UNIVERSIDADE REGIONAL DE BLUMENAU

CENTRO DE CIÊNCIAS EXATAS E NATURAIS

CURSO DE SISTEMAS DE INFORMAÇÃO – BACHARELADO

EXTENSÃO DA FERRAMENTA DELPHI2JAVA-II PARA SUPORTAR COMPONENTES DE BANCO DE DADOS

JANIRA SILVEIRA

BLUMENAU
2006

2006/1-12

JANIRA SILVEIRA

EXTENSÃO DA FERRAMENTA DELPHI2JAVA-II PARA SUPORTAR COMPONENTES DE BANCO DE DADOS

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

Prof. Joyce Martins, Mestre - Orientadora

BLUMENAU
2006

2006/1-12

EXTENSÃO DA FERRAMENTA DELPHI2JAVA-II PARA SUPORTAR COMPONENTES DE BANCO DE DADOS

Por

JANIRA SILVEIRA

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

Presidente:	Profª. Joyce Martins, Mestre – Orientadora, FURB
Membro:	Prof. Mauro Marcelo Mattos, Doutor – FURB
Membro:	Prof. Alexander Roberto Valdameri, Mestre – FURB

Blumenau, 12 de julho de 2006

Dedico este trabalho a minha família, aos amigos, aos mestres dessa Universidade e a minha orientadora.

AGRADECIMENTOS

Agradeço a Deus, pela oportunidade de concluir mais uma etapa em minha vida.

Aos meus pais, meus irmãos, meu cunhado e meu namorado, por toda a paciência, atenção, força e confiança e por compreender a minha ausência durante o desenvolvimento deste trabalho.

Ao professor Mauro Marcelo Mattos, por ter acredito na idéia, no desenvolvimento e na finalização do trabalho.

A minha orientadora Joyce Martins, por ter acreditado em mim, pela sua total dedicação, amizade e por todo apoio e ajuda no desenvolvimento deste trabalho que resultaram na conclusão do mesmo.

RESUMO

O presente trabalho apresenta uma solução desenvolvida em forma de ferramenta, para realizar a conversão de formulários elaborados na linguagem de programação Delphi para aplicações na linguagem de programação Java. Entre os diversos tipos de componentes existentes em um formulário Delphi, são convertidos componentes de visualização, componentes de visualização de dados e componentes de acesso a banco de dados, sendo esses dois últimos tipos o foco deste trabalho. Para analisar o formulário, são utilizados analisadores léxico, sintático e semântico. Para gerar as classes Java de acordo com o padrão *Model-View-Controller*, é utilizado o motor de *templates* Velocity. A ferramenta é aplicada em um estudo de caso, onde através da API JDBC, os componentes fazem acesso ao banco de dados Oracle 8i.

Palavras-chave: Formulários Delphi. *Templates*. Padrão de projeto *Model-View-Controller* (MVC).

ABSTRACT

This work describes a solution developed as a tool to convert forms elaborated in Delphi programming language to applications in Java programming language. Among several types of components in a Delphi form there are being converted view components, data view components and data access components, being the last two types the focus of this work. To analyze the form there are used lexical, syntactic and semantic analyzers. To generate the Java classes following the Model–View–Controller (MVC) project pattern there are used Velocity templates engine. The tool is applied in a case study where through the JDBC API the components make access to a database in Oracle 8i.

Keywords: Delphi forms. Templates. Model-View-Controller (MVC) project pattern.

LISTA DE ILUSTRAÇÕES

LISTA DE SIGLAS

API – *Application Program Interface*

AWT – *Abstract Window Toolkit*

BNF – *Backus-Naur Form*

GUI – *Graphical User Interface*

HTML – *HyperText Markup Language*

JDBC – *Java Data Base Connectivity*

JDK – *Java Development Kit*

J2EE – *Java 2 Enterprise Edition*

MVC – *Model-View-Controller*

ODBC – *Open Database Connectivity*

RF – Requisito Funcional

RNF – Requisito Não Funcional

RTF – *Rich Text Format*

SGBD – Sistema Gerenciador de Banco de Dados

SQL – *Structured Query Language*

UML – *Unified Modeling Language*

URL – *Uniform Resource Locator*

VTL – *Velocity Template Language*

XML – *eXtensible Markup Language*

SUMÁRIO

1 INTRODUÇÃO

"Sistema de software é um artefato evolutivo e requer constantes modificações, seja para corrigir erros, melhorar desempenho, adicionar novas funcionalidades [...]" (PERES et al., p. 1, 2003). No entanto, segundo DMSNet (2004), para muitas empresas produtoras de software e sistemas corporativos, a grande limitação de seus produtos não se encontra na evolução das regras de negócio do mesmo, mas sim, na plataforma de desenvolvimento e no banco de dados para os quais o produto foi originalmente projetado. Tal limitação afeta seu posicionamento estratégico e mercadológico, pois inviabiliza a comercialização para clientes com outras plataformas ou sistemas operacionais. Mas "existem tantos sistemas, que a completa substituição ou a reestruturação radical é financeiramente impensável para a maioria das organizações" (SOMMERVILLE, 2003, p. 533). Para solucionar esse problema, foram desenvolvidas várias ferramentas visando apoiar a migração de aplicações para plataformas distintas (FONSECA, 2005; DMSNET, 2004; TAULLI; JUNG, 1997).

Atualmente o mercado está polarizado entre duas principais plataformas de desenvolvimento: Java 2 Enterprise Edition (J2EE) e Microsoft .NET. Segundo Gartner (2003 apud CESAR, 2003), juntas, essas tecnologias terão 80% ou mais do mercado de desenvolvimento de aplicações de *e-business* até 2008. Ainda, uma recente pesquisa, "que mede o percentual de adoção de tecnologias nas empresas de software, indica que Java cresceu de 72,2% em 2003 para 77,4% hoje. Ou seja, 77,4% das empresas de software usam Java. A pesquisa ainda mostra que 6,3% esperam usar Java até o próximo ano" (ZEICHICK, 2006).

Considerando esta demanda pela adoção da tecnologia Java, muitas empresas estão reestruturando seu processo de desenvolvimento ou até mesmo migrando os sistemas desenvolvidos. Para realizar a migração de aplicações Delphi para Java, por exemplo, além da

necessidade de conversão da interface gráfica, deve-se também converter o código-fonte para que seus componentes visuais mantenham a funcionalidade original, incluindo conexão a banco de dados.

Dentre as ferramentas que podem auxiliar no processo de migração de aplicações Delphi para a plataforma Java, cita-se Delphi2Java-II (FONSECA, 2005). A partir da análise de formulários Delphi contendo os componentes de interface da aplicação, são geradas classes Java que preservam o *layout* original. No entanto, segundo Fonseca (2005, p. 57), “uma limitação da ferramenta está no grupo de componentes selecionados para conversão, o qual não implementa todo o conjunto de componentes visuais existentes no ambiente Delphi”.

Diante do exposto, o presente trabalho visa estender a ferramenta Delphi2Java-II para permitir a conversão de componentes visuais para acesso a banco de dados bem como a conversão da funcionalidade da conexão ao banco, utilizando para isto a *Application Program Interface* (API) *Java Data Base Connectivity* (JDBC).

1.1 OBJETIVOS DO TRABALHO

O objetivo deste trabalho é ampliar a ferramenta Delphi2Java-II incorporando as funcionalidades de acesso a banco de dados.

Os objetivos específicos do trabalho são:

a) migrar a ferramenta Delphi2Java-II desenvolvida em Delphi para a linguagem Java;
b) permitir a conversão de alguns componentes de visualização de dados (`TDBCheckBox`, `TDBComboBox`, `TDBEdit`, `TDBGrid`, `TDBMemo`, `TDBRadioGroup`, `TDBText`), mantendo o *layout* original, e de componentes para manipulação e acesso a banco de dados (`TDatabase`, `TQuery`, `TTable`), implementando suas

principais funcionalidades;

c) agregar ao projeto Java a funcionalidade do componente `TDataSource`[1] do Delphi, utilizando a API JDBC;

d) analisar o grau de compatibilidade entre o código gerado pela ferramenta Delphi2Java-II e um estudo de caso desenvolvido em Delphi.

1.2 ESTRUTURA DO TRABALHO

O texto está estruturado em quatro capítulos. No segundo capítulo é apresentada a fundamentação teórica utilizada para o desenvolvimento do trabalho. Trata de migração de código e geradores de código, descreve a entrada da ferramenta (formulários Delphi), bem como apresenta as tecnologias utilizadas para gerar a saída. No terceiro capítulo é apresentado o desenvolvimento, incluindo a especificação dos requisitos e do caso de uso, a modelagem estrutural das classes, as ferramentas utilizadas no processo, a implementação e a operacionalidade de Delphi2Java-II. Por último, no capítulo quatro, são apresentadas as conclusões e sugestões de trabalhos futuros.

[1] Esse componente interliga os componentes de acesso a dados aos componentes de visualização de dados.

2 FUNDAMENTAÇÃO TEÓRICA

Neste capítulo são apresentados os aspectos teóricos relacionados à contextualização do trabalho. Engloba migração de código e geradores de código, descrevendo vantagens, desvantagens, aplicações e etapas de desenvolvimento. Apresenta a entrada da ferramenta desenvolvida, ou seja, formulários Delphi, mostrando como sua estrutura pode ser analisada através do uso de compiladores. Contempla também a saída gerada pela ferramenta, traçando um breve estudo da biblioteca Swing, da API JDBC e das classes `Bean`. Trata também do padrão *Model – View – Controller* (MVC) para padronização de projetos e do motor de *templates* Velocity. E por último relata as principais características dos trabalhos correlatos que constituem a base para a criação da ferramenta desenvolvida.

2.1 MIGRAÇÃO DE CÓDIGO

O desenvolvimento de software tem se tornado cada vez mais complexo. Conseqüentemente, uma das dificuldades enfrentadas pelas empresas de tecnologia é a manutenção e a migração desses sistemas para as novas plataformas de desenvolvimento adotadas.

Para minimizar custos na evolução de sistemas, usa-se a reengenharia de software através da tradução dos mesmos, quando um código-fonte em uma linguagem de programação é traduzido para um código-fonte em uma outra linguagem. A tradução do código-fonte só é economicamente viável se um tradutor automatizado estiver disponível para fazer a maior parte da tradução (SOMMERVILLE, 2003).

> A recente alternativa da migração de software traz uma visão completamente nova. O processo de conversão exige cerca de 25% do tempo de desenvolvimento do sistema original [...]. Na prática, migrar pode gerar uma economia de até 85% em relação aos investimentos em reescrever [...]. É por isso que a alternativa da migração – com as novas tecnologias de hoje – pode representar economias de

> tempo e dinheiro tão significativas [...] (SOARES FILHO, 2003).

Portanto, diante das dificuldades e dos custos para realizar a evolução dos sistemas existentes, a migração desses sistemas para linguagens mais atuais, adotando técnicas ou ferramentas que possam facilitar essa conversão, reduz custos, tempo e trabalho. Com o auxílio de novos padrões de desenvolvimento, como ferramentas automatizadas e técnicas de conversão, os sistemas tornam-se mais simples de serem interpretados. A migração visa facilitar futuras manutenções, aumentando a qualidade dos sistemas e prolongando sua vida útil. Assim, como recurso para a migração de software, encontram-se os geradores de código.

2.2 GERADORES DE CÓDIGO

A geração de código é uma técnica de construção de código que utiliza determinadas ferramentas para gerar programas. Estas ferramentas podem variar de *scripts* de ajuda muito pequenos a grandes criações que transformam modelos abstratos de lógica de negócio em aplicações completas. Não há nenhum estilo específico para as ferramentas de geração de código (DALGARNO, 2006): podem trabalhar na linha de comando ou possuir uma *Graphical User Interface* (GUI); podem gerar código para uma ou mais linguagens; podem gerar código uma vez ou múltiplas vezes; podem ter um número ilimitado de entradas e saídas.

Segundo Dalgarno (2006), as principais vantagens das técnicas de geração de código são:

a) qualidade: a qualidade do código construído por um gerador está relacionada diretamente à qualidade do código ou dos *templates*[2] usados. Quando a qualidade desse código ou dos *templates* é aumentada, a qualidade do código gerado também

[2] É um modelo que serve como guia para a construção de código. No *template* podem-se definir trechos estáticos ou dinâmicos, que serão utilizados para gerar a saída desejada.

é aumentada;

b) consistência: o código criado por geradores de código é extremamente consistente. O nome das variáveis, dos métodos e das classes, por exemplo, é padronizado em todo o código gerado. Isso faz com que o código construído seja fácil de ser compreendido e, conseqüentemente, de ser usado. Além disso, é mais fácil de agregar funcionalidades a um código consistente;

c) abstração: alguns geradores constroem código baseados em modelos abstratos do código alvo. Pode-se, por exemplo, construir um SQL Schema[3] e uma camada de acesso a base de dados, a partir de uma definição em *eXtensible Markup Language* (XML) das tabelas, dos campos, dos relacionamentos e das consultas a uma base de dados. O valor desta abstração é que o gerador pode ser redirecionado para construir o código para outra plataforma. Com isto obtém-se portabilidade entre plataformas;

d) produtividade: é fácil reconhecer o benefício inicial de um gerador de código. Na primeira vez que o gerador é executado, a partir de uma entrada contendo os modelos abstratos, quase instantaneamente se obtém o código de saída que implementa a especificação contida nesses modelos. Entretanto, o ganho real de produtividade começa quando o gerador é executado várias vezes para gerar novos códigos baseados em mudanças na especificação definida.

Moreira e Mrack (2003) apontam algumas desvantagens das ferramentas de geração, entre elas tem-se:

a) o código é gerado em um único sentido, ou seja, caso o código gerado seja modificado pelo programador, o modelo abstrato a partir do qual o mesmo foi gerado não será alterado. Uma vez que o modelo abstrato seja alterado, a geração

[3] Um SQL Schema nada mais é que a soma de todas as tabelas, definições e relacionamentos.

deverá ser reexecutada e as alterações efetuadas anteriormente pelo programador serão perdidas e deverão ser refeitas;

b) o código gerado tem padrão dependente da ferramenta usada e geralmente não está de acordo com o padrão de codificação da equipe de desenvolvimento;

c) o código gerado não leva em consideração questões de desempenho, otimização, estrutura, documentação ou integração com outros sistemas.

Existem geradores para uma ampla variedade de aplicações. São muito comuns os geradores de interface com o usuário e os de acesso a base de dados. A codificação da camada de interface com o usuário é razoavelmente repetitiva, então é comum a utilização de geradores para simplificar esta codificação e torná-la um trabalho menos manual. Além de reduzir erros na escrita, um gerador de interface pode também fornecer uma implementação para múltiplas linguagens ou plataformas (DALGARNO, 2006).

Em se tratando de geradores de acesso a base de dados, pode-se dizer que estes são o tipo mais popular de geradores de código, uma vez que o código de acesso a base de dados também é muito repetitivo e propenso a erros de escrita. Além do mais, gerar a camada de acesso a base de dados fornece benefícios de portabilidade inerentes à plataforma. De acordo com Herrington (2003), alguns benefícios obtidos com o uso de geradores de acesso a base de dados são:

a) ganho na produtividade, pois a partir de um modelo abstrato contento a especificação de rotinas, como inclusão, exclusão e alteração, obtém-se com pouco esforço a implementação das mesmas;

b) abstração do projeto da base de dados do código que implementa o sistema, permitindo que o modelo abstrato possa ser facilmente revisto e validado;

c) projeto e interface consistentes da base de dados e do código de acesso, facilitando gerar as camadas que estão acima da base de dados.

Dalgarno (2006) e Herrington (2003) citam outras aplicações que podem ser desenvolvidas usando geradores de código, entre as quais: geradores de documentação, como o JavaDoc, que geram arquivos *HyperText Markup Language* (HTML) a partir de comentários presentes no código-fonte; geradores de *scripts*, como o Hibernate, que a partir de um mapeamento das tabelas do banco de dados geram *scripts* em *Structured Query Language* (SQL); os Wizards que, a partir de comentários embutidos no código-fonte, adicionam código implementando os requisitos especificados; geradores de código, como a ferramenta Enterprise Architect, que a partir de um modelo abstrato de classes geram classes Java.

Para a construção de um gerador de código podem ser seguidas as seguintes etapas de desenvolvimento (HERRINGTON, 2003):

a) identificar o que se deseja obter como saída do gerador, como no caso da ferramenta Delphi2Java-II (FONSECA, 2005), em que a saída são interfaces desenvolvidas em Swing e a assinatura dos métodos dos eventos associados aos componentes gráficos;
b) definir qual será a entrada e como a mesma será analisada, como no caso da ferramenta Delphi2Java-II, onde a análise do arquivo de entrada, um formulário Delphi, é feita através da identificação de cada símbolo que compõe o arquivo e da identificação da ordem em que esses símbolos se encontram;
c) interpretar e recuperar as informações da entrada, definindo a formatação e a geração da saída, como no caso da ferramenta Delphi2Java-II, onde o código a ser gerado encontra-se “embutido” na ferramenta;
d) gerar a saída a partir das informações extraídas do arquivo de entrada, que no caso da ferramenta Delphi2Java-II são as classes Java.

2.3 FORMULÁRIOS DELPHI

Para cada formulário de uma aplicação desenvolvida em Delphi é gerado um arquivo com a extensão DFM, contendo todas as informações dos componentes da interface. Cada componente tem uma identificação, uma lista de propriedades com valores associados e outros possíveis objetos componentes (SASSE, 2005). O quadro 1 mostra a estrutura de um arquivo .DFM, com um objeto `TForm` e seus respectivos objetos componentes, juntamente com suas propriedades. A figura 1 mostra a interface correspondente ao arquivo .DFM.

```
object FormBD: TFormBD
  Left = 192
  Top = 107
  Width = 553
  Height = 645
  ...
  object DBComboBoxCidade: TDBComboBox
    Left = 85
    Top = 264
    Width = 145
    Height = 21
    DataField = 'NOME'
    DataSource = DataSourceQueryCidade
    ItemHeight = 13
    TabOrder = 2
  end
  object TableTBAluno: TTable
    DatabaseName = 'BDoracle'
    TableName = 'TBALUNO'
    object TableTBAlunoMATRICULA: TStringField
      FieldName = 'MATRICULA'
      Size = 255
    end
    object TableTBAlunoNOME: TStringField
      FieldName = 'NOME'
      Size = 255
    end
    object TableTBAlunoDESCRICAO: TMemoField
      FieldName = 'DESCRICAO'
      BlobType = ftMemo
      Size = 300
    end
    ...
  end
  ...
end
```

identificação do componente

lista de propriedades

objeto componente

Quadro 1 – Estrutura de um arquivo .DFM

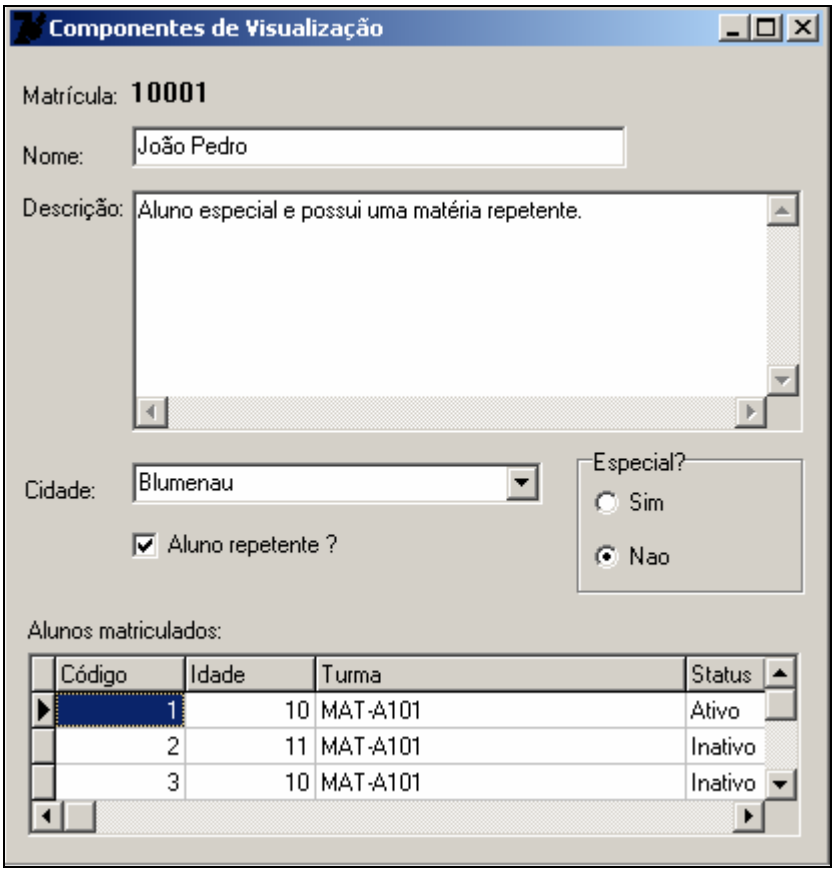


Figura 1 – Interface

Para análise de arquivos .DFM pode ser utilizado um tradutor de linguagem de programação, ou seja, um compilador. De acordo com Price e Toscani (2001), compiladores são programas bastante complexos que convertem linguagens de fácil escrita e leitura (linguagem de alto nível), em linguagens que possam ser interpretadas ou executadas pelas máquinas.

O processo de compilação é composto de análise e síntese. A análise tem como objetivo entender o código-fonte e representá-lo em uma estrutura intermediária. A síntese constrói o código-objeto a partir desta representação intermediária. A análise pode ser subdividida em (COMPILADOR, 2006):

a) análise léxica: possui como principal objetivo ler o texto de entrada, identificando unidades léxicas (*tokens*), tais como palavras reservadas, identificadores e símbolos especiais. Elimina comentários e caracteres indesejáveis, enviando mensagens de erro caso sejam encontradas seqüências de caracteres não aceitas

pela linguagem em questão;

b) análise sintática: tem por função verificar se a estrutura gramatical do programa está correta, identificando as seqüências dos *tokens* que constituem as estruturas sintáticas de comandos e expressões (PRICE; TOSCANI, 2001). Também envia mensagens de erro caso sejam encontradas seqüências de *tokens* que não correspondam à estrutura sintática da linguagem em questão;

c) análise semântica: é responsável por verificar se as estruturas do programa irão fazer sentido durante a execução, extraindo informações do programa que permitam a geração do código-objeto.

