UNIVERSIDADE REGIONAL DE BLUMENAU

CENTRO DE CIÊNCIAS EXATAS E NATURAIS

CURSO DE CIÊNCIAS DA COMPUTAÇÃO – BACHARELADO

MOTOR DE TEMPLATES PARA DELPHI 7

THIAGO KEWITZ DEMARCHI

BLUMENAU
2007

2007/2-33

THIAGO KEWITZ DEMARCHI

MOTOR DE TEMPLATES PARA DELPHI 7

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.

Profª. Joyce Martins, Mestre – Orientadora

BLUMENAU
2007

2007/2-33

MOTOR DE TEMPLATES PARA DELPHI 7

Por

THIAGO KEWITZ DEMARCHI

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

Presidente:	Profª. Joyce Martins, Mestre – Orientadora, FURB
Membro:	Prof. Maurício Capobianco Lopes, Mestre – FURB
Membro:	Prof. José Roque Voltolini da Silva – FURB

Blumenau, 26 de novembro de 2007

Dedico este trabalho a todos que de alguma forma me apoiaram durante a realização do mesmo, aos amigos, e a minha orientadora.

AGRADECIMENTOS

A Deus, por agraciar-me com serenidade na busca e conclusão deste objetivo.

À minha família, que diretamente ou indiretamente me guiou nesta caminhada.

À minha namorada, pela compreensão em relação a minha ausência para realização do trabalho, e pelo apoio e incentivo ao mesmo.

Aos meus amigos, também pelo apoio e incentivo para conclusão do trabalho.

À minha orientadora, Joyce Martins, por ter acreditado na conclusão deste trabalho, e por ter se dedicado para possibilitar isso.

RESUMO

No presente trabalho são descritas a especificação e a implementação de um motor de *templates* para Delphi, desenvolvido como uma biblioteca. A biblioteca analisa o *template*, reconhece o código estático e o código dinâmico, processa as diretivas do código dinâmico, substituindo-as por informações da aplicação e gera um texto formatado como saída. Para a análise da linguagem de *templates* são utilizados analisadores léxico, sintático e semântico. A biblioteca é aplicada em dois estudos de caso, sendo um responsável por gerar código fonte de classes em Java ou em Delphi a partir de informações pré-formatadas em arquivo texto, e outro por apresentar contatos de um catálogo de telefones, também pré-formatados em arquivo texto e permitir gerar textos formatados com base nestas informações.

Palavras-chave:*Template* . Motor de *templates*. Processador de linguagens. Delphi.

ABSTRACT

This work describes the specification and implementation of a template engine for Delphi, developed as a library. The library analyzes the template, recognizes the static and the dynamic code, processes the directives of the dynamic code and replaces it with application information and generates a formatted text as an output. For the analysis of the template language, lexical, syntactic and semantic analyzers are used. The library is applied in two case studies, one responsible for generating source code of classes in Java or Delphi from information on pre-formatted text files, and another for presenting contacts of a phone catalog, also pre-formatted on text files and allows generating formatted text on base in this information.

Key-words: Template. Template engine. Language processor. Delphi.

LISTA DE ILUSTRAÇÕES

LISTA DE SIGLAS

ASCII – *American Standard Code for Information Interchange*

BNF – *Backus Naur Form*

eNITL – *Network Improv Template Language*

GALS – Gerador de Analisadores Léxicos e Sintáticos

HTML – *HyperText Markup Language*

LL – *Left to right - Leftmost derivation*

LR – *Left to right - Rightmost derivation*

RF – Requisito Funcional

RNF – Requisito Não Funcional

RTTI – *RunTime Type Information*

PHP – *Hypertext PreProcessor*

SQL – *Structured Query Language*

UML – *Unified Modeling Language*

VTL – Velocity *Template Language*

SUMÁRIO

1 INTRODUÇÃO

O aumento da demanda por softwares[1] que sirvam as mais diversas áreas, acrescido da grande concorrência existente na indústria tecnológica, faz com que eles tenham de ser cada vez mais completos e construídos de forma a atender integralmente as necessidades do mercado. Neste sentido, os desenvolvedores devem considerar o quesito flexibilidade na concepção da aplicação, de maneira a permitir uma rápida adaptação às mudanças impostas pela dinâmica do mercado.

Uma forma de obter flexibilidade é fazendo uso de *templates* − arquivos que são dinamicamente transformados, servindo de base para a geração automática de código. Estes são formados por código estático − conteúdo que não sofre qualquer alteração no processo − e código dinâmico − conteúdo que é interpretado e dá lugar às informações obtidas da aplicação. O uso de *templates* tem diversas vantagens. Dentre elas, como afirma Parr (2004, p. 1), torna o desenvolvimento de aplicações mais fácil, aumenta a flexibilidade, reduz custos de manutenção e permite o desenvolvimento em paralelo da formatação do código e da lógica que determina o que deve ser gerado.

Os *templates* são manipulados através de motores de *templates*, que podem ser específicos, quando implementados diretamente como um módulo da aplicação, atendendo somente às necessidades da mesma; ou genéricos, quando são bibliotecas adaptáveis a qualquer aplicação e que permitem o tratamento de *templates* de forma simples. Existem vários motores de *templates*, dentre os quais pode-se citar o Velocity para Java (APACHE SOFTWARE FOUNDATION, 2004), o eNITL para C++ (BRECK, 1999) e o Smarty para PHP (FEITOSA, 2006). Existe ainda um encapsulamento do Smarty para Delphi: o FastTrac (HILL, 2004). Porém, esse motor de *templates* não apresenta um bom custo-benefício, pois possui limitações, como incompatibilidade com Delphi 7 e o núcleo de sua implementação em PHP, obrigando o uso de bibliotecas para seu funcionamento nas aplicações.

Desta forma, para aplicações desenvolvidas em Delphi os programadores são obrigados a implementar um motor de *templates* específico, conforme pode ser observado na ferramenta descrita por Orsi (2006). Diante do exposto, este trabalho propõe a criação de um motor de *templates* genérico para aplicações escritas em Delphi 7.

---

[1] Software e aplicação são utilizados como sinônimos no decorrer deste trabalho.

1.1 OBJETIVOS DO TRABALHO

O objetivo deste trabalho é desenvolver um motor de *templates* genérico para softwares desenvolvidos em Delphi 7.

Os objetivos específicos do trabalho são:

a) disponibilizar uma biblioteca com classes, atributos e métodos que implementem o motor de *templates*;
b) permitir mapear os dados da aplicação para serem referenciados nos *templates*;
c) analisar os *templates* léxica e sintaticamente, reconhecendo o código estático e o código dinâmico, bem como seus elementos;
d) interpretar e substituir o código dinâmico do *template* conforme os dados mapeados.

1.2 ESTRUTURA DO TRABALHO

O presente trabalho está estruturado em quatro capítulos. O segundo capítulo apresenta a fundamentação teórica necessária para o desenvolvimento do trabalho. Nele são discutidos *templates*, motores de *templates* e suas aplicações, além dos processadores de linguagens de programação, evidenciando os analisadores léxico, sintático e semântico. O capítulo também mostra características de alguns trabalhos correlatos.

No terceiro capítulo é apresentado o desenvolvimento do motor de *templates*, partindo de uma visão geral da biblioteca, seguida da apresentação dos requisitos do problema e de sua especificação. A especificação da biblioteca compreende a especificação da linguagem de *templates* e da biblioteca em si, sendo que para a primeira são descritas as especificações léxica, sintática e semântica e para a segunda são mostrados os diagramas de casos de uso, de seqüência e de classes. Este capítulo também discute a implementação da biblioteca, bem como os estudos de casos desenvolvidos e os resultados obtidos.

Finalizando, no quarto capítulo são apresentadas as conclusões e a lista de tarefas para continuidade do motor de *templates*.

2 FUNDAMENTAÇÃO TEÓRICA

Neste capítulo são apresentados alguns aspectos teóricos relacionados ao trabalho. É apresentado o conceito de *templates*, bem como descritas algumas de suas aplicações, focando nos geradores de código. São relacionadas características de motores de *templates*, com breve histórico de sua origem, sendo também citados o uso e os recursos das linguagens de *templates*. Na continuação são expostos aspectos teóricos sobre processadores de linguagem e seus componentes: os analisadores léxico, sintático e semântico. Na última seção são examinadas características dos trabalhos correlatos que serviram de base para o desenvolvimento da biblioteca proposta.

2.1 *TEMPLATES* E SUAS APLICAÇÕES

Templates são arquivos que servem de base para a geração de textos formatados. Seu uso em aplicações permite trabalhar em separado a lógica e a formatação de saída, possibilitando que, em vários casos, o formato do texto de saída possa ser alterado com a simples modificação do *template*, sem a necessidade de alteração do código fonte da aplicação (PARR, 2004, p. 1).

Segundo Orsi (2006, p. 26), um *template* é formado por código estático e código dinâmico. O código estático é transcrito para o texto gerado de forma literal, não sofrendo alterações. O código dinâmico é tratado e nele são substituídos trechos por dados da aplicação, para então integrar o texto formatado de saída.

Templates podem ser utilizados em geradores de código – ferramentas que auxiliam na construção de softwares pois permitem, a partir de alguma entrada, gerar código fonte como saída. Segundo Herrington (2003, p. 3, tradução nossa), “geração de código trata de escrever programas que escrevem programas.” Podem ser usados em aplicações para diversas finalidades, como por exemplo: geração de documentação relativa a código fonte; geração de código com comandos SQL a partir de instruções no código fonte; geração de interfaces com o usuário; geração de classes de interfaceamento entre camadas; entre outros. Herrington (2003, p. 15-17) cita diversos benefícios associados ao uso de geradores de código, dentre os quais destacam-se: qualidade na geração de grande volume de código, consistência e padrão

do código gerado, reutilização de código e facilidade de manutenção.

Apesar da classificação existente que diferencia os geradores de código, seus aspectos são comuns em relação às etapas de desenvolvimento. A implementação de um gerador de código, segundo Herrington (2003, p. 61-94), inicia pela especificação dos textos de saída, seguida pela definição do funcionamento do gerador, delimitando suas entradas. Desenvolver a análise dos dados de entrada e os *templates* com base no texto de saída a ser gerado são os passos posteriores. Por fim, deve-se desenvolver a ferramenta em si, a qual deve buscar os dados das entradas analisadas e alimentar um motor de *templates* que gerará o texto de saída com base nos *templates* informados.

Os geradores de código, além da flexibilização do formato da saída, permitem a geração de diversos textos usando a mesma padronização definida no *template*. Também garantem, neste caso, que uma alteração global possa ser resolvida com mudanças somente no *template*, ao invés de ser feita manualmente em todos os locais (HERRINGTON, 2003, p. 7).

2.2 MOTORES DE *TEMPLATES*

Os motores de *templates*, do inglês *template engines*, podem ser utilizados, segundo Rocha (2005), em aplicações de geração de código, como por exemplo em ferramentas de documentação de código fonte ou para a construção de páginas para a internet. Parr (2004) descreve que, originalmente, as páginas eram desenvolvidas de forma tediosa e com grande ocorrência de erros, não possibilitando serem escritas em editores gráficos. O Quadro 1 ilustra o código para geração usual de uma página HTML.

```
out.println("<html>");
out.println("<body>");
out.println("<h1>Teste</h1>");
String name = request.getParameter("name");
out.println("Olá, " + name + ".");
out.println("</body>");
out.println("</html>");
```

Fonte: adaptado de Parr (2004).

Quadro 1 – Exemplo de geração usual de página HTML

A partir da necessidade de geração de páginas para internet de forma automática, com flexibilidade e baixo custo de manutenção, a codificação da página foi embutida no código HTML (Quadro 2). Foi então que, num terceiro momento, os motores de *templates* surgiram como uma solução mais coerente neste sentido. O Quadro 3 ilustra a geração de página HTML utilizando *template*. O uso deste oferece diversas vantagens: evita o uso de código na

parte de visualização, possibilita um melhor tratamento de erros, não requer o uso de um compilador, dentre outras (HUNTER, 2000).

```
<html>
<body>
<h1>Teste</h1>
Olá, <%= request.getParameter("name")%>.
</body>
</html>
```

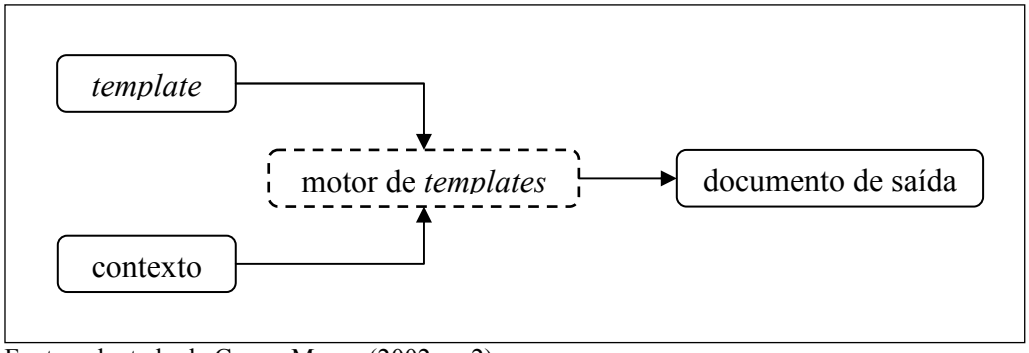
Fonte: adaptado de Parr (2004).

Quadro 2 – Exemplo de geração de página com código embutido no HTML

```
<html>
<body>
<h1>Teste</h1>
Olá, $name.
</body>
</html>
```

Quadro 3 – Exemplo de geração de página HTML utilizando *template*

Os motores de *template* são ferramentas que geram textos formatados de saída com base em *templates*. Os dados trabalhados com o *template*para a geração da saída vêm da aplicação e são informadas através de um contexto[2]. O funcionamento dos motores de *templates* é ilustrado na Figura 1.



Fonte: adaptado de Cruz e Moura (2002, p. 2).

Figura 1 – Funcionamento do motor de *templates*

Os motores de *templates* conseguem interpretar os *templates*, pois estes devem estar formados conforme a linguagem de *templates* definida por eles. Segundo Rocha (2005), as linguagens de *templates* definem os tipos de blocos especiais e como referenciar variáveis nos *templates*. Elas apresentam complexidades diferentes, dependendo do motor de *templates*, podendo prover, além da substituição de valores – através de estruturas básicas, estruturas de controle mais complexas como laços e comandos condicionais.

[2] Contexto é a estrutura que mapeia os dados da aplicação para serem disponibilizados ao motor de *templates*.

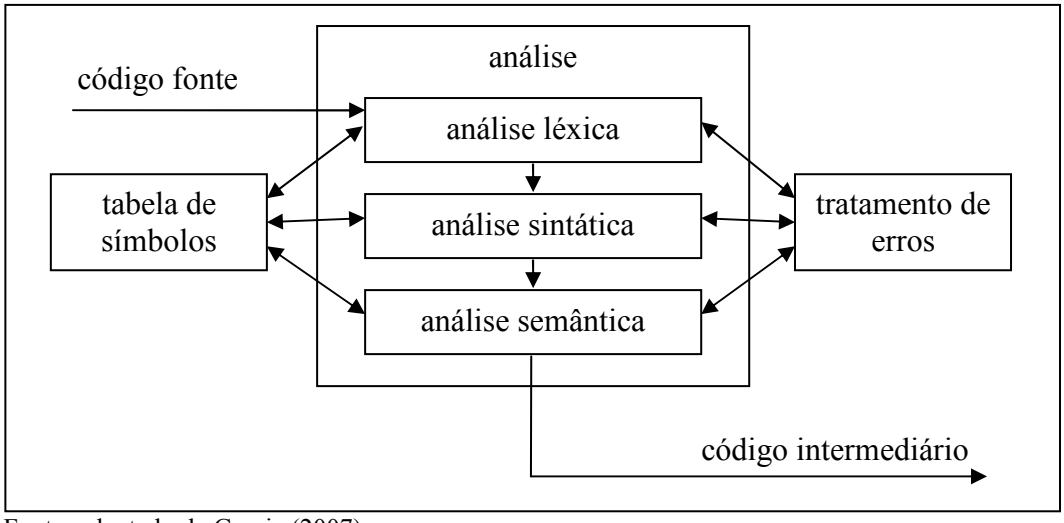
2.3 PROCESSADORES DE LINGUAGEM

A linguagem de *templates* é reconhecida e interpretada pelo motor de *templates* através de processadores de linguagem. Processador de linguagem é uma rotina responsável por ler entradas de texto fonte, validar e reconhecer o significado de suas construções – trabalho realizado pelo módulo de análise – e executar ações em função destes significados – trabalho realizado pelo módulo de síntese – com base em uma linguagem definida (BARRICO, 2002, p. 2). Em função das características dos motores de *templates*, neste trabalho apenas o módulo de análise é descrito.

O módulo de análise divide seu processo em três fases:

a) análise léxica: leitura dos caracteres da entrada e produção de uma seqüência de símbolos terminais (*tokens*) (AHO; SETHI; ULLMAN, 1995, p. 38);
b) análise sintática: verificação da seqüência dos símbolos terminais considerando a gramática da linguagem (AHO; SETHI; ULLMAN, 1995, p. 72);
c) análise semântica: verificação das regras semânticas da linguagem e reconhecimento do significado das construções sintáticas (BARRICO, 2002, p. 3).

A arquitetura básica do módulo de análise de um processador de linguagem é representada na Figura 2.



Fonte: adaptado de Garcia (2007).

Figura 2 – Arquitetura básica de um processador de linguagem

Segundo Barrico (2002), são exemplos de processadores de linguagens:

a) interpretadores: executam os programas apresentando resultados, sem traduzir os programas para outra linguagem;
b) compiladores: traduzem linguagens de programação de alto nível para baixo nível;

c) *assemblers*: traduzem linguagens de programação de baixo nível para código de máquina;
d) tradutores em geral: transformam textos escritos numa linguagem para outra linguagem;
e) pesquisadores: reconhecem questões e pesquisam em bases dedados , mostrando as respostas encontradas;
f) filtros: reproduzem o texto que receberam, retirando, expandindo ou transformando palavras ou blocos;
g) processadores de documentos: manipulamdiversos tipos de documentos .

2.3.1 Análise léxica

A análise léxica é a primeira fase do processamento de linguagens. Nela o arquivo de entrada é fragmentado em componentes básicos, os *tokens* (GARCIA, 2007). Segundo Barreto (2005), o processo ocorre varrendo o arquivo de entrada, eliminando comentários e caracteres de formatação, tornando o trabalho da análise sintática mais simples e dando eficiência às fases subseqüentes.

O analisador léxico, ou *scanner*, é um programa que implementa um autômato finito para o reconhecimento de cadeias de caracteres como *tokens* (RICARTE, 2003). Ricarte (2003) ainda descreve o algoritmo que determina se uma cadeia de caracteres pertence à linguagem definida por um autômato finito.

> A sentença a ser reconhecida é estruturada como uma lista de símbolos, que é passada como argumento para o analisador léxico juntamente com a referência para o autômato. O analisador léxico inicia sua operação definindo o estado inicial como o estado corrente. Obtém então o próximo símbolo (que inicialmente é o primeiro símbolo) da sentença σ. Se não houver próximo símbolo, é preciso verificar se o estado corrente é um estado final. Se for, o procedimento retorna *true*, indicando que a sentença foi reconhecida pelo autômato. Se o estado corrente não for um estado final e não houver mais símbolos na sentença, então não houve reconhecimento e o procedimento retorna *false*. Se houver símbolo a ser analisado, então o procedimento deve continuar o processo de reconhecimento. Para tanto, obtém o próximo estado correspondente à transição do estado atual pelo símbolo sob análise. Se não houver transição possível, então a sentença não foi reconhecida e o procedimento deve encerrar, retornando *false*. (RICARTE, 2003).

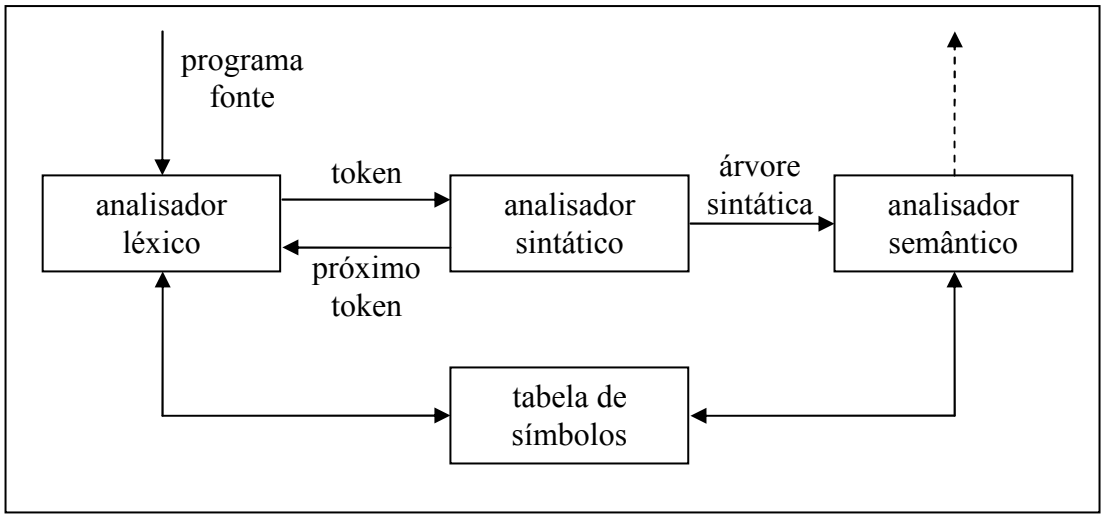
A análise léxica também é responsável pela construção da tabela de símbolos, uma estrutura de dados que contém informações sobre os nomes declarados no arquivo de entrada (PRICE; TOSCANI, 2000, p. 26).

Nesta fase são identificados poucos erros, porém, na detecção posterior de erros sintáticos ou semânticos, será possível encontrar a posição do erro através do *token* onde ele foi detectado (AHO; SETHI; ULLMAN, 1995, p. 40).

2.3.2 Análise sintática

A análise sintática é a segunda fase do processamento de linguagens, sendo responsável pela leitura de seqüências de *tokens*, reconhecidos e classificados anteriormente, verificando se tal seqüência pode ser gerada pela gramática da linguagem (GARCIA, 2007).

O programa que realiza esta função chama-se analisador sintático, ou *parser*, o qual recebe como entrada a seqüência de *tokens* reconhecida pelo analisador léxico e constrói uma árvore sintática para ser usada nas fases seguintes (RICARTE, 2003). Segundo Aho, Sethi e Ullman (1995, p. 39), comumente o analisador sintático implementa o analisador léxico como uma sub-rotina, sendo que a cada necessidade de um novo *token* é feita uma chamada ao analisador léxico (Figura 3).



Fonte: adaptado de Aho, Sethi e Ullman (1995, p. 72).

Figura 3 – Interação entre as fases do módulo de análise

Os analisadores sintáticos são classificados em três tipos (AHO; SETHI; ULLMAN, 1995, p. 72):

a) métodos universais de análise sintática: tratam qualquer tipo de gramática, porém são muito ineficientes;
b) *top-down*: constroem árvores sintáticas da raiz (topo) para as folhas (fundo), varrendo os tokens da esquerda para a direita, produzindo derivações mais à

esquerda. É eficientemente aplicável em subclasses gramaticais LL e são convenientes para implementação manual (GARCIA, 2007);

c) *bottom-up*: constroem árvores sintáticas das folhas (fundo) para a raiz (topo), varrendo os *tokens* da esquerda para a direita, produzindo derivações mais à direita. É eficientemente aplicável em subclasses gramaticais LR e são convenientes para construção automática (GARCIA, 2007).

