Relatório Técnico

Núcleo de Computação Eletrônica

Reengenharia do XUL, Uma Linguagem de Descrição da Interface do Usuário

Oliveira, C. E. T.
Pais, A. P. V.
Brasil, B.
Pereira, L. A.
Becken, V. P.

NCE - 09/01

Universidade Federal do Rio de Janeiro

Reengenharia do XUL, Uma Linguagem de Descrição da Interface do Usuário

Carlo Emmanuel Tolla de Oliveira
UFRJ, NCE
Rio de Janeiro, Brasil, 21941
carlo@nce.ufrj.br

Ana Paula Valente Pais
UFRJ, NCE
Rio de Janeiro, Brasil, 21941
anapaula@ufrj.br

Bárbara Brasil
UFRJ, NCE
Rio de Janeiro, Brasil, 21941
barbara@nce.ufrj.br

Luciane Amorim Pereira
UFRJ, NCE
Rio de Janeiro, Brasil, 21941
luciane@dcc.ufrj.br

Veronica Patron Becken
UFRJ, NCE
Rio de Janeiro, Brasil, 21941
vbecken@nce.ufrj.br

ABSTRACT

In a modern system information, the interface with the user is a concept key for the success. However, even though in the modern language of modeling "Unified Modeling Language (UML)" the specification of the interface with the user did not have the handling due. The language "XML – based User Interface Language (XUL)" [01], recently adopted for the Netscape for the specification of its interface, represents a concrete step in the route of a standard. Although complete and extensible, this language has its scope delimited to the local applications. The fact of XUL be flexible and extensible allows reengineering in its concepts resulting in a more expansive conception of interface with the user. This article considers XUL as a language of abstract interface's description that makes possible diverse rendering over distributed architectures. An abstract description of the interface allows that the final implementation is delayed and customized to the final requirements without additional cost. The proposal expands the semantics of the XUL so the interface's description is build in a model that equally contemplates local and distributed applications. This article still shows to the interactions of XUL with the Python language [06,04] and XSL [04] for automatic translation of models in UML for scripts JSP [15][16][18][20], in the case of interface's render in pages HTML.
Keywords: User Interface, Distributed Objects, UML, XML

RESUMO

Em um sistema de informação moderno, a interface com o usuário é um conceito chave para o sucesso. No entanto, até mesmo na moderna linguagem de modelagem UML a especificação da interface com o usuário não teve o tratamento devido. A linguagem XUL[01], recentemente adotada pela *Netscape* para a especificação da sua interface, representa um passo concreto na direção de um padrão. Apesar de bem completa e extensível, essa linguagem tem o seu escopo delimitado a aplicações locais. O fato de XUL ser flexível e extensível permite que se aplique uma reengenharia em seus

conceitos resultando em uma concepção mais abrangente de interface com o usuário. Este artigo propõe XUL como uma linguagem de descrição abstrata de interface que possibilite diversas renderizações[1] e leve em conta arquiteturas distribuídas. Uma descrição abstrata da interface permite que a implementação final seja postergada e adaptada aos requisitos finais sem custo adicional. A proposta expande a semântica do XUL para que a descrição da interface seja calcada em um modelo que contemple igualmente aplicações locais e distribuídas. Este artigo ainda mostra as interações de XUL com a linguagem *Python* [06] e XSL [04] para tradução automática de modelos em UML para *scripts* JSP [15][16][18][20], no caso de renderização da interface em páginas HTML.

Palavras Chaves: Interface com o Usuário, Objetos Distribuídos, UML, XML

INTRODUÇÃO

As versões antigas do *Mozilla* [14] apresentaram uma interface do usuário dependente de plataforma, que eram definidas em módulos para plataformas específicas. No sentido de melhorar sua definição de interface do usuário, nasceu o XPFE (*Cross Plataform Front End*), que compõe o XPToolkit. O XPToolkit define um conjunto de interfaces com o usuário que independe de plataforma, e a base de sua funcionalidade é o XUL (*XML-based User interface Language*) [29].

O XUL é uma aplicação de XML (*eXtensible Markup Language*) que descreve a aparência (*layout*) e os componentes da interface do usuário. O XUL é usado para definir componentes como: barra de ferramentas, menu e botões. Ele é lido em tempo de execução e permite que o programador faça algumas mudanças na interface sem ter que recompilar o código fonte.

A resposta da aplicação a algum comportamento de um componente XUL (*widget*), exprime o efeito que ele tem sobre a aplicação. Esse efeito é definido como uma combinação do comportamento predefinido da aplicação e da ligação entre esses componentes. Essa ligação é feita pela inclusão de *JavaScript* no XUL, ou pelo próprio código da aplicação. No último caso, a aplicação deverá obter o estado da interface do usuário verificando diretamente o seu *layout*, para poder processar esses dados de acordo com a lógica da aplicação. Uma aplicação real, normalmente, usará uma combinação dessas duas abordagens.

O XUL disponibiliza uma definição estática da interface do usuário. Para que exista uma interação da interface com o usuário, é necessária a utilização de *JavaScript*, ou do próprio código da aplicação, em parceria com o XUL.

A arquitetura do XUL fica então definida da seguinte maneira:

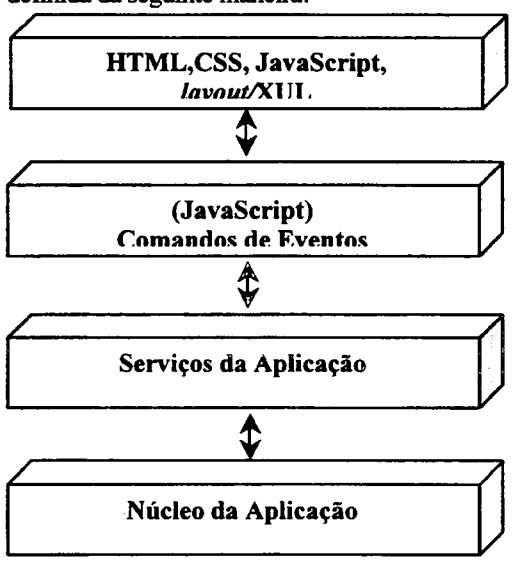

Figura 1 - Arquitetura do XUL

Netscape Gecko é um tipo completamente novo de engenho de navegador (*browser*). Ele utiliza em seu engenho de renderização toda a potencialidade do XUL. Ele foi idealizado para ser pequeno, rápido e compatível com padrões. A grande revolução que esse engenho traz é o fato de prover o desenvolvimento de aplicações *web* multi-plataformas,

[1] O termo renderizar é utilizado no sentido de transcrever a interface gráfica especificada abstratamente em elementos gráficos visíveis na tela de interface com o usuário.

mantendo uma alta performance e uma rica interface com o usuário, através do uso de *JavaScript*, XML [03] e W3C DOM.

