UNIVERSIDADE REGIONAL DE BLUMENAU

CENTRO DE CIÊNCIAS EXATAS E NATURAIS

CURSO DE CIÊNCIAS DA COMPUTAÇÃO – BACHARELADO

AVALIAÇÃO DA QUALIDADE DO CÓDIGO FONTE ESCRITO EM PL/SQL

GISELLE MAFRA SCHLOSSER

BLUMENAU
2009

2009/1-08

GISELLE MAFRA SCHLOSSER

AVALIAÇÃO DA QUALIDADE DO CÓDIGO FONTE ESCRITO EM PL/SQL

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. Everaldo Artur Grahl, Mestre – Orientador

BLUMENAU
2009

2009/1-08

AVALIAÇÃO DA QUALIDADE DO CÓDIGO FONTE

ESCRITO EM PL/SQL

Por

GISELLE MAFRA SCHLOSSER

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

Presidente:	Prof. Everaldo Artur Grahl, Mestre – Orientador, FURB
Membro:	Prof(a). Fabiane Barreto Vavassori Benitti, Doutora - FURB
Membro:	Prof. Marcel Hugo, Mestre – FURB

Blumenau, 08 de julho de 2009.

Dedico este trabalho a todos que de alguma forma me incentivaram, compreenderam e apoiaram durante a sua elaboração, especialmente meu esposo Maycon.

AGRADECIMENTOS

A Deus, pela força para superar todos os obstáculos.

À minha família, que sempre me incentivou e me acompanhou nesta caminhada.

A meu marido, pela força e compreensão, principalmente em relação a minha ausência durante a confecção do trabalho.

Aos meus amigos, pelo carinho, incentivo e ajuda.

Ao meu orientador, Everaldo Artur Grahl, por ter acreditado na conclusão deste trabalho.

O segredo é não correr atrás das borboletas... É cuidar do jardim para elas venham até você.

Mário Quintana

RESUMO

O presente trabalho demonstra o desenvolvimento de uma ferramenta para apoio a avaliação do código fonte escrito em PL/SQL. A ferramenta utiliza arquivos de programas desenvolvidos em PL/SQL, servindo de entrada para a ferramenta avaliar se a codificação está dentro dos padrões estabelecidos. Para isto, foram utilizados analisadores léxico, sintático e semântico para a análise de arquivos PL/SQL. A ferramenta disponibiliza telas para o cadastramento dos padrões de codificação desejados, bem como o cadastro de prefixos e também de um glossário de termos. Estatísticas dos principais erros gerados podem ser visualizadas por um relatório de inconsistências.

Palavras-chave: Padrões de codificação. Código fonte. Linguagem PL/SQL.

ABSTRACT

The present work shows the development of a tool for evaluation support of a source code written in PL/SQL. The tool uses files from softwares developed in PL/SQL, working as an entrance to the tool to evaluate if the coding is in the established default. Analytical lexicon, syntactic and semantic were used for the analysis of the programs. This tool provides screens to register the desired default, and the prefixes registration and also a terms glossary. The Statistics of main errors that were generated, can be viewed by a inconsistencies the report.

Key-words: Standards of coding. Source code. PL/SQL language.

LISTA DE ILUSTRAÇÕES

LISTA DE SIGLAS

BLOB – *Binary Large Object*

CLOB - *Character Large Object*

GLC – Gramática Livre de Contexto

NLOB - *National Character Large Object*

PL - *Procedural Language*

RF – Requisito funcional

RNF – Requisito Não funcional

SQL - *Structured Query Language*

UML - *Unified Modeling Language*

SUMÁRIO

1 INTRODUÇÃO

Desenvolver um software com qualidade não é mais um fator de diferenciação no mercado, e sim, condição essencial para empresas e profissionais serem bem-sucedidos. Segundo Staa (2000, p. 88), para se garantir um nível de qualidade satisfatório é necessário atuar sobre o processo de desenvolvimento, onde os programas devem estar corretos na construção e que sejam de fácil compreensão e manutenção.

> Padrões servem a dois propósitos. Eles ajudam os desenvolvedores a desenvolverem artefatos de qualidade elevada e fornecem instrumentos aos inspetores de qualidade para que possam controlar a qualidade de forma sistemática e objetiva. Estabelecem, assim, uma interface racional entre os desenvolvedores e os controladores da qualidade, contribuindo para eliminar o costumeiro jogo de empurra que ocorre sempre que alguma coisa não esteja a contendo. (STAA, 2000, p. 88).

Tentando auxiliar as empresas no processo de validação do desenvolvimento de um software, surgiu a idéia de desenvolver uma ferramenta que valide o código fonte escrito na linguagem PL/SQL (*Procedural Language / Structured Query Language*). Para isso são analisados padrões de codificação, estes que devem ser parametrizados na ferramenta, assim como um glossário de termos com abreviações de palavras, que devem ser utilizados em variáveis, nomes de tabelas e nomes de campos. Optou-se por avaliar códigos escritos em PL/SQL, devido ao fato de várias empresas da região de Blumenau utilizarem esta linguagem.

Os padrões adotados neste trabalho foram coletados em algumas empresas da região que utilizam a linguagem PL/SQL. Além disso foram pesquisados padrões existentes na literatura. Estes padrões são por exemplo, tamanho máximo para nome de tabelas, nomenclatura válida para atributos, variáveis, nomes de tabelas, pacotes, procedimentos, funções e cursores. Outro padrão adotado é a descrição de comentários e indentação.

A ferramenta Oracle Forms 6i, que produz aplicações *on-line* baseadas em telas e janelas, utiliza blocos PL/SQL. Nesta ferramenta existe uma opção para gerar toda a aplicação desenvolvida num arquivo (com extensão .TXT) no diretório em que se localiza o formulário da aplicação. Com isso, a ferramenta tem uma opção para receber este arquivo gerado e quebrá-lo em vários blocos PL/SQL, ou seja, irá gerar um arquivo para cada procedimento, função ou pacote encontrado, para que o programa não necessite avaliar códigos desnecessários.

Com o desenvolvimento deste trabalho é possível analisar a padronização dos arquivos desenvolvidos pelo Forms 6i e também qualquer outro programa PL/SQL. Estes arquivos devem ser selecionados de um diretório, onde os mesmos devem estar em formato de um

arquivo texto (com a extensão .TXT). Foram desenvolvidos analisadores léxico, sintático e semânticos para extrair do fonte as construções não padronizadas, ou seja, o que não estiver dentro do padrão estabelecido pela empresa. Após a verificação, a ferramenta gera um novo código contendo as correções que devem ser realizadas no código fonte e também lista todos os erros encontrados.

1.1 OBJETIVOS DO TRABALHO

O objetivo deste trabalho foi desenvolver uma ferramenta que permita cadastrar padrões de codificação e que faça a análise destes padrões em códigos fonte escritos em PL/SQL.

Os objetivos específicos do trabalho são:

a) disponibilizar de interfaces que permitam a parametrização dos padrões a serem adotados pela empresa;
b) listar os erros de padrões encontrados e gerar um novo código com as correções necessárias.

1.2 ESTRUTURA DO TRABALHO

O trabalho está organizado em quatro capítulos. No segundo capítulo contempla a fundamentação teórica, onde são apresentados conceitos e características sobre padrões de programação, a entrada e saída de dados utilizando compiladores e as principais características da linguagem PL/SQL. No capítulo 3 são descritos os requisitos, a especificação e a implementação da ferramenta de apoio a análise da codificação do código fonte, bem como a funcionalidade da mesma. No último capítulo são apresentadas a conclusão e sugestões para trabalhos futuros.

2 FUNDAMENTAÇÃO TEÓRICA

No presente capítulo são apresentados alguns aspectos teóricos relacionados ao trabalho. Na seção 2.1 são evidenciadas a importância dos padrões de programação. Em seguida a seção 2.2 trata das informações sobre a entrada e saída de dados, utilizando compiladores. Na seção 2.3 são apresentadas as principais características da linguagem PL/SQL. A seção 2.4 descreve os trabalhos correlatos.

2.1 PADRÕES DE PROGRAMAÇÃO

Segundo Koscianski e Soares (2006, p. 307), conhecer e compreender o estilo em que foi escrita uma linguagem implica compreender uma maneira de pensar a respeito dos problemas. Regras simples de codificação podem poupar bastante trabalho, na medida que evitam erros de compreensão durante a leitura do código.

Staa (2000, p. 106) afirma que programas são freqüentemente lidos por pessoas, portanto devem ser redigidos de modo que facilitem a leitura e compreensão. Recomenda o uso de espaços em branco e linhas em branco como um instrumento para tornar mais legível e compreensível o código fonte. Programas também são textos de referência, portanto devem facilitar a localização de elementos procurados. Da mesma forma como textos são particionados em capítulos e seções, programas devem ser bem organizados separando-se os seus componentes segundo a sua natureza.

Segundo Staa (2000, p. 107), comentários devem acrescentar informação que não conseguiria extrair facilmente do código. Se eles forem óbvios ou não elucidativos de nada adiantam. É importante assumir sempre que o leitor entende o suficiente da linguagem para conseguir ler o código. Comentários devem ser facilmente encontrados e devem ser nitidamente distinguíveis do código, para que desta forma eles estejam destacados, sendo facilmente localizados, não gerando confusão caso tenham uma redação parecida com a de código.

A cada vez que se lê um programa é necessário determinar o significado exato de cada elemento encontrado no código fonte. Staa (2000, p. 90) afirma que quanto mais exatamente os nomes dos elementos sugerirem o seu significado e sua correta forma de uso, ou seja,

quanto mais claros forem os nomes, menos esforço será gasto para se compreender completa e corretamente o significado do programa.

2.1.1 Escolha de nomes

O código fonte de um programa é lido, revisado e alterado por diferentes pessoas, em diferentes épocas e ao realizar diferentes tarefas. Staa (2000, p.106) afirma que poucas vezes o código fonte será processado por compiladores, por isso é muito importante que o nome dos elementos encontrados no código, sugiram o seu significado, pois com isso menos esforço será gasto para se compreender completa e corretamente o significado do programa.

A má escolha dos nomes é uma das causas da dificuldade da compreensão do código fonte de um programa. Por sua vez, a falta de compreensão tende a induzir erros ao programar, alterar e depurar programas.

De acordo com Koscianski e Soares (2006, p. 311) não existem normas universalmente aceitas sobre como padronizar os nomes. Algumas empresas criam diretivas internas que permitem harmonizar o trabalho em equipe.

Inicialmente, pode ser útil definir um dicionário de prefixos e sufixos utilizados para nomear elementos dentro do código. Estes prefixos devem ser identificados de seu uso no código. Um exemplo é mostrado no quadro 1.

PREFIXO	SIGNIFICADO
arq	arquivo
c	caractere
i, j, k, cont	contador
f, flag	variável booleana
fun	função
n, m, q	quantidades
r, reg	registros
s, str	cadeias de caracteres
buf	buffer
x, y, z	coordenadas
p, ptr	ponteiros

Fonte: Koscianski (2006, p. 312).

Quadro 1– Dicionário de prefixos para variáveis

2.1.2 Recuos, espaçamento e alinhamento

É comum o reconhecimento de uma estrutura em textos, contendo títulos e sub-títulos, parágrafos, figuras ou tabelas. Esse hábito é adquirido após alguns anos de leitura e também é aplicado inconscientemente quando realizada a leitura de um código fonte (Koscianski e Soares, p.314).

Muitos programadores consideram que um determinado estilo de disposição do texto é uma questão de gosto pessoal. Mas ao se trabalhar em equipe é importante se estabelecer um padrão comum. Cada desenvolvedor deve compreender a importância de aderir a um formato comum e útil à equipe toda.

Segundo Dolla (2001, p. 25), uma indentação apropriada facilita a identificação de declarações, fluxos de controle, comentários não-executáveis e outros componentes do código-fonte. É sugerido que a indentação, seja aplicada nos seguintes tópicos:

a) blocos de código, mantidos entre cláusulas `begin` e `end`;
b) comentários;
c) em níveis mais internos dentro de um mesmo bloco;
d) comandos estruturados como `if...then...end, loop...end loop`;
e) tratamento de exceção.

De acordo com Staa (2000, p. 106), o ideal é a utilização de três a cinco espaços a cada nível de indentação. A cada subrotina, é sugerido manter uma linha branca antes e após a sua especificação, com o propósito de manter clareza. Procurar utilizar somente um comando por linha.

2.1.3 Comentários e documentação

Segundo Dolla (2001, p. 26), comentários incompletos e não atualizados constantemente de acordo com o estado atual do código, podem trazer problemas para a leitura e também para a segurança do programa, pois uma tomada de decisão pode ser baseada nestes comentários ou documentação. Estes problemas podem ser minimizados padronizando-se o processo de codificação de um projeto ou de toda a organização.

Dolla (2001, p. 26), sugere comentar no início do programa:

a) o objetivo principal do programa;

b) requerimentos de funções e performance, interfaces externas que o programa ajude a implementar;
c) outros programas ou subprogramas chamados e suas dependências;
d) data de criação do programa;
e) data da última revisão, número da revisão, problema e um breve comentário;
f) comportamentos de determinadas ações;
g) entradas e saídas, incluindo arquivos, se existirem.

2.1.4 Evitar o uso de literais

De acordo com Dolla (2001, p.38), o uso explícito de literais (ex.: números ou nomes fixos) dentro de um código fonte prejudica a legibilidade e o processo de manutenção, principalmente se estas declarações são utilizadas numa passagem de parâmetros ou como fator de conversão.

É mais simples atualizar apenas um valor em uma tabela ou arquivo, de acordo a implementação, do que garantir que todos os programas que usam esta informação de forma fixa foram alterados adequadamente.

2.2 ENTRADA E SAÍDA DE DADOS UTILIZANDO COMPILADORES

De forma simplificada, um compilador é um programa que lê um programa escrito numa linguagem (linguagem fonte) e o traduz num programa equivalente numa outra linguagem (linguagem alvo). De acordo com Aho, Sethi e Ullman (1995, p. 13), como importante parte neste processo de tradução, o compilador mostra ao usuário a presença de erros no programa fonte.

Os analisadores léxico, sintático e semântico são módulos que compõem a estrutura básica de um compilador, que podem ser utilizados no desenvolvimento da análise do código fonte da linguagem PL/SQL.

2.2.1 Análise léxica

De acordo com Aho, Sethi e Ulmann (1995, p. 38), o papel do analisador léxico é de ler os caracteres de entrada e produzir um seqüência de *tokens* que são utilizados pelo *parser* para a análise sintática, além de realizar outras tarefas, como remover espaços em branco e comentários. Os *tokens* podem ser classificados como palavras reservadas, identificadores, símbolos especiais, constantes de tipos básicos (inteiro, real, literal), entre outras categorias (HIEBERT, 2003, p. 14) e são definidos através de expressões regulares conforme mostra o quadro 2:

```
digito: [0-9]
letra: [a-zA-Z]
identificador: {letra} ({letra}|{digito})*
constante_real: ("+"|"-")? {digito}+"."{digito}+
constante_inteira: ("+"|"-")? {digito}+
```

Quadro 2– Exemplo de definição de *tokens*

No quadro 2 tem-se as seguintes expressões regulares: `digito`, que representa os dígitos entre 0 e 9; `letra`, representa o alfabeto, ou seja, letras maiúsculas e minúsculas; `identificador`, uma seqüência que inicia com uma `letra` seguida ou não (`*`) de um conjunto de `letras` ou `dígitos`; `constante_real`, pode iniciar ou não (?) com um sinal unário seguido de um ou mais dígitos, seguido de um ponto (“.”) e em seqüência um ou mais dígitos; `constante_inteira` que inicia ou não (`?`) com o sinal unário seguido de um ou mais (`+`) `dígitos`.

2.2.2 Análise sintática

A respeito do analisador sintático, Aho, Sethi e Ulmann (1995, p. 72) afirmam que o mesmo obtém uma cadeia de *tokens* proveniente do analisador léxico e verifica se a mesma pode ser gerada pela gramática da linguagem-fonte.

Segundo Hiebert (2003, p. 15), a análise sintática é a segunda fase do tradutor e tem como função verificar se as construções utilizadas no programa fonte estão gramaticamente corretas. O analisador analisa o texto do programa fonte como uma sentença que deve satisfazer as regras gramaticais de uma gramática livre de contexto (GLC).

No quadro 3, encontra-se uma parte da gramática utilizada neste trabalho.

```
...
<INSERT> ::= insert into identificador
             "(" <IDENTIFICADOR1>")"
             values
             "(" <IDENTIFICADOR2> ")" ";";
<IDENTIFICADOR1>::= identificador<IDENTIFICADOR1> |
                    ","<IDENTIFICADOR1>|î;
<IDENTIFICADOR2>::= ","<IDENTIFICADOR2>|
                    constante_inteira<IDENTIFICADOR2>|
                    constante_real<IDENTIFICADOR2>|
                    constante_string<IDENTIFICADOR2>|
                    identificador|î;
...
```

Quadro 3 – Trecho da gramática PL/SQL

No quadro 3 pode-se observar que na gramática da linguagem PL/SQL, quando for um comando de `insert` devem ter as palavras reservadas `insert into`, seguida de um identificador que representa o nome da tabela, seguidos de "(" e dos nomes dos campos da tabela (outro identificador), sendo que podem ter um ou mais identificadores e ao final dos campos o símbolo ")", seguido da palavra reservada `values`, do símbolo "(", seguidos dos valores dos campos (que pode ser um identificador ou uma constante inteira, ou uma constante real, ou ainda uma constante *string*), e ao final devem ter os símbolos ")" e ";".

2.2.3 Análise semântica

O analisador semântico utiliza uma árvore sintática produzida pelo analisador sintático para determinar o significado do código fonte. São funções do analisador semântico: criar e manter a tabela de símbolos, contendo os identificadores encontrados no código fonte; identificar operadores e operandos das expressões; e também fazer verificações de correspondência entre parâmetros reais e formais. (MARCOS, 2007, p. 21).

A semântica é determinada através de ações semânticas inseridas na gramática, conforme pode ser visualizado no quadro 4.

```
...
<INSERT> ::= insert #1 into#1 identificador#5
            "(" #17<IDENTIFICADOR1>")"#17
              values#1
              "("#17 <IDENTIFICADOR2> ")"#17 ";"#14;
<IDENTIFICADOR1>::= identificador#4<IDENTIFICADOR1> |
                      ","#17<IDENTIFICADOR1>|î;
<IDENTIFICADOR2>::= ","#17<IDENTIFICADOR2>|
                      constante_inteira#18<IDENTIFICADOR2>|
                      constante_real#19<IDENTIFICADOR2>|
                      constante_string#20<IDENTIFICADOR2>|
                      identificador#2|î;
...
```

Quadro 4 – Exemplos de ações semânticas

2.2.4 Gramáticas livres de contexto

Segundo Hiebert (2003, p.16), as GLC, popularizadas pela notação BNF, formam a especificação para a estrutura sintática de uma linguagem de programação. Louden (2004, p. 97) afirma que essa especificação é muito similar à especificação da estrutura léxica de uma linguagem por expressões regulares, exceto que uma gramática livre de contexto utiliza regras recursivas, pois permitem interpretar a maioria das linguagens de programação utilizadas atualmente.

Uma gramática livre de contexto possui quatro componentes básicos:

a) um conjunto de terminais, que são os símbolos básicos da linguagens, ou *tokens*;
b) um conjunto de não-terminais, que são símbolos utilizados na descrição da linguagem;
c) um conjunto de produções gramaticais, que são regras que especificam a forma pela qual os terminais e não-terminais podem ser combinados, sendo que cada produção contém exatamente um símbolo não-terminal do lado esquerdo;
d) uma designação a um dos não-terminais como símbolo de partida.

Livre de contexto, de acordo com Hiebert (2003, p.16), significa que em qualquer contexto um símbolo não-terminal pode ser substituído por sua produção gramatical, isto é, não é necessário fazer qualquer análise dos símbolos que sucedem ou antecedem o não-terminal.

