Gerador Automático de Editores XML Baseado no XML Schema

André Vital Saúde

27 de junho de 2003

UNIVERSIDADE ESTADUAL DE CAMPINAS - UNICAMP
FACULDADE DE ENGENHARIA ELÉTRICA E DE COMPUTAÇÃO

GERADOR AUTOMÁTICO DE EDITORES XML BASEADO NO XML SCHEMA

Autor: **André Vital Saúde**

Orientador: **Prof. Dr. Roberto de Alencar Lotufo**

Comissão Julgadora: **Prof. Dr. Luiz Eduardo Buzato**
Prof. Dr. Ivan Luiz Marques Ricarte

Dissertação de Mestrado apresentada à Faculdade de Engenharia Elétrica e de Computação como parte dos requisitos para obtenção do título de Mestre em Engenharia Elétrica. Área de concentração: **Engenharia de Computação**.

27 de junho de 2003
Campinas, SP - Brasil

FICHA CATALOGRÁFICA ELABORADA PELA
BIBLIOTECA DA ÁREA DE ENGENHARIA - BAE - UNICAMP

Sa85g

Saúde, André Vital
Gerador automático de editores XML baseado no XML schema / André Vital Saúde. --Campinas, SP:[s.n.], 2003.

Orientador: Roberto de Alencar Lotufo
Dissertação (mestrado) - Universidade Estadual de Campinas, Faculdade de Engenharia Elétrica e de Computação.

1. Programas de computador. 2. Engenharia de software. 3. XML (linguagem de marcação de documento). 4. Programação orientada a objetos (Computação). 5. Editor de textos (Programas de Computador). I. Lofufo, Roberto de Alencar. II. Universidade Estadual de Campinas. Faculdade de Engenharia Elétrica e de Computação. III. Título.

Resumo

Este documento descreve a concepção e implementação do Gerador Automático de Editores XML Baseado no XML Schema. O Gerador de Editores XML é uma ferramenta de suporte ao desenvolvimento de aplicações científicas que integra o Ambiente para Desenvolvimento de Software Científico (Adesso).

O Adesso é um ambiente computacional que explora o modelo de programação baseado em componentes reutilizáveis para fornecer suporte ao desenvolvimento de componentes e sua integração a diversas plataformas de programação científica. Possui uma base de dados de componentes representada em XML e um conjunto de ferramentas de transformação para geração automática de código, documentação e empacotamento dos componentes.

Todos os sistemas que permitem ao usuário a edição direta de uma base de dados representada em XML precisam de um editor com interface gráfica de alto nível para que a manipulação dos dados seja feita com clareza e robustez. É ainda importante que o ambiente de edição seja capaz de fazer validações imediatas do conteúdo modificado pelo usuário.

Analisando as vantagens e desvantagens de usar um editor XML genérico, como os oferecidos gratuitamente pelo mercado, ou de desenvolver um editor específico para cada aplicação XML, é possível identificar uma solução que agrega os pontos positivos de cada uma das anteriores.

Usando ferramentas de transformação XML, gera-se automaticamente uma interface gráfica a partir da definição do modelo de dados da aplicação: o XML Schema. A interface gerada utiliza um modelo orientado a objetos para a representação gráfica dos tipos definidos em um XML Schema e é subdividida em módulos que permitem fácil adaptação do gerador de editores para ser utilizado em ambientes distribuídos.

Abstract

This document describes the conception and implementation of XML Editor Generator Based on the XML Schema. The XML Editor Generator is a support tool for the development of scientific software which integrates the Scientific Software Development Environment (Adesso).

The Adesso system is a computacional environment that leverages the reusable software component programming model to support the development and integration of the components to several scientific programming platforms. It is based on an XML represented components database and a set of XML document transformation tools for the automatic generation of component code, documentation and packaging.

Every system which allows user direct edition of an XML represented database needs an editor with a high-level Graphical User Interface (GUI) for data manipulation to be clear and robust. The capability of online validations of the modified content is also very important.

Confronting the possibilities of using a generic XML editor, like the most common free XML editors, or developing a specific editor for each XML application, it is possible to identify a third approach that agregates the main advantages of both others.

By using the XML tranformation tools one can automatically generate a GUI based on the application data model definition: the XML Schema. The generated GUI uses an object-oriented model for graphical representations of the XML Schema datatypes and is subdivided in modules so that the XML Editor Generator can be easily adapted to distributed systems.

A meus pais e mestres, Eulália e Edimar.

Agradecimentos

Primeiramente, um agradecimento especial a Rubens C. Machado, do Centro de Pesquisa Renato Archer (CenPRA), e ao professor Dr. Roberto A. Lotufo, meu orientador, que esforçaram-se juntos para orientar este trabalho e que contribuíram consideravelmente ao meu processo de formação científica.

Aos meus irmãos, Tânus e Nízia, pelo grande prazer e esforço em constituir, junto a nossos pais e a mim, a família alegre e unida que sempre fomos. A todos os meus tios, primos, avós e cunhados, cujos nomes, infelizmente não caberiam nesta página.

À minha querida Sheila, companheira sempre presente, por estar há anos compartilhando minhas emoções e reflexões, por ter me acompanhado e apoiado durante este trabalho desde seu verdadeiro início, por me dar amor, carinho, atenção e compreensão.

Aos meus dedos, unhas e às cordas do meu violão, pelos momentos de calma e meditação que me propiciam.

Às rochas, paredes e amigos de escalada. Aos amigos de cerveja, de festa, de diálogos. Aos amigos próximos ou distantes. Aos que estudaram, cresceram, cantaram ou riram comigo. Enfim, a todos aqueles que me transmitem boas energias em um abraço. Aos meus verdadeiros amigos.

A Deus, pelo privilégio que me dá de olhar para as linhas acima e estar seguro de que são poucas. Por impedir-me de medir esforços para o trabalho com toda a felicidade que me concede.

Sumário

Lista de Figuras

Capítulo 1

Motivação

Cada vez mais, cenários complexos que há algum tempo só existiam em filmes de ficção científica estão se tornando realidade devido ao incrível aumento da capacidade computacional e de comunicação de dados. Reconhecimento de faces, busca por conteúdo de imagens e vídeo, sistemas automáticos de vigilância eletrônica, monitoramento de tráfego, detecção precoce de tumores são apenas alguns dos recentes resultados de projetos da comunidade científica na área de processamento de imagens e reconhecimento de padrões.

Os principais resultados de projetos de pesquisa em áreas como processamento de sinais, visão computacional, reconhecimento de padrões, controle de processos, bioinformática e outras afins, são métodos e algoritmos inovadores. Ao longo destes projetos, programas de demonstração são desenvolvidos em torno desses algoritmos para a comprovação de sua viabilidade. Estes programas, embora não constituam o principal resultado de um projeto de pesquisa, são, em muitos casos, o embrião de projetos de desenvolvimento que podem levar à criação de produtos inovadores.

A criação de aplicações científicas de qualidade envolve, porém, mais que a implementação de algoritmos e métodos. Uma aplicação de sucesso deve ter, além de algoritmos de alta qualidade científica, uma implementação robusta e atualizada tecnologicamente, em conformidade com padrões de qualidade ditados pelas práticas correntes. No

atual estágio da tecnologia da informação, estes padrões são cada vez mais exigentes, principalmente em termos de interfaces gráficas sofisticadas, integração com outras aplicações, qualidade da documentação de usuário e segurança de informações.

Por outro lado, ao longo do processo de criação de protótipos científicos, grande esforço é desperdiçado na repetição do desenvolvimento de programas devido a freqüentes mudanças de padrões e tecnologias de software, temporariedade da associação de pesquisadores ao projeto, falta de documentação adequada que permita a utilização e aperfeiçoamento do software e, finalmente, aos altos custos da manutenção do software.

Todos estes fatores conduzem a modelos de desenvolvimento de software em que o reaproveitamento de código exerce um papel fundamental. Um desses modelos, cada vez mais utilizado, estabelece que componentes preexistentes são interconectados através de plataformas integradoras para a criação de soluções para problemas específicos. A funcionalidade total exigida de uma aplicação é fatorada entre sub-aplicações com interfaces bem definidas — componentes — visando sua reutilização em outros problemas.

Tendo em vista este quadro e visando amenizar os problemas do desenvolvimento de software científico, foi desenvolvido um sistema denominado **Adesso** — Ambiente para Desenvolvimento de Software Científico [1, 2]. O Adesso utiliza-se de transformações de documentos XML com o objetivo de prover ferramentas para a criação, manutenção documentação e distribuição de conjuntos de componentes de software.

1.1 O Sistema Adesso

O Adesso é um sistema de autoria de programas científicos baseado no paradigma componente/solução onde conjuntos de componentes são desenvolvidos e empacotados em *toolboxes* com interfaces programáticas bem definidas. Em uma outra fase, estes componentes são conectados através de uma *plataforma de integração*, normalmente linguagens de *scripting*, para a criação de uma aplicação.

Uma **toolbox** é constituída por um conjunto ortogonal de informações, suficientes para que um *produto* seja gerado. Um *produto* é formado pelo código (binário ou fonte, de-

pendendo da distribuição) e documentação necessários para a utilização da *toolbox* em determinada *plataforma de integração* sob determinada *plataforma computacional*. Estes *produtos* são criados através de ferramentas de transformação de documentos XML.

Assim, por exemplo, uma *toolbox* de processamento de imagens pode ser a base de um produto que permite sua utilização a partir do MATLAB em sistemas Microsoft Windows ou pode originar outro produto para Python/Linux. No estágio atual, o Adesso dispõe de geradores de código e documentação para distribuições em C, C++, Python, Tcl/Tk e MATLAB, inclusive para uso com o *compilador MATLAB*.

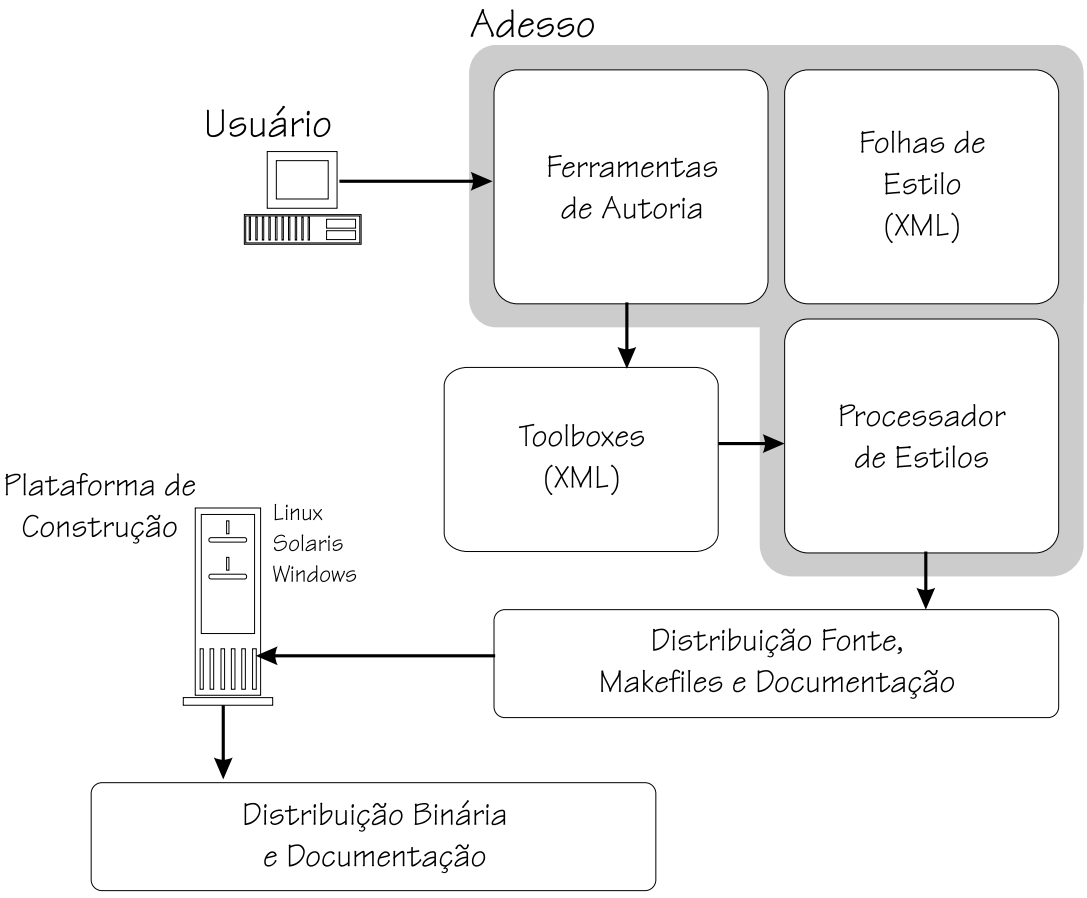

Figura 1.1: Organização do Adesso.

A Figura 1.1 mostra a organização do sistema Adesso. O Adesso já foi utilizado para a implementação de diversos produtos, incluindo uma *toolbox* de morfologia matemática denominada **SDC Morphology Toolbox**[1], uma *toolbox* de processamento de imagens[3] e uma ferramenta interativa para segmentação de objetos em vídeo[4]. A base de dados deste sistema utiliza para os componentes de software uma representação baseada na linguagem de *markup* **XML**[5].

XML é uma linguagem de *markup* para a definição e comunicação de dados estruturados através de arquivos de texto. XML especifica um conjunto de regras e convenções para o projeto de textos que representem informações de forma a facilitar a geração e o entendimento (por computadores), evitar ambigüidades e dependências de plataformas computacionais e facilitar a troca de informações entre computadores.

A base de dados do Adesso foi modelada usando a linguagem de definição de esquemas do XML (*XML Schema Definition Language*, [6, 7, 8]). A norma **XML Schema** especifica uma linguagem, baseada em XML, para a definição de esquemas de representação de dados. O propósito de um esquema XML é definir e descrever uma classe de documentos XML através da utilização dos componentes da linguagem para restringir e documentar o significado, a utilização e as relações entre as partes constituintes do documento: tipos de dados, elementos e seu conteúdo e atributos e seus valores. Em resumo, um XML Schema define um vocabulário para uma classe de documentos XML.

A partir das informações de sua base de dados e utilizando ferramentas de transformação inspiradas no modelo definido pela especificação **XSLT** (*Extensible Stylesheet Language Transform* [9]), o Adesso é capaz de gerar interfaces para diversas plataformas de *scripting* e programação de alto nível além de uma variedade de formatos de documentação. Para tanto, o Adesso dispõe de uma *Linguagem de Estilos* que, como a linguagem XSLT, provê construções, com sintaxe XML, para especificar a transformação de um documento XML em outro(s) documento(s).

Várias transformações são aplicadas aos componentes de uma *toolbox* para a geração de uma *distribuição fonte*. Uma distribuição fonte é constituída pelo código fonte ge-

[1] http://www.mmorph.com

rado para cada componente, pela sua documentação, *scripts* para orientar o processo de construção dos binários, etc. Uma *Distribuição Binária* pode então ser construída, a partir da distribuição fonte, em diferentes plataformas computacionais (Unix, Windows) através de ferramentas usuais de compilação de programas.

Todos os processos de transformação efetuados pelo Adesso são realizados através de seu **Processador de Estilos**. Este componente aceita como entrada arquivos da base de dados de uma *toolbox* juntamente com uma *folha de estilo* — um arquivo XML que descreve uma transformação a ser efetuada nos dados de entrada através da *linguagem de estilos*. O Adesso tem um conjunto extensível de folhas de estilo para a geração de interfaces e documentação para algumas plataformas de *scripting* como Tcl, Matlab e Python.

Para a criação e manutenção de seus componentes e a aplicação de transformações, o Adesso é dotado de um conjunto de ferramentas de autoria que constituem o ambiente de desenvolvimento de *toolboxes*. O ambiente de desenvolvimento inclui o produto deste trabalho: o **Gerador de Editores XML**, um gerador automático de editores XML baseado no modelo da base de dados, descrito pelo XML Schema. Todas as ferramentas de geração de código e documentação são acessíveis através do sistema de autoria.

1.2 O Ambiente de Desenvolvimento

Para abstrair a forma como os dados de uma *toolbox* são armazenados e tratados pelo sistema Adesso, é importante que o usuário tenha à sua disposição uma interface gráfica que represente esta *toolbox* da melhor maneira possível.

A base de dados do Adesso utiliza uma representação baseada na linguagem de *markup* XML. Para o usuário, porém, a abstração da forma de armazenamento dos dados é desejável. Por esta razão, independente da escolha de representação em XML, a existência de um aplicativo totalmente especializado na edição de *toolboxes*, seria ideal. Um editor projetado e desenvolvido para um problema específico certamente possui uma relação muito fiel entre as estruturas de dados e a forma de apresentá-las. O sistema Adesso

utilizou por um bom tempo um editor especializado como esses, mas foi constatado um alto custo em sua manutenção. Isso porque o modelo de dados de uma *toolbox* é modificado com bastante freqüência e a readaptação do editor necessita grande trabalho de reprogramação.

Para safar-se do alto custo de manutenção de um editor especializado, o projeto Adesso passou a utilizar editores XML genéricos como ambiente de desenvolvimento. Existe uma variedade de editores XML genéricos distribuídos gratuitamente. Um editor genérico possui um formato padrão de apresentação dos dados, independente de seus tipos ou estruturas. Esta limitação não associa o documento XML editado ao contexto da aplicação, obrigando o usuário a conhecer em detalhes o modelo de dados.

Neste trabalho foi desenvolvida uma ferramenta de edição de documentos XML que visa unir a fidelidade entre a apresentação dos dados e seus tipos, presente nos editores especializados, ao baixíssimo custo de manutenção dos editores genéricos. Esta ferramenta, denominada **Gerador de Editores XML**, utiliza algumas folhas de estilo para fazer a geração automática do código da interface gráfica a partir do modelo dos dados. Como a linguagem usada para modelar os dados, o XML Schema, é baseada no próprio XML, o modelo pode ser transformado via XSLT. O desenvolvimento desta ferramenta de edição consistiu, basicamente, na codificação das folhas de estilo que associam um XML Schema qualquer a uma interface gráfica em linguagem Java. Com a existência desta ferramenta, qualquer atualização do modelo de dados implica, somente, na atualização do XML Schema e no processo automático de transformações que geram uma nova interface gráfica. O resultado das transformações é um editor que garante uma apresentação amigável, de forte relação com o modelo de dados, e ao mesmo tempo de baixíssimo custo de manutenção, já que é gerado automaticamente.

1.3 Organização deste Documento

Algumas das principais tecnologias relacionadas com este trabalho são apresentadas no Capítulo 2. As possíveis maneiras de se editar um documento XML são mostradas no

Capítulo 3.

As informações que constituem um documento XML são modeladas através de um XML Schema. O uso do XML Schema e a criação de um diagrama de classes que o representa são discutidos no Capítulo 4.

A principal contribuição deste trabalho ao sistema Adesso e ao processo de desenvolvimento de software científico é a geração automática de um editor com interface gráfica amigável a partir de transformações de um XML Schema. Este processo de geração automática de código é discutido no Capítulo 5. Em seguida, no Capítulo 6, são apresentadas outras possíveis aplicações dos produtos concebidos por este trabalho.

O Capítulo 7 posiciona o Gerador de Editores XML no escopo de um novo projeto, o AdessoWeb, que visa a extensão do Adesso para permitir seu funcionamento em ambiente distribuído. Finalmente, no Capítulo 8, apresentamos algumas conclusões a respeito do trabalho realizado e sugerimos alguns projetos para o aperfeiçoamento do Gerador de Editores XML.

Capítulo 2

Principais Tecnologias Utilizadas

Com o intuito de construir um sistema portável para o intercâmbio e manipulação de documentos, foi criada na década de 60 a linguagem **SGML — Standard Generalized Markup Language** [10], a precursora das linguagens de marcação hoje existentes. O SGML não possui uma semântica predefinida e funciona, na realidade, como uma metalinguagem para a especificação de outras linguagens de marcação.

A sintaxe de uma linguagem específica de marcação definida através de SGML é especificada através de um **DTD — Document Type Definition**. O DTD define uma gramática para a linguagem de marcação do documento, ou seja, as possíveis organizações das marcações SGML do documento.

Por ser uma linguagem muito genérica, o SGML é hoje pouco usado diretamente. Diversas linguagens foram derivadas do SGML para contextos específicos e normalmente substituem a precursora por serem de mais fácil manipulação. É o caso do uso do XML no sistema Adesso, ao invés do SGML.

2.1 XML — Extensible Markup Language

XML — Extensible Markup Language [5] é uma linguagem de *markup* para a definição e comunicação de dados estruturados através de arquivos de texto. Dados estruturados permeiam os sistemas computacionais atuais. Planilhas, parâmetros de configuração, desenhos técnicos, *bookmarks*, transações financeiras são exemplos de dados estruturados produzidos e/ou consumidos por aplicações computacionais. XML especifica um conjunto de regras e convenções para o projeto de textos que representem tais tipos de dados de forma a facilitar a geração e o entendimento (por computadores), evitar ambigüidades e dependências de plataformas computacionais e facilitar a troca de informações entre computadores.

Um arquivo XML se parece muito com um arquivo na linguagem **HTML — Hyper Text Markup Language**. XML, todavia, é muito diferente do HTML em seus objetivos. A marcação da linguagem HTML visa especificar a aparência de seu conteúdo. A marcação XML explicita as relações estruturais existentes entre as partes componentes do documento através de etiquetas de marcação (*tags*) cuja semântica é associada ao domínio do documento. HTML oferece um conjunto de etiquetas pré-definido. XML é um método para aplicar marcadores cujo significado depende do domínio da aplicação. Ambas as linguagens de *markup* têm relação direta com o SGML. O HTML é um vocabulário específico, definido a partir do SGML, para a apresentação de documentos. O XML é um padrão equivalente ao SGML, com simplificações para facilitar seu uso.

Para ilustrar melhor, consideremos o seguinte documento XML, que define a sintaxe de utilização, os tipos de cada argumento e o valor de retorno de uma função em C.

```
<?xml version="1.0"?>
<AdFunction name="processImage">
  <!-- função criada por AVS -->
  <Return type="Image" name="out" output="no">
    Soma das imagens (out = i0 + i1)
  </Return>
  <Args>
```

```
    <Arg name="i0" type="Image" constraint="GRAY"/>
    <Arg name="i1" type="Image" constraint="GRAY"/>
    <Arg name="i2" type="Image">
      i2 = out = i0 + i1
    </Arg>
    <Arg name="i3" type="Int" optional="yes"
                                    default="99"/>
  </Args>
  <Source lang="C" file="src/processImage.c"/>
</AdFunction>
```

A definição da função *processImage* é formada por três conjuntos de definições correspondentes aos elementos *Return*, *Args* e *Source*.

O elemento *Return* especifica que a função retorna um valor do tipo *Image*, posto que *type="Image"*. Contudo, não deve ser retornado à plataforma com a qual a função é interfaceada (output="no"). Estas definições são feitas através de símbolos que são associados a valores e são denominados atributos.

A definição dos argumentos da função é feita através do elemento *Args*, que é formado por quatro elementos *Arg*. Elementos como *Args*, que podem conter outros elementos, são de *tipos complexos*, enquanto que os elementos que não podem conter outros elementos nem atributos são de *tipos simples* (estas definições estão associadas ao esquema do documento, discutido na próxima seção). Atributos sempre têm valores de tipos simples.

O elemento *Args* define que a função *processImage* aceita quatro argumentos. O argumento *i3* é opcional e, caso não seja especificado, assume o valor padrão *99*. Como o atributo *optional* não é especificado nos outros três argumentos, eles são obrigatórios.

Finalmente, o elemento *Source* especifica, através do atributo *file*, um arquivo que contém a implementação da função e, através do atributo *lang*, a linguagem de programação utilizada na implementação.

A constituição dos tipos complexos e de alguns tipos simples (existe um conjunto de

tipos simples pré-definidos) utilizados em um documento é especificada no esquema associado ao documento. Esta associação esquema-documento normalmente é definida pelo próprio documento através de um marcador específico para este fim (no exemplo, o esquema não está especificado no documento).

2.2 XML Schema

A norma **XML Schema** [6, 7, 8] especifica uma linguagem, baseada em XML, para a definição de esquemas de representação de dados. O propósito de um XML Schema é o mesmo de um DTD, ou seja, definir e descrever uma classe de documentos XML. No entanto, diferentemente do DTD, o XML Schema utiliza componentes da própria linguagem XML (meta-linguagem) para restringir e documentar o significado, a utilização e as relações entre as partes constituintes do documento: tipos de dados, elementos e seu conteúdo, atributos e seus valores. Além disso, é possível definir uma gramática mais rigorosa e detalhada com um XML Schema do que com um DTD. O XML Schema vem substituindo os DTDs por esta razão.

O XML Schema seguinte descreve um vocabulário para a definição de funções. O documento XML apresentado anteriormente está em conformidade com este fragmento de esquema.

```
<xs:schema>
  <xs:element name="AdFunction" type="AdFunctionType"/>

  <xs:complexType name="AdFunctionType">
    <xs:sequence>
      <xs:element name="Return">
        <xs:complexType>
          <xs:attribute name="name" type="xs:NMTOKEN"/>
          <xs:attribute name="type" type="typeRef"/>
          <xs:attribute name="output" type="xs:boolean"/>
```

```
        </xs:complexType>
      </xs:element>
      <xs:element name="Args">
        <xs:complexType>
          <xs:element name="Arg" minOcurrs="0"
                                  maxOcurrs="unbounded">
            <xs:complexType>
              <xs:attribute name="name" type="xs:NMTOKEN"/>
              <xs:attribute name="type" type="typeRef"/>
              <xs:attribute name="optional"
                                      type="xs:boolean"/>
              <xs:attribute name="default"
                                      type="xs:string"/>
              <xs:attribute name="constraint"
                                      type="xs:string"/>
            </xs:complexType>
          </xs:element>
        </xs:complexType>
      </xs:element>
      <xs:element name="Source">
        <xs:complexType>
          <xs:attribute name="lang" type="LangType"/>
          <xs:attribute name="file" type="FileID"/>
        </xs:complexType>
      </xs:element>
    </xs:sequence>
    <xs:attribute name="name" type="FunctionID"/>
  </xs:complexType>

  <xs:simpleType name="FunctionID">
    <xs:restriction base="xs:NMTOKEN">
      <xs:maxLength value="20"/>
```

```
    </xs:restriction>
  </xs:simpleType>

  <xs:simpleType name="langType">
    <xs:restriction base="xs:NMTOKEN">
      <xs:enumeration value="C"/>
      <xs:enumeration value="Matlab"/>
      <xs:enumeration value="Tcl"/>
      <xs:enumeration value="Python"/>
    </xs:restriction>
  </xs:simpleType>
...
</xs:schema>
```

O esquema é formado pelo elemento *schema* e seus sub-elementos, notadamente os elementos **complexType**, **simpleType** e **element**.

