "string");
      call.setReturnType(QNAME_TYPE_STRING);
      call.setOperationName(new QName (BODY_NAMESPACE_VALUE, "olaMundo"));
      call.addParameter("String_1", QNAME_TYPE_STRING, ParameterMode.PARAM_MODE_IN);

      String[] params = { new String("Andrés") };
      String result = (String)call.invoke(params);
      System.out.println(result);

    } catch (Exception ex) {
      ex.printStackTrace();
    }
  }
}
```

4.5 Cenário com o uso das tecnologias de *web service*

Atualmente, o cenário mais comum entre as empresas que disponibilizam informações pela Internet pode ser visto na Figura 10. Temos como exemplo uma agência de turismo onde o cliente encontra informações sobre hotéis, vôos e aluguel de carros para uma determinada

cidade. Os dados disponibilizados são *copiados* da fonte de origem e *colados* nas páginas da agência de turismo.

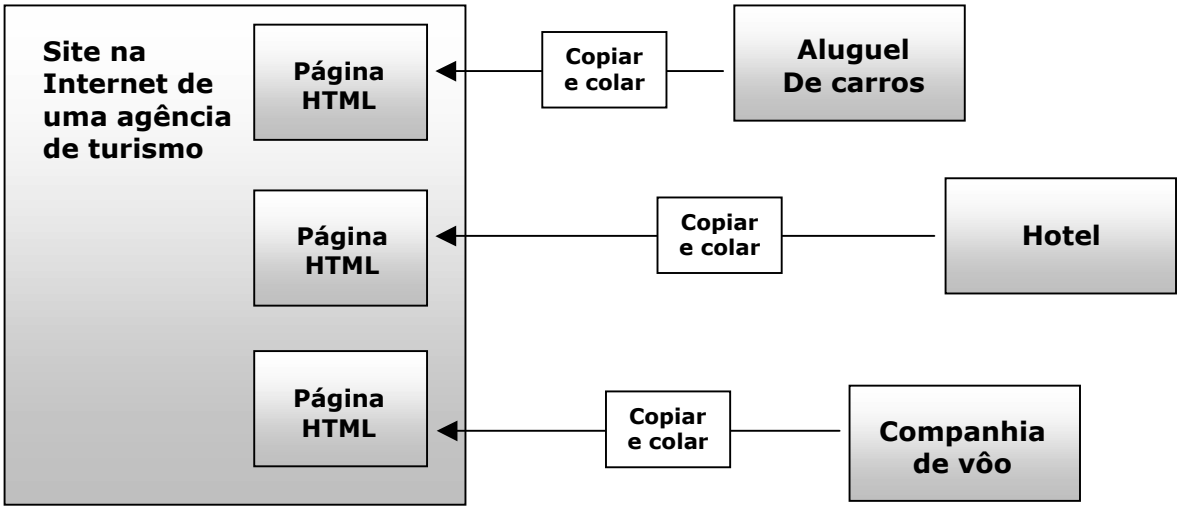

Figura 10 – Cenário sem o uso das tecnologias de web service

Neste cenário, a agência de turismo vai precisar de constantes atualizações no seu site para que possa refletir as alterações realizadas nos dados dos seus parceiros. Este encargo vai implicar em maiores custos de pessoal, já que quanto mais parceiros a agência tiver mais informações precisarão ser mantidas atualizadas. O ideal seria ter páginas dinâmicas que pudessem acompanhar as mudanças realizadas pelas empresas parceiras, evitando o *copiar e colar* para as páginas HTML.

Na Figura 11 podemos ver um cenário da mesma empresa de turismo só que com a utilização de *web services*. Novos parceiros que tenham se cadastrado em provedores de serviços podem ser encontrados usando a API JAXR. Para que os métodos remotos possam ser chamados, devemos criar as classes *stubs* usando a ferramenta `xrpcc` para a geração no lado cliente. Com a API JAX-RPC podemos executar a chamada aos métodos remotos, passar parâmetros e receber os valores de retorno. Para os parceiros que prepararam o seu *web service* para receber mensagens SOAP, podemos enviar uma mensagem usando JAXM e processar a sua resposta usando a API JAXP.

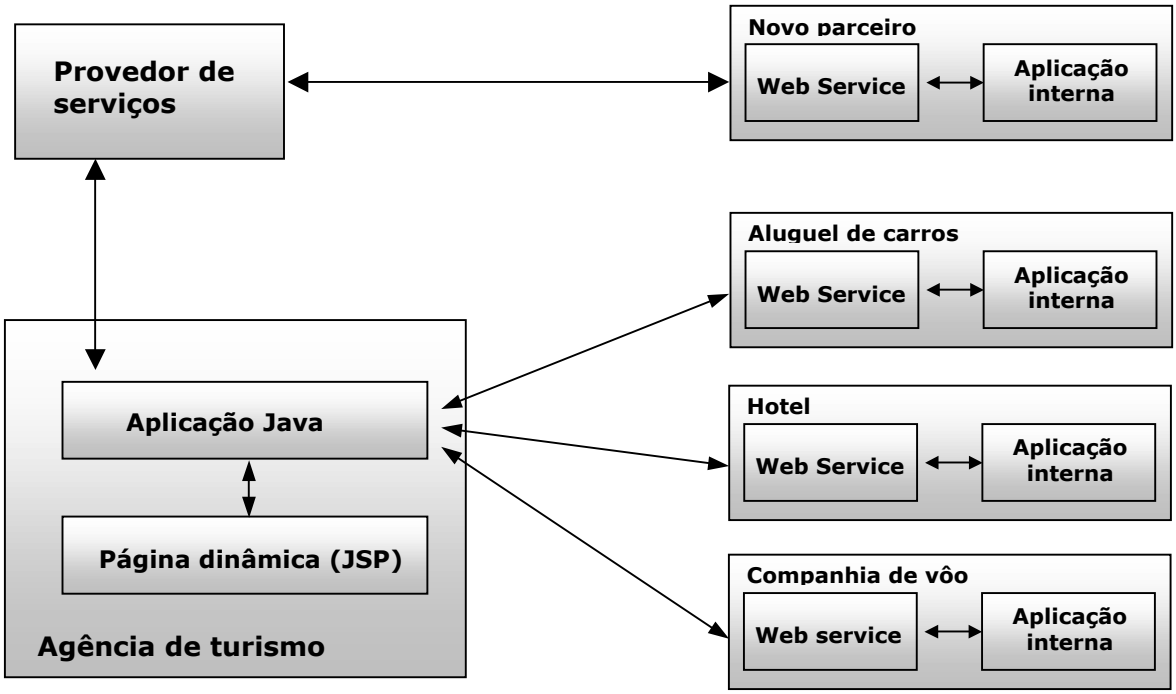

Figura 11 – Cenário com o uso das tecnologias de web service

Capítulo 5

Ferramenta de gerenciamento de web services

5.1 Introdução

Embora o pacote JWSDP forneça mecanismos de geração e publicação de *web services* através de suas APIs, é necessário ter uma ferramenta para o gerenciamento dos serviços que são criados e distribuídos nos diversos provedores de serviços.

No Capítulo 4 foram descritas várias operações que são utilizadas pela ferramenta de gerenciamento, entre elas podemos citar: consulta e publicação de registros UDDI, geração das classes *stubs* e *ties*, chamada a métodos remotos usando RPC, envio e recebimento de mensagens SOAP, além de permitir publicar o serviço no servidor web.

A ferramenta de gerenciamento mantém dados referentes a cada serviço em um de banco de dados Oracle e faz a comunicação com a linguagem Java por meio do Java Database Connectivity (JDBC). A publicação do web service é feita no servidor web Tomcat por meio de arquivos com extensão .WAR; onde são armazenadas as classes Java utilizadas no lado servidor e vários arquivos de configuração do *web service*.

5.2 Requisitos da ferramenta de gerenciamento

Como foi listado na Seção 3.4, a ferramenta de gerenciamento deve atender uma série de requisitos para que o usuário possa ter ganhos de produtividade com relação ao desenvolvimento e publicação de *web services*.

A seguir temos uma lista de necessidades que são atendidas pela ferramenta:

- Armazenamento de dados do *web service* para permitir um bom gerenciamento;
- Integração do *web service* com a sua publicação nos *services provider*;
- Geração de artefatos para publicação de *web services* no servidor web;
- Chamada de procedimentos remotos com geração de artefatos necessários para aplicações cliente;
- Envio de mensagens SOAP sem a criação de programas específicos.

5.3 Demonstração da ferramenta de gerenciamento

Veremos a seguir as funções da ferramenta de gerenciamento. Serão vistos exemplos de como o usuário deve proceder para realizar as principais operações envolvendo *web services*.

5.3.1 Geração de documentos XML

Como foi visto na Seção 4.4.2.1, para que a ferramenta `xrpcc` possa gerar as classes *ties* é necessário criar um arquivo de configuração. Este arquivo é um documento XML que define alguns parâmetros para o *web service*, como: descrição, nome da package e da interface, entre outros. Na Figura 12 podemos ver o JSP quer permite a inserção dos dados de um *web service*.

O arquivo gerado é chamado de config.xml e é passado como parâmetro na chamada da ferramenta `xrppc`. Segue a listagem de um arquivo de configuração gerado pela ferramenta de gerenciamento.

```
<?xml version="1.0" encoding="UTF-8"?>
<configuration
   xmlns="http://java.sun.com/jax-rpc-ri/xrpcc-config">
   <rmi name="OlaMundo_Service"
        targetNamespace="http://teste.org/wsdl"
        typeNamespace="http://teste.org/types">
     <service name="OlaMundo"
              packageName="teste">
       <interface name="teste.testeIF"
                  servantName="teste.testeImpl"/>
     </service>
   </rmi>
</configuration>
```

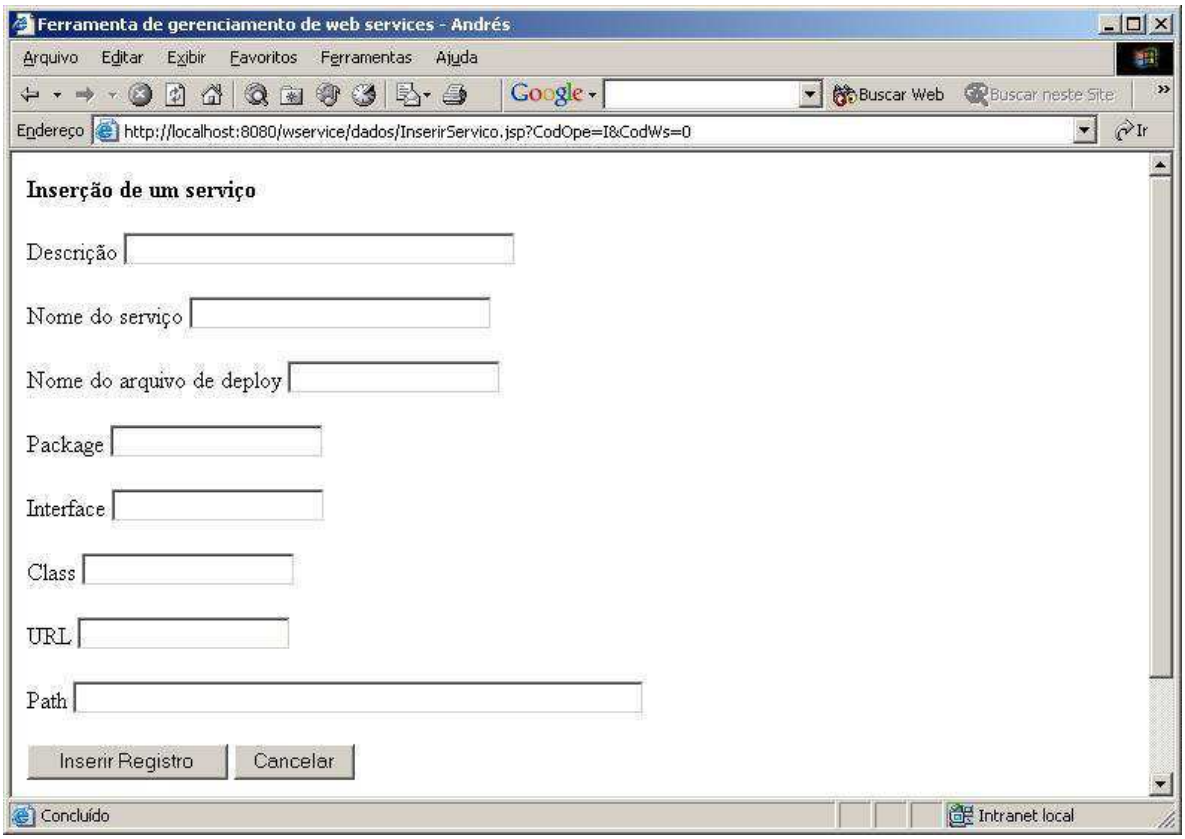

Figura 12 – JSP para inserir os dados do web service

O pacote JWSDP não fornece ajuda na criação do arquivo de configuração. Dessa forma, o usuário que não usa a ferramenta de gerenciamento deve fazer a edição do arquivo manualmente, o que pode gerar problemas de sintaxe. Para que a ferramenta possa criar o arquivo de configuração são usados os pacotes `jdbc` e `xml`, que permitem ler os dados armazenados no banco de dados e gerar o documento XML, respectivamente.

Outro documento XML que deve ser criado é o *deployment descriptor*, ele fornece os parâmetros necessários para que o servidor web possa publicar o serviço. Este arquivo normalmente recebe o nome de web.xml e deve estar presente no arquivo WAR publicado. A seguir temos a listagem de um arquivo *deployment descriptor* gerado pela ferramenta de gerenciamento. Os dados em destaque foram obtidos do JSP mostrado na Figura 12.

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app>
  <display-name>OlaMundoApplication</display-name>
  <description>Descrição da aplicação</description>
  <servlet>
    <servlet-name>JAXRPCEndpoint</servlet-name>
    <display-name>JAXRPCEndpoint</display-name>
```

```
    <description>Descrição do web service</description>
    <servlet-class>com.sun.xml.rpc.server.http.JAXRPCServlet</servlet-class>
    <init-param>
      <param-name>configuration.file</param-name>
        <param-value>/WEB-INF/OlaMundo_Config.properties</param-value>
    </init-param>
    <load-on-startup>0</load-on-startup>
  </servlet>
  <servlet-mapping>
    <servlet-name>JAXRPCEndpoint</servlet-name>
    <url-pattern>/wservice/*</url-pattern>
  </servlet-mapping>
  <session-config>
    <session-timeout>60</session-timeout>
  </session-config>
</web-app>
```

