UNIVERSIDADE REGIONAL DE BLUMENAU

CENTRO DE CIÊNCIAS EXATAS E NATURAIS

CURSO DE CIÊNCIAS DA COMPUTAÇÃO – BACHARELADO

GERADOR DE CÓDIGO JAVA A PARTIR DE ARQUIVOS DO ORACLE FORMS 6I

CLAUDIO SCHVEPE

BLUMENAU
2006

2006/2-05

CLAUDIO SCHVEPE

GERADOR DE CÓDIGO JAVA A PARTIR DE ARQUIVOS DO ORACLE FORMS 6I

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

Prof. Joyce Martins, Mestre - Orientadora

BLUMENAU
2006

2006/2-05

GERADOR DE CÓDIGO JAVA A PARTIR DE ARQUIVOS DO ORACLE FORMS 6I

Por

CLAUDIO SCHVEPE

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

Presidente:	Prof. Joyce Martins, Mestre – Orientadora, FURB
Membro:	Prof. Marcel Hugo, Mestre – FURB
Membro:	Prof. Adilson Vahldick, Especialista – FURB

Blumenau, 20 de dezembro de 2006

Dedico este trabalho aos meus pais, irmãos e amigos que sempre acreditaram em mim e me deram forças para a realização do mesmo.

AGRADECIMENTOS

Aos meus pais e irmãos, pelo apoio recebido e por compreender minha ausência. Sem vocês não teria sido possível.

Aos meus amigos, pela motivação e compreensão de minha ausência.

À empresa HBTEC e seus colaboradores, em especial a Roger Zanone da Silva, com o qual aprendi muito nos dois anos em que trabalhei lá. Á empresa Precise e seus colaboradores, pelo apoio e pelas folgas concedidas quando necessário, em muito me ajudaram.

A Anderson Dallamico Duarte, pelo apoio na editoração gráfica de Imagens.

À minha orientadora, Joyce Martins, por ter acreditado na conclusão deste trabalho quando nem eu acreditava, pela sua amizade e dedicação em todo o desenvolvimento do trabalho.

RESUMO

Este trabalho apresenta o desenvolvimento da ferramenta Converte Forms, que é uma ferramenta para a migração de sistemas legados desenvolvidos em Oracle Forms para a linguagem de programação Java. O processo de conversão é baseado em *templates* interpretados pelo motor de *templates* eNITL, sendo que são convertidos componentes visuais, *triggers* e um subconjunto da linguagem PL/SQL. Para análise do código Oracle Forms foram gerados analisadores léxico e sintático a partir do Flex e do Bison, bem como utilizada a Forms API para a manipulação dos arquivos fontes. Ainda, foram utilizados os *frameworks* TinyXML, wxWidgets e Scintilla no desenvolvimento da ferramenta.

Palavras-chave: Tradutores. Reengenharia. Geradores de código. *Templates*.

ABSTRACT

This research presents the development of the Convert Forms tool that is a tool to the migration of legacy systems developed in Oracle Forms to the programming java language. The conversion process is based in templates interpreted by templates eNITL engine, The visual components, triggers and a subgroup of the PL/SQL language are converted. To the analyze of Oracle Form code was developed lexicon and syntactic analyzers from Flex and Bison Tools, as well as used the API Forms for the manipulation of sources archives. The frameworks TinyXML, wxWidgets and Scintilla have been used in the development of the tool.

Key-words: Translators. Reengineering. Code genarators. Templates.

LISTA DE ILUSTRAÇÕES

LISTA DE SIGLAS

API – *Application Programming Interface*

AWT – *Abstract Windowing Toolkit*

BNF – *Backus-Naur Form*

DLL – *Dynamically Linked Library*

GUI – *Graphical User Interface*

HTML – *HyperText Markup Language*

HTTP – *HyperText Transfer Protocol*

JDBC – *Java Data Base Connectivity*

JVM – *Java Virtual Machine*

J2EE – *Java 2 Enterprise Edition*

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

RAD – *Rapid Application Development*

RGB – *Red-Green-Blue*

RF – Requisitos Funcionais

RNF – Requisitos Não Funcionais

SGML – *Standard Generalized Markup Language*

SMTP – *Simple Mail Transfer Protocol*

SQL – *Structured Query Language*

TCC – Trabalho de Conclusão de Curso

XML – *eXtensible Markup Language*

SUMÁRIO

1 INTRODUÇÃO

Os sistemas legados são antigos sistemas de software necessários para o apoio nas mais variadas atividades executadas dentro de uma organização. Como estes sistemas são indispensáveis, as organizações precisam mantê-los em operação (SOMMERVILLE, 2003, p. 533). Usa-se a reengenharia de software para reimplementar os sistemas legados, visando facilitar sua manutenção e adaptação às tecnologias adotadas pela organização mantenedora do software. As duas principais vantagens da reengenharia em relação a abordagens mais radicais para a evolução de sistemas são: risco reduzido e custo reduzido. Mas, segundo Sommerville (2003, p. 537), somente é viável a migração de um sistema da linguagem em que foi originalmente implementado para uma outra linguagem se for possível automatizar parte do processo de tradução do código.

Para automatizar a etapa de tradução de código, por exemplo, constróem-se geradores de código. No entanto, o processo de desenvolvimento de um gerador de código é considerado uma tarefa complexa. Para diminuir a complexidade e agilizar o processo de desenvolvimento do gerador, pode-se implementar analisadores léxicos e sintáticos para a linguagem na qual o código fonte foi desenvolvido, bem como utilizar motores de *templates*[1].

Várias ferramentas desse tipo automatizam a migração de sistemas legados para plataformas distintas, incluindo a conversão de aplicações Oracle para Java. Com o Oracle Forms é possível implementar aplicações consistentes, robustas, seguras e escaláveis. No entanto, o desenvolvimento de aplicações Oracle envolve um custo elevado com as licenças para desenvolvimento/utilização e exige a contratação de pessoal especializado para dar suporte e manter tanto o software quanto o banco de dados.

Diante do exposto, este trabalho apresenta um gerador de código que permite automatizar o processo de migração de aplicações desenvolvidas em Oracle Forms 6i para aplicações *desktop* em Java. A entrada do gerador são arquivos fontes do Oracle Forms 6i, enquanto a saída, construída a partir de *templates*, são classes Java. Observa-se que os arquivos do Oracle Forms 6i estão em um padrão binário composto de *Forms Objects*[2] e de

---

[1] Motores de *templates* são mecanismos que permitem separar código dinâmico e o código estático, possibilitando criar moldes – *templates* – para o código a ser gerado. *Templates* são arquivos que contêm variáveis e blocos especiais, que são dinamicamente transformados a partir de dados enviados pelo motor de *templates*, gerando o código-alvo nos moldes do arquivo de especificação do *template* (ROCHA, 2005).

[2] *Forms Objects* são objetos de interface ao usuario uma aplicação Oracle Forms 6i.

código PL/SQL[3]. Desta forma, é necessário utilizar a biblioteca Forms API para a manipulação dos arquivos fontes.

Com a ferramenta proposta é possível automatizar parte do processo de migração de aplicações Oracle Forms para Java, permitindo que projetos que até então utilizam uma solução proprietária, possam partir para uma solução de desenvolvimento *open source*.

1.1 OBJETIVOS DO TRABALHO

O objetivo deste trabalho é desenvolver um gerador de código para migração de aplicações desenvolvidas em Oracle Forms 6i para Java.

Os objetivos específicos do trabalho são:

a) traduzir um subconjunto de código PL/SQL, incluindo apenas as construções procedurais;
b) permitir a conversão de alguns objetos visuais (`Canvas`, `CheckBox`, `Frame`, `Image`, `Line`, `ListItem`, `PushButton`, `RadioGroup`, `Tab`, `Text`, `TextItem`, `Window`);
c) mapear eventos (gatilhos ou *triggers*) de controle de objetos visuais para eventos Java;
d) utilizar *templates* para configurar o formato do código de saída.

1.2 ESTRUTURA DO TRABALHO

Este documento está estruturado em 4 capítulos. O próximo capítulo apresenta a fundamentação teórica, contextualizando os temas abordados no desenvolvimento do trabalho, tais como: tradução de código fonte, geradores de código, visão geral de Oracle Forms e Java, motores de *templates* e trabalhos correlatos. No terceiro capítulo é apresentado o desenvolvimento da ferramenta, contendo desde a especificação até a operacionalidade. O último capítulo traz os resultados alcançados com o desenvolvimento do trabalho bem como algumas propostas de extensões para a ferramenta.

[3] PL/SQL estende o SQL adicionando construções encontradas em linguagens procedurais, resultando em uma linguagem que é mais poderosa do que SQL. A unidade básica em PL/SQL é o bloco. Todos os programas são compostos por blocos, que podem estar aninhados uns dentro dos outros. Cada bloco efetua uma ação lógica.

2 FUNDAMENTAÇÃO TEÓRICA

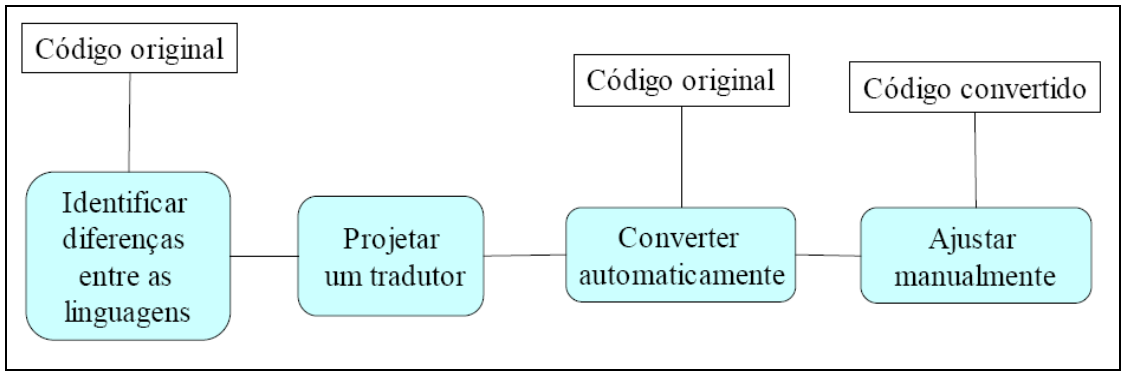
Neste capítulo é apresentada a fundamentação teórica necessária para o desenvolvimento da ferramenta. São abordados os motivos para traduzir sistemas legados para outras linguagens, bem como as etapas necessárias para construção de geradores para apoiar este processo da reengenharia de software e quais as vantagens de se utilizar deste tipo de ferramenta. É dada uma visão geral das características de Java por ser a linguagem alvo do processo de tradução. Do mesmo modo, é apresentado o Oracle Forms por ser a entrada da ferramenta. Na seqüência aborda–se sobre as etapas do processo de análise dos arquivos de entrada e discute-se o uso de motores de *templates* no processo de geração de código. Este capítulo trata também sobre a reestruturação de sistemas procedurais em sistemas orientados a objetos com base na análise do código fonte. E, para finalizar, são descritas algumas ferramentas existentes no mercado com funcionalidades similares às da ferramenta proposta.

2.1 TRADUÇÃO DE CÓDIGO FONTE

Sommerville (2003, p. 536) afirma que a tradução de código é um processo da reengenharia no qual o código fonte é traduzido automaticamente para outra linguagem. Neste processo, a estrutura e a organização do programa não são alteradas. A linguagem alvo pode ser uma versão mais atual da linguagem fonte ou pode ser outra linguagem totalmente diferente. Segundo Sommerville (2003, p. 536), os motivos que levam à tradução de um software para uma nova linguagem são:

a) atualização da plataforma de hardware ou software adotada pela empresa;
b) escassez de profissionais qualificados para manter sistemas desenvolvidos em linguagens incomuns ou que já tenham caído em desuso;
c) mudanças na política organizacional devido à padronização do ambiente de desenvolvimento para reduzir custos;
d) falta de suporte ao software por parte dos fornecedores.

A tradução de código só é economicamente viável se tiver um tradutor automatizado disponível para fazer grande parte da tradução. O tradutor pode ser um programa escrito especialmente para este fim ou pode ser uma ferramenta comprada para converter da linguagem fonte para a linguagem alvo. A figura 1 mostra o processo de tradução de código.



Fonte: Sommerville (2003, p. 537).

Figura 1 – Processo de tradução de código fonte

No processo de tradução é necessário observar as diferenças entre as linguagens envolvidas a fim de verificar quanto do código pode ser traduzido. Após definido o que será realmente traduzido com o apoio do gerador, parte-se para a etapa de especificação do tradutor.

Na grande maioria dos casos, é impossível a tradução completa por um processo automatizado. Isto é, pode não ser possível mapear algumas construções da linguagem fonte para construções da linguagem alvo. Nestes casos, é necessário fazer ajustes no código manualmente (SOMMERVILLE, 2003, p. 537).

2.2 GERADORES DE CÓDIGO

Geradores de código são basicamente programas que geram programas. Podem ser desde simples formatadores de código até ferramentas que geram aplicações complexas a partir de modelos abstratos (*templates*). De acordo com Herrington (2003, p. 3), entre as vantagens de se utilizar geradores de código para o desenvolvimento de software estão:

a) qualidade: código escrito manualmente tende a ter um nível de qualidade muito irregular visto que, durante o desenvolvimento da aplicação, podem ser propostas melhores abordagens para solucionar os problemas;
b) produtividade: quando são usados geradores de código, o volume de código digitado manualmente é bem menor se comparado ao volume de código escrito para aplicações desenvolvidas sem o uso dessas ferramentas. Sendo assim, tem-se mais tempo para outras etapas do projeto como, por exemplo, a fase de testes;
c) abstração: a definição de *templates* é bem mais simplificada que o código alvo. Com o uso de *templates* pode-se corrigir erros do projeto ou incluir novas

funcionalidades apenas reescrevendo os *templates*. Além disso, o gerador pode ser facilmente reprojetado para outras linguagens ou tecnologias.

No processo de construção de um gerador de código, a primeira coisa a fazer é verificar se o problema em que se propõe uma solução pode ser resolvido aplicando as técnicas de geração de código. Muitos arquivos de códigos ou códigos repetitivos são sinais de que a geração de código pode ser uma opção a ser usada no desenvolvimento.

Após verificar que a geração de código é a solução mais adequada para solucionar o problema, parte-se para a etapa de definição do processo de desenvolvimento. O processo de desenvolvimento do gerador, de acordo com Herrington (2003, p. 77), segue os passos abaixo:

a) escrever o código de saída manualmente: o primeiro passo na construção do gerador consiste em escrever manualmente o código alvo. Isso facilita a identificação do que é necessário extrair dos arquivos de entrada;
b) projetar o gerador: em seguida deve-se determinar a estrutura do gerador, indicando como a entrada será analisada, como a saída será gerada, quais as rotinas para manipulação de arquivos e diretórios;
c) analisar a entrada: nesta etapa constrói-se os analisadores léxico e sintático para analisar os arquivos de entrada e extrair as informações necessárias para gerar os arquivos de saída;
d) definir os *templates*: consiste em especificar o modelo do código de saída implementando no primeiro passo;
e) gerar a saída usando *templates*: a última etapa do processo de construção do gerador consiste em implementar a geração do código alvo de acordo com o modelo especificado nos *templates*.

2.3 CÓDIGO DE SAÍDA EM JAVA

Java foi desenvolvida pela Sun Microsystems por volta de 1990, pouco antes da explosão da Internet. É uma linguagem de programação orientada a objeto com sintaxe similar à da linguagem C++ (DEITEL; DEITEL, 2003, p. 59).

Um programa fonte escrito em linguagem Java (arquivo com extensão `JAVA`) é traduzido pelo compilador para os *bytecodes* (arquivos com extensão `CLASS`), ou seja, código de máquina de um processador virtual, a JVM. A JVM é um programa capaz de interpretar os *bytecodes* produzidos pelo compilador. A vantagem desta técnica é garantir uma maior

portabilidade para os programas Java.

Na programação das interfaces gráficas das aplicações, podem ser usadas as bibliotecas AWT e Swing (DEITEL; DEITEL, 2003, p.605). Os componentes originais do pacote AWT estão diretamente associados com recursos da interface gráfica da plataforma onde a aplicação é executada. Assim, quando um programa Java com componentes AWT é executado, os componentes são exibidos com aparência diferente em cada plataforma. Estes componentes são chamados de peso-pesado. Cada componente peso-pesado tem um *peer* que é responsável pelas interações entre o componente e a plataforma em questão.

Por outro lado, a maior parte dos componentes Swing são escritos, manipulados e exibidos completamente em Java. Assim sendo, podem ser construídas interfaces com aparência e comportamento padrão em todas as plataformas. Esses componentes são conhecidos como peso-leve. Além disso, apesar da maioria dos componentes Swing possuir aparência e comportamento padrão, é possível fornecer uma aparência e um comportamento particular para cada plataforma onde a aplicação será executada (DEITEL; DEITEL, 2003, p. 605).

As interações dos usuários com os componentes de interface geram eventos. Segundo Deitel e Deitel (2003, p. 611), o mecanismo de tratamento de eventos em Java é formado por três partes:

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

Para um programa estar preparado para processar eventos, precisa realizar duas tarefas: registrar o *listener* para cada componente de interface e implementar um método de tratamento de evento. Cada *listener* implementa uma ou mais interfaces *listeners* dos pacotes `java.awt.event e javax.swing.event` (DEITEL; DEITEL, 2003, p. 611).

2.4 CÓDIGO DE ENTRADA EM ORACLE FORMS

Segundo Fernandes (2002, p. 670), Oracle Forms Developer (ou Oracle Forms) é um ambiente de produtividade RAD, capaz de construir rapidamente aplicações a partir das

definições do banco de dados. As aplicações, compostas de *Forms Objects* e de código PL/SQL, podem ser implementadas para arquiteturas cliente-servidor, de três camadas ou para web.

A ferramenta constrói três tipos de módulos (figura 2):

a) `Forms` (arquivo com extensão FMB): é o formulário da aplicação, composto de itens de controle e blocos PL/SQL, sendo seus principais componentes `Triggers`, `Attached Libraries`, `Data Blocks`, `Canvases`, `Program Units` e `Windows`;

b) `Menus` (arquivo com extensão MMB): consiste de um conjunto de sub-menus e código para acionamento dos diversos módulos, `Forms` ou sub-menus. É composto de `Attached Libraries`, `Menus`, `Submenus` e `Program Units`;

c) `PL/SQL Libraries` (arquivo com extensão PLL): são programas PL/SQL que podem ser compartilhados por toda a aplicação. É composto basicamente de `Attached Libraries` e `Program Units`.

Segundo Oracle Corporation (2000, p. 4), Oracle Forms inclui uma API, a Forms API, que possibilita que programas desenvolvidos utilizando a linguagem C possam criar, carregar, editar e salvar os arquivos com extensão FMB, MMB e PLL. Nesta API encontram-se todas as funcionalidades para acessar as propriedades dos objetos do Oracle Forms.

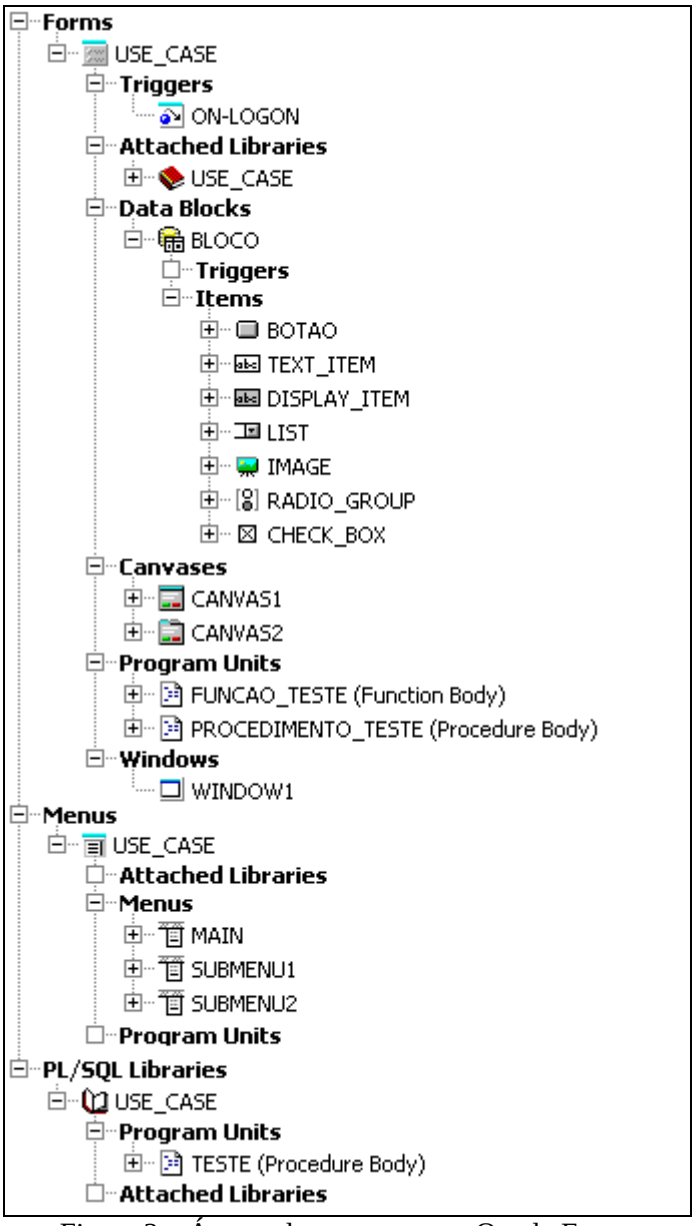


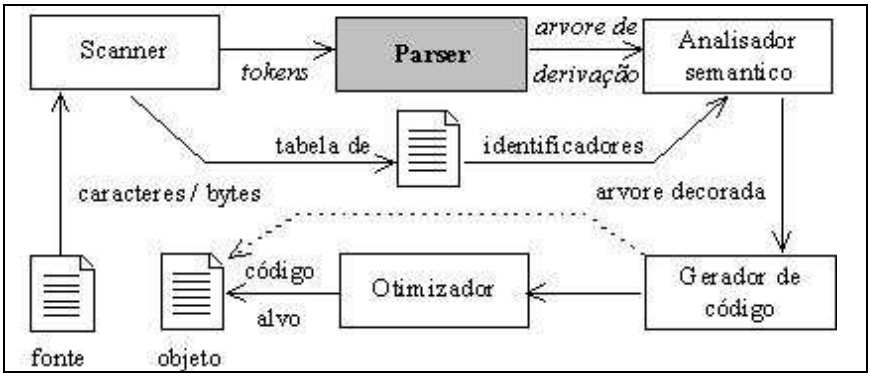
Figura 2 – Árvore de componentes Oracle Forms

2.5 ANÁLISE DA ENTRADA

Para processar a entrada de um gerador de código, podem ser usados analisadores de linguagens. Os analisadores (léxico e sintático) são módulos que compõem um compilador. Um compilador é basicamente um tradutor de linguagens de programação, ou seja, é um sistema que aceita como entrada um programa escrito em uma linguagem de programação (linguagem fonte) e produz como resultado um programa equivalente em outra linguagem (linguagem alvo) (AHO; SETHI; ULLMAN, 1995, p. 01).

