Universidade Federal Fluminense
Instituto de Computação
Departamento de Ciência da Computação

Gabriel Affonso Baims
João Arthur Beekhuizen Nogueira

Gerador Automático de Stubs para Web Services em JavaScript

Niterói-RJ
2010

GABRIEL AFFONSO BAIMS
JOÃO ARTHUR BEEKHUIZEN NOGUEIRA

GERADOR AUTOMÁTICO DE STUBS PARA WEB SERVICES EM JAVASCRIPT

Dissertação apresentada ao Departamento de Ciência da Computação da Universidade Federal Fluminense como parte dos requisitos para obtenção do Grau de Bacharel em Ciência da Computação.

Orientador: Prof. Dr. LUIZ CARLOS CASTRO GUEDES

Niterói-RJ
2010

GABRIEL AFFONSO BAIMS
JOÃO ARTHUR BEEKHUIZEN NOGUEIRA

GERADOR AUTOMÁTICO DE STUBS PARA WEB SERVICES EM JAVASCRIPT

Dissertação apresentada ao Departamento de Ciência da Computação da Universidade Federal Fluminense como parte dos requisitos para obtenção do Grau de Bacharel em Ciência da Computação.

Aprovada em julho de 2010.

BANCA EXAMINADORA

Prof. Dr. LUIZ CARLOS CASTRO GUEDES - Orientador
UFF

Prof. CARLOS ALBERTO SOARES RIBEIRO
UFF

Prof. Dr. ISABEL L. CAFEZEIRO
UFF

Niterói-RJ
2010

Agradecimentos

Gostariamos de agradecer primeiramente a Deus por nos proteger sempre que necessário e nos dar sabedoria para enfrentar os contratempos que a vida nos propôs. Em segundo lugar ao Professor Luiz Carlos Castro Guedes pelos ensinamentos e pela dedicação à orientação deste trabalho. Queremos também agradecer com igual fervor a nossa família pelo apoio incondicional, pela participação e pelos conselhos indispensáveis. Gostariamos ainda de agradecer as nossas namoradas pela paciência e compreensão pelos finais de semana perdidos para que pudessemos cumprir com nossas obrigações como alunos. Por fim, queremos agradecer aos amigos que caminharam ao nosso lado durante todo esse trajeto, companhias indispensáveis das madrugadas de estudos e das festas a cada período concluído com sucesso.

Lista de Figuras

Lista de Tabelas

Listings

Sumário

Resumo

Atualmente existem geradores de stubs para Web Services, voltados para aplicações web e feitos com linguagens como Java e frameworks como .NET. Essas por sua vez têm como principal foco aplicações mais robustas e de grande porte. Com isso, desenvolvedores menos experientes que desejem utilizar Web Services em aplicações web mais simples enfrentam um grau de dificuldade maior adaptando a sua aplicação para uma nova plataforma. O intuito desse projeto é desenvolver um gerador automático de stubs para Web Services em JavaScript, que é uma linguagem nativa da web, dando assim mais uma opção para o desenvolvedor criar aplicativos que se comuniquem com Web Services, com a finalidade de simplificar a construção de sites de menor porte.

Palavras-chave:
Gerador Automático de Stubs, Web Service, JavaScript.

Abstract

Currently exist stubs generators for Web Services, directed to web applications and made with languages like Java and frameworks like .NET, whose main focus are applications more robust and huge. Thus, less experienced developers who want to use Web Services in a simple web aplications, face a greater degree of difficulty to adapt their application to a new platform. The purpose of this project is develop an automatic stubs generator for JavaScript, which is a web native language, thus providing an additional option for the developers to create applications that communicate with Web Services, in order to simplify the construction of smaller sites.

Keywords:
Automatic Stubs Generator, Web Service, JavaScript.

Capítulo 1

Introdução

1.1 Descrição do problema

Construir um gerador automático de stubs para Web Services em JavaScript através da tradução de um arquivo WSDL, ou seja, um gerador de algoritmos em JavaScript que provêm, localmente, a abstração à chamada de métodos em uma máquina remota. Estes métodos são definidos pelo arquivo WSDL, que descreve serviços, operações e estruturas de dados disponíbilizados na máquina remota através do XML.

1.2 Motivação

A falta de um gerador de stubs voltado para sistemas menores faz com que um desenvolvedor que queira utilizar uma ferramenta de trabalho mais simples para montar uma página web, que utilize os serviços expostos por Web Services, seja obrigado a utilizar ferramentas mais complexas, que têm o principal foco em sistemas maiores. O intuito desse projeto é desenvolver um gerador automático de stubs para Web Services em JavaScript, que é uma linguagem nativa da web, dando assim mais uma opção para o desenvolvedor criar aplicativos que se comuniquem com Web Services, com a finalidade de simplificar a construção de sites de menor porte.

1.3 Proposta de trabalho

Estudar e entender o que são, para que servem e como se comportam os Web Services. Analisar como é feita a troca de mensagens entre o cliente e o servidor e a partir disso definir uma estratégia de montagem de um stub a partir da tradução de um WSDL, utilizando a linguagem Java para construir stubs em JavaScript.

Capítulo 2

Fundamentos

2.1 Web Services

Web Services é uma solução utilizada na integração de sistemas. É uma tecnologia baseada na exposição de serviços por meio da troca de mensagens padronizadas, fazendo com que sistemas construídos em plataformas distintas sejam compatíveis. Os Web Services trocam informações por meio de mensagens, normalmente em formato XML e empacotadas pelo protocolo SOAP (Simple Object Access Protocol), para que seja possível que cada aplicação tenha a sua própria "linguagem" desde que se faça uma tradução para uma linguagem universal, o formato XML.

Essencialmente, os Web Services fazem com que os recursos da aplicação do software estejam disponíveis sobre a rede de uma forma normalizada. Existe uma grande tendência à utilização dessa tecnologia, pois possibilita que diferentes aplicações comuniquem-se entre si e utilizem recursos diferentes.

Os Web Services são identificados por um URI (Unique Resource Identifier), descritos e definidos usando WSDL (Web Service Description Language), que é um arquivo escrito em XML que contém toda a descrição dos serviços expostos. Um dos motivos que tornam os Web Services atrativos é o fato de este modelo ser baseado em tecnologias padrões, em particular XML e HTTP.

2.2 Padrão

A arquitetura de Web Services é baseada em três padrões: SOAP, WSDL e UDDI.

2.2.1 SOAP (Simple Object Access Protocol)

Define os mecanismos de como é feito a chamada de Web Services e como os dados são retornados. Os SOAP clients podem chamar métodos dos SOAP services passando objetos no formato XML. SOAP é um protocolo leve para a troca de informações em um ambiente distribuído descentralizado. É um protocolo baseado em XML dividido em três partes: Um envelope que define a estrutura que descreve o que está dentro da mensagem e como processá-la; um conjunto de regras de codificação para expressar a definição dos tipos de dados e a convenção para as chamadas remotas de procedimentos (RPC). Veremos mais a

fundo esse padrão no capítulo III.

2.2.2 WSDL (Web Service Description Language)

Descreve a interface externa dos Web Services permitindo com que desenvolvedores possam criar clientes capazes de realizar chamadas remotas aos serviços oferecidos. É um XML para descrever os serviços de rede como um conjunto de parâmetros operacionais nas mensagens. As operações e mensagens são descritas abstratamente e então ligadas a um protocolo de rede e a um formato de mensagem concreto para definir um parâmetro. Parâmetros concretos relacionados são combinados em parâmetros abstratos (serviços). O WSDL é extensível para permitir descrição dos parâmetros e suas mensagens, independentemente de qual formato da mensagem ou protocolo de rede é usado para se comunicar. Veremos mais a fundo esse padrão no capítulo III.

2.2.3 UDDI (Universal Discovery, Description and Integration)

É um protocolo para web baseado em registros e que contém informações sobre os Web Services incluindo o endereço dos arquivos WSDL e dos serviços ativos. Um registro para um Web Service provê informações técnicas para permitir que um desenvolvedor possa criar aplicativos clientes capazes de se ligarem ao serviço e chamarem os métodos.

2.3 Integração de Sistemas

O conceito de Web Services tem sido cada vez mais utilizado nos ambientes empresariais. A arquitetura orientada a serviços (SOA) ganha cada vez mais espaço como ferramenta de integração de sistemas. Uma abordagem muito utilizada de SOA seria o barramento de serviços em que aplicações expõem seus serviços no barramento e as aplicações clientes chamam o barramento para utilizar esses serviços ao invés de fazer uma chamada a cada aplicação servidora. Esse tipo de abordagem possibilita a criação de workflows que utilizam mais de um serviço, formando um macro serviço, o que possibilita até mesmo a união de serviços de diferentes aplicações servidoras.

2.4 Vantagens

No modelo de Web Services, cada sistema atua como um componente independente na arquitetura de integração. Todas as interfaces, transformações de dados e comunicações entre componentes são baseados em padrões abertos e vastamente adotados, independentes de fornecedores e plataformas. Com isso a utilização, de Web Services como ferramenta de integração de sistemas nas empresas se torna mais atraente. Os principais pontos que justificam a adoção desse tipo de integração são:

- **Simplicidade**: é mais simples de se implementar que as soluções tradicionais.
- **Padrões abertos**: utilizam padrões abertos como HTTP, SOAP, UDDI, em invés de tecnologias proprietárias;

- **Flexibilidade**: alterações nos componentes são muito mais simples para o sistema como um todo do que alterações nos adaptadores tradicionais;
- **Custo**: as soluções tradicionais são muito mais caras;
- **Escopo**: cada sistema pode ser tratado de maneira individual, já que para integrá-lo basta implementar uma camada que o encapsule. Na abordagem tradicional, todos os sistemas devem ser tratados ao mesmo tempo, já que farão parte da mesma solução monolítica de integração.

Capítulo 3

Tecnologias Utilizadas

3.1 Introdução

Neste capítulo e no capítulo 4 serão abordados conceitos e fundamentos das tecnologias e padrões utilizados durante o projeto, caso o leitor já possua um conhecimento sobre os assuntos abordados é aconselhavel que comece a leitura no capítulo 5 no qual começa a descrição do trabalho proposto.

3.2 XML

XML é o acrônimo de Extensible Markup Language e foi desenhada para transportar e armazenar dados. Assim como o HTML o XML também é uma linguagem de marcação, com a diferença de não ter sido criada para exibir dados. As tags do XML não são pré-definidas tendo que ser definidas pelo próprio desenvolvedor, pode-se observar então que é uma linguagem que foi desenhada para ser auto descritiva.

Apesar de ser uma linguagem, o XML não faz nenhuma ação, apenas estrutura, armazena e transporta informação. No exemplo 3.1, podemos ver um bilhete da Dani para o Tom:

```
<Bilhete>
  <Para>Tom</Para>
  <De>Dani</De>
  <Titulo>Lembrar</Titulo>
  <Mensagem>Não se esqueça de mim essa semana!</Mensagem>
</Bilhete>
```

Código Fonte 3.1: Bilhete XML

Podemos perceber que ele é muito descritivo, pois possui informações de remetente e destinatário, um título e o corpo da mensagem. São apenas informações encapsuladas por tags que precisam de algum pedaço de software para serem enviadas, recebidas ou exibidas.

Apesar do XML ser apenas um texto simples, aplicações que sabem lidar com essa tecnologia são capazes de lidar com as tags de forma especial. O significado funcional das tags depende da natureza da aplicação.

É importante perceber que o XML não é um substituto para o HTML e sim um complemento, pois como vimos acima, o XML é utilizado para transportar dados enquanto o HTML é utilizado para exibí-los.

O XML simplifica o compartilhamento e transporte de dados, pois como é um texto simples, aplicações com tipos de dados incompatíveis podem acessar os dados armazenados em um arquivo XML e compreender o que está representado neste arquivo. Com isso, reduz-se drasticamente a complexidade de se trocar dados entre sistemas, inclusive por meio da internet.

3.2.1 Estrutura

O XML possui uma estrutura de árvore, começando no nó raiz e passando pelos ramos até as folhas. O código 3.2 ilustra um arquivo XML com um nó raiz <bilhete>, quatro nós que são filhos diretos do nó raiz e um nó <Observacao> que é um sub-filho do nó raiz e filho direto do nó <mensagem>.

```
<Bilhete>
  <Para>Tom</Para>
  <De>Dani</De>
  <Titulo>Lembrete</Titulo>
  <Mensagem>Não se esqueça de mim essa semana!
    <Observacao>Me deve um jantar</Observacao>
  </Mensagem>
</Bilhete>
```

Código Fonte 3.2: Bilhete XML com hierarquia

Todo documento XML precisa ter elemento raiz, que é o pai de todos os outros elementos. Os termos Pai, Filho e Irmão são usados para descrever o relacionamento entre os elementos. Todo Pai possui um filho, e filhos no mesmo nível da árvore são considerados irmãos e todos os elementos podem ter filhos.

A figura 3.1 exemplifica a relação entre os elementos de um documento XML no qual todos podem ter atributos ou texto, assim como em um documento HTML.

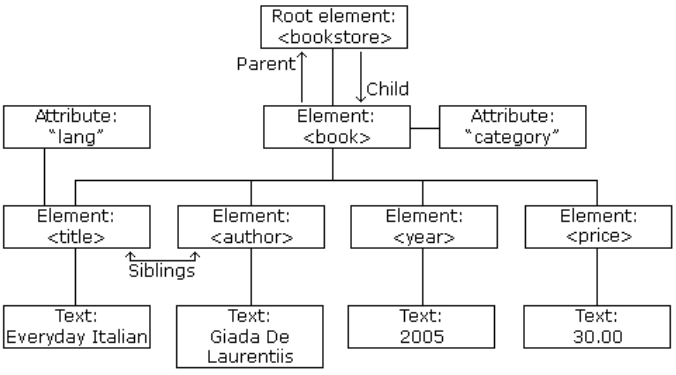

Figura 3.1: Relação de elementos XML

3.2.2 Sintaxe

As regras de sintaxe de um documento XML são relativamente simples como pode ser visto abaixo:

- Todo documento XML precisa ter elemento raiz.
- Todo elemento XML precisa ter uma tag fechando.
- As Tags XML são Sensíveis à Caixa, ou seja, <ABC> é interpretado diferentemente de <abc>.
- Os elementos precisam estar aninhados corretamente.
- Os valores dos atributos precisam estar entre aspas .
- Caracteres como "<" precisam ser substituídos por referências de entidades, para evitar erros de interpretação do analisador sintático.

Referenciais de Entidades	Caracteres
<	<
>	>
&	&
&após;	'
"	"

Tabela 3.1: Tabela de Caracteres especiais

- O comentário é similar com o HTML <!– Comentário –>

3.2.3 Simple Types

Um tipo simples é um elemento XML que contém apenas texto. Ele não pode conter nenhum outro elemento nem atributo. Entretanto, a restrição de apenas texto é enganadora. O texto pode ser de diferentes tipos, dentre eles os definidos pelo XML Schema (boolean, string, date, etc.), ou pode ser definido pelo usuário. Pode ainda conter restrições para que um tipo de dado esteja contido dentro de um limite ou requerer que o dado seja compatível com um padrão específico. A sintaxe para definir um elemento simples é vista no código 3.3:

```
<xs:element name="xxx" type="yyy"/>
```

Código Fonte 3.3: Sintaxe elemento Simples

Em que XXX é o nome do elemento e YYY é o tipo de dado do elemento. O código 3.4 mostra alguns elementos XML e suas respectivas definições como simple type:

```
<xs:element name="lastname"type="xs:string"/>
<xs:element name="age" type="xs:integer"/>
<xs:element name="dateborn" type="xs:date"/>

<lastname>Refsnes</lastname>
<age>36</age>
<dateborn>1970-03-27</dateborn>
```

Código Fonte 3.4: Elemento XML SimpleType

Os elementos simples podem ter valores default ou fixos especificados na declaração. No código 3.5 vemos o elemento "color" e o elemento University. Como pode-se observar, se não for passado nenhum valor para o elemento "color", o mesmo será vermelho. Já o elemento "Univertsity"será sempre UFF.

```
<xs:element name="color" type="xs:string" default="red"/>
<xs:element name="University" type="xs:string" fixed="UFF"/>
```

Código Fonte 3.5: Elemento SimpleType com valores

Os tipos simples não podem ter atributos, pois se tiverem, passam a ser considerados tipos complexos. Porém um atributo é sempre declarado como um tipo simples. Além das opções de valores fixos e padrão, atributos também podem ser opcionais ou obrigatórios. No código 3.6, vemos um atributo configurado como obrigatório, caso queira que seja opcional, é só não usar a cláusula "use".

```
<xs:attribute name="lang" type="xs:string" use="required"/>
```

Código Fonte 3.6: Atributo XML Obrigatório

Quando um elemento ou atributo XML possui um tipo de dado definido, acaba tendo uma restrição sobre o conteúdo do elemento ou atributo. Se um elemento XML for do tipo xs:date e contiver a string Hello World, o elemento não vai ser válido. Com XML Schemas, pode-se definir restrições próprias para os elementos e atributos XML. As restrições são usadas para definir valores aceitáveis para um elemento ou atributo XML. O código 3.7 define um elemento chamado de "age" com restrição. O valor da idade não pode ser menor que 0 nem maior que 120.

```
<xs:element name="age">
  <xs:simpleType>
    <xs:restriction base="xs:integer">
      <xs:minInclusive value="0"/>
      <xs:maxInclusive value="120"/>
    </xs:restriction>
  </xs:simpleType>
</xs:element>
```

Código Fonte 3.7: Elemento SimpleType com restrição

Para limitar o conteúdo de um elemento XML para um conjunto de valores aceitos, pode-se usar uma restrição de enumeração. O código 3.8 ilustra o elemento “car” com uma restrição de aceitar apenas os valores: Audi, Golf, BMW.

```
<xs:element name="car">
  <xs:simpleType>
    <xs:restriction base="xs:string">
      <xs:enumeration value="Audi"/>
      <xs:enumeration value="Golf"/>
      <xs:enumeration value="BMW"/>
    </xs:restriction>
  </xs:simpleType>
</xs:element>
```

Código Fonte 3.8: Elemento SimpleType com restrição por conjunto

Para limitar o conteúdo de um elemento XML, para definir uma série de números ou letras que possam ser utilizadas, pode-se usar a restrição pattern. O código 3.9, vemos algumas das formas como pode ser usado o pattern.

```
<!-- Só pode conter uma letra e minúscula-->
<xs:element name="letter">
  <xs:simpleType>
    <xs:restriction base="xs:string">
      <xs:pattern value="[a-z]"/>
    </xs:restriction>
  </xs:simpleType>
</xs:element>

<!-- Contém 3 letras maiúsculas-->
<xs:element name="initials">
  <xs:simpleType>
    <xs:restriction base="xs:string">
      <xs:pattern value="[A-Z][A-Z][A-Z]"/>
    </xs:restriction>
  </xs:simpleType>
</xs:element>

<!-- Contém 3 letras, seja maiúscula ou minuscule -->
<xs:element name="initials">
  <xs:simpleType>
    <xs:restriction base="xs:string">
      <xs:pattern value="[a-zA-Z][a-zA-Z][a-zA-Z]"/>
    </xs:restriction>
  </xs:simpleType>
</xs:element>

<!-- Só aceita um dos tres valores (x, y ou Z)-->
```

```
<xs:element name="choice">
  <xs:simpleType>
    <xs:restriction base="xs:string">
      <xs:pattern value="[xyz]"/>
    </xs:restriction>
  </xs:simpleType>
</xs:element>

<!-- Aceita 5 dígitos de 0 a 9 -->
<xs:element name="prodid">
  <xs:simpleType>
    <xs:restriction base="xs:integer">
      <xs:pattern value="[0-9][0-9][0-9][0-9][0-9]"/>
    </xs:restriction>
  </xs:simpleType>
</xs:element>

<!-- Aceita zero ou mais ocorrencias de letras minúsculas de a  z -->
<xs:element name="letter">
  <xs:simpleType>
    <xs:restriction base="xs:string">
      <xs:pattern value="([a-z])*"/>
    </xs:restriction>
  </xs:simpleType>
</xs:element>
```

Código Fonte 3.9: Elemento SimpleType com restrição por padrão

3.2.4 Complex Types

Um tipo complexo é um elemento que contém outros elementos e/ou atributos. Existem quatro tipos de elementos complexos:

- Elementos vazios.
- Elementos que contém outros elementos.
- Elementos que contém apenas texto.
- Elementos que contém tanto outros elementos quanto texto.

No código 3.10, temos um exemplo de um elemento XML complexo contendo apenas outros elementos.

```
<employee>
  <firstname>John</firstname>
  <lastname>Smith</lastname>
</employee>
```

Código Fonte 3.10: Elemento ComplexType de elementos

OS elementos complexos podem ser definidos em um schema XML de duas maneiras:

1. O elemento pode ser declarado diretamente pelo nome dele como podemos ver no exemplo de empregado:

```
<xs:element name="employee">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="firstname" type="xs:string"/>
      <xs:element name="lastname" type="xs:string"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>
```

Código Fonte 3.11: Declaração de Elemento ComplexType por nome

Ao se utilizar o método descrito no código 3.11, apenas o elemento “employee” pode usar o complex type especificado. Note que os elementos filhos, ”firstname”e ”lastname”, estão cercados pela tag <sequence>, o que significa que devem aparecer na ordem em que foram declarados.

