UNIVERSIDADE REGIONAL DE BLUMENAU

CENTRO DE CIÊNCIAS EXATAS E NATURAIS

CURSO DE SISTEMAS DE INFORMAÇÃO – BACHARELADO

FERRAMENTA PARA CONVERSÃO DE FORMULÁRIOS DELPHI EM PÁGINAS HTML

ARIANA SOUZA

BLUMENAU
2005

2005/2-03

ARIANA SOUZA

FERRAMENTA PARA CONVERSÃO DE FORMULÁRIOS DELPHI EM PÁGINAS HTML

Trabalho de Conclusão de Curso submetido à Universidade Regional de Blumenau para a obtenção dos créditos na disciplina Trabalho de Conclusão de Curso II do curso de Sistemas de Informação — Bacharelado.

Prof. Joyce Martins, Mestre – Orientadora

BLUMENAU
2005

2005/2-03

FERRAMENTA PARA CONVERSÃO DE FORMULÁRIOS DELPHI EM PÁGINAS HTML

Por

ARIANA SOUZA

Trabalho aprovado para obtenção dos créditos na disciplina de Trabalho de Conclusão de Curso II, pela banca examinadora formada por:

Presidente:	Profª. Joyce Martins, Mestre – Orientadora, FURB
Membro:	Prof. Jomi Fred Hübner, Doutor – FURB
Membro:	Prof. Luiz Bianchi, Mestre – FURB

Blumenau, 12 de dezembro de 2005

Dedico este trabalho a todos os amigos, especialmente aqueles que me ajudaram diretamente na realização deste. A todas as pessoas próximas, que souberam compreender as horas alocadas no desenvolvimento do mesmo. Aos meus pais que sempre me apoiaram e incentivaram a estudar e lutar pelos meus objetivos.

AGRADECIMENTOS

À Deus, pelo seu imenso amor e graça.

À minha família, que sempre esteve presente e dando força.

Aos meus amigos, em especial Maurício Kiniz, pelo auxilio nessa caminhada.

Ao professor, Mauro Marcelo Mattos, que propôs essa idéia como meu trabalho de conclusão de curso.

A minha orientadora, Joyce Martins, pela ótima orientação e por ter acreditado na conclusão deste trabalho.

A morte do homem começa no instante em que ele desiste de aprender.

Albino Teixeira

RESUMO

Neste trabalho é apresentado o desenvolvimento da ferramenta DelphiToWeb que auxilia no desenvolvimento de aplicações web, a partir da conversão da interface de um sistema implementado em Delphi. A ferramenta converte componentes de interface Delphi em componentes equivalentes em HTML, arquivos .LZX e arquivos XML. DelphiToWeb é um gerador de código sendo que o modelo de geração de código implementado foi baseado no modelo de geração de camada de aplicação. Tem-se que a entrada do gerador é um arquivo com extensão .DFM, contendo toda a informação necessária para construir código com as funcionalidades para a camada de interface de uma aplicação web. A análise do arquivo .DFM foi feita utilizando analisadores de linguagens (léxico, sintático e semântico), sendo que para gerar as páginas HTML foram utilizados HTML e JavaScript, e para gerar os arquivos .LZX foi usada a linguagem LZX do Laszlo.

Palavras-chave: Geração de código. Aplicações web. Arquivos .DFM. Páginas HTML. Laszlo.

ABSTRACT

Is this work the DelphiToWeb tool is introduced. It assists web applications development, from the interface conversion of Delphi implemented software. The tool converts components of Delphi interface to equivalent components in HTML, LZX and XML files. DelphiToWeb is a code generator, meaning that the implemented code generator model was based on the generator model of application layer. The input generator is a DFM file, having the necessary information to construct code with the functionalities for the interface layer of a Web application. The DFM file analysis was done using language analyzers (lexicon, syntactic and semantic); to generate the HTML pages, HTML and JavaScript were used and to generate the LZX file, the LXZ language by Laszlo was used.

Keywords: Code generator. Web applications. DFM files. HTML pages. Laszlo.

LISTA DE ILUSTRAÇÕES

LISTA DE SIGLAS

API – *Application Program Interface*

BNF – *Backus-Naur Form*

CSS – *Cascading Style Sheets*

FURB – Universidade Regional de Blumenau

HTML – *HyperText Markup Language*

IEBNF – *Indented Extended Backus Naur Form*

JLCA – *Java Language Conversion Assistant*

J2EE – *Java 2 Enterprise Edition*

MFC – *Microsoft Foundation Classes*

RF – Requisito funcional

RNF – Requisito não funcional

SQL – *Structured Query Language*

UML – *Unified Modeling Language*

URL – *Uniform Resource Locator*

XML – *eXtensible Markup Language*

SUMÁRIO

1 INTRODUÇÃO

Novas possibilidades para a implantação de serviços computacionais estão surgindo. A web é atualmente o principal veículo para a prestação destes serviços, permitindo atingir um número cada vez maior e mais diversificado de usuários (LEITE, 2002).

Nesse sentido, empresas e indústrias têm demanda por sistemas para a Internet e, consequentemente, necessidade de profissionais capacitados que atuam na área de desenvolvimento de sistemas com conhecimentos sobre as novas tecnologias de desenvolvimento para web (UNIVERSIDADE ESTADUAL DE MARINGÁ, 2005).

Embora essa demanda seja urgente, as empresas ainda possuem várias aplicações que foram desenvolvidas para *desktop*, muitas delas utilizando Delphi. Manter os softwares antigos é problemático, pois estes não conversam com novos sistemas, não podem aproveitar a infra-estrutura de comunicação da web e os desenvolvedores ou já não estão na empresa ou se encontram cada vez mais "escassos" no mercado. Por outro lado, reescrever esses sistemas exige que todo o investimento original seja refeito, tomando muito tempo e, em geral, com custo muito alto na maioria dos orçamentos das empresas brasileiras (SOARES FILHO, 2003).

A nova geração de softwares de migração faz com que não seja preciso recomeçar toda a plataforma do zero, diminuindo o custo de desenvolvimento, pois o custo da migração, em geral, representa 15% a 20% do custo original de desenvolvimento. Além disso, o processo de conversão exige cerca de 25% do tempo de desenvolvimento do sistema original e é por isso que a alternativa de migração pode representar economias de tempo e dinheiro tão significativas (SOARES FILHO, 2003).

Considerando essa demanda por aplicações web e levando em conta a quantidade de sistemas para *desktop* desenvolvidos em Delphi, propõe-se o desenvolvimento de uma

ferramenta para converter os formulários Delphi em páginas *HyperText Markup Language* (HTML). Salienta-se que não foi identificado outro aplicativo com essa funcionalidade.

1.1 OBJETIVOS DO TRABALHO

O objetivo deste trabalho é construir uma ferramenta que permita a conversão de componentes de interface do Delphi em componentes equivalentes em páginas HTML com JavaScript.

Os objetivos específicos do trabalho são:

a) identificar o subconjunto de componentes de interface do Delphi que possua *layout* equivalente em HTML e JavaScript;
b) converter os componentes de interface de um formulário Delphi, gerando uma página HTML equivalente;
c) avaliar o grau de compatibilidade entre o *layout* da interface Delphi e da página gerada.

1.2 ESTRUTURA DO TRABALHO

O trabalho está organizado em quatro capítulos. O segundo capítulo contempla a fundamentação teórica, onde são apresentados conceitos e características sobre geração de código, analisadores de linguagens e arquivos .DFM, entre outros.

No capítulo 3 são descritos os requisitos, a especificação e a implementação da ferramenta DelphiToWeb, detalhando a funcionalidade do sistema e como foi feita a conversão dos arquivos .DFM. Por fim, são apresentados os resultados obtidos.

No último capítulo são apresentadas a conclusão e sugestões para trabalhos futuros.

2 FUNDAMENTAÇÃO TEÓRICA

Este capítulo apresentada os aspectos teóricos relacionados ao trabalho. Trata de geração de código e dos modelos de geradores de código existentes, bem como das etapas para o desenvolvimento de um gerador. Descreve os módulos de um compilador, mais especificamente os analisadores léxico, sintático e semântico, que podem processar as entradas e as saídas de um gerador de código. Também apresenta a estrutura de um arquivo .DFM, entrada da ferramenta desenvolvida, assim como do HTML, do Laszlo e do *eXtensible Markup Language* (XML), que são as saídas do conversor. Na última seção são apresentados alguns trabalhos correlatos.

2.1 GERAÇÃO DE CÓDIGO

Geração de código é uma técnica que usa programas para gerar programas. Podem ser desde simples formatadores de código até ferramentas que geram aplicações complexas a partir de modelos abstratos. Hoje em dia é comum o desenvolvimento de aplicações complexas, usando *Java 2 Enterprise Edition* (J2EE), Microsoft .NET ou *Microsoft Foundation Classes* (MFC). Quanto mais complexa for a estrutura da aplicação, mais conveniente será o uso da geração automática de código (HERRINGTON, 2003).

Segundo Herrington (2003, p. 15), a geração de código fornece benefícios à Engenharia de Software em vários níveis. Alguns desses benefícios são:

a) qualidade: código gerado automaticamente tende a ter qualidade uniforme, diretamente relacionada com a qualidade dos modelos abstratos usados. Já código escrito à mão tende a ter qualidade não uniforme visto que, à medida que a aplicação vai sendo desenvolvida, novas e melhores abordagens podem ser

propostas;

b) consistência: o código que é construído por um gerador tende a ser consistente e mais fácil de ser compreendido e usado. Além disso, para incluir novas funcionalidades ou corrigir erros, é necessário alterar os modelos e executar o gerador, sendo que as alterações são aplicadas consistentemente em todo o código;

c) abstração: gerar código a partir da definição da lógica da aplicação, usando um modelo abstrato, permite que o modelo especificado seja revisto e validado mais facilmente, já que é menor e mais específico do que o código resultante. E ainda, o gerador pode ser reprojetado para traduzir a aplicação para outras linguagens, tecnologias ou plataformas;

d) produtividade: o cronograma de desenvolvimento de um projeto usando geração de código é significativamente diferente de um projeto codificado manualmente. Na programação manual, quando são feitas suposições errôneas sobre o uso de estruturas ou são tomadas decisões inadequadas, pode ser necessário reescrever todo o código. Com a geração automática de código, os programadores devem reescrever os modelos abstratos e executar o gerador. Além disso, com o uso de geração de código tem-se mais tempo para fazer projetos adequados, validar as tecnologias usadas e testar a aplicação.

A geração do código é uma ferramenta extremamente valiosa que pode ter impacto na produtividade e na qualidade de projetos. Compreender o funcionamento das ferramentas usadas no processo de desenvolvimento de software pode facilitar o uso de técnicas de geração do código (HERRINGTON, 2003, p. 27).

2.1.1 Modelos de geradores de código

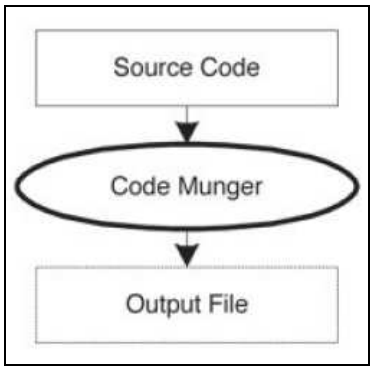
Herrington (2003) classifica as ferramentas para geração de código em cinco modelos, dependendo do uso, das entradas e das saídas. As entradas podem ser tanto representações abstratas quanto um código em alguma linguagem de programação. As saídas podem ser uma extensão do código de entrada, uma implementação parcial ou uma implementação completa. Os cinco tipos de geradores de código são descritos nas seções seguintes.

2.1.1.1 Formatação de código

Um formatador de código (ou *code munger*) é o mais simples dos modelos de geradores do código. Tem como entrada código fonte, geralmente escrito em uma linguagem de alto nível (por exemplo, C, Java, C++), e como saída um ou mais arquivos, que podem ser desde documentação HTML até uma extensão do código de entrada. Por mais simples que seja, a formatação de código pode ser usada para resolver diversos problemas. Cita-se como exemplo:

a) criar documentação HTML a partir de comentários “especiais” presentes no código fonte;
b) catalogar literais com objetivo de internacionalização;
c) catalogar identificadores no código fonte (declaração e uso);
d) criar índices de classes e métodos ou módulos (funções e procedimentos).

Um formatador de código analisa o arquivo de entrada, procurando por padrões. Uma vez que os padrões são encontrados, cria um ou mais arquivos de saída. A Figura 1 mostra o fluxo de entrada e saída desse modelo.



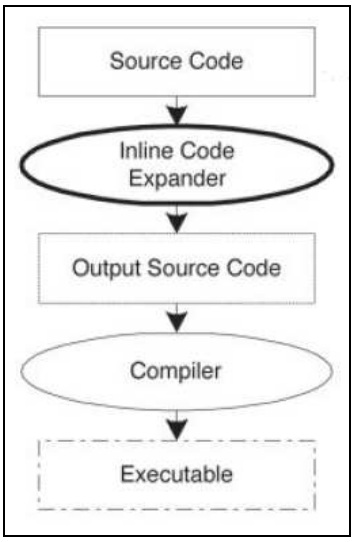
Fonte: Herrington (2003, p. 62)
Figura 1 – Fluxo de entrada e saída de um formatador de código

2.1.1.2 Expansão de código

O uso da expansão de código (ou *inline-code expander*) permite simplificar o código fonte. Para tanto, através do uso de sintaxe especializada, especifica-se os requisitos do código a ser gerado. A entrada do gerador é um programa escrito em uma linguagem de alto nível anotado com algum tipo de marcação que será trocada por código durante a execução do gerador. A saída do gerador é um código na mesma linguagem do código de entrada, que pode ser compilado e usado. Na expansão de código tem-se que:

a) é necessário estender uma linguagem de programação de alto nível para permitir a inclusão de marcações. Assim, o código de entrada não pode ser compilado antes da execução do gerador;
b) as marcações são removidas do código de saída.

A Figura 2 ilustra o fluxo de entrada e saída para o modelo de expansão de código.



Fonte: Herrington (2003, p. 79)

Figura 2 – Fluxo de entrada e saída de um gerador de expansão de código

Em geral, a expansão de código é usada para criar código em linguagens de programação de alto nível (C, Java) para gerenciar consultas a banco de dados, a partir da expansão dos comandos *Structured Query Language* (SQL) embutido no código fonte. Pode também criar código para equações matemáticas ou código para testes a partir de dados especificados em marcações especiais.

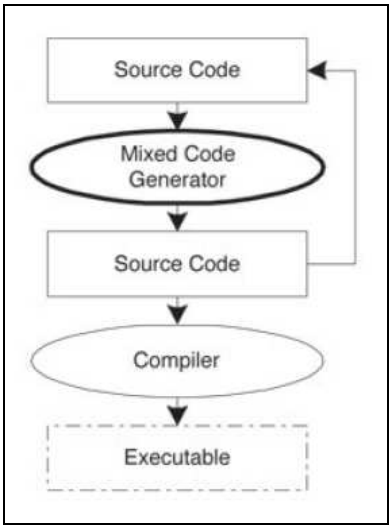
2.1.1.3 Geração mista

O gerador de código misto (ou *mixed-code generator*) é uma implementação mais prática do que o modelo de expansão de código. Na geração de código misto tem-se que:

a) as marcações são feitas usando comentários, evitando a extensão de uma linguagem de programação de alto nível;
b) os comentários são mantidos na saída do gerador e o código que implementa os requisitos especificados nos comentários localiza-se entre os mesmos, possibilitando que quando o gerador for executado novamente, o código gerado seja removido e/ou atualizado;
c) o código gerado fica localizado próximo à especificação (comentários), de tal

forma que se pode verificar uma correspondência entre o que foi especificado e como é implementado.

Além disso, a diferença principal entre os dois modelos é o fluxo de entrada e saída. No modelo de geração mista o arquivo de saída serve como entrada para o gerador, de tal forma que tanto o código de entrada quanto o código de saída podem ser diretamente compilados. A Figura 3 mostra o esquema de entrada e saída para a geração mista.



Fonte: Herrington (2003, p. 84)

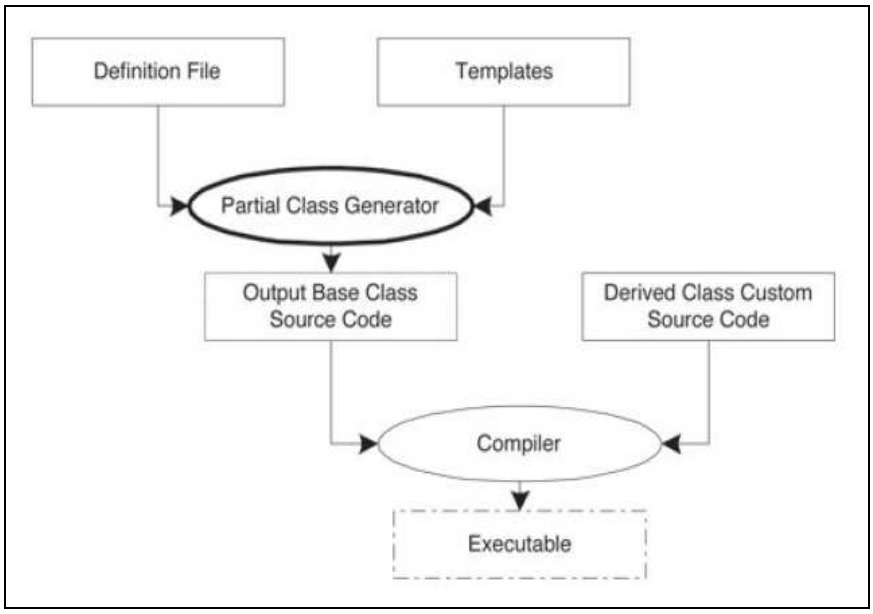
Figura 3 – Fluxo de entrada e saída para um gerador de código misto

Os usos potenciais para a geração mista são similares àqueles para a expansão de código.

2.1.1.4 Geração parcial de classes

O modelo de geração parcial de classes (ou *partial-class generator*) é o primeiro dos tipos de geradores de código que constrói código a partir de um modelo abstrato. Os geradores anteriores têm como entrada código fonte em linguagem de alto nível (por exemplo, C, Java, C++, SQL). Em vez de filtrar ou de substituir fragmentos do código, o gerador analisa uma descrição abstrata dos requisitos do código a ser criado e constrói um conjunto de classes, que deve ser estendido com a implementação de subclasses (classes derivadas). As

classes geradas e as classes derivadas são requeridas para criar uma aplicação completa. A Figura 4 mostra as entradas (definição abstrata e modelos da estrutura básica da classe) e a saída de um gerador parcial de classes.