A linha `<element name="AdFunction" type="AdFunctionType"/>` indica que a constituição do elemento *AdFunction* (raiz do documento) é ditada pela definição do tipo *AdFunctionType*. A construção que se segue é a definição deste tipo.

AdFunctionType é um tipo complexo, definido com ajuda do elemento *complexType*: possui apenas um atributo, *name*, que é do tipo *FunctionID*, definido na seqüência com o auxílio do elemento *simpleType*.

Esta definição estabelece que o valor do atributo é uma **derivação por restrição** de *NMTOKEN* (tipo simples predefinido), ou seja, *FunctionID* é ainda um *NMTOKEN*, porém restrita a, no máximo, 20 caracteres.

Além do atributo *FunctionID*, o tipo complexo *AdFunctionType* (e portanto o elemento *AdFunction*) é formado por uma seqüência de três elementos *Return*, *Args* e *Source*. Cada um destes elementos é, por sua vez, um tipo complexo. Note, no entanto, que estes tipos são definidos imediata e anonimamente, isto é, sem especificar um nome para o tipo. O formato da definição é similar.

Um dos atributos do elemento *Source* é o atributo *lang*, do tipo *LangType*. Este tipo é um tipo enumerado, ou seja, o atributo *lang* pode assumir apenas um dos valores listados. Este exemplo suporta apenas as linguagens C, Matlab, Tcl e Python.

O XML Schema é a tecnologia utilizada para a definição dos tipos e estruturas de dados de um documento XML. É dele, portanto, que devem ser extraídas as informações necessárias para a geração automática de uma interface gráfica. Esta tecnologia é melhor detalhada no Capítulo 4, onde é discutida a modelagem de um esquema para a geração automática de código.

2.3 DOM — Document Object Model

A estrutura de um documento XML pode ser comparada à estrutura de um sistema de arquivos tradicionalmente utilizado nos sistemas operacionais. Ambos podem ser representados por uma árvore onde os nós intermediários são os diretórios, no caso do sistema de arquivos, ou os elementos, no caso do XML. As folhas desta árvore representam os diferentes tipos de arquivos ou o texto e atributos do XML. O modelo **DOM — Document Object Model** é usado para fazer a representação, em árvore, de um documento XML.

O seguinte documento XML pode ser representado como na Figura 2.1.

```
<?xml version="1.0"?>
<AdFunction name="processImage">
  <!-- função criada por AVS -->
  <Return type="Image" name="out" output="no">
    Soma das imagens (out = i0 + i1)
  </Return>
  <Args>
    <Arg name="i0" type="Image" constraint="GRAY"/>
    <Arg name="i1" type="Image" constraint="GRAY"/>
    <Arg name="i2" type="Image">
```

```
      i2 = out = i0 + i1
    </Arg>
    <Arg name="i3" type="Int" optional="yes"
                                    default="99"/>
  </Args>
  <Source lang="C" file="src/processImage.c"/>
</AdFunction>
```

A Figura 2.1 mostra os diferentes tipos de nós definidos na especificação da linguagem XML, com exceção do nó tipo *namespace*, não utilizado no exemplo. Na representação gráfica foi incluído, ainda, um nó raiz pai de todos os demais. Os seis tipos de nós de uma árvore XML são os seguintes:

1. **Elementos**
   Os elementos estruturam a árvore XML, visto que são os únicos nós que podem ter filhos. Correspondem aos marcadores da linguagem.

2. **Texto**
   Estes nós contêm o texto dos elementos XML.

3. **Atributos**
   Estes nós correspondem aos pares nome/valor especificados nos marcadores de abertura dos elementos XML.

4. **Instruções de Processamento**
   No documento XML, aparecem como construções do tipo `<?...?>`

5. **Comentários**
   São representados como texto do tipo `<!-- ... -->` no documento.

6. **Espaços de Nomes** (não aparecem na figura)
   Estes nós contêm a definição de espaços de nomes usados para caracterizar e diferenciar os nomes de marcadores em diferentes contextos.

2.4 XPath

O **XPath** [11], é uma forma de localizar partes de um documento XML. O nome se origina de sua semelhança com os caminhos utilizados para localização de arquivos e

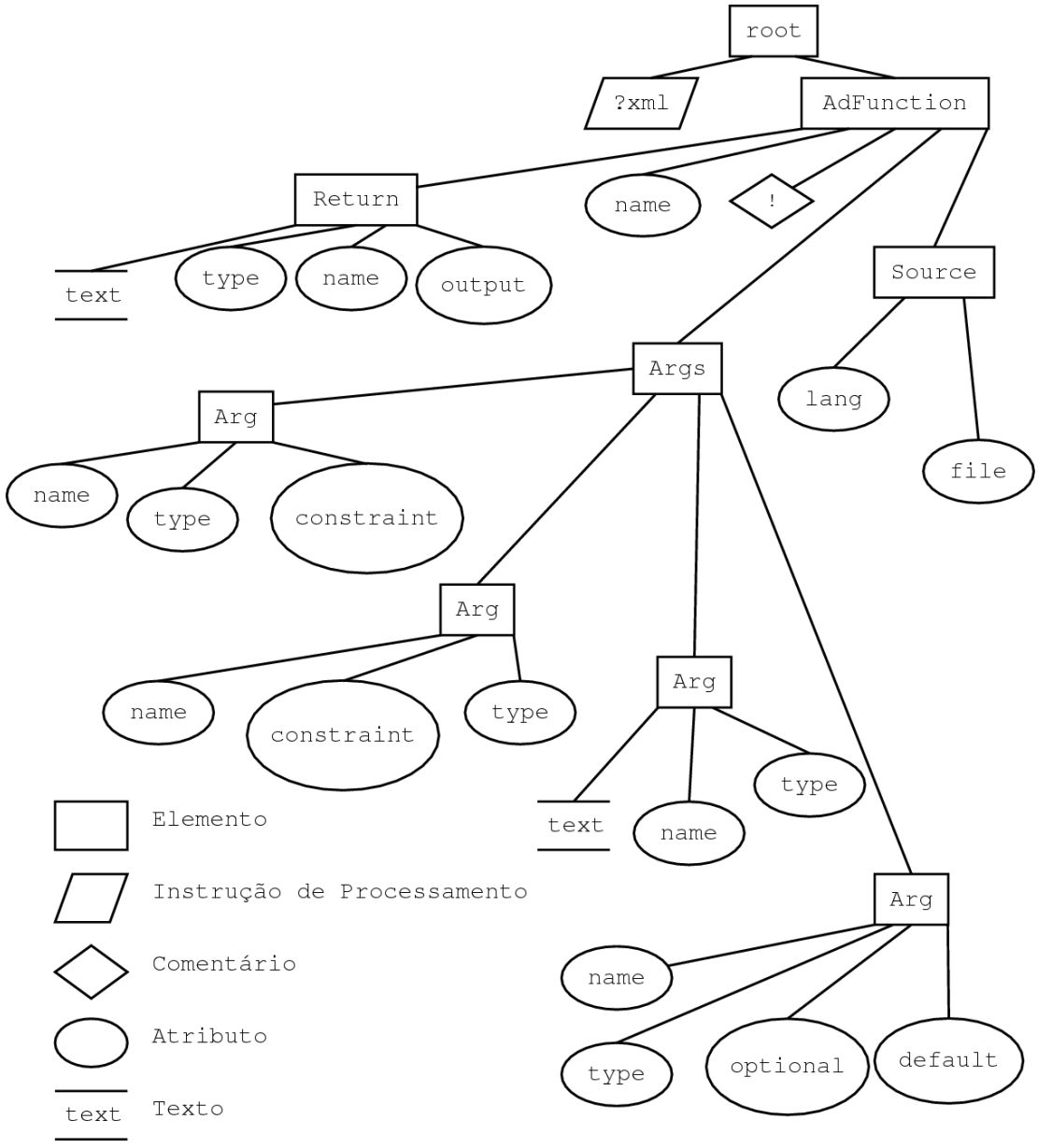

Figura 2.1: Árvore DOM.

diretórios em um sistema de arquivos.

A construção sintática básica de uma expressão **XPath** é o *location path* ou **caminho**. À maneira do caminho de um arquivo, um *location path* pode ser absoluto, quando o caminho começa pelo nó raiz, ou relativo, no caso da referência ser o *nó corrente*. O *nó corrente* é estabelecido pela aplicação que se utiliza do **XPath**. Por exemplo, em um processador XSLT o nó corrente é aquele que constitui o contexto onde um *template* é avaliado.

O *Extensible Stylesheet Language* (XSL) apresentado na próxima seção está totalmente vinculado ao XPath. O uso adequado das ferramentas *XSL Transformation* (XSLT) exige um conhecimento prévio desta norma.

Os exemplos a seguir ilustram a utilização de expressões XPath. Os comandos são aplicados ao texto XML representado na Figura 2.1:

- `/AdFunction/Return`
  seleciona o nó que contém o elemento *Return*. Note a utilização de nomes no teste (*AdFunction* e *Return*) que, no caso, referem-se aos marcadores dos elementos.

- `/AdFunction/Return/@name`
  seleciona o atributo *name* do elemento *Return*. Aqui, o caractere @ indica que *name* é um **atributo**.

- `/AdFunction/*`
  seleciona todos os elementos filhos de *AdFunction*.

- `Arg[1]`
  seleciona o primeiro elemento de nome *Arg*. Note que, aqui, a expressão não começa com `"/"`. Isso significa que ela não é avaliada a partir da raiz e sim do *nó corrente*. No exemplo, esta expressão só não retorna *vazio* se o nó corrente for o nó *Args*. A expressão `./Arg[1]` é equivalente à expressão `Arg[1]`.

- `//Arg`
  seleciona todos os elementos *Arg* da árvore. O elemento raiz não possui nenhum

descendente (filho) de nome *Arg*, porém, o uso de `"//"` ao invés de `"/"` indica que a busca deve ser feita em toda a profundidade da árvore. Se `".//"` fosse usado, a busca seria efetuada em toda a sub que tem como raiz o nó corrente.

- `..`
  seleciona o nó situado um nível acima do nó corrente. Se o nó corrente é um dos elementos *Arg*, refere-se ao elemento *Args*.

  - se o contexto é dado pelo elemento *Return*, o caminho
    ..//Arg seleciona todos os elementos *Arg* situados abaixo de *AdFunction*.

2.5 XSL — Extensible Stylesheet Language

A linguagem **XSL — Extensible Stylesheet Language**[12], é constituída por dois componentes: uma linguagem de transformação (**XSLT** [9]) e uma linguagem de formatação. Ambas as linguagens são definidas como uma aplicação XML, ou seja, utilizam a sintaxe XML. A linguagem de transformação provê construções para especificar a transformação de um documento XML em outro documento. A segunda linguagem especifica marcadores para caracterizar *objetos de formatação*.

Examinaremos em seguida algumas características da linguagem de transformação cujos princípios são utilizados como base para a criação dos geradores de código do Adesso. Para exemplificar o uso do XSL, consideremos que queremos criar um documento HTML que documente a função C descrita em XML na seção anterior.

A Figura 2.2 apresenta o modelo para uma transformação XSL. Um processador XSL aceita como entradas o documento XML a ser transformado e uma **folha de estilo**, escrita em XSLT, e cria um novo documento com a marcação desejada. O código a seguir é um exemplo de folha de estilo para transformar a definição de uma função C como a apresentada anteriormente em um documento HTML.

```
<xsl:stylesheet>
  <xsl:template match="AdFunction">
```

```
    <html>
    <head><title>
    Função <xsl:value-of select="@name"/>
    </title></head>
      <body>
        <h1>Função <b><xsl:value-of
                            select="@name"/></b></h1>
        <p><xsl:value-of select="text()"/></p>
        <h2>Argumentos</h2>
        <xsl:apply-templates select="Args"/>
        <h2>Retorno</h2>
        <xsl:apply-templates select="Return"/>
      </body>
    </html>
  </xsl:template>

  <xsl:template match="Args">
    <dl>
      <xsl:apply-templates select="Arg"/>
    </dl>
  </xsl:template>
```

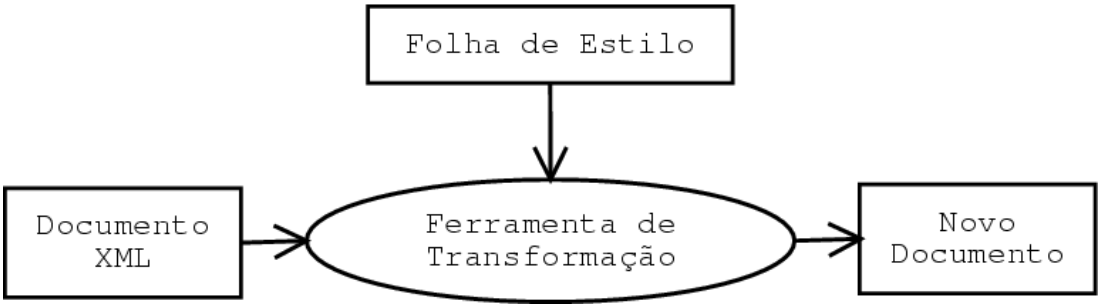

Figura 2.2: Processamento de Estilo.

```
  <xsl:template match="Arg">
    <dt>
      <xsl:value-of select="@type"/> <xsl:value-of
                                        select="@name"/>
    </dt>
    <dd>
      <xsl:value-of select="text()"/>
    </dd>
  </xsl:template>
</xsl:stylesheet>
```

O processador XSL realiza sua função através da aplicação de *templates* a nós da árvore correspondente ao documento XML de entrada. Cada definição de *template* indica, através do atributo **match**, os elementos do documento a que ela se aplica. Durante o processamento existe sempre um *nó corrente* que define o contexto para a realização de buscas.

O processamento deste estilo se inicia com o *template* que tem o atributo *match* com o valor *AdFunction*, que corresponde ao elemento raiz do documento XML. A árvore XML contida no *template* é transferida para o documento de saída depois que os elementos da linguagem XSLT, (*xsl:*), são processados.

Os elementos com marcador *xsl:value-of* ocasionam a seleção de algum componente do documento de entrada, indicado no atributo *select*, e a sua substituição pelo valor do componente. Assim, por exemplo, `<xsl:value-of select="@name"/>` faz com que seja inserido no documento de saída o valor do atributo *name* ("@x" significa "atributo x") do nó em processamento, no caso o nó .

Os elementos com marcador *xsl:apply-templates* fazem com que os *templates* definidos no estilo sejam aplicados aos nós filhos do nó corrente, de acordo com seu atributo *match*. O elemento `<xsl:apply-templates select="Args"/>` causa a aplicação do `<xsl:template match="Args"> ... </xsl:template>` ao nó , filho de AdFunction.

O documento gerado como saída do processamento é apresentado a seguir.

```
<html>
<head><title>
Função processImage
</title></head>
  <body>
    <h1>Função <b>processImage</b></h1>
    <p>Esta função efetua...</p>
    <h2>Argumentos</h2>
    <dl>
      <dt>Image i0</dt><dd></dd>
      <dt>Image i1</dt><dd></dd>
      <dt>Image i2</dt><dd>i2 = out = i0 + i1</dd>
      <dt>Int i3</dt><dd></dd>
    </dl>
    <h2>Retorno</h2>
    <p>...</p>
  </body>
</html>
```

2.6 O Processador de Estilos do Adesso

O **Processador de Estilos** do Adesso [1] é o elemento central da arquitetura do processo de transformação do sistema Adesso. É baseado na transformação de documentos XML inspirado na linguagem XSL (*Extensible Stylesheet Language*).

O modelo de transformação utilizado pelo processador de estilos do Adesso faz substituições de texto à maneira dos processadores de macros. O modelo aproveita o sistema de substituição implementado pelo *parser* da linguagem *Tcl* e faz uso intensivo da interface *DOM* e da linguagem de *query XPath*. O resultado dessa combinação é um sistema poderoso de transformação de documentos XML.

Dois motivos levaram à implementação deste processador de estilos em vez de apro-

veitar o suporte oferecido pelo XSL. À época desta decisão, a linguagem XSLT ainda não se encontrava estabilizada e era descrita apenas em documentos de trabalho muito instáveis. Por outro lado, a necessidade de geração de código, *makefiles* e outras saídas não convencionais sugeria a necessidade da flexibilidade oferecida pelas linguagens de *scripting*. O casamento natural Tcl/XML encorajou a criação do processador de estilos.

Atualmente, mesmo com a crescente ampliação da oferta de processadores *XSLT*, o processador de estilos do Adesso e sua linguagem constituem uma alternativa mais flexível para a especificação de transformações, principalmente para programadores com experiência na linguagem *Tcl*, porque permite que o usuário crie extensões do gerador de código nesta linguagem. Para programadores que não conhecem *Tcl*, a linguagem de estilos do Adesso torna-se equivalente à XSL.

O *XSLT*, porém, deve tornar-se uma linguagem padrão para transformação de documentos. Com o surgimento de pessoal treinado em *XSLT*, deve haver maior utilização do XSLT para a criação de estilos para o Adesso.

2.7 Java

Existem hoje muitas ferramentas gráficas de edição de documentos XML. Algumas delas gratuitas, como o bem conhecido **Xeena**[13] da IBM. Mas já existem também ferramentas comerciais muito poderosas, como é o caso do **XML Spy**[14].

O Gerador de Editores XML é mais uma ferramenta para edição de documentos XML. No entanto, é uma ferramenta que gera automaticamente um ambiente gráfico de edição a partir da representação da estrutura dos dados do documento XML, ou seja, a partir do XML Schema que o descreve. Basicamente, o editor gerado é uma interface capaz de representar graficamente os tipos de dados definidos num XML Schema.

Tendo em vista os objetivos e a estratégia de implementação do Gerador de Editores XML, alguns aspectos são importantes no momento da escolha da linguagem de programação a ser usada:

- **Orientação a Objetos**: O XML Schema permite a definição de um tipo através da derivação de um tipo já existente por extensão ou restrição, permitindo que tipos derivados herdem a definição do tipo base. Esta facilidade do XML Schema pode ser facilmente representada por uma linguagem orientada a objetos.

- **Programação de Alto Nível**: Tratando-se de uma interface gráfica, é importante que a linguagem a ser utilizada seja de alto nível, com componentes gráficos poderosos já implementados. O uso de um **IDE** — *Integrated Development Environment* — para a implementação da interface gráfica não é viável por tratar-se de um editor gerado automaticamente e não de um editor programado manualmente. Através de uma IDE não é possível ter uma visualização gráfica dos componentes a serem gerados por folhas de estilo.

- **Ferramentas XML**: A disponibilidade de ferramentas robustas de processamento de documentos XML é imprescindível. Já existem **parsers** XML para diversas linguagens, entre elas, Perl, C++, Python, e o próprio Tcl, usado pelo Processador de Estilos do Adesso. Os *parsers* da família Apache **Xerces** [15] são reconhecidos pela comunidade XML como muito eficientes e robustos. Possuem implementações em C++ e Java, além de um *wrapper* para a linguagem de *scripting* Perl.

- **Portabilidade**: Sendo o XML um padrão em texto simples, do qual os possíveis usuários usam diferentes plataformas (Windows, Solaris, Linux, etc.) a portabilidade torna-se um quesito importante para não restringir o uso do sistema. O Processador de Estilos do Adesso, do qual o Gerador de Editores depende, está implementado em Tcl, uma linguagem de *scripting* naturalmente portável. Da mesma maneira, espera-se que o Gerador de Editores XML também seja portável.

Java[16] é uma linguagem de programação que reúne todas essas características, ou seja, é portável, orientada a objetos e tem disponível pacotes robustos de processamento XML e de programação gráfica. Por estas razões, optou-se pelo uso desta linguagem na criação do Gerador de Editores XML.

Capítulo 3

Edição de Documentos XML

Apesar de representado em texto simples e compreensível ao olho humano, XML não visa a sua leitura e entendimento por pessoas. É voltado, sim, para sua utilização por computadores. O fato de ser representado em arquivos de texto facilita a manutenção dos dados, possibilitando a correção de problemas, em situações anormais, com a utilização de editores de texto simples. Contudo, XML é verboso e extremamente sensível a erros de sintaxe — o que desaconselha sua edição direta.

Outra decorrência da representação na forma de texto é que arquivos XML tendem a se tornar volumosos. Contudo esta é uma decisão de projeto baseada, principalmente, nas vantagens advindas desta representação e na disponibilidade de ferramentas de compressão de dados.

O usuário de qualquer sistema desejaria abstrair ao máximo a forma como os dados por ele editados são armazenados e tratados pelo computador. No sistema Adesso, por exemplo, o usuário está preocupado em editar o conteúdo de sua *toolbox* e, para isso, preferirá, certamente, utilizar uma interface gráfica especializada na edição de *toolboxes*, que melhor apresente esta *toolbox*.

É clara, portanto, a necessidade de que os sistemas baseados em XML possuam um editor com interface gráfica amigável para a manipulação de dados. Neste capítulo, são discutidas três possibilidades de editores XML a serem utilizados.

3.1 Um Editor Especializado

A necessidade de um editor com interface gráfica amigável não está ligada ao uso do XML como linguagem de representação dos dados. Esta necessidade está vinculada à maior produtividade de um usuário que concentra sua atenção no conteúdo do seu trabalho sem se preocupar com a sintaxe de uma determinada linguagem.

A existência de um aplicativo totalmente especializado na edição do conteúdo seria o ideal. Um editor projetado e desenvolvido para um problema específico certamente implica uma relação muito fiel entre as estruturas de dados e a forma de apresentá-las. Para que um editor como este, com a mais amigável das apresentações, possa ser desenvolvido, é necessário aplicar processos de desenvolvimento de software que passam pelas fases de análise, projeto e implementação. Isto é requerido para qualquer tipo de software e pode, portanto, parecer um problema pouco oneroso. O sistema Adesso utilizou por um bom tempo um editor especializado como esses.

O inconveniente está na necessidade de uma análise prévia muito bem feita das estruturas de dados. Esta análise exige que as estruturas de dados sejam definidas com antecedência e que não sofram mais mudanças. Qualquer mudança nas estruturas poderia acarretar grandes inconsistências no editor, sendo talvez necessário até mesmo que todo o processo fosse refeito.

A desvantagem de definir previamente as estruturas de dados é que, na prática, à medida que o tempo passa, novas necessidades vão surgindo e os dados pedem uma reorganização e/ou uma reestruturação. As mudanças podem ser tão grandes ou tão freqüentes que os custos de manutenção de um editor especializado tornam-se inviáveis.

3.2 Um Editor XML Genérico

As tecnologias XML foram tão bem aceitas pelo mercado que muitas organizações preocuparam-se em desenvolver ferramentas a elas relacionadas. No esforço de ocul-

tar o formato texto dos documentos XML, surgiu uma variedade de visualizadores e editores com interface gráfica amigável.

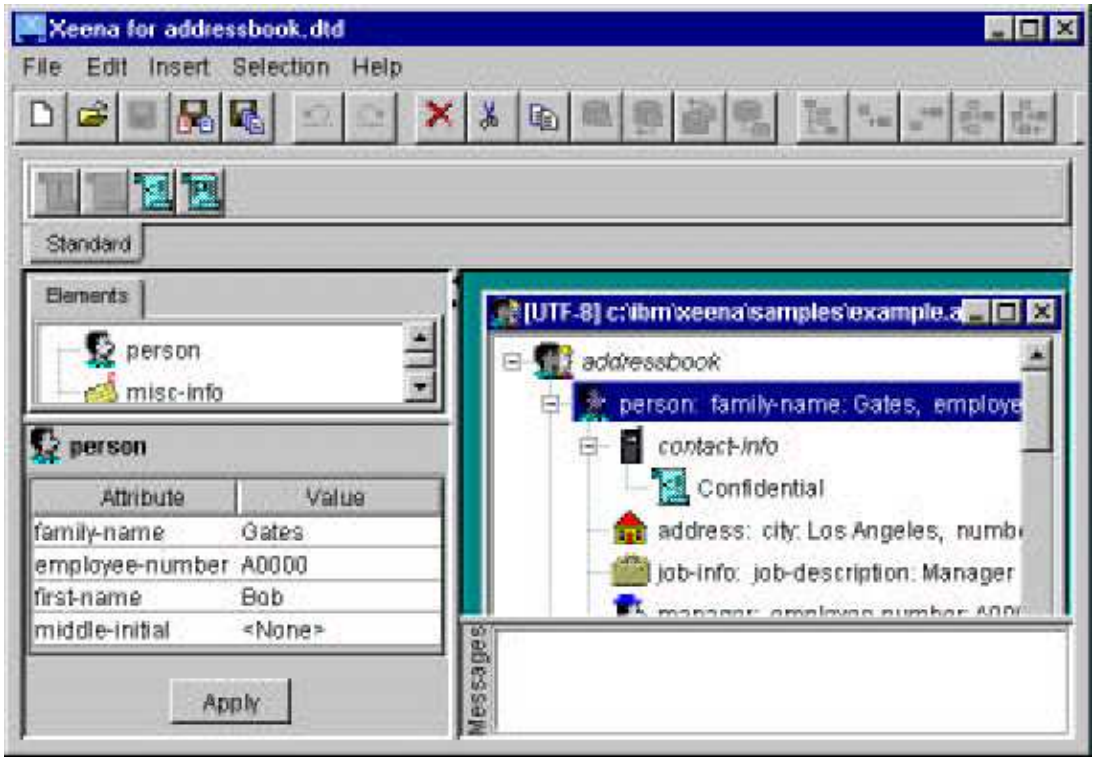

Figura 3.1: Xeena. Um editor XML genérico gratuito distribuído pela IBM.