2. O elemento pode ter um tipo de atributo que faz referência a um tipo complexo para usá-lo:

```
<xs:element name="employee" type="personinfo"/>

<xs:complexType name="personinfo">
  <xs:sequence>
    <xs:element name="firstname" type="xs:string"/>
    <xs:element name="lastname" type="xs:string"/>
  </xs:sequence>
</xs:complexType>
```

Código Fonte 3.12: Declaração de Elemento ComplexType por referência

Utilizando esse tipo de declaração, outros elementos poderão fazer referência ao mesmo tipo complexo como podemos ver no código 3.13.

```
<xs:element name="employee" type="personinfo"/>
<xs:element name="student" type="personinfo"/>
<xs:element name="member" type="personinfo"/>

<xs:complexType name="personinfo">
  <xs:sequence>
    <xs:element name="firstname" type="xs:string"/>
    <xs:element name="lastname" type="xs:string"/>
  </xs:sequence>
```

```
</xs:complexType>
```

Código Fonte 3.13: Declaração de Elementos referênciando um ComplexType

Pode-se ainda basear um elemento complexo em outro já existente e adicionar mais alguns elementos.

```
<xs:element name="employee" type="fullpersoninfo" />

<xs:complexType name="personinfo">
  <xs:sequence>
    <xs:element name="firstname" type="xs:string" />
    <xs:element name="lastname" type="xs:string" />
  </xs:sequence>
</xs:complexType>

<xs:complexType name="fullpersoninfo">
  <xs:complexContent>
    <xs:extension base="personinfo">
      <xs:sequence>
        <xs:element name="address" type="xs:string" />
        <xs:element name="city" type="xs:string" />
        <xs:element name="country" type="xs:string" />
      </xs:sequence>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>
```

Código Fonte 3.14: Declaração de Elemento ComplexType baseado em outro

Além de elementos com apenas outros elementos como vimos nos primeiros exemplos desta seção, existem elementos com apenas texto ou atributos (simpleContent) e para isso é necessário definir uma extenção ou uma restrição dentro de um elemento simpleContent como podemos ver no código 3.15:

```
<xs:element name="somename">
  <xs:complexType>
    <xs:simpleContent>
      <xs:extension base="basetype">
        ....
        ....
      </xs:extension>
    </xs:simpleContent>
  </xs:complexType>
</xs:element>

OR
```

```
<xs:element name="somename">
  <xs:complexType>
    <xs:simpleContent>
      <xs:restriction base="basetype">
        ....
        ....
      </xs:restriction>
    </xs:simpleContent>
  </xs:complexType>
</xs:element>
```

Código Fonte 3.15: Declaração de Elemento ComplexType simpleContent

Após adicionar o elemento <simpleContent> em volta do conteúdo, precisa-se definir se é uma extensão ou uma restrição para se expandir ou limitar a base do tipo simples para o elemento.

Existe ainda o tipo complexo misto, que contém tanto texto como outros elementos. O código 3.17 demonstra um elemento XML misto e a declaração do mesmo.

```
<letter>
  Dear Mr.<name>John Smith</name>.
  Your order <orderid>1032</orderid>
  will be shipped on <shipdate>2001-07-13</shipdate>.
</letter>
```

Código Fonte 3.16: Instância de um Elemento ComplexType misto

```
<xs:element name="letter">
  <xs:complexType mixed="true">
    <xs:sequence>
      <xs:element name="name" type="xs:string"/>
      <xs:element name="orderid" type="xs:positiveInteger"/>
      <xs:element name="shipdate" type="xs:date"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>
```

Código Fonte 3.17: Declaração de Elemento ComplexType misto

O tipo complexo vazio, por sua vez, não contém texto nem outros elementos, mas sim atributos que definem seu comportamento. O código 3.18 demonstra um elemento XML vazio e a declaração do mesmo.

```
<xs:element name="imagem">
  <xs:complexType>
    <xs:attribute name="path" type="xs:string" use="required"/>
    <xs:attribute name="formato" type="xs:string" use="optional"/>
```

```
  </xs:complexType>
</xs:element>
```

Código Fonte 3.18: Declaração de Elemento ComplexType vazio

Os elementos podem ser controlados com indicadores sobre como eles serão usados no documento. Existem sete indicadores divididos em três tipos:

- Indicadores de Ordem:
  - All
  - Choice
  - Sequence
- Indicadores de Ocorrência:
  - maxOccurs
  - minOccurs
- Indicadores de Grupo:
  - Group name
  - attributeGroup name

Indicadores de Ordem

Os indicadores de ordem, como o nome sugere, indicam a ordem em que um elemento pode aparecer.

O indicador <all> especifica que os elementos filhos podem aparecer em qualquer ordem e que cada elemento filho só pode ocorrer uma vez.

```
<xs:element name="person">
  <xs:complexType>
    <xs:all>
      <xs:element name="firstname" type="xs:string"/>
      <xs:element name="lastname" type="xs:string"/>
    </xs:all>
  </xs:complexType>
</xs:element>
```

Código Fonte 3.19: Declaração de Elemento ComplexType all

O indicador <choice> especifica que tanto um elemento filho quanto o outro podem ocorrer.

```
<xs:element name="person">
  <xs:complexType>
    <xs:choice>
      <xs:element name="employee" type="employee"/>
```

```
      <xs:element name="member" type="member"/>
    </xs:choice>
  </xs:complexType>
</xs:element>
```

Código Fonte 3.20: Declaração de Elemento ComplexType choice

Já o indicador <sequence> especifica que os elementos filhos devem aparecer em uma ordem específica.

```
<xs:element name="person">
   <xs:complexType>
    <xs:sequence>
      <xs:element name="firstname" type="xs:string"/>
      <xs:element name="lastname" type="xs:string"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>
```

Código Fonte 3.21: Declaração de Elemento ComplexType sequence

Indicadores de Ocorrência

O indicadores de ocorrência são usados para definir com que frequência um elemento pode ocorrer.

O indicador <maxOccurs> especifica o número máximo de vezes em que o elemento pode ocorrer:

```
<xs:element name="person">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="full_name" type="xs:string"/>
      <xs:element name="child_name" type="xs:string" maxOccurs="10"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>
```

Código Fonte 3.22: Declaração de ComplexType com <maxOccurs>

O indicador <minOccurs> especifica o número mínimo de vezes que um elemento pode ocorrer:

```
<xs:element name="person">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="full_name" type="xs:string"/>
      <xs:element name="child" type="xs:string"
      maxOccurs="10" minOccurs="0"/>
    </xs:sequence>
  </xs:complexType>
```

```
</xs:element>
```

Código Fonte 3.23: Declaração de ComplexType com <minOccurs>

O exemplo acima, indica que o elemento “child” pode ocorrer no mínimo zero e no máximo dez vezes dentro do elemento “person”.

Indicadores de Grupo

Indicadores de grupo são usados para definir conjuntos de elementos relacionados.

O elemento “group” é definido com uma declaração como a vista no código 3.24. É preciso definir um indicador de ordem (all, choice, sequence) dentro da declaração do grupo. O exemplo 3.24 define um grupo chamado “persongroup”, que define um grupo de elementos que precisam ocorrer em uma sequência exata:

```
<xs:group name="persongroup">
  <xs:sequence>
    <xs:element name="firstname" type="xs:string"/>
    <xs:element name="lastname" type="xs:string"/>
    <xs:element name="birthday" type="xs:date"/>
  </xs:sequence>
</xs:group>
```

Código Fonte 3.24: Declaração de elemento Group

Depois de se definir um grupo, pode-se fazer referencia a ele em outra definição.

```
<xs:group name="persongroup">
  <xs:sequence>
    <xs:element name="firstname" type="xs:string"/>
    <xs:element name="lastname" type="xs:string"/>
    <xs:element name="birthday" type="xs:date"/>
  </xs:sequence>
</xs:group>

<xs:element name="person" type="personinfo"/>

<xs:complexType name="personinfo">
  <xs:sequence>
    <xs:group ref="persongroup"/>
    <xs:element name="country" type="xs:string"/>
  </xs:sequence>
</xs:complexType>
```

Código Fonte 3.25: Declaração de elemento Group com referência

3.3 JavaScript

JavaScript é a linguagem script mais popular da internet e funciona com a maioria dos navegadores, tal qual Internet Explorer, Firefox, Chrome, Opera e Safari. O JavaScript foi desenhado para adicionar interatividade para as páginas HTML e, por ser uma linguagem de programação mais "leve", é anexada diretamente na página HTML. Outra vantagem do JavaScript é que, por ser uma linguagem interpretada, não precisa ser compilada para executar.

O JavaScript pode ser usado para diversas funções no ambiente web, como por exemplo: dar aos web designers uma ferramenta de programação com uma sintaxe muito simples, o que permite a pessoas que não estão familiarizadas com as linguagens de programação tradicionais colocarem uma certa inteligência em suas páginas web. O JavaScript também é capaz de reagir a eventos, como por exemplo, quando uma página terminar de carregar ou quando um usuário clicar em um elemento HTML. Outras funcionalidades são: validar dados, detectar o navegador do usuário, criar cookies (salvar e manter informações no computador do usuário), etc.

A tag HTML <script> é usada para inserir um código Javascript em uma página HTML e pode ser declarada tanto dentro do <head> (tag que define um cabeçalho de uma página web com informações sobre o documento) quanto dentro do <body> (tag que define o todo o conteúdo que será exibido no navegador) ou até mesmo nos dois ao mesmo tempo. Outra alternativa para se usar um JavaScript é utilizando um arquivo externo, como no caso de se querer executar o mesmo script em várias páginas diferentes. Ao invés de se escrever o mesmo script em todas elas, faz-se um arquivo (.js) e apenas declara em cada página que irá usar o script.

Assim como outras linguagens de script, o JavaScript nada mais é do que uma sequência de comandos a serem executados pelo navegador na ordem em que são escritos. O JavaScript pode usar também o conceito de bloco para indicar que os comandos pertencentes a ele devem ser executados em conjunto. Outros detalhes da linguagem é que ela é case sensitive e possui a opção de comentar linhas de código ou blocos inteiros.

Além de todas essas funcionalidades, o JavaScript é uma linguagem orientada a objetos, o que permite que o programador crie seus próprios objetos e tipos. Mais que isso, pode ser usada para uma série de funções como: validação de dados, animações, mapeamento de imagens e eventos por tempo. O JavaScript também pode ser usada em conjunto com o XML gerando uma nova forma de se utilizar padrões existentes. Essa nova forma é denominada AJAX e será estudada mais a fundo na próxima seção.

3.4 AJAX - Asynchronous JavaScript and XML

Ajax não é uma nova linguagem de programação, mas sim uma nova maneira de utilização do que já existe. É considerada por alguns a arte de se trocar dados com um servidor de forma assíncrona e atualizar partes da página web sem recarregar a página inteira.

Na figura 3.2 segue um exemplo funcional do AJAX:

Apesar de parecer que funciona como o antigo HTML, o AJAX possui um poder muito maior, pois é capaz de permitir que o usuário continue trabalhando na página web enquanto o servidor recebe

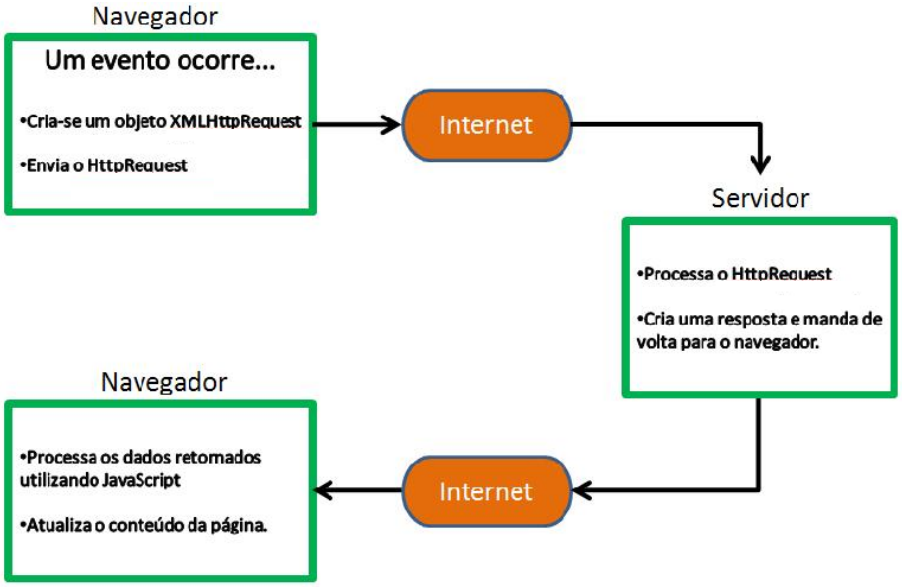

Figura 3.2: Exemplo Funcional do AJAX

algum tipo de requisição e a processa. Após processar internamente a requisição feita, o servidor devolve uma mensagem ao cliente e se esta envolver dados, os mesmos são transportados em XML. Ao chegar no navegador do cliente, esta resposta é processada e interpretada pelo JavaScript, que consegue dizer claramente o que aquele documento em XML representa e o transforma em formato HTML para que o navegador possa exibir o que for necessário.

3.4.1 Criando um Objeto XMLHttpRequest

Todos os navegadores modernos suportam o objeto XMLHttpRequest, com exceção ao Internet Explorer 5 e ao Internet Explorer 6 que usam um ActiveXObject.

O objeto XMLHttpRequest é utilizado para a troca de dados com o servidor de sem a necessidade de recarregar a página. Ao utilizar um Web Service, por exemplo, o desenvolvedor da pagina web faria uma chamada local a procedimentos e à funções que existissem no servidor e enviaria a requisição XML empacotada pelo SOAP via HttpRequest no formato esperado pelo servidor.

No código 3.26, podemos ver a sintaxe para a criação de um objeto XMLHttpRequest dependendo da versão do navegador utilizada pelo usuário.

```
getXmlHttp: function() {
    try {
        if (window.XMLHttpRequest) {
            var req = new XMLHttpRequest();
            if (req.readyState == null) {
                req.readyState = 1;
                req.addEventListener("load",
                                     function() {
                                     req.readyState = 4;
                                     if (typeof req.onreadystatechange == "function")
```

```
                                        req.onreadystatechange();
                                    },
                                    false);
            }
            return req;
        }
        if (window.ActiveXObject) {
            var parsers = ["Msxml2.XMLHTTP.5.0",
                           "Msxml2.XMLHTTP.4.0",
                           "MSXML2.XMLHTTP.3.0",
                           "MSXML2.XMLHTTP",
                           "Microsoft.XMLHTTP"];
           for(var i = 0; i < parsers.length; i++) {
                try {
                    return new ActiveXObject(parsers[i]);
                }
                catch (ex) {};
            }
            throw new Error("Impossível encontrar um Xml parser instalado");
        }
    }
    catch (ex) {}
        throw new Error("Este navegador não suporta objectos XmlHttp");
        }
```

Código Fonte 3.26: Sintaxe do objeto XMLHttpRequest por navegador

Para enviar uma requisição ao servidor, deve-se utilizar os métodos open() e send() do objeto XMLHttpRequest.

```
xmlhttp.open("GET","ajax_info.txt",true);
xmlhttp.send();
```

Código Fonte 3.27: Enviar requisição ao servidor

Ao se construir uma requisição, deve-se informar o seu tipo, a url e se deverá ser assíncrona ou não. O primeiro parâmetro pode ser do tipo GET ou POST, vamos explicar a diferença entre eles mais adiante. A url deverá conter o local do arquivo no servidor e para dizer que a requisição é assíncrona deve-se passar o último parâmetro como “true”, caso contrário, se for “false”, a requisição será considerada síncrona.

O método GET é mais simples e mais rápido que o método POST, e pode ser usado na maioria dos casos. Entretanto, em alguns casos listados abaixo, deve-se sempre utilizar o método POST.

- Arquivos em cache não são uma opção (atualizar um arquivo ou banco no servidor).
- Enviar uma grande quantidade de dados para o servidor (POST não tem limitações de tamanho.)
- Enviar uma entrada do usuário que possa conter caracteres desconhecidos. POST é mais robusto e seguro do que GET.

Para que se receba uma resposta do servidor, deve-se utilizar as propriedades “responseText” ou “responseXML” do objeto XMLHttpRequest. Como o nome mesmo já nos induz a pensar, o “responseText” é usado quando a resposta esperada é um texto simples e é passada como “string”. Já o “responseXML” que é o nosso objeto de estudo, é capaz de receber como resposta do servidor um XML, o que nos permite fazer um parser desse XML para tratar e pegar a informação que estamos esperando.

```
<html>
<head>
<script type="text/javascript">
```

```
function loadXMLDoc()
{
  xmlhttp=new XMLHttpRequest();
  xmlhttp.onreadystatechange=function(){
  if (xmlhttp.readyState==4 && xmlhttp.status==200)
    {
    xmlDoc=xmlhttp.responseXML;
    var txt="";
    x=xmlDoc.getElementsByTagName("ARTIST");
    for (i=0;i<x.length;i++)
      {
      txt=txt + x[i].childNodes[0].nodeValue + "<br />";
      }
    document.getElementById("myDiv").innerHTML=txt;
    }
  }
xmlhttp.open("GET","cd_catalog.xml",true);
xmlhttp.send();
}
```

```
</script>
</head>

<body>

<h2>My CD Collection:</h2>
<div id="myDiv"></div>
<button type="button" onclick="loadXMLDoc()">Get my CD collection</button>

</body>
</html>
```

Código Fonte 3.28: Exemplo de chamada com XMLHttpRequest

O código 3.28, mostra uma página web que deseja receber o nome dos artistas da coleção de cds do usuário. Após receber a resposta em XML, o script busca a tag ARTIST, e para cada artista atribui o valor desse nó, que seria o nome do artista, à variável txt para depois exibir na página.

```
xmlDoc=xmlhttp.responseXML;
    var txt="";
    x=xmlDoc.getElementsByTagName("ARTIST");
    for (i=0;i<x.length;i++)
      {
      txt=txt + x[i].childNodes[0].nodeValue + "<br />";
      }
    document.getElementById("myDiv").innerHTML=txt;
```

Código Fonte 3.29: Exemplo para exibir resposta em html

Para perceber que uma resposta do servidor está pronta, utilizamos o evento "onreadystatechange", que é disparado a cada mudança de estado da propriedade "readyState". A tabela 3.2 exemplifica como são tratadas as propriedades do objeto "XMLHttpRequest".

Propriedade	Descrição
onreadystatechange	Guardar uma função para ser chamada toda vez que a propriedade readyState mudar.
readyState	Guarda o status do XMLHttpRequest. Variam entre 0 e 4: 0: Requisição não inicializada. 1: Conexão com o servidor estabilizada. 2: Requisição recebida. 3: Processando Requisição. 4: Requisição terminada e resposta está Pronta.
Status	200: "OK". 404: Page not found.

Tabela 3.2: Objeto XMLHttlRequest

3.5 Java

Java é uma linguagem de programação orientada a objetos, altamente portável, podendo ser utilizada em qualquer dispositivo independente da plataforma. Para isso, basta que o dispositivo tenha uma máquina virtual Java (JVM) instalada. Pois, diferentemente das linguagens convencionais, que são compiladas para uma linguagem de máquina e por isso precisam levar em consideração o ambiente em que vão executar, o Java é compilado em tempo de execução em cima da JVM, o que permite que ele seja portável para qualquer ambiente, já que a JVM é que vai executar o código.

Outro ponto forte do Java é a garantia na hora de manipular arquivos, seja escrevendo ou lendo, de que conseguirá terminar a ação, o que permitiu com que os stubs fossem gerados para JavaScript de forma confiável.

Capítulo 4

Padrões utilizados

Existem atualmente duas versões do padrão WSDL para a descrição de Web Services, segundo o consórcio W3C: a versão 1.1, recomendada pelo W3C em março de 2003 e a mais recente, a versão 2.0 recomendada em junho de 2007.

Neste capítulo abordaremos as duas versões supracitadas de WSDL e o padrão SOAP 1.1 correlacionando os padrões utilizados para operacionalizar Web Services. Caso o leitor ja possua conhecimento sobre esses padrões é aconselhado que comece a leitura a partir do capítulo 5.

4.1 WSDL 1.1

Uma definição completa de WSDL contém toda a informação necessária para realizar uma chamada a um Web Service. Um WSDL é um documento XML que possui uma definição de elemento raiz na URL *http://schemas.xmlsoap.org/wsdl/* namespace. O schema completo de um WSDL está disponivel em *http://schemas.xmlsoap.org/wsdl/*. A definição de um elemento pode conter vários outros elementos, incluindo tipos, mensagens, PortTypes, binding e serviços.

Na versão 1.1, o WSDL era composto basicamente de sete parâmetros:

- **Types**: Um container para definição de tipos de dados usando algum sistema de tipos como o Xml Schema.
- **Message**: Uma definição abstrata dos tipo de dados que serão usados durante a comunicação.
- **Operation**: Uma definição abstrata de uma ação suportada pelo serviço.
- **PortType**: Um conjunto abstrato de operações suportadas por um ou mais pontos de saída.
- **Binding**: Definição concreta de um protocolo e da especificação de um formato de dados para um PortType particular.
- **Port**: Definição concreta de um único ponto de saída por meio da combinação de binding e endereço de rede.
- **Service**: Uma coleção de pontos de saída relacionados.

É importante observar que o WSDL não introduz uma nova linguagem de definição de tipos. O WSDL reconhece a necessidade por sistemas ricos de tipos para a descrição de formato de mensagens, e suporta o XML Schemas definitions (XSD) como o padrão do sistema. O WSDL cobre o que XML Schema não consegue, provendo um meio de agrupar as mensagens em operações e as operações em PortType. Ainda provém uma maneira de definir Bindings para cada interface e combinações de protocolos com pontos de saída.

O esquema da figura 4.1 mostra como estes sete parâmetros se relacionam entre si. As setas com um ponto representam um relacionamento do tipo "refere-se à" ou ainda "utiliza". A seta dupla indica um modificador e a seta em três dimensões indica um relacionamento do tipo "Contém". Assim, Messages usa Types e é usado por PortTypes. PortType e Binding contêm Operations e o Operations do PortType é modificado pela Operations do Binding.

Types, Messages e PortTypes são definições abstratas enquanto Bindings Services e Ports são definições concretas. Apenas Operation pode ser tanto uma definição abstrata quanto uma definição Concreta.