Estudos mostram que a análise sintática é responsável por encontrar a maioria dos erros durante o processamento das linguagens. Para tanto, as tarefas de detecção e recuperação de erros devem ser realizadas pelo analisador sintático. Todavia, erros semânticos e lógicos dificilmente são detectados nesta etapa (GARCIA, 2007).

2.3.3 Análise semântica

A análise semântica é a terceira fase do processamento de linguagens e a última do módulo de análise. “Embora a análise sintática consiga verificar se uma expressão obedece às regras de formação de uma dada gramática, seria muito difícil expressar através de gramáticas algumas regras usuais em linguagem de programação [...]” (RICARTE, 2003). Ainda segundo Ricarte (2003), o objetivo desta fase é validar o relacionamento entre partes distintas do programa, sendo algumas das tarefas básicas a verificação de tipos, do fluxo de controle e da unicidade da declaração de variáveis, além de outros tipos de verificação definidas pela linguagem.

A análise semântica pode ser implementada através de esquemas de tradução. Os esquemas de tradução são extensões de uma gramática onde atributos – podendo conter um valor numérico, uma cadeia de caracteres, um tipo de dado ou um endereço de memória, entre outras possibilidades – são associados aos símbolos gramaticais e ações semânticas – sendo avaliações de atributos ou chamadas a procedimentos e funções – às regras de produção. A execução de ações semânticas possui uma ordem determinada conforme sua dependência, já que computam valores de atributos a partir de outros atributos (PRICE; TOSCANI, 2000, p. 86).

2.4 TRABALHOS CORRELATOS

Existem diversas bibliotecas que desempenham papel semelhante ao proposto por este trabalho, sendo que a seguir são apresentados o Smarty, o FastTrac e o Velocity. Esta última possui um estudo mais aprofundado, por ser tomada como base para o desenvolvimento da biblioteca proposta.

2.4.1 Smarty

Segundo Feitosa (2006), o Smarty é uma classe de *templates* que separa a interface da lógica de programação, objetivando facilitar e melhorar o desenvolvimento de aplicações em PHP. O Smarty foi construído para ser rápido e simples, permitindo grande independência entre o trabalho dos *designers* e dos programadores da lógica da aplicação.

O Quadro 4 apresenta um exemplo de *template* interpretável pelo Smarty e o Quadro 5 o exemplo de saída gerada com este.

```
Nome                                Número de telefone
------------------------------------------------------------------------------
<--strip-->
<--section name=sec loop=$People-->
       <--section name=phones loop=$People[sec].Phones -->
       <--if $smarty.section.phones.index > 0 -->
              <--assign var=TempString value=$People[sec].Phones[phones].Type|cat:"
"|cat:$People[sec].Phones[phones].Number|indent:30-->
       <--else-->
              <--assign var=TempString value=$People[sec].Phones[phones].Type|cat:"
"|cat:$People[sec].Phones[phones].Number|string_format:"%-30.30s"-->
       <--/if-->
       <--assign var=PhoneString value=$PhoneString|cat:$TempString|cat:"\n"-->
       <--/section-->
<--$People[sec].FirstName|string_format:"%s
"|cat:$People[sec].LastName|string_format:"%'.-30.30s"--><--$PhoneString-->
<--assign var=PhoneString value=""-->
<--/section-->
<--/strip-->
```

Fonte: adaptado de Hill (2004).

Quadro 4 – Exemplo de *template* para o Smarty

```
Nome                                 Número de telefone
--------------------------------------------------------------------------------
João da Silva.................Res (47) 3348-1852
                              Cel (47) 9923-3476
Fernando Barros...............Res (47) 2376-3234
                              Com (47) 3238-9084
Alexandre Costa...............Res (47) 7643-4573
Ana Francisco.................Res (47) 6553-8356
```

Fonte: adaptado de Hill (2004).

Quadro 5 – Exemplo de saída gerada com o Smarty

2.4.2 FastTrac

Segundo Hill (2004), o FastTrac é um encapsulamento para Delphi 6 da biblioteca Smarty. O encapsulamento – programado em Delphi – implementa estruturas de dados para definição do contexto e um interfaceamento com o Smarty, que é responsável pelo processamento dos *templates*, bem como a substituição do código dinâmico pelos dados do contexto.

Um *template* interpretável pelo FastTrac também o é pelo Smarty, pois ambos possuem linguagens de *templates* idênticas. Desta forma, considerando o mesmo contexto, um *template* submetido às duas bibliotecas gera saídas iguais. Contudo, a biblioteca apresenta as seguintes características:

a) incompatibilidade com Delphi 7: para utilizar a biblioteca é requerido dos programadores sua compilação e instalação como um componente do Delphi 7;
b) distribuição de bibliotecas: sendo seu núcleo o Smarty, o qual é programado em PHP, várias bibliotecas devem ser disponibilizadas tanto no ambiente do programador como nas máquinas onde deve rodar a aplicação. Tais bibliotecas são: o Smarty em PHP, a biblioteca de interpretação de PHP, uma biblioteca de interface entre Delphi e PHP.

2.4.3 Velocity

Velocity é um motor de *templates* baseado em Java, de código aberto e desenvolvido pelo projeto Jakarta, que pode ser utilizado sozinho, para a geração de código fonte, de

páginas HTML e de relatórios, ou pode ser combinado com outras aplicações, provendo a possibilidade de utilização de *templates* (APACHE SOFTWARE FOUNDATION, 2004).

A linguagem de *templates* utilizada pelo Velocity é denominada VTL. Cruz e Moura (2002, p. 3) afirmam que a VTL é simples e fácil de assimilar, pois possui poucos recursos, dentre os quais, destacam-se: a manipulação de variáveis e diretivas de controle de fluxo e o reuso de código. O Quadro 6 apresenta um exemplo de *template* escrito em VTL e o Quadro 7 mostra uma possível saída gerada pelo Velocity utilizando este *template*. Já o Quadro 8 e o Quadro 9 resumem os recursos existentes na VTL do Velocity.

```
Lista de compras

Existem \$list.size() itens:

#foreach (\$item in \$list)
* \$item
#end
```

Fonte: adaptado de Steil (2006).

Quadro 6 – Exemplo de *template* para o Velocity

```
Lista de compras

Existem 6 itens:

* Pão
* Carne
* Feijão
* Arroz
* Açúcar
* Farinha
```

Fonte: adaptado de Steil (2006).

Quadro 7 – Exemplo de saída gerada com o Velocity

ELEMENTO	ESPECIFICAÇÃO[3]	EXEMPLO

identificador (*id*)	`[a-z A-Z][a-z A-Z 0-9 - _ ]*`	`abc`
		`a123`
		`a_123`
		`a1-b3`

declaração de variável	**$** (**!**)? (*id* | **{***id***}**)	`$abc`
		`${a123}`
		`$!a_123`
		`$!{a1-b3}`

propriedade	**$** (*id.id* | **{***id.id***}**)	`$pessoa.nome` `${pessoa.nome}`

método	**$** (*id.id* **(**(*lista de parâmetros*)?**)** | **{***id.id* **(**(*lista de parâmetros*)?**)}**)
	
	onde:
	lista de parâmetros é uma lista de valores (constantes ou referências) para os parâmetros do método
		`$pessoa.getNome()`
		`${pessoa.getTelefone("CEL")}`
		`$vendas.getEntreValores(10,$max)`

comentários	comentário de linha:
	## (*comentário*)?
	
	comentário de bloco:
	#* (*comentário*)?
	***#**
	
	onde:
	comentário é o texto do comentário
		`## comentário em uma linha`
		
		`#*`
		`  comentário em várias linhas`
		`*#`

Fonte: adaptado de Apache Software Foundation (2004) e de Cruz e Moura (2002, p. 4).

Quadro 8 – Elementos da linguagem VTL (primeira parte)

[3] A notação da especificação utiliza o símbolo ? para indicar que um conjunto de caracteres é opcional, o símbolo + para indicar que um conjunto de caracteres pode se repetir uma ou mais vezes, o símbolo * para indicar que um conjunto de caracteres pode se repetir zero ou mais vezes e o símbolo | para indicar escolha entre os conjuntos de caracteres.

ELEMENTO	ESPECIFICAÇÃO	EXEMPLO
atribuição	`#set ($ref = (arg | "arg" | 'arg'))` onde: *ref* é uma referência para variável ou propriedade *arg* é o valor que será atribuído	`#set ($valor = "true")`

controle de fluxo de execução	comando condicional:
	`#if (condição) (comandos)?`
	`( #elseif (condição) (comandos)?)*`
	`( #else (comandos)?)?`
	`#end`
	
	comando de repetição:
	`#foreach ($ref in arg) (comandos)?`
	`#end`
	
	comando de parada de análise:
	`#stop`
	
	onde:
	condição é uma expressão lógica
	comandos é um bloco de comandos
	ref é uma referência para variável
	arg é uma referência para lista
		`#if ($valor == 1) ...`
		`#elseif ($valor == 2) ...`
		`#elseif ($valor == 3) ...`
		`#else ...`
		`#end`
		
		`#foreach ($item in $lista)`
		`  ...`
		`#end`
		
		`#stop`

inclusão de arquivos	inclusão de texto de arquivos:
	`#include (arg (, arg)*)`
	
	inclusão de resultado da análise de *template*:
	`#parse (arg)`
	
	onde:
	arg é uma referência para um arquivo válido
		`#include("texto.txt")`
		
		`#parse("templAux.vm")`

reuso de código	definição de macro:
	`#macro (nome ($arg)+) (comandos)?`
	`#end`
	
	invocação de macro:
	`#nome (($ref )+)`
	
	onde:
	nome é o nome da macro
	arg é o argumento
	comandos são os comandos da macro
	ref é uma referência para parâmetro
		`#macro(mcTeste $par1 $par2)`
		`  ...`
		`#end`
		
		`#mcTeste(1 "teste")`

Fonte: adaptado de Apache Software Foundation (2004) e de Cruz e Moura (2002, p. 4).

Quadro 9 – Elementos da linguagem VTL (segunda parte)

O funcionamento do Velocity dá-se conforme a seguir:

a) os dados da aplicação são encapsulados em classes Java, acessíveis através de sua interface pública e mapeados através de apelidos em um contexto;

b) o Velocity lê o *template* escrito em VTL, reconhecendo o código estático e o código dinâmico;

c) o Velocity interpreta o código dinâmico, fazendo as devidas substituições com os dados do contexto;
d) o Velocity junta o código estático e o código dinâmico, gerando assim o texto formatado de saída.

3 DESENVOLVIMENTO DA BIBLIOTECA

O desenvolvimento do motor de *templates* envolveu as seguintes fases: especificação da linguagem de *templates*, levantamento de requisitos, especificação da biblioteca, implementação da biblioteca e implementação de estudos de caso. Neste capítulo, além do detalhamento das fases acima, são apresentados os resultados obtidos.

3.1 ESPECIFICAÇÃO DA LINGUAGEM DE *TEMPLATES*

A especificação da linguagem de *templates* foi feita através de uma BNF, escrita e validada na ferramenta GALS (GESSER, 2002). Para tal foram estudadas as gramáticas do motor de *templates* implementado por Orsi (2006) e da VTL do Velocity versão 1.4. Da primeira nada foi possível aproveitar, pois trata-se de uma especificação muito direcionada às necessidades da ferramenta para a qual foi originada. A segunda, por sua vez, serviu de base para especificação da linguagem de *templates*.

A especificação da VTL, disponível com a documentação do Velocity, contém alguns erros em relação ao seu real funcionamento, como pode ser visto no Quadro 10, onde a especificação da diretiva `#if` não conta com a palavra reservada `#end` para sua finalização, e o não-terminal `<else-statement>` não é especificado. Além de estudar a gramática da VTL, para especificação da linguagem de *templates*, foi estudado o guia de usuário do Velocity e diversos testes foram feitos com o intuito de conhecer o seu funcionamento.

```
<if-statement>
    ::= "#if" "(" <expresion> ")" <statement> [ <else-statement> ]
<else-if-statement>
    ::= "#elseif" "(" <expresion> ")" <statement> [ <else-statement> ]
```

Fonte: Apache Software Foundation (2004).

Quadro 10 – Problemas na especificação da VTL do Velocity

A linguagem de *templates* desenvolvida conta com diversos recursos dentre os quais se destacam a declaração e uso de variáveis, o controle de fluxo de execução e o reuso de código (Quadro 11).

RECURSO	ESPECIFICAÇÃO	EXEMPLO
declaração e atribuição de variáveis	**#set (**$*var* **=** *exp***)** onde: *var* é o nome da variável (existente ou não) *exp* é o valor da atribuição (expressão)	**#set** ($valor = "true") **#set** ($valor = true) **#set** ($valor = 1 + 1)
referência	**$***var* ((. *propriedade*) | (. *método* **(**(*val*)***) **))* onde: *var* é o nome da variável *propriedade* é o nome da propriedade *método* é o nome do método *val* é o valor do parâmetro do método	$nome $pessoa.nome $pessoa.primeiroNome() $pessoa.numFones("+55") $pessoa.fone(1).ramal
bloco de comandos	**#begin** (*comandos*)? **#end** onde: *comandos* é um conjunto de comandos	**#begin ... #end**
controle de fluxo – condicional	**#if (***condição***)** (*comandos*)? (**#elseif (***condição***)** (*comandos*)?)* (**#else** (*comandos*)?)? **#end** onde: *condição* é uma expressão lógica *comandos* é um conjunto de comandos	**#if** ($valor == 1) ... **#elseif** ($valor == 2) ... **#elseif** ($valor == 3) ... **#else** ... **#end**
controle de fluxo – repetição	**#foreach ($***var* **in** *val***)** (*comandos*)? **#end** onde: *var* é o nome da variável *val* é uma referência para lista *comandos* é um conjunto de comandos	**#foreach** ($item **in** $lista) ... **#end**
controle de fluxo – parada	**#stop**	**#stop**
inclusão de arquivos – texto	**#include (***arg* (, *arg*)***)** onde: *arg* é uma referência para um arquivo válido	**#include**("texto.txt")
inclusão de arquivos – *template*	**#parse (***arg***)** onde: *arg* é uma referência para um arquivo válido	**#parse**("templAux.vm")
reuso de código – definição de macro	**#macro (***nome* (**$***arg***)***) (*comandos*)? **#end** onde: *nome* é o nome da macro *arg* é o argumento *comandos* é um conjunto de comandos	**#macro**(mcTeste $arg1 $arg2) ... **#end**
reuso de código – invocação de macro	**#***nome* **(**(*val*)***)** onde: *nome* é o nome da macro *val* é o valor do parâmetro	#mcTeste(1 "teste")
comentário – linha	**##** (*comentário*)? onde: *comentário* é o texto do comentário	## comentário em linha
comentário – bloco	**#*** (*comentário*)? ***#** onde: *comentário* é o texto do comentário	**#*** comentário em várias linhas ***#**

Quadro 11 – Recursos da linguagem de *templates*

No Quadro 12 é apresentado um trecho de um *template* utilizando alguns recursos da

linguagem. A especificação da linguagem de *templates* pode ser observada no Apêndice A.

```
#foreach ($objeto in $lista)
  #set ($forma = "")
  #if ($objeto.ehQuadrado)
    #set ($forma = "quadrado")
  #elseif ($objeto.ehRedondo)
    #set ($forma = "redondo")
  #else
    #set ($forma = "outra")
  #end
  Objeto: $objeto.nome
  Forma: $forma
#end
```

Quadro 12 – Exemplo de *template*

3.1.1 Especificação léxica

Durante a análise léxica da linguagem de *templates* são reconhecidos como *tokens* desde estruturas mais simples, como diretivas e identificadores, até estruturas mais complexas, como comentários (Quadro 13). As diretivas iniciam com `#` seguido de um ou mais caracteres válidos – letras minúsculas ou maiúsculas, números, `-` ou `_` –, podendo ser reconhecidas a partir de um dos seus onze casos especiais ou de qualquer outro conjunto de caracteres que obedeça a formação descrita anteriormente. Os identificadores, por sua vez, são formados por qualquer seqüência de uma letra, minúscula ou maiúscula, e zero ou mais caracteres válidos. Os comentários de linha iniciam por dois `#`, seguidos de qualquer caractere ou símbolo, exceto quebra de linha. Na definição do comentário de bloco são utilizados recursos semelhantes aos usados no comentário de linha, apesar de possuírem formações diferentes.

```
// Definições regulares auxiliares
...
carac_extras: \127 | \128 | \129 | \130 | \131 | \132 | \133 |
\134 | \135 | \136 | \137 | \138 | \139 | \140 | \141 | \142 |
\143 | \144 | \145 | \146 | \147 | \148 | \149 | \150 | \151 |
\152 | \153 | \154 | \155 | \156 | \157 | \158 | \159 | \160
letra: [a-z A-Z]
numero: [0-9]
caractere: {letra} | {numero} | "-" | "_"
...
tudo1: [^\n\r]   | {carac_extras}
tudo2: [^"*"]    | {carac_extras}
tudo3: [^"*""#"] | {carac_extras}
...
coment_linha: "#" "#" {tudo1}*
coment_bloco: "#*" ({tudo2}  | ("*")+ {tudo3})* ("*")+ "#"
...

// Tokens
diretiva: "#" {caractere}+
identificador: {letra} {caractere}*
...
comentario: {coment_linha} | {coment_bloco}
...
// Diretivas
begin   = diretiva: "#begin"
...
// Palavras reservadas
true    = identificador: "true"
...
```

Quadro 13 – Trecho da especificação léxica da linguagem de *templates*

3.1.2 Especificação sintática

A gramática especificada para a linguagem de *templates* possui algumas construções complexas. Um *template* é composto por zero ou mais instruções, que por usa vez é composta por texto estático, comentários, referências e diversas diretivas (Quadro 14). Uma das dificuldades enfrentadas ao especificar sintaticamente a linguagem é a redundância causada por construções que podem ser tanto texto estático como texto dinâmico. Por exemplo, o `$` sozinho é considerado um texto estático, porém acompanhado de um identificador torna-se uma referência. O mesmo ocorre ao reconhecer um caso genérico de diretiva, onde sozinha é um texto estático, porém acompanhada de um `(` é considerada uma chamada de macro.

```
<template>                    // Corpo do template
 ::= <instrucoes>
 ;

<instrucoes>                  // Conjunto de instruções
 ::= <instrucoes_x>
 ;

<instrucoes_x>          // Conjunto de instruções (fatoração)
 ::= <instrucao> <instrucoes>
 |   ε
 ;

<instrucao>                   // Instrução
 ::= <estatico>
 |   comentario
 |   "$" <instrucao_x>
 |   <bloco>
 |   <if>
 |   <set>
 |   <foreach>
 |   <include>
 |   <parse>
 |   <macro>
 |   diretiva <instrucao_y>
 |   <stop>
 ;

<instrucao_x>           // Instrução (fatoração)
 ::= <referencia_x>     // Equivalente à <referencia>
 |   ε                        // Equivalente à <estatico>
 ;

<instrucao_y>           // Instrução (fatoração 2)
 ::= <macro_chamada_x>        // Equivalente à <macro_chamada>
 |   ε                  // Equivalente à <estatico>
 ;

...
```

Quadro 14 – Trecho da especificação sintática da linguagem de *templates*

3.1.3 Especificação semântica

Para a linguagem de *templates* foram especificadas ações semânticas, associadas a diversos recursos. No Quadro 15 é mostrado um trecho da gramática relativo a diretiva `#if` para controle de fluxo e no Quadro 16 é apresentado um trecho de arquivo de *template* utilizando o referido recurso.

```
...
<if>                // Diretiva "if"
 ::= if #121 "(" <expressao> ")" #125 <instrucoes> <elseif> end #122
 ;

<elseif>            // Diretiva "elseif" (complemento da "if")
 ::= ε
 |   elseif #123 "(" <expressao> ")" #125 <instrucoes> <elseif>
 |   <else>
 ;

<else>              // Diretiva "else" (complemento da "if")
 ::= else #124 <instrucoes>
 ;
...
```

Quadro 15 – Trecho da gramática da linguagem de *templates*, com ações semânticas

```
...
  #if ($objeto.ehQuadrado)      ...
  #elseif ($objeto.ehRedondo)   ...
  #else                         ...
  #end
...
```

Quadro 16 – Trecho de *template*, utilizando recurso de controle de fluxo

Para o recurso de controle de fluxo apresentado no Quadro 16, as ações semânticas são chamadas na seguinte ordem:

a) ação semântica `#121` (em `<if>`): início da diretiva `#if`, guarda o valor dos controles internos relativos à diretiva `#if` anteriormente tratada e inicializa estes controles para o processamento;
b) ação semântica `#125` (em `<if>`): reconhecimento do resultado do teste da diretiva `#if`;
c) ação semântica `#123` (em `<elseif>`): início do bloco `#elseif`, valida a possibilidade de processar este, condicionado ao não processamento de blocos anteriores;
d) ação semântica `#125` (em `<elseif>`): reconhecimento do resultado do teste da diretiva `#elseif`;
e) ação semântica `#124` (em `<else>`): início do bloco `#else`, valida a possibilidade de processar este, condicionado ao não processamento de blocos anteriores;
f) ação semântica `#122` (em `<if>`): fim da diretiva `#if`, restaura os controles relativos à diretiva `#if` anteriormente tratada.

Um recurso importante acerca da utilização de macros consiste em poder realizar sua chamada em qualquer local do *template*, mesmo estando antes de sua definição. Para isto o motor de *templates* possui o conceito de pré-processamento do *template*, que trata-se de uma análise realizada antes do processamento principal, na qual é obtida a lista das macros existentes, bem como são armazenados dados sobre elas – nome da macro, quantidade e nome

dos parâmetros e bloco de código.

A linguagem de *templates* possui um total de sessenta e seis ações semânticas, sendo sete relativas ao pré-processamento, cinqüenta e sete relativas ao processamento principal e duas relativas a ambos os processamentos (Quadro 17).

GRUPO DE AÇÕES SEMANTICAS	FUNÇÃO
Pré-processador e processador	
#1 - #2	Inicialização e finalização do pré-processamento e processamento do *template*.
Pré-processador	
#901 - #907	Obtenção da lista e dados das definições de macros.
Processador	
#3 - #6	Reconhecimento de texto estático e formatação.
#10 - #19	Reconhecimento de constantes (numérica, literal, booleana e vetorial) e de identificadores.
#21 - #29	Tratamento de operações aritméticas, lógicas e relacionais.
#31 - #40	Tratamento de referências.
#101	Reconhecimento de nome de variáveis.
#102	Processamento da diretiva `#set`.
#103 - #104	Reconhecimento de bloco de código.
#105	Reconhecimento de comentário.
#111 - #114	Processamento da diretiva `#foreach`.
#121 - #125	Processamento da diretiva `#if`.
#141 - #143	Processamento da diretiva `#include`.
#151	Processamento da diretiva `#parse`.
#161 - #162	Processamento da diretiva `#macro`.
#171	Processamento da diretiva `#stop`.
#181 - #183	Processamento da diretiva de chamada de macro.

Quadro 17 – Funcionalidade dos grupos de ações semânticas da linguagem de *templates*

3.2 REQUISITOS DO MOTOR DE *TEMPLATES*

O funcionamento de um motor de *templates* considera como entradas um *template* e um contexto, contendo dados da aplicação, e gera como saída um texto formatado a partir destas entradas. Mais especificamente, o funcionamento da biblioteca proposta dar-se-á da seguinte forma:

a) a aplicação gera para o motor de *templates* um contexto, contendo o mapeamento dos seus dados através de apelidos únicos, para uso nos *templates*;

b) a aplicação inicia o processo do motor de *templates*, informando um *template* a ser tratado;

c) o motor de *templates* analisa o *template*, interpretando e substituindo o código dinâmico conforme o contexto, e mantendo o código estático intacto;

d) o motor de *templates* retorna o texto formatado para que a aplicação faça uso.