A Figura 3.1 mostra o Xeena[13], um editor XML gratuito distribuído pela IBM. Igualmente, a maioria dos outros editores XML disponíveis são distribuídos gratuitamente. Esses editores, além de apresentarem a grande vantagem de serem gratuitos, funcionam em qualquer aplicação porque são editores genéricos. Um editor genérico é aquele que mostra qualquer documento XML usando um formato de apresentação padronizado. Este formato, na maioria das vezes, é uma árvore, semelhante a uma árvore de diretórios, que representa a estrutura do documento, ou seja, sua hierarquia de elementos, como já foi mostrado na Figura 2.1. Para cada elemento da árvore são apresentados os atributos. Os valores dos atributos podem ser editados na forma de cadeia de caracteres (strings),

ou seja, no mesmo formato como são escritos no documento XML. Como pode ser visto na Figura 3.1, o Xeena não segue exatamente esse padrão, mas o princípio é o mesmo. A maioria desses editores é também capaz de fazer a validação do documento editado confrontando-o com um DTD ou um XML Schema. No entanto, a principal desvantagem deles é o formato padronizado da apresentação, que pouco difere da edição em formato texto.

Um bom exemplo desta limitação são as chaves e referências definidas em um XML Schema, como discutido no Capítulo 4. Suponha que no documento XML editado exista a chave K e a referência KR associada a K. Usando um editor XML genérico é impossível, sem que o usuário faça uma busca manual, conhecer os valores já atribuídos à chave K para então fazer uma referência KR. No caso de um editor específico, o usuário teria uma lista de chaves já atribuídas para fazer uma escolha. O usuário é impossibilitado de fazer referência a um valor inexistente.

Apesar destas limitações, o caráter "software livre" dos editores genéricos fez com que, após reconhecer o alto custo de manutenção de um editor específico, o sistema Adesso passasse a adotá-los. Os editores genéricos têm sido utilizados nos últimos dois anos.

3.3 Um Editor Facilmente Configurável

Analisando as vantagens e desvantagens dos tipos de editores já apresentados, a conclusão que se chega é que a melhor opção é usar um editor com uma interface gráfica tão amigável quanto a de um editor especializado mas que, em caso de mudança no modelo de dados, não necessite ser reprogramado. Um editor especializado que fosse facilmente configurável ou, pensando ainda mais longe, um editor especializado que se reconfigurasse automaticamente. Para ser ainda mais claro, este seria um editor cuja interface gráfica assumiria uma forma diferente para cada diferente aplicação em que estivesse atuando, ou seja, um editor que analisaria a priori o modelo de dados a ser editado e apresentaria uma interface gráfica adequada para aquele modelo. Considerando que o XML Schema é a representação do modelo de dados de um documento XML,

neste trabalho foi desenvolvido um Gerador Automático de Editores XML Baseado no XML Schema. No Capítulo 5 são apresentados detalhes para a construção deste editor.

Capítulo 4

O XML Schema como um Diagrama de Classes

A modelagem de um XML Schema em um **diagrama de classes** é a idéia básica que resultou no modelo do Gerador de Editores XML. Um diagrama de classes é um modelo genérico do problema e pode ser implementado em qualquer linguagem orientada a objetos. Se existe a possibilidade de representar um XML Schema por um diagrama de classes, é possível, portanto, definir regras para transformar um esquema em um código que implementa o modelo de classes.

Para melhor compreensão da modelagem de um XML Schema através de um diagrama de classes, apresentamos a seguir um estudo de caso que explora detalhes do uso do XML Schema.

4.1 Estudo de Caso: o Modelo de Toolboxes do Adesso

A fim de modelar as informações de uma **toolbox** de maneira neutra e facilmente acessível a programas escritos em diversas linguagens de programação, o Adesso utiliza uma representação das informações em XML. A utilização do XML facilita a manutenção

dos dados e possibilita a utilização de uma grande (e crescente) variedade de ferramentas desenvolvidas por terceiros para edição, validação e transformação das informações.

Esta representação é formalizada através de um documento XML Schema. Os principais elementos desse esquema são apresentados nas seções seguintes.

4.1.1 A Estrutura dos Dados

A Figura 4.1 é uma captura de tela de um editor gerado pelo próprio aplicativo desenvolvido neste trabalho e mostra o exemplo de uma *toolbox*. A árvore da figura está apenas parcialmente expandida.

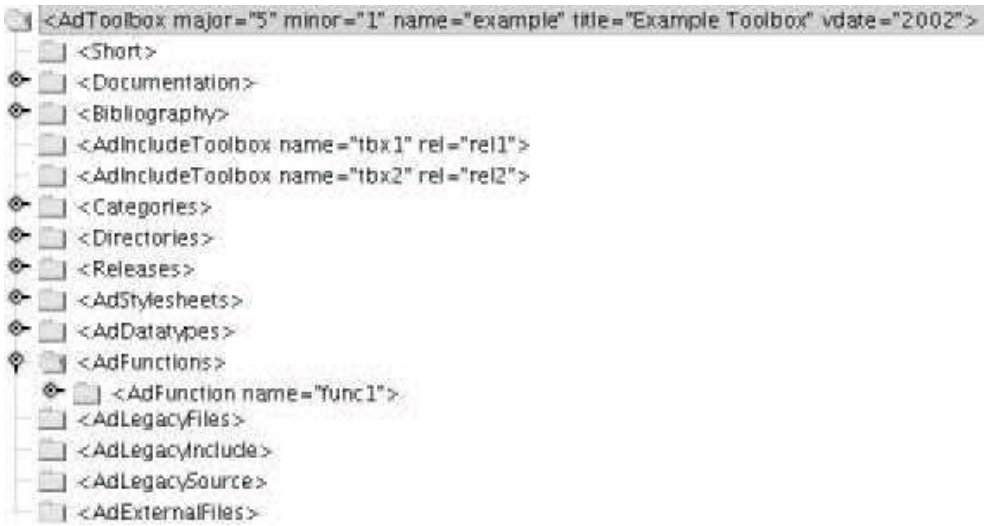

Figura 4.1: Composição de uma *toolbox*.

O **elemento raiz** de um documento XML que representa uma *toolbox* é *AdToolbox*. Uma *toolbox* contém informações sobre o conjunto de componentes que a constitui. Estas informações podem ser classificadas em seis grupos:

- **Identificação da Toolbox**
  Dados que especificam o nome da *toolbox*, prefixo para os nomes dos componentes, versão, etc. Esses dados são mostrados no mesmo nó da árvore onde está o

elemento *AdToolbox*. São atributos da *toolbox*. O elemento *Short*, primeiro filho de *AdToolbox*, é uma descrição resumida (em uma linha) da *toolbox* e também faz parte da sua identificação.

- **Documentação Global da Toolbox**
  Informações para a criação da documentação sobre o conjunto de componentes. O elemento *Documentation* permite a criação de documentos com marcação XML possibilitando sua transformação para diferentes formatos. *Bibliography* é a lista de referências bibliográficas utilizadas na documentação da *toolbox*.

- **Classificação dos Componentes**
  Permitem criar categorias (elemento *Categories*) de componentes para fins de documentação, separação em diretórios (elemento *Directories*) e criação de diferentes distribuições (elemento *Releases*).

- **Componentes da Toolbox**
  Coleção de elementos que especificam cada componente da *toolbox*. Tipos de dados e funções. São os elementos *AdDatatypes* e *AdFunctions*.

- **Dependências Externas**
  Enumera dependências externas como arquivos de dados, bibliotecas, arquivos de inclusão etc. São os elementos *AdIncludeToolbox*, *AdLegacyFiles*, *AdLegacyInclude*, *AdLegacySource* e *AdExternalFiles*.

- **Folhas de Estilo Específicas**
  O elemento *AdStylesheets* contém folhas de estilo e mapas de tipos de dados que implementam particularidades da *toolbox*, sobrecarregando as definições preexistentes.

A Figura 4.1 ainda mostra parte da sub-árvore de *AdFunctions*, que inclui um elemento *AdFunction*.

4.1.2 Definição da Estrutura com o XML Schema

A estrutura dos dados de uma *toolbox* pode ser representada usando o XML Schema [6, 7, 8]. O fragmento de esquema a seguir representa corretamente a **estrutura de uma toolbox**. A *toolbox* é mostrada na Figura 4.1.

```
<xs:element name="AdToolbox">
  <xs:complexType>
    <xs:sequence>
      <xs:element ref="Short"/>
      <xs:element ref="Documentation"/>
      <xs:element name="Bibliography" type="BibliographyType"/>
      <xs:element name="AdIncludeToolbox"
                        type="AdIncludeToolboxType"
                        minOccurs="0" maxOccurs="unbounded"/>
      <xs:element name="Categories" type="CategoriesType"/>
      <xs:element name="Directories" type="DirectoriesType"/>
      <xs:element name="Releases" type="ReleasesType"/>
      <xs:element name="AdStylesheets"
                                  type="AdStylesheetsType"/>
      <xs:element name="AdDatatypes" type="AdDatatypesType"/>
      <xs:element ref="AdFunctions"/>
      <xs:element name="AdLegacyFiles"
                                  type="AdLegacyFilesType"/>
      <xs:element name="AdLegacyInclude"
                                  type="AdLegacyIncludeType"/>
      <xs:element name="AdLegacySource"
                                  type="adtext" minOccurs="0"/>
      <xs:element name="AdExternalFiles"
                                  type="AdExternalFilesType"/>
    </xs:sequence>
    <xs:attribute name="name" type="xs:ID" use="required"/>
    <xs:attribute name="title" type="xs:string"
```

```
                                              use="required"/>
    <xs:attribute name="major" type="xs:byte" use="required"/>
    <xs:attribute name="minor" type="xs:byte" use="required"/>
    <xs:attribute name="vdate" type="xs:string"
                                              use="required"/>
    <xs:attribute name="patchlevel" type="xs:string"/>
    <xs:attribute name="cvs" type="xs:string"/>
    <xs:attribute name="cvsdate" type="xs:string"/>
    <xs:attribute name="prefix">
      <xs:simpleType>
        <xs:restriction base="xs:string">
          <xs:maxLength value="4"/>
          <xs:minLength value="2"/>
        </xs:restriction>
      </xs:simpleType>
    </xs:attribute>
  </xs:complexType>

  <!-- key and keyref declarations -->

</xs:element>
```

Este fragmento de esquema define o elemento *AdToolbox* como sendo um tipo complexo (**complexType**) composto de uma seqüência (*xs:sequence*) de elementos e um conjunto de atributos. É importante analisar detalhadamente este fragmentos para observar as possibilidades que o XML Schema nos oferece. Note que, sendo o XML Schema também um documento XML, as nomenclaturas podem se confundir. No exemplo, a linha `<xs:attribute name="cvsdate" type="string"/>` é, na ótica do XML, um **elemento** de nome *attribute* e com atributos *name* e *type*. Já na ótica do XML Schema, esta linha é a **declaração** de um atributo de nome *cvsdate* e tipo *string*. É importante não confundir *atributo* com *declaração de atributo* e *elemento* com *declaração de elemento*.

Explicação detalhada do fragmento de esquema:

- **sequence**
  A partícula *sequence* é um dos possíveis modelos de agrupamento de elementos. Ela força a ordem de ocorrência dos elementos filhos de *AdToolbox*. Isso significa que um elemento *AdToolbox* cujos filhos não obedeçam esta ordem, não será válido. Esses modelos de agrupamentos são chamados **ModelGroups**. Além do **sequence** existem mais dois modelos de agrupamento: **all** e **choice**. O primeiro indica que todos os elementos podem ocorrer em qualquer ordem e o segundo, que apenas um dos elementos pode ocorrer.

- **ref**
  As declarações de elementos ou de atributos que possuem o atributo **ref** são, na verdade, referências a declarações feitas globalmente em outra parte do esquema.

- **type**
  No XML Schema, a estrutura de um elemento pode ser definida localmente, como é feito com o elemento *AdToolbox* ou através de um tipo global. O atributo **type** é o nome do tipo global que representa o elemento ou atributo em questão. Todas as definições de tipo são extensões ou restrições dos tipos *built-in* (**Built-in Datatypes**) do XML Schema. Tipos *built-in* são tipos predefinidos na norma do XML Schema. Os tipos *string*, *ID* e *byte* são exemplos de tipos *built-in*. A Figura 4.2, retirada da especificação do XML Schema[8], ilustra a hierarquia de definição de tipos no XML Schema.

- **minOccurs** e **maxOccurs**
  Aplicam-se apenas às declarações de elementos ou de modelos de agrupamentos e nunca a declarações de atributos. Os atributos **minOccurs** e **maxOccurs** são usados para limitar o mínimo e o máximo de ocorrências de uma determinada partícula na estrutura.

- **use**
  É equivalente aos atributos *minOccurs* e *maxOccurs* das declarações de elemen-

tos mas aplicam-se apenas às declarações de atributos. Como um atributo só pode ocorrer, no máximo, uma vez em um elemento, só é necessário saber se ele é opcional (quando **use** é omitido ou se *use="optional"*) ou obrigatório (se *use="required"*).

A título ilustrativo, apresentamos a seguir um fragmento de esquema que usa a partícula *choice*. O fragmento representa o tipo complexo *BibliographyType*.

```
<xs:complexType name="BibliographyType">
  <xs:choice minOccurs="0" maxOccurs="unbounded">
    <xs:element name="Article" type="ArticleType"/>
    <xs:element name="InProceedings" type="InProceedingsType"/>
    <xs:element name="Book" type="BookType"/>
  </xs:choice>
  <xs:attribute name="bibliographystyle"
                                type="BibliographyStyleType"/>
</xs:complexType>
```

Note que *minOccurs* e *maxOccurs* são atribuídos ao *Model Group choice*. Dentro do *choice*, apenas um dos elementos pode ocorrer. Mas, neste caso, como a estrutura *choice* pode ocorrer várias vezes, o resultado é a possibilidade de ocorrência ilimitada de qualquer um dos elementos em qualquer ordem.

Outro recurso importante do XML Schema a ser discutido é o uso de **key** e **keyref**. A especificação de chaves (*key*) permite indicar se um atributo ou elemento deve ser único dentro de um certo escopo. É possível também fazer referências (*keyref*) a essas chaves. O princípio é o mesmo usado em bancos de dados. Como exemplo, vamos definir uma chave e uma referência no escopo do elemento *AdToolbox*. Como já foram discutidos, os detalhes da definição do tipo do elemento foram omitidos deste exemplo. O fragmento resultante do esquema será:

```
<xs:element name="AdToolbox">
  <xs:complexType>
```

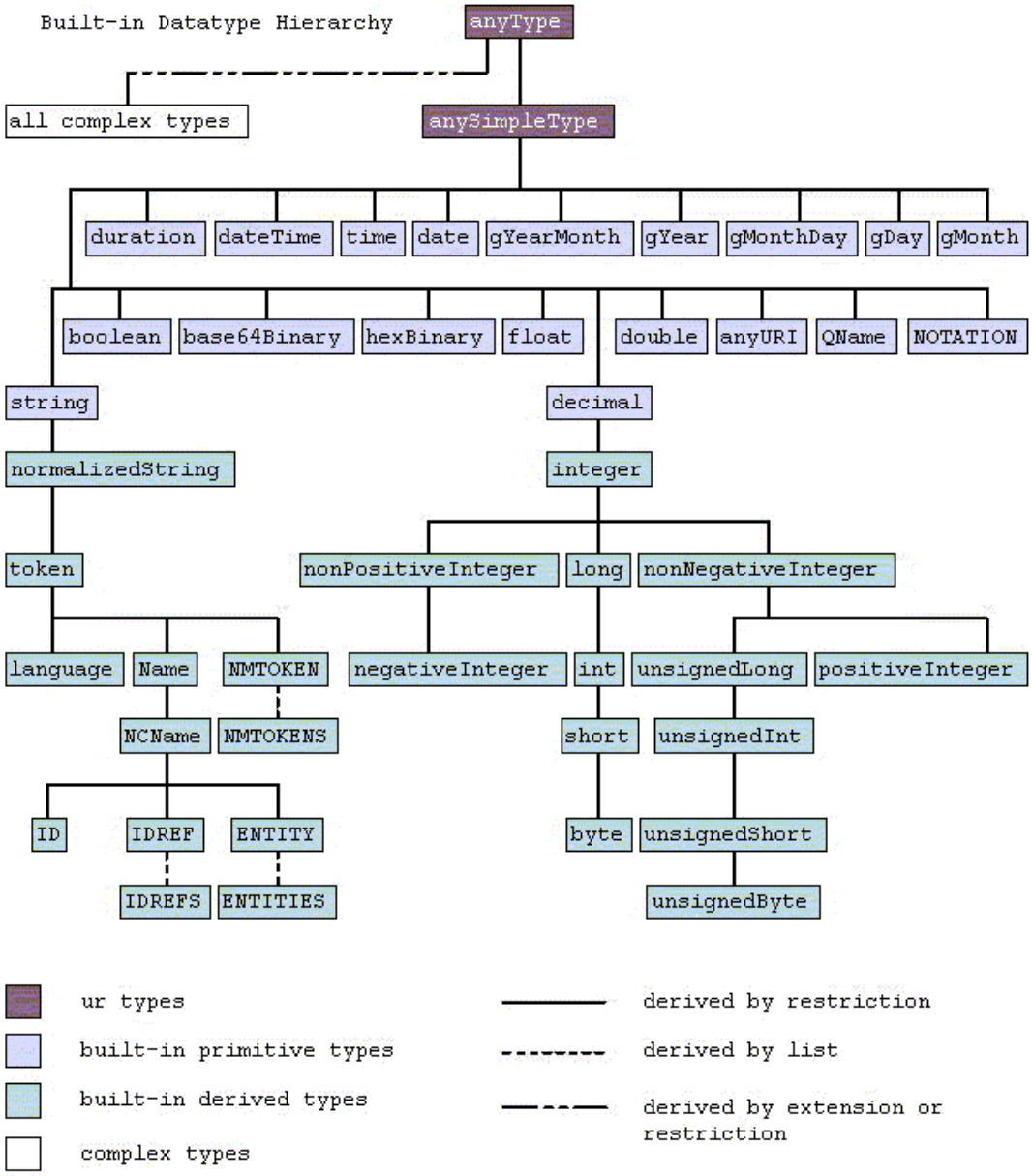

Figura 4.2: Hierarquia dos Tipos *builtin* no XML Schema.

```
    <!-- complexType details -->
  </xs:complexType>

  <xs:key name="catkey">
    <xs:selector xpath="Categories/Category"/>
    <xs:field xpath="@name"/>
  </xs:key>

  <xs:keyref name="catkeyref" refer="catkey">
    <xs:selector xpath="AdFunctions/*"/>
    <xs:field xpath="@cat"/>
  </xs:keyref>
</xs:element>
```

Primeiramente define-se uma chave (**key**) de nome **catkey**. Como nas linguagens de pesquisa em bancos de dados, existe um seletor (**selector**) e um campo (**field**). Ambos são definidos usando uma expressão XPath (Capítulo 2) relativa ao escopo. A declaração de *catkey* significa que o atributo *name* (field=@name) de qualquer elemento *Category*, filho de *Categories*, filho de *AdToolbox* são chaves e, portanto devem existir e ser únicos. Mais claramente, os nomes das categorias de uma *toolbox* não podem se repetir e não pode haver categoria sem nome.

Em seguida, define-se um **keyref** de nome **catkeyref** que faz referência à chave *catkey* (refer="catkey"). O seletor e o campo funcionam da mesma maneira que na partícula *key*. A declaração de *catkeyref* significa que o atributo *cat* (field=@cat) de qualquer elemento filho de *AdFunctions* (isso inclui todas as funções), filho de *AdToolbox*, é uma referência a *catkey*, ou seja, qualquer função da *toolbox* pode ser associada a uma categoria através do atributo *cat*. Só existem referências a categorias existentes.

O último ponto importante a ser analisado é a definição do tipo do atributo *prefix*, no elemento *AdToolbox*. Esta definição é semelhante à definição do tipo *FunctionID* discutido na Seção 2.2. Esta definição estabelece que o valor deste atributo é uma **derivação por**

restrição do tipo predefinido *string*, ou seja, *prefix* é, ainda, uma *string* porém restrita a, no mínimo, dois caracteres e, no máximo, quatro. As derivações de tipos simples podem ser por **restrição**, **lista** ou **união**. A derivação por restrição, como o próprio nome diz, gera um novo tipo com um número mais restrito de valores possíveis. A **derivação por lista** gera um novo tipo que é uma lista de valores correspondentes ao tipo base. A **derivação por união** gera um novo tipo que é a união de outros tipos, ou seja, um valor válido para este tipo é um valor válido para um ou mais dos tipos que compõem a união. Os tipos complexos podem sofrer uma diversidade maior de derivações. Podem ser derivados por restrição ou extensão, seja o tipo base um tipo simples ou um tipo complexo.

Com todos esses detalhes sobre o XML Schema, pode ser discutida sua modelagem em um diagrama de classes.

4.2 Modelagem do XML Schema

Nesta seção é discutida a modelagem de um XML Schema em um diagrama de classes. Esta modelagem é a base do Gerador de Editores XML. Os detalhes sobre a geração do código serão discutidos na Seção 5.4.

O que se busca aqui é fazer o caminho inverso ao mais comum. Normalmente, define-se o modelo dos dados de uma aplicação com um diagrama de classes e, a partir desse diagrama, escreve-se o esquema. Se todas as aplicações XML fossem desenvolvidas dessa forma, não haveria necessidade da análise do XML Schema para a criação de uma interface gráfica. Esta poderia ser criada diretamente do diagrama de classes. Todavia, o que acontece na prática é que o diagrama de classes não é feito, já que existem ferramentas que geram automaticamente um XML Schema após uma análise de um conjunto de instâncias XML.

Carlson [17] discute em detalhes a modelagem de aplicações XML usando **UML — Unified Modeling Language**. Mostra também como é possível fazer a geração automática de um XML Schema a partir do diagrama de classes e apresenta uma ferramenta,

denominada **hyperModel**, que integra um ambiente de edição UML a uma plataforma de transformação de códigos capaz de esquemas XML em diferentes padrões.

O *hyper*Model é capaz também de gerar um modelo UML a partir de um XML Schema dado. A ferramenta exporta o modelo para o formato **XMI — XML Metadata Interchange**[18], especificado pelo Object Management Group (OMG). XMI define um padrão para serialização de modelos UML como documentos XML. Ora, se é possível obter automaticamente um modelo UML descrito por um documento em formato XML, porque não usar as ferramentas XSLT para gerar o editor a partir do UML? Certamente este seria um caminho possível para a implementação do Gerador de Editores XML. Porém, esta escolha apresentaria alguns inconvenientes com relação à opção de fazer transformações diretamente do XML Schema:

- Diversas visualizações gráficas são possíveis para um mesmo esquema, assim como existem diferentes modelos UML para um mesmo esquema e diversas visualizações gráficas para um mesmo modelo UML. O passo intermediário de geração do modelo UML significaria uma multiplicação das variáveis para a geração da interface gráfica final.

- O objetivo deste projeto é a edição de dados armazenados em formato XML. A transformação direta do XML Schema supõe esta relação. O UML, por sua vez, pode definir qualquer tipo de aplicação, uma generalização que dificultaria o projeto com o aumento de variáveis.

- O *hyper*Model não é gratuito e não foi encontrada ferramenta semelhante de domínio público.

No Gerador de Editores XML, o modelo UML encontra-se codificado nas folhas de estilo que geram classes a partir de um XML Schema. A seção seguinte discute como isso é feito, a partir do mesmo estudo de caso da Seção 4.1.

4.2.1 O Modelo de um Esquema Simples

Para descrever o processo de modelagem pode-se começar pela análise da modelagem de um XML Schema bastante simplificado. Vamos supor que o XML Schema exemplo seja uma *toolbox* que possua apenas um nome, um título, algumas referências e algumas funções, como o fragmento a seguir:

```
<xs:schema>
  <xs:element name="AdToolbox">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="References" type="ReferencesType"/>
        <xs:element name="Functions">
          <xs:complexType>
            <xs:sequence minOccurs="0" maxOccurs="unbounded">
              <xs:element name="Function" type="FunctionType"/>
            </xs:sequence>
          </xs:complexType>
        </xs:element>
      </xs:sequence>
      <xs:attribute name="name" type="xs:ID" use="required"/>
      <xs:attribute name="title" type="xs:string"
                                              use="required"/>
    </xs:complexType>
  <xs:/element>

  <xs:complexType name="FunctionType">
    <xs:attribute name="name" type="xs:string" use="required"/>
    <xs:attribute name="lang" use="required">
      <xs:simpleType>
        <xs:restriction base="xs:string">
          <xs:enumeration value="C"/>
          <xs:enumeration value="Matlab"/>
```

```
          <xs:enumeration value="Python"/>
          <xs:enumeration value="Tcl"/>
        </xs:restriction>
      </xs:simpleType>
    </xs:attribute>
  </xs:complexType>

  <!-- ReferencesType definition -->
</xs:schema>
```

A Figura 4.3 mostra um diagrama de classes para este fragmento. Como o tipo *ReferencesType* ainda não foi definido, está representado como classe abstrata.

