UNIVERSIDADE FEDERAL DE PELOTAS
INSTITUTO DE FÍSICA E MATEMÁTICA
CURSO DE BACHARELADO EM INFORMÁTICA

Uma Ferramenta Para Integração de Esquemas XML Utilizando Ontologias

por

LUÍS ANDRÉ MARTINS

Trabalho de Conclusão de Curso

Prof. Sérgio Medeiros Santi, UCPEL
Orientador

Prof. Maurício Nunes Porto, UFPEL
Co-orientador

Pelotas, dezembro de 2000.

Sumário

Lista de Figuras **4**
Lista de Abreviaturas **5**
Resumo **6**
Abstract **7**
1 Introdução **8**
2 A Linguagem XML **10**
2.1 Origem e Objetivos **11**
2.2 Documentos XML **11**
2.2.1 Elementos 12
2.2.2 Referências 13
2.2.3 Seções CDATA 14
2.2.4 Instruções de Processamento 14
2.2.5 DTD (*Document Type Definition*) 14
2.2.6 Comentários 15
2.3 Documentos XML Bem Formados **15**
2.4 Documentos XML Válidos **16**
2.4.1 Declarações de Elementos 17
2.4.2 Declaração de Atributos 18
2.4.3 Declaração de Referências a Entidades 20
2.4.4 Declarações de Notações 20
2.5 Processadores XML **21**
2.5.1 SAX – *Simple API for XML* 22
2.5.2 DOM – *Document Object Model* 24
2.6 Linguagens Relativas a XML **27**
2.6.1 XML Schema 27
2.6.2 XSL – *Extensible Stylesheet Language* 27
2.6.3 XLink – *XML Linking Language* 28
2.6.4 *Namespaces* 28
2.6.5 RDF – *Resource Description Framework* 29
3 Ontologias **30**
3.1 Tipos de Ontologias **32**
3.2 Ontologias e XML **33**
3.2.1 Acessando Documentos XML com Ontologias 33
3.2.2 Representando Ontologias com XML 34
4 Integração de Esquemas com Ontologias **36**
4.1 Integração de Esquemas **36**
4.1.1 Causas da diversidade entre esquemas 38
4.1.2 O processo de integração 39
4.2 O Projeto IBM-UFRGS **40**
4.2.1 Esquemas locais 41
4.2.2 Esquema global 42
4.2.3 Integração de DTDs utilizando ontologias 42
4.3 Ontologias e Integração **43**
4.3.1 O formato da ontologia 44
4.3.2 O elemento *Ontology* 45
4.3.3 O elemento *Schema* 46
4.3.4 O elemento *Concept* 46
4.3.5 O elemento *Relantionship* 46
4.3.6 O elemento *Origin* 47
4.3.7 O *Thesaurus* 48
4.3.8 Sub-ontologias 48

4.3.9 Ontologias locais 51
5 Especificação do Protótipo JOntology 56
5.1 Requisitos do Software 56
5.2 Visão Geral da Especificação 57
5.3 Representação das Ontologias 58
5.3.1 A classe *Concept* 61
5.3.2 A classe *Relation* 61
5.3.3 A classe *Origin* 62
5.3.4 A classe *Schema* 63
5.3.5 A classe *SubOntology* 64
5.3.6 A classe *LocalOntology* 64
5.3.7 A classe *Ontology* 65
5.4 Construção e Integração das Ontologias 66
5.4.1 A construção de sub-ontologias 66
5.4.2 A construção de Ontologias Locais 69
5.4.3 O processo de integração 70
5.4.4 A classe *Thesaurus* 71
5.5 Visualização e Edição da Ontologia 72
5.5.1 Criação do Grafo a partir da ontologia 73
5.5.2 Edição da ontologia através do grafo 74
5.6 Interface do Programa JOntology 77
5.6.1 Criando uma nova sub-ontologia 78
5.6.2 Integrando esquemas e criando ontologias locais 79
5.6.3 Visualização e edição do *thesaurus* 79
5.6.4 Abrindo e salvando arquivos XML 80
6 Conclusões, Contribuições e Trabalhos Futuros 81
Anexo 1 83
7 Glossário 85
8 Bibliografia 87

Lista de Figuras

FIGURA 2.1 -Exemplo de *start-tag* e *end-tag* (a) e atributos num elemento (b) 12
FIGURA 2.2 - Exemplo de um documento XML bem formado 13
FIGURA 2.3 - Exemplo de uma DTD para o documento da fig. 2.2..... 17
FIGURA 2.4 – Exemplo de Declaração de Entidade..... 20
FIGURA 2.5 – Processamento de um documento XML 22
FIGURA 2.6 – Exemplo de um Documento XML para uso no DOM 25
FIGURA 2.7 – Exemplo de modelo DOM 26
FIGURA 3.1 – Exemplo de conceitos numa ontologia..... 31
FIGURA 3.2 – Exemplo de um documento XOL..... 35
FIGURA 4.1 - Exemplo de dois esquemas heterogêneos 37
FIGURA 4.2 - Resultado da integração dos esquemas da fig. 4.1 37
FIGURA 4.3- Arquitetura geral do projeto IBM-UFRGS 40
FIGURA 4.4 - Um esquema local integrando diferentes sub-esquemas..... 41
FIGURA 4.5 - Esquema global integrando diferentes esquemas locais 42
FIGURA 4.6 - Visão geral do processo de integração..... 43
FIGURA 4.7 - Modelo de dados do formato das ontologias 44
FIGURA 4.8 - DTD com o formato da ontologia para integração..... 45
FIGURA 4.9 - Exemplo de relacionamento numa ontologia..... 47
FIGURA 4.9 - DTD para o *Thesaurus* das ontologias..... 48
FIGURA 4.7 - DTDs sendo mapeadas para sub-ontologias 49
FIGURA 4.8 - Algoritmos para construir sub-ontologias..... 50
FIGURA 4.8 - Exemplo de geração de ontologias locais a partir de sub-ontologias..... 52
FIGURA 4.9 - Algoritmo para construção de ontologias locais 53
FIGURA 4.10 - Exemplo de conflito de cardinalidades numa integração de esquemas..... 54
FIGURA 4.11 - Exemplo de um conceito e suas origens relacionadas 54
FIGURA 5.1 - Diagrama de pacotes do programa..... 57
FIGURA 5.2 - Diagrama de classes da representação da Ontologia..... 59
FIGURA 5.3 - Diagrama de classes da ontologia associadas com as classes DOM..... 60
FIGURA 5.4 - Especificação da classe *Concept*..... 61
FIGURA 5.5 - Especificação da classe *Relation* 62
FIGURA 5.6 - Especificação da classe *Origin* 62
FIGURA 5.7- Especificação da classe *Schema*..... 63
FIGURA 5.8 - Especificação da classe *SubOntology* 64
FIGURA 5.9 - Especificação da classe *LocalOntology* 65
FIGURA 5.10 - Especificação da classe *Ontology* 65
FIGURA 5.11 - Classes utilizadas na construção de uma sub-ontologia..... 66
FIGURA 5.12 - Exemplo de uso do método *getParseTree()*..... 67
FIGURA 5.13 - Diagrama de seqüência mostrando a criação de uma sub-ontologia..... 68
FIGURA 5.14 - Diagrama de seqüência mostrando a criação de uma ontologia local..... 69
FIGURA 5.15 - Especificação da classe *Thesaurus* 71
FIGURA 5.16 - Diagrama de classes utilizadas para a representação dos grafos..... 72
FIGURA 5.17 - Diagrama de seqüência com a criação de um grafo 73
FIGURA 5.18 - Exemplo de um grafo gerado no JOntology a partir de uma sub-ontologia..... 74
FIGURA 5.19 - Diagrama de classes com as associações entre ontologia e grafo 75
FIGURA 5.20 - Janela de diálogo de um vértice do grafo..... 76
FIGURA 5.21 - Janela principal do protótipo JOntology 77
FIGURA 5.22 - Criação de uma nova sub-ontologia..... 78
FIGURA 5.23 - Criação de uma nova ontologia local..... 79
FIGURA 5.24 - Janela para visualização do *thesaurus* 80

Lista de Abreviaturas

API	Aplication Program Interface
DOM	Document Object Model
DTD	Document Type Definition
fig.	figura
GFC4Java	Graph Foundation Classes for Java
HTML	Hyper Text Markup Language
ID	identificação
IEC	Intenational Electrotechnical Commission
ISO	International Organization of Standarization
lin.	linha
p. ex.	por exemplo
PROCERGS	Cia. de Processamento de Dados do Estado do Rio Grande do Sul
RDF	Resource Description Framework
SAX	Simple API for XML
SGBD	Sistema de Gerenciamento de Bancos de Dados
SGML	Standard Generalized Markup Language
SQL	Structured Query Language
UML	Unified Modelling Language
UFPEL	Universidade Federal de Pelotas
UFRGS	Universidade Federal do Rio Grande do Sul
URI	Uniform Resource Identification
URL	Uniform Resource Locator
W3C	World Wide Web Consortium
WWW	World Wide Web
XLink	XML Linking Language
XML	Extensible Markup Language
XOL	XML-Based Ontology Language
XSL	Extensible Stylesheet Language

Resumo

Este trabalho de conclusão de curso tem como objetivo apresentar um sistema de integração de esquemas XML, ou DTDs mais especificamente, utilizando como suporte para tal as ontologias. Serão apresentadas as motivações e requisitos para a criação de sistema.

Serão apresentadas também as principais características da linguagem XML e das ontologias, tendo em vista a importância destes dois tópicos no entendimento do trabalho.

Palavras-chave: integração, esquemas, XML, DTD, ontologia.

Abstract

The main goal of this graduation work is to present a XML schemes, or DTD's more precisely, integration system, using ontologies as support. Its presented the motivations and requirements for the system creation.

Its also presented the XML language and ontologies main features, since this two topics are essential in the understanding of the this work.

Keywords: integration, schema, XML, DTD, ontology.

1 INTRODUÇÃO

A linguagem XML foi definida pelo W3C em 1998 tendo como principal objetivo construir uma linguagem de marcação, ou *markup*, que fosse enxuta e padronizada, para ser usada pelas comunidades da *web*. O seu uso já extrapolou os domínios da Internet, sendo utilizada em toda uma gama de aplicações, como distribuição e integração de informações ou padronização de formatos de dados, entre outras coisas. A principal característica da XML é ser uma metalinguagem, ou seja, uma linguagem para definição de linguagens de marcação. Esta característica permite que linguagens definidas em XML possam ser utilizadas na troca de informações que estão estruturadas de acordo com o significado do seu conteúdo, entre diversas aplicações, em diferentes formatos e para diferentes comunidades [STE 00, PIM 00].

Por outro lado, um problema crescente, sobretudo com o avanço da *web*, é a distribuição homogênea de informações. As empresas atualmente tem o interesse em distribuir os seus dados, para agilizar os seus processos. Todavia, os sistemas de informação existentes atualmente possuem diferenças entre si no que diz respeito aos seus dados e modelos de processos, e isto é uma barreira para qualquer tentativa de compartilhar informações. No caso dos bancos de dados, qualquer tentativa de distribuir informações contidas em fontes de dados heterogêneas esbarra no problema da integração dos esquemas destas fontes, tendo em vista que o principio básico dos bancos de dados é permitir uma representação unificada e não-redundante de todos os dados necessários a uma organização. Por essa razão, diversos trabalhos e metodologias estão sendo desenvolvidas para a integração e inter-operação entre bancos de dados [BON 94, BAT 86].

Um caso mais especifico da aplicabilidade de integração de esquemas é o projeto IBM-UFRGS [IBM 00], que tem como objetivo aplicar métodos pesquisados na UFRGS no acesso integrado a bases de dados legadas. Essas bases de dados, entre elas, uma base de documentos XML, possuem um conjunto variado de esquemas. Um passo importante para o projeto é justamente a integração destes esquemas em esquemas locais e globais, que possibilitem às demais aplicações do projeto acesso integrado aos dados. Para isso, estes esquemas integrados devem indicar o significado de cada conceito presente, além de indicar suas respectivas fontes de origem.

Uma solução para representar estes esquemas integrados é utilizar a idéia de ontologias. Uma ontologia é uma especificação explícita de uma conceitualização [GUA 92], isto é, uma ontologia é uma maneira formal de descrever os conceitos relativos a um determinado domínio de conhecimento. Por isso, uma ontologia pode ser considerado uma alternativa para a integração de esquemas de fontes heterogêneas. Para isso, pode-se construir ontologias que tragam consigo, além da estrutura dos esquemas integrados, todo um conjunto de informações necessários para a consulta e acesso as diferentes bases de dados.

Uma alternativa para construção destas ontologias é a utilização da linguagem XML. Isto pode ser justificado baseado no fato de que as ontologias, para possibilitar o seu acesso, devem ser compartilhadas. Representando as ontologias com XML possibilitaria a rápida distribuição das mesmas através de recursos da *web*. Além disso, as facilidades da linguagem XML estariam disponíveis, como a simplicidade do trabalho *parsing* e a sintaxe bem definida, entre outras coisas [KAR 99, COV 99].

O objetivo principal deste trabalho é então, tendo o projeto IBM-UFRGS como motivação principal, desenvolver um sistema para integração de esquemas, esquemas XML (DTDs) mais especificamente, que utilize ontologias como suporte. Para atingir tal objetivo, pode-se traçar as seguintes metas:

1. Compreender as principais características e estruturas da linguagem XML, investigando sobretudo, os esquemas XML e as ferramentas para criação e acesso a documentos XML;
2. Elaborar um breve estudo sobre ontologias, abrangendo os principais aspectos teóricos;
3. Pesquisar sobre integração de esquemas, sobretudo no caso prático desenvolvido pelo projeto IBM-UFRGS;
4. Definir os processos e métodos para atingir o objetivo principal: integrar esquemas utilizando ontologias;
5. Projetar o sistema, baseado no que foi definido em (4). A implementação deverá ser feita na linguagem Java.

Este trabalho apresenta o desenvolvimento de todas as metas acima, da seguinte maneira: no capítulo 2 será apresentada a linguagem XML e suas principais características; no capítulo 3 será apresentado um breve estudo sobre ontologias, destacando a relação entre ontologias e XML; no capítulo 4 será apresentado uma breve base teórica sobre integração de esquemas de bancos de dados, seguido da apresentação do projeto IBM-UFRGS e da proposta de integração de esquemas XML utilizando ontologias; no capitulo 5 será apresentado como está sendo implementado o protótipo do sistema de integração; e por fim, o capítulo 6 traz as conclusões, contribuições e projetos futuros relativos a este trabalho.

2 A LINGUAGEM XML

A linguagem XML (*Extensible Markup Language*) [W3C 98a] é uma linguagem de representação de documentos, recomendação do W3C (*World Wide Web Consortium*), projetada inicialmente para visualização de páginas na *web*. XML foi definida como sendo um subconjunto da linguagem SGML (*Standard Generalized Markup Language*) [ISO 86], sendo que qualquer documento XML está em conformidade com SGML. A linguagem SGML foi padronizada em 1986, baseada na *Generalized Markup Language*, inventada pela IBM em 1969. XML pode ser considerada uma simplificação de SGML para uso na *web*.

Segundo Pimentel: " XML é uma linguagem de marcação (*markup*) apropriada à representação de dados, documentos e demais entidades cuja essência fundamenta-se na capacidade de agregar informações" [PIM 00]. Com estas características, XML têm a capacidade de exprimir o significado de cada fragmento de um documento, com isso é possível a troca de informações de acordo com o seu significado entre diferentes programas, de diferentes formas e para diversos objetivos [STE 00].

Tanto XML quanto SGML são consideradas metalinguagens, isto é, são linguagens que descrevem linguagens. Com XML ou SGML é possível descrever uma nova linguagem de marcação que seja apropriada à uma determinada finalidade. Um exemplo é a conhecida linguagem HTML, utilizada em larga escala na *world wide web*. A linguagem HTML, que foi definida por Tim Berners-Lee utilizando SGML, possui um conjunto restrito de marcas para a descrição do *layout* de um documento, como <TITLE>, que descreve o título dado ao documento. Estas informações são processadas por um aplicativo, geralmente um *browser*, que terá a função de formatar o documento na tela de acordo com as instruções presentes.

Mas sendo HTML uma linguagem pré-definida, não é possível criar novas marcações, que atendam as necessidades de um determinado grupo ou comunidade, ou de um determinado aplicativo. Para permitir a possibilidade da criação de novas linguagens, adaptadas as características de um domínio, uma alternativa seria a utilização da própria SGML. Contudo, esta linguagem caracteriza-se por ser muito complexa, possuir diversas funcionalidades pouco utilizadas e de ter pouco suporte a diferentes conjuntos de caracteres. Daí a escolha de uma linguagem como XML, que por ser uma versão mais simplificada da SGML, pode ser utilizada com facilidade pelas comunidades da *web*, além de poder ser processada por programas mais leves e compactos, mais apropriados para a *web*. [PIM 00, GAR 99]

2.1 Origem e Objetivos

A linguagem XML foi desenvolvida em 1996 pelo XML *working group* do W3C, órgão responsável pela padronização na *web*. O grupo que desenvolveu a linguagem XML tinha em vista resolver os problemas descritos anteriormente. Para tal, foram estabelecidos os seguintes objetivos de projeto, segundo a recomendação do W3C [W3C 98a] :

1. XML deve ser usada sem dificuldades na Internet.
2. XML deve suportar uma grande variedade de aplicações.
3. XML deve ser compatível com SGML.
4. Deve ser fácil escrever programas que processam documentos XML.
5. O número de funcionalidades opcionais em XML deve ser o mínimo, idealmente zero.
6. Documentos XML devem poder ser legíveis pelo ser humano e razoavelmente "limpos".
7. O projeto da XML deve ser preparado rapidamente.
8. O projeto da XML deve ser formal e conciso.
9. Documentos XML devem ser fáceis de serem criados.
10. A concisão do *markup* da XML é de mínima importância.

2.2 Documentos XML

Os documentos XML bem formados são os que atendem às recomendações do W3C, descritas na seção 2.2.1. Além de ser bem formado, opcionalmente um documento XML pode ser considerado válido, se estiver de acordo com restrições especificadas numa DTD (*Document Type Definition*). [W3C 98a].

A especificação do W3C diz que "cada documento XML tem tanto uma estrutura lógica quanto uma estrutura física. Fisicamente, o documento é composto de unidades chamadas entidades. Uma entidade pode referenciar outras entidades, para causar a sua inclusão no documento" [W3C 98a]. A estrutura física diz respeito as entidades físicas, os arquivos, que podem compor um documento XML. Já estrutura lógica é o que realmente caracteriza um documento XML. Um documento XML é basicamente um documento de texto, formado por caracteres de dados e marcações. As marcações incluem elementos, referências, seções CDATA, instruções de processamento, as DTDs e os comentários. Tudo o mais que não for considerado marcação, são caracteres de dados. [W3C 98a, PIM 00]

Segue uma descrição dos seis componentes lógicos citados acima:

2.2.1 Elementos

Os elementos não nulos em XML são formados por uma marcação inicial, ou *start-tag*, e por uma marcação final, ou *end-tag*. Uma marcação é iniciada pelo caracter "<", seguido do nome do elemento e delimitada pelo caracter ">", (p.ex. <title>). Uma *end-tag* diferencia-se por iniciar com os caracteres "</", seguido do mesmo nome da sua *start-tag*, (p.ex. </title>). Este "nome" das marcações determina o tipo do elemento. Entre a marcação inicial e a marcação final ficam os dados que estão relacionados com este elemento. Estes dados podem ser qualquer componente lógico da XML, caracteres de dados ou mesmo nada [ETH 00, W3C 98a]. Na fig. 2.1(a) encontra-se um exemplo de *start-tag* e *end-tag.*

Pode-se ainda usar elementos vazios. Elementos vazios são formados por uma única marcação, onde encontra-se o nome do elemento delimitado pelos caracteres "<" e "/>", (p.ex. <paragrafo/>).

Opcionalmente, as *start-tags* e os elementos vazios podem conter atributos específicos. A declaração destes atributos vem logo após o nome do elemento. São formados pelo nome do atributo, seguido do caracter "=", após vem o valor do atributo delimitado por aspas. Na fig 2.1(b) encontra-se um exemplo de marcação com atributos.

<paragrafo> ... ... </paragrafo> **(a)**	<paragrafo alin="centro" fonte="arial">> ... ... </paragrafo> **(b)**

FIGURA 2.1 -Exemplo de *start-tag* e *end-tag* (a) e atributos num elemento (b)

No exemplo da fig. 2.2 podem ser considerados como elementos as marcações "<empresa>" (lin. 04 e 18), "<funcionário>" (lin. 05 e 09), "<nome>" (lin. 06), entre outras. A marcação "<parágrafo>" (lin. 10) é um exemplo de elemento vazio, que neste caso, serve apenas para demarcar uma determinada seção do documento. Os elementos "<empresa>" e "<funcionário>" também caracterizam-se por ter atributos. No caso de "<empresa>", um atributo identifica o nome da empresa, e em "<funcionário>", dois atributos mostram quando o funcionário entrou e quando ele saiu da empresa, além de uma identificação (ID) única para o mesmo.

Para melhor ilustrar as regras sintáticas descritas nesta seção e nas seguintes, na fig 2.2 encontra-se um exemplo simples de um documento XML bem formado.

Ainda que este exemplo não seja plenamente funcional para uso prático, ele ilustra a maioria das marcações apresentadas. A numeração das linhas em negrito serve apenas como ilustração, não fazendo parte do documento XML. Os nomes dos elementos estão destacados em negrito.

```
01: <?xml version="1.0" encoding="ISO-8859-1"?>
02: <!DOCTYPE empresa SYSTEM "empresas.dtd">
03: <!-- Exemplo de um documento XML bem formado -->
04: <empresa nome = "Byte Informática">
05: <funcionario dataInicial="22/6/90" dataFinal="30/4/99" ID="JS90001">
06:             <nome>Jose da Silva</nome>
07:             <cargo>programador</cargo>
08:             <dept>desenvolvimento de software</dept>
09:     </funcionário>
10: <paragrafo/>
11:     <funcionário dataInicial="21/7/92" ID="MF92012">
12:             <nome>
13:     Mário da Fonseca <![CDATA[  Este texto não é marcação: <> < < /> > = ]]>
14:             </nome>
15:             <cargo>vendedor</cargo>
16:             <dept>Vendas</dept>
17:     </funcionário>
18: </empresa>
```

FIGURA 2.2 - Exemplo de um documento XML bem formado

2.2.2 Referências

Existem referências a caracteres e referências a entidades. Referências a caracteres referem-se a caracteres do conjunto ISO/IEC 10646. Podem ser usadas para referenciar caracteres que não estejam diretamente disponíveis para digitação no sistema de *input* do desenvolvedor. São delimitadas pelos caracteres "&#" seguidos da representação decimal ou hexadecimal do caracter ISO/IEC e delimitados por ";". No caso da representação hexadecimal, um "x" deve preceder os dígitos correspondentes, (p.ex. &#21; e &#x2h;). [W3C 98a]

Referências a entidades são referências a caracteres de dados ou arquivos externos que devem ser substituídos após o *parser* do documento. Podem ser usados para indiretamente referenciar caracteres reservados a marcação, (p.ex."<" e "&"), dentro dos caracteres de dados do documento. Servem também para representar trechos de texto que podem aparecer repetidas vezes no documento [WAL 97]. A linguagem XML define cinco entidades padrões, como mostra Pimentel: [PIM 00]

- **&** eqüivale a &.
- **&alt;** eqüivale a <.
- **>** eqüivale a >.
- **"** eqüivale a " (aspas).
- **'** eqüivale a ' (apostrofe).

Para definir novas referências a entidades que possam ser substituídas por caracteres ou por outros arquivos, utiliza-se a DTD, que será discutido na seção 2.3

2.2.3 Seções CDATA

Seções CDATA são utilizadas para escrever porções de texto contendo caracteres que normalmente seriam reconhecidos como caracteres de marcação. Uma seção CDATA pode acontecer em qualquer área do documento destinada a caracteres de dados. Qualquer seqüência de caracteres dentro de uma seção CDATA será interpretada como caracteres de dados. Uma seção CDATA é delimitada pelas marcações "<![CDATA[" e "]]>". Qualquer caractere de marcação (p.ex. "<" ">") não será interpretado como marcação somente se estiver dentro de uma seção CDATA. A única marcação reconhecida numa seção CDATA são os caracteres "]]>". Além disso, seções CDATA não podem estar aninhadas. [PIM 00, W3C 98a]

Na fig. 2.2 encontra-se um exemplo de seção CDATA, na lin. 13. Nota-se que os caracteres utilizados como marcação "<> < < /> > =" podem ser usados somente dentro desta seção sem serem considerados marcação.

2.2.4 Instruções de Processamento

Instruções de processamento, ou PIs, são marcações especiais que não são diretamente processadas pelo *parsers*, sendo repassadas a aplicação. As instruções de processamento não fazem parte dos caracteres de dados. [W3C 98a]

As instruções de processamento são delimitadas pelos caracteres "<?" e "?>". Dentro desta marcação, fica o nome da instrução seguido dos dados referentes a instrução. O nome identifica as PIs para a aplicação, que irá processar as que tiver capacidade de reconhecer, ignorando todas as outras. Os dados que serão repassados para a aplicação podem ser de qualquer formato, sendo obrigação da aplicação interpreta-los ou não. [WAL 97]

A instrução "<?xml ...>" é a chamada "declaração XML", que opcionalmente pode vir no início do documento, indicando para a aplicação, entre outras coisas, a versão da XML utilizada e o conjunto de caracteres utilizado.[WAL 97]

Na fig. 2.2 na lin. 01 tem-se uma declaração XML. O atributo "version" indica a versão 1.0 da XML como a versão utilizada no documento. O segundo atributo "enconding", declara que o conjunto de caracteres utilizado é o ISO-8859-1.

2.2.5 DTD (*Document Type Definition*)

A presença de uma DTD num documento XML caracteriza um documento XML válido. Para indicar ao *parser* a localização e nome da DTD associada ao documento, utiliza-se a instrução "DOCTYPE". Esta declaração deve ser a primeira coisa do documento, logo após instruções de processamento ou comentários.

Na fig 2.2 na lin.02 encontra-se um exemplo da declaração “DOCTYPE”. Neste caso, a declaração aponta para um arquivo externo onde encontra-se a DTD contendo as definições das marcações, mas uma declaração “DOCTYPE” pode também apontar para um conjunto de definições internas, ou ainda pode fazer as duas coisas ao mesmo tempo.[W3C 98a, WAL 97]

O assunto DTD e documentos XML válidos será abordado na seção 2.4.

2.2.6 Comentários

Comentários são ignorados pelos processadores XML. São delimitados pelos caracteres "<!--" e "-->". Qualquer marcação presente dentro de um comentário será ignorada, com exceção da seqüência "--". Os comentários podem ser inseridos em qualquer ponto do documento, exceto dentro de outras marcações ou no início do documento.

A lin. 03 da fig. 2.2 mostra um exemplo de comentário dentro de um documento XML.

2.3 Documentos XML Bem Formados

Pimentel define que "documentos XML devem obedecer algumas regras para serem considerados documentos bem formados - de fato, para serem considerados documentos XML, já que se não forem bem formados não podem ser denominados documentos XML" [PIM 00]. Essas regras são as estabelecidas pela recomendação do W3C. Um documento que não respeite a sintaxe das marcações XML, descrita nas seções anteriores, não será considerado bem formado. Na especificação do W3C [W3C 98a], encontram-se as seguintes regras para um documento XML ser considerado bem formado:

- O documento opcionalmente pode ter a instrução de processamento declaração XML (seção 2.2.4), que deve ser a primeira marcação do documento.
- O documento deve ter um ou mais elementos.
- Deve haver somente um elemento que será o "raiz" do documento, sendo que todos os outros elementos estarão inseridos neste elemento hierarquicamente.
- Quando a *start-tag* de um elemento A estiver dentro de outro elemento B, então a *end-tag* do elemento A também deve estar dentro do elemento B. Ou seja, os elementos com *start-tags e end-tags* devem estar devidamente aninhados.