Um compilador está dividido em etapas, cada uma das quais transformando o programa

fonte de uma representação para outra. Essas etapas estão representadas na figura 3.



Fonte: Rezende (1998).

Figura 3 – Etapas de um compilador

O analisador léxico ou *scanner* tem como objetivo identificar seqüências de caracteres que formam as unidades léxicas (*tokens*). O analisador léxico lê o programa fonte da esquerda para a direita, caractere a caractere, verificando se os caracteres lidos fazem parte da linguagem, identificando os *tokens* e enviado-os para o analisador sintático.

O analisador sintático ou *parser* recebe os *tokens* do analisador léxico e agrupa-os em frases gramaticais, que são usadas pelo compilador para sintetizar a saída. As frases gramaticais do programa fonte são representadas por uma árvore gramatical.

O analisador semântico verifica o significado da estrutura hierárquica determinada pela fase de análise sintática. Faz todo o processo de verificação de tipos e de consistência entre declaração e uso de identificadores.

Após as análises léxica, sintática e semântica, alguns compiladores geram uma representação intermediária, como um programa para uma máquina abstrata. Essa representação tem o intuito de facilitar a geração e a otimização do programa alvo.

A fase final de um compilador é a geração do código, que normalmente é código de máquina, mas, como dito anteriormente, pode ser uma representação em outra linguagem de programação.

2.6 MOTOR DE *TEMPLATES*

Os motores de *templates*, segundo Rocha (2005), são mecanismos que permitem desenvolver geradores de código independentes do código alvo já que ele está externo à aplicação. Nesse caso, deve existir um programa responsável por instanciar o motor, carregar os valores das variáveis e blocos especiais do *template* e apresentar o resultado da união do

código dinâmico[4] com o código estático[5].

Cada motor de *templates* implementa sua própria linguagem através da qual os *templates* deverão ser escritos. Esta linguagem define os tipos de blocos especiais e como referenciar variáveis nos *templates*. Varia bastante em nível de complexidade, podendo ser definida apenas com estruturas básicas para substituição de valores de variáveis ou até ter estruturas de controle mais complexas como *loops* e comandos condicionais (ROCHA, 2005).

Motores de *templates*, tais como eNITL (JOU, 2000), Velocity (APACHE SOFTWARE FOUNDATION, 2005) e TinyButStrong (SKROL, 2006), podem ser utilizados em diversas áreas da computação como, por exemplo, geradores de interfaces web, geradores de aplicações e ferramentas de documentação de código fonte. Nesse trabalho optou-se por utilizar o motor de *templates* eNITL.

2.6.1 Motor de *templates* eNITL

eNITL é um motor de *templates open source* desenvolvido para ser utilizado em aplicações C++ que requerem a geração de resultados flexíveis. É especialmente útil às aplicações que empregam HTML, HTTP, SMTP, SGML, XML e outros formatos de dados baseados em texto (BRECK, 1999).

Os *templates* do eNITL são compostos por texto literal (código estático) e declarações eNITL (código dinâmico). A primeira declaração de um *template* especifica seu nome[6]. As declarações sempre se apresentam entre os símbolos `<#` e `#>`, sendo que declarações múltiplas devem ser separadas por ponto e vírgula (`<#env.command1; env.command2#>`). Um *template* termina com a declaração `<#endtmpl#>`. Ainda, um *template* pode conter qualquer número de seções de texto literal e declarações, em qualquer agrupamento.

Através das declarações, também é possível ter acesso a atributos e métodos definidos pela aplicação que instanciar o motor de *templates*. Assim, durante a execução, a aplicação deverá prover os valores para as declarações especificadas no *template*. Essas declarações são representadas pelo comando `env`.

---

[4] Código dinâmico é o código definido no *template* de forma simbólica, que passará por tranformações no processo de geração do código de saída. Dessa forma, símbolos como variáveis ou macros serão substituídos por informações enviadas ao motor de *templates.*

[5] Código estático é o código definido no *template* de forma literal que não será alterado pelo motor de *templates*.

[6] O nome é um identificador composto por uma seqüência de caracteres formada por letras, números e *underline*, que não começa com número.

Segundo Breck (1999), eNITL provê algumas operações e estruturas de controle para a elaboração do código dinâmico dos *templates*. No quadro 1 são mostrados os elementos mais usados e no quadro 2 tem-se um exemplo de *template* eNITL produzido para esse TCC.

ELEMENTO	DESCRIÇÃO	
atribuição	`:=`	atribui um valor a uma variável
incremento	`++`	incrementa o valor de uma variável em 1, sendo que o operador pode aparecer à esquerda ou à direita da variável
decremento	`--`	decrementa o valor de uma variável em 1, sendo que o operador pode aparecer à esquerda ou à direita da variável
concatenação	`$`	concatena o valor de variáveis ou o valor resultante de comandos `env`
operadores aritméticos	`+` (soma); `-` (subtração); `*` (multiplicação); `/` (divisão); `%` (módulo)	
operadores relacionais	`=` (igual); `!=` (diferente); `<` (menor); `>` (maior); `=<` (menor-igual); `=>` (maior-igual)	
operadores lógicos	`||` (ou); `&&` (e)	
comandos de controle	`if-elseif-else-endif`	comando condicional
	`while-endwhile`	comando de repetição
	`setOutput-endSetOutput`	comando para selecionar onde a saída será gerada (*e-mail*, arquivo, etc.)
	`execute`	comando para incluir outros *templates*

Quadro 1 – Elementos da linguagem de *templates* do eNITL

```
<#·MainModule·#>
<#·setoutput·file("MD"$env.module.name·$·".java")·#>
package·view;
import·java.awt.event.*;
<#if·env.module.init_mnu·!=""#>
import·view.MB<#env.upper(env.module.init_mnu)#>;
<#endif#>

·public·class·MD<#env.module.name#>·extends·javax.swing.JFrame
·{··//Construtor··default···
····public·MD<#env.module.name#>()
····{
·······theDesktop·=·new·JDesktopPane();
<#if·env.module.init_mnu·!=""#>
·······MB<#env.upper(env.module.init_mnu)#>·bar=·new·MB<#env.upper(env.module.init_mnu)#>();
·······setJMenuBar(bar);
<#endif#>·
·······getContentPane().add(theDesktop);

<#·····win:=·env.windows#>
<#·····while·win·!=·0·,·win:=·env.object(win).next#>
<#·········tmpl.CreateWin(win)#>
<#·····endwhile#>
<#·····tmpl.CreateBlocks#>

····}

·public·static·void·main(String[]·args)
·{
····MD<#env.module.name#>·application·=·new·MD<#env.module.name#>();
····application.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

·}

}
<#·endsetoutput·#>
<#·tmpl.CreateEvents("MD"$env.module.name)#>

<#endtmpl#>
```

Quadro 2 – Exemplo de *template* eNITL

Breck (1999) afirma que a máquina do eNITL tem três componentes primários usados no processamento dos *templates*: o *tokenizer*, o compilador e o *runtime*. O *tokenizer* é

responsável por varrer as definições dos *templates* a fim de encontrar os *tokens* da linguagem e enviá-los para o compilador. O compilador, por sua vez, é um *parser* cujas rotinas implementam a gramática de linguagem de *templates*. Traduz a lista de *tokens* em uma árvore binária de objetos que representam suas operações. O *runtime* percorre a árvore binária gerada pelo compilador, executando cada operação definida na árvore, administrando a alocação de memória necessária para a execução dos *templates* e mantendo variáveis, parâmetros e controles de execução em uma estrutura de dados do tipo pilha.

Se houver um erro de programação em um *template*, o eNITL não executa qualquer parte do *template* e imprime uma mensagem de erro em tempo de compilação em um fluxo (*stream*) próprio para a notificação de erros. No entanto, alguns erros só poderão ser detectados durante a execução do *template*. Estes serão diagnosticados juntamente com a saída que está sendo gerada.

A utilização do motor de *templates* eNITL consiste em derivar uma classe de `_Template::Access`. Esta classe abstrata define a interface que o eNITL usa para processar o código dinâmico bem como para ter acesso a atributos e métodos da aplicação que fornecerão os valores para as declarações representadas através dos comandos `env`. Devem ser implementados três métodos virtuais na classe derivada: `output`, `value` e `member`.

O método output é chamado para gerar a saída, escrevendo tanto o código estático quanto o valor do código dinâmico calculado pelo método value.

O método value é chamado quando o eNITL encontra um código dinâmico que precisa ser calculado antes de ser escrito no dispositivo de saída. A implementação deste método deverá devolver o valor apropriado na estrutura `_Template::Var`.

O método member é chamado quando o eNITL encontra um código dinâmico na forma `env.member[.member...].command`. Deverá devolver um ponteiro de um objeto derivado de `_Template::Access` ou nulo. Este ponteiro é usado para controlar a execução do próximo código dinâmico.

2.7 REESTRUTURAÇÃO DE SISTEMAS PROCEDURAIS EM ORIENTADOS A OBJETOS

Na reestruturação de sistemas procedurais em sistemas orientados a objetos podem ser utilizadas três abordagens (KULANDAISAMY; NAGARAJ; THONSE, 2002, p.1). A primeira envolve: a análise léxica e sintática da aplicação procedural; a elaboração de

diagramas a partir das informações extraídas da aplicação; e o mapeamento das estruturas de dados, das funções e dos procedimentos com implementação manual na linguagem orientada a objeto.

A segunda abordagem envolve o uso de técnicas de tradução de linguagens e de construção de compiladores. Ferramentas desenvolvidas conforme essa abordagem também usam a gramática da linguagem fonte para identificar e extrair da aplicação procedural as informações necessárias. Na seqüência, são aplicadas regras de tradução para gerar automaticamente a aplicação orientada a objetos.

A terceira abordagem combina as duas abordagens anteriores, através de um conjunto de modelos e etapas de tradução, incluindo: organização do código legado, recuperação do projeto, (re)projeto e (re)implementação (KLÖSCH, 1996). Klösch (1996) afirma que no processo de reestruturação faz-se necessária a integração de um perito humano conhecedor do domínio da aplicação a fim de adicionar novos requisitos e resolver possíveis conflitos que possam surgir ao longo do processo.

Embora a segunda abordagem apresente limitações, nesse trabalho optou-se por utilizá-la para a implementação da ferramenta proposta. Nesse caso, o processo de reestruturação consiste basicamente de três passos: identificação das construções sintáticas do código procedural, elaboração da regras de mapeamento, geração automática do código orientado a objetos.

2.7.1 Identificação das construções sintáticas

Aplicações desenvolvidas em linguagens procedurais são compostas, segundo Kulandaisamy, Nagaraj e Thonse (2002, p. 1), por um programa principal e um conjunto de funções e procedimentos. A persistência de dados é gerenciada através de comandos SQL ou operações sobre arquivos e, freqüentemente, não existe uma separação entre interface com o usuário e a lógica de processamento. Assim, para identificação das construções sintáticas deve-se usar analisadores léxico e sintático para a linguagem procedural.

2.7.2 Elaboração das regras de mapeamento

Nessa etapa, pode-se usar algumas regras de mapeamento propostas por Peres et al

(2003), apresentadas no quadro 3. Para tanto deve-se identificar: os arquivos ou tabelas de banco de dados; as estruturas de dados; as interfaces de interação com o ambiente externo; as unidades de programa (funções, procedimentos, blocos de comandos).

CÓDIGO PROCEDURAL	CÓDIGO ORIENTADO A OBJETOS
arquivo ou tabela de banco de dados e seus campos	classe de persistência com respectivos atributos
estrutura de dados e seus campos	classe com respectivos atributos
interface de interação e seus componentes	classe de interface com respectivos atributos, um para cada componente
unidade de programa que faz referência a um arquivo ou a uma tabela de banco de dados	método da classe correspondente ao arquivo ou à tabela de banco de dados
unidade de programa associada a um componente de interface	método da classe de interface correspondente

Quadro 3 – Mapeamento entre elementos procedurais e orientados a objetos

Peres et al (2003) salienta que nessa etapa o engenheiro de software deve validar as identificações realizadas e os respectivos mapeamentos.

2.7.3 Geração automática do código orientado a objeto

Uma vez realizado o mapeamento entre as construções sintáticas do código procedural e os elementos do código orientado a objetos, deve-se gerar automaticamente o código de saída. Em função das diferenças entre os paradigmas (procedural e orientado a objeto) pode não ser possível mapear todos os elementos de um modelo de aplicação para o outro. Esses elementos do programa procedural são chamados de “resto procedural” do sistema analisado e são reestruturados manualmente para uso orientado a objeto (KLÖSCH, 1996).

2.8 TRABALHOS CORRELATOS

Existem várias ferramentas, comerciais ou não, que oferecem suporte para migração de sistemas entre plataformas distintas, tais como Delphi2Java-II (FONSECA, 2005), VB.Net to C# Converter (VBCONVERSIONS, 2006) e VB2Java.COM (TAULLI; JUNG, 1997). Especificamente sobre migração de aplicações Oracle, pode-se citar: Forms2Net (ATX SOFTWARE, 2006), SwisSQL (ADVENTNET, 2006) e Exodus (CIPHERSOPT INC, 2006).

Com a ferramenta Forms2Net é possível migrar aplicações de Oracle Forms para aplicações equivalentes em Microsoft ASP.NET. Segundo ATX Software (2006), os motivos

que podem levar a utilizar Forms2Net são:

a) padronizar a plataforma de desenvolvimento;
b) aumentar a produtividade;
c) minimizar tempo, custo e risco do processo de migração;
d) facilitar a manutenção da aplicação desenvolvida;
e) preservar a semântica (funcionalidade) do sistema original.

Com Forms2Net é possível continuar usando um banco de dados da Oracle ou migrar para SQL Server. A conectividade com a camada de dados está baseada em componentes de acesso a dados através da tecnologia de Microsoft ADO.NET, que permite ao cliente trabalhar com bancos de dados diferentes de um modo independente. Observa-se que a migração do *schema* do banco de dados e dos próprios dados não é feita pelo Forms2Net, sendo necessário, nesse caso, o uso de outras soluções complementares. ATX Software (2006) afirma ainda que no processo de tradução, de 75% a 95% do código final são gerados automaticamente pelo *wizard* do Forms2Net - o valor exato depende da complexidade e da estrutura da aplicação original.

Conforme AdventNet (2006), SwisSQL é uma ferramenta para ajudar a converter para a linguagem Java procedimentos PL/SQL armazenados no Sistema Gerenciador de Banco de Dados (SGDB). Suas principais características são:

a) suporte a todas as construções PL/SQL, bem como pacotes, procedimentos e funções;
b) padrão de desenvolvimento baseado em API *open source*, isto é, as funções utilizadas no código gerado são amplamente difundidas e de uso gratuito;
c) código gerado é facilmente portável para outras arquiteturas.

Segundo CipherSoft Inc (2006), Exodus é uma ferramenta de conversão de Oracle Forms e PL/SQL para Java, possuindo as seguintes características:

a) migração de aplicações Oracle Forms e PL/SQL para J2EE, sendo que o código convertido é independente de plataforma e banco de dados;
b) conversão de procedimentos, funções e `PL/SQL Library`, automatizando todo o processo de migração;
c) produção de código nativo Java/XML, permitindo aplicar manutenções em todo o código gerado, além de manter o conteúdo semântico e os padrões de codificação do código convertido;
d) arquitetura *multi-tier*;
e) suporte a conexão com banco de dados via JDBC.

3 DESENVOLVIMENTO DO TRABALHO

Este capítulo apresenta o desenvolvimento da ferramenta Converte Forms, realizado através das seguintes etapas:

a) elicitação dos requisitos: detalhamento e reavaliação dos requisitos previamente definidos;
b) análise das linguagens PL/SQL e Java: identificação das diferenças entre as linguagens fonte e destino, elaborando as regras de mapeamento entre as construções PL/SQL e as construções Java;
c) especificação da saída: definição das classes Java bem como da estrutura de classes que serão geradas;
d) análise da entrada: uso da biblioteca Forms API para auxiliar na leitura dos arquivos fontes do Oracle Forms 6i e de analisadores léxico e sintático para analisar o subconjunto da linguagem PL/SQL a ser traduzido. O subconjunto em questão foi definido usando expressões regulares e notação BNF para especificar, respectivamente, os símbolos léxicos e as construções sintáticas;
e) especificação: especificação da ferramenta com análise orientada a objeto utilizando a UML;
f) implementação: a GUI foi implementada usando Microsoft Visual C++ 6.0, com o apoio da biblioteca wxWidgets-2.6.2; a biblioteca Forms API foi usada para auxiliar na leitura dos arquivos fontes do Oracle Forms 6i; os analisadores léxico e sintático foram gerados a partir das especificações feitas na etapa (d), utilizando o Flex[7] e o Bison[8]; e o motor de *templates* eNITL foi integrado aos analisadores léxico e sintático;
g) elaboração de *templates*: definição dos *templates* que especificam o modelo de saída da ferramenta;
h) testes: os testes foram realizados durante todo o processo de desenvolvimento da ferramenta.

[7] Flex é um gerador de analisador léxico para a linguangem C.

[8] Bison é um gerador de analisador sintatico para a liguangem C.

3.1 REQUISITOS PRINCIPAIS DA FERRAMENTA

A seguir são apresentados os requisitos principais da ferramenta, divididos em requisitos funcionais (quadro 4) e requisitos não funcionais (quadro 5).

REQUISITOS FUNCIONAIS
RF01: Processar os seguintes arquivos fontes da ferramenta Oracle Forms: `Form` - formulário da aplicação (arquivo com extensão `FMB`), `Menu` - conjunto de menus (arquivo com extensão `MMB`) e `PL/SQL Libray` - conjunto de programas PL/SQL (arquivo com extensão `PLL`).
RF02: Analisar léxica e sintaticamente os arquivos de entrada.
RF03: Produzir, para cada arquivo de entrada, a tradução especificada nos *templat*es correspondentes.
RF04: Traduzir o subconjunto de código PL/SQL composto de procedimentos, funções e blocos, que contenham atribuições, testes condicionais, declarações de variáveis, laços de repetição, blocos de exceções e expressões, sendo que não será traduzido nenhum código SQL.
RF05: Traduzir os objetos visuais `Canvas`, `CheckBox`, `Frame`, `Image`, `Line`, `ListItem`, `PushButton`, `RadioGroup`, `Tab`, `Text`, `TextItem` e `Window`, implementando suas principais funcionalidades.
RF06: Mapear gatilhos de controle de objetos visuais para eventos.

Quadro 4 - Requisitos funcionais

REQUISITOS NÃO FUNCIONAIS
RNF01: Implementar a ferramenta utilizando o ambiente de desenvolvimento Microsoft Visual C++ 6.0.
RNF02: Implementar a ferramenta utilizando a linguagem de programação C++.
RNF03: Utilizar a ferramenta Flex para gerar o analisador léxico.
RNF04: Utilizar a ferramenta Bison para gerar o analisador sintático.
RNF05: Utilizar o motor de *templates* eNITL.
RNF06: Utilizar a biblioteca Forms API para manipulação de arquivos do Oracle Forms 6i.
RNF07: Utilizar a biblioteca wxWidgets-2.6.2 na implementação da interface da ferramenta.

Quadro 5 – Requisitos não funcionais

3.2 MAPEAMENTO ORACLE FORMS - JAVA

Esta seção apresenta o mapeamento realizado entre objetos visuais e componentes Swing, entre gatilhos (*triggers*) e eventos e entre código PL/SQL e código Java.

3.2.1 Objetos visuais

Os objetos visuais Oracle Forms são divididos em componentes de controle, como por

exemplo TextItem, e gráficos, tal como Line. O quadro 6 apresenta o mapeamento entre componentes Oracle Forms e componentes Swing. Alguns itens tiveram mais de um componente Swing correspondente porque possuem propriedades que não estão presentes em um único componente Swing. Cita-se ListItem que possui a propriedade List style com os possíveis valores Poplist, Combo Box e Tlist. Dependendo do valor desta propriedade, utiliza-se uma ou outra classe.