Da mesma forma que o arquivo de configuração, para criar o arquivo web.xml foram usados os pacotes `jdbc` e `xml`. Os dois documentos XML são gravados em um diretório que é especificado pelo campo `path`, como mostrado no JSP da Figura 12.

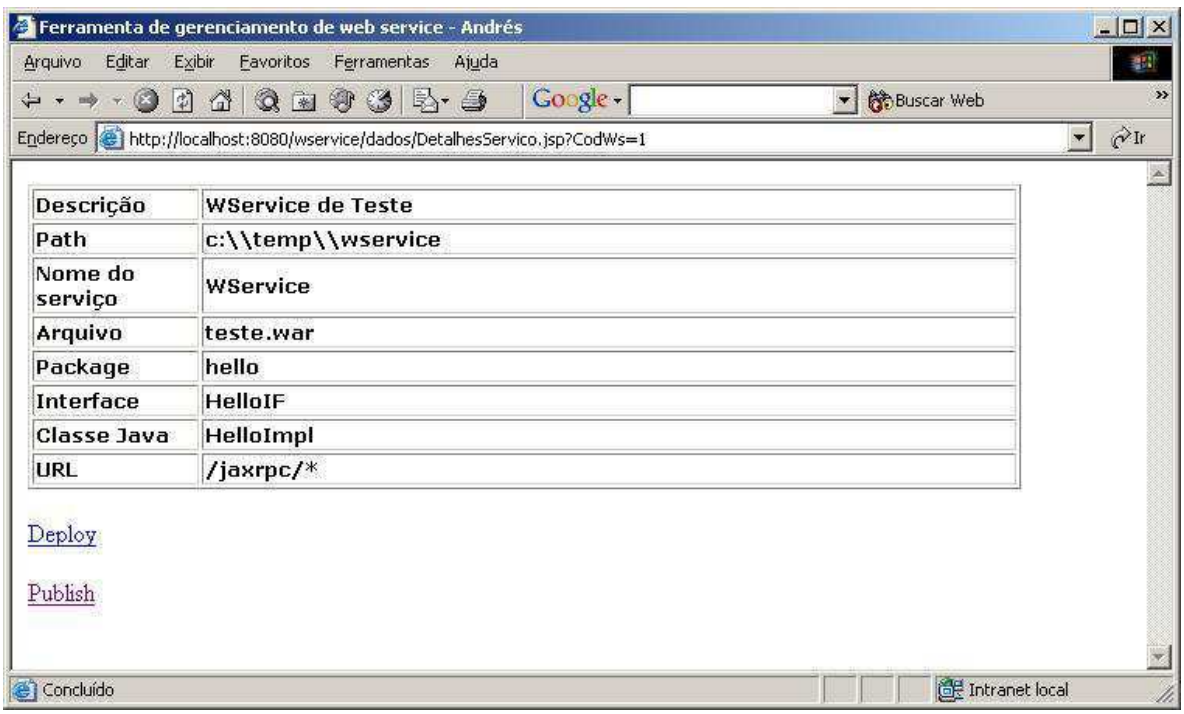

Figura 13 – JSP que permite a publicação do serviço no registro UDDI e no servidor web

5.3.2 Publicação de *web services*

Antes de publicar um *web service* no servidor web é necessário criar as classes Java que contém a lógica de negócio da aplicação. A edição e a compilação destas classes são os únicos pontos em que a ferramenta de gerenciamento não fornece ajuda. Dessa forma, o usuário deve utilizar-se de uma IDE para a geração das classes compiladas. O local onde estão

armazenadas as classes deve ser especificado no campo `path` quando for feita a inserção dos dados do *web service*.

A Figura 13 mostra o JSP que permite fazer tanto a publicação do web service no servidor web quanto a publicação em algum provedor de serviços.

Para realizar a publicação do *web service* no servidor web devemos clicar no link *Deploy*, que vai disparar a execução das seguintes tarefas:

1. Leitura dos dados do *web service* armazenados no banco de dados;
2. Geração do arquivo de configuração (config.xml) no diretório especificado;
3. Geração do *deployment descriptor* (web.xml) no diretório especificado;
4. Chamada para a ferramenta `xrpcc` que vai permitir gerar:
   4.1. As classes *ties* a partir das classes Java compiladas;
   4.2. O documento WSDL;
   4.3. O arquivo de configuração do servidor;
5. Alteração do arquivo de configuração para colocar a localização do documento WSDL;
6. Geração do arquivo para publicação no servidor web (WAR);
7. Cópia do arquivo WAR para o diretório de publicação do servidor web.

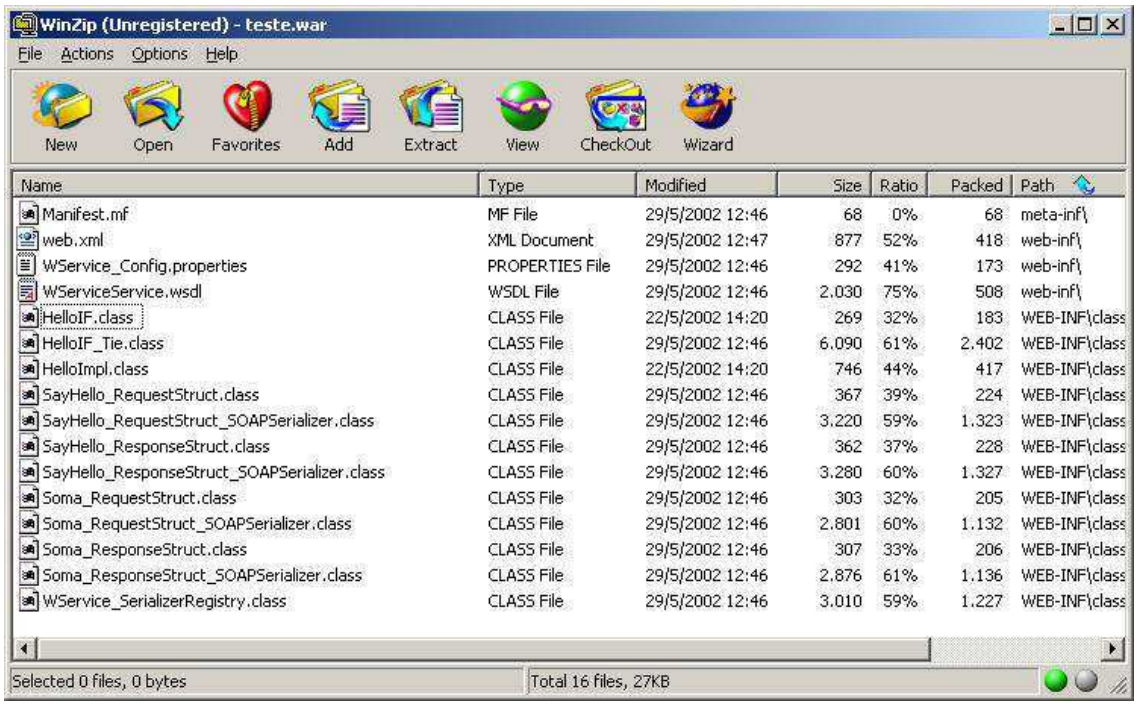

Figura 14 – Arquivo para publicação no servidor web

O arquivo WAR, que será publicado no servidor web, mostrado na Figura 14 é composto de:

- Arquivo de configuração do servidor;
- Documento WDSL;
- *Deployment descriptor*;
- Classes *ties*.

Depois que o web service foi publicado podemos verificar se ele está disponível digitando a URL: `http://localhost:8080/arquivo_war/campo_URL`, onde:

- `Arquivo_war` – corresponde ao campo `arquivo` mostrado na Figura 13;
- `Campo_URL` – corresponde ao campo `URL` também mostrado no Figura 13.

A Figura 15 mostra que o serviço está publicado no servidor web.

Figura 15 – *Web service* publicado no servidor web

O arquivo de configuração do servidor gerado, possui uma entrada que identifica a localização do documento WSDL no arquivo publicado (WAR). A sintaxe é a seguinte: `wsdl.location=/WEB-INF/Nome_do_Servico.wsdl`, onde:

- `Nome_do_Serviço` – corresponde ao campo `nome do serviço` mostrado na Figura 13.

A definição do local do documento WSDL é o que permite que aplicações cliente possam criar classes *stubs* através de arquivos de configuração baseados em documentos WSDL.

5.3.3 Registro de entidades usando UDDI

Como foi descrito na Seção 2.2.4, os provedores de serviços permitem que empresas possam publicar seus *web services* na Internet através de registros UDDI, de forma que os clientes possam ter um catálogo de informações sobre os serviços disponíveis.

O cadastro do *web service* pode ser feito manualmente, no próprio *site* do provedor de serviços, ou automaticamente através de programas com a API JAXR. Temos algumas vantagens de registrar o *web service* com a ajuda de uma ferramenta automatizada: os dados cadastrados retratam fielmente o serviço que foi publicado, mudanças no *web service* podem ser refletidas facilmente para o registro UDDI, entre outras.