Para que um documento XML seja considerado bem formado, ainda são necessárias outras restrições, que por estarem de forma mais implícita na recomendação do W3C, são listadas abaixo como descritas por Walsh: [WAL 97]

- O documento deve estar em conformidade com a gramática dos documentos XML. Algumas marcações, como referencias a entidades, são permitidas somente em lugares específicos. O documento não é bem formado se elas aparecerem em qualquer lugar.
- O texto a ser substituído por referências a entidades dentro de uma declaração, consiste de zero ou mais marcações. (Nenhuma entidade usada no documento deve consistir de somente parte de uma marcação).
- Nenhum atributo deve aparecer mais de uma vez na mesma *start-tag*.
- Valores de atributos não podem conter referencias a entidades externas.
- As entidades devem ser declaradas antes de serem usadas, exceto as entidades pré-definidas pela XML (&amp, &lt, &gt, &apos e &quot).

- Uma entidade binária, (p. ex. uma imagem), não pode ser referenciada no fluxo do conteúdo, só podem ser usadas em atributos declarados como ENTITY ou ENTITIES.
- Nem o texto ou as entidades podem ser recursivos, direta ou indiretamente.

2.4 Documentos XML Válidos

De acordo com a recomendação do W3C "a função das marcações num documento XML é descrever o seu armazenamento e estrutura lógica e associar pares atributo/valor com a sua estrutura lógica" [W3C 98a]. Quando se faz necessário criar restrições às estruturas lógicas de um documento XML, para que ele obedeça a um conjunto de regras pré-definida, uma gramática, utiliza-se uma DTD, ou definição de tipo de documento (*Document Type Definition*).

Sempre que num documento XML existir uma DTD associada, e o conteúdo do documento segue corretamente as restrições declaradas na DTD, como o aninhamento dos elementos, os atributos permitidos ou entidades declaradas, diz-se que este documento XML é válido de acordo com esta DTD. [W3C 98a, WAL 97].

Como as DTDs permitem uma melhor estruturação dos documentos XML associados, fica mais fácil construir aplicações dedicadas a processar tais documentos. Estas aplicações podem utilizar as marcações determinadas no DTD para associar algum significado aos trechos do documento. [PIM 00]

De uma forma geral, as declarações de uma DTD permitem que o *parser* que irá processar o documento XML tenha acesso a metainformações sobre o conteúdo deste documento. Considere-se estas metainformações como a seqüência e aninhamento dos elementos, os valores permitidos para os atributos, assim como o seu tipo e valor *default*, o nome de arquivos externos que poderão ser referenciados no documento, o formato de algum dado externo que não esteja em XML, e as entidades que podem ser referenciadas. [WAL 97]

Na fig. 2.3 encontra-se uma possível DTD para o documento apresentado na fig. 2.1, mostrando as declarações mais comuns.

Existem quatro tipos de declarações possíveis numa DTD: declarações de elementos, declarações de atributos, declarações de entidades e declarações de notações. Estes quatro tipos de declarações serão descritas nas próximas seções, tendo as referências [W3C 98a] e [WAL 97] como base.

```
01: <!ELEMENT empresa (funcionario, paragrafo?)+>
02:     <!ATTLIST empresa nome CDATA #REQUIRED>
03: <!ELEMENT funcionário (nome, cargo, dept)>
04:     <!ATTLIST funcionário dataInicial CDATA          #REQUIRED
05:                                       dataFinal CDATA   #IMPLIED
06:                                       ID ID           #REQUIRED>
07: <!ELEMENT nome (#PCDATA)>
08: <!ELEMENT cargo (#PCDATA)>
09: <!ELEMENT dept (#PCDATA)>
10: <!ELEMENT parágrafo EMPTY>
```

FIGURA 2.3 - Exemplo de uma DTD para o documento da fig. 2.2

2.4.1 Declarações de Elementos

As declarações de elementos apresentam os nomes dos elementos permitidos e o modelo de conteúdo referente a estes elementos. O modelo de conteúdo descreve quais outros elementos podem ser aninhados dentro deste elemento, além da ordem em que devem aparecer e com que freqüência podem ocorrer.

Na fig. 2.3 na lin. 01 encontra-se um exemplo de declaração de elemento. No caso, o elemento "empresa". A declaração é reproduzida aqui:

<!ELEMENT empresa (funcionario, paragrafo?)+>

Uma declaração de elemento é iniciada com a marcação "<!ELEMENT", seguido do nome do elemento e do seu modelo de conteúdo entre parênteses. Ao final é delimitado pela marcação ">".

O modelo de conteúdo apresenta os elementos que podem ser aninhados com este elemento, ou seja, os elementos que podem ser considerados "filhos" deste elemento. Cada nome de elemento é separado por virgula, o que determina a ordem em que os elementos devem aparecer. No exemplo acima, o elemento “empresa” deve ter aninhado entre a sua *start-tag* e *end-tag* os elementos "funcionário" e "parágrafo", nesta ordem.

Complementando o modelo de conteúdo, os elementos declarados podem trazer uma marca especial indicando a freqüência com que podem ocorrer. No caso do elemento “empresa” do exemplo, o elemento “parágrafo” pode aparecer uma ou nenhuma vez. Isso é mostrado através do ponto de interrogação logo após o nome do elemento. Ainda neste exemplo, a seqüência funcionário/empregado pode aparecer uma ou muitas vezes, como indicado pelo sinal de adição após os parênteses. Segue abaixo todos os marcadores de ocorrência:

- **? :** o elemento é opcional, pode aparecer uma ou nenhuma vez.
- *** :** o elemento é opcional, pode aparecer nenhuma ou muitas vezes.
- **+ :** o elemento deve aparecer uma ou muitas vezes.
- **Nenhuma marca:** o elemento deve aparecer uma única vez.

Outro tipo de declaração possível no modelo de conteúdo é a chamada lista de escolha. Dois ou mais elementos podem ser considerados opcionais, sendo que somente um deles pode aparecer por vez. A lin. 02 da fig. 2.3 poderia ser modificada da seguinte maneira:

<!ELEMENT funcionário (nome, (cargo | dept))>

A marcação "|" expressa uma relação do tipo "ou". Neste exemplo, após o elemento "nome", deve vir o elemento "cargo" ou o elemento "dept", mas somente um dos dois.

Os modelos de conteúdo como o modelo do elemento “empresa” ou do elemento “nome” da fig 2.3 são elementos do tipo seqüência, que expressam uma seqüência de elementos. Mas ainda podem existir três outros tipos padrões:

- **PCDATA:** Na fig. 2.3 linhas 07, 08 e 09, os elementos "nome", "cargo" e "dept" têm declarados no seu modelo de conteúdo a palavra-chave "#PCDATA" (*parsed caracter data*). Isto quer dizer que o conteúdo que deve ser encontrado dentro destes elementos deve consistir tão somente de caracteres de dados. Elementos desse tipo caracterizam-se por serem sempre "folhas" na hierarquia de elementos do documento XML.
- **ANY:** Um elemento que contenha em seu modelo de conteúdo a palavra-chave "ANY" pode ter qualquer tipo de conteúdo. Não é aconselhável o uso de elementos desse tipo, já que isso levaria a impossibilidade de validar o conteúdo do elemento.
- **EMPTY:** Um elemento pode ser vazio, como foi explicado na seção 2.2.1. Na lin. 10 da fig. 2.3 encontra-se a declaração do elemento vazio "parágrafo". Como se pode ver, o modelo de conteúdo daquele elemento contém apenas a palavra-chave “EMPTY”.

Um elemento pode ainda ser formado tanto por caracteres de dados quanto por novos elementos. Este é um elemento do tipo misto. Modificando-se a lin. 09 da fig. 2.3 para obter-se um elemento misto:

<!ELEMENT dept (#PCDATA | sala)>

Para se compor uma declaração de elemento misto, deve-se começar sempre o modelo de conteúdo com PCDATA, para depois enumerar os outros elementos. No exemplo acima, um elemento “dept” pode tanto ser formado por caracteres de dados, ou pode ter a marcação “sala” aninhada.

2.4.2 Declaração de Atributos

Como foi dito anteriormente, os elementos podem opcionalmente conter atributos. Numa DTD é possível definir restrições aos atributos através das declarações de lista de atributos. Estas restrições são: que tipo de atributos um elemento pode ter, que valores os atributos podem assumir e que valor padrão os atributos podem ter.

A lin. 04 da fig. 2.3 mostra um exemplo de declaração de lista de atributos, como reproduzido aqui:

<!ATTLIST funcionário dataInicial CDATA #REQUIRED

```
                    dataFinal CDATA   #IMPLIED
                              ID ID   #REQUIRED>
```

A declaração é delimitada pelas marcações “<!ATTLIST” e “>”. A primeira informação é o nome do elemento ao qual os atributos estão associados. No exemplo este elemento é “funcionário”, declarado na lin. 04. Logo em seguida vem a declaração dos atributos em si, na seguinte ordem: nome do atributo, tipo e valor *default*. No exemplo, a primeira declaração é a do atributo “dataInicial”, seu tipo é CDATA, ou seja, caracteres de dados, e ele é obrigatório. O atributo seguinte, “dataFinal”, também é do tipo CDATA, mas não é obrigatório, e por ultimo o atributo “ID”, do tipo ID, que é obrigatório.

Os tipos de atributos definem que tipos de dados os atributos poderão assumir. Eles podem ser de seis tipos:

- **CDATA**: atributos deste tipo podem conter qualquer tipo de texto;
- **ID**: todos os atributos do tipo ID num documento devem conter valores diferentes. Por causa disto, estes atributos são utilizados para identificar individualmente os elementos no documento.
- **IDREF ou IDREFS**: os atributos do tipo IDREF referenciam um atributo do tipo ID de outro elemento existente no documento. Atributos do tipo IDREFS podem conter uma seqüência de valores do tipo IDREF, separados por espaços em branco.
- **ENTITY ou ENTITIES**: o valor de um atributo do tipo ENTITY deve ser um único nome de uma entidade declarada no DTD. Declarações de entidades serão tratadas na seção 2.4.3. Um atributo do tipo ENTITIES pode conter múltiplos valores do tipo ENTITY, separados por espaços em branco.
- **NMTOKEN ou NMTOKENS**: o valor do atributo do tipo NMTOKEN deve consistir numa única palavra (p. ex.: tipo=“objeto”), sem maiores restrições. Atributos do tipo NMTOKENS são uma seqüência de valores do tipo NMTOKEN, separados por espaços em branco.
- **Lista de nomes**: é possível especificar um lista de nomes que deve ser tomada como possível valor do atributo. Por exemplo, um atributo de nome “tipo” poderia ter a seguinte lista de possíveis valores: (objeto | relacional).

Existem quatro possíveis valores *default* que um atributo pode assumir:

- **#REQUIRED**: o atributo deve ter um valor especificado em cada ocorrência do elemento a que pertence.
- **#IMPLIED**: o atributo não é obrigatório, e nenhum valor padrão é especificado. Caso o atributo não seja usado, ele pode ser ignorado pelo *parser*.
- **“Valor”**: qualquer valor indicado entre aspas pode ser considerado como valor *default* para este atributo, caso o mesmo não seja especificado.
- **#FIXED “valor”**: neste caso, o atributo tem um valor fixo. O atributo não é obrigatório, mas caso ocorra, deverá ter sempre o valor especificado nesta declaração.

2.4.3 Declaração de Referências a Entidades

As entidades, como foi abordado na seção 2.2.2, permitem associar um nome a um fragmento do documento. Este fragmento pode ser uma seqüência de caracteres ou um arquivo externo.

A fig. 2.4 mostra exemplos de declarações de entidades.

```
01: <!ENTITY link "www.ufpel.tche.br" >
02: <!ENTITY fig1 SYSTEM  "/figuras/logo.gif" NDATA  GIF87A >
03: <!ENTITY intro SYSTEM  "/docs/introducao.xml" >
```

FIGURA 2.4 – Exemplo de Declaração de Entidade

Uma declaração de entidade é delimitada pelas marcações "<!ENTITY" e ">". A primeira declaração é a do nome da entidade, que será usado para referenciar a mesma no documento XML, seguido da especificação da entidade.

Existem três tipos de entidades: entidades internas, entidades externas e entidades parametrizadas.

- **Entidades internas**: entidades internas são seqüências de caracteres declaradas dentro da própria DTD, como a da lin. 01 da fig. 2.4. neste caso, cada ocorrência de "&link;" no documento XML será substituído pelo *parser* por "www.ufpel.tche.br". A linguagem XML tem entidades internas pré-definidas, como mostrado na seção 2.2.2.
- **Entidades externas**: entidades externas são arquivos externos ao documento XML corrente. A marcação "SYSTEM" logo após o nome da entidade passa esta informação ao *parser*. A lin. 02 e a lin. 03 da fig. 2.4 são dois exemplos de declarações de entidades externas. A lin. 03 mostra uma entidade chamada "intro", que referencia um arquivo de texto chamado "introducao.xml" que deverá ser anexado ao documento XML a cada ocorrência de "&intro". Já na lin. 02, o arquivo referenciado é binário, do tipo GIF87A, neste caso, o *parser* irá repassar esta informação para a aplicação, que será responsável em tratar este arquivo como lhe convier.
- **Entidades parametrizadas**: entidades parametrizadas tem a mesma utilidade das entidades internas, com o diferencial de serem utilizadas apenas dentro da própria DTD. Uma entidade deste tipo é identificada colocando-se a marcação "%" na frente do nome na declaração.

2.4.4 Declarações de Notações

Uma notação identifica através de um nome o formato de uma entidade externa que não possa passar pelo *parser*, ou seja, de algum tipo de arquivo binário. As notações são utilizadas dentro da própria DTD, como na lin. 02 da fig. 2.4., no caso, a chamada "GIF87A" ao final da declaração. Uma possível declaração de notação para este caso pode ser mostrada abaixo:

<!NOTATION **GIF87A** SYSTEM "GIF" >

Cabe a aplicação receber esta informação e tomar as medidas necessárias para processar a entidade externa.

2.5 Processadores XML

Processadores XML são *parsers* dedicados que funcionam junto com alguma aplicação no processamento e validação de documentos XML. A especificação do W3C [W3C 98a] define um processador XML como sendo "um módulo de software usado para ler documentos XML e prover acesso ao seu conteúdo e estrutura". Para que possa ser considerado um processador XML, o programa precisa estar de acordo com todas as regras definidas na especificação XML do W3C, apresentadas nas seções anteriores. Entre outras coisas, um processador XML deve saber quando passar uma condição de erro fatal para a aplicação, que deverá, ou não, saber lidar com tal situação [W3C 98a, PIM 00].

Uma melhor definição de um *parser* de XML vem de Johnson [JOH 00]: "Um *parser* XML básico faz grande parte do trabalho para o programador, reconhecendo tokens, traduzindo caracteres codificados, checando a validade de alguns valores de dados e fazendo chamadas ao código de aplicações especificas, quando apropriado."

De acordo com a W3C, existem dois tipos de processadores XML: processadores validadores e processadores não-validadores.

- **Processadores Validadores**: processadores validadores devem analisar se o documento XML está em conformidade com a DTD que o define, informando eventuais violações das restrições declaradas na DTD para a aplicação. Além disso, os processadores validadores devem verificar se o documento é bem-formado, de acordo com as regras apresentadas na seção 2.3.
- **Processadores Não-Validadores**: um processador não-validador preocupa-se tão somente em verificar se o documento é bem-formado, repassando as violações encontradas para a aplicação.

A fig. 2.5, baseada em Pimentel [PIM 00], ilustra o processamento de um documento XML:

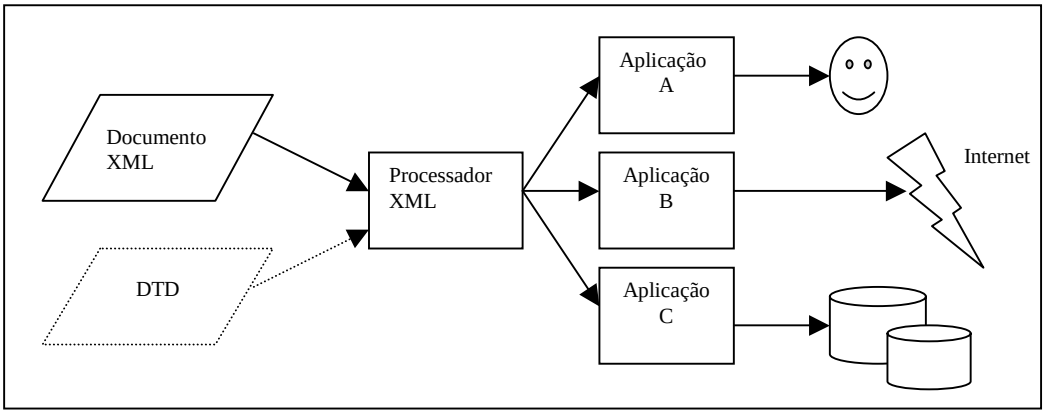


FIGURA 2.5 – Processamento de um documento XML

O processador recebe um documento XML, que opcionalmente tem uma DTD que o descreve. Cabe ao processador validar o conteúdo deste documento, tanto quanto as regras de sintaxe da XML quanto as restrições declaradas na DTD. O conteúdo do documento será então repassado para a aplicação, que se encarregará de apresentar as informações para o usuário, ou ainda transmitir pela *internet* ou armazenar num banco de dados [PIM 00]. Pode-se perceber que diferentes aplicações podem encontrar diferentes finalidades para o documento XML, mesmo que utilizem o mesmo *parser*.

Existe atualmente uma grande variedade de processadores para XML, tanto comerciais quanto gratuitos. O W3C não define nenhum padrão de como os processadores devem ser implementados, ou como devem proceder durante o *parsing*. Mas quanto ao formato em que as informações serão repassadas para a aplicação, o W3C definiu duas APIs (*Aplication Program Interface*) padronizadas. As aplicações podem utilizar estas APIs para acessar tanto a estrutura quanto os dados do documento XML. Estas APIs são o SAX (*Simple API for XML*) [MEG 98] e o DOM (*Document Object Model*) [W3C 98b].

2.5.1 SAX – *Simple API for XML*

Esta API não é exatamente uma especificação do W3C. O SAX foi desenvolvido a partir de uma iniciativa dos integrantes da lista de discussão XML-DEV, mantida pelo próprio W3C e hoje é distribuído por David Megginson. Atualmente na versão 2.0, o SAX define um conjunto de interfaces em Java para o acesso às informações em documentos XML de forma serial e baseado em eventos. [MEG 98]. Uma interface Java é uma coleção de definições de métodos e constantes que podem ser implementados por classes [SUN 00]. O SAX caracteriza-se por ser uma API bastante simples e leve, o que viabiliza seu uso em aplicações menores, como em *applets* e *servlets* por exemplo.

As interfaces que o SAX provê são as seguintes:

- **Parser**: esta interface é responsável por fazer o trabalho de *parsing*, além de invocar os métodos baseados em eventos.

- **DocumentHandler**: esta interface define os métodos que serão invocados quando as marcações e os caracteres de dados do documento XML são encontrados pelo *parser*.
- **ErrorHandler**: esta interface define os métodos, *error*, *fatalError* e *warning*, que são invocados sempre que o *parser* encontrar violações no documento.
- **DTDHandler**: os métodos definidos nesta interface são invocados durante o *parsing* da DTD.
- **EntityResolver**: o método *resolveEntity* definido nesta interface é chamado sempre que uma referência a uma entidade externa for encontrada no documento.

O SAX é uma API baseada em eventos. Para o desenvolvimento de um programa típico utilizando SAX, o programador deve criar uma classe que implemente a interface *DocumentHandler* para gerenciar os eventos. Em seguida, deve criar uma instância de *Parser*. Quando o programador chamar a operação *parse()*, deverá indicar como parâmetros, o nome do arquivo XML e da classe que ele criou. O *parser* do SAX irá então converter o documento XML numa seqüência de eventos, correspondentemente às estruturas do documento, como marcações e texto. Os métodos da classe que implementou *DocumentHandler* serão invocados a medida que estas estruturas forem sendo processadas pelo *parser* [JOH 00]. Os principais métodos da interface *DocumentHandler* são os seguintes:

- **startDocument()**: é invocado cada vez que se inicia o *parsing* de um novo documento XML.
- **endDocument()**: invocado ao final do documento XML.
- **startElement()**: este método é invocado sempre que o *parser* encontra uma *start-tag*. Como parâmetros, o método recebe o nome do elemento e um vetor com os atributos, caso o elemento os tenha .
- **endElement()**: este método é invocado a cada passagem por uma *end-tag*, recebendo como parâmetro o nome do elemento.
- **characters()**: este método recebe tanto os caracteres de dados, formados por texto regular, quanto as seções CDATA.

Cabe ao programador implementar estes métodos e definir que tratamento irá fazer para os elementos e caracteres. O acesso às informações do documento XML no SAX é sempre seqüencial e somente para leitura, não sendo possível, portanto, alterar a estrutura do documento diretamente pelo SAX.

2.5.2 DOM – *Document Object Model*

O objetivo principal do DOM é propiciar às aplicações um melhor acesso às informações contidas nos documentos XML. A especificação do W3C [W3C 98b], define o DOM da seguinte maneira: "O *Document Object Model* (DOM), ou Modelo de Objetos de Documento, é uma 'interface de programação de aplicações' (API) para documentos HTML e XML. Ele define a estrutura lógica dos documentos e o modo como um documento é acessado e manipulado". O termo interface aqui representa um conjunto de definições de métodos sem nenhuma informação quanto a sua implementação. A especificação do DOM define tanto um modelo lógico para a representação de documentos XML e HTML quanto um conjunto de interfaces para acessar e modificar esta estrutura.

A especificação do DOM do W3C [W3C 98b], é dividida em duas partes: *Core Level 1* e *HTML Level 1*. A primeira define um conjunto de nível mais fundamental de interfaces para representar qualquer documento estruturado, além de uma extensão de interfaces para a representação de documentos XML. A segunda parte define um conjunto de interfaces de nível mais alto, que usados em conjunto com as interfaces do *Core Level*, servem para a representação de documentos HTML. Uma implementação do DOM deve, no mínimo, ter todas as interfaces do *Core Level*, mas não necessariamente do *HTML Level*. Esta seção tratará apenas da especificação do *Core Level*.

O *Document Object Model* representa os documentos XML na forma de uma estrutura de árvore de objetos, onde cada nodo desta árvore representa um dos componentes de uma estrutura XML, ou seja, representa os elementos, textos, entidades, entre outros. Esta árvore de objetos reflete diretamente a hierarquia do documento XML, criada a partir do aninhamento dos elementos.

Utilizando o DOM, é possível "navegar" no documento XML, sendo possível modificar tanto o conteúdo (os caracteres de dados), quanto a estrutura lógica do documento, adicionando e removendo elementos. Pode-se dizer então, que, ao contrário do SAX, o DOM provê um acesso randômico ao documento XML. [PIM 00, W3C 98b]

O W3C decidiu definir o DOM como sendo uma API independente de plataforma, que possa ser utilizada com qualquer linguagem de programação. Para criar tal especificação, o W3C utilizou a linguagem de definição IDL (*Interface Definition Language*) [OMG 97], além de criar outras duas especificações: uma para ser utilizada por programas JAVA e outra por programas EMACScript [W3C 98b]. Não foi definida nenhuma forma de como esta árvore de objetos deverá ser gerada. Esta função é delegada a um *parser* apropriado.

Para se entender como os documentos XML são representados no modelo DOM, considere-se o seguinte documento da fig. 2.6:

```
<Ficha de Inscricao>
        <Dados Pessoais>
                <Nome>Jose da Silva</Nome>
                <Endereco>
                        <Rua>Av. dos Perdidos</Rua>
                        <Num>432</Num>
                </Endereco>
                <Profissao>Programador</Profisao>
        </Dados Pessoais>
        <Curso>
                <Descricao>Curso de XML</Descricao>
                <Codigo>89798</Codigo>
        </Curso>
</Ficha de Inscricao>
```

FIGURA 2.6 – Exemplo de um Documento XML para uso no DOM

Este documento apresenta uma ficha de inscrição para um curso, com os dados da pessoa e do curso pretendido. Neste documento, o elemento raiz é “Ficha de Inscrição”, que tens dois elementos filhos: “Dados Pessoais” e “Curso”. Caso um *parser* DOM receba este arquivo como entrada, o modelo gerado seria como o da fig 2.7:

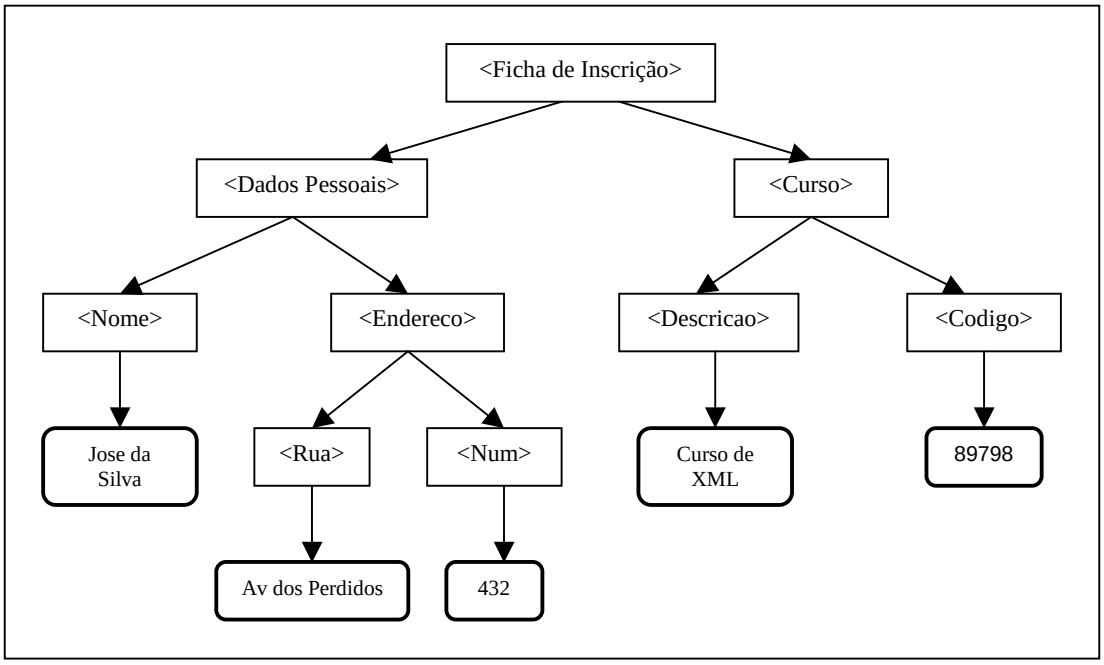


FIGURA 2.7 – Exemplo de modelo DOM

Como se pode ver, os elementos folhas serão sempre o texto, ou seja, os caracteres de dados pertencentes aos elementos.

As principais interfaces definidas pela especificação DOM são as seguintes:

- **Node:** qualquer nodo na árvore é uma instância de *Node*. Todos os outros componentes que podem entrar na árvore de objetos do DOM são especializações da interface *Node*. Operações importantes como *appendChild(Node)*, *getFirstChild()*, *getChildNodes()*, *getNodeName()*, *getNodeValue()*, entre outras, são definidos nesta interface
  [W3C 98b, PIM 00].
- **Document:** a interface *Document* representa todo o documento XML. Conceitualmente, é um nodo que serve como raiz para a árvore de objetos e provê o acesso inicial aos dados do documento [W3C 98b] . Na fig. 2.7, o nodo *Document* ficaria acima do elemento "Ficha de Inscrição".
- **Element:** é a interface mais utilizada. Define os atributos e operações específicos para um nodo do tipo elemento. Além de utilizar as operações definidas em *Node*, esta interface define mais algumas operações especificas, como *getAtrribute()* e *getElementsByTagName(String)*
  [W3C 98b, PIM 00]. Na fig. 2.7, o nodo "Endereco" é um exemplo de Element, que tem dois outros elementos filhos: "Rua" e "Num".
- **Text:** esta interface representa o conteúdo (caracteres de dados) dos elementos ou atributos. Nodos deste tipo só podem ser "folhas", como pode ser visto na fig. 2.7, nos nodos "Jose da Silva" e "432" [W3C 98b].
- **Attr:** representa os atributos contidos num objeto do tipo *Element*. Não chega a realmente ser considerado como parte da árvore de objetos, sendo considerado como uma propriedade de um objeto *Element* [W3C 98b].