OBJETO VISUAL ORACLE FORMS	DESCRIÇÃO	COMPONENTE EM SWING
Canvas	área onde são inseridos outros componentes visuais	JLayeredPane
CheckBox	caixa de seleção que pode ser marcada ou desmarcada	JCheckBox
Frame	painel com bordas	Graphics2D
Graphics	formas geométricas (polígono, quadrado, círculo, etc)	Graphics2D
Image	contêiner de imagens	ImageIcon
Line	linha	Graphics2D
ListItem	lista de itens, relacionados em formato de lista ou de caixa de valores	JComboBox JList
Menu	menu principal	JMenu
PushButton	botão que aciona um evento quando for pressionado	JButton
RadioButton	botão em forma de círculo para seleção de um único valor	JButton
RadioGroup	componente para agrupar RadioButton	ButtonGroup
Tab	painel para agrupar componentes em guias	JTabbedPane
Text	texto que não pode ser editado	JTextArea
TextItem	área com uma ou mais linhas para entrada de dados via teclado	JTextArea JPasswordField JTextField
Window	janela filha	JInternalFrame

Quadro 6 – Objetos visuais Oracle Forms x componentes Swing

A figura 4 apresenta alguns dos objetos visuais listados no quadro 6, convertidos pela ferramenta Converte Forms, a partir de arquivos do Oracle Forms (um módulo Menu e um módulo Form).

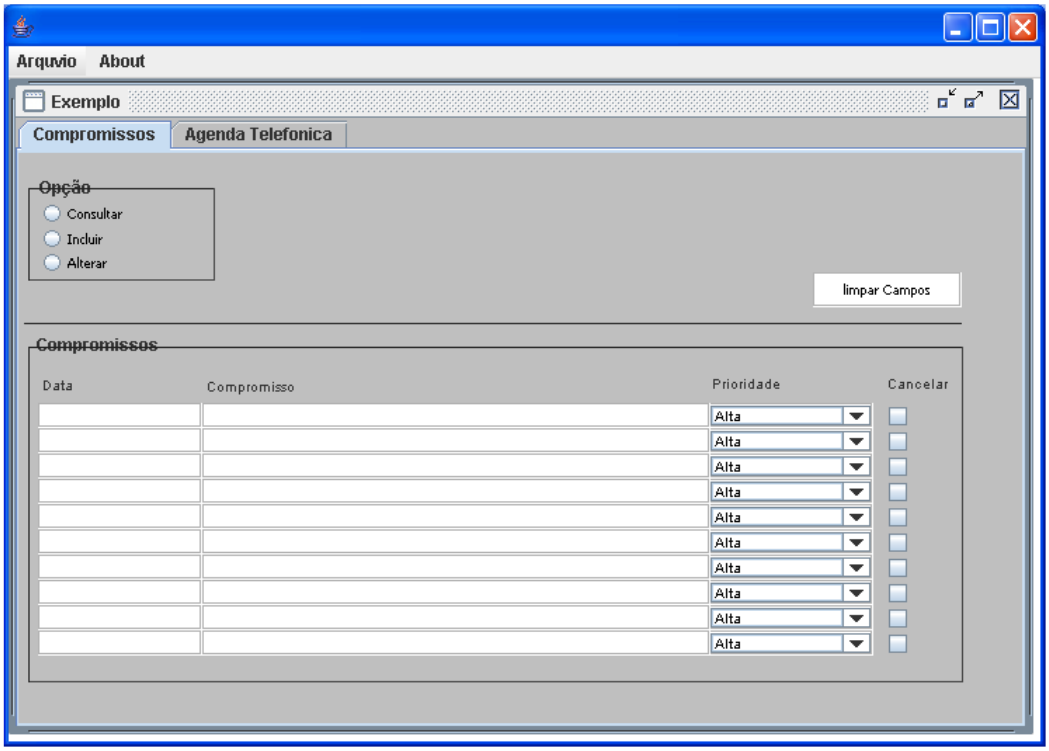


Figura 4 – Objetos visuais convertidos para Java

3.2.2 Gatilhos (*triggers*)

O mapeamento entre gatilhos (*triggers*) e eventos Java encontra-se no quadro 7. Como já mencionado na fundamentação teórica, em Java o programador precisa executar duas tarefas fundamentais para processar os eventos da interface gráfica com o usuário, sendo elas: registrar o ouvinte (*listener*) e implementar o método para o tratamento do evento.

GATILHO (*TRIGGER*)	*LISTENER*	MÉTODO
KEY-F1 a KEY-F12 KEY-OTHERS	KeyListener	keyPressed
WHEN-BUTTON-PRESSED	ActionListener	actionPerformed
WHEN-CHECKBOX-CHANGED WHEN-LIST-CHANGED	ItemListener	itemStateChanged
WHEN-MOUSE-CLICK WHEN-MOUSE-DOUBLECLICK	MouseListener	mouseClicked
WHEN-MOUSE-DOWN	MouseListener	mousePressed
WHEN-MOUSE-ENTER	MouseListener	mouseEntered
WHEN-MOUSE-LEAVE	MouseListener	mouseExited
WHEN-MOUSE-MOVE	MouseListener	mouseMoved
WHEN-MOUSE-UP	MouseListener	mouseReleased
WHEN-RADIO-CHANGED	ItemListener	itemStateChanged
WHEN-WINDOW-ACTIVATED	InternalFrameListener	internalFrameActivated
WHEN-WINDOW-CLOSED	InternalFrameListener	internalFrameClosed
WHEN-WINDOW-DEACTIVATED	InternalFrameListener	internalFrameDeactivated
WHEN-WINDOW-RESIZED	InternalFrameListener	InternalFrameListener

Quadro 7 – *Triggers* x *Listeners*

No quadro abaixo é apresentado um exemplo de código gerado pela ferramenta Converte Forms para o registro do *listener* responsável por notificar a mudança de estado de um componente do tipo `CheckBox`.

```
//-- método responsável pelos eventos disparados na tela
private void criarEventos() {
   java.awt.event.ItemListener cancelarItemListener=
                                          new java.awt.event.ItemListener(){
      public void itemStateChanged (java.awt.event.ItemEvent event ){
         blbloco.cancelarItemSelectable(event);
      }
   };
   screen.getblbloco().getcancelar_1().addItemListener(cancelarItemListener);
   screen.getblbloco().getcancelar_2().addItemListener(cancelarItemListener);
   screen.getblbloco().getcancelar_3().addItemListener(cancelarItemListener);
   screen.getblbloco().getcancelar_4().addItemListener(cancelarItemListener);
   screen.getblbloco().getcancelar_5().addItemListener(cancelarItemListener);
}

public class BLBLOCO {
   public BLBLOCO () {
   }
   private void cancelarItemSelectable(java.awt.event.ItemEvent event) {
   }
}
```

Quadro 8 – Código para registro de *listener*

3.2.3 Código PL/SQL

Os módulos do Oracle Forms foram convertidos para classes em Java, como mostrado

no quadro 9, sendo que os módulos `Forms` e `Menus` foram derivados de classes Java.

MÓDULO ORACLE FORMS	DESCRIÇÃO	SUPER CLASSE
`Forms` (arquivos `FMB`)	formulário da aplicação, composto de itens de controle e blocos PL/SQL	`JFrame`
`Menus` (arquivos `MMB`)	conjunto de sub-menus e código para acionamento dos diversos módulos	`JMenuBar`
`PL/SQL Libraries` (arquivo com extensão `PLL`)	programas PL/SQL que podem ser compartilhados por toda a aplicação	`não derivada`

Quadro 9 – Módulos Oracle Forms x classes Java

Para a conversão do código PL/SQL, buscou-se inicialmente identificar todas as construções sintáticas que seriam convertidas. Após ter identificado as construções sintáticas da linguagem Oracle Forms, procurou-se na linguagem Java as construções sintáticas equivalentes. Este mapeamento pode ser acompanhado no quadro 10.

ESTRUTURA SINTÁTICA EM ORACLE FORMS	ESTRUTURA SINTÁTICA EM JAVA

`If <expressão> then`
`  ...`
`  [ Elsif <expressão> then`
`  ... ]*`
`  [ Else`
`  ... ]?`
`End if`
onde: a cláusula `Elsif` pode ocorrer zero ou mais vezes e a cláusula `Else` é opcional
	`if (<expressão>) {`
	`  ...`
	`  }`
	`  [ else if (<expressão>) {`
	`  ...`
	`  } ]*`
	`  [ else {`
	`  ...`
	`  } ]?`

`Loop`
`  ...`
`  Exit when <expressão> ;`
`End loop`
	`while (true) {`
	`  ...`
	`  if (<expressão>) break ;`
	`}`

`identificador := <expressão> ;`	`identificador = <expressão> ;`

`For I in 1..10 loop`
`  ...`
`End loop`
	`for (long i=1; i<=10; i++) {`
	`  ...`
	`}`

`While (<expressão>) loop`
`  ...`
`End loop`
	`while (<expressão>) {`
	`  ...`
	`}`

Quadro 10 – Estruturas sintáticas Oracle Forms x estruturas sintáticas Java

Depois de determinadas as equivalências das construções sintáticas, buscou-se identificar a equivalência entre os operadores (aritméticos, relacionais e lógicos), apresentada no quadro 11.

OPERADORES	EM ORACLE FORMS	EM JAVA
aritméticos	`+, - , / , * , ||`	`+, - , / , * , +`
relacionais	`=, <=, >=, >, <, <> (ou !=)`	`==, <=, >=, >, <, !=`
lógicos	`Not, Or, And`	`!, ||, &&`

Quadro 11 – Operadores Oracle Forms x operadores Java

E, por fim, como Oracle Forms realiza as conversões de tipos de dados implicitamente,

foi necessário em Java utilizar a conversão de tipos de dados explícita. O quadro 12 exemplifica um código fonte em PL/SQL bem como o código correspondente em Java. convetido pela feramenta Converte Forms.

CÓDIGO FONTE EM ORACLE FORMS	CÓDIGO FONTE EM JAVA

PROCEDURE TESTE IS
  str   VARCHAR2(40);
  nreal NUMBER(5,2);
  nint INTEGER;
  bx   BOOLEAN;
BEGIN
 str:=   1;
 str:=   str + 3;
 	
 nreal:= nint * 5 + str;
 str:=   str + 1 || str;
 	
 nreal:= nint + 5 + 1;
 nreal:= str;
 	
 if str = 4 then
  nreal:= (nint + 5);
 	
 elsif str = 5 then
  str:= str + 3;
 	
  nint:= str;
 	
 elsif (str = 6) and (str = 5) then
 	
  str:= str + 3;
 	
 else
  str:= str + 3;
 end if;
 	
 while 1 = 1 loop
  str:= 1;
  str:= str + 3;
 	
  nint:= str;
  exit;
 end loop;
 	
 loop
  exit when 1 = 1;
 end loop;
 	
 for i in reverse 1..10 loop
  exit when i = 2;
 end loop;
 	
 for i in 1..10 loop
  null;
 end loop;
END;
	public void teste(){
	 String str;
	 double nreal;
	 long nint;
	 boolean bx;
	
	 str = String.valueOf(1);
	 str =
	  String.valueOf(Long.parseLong(str) + 3);
	 nreal = nint * 5 + Long.parseLong(str);
	 str =
	  String.valueOf(Long.parseLong(str)+1)+str;
	 nreal = nint + 5 + 1;
	 nreal = Double.parseDouble(str);
	
	 if (String.valueOf(4).equals(str)) {
	  nreal = (nint + 5);
	 }
	 else if (String.valueOf(5).equals(str)) {
	  str =
	  String.valueOf(Long.parseLong(str)+3);
	  nint = Long.parseLong(str);
	}
	 else if ((String.valueOf(6).equals(str))
	  && (String.valueOf(5).equals(str))) {
	  str =
	  String.valueOf(Long.parseLong(str) + 3);
	 }
	 else {
	  str =
	  String.valueOf(Long.parseLong(str) + 3);
	 }
	 while (1 == 1){
	  str = String.valueOf(1);
	  str =
	  String.valueOf(Long.parseLong(str)+3);
	  nint = Long.parseLong(str);
	  break;
	 }
	
	 while (true) {
	  if(1 == 1) break;
	 }
	
	 for (long i = 10; i >= 1; i--){
	  if(2 == i) break;
	 }
	
	 for (long i = 1; i <= 10; i++){
	 }
	}

Quadro 12 – Código Oracle Forms X código Java

Cada item da 1ª coluna no quadro 12 possui um item correspondente na 2ª coluna, incluindo declaração de variáveis, conversão de tipos de dados e estruturas de iteração (`if`, `while`, `loop` e `for`).

3.3 ESPECIFICAÇÃO DA SAÍDA

A geração das classes está organizada nos seguintes pacotes:

a) `view`: pacote que contém todos os componentes visuais e métodos de desenho da interface. Este pacote é originado na conversão dos componentes visuais dos módulos `Forms` e `Menus`. Cada módulo gera um arquivo com extensão `JAVA`, contendo uma classe com o nome formado por um prefixo seguido do nome do módulo. Este prefixo é utilizado para evitar coincidência de nome, sendo que para os módulos `Forms`, `Menus` e `PL/SQL Libraries` foram adotados, respectivamente, os prefixos `MD`, `MB` e `PL`. Assim, por exemplo, se o módulo convertido possui nome `EXEMPLO.FMB`, a classe gerada será `MBEXEMPLO`. Esta classe poderá ter classes internas;

b) `controller`: pacote que contém os *listeners* responsáveis pelo tratamento de eventos originados das *triggers* dos módulos `Forms` e `Menus`. Cada módulo gera um arquivo com extensão `JAVA`, contendo uma classe com o nome formado por um prefixo, seguido do nome do módulo e do sufixo `CNTRL`. Usando o mesmo exemplo do item anterior, a classe gerada será `MBEXEMPLOCNTRL`. Esta classe poderá ter classes internas;

c) `model`: pacote que contém as classes obtidas da conversão do código PL/SQL das `Program Units` encontradas em qualquer um dos três tipos de módulos. Cada módulo gera um arquivo com extensão `JAVA`, contendo uma classe com o nome formado pelo prefixo, pelo nome do módulo e pelo sufixo `MODEL`. Para o módulo `EXEMPLO.FMB`, o pacote `model` conterá a classe `MBEXEMPLOMODEL`. Neste pacote o módulo `PL/SQL Libraries` foge à regra de formação dos nomes, sendo sua classe formada apenas pelo prefixo e nome do módulo.

3.4 ANÁLISE DA ENTRADA

Arquivos Oracle Forms, conforme descrito no capítulo 2, são compostos por vários objetos, tais como `Triggers`, `Attached Libraries`, `Data Blocks`, `Program Units`, `Windows`, entre outros. Cada objeto possui uma lista de propriedades. Para acessar cada um dos objetos com suas respectivas propriedades, usa-se a biblioteca Forms API. Dessa forma,

foi necessário definir um arquivo XML (Apêndice A), contendo a lista das propriedades dos objetos, sendo que para cada propriedade tem-se o nome e a identificação usada pelo Oracle para referir-se à propriedade em questão. Para obter uma determinada propriedade de um objeto, deve-se enviar à biblioteca Forms API a identificação da propriedade juntamente com o objeto para o qual se deseja obter a propriedade.

Uma vez definido como serão coletadas todas as propriedades dos objetos, deve-se analisar léxica e sintaticamente o código PL/SQL. Para tanto, a partir da BNF especificada por Ramanathan (1997), foi especificado o subconjunto da linguagem PL/SQL que será analisado e traduzido para Java. Este subconjunto é composto de procedimentos, funções e blocos, que contenham atribuições, testes condicionais, declarações de variáveis, laços de repetição, blocos de exceções e expressões. Foram definidas as expressões regulares para os *tokens* e a BNF com as construções sintáticas da linguagem (Apêndice B). No processo da análise sintática armazena-se a árvore gramatical em memória para o uso da mesma pelo motor de *templates*.

3.5 ESPECIFICAÇÃO

Esta seção apresenta a especificação da ferramenta, contendo os casos de uso, os diagramas de classes e de seqüência que foram especificados utilizando a ferramenta Enterprise Architect.

3.5.1 Casos de uso

No diagrama de caso de uso foram especificados os casos de uso que ilustram todas as etapas necessárias para a conversão de um arquivo Oracle Forms para Java. Na figura 5 são apresentados os casos de usos relevantes ao processo de conversão que vão desde a criação de um projeto até a conversão de um arquivo para Java. Nos quadros 13 a 18 são apresentados os detalhamentos de cada caso de uso.

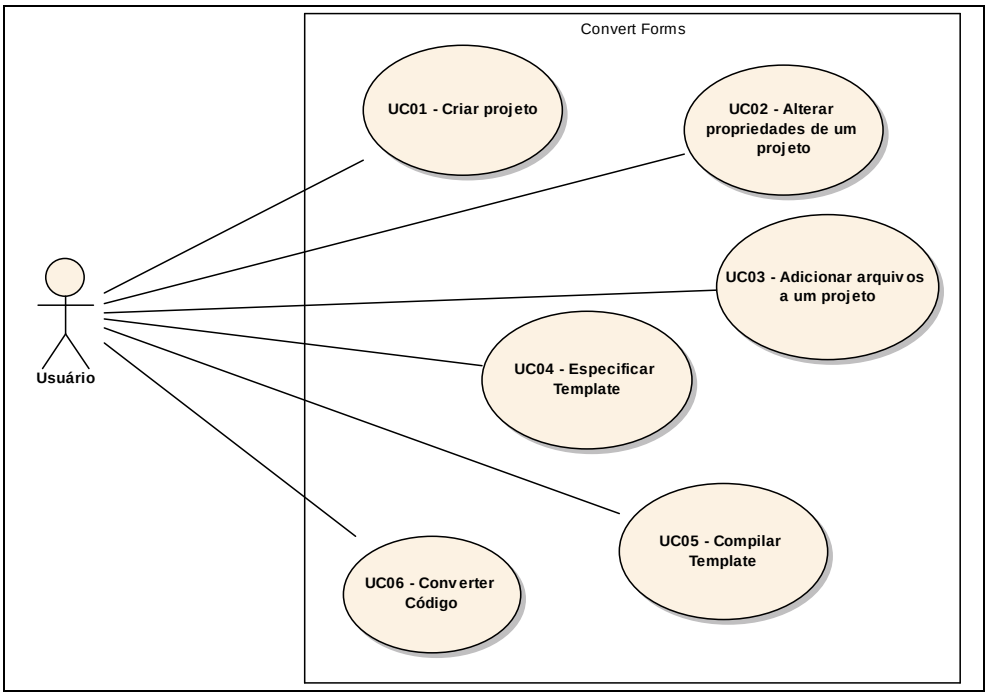


Figura 5 – Diagrama de caso de uso

UC01 – Criar projeto	
Pré-condições	Não existe.
Cenário principal	01) O usuário seleciona opção para criar um novo projeto. 02) A ferramenta solicita nome e diretório para salvar o projeto a ser criado. 03) O usuário fornece as informações solicitadas. 04) A ferramenta cria o projeto com o nome e no diretório informados e adiciona o projeto na área de trabalho (*workspace*).
Pós-condições	Um projeto foi criado e adicionado à área de trabalho.

Quadro 13 – Detalhamento do caso de uso UC01

UC02 – Alterar propriedades de um projeto	
Pré-condições	Deve haver ao menos um projeto na área de trabalho.
Cenário principal	01) O usuário seleciona um projeto. 02) A ferramenta apresenta as propriedades do projeto selecionado. 03) O usuário altera o nome do projeto, indica o *template* que será utilizado como modelo para gerar código e o diretório base onde os arquivos gerados serão gravados. 04) A ferramenta atualiza as informações fornecidas para o projeto.
Pós-condições	As propriedades de um projeto foram salvas.

Quadro 14 – Detalhamento do caso de uso UC02

UC03 – Adicionar arquivos a um projeto	
Pré-condições	Deve haver ao menos um projeto na área de trabalho. Deve haver ao menos um arquivo com extensão `FMB`, `MMB` ou `PLL`.
Cenário principal	01) O usuário seleciona um projeto. 02) A ferramenta apresenta a relação de arquivos existentes com extensão `FMB`, `MMB` ou `PLL`. 03) O usuário seleciona o arquivo desejado. 04) A ferramenta adiciona o arquivo selecionado ao projeto.
Pós-condições	Um arquivo foi adicionado a um projeto. O arquivo adicionado está habilitado para conversão.

Quadro 15 – Detalhamento do caso de uso UC03

UC04 – Especificar *template*	
Pré-condições	Deve haver ao menos um projeto na área de trabalho.
Cenário principal	01) O usuário seleciona um projeto. 02) A ferramenta apresenta a opção para especificar *templates*. 03) A ferramenta solicita nome, extensão e diretório do arquivo a ser salvo. 04) O usuário fornece as informações solicitadas. 05) O usuário especifica o *template* e manda salvá-lo. 06) A ferramenta salva o *template*, associando-o ao projeto previamente selecionado.
Pós-condições	Um *template* foi incluído na área de trabalho do respectivo projeto.

Quadro 16 – Detalhamento do caso de uso UC04

UC05 – Compilar *template*	
Pré-condições	Deve haver ao menos um *template* na área de trabalho.
Cenário principal	01) O usuário seleciona um *template*. 02) A ferramenta analisa o *template* selecionado, exibindo o resultado da compilação.