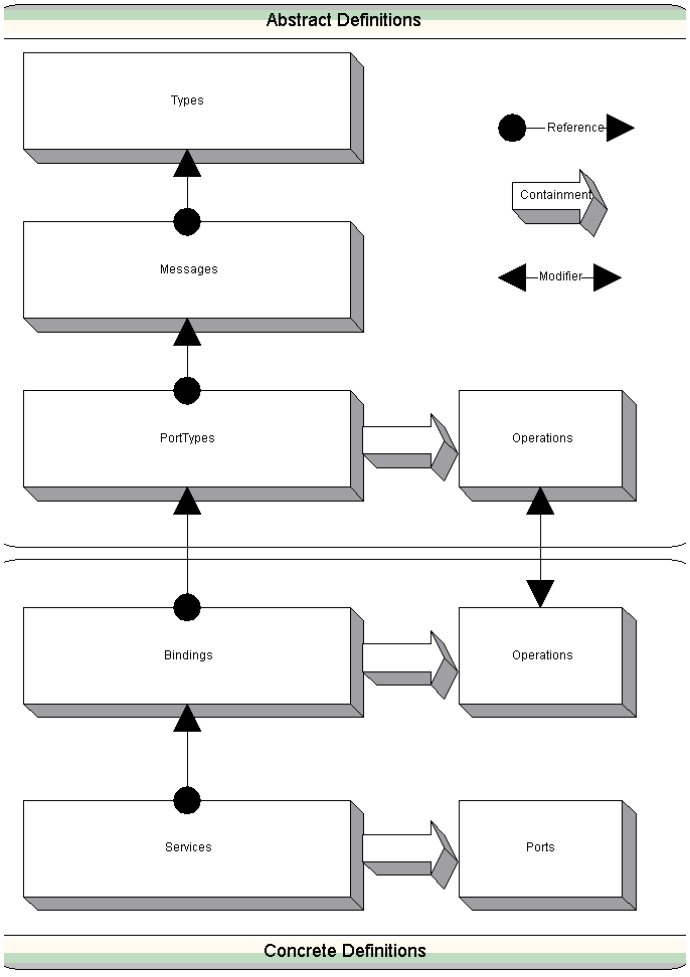

Figura 4.1: Abstract Definitions

4.1.1 Exemplo do WSDL 1.1

O código 4.1 ilustra a estrutura básica de uma definição de WSDL:

```
<!-- WSDL definition structure -->
<definitions
    name="MathService"
    targetNamespace="http://example.org/math/"
    xmlns=http://schemas.xmlsoap.org/wsdl/
>

<!-- abstract definitions -->
    <types> ...
    <message> ...
    <portType> ...

<!-- concrete definitions -->
    <binding> ...
    <service> ...

</definition>
```

Código Fonte 4.1: Estrutura básica WSDL 1.1

Note que é necessário especificar um namespace destino para a definição de um WSDL, assim como seria feito para a definição de um XML schema. Qualquer parâmetro seja nomeada na definição de um WSDL (como message, portType, etc) automaticamente se tornará parte das definições do namespace destino definidos pelo atributo targetNamespace.

Types

O elemento Types do WSDL é um container para definições de tipo XML schema. As definições de tipo inseridas nesse elemento são referenciadas no nível mais alto das definições da mensagem para definir os detalhes estruturais da mesma.

O elemento Types contém zero ou mais schemas do namespace *http://www.w3.org/2001/XMLSchema*.

Segue no código 4.2 a estrutura básica do elemento Types:

```
<definitions .... >
    <types>
        <xsd:schema .... />*
    </types>
</definitions>
```

Código Fonte 4.2: Estrutura básica elemento Types

Messages

O elemento “message” do WSDL define uma mensagem abstrata que tanto pode servir como entrada ou como saída de uma operação. O elemento “message” consiste em um ou mais elementos Part no qual cada Part é associado tanto a um Elemento, quando utiliza um estilo Document, quanto a um Type, quando utiliza um estilo RPC. o código 4.3 ilustra a estrutura básica de uma definição de mensagem:

```
<definitions ...>
    <message name="..." ... >
        <part name="..." ... >
    </message>
</definitions>
```

Código Fonte 4.3: Estrutura básica elemento Message

As mensagens e as partes precisam ser nomeadas para que possam ser referenciadas em qualquer lugar da definição do WSDL. Quando define-se um serviço estilo RPC, as partes da mensagem representam parâmetros de métodos. Neste caso, o name das partes torna-se o name de um elemento na mensagem concreta e esta estrutura é determinada pelo tipo do atributo fornecido. Quando define-se um serviço estilo document as partes referem-se a elementos XML inseridos no body.

Interfaces (portTypes)

O elemento portType do WSDL define um grupo de operações, também conhecida como interface na maioria dos ambientes. Um elemento PortType contém zero ou mais operações. A estrutura básica do portType pode ser vista no código 4.4:

```
<definitions .... >
          <portType name="nmtoken">
               <operation name="nmtoken" .... /> *
          </portType>
</definitions>
```

Código Fonte 4.4: Estrutura básica elemento portType

Cada portType deve receber um nome único para que possa ser referenciado de qualquer parte da definição do WSDL. Cada operação contém uma combinação de elementos de entrada e saída, e quando se há um elemento de saída pode-se ter ainda um elemento de falha. A ordem desses elementos define o padrão da troca de mensagens (MEP) suportada pela dada operação.

Por exemplo, um elemento de entrada seguido por um elemento de saída define uma operação de “request-response”, enquanto um elemento de saída seguido por um de entrada define uma operação de “solicit-response”. Uma operação que contém apenas o elemento de entrada define-se como uma one-way, enquanto a que contém apenas o elemento saída define-se como uma operação de notificação.

Bindings

O elemento Binding do WSDL descreve de forma concreta os detalhes do uso de um portType com um determinado protocolo. O elemento Binding contém vários elementos de extensibilidade bem como um elemento de operação do WSDL para cada operação descrita no portType. No código 4.5pode-se ver a estrutura básica do elemento Binding.

```
<wsdl:definitions .... >
    <wsdl:binding name="nmtoken" type="qname"> *

        <-- extensibility element providing binding details --> *
        <wsdl:operation name="nmtoken"> *

            <-- extensibility element for operation details --> *
            <wsdl:input name="nmtoken"? > ?
                <-- extensibility element for body details -->
            </wsdl:input>

            <wsdl:output name="nmtoken"? > ?
                <-- extensibility element for body details -->
            </wsdl:output>

            <wsdl:fault name="nmtoken"> *
                <-- extensibility element for body details -->
            </wsdl:fault>

          </wsdl:operation>
        </wsdl:binding>
</wsdl:definitions>
```

Código Fonte 4.5: Estrutura básica elemento Binding

O Binding deve ter um nome único para que possa ser consultado em outra parte na definição WSDL. Os Bindings ainda precisam especificar qual portType estão descrevendo através do atributo Type. Atualmente, os detalhes de um binding são providos utilizando elementos de extensibilidade. Esta arquitetura permite o WSDL evoluir ao longo do tempo, uma vez que qualquer elemento pode ser usado nos slots pré-definidos. A especificação WSDL disponibiliza alguns elementos bindings para descrever SOAP bindings, apesar de eles estarem em um namespace diferente. O código 4.6 mostra um binding SOAP/HTTP para o portType MathInterface:

```
<definitions
  xmlns="http://schemas.xmlsoap.org/wsdl/"
  xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
  xmlns:xs="http://www.w3.org/2001/XMLSchema"
  xmlns:y="http://example.org/math/"
  xmlns:ns="http://example.org/math/types/"
```

```
  targetNamespace="http://example.org/math/">
...

    <binding name="MathSoapHttpBinding" type="y:MathInterface">
        <soap:binding style="document"
          transport="http://schemas.xmlsoap.org/soap/http"/>
            <operation name="Add">
                <soap:operation
                  soapAction="http://example.org/math/#Add"/>
                    <input>
                        <soap:body use="literal"/>
                    </input>
                    <output>
                        <soap:body use="literal"/>
                    </output>
            </operation>
...
    </binding>
...
</definitions>
```

Código Fonte 4.6: Binding SOAP/HTTP para um portType

O elemento soap:binding indica que esse é binding SOAP 1.1, ele ainda indica o estilo padrão do serviço junto com protocolo de transporte necessário. O elemento soap:operation define o valor do cabeçalho HTTPSOAPAction para cada operação e o elemento soap:body define como as partes da mensagem vão aparecer dentro do elemento body do SOAP.

Usar o estilo document no SOAP, indica que o body conterá um documento XML e que as partes da mensagem irão especificar o elemento XML que será colocado no body. Usar o estilo RPC no SOAP indica que o body conterá uma representação XML de uma chamada de método e as partes da mensagem representarão os parâmetros para esse método.

O atributo "uses" especifica a codificação que deverá ser usada para traduzir as partes de uma mensagem abstrata em uma representação concreta.

A combinação mais comum dos atributos SOAP style/use é document/literal, pois é o que tem menos problemas de interoperabilidade.

Services

O elemento service do WSDL define uma coleção de ports ou endpoints, que expõe um binding particular. O código 4.7 ilustra a estrutura básica do elemento service.

```
<definitions .... >
    <service .... > *
        <port name="nmtoken" binding="qname"> *
            <-- extensibility element defines address details -->
        </port>
```

```
    </service>
</definitions>
```

Código Fonte 4.7: Estrutura básica elemento Service

Deve-se dar a cada porta um nome e associá-la a um binding. Então, dentro do elemento porta pode-se usar um elemento de extensibilidade para definir os detalhes específicos do endereço de um binding. O código 4.8 define um serviço MathService, que expõe o MathSoapHttpBinding na URL *http://localhost/math/math.asmx.*

```
<definitions
  xmlns="http://schemas.xmlsoap.org/wsdl/"
  xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
  xmlns:xs="http://www.w3.org/2001/XMLSchema"
  xmlns:y="http://example.org/math/"
  xmlns:ns="http://example.org/math/types/"
  targetNamespace="http://example.org/math/"
>
    ...
    <service name="MathService">
        <port name="MathEndpoint" binding="y:MathSoapHttpBinding">
            <soap:address
              location="http://localhost/math/math.asmx"/>
       </port>
    </service>
</definitions>
```

Código Fonte 4.8: Definição de um serviço

4.2 WSDL 2.0

O Web Services Description Language na Versão 2.0 (WSDL 2.0) provê um modelo e um formato XML para descrever Web Services. O WSDL 2.0 permite separar a descrição das funcionalidades abstratas oferecidas por um serviço dos detalhes concretos da descrição de um serviço, tal qual como e onde essa funcionalidade é oferecida. O WSDL 2.0 descreve um Web Service em dois estágios fundamentais, em que cada estágio, a descrição utiliza um número de construções para promover a reutilização da descrição e para separá-la, independentemente de preocupações de desing. No nível abstrato, o WSDL 2.0 descreve um Web Service em termos de mensagens que envia e recebe.

Uma operação associa um padrão de troca de mensagem com uma ou mais mensagens. O padrão de troca de mensagens identifica a sequência e a cardinalidade das mensagens enviadas e/ou recebidas bem como a quem ela está sendo logicamente enviada e/ou de quem é recebida. Uma interface agrupa operações sem qualquer compromisso com o transporte.

No nível concreto, o binding especifica o transporte e os detalhes do formato de wire para uma ou mais interfaces. Um endpoint associa um endereço de rede com um binding. Finalmente, um serviço

agrupa endpoints que implementam uma interface em comum.

4.2.1 Descrição

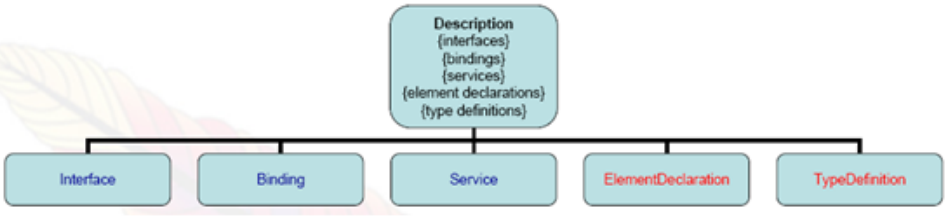

Figura 4.2: Description - Visão geral

Como pode ser visto na figura 4.2, o componente o Description é apenas um container para dois tipos de componentes de alto nível:

- Componentes WSDL (Interface, Binding, Service)
- Componentes sistemas de tipo (Element declarations, type definitions)

no código 4.9 vemos uma representação XML do componente description:

```
<xml version="1.0" encoding="utf-8" ?>
<description
    targetNamespace=...
    xmlns=http://www.w3.org/2006/01/wsdl...>
    <types>
        <xs:schema...
            <xs:elementname=...>
            <xs:complexTypename=...>
        </xs:schema>
    </types>
<interface> ... </interface>
<binding> ... </binding>
<service> ... </service>
</description>
```

Código Fonte 4.9: Representação do componente Description

O componente sistemas de tipo descreve as restrições do conteúdo da mensagem. Por padrão, essas restrições são expressas em termos de [XML Information Set], isto é. elas definem as propriedades de um elemento item de informação.

É um invólucro abstrato para algum sistema subjacente, como próprio XML schema ou outros sistemas de tipos como o RelaxNG baseado em XML ou o DTD que não utiliza XML.

O Elemento Declaration é um schema global da declaração do elemento enquanto o TypeDefinition é um schema global da definição de tipos. No código 4.10, temos uma representação XML do

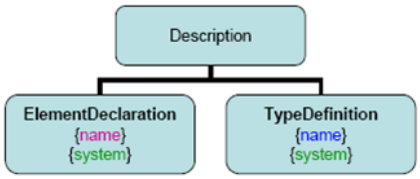

Figura 4.3: Description - Componente System

componente sistema de tipo:

```
<?xml version="1.0" encoding="utf-8" ?>
<description
  xmlns="http://www.w3.org/2006/01/wsdl"
  targetNamespace= . . . >
    <types>
        <xs:schema
          xmlns:xs="http://www.w3.org/2001/XMLSchema"
          targetNamespace="http://greath.example.com/2004/schemas/resSvc"
          xmlns="http://greath.example.com/2004/schemas/resSvc">

            <xs:element name="checkAvailability" type="tCheckAvailability"/>
            <xs:complexType name="tCheckAvailability">
                <xs:sequence>
                    <xs:element name="checkInDate" type="xs:date"/>
                    <xs:element name="checkOutDate" type="xs:date"/>
                    <xs:element name="roomType" type="xs:string"/>
                </xs:equence>
            </xs:complexType>
            <xs:element name="checkAvailabilityResponse" type="xs:double"/>
            <xs:element name="invalidDataError" type="xs:string"/>
        </xs:schema>
    </types>
...
</description>
```

Código Fonte 4.10: Representação do componente Type

Os componentes Interface, Binding, Service, Element Declaration e Type Definition estão diretamente contidos no componente Description e são referenciados como componentes de alto nível. Os componentes de alto nível do WSDL 2.0 contêm outros componentes, por exemplo: Interface Operation e EndPoint, que são referenciados como componentes aninhados. Componentes aninhados podem conter outros componentes aninhados. O componente que contém um componente aninhado é referenciado como pai desse componente. Os Componentes aninhados têm uma propriedade pai que é uma referência a seus componentes pai.

As propriedades de um componente de Descrição estão expostas a seguir:

- Interfaces (opcional) Conjunto de componentes Interface.

- Bindings (opcional) Conjunto de componentes Binding.
- Services (opcional) Conjunto de componentes Service.
- Element Declarations (opcional) Conjunto de componentes Element Declaration.
- Type Definitions (requerido) Conjunto de componentes Type Definition.

Na figura 4.4 temos um modelo dos componentes aninhados:

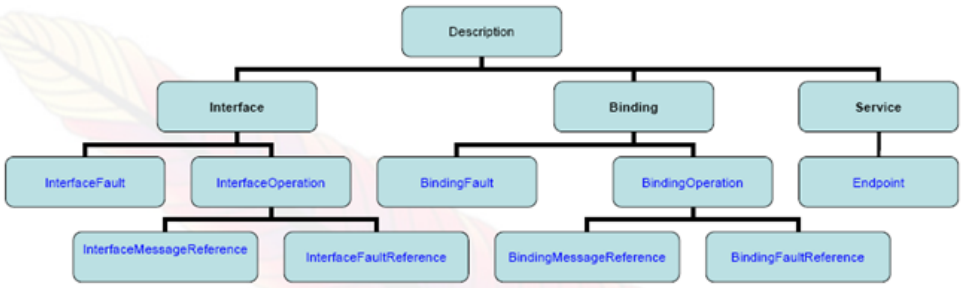

Figura 4.4: Description - Componentes

Note que eles vêm sempre abaixo de um dos componentes de alto nível (Interface, Binding, Service) e que podem conter outros componentes aninhados.

4.2.2 Componente Interface

Um componente interface descreve sequências de mensagens que um serviço envia e/ou recebe. Ele agrupa dentro de Operations, mensagens relacionadas. Um Operation é uma sequência de mensagens de entrada e saída e interface é um conjunto de Operations. Uma interface pode opcionalmente estender uma ou mais interfaces. Para evitar definições recursivas, uma interface não pode aparecer no conjunto de interfaces que ela estende, tanto diretamente quanto indiretamente. O conjunto de operações disponíveis em uma interface inclui todas as operações definidas por interfaces que ela estende direta ou indiretamente. As operações definidas diretamente em uma interface são referenciadas como operations da interface declaradas.

Na figura 4.5 vemos um modelo do componente interface e de alguns componentes aninhados e no código 4.11 a representação XML schema demonstrando os componentes em destaque no modelo.

```
<?xml version="1.0" encoding="utf-8" ?>
<description targetNamespace=...
  xmlns="http://www.w3.org/2006/01/wsd" ...>
    <types>
        <xs:schema ...>
            <xs:elementname="invalidDataError" .../>
            <xs:elementname="checkAvailability" .../>
             ...
        </xs:schema>
    </types>
    <interface name = "reservationInterface"
```

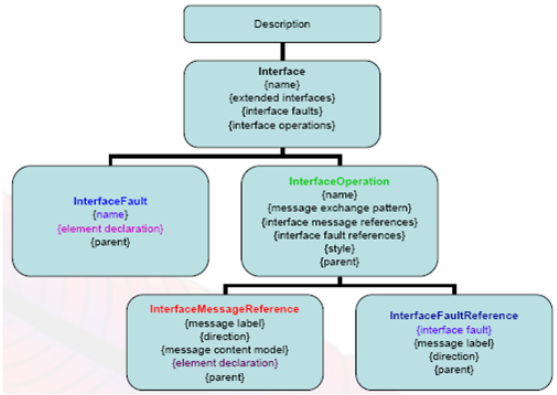

Figura 4.5: Interface - Visão Geral

```
      extends=... styleDefault=...>
        <fault name = "invalidDataFault"
          element = "ghns:invalidDataError"/>
        <operation name="opCheckAvailability"
          pattern=http://www.w3.org/2006/01/wsdl/in-out
          style="http://www.w3.org/2006/01/wsdl/style/iri">
            <input messageLabel="In"
              element="ghns:checkAvailability" />
            <output messageLabel="Out"
              element="ghns:checkAvailabilityResponse" />
            <infault ref="tns:invalidDataFault" messageLabel=In/>
            <outfault ref="tns:invalidDataFault" messageLabel="Out"/>
        </operation>
    </interface>
...
</description>
```

Código Fonte 4.11: Representação do componente Interface

4.2.3 Componente Binding

Um componente Binding descreve um formato de mensagem concreto e um protocolo de transmissão que podem ser usados pra definir um endpoint, isto é, um componente binding define os detalhes necessários para acessar um serviço.

Os componentes Binding podem ser usados para descrever tais informações em um modo reutilizável para qualquer interface ou especificamente para uma dada interface. Além disso, a informação do binding pode ser especificada em uma base por informação dentro de uma interface, além de todas as operações em uma interface.

Se um componente Binding especifica qualquer detalhe de operações específicas de binding (incluindo componente Binding Operation) ou qualquer detalhe de fault binding (incluindo componente Binding Fault), então ele deverá especificar uma interface em que o componente binding foi aplicado, assim como indicar de que interface a operação veio.

Reciprocamente, um componente binding que omite qualquer detalhe de operação específica de binding e qualquer detalhe de fault binding pode omitir especificando uma interface. Os componentes Binding que não especificam uma interface podem ser usados para especificar detalhes de operações independentes de binding para o componente Service com diferentes interfaces, isto é, tais componentes Binding são reutilizáveis em uma ou mais interfaces.

Um componente Binding que define bindings para um componente Interface precisa definir bindings para todas as operações de um componente Interface.

Na figura 4.6 vemos um modelo do componente Binding e de alguns componentes aninhados a ele e no código 4.12 a representação XML schema demonstrando os componentes em destaque no modelo.

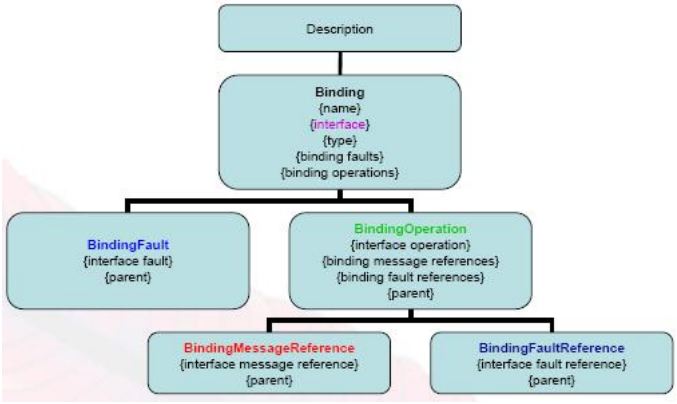

Figura 4.6: Binding - Visão Geral

```
<?xml version="1.0" encoding="utf-8" ?>
<description targetNamespace= ...
  xmlns=''http://www.w3.org/2006/01/wsdl''...>
    <types> ... </types>
    <interface name = "reservationInterface" > ... </interface>
    <binding name="reservationSOAPBinding"
      interface="tns:reservationInterface"
      type= "␣http://www.w3.org/2006/01/wsdl/soap...␣">

        <fault ref="tns:invalidDataFault"
          wsoap:code="soap:Sender"/>

        <operation ref="tns:opCheckAvailability" ...>
            <input></input>
            <output>.</output>
            <infault> . </infault>
            <outfault> . </outfault>
        </operation>
    </binding>
...
</description>
```

Código Fonte 4.12: Representação do componente Binding

4.2.4 Componente Service

Um componente Service descreve um conjunto de endpoints em que uma implantação de uma implementação particular do serviço é provida. Os endpoints, portanto, são de fato lugares alternativos em que o serviço é prestado.

