UNIVERSIDADE REGIONAL DE BLUMENAU

CENTRO DE CIÊNCIAS EXATAS E NATURAIS

CURSO DE CIÊNCIAS DA COMPUTAÇÃO – BACHARELADO

GERADOR DE DOCUMENTAÇÃO PARA LINGUAGEM C, UTILIZANDO TEMPLATES

VILMAR ORSI

BLUMENAU
2006

2006/2-27

VILMAR ORSI

GERADOR DE DOCUMENTAÇÃO PARA LINGUAGEM C, UTILIZANDO TEMPLATES

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

Prof[a]. Joyce Martin, Mestre – Orientadora

BLUMENAU
2006

2006/2-27

GERADOR DE DOCUMENTAÇÃO PARA LINGUAGEM C, UTILIZANDO TEMPLATES

Por

VILMAR ORSI

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. Roberto Heinzle, Mestre – FURB
Membro:	Prof. Maurício Capobianco Lopes, Mestre – FURB

Blumenau, 15 de dezembro de 2006

Dedico este trabalho a todos que de alguma forma me incentivaram e apoiaram durante a sua elaboração, especialmente à minha esposa Samara.

AGRADECIMENTOS

A Deus, pela força para superar todos os obstáculos.

À minha família, que sempre me incentivou e apoiou. Especialmente aos meus pais que me acompanharam em toda a caminhada.

À minha esposa, pela compreensão em função de todo o tempo dedicado a elaboração do trabalho.

À minha orientadora, Joyce Martins, por ter acreditado na conclusão deste trabalho.

Ao professor Maurício Capobianco Lopes, por ter aplicado o fruto do presente trabalho na disciplina de Programação I.

Aos alunos(as) da disciplina de Programação I, por terem utilizado a ferramenta.

Aos professores que direta ou indiretamente contribuíram para minha formação.

Tudo tem seu tempo e até certas manifestações mais vigorosas e originais entram em voga ou saem de moda. Mas a sabedoria tem uma vantagem: é eterna.

Baltasar Gracián

RESUMO

No presente trabalho são descritas a especificação e a implementação do GDC, uma ferramenta para geração automática de documentação para programas em C. A ferramenta captura as informações do código fonte e comentários e, através de *templates*, gera arquivos HTML contendo a documentação correspondente. O uso de *templates* é possível através de um motor de *templates*, também desenvolvido nesse trabalho. Para gerar a documentação automaticamente, são utilizados analisadores léxico, sintático e semântico para processar os programas e os *templates*. A ferramenta foi aplicada na disciplina de Programação I, dos cursos de Ciências da Computação e de Sistemas de Informação da FURB, sendo testada e avaliada pelos acadêmicos que a utilizaram.

Palavras-chave: Documentação de software. Geração automática de documentação de software. *Templates*.

ABSTRACT

The present work, describes the specification and implementation of the GDC, an automatic code documentation generation of C programming tool. The tool captures the information from the C source code and comments, through templates, generates HTML files with the corresponding documentation. Is possible the templates use through of a templates engine, also developed in this work. To do automatically the documentation, are used analytical lexicons, syntactic and semantic to process the programs and templates. The tool was applied in class of Programming I, of the Computer Sciences and Information Systems course's FURB, being tested and evaluated by the academics that had used it.

Key-words: Software documentation. Automatic generation of software documentation. Templates.

LISTA DE ILUSTRAÇÕES

LISTA DE TABELAS

LISTA DE SIGLAS

ANSI – *American National Standards Institute*

AST – *Abstract Syntax Tree*

BCC – Bacharelado em Ciências da Computação

eNITL – *Network Improv Template Language*

FURB – Universidade Regional de Blumenau

GALS – Gerador de Analisadores Léxicos e Sintáticos

GDC – Gerador de Documentação C

HTML – *HyperText Markup Language*

JSP – *JavaServer Pages*

MPL – *Mozilla Public License*

PDF – *Portable Document Format*

RTF – *Rich Text Format*

SIS – bacharelado em SIStemas de informação

SQL – *Structured Query Language*

UML – *Unified Modeling Language*

VTL – *Velocity Template Language*

XML – *eXtensible Markup Language*

SUMÁRIO

1 INTRODUÇÃO

A evolução da indústria de software tem acompanhado de perto, senão de forma mais acelerada, a evolução da indústria tecnológica, auxiliando cada vez mais áreas no desenvolvimento de suas atividades. Como tudo que se produz, software também precisa de manutenção e adaptação. Para reduzir o tempo gasto em manutenção e adaptação, nada melhor que um software bem construído e documentado de forma correta. Neste ponto surge um problema: considerável parte dos programas[1] escritos não é documentada corretamente em função de orçamento e prazos, entre outros fatores.

Áece (2003) afirma que a criação de documentação é uma fase muito importante no desenvolvimento de um software e que esta fase é ignorada por muitos programadores, o que vem a dificultar bastante quando o projeto é utilizado por outros. Sant'Anna (2001) acrescenta que a maioria dos programas possui pouca documentação que em geral não reflete a realidade das atualizações feitas. Além do mais, os ambientes de programação não facilitam a geração de documentação. Souza, Anquetil e Oliveira (2004) dão ênfase à importância da documentação de software, destacando que vários estudos têm sido realizados para minimizar os problemas relacionados com esse assunto.

Segundo Sowek (2003), deve-se também considerar a documentação quanto ao seu conteúdo e qualidade, levando em conta a rapidez de acesso e disponibilidade da mesma. Sowek (2003) salienta ainda que não se deve esquecer a necessidade de uma atualização periódica da documentação, de acordo com as mudanças ocorridas, para que a mesma possa refletir a realidade. Nesse sentido, Tilley e Müller (1991 apud SOUZA; ANQUETIL; OLIVEIRA, 2004) "propuseram combinar documentação e código fonte de uma maneira fácil e eficiente, utilizando ferramenta de hipertexto".

Tentando resolver parte deste problema, surgiu a idéia de desenvolver uma ferramenta que automatizasse a documentação de software, a partir da análise dos códigos fonte dos programas, assim como dos comentários encontrados nos mesmos. Comentários bem elaborados, com marcações indicando autoria, funcionalidades, entre outras características, podem ser convertidos em documentos que além de auxiliarem na reutilização de código, servem como ponto de apoio quando se fizer necessária manutenção ou adaptação de alguma das funcionalidades do software.

[1] Software e programa são utilizados no decorrer do texto como sinônimos.

Algumas linguagens de programação, como Java e C# por exemplo, já contam com ferramentas com características semelhantes. Sendo assim, optou-se por desenvolver um gerador de documentação para a linguagem C, com o intuito de aplicação no meio acadêmico, objetivando desenvolver nos "potenciais programadores" o hábito de documentarem seus códigos fonte. A escolha do C se deu pelo fato de a referida linguagem ser utilizada na disciplina de Programação I do curso de Ciências da Computação da FURB, instituição onde foi viabilizado o uso do gerador de documentação pelos acadêmicos.

Na ferramenta proposta, a documentação é gerada a partir dos códigos fonte de programas desenvolvidos na linguagem C. Tais códigos são analisados léxica e sintaticamente, tendo informações extraídas tanto das linhas de código, quanto dos comentários especiais (*DocComments*) dispostos nos mesmos. Para os códigos fonte não comentados são gerados documentos simples e para os comentados, documentos complexos. As informações, após ordenadas e modeladas, de acordo com o *template*[2] desenvolvido e/ou selecionado pelo usuário, são apresentadas através de um conjunto de arquivos no formato HTML, compondo a documentação do software.

1.1 OBJETIVOS DO TRABALHO

O objetivo deste trabalho é desenvolver uma ferramenta para a geração automática de documentação para programas escritos em C.

Os objetivos específicos do trabalho são:

a) disponibilizar analisadores léxico e sintático para a análise e extração de informações do código fonte;
b) definir um conjunto de marcadores especiais com significados particulares como autor e data, que serão utilizados quando da descrição dos comentários no código fonte dos programas;
c) gerar a documentação de programas no formato HTML;
d) utilizar *templates* para a formatação da documentação gerada.

[2] Segundo Rocha (2005), *templates* são arquivos formados por variáveis e blocos especiais que serão dinamicamente transformados a partir de dados enviados pelo Motor de *Templates*.

1.2 ESTRUTURA DO TRABALHO

O presente trabalho está estruturado em quatro capítulos. No segundo capítulo é apresentada a fundamentação teórica utilizada para o desenvolvimento do trabalho. Nele são discutidos a importância da documentação de software, os geradores de código e de documentação. Também são apresentados os analisadores de linguagens de programação, implementados através dos analisadores léxico, sintático e semântico; e os motores de *templates*, dando-se maior ênfase ao motor de *templates* Velocity. O capítulo traz ainda uma descrição de alguns trabalhos correlatos.

No terceiro capítulo é apresentado o desenvolvimento, iniciando com uma visão geral da ferramenta, seguida da apresentação dos requisitos do problema e de sua especificação. Na especificação da ferramenta, são apresentados: a coleta e armazenagem dos dados; a análise e adaptação da gramática do C ANSI, como também suas especificações léxica, sintática e semântica; as especificações (léxica, sintática e semântica) da linguagem de *DocComments* e da linguagem de *templates*. Neste capítulo também são discutidas a implementação da ferramenta e os resultados obtidos.

Finalizando, no quatro capítulo, são apresentadas as conclusões e sugestões de trabalhos futuros.

2 FUNDAMENTAÇÃO TEÓRICA

Neste capítulo são apresentados os aspectos teóricos relevantes à elaboração do presente trabalho. É realizado um estudo sobre a importância da documentação de software. São analisados os geradores de código, assim como algumas vantagens e desvantagens do uso de tais ferramentas. São também examinados os geradores de documentação, sendo elencadas as fases para o seu desenvolvimento. É apresentado um estudo dos analisadores de linguagens de programação, descrevendo as funções dos analisadores léxico, sintático e semântico. São evidenciadas as características dos motores de *templates*, dando-se destaque ao motor de *templates* Velocity. Por último, são expostas algumas das características dos trabalhos correlatos que serviram de base para o desenvolvimento da ferramenta proposta.

2.1 DOCUMENTAÇÃO DE SOFTWARE

A documentação de software é motivo de preocupação no meio acadêmico, uma vez que, se a prática de documentar os softwares que se produz não se tornar hábito neste meio, é pouco provável que venha a ser adotada no meio profissional. Por não ser vital para o desenvolvimento do software, a documentação acaba por ser a última atividade realizada, isto quando não é deixada de lado. Muitas vezes os prazos são insuficientes ou os custos do projeto não possibilitam a realização da documentação. Nunes, Soares e Falbo (2004) afirmam que a documentação, embora seja um aspecto essencial para a qualidade do software, é negligenciada em decorrência de fatores como falta de uma política organizacional e falta de ferramentas para apoiar a documentação.

Pfleeger (2004, p. 370) associa a qualidade da documentação ao sucesso do software. A documentação de software, além de se tornar um hábito no dia-a-dia dos programadores, deve ser facilitada através do desenvolvimento de ferramentas que automatizem este processo. Melhor ainda se a documentação do programa puder ser gerada de forma *in-line*, ou seja, o programador escreve a documentação, enquanto escreve o código fonte do programa.

A possibilidade de gerar documentação automaticamente, através do código fonte dos programas implementados, possibilita uma economia de custos e tempo, uma vez que basta realizar comentários concisos durante o desenvolvimento do software e ter a possibilidade de

gerar uma documentação também concisa. Outra questão importante está associada à manutenção de software, uma vez que basta que os comentários sejam alterados, refletindo a realidade atual do código fonte e a nova documentação pode ser gerada garantindo a coerência entre software e documentação.

De acordo com Herrington (2003, p. 7), códigos gerados de forma automatizada são muito mais consistentes do que os escritos a “mão”, e têm todas as funcionalidades que se possa querer. A afirmação do autor é também aplicável para a geração de documentação, uma vez que a mesma, quando automatizada, passa a seguir padrões.

2.2 GERADORES DE CÓDIGO

Herrington (2003, p. 3) define geração automática de código como o uso de programas para escrever programas. Ainda de acordo com Herrington (2003, p. 15-16), a geração de código oferece vantagens significativas à Engenharia de Software, sendo elas:

a) qualidade: códigos escritos a mão em geral não mantêm a uniformidade, visto que com a evolução da codificação, novas técnicas de programação são conhecidas e aplicadas. Já os códigos gerados automaticamente tendem a manter a uniformidade, uma vez que, se descobertas novas técnicas de programação, por exemplo, basta alterar os *templates* do gerador e a melhoria é aplicada a todo o novo código gerado;
b) consistência: códigos gerados automaticamente, além de mais consistentes, são mais fáceis de serem compreendidos, corrigidos e/ou alterados, visto a uniformidade que possuem;
c) abstração: a definição da arquitetura e o projeto de quais arquivos serão gerados independem da linguagem de programação que será utilizada. Além disso, alterações simples do código, como troca de nomes de variáveis, que dispendem de quantidades significativas de tempo, são facilmente executadas, bastando realizar tais alterações nos *templates* do gerador e efetuar nova geração do código. Ainda, os *templates* são significativamente menores e mais específicos do que o código resultante, permitindo maior atenção no projeto do código a ser gerado;
d) desenvolvimento rápido: o tempo gasto para se projetar um sistema e gerar seu código automaticamente é inferior ao tempo gasto para codificá-lo manualmente.

> Deve-se levar em conta que erros no projeto também são mais facilmente corrigidos utilizando-se um gerador de código. Em códigos escritos à mão uma gama considerável de erros pode levar a re-codificação de todo o projeto. Já em códigos gerados automaticamente, basta alterar os *templates* do gerador. Além dos fatores apresentados, deve-se considerar também que parte do tempo que seria aplicado à codificação manual pode ser direcionada para o projeto do sistema, dando-lhe maior qualidade.

Moreira e Mrack (2003, p. 2, 8) complementam o *roll* de vantagens para o uso de geradores destacando a redução do tempo e conseqüentemente dos custos de desenvolvimento e manutenção de sistemas; a aplicação efetiva de uma metodologia de trabalho em todas as etapas de desenvolvimento; a independência e distribuição de tarefas entre os colaboradores; a reusabilidade dos artefatos gerados e a especialização do trabalho.

Segundo Herrington (2003, p.62-93), as ferramentas de geração de código podem ser classificadas, de acordo com suas entradas e saídas, em cinco modelos, sendo eles:

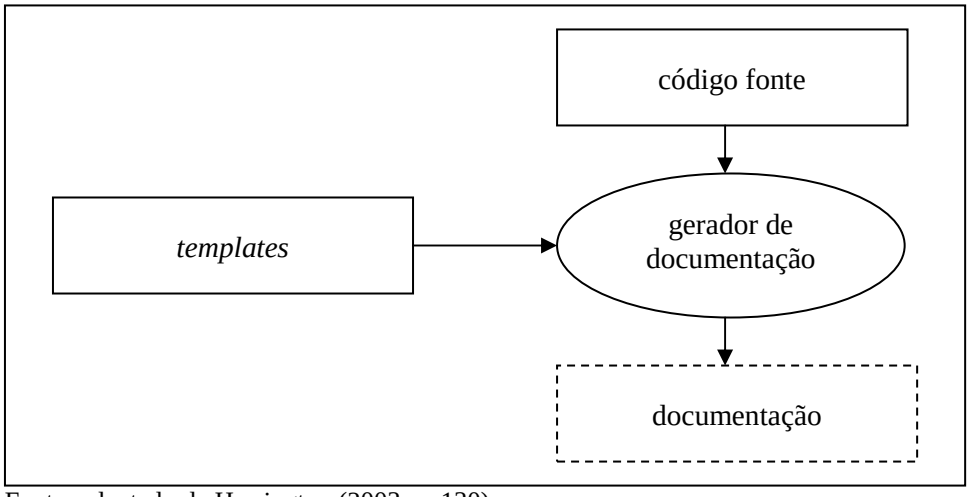
a) formatadores de código: recebem como entrada código fonte escrito em alguma linguagem de programação, como C, C++ ou Delphi, e geram como saída um ou mais arquivos que podem ser um extensão do próprio código fonte ou sua documentação;
b) expansores de código: recebem como entrada um código fonte escrito em uma linguagem de alto nível, sendo que tal código deve conter marcações especiais que são substituídas por linhas de código geradas automaticamente. A saída é um arquivo escrito na mesma linguagem do arquivo de entrada. Seu uso é sugerido para estruturar conexões com banco de dados;
c) geradores mistos: assim como os expansores de código, recebem como entrada um código fonte escrito em uma linguagem de alto nível contendo marcadores especiais. Os geradores mistos acrescentam as linhas de código (de acordo com os marcadores), sem a exclusão dos referidos marcadores. Possuem saída e sugestões de uso similares aos expansores de código;
d) geradores parciais de classes: recebem como entrada modelos abstratos contendo os requisitos do código a ser criado. Através da análise de tais modelos abstratos, são geradas estruturas de classes, que deverão ter seu código complementado manualmente. Seu uso é sugerido para o desenvolvimento de classes de interface com o usuário;
e) geradores de camadas de aplicação: assim como os geradores de classes, recebem

como entrada modelos abstratos contendo os requisitos do código a ser criado. No entanto, geram as classes com todas as suas funcionalidades, e não apenas sua estrutura. A sugestão de aplicações é semelhante a dos geradores parciais de classes.

Dentre os cinco modelos apresentados, merecem destaque os formatadores de código, visto que os geradores de documentação são exemplos de aplicações desse modelo.

2.3 GERADORES DE DOCUMENTAÇÃO

Herrington (2003, p. 62) atribui aos geradores de documentação a função de ler os arquivos com o código fonte de programas (escritos em Java, C++ ou SQL, por exemplo), analisá-los gramaticalmente e produzir como saída a documentação do software (em HTML, PDF ou arquivo texto, por exemplo), conforme pode ser observado na figura 1. Durante a análise gramatical, devem ser levados em conta a estrutura do código fonte e os *DocComments* nele encontrados.



Fonte: adaptado de Herrington (2003, p. 130).

Figura 1 – Entrada e saída de um formatador de código

Herrington (2003, p. 131) destaca que o processo de documentação de um software, no formato HTML, utilizado-se *templates*, segue as seguintes etapas:

a) leitura do(s) arquivo(s) com o código fonte;
b) identificação dos símbolos básicos (*tokens*) do(s) arquivo(s) de entrada, através da análise léxica;
c) análise dos comentários e estrutura do código fonte através da análise sintática,

extraindo e organizando as informações necessárias para gerar a documentação;

d) inicialização de uma lista de arquivos HTML;
e) para cada arquivo de código fonte:
   - criação de uma página, ou de um conjunto de páginas HTML, de acordo com o *template* utilizado,
   - armazenamento dos nomes dos arquivos HTML gerados na lista de arquivos HTML;
f) criação de um arquivo HTML contendo a lista de todos os arquivos gerados.

2.4 ANALISADORES DE LINGUAGENS DE PROGRAMAÇÃO

Os geradores de código tem como atribuição o processamento de arquivos contendo texto escrito em alguma linguagem, podendo variar entre uma linguagem de programação ou modelos abstratos. As linguagens de programação, por sua vez, podem conter construções variadas tais como comentários, constantes numéricas e alfanuméricas, declaração de variáveis e procedimentos, entre outros. A análise de comentários é possível através do uso de padrões definidos através de expressões regulares. Já para construções mais complexas, é necessária a utilização de analisadores. “Os analisadores (léxico, sintático e semântico) são módulos que compõem a estrutura básica de um compilador.” (SOUZA, 2005, p. 24).

Louden (2004, p. 1) define compiladores como programas que permitem traduzir um programa escrito em uma linguagem-fonte para uma linguagem-alvo. Na maioria das vezes a linguagem-fonte é uma linguagem de alto nível, como C ou C++, e a linguagem-alvo é o código de máquina. Louden (2004, p. 1) lembra ainda que “um compilador é um programa bastante complexo” e que “escrever um programa desses, ou mesmo entendê-lo, não é tarefa simples”, salientando que os compiladores “são usados em quase todas as formas da computação, e qualquer pessoa envolvida profissionalmente com computadores deveria conhecer a organização e as operações básicas de um compilador”.

“Um compilador é constituído internamente por passos, ou fases, para operações lógicas distintas. Essas fases [...] podem efetivamente ser escritas com operações codificadas separadamente, embora [...] sejam freqüentemente agrupadas.” (LOUDEN, 2004, p. 6). Dentre as fases de um compilador, para automatizar a geração de documentação, devem ser usados os analisadores léxico, sintático e semântico.

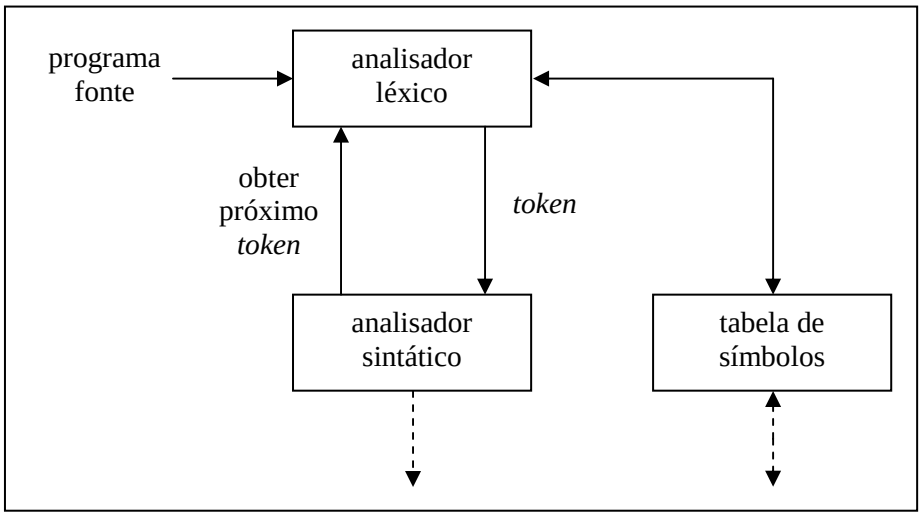
2.4.1 Analisador léxico

A análise léxica é a primeira fase do processo de compilação. Sua principal tarefa é a leitura dos caracteres de entrada e a produção de uma seqüência de símbolos básicos (*tokens*) a serem utilizados pela análise sintática, além de tarefas secundárias como remoção de comentários, de espaços em branco, de tabulações e de caracteres de avanço de linha (AHO; SETHI; ULLMAN, 1995, p. 38). Segundo Louden (2004, p. 32), os *tokens*, ou marcas, são entidades lógicas divididas em categorias, como palavras reservadas, símbolos especiais ou cadeias múltiplas de caracteres.

Price e Toscani (2000, p. 7-8) também atribuem ao analisador léxico a construção da tabela de símbolos e a emissão de mensagens de erros. “A tabela de símbolos é uma estrutura de dados gerada pelo compilador com o objetivo de armazenar informações sobre os nomes [...] definidos no programa fonte.” (PRICE; TOSCANI, 2000, p. 26). Quanto à emissão de mensagens de erros, elas são realizadas caso unidades léxicas não aceitas pela linguagem analisada sejam encontradas.

Poucos erros são identificados durante a análise léxica. No entanto, a mesma é de grande importância no momento de apresentar a posição de erros sintáticos e semânticos encontrados no código, através da informação da posição do *token* onde o referido erro foi detectado (AHO, SETHI; ULLMAN, 1995, p. 40).

De acordo com Aho, Sethi e Ullman (1995, p. 39), o analisador léxico é comumente implementado como sendo uma sub-rotina ou co-rotina do analisador sintático, visto a interação existente entre os mesmos, conforme pode ser observado na figura 2.



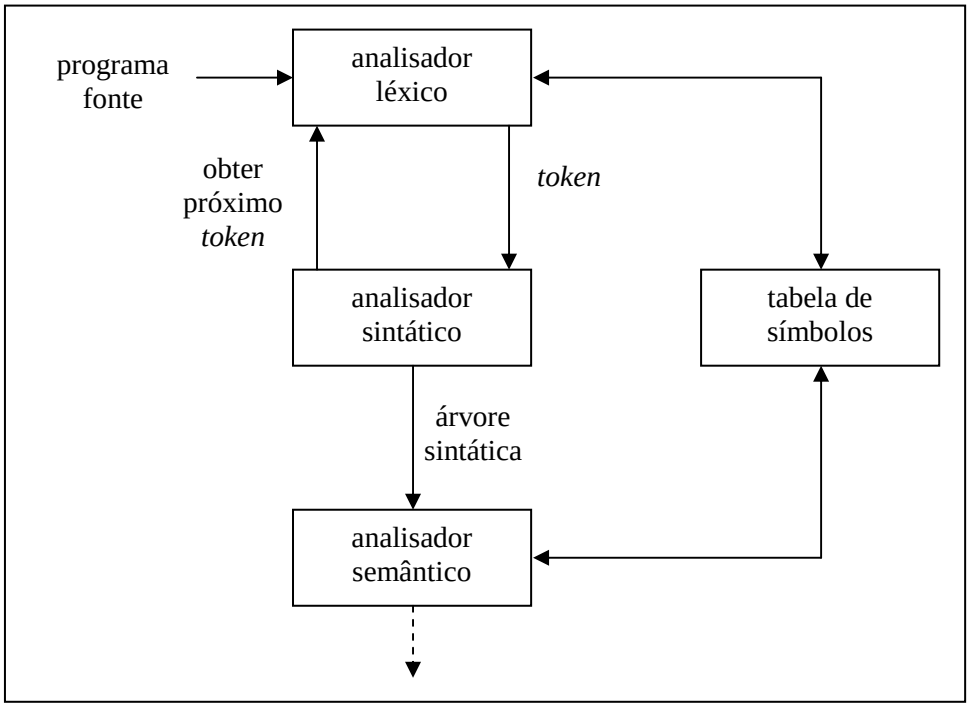
Fonte: adaptado de Aho, Sethi e Ullman (1995, p. 39).