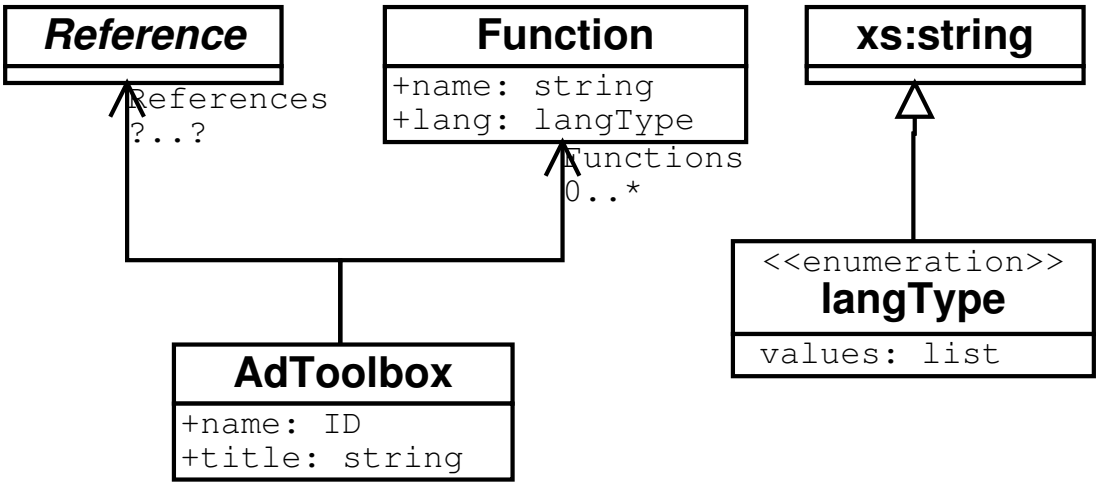

Figura 4.3: Modelo Incompleto de um XML Schema Simples.

A classe *AdToolbox* tem dois atributos e duas associações que definem sua estrutura. A classe abstrata *Reference* exerce o papel das referências. Sua multiplicidade ainda não está definida e foi escrita no modelo como *?..?*. Esta multiplicidade será ignorada, por enquanto, pois será expandida e discutida mais adiante. A classe *Function* exerce o papel das funções. A multiplicidade "0..*" está definida no *sequence*, dentro de *Functions*. Note que neste exemplo a definição do tipo do atributo *lang* é implícita, mas recebeu

um nome próprio (langType) no modelo UML e é uma classe que estende a classe que implementa o tipo predefinido *string*. O XML Schema não usa os conceitos de classes e propriedades. Possui apenas definições de tipos. Apesar disso, os tipos podem ser *estendidos* ou *restringidos* gerando subtipos. Todo e qualquer tipo que venha a ser definido em um esquema é uma extensão ou uma restrição dos tipos predefinidos.

Vamos agora mostrar a definição do tipo *ReferencesType*:

```
<xs:schema>
  <!-- AdToolbox and FunctionType definitions removed -->

  <xs:complexType name="ReferencesType">
    <xs:choice minOccurs="0" maxOccurs="unbounded">
      <xs:element ref="Article"/>
      <xs:element ref="Book"/>
    </xs:choice>
  </xs:complexType>

  <xs:element name="Article">
    <xs:complexType>
      <xs:attributeGroup ref="ReferenceAttributes"/>
      <xs:attribute name="journal" type="xs:string"
                                          use="required"/>
    </xs:complexType>
  </xs:element>

  <xs:element name="Book">
    <xs:complexType>
      <xs:attributeGroup ref="ReferenceAttributes"/>
      <xs:attribute name="publisher" type="xs:string"
                                          use="required"/>
    </xs:complexType>
  </xs:element>
```

```
  <xs:attributeGroup name="ReferenceAttributes">
    <xs:attribute name="label" type="xs:string"
                                              use="required"/>
    <xs:attribute name="author" type="xs:string"
                                              use="required"/>
    <xs:attribute name="title" type="xs:string"
                                              use="required"/>
    <xs:attribute name="year" type="xs:string"
                                              use="required"/>
  </xs:attributeGroup>
</xs:schema>
```

A Figura 4.4 mostra o diagrama completo de classes para este exemplo.

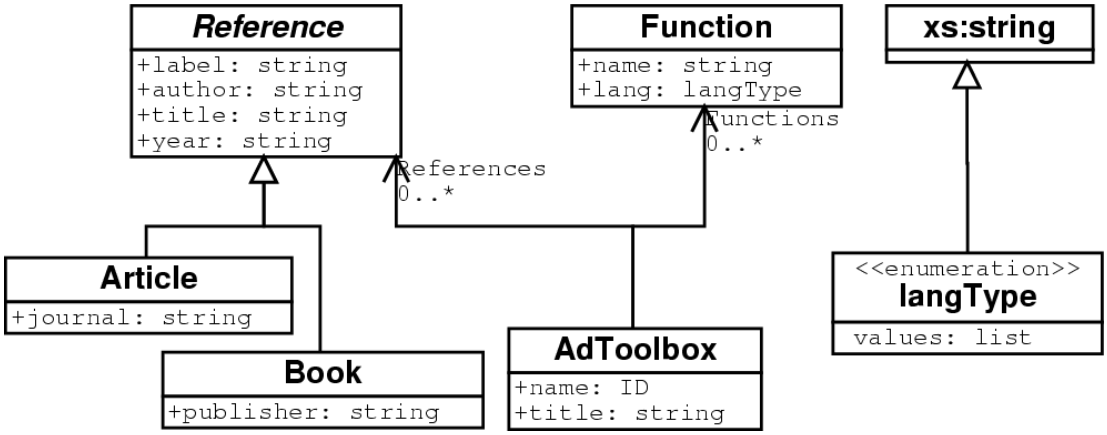

Figura 4.4: Modelo Completo de um XML Schema Simples.

Uma referência, neste exemplo, pode ser um artigo ou um livro. *Article* e *Book* possuem um conjunto de atributos em comum, definidos no grupo de atributos chamado *ReferenceAttributes*. Ambos são subclasses de *Reference* e, por conseguinte, herdam todos os seus atributos. Logo, os atributos comuns entre *Article* e *Book*, são, na verdade atributos de qualquer referência e podem ser definidos na superclasse abstrata *Reference*.

Note que *Reference* não está definido no esquema. Cuidados especiais com a nomenclatura são necessários durante a geração automática de código. Agora já é conhecida a associação direta entre *AdToolbox* e *Reference*, com multiplicidade "0..*".

4.2.2 Modelos de Esquemas Complexos

Muitos dos recursos da linguagem XML Schema não aparecem em uma esquema simples. No exemplo acima, uma boa maneira de mostrar isso é enfatizando o modelo de agrupamento do tipo complexo *ReferencesType*. Os elementos do *choice*, *Article* e *Book*, estão coincidentemente relacionados um ao outro. O modelo *choice* não obriga isso.

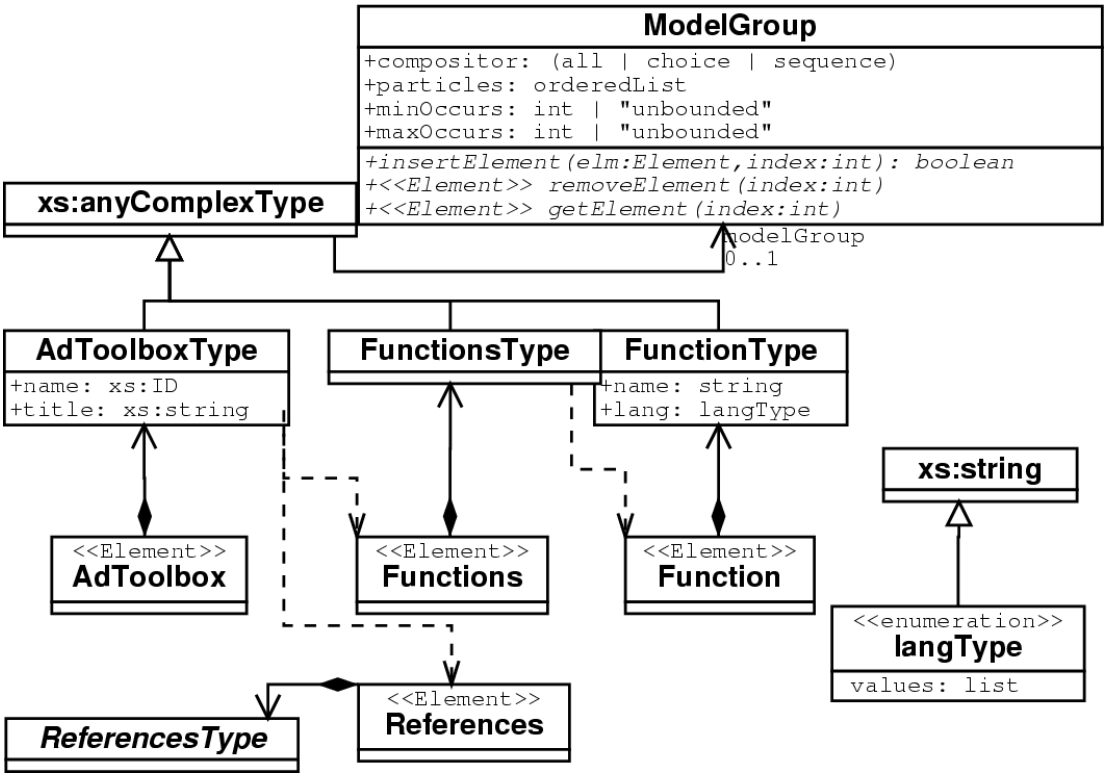

Figura 4.5: Um diagrama de classes mais complexo para o mesmo XML Schema.

Obviamente, mesmo se *Article* e *Book* fossem completamente diferentes, seria ainda correto definir ambos como subclasse de *Reference*, já que ambos são referências. Porém, os tipos desses dois elementos poderiam ter sido definidos como tipos complexos completamente distintos. Além disso, estão definidos globalmente e podem, portanto, estar relacionados a outros elementos quaisquer por algum mecanismo de herança. Todas essas possibilidades levariam a um problema muito conhecido da orientação a objetos: a **herança múltipla**.

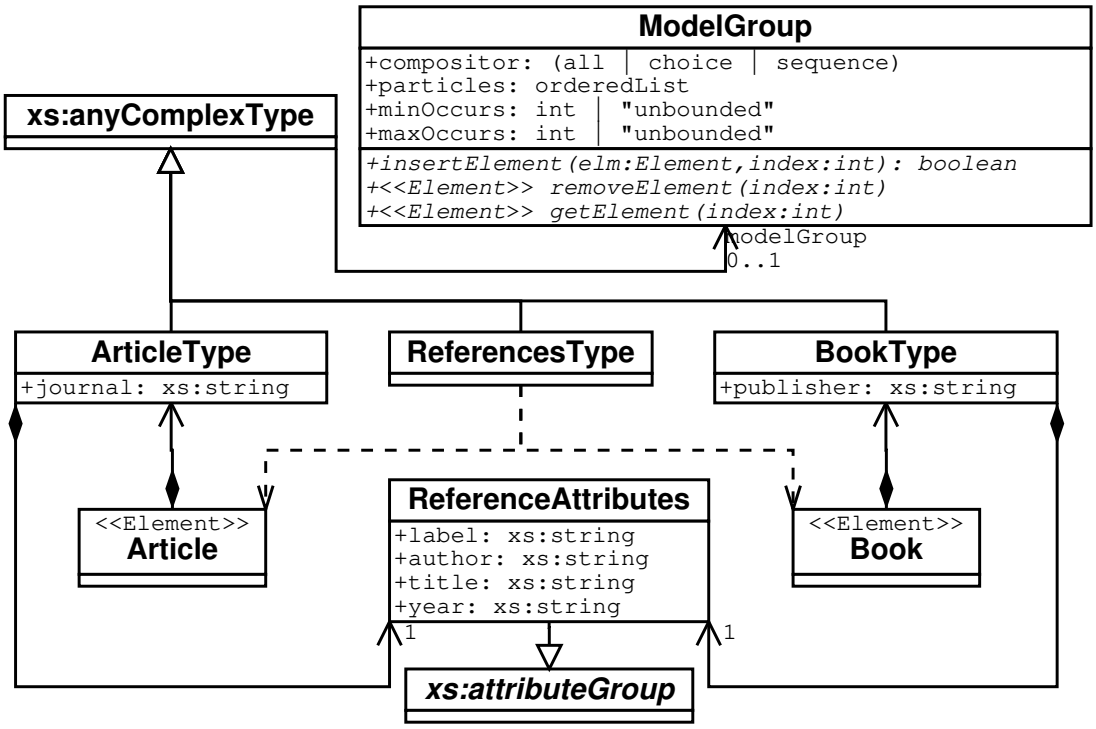

Figura 4.6: Continuação do diagrama de classes mais complexo.

A linguagem Java oferece uma maneira de simular a herança múltipla através do uso de **interfaces**, mas isso não é comum a outras linguagens como, por exemplo, o C++. Essa forma de simulação da herança múltipla também não seria suficiente para resolver o problema do *choice*, pois o problema com relação aos modelos de agrupamento é mais abrangente: como descrever em UML a ordem de disposição de elementos?

Carlson [17] responde a esta questão propondo o uso de mais de uma construção UML para cada estereótipo, o que seria especificado através de uma extensão, por ele definida, da linguagem. Essa extensão é usada na ferramenta *hyper*Model.

Essas considerações são importantes para salientar a dificuldade de modelagem do XML Schema. O objetivo também não é refazer o trabalho de Carlson, mas sim apresentar uma proposta diferente de modelagem de um XML Schema, que seja a mais genérica possível e de simples implementação em folhas de estilo. Uma implementação simples de folha de estilo é aquela que trata cada nó com o menor número de condições. Por exemplo, quando um nó *element* de um esquema tiver sempre o mesmo *template*, não importando se é filho de um *choice*, de um *sequence* ou de um *all*.

A idéia é dividir ainda mais as classes geradas. Se voltarmos ao primeiro fragmento de esquema apresentado na Seção 4.2.1, que define o elemento *AdToolbox* e o tipo complexo *FunctionType*, obteríamos o diagrama de classes apresentado na Figura 4.5.

Fica bem claro que este modelo é mais complexo do que o apresentado anteriormente na Figura 4.3. Porém, é possível observar uma relação clara e direta deste modelo com o fragmento de esquema por ele representado. O elemento raiz *AdToolbox* é composto por um tipo complexo chamado *AdToolboxType*. Este modelo considera sempre que um tipo complexo contém apenas os atributos normalmente declarados e uma associação com um modelo de agrupamento. Todos os tipos complexos declarados no esquema, seja de forma implícita (como *AdToolboxType* e *FunctionsType*), seja explícita (como *FunctionType*) são subclasses de *xs:anyComplexType*, um tipo predefinido do XML Schema. O modelo de agrupamento em si passa a ser representado por uma classe à parte (*ModelGroup*). Possui dois atributos: *compositor*, que define o tipo de agrupamento, e *particles*, uma lista ordenada dos elementos permitidos neste agrupamento. Perceba que *ModelGroup* é uma classe genérica e seu conteúdo precisa ser inicializado pelo tipo complexo que o contém. *AdToolboxType* precisa de um elemento *Functions* e um elemento *References* para iniciar o seu modelo de agrupamento. Daí a relação de dependência entre essas classes. O mesmo acontece com *FunctionsType* e *Function*.

Nesta mesma linha de raciocínio, define-se o diagrama de classes para o segundo frag-

mento de esquema apresentado na Seção 4.2.1. O modelo que define o tipo *ReferenceType* é mostrado na Figura 4.6.

Essas regras de definição do modelo garantem a simplicidade esperada das folhas de estilo. As mesmas regras podem também ser utilizadas para a geração de componentes gráficos, como mostra a Figura 4.7. Esta figura corresponde apenas ao primeiro fragmento de esquema. Não contém detalhes sobre os componentes gráficos das referências.

Em suma, a Figura 4.7 mostra que o elemento *AdToolbox*, definido no XML Schema, contém a definição de seu tipo (*AdToolboxType*). Esta definição de tipo, por sua vez,

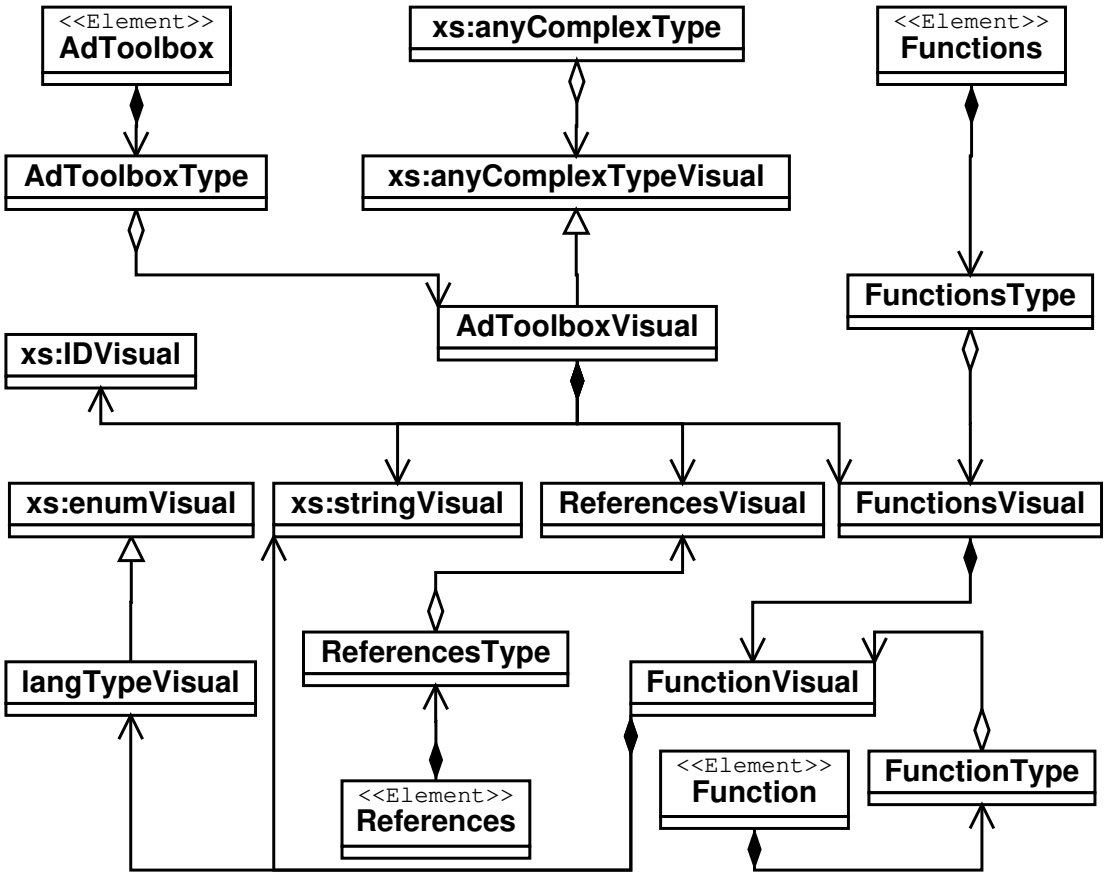

Figura 4.7: Diagrama de classes para os componentes gráficos.

possui o componente gráfico *AdToolboxVisual* agregado. *AdToolboxVisual* é um componente gráfico complexo e, portanto, contém outros componentes gráficos. Isso vale para todos os elementos declarados no XML Schema. O componente gráfico que representa um tipo simples é atômico, ou seja, não contém outros componentes gráficos. É o caso de *xs:string*, *langTypeVisual* e outros.

Vale discutir um detalhe sobre a classe *langTypeVisual*. Não obstante no esquema ela esteja definida como uma restrição de *xs:string*, na figura é colocada como uma extensão de *xs:enumVisual*. Isso porque o componente gráfico utilizado para a edição de uma cadeia de caracteres sem restrições (tipo *xs:string*) é nada mais que um campo de texto simples, enquanto que o componente gráfico mais adequado para um **tipo enumerado** seria uma caixa de rolagem (*comboBox*) ou uma caixa de opções. Esses dois últimos nada têm de semelhança com o primeiro. Por esta razão, *langTypeVisual* não estende *xs:stringVisual*, mas sim *xs:enumVisual*, que seria o componente gráfico predefinido apto para tipos enumerados.

Como pode-se observar, tanto no modelo de estruturas de dados, quanto no modelo para componentes gráficos, é feito o uso de tipos predefinidos do XML Schema (identificados com o prefixo *xs*). Isso sugere que esses tipos devem ser previamente implementados. Detalhes sobre a geração automática de classes serão discutidos no próximo capítulo.

Capítulo 5

O Gerador de Editores XML

Com o objetivo final de modelar o Gerador Automático de Editores XML, foi seguida uma linha de raciocínio partindo do modelo de um editor XML genérico.

5.1 Modelo de um Editor XML Genérico

O modelo básico de um **Editor XML Genérico** é apresentado na Figura 5.1.

Este editor, ao abrir um documento XML, faz sua validação com relação a um DTD ou XML Schema. Se o documento lido não estiver de acordo com o esquema, uma mensagem de erro é gerada e a edição interrompida. Caso contrário, se o documento for válido com relação ao esquema, o editor mostra-o de forma semelhante à da figura. Os *frames* numerados da figura seriam, basicamente:

1. Uma árvore, semelhante a uma árvore de diretórios, que representa a hierarquia dos elementos contidos no documento XML. Uma visualização gráfica da árvore DOM.

2. Os detalhes (atributos e conteúdo) do elemento selecionado na árvore do primeiro *frame*.

3. O código XML da subárvore abaixo do elemento selecionado.

4. Mensagens diversas.

Neste modelo, é possível perceber que um editor genérico só usa o XML Schema no momento da validação do documento XML a ser editado. Quando o editor mostra o documento, ele não possui outra informação além da árvore DOM. Essa informação é insuficiente para definir uma apresentação e, por conseguinte, um editor genérico acaba por mostrar todos os elementos e atributos do documento de forma idêntica: o par nome/valor, como mostra a Figura 5.2. Na Figura 5.2 é possível observar que o atributo *institute* do elemento *author* sendo editado, é uma referência ao atributo *name* do elemento *institute*, logo abaixo. Os valores são sempre representados por cadeias de caracteres (*strings*), como são naturalmente representados no código em XML.

5.2 Evoluções do Modelo

A apresentação genérica dos valores em forma de cadeia de caracteres certamente não é a melhor das visualizações possíveis além de ser pouco satisfatória do ponto de vista

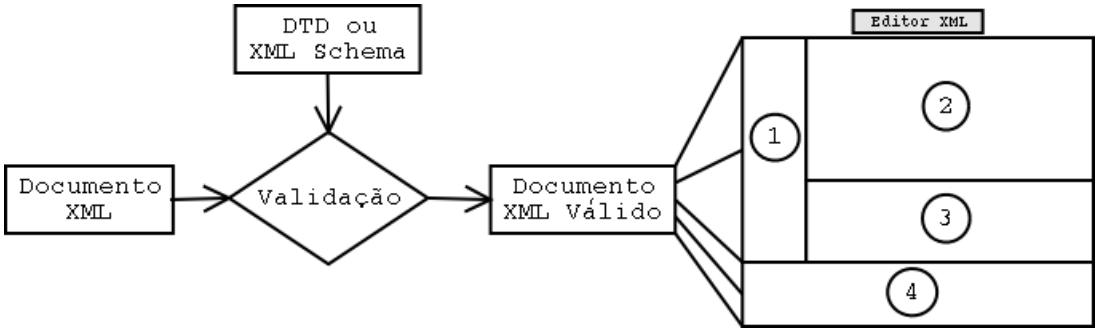

Figura 5.1: Modelo de um Editor XML Genérico. Um editor como esses mostra qualquer documento XML utilizando um formato padronizado de apresentação: o par nome/valor, sempre representados por cadeias de caracteres, que é o formato natural da linguagem XML.

prático, como já foi discutido na Seção 3.2. O ideal é que a apresentação dos detalhes de um elemento (*frame* 2 da Figura 5.1) seja feita utilizando informações do contexto ao qual o documento XML editado está associado. Esse contexto pode ser identificado no XML Schema, já que este define todo o modelo de dados, a forma e os tipos permitidos em cada documento XML válido.

Na Figura 5.3 foi introduzido um bloco denominado **Máquina de Contexto**.