2.4 INTERFACE GRÁFICA EM JAVA

Em Java, existem dois grandes pacotes mais utilizados para a construção de interfaces gráficas, o *Abstract Window Toolkit* (AWT) e o Swing. Segundo Trindade (2002), “o pacote AWT foi o primeiro pacote de classes para a construção de interfaces com o usuário. Ele faz chamadas para o sistema operacional para exibir os componentes da interface”. Assim, os componentes são exibidos com uma aparência diferente em cada plataforma. Já o Swing, que é uma extensão do AWT, possui algumas melhorias, como estrutura mais leve para a criação dos componentes e total independência do sistema operacional, possibilitando a especificação de uma aparência uniforme para todas as plataformas. Coelho (2004) aponta algumas diferenças entre Swing e AWT, entre elas pode-se citar:

a) os componentes da biblioteca Swing são desenvolvidos totalmente em Java, ao contrário dos componentes AWT, que operam tendo por base as funcionalidades do gerenciador de janelas e bibliotecas nativas do sistema operacional onde a aplicação é executada;

b) os componentes da biblioteca Swing podem ser facilmente alterados, por meio de chamadas a métodos ou criação de sub-classes;

c) os componentes da biblioteca Swing podem possuir várias formas geométricas, assim como suas cores, que podem ser alteradas. Possuem também o recurso de especificar o *look and feel* (aparência) da GUI programada. Em outras palavras, a aparência do ambiente de janelas é configurável.

Os componentes da biblioteca Swing ao sofrerem algum tipo de interação com o usuário, como por exemplo, um *click* de *mouse* ou o preenchimento de um campo de texto, geram ações conhecidas como eventos. De acordo com Coelho (2004, p. 148), “pode-se dizer que evento é um objeto que descreve uma mudança de estado na fonte do evento [ou seja, no componente], podendo ser gerado como conseqüência da interação do usuário com um elemento de interface gráfica”.

Para tratar os eventos disparados pela interface podem ser implementadas classes intermediárias, que capturam estes eventos e os tratam de forma adequada delegando a execução à camada de negócio, se necessário. Citam-se como exemplo alguns componentes de interface que geram eventos que realizam acesso a base de dados. Neste caso, a integração com o banco de dados ocorre através das classes disponibilizadas pela API JDBC. Desta forma, pode-se ter a implementação da interface gráfica de um sistema totalmente independente do banco de dados e da camada de negócio.

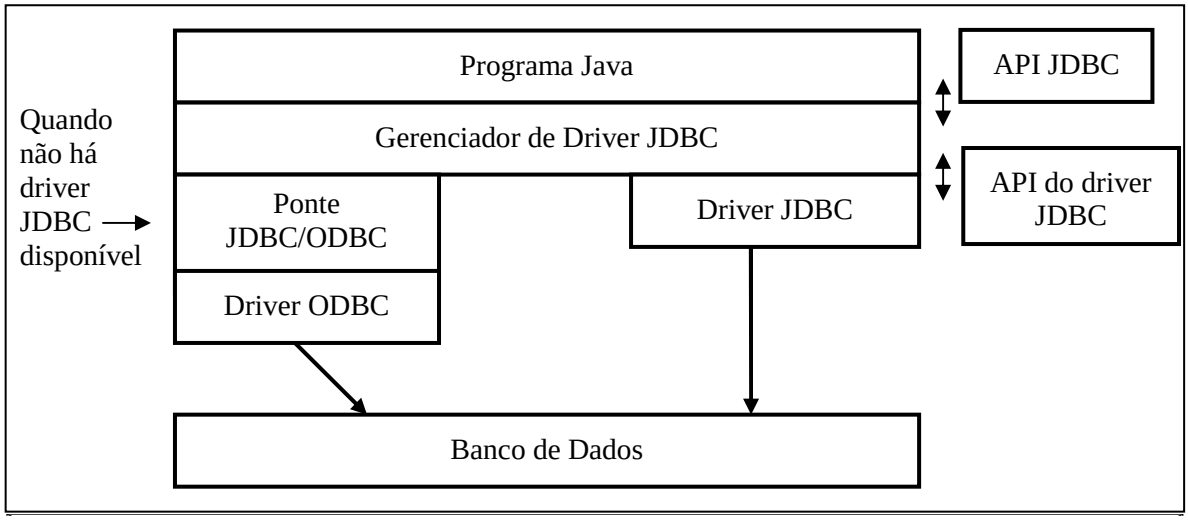
2.5 ACESSO A BANCO DE DADOS EM JAVA

Conforme Jeveaux (2004), “algumas linguagens são desenvolvidas para terem um alto desempenho, porém se tornam complexas para trabalhar com banco de dados”. Com o Java o programador possui disponível uma tecnologia para acesso a banco de dados independente de

plataforma e de fornecedor.

Devido a enorme quantidade de bancos de dados com linguagens proprietárias, os projetistas do JavaSoft[4] resolveram criar uma nova API para banco de dados. Ou seja, uma camada entre a linguagem Java e outra linguagem que todos os bancos de dados suportam. A linguagem de banco de dados escolhida foi SQL. Assim, os fabricantes de banco de dados fornecem os *drivers* para SQL e o Java fornece a API JDBC (JEVEAUX, 2004).

O funcionamento da tecnologia JDBC é ilustrado na figura 2. Os programas Java tentam conseguir acesso a um banco de dados, através de uma API JDBC, executando consultas SQL padrão. Essas consultas são enviadas ao gerenciador de *driver* JDBC. Caso exista um *driver* JDBC para o banco de dados desejado, o gerenciador passa para o mesmo as consultas SQL através da API do *driver* JDBC. Se não existir um *driver* JDBC disponível, então o gerenciador utiliza uma ponte JDBC / *Open Database Connectivity* (ODBC), que envia as consultas SQL para o *driver* ODBC (interface de programação C para SQL) (JEVEAUX,2004).



Fonte: Jevaux (2004).

Figura 2 – Tecnologia JDBC

Os *drivers* são o centro do JDBC. Há quatro tipos de *drivers*, cada um baseado numa tecnologia ou arquitetura, que fornecem características próprias para o desenvolvimento de

[4] Divisão da Sun que mantém Java desde a versão 1.0 *Java Development Kit* (JDK) (SUN MICROSYSTEMS, 2006).

um determinado tipo de aplicação. Os principais tipos de *drivers* JDBC disponíveis (BALES, 2002) são:

a) *driver* ponte JDBC/ODBC: efetua a conversão de chamadas JDBC em ODBC e as encaminha para o *driver* ODBC. É utilizado em situações para as quais não exista outro *driver* disponível;
b) *driver* de acesso nativo: converte chamadas JDBC em chamadas nativas de banco e comunica-se diretamente com o Sistema Gerenciador de Banco de Dados (SGBD);
c) *driver* de acesso por *middleware*: converte chamadas JDBC em um protocolo de rede independente do SGBD e comunica-se com um *gateway* que traduz estas requisições para o protocolo específico do SGBD. É a mais flexível alternativa de JDBC, pois a camada servidora do *driver* pode ser implementada para acessar diversos SGBDs, simplificando a migração entre bancos;
d) *driver* de acesso direto ao servidor: converte chamadas JDBC para um protocolo de rede usado diretamente pelo SGBD.

Monteiro (2005) afirma que são necessários alguns passos básicos para a utilização do JDBC, tais como: carregar e registrar o *driver* JDBC junto ao gerenciador de *driver*; configurar e obter uma conexão com o banco de dados; preparar a consulta; executar a consulta; obter e verificar os resultados; tratar possíveis erros; formatar a saída para o usuário; e fechar a conexão, ao finalizar a aplicação. Para tanto, a API JDBC disponibiliza classes Java para acesso e manipulação do banco de dados, entre as quais tem-se:

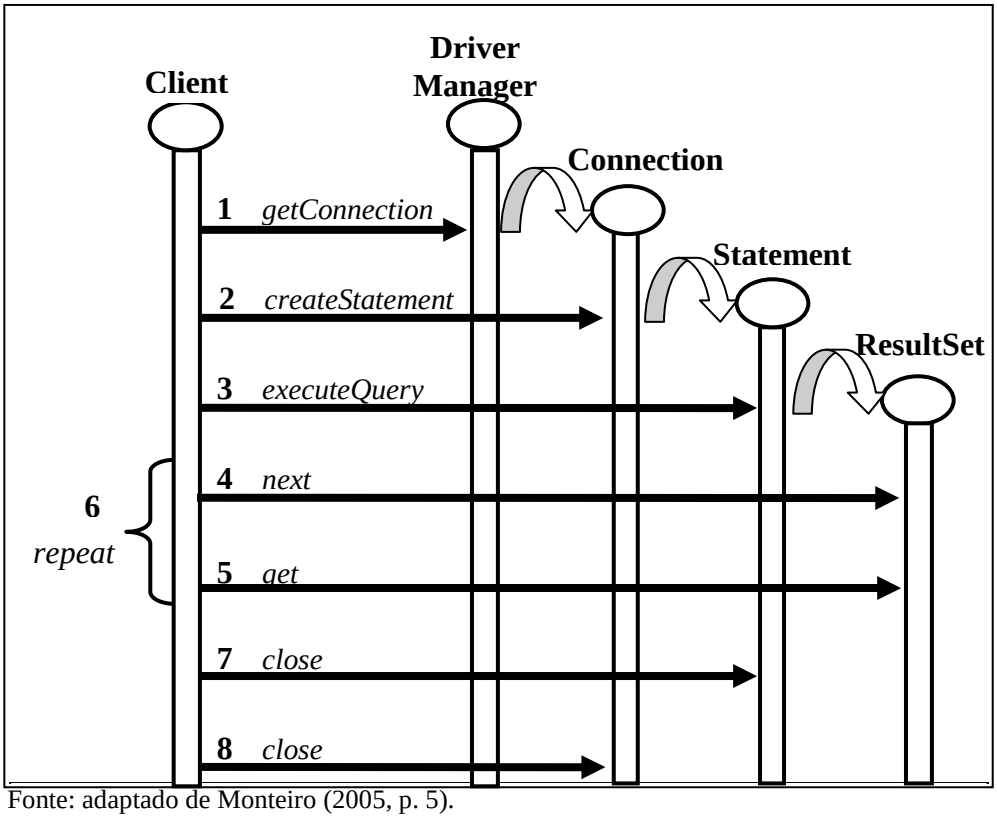
a) `java.sql.Connection`: sua função é fazer a conexão lógica com o banco de dados, possibilitando enviar consultas SQL ao banco de dados, bem como controlar a abertura ou fechamento de conexões existentes;
b) `java.sql.DriverManager`: é responsável por selecionar o *driver* apropriado para

a conexão ao banco de dados através da *Uniform Resource Locator* (URL) que lhe for passada como parâmetro;

c) `java.sql.PreparedStatement`: utilizada para executar consultas SQL que se repetem várias vezes, podendo também executar consultas parametrizadas. As instâncias de `PreparedStatement` contêm consultas SQL já compiladas;

d) `java.sql.ResultSet`: é responsável por armazenar as consultas feitas ao banco de dados.

A figura 3 mostra o cenário de uma aplicação JDBC: no passo 1 é realizada uma chamada para carregar e registrar um *driver* JDBC; no passo 2, é obtida uma conexão com o banco de dados e criada uma consulta SQL; no passo 3 a consulta SQL é executada, retornando os dados como resultado; nos passos 4 e 5 a aplicação percorre a tabela retornando os dados de cada linha, através da iteração do passo 6; no passo 7 é encerrada a consulta e, por último, no passo 8 é fechada a conexão com o banco.



Fonte: adaptado de Monteiro (2005, p. 5).

Figura 3 – Cenário de uma aplicação JDBC

2.6 CLASSES BEAN

Conforme apresentado em Fundão da Computação (2004), uma classe `Bean` é uma classe Java muito simples que funciona agrupando uma coleção de propriedades cujos valores são definidos através de métodos do tipo `set` ou recuperados através de métodos do tipo `get`. Uma classe `Bean` pode representar, por exemplo, uma tabela, onde cada atributo corresponde a uma coluna da tabela. Sendo assim, através dessa classe os componentes da API Swing podem, em conjunto com a API JDBC, definir os valores de suas propriedades a partir da base de dados e acessá-los para, por exemplo, realizar consultas que são utilizadas para o preenchimento de uma tela. No quadro 2 é ilustrado um exemplo de uma classe `Bean` da tabela `TBALUNO`, definida a partir da propriedade `TableName` do componente `TableTBAluno` do formulário representado no quadro 1.

```
public class BeanTBALUNO extends ConnectionManager{

  private Connection conexao;

  //-- Campos da tabela TBALUNO
  private String  MATRICULA;
  private String  NOME;
  ...
  //-- Construtor
  public BeanTBALUNO(Connection c) throws Exception{
      conexao = new ConnectionManager().retornaConexao();
  }

  //-- Sets e Gets dos campos
  public String getMATRICULA(){
      return MATRICULA;
  }
  public void setMATRICULA(String MATRICULA) {
      this.MATRICULA = MATRICULA;
  }

  public String getNOME(){
      return NOME;
  }
  public void setNOME(String NOME) {
      this.NOME = NOME;
  }
  ...
}
```

Quadro 2 – Classe `Bean`

2.7 PADRÃO MVC

Desenvolver aplicações de qualidade é uma atividade que requer um bom planejamento, mantendo as partes do sistema bem definidas, mapeando suas dependências e mantendo os seus módulos genéricos.

> Diariamente desenvolvedores, arquitetos, gerentes de projeto e usuários são forçados a lidar com estruturas de dados complexas, alterações em regras de negócio, mudanças de necessidades dos usuários e novas tecnologias. Naturalmente, os desenvolvedores têm menos tempo e menos recursos do que precisariam (ou gostariam) para enfrentar tudo isso. (FUNDÃO DA COMPUTAÇÃO, 2004).

Para auxiliar no desenvolvimento de sistemas, existem técnicas que compõem padrões para organização dos projetos e que sugerem soluções para problemas que ocorrem em diversos cenários de desenvolvimento, sendo essas técnicas conhecidas como *design patterns* (padrões de projeto). A utilização de padrões de projeto auxilia na construção de sistemas confiáveis, promovendo a reutilização em sistemas novos, auxiliando também na identificação de problemas comuns que ocorrem quando se constroem sistemas e projetando sistemas de tal forma que fiquem independentes da plataforma na qual serão implementados (DEITEL; DEITEL, 2003).

Uma solução dentre os padrões de projeto é o padrão MVC que tem por objetivo dividir os elementos de um sistema em três tipos de camadas: modelo (*Model*), visão (*View*) e controlador (*Controller*) (GAMMA et al., 1994).

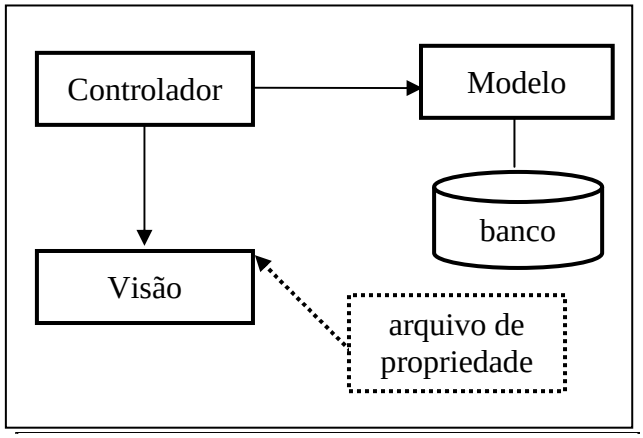
> No final dos anos 70, quando as interfaces gráficas do usuário (GUIs) estavam sendo inventadas, arquitetos de software viam as aplicações como tendo três partes maiores: a parte que gerencia os dados, a parte que cria as telas e os relatórios e a parte que lida com as interações entre o usuário e os outros sub-sistemas [...]. No início dos anos 80, o ambiente de programação ObjectWorks/Smalltalk introduziu esse grupo de três como uma estrutura de desenvolvimento. Na linguagem Smalltalk 80, o sistema de dados é denominado Model (Modelo), o sistema de apresentação é chamado de View (Exibição) e o sistema de interação é o Controller (Controlador). Muitos ambientes de desenvolvimento modernos, inclusive o Swing do Java, usam essa arquitetura Model/View/Controller (MVC). (HUSTED et al., 2004, p. 28-29).

Segundo Bodoff et al. (2002), o padrão MVC reforça um projeto modular e de fácil manutenção e força a separação de camadas. As visões representam a interface com o usuário

que em uma aplicação padrão *desktop*, consiste de telas utilizando componentes da biblioteca Swing, por exemplo. O controlador pode ser implementado com uma ou mais classes Java e o modelo de dados fornece código com acesso direto ao banco de dados.

> O MVC é útil principalmente para aplicações grandes e distribuídas onde dados idênticos são visualizados e manipulados de formas variadas. Como o MVC facilita a divisão de trabalho por conjuntos de habilidades, este pattern é bastante adequado para empresas de desenvolvimento que suportam desenvolvimento modular e concorrente com muitos desenvolvedores (CADE; ROBERTS, 2002, p. 44).

Promovendo a portabilidade de interfaces, o padrão MVC também torna fácil testar e manter suas aplicações. Kassem (2000) afirma que a chave para o MVC é a delegação de responsabilidades. As visões podem usar o modelo de dados para exibir resultados, mas elas não são responsáveis por atualizar o banco de dados. Os controladores são responsáveis por fazer alterações no modelo de dados. O modelo, por sua vez, é responsável por representar os dados da aplicação. Algumas vezes o modelo de dados também inclui a lógica de negócio e algumas vezes a lógica de negócio existe na camada do controlador. A figura 4 ilustra o padrão MVC.



Fonte: adaptado de Hansen (2006).

Figura 4 – Padrão MVC

O padrão MVC oferece algumas características que fazem dele uma boa escolha de padrão a ser utilizado no desenvolvimento de software, dentre as quais, tem-se:

a) separar dados (*Model*) da interface do usuário (*View*) e do fluxo da aplicação (*Controller*);

b) permitir que uma mesma lógica de negócio possa ser acessada e visualizada

através de várias interfaces;

c) a lógica de negócio é independente da camada de interface com o usuário (*View*).

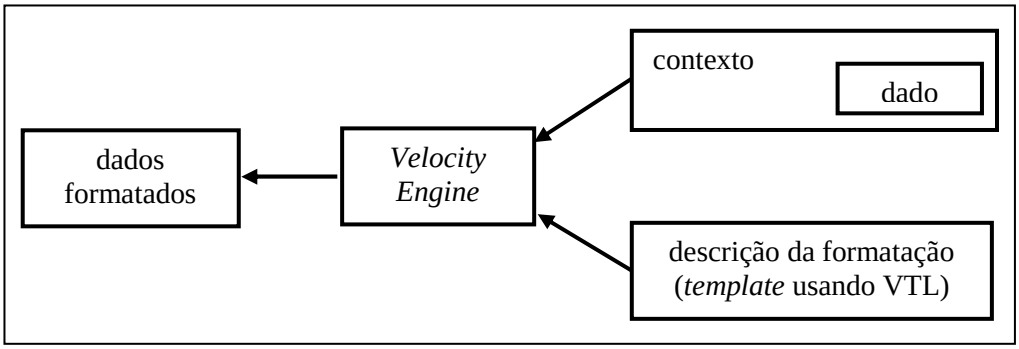
2.8 MOTOR DE TEMPLATES VELOCITY

Gerar código significa construir complexos arquivos textos estruturados. Para manter a integridade e a simplicidade do gerador, deve-se usar *templates*. Um *template* pode possuir, além de um conteúdo estático, um código dinâmico composto por variáveis e comandos estruturados que serão substituídos quando do seu processamento. Assim, com o uso de *templates* pode-se manter a formatação do código separada da lógica que determina o que deve ser construído. Essa separação entre a lógica do código a ser gerado e a formatação do código é a representação de uma abstração ideal (HERRINGTON, 2003). Conforme Moura e Cruz (2002), existem atualmente vários motores de *templates* que auxiliam de maneira rápida e fácil a construção de aplicações, como por exemplo Freemaker, WebMacro e Velocity.

Para implementar a extensão de Delphi2Java-II, conforme proposto nesse trabalho, optou-se pela utilização do Velocity que, segundo Steil (2006), é uma ferramenta *open source*, subprojeto do projeto Jakarta, da Apache Foundation. Uma de suas maiores utilidades é no desenvolvimento de aplicações cujo código da lógica de negócio fica separado do *layout*, tornando assim a aplicação muito mais modularizada e fácil de manter. Pode ser usado para: formatação de mensagens, criação de documentos *Rich Text Format* (RTF), geração de código SQL, entre outras aplicações.

Velocity é ideal para qualquer tipo de programa Java que necessite de formatação e apresentação de dados. Moura e Cruz (2002) afirmam que as funcionalidades dessa ferramenta devem ser acessadas através da API Velocity. Assim, deve-se: criar uma instância de `VelocityEngine`; criar um esquema de mapeamento (contexto) entre os objetos Java que

contêm os dados da aplicação e os elementos dos conteúdos dinâmicos definidos no *template*; carregar o *template* desejado; substituir adequadamente as referências definidas no *template* pelos valores dos objetos Java de acordo com o contexto; gerar o arquivo de saída. A arquitetura da ferramenta está esquematizada na figura 5.



Fonte: Moura e Cruz (2002).