O engenho do navegador é um pedaço de software que captura todo o conteúdo da aplicação *web* e o disponibiliza na interface do usuário. Ele é um software embutido sem uma interface própria, fazendo parte da construção de aplicações para usuários finais, tais como *Mozilla* e Netscape6.

O engenho do *Netscape Gecko* possibilita a construção de toda uma interface do usuário de uma aplicação local usando padrões *web*. Uma estrutura utilizando esse engenho poderia ter o seguinte esquema: o XML e HTML estrutura a interface do usuário, o CSS [30] a formata, o W3C DOM [31] manipula os eventos e envia os elementos da interface do usuário para serem tratados pelo controle do *script*. O RDF representa coleções de recursos, como estrutura de diretórios, e *JavaScript* tem o papel de juntar isso tudo.

Este artigo apresenta uma mudança na arquitetura do XUL para que esta passe a incorporar o conceito de modelo, definido na arquitetura MVC (Modelo Vista Controle). É apresentado também uma expansão do conceito inicial do XUL para aplicações distribuídas. A implementação deste trabalho mostra como o conteúdo da modelagem em UML é extraído para o XUL através do *Python* [07][08][09]. Uma arquitetura abstrata de renderização foi desenvolvida para expôr esses modelos de forma a se adaptar facilmente a qualquer interface gráfica. Por fim, a definição do XUL é transformada em HTML, pela utilização do XSL [23].

REENGENHARIA DO XUL DENTRO DO PARADIGMA VISTA-MODELO

O XUL provê a descrição de interface com o usuário, sem a preocupação de como as entradas fornecidas pelo usuário serão tratadas e armazenadas. O XUL não provê um mecanismo que efetive diretamente a interação do usuário com a interface. Não existe qualquer mecanismo que capture as informações da interface, e as disponibilize a quem estiver interessado. O tratamento dessas informações só é possível através de mecanismos externos, como o *JavaScript*.

A necessidade de reestruturar o XUL surge da ausência de uma estrutura própria que mantenha as informações fornecidas pelo usuário à interface. O objetivo dessa reestruturação é tornar a arquitetura do XUL algo mais próximo, por exemplo, da arquitetura *Swing* [12]. Obter uma arquitetura que mantenha uma estrutura interna correspondente ao que é exibido ao usuário, capaz de manter o estado atual do aplicativo.

A idéia que está por traz da biblioteca de componente *Swing* é a arquitetura Modelo-Vista-Controle (MVC). A arquitetura MVC divide o aplicativo de software em três partes distintas: modelo, vista e controle. O modelo é responsável por armazenar as informações sobre o estado do aplicativo. A vista determina a representação visual das informações contidas no modelo. O controle é responsável por determinar como e quando o aplicativo deverá reagir às entradas do usuário.

O *Swing* é baseado numa versão modificada da arquitetura MVC. Um componente *Swing* pode ser descrito como a combinação de dois elementos principais: vista e modelo. A vista corresponde a parte visível do componente, com a qual o usuário pode se comunicar. O modelo mantém o estado do componente, que são as informações submetidas pelo usuário. O modelo permite que determinados tipos de objetos se registrem como ouvintes, a fim de serem notificados de eventuais mudanças ocorridas no modelo. Os ouvintes estão atentos a interação do usuário com o componente.

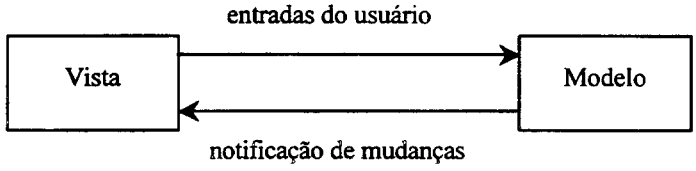

Figura 2 - canal de dados

A vista e o modelo estabelecem um canal de comunicação por onde os dados transitam. Os dados refletem as entradas do usuário, que modificam o estado do componente. No caso de uma simples caixa de texto, cada caracter digitado na interface é transmitido ao modelo. Todos os objetos interessados no conteúdo da caixa de texto se registram como ouvintes do modelo. Cada vez que o modelo sofre uma modificação, os ouvintes recebem um aviso, e podem obter do modelo o conteúdo atualizado da caixa de texto. Isto permite que o conteúdo da caixa de texto possa ser exibido por outro componente. O texto digitado pode ser exibido por um painel, cuja vista é ouvinte do modelo da caixa de texto. Cada vez que a vista recebe um aviso de alteração, ela busca o novo conteúdo e atualiza a interface.

Um componente *Swing* é descrito por uma vista e uma interface, enquanto que um elemento XUL é apenas a descrição da vista. A reestruturação do XUL tem como ponto de partida a especificação de modelos que correspondam a descrição visual. Os modelos constituem elementos capazes de armazenar qualquer tipo de informação obtida da interface ou fornecida à interface. Um botão, geralmente, tem uma ação associada que resulta em algum tipo de informação transmitida ao usuário, enquanto que uma caixa de texto permite que o usuário modifique o seu conteúdo, armazenando informações fornecidas pelo usuário.

A descrição visual dos elementos deve ser construída de tal forma, que contenha apenas características pertinentes à apresentação dos elementos na interface com o usuário. Essa descrição é o roteiro de como a vista será criada. A descrição deve conter o tipo do elemento e informações que forneçam mais detalhes de como o elemento será visto pelo usuário. A descrição de um elemento que exiba texto, por exemplo, deve especificar se esse texto é fixo, ou se esse elemento será apresentado como uma caixa de entrada de texto. Por outro lado, a descrição do elemento não precisa especificar a informação inicial apresentada ao usuário. No caso de um texto, não é necessário que a descrição do elemento contenha o texto a ser exibido na interface. O conteúdo do texto pode mudar. O conteúdo do texto corresponde ao estado do elemento visual.

A descrição de uma interface é algo estático, enquanto que a sua apresentação e seu conteúdo podem ser dinâmicos. A descrição da interface não precisa prover a descrição de seu estado inicial. O estado deve ser especificado em outro tipo de estrutura, que mantenha uma correspondência com a interface. Essa estrutura constitui o modelo. O modelo deve manter todo o tipo de informação que diz respeito ao estado do elemento, enquanto que a vista tem a responsabilidade de fornecer a representação visual do elemento. O modelo guarda a informação atual contida na interface.

