Pós-Graduação em Ciência da Computação

"Processamento de documentos XML com DOM e SAX: uma análise comparativa"

Por

Maísa Soares dos Santos

Dissertação de Mestrado

Universidade Federal de Pernambuco

posgraduacao@cin.ufpe.br

www.cin.ufpe.br/~posgraduacao

Recife, Fevereiro/2003

UNIVERSIDADE FEDERAL DE PERNAMBUCO
CENTRO DE INFORMÁTICA
PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

MAISA SOARES DOS SANTOS

"Processamento de documentos XML com DOM e SAX: uma análise comparativa"

ESTE TRABALHO FOI APRESENTADO À PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO DO CENTRO DE INFORMÁTICADA UNIVERSIDADE FEDERAL DE PERNAMBUCO COMO REQUISITO PARCIAL PARA OBTENÇÃO DO GRAU DE MESTRE EM CIÊNCIA DA COMPUTAÇÃO.

ORIENTADOR: ROBERTO SOUTO MAIOR DE BARROS

Agradecimentos

À fonte da vida, meu guia e sustentador, Deus.

Aos que muitas vezes abriram mão de si, para ajudar a realizar os meus sonhos, Painho (João Zito) e Mainha (Morena).

Aos sempre companheiros e amigos, meus irmãos: Susana, Ronan, Samara e Sam.

Ao presente de Deus, que se doou para que eu pudesse terminar este mestrado, meu amorzinho, Hélio.

Àquela, que abriu a porta da sua casa e do seu coração para mim, D. Zita.

Às amigas, que dividiram comigo tantos momentos de suas vidas, Val, Craudia e Lela.

Aos que dividiram comigo as angústias e incertezas desta caminhada, meus colegas da UESB e do CEFET.

Ao casal que se importou comigo, Walter e Cátia.

Àqueles que se dispuseram a orar por mim e muitas vezes me incentivaram, os irmãos da Igreja Batista Peniel, em especial: Liu, Ricardo, Josembergue, Erlinho, Pr. Jair.

Àquele que me incentivou mesmo quando eu não via saída, o professor e orientador Roberto.

Ao casal, que primeiro me ajudou em Recife, Chico e Mara.

Àquele, que durante muito tempo foi o único amigo no CIn, Gil.

Aos colegas descobertos em Recife: Neide, Erilson, Marília, Bruno, Ismênia.

Aos que me receberam com carinho, os amigos da Igreja Batista da Várzea.

Valeu!

"Aprendi que se depende sempre
De tanta muita diferente gente
Toda pessoa sempre é as marcas
Das lições diárias de outras tantas pessoas
E é tão bonito quando a gente entende
Que a gente é tanta gente
Onde quer que a gente vá
É tão bonito quando a gente sente
Que nunca está sozinho
Por mais que pense estar."

(Gonzaguinha)

Resumo

XML (eXtensible Markup Language) tem sido um padrão bastante usado para armazenar, manipular e trocar dados. Mas para serem úteis, esses dados precisam estar disponíveis de alguma maneira para a aplicação. Existem duas APIs (*Application Programming Interface*) que disponibilizam dados XML para as aplicações, uma baseada em objetos e outra baseada em eventos. Essas são representadas, respectivamente, por DOM - Document Object Model e por SAX - Simple API for XML. A escolha de qual API utilizar será baseado nos requisitos das aplicações e nas características das APIs.

Estas APIs são implementadas por ferramentas chamadas *parsers*. Cada *parser* possui suas próprias características. A escolha do *parser* é um importante critério para o desempenho das aplicações, pois grande parte do processamento ficará concentrada nele.

Este trabalho tem como objetivo realizar um estudo comparativo entre as APIs DOM e SAX, mostrando as características destas APIs, suas vantagens e desvantagens, onde cada uma obtém melhor desempenho, e seu comportamento em uma aplicação. Adicionalmente é feita uma análise de algumas ferramentas de processamento de documentos XML encontradas no mercado, mostrando as características e a performance de cada uma.

Palavras-Chave: XML, API , DOM, SAX, Parser.

Abstract

XML (eXtensible Markup Language) is becoming a standard for storing, manipulating and interchanging data. However, to become useful, these data need to be available to the application. There are two kinds of APIs (Application Programming Interface) that they make XML documents available to applications, one is objects-oriented - DOM (Document Object Model) and the other is events oriented - SAX (Simple API for XML). The choice of which API to use will be based on the requirements of the applications and the features of the APIs.

These APIs are implemented using parsers, which have their own strengths, limitations, and performances. Choossing an appropriate API and parser directly affect the performance of the applications, because they are responsible for a considerable part of the work.

This work compares the DOM and SAX API's, presenting their strengths and limitations and their behaviour in applications. Additionally, a number of parsers currently available are compared with regards to their features and performance.

Key Word: XML, API, DOM, SAX, Parser.

Sumário

Índice de Figuras

Índice de Gráficos

Índice de Listagens

Índice de Tabelas

Lista de Siglas

API	Application Programming Interface
DOM	Document Object Model
DTD	Document Type Definition
HTML	HyperText Markup Language
JDK	Java Development Kit
SAX	Simple API for XML
SDK	Software Development Kit
SGML	Standard Generalized Markup Language
URI	Uniform Resource Identifier
URL	Uniform Resource Locater
XML	eXtensible Markup Language
W3C	World Wide Web Consorium

Capítulo 1 - Introdução

Neste capítulo são apresentados os motivos que levaram ao desenvolvimento deste trabalho, bem como os objetivos da sua realização. Este capítulo também mostra a visão geral de toda a dissertação com o conteúdo dos capítulos seguintes.

1.1 Motivação

A Internet é um meio de comunicação poderoso e interativo. Ela aceita diversos tipos de aplicações, o que a torna indispensável ao mundo moderno. Inicialmente, ela foi solução para a publicação de documentos científicos [Marchal 2000]. Hoje, é possível encontrar na Internet aplicações como lojas on-line, operações bancárias eletrônicas, comércio e fórum variados.

A popularidade da Internet se deve em grande parte a HTML (*HyperText Markup Language*), uma linguagem simples e fácil de usar que permite a construção de páginas Web. Ela contém marcações especiais para formatar o conteúdo da página a ser apresentada.

No entanto, a HTML não atende às necessidades constantes e crescentes das aplicações. Para resolver as limitações da HTML, o W3C (*World Wide Web Consorium*) desenvolveu uma linguagem de marcação extensível, a XML (*eXtensible Markup Language*) [Bray 2000].

XML é uma meta linguagem que tem se tornado padrão para armazenamento, manipulação e troca de dados. Para dar suporte a estas funções, XML incorpora alguns padrões. Entre eles podem ser citados os padrões utilizados para acessar e manipular documentos XML: DOM (*Document Object Model*) [Byrne 1998] e SAX (*Simple API for XML*) [Megginson 2001].

Esses padrões são usados para processar documentos XML. Através deles uma aplicação acessa dados XML. Eles permitem que arquivos XML sejam lidos e suas informações sejam disponibilizadas para uma aplicação.

Apesar de ambos serem usados para processar documentos XML, DOM e SAX possuem características diferentes. Existem aplicações onde DOM mostra melhor desempenho e há aplicações onde SAX é a solução mais indicada.

DOM e SAX foram desenvolvidos em torno de interfaces. Para que uma aplicação possa utilizá-los é necessário o uso de um analisador (*parser*) que os suporte. Os *parsers* podem implementar apenas um destes padrões ou dar suporte a ambos. Cada *parser* possui suas próprias características e propriedades.

A maioria dos *parsers* XML encontrados no mercado são implementados na linguagem Java [Harold 2002]. Java, assim como XML, possui características como portabilidade e independência de plataforma que possibilitam o desenvolvimento de aplicações para Internet [Deitel 2001]. Neste trabalho os códigos e os *parsers* mostrados são escritos em Java.

1.2 Objetivo

Os objetivos deste trabalho são:

- Analisar as características das APIs DOM e SAX – cada API possui um conjunto de características que são adequadas a certos tipos de aplicações. A apresentação destas características pode auxiliar o desenvolvedor a escolher qual API se adequa melhor a sua aplicação;
- Apresentar uma implementação das APIs DOM e SAX em uma aplicação real;
- Analisar a performance de uma aplicação que utiliza as APIs DOM e SAX.;
- Analisar as características dos *parsers* – uma aplicação possui um conjunto de necessidades quanto ao uso do XML: algumas não necessitam realizar as validações dos documentos, outras precisam validar os seus documentos de acordo com regras definidas em DTDs (*Document Type Definition*) [Buck 2000] ou XML *Schema* [Fallside 2001], algumas necessitam de suporte a *namespace* [Bray 1999], entre outras. Todas essas características podem ser encontradas totalmente ou parcialmente nos *parsers* que estão no mercado. Alguns *parsers* possuem algumas características que outros não possuem. Saber se um determinado *parser* atende a todas as necessidades de uma determinada aplicação é muito importante no processo de escolha;
- Analisar a performance dos *parsers* – os *parsers* consomem uma parcela importante dos recursos do sistema (tempo de processamento e memória) de uma aplicação XML, pois são responsáveis por disponibilizar os dados XML para as mesmas. Um estudo da performance dos parsers pode auxiliar o projetista a escolher o parser que venha oferecer melhor desempenho a sua aplicação.

Neste trabalho pretendemos mostrar as características das duas principais APIs para processamento de documentos XML: DOM e SAX. Indicando onde cada uma deve ser aplicada e como obter o melhor aproveitamento das ferramentas que implementam estas APIs.

1.3 Visão Geral da Dissertação

Este trabalho é composto de oito capítulos, incluindo este introdutório. Os capítulos estão organizados da seguinte forma:

Capítulo 2 – Apresenta um estudo sobre a linguagem de marcação XML. Mostra o que é uma linguagem de marcação e qual a diferença entre as linguagens de marcação XML, SGML e HTML, e são descritas as características de XML, como seus documentos são formados, e alguns padrões que acompanham a linguagem XML.

Capítulo 3 – Aborda a interface de programa de aplicação baseada em objetos, DOM. Nesse capítulo são descritas as características, funcionalidade, bem como as interfaces e métodos básicos que compõem o *Core* (núcleo) de DOM.

Capítulo 4 – Aborda os níveis 2 e 3 de DOM: quais as diferenças entre os níveis, quais os módulos que compõem cada nível, as características e as principais interfaces definidas em cada módulo.

Capítulo 5 – Apresenta um estudo sobre a interface de programa de aplicação baseada em eventos, SAX. Este capítulo mostra as características, a funcionalidade, e as interfaces e métodos que compõem esta API.

Capítulo 6 – Apresenta uma análise comparativa entre as APIs DOM e SAX. Nesse capítulo são mostrandas as categorias de aplicações XML onde DOM e SAX podem ser usadas, uma comparação entre estas APIs destacando suas vantagens e desvantagens e quando cada uma deve ser aplicada, o estudo de caso realizado, e a performance das APIs na aplicação desenvolvida no estudo de caso.

Capítulo 7 – Aborda um estudo sobre *parsers* XML que suportam as APIs DOM e SAX. São mostradas características gerais comuns a todos os *parsers* e também características particulares dos *parsers*: JAXP [JAXP 2002b], Xerces 2 [Xerces 2002], XML for Java [XML4J 2002], Crimson [Crimson 2001] e GNU JAXP [GNU 2001], além da análise da performance destes *parsers*.

Capítulo 8 – Apresenta as conclusões sobre este trabalho, bem como, as principais contribuições e sugestões para trabalhos futuros.

Capítulo 2 - XML

Neste capítulo é apresentado um estudo sobre a linguagem de marcação XML: o que é uma linguagem de marcação, a diferença entre as linguagens de marcação SGML, HTML e XML, as características de XML, como seus documentos são formados, e os seus padrões acompanhantes.

2.1 Introdução

XML é atualmente o padrão para distribuição de informações na World Wide Web. A sua sintaxe flexível permite a descrição de virtualmente todo tipo de informação [Young 2000].

XML é uma linguagem de marcação e como tal é composta por marcas. Uma marca ou marcação é qualquer informação adicional acrescentada no texto de um documento [Light 1999]. Por exemplo, para controlar a aparência de um documento na tela ou na impressora, os processadores usam marcas que definem o tipo e tamanho da fonte, negrito, itálico, sublinhado, tamanho da página, margens, ou até marcações mais sofisticadas como rodapé, anotações, tabelas de conteúdos, registros de índice, entre outros.

Uma linguagem que só contém este tipo de marcação é chamada de linguagem de formatação, pois simplesmente ajuda a apresentar as informações e não fornecem uma estrutura mais detalhada para o documento. HTML é um exemplo de linguagem de formatação.

XML, entretanto, é composta de marcas que indicam o sentido da porção de texto dentro do documento. Linguagens com este tipo de marcação são conhecidas como linguagem de marcação generalizada. Um outro exemplo desse tipo de linguagem é a SGML (*Standard Generalized Markup Language*).

As linguagens de marcação generalizada possuem as seguintes características:

- Estruturas de documentos - linguagens de marcação generalizada são normalmente lógicas em sua ordem e altamente estruturadas. Elas incluem regras específicas que afirmam onde várias estruturas de documento devem e precisam ser colocadas.
- Marcas (*tags*) – são instruções envolvidas por < >, e normalmente são alto explicativas.
- Atributos – são itens adicionados às marcas para dar clareza as mesmas.
- Entidades – são usadas para manipular grupos de dados que são tratados como uma unidade. Uma entidade é formada por dados que não pertencem à linguagem. Por exemplo, uma entidade pode ser um arquivo inteiro ou simplesmente uma string de texto.

- Comentários – são informações adicionais que não devem ser vistas depois que o documento for processado.

2.1.1 SGML

SGML foi a primeira linguagem de marcação moderna a ser criada. Ela foi inventada em 1969 com o nome de General Markup Language (GML). A GML era uma linguagem de referencial própria para a formulação de estruturas de um conjunto arbitrário de dados, e seu propósito era ser uma meta-linguagem [Martin 2000]. Em 1986 a GML tornou-se a SGML ou Standard Generalized Markup Language [Martin 2000].

A HTML e a XML são derivadas da SGML. A mãe das linguagens de marcação fornece um esquema de marcação simples, independente de plataforma e extremamente flexível [Light 1999]. A SGML não impõe seu próprio conjunto de *tags*, ela permite que as *tags* sejam definidas para descrever dados. Por isto, a SGML é descrita como uma meta-linguagem – uma linguagem para definir linguagens de marcação.

A SGML define o que é permitido em cada documento através da *definição de tipo de documento* (DTD). Na DTD são definidos os tipos de elementos permitidos dentro do documento, as características de cada tipo de elemento, as notações e entidades que podem ser encontradas dentro do documento [Light 1999].

SGML parece ser a linguagem de marcação perfeita para descrever documentos Web [Young 2000]. Entretanto, a complexidade desta linguagem compromete a eficiência da distribuição de informações na Internet. A flexibilidade e a abundância de características providas pela SGML tornam difícil escrever software para processar e mostrar informações SGML em *browsers* [Young 2000]. Para resolver este problema o Consórcio World Wide Web (W3C) criou a XML.

2.1.2 HTML

A HyperText Markup Language (HTML) é a linguagem de marcação mais popular para criação de páginas na Web. Essa linguagem foi desenvolvida em 1992 por Tim Berners Lee e Robert Caillau no CERN, que é o Centro Europeu de Pesquisas de Física de Partículas. Ela foi inicialmente desenvolvida para auxiliar na divulgação de textos científicos. Atualmente, a maioria das páginas Web é escrita em HTML [Marchal 2000].

A HTML é uma linguagem simples, fácil de aprender e elegante, composta por *tags* especiais para formatação de texto. HTML é uma aplicação da SGML, o que significa que ela foi definida como uma DTD da SGML.

Apesar da sua popularidade, HTML apresenta algumas limitações. A linguagem de marcação mais usada na Internet tem um conjunto de tags pré-definido que não pode ser estendido pelos desenvolvedores de documentos HTML. As *tags* HTML apenas apresentam páginas nos *browsers*, elas não dão sentido aos dados.

2.2 XML

XML – eXtensible Markup Language é uma linguagem de marcação apropriada para representar dados. Ela foi desenvolvida em 1996 por um grupo de trabalho do W3C como um subconjunto da SGML. Seu objetivo é permitir que a SGML genérica seja servida, recebida e processada na Web da mesma forma que é possível com HTML. XML foi projetada para facilitar a implementação e interoperabilidade tanto com SGML como com HTML [Bray 2000]. XML é a solução para a complexidade da SGML e supera as limitações impostas pela HTML.

A XML é muito menos complexa do que a SGML, mas mantém os benefícios que a SGML oferece. Ela é uma meta linguagem, o que significa que XML provê recursos para a definição de outras linguagens. Suas marcações são generalizadas, o que permite a construção de documentos autodescritivos. Um banco, por exemplo, poderia guardar informações sobres seus clientes em um documento XML como mostra a Listagem 2.1.

Listagem 2.1 – Documento XML que armazena dados de clientes de um banco

```
<cliente>
       <nome> João Francisco </nome>
       <cpf>23988475-98</cpf>
       <endereço>Rua A, no. 23</endereço>
       <telefone>3423-9999</telefone>
       <conta_corrente>
              <numero> 3240-0</numero>
              <saldo>2135,90</saldo>
       </conta_corrente>
</cliente>
```

A XML é uma forma da solucionar problemas encontrados na HTML. Com a XML é possível controlar melhor a aparência das páginas na Internet, acessar informação do lado

do cliente, utilizar vários tipos de links, controlar melhor páginas Web grandes, entre outros.

A HTML coloca a formatação e o conteúdo dentro do mesmo documento. Isso cria dificuldades para modificar a aparência das páginas HTML. Por exemplo, para mudar o tipo de fonte do texto contido na tag <H3> é necessário percorrer todo o documento em busca dessas *tags* e definir o novo tipo de fonte desejado em cada *tag* <H3>. Na XML, a formatação pode ficar em um arquivo separado do conteúdo, este arquivo é chamado de folha de estilo. Na folha de estilo é definida a formatação de cada *tag* do documento. Para mudar a aparência de uma página, basta modificar a folha de estilo. Para mudar, por exemplo, a aparência da *tag* <H3> basta alterá-la uma única vez na folha de estilo.

A XML exige menos esforço no servidor Web devido à capacidade de acessar informação do lado do cliente. Imagine uma página HTML com um formulário para consulta de livros de uma biblioteca. O formulário envia uma solicitação de pesquisa para o servidor toda vez que o usuário faz uma consulta. Com a XML, o servidor enviaria para o cliente um documento que poderia conter a lista de todos os livros e apenas as informações solicitadas naquele momento seriam mostradas. Novas consultas seriam feitas localmente no documento, sem acessar o servidor.

A XML proporciona um controle melhor sobre documentos grandes. Um documento XML pode ser subdividido em segmentos, e ser apresentado como um único documento com vários níveis.

XML permite definir vários tipos de links. A HTML permite apenas links unidirecionais, enquanto que com XML é possível criar o que se chama de link estendido. Links estendidos são links multidirecionais que podem ter várias fontes anexadas e fornecem acesso a uma cadeia de páginas Web ou permitem ao usuário abrir várias janelas.

Outra vantagem da XML é a possibilidade de usar o sistema de caracter Unicode[1], que suporta a representação de todas as principais linguagens do mundo.

Esses recursos oferecidos pela XML são assegurados pelos padrões que acompanham a linguagem. Alguns desses padrões são descritos na Seção 2.4.

[1] http://www-cgrl.cs.mcgill.ca/~luc/standards.html

2.2.1 Documento XML

Para estruturar os dados e utilizar os recursos XML é preciso criar um documento XML. Este documento é textual, composto de unidades de armazenamento chamadas entidades, que contém dados analisados ou não-analisados. Os dados analisados são compostos de caracteres, alguns formam dados de caractere (a informação propriamente dita) e alguns formam marcação. A marcação codifica uma descrição da estrutura lógica e layout de armazenamento do documento [Bray 2000].

Listagem 2.2 – Documento XML que armazena dados de livros

```
<? xml version =”1.0” ?>
<!– Comentário: documento que guarda dados de livros -->
<LIVRARIA>
      <LIVRO>
            <TITULO> XML </TITULO>
            <AUTOR>
                  <P_NOME> Fulano </P_NOME>
                  <SOBRENOME> da Silva </ SOBRENOME>
            </AUTOR>
            <CATEGORIA> Informática </CATEGORIA>
            <PRECO MOEDA=”real”> 50,00 </PRECO>
      </LIVRO>
</LIVRARIA>
```

A Listagem 2.2 mostra um documento XML que guarda dados de livros de uma livraria, através dele é possível obter o nome, o autor, a categoria e o preço de um livro.

2.2.2 Documento XML Bem Formado

Um arquivo XML é um documento XML se ele for bem formado. Um documento XML é dito bem formado se atender às seguintes condições estabelecidas pelo W3C na especificação XML 1.0 [Bray 2000]:

- A declaração XML, se presente no documento (o que é recomendado), deve ser a primeira linha do documento.
- Incluir um ou mais elementos (ver definição de elemento na Seção 2.3.5);
- Ter exatamente um elemento, chamado raiz, ou elemento do documento, que deve conter todos os outros elementos;
- O nome da *tag* de fim (`</LIVRO>`) deve coincidir com a *tag* de início (`<LIVRO>`);

- As *tags* devem ser aninhadas adequadamente, isto é, não pode haver sobreposição;
- Cada atributo de uma *tag* deve ter um nome exclusivo.
- Cada uma das entidades analisada referida direta ou indiretamente dentro do documento deve ser bem formada.

Um exemplo de um documento XML bem formado é mostrado na Listagem 2.2. A Listagem 2.3 é um exemplo de documento que não obedece às regras de boa formação da recomendação XML. Esse documento não é bem formado, pois falta o elemento raiz e há uma sobreposição no fechamento das tags `</NOME>` e `</SOBRENOME>`. A Listagem 2.4 mostra as mesmas informações, agora seguindo as regras de boa formação definidas pelo W3C.

Listagem 2.3 – Documento mal formado

```
<CLIENTE>
       <NOME>
              <PRIMEIRO_NOME>Fernando </ PRIMEIRO_NOME>
              <SOBRENOME>Santos
       </NOME> </SOBRENOME>
</CLIENTE>
<CLIENTE>
       <NOME>
             <PRIMEIRO_NOME>João</ PRIMEIRO_NOME>
       <SOBRENOME>Freitas
       </NOME> </SOBRENOME>
</CLIENTE>
```

Listagem 2.4 – Documento XML bem formado – correção da Listagem 2.3

```
<?xml version="1.0"?>
<CLIENTES>
       <CLIENTE>
              <NOME>
                     <PRIMEIRO_NOME>Fernando </ PRIMEIRO_NOME>
                     <SOBRENOME>Santos</SOBRENOME>
              </NOME>
       </CLIENTE>
       <CLIENTE>
              <NOME>
                     <PRIMEIRO_NOME>João</ PRIMEIRO_NOME>
                     <SOBRENOME>Freitas </SOBRENOME>
              </NOME>
       </CLIENTE>
</CLIENTES>
```

Qualquer documento XML bem formado tem que ser composto por elementos que formam uma árvore hierárquica simples com um único nó raiz, chamado de “raiz do

documento". Esta contém uma árvore de elementos secundários, que também tem um nó raiz singular chamado de "elemento do documento" [Martin et al 2000].

A raiz do documento tem sempre uma subárvore de elementos e pode conter instruções de processamento e/ou comentários. Esta subárvore tem como raiz o elemento do documento. A Figura 2.1 mostra a árvore hierárquica do documento da Listagem 2.4

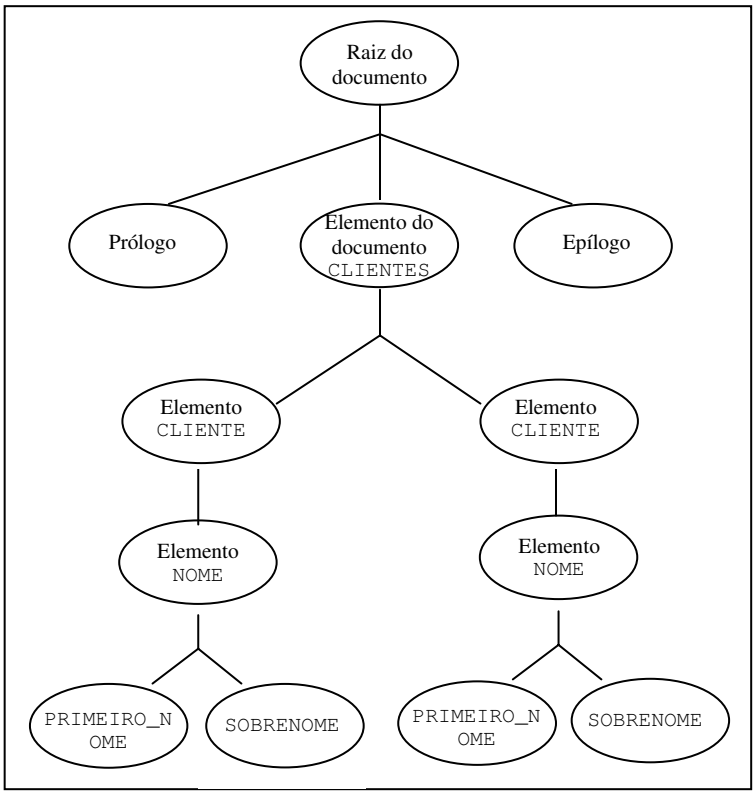


Figura 2.1 – Árvore que representa o documento da Listagem 2.4

Listagem 2.5 - **Exemplo de um documento válido**

```
<? xml version ="1.0" ?>
<!– Documento válido -->
<!DOCTYPE LIVRARIA [
   <!ELEMENT LIVRARIA (LIVRO*)>
   <!ELEMENT LIVRO (TITULO, AUTOR, CATEGORIA, PRECO)>
   <!ELEMENT TITULO (#PCDATA)>
   <!ELEMENT AUTOR (P_NOME, SOBRENOME)>
   <!ELEMENT P_NOME (#PCDATA)>
   <!ELEMENT SOBRENOME (#PCDATA)>
   <!ELEMENT CATEGORIA(#PCDATA)>
   <!ELEMENT PRECO(#PCDATA)>
   <!ATTLIST PRECO MOEDA CDATA #IMPLIED>
]>
<LIVRARIA>
       ...
</LIVRARIA>
```

2.2.3 Documento XML Válido

Um documento XML bem formado também pode ser um documento válido. O documento XML é considerado válido se houver uma definição de tipo de documento (DTD) ou XML *Schema* associado a ele, e se o documento estiver de acordo com esta DTD ou esquema. O exemplo da Listagem 2.5 mostra um documento válido. O código em destaque é a DTD usada para validar o documento mostrado na Listagem 2.2.

Um processador XML lê o documento e verifica se as construções do documento obedecem às regras de formação definidas na DTD ou no XML *Schema* associado a ele.

2.3 Estrutura do Documento XML

XML é uma linguagem de marcação simples. Seus documentos são compostos por *tags* de início, *tags* de fim, *tags* de elemento vazio, referência de entidade, referência de caracter, comentário, delimitadores de seção CDATA, declaração de tipo de documento e instruções de processamento.

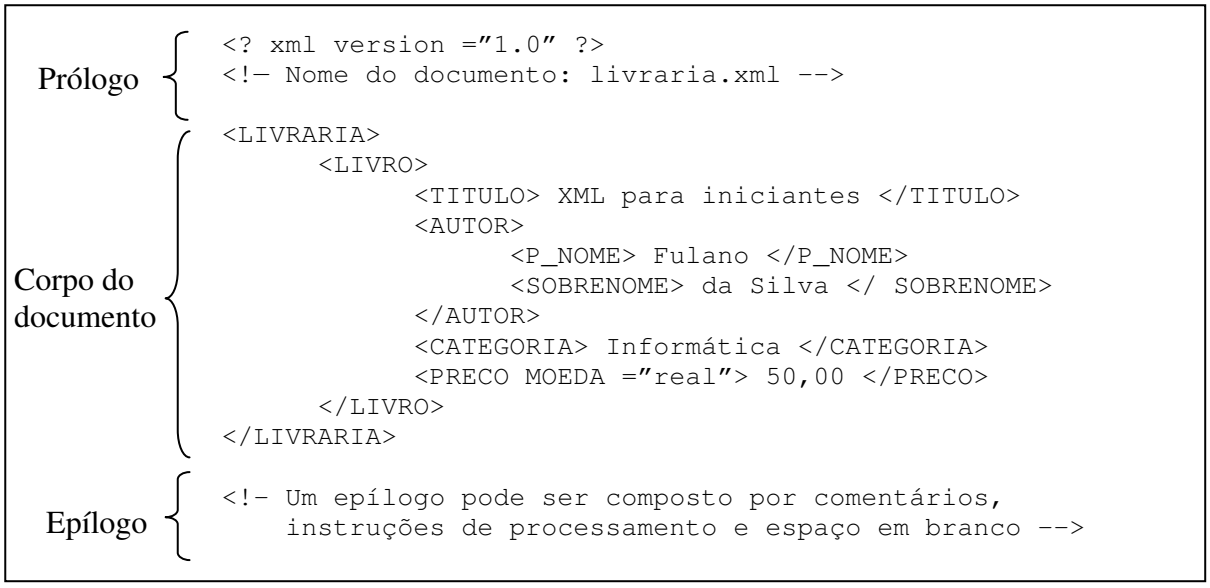


Figura 2.2 - Estrutura do documento XML

Um documento XML pode ser constituído de três partes, como mostra a Figura 2.2:

- um prólogo que é uma parte opcional;
- o corpo do documento ou elemento raiz; e

- um epílogo, parte opcional que aparece no final do documento, Ele pode ser composto de comentários, instruções de processamento, e/ou espaço em branco.

2.3.1 O Prólogo

O prólogo é usado para indicar o começo dos dados XML. No documento da Figura 2.2 o prólogo corresponde as duas primeiras linhas do documento, onde a primeira é a declaração XML e a segunda é um comentário:

```
<? xml version =”1.0” ?>
<!– Nome do documento: livraria.xml -->
```

O prólogo aparece bem no início do documento XML, é uma parte opcional que pode conter declaração XML, comentário, instrução de processamento, espaço em branco e declaração de tipo de documento. O trecho do documento XML abaixo é um exemplo de prólogo:

```
<? xml version =”1.0” ?>
<!– Documento válido -->
<!DOCTYPE LIVRARIA [
   <!ELEMENT LIVRARIA (LIVRO*)>
   <!ELEMENT LIVRO (TITULO, AUTOR, CATEGORIA, PRECO)>
   <!ELEMENT TITULO (#PCDATA)>
   <!ELEMENT AUTOR (P_NOME, SOBRENOME)>
   <!ELEMENT P_NOME (#PCDATA)>
   <!ELEMENT SOBRENOME (#PCDATA)>
   <!ELEMENT CATEGORIA(#PCDATA)>
   <!ELEMENT PRECO(#PCDATA)>
   <!ATTLIST PRECO MOEDA CDATA #IMPLIED>
]>
```

2.3.2 Declaração XML

Apesar de não ser obrigatório, todo documento XML deve ter uma declaração XML. Ela deve ser a primeira linha no documento XML, e é definida usando o elemento `<?xml?>`.

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

Uma declaração XML pode conter três atributos:

- version: indica a versão XML utilizada. Se uma declaração XML estiver sendo usada, este atributo torna-se obrigatório.
- encoding: indica a codificação de linguagem para o documento. É um atributo opcional.

- standalone: se o documento não se refere a qualquer entidade externa, esse atributo é define como “yes”, caso contrário, como “no”.

2.3.3 Comentário

Um comentário pode ser usado para incluir notas explicativas em um documento, que são ignoradas pelo processador. A sua sintaxe básica é

```
<!-- texto -->
```

onde *texto* pode ser qualquer cadeia de caracteres que não inclua “–-“ (hífen duplo).

```
<? xml version =”1.0” ?>
<!— Nome do documento: livraria.xml -->
<LIVRARIA>
<!— Início das informações sobre livros -->
<LIVRO>
      ...
      </LIVRO>
</LIVRARIA>
```

Um comentário pode aparecer em qualquer lugar em um documento, exceto dentro de uma marcação, ou antes, da declaração XML.

Exemplos de comentários inválidos:

```
<? xml version =”1.0” ?>
<LIVRARIA <!—Comentário Inválido -->>
<livro>
      .
      .
      .
      </LIVRO>
</LIVRARIA>
```

Um comentário não pode ser colocado dentro de uma tag

```
<!-- Arquivo: livraria.xml -->
<? xml version =”1.0” ?>
<LIVRARIA>
<LIVRO>
      .
      .
      .
      </LIVRO>
</LIVRARIA>
```

Um comentário não pode vir antes de uma declaração XML

2.3.4 Instrução de Processamento

As instruções de processamento contêm informações para o processador XML. Elas são delimitadas pelos caracteres <?, para indicar início, e ?>, para indicar o fim de uma instrução. A instrução que conecta uma folha de estilo a um documento XML é um

exemplo de instrução de processamento. Ela diz ao processador o tipo, o nome e endereço da folha de estilo usada para mostrar o documento.

```
<?xml version="1.0"?>
<? xml-stylesheet type="text/css" href="poema.css" ?>
<POEMA>
      ...
</POEMA>
```

As instruções de processamento precisam ser entendidas pelo processador XML, de modo que são dependentes do processador, e não embutidas na recomendação XML [Holzner 2001].