A seguir são apresentados os requisitos funcionais (Quadro 18) e não funcionais (Quadro 19) atendidos pela biblioteca.

REQUISITOS FUNCIONAIS	CASO DE USO
RF01: Permitir ao programador mapear dados de sua aplicação a um contexto.	UC01
RF02: Analisar léxica e sintaticamente arquivos de *template*, reconhecendo código estático e código dinâmico.	UC03
RF03: Gerar saída com base no *template*, interpretando e substituindo o código dinâmico conforme os dados mapeados no contexto.	UC03

Quadro 18 – Requisitos funcionais

REQUISITOS NÃO FUNCIONAIS
RNF01: Ser implementado no ambiente Delphi 7.
RNF02: Ser compatível e permitir sua inclusão e uso em aplicações desenvolvidas em Delphi 7.
RNF03: Possuir funcionalidades em sua linguagem de *templates* semelhantes às da VTL do Velocity.

Quadro 19 – Requisitos não funcionais

3.3 ESPECIFICAÇÃO DO MOTOR DE *TEMPLATES*

O motor de *templates* foi especificado através da ferramenta Enterprise Architect (SPARX SYSTEMS, 2007), utilizando os conceitos de orientação a objetos e baseando-se nos diagramas da UML, gerando como produtos os diagramas de caso de uso, de seqüência e de classes.

3.3.1 Casos de uso

A biblioteca possui três casos de uso (Figura 4), sendo que, segundo uma seqüência lógica, os dois primeiros podem ser executados em paralelo e o terceiro deve somente ser executado após os dois primeiros. Todos os casos de uso são executados pelo ator `Desenvolvedor`, que representa o programador da aplicação que utiliza o motor de *templates*.

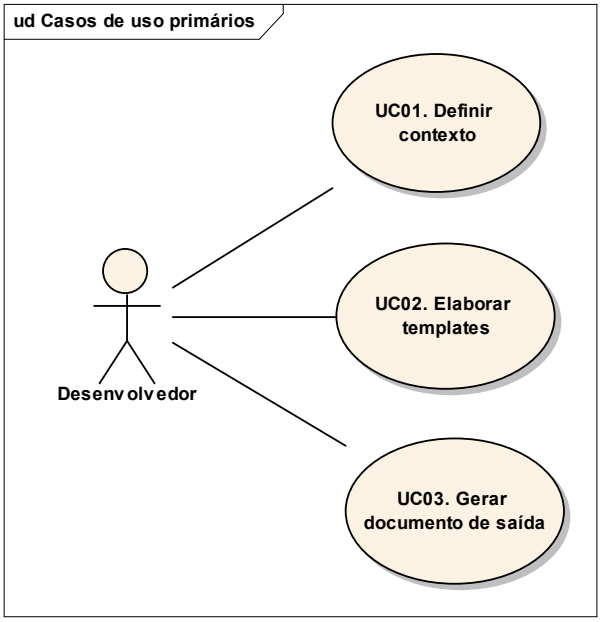


Figura 4 – Diagrama de casos de uso

O primeiro caso de uso (Quadro 20), designado `Definir contexto`, descreve como o desenvolvedor pode mapear os dados da aplicação em um contexto, para serem referenciados no *template*. Ele possui, além do cenário principal, um cenário alternativo, que permite a definição de múltiplos dados no contexto, e dois cenários de exceção, informando possíveis erros encontrados durante o processo.

UC01 – Definir contexto: possibilidade de mapear dados da aplicação, através de um identificador único, a um contexto.	
Requisitos atendidos	RF01.
Pré-condições	Não possui.
Cenário principal	1) O desenvolvedor informa uma chave e um valor a serem adicionados no contexto. 2) A biblioteca valida se a chave informada ainda não existe e se o valor informado é válido.
Fluxo alternativo 01	Novo valor: Após o passo 2 do cenário principal, caso o desenvolvedor deseje adicionar mais valores ao contexto, volta ao passo 1 do cenário principal.
Exceção 01	Chave duplicada: No passo 2 do cenário principal, se a chave informada pelo desenvolvedor já estiver contida no contexto, o valor antigo associado à chave é excluído.
Exceção 02	Valor inválido: No passo 2 do cenário principal, se o valor informado pelo desenvolvedor for inválido, uma exceção é gerada.
Pós-condições	Um contexto deve estar definido, com dados da aplicação mapeados.

Quadro 20 – Caso de uso `UC01`

O segundo caso de uso (Quadro 21), designado de `Elaborar templates`, trata da criação do *template*, um artefato essencial para o motor de *template*s. Porém o objetivo deste não é um dos objetivos da biblioteca, por isso este caso de uso não possui maior detalhamento. A biblioteca também não disponibiliza uma ferramenta para elaboração do

template, a qual pode ser feita utilizando-se um editor de texto ASCII.

UC02 – Elaborar *templates*: possibilidade de elaboração de *templates*, utilizando a linguagem de *templates* disponibilizada.	
Requisitos atendidos	RF04 e RNF03.
Pré-condições	Não possui.
Cenário principal	O usuário cria um arquivo de *template*.
Pós-condições	Um arquivo de *template* deve ter sido criado.

Quadro 21 – Caso de uso `UC02`

O terceiro caso de uso (Quadro 22), designado `Gerar documento de saída`, é o caso de uso principal da biblioteca, visto que é durante sua execução que o texto formatado de saída, objetivo do motor de *templates*, é gerado. Ele possui somente o cenário principal e quatro cenários de exceção, informando possíveis erros encontrados durante o processo. Sua execução é expressa no diagrama de seqüência apresentado naFigura 5.

UC03 – Gerar documento de saída: possibilidade de gerar o texto formatado de saída, com base em um *template*, conforme os dados mapeados em um contexto, todos previamente informados.	
Requisitos atendidos	RF02, RF03, RF04 e RNF03.
Pré-condições	Um contexto deve estar definido. Um *template* deve ter sido elaborado.
Cenário principal	1) O desenvolvedor informa o *template* a ser utilizado. 2) A biblioteca valida o *template*. 3) O desenvolvedor define o contexto a ser utilizado. 4) A biblioteca valida o contexto. 5) O desenvolvedor chama método para início da geração do texto formatado de saída. 6) A biblioteca analisa e processa o *template*. 7) O desenvolvedor informa o diretório e nome do arquivo a ser gerado como documento de saída. 8) A biblioteca gera o documento de saída.
Exceção 01	*Template* inválido: No passo 2 do cenário principal, se o *template* informado pelo desenvolvedor for inválido, uma exceção é gerada. Um *template* pode ser inválido, por exemplo, por possuir erros léxicos, sintáticos ou semânticos.
Exceção 02	Contexto inválido: No passo 4 do cenário principal, se o contexto informado pelo desenvolvedor for inválido, uma exceção é gerada. Um contexto pode ser inválido caso possua mapeamentos para objetos não suportados.
Exceção 03	Nome de arquivo ou diretório inválido: No passo 7 do cenário principal, se o nome ou diretório do arquivo de saída a ser gerado for inválido – por possuir caracteres não permitidos – uma exceção é gerada.
Exceção 04	Exceção durante o processo: Nos passos 6 e 8 do cenário principal, se ocorrer algum problema não esperado no processo de geração do documento de saída – como falta de memória – uma exceção é gerada.
Pós-condições	O documento de saída deve estar gerado.

Quadro 22 – Caso de uso `UC03`

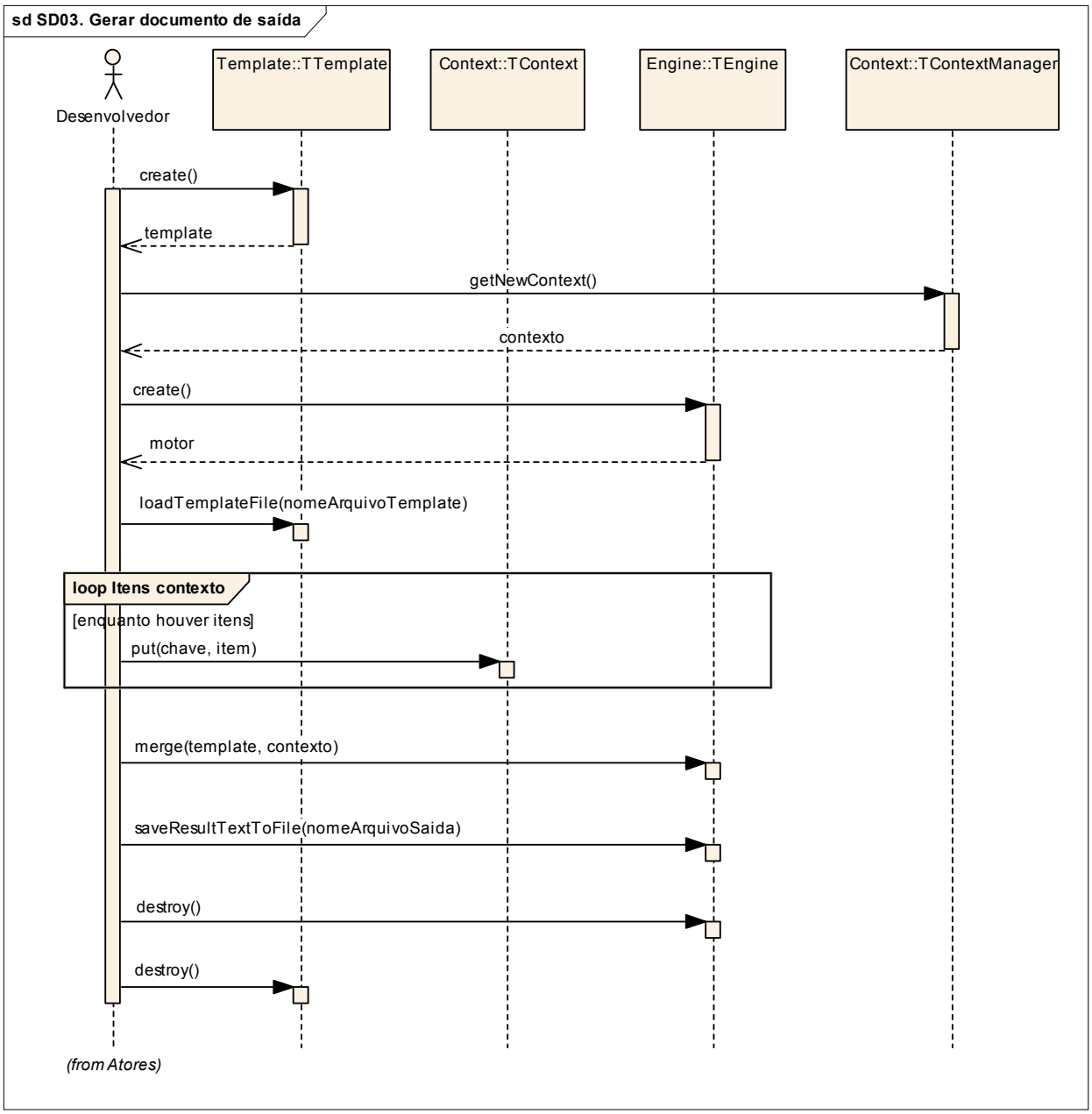


Figura 5 – Diagrama de seqüência, para `UC03`

3.3.2 Diagrama de classes

O diagrama de classes[4] apresenta uma visão de como as classes estão estruturadas e relacionadas. Como são muitas as classes necessárias para a implementação da biblioteca, elas estão reunidas, conforme as ligações lógicas entre si, em pacotes (Figura 6).

[4] Nos diagramas de classes desta seção os tipos de dado e classes não especificados são nativos do Delphi.

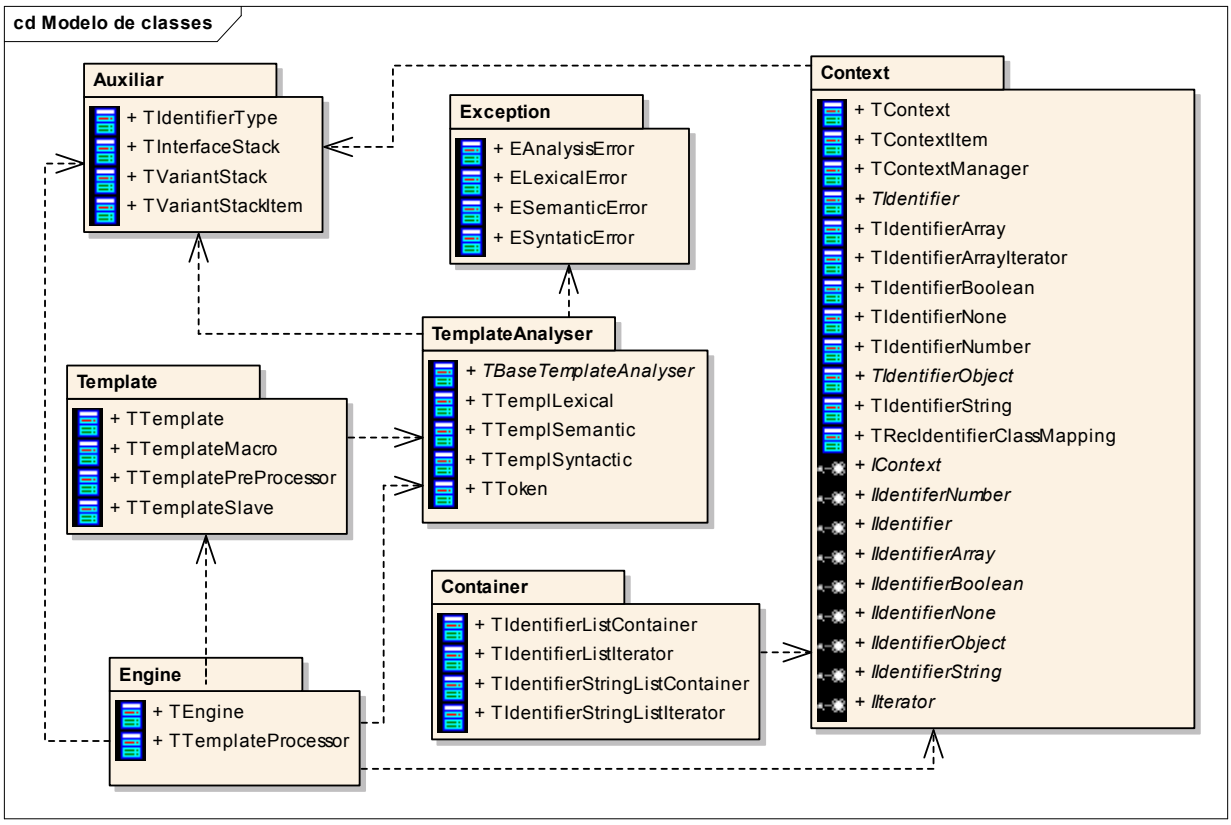


Figura 6 – Pacotes da biblioteca

3.3.2.1 Pacote `Auxiliar`

O primeiro pacote é denominado `Auxiliar` e contém classes básicas, com função auxiliar para o funcionamento da biblioteca (Figura 7). Fazem parte deste pacote as classes e enumerações:

a) `TIdentifierType`: enumeração que define os tipos de identificadores, possuindo os possíveis valores: nulo (ou sem tipo), numérico, alfanumérico, booleano, vetorial e de objeto. Esta definição é utilizada durante o processamento para verificar a validade de operações aritméticas, relacionais e lógicas, bem como definir a disponibilidade de recursos, como o uso de propriedades e métodos;

b) `TVariantStack`: classe que implementa uma pilha de valores variáveis. Esta classe é utilizada pelo analisador semântico para guardar valores de controle importantes para a análise, como marcações de posições e nomes de variáveis;

c) `TVariantStackItem`: classe que implementa um item da pilha de valores variáveis. Esta classe somente é utilizada pela `TVariantStack` para armazenamento dos valores;

d) `TInterfaceStack`: classe que implementa uma pilha de interfaces. Esta classe é utilizada durante a análise semântica para armazenamento de identificadores que vão sendo reconhecidos, calculados ou processados. É utilizada nas mais diversas rotinas, como no reconhecimento de constantes e da variável principal da diretiva `#foreach`, e em operações aritméticas, relacionais e lógicas.

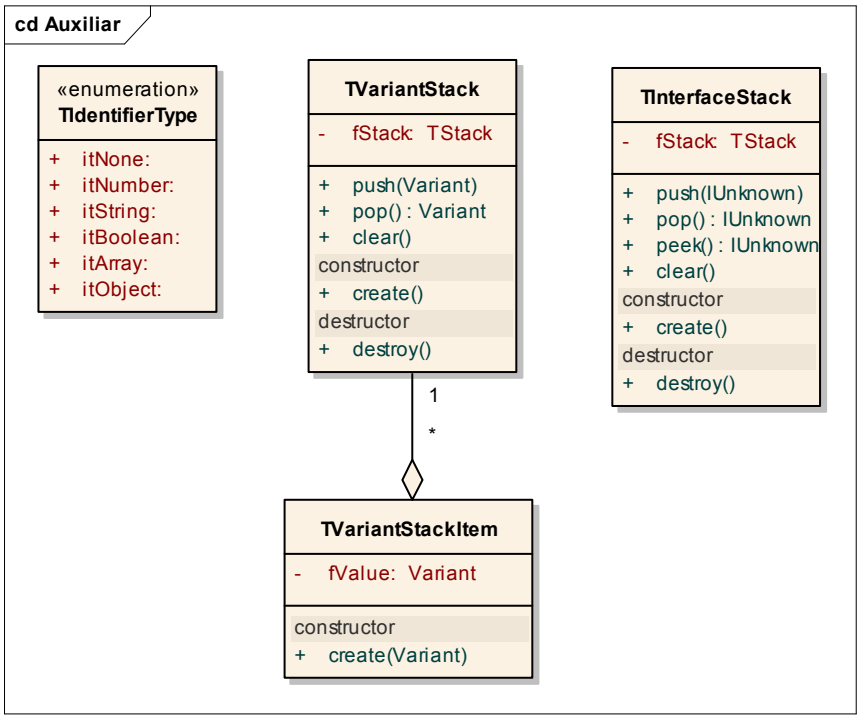


Figura 7 – Pacote `Auxiliar`

3.3.2.2 Pacote `Exception`

O segundo pacote, gerado pelo GALS, é denominado `Exception` e contém classes de exceção (Figura 8). Fazem parte deste pacote as classes:

a) `EAnalysisError`: classe que implementa a base de uma exceção no processo de análise;

b) `ELexicalError`: classe derivada da classe `EAnalysisError`, que implementa uma exceção no processo de análise léxica;

c) `ESyntaticError`: classe derivada da classe `EAnalysisError`, que implementa uma exceção no processo de análise sintática;

d) `ESemanticError`: classe derivada da classe `EAnalysisError`, que implementa uma exceção no processo de análise semântica.

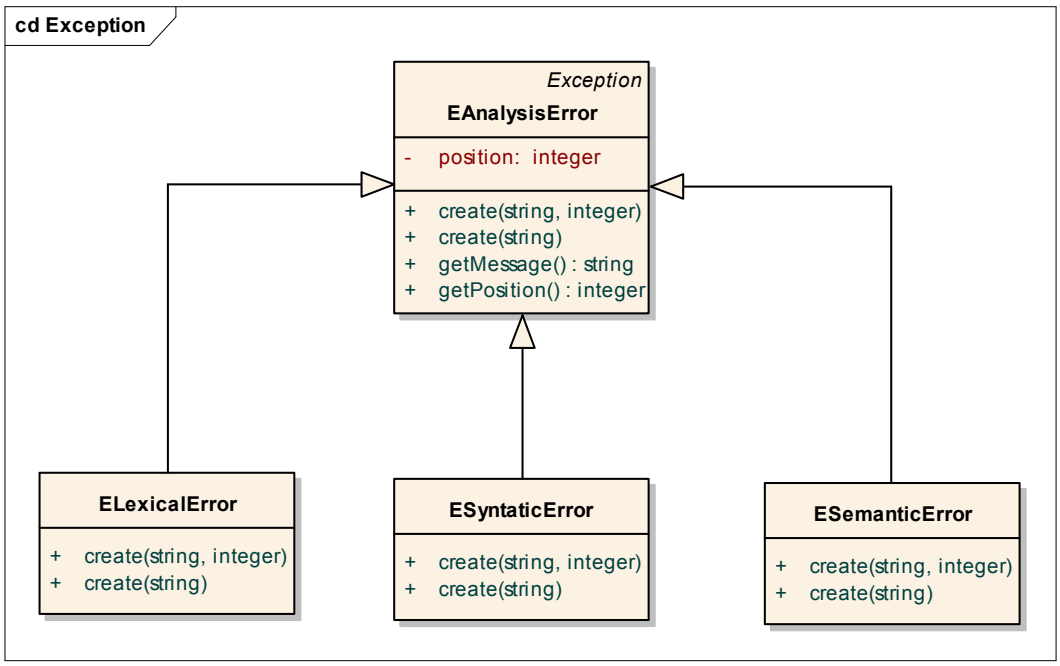


Figura 8 – Pacote `Exception`

3.3.2.3 Pacote `Context`

O terceiro pacote é denominado `Context` e contém classes relativas à implementação de identificadores e do contexto. Inicialmente é apresentado o conjunto de recursos genéricos, em seguida os agrupamentos de recursos relativos a cada tipo de identificador e ao final os recursos relativos ao gerenciamento do contexto.

Como recursos genéricos do pacote, têm-se as seguintes classes e interfaces (Figura 9):

a) `IContext`: interface que define uma implementação de contexto para o motor de *templates*. Possui como principais métodos: `get()`, que recebe a chave do identificador no contexto e retorna este; `put()`, que mapeia no contexto um identificador através de uma chave e `remove()`, que remove um identificador do contexto;

b) `IIdentifier`: interface que define uma implementação genérica de identificador. Especifica métodos para retornar o tipo do identificador – usado para caracterizar o identificador e permitir utilizar recursos específicos para cada tipo –, retornar o seu valor como alfanumérico – utilizado para, quando necessário, inserir o valor do identificador no texto formatado de saída –, indicar sua existência como iterável e retornar uma instância da interface `IIterator` para tratar esta iteração – usado

no tratamento da diretiva `#foreach` –, permitir sua cópia ou comparação com outros identificadores;

c) `IIterator`: interface que define recursos para a iteração de identificadores, instâncias da interface `IIdentifier`. Seus métodos permitem sua inicialização – para posicionamento no primeiro item da iteração –, o reconhecimento da existência de mais itens e o retorno de um próximo item. Este recurso é implementado para os identificadores que possuem itens iteráveis, sendo usada no tratamento da diretiva `#foreach`. A iteração ocorre de forma unidirecional e linear, sempre retornando um próximo item, à exceção da inicialização, após a qual o próximo item retornado é o primeiro;

d) `TContext`: classe que implementa a interface `IContext` como contexto do motor de *templates*. Para manter o mapeamento de identificadores, esta classe possui o atributo privado `fItems` do tipo `TObjectList` – lista de objetos. Cada identificador mapeado no contexto é na verdade uma instância da classe `TContextItem` adicionada à lista. Esta classe possui o método privado `getContextItem()` que é responsável por retornar a instância de `TContextItem` relativa a determinada chave;

e) `TContextItem`: classe usada internamente pela classe `TContext` para manter os mapeamentos de identificadores. Esta possui atributos e métodos com a finalidade de manter uma instância de `IIdentifier` e uma chave para a mesma;

f) `TIdentifier`: classe abstrata que implementa a interface `IIdentifier`. Possui classes derivadas para cada tipo de identificador, e define para estas o comportamento padrão dos métodos da interface.