Figura 2 – Interação do analisador léxico com o analisador sintático

2.4.2 Analisador sintático

O analisador sintático obtém *tokens* através de chamadas realizadas ao analisador léxico (figura 3), verificando se os mesmos estão ordenados em uma seqüência que possa ser gerada pela gramática da linguagem fonte. Durante este processo, erros gramaticais devem ser identificados e apresentados ao usuário (AHO; SETHI; ULLMAN, 1995, p. 72).

De acordo com Price e Toscani (2000, p. 9), durante a análise sintática, é construída uma árvore sintática, na qual é exibida a estrutura sintática do código fonte, excluindo-se redundâncias e elementos supérfluos. Esta construção pode acontecer de forma implícita ou explícita, dependendo do analisador sintático, mas de forma geral é construída implicitamente, através das chamadas das rotinas recursivas que executam a análise sintática.



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

Figura 3 – Interação do analisador sintático com o analisador semântico

Aho, Sethi e Ullman (1995, p. 72) dividem os analisadores sintáticos em três tipos, sendo eles:

a) métodos universais de análise sintática: podem tratar qualquer gramática, mas são muito ineficientes;
b) *top-down*: constroem árvores sintáticas da raiz para as folhas;
c) *bottom-up*: constroem árvores sintáticas das folhas para a raiz.

Os dois últimos tipos são os mais eficientes e ambos varrem a seqüência de *tokens* da

esquerda para a direita, símbolo a símbolo.

2.4.3 Analisador semântico

Semântica é a “parte da linguagem que cuida do significado das palavras, partes de palavras ou combinações de palavras”, ou ainda, voltada para a computação, dos “significados de palavras ou símbolos usados em programas” (VERBETE, 2001). “A semântica opõe-se com freqüência à sintaxe, caso em que a primeira se ocupa do que algo significa, enquanto a segunda se debruça sobre as estruturas ou padrões formais do modo como esse algo é expresso [...].” (SEMÂNTICA, 2006).

Louden (2004, p. 259) afirma que a fase de análise semântica recebe este nome, pois “requer a computação de informações que estão além da capacidade das gramáticas [...] e dos algoritmos padrão da análise sintática”. Desta maneira, tais informações não podem ser consideradas sintaxe. Ele salienta ainda que “a informação computada está fortemente relacionada com o significado, ou seja, a semântica, do programa traduzido” (LOUDEN, 2004, p. 259).

Já Price e Toscani (2000, p. 86) lembram dos esquemas de tradução para computar a semântica dos programas. Um esquema de tradução é uma extensão de uma gramática, “através da associação dos atributos aos símbolos gramaticais e de ações semânticas às regras de produção. [...] Ações semânticas podem ser avaliações de atributos ou chamadas a procedimentos ou funções.” Salvo no caso dos atributos dos *tokens* (computados pelo analisador léxico), os demais atributos tem seus valores calculados através da execução de ações semânticas.

De acordo com Aho, Sethi e Ullman (1995, p. 4, 147), o analisador semântico utiliza a árvore sintática, determinada pela fase de análise sintática, para identificar os operadores e operandos das expressões e enunciados. Durante este processo, verifica possíveis erros semânticos e/ou de associações de informações de tipos. Em sua discussão sobre análise semântica, Louden (2004, p. 259) a divide em duas categorias, considerando a primeira aquela requerida pelas regras da linguagem de programação, onde é verificada sua correção e garantia de execução; e a segunda aquela voltada a melhoria da eficiência de execução do programa traduzido.

2.5 MOTOR DE *TEMPLATES*

Os *templates* são arquivos que podem conter código estático e código dinâmico. O código dinâmico, composto por variáveis e comandos, será substituído dinamicamente quando o *template* for processado, gerando os arquivos de saída. Através do seu uso, pode-se manter separada a formatação dos arquivos a serem gerados, da lógica que define o que deve ser construído (SILVEIRA, 2006, p. 31).

Por sua vez, os motores de *templates*, do inglês *template engines*, são mecanismos de software que, através do uso de *templates* e de um conjunto de dados, tem por finalidade possibilitar que código dinâmico e código de interface possam estar separados, possibilitando também que sua construção possa ocorrer de forma independente. Os motores de *templates* podem ser utilizados para a construção de páginas para a internet, de geradores de código ou de ferramentas de documentação de código fonte, como o JavaDoc, por exemplo (ROCHA, 2005).

Parr (2004) associa o desenvolvimento dos motores de *templates* à necessidade de gerar páginas para a internet de forma automática, mais flexível, com menor custo de manutenção e com a possibilidade do desenvolvimento paralelo da interface com a estrutura lógica. Segundo o autor, essas necessidades levaram à proliferação dos motores de *templates*. Os primeiros *templates* para a criação de páginas para a internet eram desenvolvidos em Java não permitindo sua geração através de editores gráficos (PARR, 2004). Tal desenvolvimento acontecia de maneira monótona e trabalhosa, através de comandos de escrita (quadro 1), que uma vez executados, geravam os arquivos desejados.

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

Fonte: Parr (2004).

Quadro 1 – Geração de página HTML em Java, através de comandos `println`

Visando reduzir a monotonia e possibilitar o uso de editores gráficos na geração dos *templates*, os mesmos passaram a ser escritos em HTML, com código Java embutido (quadro 2), sendo que, durante o processamento, tal código é substituído pelas informações invocadas.

```
<html>
<body>
<h1>JSP test</h1>
Hello, <%=request.getParameter("name")%>.
</body>
</html>
```

Fonte: Parr (2004).

Quadro 2 – *Template* HTML com código Java embutido

Este avanço gerou o desenvolvimento dos motores de *templates*, que associam o *template* com as informações contidas em uma estrutura de dados, gerando os arquivos de saída, conforme apresentado na figura 4.

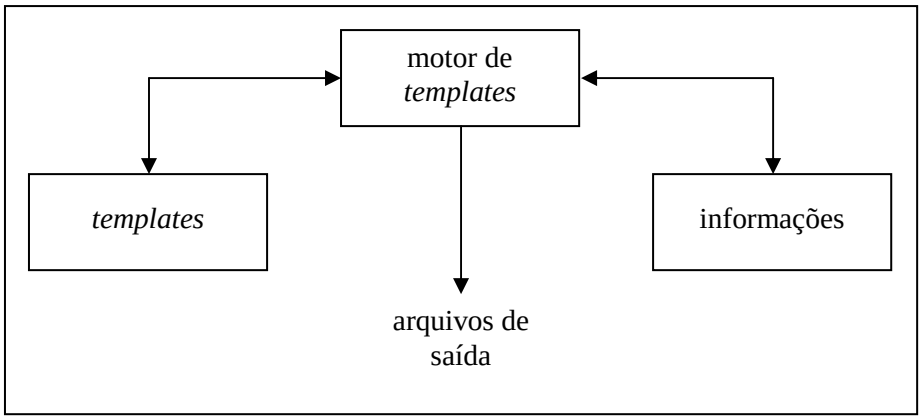


Figura 4 – Funcionamento do motor de *templates*

Rocha (2005) lembra que "cada motor de *templates* define uma linguagem através da qual os *templates* deverão ser escritos." Estas linguagens podem apresentar grandes variações quanto à complexidade, de tal forma que o motor de *templates* pode prover apenas a simples substituição de valores de variáveis ou então possuir estruturas de controle mais complexas como *loops* e comandos condicionais.

Durante o desenvolvimento do presente trabalho, foram pesquisados os seguintes motores de *templates*:

a) FastTrac: um motor de *templates* para Delphi, ainda em fase de desenvolvimento, com versões liberadas para o uso, sob a licença MPL 1.1 (HILL, 2006);
b) eNITL: um motor para linguagens de *scripts*, desenvolvido para ser utilizado através do C++ (BRECK, 1999);
c) Velocity: um motor de *templates*, *open source*, desenvolvido para a plataforma Java (CRUZ; MOURA, 2002).

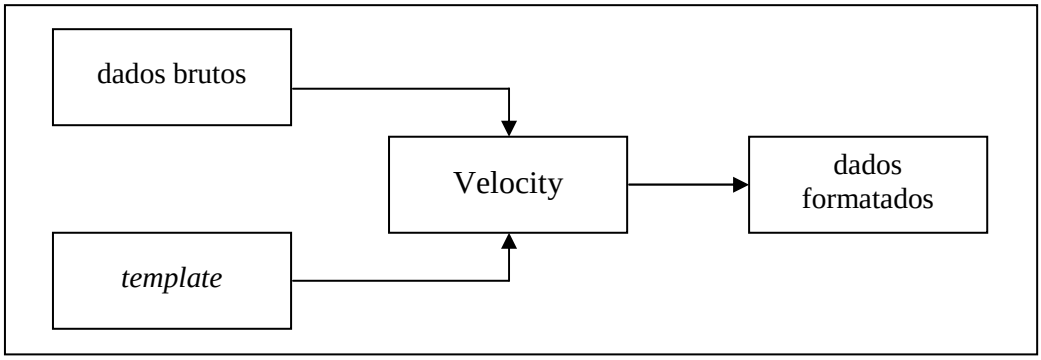
A respeito dos motores de *template* apresentados, o Velocity foi tomado como base para o desenvolvimento do motor utilizado no presente trabalho.

2.5.1 Motor de *templates* Velocity

Martin (2005) define o Velocity como um versátil *framework*[3], desenvolvido pelo grupo Jakarta, para interpretar e renderizar *templates*, tornando mais agradável a relação entre programador e *webdesigner*, permitindo que estes últimos possam trabalhar sem ter de interferir em algo mais complexo, como JSP, por exemplo.

> A atuação básica da ferramenta *Velocity* consiste na formatação textual de dados, ou seja, na camada de apresentação de dados dos sistemas (*View*). O exemplo mais comum deste tipo de aplicação ocorre na geração dinâmica de páginas na *Web*. Devido a versatilidade de seu projeto, a ferramenta *Velocity* pode também ser utilizada nas mais variadas aplicações, tais como, geração de código fonte SQL [...], *PostScript*, XML, relatórios, etc. Pode ser utilizada sozinha ou integrada com outras ferramentas de um sistema. (CRUZ; MOURA, 2002).

Conforme pode ser observado na figura 5, os dados formatados são gerados a partir dos dados brutos, que devem estar encapsulados em classes Java de forma acessível através de uma interface pública, e do *template* que deve ser escrito em VTL (CRUZ; MOURA, 2002).



Fonte: adaptado de Cruz e Moura (2002).

Figura 5 – Arquitetura do motor de *templates* Velocity

Cruz e Moura (2002) destacam que VTL é uma linguagem simples e que possui poucos comandos, permitindo desde a definição de variáveis até o controle de fluxo da execução, como pode ser observado no quadro 3.

[3] Um *framework* é uma estrutura de suporte em que um outro projeto de software pode ser organizado e desenvolvido, podendo incluir programas de apoio e bibliotecas de código (FRAMEWORK, 2006).

ELEMENTO	ESPECIFICAÇÃO	EXEMPLO
identificador	Qualquer seqüência de letras, dígitos, hífen ou *underline*, que deve começar com uma letra.	`ComboBox` `array`

declaração de variáveis	Identificador precedido por $ (cifrão) ou identificador precedido por $ (cifrão) e por ! (exclamação) ou ainda identificador entre chaves precedido por $ (cifrão).	`$ComboBox`
		`$!ComboBox`
		`${ComboBox}`

propriedades	Identificador seguido por ponto (.) e por outro identificador, sendo que pode ser usado $ (cifrão) ou ! (exclamação) ou chaves, como na declaração de variáveis.	`$ComboBox.name`
		`$!ComboBox.name`
		`${ComboBox.name}`

método	Cifrão ($) seguido por ! (exclamação) e, entre chaves, identificador seguido por ponto (.), pelo nome do método, por abre e fecha parênteses, sendo que entre os parênteses pode existir uma lista de argumentos separados por vírgula.	`$!{ComboBox.getName()}`
atribuição	`#set`	`#set($flagEstado = "true")`

controle de fluxo de execução	`#if-#elseif-#else-#end`: comando condicional
	`#foreach-#end`: comando de repetição
		`#if($!{ComboBox.getText()})`
		`  ...`
		`#end`
		
		`#foreach ($ComboBox in $array)`
		`  ...`
		`#end`

Fonte: adaptado de Cruz e Moura (2002, p. 4) e Silveira (2006, p. 32).

Quadro 3 – Elementos da linguagem VTL

De acordo com Cruz e Moura (2002), a execução do Velocity segue os seguintes passos:

a) uma instância do Velocity é criada;
b) um esquema de mapeamento entre o *template* e os objetos Java que contêm os dados brutos da aplicação é especificado;
c) o *template* é analisado e armazenado em memória em uma AST;
d) a AST é utilizada na substituição das referências (definidas no *template*) pelos valores obtidos dos objetos Java.

2.6 TRABALHOS CORRELATOS

Algumas ferramentas desempenham papel semelhante à ferramenta proposta neste trabalho, cada qual com as suas peculiaridades e para determinadas linguagens. Dentre elas, serviram como fonte de pesquisa o JavaDoc, o gerador de documentação do C#, CDoc e Doxygen.

2.6.1 JavaDoc

"**Javadoc** é um sistema de documentação criado pela Sun Microsystems para documentar a API dos programas em Java, a partir do código-fonte. O resultado é expresso em HTML." (JAVADOC, 2006, grifo do autor).

Através do uso do JavaDoc pode-se extrair os comentários do código fonte Java e criar uma documentação no formato HTML. Segundo Flanagan (2000, p. 22), a linguagem Java suporta três tipos de comentários, sendo eles: de linha, de várias linhas (ou de bloco) e especiais para a geração de documentação. Os comentários para a geração de documentação iniciam com os caracteres `/**` e finalizam com `*/` sendo chamados de *DocComments*.

É importante destacar que os comentários para geração de documentação podem incluir, além dos próprios comentários, *tags* HTML simples, tais como `<I>`, `<CODE>`, entre outros, e *tags* de *DocComment* definidos pelo JavaDoc, como por exemplo `@author`, `@param`, etc., que delimitam informações adicionais. Os *DocComments* devem aparecer imediatamente antes de uma definição de classe, interface, método ou campo (FLANAGAN, 2000, p. 214).

> O corpo de um doc comment deve começar com um resumo de uma frase para a classe, interface, método ou campo que estão sendo comentados. Essa frase pode ser exibida sozinha, como documentação de resumo, de modo que deve ser escrita para ser independente. A frase inicial pode ser seguida por várias outras frases e parágrafos que descrevam a classe, interface, método ou campo.
> Após os parágrafos descritivos, um doc comment pode conter qualquer número de outros parágrafos, cada um deles começando com um tag especial de doc-comment, tal como `@author`, `@param` ou `@returns`. Esses parágrafos com tags fornecem informações específicas sobre a classe, interface, método ou campo que o programa *javadoc* exibe de uma forma padrão. (FLANAGAN, 2000, p. 215, grifos do autor).

Pamplona (2006) destaca que os *tags* mais utilizados são:

a) `@author`: identifica o autor de uma classe ou método;
b) `@deprecated`: indica que um componente deve ser removido nas próximas versões;
c) `@param`: aponta os parâmetros de um método;
d) `@return`: indica o retorno de um método;
e) `@see`: informa onde encontrar mais informações sobre o assunto;
f) `@since`: apresenta a data da escrita do código;
g) `@throws`: apresenta possíveis exceções geradas durante a execução do método;
h) `@version`: identifica a versão de determinada classe ou método.

No quadro 4 pode ser observado um código fonte Java, com um *DocComment*, sendo que sua respectiva documentação, gerada pelo JavaDoc, é apresentada na figura 6.

```
/**
       Esta classe chamada de BemVindo é simplesmente
       um Olá Mundo! da linguagem Java.
       @author Vitor Fernando Pamplona
       @since 15/01/2005
*/
public class BemVindo {
       public static void main(String[] args) {
              System.out.println("Olá Mundo!");
       }
}
```

Fonte: Pamplona (2006).

Quadro 4 – Exemplo de código fonte Java

Fonte: adaptado de Pamplona (2006).

Figura 6 – Documentação gerada pelo JavaDoc

Segundo Paulo e Graça (2003), com o JavaDoc também é possível gerar documentação para pacotes de classes, sendo que os comentários para os mesmos devem ser escritos em um arquivo chamado `package.html`, que deve ser salvo no mesmo diretório onde forem salvos os arquivos com extensão Java, contendo o código fonte das classes do pacote que se deseja documentar. Quando for gerada a documentação, as informações contidas no arquivo `package.html` serão extraídas e adicionadas à primeira página da documentação HTML produzida para o pacote.

2.6.2 Gerador de documentação do C#

Segundo Haddad (2001, p. 375), a Microsoft desenvolveu uma ferramenta para permitir documentação automática de programas C#, seguindo o padrão XML. Basta que sejam seguidos os padrões dos marcadores XML nos comentários do código fonte, e a documentação do mesmo poderá ser gerada. Haddad (2001, p. 375) alerta que simplesmente será gerada a documentação do código fonte, de acordo com os comentários apresentados, e não uma documentação do fluxo das informações no aplicativo.

Câmara (2003a) explica que “enquanto os comentários em C# são iniciados com ‘//’, os comentários iniciados com ‘///’ indicam documentação que pode ser extraída pelo compilador e manipulada”. Sendo assim, sempre que o compilador encontrar os caracteres `///` considera tratar-se de um comentário a ser utilizado na geração da documentação e realiza a manipulação das informações contidas no comentário, de acordo com o elemento XML apresentado no mesmo.

De acordo com Galuppo, Matheus e Santos (2004, p. 467), os *tags*, ou elementos XML, utilizados quando da realização dos comentários para a geração de documentação, são padronizados e formam um conjunto de dezoito pares. Observa-se a necessidade de delimitar as informações com *tags* de início e fim, como por exemplo `<author>` Galuppo `</author>`.

Góis (2005) lembra que por se tratar de uma documentação gerada em XML, novos pares de *tags* podem ser criados, e destaca os mais usados, sendo que, dentre eles, encontram-se:

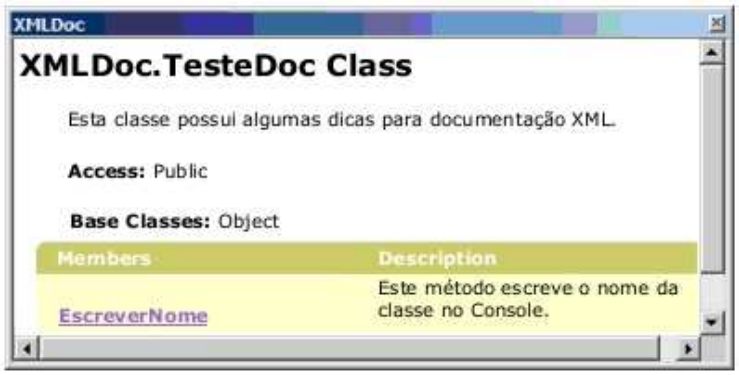
a) `<code> … </code>`: indica que o texto incluído é código da aplicação;
b) `<example> … </example>`: informa um exemplo de como usar um método;
c) `<exception> … </exception>`: informa uma exceção;
d) `<remarks> … </remarks>`: apresenta uma descrição mais detalhada.
e) `<returns> … </returns>`: informa o retorno de um método;
f) `<summary> … </summary>`: apresenta uma breve descrição de uma classe, método ou propriedade;
g) `<value> … </value>`: descreve uma propriedade.

No quadro 5 pode-se observar partes de um programa escrito em C#, sendo sua respectiva documentação apresentada na figura 7.

```
using System;
namespace XMLDoc
{
	/// <summary>
	/// Esta classe possui algumas dicas para documentação XML.
	/// </summary>
	public class TesteDoc
	{
		/// <summary>
		/// Este método escreve o nome da classe no Console.
		/// </summary>
		public void Escrever (string descricao, int numero)
		{
			...
		}
	}
}
```

Fonte: adaptado de Câmara (2003b).

Quadro 5 – Exemplo de código fonte C#



Fonte: adaptado de Câmara (2003b).

Figura 7 – Documentação gerada para um programa em C#

2.6.3 CDoc

Segundo a Software BlackSmiths (2001), o CDoc pode gerar documentação para códigos fonte de programas desenvolvidos nas linguagens C, C++ e Java. Além da documentação, o CDoc pode também gerar diagramas com a árvore de chamadas de funções, árvore hierárquica de classes, cálculo ciclomático de complexidade, entre outros.

Assim como o JavaDoc e o gerador de documentação do C# descritos nas seções anteriores, o CDoc necessita de comentários especiais para a geração de documentação complexa a partir do código fonte de programas, diferindo-se por não fazer uso de *tags* e/ou marcadores especiais. Em se tratando de programas desenvolvidos na linguagem C, Milvang (2004) apresenta que os comentários no código fonte devem ser delimitados pelos caracteres `/*?*` indicando o início do comentário e `*/` indicando o fim do mesmo. Lembra também que a abertura dos *DocComments* pode conter o termo `:subject`, sendo que este pode também

estar acompanhado ou não por um *link* de uma página com informações adicionais, como se observa no quadro 6.

```
/*?:subject*
_____________________________________________
              modulename
_____________________________________________
Name:         modulename - one line description
Syntax:       | modulename [ <option>... ] args
              | more syntax
Keyword 1:    text
              text may go over more than one line
Keyword 2:    You may underline a word by
              putting it between 'apostrophes'
              (apostrophes may be printed by
              using two apostrophes '').
*/


/*?:*
_____________________________________________
              modulename
_____________________________________________
Name:         modulename - one line description
*/


/*?:subject=manpage */
```

Fonte: adaptado de Milvang (2004).

Quadro 6 – Exemplos de *DocComments* utilizados pelo CDoc

2.6.4 Doxygen

De acordo com Heesch (2006), o Doxygen é um sistema de documentação para as linguagens C++, C, Java, Objective-C, entre outras, podendo ser utilizado como gerador de documentação a partir de comentários realizados nos códigos fonte de programas escritos nas linguagens anteriormente descritas, ou ainda podendo ser útil para extrair a estrutura do código fonte de arquivos não comentados.

Terra (2006) afirma ainda que o processo básico de funcionamento do Doxygen, na geração de documentação, consiste em buscar comentários no código fonte dos programas a serem documentados e relacioná-los às estruturas descritas nestes mesmos programas. Depois desta análise e relacionamento entre comentários e estruturas, a ferramenta gera a documentação automaticamente no formato escolhido. Quanto aos formatos em que a documentação pode ser gerada, Heesch (2006) descreve que o formato principal é o HTML, sendo que a ferramenta traz também as opções de os arquivos de saída serem dos tipos RTF ou PDF, XML, entre outros.

A sintaxe dos comentários para geração de código é bastante parecida da proposta

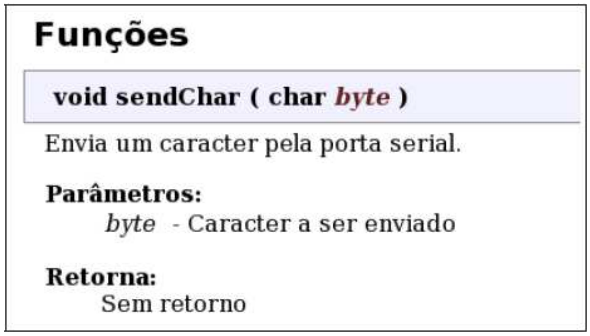
pelos demais trabalhos apresentados, porém, só são utilizadas *tags* da própria ferramenta, como por exemplo `\author`, `\class`, entre outros. Assim como o JavaDoc, o Doxygen só faz a geração de documentação a partir de comentários especiais, delimitados por `///` no início de cada linha onde o comentário for inserido, para comentários de linha, e `/** ... */`, para comentários de bloco.

No quadro 7, pode ser observado um trecho de programa, comentado para a geração de documentação pelo Doxygen, sendo a respectiva documentação apresentada na figura 8. Terra (2006) destaca a importância de que "todas as classes, atributos, métodos, parâmetros e retornos sejam documentados, com suas respectivas descrições", visando seguir um padrão na documentação.

```
1  /**
2      \brief Envia um caracter pela porta serial
3      \param byte - Caracter a ser enviado
4      \return Sem retorno
5  */
6  void sendChar(char byte);
```

Fonte: Renaldi (2006, p. 38).

Quadro 7 – Exemplos de *DocComments* formatados para o Doxygen



Fonte: Renaldi (2006, p. 38).

Figura 8 – Documentação gerada pelo Doxygen

Segundo Terra (2006), alguns caracteres como `\`, por exemplo, são reservados ao uso interno do Doxygen. A inserção destes caracteres em um comentário deve seguir o seguinte formato: `\\`, que irá gerar o caracter `\`, `\@` que irá gerar o caracter `@`, etc.

Heesch (2006) apresenta ainda uma lista de funcionalidades que devem ser adicionadas ao Doxygen, salientando que em função da quantidade e do grau de dificuldade das mesmas, a implementação não deve ser concluída com brevidade. Dentre as funcionalidades propostas, Heesch (2006) apresenta o uso de *templates* pelo Doxygen, associando a esta tarefa o grau máximo de dificuldade de desenvolvimento.

