UNIVERSIDADE REGIONAL DE BLUMENAU

CENTRO DE CIÊNCIAS EXATAS E NATURAIS

CURSO DE CIÊNCIAS DA COMPUTAÇÃO – BACHARELADO

EXTENSÃO DA FERRAMENTA DELPHI2JAVA-II PARA SUPORTAR A TRADUÇÃO DOS TRATADORES DE EVENTOS

CARLOS HENRIQUE MARIAN

BLUMENAU
2008

2008/1-05

CARLOS HENRIQUE MARIAN

EXTENSÃO DA FERRAMENTA DELPHI2JAVA-II PARA SUPORTAR A TRADUÇÃO DOS TRATADORES DE EVENTOS

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 Ciências da Computação — Bacharelado.

Prof. Mauro Marcelo Mattos – Dr. - Orientador

BLUMENAU
2008

2008/1-05

EXTENSÃO DA FERRAMENTA DELPHI2JAVA-II PARA SUPORTAR A TRADUÇÃO DOS TRATADORES DE EVENTOS

Por

CARLOS HENRIQUE MARIAN

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

Presidente:	Prof. Mauro M. Mattos Orientador, Doutor – Orientador, FURB
Membro:	Prof. Joyce Martins, Mestre – FURB
Membro:	Prof. Wilson P. Carli, Mestre – FURB

Dedico este trabalho a todos os familiares e amigos, especialmente aqueles que me ajudaram diretamente na realização deste.

AGRADECIMENTOS

A Deus, pelo seu imenso amor e graça.

À minha família que sempre esteve presente.

Aos meus amigos, pelos empurrões e cobranças.

Ao meu orientador, Prof. Mauro Marcelo Mattos, por ter acreditado na conclusão deste trabalho.

“Uma longa viagem começa com um único passo”

Lao Tsé

RESUMO

Este trabalho apresenta um protótipo desenvolvido em forma de ferramenta, para realizar a conversão dos tratadores de eventos de um código fonte desenvolvido em *Object* Pascal para código fonte expresso na linguagem Java. A partir da análise do fonte Pascal é realizado um mapeamento para uma metalinguagem a qual mantém informações sobre o fonte Pascal necessárias para a realização da conversão do código para Java.

Palavras-chave: Delphi2Java-II. Tradução Object Pascal para Java.

ABSTRACT

Abstract This work describes a software prototype to translate programs written in Object Pascal source code to Java source code. In order to make this conversion, a meta-language was specified and stores all information about the Pascal source code necessary to make the translation available.

Key-words: Delphi2Java-II. Object Pascal to Java Translation.

LISTA DE ILUSTRAÇÕES

Quadro 1 - Modelo formatador de código ou *code munger* 16
Quadro 2 - Modelo de código expandido ou *inline-code expander* 17
Quadro 3 - Modelo de código misto ou *mixed-code generator* 18
Quadro 4 - Modelo parcial de classes ou *partial-class generator* 18
Quadro 5 - Modelo de camadas da aplicação ou *tier generator* 19
Quadro 6 - Exemplo de definição de *tokens* 20
Quadro 7 - Trecho de uma BNF que define o Delphi 21
Figura 1 - Formulário em Delphi 22
Quadro 8 - Estrutura do arquivo .DFM 23
Quadro 9 - Definição de classe formulário em Delphi 23
Quadro 10 - Código fonte do arquivo .PAS 24
Quadro 11 - Principais eventos 25
Figura 2 - Hierarquia dos principais *listeners* 26
Quadro 12 - Trecho de código do uso de eventos 26
Figura 3 - Tela do Delphi2Java-II 27
Quadro 13 - Especificação dos *tokens* 28
Quadro 14 - Gramática 28
Figura 4 - Classes do Delphi2Java-II para geração de código 29
Quadro 15 - Metodo `geraArquivoConversao` da classe `Conversor` 29
Figura 5 - Formulário Delphi para conversão 30
Figura 6 - Formulário convertido pela ferramenta Delphi2Java-II 30
Quadro 16 - Assinaturas dos eventos gerados pela ferramenta Delphi2Java-II 30
Figura 7 - Diagrama de caso de uso 33
Quadro 17 - Detalhamento dos casos de uso 33
Figura 8 - Diagrama de Atividades da ferramenta 34
Figura 9 - Diagrama de classes principal do pacote `classes` 36
Figura 10 - Diagrama de classes secundário do pacote `classes` 38
Quadro 18 - BNF que define a metalinguagem 40
Quadro 19 - Exemplo de código fonte Delphi 41
Quadro 20 - Metalinguagem gerada da `Unit1` 41
Figura 11 - Diagrama de pacotes 42

Figura 12 - Diagrama de classes do pacote parserarquivo 43
Quadro 21 - Relação de métodos da classe NovoLexico 44
Figura 13 - Diagrama de classes do pacote compilador 45
Figura 14 - Diagrama de classes secundário do pacote classes.DelphiToJava 46
Figura 15 - Diagrama de classes secundário do pacote Interface 47
Quadro 23 - Método setListaArquivos da classe ConverteArquivo 50
Quadro 24 - Tratamento de um *token* chave para interface 51
Quadro 25 - Trecho de código do método processaUses 51
Quadro 26 - Gramática para identificar uses e variáveis 52
Quadro 27 - Rotina que busca o trecho de código do type 52
Quadro 28 - Trecho de código do método getTrechoMetodo 54
Quadro 29 - Código do método getMetodo da classe TInterface 55
Quadro 30 - Trecho de código do método trataComando 56
Quadro 31 - *Tokens* de comandos 57
Quadro 32 - Código fonte do método ConverteArquivos da classe Gerador 59
Quadro 33 - Código fonte do método getClasses da classe ArquivoJava 59
Quadro 34 - Fonte Delphi para conversão 60
Quadro 35 - Método que será escrito no arquivo com as assinaturas dos eventos 60
Quadro 36 - Código fonte da classe util 61
Quadro 37 - Arquivo texto com informaçãos obtidas na leitura do fonte Delphi 61
Figura 16 - Interface da ferramenta Delphi2Java-II 62
Figura 17 - Botões de ação da interface 63
Figura 18 - Selecionar arquivos DFM 63
Figura 19 - Escolhe o local onde os arquivos serão gerados 64
Figura 20 - Mensagem de fim da conversão 65

LISTA DE SIGLAS

BNF – *Backus-Naur-Form*

DFM – *Delphi Form File*

EA – *Entreprise Architect*

FURB – Universidade Regional de Blumenau

HTML – *HyperText Markup Language*

IDE – *Integrated Development Environment*

J2EE – Java 2 *Enterprise Edition*

JDK – Java *Development Kit*

MFC – *Microsoft Foundation Classes*

PL/SQL – *Procedural Language/Structured Query Language*

UML – *Unified Modeling Language*

SUMÁRIO

1 INTRODUÇÃO ... **13**
1.1 OBJETIVOS DO TRABALHO ... 14
1.2 ESTRUTURA DO TRABALHO ... 14
2 REVISÃO BIBLIOGRÁFICA ... **15**
2.1 GERAÇÃO DE CÓDIGO ... 15
2.1.1 Compilador ... 19
2.2 ARQUIVO .DFM ... 22
2.3 DEFINIÇÃO DA CLASSE QUE DEFINE O FORMULÁRIO ... 23
2.4 TRATADORES DE EVENTOS EM DELPHI ... 24
2.5 TRATADORES DE EVENTOS EM JAVA ... 25
2.6 DELPHI2JAVA-II ... 26
2.7 TRABALHOS CORRELATOS ... 31
3 DESENVOLVIMENTO ... **32**
3.1 REQUISITOS PRINCIPAIS DO PROBLEMA A SER TRABALHADO ... 32
3.2 ESPECIFICAÇÃO ... 32
3.2.1 Metalinguagem ... 35
3.2.2 Diagrama de pacotes ... 42
3.3 IMPLEMENTAÇÃO ... 47
3.3.1 Técnicas e ferramentas utilizadas ... 48
3.3.1.1 Processo inicial ... 48
3.3.1.2 Análise do fonte Delphi ... 49
3.3.1.3 Estrutura dos arquivos gerados ... 57
3.3.1.4 Geração do código Java ... 58
3.3.2 Operacionalidade da implementação ... 62
3.4 RESULTADOS E DISCUSSÃO ... 65
4 CONSIDERAÇÕES FINAIS ... **67**
4.1 EXTENSÕES ... 67
5 REFERÊNCIAS BIBLIOGRÁFICAS ... **69**
APÊNDICE A – Formulário Delphi. ... **72**
APÊNDICE B – Interface gráfica convertida pela ferramenta Delphi2Java-II ... **73**
APÊNDICE C – Classe com os tratadores de eventos. ... **75**

APÊNDICE D – Classe `Util`. 76
APÊNDICE E – Arquivo texto com os itens identificados no arquivo Delphi.................77

1 INTRODUÇÃO

Segundo Borba (2000), dentre as vantagens do desenvolvimento de aplicações usando Delphi está a facilidade e rapidez para criação de interfaces gráficas, através do uso de componentes, que proporcionam uma maior produtividade. A desvantagem do Delphi é seu estilo de programação que dificulta o reuso devido ao fato de induzir o programador a colocar a lógica de negócio junto aos tratadores de eventos.

Atualmente duas plataformas de desenvolvimento estão em grande destaque, são elas *Java 2 Enterprise Edition* (J2EE) e Microsoft .NET. Segundo Gartner (2003 apud César, 2003), juntas, as tecnologias terão 80% ou mais do mercado de desenvolvimento de aplicações de *e-business* até 2008.

Ainda no debate quanto às plataformas .NET e Java, apesar da alta produtividade oferecida pelo framework .NET — contando com a forte ajuda que representa o *Integrated Development Environment* (IDE) Microsoft Visual Studio.NET —, uma grande fraqueza de .NET em relação a Java é estar atrelado ao sistema Windows, enquanto Java é suportado em múltiplas plataformas (ALVES, 2007).

Seguindo estas tendências, a migração de antigas plataformas de software para as tecnologias Java e .NET é uma saída para as empresas evitarem a defasagem dos seus sistemas (SOARES FILHO, 2003). Mas a tarefa de reescrever exige que todo o investimento original seja refeito. Este processo toma muito tempo e em geral consome uma verba de investimento que já não existe na maioria dos orçamentos das empresas.

Um protótipo de software vem sendo desenvolvido desde 2005 e foi batizado como Delphi2Java-II numa referência explícita a sua funcionalidade: converter formulários Delphi para Java. A proposta de nome do projeto originalmente foi DelphiToJava. Contudo, durante a fase de levantamento bibliográfico para a fundamentação teórica do pré-projeto, foram encontradas diversas referências na internet para um projeto denominado Delphi2Java (WINSITE, 1997). O protótipo Delphi2Java-II passou por diversas fases e hoje permite a conversão de uma série de componentes visuais e de acesso a dados de aplicações desenvolvidas em Delphi para linguagem Java.

Diante da dificuldade que as empresas possuem para migrar suas ferramentas, este projeto visa acrescentar à ferramenta Delphi2Java-II a funcionalidade de conversão dos tratadores de eventos do ambiente Delphi para a plataforma Java.

1.1 OBJETIVOS DO TRABALHO

O objetivo deste trabalho é aprimorar a ferramenta Delphi2Java-II adicionando a ela a funcionalidade de conversão do código do tratador de eventos de interface do ambiente Delphi para Java.

Os objetivos específicos do trabalho são:

a) identificar através da leitura do arquivo *Delphi Form File* .DFM os eventos `OnClick`, `OnChange`, `OnKeyUp`, `OnKeyPress` e `OnEnter` dos componentes já suportados pela ferramenta Delphi2Java-II;
b) localizar nos arquivos .PAS do projeto os respectivos códigos dos tratadores de eventos e efetuar a sua conversão para Java;
c) analisar a compatibilidade entre o código gerado em Java e o código original através de um estudo de casos desenvolvido em Delphi, apontando eventuais discrepâncias quando for o caso.

1.2 ESTRUTURA DO TRABALHO

Esta monografia divide-se em fundamentação teórica, desenvolvimento do trabalho, e conclusões. O primeiro capítulo apresenta uma introdução do trabalho, os objetivos a serem apresentados e a estrutura do trabalho.

O segundo capítulo contempla a fundamentação teórica do trabalho e descreve a estrutura do arquivo DFM, tratadores de evento em Delphi e Java, geração de código e trabalhos correlatos.

Os requisitos do sistema, diagramas de caso de uso, de pacotes e de classes, juntamente com detalhes sobre a implementação, técnicas e ferramentas utilizadas são apresentadas no terceiro capítulo. Neste capítulo também são exibidos alguns resultados e estudos realizados durante o trabalho.

O quarto capítulo descreve as considerações finais sobre o trabalho, objetivos alcançados, os não alcançados, incluindo sugestões para extensões em trabalhos futuros.

2 REVISÃO BIBLIOGRÁFICA

Neste capítulo são apresentadas as principais características do projeto, descritas em tópicos como: geração de código, arquivo .DFM, definição da classe que define o formulário, os tratadores de eventos em Delphi, tratadores de eventos em Java, ferramenta Delphi2Java-II, além de trabalhos correlatos.

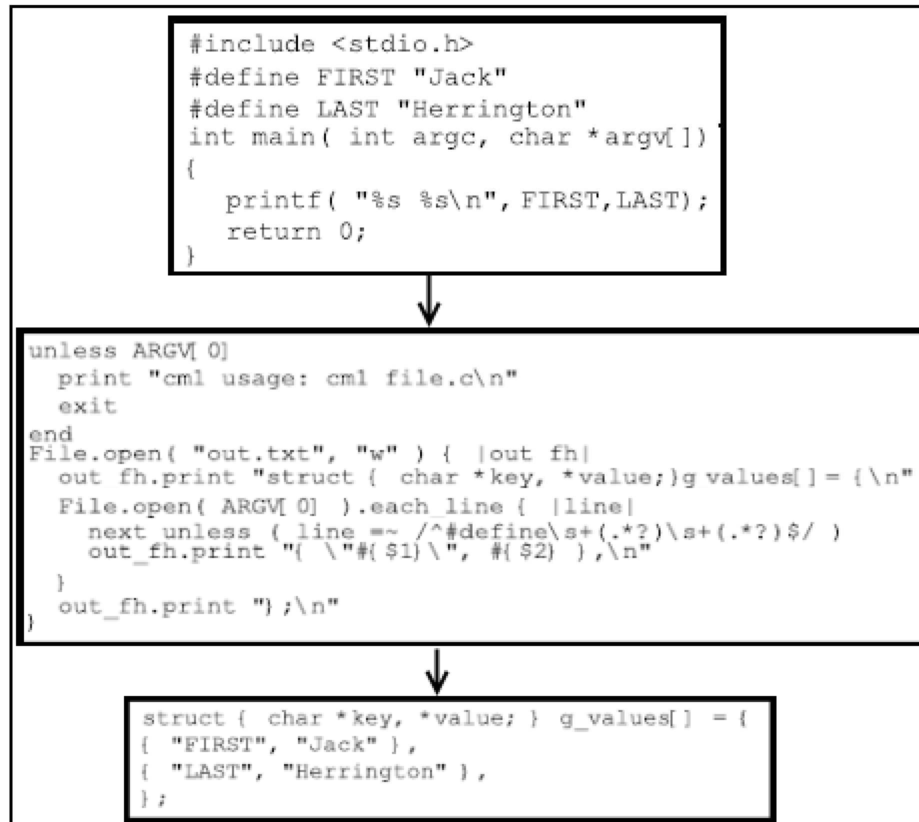
2.1 GERAÇÃO DE CÓDIGO

Conforme Herrington (2003 apud Souza 2005, p. 15),

> a 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.

As ferramentas para geração de código são classificadas por Herrington (2003 apud Souza 2005) em cinco modelos, que variam de acordo com o uso, as entradas e as saídas.

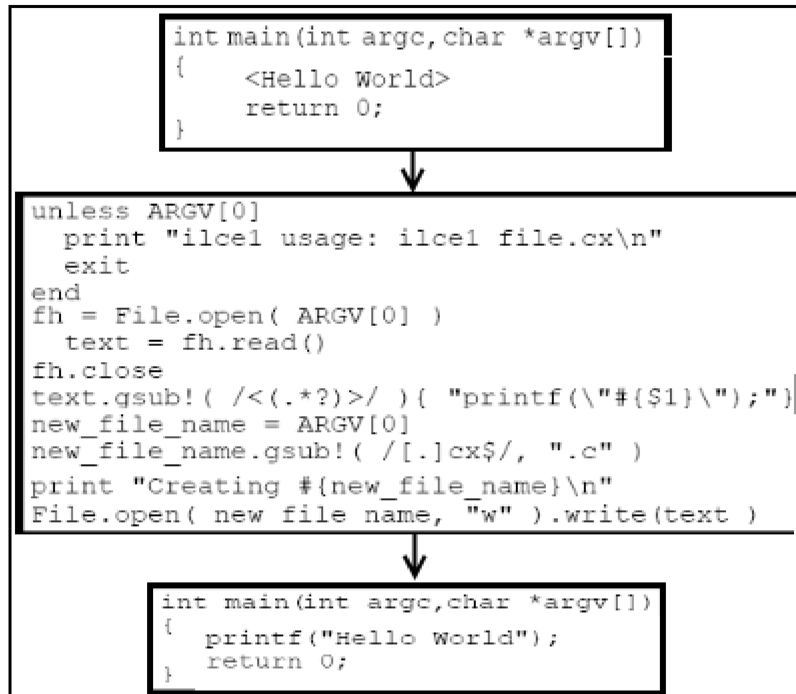
O primeiro modelo é o formatador de código ou *code munger*. Este modelo tem como entrada código fonte escrito em uma linguagem de alto nível (por exemplo, C, Java, C++) e como saída um ou mais arquivos, que podem ser documentação ou extensões do código de entrada (Quadro 1).



Fonte: adaptado de Herrington (2003, p. 63).

Quadro 1 - Modelo formatador de código ou *code munger*

O modelo gerador de código expandido ou *inline-code expander* tem o objetivo de identificar marcadores especiais no código de entrada e gerar código fonte de alto nível com base nas informações contidas nos marcadores como mostra o Quadro 2. A *tag* `<Hello Word>` do código de entrada é convertida para `printf("Hello Word");`.



```
int main(int argc,char *argv[])
{
     <Hello World>
     return 0;
}
```

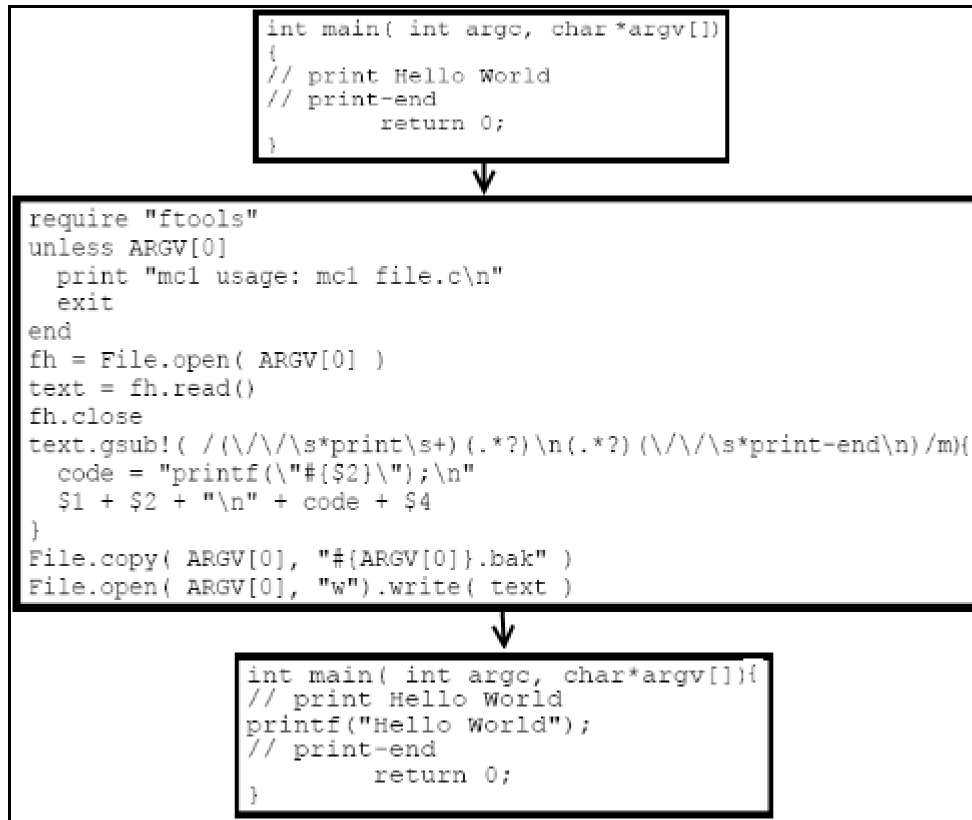
```
unless ARGV[0]
  print "ilce1 usage: ilce1 file.cx\n"
  exit
end
fh = File.open( ARGV[0] )
  text = fh.read()
fh.close
text.gsub!( /<(.*?)>/ ){ "printf(\"#{$1}\");"}
new_file_name = ARGV[0]
new_file_name.gsub!( /[.]cx$/, ".c" )
print "Creating #{new_file_name}\n"
File.open( new file name, "w" ).write(text )
```

```
int main(int argc,char *argv[])
{
  printf("Hello World");
  return 0;
}
```

Fonte: adaptado de Herrington (2003, p. 80).