Quadro 17 – Detalhamento do caso de uso UC05

UC06 – Converter código	
Pré-condições	Deve haver na área de trabalho ao menos um projeto com um arquivo (`FMB`, `MMB` ou `PLL`) adicionado e um *template* associado.
Cenário principal	01) O usuário seleciona o arquivo associado a um projeto. 02) A ferramenta analisa os *templates* associados ao projeto. 03) A ferramenta analisa o código fonte do arquivo a ser convertido, coletando e armazenando as informações necessárias para gerar código. 04) A ferramenta gera os arquivos de saída conforme especificado nos *templates*.
Pós-condições	Dois ou mais arquivos com a extensão `.java` foram gerados no diretório base do projeto.

Quadro 18 – Detalhamento do caso de uso UC06

3.5.2 Diagramas de classes

A figura 6 apresenta os pacotes para as classes especificadas. Tem-se o pacote `Plugin EngineTranslator` que contém todas as classes necessárias para a construção do *plugin*[9] de conversão, o pacote `PL/SQL - Analyzer` com classes responsáveis pela análise do código PL/SQL e o pacote `Forms Files` com classes responsáveis pelo acesso aos arquivos Oracle Forms.

[9] O conceito de *plugin* é abordado em detalhes na seção 3.6.1

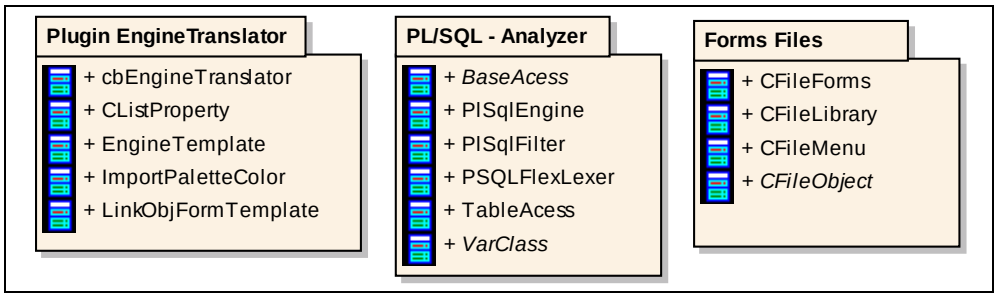


Figura 6 – Diagrama de pacotes

Na figura 7 são apresentadas as classes do pacote `Plugin EngineTranslator`, assim como o relacionamento dessas com algumas classes do pacote `Forms Files`. A classe `cbEngineTranslator` é responsável por tratar os eventos relacionados com o processo de conversão, provendo acesso aos componentes visuais da ferramenta. A classe `EngineTemplate` é instanciada pela classe `cbEngineTranslator` quando há uma solicitação de conversão de arquivo. A classe `LinkObjFormTemplate`, também instanciada por `cbEngineTemplate`, é responsável por devolver o código dinâmico dos *templates* quando solicitado pelo motor de *templates* eNITL. A classe `CListProperty` mantém, em um arquivo XML, a lista de propriedades dos arquivos Oracle Forms. Quando necessário, retorna o identificador correspondente a uma determinada propriedade. A classe `ImportPaletteColor` mantém a paleta de cores do Oracle Forms, sendo que para cada cor tem-se o nome e o valor RGB da cor (em um arquivo XML, conforme Apêndice C).

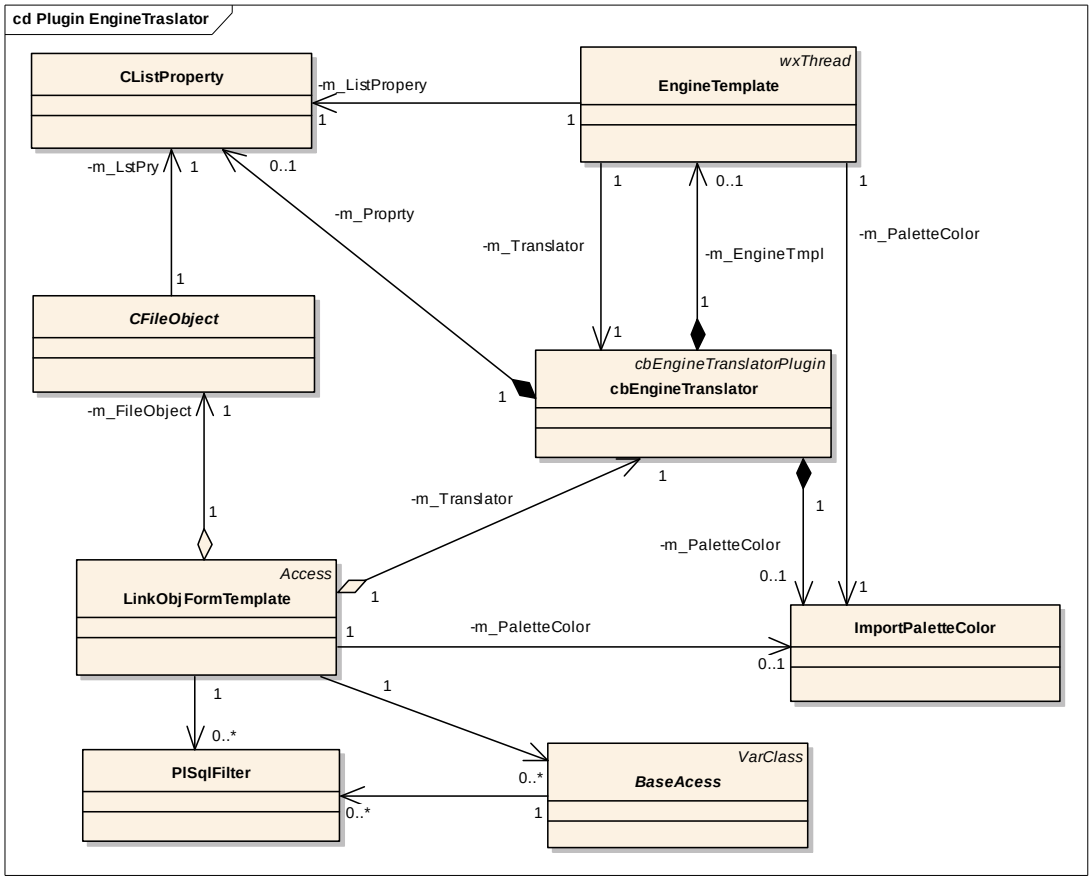


Figura 7 – Diagrama de classes do pacote `Plugin cbEngineTranslator`

A figura 8 apresenta o diagrama de classes do pacote `PL/SQL - Analyzer`. Neste pacote encontram-se as classes responsáveis pela análise do código PL/SQL. A classe `PSQLFlexLexer` realiza a análise léxica do código reconhecendo os *tokens* no arquivo de entrada, conforme especificação léxica da linguagem PL/SQL (Apêndice B). `PSQLFlexLexer` é utilizada pelo analisador sintático. A classe `PlSqlEngine` realiza a análise sintática do código PL/SQL, gerando a árvore gramatical para o arquivo de entrada. A classe `PlSqlFilter` é responsável por aplicar filtros aos *tokens* visando identificar variáveis declaradas, funções, procedimentos, blocos, entre outros. O conjunto de *tokens* resultante é armazenado em um objeto da classe `TableAcess`.

A classe `BaseAcess` é abstrata. Sua função é servir de interface para o acesso genérico das classes `PlSqlEngine` e `TableAcess` pela classe `LinkObjFormTemplate`. `BaseAcess` é a especialização de `VarClass`. A classe `VarClass` também é abstrata. Vale lembrar que classes abstratas não podem ser instanciadas. É utilizada pelo motor de *templates* eNITL, para verificar se um objeto possui referências de outros objetos.

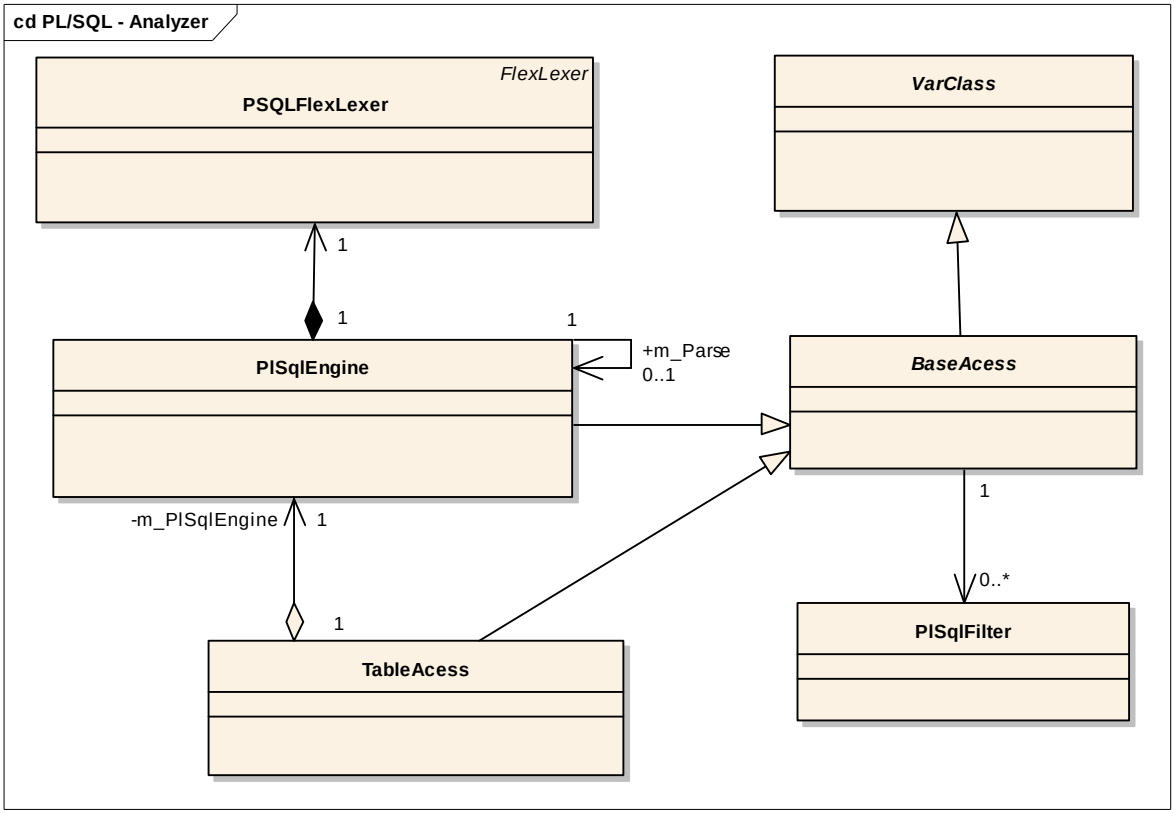


Figura 8 – Diagrama de classes do pacote `PL/SQL - Analyser`

A figura 9 apresenta as classes responsáveis pelo acesso aos arquivos do Oracle Forms. A classe `CFileObject` é uma classe abstrata, ou seja, é a interface para acesso a arquivos independente do tipo do arquivo acessado. Já a classe `CFileForms` é uma especialização dessa classe para acesso aos arquivos do tipo `FMB`, a classe `CFileMenu` é a especialização para acesso a arquivos do tipo `MMB` e a classe `CFileLibrary` para acesso a arquivos do tipo `PLL`.

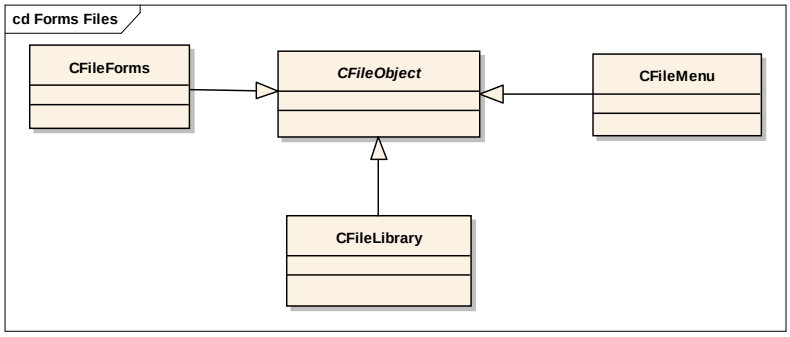


Figura 9 – Diagrama de classes do pacote `Forms Files`

3.5.3 Diagramas de seqüência

O diagrama de seqüência da figura 10 detalha o método `onConvertFile` da classe

`cbEngineTranslator`. Este método coordena toda a conversão de um arquivo Oracle Forms para Java. Ao ser disparado o evento `onConvertFile`, o *plugin* `cbEngineTranslator` cria uma instância de `EngineTemplate` e envia as mensagens para configurar o processo de conversão, que são: arquivo a converter, lista de propriedades, *template* inicial, lista de *templates* a serem usados na conversão e diretório raiz.

Então se inicia a preparação do motor de *templates* eNITL. A instância de `EngineTemplate` envia mensagem para `List` para a criação da lista de *templates* compilados, onde cada *template* compilado sem erro fica armazenado. Após a compilação de todos os *templates*, `EngineTemplate` carrega o arquivo a ser convertido a partir de uma mensagem enviada a `CFileObject`. Em seguida, `EngineTemplate` instancia um objeto da classe `LinkObjFormTemplate` que será responsável por devolver o código dinâmico dos *templates* a ser utilizado pelo motor de *templates* na geração dos arquivos de saída. Por fim, `EngineTemplate` solicitada que o motor de *templates* inicie a geração das classes Java descritas na seção 3.3. Neste diagrama não foram representadas as trocas de mensagens do motor de *templates* com `LinkObjFormTemplate`, as quais são apresentadas no diagrama da figura 11.

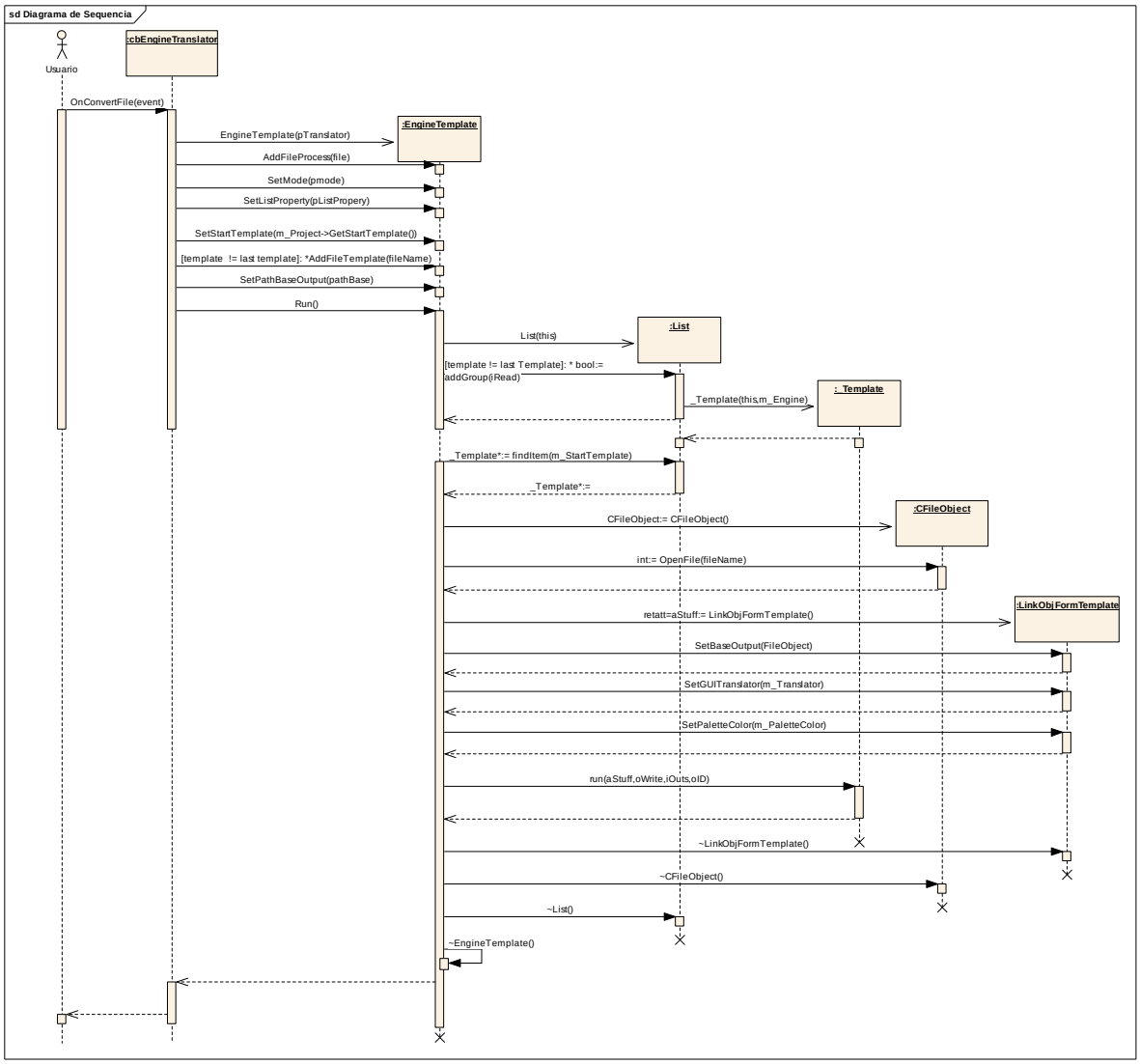


Figura 10 – Diagrama de seqüência – parte 1

A figura 11 apresenta as trocas de mensagens, entre o motor de *templates* eNITL (representado pelo objeto `_Template`) e os objetos provedores de acesso ao código dinâmico. As trocas de mensagens ocorrem basicamente na seguinte seqüência: o objeto `_Template` envia a mensagem `value`, `member` ou `output` ao objeto `LinkObjFormTemplate;` para devolver o valor solicitado para `_Template`, `LinkObjFormTemplate` poderá enviar mensagens aos objetos `CFileObject`, `PlSqlEngine` ou `BaseAcess`; após estes objetos devolverem a mensagem ao objeto `LinkObjFormTemplate`, este retorna a mensagem para `_Template` na estrutura `_Template::Var`.

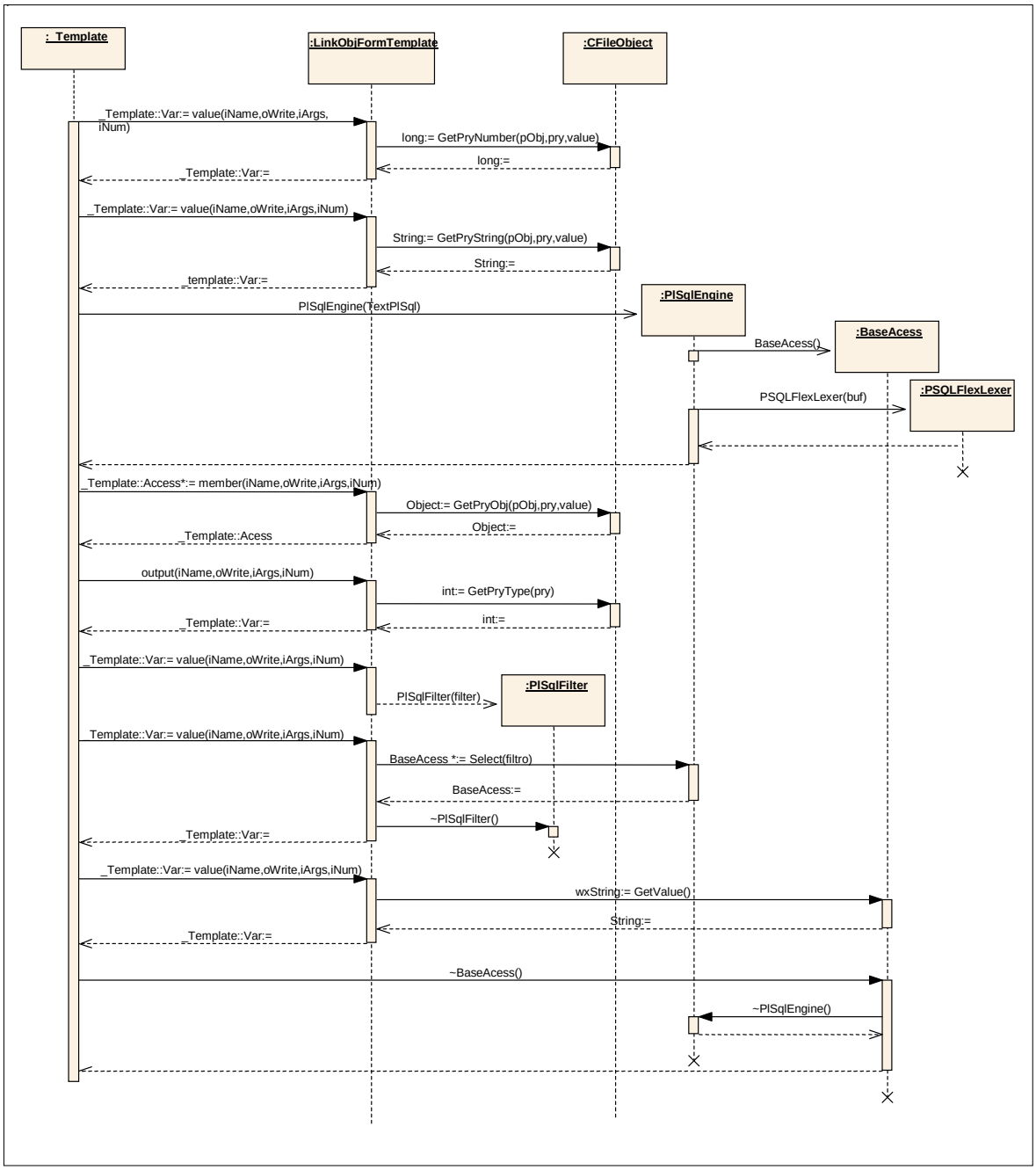


Figura 11 – Diagrama de seqüência – parte 2

3.6 IMPLEMENTAÇÃO

Nesta seção é apresentada a implementação da ferramenta Converte Forms, incluindo as técnicas e as ferramentas utilizadas no desenvolvimento.