3 DESENVOLVIMENTO DA FERRAMENTA

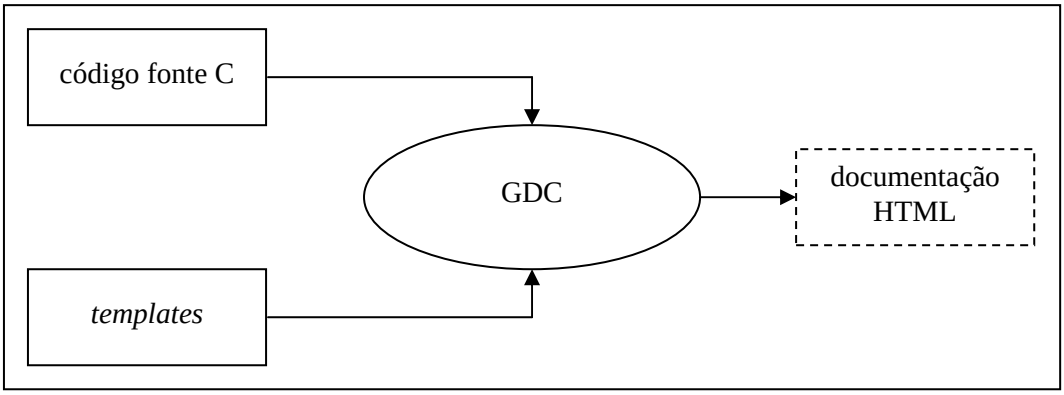
Com base nas etapas elencadas para a documentação de software no formato HTML descritas no capítulo anterior, o presente trabalho foi desenvolvido através das seguintes fases:

a) levantamento dos requisitos;
b) análise e identificação das informações necessárias para gerar a documentação;
c) adaptação da gramática C ANSI;
d) especificação de uma linguagem de *DocComments*;
e) especificação de uma linguagem de *templates*;
f) especificação da ferramenta através dos diagramas UML de casos de uso e classes;
g) implementação da ferramenta.

3.1 REQUISITOS DA FERRAMENTA

Um gerador de documentação necessita basicamente de um ou mais arquivos contendo código fonte e, opcionalmente, um ou mais *templates* contendo o modelo da saída desejada. Através dos elementos citados, são construídos um ou mais arquivos contendo a documentação.

Assim o GDC, ferramenta desenvolvida nesse trabalho, tem como entrada código fonte escrito em C e *templates*. Tal código é analisado léxica e sintática pelo gerador de documentação, tendo informações extraídas tanto das linhas de código quanto dos comentários de documentação (*DocComments*) dispostos nas mesmas. Além da análise do código fonte e extração das informações, o GDC usa os modelos apresentados nos *templates* para gerar um ou mais arquivos HTML (figura 09).



Fonte: adaptado de Herrington (2003, p. 130).

Figura 9 – Entrada e saída de dados do GDC

Na seqüência são apresentados os requisitos funcionais e não funcionais atendidos pelo GDC. No quadro 8 podem ser observados os requisitos funcionais. Já no quadro 9, podem ser observados os requisitos não funcionais.

REQUISITOS FUNCIONAIS	**CASO DE USO**
RF01: Gerar documentação, no formato HTML, de programas desenvolvidos na linguagem C, a partir do código fonte dos mesmos.	UC01
RF02: Realizar análises léxica e sintática do código fonte de programas escritos na linguagem C, localizando informações para o documento a ser gerado.	UC01
RF03: Permitir o uso de *tags* HTML para formatação de texto nos comentários.	UC01
RF04: Usar *templates* para a modelagem do documento a ser gerado.	UC01
RF05: Disponibilizar uma interface para permitir a elaboração dos *templates*.	UC02
RF06: Disponibilizar uma interface para facilitar a inserção de comentários no código fonte.	UC03
RF07: Disponibilizar um conjunto de marcadores especiais para ser utilizado nos comentários do código fonte.	UC03

Quadro 8 – Requisitos funcionais

REQUISITOS NÃO FUNCIONAIS
RNF01: Utilizar marcadores especiais no estilo dos marcadores definidos no JavaDoc.
RNF02: Ser implementado utilizando o ambiente Delphi 7.0.
RNF03: Ser compatível com o sistema operacional Windows 98, 2000 e XP.

Quadro 9 – Requisitos não funcionais

3.2 IDENTIFICAÇÃO DAS INFORMAÇÕES PARA A DOCUMENTAÇÃO

Para gerar documentação é necessário elicitar qual conteúdo que se deseja apresentar como saída. No caso do GDC, a saída é um ou mais arquivos no formato HTML, contendo a documentação para código fonte escrito em C. A partir dos trabalhos correlatos estudados, definiu-se que as diretivas listadas no quadro 10 seriam analisadas pela ferramenta.

DIRETIVA	DESCRIÇÃO
`#include`	Inclusão de arquivos: provê a substituição da linha do `#include`, pelo código do arquivo incluído.
`#define`	Definição e expansão de macros: provê a substituição das ocorrências subseqüentes do identificador definido pela seqüência de código indicada.
`#undef`	Faz com que a definição de um identificador seja esquecida.

`#if`
`#ifdef`
`#ifndef`
`#elif`
`#else`
`#endif`
	Compilação condicional: possibilita que partes de um programa possam ser compiladas condicionalmente.

`#line`	Controle de linha: faz com que o compilador considere que o número de linha da próxima linha seja o apontado pela constante numérica indicada.
`#error`	Geração de erro: gera mensagem de erro incluindo a seqüência de código indicada.
`#pragma`	Faz com que seja executada uma ação dependente da implementação.

Quadro 10 – Diretivas do C

O GDC analisa também as declarações encontradas no código fonte C. Quanto às declarações analisadas, são armazenadas informações referentes aos enumeradores (`enum`), às estruturas (`struct` e `union`), às definições de tipos (`typedef`), às variáveis e às funções.

Além de possibilitar a geração de documentação para arquivos, o GDC permite que seja gerada a documentação de projetos[4]. Assim como o JavaDoc, onde a documentação de pacotes implica na criação de um arquivo denominado `package.html`, o GDC foi especificado de forma a ser necessária a criação de um arquivo com a descrição do projeto (extensão `.PJT`), que deve ser gravado no mesmo diretório onde encontram-se os demais arquivos do projeto (figura 10).

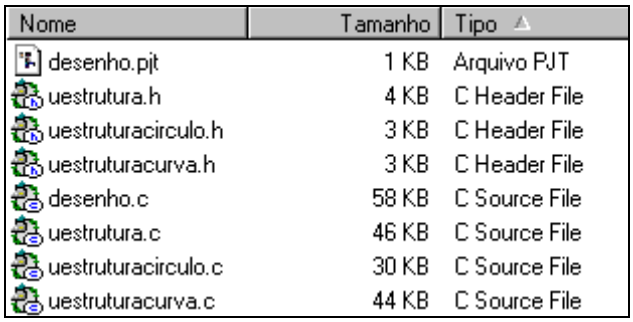


Figura 10 – Estrutura do diretório de um projeto

O arquivo `.PJT` deve listar todos os arquivos que compõem o projeto. Para tanto, usa-se a diretiva `#include` do C. Conforme pode ser observado no quadro 11, tal arquivo pode conter ainda um *DocComment*, sendo que esse deve possuir o *tag* GDC `@cabeçalho:`.

[4] É considerado projeto, um conjunto de arquivos escritos em C, que fazem parte de um mesmo programa.

```
/*@
   @cabeçalho:
   @autor: Vilmar Orsi
   @motivação: Desenvolver uma ferramenta para desenhos
*/
#include "desenho.c"
#include "uestrutura.c"
#include "uestrutura.h"
#include "uestruturacirculo.c"
#include "uestruturacirculo.h"
#include "uestruturacurva.c"
#include "uestruturacurva.h"
```

Quadro 11 – Estrutura de um arquivo `.PJT`

3.3 ANÁLISE E ADAPTAÇÃO DA GRAMÁTICA C ANSI

Durante o desenvolvimento do GDC, foi feita uma busca na tentativa de encontrar uma gramática C ANSI concisa com as construções léxicas, sintáticas e semânticas descritas pela linguagem. Dentre as gramáticas analisadas, a gramática considerada mais concisa ainda não continha todas as construções previstas pelo C, necessitando ser adaptada. As construções gramaticais que, de acordo com Kernighan (1990), são previstas pelo C ANSI e que não são aceitas pela gramática original (anexo A) podem ser observadas no quadro 12.

CONSTRUÇÃO	EXEMPLO

declaração de tipos derivados e de variáveis a partir dos tipos derivados	`struct RegistroProduto {`
	`  int codigoProduto;`
	`  char nomeProduto[30];`
	`};`
	`int main() {`
	`  struct RegistroProduto produto;`
	`}`

declaração de estruturas (`struct`, `union`) internas a funções	`int main() {`
	`  struct RegistroProduto {`
	`  int codigoProduto;`
	`  char nomeProduto[30];`
	`  };`
	`}`

inicialização de *arrays* quando da declaração	`int valores[] = {1, 2, 3, 4, 5};`
definição de tipos (`typedef`) contendo *arrays*	`typedef char[5] nome;`

definição de tipos (`typedef`) contendo estruturas (`struct`, `union`)	`typedef struct nodo {`
	`  char x;`
	`  int y;`
	`};`

definição de enumeradores (`enum`) acompanhados apenas de um identificador ou de uma listas de identificadores	`enum x;`
	`enum {x=1};`
	`enum {x=1, y=2};`

Quadro 12 – Construções sintáticas não previstas pela gramática original

A partir das alterações realizadas nas construções sintáticas previstas na gramática original, as construções acima apresentadas são aceitas pela gramática atual (apêndice A).

Deve-se salientar ainda que algumas construções previstas apenas pelo C++, foram inseridas na gramática do GDC, visto que as mesmas são utilizadas pelos acadêmicos da disciplina de Programação I, atuais usuários da ferramenta. Essas alterações são apresentadas no quadro 13.

CONSTRUÇÃO	EXEMPLO
declaração e inicialização de variáveis nos argumentos do comando `for` (o C ANSI prevê apenas a inicialização da variável)	`for (int lin=0; lin<n; lin++) {` `}`
declaração de variáveis do tipo `bool` (este não é um tipo primitivo do C ANSI)	`bool erros;`

Quadro 13 – Construções sintáticas não previstas pelo C ANSI

3.3.1 Especificação léxica

O conjunto de *tokens* foi definido seguindo a especificação léxica do C ANSI. Além dos padrões de formação dos comentários de linha e de bloco, foi definido o padrão de formação dos *DocComments*.

Um *DocComment* é um *token* que inicia com os caracteres `/*@`, seguidos de quaisquer caracteres diferentes de `*/`, e finaliza com os caracteres `*/`, podendo conter várias linhas. Um comentário de bloco diferencia-se do anterior apenas por ser inicializado com os caracteres `/*`. Já os comentários de linha iniciam com `//`, seguidos por qualquer seqüência de caracteres e finalizam com quebra de linha (quadro 14). Os comentários de bloco e de linha não são relevantes para o GDC.

```
DocComent  : (/\*) (@) (([^\*])|(\*[^/]))* ((\*)+/)
LineComent : (/)(/)+.*
TextComent : (/\*) (([^\*])|(\*[^/]))* ((\*)+/)
```

Quadro 14 – Especificação léxica dos comentários

A especificação dos demais *tokens* reconhecidos pelo GDC encontra-se no apêndice A.

3.3.2 Especificação sintática

A especificação sintática segue os padrões gramaticais definidos pelo C ANSI, acrescidos das construções apresentadas anteriormente. Salienta-se que em função dessas alterações, alguns erros gramaticais podem não ser identificados pelo GDC. No entanto, considera-se que todo código fonte que se deseja documentar, já tenha sido compilado e, conseqüentemente, esteja correto. A especificação das construções sintáticas aceitas pelo

GDC é apresentada no apêndice A.

3.3.3 Especificação semântica

Ao invés de analisar se o código está semanticamente correto, no GDC o analisador semântico extrai as declarações existentes no referido código. Para tanto, faz uso dos marcadores semânticos aplicados à gramática, a fim de armazenar as informações referentes a tais declarações, para a posterior geração da documentação. Através das ações semânticas é possível saber qual o tipo de declaração está sendo realizada; se uma variável é global ou local; se for local, a qual tipo de declaração está associada, etc.

No quadro 15, pode-se observar algumas das ações semânticas associadas às regras gramaticais para o reconhecimento da declaração de uma função.

```
<Decls>       ::= #1 <Decl> #2 <Decls>  |  ...
<Decl>        ::= <Sign> <Scalar> #13 <Pointers> <Decl_aux1>  |  ...
<Sign>        ::= ε  |  ...
<Scalar>      ::= void  |  ...
<Pointers>    ::= ε  |  ...
<Decl_aux1>   ::= Id <Decl_aux1_>  |  ...
<Decl_aux1_>  ::= ( <Params> ) #11 <BlockFunc>  |  ...
<Params>      ::= ε  |  ...
<BlockFunc>   ::= <Block>  | ...
<Block>       ::= { <StmList> }
<StmList>     ::= ε  |  ...
```

Quadro 15 – Trecho da gramática do C, com ações semânticas

A ordem de execução das ações semânticas para a análise de `void main () {}` é:

a) ação semântica `#1`: marcar a posição do início da declaração;
b) ação semântica `#13`: armazenar a posição final da palavra `void`;
c) ação semântica `#11`: marcar a posição final do cabeçalho da função, que deve ser apresentado na documentação;
d) ação semântica `#2`: verificar o tipo da declaração, extrair as informações da declaração e armazená-las.

O processamento das demais ações semânticas é semelhante para todos os tipos de declarações, devendo-se levar em conta que para cada tipo de declaração existe um conjunto de ações associadas. A gramática possui um total de dezoito ações, numeradas de `#1` a `#18` (quadro 16).

AÇÃO SEMANTICA	FUNÇÃO
#1	Marcar início da declaração.
#2	Verificar tipo da declaração; criar objetos para armazenagem das informações; extrair informações; analisar *DocComments*; armazenar informações.
#3	Indicar que a declaração é de enumerador (`enum`).
#4, #8 e #15	Setar tipo da declaração.
#5	Marcar fim da declaração de variáveis.
#6	Indicar que a declaração é de definição de tipo (`typedef`).
#7, #12, #16, #17 e #18	Controlar fluxo de informações.
#9	Setar que *token* `mod` foi encontrado.
#10	Setar fim da declaração.
#11 e #14	Setar tipo e fim da declaração.
#13	Setar fim da definição de tipo (`typedef`).

Quadro 16 – Ações semânticas da gramática do C

3.4 ESPECIFICAÇÃO DA LINGUAGEM DE *DOCCOMMENTS*

A linguagem de *DocComments* (apêndice B) foi especificada de forma a possibilitar a associação de documentação ao cabeçalho de um programa, às suas diretivas e às suas declarações, podendo ser classificada de acordo com o conteúdo que representa, através dos *tags* disponibilizados pela linguagem de *DocComments*.

A linguagem de *DocComments* possui um conjunto de dezesseis *tags*, sendo eles:

a) `@algoritmo:` descreve o algoritmo implementado no arquivo ou em uma função;
b) `@autor:` define o autor do arquivo ou de uma função. Pode também ser utilizado em declarações de variáveis ou diretivas, mas é recomendado apenas para os dois casos anteriormente citados;
c) `@cabeçalho:` define o *DocComment* como cabeçalho do arquivo. Deve ser utilizado apenas uma vez em cada arquivo. No caso de ser utilizado múltiplas vezes, apenas o primeiro registro é analisado e armazenado para a documentação;
d) `@data:` define a data de criação/alteração do arquivo. Também pode ser utilizado para definir a data de criação/alteração de uma função ou de uma declaração de variáveis, assim como para indicar a data em que uma diretiva foi adicionada ao arquivo. Não existe uma formatação específica para a data, porém sugere-se que a formatação seja padronizada pelo usuário;
e) `@descontinuado:` define que uma função ou uma diretiva (`include`) foram

descontinuados. Pode também indicar a descontinuidade do código fonte. É aconselhável que se justifique o motivo da descontinuidade e que seja sugerida uma substituição;

f) `@descrição:` descreve as funcionalidades implementadas no arquivo, em uma função ou para justificar uma diretiva ou a declaração de variáveis;
g) `@desde:` indica desde quando, ou desde que versão, uma função foi implementada, uma variável foi declarada, ou diretivas foram adicionados ao arquivo. Não existe uma formatação específica para a data ou versão, porém sugere-se que a formatação seja padronizada pelo usuário;
h) `@entrada:` descreve as entradas de dados do arquivo ou de uma função;
i) `@fonte:` indica a fonte de pesquisa para a elaboração do arquivo ou de uma função, ou para a realização de uma inclusão de arquivo (`include`). Também pode informar a fonte de onde o arquivo ou uma função foram copiados;
j) `@link:` apresenta um *link*, indicando onde informações sobre o assunto que se está comentando podem ser encontradas;
k) `@motivação:` indica a motivação para o desenvolvimento do arquivo ou de uma função;
l) `@orientação:` indica quem orientou a implementação do arquivo ou de uma função;
m) `@parâmetro:` apresenta informações sobre os parâmetros formais declarados em uma função;
n) `@retorno:` apresenta informações sobre o retorno de uma função;
o) `@saída:` indica os resultados do processamento realizado no arquivo ou em uma função;
p) `@versão:` indica a versão do arquivo. Deve ser utilizado no *DocComment* marcado como cabeçalho.

A especificação desses *tags* foi baseada nos *tags* do JavaDoc e nas sugestões do professor da disciplina de Programação I, Maurício Capobianco Lopes. Embora os mesmos sejam *case sensitive* e escritos em português, os comentários associados a eles não estão sujeitos às mesmas regras, podendo ser escritos em qualquer idioma, e utilizar caracteres em caixa alta e baixa indiscriminadamente.

Além dos *tags* definidos pelo GDC, *tags* HTML também podem ser utilizados nos *DocComments*, possibilitando que partes das informações contidas nos mesmos possam ser

evidenciadas na geração da documentação. Os *tags* HTML não são tratados pela linguagem de *DocComments*, sendo considerados comentários associados aos *tags* GDC. Sugere-se que apenas os *tags* mais simples sejam utilizados (quadro 17), facilitando a elaboração dos *DocComments*.

```
/*@
  @cabeçalho:
  @autor: <b>Vilmar Orsi</b>
  @versão: <i>Beta 1.0</i>
  @entrada: Um número <u>inteiro</u>
*/
```

Quadro 17 – Exemplo do uso de *tags* HTML em um *DocComment*

Uma vez finalizada a especificação da linguagem de *DocComments*, o próximo passo é a especificação léxica da linguagem.

3.4.1 Especificação léxica

Enquanto para o analisador léxico do C, um *DocComment* é um *token*, para o analisador léxico da linguagem de *DocComments* ele é composto por três tipos de *tokens* (quadro 18), sendo eles: identificadores, *tags* e delimitadores de *DocComments*.

Os identificadores são qualquer seqüência de caracteres, exceto quebra de linha, que não iniciam com `@`, nem com caracteres de formatação (espaço em branco, tabulação ou quebra de linha). Os *tags* iniciam com `@`, seguido por uma ou mais letras minúsculas (incluindo ã, â, í, ç) e finalizam com `:`, sendo reconhecidos a partir de um conjunto de dezesseis casos especiais do *token tag*, citando como exemplo `algoritmo`. Já o delimitador de início de *DocComments* (`inicio`) é constituído por `/*@`, seguido ou não de caracteres de formatação, e o delimitador de fim (`fim`) definido pelos caracteres `*/`.

```
// Definições regulares auxiliares
letra           : [a-zãâíç]
Ascii_          : .
Ascii_semArroba : [^@\n\t\s\r]
Whitespace      : [\n\t\s\r]
```

```
// Tokens
// Identificadores
Id  : {Ascii_semArroba} {Ascii_}*

// Tags
tag : @{letra}+:
algoritmo = tag : "@algoritmo:"
...

// Delimitadores
inicio : /\*@{Whitespace}*
fim    : \*/
```

Quadro 18 – Trecho da especificação léxica dos *DocComments*

Realizada a especificação léxica, deve-se especificar a sintaxe e a semântica da linguagem de *DocComments*.

3.4.2 Especificação sintática

Os *DocComments* possuem uma construção sintática bastante simples. Iniciam e finalizam com os delimitadores `inicio` e `fim`. Entre os delimitadores podem existir vários *tags*, seguidos por zero ou mais identificadores, conforme pode ser observado no quadro 19 (a especificação completa da linguagem encontra-se no apêndice B).

```
<DocComment> ::= inicio <Comments> fim
<Comments>   ::= <Comment> <Comments>  |  ε
<Comment>    ::= <Tag> <IdList>
<Tag>        ::= algoritmo  |  ...
<IdList>     ::= Id <IdList>  |  ε
```

Quadro 19 – Trecho da gramática da linguagem de *DocComments*

3.4.3 Especificação semântica

A cada *tag* GDC é associada uma ação semântica para classificá-lo. E cada identificador encontrado é associado ao último *tag* reconhecido através da ação semântica `#3`. Algumas ações semânticas associadas às construções sintáticas da linguagem de *DocComments* podem ser observadas no quadro 20 (a gramática completa é apresentada no apêndice B).

```
<DocComment> ::= inicio <Comments> fim
<Comments>   ::= <Comment> <Comments>  | ε
<Comment>    ::= #1 <Tag> #2 <IdList>
<Tag>        ::= algoritmo #10  |  ...
<IdList>     ::= Id #3 <IdList>  |  ε
```

Quadro 20 - Trecho da gramática da linguagem de *DocComments*, com ações semânticas

Caso esteja sendo analisado um *DocComment* que apresenta o algoritmo implementado em uma função, as ações semânticas são executadas na seguinte ordem:

a) ação semântica `#1`: atualizar o controle de fluxo de informações;
b) ação semântica `#10`: indicar que o *tag* identificado é `@algoritmo:`;
c) ação semântica `#2`: indicar que o *tag* não contém identificador associado;
d) ação semântica `#3`: associar o identificador ao último *tag* reconhecido. Esta ação pode ser chamada tantas vezes quantos forem os identificadores associados ao *tag*.

São ao todo dezenove ações (quadro 21), numeradas de `#1` a `#3` e `#10` a `#25`.

AÇÃO SEMÂNTICA	FUNÇÃO
`#1` e `#2`	Controlar fluxo de informações.
`#3`	Associar o identificador ao último *tag* reconhecido.
`#10, #11 e #13 a #25`	Setar o tipo do *tag* reconhecido.
`#12`	Marcar o *DocComment* como sendo o cabeçalho do programa.

Quadro 21 – Ações semânticas da gramática da linguagem de *DocComments*

3.5 ESPECIFICAÇÃO DA LINGUAGEM DE *TEMPLATES*

Para a implementação do GDC, foi analisado o FastTrac (HILL, 2006), motor de *templates* para Delphi. O FastTrac não se mostrou adequado para ser usado no desenvolvimento da ferramenta proposta. Desta forma, optou-se pela especificação e implementação de um motor de *templates*, tomando como base o Velocity.

O primeiro passo para a especificação de um motor de *templates* é a especificação da linguagem utilizada pelo mesmo. Assim como para os *DocComments*, optou-se pela definição de uma linguagem escrita em português e *case sensitive*. A linguagem possui identificadores, componentes, propriedades e comandos. Os identificadores são usados para escrever o código estático do *template*. Os demais elementos permitem a elaboração do código dinâmico.

Os componentes estão relacionados com declarações e diretivas encontradas no código fonte, com o arquivo de código fonte propriamente dito e com *DocComments*. Um componente possui propriedades. Assim, `@comentário` indica o componente comentário e `@comentário.autor` permite acessar a propriedade `autor` do componente `@comentário`, cujo valor será coletado do código fonte C analisado. Cada componente possui um conjunto de propriedades válidas. As propriedades representam as características dos componentes. Pode-se acessar uma propriedade de duas formas: através do componente ao qual está associada (`@comentário.autor`) ou diretamente (`@autor`).

Quanto aos comandos da linguagem de *templates*, são divididos em comandos de:

a) atribuição (`#defina`): permite que se atribua ao componente `@comentário` o valor da propriedade `comentário` de outro componente. Assim, a linha de comando `#defina (@comentário = @include.comentário)` indica ao GDC que, para qualquer referência feita às propriedades de `@comentário`, será acessado o valor

das propriedades relacionadas ao comentário associado à diretiva `#include` que está sendo documentada;

b) repetição (`#para-cada`): permite repetir um trecho de código estático ou dinâmico do *template*, para cada ocorrência de um componente. A linha de comando `#para-cada (@include) ... #fecha` irá repetir o trecho de código subseqüente (até `#fecha`), para cada `#include` existente no código fonte;

c) condicional (`#se`): permite testar a existência de componentes e respectivas propriedades. Caso o resultado do teste seja verdadeiro, o trecho de código subseqüente (até `#fecha`) será analisado.

Na linguagem de *templates* não existe um comando específico para escrever o valor das propriedades dos componentes no arquivo de saída. Assim, sempre que for encontrado um componente acompanhado de uma propriedade, exceto nos três comandos citados anteriormente, o valor correspondente será acessado e apresentado no arquivo de saída.