Na figura 4.7, vemos um modelo do componente Service com o componente endpoint aninhado a ele e no código 4.13 a representação XML schema demonstrando os componentes em destaque no modelo.

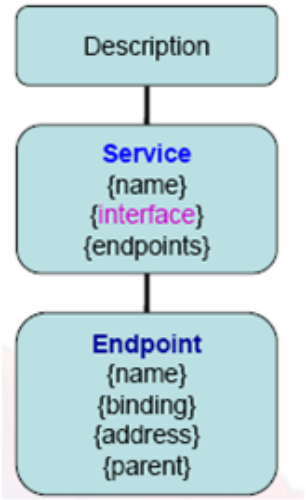

Figura 4.7: Service - Visão Geral

```
<?xml version="1.0" encoding="utf-8" ?>
<description targetNamespace=...
  xmlns=''http://www.w3.org/2006/01/wsdl''...>

    <types>...</types>
    <interface name=''reservationInterface''>
        ...
    </interface>

    <binding name=''reservationSOAPBinding''
      interface=''tns:reservationInterface''...>
      ...
    </binding>

    <service name=''reservationService''
      interface=''tns:reservationInterface''>
        <endpoint name=''reservationEndpoint''
          binding=''tns:reservationSOAPBinding''
          address=''http://greath.example.com/2004/reservation''/>
    </service>
</description>
```

Código Fonte 4.13: Representação do componente Service

4.3 Diferenças entre WSDL 1.1 e WSDL 2.0

A grande maioria de Web Services que encontramos estava escrita em WSDL versão 1.1. A primeira dificuldade apareceu quando nos deparamos com a nova versão 2.0, que apesar de ainda não ter sido massificada, já é de utilização recomendada pela W3C desde 2007. Como as duas versões apresentam diferenças significativas, ficamos em dúvida se deveríamos dividir nossas atenções para as duas ou se nos concentrávamos em uma delas. Por um lado, a versão 1.1 é ainda amplamente utilizada, enquanto a versão 2.0 nos parece ser mais enxuta, com as informações mais bem distribuídas no arquivo.

Durante os estudos, descobrimos que poderíamos adaptar qualquer arquivo WSDL 1.1 para os novos padrões do 2.0 através da api wodem apache. A seguir veremos as principais diferenças entre as duas versões de WSDL.

4.3.1 Comparação entre WSDL 1.1 e WSDL 2.0

Como podemos ver na tabela 4.1, a grande diferença entre o WSDL 1.1 e o WSDL 2.0 é que o componente message passou a ser definido dentro do componente operation.

WSDL 1.1	WSDL 2.0
<definitions>	<description>
<portType>	<interface>
<binding>	<binding>
<types>	<types>
<service>	<service>
<port>	<endpoint>
<message>	<operation>

Tabela 4.1: WSDL1.1 e WSDL2.0

Outras alterações menos significativas do ponto de vista funcional são as mudanças de nome. O componente definition passou a se chamar description, o portType passou a se chamar interface e o port a se chamar endpoint. Na figura 4.8, temos um esquema com uma correlação direta entre cada componente.

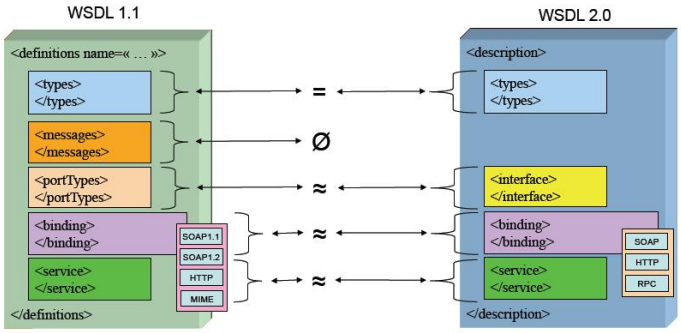

Figura 4.8: Comparativo - WSDL1.1 x WSDL2.0

Alguns pontos relevantes entre as diferenças são:

- **Sobrecarga de operações** – O WSDL 1.1 utiliza os nomes dos elementos operation, input e output, enquanto no WSDL 2.0 só precisa ser nomeado pelo componente interface.
- **targetNamesapece** - No WSDL 1.1, é opcional, enquanto no WSDL 2.0 é obrigatório.
- **Definição do serviço** - No WSDL 1.1, os componentes Services podem ter múltiplos componentes ports e cada port pode ser associado a um diferente portType. No WSDL 2.0, cada componente Service está associado a apenas um componente interface.
- **Ordem dos elementos de alto nível** - No WSDL 1.1, os filhos de definition podem aparecer em qualquer ordem, enquanto no WSDL 2.0 a ordem é estritamente definida.
- **Definição de faults** - Escopo no componente Operation no WSDL 1.1 e no componente Interface no WSDL 2.0.
- **Importações** - No WSDL 1.1, o componente Import importa o mesmo ou um targetNamespace diferente. Já o WSDL 2.0 utiliza Include para o mesmo targetNamespace e import para diferentes targetNamespace consistentes com o schema XML.

4.4 SOAP

SOAP, originalmente, significava Simple Object Access Protocol e inicialmente funcionava como um protocolo de acesso a objetos, mas com o passar do tempo, tornou-se desejável que o SOAP servisse para um público muito mais amplo. Portanto, o foco da especificação em objetos moveu-se rapidamente para um framework de mensagens XML generalizadas. Hoje em dia, as definições encontradas nas especificações não mencionam mais objetos. O SOAP é um protocolo leve para troca de informação estruturada em um ambiente distribuído descentralizado. O SOAP usa a tecnologia XML para definir um extenso framework de mensagens, que provêm uma construção de mensagens capaz de se comunicar por meio da grande variedade de protocolos. O framework foi desenhado para ser independente de qualquer modelo particular de programa e implementações com semânticas específicas. O SOAP define um meio de mover mensagens XML de um ponto A para um ponto B, como vemos na figura 4.9.

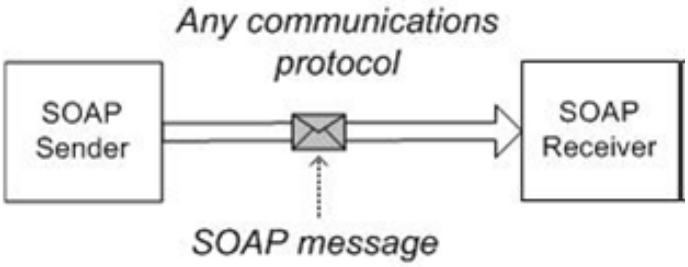

Figura 4.9: Troca simples de uma mensagem SOAP

O SOAP pode ser usado sobre qualquer protocolo de transporte, tal qual TCP, HTTP, SMPT, etc. Para manter a interoperabilidade, um protocolo padrão de bindings precisa ser definido para de-

linear as regras de cada ambiente. A especificação SOAP provém um framework flexível para definir arbitrariamente ligações dos protocolos e provém um binding específico para o HTTP.

O SOAP permite qualquer modelo de programação e não é vinculado ao RPC, além de definir um modelo para processamento individual e mensagens one-way. Pode-se, porém, combinar múltiplas mensagens em uma troca de mensagens em geral. A figura 4.9 ilustra uma mensagem simples (one-way), na qual o remetente não recebe uma resposta. O receptor pode, porém, enviar uma resposta de volta para o remetente, como vemos na figura 4.10.

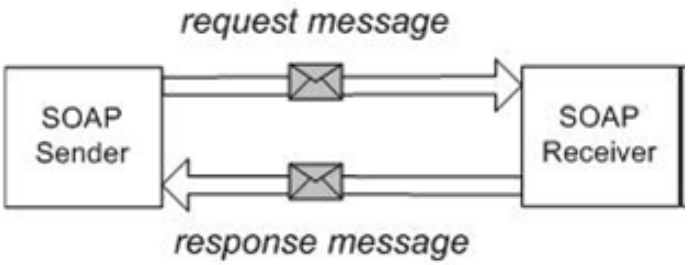

Figura 4.10: Padrão Request/Response de troca de mensagens

O SOAP permite qualquer número de padrão de troca de mensagem (MEPs), no qual request/response é apenas um. Outros exemplos são: solicit/response (inverso de request/response), notificações e longas conversações peer-to-peer.

4.4.1 Framework de Mensagens

A principal seção da especificação SOAP é o framework de mensagens. O framework de mensagens SOAP define uma suíte de elementos XML para empacotar arbitrariamente mensagens XML para transportar entre sistemas.

O framework é composto principalmente dos seguintes elementos XML: Envelope, Header, Body e Fault. Todos estão no http://schemas.xmlsoap.org/soap/envelope/ namespace no SOAP 1.1. O código 4.14 ilustra a definição completa do schema XML do SOAP 1.1.

```
<?xml version="1.0" encoding="utf-8" ?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
  xmlns:tns="http://schemas.xmlsoap.org/soap/envelope/"
  targetNamespace="http://schemas.xmlsoap.org/soap/envelope/">

<!-- Envelope, header and body -->
    <xs:element name="Envelope" type="tns:Envelope" />
    <xs:complexType name="Envelope" >
        <xs:sequence>
            <xs:element ref="tns:Header" minOccurs="0" />
            <xs:element ref="tns:Body" minOccurs="1" />
            <xs:any namespace="other" minOccurs="0"
              maxOccurs="unbounded" processContents="lax" />
        </xs:sequence>
```

```
        <xs:anyAttribute namespace="other"
          processContents="lax" />
    </xs:complexType>
    <xs:element name="Header" type="tns:Header" />
    <xs:complexType name="Header" >
        <xs:sequence>
            <xs:any namespace="other" minOccurs="0"
              maxOccurs="unbounded" processContents="lax" />
        </xs:sequence>
    <xs:anyAttribute namespace="other"
      processContents="lax" />
    </xs:complexType>
    <xs:element name="Body" type="tns:Body" />
    <xs:complexType name="Body" >
        <xs:sequence>
            <xs:any namespace="any" minOccurs="0"
              maxOccurs="unbounded" processContents="lax" />
        </xs:sequence>
        <xs:anyAttribute namespace="any"
          processContents="lax" />
    </xs:complexType>

<!-- Global Attributes -->
    <xs:attribute name="mustUnderstand" default="0" >
        <xs:simpleType>
            <xs:restriction base='xs:boolean'>
                <xs:pattern value='0|1' />
            </xs:restriction>
        </xs:simpleType>
    </xs:attribute>
    <xs:attribute name="actor" type="xs:anyURI" />
    <xs:simpleType name="encodingStyle" >
        <xs:list itemType="xs:anyURI" />
    </xs:simpleType>
    <xs:attribute name="encodingStyle"
      type="tns:encodingStyle" />
    <xs:attributeGroup name="encodingStyle" >
        <xs:attribute ref="tns:encodingStyle" />
    </xs:attributeGroup>
    <xs:element name="Fault" type="tns:Fault" />
    <xs:complexType name="Fault" final="extension" >
        <xs:sequence>
            <xs:element name="faultcode" type="xs:QName" />
            <xs:element name="faultstring" type="xs:string" />
            <xs:element name="faultactor" type="xs:anyURI"
              minOccurs="0" />
            <xs:element name="detail" type="tns:detail"
              minOccurs="0" />
        </xs:sequence>
```

```
    </xs:complexType>
    <xs:complexType name="detail">
        <xs:sequence>
            <xs:any namespace="any" minOccurs="0"
              maxOccurs="unbounded" processContents="lax" />
        </xs:sequence>
        <xs:anyAttribute namespace="any"
          processContents="lax" />
    </xs:complexType>
</xs:schema>
```

Código Fonte 4.14: Definição XML schema do SOAP 1.1

Observando a definição complexType para Envelope, pode-se perceber como esses elementos comunicam-se entre si. O modelo de mensagem no código 4.15 ilustra a estrutura de um envelope SOAP.

```
<soap:Envelope
  xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
    <soap:Header> <!-- optional -->
        <!-- header blocks go here... -->
    </soap:Header>
    <soap:Body>
        <!-- payload or Fault element goes here... -->
    </soap:Body>
</soap:Envelope>
```

Código Fonte 4.15: Estrutura de um envelope SOAP

O elemento Envelope é sempre o elemento raiz da mensagem SOAP. Isso facilita a identificação ads mensagens SOAP olhando simplesmente para o nome do elemento raiz.

O elemento Envelope contem um elemento opcional Header seguido de um elemento obrigatório Body. O elemento Body representa a mensagem real, é um container genérico que pode conter qualquer número de elementos de qualquer namespace.

O código 4.16 representa uma mensagem SOAP de requisição.

```
<soap:Envelope
  xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
    <soap:Body>
        <x:TransferFunds xmlns:x="urn:examples-org:banking">
            <from>22-342439</from>
            <to>98-283843</to>
            <amount>100.00</amount>
        </x:TransferFunds>
    </soap:Body>
</soap:Envelope>
```

Código Fonte 4.16: Mensagem SOAP de requisição

Se o receptor suportar um request/response e estiver apto a processar a mensagem com sucesso, deveria enviar uma outra mensagem SOAP de volta para o remetente inicial. Neste caso, a resposta deveria também vir contida no elemento Body, como mostra o código 4.17:

```
<soap:Envelope
  xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
    <soap:Body>
        <x:TransferFundsResponse
          xmlns:x="urn:examples-org:banking">
            <balances>
                <account>
                    <id>22-342439</id>
                    <balance>33.45</balance>
                </account>
                <account>
                    <id>98-283843</id>
                    <balance>932.73</balance>
                </account>
            </balances>
        </x:TransferFundsResponse>
    </soap:Body>
</soap:Envelope>
```

Código Fonte 4.17: Resposta de uma requisição SOAP

O framework de mensagem também define um elemento chamado Fault para representar erros dentro do elemento Body quando algo sai errado. Isso é essencial, porque sem uma representação de erro padrão, toda aplicação teria que inventar a sua própria, fazendo com que seja impossível para uma infra estrutura genérica, distinguir entre sucesso e falha.

O código 4.18 contém o elemento Fault indicando um erro de Insufficient Funds durante o processamento da requisição.

```
<soap:Envelope
  xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
    <soap:Body>
        <soap:Fault>
            <faultcode>soap:Server</faultcode>
            <faultstring>Insufficient funds</faultstring>
            <detail>
                <x:TransferError xmlns:x="urn:examples-org:banking">
                    <sourceAccount>22-342439</sourceAccount>
                    <transferAmount>100.00</transferAmount>
                    <currentBalance>89.23</currentBalance>
                </x:TransferError>
            </detail>
        </soap:Fault>
    </soap:Body>
```

```
</soap:Envelope>
```

Código Fonte 4.18: Elemento Fault SOAP

O elemento Fault precisa conter um faultcode seguido de um elemento faultstring. O elemento faultcode classifica o erro usando um nome de namespace adequado, enquanto o elemento faultstring provém uma explanação do erro inteligível para o homem.

4.4.2 Extensibilidade

A maioria dos protocolos fazem uma distinção entre o controle da informação (ex: header) e o conteúdo da mensagem, inclusive o SOAP. Além de fácil de usar, o benefício fundamental de Envelopes extensíveis é que pode ser usado com qualquer protocolo de comunicação. Os Headers sempre desempenharam um papel importante nos protocolos de aplicação, como HTTP, SMTP, etc, pois eles permitem que as aplicações nas duas pontas negociem o comportamento dos comandos suportados. O elemento Header, assim como o elemento Body, é um container para controle de informação. Ele pode conter qualquer número de elementos de qualquer namespace. Elementos colocados no elemento Header são referidos como header blocks. Assim como em outros protocolos, header blocks devem conter informações que influenciem no processo de carga.

Os Header Blocks podem ainda serem anotados pelo atributo global SOAP chamado mustUnderstand, para indicar quando o receptor precisa entender o cabeçalho antes do processamento da mensagem. O código 4.19 ilustra como requisitar o processamento.

```
<soap:Envelope
  xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
    <soap:Header>
        <!-- security credentials -->
        <s:credentials xmlns:s="urn:examples-org:security"
          soap:mustUnderstand="1">
            <username>dave</username>
            <password>evad</password>
        </s:credentials>
    </soap:Header>
...
```

Código Fonte 4.19: Exemplo de extensibilidade

Se o bloco está anotado com mustUnderstand = 1 e o receptor não foi desenhado para suportar o dado header, a mensagem não deveria ser processada e o Fault deveria retornar para o remetente com o código de status soap:MustUnderstand. Quando o mustUnderstand = 0 ou o atributo mustUnderstand não está presente, o receptor pode ignorar esses headers e continuar processando-o. O atributo mustUnderstand desempenha uma regra central sobre o modelo de processamento SOAP.

4.4.3 Modelo de Processo

O SOAP define um modelo de processamento que delineia as regras para o processamento de uma mensagem SOAP, assim como as transferências entre o SOAP remetente e o SOAP receptor.

O modelo de processo permite uma arquitetura mais interessante do que a vista anteriormente, pois contém múltiplos nós intermediários, como podemos ver na figura 4.11.

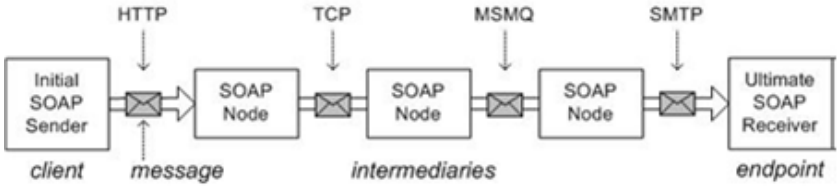

Figura 4.11: Troca de mensagens SOAP mais sofisticada

Um nó intermediário fica entre o remetente e o receptor interceptando mensagens SOAP. Um intermediário atua tanto como um remetente SOAP como um receptor SOAP ao mesmo tempo. Nós intermediários tornam possível desenhar algumas arquiteturas de rede flexíveis que podem ser influenciadas pelo conteúdo da mensagem. Roteamento SOAP é um bom exemplo de algo que aproveita fortemente os intermediários do SOAP.

Enquanto processa uma mensagem, um nó SOAP assume uma ou mais regras que influenciam como os headers SOAP são processados. As regras recebem um nome único e com isso podem ser identificadas durante o processamento. Quando o nó SOAP recebe uma mensagem para processar, precisa primeiro determinar que regras vai assumir. Isso pode ser consultado na mensagem SOAP para ajudar nessa determinação.

Uma vez determinadas as regras no qual irá agir, o nó SOAP precisa processar todos os headers obrigatórios (marcados com mustUnderstand=”1”) utilizando uma dessas regras. O nó SOAP precisa ainda escolher se processa os headers opcionais (marcados com mustUnderstand=”0”) usando uma dessas regras.

O SOAP 1.1 apenas define uma regra simples nomeada *http://schemas.xmlsoap.org/soap/actor/next*. Todo nó SOAP precisa assumir a próxima regra. Por isso, quando uma mensagem SOAP chega em qualquer nó SOAP, o nó precisa processar todos os headers obrigatórios orientados para a próxima regra.

Os Headers SOAP carregam uma regra específica por meio do atributo ator global. Se o atributo ator não está presente, o header é marcado com a última regra recebido por padrão. No código 4.20 a mensagem SOAP ilustra como usar o ator .

```
<soap:Envelope
  xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
    <soap:Header>
        <wsrp:path xmlns:wsrp="http://schemas.xmlsoap.org/rp"
          soap:actor="http://schemas.xmlsoap.org/soap/actor/next"
          soap:mustUnderstand="1">
...
```

Código Fonte 4.20: Exemplo de utilização do ator

Desde o wsrp:path header, é destinado para a próxima regra e marcado como mandatório (mustUnderstand=”1”. O primeiro nó SOAP a receber essa mensagem é requisitado para processar isso de acordo com a especificação do bloco do header, neste caso WS-Routing. Se o nó SOAP não foi desenhado para entender um header obrigatório designado para uma de suas regras, gera-se uma SOAP fault, com um status code soap:MustUnderstand e interrompe o processamento.

O elemento SOAP fault provém o elemento filho faultactor para especificar quem causou a falha que aconteceu dentro do pacote da mensagem. O valor de um atributo faultactor é uma URI que identifica o nó SOAP que causou a falha. Se um nó SOAP processar um header com sucesso, ele precisará remover o header da mensagem.

4.4.4 Protocolo Bindings

O SOAP habilita a troca de mensagens por meio de uma variedade de protocolos. Uma vez que o framework de mensagens SOAP é independente dos protocolos subjacentes, cada intermediário pode escolher usar um diferente protocolo de comunicação sem afetar a mensagem SOAP. Os protocolos Bindings padrões são necessários para assegurar o alto nível de interoperabilidade através das aplicações SOAP.

Um protocolo Binding concreto define exatamente como as mensagens SOAP deverão ser transmitidas e com que protocolo. O que o protocolo Binding realmente define, depende de muito da capacidade e das opções dos protocolos. Por exemplo, um protocolo binding para TCP deveria parecer bem diferente de um MSMQ ou SMTP. A especificação SOAP 1.1 apenas codifica um protocolo binding para HTTP, devido a sua ampla utilização.

O SOAP tem sido usado com outros protocolos além do HTTP, mas a implementação não segue a padronização do binding.

HTTP Binding

O protocolo binding HTTP define as regras para usar o SOAP sobre HTTP. O SOAP request/response é mapeado naturalmente para o modelo HTTP request/response. A figura 4.12 ilustra muitos dos detalhes do SOAP binding HTTP.

O cabeçalho Content-Type para ambas mensagens HTTP request e response, precisam ser text/xml (aplicação/soap +xml no SOAP1.2). Quanto a mensagem de request, é preciso usar POST para o verbo e a URI deve identificar o processo SOAP. A especificação SOAP ainda define um novo cabeçalho HTTP chamado SOAPAction, que precisa estar presente em todas as requisições SOAP HTTP (mesmo as vazias). O cabeçalho SOAPAction pretende expressar a intenção da mensagem. Quanto ao HTTP response, este precisa usar o código de status 200 se não ocorrerem erros e 500 se o body contiver um SOAP Fault.