Quadro 2 - Modelo de código expandido ou *inline-code expander*

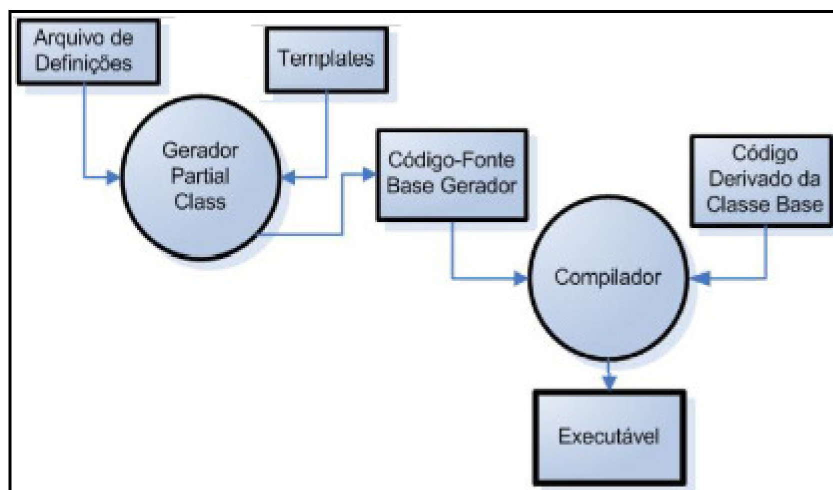
O modelo gerador de código misto ou *mixed-code generator* é uma implementação mais refinada do modelo de gerador de código expandido que, através da identificação de marcadores especiais, constrói o novo código usando como saída o mesmo código de entrada (modelo no Quadro 3). O modelo mostra que o que estiver entre o trecho `//print` e `//print-end` será reescrito como parâmetro de um comando `printf` no próprio fonte de entrada a rotina de impressão `printf("Hello Word");`.



Fonte: adaptado de Herrington (2003, p. 84).

Quadro 3 - Modelo de código misto ou *mixed-code generator*

Outro modelo é o gerador parcial de classes ou *partial-class generator* que constrói códigos a partir de modelos abstratos, como descreve o Quadro 4. Segundo Ferreira (2005, p. 9), o gerador *partial-class* é responsável por gerar parte de uma camada de aplicação. Um exemplo típico é na geração de camadas de acesso a dados. O gerador cria um código de persistência para cada classe e seus campos.

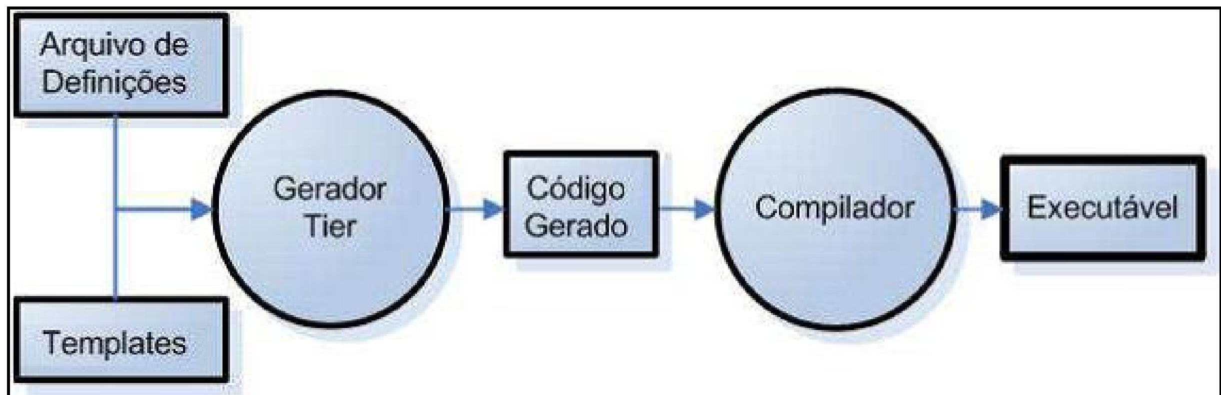


Fonte: adaptado de Ferreira (2005, p. 10).

Quadro 4 - Modelo parcial de classes ou *partial-class generator*

Por fim, no modelo gerador de camadas da aplicação ou *tier generator* a entrada é uma definição abstrata e para gerar a saída são utilizados *templates*. O código gerado pode ser

integrado diretamente ao projeto (Quadro 5). Segundo Ferreira (2005, p.,10), o gerador recebe o arquivo com as definições abstratas e o conjunto de *templates* que serão utilizados para produzir a saída.



Fonte: adaptado de Ferreira (2005, p. 10).

Quadro 5 - Modelo de camadas da aplicação ou *tier generator*

Dependendo do escopo, as definições de entrada podem ser tanto representações abstratas quanto um código em alguma linguagem de programação e a saída depende do propósito do projeto e envolve desde uma extensão do código de entrada até uma implementação completa de uma aplicação.

De acordo com Jantal (2006), existem dois grandes grupos de geradores de código: ativos e passivos. Os geradores de código passivos produzem *templates* de código que devem ser preenchidos manualmente. A versão do projeto Delphi2Java-II descrita em Fonseca (2005) enquadra-se nesta categoria, pois o código dos tratadores de evento gerados constituem-se numa declaração de método que atenderá aos eventos de interface para os botões convertidos. Os geradores de código ativo produzem código automaticamente e são responsáveis pela manutenção dos mesmos em situações de alteração nas definições.

2.1.1 Compilador

Delamaro (2004, p. 3) define que a construção de um compilador é dividida em partes com funções especificas e que podem ser divididas em:

a) o analisador léxico;
b) o analisador sintático;
c) o analisador semântico;
d) o gerador de código.

Segundo Delamaro (2004, p. 3), o analisador léxico tem a funcionalidade de separar no

programa fonte cada um dos símbolos que tenham algum significado para a linguagem ou de alertar quando algum dos símbolos encontrados não pertence à linguagem.

Esta separação do código fonte tem o objeto de produzir uma seqüencia de *tokens*. Os *tokens* podem ser classificados como palavras reservadas, identificadores, símbolos especiais, constantes de tipos básicos (inteiro, real, literal), entre outras categorias e são definidos através de expressões regulares (MARCOS, 2007, p. 20) (demonstrado no Quadro 6).

```
letra : [a-zA-Z]
digito : [0-9]
identificador : {letra} ({letra}|{digito})*
contante_inteira : (“+”|”-”)? {digito}+
```

Fonte: Marcos (2007, p. 20).

Quadro 6 - Exemplo de definição de *tokens*

No Quadro 6 é possível observar as seguintes expressões regulares: `letra` representa um alfabeto composto por letras maiúsculas e minúsculas; `digito`, representa os dígitos entre 0 e 9; `identificador` descreve uma seqüência que inicia com uma letra seguida ou não (*) de um conjunto de letras ou dígitos; e `constante_inteira` que inicia ou não (?) com o sinal unário seguido de um ou mais (+) `digito`.

O analisador sintático tem a responsabilidade de verificar a seqüência dos símbolos contida no programa fonte e ser capaz de analisar esse programa fonte e reconhecê-lo com válido (DELAMARO, 2004, p. 4).

Ainda segundo Delamaro (2004, p. 6), o analisador semântico deve verificar se os aspectos semânticos do programa estão corretos, buscando verificar se não existe incoerência quanto ao significado das construções utilizadas no programa. Para esta verificação, o analisador semântico não faz verificação no fonte do programa, mas sim na árvore sintática. Após as verificações sintáticas e semânticas o compilador pode iniciar a geração de código.

O Quadro 7 apresenta uma parte da especificação da linguagem Delphi expressa na notação *Backus-Naur-Form* (BNF). A descrição completa (mas não oficial) da linguagem contém 13 páginas o que introduz um nível de complexidade bastante importante em qualquer no processo de análise/conversão de código Delphi (BRINK, 2006).

```
{…}
<VarSection>  ::= VAR <VarDeclList>
               | THREADVAR <ThreadVarDeclList>
<VarDeclList> ::= <VarDecl>
               | <VarDeclList>  <VarDecl>
<VarDecl> ::= <IdList>':'<Type><OptAbsoluteClause><OptPortDirectives>';'
          | <IdList>':'<Type>'=' <TypedConstant> <OptPortDirectives>';'
          | <IdList> ':' <TypeSpec>
          | SynError ';'
<ThreadVarDeclList> ::= <ThreadVarDecl>
                      | <ThreadVarDeclList>  <ThreadVarDecl>
<ThreadVarDecl>     ::= <IdList> ':' <TypeSpec>
                      | SynError ';'
<OptAbsoluteClause> ::= ABSOLUTE <RefId>

<ConstSection>      ::= CONST          <ConstantDeclList>
                    | RESOURCESTRING <ConstantDeclList
<ConstantDeclList>::= <ConstantDecl>
                    | <ConstantDeclList>  <ConstantDecl>
<ConstantDecl>:= <RefId> '=' <ConstExpr><OptPortDirectives> ';'
               |<RefId>':'<Type>'='<TypedConstant> <OptPortDirectives> ';'
               | SynError ';'
<TypedConstant>       ::= <ConstExpr>
                        | <ArrayConstant>
                        | <RecordConstant>
<ArrayConstant>        ::= '(' <TypedConstList> ')'
<RecordConstant>        ::= '(' <RecordFieldConstList>     ')'
                        | '(' <RecordFieldConstList> ';' ')'
                        | '(' ')'!only to initialize globar vars"
<RecordFieldConstList>::= <RecordFieldConstant>
                        | <RecordFieldConstList> ';' <RecordFieldConstant>
<RecordFieldConstant>::= <RefId> ':' <TypedConstant>
<TypedConstList>        ::= <TypedConstant>
                        | <TypedConstList> ',' <TypedConstant>
{…}
```

Fonte: Brink (2006).

Quadro 7 - Trecho de uma BNF que define o Delphi

No Quadro 7 é apresentado o trecho de uma BNF que define a linguagem Delphi, no trecho tem-se um exemplo da definição de declaração de variável e constante.

Para apresentar o funcionamento da ferramenta foi aplicada a conversão de um formulário desenvolvido em Delphi (o formulário pode ser visto no Apêndice A). Após a conversão do formulário Delphi a ferramenta Delphi2Java-II gera dois arquivos Java, um contendo a interface gráfica (Apêndice B) e o segundo contendo as assinaturas dos eventos dos formulários.

O protótipo de conversão altera o arquivo que contém as assinaturas dos eventos (Arquivo gerado pela ferramenta Delphi2Java-II), e adiciona as rotinas dos eventos que foram convertidos (Apêndice C). Outro arquivo gerado pelo protótipo de conversão é a classe `Util`, esta classe contém as variáveis, procedimentos e funções públicas do formulário Delphi (Apêndice D). O último arquivo gerado pelo protótipo de conversão é um relatório, que apresenta todas as estruturas que foram reconhecidas no formulário Delphi (Apêndice E).

2.2 ARQUIVO .DFM

Dentro de uma aplicação desenvolvida em Delphi, cada formulário possui associado a ele um arquivo *Delphi Form File* (.DFM). O arquivo .DFM, segundo Cantú (2003, p. 29), é "[...] um arquivo binário com a descrição das propriedades de um formulário [...] e dos componentes que ele contém.". A Figura 1 mostra uma interface gráfica em Delphi que contém um botão.

Figura 1 - Formulário em Delphi

Segundo Fonseca (2005, p. 15), cada linha do arquivo .DFM armazena as informações dos componentes e cada bloco de definição de um componente se inicia com a linha contendo a palavra `object` seguida do nome do objeto e da sua classe. Todas as linhas subseqüentes descrevem as características do objeto, descartando a palavra `end` que indica o final do bloco de informações do objeto.

O Quadro 8 a seguir mostra a forma como é estruturado o arquivo .DFM e é possível identificar entre as linhas 7 e 14 as seguintes informações:

a) linha7: a palavra reservada `object` identifica a declaração de um objeto, neste caso o objeto em questão pertence à classe `TButton` e é identificada pelo nome `Button1`;
b) linha 8: a palavra reservada `Left` indica, em pontos, o deslocamento horizontal do objeto em relação ao objeto pai;
c) linha 9: a palavra reservada `Top` indica, em pontos, o deslocamento vertical do objeto em relação ao objeto pai;
d) linha 10: a palavra reservada `Width` indica, em pontos, a largura do objeto;
e) linha 11: a palavra reservada `Height` indica, em pontos, a altura do objeto;
f) linha 12: a palavra reservada `Caption` identifica o título do objeto;
g) linha 13: a palavra reservada `OnClick` indica que evento de mesmo nome está sendo utilizado, e que a ele está atrelado o procedimento ou *handler* definido pelo nome `Button1Click`, que terá sua implementação descrita no arquivo .PAS;
h) linha 14: a palavra reservada `end` indica a finalização do último objeto declarado e

ainda não finalizado, no caso o objeto `Button1`.

```
01 object Form1: TForm1
02   Left = 389
03   Top = 244
04   Width = 191
05   Height = 102
06   Caption = 'Modelo em Delphi'
07   object Button1: TButton
08     Left = 56
09     Top = 24
10     Width = 75
11     Height = 25
12     Caption = 'Button 1'
13     OnClick = Button1Click
14   end
15 end
```

Quadro 8 - Estrutura do arquivo .DFM

2.3 DEFINIÇÃO DA CLASSE QUE DEFINE O FORMULÁRIO

No Quadro 9 é apresentada a declaração de uma de classe em Delphi que define uma classe do tipo formulário. Esta classe é derivada de `TForm` que é a classe do Delphi que define formulários (ANNES, 2005).

```
type
  TForm1 = class(TForm)
    Button1: TButton;
    procedure Button1Click(Sender: TObject);
  private
    { Private declarations }
  public
    { Public declarations }
  end;
var
  Form1: TForm1;
```

Quadro 9 - Definição de classe formulário em Delphi

Após a definição da classe é declarada uma variável com visibilidade global para que outras *units* tenham acesso. A cada novo objeto adicionado ao formulário, o Delphi atribui a classe que define o formulário um atributo do tipo do objeto adicionado. Da mesma forma ocorre com os eventos que são associados aos objetos do formulário ou ao próprio formulário, o mesmo será definido na classe.

Assim, à classe exemplificada no Quadro 9 estão associados um atributo denominado `Button1` e um procedimento chamado `Button1Click` é associado ao evento `OnClick`. A associação entre o evento `Onclick` e o método `Button1Click` é feita no arquivo DFM.

A construção do código fonte do método definido na classe é feita no trecho de código definido como *implementation*, nesta parte do código Delphi são implementados todos os

procedimentos definidos na *interface*, tanto os procedimentos ligados a classe que define o formulário como os demais métodos que existam no fonte Delphi. Na *implementation* do código Delphi os métodos dos tratadores de eventos são identificados através do nome da classe seguida do nome do método, exemplo: "`Procedure TForm1.Button1Click(Sender: TObject);`", onde `TForm1` é o nome da classe, `Button1Click` é o nome do método e `Sender` é um objeto `TObject` que representa o componente que deu origem ao evento.

2.4 TRATADORES DE EVENTOS EM DELPHI

Segundo Carvalho (1998, p. 18), "os eventos correspondem a ações do usuário que a aplicação pode reconhecer e tratar". Ainda segundo Carvalho (1998, p. 18), um exemplo clássico para eventos é o clique que o usuário pode dar sobre um componente do tipo botão, ou até os eventos que são gerados a partir de ocorrências internas do sistema operacional como, por exemplo, o vencimento de um componente do tipo *timer*.

De acordo com Carvalho (1998, p. 10), no Delphi cada tipo de componente possui uma lista de eventos. Por exemplo, os componentes do tipo `TButton` possuem os seguintes eventos: `OnClick`, `OnContextPopup`, dentre outros.

Segundo Carvalho (1998, p. 18), para cada evento de um componente é possível atribuir procedimentos ou *handlers* com um código em Delphi descrito no arquivo .PAS e definir o comportamento do programa quando o evento ocorrer.

No Quadro 10 é possível identificar o código fonte do *handler* associado ao evento `OnClick` do objeto `Button1`.

```
var
  Form1: TForm1;
implementation
{$R *.dfm}
procedure TForm1.Button1Click(Sender:TObject);
begin
  ShowMessage('Mensagem ao Usuário.');
end;
end.
```

Quadro 10 - Código fonte do arquivo .PAS

No bloco delimitado por `begin` e `end;` é onde estarão descritas todas as ações que serão tomadas quando o evento for ativado. No caso do evento apresentado no Quadro 10, quando for executado ele irá exibir uma mensagem ao usuário.

2.5 TRATADORES DE EVENTOS EM JAVA

Conforme Gonçalves (2007, p. 103), para o processamento dos eventos gerados na interação com o usuário, o Java utiliza de um pacote chamado `java.awt.event`. Para os outros tipos de eventos, que são específicos da biblioteca Swing, é utilizado o pacote `java.swing.event`.

As interações dos usuários com os componentes de interface geram eventos. Segundo Schvepe (2006, p. 17), o mecanismo de tratamento de eventos em Java é formado por três partes:

a) o objeto origem, isto é, o componente que gerou o evento;
b) o evento propriamente dito, ou seja, um objeto que encapsula as informações necessárias para processar o evento gerado;
c) o objeto ouvinte (*listener*), ou seja, o objeto que é notificado pelo objeto origem quando ocorre um evento e que usa a informação da notificação para responder ao evento.

Os *listeners* são classes criadas especificamente para o tratamento de eventos.

O processamento do evento é delegado ao *listener* no aplicativo. Quando ocorre um evento existe um mecanismo de aviso (*dispatch*) para os *listener* daquele evento. O *dispatch* nada mais é do que uma delegação para que o *listener* cuide do tratamento de evento (RUIZ, 2006).

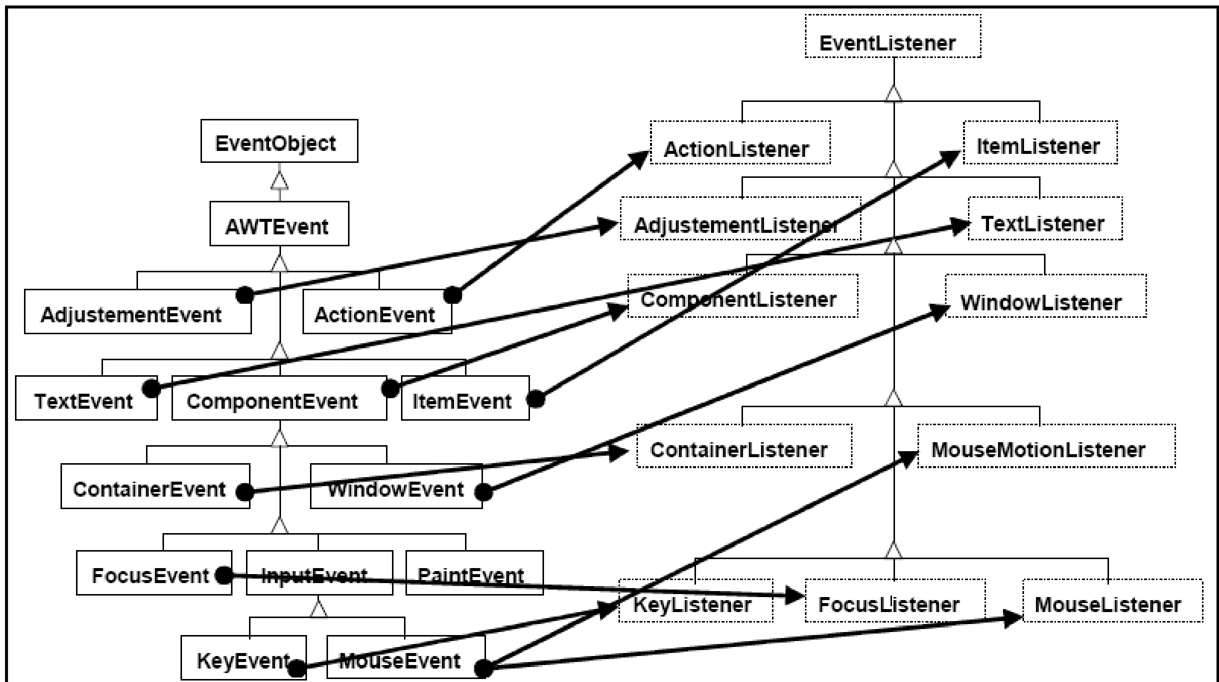
No Quadro 11 são descritos os principais eventos que podem ocorrer em uma aplicação, e em que momento elas ocorrem.