2.6 Linguagens Relativas a XML

Juntamente com a especificação da versão 1.0 da XML, o W3C dispõe de um conjunto cada vez maior de módulos e linguagens derivadas da própria XML para complementá-la em tarefas específicas. Dentre tantas tecnologias e soluções, as principais serão apresentadas de uma maneira breve nas seções seguintes.

2.6.1 XML Schema

Como foi explicado na seção 2.4, a presença de uma DTD descrevendo a estrutura de um documento caracteriza um documento XML válido. Foram mostradas as principais estruturas que as DTDs disponibilizam para esquematizar os documentos XML. Mas o W3C, pensando em estender as capacidades das DTDs, vem trabalhando na recomendação da linguagem *XML Schema* [W3C 00a], que provê um conjunto de recursos que vão além dos providos pelas DTDs.

De acordo com a W3C, o principal propósito da *XML Schema* é definir uma "classe" de documentos XML. Por isso, um documento XML que esteja de acordo com um determinado *XML Schema* é chamado de "instância de documento" [W3C 00a].

Em *XML Schema* existe a distinção entre "tipos complexos" e "tipos simples". Os elementos deverão ser uma instância de um tipo, sendo que vários elementos podem pertencer ao mesmo tipo. Um elemento será declarado como sendo do tipo complexo se ele for formado por outros elementos ou atributos, já um elemento do tipo simples nunca terá outros elementos ou atributos, apenas caracteres de dados.

Sendo assim, a declaração de um tipo definirá os elementos e atributos agregados a esse tipo ou o formato dos caracteres de dados. A grande diferença entre DTDs e *XML Schema* pode ser vista aqui. Ao contrário da DTD, em *XML Schema* é possível definir o formato dos caracteres de dados dos tipos simples, sendo possível formatar datas, restringir o número de caracteres, a capitulação, a ordem de dígitos e letras, entre várias outras coisas.

2.6.2 XSL – *Extensible Stylesheet Language*

A linguagem XML é basicamente uma linguagem para estruturação de documentos, mas ao contrário de HTML, não é exatamente para a visualização deles. Para tal, o W3C definiu a linguagem XSL [W3C 00b], que serve para expressar folhas de estilo. Com XSL, o desenvolvedor pode projetar folhas de estilo que descrevam como o conteúdo estruturado num documento XML deve ser apresentado, ou seja, descreve o *layout*, a paginação e o estilo que a apresentação do documento deve ter para o usuário final na aplicação.

Para criar tais formatações, é necessário um processador de folhas de estilo que suporte XSL. Este processador irá receber os dados do documento XML e da folha de estilo XSL associada e irá produzir uma apresentação equivalente, projetada pelo desenvolvedor. O processo para gerar esta apresentação é dividido em duas fases: a transformação de árvore e a formatação.

- **Transformação da árvore:** nesta fase, o processador irá tomar a árvore de elementos do documento XML original e modifica-la, filtrando e reordenando os elementos de acordo com as definições da folha de estilo e ainda adicionando, quando necessário, novos elementos gerados também a partir da folha de estilo. Os nodos desta árvore resultante são agora chamados de objetos de formatação.
- **Formatação:** este processo irá tomar a árvore resultante do processo de transformação e irá interpretá-la para produzir a formatação desejada ao documento e apresentá-la ao usuário.

2.6.3 XLink – *XML Linking Language*

Assim como HTML, também é possível criar *hiperlinks* entre diferentes recursos num documento XML. Para tal, o W3C definiu a *XLink*, ou *XML Linking Language* [W3C 00c]. A *XLink* é uma linguagem que permite a inserção de elementos em documentos XML para criar e descrever ligações (*links*) entre diversos recursos.

O diferencial entre as ligações em HTML e *XLink* é o fato de um *hiperlink* em HTML ser sempre unidirecional, enquanto que uma ligação em *XLink* pode ter um grau de complexidade bem maior. Com *XLink* é possível criar relacionamentos entre mais de um recurso, associar metadados com uma ligação e expressar ligações que residem em lugares separados dos recursos.

XLink utiliza dois conceitos fundamentais para criar os *hiperlinks*: *links* e *resources*. Um *resource* é um conceito universal na *www*, é qualquer unidade de informação ou serviço endereçável, como arquivos de texto, imagens, programas, consultas, entre outros.

Já um *link* é um relacionamento explícito entre dois recursos, e é expressado através de um chamado "elemento de ligação". Um elemento de ligação pode ser de dois tipos: s*imple links* se associa exatamente dois recursos, um local e outro remoto, e e*xtended links* se utilizam as demais características da *XLink*, como relacionar dois ou mais recursos e ligações de lugares separados do recurso relacionado com a ligação. [PIM 00].

2.6.4 *Namespaces*

Os *namespaces*, na verdade, são mais uma extensão da própria base da linguagem XML do que uma nova linguagem. Mas seu uso é fundamental nas diversas linguagens definidas em XML, incluindo todas as apresentadas nesta seção.

A intenção desta recomendação do W3C é evitar possíveis “colisões” entre tipos de elementos e nomes de atributos. Imaginando-se a situação em que um determinado conjunto de *tags* é utilizado por dois documentos XML diferentes, sendo que cada documento tenta exprimir um significado diferente para cada *tag*, pode acontecer de um aplicativo vir a processar estes dois documentos. Neste caso, é necessário garantir que o aplicativo tenha condições de resolver o significado de cada marcação, ainda que duas marcações do mesmo tipo tenham significados diferentes.

Para resolver estes eventuais problemas de reconhecimento e colisão de tipos de elementos e de atributos, é utilizado um *namespace*. O *namespace* irá qualificar o contexto de uma marcação e torná-la única. O W3C define um *namespace* XML como sendo "uma coleção de nomes, identificada por uma referência URI (*Uniform Resource Identifier)*, que é usada em documentos XML como tipos de elementos e nomes de atributos" [W3C 99].

Para criar um *namespace* dentro de um documento XML, basta declará-lo no inicio do documento, ligando-o diretamente a um endereço da *web*. Como os endereços *web* são únicos, eles são a maneira ideal para criar contextos únicos. Por exemplo, pode-se criar um *namespace* chamado "UFPEL" e liga-lo diretamente a URI "www.ufpel.tche.br", e então utilizá-lo em todo o documento, da seguinte forma: "UFPEL:nome_do_elemento" ou "UFPEL:nome_do_atributo". Em seguida, se procederia de forma equivalente com o *namespace* "UFRGS" em outro documento. Imaginando-se que esses dois documentos utilizam a marcação "<artigo>", ainda que com significados diferentes, e utilizando algum aplicativo, os dois documentos foram integrados num único. Graças aos *namespaces* seria possível resolver a origem de cada elemento e determinar o seu correto significado, tendo em vista que existiriam tanto ocorrências como "<UFPEL:artigo>" quanto "<UFRGS:artigo>".

2.6.5 RDF – *Resource Description Framework*

O RDF (*Resource Description Framework*) é uma linguagem criada a partir da XML. O W3C a define como "uma fundação para o processamento de metadados, provê interoperabilidade entre aplicações que trocam informações 'reconhecidas por máquina' entre si na *web*" [W3C 00d]. O que RDF faz é descrever formalmente um determinado recurso da *web* através de seus metadados.

Segundo Pimentel, RDF descreve uma sintaxe para identificar um recurso da *web*. Para tal, o modelo RDF utiliza XML para associar propriedades e valores para os recursos [PIM 00]. Dessa forma, RDF pode ser utilizado em diversas aplicações da *web*, como em mecanismos de busca, por agentes inteligentes de *software* como uma forma de compartilhar e trocar conhecimentos, para descrever os direitos legais de uma página *web* ou para descrever as preferências de um usuário.

O modelo RDF representa os recursos relacionados entre si como uma árvore, onde cada nodo é a descrição de um recurso através de propriedades e valores. Além disso, são representados em cada nodo as relações entre os diferentes recursos ali descritos.

Dada a complexidade da RDF, que foge ao escopo deste trabalho, não será detalhado aqui as suas características, recomendando-se a leitura da recomendação do W3C [W3C 00d] para mais detalhes.

3 ONTOLOGIAS

Neste capítulo será apresentado um breve estudo sobre as ontologias, visando uma melhor compreensão dos capítulos seguintes. Ainda neste capítulo será apresentado também como as ontologias e a linguagem XML estão relacionadas.

As ontologias vêm sendo utilizadas em diversas áreas da computação, entre elas na área de Inteligência Artificial, principalmente como um meio para representação do conhecimento. Graças a sua capacidade de representar domínios de uma maneira compartilhada e comum entre máquinas e seres humanos, as ontologias vêm também sendo utilizadas em áreas como integração inteligente de informações, recuperação de informações na Internet e gerenciamento do conhecimento [STU 98].

O estudo das ontologias começou na verdade na filosofia, com o grego Aristóteles e suas tentativas de classificar as coisas do mundo. Do ponto de vista da filosofia, as ontologias são vistas como sendo um forma de investigação dos conceitos que nos permitem conhecer e determinar os objetos reais [SAN 00].

A definição de ontologias mais utilizada na computação é a de Gruber: “uma ontologia é uma especificação explícita de uma conceitualização” [GRU 92].

É um tanto difícil definir o que significa ontologia, mas Studer considera como um bom conceito o que diz que “uma ontologia é uma especificação explícita e formal de uma conceitualização compartilhada”. “Conceitualização” diz respeito a um modelo abstrato de algum fenômeno, identificando-se os conceitos relevantes daquele fenômeno. “Explícita” quer dizer que os tipos de conceitos utilizados e as restrições quanto ao seu uso devem ser explicitamente especificados. “Formal” significa que a ontologia resultante deve poder ser lida por máquina (*machine readable*). E finalmente “compartilhada” diz que a ontologia deve refletir o conhecimento consensual, ou seja, aquilo que é aceito por uma determinada comunidade [STU 98].

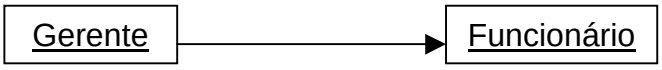
Outro conceito vem de Guarino, que diz que as ontologias podem “ser vistas como sendo o estudo da organização e da natureza do ‘mundo’, independentemente da forma do nosso conhecimento sobre ele” [GUA 95].

De uma forma geral, uma ontologia é uma maneira formal de descrever um domínio qualquer, através dos seus conceitos e relacionamentos entre conceitos, de uma maneira que agentes de software possam utilizá-la [GRU 92].

A fig. 3.1, mostra como poderiam ter sido designados dois conceitos numa ontologia. Neste caso, o domínio abordado seria o organograma de uma empresa. A ontologia descreveria então, através de conceitos, relações e restrições, as características deste domínio.

Conceito	Relação	Restrição
Gerente	Supervisiona o func.	Supervisiona no máximo 10 funcionários
Funcionário	Supervisionado pelo gerente	

(a)



(b)

Na fig 3.1(a) pode-se ver a definição de dois conceitos: “Gerente” e “Funcionário”. Uma relação pode ser estabelecida entre eles, considerando-se que o gerente é quem supervisiona os funcionários. Por fim, uma restrição foi definida para o gerente, dizendo que este não pode supervisionar mais do que 10 funcionários. Em fig. 3.1(b), pode-se observar melhor, através de um diagrama, a relação descrita anteriormente.

Uma ontologia deve prover um entendimento comum e compartilhado de um domínio que possa ser comunicado através de pessoas e computadores. Dessa forma, a ontologia pode ser considerada uma forma de se modelar um domínio. Uma ontologia provê um vocabulário de termos e relações com os quais um domínio pode ser modelado [STU 98].

Segundo Uschold, citado por [SAN 00], não existe uma forma padronizada de construir uma ontologia. Uma ontologia pode assumir diversos formatos, mas necessariamente deve incluir um vocabulário de termos e suas definições.

Outro conceito importante quando se trata de ontologias é o de “comprometimento ontológico” (*ontological commitments*). Uma ontologia pode ser construída para representar comprometimentos ontológicos para um conjunto de agentes de *software*, para que eles possam trocar informações entre si a respeito de um determinado domínio. Diz-se que um agente “compromete-se” (*commits*) com uma ontologia se suas ações são consistentes com as definições na ontologia [GUA 92].

Gruber diz ainda que “pragmaticamente, uma ontologia comum define o vocabulário com o qual consultas e asserções são trocadas entre agentes” [GUA 92]. Isto quer dizer que uma ontologia comum, isto é, uma ontologia reconhecida por vários agentes, define de que forma um agente vai se comportar, ou ainda, que caminhos ele seguirá para tomar decisões interagindo com outros agentes.

3.1 Tipos de Ontologias

Studer classifica as ontologias da seguinte maneira, de acordo com o seu nível de generalidade [STU 98]:

- **Ontologias de domínio:** capturam o conhecimento válido para um tipo particular de domínio.
- **Ontologias genéricas ou de senso comum:** capturam o conhecimento geral sobre o mundo e proporcionam noções básicas e conceitos para coisas como tempo, espaço, etc. Consequentemente, são válidas entre vários domínios.
- **Ontologias representacionais:** estas ontologias não estão comprometidas com nenhum domínio em particular, elas mostram entidades representacionais sem dizer o que deveria ser representado.
- **Ontologias de tarefa:** disponibilizam termos específicos para tarefas particulares (p. ex: o conceito "hipótese" pertence a ontologia de tarefa diagnóstico).
- **Ontologias de método:** disponibilizam termos específicos para métodos de resolução de problemas (p. ex.: "estado correto" pertence a ontologia de método Proposta-e-Revisão).

Uschold, citado por [SAN 00], diz que uma ontologia pode ainda ser classificada quanto ao seu formalismo, sendo que pode ser considerada altamente informal, ou seja, em linguagem natural, semi-informal, usando uma forma mais estruturada da linguagem formal, semi-formal, usando uma linguagem artificial formalmente definida, ou ainda rigorosamente formal.

Segundo ainda Uschold, citado por [SAN 00], as ontologias podem ser usadas para resolver os seguintes tipos de problemas:

- **Comunicação:** estabelecendo comunicação entre pessoas com diferentes necessidades e pontos de vista, existentes em função dos seus diferentes contextos.
- **Interoperabilidade:** entre sistemas de software, obtido através da tradução entre diferentes métodos de modelagem, paradigmas, linguagens e ferramentas de software.
- **Engenharia de Software:** resolve problemas de reusabilidade, confiabilidade e especificação.

3.2 Ontologias e XML

Na Internet, milhares de repositórios de informações tornam-se disponíveis a cada dia, ao mesmo tempo em que isto torna a apresentação e acesso a informação mais simples, também faz com que as tarefas de acessar e sintetizar as informações requeridas sejam muito mais difíceis. Ferramentas de busca baseadas em palavras-chave que simplesmente fazem uma busca léxica pela rede são inadequadas para consultas mais especificas. Studer diz que este não é um problema de distribuição de informação, mas sim de integração de informação [STU 98].

Um novo problema surgirá a medida que mais documentos XML forem adotados na *web*. Como Robin Cover afirma, as ferramentas de busca atuais da Internet resolvem parcialmente o problema de encontrar os recursos mais comuns, mas não servem para as necessidades de usuários e desenvolvedores de XML que queiram reutilizar módulos de informação ou desenvolver componentes. Uma ferramenta de busca para recursos XML deve ser sustentada por uma base de dados na qual os módulos e componentes XML estejam catalogados, referenciados e indexados [COV 99].

Por outro lado, surge o problema de como representar as ontologias de uma maneira eficiente e padronizada. Uma solução óbvia é utilizar a própria linguagem XML, como já vem acontecendo em alguns casos.

Estas duas situações descritas acima serão melhor discutidas a seguir.

3.2.1 Acessando Documentos XML com Ontologias

Segundo Studer, a melhor qualidade da linguagem XML, a sua extensibilidade, também é o seu maior empecilho. Quando linguagens são criadas a partir da XML, novas DTDs são especificadas e novos documentos são desenvolvidos e distribuídos, principalmente pela *web*, e espera-se que as aplicações que tenham a tarefa de interpretar estes documentos estejam de acordo com a semântica das marcações e com a estrutura destes documentos. Quando isso não acontece, ou seja, quando o receptor das informações em XML não tem compreensão da semântica e da estrutura pretendida pelo desenvolvedor, a tarefa de interpretar corretamente o significado dos dados é mais difícil. Se diferentes fontes de informação são integradas, a interpretação correta dos diferentes elementos presentes é crucial para a recuperação das informações [STU 00].

Já Robin Cover introduz o conceito de “transparência semântica”. Dentro do contexto de processamento de informações baseada em XML, transparência semântica significa que tanto máquinas quanto seres humanos devem ser apresentados à informação de uma forma não ambígua (ou seja, com um significado preciso), e também de uma forma significativamente correta (ou seja, que satisfaça simultaneamente todo um conjunto de restrições de integridade). Isto é necessário para que agentes de *software* possam trocar informações entre si, já que a transparência semântica requer que qualquer “objeto de informação” seja formalmente especificado num nível de detalhamento em termos das suas características principais, relacionamentos e restrições de integridade. Com a transparência semântica, tanto agentes de *software* quanto seres humanos podem verificar o significado e a fidelidade de informações codificadas em XML [COV 99].

Mas o próprio Robin Cover desmente que a linguagem XML tenha por si só a capacidade de garantir transparência semântica: “XML não tem nenhum mecanismo formal para suportar declarações de restrições de integridade sobre primitivas semânticas (como ontologias ou modelos relacionais), e processadores XML não tem meios de validar a semântica mesmo que ela esteja declarada informalmente numa DTD. A XML governa somente sintaxe”. [COV 99]

A afirmação de Robin Cover pode ser completada por Studer: “as aplicações não devem somente tomar cuidado com a DTD que define uma classe de documentos, elas devem também ser informadas sobre a semântica básica das marcações e da estrutura do documento”. Uma solução para acessar diferentes tipos de documentos XML é através do uso de ontologias. Uma ontologia, como já foi detalhado, provê um meio formal e compartilhado de especificar os conceitos e relações de um determinado domínio. As ontologias definem semântica num nível conceitual. Através de uma ontologia é possível elevar a informação de um nível meramente sintático para um nível mais abstrato baseado em conceitos e relações. Com isso, representando-se a semântica de diferentes documentos XML baseando-se numa ontologia, é possível integrar e acessar estes documentos a partir da ontologia [STU 00].

Uma ontologia pode representar os elementos de diferentes documentos XML como um conjunto de conceitos, abstraindo assim as diferentes estruturas e sintaxes das diversas fontes. O processamento de uma consulta a um conjunto de documentos XML pode ser feito através deste conjunto de conceitos, deixando a estruturação das fontes heterogêneas transparente para o usuário da ontologia. Mais detalhes sobre a construção de ontologias a partir de fontes XML heterogêneas serão dados no capítulo 4.

3.2.2 Representando Ontologias com XML

Sendo uma ontologia uma representação formal do conhecimento, pode-se concluir que a sua representação depende de uma linguagem que seja capaz de reproduzir tal formalismo e que também seja flexível, tendo em vista que as ontologias não seguem nenhum padrão de representação. Diversas formas de representar ontologias já foram criadas, mas a alternativa de utilizar XML para tal tarefa vem se tornando comum.

Diversas motivações podem ser apontadas para o uso de XML na representação de ontologias. A principal delas é o fato de XML ser uma linguagem voltada para Internet, ou seja, voltada para a troca de informações. Desta forma, a representação de ontologias em XML facilita o trabalho de compartilhamento e distribuição.

Explica-se a importância do compartilhamento das ontologias: pode-se ter ontologias compartilhadas sendo utilizadas por diferentes grupos que precisam resolver os mesmos problemas, sem que tenham que projetá-las de novo. Uma ontologia compartilhada que represente a integração de diferentes bases de dados facilitaria o processamento de consultas a essas bases. Por último, o compartilhamento de ontologias é importante pois uma ontologia em si representa o conhecimento que uma determinada comunidade deseja compartilhar [KAR 99].

Representações de ontologias em XML poderiam então ser facilmente publicadas na *web* ou repassadas para aplicações da *web*, tendo em vista as facilidades da linguagem XML, como a simplicidade para o trabalho de *parsing*, uma sintaxe bem definida, ser legível pelo ser humano, além da tendência atual de tanto a linguagem quanto os aplicativos desenvolvidos para ela tornarem-se padrões [KAR 99, COV 99].

Dentre as propostas atuais de utilizar XML como linguagem para representar ontologias, destaca-se a linguagem XOL (*XML-Based Ontology Exchange Language*) [KAR 99]. A intenção dos seus desenvolvedores é permitir a permutação de ontologias pela *web*. Isto é, a intenção é de tornar XOL uma linguagem intermediária para transferir ontologias entre diferentes sistemas de bancos de dados, ferramentas de desenvolvimento de ontologias ou programas aplicativos em geral.

Karp apresenta alguns exemplos de uso para a linguagem XOL: um grupo de pesquisa desenvolvendo um banco de dados pode usar o SGBD da Oracle para implementa-lo. Todavia, o grupo pode traduzir o esquema do SGBD de SQL para XOL, e publicar o arquivo resultante, ou seja, a ontologia resultante, na sua página da *web* para ser usado como referência por seus usuários ou por outros grupos que desenvolvam o mesmo tipo de banco dados. Utilizando um aplicativo apropriado, eles transformaram a definição do esquema para XOL. Quando um outro grupo encontrar esta ontologia publicada, eles podem converter o modelo XOL para o SGBD preferido deles, ou para a ferramenta de desenvolvimento de ontologias preferida [KAR 99].

A fig. 3.2 abaixo apresenta um trecho de um documento XOL, extraído de [KAR 99], mostrando uma ontologia sobre genealogias:

```
<?xml version="1.0" ?>
<!DOCTYPE module SYSTEM "module.dtd">
<module>
        <name>genealogy</name>
        <kb-type>ocelot-kb</kb-type>
        <package>user</package>
        <class>
                <name>person</name>
                <documentation>The class of all persons</documentation>
        </class>

        <class>
                <name>man</name>
                <documentation> persons whose sex is male</documentation>
                <subclass-of>person</subclass-of>
        </class>

        <class>
                <name>woman</name>
                <documentation> persons whose sex is female.</documentation>
                <subclass-of>person</subclass-of>
      </class>
 ......
 ......

</module>
```

FIGURA 3.2 – Exemplo de um documento XOL

Ainda que este seja somente um trecho da ontologia, pode-se observar que o XOL trata os conceitos como classes. Primeiro foi definido uma super-classe, a classe “person” seguida de duas sub-classes: “man” e “woman”. A ontologia definiu então que todo objeto da classe “man” ou da classe “woman” são também da classe “person”. Em seguida, poderia-se definir as peculiaridades da classe “man” e “woman” e como elas podem se relacionar.

4 INTEGRAÇÃO DE ESQUEMAS COM ONTOLOGIAS

Esta seção irá apresentar a proposta principal deste trabalho, que é a de integrar esquemas XML, ou seja, as DTDs, utilizando ontologias. Em primeiro lugar, será apresentado uma breve base teórica sobre a integração de esquemas de bancos de dados. Em seguida será apresentado um resumo do projeto IBM-UFRGS, que vem estudando acesso integrado a bases de dados legadas, mais especificamente num caso da PROCERGS. Tendo o projeto IBM-UFRGS como motivação, será então detalhada a proposta de integrar esquemas XML, ou mais especificamente DTDs, utilizando ontologias como suporte.

4.1 Integração de Esquemas

Uma das necessidades mais recorrentes hoje em dia é a integração de diferentes recursos de sistemas de informação em empresas, organizações, etc. No instante que estes sistemas são baseados em aplicações que dependem dos dados gerados pela organização, a informação torna-se um recurso estratégico, que deve ser usada de uma maneira otimizada. Todavia, os diferentes sistemas, em termos de dados e modelos de processos, possuem um grau de heterogeneidade que torna difícil qualquer tentativa de compartilhar informações e serviços [BON 94].

Na área de bancos de dados, observa-se que diversas organizações tem usado a tecnologia de bancos de dados no gerenciamento de suas aplicações, desde as mais comuns até as de nível mais estratégico. Por isso, diversas pesquisas tem sido desenvolvidas para tornar mais fácil o compartilhamento de dados. Assim surgiu o conceito de integração e/ou interoperabilidade entre bancos de dados. Para dar suporte a esta integração, pode-se citar pelo menos três arquiteturas voltadas a distribuição de informações: bancos de dados distribuídos, bases de dados legadas e *multidatabases* [BAT 86, BON 94].

Em [BAT 86] e em [KAM 91] encontram-se análises e comparações entre vários métodos e arquiteturas para integração de esquemas e bancos de dados.

Um esquema em bancos de dados, segundo a definição de Silberchatz é o "projeto geral de um banco de dados", sendo que este projeto é alterado com pouca freqüência. [SIL 99]. Batini define integração de esquemas como sendo "a atividade de integração de esquemas de bancos de dados existentes ou propostos, num esquema global unificado" [BAT 86].

Um exemplo simples de como dois esquemas heterogêneos podem ser integrados pode ser construído a partir dos esquemas da fig. 4.1:

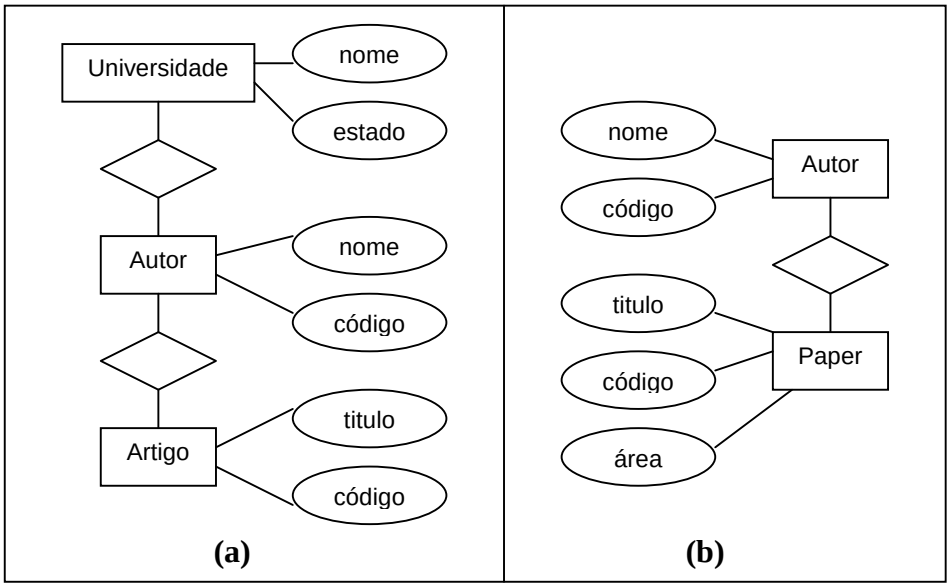


FIGURA 4.1 - Exemplo de dois esquemas heterogêneos

O esquema da fig. 4.1 (a) mostra três entidades: "Universidade", "Autor" e "Artigo". O esquema em (b) têm apenas duas entidades: "Autor" e *Paper*. Uma rápida análise prévia pode mostrar como integrar estes dois esquemas num só. Logo a principio percebe-se que os dois esquemas fazem uso de uma entidade chamada "Autor", sendo que em ambos os esquemas essas entidades tem o mesmo significado. Pode-se também perceber claramente que as entidades "Artigo" em (a) e *Paper* em (b) têm ambas o mesmo significado, apenas foram definidas com nomes diferentes. Duas decisões podem ser tomadas então: integrar as entidades "Autor" de (a) e "Autor" de (b), em uma só entidade que contenha todos os atributos e relacionamentos das entidades anteriores, e integrar as entidades "Artigo" de (a) e "Paper" de (b) em uma mesma entidade, sob o nome de uma das entidades anteriores.