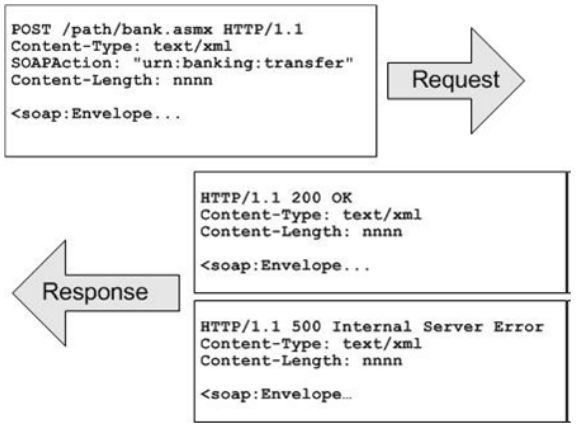

Figura 4.12: Modelo HTTP Request/Response

RPC and Encoding

Embora a especificação SOAP tenha evoluído longe de objetos, ainda define uma convenção para encapsulamento e troca de chamadas RPC usando o framework de mensagens descrito a seguir. Ao se definir um meio padrão de mapear chamadas RPC para mensagens SOAP faz com que seja possível a infra-estrutura traduzir automaticamente entre chamadas de métodos e mensagens SOAP em tempo de execução, sem redesenhar o código ao redor da plataforma Web service.

Para fazer uma chamada a um método usando SOAP, a infra-estrutura precisa seguir as seguintes informações:

1. Localização do Endpoint (URI)
2. Nome do método
3. Parâmetros names/values
4. Assinatura opcional do método
5. Dados opcionais no cabeçalho

Essa informação pode ser transmitida de diversas maneiras inclusive por bibliotecas, arquivos IDL ou arquivos WSDL. O binding do SOAP RPC define como encapsular e representar essa informação dentro do SOAP body. Faz-se isso definindo primeiramente como a assinatura do método mapeia uma estrutura simples de request/response, que possa ser codificada como XML. Os estados de binding RPC que chamam o método serão modelados como uma estrutura nomeada depois do método. A estrutura conterá um acessor para cada parâmetro [in] ou [in/out], nomeado igual ao nome do parâmetro, e na ordem definidos pela assinatura da mensagem. O método response vai também ser modelado como uma estrutura. O nome da estrutura é insignificante ainda que a convenção seja usar o nome do método seguido pela Response. A estrutura de response contém um acessor para retornar valores seguidos pelos acessores para cada parâmetro [out] ou [in/out].

O código **??** é um método C# para uma operação add:

```
double add(ref double x, double y)
```

Código Fonte 4.21: Método C para operação add

De acordo com as regras de binding RPC descritas, a estrutura de request representada na chamada de método deveria ser modelada como no código **??**:

```
struct add {
    double x;
    double y;
}
```

Código Fonte 4.22: Estrutura de request em C

Enquanto a estrutura de resposta deve parecer com a que segue no código **??**:

```
struct addResponse {
        double result;
        double x;
}
```

Código Fonte 4.23: Estrutura de resposta em C

A estrutura mapeada para XML segue a especificação SOAP que define um conjunto de regras de codificação. As regras de codificação do SOAP delimitam como mapear a maioria das estruturas de dados mais comumente usadas (como structs e arrays), para um formato XML comum.

De acordo com as regras de codificação SOAP, a estrutura de request vista no código **??** deve mapear a mensagem XML vista no código 4.24:

```
<add>
    <x>33</x>
    <y>44</y>
</add>
```

Código Fonte 4.24: Mensagem de resposta XML

Capítulo 5

Tradução para o JavaScript

5.1 Tradução dos componentes de um WSDL

5.1.1 Criação da estrutura de dados baseada no XML Schema

Cada operação descrita em um arquivo WSDL utiliza uma estrutura bem definida para mensagens de entrada e saída, descritas respectivamente em suas tags input e output.

```
<operation name="operationName"
           pattern="http://www.w3.org/ns/wsdl/in-out">
    <input messageLabel="In" element="tns:elemName"/>
    <output messageLabel="Out" element="tns:elemName"/>
</operation>
```

Código Fonte 5.1: Operação WSDL

Devido ao fato de o JavaScript não ser uma linguagem fortemente tipada, seria inviável a utilização de suas variáveis em sua forma básica para representação dos elementos do XML Schema, por isso foi necessário a criação de uma estrutura (WsdlObject) que pudesse resumir as informações de cada item do XML Schema. Esta estrutura deve ser incluída em um objeto, chamado "Schema", responsável por organizar toda a estrutura do XML Schema no Javascript.

```
var Schema = new function() {
    this.WsdlObject = function() {
        ...
    }
}
```

Código Fonte 5.2: Esqueleto do Objeto JavaScript Schema

A proposta dessa nova função é guardar três informações básicas de um objeto XML Schema: seu nome, tipo (Complexo ou Simples) e valor, assim como fornecer métodos getters e setters para o

gerenciamento das mesmas. Essas informações devem ser fornecidas no momento da criação da instância através de parâmetros.

```
var Schema = new function() {
    this.WsdlObject = function(_name, _type, _value) {
        var name = _name;
        var type = _type; //ComplexType (Complex), SimpleType (Simple)
        var value = _value;

        this.getName = function () {
            return name;
        }

        this.setName = function (_name) {
            name = _name;
        }

        this.getType = function () {
            return type;
        }

        this.setType = function (_type) {
            type = _type;
        }

        this.getValue = function () {
            return value;
        }

        this.setValue = function (_value) {
            value = _value;
        }

        ...
    }
}
```

Código Fonte 5.3: Objeto JavaScript WsdlObject

Além das informações triviais, é necessário que um WsdlObject armazene as informações dos seus subelementos, como descrito no código fonte 5.4. É importante frisar que cada Element pode existir mais de uma vez na mensagem XML caso não seja apresentado o atributo maxOccurs=“1”. Por este motivo, também é indispensável o armazenamento da posição de cada subelemento. Para este fim o uso de uma matriz se faz ideal, mas como o Javascript não fornece este tipo de estrutura, a solução encontrada foi a criação de dois vetores, chamados “attr” e“elementsArray”. O primeiro guardará uma posição, enquanto o segundo armazena todos os subelementos do objeto, como ilustrado na figura 5.1.

```
<xs:complexType name="typeName">
    <xs:sequence>
        <xs:element name="elemName" type="xs:string"/>
        ...
        <xs:element name="elemNameN" type="xs:string"/>
    </xs:sequence>
</xs:complexType>
```

Código Fonte 5.4: Subelementos do XML Schema

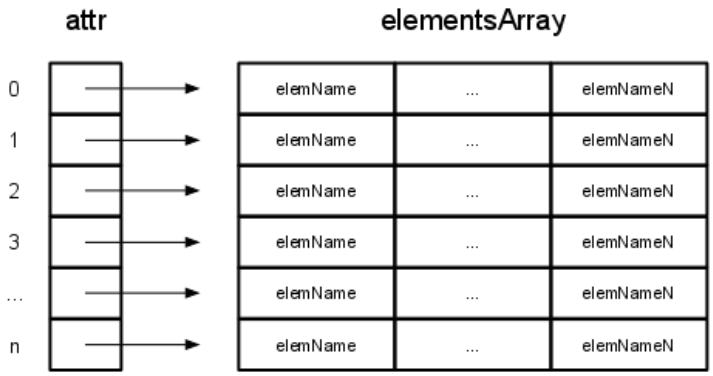

Figura 5.1: WsdlObjectArrays

Assim como existem getters and setters para as variáveis básicas, funções para o gerenciamento dos subelementos também devem ser criadas, neste caso quatro funções fazem este papel.

- addElement simplesmente adiciona o objeto do tipo WsdlObject recebido ao vetor elementArray.
- getAnAttr retorna um subelemento pelo nome e posição, sendo o argumento da posição opcional. Caso a posição não seja especificada, assume-se a primeira como requerida.
- setAnAttr atribui um subelemento do tipo WsdlObject, passado como argumento, pelo nome e posição. Sendo o argumento da posição opcional e caso a posição não seja especificada, assume-se a primeira como requerida.
- newAttr inicializa uma nova posição no vetor attr com os elementos do vetor elementsArray.

```
var Schema = new function() {
    this.WsdlObject = function (_name, _type, _value) {
        var name = _name;
        var type = _type; //ComplexType (Complex), SimpleType (Simple)
        var value = _value;
        var attr = new Array();
        var elementsArray = new Array();

        ...
```

```
        this.addElement = function(_obj) {
            elementsArray.push(_obj);
        }

        this.getAnAttr = function(_name, _pos) {
            if (_pos == undefined) _pos = 0;
            if (_pos > attr.length) throw new Error (500, "Tentativa de
            recuperar uma posição inexistente"); if (_pos == attr.length) {
                this.newAttr();
            }
            for (var p in attr[_pos]) {
                if (_name == attr[_pos][p].getName()) {
                    return attr[_pos][p];
                }
            }
        }

        this.setAnAttr = function(_attr, _name, _pos) {
            if (_pos == undefined) _pos = 0;
            if (_pos > attr.length) throw new Error (500, "Tentativa de fazer
                uma atribuição fora da ordem."); if (_pos == attr.length) {
                this.newAttr();
            }
            for (var p in attr[_pos]) {
                if (_name == attr[_pos][p].getName()) {
                        attr[_pos][p] = _attr;
                }
            }
        }

        this.newAttr = function() {
            var i = attr.length;
            attr[i] = new Array();
            for (var p in elementArray) {
                attr[i].push(p);
            }
        }
    }
}
```

Código Fonte 5.5: Funções para o gerenciamento dos subelementos

Como o Javascript não fornece mecanismos de herança, se torna dificil a reutilização deste código. Uma forma de contornar este problema é instanciar uma variável internamente (inheritFrom) com a função WsdlObject e chamá-la com os parâmetros corretos. No código fonte 5.6, por exemplo, o objeto string e int “herdam” de WsdlObject.

O WSDL propõe em sua "raiz" tags Element e ComplexType. Em função de uma melhor legibilidade, foi incluso um objeto types dentro do objeto JavaScript Schema, onde serão armazenadas as traduções de ComplexTypes, SimpleTypes e do objeto WsdlObject, conforme ilustrado no código fonte 5.6.

Existem tipos simples embutidos no XML Schema como int, string e boolean. Esses tipos devem ser traduzidos para o JavaScript para que os elements sejam instanciados de forma correta. Seguem dois exemplos desse tipo de tradução.

```
var Schema = new function() {

    types = {
        this.WsdlObject = function (_name, _type, _value) {
            ...
        },

        string: function(_name) {
            this.inheritFrom = Schema.types.WsdlObject;
            this.inheritFrom(_name, "Simple",undefined);
        },

        int: function(_name) {
            this.inheritFrom = Schema.types.WsdlObject;
            this.inheritFrom(_name, "Simple",undefined);
        }
    }
}
```

Código Fonte 5.6: Tradução de tipos para JavaScript

Os ComplexTypes definidos pelo WSDL são traduzidos seguindo a mesma linha, sendo acrescidos dos Elements pertencentes a ele. Segue abaixo a tradução de um ComplexType do WSDL utilizado, onde o nome da função é o mesmo nome do ComplexType e cada elemento adicionado possui tipo e nome equivalente ao atributos apresentados no XML. Caso a tag "All" se encontre no lugar da tag "sequence", o comportamento é o mesmo.

```
<xs:complexType name="typeName">
    <xs:sequence> <!-- <xs:all> -->
        <xs:element name="elemName1" type="xs:string"/>
        <xs:element name="elemName2" type="xs:int"/>
        <xs:element name="elemName3" type="xs:string"/>
        <xs:element name="elemName4" type="xs:string"/>
        <xs:element name="elemName5" type="xs:string"/>
    </xs:sequence> <!-- </xs:all> -->
</xs:complexType>
```

```
var Schema = new function() {
```

```
    types = {
        this.WsdlObject = function (_name, _type, _value) {
            ...
        }

        string: function(_name) {
            this.inheritFrom = Schema.types.WsdlObject;
            this.inheritFrom(_name, "Simple",undefined);
        }

        ...

        typeName: function(_name) {
            this.inheritFrom = types.WsdlObject;
            this.inheritFrom(_name, "Complex",undefined);
            this.addElement(new types.string("elemName1"));
            this.addElement(new types.int("elemName2"));
            this.addElement(new types.string("elemName3"));
            this.addElement(new types.string("elemName4"));
            this.addElement(new types.string("elemName5"));
        }
    }
}
```

Código Fonte 5.7: Tradrução do complexType

Os Elements, por sua vez, são armazenados dentro do objeto Schema apenas, com o objetivo de poderem ser declarados de forma mais intuitiva pelo usuário final. Existem duas formas de se declarar um Element, uma delas é com um SimpleType ou ComplexType definido através do atributo type. A outra é montando o seu próprio type dentro de suas tags.

Para o primeiro caso a tradução é bem simples e segue o mesmo princípio demonstrado no código anterior. No caso seguinte, o tipo deverá ser montado apenas para este Element, como seguem no exemplo do código fonte 5.8.

```
<xs:element name="elemName" type="xs:typeName"/>
<xs:element name="elemName1">
    <xs:complexType>
        <xs:sequence>
            <xs:element name="elemName2" type="tns:elemName3"/>
        </xs:sequence>
    </xs:complexType>
</xs:element>
```

```
var Schema = new function() {

    types = {
        ...
```

```
    }

    this.inheritFrom = types.WsdlObject;
    this.inheritFrom("schema","Complex",undefined);

    this.addElement(new types.typeName("elemName"));
    this.addElement(new function() {
        this.inheritFrom = types.WsdlObject;
        this.inheritFrom("elemName1", "Complex",undefined);
        this.addElement(new types.elemName3("elemName2"));
    });
}
```

Código Fonte 5.8: Mapeamento de um elemento com tipo único

A figura 5.2, exposta no final desse capítulo, ilustra a equivalência dos elementos e tipos traduzidos para o JavaScript.

5.1.2 Tradução das operações descritas no WSDL

Para fazer a tradução, definimos que todas as operações (métodos) deveriam estar contidas em algum objeto. Assim como ocorre no WSDL, onde as operações são definidas através do componente Binding especificado por um Endpoint, que por sua vez, pertence a um Service.

Para manter o contexto do WSDL, foi criado um objeto mais abrangente com o nome do serviço, que define objetos com o nome de cada Endpoint, conforme representado no código fonte 5.9. Estes objetos referentes aos Endpoints ficam responsável por conter os métodos dos Bindings correspondentes, que poderão ser chamados pelo usuário do Stub.

```
<service name="serviceName"
         interface="tns:interfaceName">
    <endpoint name="endpointName"
              binding="tns:bindingName"
              address="https://www.domain.com/onca/soap?Service=serviceName">
     </endpoint>
</service>
```

```
serviceName = {

    endpointName: {
        ...
    }
}
```

Código Fonte 5.9: Tradução do Serviço

Algumas informações importantes são necessárias para fazer as requisições ao Web Service, como o targetNamespace do WSDL, a URL contida no atributo address de cada Endpoint e a versão da

mensagem SOAP que será trocada com o servidor. O código fonte 5.10 apresenta a tradução dessas informações para o JavaScript.

O targetNamespace é único e independe do serviço para todo o arquivo WSDL, mas como o objeto mais amplo definido foi o criado a partir do nome do serviço, um atributo com esta especificação precisa ser declarado dentro deste objeto. O address é um atributo específico do Endpoint e portanto deve ser incluída no respectivo objeto do Javascript. A versão da mensagem SOAP é armazenada na extensão version, localizada na tag <binding> referenciada pelo Endpoint.

```
<description
targetNamespace="http://webservices.domain.com/serviceName/"
... >
```

```
<endpoint name="endpointName"
          ...
          address="https://www.domain.com/onca/soap?Service=serviceName">
</endpoint>
```

```
<binding name="bindingName"
         ...
         wsoap:version="X.X">
</binding>
```

```
sarviceName = {

    namespace: "http://webservices.domain.com/serviceName/",

    endpointName: {

        address:"https://www.domain.com/onca/soap?Service=serviceName",

        soapVersion: "X.X",

        ...
    }
}
```

Código Fonte 5.10: Atributos do Serviço

A tradução das operações do arquivo WSDL vem sendo feita para Javascript seguindo uma sequencia especifica, onde é necessário pesquisar qual Binding está associado ao Endpoint em questão comparando o atributo “binding” do Endpoint com o atributo “name” das Bindings, como visto acima para o caso do “bindingName”.

Para cada operação dentro do Binding, uma função deverá ser criada no Javascript. No exemplo do código fonte 5.11, o conteúdo do atributo “ref” é utilizado como nome da função e o atributo “action” é armazenado em uma variável da função.

Dois parâmetros de entrada são requisitados ao usuário, “body” e “outMethod”. O primeiro deverá conter uma instância do objeto “Schema”, que contém a estrutura de dados necessária para que a mensagem SOAP enviada seja montada corretamente. O segundo é a função de retorno que irá receber a resposta do servidor, já adaptada a estrutura mencionada anteriormente.

```
<binding name="bindingName"
         ...>
    <operation ref="tns:operationName1"
    wsoap:action="http://soap.domain.com/operationName1">
    </operation>

    <operation ref="tns:operationName2"
    wsoap:action="http://soap.domain.com/operationName2">
    </operation>
</binding>
```

```
serviceName = {

    namespace: "http://webservices.domain.com/serviceName/",

    endpointName: {

        address:"https://www.domain.com/onca/soap?Service=serviceName",

        soapVersion: "1.1",

        operationName1: function (body, outMethod) {
            var operation = "operationName1";
            var action = "http://soap.domain.com/tns:operationName1";
            ...
        },

        operationName2: function (body, outMethod) {
            var operation = "operationName2";
            var action = "http://soap.domain.com/operationName2";
            ...
        }
    }
}
```

Código Fonte 5.11: Tradução parcial das Operações

O Binding está atrelado a uma interface. Esta ligação se dá pelo seu atributo “interface”, assim como os seus “operations”, que fazem referência aos da interface. O output “element” também é necessário para complementar as funções escritas no código Javascript de modo a criar o objeto que conterá as informações retornadas através da mensagem SOAP.

Para efetuar a troca das mensagens é necessário fazer uma chamada a função responsável por

isso, além de chamar a função de retorno do usuário passando todas as informações relevantes como parâmetro. A figura 5.3 ilustra a equivalência das operações traduzidas para o JavaScript.

```
<binding name="bindingName"
         interface="tns:interfaceName"
         ...>
    <operation ref="tns:operationName"
               wsoap:action="http://soap.domain.com/oparationName1">
    </operation>

    <operation ref="tns:oparationName2"
               wsoap:action="http://soap.domain.com/operationName2">
    </operation>
```

```
<interface name="interfaceName">
    <operation name="operationName1"
               pattern="http://www.w3.org/ns/wsdl/in-out">
      <input messageLabel="In" element="tns:elemName1"/>
      <output messageLabel="Out" element="tns:elemName2"/>
    </operation>
    <operation name="operationName2"
               pattern="http://www.w3.org/ns/wsdl/in-out">
      <input messageLabel="In" element="tns:elemName3"/>
      <output messageLabel="Out" element="tns:elemName4"/>
    </operation>
```

```
serviceName = {

    namespace: "http://webservices.domain.com/serviceName/",

    endpointName: {

        address: "https://www.domain.com/onca/soap?Service=serviceName",

        soapVersion: "1.1",

        operationName2: function (body, outMethod) {
            var operation = "operationName2";
            var output = "elemName4";
            var action = "http://soap.domain.com/operationName2";
            var outParam = Schema.getAnAttr("elemName4");
            SOAPActions.call(serviceName.endpointName.address,
                             operation,
                             output,
                             outMethod,
                             serviceName.namespace,
                             action,
                             true,
```

```
                            body,
                            outParam,
                            serviceName.endpointName.soapVersion);
        },

        operationName1: function (body, outMethod) {
            var operation = "operationName1";
            var output = "elemName2";
            var action = "http://soap.domain.com/operationName1";
            var outParam = Schema.getAnAttr("elemName2");
            SOAPActions.call(serviceName.endpointName.address,
                            operation,
                            output,
                            outMethod,
                            serviceName.namespace,
                            action,
                            true,
                            body,
                            outParam,
                            serviceName.endpointName.soapVersion);
        },
    }
}
```

Código Fonte 5.12: Tradução completa das operações

5.2 Objetos auxiliares para comunicação

Foi necessário desenvolver o código JavaScript responsável por toda a comunicação com os Web Services. Esse código é independente do WSDL lido e pode ser usado com qualquer Web Service que troque mensagens utilizando o protocolo SOAP nas versões 1.1 e 1.2. As sub-seções a seguir mostram em detalhes as principais funcionalidades desses dois objetos auxiliares.

5.2.1 SOAPAction

O primeiro objeto auxiliar apresentado será o SOAPActions, este é um objeto que possui conteúdo independentemente dos Web Services utilizados, pois tem funções necessárias para qualquer Web Service.

Sua principal função se restringe a montar corretamente uma mensagem SOAP, enviá-la ao endereço dos Web Services utilizando AJAX, e receber a resposta. O tratamento das mensagens conversão do XML para uma estrutura de dados em Java Script, não é abordada pelo SOAPActions.