3.6.1 Técnicas e ferramentas utilizadas

A ferramenta Converte Forms foi desenvolvida utilizando a linguagem de programação C++, no ambiente de desenvolvimento Microsoft Visual Studio C++. Foram usadas no desenvolvimento: bibliotecas para implementação da interface; a biblioteca Forms API para auxiliar na leitura dos arquivos de entrada; as ferramentas Flex e Bison para gerar, respectivamente, os analisadores léxico e sintático da linguagem PL/SQL; e o motor de *templates* eNTIL. Estas ferramentas e bibliotecas foram utilizadas com o intuito de obter métodos mais rápidos e mais adequados para a construção da ferramenta proposta.

Além disso, a ferramenta Converte Forms usa o conceito *plugin*. Com isto, é possível construir módulos com baixíssimo nível de acoplamento, trazendo como vantagens o desenvolvimento de módulos menores e independentes. Cada novo *plugin*, desenvolvido como subclasse da classe `cbPlugin`, deve gerar uma DLL que deve ser colocada no diretório `/share/plugin/` do diretório do Converte Forms. A ferramenta, ao iniciar, tenta carregar todas as DLLs encontradas nesse diretório. Como exemplo pode-se citar ferramentas para auto complementar código, formatadores de códigos e geradores de especificação.

3.6.1.1 Interface

A implementação da GUI de Converte Forms é fortemente baseada em três projetos *open source*, quais sejam Code::Blocks Studio (MANDRAVELLOS, 2006), TinyXML (THOMASON, 2006) e Scintilla (HODGSON, 2006). Code::Blocks Studio forneceu o *layout* da GUI. Utilizou-se o TinyXML para manipulação dos arquivos XML com a lista de propriedades de objetos Oracle Forms e com as configurações dos projetos. Scintilla forneceu os recursos utilizados no editor de *templates*. Foi necessário integrar esses três projetos bem como adaptá-los, visto que, apesar de terem sido desenvolvidos para serem portáveis, foram encontrados problemas quando os mesmos foram compilados no ambiente Microsoft Visual Studio C++. Cita-se a incompatibilidade de versões C++ e o comportamento diferente do descrito pelos desenvolvedores dos projetos.

Na implementação da interface também foi usada a biblioteca wxWidgets-2.6.2 (OLLIVIER, 2006). Um dos motivos pela adoção de wxWidgets foi a utilização de partes do código de Code::Blocks Studio na implementação da ferramenta, o qual também foi

desenvolvido utilizado a biblioteca wxWidgets. Para o funcionamento correto de Converte Forms, wxWidgets teve que ser compilada como biblioteca dinâmica (DLL) em função do uso do conceito de *plugin* no desenvolvimento da ferramenta.

3.6.1.2 Forms API

A biblioteca Forms API foi utilizada no desenvolvimento da ferramenta Converte Forms para a manipulação dos arquivos Oracle Forms. As principais funções utilizadas e respectivas funcionalidades são apresentadas no quadro 19.

FUNÇÃO	FUNCIONALIDADE
`d2fobgp_GetBoolProp`	buscar propriedade de um objeto cujo valor é do tipo lógico
`d2fobgp_GetNumProp`	buscar propriedade de um objeto cujo valor é um número
`d2fobgp_GetTextProp`	buscar propriedade de um objeto cujo valor é uma cadeia de caracteres
`d2fobgp_GetObjProp`	buscar propriedade de um objeto cujo valor é outro objeto
`d2fobhp_HasProp`	identificar se a propriedade pertence ao objeto
`d2fprgt_GetType`	retornar o tipo da propriedade
`d2fctxcr_Create`	criar um contexto
`d2fctxde_Destroy`	destruir um contexto
`d2fobfo_FindObj`	procurar determinado objeto pelo nome
`d2fitmgle_GetListElem`	buscar lista de elementos para o objeto do tipo `List`
`d2ffmdld_Load`	carregar arquivos do tipo `FMB` para a memória
`d2flibld_Load`	carregar arquivos do tipo `PLL` para a memória
`d2fmmdld_Load`	carrega arquivos do tipo `MMB` para a memória

Quadro 19 – Principais funções da Forms API

3.6.1.3 Flex e Bison

Flex, segundo Estes (2006), é uma ferramenta para geração de analisadores léxicos. Foi utilizada para gerar o analisador léxico para PL/SQL a partir da especificação léxica da linguagem (Apêndice B). Para uma implementação mais adequada da ferramenta Converte Forms, foi necessário reimplementar as rotinas geradas pelo Flex para leitura dos caracteres de entrada, uma vez que para essas rotinas a entrada deve ser via arquivo ou via teclado. Nesse caso, o construtor da classe `PLSQLFlexLexer` e seus métodos de leitura foram alterados para permitir que a entrada do analisador léxico fosse obtida diretamente da memória.

Bison foi utilizado para gerar o analisador sintático da linguagem PL/SQL. Por *default*, Bison gera código estruturado para o analisador sintático (FREE SOFTWARE FOUNDATION, 2006). Este foi o único caso em que não se conseguiu utilizar a orientação a objeto no desenvolvimento do Converte Forms. Para possibilitar a geração de analisadores sintáticos orientados a objeto, seria necessária a implementação da geração do analisador sintático, através de um arquivo de *templates* do Bison. Optou-se em não alterar sua geração padrão devido a complexidade e a demora para sua implementação.

3.6.1.4 eNITL

O motor de *templates* eNITL foi utilizado para o processamento dos *templates*. Para a correta implementação da ferramenta Converte Forms foi necessário alterá-lo de forma a permitir o processamento de múltiplos arquivos de *templates*, a inclusão dos tipos ponteiro e classe, e a criação da classe `VarClass` para identificar quando um objeto não possui mais referências e pode ser destruído.

Quando foram realizados os primeiros testes com eNITL, não foi identificado nenhum problema no processamento dos *templates* especificados. Mas à medida que o tamanho e o número de *templates* aumentaram, o eNITL começou a ter um comportamento instável. Foi necessário interromper o desenvolvimento da ferramenta Converte Forms para corrigir erros no motor de *templates* eNITL. Os problemas identificados foram:

a) estruturas `ELSEIF` devem sempre possuir uma cláusula `ELSE`, caso contrário os *templates* são analisados de forma errônea;
b) erros aleatórios em decorrência da alocação de memória;
c) número máximo de *templates* que podem ser processados.

Esses dois últimos erros tiveram que ser corrigidos para não inviabilizar o uso do eNITL no desenvolvimento da ferramenta proposta. Para tanto, foi necessário estudar detalhadamente o motor de *templates* visando entender o código fonte, identificar os erros descritos e propor uma solução. Da análise feita, sabe-se que o eNITL possui uma pilha de dados de tamanho variável. À medida que o processamento dos *templates* atinge o limite máximo de memória alocada, é alocada nova área de memória, adicionando mais quatro unidades de memória à pilha. Nesta operação, ocorre uma mudança do endereço onde se encontra a pilha de dados. O problema ocorre quando o eNITL obtém o topo da pilha por referência e segue sua execução. Pelo fato de ter atingido seu limite máximo, ao fazer a

alocação da nova área de memória, o valor do topo da pilha anteriormente armazenado torna-se inválido. O outro problema, também relacionado com alocação de memória, limitava em 64 o número máximo de *templates* que podem ser processados pelo eNITL.

3.6.2 Implementação da ferramenta

O desenvolvimento da ferramenta iniciou com a compilação das bibliotecas já mencionadas na seção anterior. A fim de obter uma melhor organização e facilitar a manutenção do código fonte, Converte Forms foi dividido em vários projetos, sendo cada um um módulo `LIB`, `DLL` ou executável. Os projetos criados foram:

a) `Sdk`: conjunto de classes com funcionalidades básicas, utilizado em praticamente todos os outros módulos. É compilado como biblioteca estática (`LIB`);
b) `TinyXML`: conjunto de classes para manipulação de arquivos XML, uma vez que as configurações da ferramenta, as propriedades dos projetos e as informações da área de trabalho (*workspace*) são salvas em arquivos XML. É compilado como biblioteca estática (`LIB`);
c) `wxScintilla`: biblioteca com rotinas para a manipulação do conjunto de caracteres digitado no editor de *templates*. É compilado como biblioteca estática (`LIB`);
d) `PlSqlEngine`: módulo responsável pela análise do código PL/SQL, contemplando a implementação do pacote `PL/SQL Analyzer`. É compilado como bliblioteca dinâmica (`DLL`);
e) `ManagerConvert_Forms`: módulo responsável pelo compartilhamento de memória entre os demais módulos. É compilado como biblioteca dinâmica (`DLL`);
f) `wxDockit`: biblioteca com rotinas para manipulação de *stream* de arquivos. É compilado como biblioteca estática (`LIB`);
g) `EngineTranslator`: módulo respónsavel pela conversão de Oracle Forms em Java. Contempla a implementação dos pacotes `Plugin EngineTranslator` e `Forms Files`. É compilado como biblioteca dinâmica (`DLL`);
h) `Convert_Forms`: módulo da aplicação principal. É o executável (`EXE`).

Entre estes módulos, somente serão detalhados os módulos `EngineTranslator` e `PlSqlEngine`, pois são estes os módulos que compreendem todo o processo de conversão de

Oracle Forms para Java. As principais classes do módulo `EngineTranlator` podem ser observadas no diagrama de classes da figura 7. Na seqüência são apresentadas a implementação de algumas classes, começando pela classe `cbEngineTranslator`.

A classe `cbEngineTranslator` é derivada de `cbPlugin` que é uma interface entre a aplicação e *plugins*. Os seus principais métodos são `OnAttach`, `OnRelease`, `OnConvertFile`, `OnCompileFile`, `OnCompileAll`, `OnConvertAll`. O método `OnAttach` é executado quando a aplicação principal carrega a `DLL EngineTranlator` para a memória, criando a `toolbar`, os menus e uma guia na área de mensagens. Esse método também registra o tratamento de eventos dos objetos visuais criados pelos *plugins*. O método `OnRelease` é executado quando a aplicação é finalizada, sendo o objeto da classe destruído. Os métodos `OnConvertFile`, `OnCompileFile`, `OnCompileAll` e `OnConvertAll` são responsáveis pelo tratamento dos eventos disparados pelo usuário solicitando a conversão de um arquivo Oracle Forms para Java ou a compilação de um *template*. No quadro 20 é apresentado um trecho no método `OnConvertFile`. Os outros três métodos são muito semelhantes a este, diferenciando apenas na quantidade de arquivos processados ou no modo como `EngineTemplate` é executado (compilar - `compiler` - ou converter - `execute`).

No quadro 20 tem-se que um objeto de `EngineTemplate` é instanciado, sendo indicado o modo como `EngineTemplate` será executado (trecho 1). Na seqüência, o diretório raiz para a geração dos arquivos é setado (trecho 2), adicionado o arquivo a ser processado (trecho 3), os *templates* a serem utilizados para a conversão (trecho 4) e a lista de propriedades. Por fim, é disparada a execução da conversão através do método `run` (trecho 5).

```
...
m_EngineTmpl = new EngineTemplate(this);                                    01
m_EngineTmpl->SetMode(execute);

wxFileName pathOutput(m_Project->GetOutPutBase());
pathOutput.Assign(pathOutput.GetFullPath());
pathOutput.Normalize(wxPATH_NORM_ALL & ~wxPATH_NORM_CASE,
                     m_Project>GetBasePath());
m_EngineTmpl->SetPathBaseOutput(pathOutput.GetFullPath());                  02

m_EngineTmpl->ClearListFileProcess();
m_EngineTmpl->ClearListTemplate();

file.Assign(file.GetFullPath());                                            03
file.Normalize(wxPATH_NORM_ALL & ~wxPATH_NORM_CASE,
                m_Project->GetBasePath());
m_EngineTmpl->AddFileProcess(file);

for (int I =0;i< m_Project->GetFilesCount();i++) {                          04
      ProjectFile *prjfile = m_Project->GetFile(i);
      if (prjfile->file.GetExt().Matches(CFT_EXT))
            m_EngineTmpl->AddFileTemplate(prjfile->file);
}
ClearLog();
m_EngineTmpl->SetListProperty(GetListProperty());                           05
m_EngineTmpl->SetStartTemplate(m_Project->GetStartTemplate());
m_EngineTmpl->Run();
...
```

Quadro 20 – Método `OnConvertFile`

A classe `EngineTemplate` é derivada de `wxThread`. Assim, o processo de conversão não paralisa a GUI enquanto a conversão não for finalizada. `EngineTemplate` possui dois métodos principais `DoExecute` e `DoCompile`. Estes são ativados conforme o modo atribuído para execução de `EngineTemplate` (trecho 1 – quadro 20). O método `DoExecute` (quadro 21):

a) trecho 1: instancia a lista de *templates*, onde cada *template* será armazenado;

b) trecho 2: carrega cada *template* a ser usado no processo de conversão;

c) trecho 3: busca o *template* inicial;

d) trecho 4: instancia um objeto de `StdWriter`, indicando a saída padrão para mensagens geradas durante o processo de conversão, neste caso, a guia *log* na área de mensagens;

e) trecho 5: instancia um objeto de `Outputs` que é o gerenciador de arquivos do motor de *templates*. Esta classe é responsável por gerar os arquivos de saída. Por este motivo, recebe o diretório raiz onde serão gerados os arquivos;

f) trecho 6: instancia um objeto de `LinkObjFormTemplate` que responsável por prover o código dinâmico ao motor de *templates*;

g) trecho 7: processa cada arquivo a ser convertido, sendo que cada tipo de arquivo Oracle Forms é aberto por uma classe específica. Assim, arquivos `FMB` serão

abertos pela classe `CFileForms`, `MMB` serão abertos pela classe `CFileMenu` e `PLL` serão abertos pela classe `CFileLibrary`;

h) trecho 8: finaliza a análise do *template*.

O método `DoCompile` executa apenas os trechos 1 e 2 do método `DoExecute`.

```
...
wxFileListNode *node;
node = m_LstTmpl.GetFirst();
if (m_ListTemplate)                                                    01
      delete m_ListTemplate;
m_ListTemplate = new _Template::List(this);

while(node) {                                                          02
      wxFileName *fullNameFile = node->GetData();
      if (!LoadTemplate(fullNameFile->GetFullPath()))
            return ;
      node = node->GetNext();
}

_Template * aTmpl = m_ListTemplate->findItem(m_StartTemplate);         03
if  (!aTmpl) {
      m_Translator->AddOutputLine(_(" Start template  not found."));
      return ;
}

StdWriter aWriter(this);                                               04

Outputs aAlt(m_PathBaseOutPut);                                        05

node = m_LstFileProces.GetFirst();
LinkObjFormTemplate aStuff;                                            06
aStuff.SetBaseOutput(m_PathBaseOutPut);
aStuff.SetGUITranslator(m_Translator);
if (m_PaletteColor)
      aStuff.SetPaletteColor(m_PaletteColor);

while(node) {                                                          07
      ...
      CFileObject *FileObject;
      wxFileName *fullNameFile = node->GetData();
      m_Translator->AddOutputLine(_("converting file ") +
                                  fullNameFile->GetFullPath() +
                                  _T("..."));
      if (fullNameFile->GetExt().Matches(FMB_EXT)) {
            FileObject = new CFileForms();
            if (!m_PaletteColor) {
                  m_PaletteColor= m_Translator->GetPaletteColor(
                                              fullNameFile->GetFullPath());
                  aStuff.SetPaletteColor(m_PaletteColor);
            }
      }
      else  if (fullNameFile->GetExt().Matches(PLL_EXT))
            FileObject = new CFileLibrary();
      else if (fullNameFile->GetExt().Matches(MMB_EXT))
            FileObject = new CFileMenu();
      FileObject->SetListProperty(m_ListPropery);
      FileObject->OpenFile(fullNameFile->GetFullPath());
      aStuff.SetFileObject(FileObject);
      aTmpl->run(&aStuff, aWriter, &aAlt);                             08
      delete FileObject;
      Node = node->GetNext();
}
...
```

Quadro 21 – Método `DoExecute`

As classes `CFileForms`, `CFileMenu` e `CFileLibrary` são idênticas na sua construção diferenciando basicamente no tipo de arquivo que cada classe é capaz de abrir. As três classes são derivadas da classe `CFileObject`. No quadro 22 é apresentado o método `OpenFile` da classe `CFileForms`.

```
int CFileForms::OpenFile(wxString &fileName) {
     d2fstatus status = 0;
     if (CreateContext()){                                                    01
            char *form_name= new char[fileName.length()+1];
            strcpy(form_name,fileName.c_str());
            form_name = strtok(form_name,".");

            status = d2ffmdld_Load(pd2fctx, &m_Module,                        02
                                        (unsigned char*)form_name, FALSE);
              delete form_name;
              If (status == D2FS_SUCCESS) {
                    AttachedLib(m_Module);                                    03
              }
              return D2FS_SUCCESS;
       }
       return D2FS_FAIL;
}
```

Quadro 22 – Método `OpenFile`

Para a abertura de um arquivo utilizando a Forms API, primeiro deve-se criar um contexto (trecho 1). Em seguida, deve-se carregar o arquivo para a memória (trecho 2), sendo que para arquivos do tipo `FMB` deve-se utilizar a função `d2ffmdld_Load`, para arquivos do tipo `MMB` deve-se utilizar a função `d2fmmdld_Load` e para arquivos do tipo `PLL` deve-se utilizar a função `d2flibld_Load`. Após esta etapa, são anexados os demais arquivos `PLL` caso existam `Attached Libraries` para este módulo (trecho 3).

A classe `LinkObjFormTemplate` é responsável por obter o código dinâmico necessário para que o motor de *templates* possa gerar os arquivos de saída. Esta classe é derivada de `_Template::Access`, conforme mencionado na seção 2.6.1. Possui um *hashmap* com todas as operações executadas por ela, sendo iniciado pelo método `InitFunction` (quadro 23). O *hashmap* foi criado com o intuito de agilizar a identificação de qual valor dinâmico deve ser devolvido ao motor de *templates* quando da execução dos métodos `value` e `member`.

```
  ...
  m_MapFunction ["object"] =1;
  m_MapFunction ["next"] =2;
  m_MapFunction ["previous"] =3;
  m_MapFunction ["copyfile"] =4;
  m_MapFunction ["upper"] =5;
  m_MapFunction ["lower"] =6;
  m_MapFunction ["gettypeobj"] =7;
  m_MapFunction ["replace"] =8;
  ...
```

Quadro 23 – Método `InitFunction`

O método `value` (quadro 24) executa as sequintes operações: recebe o nome da operação que deve ser executada (parâmetro `iName`); obtém do *hashmap* o identificador desta

operação (trecho 1); adiciona um deslocamento ao identificador uma vez que objetos diferentes podem possuir operações com o mesmo nome; executa o código correspondente da operação (trecho 2), devolvendo o valor resultante para o motor de *templates*. Todas as operações são identificadas com `Opxxxxx`, onde `xxxxx` é o nome da operação.

```
...
  unsigned int idx = m_MapFunction[iName];                              01
idx += m_SelectMember;
m_SelectMember=0;
switch(idx) {
     case 0:  return OpProperty(iName);
     case 1:  return OpObject(iArgs,iNum);                              02
     case 2:  return OpProperty(iName);
     case 3:  return OpProperty(iName);
     case 4:  return OpCopyFile(iArgs,iNum);
     case 5:  return OpUpper(iArgs,iNum);
     case 6:  return OpLower(iArgs,iNum);
     case 7:  return OpGetTypeObj(iArgs,iNum);
     ...
```

Quadro 24 – Método `value`

O método `member` utiliza do mesmo artifício apresentado no método `value` para a identificação da operação que deverá ser executada, como mostra quadro 25. Como descrito anteriormente, o método member é chamado quando o eNITL encontra um código dinâmico na forma `env.member[.member...].command`. Devolve o ponteiro do objeto para o qual a operação será realizada quando da próxima execução do método `value`.

```
...
unsigned int idx = m_MapFunction[iName];
switch(idx) {
     case 1:       m_SelectMember =0;
                   m_ObjectFrm = iArgs->point;
                   break;
     case 27:      m_SelectMember =0;
                   m_ObjectFrm = m_FileObject->GetModule();
                   break;
     case 9:       m_SelectMember =50;
                   m_ObjectFrm =iArgs->point;
                   break;
}
return this;
...
```

Quadro 25 – Método `member`

O acesso às propriedades dos arquivos Oracle Forms é feito pelo método `OpProperty` (da classe `LinkObjFormTemplate` – quadro 26). Inicialmente busca-se na lista de propriedades o identificador da propriedade (trecho 1). Em seguida, obtém-se o tipo da propriedade (trecho 2) através do método `GetPryType` da classe `CFileObject` e o valor com base no tipo da mesma (trecho 03), sendo para isso executados os métodos `GetPryString`, `GetPryNumber`, `GetPryObj` e `GetPryBool`, retornando respectivamente um *string*, um número, um objeto ou um valor lógico.

```
_Template::Var LinkObjFormTemplate::OpProperty( const char * iName){
       _Template::Var aRes;
       bool bval;
       wxString  sval;
       property  pry = m_FileObject->GetIDPropery(iName);                    01
       int  pryTp = m_FileObject->GetPryType(pry);                           02
       switch(pryTp) {
           case PryTpText:                                                   03
                if (m_FileObject->GetPryString(m_ObjectFrm,pry,&sval)){...}
                else {...}
                break;
           case PryTpNumber:
                if (m_FileObject->GetPryNumber(m_ObjectFrm,pry,
                                          (unsigned long *)&aRes.num)){...}
                else {...}
           case PryTpObject:
                aRes.type = _Template::Var::ePoint;
                if  (!m_FileObject->GetPryObj(m_ObjectFrm,pry, &aRes.point))
                {
                 ...
                }
                return aRes;
           case PryTpBool:
                aRes.type = _Template::Var::eString;
                if (m_FileObject->GetPryBool(m_ObjectFrm,pry,&bval)){...}
                else {...}
                return aRes;
           default:   AddOutputLine(MSGERR0001,iName);
                      return VarVoid();
       }
       return VarVoid();
}
```