A Figura 5.3 é quase idêntica à Figura 5.1, com exceção deste novo bloco. A Máquina de Contexto é, neste novo modelo, o bloco que "conhece" o esquema e, portanto tem

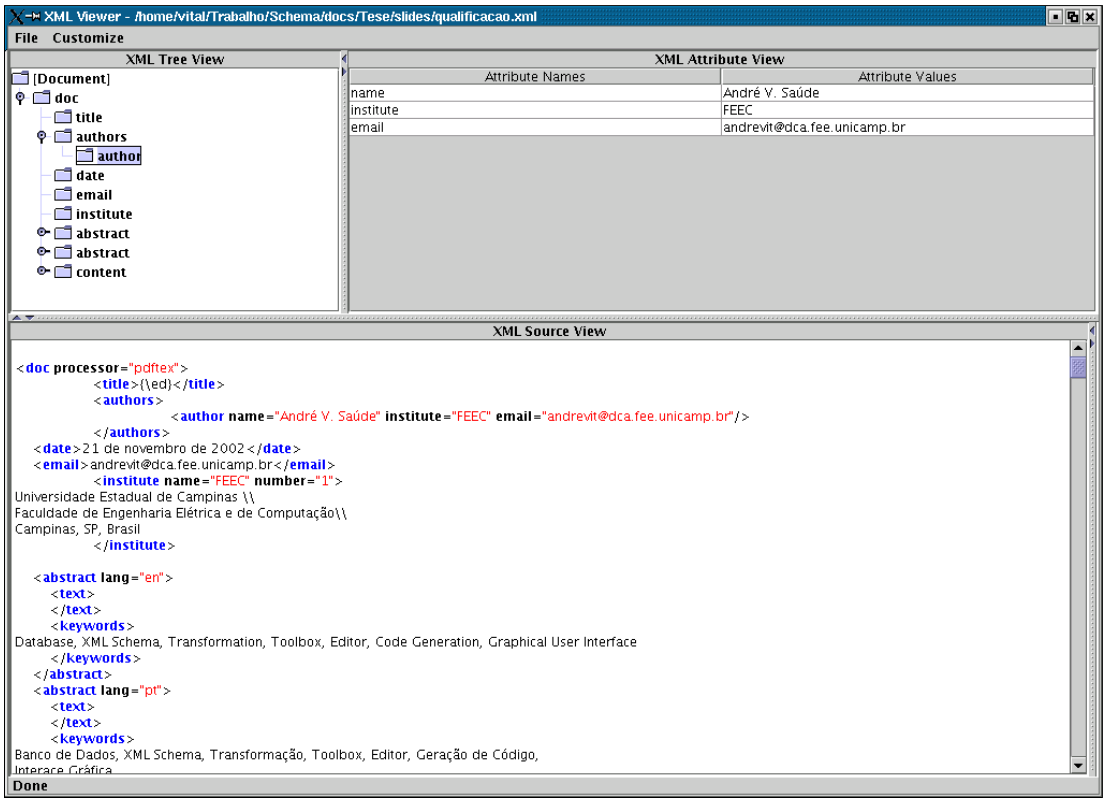

Figura 5.2: Exemplo de um Editor XML Genérico. O atributo "institute" do elemento "author", sendo editado, é uma referência ao atributo "name" do elemento "institute", logo abaixo. Este editor não faz a associação entre chaves e referências.

informações suficientes para mostrar os detalhes de um elemento com uma apresentação mais amigável. Perceba que neste novo modelo, a interface gráfica (**GUI — Graphical User Interface**) não é toda construída antecipadamente como acontecia no modelo do editor genérico. Na Figura 5.3, as setas que apontam para os *frames* da GUI indicam que o conteúdo de cada um desses *frames* é fornecido em tempo de execução pela Máquina de Contexto, de onde as setas se originam. A seleção de um elemento na árvore do *Frame* 1 faz com que a GUI requisite à Máquina de Contexto um componente visualizável para ser exibido no *Frame* 2. A GUI recebe este componente e simplesmente mostra-o. Qualquer modificação no valor deste componente gráfico precisa respeitar as regras definidas no esquema e, portanto, deve ser avisada à Máquina de Contexto. Esta passa agora a fazer validações pontuais do documento editado com relação ao esquema. Isso também vale para a inserção ou a remoção de um elemento da árvore. Logo, a árvore também deve ser monitorada pela Máquina de Contexto. Ora, se é também a Máquina de Contexto que monitora a árvore DOM e gera possíveis mensagens de erro, torna-se óbvio atribuir-lhe a responsabilidade para com os *frames* 3 e 4. Assim, a interface gráfica deste editor passa a ser muito simples e de fácil implementação, já que o maior trabalho foi delegado à Máquina de Contexto.

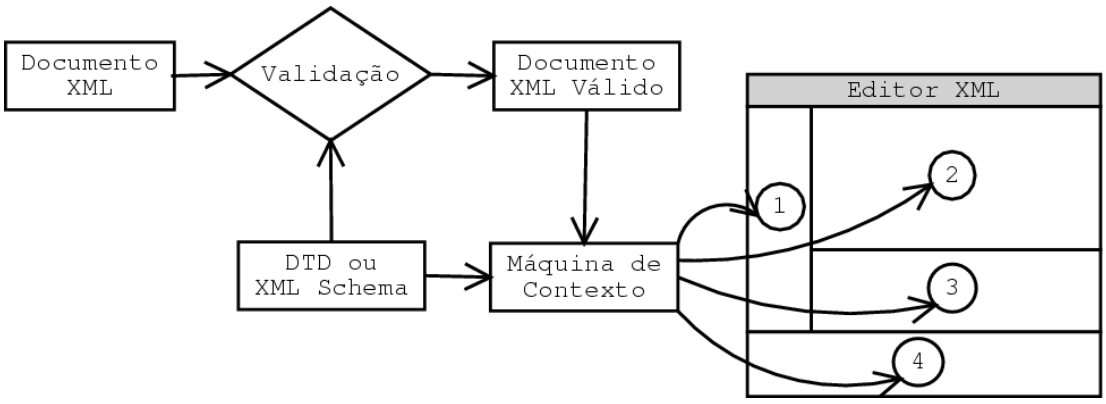

Figura 5.3: Um editor XML que observa o contexto da aplicação. Este editor não mais mostra qualquer elemento ou atributo utilizando um formato padronizado de apresentação, mas sim de acordo com o contexto da aplicação, ou seja, baseado nas informações obtidas no XML Schema que o descreve.

5.3 Modelo do Gerador de Editores XML

Levando em consideração a inserção da Máquina de Contexto no modelo, é possível fazer uma divisão mais detalhada dos papéis de cada parte do editor, gerando o modelo final do Gerador de Editores XML, mostrado na Figura 5.4.

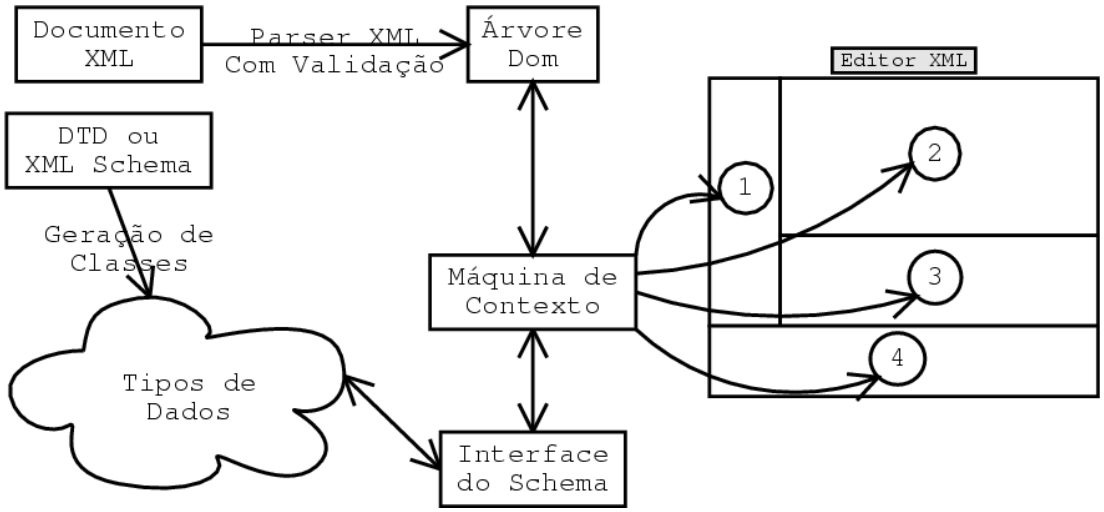

Figura 5.4: Modelo do Gerador de Editores XML. O Processador de Estilos do Adesso lê o XML Schema e gera classes com componentes gráficos que melhor representam os tipos de dados ali definidos. Para cada XML Schema, uma interface gráfica diferente é gerada.

Nesta abordagem, a GUI não tem ação sobre a edição e nada mais faz do que mostrar os componentes importantes do editor. Isso simplifica bastante o trabalho exigido para sua implementação. A interface gráfica passa a ser simplesmente uma janela com quatro *frames* cujos conteúdos são fornecidos pela Máquina de Contexto. A Máquina de Contexto torna-se responsável pela leitura do documento XML e pela geração/monitoração da árvore DOM. Diferentemente do que havia sido mostrado na Figura 5.3, o responsável pelas estruturas de dados e formas de visualização é a nova interface Schema e não a Máquina de Contexto. Essa maneira de separar os papéis visa flexibilizar o ambiente de edição, permitindo inclusive aplicações cliente/servidor sem implicar reprogramação.

Isso será discutido no Capítulo 6.

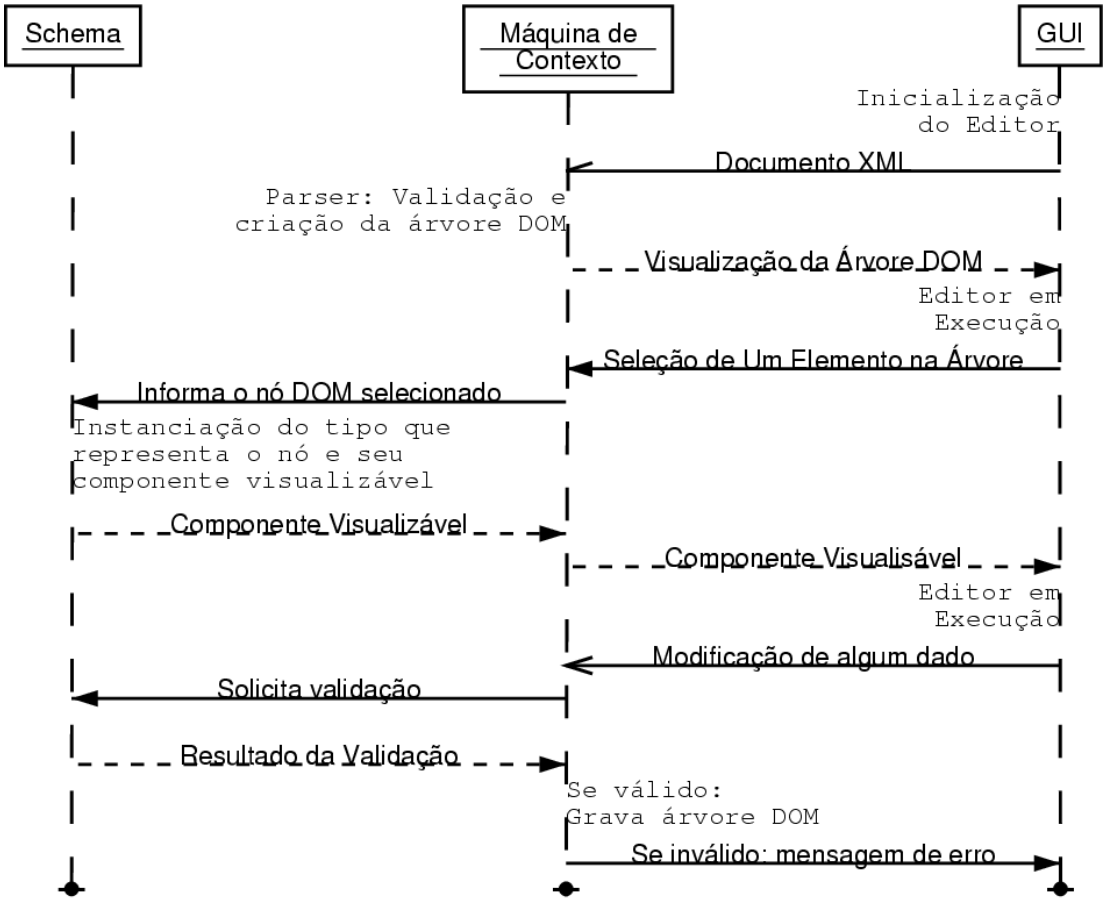

Figura 5.5: A execução de um editor gerado automaticamente. Diagrama de seqüência que ilustra a divisão dos papéis durante a execução de um editor gerado automaticamente pelo Gerador de Editores XML. Veja explicação detalhada no texto.

O diagrama de seqüência na Figura 5.5 ajuda no entendimento da função de cada bloco do modelo. O usuário inicia o editor pela GUI. A GUI passa para a Máquina de Contexto a localização do documento XML a ser editado. A Máquina de Contexto abre o documento, faz o *parsing* e a validação, gerando a árvore DOM, e retorna para a GUI uma visualização desta árvore. Durante a execução do editor, o usuário seleciona elementos nesta árvore. Isso gera um evento à Máquina de Contexto que, por sua vez, informa à

interface Schema o nó DOM selecionado. A interface Schema é responsável por fazer a instanciação da classe que representa o nó DOM selecionado e do respectivo componente gráfico. A GUI recebe esse componente e simplesmente mostra-o. Qualquer modificação de dado é feita no componente gráfico e gera um novo evento à Máquina de Contexto. Esta solicita ao Schema a validação do novo valor. Se a modificação feita for válida, a Máquina de Contexto modifica o nó DOM e grava a árvore. Caso contrário, retorna uma mensagem de erro para a GUI.

O modelo do Gerador de Editores XML discutido continua independente da linguagem de programação. Na próxima seção será discutida a geração automática da nuvem intitulada "Tipos de Dados", na Figura 5.4. Na geração dos tipos de dados chega o momento em que a escolha da linguagem torna-se necessária.

5.4 Geração Automática do Editor

Nesta seção, é apresentado o mecanismo de geração automática da nuvem intitulada **Tipos de Dados** na Figura 5.4. McLaughlin[19] apresenta uma proposta de transformação dos tipos do XML Schema em classes Java. A idéia é que seja gerada uma classe para cada tipo definido no XML Schema, com métodos que modifiquem seu conteúdo ou seus atributos. Qualquer tipo definido em um esquema é derivado dos tipos primitivos da especificação[8]. Por isso, qualquer classe gerada automaticamente deve estender as classes que representam os tipos primitivos e, logo, é necessário que haja classes que implementem os tipos primitivos.

Kawaguchi, em **Sun XML Datatypes Library**[20], implementa os tipos primitivos do Schema e sua biblioteca pode ser usada como ponto de partida. É importante lembrar, porém, que além de representar a estrutura do tipo definido, cada classe gerada deve possuir um componente gráfico a ela relacionado, inclusive as classes que representam os tipos primitivos. Assim, não foi possível usar diretamente a biblioteca de Kawaguchi[20]. As classes que representam os tipos primitivos tiveram de ser reimplementadas para a inclusão deste requisito, como será discutido no Capítulo 6. A proposta

de McLaughlin[19] usa a própria linguagem Java para a geração do código das classes. A implementação dessa proposta pode ser considerada complexa, se comparada à simplicidade da geração de código via transformações XML e folhas de estilo.

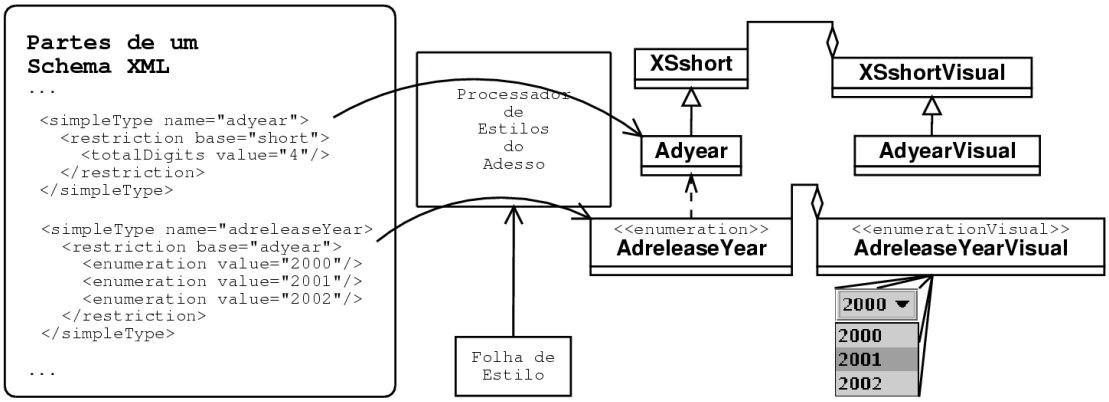

Figura 5.6: Geração de um tipo simples. Uma classe que representa um *simpleType* do XML Schema.

A Figura 5.6 ilustra a **geração de um tipo simples**, ou seja, de uma classe que representa um *simpleType* do XML Schema. Esta figura mostra o uso do conceito apresentado no Capítulo 2 e na Figura 2.2. O *simpleType* de nome *adyear* é um descendente direto de um tipo primitivo do XML Schema: o tipo *short*. A classe gerada para representar um *adyear* estende a classe que representa o tipo primitivo *short*. Da mesma forma, o tipo simples de nome *adreleaseYear* é um descendente de *adyear*. Porém, o tipo de restrição que define *adreleaseYear* faz distinção entre este e seu tipo base. Isto porque um tipo enumerado só depende do seu tipo base para verificar se os valores da enumeração são válidos, ou seja, no momento de sua construção. Todavia, mesmo sendo possível fazer uma validação prévia do esquema e garantir que os valores enumerados da classe *adreleaseYear* são corretos, essa dependência deve ser mantida. Isto porque *adreleaseYear* não é uma classe final e, portanto, pode ser estendida. Esta classe deve ser capaz de validar valores com *adyear*. Uma hierarquia similar ocorre com os componentes gráficos desses tipos. A diferença é que não existe a dependência entre *adreleaseYearVisual* e *adyearVisual*. Um componente gráfico que representa um tipo enumerado se difere

muito daquele que representa um inteiro qualquer. Perceba na Figura 5.6 que a classe *XSshort*, que representa o tipo primitivo *short*, não está sendo gerada. Esta classe e *XSshortVisual* devem ser implementadas manualmente, como será discutido na Seção 6.3.

O componente gráfico exemplificado para *adreleaseYearVisual*, que representa o *adreleaseYear* é, provavelmente, o que seria realmente usado em um editor especializado para mostrar um tipo enumerado. O usuário do editor, além de ter a facilidade de modificar o valor com apenas dois cliques, é impossibilitado de selecionar um valor inválido.

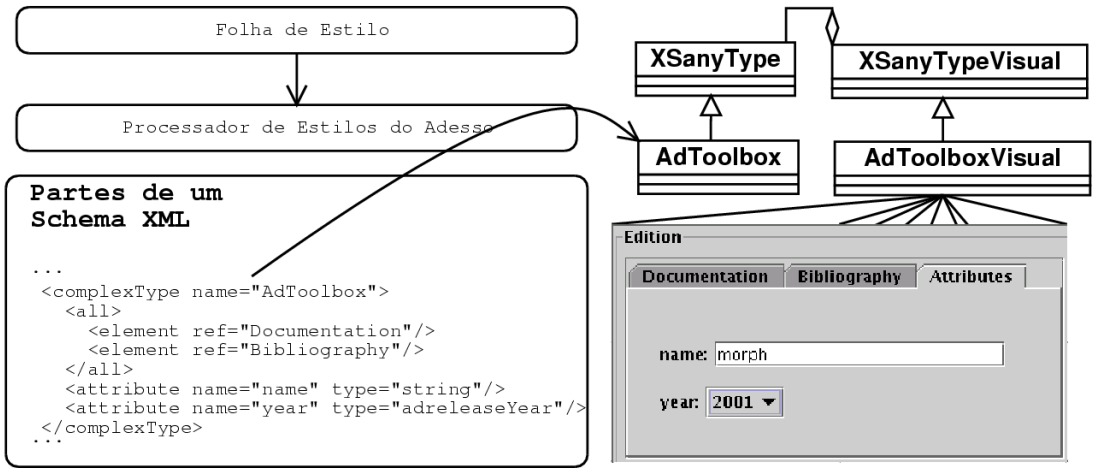

Figura 5.7: Geração de um tipo complexo. Uma classe que representa um *complexType* do XML Schema.

Um tipo simples é o único que tem uma forma de visualização bem definida. Os tipos complexos são conjuntos de outros tipos complexos ou simples e seus componentes gráficos são do tipo *container*, ou seja, podem conter outros componentes gráficos. A Figura 5.7 ilustra a **geração de um tipo complexo**. Todo tipo complexo, pela norma XML Schema, estende o tipo primitivo *anyType*. Note que a classe *AdToolbox* estende *XSanyType*. Os detalhes dos elementos *Documentation* e *Bibliography* foram omitidos neste exemplo por razões de espaço. O componente gráfico gerado para o tipo complexo *AdToolbox* está sendo exibido na Figura 5.7. Há diversas disposições possíveis dos

componentes dentro de um tipo complexo. Nenhuma pode ser considerada melhor ou pior do que a outra. Trata-se muito mais de um gosto pessoal do usuário. Um bom trabalho previsto para o futuro é analisar diversas maneiras de dispor os componentes ou de mostrar um determinados tipo para disponibilizar ao usuário um conjunto de opções de *layout*. É possível, por exemplo, que o usuário tenha definido em seu XML Schema um tipo de dados que representa uma árvore binária. Com o XML Schema não há como deixar clara esta opção do usuário. Porém, depois do editor gerado, o usuário poderia associar manualmente o seu tipo à visualização de árvores binárias. Desta forma, o usuário poderia fazer configurações pessoais no editor gerado e obter a interface gráfica que mais lhe agrada. Isso tornaria o Gerador de Editores XML ainda mais abrangente e flexível.

As folhas de estilo que descrevem as transformações do XML Schema em classes são dependentes da linguagem de implementação dessas classes. Uma folha de estilo que gera classes em Java é diferente de uma que gera classes em C++. A implementação das folhas de estilo também deve ser em módulos. Assim, se for de interesse que qualquer das partes do editor gerado esteja implementada em outra linguagem, os pontos onde as folhas de estilo deverão ser alteradas são fáceis de ser encontrados. Isto se aplica principalmente aos trechos de código que geram os componentes gráficos.

5.5 Ilustração do Editor

Finalmente, tendo descrito todo o processo de geração de interfaces gráficas com Gerador de Editores XML, a Figura 5.8 mostra a janela de um editor gerado para o sistema Adesso. Esta forma apresentada é um aplicativo em Java cujo *layout* é uma opção pre-configurada do grupo do projeto Adesso. Não só o *layout* pode ser modificado mas também a linguagem de programação usada na interface gráfica. No Capítulo 7 será apresentada uma outra forma de apresentação que pode se tornar muito útil: utilizando acesso remoto.

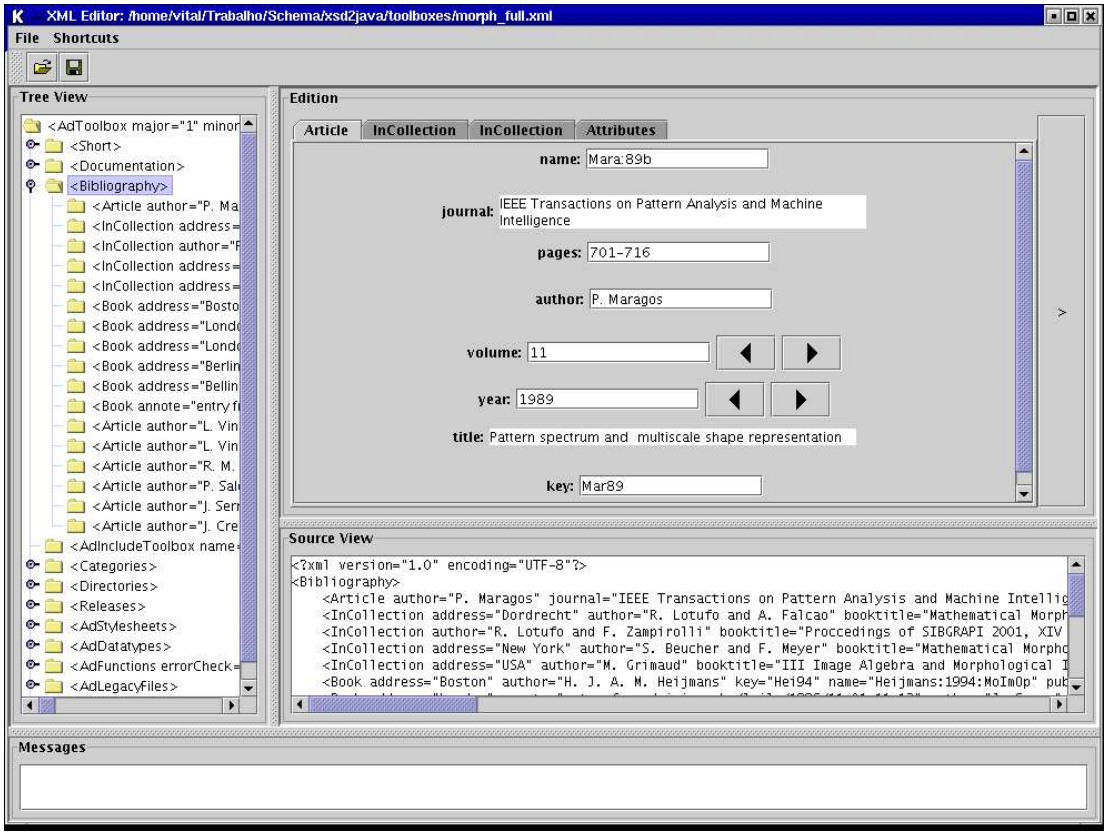

Figura 5.8: Imagem da janela de um editor gerado automaticamente. O conteúdo dos *frames* é instanciado e monitorado pela Máquina de Contexto a partir de classes geradas automaticamente pelo Gerador de Editores XML.

Capítulo 6

Aplicações do Gerador de Editores XML

Como o XML é um padrão largamente utilizado para representação da informação, o Gerador de Editores XML pode tornar-se útil para muitas outras aplicações. Qualquer aplicação baseada em XML que se utiliza do XML Schema pode usufruir desta ferramenta. Um bom exemplo de aplicação nesses moldes é o XML Résumé Library, apresentado na Seção 6.1. Algumas aplicações também podem utilizar apenas as classes geradas automaticamente para efetuar validações de modificações feitas sem o uso da interface gráfica, como é discutido na Seção 6.2. Além disso, parte do Gerador de Editores XML são pacotes que implementam requisitos da especificação do XML Schema e podem ser úteis se usados de forma independente por outros desenvolvedores. Estes pacotes e outros produtos da implementação do Gerador de Editores XML são apresentados na Seção 6.3.