2.3.5 Elementos

Elementos são blocos – contêineres – de construções básicas de uma marcação XML. Eles podem conter outros elementos, dados de caracteres, referências, instruções de processamento, comentários e/ou seções CDATA [Martin et al 2000]. A sintaxe de um elemento é:

<tag_inicio> *conteúdo* </tag_fim>

O conteúdo do elemento é delimitado por um par de tags (exceto elemento vazio), uma tag de início e uma tag de fim. A tag inicial é formada pelo nome de tipo de elemento (uma string literal) fechada por um par de parênteses angulares ("< >"). Tags finais compreendem uma barra inclinada ("/") seguida do nome de tipo de elemento, fechada por um par de parênteses angulares ("< >"). Toda tag de fim tem que corresponder a tag inicial.

O nome de *tag* é uma string literal que pode começar com uma letra ou um sublinhado, seguido por letras, dígitos, sublinhados, hifens ou pontos. Mas não pode haver espaço em branco no nome de uma tag [Bray 2000].

Exemplos de nomes de tags válidas:

<documento>, <_cliente>, <LIVRO>, <Nome>

XML considera ilegais as tags seguintes:

<1000Registro>, <.documento>, <Nome Cliente>, <Livro(ID)>,
<número*código>

Os processadores XML são sensíveis a letras maiúsculas e minúsculas, portanto, as *tags* seguintes são todas diferentes:

<BOOK>, <book>, <Book>, <BooK>

2.3.6 Elemento Vazio

A sintaxe XML também permite que elementos sem conteúdo sejam criados: eles são chamados elementos vazios. Estes elementos são compostos de um nome de tipo de elemento, seguido de uma barra inclinada (“/”), mantida entre parênteses angulares (“< >”).

Os elementos abaixo são exemplos de elementos vazios.

```
<e-mail  ref=”mss2@cin.ufpe.br”/>
<TELEFONE NÚMERO=”(77) 421-4697” />
```

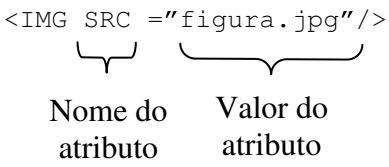
Para XML estes elementos são respectivamente equivalentes aos elementos:

```
<e-mail ref=”mss2@cin.ufpe.br”></e-mail>
<TELEFONE NÚMERO=”(77) 421-4697”></TELEFONE>
```

2.3.7 Atributos

Atributos são usados para adicionar informações a elementos, eles podem aparecer em *tags* de início ou em elementos vazios.

Um atributo tem nome e valor. Os nomes de atributo seguem as mesmas regras dos nomes de elemento (pode iniciar com uma letra ou um sublinhado e os próximos caracteres podem ser letras, dígitos, sublinhados, hífens e pontos).



O valor de atributo é sempre um texto, mesmo que seja atribuído um número a um atributo, ele será tratado com uma string de texto e delimitado com aspas ou apóstrofos.

O elemento `RETÂNGULO` abaixo possui dois atributos, `ALTURA` e `LARGURA`. O valor de cada atributo é, respectivamente, a cadeia de caracter “`5.0`” e “`10.0`”.

```
<RETÂNGULO ALTURA=’5.0’ LARGURA=’10.0’ />
```

Uma restrição ao uso de atributo é que nenhum nome de atributo pode aparecer mais de uma vez no mesmo elemento. Sendo assim, a seguinte construção é incorreta:

```
<RETÂNGULO LADO=’1.0’ LADO=’3.0’ />
```

2.3.8 Entidades

Uma entidade é uma seqüência de caracteres ou padrões de bits que podem ser tratados como uma unidade. As entidades são inseridas em um documento XML através de referência.

XML possui cinco entidades predefinidas para os caracteres ampersand (&), menor que (<), maior que (>), apóstrofo (') e aspas (”). Esses caracteres têm uma especial importância em um documento XML. O < e > compõem tags de marcação, as aspas e apóstrofos são usados para delimitar valores de atributos e o & inicia referências de entidade.

Referências de entidade são precedidas por um ampersand (&) e seguidas por ponto-e-vírgula (;).

Referência de entidade predefinida	**Caracter inserido**
&amp;	&
&lt;	<
&gt;	>
&apos;	‘
&quot;	“

Quando o documento XML é processado uma referência à entidade é substituída pela entidade correspondente. Por exemplo:

```
M&amp;M
```

seria substituído por:

```
M&M
```

Outro exemplo:

```
&quot;Jone&apos;s car&quot;
```

seria substituído por:

```
“Jone’s car”
```

Exceto estas cinco entidades definidas na especificação XML, todas as entidades têm que ser definidas antes de serem usadas em um documento. As entidades são definidas nas DTDs do documento.

2.3.9 Seção CDATA

Uma seção CDATA é usada para manter dados de caracter que não são analisados pelo processador XML. Nesta seção é possível incluir bloco de código ou marcação como parte

dos dados de caracter de um elemento, ou usar diretamente caracteres especiais de XML (< e &).

Uma seção CDATA inicia com a marcação `<![CDATA[` e termina com `]]>`. Ela pode ocorrer em qualquer lugar onde possa ocorrer dado de caracteres, mas não pode haver aninhamento de seção CDATA. A sua sintaxe é:

```
<![CDATA[ ...Conteúdo ... ]]>
```

onde *conteúdo* pode ser qualquer string de caracter que não inclua o literal "]]>".

Exemplo:

```
<Pagina>
     Exemplo de página HTML
     <![CDATA [
           <HTML>
                <HEAD>
                      <TITLE> Página teste </TITLE>
                </HEAD>
                <BOBY>
                          ....
                </BODY>
          </HTML>
     ]]>
</Pagina>
```

2.4 Padrões Acompanhantes

A XML é composta por vários padrões que a auxiliam no desempenho de seus objetivos. Esses padrões são chamados padrões acompanhantes. São eles:

- *XML Namespaces*: XML permite que usuários definam suas próprias *tags*, isto pode causar um conflito de nomes, isto é, nomes iguais podem ser usados para descrever coisas diferentes. Para resolver este problema foi criado o *namespace*, ou espaço de nome. Ele associa um proprietário aos elementos e permite a reutilização de estruturas padrões. O *namespace* define um prefixo para os nomes de *tags*, evitando confusões que possam surgir com nomes iguais para *tags* que definem dados diferentes. A definição do *namespace* é encontrada em uma URI (*Uniform Resource Identifier*) [URI 2000], chamada de *namespace* URI. Considerando o código:

```
<db:exemplo xmlns="http://namespaces.exemplo.com/">
          ...
</db:exemplo>
```

o prefixo da *tag* é `db`, e o *namespace* URI, representado pelo nome `xmlns`, é `http://namespaces.exemplo.com/`.

- *CSS e XSL*: XML define a estrutura e a semântica de um documento, e não seu formato [Holzner 2001]. Para exibir um documento XML é necessário usar uma folha de estilo, onde a configuração da aparência do documento é definida. XML é compatível com duas linguagens de folha de estilo: a CSS (Cascading Style Sheet) e a XSL (Extensible Stylesheet Language). CSS permite especificar a formatação dos elementos individuais, criar classes de estilo, configurar fontes, usar cores e até especificar a posição dos elementos na página. XSL é mais poderosa. Ela permite transformar um documento XML em algo inteiramente novo. XSL pode reordenar os elementos de um documento, mudá-los completamente, exibir alguns e ocultar outros, selecionar estilos com base não apenas nos elementos, mas também nos atributos dos elementos, selecionar estilos com base no local do elemento, entre outros. A XSL é dividida em duas partes: XSLFO para formatação e XSLT para transformação de documentos [Holzner 2001].
- *XLink e XPointers*: São duas partes de um padrão para fornecer um mecanismo que estabelece relacionamento entre documentos. Xlink permite que qualquer elemento se torne um link e os links podem ser multidirecionais. XPointers deve ser capaz de localizar partes específicas de outro documento sem forçá-lo a incluir marcação adicional no documento de destino [Holzner 2001].
- *DTD e XML Schema*: São padrões usados para validar documentos XML. Um conjunto de regas adicionais é definido e o documento para ser válido tem que obedecer estas regas. Estes padrões especificam a estrutura e sintaxe de documentos XML. O XML *Schema* é mais poderoso do que as DTDs. Com o XML *Schema*, além da sintaxe de um documento, é possível especificar os tipos de dados reais do conteúdo de cada elemento, herdar a sintaxe de outros documentos XML *Schema*, anotar documentos XML *Schema*, usar documentos XML *Schema* com vários namespaces, criar tipos de dados simples e complexos, especificar o número mínimo e máximo de vezes que um elemento

pode ocorrer, criar tipos de lista, criar grupos de atributos e muito mais [Holzner 2001].

- *DOM e SAX*: Os *parsers* XML utilizam uma interface para se comunicar com a aplicação que consome os dados XML. Esta interface permite que aplicações leiam documentos XML sem se preocupar com a sintaxe. Ela pode ser baseada em objetos ou baseada em eventos. DOM (Document Object Model) é uma API baseada em objetos que constrói na memória uma árvore de nós para representar o documento. SAX (Simple API for XML) é uma interface baseada em eventos. Um *parser* SAX, à medida que lê um documento, dispara eventos para cada entidade encontrada. Os próximos capítulos deste trabalho são dedicados a estas APIs.

2.5 Considerações Finais

Neste capítulo vimos que XML é a solução para os problemas inerentes a SGML e HTML. XML é uma metalinguagem que possui uma sintaxe simples, mas rígida. Um arquivo é um documento XML se obedece às normas de boa formação da linguagem. Além de bem formado, um documento XML pode ser validado com uma DTD ou com um esquema.

XML foi criada para ser usada na Internet. Para dar suporte aos seus objetivos alguns padrões foram desenvolvidos. Entre eles estão DOM e SAX, que são padrões usados para processar documentos. Nos próximos capítulos mostraremos como trabalhar com estas APIs.

Capítulo 3 - DOM Core

Neste capítulo é mostrado um estudo sobre a interface de programa de aplicação baseada em objetos, DOM. São descritas suas características, funcionalidade, bem como as interfaces e métodos que compõem esta API.

3.1 Introdução

Como já foi visto no Capítulo 2, XML tem sido um padrão bastante usado para armazenar informações. Mas para serem úteis a uma aplicação estes dados precisam estar disponíveis para esta aplicação de alguma maneira.

Tendo em vista o acesso aos dados de um documento XML, foram desenvolvidos vários analisadores (*parsers*) XML. Entretanto, cada *parser* tinha uma forma particular de processar os documentos e sofriam alterações constantes [Holzner 2001].

A necessidade de prover acesso padronizado às informações em documentos XML levou a W3C a desenvolver o DOM (*Document Object Model*). De acordo com a definição deste consórcio, DOM é uma API (*Application Programming Interface*) para documentos XML e HTML, que define a estrutura lógica de documentos e a forma como esses documentos são acessados e manipulados [Byrne 1998].

O objetivo principal de DOM é fornecer uma interface de programação padrão que possa ser usada em uma grande variedade de ambientes e aplicações [Byrne 1998]. A especificação DOM oferece apenas as definições de interfaces para as bibliotecas de DOM, e estas interfaces são independentes de plataforma e de linguagem [Byrne 1998].

É possível encontrar implementações DOM disponíveies em várias linguagens de programação orientadas a objetos tais como Java [Harold 2002], JavaScript [Holzner 2001], C++ [XercesC 2002], Python, e Perl [XercesP 2002].

3.2 As Especificações DOM do W3C

O W3C mantém atualmente três especificações para DOM. São elas: DOM 1, DOM 2 e DOM 3. Entretanto, a primeira versão de DOM, conhecida como DOM Nível 0, não foi formalmente especificada. Ele definia apenas o modelo de objetos que era usado no Netscape Navigator 3 e no Internet Explorer 3 para documentos HTML [Byrne 1998].

Em outubro de 1998 o W3C estabeleceu a primeira recomendação DOM, o DOM Nível 1. Essa recomendação consiste em dois módulos, DOM Core e DOM HTML. Iremos apresentar neste capítulo o módulo DOM Core. Ele define o conjunto de funcionalidades

básicas para documentos XML que devem ser implementadas por todas as aplicações que desejam estar em conformidade com DOM [Byrne 1998].

O nível 2 de DOM acrescenta algumas funcionalidades ao DOM 1. Ele inclui um modelo de objeto de folha de estilo, funcionalidade para manipular as informações de estilo anexadas a um documento, permite atravessar um documento, possui um modelo de evento interno e aceita *namespace* XML [Hors 2000a].

O DOM 3 encontra-se no estágio de planejamento e focalizará o carregamento e o salvamento de documentos, além dos modelos de conteúdo (como DTDs e esquemas), como suporte para validação de documentos. Ele também focalizará visões de documentos e formatação, eventos chave e grupos de eventos [Hors 2002].

As especificações 2 e 3 de DOM serão descritas no próximo capítulo com mais detalhes.

3.3 O que é DOM?

DOM é uma estrutura de dados abstrata que representa documentos XML como uma árvore de nós [Harold 2001]. Com DOM é possível descrever um documento XML para um outro aplicativo ou linguagem de programação em uma tentativa de manipular cada vez mais as informações da maneira desejada [Marchal 2000].

Um *parser* compatível com DOM lê todo o documento e constrói uma árvore de objetos na memória. A partir desta árvore é possível navegar por sua estrutura, adicionar, modificar, e remover elementos do documento.

3.3.1 A Árvore DOM

As informações contidas em um documento XML são organizadas em uma estrutura de árvore hierárquica, que tem um único nó raiz, e todos os nós nesta árvore, à exceção da raiz, têm um único nó pai. Além disso, cada nó tem uma lista de nós filhos, esta lista pode ser vazia, no caso de nó folha.

Ao analisar um documento, o *parser* DOM o quebra em itens individuais, que se tornam nós na árvore de objetos. Um documento XML é uma árvore composta de nós de vários tipos. Os nós são os objetos base desta árvore e podem ser classificados em doze tipos especializados: atributo, comentário, documento, elemento, entidade, fragmento de

documento, instrução de processamento, notação, referência da entidade, seção CDATA, texto, tipo de documento.

Listagem 3.1 – Documento XML que guarda nome e telefone de pessoas (agenda.xml)

```
<?xml version='1.0'?>
<AGENDA>
       <PESSOA>
                     <NOME> Carla dos Santos </NOME>
                     <FONE> 3453 - 1010 </FONE>
       </PESSOA>
       <PESSOA>
                     <NOME> Humberto Correia </NOME>
                     <FONE> 3253 - 2222 </FONE>
       </PESSOA>
</AGENDA>
```

O arquivo representado pela Listagem 3.1, `agenda.xml`, guarda informações sobre o nome e o número de telefone de pessoas. Um *parser* DOM, ao ler este documento, cria a árvore de objetos mostrada na Figura 3.1.

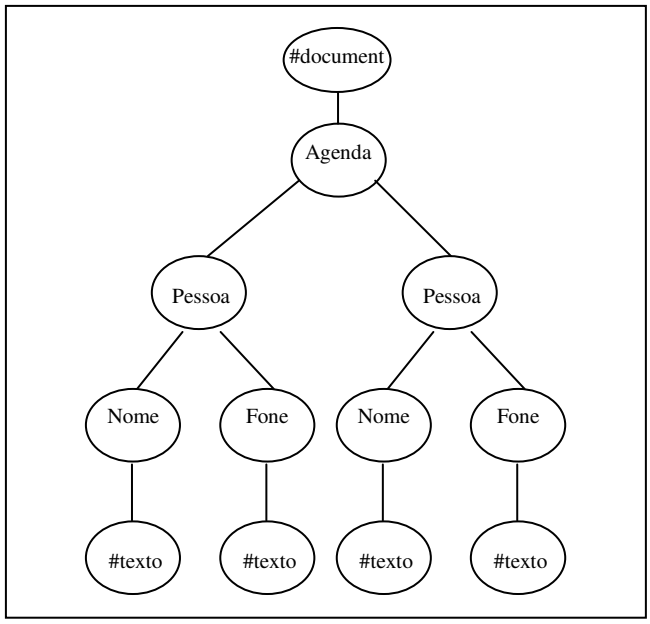


Figura 3.1 - Árvore DOM para a Listagem 3.1

O *parser* cria o objeto raiz da árvore, nó `document`. Os itens de um documento XML se tornam nós que são inseridos na árvore respeitando esta estrutura de dados. Por exemplo, para a Listagem 3.1 o *parser* cria um objeto para cada elemento do documento

`agenda.xml`. O conteúdo de um elemento no documento XML se torna filho desse elemento na árvore DOM.

A árvore de nós criada por DOM é uma representação lógica do conteúdo encontrado no arquivo XML: ela mostra as informações presentes no documento e como elas estão relacionadas [Marchal 2000].

3.4 As Interfaces DOM

DOM é composto de interfaces. Diferentes interfaces DOM representam diferentes itens do documento XML. Elementos, atributos, dados de caracteres, comentários, e instruções de processamento, por exemplo, são representados respectivamente pelas interfaces `Element`, `Attr`, `Text`, `Comment` e `ProcessingInstruction`. Todas estas interfaces são subinterfaces da interface `Node`, que provê métodos básicos para navegar e manipular a árvore.

DOM não define uma interface para representar o *parser*, por isto cada desenvolvedor define sua própria classe com nome e métodos que podem variar de um parser para o outro. Nos exemplos mostrados neste capítulo é usado o *parser* Xerces da Apache [Xerces 2002] que implementa o *parser* DOM na classe `DOMParser`. Esta classe é descrita no Capítulo 7.

3.4.1 A Interface Node

Node é a interface base para os diversos itens de um documento XML. Todos os nós na árvore DOM são uma instância desta interface. Através dela é possível adicionar, mover, remover e copiar nós na árvore. Além de ler nomes e valores dos nós.

Listagem 3.2 – Constantes da interface Node

```
public static final short ELEMENT_NODE                = 1;
public static final short ATTRIBUTE_NODE              = 2;
public static final short TEXT_NODE                   = 3;
public static final short CDATA_SECTION_NODE          = 4;
public static final short ENTITY_REFERENCE_NODE       = 5;
public static final short ENTITY_NODE                 = 6;
public static final short PROCESSING_INSTRUCTION_NODE = 7;
public static final short COMMENT_NODE                = 8;
public static final short DOCUMENT_NODE               = 9;
public static final short DOCUMENT_TYPE_NODE          = 10;
public static final short DOCUMENT_FRAGMENT_NODE      = 11;
public static final short NOTATION_NODE               = 12;
```

Cada nó da árvore possui um tipo específico. Para representar cada tipo, `Node` define um valor `short` que varia entre 1 e 12, como mostra a Listagem 3.2.

Características como nome, valor, tipo, *namespace* URI, prefixo e nome local de um nó podem ser obtidas e alteradas com os métodos mostrados na Listagem 3.3.

Listagem 3.3 – Métodos de Node para obter características dos nós

```
  // Node properties
  public String   getNodeName();
  public String   getNodeValue() throws DOMException;
  public void     setNodeValue(String nodeValue)
   throws DOMException;
  public short    getNodeType();
  public String   getNamespaceURI();
  public String   getPrefix();
  public void     setPrefix(String prefix) throws DOMException;
  public String   getLocalName();
```

Listagem 3.4 – Exemplo usando o método getNodeType()

```
...
Node n;
...
int type = n.getNodeType();

switch (type) {
       case Node.DOCUMENT_NODE: {
              System.out.println("Documento: " + n.getNodeName());
              break;
       }
       case Node.ELEMENT_NODE: {
              System.out.println("Elemento: " + n.getNodeName());
              break;
       }
       case Node. ATTRIBUTE_NODE: {
               System.out.println("Atributo: " + n.getNodeName());
       }
       case Node.CDATA_SECTION_NODE: {
              System.out.println("CDATA: " + n.getNodeValue());
              break;
       }
       case Node.TEXT_NODE: {
              System.out.println("Texto: " + n.getNodeValue());
              break;
       }
       case Node.PROCESSING_INSTRUCTION_NODE: {
              System.out.println("Instrucao de Processamento: " +
                     n.getNodeValue());
              break;
       }
}
...
```

Embora estes métodos sejam definidos para todos os nós, nem todos retornam valor significativo, isto é, alguns nós não têm a característica representada pelo método. Por

exemplo, apenas elementos e atributos possuem namespace URI. Quando outro tipo de nó invoca o método `getNamespaceURI()` o valor *null* é retornado. Para nós que não possuem nomes, o método `getNodeName()` retorna `#<tipo do nó>` (por exemplo, o nó `Document` retorna `#document`).

A Listagem 3.4 mostra um trecho de programa que obtém o tipo do nó usando o método `getNodeType()` e imprime o nome ou o valor do nó. Estes dados são acessados através dos métodos `getNodeName()` e `getNodeValue()`.

Para navegar pela árvore, `Node` especifica os métodos da Listagem 3.5. O método `getParentNode()` retorna o pai do nó. Para obter a lista de filhos de um nó esta interface define o método `getChildNodes()`, o método `hasChildNodes()` indica se o nó tem ou não tem filhos. Para acessar o primeiro filho e o último filho da lista `Node` define, respectivamente, `getFirstChild()` e `getLastChild()`. Os métodos `getPreviousSibling()` e `getNextSibling()` retornam o irmão da direita e o irmão da esquerda do nó.

Listagem 3.5 – Métodos de Node para navegar pela árvore

```
// Navigation methods
public Node         getParentNode();
public boolean      hasChildNodes();
public NodeList     getChildNodes();
public Node         getFirstChild();
public Node         getLastChild();
public Node         getPreviousSibling();
public Node         getNextSibling();
public Document     getOwnerDocument();
public boolean      hasAttributes();
public NamedNodeMap getAttributes();
```

Para verificar a existência de atributos para o nó e obter a lista dos atributos, `Node` define `hasAttributes{}` e `getAttributes()`. Ainda é possível obter a raiz da árvore, chamada de nó documento, através do método `getOwnerDocument()`. A Listagem 3.6 mostra como acessar os filhos do nó raiz da árvore.

Uma árvore DOM também pode ser alterada. Um nó pode ser inserido, removido, substituído ou anexado. Os métodos responsáveis por estas funções são mostrados na Listagem 3.7. O uso de um destes métodos pode gerar um documento mal formado. Se isto acontecer uma exceção é gerada e a operação interrompida.

Listagem 3.6 – Exemplo de como acessar os filhos de um elemento

```
import org.w3c.dom.*;
import org.apache.xerces.parsers.DOMParser;

public class NomeElemento
{
       public static void main(String[] args)
       {
              String out =" ";
              If (args.length <=0) {
                    System.out.println ("Use: java NomeElemento URL");
                    return;
              }
              try {
                     DOMParser p = new DOMParser();
                     p.parse(args[0]);
                     Document doc = p.getDocument();

                     if (doc.hasChildNodes()) {
                           NodeList filhos = doc.getChildNodes();
                           for (int i=0; i < filhos.getLength(); i++){
                                 Node item = filhos.item(i);
                                 out += item.getNodeName()+ "\r";
                           }
                     }

                     System.out.println(out);

              }catch (Exception e){
                     e.printStackTrace(System.err);
              }
       }
}
```

Listagem 3.7 – Métodos de Node para alterar um documento XML

```
  // Manipulator methods
  public Node insertBefore(Node newChild, Node refChild)
   throws DOMException;
  public Node replaceChild(Node newChild,  Node oldChild)
   throws DOMException;
  public Node removeChild(Node oldChild) throws DOMException;
  public Node appendChild(Node newChild) throws DOMException;
```

Além destes, `Node` define três outros métodos ditos utilitários, que permitem clonar (isto é, fazer uma cópia exatamente igual do nó), normalizar e verificar se um nó suporta determinada característica. A Listagem 3.8 mostra esses métodos.

Listagem 3.8 – Métodos utilitários de Node

```
  // Utility methods
  public Node cloneNode(boolean deep);
  public void normalize();
  public boolean isSupported(String feature, String version);
```

3.4.2 A Interface Document

Toda árvore DOM deve ter um nó `Document`. Ele representa a raiz da árvore e tem no mínimo um nó filho que é o elemento raiz do documento. Este nó também pode ter como filho comentário e instrução de processamento. Considerando o seguinte documento:

```
<?xml version='1.0'?>
<!-- Arquivo cliente.xml -->
<?xml-stylesheet type="text/css" href="cliente.css"?>
<cliente>
      <nome>João da Silva</nome>
      <endereco>Rua A, 23</endereco>
</cliente>
```

O objeto `Document` para este arquivo possui três filhos: um nó comentário, um nó instrução de processamento e um nó elemento para o elemento raiz `cliente`.

Listagem 3.9 – Interface Document

```
package org.w3c.dom;
public interface Document extends Node {
       public Element createElement(String tagName)
         hrows DOMException;
       public Element createElementNS(String namespaceURI,
         String qualifiedName) throws DOMException;
       public Text createTextNode(String data);
       public Comment createComment(String data);
       public CDATASection createCDATASection(String data)
         throws DOMException;
       public ProcessingInstruction createProcessingInstruction(
         String target, String data) throws DOMException;
       public Attr createAttribute(String name) throws DOMException;
       public Attr createAttributeNS(String namespaceURI, String
         qualifiedName) throws DOMException;
       public DocumentFragment createDocumentFragment();
       public EntityReference createEntityReference(String name)
         throws DOMException;
       public DocumentType getDoctype();
       public DOMImplementation getImplementation();
       public Element getDocumentElement();
       public Node importNode(Node importedNode, boolean deep) throws
         DOMException;
       public NodeList getElementsByTagName(String tagname);
       public NodeList getElementsByTagNameNS(String namespaceURI, String
         localName);
       public Element getElementById(String elementId);
}
```

A declaração XML, bem como DTD e espaços em branco presentes no prólogo de um documento não fazem parte da árvore DOM. A declaração e os espaços em branco são removidos pelo *parser*. A DTD fica disponível como uma propriedade do nó documento.

Um nó documento é uma instância da interface `Document`. Esta interface possui os métodos mostrados na Listagem 3.9. O método `getDocumentElement()` retorna o elemento raiz do documento. Para acessar os outros filhos de `document`, usamos os métodos fornecidos pela interface `Node`. Para acessar o elemento raiz do documento acima fazemos:

```
DOMParser p = new DOMParser();
p.parse ("cliente.xml");
Document doc = p.getDocument();
Node elemRaiz = doc.getDocumentElement();
```

Listagem 3.10 – Documento produto.xml

```
<produtos>
      <produto>
            <nome> Professional XML </nome>
            <preco moeda= "real"> 110,00</preco>
      </produto>
      <produto>
            <nome>XML by Example</nome>
            <preco moeda="real"> 104,00</preco>
      <produto>
</produto>
```

É possível também obter, através dos métodos `getElementsBy()`, todos os elementos de uma árvore com determinado nome. Por exemplo, suponha que uma aplicação precise encontrar o nome de todos os produtos da Listagem 3.10. Isto pode ser conseguido da seguinte forma:

```
NodeList listaNome = doc.getElementsByTagName("nome");
```

A variável `listaNome` conterá dois nós: `<nome> Professional XML </nome>` e `<nome>XML Java</nome>`.

A interface `Document` permite que instâncias de outros nós sejam criados para o documento. Ela define métodos `create` que possibilitam que nós elemento, texto, comentário, seção CDATA, instrução de processamento, atributo, fragmento de documento e referência de entidade sejam criados.

Elementos e atributos podem ser criados com ou sem *namespace*. Para criar um novo item elemento ou atributo apenas com nome local é usado a função `createElement()` ou `createAttribute()` que recebe como argumento o nome do item. Para criar item com *namespace* é preciso fornecer aos métodos `createElementNS()` e

`createAttributeNS()` *strings* que representam o *namespace* URI e o nome completo (<prefixo>:<nome local>).

Para criar comentários, seção CDATA e nó texto é necessário fornecer aos métodos `createComment()`,`createCDATASection()`e `createTextNode()` uma string com o dado que deve conter no nó.

O método `createEntityReference()` cria uma referência de entidade com o nome indicado pelo argumento passado para a função. Uma instrução de processamento é criada através do método `createProcessingInstruction()` que possui dois argumentos: o alvo e o dado da instrução.

O código abaixo ilustra como criar um elemento de nome: `produto`, e um nó texto contendo a string: `XML Java`:

```
Element produto = doc.createElement ("produto");
Text textoProduto = doc.createTextNode ("XML e Java");
```

Quando um novo nó é criado, apesar de estar associado ao objeto `document`, ele não pertence à árvore. Para fazer parte da árvore é preciso inseri-lo usando o método `insertBefore()` para adicionar um filho a `document`, ou `appendChild()` para inserir nós dentro do elemento raiz, ambos os métodos pertencem a `Node`.

O código abaixo mostra como inserir um novo filho (`produto`) no elemento raiz do documento, este novo elemento também recebe como filho o nó texto criado acima:

```
Node ElemRaiz = doc.getDocumentElement();
ElemRaiz.appendChild(produto);
produto.appendChild(textoProduto);
```

A interface `Document` ainda declara os métodos `getImplementation()` para retornar a implementação DOM que está sendo utilizada, e `importNode()` para copiar nó de outro documento.

3.4.3 A Interface Element

A interface `Element` representa os elementos de um documento XML. Um nó elemento pode ter um nome, um nome local, um *namespace* URI e um prefixo. O conteúdo é filho do elemento. Considere o seguinte elemento:

```
<db:produto xmlns:db="http://www.exemplo.com/"
  xmlns="http://namespaces.exemplo.com/">
      <nome>XML by Example</nome>
```

```
</db:produto>
```

O elemento `produto` tem o nome `db:produto`, o seu nome local é `produto`, o prefixo é `db`, e o *namespace* URI é `http://namespaces.exemplo.com/`. Ele ainda tem três filhos: um nó texto contendo espaço em branco, um nó elemento `nome` e outro nó texto com espaço em branco.

Listagem 3.11 – Interface Element

```
package org.w3c.dom;
public interface Element extends Node {
  public String  getTagName();
  public boolean hasAttribute(String name);
  public boolean hasAttributeNS(String namespaceURI,
   String localName);
  public String  getAttribute(String name);
  public void    setAttribute(String name, String value)
   throws DOMException;
  public void    removeAttribute(String name)
   throws DOMException;
  public Attr    getAttributeNode(String name);
  public Attr    setAttributeNode(Attr newAttr)
   throws DOMException;
  public Attr    removeAttributeNode(Attr oldAttr)
   throws DOMException;
  public String  getAttributeNS(String namespaceURI,
   String localName);
  public void    setAttributeNS(String namespaceURI,
   String qualifiedName, String value) throws DOMException;
  public void    removeAttributeNS(String namespaceURI,
   String localName) throws DOMException;
  public Attr    getAttributeNodeNS(String namespaceURI,
   String localName);
  public Attr    setAttributeNodeNS(Attr newAttr)
   throws DOMException;
  public NodeList getElementsByTagName(String name);
  public NodeList getElementsByTagNameNS(String namespaceURI,
   String localName);
}
```

Os métodos definidos na interface `Element` são mostrados na Listagem 3.11. A maioria dos métodos de `Element` manipula atributos. Esta interface declara funções que possibilitam verificar a existência de atributos, obter os atributos de um elemento, adicionar e remover atributos. Os outros métodos permitem obter o nome da *tag* do elemento, `getTagName()`, e acessar os filhos de um elemento com determinado nome, `getElementsByTagName()` e `getElementsByTagNameNS()`.

3.4.4 A Interface Attr

Um objeto da interface `Attr` representa um atributo que pode estar explícito no documento ou ser um atributo definido como *default* (padrão) na DTD ou no XML S*chema* associado ao documento. Atributos não são considerados como filhos do elemento, eles são anexados ao elemento.

Um atributo possui um nome, um nome local, um prefixo, um valor e um *namespace* URI. Considerando o seguinte código:

```
<db:produto xmlns:db="http://www.exemplo.com/"
  xmlns="http://namespaces.exemplo.com/">
```

o nome do atributo é `xmlns:db`, o seu nome local é `db`, o seu prefixo é `xmlns`, o valor do atributo é `http://www.exemplo.com/`, e o seu *namespace* URI é `http://namespaces.exemplo.com/`. Para obter esses dados, a interface `Attr` define os métodos mostrados na Listagem 3.12.