A função call, descrita no código fonte 5.13 faz apenas uma distinção de acordo com a preferência do usuário por uma requisição assíncrona ou síncrona.

```
    call: function(location,
                   inMethod,
```

```
                    outMethod,
                    namespace,
                    soapaction,
                    async,
                    inParam,
                    outParam) {
    if(async)
        SOAPActions.loadMessage (location,
                                 inMethod,
                                 outMethod,
                                 namespace,
                                 soapaction,
                                 async,
                                 inParam,
                                 outParam);
    else
        return SOAPActions.loadmessage (location,
                                        inMethod,
                                        outMethod,
                                        namespace,
                                        soapaction,
                                        async,
                                        inParam,
                                        outParam);
}
```

Código Fonte 5.13: Chamadas síncronas e assíncronas

A mensagem SOAP, conforme reproduzido no código fonte 5.14, depende da versão requisitada e é montada com as informações pertinentes a mensagem que os Web Services esperam receber, acrescida dos dados passados pelo usuário no objeto inParam. O método objectToXml, descrito na subseção 5.2.2, auxilia na tradução dos objetos "inParam" para o XML.

```
loadMessage: function(...) {
    if (soapVersion = "1.1")
        var message =
            "<?xml version=\"1.0\" encoding=\"utf-8\"?>" +
            "<soap:Envelope " +
            "xmlns:xsi=\"http://www.w3.org/2001/XMLSchema-instance\" " +
            "xmlns:xsd=\"http://www.w3.org/2001/XMLSchema\" " +
            "xmlns:soap=\"http://schemas.xmlsoap.org/soap/envelope/\">" +
            "<soap:Body>" +
            "<" + operation + " xmlns=\"" + namespace + "\">" +
            Utils.objectToXml(inParam) +
            "</" + operation + "></soap:Bodys></soap:Envelope>";
    else if (soapVersion = "1.2")
        var message =
            "<?xml version=\"1.0\" encoding=\"utf-8\"?>" +
```

```
            "<soap12:Envelope " +
            "xmlns:xsi=\"http://www.w3.org/2001/XMLSchema-instance\" " +
            "xmlns:xsd=\"http://www.w3.org/2001/XMLSchema\" " +
            "xmlns:soap12=\"http://www.w3.org/2003/05/soap-envelope\">" +
            "<soap12:Body>" +
            "<" + operation + " xmlns=\"" + namespace + "\">" +
            Utils.objectToXml(inParam) +
            "</" + operation + "></soap12:Body></soap12:Envelope>";
    else throw new Error (500, "Versão SOAP para a construção da mensagem não
    identificada");
    ...
},
```

Código Fonte 5.14: Montagem das mensagens SOAP

Com a mensagem pronta, é necessário abrir uma conexão utilizando AJAX para que se possa trocar mensagens com o servidor dos Web Services. A troca de mensagens é efetuada com as tradicionais distinções para o caso de o usuário desejar receber uma resposta assíncrona. Caso a opção seja por uma operação assíncrona, é necessário esperar até que o readyState seja "4", que significa que a resposta já está pronta, para que se possa trabalhar em cima da dela. Caso a operação seja síncrona, não há necessidade de realizar este tipo de verificação. O processo é descrito no código fonte 5.15

```
loadMessage: function(...) {
    ...
    var xmlHttp = Utils.getXmlHttp();
    xmlHttp.open("POST", location, async);
    xmlHttp.setRequestHeader("SOAPAction", soapaction);
    xmlHttp.setRequestHeader("Content-Type", "text/xml; charset=utf-8");
    if(async) {
        xmlHttp.onreadystatechange = function() {
            if(xmlHttp.readyState == 4)
                SOAPActions.loadedMessage(operation,
                                          operationResponse,
                                          outMethod,
                                          async,
                                          xmlHttp,
                                          outParam);
        }
    }
    xmlHttp.send(message);
    if (!async)
        return SOAPActions.loadedMessage(operation,
                                         operationResponse,
                                         outMethod,
                                         async,
                                         xmlHttp,
                                         outParam);
}
```

Código Fonte 5.15: Envio da mensagem SOAP

A função loadedMessage, descrita pelo código fonte 5.16, é responsável por tratar a resposta, a mensagem XML é traduzida para o formato de Nodos pela API DOM do Javascript, tratada por uma função do objeto Utils, apresentado na subseção 5.2.2, e a função de retorno é executada com a resposta devidamente convertida para a estrutura de dados organizada no objeto JavaScript Schema.

```
loadedMessage: function(operation,
                        operationResponse,
                        outMethod,
                        async,
                        xmlHttp,
                        outParam){
// Aloca a resposta SOAP em uma estrutura de Nó utilizando a API DOM.
var ResponseNode = Utils.getNode(xmlHttp, operationResponse);
//Converte a estrutura de Nó para a estrutura de dados de outParam (Objeto)
outParam = Utils.nodeToObject(ResponseNode, outParam);
// E retorna a resposta
outMethod(outParam);
if(!async)
    return outParam;
}
```

Código Fonte 5.16: Tratamento da resposta SOAP

5.2.2 Utils

O objeto Utils tem como função principal manter funções que serão necessárias durante todo o processo. Devido ao fato de este objeto ter ficado relativamente grande e ser utilizado para os mais diversos tipos de funções, iremos apenas mostrar as funções que apresentaram maior influência no resultado final do projeto.

objectsToXML

É a função responsável por escrever a parte dos dados provenientes do objeto JavaScript Schema em XML. Este método não se preocupa em incluir o primeiro objeto da hierarquia na conversão, pois ele possui o mesmo nome da operação a ser chamada nos Web Services e seria redundante colocá-lo no corpo da mensagem, ocasionando erro de formatação e o não entendimento da mensagem por parte dos Web Services. Uma operação “ConsultaCEP” envia os dados a partir de um elemento também chamado “ConsultaCEP”, por exemplo. Esta conversão é ilustrada no código fonte 5.17.

```
objectToXml: function (obj) {
   xmlStr = "";
```

```
   for (var i in obj.getAttr()[0]) {
      xmlStr += Utils.buildTag (obj.getAttr()[0][i]);
   }
        return xmlStr;
},

buildTag: function (obj) {
    var strTag = "";
    if (obj.getType() == "Complex") {
        if (obj.getAttr().length > 0) {
            strTag += "<" + obj.getName() + ">";
            for (var p in obj.getAttr())
                for (var i in obj.getAttr()[p])
                    strTag += Utils.buildTag (obj.getAttr()[p][i]);
            strTag += "</" + obj.getName() + ">";
        }
    } else {
        if (obj.getValue() != undefined) {
            strTag += "<" + obj.getName() + ">";
            strTag += obj.getValue().replace(/&/g, "&amp;").replace(/</g,"&lt;").
                                                        replace(/>/g,"&gt;");
            strTag += "</" + obj.getName() + ">";
        }
    }
    return strTag;
}
```

Código Fonte 5.17: Tradução de objeto para XML

nodeToObject

Esta função recebe como parâmetros uma árvore de nós, retirada da resposta XML enviada pelos Web Services, e o objeto que deve possuir estrutura similar para que a conversão funcione. Este método busca todos os valores incluídos na árvore e procura seu par na estrutura de objetos declarados no objeto JavaScript Schema, descendo na hierarquia de objetos de forma idêntica até que se encontre um nó folha e busque o seu valor com auxílio da função getValue, demonstrada no código fonte 5.18, assim como a função nodeToObject.

```
// Constrói um objeto do tipo OutParam se baseando na árvore de nós
nodeToObject: function(node, outParam) {
    if (node == null)
        return null;
    // Se o nó for do tipo texto, então retorna seu valor final
    if (node.nodeType == 3 || node.nodeType == 4)
        return Utils.getValue(node, outParam);
    nodeLocalName = Utils.getNodeLocalName(node);
    // Apenas continua se o nodeName existir no objeto outParam, caso contrário
```

```
    //ocorre um erro
    if (outParam.getName() == nodeLocalName) {
        // Se for um nó folha retorna o m?todo recursivamente chamando o nó
        filho para o mesmo outParam
        if (Utils.isLeafNode(node)) {
            return Utils.nodeToObject(node.childNodes[0], outParam);
        }
        for(var i = 0; i < node.childNodes.length; i++) {
            var pos = Utils.getSameNodesPosition (node.childNodes[i]);
            var p = Utils.nodeToObject(node.childNodes[i],
                            outParam.getAnAttr(Utils.getNodeLocalName(node.childNodes[i]),
                            pos));
           outParam.setAnAttr(p,
                             Utils.getNodeLocalName(node.childNodes[i]),
                            pos);
        }
        return outParam;
    }
    throw new Error (500, "Erro na convers?o de n?s para objetos");
}

getValue: function(node, outParam) {
    var value = node.nodeValue;
    if (outParam.getType() == "Simple") {
        outParam.setValue(value);
        return outParam;
    } else if (outParam.getType() == "Complex") {
        throw new Error (500, "Não é possível retorna o valor de um ComplexType")
    }
},
```

Código Fonte 5.18: Tradução da Arvore de nós para objeto Schema

<xs:schema ...>
...
</xs:schema>
	()
	var schema = new function() {
	types = {
	... (1)
	}
	this.inheritFrom = types.WsdlObject
	this.inheritFrom("schema", "Complex", undefined)
	... (2)
	}

<xs:complexType name="typeName">
<xs:sequence>
...
</xs:sequence>
</xs:complexType>
	(1)
	typeName: function(_name) {
	this.inheritFrom = types.WsdlObject;
	this.inheritFrom(_name, "Complex",undefined);
	... (2)
	}

<xs:element name="elemName" type="elemType"/>	(2) this.addElement(new types.elemType ("elemName"));

<xs:element name="elemName">
<xs:complexType>
<xs:sequence>
...
</xs:sequence>
</xs:complexType>
</xs:element>
	(2)
	this.addElement(new function() {
	this.inheritFrom = types.WsdlObject;
	this.inheritFrom("elemName", "Complex",undefined);
	... (2)
	});

<xs:element name="elemName">
<xs:complexType>
<xs:all>
...
</xs:all>
</xs:complexType>
</xs:element>
	(2)
	this.addElement(new function() {
	this.inheritFrom = types.WsdlObject;
	this.inheritFrom("elemName", "Simple",undefined);
	... (2)
	});

<xs:element name="elemName"> <xs:simpleType> <xs:restriction base="elemType"> ... </xs:restriction> </xs:simpleType> </xs:element>	(2) this.addElement(new types.elemType("elemName"));
<xs:element ref="elemName" />	(2) this.addElement(new Schema.getAAttr("elemName"));

Figura 5.2: Tabela de Tradução de tipo

<service name="serviceName" ...>
<endpoint name="serviceNamePort"...>
...
</endpoint>
</service>
	serviceName {
	... (1)
	() serviceNamePort {
	... (2)
	}
	}

<description targetNamespace="http://webservices.domain. com/serviceName/" ... > ... </description>	(1) namespace: "http://webservices.domain. com/serviceName/",
<endpoint ... address="https://www.domain.com/onca/soap? Service=serviceName"> </endpoint>	(2) address:"https://www.domain.com/onca/soap? Service=serviceName",
<binding name="bindingName" ... wsoap:version="X.X"> ... </binding>	(2) soapVersion: "X.X",
<operation ref="tns:operationName1" wsoap:action="http://soap.domain. com/operationName1"> ... </operation>	(2) operationName1: function (body, outMethod) { var operation = "operationName1"; var action = "http://soap.domain.com/tns: operationName1"; ... (3) },
<operation name="operationName1" ...> <input messageLabel="In" element="tns:elemName1"/> <output messageLabel="Out" element="tns:elemName2"/> </operation>	(3) var output = "elemName4"; var outParam = Schema.getAAttr("elemName4"); SOAPActions.call(serviceName.endpointName.address, operation, output, outMethod, serviceName.namespace, action, true, body, outParam, serviceName.endpointName.soapVersion);

Figura 5.3: Tabela de Tradução de Operações

Capítulo 6

Implementação do Gerador Automático

6.1 Estrutura do Tradutor

O trabalho de um tradutor pode ser dividido basicamente em três partes:

- Análise léxica.
- Análise Sintática.
- Geração do Código.

A análise léxica é o processo de converter uma sequência de caracteres em uma sequência de tokens. Analisadores léxicos consistem em um scanner e um tokenizer e podem ser entendidos como um validador. A função de um analisador léxico é dividir uma entrada de fluxo de caracteres em tokens.

A análise sintática, também conhecida como parsing, é o processo de analisar uma sequência de tokens para determinar a sua estrutura gramatical e se esta sequência atende as especificações sintáticas de uma gramática formal.

Para a implementação, utilizamos a API Apache Woden como analisador léxico e sintático. Um fator importante para essa escolha foi a prévia utilização desta ferramenta para a conversão do WSDL 1.1 para o WSDL 2.0, possibilitando que só fosse preciso desenvolver a tradução da versão 2.0 do WSDL para atender as duas versões.

Tendo os processos de parsing e scanner sido executados com sucesso, e não gerado nenhum erro, o sistema terá uma representação interna do WSDL a qual pode ser facilmente manipulada pelo último estágio do tradutor: o gerador de código, que ao contrário dos analisadores léxico e sintático, precisa ser escrito manualmente a fim de se obter o comportamento desejado.

6.2 Apache Woden

O objetivo inicial do projeto Woden é desenvolver um processador de WSDL 2.0 que implementa a especificação de WSDL 2.0 do W3C. O segundo objetivo inclui o suporte para parsing de XML com alto desempenho e o suporte para WSDL 1.1.

O processador de WSDL do Woden é implementado como um framework com pontos de extensão para que o usuário possa definir o seu comportamento. O Woden é uma API baseada em DOM e por isso possui as propriedades de tratar XML. A API Woden contém duas "sub-APIs" que representam modelos alternativos do objeto WSDL 2.0:

- A API Element representa um modelo dos elementos e atributos XML no WSDL 2.0 namespace, como descrito pelo mapeamento XML na especificação do WSDL 2.0.
- A API Component representa um modelo abstrato do componente WSDL descrito pela especificação WSDL 2.0.

API Element

A API Element permite a navegação pela hierarquia dos elementos WSDL aninhados que poderiam aparecer em um documento WSDL. Por exemplo, DescriptionElement declara os métodos getinterfaceElements, getBindingElements e getServiceElements que fornecem acesso ao nível mais alto dos elementos WSDL. InterfaceElement declara os métodos getInterfaceFaultElements e getInterfaceOperationElements e assim por diante.

A figura 6.1 representa uma parte do Diagrama de Classes da API Element, com a classe DescriptionElement se relacionando com as classes DocumantationElement, IncludeElement, TypesElement, InterfaceElement, ImportElement, BindingElement e ServiceElement.

API Component

A API Component representa o modelo abstrato do Componente WSDL. Ela difere da API Element em que certos aspectos do WSDL XML não estão representados no modelo Component. A API Component disponibiliza uma visão com permissão apenas para leitura do modelo Component do WSDL. A única forma de se criar o objeto Description é pela chamada do método toComponent em um objeto DescriptionElement. Tendo o objeto Description pode-se acessar o resto do modelo componente do WSDL, porém sem poder modifica-lo. WSDL só pode ser criado ou modificado por meio da API Element.

A figura 6.2 representa uma parte do Diagrama de classes da API Component, com a classe Descripiton no centro se relacionando com as classes Services, Interface, TypeDefinition, ElementDeclaretion e Bindings.

Os principais recursos da API Woden são:

- O mecanismo Factory para a criação de objetos Woden.
- Configurar o comportamento do Woden definindo características ou propriedades do WSDLReader.

Figura 6.1: Diagrama de Classes API Element Woden

- Personalizar o comportamento no tratamento de erro.
- Manipulação de modelos de elementos e atributos WSDL baseados em XML.
- Manipulação de modelos abstratos de componentes WSDL.

A tabela 6.1 ilustra como é tratado o mapeamento de elementos WSDL para a API.

6.3 Estrutura lógica do gerador

O gerador automático de Stubs foi implementado em java com a ajuda da API Apache Woden, que faz a análise léxica e sintática do WSDL e gera um conjunto de classes equivalentes para cada elemento do WSDL. Após termos essas classes mapeadas, foi possível iniciar a fase de desenvolviemnto do processo de tradução para o JavaScript.

O Gerador é subdividido em duas classes principais: “LaunchApp” e “JSFileStub”. No qual a primeira é a classe principal do gerador e é a que controla o fluxo de execução do programa. Já a “JSFileStub” é o coração do gerador, onde é feita toda a tradução do WSDL e a geração de códigos para a montagem da mensagem SOAP para que a integração com o Web Service possa ser feita.

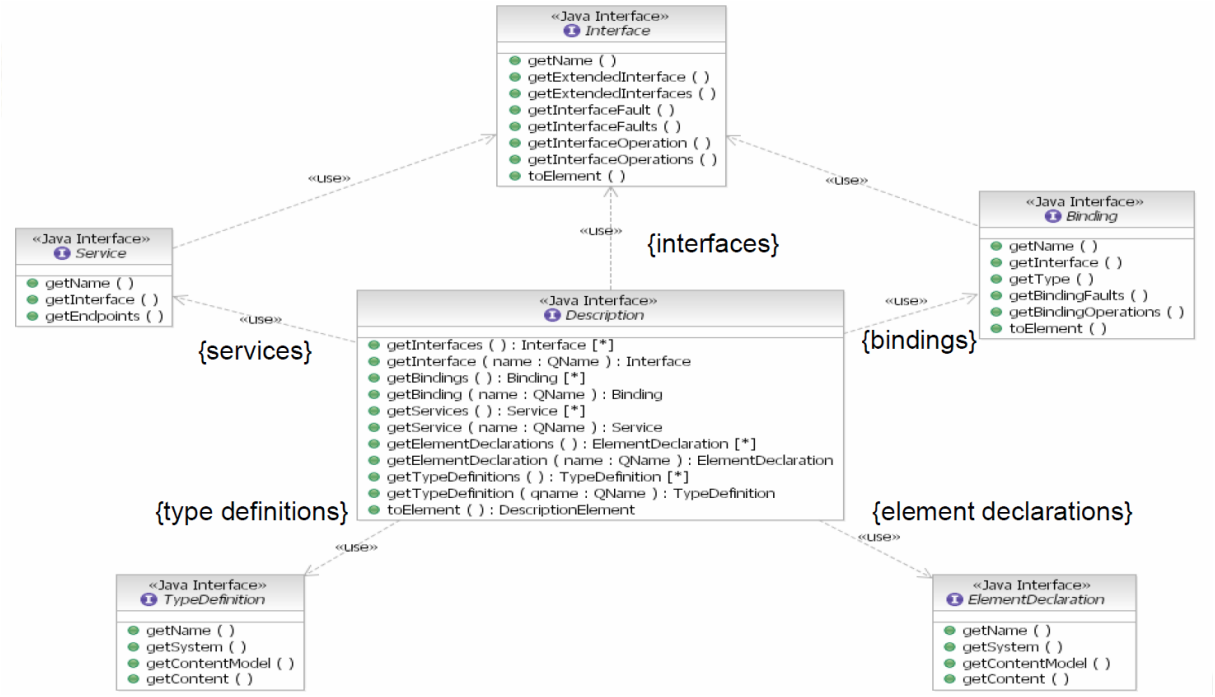

Figura 6.2: Diagrama de Classes API Component Woden

6.4 Geração de Código Auxiliar

Para que haja a conexão entre o cliente e o servidor, é preciso que o stub gerado seja capaz de criar mensagens no padrão SOAP empacotando os objetos que serão enviados para o Web Service. Foi desenvolvido o código em JavaScript capaz de não só empacotar a mensagem com o padrão SOAP como também ficar responsável pela requisição ao Web Service e aguardar a mensagem de resposta com as informações requeridas, para depois transforma-las novamente em objetos JavaScript.

A classe JSFileStub gera o código responsavel por montar a mensagem SOAP. A exemplo do que foi explicado na seção 5.2, são gerados dois objetos responsáveis por esse controle: SOAPActions e Utils.

6.5 Mapeamento para o JavaScript

Com o WSDL já mapeado para as classes Java como foi visto na seção 6.2, precisavamos agora escrever toda a tradução para o JavaScript. Definir a equivalencia de cada elemento WSDL na linguagem alvo. Para isso os elementos WSDL gerados foram divididos em alguns grupos: Operações, Elementos, Tipos e o Schema que contem todos os outros três.

Operations

As operações são traduzidas, de forma que para se fazer uma chamada a elas, só é preciso passar como parâmetro a mensagem que deseja enviar e a função de retorno. A tradução do operations foi feita a partir da definição de um WSDL conforme descrito na seção 5.1.2. Todas as operações de um WSDL possuem trechos de códigos idênticos mudando apenas os parâmetros dessas operações.

Para o mapeamento das operações utilizamos a classe Description da API Component. A partir dela foi possível armazenar todos os serviços em uma variável do tipo Service[], com isso para cada serviço presente no WSDL o tradutor monta um objeto com o nome do serviço, com a versão do SOAP e para cada Endpoint relacionado a esses serviços, o gerador busca o Binding e gera objetos com todas as operações que esse Binding possui. A figura 6.3 ilustra todas as classes percorridas pelo gerador para que ele consiga montar todas as operações de cada serviço.

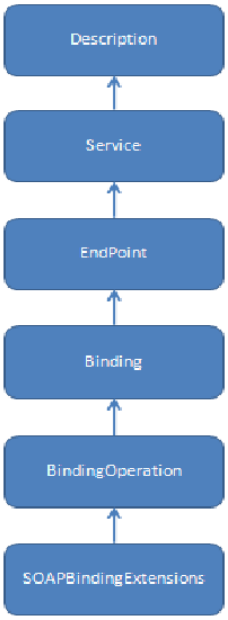

Figura 6.3: Representação Java do WSDL

Elementos e Tipos

A tradução dos elementos e tipos foram feitas a partir da definição de um WSDL conforme visto na seção 5.1.1. Todo elemento pode ser do tipo complexo ou simples e possuem algumas variações na forma como são declarados. A seção 3.2 exemplifica muitas das variações possiveis para essa declaração. Com isso, cada elemento e cada tipo decladrados no WSDL precisam ser mapeados e depois traduzido para o Javascript de acordo com as especificações definidas.

O gerador primeiramente faz a tradução todos os tipos presentes no WSDL para então começar a montar os elementos. Durante a tradução dos tipos, o gerador passa o conteudo de cada tipo para um método que verifica se o tipo é um complexType ou um simpleType e gera um objeto JavaScript para cada tipo.

Para a tradução dos elementos o gerador mantém um comportamento muito parecido, pois também passa o conteúdo de cada elemento para um método que checa se o elemento possui o atributo “name”. Caso o atributo “name”esteja presente, o objeto é criado caso contrário o gerador verifica se o próximo nó é uma instância de um complexType ou simpleType e cria os objetos de acordo com o seu tipo.