Segundo Hiebert (2003, p.16), a notação BNF e representada da seguinte forma:

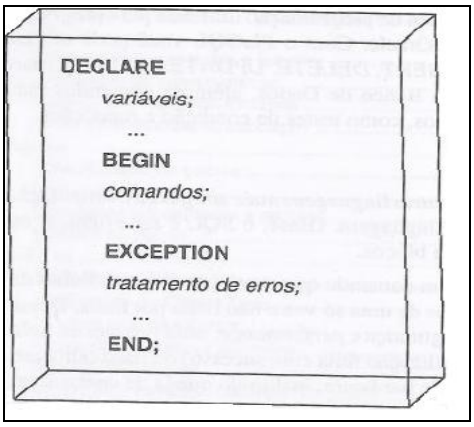
a) <x> representa um símbolo não-terminal, onde x é o nome do não-terminal;
b) a seqüência <x> ::= B representa uma regra de produção, associando o não-terminal <x> à sentença B, sendo que <x>::= B significa "<x> é definido por B";
c) o caracter | separa as diversas regras de produção que estão à direita do símbolo ::=, desde que o símbolo não-terminal à esquerda seja o mesmo;
d) o significado de <x>::=B1|B2|...|Bn é <x> é definido por B1 OU <x> é definido por B2 OU... <x> é definido por Bn;
e) o símbolo x ou X representa um símbolo terminal, dado pela cadeia x ou X de caracteres quaisquer e deve ser escrito tal como aparece nas sentenças da linguagem;
f) o símbolo î significa vazio.

2.3 LINGUAGEM PL/SQL

Segundo Oliveira (2002, p. 11) PL/SQL é uma linguagem da Oracle, que surgiu para suprir as limitações da linguagem *Structured Query Language* (SQL), com a qual é altamente integrada. Possui várias vantagens, onde pode-se destacar a performance, sendo que sua utilização reduz o tráfego na rede pelo envio de um bloco contendo vários comandos SQL agrupados para o banco de dados Oracle. Souza (2004, p. 26) relata sua portabilidade, onde aplicações desenvolvidas utilizando a linguagem PL/SQL tornam-se portáveis para qualquer sistema operacional e plataforma suportada pelo Oracle, não havendo a necessidade de customizações.

As seções que compõem um bloco (figura 1) são:

a) declaração de variáveis: cláusula DECLARE;
b) execução de comandos e lógica: cláusula BEGIN;
c) execução ou tratamento de erros: cláusula EXCEPTION.



Fonte: Oliveira (2002, p. 12).

Figura 1- Estrutura de um bloco PL/SQL

2.3.1 Tipos de declarações

Para declaração de constantes, a palavra `constant` deve aparecer antes do tipo de dado e um valor deve ser atribuído neste momento para a constante, utilizando o símbolo de atribuição (:=) ou a palavra reservada `default` seguido de um valor logo após o tipo e tamanho da constante.

As variáveis também podem ser inicializadas, o que é feito da mesma forma que se dá um valor a uma constante. As variáveis não inicializadas explicitamente receberão o valor `null`, sendo desnecessário inicializar uma variável como valor nulo.

Além disso, podem ser aplicadas restrições como `not null`, indicando que determinada variável deve ser inicializada. A restrição `not null` deverá ser colocada entre a definição de tipo e tamanho e a inicialização. No quadro 5, é apresentado a sintaxe de declaração de variáveis, onde o que tiver dentro de colchetes é opcional. E no quadro 6, são apresentados exemplos de tipos de declarações.

```
identificador [constante] tipo de dados [NOT NULL] [:= | Default expressão];
```

Quadro 5 – Sintaxe de declaração de variáveis

VARIÁVEL	TIPO
ww_nr_lidos	constant number(4) := 15;
ww_tentativa	number(4) := 0;
ww_nr_lidos_2	constant number(4) default 15;
ww_nr_tentativa2	number(4) default 0;
ww_cd_cliente	number(4) NOT NULL := 1;

Quadro 6 – Tipos de declarações

2.3.2 Tipos de dados

Segundo Dolla (2001, p. 06) todo literal ou coluna manipulada pelo banco de dados tem um tipo de dado, sendo que o tipo de dado do valor associa um conjunto de propriedades, que servem para que o banco de dados faça o tratamento correto para cada valor.

No processo de criação de tabelas e procedimentos, são utilizados tipos de dados. Estes tipos definem o domínio de valores que cada coluna ou argumento pode possuir. Os tipos de dados da linguagem PL/SQL são identificados no quadro 7.

TIPO DE DADO	DESCRIÇÃO
CHAR	Alfanumérico de tamanho fixo, máximo de 255 caracteres.
CARACTER	É um subtipo idênticos ao tipo Char. Este subtipo é utilizado em geral, para manter compatibilidade com outras versões do SQL.
VARCHAR2	Alfanumérico com tamanho máximo de 2.000 caracteres. A principal diferença deste tipo para o tipo Char é que com o tipo Varchar2 o número de caracteres que não for utilizado não ocupa espaço em branco de dados.
VARCHAR E STRING	São subtipos idênticos ao tipo Varchar2, mas são utilizados apenas para manter compatibilidade com versões diferentes ou anteriores do SQL.
LONG	Alfanumérico com tamanho máximo de 2G. (O tamanho não pode ser informado). Só pode existir um por tabela e não pode ser utilizado na cláusula WHERE de consultas.
NUMBER	Numérico com tamanho máximo de 38 caracteres. Na especificação de tipos de dados numéricos com casas decimais, primeiro é informado o número total de dígitos, que inclui as casas decimais, cujo número de dígitos estará separado do primeiro por uma vírgula.
DATE	Data e hora
RAW	Armazena valores hexadecimais com tamanho variável (máximo de 2k). Normalmente, este tipo de campo é utilizado para armazenamento de imagens.
LONG RAW	Armazena valores hexadecimais com tamanho variável (máximo de 2G). Normalmente, este tipo de campo é utilizado para armazenamento de imagens.
ROWID	String hexadecimal que representa o endereço único de uma linha em uma tabela.
BOOLEAN	Permite armazenar os valores TRUE, FALSE ou NULL.
CLOB	Objeto caracter grande contendo caracteres de byte simples. Tamanho variável de conjunto de caracteres não é permitido. Tamanho máximo de 4 gigabytes.
NCLOB	Objeto caracter grande contendo caracteres de múltiplos bytes. Tamanho máximo de 4 gigabytes. Usado para armazenar dados de conjuntos de caracteres nacionais.
BLOB	Objeto binário grande. Tamanho máximo é de 4 gigabytes.

Fonte: adaptado de Fanderuff (2000, p. 111).

Quadro 7 – Tipos de dados

2.3.3 Atribuição de valores às variáveis

De acordo com Fanderuff (2000, p.114), para atribuir valor a uma variável basta utilizar o operador de atribuição ":=" (dois-pontos e igual). Ou a partir de um comando *select* atribuir valores de campos ou expressões baseadas no conteúdo de um ou mais campos a variáveis, com o uso da cláusula *into*. No quadro 8, pode ser visualizada a sintaxe de atribuições de valores às variáveis. Um exemplo de atribuição de valores a variáveis, é identificado no quadro 9.

```
identificador := expressão;


SELECT nome_coluna
INTO variável
FROM nome_tabela;
```

Quadro 8 – Sintaxe de atribuições às variáveis

```
SELECT qt_produto
      ,vl_produto
  INTO ww_qt_produto
      ,ww_vl_produto
  FROM item_nota_fiscal;


ww_total := ww_qt_produto * ww_vl_produto;
```

Fonte: adaptado de Fanderuff (2000, p. 114).

Quadro 9 – Comando *SELECT – INTO*

2.3.4 Herança de tipo e tamanho

Fanderuff (2000, p.114) afirma que as variáveis e constantes criadas em blocos PL/SQL, podem herdar o tipo de dado de colunas, de outras variáveis e até da linha inteira de uma tabela. Este tipo de definição diminui as manutenções oriundas de alterações nas definições das colunas de tabelas. Um bloco pode ter os seguintes tipos de herança:

a) para herdar o tipo de uma coluna de uma tabela;
b) para herdar o tipo do registro (linha inteira) de uma tabela;
c) para herdar o tipo de uma variável previamente declarada.

No quadro 10, podem ser visualizados exemplos dos tipos de herança citados.

```
Nome_da_Variável   Nome_da_Tabela.Nome_da_Coluna%TYPE;
Nome_da_Variável   Nome_da_Tabela%ROWTYPE;
Nome_da_Variável   Nome_da_Variável%TYPE;
```

Quadro 10 – Tipos de herança

2.3.5 Cursores

Cursores consistem em áreas compostas de linhas e colunas em memória que servem para armazenar o resultado de uma seleção, que retorna 0 ou mais linhas. No PL/SQL os cursores podem se classificar em dois tipos: explícitos e implícitos.

2.3.5.1 Cursores explícitos

Estes cursores são utilizados para a execução de consultas que possam retornar nenhuma ou mais de uma linha. Neste caso, o cursor deve ser explicitamente declarado na área de declarações. Para nomear o resultado do cursor é necessário que o cursor e suas respectivas colunas possuam nomes, com isso algumas regras devem ser seguidas, como o nome do cursor não pode ser igual ao nome da tabela e para dar um nome a uma coluna basta colocar o nome do alias logo após a definição da coluna, segundo Fanderuff (2000, p. 131) . No quadro 11 é mostrado a sintaxe do cursor explícito. Um exemplo de cursor explícito pode ser visualizado no quadro 12.

```
CURSOR nome_cursor IS
descrição_select
```

Quadro 11 – Sintaxe de cursor explícito

```
CURSOR nome_do_cursor (relação_de_parâmetros) IS
  SELECT ds_nome  nome
        ,nr_idade idade
   FROM funcionário
  WHERE código = 1356;
```

Fonte: adaptado de Fanderuff (2000, p. 131).

Quadro 12 – Cursor Explícito

De acordo com Fanderuff (2000, p.131), após a sua declaração, o cursor pode ser manipulado com o uso de alguns comandos:

a) `OPEN`: tem a função de abrir o cursor. Ele executa a consulta (comando `select`)

relacionada ao cursor, preenchendo parte de uma área de memória e neste momento, o ponteiro do cursor estará apontando para o primeiro dos registros resultantes da consulta. A partir de então, para acessar dados deste cursor, o banco não será novamente acessado, pois todos os registros estarão disponíveis em memória após a execução do comando `open;`

b) `FETCH`: disponibiliza a linha corrente do cursor. As linhas no cursor somente podem ser processadas quando seu conteúdo for transferido para variáveis, que possam ser manipuladas no PL/SQL. Esta cópia é realizada pelo comando *`fetch`*, que, além disso, posiciona o ponteiro no próximo registro do cursor. A lista de variáveis que aparece na sintaxe do comando *`fetch`* deve conter o mesmo número de variáveis, na mesma seqüência e com tipos correspondentes às colunas selecionas no comando *`select`* da declaração do;

c) *`CLOSE`*: fecha o cursor, após a sua manipulação para que a área de memória ocupada pelo resultado da consulta gerado por ele.

No quadro 13, é apresentado a sintaxe dos comandos `open`, `fetch` e `close`. No quadro 14, são mostrados exemplos de declaração e manipulação de cursores, utilizando os comandos `OPEN`, `FETCH` e `CLOSE`.

```
OPEN nome_cursor;
FETCH nome_cursor INTO [variável 1, variável 2, ...] | nome_registro];
CLOSE nome_cursor;
```

Quadro 13 – Sintaxe dos comandos `open`, `fetch` e `close`

```
DECLARE
   ww_cd_produto produto.cd_produto%TYPE;
   ww_nm_produto produto.nm_produto%TYPE;

   CURSOR preferencia (p_cliente number) IS
     SELECT distinct i.cd_produto
           ,RPAD(p.nm_produto,30) Nome
       FROM item_nota_fiscal i
           ,nota_fiscal n
           ,produto p
      WHERE n.cd_cliente = p_ciente
        AND n.nr_nota    = i.nr_nota
        AND i.cd_produto = p.cd_produto;
BEGIN
  OPEN preferencia;
    FETCH preferência
       INTO ww_cd_produto
           ,ww_nm_produto;
  CLOSE preferencia;
END;
```

Quadro 14 – Declaração e manipulação de cursor

2.3.5.2 Abrindo cursores com o comando `for`

Fanderuff (2000, p.135) afirma que o comando `FOR...LOOP`, quando relativo a um cursor, executa, automaticamente, as seguintes ações:

a) cria a variável do tipo registro, que recebe os dados;

b) abre o cursor;

c) realiza a cópia de linhas, uma a uma (`FETCH`) ;

d) controla o final do cursor;

e) fecha o mesmo.

A cada interação do `LOOP`, o ponteiro avança pelos registros selecionados, iniciando sempre pelo primeiro, no qual estará posicionado na primeira interação do comando `FOR`.

Caso seja necessário sair do laço do comando `FOR` durante sua execução, o cursor deverá ser fechado explicitamente com o comando `CLOSE`. No quadro 15, é mostrado a sintaxe de um cursor com a utilização dos comandos `FOR...LOOP`, e no quadro 16 é apresentado um exemplo da abertura de um cursor utilizando o comando `FOR`.

```
FOR nome_registro IN nome_cursor LOOP
   Lista_comandos;
   ...
END LOOP;
```

Quadro 15 – Sintaxe do cursor utilizando os comandos `for` e `loop`

```
DECLARE
   CURSOR c_preferencia (p_cliente number) IS
     SELECT distinct i.cd_produto
           ,RPAD(p.nm_produto,30) Nome
       FROM item_nota_fiscal i
           ,nota_fiscal n
           ,produto p
      WHERE n.cd_cliente = p_ciente
        AND n.nr_nota    = i.nr_nota
        AND i.cd_produto = p.cd_produto;
BEGIN
  FOR r_preferencia IN c_preferencia(1) LOOP
    DBMS_OUTPUT.PUT_LINE('Produto: .'|| r_preferencia.nome);
  END LOOP;
END;
```

Quadro 16 – Exemplo de abertura de cursor utilizando o comando `for`

2.3.5.3 Cursores implícitos

Segundo Fanderuff (2000, p. 139), para se executar comandos que manipulam ou verificam informações no banco de dados, como `select` (que retorna uma linha), `insert`, `update` e `delete`, em blocos PL/SQL, o Oracle abre implicitamente um cursor para processar um conjunto de dados resultante. Neste caso, um cursor chamado SQL é criado. Os comandos `open`, `fetch` e *close* não podem ser aplicados a esse cursor. No entanto, os atributos de cursores (`%FOUND`, `%NOTFOUND`, `%ROWCOUNT` e `%ISOPEN`) podem ser verificados.

Para os cursores implícitos (quadro 17), os quatro atributos citados, podem ser verificados com o significado e retorno apresentados, os quais são:

a) `SQL%FOUND`: retorna o valor TRUE caso o último comando SQL tenha afetado algum registro ou se o comando `SELECT` retornou algum registro;
b) `SQL%NOTFOUND`: retorna TRUE se o último comando SQL não tenha afetado nenhum registro;

c) SQL%ROWCOUNT: retorna o número de registros afetados pelo último comando SQL ou a última quantidade de registros retornada pelo último comando SELECT (que deverá ser sempre 1) ;

d) SQL%ISOPEN: vai sempre retornar FALSE, pois o Oracle sempre fecha o cursor após a execução do comando.

```
DECLARE
  ww_vendas NUMBER(5);
  CURSOR c_produtos IS
    SELECT cd_produto
          ,vl_custo_medio
      FROM produto;
BEGIN
  FOR r_produtos IN c_produtos LOOP
    SELECT COUNT(*)
      INTO ww_vendas
      FROM item_nota_fiscal
     WHERE cd_produto = r_produtos.cd_produto;
     IF ww_vendas < 4 THEN
       UPDATE produto
          SET vl_custo_medio = vl_custo_medio * 0.95
        WHERE current of c_produtos;
        IF SQL%NOTFOUND THEN
          DBMS_OUTPUT.PUT_LINE('Erro na atualização.');
        END IF;
     END IF;
  END LOOP;
END;
```

Fonte: adaptado de Fanderuff (2000, p. 140).

Quadro 17 – Exemplo de um cursor implícito

2.3.6 Controle condicional

De acordo com Fanderuff (2000, p.121), o comando "IF" executa determinadas ações de acordo com uma ou mais condições, conforme mostra o quadro 18.

```
IF condição THEN
   relação_de_comandos1
[ELSIF condição THEN
   relação_de_comandos2]
[ELSE
   relação_de_comandos3]
END IF;
```

Fonte: adaptado de Fanderuff (2000, p. 121).

Quadro 18 – Comando IF

A `relação_de_comandos1` será executada se a condição após o comando `IF` seja verdadeira. A `relação_de_comandos2` será executada caso a condição que segue o comando `ELSIF` seja verdadeira, sendo que esta condição apenas será verificada se as condições do comando `IF` e de outros `ELSIFs` anteriores forem falsas. Já a `relação_de_comandos3` será processada apenas se nenhuma das condições anteriores for verdadeira.

Uma condição `IF`, `ELSIF` ou `ELSE` pode ter outras estruturas condicionais (comandos `IF`) aninhadas.

2.3.7 Controle iterativo

Segundo Fanderuff (2000, p.123), a estrutura "`LOOP`" permite executar uma relação de comandos até que uma condição de saída (`EXIT`) seja encontrada, caso esta condição não exista, os comandos serão executados infinitamente, conforme mostra o quadro 19.

```
LOOP
  Relação_de_comandos
  IF condição_de_saida THEN
     EXIT;
END LOOP;
```

Fonte: adaptado de Fanderuff (2000, p. 126).

Quadro 19 – Comando LOOP

2.3.8 Tratamento de exceções

Exceções são todos os erros e imprevistos que podem ocorrer durante a execução de um bloco PL/SQL, afirma Fanderuff (2000, p.143). Quando um erro ou um imprevisto ocorre, o gerenciador do banco de dados Oracle abandona a área de comandos, abortando a execução

e procura por uma área de exceções (*EXCEPTION*) o tratamento para a falha ocorrida. No quadro 20 pode ser visualizada a estrutura de um exception num bloco PL/SQL.

```
EXCEPTION
  WHEN nome_da_exceção THEN
     relação_de_comandos
  WHEN nome_da_exceção THEN
     relação_de_comandos
```

Fonte: adaptado de Fanderuff (2000, p. 143).

Quadro 20 – Tratamento de exceção - EXCEPTION

2.3.8.1 Exceções definidas pelo programador

Fanderuff (2000, p.145), afirma que exceções definidas pelo programador são aquelas que precisam ser declaradas e chamadas explicitamente pelo comando RAISE, pois o bloco PL/SQL não consegue identificar o momento em que o desvio para a área de exceção deve ser efetuado.

Estas exceções somente podem ser declaradas na área de declarações de um bloco PL/SQL (quadro 21), subprograma ou package.

```
DECLARE
  Nome_da_exceção EXCEPTION;
BEGIN
  relação_de_comandos
  IF ... THEN
    RAISE nome_da_exceção;
  END IF;
  relação_de_comandos
EXCEPTION
  WHEN nome_da_exceção THEN
    relação_de_comandos
END;
```

Fonte: adaptado de Fanderuff (2000, p. 146).

Quadro 21 – Tratamento de exceção definida

2.3.9 Subprogramas

Segundo Fanderuff (2000, p.151), subprogramas são blocos PL/SQL armazenados no banco de dados de forma compilada e podem ser invocadas por ferramentas Oracle ou em

outros procedimentos/funções armazenados no banco de dados.

O nome de um subprograma pode ter no máximo 30 caracteres. No momento de sua criação pode ser incluído o parâmetro `OR REPLACE`, que irá especificar que, caso o subprograma já exista no banco, o mesmo seja substituído pela nova versão. Esta opção possui outras vantagens, como:

a) salvar os privilégios existentes;
b) criar, mesmo que haja erro de sintaxe;
c) marcar objetos dependentes para compilação.

