FUNDAÇÃO EDSON QUEIROZ
UNIVERSIDADE DE FORTALEZA - UNIFOR

JAVA E XML

APIs PARA TRATAMENTO DE DOCUMENTOS XML EM APLICAÇÕES WEB.

Rafael de Oliveira Lopes

MONOGRAFIA APRESENTADA AO CENTRO DE CIÊNCIAS TECNOLÓGICAS DA UNIVERSIDADE DE FORTALEZA COMO PARTE DOS REQUISITOS PARA A OBTENÇÃO DO GRAU DE BACHAREL EM INFORMÁTICA.

Aprovada por: _____

Prof. Orientador Pedro Porfírio Muniz Farias

_____

Prof. Nabor das Chagas Mendonça

_____

Prof. Ângelo Roncalle Alencar Brayner

Fortaleza, CE – Brasil.

Dezembro, 2002.

LOPES, Rafael de Oliveira

JAVA E XML - APIs PARA TRATAMENTO DE DOCUMENTOS XML EM APLICAÇÕES WEB.

vii 79 p. 29,7 cm (INFORMÁTICA/UNIFOR, Programação, 2002)

Monografia – Universidade de Fortaleza, Informática

1. Java
2. XML
3. Application Programming Interface
4. Web Services
5. Java APIs for XML

I. INFORMÁTICA/UNIFOR
II. TÍTULO (série)

Dedico este trabalho a meu pai e a minha mãe, meus maiores mestres.

SUMÁRIO

INTRODUÇÂO 1
1. INTRODUÇÃO A XML 3
1.1. Características da XML 3
1.2. Sintaxe de um Documento XML 5
1.3 Validação de documentos XML 7
1.3.1. Document Type Definitions (*DTD*) 8
1.3.1.1. Declaração de Elementos 8
1.3.1.2. Declaração de atributos 9
1.3.2. Document Type Declarations 13
2. SAX - SIMPLE API FOR XML 14
2.1. Operações básicas no tratamento de documentos XML 14
2.2. O que é SAX? 16
2.3.1. Importando classes 18
2.3.2. Instanciando um parser 20
2.3.3. Implementando um content handler 22
3. DOM - DOCUMENT OBJECT MODEL 26
3.1. Document Object Model 26
3.2. Qual Linguagem utilizar? 27
3.3. Entendendo melhor o DOM 27
3.4. Implementando um exemplo com DOM 30
3.4.1. Classes necessárias à implementação 30
3.4.2. Classe principal: InvoiceCheckerDOM 31
3.5. Document Object Model versus Simple API for XML 34
3.6. Comparação final 35

4. JAVA APIs PARA XML.....37
4.1. JAVA versus XML 37
4.2. Java ApplicationProgramming Interfaces for XML 38
4.2.1. Java API for XML Processing (JAXP) 38
4.2.1.1. Transformando dados XML com XSLT.....40
4.2.2. Java API for XML-based RPC (JAX-RPC).....45
4.2.2.1. Utilizazndo JAX-RPC.....46
4.2.3. Java API for XML Messaging 49
4.2.3.1. Utilizando JAXM 52
4.2.4. Java API for XML Registries 56
4.2.4.1. Arquitetura JAXR 56
4.2.4.2. Utilizando JAXR.....58
CONCLUSÂO 66
BIBLIOGRAFIA 68

LISTA DE FIGURAS

Figura 2.1. Arquitetura básica de processamento de XML 15
Figura 3.1. Estrutura de um documento XML gerado por um DOM *parser* 28
Figura 3.2. Modelo UML das classes e interfaces utilizadas no Document Object Model..... 29
Figura 4.2. Representação da chamada a procedimentos remotos utilizando JAX-RPC 47
Figura 4.4. Objeto SOAPMessage sem anexos 51
Figura 4.4. Objeto SOAPMessage sem anexos 51
Figura 4.5. Objeto SOAPMessage com anexos 52
Figura 4.6. Arquitetura JAXR 58

LISTA DE TABELAS

Tabela 3.1. Pacotes utilizados por Java na implementação do modelo DOM.....30
Tabela 4.1. Classes do pacote javax.xml.registry.....57

RESUMO

Este trabalho analisará as formas de utilização conjunta do XML, padrão da W3C para intercâmbio de dados semi-estruturados na Web e da linguagem de programação Java, proposta pela *SUN*.

Uma das principais características da linguagem Java é de permitir código portável entre diversas plataformas, assim como documentos no formato XML. Essa semelhança entre essas duas tecnologias facilita o desenvolvimento de sistemas para a *web*.

XML e Java são tecnologias que facilitam o desenvolvimento de *web services*. XML se destina a formatação e identificação dos dados que serão transportados via *web* e a partir daí foi se tornando padrão mundial. Qualquer aplicação que trate de envio de dados via *web* deve estar apta a receber dados no formato de documentos XML.

Java cuida da validação e da alteração, inserção e exclusão de dados existentes em documentos XML. *Application Programming Interfaces* foram desenvolvidas na linguagem de programação Java para trazer aos desenvolvedores uma maior facilidade ao desenvolverem aplicações baseadas em XML.

Este trabalho apresentará *Application Programming Interfaces* que foram desenvolvidas na linguagem de programação Java para trazer aos desenvolvedores uma maior facilidade ao desenvolverem aplicações baseadas em XML. Tais APIs foram agrupadas na JAXP (*Java API for XML Processing*).

Nesta monografia, inicialmente analisaremos as APIs que compõem a JAXP. Focalizaremos a atenção nas APIs mais utilizadas: SAX e DOM, em seguida outras APIs que cuidam de operações realizadas com documentos XML: *Java API for XML Messaging*, *Java API for XML Registries* e *Java API for XML-based RPC*.

INTRODUÇÃO

Atualmente aplicações e documentos são disponibilizados na *web*. O acesso a esses documentos e aplicações é geralmente realizado por pessoas através de *browsers*, no entanto, a partir do momento em que houver uma forte integração entre as diferentes aplicações disponíveis na *web*, permitindo a troca de informações entre sistemas de computação, o grau de utilização da *web* será lançado a outro patamar, onde uma profusão de novos serviços estará disponível.

No intuito de se integrar as várias aplicações que são criadas para *web*, padrões foram criados a fim de prover essa integração e manter uma relação custo-benefício razoável para organizações que desenvolvem essas aplicações. É a partir dessa idéia de integração que surge o conceito de *web services*. *Web service* é uma tecnologia que propõe a utilização de um modelo comum de comunicação entre aplicações baseados em padrões como HTTP (Hypertext Transport Protocol), XML (*eXtensible Markup Language*), SOAP (*Simple Object Access Protocol*) entre outros.

Para a criação de *web services* a plataforma J2EE apresenta várias características como portabilidade. J2EE é uma pataforma de desenvolvimento para aplicações distribuídas e que desde o começo da linguagem Java essa plataforma veio ganhando adeptos no mundo inteiro. Várias APIs (*Application Programing Interface*) que foram desenvolvidas na linguagem de programação Java fazem o tratamento de documentos XML que se tornaram o padrão de formatação de dados na *web*.

XML é a base dos protocolos utilizados no desenvolvimento de aplicações *web* e é utilizado para a marcação de documentos que contém informações estruturadas. No primeiro capítulo é apresentada uma introdução sobre as características e a estrutura de documentos XML.

O capítulo dois apresenta um modelo de *parsing,* a *Simple API for XML. Parsing* é o processo de validação de documentos, como XML. SAX é baseado em eventos, ou seja, a validação do documento XML se dá na medida em que vai sendo lido por um *stream* que é uma das estruturas (mais especificamente um objeto) da linguagem Java.

O capítulo três aborda outro modelo de *parsing*, o *Document Object Model.* Nesse modelo é criada uma estrutura de dados que armazena todo o documento XML que foi processado. Cada um desses modelos possui suas características e se aplicam muito bem a várias aplicações dependendo do porte de documentos XML que serão utilizados. Geralmente os dois modelos são utilizados.

O capítulo quatro aborda as Java APIs para XML. Dentre essas APIs estão JAXP (*Java API for XMl processing*), JAXR (*Java API for XML Registries*), JAXM (*Java API for XML Messaging*) e JAX-RPC (*Java XML for RPC*).

1. INTRODUÇÃO A XML

Alguns dos problemas encontrados por vários desenvolvedores de aplicações estão ligados ao fato de como devem tratar arquivos com informações necessárias ao processamento de suas atividades. Muitos questionamentos do tipo “Como adequar a minha aplicação para que ela seja capaz de interagir com um formato qualquer de dados?”, são feitos por diversos desenvolvedores. Muitos desenvolvedores, levando em consideração esse tipo de questionamento, passaram a buscar um modelo de representação de dados que se tornasse um padrão e pudesse ser utilizado por desenvolvedores do mundo inteiro, e a partir daí suas aplicações poderiam interagir do modo mais simples possível.

Vários formatos de dados foram criados, porém um formato de representação de dados que à primeira vista parece complexo para muitos desenvolvedores, é sem dúvida mais robusto, extensível e muito mais simples de se trabalhar.(GABRICK, 2000) Esse formato é a *Extended Markup Language*, mais conhecida por XML, e é o enfoque deste capítulo.

XML é um formato de texto simples e bastante flexível. Foi desenvolvido originalmente para se adaptar às mudanças de publicação de documentos na *web*. XML também se propõe a facilitar no transporte de documentos e dados via *web*.

1.1. Características da XML.

Dentre várias características da XML, as mais importantes são:

- Robustez:

Por exemplo, um dado arquivo contém suas informações armazenadas em linhas distintas. Uma aplicação que utilize esse mesmo arquivo pode chegar a ter problemas caso uma informação que a aplicação espere acessar esteja em uma posição diferente da que deveria estar, pois não sabemos o que aquele valor representa senão pela sua posição no arquivo. Segundo Didier Matin (MARTIN, 2000), a representação em XML é bem mais robusta, pois a partir dela a aplicação identifica que tipo de dado é aquele através da representação do elemento e de seus atributos no documento. Além disso, qualquer erro que possa a vir acontecer com o arquivo ou com o dado que deve ser utilizado pode ser tratado, causando menores danos à aplicação.

- Portabilidade:

Uma das características mais importantes de documentos XML é o fato de serem portáveis. Documentos XML são simples textos que podem ser movidos entre várias plataformas sem nenhum problema, pois o fato de XML seguir um padrão para todas aplicações que fazem uso dos seus recursos, torna muito mais fácil o seu ajuste a qualquer plataforma.

Uma aplicação recebe um arquivo XML já possui a programação necessária para tratar um documento desse tipo sem complicações sobre o tipo de formato, pois XML é um padrão. Existe um ditado que diz “código portável, dados portáveis” em referência a combinação entre Java e XML, e a utilização dessa combinação está crescendo e sendo aplicada por desenvolvedores de todo o mundo.

- Fácil de Usar

Um documento XML, além de todas essas características, ainda é muito mais fácil de ser usado por desenvolvedores do que arquivos que possuam a identificação de seus dados separados em linhas distintas. A parte mais complicada de tratamentos de documentos XML se refere à validação do documento, e esta árdua tarefa fica a cargo de um *software* (*parser*) que é capaz de encontrar erros de sintaxe no documento além de identificar todos os elementos existentes no mesmo. Um *parser* pode ser desenvolvido por programadores da própria aplicação ou utilizar uma solução de *parser* já desenvolvida. Nos capítulos 2 e 3 serão apresentados dois modelos, SAX e DOM, utilizados para o processo de *parsing* de um documento XML.

1.2. Sintaxe de um Documento XML

Os documentos XML possuem uma especificação feita pela *Worldwide Web Consortium* (W3C). Essa especificação é feita baseada em uma gramática BNF (*Backus Normal Form*). (GRAHAM,2002) O fato de um documento XML ser bem formado é um requerimento mínimo para aceitação e utilização do mesmo.

A estrutura de um documento XML é representada por uma árvore, com nó raiz e vários nós folhas. A árvore XML possui vários tipos de nós:

- Raiz:

É um nó abstrato que possui como conteúdo todo o documento XML.

- Elemento:

É a unidade fundamental do arquivo XML. Qualquer documento XML possui pelo menos um elemento. Um elemento é composto por um nome, por atributos, por *namespaces* e pelo conteúdo do elemento. Os elementos são definidos por *tags*. Por exemplo, um elemento chamado *item* é composto por uma *tag* inicial, seu conteúdo e uma *tag* final.

<item ***atributos*****>** ***conteúdo*** *</item>*

A parte de conteúdo do elemento é opcional. Elementos podem conter outros elementos, desde que estejam corretamente aninhados como *<item> <descricao> </descricao> </item>*, e não: *<item> <descricao> </item> </descricao>*.

Segundo Didier Martin (MARTIN, 2000), elementos são os *containers* para o conteúdo do documento XML. Eles representam objetos discretos do documento, e por isso podem ser conhecidos como sendo "substantivos" para o documento XML.

- Texto:

Documentos XML são formados por uma seqüência de caracteres que seguem o conjunto de caracteres *Unicode*. Os documentos XML podem ser escritos em qualquer conjunto de caracteres desde que o *parser* conheça esse conjunto e saiba como convertê-los em *Unicode*.

- Atributos:

Podemos dizer que o atributo é um par formado por um nome e um valor. Nome e valores devem seguir o padrão definido a eles por XML. Os valores devem usar caracteres de escape para elementos como <, >e ". Como exemplo, abaixo é definido um atributo *qtd* para o elemento *item*: *<item qtd="5">*. Não existe ordem entre os atributos que definem um elemento.

O elemento *<item qtd="5" precoUnit="19.90" ></item>* é o mesmo elemento *<item precoUnit="19.90" qtd="5"></item>*.