Figura 5 – Arquitetura da ferramenta Velocity

A formatação deve ser descrita utilizando a *Velocity Template Language* (VTL). A VTL possui uma sintaxe simples e fácil de ser manipulada e, com o auxílio de *plugins*, a editoração desse tipo de *template* torna-se mais rápida. A VTL possui um número limitado de elementos para a especificação do código dinâmico. No quadro 3 é mostrada a estrutura de um *template* e no quadro 4 são descritos os elementos mais usados.

```
#*Aqui encontra-se a estrutura dos componentes DB.*#
...
#foreach ($TDBComboBox in $arrayDBComboBox)  #*-Componente DBComboBox--*#
  ...
  $!{TDBComboBox.getName()}.setBackground
               (new java.awt.Color ( $!{TDBComboBox.setarBackground()} ));
  $!{TDBComboBox.getName()}.setBounds( $!{TDBComboBox.getLeft()},
                                       $!{TDBComboBox.getTop()},
                                       $!{TDBComboBox.getWidth()},
                                       $!{TDBComboBox.getHeight()} );
  $!{TDBComboBox.getName()}.setVisible($!{TDBComboBox.getVisible()});
  #if ($!{TDBComboBox.getFontJava()})
    $!{TDBComboBox.getName()}.setFont
               (new java.awt.Font( $!{TDBComboBox.getFontJava()} ));

    $!{TDBComboBox.getName()}.setForeground
               (new java.awt.Color( $!{TDBComboBox.getForeground()} ));
  #end
  #if ($!{TDBComboBox.getText()})
    $!{TDBComboBox.getName()}.setText( "$!{TDBComboBox.getText()}" );
  #end
  #if ($!{TDBComboBox.getEnabled()})
    $!{TDBComboBox.getName()}.setEnabled( $!{TDBComboBox.getEnabled()} );
  #end
#end
...
```

Quadro 3 – Exemplo de *template* com definição do componente `TDBComboBox`

ELEMENTO	ESPECIFICAÇÃO	EXEMPLO
identificador	qualquer seqüência de letras, dígitos, hífen ou *undeline*, que deve começar com uma letra	`TDBComboBox` `arrayDBComboBox`
declaração de variáveis	identificador precedido por $ (cifrão) ou identificador precedido $ (cifrão) e por ! (exclamação) ou ainda identificador entre chaves precedido por $ (cifrão)	`$TDBComboBox`
método	cifrão ($) seguido por ! (exclamação) e, entre chaves, identificador seguido por ponto (.), pelo nome do método, por abre e fecha parênteses, sendo que entre os parênteses pode existir uma lista de argumentos separados por vírgula	`$!{TDBComboBox.getName()}`
atribuição	`#set`	`#`**`set`**`($flagEstado = "true")`

controle de fluxo de execução	`#if-#elseif-#else-#end`: comando condicional
	`#foreach-#end`: comando de repetição
		`#`**`if`**`($!TDBComboBox.getText())`
		`  ...`
		`#`**`end`**
		
		`#`**`foreach`**
		`($TDBComboBox in`
		`$arrayDBComboBox)`
		`...`
		`#`**`end`**

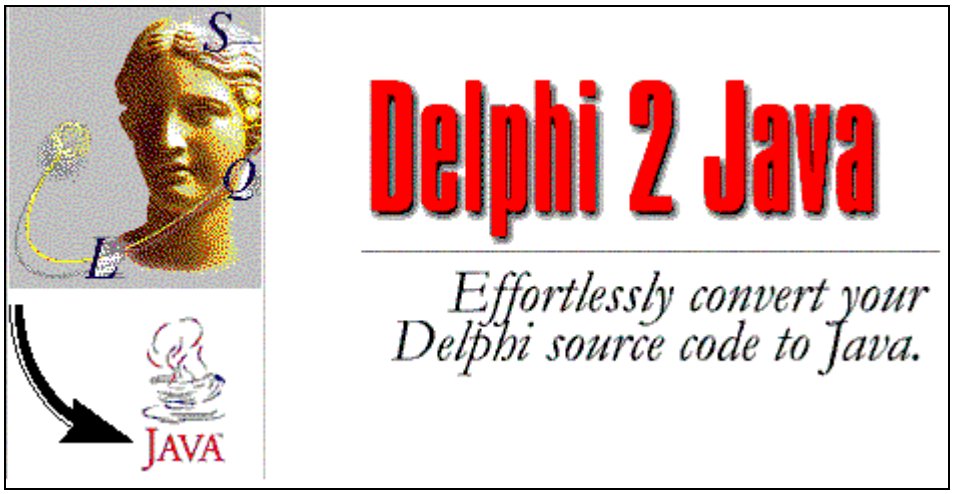
inclusão de arquivos	`#include`: permite incluir arquivos que não serão analisados pelo Velocity
	`#parse`: permite incluir *templates* que serão analisados pelo Velocity
		`#include("texto.txt")`
		
		`#parse("templateAuxiliar.vm")`

comentários	podem ser de linha, quando iniciados com ##, ou de bloco, quando iniciados com `#*` e finalizados com `*#`	`## comentário em uma linha`
		`#*`
		`  comentário em várias linhas`
		`*#`

Quadro 4 – Elementos da linguagem VTL

2.9 TRABALHOS CORRELATOS

Durante o processo de levantamento bibliográfico para a fundamentação deste trabalho, foram encontradas diversas referências na Internet para um projeto denominado Delphi2Java (ROBINSON, 2006). Trata-se de uma ferramenta comercial que se propõe a converter interfaces gráficas, componentes com conexão ao banco de dados e código-fonte de aplicações Delphi. A tela da ferramenta é apresentada na figura 6.



Fonte: Robinson (2006).

Figura 6 – Ferramenta Delphi2Java

Este projeto inspirou o desenvolvimento da ferramenta Delphi2Java-II (FONSECA, 2005), que lê o conteúdo dos arquivos .DFM e produz classes Java que apresentam o mesmo *layout* dos formulários desenvolvidos em Delphi. A ferramenta converte apenas os componentes visuais mais usuais e que possuem equivalência em Java, excetuando-se os componentes para visualização de dados e para conexão e acesso a banco de dados. Além de gerar uma classe com componentes gráficos Swing, Delphi2Java-II disponibiliza também uma classe de eventos, que contém apenas as assinaturas dos métodos que estão habilitados na aplicação original em Delphi, permitindo que o código seja futuramente inserido. A figura 7 apresenta a tela principal da ferramenta Delphi2Java-II e a figura 8 apresenta a interface gerada pela ferramenta, utilizando como entrada um arquivo .DFM similar ao quadro 1 (seção 2.3), sem os componentes de acesso a banco de dados.

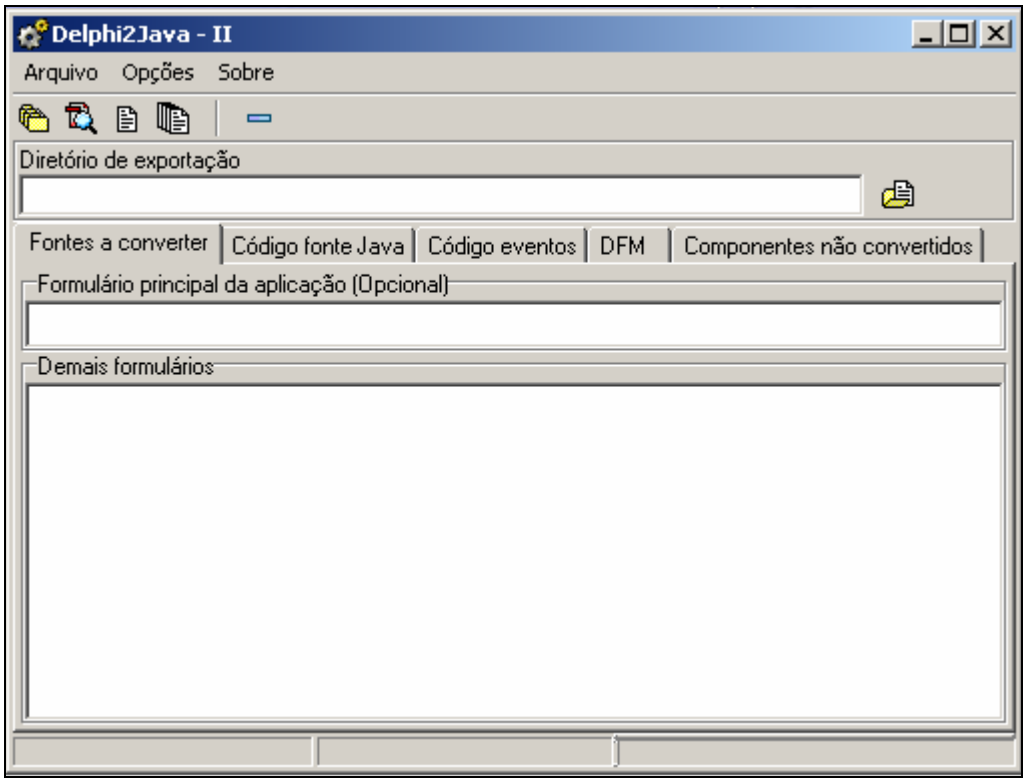


Figura 7 – Ferramenta Delphi2Java-II

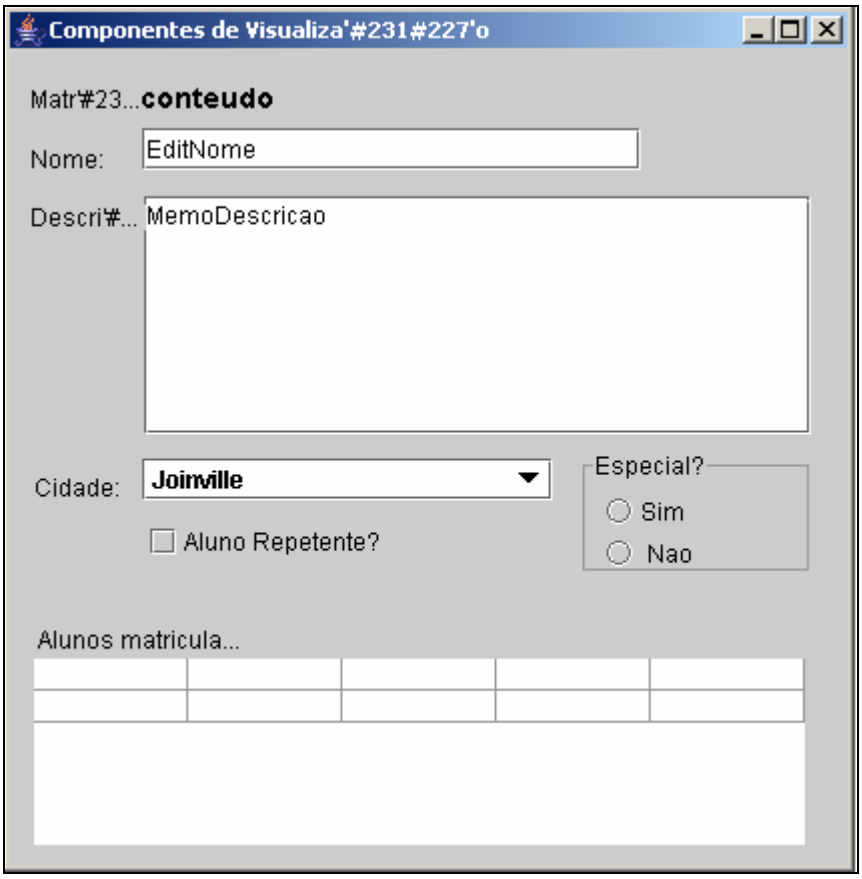


Figura 8 – Interface gerada pela ferramenta Delphi2Java-II

Seguindo essa linha, durante o segundo semestre de 2005, foi desenvolvida a

ferramenta DelphiToWeb que permite a conversão de formulários Delphi em páginas HTML (SOUZA, 2005). A ferramenta foi desenvolvida em Java, sendo que para realizar a análise (leitura e interpretação) do arquivo .DFM foram utilizados analisadores (léxico, sintático e semântico). DelphiToWeb converte um subconjunto dos componentes convertidos por Delphi2Java-II, sem também realizar a conversão de componentes de visualização de dados e de acesso a banco de dados. Além da saída HTML, são gerados também arquivos em XML e Laszlo. Na figura 9 é apresentada a tela da ferramenta DelphiToWeb e, em seguida, na figura 10 a interface gerada em HTML, a partir do mesmo arquivo .DFM usado para gerar a interface da figura 8.

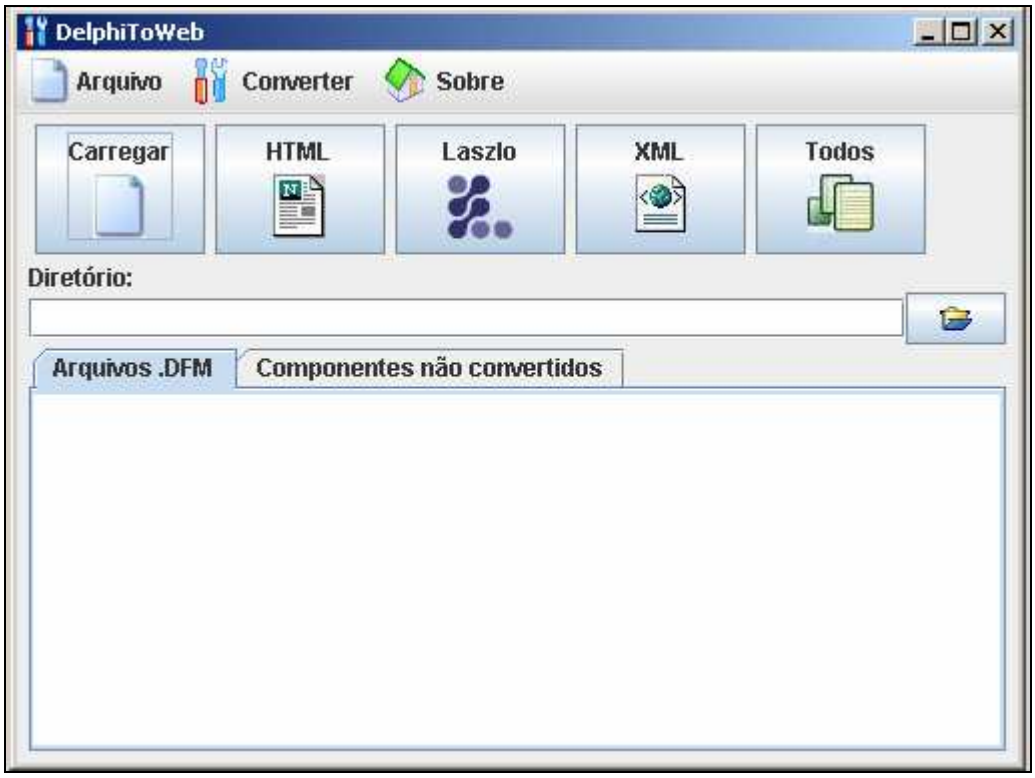


Figura 9 – Ferramenta DelphiToWeb

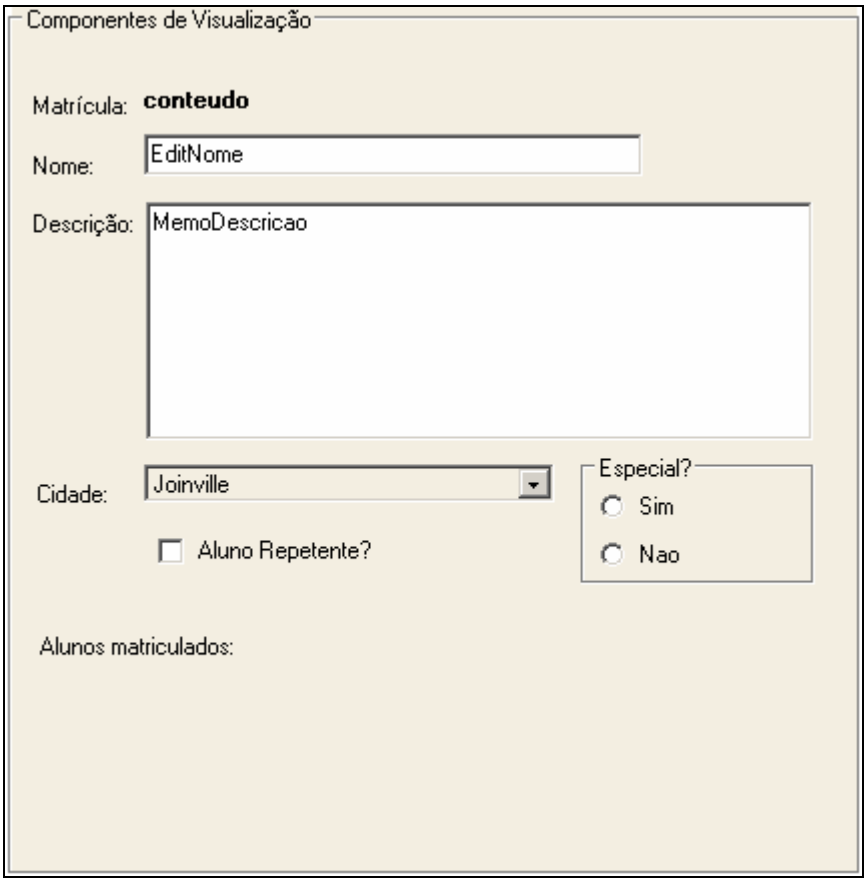


Figura 10 – Interface gerada pela ferramenta DelphiToWeb

3 DESENVOLVIMENTO DO TRABALHO

Este capítulo contextualiza e descreve o desenvolvimento de Delphi2Java-II, apresentando a migração da ferramenta, originalmente implementada em Delphi, para a linguagem Java. Descreve também a conversão dos componentes de acesso e visualização de dados. Na elaboração e construção da ferramenta Delphi2Java-II foram realizadas as seguintes etapas:

a) especificação dos requisitos: os requisitos funcionais e não-funcionais inicialmente identificados foram reavaliados e detalhados;

b) identificação de componentes: a implementação da ferramenta Delphi2Java-II (FONSECA, 2005) foi analisada para identificar como é realizada a conversão dos 27 componentes contemplados. Também foram estudados os componentes de visualização e de acesso a banco de dados disponibilizados pelo Delphi, identificando a correspondência com a biblioteca Swing e com a API JDBC;

c) especificação da saída: foram especificadas as classes Java bem como a estrutura de classes que serão geradas utilizando o padrão de projeto MVC;

d) análise da entrada: para leitura dos arquivos .DFM foi utilizada a solução para a análise desses formulários implementada na ferramenta DelphiToWeb;

e) especificação da ferramenta: foram especificados os diagramas de caso de uso e de classes utilizando os conceitos de orientação a objetos, através da *Unified Modeling Language* (UML);

f) implementação: para ler e recuperar as informações dos formulários Delphi foram utilizados os analisadores léxico, sintático e semântico da solução proposta em Souza (2005). Para gerar a saída foi utilizado o motor de *templates* Velocity. Na implementação foi usado o ambiente de desenvolvimento Eclipse 3.0.

3.1 REQUISITOS PRINCIPAIS DO PROBLEMA A SER TRABALHADO

A seguir são apresentados os requisitos funcionais (RF) e não funcionais (RNF), atendidos pela ferramenta. O quadro 5 apresenta os requisitos funcionais e sua rastreabilidade, ou seja, vinculação com o caso de uso associado. No quadro 6 são relacionados os requisitos não funcionais.

REQUISITOS FUNCIONAIS	**CASO DE USO**
RF01: Converte componentes de visualização de dados (`TDBCheckBox`, `TDBComboBox`, `TDBEdit`, `TDBGrid`, `TDBMemo`, `TDBRadioGroup`, `TDBText`) de Delphi para Java, utilizando componentes similares da biblioteca Swing.	UC01
RF02: Converter conexões com banco de dados existentes nos componentes de acesso a dados (`TDatabase`, `TQuery`, `TTable`) de Delphi para Java implementando suas principais funcionalidades.	UC01
RF03: Efetuar a interligação dos componentes de acesso a dados com os componentes de visualização de dados, implementada em Delphi através do componente `TDataSource`, utilizando a API JDBC.	UC01

Quadro 5 – Requisitos funcionais

REQUISITOS NÃO FUNCIONAIS
RNF01: Manter o *layout* original dos componentes de visualização de dados e a funcionalidade dos componentes para manipulação e acesso a banco de dados.
RNF02: Utilizar a biblioteca Swing e a Máquina Virtual Java (JVM) 1.4 para a implementação do *layout* da ferramenta.
RNF03: Utilizar o *driver* JDBC para integração do banco de dados.
RNF04: Utilizar o banco de dados Oracle8i para realizar os testes.
RNF05: Gerar código com a extensão .java.

Quadro 6 – Requisitos não funcionais

3.2 COMPONENTES CONVERTIDOS

Nesta seção são apresentados os componentes convertidos pela ferramenta Delphi2Java-II, dividindo-se em: componentes de visualização, componentes de visualização de dados e os componentes de acesso a banco de dados.

3.2.1 Componentes de visualização

Considerando que este trabalho apresenta como um de seus objetivos a migração da ferramenta Delphi2Java-II (FONSECA, 2005), de Delphi para Java, foi realizada a conversão de 22 dos 27 componentes contemplados originalmente na versão da ferramenta desenvolvida em Delphi, sendo que não foram convertidos apenas os componentes `TBitBtn`, `TPopupMenu`, `TRichEdit`, `TSpeedButton` e `TToolButton`. Foi mantida a geração da classe contendo a assinatura dos principais eventos associados com os componentes, como por exemplo, os eventos **`onCreate`**, **`onDestroy`**, **`onClick`**, **`onChange`**, **`onEnter`**, **`onExit`**, **`onCloseUp`** e **`onKeyPress`**. Os eventos **`onPopup`** e **`popupMenu`** não foram convertidos uma vez que são associados apenas com o componente `TPopupMenu`. A seguir, no quadro 7, é apresentada a lista de componentes convertidos pela versão atual de Delphi2Java-II, tendo como fundamentação a aplicação original desenvolvida em Delphi.