O XUL, na sua forma atual, é voltado para a descrição de uma interface dedicada. Um documento XUL descreve a forma final de um aplicativo local, que possui uma arquitetura subentendida. O avanço da tecnologia e as exigências do mundo moderno provocaram o surgimento de alternativas de interface com o usuário. Serviços de atendimento automático, por exemplo, usam o teclado do telefone como interface com o usuário. Portanto é desejável que a linguagem XUL seja estendida para atender uma gama maior de representações de interface.

A revisão semântica da linguagem XUL permitirá um melhor aproveitamento de seu potencial. O desenvolvimento de um aplicativo deixa de ser restrito a um determinado tipo de interface. A encarnação da descrição contida no documento XUL é feita por um engenho, responsável por traduzir elementos XUL em elementos que estabeleçam comunicação com o usuário. O mesmo engenho cria os modelos necessários, concretizando a ligação entre vista e modelo. Cada tipo de interface com o usuário tem o seu próprio engenho, que preserva a semântica contida na descrição XUL.

TRADUÇÃO DA MODELAGEM UML EM SCRIPTS XUL

A construção de um aplicativo de software inicia com um conjunto de descrições, que traduzem a visão que o usuário tem do aplicativo. Isto envolve a descrição dos serviços fornecidos pelo aplicativo. Cada serviço recebe um determinado tipo de informação do usuário, trata a informação e fornece um resultado ao usuário. A descrição desses elementos constitui a modelagem do aplicativo. A UML é uma forma padronizada de fazer essa descrição. A modelagem resulta num conjunto de diagramas que representam o funcionamento do aplicativo, e descrevem como o usuário interage com os diversos serviços que constituem o aplicativo.

A modelagem de um aplicativo, quando concluída, constitui um roteiro de como o aplicativo será construído. A análise de caso de uso [05] é uma das primeiras etapas da modelagem. O principal objetivo dessa etapa é a descrição de cenários de interação entre o usuário e o aplicativo. O resultado desta etapa é uma descrição textual do funcionamento do aplicativo. Outra etapa importante da modelagem é a análise de robustez, introduzida por Ivar Jacobson em 1991. Este conceito envolve a análise da narrativa dos casos de uso e identificação do conjunto inicial de classes que participam do caso de uso. Esses elementos são classificados em três categorias principais: elementos de interface, entidades e controles.

Elementos de interface são os elementos com os quais o usuário pode se comunicar. Entidades são objetos do domínio da aplicação. As entidades são as classes resultantes da análise de domínio. Essas classes representam as informações e conceitos manipulados pelo aplicativo. Clientes e pedidos são exemplos de entidades de uma livraria on-line. Os elementos de controle estabelecem a comunicação entre a interface e as entidades.

O diagrama de robustez é uma representação gráfica do roteiro do caso de uso. A mesma informação obtida da leitura do roteiro do caso de uso, deverá ser obtida do diagrama de robustez correspondente. A partir da leitura de ambos, deve ser possível que o usuário compreenda precisamente como o sistema irá funcionar. O diagrama de robustez mostra como as classes interagem na execução de cada um dos passos descritos no caso de uso.

O diagrama de robustez contém elementos que representam componentes da interface com o usuário. Esses elementos são vistos pelo usuário, e capazes de promover a interação do usuário com o aplicativo. A interação do usuário com a interface modifica o estado atual do aplicativo. Se o usuário pressiona um botão, por exemplo, a ação resultante pode ser a exibição de uma caixa de diálogo. Os elementos visuais podem ser organizados no diagrama de tal forma, que a leitura do diagrama forneça uma idéia preliminar de como será a interface vista pelo usuário.

A interpretação do conteúdo do diagrama pode indicar quais são as telas do aplicativo, quais são os botões e menus que aparecem nessas telas, etc. Os elementos visuais, geralmente, estabelecem entre si uma relação de agregação. Os elementos que mais facilmente podem ser representados no diagrama, são as telas e o botões. A relação de agregação entre esses elementos pode ser especificada no diagrama. A tela que estabelece relação de agregação com um botão, é a tela onde ele será exibido.

Existem outros elementos do diagrama de robustez que podem ser apresentados na interface. As entidades podem ser apresentadas na tela, geralmente, como formulários. Um novo registro, por exemplo, pode ser inserido no banco de dados como resultado do preenchimento de um formulário exibido ao usuário. As entidades são caracterizadas por um conjunto de atributos, e um subconjunto desses atributos pode ser exibido na interface do usuário. Esses atributos definidos no diagrama podem ser rotulados por um conjunto de informações. Esse conjunto de informações contém, basicamente, a identificação da tela em que o atributo será mostrado e o tipo do elemento XUL que o descreve. A entidade funcionário, por exemplo, possui o atributo nome. Um formulário de alteração de dados, pode exibir o atributo nome como uma caixa de entrada de texto. Por outro lado, a tela de confirmação de alteração de dados pode apresentar o mesmo atributo como um texto fixo.