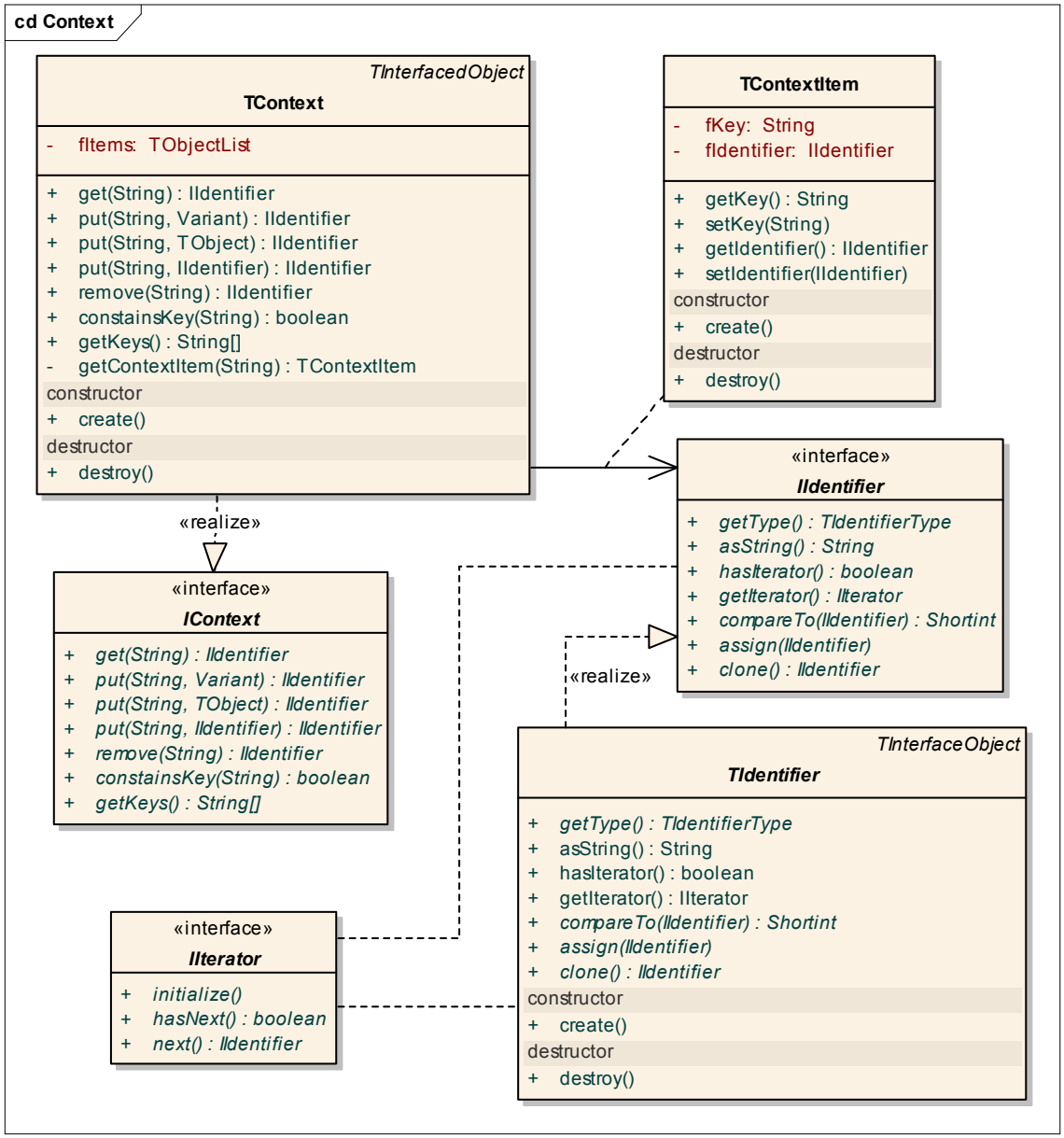


Figura 9 – Pacote `Context`, recursos genéricos

Como recursos relativos ao identificador sem tipo, ou nulo, têm-se as seguintes classes e interfaces:

a) `IIdentifierNone`: interface derivada da `IIdentifier`, que deve permitir acessar os recursos específicos para identificadores nulos. Tal interface era desnecessária na implementação atual, porém foi criada para manter a padronização em relação aos identificadores de outros tipos;

b) `TIdentifierNone`: classe derivada da `TIdentifier`, que implementa a interface `IIdentifierNone`, e atende às especificidades relativas ao controle de identificadores nulos para o motor de *templates*.

A Figura 10 apresenta o diagrama de classes para os recursos descritos acima, além de

apresentar novamente a interface `IIdentifier` e a classe `TIdentifier`, para um melhor entendimento do relacionamento entre os recursos.

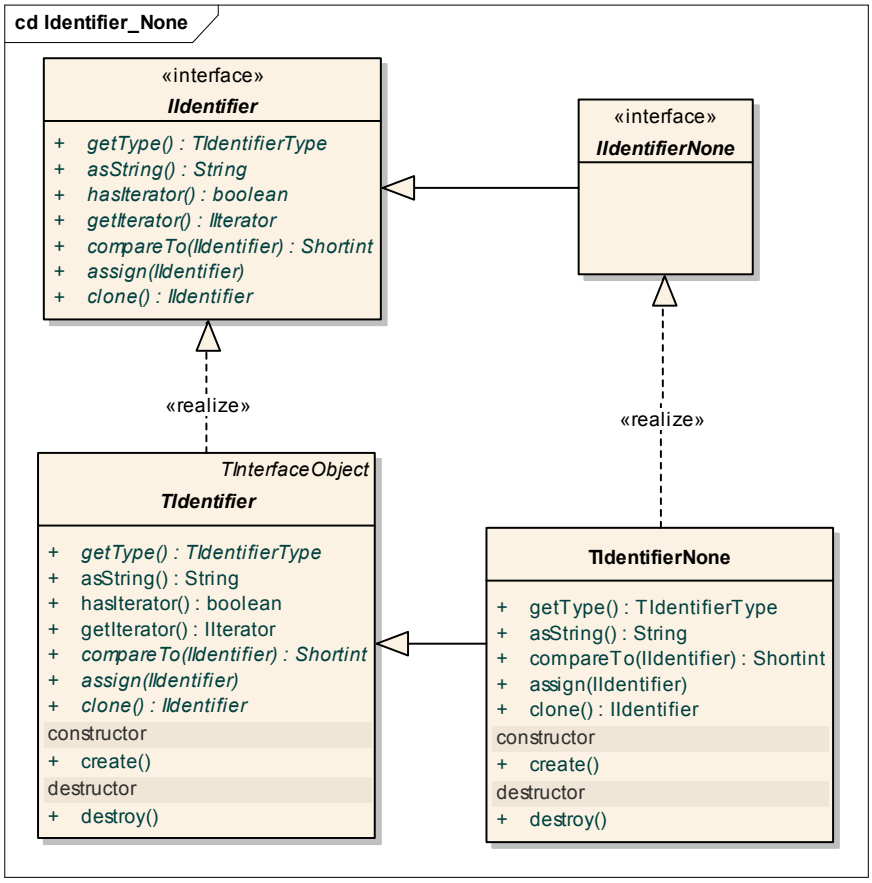


Figura 10 – Pacote `Context`, identificador sem tipo

Como recursos relativos ao identificador numérico, têm-se as seguintes classes e interfaces:

a) `IIdentifierNumber`: interface derivada da `IIdentifier`, que permite manter um valor numérico e acessar os recursos específicos para este tipo de elemento. Dentre os métodos especificados, destacam-se `getValue()` e `setValue()`, que permitem respectivamente, retornar e alterar o valor numérico do identificador. Esta interface é utilizada no motor de *templates* em recursos como reconhecimento de constantes numéricas e vetoriais, e operações aritméticas;

b) `TIdentifierNumber`: classe derivada da `TIdentifier`, que implementa a interface `IIdentifierNumber`, e atende às especificidades relativas ao controle de identificadores numéricos para o motor de *templates*.

A Figura 11 apresenta o diagrama de classes para os recursos descritos acima, bem como a interface `IIdentifier` e a classe `TIdentifier`.

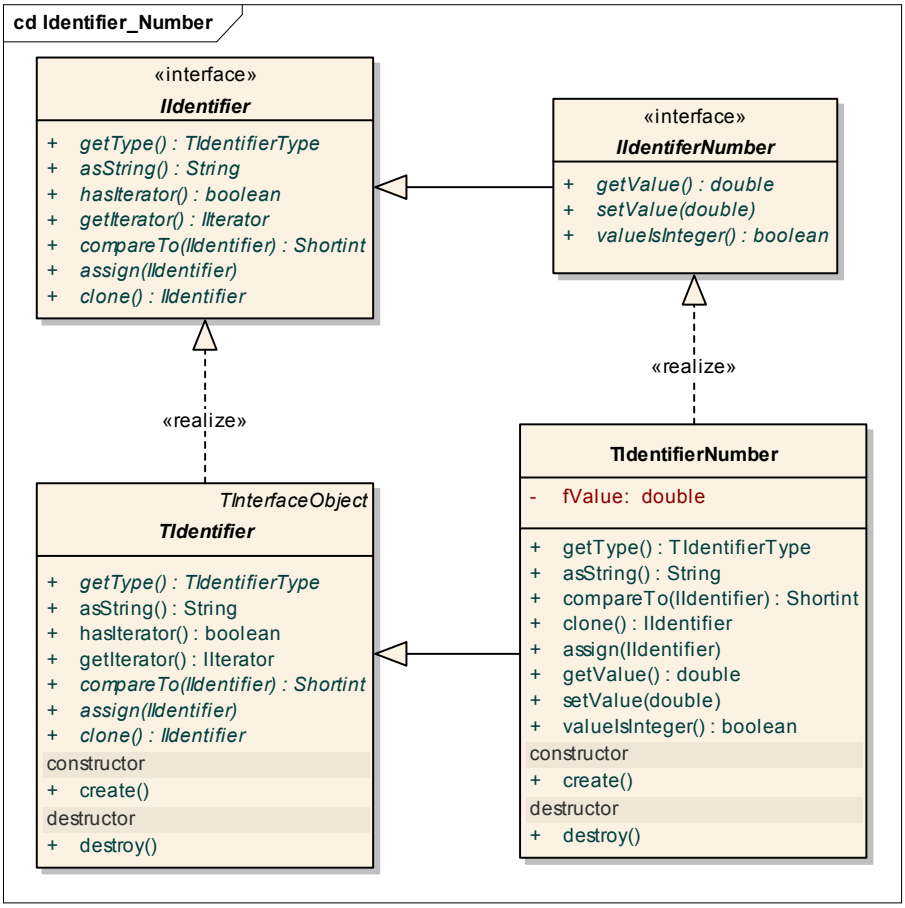


Figura 11 – Pacote `Context`, identificador numérico

Como recursos relativos ao identificador alfanumérico, têm-se as seguintes classes e interfaces:

a) `IIdentifierString`: interface derivada da `IIdentifier`, que permite manter um valor alfanumérico e acessar os recursos específicos para este tipo de elemento. Os métodos especificados são `getValue()` e `setValue()`, que permitem, respectivamente, retornar e alterar o valor alfanumérico do identificador. Esta interface é utilizada no motor de *templates* em recursos como reconhecimento de constantes alfanuméricas e vetoriais, operações de concatenação e nas diretivas `#include` e `#parse` – para reconhecimentos do nome dos arquivos informados;

b) `TIdentifierString`: classe derivada da `TIdentifier`, que implementa a interface `IIdentifierString`, e atende às especificidades relativas ao controle de identificadores alfanuméricos para o motor de *templates*.

A Figura 12 apresenta o diagrama de classes para os recursos descritos acima, assim como a interface `IIdentifier` e a classe `TIdentifier`.

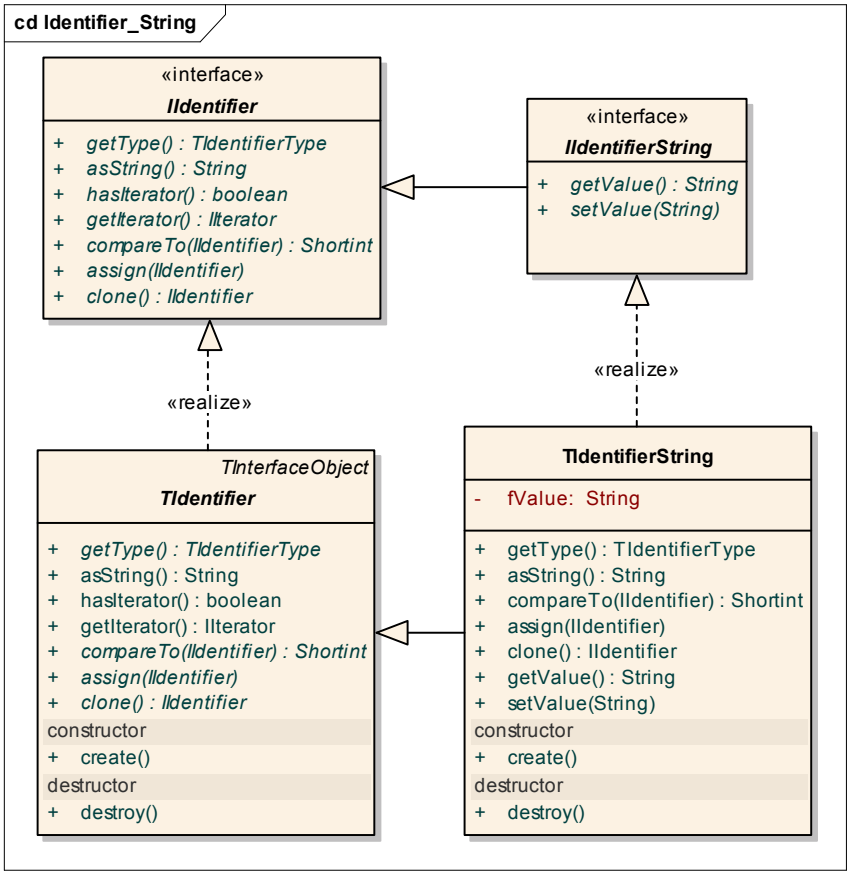


Figura 12 – Pacote `Context`, identificador alfanumérico

Como recursos relativos ao identificador booleano, têm-se as seguintes classes e interfaces:

a) `IIdentifierBoolean`: interface derivada da `IIdentifier`, que permite manter um valor booleano e acessar os recursos específicos para este tipo de elemento. Os métodos especificados são `getValue()` e `setValue()`, que permitem, respectivamente, retornar e alterar o valor booleano do identificador. Esta interface é utilizada no motor de *templates* em recursos como reconhecimento de constantes booleanas e vetoriais, operações relacionais e lógicas, e na diretiva `#if` – para verificação do resultado dos testes;

b) `TIdentifierBoolean`: classe derivada da `TIdentifier`, que implementa a interface `IIdentifierBoolean`, e atende às especificidades relativas ao controle de identificadores booleanos para o motor de *templates*.

A Figura 13 apresenta o diagrama de classes para os recursos descritos acima.

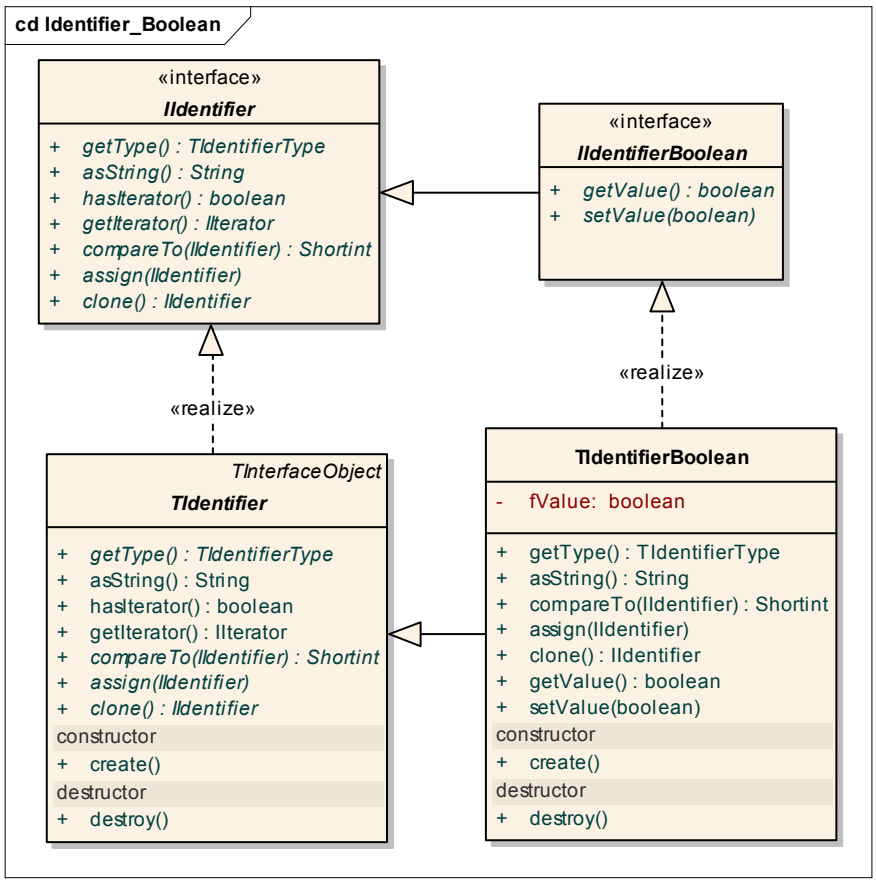


Figura 13 – Pacote `Context`, identificador booleano

Como recursos relativos ao identificador vetorial, têm-se as seguintes classes e interfaces:

a) `IIdentifierArray`: interface derivada da `IIdentifier`, que permite manter um vetor de valores e acessar os recursos específicos para este tipo de estrutura. Um identificador de vetor é na verdade uma coleção de outros identificadores de qualquer tipo, sendo que seus elementos são instâncias da interface `IIdentifier`. Dentre os métodos especificados destacam-se `getValue()` e `setValue()`, que permitem, respectivamente, retornar e alterar cada identificador, elemento do vetor, e `getCount()`, que retorna a quantidade de elementos do vetor. Esta interface é utilizada no motor de *templates* no reconhecimento de constantes vetoriais;

b) `TIdentifierArray`: classe derivada da `TIdentifier`, que implementa a interface `IIdentifierArray` e atende às especificidades relativas ao controle de identificadores vetoriais para o motor de *templates*;

c) `TIdentifierArrayIterator`: classe que implementa a interface `IIterator`, propiciando a iteração entre elementos de identificadores vetoriais.

A Figura 14 apresenta o diagrama de classes para os recursos descritos acima, além de

apresentar novamente as interfaces `IIdentifier` e `IIterator`, e a classe `TIdentifier`, para um melhor entendimento do relacionamento entre os recursos.

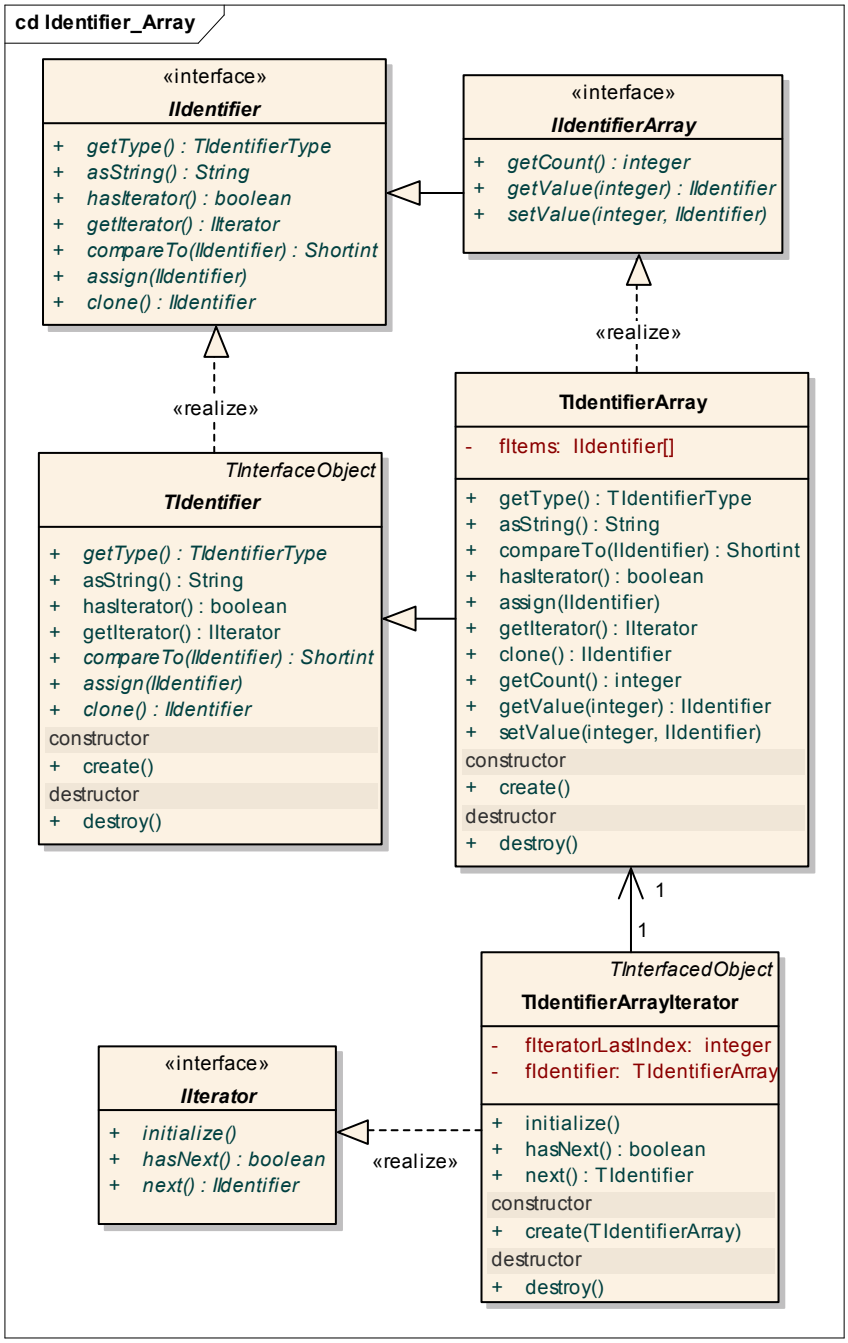


Figura 14 – Pacote `Context`, identificador vetorial

Como recursos relativos ao identificador de objeto, têm-se as seguintes classes e interfaces:

a) `IIdentifierObject`: interface derivada da `IIdentifier`, que permite manter uma referência para um objeto e acessar os recursos específicos para este tipo de elemento. Os métodos especificados permitem o retorno do valor de propriedades e a execução de métodos do objeto, sendo estes definidos e implementados nas

classes contêineres para os objetos. Esta interface é utilizada no motor de *templates* no reconhecimento de propriedades e métodos de identificadores de objeto;

b) `TIdentifierObject`: classe abstrata derivada da `TIdentifier`, que implementa a interface `IIdentifierObject`. As classes derivadas desta são implementações de contêineres para objetos de outras classes, sendo que nestes estão programados quais propriedades e métodos existem – métodos `hasProperty()` e `hasMethod()`, respectivamente – e a chamada a estes – métodos `getProperty()` e `executeMethod()` –, além da verificação da possibilidade de iteração de seus valores, bem como o retorno da classe para iteração – métodos `hasIterator()` e `getIterator()` da interface `IIdentifier`. Esta classe também utiliza o recurso RTTI[5] do Delphi para identificar atributos definidos como `published` das classes, e ter acesso a elas sem a necessidade de programar seu acesso no contêiner.