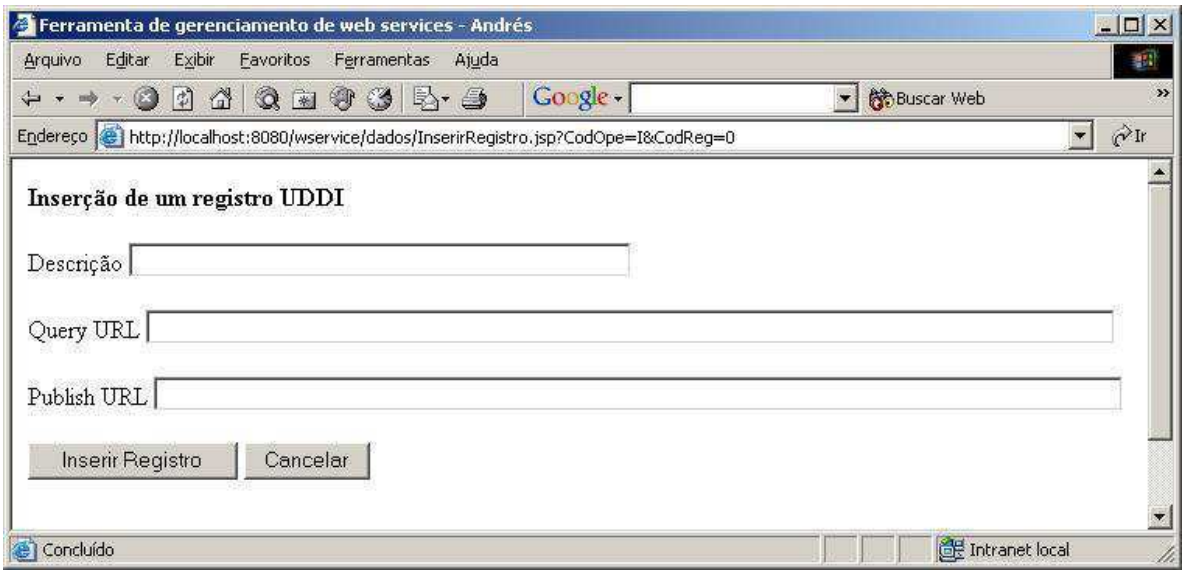

Figura 16 – JSP para inserção de provedores de serviços

A ferramenta de gerenciamento permite cadastrar provedores de serviços para que possam ser usados na publicação do *web services*. A Figura 16 mostra o JSP que insere os

dados referentes ao provedor de serviços. Os campos `query URL` e `publish URL` servem para configurar as propriedades da conexão, como foi visto na Seção 4.2.2.1.

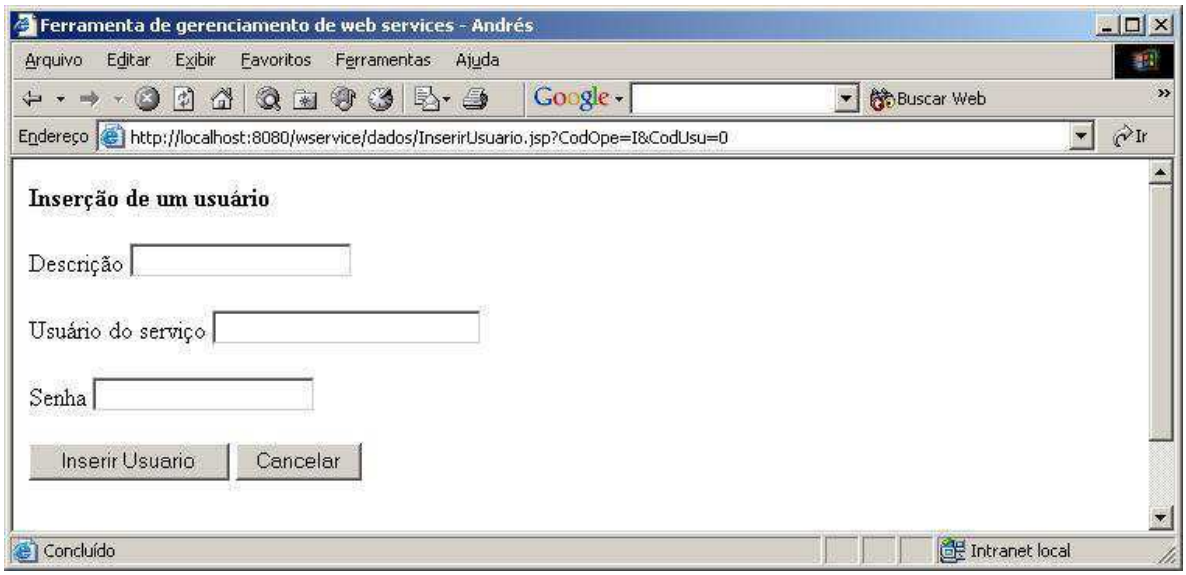

Figura 17 – JSP para inserir usuários de provedores de serviços

Para inserir novos registros e fazer a manutenção dos registros já publicados, os provedores de serviços obrigam ao usuário a criar um cadastro onde deverá ser definido o usuário e a senha. A ferramenta de gerenciamento permite manter um cadastro de usuários para que sejam usados quando for aberta uma conexão com autorização para um provedor de serviços. O JSP mostrado na Figura 17 mostra os dados para a inserção de um usuário.

Os dados necessários para o cadastro do registro UDDI são: organização, contato, serviços e pontos de acesso. Estes dados devem ser armazenados na ferramenta de gerenciamento para que a mesma fazer a sua publicação no provedor de serviços escolhido. O JSP mostrado na Figura 18 indica os dados necessários para a publicação da organização.

Somente depois de cadastrar na ferramenta de gerenciamento o provedor de serviços, o usuário e a organização é que o web service pode ser publicado. Ao clicar no link *publish* do JSP mostrado na Figura 13 será chamado um outro JSP que permite escolher os parâmetros para a publicação do serviço. Estes parâmetros são, na verdade, os cadastros de provedores de serviços, usuários e organizações.

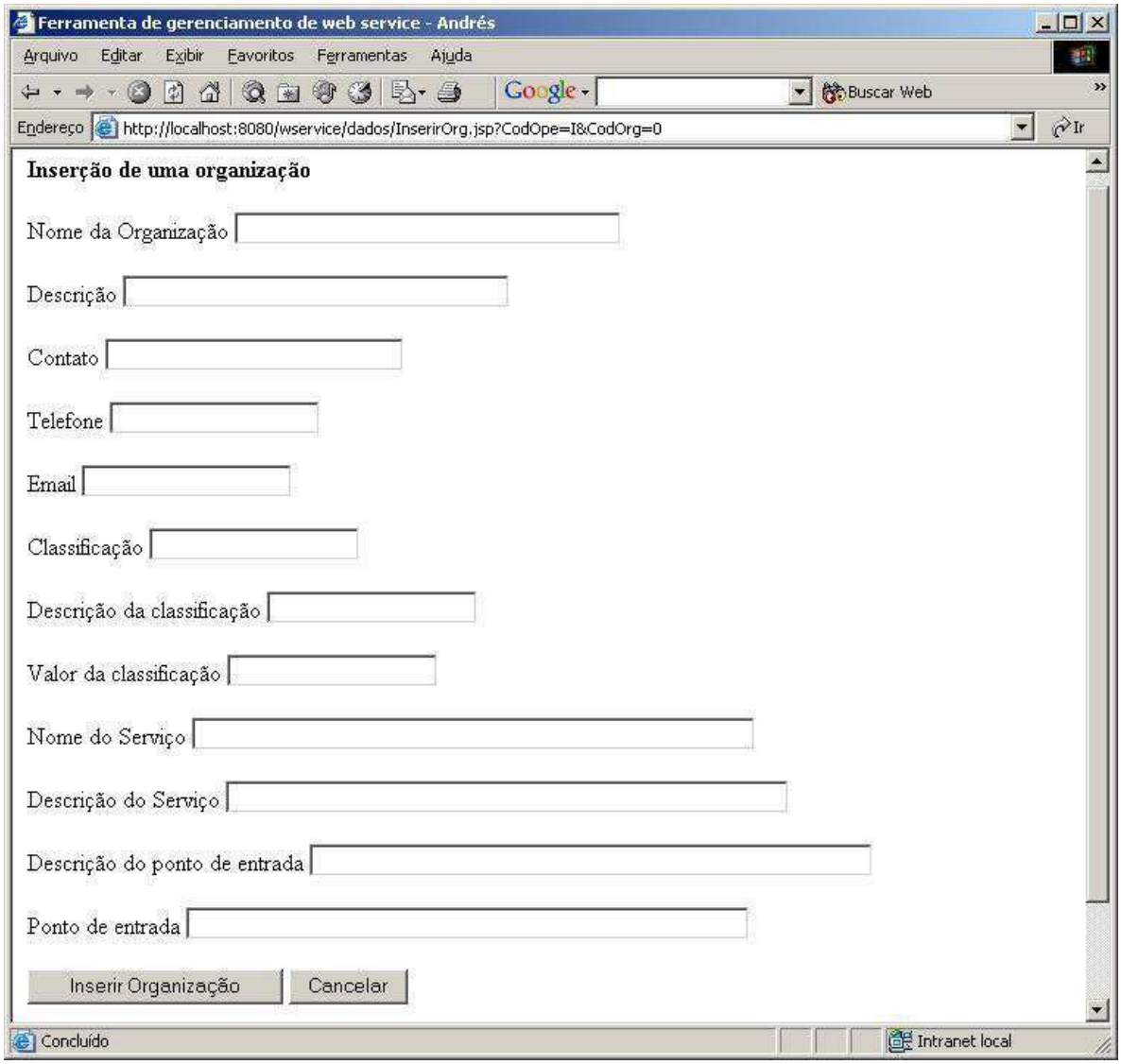

Figura 18 – JSP para cadastro dos dados da organização

Ao clicar no botão `publicar`, mostrado na Figura 19, o JSP enviará os parâmetros escolhidos para a classe `adicionaRegistro` que vai se encarregar de:

1. Ler os dados do provedor de serviços escolhido;
2. Configurar a propriedades para estabelecer a conexão;
3. Abrir uma conexão autorizada baseado no usuário e senha que foram escolhidos;
4. Preencher os dados da organização e contato;
5. Preencher os dados do serviço e do ponto de acesso na rede;
6. Adicionar os dados da organização no provedor de serviços;
7. Armazenar no banco de dados a chave que foi gerada para a organização.

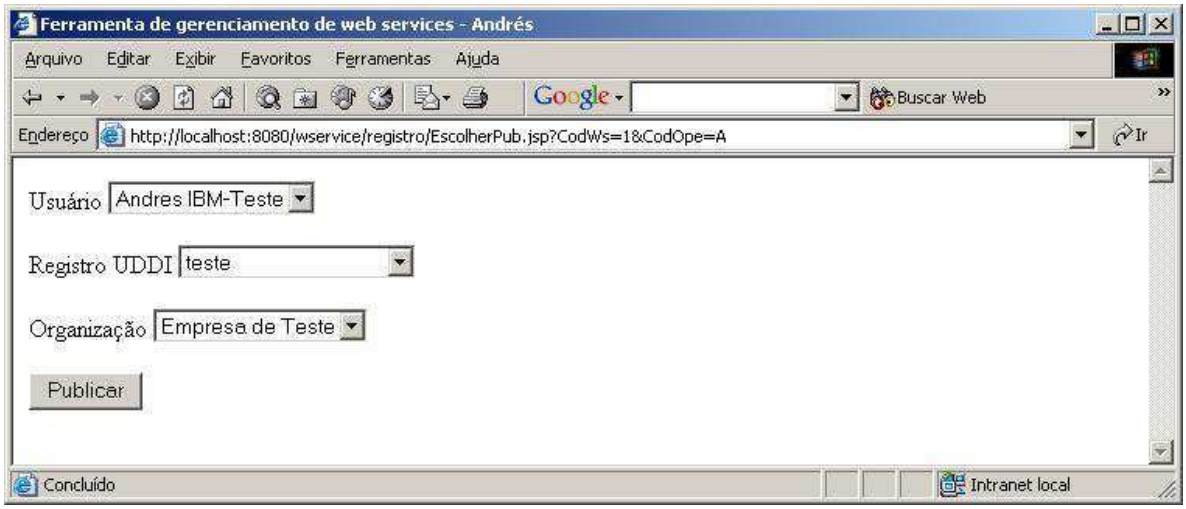

Figura 19 – JSP para publicar o serviço no registro UDDI

Na Figura 20 podemos ver o provedor de serviços da IBM depois da inserção de uma organização por meio da ferramenta de gerenciamento.