Quadro 26 – Método `OpProperty`

O método `GetPryType` (quadro 27) obtém o tipo de retorno de uma propriedade.

```
int CFileObject::GetPryType(property pry) {
       return d2fprgt_GetType(pd2fctx, pry);
}
```

Quadro 27 – Método `GetPryType`

O método `GetPryString` obtém o valor de uma propriedade cujo tipo de retorno é um *string*. Para tanto, recebe como parâmetro o objeto e o identificador da propriedade; verifica se a propriedade é válida para o objeto (trecho 1); e, em caso afirmativo, executa a função `d2fobgt_GetTextProp` da Forms API, para obter o valor da propriedade. Para as os métodos `GetPryNumber`, `GetPryObj` e `GetPryBool` a diferença em relação ao método apresentado está na função utilizada para buscar o valor da propriedade.

```
bool CFileObject::GetPryString (FrmObject* pObj,
                                property pry,
                                wxString* value) {
      text  * val = NULL;
    if (GetHashPry(pObj,pry)) {                                          01
                  if (d2fobgt_GetTextProp(pd2fctx,pObj,pry,&val)!=0) {
                  ...
                  return false;
            }
            else {
                  value->Clear();
                  value->Append(val);
                  return true;
            }
      }
      else {
            return false;
      }
      return true;
}
```

Quadro 28 – Método `GetPryString`

O módulo `PlSqlEngine` possui as classes `PSQLFlexLexer`, `PlSqlEngine`, `TableAcess` e `PlSqlFilter`. A função do módulo `PlSqlEngine` é analisar o código PL/SQL. Esta análise é realizada ao processar o comando `env.parse("código PL/SQL")` em um *template*. Para tanto, executa-se o método `OpParse` apresentado no quadro 29, no qual é instanciado um objeto de `PlSqlEngine`. O processamento de um código PL/SQL inicia com a análise léxica, realizada pela classe `PSQLFlexLexer`. Cada *token* encontrado é enviado ao analisador sintático para ser armazenado em uma estrutura de dados do tipo vetor. Para cada *token*, tem-se: o identificador do *token* (valor definido internamente pelo Flex), a regra sintática a partir da qual o *token* foi reconhecido, o *token* propriamente dito, o tipo do *token* (no caso de declaração de variáveis, parâmetros e funções) e seu antecessor. Neste processo, a árvore gramatical fica armazenada na classe `PlSqlEngine`.

```
_Template::Var LinkObjFormTemplate::OpParse(_Template::Var* iArgs,
                                            uint iNum) {
      _Template::Var aRes;
      PlSqlEngine * psql =  new PlSqlEngine(iArgs->str);
      aRes.type  = _Template::Var::eClass;
      aRes.point =  psql;
#if 1
      #if DEBUG
            int static seq =0;
            wxString fileName = m_BaseOutput+"\\debug_";
            fileName<<seq<<".xml";
            seq++;
            psql->Save(fileName);
      #endif
#endif
      return aRes;
}
```

Quadro 29 – Método `OpParse`

No quadro 30 é apresentado um trecho da gramática especificada para geração do analisador sintático.

```
//RULE 048
NumOrID:
  S_IDENTIFIER  {TK_RULE(48); TK_TYPE(TK_GET_TYPE);}
| S_PLUS        {TK_RULE(48); TK_INC_PARENTS} TypeNumber {TK_PARENTS(-2)}
| S_MINUS       {TK_RULE(48); TK_INC_PARENTS} TypeNumber {TK_PARENTS(-2)}
| TypeNumber
;

//RULE 079
IfStatement:
 K_IF   {TK_RULE(79); TK_INC_PARENTS} PlSqlExpression {TK_PARENTS(-2);}
 K_THEN {TK_RULE(79); TK_INC_PARENTS}
     SequenceOfStatements {TK_PARENTS(-6)}
     ListElsIfStatement   {TK_PARENTS(-8)}
     ElseStatement        {TK_PARENTS(-6)}
 K_END {TK_RESET_PARENTS; TK_RULE(79);}
 K_IF  {TK_RESET_PARENTS; TK_RULE(79);} EndPLSQLStatement {TK_PARENTS(-16)};
```

Quadro 30 – Trecho da gramática PL/SQL para implementação do analisador sintático

Para facilitar a implementação do analisador sintático, foram definidas macros na linguagem C++ (quadro 30), sendo elas:

a) `TK_RULE`: determina a regra sintática do *token* reconhecido;

b) `TK_TYPE`: determina o tipo do *token* reconhecido;

c) `TK_GET_TYPE`: busca o tipo de uma variável, parâmetro ou função;

d) `TK_INC_PARENTS`: cria um novo valor para o atributo antecessor;

e) `TK_PARENTS`: reinicializa variável controladora do atributo antecessor, pois a análise de uma regra subseqüente pode ter alterado o valor, sendo necessário a restauração do antecessor quando finalizada a análise de regras de nível inferior. Esta restauração é realizada copiando o valor do antecessor, conforme deslocamento de regra passada como parâmetro;

f) `TK_RESET_PARENTS`: atribui novamente antecessor ao *token* atual.

Quando a ferramenta Converte Forms é executada em modo *debug*, é gerado um arquivo XML (quadro 31) com a estrutura armazenada na análise de cada código PL/SQL, a fim de ajudar a identificar a correta geração da árvore gramatical.

```
<token id =366 value="nreal" data_type=3 parents=130 rule=74>
   <token id =372 value=":=" data_type=0 parents=137 rule=84>
      <token id =381 value="(" data_type=0 parents=138 rule=55>
         <token id =366 value="nint" data_type=2 parents=139 rule=55>
            <token id =385 value="+" data_type=0 parents=140 rule=54>
               <token id =364 value="5" data_type=2 parents=141 rule=49>
               </token>
            </token>
            <token id =380 value=")" data_type=0 parents=140 rule=55>
            </token>
         </token>
      </token>
      <token id =379 value=";" data_type=0 parents=138 rule=84>
      </token>
   </token>
</token>
```

Quadro 31 – Arquivo XML contendo a árvore gramatical

Quando há necessidade de realizar um filtro na árvore gramatical com o intuito de acessar apenas determinados *tokens*, este é realizado pelo método `select` que recebe como parâmetro um objeto de `PlSqlFilter` e retorna um objeto da classe `TableAcess`, que nada mais é que uma classe que armazena os índices válidos para o filtro. Por este motivo, a classe `TableAcess` foi implementada como `friend` de `PlSqlEngine` para ter acesso direto ao vetor de *tokens*. A classe `TableAcess` permite que novos filtros sejam aplicados com base nos índices obtidos. No quadro 32 é apresentado o método `select` da classe `PlSqlEngine`. O objeto de `PlSqlFilter` é responsável por verificar se determinado *token* atende ao filtro solicitado (trecho 1). Em caso afirmativo, é adicionado o índice para acesso ao *token* (trechos 1 e 2).

```
BaseAcess * PlSqlEngine::Select(PlSqlFilter * filter) {
	TableAcess * acess = new TableAcess(this);

	if (filter->m_Type != eRule) {
		for (unsigned int i=0; i<m_ArryToken.Count();i++) {
			if (filter->Cmp(m_ArryToken[i]))                    01
				acess->m_IndexAcess.Add(i);
		}
	}
	else {
		bool FlgNotFound;
		unsigned int r;
		for (unsigned int i=0; i<m_ArryToken.Count();i++) {
			FlgNotFound = true;
			if (filter->Cmp(m_ArryToken[i])) {                  01
				if (filter->m_FlgNot)
					acess->m_IndexAcess.Add(i);
				FlgNotFound =false;
			}
			else {
				r =m_ArryToken[i]->Parents;
				while (r!=INVALID_PARENTS) {
					if (filter->Cmp(m_ArryToken[r])) {  01
						if (filter->m_FlgNot)
							acess->m_IndexAcess.Add(i);
						FlgNotFound =false;
						break;
					}
					r = m_ArryToken[r]->Parents;
				}
			}
			if (FlgNotFound && !filter->m_FlgNot)               02
				acess->m_IndexAcess.Add(i);
		}
	}
	return acess;
}
```

Quadro 32 - Método `select` da classe `PlSqlEngine`

Os filtros válidos são apresentados no quadro 33.

FILTRO	DESCRIÇÃO	EXEMPLO
`T`	Filtra a árvore gramatical retornando os *tokens* do(s) tipo(s) indicado(s). No exemplo, retorna a lista com os *tokens* do tipo `ident` e `if`.	`T: ident, if`
`R`	Filtra a árvore gramatical retornando os *tokens* cujas regras sejam do(s) tipo(s) indicado(s). No exemplo, retorna a lista com os *tokens* cujas regras sejam `parameter`.	`R: parameter`
`V`	Filtra a árvore gramatical retornando os *tokens* cujos valores sejam do(s) tipo(s) indicado(s), sendo que cada valor deve estar entre @. No exemplo, retorna a lista com os *tokens* cujos valores sejam `str` ou `begin`.	`V: @STR@, @BEGIN@`
`D`	Filtra a árvore gramatical retornando os *tokens* cujos valores sejam do(s) tipo(s) de dados indicado(s). No exemplo, retorna a lista com os *tokens* cujos tipos de dados sejam `real` ou `date`.	`D: treal, tdate`
`~`	Símbolo utilizado para a negação do filtro, ou seja, retorna todos os *tokens* exceto aqueles que atendem o filtro.	`~R: parameter`

Quadro 33 – Filtros válidos para o método `Select`

3.6.3 Operacionalidade da implementação

Esta seção apresenta a operacionalidade da ferramenta. A interface da ferramenta Converte Forms está dividida em quatro áreas principais (figura 12), que são:

a) barra de ferramentas (item 01);
b) área de trabalho: possui os projetos carregados e seus arquivos (*templates* e Oracle Forms) (item 02);
c) área de edição: contém os editores de *template* (item 03);
d) área de mensagens (item 04).

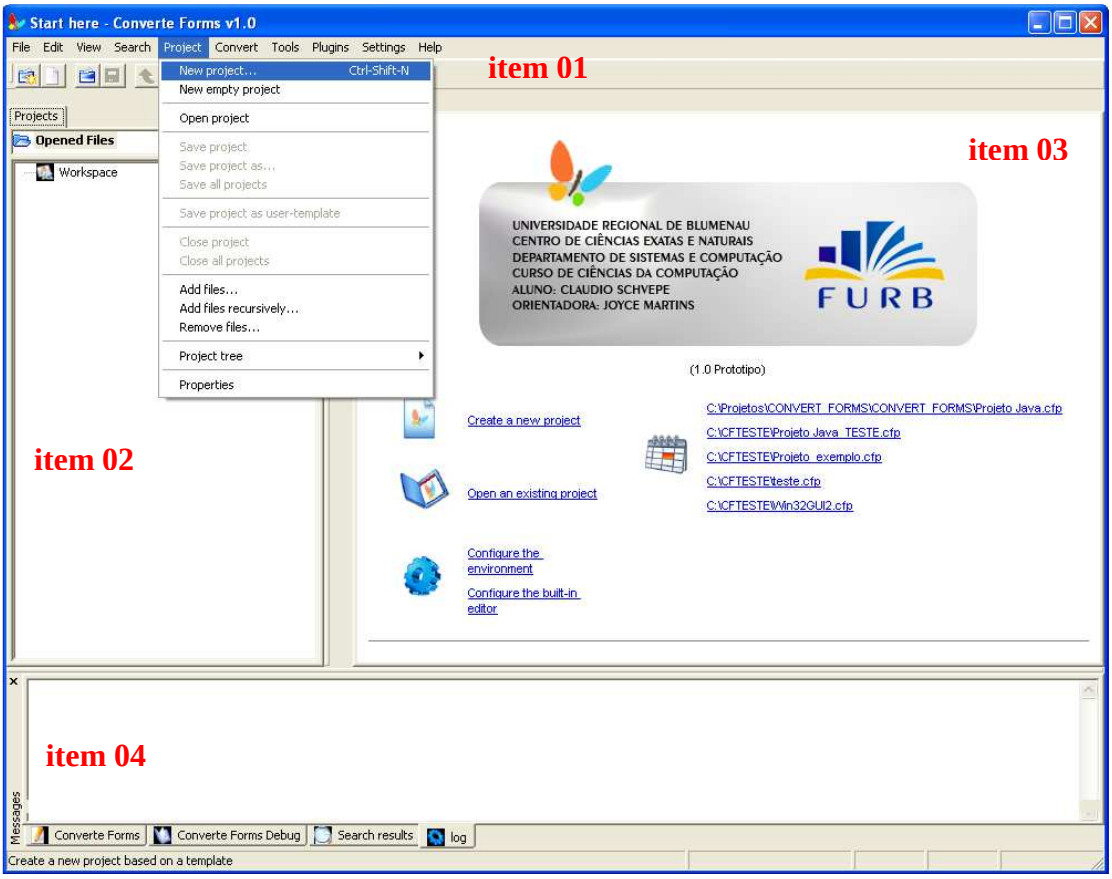


Figura 12 – A ferramenta Converte Forms: janela principal

A ferramenta Converte Forms possui opção para criar um novo projeto (figura 12). Ao ser selecionada a opção `New project`, nova janela é aberta (figura 13). Deve-se selecionar a categoria e pressionar o botão `Create`.

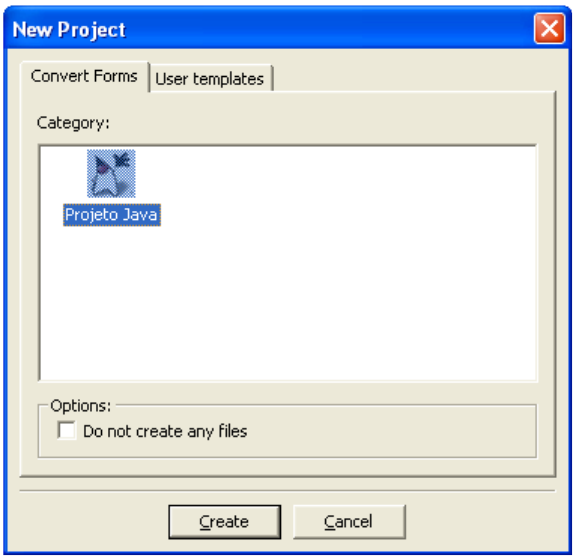


Figura 13 – Criando um projeto

Em seguida, conforme figura 14, é necessário informar o nome do projeto a ser criado e o diretório onde o mesmo será salvo.

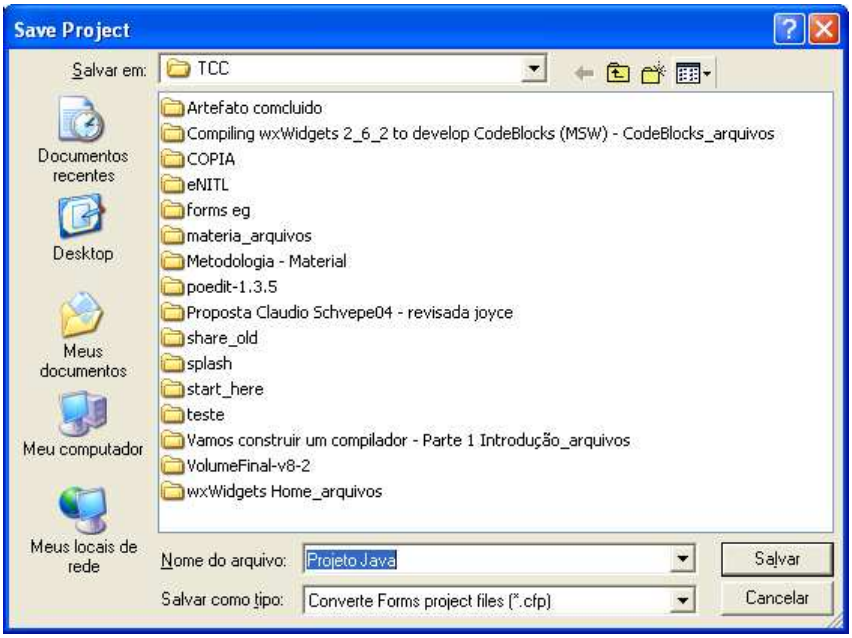


Figura 14 – Salvando um projeto

Uma vez criado o projeto, é possível alterar as propriedades do mesmo. Para tanto, seleciona-se um projeto na área de trabalho e clica-se com botão direito do *mouse* sob nome do projeto selecionado. Será apresentado um menu (figura 15). Deve ser selecionada a opção `Properties`.

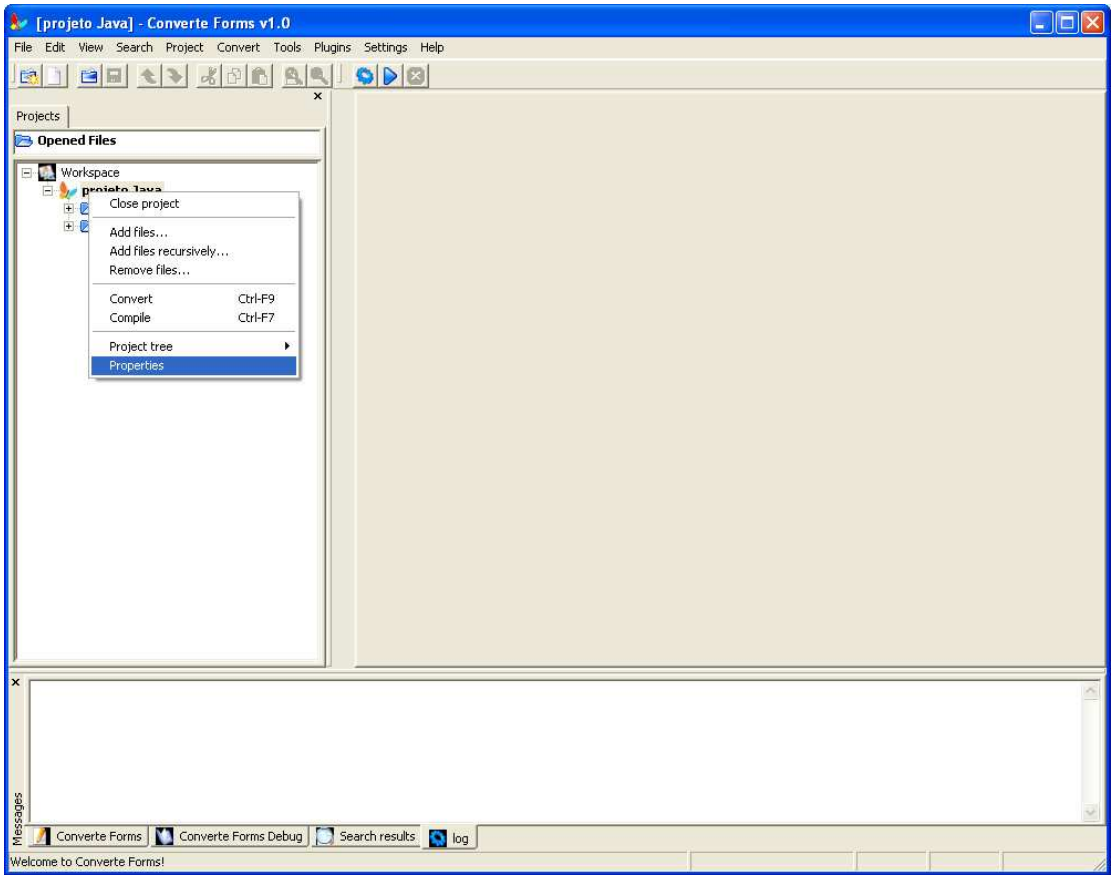


Figura 15 – Selecionado um projeto

As propriedades do projeto selecionado são apresentadas na janela da figura 16. É

possível alterar: o nome projeto, o *template* inicial que será usado como modelo para gerar o código e o diretório raiz onde os arquivos gerados serão gravados. Caso o diretório não possua o caminho completo, considera-se o caminho a partir do diretório onde o projeto foi salvo.

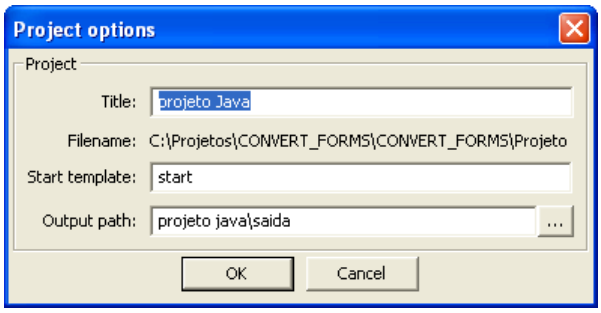


Figura 16 – Alterando as propriedades de um projeto

Em seguida, através da opção `Add files` do menu apresentado na figura 15, adicionam-se ao projeto os arquivos a serem convertidos. Na figura 17 é apresentada a relação de arquivos existentes no diretório corrente com extensão `FMB`, `MMB` ou `PLL`. O usuário deve selecionar o arquivo desejado e pressionar o botão `Abrir`. A figura 18 apresenta o projeto com os arquivos adicionados, os quais podem ser convertidos.