No quadro 22 é apresentado um trecho de um *template*. A especificação da linguagem de *templates* pode ser observada no apêndice C.

```
#se (@arquivo.possui(@includes))
   <tr>
   #para-cada (@include)
      #defina (@comentário = @include.comentário)
      <tr>
         <td width="773">
         <b>Arquivo incluído: @include.arquivo.@include.extensão
         #se(@comentário.possui(@autor))
            <i><b>Autor: </b> @comentário.autor</i>
         #fecha
   #fecha
#fecha
```

Quadro 22 – Trecho de um *template*

3.5.1 Especificação do modelo de *templates*

O motor de *templates* foi especificado de forma a ser necessária a criação de um arquivo contendo a relação de todos os *templates* (arquivos `.HTML`) que serão usados para gerar a documentação. Esse arquivo com extensão `.TEM` é denominado de modelo de *templates*. O arquivo `.TEM` deve possuir, além dos nomes dos arquivos `.HTML` que pertencem ao referido modelo, o diretório onde estão gravados, assim como o *tag* `#complemento` (quadro 23). Através do *tag* `#complemento`, define-se que o arquivo gerado terá o mesmo nome do arquivo que contém o código fonte C analisado, acrescido da informação contida

entre parênteses.

```
arquivo_frame\index.html         #complemento(_index.html)
arquivo_frame\menu.html          #complemento(_menu.html)
arquivo_frame\conteudo.html      #complemento(_sobre.html)
arquivo_frame\enumeradores.html #complemento(_enumeradores.html)
arquivo_frame\diretivas.html     #complemento(_diretivas.html)
arquivo_frame\funcoes.html       #complemento(_funcoes.html)
arquivo_frame\includes.html      #complemento(_includes.html)
arquivo_frame\structs.html       #complemento(_structs.html)
arquivo_frame\tipos.html         #complemento(_tipos.html)
arquivo_frame\unioes.html        #complemento(_unioes.html)
arquivo_frame\variaveis.html     #complemento(_variaveis.html)
```

Quadro 23 – Estrutura de um arquivo `.TEM`

Sugere-se que os arquivos `.HTML` sejam criados em um diretório com o mesmo nome do arquivo `.TEM` (figura 11).

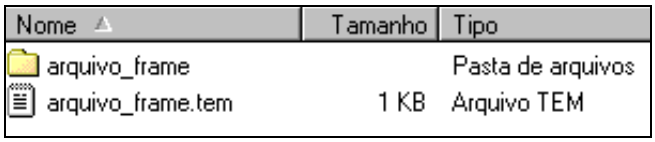


Figura 11 – Estrutura do diretório de um modelo de *templates*

O modelo de *templates*, apresentado no quadro 23, contém onze arquivos HTML, gravados no diretório `arquivo_frame`, conforme pode-se observar na figura 12.

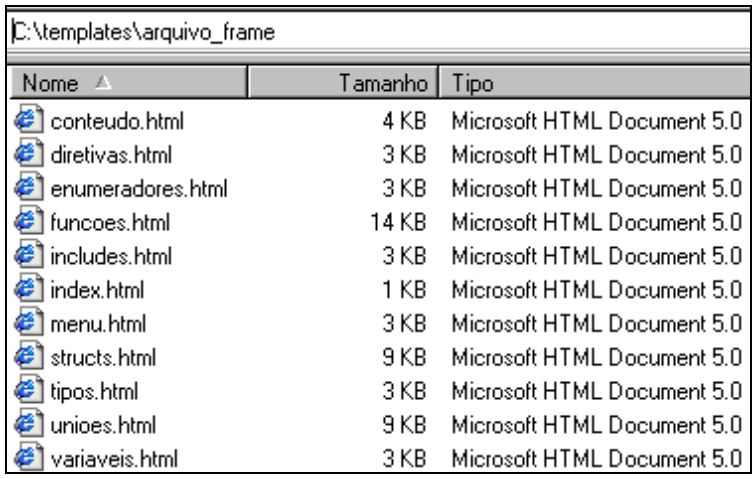


Figura 12 – Arquivos HTML pertencentes a um modelo de *templates*

Para cada modelo de *templates*, existem no mínimo dois arquivos, sendo o primeiro o modelo propriamente dito (arquivo `.TEM`) e, os demais, arquivos `.HTML` com a descrição do formato e os comandos da linguagem de *templates*. O número de arquivos `.HTML` depende do formato desejado para a saída. Pode-se criar apenas um arquivo, que contenha todas as informações necessárias, ou uma estrutura de *frames*, contendo um arquivo para a geração do menu e um para cada tipo de informação contida no arquivo. Durante o processo de análise do modelo de *templates*, o arquivo `.TEM` é carregado e analisado. Cada arquivo `.HTML` relacionado é carregado e tem seu código utilizado para gerar a documentação.

3.5.2 Especificação léxica

Durante a análise léxica da linguagem de *templates*, são reconhecidos propriedades, componentes, instruções e identificadores. As propriedades iniciam com `@`, seguido de uma ou mais letras minúsculas (incluindo `â`, `ã`, `á`, `í`, `ó`, `õ`, `ç`, `\`, `-`). São reconhecidas a partir de um conjunto de trinta e nove casos especiais do *token* `propriedade`. Os componentes, reconhecidos junto de suas respectivas propriedades, também iniciam com `@`, seguido de uma ou mais letras minúsculas, de um . (ponto) e de mais uma seqüência de uma ou mais letras minúsculas. São reconhecidas a partir de um conjunto de quarenta e nove casos especiais do *token* `componente`. As instruções iniciam com #, também seguido de uma seqüência de letras minúsculas, sendo reconhecidos a partir de um conjunto de quatro casos especiais do *token* `instrucao`. Os identificadores são qualquer seqüência de caracteres, exceto quebra de linha, desde que a referida seqüência seja diferente dos casos especiais reconhecidos pela linguagem (quadro 24).

```
// Definições regulares auxiliares
Ascii_     : .
Letra      : [a-zâãáíóõç\-]
Whitespace : [\n\t\s\r]
```

```
// Tokens
propriedade  : @{letra}+
componente   : @{letra}+"."{letra}+
instrucao    : #{letra}+
identificador: {Ascii_}

// Instruções
defina = instrucao: "#defina"
...
// Componentes
comentario_do_arquivo = componente: "@arquivo.comentário"
...
// Propriedades
algoritmo = propriedade : "@algoritmo"
```

Quadro 24 – Trecho da especificação léxica dos *templates*

3.5.3 Especificação sintática

Embora a gramática da linguagem de *templates* seja um pouco mais complexa que a de *DocComments*, também é uma linguagem bastante simples (apêndice C), contendo apenas os comandos específicos para a documentação de programas em C. Um *template* é composto por um ou mais elementos, podendo ser um símbolo, uma propriedade, um componente ou um comando. Parte da gramática que especifica o comando de atribuição é apresentada no quadro

25.

```
<template>    ::= <elemento> <lista>
<lista>       ::= <elemento> <lista>   | ε
<elemento>    ::= <símbolo> | propriedade | componente | <comando>
<comando>     ::= <defina>  |  ...
<defina>      ::= defina ( @comentario = <componente_com_comentario> )
<componente_com_comentario>::= comentario_do_arquivo | ...
```

Quadro 25 – Trecho da gramática da linguagem de *templates*

3.5.3.1 Especificação semântica

Cada comando da linguagem de *templates* possui um conjunto de ações semânticas associadas. Parte das ações semânticas associadas ao comando de atribuição é apresentada no quadro 26, e um trecho de um arquivo HTML, onde é utilizado o referido comando, no quadro 27.

```
<defina> ::= defina ( @comentario = <componente_com_comentario> ) #5
<componente_com_comentario>::= comentario_do_arquivo #90 | ...
```

Quadro 26 – Trecho da gramática da linguagem de *templates*, com ações semânticas

```
...
  </tr>
  <tr>
    <td width="782"><font size="3">
      #defina (@comentário = @arquivo.comentário)
      <b>Descrição</b>:</font></td>
  </tr>
  <tr>
...
```

Quadro 27 – Trecho de um arquivo HTML, com um comando de atribuição

Para o comando de atribuição apresentado no quadro 27, as ações semânticas são chamadas na seguinte ordem:

a) ação semântica `#90`: setar a propriedade comentário com as informações do comentário (cabeçalho) do arquivo;
b) ação semântica `#5`: setar a posição do fim do comando.

A linguagem de *templates* possui um total de noventa e quatro ações semânticas (quadro 28), numeradas de `#1` a `#3`, `#5`, `#8` a `#45`, `#50` a `#83`, `#90` a `#98` e `#100` a `#107`. O grande número de ações, se justifica pela grande variedade de informações armazenadas, que podem se utilizadas na documentação.

AÇÃO SEMANTICA	FUNÇÃO
#1, #3, #5, #8, #9 e #19	Controlar fluxo de informações.
#2	Processar comando #se.
#10 - #16	Definir o tipo do componente analisado no comando #se.
#17	Processar comando #para-cada.
#18	Atualizar as informações do arquivo de saída.
#20 - #45	Definir o tipo da propriedade testada no comando #se.
#50 – 83	Atualizar as informações do arquivo de saída, incluindo o valor das propriedades.
#90 - #98	Associar a @comentário o valor da propriedade comentário de um determinado componente.
#100 - #107	Definir o tipo do componente analisado no comando #para-cada

Quadro 28 – Ações semânticas da gramática da linguagem de *templates*

3.6 ESPECIFICAÇÃO DA FERRAMENTA

O GDC foi especificado através da ferramenta Entreprise Architect, utilizando os conceitos de orientação a objetos e baseando-se nos diagramas da UML, gerando como produtos os diagrama de caso de uso e classes.

3.6.1 Casos de uso

O GDC possui três casos de uso (figura 13). Não existe uma seqüência obrigatória para execução dos casos de uso, visto que os mesmos são independentes uns dos outros.

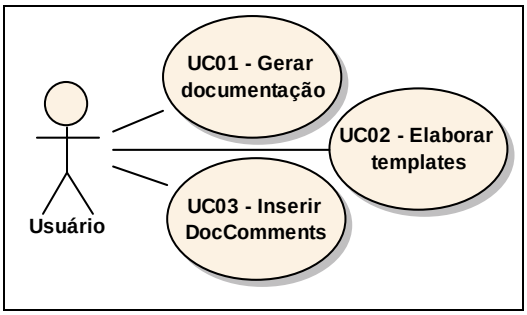


Figura 13 – Diagrama de casos de uso

O primeiro caso de uso (quadro 29), designado de Gerar documentação, pode também ser considerado o caso de uso principal do GDC, visto que é durante sua execução que a documentação do código fonte é gerada. Ele possui, além do cenário principal, dois cenários alternativos, dando ao usuário maior controle sobre o processo de documentação, e

três cenários de exceção, informando possíveis erros encontrados durante o processo.

UC01 – Gerar documentação: através deste caso de uso é possível gerar a documentação de um programa escrito em C, permitindo também que o usuário selecione o modelo de *templates* desejado.	
Pré-condições	O arquivo escrito em linguagem C, deve estar léxica e sintaticamente correto. O modelo de *templates* deve estar léxica, sintática e semanticamente correto.
Cenário principal	01) O GDC apresenta as opções para carregar arquivo/projeto e selecionar diretório para salvar a documentação a ser gerada. 02) O usuário seleciona o arquivo/projeto a ser aberto e o diretório onde deseja salvar os arquivos de saída. 03) O GDC apresenta a relação de arquivos/projetos abertos. 04) O usuário seleciona o arquivo/projeto aberto para o qual deseja gerar a documentação. 05) O GDC analisa o código fonte do arquivo/projeto a ser documentado, coleta e armazena as informações necessárias para a geração da documentação. 06) O GDC analisa e processa o modelo de *templates* (selecionado pelo usuário ou *default* da ferramenta), gerando os arquivos com a documentação. 07) O GDC apresenta a documentação gerada.
Fluxo alternativo 01	No passo 02: 02.1) O usuário opta por abrir os últimos arquivos/projetos anteriormente processados. 02.2) O GDC apresenta a relação os dez últimos arquivos/projetos processados.
Fluxo alternativo 02	No passo 04, após selecionar o arquivo/projeto a ser documentado, caso o usuário queira selecionar o modelo de *templates* a ser usado para gerar a documentação: 04.1) O GDC apresenta a relação de modelos de *templates* existentes. 04.2) O usuário seleciona um dos modelos de *templates* apresentados.
Exceção 01	No passo 05, caso seja encontrado algum erro no código fonte: 05.1) O GDC apresenta mensagem de erro e encerra a geração de documentação.
Exceção 02	No passo 05, caso seja encontrado algum erro em um *DocComment*: 05.1) O GDC apresenta o *DocComment* contendo o erro, identificando o erro e sua posição, e encerra a geração de documentação.
Exceção 03	No passo 06, caso seja encontrado algum erro léxico, sintátio ou semântico no modelo de *templates*: 06.1) O GDC apresenta o arquivo HTML (do modelo de *templates*), identificando o erro e sua posição, e encerra a geração de documentação.
Pós-condições	A documentação do programa C deve ter sido gerada.

Quadro 29 – Caso de uso 01

O segundo caso de uso (quadro 30), designado `Elaborar templates`, descreve o processo de criação dos *templates*. Ele possui um cenário principal e três cenários alternativos, não possuindo nenhum cenário de exceção, visto que suas construções léxica, sintática e semântica não são analisadas durante a criação/edição. Possíveis erros nos modelos de *templates* são apresentados no momento da seleção de tais modelos de *templates*, ou no momento da geração da documentação.

UC02 – Elaborar *templates*: permite criar/editar modelos de *templates*.	
Pré-condições	Não há pré-condições.
Cenário principal	01) O GDC apresenta a opção para elaborar *templates*. 02) O usuário cria o arquivo, podendo ser um modelo (arquivo `.TEM`) ou um *template* (arquivo .HTML), e manda salvá-lo. 03) O GDC solicita nome, extensão e diretório do arquivo a ser salvo. 04) O usuário fornece as informações solicitadas. 05) O GDC salva o arquivo.
Fluxo alternativo 01	No passo 02, caso o usuário queira editar um arquivo existente: 02.1) O GDC apresenta a relação de arquivos existentes. 02.2) O usuário seleciona o arquivo desejado (um modelo ou um *template*). 02.2) O GDC apresenta o arquivo selecionado. 02.3) O usuário edita o arquivo, manda salvá-lo e segue para o passo 05.
Fluxo alternativo 02	No passo 02, caso o usuário queira usar o editor de comandos da linguagem de *templates*: 02.1) O usuário seleciona a posição onde deseja incluir o comando. 02.2) O GDC apresenta as opções para edição de comandos. 02.3) O usuário seleciona o comando desejado. 02.4) O GDC insere o comando editado e retorna para o passo 02.
Fluxo alternativo 03	No passo 02, caso o usuário queira salvar um arquivo existente com outro nome ou em outro diretório: 02.1) O GDC solicita nome, extensão e diretório do arquivo a ser salvo. 02.2) O usuário fornece as informações solicitadas. 02.3) O GDC salva o arquivo.
Pós-condições	Um modelo ou um *template* deve ter sido criado/editado.

Quadro 30 – Caso de uso 02

O terceiro caso de uso, designado `Inserir DocComments`, descreve o processo de inserção de *DocComments* no código fonte. Ele possui um cenário principal e três cenários alternativos (quadro 31). Não possui cenário de exceção, visto que os *DocComments* inseridos não são validados nesta fase, mas no momento da geração da documentação.

UC03 – Inserir *DocComments*: permite a inserção de *DocComments* no código fonte.	
Pré-condições	Não há pré-condições.
Cenário principal	01) O GDC apresenta a opção para carregar arquivo. 02) O usuário seleciona o arquivo no qual deseja inserir *DocComments*. 03) O GDC apresenta o arquivo selecionado e a opção para inclusão de *DocComments*. 04) O usuário seleciona a posição onde deseja incluir o *DocComment*. 05) O GDC solicita as informações necessárias para a edição do *DocComment*. 06) O usuário fornece as informações solicitadas. 07) O GDC inclui o *DocComment* na posição indicada. 08) O usuário manda salvar o arquivo. 09) O GDC salva o arquivo.
Fluxo alternativo 01	No passo 05, caso o usuário queira digitar o *DocComment*: 05.1) O usuário digita o *DocComment* e segue para o passo 08.
Fluxo alternativo 02	No passo 08, caso o usuário queira inserir mais *DocComments*: 08.1) O usuário seleciona a opção para inclusão de *DocComments* e vai para o passo 04.
Fluxo alternativo 03	No passo 08, caso o usuário queira digitar mais *DocComments*: 08.1) O usuário retorna para o passo 04, digita o *DocComment* e segue para o passo 08.
Pós-condições	Um arquivo com código fonte C deve ter *DocComments* inseridos.

Quadro 31 – Caso de uso 03

3.6.2 Diagrama de classes

O diagrama de classe apresenta uma visão de como as classes estão estruturadas e relacionadas. Em função da quantidade de classes e do grande número de relacionamentos existentes, na figura 14 são apresentados os pacotes (*packages*) detalhados nesta seção.

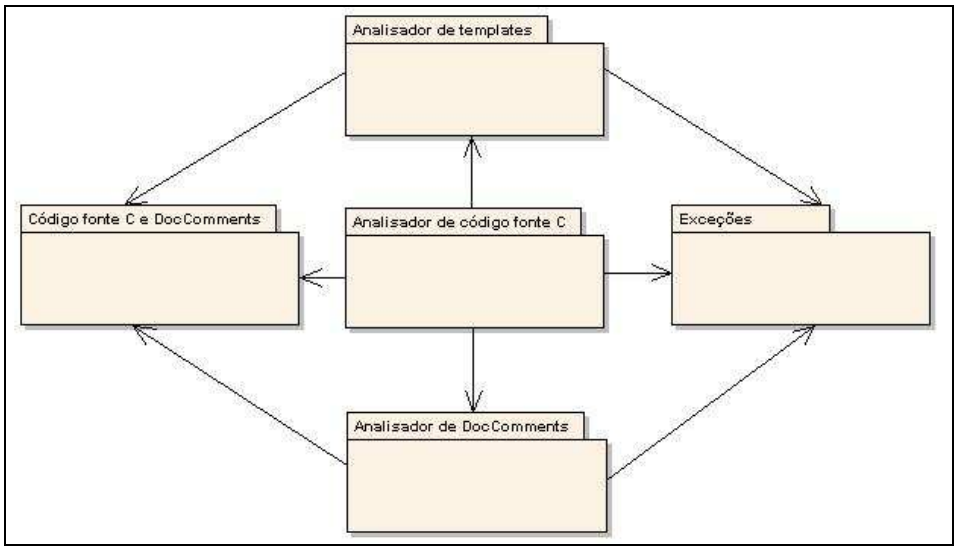


Figura 14 – Pacotes

Os pacotes foram gerados de acordo com a ligação lógica existente entre as classes que

os compõem e são explicados na seqüência. O primeiro pacote apresentado é o que contém as classes responsáveis pela análise do código fonte C, incluindo coleta e armazenamento das informações utilizadas para gerar a documentação (figura 15).

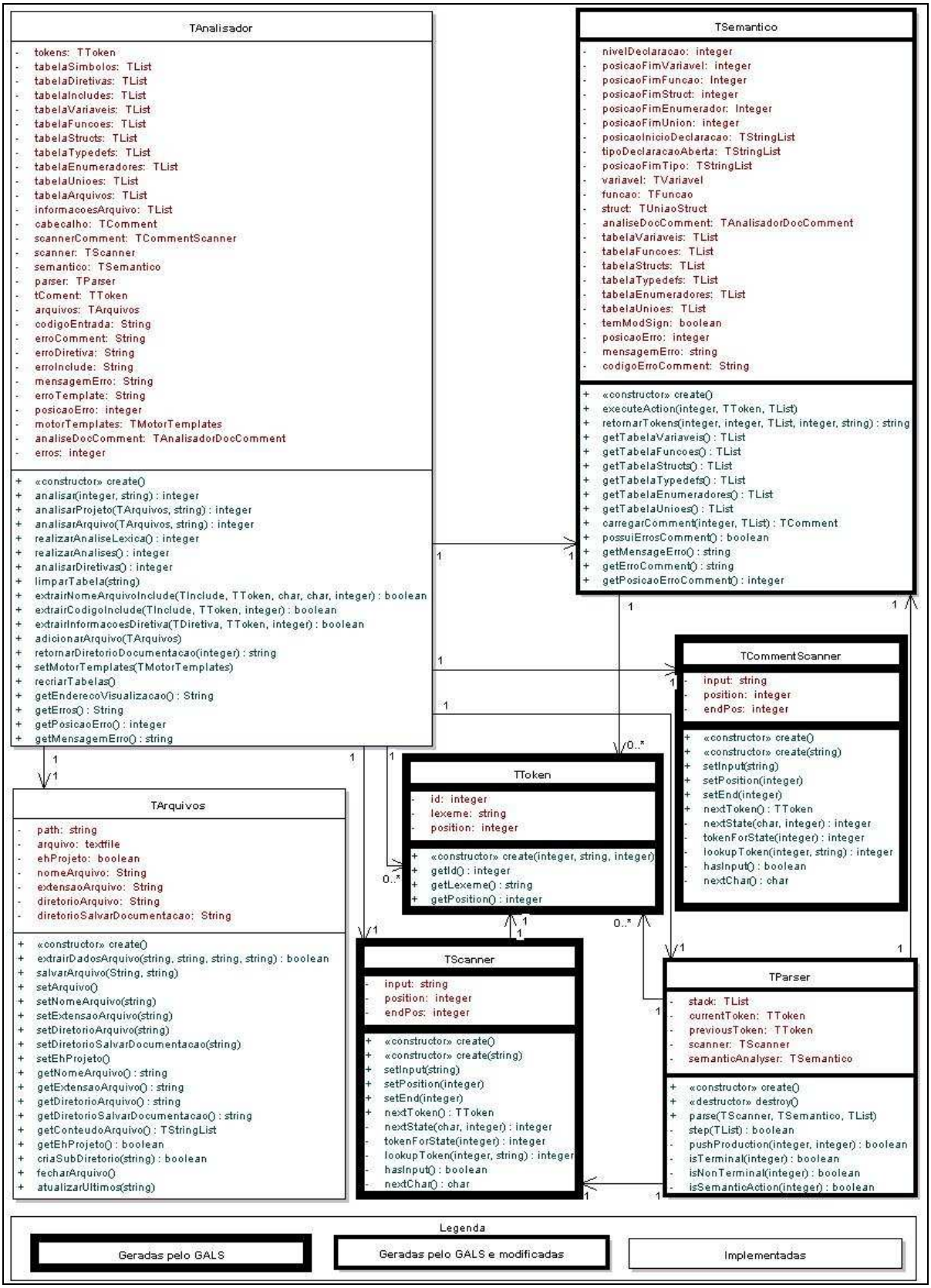


Figura 15 – Diagrama de classes do pacote `Analisador de código fonte C`

A classe principal deste pacote é `TAnalisador`, responsável por coletar as informações durante as análises léxica, sintática e semântica, e coordenar a geração da documentação dos arquivos analisados. Seus principais métodos são apresentados no quadro 32.

MÉTODO	DESCRIÇÃO
`analisarProjeto`	Gera a documentação de projetos. Para tanto, analisa o arquivo descritor do projeto e realiza tantas chamadas ao método `analisarArquivo`, quantos forem os arquivos do projeto, passando o referido arquivo como parâmetro.
`analisarArquivo`	Gera a documentação de arquivos.
`realizarAnaliseLexica`	Realiza a análise léxica e a geração da tabela de símbolos.
`analisarDiretivas`	Analisa cada diretiva encontrada na tabela de símbolos. Também realiza a análise do cabeçalho do arquivo.
`realizarAnalises`	Realiza as análises sintática e semântica.

Quadro 32 – Principais funções da classe `TAnalisador`

A classe `TCommentScanner` é responsável pela análise léxica, contendo na tabela de símbolos todos os *tokens* reconhecidos, inclusive os comentários. A classe `TScanner` ignora os comentários, o que a diferencia da classe `TCommentScanner`. Durante a análise léxica, utiliza-se de um objeto da classe `TToken`. Esse objeto é criado sempre que um *token* for reconhecido no código fonte. Possui como atributos a classe, a posição e o *token* propriamente dito.

A classe `TParser` comanda todo o processo de análise, solicitando *tokens* à classe `TScanner` e ativando métodos da classe `TSemantico` para a extração das declarações contidas no código fonte. Caso algum erro ocorra durante a execução dos métodos das referidas classes, as classes do pacote de `Exceção` são chamadas.

Por último, a classe `TArquivos` é responsável por todas as operações relacionadas a arquivos, tais como manter as informações dos arquivos; abri-los; fechá-los e gerar os arquivos de documentação.

A análise do código fonte C para a coleta das informações a serem utilizadas na documentação, acontece de acordo com as atividades expressas no diagrama de atividades apresentado na figura 16.