Para o mapeamento dos elementos e tipos utilizamos a classe Description da API Component. A

partir dela foi possível armazenar todas as definições de tipo que estão na raiz em uma variável do tipo TypeDefinition[] e as declarações de elementos em uma variável do tipo ElementDeclaration[]. A exemplo do que foi feito durante a tradução das operações o gerador vai descendo na hierarquia até atingir os nós com as informações necessárias.

Schema

O Objeto Schema é na verdade o resultado final da tradução e geração de códigos auxiliares feitos pelo Gerador Automático de Stubs. Nele se encontram os tipos, operações e elementos. A criação de um objeto que encapsulasse todos os outros gerados, foi a maneira encontrada para simular uma herança no JavaScript e evitar que o mesmo código fosse reescrito diversas vezes sem necessidade. As figuras 5.2 e 5.3 representam um esquema de como foi mapeado o WSDL para o JavaScript.

O fato de todo o código do stub gerado estar contido dendro do objeto Schema facilita para o desenvolvedor que pretende utilizar algum Web Service, pois só precisará adicionar a sua declaração de script um arquivo.

Exemplo de utilização do Stub gerado

O código fonte 6.1 descreve como o usuário deverá utilizar o stub gerado. Primeiramente, o usuário precisa instanciar uma variável com o objeto requerido pela operação utilizada. Através desta variável os valores escolhidos podem ser atribuídos. Com a variável pronta e com seus valores customizados, o usuário faz uma chamada ao método da operação do WSDL, passando como parâmetros a variável contendo as informações e a função de retorno. A função de retorno é responsável por tratar a resposta recebida pelo Web Service, neste exemplo são usados alerts apenas para fins ilustrativos. Nesse caso específico utilizamos o WSDL "cep.wsdl2", que pode ser visto em anexo junto com o stub em JavaScript que foi gerado.

```
Consultar = function () {
    var parameters = Schema.getAnAttr("ConsultaCEP");
    parameters.getAnAttr("CEP").setValue("XXXXX-XXX");
    CEPWebService.CEPWebServiceSoap.ConsultaCEP (parameters, ConsultaResposta);
}
```

Código Fonte 6.1: Função de chamada do Stub

```
ConsultaResposta = function (parameters) {
    alert(parameters.getAnAttr("ConsultaCEPResult").getAnAttr("CEP").getValue());
    alert(parameters.getAnAttr("ConsultaCEPResult").getAnAttr("Logradouro").getValue());
    alert(parameters.getAnAttr("ConsultaCEPResult").getAnAttr("TipoLogradouro").getValue());
    alert(parameters.getAnAttr("ConsultaCEPResult").getAnAttr("Bairro").getValue());
    alert(parameters.getAnAttr("ConsultaCEPResult").getAnAttr("Cidade").getValue());
    alert(parameters.getAnAttr("ConsultaCEPResult").getAnAttr("UF").getValue());
    alert(parameters.getAnAttr("ConsultaCEPResult").getAnAttr("Mensagem").getValue());
    alert(parameters.getAnAttr("ConsultaCEPResult").getAnAttr("Status").getValue());
}
```

Código Fonte 6.2: Função de retorno do Stub

Elementos WSDL		
Elemento WSDL	**API Element**	**API Component**
<description>	DescriptionElement	Description
<documentation>	DocumentationElement	
<import>	ImportElement	
<include>	IncludeElement	
<types>	TypesElement	
<interface>	InterfaceElement	Interface
<fault>	InterfaceFaultElement	InterfaceFault
<operation>	InterfaceOperationElement	InterfaceOperation
<input>	InterfaceMessageReferenceElement	InterfaceMessageReference
<output>	InterfaceMessageReferenceElement	InterfaceMessageReference
<infault>	FaultReferenceElement	InterfaceFaultReference
<outfault>	FaultReferenceElement	InterfaceFaultReference
<binding>	BindingElement	Binding
<fault>	BindingFaultElement	BindingFault
<operation>	BindingOperationElement	BindingOperation
<input>	BindingMessageReferenceElement	BindingMessageReference
<output>	BindingMessageReferenceElement	BindingMessageReference
<infault>	FaultReferenceElement	BindingFaultReference
<outfault>	FaultReferenceElement	BindingFaultReference
<service>	ServiceElement	Service
<endpoint>	EndpointElement	Endpoint
<feature>	FeatureElement	Feature
<property>	PropertyElement	Property
Elementos XML Schema		
<xs:import>	ImportedSchema	
<xs:schema>	InlinedSchema	
<xs:element name="...">		ElementDeclaration
<xs:complexType name="...">		TypeDefinition

Tabela 6.1: Mapeamento dos Componentes

Capítulo 7

Conclusão

A proposta do trabalho era implementar um gerador automático de stubs em JavaScript a partir de um arquivo WSDL, para permitir que os desenvolvedores de aplicações web que queiram utilizar Web Services em suas aplicações clientes tenham mais uma linguagem como opção. Além disso, por ser uma linguagem nativa da web, o JavaScript possibilita uma maior compatibilidade com os padrões utilizados na web.

O gerador implementado em java faz a tradução de todos os componentes presentes no WSDL para um arquivo JavaScript o que permite que seja anexado as páginas web com facilidade. Durante o mapeamento de XML para JavaScript além dos objetos contendo os elementos usados nas transações, são gerados objetos para as funções que serão chamadas de forma local e objetos de suporte para que a mensagem SOAP possa ser montada e enviada para o servidor. Dessa forma o desenvolvedor utiliza métodos do servidor por meio de Web Services como se estivesse utilizando apenas funções de um JavaScript local adicionado à página na declaração dos scripts.

Como uma primeira estratégia, utilizamos uma abordagem bottom-up em que escolhemos um Web Service específico. Escrevemos um stub em Java Script manualmente e avaliamos os resultados da utilização do stub juntamente com o Web Service.

Ao longo do projeto pudemos adquirir uma visão global do mecanismo que envolve a troca de mensagens síncronas e assíncronas por meio de Web Services. Com isso conseguimos realizar uma prova de conceito do processo de tradução de um WSDL para outras linguagens. Esse protótipo, feito de forma manual, se mostrou essencial para que fosse possivel a automatização deste processo.

Concluimos então que apesar das dificuldades enfrentadas para a automatização da tradução do WSDL o gerador está atuando de forma satisfatória e se mostrou uma ferramenta util para desenvolvedores que desejem fazer chamadas a Web Services por meio da linguagem JavaScript.

7.1 Problemas Enfrentados

O primeiro problema encontrado foi o fato de apenas o Internet Explorer suportar XMLHTTPRequest entre domínios diferentes. Essa função é essencial para a comunicação com Web Services via AJAX. Essa restrição é devido a política “Same-origin policy”, no qual após uma conferência de segurança foi acordado

que scripts poderiam executar em páginas originárias de um mesmo domínio e utilizar livremente métodos e propriedades entre outras páginas, porém não podem realizar chamadas a métodos e propriedades de domínios diferentes.

Apesar de o gerador estar implementado de forma que todos os browsers poderiam utilizar Web Services, pela restrição de “cross-Domain”as requisições nao são enviadas nem recebidas com sucesso.

Um segundo ponto de dificuldade foi o fato de no Internet Explorer a API DOM não dar suporte a namespace. Com isso métodos como getElementsByTagNameNS() e getAttributerNodeNS()e propriedades como prefix, nameSpaceURI e baseURI não puderam ser utilizados, fazendo com que a estratégia para se mapear os elementos do XML tivesse que ser repensada pois não era possível tratar o namespace da mensagem SOAP.

Capítulo 8

Sugestões para trabalhos futuros

Ao longo do trabalho enfrentamos alguns problemas durante a implementação do gerador, tanto no tratamento da linguagem de origem como no bom funcionamento do código gerado em JavaScript. Muitos deles devido a limitações da linguagem alvo ou por convensões de boas práticas da internet. Com isso apesar de solucionarmos a maioria dos conflitos que surgiram, alguns, mesmo que não alterando o resultado final que foi proposto, ainda ficaram pendentes.

Deixamos como segestão para trabalhos futuros o estudo para tentar compatibilizar o stub gerado com todos os navegadoes de internet, resolvendo com isso a falta de suporte para o XMLHTTPRequest.

Outro ponto interessante seria tentar melhorar a integração do Internet Explorer com a API DOM de forma que este passasse a suportar o tratamento de namespace.

Por ulimo sugerimos ainda que fosse feito um tratamento mais específico para a tag <restriction> pois apesar de estar sendo mapeada para o JavaScript, o gerador nao restringe os valores como a tag sugere deixando isso a cargo do serviço que irá receber a requisição.

Referências Bibliográficas

[1] A. M. Software Services Web Services, **Description Language (WSDL) Explained**, Inc. Knowledge Base Whitepaper

[2] Canosa, John**Introduction to Web Services**Courtesy of Embedded Systems Programming Fev 1 2002

[3] CHRISTENSEN, Erik et al. **Web Services Description Language (WSDL) 1.1**. Microsoft, IBM Research, 2001.

[4] Dextra, **Web Services na Integração de Sistemas Corporativos** – <*http://www.dextra.com.br/empresa/artigos/webservices.htm*>. Acessado em: 15 out. 2009

[5] Engelen, Robert van**Code Generation Techniques for Developing LightWeight XML Web Services for Embedded Devices**.Department of Computer Science, Florida State University, Tallahassee

[6] Heshan T. Suriyaarachichi, <http://www.slideshare.net/heshans/lexical-analyzers-and-parsers-presentation>. Acessado em: 25 jun 2010

[7] Kaputin, John; Huges, Jeremy**Woden WSDL 2.0 Processor**

[8] Microsoft Corporation, **Understanding WSDL**.

[9] Ogbuji, Uche**Using WSDL in SOAP applications An introduction to WSDL for SOAP programmers**, Consultant, Fourthought, Inc. November 2000

[10] Skonnard, Aaron. **Understanding SOAP**, DevelopMentor, March 2003

[11] Sun Microsystems, **Building Web Services** – Sun ONE Studio 5 Programming Service, Revision A, Inc. 2003.

[12] W3C Note, **Web Services Description Language (WSDL) Version 2.0 Part 1**: Core Language

[13] <http://ws.apache.org/woden> - Acessado em: 24 jun. 2010

[14] <http://www.w3schools.com/>. Acessado em 15 mai 2010

[15] Wolter, Roger. **XML Web Services Basics** . Microsoft Corporation. December 2001

Capítulo 9

Anexos

9.1 WSDL Consulta CEP

```
<description xmlns="http://www.w3.org/ns/wsdl"
             targetNamespace="CEP"
             xmlns:soapenc="http://schemas.xmlsoap.org/soap/encoding/"
             xmlns:tm="http://microsoft.com/wsdl/mime/textMatching/"
             xmlns:s="http://www.w3.org/2001/XMLSchema"
             xmlns:wsoap="http://www.w3.org/ns/wsdl/soap"
             xmlns:tns="CEP"
             xmlns:http="http://schemas.xmlsoap.org/wsdl/http/"
             xmlns:mime="http://schemas.xmlsoap.org/wsdl/mime/"
             xmlns:soap12="http://schemas.xmlsoap.org/wsdl/soap12/">
  <types>
    <s:schema elementFormDefault="qualified" targetNamespace="CEP">
      <s:element name="ConsultaCEP">
        <s:complexType>
          <s:sequence>
            <s:element maxOccurs="1" minOccurs="0" name="CEP" type="s:string"/>
           </s:sequence>
        </s:complexType>
      </s:element>
      <s:element name="ConsultaCEPResponse">
        <s:complexType>
          <s:sequence>
            <s:element maxOccurs="1" minOccurs="0" name="ConsultaCEPResult"
           type="tns:CEP"/> </s:sequence>
        </s:complexType>
      </s:element>
      <s:complexType name="CEP">
        <s:sequence>
          <s:element maxOccurs="1" minOccurs="0" name="CEP" type="s:string"/>
          <s:element maxOccurs="1" minOccurs="0" name="UF" type="s:string"/>
          <s:element maxOccurs="1" minOccurs="0" name="Cidade" type="s:string"/>
          <s:element maxOccurs="1" minOccurs="0" name="Bairro" type="s:string"/>
```

```
      <s:element maxOccurs="1" minOccurs="0" name="TipoLogradouro" type="s:string"/>
      <s:element maxOccurs="1" minOccurs="0" name="Logradouro" type="s:string"/>
      <s:element maxOccurs="1" minOccurs="0" name="Mensagem" type="s:string"/>
      <s:element maxOccurs="1" minOccurs="1" name="Status" type="s:boolean"/>
    </s:sequence>
  </s:complexType>
  <s:element name="EstadosResponse">
    <s:complexType>
      <s:sequence>
        <s:element maxOccurs="1" minOccurs="0" name="EstadosResult"
        type="tns:ArrayOfEstados"/> </s:sequence>
    </s:complexType>
  </s:element>
  <s:complexType name="ArrayOfEstados">
    <s:sequence>
      <s:element maxOccurs="unbounded" minOccurs="0" name="Estados"
                                nillable="true" type="tns:Estados"/>
    </s:sequence>
  </s:complexType>
  <s:complexType name="Estados">
    <s:sequence>
      <s:element maxOccurs="1" minOccurs="1" name="ID_Estado" type="s:int"/>
      <s:element maxOccurs="1" minOccurs="0" name="CodigoEstado" type="s:string"/>
      <s:element maxOccurs="1" minOccurs="0" name="Estado" type="s:string"/>
    </s:sequence>
  </s:complexType>
  <s:element name="Cidades">
    <s:complexType>
      <s:sequence>
        <s:element maxOccurs="1" minOccurs="1" name="ID_Estado" type="s:int"/>
      </s:sequence>
    </s:complexType>
  </s:element>
  <s:element name="CidadesResponse">
    <s:complexType>
      <s:sequence>
        <s:element maxOccurs="1" minOccurs="0" name="CidadesResult"
                                        type="tns:ArrayOfCidades"/>
      </s:sequence>
    </s:complexType>
  </s:element>
  <s:complexType name="ArrayOfCidades">
    <s:sequence>
      <s:element maxOccurs="unbounded" minOccurs="0" name="Cidades"
                                nillable="true" type="tns:Cidades"/>
    </s:sequence>
  </s:complexType>
  <s:complexType name="Cidades">
    <s:sequence>
```

```
        <s:element maxOccurs="1" minOccurs="1" name="ID_Cidade" type="s:int"/>
        <s:element maxOccurs="1" minOccurs="0" name="Cidade" type="s:string"/>
      </s:sequence>
    </s:complexType>
    <s:element name="Bairros">
      <s:complexType>
        <s:sequence>
          <s:element maxOccurs="1" minOccurs="1" name="ID_Cidade" type="s:int"/>
        </s:sequence>
      </s:complexType>
    </s:element>
    <s:element name="BairrosResponse">
      <s:complexType>
        <s:sequence>
          <s:element maxOccurs="1" minOccurs="0" name="BairrosResult"
                                              type="tns:ArrayOfBairros"/>
        </s:sequence>
      </s:complexType>
    </s:element>
    <s:complexType name="ArrayOfBairros">
      <s:sequence>
        <s:element maxOccurs="unbounded" minOccurs="0" name="Bairros"
                                    nillable="true" type="tns:Bairros"/>
      </s:sequence>
    </s:complexType>
    <s:complexType name="Bairros">
      <s:sequence>
        <s:element maxOccurs="1" minOccurs="1" name="ID_Bairro" type="s:int"/>
        <s:element maxOccurs="1" minOccurs="0" name="Bairro" type="s:string"/>
      </s:sequence>
    </s:complexType>
    <s:element name="CEP" nillable="true" type="tns:CEP"/>
    <s:element name="ArrayOfEstados" nillable="true" type="tns:ArrayOfEstados"/>
    <s:element name="ArrayOfCidades" nillable="true" type="tns:ArrayOfCidades"/>
    <s:element name="ArrayOfBairros" nillable="true" type="tns:ArrayOfBairros"/>
  </s:schema>
</types>
<interface name="CEPWebServiceSoap">
  <operation name="ConsultaCEP"
             pattern="http://www.w3.org/ns/wsdl/in-out">
    <input messageLabel="In" element="tns:ConsultaCEP"/>
    <output messageLabel="Out" element="tns:ConsultaCEPResponse"/>
  </operation>
  <operation name="Estados"
             pattern="http://www.w3.org/ns/wsdl/in-out">
    <input messageLabel="In" element="tns:Estados"/>
    <output messageLabel="Out" element="tns:EstadosResponse"/>
  </operation>
  <operation name="Cidades"
```

```
            pattern="http://www.w3.org/ns/wsdl/in-out">
      <input messageLabel="In" element="tns:Cidades"/>
      <output messageLabel="Out" element="tns:CidadesResponse"/>
    </operation>
    <operation name="Bairros"
            pattern="http://www.w3.org/ns/wsdl/in-out">
      <input messageLabel="In" element="tns:Bairros"/>
      <output messageLabel="Out" element="tns:BairrosResponse"/>
    </operation>
  </interface>
  <interface name="CEPWebServiceHttpGet">
    <operation name="ConsultaCEP"
            pattern="http://www.w3.org/ns/wsdl/in-out">
      <input messageLabel="In"/>
      <output messageLabel="Out" element="tns:CEP"/>
    </operation>
    <operation name="Estados"
            pattern="http://www.w3.org/ns/wsdl/in-out">
      <input messageLabel="In" element="#none"/>
      <output messageLabel="Out" element="tns:ArrayOfEstados"/>
    </operation>
    <operation name="Cidades"
            pattern="http://www.w3.org/ns/wsdl/in-out">
      <input messageLabel="In"/>
      <output messageLabel="Out" element="tns:ArrayOfCidades"/>
    </operation>
    <operation name="Bairros"
            pattern="http://www.w3.org/ns/wsdl/in-out">
      <input messageLabel="In"/>
      <output messageLabel="Out" element="tns:ArrayOfBairros"/>
    </operation>
  </interface>
  <interface name="CEPWebServiceHttpPost">
    <operation name="ConsultaCEP"
            pattern="http://www.w3.org/ns/wsdl/in-out">
      <input messageLabel="In"/>
      <output messageLabel="Out" element="tns:CEP"/>
    </operation>
    <operation name="Estados"
            pattern="http://www.w3.org/ns/wsdl/in-out">
      <input messageLabel="In" element="#none"/>
      <output messageLabel="Out" element="tns:ArrayOfEstados"/>
    </operation>
    <operation name="Cidades"
            pattern="http://www.w3.org/ns/wsdl/in-out">
      <input messageLabel="In"/>
      <output messageLabel="Out" element="tns:ArrayOfCidades"/>
    </operation>
    <operation name="Bairros"
```

```
                pattern="http://www.w3.org/ns/wsdl/in-out">
      <input messageLabel="In"/>
      <output messageLabel="Out" element="tns:ArrayOfBairros"/>
    </operation>
  </interface>
  <binding name="CEPWebServiceSoap"
           interface="tns:CEPWebServiceSoap"
           type="http://www.w3.org/ns/wsdl/soap"
           wsoap:version="1.1"
           wsoap:protocol="http://www.w3.org/2006/01/soap11/bindings/HTTP">
    <operation ref="tns:ConsultaCEP" wsoap:action="CEP/ConsultaCEP">
    </operation>
    <operation ref="tns:Estados" wsoap:action="CEP/Estados">
    </operation>
    <operation ref="tns:Cidades" wsoap:action="CEP/Cidades">
    </operation>
    <operation ref="tns:Bairros" wsoap:action="CEP/Bairros">
    </operation>
  </binding>
  <service name="CEPWebService"
           interface="tns:CEPWebServiceSoap">
    <endpoint name="CEPWebServiceSoap"
              binding="tns:CEPWebServiceSoap"
              address="http://www.i-stream.com.br/webservices/cep.asmx">
    </endpoint>
  </service>
</description>
```