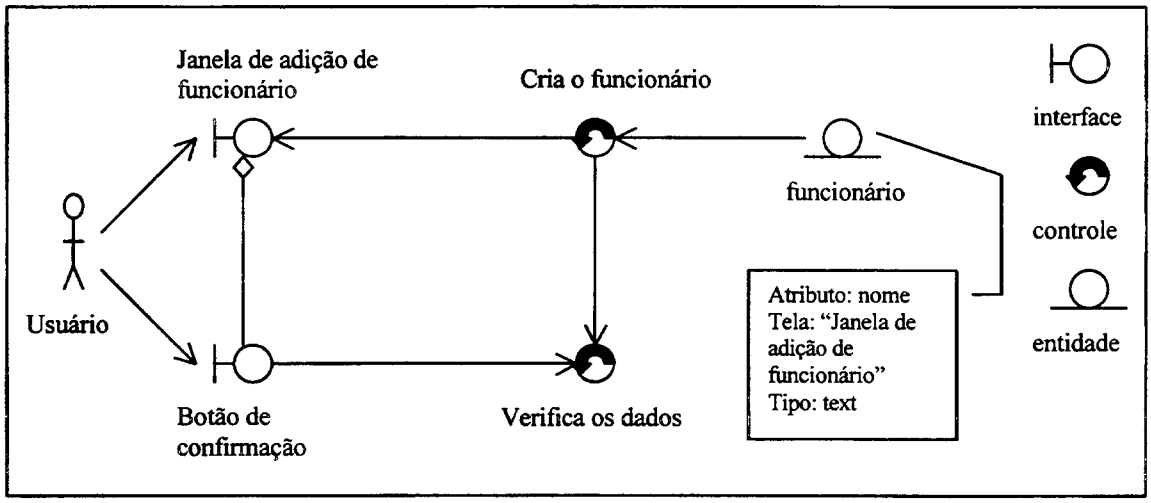

Figura 3 – diagrama de robustez

O diagrama de robustez que possua o nível de detalhamento especificado acima, pode fornecer uma descrição preliminar de como será a interface do aplicativo. As informações pertinentes a interface visual podem ser extraídas automaticamente do diagrama de robustez e organizadas em um novo diagrama - o digrama de telas. O diagrama de telas tem o objetivo de ampliar a descrição preliminar da interface, possibilitando que os elementos visuais sejam descritos mais especificamente com as características visuais desejadas pelo usuário.

O diagrama de telas, na verdade é o diagrama de objetos visto sob outro ponto de vista. Neste caso, o diagrama de objetos possui apenas os objetos de interface, dispostos da maneira como o usuário verá a interface. Esses objetos são acrescidos de um conjunto de especificações, que caracterizam a forma sob a qual esses elementos serão apresentados na interface. O diagrama de telas descreve a interface visual do aplicativo. Esse diagrama é composto por elementos que correspondem aos elementos XUL. A informação contida no diagrama de telas pode ser traduzida automaticamente para um documento XUL.

A geração automática do diagrama de telas a partir do diagrama de robustez é possível a partir da interpretação da relação entre os elementos contidos no diagrama de robustez. O diagrama de robustez é concebido usando as regras da UML. As classes contidas no diagrama são divididas em categorias, que podem ser reconhecidas através de estereótipos. As três principais categorias dizem respeito a classificação dos elementos em vista, controle e modelo.

A ferramenta de modelagem utilizada para testar os conceitos apresentados neste trabalho, foi o *ObjectDomain* [10]. Esta ferramenta é escrita em *Java* [11]. Seus diagramas expõem uma interface, que permite o acesso de seu conteúdo por um módulo externo a ferramenta. A comunicação dos módulos externos com a ferramenta é possível através de um console *JPython*, que está conectado à ferramenta. O módulo externo pode ser concebido na forma de um *script* Python, que é interpretado pelo *JPython*. O módulo Python pode acessar o conteúdo de qualquer objeto pertencente ao aplicativo, apenas explorando a interface pública desse objeto.

O módulo Python acessa o conteúdo do diagrama de robustez, coletando todas as informações necessárias a sua interpretação. Os estereótipos dos elementos são analisados, assim como a relação entre eles. As entidades encontradas são examinadas, com o propósito de identificar se algum de seus atributos será apresentado na interface visual. Essas informações são trazidas para uma estrutura interna ao módulo, que retrata o conteúdo do diagrama interpretado.

O módulo Python possui uma estrutura de objetos que corresponde aos elementos do diagrama do robustez. Essa estrutura armazena todas as informações extraídas do diagrama de robustez. A partir dela, é possível saber quais são os elementos de interface presentes no caso de uso, e qual é a relação entre eles. A geração automática do XUL a partir do diagrama de telas, assim como a geração desse diagrama a partir do diagrama de robustez, foram possíveis pelo uso do padrão *Projector*. O padrão *Projector* foi desenvolvido durante a implementação deste trabalho, a partir de experiências anteriores com padrões de projeto (*design patterns*) [13]. O objetivo do padrão *Projector* é expressar o mesmo conteúdo em diferentes linguagens. Possibilitando que o conteúdo expresso em uma linguagem seja extraído, e traduzido para outro tipo de linguagem.

A implementação do padrão *Projector* faz uso de alguns padrões de projeto, usando principalmente o padrão *Visitor*. A primeira etapa do *Projector* é possibilitar que o conteúdo de uma linguagem seja extraído para uma estrutura de objetos equivalente, e vice-versa. A estrutura de objetos, geralmente, é construída usando o padrão *Composite*, e mantém uma correspondência direta com a estrutura da linguagem. A segunda etapa consiste em desenvolver o mecanismo capaz de analisar o conteúdo da linguagem e criar os objetos correspondentes, assim como traduzir o conteúdo da estrutura de objetos em elementos da linguagem. Este mecanismo é implementado pelo padrão *Visitor*. Para tanto, é criado um *Visitor* que leva o conteúdo da linguagem para a estrutura de objetos. Esse *Visitor* percorre a estrutura da linguagem, analisa o conteúdo de cada elemento e cria o objeto correspondente. Analogamente, é desenvolvido outro *Visitor* que percorre a estrutura de objetos, refletindo o seu conteúdo na estrutura da linguagem.