A Figura 15 apresenta o diagrama de classes para os recursos descritos acima.

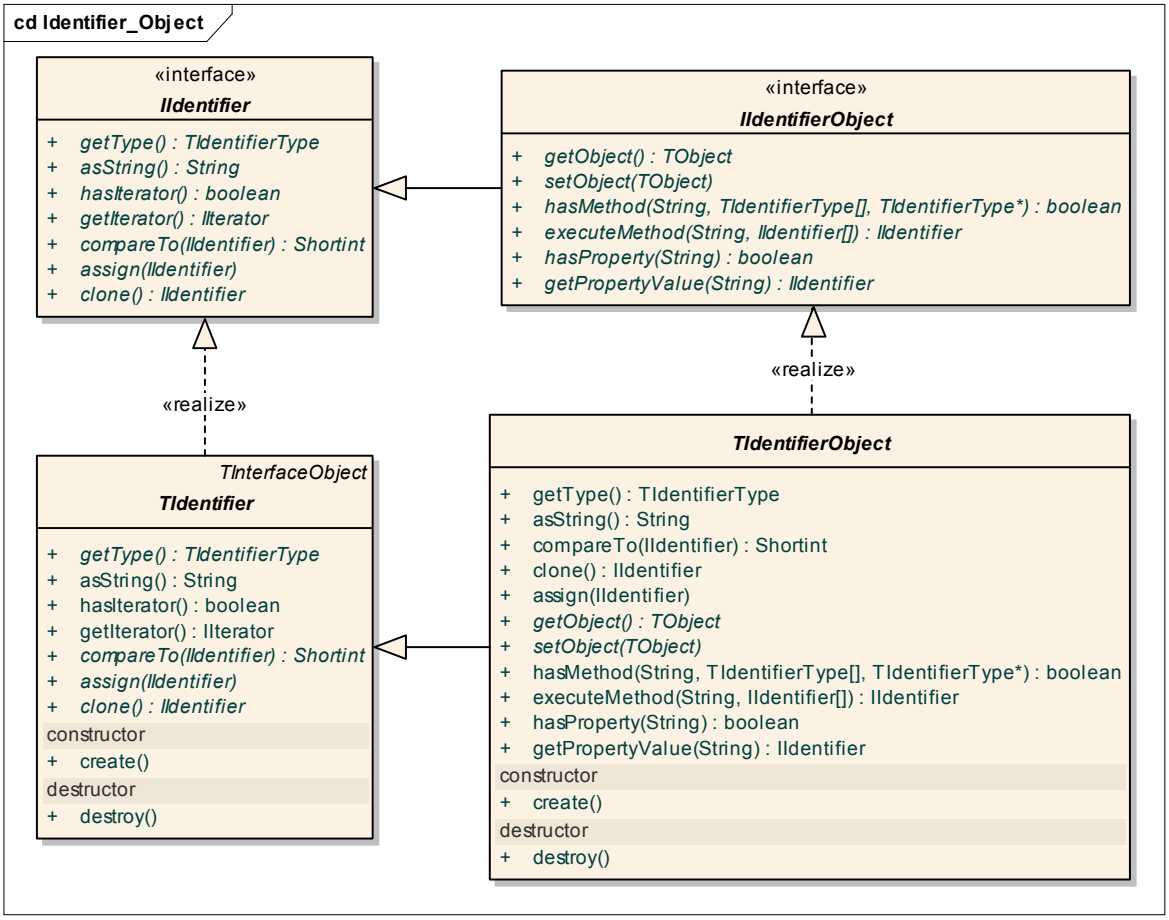


Figura 15 – Pacote `Context`, identificador de objeto

[5] RTTI é um recurso utilizado pelo Delphi para permitir a montagem do seu ambiente de desenvolvimento, mas que também está disponível para o uso dos programadores.

Como recursos referentes ao gerenciamento do contexto, têm-se as seguintes classes e registros (Figura 16):

a) `TContextManager`: classe de instância única que permite gerenciar o contexto do motor de *templates*. Ela permite a criação de instâncias de contexto, bem como de todos os tipos de identificadores. O método `getInstance()` permite acesso a instância única da classe. Porém a classe tem por principal função manter um mapeamento entre classes de objetos e classes de contêineres destes objetos, derivadas da classe `TIdentifierObject`, a fim de que, quando um objeto for mapeado no contexto, saiba-se qual classe contêiner deve ser instanciada para manter a referência ao objeto e intermediar o acesso aos dados do mesmo;

b) `TRecIdentifierClassMapping`: registro usado internamente pela classe `TContextManager` para manter o mapeamento entre uma classe de objeto e sua classe contêiner. Fazem parte do registro os atributos `objectClass` e `identifierClass`, que referenciam a classe de objeto e a classe contêiner, respectivamente.

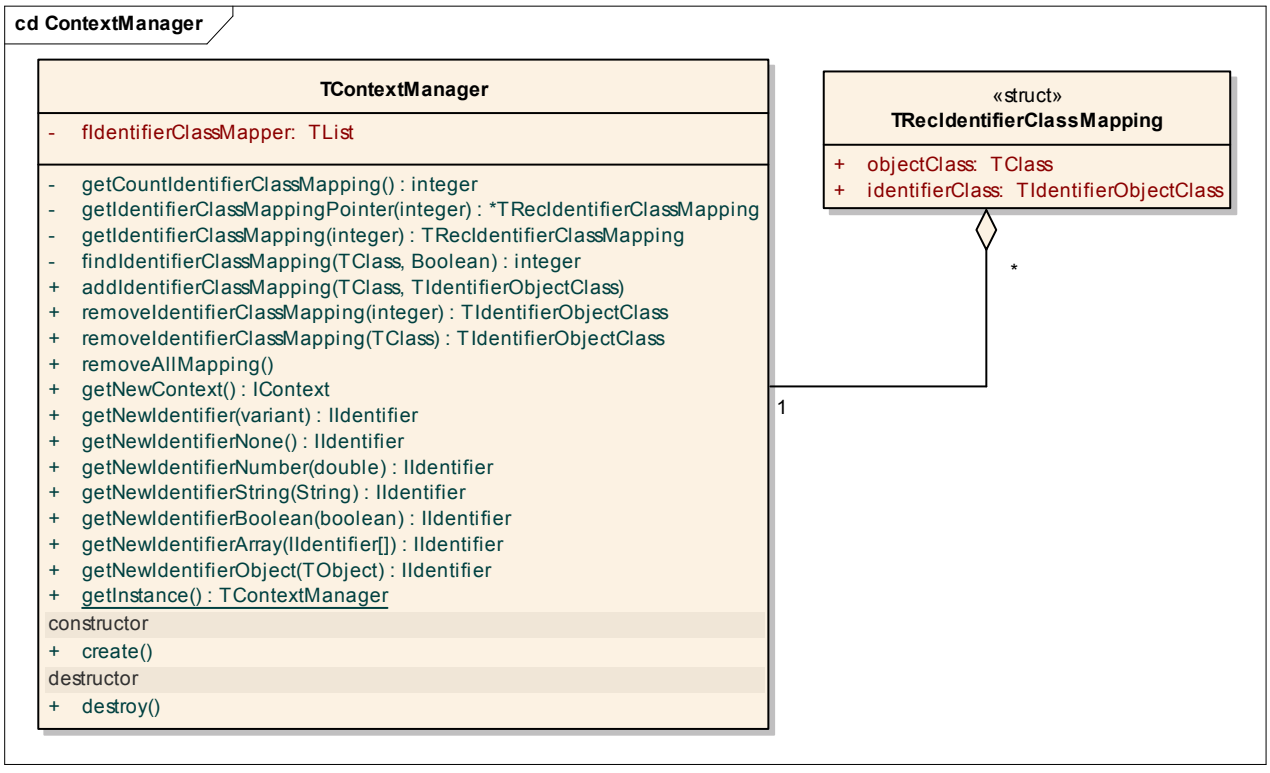


Figura 16 – Pacote `Context`, gerenciamento do contexto

3.3.2.4 Pacote `Container`

O quarto pacote (Figura 17) é denominado `Container` e contém classes que servem de contêineres a fim de permitir que outras classes sejam mapeadas como identificadores no contexto. Estão implementados contêineres somente para duas classes comumente utilizadas no Delphi: `TList` e `TStringList`. Fazem parte deste pacote as classes:

a) `TIdentifierListContainer`: classe derivada da `TIdentifierObject`, e que implementa o contêiner para objetos da classe `TList` do Delphi. Esta disponibiliza algumas propriedades e métodos para serem utilizados no *template*, além de permitir a iteração entre seus elementos na diretiva `#foreach` através de uma instância da classe `TIdentifierListIterator`;

b) `TIdentifierListIterator`: classe que implementa a interface `IIterator`, e possibilita a iteração entre os elementos da classe `TIdentifierListContainer`;

c) `TIdentifierStringListContainer`: classe derivada da `TIdentifierObject`, e que implementa o contêiner para objetos da classe `TStringList` do Delphi. Esta disponibiliza algumas propriedades e métodos para serem utilizados no *template*, além de permitir a iteração entre seus elementos na diretiva `#foreach` através de uma instância da classe `TIdentifierStringListIterator`;

d) `TIdentifierStringListIterator`: classe que implementa a interface `IIterator`, e possibilita a iteração entre os elementos da classe `TIdentifierStringListContainer`.

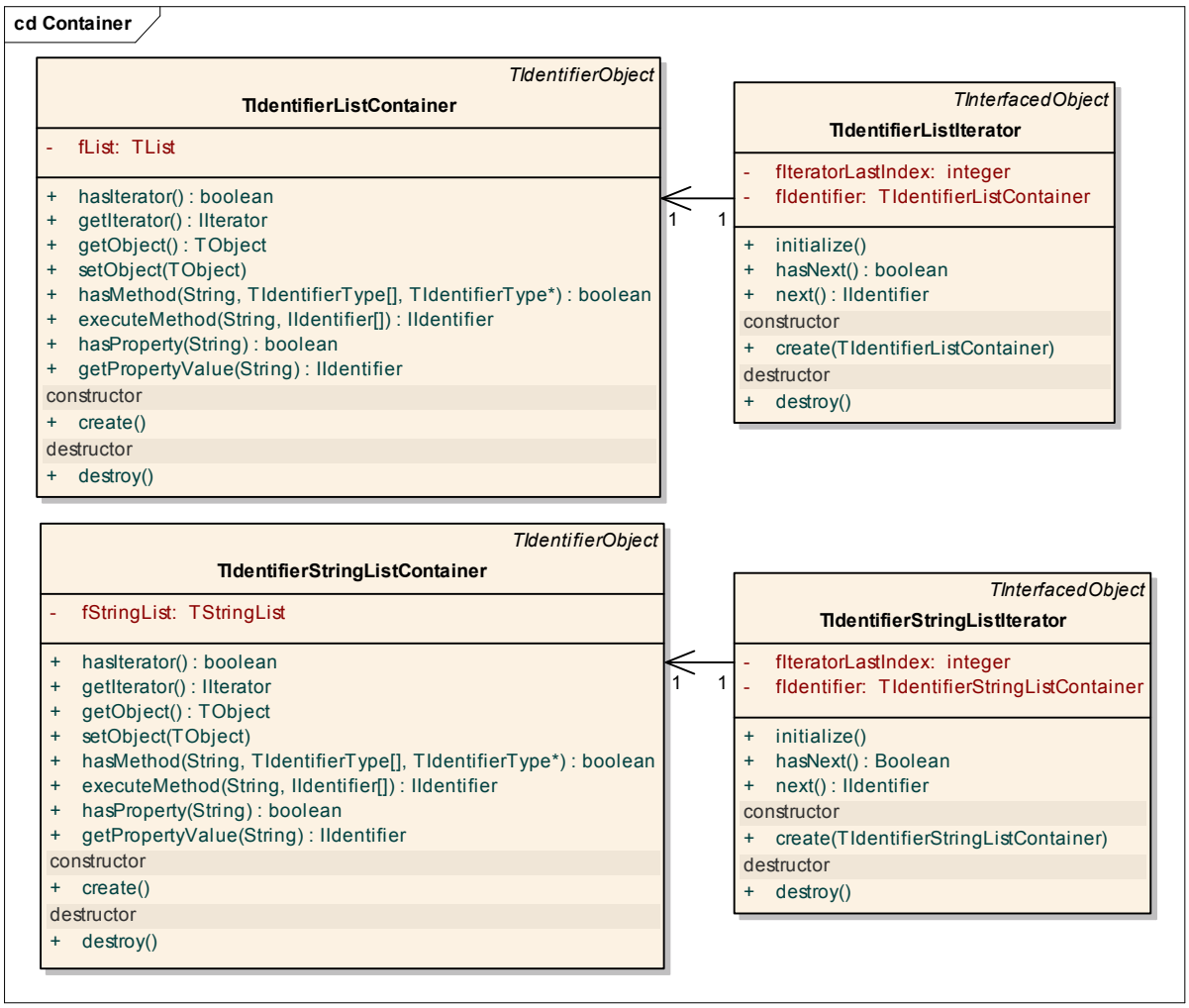


Figura 17 – Pacote `Container`

3.3.2.5 Pacote `TemplateAnalyser`

O quinto pacote é denominado `TemplateAnalyser` e contém classes que tratam das análises léxica, sintática e semântica dos *templates* (Figura 18). Fazem parte deste pacote as classes:

a) `TToken`: classe gerada pelo GALS, que representa um *token* reconhecido durante a análise léxica. Esta classe é utilizada durante as análises sintática e semântica, a fim de validar e reconhecer as construções da linguagem. Cada instância desta possui o lexema e a posição do *token* no *template*;

b) `TTemplLexical`: classe gerada pelo GALS, que possui a função de ler seqüencialmente os caracteres do *template* e agrupá-los em *tokens*, os quais são encapsulados em instâncias da classe `TToken`. Esta classe também é responsável

por reconhecer os erros léxicos, porém sua principal contribuição quando a identificação de erros é determinar a posição dos *tokens*, informação importante no reconhecimento de erros sintáticos e semânticos;

c) `TTemplSyntactic`: classe gerada pelo GALS, porém alterada em dois pontos: um método público foi removido por não ser utilizado e o atributo `tokenAtual` foi criado, com o objetivo de publicar o valor do atributo privado `currentToken` para que o lexema deste possa ser usado pela classe `TBaseTemplateAnalyser` na montagem da mensagem de erro sintático. Sua função é analisar sintaticamente o *template*, requisitando ao analisador léxico seqüencialmente os *tokens*, reconhecendo as construções conforme a gramática e invocando os métodos do analisador semântico. Ela também reconhece os erros sintáticos quando alguma seqüência de *tokens* não corresponde à gramática, sendo que a mensagem do erro é tratada pela classe `TBaseTemplateAnalyser`;

d) `TTemplSemantic`: classe gerada pelo GALS, porém alterada para que seu único método – `executeAction()` – possuísse o modificador `virtual`, assim a classe `TBaseTemplateAnalyser`, derivada desta, pôde implementar tal método. Esta classe foi mantida somente para compatibilidade com os demais códigos fonte gerados pelo GALS;

e) `TBaseTemplateAnalyser`: classe que implementa os recursos básicos para o analisador de *templates*, e deriva nas classes que implementam a análise – `TTemplatePreProcessor` e `TTemplateProcessor` –, detalhadas posteriormente. Dentre os métodos dessa classe, destacam-se `getTemplateCode()`, abstrato e implementado pelas duas classes derivadas para retornar o texto do *template*, e `process()`, que instancia os analisadores léxico e sintático e invoca a análise utilizando-se a si mesmo como analisador semântico – característica implementada pelas classes derivadas. O método `process()` também trata os possíveis erros encontrados durante o processo, informando linha e coluna onde reside o erro.

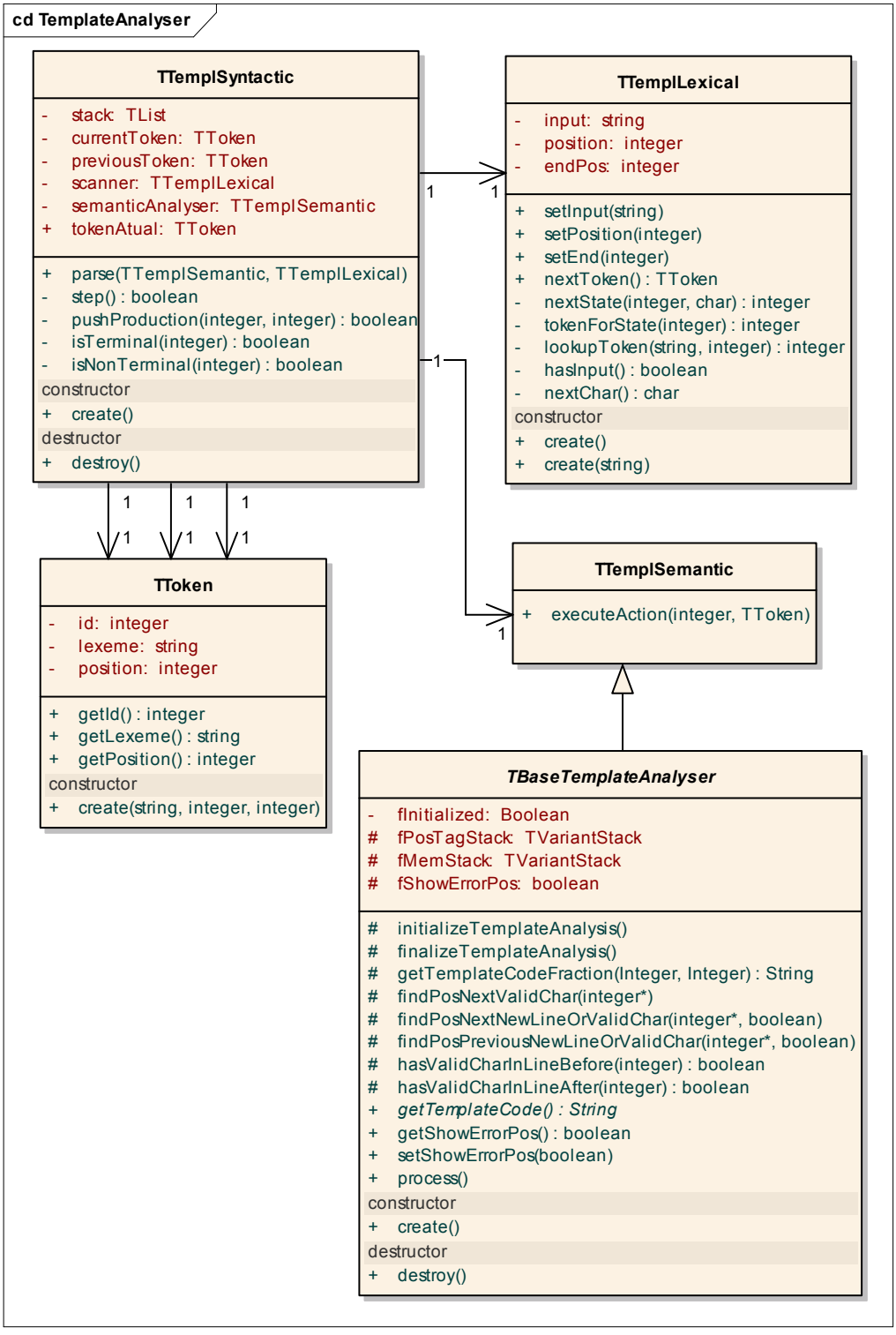


Figura 18 – Pacote `TemplateAnalyser`

3.3.2.6 Pacote `Template`

O sexto pacote é denominado `Template` e contém classes que mantêm os dados

referentes aos *templates*, além da classe responsável pelo seu pré-processamento (Figura 19). Fazem parte deste pacote as classes:

a) `TTemplate`: classe responsável por manter os dados referentes a um *template*. Esta classe mantém o código fonte do *template* e a listagem de macros, dentre outras informações. Ao receber o código fonte do *template*, a classe desencadeia o pré-processamento deste através da classe `TTemplateProcessor`, a fim de descobrir os dados de definição das macros, que são armazenados em instâncias da classe `TTemplateMacro` na lista interna `fMacros`. O diretório de onde o *template* foi carregado é outra informação guardada, para ser usada na chamada das diretivas `#include` e `#parse`, quando nestas são informados caminhos relativos como caminhos dos arquivos. Porém, esta informação só é sabida quando o *template* é carregado através do método `loadTemplateFile()`;

b) `TTemplateSlave`: classe derivada da `TTemplate`, que mantém um *template* derivado de outro. Este recurso é utilizado no processamento de constantes literais com aspas – a qual difere da constante literal com apóstrofe, pois seu conteúdo é analisado como um *template* –, na diretiva `#foreach` e na chamada de macros, para permitir processar trechos do *template* original, obtendo seu resultado e inserindo este no texto formatado de saída. O uso desta classe, em detrimento à `TTemplate`, nos casos mencionados anteriormente se justifica pelo fato destes utilizarem a lista de macros do *template* original;

c) `TTemplateMacro`: classe responsável por manter as informações referentes às macros dos *templates*. As informações da macro são: nome; lista de parâmetros; código fonte e posição inicial do código fonte, sendo esta última utilizada quando ocorrem erros no processamento do código fonte da macros – processada como um *template* isolado, da classe `TTemplateSlave` –, para ajustar a posição do erro ocorrido em relação ao *template* original. Instâncias desta classe são criadas no pré-processamento do *template* pela classe `TTemplatePreProcessor`, e são utilizadas na chamada de macros durante o processamento principal pela classe `TTemplateProcessor`;

d) `TTemplatePreProcessor`: classe derivada da `TBaseTemplateAnalyser`, que implementa o pré-processador de *templates*. Nela estão as ações do analisador semântico para o pré-processamento, as quais objetivam encontrar as macros em meio ao código fonte do *template* e guardar seus dados.