COMPONENTE EM DELPHI	DESCRIÇÃO	COMPONENTE EM SWING
TForm	janela principal da aplicação, ou seja, o formulário onde são inseridos os demais componentes	JDesktopPane
TButton	botão que aciona um evento quando for pressionado	JButton
TChebckBox	caixa de seleção que pode ser marcada ou desmarcada	JCheckBox
TComboBox	lista de itens dos quais apenas um pode ser selecionado	JComboBox
TCoolBar	barra de ferramentas com subdivisões, que podem ser movidas ou redimensionadas	JLayeredPane
TEdit	área com uma única linha para entrada de dados através do teclado	JFormattedTextField, JPasswordField
TLabel	texto que não pode ser editado	JLabel
TListBox	lista de itens dos quais vários podem ser selecionados	JList, JScrollPane
TMainMenu	menu principal do formulário	JMenuBar
TMemo	área com várias linhas para entrada de dados através do teclado	JtextArea, JScrollPane
TMenuItem	opções do menu principal	JMenu, JMenuItem
TPageControl	painel para agrupar e controlar várias páginas (TTabSheet)	JTabbedPane
TPanel	painel para agrupar vários componentes	JDesktopPane
TProgressBar	barra para acompanhamento do progresso de algum processo em execução	JProgressBar
TRadioButton	botão em forma de círculo para seleção de um único valor	JRadioButton
TRadioGroup	painel para agrupar TRadioButtons	JDesktopPane
TScrollBar	barra de rolagem, podendo ser horizontal ou vertical	JScrollBar
TStatusBar	área para exibir mensagens e informações do estado da aplicação	JDesktopPane
TStringGrid	caixa em formato de tabela composta por várias células para apresentar dados textuais	JTable
TSpinEdit	área com numeração inteira que pode ser incrementa ou decrementada utilizando-se de botões existentes no componente	JSpinner
TTabSheet	página para agrupar informações	JDesktopPane
TToolBar	barra de tarefas	JLayeredPane

Quadro 7 – Componentes de visualização

Na figura 11 é apresentado o *layout* de uma interface que contém todos os componentes convertidos.

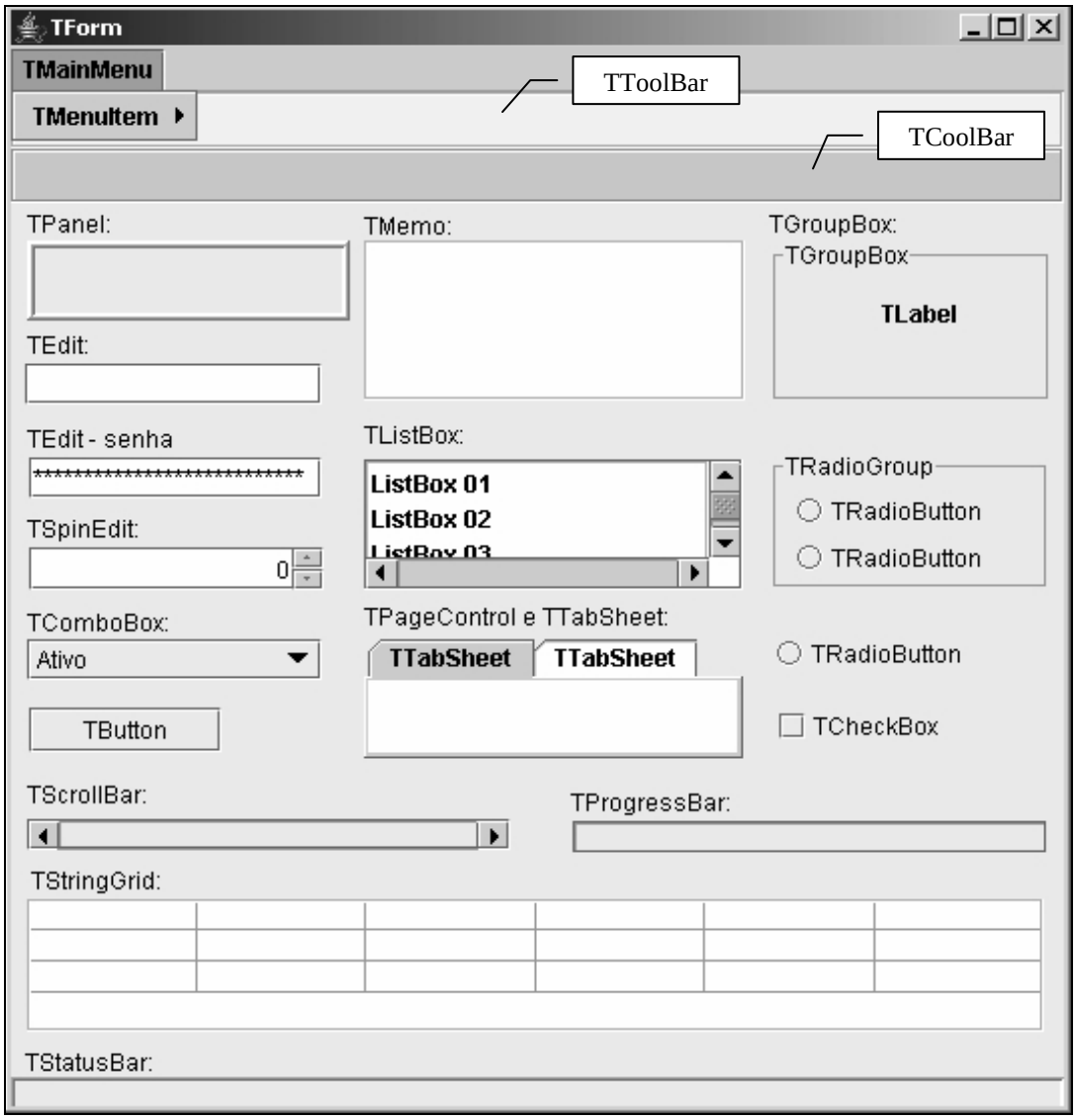


Figura 11 – Componentes de visualização gerados pela ferramenta Delphi2Java-II - versão atual

3.2.2 Componentes de visualização de dados

A ferramenta desenvolvida apresenta como um dos seus objetivos a conversão de alguns componentes de visualização de dados, que são componentes que mostram informações provenientes de um banco de dados ou de uma estrutura de dados. No quadro 8 são listados os componentes de visualização de dados que são convertidos pela ferramenta, apresentando a descrição e a representação correspondente em Java. Na figura 12 é mostrada uma interface gerada pela ferramenta contendo todos os componentes listados.

COMPONENTE EM DELPHI	DESCRIÇÃO	COMPONENTE EM SWING
`TDBCheckBox`	semelhante ao componente `TCheckBox`, mostrar uma opção, correspondente a um campo lógico, que pode ser alterada	`JCheckBox`
`TDBComboBox`	semelhante ao componente `TComboBox`, mostra um conjunto de valores, especificado geralmente através de uma consulta SQL definida num componente `TQuery`, dos quais um pode ser selecionado	`JComboBox`
`TDBEdit`	semelhante ao componente `TEdit`, mostra um campo de uma tabela que pode ser alterado	`JTextField`
`TDBGrid`	semelhante ao componente `TStringGrid`, mostra uma tabela inteira, permitindo rolagem e navegação	`JTable`
`TDBMemo`	semelhante ao componente `TMemo`, mostra um campo com várias linhas que pode ser modificado	`JTextArea`
`TDBRadioGroup`	semelhante ao componente `TRadioGroup`, mostra um conjunto de opções mutuamente exclusivas, através de vários botões `TRadioButton`	`ButtonGroup` `JDesktopPane`, `JRadioButton`
`TDBText`	semelhante ao componente `TLabel`, mostra um campo de uma tabela que não pode ser modificado	`JLabel`

Quadro 8 – Componentes de visualização de dados

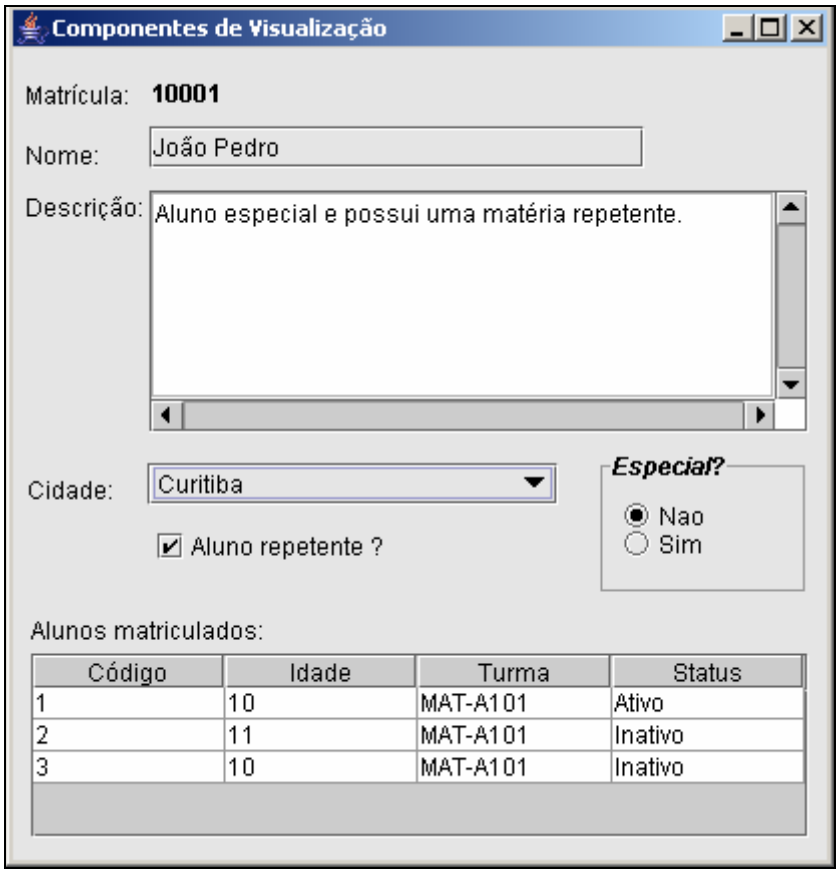


Figura 12 – Componentes de visualização de dados gerados pela ferramenta Delphi2Java-II – versão atual

3.2.3 Componentes de acesso a dados

Os componentes de visualização de dados apenas apresentam os dados de uma tabela ou consulta, ou seja, não efetuam conexão com o banco de dados. Para auxiliar neste processo, existem outros componentes que fazem a ponte entre a visualização de dados e o acesso propriamente dito ao banco de dados desejado. No quadro 9 são apresentados os componentes de acesso a dados que são convertidos pela ferramenta, assim como a sua representação em Java.

COMPONENTE EM DELPHI	DESCRIÇÃO	REPRESENTAÇÃO DO COMPONENTE EM JAVA
`TDatabase`	mantém uma conexão com o banco de dados, necessitando um *login* e uma senha, entre outras informações referentes ao banco	é representado através da classe `ConnectionManager`, que utiliza como meio de acesso ao banco de dados, as classes da API JDBC
`TDataSource`	atua como uma ponte entre os componentes de acesso a dados (`TQuery`, `TTable`) e os componentes de visualização de dados	é utilizado para identificar a qual componente de acesso a dados um componente de visualização de dados pertence, não possuindo uma saída física, ou seja, um arquivo .java. É apenas utilizado na lógica da construção dos componentes de visualização de dados
`TTable`	acessa uma tabela do banco de dados através de uma conexão com o componente `TDatabase` e carrega os dados para um ou mais componentes de visualização de dados através do componente `TDataSource`	é representado através de classes com o prefixo `Cursor`, que contêm a conexão corrente cuja consulta SQL é criada através das classes da API JDBC
`TQuery`	utiliza consultas SQL para recuperar dados de uma tabela via conexão com o componente `TDataBase` e carrega os dados para um ou mais componentes de visualização de dados através de um `TDataSource`	também é representado através de classes com o prefixo `Cursor`

Quadro 9 – Componentes de acesso a dados

3.3 ESPECIFICAÇÃO DA SAÍDA

Os componentes de visualização, de visualização de dados e de acesso a banco de

dados definidos em arquivos .DFM são convertidos para classes em Java. Assim, para padronizar e dividir em camadas a saída gerada pela ferramenta, delegando a cada classe a sua responsabilidade, é utilizado o padrão MVC.

As classes geradas são estruturadas da seguinte forma:

a) `view`: pacote que possui a classe que "desenha" os componentes de visualização e de visualização de dados do .DFM analisado. O nome da classe é composto pelo nome do arquivo DFM seguido por `FRM.java`;

b) `model`: pacote que contém as classes de acesso à camada de banco de dados, geradas a partir dos componentes de acesso a dados e sua relação com os de visualização de dados, juntamente com a classe que irá efetuar a conexão com o banco de dados via JDBC, a classe `ConnectionManager.java`. A partir desse pacote, foram criados mais dois pacotes internos:

   - `bean`: pacote que contém todas as classes `Bean`, que representam os campos das tabelas utilizadas, assim como seus métodos de inclusão, alteração e exclusão. É representada pela descrição `Bean` seguida do nome da tabela em questão seguido da extensão `.java`,
   - `cursor`: pacote que possui todas as classes com o prefixo `Cursor`, que representam as consultas a serem utilizadas pelos componentes de visualização de dados. Para cada componente de acesso a dados é criada uma classe denominada `Cursor` + `nome do componente de acesso a dados` + `.java`;

c) `controller`: esse pacote contém a classe de eventos da interface, cuja funcionalidade é manipular todos os eventos dos componentes de interface, assim como suas validações, carregando valores para os componentes de visualização de dados. É descrita pelo nome do arquivo DFM seguido de `Event.java`.

A partir da estrutura de classes geradas seguindo o padrão MVC, para utilizar a API

JDBC é necessário adicionar ao projeto os pacotes para acesso ao *driver* JDBC (`classes12.jar`, `classes12dms.jar`, `nls_charset.jar`), que compõem o *driver* de acesso direto ao servidor, conforme apresentado na seção 2.5.

Um exemplo de uma estrutura de classes geradas no padrão MVC encontra-se na figura 13.

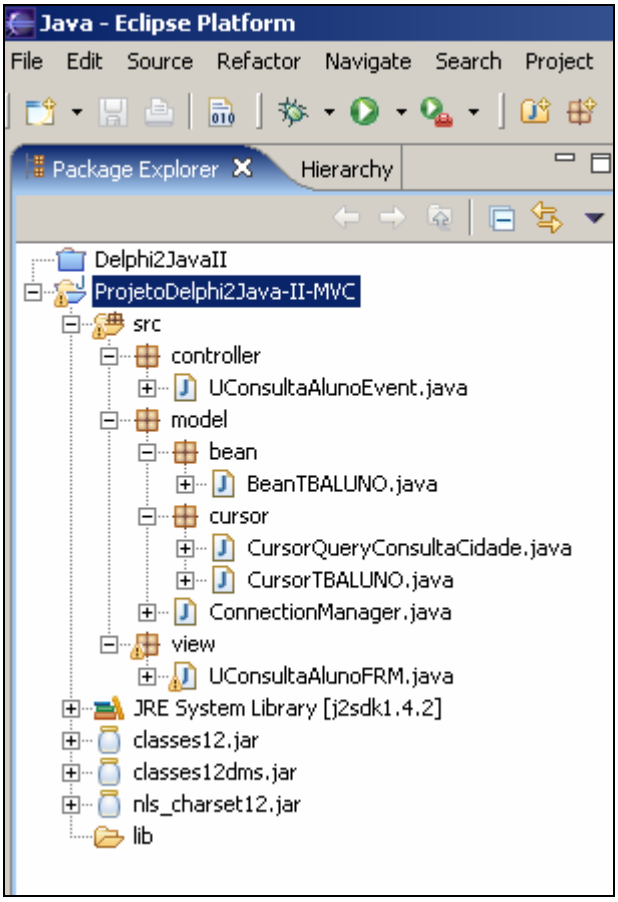
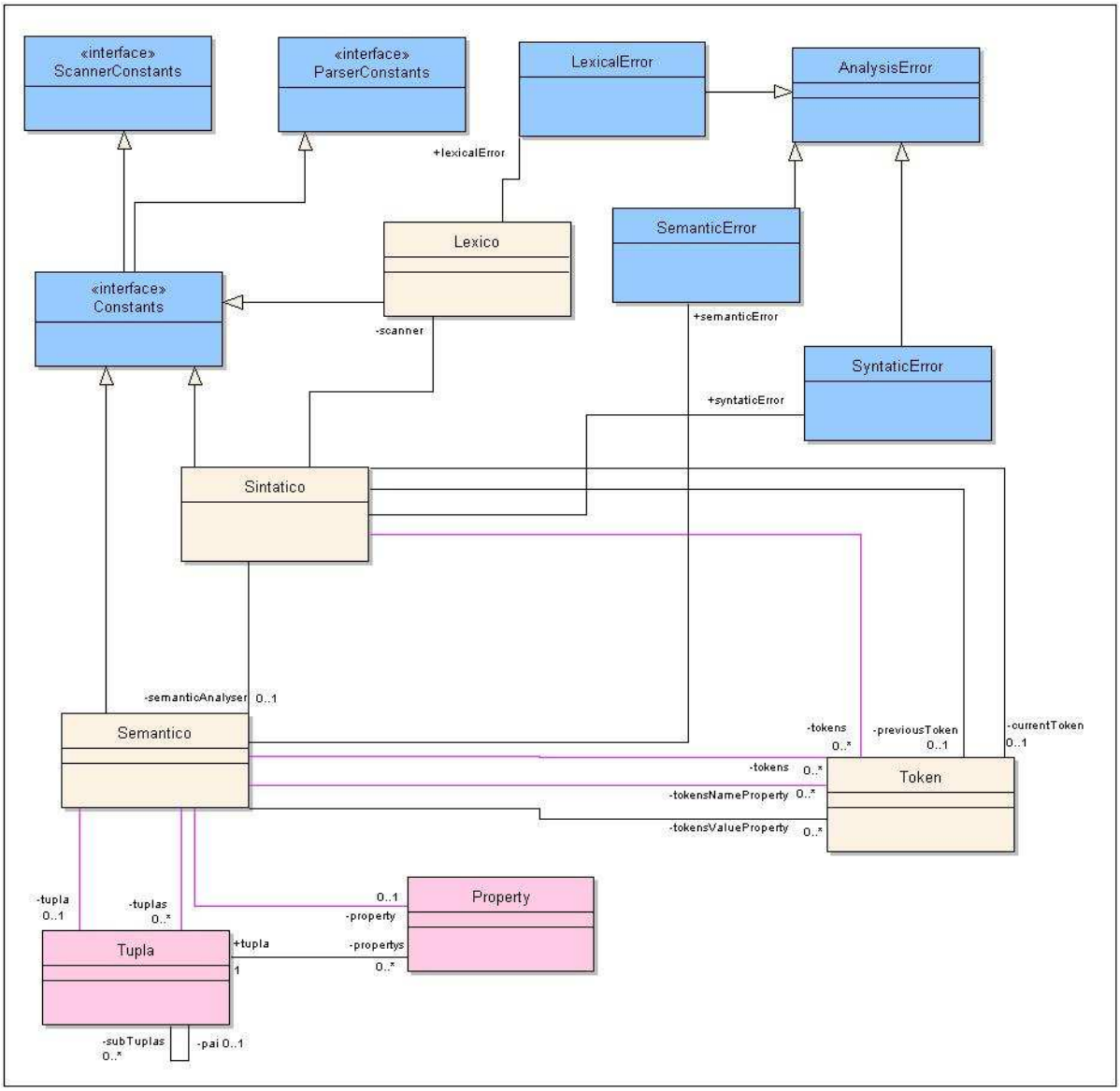


Figura 13 – Estrutura do projeto MVC gerado pela ferramenta Delphi2Java-II – versão atual

3.4 ANÁLISE DA ENTRADA

Para realizar a análise de entrada do arquivo .DFM, foi utilizada como ponto de partida a solução proposta por Souza (2005) na implementação da ferramenta DelphiToWeb. A partir da definição da estrutura de um arquivo .DFM (Anexo A), escrita usando a notação da *Backus-Naur-Form* (BNF), são geradas pelo GALS (GESSER, 2003), os analisadores léxico,

sintático e semântico. Desta maneira, para melhor apresentar as classes utilizadas segundo o modelo implementado (SOUZA, 2005, p. 39), segue na figura 14 o diagrama de classes, onde as classes em azul foram geradas pelo GALS, os analisadores léxico, sintático e semântico, também gerados pelo GALS, são representados na cor amarela, uma vez que foram adaptadas na implementação de DelphiToWeb e as classes na cor rosa, foram criadas a partir do desenvolvimento da referida ferramenta para armazenar os dados extraídos dos arquivos .DFM.



Fonte: Souza (2005, p. 42).

Figura 14 – Classes para análise do arquivo .DFM

3.5 ESPECIFICAÇÃO

Delphi2Java-II foi especificada com a ferramenta Enterprise Architect, utilizando os conceitos de orientação a objetos e baseando-se nos diagramas da UML, gerando como produtos o diagrama de caso de uso e os diagramas de classes.

3.5.1 Diagrama de caso de uso

O diagrama de caso de uso representa a interação do usuário com a ferramenta, destacando as ações que podem ser realizadas. A figura 15 mostra o diagrama de caso de uso da ferramenta e no quadro 10 é apresentado o detalhamento deste caso de uso.

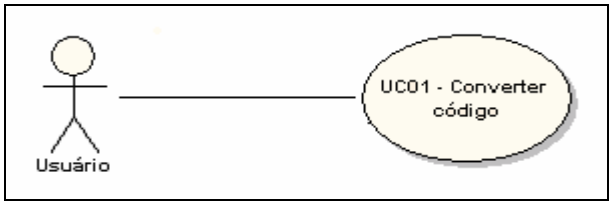


Figura 15 – Diagrama de caso de uso UC01

UC01 - Converter código

Pré-condição: O(s) arquivo(s) .DFM não pode(m) estar corrompido(s).