Listagem 3.12 – Interface Attr

```
package org.w3c.dom;
public interface Attr extends Node {
  public String  getName();  // Prefixed name
  public String  getValue();
  public Element getOwnerElement();
  public boolean getSpecified();
  public void    setValue(String value) throws DOMException;
}
```

Os métodos definidos em `Attr` permitem: obter o nome completo de um atributo usando `getName()`; determinar se o atributo é especificado no documento com `getSpecified()`; obter o elemento ao qual o atributo pertence através de `getOwnerElement()`; obter e alterar o valor do atributo usando, respectivamente, `getValue()` e `setValue()`.

Além destes métodos, é possível manipular atributos usando funções definidas na interface `Element`. Esta interface define métodos como `hasAttribute()` que verifica se um elemento tem atributos, `getAttribute()` que retorna o valor do atributo indicado, `setAttribute()` que atribui um novo valor a um atributo, `removeAttribute()` que remove o atributo de um elemento, entre outros. A Listagem 3.13 verifica se um elemento tem atributo e então imprime o nome e valor de cada atributo.

Listagem 3.13 – Exemplo de uso dos métodos de Attr

```
...
public static void mostrarAtr(Node node)
{
       if (node.getNodeType() == 1) //eh um elemento
       {
              if (node.hasAttributes()){
                     Attr atr;
                     Int numAtr = node.getAttributes().getLength();
                     for (int i=0; i < numAtr; i++){
                            atr = (Attr)node.getAttributes().item(i);
                            System.out.printl(atr.getName() + "=\""
                                   + atr.getValue());
                     }
              }
       }
...
```

3.4.5 A Interface NodeList

A interface `NodeList` representa uma lista indexada de nós, cada nó da lista é associado a um índice que varia de zero até o tamanho da lista menos um (índice –1). Uma instância desta interface pode ser obtida através do método `getChildNodes()` que retorna a lista dos filhos de um nó. Para obter os itens e o tamanho de uma lista usamos os métodos `item()` e `getLength()`, respectivamente.

Listagem 3.14 – Interface NodeList

```
package org.w3c.dom;
public interface NodeList {
        public Node item(int index);
        public int  getLength();
}
```

O trecho de código abaixo ilustra como usar os métodos de `NodeList` para obter cada item de uma lista de filhos de um nó e acessar os nomes destes nós:

```
NodeList lista = node.getChildNodes();
for (int i=0; i < lista.getLength(); i++){
      Node filho = lista.item(i);
      System.out.println(filho.getNodeName());
...
}
```

As alterações realizadas em um objeto `NodeList` refletem na árvore DOM. Por exemplo, se um nó da lista for removido, o mesmo nó será removido da árvore.

3.4.6 Outras Interfaces DOM

Além das interfaces anteriores, DOM define:

3.4.6.1 A Interface NamedNodeMap

É uma lista de nós com um nome anexado a eles. Ele aceita as mesmas propriedades e métodos de `NodeList`, mas também possui métodos especiais para acessar os nós pelo nome como mostra a Listagem 3.16.

Listagem 3.15 – Interface NamedNodeMap

```
public interface NamedNodeMap {

  public Node item(int index);
  public int  getLength();

  // para trabalhar com itens particulares da lista
  public Node getNamedItem(String name);
  public Node setNamedItem(Node arg) throws DOMException;
  public Node removeNamedItem(String name)
   throws DOMException;
  public Node getNamedItemNS(String namespaceURI,
   String localName);
  public Node setNamedItemNS(Node arg) throws DOMException;
  public Node removeNamedItemNS(String namespaceURI,
   String localName) throws DOMException;
}
```

3.4.6.2 A Interface ProcessingInstruction

Um objeto desta interface representa uma instrução de processamento. Ele é usado para guardar informações específicas de processador dentro de um documento XML. A assinatura dessa interface é mostrada na Listagem 3.14.

Uma instrução de processamento é dividia em duas partes: o alvo (*targat*) da instrução e o dado. Considerando a instrução de processamento mostrada abaixo:

```
<?xml-stylesheet type="text/css" href="saudacao.xml"?>
```

`xml- stylesheet` é o valor do alvo, e `type="text/css" href="saudacao.xml"` é o dado.

Listagem 3.16 – Interface ProcessingInstruction

```
public interface ProcessingInstruction extends Node {
       public String getTarget();
       public String getData();
       public void   setData(String data) throws DOMException;
}
```

A interface `ProcessingInstruction` herda atributos e métodos de `Node`, e não tem filhos na árvore DOM.

3.4.6.3 A Interface CharacterData

É a interface que representa os dados de caracteres em DOM. Ela é base para as interfaces: `Text`, `CDATASection` e `Comment`. Ela herda atributos e métodos de `Node` e também define novos métodos como mostra a Listagem 3.17.

Listagem 3.17 – Interface CharacterData

```
public interface CharacterData extends Node {
  public String getData() throws DOMException;
  public void   setData(String data) throws DOMException;
  public int    getLength();
  public String substringData(int offset, int length)
   throws DOMException;
  public void   appendData(String data) throws DOMException;
  public void   insertData(int offset, String data)
   throws DOMException;
  public void   deleteData(int offset, int length)
   throws DOMException;
  public void   replaceData(int offset, int length, String data)
   throws DOMException;
}
```

3.4.6.4 A Interface Comment

É a interface descendente de `CharacterData` que representa o conteúdo de um comentário. Considerando o comentário abaixo:

```
<!-- Início do documento -->
```

o valor do nó `Comment` é a string `Início do documento`.

Um comentário é um nó folha na árvore DOM, o que significa que ele não pode ter filhos.

3.4.6.5 A Interface Text

A interface `Text` representa o conteúdo de texto de um nó elemento, atributo, ou referência de entidade. Ele é descendente do objeto `CharacterData` e também define o método `splitText (int offset)` que divide o nó em dois no deslocamento (offset) indicado.

3.4.6.6 A Interface CDATASection

Essa interface representa uma seção CDATA em um documento XML. Ele é descendente da interface `Text` e não acrescenta atributos nem métodos extras.

3.4.6.7 A Interface DocumentType

A interface `DocumentType` representa uma declaração de tipo de documento (DTD) ou um XML *Schema*. Ela possui os métodos mostrados na Listagem 3.18.

Uma DTD pode ter quatro partes: o nome do elemento raiz, o public ID, o system ID e a parte interna da DTD que fica entre `[` e `]`. Por exemplo, para a DTD abaixo:

```
<!DOCTYPE mml:math PUBLIC "-//W3C//DTD MathML 2.0//EN"
 "http://www.w3.org/TR/MathML2/dtd/mathml2.dtd" [
  <!ENTITY % MATHML.prefixed "INCLUDE">
  <!ENTITY % MATHML.prefix "mml">
]>
```

- elemento raiz é: `mml:math`
- public ID: `-//W3C//DTD MathML 2.0//EN`
- system ID: `http://www.w3.org/TR/MathML2/dtd/mathml2.dtd`
- subconjunto interno:

```
<!ENTITY % MATHML.prefixed "INCLUDE">
<!ENTITY % MATHML.prefix "mml">
```

Listagem 3.18 – Interface DocumentType

```
public interface DocumentType extends Node {
  public String       getName();
  public NamedNodeMap getEntities();
  public NamedNodeMap getNotations();
  public String       getPublicId();
  public String       getSystemId();
  public String       getInternalSubset();
}
```

3.4.6.8 A Interface Entity

Representa uma entidade declarada com um elemento `<!ENTITY...>` na DTD. Ele é descendente de `Node`. O mapa das entidades declaradas em um documento é obtido pelo método `getEntities()` da interface `DocumentType`.

3.4.6.9 A Interface Notation

É um objeto descendente de `Node` que representa uma notação declarada em uma DTD ou XML *Schema* com o elemento `<NOTATION...>`. Como `Entity`, um objeto `Notation` não faz parte da árvore DOM. Diferente da interface `Entity`, a interface `Notation` não tem filhos.

3.4.6.10 A Interface EntityReference

Esta interface representa um nó referência de entidade dentro do documento XML. Os métodos e os atributos desse objeto são herdados de `Node`.

Uma referência de entidade pode fazer parte ou não da árvore DOM. Isso depende do *parser* utilizado. Caso o *parser* não valide o documento, a referência de entidade não será substituída pelo conteúdo referido, logo a árvore terá um nó `EntityReference`. Se o *parser* é de validação, ele pode escolher entre substituir o conteúdo referenciado ou manter a referência.

Para acessar o nome de uma referência de entidade é usado o método `getNodeName()` de `Node`. O seu conteúdo texto pode ser acessado por métodos como `getFirstChild()`. Entretanto, não é possível alterar os filhos de um objeto `EntityReference` ou o seu nome usando os métodos `appendChild()`, `replaceChild()` ou `setNodeName()`.

3.4.6.11 A Interface DocumentFragment

Esta interface representa fragmentos de documento que podem ser criados para auxiliar nas operações de árvore DOM. Por exemplo, um novo fragmento de documento pode ser criado e elementos inseridos nele, depois todo o fragmento pode ser inserido em um documento existente. Os atributos e métodos desse objeto são herdados do objeto `Node`.

3.4.6.12 A Interface DOMImplementation

Esta interface é um *abstract factory* responsável por criar novos objetos `Document` e `DocumentType`. Além dos métodos `create`, ela define o método `hasFeature()` que é

utilizado para verificar se a implementação de DOM suporta determinada característica da versão de DOM indicada.

Listagem 3.19 – Interface DOMImplementation

```
public interface DOMImplementation {
  public DocumentType createDocumentType(String rootElementQualifiedName,
   String publicID, String systemID) throws DOMException;
  public Document createDocument(String rootElementNamespaceURI,
   String rootElementQualifiedName, DocumentType doctype)
   throws DOMException;
  public boolean hasFeature(String feature, String version);
}
```

3.5 Considerações Finais

Este capítulo apresentou uma descrição geral do DOM, incluído porque ele foi criado, qual a sua aplicação e quais são as linguagens e *parsers* que implementam este padrão. Foi mostrado que este padrão possui três versões, também chamados de níveis 1, 2 e 3. Foi apresentado ainda, como é a estrutura de dados do DOM para processamento dos dados XML. Por último foi feita uma descrição pormenorizada de todas as interfaces da API.

O próximo capítulo trata das diferenças das versões 2 e 3 em relação à especificação inicial.

Capítulo 4 - DOM 2 e DOM 3

Neste capítulo são apresentados estudos sobre os níveis 2 e 3 de DOM. Nele são mostrados os módulos que compõem estas especificações, a descrição de suas características e as principais interfaces definidas em cada módulo.

4.1 DOM2

DOM2 (DOM nível 2) se tornou uma recomendação do W3C em novembro de 2000 [Hors 2000a]. A principal mudança introduzida neste nível de DOM foi o suporte a *namespace*. Uma *tag* com *namespace* tem um prefixo e um URI *namespace* associados a ela (veja seção 2.4). Para dar suporte ao *namespace* XML, o nível 2 de DOM acrescentou atributos e métodos às interfaces de DOM Core nível 1 para criar e manipular elementos e atributos associados a um *namespace* [Hors 2000a].

Além do suporte a *namespace*, DOM2 inclui módulos que dão suporte a visões, folhas de estilo, eventos, atravessamento e escala de documento [Hors 2000a, Means 2001b]. DOM2 está dividido em quatorze módulos, são eles: Core, XML, HTML, Views, StyleSheets, CSS, CSS2, Events, UIEvents, MouseEvents, MutationEvents, HTMLEvents, Traversal, Range, como mostra a Figura 4.1. DOM1 corresponde aproximadamente aos módulos Core e XML.

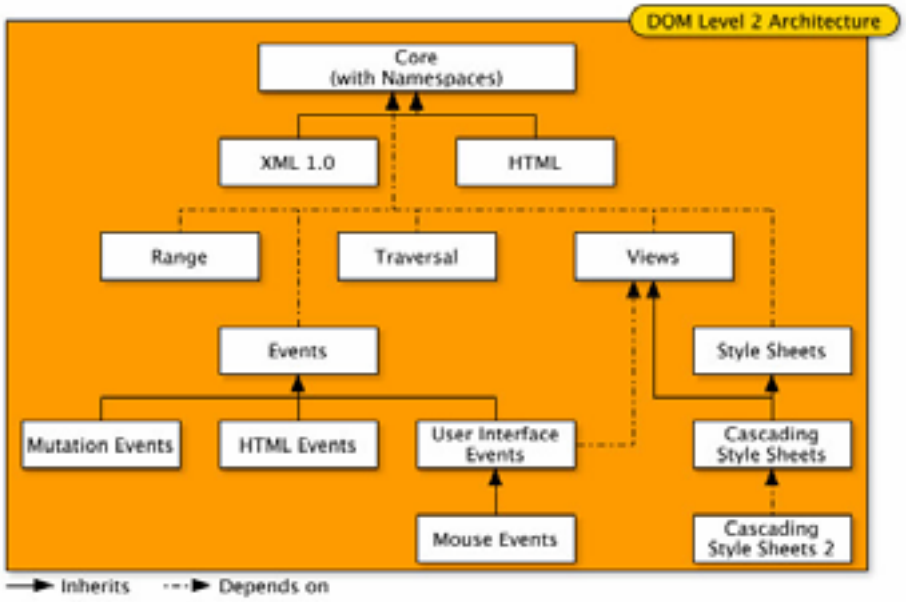


Figura 4.1 - Arquitetura de DOM2 [Hégaret 2002]

Para verificar se um *parser* DOM implementa determinado módulo desta especificação, a interface `DOMImplementation` define o método `hasFeature(recurso,`

`versão)`. Este método retorna `true` se a implementação suporta o módulo, e `false`, caso contrário.

4.1.1 O Módulo Core

O módulo Core de DOM nível 2 suporta todas as características do DOM1 Core e acrescenta suporte a *namespace*. Em conseqüência disto, as interfaces Attr, Document, NamedNodeMap, Node, DocumentType, DOMImplementation e Element receberam atributos e/ou métodos novos. A maioria dos novos métodos definidos em DOM2 faz referência a *namespace* e tem parâmetros adicionais para representá-lo (*namespace*URI, prefix, localName),.

Além do suporte *namespace*, outras características foram acrescentadas ao DOM2 Core. Por exemplo:

- a interface `Attr` ganhou o atributo `ownerElement` que indica a qual elemento o atributo pertence;
- a interface `Document` define o novo método `importNode`, que importa um nó de outro documento;
- a interface `Node` ganhou os métodos `isSupported` e `hasAttributes`;
- a interface DOMImplementation define métodos para criar objeto de documento XML e nó de tipo de documento (`createDocument` e `createDocumentType`);
- `DOMException` tem cinco novas exceções (`INVALID_STATE_ERR, SYNTAX_ERR, INVALID_MODIFICATION_ERR, NAMESPACE_ERR E INVALID_ACCESS_ERR`).

4.1.2 O Módulo Traversal

O módulo *Traversal* permite atravessar a árvore de um documento filtrando itens especificados. Isto é possível graças às interfaces `NodeIterator`, `TreeWalker` e `NodeFilters` definidas neste módulo.

As interfaces `NodeIterator` e `TreeWalker` representam visões lógicas da árvore DOM que podem não ter todos os nós que estão na árvore do documento. Estas visões têm formas diferentes de representar os nós de uma árvore. A `NodeIterator` apresenta uma

visão plana da árvore como uma lista ordenada de nós. A `TreeWalker` mantém os relacionamentos hierárquicos de árvore [Kesselman 2000, Fesler 2001].

Para determinar os nós que devem ou não fazer parte de uma visão, é possível usar um *flag* `whatToShow` ou um filtro da interface `NodeFilter`. Quando um objeto `NodeIterator` ou `TreeWalker` é criado, pode ser associado a ele um filtro que examina cada nó e determina se ele deve aparecer na visão lógica. O *flag* pode ser usado para especificar que tipo de nó deve ocorrer na visão.

`NodeIterators` e `TreeWalkers` ajustam-se automaticamente às mudanças na árvore subjacente.

4.1.2.1 A Interface NodeIterator

A interface `NodeIterator` extrai um sub conjunto dos nós de um documento e o apresenta como uma lista na ordem que aparece no documento: primeiro o nó do documento, nó pai antes dos filhos e nós irmãos na mesma ordem que suas *tags* de abertura aparecem no documento [Harold 2002]. Por exemplo, considere o documento abaixo:

```
<A>
  <B>text1</B>
  <C>
    <D>child of C</D>
    <E>another child of C</E>
  </C>
  <F>moreText</F>
</A>
```

Uma representação plana do `NodeIterator` para este documento XML é:

```
A  B  C  D  E  F
```

Neste tipo de visão não há relacionamento de pai e filho. Por isto, para acessar os nós da lista a interface `NodeIterator` define métodos que permitem mover-se para frente e para trás na lista como mostra a Listagem 4.1.

Cada *iterator* pode ser pensado como tendo um cursor que é inicialmente posicionado antes do primeiro nó na lista. O método `nextNode()` retorna o nó imediatamente depois do cursor e avança o cursor uma posição. O método `previousNode()` retorna o nó imediatamente antes do cursor e ajuda o cursor para uma

posição anterior. Se o *iterator* está posicionado no fim da lista, `nextNode()` retorna nulo. Se o *iterator* está posicionado no começo da lista, `previousNode()` retorna nulo [Kesselman 2000].

Listagem 4.1 – Interface NodeIterator

```
package org.w3c.dom.traversal;
public interface NodeIterator {
       public Node       getRoot();
       public int        getWhatToShow();
       public NodeFilter getFilter();
       public boolean    getExpandEntityReferences();
       public Node       nextNode() throws DOMException;
       public Node       previousNode() throws DOMException;
       public void       detach();
}
```

Para criar um `NodeIterator` é necessário definir um objeto da interface `DocumentTraversal` e invocar o método `createNodeIterator()` deste objeto. Quando um `NodeIterator` é criado, *flags* podem ser utilizados para determinar que tipos de nó serão "visíveis" e que nós serão "invisíveis" enquanto se atravessa a árvore.

O método `createNodeIterator()` recebe como parâmetros quatro argumentos:

- root – o nó por onde o *iterator* começa a atravessar. Apenas este nó e os seus descendentes são atravessados pelo *iterator*.
- whatToShow – uma constante inteira que especifica que tipo de nó o *iterator* irá incluir. As constantes são:

NodeFilter.SHOW_ELEMENT	0x00000001
NodeFilter.SHOW_ATTRIBUTE	0x00000002
NodeFilter.SHOW_TEXT	0x00000004
NodeFilter.SHOW_CDATA_SECTION	0x00000008
NodeFilter.SHOW_ENTITY_REFERENCE	0x00000010
NodeFilter.SHOW_ENTITY	0x00000020
NodeFilter.SHOW_PROCESSING_INSTRUCTION	0x00000040
NodeFilter.SHOW_DOCUMENT	0x00000080
NodeFilter.SHOW_DOCUMENT_TYPE	0x00000100
NodeFilter.SHOW_DOCUMENT_FRAGMENT	0x00000200
NodeFilter.SHOW_NOTATION	0x00000400
NodeFilter.SHOW_ALL	0xFFFFFFFF

- filter – um objeto `NodeFilter` que seleciona os nós que farão parte do *iterator*. Apenas os nós que passarem pelo filtro ficarão de fora. O valor nulo indica que nenhuma filtragem deve ser realizada.

- entityReferenceExpansion – um *boolean* que determina se os nós de referência de entidade serão expandidos ou não.

O código seguinte cria um *iterator* com todos os elementos de uma árvore e imprime o nome de cada elemento presente na lista:

```
NodeIterator iterator=
(DocumentTraversal)document).createNodeIterator( root,
                          NodeFilter.SHOW_ELEMENT, null, true);
Node n;
while ((n = iterator.nextNode()) != null)
          System.out.println (n.getNodeName());
```

Se ocorrer alguma mudança na árvore do documento, o *iterator* reflete tais mudanças. Por Exemplo, considere o *iterator* seguinte, a posição do *iterator* é indicada pelo asterisco (*)

```
A  B  C * D  E
```

Se o nó `D` for apagado da árvore e o método `nextNode()` for chamado, ele retornará o nó `E`. Se inserir o nó `X` entre `B` e `C`, e chamar `previousNode()`, o nó retornado será o `X`.

Deve-se observar que a posição corrente de um *iterator* é sempre entre dois nós ou antes do primeiro nó ou depois do último nó. Nunca a posição de um *iterator* será um nó.

4.1.2.2 A Interface NodeFilter

O argumento `whatToShow` permite selecionar apenas nós de tipos específicos em uma subárvore. Para realizar uma seleção mais rigorosa dos nós, DOM2 define a interface `NodeFilter`, que permite que o usuário crie objetos que "filtram" nós. Cada filtro contém a função `acceptNode()`, escrita pelo usuário, que olha um nó e determina se ele deve ou não fazer parte da visão lógica do *traversal* do documento [Kesselman 2000].

O método `acceptNode()` pode retornar uma das três constantes seguintes:

- NodeFilter.FILTER_ACCEPT - aceita o nó;
- NodeFilter.FILTER_REJECT - rejeita o nó e os seus filho;
- NodeFilter.FILTER_SKIP - rejeita o nó, mas permite que os seus filhos sejam verificados.

Para um *iterator*, que não tem a relação de nó pai e nó filho, só dois valores de constante fazem sentido: FILTER_ACCEPT e FILTER_REJECT (a constante FILTER_REJECT é semelhante a FILTER_SKIP).

4.1.2.3 A Interface TreeWalker

O objetivo de `TreeWalker` é o mesmo de `NodeIterator`, atravessar uma árvore DOM e filtrar nós. Entretanto, `TreeWalker`, é baseado em uma árvore com pai, filhos, e irmãos.

Objetos `TreeWalker` são criados da mesma forma que objetos `NodeIterator`, isto é, chamando o método `createTreeWalker()` da interface `DocumentTraversal` Este método recebe os mesmos argumentos que `createNodeIterator()`: o nó raiz da subárvore, uma constante inteira que especifica o tipo de nós a ser exibido, um objeto `NodeFilter` ou nulo, e um boolean que indica se as referências de entidade devem ou não ser expandidas [Kesselman 2000].

Considerando o documento XML seguinte, a estrutura de árvore que o representa é mostrada na Figura 4.2.

```
<A>
  <B>text1</B>
  <C>
    <D>child of C</D>
    <E>another child of C</E>
  </C>
  <F>moreText</F>
</A>
```

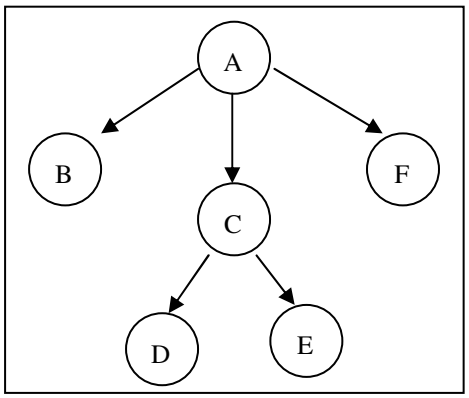


Figura 4.2 – Exemplo da árvore TreeWalker

Para acessar os nós da estrutura de árvore, a interface `TreeWalker` define os métodos listados na Listagem 4.2.

Em geral, `TreeWalkers` são melhores para tarefas em que a estrutura do documento em volta dos nós selecionados será manipulada, enquanto `NodeIterators` são melhores para tarefas que enfocam o conteúdo de cada nó selecionado [Kesselman 2000].

Listagem 4.2 – Interface TreeWalker

```
package org.w3c.dom.traversal;
public interface TreeWalker {
  public Node       getRoot();
  public int        getWhatToShow();
  public NodeFilter getFilter();
  public boolean    getExpandEntityReferences();
  public Node       getCurrentNode();
  public void       setCurrentNode(Node currentNode)
   throws DOMException;
  public Node       parentNode();
  public Node       firstChild();
  public Node       lastChild();
  public Node       previousSibling();
  public Node       nextSibling();
  public Node       previousNode();
  public Node       nextNode();
}
```

4.1.3 O Módulo Range (Intervalo)

Um Range representa uma seleção de uma parte de um documento ou fragmento de documento. Ele é formado pelo conteúdo da seleção delimitada por um par de pontos limites, o ponto inicial e o ponto final [Kesselman 2000]. Um objeto Range facilita operações tais como cortar, copiar, colar ou deletar conteúdo de uma árvore DOM [Kesselman 2000].

A posição de cada ponto limite em uma árvore de documento ou fragmento de documento é caracterizada por um nó e um deslocamento. O nó onde está o conteúdo do intervalo é chamado de *container* e o deslocamento dentro do nó é chamado de *offset*. Se o *container* é um nó atributo, documento, fragmento de documento, elemento ou referência de entidade, o *offset* está entre os nós filhos do container. Se o container é um dado de caracter, um comentário ou uma instrução de processamento, o *offset* está entre os caracteres deste *container* [Kesselman 2000].

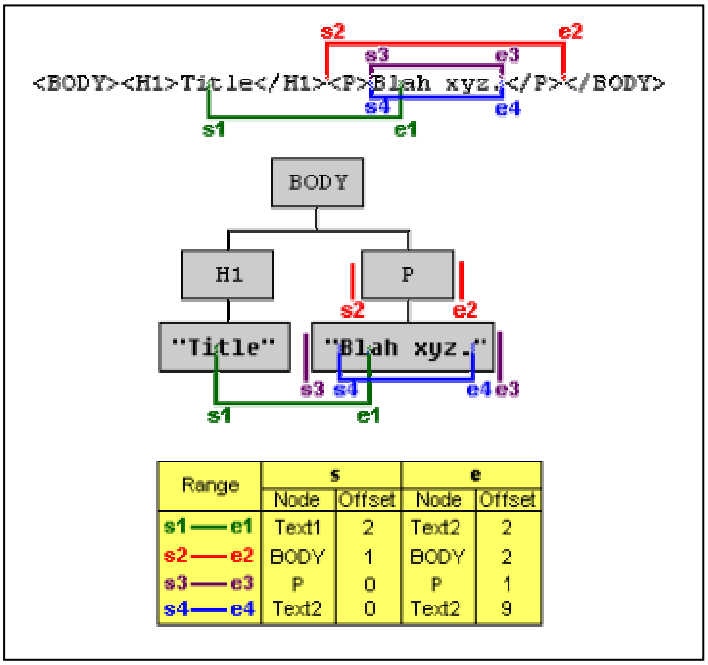
A interface Range especifica propriedades para definir os pontos limites de um intervalo, são elas:

- `startContainer`, que representa o container do ponto limite inicial;
- `startOffset`, representa o deslocamento do ponto inicial dentro do `startContainer`;

- `endContainer`, representa o container do ponto final do intervalo;
- `endOffset`, representa deslocamento dentro de `endContainer`.

Além dessas, são definidas também as propriedades `commonAncestorContainer` que referencia o primeiro ancestral que contém ambos os nós `startContainer` e `endContainer`; e `collapsed` que indica se os pontos limites do intervalo são os mesmos pontos no DOM. `Collapsed` é true se `startContainer` for o mesmo nó que `endContainer` e `startOffset` for igual a `endOffset`.

A Figura 4.3 mostra os pontos limites e o conteúdo de quatro intervalos, `s#` indica o ponto inicial da escala e `e#` o ponto final, onde `#` representa o índice do intervalo (1, 2, 3 ou 4). Por exemplo, Range 1 (R1) é delimitado pelo ponto inicial `s1` e pelo ponto final `e1`.



Range	s		e	
	Node	Offset	Node	Offset
s1—e1	Text1	2	Text2	2
s2—e2	BODY	1	BODY	2
s3—e3	P	0	P	1
s4—e4	Text2	0	Text2	9

Figura 4.3 – Exemplo Range [Kesselman 2000]

O conteúdo do Range 1 é: “tle</H1><P>Bl”. O *container* de `s1` é o nó texto (`Title`), o *offset* é determinado contando o deslocamento do ponto dentro do *container* a partir da posição zero, isto é, a posição antes do primeiro caracter, logo o *offset* de `s1` é 2. O ponto final `e1` também está em um nó texto e o seu deslocamento dentro desse nó é 2.

O Range 2 (R2) começa imediatamente depois do primeiro filho do elemento `BODY` (H1), assim o *container* de `s2` é `BODY` e o *offset* é o deslocamento entre os filhos desse

elemento (*offset* igual a 1). R2 termina imediatamente depois do elemento `P`, segundo filho de `BODY`, o que significa que o *container* de `e2` é `BODY` e o *offset* é 2.

A interface Range define um conjunto de métodos de alto nível que permite acessar e manipular a árvore de documento, além de métodos para obter e ajustar os pontos limites como mostra a Listagem 4.3.

Um objeto Range é criado chamando o método `createRange()` da interface `DocumentRange`. Esta interface pode ser obtida a partir do objeto da interface `Document` que implementa a especificação `DocumentRange`.

Listagem 4.3 – Interface Range

```
public interface Range {
    void setStart(Node refNode, long offset) throws RangeException,
      DOMException;
    void setEnd(Node refNode, long offset) throws RangeException, DOMException;
    void setStartBefore(Node refNode) throws RangeException, DOMException;
    void setStartAfter(Node refNode) throws RangeException, DOMException;
    void setEndBefore(Node refNode) throws RangeException, DOMException;
    void setEndAfter(Node refNode) throws RangeException, DOMException;
    void collapse(boolean toStart) throws DOMException);
    void selectNode(Node refNode) throws RangeException, DOMException;
    void selectNodeContents(Node refNode) throws RangeException, DOMException;
    short compareBoundaryPoints(unsigned short how, Range sourceRange) throws
      DOMException;
    void deleteContents()throws DOMException;
    DocumentFragment extractContents()throws DOMException;
    DocumentFragment cloneContents()throws DOMException;
    void insertNode( Node newNode) throws DOMException, RangeException;
    Void  surroundContents(Node newParent) throws DOMException,  RangeException;
    Range cloneRange()throws DOMException;
    DOMString toString()throws DOMException;
    void detach()throws DOMException;
}
```

4.1.4 O Módulo Events

O modelo *Events* de DOM 2 tem como principal objetivo a padronização de um sistema de evento genérico que permite registro de tratadores de eventos, descreve fluxo de eventos através de uma estrutura de árvore e fornece informações contextuais básicas para cada evento. Também é papel deste modelo fornecer módulos padrões de eventos para controle de interface de usuário e de notificação de mutação de documento, e definir informações contextuais para cada um desses módulos, além de fornecer um subconjunto do sistema de evento atual usado no DOM nível 0 [Pixley 2000].

Para cumprir seus objetivos, a especificação do modelo de eventos define o modelo de eventos DOM, que consiste da propagação do evento, registro de *listener* de eventos e a interface `Event`, e módulos de eventos designados para serem usados dentro do modelo [Pixley 2000].

Um evento pode ser gerado pela interação do usuário através de um instrumento externo como mouse ou teclado, como pode ser fruto de troca de mensagem ou notificação de elementos, ou também pode ser causado por alguma ação que modifica a estrutura do documento.

Todo evento segue um fluxo, ele se origina da implementação DOM e é passado para dentro do DOM. Este fluxo pode ser de dois tipos: *capture* ou *bubbling*. *Capture* é o processo no qual o evento é tratado por todos os ancestrais do alvo do evento antes de ser tratado pelo nó alvo do evento. Este processo acontece no sentido do topo da árvore para baixo. O processo de *bubbling* tem sentido oposto ao capture, ele opera do nó alvo para cima, nele o evento propaga através dos ancestrais do alvo do evento depois de ter sido tratado pelo nó alvo do evento.

Um evento precisa de um *listener* (ouvidor) de evento; para defini-lo é necessário que a implementação DOM implemente a interface `EventTarget`, que associa eventos a nós da árvore. Esta interface define métodos que registra e remove `EventListener`, respectivamente:

```
node.addEventListener (eventType, function, useCapture);
node.removeEventListener (eventType, function);
```

O argumento `eventType` é o tipo do evento que está sendo registrado, `function` é o nome da função de tratamento definida pelo usuário que deve ser chamada quando o evento ocorrer e `useCapture` é um valor booleano (se verdadeiro a fase da captura deve ser iniciada).

A função responsável por tratar um evento deve receber como argumento um objeto da interface `Event`, este objeto é usado para fornecer informações contextuais sobre o evento. Nesta interface são definidos métodos para inicializar o valor de um evento criado através da interface `DocumentEvent`, para cancelar eventos, caso ele tenha sido especificado como cancelável, e para bloquear propagações de eventos.

A especificação do modelo de eventos de DOM 2 também define um módulo de evento de interface de usuário (*UI Event*), que é composto de eventos listados em HTML 4.0 e adiciona eventos suportados nos browsers de DOM Level 0, um módulo de evento lógico de UI (*UI Logical Events*), e um módulo de evento de mutação de documento (*Mutation Event*), que é designado para permitir notificação de mudanças na estrutura de um documento, como modificação de atributos e texto.