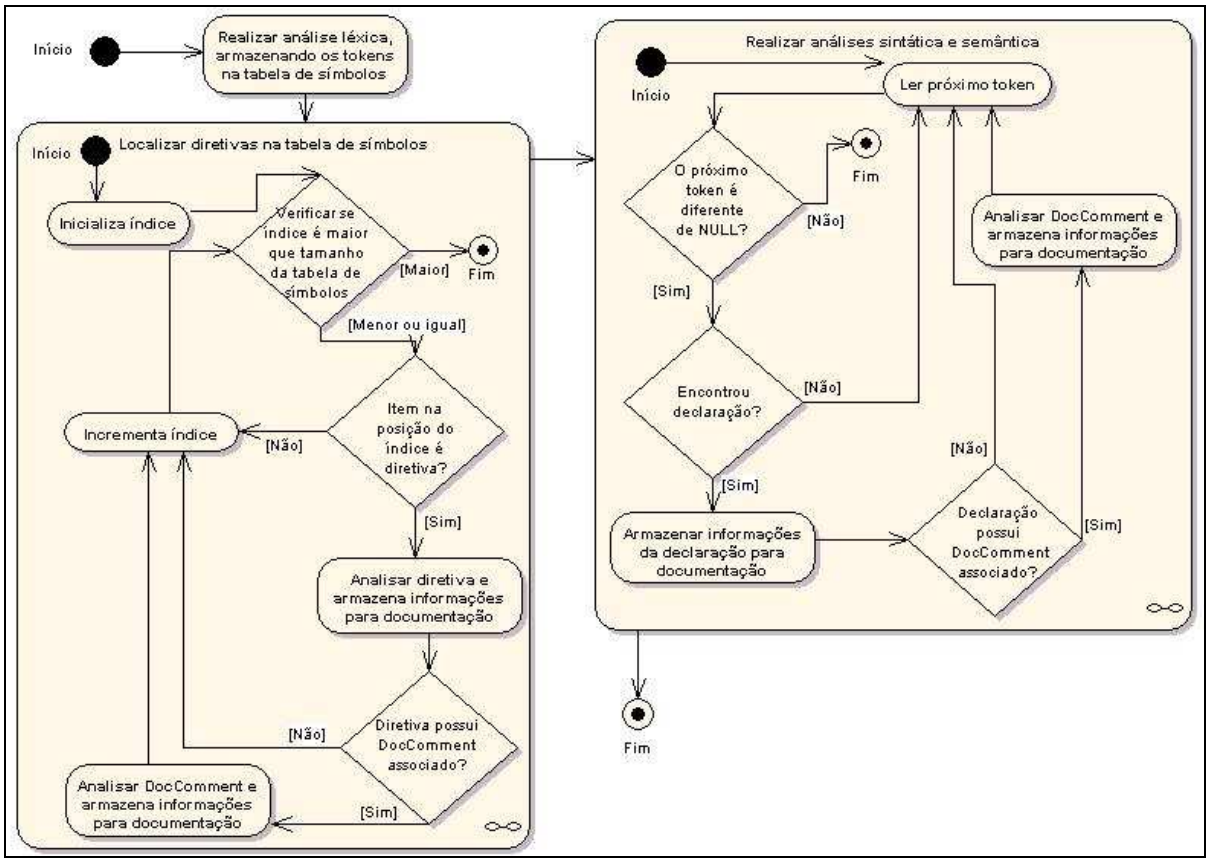


Figura 16 – Diagrama de atividades do `Analisador de código fonte C`

O segundo pacote (figura 17), composto pelas classes `TAnalisadorDocComment`, `TTagToken`, `TTagScanner`, `TTagParser` e `TTagSemantico`, é responsável pela análise e extração das informações contidas nos *DocComments*. As classes `TTagToken`, `TTagScanner` e `TTagParser` são semelhantes e com métodos idênticos aos das classes `TToken`, `TScanner` e `TParser` apresentadas no primeiro pacote, diferindo-se basicamente pela descrição dos *tokens* e pelas construções gramaticais analisadas por cada classe. Observa-se que não foram utilizados os princípios da orientação a objetos para herança e polimorfismo, visto que as classes para as análises léxica e sintática são geradas automaticamente pelo GALS. A classe `TTagSemantico` possui métodos para cada uma das ações semânticas especificadas na seção 3.4.3.

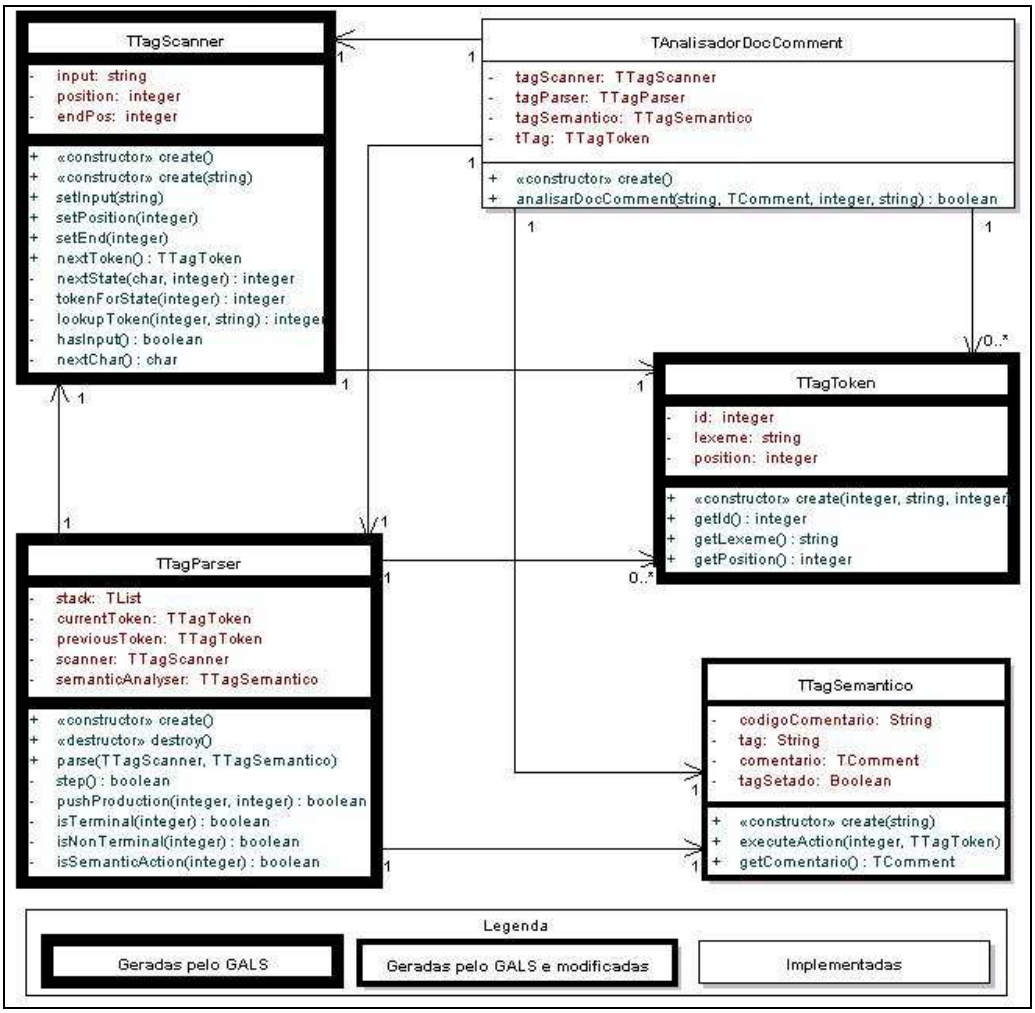


Figura 17 - Diagrama de classes do pacote `Analisador de DocComments`

A classe `TAnalisadorDocComment`, principal classe do pacote, é responsável por comandar o processo de análise dos *DocComments*. Além de seu construtor, possui ainda um método (`analisarDocComment`) responsável por controlar a análise dos *DocComments*, retornando um objeto de `TComment`, com as informações contidas no *DocComments* analisado.

O terceiro pacote (figura 18), denominado `Analisador de templates`, é responsável por analisar os modelos de *templates*, substituir as referências definidas nos *templates* pelas informações coletadas do código fonte C analisado e gerar os arquivos HTML com a documentação. Através das classes `TTemplateToken`, `TTemplateScanner`, `TTemplateParser` e `TTemplateSemantico`, realiza as análises léxica, sintática e semântica dos *templates* (arquivos `.HTML`) pertencentes aos modelos de *templates* (arquivos `.TEM`). As referidas classes têm métodos e atributos semelhantes aos citados nos pacotes anteriores, diferindo-se pela descrição dos *tokens* e pelas construções gramaticais analisadas. A classe `TTemplateSemantico` implementa as funcionalidades das ações semânticas especificadas para a linguagem de *templates*.

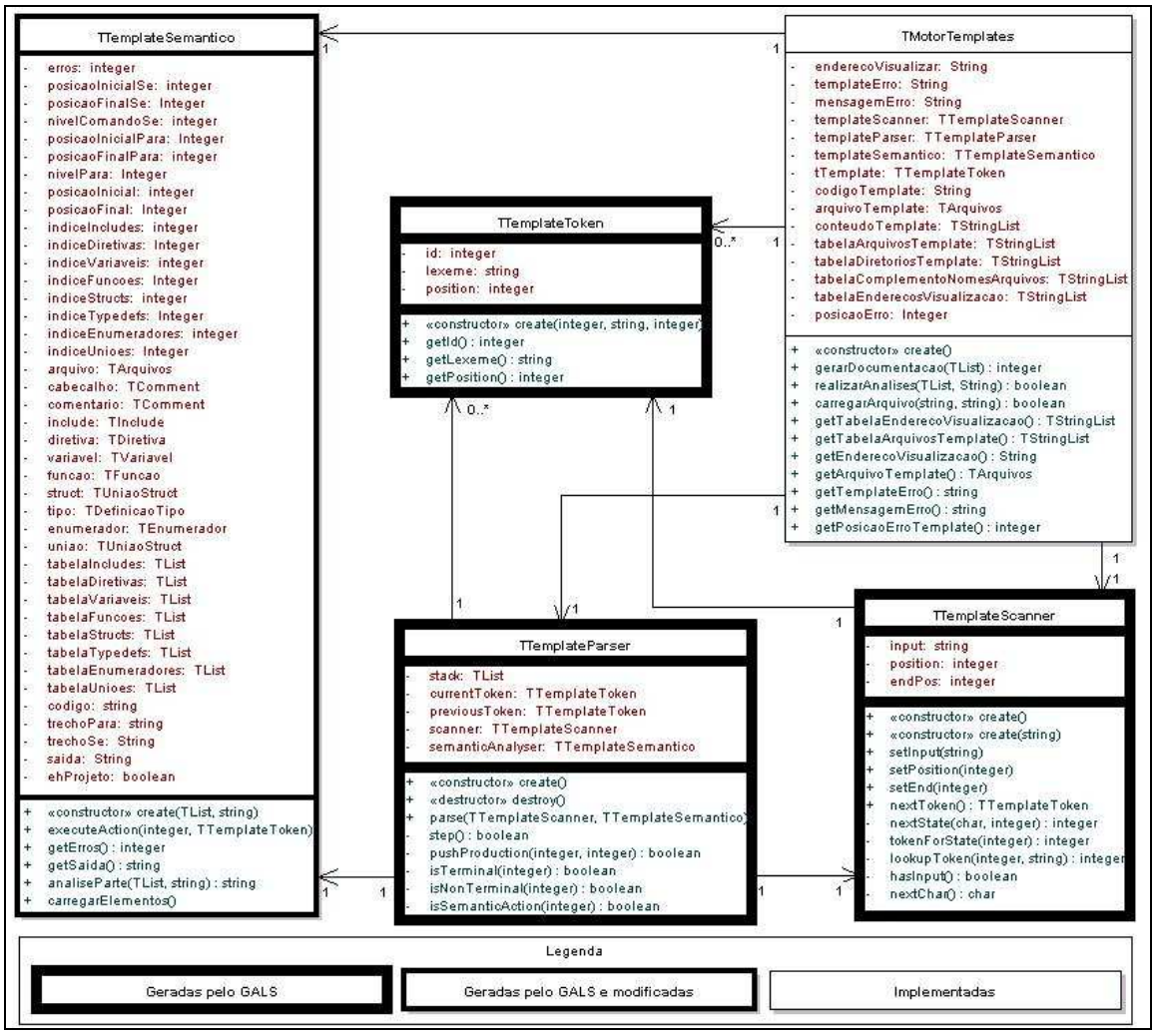


Figura 18 - Diagrama de classes do pacote `Analisador de templates`

A classe `TMotorTemplates` é responsável pelo controle do processo de análise dos arquivos HTML pertencentes aos modelos de *templates*. Seu principal método (`gerarDocumentacao`) instancia objetos das classes `TTemplateToken`, `TTemplateScanner`, `TTemplateParser` e `TTemplateSemantico`, que realizam as análises léxica, sintática e semântica de cada arquivo HTML associado ao modelo de *templates* utilizado para gerar a documentação.

O quarto pacote (figura 19) é composto por classes que possuem as informações coletadas durante a análise do código fonte C e dos *DocComments*. Os atributos de cada uma das classes são descritos no apêndice D.

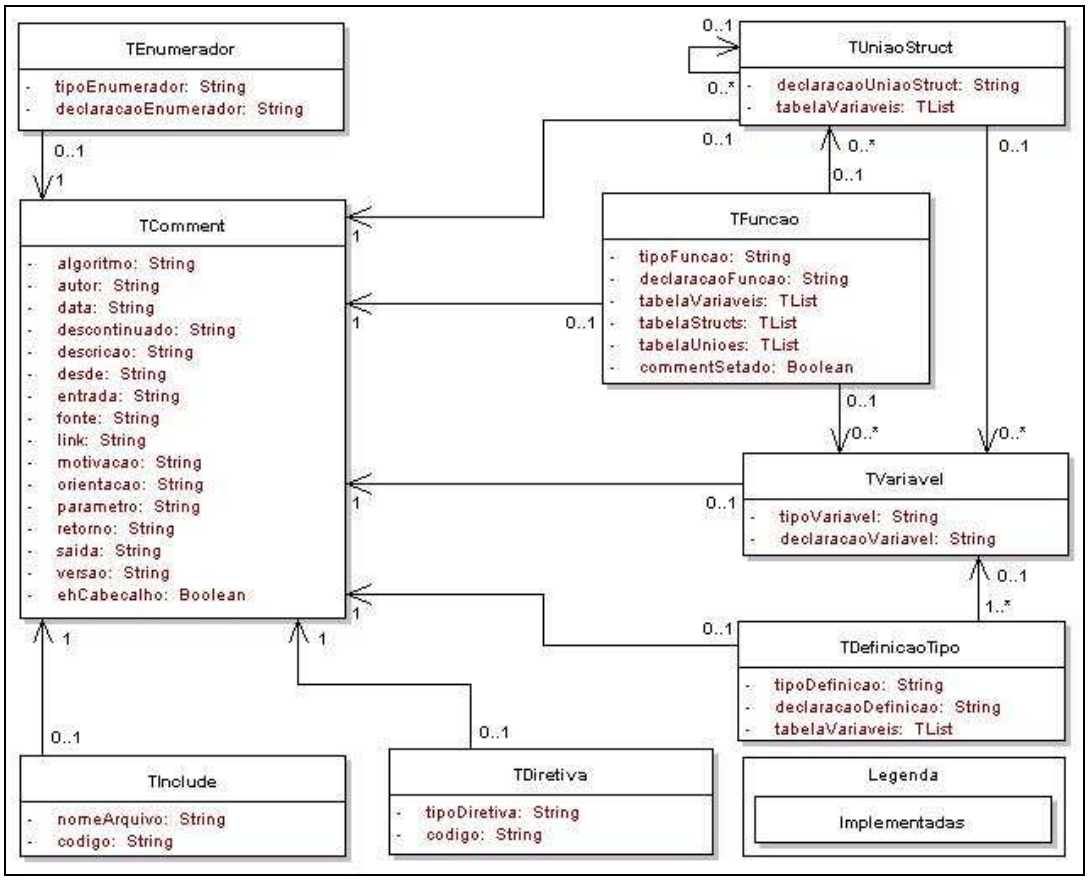


Figura 19 – Diagrama de classes do pacote de `Código fonte C e DocComments`

Foram especificadas as seguintes classes:

a) `TComment`: classe que possui informações pertinentes aos comentários. Tais comentários podem estar associados ao cabeçalho do programa, às diretivas ou às declarações;

b) `TDefinicaoTipo`: classe que possui informações de definições de tipo (`typedef`);

c) `TDiretiva`: classe que possui informações referentes às diretivas (`define`, `undef`, `if`, etc.), excluindo-se as diretivas `include` que têm suas informações armazenadas em classe própria;

d) `TEnumerador`: classe que possui informações referentes aos enumeradores (`enum`);

e) `TFuncao`: classe que possui informações referentes às declarações de funções;

f) `TInclude`: classe que possui informações referentes às inclusões de arquivos (`include`);

g) `TUniaoStruct`: classe que possui informações referentes às declarações de estruturas (`struct` e `union`);

h) `TVariavel`: classe que possui informações referentes às declarações de variáveis existentes no código fonte.

O quinto pacote (figura 20), denominado `Exceções`, como o próprio nome sugere, é

responsável pelo tratamento de erros ocorridos durante os processos de análise dos programas C, dos *DocComments* e dos arquivos HTML associados aos modelos de *templates*. Caso algum erro seja encontrado, objetos das classes `ELexicalError`, `ESyntaticError` e `ESemanticError` são criados para efetuar o diagnóstico do erro ocorrido. Tais classes podem ser chamadas pelas classes `TAnalisador`, `TAnalisadorDocComment` e `TMotorTemplates`.

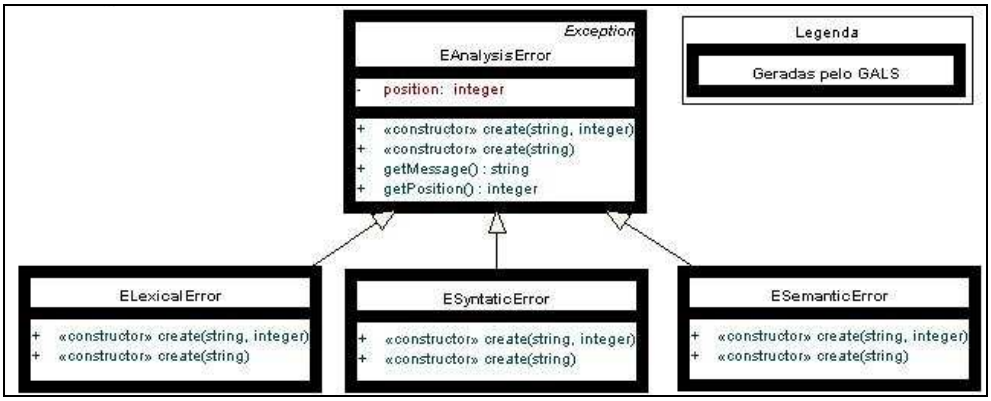


Figura 20 - Diagrama de classes do pacote `Exceções`

3.7 IMPLEMENTAÇÃO

Nesta seção são apresentadas informações sobre as técnicas e ferramentas utilizadas para a implementação do GDC, assim como o próprio processo de implementação.

3.7.1 Técnicas e ferramentas utilizadas

O GDC foi implementado na linguagem *Object Pascal*, utilizando o ambiente de desenvolvimento Delphi 7.0. Os três conjuntos de analisadores léxicos e sintáticos (para a análise dos arquivos de código fonte C, dos *DocComments* e dos arquivos HTML associados aos modelos de *templates*) foram gerados pelo GALS a partir das especificações apresentadas nos apêndices A, B e C.

3.7.2 Implementação do GDC

Para facilitar o entendimento, o processo de implementação do GDC foi dividido em etapas. Em um primeiro momento são apresentadas características da implementação dos analisadores do código fonte C e dos *DocComments*. Na seqüência são descritas a análise dos *templates* e a geração da documentação.

3.7.2.1 Analisadores do código fonte C e dos *DocComments*

O GDC utiliza três conjuntos de analisadores léxico, sintático e semântico, sendo um para a análise do código fonte C, outro para os *DocComments* e outro para os modelos de *templates*. Os analisadores léxico e sintático foram gerados pelo GALS, já os analisadores semânticos são parcialmente gerados pela referida ferramenta, sendo que os métodos que implementam as ações semânticas são implementados manualmente.

Para análise do código fonte C, foram geradas as classes `TToken`, `TSemantico`, `TParser` e `TScanner`, além das classes do pacote `Exceções`. Considerando que o analisador léxico deve reconhecer os comentários, para que estes sejam utilizados na documentação, foi necessário gerar outro analisador léxico (`TCommentScanner`) que tem a função de armazenar os *tokens* reconhecidos na tabela de símbolos, utilizada durante todo o processo de coleta de informações. O processo de análise do código fonte C para reconhecimento dos *tokens* e geração da tabela de símbolos é invocado pelo método `realizarAnaliseLexica` da classe `TAnalisador` (quadro 33).

```
function TAnalisador.realizarAnaliseLexica: integer;
begin
  ...
  try
    tComment:= ScannerComment.nextToken;
    while (tComent <> nil) do begin // para cada token reconhecido
      ...
      tabelaSimbolos.Add(token);    // acrescentar na tabela de símbolos
    end;
    result:= 0;
  except  // tratamento erros
    on tComent: ELexicalError do begin
      result:= 1;
      tabelaSimbolos.Destroy;
    end;
 end;
 ScannerComment.destroy;
end;
```

Quadro 33 – Trecho de código do método `realizarAnaliseLexica`

Finalizada a análise léxica, todas as diretivas encontradas na tabela de símbolos são analisadas e têm suas informações armazenadas para a documentação. Na seqüência são chamadas as análises sintática e semântica (quadro 34). Em função do analisador semântico (`TSemantico`) precisar conhecer a tabela de símbolos criada na primeira etapa da análise pela classe `TCommentScanner`, a classe `TParser` foi alterada visando poder receber a tabela de símbolos como um parâmetro, e possibilitando passar a referida tabela para a classe `TSemantico`.

```
function TAnalisador.realizarAnalises: integer;
var
  token: TToken;
begin
  // criar objetos para análise sintática e semântica
  scanner  :=  TScanner.create;
  parser   := TParser.create;
  semantico:= TSemantico.create;
  scanner.setInput(codigoEntrada);  // setar a entrada do léxico
  try
    parser.parse(scanner, semantico, tabelaSimbolos);
    ...
  except  // tratamento erros
    ...
  end;
  // destruir objetos utilizados
  scanner.destroy;
  parser.destroy;
  semantico.destroy;v
end;
```

Quadro 34 – Trecho de código do método `realizarAnalises`

Para análise dos *DocComments* foram geradas, pelo GALS, as classes `TTagToken`, `TTagScanner`, `TTagParser` e `TTagSemantico`, sendo que a classe `TTagSemantico` teve seus métodos implementados manualmente. O processo de chamada dos analisadores é semelhante ao da análise do código fonte C.

3.7.2.2 Análise dos *templates* e geração da documentação

A implementação do motor de *templates* é bastante parecida com a descrição dos demais analisadores. Para ele foram geradas as classes `TTemplateToken`, `TTemplateScanner`, `TTemplateParser` e `TTemplateSemantico`. As três últimas classes são responsáveis, respectivamente, pelas análises léxica, sintática e semântica. Assim como na implementação dos analisadores do código fonte C, o analisador sintático (`TTemplateParser`) dos *templates* também teve que ser alterado, de forma a receber como parâmetro a tabela de símbolos contendo as informações para gerar a documentação. Na análise dos *templates*, cabe à classe `TTemplateSemantico` acionar os métodos correspondentes às ações semânticas.

O processo de análise é iniciado com a criação de um objeto da classe `TMotorTemplates`. Este objeto recebe as informações coletadas e armazenadas durante as análises do código fonte C e dos *DocComments*, e é responsável por substituir as referências definidas no modelo de *templates* selecionado pelas respectivas informações, gerando o(s) arquivo(s) de saída. Para efetuar a substituição do código dinâmico pelas informações coletadas, é necessário carregar cada *template* em um objeto da classe `TArquivo` (objeto este instanciado pelo motor de *templates*). Tais objetos têm seu conteúdo, juntamente com as informações referentes à documentação, encaminhados para as análises léxica, sintática e semântica. As análises léxica e sintática são similares aos analisadores anteriormente citados. Já o analisador semântico possui algumas particularidades descritas a seguir.

Ao iniciar a análise semântica de um *template*, é setada uma variável (`posicaoInicial`) que indica a posição do início dos identificadores reconhecidos. No momento em que um comando é encontrado, é setada a posição final dos identificadores (ação semântica `#18`), conhecendo-se assim o trecho de código estático que deve ser copiado para a saída (quadro 35). Após analisar o comando, a variável `posicaoInicial` é novamente setada, a fim de aguardar pelo próximo comando para ser atualizada (ação semântica `#5`).

```
5 : begin
      if (not seAtivo) and (not paraAtivo) then
        posicaoInicial:= token.getPosition + length(token.getLexeme);
    end;
...
18: begin
      ...
      if (not seAtivo) and (not paraAtivo) then
        posicaoFinal:= token.getPosition+1;
      ...
    end;
```

Quadro 35 – Trecho de código das ações semânticas `#5` e `#18`

Cada tipo de comando da linguagem é analisado de maneira diferente. Quando um componente com uma propriedade associada é encontrado no *template*, é substituído pela informação que está referenciando. A cada combinação possível (componente-propriedade) está associada uma ação semântica. No momento em que a ação semântica é invocada, o método correspondente é chamado, sendo o resultado escrito no arquivo de saída, junto com conteúdo existente entre a `posicaoInicial` e a `posicaoFinal` setadas nas variáveis de controle. Através da ação semântica `#72`, a função `getTipoFuncao`, de um objeto da classe `TFuncao`, é chamada retornando o tipo da função que é concatenado ao atributo da classe `TTemplateSemantico` que armazena a saída que está sendo gerada.