Evento	**Descrição**
ActionEvent	Usuário clica em um botão, pressiona return dentro de um textfield ou escolhe um item de menu
WindowEvent	Usuário fecha, minimiza, maximiza, etc., uma janela
MouseEvent	Usuário pressiona entra ou sai com o mouse de dentro de um componente
MouseMotionEvent	Usuário move o mouse sobre um componente

Fonte: Fernandes (2002).

Quadro 11 - Principais eventos

Os vários tipos de *listeners* são especificados através de um uma interface. A Figura 2 exibida abaixo mostra uma hierarquia dos principais *listeners* do pacote `java.awt`.



Fonte: Fernandes (2002).

Figura 2 - Hierarquia dos principais *listeners*

O Quadro 12 detalha um trecho de código que descreve o uso do tratador de eventos quando um botão é pressionado.

```
// 1 - Criando um componente visual
Button b1 = new Button("Salvar");
// 2 - Criando um observador de Eventos através de uma classe anônima[1]
 ActionListener observadorDeB1 = new java.awt.event.ActionListener() {
 // 2.1 - implementando o método para receber uma notificação
 public void actionPerformed(ActionEvent e) {
  // 2.1.1 - invoca o método da fachada[2]
  System.out.println("O botão salvar foi pressionado!");
 }
}
// 3 - Registrando um observador (listener) junto a um componente
B1.addActionListener(observadorDeB1);
```

Fonte: Fernandes (2002).

Quadro 12 - Trecho de código do uso de eventos

2.6 DELPHI2JAVA-II

A ferramenta Delphi2Java-II que foi desenvolvida como Trabalho de Conclusão de Curso (FONSECA, 2005) no sentido de desenvolver-se uma ferramenta que, a partir de

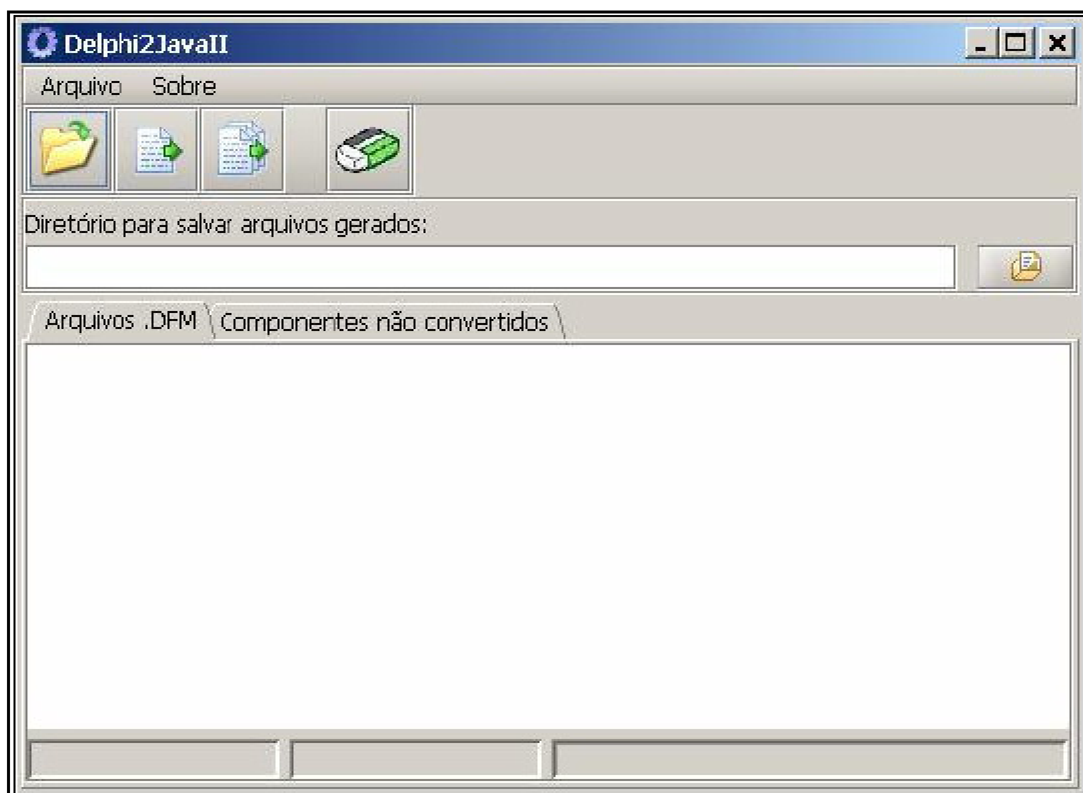
[1] Classe anônima é uma classe local sem nome definida e instânciada em uma uma expressão (FLANAGAN, 2006, p. 153).

[2] O padrão Fachada ou *Façade* oferece uma interface única de nível mais elevado para um conjunto de interfaces de um subsistema (ROCHA, 2003).

formulários gerados no ambiente Delphi pudesse gerar código equivalente em Java tendo em vista facilitar a utilização de aplicações orientadas a formulários nas disciplinas introdutórias de programação OO. Ainda em 2005 foi desenvolvido um subprojeto (SOUZA, 2005) que possui as mesmas funcionalidades do Delphi2Java-II, porém destinado a converter formulários Delphi em páginas *HyperText Markup Language* (HTML).

O trabalho de conclusão de curso (Sistemas de Informação) de Silveira (2006) foi concluído incorporando ao escopo do projeto a funcionalidade de acesso a banco de dados e convertendo a ferramenta para linguagem Java.

Conforme Mattos et al (2006), o projeto Delphi2Java-II vem sendo desenvolvido em paralelo aos trabalhos de conclusão de curso como um projeto de pesquisas PIBIC/CNPq desde 2005. A última versão liberada foi utilizada como ponto de partida para o desenvolvimento do presente projeto, a interface gráfica se mantém a mesma do projeto liberado por Silveira (2006), apenas foram adicionados novos tratamentos ao projeto com o objetivo de aumentar a robustez do projeto.



Fonte: Silveira (2006, p. 67).

Figura 3 - Tela do Delphi2Java-II

Conforme Silveira (2006), a análise para conversão dos formulários Delphi para Java é feita com base nas informações descritas no arquivo DFM. Este arquivo contém todas as informações necessárias para construção do código com as funcionalidades para a camada de interface de uma aplicação. Os arquivos DFM são processados utilizando analisadores

léxicos, sintáticos e semânticos. No Quadro 13 é apresentada a estrutura dos *tokens* que identificam as informações no arquivo DFM.

```
//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]
```

Fonte: Souza (2005, p. 38).

Quadro 13 - Especificação dos *tokens*

A estrutura sintática de um arquivo .DFM, especificada utilizando a notação BNF utilizada na ferramenta, encontra-se no Quadro 14.

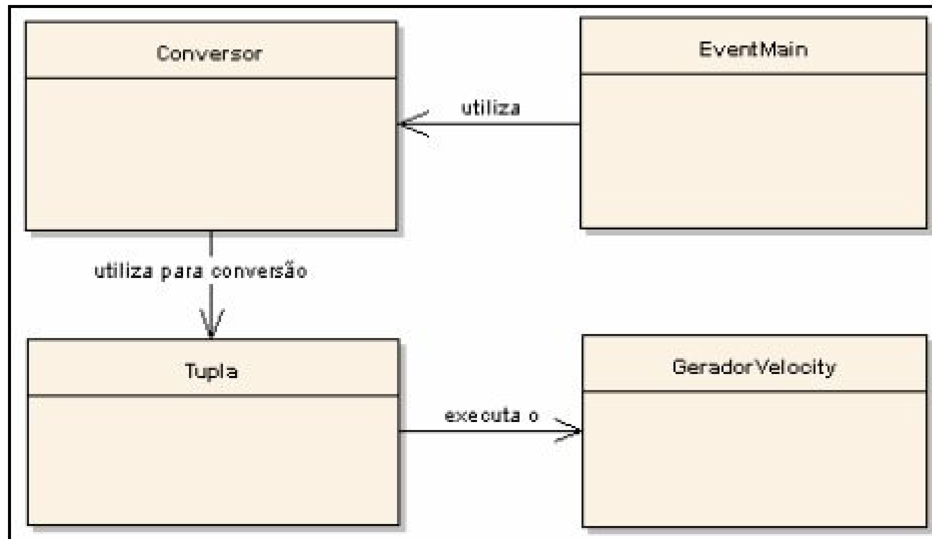
```
<dfm>::= #³1 <object> #10
<object>::= OBJECT identifier #2 ":" <type> #3 <propertyList> <objectList> END
<type>::= identifier | TBitBtn | TButton | TCheckBox | TcomboBox | Tedit
| TGroupBox | TLabel | TListBox | TMainMenu | TMemo | TmenuItem | TPanel
| TPageControl | TPopupMenu | TProgressBar | TSpeedButton | 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 ;
```

Fonte: Souza (2005, p. 39).

Quadro 14 - Gramática

Abaixo na Figura 4 são apresentadas as classes do Delphi2Java-II responsáveis pela conversão do formulário Delphi para Java.

[3] O caracter # seguido de número define ações semânticas dentro das produções sintáticas (GESSER, 2003).



Fonte: Silveira (2006, p. 49).

Figura 4 - Classes do Delphi2Java-II para geração de código

Dentro da estrutura do projeto, a classe `Conversor` efetua a análise do arquivo DFM utilizando os analisadores léxicos, sintáticos e semânticos (trecho 01). Caso não encontre nenhum erro é gerada uma instância da classe `Tupla`, contendo as informações extraídas do formulário. Após o sucesso na análise do arquivo DFM é invocado o método `converteSwing` (trecho 02), que irá iniciar o processo de conversão da interface gráfica. O Quadro 15 mostra o trecho de código citado acima.

```
public String geraArquivoConversao(File file,boolean formPrincipal)
                                                    throws Exception{
  ...
  try {
    verifica(file,reader);    01
    Tupla element = (Tupla) tuplas.get(0);
    ...
    element.convertSwing(nomeUnitForm, nomeArquivo, formPrincipal, this.screen); 02
  } catch (AnalysisError e) {
    throw new AnalysisError("O arquivo " + file.getName()+ " está com erro!");
  }
  ...
}
```

Fonte: adaptado de Silveira (2006, p. 54).

Quadro 15 - Metodo `geraArquivoConversao` da classe `Conversor`

Inicialmente o método `converteSwing` extrai da classe `Tupla` um componente do tipo `TForm`, sendo criada uma instância da classe `TJForm` que será o objeto pai da estrutura. A partir deste objeto pai é iniciada a identificação das propriedades, eventos e objetos filhos mapeados a partir do arquivo DFM.

Todos os eventos identificados pelo analisador são armazenados em listas específicas para cada evento. Dentro do método `converteSwing` é criada uma instância da classe `GeradorVelocity` para realizar a geração das classes Java utilizando *templates*, interpretados pelo motor de *template Velocity*.

A partir dos *templates* definidas e do conjunto de informações extraídas do arquivo DFM, é dado inicio ao processo de geração dos arquivos Java. Para cada arquivo DFM são gerados dois arquivos Java, um contendo a interface gráfica, com os componentes convertidos, e um segundo arquivo contendo as assinaturas dos eventos identificados no arquivo DFM.

Na Figura 5 é apresentado um formulário Delphi para conversão pela ferramenta Delphi2Java-II.

Figura 5 - Formulário Delphi para conversão

Como descrito anteriormente, a ferramenta gera dois arquivos Java, um com a interface gráfica (Figura 6), e um segundo contendo as assinaturas dos eventos (Quadro 16).

Figura 6 - Formulário convertido pela ferramenta Delphi2Java-II

No trecho 01 do Quadro 16, é apresentada a assinatura de um evento identificado pela ferramenta Delphi2Java-II, no caso o click no botão do formulário.

```
public class uForm1Event{
 private uForm1FRM screen;
 public uForm1Event(uForm1FRM screen) {
   this.screen = screen;
 }
 public void inicializar() throws  Exception {
   CriarEventos();
 }
 private void CriarEventos() {
   screen.getbtnAtribuir().addActionListener(new java.awt.event.ActionListener(){
       public void actionPerformed(java.awt.event.ActionEvent evt) {
          btnAtribuirActionPerformed(evt);
   }});
 }
 private void btnAtribuirActionPerformed(java.awt.event.ActionEvent evt) {
 }                                                                     01
 public static void main(String args[]) throws  Exception{
      new uForm1FRM().setVisible(true);
 }
}
```

Quadro 16 - Assinaturas dos eventos gerados pela ferramenta Delphi2Java-II

2.7 TRABALHOS CORRELATOS

Seibt (2001) descreve uma ferramenta capaz de analisar código fonte de um projeto orientado a objetos em *Object Pascal* – Delphi, extraindo as classes, seus métodos e atributos para posterior cálculo de métricas de projeto. Para identificar e extrair as informações das classes do projeto são utilizados diagramas de sintaxe, estes diagramas representam a definição formal da linguagem *Object* Pascal.

Delphi2Java (RE-CODING, 2005) é uma ferramenta comercial que se propõe a converter interfaces gráficas, componentes de acesso a banco de dados e código fonte. O código fonte gerado pela ferramenta necessita ser ajustado manualmente para que possa passar pelo compilador Java.

A ferramenta Converte Forms (SCHVEPE, 2006) migram os sistemas legados desenvolvidos em Oracle Forms para linguagem de programação Java. O processo de conversão é baseado em *templates* interpretados pelo motor de *template* eNITL, sendo que são convertidos componentes visuais, *triggers* e um conjunto do linguagens PL/SQL.

Delphi2CS (NETCOOLE, 2007) é uma ferramenta para auxiliar na conversão de projetos Delphi 5, 6, 7 e Delphi.NET para linguagem C#. Delphi2CS cria um novo projeto, converte cada arquivo a partir do projeto original para o novo projeto (.csproj) e gera um relatório detalhando o que foi feito. Após a conversão do projeto a ferramenta Delphi2CS faz a inserção de comentários nos códigos C# gerados alertando trechos que devem ser convertidos de forma manual.

3 DESENVOLVIMENTO

Na fase de desenvolvimento do protótipo foi utilizada a linguagem Java e a IDE NetBeans 6.1 (NETBEANS, 2008) para a geração das rotinas que fazem a análise e a geração de código fonte dos tratadores de eventos. Para alterações na ferramenta Delphi2Java-II de forma a garantir a integração entre as duas ferramentas foi utilizada a IDE Eclipse 3.3.1.1.

3.1 REQUISITOS PRINCIPAIS DO PROBLEMA A SER TRABALHADO

O trabalho desenvolvido deve atender aos seguintes requisitos:

a) efetuar a análise dos arquivos fonte Delphi identificando os métodos de tratamento de eventos para o subconjunto de eventos;
b) definir uma Metalinguagem para facilitar a geração de código Java;
c) produzir código fonte Java para o conjunto de métodos identificados no item b;
d) integrar com a ferramenta Delphi2Java-II.

A ferramenta deverá conter os seguintes requisitos não funcionais:

a) ser implementado utilizando o ambiente NetBeans versão 6.1;
b) ser implementado na linguagem Java;
c) ser compatível com o sistema operacional Windows 2000 e XP;
d) utilizar técnicas de compilação e geração de código como ferramenta de apoio.

3.2 ESPECIFICAÇÃO

Nesta seção é feita a apresentação detalhada da arquitetura do protótipo desenvolvido.

A tarefa de conversão dos tratadores de eventos do fonte Delphi para Java se divide em três etapas: a primeira é a seleção dos arquivos que serão convertidos, a segunda etapa é o processamento de cada um dos fontes visando encontrar os comandos Delphi e carregar essas informações em uma metalinguagem, a terceira parte consiste em percorrer a metalinguagem montada e gerar o código fonte Java. Na Figura 7 é apresentado o diagrama de caso de uso da

ferramenta e o Quadro 17 apresenta o detalhamento de cada um dos casos de uso.

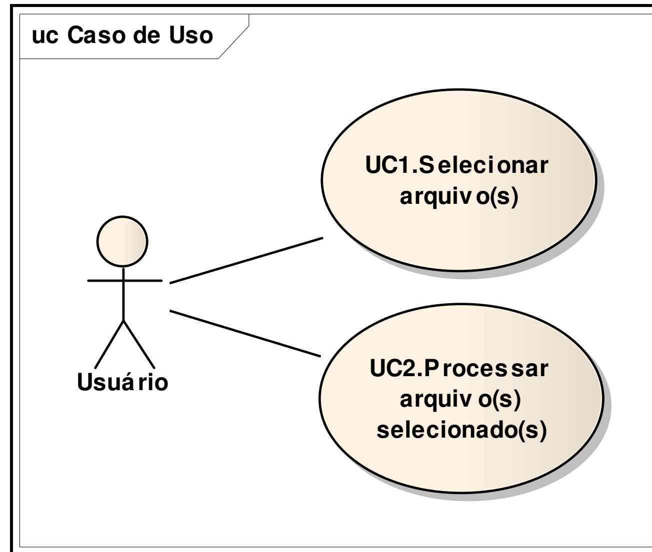


Figura 7 - Diagrama de caso de uso

UC1. Selecionar Arquivo **Pré-condição:** O arquivo .PAS não pode estar corrompido. **Cenário principal:** 1. A ferramenta Delphi2Java-II gera os arquivos com a interface gráfica e as assinaturas dos eventos. 2. A ferramenta Delphi2Java-II instancia o protótipo atribuindo o arquivo .PAS para conversão. 3. O protótipo armazena o caminho para o arquivo selecionado. **Pós-condição:** Um caminho para o arquivo que será gerado.
UC2. Processa Arquivo Selecionado **Pré-condição:** Deve ser atribuído um arquivo .PAS **Cenário principal:** 1. A ferramenta Delphi2Java-II chama o método para processar o arquivo. 2. o Protótipo faz a análise e geração da metalinguagem do arquivo. 3. O protótipo faz a criação dos códigos fonte Java necessários e escreve no arquivo de assinaturas os eventos. **Pós-condição:** Uma metalinguagem gerada para cada arquivo selecionado e um ou mais arquivos com extensão .Java serão gerados para o arquivo .PAS.

Quadro 17 - Detalhamento dos casos de uso

A Figura 8 apresenta o diagrama de atividades do protótipo, demonstrando a interação com a ferramenta Delphi2Java-II e o processo de conversão dos tratadores de eventos.

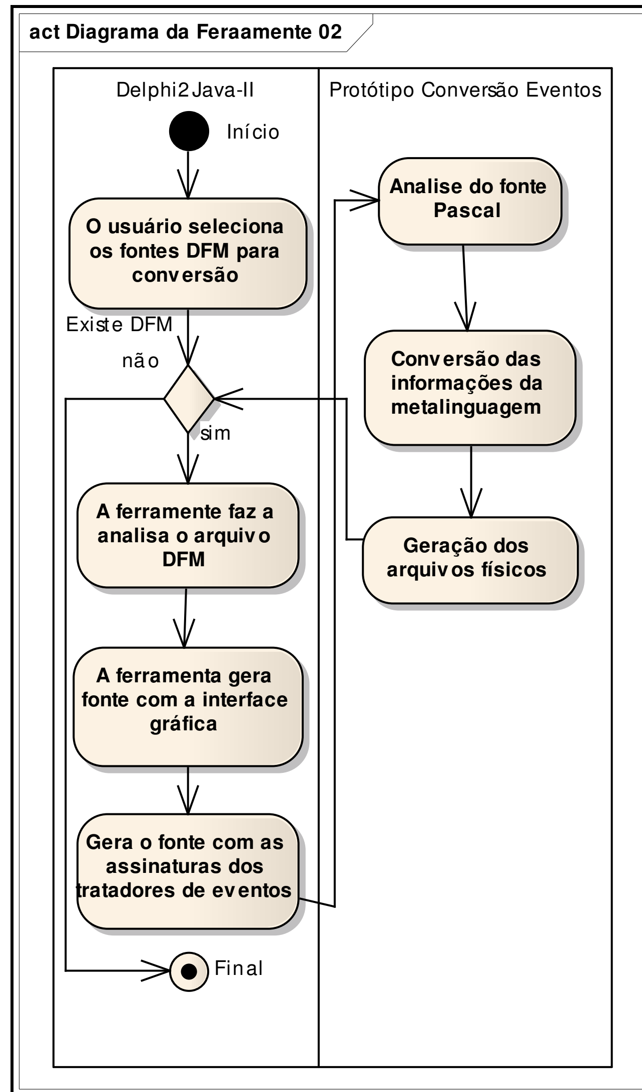


Figura 8 - Diagrama de Atividades da ferramenta

O diagrama apresentado na Figura 8 mostra as etapas de processamento dos arquivos Delphi para a geração dos fontes Java, desde a análise dos fontes até a geração do fonte final. O processo de conversão se inicia na seleção dos fontes DFM pela ferramenta Delphi2Java-II que terão inicialmente a sua interface gráfica convertida. Para cada arquivo DFM selecionado o protótipo busca um arquivo .PAS com o mesmo nome do DFM e é feita a varredura do fonte com o objetivo de extrair as informações necessárias para a geração da metalinguagem. Após a análise do fonte .PAS são gerados os códigos fonte Java a partir da metalinguagem do fonte Delphi, o código gerado é armazenado na estrutura e será usado para geração do arquivo físico e para reescrever o arquivo com as assinaturas dos eventos, que é gerado inicialmente pela ferramenta Delphi2Java-II.