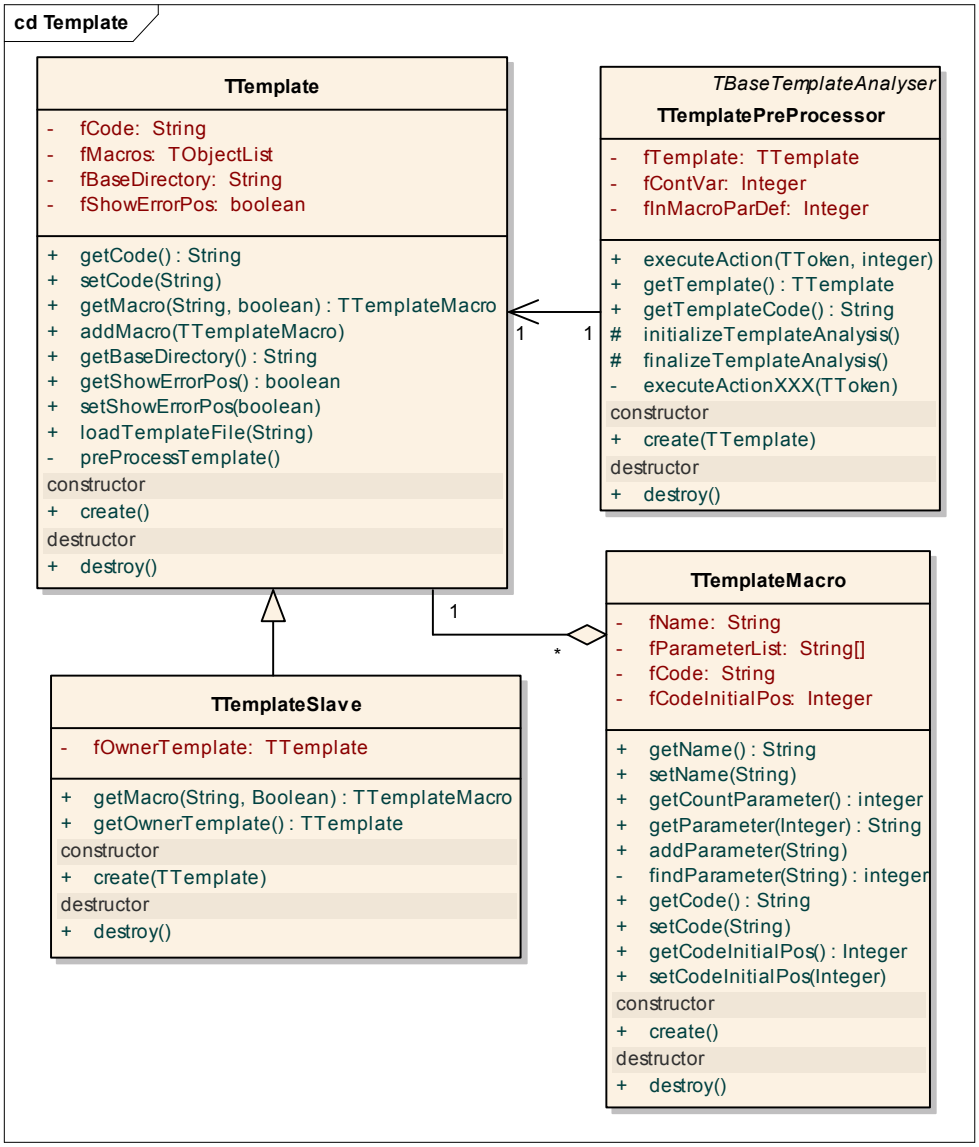


Figura 19 – Pacote `Template`

3.3.2.7 Pacote `Engine`

O sétimo e último pacote é denominado `Engine` e contém classes para controle do motor de *templates* (Figura 20). Fazem parte deste pacote as classes:

a) `TEngine`: classe que implementa a chamada de execução do motor de *templates*. É utilizada como interface entre o motor de *templates* e o desenvolvedor, ou nas execuções internas do motor de *templates* – no processamento de constantes literais com aspas, das diretivas `#foreach` e `#parse`, e da chamada de macros. Esta classe possui o método `merge()`, no qual é passado o *template* – classe `TTemplate` – e o contexto – interface `IContext` –, e é invocada a classe

`TTemplateProcessor` para realizar o processamento do *template*. O método `getResultText()` retorna o texto formatado, resultado do processamento do *template*. Já o método `isProcessStopped()` indica se no processamento do *template* foi executada a diretiva `#stop`, para que esta surta efeito também no processamento principal, quando a classe é utilizada internamente para tratar trechos de *template*;

b) `TTemplateProcessor`: classe derivada da `TBaseTemplateAnalyser`, que implementa o processador de *templates*. Nela estão as ações do analisador semântico para o processamento principal, as quais objetivam obter o texto formatado de saída.

Figura 20 – Pacote `Engine`

3.4 IMPLEMENTAÇÃO

A seguir são mostradas as técnicas e ferramentas utilizadas na implementação e detalhes sobre a implementação do motor de *templates*.

3.4.1 Técnicas e ferramentas utilizadas

A biblioteca foi implementada na linguagem Object Pascal, utilizando o ambiente de desenvolvimento Delphi 7. Os analisadores léxico e sintático foram gerados pelo GALS (GESSER, 2002) a partir da especificação apresentada no Apêndice A, sendo que o analisador semântico teve apenas sua estrutura básica gerada pela ferramenta.

Desde o início da implementação da biblioteca, a garantia de sua qualidade foi considerada primordial. Para tanto foi concebida uma ferramenta utilitária, denominada TestCase, que permite cadastrar cenários contendo um *template* e o resultado esperado deste quando submetido ao motor de *templates*, seja este resultado o texto formatado de saída ou um erro (Figura 21). Esta ferramenta permite a qualquer momento processar todos os cenários, onde para cada cenário o *template* é submetido ao motor de *templates* e o resultado obtido é comparado com o resultado esperado, sendo todas as inconsistências mostradas ao usuário. Novos recursos ou problemas na biblioteca geravam pelo menos um cenário de teste, e, a cada alteração na implementação, a ferramenta era executada, garantindo desta forma o funcionamento básico correto dos recursos e a não reincidência de erros.

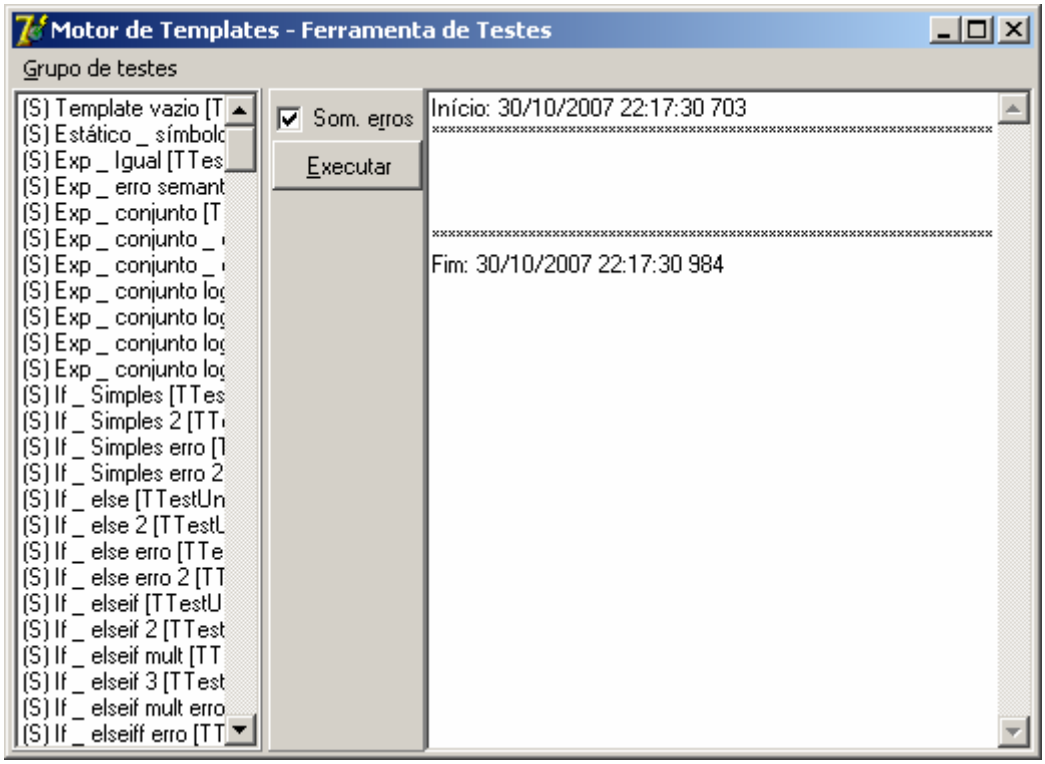


Figura 21 – Tela da ferramenta TestCase, alimentada com diversos cenários de teste

3.4.2 Implementação do motor de *templates*

A implementação da biblioteca partiu da especificação da linguagem de *templates* e da subseqüente geração, através do GALS, das classes `TTemplLexical`, `TTemplSyntactic`, `TTemplSemantic` e `TToken` disponíveis no pacote `TemplateAnalyser` e das classes de exceção disponíveis no pacote `Exception`. Duas alterações foram realizadas entre a versão gerada pelo GALS e a versão final: uma na classe `TTemplSyntactic`, onde foi criado o atributo `tokenAtual` publicando o valor do atributo privado `currentToken` e outra na classe `TTemplSemantic`, onde foi colocado o modificador `virtual` no método `executeAction()` para permitir que o mesmo seja sobrescrito.

Diante da existência de dois processamentos – um pré-processamento e outro processamento principal – na execução do motor de *templates*, houve a necessidade de implementar duas análises. Para tanto foi criada a classe `TBaseTemplateAnalyser`, como generalização da classe básica de analisador semântico `TTemplSemantic`, com o objetivo de centralizar os recursos básicos para o processamento da mesma. Esta classe possui um método público denominado `process` (Quadro 23) responsável por invocar os analisadores léxico

TTemplLexical e sintático TTemplSyntactic, e processar a análise usando a si mesmo como analisador semântico.

```
// Processa a análise
procedure TBaseTemplateAnalyser.process;
var
  lexical: TTemplLexical;
  syntactic: TTemplSyntactic;
  ...
begin
  lexical := TTemplLexical.create; // Cria a classe de análise léxica
  syntactic := TTemplSyntactic.create; // Cria a classe de análise sintática
  try
    lexical.setInput(getTemplateCode); // Carrega o código do template
    try
      syntactic.parse(lexical, self); // Executa as análises
    except
      ...
    end;
  finally
    lexical.free;
    syntactic.free;
  end;
end;
```

Quadro 23 – Método process() da classe TBaseTemplateAnalyser

Com o objetivo de implementar as ações semânticas do pré-processador e do processador principal foram criadas, respectivamente, as classes TTemplatePreProcessor (pacote Template) e TTemplateProcessor (pacote Engine) como generalizações da classe TBaseTemplateAnalyser.

As classes contidas no pacote Template foram criadas com o objetivo de guardar informações referentes ao *template*. A classe TTemplate é responsável por manter dados do *template*, sendo o código fonte diretamente informado pelo método setCode() ou carregado de um arquivo pelo método loadTemplateFile(), que internamente carrega o texto do arquivo e chama o método setCode(), além de alimentar o atributo fBaseDirectory. O método setCode() por sua vez, além de alimentar o atributo fCode invoca o método preProcessTemplate(), o qual cria uma instância da classe TTemplatePreProcessor recebendo o *template* como parâmetro e alimentando o atributo fShowErrorPos deste com o valor de seu atributo de mesmo nome.

A classe TTemplatePreProcessor trata do pré-processamento do *template*, sendo seu processo iniciado pelo método process() – implementado pela classe TBaseTemplateAnalyser – onde, através da análise, invoca as ações semânticas implementadas. O objetivo do pré-processamento é obter os dados referentes a todas as macros existentes no *template*, sendo que, para cada macro encontrada durante este processo, são criadas instâncias da classe TTemplateMacro e estas inseridas na lista de macros do *template* – atributo fMacros da classe TTemplate – através do método addMacro(). O

Quadro 24 apresenta a implementação da ação semântica responsável por criar e inserir uma macro reconhecida no *template*.

```
// Fim da diretiva macro
procedure TTemplatePreProcessor.executeAction904(const token: TToken);
var
  posInitial, posFinal: Integer;
  macro: TTemplateMacro;
  params: TStringArray;
  i: integer;
  contPar: integer;
begin
  // Cria a macro
  macro := TTemplateMacro.create;
  try
    contPar := fMemStack.pop; // Recupera a quantidade de parâmetros [#906]
    if (contPar > 0) then // Se foram encontrados parâmetros [#903, #907]
    begin
      // Carrega os parâmetros em uma estrutura intermediária (coloca na ordem)
      SetLength(params, contPar);
      for i := contPar - 1 downto 0 do // Percorre a quantidade de parâmetros
        params[i] := fMemStack.pop; // Retorna o nome do parâmetro [#907]

      // Adiciona os parâmetros na macro
      for i := 0 to contPar - 1 do // Percorre a quantidade de parâmetros
        macro.addParameter(params[i]);
    end;

    macro.setName(fMemStack.pop); // Informa o nome da macro [#905]

    posInitial := fPosTagStack.pop; // Busca a posição de início do código [#906]
    findPosNextNewLineOrValidChar(posInitial, False);
    posFinal := token.getPosition;
    findPosPreviousNewLineOrValidChar(posFinal, False);

    macro.setCodeInitialPos(posInitial);
    macro.setCode(getTemplateCodeFraction(posInitial, posFinal));

    getTemplate.addMacro(macro); // Adiciona a macro ao template
  except
    macro.free;
    raise;
  end;
end;
```

Quadro 24 – Ação semântica `#904` responsável por guardar as macros reconhecidas do *template*

A Figura 22 apresenta a interação entre as classes para a realização do pré-processamento do *template*, onde o desenvolvedor cria uma instância da classe `TTemplate` e invoca o método `loadTemplateFile()` para que haja a leitura do arquivo de *template*. Em seguida é executado o método `preProcessTemplate()`, que por sua vez cria uma instância da classe `TTemplatePreProcessor`. A partir do método `process()` desta classe, o código do *template* é analisado e para cada definição de macro encontrada uma instância da classe `TTemplateMacro` é criada, alimentada e adicionada à lista de macros do *template* através do método `addMacro()` da classe `TTemplate`.

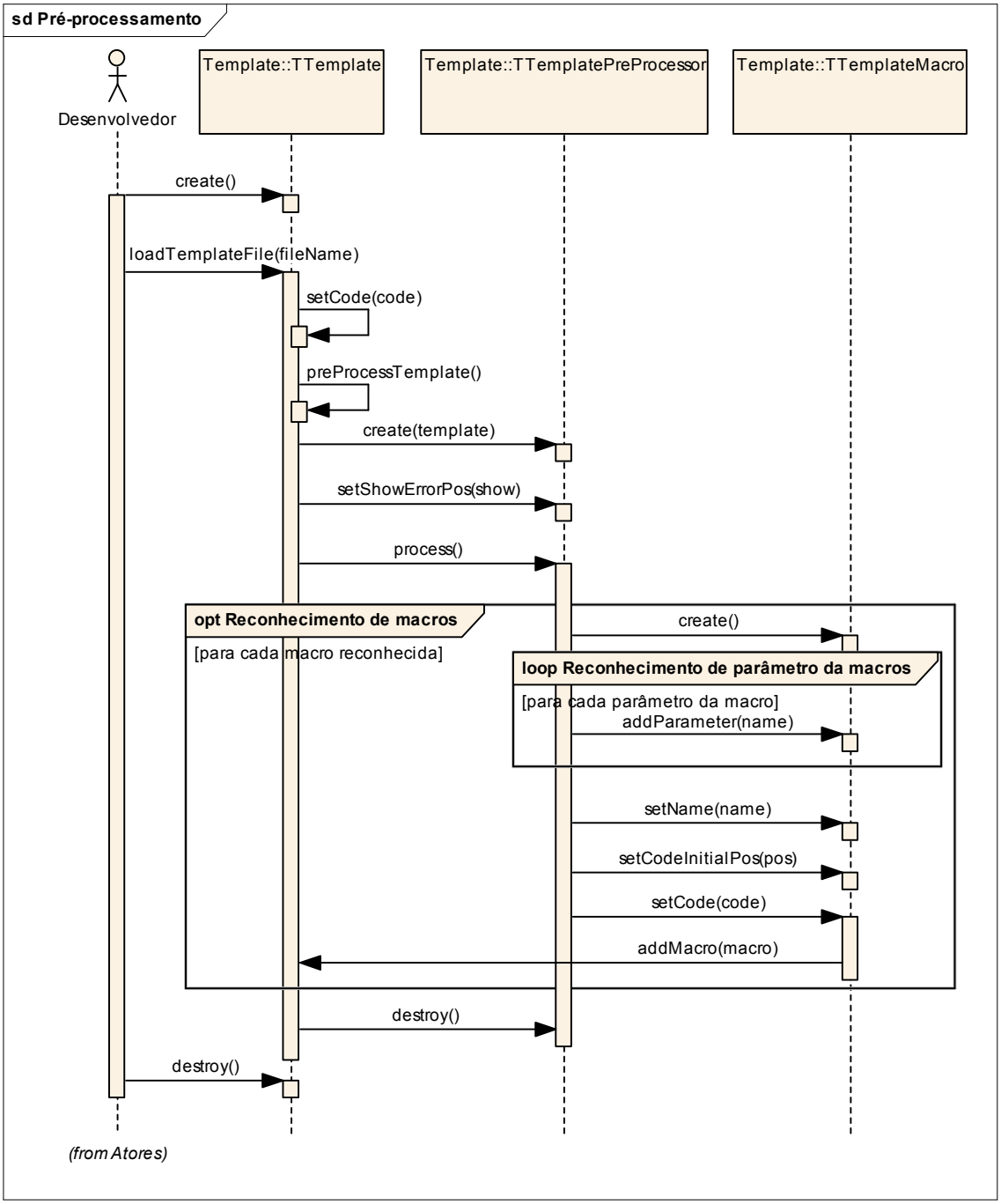


Figura 22 – Diagrama de seqüência do pré-processamento

No pacote `Context` estão implementados recursos referentes ao contexto do motor de *templates*. A classe `TContext` faz a implementação do contexto, sendo que possui o atributo `fItems`, cujo objetivo é manter a lista de identificadores ligados ao mesmo. A adição de identificadores ao contexto é feita pelos métodos `put()`, que possui sobrecarga para aceitar todos os tipos de elementos. O retorno de um elemento é feito pelo método `get()`, enquanto a exclusão é feita pelo método `remove()`, sendo que em todos estes métodos é informada a chave do identificador, que é única e identifica o mesmo. Cada elemento do contexto é na verdade uma instância da classe `TContextItem` que somente possui atributos e métodos para

manipular a chave e o identificador.

Todo identificador mapeado ao contexto é implementação da interface `IIdentifier`, sendo também utilizado para guardar resultados de construções, como valores de constantes, operações aritméticas, propriedades, entre outros, durante a análise semântica. Da implementação básica dos identificadores feita pela classe `TIdentifier` derivam as implementações específicas para cada tipo de identificador: `TIdentifierNone` para sem tipo; `TIdentifierNumber` para numérico; `TIdentifierString` para alfanumérico; `TIdentifierBoolean` para booleano; `TIdentifierArray` para vetorial e `TIdentifierObject` para identificador de objeto. Cada uma destas implementações traz métodos específicos para tratar o seu tipo de conteúdo, como por exemplo os métodos `getValue()` e `setValue()` presentes nas implementações numéricas, alfanuméricas, booleanas e vetoriais, que, respectivamente, retornam e alteram o valor do identificador – ou de um elemento do identificador, no caso do vetorial.

Na implementação de identificador vetorial há ainda um tratamento para os métodos `hasIterator()` e `getIterator()`, permitindo o uso deste na diretiva `#foreach`, para viabilizar a iteração entre seus elementos através de uma implementação da interface `IIterator` feita pela classe `TIdentifierArrayIterator`. Esta possui, além de uma referência para o identificador, um atributo chamado `fIteratorLastIndex`, que indica o índice do último elemento retornado, sendo que a cada chamada do método `next()` retorna-se o elemento do vetor cujo índice seja igual ao valor do atributo e incrementa este em um.

Já a implementação do identificador de objeto é a mais diferenciada, pois a classe `TIdentifierObject` é abstrata e introduz seis novos métodos, a fim de que classes derivadas sejam contêineres de objetos, permitindo que os mesmos sejam inseridos como parte do contexto. Dentre os novos métodos, destacam-se `hasProperty()` e `getPropertyValue()`, que permitem definir e implementar propriedades do identificador de objeto para serem utilizadas no *template*, e `hasMethod()` e `executeMethod()` que fazem o mesmo para métodos. A implementação nativa dos métodos descritos disponibiliza a propriedade `className`, que retorna o nome da classe do objeto encapsulado, e ainda trata o recurso RTTI do Delphi, onde atributos contidos na seção `published` da classe encapsulada são automaticamente disponibilizados para o *template* como propriedades do identificador de objeto(Quadro 25). Diferentemente do Velocity, as propriedades e métodos dos objetos – à exceção do caso descrito anteriormente – não são automaticamente disponibilizados para o *template*, pois a linguagem Delphi não possui recursos que permitam a reflexão, onde todas as

características das classes são disponibilizadas para verificação em tempo de execução.

```
// Retorna se o objeto possui determinada propriedade
function TIdentifierObject.hasProperty(name: String): boolean;
var
  propInfo: PPropInfo;
begin
  if (SameText(name, 'className')) then
    result := True
  else
  begin
    // Utiliza RTTI para verificar se existe uma propriedade no objeto
    propInfo := GetPropInfo(getObject, name);
    result := ((propInfo <> nil) and (propInfo^.GetProc <> nil));
  end;
end;

// Busca o valor de determinada propriedade do objeto
function TIdentifierObject.getPropertyValue(name: String): IIdentifier;
var
  propInfo: PPropInfo;
  value: Variant;
begin
  if (SameText(name, 'className')) then
    result :=
TContextManager.getInstance.getNewIdentifierString(getObject.ClassName)
  else
  begin
    result := nil;

    // Utiliza RTTI para buscar o valor de uma propriedade no objeto
    propInfo := GetPropInfo(getObject, name);
    if ((propInfo <> nil) and (propInfo^.GetProc <> nil))
    begin
      value := getPropValue(getObject, name, False);
      result := TContextManager.getInstance.getNewIdentifier(value);
    end;
  end;
end;
```

Quadro 25 – Implementação dos métodos `hasProperty()` e `getPropertyValue()` da classe `TIdentifierObject`

O contexto do motor de *templates* é obtido através do método `getNewContext()` de uma instância da classe `TContextManager`. Esta por sua vez possui implementação de instância única, retornada pelo seu método de classe `getInstance()`. Instâncias de identificadores também são criadas através de métodos desta classe, sendo eles: `getNewIdentifier()`, que retorna um identificador cujo tipo é definido conforme o valor informado; `getNewIdentifierNone()`, que retorna o identificador sem tipo; `getNewIdentifierNumber()`, que retorna um identificador numérico; `getNewIdentifierString()`, que retorna um identificador alfanumérico; `getNewIdentifierBoolean()`, que retorna um identificador booleano; `getNewIdentifierArray()`, que retorna um identificador vetorial e `getNewIdentifierObject()`, que retorna um identificador de objeto.

Porém, a classe `TContextManager` também é responsável por controlar o mapeamento de classes de objetos com suas classes contêineres. Para tanto, ela conta com métodos como

`addIdentifierClassMapping()` e `removeIdentifierClassMapping()` que adicionam e removem mapeamentos, os quais são mantidos no atributo `fIdentifierClassMapper` através de várias instâncias do registros `TRecIdentifierClassMapping`.

As classes contêineres `TIdentifierListContainer` e `TIdentifierStringListContainer` estão implementadas nativamente no pacote `Container` da biblioteca, sendo que elas encapsulam as classes `TList` e `TStringList`, respectivamente. A implementação de ambas conta com uma instância do objeto, o qual se deseja inserir no contexto, e disponibiliza propriedades e métodos, além da iteração entre seus elementos. Esta iteração é feita por instâncias, retornadas pelo método `getIterator()`, das classes `TIdentifierListIterator` e `TIdentifierStringListIterator`, as quais possuem funcionamento similar à classe de iteração do identificador vetorial, porém com a referência ao contêiner. Para o contêiner `TIdentifierListContainer` está disponível a propriedade `count`, que retorna a quantidade de elementos, e o método `getItem()` com um parâmetro numérico, que retorna o elemento cujo índice seja igual ao valor do parâmetro. Já no contêiner `TIdentifierStringListContainer` está disponível a propriedade `count`, que retorna a quantidade de linhas, e o método `getLine()` com um parâmetro numérico, que retorna a linha cujo índice seja igual ao valor do parâmetro.