2.3.9.1 Parâmetros

De acordo com Fanderuff (2000, p.151), os subprogramas podem ou não receber parâmetros. Os mesmos são passados para os subprogramas de três maneiras, entrada, saída ou entrada/saída, conforme são apresentados abaixo:

a) `IN` (padrão): passa um valor do ambiente chamador para o subprograma e este valor não pode ser alterado dentro do subprograma;
b) `OUT`: passa um valor do subprograma para o ambiente chamador;
c) `IN OUT`: é passado um valor do ambiente chamador para o subprograma e este valor pode ser alterado dentro do mesmo e retornado com o valor atualizado para o ambiente chamador.

Subprogramas não possuem o comando `DECLARE`, sendo que sua área de declarações se localiza entre as palavras reservadas `IS` e `BEGIN`.

2.3.9.2 Procedures

Procedures são subprogramas que têm por objetivo executar uma ação específica, de acordo com Fanderuff (2000, p.151). Elas não retornam valores, não sendo, portanto, utilizadas em atribuições a variáveis ou como argumento em um comando `SELECT`. O exemplo de uma estrutura de uma *procedure* pode ser visualiza no quadro 22.

```
CREATE OR REPLACE PROCEDURE nome_procedure (argumento1 modo tipo_de_dados
                                           ,argumento2 modo tipo_de_dados
                                           ,argumenton modo tipo_de_dados)
IS ou AS
  Variáveis locais, constantes,...
BEGIN
  Bloco PL/SQL
END nome_procedure;
```

Fonte: adaptado de Fanderuff (2000, p. 152).

Quadro 22 – Estrutura de uma *Procedure*

De acordo com a estrutura do quadro 22, o argumento é o nome da variável que será enviada ou retornada do ambiente chamador para o procedimento e pode ser passada em um dos três modos: `IN`, `OUT` ou `IN OUT`.

2.3.9.3 Function

Segundo Fanderuff (2000, p.153), f*unction* são subprogramas com o objetivo de retornar algum resultado ou valor. As funções de base podem ser utilizadas nas aplicações como qualquer função predefinida, ou seja, em atribuições a variáveis ou como argumento em comandos `SELECT`, como pode ser visualizado no quadro 23.

```
CREATE OR REPLACE FUNCTION nome_função (argumento1 IN tipo_de_dados
                                       ,argumento2 IN tipo_de_dados
                                       ,argumenton IN tipo_de_dados)
RETURN tipo_de_dados IS ou AS
BEGIN
  Bloco PL/SQL
END nome_função;
```

Fonte: adaptado de Fanderuff (2000, p. 153).

Quadro 23 – Estrutura de uma *Function*

2.3.10 Package

Package, de acordo com Fanderuff (2000, p.161), são objetos do banco de dados, equivalente a bibliotecas, que guardam:

a) *procedures*;

b) *functions*;

c) definições de cursores;

d) variáveis e constantes;

e) definições de exceções.

Uma *package* é composta por duas partes: especificação e corpo.

2.3.10.1 Especificação

De acordo com Fanderuff (2000, p.161), nesta área são realizadas as declarações públicas, ou seja, as variáveis, constantes, cursores, exceções e subprogramas que estarão disponíveis para o uso externo ao pacote. No quadro 24 pode ser visualizada a estrutura da especificação de uma package.

```
CREATE OR REPLACE PACKAGE nome_da_package IS
  Para procedures e functions, só os cabeçalhos (interface)
  PROCEDURE nome_da_procedure (lista_de_parâmetros);
  FUNCTION nome_da_function (lista_de_parâmetros);
  Declaração de variáveis, constantes, exceções e cursores públicos
END nome_da_package;
```

Quadro 24 – Estrutura da especificação de uma *package*

2.3.10.2 Corpo

Fanderuff (2000, p.161), afirma que nesta área são feitas as declarações privadas, que estarão disponíveis apenas dentro da própria *package*. No quadro 25, é mostrada a estrutura do corpo de uma *package*.

```
CREATE OR REPLACE PACKAGE BODY nome_da_package IS
  Declaração de variáveis, constants, exceções e cursores privados

  PROCEDURE nome_da_procedure (lista_de_parâmetros) IS
  BEGIN
  END;
  FUNCTION nome_da_procedure (lista_de_parâmetros) IS
  RETURN tipo
  BEGIN
    RETURN
  END;
END nome_da_package;
```

Quadro 25 – Estrutura de um corpo de uma *package*

2.3.11 Comentários

De acordo com Dolla (2001, p. 5) os comentários aumentam a legibilidade das aplicações, podendo ser utilizado sempre que possível e não afetam em nenhum momento a execução de procedimentos SQL.

Os comentários podem ser expressos de duas formas:

a) linha única: é representado pelo delimitador "--" ;
b) linhas múltiplas: é representado pelos delimitadores "/*" para iniciar um comentário, e "*/" para encerrar o mesmo.

No quadro 26, pode ser visualizado um exemplo de comentário de linhas múltiplas e no quadro 27 de linha única.

```
/*  Data: 17/04/2009
Objetivo: Rotina responsável por avisar os responsáveis se um workflow está
aguardando aprovação a mais de 3 dias.
*/

PROCEDURE prc_avisa_responsavel IS
....
BEGIN
END;
```

Quadro 26 – Comentário de linhas múltiplas

```
IF ww_valor = 10 THEN
  ww_valor := ww_valor + 1;
  --ww_valor := 12;             Está linha não será considerada
END IF;
```

Quadro 27 – Comentário de linha única

2.4 TRABALHOS CORRELATOS

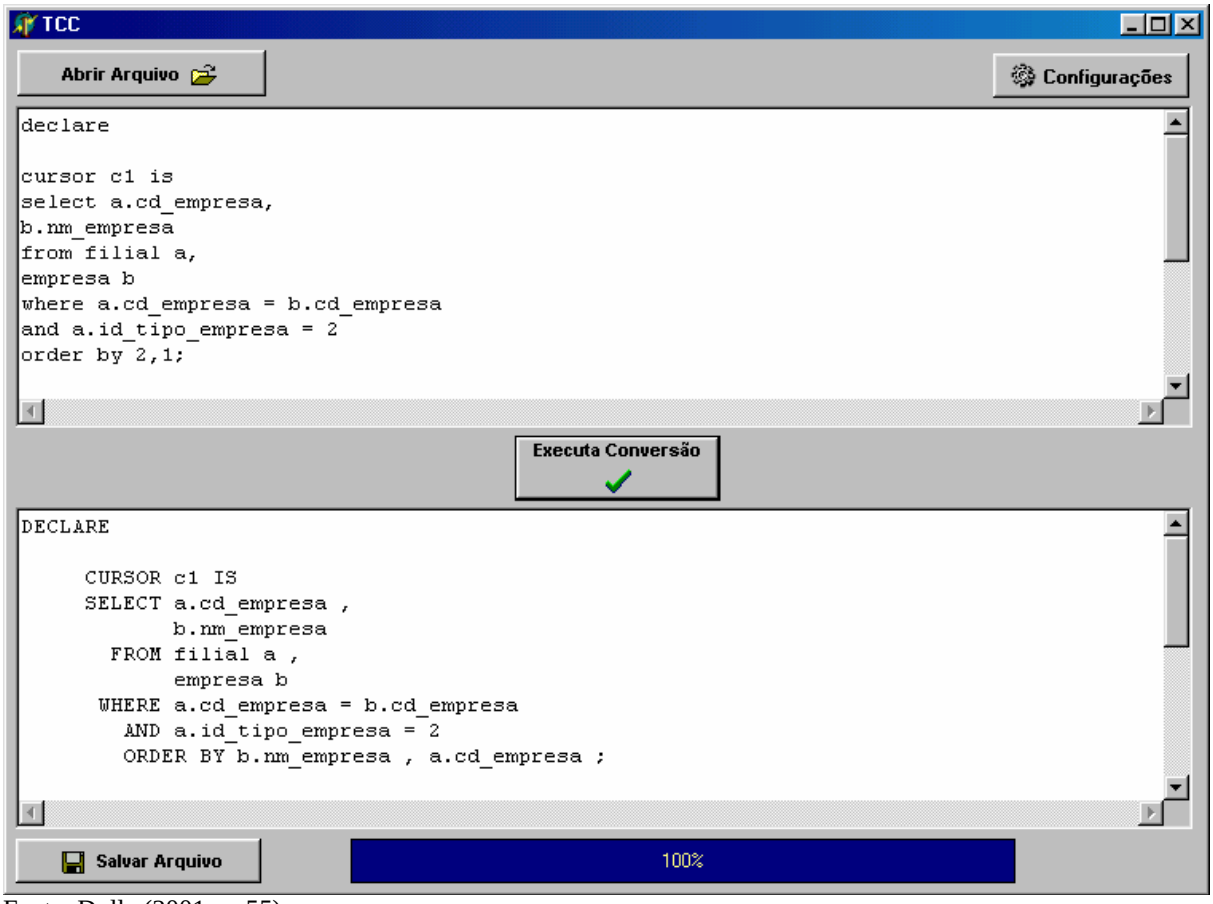
Algumas ferramentas desempenham papel semelhante ao proposto no presente trabalho, onde cada uma possui suas características específicas. Dentre elas foram selecionadas: ferramenta de apoio a reestruturação de código fonte em linguagem PL/SQL baseado em padrões de legibilidade (DOLLA, 2001), ferramenta de apoio a reestruturação de código fonte em linguagem C++ baseado em padrões de legibilidade (DALMOLIN, 2000) e gerador de documentação e apoio a padronização de software implementados na linguagem Progress 4GL (MAAS, 2004).

2.4.1 Ferramenta de apoio a reestruturação de código fonte em linguagem PL/SQL baseado em padrões de legibilidade

Segundo Dolla (2001, p. 13), através do uso desta ferramenta pode-se fazer a reestruturação de um código fonte escrito na linguagem PL/SQL utilizando padrões de legibilidade.

A ferramenta recebe como entrada um código fonte e valida se os padrões definidos na ferramenta encontram-se no mesmo, se não encontrar gera um novo código contendo estes padrões, permitindo que o usuário salve esta validação em um diretório específico.

Alguns dos padrões verificados pela ferramenta são: indentação, formatação da cláusula `ORDER BY`, formatação das palavras chave/reservadas, inclusão de tratamento de exceção, geração de documentação para o programa e suas subrotinas. São gerados avisos de má construção do código fonte, de acordo com padrões sugeridos e estabelecidos na construção da ferramenta. Na figura 2 é mostrado um exemplo de formatação de palavras reservadas.

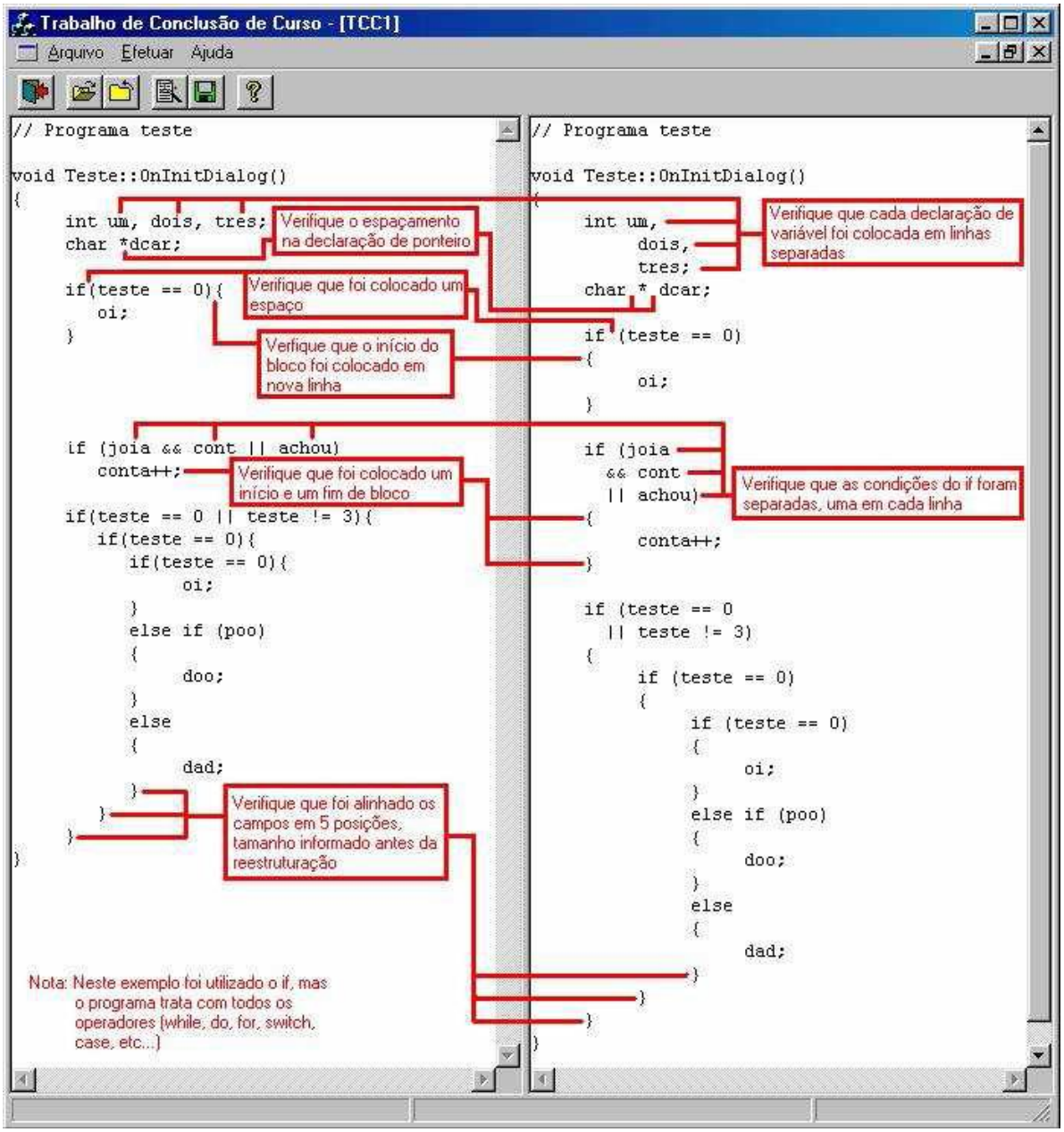


Fonte: Dolla (2001, p. 55).

Figura 2 - Exemplo de formatação de ORDER BY e palavras reservadas

2.4.2 Ferramenta de apoio a reestruturação de código fonte em linguagem C++ baseado em padrões de legibilidade

De acordo com Dalmolin (2000, p. 29), a ferramenta desenvolvida faz uma análise em um código fonte desenvolvido em C++, gerando avisos ao usuário sobre comandos não permitidos e construções que não estão de acordo com algum padrão estabelecido na construção da ferramenta, como: não utilizar comandos `goto`, `continue`, entre outros. É também realizada a cópia do código fonte, porém esta contém os comandos com todos os padrões estabelecidos pela ferramenta. Na figura 3 tem um exemplo do antes e depois da reestruturação do código fonte.



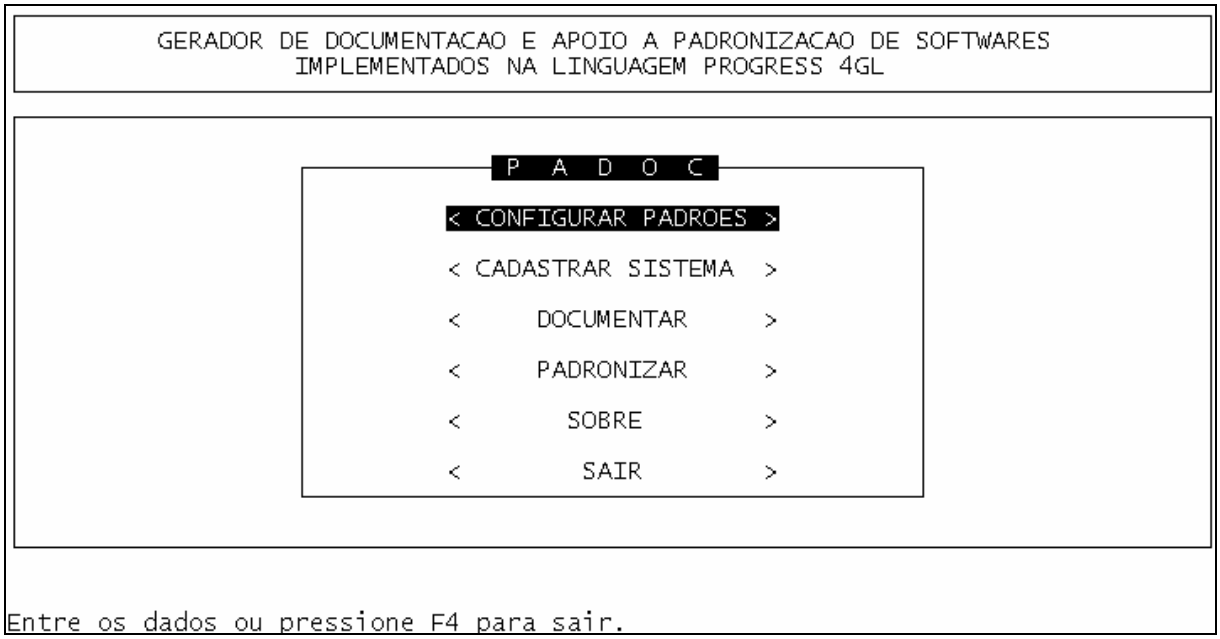
Fonte: Dalmolin (2000, p. 35).

Figura 3 - Tela de software com exemplo de código fonte antes e depois da reestruturação

2.4.3 Gerador de documentação e apoio a padronização de software implementados na linguagem progress 4GL

Maas (2004, p. 10) relata que a ferramenta tem a utilidade de auxiliar o controle de padronização dos códigos fonte, com o objetivo de ajudar o desenvolvedor de sistema, para que o mesmo tenha a possibilidade de identificar facilmente as diferenças entre variáveis, nomes de campos das tabelas e palavras reservadas na linguagem Progress 4GL, através dos tipos de nomenclatura, definidas pelo usuário na própria ferramenta.

Além da padronização, a ferramenta auxilia no processo de documentação dos fontes, através de uma varredura em busca de comandos que façam qualquer tipo de acesso ao banco de dados. Na figura 4 pode-se ver a tela principal da ferramenta desenvolvida.



Fonte: Maas (2004, p. 40).

Figura 4 - Tela inicial da ferramenta

3 DESENVOLVIMENTO DO TRABALHO

A motivação para o desenvolvimento deste trabalho está relacionada com a possibilidade de ter uma ferramenta para identificar padrões e boas práticas de programação, sem haver a necessidade de abrir o código e identificar manualmente. Possibilita uma economia de custo e tempo, além de não correr o risco de deixar passar alguma coisa que não foi identificada pelo avaliador.

De acordo com Staa (2000, p. 90), o código fonte é lido, alterado e revisado por diferentes pessoas e em diferentes épocas. A cada vez que se lê o programa, é necessário identificar o que é cada elemento no código fonte. Quanto mais significativos forem os nomes dos elementos em relação a sua correta forma de uso, menos esforço será gasto para se compreender corretamente o significado do programa.

Então, não só a ferramenta será importante para os avaliadores da qualidade, mas também para que no futuro, quando outras pessoas realizarem manutenção, terão facilidade no entendimento, agilizando o processo de desenvolvimento.

3.1 REQUISITOS PRINCIPAIS

Nos quadros 28 e 29 são apresentados, respectivamente, os requisitos não funcionais e funcionais da ferramenta.

REQUISITOS NÃO FUNCIONAIS	
RNF01	Ser compatível com o sistema operacional Windows.
RNF02	Ser desenvolvida utilizando o ambiente Borland Delphi 7.0.
RNF03	Utilizar o banco de dados Interbase.
RNF04	O sistema deve destacar em vermelho o conteúdo que se apresenta fora dos padrões.