6.1 O XML Résumé

Um exemplo de aplicação baseada em XML que se utiliza do XML Schema é o **XML Résumé Library** [21], um sistema baseado em XML e XSLT para marcação e for-

matação de *curricula vitae*, disponível na Internet. Neste sistema, um usuário escreve seu **curriculum vitae** manualmente em um documento XML fonte utilizando um editor qualquer e executa o XML Résumé. O aplicativo faz as transformações necessárias e devolve ao usuário o *curriculum vitae* em formatos HTML, PDF ou texto não formatado. Este sistema usa as ferramentas gratuitas do Apache XML Project[1] em Java, as mesmas usadas pelo Gerador de Editores XML. Obviamente, para que as transformações sejam feitas corretamente, o documento fonte escrito pelo usuário deve seguir determinadas regras predefinidas, ou seja, deve estar correto perante um **DTD**. Do usuário são exigidos, portanto, dois requisitos prévios: um pequeno conhecimento do padrão XML e o conhecimento das normas de utilização do sistema. O primeiro é pouco complexo, já que o XML é um formato de fácil compreensão. O segundo é mais demorado de se adquirir porque exige a compreensão de um manual de usuário ou diretamente do DTD.

Se existisse um editor próprio para *curriculum vitae*, o usuário desse sistema não precisaria de pré-requisitos para começar a editar seu currículo. Para isso, o Gerador de Editores XML pode ser usado. Se o DTD for transformado para o formato XML Schema (existem ferramentas gratuitas que fazem isso automaticamente[2]), o Gerador de Editores XML pode gerar automaticamente um editor de currículos para o XML Résumé, como o editor mostrado na Figura 6.1. Na figura, o elemento editado é o nome do usuário. Clicando com o botão direito, sabe-se, sem necessidade de consultar um manual, quais são os elementos ou atributos permitidos naquele ponto de edição. O menu mostra que o único atributo permitido para o elemento "name" é o atributo "id". Assim seria para todos os elementos, ao mesmo tempo guiando o usuário e impedindo-o de gerar documentos inválidos.

A proposta é usar o Gerador de Editores XML para gerar e compilar um editor de currículos. Este editor compilado seria distribuído juntamente com o XML Résumé.

Esta proposta aponta possíveis expansões do Gerador de Editores XML. Na situação atual do gerador, o usuário do XML Résumé só poderia utilizar o editor compilado

[1] http://xml.apache.org/

[2] Algumas destas ferramentas podem ser encontradas em http://www.w3.org/XML/Schema

para escrever e salvar seu *curriculum*. O XML Résumé deveria ser executado separadamente, pela linha de comando, para fazer as transformações e gerar os formatos de apresentação (PDF, HTML, etc.). A distinção entre o editor e o aplicativo que faz as transformações é inconveniente, já que o objetivo da adição de um editor com interface gráfica amigável é exatemente atingir um nicho de usuários mais amplo, incluindo muitos usuários leigos. Além do trabalho que o usuário teria de gravar o arquivo e depois executar outro aplicativo pela linha de comando, existe ainda um problema com respeito às atualizações e lançamentos de versões do sistema. Qualquer mudança no XML

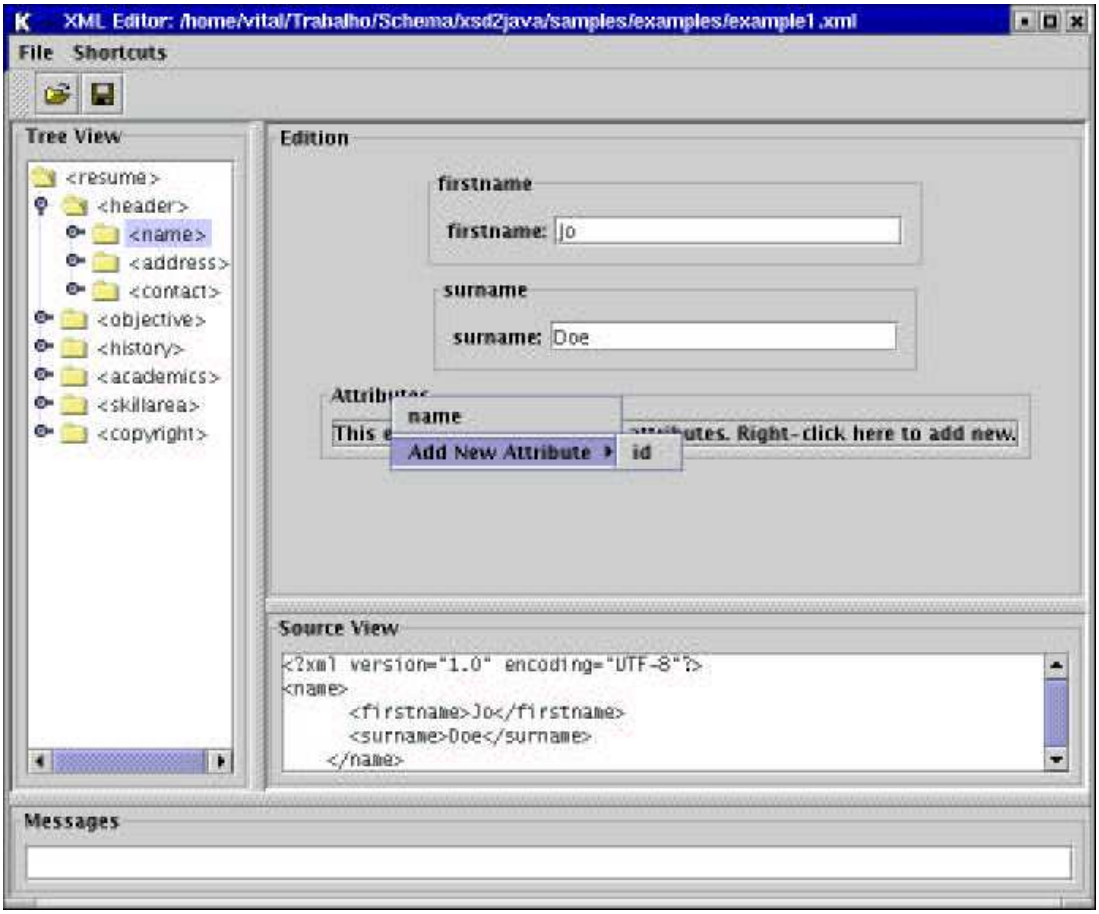

Figura 6.1: Um editor gerado a partir do DTD do XML Résumé Library.

Schema gera mudanças tanto no editor de *curricula vitae*, quanto nas folhas de estilo utilizadas pelo XML Résumé nas transformações. É devido às freqüentes mudanças nas folhas de estilo que o sistema XML Résumé, atuando de forma padrão, as obtém diretamente de seu servidor na Internet e não da máquina local. O XML Résumé atualmente funciona fazendo requisições pela Internet. A possibilidade de adicionar ao editor gerado menus que façam chamadas a executáveis externos é uma boa proposta de expansão do Gerador de Editores XML para contornar esse problema rapidamente. Atualmente, esta expansão pode ser feita através da desaconselhada edição manual do código gerado. Certamente, o melhor caminho é criar uma interface de definição de programas externos, denominada *External*, e criar um diretório específico no sistema de arquivos para armazenar as classes que implementam esta interface. Esta idéia é semelhante à utilizada pelos navegadores web para seus *plugins*. A interface *External* teria apenas o método *getAppName*, para obter o nome do aplicativo externo, e o método *call*, que recebe como parâmetro o documento XML sendo editado, para fazer a chamada do programa externo. Assim, ao iniciar sua execução, o editor buscaria nesse diretório todas as classes que porventura implementassem a interface *External* e faria, para cada uma delas, a ligação entre um novo menu e o método *call*.

Se uma chamada ao XML Résumé for adicionada à barra de menus do Gerador de Editores XML, o usuário terminaria a edição de seu documento e solicitaria a geração dos novos formatos sem sair do editor. Assim, o editor gerado automaticamente Gerador de Editores XML poderia ser distribuído já com o XML Résumé incorporado, deixando transparente a etapa de transformações. Veja que a chamada ao aplicativo externo é intermediada por uma classe que implementa a interface *External*. O método *call*, desta classe, pode ser extenso o suficiente para gerenciar condições ou até mesmo apresentar uma caixa de diálogo com opções, tais como o tipo de arquivo a ser gerado (PDF, HTML, etc.) ou o nome do programa que o usuário deseja utilizar para visualizar o formato gerado.

Voltemos agora ao problema de atualização de versões. Se apenas a apresentação em PDF ou HTML for mudada, as atualizações do sistema continuariam como são hoje, ou seja, o usuário iria manter o mesmo editor e atualizaria apenas o XML Résumé. Caso

haja mudança também no DTD, a equipe do XML Résumé pode gerar um novo editor e redistribuí-lo.

Existem diversas aplicações nas quais um *curriculum vitae* é usado. Na comunidade científica, por exemplo, é muito importante um cadastro de pesquisadores contendo, além de seus dados pessoais, referências a suas publicações e dados sobre seu grupo de pesquisa. Essa idéia é defendida pelo Conselho Nacional de Desenvolvimento Científico e Tecnológico (CNPq), que disponibiliza o Sistema de Currículos **Lattes**[3]. Nesse sistema, o currículo de um pesquisador pode ser editado tanto pela página do CNPq quanto por um aplicativo específico, ambos através da Internet. Para o Gerador de Editores XML, o mais relevante é o formato como o Lattes armazena os dados dos currículos: utilizando a linguagem XML. Atualmente, ambos os editores de currículos Lattes, seja o da Web, seja o aplicativo local, são editores construídos especificamente para este fim. São editores especializados tais quais o apresentado na Seção 3.1. A grande desvantagem desse tipo de editor é o custo de atualizações no modelo de dados. O Gerador de Editores XML é, também para a plataforma Lattes, uma alternativa que permitiria um modelo de dados muito mais flexível, sem custo adicional de atualização dos ambientes de edição dos dados.

6.2 O Editor Usado Pela Máquina

Até o momento foram apresentadas apenas utilizações do Gerador de Editores XML por usuários humanos. De fato, a interface gráfica não tem outro objetivo senão este. O XML, no entanto, nem sempre é utilizado para armazenar dados que são editados manualmente. A linguagem XML pode ser usada para representar pequenas bases de dados atualizadas dinamicamente por um ou mais programas de computador. Além disso, uma mesma base de dados pode ter acesso compartilhado por inúmeros usuários, sejam eles humanos ou máquinas.

Sejam os acessos concorrentes ou não, o importante a observar com relação às bases de

[3] http://lattes.cnpq.br

dados é que, para qualquer tentativa de modificação de um determinado valor, é essencial que seja feita uma validação. No caso de dados armazenados no formato XML, a maneira mais fácil e simples de se implementar uma validação é fazer a modificação, armazená-la e, em seguida, utilizar uma ferramenta gratuita que faz validações de documentos XML. Sistemas pouco complexos podem utilizar-se desta estratégia, embora não seja uma boa solução. Não é uma boa solução porque as ferramentas mais comuns disponíveis para validação de documentos XML não são capazes de fazer validações de partes de um documento, mas apenas do documento completo. Esta limitação é muito grave quando se trata de um documento muito extenso. A validação de um documento grande pode ser mais demorada que a chegada de novas requisições de validação, o que causaria um congestionamento na fila de espera pelo serviço. Analisando agora os acessos concorrentes, se existem muitos processos concorrendo por uma validação, todos eles devem enfrentar a mesma fila, independente de qual ponto do documento foi modificado. Fica clara a importância de validações pontuais de partes dos documentos XML, sempre que possíveis. Esta discussão faz ressurgir a problemática sobre os editores genéricos e os editores especializados, discutida no Capítulo 3. A primeira estratégia de validação apresentada (validação do documento completo) é idêntica à dos editores XML genéricos. Todavia, sistemas com alguma complexidade devem certamente optar por validações pontuais[4]. Esses sistemas, portanto, passam por todos os processos de análise, projeto e desenvolvimento, durante os quais as fórmulas de validações pontuais são definidas. Exatamente como acontece com um editor especializado. O Gerador de Editores XML faz validações pontuais sem a necessidade de passar por todo o processo de desenvolvimento de *software*.

Se for gerado um editor sem a interface gráfica, ele pode ser usado por um sistema como esses. A Figura 6.2 ilustra o modelo de um editor automaticamente gerado para este novo ambiente.

Todos os programas que utilizarem o editor gerado sem interface gráfica devem sim-

[4]Existem casos em que não é possível fazer validações pontuais porque os aspectos de consistência dos dados podem envolver outras partes do documento. Estes casos porém são minoria na maioria das aplicações.

plesmente implementar a interface da Máquina de Contexto. O processo de geração automática do editor pode ser feita com restrições, para que este não possua código inutilizado. Nenhum dos componentes gráficos precisa ser gerado. No Capítulo 5 foi mostrada a necessidade de separar corretamente as partes das folhas de estilo que geram os componentes gráficos, das partes que geram os componentes em si, para facilitar a criação de componentes gráficos em outras linguagens. A possibilidade de gerar um editor sem interface gráfica é mais um motivo para se fazer esta distinção.

Talvez esta proposta não seja aplicável a sistemas de complexidade muito elevada, porque esses sistemas podem não possuir a facilidade de se modificar a definição da forma dos documentos XML simplesmente modificando o XML Schema a eles vinculado. Pa-

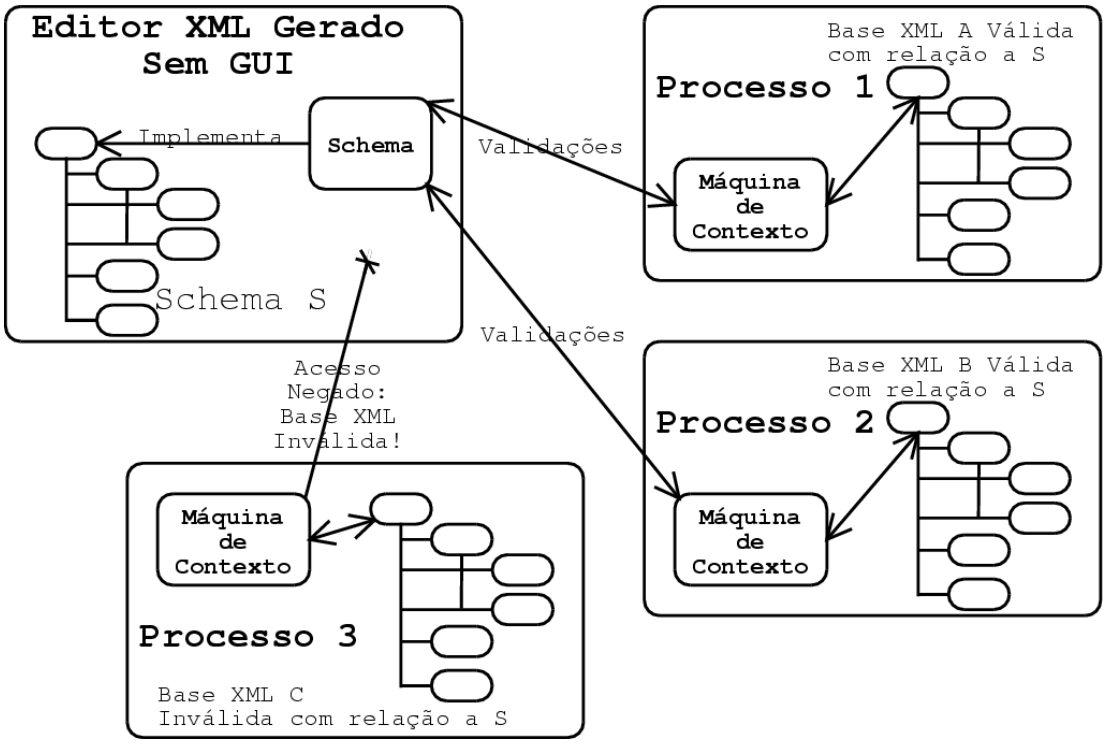

Figura 6.2: Um editor sem interface gráfica. Os processos utilizam somente o módulo Schema do editor. Apenas com o objetivo de fazer validações pontuais do documento XML com o qual trabalha.

ra que este editor seja útil, esta facilidade é primordial. É importante observar que no momento em que a interface gráfica é removida, a execução do editor ganha velocidade, pois não precisa instanciar componentes gráficos.

6.3 Produtos da Implementação

Apesar de muito recentes, as tecnologias XML já alavancaram a construção de diversas ferramentas a ela associadas. Versões atualizadas dessas ferramentas são lançadas com grande freqüência, juntamente com novas ferramentas, implementadas à medida que a necessidade aparece. O XML Schema vem sendo largamente utilizado na definição da estrutura de documentos XML e, conseqüentemente, na validação desses documentos. A informação de uma aplicação qualquer normalmente está armazenada na instância do documento XML, sendo portanto este o documento realmente processado pelas aplicações. Assim sendo, o programador deve desenvolver seu aplicativo baseado nas possíveis instâncias de documentos a serem processados. O número de aplicações que precisam processar o XML Schema, como é o caso do Gerador de Editores XML, é muito inferior. Por esta razão, ferramentas específicas para o processamento de esquemas são muito difíceis de serem encontradas ou simplesmente não estão disponíveis à comunidade, podendo até mesmo sequer existir. Algo meramente estranho se for considerado que a criação de tais ferramentas, uma vez que os marcadores do XML Schema são predefinidos, é a criação de ferramentas de uso globalizado. As ferramentas mais comuns já existentes são os editores, visualizadores e os validadores de esquema. Para a implementação do Gerador de Editores XML, essas ferramentas não foram suficientes. O Gerador de Editores XML precisa de ferramentas voltadas para o desenvolvimento, ou seja, implementações dos diversos conceitos presentes na norma XML Schema[6, 7, 8].

Esses pacotes de implementações dos conceitos do XML Schema vão sendo construídos à medida que os problemas surgem. Na composição do Gerador de Editores XML pacotes como esses foram necessários mas, infelizmente, nem todos foram encontrados. Isso obrigou a implementação de alguns conceitos e essas implementações podem

servir como contribuição para a comunidade da área em forma de pacotes compilados distribuídos gratuitamente. Os conceitos que precisaram de implementações específicas são apresentados a seguir:

- **Os Tipos Simples Predefinidos**
  Todos os tipos definidos em um Schema são derivações ou composições dos tipos básicos predefinidos. No Capítulo 2 são apresentados esses tipos. Kawaguchi os implementa em Sun XML Datatypes Library [20]. Sua biblioteca é suficiente para fazer todas as validações com relação ao formato do valor de um elemento ou atributo. A limitação desta biblioteca está na ausência de um tratamento de unicidades globais ou locais, representadas no XML Schema pelos *IDs*, *IDREFs*, *keys*, *keyrefs* e *uniques*. Esta biblioteca foi muito útil para o trabalho, facilitando a construção de um pacote que adiciona funcionalidades a ela, tal como a instanciação do componente gráfico de representação do tipo simples. Uma nova interface é definida para interação com os tipos simples, como mostramos a seguir:

```
package adesso.schema;

import javax.swing.*;
import com.sun.msv.datatype.xsd.*;

/**
 * A possible Adesso interface for generic simpleTypes.
 */
public interface XSanySimpleType extends XSanyType {
    /* Get and Set methods for the XSDatatype
       representation of this simpleType. XSDatatype is
       the interface used for validation, implemented by
       the Sun XML Datatypes Library. */
    public XSDatatype getXSDatatype();
    protected void setXSDatatype(XSDatatype xsdatatype);

    /* Get and Set value methods. Implementation of these
```

```
        methods use the Sun XML Datatypes Library */
    public String getValue();
    public boolean setValue(String newValue);

    // ... other get/set methods

    /**
     * Returns a Swing Component representing the content
     * of this simpleType
     */
    public JComponent getVisualizable();

    /**
     * Checks if the value of this simpleType can be set to
     * 'value'.
     */
    public boolean isValid(String value);
}
```

Como se pode ver, apesar da existência do método *getXSDatatype*, a biblioteca de validação de tipos é utilizada de forma transparente dentro dos métodos *isValid* e *setValue*. A necessidade de sobrescrever esses métodos está ligada ao controle de unicidades, ausente na biblioteca base. O controle de unicidade será detalhado mais adiante.

- **Derivações de Tipos Simples**
  Os tipos predefinidos do XML Schema podem ser primitivos ou derivados, como já foi mostrado na Figura 4.2. Qualquer novo tipo pode ser derivado de um tipo primitivo ou mesmo de outro tipo derivado. As derivações podem ser através da restrição de outro tipo simples, da união de dois tipos simples ou da criação de uma lista. A biblioteca de Kawaguchi[20] faz todas as derivações necessárias e, portanto, cobre a parte de validação de tipos simples, exceto pelos problemas de

unicidade já citados. Tal como para os tipos predefinidos, cria-se um pacote com as implementações de todos os tipos derivados, de acordo com a interface mostrada acima. O código das classes é gerado automaticamente e utiliza a biblioteca base para fazer derivações.

- **Estruturas e Modelos de Agrupamento**
  A segunda parte da especificação do XML Schema[7] define estruturas e modelos de agrupamento de elementos e atributos. Não foram encontradas implementações disponíveis dessas estruturas. Provavelmente, alguns validadores já as implementaram mas não em forma de um pacote a ser distribuído separadamente. A criação de objetos que representam um Schema torna indispensável a existência de classes que implementem essas estruturas. Logo, antes de mais nada, esta tarefa tornou-se parte do desenvolvimento do Gerador de Editores XML e pode vir a ser disponibilizado à comunidade em formato de um pacote não comercial.

- **Os Tipos Complexos**
  Os tipos complexos são composições de tipos simples ou de outros tipos complexos. Essas composições são representadas através das estruturas descritas na segunda parte da especificação [7]. Não foi encontrada implementação dos tipos complexos, já que ela depende de todas as estruturas do XML Schema e essas implementações também não foram encontradas. A implementação desses tipos complexos foi necessária para o Gerador de Editores XML e o resultado disso pode compor o pacote das estruturas sugerido no item anterior. Este grande pacote, juntamente com o já existente de Kawaguchi[20], seria uma ferramenta muito útil e poderosa.

- **Unicidade (Chaves e Identificadores)**
  A grande limitação da biblioteca dos tipos básicos usada está ligada ao tratamento de unicidade global ou local de valores. Um dos tipos predefinidos do Schema é o tipo *ID*. O *ID* é um identificador global, ou seja, qualquer valor do tipo *ID* deve ser único no documento inteiro. Por mais que se estenda esse tipo, o critério de unicidade continua sendo global, o que nem sempre é interessante. O Schema

permite a criação de chaves para escopos a serem escolhidos, através do uso de *key* ou *unique* na declaração de um elemento. Em primeiro lugar, a biblioteca de Kawaguchi [20], apesar de validar o tipo *ID*, não valida seu critério de unicidade. Em segundo lugar, as chaves, de uso muito freqüente, não são tipos, mas sim, parte da declaração de um elemento e, portanto, se encaixariam na implementação das **estruturas** do XML Schema e não dos **tipos**. A dificuldade de implementação dos critérios de unicidade existe porque esses critérios dependem diretamente do documento XML sendo processado. Não é simples construir um pacote separado que cuide desses critérios. A implementação feita desses critérios para o Gerador de Editores XML está longe de se tornar um pacote fechado mas as estratégias utilizadas são interessantes e merecem ser citadas. A idéia principal é impedir a instanciação direta de qualquer classe que represente os tipos do XML Schema. Para isso, a implementação da interface Schema, uma das três partes em que foi dividido o editor (veja o Capítulo 5 para mais detalhes), é quem fica responsável por isso. Desta maneira, ela pode gerenciar tanto as chaves quanto os IDs. Se todas as classes que ele instancia possuírem o método *setSchema*, suas instâncias terão conhecimento do objeto Schema que a criou e podem, assim, pedir permissões quando da modificação de seus valores. O método da interface Schema que retorna instâncias, recebe sempre como parâmetro um nó da árvore DOM, possibilitando assim associar o escopo do nó às declarações de chaves do Schema. Todo esse processo está ilustrado na Figura 6.3.

Além dos conceitos acima, considerados necessários para o trabalho de criação do Gerador de Editores XML, outras implementações também podem tornar-se pacotes compilados. São as implementações feitas para o objetivo específico de visualização gráfica dos tipos. São elas as representações gráficas de:

- **Tipos Simples Predefinidos do Schema**
  Apesar da existência de uma biblioteca que implementa os tipos simples predefinidos do XML Schema, não encontram-se disponíveis visualizações gráficas desses tipos. Diante da necessidade desses componentes gráficos para o Gerador de Edi-

tores XML, uma implementação foi feita seguindo os padrões dos componentes gráficos **Swing**[22]. Todos os componentes do Swing são implementados seguindo **padrões de projeto**, como os apresentados por Gamma [23] e rediscutidos, com o uso da linguagem Java, por Cooper [24]. Além disso, todos os componentes do Swing e, conseqüentemente, todas as visualizações dos tipos simples predefinidos do Schema são **JavaBeans** [25] e podem ser utilizados em ambientes integrados de desenvolvimento (**IDE** — *Integrated Development Environment*). Esses componentes gráficos podem ser distribuídos em um pacote independente.