Fonte: Herrington (2003, p. 84)

Figura 4 – Um gerador parcial de classes usando um arquivo de definição como entrada

Um gerador parcial de classes pode ser usado para:

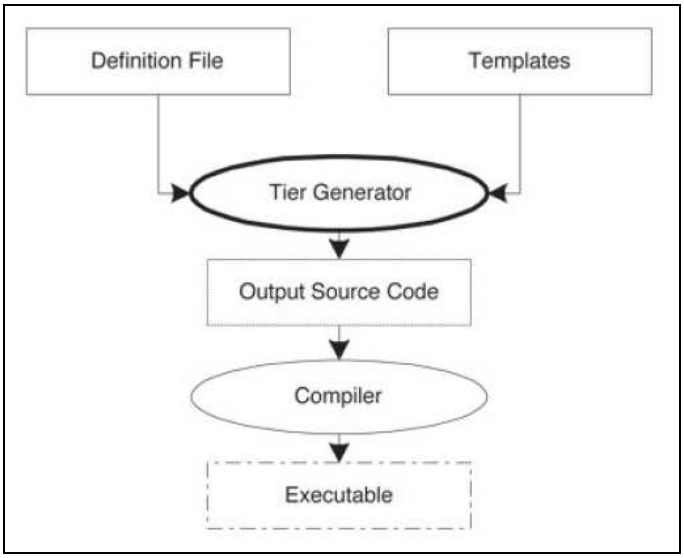
a) construir classes básicas para acesso a base de dados. As subclasses implementam o comportamento customizado;
b) gerar classes para interfaces com usuário;
c) gerar classes para armazenamento estruturado de dados, baseado em um conjunto definido de campos.

Herrington (2003) destaca que nunca devem ser criadas instâncias das classes geradas: deve-se instanciar as classes derivadas. Isso permite "migrar" do gerador parcial de classes para um gerador de camadas da aplicação.

2.1.1.5 Geração de camadas da aplicação

Um gerador de camadas da aplicação (ou *tier generator*) constrói todo o código para

uma camada de aplicação. A entrada do gerador é uma definição abstrata dos requisitos que contém toda a informação necessária para construir o código completo de um nível da aplicação. A saída do gerador, construída usando um conjunto de *templates*, é o código com todas as funcionalidades (classes e métodos). O código gerado pode ser diretamente integrado ao projeto que está sendo desenvolvido. A Figura 5 apresenta as entradas e a saída de um gerador desse modelo.



Fonte: Herrington (2003, p. 84)

Figura 5 – Entrada e saída de um gerador de camada da aplicação

Os exemplos são similares aos dos geradores parciais de classe, sendo mais comum as ferramentas que constroem a camada para acesso a base de dados de aplicações cliente/servidor.

A diferença entre esse modelo e o modelo anterior é que a saída de um gerador parcial de classes deve ser usada junto com as classes derivadas. Isto é, tanto as classes geradas quanto as classes derivadas são necessárias para criar uma aplicação com funcionalidade. Já o gerador de camadas da aplicação não necessita de nenhuma classe derivada porque gera e mantém o código para uma camada completa.

2.1.2 Etapas de desenvolvimento de um gerador

Para desenvolver um gerador de código, Herrington (2003) recomenda que as etapas abaixo sejam seguidas:

a) construir manualmente o código de saída: deve-se determinar o que e como deve ser a saída do gerador. Isso facilita a identificação do que é necessário extrair dos arquivos de entrada;

b) projetar o gerador: deve-se determinar qual o formato do arquivo de entrada, qual o tipo da entrada (um arquivo ou um conjunto de arquivos lidos de um diretório), como a entrada será analisada (com expressões regulares ou com analisadores léxicos e sintáticos) e como o código de saída será criado;

c) desenvolver a análise da entrada: esta é a primeira etapa da codificação, onde será implementado o código para ler o arquivo de entrada e extrair as informações necessárias para construir a saída;

d) desenvolver a geração da saída: nesta etapa é implementado o processamento que faz análise dos dados extraídos da entrada gerando código conforme especificado na primeira etapa. O código de saída deve ser armazenado em arquivos apropriados.

Existem algumas regras que podem ser seguidas no desenvolvimento de geradores:

a) respeitar o código escrito à mão: mesmo com o uso de geradores de código pode ser necessário codificar manualmente os casos especiais. No entanto, o tempo das equipes de desenvolvimento é extremamente valioso e deve-se evitar desperdiçá-lo em tarefas repetitivas. Portanto, deve-se sempre que possível desenvolver ou usar geradores para automatizar essas tarefas;

b) escrever a saída do gerador manualmente: deve ser inteiramente compreendida a

estrutura da aplicação antes de desenvolver o gerador de código, como descrito no item “a” das etapas de desenvolvimento;

c) escolher a linguagem de implementação apropriada: as ferramentas usadas para construir o gerador não têm que ser as mesmas ferramentas usadas para escrever a aplicação. O problema que o gerador está tentando resolver é completamente diferente do problema que está sendo resolvido pela aplicação. Por essa razão, o gerador deve ser considerado como um projeto independente;

d) incluir avisos: o gerador deve sempre colocar avisos indicando o código que foi gerado automaticamente;

e) fazer o gerador de código amigável: o gerador deve dizer ao programador o que fez e quais arquivos alterou ou criou. Deve também emitir mensagens de erro adequadas. Uma ferramenta que seja difícil de ser usada, pode acabar sendo ignorada e o trabalho para desenvolvê-la será em vão.

2.2 ANALISADORES DE LINGUAGENS

Segundo Herrington (2003, p. 49), geradores de código processam arquivos contendo texto escrito em alguma linguagem. Pode ser linguagens de programação com construções variadas como comentários, classes e métodos, ou arquivos com modelos abstratos. Para análise de comentários podem ser utilizados padrões definidos através de expressões regulares. Para construções mais complexas é necessário usar analisadores. Os analisadores (léxico, sintático e semântico) são módulos que compõem a estrutura básica de um compilador.

Price e Toscani (2001) afirmam que compiladores são programas bastante complexos que convertem linguagens de fácil escrita e leitura (linguagem de alto nível), em linguagens

que possam ser interpretadas ou executadas pelas máquinas. Um compilador está configurado para seguir regras determinadas pela linguagem para a qual foi programado para analisar. Caso seja identificado um erro, o compilador pára o processo e emite uma mensagem indicando o melhor caminho para a correção do programa. Apenas programas corretos podem ser traduzidos para a linguagem-alvo (CHAGAS, 2003).

Além das fases de análise, compiladores possuem uma fase de síntese constituída por geração de código intermediário, otimização de código e geração de código objeto. Esses três últimos módulos não serão detalhados uma vez que não estão no escopo desse trabalho.

2.2.1 Análise léxica

O objetivo desta fase é, de acordo com Price e Toscani (2001), ler o programa fonte e transformar seqüências de caracteres que constituem unidades léxicas (*tokens*) em uma representação interna composta por três informações (classe, valor e posição). A classe representa o tipo do *token* reconhecido, podendo ser identificador, constante numérica, cadeia de caracteres, palavra reservada, operadores ou separadores, enfim, qualquer parte do código fonte que faça algum sentido. O valor do *token* depende da classe. Por exemplo, para *tokens* da classe constante inteira, o valor pode ser a seqüência de dígitos representando o número inteiro. No caso de *tokens* da classe identificador, o valor é a seqüência de caracteres que representa o identificador. Por fim, a posição do *token* indica o local (linha e coluna) no programa fonte onde o mesmo se encontra e é utilizada principalmente na localização dos erros detectados.

O Quadro 1 apresenta uma seqüência de caracteres cujas unidades léxicas reconhecidas são: (`identificador, **width**, linha 1`); (`símbolo especial, **=**, linha 1`) e (`constante inteira, **169**, linha 1`).

```
001          width = 169
```

Quadro 1 – Propriedade de um componente de interface Delphi

Para seqüências de caracteres que não estejam de acordo com a definição da linguagem, erros léxicos são gerados. Na especificação dos *tokens* são usadas expressões regulares.

2.2.2 Análise sintática

Análise sintática tem por função verificar se a estrutura gramatical do programa está correta, identificando as seqüências de *tokens* que compõem as construções da linguagem. Os itens léxicos são fornecidos ao analisador sintático que os agrupa em uma estrutura denominada de árvore sintática ou árvore de derivação. Outra função da análise sintática é a identificação de erros de sintaxe que indicam a posição e tipo de erro ocorrido. Segundo Price e Toscani (2001, p. 10), para a especificação da estrutura gramatical pode ser usada a notação *Backus-Naur Form* (BNF).

2.2.3 Análise semântica

A análise semântica analisa a árvore sintática determinando o sentido ou significado do programa. Uma das tarefas mais importantes é a análise de contexto. É na analise semântica que são detectados, por exemplo, a ocorrência da declaração e do uso de identificadores; os conflitos entre tipos; a ausência de declarações de variáveis, funções e procedimentos (CHAGAS, 2003).

2.3 ARQUIVO .DFM

Para cada formulário de uma aplicação desenvolvida em Delphi é gerado um arquivo .DFM. Esse arquivo contém todas as informações dos componentes da interface. Assim, para cada componente, tem-se uma identificação, uma lista de propriedades com valores associados e outros possíveis objetos componentes (SASSE, 2005). O Quadro 2 a seguir mostra a estrutura de um arquivo .DFM com um *TButton* e a Figura 6 mostra a interface correspondente.

```
object Form1: TForm1
  Left = 192
  Top = 114
  Width = 213
  Height = 169
  Caption = 'Form1'
  Color = clBtnFace
  Font.Charset = DEFAULT_CHARSET
  Font.Color = clWindowText
  Font.Height = -11
  Font.Name = 'MS Sans Serif'
  Font.Style = []
  OldCreateOrder = False
  PixelsPerInch = 96
  TextHeight = 13
  object Button1: TButton
    Left = 64
    Top = 48
    Width = 75
    Height = 25
    Caption = 'Abrir'
    TabOrder = 0
  end
end
```

identificação

propriedades

objeto componente

Quadro 2 – Estrutura de um arquivo .DFM

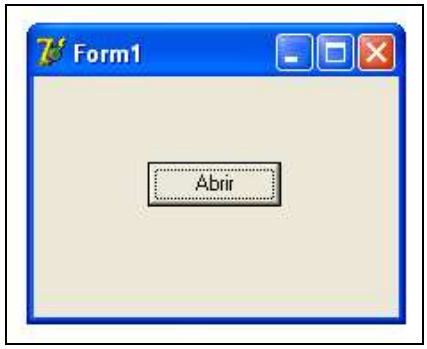


Figura 6 – Interface

Como foi mostrado no Quadro 2, um arquivo .DFM é organizado de uma maneira hierárquica. Dessa forma, sua estrutura pode ser descrita utilizando a gramática[1] do Quadro 3 abaixo.

[1] Em Colibri (2005) a gramática foi especificada usando a notação *Indented Extended Backus Naur Form* (IEBNF), onde: símbolos não terminais são escritos em itálico; símbolos terminais são escritos letra minúscula; palavras reservadas em letras maiúsculas e símbolos especiais entre aspas simples. Além disso, os símbolos entre chaves podem ocorrer várias vezes e os símbolos entre colchetes são opcionais.

```
dfm= object
  object= OBJECT name ':' type_name
              property { property } { object }
          END
    property= name [ '.' name ] '=' value
      value= integer | double | TRUE | FALSE | string
           | name [ '.' name ]
           | '[' [ value { ',' value } ] ']'
           | '(' value  { [ '+' ] value }  ')'
           | '{' value { value } '}'
           | '<' collection_item { collection_item } '>'
       collection_item= name property { property } END
```

Fonte: adaptado de Colibri (2005)

Quadro 3 – Gramática do .DFM

2.4 HMTL

HTML é uma linguagem utilizada para criar arquivos que podem ser utilizados na World Wide Web. É uma linguagem descritiva, criada para ser lida em qualquer computador que tenha instalado um programa navegador ou *browser* (CONTI, 2005). No entanto, segundo DevMedia (2004), "Ao contrário do que muitas pessoas pensam, não é necessário nenhum programa específico para criar documentos HTML, bastando utilizar um simples processador de texto como o bloco de notas, e conhecer a linguagem HTML".

A linguagem HTML tem definida uma estrutura baseada na utilização das *tags*. *Tags* são etiquetas ou marcações que consistem em breves instruções tendo uma marca de início e, algumas vezes, outra de fim, mediante as quais se determina a formatação do texto, das imagens e dos demais elementos que compõem uma página HTML (WIKIPEDIA, 2005). Para montar um formulário em HTML, por exemplo, a linguagem oferece um conjunto de componentes para entrada de dados e submissão para um programa, em um servidor, onde será realizado o seu processamento. Um componente de formulário é um *container* para vários componentes de interação com usuário. São exemplos de componentes: *input*, *label*, *button*, *select e textArea*.

O Quadro 4 mostra a estrutura de um arquivo HTML com um botão e a Figura 7

mostra a interface correspondente.

```
<HTML>
  <HEAD>
      <TITLE>Form1</TITLE>
  </HEAD>
  <BODY>
      <FIELDSET
           name="Form1"
           style="position: absolute;
           font-family: MS Sans Serif;
           font-size: 8;
           color: #000000;
           background-color: #F3EEE1;
           left: 3; top: 3;
           width: 213; height: 169;">

          <LEGEND>Form1</LEGEND>

           <FORM>                          marca de início
             <BUTTON
                  name="Button1"
      tag         style="position: absolute;
    <button>      font-family: MS Sans Serif;
                  font-size: 8;
                  left: 64; top: 73;
                  width: 75; height: 25;">
                  Abrir
             </BUTTON>
           </FORM>
      </FIELDSET>                          marca de fim
  </BODY>
</HTML>
```

Quadro 4 – Estrutura de um arquivo HTML

Figura 7 – Interface

2.5 LASZLO

Laszlo é uma ferramenta livre que foi criada para ser utilizada na construção de aplicações web com base no *plugin* Flash. A arquitetura da ferramenta combina o poder e a usabilidade do projeto cliente/servidor com as vantagens e a eficácia de custos administrativos de aplicações web (LASZLO SYSTEMS, 2005). Possui origem comercial e compete com o Flex da Macromedia e outras iniciativas menos robustas com base no *plugin* Flash.

As interfaces bem elaboradas das aplicações desenvolvidas em Laszlo "encantam" usuários finais, mostrando real diferenciação se comparadas a modelos de interface web anteriores, com base em HTML ou variantes. Laszlo possui características de animação *built-*

in inclusive superiores às *Application Program Interface* (APIs) gráficas de interfaces *desktop* (LASZLO SYSTEMS, 2005).

As aplicações em Laszlo são escritas na linguagem LZX baseada em XML. No arquivo com extensão .LZX é definido o comportamento de todas as *tags* XML com JavaScript. Para compilar esses arquivos deve ser iniciado o servidor do Laszlo com a URL `localhost:8080/lps-3.0/caminho-do-arquivo/arquivo.LZX`. O Quadro 5 possui um exemplo do código LZX e a Figura 8 o resultado da sua execução.

```
<CANVAS width="1000" height="1000">
    <WINDOW
        closeable="true"
        font="MS Sans Serif"
        bgcolor="#F3EEE1"
        title="Form1"
        name="Form1"
        x="192" y="114"
        width="225" height="189">

        <BUTTON
            font="MS Sans Serif"
            fontsize="8"
            name="Button1"
            x="64" y="48"
            width="75" height="25">
            Abrir
        </BUTTON>
    </WINDOW>
</CANVAS>
```

tag <button>

Quadro 5 – Código fonte de um arquivo .LZX

Figura 8 – Interface

2.6 XML

Documentos XML são geralmente armazenados em arquivos texto com extensão .XML, embora não seja um requisito. Pode ser usado qualquer editor de texto para criar um documento XML. Segundo Deitel et al (2003), todos os documentos devem conter um único elemento como nó raiz, uma marca de abertura e finalização para cada elemento componente, marcas corretamente aninhadas e valores de atributos entre aspas. Os atributos podem descrever dados textuais, imagens, gráficos vetoriais, animações ou qualquer outro tipo de

dado para qual a XML seja estendida.

Deitel et al (2003) afirma que a independência dos dados, a separação de conteúdo e sua apresentação são as características essenciais de XML. Uma vez que um documento XML descreve dados, pode-se imaginar que ele possa ser processado por um aplicativo. A ausência de instruções de formatação torna a XML uma estrutura de referência que pode ser usada para intercâmbio de dados.

Marchal (2000) cita algumas das áreas onde a XML é utilizada, entre elas:

a) manutenção de grandes *site*s web;
b) troca de dados;
c) aplicações de comércio eletrônico;
d) aplicações científicas com linguagens de marcação para fórmulas matemáticas e químicas;
e) livros eletrônicos com linguagens de marcação para expressar propriedades e direitos;
f) telefones inteligentes e dispositivos portáteis com linguagens de marcação otimizadas.

2.7 JAVASCRIPT

JavaScript é uma linguagem desenvolvida pela Netscape para ajudar os desenvolvedores na implementação de *sites* mais interativos. Mesmo sendo uma linguagem que compartilha de muitas das características e das estruturas da linguagem Java, foi desenvolvida independentemente e é uma linguagem aberta que permite que qualquer pessoa possa utilizá-la sem a necessidade de comprar uma licença (BRILLINGER; EPP, 2005).

Todas as páginas escritas com HTML são estáticas e fazem com que o *browser* apenas

leia o que há no código e reproduza aquele conjunto de instruções. Um dos principais recursos do JavaScript é a possibilidade de fazer com que a página HTML seja dinâmica, de tal forma que o usuário possa interagir com a mesma. Os *scripts* escritos em JavaScript também permitem a manipulação dos objetos de uma página HTML, tendo a facilidade para alterá-los automaticamente. São inseridos nas páginas de uma maneira especial, podendo ter apenas uma área para *scripts* ou estando espalhados pelo código, do jeito que melhor se adapte às necessidades de cada aplicação (LOURENÇO, 2005).

2.8 TRABALHOS CORRELATOS

Durante o primeiro semestre de 2005 foi desenvolvida na Universidade Regional de Blumenau (FURB), uma ferramenta que permite a conversão de formulários Delphi em aplicações Java (FONSECA, 2005). Esta aplicação lê o conteúdo dos arquivos .DFM e produz classes Java que apresentam o mesmo comportamento dos formulários desenvolvidos em Delphi. A ferramenta converte apenas um grupo de componentes, selecionados entre os mais usuais e que possuíam equivalência em Java. Os arquivos convertidos preservam o nome dos objetos presentes no formulário original e suas principais propriedades. São gerados dois arquivos, sendo um para a criação dos componentes e atribuição das suas propriedades e outro para manipulação dos eventos.