Cenário principal:

1. A ferramenta apresenta as opções para carregar o(s) arquivo(s) e o diretório para salvar os arquivos gerados.
2. O usuário seleciona o(s) arquivo(s) que deseja carregar e o diretório onde deseja salvar os arquivos gerados.
3. A ferramenta apresenta a relação de arquivos carregados.
4. O usuário seleciona o(s) arquivo(s) que deseja converter.
5. A ferramenta apresenta as opções para gerar código para arquivo(s) selecionado(s) e para gerar código para todos os arquivos carregados.
6. O usuário seleciona a opção desejada.
7. A ferramenta faz a conversão do código.

Pós-condição: Dois ou mais arquivos com a extensão .java gerados para cada arquivo selecionado.

Quadro 10 – Detalhamento do caso de uso

3.5.2 Diagrama de classes

O diagrama de classes mostra uma visão lógica de como as classes da ferramenta estão associadas. Como já citado anteriormente, no diagrama da figura 14 estão representadas as classes de análise do arquivo .DFM, onde apenas a classe `Tupla`, implementada na ferramenta DelphiToWeb, foi customizada para as necessidades da ferramenta Delphi2Java-II, permanecendo as demais inalteradas. A seguir, no quadro 11, é dada uma breve descrição das classes da ferramenta DelphiToWeb que estão sendo reutilizadas no desenvolvimento da ferramenta proposta.

CLASSE	DESCRIÇÃO
`AnalysisError`	classe para tratamento de erros ocorridos quando da análise do arquivo .DFM
`LexicalError`	classe, descendente de `AnalysisError`, para tratamento de erros léxicos
`SyntaticError`	classe, descendente de `AnalysisError`, para tratamento de erros sintáticos
`SemanticError`	classe, descendente de `AnalysisError`, para tratamento de erros semânticos
`Constants`	constantes utilizadas pelos analisadores léxico e sintático
`ScannerConstant`	constantes com as palavras reservadas utilizadas em arquivos .DFM (`OBJECT`, `TRUE`, `FALSE`, `END`, entre outras), bem como as mensagens de erro léxico
`ParserConstant`	constante com as mensagens de erro sintático
`Lexico`	efetua a análise léxica do arquivo .DFM, identificando *tokens*
`Sintatico`	efetua a análise sintática do arquivo .DFM, possuindo associação com a classe `Lexico` para reconhecimento dos *tokens*, com a classe `Token` para armazenar os *tokens* corrente e anterior, e com a classe `Semantico` para acionar as ações semânticas, conforme especificado em Souza (2005)
`Semantico`	efetua a análise semântica do arquivo .DFM, ou seja, armazena em objetos das classes `Tupla` e `Property` as informações extraídas do arquivo
`Token`	armazena a classe (palavra reservada, identificador, símbolo especial), a posição (no arquivo .DFM) e o valor do *token* (por exemplo, [identificador, 17, `TableTBAluno`])
`Tupla`	armazena as informações de cada componente do arquivo .DFM, sendo elas: o identificador do componente (por exemplo, `TableTBAluno`), o tipo (`TTable`), a lista de propriedades a partir da associação com a classe `Property` e os possíveis objetos componentes a partir da associação com ela mesma
`Property`	armazena as propriedades de cada componente do arquivo .DFM, tendo cada uma um nome (por exemplo, `DatabaseName`) e um valor (`'BDoracle'`)

Quadro 11 – Relação de classes reutilizadas da ferramenta DelphiToWeb

O diagrama de classes apresentado na figura 16 mostra a visão do modelo estrutural, no que se refere às classes que dão início à geração de conversão de código da ferramenta. A classe `EventMain` é a classe que trata os eventos da ferramenta. Utiliza uma instância da

classe `Conversor` para executar a conversão do arquivo .DFM selecionado. A classe `Conversor` é responsável por ler o arquivo .DFM e extrair as informações necessárias para gerar código, através da execução das análises léxica, sintática e semântica. Cria uma instância de `Tupla`, que contém toda a estrutura do arquivo .DFM, e utiliza essa instância para executar `GeradorVelocity`, classe responsável por acionar o motor de *templates* Velocity, a partir do qual são geradas as classes de saída.

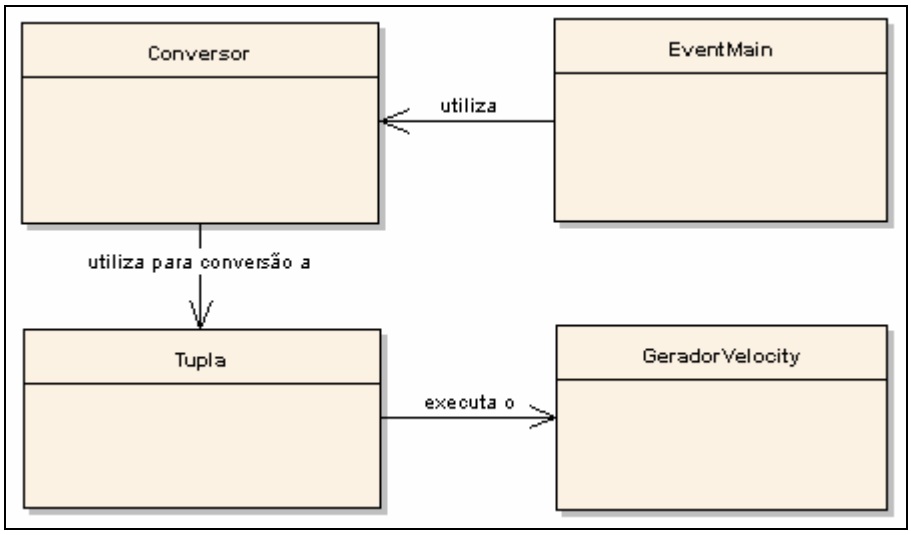


Figura 16 – Classes para geração de código

A classe `Component` é responsável por definir todas as propriedades e ações comuns aos objetos que contêm as informações dos componentes do arquivo .DFM. Esta classe possui como descendentes as classes de visualização, representadas na figura 17, bem como as classes que representam os componentes de visualização de dados e as classes que representam o acesso a banco de dados, visualizadas nas figuras 18 e 19, respectivamente.

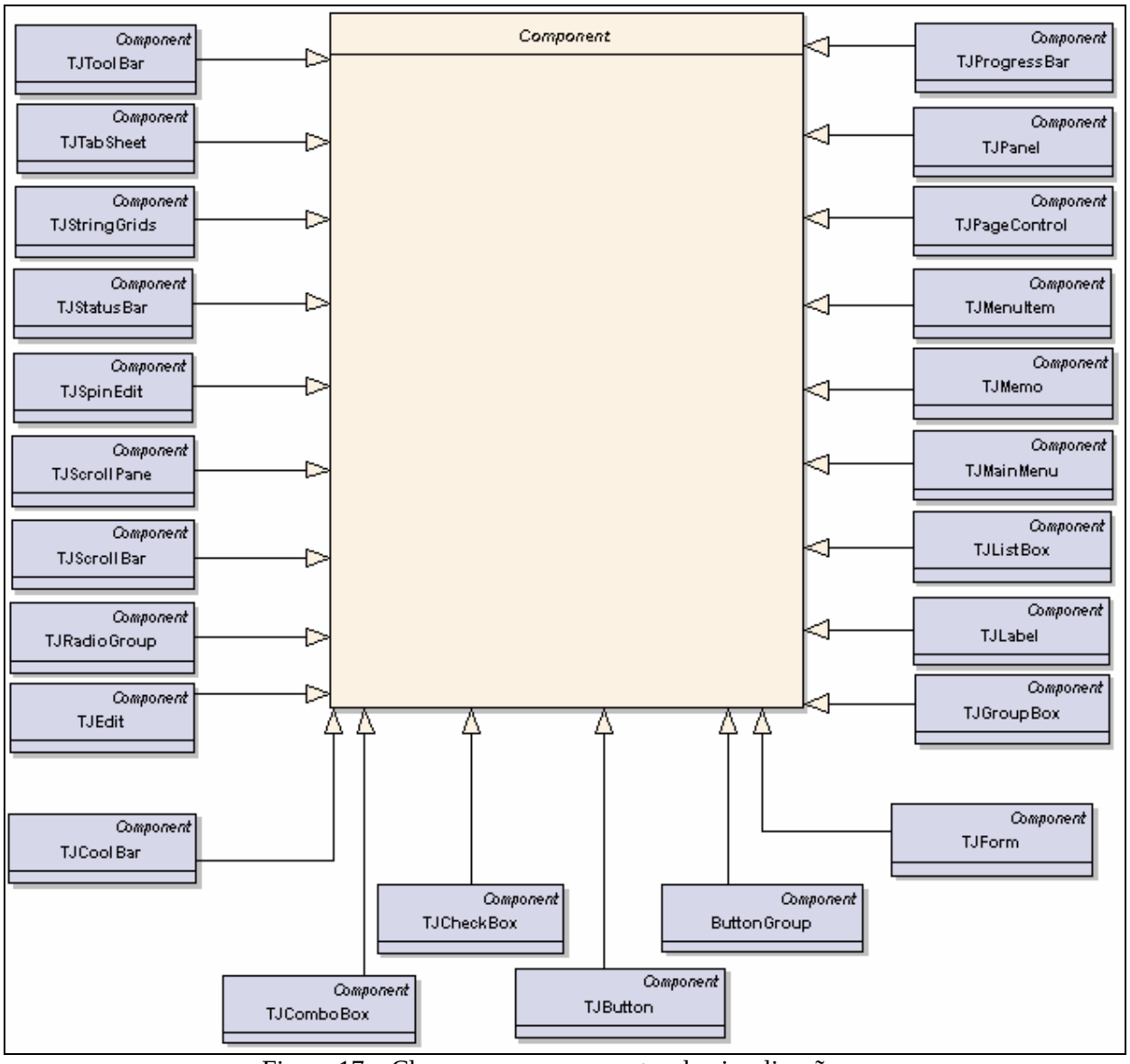


Figura 17 – Classes para componentes de visualização

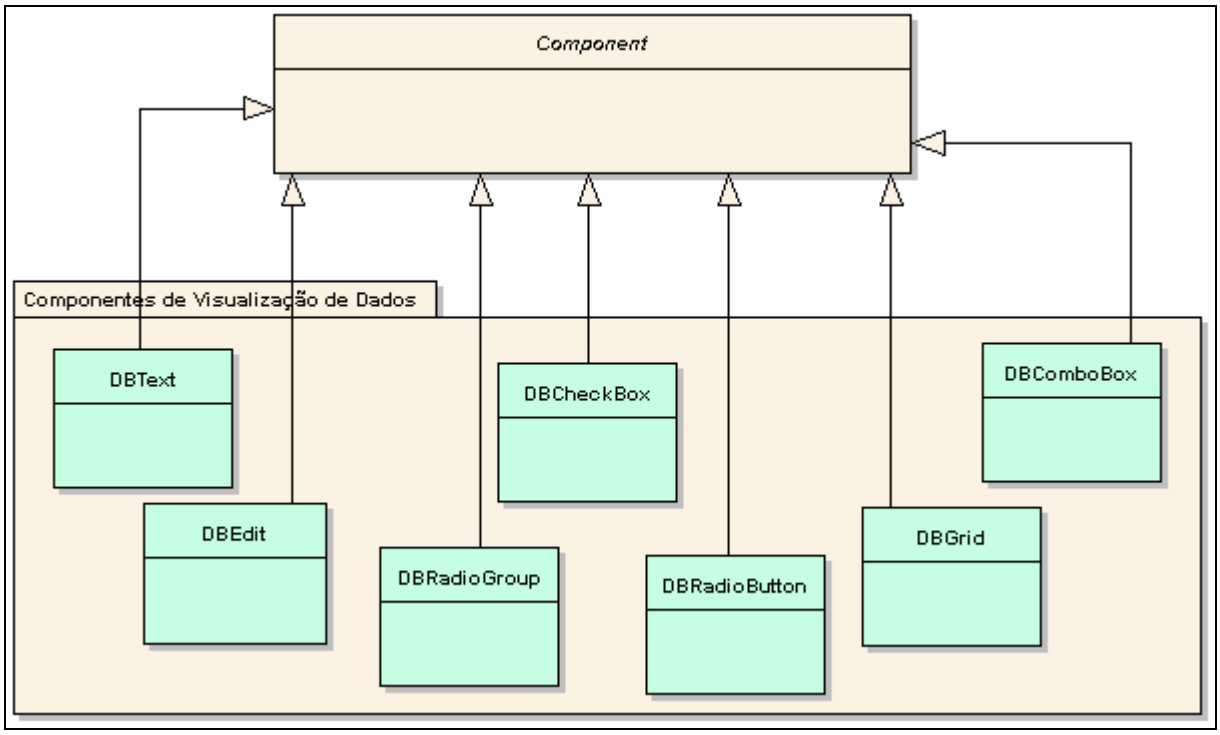


Figura 18 – Classes para componentes de visualização de dados

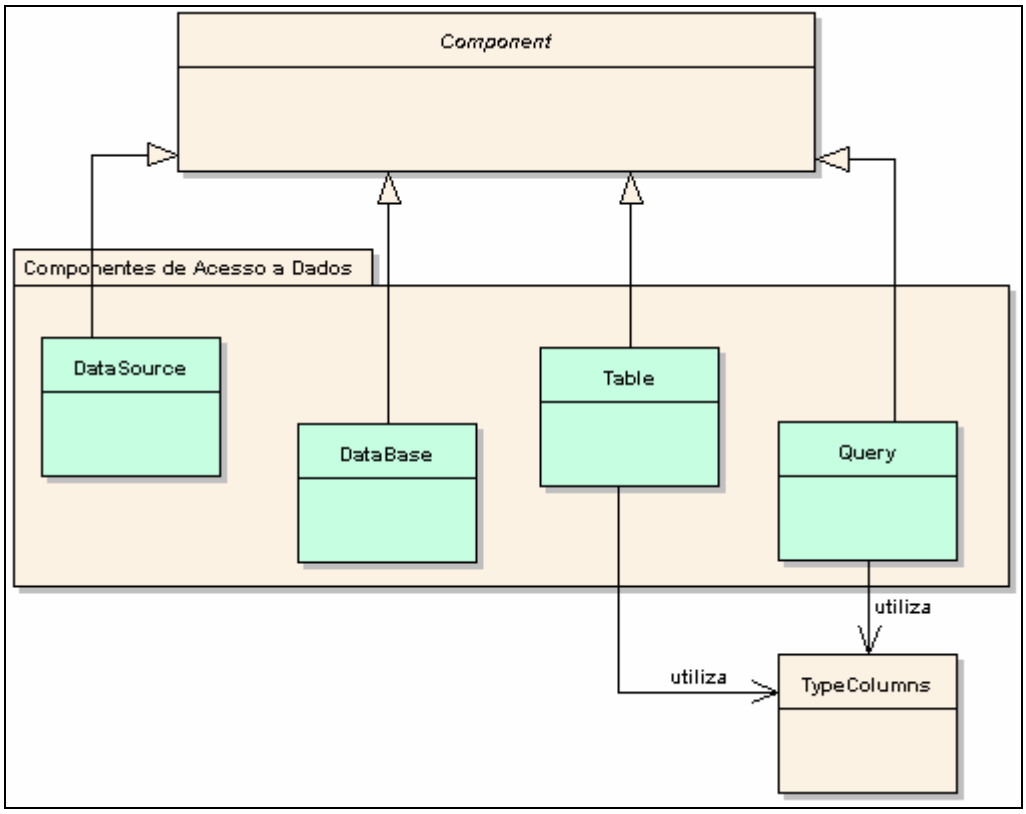


Figura 19 – Classes para componentes de acesso a banco de dados

Os principais métodos das classes que fazem acesso a banco de dados são descritos no quadro 12.

CLASSE	MÉTODO / DESCRIÇÃO
`Database`	**`setProperties`** define as propriedades da conexão, tais como o nome do banco de dados, o usuário, a senha e o *driver*
	`getUrl` retorna a URL (`"jdbc:oracle:thin:@localhost:1521:BDOracle"`) da conexão, a partir do nome do banco
	`getDriver`, **`getNomeBanco`**, **`getSenha`**, **`getUsuario`** retornam, respectivamente, o *driver* JDBC a ser utilizado, o nome do banco de dados, a senha e o usuário da conexão
	`setDriver`, **`setNomeBanco`**, **`setSenha`**, **`setUsuario`** definem o *driver* JDBC (`"oracle.jdbc.driver.OracleDriver"`) , o nome do banco de dados, a senha e o usuário da conexão, sendo esses três últimos valores recuperados do arquivo .DFM
`DataSource`	**`getDataSet`** retorna o componente de acesso a banco de dados (`TQuery` ou `TTable`) correspondente ao objeto `DataSource`
	`getName` retorna o nome do componente
`Table`	**`setProperties`** define as propriedades da tabela, ou seja, o nome da tabela no banco de dados
	`getTableName` retorna o nome da tabela no banco de dados
	`setTableName` define o nome da tabela no banco de dados
	`getArrayColumnTypeInteger`, **`getArrayColumnTypeString`**, **`getArrayColumnTypeDate`** retornam estruturas de dados com as colunas do tipo `Integer`, do tipo `String` e do tipo `Date`, respectivamente
	`getArrayNameFieldColumn` retorna uma estrutura de dados contendo o nome das colunas encontradas na tabela
	`getSubTypeObjectsFromComponentsDB` recupera, através do componente `TDataSource`, o nome das colunas utilizadas pelos componentes de visualização de dados associados ao componente `TTable`
	`getConteudoClasseBean`, **`getConteudoClasseCursor`** retornam, respectivamente, o conteúdo das classes `Bean` e `Cursor` para a geração do código de saída
	`setConteudoClasseBean`, **`setConteudoClasseCursor`** definem, respectivamente, o conteúdo das classes `Bean e Cursor`
`Query`	**`setProperties`** define o conteúdo da consulta SQL
	`getArrayNameFieldColumn` retorna uma estrutura de dados com o nome das colunas utilizadas na consulta SQL
	`getFieldFromComponentsDB` recupera, através do componente `TDataSource`, o nome das colunas utilizadas pelos componentes de visualização de dados associados ao componente `TQuery`
	`getSqlStrings` retorna a consulta SQL
	`setSqlStrings` define a consulta SQL
	`getConteudoClasseCursorQuery` retorna o conteúdo da classe `Cursor` para a geração do código de saída
	`setConteudoClasseCursorQuery` define o conteúdo da classe `Cursor`

Quadro 12 – Métodos das classes para os componentes de acesso a dados

3.6 IMPLEMENTAÇÃO

Nessa seção são descritas as ferramentas e técnicas utilizadas, bem como a implementação do trabalho.

3.6.1 Técnicas e ferramentas utilizadas

A ferramenta aqui apresentada foi construída na linguagem Java, utilizando o ambiente de desenvolvimento Eclipse, a biblioteca gráfica Swing e o JDK na versão 1.4.2. O motor de *templates* Velocity foi utilizado através da biblioteca `velocity-dep-1.3.jar`. Para elaboração dos arquivos com extensão DFM foi utilizado o ambiente Delphi 7.0. Para os testes foi utilizado o banco Oracle8i e um projeto em Delphi, `ProjetoDelphi2Java-II-MVC`, contendo os componentes de visualização de dados e de acesso a banco de dados.

3.6.2 Implementação da ferramenta

A implementação da ferramenta iniciou com a migração do código-fonte da ferramenta Delphi2Java-II (FONSECA, 2005), de Delphi para Java. Como foi dito anteriormente, Delphi2Java-II gera classes Java a partir de arquivos .DFM. Para analisar esses arquivos, é feita a leitura dos mesmos caracter a caracter, identificando tanto os símbolos quanto a estrutura de formulários Delphi. Na implementação descrita em Fonseca (2005) não foram usadas técnicas para construção de compiladores. Portanto, para facilitar a leitura e validação dos arquivos .DFM, foi visto que seria necessária a utilização de compiladores para fazer a análise correta dos arquivos. Sendo assim, para processar a análise do formulário Delphi, foram utilizadas as classes da ferramenta DelphiToWeb, conforme descrito na seção 3.4, que

implementam as análises léxica, sintática e semântica.

A análise da entrada ocorre na classe `Conversor`, que efetua a validação do arquivo .DFM, através da execução das análises léxica, sintática e semântica (trecho 01). Caso não ocorra nenhum erro, como arquivo corrompido, uma instância da classe `Tupla`, contendo as informações extraídas do formulário principal da aplicação, é criada. A partir deste objeto, é invocado o método **`convertSwing`**, que irá iniciar o processo de conversão. O quadro 13 mostra um trecho do código citado.

```
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);
  } catch (AnalysisError e) {
    throw new AnalysisError("O arquivo " + file.getName()+ " está com erro!");
  }
  ...
}
```

Quadro 13 – Código-fonte do método **`geraArquivoConversao`** da classe `Conversor`

Inicialmente, no método **`convertSwing`** da classe `Tupla` é verificado se o primeiro componente é do tipo `TForm`, sendo criada uma instância da classe `TJForm` através da chamada do método **`getJForm`**. A partir do objeto pai, no caso o componente `TForm`, é iniciado o mapeamento das propriedades, eventos e objetos filhos de `TJForm`, criando assim toda a estrutura de varredura dos outros componentes existentes no .DFM.