- Comentários:

Não há mistério na definição de comentários em documentos XML. A definição é a mesma utilizada em HTML (*Hypertext Makup Language*). Comentários seguem a seguinte notação: *<!-- comentários -->*. *Parsers* procuram ignorar os comentários definidos em um documento XML, mas alguns deles podem tornar disponíveis caso desejado.

- Instruções de Processamento:

Esses elementos são usados pra indicar a um *software* em particular, como deve ser tratado o documento XML depois de ser validado por um *parser*. Geralmente são usadas para meta-informação que se aplicam a muitos documentos.(GABRICK, 2000) Um exemplo de instrução de processamento é a instrução *xml-stylesheet* que avisa ao *browser* onde se encontra a folha de estilo que deve ser aplicado ao documento que possui a definição da instrução. A sintaxe da instrução de processamento segue o seguinte modelo:

```
<?xml-stylesheet type="text/html" href="limited.xsl"?>
```

Em uma aplicação, quando um *parser* encontra uma instrução de processamento, ela reporta a localização e o conteúdo da instrução para o cliente da aplicação. O *parser* não provê suporte para interpretação do conteúdo da instrução.

1.3. Validação de documentos XML

Uma das inovações trazidas por XML é a habilidade de se poderem especificar condições que permitem dizer quais os atributos de um determinado elemento, se um elemento pode ou deve estar contido outro, se um atributo pode estar presente mais de uma vez num mesmo elemento. Especificar uma lista de condições como estas faz parte do processo de validação. Validação de documentos XML não é obrigatória, mas é muito importante em qualquer aplicação.

Há diversas formas de se validar documentos XML. Existem as chamadas *schemas languages* que podem ser utilizadas para especificar as condições de validação. A lista das condições é especificada em documento chamado *schema*. A linguagem mais encontrada nos *parsers* de XML é a *Document Type Definition* (DTD) e é dela que iremos comentar agora.

1.3.1. Document Type Definitions *(DTD)*

O foco da DTD está na estrutura dos elementos que irão compor o XML.

1.3.1.1. Declaração de elementos

Cada elemento que fará parte de um documento XML é especificado no DTD pela declaração *ELEMENT*. A declaração abaixo especifica que o elemento *taxa* não contém nenhum nó filho.

```
<!ELEMENT  taxa (#PCDATA)>
```

Elementos que possuem nós filhos devem declarar uma lista de nomes dos seus filhos separados por vírgulas. Esta lista é chamada de *content model.* Veja o exemplo abaixo:

<!ELEMENT item (id, precoUnit, qtd)>

É possível ainda determinar particularidades dos elementos e seus filhos. Se colocarmos um ponto de interrogação (?) após um elemento da lista de atributos, este se torna opcional para o elemento. Se colocarmos um asterisco (*) após um elemento da lista significa que este elemento pode existir zero ou mais vezes naquela posição. Caso exista um sinal de adição (+) indica que deve existir pelo menos uma instância daquele elemento, podendo também existir mais de uma. Caso um elemento não venha a possuir nenhum conteúdo, é possível especificar esta declaração substituindo a *content model* pela palavra *EMPTY.*

1.3.1.2. Declaração de atributos

Um DTD também é capaz de especificar os atributos que devem existir em cada elemento. Cada atributo é declarado com a declaração *ATTLIST*. Essa declaração especifica o elemento ao qual o atributo pertence, o nome do atributo, o tipo do atributo e o valor *default* do atributo. A declaração abaixo especifica que cada elemento *item* possui um atributo *id* do tipo *ID* e que é requerido.

<!ATTLIST item id ID #REQUIRED>

Um DTD pode definir dez tipos para atributos. Estes tipos estão listados abaixo:

- *CDATA*

Este é o tipo para *strings* de texto. É o valor *default* para atributos não declarados em documentos inválidos.

- *NMTOKEN(S)*

O tipo *NMTOKEN* especifica uma *string* composta de um ou mais nomes de caracteres legais em XML. Um XML *token* pode começar com um dígito, diferente de um nome qualquer em XML. *NMTOKENS* especifica uma lista de *tokens* separados por espaços em branco.

- *ID*

É um tipo que especifica que o valor do atributo é único.

- *IDREF(S)*

Especifica um valor único para um outro elemento contido no documento XML. *IDREFS* identifica uma lista de nomes usados como valores em alguma posição do documento.

- *ENTITY (IES)*

Especifica o tipo de um nome de uma entidade declaradas no DTD. *ENTITIES* especifica uma lista de entidades declaradas no DTD.

- *NOTATION*

É o tipo que indica uma notação que foi declarada no DTD.

- *ENUMERATION*

Especifica uma lista de todos os valores legais para um atributo, separados por barras verticais (|). Cada valor possível deve ser um *token* em XML.

DTD's também possibilitam a especificação de valores default para os atributos. Quatro valores são especificados abaixo:

- *#REQUIRED*

Essa declaração define que um elemento deve conter, obrigatoriamente, um valor para o atributo em questão.

- *#IMPLIED*

A declaração *#IMPLIED* especifica que um elemento pode ou não indicar um valor para o atributo que contêm a declaração, porém, caso este não especifique valor algum, o DTD não proverá um valor *default* ao atributo.

- *#FIXED "value"*

O atributo sempre possuirá um valor que será especificado após a declaração *#FIXED* entre aspas duplas ou simples.

- *"value"*

Essa declaração define um valor *default* para o atributo em questão. Esse valor deve estar entre aspas duplas ou simples.

A estrutura apresentada abaixo exemplifica a declaração completa de um DTD. Ele faz utilização das declarações de elementos e atributos apresentadas anteriormente.

```
<!ELEMENT pedido (cliente, produto+, subtotal, taxa, total)>
<!ELEMENT cliente (#PCDATA)>
<!ATTLIST cliente id ID #REQUIRED>
<!ELEMENT item (nome, SKU, qtd, precoUnit, desconto?, descricao)>
<!ELEMENT nome (#PCDATA)><!ELEMENT qtd (#PCDATA)>
<!ELEMENT preco (#PCDATA)><!ELEMENT desconto (#PCDATA)>
<!ELEMENT enviarPara (rua+, cidade, estado, cep)>
<!ELEMENT rua (#PCDATA)>
<!ELEMENT cidade   (#PCDATA)>
<!ELEMENT estado  (#PCDATA)>
<!ELEMENT cep    (#PCDATA)>
<!ELEMENT descricao (#PCDATA)>
<!ELEMENT subtotal (#PCDATA)>
<!ELEMENT taxa (#PCDATA)>
<!ATTLIST taxa currency (USD | CAN | GBP) #REQUIRED rate CDATA "0.0">
<!ELEMENT total (#PCDATA)>
<!ATTLIST totalPedido (USD | CAN | GBP) #REQUIRED>
```

Exemplo 1.1 - Declaração de um Document Type Definition

1.3.2. Document Type Declarations

Os documentos XML estão associados a DTDs através da utilização de *document type declarations*. A declaração abaixo é especificada em documentos que relacionem o DTD apresentado anteriormente.

```
<!DOCTYPE Pedido SYSTEM “pedido.dtd”>
```

Essa declaração deve estar localizada após a declaração *<?xml ...?>* para que o documento possa ser considerado válido. Outra restrição para essa declaração é que a mesma deve aparecer antes do elemento raiz do documento.

```
<?xml version="1.0" encoding="ISO-8859-1"?>

<!DOCTYPE pedido SYSTEM "order.dtd">

<pedido>  ... </pedido>
```

Exemplo 1.2 – Document Type Declaration

DTDs, além de serem utilizados para definição de validações a documentos XML, são utilizados também para definição de entidades, de notações e, como vimos, prover valores *default* para os atributos definidos. Dificilmente programadores necessitam saber qual valor *default* foi aplicado a um atributo ou de qual entidade veio um determinado elemento, porém algumas APIs podem indicar esses dados ao programador.

2. SAX – SIMPLE API FOR XML

Neste capítulo iremos tratar de uma *Application Programming Interface*, mais conhecida pela sigla API, usada para acesso serial de arquivos XML fazendo uso da linguagem Java. Para se tratar um arquivo XML, primeiramente é necessário que este possa ser validado através de um processo de *parsing*, como já foi comentado no capítulo anterior. Após o documento ter sido processado pelo *parser*, os dados contidos no documento se tornam disponíveis para utilização em uma aplicação.

Este capítulo descreve como são as classes Java que implementam essa funcionalidade tendo como base esse modelo (SAX) para o tratamento dos dados contidos em um documento XML.

Segundo Kurt A. Gabrick e David B. Weiss (GABRICK, 2000) SAX é o protocolo de acesso que a maioria dos servlets e programas network-oriented irão usar para transmitir e receber documentos XML, porque é o mecanismo mais rápido e que faz uso de pouca memória que está disponível para lidar com documentos XML.

2.1. Operações básicas no tratamento de documentos XML

A arquitetura básica do processamento de arquivos XML é mostrada na figura 2.1. Pode-se observar que há uma iteração entre os documentos XML e a aplicação que necessita

deles para funcionar. Entre eles existem fatores que serão explicados mais a seguir, ou no próximo capítulo.

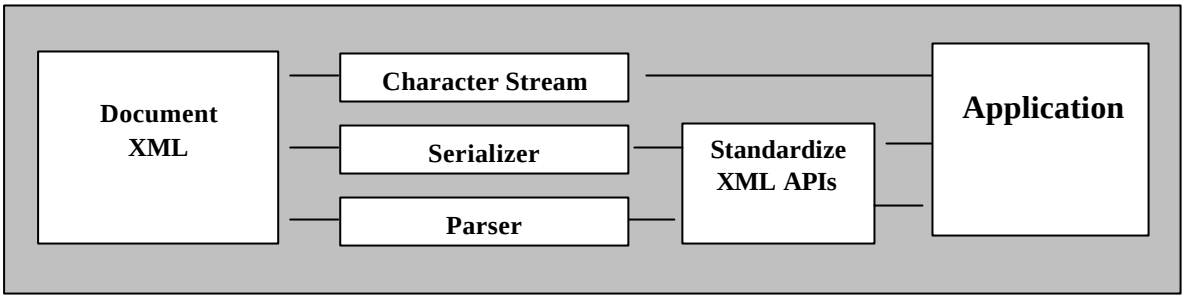

Figura 2.1. (GABRICK,2000) Arquitetura básica de processamento de XML

Para que uma aplicação esteja apta a trabalhar com documentos XML, ela primeiramente deve estar apta a validar esses documentos. Uma atividade de validação e checagem do padrão do documento XML para uma determinada aplicação é conhecida por um processo denominado *parsing*. Segundo Simon Brown (BROWN, 2001) *parsing* se baseia em recuperar um documento XML e dividi-lo em pedaços (nós) tomando como base para esse processo as *tag*s, os atributos, os comentários, as instruções de processamento, enfim, todos elementos que fazem parte do documento XML.

O enfoque deste capítulo se baseia em um dos modelos de *parsing* mais comumente usados. O modelo que será apresentado é especificado como um modelo do tipo *push parsing*. No modelo *push parsing*, o *parser* envia uma notificação para a aplicação informando-a sobre os elementos que deverão, ou que podem, serem encontrados durante o processo de *parsing*. As notificações são enviadas na ordem em que aparecem no documento, ou seja, a medida em que vão sendo lidas, são enviadas. Essas notificações já são esperadas pela aplicação através de implementações em eventos no código da aplicação. Baseado neste fato, esse tipo de processamento de *parsing* é conhecido também por *event-based parsing*. A comunidade XML criou um padrão para o modelo *push parsing* e que é bastante utilizado. O modelo é a Simple API for XML, mais comumente conhecido por SAX.

2.2. O que é SAX?

SAX é um mecanismo de acesso serial utilizado para acessar documentos XML. Este protocolo de acesso é um dos mais usados para a transmissão e recepção de documentos XML, porque é bastante rápido e faz uso de pouca memória no tratamento desses documentos. Em aplicações onde é necessário realizar alterações em documentos XML não se faz viável a utilização desse protocolo e sim outro protocolo, DOM, que será apresentado no capítulo seguinte.

Java apresenta interfaces para o tratamento de documentos XML baseados em SAX. Essas interfaces são bastante simples e em pouco número. A partir de agora será exemplificada a utilização dessas interfaces passo a passo para a criação de uma simples aplicação que faz o tratamento de um arquivo XML.

2.3. Criando classes para processamento de documentos XML usando SAX

Para poder começar a detalhar o que vem a ser esse processamento de um documento XML usando a Simple API for XML, utilizaremos um documento XML como exemplo. O documento que iremos utilizar é apresentado abaixo:

```
<?xml version="1.0" encoding="UTF-8" ?>
<invoice id="43871" submitted="2002-11-02">
<pgtoPara id="addr-1">
<compania>Lojas Americanas</compania>
```

```
<rua>Avenida Washington Soares</rua>
 <bairro>Edson Queiroz</bairro>
<complemento>Iguatemi</complemento>
<cidade>Fortaleza</cidade>
 <estado>CE</estado>
<codigopostal>60515112</ codigopostal >
</ pgtoPara >
<pedido ref=”addr-1”>
<item sku="318-BP" qtd="4" precoUnit="29.95">
<descricao>Camisas gola polo – Pólo </descricao>
</item><item sku="318-BP" qtd ="2" precoUnit ="129.00">
< descricao >Racker para computador</descricao>
</item>
<item sku="008-PR" qtd ="4" precoUnit ="0.00">
<descricao>Promotional: Capas para computador </descricao>
 </item>
</pedido>
<taxa>18.89</ taxa >
<custoTotal>396.69</ custoTotal >
 </invoice >
```

Exemplo 1 – Documento XML a ser utilizado pela aplicação detalhada neste capítulo

Com o documento em mãos estamos prontos para iniciar a codificação da classe que utilizaremos para validar o mesmo. Como podemos observar, o documento acima possui elementos que descrevem uma conta de uma compra realizada nas Lojas Americanas situada no shopping Iguatemi, assim como os itens que foram comprados. A classe que iremos criar

servirá para validarmos o preço da compra, ou seja, para verificarmos se a soma dos preços dos itens comprados será igual ao preço informado na *tag custoTotal* ao final do documento.