Quadro 28 – Requisitos não funcionais

REQUISITOS FUNCIONAIS	
RF01	O sistema deve permitir cadastrar o glossário de palavras abreviadas para o entendimento do domínio da aplicação.
RF02	O sistema deve permitir cadastrar os prefixos.
RF03	O sistema deve permitir cadastrar os parâmetros estabelecidos pela empresa para a análise dos padrões.
RF04	O sistema deve permitir gerar vários arquivos contendo blocos PL/SQL a partir de um arquivo gerado por uma aplicação Forms 6i.
RF05	O sistema deverá gerar um arquivo contendo o conteúdo do arquivo PL/SQL reestruturado de acordo com os padrões cadastrados.
RF06	O sistema deverá gerar uma listagem contendo os erros encontrados, listando linha a linha e identificando qual foi o comando que apresentou o erro.
RF07	O sistema deve permitir gerar um relatório estatístico com os principais problemas identificados em vários programas.

Quadro 29 – Requisitos funcionais

3.2 ESPECIFICAÇÃO

Foi utilizada a UML como linguagem de especificação dos diagramas de casos de uso e de atividades, com a utilização da ferramenta *Enterprise Architect.* Os diagramas são detalhados na seção 3.2.1 e 3.2.2. O modelo de entidade e relacionamento (MER) foi desenvolvido com a ferramenta *DBDesigner*, no qual o modelo é detalhado na seção 3.2.3.

3.2.1 Diagrama de casos de uso

O diagrama de casos de uso demonstra as interações do usuário com o programa. Cada caso de uso representa uma ação que o usuário pode realizar na aplicação. Na figura 5 encontram-se os casos de uso da aplicação relacionada aos cadastros, na figura 6 estão os casos de uso dos processos e nos quadros 30, 31, 32, 33, 34 e 35 o detalhamento de cada um deles.

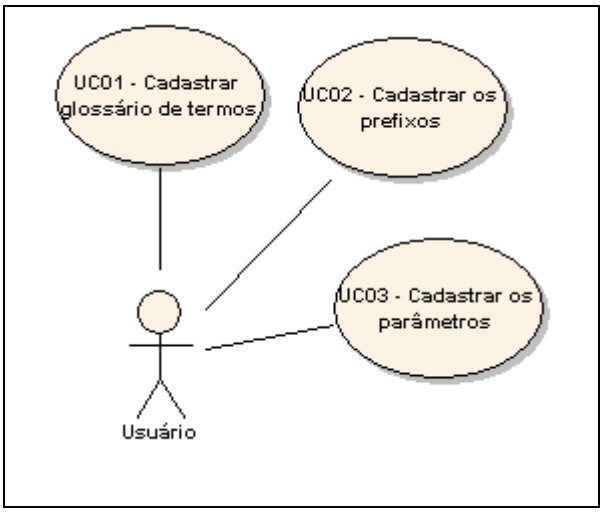


Figura 5 - Diagrama de casos de uso (cadastros)

UC01. Cadastrar Glossário de Termos
Cenário principal: 1. O usuário solicita cadastrar um glossário de termos. 2. O sistema apresenta a tela para cadastro de glossários. 3. O usuário seleciona a opção para criar um novo glossário a partir do botão **Novo**. 4. O usuário informa o código e a descrição do glossário e seleciona a opção para salvar o glossário, a partir do botão **Salvar**. 5. Sistema salva as informações. 6. O usuário opta por outra operação ou encerra o caso de uso.
Cenários alternativos: A1. Alterar: no passo 2, caso usuário opte por alterar um glossário: 2.1 Sistema apresenta tela para busca de glossário. 2.2 Usuário seleciona um glossário. 2.3 Sistema apresenta as informações do glossário. 2.4 Usuário altera os dados necessários. 2.5 Usuário seleciona o botão **Salvar**. 2.6 Sistema salva o glossário. A2. Excluir: no passo 2, caso usuário opte por excluir um glossário. 2.1 Sistema apresenta tela para busca de glossário. 2.2 Usuário seleciona um glossário. 2.3 Sistema apresenta as informações do glossário. 2.4 Usuário seleciona o botão **Excluir**. 2.5 Sistema exclui o glossário.
Cenário de exceção: E1. No passo 4, caso o glossário informado já existir: 4. A ferramenta apresenta a seguinte mensagem: “Glossário já cadastrado!”.
Pós-condição: Um novo glossário foi cadastrado no sistema. **Pós-condição**: Um glossário foi alterado ou excluído do sistema. **Pós-condição**: Os dados de um glossário foram consultados no sistema.

Quadro 30 – Detalhamento do caso de uso `Cadastrar Glossário de Termos`

UC02. Cadastrar Prefixos
Cenário principal: 1. O usuário solicita cadastrar um prefixo. 2. O sistema apresenta a tela para cadastro de prefixos. 3. O usuário seleciona a opção para criar um novo prefixo a partir do botão **Novo**. 4. O usuário informa o código, a descrição e a observação do prefixo e seleciona a opção para salva-lo a partir do botão **Salvar**. 5. Sistema salva as informações. 6. O usuário opta por outra operação ou encerra o caso de uso.
Cenários alternativos: A1. Alterar: no passo 2, caso usuário opte por alterar um prefixo: 2.1 Sistema apresenta tela para busca de prefixos. 2.2 Usuário seleciona um prefixo. 2.3 Sistema apresenta as informações do prefixos. 2.4 Usuário altera os dados necessários. 2.5 Usuário seleciona o botão **Salvar**. 2.6 Sistema salva informações do prefixo. A2. Excluir: no passo 2, caso usuário opte por excluir um prefixo. 2.1 Sistema apresenta tela para busca de prefixos. 2.2 Usuário seleciona um prefixo. 2.3 Sistema apresenta as informações do prefixo. 2.4 Usuário seleciona o botão **Excluir**. 2.5 Sistema excluir o prefixo.
Cenário de exceção: E1. No passo 4, caso o prefixo cadastrado já existir: 4. A ferramenta apresenta a seguinte mensagem: “Prefixo já cadastrado!”.
Pós-condição: Um novo prefixo foi cadastrado no sistema. **Pós-condição**: Um prefixo foi alterado ou excluído do sistema. **Pós-condição**: Os dados de um prefixo foram consultados no sistema.

Quadro 31 – Detalhamento do caso de uso `Cadastrar Prefixos`

UC03. Cadastrar Parâmetros
Cenário principal: 1. O usuário solicita cadastrar um parâmetro. 2. O sistema apresenta a tela para cadastro de parâmetros. 3. O usuário seleciona a opção para criar um novo parâmetro a partir do botão **Novo**. 4. O usuário informa os dados para os parâmetros e seleciona a opção para salva-lo a partir do botão **Salvar**. 5. Sistema salva as informações do parâmetro. 6. O usuário opta por outra operação ou encerra o caso de uso.
Cenários alternativos: A1. Alterar: no passo 2, caso usuário opte por alterar um parâmetro: 2.1 Sistema apresenta tela para busca de parâmetros. 2.2 Sistema apresenta as informações do parâmetro já cadastrado. 2.3 Usuário altera os dados necessários. 2.4 Usuário seleciona o botão **Salvar**. 2.5 Sistema salva as informações do parâmetro. A2. Excluir: no passo 2, caso usuário opte por excluir o parâmetro. 2.1 Sistema apresenta tela para busca de parâmetros. 2.2 Sistema apresenta as informações do parâmetro. 2.3 Usuário seleciona o botão **Excluir.** 2.4 Sistema exclui o parâmetro.
Cenário de exceção: E1. No passo 1, caso o usuário tentar cadastrar mais de um parâmetro: 1. A ferramenta apresenta a seguinte mensagem: "Parâmetro já cadastrado!".
Pós-condição: Um novo parâmetro foi cadastrado no sistema. **Pós-condição**: Um parâmetro foi alterado ou excluído do sistema. **Pós-condição**: Os dados de um parâmetro foram consultados no sistema.

Quadro 32 – Detalhamento do caso de uso `Cadastrar Parâmetros`

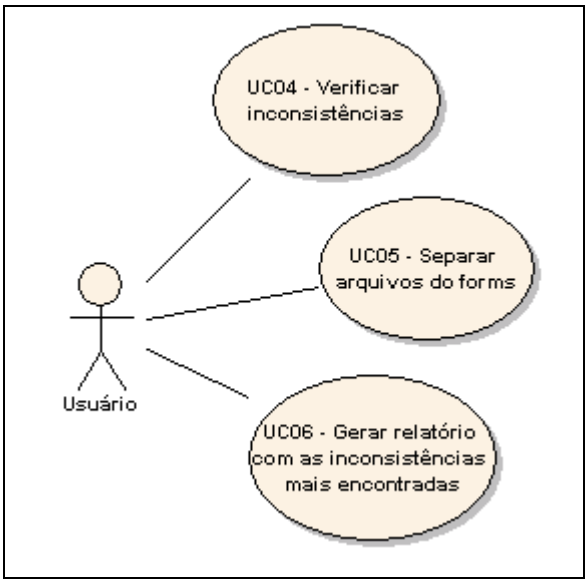


Figura 6 - Diagrama de casos de uso (processos)

UC04. Verificar Inconsistências
Cenário principal: 1. O usuário solicita validar um fonte. 2. O sistema apresenta a tela para validação dos fontes PL/SQL. 3. O usuário seleciona um arquivo fonte, através do botão **Abrir**. 4. O sistema coloca o conteúdo do arquivo no campo da tela. 5. O usuário pressiona o botão **Executar**. 6. O sistema faz as validações necessárias, listando todas as inconsistências encontradas e informando em quais linhas encontram-se as mesmas. 7. O usuário opta por outra operação ou encerra o caso de uso.
Cenários alternativos: A1. Selecionar outro arquivo: a partir do passo 3, o usuário pode optar por selecionar outro arquivo: 3.1 O usuário pressiona o botão **Abrir.** 3.2 O usuário seleciona outro arquivo em qualquer diretório. 3.3 A ferramenta lista o conteúdo deste arquivo num campo da tela. Retorna ao passo 5.
Cenário de exceção: E1. No passo 5, caso o usuário tentar pressionar o botão **Executar** e não tiver nenhum parâmetro cadastrado: 5. A ferramenta apresenta a seguinte mensagem: "Não existe nenhum parâmetro cadastrado!".
Pré-condição: Os parâmetros utilizados para avaliação do código precisam estar cadastrados. **Pós-condição**: Uma nova execução foi realizada no sistema.

Quadro 33– Detalhamento do caso de uso `Verificar Inconsistências`

UC05. Separar arquivos do Forms
Cenário principal: 1. O usuário solicita a tela de separação de arquivos do Forms. 2. O sistema apresenta a tela para o usuário. 3. O usuário seleciona um arquivo do Forms, através do botão **Abrir**. 4. O sistema coloca o conteúdo do arquivo no campo da tela. 5. O usuário pressiona o botão **Executar**. 6. O sistema separa o arquivo em vários outros arquivos e grava no mesmo diretório em que foi selecionado o arquivo do Forms, sendo que os nomes dos arquivos serão o mesmo nome do arquivo inicial, porém com um número seqüencial no final do nome. 7. O usuário opta por outra operação ou encerra o caso de uso.
Cenários alternativos: A1. Selecionar outro arquivo: a partir do passo 3, o usuário pode optar por selecionar outro arquivo: 3.1 O usuário pressiona o botão **Abrir.** 3.2 O usuário seleciona outro arquivo em qualquer diretório. 3.3 A ferramenta lista o conteúdo deste arquivo num campo da tela. Retorna ao passo 5.
Cenário de exceção: E1. No passo 5, caso o usuário tentar pressionar o botão **Executar** e não tiver nenhum arquivo selecionado: 5. A ferramenta apresenta a seguinte mensagem: "Arquivo não encontrado!".
Pós-condição: Uma nova quebra de arquivos foi realizada no sistema.

Quadro 34– Detalhamento do caso de uso `Separar arquivos do Forms`

UC06. Gerar relatório com as inconsistências mais encontradas
Cenário principal: 1. O usuário solicita o relatório de inconsistências. 2. O sistema apresenta a tela de parâmetros para a execução do relatório de inconsistências. 3. O usuário informa os parâmetros necessários para a execução do relatório. 4. O usuário pressiona o botão **Gerar**. 5. O sistema lista um relatório contendo as inconsistências mais encontradas no sistema. 6. O usuário opta por outra operação ou encerra o caso de uso.
Cenários alternativos: A1. Imprimir o relatório: a partir do passo 5, o usuário pode optar por imprimir o relatório: 5.1 O usuário pressiona o botão **Imprimir.** 5.2 O usuário pode selecionar as opções de impressão desejadas e pressionar o botão **OK**. 5.3 O sistema envia o relatório para a impressora. Retorna ao passo 6.
Cenário de exceção: E1. No passo 4, caso o usuário tentar pressionar o botão **Gerar** e o sistema identificar que não existe nenhuma informação para ser visualizada: 4. A ferramenta apresenta a seguinte mensagem: "Não existe nenhuma inconsistência para os parâmetros informados!".
Pré-condição: Precisa existir alguma inconsistência cadastrada para que o relatório possa ser gerado. **Pós-condição**: Um relatório foi gerado.

Quadro 35– Detalhamento do caso de uso `Gerar relatório` com as inconsistências mais encontradas

3.2.2 Diagrama de Classes Conceitual

No diagrama de classes da figura 7 é demonstrado o diagrama de classes conceitual, que mostra de forma macro a implementação do processamento dos arquivos.

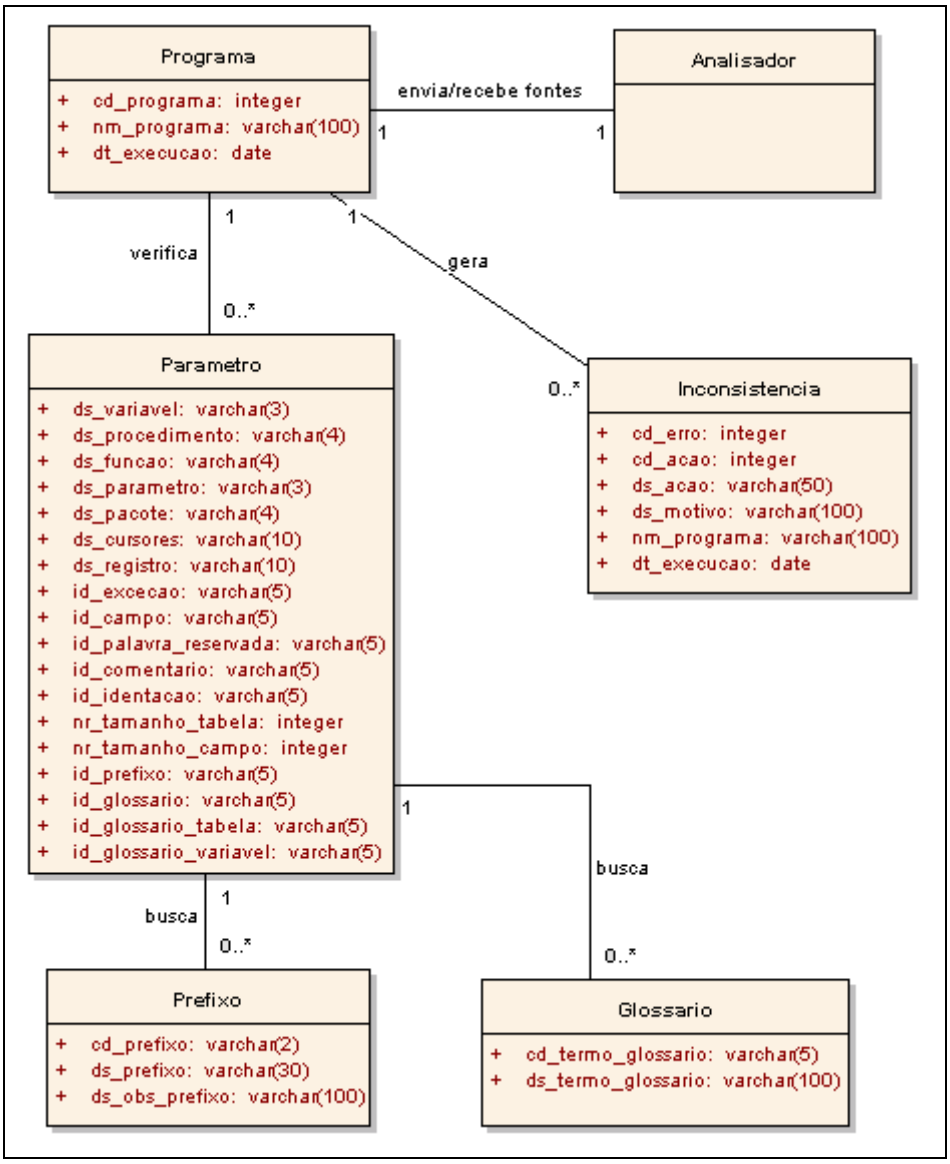


Figura 7 - Diagrama de Classes

3.2.3 Diagrama de Atividades

No diagrama de atividades apresentado na figura 8 é mostrado detalhes dos passos que a ferramenta utiliza para o caso de uso `Verificar Inconsistências`.

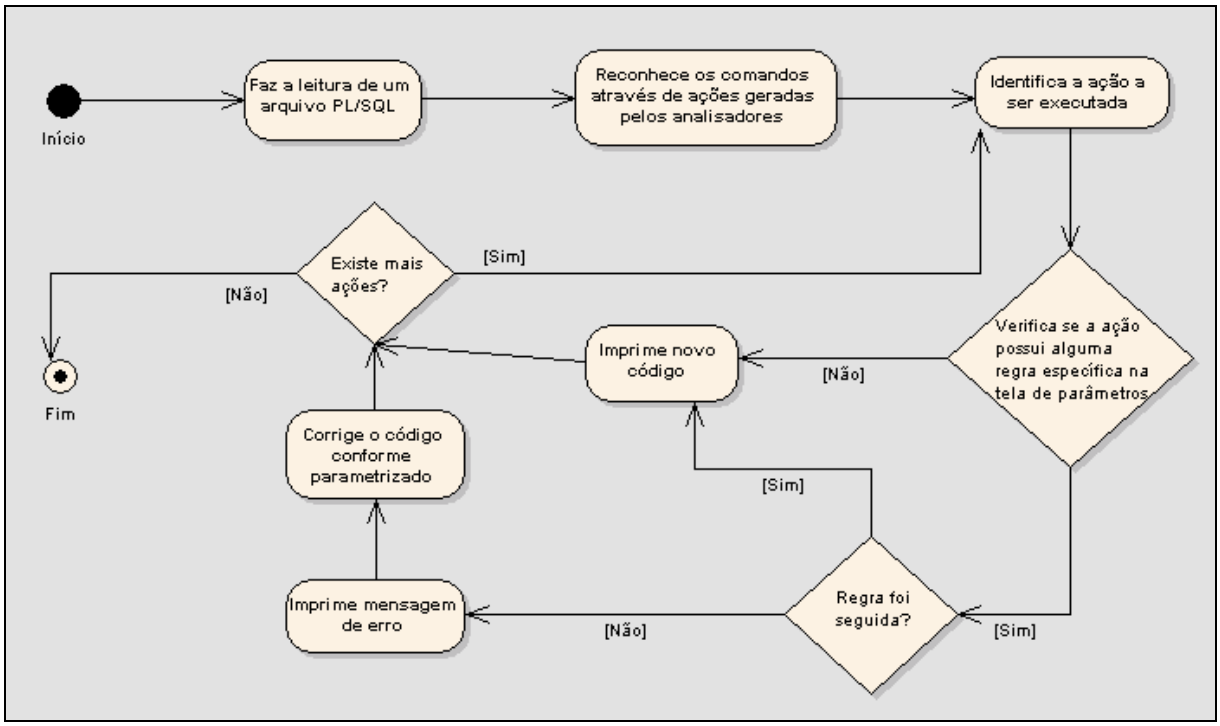


Figura 8 - Diagrama de atividades

3.3 IMPLEMENTAÇÃO

Nesta seção são apresentadas as técnicas e ferramentas utilizadas na fase de desenvolvimento do trabalho, bem como a operacionalidade da implementação.