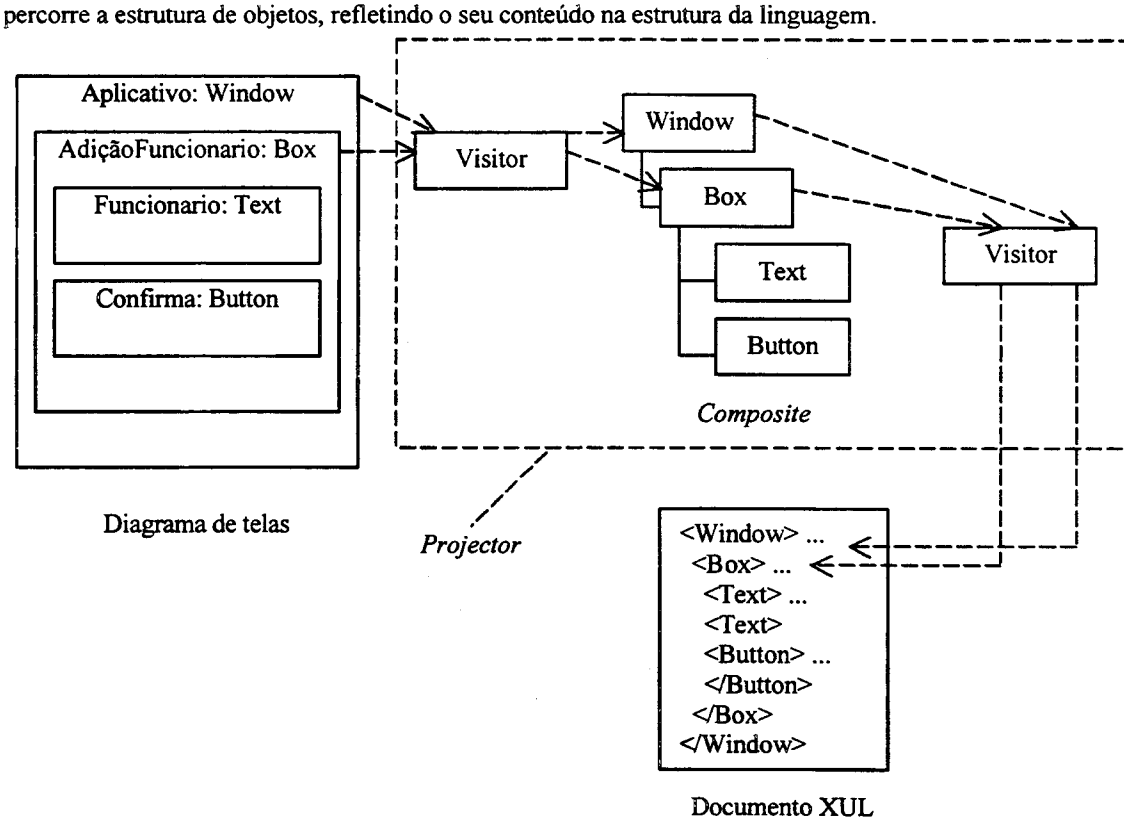

Figura 4 – Implementação do *Projector* na geração automática do XUL

O *Projector* utilizado neste sistema foi implementado em Python [24][25][26][27]. Este padrão provou ser uma ferramenta poderosa, pois ele foi reutilizado em duas etapas da geração automática. Um resultado importante, é que este padrão aplicado em dois pontos distintos do processo, garante uma coerência entre a intenção do projetista, capturada na abstração do modelo, e a especificação abstrata que representa a forma final da mesma idéia. O princípio fundamental que está por traz do *Projector*, é a preservação de uma mesma semântica que se encontra representada em diversas linguagens.

ARQUITETURA JAVABEAN CORRESPONDENTE A MODELAGEM XUL

No pacote *swing* do *Java* encontramos componentes gráficos capazes de produzir uma interface com precisão, mas dependendo do resultado pretendido, esta tarefa não é muito simples. Um outro problema se encontra na organização dos componentes na tela, que por vezes despende um certo trabalho devido aos diversos estilos de *layout* que o *swing* oferece.

Procurando abstrair ao máximo uma forma completa e flexível de descrever os componentes gráficos, a especificação XUL foi adaptada às necessidades de uma nova arquitetura que estava sendo gerada. O objetivo era obter uma descrição genérica de tal forma que fosse possível sua renderização nas diversas tecnologias existentes.

Assim como no *swing*, os componentes XUL são descritos sob uma determinada hierarquia. Por exemplo, alguns componentes são herdados de *box*, o que quer dizer que os atributos de *box* também servem para os seus derivados. O XUL também especifica *containers*, ou seja, é possível dispor itens dentro de outros itens.

O comportamento de um componente é um aspecto muito importante e não foi deixado de lado no XUL. Sua especificação contempla eventos, que estão presente em diversos componentes, como o *oncommand* e *onclick* no componente *button*.

Muitos dos componentes que encontramos no XUL tem uma correspondência muito forte com os originados do *swing*, não só pelo *layout* como também pelo tipo de dado que manipulam. Contudo, no XUL vamos encontrar componentes um tanto mais sofisticados que não tem sua correspondência no *swing*. Este é o caso do componente *tree*.

A *tag* [2]*tree* é o elemento raiz de uma série de *tags* que auxiliam na descrição dessa árvore, cuja habilidade é mostrar múltiplos itens num formato de tabela. Este componente é bastante similar a estrutura de uma tabela em HTML.

Uma outra vantagem do XUL é que não há necessidade de classes auxiliares para organizar os objetos na tela, como o *Layout Manager* do *swing*. Os próprios componentes possuem atributos que determinam sua posição relativa na tela.

Obter uma renderização em *swing* de um conjunto de componentes descritos em XUL não apresentou muita dificuldade devido a semelhança entre essas duas linguagens e a facilidade de renderização do *swing*. Porém, o mesmo não pode ser dito a respeito do HTML, que tem uma especial particularidade na forma como obtém e manipula os dados que disponibiliza na tela.

Uma aplicação HTML tem basicamente dois momentos. O primeiro onde se obtém uma descrição estática do que deve ser apresentado na tela, e o segundo quando é repassado ao servidor as alterações que foram feitas no ambiente descrito. A atribuição de valores não se dá de forma automática. Por este e outros contratempos surgiu a necessidade de se dispor de um conjunto de componentes autosuficientes que pudessem dar apoio a esse tipo de renderização.

Um *framework* de componentes, sob a tecnologia JavaBeans [17], foi desenvolvido especialmente para a renderização em HTML e a este foi dado o nome *twist*. Um componente *twist* tem maior semelhança com os componentes encontrados no XUL, mas renderizam HTML. Cada componente *twist* é responsável por fornecer o HTML necessário para a sua renderização, e por mais complicado que seja, isso fica transparente quando a interface gráfica é modelada.

Uma vez renderizados os objetos *twist* são inseridos na sessão (objeto que mantém o estado da interação do usuário com o aplicativo) e a partir daí toda a manipulação da informação que será apresentada na tela é feita através destes objetos. O acesso aos componentes *twist* será explanado com maior detalhe na próxima seção.

Dois aspectos são fundamentais neste trabalho. O primeiro visa a generalização da descrição da interface gráfica, tanto em relação a sua apresentação bem como ao seu comportamento. O objetivo é possibilitar a renderização de um sistema completo, independente da tecnologia utilizada na interface com o usuário do sistema. O segundo procura alcançar o desacoplamento seguindo o conceito vista-modelo já apresentado.

Esses conceitos foram concretizados numa arquitetura especial de renderização, orientada a objetos e desenvolvida segundo alguns padrões de projeto. Esta arquitetura é responsável por gerar os componentes abstratos a partir da

[2] Deve-se entender por tag uma produção delimitada que encerra um bloco de comando em linguagem tipo script.

descrição gerada com base no diagrama de robustez, associar os modelos correspondentes, e por fim , passar o controle para a última camada da interface gráfica, que se refere à tecnologia utilizada para renderização final.