2.3.1. Importando classes

As importações das classes definidas abaixo serão necessárias à nossa aplicação. São elas:

```
import java.io.*;
import org.xml.sax.Attributes;
import org.xml.sax.SAXException;
import javax.xml.parsers.SAXParser;
import javax.xml.parsers.SAXParserFactory;
import org.xml.sax.helpers.DefaultHandler;
```

As classes no pacote *java.io* são necessárias para a realização do *input/output*, obviamente. Mais especificamente faremos uso da *java.io.InputStream* nessa aplicação. O pacote mais importante para a nossa aplicação é o *org.xml.sax*. Esse pacote define todas as interfaces que são utilizadas para o *parser* SAX.

A classe *SAXParserFactory* cria uma instancia que usaremos na aplicação. Essa instancia será atribuída ao objeto da classe *SAXParser*. Essa classe define vários tipos de métodos *parse()*. Geralmente é passado um documento XML e um objeto do tipo *DefaultHandler* para o *parser* que processa o documento XML e invoca os métodos apropriados ao objeto de *DefaultHandler*.

A classe *DefaultHandler* implementa as interfaces *Content Handler*, *Error* Handler, *DTDHandl*er e *Entity Resolver* que estão descritas abaixo. Estas interfaces possuem métodos que serão chamados quando disparados os eventos pelo *parser* (*event-based parser*). A aplicação deve implementar os eventos que forem interessantes para tratar o documento em questão.

- ContentHandler:

Contêm os métodos básicos a todos os *parser*s. A aplicação que iremos mostrar faz uso desses métodos.

- ✓ *Void startDocument():* Recebe a notificação do início do documento.
- ✓ *Void endDocument():* Recebe a notificação do fim do documento.
- ✓ *Void startElement(String namespaceURI, String localName, String qName, Attributes att)*: Recebe a notificação do início de um elemento.
- ✓ *Void characters(char[] ch, int start, int length)*: Recebe a notificação dos caraceteres.

- *ErrorHandler:*

Contêm os métodos para a recepção de notificações de erros. A interface *DefaultHandler* implementa erros fatais que possam ocorrer. Não há implementação de erros mais simples, e devam ser tratados pelo(s) autor(es) da aplicação.

- ✓ *Void error(SAXParseException exception):* Recebe a notificação de um erro recuperável.Um exemplo de tipos de exceções, são os erros que devem ser tratados pelo programador da aplicação.

- ✓ *Void fatalError(SAXParserException exception):* Recebe a notificação de um erro não recuperável. Erros não recuperáveis são, por exemplo, erros de má formação do documento XML.

- *DTDHandler:*

Contêm os métodos para lidar com as entidades definidas em um DTD, caso tenha sido definido, de uma aplicação.

- *EntityResolver*:

Contêm os métodos que servem para ajudar na localização de entidades externas.

Como já foi dito, SAX define um modelo de *parsing* baseado em eventos. Um SAX *parser* irá invocar os métodos dessas interfaces a medida que for trabalhando com o documento XML. Devido à simplicidade desse modelo, o *parser* não necessita manter o estado da informação na memória. Isto explica porque *parser*s baseados em SAX são muito rápidos e altamente eficientes.(GRAHAM, 2002)

2.3.2. Instanciando um parser

Agora iremos iniciar a construção da classe que será utilizada na validação do total da compra feita por um cliente que foi informado no documento XML. Abaixo aparece o cabeçalho da nossa classe. Foi definida uma interface denominada *InvoiceChecker,* pois a aplicação poderá escolher outro tipo de *parser*, que será mostrado no capítulo seguinte.

```
// InvoiceChecker.javaimport
 java.io.InputStream;
public interface InvoiceChecker{
void checkInvoice(InputStream invoiceXML) throws Exception;
}
```

A classe herda de *DefaultHandler* e realiza implementações dos métodos que foram apresentados anteriormente. As implementações dos métodos serão apresentadas na próxima seção. Algumas variáveis são definidas para controle dos valores existentes no documento.

- ✓ *runningTotal:* guardará o valor total de todos os itens existentes no documento.
- ✓ *total:* guardará o valor armazenado pela *tag* < custoTotal>.
- ✓ *amount:* armazenará os valores que forem lidos e que serão atribuídos às variáveis acima definidas.
- ✓ *isMoneyContent:* indicará se uma *tag* possui valores de preços que deverão ser verificados ou não.

Mais abaixo na aplicação instanciamos o objeto que realizará o processo de *parsing* do documento em questão. Fazemos uso das classes que foram apresentadas anteriormente, *SAXParserFactory* e *SAXParser*. Depois de instanciados os objetos, fazemos o chamado ao método *saxParser.parse(invoiceXML, this*), que realizará o *parse* do documento XML passando como parâmetros o documento em questão, que foi recebido como parâmetro pelo método *checkInvoice(InputStream invoiceXML)*, e a referencia do objeto.

```
//InvoiceCheckerSAX.java
public class InvoiceCheckerSAX extends DefaultHandler implements InvoiceChecker{
//Variáveis de controle
```

```
double runningTotal = 0.0;
double total = 0.0;
boolean isMoneyContent = false;
double amount = 0.0;
//Instanciação do parser
public void checkInvoice(InputStream invoiceXML) throws Exception{
    SAXParserFactory factory = SAXParserFactory.newInstance();
   SAXParser saxParser = factory.newSAXParser();
   saxParser.parse(invoiceXML, this);
}

...
```

2.3.3. Implementando o content handler

Nesta seção faremos as implementações dos métodos que irão checar as *tag*s existentes no documento e fazer a validação dos valores informados.

No método *startDocument* serão iniciadas as variáveis que farão o controle dos valores.

```
...
public void startDocument() throws SAXException{
runningTotal = 0.0;
total = 0.0;
isMoneyContent = false
}
```

```
...
```

No método *startElement*, fazemos a verificação se a *tag* no momento do processo de *parsing* é a *tag item*. Se a *tag* for a que a aplicação espera que seja, é feita então a atribuição do valor da multiplicação entre o preço unitário daquele item e a quantidade de itens que foram comprados para a variável *runningTotal*. Caso a *tag* em questão seja *custoTotal* ou *taxa*, o flag *isMoneyContet* é setado para *true* indicando que os valores de tais *tag*s devem ser considerados no método *characters*, pois essas *tag*s não possuem atributos definidos a elas.

```
...
public void startElement(String namespaceURI, String localName,
      String qualifiedName, Attributes attrs) throws SAXException{
     if (localName.equals("item")){
runningTotal+=Integer.valueOf( attrs.getValue(namespaceURI,"qtd")).intValue() *
Double.valueOf(attrs.getValue(namespaceURI,"precoUnit")).doubleValue();
} else if(localName.equals("taxa") || localName.equals("custoTotal")){
isMoneyContent = true;
}
}
...
```

O próximo método, *characters(char buf[], int offset, int len)*, faz a leitura dos conteúdos das *tag*s que possuem setados com o valor *true* a variável *isMoneyContent*, pois devem ser considerados também para a validação dos valores informados no documento.

```
…
public void characters(char buf[], int offset, int len) throws SAXException{
```

```
if (isMoneyContent){
String value = new String(buf,offset,len);
amount = Double.valueOf(value).doubleValue();
}
}
...
```

O próximo método, *endElement (String namespaceURI, String localName, String qualifiedName)*, da nossa classe é chamado para recuperar o valor da *tag custoTotal* que foi atribuído a variável *amount* no método anterior. O valor recuperado é atribuído a variável *total* para que seja validado ao fim do documento. Nesse método as variáveis de armazenamento, *runningTotal* e *total* são comparadas a um valor determinado para que não haja erros aritméticos binários na aplicação. No caso, o valor aplicado equivale à metade de um centavo.

No último método, *endDocument()*, caso a validação dos valores não esteja de acordo com o esperado, é lançada uma exceção avisando que o valor calculado é diferente do valor esperado.

```
...
public void endElement (String namespaceURI, String localName,
String qualifiedName) throws SAXException{
if (isMoneyContent){
if (localName.equals("custoTotal")){
total = amount;
}else{
runningTotal += amount;
```

```
    }
    isMoneyContent = false;
      }
       public void endDocument() throws SAXException{
          if (Math.abs(runningTotal - total) >= 0.005){
                throw new SAXException("Erro ao checar documento: total é "+
                                          Double.toString(total) +
                                          " enquanto nossos cálculos mostram o total de "+
                                         Double.toString(Math.round(runningTotal * 100) /
                                           100.0)
                              );
          }
        }
    }
```

A implementação apresentada é bastante simples, porém muitas coisas ainda podem ser adicionadas a ela. Pode–se ainda fazer tratamento de erros de entrada, ou seja, fazer uma validação mais apurada, se for o caso.

Exemplificamos as funções básicas para tratar documentos XML utilizando SAX. Características a mais desse modelo poderão ser vistas no próximo capítulo em comparação com o *Document Object Model*, próximo modelo a ser apresentado.

3. DOM – DOCUMENT OBJECT MODEL

Neste capítulo serão discutidas as características do *Document Object Model* e apresentadas as suas distinções em relação a SAX. DOM é uma API bem diferente de SAX e a complementa em vários aspectos. Muitas aplicações fazem uso de ambas APIs, portanto é necessário ao desenvolvedor que possua conhecimentos sobre ambas.

3.1. Document Object Model

DOM, diferentemente de SAX, possui suas origens na *World Wide Web Consortium* (W3C). DOM não foi criado especificamente para utilização com Java, mas sim para representar o conteúdo e o modelo dos documentos em várias linguagens de programação. DOM pode ser utilizado, por exemplo, em JavaScript, CORBA, e em outras linguagens fazendo com que este modelo interaja entre diferentes linguagens e plataformas.(GABRICK, 2000)

DOM é organizado pela W3C em várias fases. A primeira fase detalha a funcionalidade e a navegação do conteúdo de um documento. Em DOM outros modelos de documentos podem ser analisados, como HTML. A segunda fase, finalizada em 08 de Novembro de 2002, adiciona funcionalidades à fase 1 provendo módulos e opções para tratamento direcionado a modelos de conteúdo específicos, como XML, HTML e *Cascading Style Sheets* (CSS). A

terceira fase traz mais funcionalidades e facilidade para modelos de documentos mais específicos, assim como validações de documentos XML mais apuradas.

3.2. Qual linguagem utilizar?

Usar DOM para uma linguagem de programação específica requer um conjunto de interfaces e classes que definem e implementem esse modelo. É necessário, portanto, que haja uma implementação, *language binding*, que faça a ligação entre a linguagem que está sendo utilizada e o modelo existente na especificação do *Document Object Model*. No caso dessa monografia, será utilizada a *Java language binding* que pode ser encontrada no seguinte link para *download*, http://www.w3.org/TR/DOM-Level-2/java-binding.html.

Depois de checar se os pacotes utilizados por DOM estão indicados no *classpath*, é necessário adicionar as classes do pacote *org.w3c.dom* assim como seus sub-pacotes.

3.3. Entendendo melhor o DOM

Document Object Model é um tipo de *parsing* diferente do modelo *push parsing*, ao qual se aplica a *Simple API for XML*. DOM se aplica ao modelo *one-step parsing*, onde o *parser* lê todo o documento e gera uma estrutura de dados descrevendo todos os elementos do documento.(GRAHAM,2002)

A especificação de DOM para XML definida pelo W3C apresenta os tipos de objetos, suas propriedades e operações que serão incluídos na estrutura de dados gerada pelo *parser*. DOM é tão popular que o tipo de modelo *one-step parsing*, é mais conhecido como *DOM parsing*. Um modelo da estrutura gerada pelo *parser* é apresentado na figura 3.1.

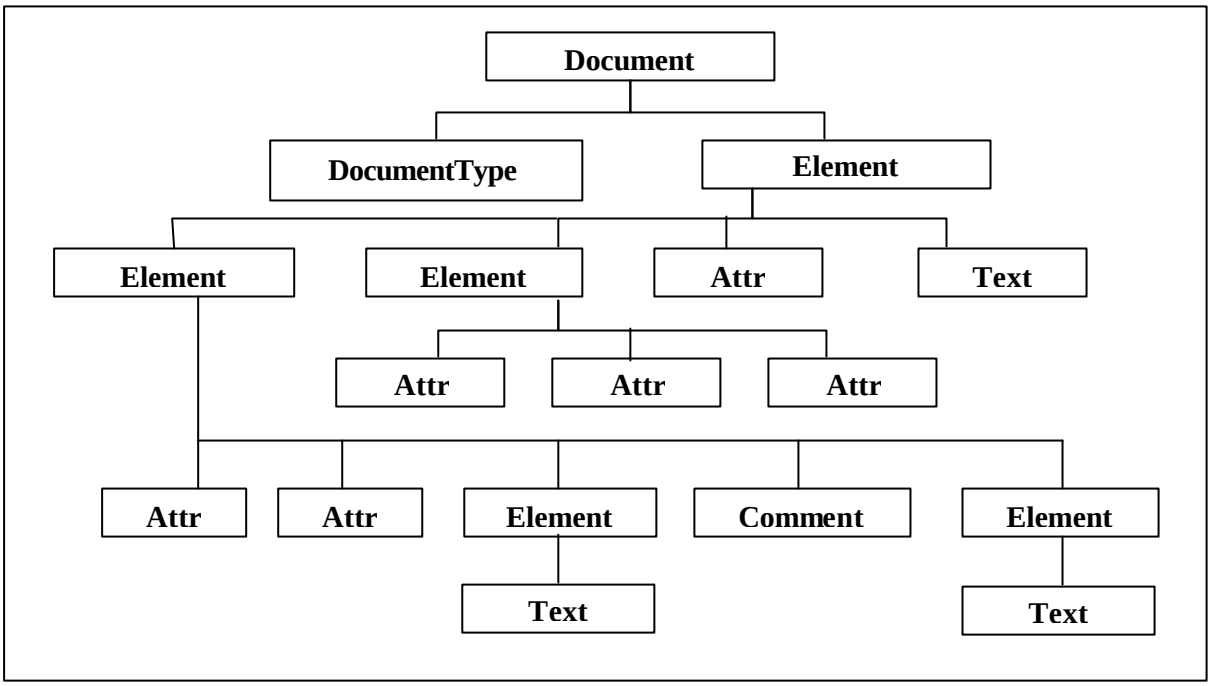

Figura 3.1. (BROWN,2001) Estrutura de um documento XML gerado por um DOM *parser*

O maior problema encontrado em DOM, é que o *parser* não detalha a estrutura gerada no processo de *parsing*, dessa forma é mais complicado tratar a estrutura gerada a partir da linguagem nativa. Por causa disso a comunidade Java criou uma especificação que simplificasse esse tratamento.