A fig. 4.2 mostra como seria o esquema resultante da integração dos dois esquemas da fig. 4.1, tomando como base as decisões apontadas.

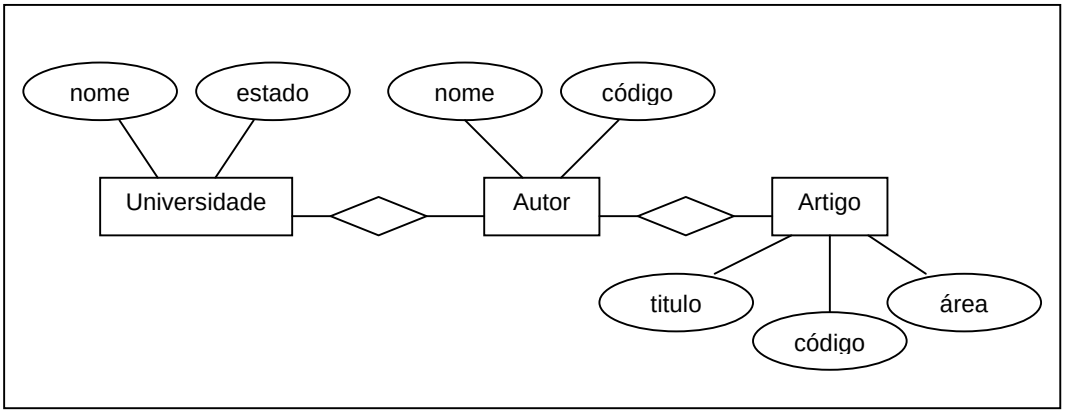


FIGURA 4.2 - Resultado da integração dos esquemas da fig. 4.1

A entidade "Artigo" ficou sendo o resultado da combinação entre "Artigo" do esquema (a) e "Paper" do esquema (b). Como pode-se notar, todos os atributos da entidade (a) e da entidade (b) foram agregados a esta entidade. Já a entidade "Autor", que estava presente nos dois esquemas, foi integrada diretamente neste esquema resultante. O relacionamento que "Autor" tinha com "Universidade" foi agregado na entidade resultante.

Analisando este exemplo, pode-se perceber duas características importantes quanto a integração de esquemas: as causas da heterogeneidade entre os esquemas, e o processo de integração em si. Estas características serão melhor discutidas a seguir.

4.1.1 Causas da diversidade entre esquemas

A tentativa de integrar diferentes esquemas mostra a existência de diferenças estruturais e semânticas. Segundo Batini, existem três possíveis causas para a diversidade entre esquemas: diferentes perspectivas, equivalência entre construções do modelo e especificações de projeto incompatíveis [BAT 86].

1. Diferentes perspectivas: diferentes projetistas tendem a ter diferentes visões da modelagem de um mesmo objeto de um domínio. Um exemplo disso são as entidades "Artigo" e "Paper" da fig. 4.1, que representam um mesmo conceito, apesar de terem nomes diferentes. Diferentes pontos de vista sobre um objeto do domínio podem produzir tanto diferenças semânticas quanto diferenças estruturais, como por exemplo, a criação de diferentes relacionamentos.
2. Equivalência entre construções do modelo: um mesmo domínio pode ter diferentes construções do seu modelo, ainda que sejam equivalentes. Dessa forma, determinadas situações podem ter diversas variações na forma de modelagem. Essas variações podem ser desde um esquema apresentar um atributo que em outro esquema foi modelado como um relacionamento até utilização de uma generalização em um esquema e de um atributo em outro, para a mesma situação.
3. Especificações de projeto incompatíveis: sempre pode acontecer de uma especificação conter erros, como má escolha de nomes, tipos, restrições, entre outros. Esses erros poderiam dificultar uma tentativa de integração de esquemas. É necessário distinguir dentro do domínio das aplicações um conjunto de conceitos que sejam comuns a todos os esquemas. Também é necessário distinguir um conjunto de diferentes conceitos dos diferentes esquemas que tenham algum relacionamento semântico, Batini chama isso de "propriedades inter-esquema".

Dois conceitos distintos, que representem o mesmo objeto de um domínio, em dois diferentes esquemas, podem ter quatro tipos de relacionamentos semânticos [BAT 86]. São eles:

- **Idênticos:** os dois conceitos são exatamente iguais. Isso só pode ser considerado se exatamente as mesmas construções forem usadas nas duas representações, e não existem diferenças entre as duas.
- **Equivalentes:** os dois conceitos não são exatamente o mesmo porque diferentes definições foram utilizadas na modelagem, ainda que estas definições sejam equivalentes. O significado de ambos ainda é coerente entre si.

- **Compatíveis:** os conceitos nesse caso não são nem idênticos nem equivalentes, mas as definições de modelagem, a percepção do projeto e as restrições de integridade não são contraditórias.
- **Incompatíveis:** nesse caso, os dois conceitos são totalmente contraditórios por incoerência entre suas especificações

4.1.2 O processo de integração

Existem diversas metodologias para integrar esquemas, a maioria delas lida com esquemas que descrevem dados armazenados em bancos de dados ou em arquivos, ou ainda utilizados por aplicações. Estas metodologias, em sua maioria, dividem o processo de integração em etapas, sendo que as principais são: pré-integração, comparação, integração e transformação de esquema [BON 94].

- **Pré-integração:** nesta fase é feita uma análise dos esquemas para a tomada de certas decisões antes da integração. As principais decisões são: os esquemas a serem integrados e a ordem em que serão integrados. Além disso, estratégias globais de integração, como o volume de interação com o projeto e o número de esquemas a ser integrado de uma vez só, também são decididas nesta fase [BAT 86]. Além disso, segundo Bonjour, é interessante também nesta fase que os esquemas a serem integrados sejam traduzidos para um "modelo canônico" que expresse claramente os conceitos e relacionamentos dos esquemas, numa forma que facilite as demais fases de integração [BON 94].
- **Comparação:** nesta fase os esquemas são comparados para determinar as possíveis correspondências entre os seus conceitos e detectar possíveis conflitos. A fase de comparação é geralmente feita de forma automatizada, utilizando-se probabilidades ou técnicas de comparação de padrões. Qualquer procedimento automatizado nesta fase deve ser acompanhada por um especialista, para aceitar ou rejeitar as sugestões feitas pelo sistema [BAT 86, BON 94].
- **Integração:** o processo de integração em si, consiste geralmente numa tarefa semi-automática, onde os esquemas integrados são construídos a partir dos esquemas originais (ou dos modelos canônicos originais), tendo como base as correspondências entre os esquemas e as regras de integração [BON 94].
- **Transformação de esquema:** os resultados da fase anterior devem agora ser analisados e re-estruturados. Os seguintes critérios devem ser avaliados nos esquemas resultantes: os esquemas integrados devem representar os conceitos dos esquemas originais de forma completa e correta; conceitos repetidos nos esquemas originais devem ter somente uma representação no esquema integrado e o esquema integrado deve ser fácil de entender tanto para o usuário quanto para o desenvolvedor
  [BAT 86].

4.2 O Projeto IBM-UFRGS

O objetivo desta seção é apresentar o projeto IBM-UFRGS de acesso integrado a bases de dados legadas [IBM 00], e mostrar em qual contexto deste projeto esta monografia está de certa forma inserida.

Bases de dados legadas são uma coleção de bancos de dados em que o compartilhamento de dados é feito de forma mais explícita através de esquemas exportados, que definem a parte compartilhada de cada base de dados local [BAT 86].

Na referência [IBM 00] encontra-se a especificação do projeto, que será resumida aqui. O projeto IBM-UFRGS tem como objetivos aplicar métodos e técnicas estudadas e desenvolvidas no Instituto de Informática da UFRGS sobre um caso prático na PROCERGS. Estas técnicas e métodos abrangem os seguintes problemas:

- Integração de fontes heterogêneas de dados
- Acesso a bases de dados legadas
- Gestão de documentos XML

A fig. 4.3, logo abaixo, mostra a arquitetura geral do projeto:

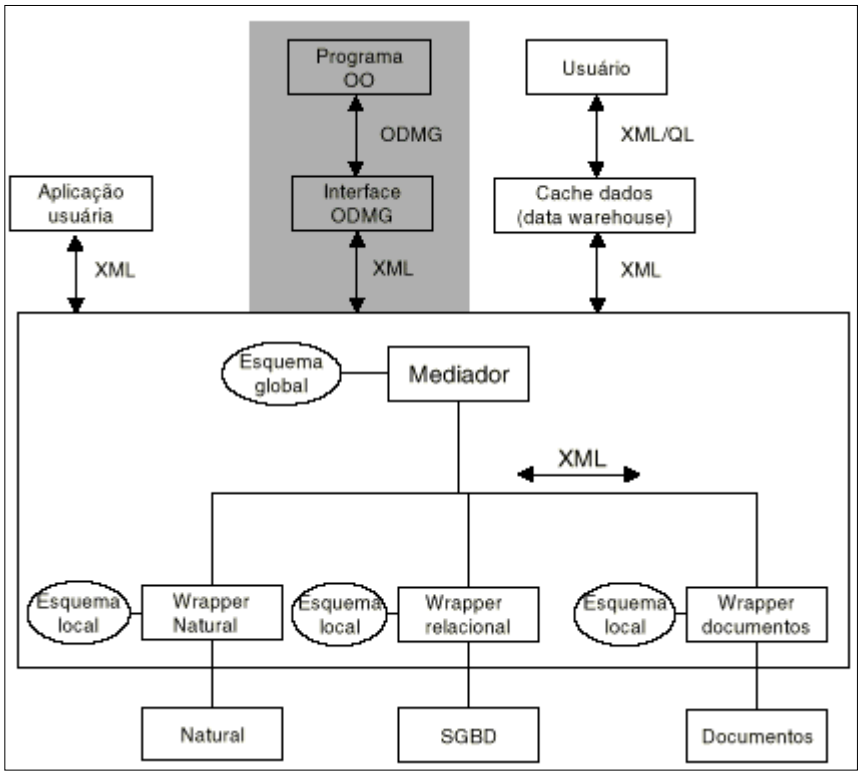


FIGURA 4.3- Arquitetura geral do projeto IBM-UFRGS

Na parte inferior da fig. 4.3, pode-se ver os três tipos de bases de dados que existem atualmente na PROCERGS: bancos de dados do sistema Natural, depois vários SGBDs relacionais e por último, uma base de documentos HTML e XML. O propósito aqui é acessar a informação compartilhada nessas bases de dados de uma maneira integrada. Para isso, utiliza-se a linguagem XML como uma tecnologia integradora, tendo em vista que com XML é possível representar dados acessados em bases de dados estruturadas e também de fontes semi-estruturadas, como documentos HTML.

Os *wrappers* mostrados na fig. 4.3 são ferramentas que, por conhecerem o modelo conceitual de uma base de dados, são responsáveis por atender as consultas e atualizações numa fonte de dados. O modelo conceitual utilizado pelo *wrapper* é representado pela elipse chamada "esquema local" na fig. 4.3. Os pedidos de atualização e consulta nas bases de dados são todas feitas em XML.

No centro da fig. 4.3 encontra-se um mediador. O mediador funciona aqui como integrador das diversas fontes heterogêneas. O mediador recebe as requisições das aplicações com base no esquema global. A partir daí ele têm a capacidade de fazer o mapeamento do esquema global para o esquema local dos *wrappers*. As requisições que o mediador recebe têm a mesma codificação XML que os *wrappers* também recebem.

O foco de atenção deste trabalho recai sobre os esquemas locais e globais. Eles representam a integração de vários esquemas existentes nas bases de dados. As próximas seções detalham melhor esta questão.

4.2.1 Esquemas locais

Os esquemas locais são mapeamentos de todos os "sub-esquemas" existentes numa determinada base de dados. Um esquema local é então um esquema integrado de todas os esquemas de um determinado tipo de fonte de dados. Na fig. 4.3 o esquema local, por exemplo, associado ao "*wrapper* documentos", é um esquema que representa a integração de todos os sub-esquemas de documentos XML e HTML existentes na base de dados documentos. A fig. 4.4 abaixo mostra isso melhor:

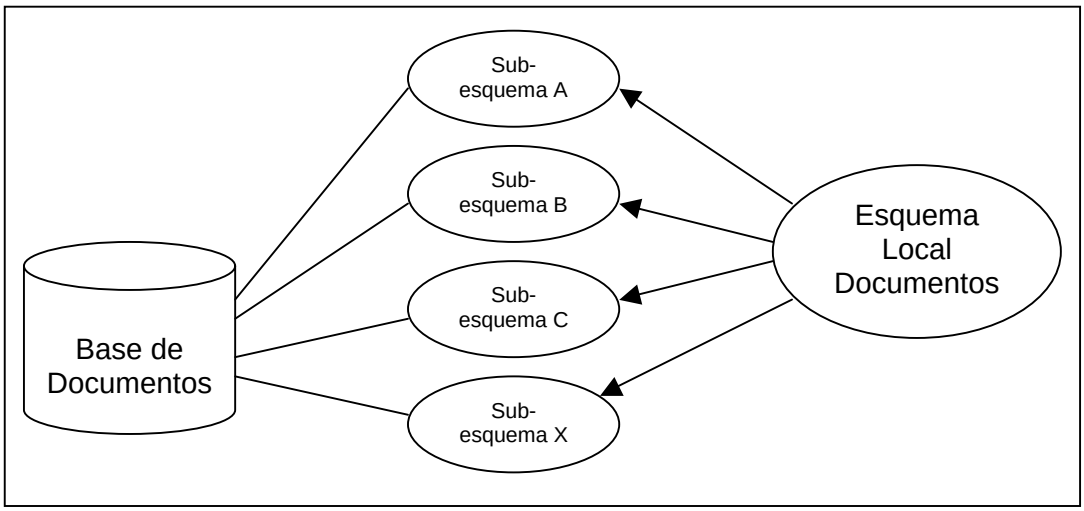


FIGURA 4.4 - Um esquema local integrando diferentes sub-esquemas

Os sub-esquemas aqui nada mais são do que os esquemas de cada fonte de dados existente. O esquema local é um esquema integrado de todos os sub-esquemas dessas fontes. No caso da base de dados de documentos, os sub-esquemas tratam-se de DTDs, tendo em vista que essa base é formada por documentos XML.

A mesma representação da fig 4.4, que é utilizada pelo *wrapper* documentos, também vale para os outros *wrappers*.

4.2.2 Esquema global

O esquema global é quem por fim integra todas as fontes de dados heterogêneas. O esquema global da fig. 4.3, pode ser melhor ilustrado na fig. 4.5 abaixo:

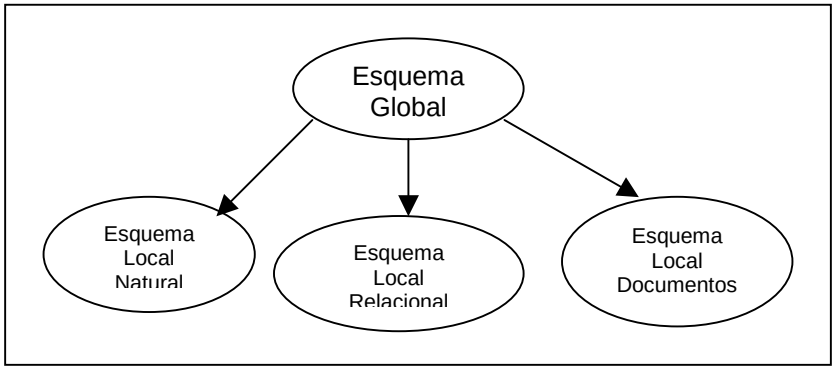


FIGURA 4.5 - Esquema global integrando diferentes esquemas locais

O esquema global é um esquema integrado de todos os esquemas locais, e consequentemente, todos os sub-esquemas existentes nas diferentes bases de dados. Por isso o mediador da fig. 4.3 pode utilizar este esquema global para enviar requisições de atualização e consulta para as bases de dados referentes.

4.2.3 Integração de DTDs utilizando ontologias

Como foi mostrado nas seções anteriores, os esquemas integrados são uma parte importante da arquitetura do projeto IBM-UFRGS. Os esquemas locais e global servem para tornar o acesso às fontes heterogêneas transparente para as aplicações, sendo utilizados pelo mediador e pelos *wrappers* para isso. No caso especifico da base de documentos, o esquema local é construído a partir de diversos sub-esquemas XML, ou seja, de DTDs.

No capítulo 2 foi discutido que as ontologias são uma metodologia formal para representar o conhecimento sobre um domínio, e que é possível tanto utilizar ontologias para acessar documentos XML quanto utilizar XML para representar ontologias.

O que se propõe aqui é a utilização de ontologias no formato XML para representar os esquemas locais e globais. É possível tomar as DTDs da base de documentos e construir a partir de um processo de integração, uma ontologia que as integre e que seja representada formalmente. Esta ontologia resultante seria o próprio esquema local mostrado na fig. 4.3. As ontologias aqui podem ser consideradas como o "modelo canônico" citado na seção 4.1.2, que irá detalhar todos os conceitos e relacionamentos existentes nos esquemas, ou neste caso, das DTDs.

Utilizando ontologias, seria possível construir uma *interface* de consulta que integre todas as bases de dados. Um processador de consulta poderia avaliar a ontologia e descobrir a origem de cada conceito representada nela e trazer a informação de volta para o usuário.

E ainda existe a possibilidade de os esquemas relacionais e da base de dados Natural serem mapeados para DTDs, então esses esquemas também poderiam ser integrados em ontologias. E por fim, seria possível utilizar o mesmo processo de integração para construir o esquema global, que seria uma ontologia mais abrangente que as demais.

A partir desta idéia, na próxima seção será mostrado como integrar DTDs em ontologias.

4.3 Ontologias e Integração

Nesta seção será especificado como foi projetada a integração de DTDs utilizando ontologias. O processo de integração, de uma forma geral, pode ser visto na fig. 4.6:

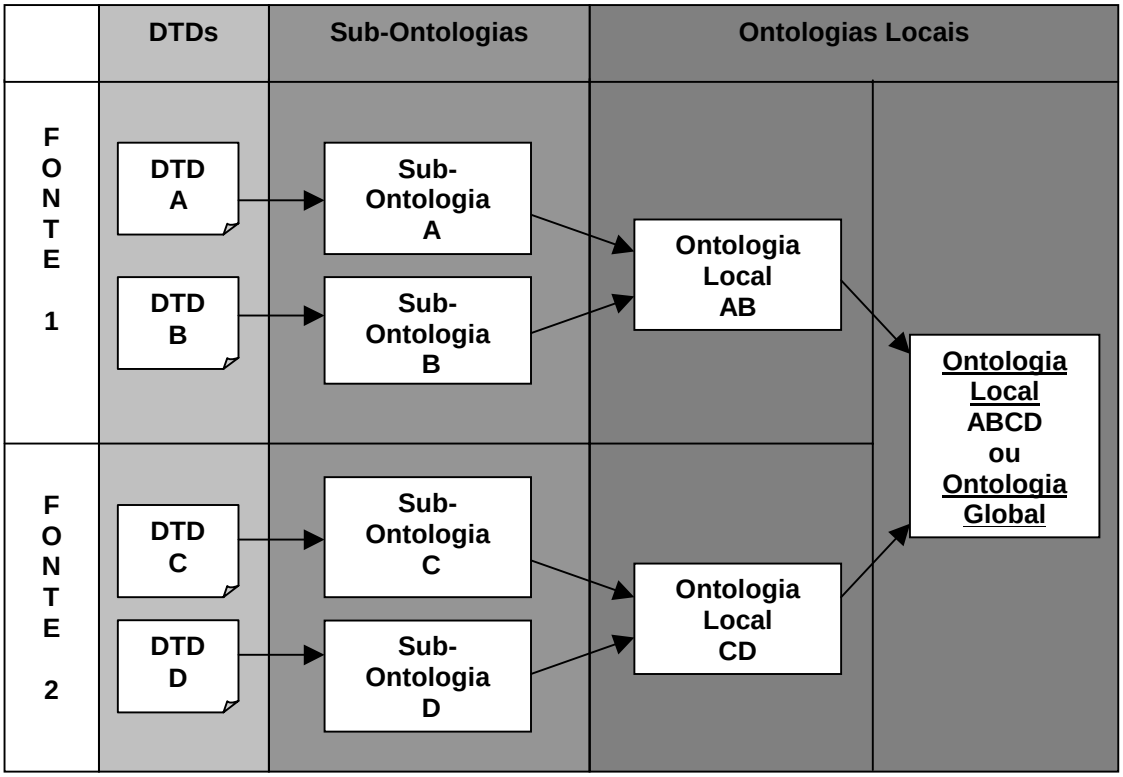


FIGURA 4.6 - Visão geral do processo de integração

Na fig. 4.6 estão definidas três "camadas". Num primeiro nível tem-se uma base de DTDs pertencentes a uma determinada fonte (uma Universidade, um departamento, etc..), logo em seguida, as DTDs passam por um processo de mapeamento para tornarem-se sub-ontologias, mostradas aqui como uma camada intermediária. As sub-ontologias de cada fonte são integradas, passando para a camada de ontologias locais.

Daqui em diante, as ontologias locais podem ser integradas entre si infinitamente, tantas vezes quanto existirem outras ontologias locais. Num dado momento, como mostrado na fig. 4.6, pode-se determinar, devido ao seu nível de abrangência, que um ontologia local seja considerada uma ontologia "global" por representar todas as fontes participantes. Esta ontologia global é uma ontologia local como qualquer outra, a distinção entre "local" e "global" é puramente relativa ao ponto de vista da aplicação. Por exemplo, na fig. 4.6, a ontologia local "CD" pode ser considerada como a ontologia global da fonte 2, se o ponto de vista do desenvolvedor estiver voltado apenas para a fonte 2.

As próximas seções mostram mais detalhadamente o formato da ontologia, detalhando-se cada componente da mesma. As características das sub-ontologias e das ontologias locais será também detalhado a partir da seção 4.3.8.

4.3.1 O formato da ontologia

As ontologias não têm um formato padrão, como foi discutido no capítulo 2. A estrutura apresentada aqui têm como propósito atender a uma necessidade específica, que é a de integrar esquemas de diferentes fontes de dados. A fig. 4.7 mostra através de um modelo de dados como foi projetada a ontologia:

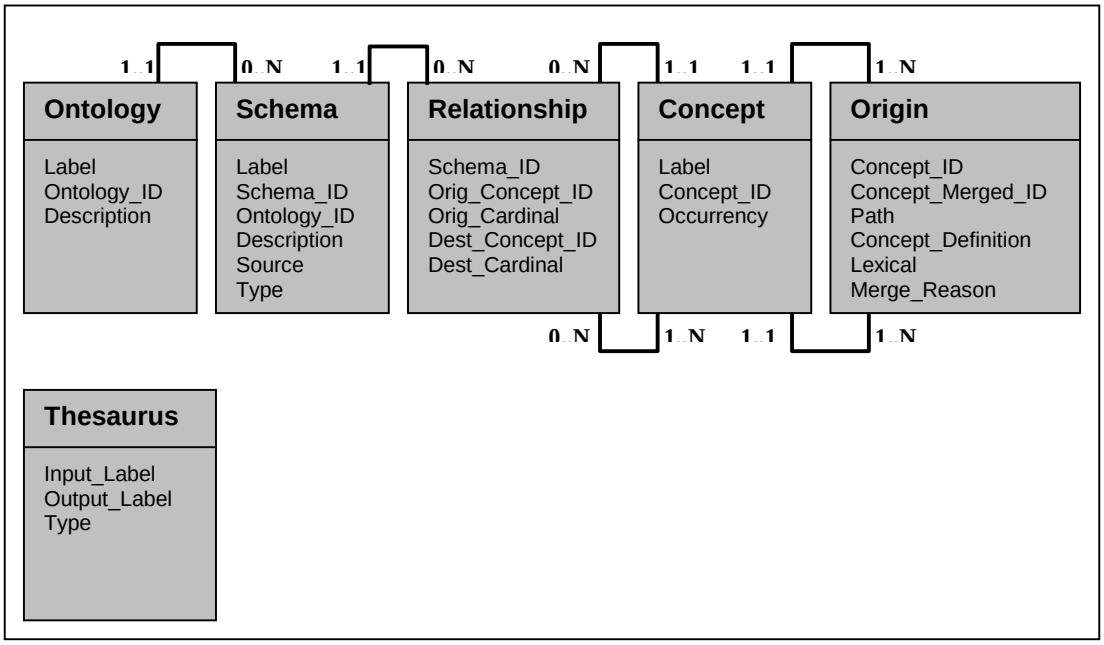


FIGURA 4.7 - Modelo de dados do formato das ontologias

Uma rápida interpretação do modelo de dados da fig. 4.7 diz que uma ontologia é formada por vários esquemas. Estes esquemas são as sub-ontologias e ontologias locais. Cada esquema é constituído por um conjunto de relacionamentos, sendo que cada relacionamento relaciona dois conceitos, um origem e um destino. E por fim, um conceito pode estar vinculado a mais de uma origem. O *thesaurus* é nada mais que uma tabela listando diversos sinônimos conhecidos, que serve para auxiliar no processo de integração.

O modelo de dados mostrado na fig. 4.6 serve para uma melhor compreensão de como se relacionam os diversos elementos da ontologia. A partir deste modelo pode-se melhor entender como a ontologia pode ser construída e corretamente utilizada.

Agora já é possível visualizar a DTD da ontologia. A fig. 4.8 abaixo mostra como foram declarados os elementos que constituem a ontologia:

```
<!ELEMENT Ontology (Label, Ontology_ID, Description, Schema*) >
<!ELEMENT Label (#PCDATA) >
<!ELEMENT Ontology_ID (#PCDATA) >
<!ELEMENT Description (#PCDATA) >
<!ELEMENT Schema (Label, Schema_ID, Source, Type, Concept*, Relationship*, Origin*)>
<!ELEMENT Schema_ID (#PCDATA) >
<!ELEMENT Source (#PCDATA) >
<!ELEMENT Type (#PCDATA) >
<!ELEMENT Concept (Label, Concept_ID, Occurrency) >
<!ELEMENT Concept_ID (#PCDATA) >
<!ELEMENT Occurrency (#PCDATA) >
<!ELEMENT Relationship (Schema_ID, Orig_Concept_ID, Orig_Cardinal, Dest_Concept_ID, Dest_Cardinal) >
<!ELEMENT Orig_Concept_ID (#PCDATA) >
<!ELEMENT Orig_Cardinal (#PCDATA) >
<!ELEMENT Dest_Concept_ID (#PCDATA) >
<!ELEMENT Dest_Cardinal (#PCDATA) >
<!ELEMENT Origin (Concept_ID, Concept_Merged_ID, Path, Concept_Definition, Lexical, MergeReason) >
<!ELEMENT Concept_Merged_ID (#PCDATA) >
<!ELEMENT Path (#PCDATA) >
<!ELEMENT Concept_Definition (#PCDATA) >
<!ELEMENT Lexical (#PCDATA) >
<!ELEMENT MergeReason (#PCDATA) >
```

FIGURA 4.8 - DTD com o formato da ontologia para integração

Como pode-se ver na fig. 4.8, os elementos *Relantionship*, *Concept* e *Origin* estão aninhados dentro de *Schema* que por sua vez está aninhado em *Ontology*.

Nas próximas seções serão melhor detalhados os principais elementos da ontologia.

4.3.2 O elemento *Ontology*

A elemento *Ontology* representa a ontologia em si. A ontologia tem três informações importantes:

- **Label:** um nome para designar esta ontologia;
- **Ontology_ID:** o ID da ontologia será um número que funcionará como identificador;
- **Description:** um texto que descreva adequadamente a ontologia em questão.