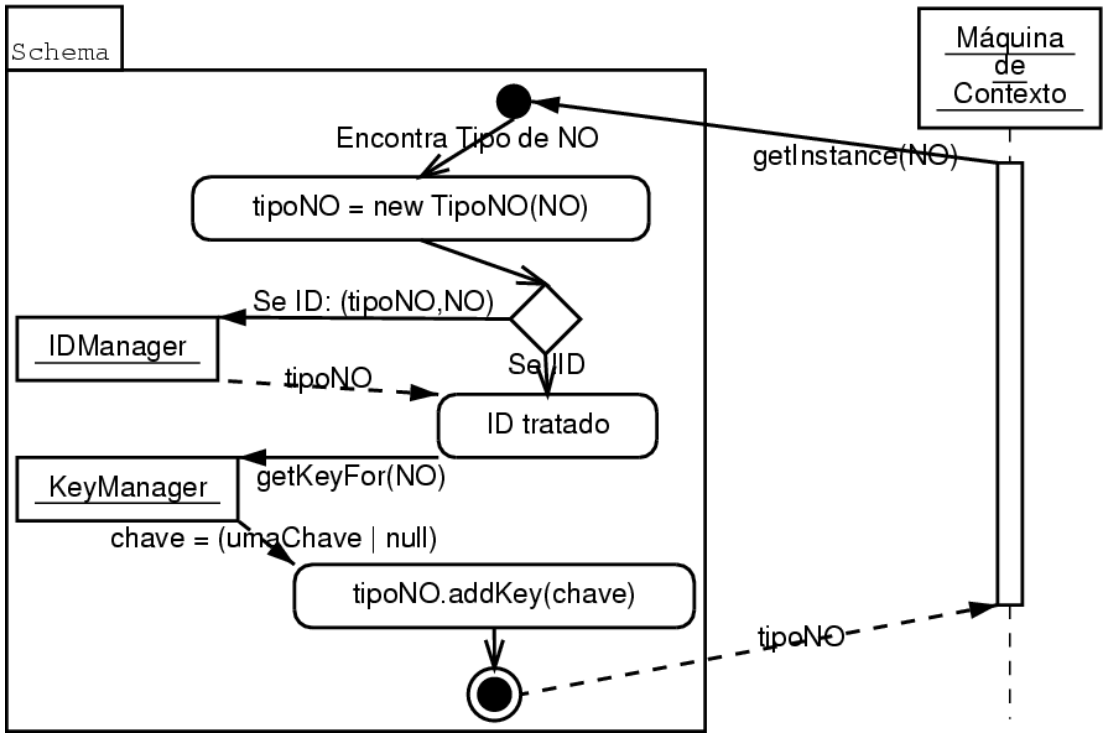

Figura 6.3: Instanciação de um tipo do XML Schema: tratamento de unicidade. Quando uma classe é instanciada, é necessário saber se o nó a ser representado é uma chave (*key*) ou um identificador global (*ID*). Em caso afirmativo, a classe instanciada é informada pelo *IDManager*, para os identificadores, ou através da chamada do método *addKey*, para as chaves.

- **Derivações dos Tipos Simples**
  Baseado nos componentes gráficos dos tipos predefinidos, é possível definir outros que representem tipos derivados, observando a forma de derivação. Utilizando o Processador de Estilos do Adesso, faz-se a geração automática desses componentes gráficos a partir de suas definições no XML Schema. Os componentes gerados também são JavaBeans e podem ser agrupados em um único pacote. O pacote gerado, porém, depende dos componentes gráficos dos tipos predefinidos, já que muitas de suas classes estenderão classes predefinidas. Existe a possibilidade de criação de uma ferramenta que una a implementação dos tipos predefinidos à geração automática de tipos derivados.

- **Modelos de Agrupamento**
  Existem inúmeras maneiras de organizar visualizações gráficas de elementos para um mesmo modelo de agrupamento. A escolha de uma determinada organização depende do gosto pessoal do programador e do tipo de aplicação. A organização escolhida para o Gerador de Editores XML baseou-se no Adesso, com o objetivo de se fazer a edição de *toolboxes*. A busca de novas organizações através da análise de diferentes aplicações é uma proposta de continuação deste trabalho. O uso da visualização gráfica de um modelo de agrupamento está fortemente atrelado à funcionalidade de um editor. A importância real dos componentes que representam modelos de agrupamento está em compor os tipos complexos, discutidos no próximo item.

- **Tipos Complexos**
  Um tipo complexo possui atributos e conteúdo. A representação gráfica de um tipo complexo pode ser dividida em um painel de atributos e um painel de conteúdo. Seu conteúdo está organizado seguindo um modelo de agrupamento e sua forma de visualização é nada mais que a visualização do modelo de agrupamento, já discutida no item anterior. O painel de atributos agrupa os componentes gráficos dos atributos já declarados e oferece opções de adição de novos ou remoção. Como não existem tipos complexos predefinidos, todas as classes que representam tipos

complexos são geradas automaticamente. É imediato pensar na criação de um JavaBean para cada tipo complexo, contendo métodos get/set para seus atributos. Isso sem dúvida é inevitável se for preciso usar os componentes gráficos em um IDE. Mas, quando se trata de código gerado, este padrão de projeto se complica. Isso porque grande parte da implementação dos tipos complexos pode ser feita em uma superclasse abstrata, para evitar repetição de código e a geração de um programa muito extenso. É impossível para esta superclasse conhecer os nomes dos atributos da subclasse senão através de uma lista desses nomes. Isso obriga chamadas de métodos através de reflexão, igualmente aumentando a quantidade de linhas de código com os muitos gerenciamentos de exceção necessários. Por esta razão, foi mais fácil criar o método *getAttribute(String name)* e seu respectivo *set*, fugindo do padrão JavaBeans. Todavia, nada impede que um Bean seja gerado. É possível fazer a opção no momento da geração de código.

A geração automática de código exige muita atenção por parte do programador na tentativa de reduzir o número de linhas de código. É necessário definir de antemão alguns padrões para todas as implementações, já que elas são feitas apenas uma vez nas folhas de estilo. Numa aplicação como o Gerador de Editores XML, a busca pela clareza de código de programa deve estar presente tanto no momento da implementação das classes de código fixo tanto no momento da escrita das folhas de estilo. Tanto a folha de estilo deve ser clara quanto o código por ela gerado.

É importante notificar o surgimento recente de duas ferramentas que implementam parte dos pacotes descritos nesta seção. A primeira delas, parte integrante do ambiente de desenvolvimento Eclipse, surgiu em outubro de 2002 sob o nome de XSD Infoset Model[26]. O XSD Infoset Model é uma biblioteca para manipulação de componentes de um XML Schema. Para isso, implementa tipos simples e complexos e modelos de agrupamento. No entanto, estas implementações estão fortemente enraizadas em um conjunto de outras bibliotecas que acompanham o ambiente de desenvolvimento e não são de fácil uso de forma isolada. Uma segunda ferramenta, denominada XMLBeans[27], é ainda mais recente, tendo sido anunciada em fevereiro de 2003. Trata-se de uma ferramenta que gera automaticamente JavaBeans que implementam os

tipos de dados definidos em um XML Schema. É exatamente o que faz o Gerador de Editores XML, exceto que não gera representações gráficas dos tipos do esquema. Mesmo considerando que, de posse de uma ferramenta como esta, muito trabalho teria sido poupado na implementação do Gerador de Editores XML, o uso da versão em questão teria sido descartado por ser uma versão que ainda não possui uma licença de uso muito clara. O desenvolvedor disponibilizou gratuitamente o XML Beans por um período experimental e ainda não se manifestou oficialmente sobre as formas de distribuição e/ou comercialização desta ferramenta.

Capítulo 7

O Gerador de Editores XML para a Web

A ubiquidade da Internet nos dias atuais salienta ainda mais as qualidades do software baseado em componentes — a Internet é um vasto ambiente de integração de componentes, causando um forte movimento dos maiores produtores de software nesta direção. A oferta de soluções de integração de software baseadas na rede mundial cresce a cada dia, configurando um mercado amplo e diversificado. O Adesso é baseado em uma das principais tecnologias oriundas da Internet, o XML, e se adapta naturalmente à utilização neste ambiente.

Recentemente, iniciou-se a execução de um projeto que visa complementar o Adesso com funcionalidades para operação distribuída através da Internet. Este projeto, denominado **AdessoWeb**[28], estabelece um ambiente de colaboração via Web para o desenvolvimento de software científico.

7.1 Apresentação do AdessoWeb

O projeto AdessoWeb visa a criação de um ambiente distribuído, utilizando a Internet, para gerenciar o trabalho colaborativo de desenvolvimento de aplicações científicas. O projeto parte do sistema Adesso para transformá-lo em um sistema de objetos distribuídos baseado na Web, fazendo uso das tecnologias mais recentes associadas à Internet.

O AdessoWeb tem como objetivos abordar os seguintes aspectos fundamentais:

- desenvolvimento colaborativo distribuído, utilizando interfaces de usuário baseadas em *navegadores Web*;
- geração de distribuições binárias para várias plataformas através de um *parque de compilação*;
- suporte à criação de aplicações distribuídas;
- introdução do conceito de *projeto*, significando um conjunto de atividades, distribuídas no tempo e entre usuários com diferentes funções, visando um determinado objetivo final;
- consolidação do sistema Adesso visando atender a demandas de outros grupos de pesquisa.

O AdessoWeb estende o Adesso através de sua inserção em uma plataforma de serviços distribuídos via Web e da inclusão da noção de *projeto*, um ambiente de colaboração entre vários usuários para o desenvolvimento de uma aplicação. Um *projeto*, neste sentido, envolve vários agentes (desenvolvedores) com funções e direitos de acesso específicos, definidos com auxílio do AdessoWeb, que cooperam para a criação de uma ou mais *toolboxes*.

O AdessoWeb é um sistema de *objetos distribuídos* operados, primariamente, através da rede mundial de computadores, com auxílio de navegadores Web, presentes em todos os sistemas computacionais da atualidade. A Figura 7.1 mostra a arquitetura prevista para o AdessoWeb. Os principais componentes são descritos na seqüência.

O AdessoWeb é formado por um conglomerado de computadores e programas interconectados por uma rede local que prestam serviços a seus clientes através da Internet. O componente central deste conglomerado é o **Servidor AdessoWeb**. Este servidor hospeda os programas que implementam os principais serviços do sistema.

- **Servidor Web**
  O sistema proposto é desenhado de forma a se aproveitar da onipresença dos navegadores Web para facilitar sua utilização remota. O servidor Web é o elemento que propicia o contato do AdessoWeb com seus clientes. Contudo, as maneiras tradicionais de implementação de *sites* dinâmicos são substituídas, neste projeto, por técnicas de última geração que exploram a aplicação de conceitos advindos da

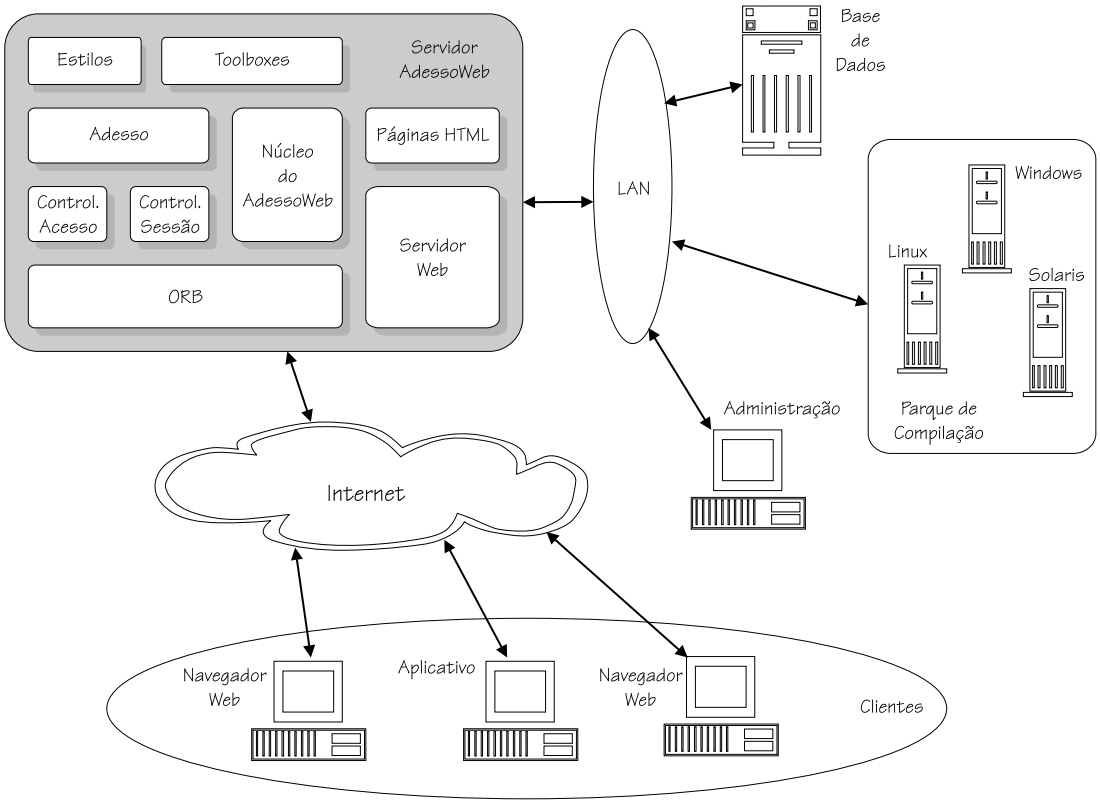

Figura 7.1: Organização do AdessoWeb.

programação orientada a objetos. Os navegadores têm, porém, importante papel como ambiente de execução de *agentes* enviados pelo servidor.

- **ORB**
  O elemento central do sistema de comunicações do AdessoWeb é seu **ORB**, *Object Request Broker*. A filosofia da plataforma distribuída proposta é fortemente calcada nas idéias que constituem a tecnologia **CORBA**, sendo que a implementação provavelmente utilizará componentes CORBA existentes no mercado. A função do ORB é possibilitar que objetos residentes em diferentes máquinas possam ser acessados como se fossem objetos locais. O ORB, embora na figura apareça apenas na representação do servidor AdessoWeb, é parte de todos os demais componentes, inclusive dos agentes remotos que são executados em computadores dos clientes.

- **Controle de Acesso**
  Os serviços de controle de acesso permitem a atribuição de diferentes papéis em um projeto a diferentes usuários, principalmente pela restrição do acesso a diferentes funcionalidades do sistema. Os serviços de controle têm, também, a função importante de garantir a segurança das informações contidas no servidor.

- **Controle de Sessão**
  O controle de sessão possibilita a persistência de estado entre acessos, garantindo a continuidade da sessão e habilitando funcionalidades como a de *desfazer* ações incorretas. Permite ainda que uma determinada sessão seja retomada por seu cliente em outra ocasião.

- **Núcleo do AdessoWeb**
  Este módulo implementa o sistema de objetos distribuídos que incorpora ao Adesso as funcionalidades de um ambiente distribuído para o desenvolvimento cooperativo.

  As diversas classes de objetos e suas instâncias estão, na verdade, distribuídas por todo o sistema, seja em computadores conectados à rede local, seja nas máquinas

cliente. Estes objetos podem assumir diferentes formas: (a) agentes que se utilizam do ambiente de execução de navegadores Web; (b) agentes que executam em aplicativos específicos residentes no lado do cliente; (c) programas residentes em máquinas do AdessoWeb que disponibilizam determinados serviços; etc.

As principais funcionalidades acrescidas ao Adesso são:

- controle de transações e persistência dos dados através de bases de dados orientadas a objetos; os dados são modelados como árvores XML, usando o XML Schema;
- agentes executados no cliente para a criação e manutenção de *toolboxes*, geração de código e empacotamento, baseados em ferramentas XML;
- controle do processo de construção de distribuições binárias em diversas plataformas computacionais;
- suporte ao desenvolvimento de aplicações distribuídas;
- controle de versões;
- gerenciamento de *projetos*.

- **Adesso**
  Este componente é o sistema Adesso, com seus estilos de transformação e *toolboxes*, já desenvolvido e descrito neste documento.

- **Parque de Compilação**
  É formado por um grupo de máquinas com sistemas operacionais diferentes e com o suporte necessário para a criação de distribuições binárias das *toolboxes* gerenciadas pelo Adesso (compiladores e ferramentas associadas).

- **Base de Dados**
  As informações mantidas pelo AdessoWeb, *toolboxes*, estilos etc., serão armazenadas em uma base de dados com interface baseada em XML.

- **Interfaces de Administração**
  São sub-sistemas que permitem a configuração e manutenção da plataforma através da rede local ou via agentes em máquinas remotas.

7.2 O Gerador de Editores XML e o AdessoWeb

A adaptação do Gerador de Editores XML para o AdessoWeb é extremamente desejável, já que, com a conclusão desse projeto, todos os usuários do sistema Adesso migrar-se-ão automaticamente. Obviamente, toda adaptação tem um determinado custo, principalmente quando exige-se grande quantidade de reescrita de código. Quando no projeto de uma aplicação são feitas corretamentes as previsões de uso futuro, o modelo pode ser antecipadamente estruturado para receber com facilidade as mudanças ou inclusões de código. Independente da previsão que o Adesso já fazia com relação ao AdessoWeb, seria inaceitável se, no início do desenvolvimento deste gerador de editores, fosse descartada a hipótese de seu uso na Internet. Isso porque o XML está intimamente ligado à Internet e tem, nesse mesmo ambiente, sólido suporte da linguagem Java, escolhida para a implementação dos editores gerados. Esta adaptação não pode ser muito onerosa.

Antes de descrever os passos necessários para a adaptação do editor gerado, vamos posicioná-lo em seu novo ambiente. Voltando à Figura 7.1, é imediato considerar que o editor deve ser colocado do lado do cliente, como um aplicativo, e trocar mensagens com o sistema AdessoWeb através de trocas de mensagens pela Internet. Isso significaria, basicamente, que as funcionalidades do editor gerado, incluindo edição e validação em tempo de execução, estariam, na sua totalidade, presentes na máquina do cliente. As mensagens trocadas via Internet seriam as requisições dos serviços do sistema simplesmente. Mas existem outras abordagens possíveis.

No Capítulo 5, foi apresentado o modelo do Gerador de Editores XML e foi apresentada uma alternativa que subdivide os papéis do editor gerado em três grandes módulos, como mostra a Figura 7.2. A comunicação entre quaisquer dos módulos poderia ser executada via Internet, se fosse desejado. Esta mobilidade pode ser explorada em virtude

de uma análise comparativa de custos e benefícios de diferentes abordagens.

Os três módulos podem encontrar-se no cliente, como mostra a Figura 7.3. Neste caso, o editor seria gerado exatamente como é feito hoje, compilado e disponibilizado para *download*. Na Seção 6.1, foi apresentada a possibilidade de associação de um menu do editor à execução de um aplicativo externo qualquer. Pode-se executar, por exemplo, um cliente de acesso ao servidor AdessoWeb, e o editor estaria pronto para o novo ambiente. O custo desta solução é muito baixo, pois implica apenas na implementação de uma única classe, que acessa o servidor. A grande limitação desta abordagem é que o documento XML fonte precisa estar inteiro no cliente e poderia causar conflitos com outros usuários que estivessem manipulando o mesmo documento. Não seria possível fazer atualizações pontuais de partes do documento. Haveria apenas uma atualização completa em algum sistema de controle de versão, como o CVS. Além disso, qualquer atualização do esquema exigiria a geração de um novo editor e, conseqüentemente, novo *download*.

O problema de atualização do esquema é facilmente resolvido se o módulo Schema for transferido para o lado do servidor, como mostra a Figura 7.4. Em uma eventual modificação do esquema, apenas o módulo Schema precisa ser novamente gerado e atualizado

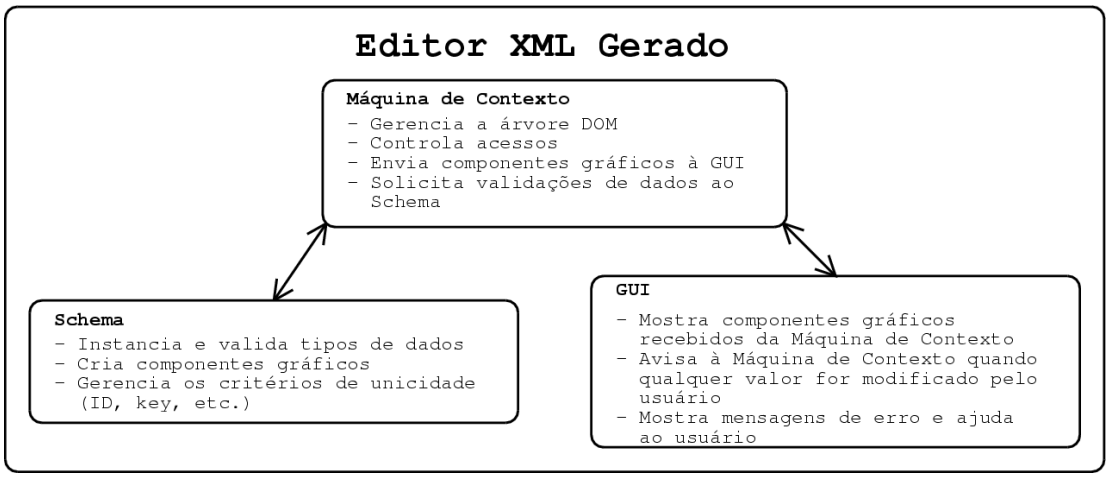

Figura 7.2: O editor XML subdividido em três módulos.

no servidor, não sendo necessário novo *download*. O custo desta segunda abordagem, mesmo que um pouco superior, seria semelhante ao da anterior. Seria exatamente o custo da criação de elementos cliente/servidor para intermediar a comunicação entre os módulos Máquina de Contexto e Schema. O problema da localização do documento fonte e edição concorrente de um mesmo documento ainda não é resolvido pois a representação dinâmica do documento XML fonte, a árvore DOM, continua do lado do cliente, na Máquina de Contexto.

Se também a Máquina de Contexto for transferida para o servidor, passaria a existir

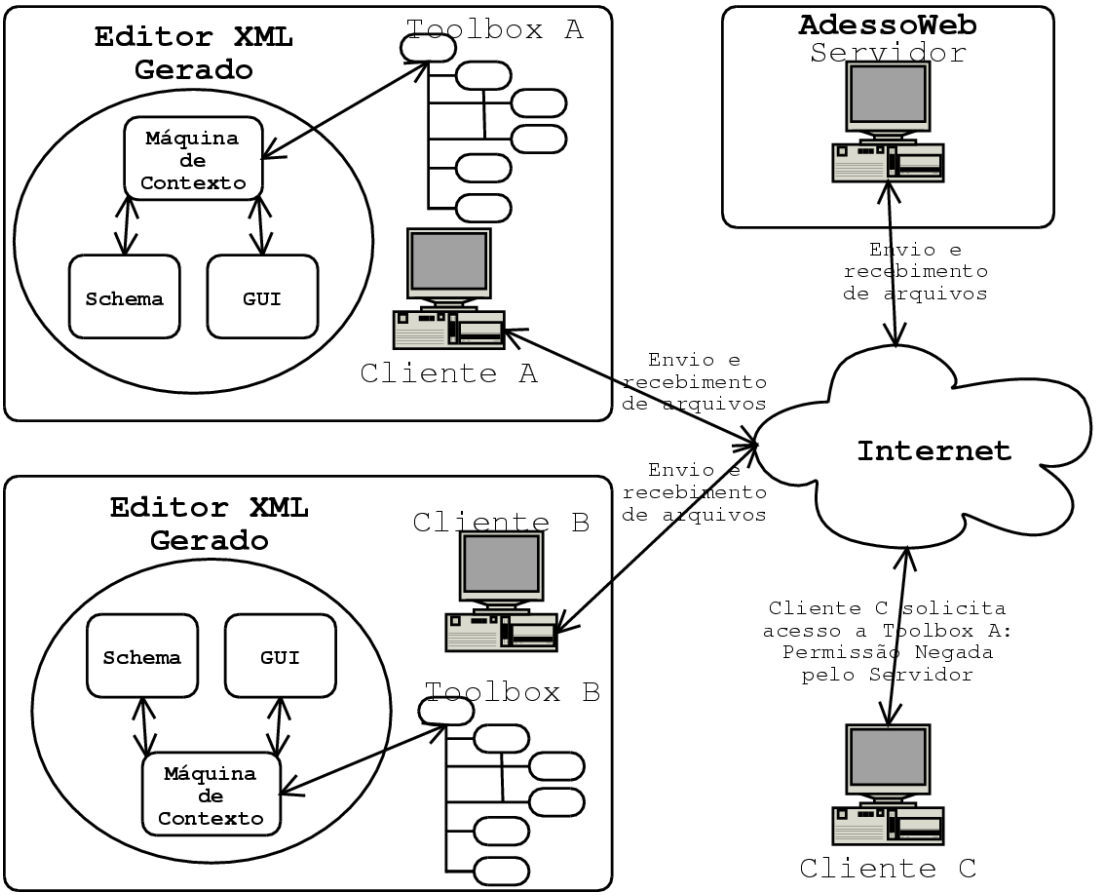

Figura 7.3: O editor no AdessoWeb: três módulos do lado do cliente.

apenas uma árvore DOM instanciada no servidor para cada documento editado, como mostra a Figura 7.5. O cliente teria somente a janela de edição (GUI) na máquina local. Todo o resto ficaria no servidor. Cada elemento diferente do documento XML que fosse selecionado no editor durante a execução, geraria uma requisição de serviço, via Web, à Máquina de Contexto. Os objetos recebidos e mostrados pela janela de edição não só seriam capazes de fazer validações em tempo de execução (sem requisições)