3.3.1 Técnicas e ferramentas utilizadas

A ferramenta construída foi desenvolvida em Delphi 7. O código que faz as análises léxica, sintática e semântica foi gerado através da ferramenta GALS (GESSER, 2003).

3.3.2 Implementação da ferramenta

Para validar o código fonte escrito em PL/SQL, foram construídos analisadores léxico, sintático e semântico. Foi construída uma gramática da linguagem PL/SQL que foi utilizada como ponto inicial para análise de entrada da ferramenta. No quadro 31, pode ser visualizada

uma parte da gramática, porém ela completa está apresentada no apêndice A.

```
<PLSQL> ::=  <DECLARE>;
<DECLARE> ::= declare #1 <DECLARACAO> begin #1 <OPERACOES>;
              <DECLARACAO>::=  <DEC_VARIAVEL><DECLARACAO2><DECLARACAO>|
              <DEC_CURSORES><DECLARACAO2><DECLARACAO>|<COMENTARIO_1>| î;
<DECLARACAO2>::= <DEC_VARIAVEL><DECLARACAO2>|<DEC_CURSORES><DECLARACAO2>|î;
<DEC_VARIAVEL> ::= identificador #2 <DEC_VARIAVEL0> <DEC_VARIAVEL1>;
<DEC_VARIAVEL0> ::= identificador #3<DEC_VARIAVEL6>|î;
<DEC_VARIAVEL1>::= <DEC_VARIAVEL2>|<DEC_VARIAVEL3>|<DEC_VARIAVEL4>|
                   <DEC_VARIAVEL5>|<DEC_VARIAVEL>|î;
<DEC_VARIAVEL2>::= ";"#14|<ATRIBUIR>";"#14;
<DEC_VARIAVEL3>::= "("#3constante_inteira#3")"#3<DEC_VARIAVEL2>;
<DEC_VARIAVEL4>::= "%"#15<DEC_VARIAVEL7>";"#14;
<DEC_VARIAVEL5>::= exception #3";"#14;
<DEC_VARIAVEL6>::= "."#15identificador#15|î;
<DEC_VARIAVEL7>::= type #15|rowtype #15;
<ATRIBUIR>::= ":="#17 <ATRIBUIR_1>|<ATRIBUIR_1>;
<ATRIBUIR_1>::= <DADOS1>|null #1;
<ATRIBUIR_2>::= identificador#2<ATRIBUIR>";"#14;
<DADOS1>::= constante_inteira#18 | constante_real#19 | constante_string#20;
<DADOS2>::= ","#14<DADOS1><DADOS2>|î;
<DADOS3>::= constante_inteira#18|constante_real#19|constante_string#20|
            identificador#4 ;
<DEC_CURSORES>::= cursor#1 identificador#8 is#1 <SELECT>;
<OPERACOES> ::= <OPERACOES2>;
<OPERACOES2>::= <INSERT><OPERACOES2>|
                <UPDATE><OPERACOES2>|
                <DELETE><OPERACOES2>|
                <IF><OPERACOES2>|
                <COMANDOS><OPERACOES2>|
                <END><OPERACOES2>|
                <ATRIBUIR_2><OPERACOES2>|
                <SELECT_INTO><OPERACOES2>|
                <ABERTURA_CURSOR><OPERACOES2>|
                <COMENTARIO_1><OPERACOES2>|
                <LOOP><OPERACOES2>|î;
```

Quadro 36 – Gramática da linguagem PL/SQL

As ações semânticas identificadas no quadro 36 (símbolo # seguido de um número inteiro) bem como em toda a gramática, apresentam os significados descritos no quadro 37.

AÇÃO	SIGNIFICADO
1	identifica palavras reservadas
2	identifica o nome de variáveis
3	identifica o tipo de variáveis na declaração
4	identifica nome de campos da tabela
5	identifica o nome da tabela
6	identifica nome de parâmetros
7	identifica o nome da procedure
8	identifica o nome de cursores
9	identifica o nome de packages
10	comentário
11	identifica o nome de funções
12	identifica o alias da tabela
13	fim do programa
14	finaliza linha
15	tipo de variável baseada em campo de tabela
16	identifica o alias do campo da tabela
17	símbolos especiais
18	constante_inteira
19	constante_real
20	constante_string
21	identifica chamada de rotinas

Quadro 37 – Significado das ações semânticas

A ferramenta identifica cada comando do código fonte através das ações semânticas, configuradas na gramática do GALS e tratadas pela ferramenta, onde ela guarda os dados necessários, através do método *executeAction* como é mostrado um trecho do mesmo no quadro 38.

```
procedure TSemantico.executeAction(action : integer; const token : TToken);
begin
  case action of
    //#1: PALAVRAS RESERVADAS
    1: prc_acao_1(token.getLexeme, token.getPosition, action);

    //#2: NOME DE VARIÁVEL
    2: prc_valida_variavel(token.getLexeme, token.getPosition, action);

    //#3: TIPO DE VARIÁVEL
    3: begin
         if token.getLexeme = '(' then
         begin
           FrmPrincipal.tipo_declaracao := TRUE;
           FrmPrincipal.Concatena_Linha:=FrmPrincipal.Concatena_Linha+
           token.getLexeme;
         end
         else
           if FrmPrincipal.tipo_declaracao = FALSE then
             FrmPrincipal.Concatena_Linha:=FrmPrincipal.Concatena_Linha+'
             '+LowerCase(token.getLexeme)
           else
           begin
             FrmPrincipal.Concatena_Linha:=FrmPrincipal.Concatena_Linha+
             LowerCase(token.getLexeme);
             if token.getLexeme = ')' then
               FrmPrincipal.tipo_declaracao := FALSE;
           end;
       end;

  //#4: NOME DOS CAMPOS DA TABELA
    4: begin
        prc_valida_prefixo(token.getLexeme, token.getPosition, action);
        prc_valida_glossario(token.getLexeme, token.getPosition, action);
        prc_valida_nome_campo(token.getLexeme, token.getPosition, action);
        FrmPrincipal.Concatena_Linha:=FrmPrincipal.Concatena_Linha+
        token.getLexeme;
       end;
```

Quadro 38 – Implementação das ações semânticas

Todos os tokens que forem identificados pela ferramenta com a ação 2 são interpretados como nome de variáveis e estas são validadas de acordo com o método mostrado no quadro 39.

```
Procedure TSemantico.prc_valida_variavel(token:string;posicao:integer;
acao:integer);
var
  ww_variavel: string;
  ww_nova_variavel: string;
  ww_tamanho: integer;
  ww_tamanho_token: integer;
begin
 //chama rotina para verificar se o programa deve realizar identação ou não.
  prc_valida_identacao;
  //busca a linha em que se encontra o token.
  RetY(posicao);
  FrmPrincipal.linha_atual := linha;
  //busca o parêmetro cadastrado para a variável na tabele de parametro.
  with FrmParametros.sqlParametro, SQL do
  begin
    Close;
    Clear;
    Add('SELECT DS_VARIAVEL FROM PARAMETRO');
    Open;
    ww_tamanho:= length(FieldByName('DS_VARIAVEL').AsString);
    ww_nova_variavel := LowerCase(FieldByName('DS_VARIAVEL').AsString);
    //pega as primeiras posições do token, de acordo com o que foi cadastrado nos
    parâmetros.
    ww_variavel:= LeftStr(token,ww_tamanho);
    //Verifica se o parâmetro para a variável não foi cadastrado.
    IF FieldByName('DS_VARIAVEL').AsString = '' THEN
      ww_variavel := FieldByName('DS_VARIAVEL').AsString;
      //Verifica se o token é diferente ao do parâmetro cadastrado para nome de
      variável.
if UpperCase(ww_variavel) <> UpperCase(FieldByName('DS_VARIAVEL').AsString) then
  begin
    FrmProcessa.lbErros.Items.Add('Linha '+IntToStr(Linha)+': Variável '+token+'
    está com o nome incorreto;');
    //Tratamento de cor
    AlteraCor(posicao, (FrmPrincipal.nivel_atual-1), Length(token));
    prc_insere_inconsistencia(acao,'NOME  DE  VARIÁVEL','VARIÁVEL  ESTÁ  COM  NOME
INCORRETO');
    ww_tamanho_token := length(token);
    ww_nova_variavel    :=    ww_nova_variavel    +    Copy(token,    ww_tamanho+1,
ww_tamanho_token);
   if FrmPrincipal.linha_atual <> FrmPrincipal.linha_anterior then
     begin
      //verifica se a opção identação for marcada nos parâmetros.
      if FrmPrincipal.executa_identacao = true then
```

```
        FrmPrincipal.Concatena_Linha:=     get_nivel(FrmPrincipal.nivel_atual)     +
ww_nova_variavel//LowerCase(token.getLexeme);
  else
    FrmPrincipal.Concatena_Linha := ww_nova_variavel;
    FrmPrincipal.linha_anterior := linha;
  end
  else
    FrmPrincipal.Concatena_Linha:= FrmPrincipal.Concatena_Linha +' '+
    ww_nova_variavel;//LowerCase(token.getLexeme);
  end
else
  begin
    if FrmPrincipal.linha_atual <> FrmPrincipal.linha_anterior then
      begin
        //verifica se a opção identação for marcada nos parâmetros.
        if FrmPrincipal.executa_identacao = true then
         FrmPrincipal.Concatena_Linha:=    get_nivel(FrmPrincipal.nivel_atual)    +
LowerCase(token)
        else  //sem identação
          FrmPrincipal.Concatena_Linha:= LowerCase(token);
          FrmPrincipal.linha_anterior := linha;
        end
   else
     FrmPrincipal.Concatena_Linha:= FrmPrincipal.Concatena_Linha +' '+
LowerCase(token);
   end;
  end;
end;
```

Quadro 39 – Implementação da validação das palavras reservadas

No método `prc_valida_variavel`, a ferramenta verifica se a opção de validar variável foi marcada na tela de parâmetros, se foi então verifica se a variável do arquivo está dentro dos padrões estabelecidos pela empresa.

3.3.3 Operacionalidade da implementação

Nesta seção é apresentado um estudo de caso para demonstrar a funcionalidade da ferramenta. A figura 9 apresenta a tela principal da ferramenta.

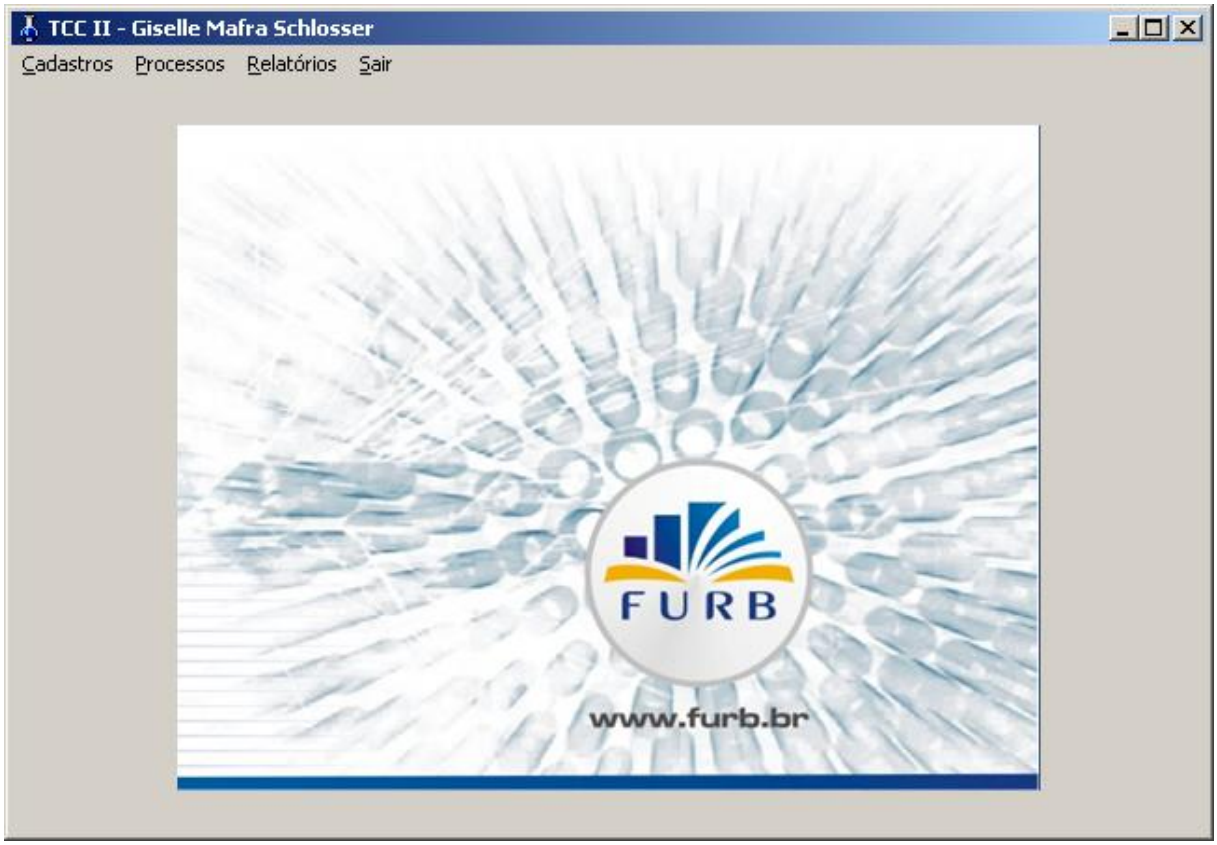


Figura 9 - Tela principal

Na figura 10, é apresentada a tela de cadastro de glossário de termos, onde os mesmos são utilizados durante a validação do código fonte, para identificar se o nome das tabelas, nome de campos e de variáveis possui o seu nome ou abreviação cadastrada nesta tela.

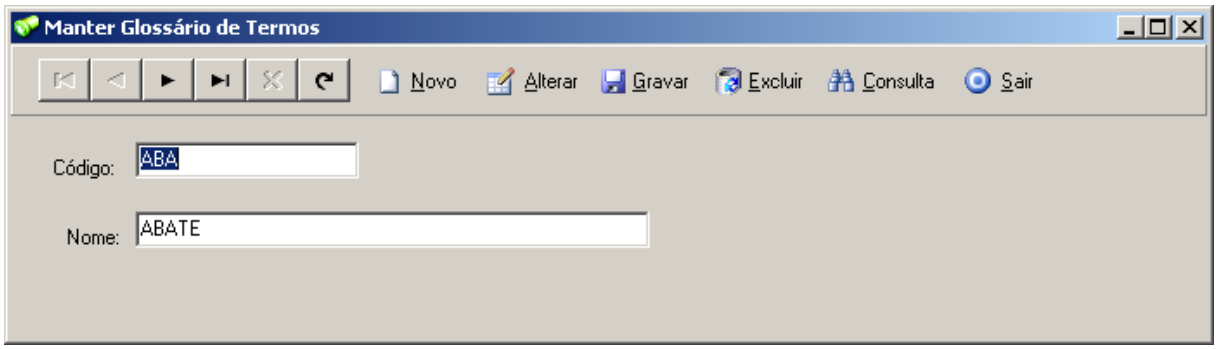


Figura 10 - Cadastro de glossário de termos

A figura 11 apresenta o cadastro de prefixos, estes que devem ser utilizados para a validação do nome de campos de tabelas, ou seja, se o campo for representar um código, por exemplo, então o mesmo deve se chamar “cd_” seguido de qualquer nome que esteja no glossário de termos (figura 10).

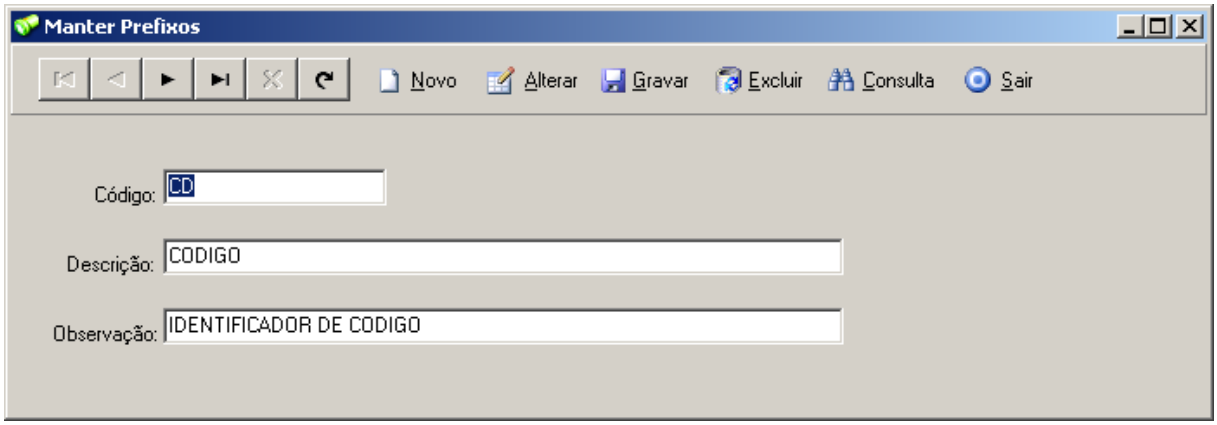


Figura 11 - Cadastro de prefixos

Na figura 12, é apresentada a tela de parâmetros, onde serão definidos pela empresa, permitindo que a mesma escolha o que a ferramenta deve validar no código fonte e criar os seus próprios padrões.

Os campos podem ser preenchidos ou não, dependo da necessidade do usuário, sendo que se o mesmo não for preenchido, então a ferramenta não irá validar. Abaixo, seguem detalhes dos campos:

a) variáveis: o usuário define como deve iniciar o nome de uma variável;
b) *procedures*: define como deve iniciar o nome de uma *procedure*;
c) *function*: neste campo, deve ser informado o início do nome de uma função;
d) parâmetros: define como deve iniciar o nome dos parâmetros;
e) *packages*: neste campo, deve-se informar como deverá ser o início do nome das *packages*;
f) cursores: o usuário define como deve iniciar o nome dos cursores;
g) nome campos (prefixo): se este campo estiver marcado, então a ferramenta leva em consideração que todo nome de campo de tabela deve ter no início do seu nome um prefixo, conforme foi cadastrado na tela de prefixos (figura 9);
h) nome campos (glossário): se estiver marcado, então a ferramenta deve levar em consideração que todo nome de campo de tabela deve ter em seu nome uma abreviação ou nome cadastrado na tela de glossário (figura 8);
i) nome tabela (glossário): se este campo estiver marcado, então a ferramenta deve validar se o nome da tabela tem em seu nome uma abreviação ou um nome que esteja cadastrado na tela de glossário (figura 8);
j) variáveis: se o campo estiver marcado, então a ferramenta vai validar se o nome da variável tem em seu nome palavras cadastradas na tela de glossário;

k) palavras reservadas maiúscula: se este campo estiver marcado, então a ferramenta deve verificar se todas as palavras reservadas no código fonte selecionado estão em maiúsculo;

l) comentários no início do programa: se o campo estiver marcado, então a ferramenta irá verificar se existe algum comentário no início do programa fonte;

m) identação: se este campo estiver marcado, então a ferramenta irá realizar a identação no novo arquivo que será gerado, contendo as correções;

n) final programa: se esta opção estiver marcada, então a ferramenta deve verificar se existe um tratamento de exceção no final do programa fonte lido;

o) `insert`: se este campo estiver marcado, então a ferramenta deve verificar se após uma inserção existe um tratamento de exceção;

p) `update`: se estiver marcado, então a ferramenta deve verificar se após uma alteração existe um tratamento de exceção;

q) `delete`: se este campo estiver marcado, então a ferramenta irá verificar se após uma exclusão existe um tratamento de exceção;

r) tabela: neste campo o usuário deve informar qual poderá ser o tamanho máximo para o nome de uma tabela. Se este campo não tiver preenchido a ferramenta desconsidera o tamanho;

s) campos: este campo possui a mesma regra do campo tabela, porém neste caso é para nome de campos.