Na DTD da ontologia, O elemento *Ontology* está declarada como o elemento raiz do documento XML, será formado por zero ou muitos elementos *Schema*. Esta agregação de esquemas é que pode ser considerado o "corpo" da ontologia.

4.3.3 O elemento *Schema*

Os esquemas em questão aqui são as sub-ontologias e ontologias locais que compõem a ontologia em si. Tendo em vista que tanto as sub-ontologias quanto as ontologias locais têm a mesma estrutura, elas foram de certa forma "generalizadas" na entidade *Schema*.

As informações mais relevantes de *Schema* são:

- **Ontology_ID:** o ID da ontologia onde este esquema foi gerado;
- **Schema_ID:** o número de identificação do esquema em si.
- **Source:** contém o caminho ou URL da fonte DTD que deu origem ao esquema em questão;
- **Type:** o tipo de esquema, ou seja, se trata-se de uma sub-ontologia ou de uma ontologia local/global;

Dentro do elemento *Schema* estão aninhados todos os relacionamentos, conceitos e origens que o compõem.

4.3.4 O elemento *Concept*

Os elementos *Concept* descrevem cada conceito presente numa ontologia. Estes elementos trazem as seguintes informações:

- **Label:** o rótulo do conceito, que de certa forma irá expressar o significado do mesmo;
- **Concept_ID:** o número de identificação do conceito;
- **Occurrency:** o número de ocorrências do conceito, isto é, depois de sucessivas integrações, quantas vezes este conceito já apareceu em diferentes esquemas.

Toda vez que um conceito for integrado com outro, o valor do seu elemento *Occurrency* deve ser incrementado.

4.3.5 O elemento *Relantionship*

Os elementos *Relationship* enumeram todos os relacionamentos entre conceitos existentes numa ontologia. Um elemento deste tipo irá trazer, além do ID do *Schema* a que pertence, as seguinte informações:

- **Orig_Concept_ID:** o ID do conceito que é a origem do relacionamento;
- **Orig_Cardinal:** a cardinalidade em relação ao conceito origem do relacionamento;
- **Dest_Concept_ID:** o ID do conceito que é o destino do relacionamento;
- **Dest_Cardinal:** a cardinalidade em relação ao conceito destino do relacionamento.

A fig. 4.9, logo abaixo, exemplifica como seria representado um relacionamento:

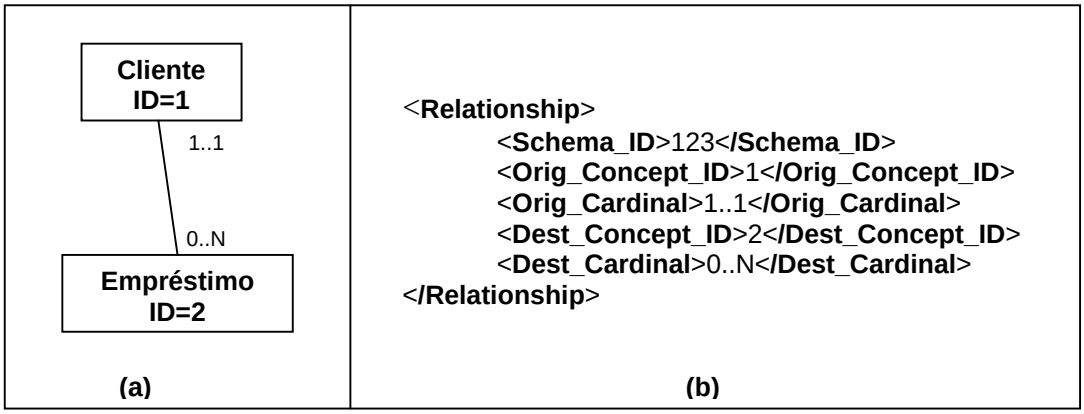


FIGURA 4.9 - Exemplo de relacionamento numa ontologia

A fig. 4.9(a) mostra um trecho de uma ontologia em forma de diagrama, mostrando dois conceitos, "Cliente" e "Empréstimo" e um relacionamento com as respectivas cardinalidades entre os dois conceitos. Em (b) pode-se ver como seria representado o relacionamento através do elemento *Relationship*. Os conceitos são representados pelos seus IDs, sendo o valor 1 para Cliente e o valor 2 para Empréstimo.

4.3.6 O elemento *Origin*

Os elementos *Origin* desempenham um papel importante na integração das ontologias. Como pode-se ver no modelo de dados da fig. 4.7, um conceito pode ter uma ou mais origens relacionadas a si. A origem na ontologia descreve de onde veio o conceito, podendo informar qual era a sua DTD de origem ou com qual outro conceito ele foi integrado.

Para isso, o elemento *Origin* traz as seguinte informações:

- **Concept_ID:** o ID do conceito que o elemento *Origin* representa;
- **Concept_Merged_ID:** o ID do conceito que foi integrado com o conceito representado, caso ele tenha sido integrado com alguém;
- **Path:** o caminho, ou URL da DTD que originou o conceito;
- **Concept_Definition:** uma breve definição do conceito;
- **Lexical:** identifica se o conceito é "léxico" ou "não-léxico";
- **Merge_Reason:** a razão que fez com que o conceito fosse integrado com outro.

O valor de *Lexical* determina se o conceito originalmente era um elemento "léxico", quer dizer, se ele era um elemento do tipo "PCDATA" na DTD, ou por outro lado, era "não-léxico", ou seja, tinha outros elementos aninhados a si.

O elemento *Merge_Reason* pode ter os valores: *UserSynonym*, se a integração foi determinada pelo usuário, que considerou os conceitos combinados como sinônimos; *ThesaurusSynonym*, se a integração foi determinada automaticamente, tendo em vista que os dois conceitos combinados estavam definidos como sinônimos no *thesaurus* do sistema, *AutomaticMerge*, se a integração foi determinada automaticamente pelo fato dos dois conceito terem *labels* iguais; ou *NoMergeAccomplished*, caso o conceito não tenha sido combinado com nenhum outro.

4.3.7 O *Thesaurus*

O *thesaurus* é um pequeno dicionário de sinônimos que é utilizado para estabelecer as equivalências entre conceitos. Também foi utilizado XML para estruturar o *thesaurus* e armazená-lo como um arquivo texto em anexo, tornando-o disponível para o desenvolvedor editá-lo.

A fig. 4.9 mostra a DTD para o *thesaurus*:

```
<!ELEMENT Thesaurus (Synonym*) >
<!ELEMENT Synonym (InputLabel, OutputLabel, Type) >
<!ELEMENT InputLabel (#PCDATA) >
<!ELEMENT OutputLabel (#PCDATA) >
<!ELEMENT Type (#PCDATA) >
```

FIGURA 4.9 - DTD para o *Thesaurus* das ontologias

O *thesaurus* é formado por diversos sinônimos, onde cada sinônimo têm um *InputLabel* e um *OutputLabel*, além de elemento que determina o seu tipo. Os tipos podem ser *UserSynonym*, para os sinônimos adicionados pelo usuário e *ThesaurusSynonym*, para os sinônimos próprios do *thesaurus.*

4.3.8 Sub-ontologias

As aqui chamadas sub-ontologias são na verdade um mapeamento direto de DTDs para o formato de ontologias. As sub-ontologias sozinhas não chegam a ter uma utilidade. Sua real função é servir como uma camada intermediária entre as DTDs e as ontologias locais. É muito mais simples integrar sub-ontologias numa ontologia local do que integrar diretamente DTDs para uma ontologia. O formato de uma sub-ontologia é o mesmo de uma ontologia local, o que muda de uma para outra é a forma como cada uma é criada e o seu grau de abrangência.

A fig. 4.7 abaixo mostra diferentes DTDs sendo mapeadas para sub-ontologias:

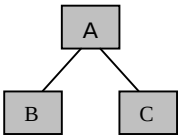
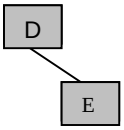
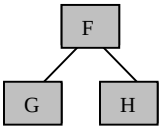
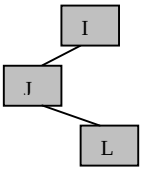
DTD	Sub-Ontologia
<!ELEMENT **A** (**B**, **C**) > <!ELEMENT **B** (#PCDATA) > <!ELEMENT **C** (#PCDATA) >	
<!ELEMENT **D** (**E**) > <!ELEMENT **E** (#PCDATA) >	
<!ELEMENT **F** (**G**, **H**) > <!ELEMENT **G** (#PCDATA) > <!ELEMENT **H** (#PCDATA) >	
<!ELEMENT **I** (**J**) > <!ELEMENT **J** (**L**) > <!ELEMENT **L** (#PCDATA) >	

FIGURA 4.7 - DTDs sendo mapeadas para sub-ontologias

Por um motivo de simplicidade, as sub-ontologias estão sendo mostradas nesta figura na forma de grafos, como na fig. 3.1 do capítulo 2. Cada quadrado em tom cinza representa um conceito da ontologia, enquanto que os traços representam os relacionamentos entre conceitos. Outro componente importante da ontologia que não pode ser visto no grafo é a origem do conceito.

A construção de uma sub-ontologia é baseada na análise de uma DTD, ou mais especificamente, das declarações de elementos presentes na DTD. Cada elemento declarado na DTD será considerado como um conceito na sub-ontologia, e terá uma origem, baseada nas informações disponíveis sobre a DTD. A partir do modelo de conteúdo das declarações, é possível estabelecer como os conceitos estão relacionados. A fig. 4.8 logo abaixo ,mostra um algoritmo genérico para construir sub-ontologias partir de DTDs:

```
01: Procedimento criaSubOntologia
02:     Para cada elemento na DTD
03:             CriaConceito(elemento);
04:     CriaOrigens;
05:
06: Procedimento CriaConceito(elemento)
07:     Se o elemento já não foi criado
08:             Novo Conceito(elemento.Nome);
09:             Para cada elemento relacionado
10:                     Novo Relacionamento(conceito, CriaConceito(elementoRelac.);
11:                     ResolveCardinalidade do relacionamento;
12:                     Adiciona relacionamento;
13:             Adiciona conceito;
14:     Fim Se ;
15:     Retorna conceito;
16:
17: Procedimento CriaOrigens
18:     Para cada conceito na subOntologia
19:             Nova Origem(conceito)
```

FIGURA 4.8 - Algoritmos para construir sub-ontologias

Obviamente este algoritmo está apenas num nível conceitual, servindo como uma ilustração do real processo de criação de sub-ontologias. Na lin. 08, a chamada para "Novo Conceito" irá criar um conceito com o mesmo nome do elemento declarado na DTD. Em seguida, é analisado o modelo de conteúdo do elemento para determinar os relacionamentos. Caso o elemento não seja do tipo "PCDATA" ou nulo, isto é, caso ele tenha outros elementos aninhados a ele, será então criado um novo relacionamento para cada elemento presente. Na lin. 10, a chamada de "Novo Relacionamento" toma como parâmetros os dois conceitos que estão relacionados. O primeiro conceito será o que acabou de ser criado, e o segundo será criado com uma chamada recursiva para "criaConceito". Assim, recursivamente o algoritmo irá descendo na árvore de elementos da DTD e criando os conceitos e relacionamentos. Ao final, na lin. 4, são criadas as origens de cada conceito.

Uma última consideração aqui é quanto as cardinalidades, que são resolvidas na lin. 11. Elas são na verdade um mapeamento dos sinais de ocorrência das declarações de elementos (seção 2.4.1) das DTDs. Essa informação deve ser explicitada nos relacionamentos. Abaixo pode-se ver como cada símbolo de ocorrência será mostrado na ontologia como uma cardinalidade:

- **? : 0..1**
- *** : 0..N**
- **+ : 1..N**
- **nenhuma marca: 1..1**

Os relacionamentos trazem duas cardinalidades: a de origem e destino. As marcas de ocorrência equivalem às cardinalidades de destino numa DTD, mas não é possível determinar com certeza qual será a cardinalidade destino. Por exemplo, seja a seguinte declaração de elemento:

```
<!ELEMENT Filme (categoria, ator+) >
```

Os relacionamentos do conceito "Filme" com "categoria" e "ator" teriam cardinalidades de destino, respectivamente, "1..1" e "1..N". Mas não é possível saber ao certo qual é a cardinalidade de origem, sabe-se que o valor mínimo é 1, mas faltam informações para determinar o valor máximo da cardinalidade. Por convenção, as cardinalidades de origem são todas marcadas como "1..?". Esta notação visa chamar a atenção do usuário para a cardinalidade que precisa ser revisada. Existe aqui a possibilidade desta cardinalidade ser completada automaticamente, caso durante um processo de integração outro esquema contiver os mesmos conceitos relacionados em ordem inversa, bastaria completar o valor que falta na cardinalidade de origem com o valor da cardinalidade de destino do outro esquema.

4.3.9 Ontologias locais

As ontologias locais são as ontologias resultantes da integração de sub-ontologias ou de outras ontologias locais. Durante a construção de uma ontologia local é que a integração realmente ocorre. Durante esta fase serão combinados os conceitos que possuam alguma equivalência. Esta combinação de conceitos é que irá dar forma a ontologia local. Existem três casos em que um conceito poderá ser combinado com outro:

1. Os conceitos têm a mesma definição, isto é, foram definidos com o mesmo rótulo, e provavelmente têm o mesmo significado;
2. Os conceitos são sinônimos, ou seja, mesmo com nomes diferentes, têm significados equivalentes. Neste caso, os rótulos dos dois conceitos estão definidos no *thesaurus* como sinônimos;
3. O desenvolvedor pode estabelecer que dois conceitos são semelhantes, mesmo que nenhum dos dois casos anteriores tenha ocorrido, ou ao contrário, ele pode desfazer uma combinação conflitante. Esta interação do desenvolvedor é necessária por dois motivos: primeiro, mesmo que o sistema de integração automatizada utilize um *thesaurus*, este nunca será completo, e segundo, a automatização não é perfeita, já que sempre poderá haver o caso de conceitos homônimos, por exemplo.

Tomando-se a fig. 4.7, pode-se imaginar algumas equivalências entre os conceitos das sub-ontologias:

- **B ≅ D**
- **B ≅ H**
- **G ≅ J**
- **C ≅ I**

A fig. 4.8 mostra como seria se as sub-ontologias da fig. 4.7 fossem integradas tendo em vista as equivalências imaginadas:

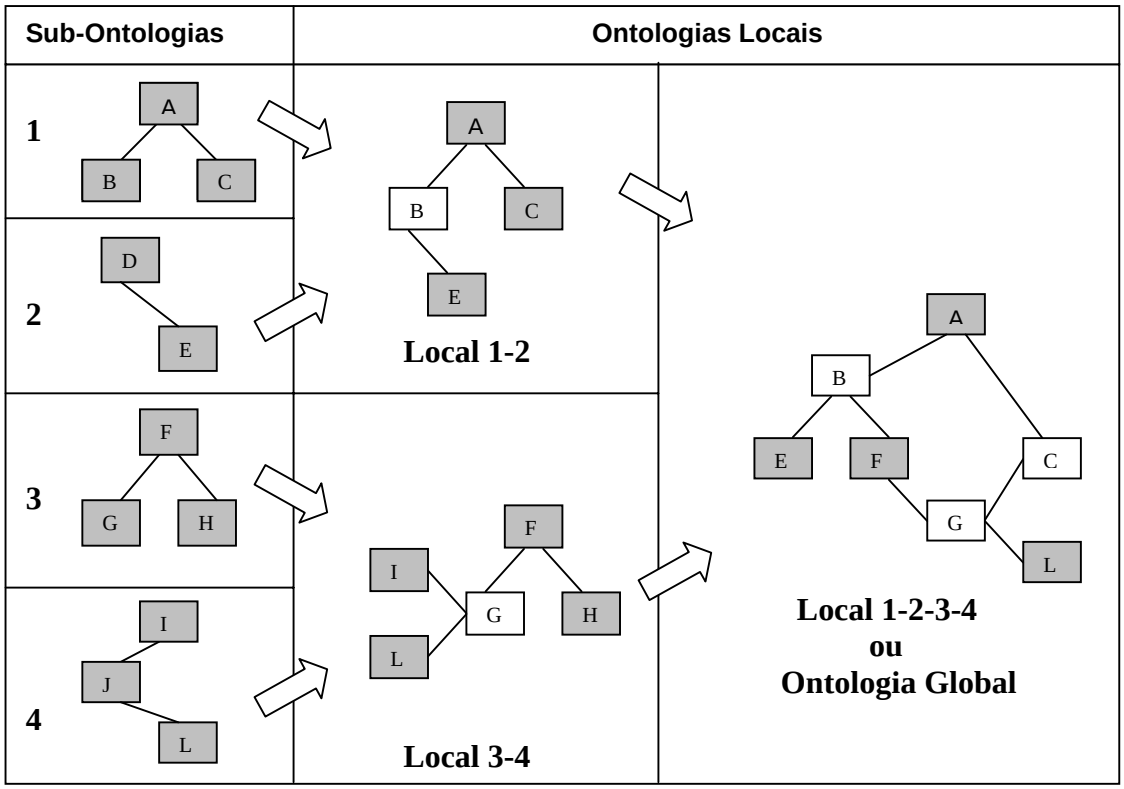


FIGURA 4.8 - Exemplo de geração de ontologias locais a partir de sub-ontologias

A fig. 4.8 mostra que foram integradas as sub-ontologias 1 e 2, gerando a ontologia local 1-2. Em seguida foram integradas as sub-ontologias 3 e 4, gerando a ontologia local 3-4. Por fim, as duas ontologias locais foram integradas numa única ontologia.

A ontologia local 1-2 mostra o conceito B numa cor diferente, tendo em vista que este conceito é uma combinação dos conceitos B e D, e neste caso, permaneceu o nome B. O mesmo vale para o conceito G da ontologia local 3-4 e para o conceito C da ontologia global, todos esses conceitos foram combinados a partir das equivalências estabelecidas anteriormente. Um conceito combinado deve conter todos os relacionamentos dos seus conceitos originais, como o conceito B da ontologia local 1-2, que por ser a combinação de B e D acabou "herdando" o relacionamento com conceito E.

Com já foi dito, a construção de uma ontologia local baseia-se na integração de dois ou mais esquemas, ou melhor dizendo, de sub-ontologias ou ontologias locais. Um possível algoritmo para a construção de ontologias locais é o seguinte:

```
01: Procedimento criaOntologiaLocal
02:     Para cada esquema (sub-ontologia ou ontologia local)
03:         Para cada relacionamento do esquema
04:             Se possível combinar conceito-orig com conceito-existente
05:                 Combina(conceito-origem, conceito-existente);
06:             Senão Adiciona conceito-origem;
07:             Se possível combinar conceito-dest com conceito-existente
08:                 Combina(conceito-destino, conceito-existente);
09:             Senão Adiciona conceito-destino;
10:             Verifica cardinalidades;
11:         Fim Para;
12:         Adiciona Relacionamentos;
13:         Adiciona Origens;
14:     Fim Para;
15:
16: Procedimento Combina(conceito, conceito-existente);
17:     Atualiza conceito-existente;
18:     Atualiza Relacionamentos;
19:     Nova Origem(conceito-existente, conceito);
20:     Descarta conceito;
```

FIGURA 4.9 - Algoritmo para construção de ontologias locais

O algoritmo da fig. 4.9 toma um esquema de cada vez para ser integrado (lin 02) e ir formando a nova ontologia local. Depois, serão testados cada conceito participante a partir dos relacionamentos, lembrando que um relacionamento inclui sempre dois conceitos: um origem e um destino.

Os conceitos a serem adicionados são testados sobre todos os outros conceitos já agregados à ontologia local para determinar alguma situação que provoque a combinação de conceitos. Quando a combinação de dois conceitos é possível, um dos conceitos será "descartado" na ontologia local, pois suas informações agora pertencerão ao conceito integrado, como no exemplo da fig. 4.8, em que o conceito B e o conceito D foram combinados. Neste caso, o conceito D não aparece mais na ontologia local, mas o seu relacionamento com o conceito E ainda existe, pertencendo agora ao conceito B. Tendo a combinação de B com D da fig. 4.8 como exemplo, pode-se visualizar como o algoritmo da fig. 4.9 procedeu para integrar os dois conceitos:

1. O conceito B é atualizado, seu número de ocorrências é incrementado (lin 17);
2. Todos os relacionamentos que tinham D como participante devem ser atualizados. No caso, o relacionamento D com E será alterado, passando a ser B com E (lin. 18);
3. Uma nova origem é criada para o conceito integrado (lin 19), sendo que o valor de *Concept_ID* desta origem será o ID de B, e o valor de *Concept_Merged_ID* será o ID de D.
4. O conceito D não terá mais utilidade na nova ontologia local, podendo ser descartado da mesma (lin 20).

Quando um conceito a ser adicionado não tem nenhuma possibilidade de integração, ele é adicionado sem nenhuma alteração (lin 06 e lin. 09). Após todos os conceitos terem sido testados e adicionados à ontologia local é que então os relacionamentos e as origens, que foram atualizadas (ou não) em decorrência das integrações, podem ser adicionadas.

Na lin. 10 encontra-se um teste para uma possível situação de colisão de cardinalidades num caso de integração. Esta situação está ilustrada na fig. 4.10:

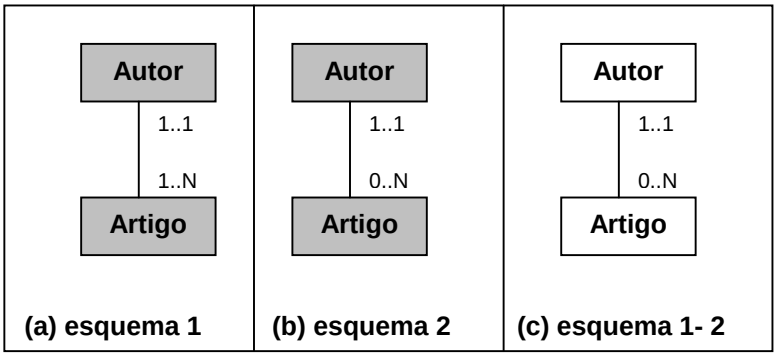


FIGURA 4.10 - Exemplo de conflito de cardinalidades numa integração de esquemas

O que a fig. 4.10 mostra são dois esquemas aparentemente idênticos, mas que foram definidos com diferentes cardinalidades, no caso "1..N" em (a) e "0..N" em (b). Uma tentativa de integrar esses dois esquemas têm que lidar com esse conflito, escolhendo entre as duas cardinalidades a que seja mais abrangente, como pode ser visto no esquema resultante (c) da integração de (a) e (b).

Para finalizar a explanação sobre a integração e construção das ontologias, é importante ressaltar uma característica importante, que é o estado final das origens dos conceitos integrados. Tomando ainda como exemplo os conceitos B e D da fig. 4.8, a fig. 4.11 ilustra as origens relacionadas ao conceito resultante da integração de B e D:

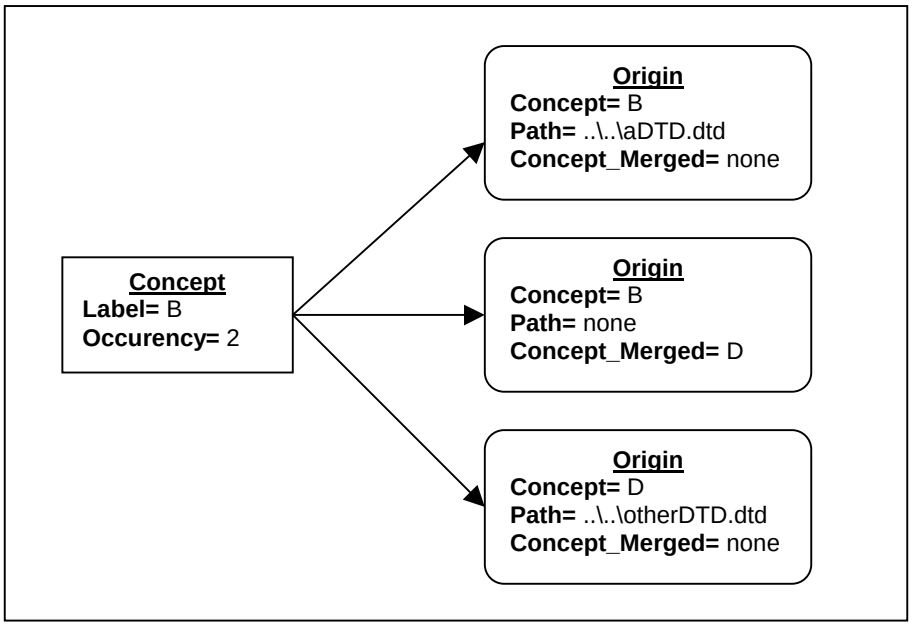


FIGURA 4.11 - Exemplo de um conceito e suas origens relacionadas

O conceito B da ontologia local 1-2 da fig. 4.8 terá três origens relacionadas a si. A primeira é a origem do conceito B da sub-ontologia 1, esta origem diz qual a DTD em que este conceito se originou. A segunda origem é a do conceito B integrado, ou seja o conceito que representa B (no valor de *Concept*) e D (no valor de *Concept_Merged*). Por fim, a última origem é a do conceito D, que não mais aparece na ontologia local, mas que está agora integrado ao conceito B, da mesma forma, a origem diz em qual DTD o conceito D se originou. Para realizar uma consulta nas fontes de dados a partir da ontologia local, estas origens são imprescindíveis, pois dizem que B equivale a D, e onde encontrar os originais B e D.

5 ESPECIFICAÇÃO DO PROTÓTIPO JONTOLOGY

Neste capítulo será detalhado como está sendo projetado e implementado o protótipo JOntology, que tem como objetivo principal efetuar a integração de esquemas descrita no capítulo anterior. A implementação da ferramenta JOntology está sendo feita na linguagem Java versão 1.3.0 [SUN 99]. Sendo esta linguagem orientada a objetos, sua especificação utilizará diagramas definidos pela linguagem UML (*Unified Modeling Language)* [OMG 99], apropriados para o desenvolvimento orientado a objetos.

Na especificação a seguir, serão mostradas as principais classes que estão sendo construídas, assim como as principais associações entre elas. Serão mostrados apenas os principais processos que o protótipo desempenha. Por razões de simplicidade, detalhes de menor importância serão omitidos ou apenas citados.

5.1 Requisitos do Software

Os principais objetivos do software JOntology estão relacionados abaixo:

1. Fazer o mapeamento de ontologias a partir das DTDs da linguagem XML;
2. Integrar diferentes ontologias construídas a partir do objetivo 1;
3. Resolver problemas de integração, como conflitos de cardinalidades e resolução de sinônimos;
4. Permitir a visualização das ontologias num formato de grafo;
5. Permitir ao usuário editar e corrigir as ontologias, tendo em vista o processo semi-automático de integração;
6. Armazenar as ontologias resultantes em arquivos XML;

Como já foi dito, para implementar tais objetivos foi utilizada a linguagem de programação Java, mas para dar suporte aos objetivos 1 e 6, que envolvem o uso da linguagem XML foi utilizado o processador XML *Oracle XML Parser v2* [ORA 00]. Este processador XML é como os descritos no capítulo 2: possui um conjunto de classes Java para realizar o trabalho de *parsing* sobre documentos XML e possui suporte ao modelo DOM (seção 2.5.2). O que diferenciou este *parser* XML dos demais disponíveis foi o suporte ao *parsing* de DTDs, imprescindível para este trabalho. A maioria dos processadores XML utiliza as DTDs apenas para validar o documento XML, não disponibilizando recursos de acesso às DTDs. Já o processador da Oracle tem um conjunto de classes para analisar sintaticamente uma DTD, independente de documentos XML vinculados. Após o *parsing*, a DTD fica representada numa estrutura de objetos, facilitando o seu acesso. Esta estrutura será melhor detalhada nas seções seguintes.

Outro motivo que torna a utilização de um processador XML importante nesta implementação é o modelo DOM. Como será melhor descrito nas próximas seções, a representação da ontologia foi feita utilizando-se o próprio modelo de documentos DOM como suporte.