A DMSNet Soluções em TI (DMSNET, 2004) desenvolveu uma solução que oferece suporte metodológico e tecnológico para a migração de sistemas para a plataforma Java (J2EE). Para isto são usadas duas ferramentas de apoio no processo de migração:

a) *Code Translator*, que permite a conversão do código fonte, seja COBOL, PowerBuilder, Borland Delphi ou outro para a linguagem Java, preservando a funcionalidade original;

b) *Data Translator*, que permite migrar do banco de dados corporativo para outro banco de dados, como MS SQL Server, Oracle SQL Server, IBM DB2 Server, etc.

Seguindo essa linha, a Microsoft desenvolveu um assistente de conversão da linguagem Java para a plataforma Microsoft, o *Java Language Conversion Assistant* (JLCA). Essa ferramenta, destinada aos desenvolvedores de software e de aplicações na linguagem Java, permite a migração de aplicações na linguagem Java para C#. O assistente de conversão JLCA foi construído sobre a tecnologia de migração ArtinSoft e converte automaticamente a maior parte de códigos-fonte escritos na linguagem Java para o C#, para a sintaxe da linguagem de migração e para as chamadas de biblioteca (LINHA DE CÓDIGO, 2002).

3 DESENVOLVIMENTO DO TRABALHO

A motivação para o desenvolvimento deste trabalho surgiu a partir da proposta do professor Mauro Marcelo Mattos para desenvolver uma ferramenta para converter formulários Delphi em páginas HTML, visando facilitar a migração de sistemas desenvolvidos em Delphi para web.

Assim, a ferramenta DelphiToWeb é um gerador de camada de aplicação sendo que o modelo usado difere ligeiramente daquele apresentado no capítulo anterior. Tem-se que a entrada do gerador é um arquivo com extensão .DFM, contendo toda a informação necessária para construir código com as funcionalidades para a camada de interface de uma aplicação web. A saída desse gerador são páginas HTML ou arquivos .LZX e XML.

No desenvolvimento do gerador foram seguidos os seguintes passos:

a) especificação dos requisitos (seção 3.1);
b) implementação manual da saída, determinando quais componentes devem ser convertidos pela ferramenta e como deve ser o código gerado (seção 3.2);
c) especificação dos analisadores léxico, sintático e semântico para analisar a entrada (seção 3.3);
d) modelagem da ferramenta (seção 3.4);
e) implementação do processamento da entrada e da formatação da saída (seção 3.5).

3.1 REQUISITOS PRINCIPAIS DO PROBLEMA A SER TRABALHADO

Os requisitos funcionais (RF) e não funcionais (RNF) da ferramenta são apresentados, respectivamente, nos Quadros 6 e 7, sendo identificados os requisitos funcionais que foram implementados e os requisitos não funcionais que são contemplados na implementação.

REQUISITOS FUNCIONAIS	IMPLEMENTADOS
RF01: A ferramenta deve executar a conversão de formulários Delphi para páginas HTML.	X
RF02: A ferramenta deve executar a conversão de formulários Delphi para arquivo com extensão .LZX.	X
RF03: A ferramenta deve executar a conversão de formulários Delphi para arquivo com extensão .XML.	X

Quadro 6 - Requisitos funcionais

REQUISITOS NÃO FUNCIONAIS	CONTEMPLADOS
RNF01: A ferramenta deve permitir selecionar vários arquivos para conversão.	X
RNF02: A ferramenta deve manter o *layout* dos componentes dos formulários Delphi convertidos.	X
RNF03: Para a implementação da ferramenta deve ser utilizado o ambiente Eclipse 3.1.	X
RNF04: Deve ser utilizado *Java Software Development Kit* (J2SDK), versão 1.5.0_02.	X

Quadro 7 - Requisitos não funcionais

3.2 COMPONENTES CONVERTIDOS

Uma vez que os requisitos foram especificados, partiu-se para a determinação de quais componentes Delphi serão convertidos pela ferramenta. Os critérios para seleção desses componentes foram: identificação dos que são mais usados e dos que tinham componentes equivalentes em HTML e no Laszlo.

O Quadro 8 apresenta a funcionalidade de cada componente que é convertido pela ferramenta DelphiToWeb bem como seu correspondente em HTML e no Laszlo. Na Figura 9 é mostrado um *Form* Delphi contendo todos os componentes descritos.

	COMPONENTE	**DESCRIÇÃO**	***tag* HTML**	***tag* LZX**
1	*TForm*	formulário ou janela da aplicação	`<fieldset>`	`<window>`
2	*TMainMenu*	*menu* principal	`<div>`	`<menubar>`
3	*TMenuItem*	opções do *menu* principal ou de *menus popup*	`<div>` `<button>`	`<menuitem>`
4	*TToolBar*	barra de tarefas	`<div>`	`<view>`
5	*TToolButton*	botões da barra de tarefas	`<button>`	`<button>`
6	*TLabel*	texto (rótulo)	`<font>`	`<text>`
7	*TEdit*	caixa para digitação de texto com uma única linha	`<input>`	`<edittext>`
8	*TButton*	botão sem figura	`<button>`	`<button>`
9	*TSpeedButton*	botão sem foco com opção de manter-se pressionado	`<button>`	`<button>`
10	*TBitBtn*	botão com figura opcional	`<button>`	`<button>`
11	*TComboBox*	caixa de seleção, com uma única linha, com uma lista de opções	`<select>`	`<combobox>`
12	*TMemo*	caixa para digitação de texto com barra de rolagem opcional	`<textarea>`	`<edittext>`
13	*TRichEdit*	similar ao *TMemo* com propriedades adicionais	`<textarea>`	`<edittext>`
14	*TListBox*	caixa com uma lista de opções com barra de rolagem	`<select>`	`<list>`
15	*TGroupBox*	painel para agrupar componentes, com título opcional e linha em baixo relevo	`<fieldset>`	`<view>`
16	*TRadioGroup*	semelhante ao *TGroupBox* com itens que possuem a mesma aparência do *TRadioButton*	`<fieldset>`	`<view>`
17	*TRadioButton*	botão em forma de círculo seguido de um rótulo	`<input>`	`<radiobutton>`
18	*TPanel*	painel para agrupar componentes	`<fieldset>`	`<view>`
19	*TCheckBox*	caixa de seleção	`<input>`	`<checkbox>`
20	*TPopupMenu*	*menu* acionado com o botão direito do *mouse*	`<div>` `<button>`	-
21	*TPageControl*	controlador de páginas (*TTabSheet*)	`<table>`	`<tabs>`
22	*TTabSheet*	página inserida no *TPageControl*	`<div>`	`<tabpane>`

Quadro 8 – Descrição dos componentes

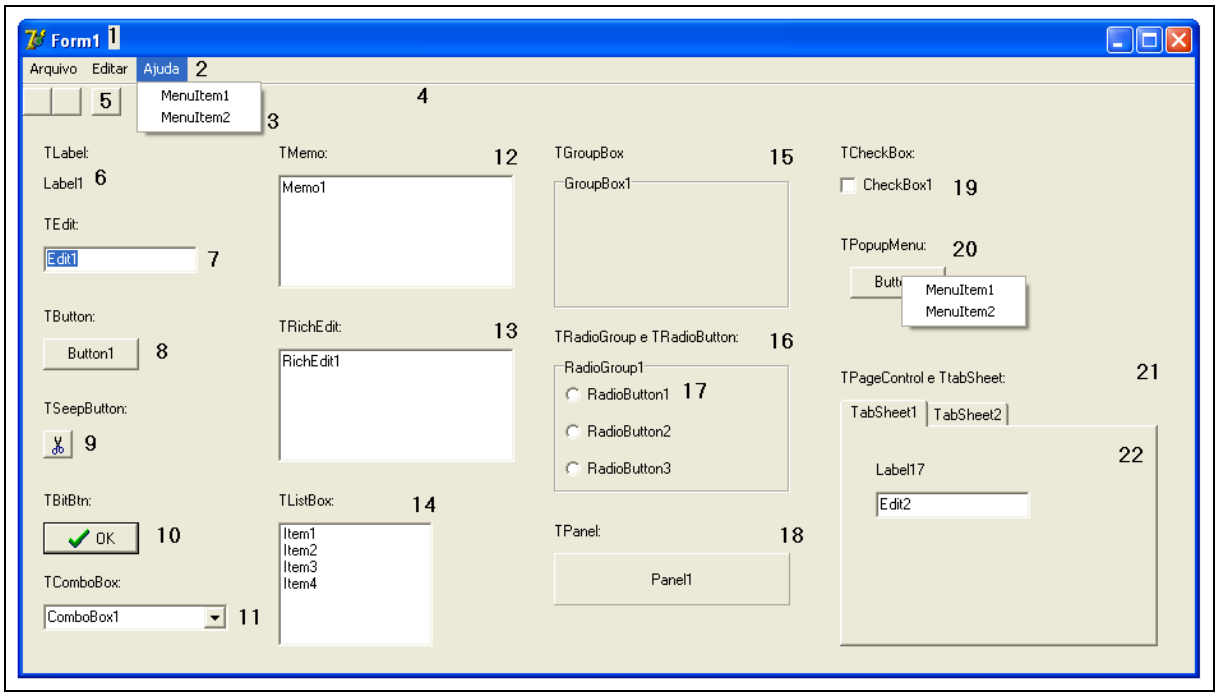


Figura 9 – *Form* Delphi

3.3 ANÁLISE DE ARQUIVOS .DFM

Os arquivos .DFM podem ser processados por analisadores léxico, sintático e semântico. Assim, em um arquivo .DFM, conforme especificação apresentada no capítulo 2, é possível identificar os seguintes *tokens*:

a) identificador (`identifier`): é composto por uma seqüência de letras, dígitos ou *underline*, iniciando com uma letra;

b) constante inteira (`integer_constant`): é composta por um ou mais dígitos;

c) constante real (`real_constant`): é composta por dígitos separados por ponto;

d) constante literal (`string_constant`): pode ser uma seqüência de caracteres entre apóstrofos ou dígitos precedidos pelo símbolo #;

e) palavras reservadas: podem ser `OBJECT`, `END`, `FALSE`, `TRUE` ou o nome dos componentes descritos na seção anterior (`TBitBtn`, `TButton`, etc.);

f) símbolos especiais: são os símbolos que compõe a sintaxe do arquivo .DFM.

A especificação dos *tokens* encontra-se no Quadro 9, onde `dígito`, `letra` e `string` são definições regulares auxiliares.

```
// tokens
identifier: {letra} ({letra}|{digito})*
integer_constant: {digito}+
real_constant: {digito}+ "." {digito}+
string_constant: ('{string}*' | #{digito}+)+

// palavras reservadas
OBJECT = identifier : "OBJECT"
END = identifier : "END"
FALSE = identifier : "False"
TRUE = identifier : "True"
TBitBtn = identifier : "TBitBtn"
TButton = identifier : "TButton"
...

// símbolos especiais
: . = [ ] ( ) , < > { } + - *

// definições regulares auxiliares
digito : [0-9]
letra  : [a-zA-Z_]
string: [^'\n\r]
```

Quadro 9 – Especificação dos *tokens*

A estrutura sintática de um arquivo .DFM, especificada utilizando a notação BNF, encontra-se no Quadro 10.

```
<dfm>::= #1 <object> #10
<object>::= OBJECT identifier #2 ":" <type> #3 <propertyList> <objectList> END

<type>::= identifier | TBitBtn | TButton |  TCheckBox | TcomboBox | Tedit
        | TGroupBox | TLabel | TListBox | TMainMenu | TMemo | TmenuItem | TPanel
        | TPageControl | TPopupMenu | TProgressBar | TSpeedButton | TSpinEdit
        | TTabSheet | TtoolBar | TToolButton | TRadioButton  | TRadioGroup
        | TRichEdit | TStringGrid

<objectList>::= ε | #8 <object> #9 <objectList>
<propertyList>::= ε | <property> <propertyList>
<property>::= #4 <name> #5 "=" #6 <value> #7

<value>::=  <number> | string_constant | <name> | <booleanConstant>
          | "[" <valueList1> "]" | "(" <valueList2> ")"
          | "{" <valueList2> "}" | <collection>
<name>::= identifier <name_>
<name_>::= ε | "." identifier

<number>::= <signal> <number_> ;
<number_>::= integer_constant | real_constant;
<signal>::= ε | "+" | "-";

<booleanConstant>::= FALSE | TRUE ;
<valueList1>::= ε | <value>  <valueList1_> ;
<valueList1_>::= ε | ","<value> <valueList1_> ;
<valueList2>::= <value> <valueList2_> ;
<valueList2_>::= ε | <valueList2> ;
<collection>::= "<" <collectionList> ">";
<collectionList>::= ε | <collectionItem> <collectionList> ;
<collectionItem>::= identifier <propertyList> END ;
```

Quadro 10 – Gramática

No quadro anterior foram incluídas ações semânticas marcadas com o símbolo # seguido de um número para indicar a análise de contexto do arquivo .DFM. A especificação das ações semânticas encontra-se no Quadro 11.

AÇÃO	DESCRIÇÃO
1	reconhecimento do início do arquivo .DFM: criar tupla (pai) para armazenar o identificador do objeto, o tipo, as propriedades e os sub-objetos
2	armazenar o identificador do objeto na tupla
3	armazenar o tipo do objeto na tupla
4 e 5	armazenar o nome da propriedade na tupla
6 e 7	armazenar o valor da propriedade na tupla
8	reconhecimento de um sub-objeto: criar tupla para armazenar o identificador do sub-objeto, o tipo, as propriedades, os sub-objetos e o objeto pai
9	finalizar reconhecimento de sub-objeto
10	finalizar reconhecimento do arquivo .DFM

Quadro 11 – Ações semânticas

3.4 ESPECIFICAÇÃO

Foi utilizada a *Unified Modeling Language* (UML) como linguagem de especificação dos diagramas de casos de uso e de classes do sistema, com a utilização da ferramenta Enterprise Architect (EA).

3.4.1 Diagrama de casos de uso

Na Figura 10 encontram-se os casos de uso da aplicação e no Quadro 12 o detalhamento de cada um deles.

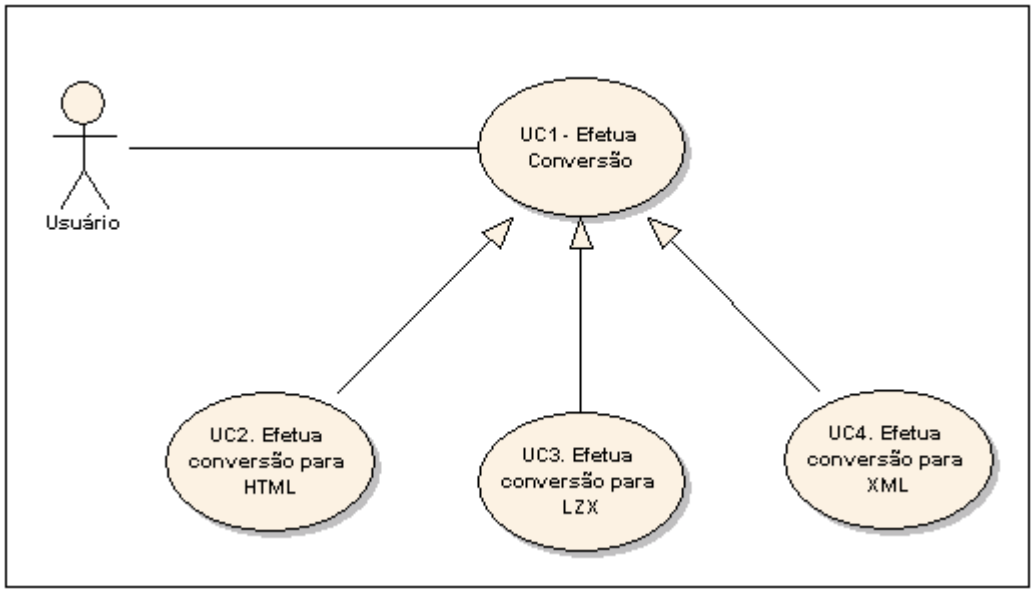


Figura 10 - Diagrama de casos de uso

UC1. Efetuar conversão **Pré-condição**: Os arquivos .DFM não podem estar corrompidos. **Cenário principal:** 1. A ferramenta apresenta as opções para selecionar os arquivos e o diretório de saída. 2. O usuário seleciona o arquivo a ser convertido e o diretório onde deseja gerar os arquivos de saída. 3. A ferramenta tem opções (HTML, Laszlo ou XML) para conversão de arquivos .DFM. 4. O usuário seleciona a opção desejada. 5. A ferramenta faz a conversão do código. **Pós-condição**: Um ou vários arquivos gerados.
UC2. Efetua conversão para HTML **Pré-condição**: idem UC1 **Cenário principal:** 1. Passos 1 a 3 idênticos aos do cenário principal do UC1. 2. O usuário escolhe a opção de converter para HTML. 3. A ferramenta faz a conversão do código para HTML. **Pós-condição**: Um arquivo com extensão .HTML é gerado.
UC3. Efetua conversão para Laszlo **Pré-condição**: idem UC1 **Cenário principal:** 1. Passos 1 a 3 idênticos aos do cenário principal do UC1. 2. O usuário escolhe a opção de converter para Laszlo. 3. A ferramenta faz a conversão do código para Laszlo. **Pós-condição**: Um arquivo com extensão .LZX é gerado.
UC4. Efetua conversão para XML **Pré-condição**: idem UC1 **Cenário principal:** 1. Passos 1 a 3 idênticos aos do cenário principal do UC1. 2. O usuário escolhe a opção de converter para XML. 3. A ferramenta faz a conversão do código para XML. **Pós-condição**: Um arquivo com extensão .XML é gerado.

Quadro 12 – Detalhamento dos casos de uso

3.4.2 Diagrama de classes

No diagrama de classe da Figura 11 foram modeladas as classes que representam as

análises léxica, sintática e semântica e as classes para armazenar os dados extraídos dos arquivos .DFM necessários para conversão.