A classe `TEngine` faz a interface do motor de *templates* com o código fonte do desenvolvedor, pois quando invocado seu método `merge()`, passando um *template* e um contexto, a classe `TTemplateProcessor` é instanciada e a rotina é executada (Quadro 26). A Figura 23 apresenta um diagrama de seqüência do processamento de *template*. Após a execução, seu resultado pode ser obtido pelo método `getResultText()` ou salvo em um arquivo pelo método `saveResultTextToFile()`. Antes da execução do motor de *templates*, pode-se alterar o atributo `fShowErrorPos`, cujo valor padrão é verdadeiro, que indica na ocorrência de alguma exceção na análise léxica, sintática ou semântica, se a linha e a coluna do *template* onde este ocorreu devem ser concatenadas à mensagem. Este último recurso é utilizado internamente, quando uma análise auxiliar é necessária, para que, na ocorrência de uma exceção, a mensagem não contenha a posição do erro, pois esta posição é inválida, sendo que nestes casos a posição do erro é calculada em relação à posição de ambos os processamentos. Já após a execução, pode-se invocar o método `isProcessStopped()`, que indica se o processamento foi interrompido pelo uso da diretiva `#stop`. Este recurso também é utilizado internamente, para o caso onde é executada a diretiva `#stop` em uma análise auxiliar, pois a parada deve influenciar também a análise principal.

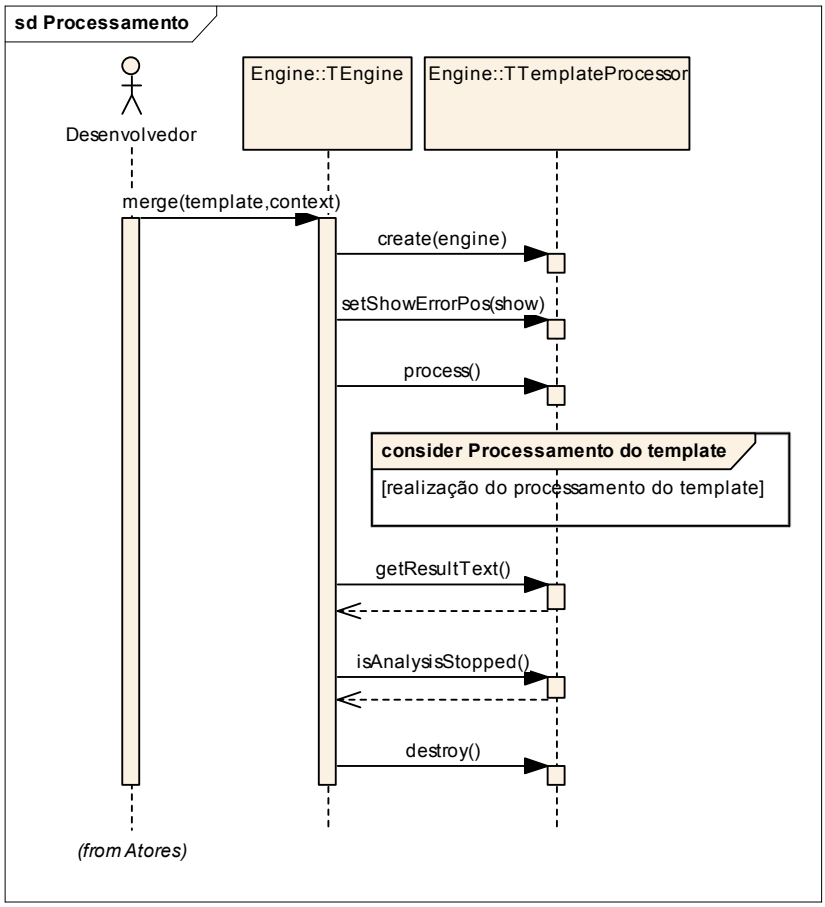


Figura 23 – Diagrama de seqüência do processamento

```
// Processa a consolidação do template com o contexto
procedure TEngine.merge(template: TTemplate; context: IContext);
var
  processor: TTemplateProcessor;
begin
  fResultText := '';
  fProcessStopped := False;
  processor := TTemplateProcessor.Create(self);
  try
    fTemplate := template;
    fContext := context;

    processor.setShowErrorPos(getShowErrorPos);
    processor.process;
    fResultText := processor.getResultText;
    fProcessStopped := processor.isAnalysisStopped;
  finally
    processor.free;
    fTemplate := nil;
    fContext := nil;
  end;
end;
```

Quadro 26 – Método `merge()` da classe `TEngine`

No pacote `Engine` está alocada a classe responsável pelo processamento do motor de *templates*, denominada `TTemplateProcessor`, que deriva da `TBaseTemplateAnalyser` e, assim como a classe de pré-processamento, é quem implementa as ações semânticas, sendo nesta tratadas todas as construções disponíveis na linguagem de *templates*, à exceção da

definição de macros que é reconhecida durante o pré-processamento. O acesso ao *template* e ao contexto é provido pela instância da classe `TEngine` que invocou o processamento, a qual está disponível através do atributo `fEngine`.

Os recursos são tratados de forma individual na análise semântica, cada um possuindo controles próprios que garantem o funcionamento do mesmo, como, por exemplo, a diretiva `#if` que possui como um controle o atributo privado `fInIf`, o qual indica se o código sendo analisado está dentro da diretiva. Assim, a análise dos demais recursos passa a estar condicionada ao fato de se estar dentro do bloco aceito da diretiva `#if`. Recurso semelhante a este existe também para as diretivas `#foreach` e `#macro`.

A implementação das diretivas `#foreach` e `#parse`, da chamada de `#macro` e do reconhecimento de constantes literais com aspas, utilizam uma execução auxiliar do motor de *templates*, sendo para esta criado um *template* auxiliar – classe `TTemplateSlave` – que, juntamente com o contexto da execução principal, é usado na invocação do método `merge()` de uma nova instância de motor de *templates* – classe `TEngine`. O Quadro 27 apresenta o uso do motor de *templates* auxiliar no reconhecimento de constantes literais com aspas. Mostra também o tratamento de erros de análise do motor de *templates*, onde inicialmente é chamado o método `setShowErrorPos()` do motor e do *template* auxiliar, a fim de que estes não mais concatenem a posição do erro à sua mensagem, e por final existe um bloco de tratamento de exceções, o qual ajusta a posição do erro através de dados contidos nas análises principal e auxiliar.

```
var
  value: String;
  engine: TEngine;
  template: TTemplate;
  ...
begin
  ...
  // Processa o script da constante literal
  engine := nil;
  template := nil;
  try
    engine := TEngine.create;
    template := TTemplateSlave.create(getEngine.getTemplate);
    engine.setShowErrorPos(False);
    template.setShowErrorPos(False);
    try
      template.setCode(value);
      engine.merge(template, getEngine.getContext);
    except
      on e: EAnalysisError do
        raise EAnalysisErrorClass(e.ClassType).create(e.getMessage,
e.getPosition + token.getPosition);
      else
        raise;
    end;
    value := engine.getResultText;
  finally
    template.free;
    engine.free;
  end;
  ...
end;
```

Quadro 27 – Trecho da ação semântica `#012`, onde o conteúdo de uma constante literal com aspas é submetido a uma análise auxiliar

Além de todos os recursos mencionados existentes no processamento do motor de *templates* há o reconhecimento do código estático e sua transcrição para o texto formatado de saída, sendo este implementado diretamente pelas ações semânticas `#003` e `#004`, que identificam, respectivamente, o início e o fim do texto estático, sendo na primeira guardada a posição inicial e na segunda, então de posse da posição final, copiado o texto contido entre estas posições. Porém a implementação também deve considerar os caracteres de formatação (espaço, tabulação e quebra de linha), ignorados pelo analisador léxico, como parte do código estático.

O reconhecimento da formatação existente entre o código dinâmico e o código estático, e vice-versa, requereu atenção durante o desenvolvimento. Além do fato de não ser reconhecida pelo analisador léxico e não ter *tokens* associados a si, também possui regras diferenciadas de aceitação como parte do código estático, já que nem todas as ocorrências desses caracteres de formatação devem ser consideradas como parte deste tipo de código. Esta abordagem garante uma maior flexibilidade – pois se pode obter mais resultados em relação à outra abordagem – e clareza – pois para obter o resultado esperado não é necessário concatenar código estático e dinâmico na mesma linha – ao *template*. No Quadro 28 é

apresentado um exemplo de *template* que possui caracteres de formatação, no Quadro 29 é apresentado o resultado de seu processamento caso todos esses caracteres sejam considerados como código estático, e no Quadro 30 é apresentado o resultado real quando o *template* é submetido ao processamento, onde alguns caracteres de formatação são ignorados.

```
ini
  #set ($var = "texto")
  $var
fim
```

Quadro 28 – Exemplo de *template* contendo caracteres de formatação entre código estático e dinâmico

```
ini

  texto
fim
```

Quadro 29 – Resultado do processamento do *template* considerando todos os caracteres de formatação como código estático

```
ini
  texto
fim
```

Quadro 30 – Resultado real do processamento do *template*, onde nem todos os caracteres de formatação são considerados código estático

3.5 ESTUDOS DE CASO

Neste capítulo são apresentados dois estudos de casos implementados utilizando o motor de *templates*.

3.5.1 Gerador de código fonte

Como um estudo de caso simples para o motor de *templates* foi implementado um gerador de código fonte que lê um arquivo texto, contendo informações de classes e atributos, e gera arquivos fonte na linguagem selecionada.

Este aplicativo recebe como entradas um arquivo texto de dados, a linguagem destino e o diretório onde os arquivos fonte devem ser gerados (Figura 24). O arquivo de entrada possui uma formatação pré-definida, onde cada linha possui o nome da classe e o nome dos atributos da mesma (Quadro 31).

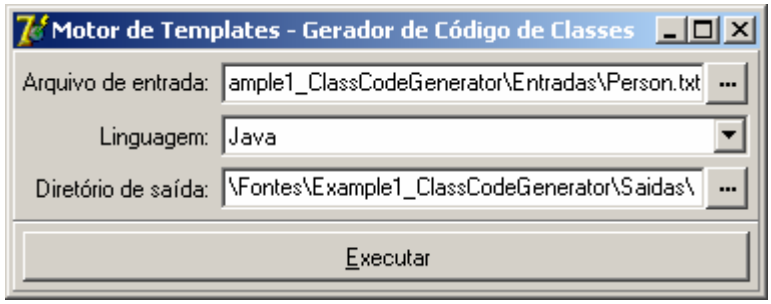


Figura 24 – Tela do estudo de caso de gerador de código de classes

```
Person=first,middle,last
```

Quadro 31 – Exemplo de arquivo de entrada do gerador de código de classes

Ao pressionar o botão `Executar`, o arquivo de entrada é lido e, para cada linha deste, o motor de *templates* é invocado, sendo passado o *template* correspondente à linguagem selecionada (Quadro 32) – disponível em um diretório padrão e com um nome padrão – e um contexto contendo o nome da classe e um vetor com o nome dos atributos – mapeados para os identificadores `class_name` e `fields`, respectivamente –, sendo assim gerado um arquivo fonte com o resultado do processamento (Quadro 33).

```
public class ${class_name}Base {

  #foreach ($field in $fields)
  protected String _$field;
  #end

  public ${class_name}Base() {
    #foreach ($field in $fields)
    _$field = new String();
    #end
  }

  #foreach ($field in $fields)
  public String get${field}() {
    return _$field;
  }

  public void set${field}(String value) {
    _$field = value;
  }

  #end
}
```

Quadro 32 – *Template* do gerador de código de classes para a linguagem Java

```
public class PersonBase {

  protected String _first;
  protected String _middle;
  protected String _last;

  public PersonBase() {
    _first = new String();
    _middle = new String();
    _last = new String();
  }

  public String getfirst() {
    return _first;
  }

  public void setfirst(String value) {
    _first = value;
  }

  public String getmiddle() {
    return _middle;
  }

  public void setmiddle(String value) {
    _middle = value;
  }

  public String getlast() {
    return _last;
  }

  public void setlast(String value) {
    _last = value;
  }

}
```

Quadro 33 – Arquivo fonte gerado pelo gerador de código de classes

O Quadro 34 apresenta o trecho de código do estudo de caso em que há a invocação do motor de *templates*. Nele as linhas de código demarcadas com `(1)` tratam da leitura do arquivo de *templates*, o qual deve ser especificado pelo desenvolvedor da aplicação conforme definido pelo caso de uso `UC02 Elaborar templates`; as linhas de código demarcadas com `(2)` tratam da criação e alimentação do contexto, definido no caso de uso `UC01 Definir contexto`; e as linhas de código demarcadas com `(3)` tratam do processamento do *template* e geração do resultado, definido no caso de uso `UC03 Gerar documento de saída`.

```
var
  ...
  template: TTemplate;
  context: IContext;
  engine: TEngine;
begin
  ...
  engine := nil;
  template := nil;
  ...
  try
    engine := TEngine.Create;
    template := TTemplate.Create;
    ...
    (1)
    template.loadTemplateFile(templateFile); // Carrega o template
    ...
    (2)
    context := TContextManager.getInstance.getNewContext; // Cria o contexto
    context.put('class_name', className); // Informa o nome da classe ao contexto
    context.put('fields', fieldsArray); // Informa os campos ao contexto
    (3)
    engine.merge(template, context); // Processa o arquivo
    ...
  finally
    ...
    template.Free;
    engine.Free;
  end;
  ...
end;
```

Quadro 34 – Trecho de código que invoca o motor de *templates*

3.5.2 Catálogo de telefones

Um estudo de caso mais complexo foi implementado através da elaboração de um catálogo de telefones. Este aplicativo não tem por objetivo controlar o catálogo, através da inclusão, alteração ou exclusão de registros, mas sim apresentar pessoas e telefones catalogados e permitir a geração de documentos, como cartas e outros, com estes dados.

O diagrama de classes elaborado para o estudo de caso (Figura 25) conta com uma enumeração e três classes de negócio, além de três classes que representam os contêineres das classes de negócio e uma classe que implementa a iteração entre elementos de um destes. As classes do estudo de caso são:

a) `TPhoneKind`: enumeração que indica os tipos de telefones que podem existir. Esta enumeração é utilizada na classe `TPhone` para classificar o tipo do telefone. Pode assumir valores que indicam: nenhum – cuja função é indicar que o telefone não foi classificado quanto ao tipo –, residencial, comercial e celular;

b) `TBook`: classe de negócio que representa o catálogo de pessoas e telefones. Possui o atributo privado `fPersons`, o qual trata-se de uma lista de pessoas, instâncias da classe `TPerson`;

c) `TPerson`: classe de negócio que representa uma pessoa no catálogo. Possui como atributos o nome da pessoa e a lista de telefones desta, instâncias da classe `TPhone`;

d) `TPhone`: classe de negócio que representa o telefone de alguma pessoa do catálogo. Possui como atributos: tipo do telefone – utilizando a enumeração `TPhoneKind`, número do telefone, ramal e um indicativo se é o telefone preferencial da pessoa. Possui também dois métodos de classe, que retornam a descrição de um tipo de telefone e o tipo de telefone relativo a uma descrição;

e) `TBookContainer`: classe derivada de `TIdentifierObject` que implementa o contêiner da `TBook` para o motor de *templates*, disponibilizando propriedades e métodos para serem utilizados no *template*. Este contêiner ainda disponibiliza a iteração entre pessoas – classe `TPerson` – do catálogo – classe `TBook` –, a qual é feita por uma instância da classe `TBookContainerIterator`;

f) `TPersonContainer`: classe derivada de `TIdentifierObject` que implementa o contêiner da `TPerson` para o motor de *templates*, disponibilizando propriedades e métodos para serem utilizados no *template*;

g) `TPhoneContainer`: classe derivada de `TIdentifierObject` que implementa o contêiner da `TPhone` para o motor de *templates*, disponibilizando propriedades para serem utilizadas no *template*;

h) `TBookContainerIterator`: classe que implementa a interface `IIterator`, e disponibiliza a iteração entre pessoas – instâncias de `TPerson` – existentes no catálogo – instância de `TBook`. É disponibilizada pelo método `getIterator()` da classe `TPhoneContainer`.

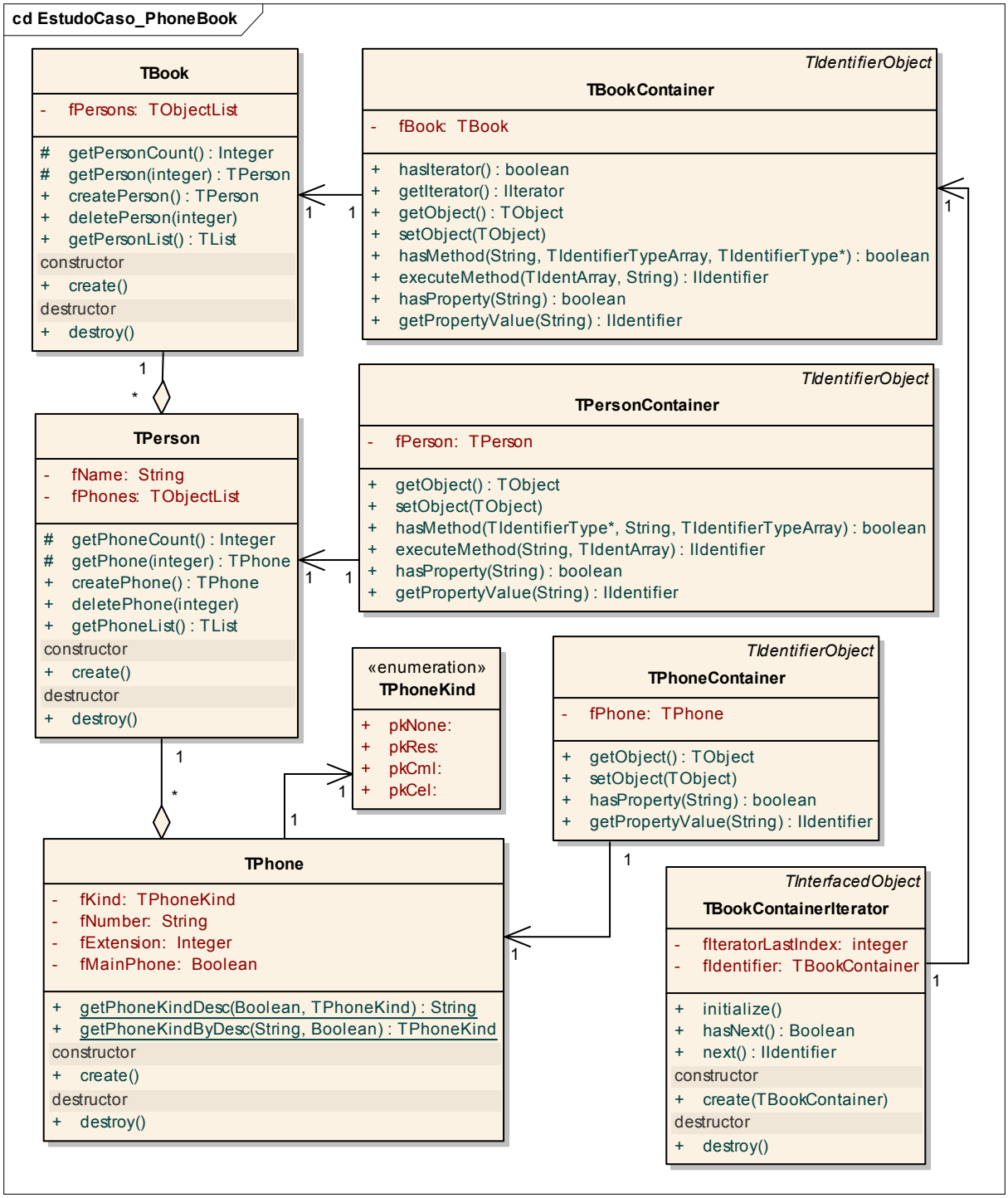


Figura 25 – Diagrama de classes do estudo de caso de catálogo de telefones

Na inicialização do aplicativo, o catálogo de telefones é carregado de um arquivo texto pré-formatado – disponível em um diretório e com um nome pré-definido. A formatação deste arquivo obedece à regra de uma linha com dados da pessoa – quantidade de telefones e nome da pessoa – separados por vírgula e as linhas seguintes – conforme a quantidade de telefones da pessoa – contendo os dados do telefone – tipo, número, ramal e indicativo de preferencial – separados por vírgula, depois podendo haver mais dados de pessoas ou o fim do arquivo

(Quadro 35).

```
3,"Thiago Demarchi"
"Res","(47) 3333-3333",0,0
"Cml","(47) 3221-3300",406,0
"Cel","(47) 9999-1111",0,1
2,"Silva"
"Res","(21) 1236-1458",0,0
"Cel","(21) 9999-2345",0,1
0,"Zezinho"
1,"Teresinha"
"Res","(14) 9876-5443",0,0
```

Quadro 35 – Exemplo de arquivo de entrada do catálogo de telefones

Os dados do catálogo, que contém as pessoas, que por sua vez contém os telefones, são armazenados respectivamente em instâncias das classes `TBook`, `TPerson` e `TPhone`. Estes dados são carregados visualmente na interface da aplicação com o usuário (Figura 26) em grades, permitindo ao usuário consultá-los e selecioná-los.

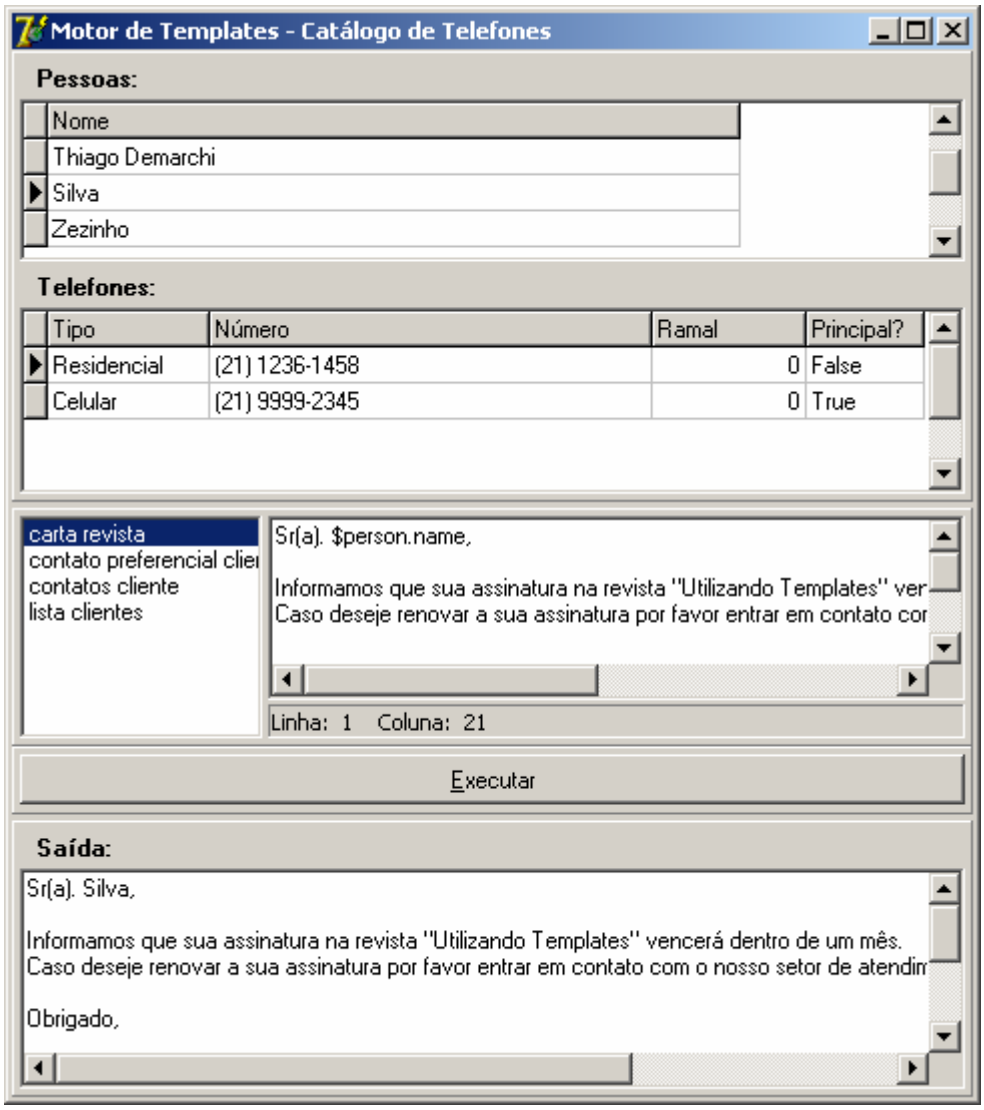


Figura 26 – Tela do estudo de caso de catálogo de telefones

É permitido ao usuário selecionar um dos *templates* disponíveis – alocados em um diretório padrão – e com base neste e nos dados do catálogo gerar o texto formatado de saída.