DOM faz o tratamento em Java das estruturas geradas a partir da interface *org.w3c.dom.Node*. A partir dessa interface, DOM provê outras interfaces mais especificas como *Element*, *Document*, *Attr*, e *Text*. Devido a este modelo, todas as estruturas geradas podem ser tratadas atrtavés de dois tipos: fazendo uso do tipo genérico *Node*, ou fazendo uso de um tipo específico (*Element*, *Attr*, etc.). Muitos métodos utilizados para a navegação na estrutura gerada, como *getParent( )* e *getChildren( )*, pertencem a interface *Node*, tornando possível o uso de qualquer um dos tipos apresentados para percorrer toda a estrutura, sem precisar se preocupar qual o tipo da estrutura em questão.

Uma diferença entre DOM e SAX, é que DOM define suas próprias estruturas. É necessário que se faça uso das classes *NodeList* and *NamedNodeMap* quando se trabalha com DOM, ao invés de usar as *collections* em Java. A figura 3.2. mostra uma estrutura UML com as interfaces e classes utilizadas em DOM.

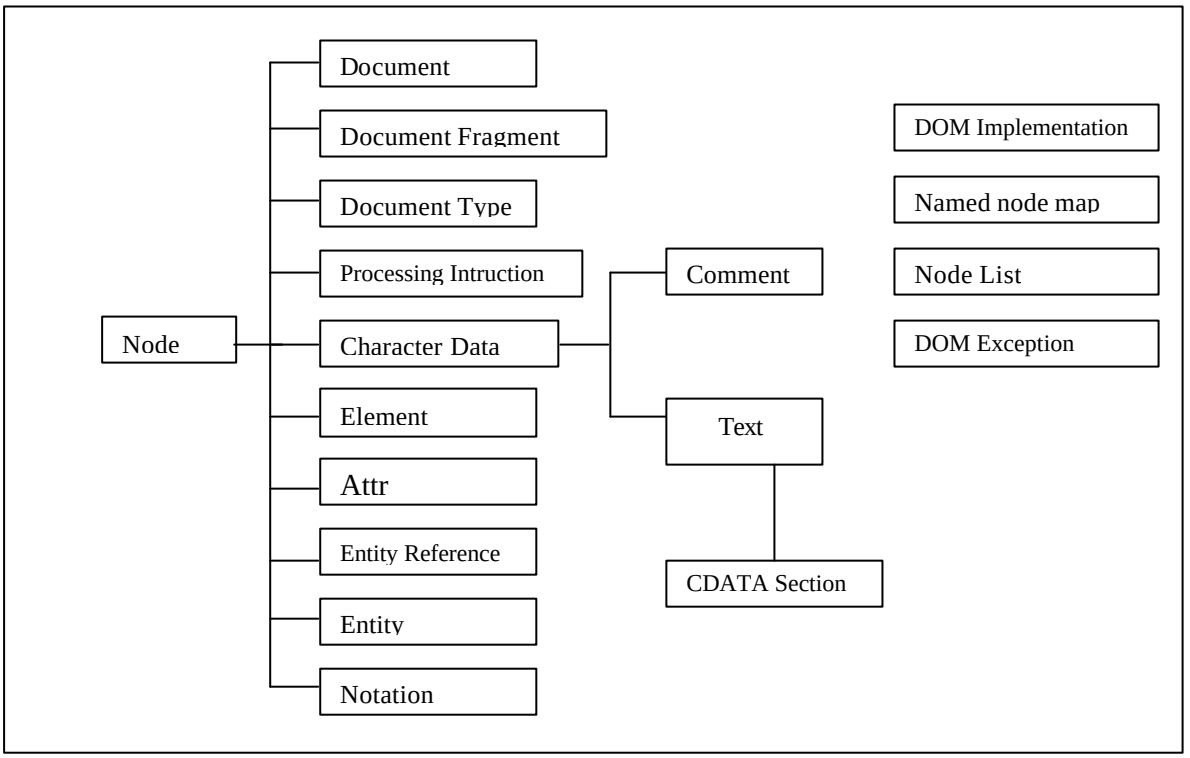

Figura 3.2. (BROWN,2001) Modelo UML das classes e interfaces utilizadas no Document Object Model

A interface principal é *Node*. É nela que estão os métodos que recuperam valores como:

- ✓ *getNodeName();*
- ✓ *getNodeType(): (attributes, document, element, character data, text node, comment, processing instructions)*
- ✓ *getNodeAttributes();*

Para percorrer os nós da estrutura, existem os métodos *getParentNode()* e *getChildNodes()*. Porém as operações mais importantes envolvem criar nós, associar os nós na estrutura e localizar nós pelo nome (*getElementsByTagName()*). Este último método é bastante importante, pois não se faz necessário percorrer toda a estrutura em busca do nó específico.

3.4. Implementando um exemplo com DOM

Vamos a partir e agora iniciar a implementação tendo como modelo o DOM e fazendo uso da API para processamento de XML implementadas em Java para esse modelo.

3.4.1. Classes necessárias à implementação

DOM faz uso dos seguintes pacotes:

Pacote	Descrição
org.w3c.dom	Define as interfaces necessárias para programar aplicações para XML, como especificado pela W3C.
javax.xml.parsers	Define as classe *DocumentBuilder* e *DocumentBuilderFactory*.

Tabela 3.1. – Pacotes utilizados por Java na implementação do modelo DOM

O exemplo que será detalhado necessitará das seguintes classes listadas abaixo.

```
import java.io.InputStream;
import org.w3c.dom.Node;
import org.w3c.dom.NodeList;
import org.w3c.dom.Document;
import org.w3c.dom.Element;
import org.w3c.dom.CharacterData;
import javax.xml.parsers.DocumentBuilder;
import javax.xml.parsers.DocumentBuilderFactory;
…
```

3.4.2. Classe principal: InvoiceCheckerDOM.

A classe implementada não necessitará de variáveis de controle para manter o contexto do processo de *parsing*, pois este já está implícito na hierarquia da estrutura criada em DOM no resultado do processo. As variáveis de armazenamento são as mesmas do exemplo do capítulo anterior (*runingTotal, total*). A classe do nosso exemplo implementa a interface *InvoiceChecker* de modo a prover implementação ao método *checkInvoice(InputStream invoiceXML)* para a validação do total da compra informada no documento XML, que será o mesmo utilizado na exemplificação do capítulo anterior.

```
…
public class InvoiceCheckerDOM implements InvoiceChecker{
public void checkInvoice(InputStream invoiceXML) throws Exception{
Double runningTotal = 0.0;
```

```
…
```

A *factory* utilizada para realizar o processo de *parsing* no documento XML é a mesma utilizada no processo de *parsing* em SAX, com a diferença que ela faz o uso das classes *DocumentBuilder* e *DocumentBuilderFactory*. O método *parse()* utilizado no modelo DOM, diferentemente do utilizado em SAX, retorna um objeto da interface *Document* que armazena toda árvore resultado do processo de *parsing*.

```
…
DocumentBuilderFactory factory = DocumentBuilderFactory.newInstance();
DocumentBuilder builder = factory.newDocumentBuilder();
Document doc = builder.parse(invoiceXML);
…
```

Com a estrutura armazenada no objeto *doc*, a aplicação pode iniciar o tratamento dos elementos da mesma, caso o processo de *parsing* tenha ocorrido corretamente, sem nenhum problema no documento XML. É criado um objeto do tipo *NodeList* que receberá uma lista contendo todos os nós de *tag* com nome *item* através da chamada do método *getElementsByTagName()* pelo objeto *doc*. É realizado um laço para percorrer todo o objeto *NodeList* e receber todos os valores dos atributos das *tags* contidas no objeto e que são importantes para a aplicação.

```
…
NodeList itemList = doc.getElementsByTagName("item");
for (int i=0; i<itemList.getLength(); i++){
Element item = (Element)itemList.item(i);
Integer qty = Integer.valueOf(item.getAttribute("qtd"));
Double price = Double.valueOf(item.getAttribute(" precoUnit"));
```

```
runningTotal += qty.intValue() * price.doubleValue();
}
…
```

Na parte do código apresentado abaixo é utilizado o método *getElementsByTagName()* para recuperar os valores da *tag taxa* e *custoTotal* para que seja realizada a validação que se espera obter com esta aplicação. O método *doubleValue(Node node)* realiza a mesma função que foi implementada pelo método *characters(char buf[], int offset, int len)* da interface *DefaultHandler* na exemplificação do capítulo anterior sobre SAX.

```
…
Node nodeTax = doc.getElementsByTagName("taxa").item(0);
runningTotal += doubleValue(nodeTax);
Node nodeTotalCost = doc.getElementsByTagName("custoTotal").item(0);
double total = doubleValue(nodeTotalCost);
 if (Math.abs(runningTotal - total) >= 0.005){
            throw new Exception("Erro ao checar documento: total é "+
                                    Double.toString(total)+
                                    " enquanto nossos cálculos mostram o total de "+
                                    Double.toString(Math.round(runningTotal*100)/100.0)
                 );
              }
}
public double doubleValue(Node node) throws Exception{
        String value = ((CharacterData)node.getFirstChild()).getData();
```

```
            return Double.valueOf(value).doubleValue();
        }
}
```

Exemplificamos uma aplicação simples e que pode ser extendida e pode incorporar tratamento de exceções próprias da aplicação de modo a trazer ao usuário erros mais significativos.

3.5. Document Object Model versus Simple API for XML

Depois do que foi apresentado, é possível iniciar uma aplicação que faça o uso de um dos dois modelos discutidos. Porém deve ficar bem claro que não existe apenas um único modo ou o melhor modo de acessar documentos XML com Java. É importante saber quando fazer o uso de um dos modelos, e, aqui nesta seção, são apresentadas algumas características que ajudarão a fazer essa escolha.

Abaixo são listadas algumas razões pelas quais deve-se fazer o uso de DOM, assim como algumas limitações do modelo:

DOM é muito flexível e genérico de modo que a partir do modelo especificado pela W3C vários tipos de documentos podem ser tratados, incluindo qualquer um que contenha a sintaxe especificada a documentos XML.(BROWN, 2001)

Segundo Simon Brown (BROWN, 2001), DOM não pode ser considerado um padrão para qualquer tipo de projeto. Pelo fato desse modelo criar uma estrutura de dados que

armazena todo o documento faz com que sejam necessários alguns cuidados com o espaço de memória, além de que o tempo de processamento é bem maior do que de um objeto padrão. Ou seja, o modelo DOM não é uma boa alternativa quando se faz o uso de documentos XML muito grandes, a não ser que haja ótimo suporte a XML.

Apresentaremos agora alguns fatores que podem ser considerados para que se faça, ou não, o uso de SAX. Eles são:

- SAX é um modelo de *parsing* baseado em eventos que faz a representação de um documento XML provendo um método para transformá-lo em um *stream data*, que pode ter seu processamento realizado pelo programador da aplicação.
- SAX não pode modificar diretamente o *stream data* criado por ele através de um documento XML. SAX pode ser considerado um processo somente de leitura, pois uma vez que o programador tenha recebido uma parte documento que já sofreu o processo de *parsing*, então fica a cargo do mesmo decidir o que será feito com os dados recebidos.
- SAX é o modelo mais difícil de ser utilizado quando é necessário um processamento não seqüencial do documento XML. Fugir do que se propõe o modelo de SAX só irá trazer dor de cabeça ao programador e trará perdas à eficiência do modelo. Portanto é bem mais simples fazer uso de um modelo como o DOM que já traz essas implementações, apesar de se ter cuidado com suas requisições.

3.6. Comparação final

"Será necessário fazer uso de todos esses modelos para tratar documentos XML?". A resposta a esse questionamento é sim! Pelo fato de XML ser bastante simples, ele é usado em diferentes locais e em diferentes projetos, fazendo com que seja importante conhecer várias formas de se tratar documentos XML. Isso garante que nem sempre uma única API para XML poderá ser usada para qualquer projeto.

Cada uma das APIs desenvolvidas em Java para XML oferece recursos distintos relativos a suas especificações. *Simple API for XML*, com o seu processamento seqüencial torna o tratamento de documentos XML bastante rápidos, além de requerer pouca memória e de se poder tratar partes do documento que vão sendo processados, sem precisar esperar pelo processamento total do documento. Segundo Simon Brown (BROWN, 2001), para documentos extensos, SAX é a única solução viável. Logo DOM, que é baseado nas especificações da W3C, está muito mais próximo do que SAX de se tornar um padrão. SAX, apesar de ser utilizado bastante pela comunidade Java e de ter sido criado para resolver alguns problemas apresentados na recomendação de DOM, não é um padrão oficial. É importante avaliar os fatos e saber quando é melhor usar um dos dois modelos.