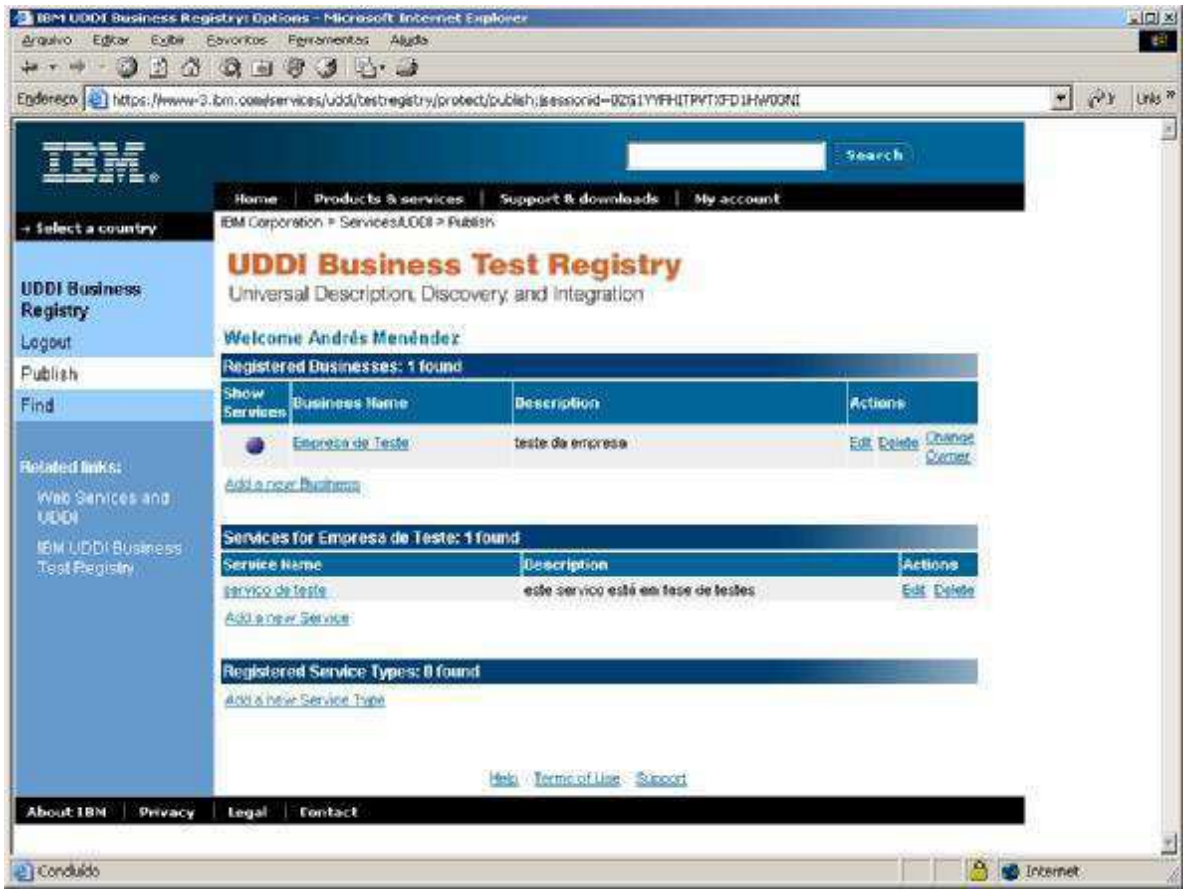

Figura 20 – Provedor de serviços de teste da IBM

5.3.4 Geração de *stubs* para aplicações cliente

A ferramenta de gerenciamento permite controlar aplicações cliente que fazem acesso a *web services*. Para realizar a comunicação é necessário criar as classes *stubs* utilizando o documento WSDL que foi disponibilizado pelo *web service*.

Como foi visto na Seção 4.4, as classes *stubs* são responsáveis pela comunicação entre a aplicação cliente e JAX-Runtime, que é o responsável pela troca de mensagens SOAP com o lado servidor. As classes *stubs* são equivalentes às classes *ties*, com a diferença que elas atuam no lado cliente.

O cadastro do serviço cliente é composto pelos campos mostrados na Figura 21. O campo `URL` indica onde está localizado o documento WSDL do *web service* que se deseja acessar. Em `package` podemos definir o nome da *package* que as classes *stubs* serão geradas e o campo `path` indica a localização no disco.

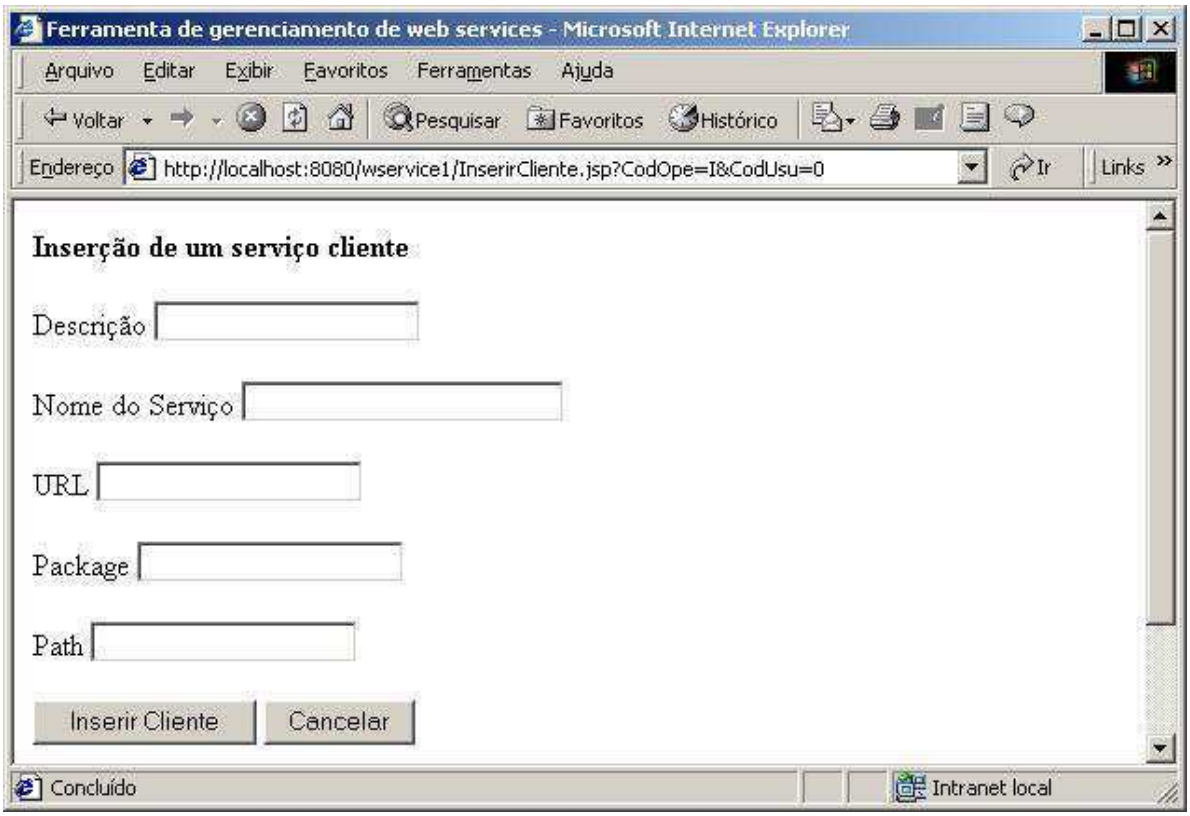

Figura 21 – Cadastro de dados para aplicações cliente

5.3.5 Envio de mensagens SOAP

Para enviar mensagens SOAP usando JAXM devemos criar um documento XML com a especificação SOAP 1.1, de acordo com o que visto na seção 4.3.1. A criação da mensagem SOAP pode ser feita manualmente, com editores de texto, ou com o uso da ferramenta de gerenciamento.