A tela permite que somente seja cadastrado um parâmetro, somente permite alterar o que já está cadastrado ou inserir um novo se ainda não estiver cadastrado.

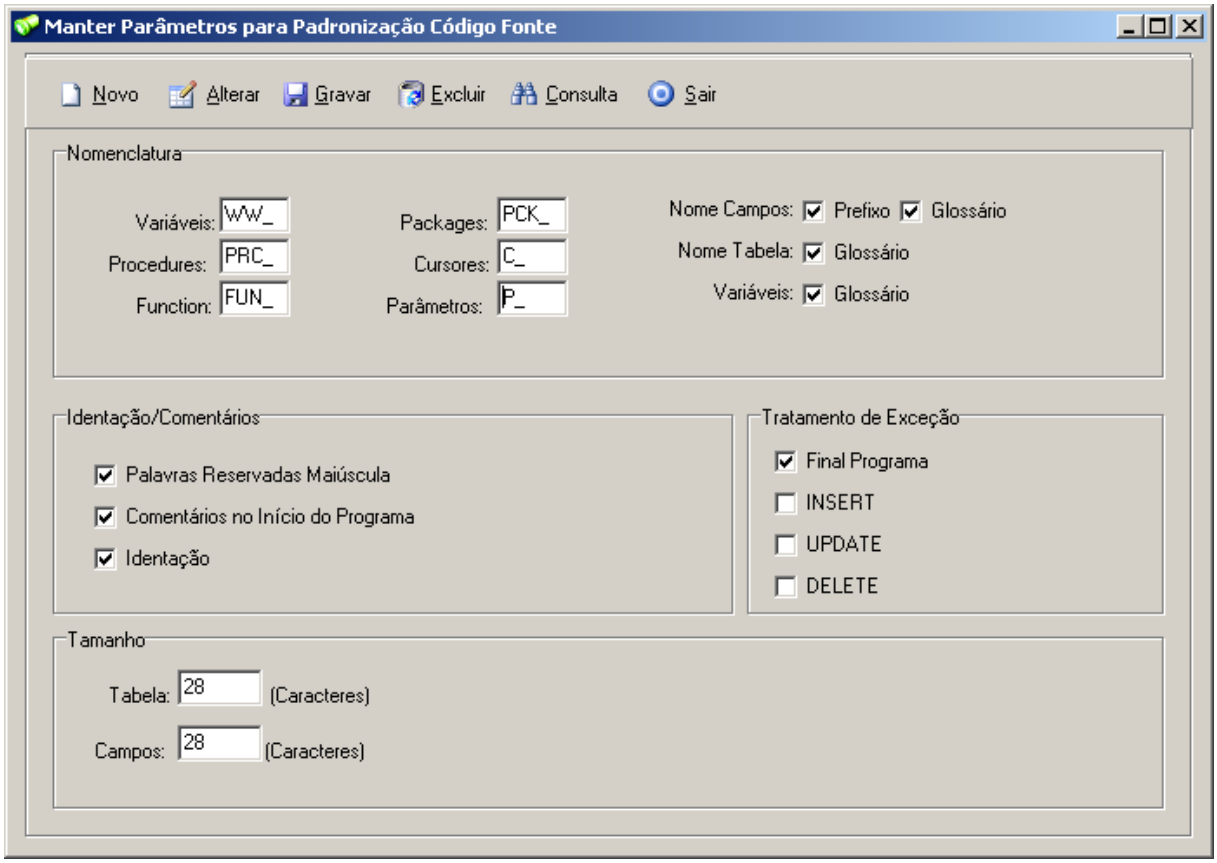


Figura 12 - Cadastro de parâmetros

A figura 13 apresenta a tela que transforma um arquivo .txt gerado pelo *forms* em outros arquivos menores. Nesta tela, o usuário deve selecionar um arquivo *txt* gerado pelo *forms* e pressionar o botão *gerar*. A ferramenta verificará todas as *procedures* e *functions* deste arquivo e irá gerá-las em outros arquivos separados, sendo que o nome do arquivo será o mesmo nome da *procedure* ou *function* que as mesmas possuem no arquivo origem. O objetivo de gerar em vários arquivos menores e não em somente um para que o processo ficasse mais automatizado, é justamente para não haver a necessidade de avaliar rotinas que não foram alteradas pelos programadores, realizando a validação somente no que foi alterado.

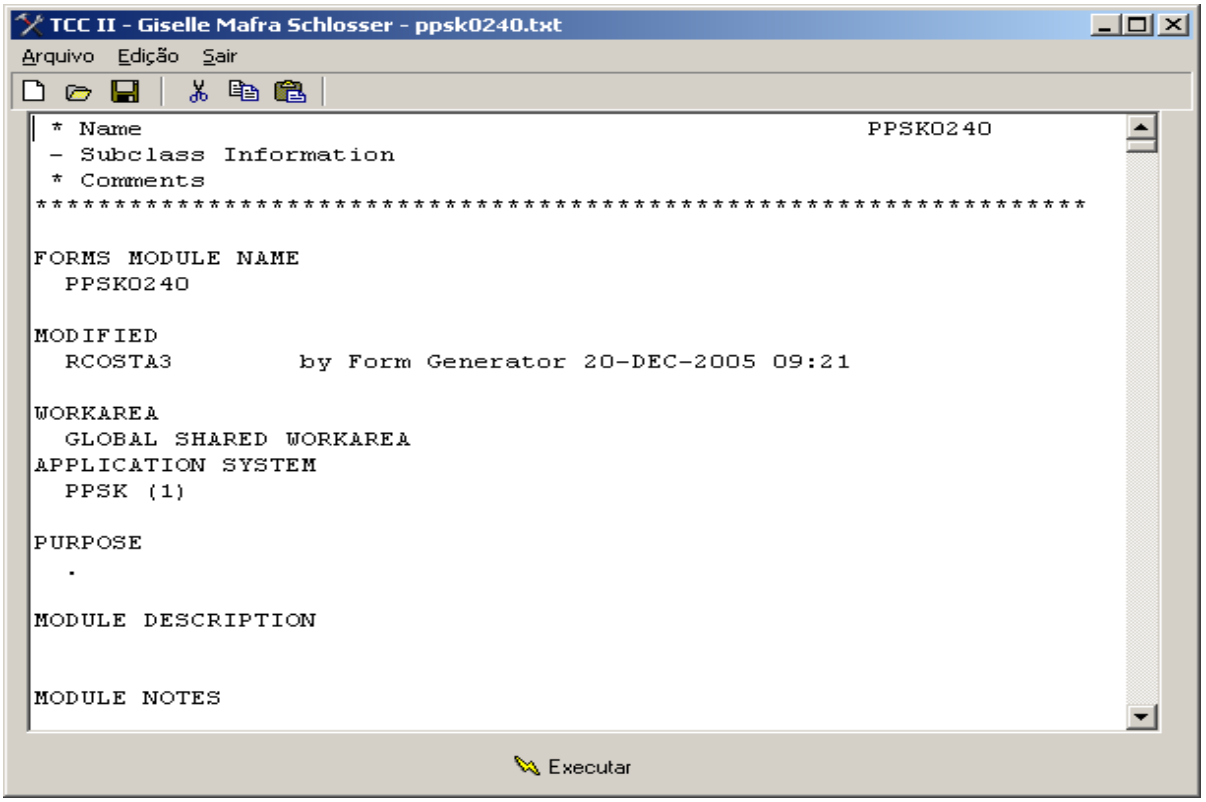


Figura 13 - Transformando arquivo `forms`

A figura 14 mostra a tela que irá realizar a validação do código fonte escrito em PL/SQL. O usuário deve selecionar um arquivo e o conteúdo deste arquivo será visualizado no campo em azul.

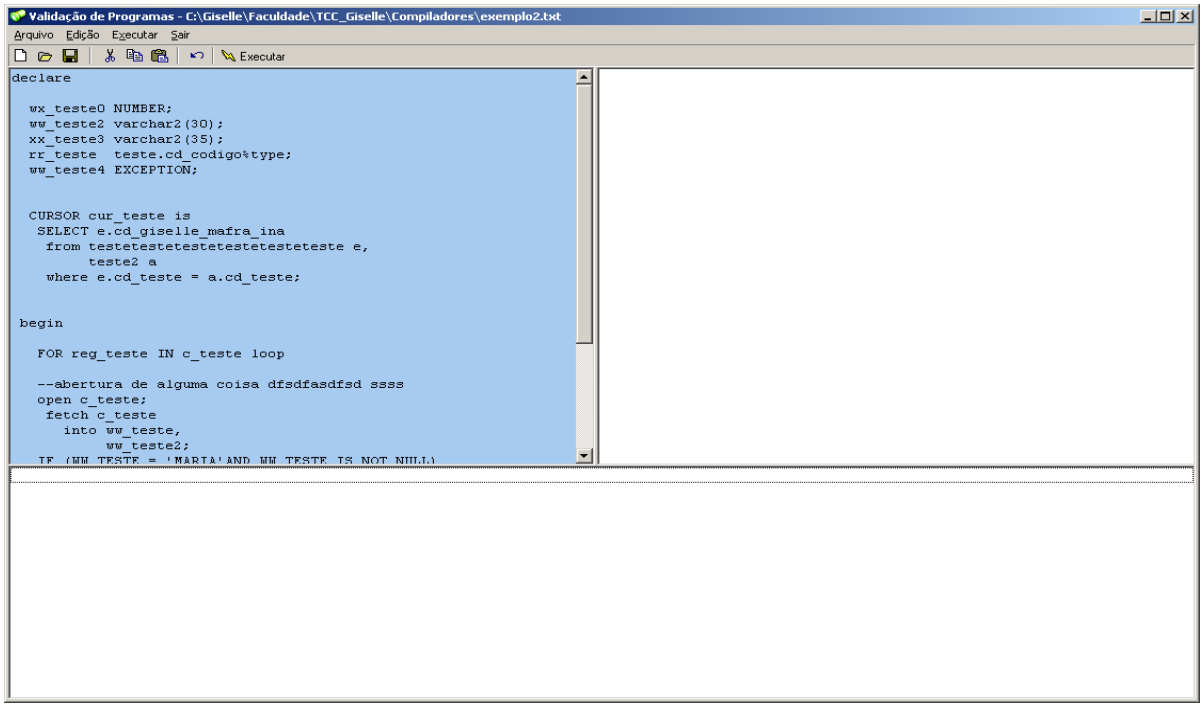


Figura 14 - Validação de programas

Após pressionar o botão `Executar`, o usuário poderá visualizar uma listagem com os

erros encontrados. Estes erros são apresentados linha a linha, conforme mostra a figura 15. Estes erros encontrados também são identificados no campo em azul, sendo que cada comando errado fica em vermelho após a execução.

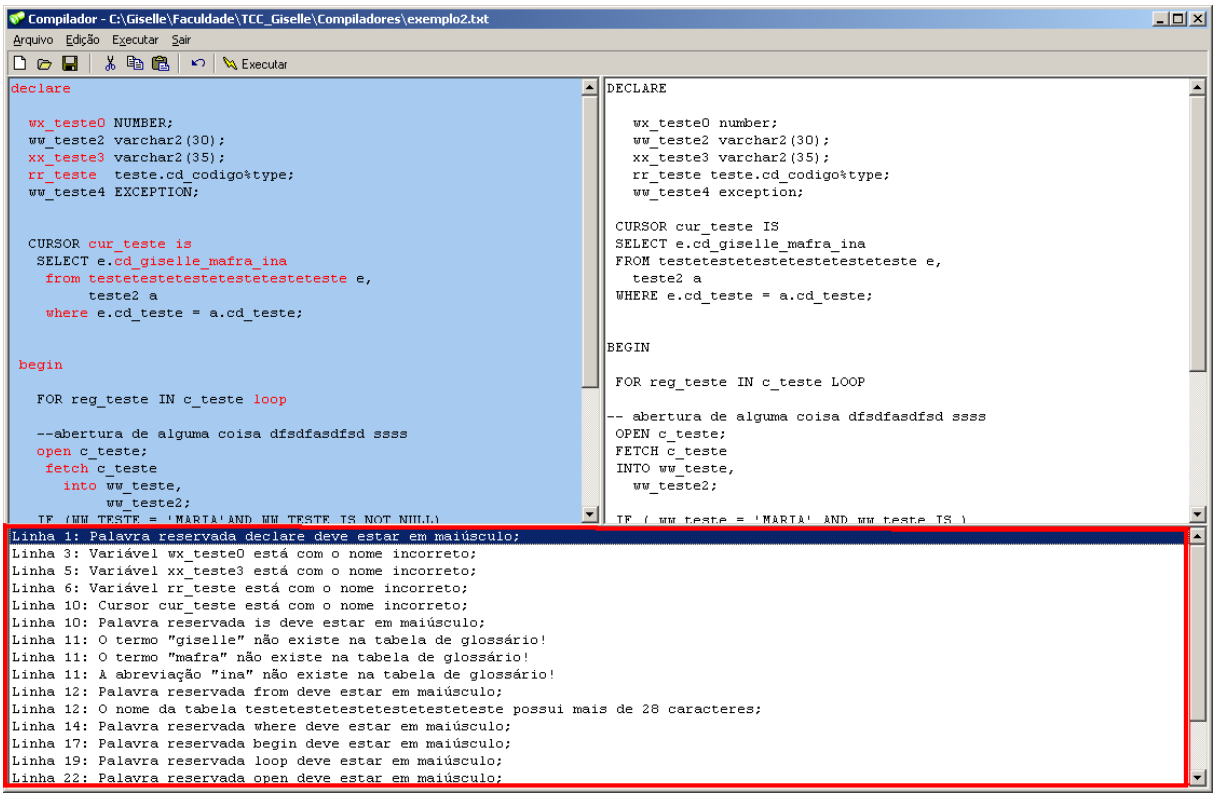


Figura 15 - Mensagens de erros geradas pela ferramenta

No campo à direita, o usuário pode visualizar a reestruturação do código selecionado, sendo que a reestruturação também obedece às regras dos parâmetros, ou seja, só será reestruturado o que tiver sido definido na tela de parâmetros.

A ferramenta também disponibiliza ao usuário a opção de identificar quais são os erros mais cometidos pelos programadores, ou seja, quais foram os padrões exigidos pela empresa que não estão sendo seguidos, através do relatório de inconsistências. Sendo que na figura 16, são visualizados os parâmetros que podem ser optados na execução no relatório. O usuário pode informar o nome do programa que foi verificado as inconsistências, o período inicial e final da data de execução e também as opções de listar as ações com mais erros ou então todas as ações. Se a opção “Ação com Mais Erros” for selecionada, então o usuário poderá visualizar um relatório contendo uma listagem da ação que teve mais erros de acordo com o restante dos parâmetros informados. Um exemplo deste relatório pode ser visualizado na figura 17. Se a opção “Todas Ações” for selecionada, então o relatório vai imprimir todas as ações que tiveram problemas durante a validação, este pode ser visualizado na figura 18.

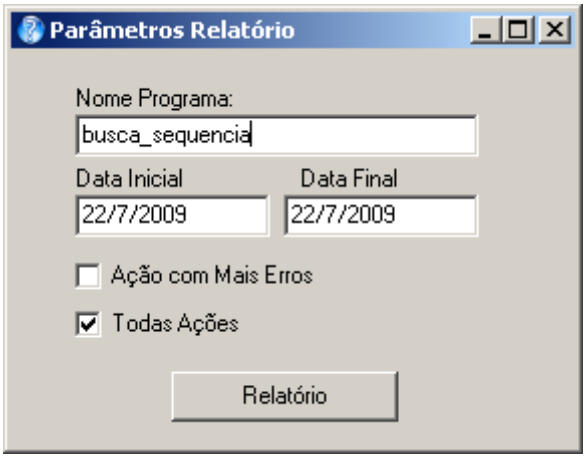


Figura 16 - Tela de Parâmetros do Relatório

Página: 1 **Relatório de Inconsistências** 23/7/2009 19:37:43

Programa	Data	Ação		Motivo	Linha	Comando
busca_sequencia	22/7/2009	1	Palavra reservada	A palavra reservada não está em maiúsculo.	11	is
busca_sequencia	22/7/2009	1	Palavra reservada	A palavra reservada não está em maiúsculo.	16	and
busca_sequencia	22/7/2009	1	Palavra reservada	A palavra reservada não está em maiúsculo.	13	from
busca_sequencia	22/7/2009	1	Palavra reservada	A palavra reservada não está em maiúsculo.	21	loop
busca_sequencia	22/7/2009	1	Palavra reservada	A palavra reservada não está em maiúsculo.	20	begin
busca_sequencia	22/7/2009	1	Palavra reservada	A palavra reservada não está em maiúsculo.	15	where
busca_sequencia	22/7/2009	1	Palavra reservada	A palavra reservada não está em maiúsculo.	37	return
busca_sequencia	22/7/2009	1	Palavra reservada	A palavra reservada não está em maiúsculo.	1	busca_sequencia
		Total por ação: 8				
		Total Geral: 8				

Figura 17 - Relatório da ação com mais erros

Página: 1 **Relatório de Inconsistências** 23/7/2009 19:37:43

Programa	Data	Ação		Motivo	Linha	Comando
busca_sequencia	22/7/2009	1	Palavra reservada	A palavra reservada não está em maiúsculo.	11	is
busca_sequencia	22/7/2009	1	Palavra reservada	A palavra reservada não está em maiúsculo.	16	and
busca_sequencia	22/7/2009	1	Palavra reservada	A palavra reservada não está em maiúsculo.	13	from
busca_sequencia	22/7/2009	1	Palavra reservada	A palavra reservada não está em maiúsculo.	21	loop
busca_sequencia	22/7/2009	1	Palavra reservada	A palavra reservada não está em maiúsculo.	20	begin
busca_sequencia	22/7/2009	1	Palavra reservada	A palavra reservada não está em maiúsculo.	15	where
busca_sequencia	22/7/2009	1	Palavra reservada	A palavra reservada não está em maiúsculo.	37	return
busca_sequencia	22/7/2009	1	Palavra reservada	A palavra reservada não está em maiúsculo.	1	busca_sequencia
		Total por ação: 8				
busca_sequencia	22/7/2009	2	Nome de variável	Variável está com nome incorreto	7	xx_teste4
busca_sequencia	22/7/2009	2	Nome de variável	Variável está com nome incorreto	8	xx_teste5
		Total por ação: 2				
busca_sequencia	22/7/2009	4	Nome de campo	O nome do campo, não possui o caracter "_"	24	nome
busca_sequencia	22/7/2009	4	Nome de campo	O nome do campo, não possui o caracter "_"	24	valor
busca_sequencia	22/7/2009	4	Nome de campo	O nome do campo, não possui o caracter "_"	37	false
		Total por ação: 3				
		Total Geral: 13				

Figura 18 - Relatório de todas as ações que apresentaram erros

3.4 RESULTADOS E DISCUSSÃO

A funcionalidade do protótipo atingiu a meta esperada. Foram utilizados 20 arquivos de teste com exemplos de código fonte escritos em PL/SQL. Estes exemplos foram buscados de empresas da região e outros foram criados manualmente para atingir exatamente as situações em que o programa deveria tratar.

No apêndice A foi incluído a gramática da linguagem PL/SQL criada no GALS a fim de tratar os comandos da linguagem conforme as ações enviadas pelo GALS.

No apêndice B foram incluídas as listagens de 3 programas que foram submetidos a geração de validação da ferramenta. Antes de cada exemplo, tem uma tela de parâmetros para que seja visualizado quais foram os parâmetros que foram levados em consideração nos exemplos do apêndice.