3.2.1 Metalinguagem

Uma metalinguagem[4] é uma estrutura de dados que armazena informações sobre uma linguagem. Esta foi a estratégia utilizada para mapear as convenções sintáticas entre Delphi e Java. As principais classes que compõem a metalinguagem estão definidas no pacote `classes` são apresentadas na Figura 9.

[4] Linguagem (natural ou formalizada) que serve para descrever ou falar sobre uma outra linguagem, natural ou artificial [As línguas naturais podem ser usadas como sua própria metalinguagem.] (HOUAISS, 2007)

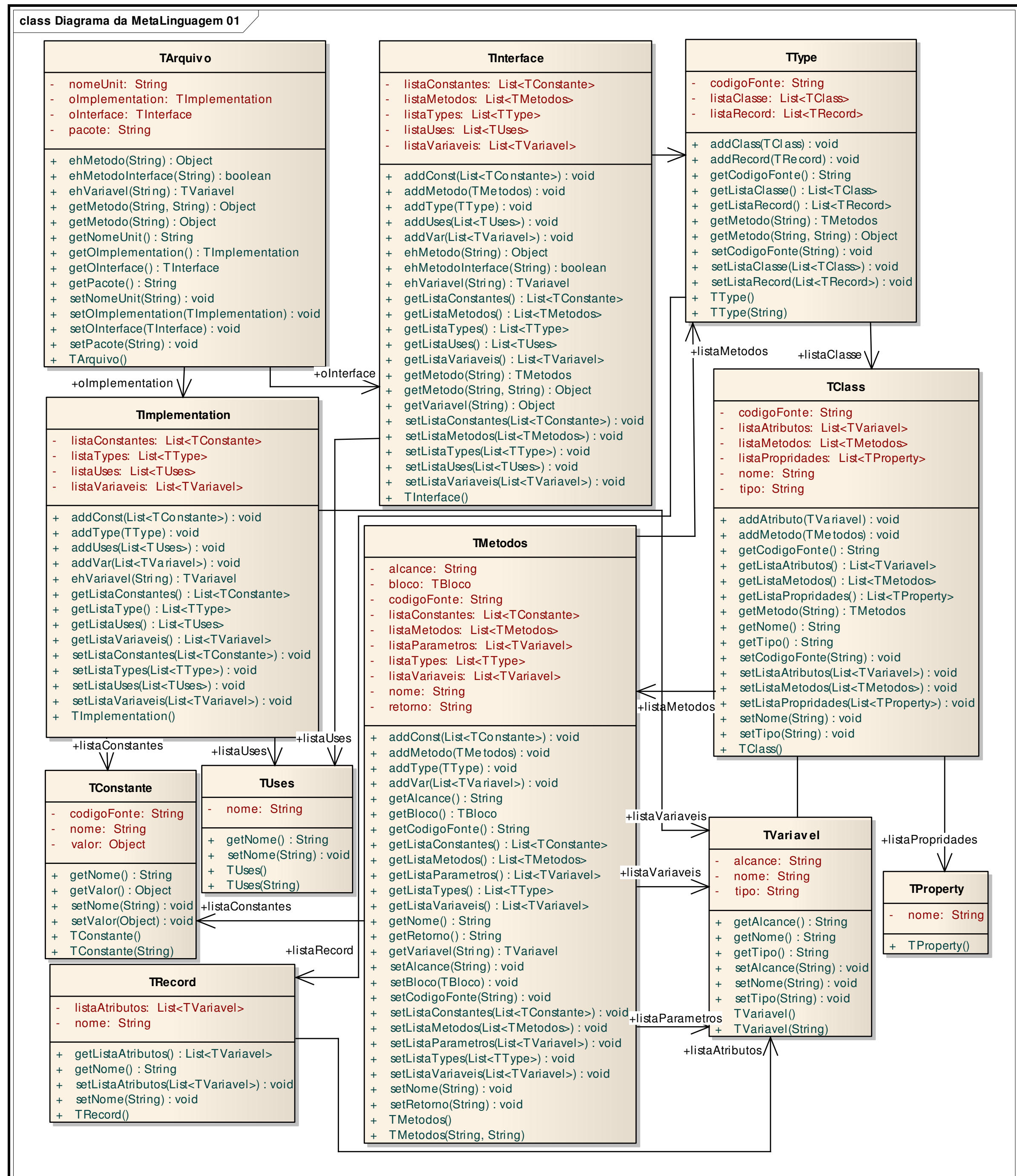


Figura 9 - Diagrama de classes principal do pacote `classes`

A organização desta estrutura se iniciar na classe `TArquivo`, este classe contém duas subclasses, a `TInterface` e a `TImplementation`. A classe `TInterface` contém as informações da *interface* do fonte Delphi e agrupa uma estrutura contendo listas de classes `TConstante`, `TMetodos`, `TType`, `TUses` e `TVariavel`. Já a classe `TImplementation` contém as informações encontradas na *Implementation* do código Delphi e agrupa uma estrutura contendo, listas de classes `TConstante`, `TUses` e `TVariavel`.

A classe `TConstante` contém informações sobre as constantes declaradas no fonte, como por exemplo, o nome da constante e o seu valor.

A classe TUses contém o nome dos *use* encontrados no fonte Delphi.

A classe TVariavel agrupa informações sobre as variáveis do fonte, como por exemplo seu nome, tipo e alcance.

A classe TType contém informações sobre os *types* encontrados no fonte, e esta classe é formada por uma ou várias classes TVariavel ou TRecord.

A classe TMetodos, contém informações sobre os procedimentos e funções encontrada no código fonte, e é composta por listas de classes do tipo TConstantes, TVariaveis, TTypes e TMetodos declarados dentro do procedimento ou função. Outra classe que compõem a estrutura da classe TMetodos é a classe TBloco (a classe TBloco será descrita na Figura 10).

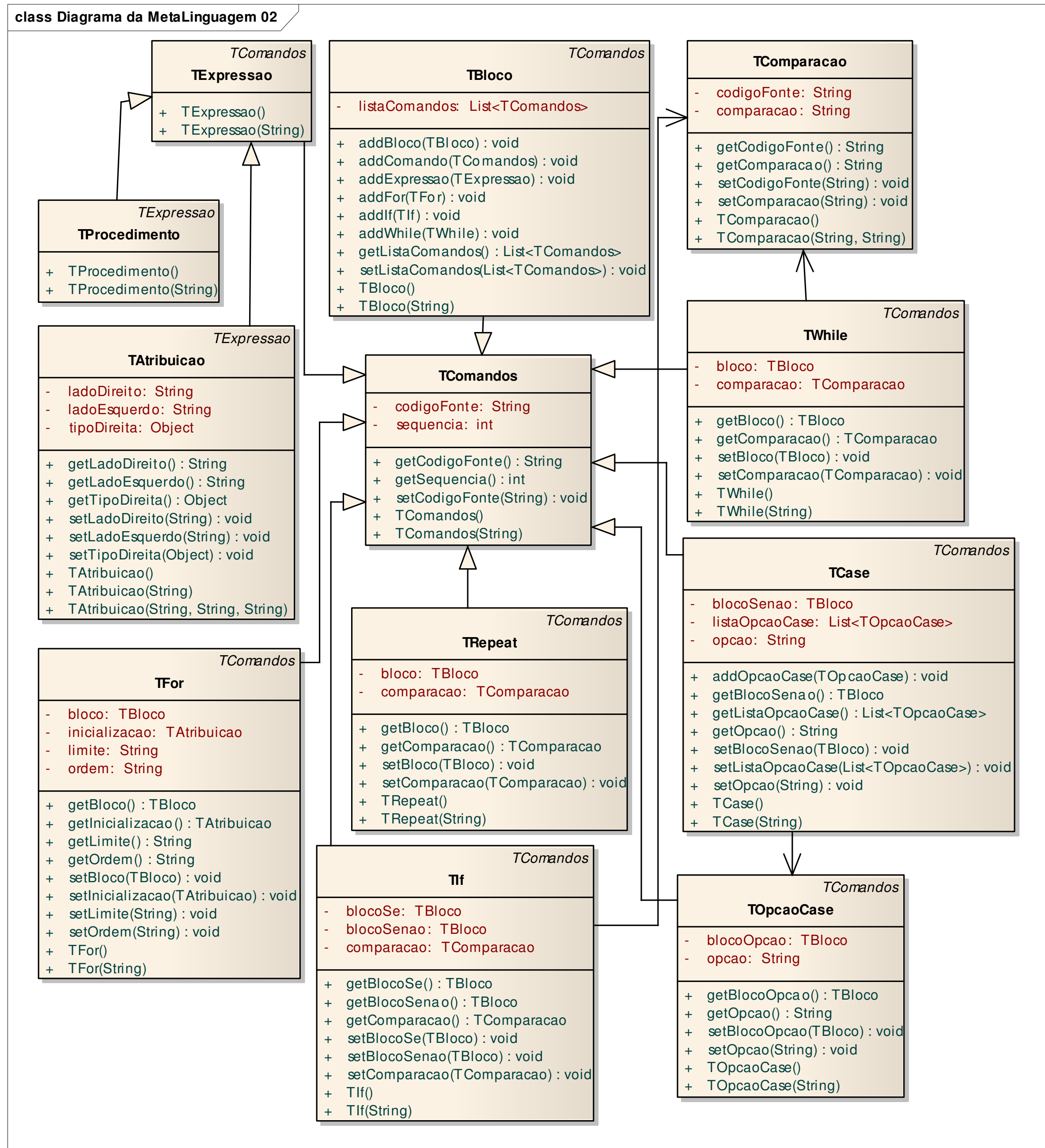


Figura 10 - Diagrama de classes secundário do pacote `classes`

No diagrama classes apresentado na Figura 10 são descritos as demais classes do pacote `classes`, iniciando a pela classe `TBloco`, esta classe representa na metalinguagem a lista de comandos da linguagem Delphi que foi interpretada dentro do bloco `Begin end`.

Uma classe `TBloco` é composta por uma lista de objetos da classe `TComando`. A classe `TComando` representa os comandos encontrados dentro do código fonte com, por exemplo, `If`, `for` e `while`. Para cada comando da linguagem Delphi foi criada uma subclasse da classe `TComando`, dentre as subclasses criadas pode-se descrever as classes `TIf`. A classe `TIf` que representa o comando `If` do Delphi é composta por dois objetos do tipo `TBloco`, um para representar os comandos do bloco "se" e outro para representar os comandos do bloco

"senão", e ainda completa a estrutura da classe um objeto do tipo `TComparacao`.

Outra importante subclasse é a classe `TCase` que representa o comando `case` do Delphi. A estrutura da classe `TCase` é composta por um objeto do tipo `TBloco` para representar o senão do comando `case`, um atributo que recebe o nome da variável que será comparada e uma lista de objetos do tipo `TOpcaoCase`. A classe `TOpcaoCase` representa uma opção do comando `case` e esta classe é composta por um objeto do tipo `TBloco` onde serão armazenados os comandos da opção e também por um atributo que recebe o valor válido para a opção.

A BNF que define a estrutura de formação da metalinguagem pode ser vista Quadro 18.

```
L  : [A-Za-z]
D  : [0-9]
WS : [\ \t\n\r]

nome : {L} ( {L} | {D} | _ )*
pacote : {L} ( {L} | {D} | _ )*
valor : ({L} ( {L} | {D} | _ )*| "'")
tipo : {L} ( {L} | {D} | _ )*
alcance : {L} ( {L} | {D} | _ )*
opcao : {L} ( {L} | {D} | _ )*
inicializacao: {L} ( {L} | {D} | _ )*
limite : {L} ( {L} | {D} | _ )*
ordem : {L} ( {L} | {D} | _ )*
procedimento : {L} ( {L} | {D} | _ )*
ladoDireito : {L} ( {L} | {D} | _ )*
ladoEsquerdo : {L} ( {L} | {D} | _ )*

//Gramática.
<TArquivo> ::= nome <lsTUses> <TInterface> <TImplementation> pacote;
//lista de Uses pode ser composta por, um Use, uma lista de Uses ou estar vazia.
<lsTUses> ::= <TUses> | <TUses> <lsTUses> | î;
<TUses> ::= nome;
//uma interface é composta por uma lista de uses, uma lista de constantes, uma
lista de métodos e uma lista de Types.
<TInterface> ::= <lsTUses> <lsTConstantes> <lsTVariaveis> <lsTMetodos> <lsTTypes>;
//lista de constantes pode ser composta por, uma constante, uma lista de constantes
ou esta vazia.
<lsTConstantes> ::= <TConstantes> | <TConstantes> <lsTConstantes> | î;
<TConstantes> ::= nome valor;
//lista de variáveis pode ser composta por, uma variável, uma lista de variáveis ou
estar vazia.
<lsTVariaveis> ::= <TVariaveis> | <TVariaveis> <lsTVariaveis> | î;
<TVariaveis> ::= tipo nome;
//lista de métodos pode ser composta por, um método, uma lista de métodos ou estar
vazia.
<lsTMetodos> ::= <TMetodos> | <TMetodos> <lsTMetodos> | î;
//um método pode ser composto por um {alcance}, um retorno, um {nome}, uma lista de
variáveis parâmetros, uma lista de parâmetros uma lista de constantes, uma lista de
métodos e um bloco.
<TMetodos> ::= <decMetodo><lsTVariaveis><lsTConstantes><lsTMetodos> <TBloco>;
//declaração de método.
<decMetodo> ::= alcance <retorno> nome <lsTVariaveis>;
<retorno> ::= tipo | î;
<lsTTypes> ::= <TTypes> | <lsTTypes> | î;
<TTypes> ::= <lsTClass> | <lsTRecord>;
<lsTClass> ::= <TClass> | <lsTClass>;
<TClass> ::= nome tipo <lsTVariaveis> <lsTMetodos> <lsTProperty>;
<lsTProperty> ::= <TProperty> | <TProperty> <lsTProperty> | î;
//atualmente o property só contém um {nome}.
<TProperty> ::= nome;
<lsTRecord> ::= <TRecord> | <TRecord> <lsTRecord> | î;
<TRecord> ::= nome tipo <lsTVariaveis>;
<TBloco> ::= <lsTComandos>;
<lsTComandos> ::= <TComandos> | <TComandos> <lsTComandos> |î;
<TComandos> ::= <TIf>|<TCase>|<TWhile>|<TFor>|<TRepeat>|<lsTExpressao>;
<TIf> ::= <TComparacao> <TBloco> <TBlocoSenao>;
<TBlocoSenao> ::= <TBloco> | î;
<TCase> ::= opcao <lsTOpcaoCase> <TBlocoSenao>;
<lsTOpcaoCase> ::= <TOpcaoCase> | <TOpcaoCase> <lsTOpcaoCase>;
<TWhile> ::= <TComparacao> <TBloco>;
<TFor> ::= inicializacao limite ordem <TBloco>;
<TRepeat> ::= <TBloco> <TComparacao>;
<lsTExpressao> ::= <TExpressao> | <TExpressao> <lsTExpressao>;
<TExpressao> ::= <Tprocedimento> | <TAtribucao>;
<Tprocedimento> ::= procedimento;
<TAtribucao>::= ladoDireito ladoEsquerdo tipo;
```

Quadro 18 - BNF que define a metalinguagem

Para exemplificar a metalinguagem gerada para um arquivo será usado o fonte Delphi descrito no Quadro 19. Este fonte possui somente um botão e um método associado ao evento `OnClick` que faz atribuições a uma variável e chama um método do Delphi. As principais informações extraídas deste formulário podem ser vistas no Quadro 20.

```
unit Unit1;
interface
uses
Windows, Messages, Classes, Graphics, Controls, Forms,Dialogs, StdCtrls;
type
  TForm1 = class(TForm)
    btnOlaMundo: TButton;
    procedure btnOlaMundoClick(Sender: TObject);
  private
    { Private declarations }
  public
    { Public declarations }
  end;
var
  Form1: TForm1;
implementation
{$R *.dfm}
procedure TForm1.btnOlaMundoClick(Sender: TObject);
var ind: integer;
begin
  ind:= 0;
  ShowMessage('Olá Mundo!');
  ind:= ind + 1;
end;
end.
```

Quadro 19 - Exemplo de código fonte Delphi

As principais informações extraídas deste formulário podem ser vistas no Quadro 20.

```
TArquivo
 |-NomeUnit = "Unit1"
 |->TInterface
    |->ListaTVariavel
       |->TVariavel
          |-Nome = "Form1"
          |-Tipo = "TForm1"
    |->ListaTUses
       |->TUses
          |-Nome = "Windows"
       ...
       |->TUses
          |-Nome = "StdCtrls"
    |->ListaTTypes
       |->TTYpe
          |->ListaTClass
             |->TClass
                |-Nome = "TForm1"
                |-Tipo = "TForm"
                |->ListaAtributos
                   |->TVariavel
                      |-Nome = "btnOlaMundo"
                      |-Tipo = "TButton"
                |->ListaTMetodos
                   |->TMetodos
                      |-Nome = "btnOlaMundoClick"
                      |->ListaParametros
                         |->TVariavel
                            |-Nome = "Sender"
                            |-Tipo = "TObject"
                      |->ListaTVariaveis
                         |->TVariavel
                            |-Nome = "ind"
                            |-Tipo = "integer"
                      |->Bloco
                         |->ListaTComandos
                            |->TAtribuicao
                            |->TProcedimento
                            |->TAtribuicao
 |->TImplementation
```

Quadro 20 - Metalinguagem gerada da `Unit1`

Como apresentado no Quadro 20, foram localizados o nome da `unit`, um objeto do

tipo `TInterface` que possui, por exemplo, um atributo do tipo `TVariavel`, uma lista de `TUses`, um `TType` entre outras informações.

3.2.2 Diagrama de pacotes

O protótipo foi dividido em quatro pacotes principais e mais um pacote contendo as classes para interação com a ferramenta Delphi2Java-II (como é descrito na Figura 11).

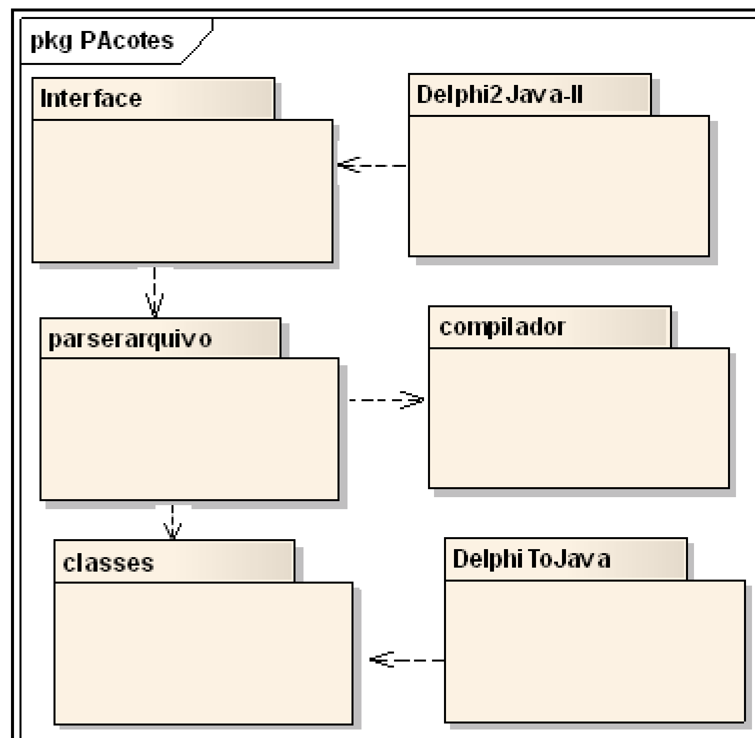


Figura 11 - Diagrama de pacotes

O principal pacote da ferramenta é chamado `parserarquivo`. Este contém as classes responsáveis pela análise do código fonte Delphi, geração da metalinguagem, gerenciamento da montagem do código fonte Java e a alteração no fonte gerado pelo Delphi2Java-II que possui as assinaturas dos eventos identificados no formulário Delphi. O pacote `Delphi2Java-II` representa a interação com a última versão disponível da ferramenta Delphi2Java-II. Na Figura 12 é apresentado o Diagrama de Classes do pacote `parserarquivo`, mostrando as classes que o compõem.