A ferramenta permite que seja definido tanto o cabeçalho da mensagem quanto os atributos que a compõem. Na Seção 2.2.2.3 vimos uma mensagem SOAP que pode ser enviada para o site de pesquisa Google. Os atributos usados para a montagem da mensagem estão armazenados no banco de dados utilizado pela ferramenta de gerenciamento. A seguir temos a mensagem SOAP enviada com destaque para os campos que foram retirados do banco.

```
<SOAP-ENV:Envelope
          xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
          xmlns:xsi="http://www.w3.org/1999/XMLSchema-instance"
          xmlns:xsd="http://www.w3.org/1999/XMLSchema">
    <SOAP-ENV:Body>
       <nsl: doGoogleSearch xmlns:ns1="urn:GoogleSearch"
             SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
             <key xsi:type="xsd:string">CcnEHaJ3SGmaphcqn0ALb8GbJ0azC2fz</key>
             <q xsi:type="xsd:string">"web service"</q>
             <start xsi:type="xsd:int">0</start>
             <maxResults xsi:type="xsd:int">10</maxResults>
             <filter xsi:type="xsd:boolean">true</filter>
             <restrict xsi:type="xsd:string"></restrict>
             <safeSearch xsi:type="xsd:boolean">false</safeSearch>
             <lr xsi:type="xsd:string"></lr>
             <ie xsi:type="xsd:string">latin1</ie>
             <oe xsi:type="xsd:string">latin1</oe>
       </ns1: doGoogleSearch >
    </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Além dos atributos é necessário armazenar também a URL para onde será enviada a mensagem e o nome do arquivo onde será gravado o documento de resposta. Este arquivo pode ser exibido ou processado de acordo com as necessidades de cada aplicação.

Ao clicar no link *Enviar* do JSP exibido na Figura 22, são executadas as seguintes tarefas:

1. Ler os dados da mensagem;
2. Criar a mensagem SOAP;

3. Preencher o envelope da mensagem SOAP;
4. Preencher o corpo (*body*) da mensagem SOAP;
5. Enviar a mensagem SOAP para a URL de destino;
6. Receber a resposta e gravar em um arquivo XML.

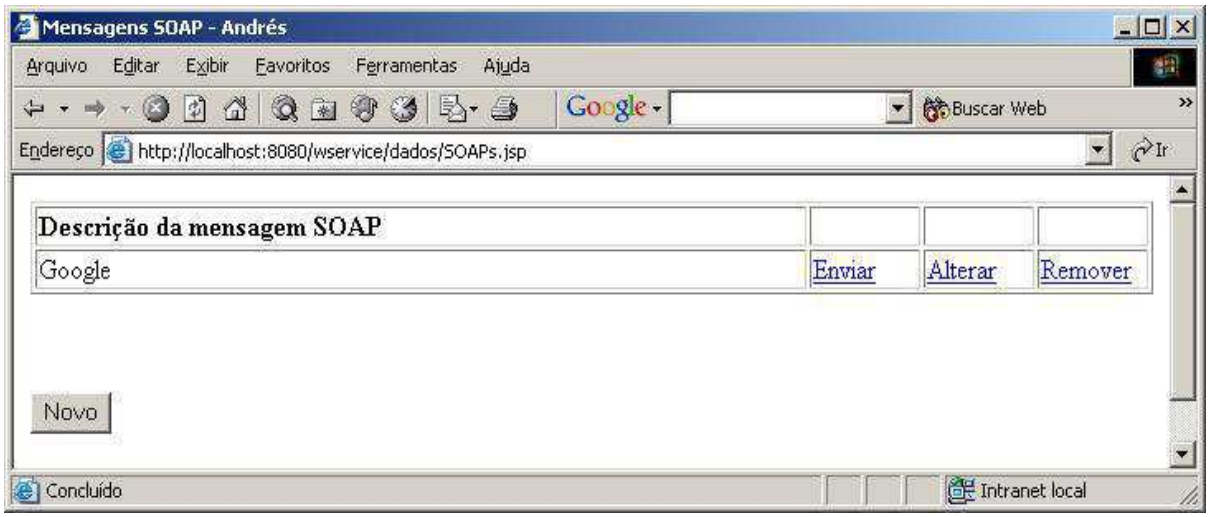

Figura 22 – JSP para mostrar e enviar mensagens SOAP

5.4 Arquitetura da ferramenta de gerenciamento

A implementação da ferramenta de gerenciamento foi dividida em vários pacotes, onde cada um define tarefas específicas. Os pacotes são: dados, rpc, uddi, soap, jdbc e xml.

A API JDBC permite que aplicações Java possam se comunicar com bancos de dados de uma maneira padronizada. Entretanto, esta API possui várias particularidades que a tornam trabalhosa do ponto de vista de codificação dos programas. Para minimizar este trabalho foi desenvolvido o pacote `jdbc`, que possui um conjunto de classes que facilitam tanto o trabalho de conexão com banco de dados quanto da execução de comandos SQL e *stored procedures*.

O pacote `xml` é composto por um conjunto de classes que permitem a leitura de documentos XML utilizando SAX e DOM. Além disso, o pacote `xml` permite a gravação de arquivos XML baseados em DOM.

O pacote `dados` possui um conjunto de classes que permitem a manutenção dos dados gerenciados pela ferramenta: usuários, organizações, serviços, serviços cliente, registros

UDDI e mensagens SOAP. Cada classe é composta por um conjunto de métodos que permitem fazer inclusões, alterações e remoções nos dados. As classes do pacote `dados` se utilizam das classes do pacote `jdbc` para manter os dados e do pacote `xml` para verificar que tipo de operação deve ser realizada, já que os comandos SQL referentes a cada classe estão armazenados em arquivos XML.

O pacote `rpc` possui as classes que geram os artefatos necessários para a publicação de *web services* no servidor web, publicam o serviço no provedor de serviços e permitem a criação das classes *stubs* para aplicações cliente. As classes do pacote `rpc` utilizam as classes do pacote `xml` e do pacote `dados`.

O pacote `uddi` fornece as classes que são necessárias para inserir, remover e consultar registros UDDI armazenados nos provedores de serviços. As classes `adicionaRegistro` e `removeRegistro` utilizam o pacote `dados` para executar as suas tarefas.

O pacote `soap` é composto das classes que permitem enviar mensagens SOAP para um *web service*. A montagem da mensagem é realizada por meio da leitura dos dados armazenados no banco, que é enviada para a URL destino.

Na Figura 23 podemos ver o diagrama de classes da ferramenta de gerenciamento de web services.

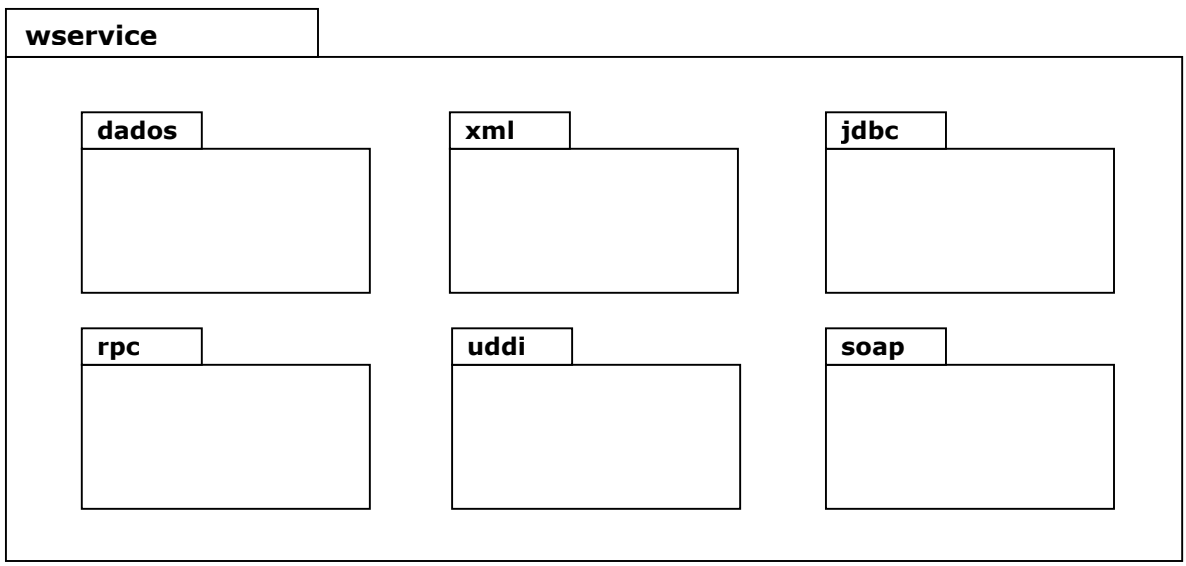

Figura 23 – Pacotes da ferramenta de gerenciamento

5.5 Comparativo entre as ferramentas de desenvolvimento de *web services*

A Tabela 3 mostra um comparativo entre o pacote WSTK da IBM, o pacote JWSDP sozinho e a ferramenta de gerenciamento sobre o JWSDP.

Funcionalidade	WSTK	JWSDP	Ferramenta
Parser XML para Java			
Manutenção de registros UDDI públicos e privados			
Geração de documentos WSDL			
Geração de arquivos de configuração			
Envio e recebimento de mensagens usando SOAP			
Geração de classes de baixo nível para comunicação com o protocolo SOAP			
Compilação de classes Java			
Publicação de web services no servidor web			
Gerenciamento de usuários dos provedores de serviços			
Gerenciamento das organizações e serviços disponíveis			
Gerenciamento dos web services disponíveis			
Gerenciamento dos serviços cliente disponíveis			
Geração e envio de mensagens SOAP sem programação específica			
Integração entre o *web service* e a sua publicação no provedor de serviços			

Tabela 3 – Comparativo entre as ferramentas de desenvolvimento

Como pode ser visto, o pacote JWSDP sendo usado sem a ferramenta de gerenciamento tem praticamente as mesmas funcionalidades da ferramenta WSTK da IBM.

Na Seção 5.2 relacionamos as necessidades da ferramenta de gerenciamento. Todas elas estão relacionadas nas funcionalidades da Tabela 3, sendo que a grande maioria está disponível somente na implementação da ferramenta de gerenciamento.

Além do gerenciamento dos dados relacionados ao *web service*, a ferramenta de gerenciamento permite uma integração entre as classes e artefatos necessários para a geração do serviço e os dados que estão publicados no provedor de serviços. Outro ponto de destaque é a possibilidade de enviar mensagens SOAP sem ter que fazer programas em Java para configurar os parâmetros da mensagem a ser enviada.

Capítulo 6

Exemplo prático do uso da ferramenta

Para mostrar o uso prático da ferramenta de gerenciamento de *web services* foi implementada uma aplicação que pode ser dividida em duas partes. A primeira parte é a criação de um *web service* para um hotel fictício que permitirá cadastrar clientes e fazer a reserva de quartos. A segunda parte é a criação de uma aplicação cliente para uma empresa de turismo que vai usar o *web service* disponibilizado pelo hotel.

6.1 Implementação da interface provedora

O hotel BoaViagem deseja disponibilizar a reserva de quartos pela Internet como um *web service*. Dessa forma, aplicações escritas para agências de turismo podem executar a reserva de quartos para seus clientes de uma maneira muito mais eficiente. O diagrama de classes da Figura 24 mostra de forma simplificada através de notação UML (*Unified Modeling Language*) como é a aplicação do hotel BoaViagem.

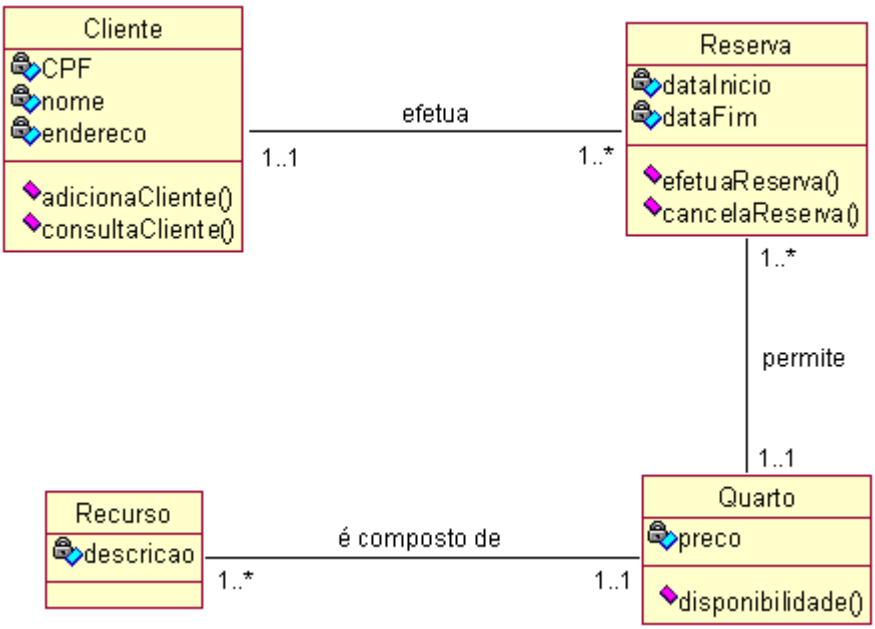

Figura 24 – Diagrama de classes do hotel BoaViagem

O *web service* que se deseja disponibilizar precisa atender as seguintes necessidades:

- Consulta de quartos, seus recursos e preços;
- Consulta de disponibilidade de quartos para um determinado período;
- Inserção de novos clientes;
- Reserva de quartos.

O primeiro passo é criar a interface e a classe que a implementa. Nesta etapa, a ferramenta de gerenciamento não ajuda o desenvolvedor que deve utilizar uma IDE para editar e compilar as classes Java. A seguir podemos ver as classes depois de implementadas:

```
package hotel;

import java.rmi.Remote;
import java.rmi.RemoteException;
import java.util.Date;
import java.lang.Boolean;

public interface hotelIF extends Remote {
    public String[] quartos() throws RemoteException;
    public String[] disponibilidade(Date inicio, Date fim) throws RemoteException;
    public Boolean inserirCliente(String cpf, String nome,
                                  String endereco) throws RemoteException;
    public Boolean reservaQuarto(String cpf, String quarto,
                                 Date inicio, Date fim) throws RemoteException;
}
```

```
package hotel;

import hotel.dados.*;
import java.util.*;
import java.lang.Boolean;

public class hotelImpl implements hotelIF {

    public String[] quartos() {
      Quartos quarto = new Quartos();
      Collection colQuartos = quarto.dadosQuarto();

      String[] resultado = new String[colQuartos.size()];

      Iterator itQuartos = colQuartos.iterator();
      int i = 0;
      while (itQuartos.hasNext() ) {
        resultado[i] = (String) itQuartos.next();
        i++;
      }
      return resultado;
    }

    public String[] disponibilidade(Date inicio, Date fim) {
      Quartos quarto = new Quartos();
      Collection colQuartos = quarto.quartosDisp(inicio, fim);
```

```
      String[] resultado = new String[colQuartos.size()];

      Iterator itQuartos = colQuartos.iterator();
      int i = 0;
      while (itQuartos.hasNext() ) {
        resultado[i] = (String) itQuartos.next();
        i++;
      }
      return resultado;
    }

    public Boolean inserirCliente(String cpf, String nome, String endereco) {
      Clientes cliente = new Clientes();
      boolean ok = cliente.inserir(cpf, nome, endereco);
      Boolean b = new Boolean(ok);
      return b;
    }

    public Boolean reservaQuarto(String cpf, String codQuarto, Date inicio, Date
fim) {
      Quartos quarto = new Quartos();
      boolean ok = quarto.disponibilidadeQuarto(codQuarto, inicio, fim);
      if ( ok ) {
        ok = quarto.reservaQuarto(cpf, codQuarto);
      }
      Boolean b = new Boolean(ok);
      return b;
    }