4. JAVA APIs PARA XML

O enfoque deste capítulo se destina a apresentar outras APIs implementadas através da linguagem Java e que são bastante usadas para a construção de aplicações para a *web* que utilizam como formato de dados padrão a XML. Exemplificaremos a utilização dessas APIs de modo a dar uma visão mais clara sobre as mesmas enfocando a simplicidade de seus usos.

4.1. JAVA *versus* XML

A tecnologia de *webservices está se tornando um padrão para desenvolvimento de aplicações* O assunto *web services* está sendo bastante discutido atualmente. Como o próprio termo apresenta, *web services* são serviços oferecidos e acessados pela *web*. No contexto geral, *web services* podem ser descritos como aplicações que enviam *requests*, requisitando um serviço localizado através de uma determinada URL fazendo uso de um protocolo baseado em XML como o *Simple Object Access Protocol* (*SOAP*) via HTTP. O serviço requisitado recebe o *request*, processa-o e manda como retorno um *response*. Um exemplo de *web service* pode ser uma consulta sobre o preço das ações no mercado, que representa o *request* da aplicação, e como retorno, *response*, tenho o preço da ação que foi requisitada.

XML (*Extensible Markup Language*), como já foi mostrado no capítulo 1, é uma linguagem portável e é utilizada na aplicação de *web services* por atender muito bem às necessidades dos mesmos. Organizações em geral têm notado que o uso de XML traz bastantes benefícios quando se trata de transporte de dados entre departamentos internos da

empresa e troca de informações entre outras empresas. Devido a essa habilidade de integração de dados, XML se tornou uma das bases para *web services*.

Web services também são desenvolvidos para realizar o acesso a serviços utilizando diferentes plataformas, de forma que possa haver comunicação entre entidades que participam desses serviços. A plataforma Java se adequa muito bem a esses requerimentos, pois é capaz de manter o código da aplicação portável e é uma ótima escolha para o desenvolvimento de *web* services. Essa escolha ainda se torna muito mais interessante devido Java apresentar várias APIs para XML tornando muito mais fácil o desenvolvimento da aplicação na linguagem Java. Essas APIs serão apresentadas em tópicos posteriores.

4.2. Java Application Programming Interfaces for XML

Uma das importantes características das APIs desenvolvidas com o uso da linguagem Java para tratamento de documentos XML é que elas suportam todos os padrões industriais utilizados, e com isso ajudam a garantir interoperabilidade entre diversas plataformas. Organizações, como a W3C e a *Organization for the Advancement of Structured Information Standards* (OASIS) definiram padrões capazes de serem utilizados de modo a fazer com que dados e aplicações possam interagir através da *web*. Outra característica que se aplica a essas APIs, é o fato de elas permitirem um alto grau de flexibilidade.

4.2.1. Java API for XML Processing (JAXP)

JAXP foi desenvolvida com intuito de agrupar APIs que fazem tratamentos a documentos XML. Nas discussões sobre SAX e DOM apresentadas nos capítulos 2 e 3 foram apresentadas utilizações de pacotes que constituem a JAXP. JAXP deixa a cargo do desenvolvedor a escolha de como tratar seus documentos XML: baseado em eventos,

utilizando o modelo SAX, ou representando a estrutura do documento em um objeto, utilizando DOM. Discutimos que o modelo de *parsing* que for ser utilizado na aplicação pode ser escolhido com bastante flexibilidade.

Além de DOM e SAX, JAXP também traz suporte ao padrão XSLT (*XML Stylesheet Language Transformations*), dando o controle sobre a representação dos dados e podendo realizar conversões desses dados em documentos XML ou em outros formatos como o formato HTML. JAXP ainda prover suporte a *namespaces* (conjunto de nomes únicos definidos para um modelo de documento XML) permitindo realizar aplicações sem que haja conflitos de nomes.(Sun-JAXP, 2002)

Desenvolvida a fim de prover flexibilidade, JAXP permite o uso de qualquer *parser* e isso é possível devido a existência de uma camada de *plug in* (*pluggability layer*), permitindo realizar um *plug in* com qualquer implementação que faz uso das APIs para SAX ou DOM.(GABRICK, 2001) Essa camada também torna possível o *plug in* em um processador XSL, que permite a transformação dos dados do XML de várias formas, inclusive em seu formato de apresentação. Essas transformações ficam a cargo das interfaces que constituem o pacote *javax.xml.transform*.

javax.xml.transform define a classe que será usada para obter um objeto *Transformer* que irá realizar as principais operações no processo de transformação dos dados. A transformação ocorre quando é invocado o método *transform()*, mas antes da invocação do método é necessária que se faça uma configuração para indicar o objeto como *input* (*source*) ou *output* (*result*). Os objetos (*source / result*) são criados fazendo uso dos pacotes *javax.xml.transform.dom*, *javax.xml.transform.sax* e *javax.xml.transform.stream* descritos abaixo:

- *javax.xml.transform.dom*

Esse pacote traz a definição das classes *DOMSource* e *DOMResult* que prepara a configuração do objeto como *input* ou *output* quando se está usando Document Object Model no processamento dos documentos XML.

- *javax.xml.transform.sax*

Esse pacote possui a mesma funcionalidade do pacote anterior, porém é aplicado quando se faz o uso de SAX. Nele estão definidos as classes *SAXSource* e *SAXResult*.

- *javax.xml.transform.stream*

Esse pacote define os *streams* utilizados na operação de I/O quando realizada uma transformação. As classes que fazem parte desse pacote são *StreamSource* e *StreamResult*.

4.2.1.1. Transformando dados XML com XSLT

A XSLT (*XML Stylesheet Language Transformations*) pode ser utilizada para vários propósitos, geralmente esse padrão é utilizado para geração de formatos HTML ou para criação de uma representação de dados alternativa ao XML. Nesta seção iremos exemplificar uma aplicação que transforma um documento XML em um formato HTML, após ter passado por dois filtros baseados em XSLTs. XSLT não é o enfoque deste trabalho, portanto não iremos detalhar a sua estrutura, será apresentada somente as funcionalidades da JAXP para transformações de documentos XML.

Para visualizarmos e entendermos melhor o processo que será implementado no exemplo a seguir é apresentado a figura 4.1. com a descrição do processo abaixo:

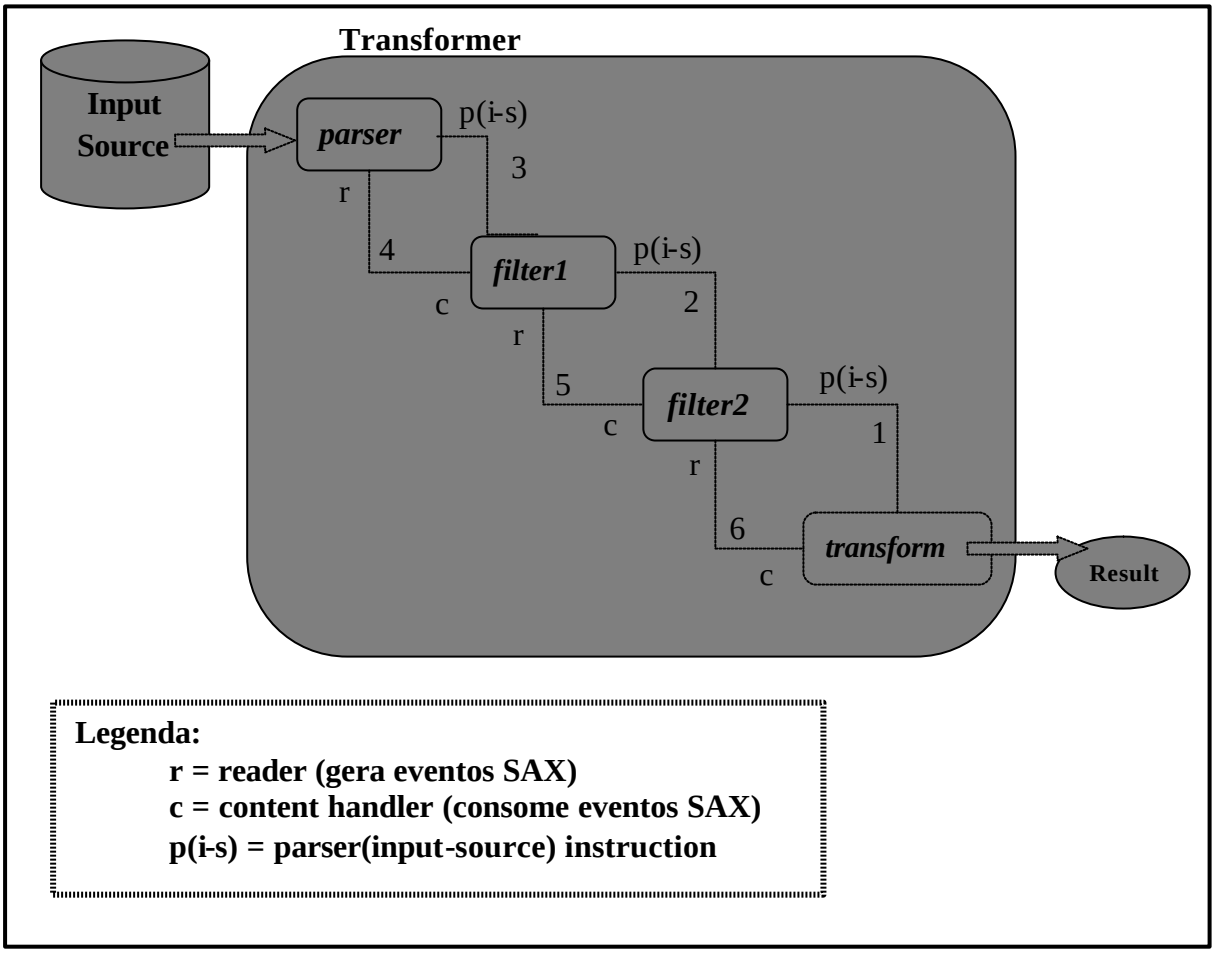

Figura 4.1. (Sun-JAXP, 2002) Estrutura do processo de transformação do exemplo anterior.

Os passos seguintes explicam o que acontece na invocação do método *transform*:

1. O *transformer* cria um objeto interno que serve como *content handler* para o objeto *filter2* e lança uma instrução de que este deve realizar o processo de parsing no objeto *input*.
2. Por sua vez, *filter2* faz o papel de *content handler* para o objeto *filter1* e também lança uma instrução para que *filter1* realize o processo de *parsing* no objeto *input*.
3. Então o objeto *filter1* envia ao objeto *parser* uma instrução de que ele deve realizar o processo de parsing no objeto *input*.
4. O objeto *parser* realiza a sua função, gera os eventos baseados em SAX e envia-os ao objeto *filter1*

5. Na posição de *content handler*, *filter1* processa os eventos e realiza as transformações, baseado em um dos arquivos XSLT. E como *reader* gera os eventos para o objeto *filter2*.
6. *filter2*, por sua vez, processa os eventos e realiza as suas devidas transformações, gerando assim o objeto *result* contendo o resultado do processamento da aplicação.

Após entendermos o processo de transformação de um documento XML, podemos agora implementar o processo apresentado. Abaixo podemos visualizar a declaração das interfaces que serão utilizadas no nosso exemplo.

```
import javax.xml.parsers.FactoryConfigurationError;
import javax.xml.parsers.ParserConfigurationException;
import javax.xml.parsers.SAXParser;
import javax.xml.parsers.SAXParserFactory;
import org.xml.sax.SAXException;
import org.xml.sax.SAXParseException;
import org.xml.sax.InputSource;
import org.xml.sax.XMLReader;
import org.xml.sax.XMLFilter;
import javax.xml.transform.Transformer;
import javax.xml.transform.TransformerException;
import javax.xml.transform.TransformerFactory;
import javax.xml.transform.TransformerConfigurationException;
import javax.xml.transform.sax.SAXTransformerFactory;
import javax.xml.transform.sax.SAXSource;
import javax.xml.transform.sax.SAXResult;
import javax.xml.transform.stream.StreamSource;
```

```
import javax.xml.transform.stream.StreamResult;
import java.io.*;
```

A classe definida abaixo, *TransformExample* recebe três parâmetros, dois arquivos XSLT e um arquivo XML. Caso não sejam passados esses parâmetros, a aplicação retorna uma mensagem de como se deve executar a aplicação e finaliza a mesma. Caso contrário, os três parâmetros são atribuídos a objetos do tipo *File*. Logo após é obtido um *parser* e um *reader* para tratar o objeto *datafile* que contém o arquivo XML.

```
public class TransformExample{
public static void main (String argv[]) {
        if (argv.length != 3) {
                System.err.println ( "RUN: java stylesheet1 stylesheet2 datafile");
                System.exit (1);
        }
        try {
          // Argumentos
           File stylesheet1 = new File(argv[0]);
           File stylesheet2 = new File(argv[1]);
           File datafile = new File(argv[2]);
         // Set input stream
        BufferedInputStream bis = new BufferedInputStream(new FileInputStream(datafile));
        InputSource input = new InputSource(bis);
        SAXParserFactory spf = SAXParserFactory.newInstance();
        spf.setNamespaceAware(true);
        SAXParser parser = spf.newSAXParser();
```

```
XMLReader reader = parser.getXMLReader();
…
```

No próximo passo do nosso exemplo, são criados os filtros, como objetos *XMLFilter* os quais receberão os arquivos XSLT passados como parâmetros e serão utilizados para transformação do arquivo XML.

```
…
// Criação dos filtros
SAXTransformerFactory stf = (SAXTransformerFactory)TransformerFactory.newInstance();
XMLFilter filter1 = stf.newXMLFilter(new StreamSource(stylesheet1));
XMLFilter filter2 = stf.newXMLFilter(new StreamSource(stylesheet2));
…
```

O objeto *XMLFilter* é tanto um *SAX reader* como um *SAX content handler*. Como *reader* o objeto *XMLFilter* gera eventos baseados no *SAX*, e como *content handler* consomem os mesmos eventos gerados. A partir da criação desses objetos as definições de quais objetos sofrerão o processo de parsing e de transformação de cada filtro gerado é realizada pela chamada ao método *setParent*, que criará uma espécie de encadeamento entre os objetos. Esse encadeamento será realizado após a chamada do método *transform* e fará com que o objeto *transformSource* sofra o processo de *parsing* e seja transformado de acordo com as especificações contidas nos filtros criados a partir dos arquivos XSLT.

```
…
StreamResult result = new StreamResult(System.out);
```

```
Transformer transformer = stf.newTransformer();

SAXSource transformSource = new SAXSource(filter2, input);

transformer.transform(transformSource, result);

}catch ( Exception ex){

        ex.printStackTrace();

}

}
```

Como podemos notar, JAXP traz métodos de transformação de documentos XML, complementando assim, juntamente com as interfaces apresentadas nos capítulos 1 e 2, o conjunto de interfaces para processamento de documentos XML.