Todo os componentes existentes no .DFM que são convertidos pela ferramenta são armazenados pelo método **`getSubObjects`** da classe `Component` (quadro 14) em estruturas de dados (do tipo `ArrayList`), sendo que existe uma estrutura definida para cada tipo de componente e a mesma pode ter vários objetos componentes armazenados. Cita-se como exemplo a estrutura `arrayDBComboBox` que irá armazenar todos os componentes do tipo `TDBComboBox` existentes no formulário .DFM (trecho 01). Isso visa facilitar a manipulação de todos os objetos quando da geração do código de saída. Os componentes que não são

convertidos pela ferramenta são relacionados em uma área específica (trecho 02).

```
public void getSubObjects(ArrayList subObjetos) {
  Conversor     conv  = Conversor.getInstance();
  ...
  if (subObjetos != null) {
    for (int i = 0; i < subObjetos.size(); i++) {
      Tupla tu = (Tupla) subObjetos.get(i);
      //------------------------------------------- Componentes de visualização
      if (tu.getType().equals("TLabel")) {
        addLabel(new TJLabel(tu));
        acionaEvento(tu);
      ...
      //------------------------------------------- Componentes de banco de dados
      }else if (tu.getType().equals("TDatabase")){
        new Database(tu);
        setPossuiBD(true);
      //------------------------------------------- TDBComboBox
      }else if (tu.getType().equals("TDBComboBox")){
        addAllComponentDB(new DBComboBox(tu));
        acionaEvento(tu);
        addDBComboBox(new DBComboBox(tu));   01
        setPossuiComponentInterfaceDB(true);
      ...
      //------------------------------------------- Componentes não convertidos
      }else ... {
        naoConvertidos.append("Componente :" +                              02
                              tu.getType() +
                              " - Arquivo: " +
                              Conversor.getInstance().getFileName() + "\n");
        }
    }
  }
}
```

Quadro 14 – Código-fonte do método **`getSubObjects`** da classe `Component`

No quadro 15 é mostrado um trecho da implementação do método **`acionaEvento`**, contido na classe `Component`, responsável por criar os eventos para cada componente do formulário, caso o mesmo exista no arquivo .DFM. Os eventos também são adicionados em estruturas de dados de eventos (trechos 01 e 02), sendo uma para cada tipo de evento identificado (`arrayComponentesOnEnter`, `arrayComponentesOnExit`, entre outros).

```
public void acionaEvento(Tupla tu){
  ...
  ArrayList properties = tu.getPropertys();
  if (properties != null) {
    for (int c = 0; c < properties.size(); c++) {
      Property pro = (Property) properties.get(c);
      ...
      //------------------------------------------- Evento OnEnter
      if (pro.getName().equals("OnEnter")){
        String onEnter = pro.getValue();
        if (onEnter != null && !onEnter.trim().equalsIgnoreCase("")){
          addComponentesEventoOnEnter(tu.getIdentify());   01
        }
      //------------------------------------------- Evento OnExit
      }else if (pro.getName().equals("OnExit")){
        String onExit = pro.getValue();
        if (onExit != null && !onExit.trim().equalsIgnoreCase("")){
          addComponentesEventoOnExit(tu.getIdentify());   02
        }
      ...
}
```

Quadro 15 – Código-fonte do método **acionaEvento** da classe Component

Com os dados necessários armazenados nas estruturas de dados, o método **convertSwing** cria uma instância da classe GeradorVelocity para realizar a geração da saída a partir dos *templates*. Para cada *template* definido, é executado um método para analisá-lo, como ilustrado no quadro 16.

```
public GeradorVelocity(TJForm form,
                       FrmMain screen,
                       String nomeUnitForm,
                       String nomeUnitEventos,
                       boolean formPrincipal) throws Exception{
  ...
  //-- Caminho de acesso aos templates
  caminhoDiretorioTemplates.setProperty("file.resource.loader.path",
                                        "./src/templates/");
  //------------------------------------------- Código de saída
  geraSaidaInterface();
  geraSaidaEventos();
  //-- caso possua acesso a banco de dados, cria a classe de conexão ao banco
  if (Component.isPossuiBD()== true){
    geraClasseConexaoBD();
  }
  //-- caso possua componente TTable, cria Bean e Cursor específicos
  if (Component.isPossuiTable()== true){
    for (int i=0 ;i < Component.getArrayTable().size();i++){
      geraClasseBean((Table)Component.getArrayTable().get(i));
      geraClasseCursor((Table)Component.getArrayTable().get(i));
    }
  }
  //-- caso possua componente TQuery, cria Cursor específico
  if (Component.isPossuiQuery() == true){
    for (int j = 0; j < Component.getArrayQuery().size(); j++){
      geraClasseCursor((Query) Component.getArrayQuery().get(j));
    }
  }
  geraSaidaHTML();
}
```

Quadro 16 – Código-fonte do construtor da classe GeradorVelocity

A ferramenta possui 10 *templates* usados para a construção das classes de saída. No quadro 17 estão relacionados os *templates* definidos, assim como suas finalidades.

TEMPLATE	FINALIDADE
`templateBeans.vm`	molde da classe `Bean` – usado por: **`geraClasseBean`**
`templateConnectionManager.vm`	molde da classe de conexão com o banco de dados via JDBC – usado por: **`geraClasseConexaoBD`**
`templateCursor.vm`	molde da classe `Cursor` do componente de acesso a banco de dados (`TTable` ou `TQuery`) – usado por: **`geraClasseCursor`**
`templateDeclaracaoComponentesSwing.vm`	molde para a declaração dos componentes de visualização e dos componentes de visualização de dados – usado por: **`geraSaidaInterface`**
`templateEventos.vm`	molde da classe que contém as assinaturas dos eventos dos componentes, assim como os métodos de validação do banco de dados – usada por: **`geraSaidaEventos`**
`templateHTML.vm`	molde para gerar arquivo HTML – usado por: **`geraSaidaHTML`**
`templateImplementacaoDBSwing.vm`	molde dos métodos de criação e manipulação dos componentes de banco de dados – usado por: **`geraSaidaEventos`**
`templateInterfaceDBSwing.vm`	molde dos componentes de visualização de dados – usado por: **`geraSaidaInterface`**
`templateInterfaceSwing.vm`	molde da classe de interface contendo os componentes de visualização e de visualização de dados – usado por: **`geraSaidaInterface`**
`templateSetGetComponentesSwing.vm`	molde dos métodos `set` e `get` de cada componente da classe de interface – usado por: **`geraSaidaInterface`**

Quadro 17 – *Templates* da ferramenta Delphi2Java-II

Em cada um dos métodos da classe `GeradorVelocity` que analisa os *templates*, é necessário especificar um contexto para o mapeamento entre as estruturas de dados criadas pelo método **`convertSwing`** e os elementos dinâmicos definidos no *template*. No quadro 18 é visualizado um método que analisa um *template* Velocity. O método **`geraClasseCursor`** é responsável por carregar o *template* `templateCursor.vm` e criar um contexto (trecho 01) para associar `query`, estrutura que armazena os dados extraídos do arquivo .DFM, às variáveis dinâmicas `comandoSql`, `nameTable` e `getArrayNameFieldCollumn` (trecho 02). No quadro

19 é apresentado o *template* correspondente.

```
public void geraClasseCursor(Query query){
  try{
    //-- inicializando o velocity
    VelocityEngine vCursorQuery = new VelocityEngine();
    vCursorQuery.init(caminhoDiretorioTemplates);

    VelocityContext context = new VelocityContext();                    01
    Template tCursorQuery = vCursorQuery.getTemplate("templateCursor.vm");

    context.put("comandoSql",query.getSqlStrings());                    02
    context.put("nameTable",query.getName()); //Nome da Tabela
    context.put("getArrayNameFieldColumn",query.getArrayNameFieldColumn());

    StringWriter writerCursorQuery = new StringWriter();
    tCursorQuery.merge(context, writerCursorQuery);
    query.setConteudoClasseCursorQuery(writerCursorQuery.toString());
  ...
  }
}
```

Quadro 18 – Código-fonte do método **`geraClasseCursor`** da classe `GeradorVelocity`

```
package model.cursor;

/**
 * Import's utilizados.
 **/
import java.sql.*;
import java.util.*;
import model.ConnectionManager;

/**
 * Cursor gerado a partir da TABELA $!{nameTable}.
 **/
public class Cursor$!{nameTable} extends ConnectionManager{

  //--retorna dados da tabela $!{nameTable}
  public ArrayList getDados$!{nameTable}() throws SQLException, Exception{

    Connection conexao = this.retornaConexao();
    String     sql = "$!{comandoSql}";
    PreparedStatement preparedStatement = conexao.prepareStatement(sql);
    ResultSet  resultado = preparedStatement.executeQuery();
    HashMap    hmResult$!{nameTable} = new HashMap();
    ArrayList  arrayResult$!{nameTable} = new ArrayList();

    int cont = 0;

    while (resultado.next()){
      hmResult$!{nameTable} = new HashMap();
      #foreach ($nameFieldColumn in $getArrayNameFieldColumn)
         hmResult$!{nameTable}.put("$!{nameFieldColumn}",
                                   resultado.getString("$!{nameFieldColumn}"));
      #end
      arrayResult$!{nameTable}.add(cont++, hmResult$!{nameTable});
    }

    preparedStatement.close();

    return arrayResult$!{nameTable};
  }
}
```

Quadro 19 – `templateCursor.vm`

3.6.3 Classes geradas

Como descrito na seção 3.3, a partir dos *templates*, são geradas classes estruturadas conforme o padrão MVC.

3.6.3.1 Classe com componentes de visualização de dados (pacote `view`)

Na mesma classe onde são criados os componentes de visualização, são também criados os componentes de visualização de dados. Os componentes de visualização de dados são criados a partir das classes `DBCheckBox`, `DBComboBox`, `DBEdit`, `DBGrid`, `DBMemo`, `DBRadioGroup` e `DBText`. A representação dos componentes de visualização dados segue o mesmo padrão das classes de visualização, onde para cada componente foi definido, no *template*, como deveria ser a saída do mesmo. Um exemplo de código gerado para um componente de visualização de dados incluindo métodos **`set`** e **`get`** é mostrado no quadro 20.

```
...
  FormBD.add(classeEvento.criaComboBoxDBComboBoxCidade(),
  javax.swing.JLayeredPane.DEFAULT_LAYER);
  DBComboBoxCidade.setBackground(new java.awt.Color(229,229,229));
  DBComboBoxCidade.setBounds(58, 207, 205, 21);
  DBComboBoxCidade.setVisible(true);
  DBComboBoxCidade.setFont(new java.awt.Font("MS Sans Serif", 0,12));
  DBComboBoxCidade.setForeground(new java.awt.Color(0, 0, 0));
...

  /**
   * @return Returns the DBComboBoxCidade.
   **/
  public javax.swing.JComboBox getDBComboBoxCidade() {
    return DBComboBoxCidade;
  }

  /**
   * @param comboBoxCidade The dBComboBoxCidade to set.
   **/
  public void setDBComboBoxCidade(javax.swing.JComboBox arg) {
    DBComboBoxCidade = arg;
  }
...
```

Quadro 20 – Código-fonte com a criação do componente **`DBComboBox`** gerado pela ferramenta

3.6.3.2 Classe de conexão com banco de dados (pacote `model`)

A partir de um formulário Delphi contendo o componente `TDatabase`, são recuperadas as informações necessárias para a criar uma conexão com o banco de dados, utilizando a classe `Connection` da API JDBC. A classe `ConnectionManager.java`, gerada para manipular a conexão com o banco de dados, possui basicamente três métodos, apresentados no quadro 21. O método **`getConnection`**, como destacado no trecho 01, recebe como parâmetros as informações recuperadas do componente `TDataBase`, quais sejam o `driver`, a `url`, o `usuário` e a `senha` de acesso ao banco de dados; realiza o registro do *driver* JDBC (trecho 02) e obtém conexão com o banco de dados (trecho 03). Para recuperar o objeto que representa a conexão, existe o método **`retornaConexao`** (trecho 04). E para finalizar a conexão criada existe o método **`fechaConexao`** (trechos 05 e 06).

```
public class ConnectionManager {

  private static java.sql.Connection conexao;

  public java.sql.Connection getConnection(String driver,                    01
                                           String url,
                                           String usuario,
                                           String senha) throws Exception{
    try{
      Class.forName(driver);   02
      if(ConnectionManager.conexao == null){
    03  ConnectionManager.conexao = DriverManager.getConnection(url,usuario,senha);
        ConnectionManager.conexao.setAutoCommit(false);
      }
    } catch (Exception e) {
        JOptionPane.showMessageDialog(null, "Erro na conexão: " + e.getMessage());
    }
    return ConnectionManager.conexao;
  }

  public java.sql.Connection retornaConexao() throws Exception{              04
    java.sql.Connection conexao = null;
    conexao = this.getConnection("oracle.jdbc.driver.OracleDriver",
                                 "jdbc:oracle:thin:@localhost:1521:BDORACLE",
                                 "janira",
                                 "janira");
    return conexao;
  }

  public void fechaConexao(Connection conn){                                 05
    try {
      conexao.close();  06
    } catch (SQLException e) {
        e.printStackTrace();
        JOptionPane.showMessageDialog(null,"Erro para fechar a conexão!");
    }
  }
}
```

Quadro 21 – Código-fonte do arquivo `ConnectionManager.java` gerado pela ferramenta

3.6.3.3 Classe `Bean` (pacote `model`)

A partir do componente `TTable`, recuperado durante a análise do arquivo .DFM, são populados os campos (colunas) da tabela corrente de acordo com o tipo definido. As colunas podem ser apenas dos tipos `String`, `Integer` e `Date`, pois os tipos definidos no formulário .DFM são `TStringField`, `TMemoField`, `TDateTimeField` e `TFloatField`. Assim, a representação dos tipos é feita da seguinte forma: caso seja encontrada uma coluna definida como `TStringField` ou `TMemoField`, seu correspondente em Java será String; caso seja

encontrada uma coluna definida como `TDateTimeField` seu correspondente em Java será `Date` e caso seja encontrada uma coluna definida como `TFloatField` seu correspondente em Java será definido como tipo `Integer`. A correspondência entre os tipos é feita através dos métodos da classe `Table`.

Como saída tem-se uma classe `Bean` com atributos correspondendo às colunas da tabela e métodos **`set`** e **`get`** para cada coluna. Também são gerados os métodos responsáveis pelos comandos de *insert*, *update* e *delete*.

O método responsável pelo comando *insert* estabelece uma conexão com o banco de dados, recuperada através do método **`retornaConexao`** da classe `ConnectionManager`, e monta a estrutura do comando *insert* com todas as colunas recuperadas da tabela em questão dentro do objeto `PreparedStatement` (quadro 22, trecho 01). Para cada coluna são definidos os seus valores (trecho 02), e então é acionado o comando *execute* (trecho 03), responsável pela execução. Caso aconteça algum erro, o comando *rollback* (trecho 04) desfaz a operação realizada. Caso o comando *execute* seja concluído com sucesso, o comando *commit* (trecho 05) grava as alterações no banco de dados.

O método *update* parte do mesmo princípio que o método *insert*. No entanto, no *update* não é possível montar o trecho da cláusula *where* do comando, devido ao fato de não ter como saber qual a chave primária da tabela corrente a partir da leitura do .DFM. Portanto, não é implementada a cláusula *where* do `Statement`. No método *delete*, assim como no método *update*, não é possível especificar a cláusula *where*.

```
...
public void insertIntoTBALUNO() throws Exception {
   boolean indErro  = false;
   String comando   = "";
   String sql       = "";
   String sqlValues = "";

   conn = new ConnectionManager().retornaConexao();                    01

   sql = "insert into TBALUNO (";
   sql += "MATRICULA, ";
   sql += "NOME, ";
   sql += "DESCRICAO, ";
   sql += "FLGREMETENTE, ";
   sql += "FLGESPECIAL, ";
   sql += "ID, ";
   sql += "IDADE, ";
   sql += "TURMA, ";
   sql += "STATUS ";
   sql += ")";
   ...
   sqlValues +=" values (";
   sqlValues += "?,";
   sqlValues += "?,";
   sqlValues += "?,";
   sqlValues += "?,";
   sqlValues += "?,";
   sqlValues += "?,";
   sqlValues += "?,";
   sqlValues += "?,";
   sqlValues += "?,";
   sqlValues+= ")";
   ...
   comando =  sql + sqlValues;
   PreparedStatement pst = conn.prepareStatement(comando);

   try{
        pst.setString(1,this.getMATRICULA());                          02
        pst.setString(2,this.getNOME());
        pst.setString(3,this.getDESCRICAO());
        pst.setString(4,this.getFLGREMETENTE());
        pst.setString(5,this.getFLGESPECIAL());
        pst.setString(6,this.getID());
        pst.setString(7,this.getIDADE());
        pst.setString(8,this.getTURMA());
        pst.setString(9,this.getSTATUS());

        pst.execute();   03

  } catch (SQLException e) {
        indErro = true;
        conn.rollback();   04
        e.printStackTrace();
  } catch (Exception exc) {
        indErro = true;
        conn.rollback();   04
        exc.printStackTrace();
  }
  if (!indErro){
        conn.commit();   05
        pst.close();}
}
...
```

Quadro 22 – Código-fonte do método *insert* da classe de `Bean` gerada pela ferramenta

3.6.3.4 Classe `Cursor` (pacote `model`)

A partir de cada componente `TTable` ou `TQuery` é gerada uma saída que representa a consulta SQL do componente de acesso a dados. No quadro 23 é apresentada a classe `Cursor`, gerada a partir de um objeto `TTable`, onde no trecho 01 é mostrado o objeto de conexão, criado a partir do método **`retornaConexao`**. Ainda no trecho 01 é apresentado o objeto `PreparedStatement` executando a consulta SQL montada. No caso dos componentes do tipo `TTable`, a consulta é montada a partir de um comando fixo, apenas concatenando o nome da tabela. No caso do componente `TQuery` é montada a mesma consulta SQL definida no componente lido. A partir da execução da consulta, é criado o objeto `ResultSet`, que armazena a consulta executada. No trecho 02 é montada uma estrutura de dados a partir de cada campo percorrido na tabela, facilitando assim a recuperação dos valores pelos componentes de visualização de dados. No quadro 24 é apresentado um trecho de um arquivo .DFM com a definição de um componente `TTable`.

```
public class CursorTBALUNO extends ConnectionManager{
    public ArrayList getDadosTBALUNO() throws SQLException, Exception{
        Connection conn = this.retornaConexao();                                    01
        String sql = "Select * from TBALUNO";
        PreparedStatement preparedStatement = conn.prepareStatement(sql);
        ResultSet       resultado = preparedStatement.executeQuery();

        HashMap    hmResultTBALUNO = new HashMap();
        ArrayList arrayResultTBALUNO = new ArrayList();
        int cont = 0;
        while (resultado.next()){                                                   02
           hmResultTBALUNO = new HashMap();
           hmResultTBALUNO.put("MATRICULA", resultado.getString("MATRICULA"));
           hmResultTBALUNO.put("NOME", resultado.getString("NOME"));
           hmResultTBALUNO.put("DESCRICAO", resultado.getString("DESCRICAO"));
           hmResultTBALUNO.put("FLGREMETENTE",resultado.getString("FLGREMETENTE"));
           hmResultTBALUNO.put("FLGESPECIAL", resultado.getString("FLGESPECIAL"));
           hmResultTBALUNO.put("ID", resultado.getString("ID"));
           hmResultTBALUNO.put("IDADE", resultado.getString("IDADE"));
           hmResultTBALUNO.put("TURMA", resultado.getString("TURMA"));
           hmResultTBALUNO.put("STATUS", resultado.getString("STATUS"));
           arrayResultTBALUNO.add(cont++, hmResultTBALUNO);
        }
        preparedStatement.close();
        return arrayResultTBALUNO;
    }
}
```

Quadro 23 – Código-fonte da classe `Cursor` gerada pela ferramenta

```
...
object TableTBAluno: TTable
    Active = True
    DatabaseName = 'BDoracle'
    TableName = 'TBALUNO'
    Left = 328
    Top = 32
end
...
```

Quadro 24 – Componente `TTable` definido num arquivo .DFM

3.6.3.5 Classe de manipulação dos eventos de banco de dados (pacote `controller`)

Cada componente de visualização de dados Delphi possui propriedades referentes as suas estruturas e seus componentes de acesso a dados, tais como as propriedades `datafield` e `datasource` que indicam, respectivamente, qual o nome da coluna da tabela que o componente irá referenciar e qual objeto de ligação com o banco de dados (`TDataSource`) está sendo utilizado. Com essas informações, é possível determinar a origem dos dados, que pode estar referenciando um componente do tipo `TTable` ou do tipo `TQuery`.