4.1.5 O Módulo Style Sheets (folha de estilo)

O módulo *Style Sheets* da especificação de DOM2 define interfaces bases utilizadas para representar qualquer tipo de folha de estilo (CSS, CSS2, XSL). A expectativa é que os módulos de DOM, que representam uma linguagem de folha de estilo específica (CSS, CSS2, XSL), possam conter interfaces derivadas das interfaces do módulo Style Sheets [Wilson 2000].

As interfaces definidas neste módulo são:

- styleSheet – representa uma folha de estilo associada com um documento estruturado.
- StyleSheetList – provê a abstração de uma coleção ordenada de folhas de estilo.
- MediaList – provê a abstração de uma coleção ordenada de mídias, sem definir como esta coleção será implementada.
- LinkStyle – provê um mecanismo pelo qual uma folha de estilo pode ser encontrada (restaurada) do nó responsável por ligá-lo dentro do documento.
- DocumentStyle – prover mecanismo que permite uma folha de estilo embutida em um documento seja restaurada.

4.1.6 O Módulo CSS

O Cascading Style Sheets (CSS) possui uma sintaxe declarativa para definir regras de apresentação, propriedades e construções usadas para formatar documentos Web. Para acessar e manipular o estilo de apresentação CSS o nível 2 de DOM define o módulo CSS que possibilita a associação de folhas de estilo CSS com documentos XML e HTML.

Este módulo define interface para cada regra CSS2 (por exemplo, declarações de estilo, regras @import, ou regras @font-face) e a interface genérica `CSSRule`. A interface `CSSStyleSheet` permitem acessar a coleção de regras dentro de uma folha de estilo CSS e os seus métodos possibilitam a modificação das regras. Há também uma interface opcional, `CSS2Properties`, que fornece atalhos para os valores de strings de todas as propriedades CSS2.

Uma implementação DOM para suportar este módulo deve também implementar o módulo Core e o módulo Views de DOM2.

4.1.7 O Módulo Views

Um documento pode ter uma ou mais "visões" associadas a ele. Uma visão é uma representação ou uma apresentação alternativa associada com um documento fonte [Hors 2000b].

Uma visão pode ser estática, refletindo o estado do documento quando a visão foi criada, ou dinâmica, refletindo mudanças no documento alvo que ocorrem depois da visão criada [Hors 2000b].

Este módulo define a interface `AbstractView` que fornece uma interface base da qual todas as visões são derivadas. Ela define um atributo que referencia o documento alvo do `AbstractView`, este atributo cria uma associação entre uma visão e seu documento alvo.

4.2 DOM 3

O DOM 2 é usado em grande número de aplicações. Entretanto, esta versão de DOM tem algumas limitações: ela não fornece uma interface para definir o *parser* DOM; comparações entre nós são limitadas; e alguns itens de um documento XML, por exemplo a declaração XML, não são disponíveis [McLaughlin, 2001].

A especificação DOM Level 3 ainda não é uma recomendação do W3C, ela está em processo de planejamento e acrescentará alguns recursos à versão DOM 2. O nível 3 estenderá o nível 2 para apoiar XML Infoset e suportar XML Base [Hors 2002], além de estender os eventos de interface de usuário. Ele também somará apoio a esquemas abstratos (DTDs, XML Schema), a capacidade para carregar e salvar um documento ou um esquema abstrato, e suportará XPath.

DOM 3 é composto pelos módulos de DOM 2 e acrescenta os módulos: Abstract Schemas, Load and Save e Xpath, como mostra a Figura 4.4.

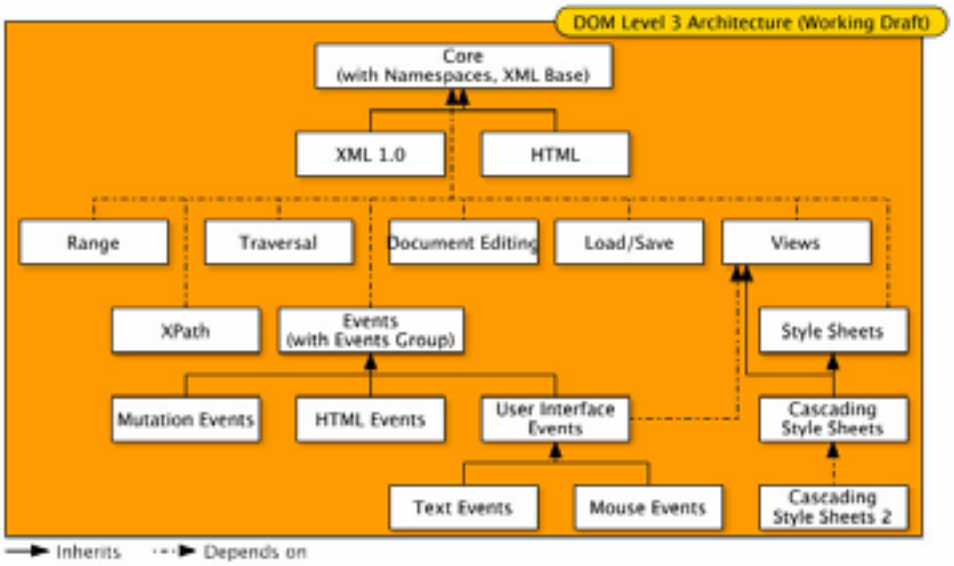


Figura 4.4 - Arquitetura de DOM3 [Hégaret 2002]

4.2.1 O Módulo Core

O módulo Core de DOM 3 acrescentará suporte à implementação de DOM e permitirá comparações estruturais entre nós. Para isso serão definidos métodos que verificam se os conteúdos texto dos nós são idênticos ou se dois nós têm os mesmos filhos, por exemplo. DOM 3 também definirá atributos e métodos para obter e alterar dados de declaração XML.

4.2.2 O Módulo Events

O modelo *Events* de DOM 3, além de adicionar novos atributos e métodos às interfaces definidas no modelo de DOM 2, também define as interfaces `EventGroup` e `CustomEvent`, e o novo módulo de evento de interface de usuário, `TextEvent`, que fornece informações contextuais sobre eventos de texto.

A interface `EventGroup` é uma interface de registro de eventos que funciona como um lugar de suporte para separar fluxo de evento quando há múltiplos grupos de *listeners* para uma árvore DOM.

A interface `CustomEvent` é usada pela implementação DOM para acessar o nó subjacente enquanto propaga o evento na árvore. Ela define os métodos

`setCurrentTarget`, que permite a implementação DOM alterar o atributo que especifica o alvo do evento, e `setEventPhase` que permite que o atributo `EventPhase` da interface `Event` seja alterado.

Nesta versão do modelo de evento também foram adicionados os seguintes métodos à interface `EventTarget`:

- addEventListenerNS
- canTrigger
- isRegisteredHereNS
- RemoveEventListenerNS

4.2.3 O Módulo Abstract Schemas

O módulo opcional *Abstract Schemas* (AS) de DOM 3 fornece uma representação para esquemas abstratos de XML (DTD, XML Schema), operações sobre estes esquemas, e disponibiliza o modo como as informações do esquema poderiam ser aplicadas nos documentos XML [Chang 2002].

Este módulo define três tipos de recursos: "AS-READ", que permite apenas leitura do esquema abstrato, "AS-EDIT", que permite a edição de esquemas abstratos, e "AS-DOC", que permite que documentos sejam editados.

De acordo com os recursos, as interfaces neste módulo podem ser divididas em: interfaces de esquema abstrato básicas, interface de esquema abstrato apenas de leitura, interface de edição de esquema abstrato e interface de edição de documento.

As interfaces básicas representam o conjunto de interfaces que são comuns para AS-READ e AS-EDIT, são elas: ASConstants, ASObject, ASDataType, ASObjectList e ASNamedObjectMap.

As interfaces que fornecem acesso apenas de leitura para esquemas abstratos são: ASModel, ASContentModel, ASElemntDecl, ASAttributeDecl, ASEntityDecl e ASNotationDecl.

As interfaces para edição de estrutura de dados de esquema abstratos e seus métodos são: ASWModel, ASWNamedObjectMap, ASWElementDecl, ASWContentModel, ASWAttributeDecl, ASWEntityDecl e ASWNotationDecl. Para

edição de documento são definidas as interfaces: DocumentEditAS, NodeEditAS, ElementEditAS e CharacterDataEditAS.

4.2.4 O Módulo Load and Save

O módulo *Load and Save* fornecerá uma API para carregar documentos XML em uma representação DOM e para salvar uma representação DOM como um documento XML [Chang 2002].

O nível 3 de DOM adicionará o pacote org.w3c.dom.ls para carregar e salvar documentos de modo que seja possível escrever programas DOM independentes de implementação. Este pacote conterá diversas classes para escrever documentos XML em arquivos, na rede, ou qualquer outro fluxo de saída (*OutputStream*). A interface `DOMBuilder` é responsável pela parte de carregamento de documento, enquanto que a parte de salvamento é baseada na interface `DOMWriter` [Harold 2002].

`DOMBuilder` é uma interface para um objeto que é capaz de construir uma árvore DOM de várias fontes de entrada. Esta interface fornece uma API para analisar documentos XML e construir as árvores DOM para os documentos correspondentes. Uma instância de `DOMBuilder` é obtida invocando o método `createDOMBuilder` da interface `DOMImplementationLS` [Chang 2002].

`DOMWriter` pode copiar um objeto nó da memória em bytes ou em caracteres serializados. Esta interface define métodos para escrever nós XML em um *OutputStream* Java ou uma string. O tipo mais comum de nó escrito é `Document`, mas é possível escrever todos os outros tipos de nó também como elemento, atributo, e texto. Esta interface tem também os métodos para controlar exatamente como a saída é formatada e como os erros são relatados. Os métodos dessa interface são mostrados na Listagem 4.4.

Para criar um `DOMWriter`, a interface `DOMImplementationLS`, que é derivada da interface `DOMImplementation`, define métodos para criar `DOMBuilders`, `DOMWriters`, e `DOMInputSources`.

Para controlar a saída de um escritor é possível instalar um filtro em um `DOMWriter`. Um filtro é um objeto da interface `DOMWriterFilter` que é uma sub interface de `NodeFilter`, e trabalha quase exatamente como ela.

Listagem 4.4 - Interface DOMWriter

```
package org.w3c.dom.ls;
public interface DOMWriter {
       public void    setFeature(String name, boolean state)
          throws DOMException;
       public boolean canSetFeature(String name, boolean state);
       public boolean getFeature(String name) throws DOMException;
       public String  getEncoding();
       public void    setEncoding(String encoding);
       public String  getLastEncoding();
       public String  getNewLine();
       public void    setNewLine(String newLine);
       public DOMErrorHandler getErrorHandler();
       public void setErrorHandler(DOMErrorHandler errorHandler);
       public boolean writeNode(OutputStream out, Node node)
         throws Exception;
       public String writeToString(Node node) throws DOMException;
}
```

4.3 Parsers DOM

Existem hoje vários *parsers* XML implementados em diversas linguagens de programação, sendo a maioria em Java. Mas nem todos os recursos de DOM 2 e DOM3 mostrados nas seções anteriores estão presentes nestas implementações. Por exemplo, os *parsers* Xerces2 [Xerces 2002], GNU [GNU 2001], Oracle [Oracle] e Crimson [Crimson 2001] suportam XML 1.0 e implementam a especificação DOM 1 da W3C, nem todos suportam os módulos definidos no nível 2 de DOM e apenas um deles implementa de forma incompleta módulos de DOM3.

Xerces2 Java *Parser*, na sua versão 2.0.2, além de dar suporte a XML 1.0, suporta os padrões *Namespace*, DOM level 2 (módulos Core, Events e Traversal), SAX2 (veja Capítulo 5) e XML Schema 1.0. Esta versão de Xerces também implementa de forma experimental os módulos Core, Abstract Schemas e Load and Save de DOM level 3. Entretanto, no que diz respeito ao modelo Abstract Schemas, Xerces não fornece implementação para editar esquemas abstratos (AS-EDIT), nem para editar documentos (AS-DOC) [Xerces 2002].

O Oracle XML *Parser* for Java v2 foi construído para suportar XML 1.0 e os seguintes recursos adicionais DOM 1, DOM 2 Core, DOM2 Traversal and Range e SAX 2.0 [Oracle 2002].

GNU é um parser baseado em Java que dá suporte ao nível 1 de DOM e implementa os modelos Core, Events (incluindo os módulos UI Events, Mutation Events e HTML Events), e parcialmente o módulo Traversal do DOM 2.

O *parser* Crimson versão 1.1.3 implementa apenas a especificação DOM level 1 e o módulo Core de DOM level 2.

A Tabela 4.1 mostra um resumo dos padrões suportados pelos *parsers* citados. Uma avaliação mais completa de alguns parsers XML é apresentada no Capítulo 7.

Tabela 4.1 – Tabela de *parsers* XML

	Xerces	**GNU**	**Oracle**	**Crimson**
DTDs	X	X	X	X
Schemas	X		X	X
Namespaces	X	X	X	X
Lazy DOM	X			
HTML DOM	X			
Views				
Style Sheets				
CSS				
CSS2				
Events	X	X	X	
UI Events		X		
Mouse Events				
Mutation Events	X	X		
HTML Events		X		
Traversal	X	partial	X	
Range			X	
XSLT/XPath	Via Xalan-J		X	
Xinclude		X		
Core DOM3	X			
Abstract Schemas	X			
Load and Save	X			
Events DOM3				

4.4 Considerações Finais

Este capítulo apresentou as versões 2 e 3 do DOM ressaltando quais os módulos acrescentados em cada versão. Foi mostrado que a versão 2 do DOM oferece suporte a *namespace*, a eventos e a folha de estilo e que a versão 3, além de outras características, dá suporte à persistência dos documentos XML.

Entretanto, nem todos os módulos definidos nesses níveis são atualmente suportados pelos *parsers* XML.

Capítulo 5 - SAX

Neste capítulo é mostrado um estudo sobre a interface de programa de aplicação baseada em eventos, SAX. São descritas suas características, funcionalidade, bem como as interfac es e métodos que compõem esta API.

5.1 Introdução

Simple API for XML ou SAX nasceu na lista de discussão XML-DEV [Megginson 2001]. Muitos desenvolvedores de aplicações XML sofriam com a incompatibilidade dos analisadores dessa linguagem. Com o objetivo de tornar compatíveis os diferentes analisadores XML, os membros desta lista começaram em dezembro de 1997 a discussão sobre a criação de um *parser*. Com a ajuda de vários colaboradores, David Megginson liberou em 11 de maio de 1998 a primeira versão de SAX [Megginson 2001].

SAX é uma API de domínio público, não pertence a qualquer consórcio ou agência de padrões, nem a qualquer empresa ou indivíduo [Megginson 2001]. Entretanto, é um padrão bastante usado.

Inicialmente, ela foi desenvolvida em Java. Hoje, é possível encontrar *parsers* SAX escrito em várias linguagens de programação, entre elas C++, Visual Basic, Python e, Perl [Harold 2002].

5.2 Interface Baseada em Eventos

SAX é uma interface orientada a eventos e, como tal, funciona de forma bem diferente de uma interface baseada em objetos. Um *parser* baseado em eventos diz à aplicação o que está no documento ao notificar a aplicação de um fluxo de eventos analisados [Martin 2000]. Isto é, o *parser* baseado em eventos lê um documento e diz à aplicação quais os símbolos que ele encontra, à medida que os símbolos são encontrados. Por exemplo, ele notifica a aplicação que encontrou uma *tag* inicial, um dado de caracter, uma *tag* de fim, e assim por diante.

Considere o seguinte arquivo XML:

```
<? xml version= "1.0" encoding="UTF-8"?>
<produtos>
      <produto tipo="livro">
            XML
      </produto>
</produtos>
```

Um *parser* SAX ao processar este documento dispara os seguintes eventos:

```
startDocument()
startElement ("produtos")
```

```
startElement ("produto")
characters ("XML")
endElement ("produto")
endElement ("produtos")
endDocument ()
```

Quando o *parser* XML processa este documento ele gera uma seqüência de eventos para tudo que ele reconhece no documento. Primeiro ele dispara o evento que indica o início do documento (`startDocument`). Ao ler a *tag* `<produtos>`, o parser gera um segundo evento, que notifica que o elemento `produtos` está começando (`startElement`). Depois ele encontra a *tag* de abertura `<produto>` e gera um novo evento para início de elemento. Em seguida ele encontra o conteúdo do elemento `<produto>` (`XML`) e dispara o evento correspondente a dados de caracteres (`characters`). Quando encontra a *tag* de fechamento `</produto>`, o *parser* dispara o evento de fim de elemento (`endElement`). Para a *tag* `</produtos>` ele também gera um evento que indica o fim deste elemento e depois gera o último evento de fim de documento (`endDocument`). A Figura 5.1 ilustra este processo.

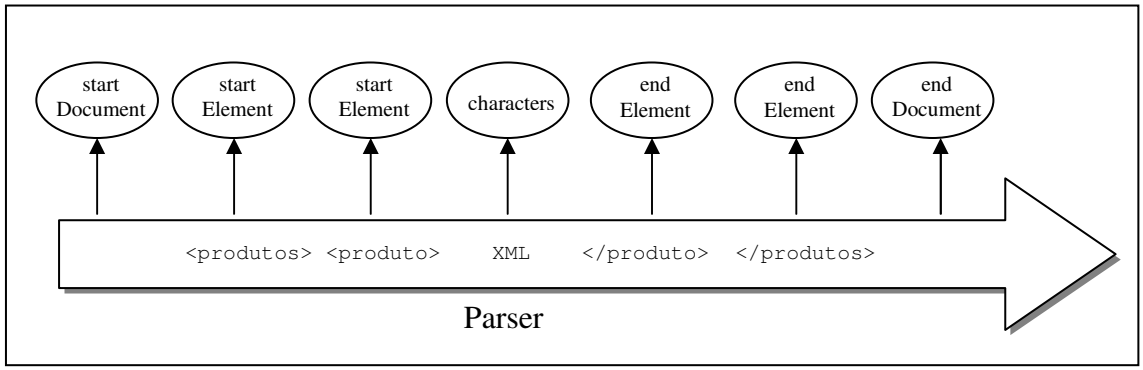


Figura 5.1 – Eventos gerados por um parser SAX a medida que ler o documento XML

5.3 SAX e SAX 2

Depois de sua criação em 1998, SAX sofreu algumas alterações. No final de 1999, uma reformulação de SAX começou a ser feita e, em maio de 2000, a versão 2.0 de SAX estava completa. Essa nova versão mantém a mesma arquitetura orientada a eventos, mas substitui várias classes de SAX1, além de criar novas classes. A principal mudança da versão 1.0

para a versão 2.0 foi tornar SAX2 sensível a namespace, além de acrescentar filtros e suporte a eventos léxicos e DTDs [Megginson 2001].

As interfaces `AttibuteList`, `AttributeListImpl`, `DocumentHandler`, `HandlerBase` e `Parser` definidas em SAX1 foram substituídas na versão 2, respectivamente, pelas interfaces: `Attibutes`, `AttributesImpl`, `ContentHandler`, `DefaultHandler` e `XMLReader`. Além destas, SAX2 define sete novas interfaces, são elas: `XMLFilter`, `NamespaceSupport`, `XMLFilterImpl`, `ParserAdapter`, `SAXNotSupportedException`, `SAXNotRecognizedException` e `XMLReaderAdapter`.

A versão mais atual de SAX é a 2.0.1 [Megginson 2001], que não traz mudanças radicais com relação a SAX 2.0, apenas acrescenta alguns bits de informação do documento XML que não são expostos por SAX2, tal como a declaração de codificação. Entretanto, nenhuma classe, interface, ou método de SAX2 são desprezados em SAX 2.0.1.

5.4 As Interfaces SAX

SAX foi projetado em torno de interfaces. As interfaces básicas de SAX2 são `XMLReader` e `ContentHandler`. A primeira representa o *parser*, e `ContentHandler` é a interface que trata eventos produzidos pelo *parser*. Através desta interface, o *parser* notifica a aplicação o que está lendo no documento XML.

Em SAX, o *parser* é uma instância da interface `XMLReader`, a classe específica que implementa esta interface varia para cada *parser*. Depois de criar o *parser*, os objetos `InputSource`, que contêm o documento XML, são passados para o método `parse()` de `XMLReader`. O *parser* lê o documento e, caso detecte algum erro de formação, ele gera uma exceção definida na interface `SAXException`.

À medida que o *parser* lê o documento ele invoca métodos da interface `ContentHandler` para os itens encontrados. Por exemplo, quando o *parser* lê uma *tag* de abertura, ele chama o método `startElement()`; quando lê um conteúdo de texto o método invocado é `characters()`. Se o *parser* encontra uma *tag* de fechamento, ele invoca o método `endElement()`, e assim por diante, até chegar ao fim do arquivo XML e invocar o método `endDocument()`.

5.5 A Interface XMLReader

SAX representa o *parser* como uma instância da interface `XMLReader`. Como interface, ela não possui construtor. Para criar uma instância de `XMLReader` é usado um dos métodos `createXMLReader()` da classe `XMLReaderFactory`:

```
public static XMLReader createXMLReader()
      throws SAXException;
public static XMLReader createXMLReader(String className)
      throws SAXException;
```

O primeiro método retorna a implementação *default* de `XMLReader`. Os desenvolvedores de *parser* modificam este método para retornar o seu próprio *parser*. No segundo método é possível especificar a classe do desenvolvedor que implementa `XMLReader`. Esta classe varia para cada desenvolvedor. Por exemplo, no parser Xerces e XML for Java a classe é `org.apache.xerces.parsers.SAXParser` [Xerces 2002], em Aelfred a classe é `gnu.xml.aelfred2.XmlReader` [GNU 2001], em Crimson a classe é `org.apache.crimson.parser.XMLReaderImpl` [Crimson 2001], enquanto que no parser da Oracle `XMLReader` é implementada pela classe `oracle.xml.parser.v2.SAXParser` [Oracle 2002].

Listagem 5.1 – Interface XMLReader

```
package org.xml.sax;
public interface XMLReader {
    public ContentHandler getContentHandler();
    public void setContentHandler(ContentHandler handler);
    public DTDHandler getDTDHandler();
    public void setDTDHandler(DTDHandler handler);
    public ErrorHandler getErrorHandler();
    public void setErrorHandler(ErrorHandler handler);
    public EntityResolver getEntityResolver();
    public void setEntityResolver(EntityResolver resolver);
    public boolean getFeature(String name) throws SAXNotRecognizedException,
       SAXNotSupportedException;
    public void setFeature(String name, boolean value) throws
       SAXNotRecognizedException, SAXNotSupportedException;
    public Object getProperty(String name) throws SAXNotRecognizedException,
       SAXNotSupportedException;
    public void setProperty(String name, Object value) throws
       SAXNotRecognizedException, SAXNotSupportedException;
    public void parse(String systemID) throws SAXException;
    public void parse(InputSource in) throws SAXException;
}
```

Para criar um objeto XMLReader utilizando o XML for Java, por exemplo, é possível declarar:

```
XMLReader parser = XMLReaderFactory.createXMLReader(
        "org.apache.xerces.parsers.SAXParser");
```

ou chamar o construtor da classe que implementa esta interface:

```
XMLReader parser = new SAXParser();
```

A interface `XMLReader` é definida no pacote `org.xml.sax` e possui os métodos mostrados na Listagem 5.1.

Os métodos definidos nesta interface permitem que uma aplicação atribua e questione características e propriedades no *parser*, registre manipuladores de evento para processar um documento, e inicie a análise de um documento [Megginson 1998].

Tabela 5.1 – Propriedade padrão

Propriedade	Descrição
http://xml.org/sax/properties/declaration-handler	Esta propriedade identifica o objeto da interface opcional `DeclHandler` do *parser*.
http://xml.org/sax/properties/dom-node	Guarda o objeto `Node` correspondente ao evento SAX atual.
http://xml.org/sax/properties/lexical-handler	Guarda um objeto da interface opcional `LexicalHandler` que representa os eventos léxicos do documento.
http://xml.org/sax/properties/xml-string	Guarda o texto que corresponde ao evento SAX atual

Tabela 5.2 – Característica padrão

Características*	Descrição
http://xml.org/sax/features/external-general-entities	Se esta característica for true o *parser* trata toda referência à entidade externa. Se o *parser* for de validação esta característica deve ser true.
http://xml.org/sax/features/external-parameter-entities	Se for true, o *parser* trata toda referência a entidade de parâmetro externo. Para um *parser* de validação esta característica deve ser true.
http://xml.org/sax/features/string-interning	Se true, o *parser* pode comparar nome de elemento, prefixo, nome de atributo, namespace URI e nome local usando ==
http://xml.org/sax/features/validation	Se true, o *parser* valida o documento de acordo com a DTD especificada.
http://xml.org/sax/features/namespace	Se true, namespace URI e local name são analisados por `startElement()` e `endElement()`.
http://xml.org/sax/features/namespace-prefixes	Se true, o namespace URI e o prefixo de um atributo são incluídos na lista de atributos passada para `startElement()`, e o nome qualificador é passado como terceiro argumento deste método.

*O valor default destas características varia para cada parser.

As Tabelas 5.1 e 5.2 mostram, respectivamente, a lista das propriedades e das características padrões que podem ser usadas para configurar um objeto `XMLReader`. O

estado de uma característica é alterado pelo método `setFeature()`. Para mudar o objeto de um propriedade o método usado é o `setProperty()`.

Os métodos `setContentHandler()`, `setDTDHandler()`, `setErrorHandler()`, `setEntityResolver()` fazem a comunicação entre o *parser* e a aplicação. Eles permitem que o *parser* invoque os métodos de *callback* implementados na aplicação.

O método `parse()` é responsável por analisar o documento XML. Ele lê o documento e verifica sua formação: se o documento não estiver bem formado uma exceção é gerada. Ele pode receber como parâmetro uma string ou um objeto InputSource.

5.5.1 A Classe InputSource

O *parser* SAX utiliza o objeto `InputSource` para determinar como ler um documento de entrada XML. Esta entrada pode ser passada ao *parser* através de um identificador público, um identificador de sistema, um fluxo de byte e/ou um fluxo de caracter. Qualquer que seja a fonte, ela é encapsulada por um objeto `InputSource` [Megginson 1998].

Há dois locais onde uma aplicação pode devolver uma fonte de entrada para o *parser*: como argumento do método parse() ou como o valor de retorno do método `resolveEntity()` da interface `EntityResolver`.

Listagem 5.2 – Classe InputSource

```
public class InputSource {
  public InputSource()
  public InputSource(String systemID)
  public InputSource(InputStream byteStream)
  public InputSource(Reader characterStream)
  public void        setPublicId(String publicID)
  public String      getPublicId()
  public void        setSystemId(String systemID)
  public String      getSystemId()
  public void        setByteStream(InputStream byteStream)
  public InputStream getByteStream()
  public void        setEncoding(String encoding)
  public String      getEncoding()
  public void        setCharacterStream(Reader characterStream)
  public Reader      getCharacterStream()
}
```

5.6 As Interfaces Callback

Os métodos que o *parser* SAX chama para informar a ocorrência de eventos são chamados de métodos *callback*. As interfaces de *callback* são aquelas que definem métodos de

callback. As interfaces que oferecem este tipo de método são: `ContentHandler`, `DTDHandler`, `EntityResolver` e `ErrorHandler`.

5.6.1 A Interface ContentHandler

A grande maioria das aplicações que usam SAX implementa a interface `ContentHandler`. Os métodos definidos nesta interface permitem que uma aplicação receba informações sobre o início de um documento, início de elemento, dados de caracteres, fim de documento, fim de elemento, espaço em branco ignorável, instruções de processamento, entidades não analisadas e localizadores. Ela possui a assinatura mostrada na Listagem 5.3.

Listagem 5.3 – Interface ContentHandler

```
package org.xml.sax;
public interface ContentHandler {
   public void setDocumentLocator(Locator locator);
   public void startDocument() throws SAXException;
   public void endDocument() throws SAXException;
   public void startPrefixMapping(String prefix, String uri)
     throws SAXException;
   public void endPrefixMapping(String prefix)
      throws SAXException;
   public void startElement(String namespaceURI, String localName,
      String qualifiedName, Attributes atts) throws SAXException;
   public void endElement(String namespaceURI, String localName,
      String qualifiedName) throws SAXException;
   public void characters(char[] text, int start, int length)
      throws SAXException;
   public void ignorableWhitespace(char[] text, int start,
      int length) throws SAXException;
   public void processingInstruction(String target, String data)
      throws SAXException;
   public void skippedEntity(String name)
      throws SAXException;
}
```

5.6.1.1 Documento

Um `XMLReader` pode analisar vários documentos em série com o mesmo objeto `ContentHandler`: para fornecer a informação de onde começa e onde termina cada documento o *parser* invoca, respectivamente, os métodos `startDocument()` e `endDocument()`.

O `startDocument()` é o primeiro método a ser invocado quando do início do processamento de um documento. O `endDocument()` marca o fim da análise do documento.

Listagem 5.4 – Exemplo que usa os métodos SAX referentes a documento e elemento

```
import org.xml.sax.*;
import org.apache.xerces.parsers.SAXParser;
import java.io.*;
public class ElemExtractor implements ContentHandler {
       private Writer out;
       public void startDocument() {
              out = new OutputStreamWriter(System.out);
       }
       public void startElement(String namespaceURI, String localName,
              String qName, Attributes atts) throws SAXException {
              try {
                     String aux = "Inicio:" + qName+ "\r\n";
                     out.write(aux);
              } catch (IOException e) {
                     throw new SAXException(e);   }
       }
       public void endElement(String namespaceURI, String localName,
              String qName) throws SAXException {
              try {
                     String aux = "Fim:" + qName + "\r\n";
                     out.write(aux);
              } catch (IOException e) {
                     throw new SAXException(e);   }
       }
       public void endDocument() {
              try {
              out.flush();
              } catch (Exception e) {
                     System.err.println(e); }
       }
       // Métodos que não fazem nada
       public void setDocumentLocator(Locator locator) {}
       public void characters(char[] text, int start, int length){}
       public void startPrefixMapping(String prefix, String uri) {}
       public void endPrefixMapping(String prefix) {}
       public void ignorableWhitespace(char[] text, int start, int length)
              throws SAXException {}
       public void processingInstruction(String target, String data){}
       public void skippedEntity(String name) {}
       //Metodo main
       public static void main(String[] args) {
              if (args.length <= 0) {
                     System.out.println( "Usage: java ExtractorDriver url");
                     return;  }
              try {
                     XMLReader parser = new SAXParser();
                     ContentHandler handler = new ElemExtractor();
                     parser.setContentHandler(handler);
                     parser.parse(args[0]);
              } catch (Exception e) {
                      System.err.println(e);    }
       }
}
```

5.6.1.2 Elemento

Na verdade SAX reporta tags, não elementos. Quando uma tag de início é encontrada o método `startElement()` é chamado. Quando uma tag de fim é encontrada, o método

chamado é `endElement()`. Se uma tag de fechamento não tiver uma tag de início correspondente, então o parser gera uma exceção [Harold 2001].

`startElement()` e `endElement()` têm argumentos similares:

```
public void startElement(String namespaceURI, String localName,
   String qualifiedName, Attributes atts) throws SAXException;
public void endElement(String namespaceURI, String localName,
   String qualifiedName) throws SAXException;
```

Onde:

- `namespaceURI` é passado como uma string. Se o elemento não tiver namespace, este argumento é vazio;
- `localName` é à parte do nome depois do prefixo. Por exemplo, o elemento chamado `soap_event:Body`, seu localName é `Body`. Independente do elemento ter ou não prefixo, o localName continua o mesmo (Body);
- `qualifiedName` é o nome completo do elemento, incluindo o prefixo e os dois pontos (`soap_event:Body`);

O método `startElement()` ainda possui um quinto argumento, o `atts`. Este é um objeto `Attributes` que representa o conjunto de atributos do elemento.

A Listagem 5.4 mostra um exemplo de como os métodos que recebem informação sobre documento e elemento podem ser implementados em uma aplicação. Neste exemplo, o programa recebe um documento XML como argumento, analisa-o com um *parser* SAX e, se estiver bem formado, gera a saída mostrada na Listagem 5.5.

Listagem 5.5 – Resultado da execução do programa da Listagem 5.4

```
C:\Exemplos_SAX>java ElemExtractor pedidos.xml |more
Inicio:PEDIDOS
Inicio:PEDIDO
Inicio:CLIENTE
Fim:CLIENTE
Inicio:CD
Inicio:NOME
Fim:NOME
Inicio:ARTISTA
Fim:ARTISTA
Inicio:PRECO
Fim:PRECO
Fim:CD
Fim:PEDIDO
Inicio:PEDIDO
Inicio:CLIENTE
-- Mais --
```

Quando o documento começa a ser analisado e o método `startDocument()` é invocado, a variável `out`, que guarda todos os nomes das *tags*, é inicializada. Em `endDocument()` o conteúdo de `out` é impresso.