4.2.2. Java API for XML-based RPC (JAX-RPC)

JAX-RPC é uma API utilizada no desenvolvimento de *web services* e clientes que fazem uso de chamadas a procedimentos remotos (RPC) e de XML. O mecanismo RPC habilita clientes a executarem procedimentos em outros sistemas. É bastante utilizado no modelo distribuído cliente/servidor. (Sun-JAX-RPC, 2002)

Uma chamada a procedimento remoto é representada por protocolos baseados em XML como o SOAP (*Simple Object Access Protocol*). A especificação do SOAP define estruturas, convenções e regras para a representação das chamadas a procedimentos e seus respectivos retornos (*responses*). Essas chamadas são transmitidas como mensagens representadas pelo SOAP via HTTP.

Apesar de JAX-RPC depender de protocolos complexos, a API esconde dos desenvolvedores a complexidade da aplicação. Do lado servidor, são implementados procedimentos remotos na linguagem de programação Java. Do lado cliente a programação também é bastante simples, o desenvolvedor cria um *proxy*, um objeto representando o serviço local e a invocação do procedimento remoto através do *proxy* criado.

Com JAX-RPC, clientes e *web services* ganham uma importante vantagem – a independência de plataformas trazidas pela programação Java. Fazendo uso dessa API se tem ainda outra vantagem, a de se ter um lado cliente que pode acessar um *web service* que não necessita está rodando na plataforma Java e vice-versa. Segundo Steve Graham (GRAHAM, 2002), essa flexibilidade de plataformas se dá pelo fato de JAX-RPC fazer uso de tecnologias definidas pela W3C, como: HTTP, SOAP, e a *Web Service Description Language* (WSDL). Interoperabilidade, talvez essa seja a maior característica de *web services* e que é bem suportada por JAX-RPC.

RPC envolve complicações a respeito de sua estrutura, mas JAX-RPC trata de abstrair detalhes de implementação do cliente e do desenvolvedor da aplicação.(Sun-JAX-RPC, 2002) Em um dado *web* service um cliente faz somente chamadas a métodos remotos e todo o trabalho interno, como o tratamento e transmissão dos dados, é feita automaticamente. Já no lado do servidor, são implementados os serviços oferecidos não sendo necessário se preocupar a respeito de mecanismos mais aprofundados para a implementação.

JAX-RPC, por ser bastante fácil de ser utilizada, é a principal Java API utilizada em *web services*, tanto do lado cliente como do servidor. JAX-RPC tem seu foco no mecanismo de mensagem ponto-a-ponto utilizado em SOAP, pois este é o mais utilizado por clientes de *web services*. Uma alternativa a esse mecanismo apresentado por SOAP é a *Java API for XML Messaging* (JAXM) que será discutida mais a seguir. No passo seguinte ser mostrado um exemplo simples de *web service* fazendo uso da API em questão.

4.2.2.1. Utilizando JAX-RPC

Exemplificaremos a implementação de duas classes e uma interface. Esse exemplo apresentará uma chamada a um método a partir do cliente e que resultará na apresentação de uma mensagem.

O entendimento da nossa aplicação se dará de forma mais fácil após entendermos o funcionamento do processo de chamadas remotas a procedimentos utilizando a JAX-RPC. Na figura 4.2. temos a representação do processo e a seguinte descrição do mesmo.

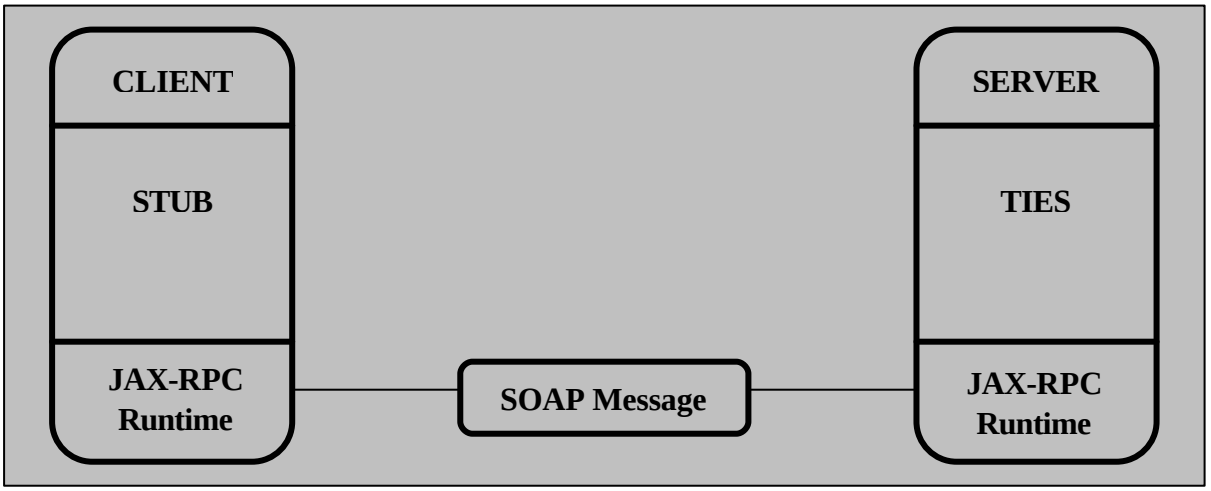

Figura 4.2. (Sun-JAX-RPC, 2002) Representação da chamada a procedimentos remotos utilizando JAX-RPC

A representação acima pode ser simplificada por:

1. Criação de um *stub* e um objeto local representando o serviço requisitado.

2. Invocação do método, que é convertida em uma mensagem SOAP e enviada ao servidor.

3. O servidor retorna o resultado da chamada como uma mensagem SOAP que é convertida pela *JAX-RPC Runtime.*

A interface abaixo possui o cabeçalho do método que será invocado pelo lado cliente da aplicação. Ela fará uso do pacote *java.rmi* fazendo uso das classes *Remote* e *RemoteException*.

```
import java.rmi.Remote;
import java.rmi.RemoteException;
public interface RPCInterf extends Remote {
  public String showMess(String s) throws RemoteException;
}
```

A próxima classe será a que representa o lado do servidor onde o método a ser invocado recebe sua implementação. O método *showMess(String s)* recebe um argumento e retorna a mensagem concatenada ao objeto *message*.

```
public class RPCImpl implements RPCInterf {
        public String message = "Java ";
         public String showMess(String s) {
                return message + s;
         }
}
```

Até o devido momento, não foram apresentadas complicações a respeito da implementação desse simples exemplo. Mas devemos notar que ainda não fizemos a implementação do lado do cliente que deverá fazer a chamada ao método e, portanto é considerada a parte mais complicada da aplicação. Como já havia exposto anteriormente, JAX-RPC trata de esconder essas complicações e detalhes da aplicação do desenvolvedor e do usuário.

```
import javax.xml.rpc.Stub;
public class RPCClient {
public static void main(String[] args) {
        try {
                Stub stub = createProxy();
                RPCInterf rpcInterf = (RPCInterf)stub;
                System.out.println(rpcInterf.showMess("RPC!"));
        } catch (Exception ex) {
                 ex.printStackTrace();
        }
}
private static Stub createProxy() {
         return (Stub)(new RPCEspecif_Impl().getRPCInterfPort());
}
}
```

Na implementação acima se faz necessária criação de um *stub* que será utilizado para a invocação do método localizado no lado servidor da aplicação. Para isso a classe *RPCClient* invoca o método privado *createProxy*. O código deste método é específico da aplicação e pode não ser portável. O código pertence à implementação de *RPCEspecif_Impl*. Após a criação do *stub*, o programa cliente realiza um *cast* do objeto *Stub* para o tipo *RPCInterf*, que é parte do servidor tornando possível a chamada do método remoto que lá se encontra.

A utilização dessa interface é bem simples como podemos visualizar no exemplo acima. Porém várias ferramentas devem ser utilizadas de modo a configurar a aplicação, ajudar na criação de *stubs* e gerar a especificação WSDL para o serviço.

4.2.3. Java API for XML Messaging

A utilização da *Java API for XML Messaging* (JAXM) torna possível a criação e envio de mensagens baseadas em XML pela *internet*. Essa API está de acordo com a especificação do protocolo SOAP e pode ser estendida de modo a possibilitar uma aplicação de mais alto nível fazendo uso de outros protocolos de envio de mensagem.(Sun-JAXM, 2002) JAXM é representada por dois pacotes:

- *javax.xml.soap:*

Este é um pacote que contém métodos para a utilização de mensagens através do SOAP. Esse pacote contém a API para criação e população de mensagens baseada em SOAP. Ele contém toda a API necessária para envio de mensagens (*request-response*).

- *javax.xml.messaging :*

Esse pacote contém a API necessária para prover transporte de mensagens e torna possível sua transmissão em sentido único, como apresentado na figura 4.3.

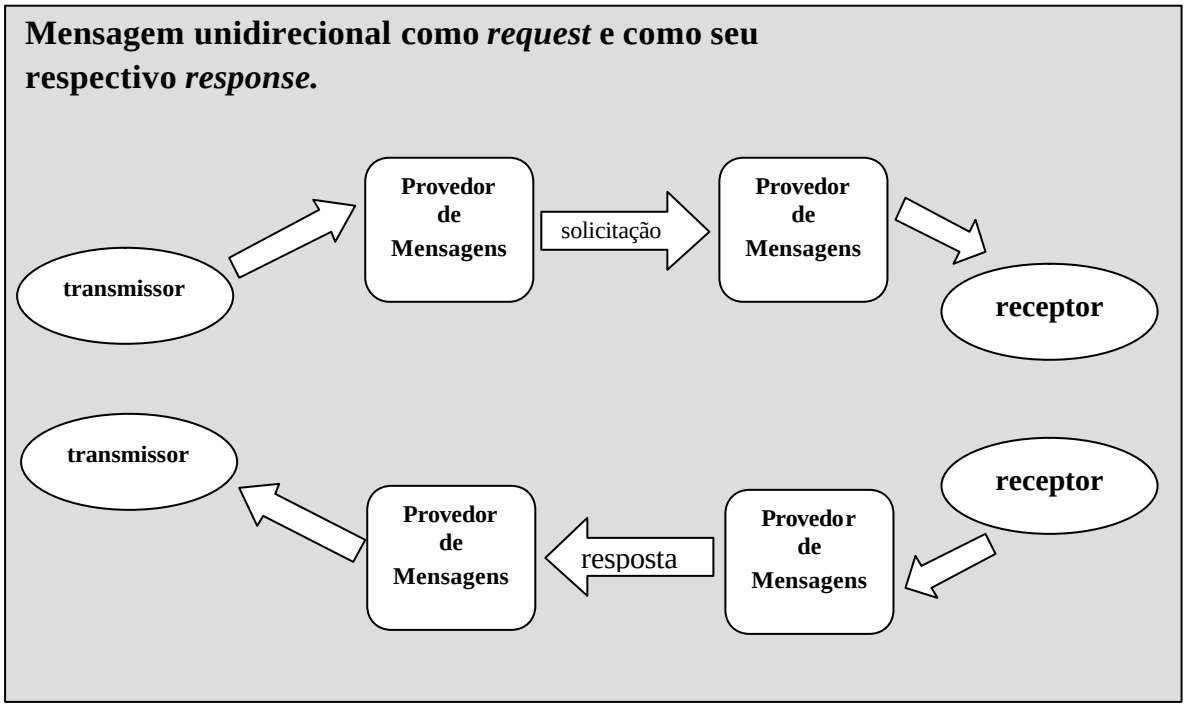

Figura 4.3. (Sun-JAXM, 2002) Sentido único de envio de mensagens provido por JAXM.

As mensagens são especificadas com base na especificação de SOAP, que define dois tipos principais de mensagens. Mensagens com anexos e sem anexos. A SAAJ (*SOAP with Attachments API for Java*) especifica classes (*SOAPMessage*, *SOAPPart*, *SOAPEnvelope*) que fazem a representação de uma mensagem completa no modelo da especificação de SOAP.(Sun-JAXM, 2002)

Na figura 4.4. está a representação de uma mensagem sem anexos. Uma mensagem é representada por vários objetos, obrigatórios e opcionais, controlados pelas APIs. Um objeto *SOAPEnvelope* pode conter mais de um objeto do tipo *SOAPHeader*, porém esse objeto é opcional. Um objeto *SOAPBody*, que sempre acompanha um *SOAPHeader*, caso exista, prover uma maneira simples de envio da mensagem a algum receptor.