A ação semântica associada ao comando de atribuição (`#defina`) faz com que a um objeto da classe `TComment` seja setado o comentário referenciado (quadro 36). Através da ação

semântica #98, por exemplo, a função getComment retorna um objeto da classe TComment, cujo conteúdo é o *DocComment* associado à declaração de variáveis que está sendo analisada.

```
98  : begin
         if (not seAtivo) and (not paraAtivo) then
            comentario:= variavel.getComment;
      end;
```

Quadro 36 – Trecho de código das ações semânticas #98

Os comandos de repetição e condicional utilizam recursividade para a análise de seus conteúdos. Todo o conteúdo do comando (em negrito no quadro 37), sejam identificadores e/ou outros comandos, é atribuído a uma variável do tipo *string*.

```
#para-cada (@include)
   #defina (@comentário = @include.comentário)
   <tr> <td width="773">
      <font size="3">
         <b>Arquivo incluído:@include.arquivo.@include.extensão </b>
      </font><br>
      #se (@comentário.possui(@autor))
        <font size="2">
           <i><b>Autor: </b> @comentário.autor</i></font><br>
      #fecha
      ...
   </td></tr>
#fecha
```

Quadro 37 – Trecho de um *template*, evidenciando o conteúdo do comando #para-cada

Para analisar o conteúdo dos comandos de repetição e condicional são criados outros objetos das classes TTemplateScanner, TTemplateParser e TTemplateSemantico. Esses objetos recebem como parâmetro o conteúdo dos referidos comandos. Observa-se que podem existir comandos #para-cada e #se dentro de outros comandos do mesmo tipo. O resultado da análise de cada trecho é concatenada com a saída da análise que a invocou, e esta recursivamente, até a mais externa. No caso dos comandos condicionais, primeiro é avaliada a expressão apresentada no comando. Caso a resposta seja afirmativa,é realizada a análise ao seu conteúdo. Caso contrário, o conteúdo é ignorado, e a análise continua na posição após o #fecha correspondente Para os comandos de repetição, a cada execução é verificado se existe mais um componente (dos componentes referenciados no comando) para ser analisado. Caso exista, a análise é invocada, caso contrário, o analisador passa o controle da análise do template para a posição após o fechamento do referido comando.

Finalizado o processo de análise dos comandos, os identificadores encontrados após o último comando são incluídos no atributo da classe TTemplateSemantico que armazena a saída que está sendo gerada .

3.7.3 Operacionalidade da implementação

Nesta seção é apresentada a operacionalidade do GDC. A ferramenta é apresentada na figura 21, possuindo algumas informações básicas e botões para operações com arquivos, com modelos de *templates*, inserção de *DocComments*, entre outros.

Figura 21 – Interface principal do GDC

Através do botão `Arquivos`, é possível acessar a interface que permite carregar arquivos/projetos e selecionar o diretório para salvar a documentação a ser gerada (figura 22).

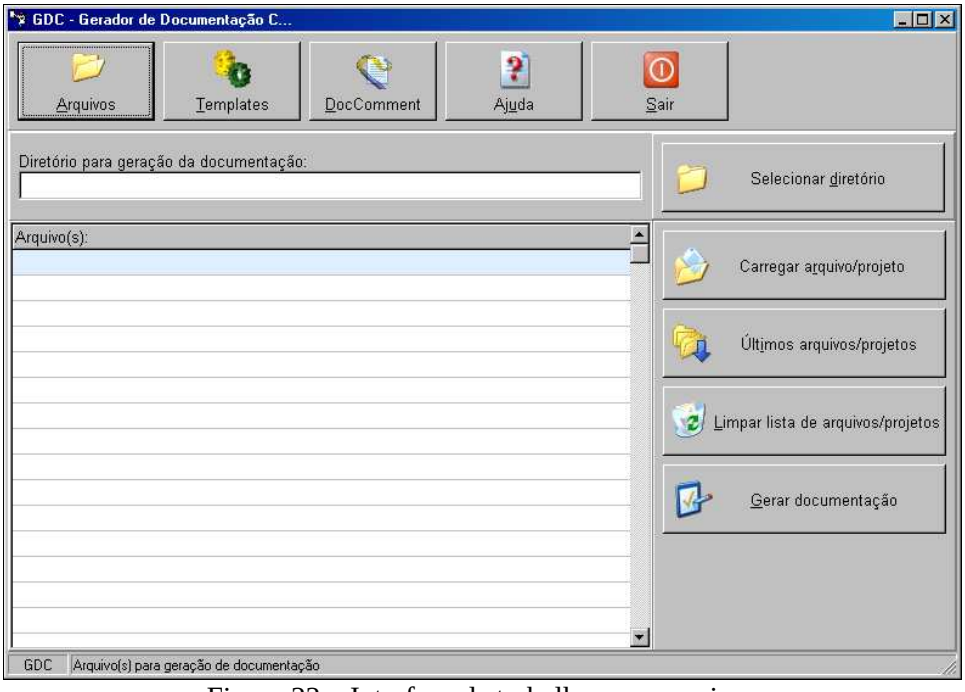


Figura 22 – Interface de trabalho com arquivos

Ao pressionar o botão `Selecionar diretório`, é apresentada a janela da figura 23.

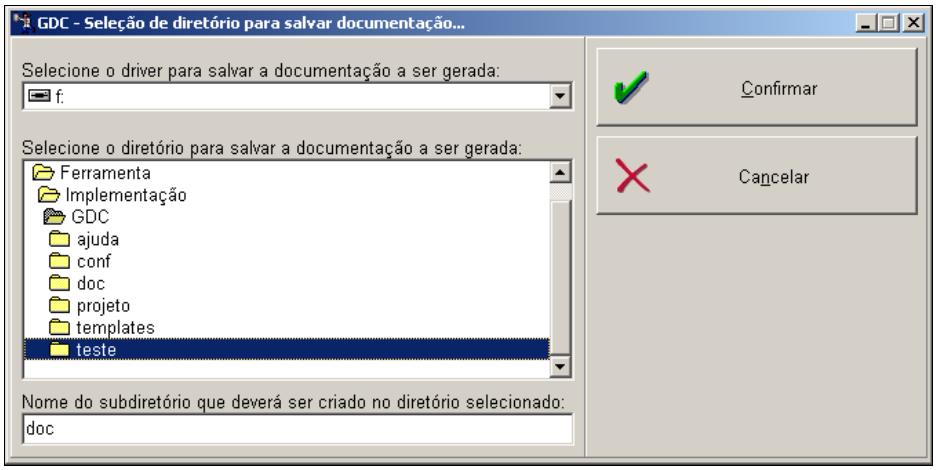


Figura 23 – Janela de seleção de diretório para documentação

O usuário deve selecionar a unidade de disco e o diretório onde deseja salvar os arquivos de saída. Também é possível informar o nome da pasta que deve ser criada para o armazenamento da documentação. Caso o botão `Confirmar` seja pressionado, a janela é fechada e o diretório selecionado é setado (figura 24). De outra forma, a janela é fechada e as ações realizadas são canceladas.

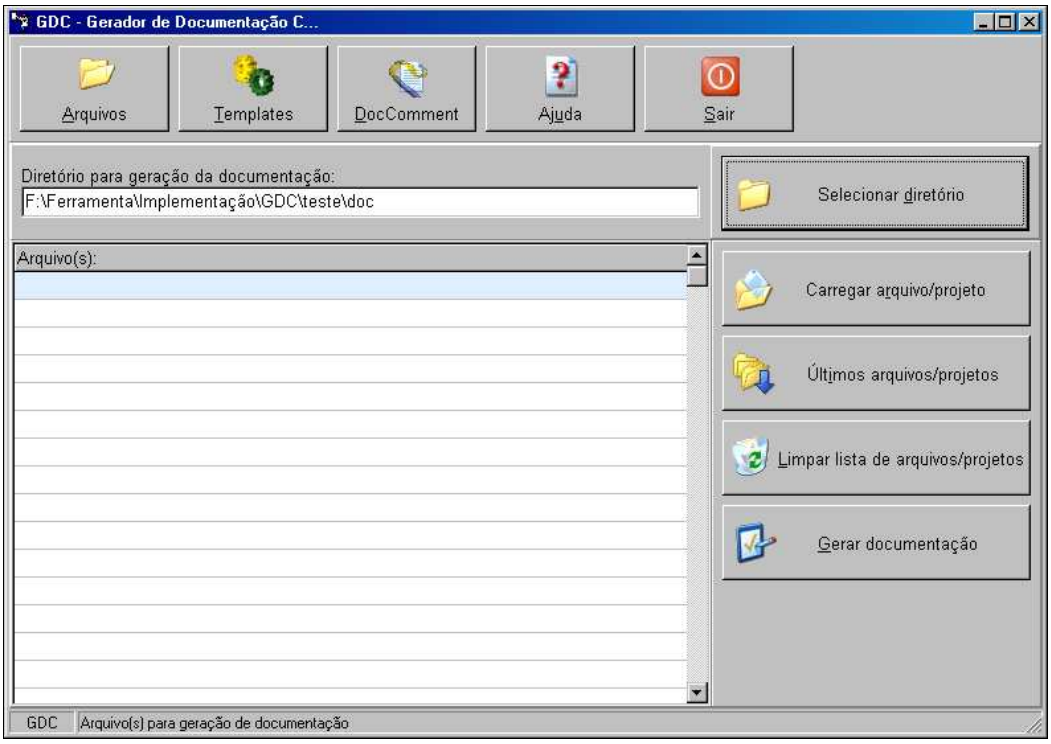


Figura 24 – Interface de trabalho com arquivos, com diretório setado

Além da opção para setar diretório, é possível `Carregar arquivo/projeto`, carregar `Últimos arquivos/projetos` e `Limpar lista de arquivos/projetos`. Caso o usuário pressione o botão `Últimos arquivos/projetos`, os dez últimos arquivos ou projetos analisados anteriormente são carregados pelo GDC. Como o próprio nome sugere, a opção

`Limpar lista de arquivos/projetos`, limpa a lista de arquivos carregados. A opção `Carregar arquivo/projeto` abre a janela para a seleção de arquivo (figura 25).

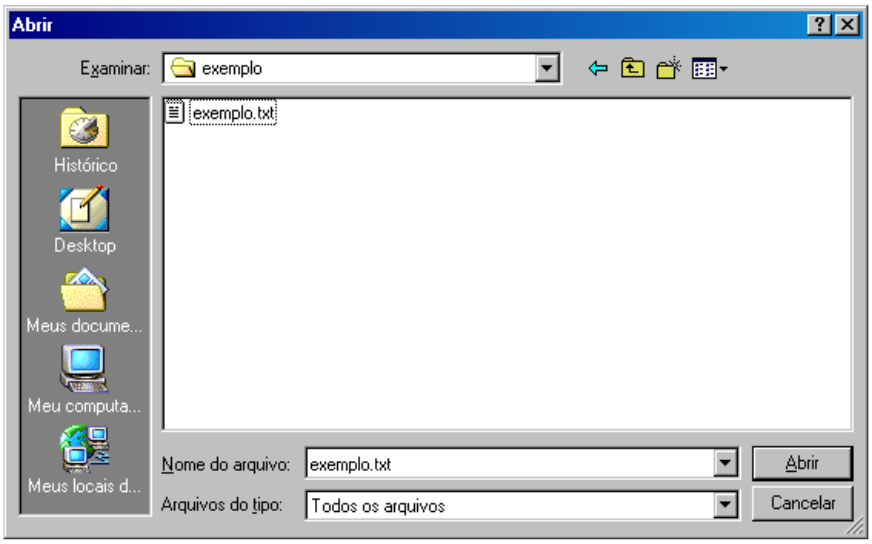


Figura 25 – Janela para seleção de arquivo

O usuário deve selecionar o arquivo que deseja carregar, sendo que ao pressionar o botão `Abrir`, o nome do arquivo é listado na interface para trabalho com arquivos (figura 26).

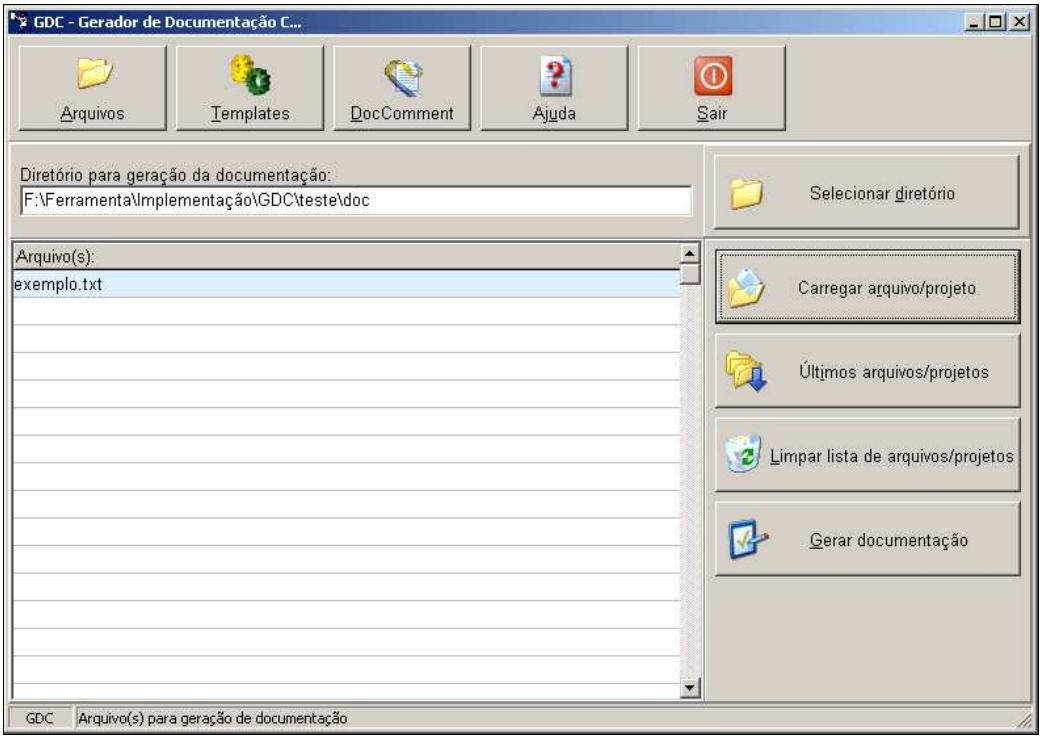


Figura 26 – Interface de trabalho com arquivos, com arquivo carregado

Para gerar documentação para um arquivo/projeto, deve-se selecionar o arquivo/projeto desejado e pressionar no botão `Gerar documentação`. Observa-se que o primeiro arquivo/projeto da relação de arquivos abertos é selecionado automaticamente. Além disso, é possível escolher o modelo de *templates* que será usado para gerar a documentação. Através do botão `Templates`, é aberta a interface de trabalho com modelos de *templates*

(figura 27). Acompanham o GDC três modelos de *templates*, sendo que novos modelos podem ser criados. Sempre que a interface de trabalho com modelos de *templates* for aberta pela primeira vez, o primeiro modelo é setado automaticamente.

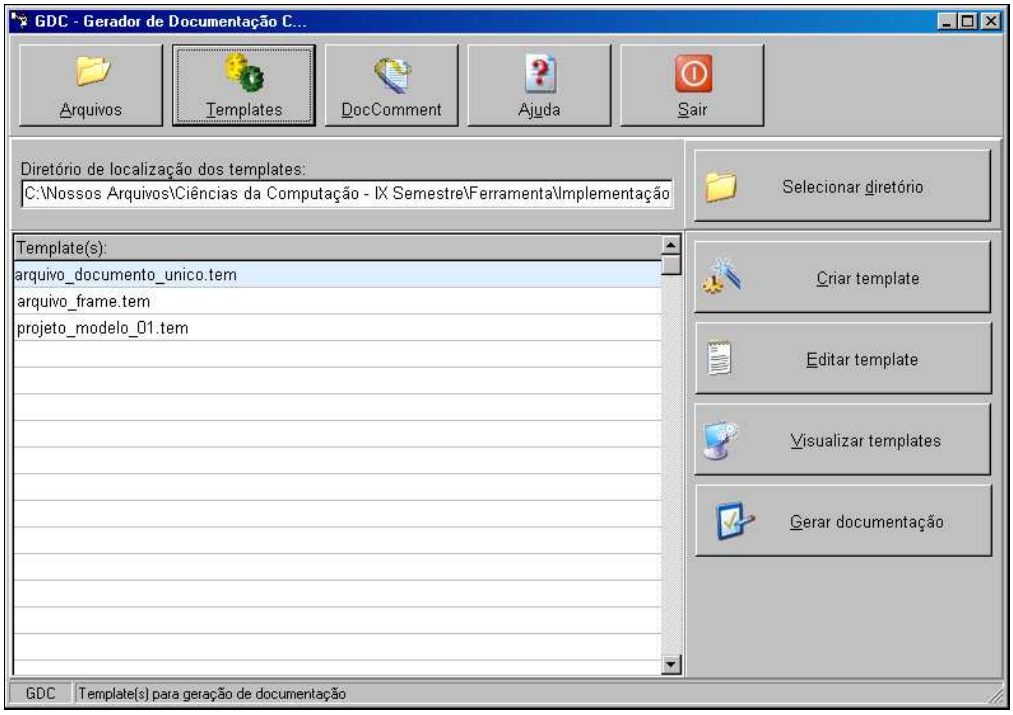


Figura 27 – Interface de trabalho com modelos de *templates*

Com o botão `Selecionar diretório` o usuário pode indicar o diretório onde modelos de *templates* estão localizados. Para selecionar um modelo, basta clicar sobre o mesmo (figura 28).

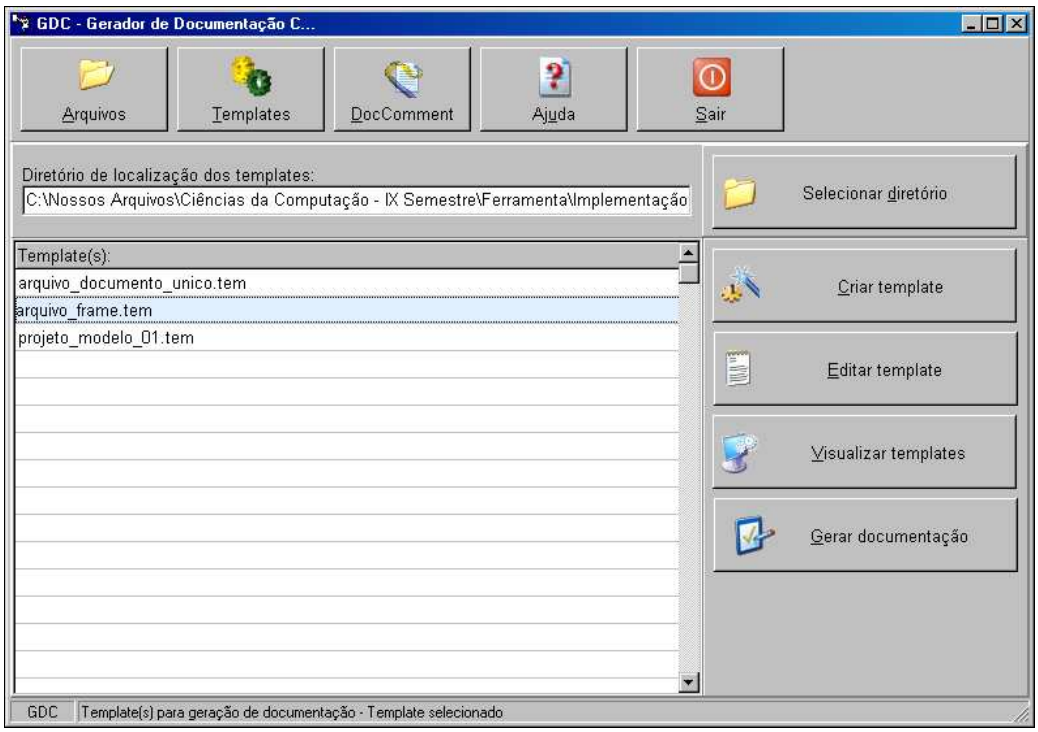


Figura 28 – Interface de trabalho com modelos de *templates*, com modelo selecionado

Através do botão `Gerar documentação`, é apresentada a interface de documentação (figura 29). Com a documentação gerada para o arquivo ou projeto selecionado usando o modelo de *templates* também selecionado. Salienta-se que o referido botão também está presente na interface de trabalho com arquivos, possuindo igual função, tendo sido duplicado apenas para facilitar o uso do GDC. A interface de documentação possui apenas o botão `Voltar`, possibilitando a navegação na documentação gerada.

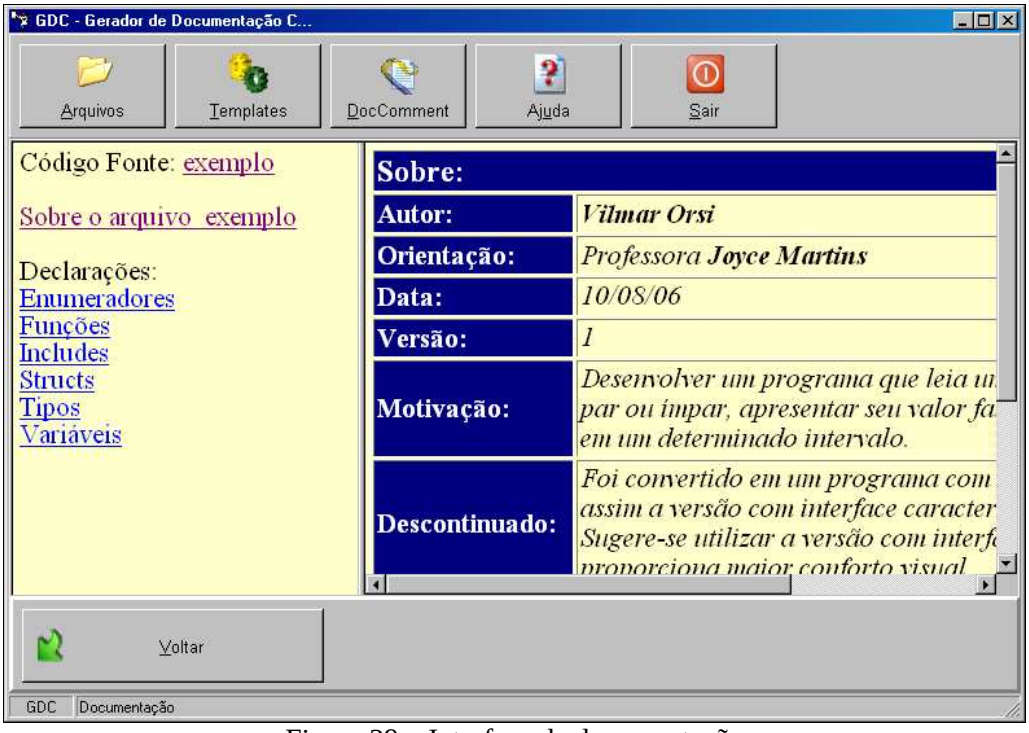


Figura 29 – Interface de documentação

Conforme apresentado na figura 28, a interface de trabalho com modelos de *templates* possui ainda os botões `Criar template`, utilizado para a criação de novos modelos de *templates*; `Visualizar templates`, usado para visualizar a formatação aplicada aos modelos de *templates*; e `Editar template`, para acessar a interface de edição de *templates*. No momento em que o referido botão é pressionado, é aberta a janela para seleção do arquivo, associado ao modelo de *templates*, que se deseja editar (figura 30).

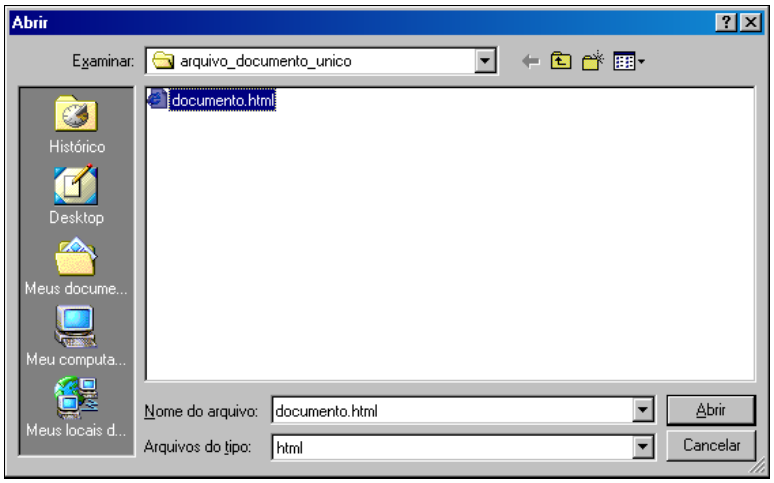


Figura 30 – Janela para seleção do arquivo associado ao modelo de *templates*

Ao selecionar o arquivo, e na seqüência o botão `Abrir`, o sistema apresenta a interface para edição dos arquivos pertencentes aos modelos de *templates* (figura 31).