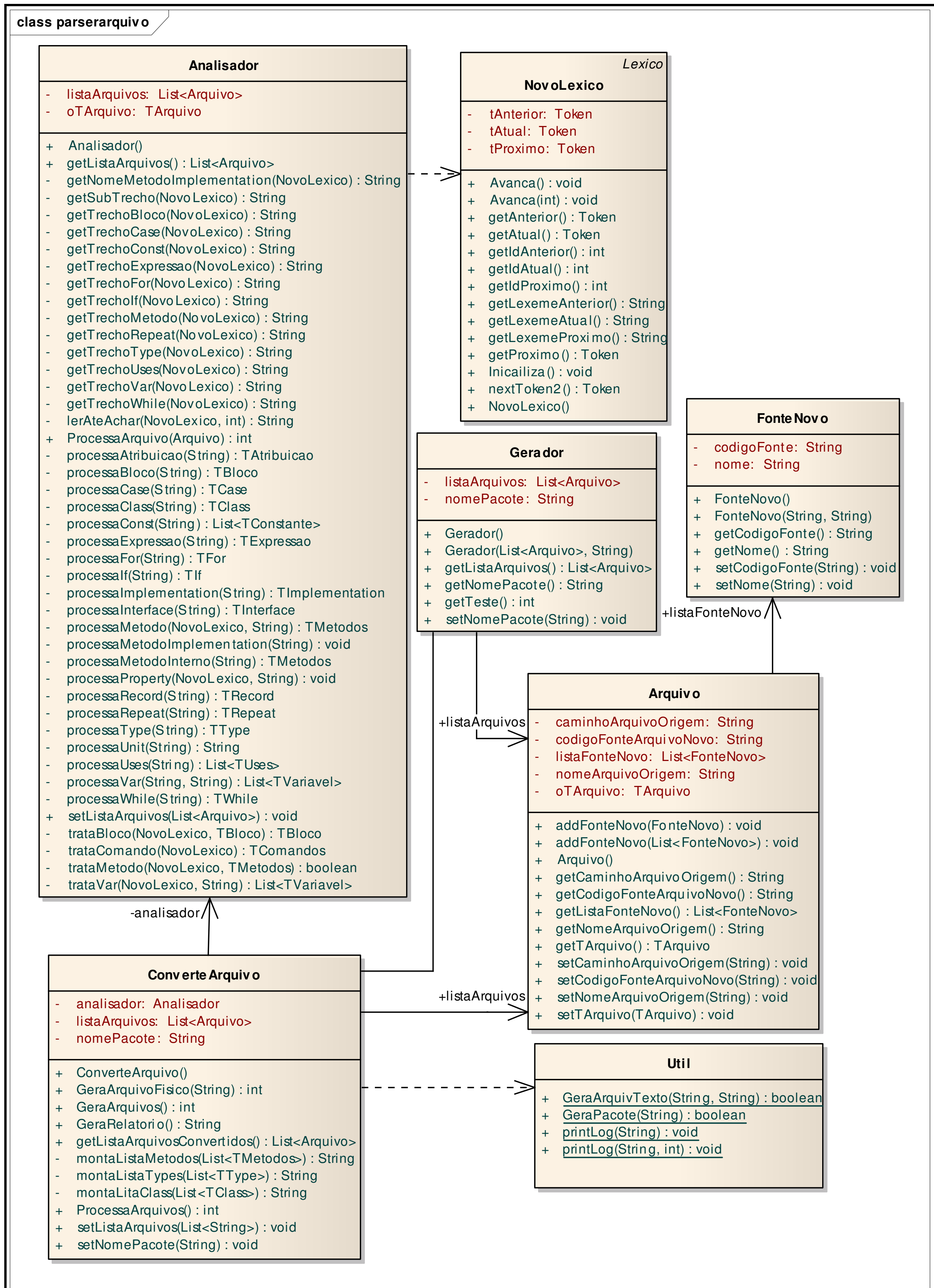


Figura 12 - Diagrama de classes do pacote `parserarquivo`

A classe `ConverteArquivo` tem o objetivo de gerenciar o processamento do arquivo, desde a validação do fonte que será analisados até a geração do código fonte Java. Comanda o processo de análise do fonte Delphi e a geração do fonte Java.

A responsabilidade por efetuar a varredora do fonte Delphi e montar a metalinguagem é da classe `Analisador`. O processo de varredura do fonte se inicia no método `Analisador.ProcessaArquivo`, este método recebe como parâmetro um objeto da classe `Arquivo` que contém inicialmente o caminho para o arquivo fonte e o seu nome.

O processo de análise do código fonte se inicia com a divisão do código fonte e três partes, uma contendo a declaração da `unit`, uma segunda parte que contém o trecho da `interface` e outro contendo a `implementation`. A descrição do processo de leitura e processamentos destes três trechos de código será apresentada por completo a partir da seção 3.3. A classe `NovoLexico` estende a classe `Lexico` (esta classe é descrita no diagrama da Figura 13) e tem o objetivo de receber um trecho de código e efetuar a análise léxica do trecho de código identificando os *tokens*. A classe `NovoLexico` adiciona à classe `Lexico` métodos que permitam identificar qual foi o último *token* encontrado e qual será o próximo (Quadro 21).

MÉTODO	**DESCRIÇÃO**
Avanca()	Busca um novo *token* e atualiza o anterior e o próximo *token*.
Avanca(int avanca)	Avança um número de vezes que é passado como parâmetro.
Inicializa()	Este inicializa os atributos tAnterior com nulo, tAtual buscando o novo *token*, tProximo também buscando o novo *token*.
getAnterior()	Busca o *token* que está no atributo tAnterior
getAtual()	Busca o *token* que está no atributo tAtual
getProximo()	Busca o *token* que está no atributo tProximo
getIdAnterior()	Retorna o identificador do *token* tAnterior
getIdAtual()	Retorna o identificador do *token* tAtual
getIdProximo()	Retorna o identificador do *token* tProximo
getLexemaAnterior()	Retorna a descrição do *token* tAnterior
getLexemaAtual()	Retorna a descrição do *token* tAtual
getLexemaProximo()	Retorna a descrição do *token* tProximo

Quadro 21 - Relação de métodos da classe `NovoLexico`

Após o processo de análise do código fonte Delphi e montagem da metalinguagem é possível iniciar a geração do código fonte Java. A geração é feita pela classe Gerador que com base na metalinguagem gerencia o processo de montagem dos códigos Java.

O processamento da metalinguagem gera instâncias da classe `FonteNovo`, esta classe armazena o nome e o código fonte Java.

Por fim a classe `Util`, contém funcionalidades de uso comum de toda a aplicação, como por exemplo, o gerenciamento de *log* e geração dos arquivos físicos.

Um segundo pacote de classe do protótipo é o pacote `Compilador`, que contém as classes geradas pela ferramenta GALS e que tem o objetivo de auxiliar no processo de análise do código fonte Delphi. Este pacote é representado por completo na Figura 13.

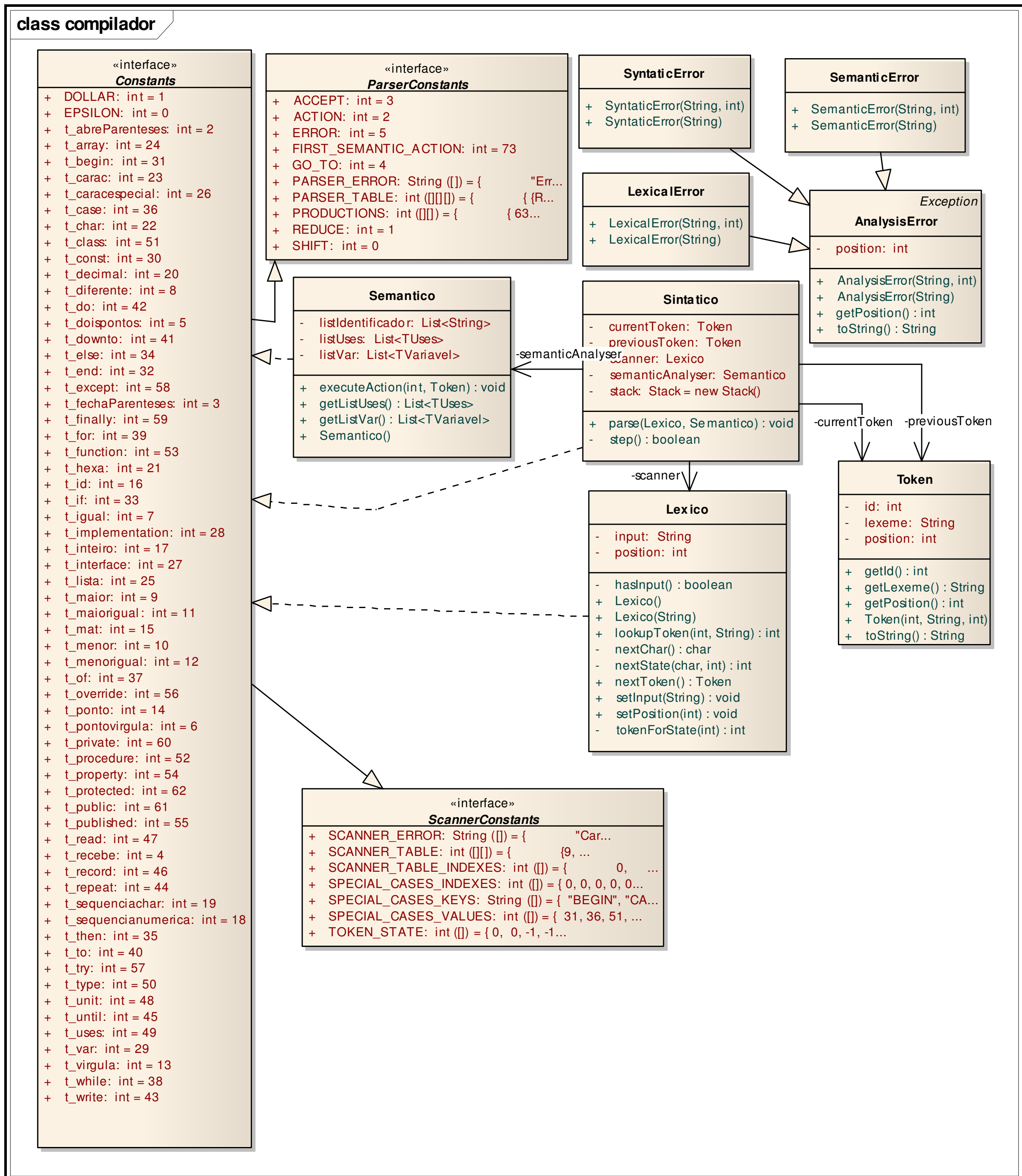


Figura 13 - Diagrama de classes do pacote `compilador`

A classe `Lexico` é a principal classe do pacote `compilador`, esta classe tem o objetivo de montar uma seqüencia de *tokens* que serão percorridos pela classe `Analisador` do pacote `parserarquivo` com o objetivo de montar a metalinguagem. A classe `Lexico` implementa a classe `Constants` que contém os identificadores dos *tokens* que foram descritos na BNF e efetua a análise léxica do fonte Delphi. A classe `Token` contém o identificador da expressão reconhecida, a posição onde o identificador foi encontrado e a descrição do identificado.

A classe `Semantico` tem o objetivo de montar as listas de `uses` e variáveis de acordo com as ações semânticas defininas na gramática. Através dos métodos `getListaUses()` e

getListaVar() é possível extrair as listas encontradas no código analisado.

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.

No pacote classes.DelphiToJava, estão as classes responsáveis por traduzir para a linguagem Java as informações contidas na metalinguagem. A Figura 14 apresenta o diagrama de classes do pacote classes.DelphiToJava.

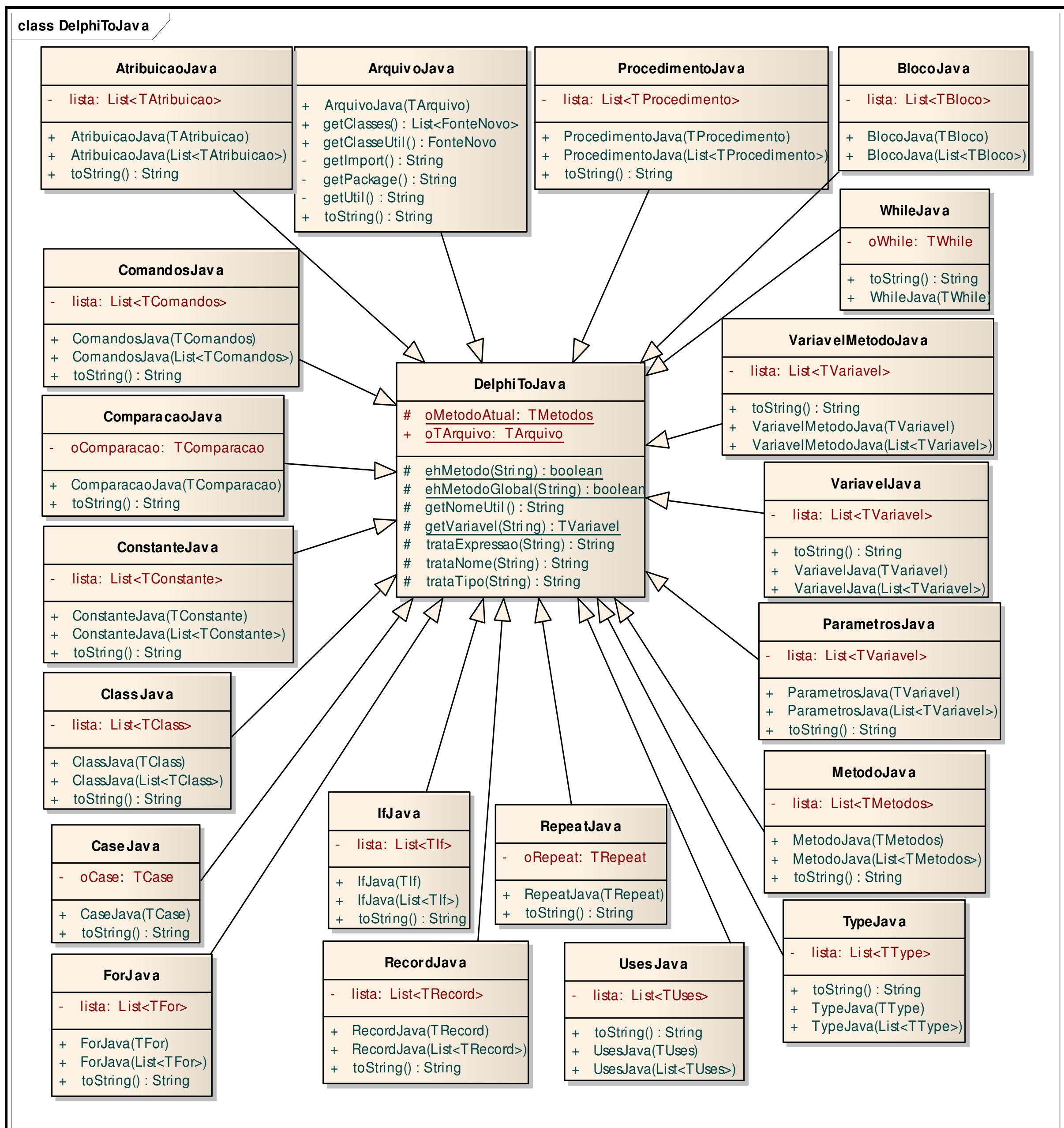


Figura 14 - Diagrama de classes secundário do pacote classes.DelphiToJava

Todas as classes do pacote classes.DelphiToJava estendem a classe

`DelphiToJava`. Esta classe contém um objeto do tipo `TArquivo` que representa o arquivo atual que está sendo traduzido e também uma lista de métodos estáticos que serão usados pelas demais classes para tratar ações comuns entre a maioria das classes.

A estrutura das demais classes que compõem este pacote é similar, cada uma delas trata uma das classes que compõe a estrutura da metalinguagem.

Um exemplo é a classe `ArquivoJava`, que trata a classe `TArquivo` da metalinguagem. Esta classe recebe como parâmetro no seu construtor um objeto do tipo `TArquivo` e a partir das informações contidas neste objeto inicia o processo de montagem do fonte Java.

Na seção 3.3 é descrito de forma mais detalha como é feita a geração do fonte Java.

Outra pacote que existe é o pacote Interface, este pacote contém os métodos de integração com a ferramenta Delphi2Java-II. Através das classes deste pacore a ferramenta Delphi2Java-II inicia o processo de conversão dos eventos. Na Figura 15 é apresentado o diagrama de classes deste pacote.

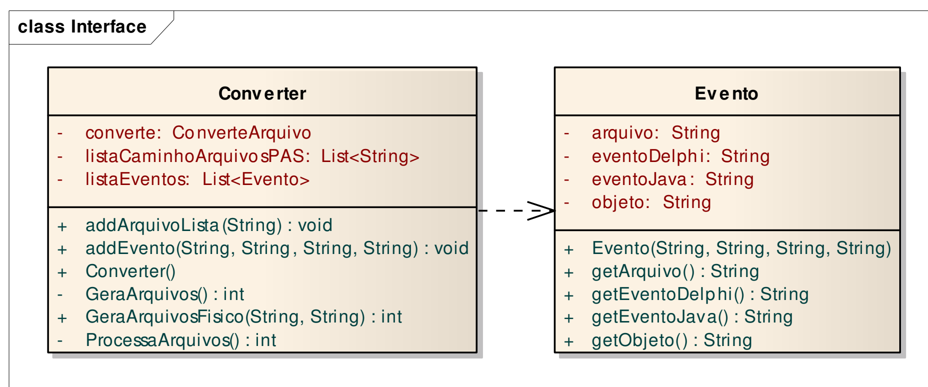


Figura 15 - Diagrama de classes secundário do pacote Interface

Através da classe `Converter` a ferramenta Delphi2Java-II adiciona o arquivo Delphi que deve ser convertido, processa e gera o fonte Java.

3.3 IMPLEMENTAÇÃO

A seguir são mostradas as técnicas e ferramentas utilizadas e a operacionalidade do protótipo.

3.3.1 Técnicas e ferramentas utilizadas

O propósito do protótipo é de ler um fonte Delphi e converter os tratadores de eventos para a linguagem Java, de forma que essa conversão seja capaz de contemplar o maior número de rotinas que compõem os tratadores de evento do fonte Delphi.

A linguagem utilizada para o desenvolvimento do protótipo de conversão dos tratadores de eventos foi Java, utilizando o ambiente de desenvolvimento NetBeans versão 6.1 e a JDK 1.6 e a IDE Eclipse versão3.3.1.1.

Outra ferramenta utilizada durante o desenvolvimento para a tarefa de gerar os analisadores léxicos e sintáticos é o Gerador de Analisadores Léxicos e Sintáticos (GALS) (GESSER, 2003). Essa ferramenta permitiu a geração das rotinas que fazem parte do analisador léxico, sintático e semântico, que foi de fundamental importância para implementar a leitura do fonte Delphi e montagem da metalinguagem.

A seguir são descritas as fases que compõem o processo de conversão dos tratadores de eventos do Delphi para a linguagem Java.

3.3.1.1 Processo inicial

Todo o processo de conversão dos tratadores de eventos se inicia após a ferramenta Delphi2Java-II gerar o arquivo de interface gráfica e o arquivo com as assinaturas dos eventos. Neste momento a ferramenta instância a classe `Converter` e atribui para a classe o arquivo Delphi ligado ao arquivo DFM que foi analisado inicialmente. Outra informação atribuída ao protótipo é uma lista contendo os eventos que a ferramenta Delphi2Java-II foi capaz de identificar no arquivo DFM.

O Quadro 22 demonstra o trecho de código do Delphi2Java-II que faz a interação com o protótipo de conversão dos tratadores de eventos

```
...
1 Converter convEvent = new Converter();
2 arquivoPAS = buscaArquivoPas(arquivoPAS);
3 convEvent.addArquivoLista(arquivoPAS);
4 for(int lint = 0;lint<lstEventos.size();lint++){
5    Evento eve = lstEventos.get(lint);
6    convEvent.addEvento(eve.getArquivo(), eve.getObjeto(),eve.getEventoDelphi(),
eve.getEventoJava());
7 }
8 File fileDest = this.getDiretorio();//caminho onde esta sendo gerada a saída.
9 int retorno = convEvent.GeraArquivosFisico(fileDest.getAbsolutePath(),
arquivoEvento);
...
```

Quadro 22 - Trecho de código do delphi2java-II da interação com o protótipo

É possível identificar as seguintes informações nas linhas do trecho de código acima:

a) na 1ª linha é criada uma instância do protótipo de conversão;
b) na 2ª linha a rotina busca o arquivo Delphi ligado ao arquivo DFM inicialmente analisado;
c) na 3ª linha é atribuição deste caminho para o protótipo;
d) da 4ª até a 6ª linha são as rotinas que atribuem os eventos da ferramenta Delphi2Java-II pode identificar no arquivo DFM;
e) na 9ª linha é chamado o método que faz a conversão.