SOAPPart deve conter somente o conteúdo XML da mensagem, e qualquer outro conteúdo é representado por um anexo. Na figura 4.5. é representada uma mensagem que contêm anexos. Esse tipo de mensagem é também provido pela SAAJ através da classe *AttachmentPart* e é opcional, por isso é necessário se criar um objeto à parte pra contê-los. Cada anexo deve vir com um cabeçalho MIME (*Multipurpose Internet Mail Extensions*) para indicar o tipo de conteúdo do anexo. Quando uma mensagem possui pelo menos um objeto de anexo, o objeto *SOAPPart* pode ou não possuir algum conteúdo.

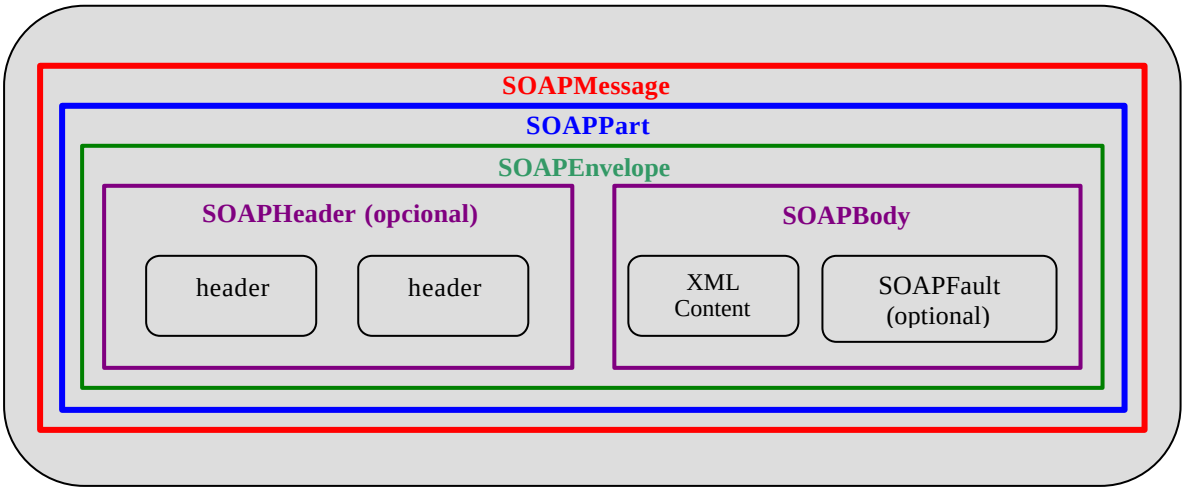

Figura 4.4. (Sun-JAXM,2002) Objeto SOAPMessage sem anexos

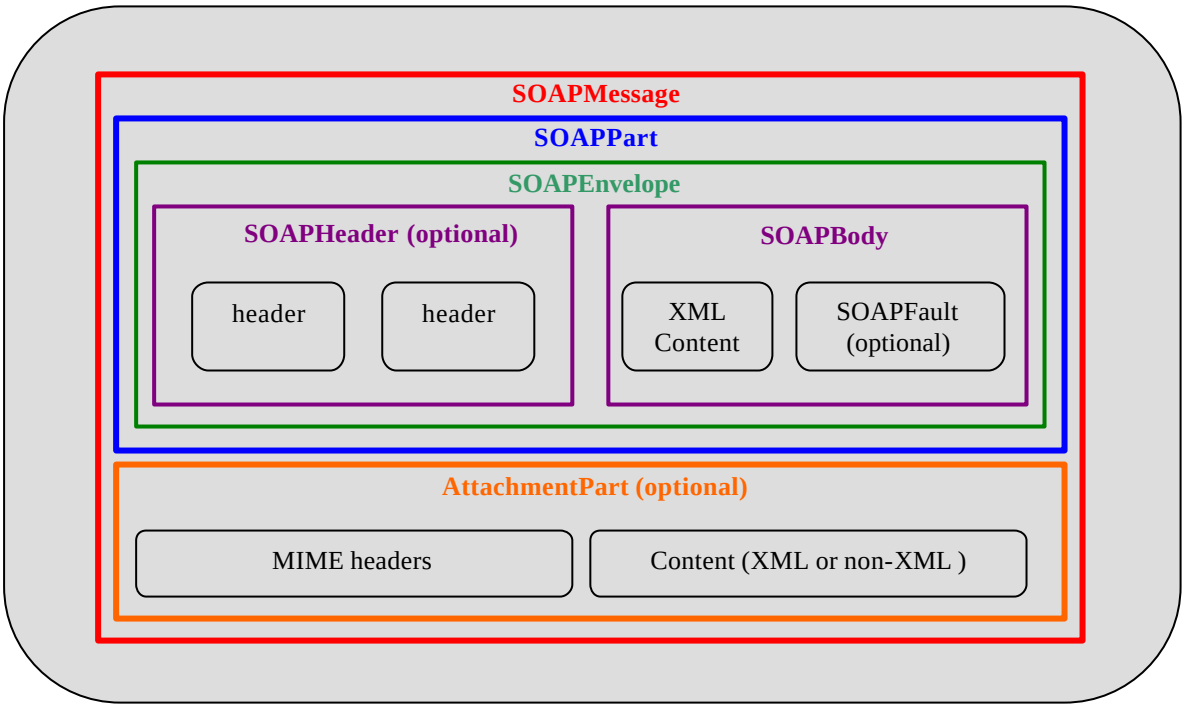

Figura 4.5. (Sun-JAXM,2002) Objeto SOAPMessage com anexos

4.2.3.1. Utilizando JAXM

Exemplificaremos agora uma aplicação mostrando a criação de conexões e mensagens fazendo uso da JAXM. Logo abaixo estão declarados os pacotes que serão utilizados no exemplo.

```
import javax.xml.soap.*;

import java.util.*;

import java.net.URL;
```

A primeira parte necessária em uma aplicação deste tipo é obter a conexão. Isso é feito criando a instancia do objeto da classe *SOAPConnectionFactory* e a partir dela faz-se a chamada ao método *createConnection()* que retorna um objeto do tipo *SOAPConnection*.

```
public class Request {  public static void main(String[] args)  {
  try {
      SOAPConnectionFactory scFactory  =   SOAPConnectionFactory.newInstance();
      SOAPConnection con = scFactory.createConnection();
…
```

O próximo passo se trata da criação da mensagem, que se dá pelo uso do objeto da classe *MessageFactory*. É possível fazer a sua própria implementação da *factory*, mas aqui utilizaremos a implementação da classe *MessageFactory*.

```
…
    MessageFactory factory =        MessageFactory.newInstance();
    SOAPMessage message = factory.createMessage();
…
```

Como foi representado nas figuras anteriores, iremos agora criar as partes da mensagem. No trecho e código abaixo está sendo criado um objeto do tipo *SOAPHeader*, porém vimos anteriormente que esse objeto não se torna obrigatório. Estão sendo acessados elementos da mensagem, cada um com seu respectivo método. O objeto *header* está no exemplo abaixo para efeito de conhecimento e como não é feito uso dele, o elemento é excluído através do método *detachNode()*. Isso é possível pelo fato de que todos os objetos *SOAPHeader*, assim como os de *SOAPElements*, são derivados da interface *Node*.

```
...
    SOAPPart soapPart = message.getSOAPPart();
```

```
        SOAPEnvelope envelope = soapPart.getEnvelope();
        SOAPHeader header = envelope.getHeader();
        Body body = envelope.getBody();
        header.detachNode();
    …
```

Agora iremos mostrar a criação do conteúdo do corpo (*body*) da mensagem. É necessário criar um elemento do tipo *Name* para identificar o elemento do corpo da mensagem. Os parâmetros passados no método de criação do nome referem-se ao nome local do elemento, o prefixo para referenciar o *namespace* usado e uma URI para o *namespace*. A especificação de um *namespace* para o elemento traz mais segurança, evitando ter conflitos entre elementos iguais. Após a criação da identificação do elemento, é feita a criação do próprio elemento, *SOAPBodyElement*, que adiciona a identificação criada para ele.

```
…
 NamebodyName=envelope.createName( "getAPrice", “m","http://www.javaexml.com.br" );
 SOAPBodyElement gltp =            body.addBodyElement(bodyName);
…
```

O próximo passo se trata do envio da mensagem. O envio se faz através da chamada do método *call* pelo objeto criado para conexão. É criado um objeto URL que será o destino da mensagem gerada. Após a chamada de envio da mensagem se torna viável liberar a conexão, pelo fato de ela alocar muitos recursos.

```
    …
```

```
  URL endpoint = new URL ("http:// www.lojasamericanas.com.br ");
  SOAPMessage response = con.call(message, endpoint);
  con.close();
…
```

Observamos no momento da chamada do método de envio da mensagem, *con.call(message, endepoint)*, que ele retorna um objeto, uma mensagem de resposta com o que estava sendo requisitado. O código quer saber o preço de um item vendido nas Lojas Americanas e aguarda, portanto, o valor como retorno. Com o objeto *response* as partes da mensagem de retorno são acessadas e o objeto em questão é recuperado a partir da informação do *name* criado para ele ao método *getChildElements()*. Todos os elementos com aquela identificação são retornados para um objeto do tipo *Iterator*, e a partir dele o valor do preço requisitado foi recuperado pelo método *getValue()* do objeto *bodyElement*.

```
...
    SOAPPart sp = response.getSOAPPart();
    SOAPEnvelope se = sp.getEnvelope();
    SOAPBody sb = se.getBody();
    Iterator it = sb.getChildElements(bodyName);
    SOAPBodyElement bodyElement = (SOAPBodyElement)it.next();
    String price = bodyElement.getValue();
    System.out.print("O preço da requisitado é");
   System.out.println(price);
} catch (Exception ex)
{     ex.printStackTrace();
```

```
    }
    }
    }
```

A partir do exemplo apresentado podemos ter uma idéia do poder de mais uma API desenvolvida em Java para XML e como é bastante simples a sua utilização. Aplicações mais complexas podem ser feitas facilmente com o uso da JAXM. O exemplo apresentado não fez uso de um provedor de mensagens, porém a JAXM prover suporte a esse uso através de classes como *ProviderConnection*, *ProviderConnectionFactory* e *ProviderMetaData*.

4.2.4. Java API for XML Registries

Um registro XML é uma estrutura que habilita a construção e ajuda na identificação do posicionamento e da descoberta de *web services*. JAXR traz facilidades para a realização de transações B2B (*business-to-business*). Registros são compartilhados por organizações que poderão realizar transações de determinados *web services*. (Sun-JAXR, 2002)

JAXR permite aos desenvolvedores Java acessar variedades de registros XML. O modelo unificado de JAXR descreve os conteúdos e metadados dos registros XML. JAXR prover a possibilidade de criação de programas clientes de registros que sejam portáveis dentre vários modelos de registro.

Existem várias especificações para registros XML. Dentre elas estão o padrão *ebXML Registry and Repository*. Esse padrão é patrocinado pela *Organization for the Advancement of Structured Information Standards* (OASIS) e pelo *United Nations Centre for the Facilitation of Procedures and Practices in Administration, Commerce and Transport* (U.N./CEFACT), e o projeto *Universal Description, Discovery, and Integration (UDDI)*.

4.2.4.1. Arquitetura JAXR

A arquitetura do JAXR é baseada em um lado cliente que acessa um registro via um provedor JAXR, e o próprio provedor JAXR que prover acesso a um provedor de registros específico ou a uma classe de provedores baseados em uma especificação comum a vários modelos de registros.(Sun-JAXR, 2002)

JAXR contêm dois importantes pacotes, *javax.xml.registry*, que define classes e interfaces para o acesso aos registros, e *javax.xml.registry.infomodel*, que consiste em interfaces que definem o modelo para o JAXR. Essas interfaces definem os tipos de objetos que estarão no registro e a forma como eles estão interligados. A interface básica deste pacote é a *RegistryObject* que contem as seguintes subclasses *Organization*, *Service*, e *ServiceBinding*.(Sun-JAXR, 2002)

Logo abaixo, na tabela 4.1, estão especificadas as principais classes do pacote *javax.xml.registry*:

Interface (*javax.xml.registry*)	Descrição
Connection	A partir dessa interface o cliente deverá criar uma conexão com o provedor JAXR a fim de acessar o registro necessário.
RegistryService	Um objeto dessa interface é obtido através da conexão. Esse objeto prover ao cliente a interface necessária ao acesso do registro.
BusinessQueryManager	Essa interface é utilizada para realizar pesquisas sobre os serviços fazendo uso das interfaces do pacote *javax.xml.registry.infomodel*. Ainda há uma interface opcional, *DeclarativeQueryManager*, que habilita o cliente a fazer consultas com a sintaxe SQL (*Structured Query Language*).
BusinessLifeCycleManager	Essa interface permite realizar operações de mudança em um registro (*delete / update*).

Tabela 4.1. Classes do pacote *javax.xml.registry*

Na figura 4.6. é representada a arquitetura de JAXR. O cliente usa as interfaces da API para acessar algum provedor de registros (*ebXML*, *UDDI*, ...).