A maneira de execução do protótipo é similar ao utilizado em Dalmolin (2000) e Dolla (2001), porém neste protótipo são tratadas as nomenclaturas, permitindo também que as mesmas sejam definidas pelo próprio usuário da ferramenta e também que essa nomenclatura siga um padrão de prefixos e glossários, que também são cadastrados pelo usuário. Outro ponto diferenciado deste protótipo, foi a criação de uma gramática da linguagem PL/SQL, permitindo que um fonte fosse tratado, de acordo com cada comando encontrado. A ferramenta permite que seja identificado quais são os maiores problemas no desenvolvimento do código fonte, através de um relatório de inconsistências, permitindo auxiliar o usuário a identificar quais são os principais erros cometidos pelos programadores durante o desenvolvimento, tratando de padrões de codificação.

4 CONCLUSÕES

A ferramenta construída pode auxiliar as empresas a definirem padrões a serem estabelecidos e utilizados por desenvolvedores que prestam serviços. Estabelecer padrões no desenvolvimento de um software, além de obter um nível de qualidade satisfatório, facilita o entendimento do negócio, bem como auxilia os desenvolvedores em futuras manutenções, e mais do que isso, codificar corretamente é uma condição essencial para poder trabalhar em equipe.

A ferramenta apresentada analisa um código fonte escrito em PL/SQL, verificando a qualidade do software, analisando se todos os padrões estabelecidos pela empresa estão sendo utilizados. As informações necessárias para a verificação destes padrões serão extraídas dos códigos fonte escritos em PL/SQL através de analisadores léxico e sintático.

Em relação aos trabalhos correlatos apresentados, pode-se afirmar que a ferramenta proposta possui várias características encontras nas ferramentas citadas, bem como a verificação de padronização do código fonte, a utilização incorreta de determinados comandos, a partir de estatísticas de utilização. Porém, o que diferencia a ferramenta proposta é dar a flexibilidade às empresas cadastrarem os seus próprios padrões, permitindo que a mesma possa ser utilizada por diversas empresas. Outro ponto que não foi tratado nas ferramentas correlatas é a verificação da nomenclatura de variáveis, de campos, de nomes de tabelas, identificação do tamanho dos nomes das tabelas e dos campos. E também a possibilidade de ter um relatório para identificar os erros mais freqüentes cometidos pelos programadores.

4.1 EXTENSÕES

Como extensões para este trabalho sugere-se: aumentar a gramática da linguagem PL/SQL no GALS, visto que alguns comandos não foram tratados neste trabalho e também em desenvolver uma gramática para comandos do Forms, pois os arquivos extraídos do Forms utilizados como exemplo neste trabalho, apenas utilizavam comandos específicos da linguagem PL/SQL.

REFERÊNCIAS BIBLIOGRÁFICAS

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

DALMOLIN, Denis A. **Ferramenta de apoio a reestruturação de código fonte em linguagem C++ baseado em padrões de legibilidade**. 2000. 74 f. Trabalho de Conclusão de Curso (Bacharelado em Ciências da Computação) - Centro de Ciências Exatas e Naturais, Universidade Regional de Blumenau, Blumenau.

DOLLA, Dyckson D. **Ferramenta de apoio a reestruturação de código fonte em linguagem PL\SQL baseado em padrões de legibilidade.** 2001.72 f. Trabalho de Conclusão de Curso (Bacharelado em Ciências da Computação) - Centro de Ciências Exatas e Naturais, Universidade Regional de Blumenau, Blumenau.

FANDERUFF, Damaris. **Oracle 8i utilizando SQL* plus e PL/SQL.** SÃO Paulo: Makron Books, 2000.

GESSER, Carlos E. **GALS**: gerador de analisadores léxicos e sintáticos. [S.l.], 2003. Disponível em: <http://sourceforge.net/projects/gals>. Acesso em: 10 fev. 2009.

HIEBERT, Dennis. **Protótipo de um compilador para a linguagem PL/SQL**. 2003. 52 f. Trabalho de Conclusão de Curso (Bacharelado em Ciências da Computação) – Centro de Ciências Exatas e Naturais, Universidade Regional de Blumenau, Blumenau. Disponível em: <http://www.bc.furb.br/docs/MO/2002/266472_1_1.pdf> Acesso em: 30 mar. 2009.

KOSCIANSKI, André; SOARES, Michel S. **Qualidade de software:** aprenda as metodologias e técnicas mais modernas para o desenvolvimento de software. São Paulo: Novatec, 2006.

MAAS, Julio A. **Gerador de documentação e apoio a padronização de softwares implementados na linguagem Progress 4GL.** 2004. 64 f. Trabalho de Conclusão de Curso (Bacharelado em Ciências da Computação) - Centro de Ciências Exatas e Naturais, Universidade Regional de Blumenau, Blumenau.

MARCOS, Adriana Fronza. **Ferramenta de apoio à automatização de testes através do testcomplete para programa desenvolvidos em Delphi.** 2007. 76 f. Trabalho de Conclusão de Curso (Bacharelado em Ciências da Computação) – Centro de Ciências Exatas e Naturais, Universidade Regional de Blumenau, Blumenau. Disponível em: < http://campeche.inf.furb.br/tccs/2007-I/2007-1adrianafronzamarcosvf.pdf > Acesso em: 10 mar. 2009.

OLIVEIRA, Celso H. P. **Oracle 9i PL/SQL:** guia de consulta rápida. São Paulo: Novatec, 2002.

SOUZA, Marco A. **SQL, PL/SQL, SQL * Plus:** manual de referência completo e objetivo. Rio de Janeiro: Ciência Moderna, 2004.

STAA, Arndt von. **Programação modular:** desenvolvendo programas complexos de forma organizada e segura. Rio de Janeiro: Campus, 2000.

APÊNDICE A – Gramática da linguagem PL/SQL desenvolvida no GALS

No Apêndice A, pode ser visualizada a gramática completa da linguagem PL/SQL criada no GALS, conforme mostra o quadro 40.

```
//#1: Identifica palavras reservadas;
//#2: Para o bloco PL/SQL identifica o nome de variáveis;
//#3: Para o bloco PL/SQL identifica o tipo de variáveis;
//#4: Identifica nome de campos de tabela em select ou insert
//#5: Nome da tabela no select, insert, delete
//#6: Identifica parâmetros
//#7: Identifica o nome da procedure
//#8: Para o bloco PL/SQL identifica o nome de cursores;
//#9: identifica o nome de package
//#10: comentário
//#11: identifica o nome de funções;
//#12: identifica o alias da tabela
//#13: fim do programa
//#14: finaliza linha
//#15: tipo de variável baseada em campos de tabela.
//#16: identifica o alias do campo da tabela.
//#17: símbolos especiais
//#18: constante_inteira
//#19: constante_real
//#20: constante_string
//#21: identifica chamada de rotinas

<PLSQL> ::=  <DECLARE>;

<DECLARE> ::=  <COMENTARIO><DECLARE2>;
<DECLARE2>::=  declare #1<COMENTARIO> <DECLARACAO> begin #1 <OPERACOES>|
                procedure#1 identificador#7 <PARAMETROS_1> is#1 <DECLARACAO> begin
#1 <OPERACOES>|
                function#1 identificador#21 <PARAMETROS_1> return#1 identificador#3
is#1 <DECLARACAO> begin #1 <OPERACOES>|
             create#1 or#1 replace#1 package#1 identificador#9 is#1
             <DECLARACAO_PACKAGE>
             end identificador#9";"#17 <CORPO_PACKAGE>;

<DECLARACAO>::=  <DEC_VARIAVEL><DECLARACAO2><DECLARACAO>|
                 <DEC_CURSORES><DECLARACAO2><DECLARACAO>|ĩ;

<DECLARACAO2>::= <DEC_VARIAVEL><DECLARACAO2>|<DEC_CURSORES><DECLARACAO2>|ĩ;

<DECLARACAO_PACKAGE>::= <COMENTARIO><DECLARACAO_PACKAGE2>;
<DECLARACAO_PACKAGE2>::= procedure#1 identificador#7 <PARAMETROS_1>";"
                         <DECLARACAO_PACKAGE>|
                         function#1 identificador#21 <PARAMETROS_1> return#1
identificador#3";"<DECLARACAO_PACKAGE>|ĩ;

<CORPO_PACKAGE>::= create#1 or#1 replace#1 package#1 body#1 identificador#9 is#1
                    <DECLARACAO><CORPO_PACKAGE2>;
<CORPO_PACKAGE2>::= procedure#1 identificador#7 <PARAMETROS_1> is#1 <DECLARACAO>
                   begin #1 <OPERACOES><CORPO_PACKAGE2>|
                   function#1 identificador#21 <PARAMETROS_1> return#1
                   identificador#3 is#1
                   <DECLARACAO> begin #1 <OPERACOES><CORPO_PACKAGE2>|ĩ;

<DEC_VARIAVEL> ::= identificador #2 <DEC_VARIAVEL0> <DEC_VARIAVEL1>;
<DEC_VARIAVEL0> ::= identificador #3<DEC_VARIAVEL6>|ĩ;
<DEC_VARIAVEL1>::= <DEC_VARIAVEL2>|<DEC_VARIAVEL3>|<DEC_VARIAVEL4>|
                   <DEC_VARIAVEL5>|<DEC_VARIAVEL>|ĩ;
<DEC_VARIAVEL2>::= ";"#14|<ATRIBUIR>";"#14;
```

```
<DEC_VARIAVEL3>::= "("#3constante_inteira#3")"#3<DEC_VARIAVEL2>;
<DEC_VARIAVEL4>::= "%"#15<DEC_VARIAVEL7>";"#14;
<DEC_VARIAVEL5>::= exception #3";"#14;
<DEC_VARIAVEL6>::= "."#15identificador#15|î;
<DEC_VARIAVEL7>::= type #15|rowtype #15;
<PARAMETROS_1>::= "("#17<PARAMETROS_2><PARAMETROS_4>|î;
<PARAMETROS_2>::= identificador#6<PARAMETROS_3><PARAMETROS_3>identificador#3;
<PARAMETROS_3>::= in#1|out#1|î;
<PARAMETROS_4>::= ")"#17|","#17<PARAMETROS_2><PARAMETROS_4>;

<ATRIBUIR>::= ":="#17 <ATRIBUIR_1>|<ATRIBUIR_1>;
<ATRIBUIR_1>::= <DADOS1>|null #1;
<ATRIBUIR_2>::= identificador#2<ATRIBUIR>";"#14;
<ATRIBUIR_3>::= identificador#2<ATRIBUIR>;

<DADOS1>::= constante_inteira#18 | constante_real#19 | constante_string#20;
<DADOS2>::= ","#14<DADOS1><DADOS2>|î;
<DADOS3>::= constante_inteira#18 | constante_real#19 | constante_string#20|
            identificador#4 ;

<DEC_CURSORES>::= <COMENTARIO>cursor#1 identificador#8 is#1 <SELECT>;

<OPERACOES> ::= <OPERACOES2>;
<OPERACOES2>::= <INSERT><OPERACOES2>|
                <UPDATE><OPERACOES2>|
                <DELETE><OPERACOES2>|
                <IF><OPERACOES2>|
                <COMANDOS><OPERACOES2>|
                <END><OPERACOES2>|
                <ATRIBUIR_2><OPERACOES2>|
                <SELECT_INTO><OPERACOES2>|
                <ABERTURA_CURSOR><OPERACOES2>|
                <COMENTARIO_1><OPERACOES2>|
                <LOOP><OPERACOES2>|
                <RETURN><OPERACOES2>|î;

<SELECT> ::= <SELECT_1><SELECT_FROM>;
<SELECT_1>::= select#1 <CAMPO_TABELA>;
<SELECT_FROM>::= from#1 <SELECT_2>;
<SELECT_2> ::= <FROM><SELECT_3>;
<FROM> ::= identificador#5 <FROM_1>;
<FROM_1>::= ";"#14|identificador#12 <FROM_1>|","#17identificador#5<FROM_1>|î;

<SELECT_3> ::=where#1 <WHERE><SELECT_4><SELECT_5>;
<SELECT_4> ::=group#1 by#1 <GROUP_BY>|î;
<SELECT_5>::= order#1 by#1 <ORDER_BY>|î;

<WHERE> ::= <WHERE_1>identificador #4<WHERE_2>;
<WHERE_1>::= <FUNCOES_GRUPO>|identificador#16"."#16|î;
<WHERE_2>::= <SIMBOLOS> <WHERE_1><DADOS3><WHERE_2>|
               and #1 <WHERE_1>identificador #4 <WHERE_2>|
               ";"#14|
               <WHERE>|
               <COMANDOS><WHERE_2>|î;

<COMANDOS>::= commit#1";"#14| rollback#1";"#14|is#1 <COMANDOS_2>|begin#1;
<COMANDOS_2>::= null| not null;

<INTO>::= into#1 identificador#2<INTO>|","#14identificador#2<INTO>|î;

<SELECT_INTO>::= <SELECT_1><INTO><SELECT_FROM>;

<CAMPO_TABELA> ::=  identificador#16"."#16identificador#4 <CAMPO_TABELA1>|
                    <FUNCOES_GRUPO><CAMPO_TABELA>|
                    <DECODE><CAMPO_TABELA1>|î;

<CAMPO_TABELA1>::= identificador#4<CAMPO_TABELA1><CAMPO_TABELA>|
                   ","#14 <CAMPO_TABELA>|
```

```
                    <DADOS1><CAMPO_TABELA>|
                    î;
<GROUP_BY>::=  identificador#16"."#16identificador#4<GROUP_BY1>;
<GROUP_BY1>::= ";"#14|","#14<GROUP_BY>|î;
<ORDER_BY> ::= identificador#16"."#16identificador#4<ORDER_BY1>;
<ORDER_BY1>::= ";"#14|","#14<ORDER_BY>|î;

<SIMBOLOS> ::=
"="#17|"<"#17|">"#17|"<>"#17|"=>"#17|"=<"#17|">="#17|"<="#17|":"#17|"."#17|"'"#17;

<FUNCOES_GRUPO>::= min#1"("#17 identificador #4")"#17<FUNCOES_GRUPO>|
                    max#1"("#17 identificador #4")"#17<FUNCOES_GRUPO>|
                    count#1"("#17 identificador #4")"#17<FUNCOES_GRUPO>|
                   sum#1"("#17 identificador #4")"#17<FUNCOES_GRUPO>|
                    avg#1"("#17 identificador #4")"#17<FUNCOES_GRUPO>|
                    nvl#1"("#17 identificador#16"."#16identificador
#4","#17<DADOS3>")"#17<ALIAS><FUNCOES_GRUPO>|
<SELECT_FROM>;

<ALIAS>::= identificador#12<ALIAS>|","#14|î;
<DECODE>::= decode#1"("#17identificador#4","#17<DADOS1>","#17<DADOS1> ")"#17;

<INSERT> ::= insert #1 into#1 identificador#5
             "(" #17<IDENTIFICADOR1>")"#17
             values#1
             "("#17 <IDENTIFICADOR2> ")"#17 ";"#14<TRATAMENTO_EXCECAO>;

<IDENTIFICADOR1>::= identificador#4<IDENTIFICADOR1> |
                     ","#17<IDENTIFICADOR1>|î;
<IDENTIFICADOR2>::= ","#17<IDENTIFICADOR2>|
                     constante_inteira#18<IDENTIFICADOR2>|
                     constante_real#19<IDENTIFICADOR2>|
                     constante_string#20<IDENTIFICADOR2>|
                     identificador#2|î;

<IF>::= if#1 <IF_1> then#1;
<IF_1>::= <IF_2><IF_3><IF_1>|and#1 <IF_1>|or#1 <IF_1>|
          "("#17<IF_1>|")"#17<IF_1>|î;
<IF_2>::= identificador#2|<DADOS1>;
<IF_3>::=  <SIMBOLOS> <IF_2>|<COMANDOS>;

<END>::= end #1 <END_2>;
<END_2>::= ";"#14|if#1 ";"#14|loop#1 ";"#14|identificador#7 ";";

<COMENTARIO>::= <COMENTARIO_1>|î;
<COMENTARIO_1>::= "-"#10<COMENTARIO_2>|"/"#10<COMENTARIO_2>;
<COMENTARIO_2>::="-"#10<COMENTARIO_2>|identificador#10<COMENTARIO_2>|
                 <DADOS1><COMENTARIO_2>|
                 "*"#10<COMENTARIO_2>|"/"#10|<SIMBOLOS><COMENTARIO_2>|î;

<DELETE> ::= delete #1 <DELETE_1> identificador #5 <DELETE_2>;
<DELETE_1>::= from #1|î;
<DELETE_2>::= ";"#14<TRATAMENTO_EXCECAO>|where #1 <WHERE>;

<UPDATE> ::= update #1 identificador #5
             set #1 identificador #4"="<DADOS3> <UPDATE1>;
<UPDATE1>::= where #1 <WHERE>|";"#14<TRATAMENTO_EXCECAO>|","#14identificador
#4"="#17<DADOS3><UPDATE1>|î;

<ABERTURA_CURSOR>::= open #1 identificador#8";"#14
                      fetch #1 identificador#8
                      <INTO> ";"#14|
                      close #1 identificador#8";"#14;

<LOOP>::= for#1 identificador#11 in#1 identificador#8 loop #1;

<RETURN>::= return#1 <DADOS3>";"#17;
```

```
<TRATAMENTO_EXCECAO>::= exception#1
                          when#1 others#1 then#1<TRATAMENTO_EXCECAO2>";"#17|
                         <TRATAMENTO_EXCECAO2>|î;

<TRATAMENTO_EXCECAO2>::=<ATRIBUIR_3><TRATAMENTO_EXCECAO3> end#1|

raise_application_error#1"("#17<DADOS1>","#17<DADOS3><TRATAMENTO_EXCECAO3>")"#17
                       ";"#17end#1;
<TRATAMENTO_EXCECAO3>::=  "||"#17 <TRATAMENTO_EXCECAO3>|
                            ";"#17|
                            <DADOS3>|
                            sqlerrm#1<TRATAMENTO_EXCECAO3>|î;
 //"."#17 identificador#22<CHAMADA_ROTINAS2>|
<CHAMADA_ROTINAS>::= rotina#22<CHAMADA_ROTINAS2>;
<CHAMADA_ROTINAS2>::=  identificador#22<CHAMADA_ROTINAS3>|";";
<CHAMADA_ROTINAS3>::= ")"#17";"|"=>"#17<DADOS3><CHAMADA_ROTINAS3>|
                        ","#17identificador#22<CHAMADA_ROTINAS3>;
```

Quadro 40 – Gramática da linguagem PL/SQL

APÊNDICE B – Exemplos de código fonte

Para demonstrar mais detalhes dos parâmetros e tratamentos utilizados neste trabalho, são mostrados nos quadros 41 e 42, exemplos de código fonte contendo o mesmo antes e depois da validação da ferramenta, sendo que antes de cada quadro são visualizadas telas específicas de parâmetros para cada caso exemplificado.

A figura 17 demonstra os parâmetros utilizados para a geração do exemplo do quadro 41.