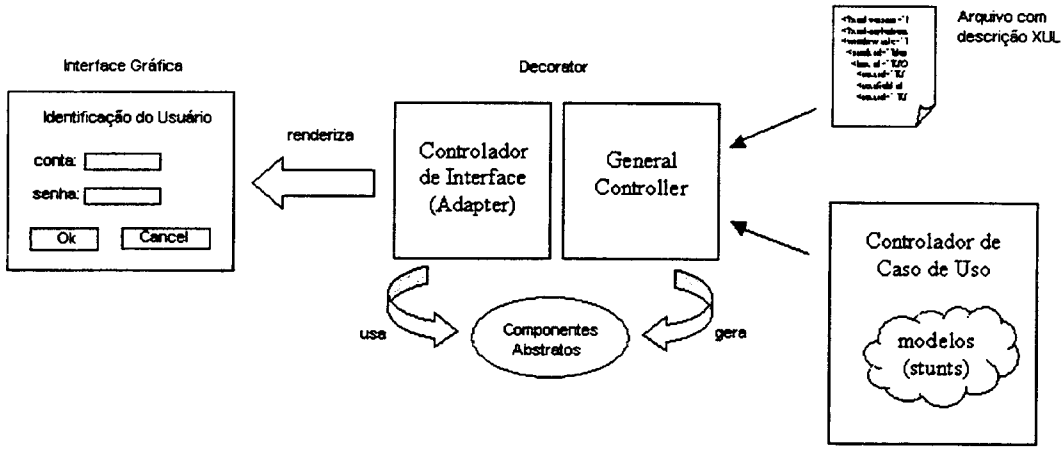

Figura 5 – Diagrama de renderização da interface gráfica.

Baseado na especificação do XUL foram desenvolvidos os componentes abstratos, que são classes que traduzem a descrição, provinda de um documento XUL, em objetos. Esses componentes mantém atributos e, por conseqüência, toda a flexibilidade encontrada no XUL. Esses componentes descrevem uma interface genérica, um padrão, a qual pode ser facilmente adaptada às renderizações existentes no *swing* do *Java* e no próprio HTML.

Os modelos utilizados no sistema, também gerados a partir do diagrama de robustez, são chamados *stunts*. Um *stunt* representa um item de um caso de uso que, na maioria das vezes, se apresenta na tela como atributo de uma classe de domínio. Todo caso de uso é gerenciado por um controlador que mantém referência para todos os *stunts* que participam do caso de uso. Os modelos são requisitados apenas quando precisam ser renderizados, o que acontece somente quando o caso de uso começa a ser executado.

Um arquivo contendo a descrição da interface em XUL é gerado a partir do diagrama de robustez. Foi estudada uma forma de associar essa descrição aos modelos do caso de uso. Esse é o papel dos controladores de interface que se acoplam como *Decorators*. O *Decorator* é o padrão de projeto que adiciona dinamicamente funcionalidade à outro objeto. O controlador mais externo, chamado *General Controller*, gera os componentes abstratos respeitando a hierarquia de composição do arquivo lido. À medida que os componentes abstratos são gerados, o controlador de interface associa o modelo correspondente. Esse modelo é obtido do controlador de caso de uso através de um identificador único.

Os demais controladores de interface dão a particularidade da interface gráfica. De forma que na renderização *twist*, por exemplo, um controlador *twist* funciona como o padrão de projeto *Adapter*, ou seja, para cada componente abstrato faz uma adaptação procurando um componente *twist* correspondente. Na verdade, os componentes são adaptados a uma renderização, de acordo com a tecnologia que está sendo utilizada. Os modelos são preservados e a mesma facilidade que os componentes abstratos possuíam para manipular dados é estendida aos componentes adaptadores de cada interface gráfica.

O fato de se trabalhar com uma descrição abstrata e completa é o que possibilita a renderização nos mais diversos ambientes. Esta arquitetura proporciona tanta flexibilidade a um sistema, que é possível renderizá-lo em uma nova interface apenas criando um novo controlador de interface, ou seja, um adaptador para este novo ambiente.

RENDERIZAÇÃO DO XUL EM HTML/JSP USANDO XUL

A tecnologia de renderização em HTML teve maior foco neste trabalho. Alguns foram os motivos que levaram a essa decisão, tais como a vasta utilização da interface HTML hoje em dia e o fácil acesso ao sistema por parte do usuário, entre outros. A proposta é partir de uma descrição da interface gráfica escrita em XUL e possibilitar a renderização na web.

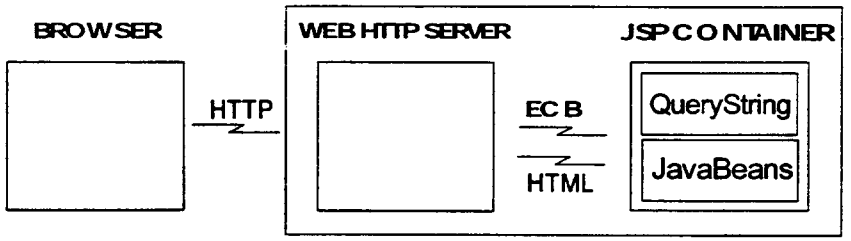

Figura 6 – Processamento de informações via web

Apesar dos documentos XUL conterem todos os dados necessários sobre um certo elemento visual, infelizmente, os browsers atuais, com exceção do *Mozilla*, não interpretam XUL. Por isso, é necessário usar a linguagem de formatação XSL [04], que define como deve ser tratado cada elemento presente no documento XUL.

A figura 6 mostra o processo de tratamento de dados em uma interação HTTP via web. Quando o usuário faz uma requisição HTTP para o browser, a página é submetida e o browser envia as informações contidas na página para o servidor. O servidor invoca o *container* que trata aquela informação. O *container* é um aplicativo que estende a funcionalidade do servidor de modo a interpretar linguagens como JSP[22], XSL ou *Java*. O *container* JSP concentra as informações recebidas num objeto chamado *QueryString* e invoca o interpretador XSL para processar os dados recebidos mediante uma página XUL indicada.

A transformação XSL adotada como solução tem como objetivo invocar componentes *Java* para tratar a informação recebida. Para tanto, o *script* XSL converte a descrição abstrata XUL em um *script* JSP. *Java Server Pages* são arquivos texto, com a extensão .jsp, que permitem misturar HTML estático com conteúdo dinâmico gerado por código servidor *Java*. Para cada *tag* encontrada no documento XUL, uma classe *Java* é instanciada. No caso, a classe *Java* é um JavaBeans, um componente encapsulado que encerra a semântica associada à *tag* XUL referenciada.