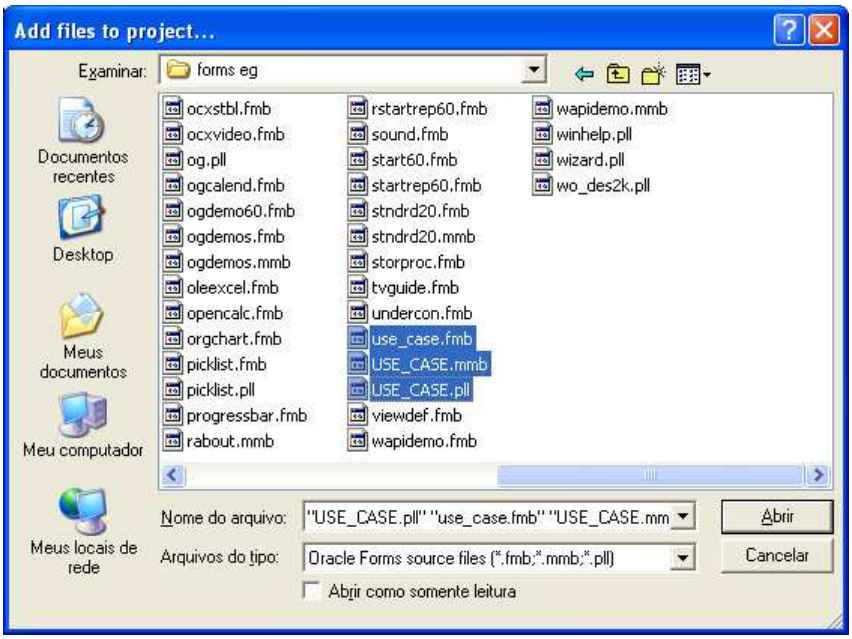


Figura 17 – Adicionado arquivos a um projeto

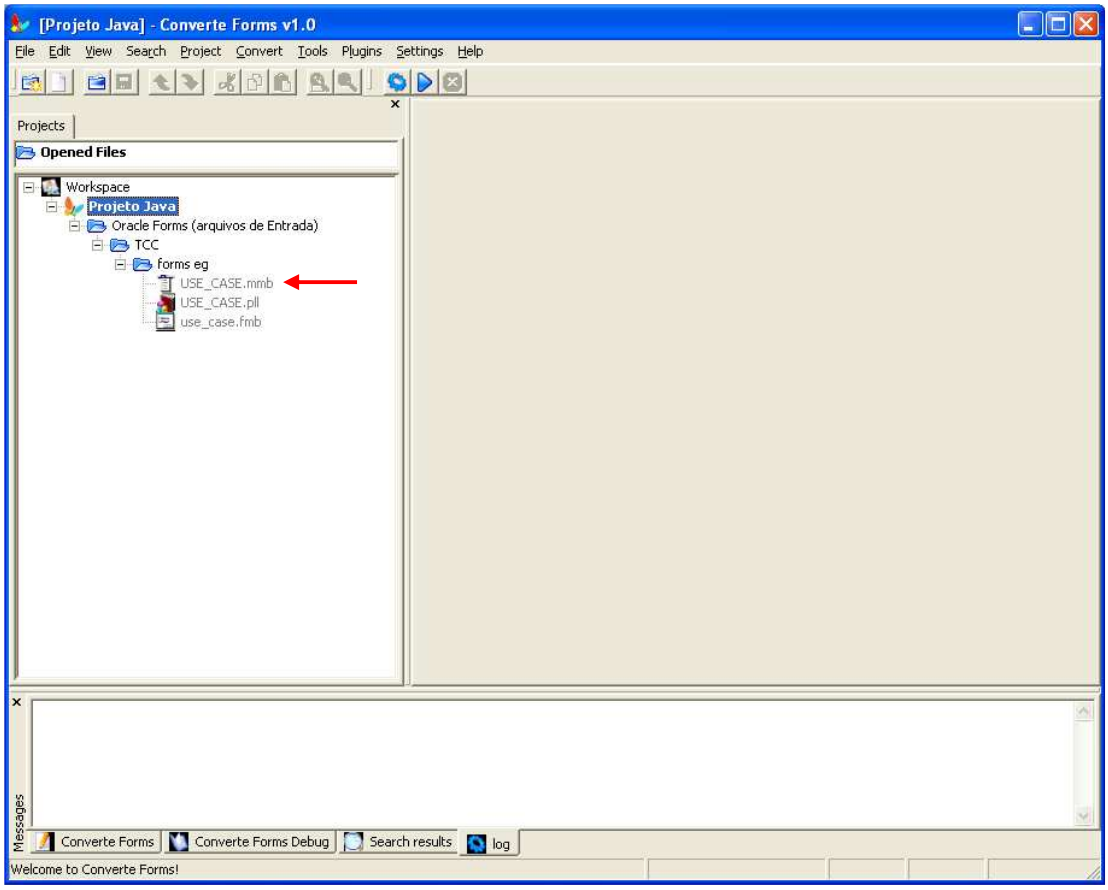


Figura 18 – Projeto com arquivos adicionados

Após adicionados os arquivos ao projeto, deve-se especificar os *templates* a serem usados no processo de conversão. Para tanto, o usuário deve selecionar o *template* associado a um projeto, como mostra a figura 19, na área de trabalho, sendo possível editar o *template* selecionado.

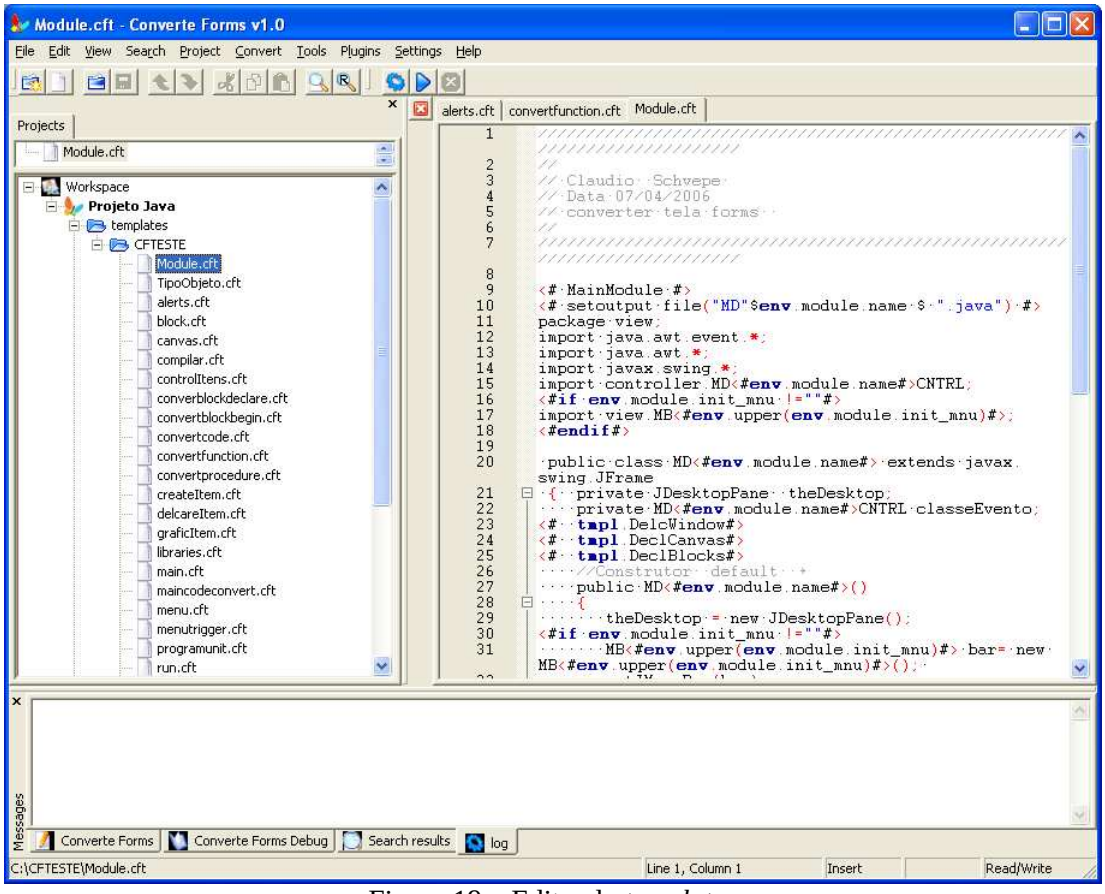


Figura 19 – Editando *templates*

O processo de conversão inicia quando for pressionado o botão indicado pela flecha na figura 20. Nesse caso, serão convertidos todos os arquivos adicionados ao projeto selecionado. Também é possível converter um único arquivo por vez. Para tanto, deve-se: selecionar o arquivo desejado, acessar o menu do arquivo clicando com botão direito do *mouse* sob nome do arquivo, selecionar a opção para conversão. Na figura 20 é possível observar as mensagens geradas ao longo do processo de conversão na guia `log` da área de mensagens.

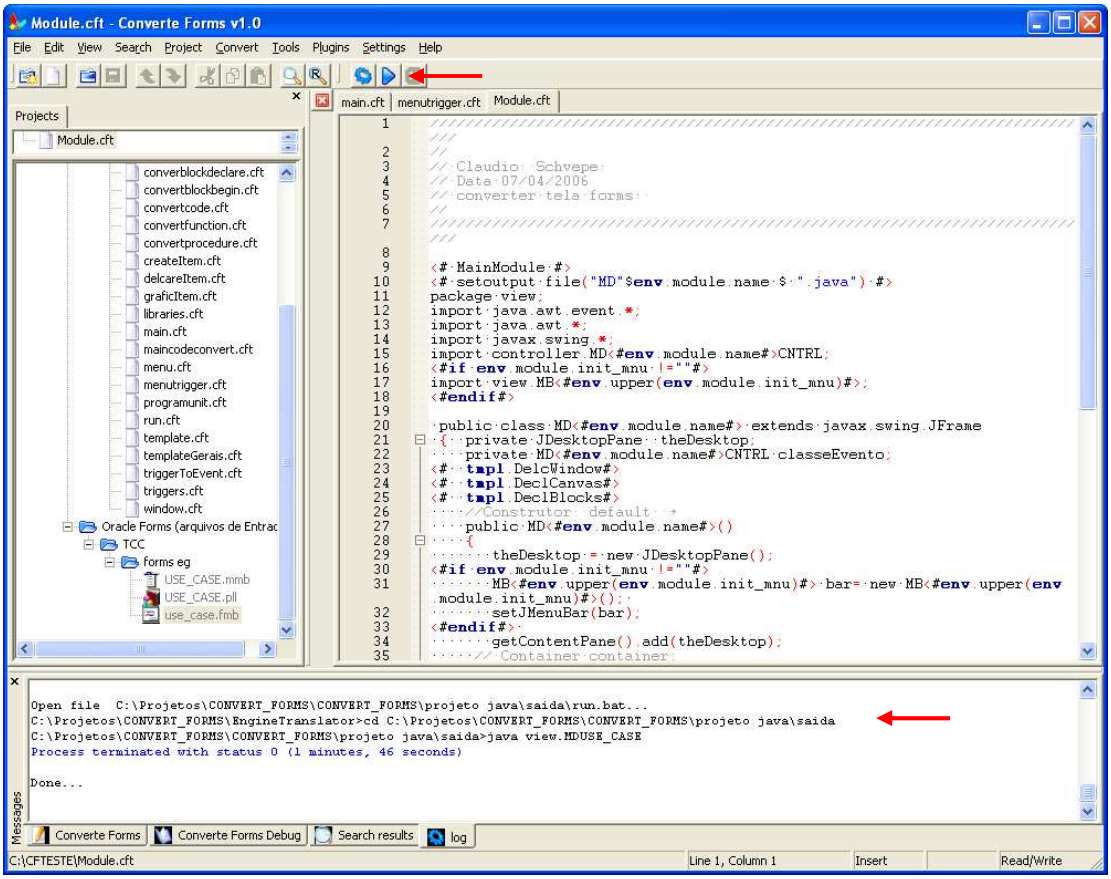


Figura 20 – Convertendo um arquivo

Por fim, na figura 21 é apresenta a interface de uma aplicação Oracle Forms e na figura 22 a interface da aplicação convertida para a linguagem Java.

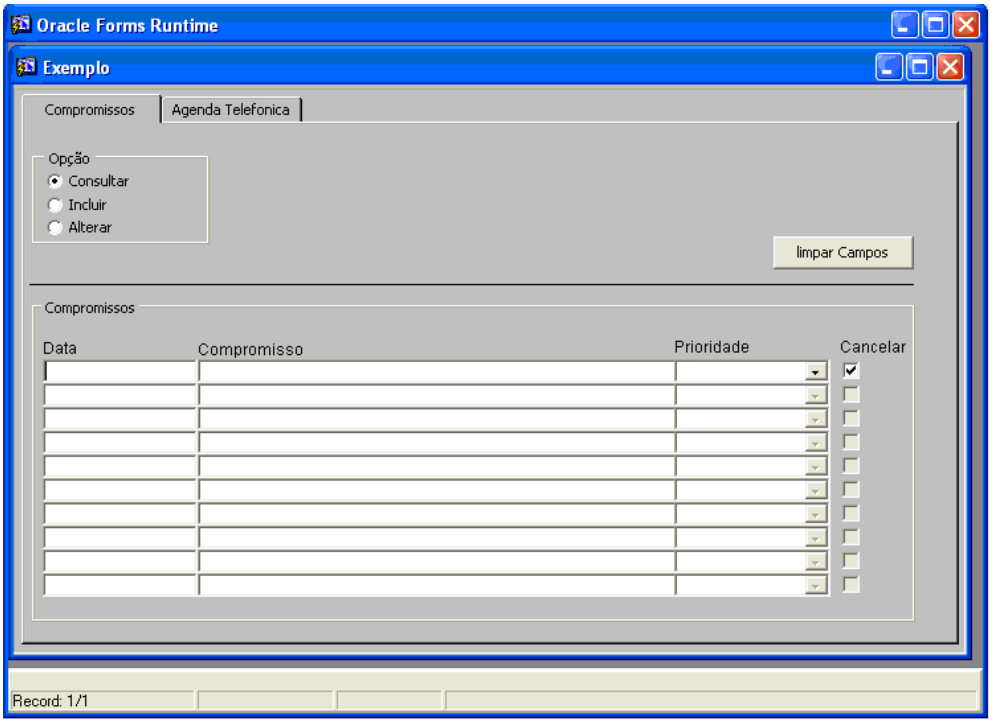


Figura 21 – Interface de uma aplicação Oracle Forms

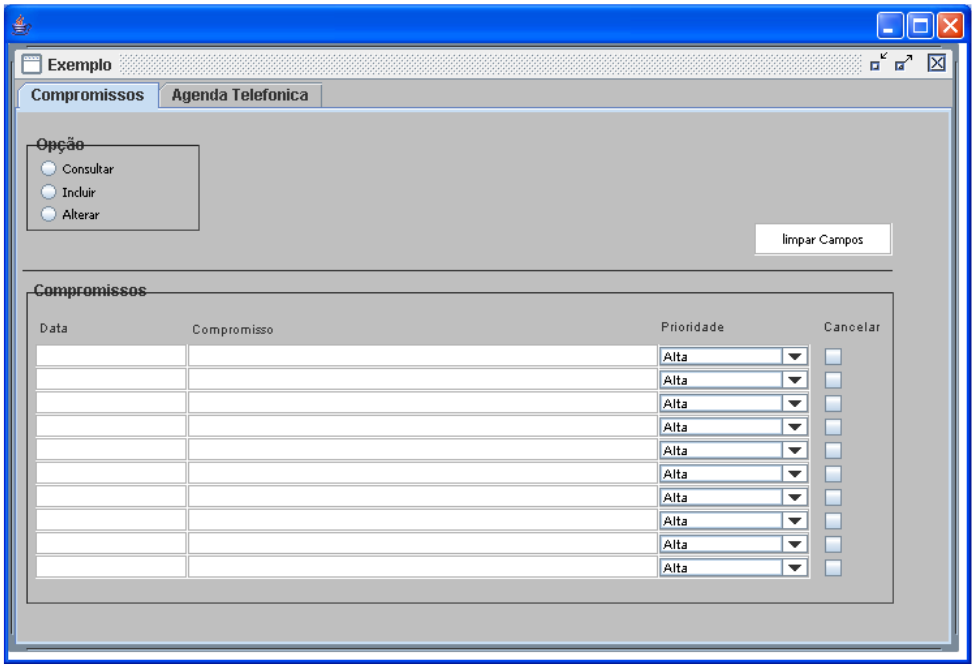


Figura 22 – Interface da aplicação convertida para linguagem Java

3.7 RESULTADOS E DISCUSSÃO

A ferramenta proposta não atendeu completamente os requisitos identificados uma vez que faltou a elaboração de *templates* para a conversão de chamadas de funções com parâmetros. Além disso, as funções nativas do Oracle Forms não foram convertidas para Java uma vez que estas funções são baseadas em nomes e Java utiliza objetos, sendo necessário implementar *templates* para suportar esta conversão.

Não foram executadas traduções com as ferramentas correlatas mencionadas na seção 2.8. Mas, com base na descrição das características apresentadas pelos seus fabricantes, foi possível estabelecer uma comparação entre as mesmas e a ferramenta desenvolvida. Vale ressaltar que Converte Forms é a única que utiliza *templates* para a geração do código de saída. Tem-se ainda que:

a) com exceção de Form2Net, todas as ferramentas utilizam no código gerado apenas API *open source*;
b) Exodus e SwisSQL prometem converter todo o código PL/SQL, sendo que Converte Forms converte um subconjunto do PL/SQL já detalhado nas seções anteriores;
c) SwisSQL é a única ferramenta que converte somente o código PL/SQL. As

demais convertem também componentes visuais. Observa-se que Converte Forms não converte todos componentes visuais do Oracle Forms, restando converter apenas sete componentes (`Hierarchical Tree`, `AtiveX Control`, `OLE Container`, `Sound`, `User Área`, `VBX Control`, `Bean Area`);

d) quanto ao suporte a JDBC e ao padrão de desenvolvimento J2EE, apenas Exodus menciona a sua utilização;

e) com exceção de Form2Net, todas as ferramentas geram código para a linguagem Java.

O quadro 34 sintetiza a comparação realizada.

CARACTERÍSTICA	Exodus	SwisSQL	Forms2Net	Converte Forms
código gerado baseado em API *open source*	sim	sim	não	sim
suporte a todas as construções PL/SQL	sim	sim	não	instruções PL
conversão de componentes visuais	sim	não	sim	sim
migração para J2EE	sim	não mencionado	não	não
suporte a JDBC	sim	não mencionado	não mencionado	não
linguagem alvo	Java	Java	ASP.NET	Java
uso de *templates* para a geração de código	não	não	não	sim

Quadro 34 – Comparação entre as ferramentas

4 CONCLUSÕES

Verificou-se que a reengenharia fornece métodos menos agressivos à evolução de sistemas legados, como a tradução de código. Como este método é muito demorado e seu custo elevado, fica desaconselhada qualquer tentativa de tradução de sistemas legados sem auxílio de ferramentas para automatizar o processo. Portanto, para automatizar o processo de tradução, deve-se construir geradores de código como implementado nesse trabalho.

O desenvolvimento da ferramenta envolveu a análise das diferenças entre as linguagens Oracle Forms e Java, a implementação dos analisadores léxico e sintático para analisar o código PL/SQL e a utilização de *templates* para gerar o código de saída.

A utilização de motor de *templates* tornou a ferramenta flexível permindo que apenas com a alteração dos *templates* seja possível migrar aplicações de Oracle Forms para outras linguagens. Observa-se que apesar do motor de *templates* eNITL possuir *bugs*, mostrou-se adequado para o desenvolvimento da ferramenta proposta. No entanto, a correção desses *bugs* atrasou o desenvolvimento da ferramenta, já que foi necessário o estudo do código do motor de *templates* para corrigi-los.

A Forms API mostrou-se adequada para a manipulação dos arquivos Oracle Forms, porém a documentação da API não abrange todas as informações necessárias para sua completa utilização. Além disso, há inconsistências entre a implementação e documentação. Por este motivo, não foi possível acessar todas as propriedades dos arquivos como, por exemplo, `coordinate system` na qual é definida a unidade de medida utilizada no módulo `Forms`. Também não foi possível exportar imagens armazenadas nos arquivos Oracle Forms.

Com a ferramenta Converte Forms no atual estado de desenvolvimento é possível automatizar parte do processo de conversão de aplicativos desenvolvidos em Oracle Forms para Java. Registra-se também que além de utilizada para conversão de aplicações Oracle Forms, pode-se utilizar Converte Forms para gerar documentação de aplicações Oracle Forms bastando implementar *templates* para esta tarefa.

As restrições identificadas na ferramenta são: o sistema de medidas deve ser *pixel*, ou seja, é necessário atribuir `pixel` à propriedade `coordinate system` do módulo `Forms` antes de converter os arquivos, e os arquivos Oracle Forms deverão conter somente as construções do subconjunto do código PL/SQL definido na implementação.