Listagem 5.6 – Exemplo com a classe DefaultHandler

```
import org.xml.sax.*;
import org.xml.sax.helpers.*;
import org.apache.xerces.parsers.SAXParser;
import java.io.*;
public class ElemExtractor2 extends DefaultHandler {
       private Writer out;
       public void startDocument() {
              out = new OutputStreamWriter(System.out);
       }
       public void startElement(String namespaceURI, String localName,
              String qName, Attributes atts) throws SAXException {
              try {
                     String aux = "Inicio:" + qName+ "\r\n";
                     out.write(aux);
              } catch (IOException e) {
                     throw new SAXException(e);
              }
       }
       public void endElement(String namespaceURI, String localName,
              String qName) throws SAXException {
              try {
                     String aux = "Fim:" + qName + "\r\n";
                     out.write(aux);
              } catch (IOException e) {
                     throw new SAXException(e);
              }
       }
       public void endDocument() {
              try {
              out.flush();
              } catch (Exception e) {
                     System.err.println(e);
              }
       }
       //Metodo main
       public static void main(String[] args) {
              if (args.length <= 0) {
                     System.out.println( "Usage: java ExtractorDriver url");
                     return;
              }
              try {
                     XMLReader parser = new SAXParser();
                     ContentHandler handler = new ElemExtractor2();
                     parser.setContentHandler(handler);
                     parser.parse(args[0]);
              } catch (Exception e) {
                      System.err.println(e);
              }
       }
}
```

5.6.1.3 A Classe DefaultHandler

Poucos programas SAX usam todos os métodos definidos em `ContentHandler`. Para evitar que estes programas sejam obrigados a especificar os métodos que não são usados como métodos *do-nothing* (faz nada), a especificação SAX inclui a classe `DefaultHandler` no pacote `org.xml.sax.helpers`, que fornece uma implementação *default* para as interfaces de callback (`EntityResolver`, `DTDHandler`, `ContentHandler` e `ErrorHandler`). O exemplo da Listagem 5.4 seria implementado como mostra a Listagem 5.6.

5.6.1.4 Atributos

SAX define a interface `Attributes` para representar atributos. Eles não são reportados através de callbacks. Um objeto `Attributes` contendo todos os atributos de um elemento é passado para o método `startElement()`.

Listagem 5.7 – Interface Attributes

```
public interface Attributes {
  public int    getLength ();
  public String getQName(int index);
  public String getURI(int index);
  public String getLocalName(int index);
  public int    getIndex(String uri, String localPart);
  public int    getIndex(String qualifiedName);
  public String getType(String uri, String localName);
  public String getType(String qualifiedName);
  public String getType(int index);
  public String getValue(String uri, String localName);
  public String getValue(String qualifiedName);
  public String getValue(int index);
}
```

Os métodos de `Attributes`, mostrados na Listagem 5.7, permitem obter o valor e o tipo de um atributo, caso o `qualifiedName` ou o `namespace` e `localName` sejam conhecidos. Se os nomes do atributo não são conhecidos, é possível obter estes dados (`qualifiedName`, `namespace`, `localName`) através do índice do atributo.

A interface `Attributes` é designada como uma lista, cada atributo possui um índice nesta lista. Entretanto, a ordem dos atributos na lista não é necessariamente a mesma ordem que aparece no documento. SAX2 inclui a classe `AttributesImpl` que implementa esta interface.

O exemplo da Listagem 5.8 altera a Listagem 5.6 para que o nome e valor dos

atributos de um elemento sejam também impressos na saída do programa. Ele acessa os itens da lista de atributos por meio do índice e utiliza os métodos `getLocalName()` e `getValue()`.

Listagem 5.8 – Exemplo que usa a interface Attributes

```
import org.xml.sax.*;
import org.xml.sax.helpers.*;
import org.apache.xerces.parsers.SAXParser;
import java.io.*;
public class AttrExtractor extends DefaultHandler {
       private Writer out;
       public void startDocument() {
              out = new OutputStreamWriter(System.out);
       }
       public void startElement(String namespaceURI, String localName,
              String qName, Attributes atts) throws SAXException {
              try {
                     String aux = "Elemento:" + qName + "\t";
                     if (atts.getLength()> 0){
                            aux += "Atributos: ";
                            for (int i=0;  i < atts.getLength(); i++)
                                   aux += atts.getLocalName(i) + "= " +
                                                 atts.getValue(i) +"\t";
                     }
                     aux += "\r\n";
                     out.write(aux);
              } catch (IOException e) {
                     throw new SAXException(e);   }
       }
       public void endDocument() {
              try {
              out.flush();
              } catch (Exception e) {
                     System.err.println(e); }
       }
       //Metodo main
       public static void main(String[] args) {
              if (args.length <= 0) {
                     System.out.println( "Usage: java ExtractorDriver url");
                     return;
              }
              try {
                     XMLReader parser = new SAXParser();
                     ContentHandler handler = new AttrExtractor();
                     parser.setContentHandler(handler);
                     parser.parse(args[0]);
              } catch (Exception e) {
                      System.err.println(e); }
       }
}
```

5.6.1.5 Caracteres

Quando o parser lê itens do tipo #PCDATA[2] ele passa o texto para o método `characters()` como um array de char.

Se há uma grande quantidade de texto entre duas tags sem intervenção de marcação, o parser pode escolher chamar `characters()` várias vezes, o que causa certo desconforto. Para acessar o conteúdo inteiro de um elemento como uma unidade é preciso criar uma variável do tipo *booleano* que indica o início do elemento, armazenar os dados em um buffer e apenas usá-los quando atingir a tag de fechamento do elemento no método `endElement()` [Harold 2001], como mostra a Listagem 5.9.

Listagem 5.9 – Exemplo do uso do método characters()

```
import org.xml.sax.*;
import java.io.*;
public class TextExtractor extends DefaultHandler {
  private Writer out;
  private boolean eTexto = false;

  public void startElement(String namespaceURI, String localName, String qName,
                                          Attributes atts) throws SAXException {
         eTexto = true;
         out = new OutputStreamWriter(System.out);
  }
  public  void  endElement(String  namespaceURI,  String  localName,  String  qName)
       throws SAXException {
       try {
           out.flush();
           eTexto = false;
         }catch (IOException e) {
             throw new SAXException(e);   }
  }
  public void characters(char[] text, int start, int length) throws SAXException{
    try {
       if (eTexto){
             out.write(text, start, length);
       }
    }catch (IOException e) {
      throw new SAXException(e);   }
  }
  ...
}
```

5.6.1.6 Instruções de Processamento

As instruções de processamento lidas pelo *parser* são passadas para o método `processingInstruction()`,: estão incluídas aqui as instruções que ocorrem antes e depois do elemento raiz.

[2] PCDATA = Nomenclatura usada em DTDs para determinar que o conteúdo de um elemento é formado por cadeia de carateres.

O método `processingInstruction()` possui dois argumentos: `target`, o alvo da instrução, e `data`, o dado. Ambos os argumentos são do tipo string. Se, por exemplo, uma desenvolvedor de aplicações quiser dividir o dado de uma folha de estilo em atributo e valor, terá que escrever o código que faça isto, SAX não fornece este recurso.

A declaração XML não é uma instrução de processamento. Ela não é passada para o método `processingInstruction()`. Na versão 2.1 de SAX são adicionadas algumas características e propriedades para recuperar o valor dos atributos `version`, `standalone` e `encoding`.

5.6.1.7 Espaço em Branco Ignorável (whitespace)

Espaço em branco é usado para organizar o documento de forma a torná-lo mais legível ao ser humano. Espaço em branco ignorável pode ser: espaço ( ), tab (	), carriage return (
), e line feed (
).

Considerando o documento abaixo, os espaços em branco entre `<aluno>` e `<nome>`, entre `</nome>` e `<disciplina>` são espaços em branco ignoráveis. Quando o parser validador encontra este tipo de espaço ele invoca o método `ignorableWhiteSpace()`.

```
<aluno>
      <nome> João</nome>
      <disciplina>Matemática</disciplina>
      <nota>9,00</nota>
<aluno>
```

Espaço em branco é considerado ignorável apenas onde #PCDATA é inválido. Os caracteres espaço e line break em um elemento string não são ignoráveis por que a DTD permite estes caracteres no conteúdo #PCDATA.

5.6.1.8 Mapeamento de Namespace

Atributos de declaração de namespace, tais como xmlns=”http://www.w3c.org/1999/xlink” não são incluídos na lista de atributos passada para o método `startElement()`. Este tipo de atributo é sinalizado por uma chamada ao método `startPrefixMapping()` imediatamente antes da chamada do `startElement()` correspondente à tag de início do elemento onde a declaração aparece. Além disso, a chamada `endElement()` correspondente à tag final deste

elemento é imediatamente seguida por uma chamada a `endPrefixMapping()` [Harold 2001].

A maioria das aplicações está interessada apenas nos atributos fornecidos ao método `startElement()`, ignorando assim os eventos de mapeamento de *namespace*.

5.6.1.9 Localizadores (Locators)

Localizador é usado freqüentemente para conhecer exatamente onde um item particular ocorre no documento. Para prover esta informação os *parsers* podem[3] implementar a interface `Locator`.

Um objeto Locator sabe em que ponto e em que arquivo o último evento foi disparado. Ele oferece identificadores `public` e `system` para a entidade em que a *tag* de início, *tag* de fim, instrução de processamento, etc. for encontrada. Ele informa em que linha e coluna o item começa.

Listagem 5.10 – Interface Locators

```
public interface Locator {
  public String getPublicId();
  public String getSystemId();
  public int    getLineNumber();
  public int    getColumnNumber();
}
```

Se o *parser* fornece informação de localização, então ele invocará o método `setDocumentLocator()` de `ContentHandler` antes de chamar o `startDocument()`. Se o mesmo `ContentHandler` é usado para analisar vários documentos, um novo objeto `Locator` será recebido para cada documento. No fim de um documento, os valores retornados pelo `Locator` não são válidos.

A Listagem 5.11 demonstra como usar um objeto `Locator` para informar a linha e a coluna onde começa cada um dos itens de um documento. O Resultado deste programa é mostrado na Listagem 5.12.

Listagem 5.11 – Exemplo de como usar a interface Locators

```
import org.xml.sax.*;
import org.xml.sax.helpers.XMLReaderFactory;
```

[3] Um parser não é obrigado a implementar a interface `Locator` apesar de ser recomendado.

```
public class LocatorDemo implements ContentHandler {
  private Locator locator;
  public void setDocumentLocator(Locator locator) {
    this.locator = locator;
  }
  private void printLocation(String s) {
    int line = locator.getLineNumber();
    int column = locator.getColumnNumber();
    System.out.println(s + " at line " + line + "; column " + column);
  }
  public void startDocument() {
    printLocation("startDocument()");
  }
  public void endDocument() {
    printLocation("endDocument()");
  }
  public void startElement(String namespaceURI, String localName,
   String qualifiedName, Attributes atts) {
   printLocation("startElement()");
  }
  public void endElement(String namespaceURI, String localName,
  String qualifiedName) {
    printLocation("endElement()");
  }
  public void characters(char[] text, int start, int length) {
    printLocation("characters()");
  }
  public void startPrefixMapping(String prefix, String uri) {
    printLocation("startPrefixMapping()");
  }
  public void endPrefixMapping(String prefix) {
    printLocation("endPrefixMapping()");
  }
  public void ignorableWhitespace(char[] text, int start,
   int length) {
    printLocation("ignorableWhitespace()");
  }
  public void processingInstruction(String target, String data) {
    printLocation("processingInstruction()");
  }
  public void skippedEntity(String name) {
    printLocation("skippedEntity()");
  }
  public static void main(String[] args) {
    if (args.length == 0) {
      System.out.println("Usage: java SAXSpider URL1");
    }
    String uri = args[0];
    try {
      XMLReader parser = XMLReaderFactory.createXMLReader(
                                 "org.apache.xerces.parsers.SAXParser");
      // Install the ContentHandler
      ContentHandler handler = new LocatorDemo();
      parser.setContentHandler(handler);
      parser.parse(uri);
    }
    catch (Exception e) {
      System.err.println(e);
    }
  } // end main
}
```

Listagem 5.12 – Resultado da execução do programa da Listagem 5.11

```
C:\Exemplos_SAX>java LocatorDemo pedidos.xml |more
startDocument() at line 1; column 1
startElement() at line 3; column 10
characters() at line 4; column 2
startElement() at line 4; column 10
characters() at line 5; column 3
startElement() at line 5; column 12
characters() at line 5; column 27
endElement() at line 5; column 37
characters() at line 6; column 3
startElement() at line 6; column 7
- Mais --
```

5.6.2 Outras Interfaces de Callback

A interface `ContentHandler` fornece a maioria das informações que uma aplicação precisa saber sobre um documento XML. As outras informações podem ser obtidas através das outras interfaces *callbacks* definidas por SAX:

5.6.2.1 A Interface DTDHandler

Através dessa interface é possível acessar notações e entidades não analisadas no corpo do documento. Caso a aplicação deseje receber notificações de eventos relacionadas a DTD, ela deve implementar essa interface.

Listagem 5.13 - Interface DTDHandler

```
package org.xml.sax;
public interface DTDHandler {
  public void notationDecl(String name, String publicID,
   String systemID) throws SAXException;
  public void unparsedEntityDecl(String name, String publicID,
   String systemID, String notationName) throws SAXException;
}
```

Listagem 5.14 - Interface EntityResolver

```
package org.xml.sax;
public interface EntityResolver {
  public InputSource resolveEntity(String publicId,
   String systemId) throws SAXException, IOException;
}
```

5.6.2.2 A Interface EntityResolver

Esta é uma interface básica para resolver entidades, isto é, quando uma entidade externa é encontrada em um documento XML, o parser automaticamente localiza o arquivo

referenciado e o analisa. A interface `EntityResolver` pode ser implementada para que o este comportamento seja alterado.

5.6.2.3 A Interface ErrorHandler

A especificação XML define três classes de problemas que podem ocorrer em um documento XML:

- Fatal error – erro de formação (documento não está bem formado)
- Error – o tipo mais comum deste problema é erro de validação
- Warning – aviso, advertência.

Um parser deve obrigatoriamente gerar uma exceção quando um erro fatal ocorrer. Os demais problemas podem ou não ser informados. Para que uma aplicação tome conhecimento sobre eles é necessário que a interface `ErrorHandler` seja implementada e esta implementação seja registrada com o `XMLReader` através do método `setErrorHandler()`.

Listagem 5.15 - Interface ErrorHandler

```
package org.xml.sax;
public interface ErrorHandler {
  public void warning(SAXParseException exception)
   throws SAXException;
  public void error(SAXParseException exception)
   throws SAXException;
  public void fatalError(SAXParseException exception)
   throws SAXException;
}
```

As exceções SAX são derivadas da classe genérica `SAXException`. Tanto o método `parse()` quanto os métodos de *callback* são declarados para disparar esta exceção.

5.7 Considerações Finais

Neste capítulo mostramos como trabalha uma interface baseada em eventos e, em particular, descrevemos as interfaces básicas de SAX: `XMLReader` e `ContentHandler`. A primeira representa o *parser*, enquanto `ContentHandler` envia a aplicação informações sobre os itens lidos no documento XML.

No próximo capítulo será realizado um estudo comparativo sobre parsers DOM e SAX.

Capítulo 6 - Análise Comparativa das APIs DOM e SAX

Este capítulo apresenta uma análise comparativa entre as APIs DOM e SAX. São mostrandas as categorias de aplicações XML onde DOM e SAX podem ser usadas, uma comparação entre estas APIs destacando suas vantagens e desvantagens e quando cada uma deve ser aplicada, o estudo de caso realizado, e a performance das APIs na aplicação desenvolvida no estudo de caso.

6.1 Introdução

XML possui inúmeras aplicações, devido ao seu formato flexível, aberto e de fácil leitura. Estas aplicações podem ser categorizadas da seguinte forma [Bosak 1997]:

1. Transferência de dados entre sistemas heterogêneos – O uso das APIs DOM e SAX nesta categoria não é estritamente necessária, pois os dados podem ser transmitidos em forma de arquivos textos;
2. Distribuição de processamento entre o cliente e o servidor – Neste tipo de aplicação, o uso das APIs DOM e SAX vai depender do tipo de processamento. Alguns *browsers*, como o Internet Explorer 5.0 ou superior, suportam o padrão DOM. Por exemplo, o servidor poderia enviar para o *browser* um pedido com a lista de itens e o total do pedido ser calculado pelo *browser*;
3. Apresentação de diferentes visões dos mesmos dados para diferentes usuários – Apesar de ser possível fazer isto utilizando DOM, o W3C criou o padrão XSL [Martin 2000], que é especifico para transformações e apresentação de documentos XML;
4. Armazenamento de dados em XML – Este tipo de aplicação necessitam acessar os dados XML que devem estar disponíveis de alguma forma. As APIs DOM e SAX são apropriadas para este tipo de aplicação.

A aplicação desenvolvida no estudo de caso pertence à categoria 4. Existem algumas vantagens em armazenar os dados em XML:

- Portabilidade – Os arquivos XML são arquivos textos suportados virtualmente por qualquer sistema operacional;
- Formato auto descritivo – As próprias *tags* embutidas nos documentos XML descrevem os seus dados; diferentemente de um arquivo texto simples, onde exige uma documentação extra por parte do programador;
- Disponibilidade de APIs - As APIs DOM e SAX servem para disponibilizar os dados XML dos documentos para a aplicação;

- Validação dos dados – Os dados podem ser validados na hora de serem recuperados pela aplicação. Os dois principais padrões para a validação de documentos XML são DTD e XML Schema [Holzner 2001, Marchal 2000, Martin 2000];
- Integração com outros sistemas – XML é um padrão aberto e existe uma variedade de ferramentas, parsers, SGBDs (Sistemas Gerenciadores de Banco de Dados) que suportam este padrão.

A próxima seção apresenta as vantagens e desvantagens em utilizar as APIs DOM e SAX.

6.2 DOM X SAX

Como foi mostrado nos capítulos anteriores, DOM e SAX são APIs com propriedades e características diferentes. Estas propriedades e características definem pontos positivos e negativos em cada um destes padrões.

6.2.1 Vantagens de DOM

A API DOM, em função das suas características descritas nos Capítulos 3 e 4, possui várias vantagens em relação a API SAX.

DOM define uma estrutura de árvore que oferece uma visão que é exatamente a estrutura abstrata de um documento. Esta estrutura garante que as alterações realizadas em um documento não geram problemas de má formação, tais como, *tag* de início sem *tag* de fechamento, e aninhamento incorreto de elementos.

A especificação DOM define métodos do tipo create, insert e add que permitem modificar documentos. Estas modificações podem ser gravadas em arquivo XML utilizando o módulo Load and Save de DOM 3, como visto anteriormente.

Além destas vantagens, este padrão é suportado por navegadores populares como o Internet Explorer e o Netscape [Holzner 2001].

6.2.2 Desvantagens de DOM

DOM é uma interface baseada em objetos que cria um modelo de objetos na memória para representar o documento XML. Esta característica torna DOM fácil de usar, mas acarreta algumas limitações tais como: problemas de eficiência, grande consumo de memória,

tempo de espera para criar a árvore, estrutura de dados já predefinida, entre outros [Marchal 2000].

Para cada item do documento XML, DOM cria um nó na árvore de objetos. Esta árvore ocupa mais espaço de memória do que o documento original, consumindo assim muito recurso do sistema. Logicamente, a quantidade de memória usada pelo DOM depende do *parser*, conforme será discutido no Capítulo 7. Mas, em muitos casos, pelos testes realizados, a árvore de nós chega a ocupar mais de quatro vezes o tamanho do documento.

Em DOM, para que uma aplicação possa processar dados XML, o documento inteiro deve estar carregado na memória. Isto implica em esperar até que a árvore esteja completamente construída [Hustead 2000].

Outra questão que também deve ser observada é que algumas aplicações não usam realmente a estrutura genérica de DOM [Hustead 2000]. Assim a árvore DOM é carregada na memória e o programa copia os dados em um modelo de objeto específico para o domínio do problema.

Além disso, o código escrito para DOM deve fazer a varredura do documento XML duas vezes. Na primeira vez a estrutura de árvore na memória é criada, na segunda vez encontra os dados XML que o programa está interessado. Alguns estilos de codificação podem atravessar a estrutura DOM diversas vezes ao encontrar partes diferentes de dados XML [Hustead 2000].

6.2.3 Vantagens de SAX

Como foi apresentado no Capítulo 5, SAX é uma API baseada em eventos. Ela gera eventos que informam a aplicação sobre os itens encontrados no documento analisado. Este padrão não define uma estrutura para ser utilizada, a aplicação pode usar sua própria estrutura de dados para representar o documento. Isto também garante ao SAX um consumo menor de recursos do sistema para processamento de documentos.

Em SAX, os documentos são analisados à medida que estão sendo lidos. A aplicação pode, então, iniciar a execução antes da leitura completa do documento. Como ele não precisa estar completamente carregado na memória, é possível analisar documentos de qualquer tamanho [Means 2001a].

Outra vantagem deste padrão é que ele é capaz de processar apenas partes do documento, além de processar muitos documentos simultaneamente.

6.2.4 Desvantagens de SAX

Como qualquer outra API, SAX também possui alguns problemas. Além das dificuldades referentes à complexidade do código, acesso seqüencial e falta de representação do documento, o paradigma orientado a eventos não é muito familiar a muitos desenvolvedores [Clark 2001] e SAX não é suportado pelos navegadores atuais.

Um *parser* SAX lê um documento XML seqüencialmente e dispara eventos para cada item que encontra. A aplicação, por sua vez, tem que lidar com os dados na ordem que eles chegam, pois não há uma representação do documento na memória. O que impede acesso não seqüencial ao documento.

Por causa do acesso em série, a interface SAX pode ser bem difícil de usar quando o documento contiver muitas referências cruzadas internas, por exemplo, usando atributos ID e IDREF[4] [Martins 2000].

Muitas vezes implementações que usam SAX requerem um pouco mais de trabalho de programação, pois é responsabilidade do desenvolvedor implementar as funções que serão executadas através das chamadas dos eventos, e manter as estruturas de dados contendo qualquer informação de contexto necessária.

A falta de uma representação do documento torna a manipulação, serialização e o atravessamento do documento XML um desafio maior [Franklin]. Além disso, SAX é projetado para ler documentos XML, e não define métodos para criar ou modificar um documento.

6.3 Quando usar DOM e Quando usar SAX

Os desenvolvedores de aplicações XML têm a opção de trabalhar com a API DOM ou com SAX. Para escolher o padrão que seja mais adequado para o problema é importante, além de conhecer as características de cada API, saber onde cada uma deve ser melhor aplicada.

[4] ID = atributo identificador de XML e IDREF = atributo XML que faz referência a um identificador.

A seguir são apresentados alguns requisitos que podem influenciar na escolha da API adequada.

Quando é melhor usar DOM:

- Quando o programa necessita acessar partes extensamente separadas do documento ao mesmo tempo, por exemplo o primeiro e o último filho da raiz do documento;
- Quando é necessário manipular a estrutura do documento e modificá-lo [Idris 1999];
- Quando é preciso compartilhar o documento com outras aplicações;
- Quando o tamanho do documento não ocupar uma grande quantidade de recursos, de modo que a performance da aplicação não seja um fator crítico.

Quando é melhor usar SAX

- Quando o documento é grande [Means 2001a];
- Quando o documento inteiro não estiver disponível. Um parser SAX é capaz de processar o documento em pequenos pedaços contínuos;
- Quando a performance é um fator crítico para sua aplicação;
- Quando é preciso processar os elementos de forma linear;
- Quando não há necessidade de manipular a estrutura do documento [Idris 1999].

Existem ocasiões onde é possível usar SAX e DOM conjuntamente. Por exemplo, é possível usar um `XMLReader` de SAX, e com a saída deste processo construir um `Document` DOM. Ou então, atravessar uma árvore DOM enquanto envia eventos para `ContentHandler` SAX [Harold 2002].

A seção seguinte apresenta um exemplo de uma aplicação que armazena dados em documentos XML. Para efeito de análise de performance e evidências das características das APIs, desenvolveu-se duas implementações para os mesmos serviços, uma utilizando a API DOM e a outra SAX.

6.4 Uma Aplicação Exemplo - SRC

Nesta seção é apresentado um protótipo da aplicação SRC (Sistema de Representação

Comercial) que gerencia pedidos de clientes. SRC permite cadastrar pedidos de clientes, adicionar novos produtos aos pedidos e alterar pedidos existentes.

Para cadastrar um pedido é necessário fornecer os dados do cliente (nome, endereço e telefone), os dados dos produtos, chamados de itens do pedido (código, nome, quantidade, valor), e os dados do fornecedor (nome, endereço e telefone). Uma vez cadastrado um pedido, esse pode ser alterado ou removido. As classes que compõem o sistema e os seus relacionamentos são mostrados na Figura 6.1.

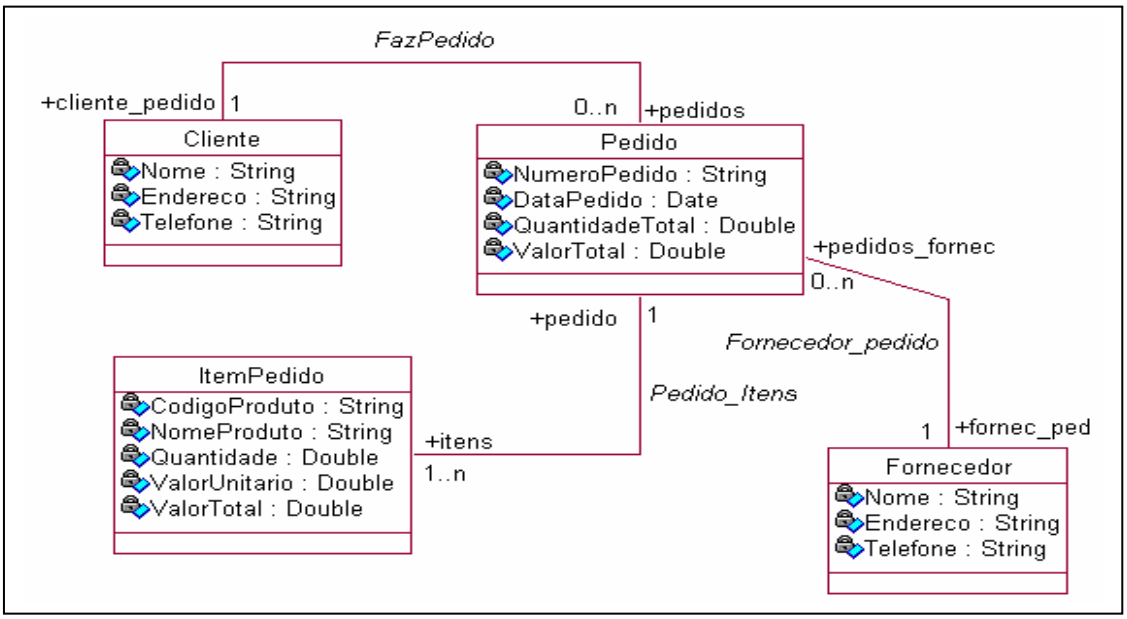


Figura 6.1 – Modelo de classes da aplicação SRC

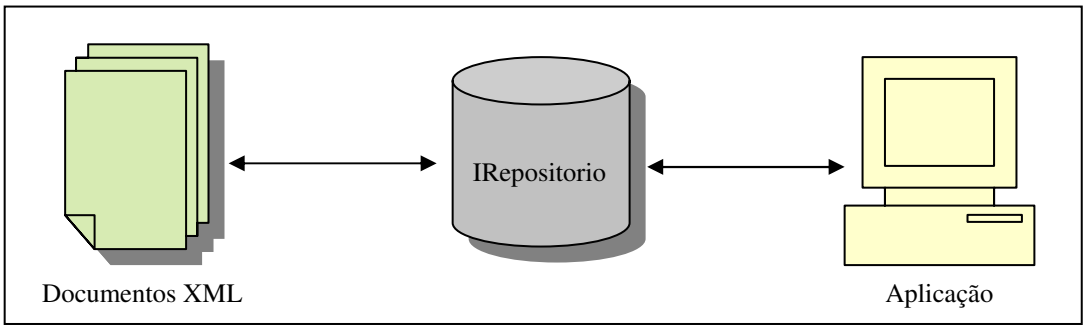


Figura 6.2 - Arquitetura da aplicação SRC

Os pedidos são armazenados em documentos XML para consultas posteriores. Os dados desses documentos são disponibilizados para a aplicação através de uma interface chamada `IRepositorio`, que oferece diversos métodos para recuperar e atualizar os objetos

armazenados nos documentos XML. A arquitetura dessa aplicação é apresentada na Figura 6.2.

6.4.1 Implementação do SRC

Como foi dito anteriormente, o SRC utiliza sistema de arquivo para armazenar os documentos XML, porém outras formas de persistência poderiam ser utilizadas [Bourret 2000]: banco de dados realcional, banco de dados objeto-relacional, banco de dados objeto ou banco de dados XML nativo.

O objetivo deste protótipo é mostrar de forma simples a manipulação de documentos XML com DOM e SAX, por isto fatores como atomicidade, concorrência, segurança e integridade não foram considerados.

Na implementação desse sistema foi utilizado o SDK[5] 1.4 e o *parser* Xerces 2. Esse *parser* foi escolhido porque suporta ambas as APIs, DOM e SAX, e implementa o módulo *Load and Save* de DOM3. Mais detalhes sobre Xerces 2 são encontrados no Capítulo 7.

Foram construídas duas DTDs equivalentes ao modelo de classes da Figura 6.1. Cada classe do modelo é representada por um elemento definido na DTD. Os atributos das classes são definidos como atributos do elemento correspondente. Os relacionamentos estão implícitos na própria hierarquia do documento XML.

As duas DTDs validam dois tipos de documentos: um para armazenar a lista de todos os pedidos e um outro para armazenar cada pedido individualmente. As Listagens 6.1 e 6.2 apresentam estas DTDs, e a Figura 6.3 mostra dois exemplos de documentos XML validados pelas duas DTDs.

Listagem 6.1 – DTD para o documento pedidos

```
<!ELEMENT Pedidos (Pedido*)>
<!ELEMENT Pedido EMPTY>
<!ATTLIST Pedido
       numeroPedido CDATA #REQUIRED
       DataPedido CDATA #REQUIRED
       QuantidadeTotal CDATA #REQUIRED
       ValorTotal CDATA #REQUIRED
>
```

[5] Antes do Java 2, o SDK era chamado de JDK (*Java Development Kit*)

Listagem 6.2 – DTD para o documento pedido

```
<!ELEMENT Pedido (Fornecedor,Cliente, ItemPedido+)>
<!ATTLIST Pedido
       numeroPedido CDATA #REQUIRED
       DataPedido CDATA #REQUIRED
       QuantidadeTotal CDATA #REQUIRED
       ValorTotal CDATA #REQUIRED
>
<!ELEMENT Fornecedor EMPTY>
<!ATTLIST Fornecedor
       Nome CDATA #REQUIRED
       Endereco CDATA #REQUIRED
       Telefone CDATA #REQUIRED
>
<!ELEMENT Cliente EMPTY>
<!ATTLIST Cliente
       Nome CDATA #REQUIRED
       Endereco CDATA #REQUIRED
       Telefone CDATA #REQUIRED
>
<!ELEMENT ItemPedido EMPTY>
<!ATTLIST ItemPedido
       CodigoProduto CDATA #REQUIRED
       NomeProduto CDATA #REQUIRED
       Quantidade CDATA #REQUIRED
        ValorUnitario CDATA #REQUIRED
       ValorTotal CDATA #REQUIRED
>
```

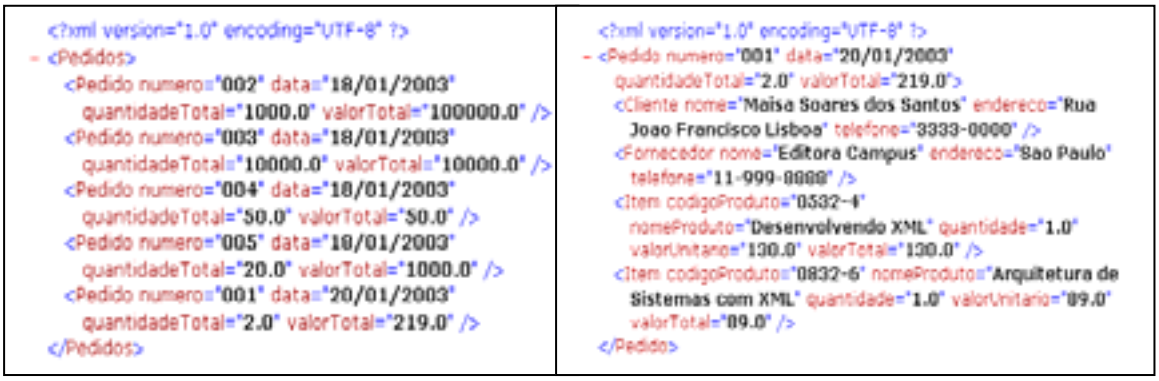


Figura 6.3 – Exemplos dos documentos pedidos.xml e pedido.xml

A interface `IRepositorio`, que disponibiliza os dados para a aplicação, é mostrada na Listagem 6.3. Os dois métodos `RecuperarPedidos()` e `SalvarPedidos()` atuam sobre o documento de pedidos. Os outros métodos atuam sobre um pedido individual. Os métodos `Recuperar` e `Atualizar` recebem como parâmetro o pedido referente ao documento pedido a ser recuperado.