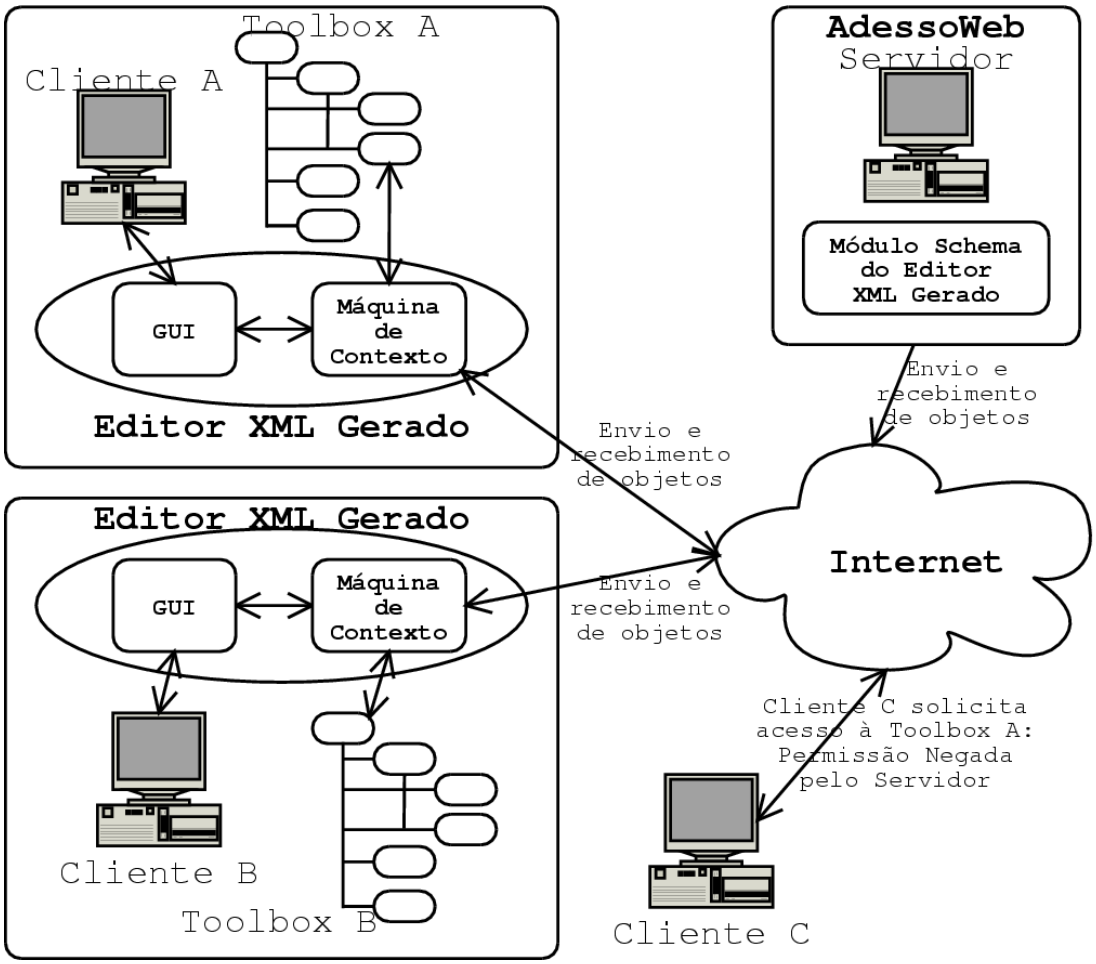

Figura 7.4: O editor no AdessoWeb: Schema no servidor. Com esta organização, não existe mais apenas troca de arquivos pela Internet. Todos os componentes gráficos são instanciados no módulo Schema, presente no servidor, e transferidos para o cliente.

como também poderiam estar sincronizados com outros usuários quaisquer. Assim, um mesmo documento poderia ser editado por vários usuários ao mesmo tempo, desde que

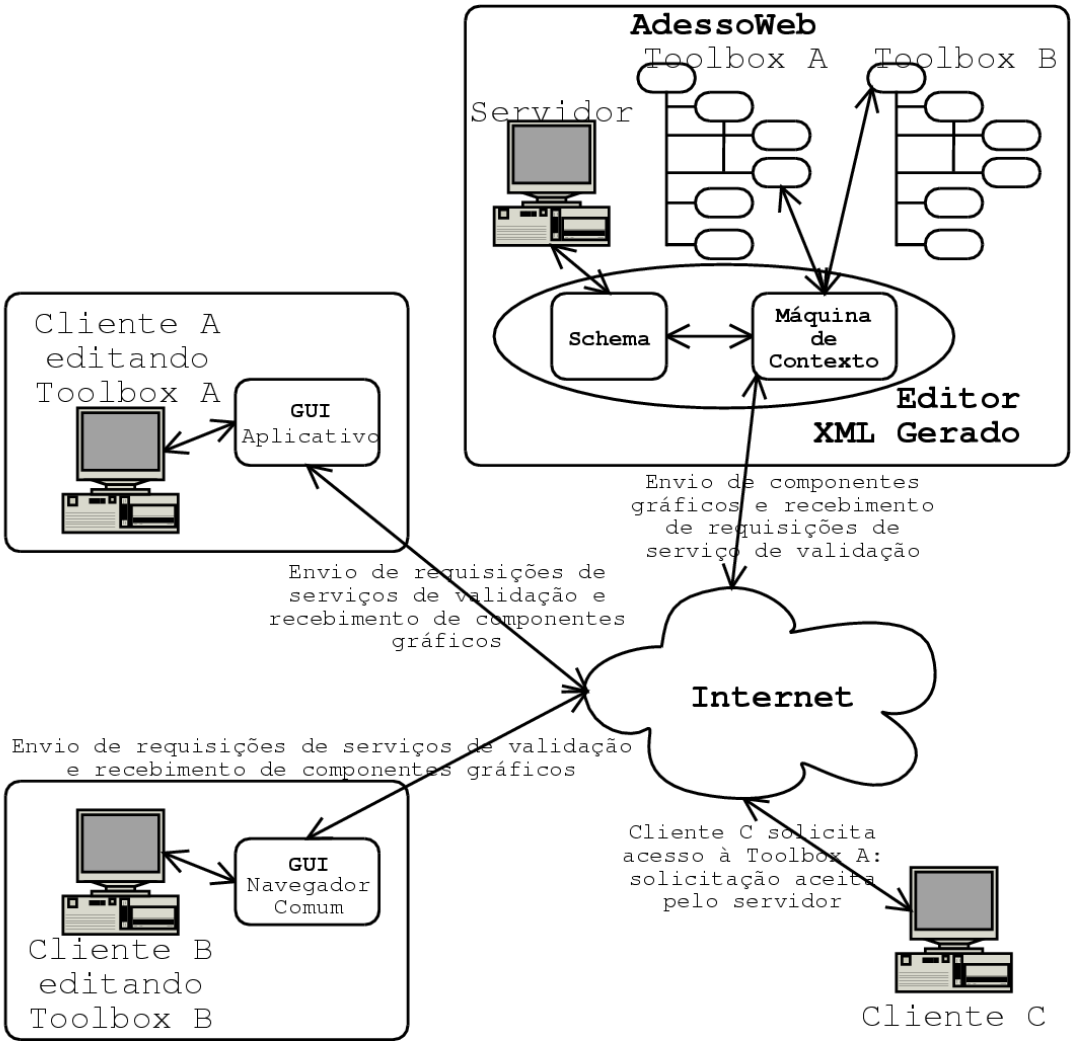

Figura 7.5: O editor no AdessoWeb: Schema e Máquina de Contexto no servidor. Com esta organização, a árvore DOM é gerenciada dentro do servidor pela Máquina de Contexto. Um mesmo documento pode ser editado por mais de um cliente ao mesmo tempo. Esta organização pode também ser implementada usando um Navegador comum como janela de edição.

estes estivessem trabalhando em diferentes nós da árvore DOM. A principal vantagem desta abordagem é que as únicas diferenças dela para os editores gerados atualmente são a questão da transferência de objetos pela Internet e o sincronismo multiusuário. O custo aqui também seria dimensionado pelo custo da criação de elementos cliente/servidor que serviriam de intermediários na comunicação entre dois módulos, sendo, no caso, entre a GUI e a Máquina de Contexto.

Em ambos os casos em que o Schema está no servidor, existe a desvantagem do grande tráfego de objetos pela rede. Uma característica inerente à linguagem utilizada. Componentes gráficos em Java são pesados e o desempenho do editor cairia drasticamente.

Outra opção, e talvez a mais viável do ponto de vista dos clientes, é ter como janela de edição um navegador web comum, como será discutido a seguir.

7.3 Um Navegador Comum

Adotar um navegador web comum como janela de edição significa que os componentes visualizáveis enviados pela Máquina de Contexto devem ser implementados em linguagens a ele familiares. Seja em applets Java, seja em qualquer outra linguagem da Internet. No Capítulo 5 foi mostrado que para cada tipo simples do esquema são geradas duas classes, sendo uma para representá-lo e validá-lo e outra para visualizá-lo e interagir com o usuário.

Tomando como exemplo as classes *Adyear* e *AdyearVisual* que são, respectivamente, os componentes de validação e de apresentação do tipo *Adyear*, apresentados no Capítulo 5. Alguma modificação feita no valor de um componente *AdyearVisual*, este componente deve solicitar a *Adyear* a validação do novo valor. A validação e a apresentação não são feitas por duas classes distintas devido à distinção entre seus papéis. O uso do mecanismo de herança para implementar os componentes de validação é muito importante para preservar os conceitos da norma XML Schema, além de garantir maior clareza do código. Os componentes de apresentação ou visualização, por sua vez, não exigem a presença desse mecanismo e podem até mesmo ser implementados por uma linguagem

qualquer que não seja orientada a objetos. Formulários HTML são um bom exemplo de que é possível definir componentes de interface gráfica de forma simples, mantendo um desempenho aceitável.

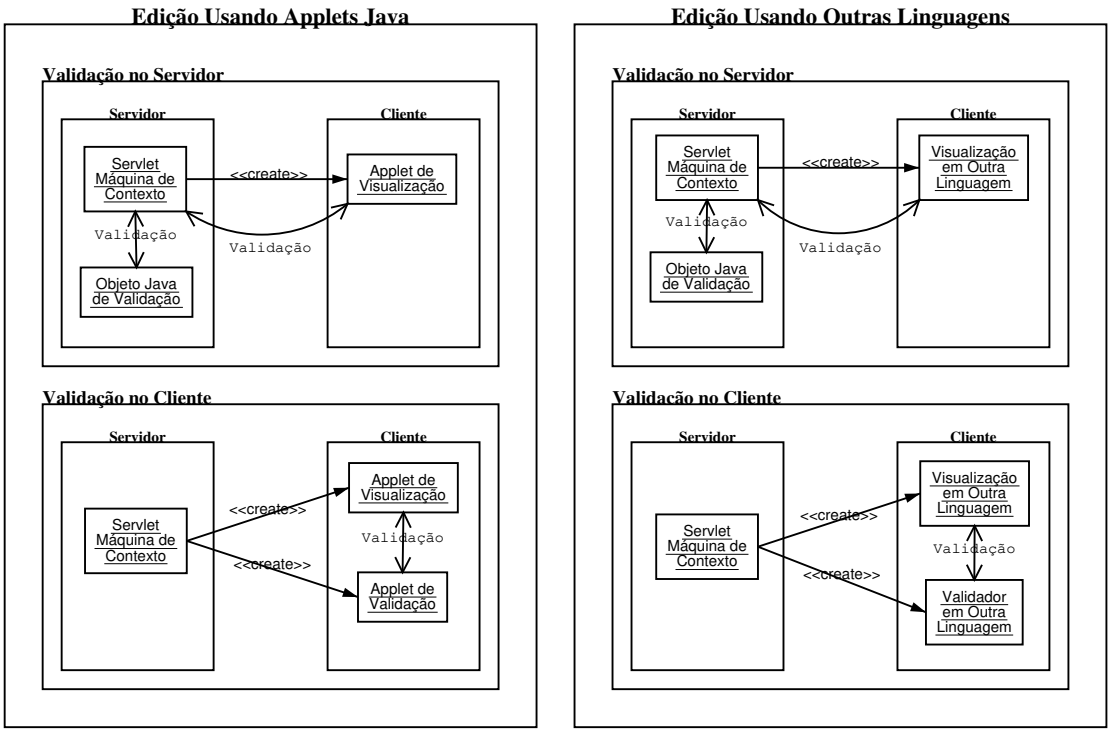

Figura 7.6: Como modificar o Gerador de Editores XML. Algumas maneiras de modificar o Gerador de Editores XML para inserí-lo no ambiente distribuído do AdessoWeb usando um navegador comum.

Algumas maneiras de implementação do Gerador de Editores XML utilizando um navegador comum são mostradas na Figura 7.6. Nesta figura, a idéia que menos implicaria em trabalho adicional seria a transformação das classes de apresentação em applets Java. A diferença entre uma classe Java comum e um applet Java é muito pequena, porém a ineficiência da execução de applets complexos em navegadors web é conhecida e pode ser evitada. A segunda idéia é fazer a apresentação usando HTML ou JavaScript, linguagens que apresentam melhor desempenho que o Java na Internet, e fazer a validação

através de requisições ao servidor. O servidor usaria as próprias classes Java de representação e validação já existentes. O trabalho adicional seria a geração de visualizações em outra linguagem. Um inconveniente seria a necessidade de requisições de validação que, feitas pela Internet podem ficar mais lentas. A terceira opção é possibilitar ao cliente que faça a validação localmente, sem requisições ao servidor. Para isso é necessário (evitando ainda os applets) gerar também os componentes de representação e validação em outra linguagem. Neste caso é importante lembrar do modelo de classes do Schema, que utiliza o mecanismo de herança, como já falamos. É necessário que a linguagem usada atenda a este requisito. O JavaScript é uma linguagem nesses padrões e é eficiente nos navegadores web. Gerar os componentes de representação e validação, no entanto, não é nada simples e sua implementação pode ser onerosa.

Finalmente, fazendo um balanço de custos e requisitos de todas as possibilidades de adaptação do Gerador de Editores XML para o AdessoWeb, pode-se concluir que:

- Dos três módulos do editor, apenas a janela de edição (GUI) deve ficar do lado do cliente. A presença da Máquina de Contexto no servidor é importante para o acesso concorrente de um determinado documento XML.

- A transferência de objetos complexos pela rede afeta o desempenho do aplicativo. É mais prudente usar requisições de serviço para a validação de valores do que transferir para o cliente toda a informação sobre um determinado elemento. Requisições de serviço permitem que um navegador web comum seja usado como janela de edição.

- É necessário implementar os componentes gráficos em uma linguagem que possa ser compreendida pelos navegadores comuns.

- Apesar de a adaptação menos onerosa seja a criação de applets Java, estes devem ser usados com cautela. Uma grande quantidade de applets afeta o desempenho. Linguagens usuais da Internet, como JavaScript, seriam mais eficientes para implementar os componentes gráficos.

- Modificar a linguagem dos componentes de validação, presentes no módulo Schema acarretariam alto custo de programação.

O custo final da adaptação, dentro desses requisitos seria:

- Transformar a Máquina de Contexto em um servidor. Isso implica apenas em fazê-la implementar uma interface de serviços web.
- Implementar os componentes gráficos em uma linguagem leve para uso em navegadores comuns. O custo desta adaptação é alto, mas bem menor do que a primeira implementação dos componentes gráficos.

O uso do modelo de classes e a divisão do Gerador de Editores XML foram primordiais para a viabilização destas adaptações. As modificações a serem feitas são bem localizadas.

Capítulo 8

Conclusões

Este trabalho criou um Gerador Automático de Editores XML, basedo na análise do modelo de dados definido por um XML Schema. O Gerador de Editores XML pode ser útil para uma grande variedade de aplicações.

O conceito fundamental que embasa o aplicativo desenvolvido é o uso de transformações XML através do Processador de Estilos do Adesso ou de ferramentas XSLT para a geração automática da interface gráfica de um editor. Diferentemente das mais comuns transformações XML utilizadas, este aplicativo não processa um documento XML comum mas, sim, o XML Schema, um documento XML especial que define o modelo dos dados permitidos em outros documentos XML. Qualquer modificação necessária do modelo de informação a ser editado passa a ser feito apenas no XML Schema, que fornece informações suficientes para a geração automática de um novo editor, sem necessidade de um programador especializado na linguagem usada para a interface gráfica.

O aplicativo resultante, denominado **Gerador de Editores XML**, permite que aplicações XML em diferentes contextos possuam um editor especializado para seus documentos, simplesmente através da definição de um XML Schema para descrever o modelo da informação.

A utilização do Gerador de Editores XML para a edição de *toolboxes* dentro do sistema Adesso comprovou a funcionalidade e a eficácia da geração automática de uma interface

gráfica, uma vez que as mudanças no modelo de informação das *toolboxes* do Adesso é bastante freqüente.

A criação de um modelo orientado a objetos do editor facilitará uma eventual adaptação do Gerador de Editores XML para que possa gerar editores para outros ambientes como a Internet ou ambientes distribuídos. Permitiu também a decomposição do editor em diversos pequenos pacotes que podem ser disponibilizados e utilizados independentemente. A garantia de portabilidade está no uso da linguagem Java para implementação da interface gráfica e de folhas de estilo para a definição de regras de geração automática de código.

As principais contribuições deste trabalho são listadas a seguir:

- A manutenção das informações das *toolboxes* do sistema Adesso em bases de dados estruturadas é importante para garantir a consistência dos documentos e código gerados para cada *toolbox*. O uso do XML com este objetivo supõe o uso, em conjunto, de um editor. É desejável que se utilize um editor que não permita a geração de documentos inválidos, que possua uma interface gráfica compatível com a aplicação e que possa ser facilmente modificado, devido à freqüência em que ocorrem modificações no modelo da informação. O Gerador de Editores XML atende esses requisitos, sendo vantajoso com relação a um editor especializado, devido ao custo minimizado de modificação da sua interface, e também com relação aos editores XML genéricos, devido à sua interface gráfica muito mais amigável e compatível com a aplicação.

- Por ser gerado automaticamente a partir do XML Schema, a funcionalidade deste editor transcende o sistema Adesso, podendo ser utilizado em qualquer outro sistema baseado em XML. Este editor pode tornar-se mais uma das muitas ferramentas de processamento XML distribuídas gratuitamente.

- Não só o editor completo é uma ferramenta que pode ser disponibilizada gratuitamente, como também partes dele, separadamente. A divisão do editor em pequenos fragmentos bem funcionais possibilitou a criação de pacotes em Java para o

processamento específico do XML Schema. Neste trabalho foram desenvolvidos pacotes que implementam tanto as estruturas quanto os modelos de agrupamentos do XML Schema, além dos componentes gráficos de visualização dos tipos predefinidos.

- Com exceção dos pacotes básicos de código fixo, o desenvolvimento do Gerador de Editores foi feito utilizando folhas de estilo. As folhas de estilo são muito claras e compreensíveis, sendo, conseqüentemente facilmente estensíveis. Com a disponibilização do código aberto as sugestões de aperfeiçoamento do produto aparecerão rapidamente.

Trabalhos Futuros

Os trabalhos realizados no âmbito deste projeto originaram algumas possibilidades de continuação, seja para aperfeiçoamento do gerador de editores através do desenvolvimento de aplicações para serem a ele incorporadas ou a exploração de diversos tipos de aplicação para identificação de novas possibilidades de visualização gráfica, seja para a exploração de sua utilização em ambientes baseados na Internet, aproveitando a larga demanda que esta faz de aplicativos ligados ao XML e seguindo a evolução do projeto Adesso para o AdessoWeb. Qualquer aperfeiçoamento do sistema Adesso pode implicar necessidade de aperfeiçoamento do Gerador de Editores XML.

- **Aperfeiçoamento do Sistema**

  No sistema Adesso, após a edição de uma *toolbox*, são necessários processamentos posteriores para a geração da documentação e de exemplos, a compilação e o empacotamento. Os processamentos posteriores atualmente são realizados separadamente, seja utilizando a linha de comando, seja utilizando o console do Adesso. Seria muito interessante que esses processamentos podessem ser acessados diretamente do editor gerado, para transformá-lo no ambiente completo de desenvolvimento para o sistema Adesso. Além disso, o desenvolvimento de *toolboxes* para o Adesso tem sido feito com o suporte de um sistema de controle de versões, o CVS. Uma maior integração entre os dois sistemas é muito desejável

para facilitar o uso do CVS e facilitar a gerência de configurações. O editor poderia também implementar o acesso à base de dados do CVS. Apesar de as duas propostas acima serem específicas para o sistema Adesso, outras aplicações certamente teriam necessidades semelhantes. A saída é permitir ao usuário especificar, no momento da geração do seu editor, caminhos para aplicativos externos que ele deseja que se tornem diretamente acessíveis.

- **Análise de Diferentes Aplicações**
  É sabido que qualquer interface gráfica carrega as características escolhidas pelo seu programador, influenciado por seu gosto pessoal e, principalmente pela aplicação à qual a interface servirá. A interface do Gerador de Editores XML sofreu grandes influências do que era desejado para a edição de *toolboxes* para o sistema Adesso. Mesmo assim, foi possível perceber as inúmeras possibilidades de agrupamento dos componentes gráficos para a visualização de um mesmo conteúdo. A idéia, é adicionar ao editor a facilidade de se definir **preferências do usuário**, fornecendo a ele sugestões de agrupamento e permitindo assim uma configuração manual da interface gráfica à posteriori. Com este objetivo, é importante que se faça uma extensa exploração de diferentes aplicações XML para o reconhecimento de possíveis padrões de agrupamento. Com a adição desta facilidade, o pontencial do editor seria fortemente ampliado e o número de interessados por esta ferramenta se multiplicaria.

- **Internet**
  O projeto AdessoWeb visa a integração do Adesso a um sistema de autoria baseado na Internet. A utilização do Adesso em conjunto com um sistema de gerência de conteúdo para a Web abre novas possibilidades para o desenvolvimento de software científico, enfatizando a cooperação entre os desenvolvedores. Também está prevista a criação de parques de compilação que viabilizarão a criação centralizada de binários para diferentes plataformas computacionais. Em conjunto com todos esses avanços é necessária a evolução do ambiente de edição das *toolboxes*, tornando-o um aplicativo utilizado pela Internet para a edição de documentos re-

motos. O modelo atual do Gerador de Editores XML já foi elaborado com esta visão futura e, portanto, tenta facilitar ao máximo o trabalho de implementação desse requisito.

Índice Remissivo

Referências Bibliográficas

[1] R. C. Machado, "Adesso - Ambiente Para Desenvolvimento de Software Científico," M.S. thesis, FEEC - UNICAMP, Brasil, jun 2002.

[2] R. C. Machado, R. A. Lotufo, A. G. Silva, and A. V. Saúde, "Adesso - Scientific Software Development Environment," *Journal of Computer Science and Technology*, vol. 3, no. 1, pp. 1–6, April 2003.

[3] A. G. Silva, R. A. Lotufo, R. C. Machado, and A. V. Saúde, "Toolbox of Image Processing Using the Python Language," in *ICIP - International Conference on Image Processing*, 2003.

[4] R. Lotufo, R. Machado, F. Flores, A. Falcão, R. Koo, G. Mazzela, and R. Costa, "Prontovideo – an image sequence segmentation tool applied to video edition," *Sibgrapi*, October 2001, Florianópolis, Brazil, IEEE.

[5] W3C Recommendation, *Extensible Markup Language (XML) 1.0*, second edition edition, Oct 2000, http://www.w3.org/TR/REC-xml.

[6] W3C Recommendation, 2 May 2001, *XML Schema Part 0: Primer*, http://www.w3.org/TR/xmlschema-0.

[7] W3C Recommendation, 2 May 2001, *XML Schema Part 1: Structures*, http://www.w3.org/TR/xmlschema-1-20010502/.

[8] W3C Recommendation, 2 May 2001, *XML Schema Part 2: Datatypes*, http://www.w3.org/TR/xmlschema-2-20010502.

[9] W3C Recommendation 16-Nov-99, *XSL Transformations (XSLT) Version 1.0*, http://www.w3.org/TR/xslt.

[10] Charles F. Goldfarb, "A Generalized Approach to Document Markup," *SIGPLAN Notices*, June 1981, http://www.sgmlsource.com/history/AnnexA.htm.

[11] W3C Recommendation 16-Nov-99, *XML Path Language (XPath) Version 1.0*, http://www.w3.org/TR/xpath.

[12] W3C Recommendation 15 October 2001, *Extensible Stylesheet Language*, Oct 2001, http://www.w3.org/TR/xsl/.

[13] IBM, *Xeena, a Visual XML Editor*, http://www.alphaworks.ibm.com/tech/xeena.

[14] Inc. Altova, *XML Spy 5 User & The Official XML Spy Handbook*, Vervante, September 2002.

[15] Apache XML Project, *Xerces Java Parser*, http://xml.apache.org/xerces-j/index.html.

[16] Campione, Walrath, Huml, and Tutorial Team, *The Java Tutorial Continued: The Rest of JDK*, Addison-Wesley, 1998.

[17] David Carlson, *Modeling XML Applications with UML: Practical e-Business Applications*, Addison-Wesley Pub Co, 1st edition, April 2001.

[18] OMG Object Management Group, *XML Metadata Interchange (XMI) 2.0 Specification*, May 2003, http://www.omg.org/technology/documents/formal/xmi.htm.

[19] Brett McLaughlin, *Java & XML Data Binding*, O'Reilly, fisrt edition, may 2000, ISBN: 0-596-00278-5.

[20] Kohsuke KAWAGUCHI, "Sun's Java[tm] Technology Implementation of XML Schema Part 2," Tech. Rep., Sun Microsystems, http://wwws.sun.com/software/xml/developers/xsdlib/, Apr 2001, Preview Version 1.

[21] Bruce Christensen, "XML Résumé Library," http://xmlresume.sourceforge.net, July 2002.

[22] Campione, Kathy, Mary, and Walrath, *The JFC Swing Tutorial: A Guide To Constructing GUIs*, Addison-Wesley, 1 edition, 1999.

[23] Erich Gamma et al., *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1995.

[24] James W. Cooper, *The Design Patterns Java Companion*, Design Pattern Series. Addison-Wesley, oct 1998, http://www.patterndepot.com/put/8/JavaPatterns.htm.

[25] MageLang Institute, *JavaBeans Short Course*, Sun Microsystems, http://developer.java.sun.com/developer/onlineTraining/Beans/JBShortCourse/.

[26] Shane Curcuru, *Analyzing XML schemas with the Schema Infoset Model*, XSD Schema Infoset Model Documents in the Eclipse Project, July 2002, http://www.eclipse.org/xsd.

[27] David Bau, *XML Beans*, BEA Systems, March 2003, http://dev2dev.bea.com/technologies/xmlbeans/articles/Bau.jsp.

[28] R. C. Machado and R. A. Lotufo et al., *AdessoWeb: Ambiente Distribuído para Desenvolvimento de Software para Processamento de Imagens*, FEEC-UNICAMP/CenPRA-Campinas, 2001, Projeto Fapesp.