Outra ferramenta utilizada para se atingir os requisitos foi o pacote de classes denominado *Graph Foundation Classes for Java* da IBM/Alphaworks. Como o próprio nome já diz, este pacote de classes tem por objetivo construir representações de grafos. Como os objetivos 4 e 5 referem-se à representação gráfica das ontologias como forma de interação entre o usuário e as ontologias, então a escolha do pacote da IBM/Alphaworks neste caso faz bastante sentido, tendo em vista as facilidades de construção, visualização e *layout* de grafos que este pacote disponibiliza.

5.2 Visão Geral da Especificação

Com base no que foi descrito na seção anterior, pode-se construir um modelo abstrato para especificar o JOntology. Para tal será utilizado um diagrama de pacotes da UML. Segundo Fowler, a linguagem UML não define nenhum tipo de "diagrama de pacotes". Este tipo de diagrama não passa de um diagrama de classes que mostra apenas pacotes e dependências [FOW 00]. Um pacote em UML é um grupo de elementos de modelagem UML, em especial classes [OMG 99].

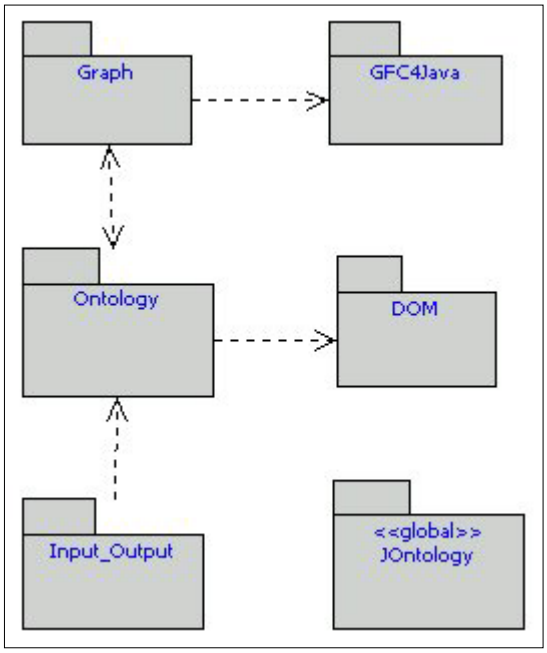


FIGURA 5.1 - Diagrama de pacotes do programa

A fig. 5.1 mostra, num diagrama de pacotes, uma visão geral do projeto do programa JOntology. Os pacotes mostrados na figura representam apenas classes. As setas indicam dependências entre pacotes, por exemplo: o pacote *Graph* depende de serviços providos pelo pacote GFC4Java, mas não o contrário.

O pacote "JOntology" é o pacote principal, da onde é iniciada a execução do programa, onde é criada a interface e de onde são detectadas a maioria das ações do usuário. A classe *JOntology*, pertencente ao pacote de mesmo nome, é responsável por criar a maioria das instâncias das outras classes. Por isso, este pacote foi marcado com o estereótipo "<<global>>" que indica que todos os outro pacotes tem algum tipo de dependência com ele.

Os demais pacotes têm o seguinte papel:

- **Ontology:** contém um conjunto de classes responsável pela representação das ontologias através de objetos. A maioria das classes aqui são especializações de classes do pacote DOM.
- **DOM:** este é o pacote *org.w3c.dom* do processador da Oracle, ou seja, é a implementação do modelo DOM feita pela Oracle. O pacote Ontology depende diretamente deste pacote, tendo em vista que a representação das ontologias foi feita com o modelo DOM.
- **Graph:** as classes deste pacote utilizam o pacote GFC4Java para apresentar as ontologias visualmente em forma de grafo. A partir desta representação, o usuário pode também alterar as ontologias representadas no pacote Ontology.
- **GFC4Java:** como já foi dito, o GFC4Java é um pacote de classes desenvolvido pela IBM/Alphaworks para geração, visualização e *layout* de grafos, utilizado aqui como suporte na geração dos grafos das ontologias.
- **Input_Output:** contém classes que são responsáveis pela entrada de dados, ou seja, de DTDs e ontologias existentes, e saída de dados, ou seja, o armazenamento de arquivos XML a partir das ontologias geradas.

5.3 Representação das Ontologias

Nesta seção será apresentado como foi modelado a representação das ontologias no JOntology e como elas são geradas a partir de DTDs.

A idéia principal para a modelagem das ontologias foi utilizar o *Document Object Model*, ou DOM. Pode-se justificar o uso do DOM da seguinte maneira: baseado no fato de que o objetivo principal do programa é construir ontologias no formato de documentos XML, mais especificamente no formato definido na DTD do capítulo 4 (fig. 4.8), e tendo em vista que as classes definidas no DOM servem justamente para a representação de documentos XML, então o uso do mesmo é justificável. Outro fator importante é a integração das ontologias, para tal é necessário, entre outras coisas, resolver sinônimos utilizando um *thesaurus*, que também é armazenado em XML, como foi definido na seção 4.3.7.

Em primeiro lugar, pode-se ver como o pacote "Ontology" foi projetado, pelo diagrama da fig. 5.2:

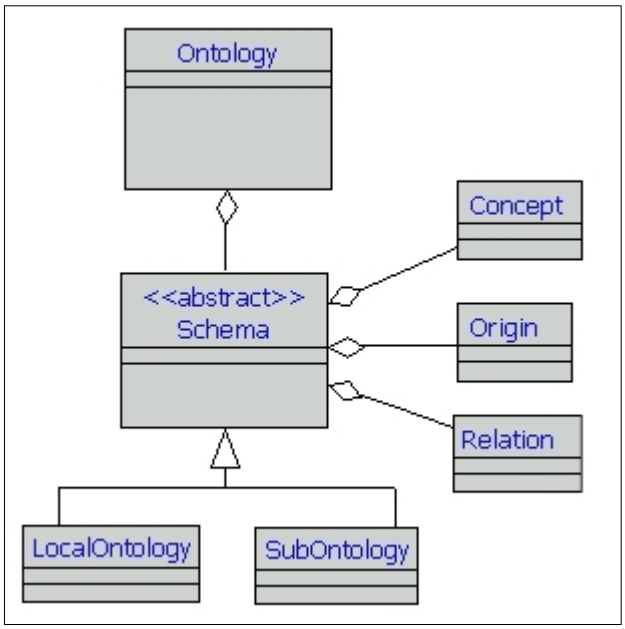


FIGURA 5.2 - Diagrama de classes da representação da Ontologia

Este modelo é em parte baseado na representação de ontologias que foi definido no capítulo 4. Um objeto da classe *Ontology* é um agregado de *Schemas*, que podem ser sub-ontologias (classe *SubOntology*) ou ontologias locais (classe *LocalOntology*). A classe abstrata *Schema* define o formato comum para qualquer tipo de ontologia, que é um agregado de conceitos (classe *Concept*), um agregado de origens (classe *Origin*) e um agregado de relacionamentos (classe *Relation*), entre outras coisas. No que diz respeito ao formato dos dados, tanto sub-ontologias quanto ontologias locais têm o mesmo formato. O que as difere é a forma como cada uma é construída, por isso as duas classes que as representam herdam as características da classe *Schema* e adicionam características próprias.

Além disso, é importante destacar como são feitas as associações entre os objetos da fig. 5.2. Para isso, foi utilizado o DOM, que em suas classes já define operações específicas para manipular nodos. Lembrando que o DOM é um modelo de objetos de documentos XML, mas que não necessita de um documento XML existente, isto é, é possível criar os objetos do DOM independentemente, exatamente o que é feito aqui. A fig. 5.3, logo abaixo, ilustra como estas características foram aproveitadas:

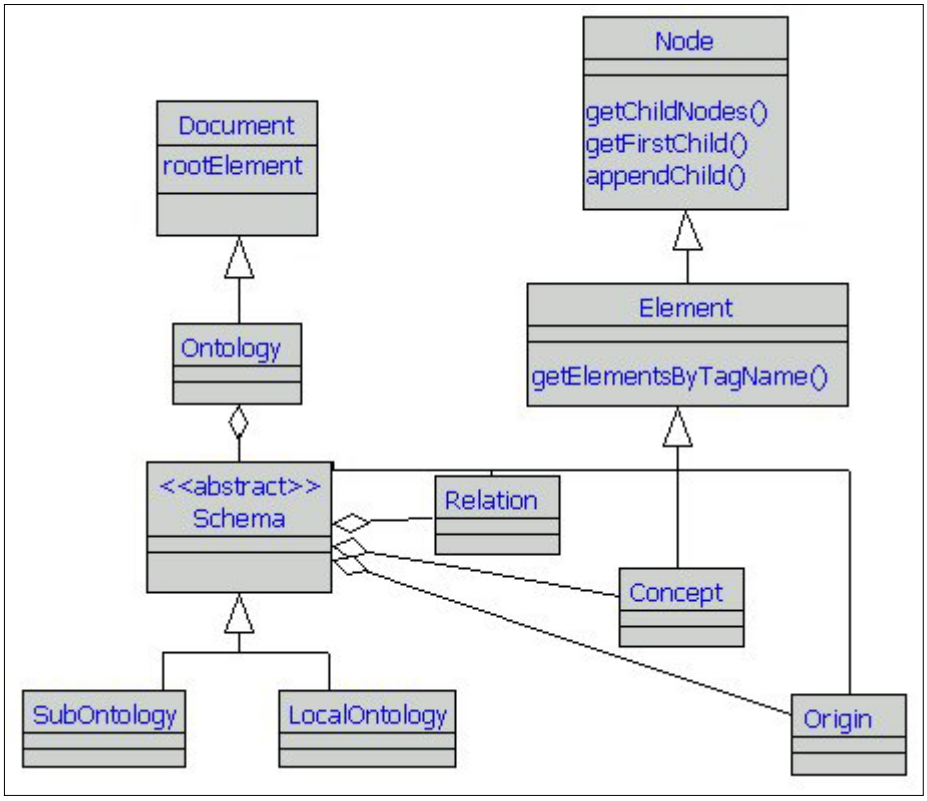


FIGURA 5.3 - Diagrama de classes da ontologia associadas com as classes DOM

A fig 5.3 mostra a dependência entre o pacote Ontology e o pacote DOM, mostrado na fig. 5.1. As classes *Node*, *Element* e *Document* são do pacote DOM implementado no processador XML da Oracle. Como foi explicado na seção 2.5.2, no DOM a maioria das classes são especializações da classe *Node*, sendo que a mais importante é a classe *Element*, que representa os elementos do documento XML. Pois como o objetivo aqui é justamente construir elementos XML, foi definido que as classes *Relation*, *Origin*, *Concept* e *Schema* seriam especializações de *Element*, e consequentemente, também de *Node*. Com isso, o trabalho de criar as associações entre esses objetos tornou-se muito mais direto, bastando utilizar as operações herdadas da classe *Node*, como as mostradas no diagrama: *appendChild()*, *getFirstChild()* e *getChildNodes()*.

Já a classe *Ontology* é uma especialização da classe *Document*, também do DOM. Um objeto *Document* representa todo o documento XML, funcionando como "raiz" do modelo DOM. Isto mostra que a representação de uma ontologia é sempre baseada num objeto *Ontology*, de onde são adicionados os demais elementos.

Nas próximas seções será delineado o papel e as características principais de cada classe mostrada na fig. 5.3.

5.3.1 A classe *Concept*

A classe *Concept* constrói um elemento XML no DOM para representar os conceitos da ontologia, ou seja, cria o elemento <*Concept*> e seus elementos filhos. A fig. 5.4 mostra a especificação da classe *Concept.*

FIGURA 5.4 - Especificação da classe *Concept*

Quando um objeto deste tipo é construído, ele recebe como parâmetros o rótulo de conceito (ou *label*), o número do seu ID e o número de ocorrências do conceito. Em seguida, o objeto irá criar novos elementos XML, para representar estas informações, como foi definido na DTD da fig. 4.8, e irá agregá-los ao elemento <*Concept*>.

5.3.2 A classe *Relation*

A classe *Relation* é responsável pela criação dos elementos que representam os relacionamentos das ontologias. A classe *Relation* tem a seguinte especificação:

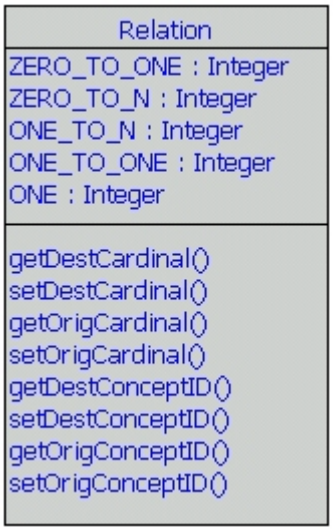


FIGURA 5.5 - Especificação da classe *Relation*

O construtor de *Relation* recebe como parâmetros o ID da ontologia a qual pertence, o ID do conceito origem, a cardinalidade do conceito origem, o ID do conceito destino e a cardinalidade do conceito destino. Estas informações são então repassadas aos elementos filhos do elemento *<Relationship>*, como definido na DTD das ontologias.

A classe *Relation* também mantém quatro constantes numéricas que servem para representar as cardinalidades. Estas constantes têm visibilidade pública, pois também são usadas por outras classes. Essas constantes são *ONE*, *ONE_TO_N*, *ONE_TO_ONE*, *ZERO_TO_ONE* e *ZERO_TO_N*. A constante *ONE* representa as cardinalidades que devem ser marcadas com "1..?", como foi discutido no capítulo 4.

5.3.3 A classe *Origin*

A classe *Origin* representa as origens dos conceitos nas ontologias. A estrutura da classe *Origin* é a seguinte:

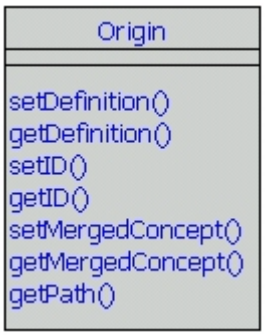


FIGURA 5.6 - Especificação da classe *Origin*

O elemento *Origin* traz consigo, além do ID do conceito que representa e o caminho ou URL da DTD em que este conceito se originou, o ID de um possível conceito que tenha sido integrado a este conceito. Esta informação está presente no elemento <*Concept_Merged_ID*>. O método *setMergedConcept(Concept)* serve para modificar esta informação.

5.3.4 A classe *Schema*

A classe *Schema* é uma classe abstrata. Segundo o manual da SUN, uma classe abstrata em Java é uma classe que serve somente para ser especializada, ou seja, não pode ser instanciada, já que representa conceitos abstratos [SUN 99].

No caso da classe *Schema*, a opção por uma classe abstrata se dá por motivos óbvios. As classes que realmente serão instanciadas são as classes *SubOntology* e *LocalOntology*. Como estas duas classes têm um conjunto de operações e atributos em comum, e a estrutura dos elementos XML tanto nas sub-ontologias quanto nas ontologias locais é o mesmo, justifica-se então a opção de se utilizar uma classe abstrata, que funcione como uma classe genérica para estas duas classes, como pode ser visto no diagrama da fig 5.3.

De uma forma geral, a principal função desta classe é criar a estrutura da representação de ontologias, sejam sub-ontologias ou ontologias locais. Assim que um objeto da classe *SubOntology* ou da classe *LocalOntology* é instanciado, o construtor da classe *Schema* entra em ação construindo a estrutura básica de elementos XML como mostrado na DTD da fig 4.8.

A classe *Schema* foi construída como mostra a fig. 5.7:

FIGURA 5.7- Especificação da classe *Schema*

O único parâmetro passado para o construtor de *Schema* é o ID do elemento *Ontology*. Um ID para este esquema é gerado a partir da classe *SchemaID*. As demais informações, como *label*, tipo, descrição e fonte serão devidamente informadas pelas classes *SubOntology* e *LocalOntology*.

Os principais métodos definidos pela classe *Schema* servem para consultar e modificar as principais informações contidas nos esquemas, como os métodos *addRelations(Vector)* e *addOrigins(Origins)* que são utilizados durante a construção das ontologias.

Destacam-se também os métodos para recuperar as informações dos conceitos, origens e relacionamentos que serão agregados ao esquema. Alguns destes métodos são: *getRelationsByID(), getConceptByID(), getRelations(), getConcepts()*, entre outros.

Por fim, a classe *Schema* define para suas classes herdeiras um atributo do tipo *Graph*, que é o grafo que representa este esquema. A parte de geração e visualização de grafos será melhor discutida mais adiante.

5.3.5 A classe *SubOntology*

A classe *SubOntology* é uma especialização da classe abstrata *Schema*. A função principal desta classe é representar as sub-ontologias criadas pelo usuário. A classe *SubOntology* foi definida da maneira mostrada na fig. 5.8:

FIGURA 5.8 - Especificação da classe *SubOntology*

Além de disponibilizar os métodos definidos na sua superclasse *Schema*, a classe *SubOntology* define o método *addConcepts(Vector)*. Este método, que é utilizado na construção da sub-ontologia, foi definido apenas nesta classe pois observou-se que na classe *LocalOntology* seu formato seria diferente, como será mostrado mais adiante.

5.3.6 A classe *LocalOntology*

A classe *LocalOntology* é a outra subclasse da classe *Schema*. Esta classe representa as ontologias locais criadas a partir de processos de integração de outros esquemas. A forma como esta integração é feita foi abordada no capítulo 4 e será melhor visualizada na seção 5.4.3.

A fig. 5.9 mostra o formato da classe *LocalOntology*:

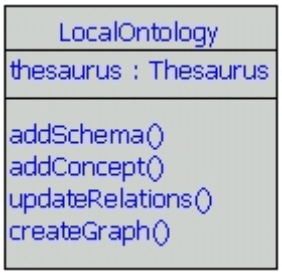


FIGURA 5.9 - Especificação da classe *LocalOntology*

Os principais métodos da classe *LocalOntology* são *addSchema(Schema)* e *addConcept(Concept, Schema)*. O primeiro método recebe um esquema como parâmetro e o integra à ontologia local existente. O segundo é responsável por adicionar os novos conceitos e testar situações de possível integração. Outro método que também é importante durante a integração é *updateRelations()*. Além disso, é a classe *LocalOntology* que faz uso da classe *Thesaurus*, mantendo um objeto deste tipo como atributo.

5.3.7 A classe *Ontology*

Por fim, a classe *Ontology* representa toda a ontologia definida no capítulo 4. Como mostra o diagrama de classes da fig. 5.3, um objeto do tipo *Ontology* será formado por uma agregação de objetos do tipo *Schema*, que na verdade serão ou instâncias de *SubOntology* ou de *LocalOntology*.

Um objeto deste tipo será uma extensão da classe *Document* do DOM. Isto quer dizer que este objeto servirá de "raiz" para o modelo DOM que está sendo criado, e que todos os outros elementos serão filhos deste objeto. Vale lembrar que o modelo DOM que está sendo criado é a própria representação da ontologia em formato XML.

O formato da classe *Ontology* pode ser visto na fig. 5.10:

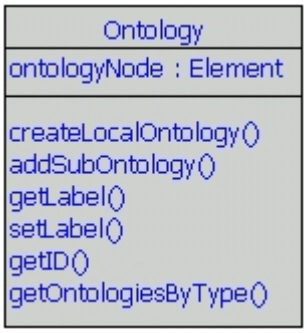


FIGURA 5.10 - Especificação da classe *Ontology*

O atributo do tipo *Element* chamado *ontologyNode* representa o elemento <*Ontology*>. Nele é que são anexadas as informações *Label*, *Ontology_ID* e *Description*, como definido na DTD da fig. 4.8. O ID da ontologia é definido automaticamente por um objeto *OntologyID*, já o *label* e a descrição da ontologia são definidos pelo usuário.

Dois métodos destacam-se na classe *Ontology*. O método *addSubOntology(SubOntology)* anexa uma sub-ontologia recém criada. Já o método *createLocalOntology()* irá tomar um conjunto de esquemas existentes no momento e repassá-los para uma nova ontologia local. Outro método importante é *getOntologiesByType(int)*.

5.4 Construção e Integração das Ontologias

Na seção 5.3 foi mostrado, através de diagramas de classes, como as ontologias são representadas no programa JOntology. Nesta seção será mostrado como essas representações são construídas, tanto através do mapeamento de DTDs quanto através de integração.

5.4.1 A construção de sub-ontologias

As sub-ontologias são construídas a partir de DTDs. Uma única sub-ontologia representa uma única DTD.

Para se construir uma sub-ontologia, a classe *SubOntology* recorre a classes auxiliares. O diagrama de classes da fig. 5.11 ilustra isso mais claramente:

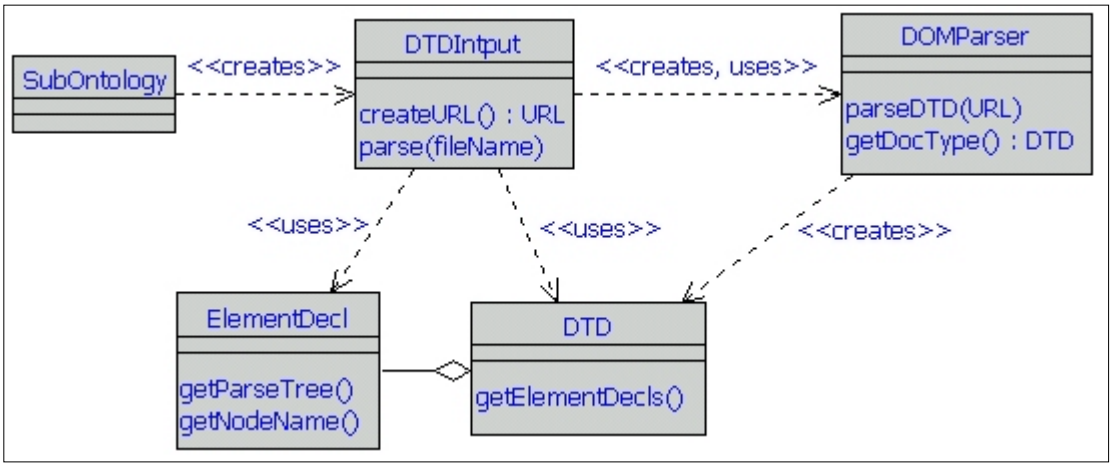


FIGURA 5.11 - Classes utilizadas na construção de uma sub-ontologia

As classes *DOMParser*, *DTD* e *ElementDecl* pertencem ao pacote de classes do processador XML da Oracle. A classe *DOMParser*, apesar de ter esse nome, não constrói apenas modelos DOM a partir de documentos XML. Como mostrado no diagrama, esta classe possui um método chamado *parseDTD(URL)*, que tem a capacidade de fazer *parsing* num arquivo DTD, criando uma instância da classe *DTD*, responsável por representar o arquivo processado. A classe *DTD* pode ser vista, entre outras coisas, como um agregado de objetos do tipo *ElementDecl*, que representam cada declaração de elementos presente na DTD.

A classe *ElementDecl* têm dois métodos importantes, como mostrado no diagrama: *getNodeName()* e *getParseTree()*. O primeiro, como pode-se deduzir, retorna o nome do elemento declarado. O segundo, retorna uma árvore de nodos, ou melhor, de objetos da classe *Node*, representado o modelo de conteúdo do elemento declarado. Este método é muito importante para a interpretação das informações contidas no modelo de conteúdo do elemento. A fig. 5.12 exemplifica o uso do método *getParseTree()*.

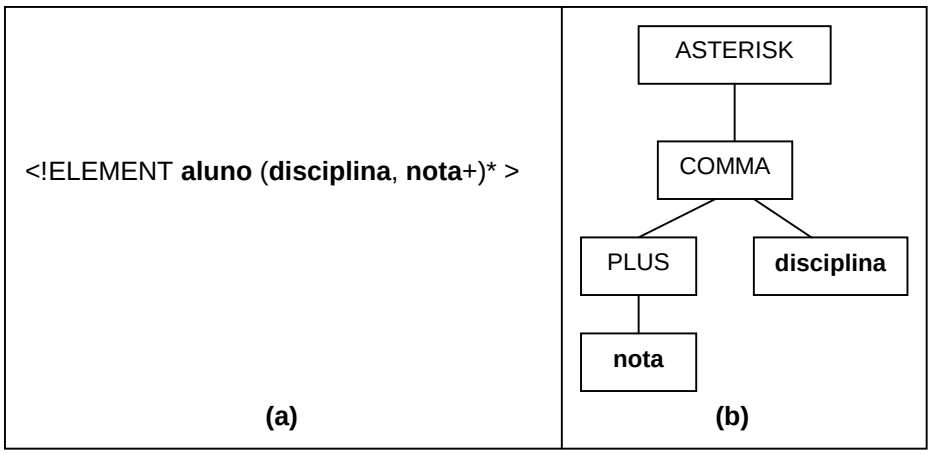


FIGURA 5.12 - Exemplo de uso do método *getParseTree()*

O nodo chamado "COMMA" na fig. 5.12 (b) representa a vírgula, ou seja, uma seqüência de declarações no modelo de conteúdo da declaração do elemento "aluno" em (a). Os nodos "ASTERISK" e "PLUS" representam os sinais de ocorrência dos elementos declarados e por fim "nota" e "disciplina" representam os elementos declarados dentro do elemento "aluno". Analisando as árvores do modelo de conteúdo das declarações de elementos, é possível então determinar quais elementos estão relacionados entre si e quais as cardinalidades destas relações.

Com os recursos apresentados anteriormente, pode-se agora traçar a seqüência de passos que levam à construção de uma sub-ontologia no JOntology. Boa parte do que será mostrado é baseado na seção 4.3.8, em especial no algoritmo de geração de sub-ontologias da fig. 4.8. O algoritmo mostrou qual é a lógica por trás da geração de sub-ontologias. Já na fig. 5.13 será mostrado como os objetos apresentados anteriormente interagem entre si para consolidar a seqüência de passos do mesmo algoritmo.

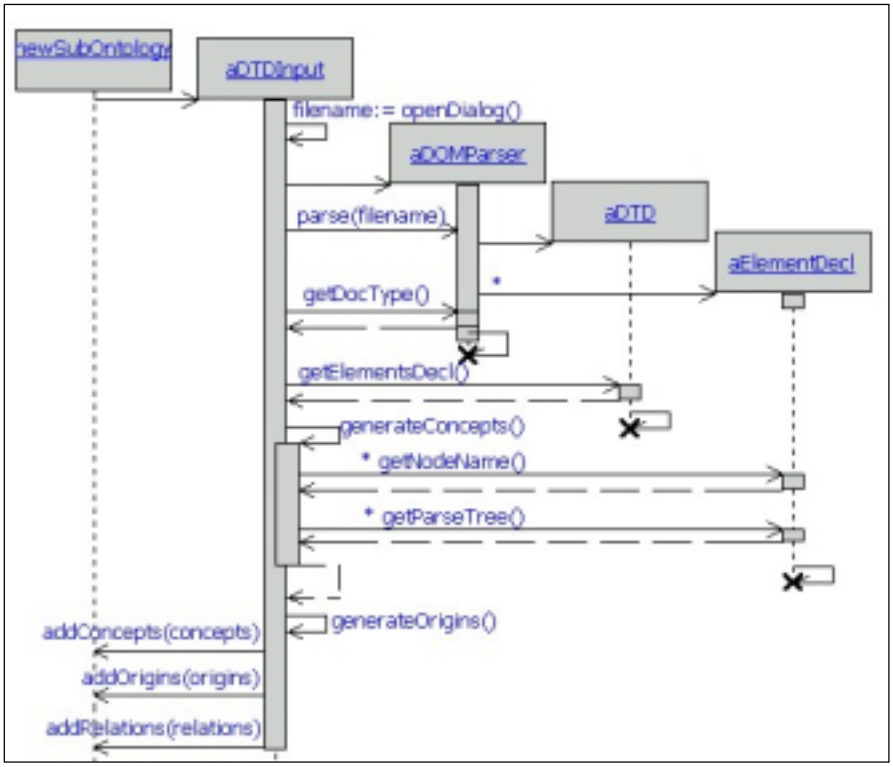


FIGURA 5.13 - Diagrama de seqüência mostrando a criação de uma sub-ontologia

A fig. 5.13 é um diagrama de seqüência em UML mostrando como é criada uma sub-ontologia, indicando os principais objetos envolvidos.