Listagem 6.3 – A interface do repositório

```
public interface IRepositorio {
       java.util.List recuperarPedidos();
       void salvarPedidos();
       Pedido criarNovoPedido(String numPedido, String dataPedido);
       void deletarPedido (Pedido pedido);
       void salvarPedido (Pedido pedido);
       Pedido recuperarPedido (String numeroPedido);
       void atualizarPedido (Pedido pedido);
       Cliente recuperarCliente (Pedido pedido);
       void atualizarCliente (Pedido pedido, Cliente cliente);
       Fornecedor recuperarFornecedor (Pedido pedido);
       void atualizarFornecedor (Pedido pedido, Fornecedor fornecedor);
       void adicionarItemPedido (Pedido pedido, ItemPedido item);
       ItemPedido removerItem (Pedido pedido, String codigoProduto);
       void limparListaItem (Pedido pedido);
}
```

Para realizar uma comparação entre as APIs DOM e SAX foram construídas duas implementações da interface `IRepositorio`, uma para a API DOM, de nome `RepositorioXMLDOM`, e a outra para a API SAX, de nome `RepositorioXMLSAX`, que serão apresentadas nas próximas seções.

6.4.2 Repositório DOM

O repositório DOM, neste sistema representado pela classe `RepositorioXMLDOM`, tem como papel principal manipular documentos XML, onde são armazenados dados de pedidos de clientes, usando interfaces e métodos definidos nas especificações DOM mostradas nos Capítulos 3 e 4. A Listagem 6.4 apresenta as estruturas de dados utilizadas por esta classe.

Listagem 6.4 - A classe RepositorioXMLDOM

```
public class RepositorioXMLDOM implements IRepositorio{
       private static final String caminhoDoc = "H:\\Estudodecaso\\Dados\\P";
       private Document doc, docPedidos;
       private String numPedAtual;
       ...
}
```

Os atributos `doc` e `docPedidos` armazenam, respectivamente, o documento referente a um determinado pedido e o documento referente a todos os pedidos.O atributo `numPedAtual` armazena o número do pedido corrente e a constante `caminhoDoc` armazena o diretório onde o pedido deve ser armazenado.

O repositório DOM utiliza a própria estrutura DOM para armazenar os pedidos. Essa estrutura é chamada de *cache*. Os métodos da interface `IRepositorio` são executados sobre o *cache*. Toda vez que um método for acessar a estrutura DOM ele executará o método privado `recuperarPedidoXML`, mostrado na Listagem 6.5. Este método verifica se existe um pedido no *cache*, se existir ele atualiza o pedido no arquivo XML e, logo após, carrega para o *cache* o novo pedido.

Listagem 6.5 – Método para recuperar pedido XML utilizando DOM

```
private void recuperarPedidoXML (String numPedido) {
    if (numPedAtual != null){
      salvarPedidoAtual();
    }
    String uri =  caminhoDoc + numPedido + ".xml";

    try {

        DOMParser parser = new DOMParser();
        parser.parse(uri);
        doc = parser.getDocument();
        numPedAtual =numPedido;
    } catch (Exception e){
      e.printStackTrace(System.err);
   }
}
```

Este método poderia ainda, realizar a validação dos dados XML utilizando a DTD da Listagem 6.2. Se o documento XML não for válido, uma exceção poderia ser emitida, dizendo que os dados não são válidos. Para isto, bastaria configurar o *parser* para fazer a validação (veja Capítulo 7) e colocar a referência da DTD no documento XML.

Listagem 6.6 – Método para recuperar cliente do repositório DOM

```
public Cliente recuperarCliente (Pedido pedido){
   Cliente cliente;
   try {
       if (!verificarPedidoAtual(pedido.getNumeroPedido())) {
          recuperarPedidoXML(pedido.getNumeroPedido());
        }
       cliente =
             FabricaXML.NodeToCliente((Element)doc.getDocumentElement().
                                getElementsByTagName("Cliente").item(0));
    }
    catch (Exception e){
      e.printStackTrace(System.err);}
    return cliente;
}
```

6.4.2.1 Recuperando um Objeto do Cache

O método que recupera um determinado objeto do *cache* verifica se o objeto a ser recuperado pertence ao pedido corrente, se não pertencer, ele invoca o método privado `recuperarPedidoXML` da Listagem 6.5. Logo após, ele invoca o método da classe `FabricaXML` que recebe como parâmetro um nó XML e gera o objeto da consulta. A Listagem 6.6 mostra o método para recuperar o cliente de um pedido.

6.4.2.2 Atualizando um Objeto no Cache

O método para atualizar um determinado objeto recebe como parâmetro o objeto a ser atualizado e o transforma em um nó XML na estrutura DOM. Antes de atualizar o nó, ele verifica se o pedido corrente é o mesmo passado como parâmetro. Este método invoca o método da classe `FabricaXML` que transforma um objeto em um nó XML. A Listagem 6.7 apresenta a implementação do método para atualizar um objeto fornecedor no *cache*.

Listagem 6.7 – Método para atualizar fornecedor no repositório DOM

```
public void atualizarFornecedor (Pedido pedido, Fornecedor fornecedor){
   try{
       if (!verificarPedidoAtual(pedido.getNumeroPedido())) {
          recuperarPedidoXML(pedido.getNumeroPedido());
        }
        Node nodeAntigo= doc.getDocumentElement().getElementsByTagName(
                                                    "Fornecedor").item(0);
        Node nodeNovo = FabricaXML.FornecedorToNode(
                                      fornecedor, "Fornecedor", doc);

        doc.getDocumentElement().replaceChild(nodeNovo, nodeAntigo);
    } catch (Exception e){
      e.printStackTrace(System.err);
    }
}
```

6.4.2.3 Outros Métodos do Repositório

Além dos métodos mostrados anteriormente, o repositório possui outros métodos importantes, como:

- `salvarPedido(Pedido pedido)` – atualiza o arquivo XML no disco a partir do *cache* DOM.;
- `criarNovoPedido (String numPedido, String dataPedido)` – cria um novo documento XML referente a um novo pedido;

- `deletarPedido (Pedido pedido)` – remove o documento XML do disco referente ao pedido passado como parâmetro.

6.4.2.4 Fábrica XML

Uma classe importante no repositório DOM é a `FabricaXML` que tem como objetivo transformar os objetos de negócio em nó XML e vice-versa. O método que transforma um determinado objeto em nó XML cria um elemento XML referente ao objeto e os atributos desse objeto são armazenados como atributos do elemento. Se o objeto possui uma referência a um outro objeto esse é transformado em subelemento. A Listagem 6.8 apresenta o método para criar um nó XML a partir do objeto pedido.

Listagem 6.8 – Método para criar um nó XML a partir de um objeto

```
public static Node pedidoToNode(Pedido pedido, String nomeElem, Document doc) {
  Element elemento = null;
  if (doc !=null) {
     elemento = doc.createElement(nomeElem);
     elemento.setAttribute("numero", pedido.getNumeroPedido());
     elemento.setAttribute("data", pedido.getDataPedido());
       ...
     elemento.appendChild(clienteToNode(pedido.getCliente(), "Cliente",
                                                                    doc));
     elemento.appendChild(fornecedorToNode(pedido.getFornecedor(),
                                                  "Fornecedor", doc));
     for (java.util.Iterator it = pedido.getListaItens().iterator();
                                                      it.hasNext();){
        elemento.appendChild(itemPedidoToNode((ItemPedido)it.next(),
                                                        "Item", doc));
     }
    }
    return elemento;
}
```

O método para criar um objeto a partir de um nó XML está apresentado na Listagem 6.9. Os valores dos atributos XML são copiados para os atributos do objeto criado.

Listagem 6.9 – Método para criar um objeto a partir de um nó XML

```
public static Cliente nodeToCliente (Element elemento) {
    Cliente cliente = new Cliente();
    if (elemento!=null) {
       cliente.setNome(elemento.getAttribute("nome"));
       cliente.setEndereco(elemento.getAttribute("endereco"));
       cliente.setTelefone(elemento.getAttribute("telefone"));
    }
    return cliente;
}
```

6.4.3 Repositório SAX

A classe `RepositorioXMLSAX` que implementa a interface `IRepositorio` apresentada anteriormente, utiliza a API SAX para acessar os documentos XML. Porém, como SAX não possui nenhuma estrutura para armazenamento dos dados XML em memória, foi definido como estrutura para o *cache* do repositório SAX os próprios objetos das classes base do sistema. A Listagem 6.10 apresenta as estruturas de dados utilizadas por esta classe.

Listagem 6.10 - A classe RepositorioXMLSAX

```
public class RepositorioXMLSAX implements IRepositorio{
       private static final String caminhoDoc = "H:\\Estudodecaso\\Dados\\P";
       private Pedido cachePedido;
       private java.util.List cachePedidos;
       private String numPedAtual;
       ...
}
```

Os atributos `cachePedido` e `cachePedidos` armazenam, respectivamente, o pedido que está aberto e a lista de todos os pedidos. O atributo `numPedAtual` armazena o número do pedido corrente que está no *cache*.

Listagem 6.11 – Método para recuperar pedido XML utilizando SAX

```
private void recuperarPedidoXML (String numPedido) {
    if (numPedAtual != null){
      salvarPedidoAtual();
    }
    String uri = caminhoDoc + numPedido + ".xml";
    try {
        XMLReader parser = new SAXParser();
        HandleSAX repsax = new HandleSAX();
        parser.setContentHandler(repsax);
        parser.parse(uri);
        cachePedido = repsax.getPedido();
        numPedAtual =numPedido;
    } catch (Exception e){
      e.printStackTrace(System.err);    }
}
```

Desta maneira, os métodos r`ecuperar` e `atualizar` são executados sobre o objeto `cachePedido` e sobre a lista `cachePedidos`. Da mesma forma que a implementação DOM, toda vez que um método for acessar a estrutura *cache* ele executará o método privado r`ecuperarPedidoXML`, apresentado na Listagem 6.11. Este método verifica se existe um pedido no *cache*, se existir ele atualiza o pedido no arquivo XML e, logo após, carrega para o *cache* o novo pedido.

Listagem 6.12 – Classe que implementa os eventos de SAX

```
public class HandleSAX extends org.xml.sax.helpers.DefaultHandler {
...
  public void startElement(String namespaceURI, String localName,
                   String qName, Attributes atts) throws SAXException {
        try {
           if (localName.equals("Pedido")) {
               String numero ="";
               String data = "";
               double qtd =0;
               double vTotal = 0;
              for (int i=0;  i < atts.getLength(); i++){
                 if ((atts.getLocalName(i)).equals("numero")) {
                   numero = atts.getValue(i);
                 }
                 if ((atts.getLocalName(i)).equals("data")) {
                   data = atts.getValue(i);
                 }
                   ...
              }
              pedAtual = new Pedido();
              pedAtual.setNumeroPedido(numero);
              pedAtual.setDataPedido(data);
              pedAtual.setQuantidadeTotal(qtd);
              pedAtual.setValorTotal(vTotal);
              listapedidos.add(pedAtual);
          }
          if (localName.equals("Cliente")){
            for (int i=0;  i < atts.getLength(); i++){
              if ((atts.getLocalName(i)).equals("nome"))
                      pedAtual.getCliente().setNome(atts.getValue(i));
              ...
              }
          }
          if (localName.equals("Fornecedor")){
                ...
          }
          if (localName.equals("Item")){
             ItemPedido item = new ItemPedido();
             for (int i=0;  i < atts.getLength(); i++){
                    if ((atts.getLocalName(i)).equals("codigoProduto"))
                      item.setCodigoProduto(atts.getValue(i));
                    ...
             }//for
             pedAtual.AdicionarItem(item);
          }
          } catch (Exception e) {
             throw new SAXException(e); }
 }
}
```

O método recuperar da Listagem 6.11 carrega para o *cache* do repositório um determinado pedido a partir de um documento XML utilizando a interface SAX. Como explicado no Capítulo 5, é necessário definir uma classe que implementa os métodos referentes aos eventos gerados pela API SAX. Para essa aplicação foi definida a classe `HandleSAX` e implementado apenas o método `startElement()`, apresentado na Listagem

6.12. Esse método é executado toda vez que um novo elemento for encontrado durante o processamento do documento XML.

Durante o processamento do arquivo XML, os dados são copiados para as estruturas do repositório. Ao final do processamento todo o pedido já estará no *cache*.

6.4.3.1 Recuperando um Objeto do Cache

Os métodos para recuperar os objetos do *cache* não requerem nenhuma transformação, ou seja, os dados a serem retornados estão no próprio *cache*. A Listagem 6.13 apresenta o método para recuperar o cliente do pedido.

Listagem 6.13 – Método para recuperar cliente do repositório SAX

```
public Cliente recuperarCliente (Pedido pedido){
       if (!verificarPedidoAtual(pedido.getNumeroPedido())) {
          recuperarPedidoXML(pedido.getNumeroPedido());
        }
    return cachePedido.getCliente();
}
```

6.4.3.2 Atualizado um Objeto de Negócio no Cache

Os métodos para atualizar os objetos no *cache* envolvem apenas a cópia dos valores dos atributos de um objeto para outro. A Listagem 6.14 apresenta a implementação do método para atualizar um objeto fornecedor no *cache*.

Listagem 6.14 - Método para atualizar fornecedor no repositório SAX

```
public void atualizarFornecedor (Pedido pedido, Fornecedor fornecedor){
     if (!verificarPedidoAtual(pedido.getNumeroPedido())) {
          recuperarPedidoXML(pedido.getNumeroPedido());
     }
     cachePedido.getFornecedor().setNome(fornecedor.getNome());
     cachePedido.getFornecedor().setEndereco(fornecedor.getEndereco());
     cachePedido.getFornecedor().setTelefone(fornecedor.getTelefone());
}
```

6.4.4 Construindo um Cliente

Para efeito de exemplo, esta seção apresenta uma aplicação que utiliza os métodos da interface `IRepositorio`. Foi criado um *frame* Java que permite cadastrar pedidos e escolher a API que deseja usar. A Figura 6.4 mostra as telas do sistema que permite adicionar, remover e abrir e cadastrar os dados de um pedido, além de permitir escolher a API que será utilizada na próxima execução do programa.

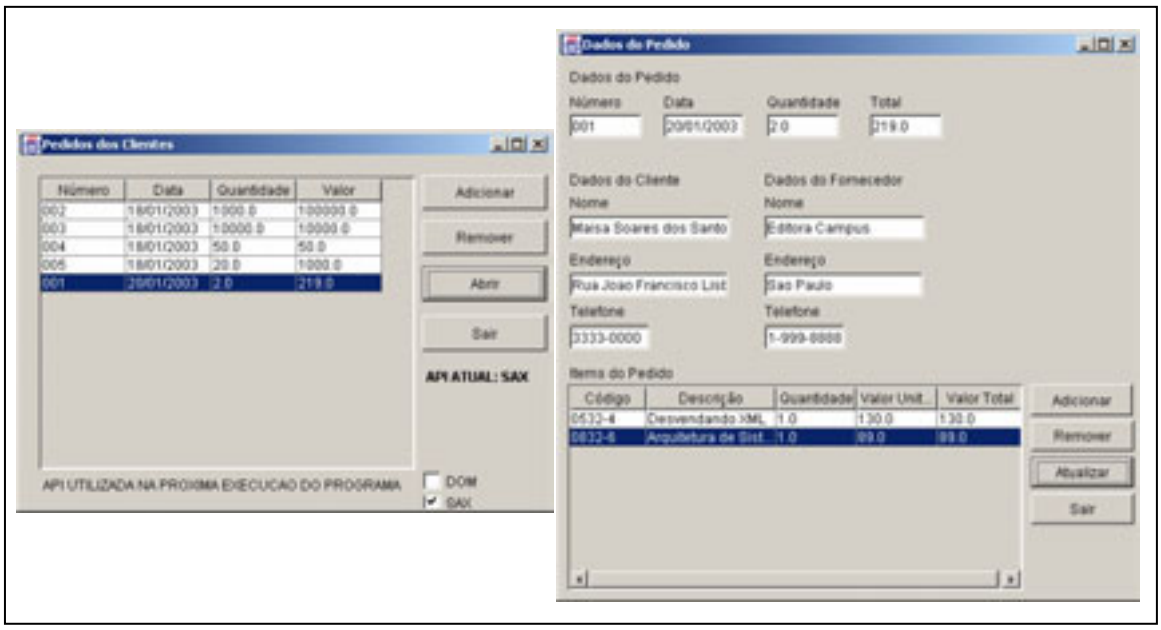


Figura 6.4 – Telas do protótipo SRC

6.5 Uma Comparação Entre as Implementações DOM e SAX

O protótipo SRC, descrito na seção anterior, utilizou as duas implementações do repositório (`RepositorioXMLDOM`, `RepositorioXMLSAX`). Por causa das características já citadas das APIs DOM e SAX, cada implementação obteve desempenho diferente. Para avaliar esse desempenho, foram testadas três funções que mostram as diferentes características das APIs DOM e SAX: recuperar pedido, atualizar pedido e salvar pedido.

Os testes foram realizados da seguinte forma:

1) Foram selecionados quatro documentos XML de acordo com a quantidade de itens do pedido (pequeno, médio, grande). A relação dos documentos é mostrada na Tabela 6.1;

Tabela 6.1 - Documentos XML usados nos testes do SRC

Arquivo	Tamanho (KB)	Num. de Itens de Pedidos
P001	10	75
P002	50	423
P003	100	854
P004	400	3516

2) Os métodos recuperar, atualizar e salvar de cada repositório foram alterados para medir o tempo gasto em suas execuções. O método utilizado para obter o tempo de execução foi `System.currentTimeMillis()`. Os tempos obtidos foram automaticamente gravados em um arquivo texto como mostra a Listagem 6.15;

Listagem 6.15 – Método SalvarPedido alterado para medir o tempo de execução

```
public void SalvarPedido (Pedido pedido){
    if (VerificarPedidoAtual(pedido.getNumeroPedido())) {
      long tempoinicial = System.currentTimeMillis();
      SalvarPedidoAtual();
      long tempofinal = System.currentTimeMillis();
      try{
       log.write("\nTEMPO p/ Salvar Pedido DOM : " +
                          Double.toString(tempofinal - tempoinicial));
      } catch (Exception e){
       e.printStackTrace(System.err);
      }
    }
}
```

3) Para cada documento, os métodos de cada API foram executados dez vezes; e
4) Os tempos que ocorreram com maior freqüência foram selecionados. Os resultados selecionados podem ser conferidos na Tabela 6.2, para o repositório DOM, e na Tabela 6.3, para o repositório SAX.

Tabela 6.2 – Tempo (ms) para executar os métodos com DOM

	P001	P002	P003	P004
Recuperar	120	401	621	1943
Atualizar	20	80	90	381
Salvar	331	300	330	751

Tabela 6.3 – Tempo (ms) para executar os métodos com SAX

	P001	P002	P003	P004
Recuperar	90	261	411	1162
Atualizar*	0	0	0	0
Salvar	40	270	321	600

* tempo inferior a 1 milisegundo (ms)

O Gráfico 6.1 mostra o tempo gasto por cada API para recuperar o pedido dos arquivos P001, P002, P003 e P004. É possível perceber que a medida que o tamanho do arquivo aumenta a diferença de desempenho entre as APIs também aumenta. Para o

documento P004 (400 KB) a API SAX é, aproximadamente, 40% mais rápida do que a API DOM.

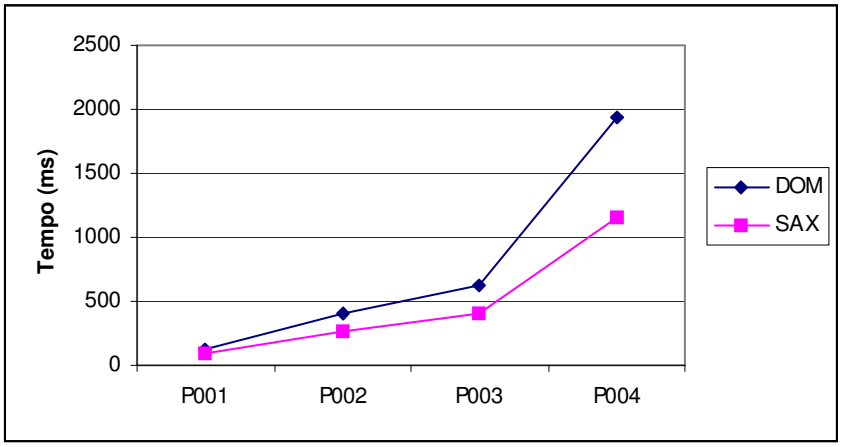


Gráfico 6.1 – Tempo consumido na execução do método Recuperar

Esta diferença acontece porque o método recuperar de DOM cria na memória a árvore para o documento e então recupera o pedido corrente. O método recuperar de SAX chama o `HandleSAX` que lê o documento e invoca o método `startElement()` para obter o pedido corrente.

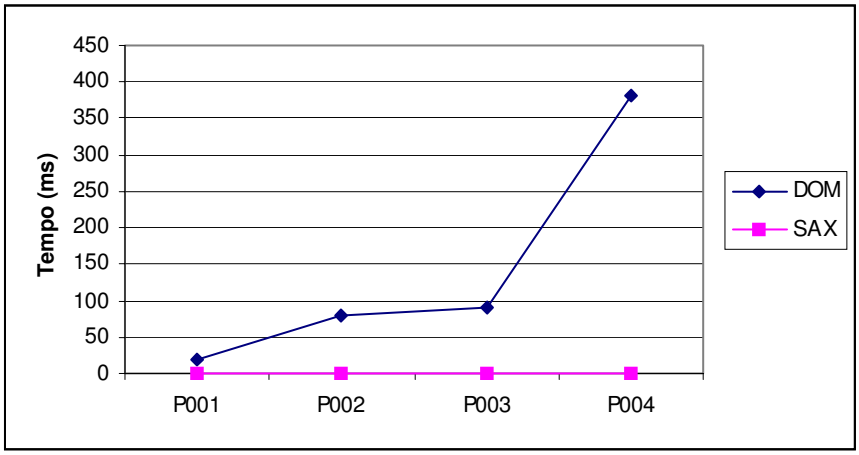


Gráfico 6.2 – Tempo consumido na execução do método Atualizar

O Gráfico 6.2 apresenta o desempenho das APIs para atualizar um pedido. Esse gráfico mostra que SAX é consideravelmente mais rápido do que DOM. Isto se deve ao fato de que o método atualizar de SAX apenas copia os valores dos atributos do objeto

parâmetro para o objeto do *cache*. A mesma função em DOM tem que acessar a árvore de dados para atualizar o pedido corrente.

O método `salvar` copia para o disco os dados de um pedido. Em DOM este método é implementado usando a interface `DOMWriter` definida em DOM3. Esta interface possui o método `writeNode()` que recebe como parâmetro um objeto `OutputStream` e um nó XML. Ele, então, copia o nó e todo o seu conteúdo para o fluxo de saída. Um problema com relação a este método é que atualmente poucas ferramentas de *parser* implementam o nível 3 de DOM.

A especificação SAX não define interface ou método para salvar documentos XML. O método salvar do repositório SAX foi implementado da seguinte maneira: os valores dos atributos dos objetos do *cache* foram gravados diretamente para um arquivo texto.

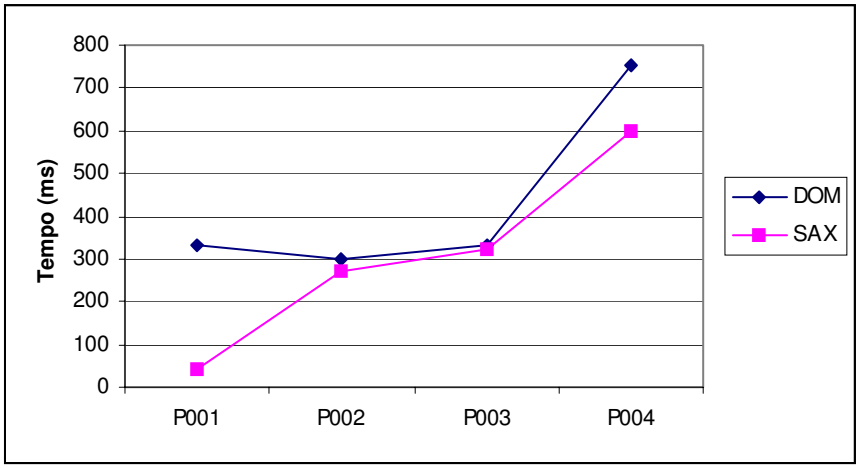


Gráfico 6.3 – Tempo consumido na execução do método Salvar

O Gráfico 6.3 mostra o desempenho dos métodos pra salvar os dados do *cache* para o arquivo XML nos repositórios DOM e SAX.

6.6 Considerações Finais

Este capítulo apresentou um estudo comparativo entre as APIs DOM e SAX. Inicialmente foram apresentadas as categorias de aplicações XML e em quais categorias podemos utilizar estas APIs. Logo após foram descritas as vantagens e desvantagens de cada padrão.

Foi apresentada, também, uma aplicação exemplo onde foi possível mostrar as características de cada API e realizar um conjunto de testes sobre o desempenho das mesmas. As principais diferenças entre as implementações DOM e SAX apresentadas no estudo de caso são:

- As estruturas de dados utilizadas por cada implementação. Em DOM utilizou-se a própria árvore de objetos, enquanto que em SAX utilizou-se objetos das classes base da aplicação;
- O método para recuperar um documento XML. Utilizando a API SAX, a complexidade desse método é proporcional à complexidade da estrutura do documento;
- O método para salvar um documento. Para implementar esse método com DOM, foi utilizado o método da especificação DOM3. Para o repositório SAX, este foi implementado copiando os valores dos atributos dos objetos para o arquivo XML já que SAX não define métodos para salvar documentos;
- Os métodos para atualizar e recuperar objetos no *cache*. Em DOM, é necessária uma classe que transforma os objetos em nós XML e vice-versa. Em SAX, isto não é necessário.

No próximo capítulo veremos um estudo sobre algumas ferramentas de *parser* que suportam as APIs DOM e SAX.

Capítulo 7 - Parsers

Neste capítulo é apresentado um estudo sobre *parsers* XML que suportam as APIs DOM e SAX. São mostradas características gerais comuns a todos os *parsers* e também características particulares dos *parsers*: JAXP, Xerces 2, XML for Java, Crimson e GNU JAXP, além da análise da performance destes *parsers*.

7.1 Introdução

Como dito anteriormente, um *parser* é um modulo de software que lê o documento XML, verifica se ele é bem formado e disponibiliza as informações para a aplicação. Ele atua entre os dados XML e a aplicação. O seu papel é transformar dados XML para um formato que outra aplicação possa utilizá-los. Esta comunicação entre o *parser* e a aplicação pode ser realizada através da interface orientada a objetos (DOM) ou da interface orientada a eventos (SAX).

Um *parser* XML Java é na verdade um conjunto de classes armazenadas em arquivos JAR (Java Archive). Os *parsers*, além de implementar as interfaces e classes definidas na especificação das APIs por eles suportadas, definem suas próprias classes que possibilitam criar instâncias de *parsers* para DOM e SAX. Estas classes recebem nomes e métodos que variam para cada parser. O mesmo acontece com os arquivos JAR: o número e o nome dos arquivos variam de um parser para outro.

Vários *parsers* podem ser instalados em uma mesma máquina. Para garantir que as classes utilizadas na aplicação sejam procuradas no arquivo JAR correto é necessário usar a variável CLASSPATH ou a chave `-classpath`.

No MSDOS é possível definir o caminho do arquivo JAR através do comando SET:

```
SET CLASSPATH=%CLASSPATH%;<parser_HOME>/<arquivo>.jar;
```

A chave `-classpath` pode ser usada com as ferramentas `javac` e `java`:

```
javac -classpath <parser_HOME>/<arquivo>.jar; <nome do programa>.java
java -classpath <parser_HOME>/<arquivo>.jar; <nome do programa>
```

Existem hoje disponíveis na Internet vários *parsers* XML implementados em Java. Para escolher um deles, alguns fatores devem ser considerados:

- Características – um *parser* pode ser de gratuito ou privado. Ele, além de verificar se um documento é bem formado, pode ou não ler DTDs e/ou XML *Schema*, suportar *namespaces*, validar documentos, etc;

- APIs – a maioria dos *parsers* suportam DOM e SAX [Harold 2002], mas nem todas as versões destas APIs são implementadas em todos os parsers; e
- Eficiência – quanto de recurso do sistema um *parser* consome para processar documentos.

Neste trabalho são analisados os *parsers*: JAXP [JAXP 2002b], Xerces 2 [Xerces 2002], XML for Java [XML4J 2002], Crimson [Crimson 2001] e GNU JAXP [GNU 2001]. Estes *parsers* foram escolhidos, pois são *parsers* comerciais amplamente usados e suportam muitos recursos das APIs DOM e SAX.

7.2 JAXP

Java APIs for XML Processing (JAXP) é uma tecnologia desenvolvida pela Sun como parte do Java XML Pack. Esta tecnologia processa dados XML usados em aplicações escritas em Java [Armstrong 2001].

A versão atual de JAXP é a 1.2. Ela conserva as características da versão 1.1 e acrescenta suporte a XML *Schema* [JAXP 2002a, JAXP 2002b]. Além deste padrão esta ferramenta suporta:

- XML 1.0 segunda edição;
- XML Namespace;
- DOM level 2 Core; e
- SAX 2.0 e SAX 2Extension.

Uma importante característica desta tecnologia é o fato dela permitir que documentos XML sejam analisados e/ou transformados usando uma API independente da implementação particular de um parser XML [JAXP 2002a].

As APIs JAXP que permitem esta independência são definidas no pacote `javax.xml.parsers`. Neste pacote é possível encontrar classes *factory* para criar *parser* DOM e *parser* SAX.

O JAXP 1.2 é constituído de seis arquivos JAR: jaxp-api.jar, sax.jar, dom.jar, xercesImpl.jar, xalan.jar, xsltc.jar[6]. No arquivo jaxp-api.jar encontra-se as classes dos pacotes javax.xml.parsers e javax.xml.transform, Nos arquivos dom.jar e sax.jar estão as

[6] Os arquivos xalan.jar e xsltc.jar não fazem parte do escopo deste trabalho.

implementações das classes e interfaces definidas nas especificações DOM e SAX, respectivamente. No arquivo xercesImpl.jar encontra-se as implementações do parser Xerces para os *parsers* DOM e SAX, e também para JAXP.

7.2.1 Parser DOM

JAXP define duas classes abstratas para criar um *parser* DOM: `DocumentBuilder` e `DocumentBuilderFactory`. A primeira representa o *parser*, enquanto que `DocumentBuilderFactory` define métodos para criar o objeto `DocumentBuilder` e configurá-lo como mostra a Listagem 7.1

Listagem 7.1 - Calsse DocumentBuilderFactory

```
public abstract class DocumentBuilderFactory {
    public abstract DocumentBuilder newDocumentBuilder() throws
        ParserConfigurationException
    public static DocumentBuilderFactory newInstance() throws
        FactoryConfigurationError
    public boolean isCoalescing()
    public void setCoalescing(boolean coalescing)
    public boolean isExpandEntityReferences()
    public void setExpandEntityReferences(boolean expandEntityRef)
    public boolean isIgnoringComments()
    public void setIgnoringComments(boolean ignoreComments)
    public boolean isIgnoringElementContentWhitespace()
    public void setIgnoringElementContentWhitespace(boolean whitespace)
    public boolean isNamespaceAware()
    public void setNamespaceAware(boolean awareness)
    public boolean isValidating()
    public void setValidating(boolean validating)
    public abstract Object getAttribute(String name)throws
        IllegalArgumentException
    public abstract void setAttribute(String name, Object value) throws
        IllegalArgumentException
}
```

Para criar um *parser* com JAXP é preciso criar um objeto `DocumentBuilderFactory`. A partir deste objeto criar uma instância de `DocumentBuilder` que é responsável por analisar o documento.