A transformação do XUL ocorre da seguinte forma: para cada *tag* no XUL existe um molde (*template)* XSL que trata aquele *tag* [19] e a transforma em HTML ou JSP. A parte HTML das páginas JSP são passadas para o cliente inalteradas. O *template* especifica como cada elemento deve ser visto, e também manipula informações originárias de documentos XUL, como por exemplo os atributos das *tags*. Por isso, costuma-se dizer que XSL não só trata da forma, mas também do conteúdo.

Os atributos encontrados na *tag* XUL são repassados para o componente *JavaBean*, que tem a responsabilidade de refletir essas informações em seu modelo. Para usar JavaBean com JSP [21], basta usar a *tag* *<useBean/>* para instanciar o componente, e as *tags* *<setProperty/>* e *<getProperty/>* para atribuir e acessar o valor de um atributo, respectivamente.

Uma *property* do componente é um atributo privado da classe que é modificado através de métodos públicos. As *properties* do componente devem ter dois métodos públicos: um para obter e outro para atribuir um valor da *property*. Um método *get* é um método que lê o valor da *property*. Um método *set* é um método que escreve o valor da *property*. A convenção para o nome dos métodos é: *public void setXxx (<type> arg), public <type> getXxx()*; onde *Xxx* é o nome da *property*.

Abaixo é apresentado um exemplo onde existe uma *tag text* no documento XUL com um atributo *id*. No caso desta *tag*, a correspondência é dada pela classe *TwTextField*, que representa um componente visual do *framework twist*. Uma instância desta classe é gerada e os atributos especificados no XUL são refletidos no modelo deste componente visual com o auxílio do JSP através da *tag* *<setProperty/>*.

Parte de código XUL:

```
<box orient="horizontal" border='0' id="ObtendoIdentificacaoStunt" flex="1">
        <text value="nome" id="UserNameLabel" flex="2"/>
        <textfield value="" id="UserName" flex="3"/>
</box>
```

Parte do código XSL que trata a *tag text*:

```
<xsl:template match="text">
    <jsp:useBean id="{@id}" class="pjb.twist.TwTextField" scope="session" />
    <jsp:getProperty name="{@id}" property="value"/>
</xsl:template>
```

Quando a *tag* *<jsp:setProperty name="nomeDoBean" property="id" value="{@id}"/>* é utilizada, o método *setId* é chamado passando *value* como parâmetro. O operador @ é usado quando um atributo do elemento XUL está sendo referenciado.

A *tag* *<usebean>* deve ser utilizada apenas para instanciar o objeto. Uma vez criado o objeto, a *tag* não pode ser mais utilizada para aquele objeto. Logo, é necessário ter cuidado ao escrever o XSL, pois os componentes referenciados pelo *<usebean>* devem ser instanciados com nomes diferentes.

As *tags set* e *get* podem ser usadas dentro da *tag* *<usebean/>*. Quando a *tag set* é utilizada dessa forma, o componente é inicializado com os valores dos atributos. Se o usuário requisita a página novamente, as *tags* que estão dentro da *tag* *<usebean/>* serão ignoradas. A *tag set* pode ser usada de duas formas: usando o atributo *value*, assim o valor especificado no atributo será atribuído à *property*; ou usando o atributo *param*, onde o valor especificado na *QueryString* é atribuído à *property*. O formato da *QueryString* é *pagina?nome=valor&nome2=valor2*. Quando o atributo *param* é utilizado, associa-se o valor do *param* a um nome na *QueryString* e o valor atribuído ao nome é retornado.

O resultado do processamento do XUL pelo XSL descrito acima é uma combinação de HTML estático e dinamicamente gerado pelo pós-processamento JSP. Por exemplo, o *box* é um elemento XUL que possui um atributo chamado orient. O atributo *orient* é tratado da seguinte forma: Se o *box* tem o atributo *orient* com valor igual a horizontal, quer dizer que os objetos que serão colocados dentro do *box* ficarão um ao lado do outro e se o *box* tem o atributo *orient* igual a vertical, quer dizer que os mesmos objetos ficarão um abaixo do outro.

Isso é tratado no HTML da seguinte forma: Sempre que um *box* é encontrado, uma tabela HTML é criada, se o *orient* é igual a vertical, só existirá uma célula por linha da tabela e se *orient* é horizontal a tabela só tem uma linha. Se o *box* não tem o atributo *orient*, é assumido como vertical. O trecho do código abaixo é o código XSL necessário para que a renderização possa ser feita.

```
<table>
   <xsl:choose>
      <xsl:when test="@orient='vertical'">
            <xsl:for-each select="*">
                  <tr><td>
                  <xsl:apply-templates select="."/>
                  </td></tr>
            </xsl:for-each>
      </xsl:when>
      <xsl:when test="@orient='horizontal'">
      <tr>
            <xsl:for-each select="*">
                  <td>
                  <xsl:apply-templates select="."/>
                  </td>
            </xsl:for-each>
      </tr>
      </xsl:when>
      <xsl:otherwise>
            <tr><td>
                  <xsl:apply-templates/>
            </td></tr>
      </xsl:otherwise>
   </xsl:choose>
 </table>
```

O *box* também possui um atributo chamado *flex*. O *flex* determina qual a porção da tela que deve ser destinada àquele *box*. Por exemplo, se existem dois boxes, um com *flex* igual a 2 e outro com *flex* igual a 3. O primeiro ocupará 2/5 da tela e o segundo 3/5.

Para cada *box*, é criada uma tabela, na qual o número de colunas é igual a soma dos *flexs* dos elementos que estão dentro do *box*. Cada elemento ocupará o número de colunas igual ao *flex*. Supondo que só existe um *box* e que ele contém os boxes acima. Uma tabela é criada com cinco colunas, duas para o primeiro *box* e três para o segundo.

Exemplificamos através de uma aplicação *web*, todo o mecanismo apresentado, onde a partir de uma informação abstrata sobre a interface gráfica e, através da geração de modelos, são criados elementos visuais concretos para esta abstração.

CONCLUSÃO

O XUL pode ser uma resposta para a falta de linguagens de descrição da interface com o usuário. A adoção do XUL por um navegador internet popular é um passo importante para torná-lo um padrão de fato. A reengenharia proposta neste artigo generaliza o escopo de sua aplicação em conformidade com a prática de engenharia de sistemas vigente. A construção de uma arquitetura baseada no paradigma MVC é a base para sintonizar o XUL com o estilo de projeto usado em sistemas orientado a objetos. A abstração obtida desta maneira permite que o XUL seja uma linguagem ponte entre a modelagem expressa em UML e a forma final da interface com o usuário.