}
```

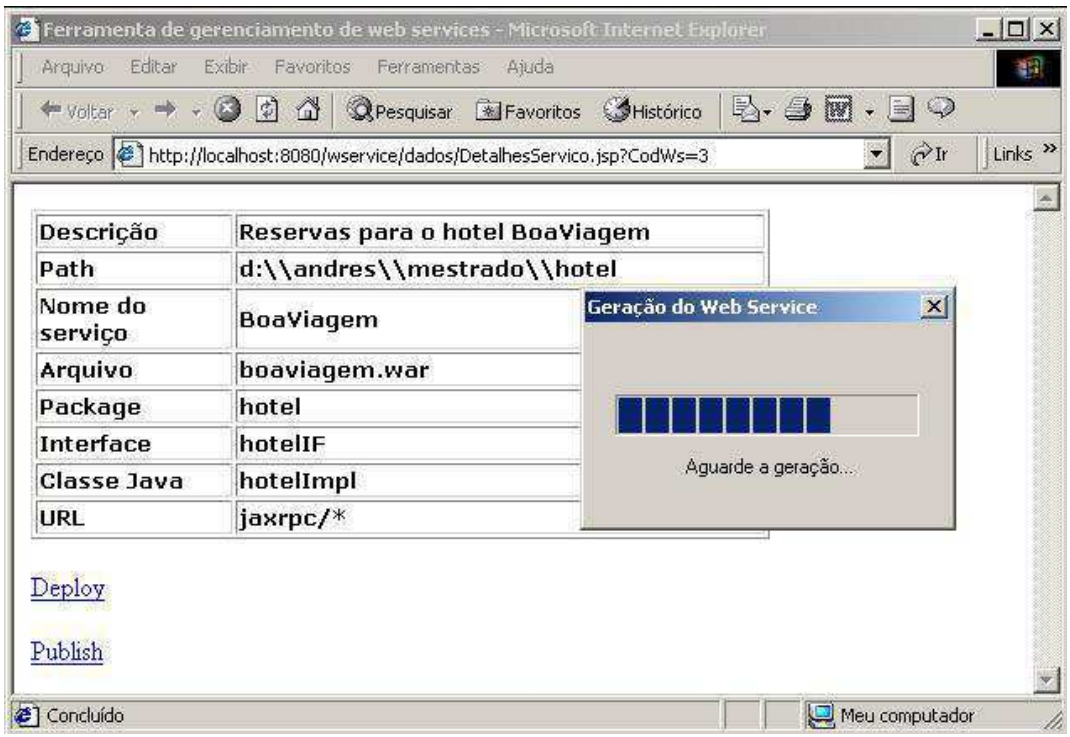

Figura 25 – Geração dos artefatos para publicação do web service do hotel BoaViagem

Depois de editar e compilar as classes, devemos adicionar um novo *web service* preenchendo os dados mostrados no JSP da Figura 12. Podemos ver na Figura 25 o momento

em que estão sendo geradas as classes *ties* e o arquivo WAR. Esta tarefa vai permitir que a aplicação possa ser publicada como serviço.

No final do processo de publicação do *web service*, o arquivo WAR é copiado para o diretório de aplicações web do Tomcat. Este nome, que é definido pelo campo "`nome do arquivo de deploy`", será usado juntamente com o campo `URL`, para compor a URL de chamada do serviço. Para verificar se o serviço está publicado devemos digitar no browser a seguinte URL: `http://localhost:8080/boaviagem/jaxrpc`. Se o serviço foi publicado com sucesso, o browser a ser mostrado deverá ser ter um aspecto como o da Figura 26.

Até este ponto, o serviço está disponível no servidor web, mas ainda não foi publicado no provedor de serviços. Preenchendo os dados da organização, mostrados na Figura 19, podemos publicar o *web service* como um registro UDDI. Antes de solicitar a publicação, os cadastros de usuários e de provedores de serviços devem ter sido digitados na ferramenta de gerenciamento.

A partir do ponto que o web service foi colocado no provedor de serviços, usuários conectados a Internet podem encontra-lo usando qualquer ferramenta de procura de registros UDDI, isto é, ele está pronto para receber solicitações de aplicações cliente.

Figura 26 – Publicação do web service de reserva do hotel BoaViagem

O arquivo WSDL gerado pela ferramenta `xrpcc` para o *web service* do hotel pode ser encontrado digitando a URL `http://localhost:8080/boaviagem/jaxrpc?WSDL`. Este endereço deve ser usado na geração das classes *stubs* quando for implementar a aplicação cliente. A seguir temos o documento WSDL gerado para o *web service* do hotel.

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

<definitions name="BoaViagemService"
             targetNamespace="http://boaviagem.org/wsdl"
             xmlns:tns="http://boaviagem.org/wsdl"
             xmlns="http://schemas.xmlsoap.org/wsdl/"
             xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
             xmlns:ns2="http://schemas.xmlsoap.org/soap/encoding/"
             xmlns:xsd="http://www.w3.org/2001/XMLSchema"
             xmlns:ns3="http://boaviagem.org/types">
  <types>
    <schema targetNamespace="http://boaviagem.org/types"
            xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
            xmlns:tns="http://boaviagem.org/types"
            xmlns:soap-enc="http://schemas.xmlsoap.org/soap/encoding/"
            xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
            xmlns="http://www.w3.org/2001/XMLSchema">
      <complexType name="ArrayOfstring">
        <complexContent>
          <restriction base="soap-enc:Array">
            <attribute ref="soap-enc:arrayType" wsdl:arrayType="string[]"/>
          </restriction></complexContent></complexType></schema></types>
  <message name="disponibilidade">
    <part name="Date_1" type="xsd:dateTime"/>
    <part name="Date_2" type="xsd:dateTime"/></message>
  <message name="disponibilidadeResponse">
    <part name="result" type="ns3:ArrayOfstring"/></message>
  <message name="inserirCliente">
    <part name="String_1" type="xsd:string"/>
    <part name="String_2" type="xsd:string"/>
    <part name="String_3" type="xsd:string"/></message>
  <message name="inserirClienteResponse">
    <part name="result" type="ns2:boolean"/></message>
  <message name="quartos"/>
  <message name="quartosResponse">
    <part name="result" type="ns3:ArrayOfstring"/></message>
  <message name="reservaQuarto">
    <part name="String_1" type="xsd:string"/>
    <part name="String_2" type="xsd:string"/>
    <part name="Date_3" type="xsd:dateTime"/>
    <part name="Date_4" type="xsd:dateTime"/></message>
  <message name="reservaQuartoResponse">
    <part name="result" type="ns2:boolean"/></message>
  <portType name="hotelIF">
    <operation name="disponibilidade">
      <input message="tns:disponibilidade"/>
      <output message="tns:disponibilidadeResponse"/></operation>
    <operation name="inserirCliente">
      <input message="tns:inserirCliente"/>
      <output message="tns:inserirClienteResponse"/></operation>
    <operation name="quartos">
      <input message="tns:quartos"/>
      <output message="tns:quartosResponse"/></operation>
    <operation name="reservaQuarto">
      <input message="tns:reservaQuarto"/>
      <output message="tns:reservaQuartoResponse"/></operation></portType>
  <binding name="hotelIFBinding" type="tns:hotelIF">
    <operation name="disponibilidade">
      <input>
        <soap:body encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
                   use="encoded" namespace="http://boaviagem.org/wsdl"/></input>
      <output>
        <soap:body encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
                   use="encoded" namespace="http://boaviagem.org/wsdl"/></output>
      <soap:operation soapAction=""/></operation>
```

```
    <operation name="inserirCliente">
      <input>
        <soap:body encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
                   use="encoded" namespace="http://boaviagem.org/wsdl"/></input>
      <output>
        <soap:body encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
                   use="encoded" namespace="http://boaviagem.org/wsdl"/></output>
      <soap:operation soapAction=""/></operation>
    <operation name="quartos">
      <input>
        <soap:body encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
                   use="encoded" namespace="http://boaviagem.org/wsdl"/></input>
      <output>
        <soap:body encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
                   use="encoded" namespace="http://boaviagem.org/wsdl"/></output>
      <soap:operation soapAction=""/></operation>
    <operation name="reservaQuarto">
      <input>
        <soap:body encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
                   use="encoded" namespace="http://boaviagem.org/wsdl"/></input>
      <output>
        <soap:body encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
                   use="encoded" namespace="http://boaviagem.org/wsdl"/></output>
      <soap:operation soapAction=""/></operation>
    <soap:binding transport="http://schemas.xmlsoap.org/soap/http" style="rpc"/>
  </binding>
  <service name="BoaViagem">
    <port name="hotelIFPort" binding="tns:hotelIFBinding">
      <soap:address location="REPLACE_WITH_ACTUAL_URL"/>
    </port>
  </service>
</definitions>
```

6.2 Implementação da interface cliente

A agência de turismo BoaTur deseja implementar uma aplicação em Java que possa fazer chamadas aos procedimentos remotos, disponibilizados pelo hotel BoaViagem, por meio de um *web service*.

A interface será feita utilizando JSPs para que possa ficar disponível na Internet. As funções básicas da aplicação serão:

- Consultar os quartos existentes no hotel bem como os seus recursos e preços;
- Consultar a disponibilidade de quartos para um determinado período;
- Inserir ou consultar clientes;
- Reservar um quarto do hotel durante um determinado período.

Para permitir que a aplicação da BoaTur acesse os procedimentos remotos é necessário gerar as classes *stubs*, com base no documento WSDL do *web service*. Este documento é usado pelo arquivo de configuração, passado como parâmetro para a ferramenta `xrpcc`, que é

quem realmente gera as classes. Todo este processo foi automatizado pela ferramenta de gerenciamento de forma a diminuir o tempo de desenvolvimento.

Preenchendo os campos da Figura 22, mostrada na Seção 5.7, é que damos início à criação de classes *stubs* da aplicação cliente. A Figura 27 permite visualizar o momento que as classes estão sendo geradas pela ferramenta de gerenciamento.

Depois da geração das classes *stubs* é que podemos construir programas Java que permitem a chamada aos procedimentos remotos. A classe `Reserva`, mostrada a seguir, faz a chamada ao procedimento remoto `reservaQuarto` que vai tentar executar uma reserva para um determinado quarto do hotel BoaViagem.

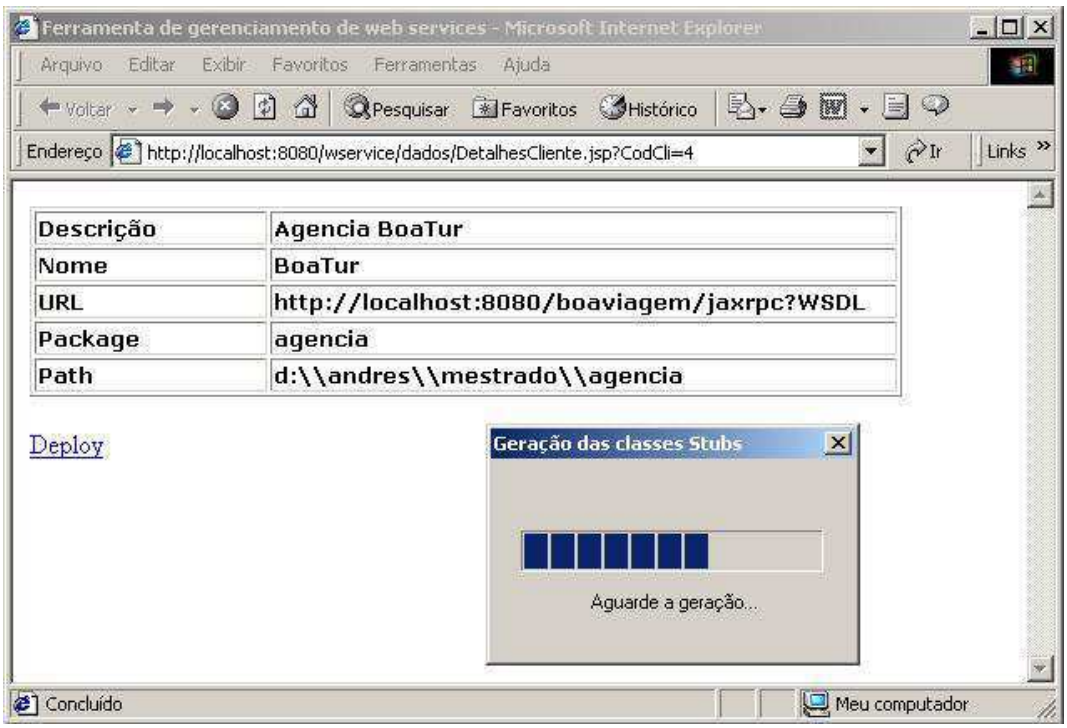

Figura 27 – Geração das classes *stubs* pela ferramenta de gerenciamento

```
package boatur;

import boatur.agencia.hotelIFPort_Stub;
import boatur.agencia.hotelBoaViagem_Impl;
import java.lang.Boolean;

public class Reserva {

  hotelIFPort_Stub stub = null;

  public void Rerserva() throws Exception {
    try {
```

```
      stub = (hotelIFPort_Stub)(new hotelBoaViagem_Impl().gethotelIFPort());
      stub._setProperty( javax.xml.rpc.Stub.ENDPOINT_ADDRESS_PROPERTY,
                         "http://localhost:8080/boaviagem/jaxrpc/hotelIF");
    } catch (Exception ex) {
      throw new Exception(ex.getMessage());
    }
  }

  public boolean Confirmar(String cpf, String codQuarto,
                           Date inicio, Date fim) throws Exception {
    try {
      Boolean ok = stub.reservaQuarto(cpf, codQuarto, inicio, fim);
      return ok.booleanValue();
    } catch (Exception ex) {
      throw new Exception(ex.getMessage());
    }
  }

}
```

Ao clicar no botão *Confirmar Reserva*, do JSP mostrado na Figura 28, a aplicação da agência BoaTur vai criar uma instância da classe `Reserva` e chamar o método `Confirmar`. A resposta do método indicará o sucesso da operação de reserva do quarto.