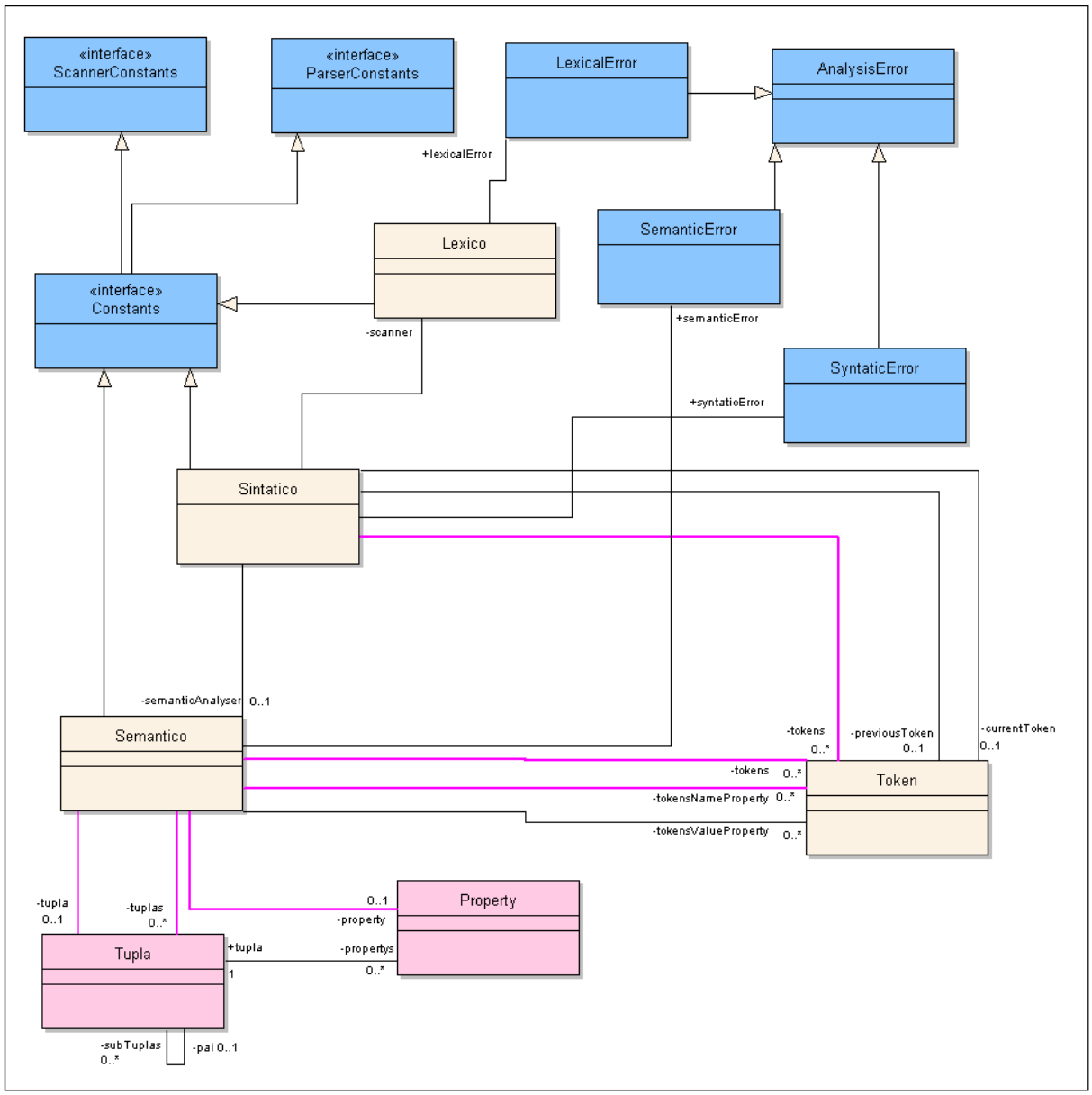


Figura 11 – Diagrama de classes para análise do arquivo .DFM

Na figura anterior as classes em azul e amarelo são as classes geradas pelo GALS (GESSER, 2003), sendo que essas últimas foram adaptadas conforme o desenvolvimento da ferramenta. As classes em rosa foram criadas para armazenar os dados necessários para montar a saída da conversão.

A Figura 12 detalha as classes responsáveis pela detecção dos erros na análise de arquivos .DFM. A classe `AnalysisError` é subclasse da classe `Exception` do Java e é a

classe base das classes de erro dos analisadores. Possui o atributo `position` para guardar a posição da mensagem de erro que se encontra em vetores nas classes `ScannerConstants` e `ParserConstants`.

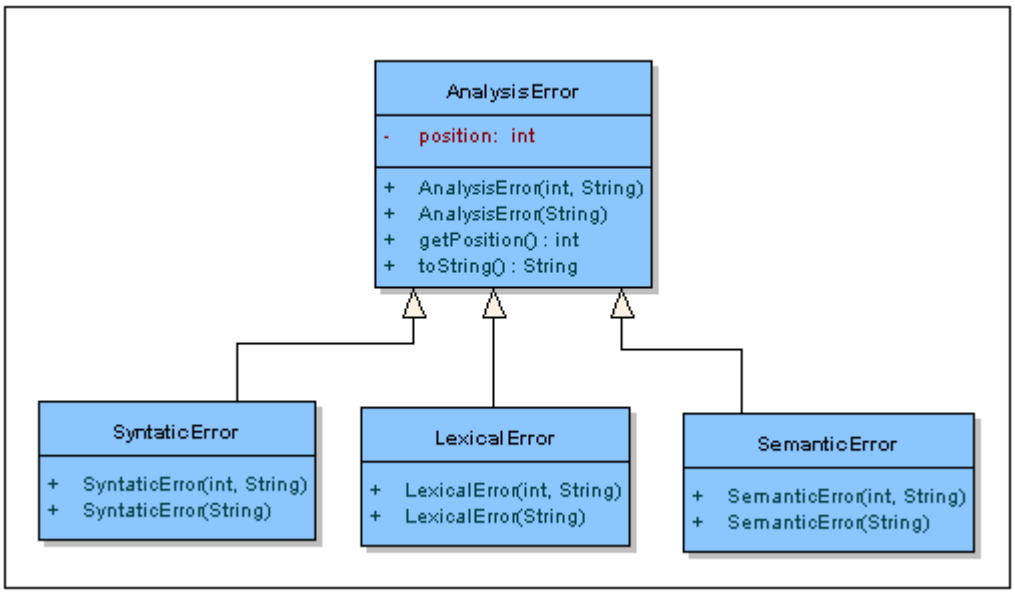


Figura 12 – Classes para detecção de erro

As classes da Figura 13 são interfaces implementadas pelas classes que fazem a análise léxica e sintática (`Lexico` e `Sintatico`, respectivamente). Os atributos da classe `ScannerConstants` são: um vetor (`SCANNER_ERROR`) contendo as mensagens de erro; um vetor (`SPECIAL_CASES_KEYS`) com as palavras reservadas; e vetores auxiliares (`TOKEN_STATE`, `SPECIAL_CASES_INDEXES`, `SPECIAL_CASES_VALUES`) usados no reconhecimentos dos *tokens*. A interface `ParserConstants` possui somente um atributo, um vetor (`PARSER_ERROR`) que contém as mensagem de erro de sintaxe. A interface `Constants` tem como base as duas interfaces descritas anteriormente e possui 29 atributos, que são constantes inteiras usadas na análise do arquivo .DFM.

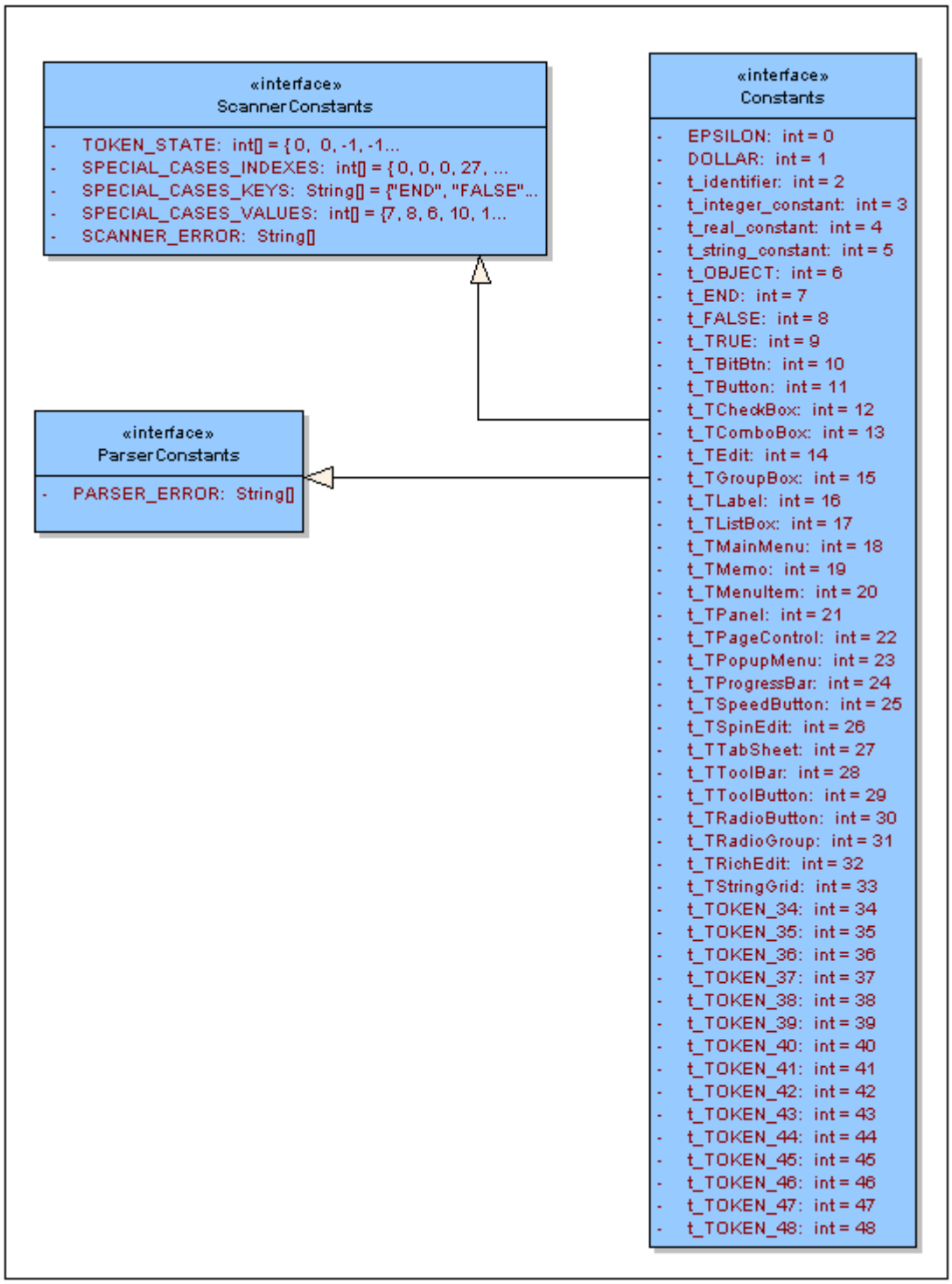


Figura 13 – Classes Constants, ScannerConstants e ParserConstants

As classes `Lexico`, `Sintatico` e `Semantico` (Figura 14) efetuam a análise de um arquivo .DFM. A classe `Lexico` tem os atributos `input` e `position`. No atributo `input` é guardado o arquivo .DFM passado como parâmetro no construtor da classe, já o atributo `position` serve para o controle da posição de cada `token` quando é feita a análise léxica.

A classe `Sintatico` só tem associações: duas com a classe `Token`, sendo elas

`currentToken` para armazenamento do *token* corrente reconhecido e `previousToken` para armazenamento do *token* anterior; uma com a classe `Lexico` para reconhecimento dos *tokens* presentes no arquivo .DFM; e a associação `semanticAnalyses`, para efetuar chamadas às ações semânticas conforme determinado na gramática do DFM apresentada na seção anterior. Foram adicionadas a essa classe duas coleções, uma (`tokens`) contendo todos os tokens reconhecidos e outra contendo o nome dos componentes que não são convertidos.

Finalmente, na classe `Semantico` tem-se o método `executeAction()`, que é chamado cada vez que uma ação semântica é encontrada e tem a função de guardar em objetos das classes `Tupla` e `Property` os dados extraídos do arquivo .DFM.

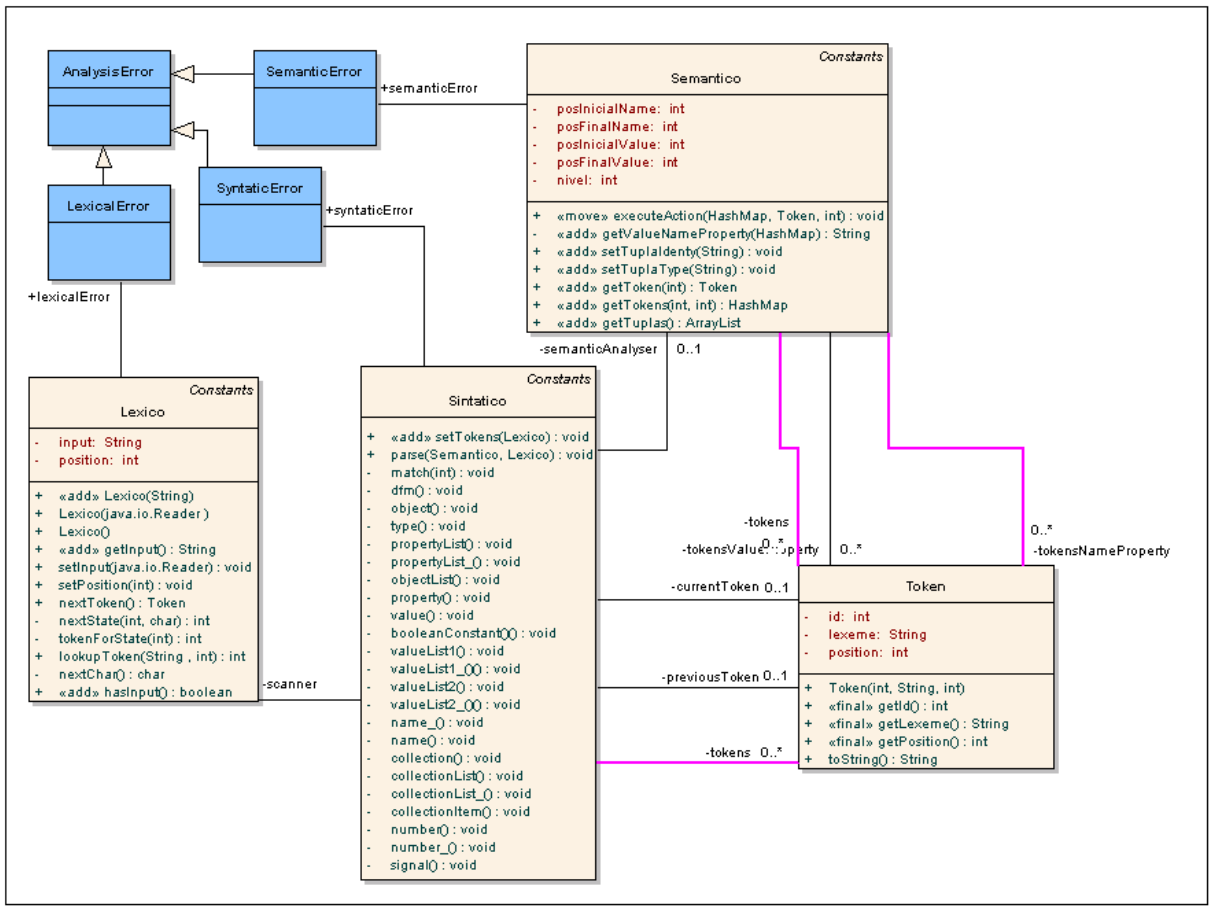


Figura 14 – Classes das análises léxica, sintática e semântica

A classe `Tupla` (Figura 15) possui os atributos `identifier` e `type`, sendo que o primeiro atributo é responsável por armazenar o identificador do componente e o segundo atributo é responsável por armazenar o tipo do componente (por exemplo, `identifier =`

`button1, type = TButton)`. Possui uma coleção de propriedades chamada `propertys`, representada pela associação com a classe `Property`, e uma coleção de sub-objetos chamada `subObjects`, representada através da associação com a própria classe.

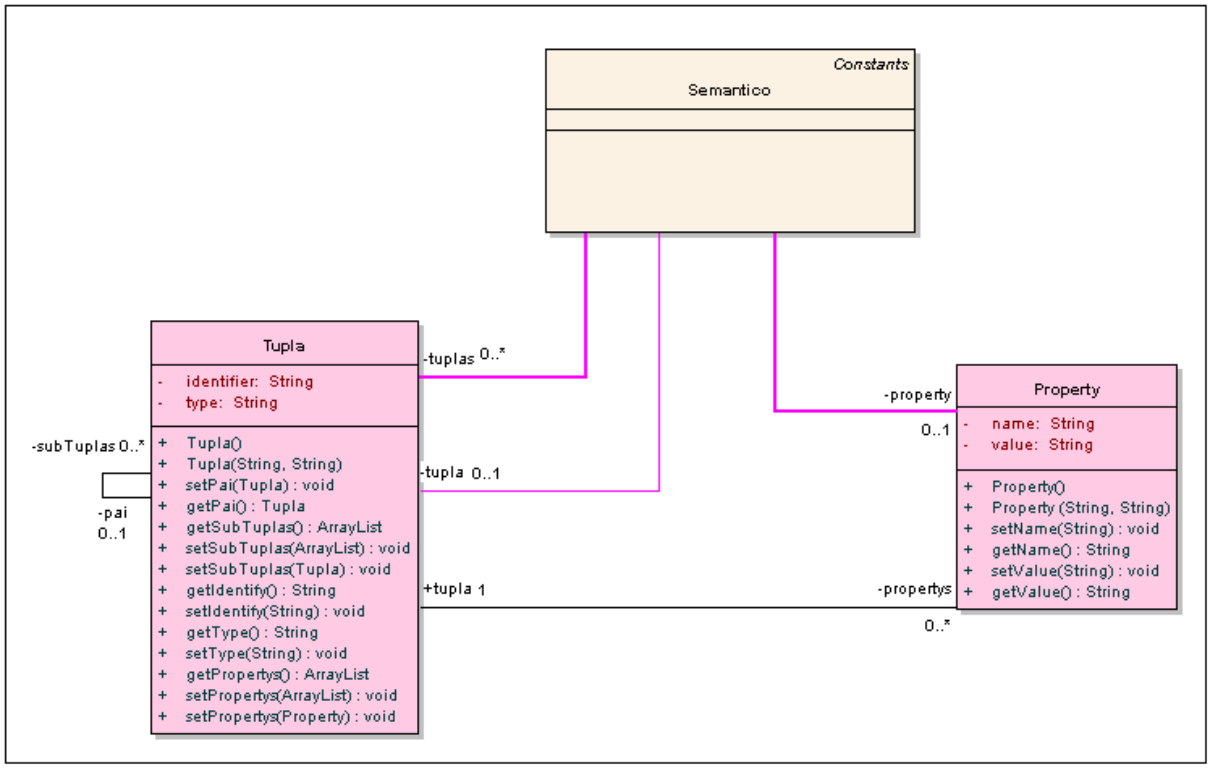


Figura 15 – Classes `Tupla` e `Property`

No diagrama de classes da Figura 16 foram modeladas as classes que representam os componentes convertidos, usadas para geração de código. Cada classe é uma subclasse da classe `Component` e implementa os métodos `geraHTML()` e `geraLaszlo()`, os quais têm a função de montar a saída de código de cada componente. A classe `Component` contém os atributos que correspondem a algumas das propriedades que são comuns para todos os componentes. Nas figuras 17 a 23 são detalhadas cada uma das subclasses de `Component`, separadas por finalidades parecidas.