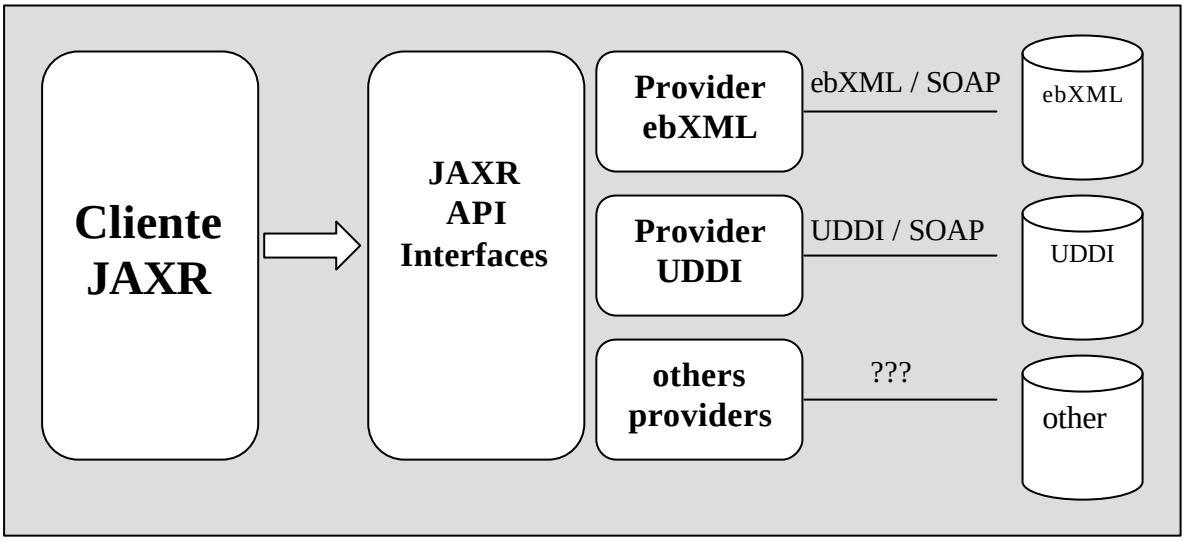

Figura 4.6. (Sun-JAXR, 2002) Arquitetura JAXR

4.2.4.2. Utilizando JAXR

Após a definição de alguns conceitos a respeito de registros XML e da própria API JAXR, será apresentada uma aplicação desenvolvida e localizada no tutorial desenvolvido pela Sun sobre JAXR (Sun-JAXR,2002) onde uma classe do lado cliente será implementada a fim de realizar uma pesquisa em um registro buscando informações sobre organizações que já efetuaram alterações de dados nesse registro, baseadas em uma *string* fornecida pelo usuário.

Abaixo estão declarados os pacotes que serão utilizados no exemplo.

```
import javax.xml.registry.*;

import javax.xml.registry.infomodel.*;

import java.net.*;
```

```
import java.util.*;

…
```

O primeiro passo para uma aplicação em JAXR é estabelecer uma conexão de acesso aos registros. Porém, antes da criação de uma conexão é necessário possuir permissão de acesso aos registros e para isso é necessário realizar o acesso a um provedor de registros e realizar o cadastro do usuário. É interessante realizar o cadastro em um provedor para que se possa entender melhor a aplicação. Como sugestão indicamos dois provedores UDDI criados para teste, *http://uddi.microsoft.com* e *http://uddi.ibm.com/testregistry/registry.html.* Abaixo está o código com o método principal

```
...
public class JAXRQuery {
Connection connection = null;
 public JAXRQuery() {}
public static void main(String[] args) {
        String queryURL = "http://uddi.ibm.com/testregistry/inquiryapi";
        String queryString = new String(args[0]);
        JAXRQuery jq = new JAXRQuery();
         jq.makeConnection(queryURL);
        jq.executeQuery(queryString);
}
…
```

Para a criação da conexão, é necessária que sejam setadas as propriedades de configuração da mesma, como o *host* e a porta de conexão, e propriedades como a(s) URL(s)

dos registros a serem acessados. No caso, é setada uma URL para o serviço de gerenciamento de consulta. Ao ponto em que as propriedades foram setadas à conexão, a mesma pode ser criada.

```
…
 public void makeConnection(String queryUrl) {
        ResourceBundle registryBundle = ResourceBundle.getBundle("AppExemplo");
        String httpProxyHost = registryBundle.getString("http.proxyHost");
        Properties props = new Properties();
        props.setProperty("javax.xml.registry.queryManagerURL",queryUrl);
        props.setProperty("com.sun.xml.registry.http.proxyHost",httpProxyHost);
        props.setProperty("com.sun.xml.registry.http.proxyPort",”8080”);
       try {
            ConnectionFactory factory = ConnectionFactory.newInstance();
            factory.setProperties(props);
            connection = factory.createConnection();
        } catch (Exception e) {
            e.printStackTrace();
            if (connection != null) {
                try {  connection.close();
                } catch (JAXRException je) {}
            }
        }
    }
…
```

O modo mais simples do cliente JAXR utilizar um registro é fazendo uma consulta a ele a fim de obter informações sobre organizações e os serviços que prestam.(Sun-JAXR, 2002) Depois da criação da conexão, a mesma será usada para obter um objeto *RegistryService* e a partir daí fará acesso às interfaces necessárias a aplicação.

A interface *BusinessQueryManager* possui métodos de pesquisa utilizando o modelo de informação JAXR. A maioria desses métodos retorna um tipo *BulkResponse*, que é uma coleção de objetos baseados em critérios fornecidos pelos argumentos do método. O método utilizado no nosso exemplo foi *findOrganizations*, que retorna uma lista de organizações baseados na *string* de consulta informada no início da aplicação e uma coleção com a classificação de retorno da consulta.

```
…
public void executeQuery(String qString) {
    RegistryService rs = null;
    BusinessQueryManager bqm = null;
    try {
        rs = connection.getRegistryService();
        bqm = rs.getBusinessQueryManager();
        Collection findQualifiers = new ArrayList();
        findQualifiers.add(FindQualifier.SORT_BY_NAME_DESC);
        Collection namePatterns = new ArrayList();
        namePatterns.add("%" + qString + "%");
        BulkResponse response = bqm.findOrganizations(findQualifiers, namePatterns, null,
                                                      null, null, null);
...
```

Depois de realizada a consulta, caso haja retorno, os dados serão exibidos após serem recuperados por *collections*.da linguagem Java, no caso objetos do tipo *Collection* e *Iterator*. São apresentados os dados das organizações que participam do registro pesquisado, assim como o contato primário de cada organização através do método *getPrimaryContact()*.

```
...
        Collection orgs = response.getCollection();
        Iterator orgIter = orgs.iterator();
        if (!(orgIter.hasNext())) {
            System.out.println("Nenhuma organização foi encontrada");
        } else while (orgIter.hasNext()) {
            Organization org = (Organization) orgIter.next();
            System.out.println("Nome: " + getName(org) );
            System.out.println("Descrição org: " + getDescription(org));
            System.out.println("Key id : " + getKey(org));
            User pc = org.getPrimaryContact();
            if (pc != null) {
                PersonName pcName = pc.getPersonName();
                System.out.println(" Contact name: " + pcName.getFullName());
                Collection phNums =  pc.getTelephoneNumbers(null);
                Iterator phIter = phNums.iterator();
                while (phIter.hasNext()) {
                    TelephoneNumber num = (TelephoneNumber) phIter.next();
                    System.out.println("  Phone number: " + num.getNumber());
```

```
            }
            Collection eAddrs = pc.getEmailAddresses();
            Iterator eaIter = eAddrs.iterator();
            while (eaIter.hasNext()) {
                EmailAddress eAd = (EmailAddress) eaIter.next();
                System.out.println("  Email Address: " + eAd.getAddress());
            }
        }
...
```

Abaixo está o resultado principal do nosso exemplo. São recuperados os serviços providos pelas organizações retornadas na pesquisa realizada. Dados como o nome, a descrição e a URL de acesso do serviço podem ser recuperados através de pesquisas a registros XML.

```
...
    Collection services = org.getServices();
        Iterator svcIter = services.iterator();
        while (svcIter.hasNext()) {
            Service svc = (Service) svcIter.next();
            System.out.println(" Service name: " +  getName(svc));
            System.out.println(" Service description :  "+ getDescription(svc) );
            Collection serviceBindings = svc.getServiceBindings();
            Iterator sbIter = serviceBindings.iterator();
            while (sbIter.hasNext()) {
                ServiceBinding sb = (ServiceBinding) sbIter.next();
```

```
                System.out.println("  Binding Description: "+ getDescription(sb));
                System.out.println("  Access URI: " + sb.getAccessURI());
            }
        }
    }
  } catch (Exception e) {
    e.printStackTrace();
  } finally  {
    if (connection != null) {
        try {                    connection.close();
        } catch (JAXRException je) {}
    }
  }
}
…
```

Abaixo estão as implementações dos métodos *getName*, *getDescription* e *getKey* que fazem parte da interface *RegistryObject* e que irão retornar os valores encontrados na pesquisa do registro sobre as organizações. Caso o objeto não possua valor a ser recuperado ele lançará uma exceção *NullPointerException* por estar invocando um método a partir de um objeto com valor *null*.

```
…
    private String getName(RegistryObject ro)   throws JAXRException {
        try {
            return ro.getName().getValue();
        } catch (NullPointerException npe) {
```

```
            return "Sem nome";
        }
    }
    private String getDescription(RegistryObject ro) throws JAXRException {
        try {
            return ro.getDescription().getValue();
        } catch (NullPointerException npe) {
            return "Sem descrrição";
        }
    }
    private String getKey(RegistryObject ro) throws JAXRException {
        try {
            return ro.getKey().getId();
        }
  catch (NullPointerException npe) {
            return "Sem Key Id";
        }
    }
}
```

Muitas outras alterações podem ser feitas afim de que se possam realizar alterações nos dados dos registros (inserção, alteração e exclusão), e para isso é necessário que fazer uma requisição para receber autorização de alteração dos registros a partir do nome de usuário e da senha cadastrados na hora da requisição de acesso ao serviço realizado antes do início da implementação do nosso exemplo.

CONCLUSÃO

Muitas organizações ainda não fazem a utilização de Java e buscam tecnologias alternativas. Porém, Java já conquistou vários adeptos e organizações como a IBM, a Oracle e até mesmo a Microsoft buscam oferecer, dentro de seus ambientes, facilidades para o desenvolvimento de aplicações *web* fazendo uso de tecnologias com Java.

XML é um padrão da W3C bastante simples de ser utilizado e traz bastantes facilidades de uso em aplicações *web*. Os padrões baseados em XML estão apenas iniciando. Podemos citar padrões de metadados como o RDF, *Resource Description Framework*, (http://www.w3.org/RDF/) e linguagens para representação de ontologias como OWL, *Web Ontology Language*, (http://www.w3.org/2001/sw/) .

XML e Java trazem conceitos bastante complexos quando são usadas em conjunto para o desenvolvimento de uma aplicação *web*. Porém, Java vem através de suas APIs amenizar essa complexidade de desenvolvimento tornando bem mais fácil e legível esses tipos de aplicações para desenvolvedores e usuários.

A partir deste trabalho pude obter a informação necessária para desenvolvimento de aplicações web fazendo uso de Java e XML. Java prover várias implementações de diferentes modos de utilização de documentos XML. Em comum essas aplicações utilizam a JAXP para processamento e validação de documentos XML. Já outras APIs são utilizadas para propósitos mais específicos. A JAXR é utilizada para acesso a registros XML a fim de

identificar serviços e prover a iteração entre aplicações e serviços disponíveis na web, os *web services*.

Outras APIs como JAX-RPC e JAXM podem ser utilizadas em conjunto caso não se queira utilizar a implementação de envio de mensagens para chamada de métodos remotos provida pela JAX-RPC. JAXM é utilizada para comunicação entre aplicações utilizando-se provedores de mensagens baseados em XML e a representação das mensagens é feita baseada no protocolo SOAP.

É realmente necessário que o desenvolvimento de aplicações com Java e XML obedeçam a um padrão em comum de modo a garantir compatibilidade entre os *web services* e que essas aplicações sejam independentes de plataformas, caso contrário a missão dessas tecnologias não será cumprida.

Certamente Java e XML são tecnologias que continuarão a liderar o mercado de desenvolvimento *web* por muito tempo, apresentado cada vez mais inovações a todos os que se dispõe a trabalhar com elas, e fazendo com que aqueles que ainda são contra as essas duas tecnologias começar a se unir e ajudar a aumentar o conhecimento e a tecnologia que ainda pode ser aprimorada a partir delas.

BIBLIOGRAFIA

(ALLAMRAJU, 2000) ALLAMRAJU, Subrahmanyam et al. **Professional Java Server Programming - J2EE Edition**. Wrox Press, 2000.

(GABRICK, 2000) GABRICK, Kurt A., WEISS, David B. **J2EE and XML Development**. Hanning, 2000.

(MARTIN, 2000) MARTIN, Didier et al. **Professional XML**. Wrox Press, 2000

(BROWN, 2001) BROWN, Simon et al. **Professional JSP 2nd Edition**. Wrox Press, 2001.

(WUTKA, 2001) WUTKA, Mark. **Using Java 2 - Enterprise Edition**. Indianapolis: QUE, 2001

(GRAHAM, 2002) GRAHAM, Steve et al. **Building Web Services with Java**. Indianapolis: SAMS, 2002.

(Sun-WebServices, 2002) Sun Microsystems, Inc. **Tutorial sobre Web Services**. Disponível em: http://java.sun.com/webservices/docs/1.0/tutorial/index.html Acesso: 4 de Novembro de 2002.

(Sun-JAXP, 2002) Sun Microsystems, Inc. **Tutorial sobre a Java API for XML Processing (JAXP)**. Disponível em: http://java.sun.com/xml/jaxp/index.html. Acesso: 25 de Outubro de 2002.

(Sun-JAXM, 2002) Sun Microsystems, Inc. **Tutorial sobre a Java API for XML Messaging (JAXM)**. Disponível em: http://java.sun.com/xml/jaxm/index.html Acesso: 25 de Outubro de 2002.

(Sun-JAXR, 2002) Sun Microsystems Inc. **Tutorial sobre a Java API for XML Registries (JAXR)**. Disponível em: http://java.sun.com/xml/jaxr/index.html. Acesso: 25 de Outubro de 2002.

(Sun-JAX-RPC, 2002) Sun Microsystems, Inc. **Tutorial sobre a Java API for XML-based RPC (JAX-RPC)**. Disponível em: http://java.sun.com/xml/jaxrpc/index.html. Acesso: 25 de Outubro de 2002