4.1 EXTENSÕES

Como extensões da ferramenta sugere-se:

a) efetuar a conversão dos demais componentes visuais e de todo o código PL/SQL, incluindo a tradução de código SQL;
b) implementar classes Java para prover comportamentos idênticos ao objeto `Block` do Oracle Forms, que é responsável por recuperar e manter os dados obtidos no banco de dados, bem como ligá-los aos componentes visuais;
c) implementar conectividade com banco de dados para as classes geradas;
d) permitir traduzir todas as funções nativas da linguagem Oracle Forms como, por exemplo, `set_item_property`, `set_window_property`. O Oracle Forms utiliza-se de nomes para acessar qualquer objeto, por este motivo é necessário criar funções para converter estes nomes em objetos ou implementar em Java as classes que permitam o acesso ao objeto através de nomes;
e) possibilitar o acesso à propriedade `coordinate system` dos módulos `Forms`, de forma que não seja necessário alterar o valor da mesma manualmente antes do processo de conversão;
f) exportar imagens contidas nos módulos `Forms`;
g) especificar *templates* para a geração de código para outra linguagem, como por exemplo, C#;
h) implementar *plugin* para gerar a especificação a partir dos arquivos do Oracle Forms, visando utilizá-la na geração de um código orientado a objeto mais próximo dos padrões de desenvolvimento.

REFERÊNCIAS BIBLIOGRÁFICAS

ADVENTNET. **SwisSQL**: Oracle to Java migration tool. [S.l.], 2006. Disponível em: <http://www.swissql.com/products/oracle-to-java/oracle-to-java.html>. Acesso em: 15 abr. 2006.

AHO, Alfred V.; SETHI, Ravi; ULLMAN, Jeffrey D. **Compiladores**: princípios, técnicas e ferramentas. Tradução Daniel de Ariosto Pinto. Rio de Janeiro: LTC, 1995.

APACHE SOFTWARE FOUNDATION. **Velocity.** [S.l.], 2005. Disponível em: <http://jakarta.apache.org/velocity/>. Acesso em: 03 jun. 2006.

ATX SOFTWARE. **Forms2Net**: from Oracle Forms to Microsoft .NET. [S.l.], 2006. Disponível em: <http://forms2net.atxsoftware.com/site/Welcome.do>. Acesso em: 01 abr. 2006.

BRECK, Liam. **eNITL:** the network improv template language. [S.l.], 1999. Documento eletrônico disponibilizado com a biblioteca eNITL.

CIPHERSOFT INC. **Main features of Exodus**. [S.l.], [2006?]. Disponível em: <http://www.ciphersoftinc.com/solutions/features.html>. Acesso em: 22 out. 2006.

DEITEL, Harvey M.; DEITEL, Paul J. **Java:** como programar. 4. ed. Tradução Carlos Arthur Lang Lisboa. Porto Alegre: Bookmam, 2003.

ESTES, Will (Ed.). **FLEX**: the fast lexical analyzer. [S.l.], 2006. Disponível em: <http://flex.sourceforge.net/>. Acesso em: 01 out. 2006.

FERNANDES, Lúcia. **Oracle 9i para desenvolvedores:** curso completo. Rio de Janeiro: Axcel Books, 2002.

FONSECA, Fabricio. **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.

FREE SOFTWARE FOUNDATION. **Bison**. [S.l.], 2006. Disponível em: <http://www.gnu.org/software/bison/>. Acesso em: 01 out. 2006.

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

HODGSON, Neil (Ed.). **Scintilla**: a free source code editing component for Win32 and GTK+. [S.l.], 2006. Disponível em: <http://scintilla.sourceforge.net/index.html>. Acesso em: 01 out. 2006.

JOU, Herng-Jeng. **eNITL**. [S.l.], 2000. Disponível em: <http://ceu.fi.udc.es/SAL/F/1/ENITL.html>. Acesso em: 03 jun. 2006.

KLÖSCH, René R. **Reverse engineering:** why and how to reverse engineer software. [Vienna], [1996?]. Disponível em: <http://www.infosys.tuwien.ac.at/staff/hg/papers/css96.pdf>. Acesso em: 06 jun. 2006.

KULANDAISAMY, Patrick D. J.; NAGARAJ, N. S.; THONSE, Srinivas. Representing procedural source in UML. In: WORKSHOP ON UML FOR ENTERPRISE APPLICATIONS: MODEL DRIVEN SOLUTIONS FOR THE ENTERPRISE, 3rd., 2002, San Francisco. **Proceedings**… [S.l.]: OMG, 2002. Não paginado. Disponível em: <www.omg.org/news/meetings/workshops/UML2002-Manual/04-2_Reverse_Engineering_Procedural_Code_using_UML.pdf>. Acesso em: 18 nov. 2006.

MANDRAVELLOS, Yiannis. **Code::Blocks Studio:** the open source, cross platform free C++ IDE. [S.l.], 2006. Disponível em: <http://www.codeblocks.org>. Acesso em: 01 out. 2006.

OLLIVIER, Kevin (Ed.). **wxWidgets**: cross-platform GUI library. [S.l.], [2006?]. Disponível em: <http://www.wxwidgets.org/>. Acesso em: 01 out. 2006.

ORACLE CORPORATION. **Using the Oracle Forms Application Programming Interface (API)**. [S.l.], 2000. Documento eletrônico disponibilizado com o Ambiente Oracle Forms Developer 6i.

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. Não paginado. Disponível em: <http://www.cin.ufpe.br/~sugarloafplop/final_articles/12_TB-REPP-Final.pdf>. Acesso em: 18 nov. 2006.

RAMANATHAN, S. **Grammar for PL/SQL inside Oracle*Forms 4.5**. [S.l.], 1997. Disponível em: <http://cobase-www.cs.ucla.edu/pub/javacc/plsql/FormsPlSqls.jj>. Acesso em: 25 nov. 2006.

ROCHA, Lucas. **APE:** plataforma para o desenvolvimento de aplicações web com PHP. [Salvador], 2005. Disponível em: <http://twiki.im.ufba.br/bin/view/Aside/ProjetoConclusaoDeCursoAPEMonografia#4_2_Motores_de_templates>. Acesso em: 01 abr. 2006.

REZENDE, Pedro A. D. **Elementos de um compilador**. [Brasília], [1998?]. Disponível em: <www.cic.unb.br/docentes/pedro/materia2.htm>. Acesso em: 03 abr. 2006.

SKROL, Jean B. **TinyButStrong**: template engine for pro and beginners for PHP. [S.l.], 2006. Disponível em: <http://www.tinybutstrong.com/>. Acesso em: 03 jun. 2006.

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

TAULLI, Tom; JUNG, David. **VB2Java.COM**: moving from Visual Basic to Java. [S.l.], 1997. Disponível em: <http://www.vb2java.com/about.html>. Acesso em: 04 jun. 2006.

THOMASON, Lee. **TinyXML.** [S.l.], 2006. Disponível em: <http://www.grinninglizard.com/tinyxml/index.html>. Acesso em: 01 out. 2006.

VBCONVERSIONS. **VB.NET to C# converter**: Visual Basic.NET to C# converter. [S.l.], 2006. Disponível em: <http://www.filesrepository.com/preview/vbconversions/vb-net-to-c-converter.html>. Acesso em: 03 jun. 2006.

APÊNDICE A – Lista de propriedades Oracle Forms em XML

No quadro abaixo é mostrado um trecho do arquivo `module_property.xml`, que contém a lista de propriedades, 500 no total, de um arquivo Oracle Forms. Este aquivo encontra-se no diretório `share\enginetranslator` partindo-se do diretório corrente da aplicação.

```
<?xml version="1.0"?>
<!DOCTYPE Convert_Forms - Property Forms Object>
<Convert_Forms_PropertyFormsObject>
	<Properties>
		<Property name="ATT_LIB" id=15 />
		<Property name="BACK_COLOR" id=25 />
		<Property name="CANVAS" id=40 />
		<Property name="ENABLED" id=140/>
		<Property name="FONT_NAM" id=153/>
		<Property name="FONT_SCALEABLE" id=154/>
		<Property name="FONT_SIZ" id=155/>
		<Property name="FONT_WGHT" id=158/>
		<Property name="FORE_COLOR" id=159/>
		<Property name="GRAPHIC" id=180/>
		<Property name="GRAPHICS_TYP" id=181/>
		<Property name="HEIGHT" id=190/>
		<Property name="ITEM" id=229/>
		<Property name="NAME" id=300/>
		<Property name="NEXT" id=302/>
		<Property name="WINDOW" id=530/>
	</Properties>
</Convert_Forms_PropertyFormsObject>
```

Quadro 35 – Lista de Propriedade Oracle Forms em XML

APÊNDICE B – Especificação da linguagem PL/SQL

Nos quadros 36 e 37 são mostradas, respectivamente, as especificações léxica e sintática da linguagem PL/SQL, usadas na implementação da ferramenta.

TOKEN	EXPRESSÃO REGULAR
palavras-chave	AND, AS, BEGIN, BETWEEN, BINARY_INTEGER, BOOLEAN, CHAR, CONSTANT, DATE, DECIMAL, DECLARE, DEFAULT, DELETE, DESC, DISTINCT, DO, ELSE, END, EXCEPTION, EXCEPTION_INIT, EXIT, FLOAT, FOR, FUNCTION, GLOBAL, IF, IN, INSERT, INTEGER, ITEM, IS, LIKE, LOCK, LOOP, NATURAL, NOT, NULL, NUMBER, OF, OR, OTHERS, OUT, PRAGMA, PROCEDURE, RAISE, READ, REAL, RETURN, REVERSE, SMALLINT, THEN, TO, VARCHAR2, VARCHAR, WHEN, WHERE, WHILE
constante inteira	[0-9]+
constante real	[0-9]* \. [0-9]+
constante literal	\' [^\']* \' (\' [^\'] \')*
identificador	[a-zA-Z]+[a-zA-Z0-9\$_]*
variáveis de ligação	\: [a-zA-Z]+[a-zA-Z0-9\$_]* \. [a-zA-Z]+[a-zA-Z0-9\$_]*
símbolos especiais	:= .. || ** != = <> > >= < <= ; ( ) , . + - * /
comentário de linha	\" [^\n\r\"]* \"
comentário de bloco	"/*" ([^*] | (*)+ [^* "/"])* (*)+ "/"

Quadro 36 – Especificação léxica da linguagem PL/SQL

Gramática (BNF)
<SequenceOfStatements>:: <SequenceOfStatements> <PLSQLStatement> | <PLSQLStatement> | <FunctionDeclaration> | <ProcedureDeclaration>
<PLSQLStatement>:: <NullStatement> | <BeginEndBlock> | <AssigSubCallStatement> | <IfStatement> | <ReturnStatement> | <ExitStatement> | <LoopStatement> | <RaiseStatement> | <DeclarationSection> <BeginEndBlock>
<NullStatement>:: NULL ;
<BeginEndBlock>:: BEGIN <SequenceOfStatements> <ExceptionStatement>
<ExceptionStatement>:: <ExceptionBlock> END <EndPLSQLStatement> | END <EndPLSQLStatement>
<ExceptionBlock>:: EXCEPTION <ExceptionHandler>;
<ExceptionHandler>:: WHEN <ExceptionIdentifier>
<ExceptionIdentifier>:: <ExceptionHandler><_ExceptionHandler> | <ExceptionOthers>
<_ExceptionHandler>:: WHEN <__ExceptionHandler> | ε
<__ExceptionHandler>:: <ExceptionHandler> <_ExceptionHandler> | <ExceptionOthers>
<ExceptionOthers>:: OTHERS THEN <SequenceOfStatements>
<ExceptionHandler>:: <identificador> <ExceptionORHandler> THEN <SequenceOfStatements>
<ExceptionORHandler>:: <_ExceptionORHandler> <_ExceptionHandler> | ε

```
<_ExceptionORHandler>::
                      OR <identificador> | <_ExceptionORHandler> OR <identificador>
<End_PLSQLStatement>:: <identificador> ; | ;
<FunctionDeclaration>:: FUNCTION <identificador> FunctionDeclarationParm>
<FunctionDeclarationParm>::
  ( <ParameterList> ) RETURN <TypeDeclaration> <FunctionDeclarationSpecification>
| RETURN  <TypeDeclaration> <FunctionDeclarationSpecification>
<FunctionDeclarationSpecification>:: IS <FunctionBody> | ;
<FunctionBody>::
           <DeclarationsList> <DeclarationsFunctionProcedureList> <BeginEndBlock>
<DeclarationsFunctionProcedure>:: <ProcedureDeclaration> | <FunctionDeclaration>
<DeclarationsFunctionProcedureList>:: <_DeclarationsFunctionProcedure> | ε
<_DeclarationsFunctionProcedure>::
   <DeclarationsFunctionProcedure>
| <_DeclarationsFunctionProcedure> <DeclarationsFunctionProcedure>
<ProcedureDeclaration>:: PROCEDURE <identificador> <ProcedureDeclarationParam>
<DeclarationsList >:: <_DeclarationsList> | ε
<_DeclarationsList>:: <Declarations> | <_DeclarationsList> <Declarations>
<Declarations>:: <IdentifierDeclaration> | <PragmaDeclaration>
<IdentifierDeclaration>:: <identificador> <_IdentifierDeclaration> ;
<PragmaDeclaration>:: PRAGMA _EXCEPTION_INIT( <NumOrID> , <NumOrID> );
<ProcedureDeclarationSpecification> :: ; | IS <ProcedureBody>
<ProcedureDeclarationParam> ::
   (<ParameterList>) <ProcedureDeclarationSpecification>
|  <ProcedureDeclarationSpecification>
<ProcedureBody> ::
          <DeclarationsList > <DeclarationsFunctionProcedureList > <BeginEndBlock>
<ParameterList> :: <Parameter> | <ParameterList>,  <Parameter>
<Parameter>:: <identificador> [IN | OUT] <TypeParamDeclaration> <OptionDefault>
<TypeParamDeclaration>::CHAR | VARCHAR | VARCHAR2
                       | INTEGER | NUMBER | NATURAL | REAL | FLOAT | DATE
                       | BINARY_INTEGER | BOOLEAN
<TypeDeclaration>:: <BasicDataTypeDeclaration>
<OptionDefault>:: := <PlSqlExpression> | DEFAULT <PlSqlExpression> | ε
<_IdentifierDeclaration>:: <ConstantDeclaration>
                         | <VariableDeclaration>
                         | <ExceptionDeclaration>
<ConstantDeclaration>::
                  CONSTANT <TypeDeclaration> [NOT] NULL  <AttributionDeclaration> ;
<BasicDataTypeDeclaration>:: CHAR | VARCHAR | VARCHAR2
                       | INTEGER | NUMBER | NATURAL | REAL | FLOAT | DATE
                       | BINARY_INTEGER | BOOLEAN | ITEM
<LengthTypeScale>:: ( <inteiro> <PrecisionType> ) | ε
<LengthType>::  ( <inteiro> ) | ε
<PrecisionType> :: , <inteiro> | ε
<LengthTypeRequested>:: ( <inteiro> )
<VariableDeclaration >::  <TypeDeclaration> [NOT] NULL  <AttributionDeclaration>
<ExceptionDeclaration>:: EXCEPTION
<NumOrID>:: <identificador> | + <TypeNumber> | - <TypeNumber> | <TypeNumber>
<TypeNumber>:: <real> | <inteiro>
<AttributionDeclaration>:: := <PlSqlExpression> | DEFAULT  <PlSqlExpression> | ε
<PlSqlExpression>:: <PlSqlAndExpression> <_PlSqlExpression>
<PlSqlAndExpression>:: <PlSqlUnaryLogicalExpression> <_PlSqlAndExpression>
<_PlSqlAndExpression>:: AND <PlSqlUnaryLogicalExpression> | ε
```

```
<PlSqlUnaryExpression>::
  +<PlSqlPrimaryExpression> | -<PlSqlPrimaryExpression> | <PlSqlPrimaryExpression>
<PlSqlPrimaryExpression>:: NULL
                          | <identificador> <_PlSqlPrimaryExpression>
                          | <TypeNumber>
                          | <caracteres literal>
                          | <bind>
                          | GLOBAL  .  <identificador>
                          | ( <PlSqlExpression> )
<PlSqlUnaryLogicalExpression>:: [NOT] <PlSqlRelationalExpression>
<_PlSqlPrimaryExpression>:: (<Arguments>) | <ItemcompositionAssignmentStatement>
<Arguments>:: <PlSqlExpressionList>
<ItemcompositionAssignmentStatement>:: . <identificador>
<PlSqlExpressionList>:: <PlSqlExpression>
                       | <PlSqlExpressionList> ,  <PlSqlExpression>
<PlSqlRelationalExpression>::
                              <PlSqlSimpleExpression> <_PlSqlRelationalExpression>;
<_PlSqlRelationalExpression>::  <Relop> <PlSqlSimpleExpression>
                              |  <PlSqlInClause>
                              |  <PlSqlBetweenClause>
                              |  <PlSqlLikeClause>
                              |  <sNullClause>
                              |  ε
<Relop>:: = | <> | != | < | <= | > | >=
<PlSqlInClause>:: [NOT] IN ( <PlSqlExpressionList> )
<PlSqlSimpleExpression>::
                           <PlSqlMultiplicativeExpression> <_PlSqlSimpleExpression>;
<_PlSqlSimpleExpression>:: +  <PlSqlMultiplicativeExpression>
                           |  - <PlSqlMultiplicativeExpression>
                           |  || <PlSqlMultiplicativeExpression>
                           |  ε
<_PlSqlExpression>:: OR <PlSqlAndExpression> | ε
<PlSqlMultiplicativeExpression>::
                         <PlSqlExpotentExpression> _PlSqlMultiplicativeExpression ;
<PlSqlExpotentExpression>::
    <PlSqlUnaryExpression> | <PlSqlExpotentExpression> **  <PlSqlUnaryExpression>
<_PlSqlMultiplicativeExpression>::
                  <__PlSqlMultiplicativeExpression> | <PlSqlExpotentExpression> | ε
<__PlSqlMultiplicativeExpression>::
  <___PlSqlMultiplicativeExpression>
| <__PlSqlMultiplicativeExpression> <___PlSqlMultiplicativeExpression>
<___PlSqlMultiplicativeExpression>::
                             <PlSqlExpotentExpression> | / <PlSqlExpotentExpression>
<PlSqlBetweenClause>::
                  [NOT] BETWEEN <PlSqlSimpleExpression> AND <PlSqlSimpleExpression>
<AssigSubCallStatement>::
   <identificador> <ItemcompositionAssignmentStatement> AssigSubCall
|  <bind> := <PlSqlExpression>
<AssigSubCall>:: <SubroutineCallArguments> | <AssignmentStatement>
<PlSqlLikeClause>:: [NOT] LIKE <PlSqlSimpleExpression>
<sNullClause>:: IS  [NOT]  K_NULL
<IfStatement>:: IF <PlSqlExpression> THEN <SequenceOfStatements>
                <ListElsIfStatement>
                <ElseStatement>
                END IF <EndPLSQLStatement>
```

```
<ListElsIfStatement>:: <_ListElsIfStatement> | ε
<_ListElsIfStatement>:: <_ListElsIfStatement> <ElsIfStatement> | <ElsIfStatement>
<ElsIfStatement>:: ELSIF <PlSqlExpression> THEN <SequenceOfStatements>
<ElseStatement>:: ELSE <SequenceOfStatements> | ε
<AssignmentStatement>:: := <PlSqlExpression> ;
<LoopStatement>:: <NormalLoop> | <WhileLoop> | <NumericForLoop>
<WhileLoop>:: WHILE <PlSqlExpression> <NormalLoop>
<NormalLoop>:: LOOP <SequenceOfStatements> END LOOP <EndPLSQLStatement>
<NumericForLoop>:: FOR <identificador>
                   IN [REVERSE] <PlSqlSimpleExpression>..<PlSqlSimpleExpression>
                   <NormalLoop>
<RaiseStatement>:: RAISE <RaiseStatementIdentifier>
<RaiseStatementIdentifier>:: <identificador> | ε
ReturnStatement>:: RETURN <ReturnStatement>Type ;
<ReturnStatementType>:: <PlSqlExpression> | ε
<ExitStatement>:: EXIT <ExitWhenStatement> ;
<ExitWhenStatement>:: <identificador> <_ExitWhenStatement> | <_ExitWhenStatement>
<_ExitWhenStatement>:: WHEN <PlSqlExpression> | ε
<SubroutineCallArguments>:: ( <Arguments> ); | ε
<DeclarationSection>::
DECLARE <DeclarationsList> <DeclarationsFunctionProcedureList>
```

Quadro 37 – Especificação sintática da linguagem PL/SQL

APÊNDICE C – Lista de cores Oracle Forms em XML

No quadro 38 é mostrado um trecho do arquivo `palettecolor.xml` que contém a lista de cores de um arquivo Oracle Forms a fim de ilustrar a composição do arquivo. Este aquivo encontra-se no diretório `share\enginetranslator` partindo se do diretório corrente da aplicação..

```
<?xml version="1.0"?>
<!DOCTYPE Convert_Forms - Color Palette>
<Convert_Forms_ColorPalette>
	<PaletteColor>
		<Color name="r0g100b50" red=0 green=255 blue=127/>
		<Color name="custom8" red=255 green=255 blue=255/>
		<Color name="r75g100b75" red=191 green=255 blue=191/>
		<Color name="r88g75b50" red=223 green=191 blue=127/>
		<Color name="r50g75b88" red=127 green=191 blue=223/>
		<Color name="r88g50b100" red=223 green=127 blue=255/>
		<Color name="r25g0b75" red=63 green=0 blue=191/>
		<Color name="r25g50b50" red=63 green=127 blue=127/>
	</PaletteColor>
</Convert_Forms_ColorPalette>
```

Quadro 38 – Lista de cores Oracle Forms em XML