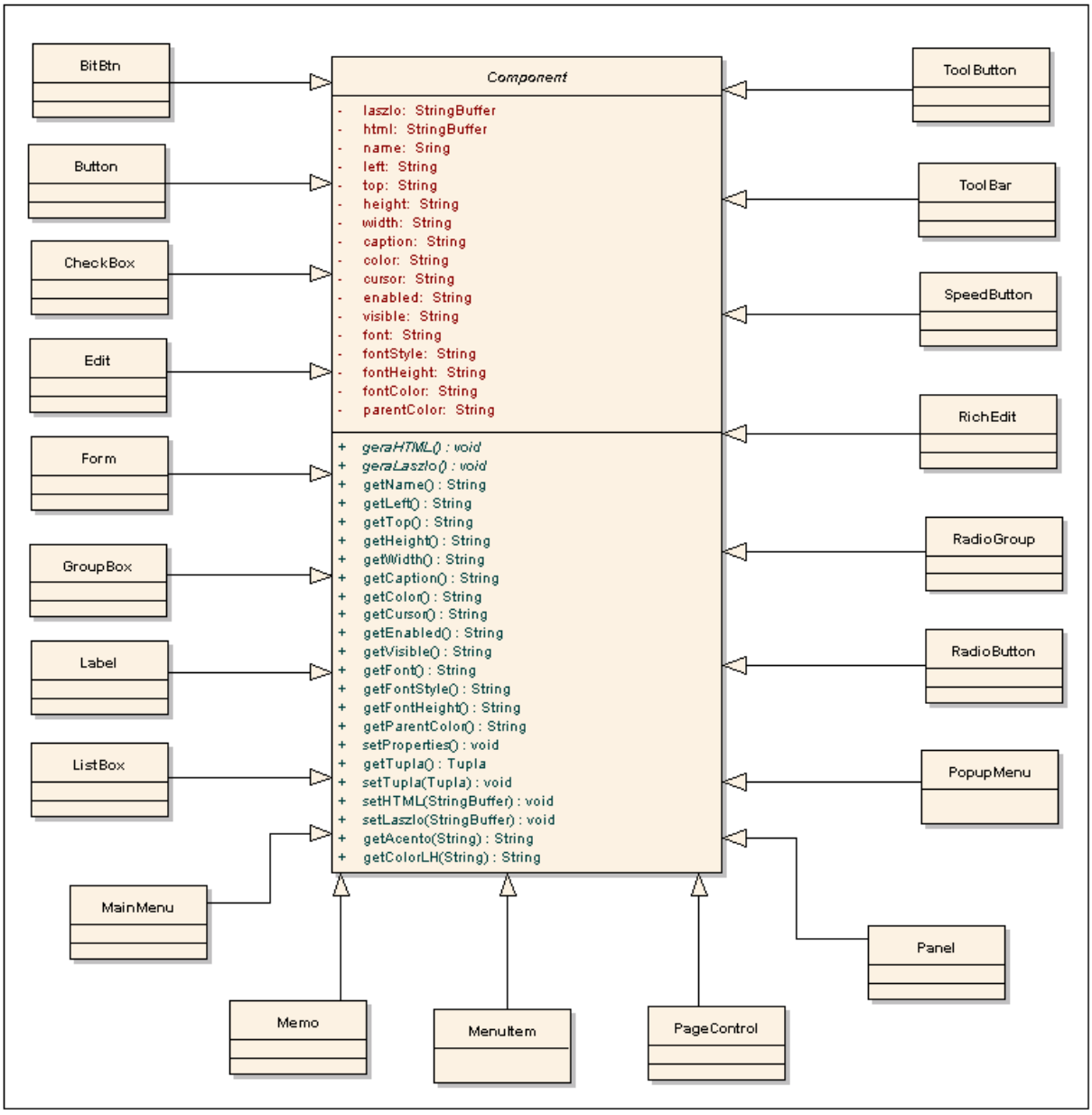


Figura 16 – Classes dos componentes

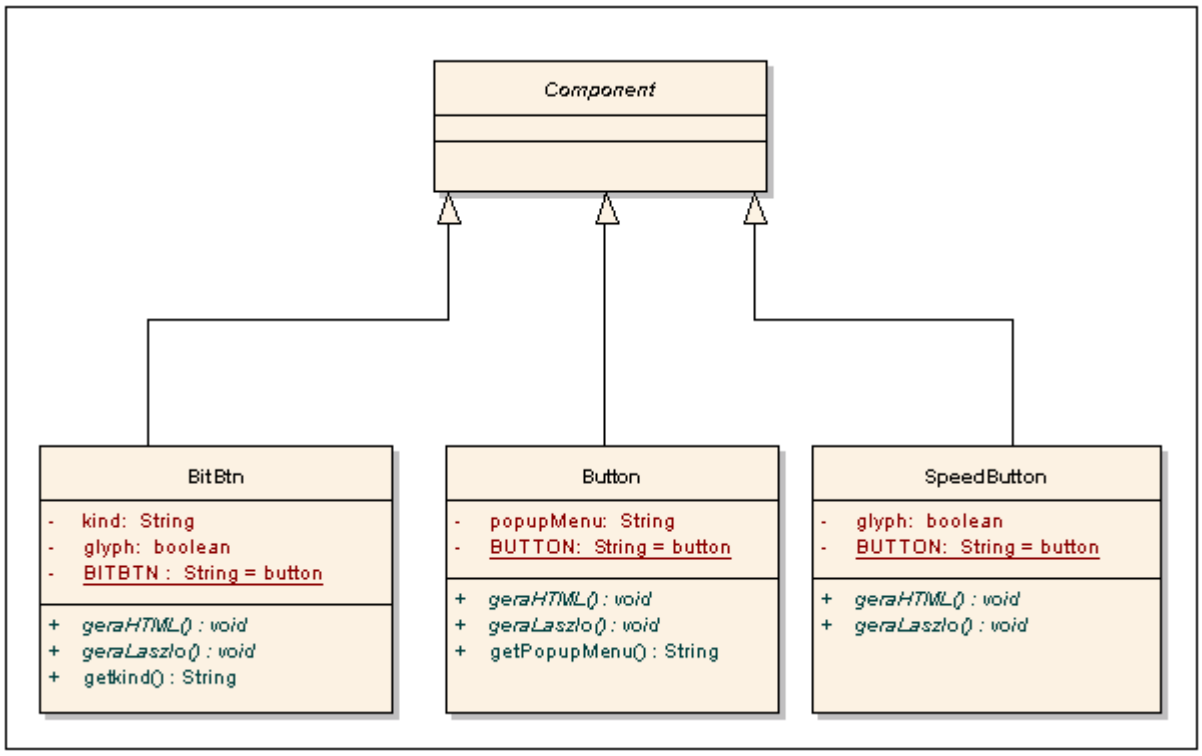


Figura 17 – Classes para os botões

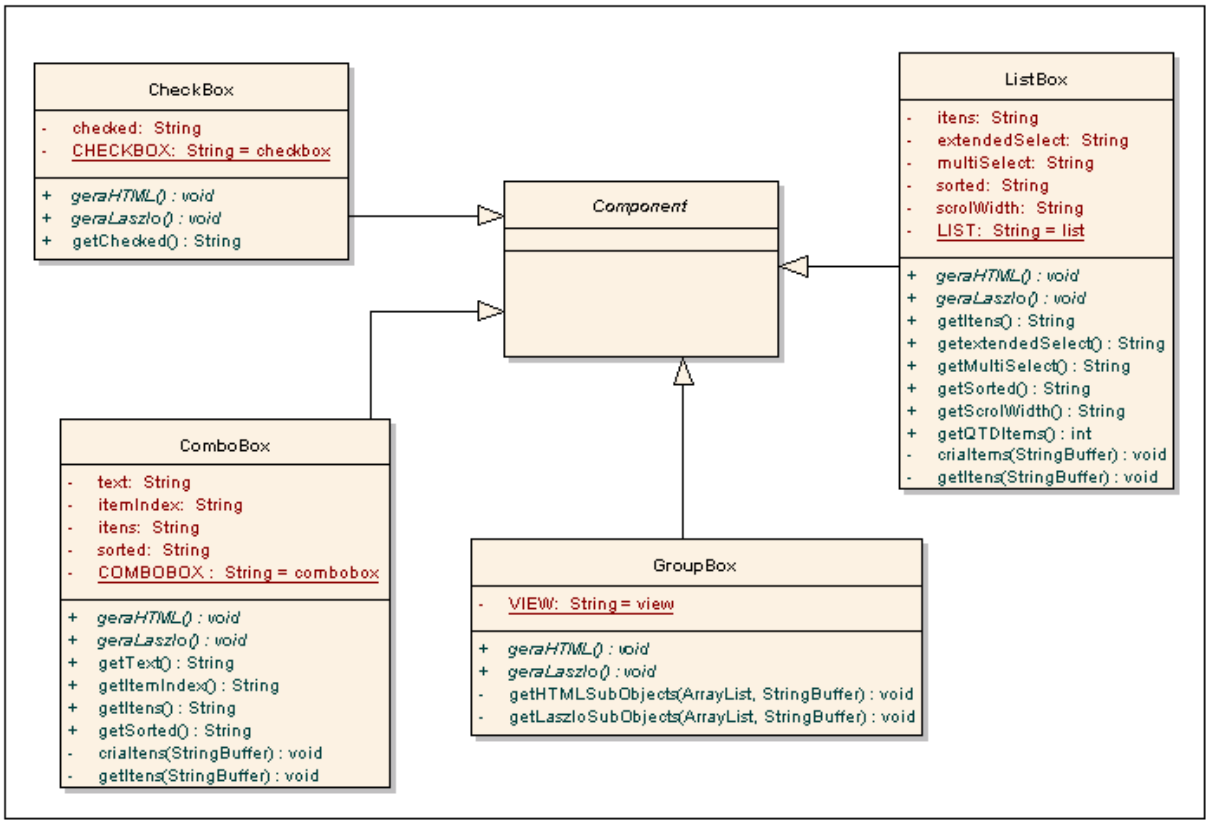


Figura 18 – Classes para os "Boxes"