Diante deste cenário, é necessário carregar os componentes de visualização de dados de acordo com a origem dos dados, sendo que para cada objeto `Table` ou `Query` é criado um método na classe de eventos, responsável por atribuir os valores provenientes do banco de dados aos campos aos quais estão associados. No quadro 25 é apresentado um método para carregar os valores dos campos da tabela `TBALUNO` e associá-los aos componentes de visualização de dados (trecho 01).

```
public void carregaValoresCamposTBALUNO() throws SQLException, Exception {
  CursorTBALUNO  TBALUNOCursor = new CursorTBALUNO();
  ArrayList   arrDadosTBALUNO = new ArrayList();
  arrDadosTBALUNO  = TBALUNOCursor.getDadosTBALUNO();
  if (arrDadosTBALUNO.size() > 0){
    HashMap mapdados = (HashMap) arrDadosTBALUNO.get(0);
    screen.getDBTextMatricula().setText(mapdados.get("MATRICULA") == null ? "" :
                                        mapdados.get("MATRICULA").toString());   01
    screen.getDBEditNome().setText(mapdados.get("NOME") == null ? "" :
                                   mapdados.get("NOME").toString());
   ...
   }
}
```

Quadro 25 – Código-fonte do método **`carregaValoresCamposTBALUNO`** gerado pela ferramenta

Para que os componentes de visualização de dados do tipo `TDBComboBox` e `TDBGrid` possam apresentar os dados do banco de dados, é necessário um método específico para cada componente. Observa-se que ambos possuem associação com componentes de acesso a dados (`TQuery` ou `TTable`) e a partir desses dois componentes são geradas classes do tipo `Cursor`. Assim, são gerados métodos para criar os objetos `JComboBox` e `JTable`, a partir dos componentes `TDBComboBox` e `TDBGrid`, utilizam a classe `Cursor` com a qual possuem associação. Os quadros 26 e 27 mostram os métodos que criam os componentes `JComboBox` e `JTable`, respectivamente.

```
public JComboBox criaComboBoxDBComboBoxCidade() throws SQLException, Exception
  CursorQueryConsultaCidade QueryConsultaCidadeCursor
                                                = new   CursorQueryConsultaCidade();
  ArrayList arrQueryConsultaCidade = new ArrayList();
  arrQueryConsultaCidade = QueryConsultaCidadeCursor.getDadosQueryConsultaCidade();
  Object descricao[] = new Object[arrQueryConsultaCidade.size()];
  for (int i = 0; i < arrQueryConsultaCidade.size(); i++){
    HashMap hmDados = (HashMap)arrQueryConsultaCidade.get(i);
    descricao[i] = hmDados.get("NOME") == null ? null : hmDados.get("NOME");
  }
  screen.setDBComboBoxCidade(new JComboBox(descricao));
  screen.getDBComboBoxCidade().addItem("");
  screen.getDBComboBoxCidade().setSelectedIndex
                                (screen.getDBComboBoxCidade().getItemCount() - 1);
  ...
  return screen.getDBComboBoxCidade();
}
```

Quadro 26 – Código-fonte do método **`criaComboBoxDBComboBoxCidade`** gerado pela ferramenta

```
public JTable criaGridDBGridResultado() throws Exception{
  String[]      columnNames =  {"Código" ,"Idade" ,"Turma" ,"Status" };
  ArrayList     arrResultado = new ArrayList();
  CursorTBALUNO TBALUNOCursor = new CursorTBALUNO();
  arrResultado = TBALUNOCursor.getDadosTBALUNO();

  if (arrResultado != null){
    Object data[][] = new Object[arrResultado.size()][4];
    for (int i= 0; i<arrResultado.size();i++ ){
      HashMap mapdados = (HashMap) arrResultado.get(i);
      String ID = mapdados.get("ID") == null ? "" : mapdados.get("ID").toString();
      data[i][0] = ID;
      String IDADE = mapdados.get("IDADE") == null ? "" :
                     mapdados.get("IDADE").toString();
      data[i][1] = IDADE;
      ...
    }
    screen.setDBGridResultado(new JTable(data,columnNames));
  }else {
    screen.setDBGridResultado(new JTable());
  }
  ...
  return screen.getDBGridResultado();
}
```

Quadro 27 – Código-fonte do método **`criaGridDBGribResultado`** gerado pela ferramenta

3.6.4 Operacionalidade da implementação

Nesta seção é apresentada a operacionalidade da ferramenta. No Apêndice A encontra-se a documentação de Delphi2Java-II na forma de um arquivo `leiame.txt`. A ferramenta Delphi2Java-II é apresentada na figura 20.

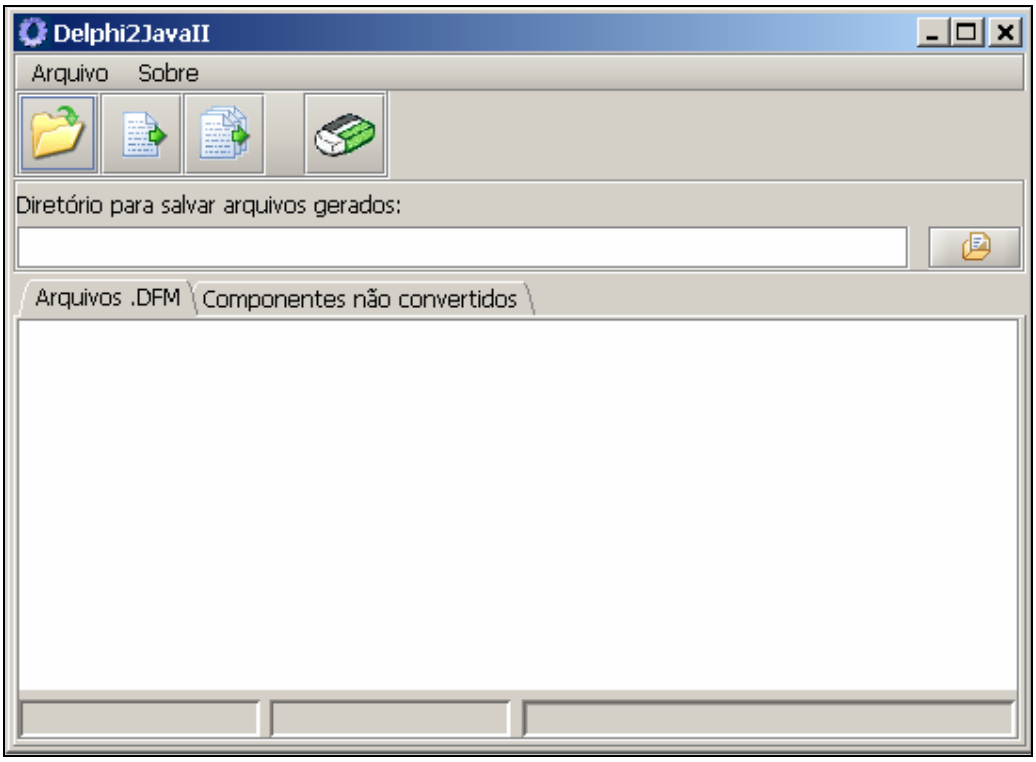


Figura 20 – Tela principal da ferramenta Delphi2Java-II

Delphi2Java-II possui opção para carregar os arquivos .DFM a serem convertidos, como apresentado na figura 21.

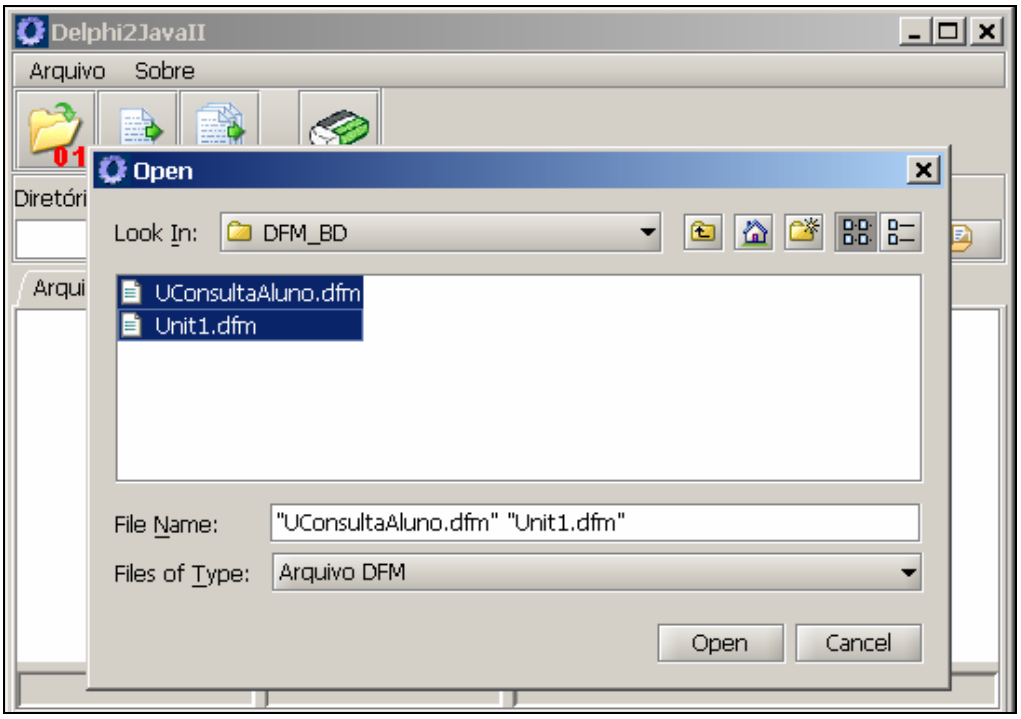


Figura 21 – Opção para carregar arquivo com extensão .DFM

Após carregar os arquivos .DFM desejados, é apresentada (em **Arquivos .DFM)** a lista de arquivos selecionados para a conversão, conforme figura 22.

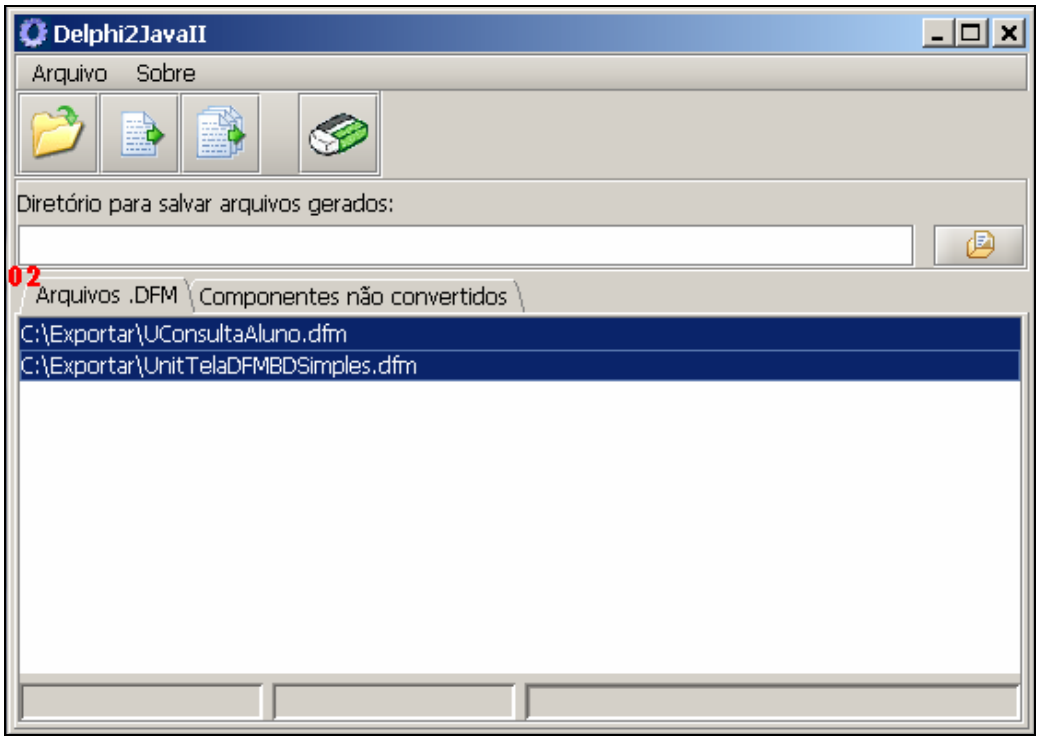


Figura 22 – Lista de arquivo(s) selecionado(s)

Deve-se escolher o diretório para salvar os arquivos que serão gerados pela ferramenta (figura 23).

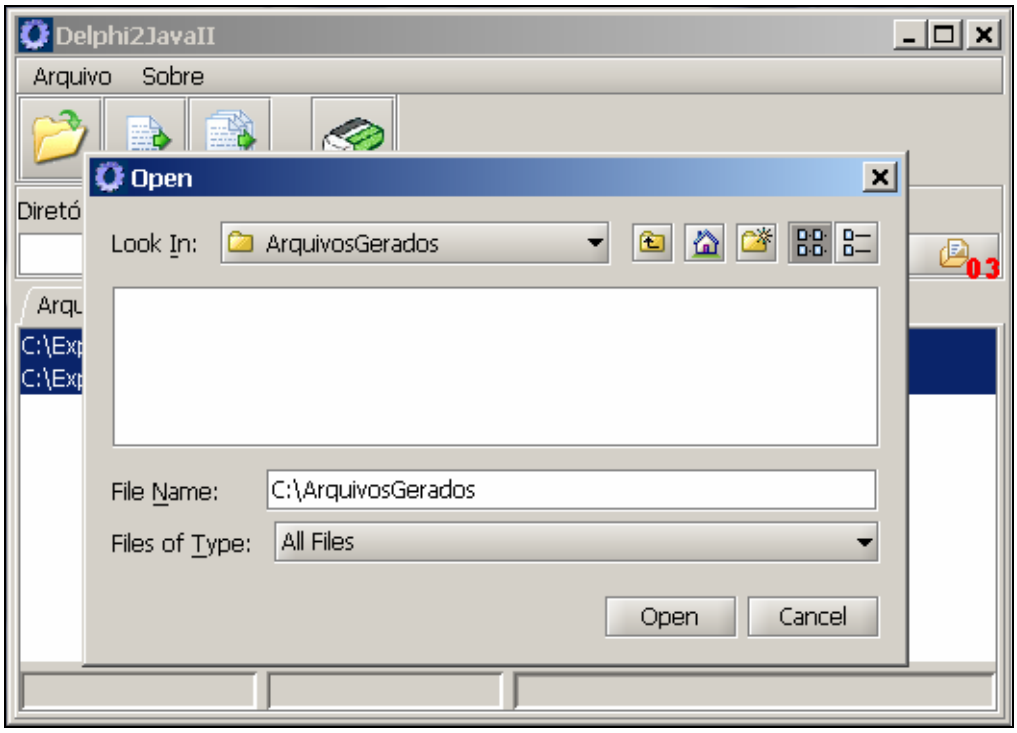


Figura 23 – Diretório para salvar os arquivos gerados

Uma vez selecionado, o caminho do diretório é mostrado (na figura 24).

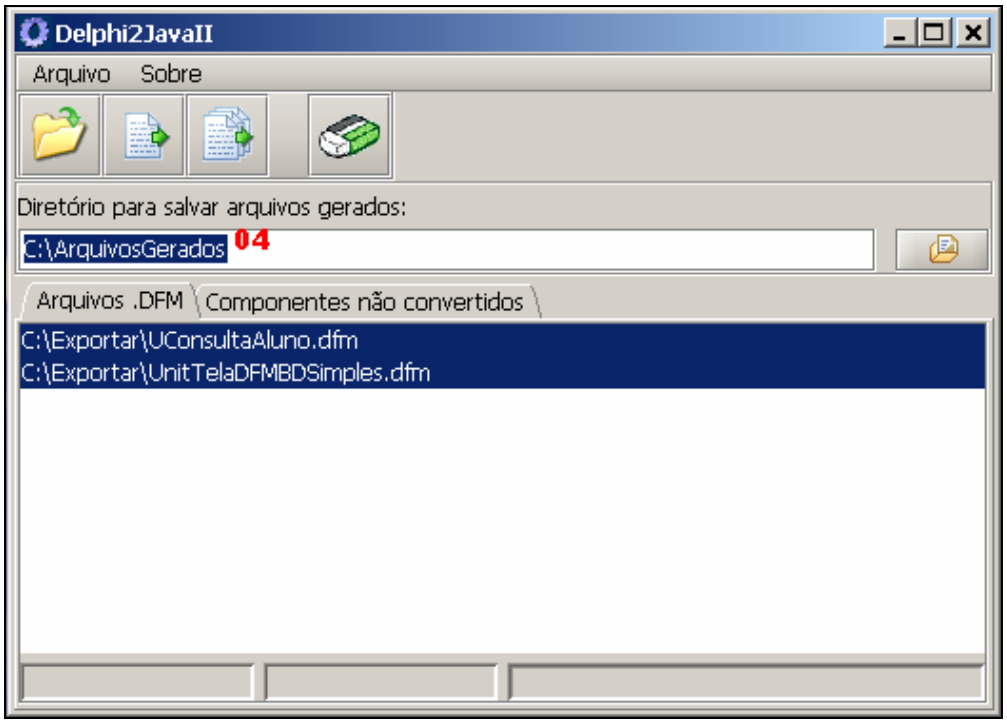


Figura 24 – Diretório selecionado

Para realizar a conversão do formulário existem duas opções (figura 25): gerar código **Para arquivo(s) selecionado(s)** e gerar código **Para todos os arquivos** carregados (em **Arquivos .DFM**).

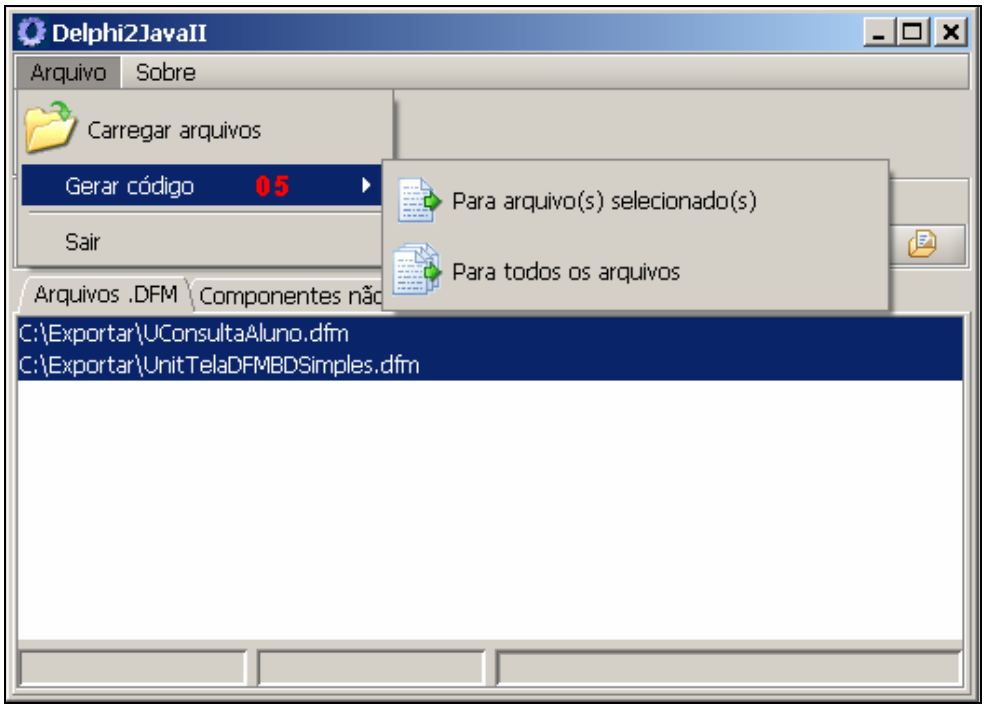


Figura 25 – Opções para gerar código

Os componentes que não são contemplados são listados em **Componentes não convertidos** (figura 26). Pode-se também limpar o conteúdo dos campos (opção 07).

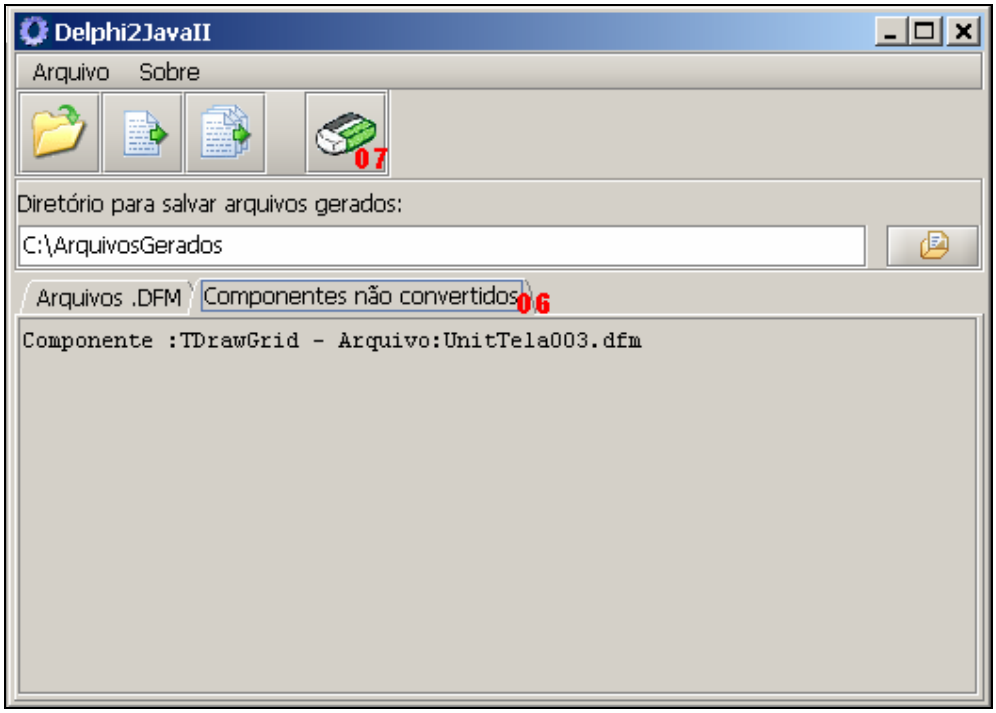


Figura 26 – Lista de componentes não convertidos

Na figura 27 são apresentadas informações Sobre a ferramenta.

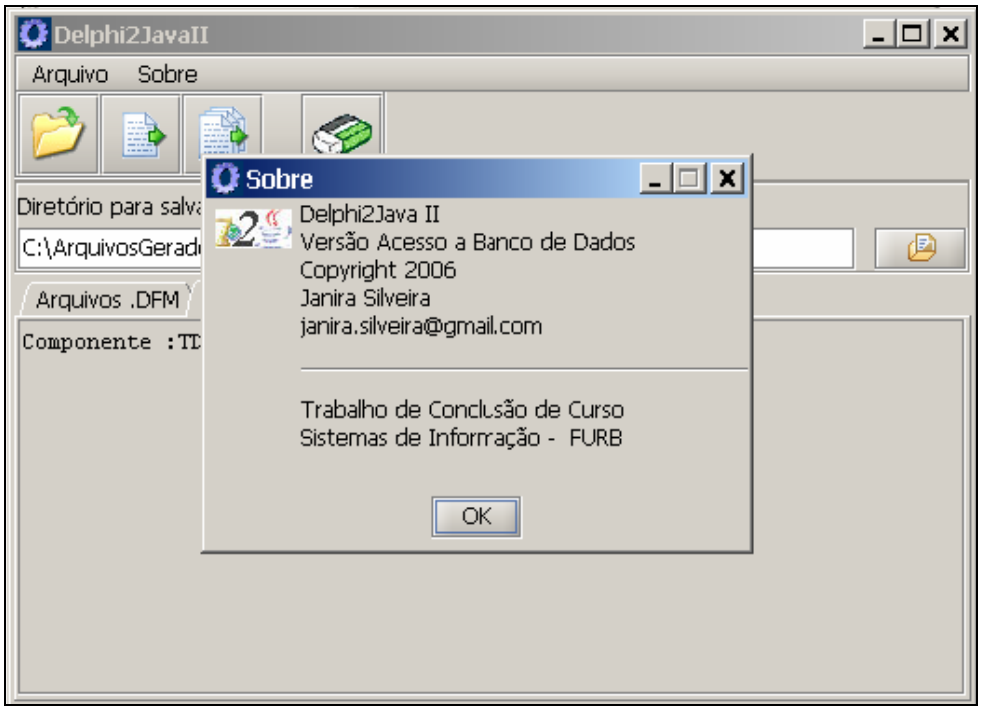


Figura 27 – Tela Sobre da ferramenta Delphi2Java-II

Por fim, a partir de um arquivo .DFM (figura 28) foi gerada a interface Swing da figura 29.