Após atribuir ao protótipo as informações necessárias para análise do arquivo Delphi, a ferramenta chama o método `GeraArquivosFisico`, passando como parâmetros o local onde estão sendo gerados os arquivos Java e o nome do arquivo Java que contém as assinaturas dos eventos. A partir deste momento a conversão passa a ser gerenciada pelo protótipo e a ferramenta Delphi2java-II só aguarda o retorno do método.

3.3.1.2 Análise do fonte Delphi

Após receber o caminho para o arquivo Delphi que será convertido, a interface atribui esse caminho a classe `parserarquivo.ConverteArquivo`. Esta tarefa é feita através do método `setListaArquivos`, que recebe como parâmetro o caminho. Dentro deste método é feito o processo de validação do caminho consistindo se o arquivo existe.

Para o caminho válido é criado um objeto do tipo `Arquivo`, que irá conter inicialmente o caminho e o nome do arquivo. Durante o processo de conversão novas informações serão atribuídas a estes objetos. Este objeto `Arquivo` será armazenado em uma lista (abaixo é apresentado o Quadro 23 contendo trecho do método `setListaArquivos`).

```
{...}
public void setListaArquivos(List<String> listaCaminhoArquivo) {
try {
    this.listaArquivos.clear();//limpa a lista.
    //varre a lista de caminhos.
    for (String caminhoArquivo : listaCaminhoArquivo) {
       File f = new File(caminhoArquivo);
       if (f.exists()) {
          Arquivo arq = new Arquivo();
          //Atribui o caminho para ao arquivo.
          arq.setCaminhoArquivoOrigem(f.getAbsolutePath());
          arq.setNomeArquivoOrigem(f.getName());
          //Adiciona o objeto do tipo Arquivo a lista.
          this.listaArquivos.add(arq);
       }
    }
 {..}
 }
 {..}
```

Quadro 23 - Método `setListaArquivos` da classe `ConverteArquivo`

Após o processo de mapeamento do fonte que será convertido, a classe `ConverteArquivo` inicia a análise. Este processo ocorre no método `ConverteArquivo.ProcessaArquivos()`. Nesta fase do processo a classe `Analisador` através do método `ProcessaArquivo()` inicia a análise do arquivo

O processo de análise ocorre para o objeto do tipo `Arquivo`, inicialmente é feita a leitura linha a linha do fonte separando em três partes. Uma parte contendo a declaração da `unit`, outra contendo todo o código fonte pertencente à `interface` (`uses`, `types`, `procedures`,...) e uma terceira parte que recebe o trecho na `implementation`, onde fica a implementação dos procedimentos e funções.

Após a divisão do fonte é criado o objeto `oTArquivo` que é da classe `TArquivo`, este objeto é o nó inicial da metalinguagem de um fonte Delphi. O primeiro processamento é feito no trecho de código que contém a declaração da `unit`, este processo tem o objetivo de extrair o nome da `unit` e atribuir esta informação ao objeto `oTArquivo`.

O segundo processamento é feito no trecho de código da `interface`, e tem o objetivo de extrair um objeto do tipo `TInterface`, que irá conter informações da `interface` como, por exemplo, declaração de procedimentos, variáveis, constantes e `types`. Esta análise é feita no método `processaInterface`, este método recebe como parâmetro o trecho de código da interface e retorna um objeto do tipo `TInterface`.

Para fazer a análise do fonte é necessário inicializar a leitura dos *tokens* com base no trecho de código que o método recebeu como parâmetro. Para esta tarefa é criado o objeto interno do tipo `NovoLexico` (denominado `léxico`). Este objeto recebe o trecho de código e inicializa o processo de leitura dos *tokens*.

O processo de leitura do novo *token* se dá através do método `NovoLexico.Avanca()`. Este método faz com que o analisador léxico varra os caracteres em busca do próximo caractere afim de encontrar um estado final. Para cada trecho de código existe um estado final

específico que identifica o fim do bloco.

Após a fase de inicialização do analisador léxico, inicia-se o processo de localizar *tokens* chave no trecho de código da interface. A lista de *tokens* chave para a interface é `uses`, `types`, `var`, `const`, `procedure` e `function`. O que não fizer parte desta lista é omitido.

Para cada *token* chave existem métodos específicos para buscar o trecho de código do *token* chave e outro método para processar o trecho de código encontrado, com o objetivo de identificar e tratar um nó da metalinguagem (o Quadro 24 apresenta o código).

```
while (lexico.getAtual() != null){//varre até o último token.
  switch (lexico.getIdAtual()){//busca o identificador para o token atual.
    case Constants.t_uses://constante que identifica o token uses.
      //chama o método que monta o trecho de código do uses.
      String _uses = getTrechoUses(lexico);
      List<TUses> listaUses = null;
      //Chama o método que processa o trecho encontrado.
      listaUses = processaUses(_uses);
      if (listaUses != null){
         oInterface.addUses(listaUses);
      }
      break;
    case Constants.t_type:// constante que identifica o token type.
   {...}
    default:
          //omite tokens não identificados.
          break;
{...}
```

Quadro 24 - Tratamento de um *token* chave para interface

O método `Analisador.getTrechoUses()` tem o objetivo de varrer os *tokens* que são gerados a partir do trecho de código, até encontrar o *token* ponto e vírgula que é o *token* final para o trecho dos `uses`. Já o método `Analisador.processaUses()` (Apresentado no Quadro 25), recebe este trecho de código do `uses` como parâmetro e através de uma analise semântica deste trecho de código é gerada uma lista de objetos do tipo `TTUses` o retorno do método `Analisador.processaUses()` é adicionado ao objeto `TInterface` ao quais esses `uses` pertencem. Isso é feito através do método `TInterface.addUses()` que recebe a lista de `TUses` e adiciona a `Interface`.

```
private List<TUses> processaUses(String fonte){
   ...
 try {
   Lexico lexico = new Lexico(fonte);
   Sintatico sintatico = new Sintatico();
   Semantico semantico = new Semantico();
   sintatico.parse(lexico, semantico);
   return semantico.getListUses();
 }
 ...
}
```

Quadro 25 - Trecho de código do método `processaUses`

No Quadro 26 é apresentada a gramática para análise do trecho de código da declaração dos uses e da declaração das variáveis.

```
<Prog> ::= <Uses> | <Vars> | î ;
<Uses> ::= uses #20 <LIdent> #21 pontovirgula;
<Vars> ::= var #30 <LVars> pontovirgula | <Vars> var #30 <LVars> pontovirgula ;
<LVars> ::= <LIdent> doispontos #31 <Tipo> #32  | <LVars> pontovirgula <LIdent>
doispontos #31 <Tipo> #32 ;
<Tipo> ::=  hexa | decimal | caracespecial | carac | id;
<LIdent> ::= <Ident> | <LIdent> virgula <Ident> ;
<Ident> ::= id #10;
```

Quadro 26 - Gramática para identificar `uses` e variáveis

Outro exemplo de *token* chave para o trecho de código da interface é o `type`. Ao encontrar o *token* `type` a rotina faz o tratamento similar ao dado ao *token* `uses`. Ele busca o trecho de código do `type` através de um método específico. O método responsável por esta tarefa é o `Analisador.getTrechoType()` que retorna o trecho de código da estrutura `type`. No Quadro 27 é demonstrada a rotina que trata o trecho de código da estrutura `type`.

```
private String getTrechoType(NovoLexico lexico) {
  String retorno = "";
  boolean ehFora = false;
  try {
     if (lexico.getAtual() != null){
        if (lexico.getIdAtual() == Constants.t_type){
             lexico.Avanca();
        }
        do{
          retorno += " " + lexico.getLexemeAtual();
          lexico.Avanca();
          if (lexico.getIdAtual() == Constants.t_class ||
              lexico.getIdAtual() == Constants.t_record) {
            ehFora = false;
          } else if (lexico.getIdAtual() == Constants.t_end) {
            ehFora = true;
          }
        //cláusula que indica o final da estrutura.
        }while(lexico.getIdProximo() != Constants.t_type &&
        (lexico.getIdProximo()!= Constants.t_const || !ehFora) &&
        (lexico.getIdProximo()!= Constants.t_var || !ehFora) &&
        (lexico.getIdProximo()!= Constants.t_procedure || !ehFora) &&
        (lexico.getIdProximo()!= Constants.t_function || !ehFora) &&
        (lexico.getIdProximo()!= Constants.t_implementation));
        return retorno;
     } else {
        Util.printLog("  ATENCAO o 'léxico' entrou com valendo nulo no método 'getTrechoType'
antes de encontrar o token final;");
        return null;
     }
     {...}
    }
```

Quadro 27 - Rotina que busca o trecho de código do `type`

O método `getTrechoType()` vai armazenando os valores dos *tokens* até encontrar o final da estrutura, que no caso do `type` seria uma declaração de constante, variável, procedimentos ou funções da `unit`, uma declaração de outro `type` ou a definição da `implementation`.

Após ter retornado o trecho de código da estrutura `type`, é chamado o método `Analisador.processaType()`, que recebe como parâmetro o trecho de código que foi encontrado. O método `processaType()` segue mesma lógica do método `processaIterface()`. Ele instância um objeto da classe `NovoLexico` e inicializa como o trecho de código que recebeu como parâmetro.

Como ocorre no `processaInterface()`, o `processaType()` vai varrer os *tokens*

encontrados buscando por *tokens* chaves para a estrutura `type`, com, por exemplo, `class` e `record`. Ao encontrar um *token* chave ele trata cada um deles, da mesma forma como é feito o tratamento para os *tokens* chaves da interface, buscando o trecho de código do *token* e processando o trecho com um método específico.

O processo de tratamento para os demais *tokens* chaves do trecho da interface é similar ao tratamento aplicado no *token* `uses` e `type`. Existe um método para encontrar o trecho de código do *token* e outro método para tratar este trecho de código e retornar um objeto específico para *token*.

Com o fim do procedimento de análise da interface, o método `processaInterface()` retorna um objeto do tipo `TInterface`. Este retorno é adicionado ao atributo `oInterface` do objeto `oTArquivo` através do método `setOInterface()`.

Depois de finalizada a tarefa de análise do trecho de código da interface, se inicia a análise do trecho de código da `implementation`. Esta tarefa é realizada no método `processaImplementation()`, que recebe como parâmetro o trecho de código da `implementation`. O principal objetivo do tratamento deste trecho de código é encontrar e converter a implementação dos tratadores de eventos que foram declarados na interface.

O processo é similar ao aplicado na análise dos demais trechos. É instanciado um objeto do tipo `NovoLexico` com o nome `lexico` que é responsável pela análise léxica do trecho de código. Depois de inicializado o analisador léxico, é feita a varredura dos *tokens* em busca de *tokens* chave para a `implementation`. Os *tokens* chave que são tratados na `implementation` são `uses`, `var`, `const`, `procedure` e `function`. O tratamento aplicado aos trechos de código da declaração de `uses`, variáveis e constantes da `implementation` são os mesmos processos utilizados nos trechos de código da interface.

É importante detalhar somente o tratamento dado quando é encontrado um *token* `procedure` ou `function`. Nos dois casos a captura do código fonte de cada *token* é extraído utilizando o método `getTrechoMetodo`, que recebe como parâmetro o `NovoLexico` atual. Este método varre os *tokens* até que encontre uma situação onde seja contatado fim do procedimento ou função (o Quadro 28 apresenta o código do método).

```
private String getTrechoMetodo(NovoLexico lexico) {
  String retorno = "";
  int qtdBegin = 0;
  int qtdEnd = 0;
  int qtdMet = 0;//isso é usado, pois podem existir métodos declarados dentro de métodos.
  try {
     do {
        if (lexico.getIdAtual() == Constants.t_procedure ||
            lexico.getIdAtual() == Constants.t_function) {
            qtdMet++;
        }
     //Sempre que o nro de Begin e END forem iguais, verifica se é final.
        if (qtdBegin == qtdEnd && qtdBegin > 0) {
            qtdMet--;
            qtdBegin--;
            qtdEnd--;
            //Se fechou o nro de métodos, sai fora.
            if (qtdMet == 0) {
                break;
            }
        }
        //Verifica se é um início de bloco, um bloco pode ser um Begin, case, except ou
finally.
        if (lexico.getIdAtual() == Constants.t_begin ||
            lexico.getIdAtual() == Constants.t_case ||
            lexico.getIdAtual() == Constants.t_except ||
            lexico.getIdAtual() == Constants.t_finally) {
            qtdBegin++;
        }
        //Verifica se é fim de bloco.
        if (lexico.getIdAtual() == Constants.t_end) {
            qtdEnd++;
        }
        retorno += " " + lexico.getLexemeAtual().toString();
        lexico.Avanca();
    }while(lexico.getAtual() != null);
    return retorno;
  {...}
}
```

Quadro 28 - Trecho de código do método `getTrechoMetodo`

É importante ressaltar que na `implementation` o método `getTrechoMetodo` busca todo o bloco de código da `function` ou `procedure com suas` variáveis, constantes e sub-métodos.

Após ter encontrado o trecho de código, se inicia a fase de análise com o objetivo de buscar o bloco de código ligado a este método, suas variáveis, constantes e possíveis sub procedimentos. A fase de análise deste trecho é similar as análises aplicadas aos demais trechos de código, é criado um objeto do tipo `NovoLexico` denominado `lexico` que é responsável pela análise léxica do trecho específico.

Primeiramente o analisador léxico busca o nome do método que se esta analisando. Para isso existe o método `Analisador.getNomeMetodoImplementation()`, que recebe como parâmetro o objeto léxico atual e varre os *tokens* buscando as informações de nome e a quem o procedimento ou função pertence. Esta varredura se dá até encontrar o final de uma declaração de método que se dá quando o próximo *token* for uma declaração de variável, declaração de constante, procedimento, função ou bloco de código.

Tendo sido recuperado nome do procedimento ou função, é preciso encontrar este nome na lista de métodos na interface do arquivo. Para isso o objeto `oTArquivo` que é do tipo `TArquivo` e que pertence à classe `Analisador` contém um método denominado

TArquivo.getMetodo(), que recebe como parâmetro o nome do método que se quer localizar. Dentro do método getMetodo da classe TArquivo é feita uma chamada ao objeto oInterface da classe TArquivo e é passada a ele a responsabilidade de encontrar em sua estrutura o procedimento ou função com o nome informado (o Quadro 29 apresenta o código do método getMetodo da classe TInterface). A busca é feita inicialmente na lista de métodos da interface e se não obtiver sucesso passa a buscar o método em cada um dos TTypes da estrutura.

```
public TMetodos getMetodo(String nomeClasse, String nomeMetodo) {
  //Varrer a lista de métodos da interface.
  for (TMetodos oMetodo : this.listaMetodos) {
    if (oMetodo.getNome().equals(nomeMetodo)) {
      return oMetodo;
    }
  }
  //Varrer a lista de types da interface a procura do método.
  for (TType oType : this.listaTypes) {
    //Passa ao type a responsabilidade por fazer a consulta em sua lista.
    TMetodos ret = oType.getMetodo(nomeClasse, nomeMetodo);
    if (ret != null) {
       return ret;
    }
  }
  return null;// se não encontrar o método retorna nulo.
}
```

Quadro 29 - Código do método getMetodo da classe TInterface

Se for obtido sucesso na localização do método, é dado início ao processo de tratamento do código do método. Este procedimento é tratado dentro do método Analisador. trataMetodo, que recebe como parâmetro o objeto lexico atual e o método encontrado anteriormente.

A análise do trecho de código é similar aos procedimentos adotados para análise dos demais trechos: é feita uma varredura dos *tokens* a procura de *tokens* chave, que para o trecho de código da *implementation* do método são: procedure, function, var, const, type e begin, os demais *tokens* encontrados nesta etapa do processo serão descartados.

Nesta fase do processamento é importante dar uma ênfase maior ao tratamento dado a estes *tokens* chaves, já que está sendo analisando o tratador de evento em si, que é o alvo deste trabalho.

Para os *tokens* function e procedure são dados os mesmos tratamentos, porque para a metalinguagem estes são somente métodos. A única diferença entre os dois é que pode ou não ter retorno. A forma de processamento dos *tokens* é similar em todos os casos, e foi descrita anteriormente. A primeira etapa é buscar o trecho de código da function ou procedure, e para esta tarefa é utilizado o método Analisador. getTrechoMetodo() que é o mesmo método utilizado para extrair o trecho de código dos métodos da implementation (item já detalhado no Quadro 28).

Tendo obtido o trecho de código é dado início ao processo de análise do mesmo. Para

esta tarefa existe o método `Analisador.processaMetodoInterno()` que recebe como parâmetro o trecho de código encontrado anteriormente.

Como este sub procedimento ou função não foi declarado na interface do fonte Delphi, o método `Analisador.processaMetodoInterno()` trabalha de uma forma um pouco diferente do método `Analisador.processaMetodo()`, porque além de busca as variáveis, constantes e o código fonte, deve buscar seu nome e criar um novo objeto do tipo `TMetodos`.

Após esta fase inicial de declaração do novo submétodo é chamado o método `Analisador.trataMetodo()` passando com parâmetro o objeto `lexico` atual e o objeto do tipo `TMetodos` que foi criado, para que seja feito o processamento interno deste procedimento ou função. Desta forma, independe o número de submétodos que forem encontrados.

Para os *tokens* `var`, `const` e `type` o processamento é o mesmo aplicado quando esses *tokens* são encontrados no processamento da interface do fonte Delphi. A única diferença é que na metalinguagem estas serão atribuídos ao método atual que se está sendo analisado, independente se este é um método principal ou um submétodo.

Já o tratamento aplicado o *token* `begin`, é feito de forma especial através do método `Analisador.trataBloco()`, que recebe como parâmetro o objeto `lexico` atual e um objeto do tipo `TBloco` que inicialmente está valendo nulo. Dentro do método é feito o tratamento dos comandos Delphi que compõem o bloco (exemplo: `If`, `For`, `While`, ...), através do método `Analisador.trataComando()` passando como parâmetro o objeto `lexico` que o método `trataBloco` recebeu como parâmetro (no Quadro 30 é apresentado um trecho de código do método `trataComando`).

```
private TComandos trataComando(NovoLexico lexico) {
 try {
   TComandos oComando = null;
   String _Trecho = "";
   switch (lexico.getIdAtual()) {
     case Constants.t_if://trata o if.
       _Trecho = this.getTrechoIf(lexico);
       oComando = this.processaIf(_Trecho);
       break;
     case Constants.t_while://trata o while.
       _Trecho = getTrechoWhile(lexico);
       oComando = this.processaWhile(_Trecho);
       break;
     case Constants.t_for://trata o for.
{…}
}//fim do método trataComando.
```

Quadro 30 - Trecho de código do método `trataComando`

Para cada *token* que identifica um comando Delphi que é tratado pelo protótipo (o Quadro 31 relaciona os *tokens* encontrados nos comandos Delphi reconhecidos pelo protótipo), é aplicado um tratamento específico.

NOME TOKEN	COMANDO DELPHI ASSOCIADO
if	Comando condição If
for	Comando de repetição for
while	Comando de repetição while
repeat	Comando de repetição repeat
Case	Comando de condição Case
Id	Identifica uma atribuição ou chamada de procedimento

Quadro 31 - *Tokens* de comandos

Ao identificar um dos *tokens*, é chamado um método específico para buscar o trecho de código que trata o comando associado ao *token*. Após isso é chamado outro método específico para cada *token* que recebe como parâmetro o trecho que código identificado no método anterior. Como as classes que identificam os comandos da linguagem Delphi na metalinguagem estendem a classe do tipo `TComandos` todos os métodos que tratam o trecho de código de cada comando são armazenados em uma lista de `TComandos`.

Um exemplo desta lógica é o tratamento aplicado ao *token* `If`. É chamado o método `Analisador.getTrechoIf()` passando como parâmetro o objeto `lexico` que o método `trataComando` recebeu como parâmetro. Este método extrai o trecho de código do comando `If`, e atribui este retorno a uma variável que será passada como parâmetro para o método `Analisador.processaIf()`. O método `processaIf()` tem o objetivo de analisar o trecho de código que foi passado como parâmetro e retornar um objeto do tipo `TIf`.

Com o trecho de código que existe entre o *token* `If` e o token `then` é criado um objeto do tipo `TComparacao` e atribuído ao atributo `comparacao` da classe `TIf` através do método `setComparacao()`. Após a tarefa de atribuir a comparação ao objeto da classe `TIf` é iniciado o processo de análise dos comandos que estão vinculados ao comando `If`. Isso é feito através do comando `trataComando()`, que recebe como parâmetro o objeto `lexico` criado no método `processaIf()`. Este objeto `TComandos` resultante do método `trataComando()` é atribuído bloco "se" da classe `TIf`.