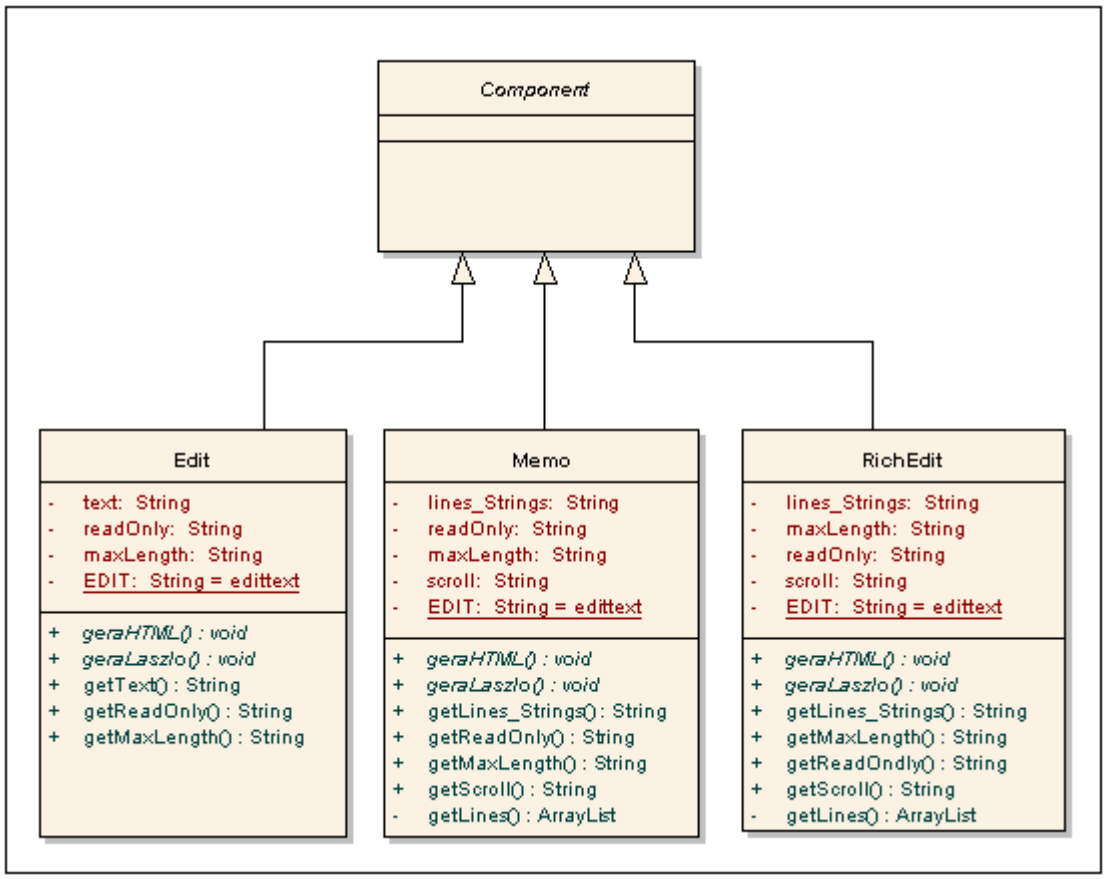


Figura 19 – Classes para os componentes de edição de texto

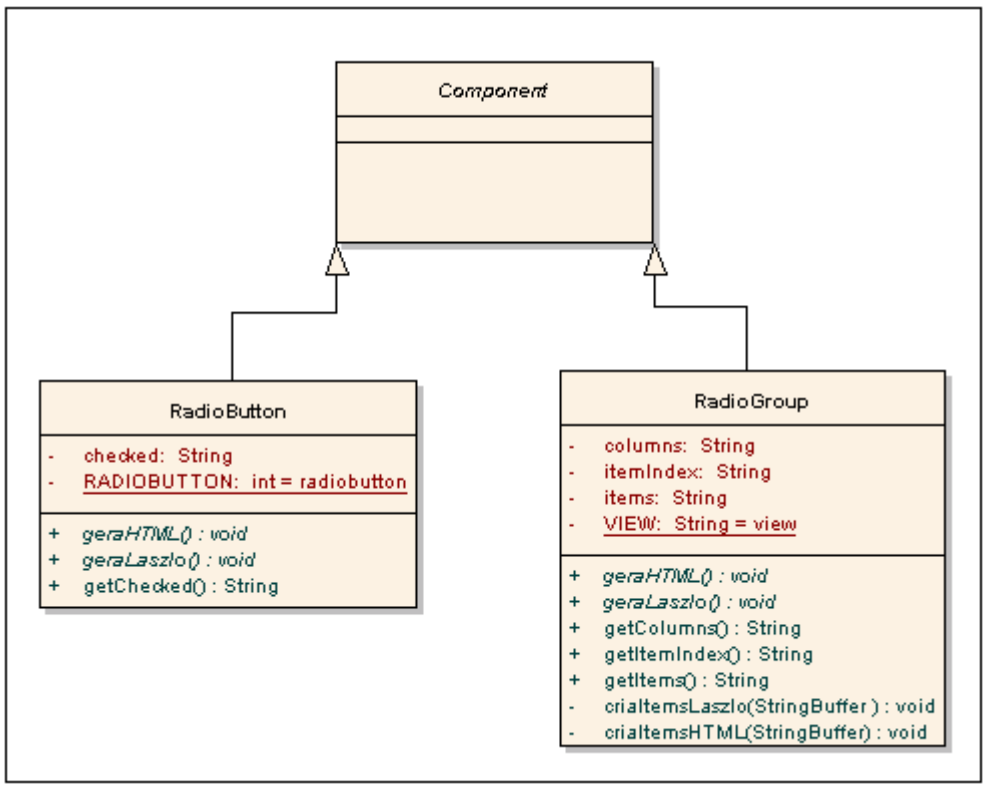


Figura 20 – Classes `RadioButton` e `RadioGroup`

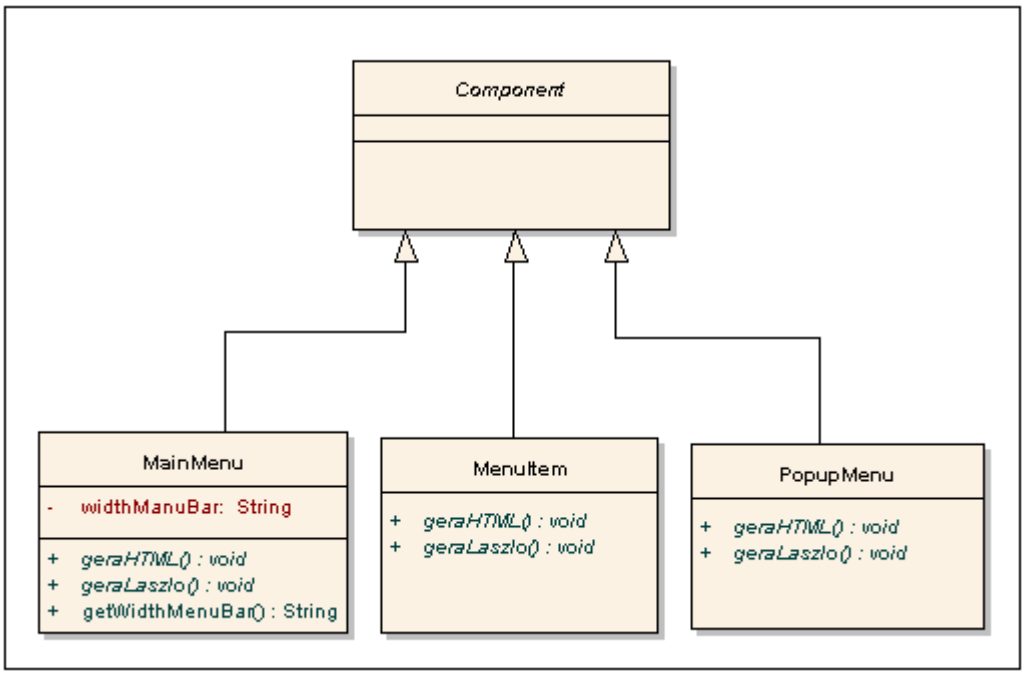


Figura 21 – Classes para os *menus*

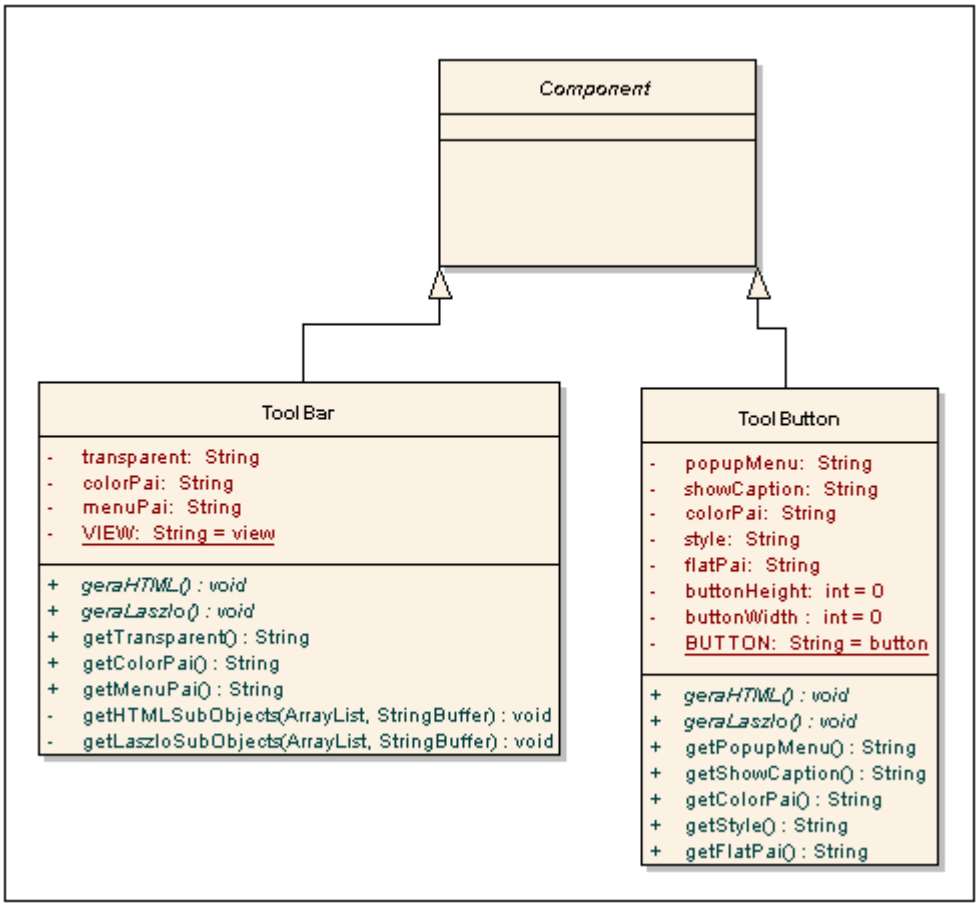


Figura 22 – Classes `ToolBar` e `ToolButton`

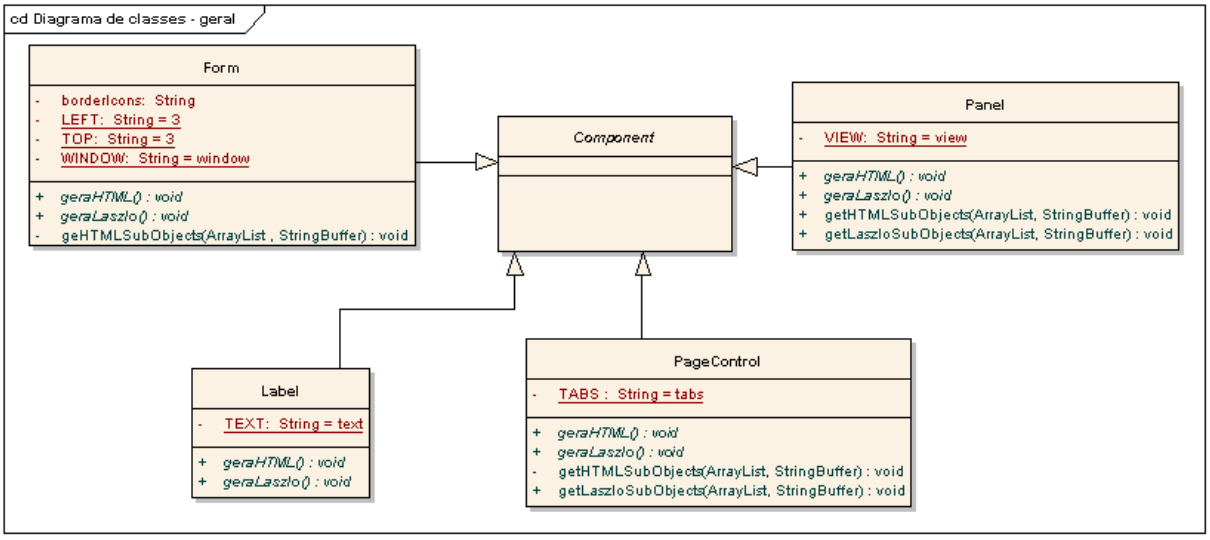


Figura 23 – Classes para os demais componentes

3.5 IMPLEMENTAÇÃO

Nessa seção são apresentados os aspectos sobre a implementação da ferramenta DelphiToWeb.

3.5.1 Ferramentas utilizadas

A ferramenta DelphiToWeb foi desenvolvida em Java utilizando o ambiente Eclipse. Na implementação também foi utilizado o GALS, uma ferramenta de código livre para geração de analisadores léxicos e sintáticos em Java, C++ ou Delphi.

O GALS possui opções para gerar somente o analisador léxico, somente o analisador sintático ou ambos. O analisador léxico é gerado a partir da definição dos aspectos léxicos especificados usando expressões regulares. O analisador sintático é gerado a partir da definição dos aspectos sintáticos especificados usando a notação BNF. Para o analisador semântico é gerada a assinatura do método `executeAction()` cuja implementação deve ser feita manualmente (GESSER, 2003).

3.5.2 Implementação da ferramenta

O último passo no desenvolvimento de um gerador de código é a implementação do processamento da entrada e da formatação da saída. A análise de entrada é feita utilizando as classes geradas pelo GALS, que fazem as análises léxica e sintática do arquivo .DFM. No Quadro 13 tem-se uma parte do código do analisador léxico.

```
public void setInput(java.io.Reader input) {
    StringBuffer bfr = new StringBuffer();
    try {
       int c = input.read();
       while (c != -1) {
           bfr.append((char) c);
           c = input.read();
       }
       this.input = bfr.toString();
    } catch (java.io.IOException e) {
        e.printStackTrace();
    }
    setPosition(0);
}

public void setPosition(int pos) {
    position = pos;
}

public Token nextToken() throws LexicalError {
    if (!hasInput())
        return null;
...
```

Seta o texto de entrada do analisador léxico, no caso o arquivo .DFM

Seta a posição do *token*

Reconhece o próximo *token* da entrada

Quadro 13 – Código fonte da classe `Lexico.java`

O método `parse` mostrado no Quadro 14 da classe `Sintatico.java` é responsável por realizar toda a análise da entrada. A partir dele são realizadas as análises léxica e semântica.

```
public void parse(Lexico scanner, Semantico semanticAnalyser)
    throws AnalysisError{

    //TODO Adaptei essa parte
    Lexico lexico = new Lexico(scanner.getInput());
    setTokens(lexico);
    //TODO Adaptei essa parte

    this.scanner = scanner;
    this.semanticAnalyser = semanticAnalyser;

    currentToken = scanner.nextToken();          <-- chama o analisador léxico
    if (currentToken == null)
        currentToken = new Token(DOLLAR, "$", 0);

    dfm();

    if (currentToken.getId() != DOLLAR)
    throw new SyntaticError(PARSER_ERROR[DOLLAR], currentToken.getPosition());
}

private void dfm() throws AnalysisError {          <-- chama o analisador semântico
    switch (currentToken.getId()) {
        case 6: // OBJECT
            semanticAnalyser.executeAction(1, previousToken, tokens);
            object();
            semanticAnalyser.executeAction(10, previousToken, tokens);
            break;
        default:
            throw new SyntaticError(PARSER_ERROR[49],currentToken.getPosition());
    }
}
```

Quadro 14 – Método `parse` da classe `Sintatico.java`

Como foi mostrado no quadro anterior, o método `executeAction()` da classe `Semantico.java` é invocando em determinados pontos da análise sintática, facilitado a implementação da análise de contexto do arquivo .DFM. O método `executeAction` tem como parâmetros o número da ação semântica, o *token* previamente reconhecido e toda a coleção de *tokens* extraída na análise léxica. Na implementação do método `executeAction` existe um *case* onde é determinada qual ação deverá ser executada. Algumas dessas ações são apresentadas no quadro a seguir (Quadro 15).

```
public void executeAction(int action, Token token, HashMap tokes)
    throws SemanticError {

    this.tokens = tokes;

    switch (action) {
        case 1: // Ação 1 inicio do arquivo .DFM
            nivel++;
            tupla = new Tupla();
            tuplaPrincipal = tupla;
            break;

        case 2: // Ação 2 pega identificardor do Objeto
            setTuplaIdenty(token.getLexeme());
            break;

        case 3: // Ação 3 pega tipo do Objeto
            setTuplaType(token.getLexeme());
            break;

        case 4: // Ação 4 pega o inicio da posição do nome da propriedade
            posInicialName = token.getPosition() + 1;
            break;
        ...
```

ação nº 1 — ação nº 2 — ação nº 3 — ação nº 4

Quadro 15 – Código fonte da classe `Semantico.java`

A formatação da saída implementa toda a montagem do código dos arquivos gerados. Isso é feito pela classe `Semântico.java`, através dos métodos `convertHTML()` e `convertLaszlo()`. A representação da chamada de um desses métodos é apresentada no Quadro 16.

```
for (int i = 0; i < itensJList.length; i++) {
    File file = (File) itensJList[i];
    FileReader reader = null;
    try {
        reader = new FileReader(file);
    } catch (FileNotFoundException e5) {
        e5.printStackTrace();
    }
    verifica(file, reader);
    Tupla element = (Tupla) tuplas.get(i);

    StringBuffer buffer = element.convertHTML();
    ...
```

invoca o método `convertHTML()` para cada elemento de `Tupla`

Quadro 16 – Código fonte da classe `TelaPrincipal.java`

Como os dados necessários para conversão estão guardados em uma estrutura hierárquica de tuplas, tupla pai e suas tuplas filhas, quando os métodos `convertHTML()` e `convertLaszlo()` são invocados, é verificado o atributo `type` de cada tupla. O atributo `type` determina o tipo do componente, como por exemplo `TButton` ou `TForm`. A partir dele é instanciada a classe conforme cada tipo de componente e são chamados os métodos

`geraHTML()` ou `geraLaszlo()` para montagem do código em HTML ou Laszlo, como mostrado no Quadro 17.

```
public StringBuffer convertHTML() {
    StringBuffer conv = new StringBuffer();

    Component comp = getComponet();          → cria objeto do tipo Componet

    if (comp != null) {
       conv = comp.geraHTML();               → chama o método para gerar o código HTML
    }
    return conv;
}

private Component getComponet() {
    Component comp = null;

    if ((type.substring(0, 2)).equalsIgnoreCase("TF")) {
        comp = new Form(this);
    } else if (type.equals("TButton")) {
        comp = new Button(this);
    } else
    ...                                      → instancia a classe conforme o tipo do componente
```

Quadro 17 – Código fonte da classe `Tupla.java`

O Quadro 18 a seguir mostra o método `geraHTML()` da classe `Button`.

```
public StringBuffer geraHTML() {
    StringBuffer html = new StringBuffer();
    html.append("<font ");
    html.append(UtilHTML.getName(getName()));
    html.append("style=\"position: absolute;");
    html.append(UtilHTML.getVisible(getVisible()));
    html.append(UtilHTML.getFont(getFont()));
    html.append(UtilHTML.getFontHeight(getFontHeight()));
    html.append(UtilHTML.getFontStyle(getFontStyle()));
    html.append(UtilHTML.getFontColor(getFontColor()));
    html.append(UtilHTML.getLeft(getLeft()));

    if (getTupla().getPai().getType().subSequence(0, 2).equals("TF")) {
        html.append
        (UtilHTML.getTop(String.valueOf(Integer.parseInt(getTop())+ 25)));
    } else if (getTupla().getPai().getType().subSequence(0,4).equals("TTab")) {
        html.append
        (UtilHTML.getTop(String.valueOf(Integer.parseInt(getTop())+30)));
    } else{
        html.append(UtilHTML.getTop(getTop()));
    }

    html.append(UtilHTML.getWidth(getWidth()));
    html.append(UtilHTML.getHeight(getHeight()));
    html.append("\">\n");
    html.append(UtilHTML.getCaption(getCaption()));
    html.append("</font>");

    setHtml(html);
     return html;
}
```

Quadro 18 – Método `geraHTML` da classe `Button.java`

Para ajudar a montagem do código HTML e do Laszlo, foram criadas as classes `UtilHTML.java` e `UtilLaszlo.java`. Cada classe possui um método para cada umas das principais propriedades dos componentes, que recebe o valor da propriedade e monta o código HTML ou Laszlo equivalente. O código de uma dessas classes pode ser observado no quadro a seguir.

```
public class UtilHTML {
    // Property COLOR
    public static String getColor(String color) {
        String htmlColor = new String();
        if (color != null) {
            htmlColor = "background-color: " + getColorLH(color) + ";\n";
        }
        return htmlColor;
    }

    // Property edinabled
    public static String getEdinabled(String edinabled) {
        String htmlEdinabled = new String();
        if (edinabled != null && edinabled.equals("False")) {
            htmlEdinabled = "disabled=\"disabled\"\n";
        }
        return htmlEdinabled;
    }
...
```

Quadro 19 – Código fonte da classe `UtilHTML.Java`

Para a montagem do arquivo XML foi utilizada a classe `XMLEncoder.java` da API Java, onde para seu construtor é passado o nome do arquivo .XML que se deseja criar e para o método `writeObject` é passada a `Tupla` com os dados do arquivo .DFM. A estrutura do XML fica conforme atributos e associações da classe `Tupla`. O código para geração de um arquivo .XML na ferramenta DelphiToWeb é mostrado no Quadro 20.

Geração do arquivo .XML

```
    Tupla element = (Tupla) tuplas.get(i);
    XMLEncoder xmlEn;

    if (jTextField != null && !jTextField.getText().equals("")) {
    try {
        xmlEn = new XMLEncoder(new BufferedOutputStream(
                    new FileOutputStream(jTextField.getText()
                    + (file.getName().substring(0, file.getName().length() - 4))
                    + ".xml")));
        xmlEn.writeObject(element);
        xmlEn.close();
        ...
```

Quadro 20 – Código fonte para gerar arquivo .XML

3.5.3 Operacionalidade da implementação

Nesta seção é apresentado um estudo de caso para demonstrar a funcionalidade da ferramenta DelphiToWeb. A ferramenta é usada para gerar código da camada de interface de uma aplicação web a partir de uma aplicação desenvolvida em Delphi. O resultado final da conversão são páginas HTML ou arquivos .LZX. Outra opção dessa ferramenta é a geração de uma representação intermediária em arquivos XML contendo os dados do .DFM. A Figura 24 apresenta a ferramenta DelphiToWeb.

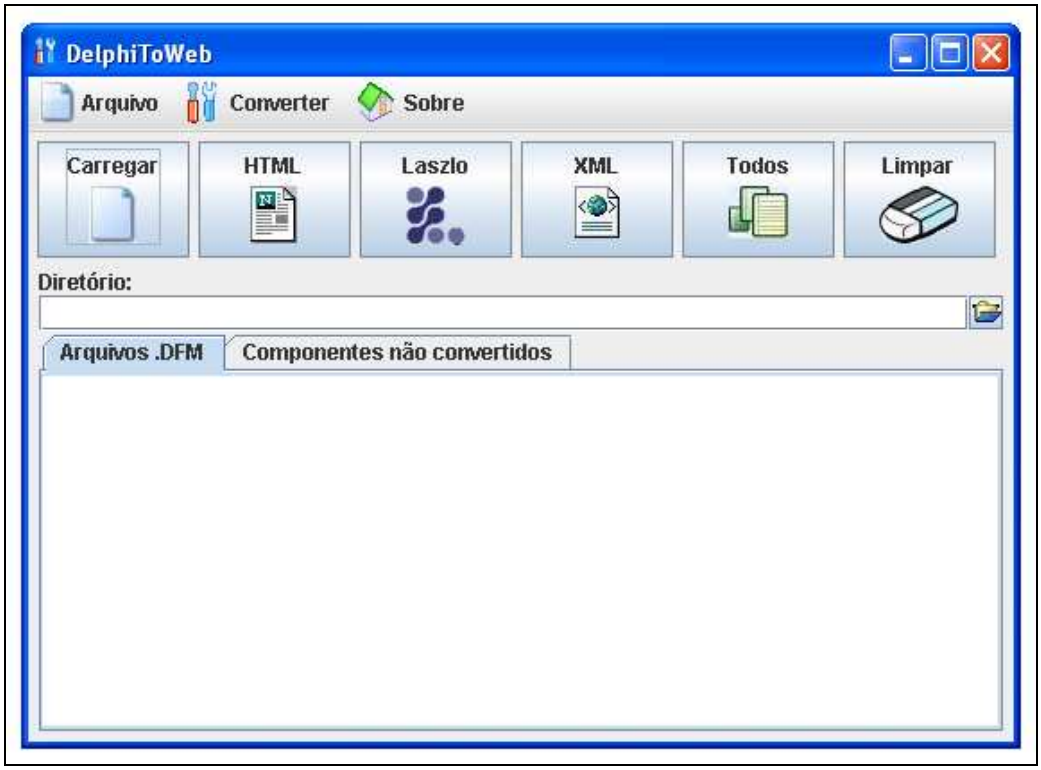


Figura 24 – Tela da ferramenta DelphiToWeb

Ao pressionar o botão **Carregar**, é possível selecionar os arquivos .DFM que serão convertidos (Figura 25). Os arquivos carregados são listados em **Arquivos .DFM** (Figura 26).