Um objeto `DocumentBuilderFactory` é criado através do seu próprio método `newInstance()`. Este método estático retorna uma nova instância de `DocumentBuilderFactory`:

```
DocumentBuilderFactory factory =
        DocumentBuilderFactory.newInstance();
```

Com este objeto é possível chamar o método `newDocumentBuilder()` que retorna uma nova instância de `DocumentBuilder`:

```
DocumentBuilder builder = factory.newDocumentBuilder();
```

Se o *parser* não puder ser criado com a configuração especificada pelo `DocumentBuilderFactory` uma exceção é gerada.

Além dos métodos citados, `DocumentBuilderFactory` especifica métodos para definir o comportamento do *paser*. Através dos métodos `set` é possível configurar o *parser* para validar documento, reconhecer *namespace*, tratar referências de entidade e seção CDATA e ignorar espaço em branco e comentário.

Estes métodos recebem como argumento um valor *boolean* que define o estado que a característica deve assumir, isto é, se o *parser* deve ou não reconhecer tal característica.

Os métodos cujo nomes estão no formato `is*()` retornam o valor (`true` ou `false`) que uma característica está assumindo.

Os dois métodos `coalescing` determinam se as seções CDATA presentes em um documento serão representadas como nós texto ou nós seção CDATA na árvore DOM. Se o valor desta característica for `true` as seções CDATA estarão presentes na árvore como nós texto, não contendo assim seções CDATA na árvore. O valor *default* para esta característica é `false`.

Os métodos `ExpandEntityReferences` definem se as referências de entidades devem ser expandidas ou não. O valor padrão para `ExpandEntityReferences` é `true`. Se um *parser* é validador, as referências à entidade serão expandidas mesmo que esta característica seja falsa.

`IgnoringComments` é a característica que diz se comentários farão parte da árvore. O valor padrão desta característica é `false`, o que significa que nós comentários estarão na árvore DOM.

Para ignorar espaços em branco entre as *tags* que normalmente fazem parte da árvore é preciso alterar o valor de `IgnoringElementContentWhitespace` para `true`, caso contrário estes espaços serão nós da árvore.

JAXP por *default* não trata os *namespaces* de um documento. Para que prefixos e *namespaces* URI sejam consideradas pelo *parser* é preciso atribuir o valor `true` à característica `NamespaceAware`.

A característica que define se um *parser* deve validar ou não um documento é `Validating`. Na maioria dos parsers esta característica tem valor *default* `false`. Para alterá-la, JAXP define o método `setValidating()` que deve receber como parâmetro o valor `true`. O código abaixo ilustra como definir um *parser* que valida documento e reconhece *namespace*:

```
DocumentBuilderFactory factory =
          DocumentBuilderFactory.newInstance();
factory.setValidating(true);
factory.setNamespaceAware (true);
DocumentBuilder builder = factory.newDocumentBuilder();
```

A característica `Validating` habilita o *parser* a validar documentos com DTD. Para validar um documento utilizando XML schema é preciso usar o método `setAttribute()` que recebe como parâmetro o nome da propriedade e seu objeto valor.

Depois de criar o objeto `DocumentBuilder` podemos utilizar seus métodos para analisar o documento, obter a implementação DOM usada, criar novos objetos `Document`, verificar se o *parser* pode validar documentos e fornecer dados sobre *namespace* e registrar tratamento de erro e de entidade. Os métodos desta classe são listados na Listagem 7.2.

Listagem 7.2 - Classe DocumentBuilder

```
public abstract class DocumentBuilder {
    public abstract DOMImplementation getDOMImplementation();
    public abstract boolean isNamespaceAware();
    public abstract boolean isValidating();
    public abstract Document newDocument();
    public Document parse(File) throws IOException, SAXException,
         IllegalArgumentException;
    public abstract Document parse(InputSource) throws IOException,SAXException,
         IllegalArgumentException;
    public Document parse(InputStream) throws IOException, SAXException,
         IllegalArgumentException;
    public Document parse(InputStream, String) throws IOException, SAXException,
         IllegalArgumentException;
    public Document parse(String) throws IOException, SAXException,
         IllegalArgumentException;
    public abstract void setEntityResolver(EntityResolver);
    public abstract void setErrorHandler(ErrorHandler);
}
```

`A classe DocumentBuilder` define cinco métodos `parse()`. Estes métodos recebem como entrada um documento XML, que pode ser representado como um objeto File, InputSource, InputStream ou String, criam a árvore DOM e retornam um objeto `Document`.

```
DocumentBuilder builder = factory.newDocumentBuilder();
Document document = builder.parse(xmlFile);
```

Para possibilitar que uma aplicação construa um novo documento XML, JAXP define o método `newDocument()` em `DocumentBuilder`. Esta função cria um novo objeto `Document` para construir a árvore DOM.

O método `setErrorHandler()` especifica o objeto `ErroHandler` que será usado para reportar os erros encontrados no documento que está sendo processado. Da mesma forma `setEntityResolver()` especifica o `EntityResolver` usado para tratar entidades externas.

Listagem 7.3 – Programa para validar documento com JAXP

```
import javax.xml.parsers.*;
import org.xml.sax.*;
import org.w3c.dom.*;
import java.io.*;
public class ValidarDocJAXP
{
	public static void main(String[] args) {
		if (args.length <=0) {
		System.out.println ("Use: java ValidarDocJAXP URL");
		return;
	}
	String xmlFile = args[0];
	try {
		OutputStreamWriter errorWriter = new
			OutputStreamWriter(System.err, "UTF-8");
		DocumentBuilderFactory dbf =
			DocumentBuilderFactory.newInstance();
		dbf.setNamespaceAware(true);
		dbf.setValidating(true);
		DocumentBuilder db = dbf.newDocumentBuilder();
		db.setErrorHandler(new MyErrorHandler(new PrintWriter(errorWriter,
		true)));
		Document doc = db.parse(xmlFile);
		System.out.println(xmlFile + " eh valido");

	} catch (SAXException e) {
		System.err.println(e.getMessage());
		System.exit(1); }
	catch (IOException e) {
		System.err.println(e);
		System.exit(1); }
	catch (FactoryConfigurationError e) {
		System.out.println("Could not locate a factory class"); }
	catch (ParserConfigurationException e) {
		System.out.println("Could not locate a JAXP parser");}
}
// Error handler para reportar erros e warnings
private static class MyErrorHandler implements ErrorHandler {
	...
 }
}
```

A Listagem 7.3 demonstra como usar JAXP para validar um documento usando as classes citadas e os seus métodos.

7.2.2 Parser SAX

As classes definidas por JAXP para criar um *parser* SAX são: `SAXParserFactory` e `SAXParser`. A classe `SAXParserFactory` , mostrada na Listagem 7.4, é uma classe *factory* que define, dentre outros, o método para criar o objeto `SAXParser`. Através deste objeto é possível obter um `XMLReader`.

Listagem 7.4 - Classe SAXParserFactory

```
public abstract class SAXParserFactory {
      public abstract boolean getFeature(String) throws
        ParserConfigurationException, SAXNotRecognizedException,
        SAXNotSupportedException
      public boolean isNamespaceAware()
      public boolean isValidating()
      public static SAXParserFactory newInstance() throws
         FactoryConfigurationError
      public abstract SAXParser newSAXParser() throws
        ParserConfigurationException, SAXException
      public abstract void setFeature(String, boolean) throws
        ParserConfigurationException, SAXNotRecognizedException,
        SAXNotSupportedException
      public void setNamespaceAware(boolean)
      public void setValidating(boolean)
}
```

O método `newInstance()` é o responsável por criar uma instância de `SAXParserFactory`, A partir dela o método `newSAXParser()` é invocado e retorna um novo objeto `SAXParser`.

Como acontece com o DOM, é possível atribuir algumas características ao *parser* SAX usando os métodos `set` definidos em `SAXParserFactory`. Para que o *parser* possa validar documentos e reconhecer *namespace*, utiliza-se os métodos `setNamespaceAware()` e `setValidating()` passando `true` como argumento. O código abaixo ilustra como criar um novo `SAXParser` de validação e que trata *namespace:*

```
SAXParserFactory spf = SAXParserFactory.newInstance();
spf.setValidating (true);
spf.setNamespaceAware (true);
SAXPar ser saxParser = spf.newSAXParser();
```

As demais características que um *parser* pode assumir são atribuídas através do método `setFeature()` que é equivalente ao método de mesmo nome definido na interface

`XMLReader`. Diferente de DOM, a especificação SAX define a interface `XMLReader` para representar o *parser*. Sendo assim as características podem ser atribuídas ao *parser* através de métodos desta interface.

Listagem 7.5 - Métodos da classe SAXParser para SAX2

```
public abstract XMLReader getXMLReader() throws SAXException;
public abstract Object getProperty(String name) throws SAXNotRecognizedException,
    SAXNotSupportedException;
public abstract boolean isValidating();
public void parse(InputStream is, DefaultHandler dh) throws SAXException,
    IOException;
public void parse(InputStream is, DefaultHandler dh, java.lang.String systemId)
    throws SAXException, IOException;
public void parse(String uri, DefaultHandler dh) throws SAXException,
    IOException;
public void parse(File f, DefaultHandler dh) throws SAXException, IOException;
public void parse(InputSource is, DefaultHandler dh) throws SAXException,
    IOException;
public abstract void setProperty(String name, Object value) throws
    SAXNotRecognizedException, SAXNotSupportedException;
```

Depois de criar o objeto `SAXParser` pode-se utilizar os métodos desta classe para analisar documentos. A classe `SAXParser` define os métodos mostrados na Listagem 7.5 para SAX2.

Listagem 7.6 – Exemplo1 – usando as classes SAXParserFactory e SAXParser

```
import org.xml.sax.*;
import java.io.IOException;
import org.xml.sax.helpers.DefaultHandler;
import javax.xml.parsers.*;
public class ImprimirElem extends DefaultHandler{
      public static void main(String[] args)  {
            try {
                  SAXParserFactory spf = SAXParserFactory.newInstance();
                  SAXParser sp = spf.newSAXParser();
                  XMLReader parser = sp.getXMLReader();
                  parser.setContentHandler(new ImprimirElem());
                  parser.parse(args[0]);
            }
            catch (SAXException e) {
                  System.err.println(e);
      ...
}
```

O método `getXMLReader()` retorna um objeto `XMLReader`. Com este objeto é possível utilizar as funções definidas na interface que ele representa, a `XMLReader`. A Listagem 7.6 demonstra como usar essa função.

A classe `SAXParser` também define métodos `parse()` que analisam documentos. Estes métodos devem receber com argumento o documento XML e um objeto `DefaultHandler`. A Listagem 7.7 ilustra o uso de um desses métodos.

Listagem 7.7 – Exemplo 2 – usando as classes SAXParserFactory e SAXParser

```
import org.xml.sax.*;
import java.io.IOException;
import org.xml.sax.helpers.DefaultHandler;
import javax.xml.parsers.*;

public class ImprimirElem extends DefaultHandler{
       public static void main(String[] args)  {
              ...
              try {
                     SAXParserFactory spf = SAXParserFactory.newInstance();
                     SAXParser sp = spf.newSAXParser();
                     sp.parse(args[0], new ImprimirElem());
              }
              catch (SAXException e) {
                     System.err.println(e);
              ...
}
```

7.3 Parsers Apache

Por uma razão histórica, a Apache Software Foundation mantém hoje três parsers Java diferentes: Crimson [Crimson 2001], XML4J [XML4J 2002] e Xerces 2 [Xerces 2002]. No dia 9 de novembro de 1999 a Apache anunciou a criação do projeto de `xml.apache.org` para soluções *Open Source XML* [XML4J 2002]. Com isto ela recebeu duas doações de empresa diferentes. A IBM doou o XML for Java e a Sun doou seu Project X que se tornou Apache Crimson. O Xerces 2 é um novo parser, reescrito com base nos parsers doados [JAXP 2002b].

7.3.1 Xerces 2

Xerces 2 é a nova geração de *parsers* XML da Apache Xerces [Xerces 2002]. A sua versão mais recente é 2.2.0.

Este é o parser validador mais completo no que diz respeito aos padrões implementados. Xerces suporta os seguintes padrões e APIs [Xerces 2002]:

- Recomendação XML 1.0 segunda edição
- XML Namespace

- DOM level 2 Core, Events, e Traversal and Range
- SAX 2.0 Core, e Extension
- JAXP 1.1
- XML Schema 1.0 Structures and Datatypes
- Além de implementar parcialmente os módulos Core, Load and Save e Abstract Schema de DOM level 3.

As implementações destes recursos na versão 2.2.0 de Xerces são mantidas nos arquivos: `XercesImpl.jar`, `XercesSample.jar` e `xmlParserAPIs.jar`.

7.3.1.1 A Classe DOMParser

Em Xerces, a classe que representa o *parser* DOM é chamada de `DOMParser` mostrada na Listagem 7.8. Esta classe é definida no pacote `org.apache.xerces.parsers`. Através dela é possível analisar documentos, obter e alterar o estado das características e propriedade do *parser*, bem como especificar tratador de erros e de entidades externas.

Listagem 7.8 - Classe DOMParser

```
public class DOMParser {
    public EntityResolver getEntityResolver();
    public ErroHandler getErrorHandler();
    public boolean getFeature (String feature) throws SAXNotRecognizedException,
        SAXNotSupportedException;
    public Object getProperty (String property) throws SAXNotRecognizedException,
        SAXNotSupportedException;
    public void parse (InputSource inputSource) throws SAXException,IOException;
    public void parse (String systemId); throws SAXException, IOException;
    public void setEntityresolver (EntityResolver resolver);
    public void setErrorHandler (ErrorHandler error);
    public     void     setFeature     (String     feature,     boolean     state)
        throwsSAXNotRecognizedException, SAXNotSupportedException;
    public   void   setProperty   (String   property,   Object   value)   throws
        SAXNotRecognizedException, SAXNotSupportedException;
}
```

Para analisar um documento, primeiro é preciso criar um objeto `DOMParser`, chamando o construtor da classe:

```
DOMParser parser = new DOMParser();
```

Com este objeto é possível invocar o método `parse()` para analisar o documento e criar a árvore DOM.

`DOMParser` define dois métodos `parse()`, sendo que `parse (String systemId)` é equivalente a `parse(new InputSource(systemId))`. Ambos os métodos não têm valor de

retorno. Para obter a raiz da árvore, o *parser* chama o método `getDocument()`. Este método é definido na classe `AbstractDOMParser`, da qual `DOMParser` é derivada.

```
DOMParser parser = new DOMParser();
parser.parse("teste.xml");
Document doc = parser.getDocument();
```

O *parser* criado pelo construtor de `DOMParser` define os seguintes valores *default* para as características (feature) padrões descritas no Capítulo 5.

Feature	**Default**
http://xml.org/sax/features/namespaces	True
http://xml.org/sax/features/validation	False
http://xml.org/sax/features/external-general-entities	True
http://xml.org/sax/features/external-parameter-entities	True
http://xml.org/sax/features/string-interning	não reconhece
http://xml.org/sax/features/namespace-prefix	não reconhece

Tabela 7.1 - Características específicas de Xerces

Feature	Default
http://apache.org/xml/features/validation/dynamic	False
http://apache.org/xml/features/validation/schema	False
http://apache.org/xml/features/validation/schema-full-checking	False
http://apache.org/xml/features/validation/schema/normalized-value	True
http://apache.org/xml/features/validation/schema/element-default	True
http://apache.org/xml/features/validation/schema/augment-psvi	True
http://apache.org/xml/features/validation/warn-on-duplicate-attdef	False
http://apache.org/xml/features/validation/warn-on-undeclared-elemdef	False
http://apache.org/xml/features/warn-on-duplicate-entitydef	False
http://apache.org/xml/features/allow-java-encodings	False
http://apache.org/xml/features/continue-after-fatal-error	False
http://apache.org/xml/features/nonvalidating/load-dtd-grammar	True
http://apache.org/xml/features/nonvalidating/load-external-dtd	True
http://apache.org/xml/features/scanner/notify-char-refs	False
http://apache.org/xml/features/scanner/notify-builtin-refs	False
http://apache.org/xml/features/dom/defer-node-expansion*	true
http://apache.org/xml/features/dom/create-entity-ref-nodes*	True
http://apache.org/xml/features/dom/include-ignorable-whitespace*	True

* Características específicas do parser DOM

Para alterar o valor de um característica, `DOMParser` implementa o método `setFeature()`. Este método recebe dois argumentos: uma *string* com nome da característica, e um *boolean* que indica o novo valor que ela deve assumir. Para verificar que estado (`true` ou `false`) de uma característica é usada a função `getFeature()`. Além

das características padrões, Xerces também define as características que são mostradas na Tabela 7.1.

Da mesma forma, Xerces implementa os métodos `setProperty()` e `getProperty()` para especificar um propriedade e obter o objeto desta propriedade, respectivamente. Além das propriedades padrões mostradas no Capítulo 5, este parser ainda implementa as propriedades listadas na Tabela 7.2.

Tabela 7.2 - Propriedades específicas de Xerces

Property	Type
http://apache.org/xml/properties/schema/external-schemaLocation	String
http://apache.org/xml/properties/schema/external-noNamespaceSchemaLocation	String
http://apache.org/xml/properties/input-buffer-size	Integer
http://apache.org/xml/properties/dom/current-element-node *	Element
http://apache.org/xml/properties/dom/document-class-name *	DocumentImpl

* Propriedades específicas para Parser DOM

7.3.1.2 Criando um Parser

Além da classe `DOMParser`, Xerces permite outras duas opções para criar um parser DOM: JAXP, como foi mostrado na seção anterior, e DOM 3 Load and Save.

Para criar um *parser* com DOM3 é preciso obter um objeto da interface `DOMImplementationLS`, e através deste criar um objeto `DOMBuilder` que é responsável por construir a árvore DOM.

Xerces define no pacote `org.apache.xerces.dom` a classe `DOMImplementationSourceImpl`. Esta classe implementa a interface `DOMImplementationSource` de DOM3. Através de um objeto desta interface é possível obter uma instância de `DOMImplementationLS` usando o método `getDOMImplementation(String features)`:

```
import org.apache.xerces.dom.DOMImplementationSourceImpl;
...

DOMImplementationSourceImpl souce = new
      DOMImplementationSourceImpl();
 DOMImplementationLS impl =
      (DOMImplementationLS)souce.getDOMImplementation("LS-Load");
```

A interface `DOMImplementationLS` define o método `createDOMBuilder()` para instanciar um objeto da interface `DOMBuilder`. Este método recebe como argumento um inteiro *short* que identifica se o objeto criado vai trabalhar de forma síncrona ou assíncrona

(`MODE_SYNCHRONOUS` ou `MODE_ASYNCHRONOUS`), e uma *string* que representa a linguagem de esquema usada para carregar o documento (este valor pode ser null). Estas interfaces são implementadas no pacote `org.w3c.dom.ls`.

```
DOMBuilder builder =
impl.createDOMBuilder(DOMImplementationLS.MODE_SYNCHRONOUS, null);
```

Com o objeto `DOMBuilder` é possível configurar o parser usando o método `setFeature()` e analisar o documento:

```
builder.setFeature("http://xml.org/sax/features/validation",true);
Document doc = builder.parseURI(document);
```

Listagem 7.9 – Exemplo DOM3 usando DOMImplementationSourceImpl

```
import org.w3c.dom.*;
import java.io.IOException;
import org.w3c.dom.ls.*;
import org.apache.xerces.dom.*;

public class DOM3 {
  public static void main(String[] args) {
    String document = teste.xml;
    try {
       DOMImplementationSourceImpl souce = new
                   DOMImplementationSourceImpl();
        DOMImplementationLS impl =
              (DOMImplementationLS)souce.getDOMImplementation("LS-Load");
        // create DOMBuilder
        DOMBuilder builder = impl.createDOMBuilder(
                  DOMImplementationLS.MODE_SYNCHRONOUS, null);
        // parse document
        System.out.println("Parsing "+ document +"...");
        Document doc = builder.parseURI(document);
         System.out.println(document + " is well-formed.");
    }
    catch (ClassCastException e) {
      System.err.println("The current DOM implementation does"
       + " not support DOM Level 3 Load and Save"); }
    catch (DOMException e) {
       System.err.println(document + " is not well-formed"); }
     catch (Exception e) {
       e.printStackTrace();
       System.exit(1); }
  }
}
```

A Listagem 7.9 mostra como verificar se um documento é bem formado usando a classe e interfaces citadas.

Xerces também permite criar um *parser* usando a classe `DOMImplementationRegistry`. Com um objeto desta classe é possível obter uma implementação DOM e então criar um objeto `DOMBuilder` para analisar documento.

Entretanto, para usá-la é preciso especificar uma propriedade do sistema que recebe como argumentos o campo `PROPERTY` de `DOMImplementationRegistry` e a classe `DOMImplementationSourceImpl`, como ilustra a Listagem 7.10.

Listagem 7.10 – Exemplo de DOM3 usando DOMImplementationRegistry

```
import   org.apache.xerces.dom3.DOMImplementationRegistry;
import  org.w3c.dom.Document;
import  org.w3c.dom.ls.DOMImplementationLS;
import  org.w3c.dom.ls.DOMBuilder;
...
         // Obtém a implementação DOM usando o registro DOM
         System.setProperty(DOMImplementationRegistry.PROPERTY,
                  "org.apache.xerces.dom.DOMImplementationSourceImpl");
         DOMImplementationRegistry registry =
                  DOMImplementationRegistry.newInstance();

         DOMImplementationLS impl =
         (DOMImplementationLS)registry.getDOMImplementation("LS-Load");
         // Cria o objeto DOMBuider
         DOMBuilder builder = impl.createDOMBuilder(
                  DOMImplementationLS.MODE_SYNCHRONOUS, null);
         Document document = builder.parseURI("teste.xml");
...
}
```

7.3.1.3 A Classe SAXParser

Para criar *parser* SAX, Xerces define a classe `SAXParser` no pacote `org.apache.xerces.parsers`. O construtor desta classe cria um objeto `XMLReader`. Com este objeto é possível analisar documentos e utilizar todos os métodos especificados na interface `XMLReader`.

A classe `SAXParser` pode ser usada de duas formas. É possível chamar diretamente o seu construtor:

```
XMLReader parser = new SAXParser();
```

Ou passá-la como parâmetro do método `XMLReaderFactory.createXMLReader`:

```
XMLReader parser = XMLReaderFactory.createXMLReader(
        "org.apache.xerces.parsers.SAXParser");
```

O objeto padrão de `SAXParser` possui como valor os seguintes valores *default* para as características (feature) padrões descritas no Capítulo 5:

Feature	Default
http://xml.org/sax/features/namespaces	True
http://xml.org/sax/features/validation	False
http://xml.org/sax/features/external-general-entities	True

http://xml.org/sax/features/external-parameter-entities	True
http://xml.org/sax/features/string-interning	True
http://xml.org/sax/features/namespace-prefix	False

Além de `SAXParser`, é possível definir um *parser* SAX em Xerces usando JAXP, como foi mostrado na Seção 7.2.

7.3.2 XML for Java (XML4J)

A versão mais recente de XML4J é a 4.0.5, lançada em setembro de 2002. Esta versão é baseada na versão 2.0 do Apache Xerces. Muitas das características do Xerces 2 estão presentes também neste parser [XML4J 2002].

XML Parser for Java é um parser validador que, na sua mais recente versão, incorpora suporte a [XML4J 2002]:

- W3C XML Schema Recommendation
- SAX 1.0 and 2.0
- DOM Level 1, DOM Level 2, e algumas características experimentais de DOM Level 3 Core, Abstract Schema, e Load/Save
- JAXP 1.1

XML4J define os mesmos arquivos JAR que Xerces 2: XercesImpl.jar, xmlParserAPIs.jar e XercesSamples.jar.

7.3.2.1 Parser DOM

Assim como Xerces, é possível definir um *parser* DOM em XML4J de três formas. Este parser suporta a versão 1.1 de JAXP, então as classes definidas nesta tecnologia podem ser usadas para criar *parser* DOM. Outra opção é usar a classe `DOMParser` implementada no pacote `org.apache.xerces.parsers` que funciona exatamente como descrito na seção sobre *parser* DOM Xerces, ou ainda usar o módulo Load and Save de DOM3.

As interfaces e classes da especificação DOM 3 em XML4J são implementadas nos pacotes `org.apache.xerces.dom3`, `org.apache.xerces.dom3.as` e `org.apache.xerces.dom3.ls`. Então, para criar um *parser* é preciso importar os pacotes corretos para XML4J.

Neste parser as classes `DOMImplementationLS` e `DOMBuilder` são implementadas no pacote `org.apache.xerces.dom3.ls`. Enquanto que `DOMImplementationSourceImpl` se encontra no pacote `org.apache.xerces.dom` e `DOMImplementationRegistry` no pacote `org.apache.xerces.dom3`.

A Listagem 7.11 mostra como verificar se um documento é bem formado usando DOM3 em XML4J. Este parser implementa o método `createDOMBuilder()` para recebe apenas o parâmetro *short* (`MODE_SYNCHRONOUS` ou `MODE_ASYNCHRONOUS`).

Listagem 7.11 – Exemplo de DOM3 com XML4J

```
import org.w3c.dom.*;
import org.apache.xerces.dom3.ls.*;
import java.io.IOException;
import org.apache.xerces.dom.*;

public class DOM3Checker {
  public static void main(String[] args) {
    String document = args[0];
    try {
      DOMImplementationLS impl =
        (DOMImplementationLS)DOMImplementationImpl.getDOMImplementation();
      DOMBuilder parser =
             impl.createDOMBuilder(DOMImplementationLS.MODE_SYNCHRONOUS);
      Document doc = parser.parseURI(document);
      System.out.println(document + " is well-formed.");
    }
    catch (ClassCastException e) {
    ...
  }
}
```

O parser SAX pode ser criado usando exatamente os mesmos recursos descritos para definir um parser deste tipo em Xerces.

7.3.3 Crimson

Crimson é um parser Java mantido pela Apache cujo código é baseado no parser Sun, Project X. A sua versão mais recente, 1.1.3, foi lançada em outubro de 2001. Ele foi incorporado a ferramenta SDK1.4 [SDK 2002].

Crimosn é um *parser* de validação que implementa as seguintes APIs:

- JAXP 1.1
- SAX 2.0
- SAX2 Extension versão 1.0

- DOM Level 2 Core

As implementações de interfaces e classes referentes a estes padrões estão no arquivo `crimson.jar`. Caso a versão do JDK utilizada não seja a 1.4, é necessário colocar este arquivo no classpath:

```
javac –classpath <crimson_HOME>/crimson.jar; CrimsonTeste.java
java –classpath <crimson_HOME>/crimson.jar; CrimsonTeste
```

Crimson não define uma classe própria para criar parser DOM. Um parser desta API é criado usando as classes `DocumentBuilderFactory` e `DocumentBuilder` de JAXP 1.1, da mesma forma que foi mostrado na Seção 7.2:

```
DocumentBuilderFactory factory = DocumentBuilderFactory.newInstance();
DocumentBuilder parser = factory.newDocumentBuilder();
Document doc = parser.parse(doc);
```

Um parser SAX em Crimson pode ser criado de duas maneiras: usando as classes definidas em JAXP, mostradas na seção sobre JAXP, ou usando a classes `XMLReaderImpl` que implementa a interface `XMLReader` de SAX.

Para usar a classe específica deste parser pode-se chamar diretamente o construtor desta classe:

```
XMLReader parser = new XMLReaderImpl();
```

ou usar o método `createXMLReader` e passar esta classes como parâmetro:

```
XMLReader parser = XMLReaderFactory.createXMLReader(
        "org.apache.crimson.parser.XMLReaderImpl");
```

Um parser definido com a classes XMLReaderImpl possui os seguintes valores *default* para as propriedades padrões de SAX:

Feature	**Default**
http://xml.org/sax/features/namespaces	true
http://xml.org/sax/features/validation	false
http://xml.org/sax/features/external-general-entities	true
http://xml.org/sax/features/external-parameter-entities	true
http://xml.org/sax/features/string-interning	true
http://xml.org/sax/features/namespace-prefix	false

7.4 GNU JAXP

GNU JAXP é um parser gratuito que faz parte do projeto GNU Classpath Extensions. A sua versão 1.0 beta1 suporta os seguintes padrões [GNU 2001]:

- SAX2
- DOM level 2 Core e Event
- JAXP 1.1

As classes que GNU define para implementar estes padrões estão agrupadas no arquivo `gnujaxp.jar`. Este parser não define classe própria para construir *parser* DOM. Entretanto, um *parser* dessa API é instanciado através das classes `DocumentBuilderFactory` e `DocumentBuilder` definidas por JAXP. Como mostrado na seção 7.2 é preciso criar um objeto `DocumentBuilderFactory` chamando o seu próprio método `newInstance()`, e através dele obter um objeto `DocumentBuilder`.

Listagem 7.12 – Exemplo DOM com GNU

```
import javax.xml.parsers.DocumentBuilder; //JAXP
import javax.xml.parsers.DocumentBuilderFactory; //JAXP
import org.w3c.dom.Document;
import org.xml.sax.SAXException;
import java.io.*;
...
        String doc = "teste.xml";
         try {
              FileInputStream fi = new FileInputStream(doc);

              DocumentBuilderFactory factory =
                    DocumentBuilderFactory.newInstance();
              DocumentBuilder parser = factory.newDocumentBuilder();
              Document document = parser.parse(fi);
         }
       ...
}
```

A Listagem 7.12 mostra um trecho de código onde um *parser* DOM é criado para analisar um documento XML.

Um *parser* SAX, por sua vez, pode ser criado com JAXP, ou usando as classes `SAXDriver` ou `XmlReader` presentes no pacote `gnu.xml.aelfred2`.

`A classe SAXDriver` cria *parser* que não podem validar documentos. É possível definir um objeto desta classe chamando o seu construtor:

```
XMLReader parser = SAXDriver();
```

ou chamando o método `createXMLReader()`:

```
XMLReader parser = XMLReaderFactory.createXMLReader(
        "gnu.xml.aelfred2.SAXDriver");
```

A classe `XmlReader` define *parser* de validação. A Listagem 7.13 mostra como criar um objeto desta classe e usá-lo para analisar um documento.

Listagem 7.13 – Exemplo SAX com GNU

```
import org.xml.sax.*;
import java.io.IOException;
import java.io.FileInputStream;
import gnu.xml.aelfred2.*;
...
    try {
      InputSource src=new InputSource (new FileInputStream("teste.xml"));
      XMLReader parser = new XmlReader();
      parser.parse(src);
       ...
    }
    catch (SAXException e) {
...
}
```

7.5 Desempenho dos Parsers

Nesta seção são mostrados os resultados de alguns testes realizados com os *parsers* citados nas seções anteriores: JAXP, Xerces, XML4J, Crimson e GNU. Estes testes foram realizados em uma máquina Intel Celeron 1.06GHz com 248MB de memória RAM com Windows XP.

Para avaliar o desempenho dos *parsers* foi medida a quantidade de tempo e espaço de memória consumido por cada um para executar as seguintes aplicações com DOM e com SAX: 1) verificar a boa formação de documentos XML; 2) acessar todos os elementos do documento; 3) validar documentos com DTD; e 4) validar documentos com XML *Schema*.

Os testes de cada *parsers* foram realizados da seguinte forma:

1. Foram selecionados documentos XML com tamanhos variados: pequeno, médio e grande. A lista dos documentos é mostrada na Tablea 7.3;

Tabela 7.3 – Documentos XML usados nos testes dos *parsers*

Nome	Tamanho (KB)	Nº elementos
V10	10	78
V100	100	846
V500	500	4351
V1000	1.000	8709
V5000	5.000	43421

2. Para medir o tempo consumido pelas aplicações foi usado o método `System.currentTimeMillis()`. Para medir a quantidade de memória foram utilizados os métodos `Runtime.totalMemory()` e `Runtime.freeMemory()`. A unidade de tempo utilizada foi milisegundos (ms) e a unidade de memória foi KBytes (KB). A Listagem 7.14 mostra como esses métodos foram usados.

Listagem 7.14 – Medindo tempo e a quantidade de memória utilizadas nas aplicações

```
public class BemFormado {

    Runtime rt = Runtime.getRuntime();
    long startMem = rt.totalMemory() - rt.freeMemory();

    long startTime = System.currentTimeMillis();
    ...

    try {
      ...

      long endMem = rt.totalMemory() - rt.freeMemory();
      long endTime = System.currentTimeMillis();

      log.write ("Tempo de processamento: " + (endTime - startTime)/100
                    + " milisegundos.");
      log.write ("Memoria usada: " + (endMem - startMem)/1024 + " KB.");
    } catch (SAXException e) {
      ...
}
```

3. Para cada documento, as aplicações foram executados dez vezes;
4. Os resultados selecionados foram o tempo e a quantidade de memória[7] que ocorreram com maior freqüência. Esses resultados foram organizados em tabelas e gráficos; e
5. Para cada documentos, foi calculado a média de tempo gasto pelos *parsers* para executar cada aplicação. Essa média serve de parâmetro para avaliar o desempenho dos *parsers*: os resultados menores ou iguais a média são considerados bons, os resultados maiores que a média são considerados ruins.