O artigo exemplifica uma utilização prática desta abordagem descrevendo um sistema que efetivamente transporta consistentemente a abstração do modelo para a implementação. Um resultado importante deste trabalho é a identificação do padrão de projeto *Projector* que captura a mecânica de transcrição entre representações descritas em linguagens distintas. O reuso deste padrão em transformações sucessivas garantiu a consistência entre as diversas representações, ao mesmo tempo que demonstrou o poder da linguagem *Python* como facilitadora de interações entre linguagens distintas como UML, XUL e *Java*.

O outro resultado importante foi a prova de conceito do princípio de abstração da interface com o usuário. O sistema construído foi capaz de gerar um aplicativo em sua forma final a partir da descrição abstrata. O aplicativo final, ao contrário da idéia original do XUL, independe da adição de *scripts* ou outra lógica extra para a troca de dados entre interface e modelo.

O *script Python* trata a descrição abstrata em XUL, gerando todo o código em HTML e *Java* segundo a arquitetura ativa do MVC que garante o transporte automático de dados. O artigo detalha apenas a geração de interface em HTML e JSP, mas um transcritor para interface *Swing* também foi construído. Um novo aplicativo foi gerado para esta representação a partir da mesma abstração demonstrando a validade da generalização proposta. Outros transcritores estão sendo desenvolvidos para interfaces applet (AWT) e XML [02][28]. O produto final está sendo consolidado em torno de uma ferramenta de geração automática de sistemas de informação.

A abstração da interface com o usuário provou ser um conceito poderoso, capaz de economizar significativamente tempo de implementação. O aplicativo final apresenta uma maior qualidade de software, correto por construção e com grande adequação ao requisitos originais, garantida pela abstração da interface e pelo processo padronizado de geração.

BIBLIOGRAFIA

[01] XUL Programmer's Reference Manual – http://www.mozilla.org/xpfe/xulref/XUL_Reference.HTML
[02] McLaughlin, B.; "*Java* and XML", O'Reilly, 2000.
[03] Hunter, J. e McLaughlin,B.; "Easy *Java*/XML integration with JDOM", JavaWorld - http://www.javaworl.com/javaworld/jw-05-2000/jw-0518-jdom.HTML
[04] (xsl) Harold, E. R.; "XML Bible"; IDG Books Worldwide; 1999.
[05] Rosemberg, D.; Scott, K.; "Use Case Driven Object Modeling With UML: A Pratical Approach"; Addison-Wesley; 1999.
[06] Lutz, M.; Ascher, D.; "Learning *Python*"; O'Reilly; 1999.
[07] http://www.jpython.com
[08] http://www.jython.com
[09] http://www.python.org
[10] http://www.objectdomain.com
[11] Eckel, B.; "Thinking in *Java*"; Prentice Hall PTR; 1998.
[12] Robinson, M.; Vorobiev, P.; "*Swing*"; Manning Publications Co.; 1999.
[13] Gamma, E.; Helm, R.; Johnson, R.; Vlissides, J.; "Design Patterns – Elements of Reusable Object-Orient Software"; Addison-Wesley; 1998.
[14] http://www.mozilla.org
[15] Avedal, K.; Ayers, D.; Briggs, T.; etc; "Professional JSP"; Wrox Press Ltd; 2000.
[16] Hall, M.; "Core Servlets and JavaServer Pages"; Prentice Hall PTR; 2000.
[17] Doherty, Dr. D.; Leinecker, R.; "JavaBeans Unsleashed"; Sams Publishing; 1999.
[18]Adamovic, M.; "Process JSPs effectively with JavaBeans"; JavaWorld; 2001; http://www.javaworld.com/javaworld/jw-01-2001/jw-0119-jspframe.html.
[19]Santiago, S.; "Combine the power of XPath and JSP *tag* libraries"; JavaWorld; 2001; http://www.javaworld.com/javaworld/jw-01-2001/jw-0126-xpath.html

[20] Dick, Timothy; "JavaServer Pages"; ZDNet;
http://www.zdnet.com/products/stories/reviews/0,4161,2659991,00.html
[21] Eden, T.; "JavaPro - Using JavaBeans with JSP"; Ed. Lucke.
[22]JJKuslich; "Introduction to JavaServerPages"; Developer
[23]Benoît Marchal; Xalan: Extending XSLT with *Java*; EarthWeb; 2000 - http://www.earthweb.com/dlink.resource-jhtml.72.1061.|repository||softwaredev|content|article|2000|08|03|SDMarchalXSLT|SDMarchalXSLT~xml.0.jhtml?cda=true
[24]Richard Baldwin; Learn to Program using Python: Nesting, Sorting, Deleting, and Membership Testing Dictionary Elements; Developer; 2001 - http://developer.earthweb.com/dlink.resource-jhtml.72.1063.|repository||softwaredev|content|article|2001|01|16|SDbaldwindictel|SDbaldwindictel~xml.0.jhtml?cda=true
[25]Richard Baldwin; Learn to Program using Python: Valid Keys, Key Lists, Iteration; Developer; 2001 - http://developer.earthweb.com/dlink.resource-jhtml.72.1063.|repository||softwaredev|content|article|2000|12|12|SDbaldwinkey|SDbaldwinkey~xml.0.jhtml?cda=true
[26]Richard Baldwin; Learn to Program using Python: Using Lists Part 2; Developer; 2000 - http://developer.earthweb.com/dlink.resource-jhtml.72.1063.|repository||softwaredev|content|article|2000|08|07|SDBaldwinlists2|SDBaldwinlists2~xml.0.jhtml?cda=true
[27]Richard Baldwin; Learn to Program using Python: Variables and Identifiers; Developer; 2000 - http://developer.earthweb.com/dlink.resource-jhtml.72.1063.|repository||softwaredev|content|article|2000|06|20|BaldwinPyth08|BaldwinPyth08~xml.0.jhtml?cda=true
[28]R. Allen Wyke; Is XML changing the future of Web publishing?; Unix Insider; 1999 - http://www.unixinsider.com/swol-09-1999/swol-09-webmaster.html
[29]Bryan Formidoni, Danielle Anthony ; Transforming XML; Unix Insider - http://www.unixinsider.com/swol-10-2000/swol-1027-webmaster-af.html; 2001
[30] Cascating Style Sheet - http://developer.netscape.com/tech/css/
[31] W3C DOM - http://developer.netscape.com/tech/dom/