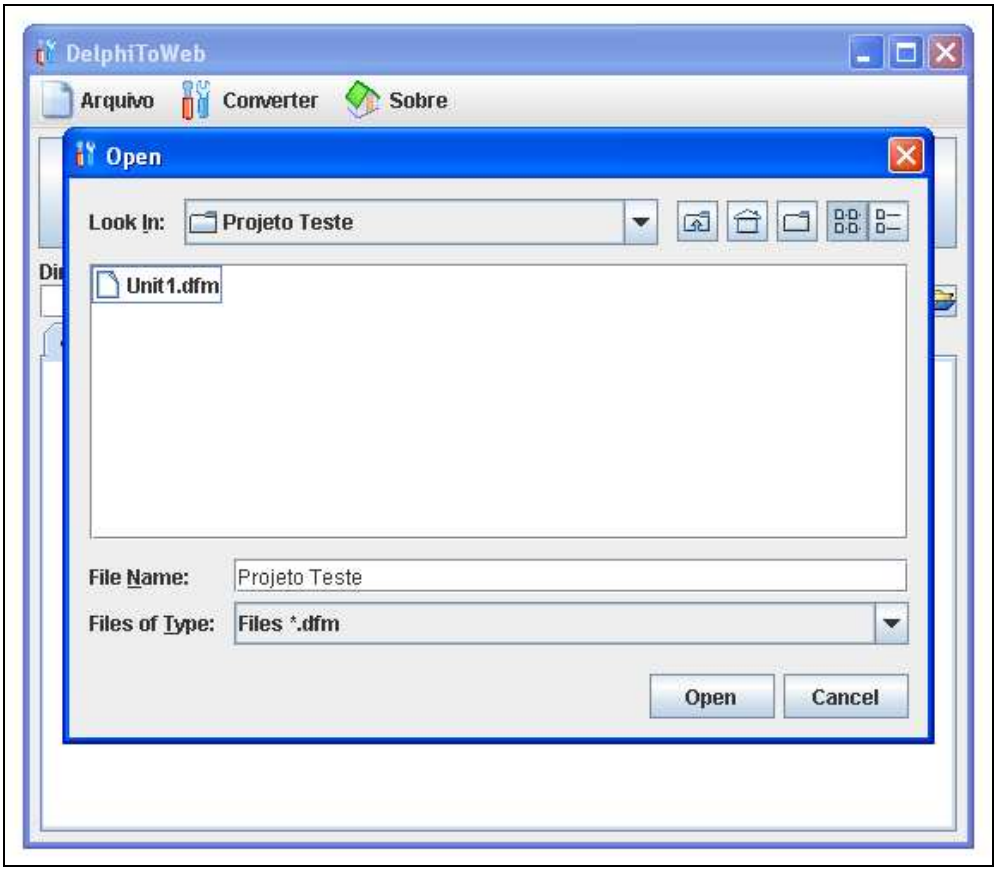


Figura 25 – Seleção de arquivo .DFM

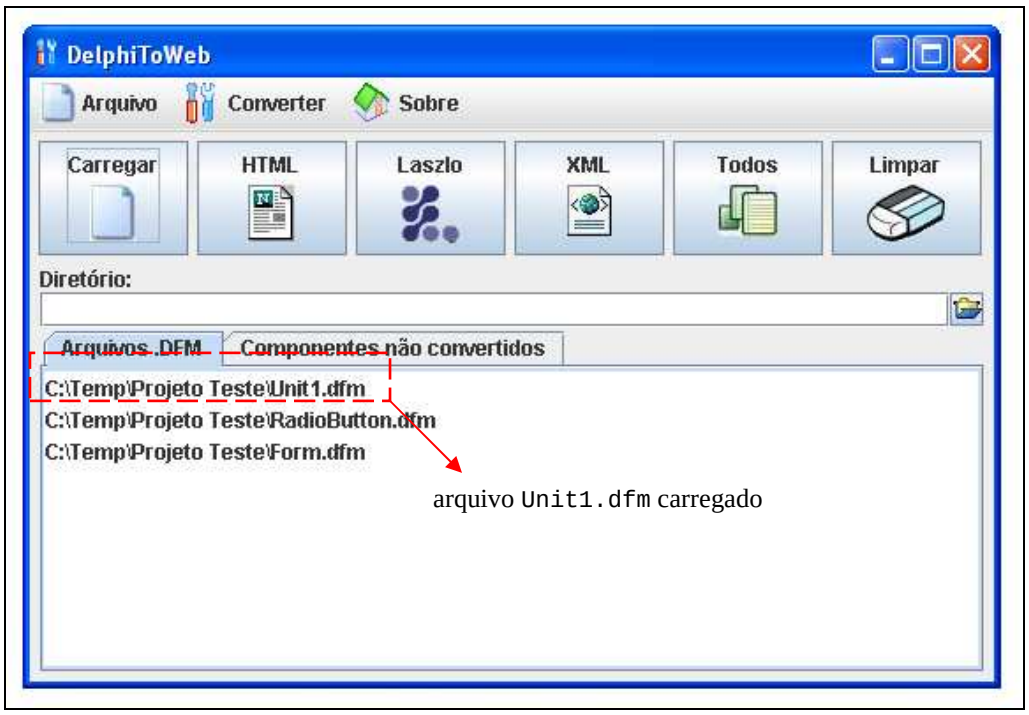


Figura 26 – Arquivos .DFM carregados

Antes de efetuar a conversão, o usuário deve informar o diretório onde deseja que os arquivos sejam criados. Para tanto, pode digitar o diretório no campo correspondente ou

navegar na estrutura de diretórios para selecionar o desejado (Figura 27).

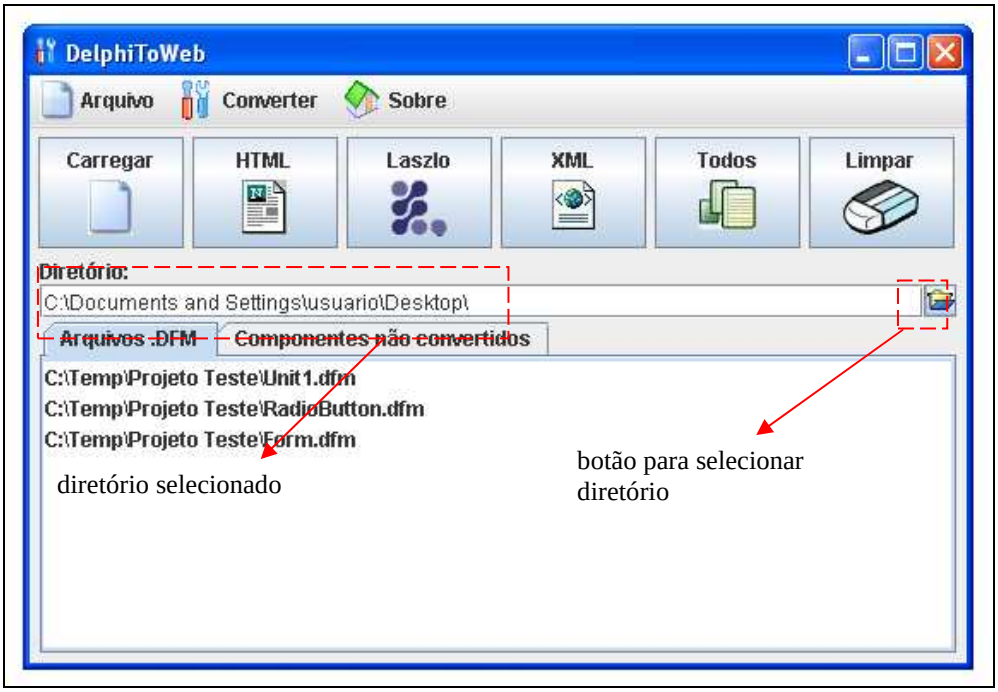


Figura 27 – Diretório selecionado

Os botões **HTML**, **Laszlo**, e **XML** fazem a conversão individualmente do arquivo .DFM para cada um desses formatos. O botão **Todos** converte para os três tipos de uma só vez. Para efetuar a conversão somente para HTML deve-se selecionar o arquivo .DFM, por exemplo `Unit1.dfm`, e clicar no botão **HTML**.

Existem duas situações possíveis: o arquivo .DFM apresentar erros ou o arquivo foi convertido com sucesso. No primeiro caso é apresentada uma mensagem indicando a ocorrência de erro (Figura 28). Não foi feito um diagnóstico preciso porque esse arquivo é gerado automaticamente pelo Delphi e, portanto, não deve possuir erros. Mas serão detectados caso o arquivo tenha sido alterado manualmente de forma incorreta.

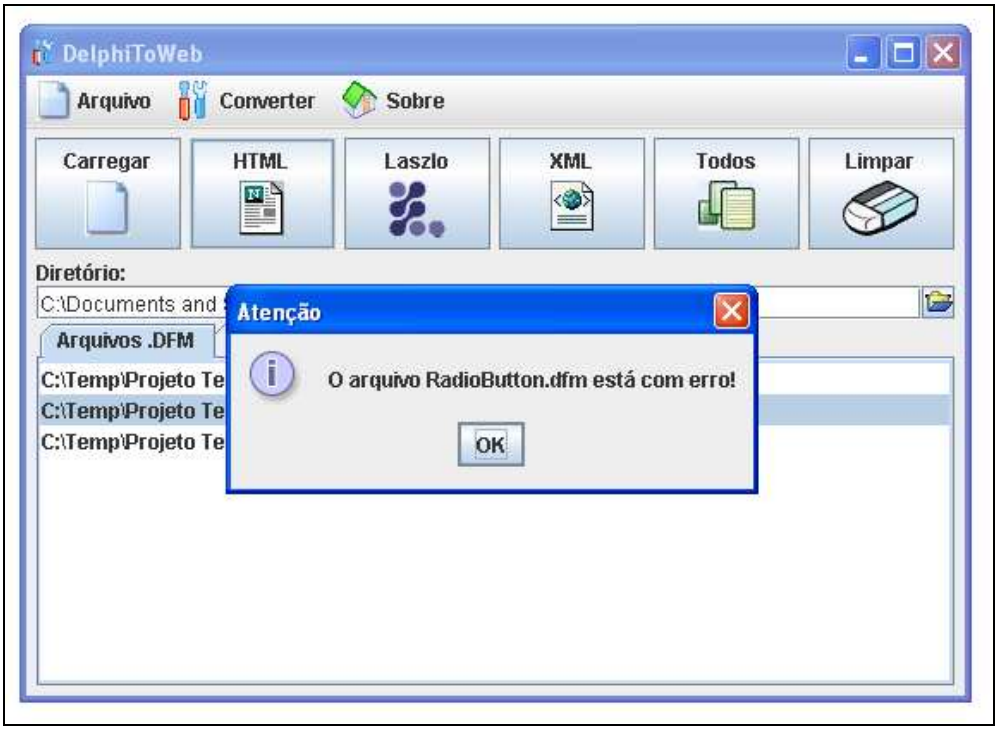


Figura 28 – Mensagem de erro

No segundo caso, é criado um arquivo com o mesmo nome do arquivo .DFM e com extensão .HTML no diretório selecionado (Figura 29). Além disso, os componentes que não são convertidos pela ferramenta são listados em **Componentes não convertidos** (Figura 30).

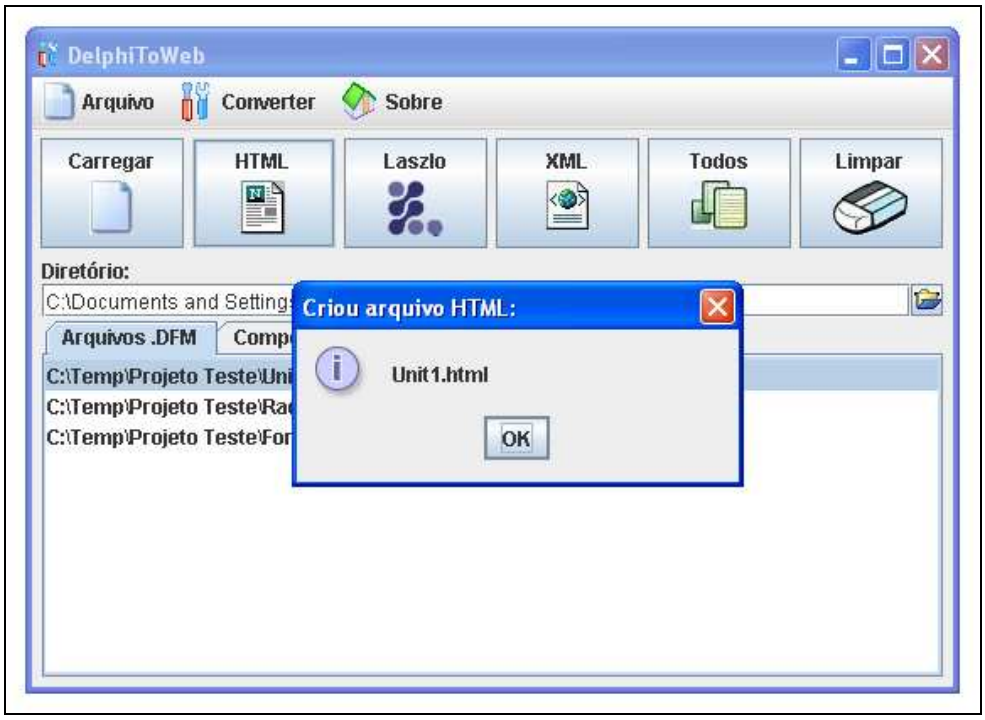


Figura 29 – Conversão para HTML

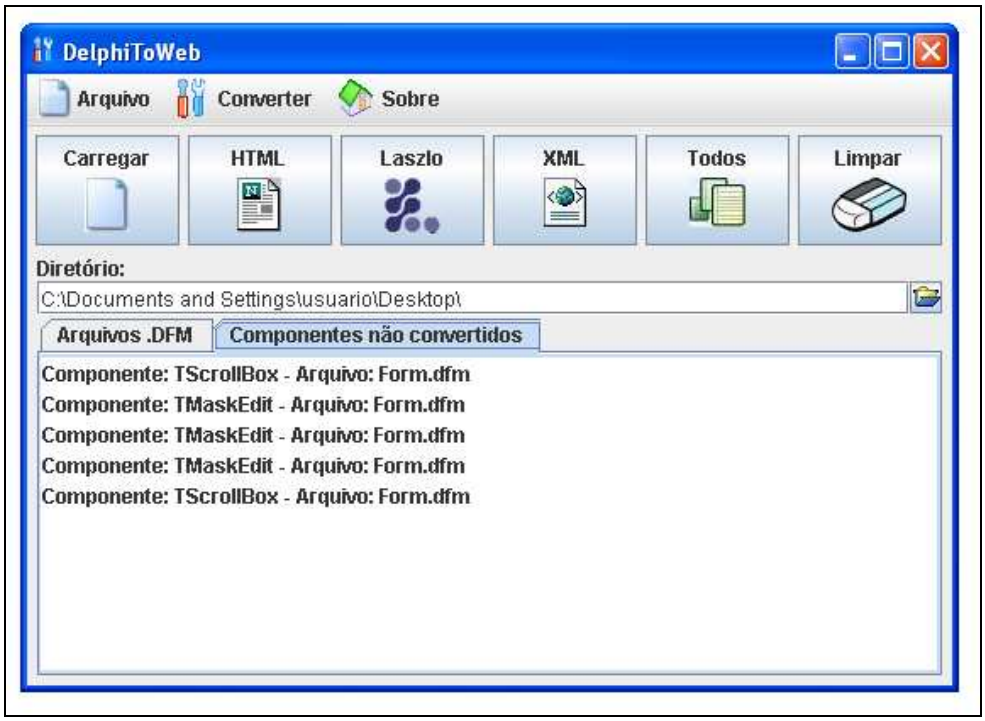


Figura 30 – Componentes não convertidos

O formulário Delphi (`Unit1.dfm`) e a página HTML criada são apresentadas nas figuras a seguir.

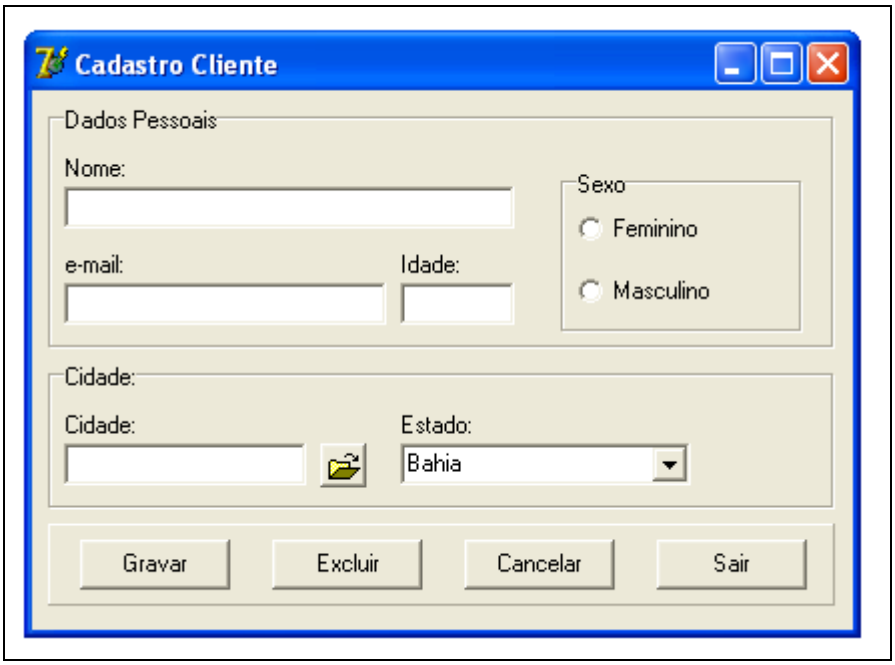


Figura 31 – Formulário Delphi

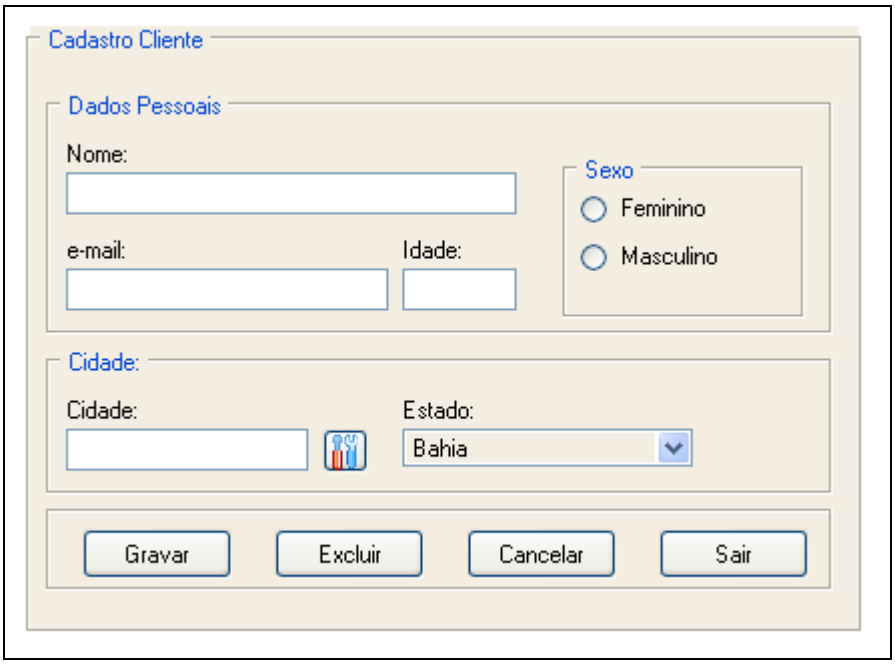


Figura 32 – Resultado da conversão para HTML

O mesmo formulário convertido para Laszlo fica conforme Figura 33.