Quando um novo objeto do tipo *SubOntology* é criado, ele cria uma instância da classe *DTDInput*. O papel principal deste objeto *DTDInput* é providenciar o processamento de um arquivo DTD, analisá-lo e então construir o "corpo" principal da sub-ontologia.

A primeira coisa que o objeto do tipo *DTDInput* fará será abrir uma janela de diálogo para o usuário indicar a localização do arquivo DTD. De posse do caminho e do nome do arquivo, o *DTDInput* irá criar uma instância da classe *DOMParser* e em seguida enviará a mensagem *parseDTD(String)* tendo o caminho do arquivo como parâmetro. Durante o trabalho de *parsing*, a classe *DOMParser* cria mais dois tipos de objetos: um objeto do tipo *DTD* e vários objetos do tipo *ElementDecl*, tantos quantos houverem declarações de elementos no arquivo DTD em questão.

Terminado o *parsing* do arquivo, o *DTDInput* recebe o objeto *DTD* gerado pelo *DOMParser* através da mensagem *getDocType()*. A partir de agora, o objeto *DTDInput* pode finalmente analisar a estrutura do DTD para gerar os conceitos, origens e relações da sub-ontologia.

É chamado então o método *generateConcepts()* que irá analisar cada objeto *ElementDecl*. Este método é recursivo, e para gerar um conceito, ele se vale do método *getNodeName()* da classe *ElementDecl*, para determinar o nome do elemento, e consequentemente, o nome do conceito. Depois, com o método *getParseTree()*, é possível determinar quais elementos estão relacionados com o elemento atual, gerando então os relacionamentos à medida que novos conceitos são gerados.

Mais adiante, os conceitos recém criados são analisados pelo método *generateOrigins()*, para a geração das origens.

Por fim, conceitos, origens e relacionamentos são devidamente agregados ao objeto *SubOntology*.

5.4.2 A construção de Ontologias Locais

As ontologias locais são construídas a partir da integração de outros esquemas, ou seja, de sub-ontologias e ontologias locais já existentes. Nesta seção será mostrado de uma forma geral como esse processo ocorre. Dois detalhes importantes, que são o processo de integração em si e a utilização de um *thesaurus*, serão apresentados nas próximas seções.

A fig. 5.14 mostra o diagrama de seqüência da construção de uma ontologia local. O primeiro passo é quando o objeto *Ontology* recebe uma chamada ao método *createLocalOntology()* vinda do objeto *JOntology*.

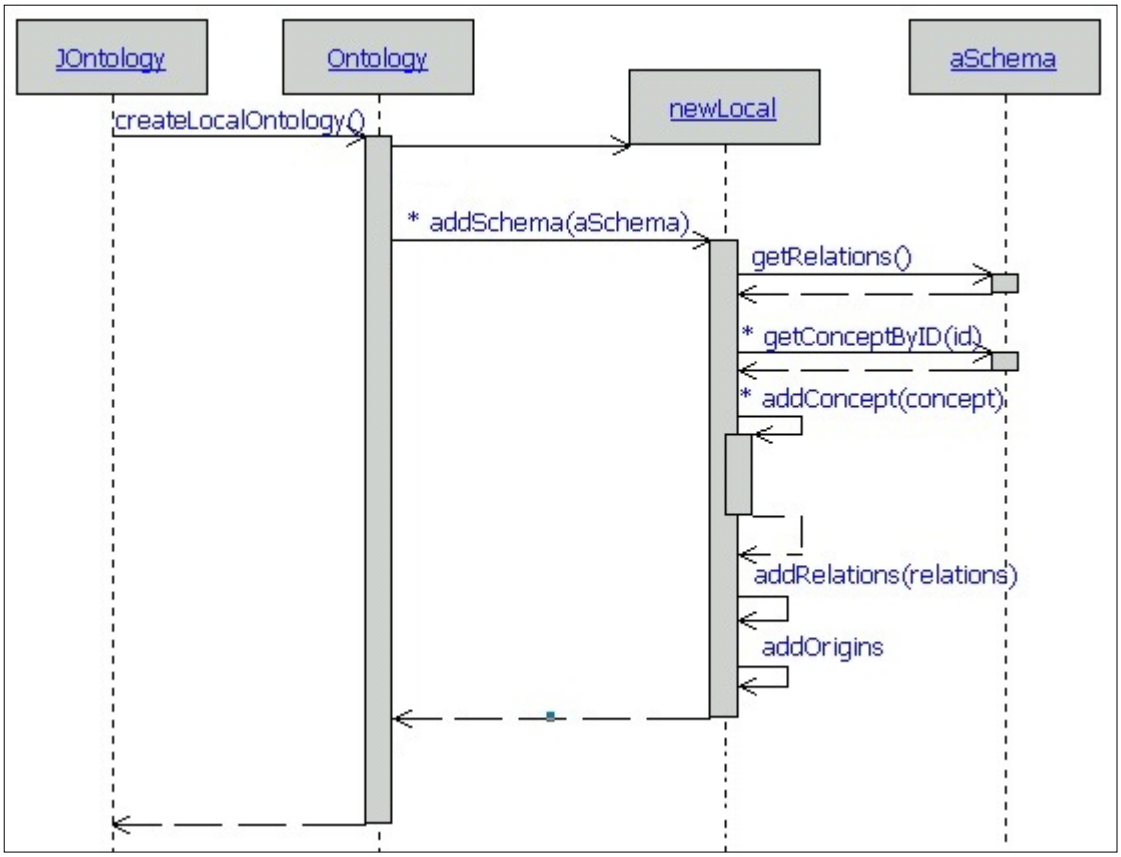


FIGURA 5.14 - Diagrama de seqüência mostrando a criação de uma ontologia local

Em primeiro lugar são tomadas todas os sub-ontologias ou ontologias locais que foram selecionadas pelo usuário, que serão adicionadas uma a uma na nova ontologia local através do método *addSchema(Schema)*.

O método *addSchema(Schema)* definido na classe *LocalOntology*, não faz distinção entre sub-ontologias e ontologias locais, ele trata a ontologia recebida como parâmetro como um objeto da superclasse *Schema*, isto se faz possível porque os métodos definidos na classe *Schema* são suficientes para a integração. Este método irá preparar o processo de integração para o método *addConcept(Concept, Schema)*, que será melhor descrito na próxima seção.

O método *addSchema* executa os seguintes procedimentos:

1. Cria cópias de todas os relacionamentos e origens do esquema a ser adicionado, armazenando estas cópias em vetores. Daqui em diante, somente os objetos destes vetores serão utilizados, exceto os conceitos, que serão copiados mais adiante.
2. Toma os relacionamentos um por um. Cada relacionamento tem o ID de um conceito origem e o ID de um conceito destino.
3. Toma o conceito origem do relacionamento através do método *getConceptByID(int)* e o copia. É verificado se este conceito já não foi adicionado anteriormente, tendo em vista que um conceito pode estar em mais de um relacionamento. Neste caso, não será necessário adicioná-lo novamente.
4. Envia a cópia do conceito para o método *addConcept*. Caso aconteça integração, este método retornará o conceito que foi atualizado.
5. Repete os passos 3 e 4 para o conceito destino do relacionamento.
6. Caso os dois conceitos do relacionamento tenham sido integrados, é necessário atualizar as cardinalidades desta relação.

A situação 6 foi descrita no capitulo 4, na seção 4.3.9, com exemplo na fig. 4.10. É a situação em que a cardinalidade do relacionamento tem que ser revisada por estar conflitante com um novo relacionamento que está sendo integrado.

Ao final, depois de realizar os passos de 1 a 6 para todas os relacionamentos do novo esquema, o método *addSchema* finaliza chamando os métodos *addRelations(Vector)* e *addOrigins(Vector)*, que adicionam as cópias dos relacionamentos e origens na recém criada ontologia local.

5.4.3 O processo de integração

Nesta seção será descrito em mais detalhes o que acontece durante a execução do método *addConcept* da classe *LocalOntology*. Esse método é o responsável em adicionar os novos conceitos na ontologia local e testar as situações onde a integração de conceitos se faz possível. As situações em que dois conceitos devem ser combinados foram descritas no capítulo 4.

O método *addConcept(Concept, Schema)* recebe como parâmetros o conceito a ser adicionado e o esquema a que ele pertence, seja uma sub-ontologia ou outra ontologia local. A partir daí, o método irá tomar cada conceito já existente na nova ontologia local e compará-los com o novo conceito. É então que três situações podem acontecer:

1. O rótulo do novo conceito é igual ao de um conceito já existente. Neste caso, deve acontecer integração.

2. Caso a situação 1 não aconteça, o método pergunta ao objeto *thesaurus*, através do método *findSyns(string)*, se o novo conceito tem algum sinônimo. Em caso positivo, será testado se algum destes sinônimos tem o mesmo rótulo do conceito existente, neste caso, a integração deve acontecer.
3. Caso nenhuma das situações acima aconteça, então o conceito não pode ser integrado, e será diretamente adicionado a ontologia local, sem maiores alterações.

Quando as situações 1 ou 2 acontecem, têm-se uma situação de integração entre dois conceitos. Isto quer dizer que o novo conceito não será adicionado, mas sim que diversas modificações terão que acontecer. Estas alterações foram descritas na seção 4.3.9, durante a explicação sobre o algoritmo de criação de ontologias locais da fig. 4.9. O que acontece aqui pode ser resumido da seguinte maneira:

- **Conceitos**: o valor do elemento *<Occurrency>* do conceito existente deverá ser incrementado.
- **Origens**: uma nova origem deve ser criada e adicionada à ontologia local.
- **Relacionamentos**: os relacionamentos em que o novo conceito participava terão que ser alteradas. O valor dos elementos *<Orig_Concept_ID>* e *<Dest_Concept_ID>* que antes eram o ID do novo conceito, agora terão que ser do conceito existente. O método *updateRelations()* é chamado para esta função.

É importante ressaltar aqui que todos os objetos *Relation*, *Concept* e *Origin* do esquema que está sendo adicionado à nova ontologia local foram copiados anteriormente pelo método *addSchema*. Isto é feito para garantir que este esquema continue existindo em seu formato original.

5.4.4 A classe *Thesaurus*

A classe *Thesaurus* participa ativamente do processo de integração. Quando um objeto desta classe é criado, o primeiro procedimento do seu construtor será criar uma instância do *DOMParser* para construir o modelo DOM do arquivo "thesaurus.xml". Este arquivo, em formato XML, mantêm registrado todos os sinônimos conhecidos pelo *thesaurus* do programa.

A fig. 5.15 mostra o formato da classe *Thesaurus*:

FIGURA 5.15 - Especificação da classe *Thesaurus*

O método *findSyns(String)* pesquisa no modelo DOM do *thesaurus* pela palavra procurada e retorna um vetor contendo todos os sinônimos encontrados, ou um objeto nulo caso não encontre nenhum.

5.5 Visualização e Edição da Ontologia

Como foi explicado anteriormente, para permitir ao usuário uma forma de visualizar as ontologias que estão sendo representadas pelo programa, de uma forma mais clara e intuitiva, foi utilizado um formato visual de grafo. Caso não houvesse nenhum meio para tal, a única forma que o usuário teria para visualizar as ontologias seria analisando diretamente os arquivos XML. Mas os documentos XML resultantes da construção de ontologias são extensos e um tanto difíceis de serem analisados.

Para desenvolver os grafos na linguagem Java, foi utilizado o pacote de classes GFC4Java da IBM/Alphaworks. Este pacote tem as seguintes características [IBM 99]:

1. Uma biblioteca de classes para representação de grafos, que é o *Graph Foundation Classes*.
2. Um *framework* para *layout* e desenho de grafos.

O diagrama de pacotes da fig. 5.1 mostra uma dependência entre o pacote *Graph*, desenvolvido para o programa JOntology e o GFC4Java, uma parte desta dependência pode ser vista no diagrama de classes da fig. 5.16 , logo abaixo:

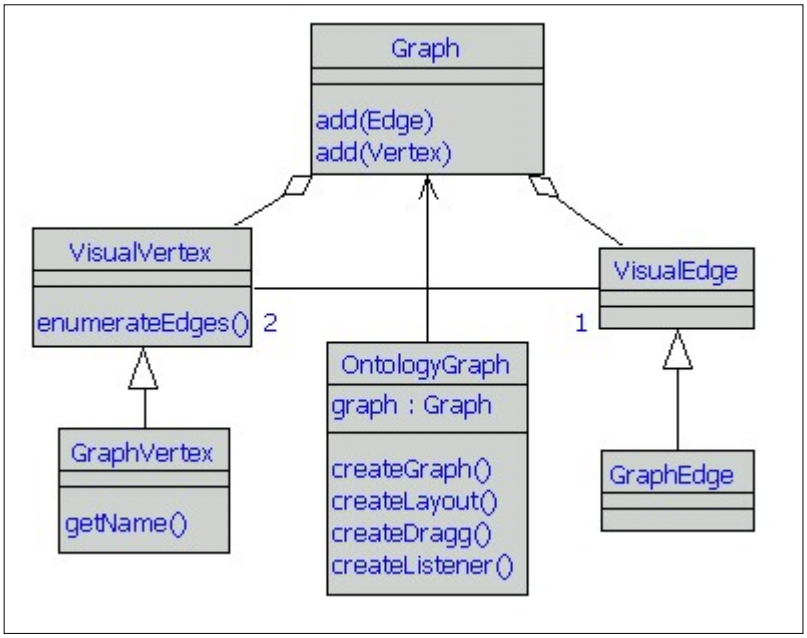


FIGURA 5.16 - Diagrama de classes utilizadas para a representação dos grafos

As classes *Graph*, *VisualVertex* e *VisualEdge* são do GFC4Java e as classes *GraphVertex*, *OntologyGraph* e *GraphEdge* foram desenvolvidas para o programa JOntology.

Como pode-se ver no diagrama, um objeto do tipo *Graph* é um agregado de vértices, da classe *VisualVertex* e arestas, da classe *VisualEdge*. Já *GraphVertex* e *GraphEdge* são especializações de *VisualVertex* e *VisualEdge*. O objeto *OntologyGraph* é o responsável por analisar uma ontologia e criar as instâncias de *GraphVertex*, *GraphEdge* e *Graph* equivalentes.

5.5.1 Criação do Grafo a partir da ontologia

O processo de criação do grafo é bastante simples, e se dá logo após a criação de uma nova ontologia. Tudo começa com a criação de uma instância da classe *OntologyGraph*, passando-se como parâmetro ao seu construtor o objeto *Schema* da ontologia em questão. Como mostra o diagrama anterior, o objeto *OntologyGraph* mantêm como atributo um objeto da classe *Graph*.

O diagrama de seqüência da fig 5.17 mostra os principais passos na criação de um grafo representativo de uma ontologia.

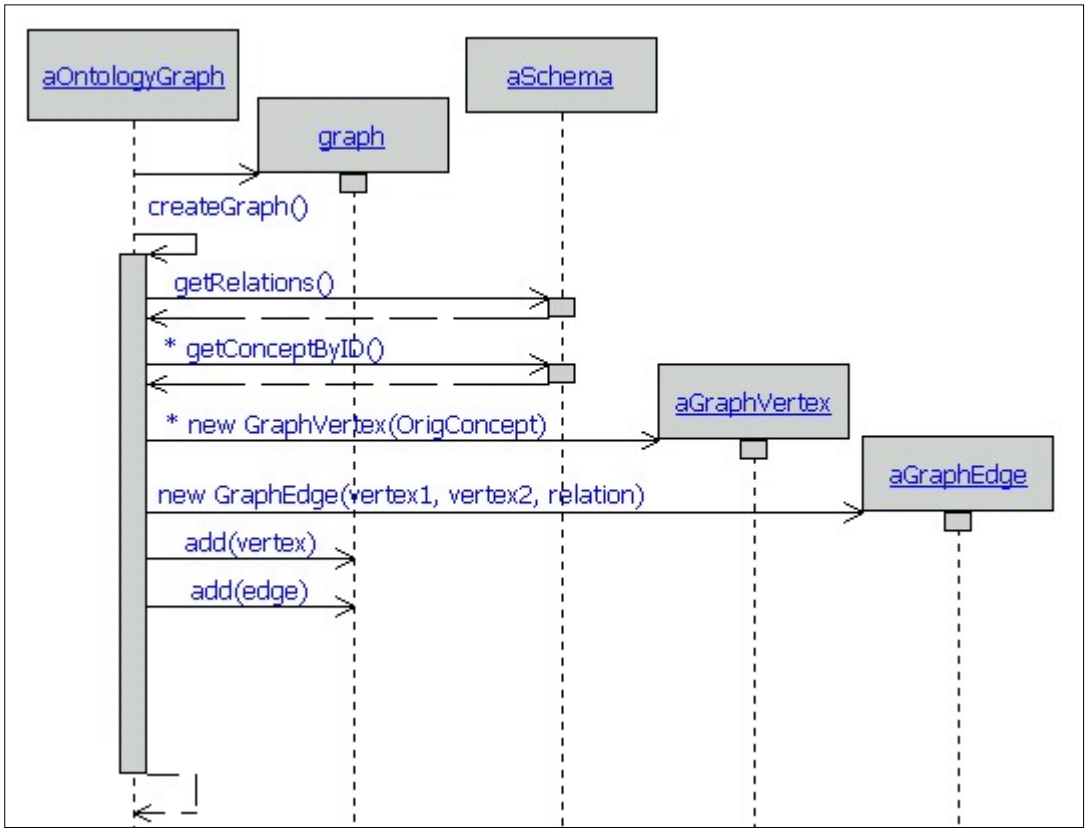


FIGURA 5.17 - Diagrama de seqüência com a criação de um grafo

O método *createGraph()* da classe *OntologyGraph* é o responsável por analisar o esquema e construir o grafo em si. Essa análise é baseada no conjunto de relacionamentos do esquema a ser representado. Cada relacionamento traz o ID de dois conceitos, o origem e o destino. Através do método *getConceptByID(int)* é possível localizar esses conceitos no seu respectivo esquema e construir dois vértices para representa-los, ou seja, dois objetos *GraphVertex* são criados tendo como parâmetros os dois conceitos mencionados. Um controle deve ser feito neste ponto, para que um conceito não seja analisado mais de uma vez, tendo em vista que um conceito pode estar em mais de um relacionamento.

A seguir, um objeto *GraphEdge* é criado, tendo como argumentos os dois vértices criados anteriormente mais o relacionamento, ou seja, o objeto *Relation* em si. Por fim tanto os dois vértices quanto a aresta são adicionados ao objeto *graph*.

A fig. 5.18 abaixo mostra uma *screenshot* de um grafo gerado pelo JOntology a partir da sub-ontologia Cadastro de Alunos:

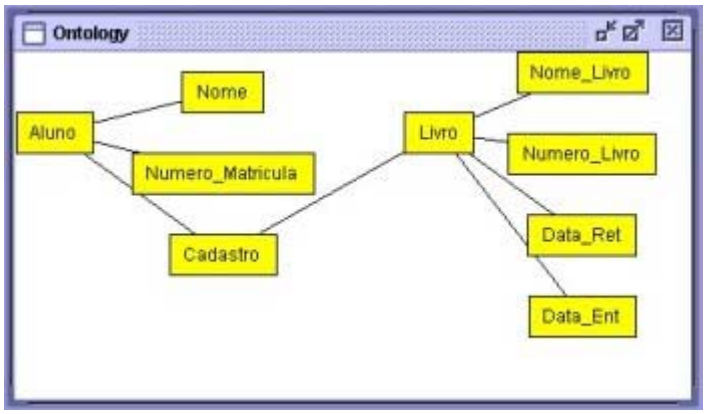


FIGURA 5.18 - Exemplo de um grafo gerado no JOntology a partir de uma sub-ontologia

O *layout* dos vértices, ou seja, a posição de cada vértice na tela, é feito através de um recurso provido pelo pacote GFC4Java.

5.5.2 Edição da ontologia através do grafo

O diagrama de classes da fig. 5.19 mostra as classes relacionadas com a representação da ontologia em forma de grafo. Quando o usuário quiser alterar ou verificar algum detalhe na ontologia, será através dos objetos *GraphVertex* e *GraphEdge* que ele acessará a ontologia. Pode-se dizer então que estes dois objetos visuais são a interface entre a ontologia representada no modelo DOM (fig 5.2 e fig. 5.3) e o usuário. O diagrama de classes da fig. 5.19 esclarece como as classes relativas aos grafos, da fig. 5.16, estão associadas com as classes relativas às ontologias, da fig. 5.2.

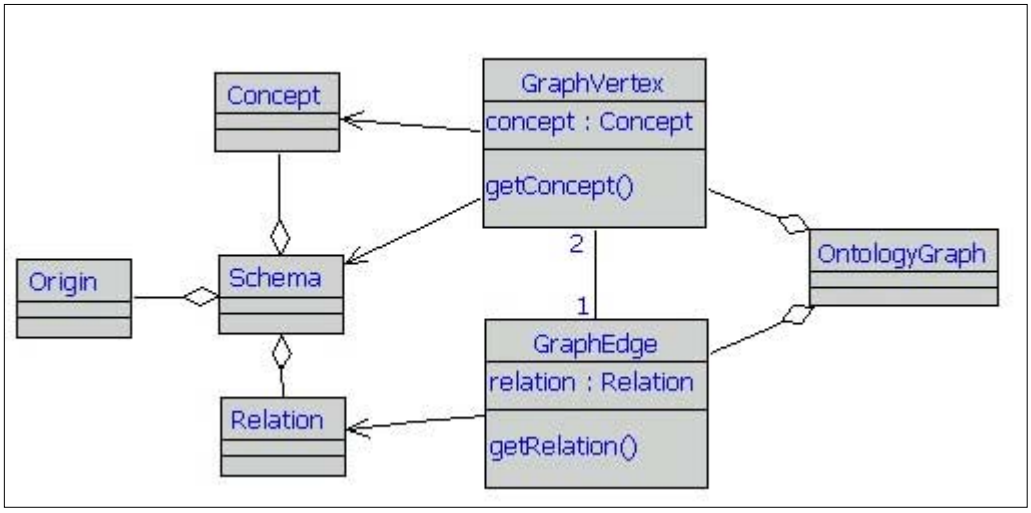


FIGURA 5.19 - Diagrama de classes com as associações entre ontologia e grafo

O diagrama da fig. 5.19 mostra a dependência entre o pacote *Ontology* e o pacote *Graph*, mostrado no diagrama de pacotes da fig. 5.1. Basicamente, os objetos *GraphVertex* e *GraphEdge*, que são respectivamente os vértices e arestas do grafo, mantêm como atributo o objeto da ontologia que eles estão representando. Em outras palavras, um objeto do tipo *GraphVertex* mantêm um ponteiro para o respectivo objeto *Concept* que ele representa, e um objeto do tipo *GraphEdge* mantêm um ponteiro para o respectivo objeto *Relation*.

Projetado desta forma, é possível através do objeto *GraphVertex* acessar diretamente o conceito por ele representado. Isto simplifica os processos de quando o usuário quiser acessar as informações do conceito em questão e para atualizar qualquer alteração. Outro fator importante, também visível pelo diagrama, é que cada objeto *GraphVertex* mantêm um ponteiro para o próprio objeto *Schema* ao qual o conceito pertence. Essa característica foi utilizada tendo em vista que grande parte da informação necessária não está diretamente contida no objeto *Concept*. As informações relativas as origens do conceito são mantidas pelos objetos *Origin*, além de informações pertinentes ao próprio esquema.

Um objeto especial chamado *ClickGraphCanvasEventListener* é mantido ativo desde a criação de um novo grafo. Este objeto é responsável em detectar eventuais eventos de mouse do usuário em algum vértice do grafo. Quando o usuário clicar sobre algum vértice, uma janela de diálogo será apresentada ao usuário, mostrando todo um conjunto de informações pertinentes ao conceito em questão. A fig. 5.20, mostra um exemplo:

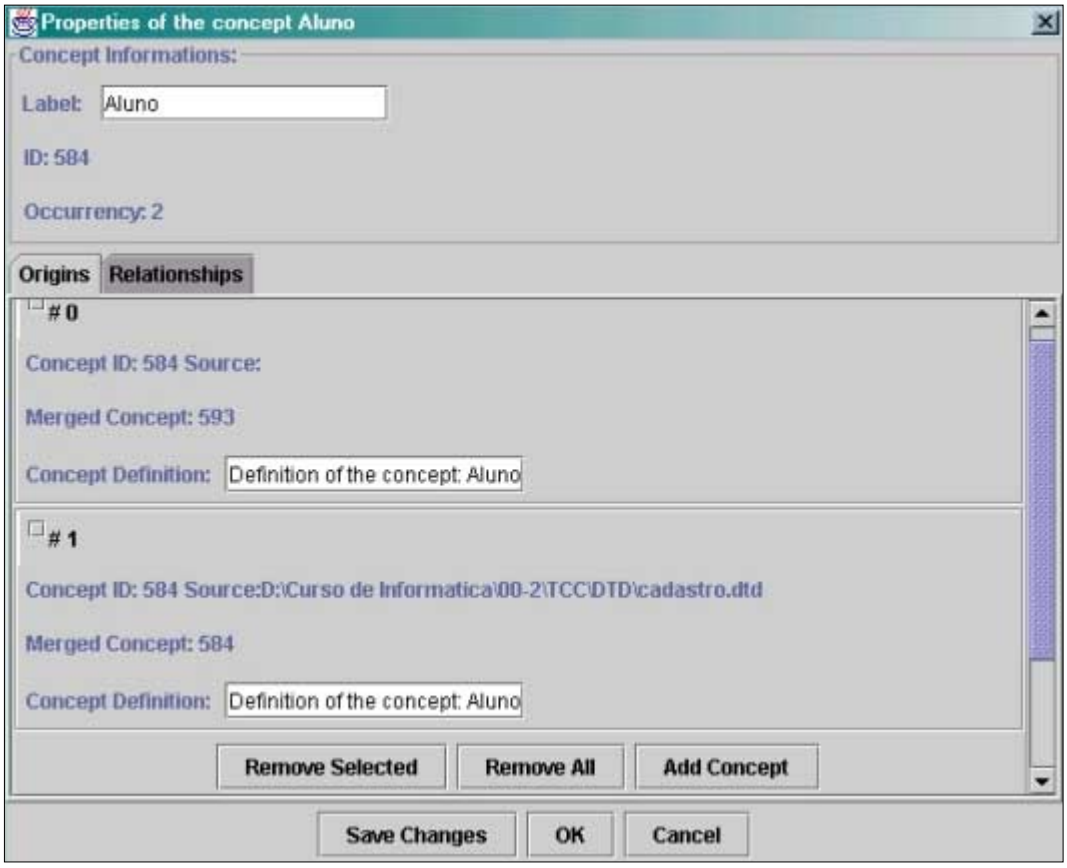


FIGURA 5.20 - Janela de diálogo de um vértice do grafo

A fig. 5.20 mostra a janela de diálogo de um conceito rotulado como "Aluno". Na parte superior da janela encontram-se as informações pertinentes ao conceito em si: o *label*, o ID e o número de ocorrências do conceito, sendo permitido ao usuário modificar o *label*. Mais abaixo, um objeto do tipo *JTabbedPane* com duas *tabs*, ou folhas: *Origins* e *Relationships*. A primeira folha mostra as origens relativas ao conceito, mostrando o valor de *Concept_ID*, *Source*, *Merged_Concept* e de *Concept_Definition*, sendo possível ao usuário modificar a definição do conceito. A segunda folha, que não está visível na fig. 5.20, mostra todos os relacionamentos que este conceito participa, mostrando também as cardinalidades.