[7] O espaço de memória ocupado por um *parser* para determinado documento XML se manteve constante em todas as execuções.

7.5.1 Desempenho dos *Parsers* para Verificar Boa Formação

A Tabela 7.4 mostra os resultados obtidos na execução do programa que verifica se um documento é bem formado. Com base nos dados obtidos, é possível observar que o *parser* mais rápido é o Crimson. Mas, como mostra o Gráfico 7.1, o seu desempenho cai à medida que o tamanho do arquivo XML aumenta.

Tabela 7.4 – Resultados do programa para verificar boa formação dos documentos

		Tempo (ms)				
	API	**V10**	**V100**	**V500**	**V1000**	**V5000**
Crimson	DOM	1	3	6	10	40
	SAX	2	3	4	5	14
JAXP	DOM	5	7	10	13	34
	SAX	5	6	8	10	21
XML4J	DOM	5	6	10	13	36
	SAX	5	6	8	9	21
GNU	DOM	3	5	16	45	998
	SAX	1	2	3	5	16
Xerces	DOM	6	7	9	12	32
	SAX	5	6	8	10	21
Tempo	DOM	4	5.6	10,2	18,6	35.5*
Médio	SAX	3.6	4.6	6.2	7.8	18.6

Memória (KB)				
V10	**V100**	**V500**	**V1000**	**V5000**
180	742	3776	7504	37531
475	480	211	376	416
270	637	2487	4687	22577
437	428	181	380	359
172	554	2435	4629	23075
226	140	405	604	583
288	1155	4557	11761	56740
52	268	225	325	58
366	710	2388	4763	22615
490	472	225	424	403

* Não foi utilizado o tempo do GNU para calcular esse resultado

Os *parsers* JAXP, XML4J e Xerces obtiveram resultados bem próximos, mas os seus desempenhos só são considerados bons para arquivos maiores que 1MB, justamente onde Crimson obteve os piores resultados.

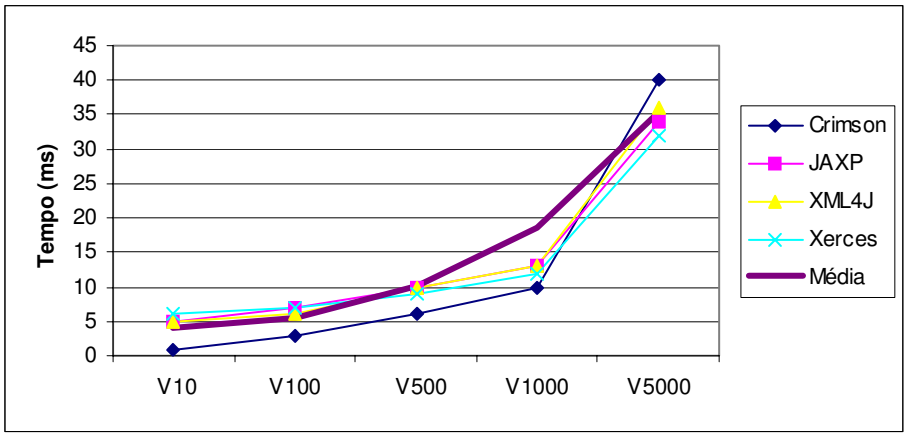


Gráfico 7.1 – Tempo para analisar documentos com DOM

Os resultados obtidos com o GNU são omitidos do Gráfico 6.1 pois eles são muito maiores que os resultados dos demais *parsers*. O que deixa claro que o GNU não é indicado para aplicações que usam DOM.

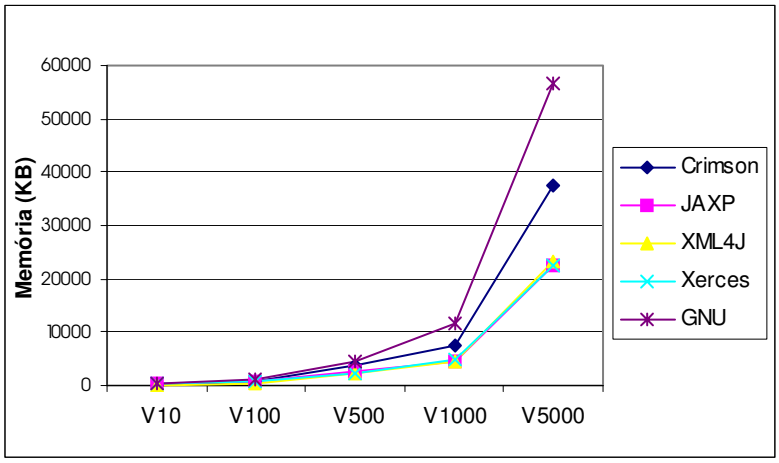


Gráfico 7.2 – Memória para analisar documentos com DOM

O Gráfico 7.2 mostra que, apesar de mais rápido, Crimson consome mais espaço de memória para construir a árvore DOM do que os parsers JAXP, XML4J e Xerces. Esse gráfico mostra também que à medida que o tamanho dos arquivos aumenta a diferença de desempenho entre os *parsers* também aumenta. Para o maior arquivo (V5000) o GNU consome aproximadamente 50% a mais de memória que o Crimson. Esse por sua vez consome 60% a mais de memória que o JAXP, que apresenta o menor consumo.

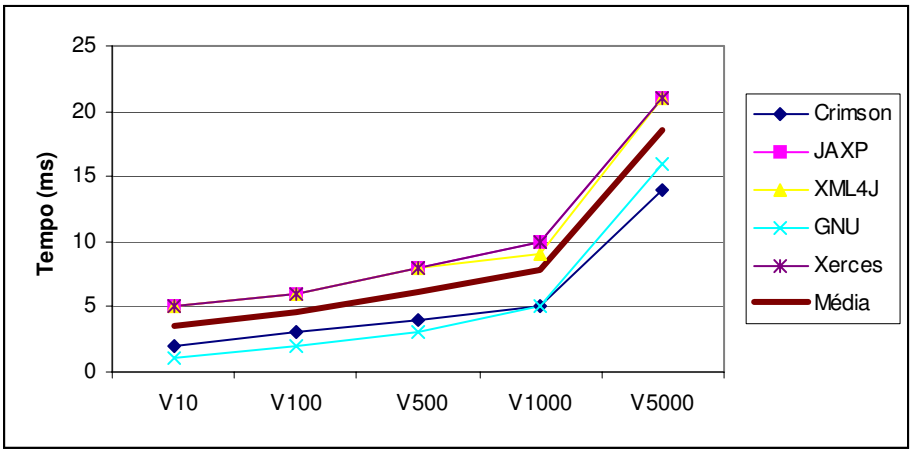


Gráfico 7.3 – Tempo para analisar documento com SAX

O Gráfico 7.3 mostra o tempo que os *parsers* gastam para analisar os documentos usando SAX. Mais uma vez o Crimson obteve bons resultados. O GNU também obteve um bom desempenho, chegando algumas vezes a ser mais rápido que o Crimson. Os *parsers* JAXP, XML4J e Xerces tiveram resultados ruins, isto é, acima da média de tempo.

7.5.2 Desempenho dos *Parsers* para Acessar Elementos

A Tabela 7.5 mostra os resultados obtidos com os experimentos que acessam todos os elementos dos documentos. Comparando esta tabela com a Tabela 7.4 é possível notar que há um pequeno aumento no tempo médio dos *parsers*. Os responsáveis por esta alteração são JAXP, XML4J e Xerces. Os outros *parsers* mantiveram o mesmo desempenho .

Tabela 7.5 – Resultados do programa para acessar os elemento dos documentos

		Tempo (ms)				
	API	**V10**	**V100**	**V500**	**V1000**	**V5000**
Crimson	DOM	1	3	6	10	40
	SAX	2	3	4	5	14
JAXP	DOM	5	7	10	14	39
	SAX	5	6	8	10	22
XML4J	DOM	5	7	11	14	39
	SAX	5	6	8	10	22
GNU	DOM	3	5	16	45	969
	SAX	1	2	3	4	14
Xerces	DOM	6	8	10	14	37
	SAX	5	6	8	10	21
Tempo	DOM	4	6	10.6	19.4	38.75*
Médio	SAX	3.6	4.6	6.2	7.8	18.6

Memória (KB)				
V10	**V100**	**V500**	**V1000**	**V5000**
180	742	3776	7504	37531
475	482	213	378	417
364	763	2741	5269	25929
447	443	195	395	373
200	649	2682	5280	25942
254	244	509	196	175
287	1154	4556	11760	56738
458	261	137	237	482
406	810	2762	5317	25970
510	497	249	448	427

* Não foi utilizado o tempo do GNU para calcular esse resultado

7.5.3 Desempenho dos *Parsers* para Validar Documentos

Um documento XML pode ser validado com uma DTD ou com um esquema. Todos os *parsers* testados nesse trabalho permitem validar documento com DTD, mas apenas JAXP, XML4J e Xerces têm suporte para XML *Schema*.

A Tabela 7.6 mostra os resultados obtidos para validar documentos com DTD. Comparando esses valores com os valores da Tabela 7.4, é possível perceber que não há variação do consumo de memória entre elas. Entretanto, os *parsers* gastam um pouco mais

de tempo para validar os documentos, pois além de verificar se o documento é bem formado, o *parser* deve verificar se o documento esta de acordo com as regas definidas na DTD.

Tabela 7.6 – Resultados da validação de documento com DTD

		Tempo (ms)				
	API	**V10**	**V100**	**V500**	**V1000**	**V5000**
Crimson	DOM	2	4	8	12	44
	SAX	2	3	4	6	16
JAXP	DOM	6	8	11	16	45
	SAX	5	7	9	11	26
XML4J	DOM	6	8	11	15	41
	SAX	5	6	8	10	25
GNU	DOM	3	6	17	46	955
	SAX	2	3	5	7	21
Xerces	DOM	7	9	12	16	43
	SAX	5	7	9	11	25
Tempo	DOM	4.8	7	11.8	21	43.25*
Médio	SAX	3.8	5.2	7	9	22.6

Memória (KB)				
V10	**V100**	**V500**	**V1000**	**V5000**
198	926	3833	7604	37355
479	486	217	382	422
535	631	2745	5270	24608
514	536	624	872	2019
317	676	2677	5201	25013
293	326	477	728	2436
365	918	4430	11488	62881
117	389	543	471	357
271	690	2806	5175	24657
123	547	638	885	2336

* Não foi utilizado o tempo do GNU para calcular esse resultado

7.5.3.1 Validação com XML Schema

A Tabela 7.7 mostra os resultados da validação dos documentos utilizando XML *Schema*. Como pode ser observado, a validação usando este padrão é mais lenta do que usando DTD, chegando, em alguns casos, a gastar o dobro do tempo.

Tabela 7.7 – Resultado da validação de documentos com XML Schema

		Tempo (ms)				
	API	**V10**	**V100**	**V500**	**V1000**	**V5000**
JAXP	DOM	12	16	23	29	74
	SAX	12	15	20	23	52
XML4J	DOM	10	13	20	26	67
	SAX	10	13	17	21	46
Xerces	DOM	11	14	19	24	55
	SAX	11	14	17	19	38
Tempo	DOM	11	14.3	20.7	26.3	65.3
Médio	SAX	11	14	18	21	45.3

Memória (KB)				
V10	**V100**	**V500**	**V1000**	**V5000**
414	949	2754	5537	27466
525	540	321	266	544
341	881	3094	5713	26905
532	551	332	277	555
487	775	2822	5032	23401
252	523	492	311	238

O processo de validação com XML *Schema* é mais lento do que com DTD por um conjunto de razões:

- A especificação de XML Schema é maior do que a de DTD. Portanto, as ferramentas que implementam essa especificação tendem a ser mais complexas ocupando mais tempo de processamento;
- XML Schema possui um conjunto de tipos maior do que DTD;
- As gramáticas XML Schema são maiores que as de DTD, por isto ocupam mais espaço para armazenamento na memória principal.

Por outro lado, XML *Schema* possui um conjunto de vantagens:

- Possui recursos poderosos de tipificação de dados;
- Suporte a *namespace*;
- Permite especificar a cardinalidade com maior precisão: o usuário pode definir tanto a cardinalidade máxima quanto a mínima, através de um número inteiro. Em DTD isto não é possível.

O Gráfico 7.4 apresenta uma comparação entre a validação de documentos utilizando DTD e utilizando XML *Schema*. Os testes foram realizados utilizando a API DOM e o parser Xerces. Como pode ser observado nesse gráfico a validação com DTD é cerca de 37% mais rápida que a validação com XML *Schema* com pouca variação por conta do tamanho do documento utilizado.

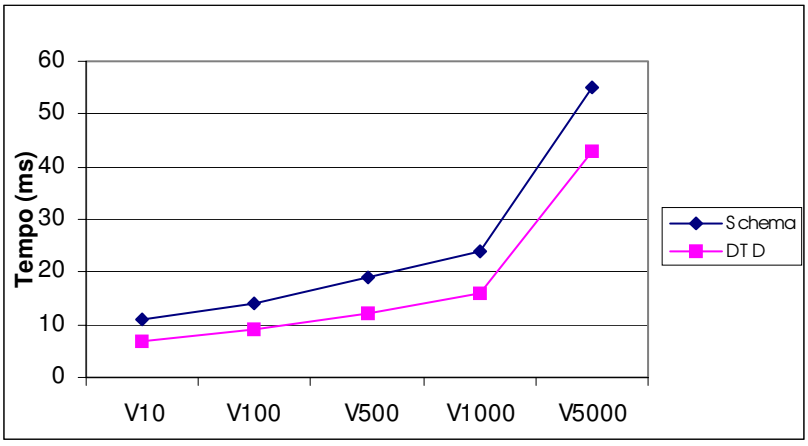


Gráfico 7.4 – Validação com DTD x validação com XML Schema

7.6 Quadro de Decisão

Como foi dito anteriormente, a escolha do *parser* depende dos requisitos da aplicação. A Tabela 7.8 apresenta algumas informações que podem auxiliar os projetistas na escolha do *parser* mais adequado à sua aplicação.

Tabela 7.8 – Tabela de auxílio na escolha do *parser*

Requisitos	JAXP	Xerces 2	XML4J	Crimson	GNU
Maior rapidez utilizando SAX				X	X
Melhor desempenho com DOM para processar documento maior que 1MB	X	X	X		
Melhor desempenho com DOM para processar documento menor que 1MB				X	
Reconhecer *namespace*	X	X	X	X	X
Validar com DTD	X	X	X	X	X
Validar com XML *Schema*	X	X	X		
Suportar eventos com DOM					X
Salvar documentos usando DOM		X	X		
Necessidade de fornecer o *parser* junto com a aplicação				X	X

Para as aplicações que utilizam a interface SAX os *parsers* que tiveram melhor desempenho foram o Crimson e o GNU. Já as aplicações que utilizam a interface DOM, os *parsers* JAXP, XML4J e Xerces2 mostraram melhor desempenho para documentos maiores que 1MB. O Crimson mostrou melhor desempenho processando documentos menores que 1MB.

Algumas características como suporte a *namespace* e validação com DTD foram encontrada em todos os *parsers* analisados. Entretanto, a validação com XML *Schema* é possível através dos *parsers* JAXP, XML4J e Xerces2.

Uma outra característica importante é o tamanho dos *parsers*. Alguns *parsers* são maiores que outros. Se a aplicação precisa ser distribuída para vários usuários, será mais conveniente utilizar um *parser* pequeno ou um *parser* que já está incluído no SDK. Entre os *parsers* analisados o Crimson e o GNU são os menores. O Crimson já está incluído na versão 1.4 do SDK.

7.7 Como Melhorar a Performance do Parser

Os *parsers* possuem desempenhos diferentes como pôde ser visto nas seções anteriores. No entanto, algumas medidas podem ajudar a melhorar a performance de um *parser*:

- Usar o mesmo *parser* para analisar vários documentos, evitando assim criar um novo *parser* cada vez que um documento XML for analisado;

- Configurar o *parser* apenas com as características que são necessárias. Por exemplo, se os documentos não precisam ser válidos evite configurar o *parser* com a característica `Validation`;
- Evitar o uso de entidades e DTDs externas;
- Evitar o uso excessivo de espaços em branco, pois normalmente o *parser* irá analisá-los;
- Manter o documento o menor possível. Documentos menores são processados mais rapidamente; e
- Evitar atributos definidos como *default* em DTD ou XML *Schema*.

7.8 Considerações Finais

Uma aplicação XML necessita de um software para analisar documentos. Neste capítulo foi mostrado com criar *parser* DOM e *parser* SAX para manipular documentos XML usando os parsers JAXP, Xerces, XML4J, Crimson e GNU.

Estes *parsers* possuem características próprias: JAXP define classes *factory* para criar *parser* DOM e *parser* SAX independentes do parser. Xerces e XML4J são os mais completos com relação aos padrões implementados.

Apesar das ferramentas citadas implementarem basicamente os mesmos padrões, há algumas variações quanto ao tempo e espaço de memória consumidos por cada uma para criar *parsers* DOM e SAX. O *parser* com melhor desempenho de tempo é o Crimson, mas consome mais memória que a maioria dos *parsers* testados. Ele suporta SAX1 e SAX2, mas não implementa a maioria dos módulos de DOM2 e não tem suporte para DOM3. O GNU mostrou o pior desempenho para *parser* DOM, mas é um dos mais rápidos para *parser* SAX.

Capítulo 8 - Conclusões

Neste capítulo são feitas as considerações finais sobre o trabalho, são apresentadas as principais contribuições, trazidas com os estudos realizados durante este trabalho, e os possíveis trabalhos futuros.

8.1 Considerações Finais

XML é uma meta linguagem criada para atender às necessidades de aplicações Web. Um documento XML pode armazenar dados e, por ser um arquivo texto, possibilita a troca de informação entre diferentes sistemas.

Para que os dados presentes em um documento sejam disponibilizados para uma aplicação, dois padrões de manipulação de documentos XML foram criados: DOM e SAX.

DOM é uma API orientada a objetos que foi projetada pelo W3C. Este consórcio mantém atualmente três níveis de DOM: DOM1, DOM2 e DOM3. A cada nível foram acrescidas novas características para atender as constantes necessidades das aplicações. As especificações dessa API são grandes, e possuem interfaces e métodos muitas vezes redundantes, como pode ser observado nos Capítulos 3 e 4.

SAX é um padrão baseado em um paradigma que não é muito familiar a muitos programadores, o paradigma orientado a eventos. No entanto, esta API é simples e pode ser utilizada em diversas aplicações.

SAX gera eventos para quase todos os itens encontrados em um documento XML. Ele foi projetado para informar a aplicação apenas sobre os dados presentes no documento. Inicialmente, os itens léxicos, que muitas vezes não são significativos para a aplicação, eram ignorados. Na sua segunda versão várias alterações foram feitas: SAX passou a suportar *namespace*, DTDs, e reconhecer alguns conteúdos léxicos. Outro problema ainda presente em SAX é a falta de métodos que possibilitem alterar e salvar documentos na memória.

Apesar destes problemas, em muitas aplicações, SAX obtém um desempenho melhor do que DOM, como pode ser observado nos testes realizados no Capítulo 6.

Cada uma dessas APIs possui vantagens e desvantagens. Os requisitos de uma aplicação é que vão definir a melhor API a ser utilizada. Em algumas situações elas podem ser usadas em conjunto para obter um melhor resultado.

Além de escolher a API, é importante para o desempenho do sistema a escolha da ferramenta de processamento mais adequada para sua aplicação. Estes software podem

implementar ambas as APIs, ou apenas uma, ou ainda implementar parcialmente alguns níveis de DOM e de SAX.

Neste trabalho foi realizado um estudo descritivo e comparativo entre os *parsers*: JAXP, Xerces2, XML4J, Crimson e GNU. Cada *parser* define suas próprias classes para criar *parser* DOM e *parser* SAX. Os nomes e os métodos destas classes variam de uma ferramenta para outra. Isto pode causar certas dificuldades como, por exemplo, ter que recompilar o sistema em caso de mudança da ferramenta de processamento. Para resolver esta questão, a Sun definiu em sua ferramenta JAXP classes *factory* para criar *parser* DOM e *parser* SAX que são independentes do *parser*. Todos os software citados no trabalho suportam JAXP.

Todas as ferramentas estudadas podem validar documentos com DTD e suportam *namespace*, mas apenas JAXP, Xerces e XML4J validam utilizando XML *Schema*.

Elas implementam as versões 1 e 2 de SAX, mas nenhum suporta completamente todos os módulos de DOM2 e DOM3. Entretanto, muitas aplicações utilizam apenas o modulo Core de DOM.

Quanto ao desempenho dos *parsers*, é importante observar o tempo de processamento de cada um e também a quantidade de memória consumida por eles. O Crimson, por exemplo, é a ferramenta mais rápida para criar a árvore DOM, mas ocupa mais memória do que os outros *parsers* testados. GNU mostra um bom desempenho com *parser* SAX. As ferramentas JAXP, Xerces e XML4J implementam um conjunto maior de padrões e possuem desempenhos equivalentes.

8.2 Contribuições

As principais contribuições deste trabalho são:

- Análise das características das APIs de processamento de documento XML: DOM e SAX. Essas duas APIs pertencem a dois paradigmas diferentes, um orientado a objeto e o outro orientado a eventos. Em determinadas situações é mais adequado utilizar uma ou outra.
- Análise comparativa das APIs DOM e SAX, apresentando uma implementação de uma aplicação real usando essas APIs e comparando seus desempenhos.

- Análise das características dos parsers: JAXP, Xerces2, XML4J, Crimson e GNU. Cada aplicação possui um conjunto de necessidades quanto ao uso do XML. Algumas não necessitam realizar as validações dos documentos, outras precisam validar os seus documentos com DTDs apenas, outras precisam de XML *Schema*, enquanto que outras necessitam de suporte a *namespace*. Todas essas características podem ser encontradas totalmente ou parcialmente nos parsers que estão no mercado. Alguns parsers possuem características que outros não possuem: saber se um determinado parser atende a todas as necessidades de uma determinada aplicação é muito importante no processo de escolha. Apesar de serem encontrados alguns trabalhos realizando comparações entre *parsers* XML [Harold 2002], estes avaliam o desempenho de versões anteriores. Como os parsers estão em constante desenvolvimento, estes trabalhos ficaram desatualizados, além do que, não reunem os mesmos *parsers* mostrados neste trabalho.
- Análise do desempenho dos parsers. A forma com que os parsers são implementados tem impacto direto na performance dos mesmos. Numa aplicação XML, os parsers são muito utilizados, pois os dados XML são disponibilizados por eles. Desta maneira a performance do parser será um ponto crítico para a performance da aplicação como um todo. Um estudo comparativo sobre as performances dos parsers pode auxiliar o gerente de projetos a escolher o parser que venha oferecer melhor performance para a sua aplicação.

8.3 Trabalhos Futuros

Como trabalhos futuros podem ser destacados:

- Analisar aplicações reais onde módulos específicos de DOM2 e DOM3 são ou podem ser utilizados;
- Desenvolver uma ferramenta de suporte aos módulos de DOM3 e alguns módulos de DOM2 ainda não suportados pelos parsers analisados;

- Analisar o desempenho dos parsers com validação de documentos com gramáticas maiores.
- Analisar outros parsers de documentos XML, como por exemplo, Oracle XML Parser e Piccolo XML Parser for Java.

Referências Bibliográficas

[Armstrong 2001] ARMSTRONG, Eric. at al. *Workin whit XML The Java API for XML Processing (JAXP) Tutorial* – 10 Dec 2001. Disponível em: http://java.sun.com/xml/jaxp/dist/1.1/docs/tutorial/index.html. Último acesso em 24/04/2003.

[Bosak 1997] BOSAK, Jon. *XML, Java and the future of the Web* – 10/03/1997. Disponível em: http://www.ibiblio.org/pub/sun-info/standards/xml/why/xmlapps.htm. Último acesso em 24/04/2003.

[Bourret 2000] BOURRET, R. P. *XML and Data Base* - 2000. Disponível em: http://www.rpbourret.com/xml/XMLAndDatabases.htm. Último acesso em 24/04/2003.

[Bray 2000] BRAY, Tim et al. *W3C Recommendation XML* - 06/10/2000. Disponível em: http://www.w3.org/TR/REC-xml. Último acesso em 24/04/2003.

[Byrne 1998] BYRNE, Steve et al. *Document Object Model (DOM) Level 1 Specification Version 1.0* - 01/10/1998. Disponível em: http://www.w3.org/TR/REC-DOM-Level-1/. Último acesso em 24/04/2003.

[Chang 2002] CHANG, Ben; LITANI, Elena. *Document Object Model (DOM) Level 3 Abstract Schemas and Load and Save Specification* - 09 April 2002. Disponível em: http://www.w3.org/TR/DOM-Level-3-ASLS/. Último acesso em 24/04/2003.

[Clark 2001] CLARK, Kendall Grant. *DOM and SAX Are Dead, Long Live DOM and SAX* – 14 Novembro, 2001. Disponível em: http://www.xml.com/lpt/a/2001/11/14/dom-sax.html. Último acesso em 24/04/2003.

[Crimson 2001] *Crimson 1.1 Release* – Last modified 3 Oct, 2001. Disponível em: http://xml.apache.org/crimson/index.html. Último acesso em 24/04/2003.

[Deitel 2001] DEITEL, H. M. e DEITEL, P. J. *Java How Program* - 3.ed. Prentice

Hall, Inc., 2000.

[Fesler 2001] FESLER, Stephanie. *Using DOM to Traverse XML* – 08 February 2001. Disponível em: http://www.onjava.com/lpt/a//onjava/2001/02/08/dom.html. Último acesso em 24/04/2003.

[Franklin] FRANKLIN, Steve. *XML Parsers: DOM and SAX Put to the Test*. Disponível em: http://www.devx.com/xml/articles/sf020101/sf0201-1.asp. Último acesso em 24/04/2003.

[GNU 2001] *The GNU JAXP Project* – Last modified 11/24/2002. Disponível em: http://www.gnu.org/software/classpathx/jaxp/. Último acesso em 24/04/2003.

[Harold 2002] HAROLD, Eliotte Rusty. *Processing XML with Java: A Guide to SAX, DOM, JDOM, JAXP, and TrAX* – Edition Paperback, 2002.

[Hégaret 2002] HÉGARET Philippe Le. *Document Object Model (DOM) Activity Statement* – Last modified 17/06/2002. Disponível em: http://www.w3c.org/DOM/Activity. Último acesso em 24/04/2003.

[Holzner 2001] HOLZNER, Steven. *Desvendando XML* – Rio de Janeiro: Campus , 2001.

[Hors 2000a] HORS, Arnaud Le. HÉGARET, Philippe Le. et al. *Document Object Model (DOM) Level 2 Core Specification version 1.0* – November, 2000 http://www.w3.org/TR/DOM-Level-2-Core/. Último acesso em 24/04/2003.

[Hors 2000b] HORS, Arnaud Le, CABLE, Laurence. *Document Object Model (DOM) Level 2 Views Specification* - November, 2000 http://www.w3.org/TR/DOM-Level-2-Views/. Último acesso em 24/04/2003.

[Hors 2002] HORS, Arnaud Le; HÉGARET, Philippe Le et al. *Document Object Model (DOM) Level 3 Core Specification Version 1.0* - 09 April 2002. Disponível em: http://www.w3.org/TR/DOM-Level-3-Core/. Último acesso em 24/04/2003.

[Hustead 2000] HUSTEAD, Robert. *Mapping XML to Java, Part 1* – 08/2000.

disponível em: http://www.javaworld.com/javaworld/jw-08-2000/jw-0804-sax.html

[Idris 1999] IDRIS, Nazmul. *Should I use SAX or DOM?* – 23 May 1999. Disponível em: http://developerlife.com/saxvsdom/default.htm. Último acesso em 25/04/2003.

[JAXP 2002a] *Java™ API for XML Processing (JAXP) Frequently Asked Questions* – Last updated 24 Oct, 2002. Disponível em: http://java.sun.com/xml/jaxp/faq.html Último acesso em 25/04/2003.

[JAXP 2002b] Java™ API for XML Processing README.

[Kesselman 2000] KESSELMAN, Jeo. Robie, Jonathan. Champion, Mike. *Document Object Model (DOM) Level 2 Traversal and Range Specification* – November, 2000. Disponível em: http://www.w3.org/TR/DOM-Level-2-Traversal-Range/. Último acesso em 24/04/2003.

[Light 1999] LIGHT, Richard. *Iniciando em XML* – São Paulo: Makron Books, 1999.

[Marchal 2000] MARCHAL, Benoît. *XML conceitos e aplicações* – São Paulo: Berkeley Brasil, 2000.

[Martin 2000] MARTIN, Didier et al. *Professional XML* – Rio de Janeiro: Editora Ciência Moderna Ltda., 2001

[McLaughlin 2001] MCLAUGHLIN, Brett. *Just over the horizon... a new DOM-A preview of DOM Level 3* – August 2001. Disponível em: http://www-106.ibm.com/developerworks/xml/library/x-dom3.html?dwzone=xml. Último acesso em 25/04/2003.

[Means 2001a] MEANS, W. Scott. *Processing Large XML Documents Using SAX 2.0* – 9 May 2001. Disponível em: http://xml.oreilly.com/news/xmlnut_0501.html. Último acesso em 01/04/2002.

[Means 2001b] MEANS, W. Scott. *What´s New in the DOM Level 2 Core?* – 11 February, 2001. Disponível em: http://xml.oreilly.com/news/xmlnut_0201.html. Último acesso em 25/04/2003.

[Megginson 2001] MEGGINSON, David et al. *SAX 1.0: The Simple API for XML* – Última versão 12/11/2001. Disponível em: http://www.saxproject.org/. Último acesso em 20/02/2003.

[Mohseni] MOHSENI, Piroz. *Choose Your Java XML Parser* – Disponível em: http://archive.devx.com/xml/articles/pm020101/pm020101-1.asp. Último acesso em 25/04/2003.

[Oracle] *Oracle XML Developer's Kit for Java* – Disponível em: http://technet.oracle.com/tech/xml/xdk_java/content.html. Último acesso em 25/04/2003.

[Oracle 2002] *Tutorial Series: Oracle XML Parser Techniques* – October, 2002. Disponível em: http://otn.oracle.com/sample_code/tutorials/parser/parsertoc.htm. Último acesso em 25/04/2003.

[Pitts-Moultis 2000] PITTS-MOULTIS, Natanya / KIRK, Cheryl *XML Black Book* - São Paulo: MAKRON Book, 2000.

[Pixley 2000] PIXLEY, Tom. *Document Object Model (DOM) Level 2 Events Specification* – November, 2000 http://www.w3.org/TR/DOM-Level-2-Events/. Último acesso em 24/04/2003.

[Pixley 2002] PIXLEY, Tom. *Document Object Model (DOM) Level 3 Events Specification* - 08 February 2002. Disponível em: http://www.w3.org/TR/DOM-Level-3-Events/. Último acesso em 24/04/2003.

[SDK 2002] *Java 2 SDK Platform, Standard Edition (J2SE)* – 2002. Disponível em: http://java.sun.com/j2se/1.4/. Último acesso em 14/04/2003.

[ÚRI 2000] *Uniform Resource Identifier (URI) Activity Statement* – July, 2000. Disponível em: http://www.w3.org/Addressing/Activity. Último acesso em 24/04/2003.

[Xerces 2002] *Xerces2 Java Parser Readme* Disponível em: http://xml.apache.org/xerces2-j/index.html. Último acesso em 25/04/2003.

[XercesC 2002] *Xerces C++ Parser* Disponível em: http://xml.apache.org/xerces-

c/index.html. Último acesso em 25/04/2003.

[XercesP 2002] *Xerces Perl Parser* Disponível em: http://xml.apache.org/xerces-p/index.html. Último acesso em 25/04/2003.

[XML4J 2002] *XML Parser for Java* – Disponível em http://www.alphaworks.ibm.com/tech/xml4j. Último acesso em 25/04/2003.

[Whitmer 2000] WHITMER, Ray. *Document Object Model (DOM) Level 3 Views and Formatting Specification* - 15 November 2000. Disponível em: http://www.w3.org/TR/DOM-Level-3-Views/. Último acesso em 24/04/2003.

[Wilson 2000] WILSON, Chris. HÉGARET Philippe Le. APPARAO, Vidur. *Document Object Model (DOM) Level 2 Style Specification* – November, 2000 http://www.w3.org/TR/DOM-Level-2-Style/. Último acesso em 24/04/2003.

[Wood 2000] WOOD, Lauren; HORS, Arnaud Le et al. *Document Object Model (DOM) Level 1 Specification (Second Edition)* - 29 September, 2000. Disponível em: http://www.w3.org/TR/2000/WD-DOM-Level-1-20000929/. Último acesso em 24/04/2003.

[Young 2000] YOUNG, Michael. *XML Step by Step* – Redmond: Microsoft Press, 2000.