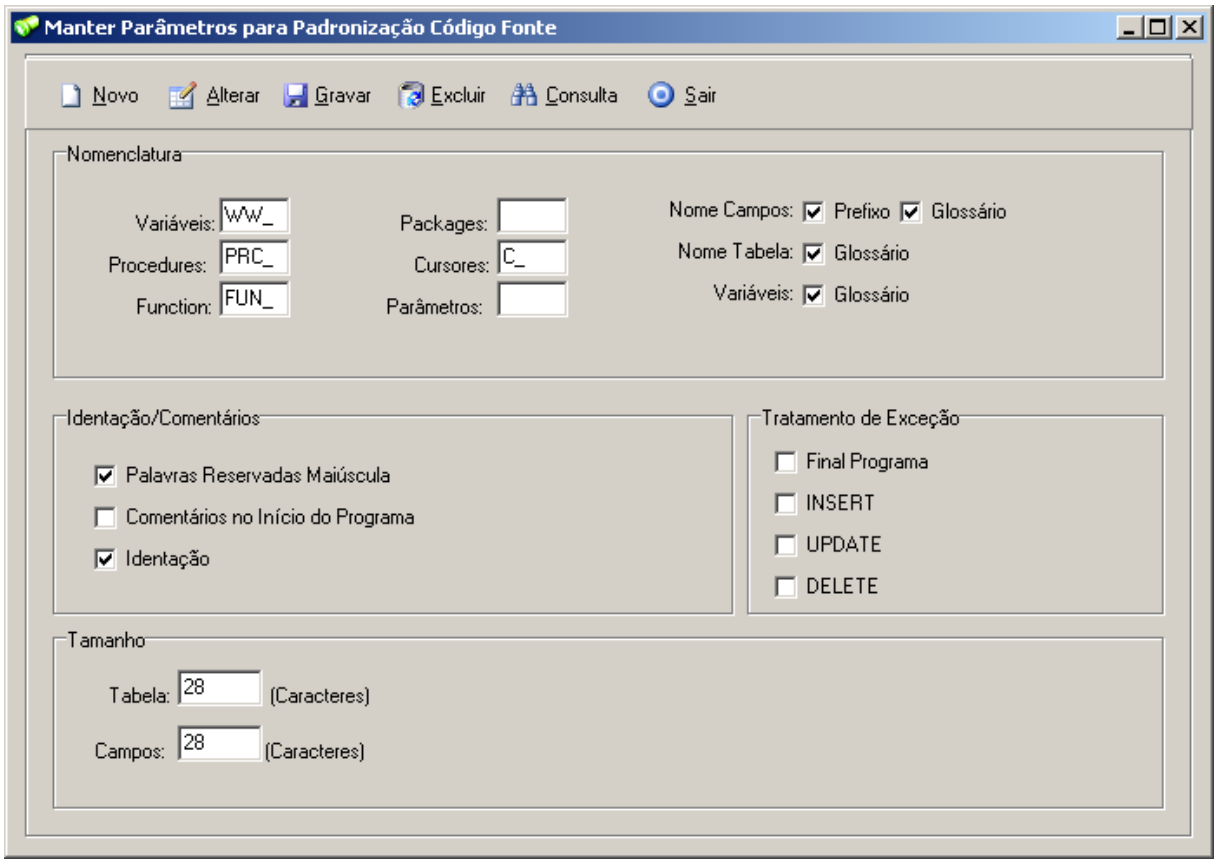


Figura 17 - Tela de parâmetros do relatório – exemplo 1

```
--ANTES

declare
  wx_empresa number;
  wx_nm_empresa varchar(35);
  ww_filial  varchar2(30);
  ww_nome_filial varchar2(100);
  xx_base_fornecedor  fornecedor.cd_base_fornecedor%type;
  rr_estab_fornecedor fornecedor.cd_estab_fornecedor%type;
  ww_erro exception;
  sp_valor number;

  CURSOR cur_busca_filial is
   SELECT a.cd_empresa
         ,a.nm_empresa
         ,e.cd_filial
          e.nm_filial
    from filial_cgc e,
         empresa a
    where e.cd_empresa = a.cd_empresa;

begin
   --busca dados empresa e filial
   open cur_busca_filial;
    fetch cur_busca_filial
      into wx_empresa,
           wx_nm_empresa,
           ww_filial,
           ww_nome_filial;
   IF (wx_empresa = 30 AND ww_filial = 109)
   OR nm_empresa = 'TESTE' THEN
    sp_valor := 300;
   END IF;
   INSERT INTO pagamento_empresa_filial_acerto
   (cd_filial, nm_filial, cd_empresa, nm_empresa, vl_filial)
   VALUES(ww_filial,ww_nome_filial,wx_empresa,wx_nm_empresa,sp_valor);
 END;

--DEPOIS

DECLARE
  ww_empresa number;
  ww_nm_empresa varchar(35);
  ww_filial  varchar2(30);
  ww_nome_filial varchar2(100);
```

```
  ww_base_fornecedor  fornecedor.cd_base_fornecedor%type;
  ww_estab_fornecedor fornecedor.cd_estab_fornecedor%type;
  ww_erro exception;
  ww_valor number;

  CURSOR c_busca_filial IS
   SELECT a.cd_empresa
         ,a.nm_empresa
         ,e.cd_filial
          e.nm_filial
    FROM filial_cgc e,
         empresa a
    WHERE e.cd_empresa = a.cd_empresa;

BEGIN
   --busca dados empresa e filial
   OPEN cur_busca_filial;
    FETCH cur_busca_filial
      INTO ww_empresa,
           ww_nm_empresa,
           ww_filial,
           ww_nome_filial;

   IF (ww_empresa = 30 AND ww_filial = 109)
   OR nm_empresa = 'TESTE' THEN
    ww_valor := 300;
   END IF;

   INSERT INTO pagamento_filial
   (cd_filial, nm_filial, cd_empresa, nm_empresa, vl_filial)
   VALUES(ww_filial,ww_nome_filial,ww_empresa,ww_nm_empresa,ww_valor);
 END;
```

Quadro 41– Exemplo 1

Na figura 18 demonstra os parâmetros utilizados para a execução do exemplo do quadro 42.

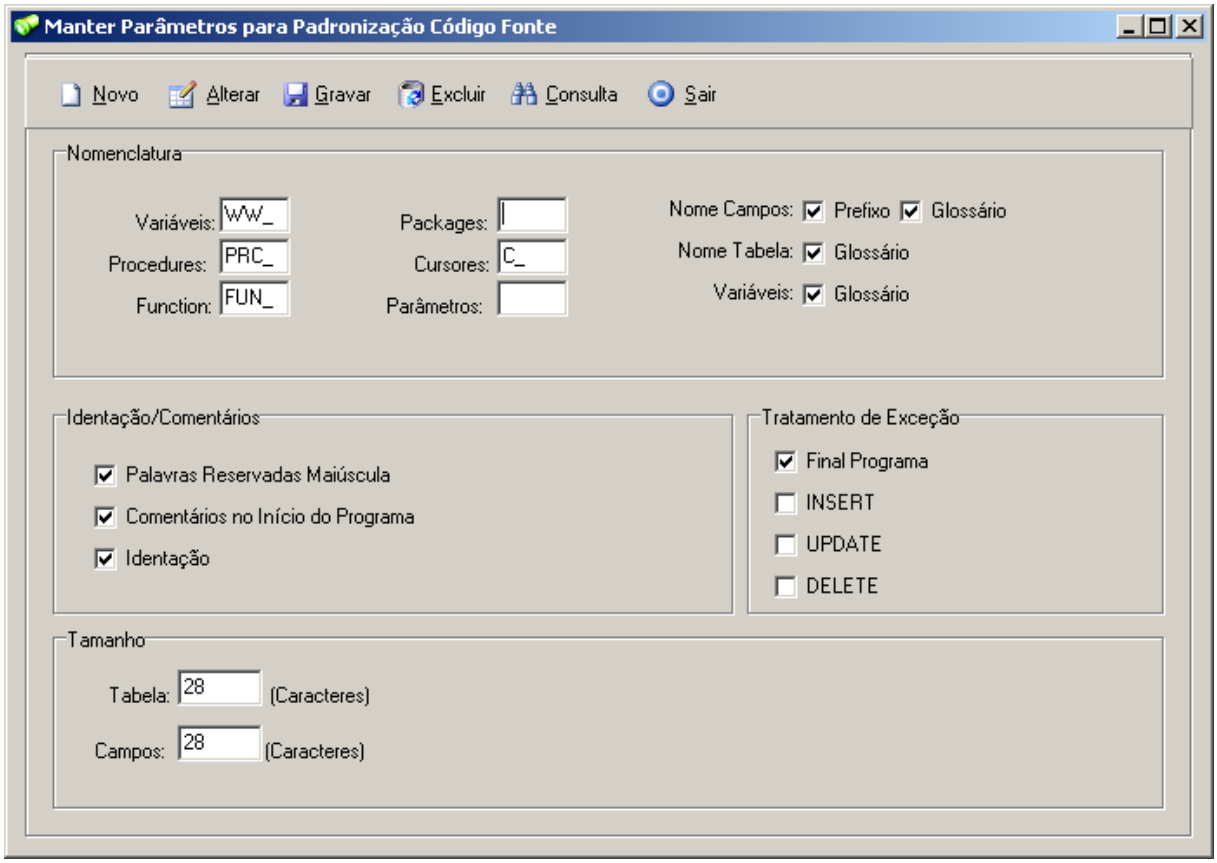


Figura 18 - Tela de parâmetros do relatório – exemplo 2

```
--ANTES
FUNCTION BUSCA_SEQUENCIA (P_CD_TABELA IN VARCHAR2) RETURN NUMBER AS

  w_cd_sequencia  NUMBER(10);

  --cursor busca sequencia no brim (27)
  CURSOR C001 IS
    SELECT NVL(MAX(INFINITYRECNO),2140000001) + 1
    FROM   BRIMJOURNALTRANS
    WHERE  INFINITYRECNO > 2140000000;

  --cursor busca sequencia no bim (23,24)
  CURSOR C002 IS
    SELECT NVL(MAX(INFINITYRECNO),2140000001) + 1
    FROM   BIMJOURNALTRANS
    WHERE  INFINITYRECNO > 2140000000;

-----
BEGIN
   IF (P_CD_TABELA = 'BRIM') THEN
      OPEN C001;
         FETCH C001 INTO W_CD_SEQUENCIA;
      CLOSE C001;
```

```
   ELSIF (P_CD_TABELA = 'BIM') THEN
      OPEN C002;
        FETCH C002 INTO W_CD_SEQUENCIA;
      CLOSE C002;
   END IF;
  RETURN(W_CD_SEQUENCIA);
END BUSCA_SEQUENCIA;

--DEPOIS
/*************************************************************
Autor....:
Empresa..:
Data.....: 15/05/2009
Descrição: Comentário gerado pelo TCC-II Giselle Mafra Schlosser
*************************************************************/
FUNCTION func_busca_sequencia (p_cd_tabela IN varchar2) RETURN number AS

  ww_cd_sequencia  number(10);

--cursor busca sequencia no brim (27)
  CURSOR c_c001 IS
    SELECT nvl(max(infinityrecno),2140000001) + 1
      FROM brimjournaltrans
     WHERE infinityrecno > 2140000000;

  --cursor busca sequencia no bim (23,24)
  CURSOR c_c002 IS
    SELECT nvl(max(infinityrecno),2140000001) + 1
      FROM bimjournaltrans
     WHERE infinityrecno > 2140000000;
-----
BEGIN
  IF (p_cd_tabela = 'BRIM') THEN
    OPEN c001;
      FETCH c001 INTO w_cd_sequencia;
        CLOSE c001;
  ELSIF (p_cd_tabela = 'BIM') THEN
    OPEN c002;
      FETCH c002 INTO w_cd_sequencia;
        CLOSE c002;
  END IF;

  RETURN(w_cd_sequencia);

EXCEPTION
```

```
  WHEN OTHERS THEN
    Raise_Application_Error(-2001,'Erro na rotina func_busca_sequencia.');
END busca_sequencia;
```

Quadro 42 – Exemplo 2

A figura 19 demonstra os parâmetros utilizados para a execução do exemplo do quadro 38.

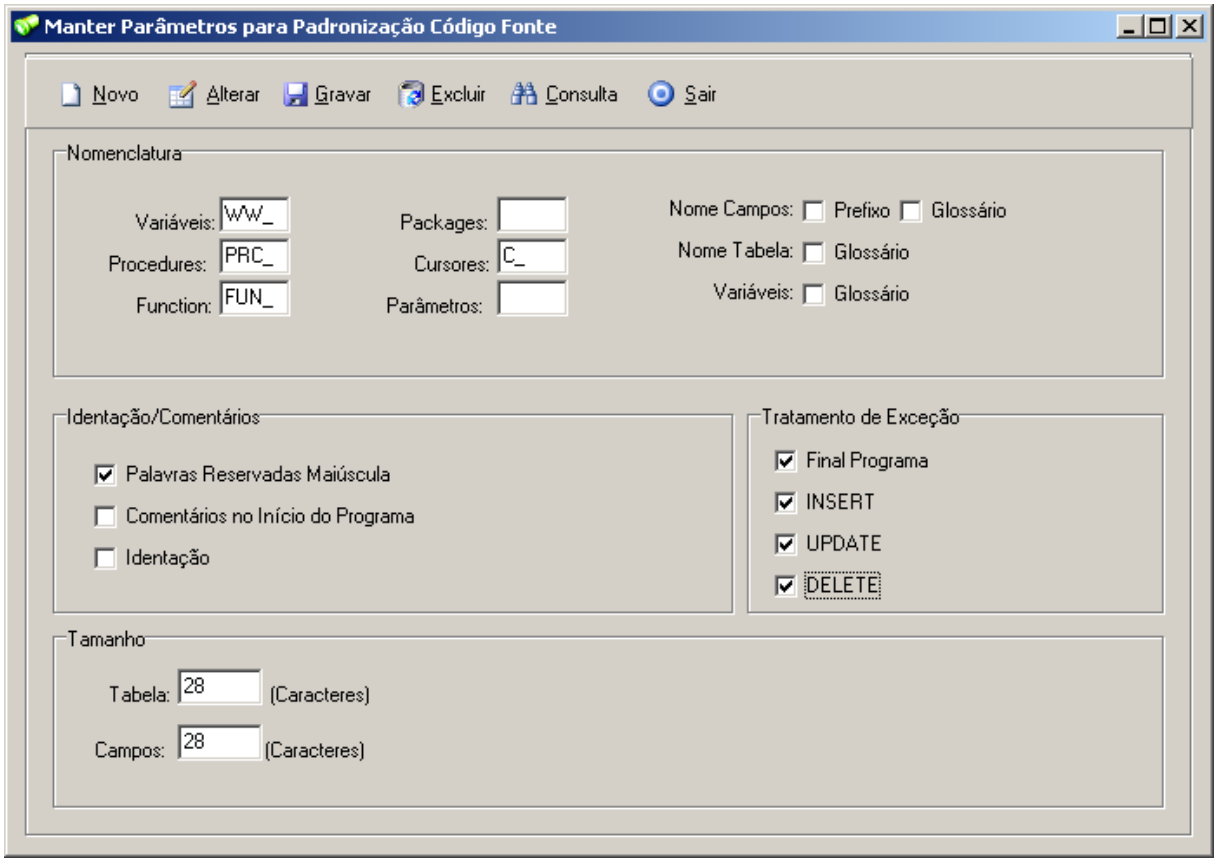


Figura 18 - Tela de parâmetros do relatório – exemplo 3

```
--ANTES

CREATE OR REPLACE PACKAGE package_teste IS
  PROCEDURE p_teste1;
  PROCEDURE p_teste2(x_codigo in number
                    ,x_descricao in varchar);
  PROCEDURE controle;
END package_teste;

CREATE OR REPLACE PACKAGE BODY package_teste IS
  PROCEDURE p_teste1 is
    wx_teste0 NUMBER;
    ww_teste2 varchar2(30);
    xx_teste3 varchar2(35);
    rr_teste  teste.cd_codigo%type;
    ww_teste4 EXCEPTION;

  CURSOR cur_teste is
   SELECT e.cd_teste,
          e.ds_teste_teste_ina
```

```
    from teste1 e,
         teste2 a
    where e.cd_teste = a.cd_teste;
  begin
    FOR reg_teste IN c_teste loop
      open c_teste;
        fetch c_teste
          into wx_teste0,
               xx_teste3;
      IF (ds_teste_teste_ina = 'MARIA'AND wx_teste0 IS NOT NULL)
      OR wx_teste0 = 20 THEN
        rr_teste := NULL;
      END IF;
    END LOOP;
     INSERT INTO TESTE
     (CD_TESTE, CD_TESTEQ, NOME, VALOR)
     VALUES(2,5,'TESTE',89);
     p_teste2(wx_teste0
              ,xx_teste3);
  END p_teste1;

  PROCEDURE p_teste2(x_codigo in number
                     ,x_descricao in varchar) IS
  BEGIN
     update teste21
        set nm_empresa = 'EMPRESA'
      where cd_empresa = 30;

      delete from teste5
       where cd_empresa = 50
         and cd_filial = 220;
  END p_teste2;

  PROCEDURE controle IS

    CURSOR cur_teste is
   SELECT e.cd_teste,
          e.ds_teste_teste_ina,
          e.vl_teste
    from teste1 e,
         teste2 a
    where e.cd_teste = a.cd_teste;
  BEGIN
    for reg_teste in cur_teste loop
      update teste7
         set cd_codigo = reg_teste.cd_teste
            ,ds_empresa = reg_teste.ds_teste_teste_ina
            ,vl_custo = vl_teste
       where cd_empresa = 150;
    end loop;

  END controle;
END package_teste;


--DEPOIS

CREATE OR REPLACE PACKAGE pck_package_teste IS

  PROCEDURE prc_teste1;

  PROCEDURE prc_teste2(p_codigo in number
                       ,p_descricao in varchar);

  PROCEDURE prc_controle;

END pck_package_teste;
```

```
CREATE OR REPLACE PACKAGE BODY pck_package_teste IS

  PROCEDURE prc_teste1 IS

    ww_teste0 number;
    ww_teste2 varchar2(30);
    ww_teste3 varchar2(35);
    ww_teste  teste.cd_codigo%type;
    ww_teste4 EXCEPTION;

  CURSOR c_teste is
   SELECT e.cd_teste,
          e.ds_teste_teste_ina
    FROM teste1 e,
         teste2 a
    WHERE e.cd_teste = a.cd_teste;

  BEGIN

    FOR reg_teste IN c_teste LOOP
      OPEN c_teste;
        FETCH c_teste
          INTO ww_teste0,
               ww_teste3;
      IF (ds_teste_teste_ina = 'MARIA'AND ww_teste0 IS NOT NULL)
      OR ww_teste0 = 20 THEN
        ww_teste := NULL;
      END IF;
    END LOOP;

    BEGIN
     INSERT INTO TESTE
     (CD_TESTE, CD_TESTEQ, NOME, VALOR)
     VALUES(2,5,'TESTE',89);
    EXCEPTION
      WHEN OTHERS THEN
        Raise_Application_Error(-2001,'Erro no insert da tabela TESTE.');
    END prc_teste1;

    p_teste2(ww_teste0
            ,ww_teste3);

  PROCEDURE prc_teste2(p_codigo in number
                      ,p_descricao in varchar) IS
  BEGIN
    UPDATE teste21
       SET nm_empresa = 'EMPRESA'
     WHERE cd_empresa = 30;
  WHEN OTHERS THEN
    Raise_Application_Error(-2002,'Erro no update da tabela teste.');
  END prc_teste2;

  BEGIN
    DELETE FROM teste5
     WHERE cd_empresa = 50
       AND cd_filial = 220;
  WHEN OTHERS THEN
    Raise_Application_Error(-2003,'Erro no delete da tabela teste5.');
  END prc_teste2;

  PROCEDURE prc_controle IS

  CURSOR c_teste IS
   SELECT e.cd_teste,
          e.ds_teste_teste_ina,
          e.vl_teste
    FROM teste1 e,
         teste2 a
```

```
    WHERE e.cd_teste = a.cd_teste;
  BEGIN
    FOR reg_teste IN c_teste LOOP
      UPDATE teste7
         SET cd_codigo = reg_teste.cd_teste
            ,ds_empresa = reg_teste.ds_teste_teste_ina
            ,vl_custo = vl_teste
       WHERE cd_empresa = 150;
    END LOOP;
  WHEN OTHERS THEN
    Raise_Application_Error(-2004,'Erro na rotina prc_controle.');
  END prc_controle;

WHEN OTHERS THEN
  Raise_Application_Error(-2005,'Erro na rotina pck_package_teste.');
END pck_package_teste;
```

Quadro 43 – Exemplo 3