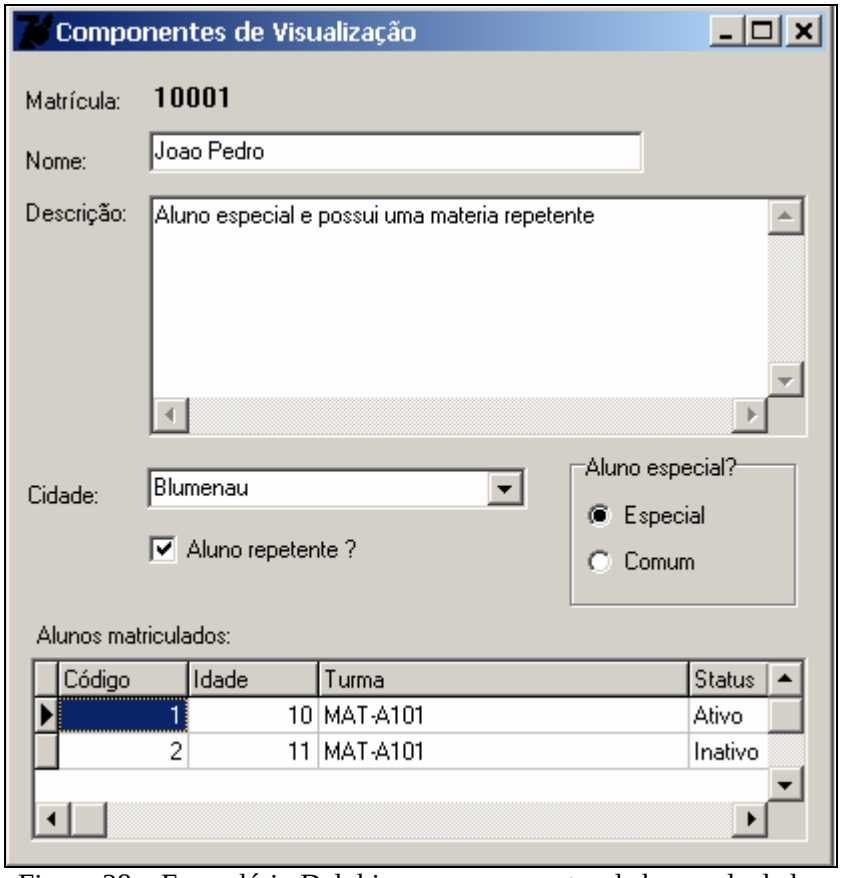


Figura 28 – Formulário Delphi com componentes de banco de dados

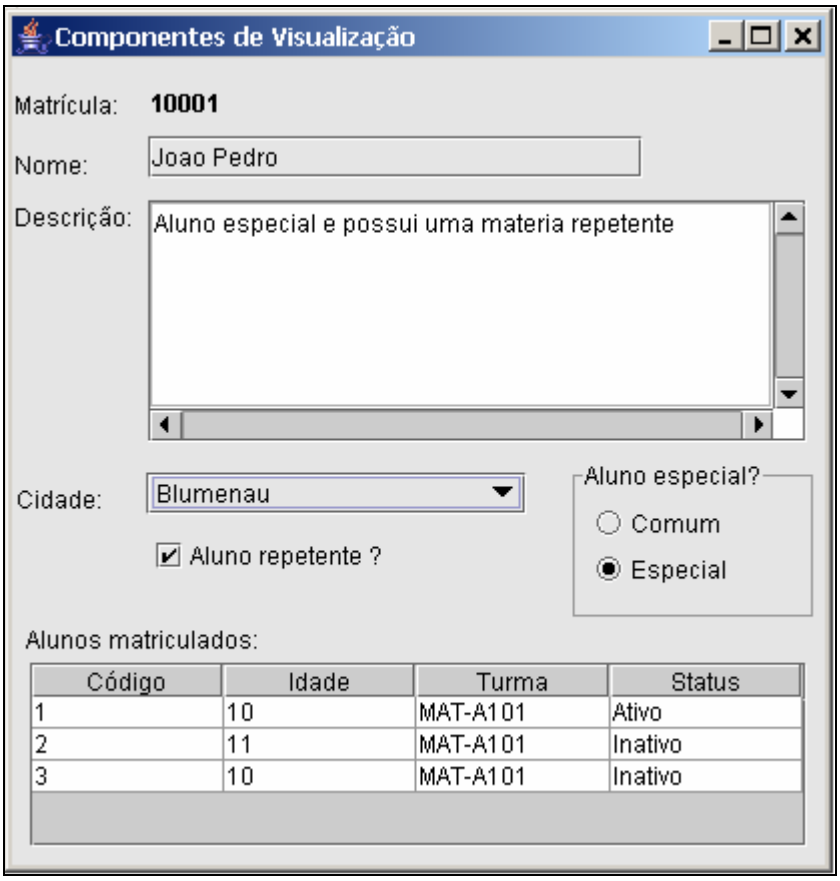


Figura 29 – Formulário Swing com componentes de banco de dados

3.7 RESULTADOS E DISCUSSÃO

A ferramenta apresentada atende aos requisitos propostos, seguindo o mesmo estilo de usabilidade das ferramentas Delphi2Java-II (FONSECA, 2005) e DelphiToWeb (SOUZA, 2005). As interfaces geradas pelas três ferramentas apresentam muita semelhança no que se refere à conversão dos componentes de visualização. No entanto, Delphi2Java-II (FONSECA, 2005) e DelphiToWeb (SOUZA, 2005) não convertem formulários que contenham componentes de visualização de dados e componentes de acesso a banco de dados.

Quanto aos eventos encontrados no arquivo .DFM, as duas versões de Delphi2Java-II geram código para manipulá-los. A primeira versão de Delphi2Java-II (FONSECA, 2005) disponibiliza uma classe de eventos contendo apenas as assinaturas dos métodos que estão habilitados na aplicação original em Delphi. A versão atual, além de gerar as assinaturas

desses métodos, inclui outros para carregar valores para os componentes de visualização de dados. Cabe ressaltar que, devido ao fato de apenas a versão de Delphi2Java-II desenvolvida nesse trabalho efetuar a geração de mais de dois arquivos de saída, é a única que padroniza e divide em camadas o código gerado, utilizando o padrão MVC.

No que se refere às técnicas utilizadas no desenvolvimento das ferramentas, DelphiToWeb lê e recupera as informações dos formulários Delphi a partir de analisadores léxico, sintático e semântico. Essa solução descrita em Souza (2005) foi reutilizada para implementar a ferramenta proposta nesse trabalho. Por fim, foram especificados *templates*, modelos que servem como guia para a geração de código de saída.

No quadro 27 é apresentada uma comparação entre as ferramentas levando em consideração suas principais características.

CARACTERÍSTICAS	**Delphi2Java-II (FONSECA, 2005)**	**DelphiToWeb**	**Delphi2Java-II**
conversão de componentes de visualização	27	22	22
conversão de componentes de visualização de dados	-	-	7
conversão de componentes de acesso a banco de dados	-	-	4
tratamento de eventos	sim	-	sim
geração de código de saída utilizando o padrão MVC	-	-	sim
uso de analisadores (léxico, sintático e semântico) para leitura do arquivo .DFM	-	sim	sim
uso de *templates* para geração de código	-	-	sim

Quadro 28 – Comparação entre ferramentas

4 CONCLUSÕES

Pôde-se constatar, através do desenvolvimento deste trabalho, que há uma polarização no desenvolvimento de aplicações baseadas nas tecnologias J2EE e .NET, onde a conversão de aplicações desenvolvidas em Delphi para estas plataformas demanda tempo e trabalho. Assim, a extensão da ferramenta Delphi2Java-II para suportar componentes de visualização de dados e componentes de acesso a banco de dados apresenta-se como uma opção para agilizar a migração de aplicações desenvolvidas em Delphi para aplicações Java.

No desenvolvimento desse trabalho tomou-se como base os trabalhos correlatos apresentados. A implementação de Delphi2Java-II (FONSECA, 2005) foi analisada para identificar como é realizada a conversão dos componentes de visualização contemplados. Também foi estudada a solução proposta em Souza (2005) para a análise de arquivos .DFM.

Assim, para leitura e interpretação dos arquivos .DFM foram utilizados os analisadores léxico, sintático e semântico, implementados na ferramenta DelphiToWeb (SOUZA, 2005). E para gerar a saída foram definidos *templates* que servem de modelo para todas as classes geradas pela ferramenta. A utilização de *templates* agilizou o desenvolvimento da ferramenta, resultando assim em um aumento na produtividade, na qualidade e na consistência do código gerado. Além disso, os *templates* podem ser facilmente validados e alterados.

A ferramenta também apresentou como solução a utilização do padrão de projeto MVC, facilitando a organização das classes geradas e visando melhorar a manipulação do código de saída.

No presente trabalho o recurso de acesso a banco de dados via JDBC mostrou-se de fácil manipulação, sendo utilizado para carregar os componentes de visualização de dados.

4.1 EXTENSÕES

Como extensões da ferramenta sugere-se: efetuar a conversão de novos componentes de acesso a banco de dados, entre eles o `TDBNavigator`; efetuar a análise e conversão da *unit* (arquivo .PAS) correspondente ao arquivo .DFM; e adicionar, na versão atual, a possibilidade de seleção do tipo de saída desejada em função dos *templates* definidos. Com essa opção, será possível gerar saída em outra linguagem, como HTML ou JavaScript, para os componentes contemplados, bastando definir o *template* que servirá de modelo para o código a ser gerado.

Existe também a possibilidade da geração de código utilizando outros *drivers* JDBC, uma vez que a API possui portabilidade suficiente para ser usada sobre qualquer banco de dados, como por exemplo, MySQL, bem como a possibilidade de efetuar a conversão para *driver* ODBC. Nesse caso, é necessário alterar apenas a camada de modelo gerada pela ferramenta, mais especificamente a classe `ConnectionManager`.

Devido ao fato das classes `Bean` geradas pela ferramenta não conterem os métodos **`update`** e **`delete`** definidos corretamente, já que o arquivo .DFM não possui todas as informações referentes a estrutura de uma tabela, tais como campos obrigatórios e chaves primárias, sugere-se a implementação de um módulo para recuperar, através do metadados, as informações necessárias para a montagem desses métodos.

REFERÊNCIAS BIBLIOGRÁFICAS

BALES, D. **Java programming with Oracle JDBC**. California: O'Reilly, 2002.

BODOFF, S. et al. **The J2EE tutorial**. California: Sun Microsystems Press, 2002.

CADE, M.; ROBERTS, S. **Enterprise architect for J2EE technology**. California: Sun Microsystems Press, 2002.

CESAR, R. Java X .NET: disputa acirrada no mercado nacional. **ComputerWorld**, São Paulo, n. 387, jun. 2003. Disponível em: <http://computerworld.uol.com.br/AdPortalv5/adCmsDocumentShow.aspx?DocumentID=75187>. Acesso em: 26 mar. 2005.

COELHO, L. Desenvolvimento de aplicações com acesso a banco de dados em Java. In: ENCONTRO DE CIÊNCIA E TECNOLOGIA, 3., 2004, Lages. **Anais...** Lages: Universidade do Planalto Catarinense, 2004. 200 f. Disponível em: <http://www.uniplac.net/ectec/>. Acesso em: 15 mar. 2005.

COMPILADOR. In: WIKIPÉDIA, a enciclopédia livre. [S.l.]: Wikimedia Foundation, 2006. Disponível em: <http://pt.wikipedia.org/wiki/Compilador>. Acesso em: 02 maio 2006.

DALGARNO, M. **Frequently asked questions about code generations**. [S.l.], 2006. Disponível em:<http://www.codegeneration.net/tiki-index.php?page=FrequentlyAskedQuestions>. Acesso em: 29 abr. 2006.

DEITEL, H. M.; DEITEL, P. J. **Java**: como programar. 4. ed. Tradução Carlos Arthur Lang Lisboa. Porto Alegre: Bookman, 2003.

DMSNET. **Migração / conversão de sistemas para Java (J2EE).** [S.l.], 2004. Disponível em: <http://www.dmsnet.com.br/conversor.htm>. Acesso em: 15 mar. 2005.

FONSECA, F. **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.

FUNDÃO DA COMPUTAÇÃO. **Design patterns fundamentais do J2EE**. [S.l.], 2004. Disponível em: <http://www.linhadecodigo.com.br/artigos.asp?id_ac=363>. Acesso em: 9 maio 2006.

GAMMA, E. et al. **Design patterns**: elements of reusable object-oriented software. Massachusetts: Addison-Wesley, 1994.

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

HANSEN , K. H. **StrutsTestCase**: the tool for Struts unit testing. [S.l.], [2006?]. Disponível em: <http://javaboutique.internet.com/tutorials/StrutsTestCase>. Acesso em: 11 jun. 2006.

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

HUSTED, T. et al. **Struts em ação**. Tradução Eveline Vieira Machado. São Paulo: Ciência Moderna, 2004.

JEVEAUX, P. C. M. **Aprenda a utilizar JDBC**. [S.l.], 2004. Disponível em: <http://portaljava.com/home/modules.php?name=Content&pa=list_pages_categories&cid=8> . Acesso em: 19 mar. 2005.

KASSEM, N. **Designing enterprise applications with the Java 2 Platform, Enterprise Edition**. California: Sun Microsystems Press, 2000.

MONTEIRO, J. M. **Desenvolvimento de aplicações em Java**. [S.l.], 2005. Disponível em: <http:// www.lia.ufc.br/eti/menu/modulos/POOJAVA/POOJAVA-aula8-JDBC.pdf>. Acesso em: 12 fev. 2006.

MOREIRA, D.; MRACK, M. Sistemas dinâmicos baseados em metamodelos. In: WORKSHOP DE COMPUTAÇÃO E GESTÃO DA INFORMAÇÃO, 2., 2003, Lajeado. **Anais eletrônicos...** [Lajeado]: UNIVATES, 2003. Disponível em: <http://www.univates.br/sicompi/wcompi2003/09-moreira-mrack.pdf>. Acesso em: 02 maio 2006.

MOURA, M. F.; CRUZ, S. A. B. **Formatação de dados usando a ferramenta Velocity**. Campinas, 2002. Disponível em: <http://www.cnptia.embrapa.br/modules/tinycontent3/index.php?id=2>. Acesso em: 9 maio 2006.

PERES, D. R. et al. TB-REPP: padrões de processo para a engenharia reversa baseado em transformações. In: LATIN AMERICAN CONFERENCE ON PATTERN LANGUAGES OF PROGRAMMING, 3rd, 2003, Porto de Galinhas. **Proceedings…** Recife: CIN/UFPe, 2003. Disponível em: <http://www.cin.ufpe.br/~sugarloafplop/final_articles/12_TB-REPP-Final.pdf>. Acesso em: 01 maio 2005.

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

ROBINSON, S. L. **Delphi to Java conversions**. [S.l.], [2006?]. Disponível em: <http://spritemaster.com/delphi_to_java_conversions.html>. Acesso em: 31 maio 2006.

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

SOARES FILHO, A. **Migrar software reduz custos e economiza tempo**. [S.l.], [2003]. Disponível em: <http://webinsider.uol.com.br/vernoticia.php/id/1844>. Acesso em: 30 abr. 2006.

SOMMERVILLE, I. **Engenharia de software**. 6. ed. Tradução André Maurício de Andrade. São Paulo: Addison Wesley, 2003.

SOUZA, A. **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.

STEIL, R. **Introdução ao Velocity**. [S.1.], [2006?]. Disponível em: <http://www.guj.com.br/java.artigo.18.1.guj>. Acesso em: 04 mar. 2006.

SUN MICROSYSTEMS. **Java technology**. [S.l.], 2006. Disponível em: <http://java.sun.com/>. Acesso em: 3 mar. 2006.

TAULLI, T.; JUNG, D. **VB2Java.COM**: moving from Visual Basic to Java. [S.l.], 1997. Disponível em: <http://www.vb2java.com/vb2java.html>. Acesso em: 01 maio 2005.

TRINDADE, C. **Java**: uma visão geral do pacote Swing. [S.l.], 2002. Disponível em: <http://www.imasters.com.br/artigo.php?cn=410&cc=21>. Acesso em: 26 mar. 2005.

ZEICHICK, A. **Java wars**: enterprise developers show loyalty. [S.l.], 2006. Disponível em: <http://www.sdtimes.com/article/story-20060201-07.html>. Acesso em: 8 maio 2006.

APÊNDICE A – LEIAME.TXT

DELPHI2JAVA-II

Objetivo:

Converter formulários Delphi (arquivos com a extensão DFM) em classes Java, utilizando a biblioteca Swing para *layout* de tela e fazendo acesso ao banco de dados através da API JDBC.

Como instalar e como executar:

Para instalar DELPHI2JAVA-II deve-se executar o arquivo **build.xml** que encontra-se no diretório raiz do projeto. Será gerado o arquivo **.JAR** da ferramenta no diretório **..\Delphi2JavaII\dist**, que poderá ser diretamente executado.

Pode ser criado um ícone de atalho, que possua como destino:

"C:\...\j2re1.4.2\bin\javaw.exe" -cp Delphi2JavaII.jar gui.FrmMain"

e como inicialização:

"C:\Documents and Settings\Administrador\workspace\Delphi2JavaII\dist"

Estrutura de arquivos e diretórios gerados:

Para as classes geradas deverá ser criado um projeto contendo os pacotes **model**, **view** e **controller**, seguindo o padrão MVC, sendo que o pacote **model** deverá conter os pacotes **bean** e **cursor**.

São geradas 5 classes que deverão ser dispostas em um dos pacotes citados anteriormente da seguinte forma:

a) classe xxxxxxFRM.java no pacote **view**: classe com os componentes de visualização (*layout* da interface), sendo **xxxxxx** o nome do arquivo DFM;
b) classe xxxxxxEvent.java no pacote **controller**: classe de eventos de interface, sendo **xxxxxx** o nome do arquivo DFM;
c) classe ConnectionManager.Java no pacote **model**: classe de conexão com o banco de dados;
d) classe BeanTByyyyyy.java no pacote **bean**: classe que representa as tabelas acessadas, sendo **yyyyyy** o nome da tabela. Podem ser geradas várias classes Bean, sendo um arquivo para cada tabela acessada;
e) classe CursorTByyyyyy.java no pacote **cursor**: classe que contém as consultas a serem executadas, sendo **yyyyyy** o nome do componente de acesso a dados. Podem ser geradas várias classes desse tipo, sendo um arquivo para cada componente de acesso a dados.

Deverão ser adicionadas, nas propriedades do projeto, as bibliotecas da API JDBC (classes12.jar, classes12dms.jar e nls_charset12.jar) para acesso ao banco de dados.

Estrutura de arquivos e diretórios da ferramenta:

A implementação da ferramenta está organizada em pacotes da seguinte maneira:

a) **compiler**: contém as classes utilizadas para análise do formulário Delphi;
b) **components**: contém as classes que representam os componentes de visualização a serem convertidos pela ferramenta, sendo que o pacote **db** contém os componentes de visualização de dados e acesso a dados a serem convertidos e o pacote **types** contém a classe para identificação dos tipos dos campos das tabelas;
c) **events**: contém a classe de eventos da ferramenta e a classe responsável pela análise dos *templates*;
d) **gui**: contém a tela principal e a tela de informações da ferramenta;
e) **icons**: contém os ícones utilizados;
f) **templates**: contém os *templates* utilizados para a geração da saída;
g) **util**: contém algumas classes com funcionalidades básicas, como arquivo de mensagens da ferramenta e classes para definição de cores e acentuação.

Informações sobre as ferramentas usadas no desenvolvimento:

DELPHI2JAVA-II foi implementada na linguagem Java, utilizando o JDK na versão 1.4.2. Foram usadas as seguintes ferramentas e bibliotecas:

a) para o desenvolvimento: Eclipse 3.0;
b) para manipulação de *templates*: biblioteca velocity-dep-1.3.jar;
c) para construção da interface: biblioteca Swing;
d) para elaboração do *layout* da interface: biblioteca looks-2.0.1.jar;
e) para acesso ao banco de dados: bibliotecas da API JDBC (classes12.jar, classes12dms.jar e nls_charset12.jar);

para efetuar os testes: ambiente de desenvolvimento Delphi 7.0 para elaboração dos arquivos .DFM e banco de dados Oracle8i para acesso ao banco de dados.

ANEXO A – Gramática do .DFM

No quadro abaixo é mostrada a gramática, escrita na notação BNF, que especifica a estrutura sintática de arquivos .DFM.

```
<dfm>::= <object>

<object>::= OBJECT identifier ":" <type>  <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 |  <object>  <objectList>
<propertyList>::= ε | <property> <propertyList>
<property>::=  <name>  "="  <value>
<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. 36).

Quadro 29 - Gramática