Se o próximo *token* não for nulo isso indica ao método que ele deve buscar mais uma lista de comandos e atribuir este bloco "senão" do objeto do tipo `TIf`.

Durante a análise dos *tokens* se forem encontrados *tokens* que identificam comandos esses serão processados e adicionados ao comando atual, e desta forma até encontrar o nó mais extremo da lista de comandos que formam a metalinguagem.

3.3.1.3 Estrutura dos arquivos gerados

Após a classe `Analisador` ter obtido sucesso na análise do fonte Delphi e ter sido

possível carregar as informações para a metalinguagem, o protótipo da início ao processo de geração do código fonte Java. O processo de varredura da metalinguagem e da geração dos fontes Java serão descritos a seguir.

Antes de descrever a forma de geração dos arquivos Java, será descrita a estrutura dos arquivos que são gerados. Para cada fonte Delphi será criada uma pasta ou pacote com o nome do fonte dentro desta pasta será adicionados todos os arquivos Java que forem gerados pelo protótipo. Para cada arquivo Delphi convertido, será gerada uma classes Java contendo as declarações de variáveis, procedimentos e funções globais e seu nome por padrão será o nome da `unit` do fonte Delphi acrescido do sufixo "`_Util.java`".

Para cada `class` ou `record` que estiver declarado na `unit`, será criada uma nova classe Java. Esta classe será gerada com o nome da `class` ou `record` que originou o arquivo. Todas as classes Java que forem geradas farão parte do mesmo pacote, e o pacote terá o mesmo nome da pasta onde os arquivos estão sendo gerados.

3.3.1.4 Geração do código Java

O processo de geração dos arquivos Java ocorrem através o método `GeraArquivos()` da classe `ConverteArquivo`. Este método recebe como parâmetros o caminho onde a ferramenta Delphi2Java-II gerou os arquivos contendo a interface gráfica e o arquivo com as assinaturas dos métodos dos tratadores de eventos e o nome do arquivo com as assinaturas dos tratadores de eventos.

O método `GeraArquivos()` instancia um objeto da classe `Gerador` denominando `ger`. Durante o processo de criar o objeto do tipo `Gerador` é passado como parâmetro para o construtor a lista de objetos do tipo `Arquivo` que contém a metalinguagem de cada arquivo analisado com sucesso.

A classe `Gerador` possui o método `ConverteArquivos()` que tem o objetivo de dar início ao processo de geração dos fontes Java, e atribuir os resultado obtidos ao objeto do tipo `Arquivo`. O Quadro 32 mostra o trecho de código do método `ConverteArquivos()`.

O processo de geração do código fonte Java consiste basicamente, em instanciar um objeto da classe que tem o conhecimento sobre determinado nó da metalinguagem e chamar o método `toString()` de cada um deles. Estas classes que tem o conhecimento sobre determinado nó da metalinguagem estão agrupadas no pacote `classes.DelphiToJava`. Este

pacote foi descrito no Figura 14.

```
public int ConverteArquivos() {
  for (Arquivo oArquivo : this.getListaArquivos()) {
    ArquivoJava oArquivoJava = new ArquivoJava(oArquivo.getTArquivo());
    String fonte = oArquivoJava.toString();
    //método para gerar as classes.
    oArquivo.addFonteNovo(oArquivoJava.getClasses());
    oArquivo.setCodigoFonteArquivoNovo(fonte);
  }
  return 1;
}
```

Quadro 32 - Código fonte do método `ConverteArquivos` da classe `Gerador`

No Quadro 32 é apresentado como exemplo o objeto `oArquivoJava` da classe `ArquivoJava` recebe como parâmetro no seu construtor um objeto do tipo `TArquivo`, este objeto chama o método `getClasses()` que retorna a lista de objetos do tipo `FonteNovo`. Esta lista é adicionada ao atributo `listaFonteNovo` da classe `Arquivo`.

No Quadro 33 é apresentado o método `getClasses` da classe `ArquivoJava`. Este método cria as classes Java e descreve como será gerado o código fonte Java.

```
public List<FonteNovo> getClasses(){
   List<FonteNovo> listaFonte = new ArrayList<FonteNovo>();
   //adiciona a classe Util.
   String fonte = this.getPackage();
   fonte += this.getImport();
   fonte += this.getUtil();
   listaFonte.add(new FonteNovo(this.getNomeUtil() + ".java", fonte));
   for(TType oType:oTArquivo.getOInterface().getListaTypes()){
     //adiciona um NovoFonte para cada class do type.
     for(TClass oClass :oType.getListaClasse()){
        String nome = oClass.getNome();
        fonte = this.getPackage() + "\n";
        //só cria novos fontes para classes que não são de formulário.
        if(!oClass.getTipo().toLowerCase().equals("tform")){
           fonte += (new ClassJava(oClass)).toString();
           listaFonte.add(new FonteNovo(nome + ".java", fonte));
        }
     }
     //adiciona um NovoFonte para cada record do type.
     for(TRecord oRecord:oType.getListaRecord()){
        String nome = oRecord.getNome();
        fonte = this.getPackage() + "\n";
        fonte += (new RecordJava(oRecord)).toString();
        listaFonte.add(new FonteNovo(nome + ".java", fonte));
     }
   }
   ...
   return listaFonte;
}
```

Quadro 33 - Código fonte do método `getClasses` da classe `ArquivoJava`

Para geração da classe `util` inicialmente é chamado o método `getPackage()` que retorna um valor do tipo String contendo a declaração do pacote a que este fonte irá pertencer. Após adicionar o pacote é chamado o método `getImport()`. Este método escreve todos os `imports` que o arquivo Java irá ter. Cada `import` equivale a um `uses` do fonte Delphi.

Em seguida é chamado o método `getUtil()`. Este método tem o objetivo de montar a classe principal que contém as declarações de variáveis, constantes, procedimentos e funções globais de um fonte Delphi.

Depois de montar a classe `Util`, o método `getClasses` varre todos os `types` do `TArquivo` gerando um novo objeto do tipo `NovoFonte` para cada `class` e `record` que ele

encontrar na metalinguagem. As classes do tipo `tform` não serão geradas neste momento, pois seus métodos serão inseridos na classe contendo as assinaturas dos métodos e que foi gerada pela ferramenta Delphi2Java-II.

No Quadro 34 é demonstrado um exemplo de código Delphi para conversão. Este código possui um `class` do tipo `TForm` que contém um atributo e um procedimento, possui uma `record` com dois atributos. Existe também um procedimento global para o fonte denominado `Mensagem`.

```
unit uForm1;
interface
uses
  Windows, Messages, SysUtils, Variants, Classes, Graphics, Controls, Forms,
  Dialogs, StdCtrls;
type
  TForm1 = class(TForm)
    btnAtribuir: TButton;
    editRecebe: TEdit;
    Label1: TLabel;
    procedure btnAtribuirClick(Sender: TObject);
  private
    { Private declarations }
  public
    { Public declarations }
  end;
var
  Form1: TForm1;

implementation

{$R *.dfm}
procedure TForm1.btnAtribuirClick(Sender: TObject);
var valor : String;
begin
  valor := 'Valor fixo. Este foi definido no Evento Click.';
  editRecebe.Text := valor;
end;
end.
```

Quadro 34 - Fonte Delphi para conversão

Para o arquivo Delphi descrito no Quadro 34 serão gerados inicialmente pela ferramenta Delphi2Java-II dois arquivos Java, um contendo a interface gráfica e outro contendo as assinaturas dos eventos.

Após este processo o protótipo escreve no arquivo que contém as assinaturas dos eventos, os métodos convertidos com base nas informações contidas na metalinguagem. O Quadro 35 mostra o método que será escrito no fonte Java.

```
private void btnAtribuirActionPerformed(java.awt.event.ActionEvent evt){
    valor = "Valor fixo. Este foi definido no Evento Click.";
    editRecebe.text = valor;
}
```

Quadro 35 - Método que será escrito no arquivo com as assinaturas dos eventos

O próximo arquivo a ser gerado é o `util`, que irá conter os métodos, variáveis e constantes identificados no fonte Delphi. No Quadro 36 é possível visualizar o arquivo gerado. O simbolo `*NP*` siginfica não processado, indicando que esta `unit` não foi encontrada pelo Delphi2Java-II.

```
package uForm1;
 // *NP* import Windows;
 // *NP* import Messages;
 // *NP* import SysUtils;
 // *NP* import Variants;
 // *NP* import Classes;
 // *NP* import Graphics;
 // *NP* import Controls;
 // *NP* import Forms;
 // *NP* import Dialogs;
 // *NP* import StdCtrls;

//Classe principal, contem os procedimentos e metodos globais.
 public class uForm1_Util {
   //lista de variaveis da INTERFACE
    private TForm1 form1;
//lista de variaveis da IMPLEMENTATION
//lista de Constantes da INTERFACE
//lista de Constantes da IMPLEMENTATION
//lista de METODOS da UNIT
}//class Util
```

Quadro 36 - Código fonte da classe `util`

Após o término da geração dos arquivos Java é gerado um arquivo com formato texto que apresenta um resumo do que o protótipo identificou no fonte Delphi. O conteúdo deste arquivo é apresentado no Quadro 37.

```
=======================================================
 ARQUIVO: uform1.pas
 Unit: uForm1
 Encontrados 10 USES para a INTERFACE deste arquivo:
 Encontrados 0 USES para a IMPLEMENTATION deste arquivo:
 Encontrados 1 VARIAVEIS para a INTERFACE deste arquivo:
 Encontrados 0 VARIAVEIS para a IMPLEMENTATION deste arquivo:
 Encontrados 0 CONSTANTES para a INTERFACE deste arquivo:
 Encontrados 0 CONSTANTES para a IMPLEMENTATION deste arquivo:
 Encontrados 1 TYPES para a INTERFACE deste arquivo:
  TYPES
  Encontrados 1 CLASS para a TYPE deste arquivo:
   CLASS 'TForm1' do TYPE
   Encontrados 3 ATRIBUTOS para esta CLASS:
   Encontrados 0 PROPERTY para esta CLASS:
   Encontrados 1 METODOS para esta CLASS:
 Encontrados 0 RECORD para a TYPE deste arquivo:
 Encontrados 0 TYPES para a IMPLEMENTATION deste arquivo:
 ------------------------------------------------------
```

Quadro 37 - Arquivo texto com informações obtidas na leitura do fonte Delphi

Após a geração com sucesso dos fontes Java, o método `Gerador.ConverteArquivos()` retorna 1 para o método `ConverteArquivo.GeraArquivos()` que devolve para a interface este resultado. Com o sucesso na operação de geração a interface do protótipo inicia o processo de geração dos arquivos físicos através do método `ConverteArquivo.GeraArquivoFisico()`, que recebe como parâmetro o local onde deverá ser gerado os fontes físicos.

Este método `GeraArquivoFisico()` tem a única tarefa de varrer a lista de objetos do tipo `Arquivo` gerando um arquivo físico para cada um dos objetos `NovoFonte` do `Arquivo`. Este arquivo é gerado com o conteúdo do atributo `codigoFonte` do objeto `NovoFonte` finalizando assim o processo de geração dos fontes Java.

3.3.2 Operacionalidade da implementação

Como o objetivo do trabalho foi de estender a ferramenta Delphi2Java-II, a interface se maneve a mesma do projeto atual. Como é apresentada na Figura 16.

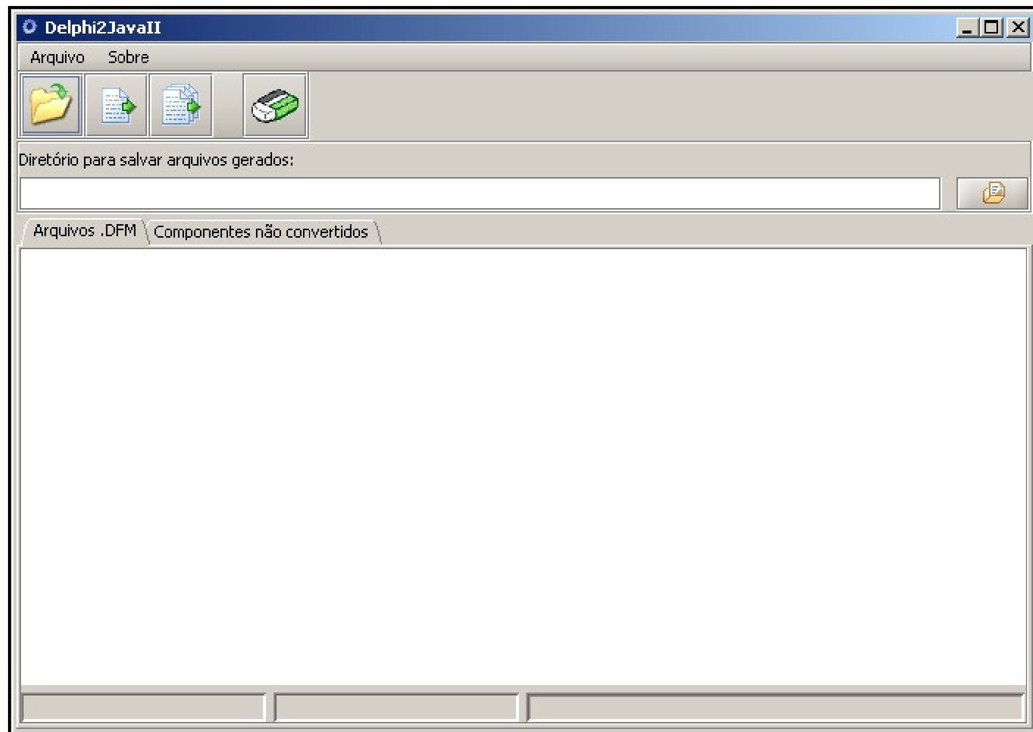


Figura 16 - Interface da ferramenta Delphi2Java-II

A interface possui um menu de opções, as principais funções estão também disponíveis em uma barra de menu. Esta barra possui quatro botões sendo o primeiro para seleção dos arquivos DFM que serão convertidos, um segundo botão faz a conversão de um determinado arquivo da lista de arquivos que foram selecionados para conversão, um terceito botão aplica a conversão a todos os arquivos da lista e um quarto botão limpa as informações da janela. A Figura 17 mostra os botões descritos acima.

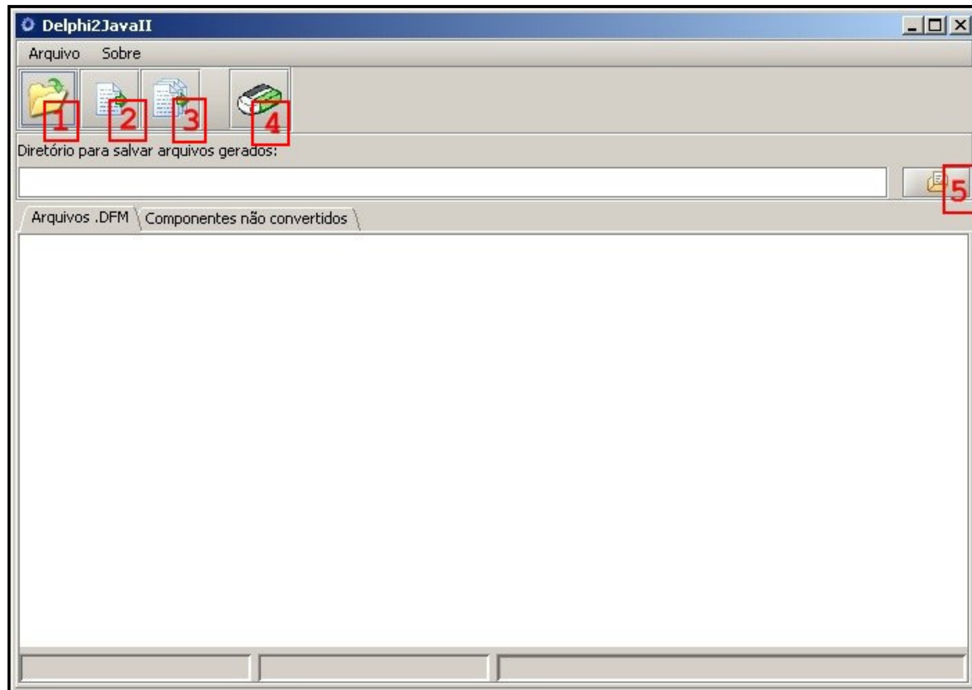


Figura 17 - Botões de ação da interface

Ainda na interface existe um quinto botão, que tem a função de selecionar a pasta onde os arquivos serão gerados.

Ao pressionar o botão 1 a ferramenta disponibiliza uma janela onde é possível selecionar os arquivos DFM que serão convertidos (Figura 18).

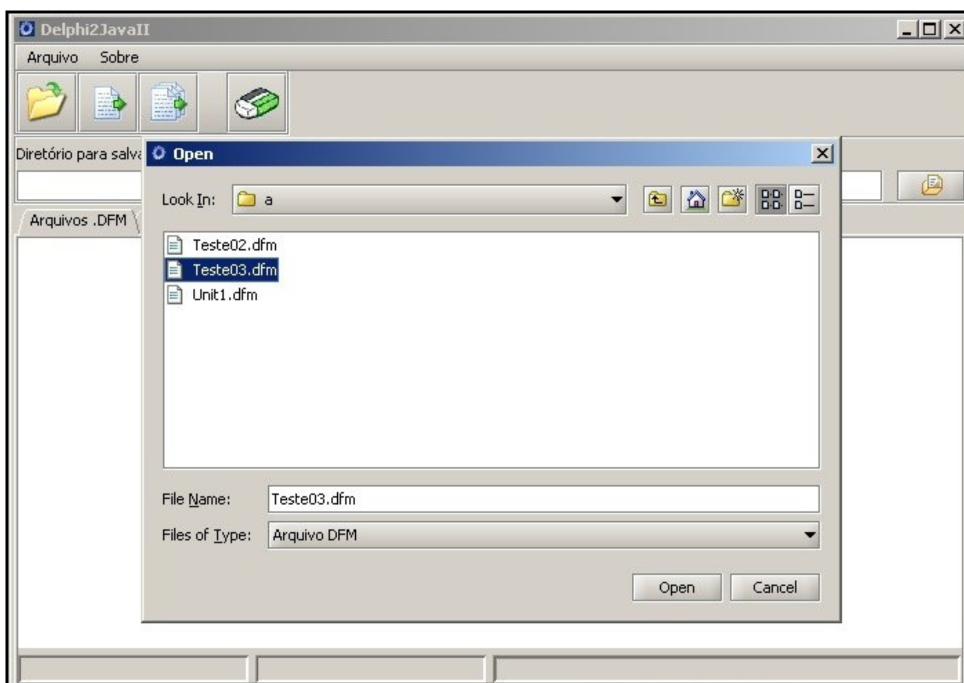


Figura 18 - Selecionar arquivos DFM

Após selecionar os arquivos que serão convertidos deve informar o destino onde os arquivos serão gerados. Ao pressionar o botão número 5 (Figura 17) a ferramenta disponibiliza uma janela para seleção deste caminho de destino (Figura 19).

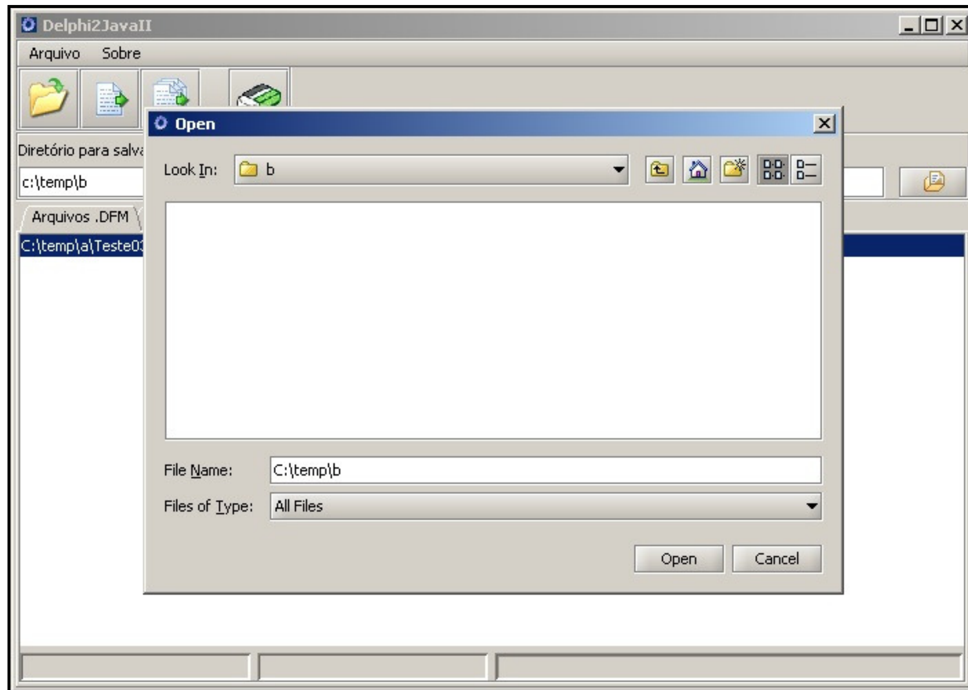


Figura 19 - Escolhe o local onde os arquivos serão gerados

Com um click sobre o botão 2 ou 3 (Figura 17) a ferramenta inicia o processo de conversão dos arquivos Delphi, com o fim do processo é apresentada uma mensagem indicando se houve ou não sucesso na geração dos arquivos (Figura 20).