Na folha *Origins* é possível excluir e incluir origens. Ao incluir uma origem de outro conceito, o usuário estará na verdade indicando que estes conceitos são sinônimos, portanto devem ser integrados e o conceito em questão terá seu número de ocorrências incrementado, além de serem incluídos no *thesaurus*. Quando isto acontecer, após o processo de integração, o grafo deverá ser atualizado. Se por outro lado, o usuário excluir uma origem, o número de ocorrências deste conceito deve ser decrementado. Já se excluir todas as origens do conceito, ele estará indicando que o conceito em si deve ser excluído.

A folha *Relationships* mostra todos os relacionamentos dos quais este conceito participa, mostrando os conceitos origens e conceitos destinos de cada relacionamento, assim como as respectivas cardinalidades. O usuário pode aqui remover ou adicionar algum relacionamento, além de poder alterar o valor das cardinalidades.

Cabe ressaltar aqui, que esta parte do programa, o diálogo com o usuário para edição da ontologia, ainda encontra-se em fase de desenvolvimento, por isso a *interface* mostrada na fig.5.20 e os procedimentos descritos ainda não são os ideais.

5.6 Interface do Programa JOntology

Nesta última seção será mostrado como o usuário interage com o programa JOntology. Será apresentado sobretudo as principais características da classe *JOntology*, que é a classe principal do programa, responsável principalmente em construir a interface gráfica do mesmo. Será brevemente descrito alguns processos iniciados pelo usuário, como por exemplo, a abertura e gravação de arquivos.

Para construir a interface do JOntology, foi utilizado o pacote de classes denominado *Swing* [SUN 99], da SUN, que é incluso na própria linguagem Java. Até o momento, o protótipo do JOntology tem a seguinte janela principal, mostrada na fig. 5.21:

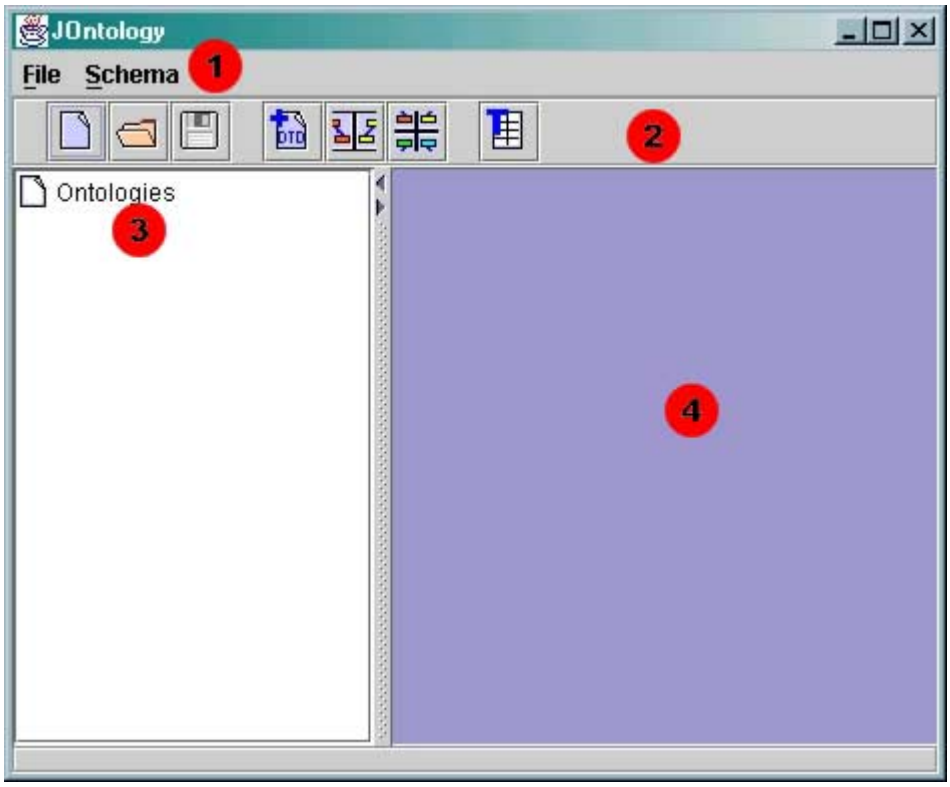


FIGURA 5.21 - Janela principal do protótipo JOntology

Os círculos numerados na fig. 5.21, que não fazem parte do programa, servem aqui para demarcar os principais componentes da janela principal, que são:

(1) Menus com as principais funções. O menu *File* possui as opções: "*New Ontology*", "*Open Ontology*" e "*Save Ontology as XML*". Já o menu *Schema* possui: "*Add DTD (create SubOntology)*", "*Integrate Selected*" e "*Integrate All*".

(2) Uma barra de botões também com as principais funções do programa. A função de cada botão, da esquerda para a direita é: criar nova ontologia, abrir ontologia existente, salvar ontologia atual em XML, criar uma sub-ontologia a partir de uma DTD, integrar esquemas selecionados, integrar todos os esquemas e visualizar/editar *thesaurus*.
(3) Este componente é um objeto *swing* do tipo *JTree*, que têm como função criar representações hierárquicas, no formato de "pastas" ou "nodos". Cada sub-ontologia ou ontologia local existente na ontologia atual será representada aqui como um nodo. A partir deste nodo o usuário pode tomar decisões relativas ao esquema em questão como: excluir, visualizar ou selecionar.
(4) Esta área é um objeto *swing* chamado *JDesktop*, dentro desta área é possível simular um ambiente tipo *desktop*, com diversas janelas internas. As janelas internas que serão utilizadas aqui conterão as representações visuais das ontologias, como a da fig. 5.17.

Já foi mostrado como o usuário pode interagir com um grafo representativo de uma ontologia, na seção 5.5.2. Será mostrado nas próximas seções como o usuário deve proceder para executar as principais tarefas do programa.

5.6.1 Criando uma nova sub-ontologia

Para criar uma nova sub-ontologia, utiliza-se o botão indicado na fig. 5.22.

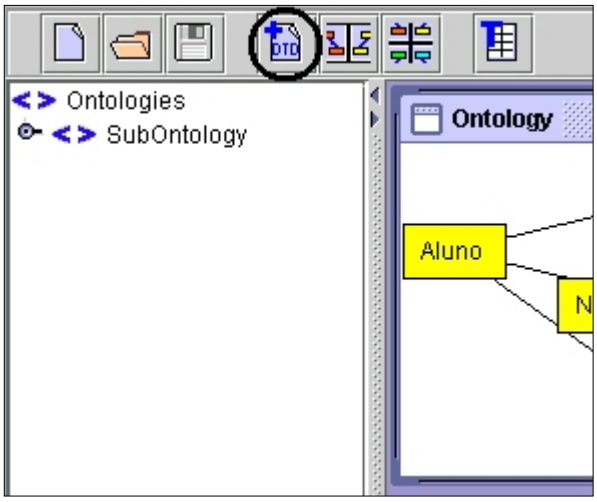


FIGURA 5.22 - Criação de uma nova sub-ontologia

Em primeiro lugar, o programa irá requisitar ao usuário o caminho da DTD que deverá ser transformada em sub-ontologia. Ao criar a sub-ontologia, automaticamente será criado o grafo no *JDesktop* e o nodo na *JTree* que a representam, como pode ser visto na fig. 5.22. Clicando sobre o nodo, surgirá um *pop-up* menu com duas opções: visualizar janela do grafo, caso a mesmo tenha sido fechada ou remover este esquema da ontologia atual. Clicando sobre o mesmo nodo com a tecla *shift* ou *control* pressionadas, o usuário estará selecionando este nodo.

5.6.2 Integrando esquemas e criando ontologias locais

Para integrar as sub-ontologias ou ontologias locais existentes no momento e, conseqüentemente, gerar uma nova ontologia local, existem duas opções: integrar os esquemas selecionados ou integrar tudo. Para isso são utilizados os botões mostrados na fig. 5.23, lembrando que as mesmas funções também são encontradas no menu *Schema*.

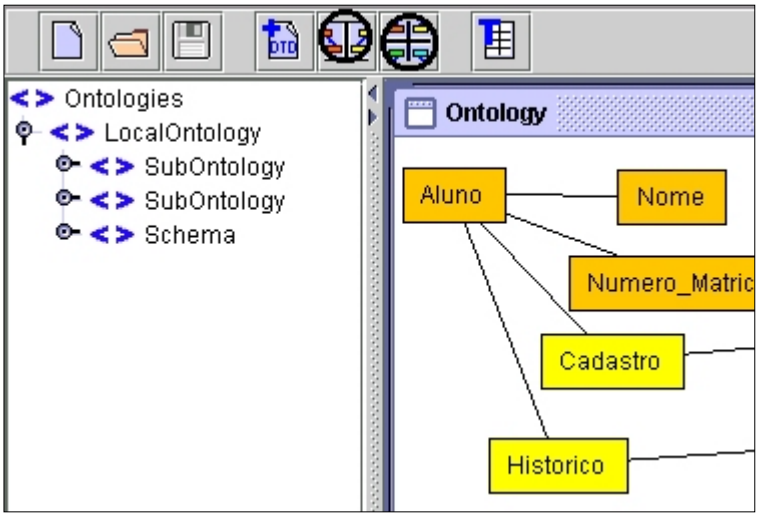


FIGURA 5.23 - Criação de uma nova ontologia local

O primeiro botão em destaque na fig. 5.23 irá integrar somente os esquemas selecionados na *JTree*, enquanto que o segundo botão irá integrar todos os esquemas existentes na *JTree*. Após a integração, os nodos da *JTree* que representam os esquemas que foram integrados irão tornar-se filhos do nodo da nova ontologia local, como mostrado na fig. 5.23. O nodo da ontologia local tem também um outro nodo filho chamado *Schema*, que é uma representação do elemento <*Schema*> relativo a esta ontologia local. Além disso, a janela com o respectivo grafo surgirá automaticamente.

No caso específico do grafo da ontologia local, os conceitos que passaram por alguma integração são marcados com uma cor mais escura, como é o caso de "Aluno", "Nome" e "Numero_Matricula" na fig. 5.23.

5.6.3 Visualização e edição do *thesaurus*

Clicando no botão para a visualização/edição do *thesaurus* faz surgir uma janela contendo um objeto *swing* do tipo *JTable*. A fig. 5.24 mostra um exemplo do *thesaurus*:

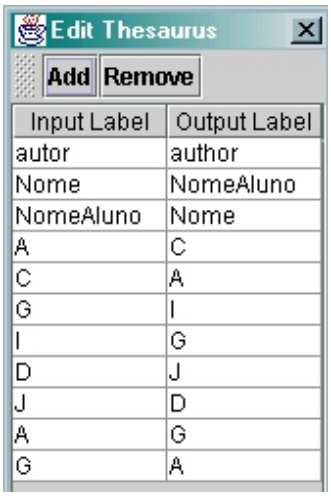


FIGURA 5.24 - Janela para visualização do *thesaurus*

O que aparece na tabela da fig. 5.24 é o resultado da análise do arquivo "thesaurus.xml". Qualquer alteração feita pelo usuário deve ser refletida neste arquivo.

5.6.4 Abrindo e salvando arquivos XML

Para salvar a ontologia atual, o usuário deverá clicar sobre o respectivo botão na barra de botões ou utilizar o menu *File*. Após receber a indicação do nome do arquivo e do local onde o arquivo deverá ser salvo, o programa chamará o método *print(URL)* no objeto *Ontology*. Este método é definido na classe *Document* do DOM e sua função é justamente imprimir o estado atual do documento XML em um arquivo.

Quando quiser abrir uma ontologia existente, o usuário também pode utilizar ou o respectivo botão ou o item *Open* do menu *File*. O processo de abrir uma ontologia de um arquivo XML e analisar os esquemas existentes nela ainda está sendo implementado.

6 CONCLUSÕES, CONTRIBUIÇÕES E TRABALHOS FUTUROS

A motivação principal deste trabalho é desenvolver um sistema de integração de esquemas XML, as DTDs, utilizando ontologias. Para isso, foi estudado e apresentado as principais características da linguagem XML e das ontologias.

A linguagem XML encontra-se atualmente numa grande fase de expansão, sendo criadas novas soluções que tentam aproveitar todo o potencial desta nova linguagem, que já extrapolou seus objetivos iniciais a muito tempo. Um exemplo prático disso é o projeto IBM-UFRGS, no qual este trabalho é baseado, que utiliza toda uma plataforma baseada em XML para possibilitar o acesso a bases de dados legadas. Com todo um conjunto de padronizações, lançadas pela W3C, o trabalho de desenvolver aplicações que suportem XML torna-se mais simples e direto. As implementações do modelo DOM, que seguem a padronização do W3C, são sem dúvida a maneira mais eficaz de construir aplicações que acessam ou criam documentos XML. Um exemplo de utilização eficaz do DOM foi o próprio desenvolvimento do protótipo JOntology, que utilizou as classes DOM implementadas pela Oracle para gerar representações de ontologias diretamente em XML.

A contribuição que este trabalho tenta deixar em relação a linguagem XML, além de um resumo das suas principais características, é demostrar como a linguagem pode ser utilizada num caso prático, que é a integração de esquemas.

A integração de esquemas, em especial de esquemas XML, demonstra sua importância em aplicações com objetivo de distribuir informações. Por exemplo, sistemas com bases de documentos XML, podem utilizar a integração dos esquemas das suas diferentes fontes de documentos para a construção de mecanismos de busca mais eficientes do que os convencionais mecanismos baseados em busca léxica.

De uma forma geral, as principais contribuições deste trabalho são as seguintes:

1. Permitir através da pesquisa realizada em torno da linguagem XML, que novos estudos sejam empreendidos sobre a mesma, no ambiente do curso de Informática da UFPel (capítulo 2);
2. Apresentou um estudo preliminar sobre ontologias, mostrando como esta metodologia pode ser utilizada na implementação de diversas soluções, entre elas, o acesso integrado a documentos XML (capítulo 3) ;
3. Apresentou o problema da integração de esquemas e a sua importância nos sistemas de informação atuais (capítulo 4);
4. A definição de um método para integrar esquemas que utiliza ontologias como suporte (capítulo 4);
5. A implementação de um protótipo para a integração de DTDs, que utiliza o método citado anteriormente (capítulo 5).

Como sugestões para projetos futuros, pode-se citar novos trabalhos que apresentem soluções e estudos de caso envolvendo a linguagem XML. Outra sugestão é a de trabalhos mais detalhados sobre ontologias, tendo em vista a vasta gama de aplicações e possibilidades em torno deste tópico, inclusive em outras áreas, como Inteligência Artificial, por exemplo. Por fim, pode-se também sugerir trabalhos investigativos sobre integração de esquemas que utilizem XML, além de trabalhos que aprimorem ou complementem o que foi proposto neste trabalho no que tange a integração de esquemas XML.

Anexo 1

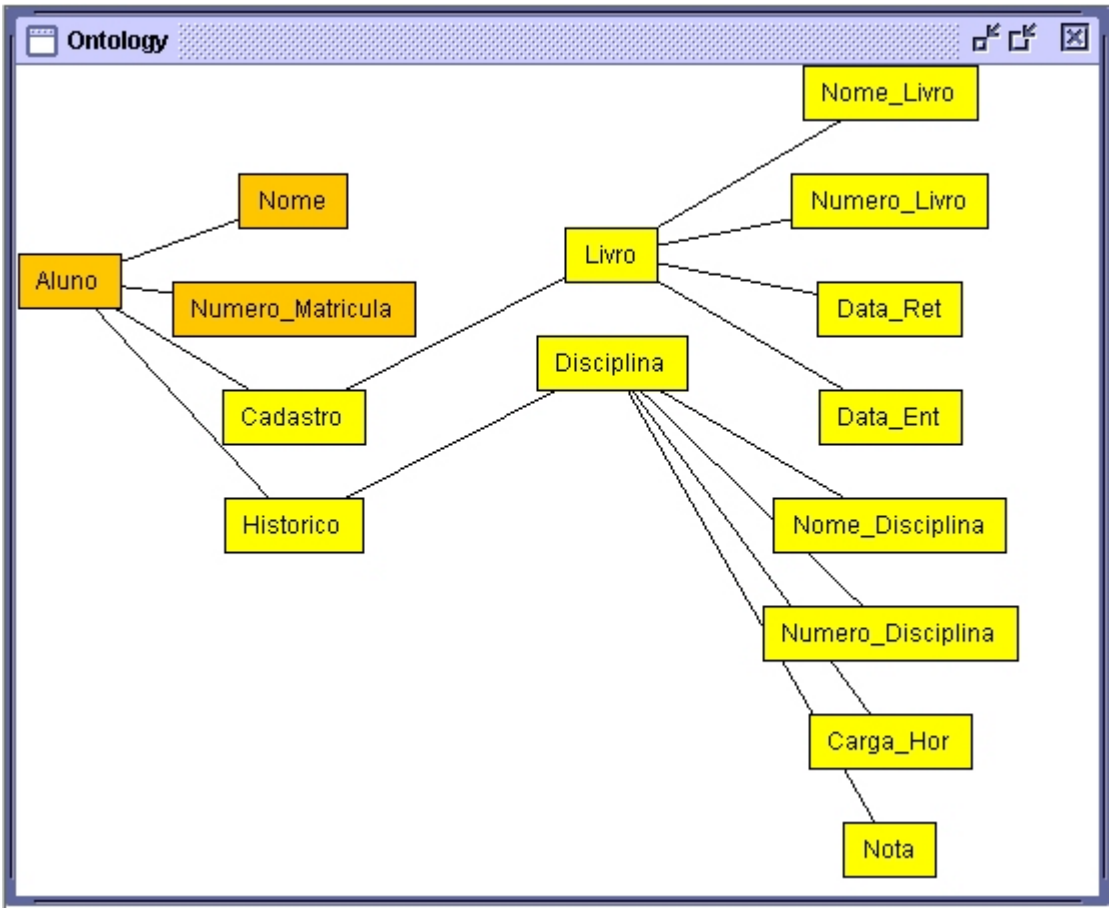
```
<!-- DTD de cadastro de alunos numa biblioteca (cadastro.dtd) -->

<!ELEMENT Cadastro (Aluno, Livro+) >
<!ELEMENT Aluno (Nome, Numero_Matricula)>
<!ELEMENT Nome (#PCDATA)>
<!ELEMENT Numero_Matricula (#PCDATA)>
<!ELEMENT Livro (Nome_Livro, Numero_Livro, Data_Ret, Data_Ent?)>
<!ELEMENT Nome_Livro (#PCDATA)>
<!ELEMENT Numero_Livro (#PCDATA)>
<!ELEMENT Data_Ret (#PCDATA)>
<!ELEMENT Data_Ent (#PCDATA)>
```

```
<!-- DTD do historico escolar dos alunos (historico.dtd) ) -->

<!ELEMENT Historico (Aluno, Disciplina+) >
<!ELEMENT Aluno (NomeAluno, Numero_Matricula)>
<!ELEMENT NomeAluno (#PCDATA)>
<!ELEMENT Numero_Matricula (#PCDATA)>
<!ELEMENT Disciplina (Nome_Disciplina, Numero_Disciplina, Carga_Hor, Nota?)>
<!ELEMENT Nome_Disciplina (#PCDATA)>
<!ELEMENT Numero_Disciplina (#PCDATA)>
<!ELEMENT Carga_Hor (#PCDATA)>
<!ELEMENT Nota (#PCDATA)>
```

Grafo gerado a partir da ontologia local que integrou as DTDs cadastro e histórico:



7 GLOSSÁRIO

API
Aplication Programming Interface. Uma especificação de como um programador escrevendo uma aplicação acessa o comportamento e estado de classes e objetos.

Applet
Um programa escrito na linguagem Java para rodar dentro de um *browser* compativel com a plataforma Java, como o Netscape Navigator.

Clicar
Neologismo para definir o ato de pressionar o botão do mouse para inicializar alguma ação ou selecionar um ponto da tela.

Framework
Estrutura, composição.

Grafo
Estrutura que representa relacionamentos entre diferentes objetos de maneira não hierárquica.

Hiperlink
Ligação entre dois recursos da *web*.

HTML
HyperText Markup Language. Linguagem de marcação, usada para criar documentos de hipertexto que são portáveis de uma plataforma para outra.

Input
A informação que é apresentada para o computador./ Enviar a informação para um computador, para processamento ou armazenamento.

Layout
Arranjo, plano. / Planejamento gráfico.

Parser
Analisador sintático e decompositor gramatical de uma seqüência de caracteres.

Parsing
Ato de utilizar um *parser*. Ver *parser*.

Screenshot
Imagem capturada, mostrando um estado de toda a tela do computador.

Servlet
Um programa que funciona do lado dos servidores (*server-side*) que provê funcionalidades adicionais para servidores que utilizam tecnologia Java.

Tag
Etiqueta. / Parte de uma marcação em linguagens de marcação, com HTML.

Thesaurus
Coleção de sinônimos.

Token
Sinal, símbolo.

Web
Ver WWW.

Wrapper

Ferramenta encarregada de atender a consultas e atualizações a uma determinada fonte de dados.

WWW

A rede de sistemas e os dados contidos neles que formam a Internet.

8 BIBLIOGRAFIA

[BAT 86] BATINI, C.; LENZERINI, M.; NAVATHE S. A Comparative Analysis of Methodologies for Database Schema Integration. **ACM Computing Surveys**, Vol 18, Nº 4. Dezembro de 1986.

[BON 94] BONJOUR, Michel; FALQUET, Gilles. **Concept Bases: A Support to Information Systems Integration.** CAiSe '94 Conference, Utrech. 1994. Disponível por WWW em: http://cui.unige.ch/db-research/Members/mb/papers/caise94/CAISE94_1.html

[COV 99] COVER, Robin. **Managing Names and Ontologies: An XML Registry and Repository.** 1999. Disponível por WWW em www.oasis-open.org/cover.

[ETH 00] ETHOS. The Extensible Markup Language (XML) - **ETHOS Technology Briefings Series 1: Developments Shaping Internets and Intranets**. 2000, European Telematics Horizontal Observatory Service. Disponível por WWW em: www.oasis-open.org/cover/bryanEthos980120.html

[FOW 00] FOWLER, Martin; SCOTT, Kendall . **Uml Essencial - Um Breve Guia Para A Linguagem-Padrao De Modelagem De Objetos.** São Paulo: Bookman Companhia Ed, 2000. 169p.

[GAR 99] GARSHOL, Lars. **Introduction to XML.** 1999. Disponível por WWW em: www.stud.ifi.uio.no/~lmariusg/download/xml/xml_eng.html

[GRU 92] GRUBER, Thomas. **Toward Principles for the Design of Ontologies Used for Knowledge Sharing.** In Formal Ontology in Conceptual Analysis and Knowledge Representation, Padova – Italy, 1992.

[GUA 95] GUARINO, Nicola. **Formal Ontology, Conceptual Analysis and Knowledge Representation.** International Journal of Human and Computer Studies, special issue on The Role of Formal Ontology in the Information Technology, vol 43 no. 5/6, 1995.

[IBM 00] IBM - UFRGS**. Projeto IBM-UFRGS de Acesso Integrado a Bases de Dados Legadas.** 17 de Agosto de 2000, Porto Alegre: Universidade Federal do Rio Grande do Sul, Instituto de Informática. Disponível por FTP em: ftp.heuser.inf.ufrgs.br/pub/pdf/SolectronOverview.pdf

[IBM 99] IBM. **Graph Foundation Classes for Java (GFC) - Version 1.1.2.** 1999. Disponível por WWW em: www.alphaworks.ibm.com

[ISO 86] ISO. **Standard Generalized Markup Language (SGML)**. International Organization for Standarization , 1986.

[JOH 00] JOHNSON, Mark. **Programming XML in Java.** Março de 2000. Disponível por WWW em: www.javaworld.com/jw-03-2000/jw-03-xmlsax.html

[KAM 91] KAMBAYASHI, M.; RUSINKIEWICZ, A.P. **First International Workshop on Interoperability in Multidatabase Systems**. Kyoto, 1991.

[KAR 99] KARP, Peter; CHAUDHRI, Vinay; THOMERE, Jerome. **XOL: An XML-Based Ontology Exchange Language**. Agosto de 1999, Pangea Systems. Disponível por FTP em: ftp://smi.stanford.edu/pub/bio-ontology/xol.doc

[MEG 98] MEGGINSON, David. **Sax: The Simple API for XML**. Maio de 1998. Disponível por WWW em: www.megginson.com/SAX/index.html

[OMG 97] OMG. **IDL - Interface Definition Language**. 1997, Object Management Group (OMG). Disponível por WWW em: http://cgi.omg.org/cgi-bin/doc?omg/97-05-03.txt

[OMG 99] OMG. **OMG Unified Modeling Language Specification.** Junho de 1999, Object Management Group (OMG). Disponível por WWW em: www.omg.org/uml/

[ORA 00] ORACLE. **Oracle XML Parser 2.0.2.9.0.** 11 de Julho de 2000. Disponível por WWW em: www.oracle.com/xml

[PIM 00] PIMENTEL M.; TEIXEIRA, C; SANTANCHÈ A. **XML: Explorando suas aplicações na Web**. Anais da XIX Jornada de Atualização em Informática - SBC, 2000, Curitiba. ed Champagnat, pg. 1-43.

[SAN 00] SANTI, Sérgio M., **Ontologias - Abordagem de Construção e Aplicações,** 2000. Trabalho Individual I - Porto Alegre, UFRGS.

[SIL 99] SILBERCHATZ, Abraham; KORTH, Henry; SUDARSHAN S. Introdução. In: **Sistemas de Bancos de Dados.** São Paulo: Makron Books, 1999. 778p. p8.

[STE 00] STEP. **Introduction to XML.** 2000, Stürtz Electronic Publishing GmbH. Disponível por WWW em: http://www.oasis-open.org/html/step_intro_to_xml.html

[STU 00] STUDER, Rudi; ERDMANN, M. **How to Structure and Access XML Documents With Ontologies.** Institut für Angewandte Informatik und Formale Beschreibungsverfahren (AIFB) - Universät Karlsruhe (TH), 2000.

[STU 98] STUDER, Rudi; FENSEL, Dieter; DECKER, Stefan; BENJAMINS, Richard. **Knowledge Engineering: Survey and Future Directions.** Universität Karlsruhe, Institut AIFB, 1998.

[SUN 99] SUN MICROSYSTENS**. The Java Tutorial**. 1999. Disponível em WWW em: http://java.sun.com/docs/books/tutorial

[W3C 00a] W3C. **XML Schema Part 0: Primer**. Abril de 2000, World Wide Web Consortium (W3C). Disponível por WWW em: www.w3c.org/TR/xml/schema-0/

[W3C 00b] W3C. **Extensible Stylescheet Language (XSL) - Version 1.0**. 27 de Março de 2000, World Wide Web Consortium(W3C). Disponível por WWW em: www.w3c.org/TR/xsl/

[W3C 00c] W3C. **XML Linking Language (XLink)**. 21 de Fevereiro de 2000, World Wide Web Consortium(W3C). Disponível por WWW em: www.w3c.org/TR/XLink/

[W3C 00d] W3C. **Resource Description Framework (RDF), Model and Syntax Specification**. 22 de Fevereiro de 2000, World Wide Web Consortium(W3C). Disponível por WWW em: www.w3c.org/TR/REC-xml-names/

[W3C 98a] W3C. **XML - W3C Recommendation.** 10 de fevereiro de 1998, World Wide Web Consortium (W3C). Disponível por WWW em: www.w3c.org/TR/REC-mis/

[W3C 98b] W3C. **Document Object Model (DOM) Level 1 Specification - Version 1.0.** 1º de Outubro de 1998, World Wide Web Consortium (W3C). Disponível por WWW em: www.w3c.org/TR/REC-DOM-Level-1/

[W3C 99] W3C. **Namespaces in XML, W3C Recommendation** . 14 de Janeiro de 1999, World Wide Web Consortium(W3C). Disponível por WWW em: www.w3c.org/TR/REC-xml-names/

[WAL 97] WALSH, Norman. **A Technical Introduction to XML.** Novembro de 1997. Disponível por WWW em: www.arbortext.com/Think_Tank/XML_Resources/A_Technical_Introduction_to_XM/body_a_technical_introduction_to_xm.html