Para o contexto utilizado pelo motor de *templates*, são mapeadas as seguintes informações: o objeto de catálogo – instância de `TBook` – através da chave `book`; o objeto de pessoa selecionado na grade – instância de `TPerson` – através da chave `person`; o objeto de telefone selecionado na grade – instância de `TPhone` – através da chave `phone` e a lista de telefones da pessoa selecionada na grade através da chave `persons`.

No Quadro 36 é apresentado um exemplo de *template* que tem por finalidade mostrar a listagem dos dados de todos os telefones da pessoa selecionada. Para tanto é usada a diretiva `#foreach` iterando sobre o identificador de nome `phones`, que contém a listagem de telefones da pessoa selecionada. Para imprimir o nome da pessoa e os dados dos telefones, foi utilizado o recurso de propriedades, sendo o valor dos mesmos obtidos através do recurso RTTI do Delphi – atributos `published` na definição da classe – à exceção da propriedade `kind` para objetos da classe `TPhone`, que retorna o tipo do telefone, o qual, apesar de também estar disponível como as demais propriedades e retornando valores inteiros – que representam o índice do valor na enumeração `TPhoneKind` – foi tratado na classe `TPhoneContainer` para retornar a descrição do mesmo. No Quadro 37 é apresentado o texto formatado de saída gerado pelo motor de *templates* com base no *template* anterior.

```
Listagem dos contatos do cliente:

Nome: $person.name

Telefone            |   Ramal    |   Preferencial    |   Tipo
-------------------------------------------------------------------------------
#foreach ($phone in $phones)
$phone.number|   $phone.extension            |  #if ($phone.mainPhone == true) Sim
#else Não #end            |   $phone.kind
#end
```

Quadro 36 – Exemplo *template* da aplicação de catálogo de telefones

```
Listagem dos contatos do cliente:

Nome: Silva

Telefone            |   Ramal    |   Preferencial    |   Tipo
-------------------------------------------------------------------------------
(21) 1236-1458      |   0        |   Não             |   Residencial
(21) 9999-2345      |   0        |   Sim             |   Celular
```

Quadro 37 – Saída do motor de *templates* para o *template* do Quadro 36

3.6 RESULTADOS E DISCUSSÃO

A biblioteca implementada atende aos requisitos propostos, disponibilizando os

recursos de um motor de *templates* para ser utilizado em aplicações desenvolvidas em Delphi 7, sem a necessidade da utilização de bibliotecas de terceiros, sendo a única entre as estudadas a combinar estas características. Esta possui uma linguagem de *templates* similar à VTL do Velocity, sendo simples e disponibilizando recursos declaração de variáveis, controle de fluxo e reuso de código.

No Quadro 38 é apresentado um comparativo entre a biblioteca implementada e os trabalhos correlatos apresentados.

CARACTERÍSTICA	**Smarty**	**FastTrac**	**Velocity**	**Motor de *templates* para Delphi 7**
linguagem para uso da biblioteca	PHP	Delphi 6	Java	Delphi 7
utilização de bibliotecas terceiras para o funcionamento do motor de *templates*	não	sim	não	não
linguagem de *templates* simples	não	não	sim	sim

Quadro 38 – Comparativo entre os motores de *templates*

4 CONCLUSÕES

A partir da percepção das vantagens da utilização de um motor de *templates* em aplicações que necessitam gerar documentos e da quantidade despendida de trabalho para implementar um motor de *templates* específico para determinada aplicação, bem como das vantagens e facilidades da utilização de um motor de *templates* genérico e da não existência de um destes, disponível para aplicações programadas em Delphi, que traga um bom custo-benefício, foi verificada a possibilidade da implementação deste trabalho.

A biblioteca implementada interpreta arquivos de *templates* através de analisadores léxico, sintático e semântico, sendo os dois primeiros gerados com a ferramenta GALS (GESSER, 2002), o que reduziu significantemente o trabalho de implementação dos mesmos. A análise léxica separa os *tokens* usados na análise sintática para a identificação do código estático e das estruturações do código dinâmico, que permitem a manipulação de variáveis, o controle de fluxo e a reutilização de código. A análise semântica, por sua vez, processa as ações definidas pelas estruturas do código dinâmico, substituindo neste os dados definidos no contexto e mescla o resultado ao código estático, formando assim o texto formatado de saída.

As informações disponíveis para o uso nas estruturas do código dinâmico são definidas pela aplicação que utiliza a biblioteca, a qual mapeia seus dados em um contexto através de um apelido único, que deve ser utilizado no *template* para referenciar o dado. Para os casos onde um objeto é mapeado no contexto, a biblioteca possibilita a sua extensão através da criação e registro de classes contêineres, a fim de disponibilizar para tais objetos, propriedades e métodos a serem utilizadas nos *templates*, já que a linguagem Delphi somente possibilita que, em tempo de execução, tenha-se conhecimento de algumas propriedades específicas dos objetos, as quais são disponibilizadas automaticamente para o *template*.

Por fim, obteve-se uma biblioteca que pode trazer agilidade na programação e flexibilidade para aplicativos que necessitem gerar documentos de saída, o que é comum em trabalhos curriculares e finais de cursos na área de computação, além de ser passível de utilização em aplicações comerciais.

4.1 EXTENSÕES

Durante o desenvolvimento do trabalho foram verificadas diversas possíveis melhorias na biblioteca, muitas das quais não puderam ser contempladas neste trabalho. Portanto, as sugestões para extensão do trabalho são mostradas através de uma lista de tarefas, ou do inglês *to-do list*. A lista de tarefas é apresentada no Quadro 39, onde há a descrição de cada tarefa e sua complexidade – apresentada em escala crescente de complexidade, de um a dez.

TAREFA	COMPLEXIDADE
Criação de um editor gráfico de *templates*.	10
Possibilidade de invocação de propriedades e métodos para qualquer tipo de identificador. Criação de métodos para identificadores alfanuméricos para: retornar parte da cadeia de caracteres; converter para numérico; converter primeira letra para maiúsculo. Criação de métodos para identificadores numéricos para: retornar o número formatado, inclusive como data; arredondar valor. Criação de métodos para identificadores booleanos para: converter para alfanumérico ou numérico.	5
Possibilitar a passagem de parâmetros para macros sempre como referência. Atualmente quando um valor de tipo diferente é atribuído dentro da macro para o parâmetro, é criada uma nova referência para o valor, mantendo o valor original na origem da invocação da macro.	8
Implementação de contêineres para diversas classes nativas do Delphi.	1
Implementação de prioridade no mapeamento entre classe e classe contêiner, pois atualmente se dois contêineres são mapeados para duas classes, uma derivada da outra, pode haver conflito ao selecionar qual contêiner deve tratar cada classe.	3
Implementação da diretiva `#break` para terminar abruptamente o processamento da diretiva de controle de fluxo por repetição (`#foreach`).	4
Implementação da diretiva `#continue` para terminar abruptamente o processamento do laço atual da diretiva de controle de fluxo por repetição (`#foreach`), partindo para o processamento do próximo laço.	4

Quadro 39 – Lista de tarefas da biblioteca

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.

APACHE SOFTWARE FOUNDATION. **The Apache Velocity project**. [S.l.], [2004?]. Disponível em: <http://velocity.apache.org>. Acesso em: 05 jan. 2007.

BARRETO, Luciano P. **Construção de compiladores**. [Salvador], [2005?]. Disponível em: <http://www.lasid.ufba.br/graduacao/compiladores/>. Acesso em: 30 ago. 2007.

BARRICO, Carlos. **Linguagens e processadores**. [S.l.], out. 2002. Disponível em: <www.di.ubi.pt/~cbarrico/Disciplinas/Compiladores/Downloads/Capitulo1.pdf>. Acesso em: 09 abr. 2007.

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

CRUZ, Sérgio A. B.; MOURA, Maria F. **Formatação de dados usando a ferramenta Velocity**. Campinas: Embrapa, 2002. Disponível em: <http://www.cnptia.embrapa.br/modules/tinycontent3/content/2002/comuntec20.pdf >. Acesso em: 10 jan. 2007.

FEITOSA, Ciro. **Smarty e PHP**: tudo a ver. [S.l.], 2006. Disponível em: <http://cirofeitosa.com.br/post/smarty-e-php-tudo-a-ver>. Acesso em: 05 fev. 2007.

GARCIA, Rogério E. **Compiladores**. São Paulo, mar. 2007. Disponível em: <http://www4.fct.unesp.br/rogerio/Compiladores/Compiladores_03.pdf>. Acesso em: 01 set. 2007.

GESSER, Carlos E. **GALS**: gerador de analisadores léxicos e sintáticos. Florianópolis, [2002?]. Disponível em: <http://gals.sourceforge.net/>. Acesso em: 16 out. 2007.

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

HILL, Doug. **FastTrac template engine for Delphi**. [S.l.], 2004. Disponível em: <http://sourceforge.net/projects/ftopf/>. Acesso em: 05 jan. 2007.

HUNTER, Jason. **The problems with JSP**. [S.l.], 2000. Disponível em: <http://www.servlets.com/soapbox/problems-jsp.html>. Acesso em: 24 ago. 2007.

ORSI, Vilmar. **Gerador de documentação para linguagem C, utilizando templates**. 2006. 98 f. Trabalho de Conclusão de Curso (Bacharelado em Ciências da Computação) – Centro de Ciências Exatas e Naturais, Universidade Regional de Blumenau, Blumenau.

PARR, Terence. **Enforcing strict modelview separation in template engines**. San Francisco, [2004?]. Disponível em: <http://www.cs.usfca.edu/~parrt/papers/mvc.templates.pdf>. Acesso em: 29 jan. 2007.

PRICE, Ana M. A.; TOSCANI, Simão S. **Implementação de linguagens de programação**: compiladores. 2. ed. Porto Alegre: Sagra Luzzatto, 2000.

RICARTE, Ivan L. M. **Programação de sistemas:** uma introdução. Campinas, 2003. Disponível em: <http://www.dca.fee.unicamp.br/cursos/EA876/apostila/HTML/progsist2003.html>. Acesso em: 30 ago. 2007.

ROCHA, L. **APE**plataforma para o desenvolvimento de aplicações web com PHP. [Salvador], 2005. Disponível em: <http://twiki.im.ufba.br/bin/view/Aside/ProjetoConclusaoDeCursoAPEMonografia#4_2_Motores_de_templates>. Acesso em: 24 ago. 2007.

SPARX SYSTEMS. **Enterprise Architect**. [S.l.], [2007?]. Disponível em: <http://www.sparxsystems.com.au/products/ea.html>. Acesso em:24 out. 2007.

STEIL, Rafael. **Introdução ao Velocity**. [S.l.], [2006?]. Disponível em: <http://www.guj.com.br/java.tutorial.artigo.18.1.guj>. Acesso em: 06 abr. 2007.

APÊNDICE A – Especificação da linguagem de *templates*

No Quadro 40 é apresentada a especificação da linguagem de *templates*.

```
#Definições regulares
carac_extras: \127 | \128 | \129 | \130 | \131 | \132 | \133 | \134 | \135 | \136 |
\137 | \138 | \139 | \140 | \141 | \142 | \143 | \144 | \145 | \146 | \147 | \148 |
\149 | \150 | \151 | \152 | \153 | \154 | \155 | \156 | \157 | \158 | \159 | \160
// Caracteres que não são reconhecidos na formação "[^xxx]" (só funcionam no
Delphi, o GALS não reconhece estes caracteres)
letra: [a-z A-Z]
numero: [0-9]
caractere: {letra} | {numero} | "-" | "_"

simbolo: . | {carac_extras}

tudo1: [^\n\r] | {carac_extras}
tudo2: [^"*"] | {carac_extras}
tudo3: [^"*""#"] | {carac_extras}
tudo4: [^\n\r"\""'"] | {carac_extras}
tudo5: [^\n\r"\"\"] | {carac_extras}
tudo6: [^\n\r"\"] | {carac_extras}

coment_linha: "#" "#" {tudo1}*
coment_bloco: "#*" ({tudo2}  | ("*")+ {tudo3})* ("*")+ "#"

const_lit_apost: "'" ( {tudo4} | "\" "\" | "\" {tudo6} )* "'"
const_lit_aspas: \" ( {tudo5} | "\" "\" | "\" {tudo6} )* \"
```

```
#Tokens
// Caracteres à ignorar:
:[\s\t\n\r]

// Símbolos especiais
"("
")"
"{"
"}"
"."
"="
","
"$"
"||"
"&&"
"!"
"=="
"!="
">"
">="
"<"
"<="
"+"
"-"
"*"
"/"
"%"
"["
"]"
".."

// Tokens
diretiva: "#" {caractere}+
identificador: {letra} {caractere}*
const_int: ("-")? {numero}+
const_flt: ("-")? {numero}+ "." {numero}+
const_lit_apost: {const_lit_apost}
```

```
const_lit_aspas: {const_lit_aspas}
comentario: {coment_linha} | {coment_bloco}
simbolo: {simbolo}
escape:! "\" ("#" | "$" | "'" | \" | "\")

// Diretivas
begin   = diretiva: "#begin"
end     = diretiva: "#end"
set     = diretiva: "#set"
if      = diretiva: "#if"
elseif  = diretiva: "#elseif"
else    = diretiva: "#else"
foreach = diretiva: "#foreach"
include = diretiva: "#include"
parse   = diretiva: "#parse"
stop    = diretiva: "#stop"
macro   = diretiva: "#macro"

// Palavras reservadas
true    = identificador: "true"
false   = identificador: "false"
in      = identificador: "in"
```

```
#Gramática
<template>                // Corpo do template
 ::= #1 #901 <instrucoes> #902 #2
 ;

<instrucoes>              // Conjunto de instruções
 ::= <instrucoes_x> #5
 ;

<instrucoes_x>            // Conjunto de instruções (fatoração)
 ::= <instrucao> <instrucoes>
 |   ε
 ;

<instrucao>               // Instrução
 ::= #3 <estatico> #4
 |   #5 comentario #105
 |   #3 "$" <instrucao_x> // Tratamento para reconhecer só um "$" ou uma referência
// | <referencia> // Retirado pois causa conflito, tratado por <instrucao_x>
 |   #5 <bloco>
 |   #5 <if>
 |   #5 <set>
 |   #5 <foreach>
 |   #5 <include>
 |   #5 <parse>
 |   #5 <macro>
 |   #3 diretiva <instrucao_y> // Tratamento para reconhecer uma diretiva sozinha
(texto estático) ou uma diretiva completa
// |   <macro_chamada> // Retirado pois causa conflito, tratado por <instrucao_y>
 |   #5 <stop>
 ;

<instrucao_x>             // Instrução (fatoração)
 ::= #6 <referencia_x> #34 // Equivalente à <referencia>
 |   ε #4 // Equivalente à <estatico>
 ;

<instrucao_y>             // Instrução (fatoração 2)
 ::= #6 <macro_chamada_x> // Equivalente à <macro_chamada>
 |   ε #4 // Equivalente à <estatico>
 ;

<estatico>                // Código estático do template (deve ser tudo, menos as
diretivas específicas)
 ::= simbolo
 |   escape
```

```
// | diretiva // Retirado pois causa conflito, tratado por <instrucao_y>
 |   identificador
 |   const_int
 |   const_flt
 |   const_lit_apost
 |   const_lit_aspas
 |   true
 |   false
 |   in
 |   "("
 |   ")"
 |   "{"
 |   "}"
 |   "."
 |   "="
 |   ","
// | "$" // Retirado pois causa conflito, tratado por <instrucao_x>
 |   "||"
 |   "&&"
 |   "!"
 |   "=="
 |   "!="
 |   ">"
 |   ">="
 |   "<"
 |   "<="
 |   "+"
 |   "-"
 |   "*"
 |   "/"
 |   "%"
 |   "["
 |   "]"
 |   ".."
 ;

<bloco>                     // Bloco de instruções
 ::= begin #103 <instrucoes> end #104
 ;

<set>                       // Diretiva "set"
 ::= set "(" <variavel> "=" <expressao> #102 ")"
 ;

<if>                        // Diretiva "if"
 ::= if #121 "(" <expressao> ")" #125 <instrucoes> <elseif> end #122
 ;

<elseif>                    // Diretiva "elseif" (complemento da "if")
 ::= ε
 |   elseif #123 "(" <expressao> ")" #125 <instrucoes> <elseif>
 |   <else>
 ;

<else>                      // Diretiva "else" (complemento da "if")
 ::= else #124 <instrucoes>
 ;

<foreach>                   // Diretiva "foreach"
 ::= foreach #111 "(" #31 <variavel> in <foreach_argumento> #32 #113 ")" #114
<instrucoes> end #112
 ;

<foreach_argumento>         // Argumento da diretiva "foreach"
 ::= <referencia>
 |   <vetor>
 ;
```

```
<include>                  // Diretiva "include"
 ::= include #141 "(" <lista_elementos_sep> ")" #142
 ;

<parse>                    // Diretiva "parse"
 ::= parse "(" <elemento> #151 ")"
 ;

<macro>                    // Diretiva "macro"
 ::=  macro  #161  #903  "("  identificador  #905  <lista_variaveis_opc>  ")"  #906
<instrucoes> end #904 #162
 ;

<macro_chamada_x>          // Chamada de uma macro (fatoração)
 ::= #181 "(" <lista_elementos_opc> ")" #182
 ;

<stop>                     // Diretiva "stop"
 ::= stop #171
 ;

<referencia>               // Referência para algum identificador
 ::= "$" <referencia_x> // Notar que, por motivos de conflito, <instrucao_x> também
possui construção equivalente a esta
 ;

<referencia_x>             // Referência para algum identificador (fatoração 1)
 ::= #35 <referencia_y>
 |   #36 "!" <referencia_y>
 ;

<referencia_y>             // Referência para algum identificador (fatoração 2)
 ::= <refer_ident>
 |   "{" <refer_ident> "}"
 ;

<refer_ident>              // Identificador da referência
 ::= identificador #33 <refer_ident_x>;

<refer_ident_x>            // Identificador da referência (fatoração 1)
 ::= "." identificador <refer_ident_y>
 |   ε
 ;

<refer_ident_y>            // Identificador da referência (fatoração 2)
 ::= #38 "(" <lista_elementos_sep_opc> ")" #39 <refer_ident_x>
 |   #37 <refer_ident_x>
 ;

<variavel>                 // Referência para variável
 ::= "$" identificador #907 #101
 ;

<lista_variaveis_opc>      // Lista de variáveis (opcional)
 ::= ε
 |   <lista_variaveis>
 ;

<lista_variaveis>          // Lista de variáveis
 ::= <variavel> <lista_variaveis_x>
 ;

<lista_variaveis_x>        // Lista de variáveis (fatoração)
 ::= ε
 |   <lista_variaveis>
 ;

<expressao>                // Expressão
```

```
 ::= <logica>
 ;

<logica>                    // Expressão lógica
 ::= <elemLogica> <logica_x>
 ;

<logica_x>                  // Expressão lógica (fatoração)
 ::= ε
 |   <operBooleano> #28 <elemLogica> #29 <logica_x>
 ;

<elemLogica>                // Elemento da operação lógica
 ::= <elemRelacional> <elemLogica_x>
 ;

<elemLogica_x>              // Elemento da operação lógica (fatoração)
 ::= ε
 |   <operRelacional> #26 <elemRelacional> #27
 ;

<elemRelacional>            // Elemento da operação relacional
 ::= <operBooleanoNot_Opc> <aritmetica> #25
 ;

<operRelacional>            // Operador relacional
 ::= "=="
 |   "!="
 |   ">"
 |   ">="
 |   "<"
 |   "<="
 ;

<operBooleano>              // Operador booleano
 ::= "||"
 |   "&&"
 ;

<operBooleanoNot_Opc>       // Operador booleano "not" (opcional)
 ::= #23 ε
 |   #24 "!"
 ;

<aritmetica>                // Expressão aritmérica
 ::= <aritm_pri2> <aritm_pri1>
 ;

<aritm_pri1>                // Expressão aritmérica, prioridade baixa
 ::= ε
 |   <operAritm_pri1> #21 <aritm_pri2> #22 <aritm_pri1>
 ;

<aritm_pri2>                // Expressão aritmérica, prioridade alta
 ::= <elemento> <aritm_pri2_x>
 ;

<aritm_pri2_x>              // Expressão aritmérica, prioridade alta (fatoração)
 ::= ε
 |   <operAritm_pri2> #21 <elemento> #22 <aritm_pri2_x>
 ;

<operAritm_pri1>            // Operador aritmético de prioridade baixa
 ::= "+"
 |   "-"
 ;

<operAritm_pri2>            // Operador aritmético de prioridade alta
```

```
 ::= "*"
 |   "/"
 |   "%"
 ;

<elemento>                // Elemento de uma expressão
 ::= <referencia> #19
 |   <numero>
 |   <literal>
 |   <vetor>
 |   true #13
 |   false #13
 |   "(" <expressao> ")"
 ;

<lista_elementos_opc>     // Lista de elementos (opcional)
 ::= ε
 |   <lista_elementos>
 ;

<lista_elementos>         // Lista de elementos
 ::= <elemento> #183 <lista_elementos_opc>
 ;

<lista_elementos_sep_opc> // Lista de elementos com separador (opcional)
 ::= ε
 |   <lista_elementos_sep>
 ;

<lista_elementos_sep>     // Lista de elementos com separador
 ::= <elemento> #143 #40 <lista_elementos_sep_x>
 ;

<lista_elementos_sep_x>   // Lista de elementos (fatoração)
 ::= ε
 |   "," <lista_elementos_sep>
 ;

<numero>                  // Definição de um número
 ::= const_int #10
 |   const_flt #10
 ;

<literal>                 // Definição de um literal
 ::= const_lit_apost #11
 |   const_lit_aspas #12
 ;

<vetor>                   // Definição de um vetor
 ::= #14 "[" <vetor_x> "]" #15
 ;

<vetor_x>                 // Definição de um vetor (fatoração)
 ::= ε
 |   <elemento> #16 <vetor_y>
 ;

<vetor_y>                 // Definição de um vetor (fatoração 2)
 ::= #17 ".." <elemento>
 |   #18 <vetor_lista_ident>
 ;

<vetor_lista_ident>       // Lista de identificadores do vetor
 ::= ε
 |   "," <elemento> #16 <vetor_lista_ident>
 ;
```

Quadro 40 – Especificação da linguagem de *templates*