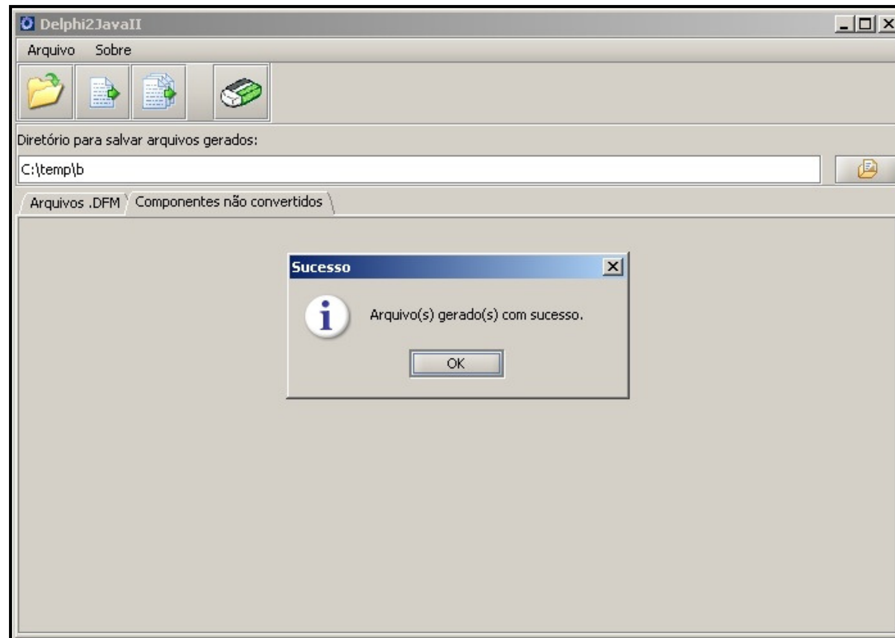


Figura 20 - Mensagem de fim da conversão

3.4 RESULTADOS E DISCUSSÃO

Os resultados alcançados pelo protótipo vem a agregar ao projeto Delphi2Java-II a funcionalidade de conversão parcial dos tratadores de eventos. Esta funcionalidade não era tratada nos projetos anteriores como, por exemplo, Fonseca (2005) ou Silveira (2006).

Como descrito no texto, o processo de analise dos fontes Delphi envolve a identificação de trechos característicos de uma `unit` quais sejam região de interface (declaração de tipos, constantes, classes e variáveis globais ao projeto) e região de *implementation* (declaração de tipos, constantes, variáveis, classes, corpo dos métodos e demais procedimentos e funções).

Toda a informação mapeada através da leitura do fonte Delphi é agrupada em uma metalinguagem a qual constitui a base para a geração do fonte Java.

A partir das informações da metalinguagem o protótipo é capaz de reescrever o arquivo gerado pela ferramenta Delphi2Java-II contendo as assinaturas dos eventos e a geração dos demais arquivos Java contendo as outras estruturas como classes e `records` declarados no fonte Delphi.

Apesar da conversão sintática de um fonte Delphi ter sido realizada, o fonte gerado não é completo, visto que não são considerados: (i) as `units` das bibliotecas de *run-time* do

Delphi, (ii) as `units` pertencentes a componentes instalados no ambiente Delphi e, (iii) as demais `units` componentes do projeto sendo convertido. Com isto, uma série de elementos declarados na parte de interface destes módulos não podem ser devidamente traduzida, impossibilitando em alguns casos a compilação do fonte Java gerado.

4 CONSIDERAÇÕES FINAIS

Segundo Fonseca (2005, p. 57), uma limitação da ferramenta Delphi2java-II era relacionada ao grupo de componentes selecionados para conversão, o qual não implementava todo o conjunto de componentes visuais existentes no ambiente Delphi. Embora o sistema tenha sido ampliado para suportar componentes de banco de dados conforme descrito em Silveira (2006) permaneceu a restrição de que o usuário tem que converter manualmente o código dos tratadores de eventos, o que conduz a erros e dificulta o processo de conversão.

Em função do tempo disponível para a conclusão do trabalho e de evidente complexidade do mesmo, optou-se pelo uso de alguns trechos da BNF para identificação de declaração de variáveis e em outros momentos lançou-se mão do uso de expressões regulares.

Esta decisão de projeto viabilizou a condução do trabalho embora seja necessário algum esforço de programação por parte do conversor de forma a tornar o fonte gerado compatível em Java. Portanto o presente projeto pode ser caracterizado como prova de concreta na medida em que demonstra a viabilidade de conversão do código fonte Object Delphi para Java.

Um dos resultados mais expressivos para o protótipo foi a montagem da metalinguagem. Esta estrutura apresentou um grau de abstração do código fonte origem afim de dar suporte a geração de código fonte em Java e futuras extensões.

Como destacado não foi possível a conversão por completo do código fonte afim de permitir sua compilação, mas a estrutura resultante da conversão auxilia o desenvolvedor no seu trabalho, pois algumas estruturas como declaração de variáveis, definição de classes usadas no fonte entre outras estão bem próximas da fase final de conversão em alguns casos.

4.1 EXTENSÕES

Como extensões para este trabalho sugerem-se:

a) conversão sintática das bibliotecas de *run-time* do Delphi para Java;
b) conversão semântica das bibliotecas de *run-time* do Delphi para as bibliotecas de *run-time* do Java;
c) indicação da necessidade de conversão sintática e semântica das funcionalidades de

componentes do Delphi utilizados em um projeto a ser convertido;

d) varredura e conversão de todos os arquivos que compreendem um determinado projeto Delphi a ser convertido;
e) integração das rotinas do protótipo com a ferramenta de projeto DelphiToJava-II, a fim de ampliar as funcionalidades já existentes no projeto.

5 REFERÊNCIAS BIBLIOGRÁFICAS

ALVES, Pedro. **O Projeto Mono e a disputa Java × .NET.** [S.l.], 2007. Disponível em: <http://www.dotnug.com/index.php?q=o-projeto-mono-e-a-disputa-java-net>. Acesso em: 06 nov. 2007.

ANNES, Ricardo. **Programando em Delphi 6.** [S.l.], 2005. Disponível em: <http://pucrs.campus2.br/~annes/delphi6int.pdf>. Acesso em: 05 jun. 2008.

BORBA, Paulo. **Programação 3:** orientação a objetos e Java. [S.l.], [2000?]. Disponível em: <http://www.cin.ufpe.br/~if101/turmaatual/aulas/aula8/transparencias/javavsdelphi.html>. Acesso em: 02 nov. 2007.

BRINK, Rob F. M. van den. **Delphi 7.0 Object Pascal grammar.** Netherlands, 2006. Disponível em: <https://www2.fh-augsburg.de/informatik/projekte/idb/ss2007/admin/modulo/Structorizer/D7Grammar.grm>. Acesso em: 18 nov. 2007.

CANTU, Marco. **Dominando o Delphi 7:** a bíblia. São Paulo : Pearson Education do Brasil, c2003. xxviii, 801p, il. Tradução de: Mastering Delphi 7.

CARVALHO, Faical Farhat de. **Programação orientada a objetos usando o Delphi 3.** 2. ed. São Paulo: Erica, 1998. 505p, il.

CESAR, Ricardo. Java X .Net: disputa acirrada no mercado nacional. **ComputerWord**, São Paulo. Disponível em: <http://computerworld.uol.com.br/infra_estrutura/2003/06/17/idgnoticia.2006-05-15.5204280289>. Acesso em: 05 out. 2007.

DELAMARO, Márcio Eduardo. **Como construir um compilador utilizando ferramentas Java.** São Paulo: Novatec, 2004. xii, 308 p, il.

FERNANDES, Jorge H. C. **Arquitetura de tratamento de eventos em programas com GUI - graphical user interfacei.** [S.l.], 2002. Disponível em: <http://www.cic.unb.br/~jhcf/MyBooks/itjava/slides/ArquiteturaDeTratamentoDeEventosEmJavaAWT-25slides.pdf>. Acesso em: 01 jun. 2008.

FERREIRA, João P. A. de O. **XO$_2$**: Um gerador de código MDA baseado em mapeamento de modelos. 2005. 108 f. Dissertação (Mestrado em Ciências da Computação) - Universidade Federal de Pernanbuco, Pernanbuco, 2005.

FLANAGAN, David. **Java**: o guia essencial: [abrange Java 5.0]. 5. ed. Porto Alegre: Bookman, 2006. 1099 p.

FONSECA, Fabrício. **Ferramenta conversora de interfaces gráficas:** Delphi2Java-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, 2005.

GESSER, Carlos E. **GALS**: gerador de analisadores léxicos e sintáticos. [S.l.], 2003. Disponível em: <http://sourceforge.net/projects/gals>. Acesso em: 06 mar. 2008.

GONÇALVES, Edson. **Desenvolvendo aplicações Web com NetBeans IDE 5.5.** Rio de Janeiro: Ciência Moderna, 2007. 562 p.

HERRINGTON, Jack. **Code generation in action**. Greenwich, CT: Manning, c2003. xxvi, 342 p, il.

HOUAISS, A. B. H. **Dicionário Eletrônico Houaiss da Língua Portuguesa**. Versão 2.0. Rio de Janeiro: Objetiva, 2007. 1 CD-ROM.

JANTAL, Christofer. **Real-time UML to XMI conversion**. [S.l.], 2006. Disponível em: <http://http://www.nada.kth.se/utbildning/exjobb/rapportlistor/2006/rapporter06/>. Acesso em: 04 jun. 2008.

MARCOS, Adriana F. **Ferramente de apoio à automação de testes através do testcomplete para programas desenvolvidos em delphi.** 2007. 75 f. Trabalho de Conclusão de Curso (Bacharelado em Ciências da Computação) – Centro de Ciências Exatas e Naturais, Universidade Regional de Blumenau, Blumenau, 2007.

MATTOS, Mauro. M. ; MARTINS, Joyce. ; MEDEIROS, I. D. ; SILVEIRA, Janira. **Delphi2Java-II**: uma ferramenta para conversão de formulários Delphi em código Java. In: XV Seminário de Computação, 2006, Blumenau. Anais do XV SEMINCO - Seminário de Computação FURB, 2006.

NETBEANS. **Welcome to NetBeans**. [S.l.], 2008. Disponível em: <http://www.netbeans.org/>. Acesso em: 28 fev 2008.

NETCOOLE. **Migrates from Delphi to C#.** [S.l.], 2007. Disponível em: <http://www.netcoole.com/delphi2cs.html>. Acesso em: 12 nov. 2007.

RE-CODING. **Delphi2java::re-coding.** [S.l.], 2005. Disponível em: <http://www.re-coding.com//pricelist.php/>. Acesso em: 15 nov. 2007.

ROCHA, Helder L. S. da. **Padrões de projeto.** [S.l.], 2003. Disponível em: <http://www.argonavis.com.br/cursos/java/j930/J930_01.pdf >. Acesso em: 06 maio. 2008.

RUIZ, Evandro E. S. **GUI e componentes.** [S.l.], 2006. Disponível em: <http://dfm.ffclrp.usp.br/~evandro/ibm1030/swing/intro.html>. Acesso em: 25 out. 2007.

SCHVEPE, Claudio. **Gerador de código Java a partir de arquivos do Oracle Forms 6I.** 2006. 76 f. Trabalho de Conclusão de Curso (Bacharelado em Ciências da Computação) – Centro de Ciências Exatas e Naturais, Universidade Regional de Blumenau, Blumenau, 2006.

SEIBT, Patrícia Regina Ramos da Silva. **Ferramenta para cálculo de métricas em software orientado a objetos.** 2001. 85 f. Trabalho de Conclusão de Curso (Bacharelado em Ciências da Computação) – Centro de Ciências Exatas e Naturais, Universidade Regional de Blumenau, Blumenau, 2001.

SILVEIRA, Janira. **Extensão da ferramenta Delphi2Java-II para suportar componentes de banco de dados.** 2006. 75 f. Trabalho de Conclusão de Curso (Sistemas de Informação) – Centro de Ciências Exatas e Naturais, Universidade Regional de Blumenau, Blumenau, 2006.

SOARES FILHO, Alcides. **Migrar software reduz custos e economiza tempo.** [S.l.], 2003. Disponível em: <http://webinsider.uol.com.br/vernoticia.php/id/1844 >. Acesso em: 05 nov. 2007.

SOUZA, Ariana. **Ferramenta para conversão de formulários Delphi em páginas HTML.** 2005. 67 f. Trabalho de Conclusão de Curso (Bacharelado em Sistemas de Informação) – Centro de Ciências Exatas e Naturais, Universidade Regional de Blumenau, Blumenau, 2005.

WINSITE. **Winsite featured software**. [S.l.], 1997. Disponível em: <http://www.winsite.com/bin/Info?3253>. Acesso em: 15 mar. 2008.

APÊNDICE A – Formulário Delphi.

APÊNDICE B – Interface gráfica convertida pela ferramenta Delphi2Java-II

```
/*
Código gerado a partir do Delphi2Java-II
Projeto de pesquisa FURB/CNPq
Coordenador: Mauro Marcelo Mattos
Bolsista: Israel Damásio Medeiros
*/
package view;
import java.awt.Dimension;
import controller.uForm1Event;

public class uForm1FRM extends javax.swing.JFrame {
    protected javax.swing.JDesktopPane Form1;
    protected javax.swing.JLabel Label1 = new javax.swing.JLabel();
    protected javax.swing.JButton btnAtribuir= new javax.swing.JButton();
    protected javax.swing.JFormattedTextField editRecebe= new
javax.swing.JFormattedTextField();
    private uForm1Event classeEvento;
      public uForm1FRM() throws  Exception {
         classeEvento = new uForm1Event(this);
         initComponents();
         classeEvento.inicializar();
         setBounds(192, 107, 375, 110);
         setResizable(false);
    }
      private void initComponents() throws  Exception {
         setTitle("Formulário para Teste");
              addWindowListener(new java.awt.event.WindowAdapter() {
             public void windowActivated(java.awt.event.WindowEvent evt){
                 Form1WindowActivated(evt);
             }
             public void windowClosing(java.awt.event.WindowEvent evt){
                   exitForm1(evt);
             }
         });
         Form1  = new javax.swing.JDesktopPane();
         Form1.setBackground(new java.awt.Color(199,199,199));
         Form1.setVisible(true);
         getAccessibleContext().setAccessibleName("uForm1FRM Frame");
         getContentPane().add(Form1, java.awt.BorderLayout.CENTER); //adiciona o
Formulário ao JFrame.
         Form1.getAccessibleContext().setAccessibleName("Form1 Desktop");
         Form1.getAccessibleContext().setAccessibleDescription("Form1 desktop");
         Form1.add(Label1, javax.swing.JLayeredPane.DEFAULT_LAYER);  //adiciona o
Label ao componente Form1.
         Label1.setBackground(new java.awt.Color(199,199,199));
         Label1.setBounds(131, 24, 42, 13);
         Label1.setFont(new java.awt.Font("MS Sans Serif", 0,12));
         Label1.setForeground(new java.awt.Color(0, 0, 0));
         Label1.setText("Valor:");
         Label1.setOpaque(true);
         Label1.setVisible(true);
         Form1.add(btnAtribuir, javax.swing.JLayeredPane.DEFAULT_LAYER);
//adiciona o Botao ao componente Form1.
         btnAtribuir.setBackground(new java.awt.Color(199,199,199));
         btnAtribuir.setFont(new java.awt.Font("MS Sans Serif", 0,12));
         btnAtribuir.setForeground(new java.awt.Color(0, 0, 0));
         btnAtribuir.setText("Atribuir Valor");
         btnAtribuir.setFocusable(false);
         btnAtribuir.setVisible(true);
         btnAtribuir.setBounds(8, 24, 121, 41);
         Form1.add(editRecebe, javax.swing.JLayeredPane.DEFAULT_LAYER);  //adiciona
o Edit ao componente Form1.
         editRecebe.setBackground(new java.awt.Color(255,255,255));
         editRecebe.setBounds(136, 40, 225, 21);
         editRecebe.setVisible(true);
        editRecebe.setFont(new java.awt.Font("MS Sans Serif", 0,12));
         editRecebe.setForeground(new java.awt.Color(0, 0, 0));
         editRecebe.setText("");
}//fim da inicialização dos componentes.
      private void Form1WindowActivated(java.awt.event.WindowEvent evt) {
      }
```

```
        private void exitForm1(java.awt.event.WindowEvent evt) {
           System.exit(0);
     } //fim exitForm
        public javax.swing.JLabel getLabel1() {
              return Label1;
        }
        public void setLabel1(javax.swing.JLabel arg) {
              Label1 = arg;
        }
        public javax.swing.JButton getbtnAtribuir() {
              return btnAtribuir;
        }
        public void setbtnAtribuir(javax.swing.JButton arg) {
              btnAtribuir = arg;
        }
        public javax.swing.JFormattedTextField geteditRecebe() {
              return editRecebe;
        }
        public void seteditRecebe(javax.swing.JFormattedTextField arg) {
              editRecebe = arg;
        }
}
```

APÊNDICE C – Classe com os tratadores de eventos.

```
/*
Código gerado a partir do Delphi2Java-II
Projeto de pesquisa FURB/CNPq
Coordenador: Mauro Marcelo Mattos
Bolsista: Israel Damásio Medeiros
*/
package controller;
import view.uForm1FRM;
public class uForm1Event{
     private uForm1FRM screen;
     public  uForm1Event(uForm1FRM screen) {
          this.screen = screen;
     }
     public void inicializar() throws  Exception {
          CriarEventos();
     }
     private void CriarEventos() {
          screen.getbtnAtribuir().addActionListener(new
java.awt.event.ActionListener() {
                public void actionPerformed(java.awt.event.ActionEvent evt) {
                   btnAtribuirActionPerformed(evt);
               }
          });
 private void btnAtribuirActionPerformed(java.awt.event.ActionEvent evt){
    valor = "Valor fixo. Este foi definido no Evento Click.";
    editRecebe.text = valor;
}
     public static void main(String args[]) throws  Exception{
          new uForm1FRM().setVisible(true);
     }
}
```

APÊNDICE D – Classe `Util`.

```
/**
*Codigo gerado por: Delphi2Java-II - Conversor de regras de negocio V 0.1.4
*Data: Wed Jul 30 23:14:57 GMT-03:00 2008
**/
package uForm1;
 // *NP* import Windows;
 // *NP* import Messages;
 // *NP* import SysUtils;
 // *NP* import Variants;
 // *NP* import Classes;
 // *NP* import Graphics;
 // *NP* import Controls;
 // *NP* import Forms;
 // *NP* import Dialogs;
 // *NP* import StdCtrls;
//Classe principal, contem os procedimentos e metodos globais.
 public class uForm1_Util {
//lista de variaveis da INTERFACE
    private TForm1 form1;
//lista de variaveis da IMPLEMENTATION
//lista de Constantes da INTERFACE
//lista de Constantes da IMPLEMENTATION
//lista de METODOS da UNIT
}//class Util
```

APÊNDICE E – Arquivo texto com os itens identificados no arquivo Delphi.

```
=================================================
 ARQUIVO: uform1.pas
 Unit: uForm1
 Encontrados 10 USES para a INTERFACE deste arquivo:
 Encontrados 0 USES para a IMPLEMENTATION deste arquivo:
 Encontrados 1 VARIAVEIS para a INTERFACE deste arquivo:
 Encontrados 0 VARIAVEIS para a IMPLEMENTATION deste arquivo:
 Encontrados 0 CONSTANTES para a INTERFACE deste arquivo:
 Encontrados 0 CONSTANTES para a IMPLEMENTATION deste arquivo:
 Encontrado(s) 1 TYPES para a INTERFACE deste arquivo:
  TYPES
  Encontrados 1 CLASS para a TYPE deste arquivo:
   CLASS 'TForm1' do TYPE
   Encontrados 3 ATRIBUTOS para esta CLASS:
   Encontrados 0 PROPERTY para esta CLASS:
   Encontrados 1 METODOS para esta CLASS:
 Encontrados 0 RECORD para a TYPE deste arquivo:
 Encontrado(s) 0 TYPES para a IMPLEMENTATION deste arquivo:
 ------------------------------------------------
```