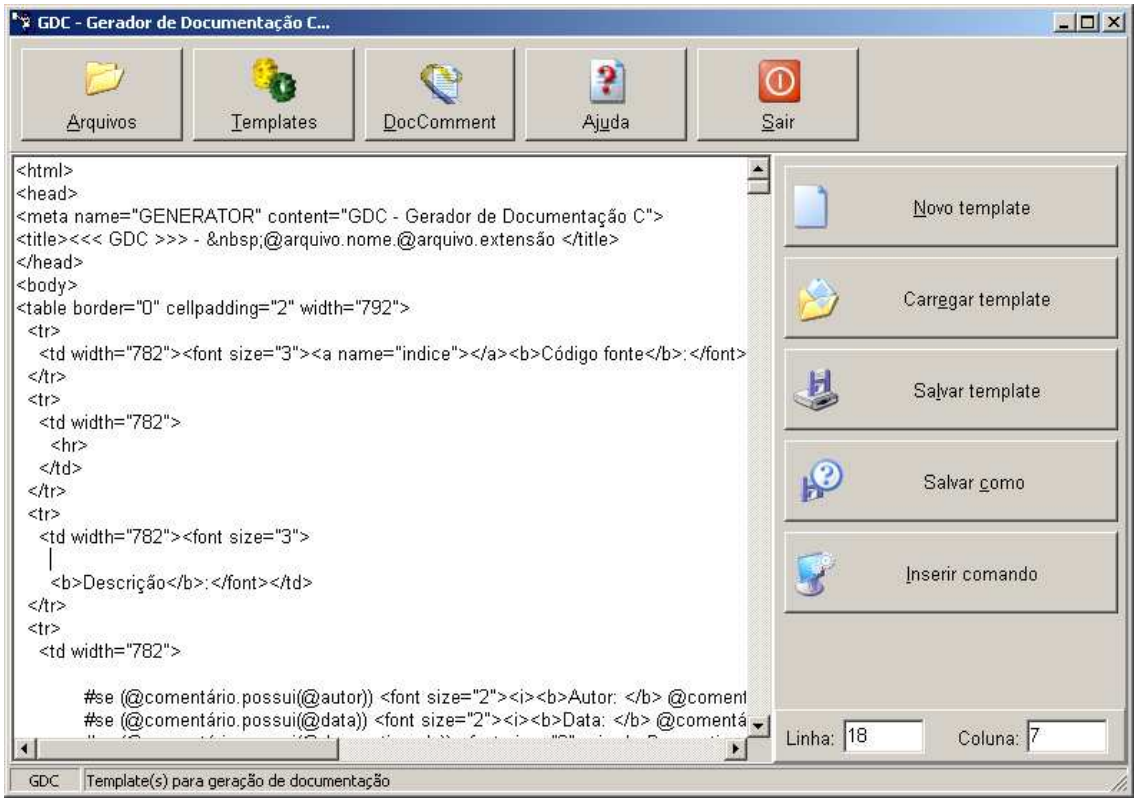


Figura 31 – Interface para edição de arquivos associados aos modelos de *templates*

Na referida interface, pode-se acionar os botões: `Novo template`, para criação de um novo arquivo que deve ser associado a um modelo de *templates*; `Carregar template`, para carregar um arquivo pertencente a um modelo de *templates*; `Salvar template`, para salvar o arquivo que está sendo editado; `Salvar como`, para salvar o arquivo com um nome diferente, ou em local diferente do original; e `Inserir comando`, através do qual o usuário pode inserir um comando da linguagem de *templates* no arquivo HTML que está sendo editado. Ao pressionar tal botão, abre-se a janela para seleção do comando desejado (figura 32).

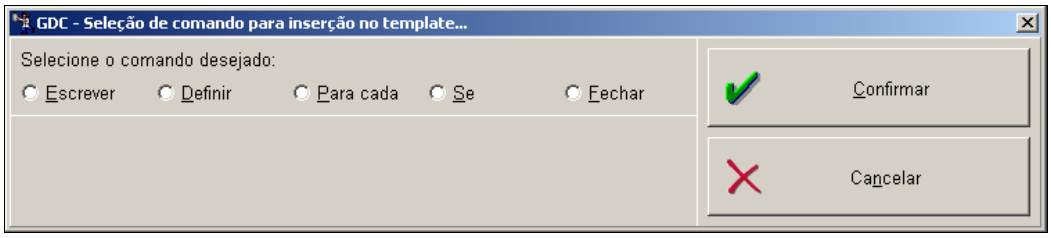


Figura 32 – Janela para seleção de comando da linguagem de *templates*

Basta então que sejam setadas as opções do comando desejado (figura 33).

Figura 33 – Janela com o comando da linguagem de *templates* selecionado

Ao selecionar o botão `Confirmar`, o comando é incluído no arquivo HTML que está sendo editado/criado, na posição do cursor (figura 34).

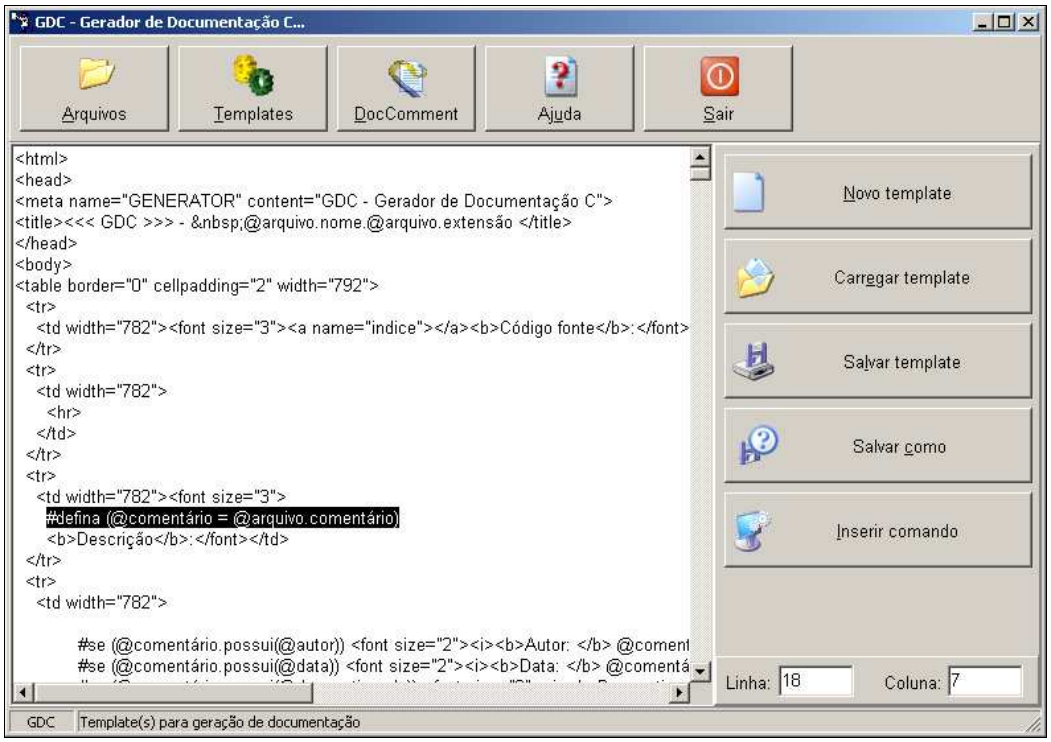


Figura 34 – Interface para edição de arquivos associados aos modelos de *templates*, com comando da linguagem de *templates* inserido

Ao pressionar o botão `DocComment`, na barra de ferramentas superior da interface do GDC, o usuário tem acesso à interface para inserção de *DocComments* (figura 35).

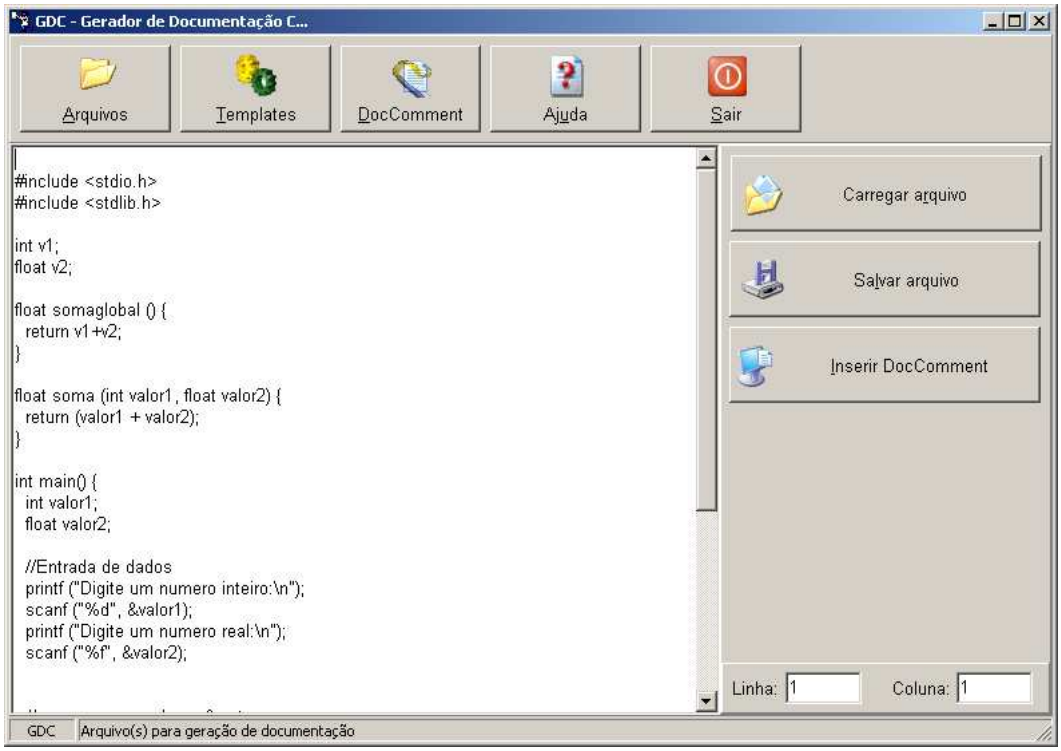


Figura 35 – Interface para inserção de *DocComments*

O usuário tem as opções de: `Carregar arquivo`, devendo ser este um arquivo de código fonte; `Salvar arquivo`, guardando as alterações realizadas; e `Inserir DocComment`, através da qual abre-se a janela para edição de *DocComments*, na qual o usuário deve preencher os campos desejado (figura 36).

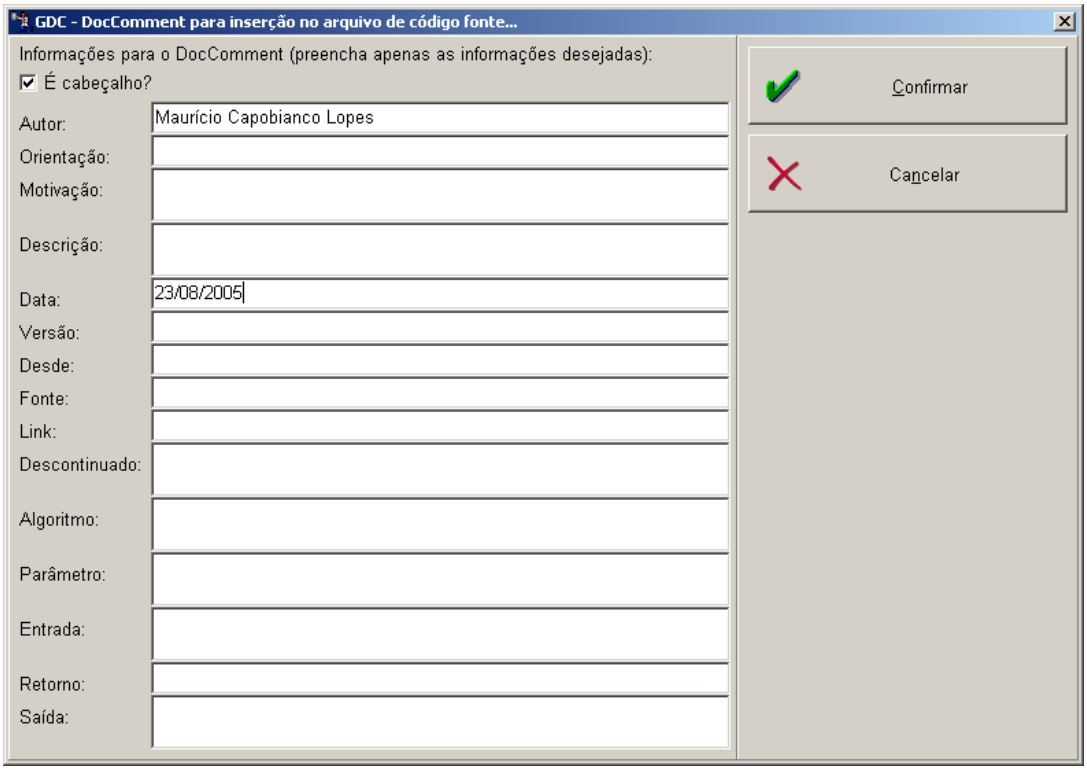


Figura 36 – Janela para edição de *DocComments*

Uma vez preenchidos os campos desejados e selecionada a opção `Confirmar`, o *DocComment* é incluído no código fonte, na posição do cursor (figura 37).

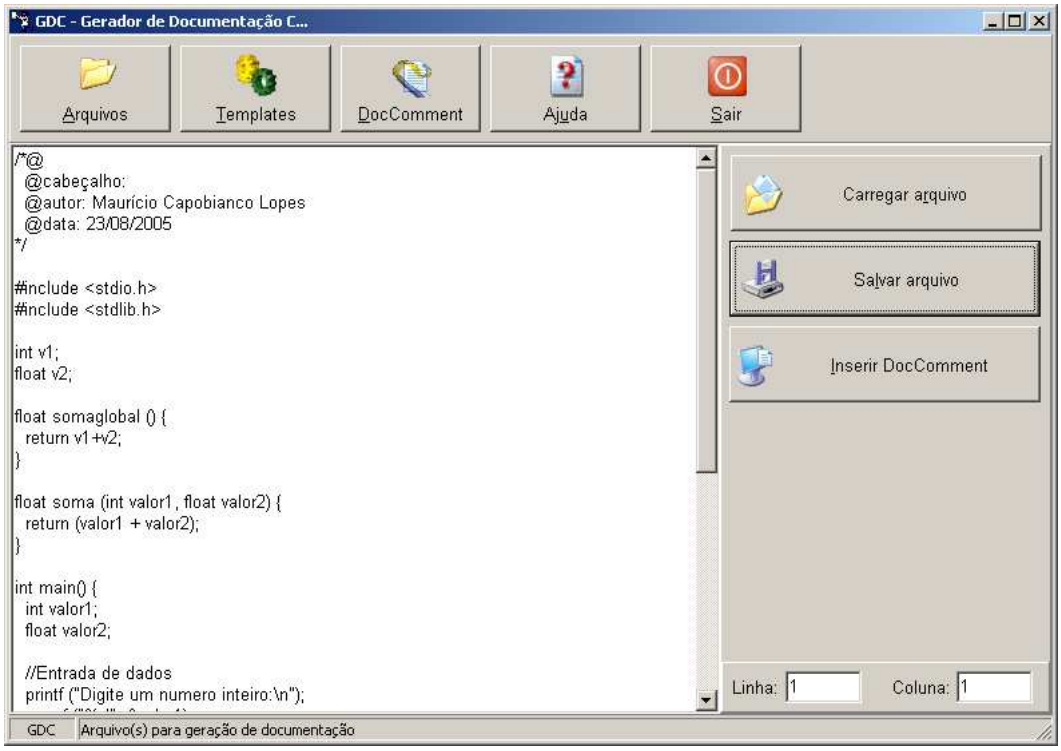


Figura 37 – Interface para inserção de *DocComments*, com *DocComment* inserido

O GDC dispõe também de um menu de ajuda (figura 38).

Figura 38 – Interface de ajuda

Através de cada um dos botões disponíveis na interface, o usuário tem acesso à documentação sobre os assuntos correspondentes.

3.8 RESULTADOS E DISCUSSÃO

Além dos testes realizados durante sua implementação, o GDC foi testado pelos acadêmicos da disciplina de Programação I, com aulas ministradas pelo professor Maurício Capobianco Lopes. Foram disponibilizadas três versões da ferramenta. A primeira versão (`Beta 1.0`) possibilitava a geração de documentos para programas que utilizam apenas as construções sintáticas previstas pela gramática original do C ANSI. A segunda versão (`Beta 1.1`) teve a gramática parcialmente alterada, possibilitando também a documentação de programas contendo `union` e `struct`. A terceira e última versão (`Beta 2.0`) permite a documentação de programas contendo todas as construções previstas pelo C ANSI, além de algumas construções do C++, utilizadas na disciplina de Programação I.

Diferente do previsto, a ferramenta foi utilizada também pelos acadêmicos do curso de Sistemas de Informação da FURB. Durante os testes, por parte dos acadêmicos, alguns erros foram encontrados, mas depois de analisados, verificou-se que se tratavam de erros nos códigos fonte que estavam sendo documentados, ou nos *DocComments* existentes nos mesmos. Sendo assim, para o conjunto de testes realizados com o GDC, o mesmo não apresentou erros.

Visando uma melhor avaliação do GDC, foi aplicado um questionário contendo 16 perguntas, a vinte e cinco acadêmicos que utilizaram a ferramenta durante o semestre 2006-II, sendo treze do curso de Ciências da Computação e doze do curso de Sistemas de Informação.

A primeira questão objetivou avaliar o grau de dificuldade do uso do GDC. Tendo sido implementado para aplicação acadêmica, a facilidade de uso foi preocupação constante no desenvolvimento. A maioria dos acadêmicos achou muito fácil ou fácil utilizar o GDC, sendo que, neste quesito, a ferramenta foi melhor avaliada pelos alunos de BCC (tabela 01).

Tabela 01 – Sobre o uso do GDC

Respostas	Curso				Todos	
	BCC		SIS			
	Qtd.	Perc. (%)	Qtd.	Perc. (%)	Qtd.	Perc. (%)
Muito fácil	1	7,69	2	16,67	3	12,00
Fácil	9	69,23	2	16,67	11	44,00
Razoável	3	23,08	7	58,33	10	40,00
Difícil	0	0,00	1	8,33	1	4,00
Muito difícil	0	0,00	0	0,00	0	0,00
Não respondeu	0	0,00	0	0,00	0	0,00
Total	13	100	12	100	25	100

A segunda questão visou avaliar a qualidade da interface do GDC. Também neste quesito a ferramenta foi melhor avaliada pelos acadêmicos do BCC, onde mais de 76% consideraram a interface como sendo boa (tabela 02).

Tabela 02 – Sobre a interface do GDC

Respostas	Curso				Todos	
	BCC		SIS			
	Qtd.	Perc. (%)	Qtd.	Perc. (%)	Qtd.	Perc. (%)
Muito boa	0	0,00	2	16,67	2	8,00
Boa	10	76,92	7	58,33	17	68,00
Razoável	2	15,38	3	25,00	5	20,00
Ruim	1	7,69	0	0,00	1	4,00
Muito ruim	0	0,00	0	0,00	0	0,00
Não respondeu	0	0,00	0	0,00	0	0,00
Total	13	100	12	100	25	100

Os atalhos e ícones utilizados no GDC foram avaliados através da terceira questão, onde os acadêmicos tiveram que avaliar se os mesmos eram condizentes com as funções associadas. Novamente a ferramenta foi melhor avaliada pelos acadêmicos do BCC, onde 100% afirmou que sempre ou quase sempre os atalhos e ícones são condizentes, contra 75% dos acadêmicos de SIS (tabela 03).

Tabela 03 – Sobre os ícones e os atalhos do GDC

Respostas	Curso				Todos	
	BCC		SIS			
	Qtd.	Perc. (%)	Qtd.	Perc. (%)	Qtd.	Perc. (%)
Sempre	4	30,77	6	50,00	10	40,00
Quase sempre	9	69,23	3	25,00	12	48,00
Algumas Vezes	0	0,00	2	16,67	2	8,00
Quase nunca	0	0,00	0	0,00	0	0,00
Nunca	0	0,00	1	8,33	1	4,00
Não respondeu	0	0,00	0	0,00	0	0,00
Total	13	100	12	100	25	100

Na quarta questão, o objetivo foi saber quantos alunos conseguiram gerar documentação utilizando o GDC. Dezenove dos alunos envolvidos nos testes da ferramenta conseguiram gerar a documentação (76%), sendo que dentre os alunos de BCC, o índice foi de 100% (tabela 04).

Tabela 04 – Sobre gerar a documentação

Respostas	Curso				Todos	
	BCC		SIS			
	Qtd.	Perc. (%)	Qtd.	Perc. (%)	Qtd.	Perc. (%)
Sim	13	100,00	6	50,00	19	76,00
Não	0	0,00	6	50,00	6	24,00
Não respondeu	0	0,00	0	0,00	0	0,00
Total	13	100	12	100	25	100

A quinta questão foi respondida apenas pelos dezenove alunos que conseguiram gerar a documentação, e teve como objetivo avaliar a qualidade da documentação gerada. A resposta de que a qualidade da documentação é muito boa ou boa, foi assinalada por mais de 78% dos alunos (tabela 05). Os demais acadêmicos responderam que a qualidade é razoável. Deve-se salientar que os acadêmicos utilizaram apenas os *templates* oferecidos pela ferramenta, não criando seus próprios *templates*, o que possivelmente melhoraria tal avaliação.

Tabela 05 – Sobre a qualidade da documentação gerada

Respostas	Curso				Todos	
	BCC		SIS			
	Qtd.	Perc. (%)	Qtd.	Perc. (%)	Qtd.	Perc. (%)
Muito boa	2	15,38	3	50,00	5	26,32
Boa	8	61,54	2	33,33	10	52,63
Razoável	3	23,08	1	16,67	4	21,05
Ruim	0	0,00	0	0,00	0	0,00
Muito ruim	0	0,00	0	0,00	0	0,00
Não respondeu	0	0,00	0	0,00	0	0,00
Total	13	100	6	100	19	100

Na sexta questão, os acadêmicos foram questionados sobre a ocorrência de erros durante o uso do GDC. Para quinze, dos vinte e cinco alunos, ocorreram erros, totalizando 60% (tabela 06). No entanto, como pode-se observar na tabela 07, contendo as respostas da sétima questão sobre os tipos de erros ocorridos, em nenhum dos casos foi indicado erro do GDC. Na maior parte dos casos, os erros estavam no código fonte dos programas em C (53,33%), seguido por erros nos *DocComments* (26,67%).

Tabela 06 – Sobre as ocorrências de erros durante o uso do GDC

Respostas	Curso				Todos	
	BCC		SIS			
	Qtd.	Perc. (%)	Qtd.	Perc. (%)	Qtd.	Perc. (%)
Sim	9	69,23	6	50,00	15	60,00
Não	4	30,77	6	50,00	10	40,00
Não respondeu	0	0,00	0	0,00	0	0,00
Total	13	100	12	100	25	100

Tabela 07 – Sobre os tipos de erros apresentados

Respostas	Curso				Todos	
	BCC		SIS			
	Qtd.	Perc. (%)	Qtd.	Perc. (%)	Qtd.	Perc. (%)
Falhas do GDC	0	0,00	0	0,00	0	0,00
Falhas nos *DocComments*	3	33,33	1	16,67	4	26,67
Erros no código fonte	5	55,56	3	50,00	8	53,33
Não descobriu o erro	1	11,11	2	33,33	3	20,00
Não respondeu	0	0,00	0	0,00	0	0,00
Total	9	100	6	100	15	100

A oitava questão teve por objetivo avaliar o grau de dificuldade encontrado pelos acadêmicos para escrever os *DocComments*. É importante salientar que nas primeiras versões da ferramenta, os comentários tinham que ser escritos manualmente. Na versão `Beta 2.0` foi incorporada a função que auxilia a inserção dos *DocComments* no código fonte, bastando que o acadêmico preencha os campos referentes aos *tags* desejados. De acordo com as respostas dos alunos (tabela 08), a maior parte considerou o grau de dificuldade razoável.

Tabela 08 – Sobre o grau de dificuldade para escrever os *DocComments*

Respostas	Curso				Todos	
	BCC		SIS			
	Qtd.	Perc. (%)	Qtd.	Perc. (%)	Qtd.	Perc. (%)
Muito fácil	0	0,00	1	8,33	1	4,00
Fácil	6	46,15	2	16,67	8	32,00
Razoável	7	53,85	7	58,33	14	56,00
Difícil	0	0,00	2	16,67	2	8,00
Muito difícil	0	0,00	0	0,00	0	0,00
Não respondeu	0	0,00	0	0,00	0	0,00
Total	13	100	12	100	25	100

Quando questionados sobre os *tags* GDC serem suficientes (nona questão), todos os acadêmicos responderam que sim (tabela 09). Sendo assim, a décima questão, com sugestões de novos *tags* não foi respondida por nenhum aluno.

Tabela 09 – Sobre os *tags* GDC serem suficientes

Respostas	Curso				Todos	
	BCC		SIS			
	Qtd.	Perc. (%)	Qtd.	Perc. (%)	Qtd.	Perc. (%)
Suficientes	13	100,00	12	100,00	25	100,00
Insuficientes	0	0,00	0	0,00	0	0,00
Não respondeu	0	0,00	0	0,00	0	0,00
Total	13	100	12	100	25	100

Sobre a seção de ajuda do GDC, primeiramente os acadêmicos foram questionados sobre terem utilizado ou não a referida seção (questão 11), ao que 56% responderam que

utilizaram, sendo oito acadêmicos do curso de SIS e seis de BCC (tabela 10). Aos alunos que responderam afirmativamente sobre o uso da ajuda, foi questionada a qualidade da referida seção. O GDC foi novamente melhor avaliado pelos acadêmicos do BCC, onde 66,67% consideraram a seção como sendo muito boa ou boa, contra 50% dos acadêmicos do SIS (tabela 11).