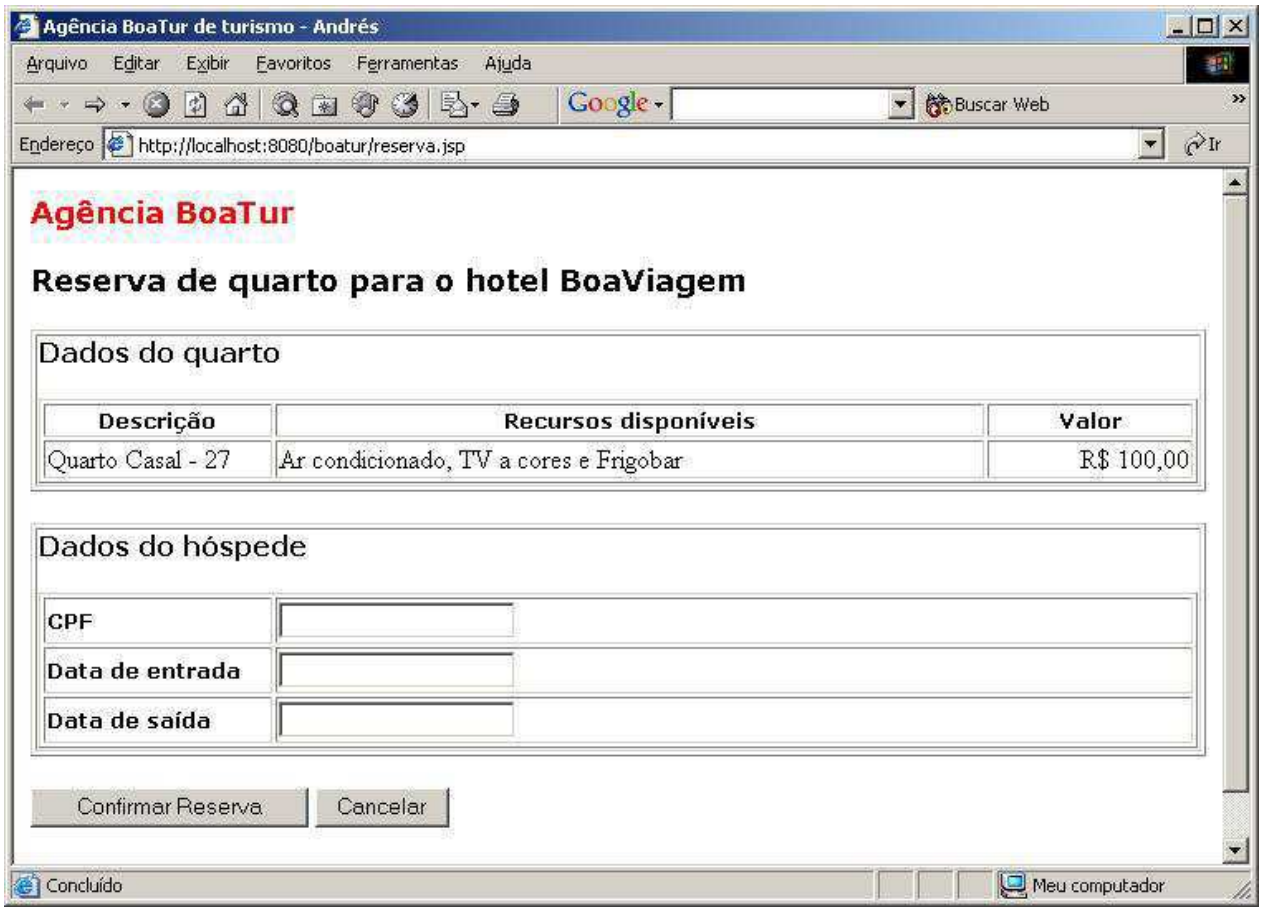

Figura 28 – Reserva de quartos na agência de turismo BoaTur

6.3 Comparativo de desenvolvimento com e sem ajuda da ferramenta

Para verificar a eficiência da ferramenta de gerenciamento, foi desenvolvido o mesmo *web service* do hotel BoaViagem sem ajuda da ferramenta, isto é, usando somente o pacote JWSDP.

Um dos parâmetros que devem ser analisados é o tempo de desenvolvimento e publicação do web service. A Tabela 4 mostra os tempos gastos no *web service* do hotel usando a ferramenta de gerenciamento.

Tarefa	6.3.1.1.1.1.1.1 Tempo gasto
1. Criação da interface Java `hotelIF`	5 minutos
2. Criação da classe Java `hotelImpl`	15 minutos
3. Compilação e cópia do diretório com os arquivos .class	1 minuto
4. Preenchimento e cadastro dos dados do serviço	1 minuto
5. Geração dos artefatos de publicação do *web service*	2 minutos
6. Publicação do serviço no servidor web	1 minuto
7. Preenchimento e cadastro dos dados da organização e do serviço	2 minutos
8. Publicação do serviço no provedor de serviços	1 minuto

Tabela 4 – Tempo de desenvolvimento do web service com ajuda da ferramenta

Como podemos ver na Tabela 4, o tempo total gasto até a publicação do serviço nos provedores de serviços, foi de vinte e oito minutos. Desse total, vinte minutos foram gastos com as tarefas 1 a 3, onde a ferramenta de gerenciamento não ajuda o desenvolvedor. Esses vinte e um minutos representam 75,0% do tempo total, enquanto que para as tarefas onde a ferramenta ajuda no processo de desenvolvimento – etapas 4 a 7 – o percentual é de 25,0%.

A Tabela 5 mostra as tarefas e o tempo gasto para cada uma delas, quando um web service foi criado e publicado sem a ajuda da ferramenta de gerenciamento. As tarefas 1 a 3 foram feitas somente uma vez e usadas nos dois casos. Dessa forma, o tempo gasto é idêntico tanto na Tabela 4 quanto na Tabela 5.

Tarefa	Tempo gasto
1. Criação da interface Java `hotelIF`	5 minutos
2. Criação da classe Java `hotelImpl`	15 minutos
3. Compilação e cópia do diretório com os arquivos .class	1 minuto
4. Criação do arquivo de configuração (config.xml)	5 minutos
5. Criação do *deployment descriptor* (web.xml)	8 minutos
6. Geração dos artefatos de publicação do web service	3 minutos
7. Geração do arquivo WAR	2 minutos
8. Publicação do serviço no servidor web	1 minuto
9. Publicação do serviço no provedor de serviços	11 minutos

Tabela 5 – Tempo de desenvolvimento do web service sem ajuda da ferramenta

Embora algumas tarefas sejam distintas, o produto final – *web service* publicado no servidor web e no provedor de serviços – será o mesmo ao completar todas as etapas. Podemos perceber que o tempo total sem a ajuda da ferramenta subiu para 51 minutos. Neste caso, o percentual para as etapas 1 a 3 caiu para 41,2%, enquanto que para o restante das etapas subiu para 58,8%.

A subida de tempo total de 28 para 51 minutos representa um aumento de 82,1%. Porém, considerando somente o tempo onde a ferramenta auxilia o desenvolvimento teremos: 7 minutos para as etapas com ajuda da ferramenta e 30 minutos quando usamos somente o pacote JWSDP, isto representa um aumento de 328,5% no tempo de desenvolvimento. Portanto, com este exemplo, podemos provar que usando a ferramenta de gerenciamento temos um grande ganho de produtividade na criação e publicação de *web services*.

Na Figura 29 temos um gráfico que ilustra o tempo de desenvolvimento para as tarefas em função do tempo. Podemos perceber que, como foi dito anteriormente, até o vigésimo primeiro minuto, etapas 1 a 3, existe uma sobreposição dos tempos. Da etapa quatro em diante é que temos uma diferença entre os tempos, que ao final é de 23 minutos.

Um detalhe que não pode ser percebido no exemplo é que a criação e publicação do *web service*, sem a ajuda da ferramenta, só pode ser feita depois que os detalhes técnicos das tecnologias são bem conhecidos e dominados. Usando a ferramenta de gerenciamento, o

desenvolvedor não precisa conhecer as tecnologias que estão envolvidas; a única exigência é que saiba criar programas na linguagem Java.

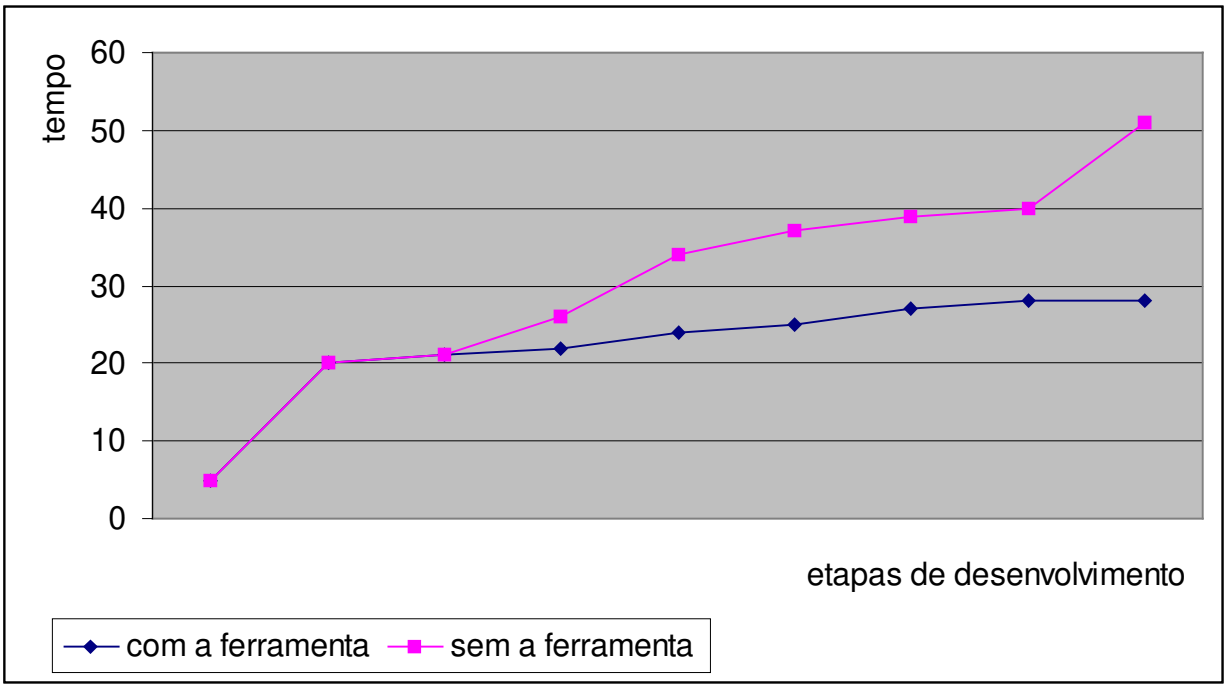

Figura 29 – Tempo de desenvolvimento do web service do hotel BoaViagem

Esconder os detalhes de implementação e aumentar a produtividade torna a ferramenta uma peça indispensável para a criação de *web services*. Os desenvolvedores precisam preocupar-se apenas com a lógica do negócio, ao invés de ter que dominar UDDI, SOAP, WSDL e outras tecnologias.