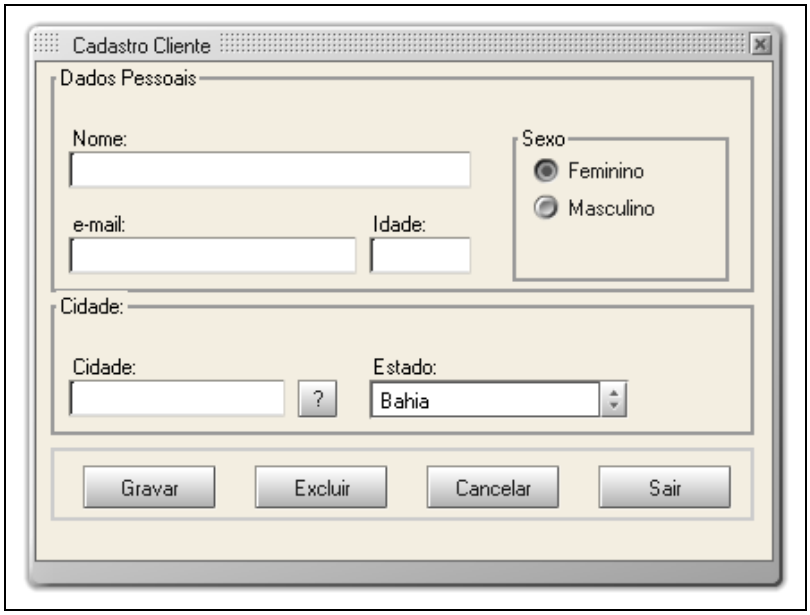


Figura 33 – Resultado da conversão para LZX

Convertendo esse mesmo formulário para XML, o código gerado fica como mostrado no Quadro a seguir.

```
<?xml version="1.0" encoding="UTF-8"?>
<java version="1.5.0_02" class="java.beans.XMLDecoder">
  <object id="Tupla0" class="com.ariana.analisador.Tupla">
     <void property="identify">
        <string>Form1</string>
     </void>
     <void property="propertys">
        <object class="java.util.ArrayList">
           <void method="add">
              <object class="com.ariana.analisador.Property">
                <void property="name"> <string>Left</string> </void>
                <void property="value"> <string>208</string> </void>
              </object>
           </void>

           ...

        </object>
     </void>

      ...

  </object>
</java>
```

Quadro 21 – Resultado da conversão para XML

Para limpar os arquivos .DFM carregados, pode-se utilizar o botão **Limpar**. Na barra de ferramentas tem-se: o menu **Arquivo** com as opções **Carregar** e **Limpar**; o menu **Converter** com opções para converter para **HTML**, **Laszlo**, **XML** e **Todos** e no menu **Sobre** a tela para identificação da ferramenta, mostrada na Figura 34.

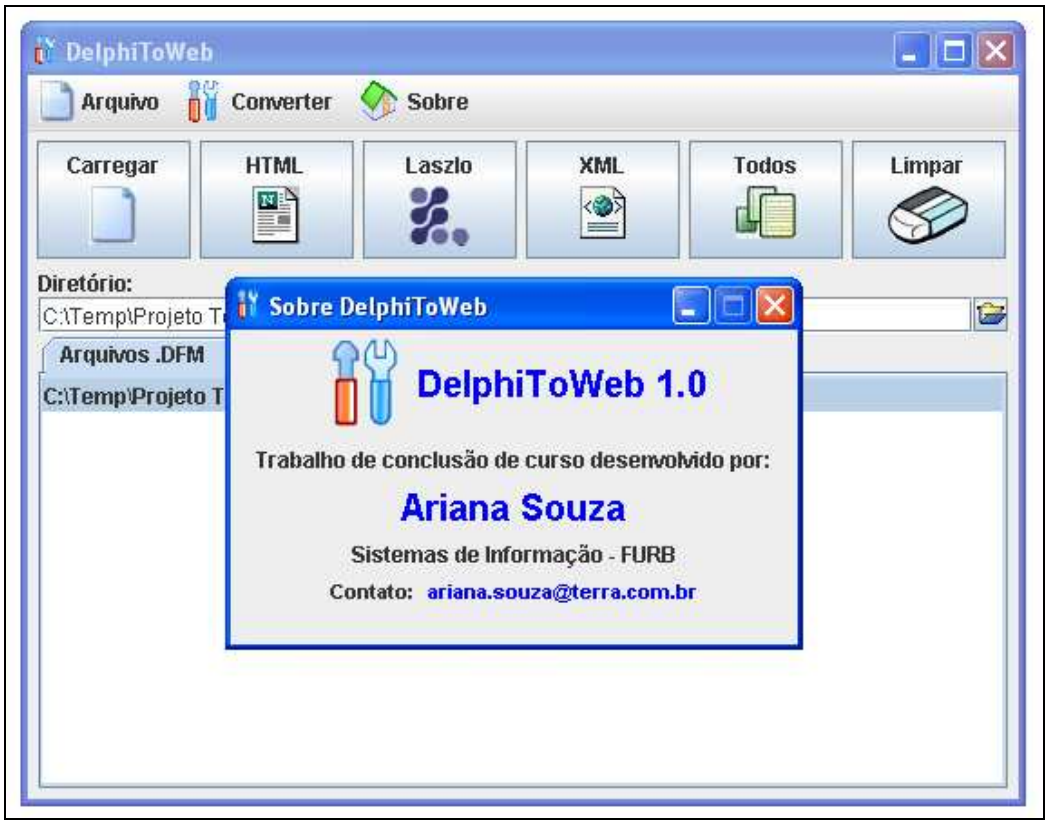


Figura 34 – Tela Sobre

3.6 RESULTADOS E DISCUSSÃO

O presente trabalho seguiu a mesma filosofia do trabalho de Fonseca (2005) no sentido de ser uma ferramenta para conversão de arquivos .DFM. Porém, o resultado final é a produção de interfaces web ao invés da produção de código fonte Java. Salienta-se que foram implementados mais dois requisitos funcionais, além daquele descrito na proposta deste trabalho. Inicialmente, seriam convertidos arquivos .DFM somente para páginas HTML, mas a ferramenta possui mais duas opções de conversão: para Laszlo e para XML.

Ao comparar a ferramenta DelphiToWeb com a ferramenta Delphi2JavaII (FONSECA, 2005), observa-se que a qualidade da conversão em termos de compatibilidade com os formulários Delphi são similares. Essa afirmação foi feita a partir da conversão do formulário usado para o estudo de caso pela ferramenta Delphi2JavaII. O resultado é mostrado na Figura 35.

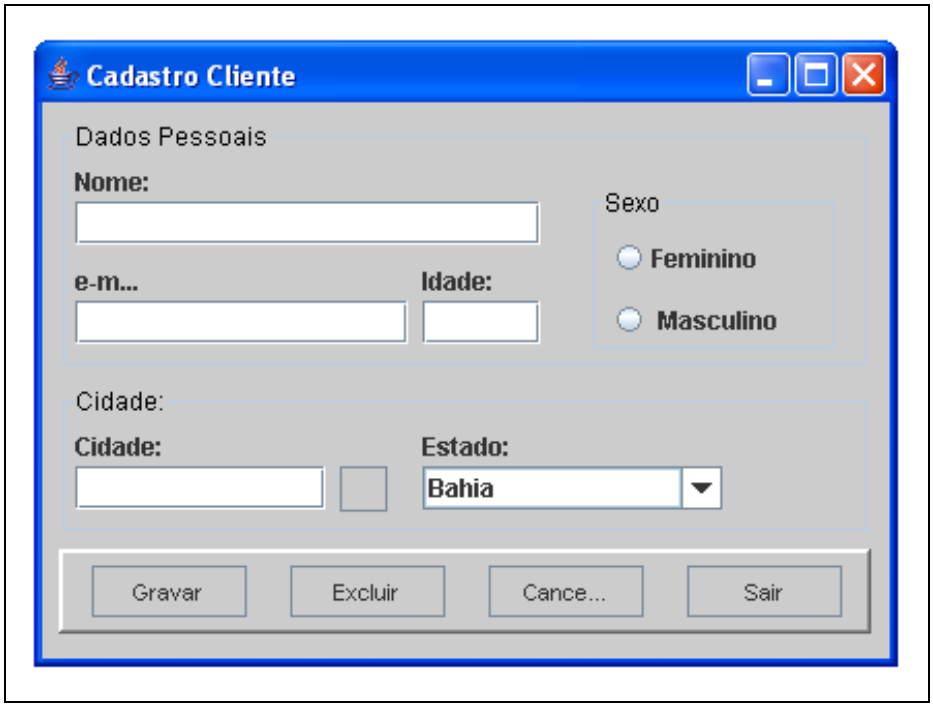


Figura 35 – Resultado da conversão feita pela ferramenta Delphi2Java-II

No Quadro 21 é apresentada uma comparação entre os componentes que foram convertidos do formulário usado para o estudo de caso (Figura 31).

COMPONENTES	DelphiToWeb		Delphi2Java-II
	para HTML	**para Laszlo**	**para SWING**
TForm	correto	correto	correto
TEdit	correto	correto	correto
TGroupBox	correto	correto	margem não visível
TLabel	correto	correto	rótulo incompleto
TPanel	correto	correto	correto
TRadioGroup e seus itens	correto	correto	margem do *TRadioGroup* não visível
TSpeedButton	correto	correto	não tem como saber se possuía imagem
TButton	correto	correto	*caption* incompleto
TComboBox	correto	correto	correto

Quadro 22 – Comparação DelphiToWeb X DelphiToJavaII

Observou-se que a ferramenta DelphiToWeb ao converter o formulário usado no estudo de caso, apresentou uma interface mais próxima do original, conforme análise de cada componente feita no quadro anterior. Portanto, o resultado obtido foi satisfatório apesar dos obstáculos encontrados durante o desenvolvimento do trabalho. Dos vinte e dois componentes convertidos para HTML, somente um deles não foi possível implementar em Laszlo, o `TPopupMenu`. Isso porque ao executar um arquivo .LZX, o servidor Laszlo transforma-o em um arquivo .FLA (do Flash). Assim, ao clicar com o botão direito do *mouse* em um `TPopupMenu` obtém-se o próprio menu do Flash.

4 CONCLUSÕES

DelphiToWeb é uma ferramenta para gerar código com funcionalidade para a camada de interface de uma aplicação web. A ferramenta apresenta-se como uma opção para agilizar o desenvolvimento de aplicações uma vez que permite a migração de produtos desenvolvidos em Delphi para aplicações web. Assim, reduz-se o tempo de elaboração da interface, além de deixar o sistema com o mesmo *layout* da aplicação *desktop*, o que geralmente é uma exigência dos usuários.

Para fazer a migração, a partir de um arquivo com extensão .DFM, contendo toda a informação necessária para construir código para a interface de uma aplicação *desktop*, DelphiToWeb gera páginas HTML utilizando *Cascading Style Sheets* (CSS) e JavaScript, e arquivos .LZX usando a linguagem LZX do Laszlo. No contexto do presente trabalho, os recursos de JavaScript foram utilizados visando complementar as limitações de funcionalidade que os recursos da linguagem HTML apresentam em relação ao conjunto de funcionalidades que os componentes de interface de uma aplicação Delphi dispõem. Por exemplo, o componente `TMainMenu` não possui equivalente em HTML. Assim, a página gerada pela ferramenta insere código JavaScript que implementa esta funcionalidade de modo a permitir que a interface em HTML apresente *layout* equivalente à interface correspondente desenvolvida em Delphi. Os testes feitos resultaram em interfaces bem próximas às interfaces Delphi.

DelphiToWeb traz também a opção de conversão do arquivo .DFM para XML com a intenção de permitir troca de dados. Com a conversão para XML o usuário poderá utilizar as informações do arquivo .DFM para quaisquer fins..

A ferramenta atingiu todos os objetivos propostos para o desenvolvimento deste trabalho, agregando conhecimentos em HTML, CSS, JavaScript, Laszlo, XML e analisadores

de linguagens. A ferramenta GALS, utilizada para a geração dos analisadores léxico e sintático, facilitou o desenvolvimento do trabalho, demonstrando ser uma boa opção para geração desses analisadores para qualquer linguagem.

4.1 EXTENSÕES

Como extensões para este trabalho sugere-se: implementar a conversão de mais componentes (a ferramenta está limitada a 22 componentes), preparar o código para implementação dos eventos e converter o código fonte dos eventos. Além disso, também pode ser utilizado o *Design Pattern Visitor*, para modelar as classes dos componentes, visando facilitar a reutilização do código quando for necessária a geração de um novo formato de arquivo de saída.

REFERÊNCIAS BIBLIOGRÁFICAS

BRILLINGER, W.; EPP, B. **JavaScript**. Tradução Júlio César Cunha. [S.l.], [2005?]. Disponível em: <http://carteiro.cena.usp.br/openwebmail/help/pt_BR/def/javascript.html>. Acesso em: 02 abr. 2005.

CHAGAS, W. **Compiladores**: como surge um programa binário a partir de if's e while's. [S.l.], [2003]. Disponível em: <http://www.geocities.com/SiliconValley/Bay/1058/compiler.html>. Acesso em: 06 nov. 2005.

COLIBRI, F. J. **DFM Parser**. [S.l.], [2005]. Disponível em: <http://www.felix-colibri.com/papers/colibri_utilities/dfm_parser/dfm_parser.html>. Acesso em: 18 ago. 2005.

CONTI, F. **A linguagem HTML**. [Pará], 2005. Disponível em: <http://www.cultura.ufpa.br/dicas/htm/htm-intr.htm>. Acesso em: 08 maio 2005.

DEITEL, H. M. et al. **XML**: como programar. Tradução Luiz Augusto Salgado e Edson Furmankiewicz. Porto Alegre: Bookman, 2003.

DEVMEDIA. **Introdução à linguagem HTML**. [S.l.], 2004. Disponível em: <http://www.infodesktop.com/web/html/artigocompleto/1>. Acesso em: 08 maio 2005.

DMSNET. **Migração/conversão de sistemas para Java/J2EE e Microsoft.NET**. [S.l.], 2004. Disponível em: <http://www.dmsnet.com.br/migracao_sistemas.php>. Acesso em: 02 abr. 2005.

FONSECA, F. **Ferramenta conversora de interfaces gráficas**: Delphi2 Java-II. 2005. 59 f. Trabalho de Conclusão de Curso (Bacharelado em Ciências da Computação) - Centro de Ciências Exatas e Naturais, Universidade Regional de Blumenau, Blumenau.

GESSER, C. E. **GALS**: gerador de analisadores léxicos e sintáticos. 2003. 150 f. Trabalho de Conclusão de Curso (Bacharelado em Ciência da Computação) – Centro Tecnológico, Universidade Federal de Santa Catarina, Florianópolis.

HERRINGTON, Jack. **Code generation in action.** California: Manning, 2003.

LASZLO SYSTEMS. **Software engineer's guide to developing OpenLaszlo applications**. [S.l.], 2005. Disponível em: <http://www.laszlosystems.com/lps-3.1/docs/guide/>. Acesso em: 8 nov. 2005.

LEITE, J. C. **Desenvolvimento de interfaces de usuário de sistemas web**. [S.l.], [2002?]. Disponível em: <http://www.dimap.ufrn.br/~jair/diuweb/>. Acesso em: 20 nov. 2005.

LINHA DE CÓDIGO. **Microsoft lança conversor de linguagem Java para C#**. [S.l.], 2002. Disponível em: <http://www.linhadecodigo.com.br/noticias.asp?id_noticia=194>. Acesso em: 30 abr. 2005.

LOURENÇO, R. A. **Aprenda JavaScript**. [S.l.], [2005?]. Disponível em: <http://www.magnet.com.br/classic/mao/javascript.html>. Acesso em: 07 maio 2005.

MARCHAL, B. **XML**: conceitos e aplicações. Tradução Daniel Vieira. São Paulo: Berkeley, 2000.

PRICE, A. M. A.; TOSCANI, S. S. **Implementação de linguagens de programação**: compiladores. 2. ed. Porto Alegre: Sagra Luzzatto, 2001.

SASSE, E. **Convertendo arquivos DFM binários para texto**. [S.l.], [2005?]. Disponível em: <http://www.clubedelphi.net/Novo/Colunistas/Erick_Sasse/02.asp>. Acesso em: 8 maio 2005.

SOARES FILHO, A. **Migrar software reduz custos e economiza tempo**. [S.l.], [2003]. Disponível em: <http://webinsider.uol.com.br/vernoticia.php/id/1844>. Acesso em: 27 nov. 2005.

UNIVERSIDADE ESTADUAL DE MARINGÁ. **Especialização em desenvolvimento de sistemas para web**. [Maringá], [2005]. Disponível em: <http://www.bs2.com.br/espweb/descricao.html>. Acesso em: 20 nov. 2005.

WIKIPEDIA. **HyperText Markup Language**. [S.l.], 2005. Disponível em: <http://pt.wikipedia.org/wiki/HTML>. Acesso em: 08 maio 2005.