Código Fonte 9.1: WSDL Consulta CEP

9.2 Stub Gerado em JavaScript

```
var Schema = new function() {

   types = {

      this.WsdlObject = function(_name, _type, _value) {
         var name = _name;
         var type = _type;
         var value = _value;
         var attr = new Array();
         var elementsArray = new Array();

         this.getName = getName;
         this.getType = getType;
         this.getValue = getValue;
         this.getAttr = getAttr;
         this.setName = setName;
```

```
        this.setType = setType;
        this.setValue = setValue;
        this.setAttr = setAttr;
        this.addElement = addElement;

        this.getAnAttr = getAnAttr;
        this.setAnAttr = setAnAttr;
        this.newAttr = newAttr;

        function getName() {
           return name;
        }

        function getType() {
           return type;
        }

        function getValue() {
           return value;
        }

        function getAttr() {
           return attr;
        }

        function setName(_name) {
           name = _name;
        }

        function setType(_type) {
           type = _type;
        }

        function setValue(_value) {
           value = _value;
        }

        function setAttr(_attr) {
           attr = _attr;
        }

        function addElement(_obj) {
           elementsArray.push(_obj);
        }

        function getAnAttr(_name, _pos) {
           if (_pos == undefined) _pos = 0;
           if (_pos > attr.length) throw new Error (500, "Tentativa de
                                   recuperar uma posição inexistente");
```

```
        if (_pos == attr.length) {
            this.newAttr();
        }
        for (var p in attr[_pos]) {
            if (_name == attr[_pos][p].getName()) {
                return attr[_pos][p];
            }
        }
    }

    function setAnAttr(_attr, _name, _pos) {
        if (_pos == undefined) _pos = 0;
        if (_pos > attr.length) throw new Error (500, "Tentativa de fazer
                                        uma atribuição fora da ordem.");

        if (_pos == attr.length) {
            this.newAttr();
        }
        for (var p in attr[_pos]) {
            if (_name == attr[_pos][p].getName()) {
                attr[_pos][p] = _attr;
            }
        }
    }

    function newAttr() {
        var i = this.getAttr().length;
        this.getAttr()[i] = new Array();
        for (var j = 0; j < elementsArray.length; j++) {
            this.getAttr()[i].push(elementsArray[j]);
        }
    }
},

this.ArrayOfEstados = function(_name) {

    this.inheritFrom = types.WsdlObject;
    this.inheritFrom(_name, "Complex",undefined);
    this.addElement(new types.Estados("Estados"));

},

this.CEP = function(_name) {

    this.inheritFrom = types.WsdlObject;
    this.inheritFrom(_name, "Complex",undefined);
    this.addElement(new types.string("CEP"));
    this.addElement(new types.string("UF"));
```

```
        this.addElement(new types.string("Cidade"));
        this.addElement(new types.string("Bairro"));
        this.addElement(new types.string("TipoLogradouro"));
        this.addElement(new types.string("Logradouro"));
        this.addElement(new types.string("Mensagem"));
        this.addElement(new types.boolean("Status"));

    },

    this.Cidades = function(_name) {

        this.inheritFrom = types.WsdlObject;
        this.inheritFrom(_name, "Complex",undefined);
        this.addElement(new types.int("ID_Cidade"));
        this.addElement(new types.string("Cidade"));

    },

    this.Bairros = function(_name) {

        this.inheritFrom = types.WsdlObject;
        this.inheritFrom(_name, "Complex",undefined);
        this.addElement(new types.int("ID_Bairro"));
        this.addElement(new types.string("Bairro"));

    },

    this.ArrayOfCidades = function(_name) {

        this.inheritFrom = types.WsdlObject;
        this.inheritFrom(_name, "Complex",undefined);
        this.addElement(new types.Cidades("Cidades"));

    },

    this.Estados = function(_name) {

        this.inheritFrom = types.WsdlObject;
        this.inheritFrom(_name, "Complex",undefined);
        this.addElement(new types.int("ID_Estado"));
        this.addElement(new types.string("CodigoEstado"));
        this.addElement(new types.string("Estado"));

    },

    this.ArrayOfBairros = function(_name) {

        this.inheritFrom = types.WsdlObject;
        this.inheritFrom(_name, "Complex",undefined);
```

```
        this.addElement(new types.Bairros("Bairros"));

    },

    this.ENTITIES = function(_name) {

        this.inheritFrom = types.WsdlObject;
        this.inheritFrom(_name, "Simple",undefined);

    },

    this.language = function(_name) {

        this.inheritFrom = types.WsdlObject;
        this.inheritFrom(_name, "Simple",undefined);

    },

    this.unsignedByte = function(_name) {

        this.inheritFrom = types.WsdlObject;
        this.inheritFrom(_name, "Simple",undefined);

    },

    this.IDREFS = function(_name) {

        this.inheritFrom = types.WsdlObject;
        this.inheritFrom(_name, "Simple",undefined);

    },

    this.int = function(_name) {

        this.inheritFrom = types.WsdlObject;
        this.inheritFrom(_name, "Simple",undefined);

    },

    this.double = function(_name) {

        this.inheritFrom = types.WsdlObject;
        this.inheritFrom(_name, "Simple",undefined);

    },

    this.normalizedString = function(_name) {

        this.inheritFrom = types.WsdlObject;
```

```
        this.inheritFrom(_name, "Simple",undefined);

    },

    this.byte = function(_name) {

        this.inheritFrom = types.WsdlObject;
        this.inheritFrom(_name, "Simple",undefined);

    },

    this.gMonthDay = function(_name) {

        this.inheritFrom = types.WsdlObject;
        this.inheritFrom(_name, "Simple",undefined);

    },

    this.base64Binary = function(_name) {

        this.inheritFrom = types.WsdlObject;
        this.inheritFrom(_name, "Simple",undefined);

    },

    this.time = function(_name) {

        this.inheritFrom = types.WsdlObject;
        this.inheritFrom(_name, "Simple",undefined);

    },

    this.ENTITY = function(_name) {

        this.inheritFrom = types.WsdlObject;
        this.inheritFrom(_name, "Simple",undefined);

    },

    this.ID = function(_name) {

        this.inheritFrom = types.WsdlObject;
        this.inheritFrom(_name, "Simple",undefined);

    },

    this.boolean = function(_name) {

        this.inheritFrom = types.WsdlObject;
```

```
        this.inheritFrom(_name, "Simple",undefined);

    },

    this.NOTATION = function(_name) {

        this.inheritFrom = types.WsdlObject;
        this.inheritFrom(_name, "Simple",undefined);

    },

    this.nonPositiveInteger = function(_name) {

        this.inheritFrom = types.WsdlObject;
        this.inheritFrom(_name, "Simple",undefined);

    },

    this.NCName = function(_name) {

        this.inheritFrom = types.WsdlObject;
        this.inheritFrom(_name, "Simple",undefined);

    },

    this.anyURI = function(_name) {

        this.inheritFrom = types.WsdlObject;
        this.inheritFrom(_name, "Simple",undefined);

    },

    this.gDay = function(_name) {

        this.inheritFrom = types.WsdlObject;
        this.inheritFrom(_name, "Simple",undefined);

    },

    this.float = function(_name) {

        this.inheritFrom = types.WsdlObject;
        this.inheritFrom(_name, "Simple",undefined);

    },

    this.long = function(_name) {

        this.inheritFrom = types.WsdlObject;
```

```
        this.inheritFrom(_name, "Simple",undefined);

    },

    this.nonNegativeInteger = function(_name) {

        this.inheritFrom = types.WsdlObject;
        this.inheritFrom(_name, "Simple",undefined);

    },

    this.string = function(_name) {

        this.inheritFrom = types.WsdlObject;
        this.inheritFrom(_name, "Simple",undefined);

    },

    this.IDREF = function(_name) {

        this.inheritFrom = types.WsdlObject;
        this.inheritFrom(_name, "Simple",undefined);

    },

    this.positiveInteger = function(_name) {

        this.inheritFrom = types.WsdlObject;
        this.inheritFrom(_name, "Simple",undefined);

    },

    this.Name = function(_name) {

        this.inheritFrom = types.WsdlObject;
        this.inheritFrom(_name, "Simple",undefined);

    },

    this.duration = function(_name) {

        this.inheritFrom = types.WsdlObject;
        this.inheritFrom(_name, "Simple",undefined);

    },

    this.unsignedLong = function(_name) {

        this.inheritFrom = types.WsdlObject;
```

```
        this.inheritFrom(_name, "Simple",undefined);

    },

    this.integer = function(_name) {

        this.inheritFrom = types.WsdlObject;
        this.inheritFrom(_name, "Simple",undefined);

    },

    this.negativeInteger = function(_name) {

        this.inheritFrom = types.WsdlObject;
        this.inheritFrom(_name, "Simple",undefined);

    },

    this.unsignedInt = function(_name) {

        this.inheritFrom = types.WsdlObject;
        this.inheritFrom(_name, "Simple",undefined);

    },

    this.decimal = function(_name) {

        this.inheritFrom = types.WsdlObject;
        this.inheritFrom(_name, "Simple",undefined);

    },

    this.token = function(_name) {

        this.inheritFrom = types.WsdlObject;
        this.inheritFrom(_name, "Simple",undefined);

    },

    this.hexBinary = function(_name) {

        this.inheritFrom = types.WsdlObject;
        this.inheritFrom(_name, "Simple",undefined);

    },

    this.short = function(_name) {

        this.inheritFrom = types.WsdlObject;
```

```
        this.inheritFrom(_name, "Simple",undefined);

    },

    this.gYear = function(_name) {

        this.inheritFrom = types.WsdlObject;
        this.inheritFrom(_name, "Simple",undefined);

    },

    this.gMonth = function(_name) {

        this.inheritFrom = types.WsdlObject;
        this.inheritFrom(_name, "Simple",undefined);

    },

    this.dateTime = function(_name) {

        this.inheritFrom = types.WsdlObject;
        this.inheritFrom(_name, "Simple",undefined);

    },

    this.QName = function(_name) {

        this.inheritFrom = types.WsdlObject;
        this.inheritFrom(_name, "Simple",undefined);

    },

    this.NMTOKENS = function(_name) {

        this.inheritFrom = types.WsdlObject;
        this.inheritFrom(_name, "Simple",undefined);

    },

    this.gYearMonth = function(_name) {

        this.inheritFrom = types.WsdlObject;
        this.inheritFrom(_name, "Simple",undefined);

    },

    this.NMTOKEN = function(_name) {

        this.inheritFrom = types.WsdlObject;
```

```
        this.inheritFrom(_name, "Simple",undefined);

    },

    this.date = function(_name) {

        this.inheritFrom = types.WsdlObject;
        this.inheritFrom(_name, "Simple",undefined);

    },

    this.unsignedShort = function(_name) {

        this.inheritFrom = types.WsdlObject;
        this.inheritFrom(_name, "Simple",undefined);

    }


};

this.inheritFrom = types.WsdlObject;
this.inheritFrom("schema","Complex",undefined);

this.addElement(new types.ArrayOfEstados("ArrayOfEstados"));
this.addElement(new types.CEP("CEP"));
this.addElement(new function() {

    this.inheritFrom = types.WsdlObject;
    this.inheritFrom("BairrosResponse", "Complex",undefined);
    this.addElement(new types.ArrayOfBairros("BairrosResult"));

});

this.addElement(new function() {

    this.inheritFrom = types.WsdlObject;
    this.inheritFrom("Cidades", "Complex",undefined);
    this.addElement(new types.int("ID_Estado"));

});

this.addElement(new function() {

    this.inheritFrom = types.WsdlObject;
    this.inheritFrom("EstadosResponse", "Complex",undefined);
    this.addElement(new types.ArrayOfEstados("EstadosResult"));

});
```

```
    this.addElement(new types.ArrayOfCidades("ArrayOfCidades"));
    this.addElement(new function() {

        this.inheritFrom = types.WsdlObject;
        this.inheritFrom("Bairros", "Complex",undefined);
        this.addElement(new types.int("ID_Cidade"));

    });

    this.addElement(new function() {

        this.inheritFrom = types.WsdlObject;
        this.inheritFrom("CidadesResponse", "Complex",undefined);
        this.addElement(new types.ArrayOfCidades("CidadesResult"));

    });

    this.addElement(new function() {

        this.inheritFrom = types.WsdlObject;
        this.inheritFrom("ConsultaCEPResponse", "Complex",undefined);
        this.addElement(new types.CEP("ConsultaCEPResult"));

    });

    this.addElement(new types.ArrayOfBairros("ArrayOfBairros"));
    this.addElement(new function() {

        this.inheritFrom = types.WsdlObject;
        this.inheritFrom("ConsultaCEP", "Complex",undefined);
        this.addElement(new types.string("CEP"));

    };


};


CEPWebService = {
    namespace: "CEP",

    CEPWebServiceSoap: {

        address: "http://www.i-stream.com.br/webservices/cep.asmx",

        soapVersion: "1.1",

        ConsultaCEP: function (body, outMethod) {
```

```
        var operation = "ConsultaCEP";
        var output = "ConsultaCEPResponse";
        var action = "CEP/ConsultaCEP";
        var outParam = Schema.getAnAttr("ConsultaCEPResponse");
        SOAPActions.call(CEPWebService.CEPWebServiceSoap.address,
                         operation,
                         output,
                         outMethod,
                         CEPWebService.namespace,
                         action,
                         true,
                         body,
                         outParam,
                         CEPWebService.CEPWebServiceSoap.soapVersion);
      },

    Estados: function (body, outMethod) {
        var operation = "Estados";
        var output = "EstadosResponse";
        var action = "CEP/Estados";
        var outParam = Schema.getAnAttr("EstadosResponse");
        SOAPActions.call(CEPWebService.CEPWebServiceSoap.address,
                         operation,
                         output,
                         outMethod,
                         CEPWebService.namespace,
                         action,
                         true,
                         body,
                         outParam,
                         CEPWebService.CEPWebServiceSoap.soapVersion);
      },

    Cidades: function (body, outMethod) {
        var operation = "Cidades";
        var output = "CidadesResponse";
        var action = "CEP/Cidades";
        var outParam = Schema.getAnAttr("CidadesResponse");
        SOAPActions.call(CEPWebService.CEPWebServiceSoap.address,
                         operation,
                         output,
                         outMethod,
                         CEPWebService.namespace,
                         action,
                         true,
                         body,
                         outParam,
                         CEPWebService.CEPWebServiceSoap.soapVersion);
      },
```

```
      Bairros: function (body, outMethod) {
         var operation = "Bairros";
         var output = "BairrosResponse";
         var action = "CEP/Bairros";
         var outParam = Schema.getAnAttr("BairrosResponse");
         SOAPActions.call(CEPWebService.CEPWebServiceSoap.address,
                          operation,
                          output,
                          outMethod,
                          CEPWebService.namespace,
                          action,
                          true,
                          body,
                          outParam,
                          CEPWebService.CEPWebServiceSoap.soapVersion);
          },

   },
}



SOAPActions = {

   call: function(location,
                  operation,
                  operationResponse,
                  outMethod,
                  namespace,
                  soapaction,
                  async,
                  inParam,
                  outParam,
                  soapVersion) {
      if(async) {
         SOAPActions.loadMessage (location,
                                  operation,
                                  operationResponse,
                                  outMethod,
                                  namespace,
                                  soapaction,
                                  async,
                                  inParam,
                                  outParam,
                                  soapVersion);
     } else {
         return SOAPActions.loadMessage (location,
                                  operation,
```

```
                                    operationResponse,
                                    outMethod,
                                    namespace,
                                    soapaction,
                                    async,
                                    inParam,
                                    outParam,
                                    soapVersion);
    }
},

loadMessage: function(location,
                      operation,
                      operationResponse,
                      outMethod,
                      namespace,
                      soapaction,
                      async,
                      inParam,
                      outParam,
                      soapVersion) {
    if (soapVersion = "1.1") {
        var message =
        "<?xml version=\"1.0\" encoding=\"utf-8\"?>" +
        "<soap:Envelope " +
        "xmlns:xsi=\"http://www.w3.org/2001/XMLSchema-instance\" " +
        "xmlns:xsd=\"http://www.w3.org/2001/XMLSchema\" " +
        "xmlns:soap=\"http://schemas.xmlsoap.org/soap/envelope/\">" +
        "<soap:Body>" +
        "<" + operation + " xmlns=\"" + namespace + "\">" +
        Utils.objectToXml(inParam) +
        "</" + operation + "></soap:Body></soap:Envelope>";
    } else if (soapVersion = "1.2") {
        var message =
        "<?xml version=\"1.0\" encoding=\"utf-8\"?>" +
        "<soap12:Envelope " +
        "xmlns:xsi=\"http://www.w3.org/2001/XMLSchema-instance\" " +
        "xmlns:xsd=\"http://www.w3.org/2001/XMLSchema\" " +
        "xmlns:soap12=\"http://www.w3.org/2003/05/soap-envelope\">" +
        "<soap12:Body>" +
        "<" + operation + " xmlns=\"" + namespace + "\">" +
        Utils.objectToXml(inParam) +
        "</" + operation + "></soap12:Body></soap12:Envelope>";
    }
    else throw new Error (500, "Versão SOAP para a construção da mensagem não
                                                            identificada");

    var xmlHttp = Utils.getXmlHttp();
    xmlHttp.open("POST", location, async);
```

```
      xmlHttp.setRequestHeader("SOAPAction", soapaction);
      xmlHttp.setRequestHeader("Content-Type", "text/xml; charset=utf-8");
      if(async) {
         xmlHttp.onreadystatechange = function() {
            if(xmlHttp.readyState == 4) {
               SOAPActions.loadedMessage(operation,
                                         operationResponse,
                                         outMethod,
                                         async,
                                         xmlHttp,
                                         outParam);
            }
         }
      }
      xmlHttp.send(message);
      if (!async) {
         return SOAPActions.loadedMessage(operation,
                                         operationResponse,
                                         outMethod,
                                         async,
                                         xmlHttp,
                                         outParam);
      }
   },

   // Quando o readyState for ok (4), Trabalhamos em cima da resposta e
   //chamamos o método OutMethod que foi definido pelo usuário

   loadedMessage: function(operation,
                           operationResponse,
                           outMethod,
                           async,
                           xmlHttp,
                           outParam); {

      // Aloca a resposta SOAP em uma estrutura de Nó utilizando a API DOM.
      var ResponseNode = Utils.getNode(xmlHttp, operationResponse);
      //Converte a estrutura de Nó para a estrutura de dados de outParam (Objeto)
      outParam = Utils.nodeToObject(ResponseNode, outParam);
      // E retorna a resposta
      outMethod(outParam);
      if(!async) {
         return outParam;
      }
   }
}


Utils = {
```

```
isArray: function(object) {
   return object != null && typeof object == "object" && 'splice' in object
                                                        && 'join' in object; },

samePreviousSibling: function(node) {
   try {
      return (node.nodeName == node.previousSibling.nodeName);
   }
   catch (ex) {}
   return false;
},

sameNextSibling: function(node) {
   try {
      return (node.nodeName == node.nextSibling.nodeName);
   }
   catch (ex) {}
   return false;
},

objectToXml: function (obj) {
   xmlStr = "";
   for (var i in obj.getAttr()[0]) {
      xmlStr += Utils.buildTag (obj.getAttr()[0][i]);
   }
   return xmlStr;
},

buildTag: function (obj) {
   var strTag = "";
   if (obj.getType() == "Complex") {
      if (obj.getAttr().length > 0) {
         strTag += "<" + obj.getName() + ">";
         for (var p in obj.getAttr()) {
            for (var i in obj.getAttr()[p]) {
               strTag += Utils.buildTag (obj.getAttr()[p][i]);
            }
         }
         strTag += "</" + obj.getName() + ">";
      }
   } else {
      if (obj.getValue() != undefined) {
         strTag += "<" + obj.getName() + ">";
         strTag += obj.getValue().replace(/&/g,"&amp;").
                        replace(/</g,"&lt;").replace(/>/g, "&gt;");

         strTag += "</" + obj.getName() + ">";
      }
```

```
    }
    return strTag;
},

getNodeLocalName: function (node) {
    return node.nodeName.substring(node.nodeName.indexOf(":") + 1);
},

isLeafNode: function(node) {
    return (node.childNodes.length == 1 && (node.childNodes[0].nodeType == 3
                                    || node.childNodes[0].nodeType == 4)); },

getSameNodesPosition: function(node) {
    var result = 0;
    while (Utils.samePreviousSibling(node)) {
        node = node.previousSibling;
        result++;
    }
    return result;
},

nodeToObject: function(node, outParam) {
    if (node == null) {
        return null;
    }
    // Se o nó for do tipo texto, então retorna seu valor final
    if (node.nodeType == 3 || node.nodeType == 4) {
        return Utils.getValue(node, outParam);
    }
    nodeLocalName = Utils.getNodeLocalName(node);
    // Apenas continua se o nodeName existir no objeto outParam, caso
   // contrário ocorre um erro

    if (outParam.getName() == nodeLocalName) {
        // Se for um nó folha retorna o m?todo recursivamente chamando o nó
        //filho para o mesmo outParam

        if (Utils.isLeafNode(node)) {
            return Utils.nodeToObject(node.childNodes[0], outParam);
        }
        for(var i = 0; i < node.childNodes.length; i++) {
            var pos = Utils.getSameNodesPosition (node.childNodes[i]);
            var p = Utils.nodeToObject(node.childNodes[i],
                        outParam.getAnAttr(Utils.getNodeLocalName(node.childNodes[i]),
                        pos));
          outParam.setAnAttr(p, Utils.getNodeLocalName(node.childNodes[i]), pos);
        }
        return outParam;
    }
```

```
    throw new Error (500, "Erro na convers?o de n?s para objetos");
},

getValue: function(node, outParam) {
    var value = node.nodeValue;
    if (outParam.getType() == "Simple") {
        outParam.setValue(value);
        return outParam;
    } else if (outParam.getType() == "Complex") {
        throw new Error (500, "Não é possível retorna o valor de um ComplexType")
    }
},

// Retorna o nó que possui o local name passado por parâmetro, caso exista
//mais de um os namespaces seriam necess?rios e ocasionará em um erro, visto
//que o IE não trabalha com namespaces.

getNodeByLocalName: function (nodeList, localName) {
    var qtd = 0;
    for (i=0; i < nodeList.length; i++) {
        if (nodeList[i].nodeName.substring(nodeList[i].nodeName.indexOf(":") +
                                                        1).match(localName)) {
            qtd = qtd + 1;
            node = nodeList[i];
        }
    }
    if (qtd < 1) throw new Error(500, "Tentativa de recuperar um n? inexistente.");
    if (qtd > 1) throw new Error(500, "Existe mais de um n? para este LocalName.");
    return node;
},

// Ajuda a construir uma estrutura de N? a partir da um xml.
//getElementsByTagName não funciona no IE6 e anteriores.

getNode: function(xml, tag) {
    try {
        nodeList = xml.responseXML.selectNodes(".//*[local-name()=\""+ tag +"\"]");
        return Utils.getNodeByLocalName(nodeList, tag);
    }
    catch (ex) {}
    nodeList = xml.responseXML.getElementsByTagName('*');
    return Utils.getNodeByLocalName(nodeList, tag);
},

//Testa os parsers para uma compatibilidade com diversos browsers.
getXmlHttp: function() {
    try {
        if (window.XMLHttpRequest) {
            var req = new XMLHttpRequest();
```

```
            if (req.readyState == null) {
               req.readyState = 1;
               req.addEventListener("load", function() {
                  req.readyState = 4;
                  if (typeof req.onreadystatechange == "function") {
                     req.onreadystatechange();
                  }
               },
               false);
            }
            return req;
         }
         if (window.ActiveXObject) {
            var parsers = ["Msxml2.XMLHTTP.5.0",
                           "Msxml2.XMLHTTP.4.0",
                           "MSXML2.XMLHTTP.3.0",
                           "MSXML2.XMLHTTP",
                           "Microsoft.XMLHTTP"];
            for(var i = 0; i < parsers.length; i++) {
               try {
                  return new ActiveXObject(parsers[i]);
               }
               catch (ex) {};
            }
            throw new Error("Imposs?vel encontrar um Xml parser instalado");
         }
      }
      catch (ex) {}
      throw new Error("Este navegador n?o suporta objectos XmlHttp");
   }
}
```

Código Fonte 9.2: Stub Consulta CEP