Capítulo 7

Conclusões

7.1 Avaliação e satisfação dos requisitos

Inicialmente o trabalho levou ao estudo das principais tecnologias envolvidas para a criação de *web services*, depois a proposta foi para criar uma ferramenta que ajudasse no desenvolvimento e publicação dos mesmos. Foram detectados alguns dos problemas existentes nas soluções disponíveis atualmente no mercado. O objetivo era fazer um levantamento de requisitos para a criação de uma ferramenta que ajudasse no desenvolvimento.

O pacote da Sun Microsystems – JWSDP – foi escolhido para servir como base para a ferramenta de gerenciamento. Foram estudadas as APIs que compõem o JWSDP com o objetivo de aprender como usá-las. Além disso, teve que ser verificado como é o funcionamento interno do pacote JWSDP e como ele trabalha para conseguir realizar as operações mais comuns de *web services*.

Mais uma vez foram detectados vários pontos onde a ferramenta de gerenciamento poderia ajudar os desenvolvedores específicos do JWSDP. Esses pontos foram incorporados na ferramenta sempre com o objetivo de tornar o desenvolvimento mais produtivo.

Os requisitos levantados na Seção 3.4 estão listados a seguir:

f) Armazenar dados do provedor de serviços para permitir uma rápida publicação do serviço;
g) Gerenciar os *web services* desenvolvidos para permitir uma fácil manutenção;
h) Enviar mensagens SOAP sem a necessidade de criação de programas específicos;
i) Gerenciar serviços cliente que fazem acesso a *web services* publicados;

j) esconder do desenvolvedor todos os detalhes técnicos da criação e publicação de *web services*.

O item (a) foi incorporado na ferramenta de gerenciamento e melhorou consideravelmente o tempo de publicação do serviço nos provedores de serviços. Além disso, o desenvolvedor não precisa conhecer os detalhes da tecnologia UDDI e nem cadastrar manualmente o serviço.

Os dados do *web service*, que são armazenados pela ferramenta, permitem que qualquer mudança nas regras de negócio possa ser facilmente transferida para o *web service*. O item (b) é atendido pela ferramenta uma vez que manutenções nos serviços são simples de realizar.

Existem duas formas de enviar mensagens SOAP para um *web service*: montar o arquivo XML manualmente ou criar um programa em Java. O item (c) propõe enviar mensagens sem que o usuário crie um programa específico. A ferramenta resolveu esse problema da seguinte forma: o usuário preenche somente os dados da mensagem e diz qual o seu destino. Mais uma vez a ferramenta de gerenciamento esconde os detalhes de implementação, desta vez da tecnologia SOAP.

O item (d) foi implementado pela ferramenta. Para que serviços cliente possam fazer chamadas a procedimentos remotos usando RPC, é necessário criar classes que realizam a comunicação com o lado servidor. A criação destas classes é transparente para o usuário da ferramenta, que deve preocupar-se somente com as regras de negócio.

O item (e) foi satisfeito pela ferramenta ao longo dos outros itens. Os detalhes técnicos, das tecnologias envolvidas no uso de *web services*, foram escondidos dos desenvolvedores sempre com o objetivo de aumentar a produtividade dos mesmos.

Como podemos ver pelos parágrafos anteriores, os requisitos propostos para a ferramenta foram alcançados. Entretanto, vários pontos ainda podem ser melhorados na ferramenta de gerenciamento, como por exemplo: independência do banco de dados e de plataforma. Embora a linguagem Java ajude a resolver estes problemas, a ferramenta de

gerenciamento ainda está atrelada ao sistema operacional Windows e ao banco de dados Oracle.

7.2 Problemas enfrentados

É evidente que qualquer tecnologia nova vai trazer vários problemas, o mais comum deles é a imaturidade. Da data de lançamento do pacote JWSDP, no final de janeiro de 2002, até o final do mês de junho de 2002 saíram quatro versões, sempre com grandes mudanças. Embora as APIs quase não tenham sofrido alterações estruturais, a maneira como as ferramentas trabalhavam mudava de uma versão para outra.

Em duas oportunidades, depois de muitas tentativas sem sucesso de rodar programas um pouco além dos triviais, a instalação de uma versão mais nova do servidor web fez com que os programas funcionassem. Os nomes das organizações para a consulta de registros UDDI, feita com a API JAXR, deixaram de aparecer em uma das versões, este problema foi igualmente resolvido com a instalação da nova versão da API.

Outra dificuldade foi a grande mudança de paradigma. Os conceitos de XML para troca de informação, SOAP para envio de mensagens usando o protocolo HTTP, WSDL para a definição dos web services e UDDI para a sua publicação, tiveram que ser profundamente estudados para dar o embasamento teórico da dissertação.

A falta de integração do pacote JWSDP é um dos seus pontos fracos. Uma dificuldade adicional para a ferramenta foi conseguir a integração entre as APIs para dar uma melhor funcionalidade aos recursos disponíveis.

7.3 Trabalhos futuros

A tecnologia de *web services* ainda está engatinhando e muitas são as opções de trabalhos futuros. Neste trabalho a abordagem de *web services* foi feita com J2SE (Java 2 Standard Edition), porém ele poderia ser feito utilizando toda a robustez da plataforma J2EE (Java 2 Enterprise Edition), com a utilização de EJB (Enterprise Java Beans).

Um outro ponto que poderia ser explorado é o fato da API JAX-RPC suportar apenas alguns tipos da linguagem Java. Tipos mais complexos, como `Collection`, não podem ser usados como retorno de uma chamada a um procedimento remoto, o que faz diminuir o leque de opções do desenvolvedor. Um trabalho futuro poderia ser a criação de interfaces que pudessem mapear as classes Java, baseado na sua definição, para os tipos XML de forma a permitir fazer a serialização e envio dos dados.

Outro trabalho poderia ser o de tornar a ferramenta de gerenciamento um plug-in para uma IDE Java. Algumas mudanças deveriam ser realizadas: o armazenamento dos dados não poderia ser feito em um banco de dados, a entradas de dados que atualmente são através de JSPs passariam a ser realizadas com o uso de componentes, publicação e registros UDDI também devem ser baseados em componentes.

Embora o pacote JWSDP permita que aplicações cliente possam descobrir e chamar métodos remotos em tempo de execução através de DII, a ferramenta de gerenciamento não utiliza este recurso. Um trabalho futuro poderia descobrir *web services*, baseado em consultas UDDI disponíveis na ferramenta, e armazenar os dados para permitir a chamada a procedimentos remotos usando DII, isto é, sem a geração das classes *stubs* no lado cliente.

Referências bibliográficas

[1] Bansal, Sonai, The Web at your (machine's) service. Disponível em: http://www.javaworld.com/javaworld/jw-09-2001/jw-0928-smsservice_p.html. Acessado em 29 de junho de 2002.

[2] Ort, Ed. Java Web Services Developer Pack. Disponível em: http://developer.java.sun.com/developer/technicalArticles/WebServices/WSPack/. Acessado em 29 de junho de 2002.

[3] Kao, James. Developer's Guide to Building XML-based Web Services. Disponível em: http://www.theserverside.com/resources/articles/WebServices-Dev-Guide/article.html. Acessado em 29 de junho de 2002.

[4] Sun Microsystems. Web Services Made Easier – The Java APIs and Architectures for XML. Disponível em: http://java.sun.com/xml/webservices.pdf. Acessado em 29 de junho de 2002.

[5] Jewell, Tyler e Chappell, David, UDDI - Universal Description, Discovery, and Integration. Disponível em: http://www.onjava.com/pub/a/onjava/excerpt/jws_6/index1.html. Acessado em 29 de junho de 2002.

[6] MacRoibeaird, Sean. Developer Connection: UDDI. Disponível em: http://wwws.sun.com/software/xml/developers/uddi/. Acessado em 29 de junho de 2002.

[7] Boubez, Toufic, et al. UDDI Data Structure Reference V1.0. Disponível em: http://www.uddi.org/pubs/DataStructure-V1.00-Open-20000930.html. Acessado em 29 de junho de 2002.

[8] Box, Don, et al. Simple Object Access Protocol (SOAP) 1.1. Disponível em: http://www.w3.org/TR/2000/NOTE-SOAP-20000508. Acessado em 29 de junho de 2002.

[9] Conolly, Dan. Naming and Addressing: URIs, URLs, ... Disponível em: http://www.w3.org/Addressing/. Acessado em 29 de junho de 2002.

[10] Christensen, Erik. et al. Web Services Description Language (WSDL) 1.1. Disponível em: http://www.w3.org/TR/2001/NOTE-wsdl-20010315. Acessado em 29 de junho de 2002.

[11] Sun Microsystems. The Java Web Service Tutorial. Disponível em: http://java.sun.com/webservices/docs/1.0/tutorial/index.html. Acessado em 29 de junho de 2002.

[12] E. Whitehead, et al "XML Media Types", RFC2376. Disponível em: http://www.normos.org/ietf/rfc/rfc2376.txt. Acessado em 29 de junho de 2002.

[13] Feller, John. IBM Web Services ToolKit – A showcase for emerging web services technologies. Disponível em: http://www-3.ibm.com/software/solutions/webservices/wstk-info.html. Acessado em 29 de junho de 2002.

[14] IBM. The WSTK 3.0 Tutorial. Disponível em: http://www6.software.ibm.com/developerworks/education/ws-wstk30/index.html. Acessado em 29 de junho de 2002.

[15] Suoboda, Zdenek. Creating a web service in 30 minutes. Disponível em: http://www.theserverside.cm/resources/articles/Systinet-web-services-part-1/article.html. Acessado em 29 de junho de 2002.

[16] Systinet Corp. Introduction to Web Services. 2002. Disponível em: http://www.systinet.com/download/5178f49dd2ab5455f917280d301783a0/wp_Introduction_to_Web_Services.pdf. Acessado em 29 de junho de 2002.

[17] Systinet Corp. WASP 4.0 Technical Overview. 2002. Disponível em: http://www.systinet.com/download/5178f49dd2ab5455f917280d301783a0/wp_WASP4_Technical_Overview.pdf. Acessado em 29 de junho de 2002.

[18] Cape Clear Software. CapeStudio 3 Technical Overview. 2002. Disponível em: http://www.capeclear.com/products/whitepapers/CSTechnicalOverview.pdf. Acessado em 29 de junho de 2002.

[19] Bea Systems Inc. Web Services and the BEA WebLogic E-Business Platform. Disponível em: http://www.bea.com/products/weblogic/server/paper_webservices.pdf. Acessado em 29 de junho de 2002.

[20] Rodoni, Jennifer. JAXM Client Programming Model. Disponível em: http://developer.java.sun.com/developer/technicalArticles/xml/introJAXMclient. Acessado em 29 de junho de 2002.

[21] Gupta, Arun. Getting Started with JAX-RPC. Disponível em: http://developer.java.sun.com/developer/technicalArticles/WebServices/getstartjaxrpc. Acessado em 29 de junho de 2002.

[22] Sharma, Rahul. Java API for XML-based RPC (JAX-RPC): A Primer. Disponível em: http://java.sun.com/xml/jaxrpc/primerarticle.html. Acessado em 29 de junho de 2002.

[23] Larman, Graig. Utilizando UML e padrões: uma introdução à análise e ao projeto orientado a objetos. 1ª ed. Porto Alegre, Bookman, 2000.

[24] George, Reese. JDBC e Java: Programação para Banco de Dados. 2ª ed. Rio de Janeiro, Berkeley, 2001.

[24] Horstmann, Cay S. et al Core Java 2 vol. 1 – Fundamentos, 1ª ed. Rio de Janeiro, Makron Books, 2000.

[25] Horstmann, Cay S. Core Java 2 vol. 2 – Recursos Avançados, 1ª ed. Rio de Janeiro, Makron Books, 2000.