Tabela 10 – Sobre o uso da seção de ajuda

Respostas	Curso				Todos	
	BCC		SIS			
	Qtd.	Perc. (%)	Qtd.	Perc. (%)	Qtd.	Perc. (%)
Sim	6	46,15	8	66,67	14	56,00
Não	7	53,85	3	25,00	10	40,00
Não respondeu	0	0,00	1	8,33	1	4,00
Total	13	100	12	100	25	100

Tabela 11 – Sobre a qualidade da seção de ajuda

Respostas	Curso				Todos	
	BCC		SIS			
	Qtd.	Perc. (%)	Qtd.	Perc. (%)	Qtd.	Perc. (%)
Muito boa	1	16,67	1	12,50	2	14,29
Boa	3	50,00	3	37,50	6	42,86
Razoável	2	33,33	4	50,00	6	42,86
Ruim	0	0,00	0	0,00	0	0,00
Muito ruim	0	0,00	0	0,00	0	0,00
Não respondeu	0	0,00	0	0,00	0	0,00
Total	6	100	8	100	14	100

Finalizada a avaliação dos quesitos individuais do GDC, a questão treze questionou os acadêmicos sobre considerarem necessário o uso de geradores de documentação. A quase totalidade dos acadêmicos (92%) considerou o uso de geradores de documentação como sendo necessário ou útil (tabela 12).

Tabela 12 – Sobre o uso de geradores de documentação

Respostas	Curso				Todos	
	BCC		SIS			
	Qtd.	Perc. (%)	Qtd.	Perc. (%)	Qtd.	Perc. (%)
Necessária	1	7,69	1	8,33	2	8,00
Útil	12	92,31	9	75,00	21	84,00
Desnecessária	0	0,00	1	8,33	1	4,00
Não respondeu	0	0,00	1	8,33	1	4,00
Total	13	100	12	100	25	100

Quando questionados sobre continuar utilizando o GDC, 60% afirmaram que desejam continuar usando a ferramenta, contra 28% que responderam não querer mais utilizá-la (tabela 13). Três alunos do curso de SIS não responderam à questão.

Tabela 13 – Sobre continuar utilizando o GDC

Respostas	Curso				Todos	
	BCC		SIS			
	Qtd.	Perc. (%)	Qtd.	Perc. (%)	Qtd.	Perc. (%)
Sim	8	61,54	7	58,33	15	60,00
Não	5	38,46	2	16,67	7	28,00
Não respondeu	0	0,00	3	25,00	3	12,00
Total	13	100	12	100	25	100

Através da décima quinta questão, os acadêmicos tiveram a oportunidade de atribuir uma nota, de 0 a 5, ao GDC. A nota máxima foi aplicada por apenas um acadêmico, sendo que a maior parte dos alunos atribuiu a nota 4 (56%), totalizando 60% das avaliações para as notas 4 e 5 (tabela 14).

Tabela 14 – Nota do GDC

Respostas	Curso				Todos	
	BCC		SIS			
	Qtd.	Perc. (%)	Qtd.	Perc. (%)	Qtd.	Perc. (%)
Nota 5	0	0,00	1	8,33	1	4,00
Nota 4	7	53,85	7	58,33	14	56,00
Nota 3	6	46,15	2	16,67	8	32,00
Nota 2	0	0,00	1	8,33	1	4,00
Nota 1	0	0,00	0	0,00	0	0,00
Nota 0	0	0,00	0	0,00	0	0,00
Não respondeu	0	0,00	1	8,33	1	4,00
Total	13	100	12	100	25	100

A última questão deu a oportunidade aos acadêmicos para que apresentassem algum comentário sobre o GDC. Alguns alunos comentaram que os erros no código fonte, nos *DocComments* e nos *templates* não eram diferenciados. Esse problema foi resolvido na versão `Beta 2.0` da ferramenta, onde erros nos *DocComments* e nos *templates* passaram a ser apresentados detalhadamente, indicando-se, além do erro, a linha onde o mesmo se encontra. No caso do código fonte, continua sendo apenas informado que o programa apresenta erros, visto que, conforme já mencionado, considera-se que todo e qualquer programa que se queira documentar esteja correto.

Em relação aos trabalhos correlatos estudados, pode-se afirmar que:

a) o GDC utiliza um conjunto de *tags* semelhantes ao JavaDoc, acrescido dos *tags* `@algoritmo:`, `@entrada:` e `@saída:`, solicitados pelo professor Maurício Capobianco Lopes. Além dos *tags* próprios da linguagem, assim como o JavaDoc, os *DocComments* do GDC também podem conter *tags* HTML;
b) o GDC permite a documentação de pacotes, assim como o JavaDoc;

c) o GDC, da mesma maneira que o JavaDoc, o gerador de documentação do C#, o CDoc e o Doxygen, faz uso de comentários especiais para a geração da documentação, salientando que, dentre todas as ferramentas, o CDoc não utiliza *tags* para a classificação dos comentários;

d) o GDC diferencia-se de todos os trabalhos correlatos por permitir a personalização da documentação gerada através do uso de *templates*.

No quadro 38 pode ser observado um comparativo entre o GDC e os trabalhos correlatos estudados.

FERRAMENTA	**LINGUAGEM**	**USO DE *TEMPLATES***	**FORMATO DE SAÍDA**	***TAGS* DA FERRAMENTA**	***TAGS* HTML**
JavaDoc	Java	Não	HTML	Sim	Sim
Gerador de documentação do C#	C#	Não	XML	Sim	Não
CDoc	C C++ Java	Não	HTML	Não	Não
Doxygen	C C++ Java Objective C	Não	HTML RTF PDF XML Entre outros	Sim	Não
GDC	C ANSI	Sim	HTML	Sim	Sim

Quadro 38 – Comparativo entre o GDC e os trabalhos correlatos estudados

Além da geração da documentação em HTML, há a possibilidade de realizar a geração em outros formatos, necessitando apenas algumas alterações no motor de *templates*, de forma a gerar a saída com a formatação desejada (parágrafos, tabulações, entre outras), e na forma de escrever os *DocComments.* No quadro 39 é apresentado um exemplo de *template*, apontando a possibilidade de gerar um arquivo texto (extensão `.TXT`), que pode ser visualizado na figura 39.

```
Este é um exemplo de que é possível gerar arquivos txt
com a documentação de um programa em C.

Nome do arquivo:  @arquivo.nome.@arquivo.extensão
```

Quadro 39 – Exemplo de *template* para documentação no formato texto

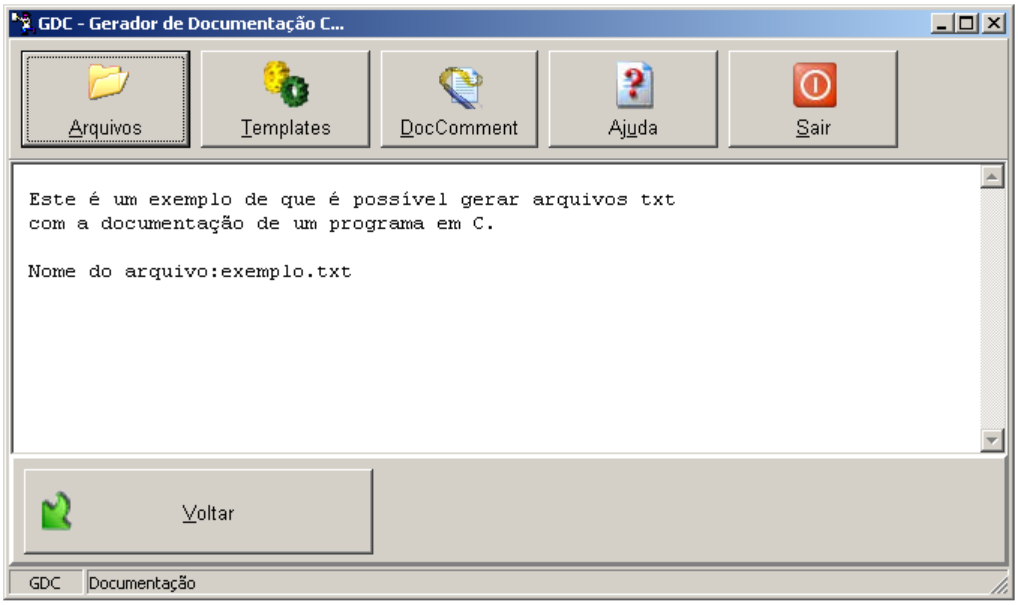


Figura 39 – Documentação gerada no formato texto

4 CONCLUSÕES

Da análise dos problemas envolvendo a questão da documentação de software, surgiu a idéia do desenvolvimento do GDC, uma ferramenta para a geração automática de documentação a partir do código fonte de programas escritos em C. O desenvolvimento da referida ferramenta voltou-se à geração de documentos de qualidade, que possam auxiliar no processo de implementação, manutenção e adaptação de software.

O GDC faz uso de marcações especiais, dentro dos comentários de blocos, e gera a documentação no formato HTML, com as informações dispostas conforme estabelecido pelo usuário quando da elaboração do modelo de *templates* para os referidos documentos. Para códigos fonte não comentados, é possível gerar documentação com características básicas dos mesmos, tais como declaração de variáveis, cabeçalho de procedimentos e de funções. Quando da utilização de comentários, são gerados documentos mais complexos a partir dos referidos marcadores especiais, que são usados para delimitar informações como autor, data ou versão do código fonte.

As informações necessárias para gerar documentação são extraídas dos códigos fonte escritos em C através de três conjuntos de analisadores léxico, sintático e semântico, sendo um dos conjuntos para a análise e coleta de informações no código fonte dos programas em C, outro para os *DocComments* e o terceiro para os *templates*. Um quarto analisador léxico armazena em uma tabela de símbolos todos os *tokens* reconhecidos no código fonte analisado, incluindo os comentários. É importante frisar que os analisadores léxicos e sintáticos foram gerados pelo GALS, o que veio a reduzir significativamente o trabalho de implementação de tais módulos.

A documentação gerada pode ser padronizada pelos usuários, através dos modelos de *templates*, que são utilizados para formatar as informações agrupadas durante as análises léxica, sintática e semântica. A utilização dos modelos de *templates* está vinculada à liberdade concedida ao usuário de selecionar quais informações e em que ordem devem ser apresentadas na documentação. Deve-se destacar que para alcançar o objetivo de utilizar *templates* para a geração da documentação, foi necessário especificar e implementar um motor de *templates*, visto que não foi encontrado nenhum motor que satisfizesse as necessidades do gerador. Heesch (2006) quando menciona o uso de *templates*, atribui a esta atividade o grau máximo de dificuldade dentre as funcionalidades que propõe que sejam implementadas para o Doxygen.

Observa-se que, em adição aos objetivos propostos foram implementadas funcionalidades para inserção dos comandos da linguagem de *templates* nos modelos de *templates* e de *DocComments* nos códigos fonte a serem documentados, o que reduz a chance de erros na documentação de programas.

Outro fator importante foi a avaliação realizada pelos acadêmicos da disciplina de Programação I, dos cursos de Ciências da Computação e Sistemas de Informação, ministradas pelo professor Maurício Capobianco Lopes, que possibilitou a verificação de possíveis erros e aumentou a confiabilidade no GDC.

4.1 EXTENSÕES

Como extensões para o GDC propõe-se:

a) melhorar a inserção de *DocComments*, de maneira que, quando a inserção de um comentário for solicitada e o cursor se encontrar no interior de um *DocComment* já existente, as informações do mesmo sejam carregadas na interface de edição de comentários, e que quando a confirmação da inserção do mesmo acontecer, o referido *DocComment* seja sobrescrito;
b) melhorar as mensagens de erros, de forma a identificar também os erros no código fonte que se está documentando;
c) implementar um editor gráfico para os *templates*;
d) aprimorar a geração de documentação, possibilitando informar quais funções de um *include* estão sendo utilizadas;
e) implementar um controle de versões para a documentação gerada;
f) adaptar a ferramenta permitindo a geração do algoritmo do programa através de comentários de linha, aplicados no código fonte;
g) incluir na ferramenta a opção de seleção de idioma, permitindo ao usuário escrever seus *DocComments* em inglês;
h) especificar e implementar um motor de *templates* genérico para Delphi;
i) adaptar a ferramenta, tranformando-a em um *plugin* para algum ambiente de programação;
j) adaptar a ferramenta, possibilitando analisar a qualidade dos códigos fontes desenvolvidos.

REFERÊNCIAS BIBLIOGRÁFICAS

ÁECE, I. **Criação de comentários e documentação em VB.NET**. [S.l.], 2003. Disponível em: <http://www.csharpbr.com.br/mostra_artigo.asp?id=0025>. Acesso em: 26 fev. 2006.

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

BRECK, L**. eNITL**: the Network Improv Template Language. [S.l.], 1999. Disponível em: <http://www.codeguru.com/cpp/cpp/cpp_mfc/xlinks/article.php/c773/>. Acesso em: 25 out. 2006.

CÂMARA, F. **O processo de compilação no C#**. [São Paulo], 2003a. Disponível em: <http://www.linhadecodigo.com.br/artigos.asp?id_ac=135&pag=4>. Acesso em: 26 fev. 2006.

_____. **Comentários XML no Visual Studio.NET**. [São Paulo], 2003b. Disponível em: <http://www.csharpbr.com.br/mostra_artigo.asp?id=0008>. Acesso em: 16 out. 2006.

CRUZ, S. A. B.; MOURA, M. F. **Formatação de dados usando a ferramenta Velocity.** Campinas, 2002. Disponível em: <http://www.cnptia.embrapa.br/modules/tinycontent3/content/2002/comuntec20.pdf>. Acesso em: 15 out. 2006.

FLANAGAN, D. **Java:** o guia essencial. 3. ed. Tradução Kátia Roque. Rio de Janeiro: Campus, 2000.

FRAMEWORK. In: WIKIPEDIA, a enciclopédia livre. [S.l.]: Wikimedia Foundation, 2006. Disponível em: <http://pt.wikipedia.org/wiki/Framework>. Acesso em: 16 out. 2006.

GALUPPO, F.; MATHEUS, V.; SANTOS, W. **Desenvolvendo com C#**. Porto Alegre: Bookman, 2004.

GOIS, H. M. C. C. **Programando com C#**. Santa Maria, 2005. Disponível em: <http://www.juliobattisti.com.br/tutoriais/herbertgois/programandocsharp004.asp>. Acesso em: 16 out. 2006.

GRAMMAR downloads: examples and full programming languages. [S.l.], [2006?]. Disponível em: <http://www.devincook.com/goldparser/grammars/#Languages>. Acesso em: 26 jun. 2006.

HADDAD, R. I. **C#:** aplicações e soluções. São Paulo: Érica, 2001.

HEESCH, D. **Doxygen**. [S.l.], 2006. Disponível em: <http://www.stack.nl-/~dimitri/doxygen/>. Acesso em: 16 out. 2006.

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

HILL, D. **FastTrac template engine for Delphi**. [S.l.], 2004. Disponível em: <http://sourceforge.net/projects/ftopf/>. Acesso em: 17 set. 2006.

JAVADOC. In: WIKIPEDIA, a enciclopédia livre. [S.l.]: Wikimedia Foundation, 2006. Disponível em: <http://pt.wikipedia.org/wiki/Javadoc>. Acesso em: 15 out. 2006.

KERNIGHAN, B. W. **C, a linguagem de programação:** padrão ANSI. Tradução Daniel Vieira. Rio de Janeiro: Campus, 1990.

LOUDEN, K. C. **Compiladores**: princípios e práticas. Tradução Flávio Soares Corrêa da Silva. São Paulo: Thomson Pioneira, 2004.

MARTIN, F. **Velocity**. Vitória, 2005. Disponível em: <http://www.imasters.com.br/artigo/3240>. Acesso em: 17 out. 2006.

MILVANG, O. **CDoc**. Oslo, 2004. Disponível em: <http://www.ifi.uio.no/forskning/grupper/dsb/Software/Xite/ReferenceManual/cdoc_1.html>. Acesso em: 13 mar. 2006.

MOREIRA, D.; MRACK, M. Sistemas dinâmicos baseados em metamodelos. In: WORKSHOP DE COMPUTAÇÃO E GESTÃO DA INFORMAÇÃO, 2., 2003, Lajeado. **Anais eletrônicos...** [Lajeado]: UNIVATES, 2003. Não paginado. Disponível em: <http://www.univates.br/sicompi/wcompi2003/09-moreira-mrack.pdf>. Acesso em: 15 out. 2006.

NUNES, V. B.; SOARES, A. O.; FALBO, R. A. Apoio à documentação em um ambiente de desenvolvimento de software. In: WORKSHOP IBEROAMERICANO DE INGENIERA DE REQUISITOS Y DESARROLLO DE AMBIENTES DE SOFTWARE, 7., 2004, Arequipa. **Anais...** Arequipa: IDEAS, 2004. Não paginado. Disponível em: <http://www.inf.ufes.br/~falbo/download/pub/2004-IDEAS-1.pdf>. Acesso em: 05 mar. 2006.

PAMPLONA, V. F. **Tutorial Java**: o que é Java? [Blumenau.], 2006. Disponível em: <http://www.javafree.org/content/view.jf?idContent=84>. Acesso em: 16 out. 2006.

PARR, T. **Enforcing strict model-view separation in template engines**. São Francisco, 2004. Disponível em: <http://www.cs.usfca.edu/~parrt/papers/mvc.templates.pdf>. Acesso em: 11 jun. 2006.

PAULO, J. L.; GRAÇA, J. **Uma introdução ao JavaDoc**. Lisboa, 2003. Disponível em: <http://po.tagus.ist.utl.pt/2005/apoio/Introjavadoc.html>. Acesso em: 12 mar. 2006.

PFLEEGER, S. L. **Engenharia de software**: teoria e prática. Tradução Dino Franklin. São Paulo: Prentice Hall, 2004.

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

RENALDI, F. **Genos**: protótipo de um montador de sistemas operacionais para sistemas embarcados. 2006. 69 f. Trabalho de Conclusão de Curso (Bacharelado em Ciências da Computação) - Centro de Ciências Exatas e Naturais, Universidade Regional de Blumenau, Blumenau.

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: 15 mar. 2006.

SANT´ANNA, M. **Documentação em programas C#**. São Paulo, 2001. Disponível em: <http://www.mas.com.br/Artigos/Doc_CSharp.htm>. Acesso em: 26 fev. 2006.

SEMÂNTICA. In: WIKIPEDIA, a enciclopédia livre. [S.l.]: Wikimedia Foundation, 2006. Disponível em: <http://pt.wikipedia.org/wiki/Sem%C3%A2ntica>. Acesso em: 25 out. 2006.

SILVEIRA, J. **Extensão da ferramenta Delphi2Java-II para suportar componentes de banco de dados**. 2006. 81 f. Trabalho de Conclusão de Curso (Bacharelado em Sistemas de Informação) - Centro de Ciências Exatas e Naturais, Universidade Regional de Blumenau, Blumenau.

SOFTWARE BLACKSMITHS. **Welcome to Software Blacksmiths**: CDoc. Mississauga, [2001?]. Disponível em: <http://www.swbs.com/>. Acesso em: 13 mar. 2006.

SOUZA, A. **Ferramenta para conversão de formulários Delphi em páginas HTML.** 2005. 69 f. Trabalho de Conclusão de Curso (Bacharelado em Sistemas de Informação) - Centro de Ciências Exatas e Naturais, Universidade Regional de Blumenau, Blumenau.

SOUZA, S. C. B.; ANQUETIL, N.; OLIVEIRA, K. M. Documentação essencial para manutenção de software. In: WORKSHOP DE MANUTENÇÃO DE SOFTWARE MODERNA, 1., 2004, Brasília. **Anais Eletrônicos...** Brasília: UCB, 2004. Não paginado. Disponível em: <www.ucb.br/ucbtic/wmswm-04/DocumentacaoEssencialManutencaoII.pdf>. Acesso em: 15 out. 2006.

SOWEK, C. A. **A importância da documentação**. Curitiba, 2003. Disponível em: <http://www.pr.gov.br/batebyte/edicoes/1999/bb88/importancia.htm>. Acesso em: 15 out. 2006.

TERRA, F. M. **Manual de documentação básica para o GSM no estilo Doxygen**. Belo Horizonte, [2006?]. Disponível em: <http://www.cpdee.ufmg.br/~gopac/gsm/manual.pdf>. Acesso em: 10 out. 2006.

VERBETE. In: MELHORAMENTOS: dicionário prático de informática. São Paulo: Melhoramentos, 2001. Formato eletrônico.

APÊNDICE A – Especificação do C ANSI adaptada para o GDC

No quadro 48 é apresenta a especificação (léxica e sintática) utilizada pelo GDC para análise do código fonte em C.

```
#RegularDefinitions
Digit      : [0-9]
OctDigit   : [01234567]
HexDigit   : ([0-9]|[abcdefABCDEF])
Letter     : [a-zA-Z]
Whitespace : [\n\t\s\r]
```

```
#Tokens
Id : ({Letter}|[_])({Letter}|{Digit}|[_])*
DecLiteral       : [123456789]{Digit}*
OctLiteral       : 0{OctDigit}*
HexLiteral       : 0x{HexDigit}*
FloatLiteral     : {Digit}*\.{Digit}+
CharLiteral      : ' [^'] '
StringLiteralAps : ' ((\') | [^'] )* '
StringLiteralAsp : \" ((\ \") | [^\"])* \"

// Comentários
DocComent  : (/\*) (@) (([^\*])|(\*[^/]))* ((\*)+/)
LineComent : (/)(/)+.*
TextComent : (/\*) (([^\*])|(\*[^/]))* ((\*)+/)

// Diretivas
DiretivaDefine : (#)(\ )*(define).*
DiretivaUndef  : (#)(\ )*(undef).*
DiretivaLine   :  (#)(\ )*(line).*
DiretivaError  : (#)(\ )*(error).*
DiretivaPragma : (#)(\ )*(pragma).*
DiretivaIfdef  : (#)(\ )*(ifdef).*
DiretivaIfndef : (#)(\ )*(ifndef).*
DiretivaIf     : (#)(\ )*(if).*
DiretivaElif   : (#)(\ )*(elif).*
DiretivaElse   : (#)(\ )*(else).*
DiretivaEndif  : (#)(\ )*(endif).*
DiretivaIncludeArq1 : (#)(\ )*(include)(\ )*(<)(\ )*(.+)(\ )*(>)(\ )*
DiretivaIncludeArq2 : (#)(\ )*(include)(\ )*(\")(\ )*(.+)(\ )*(\")(\ )*
DiretivaIncludeCod : (#)(\ )*(include).*
DiretivaNula   : (#).*

// Palavras reservadas
auto     = Id : "auto"        bool     = Id : "bool"      break    = Id : "break"
case     = Id : "case"        char     = Id : "char"      const    = Id : "const"
continue = Id : "continue"    default  = Id : "default"   do       = Id : "do"
double   = Id : "double"      else     = Id : "else"      enum     = Id : "enum"
extern   = Id : "extern"      float    = Id : "float"     for      = Id : "for"
goto     = Id : "goto"        if       = Id : "if"        int      = Id : "int"
long     = Id : "long"        register = Id : "register"  return   = Id : "return"
short    = Id : "short"       signed   = Id : "signed"    sizeof   = Id : "sizeof"
static   = Id : "static"      struct   = Id : "struct"    switch   = Id : "switch"
typedef  = Id : "typedef"     union    = Id : "union"
unsigned = Id : "unsigned"    void     = Id : "void"
volatile = Id : "volatile"    while    = Id : "while"

// Tags
inicio : <@doc>              fim : <@@doc>

// Símbolos especiais
;  (  )  {  }  [  ]  ,  :  =  +=  -=  *=  /=  ^=  &=  |=  >>=  <<=  ?  ||  &&  |
^     &  ==  !=  <  >  <=  >=  <<  >>  +  -  *  /  %  !  ~  &  ++  --  .  ->  '
```

```
#Grammar
<Doc>         ::= inicio <Decls> fim
<Decls>       ::= ε  |  #1 <Decl> #2 <Decls>
<Decl>        ::= <Sign> <Scalar> #13 <Pointers> <Decl_aux1>
                |   typedef #6 <Type> <TypedefDecl_>
                |   <Mod> #9 <VarDecl_>
                |   Id #13 <Decl_aux2>
                |   struct #8 <Decl_aux3>
                |   union #15 <Decl_aux4>
                |   enum #3 #13 <Decl_aux5>
<Decl_aux1>   ::= Id <Decl_aux1_>  | & Id <Array> <Var_> <VarList> ; #4
<Decl_aux1_>  ::= ( <Params> ) #11 <BlockFunc>
                |   <Array> <Var_> <VarList> ; #10 #14 #4
                |   { <EnumDef> }  ; #14
<Decl_aux2>   ::= <Pointers> <Decl_aux1>  | ( <Params> ) #11 <BlockFunc>
<Decl_aux3>   ::= Id #13 <Decl_aux3_>
                |   #13 { <StructDef> } #10 <Pointers> <Decl_aux1>
<Decl_aux3_>  ::= #13 { <StructDef> } <Decl_aux6> ; #10
                |   #13 <Pointers> <Decl_aux1>
<Decl_aux4>   ::= Id #13 <Decl_aux3_>
                |   #13 { <StructDef> } #10 <Pointers> <Decl_aux1>
<Decl_aux5>   ::= <Pointers> <Decl_aux1>  |  { <EnumDef> }  ; #14
<Decl_aux6>   ::= ε  |  Id
//============================================================================
// Function  Declaration
//============================================================================
<Params>      ::= ε  |  <ParamsList>
<ParamsList>  ::= <Type> <ParamsList_>  |  const <Type> <ParamsList_>
<ParamsList_>::= <ParamsList__>
                |   & Id <Array> <ParamsList__>
                |     Id <Array> <ParamsList__>
<ParamsList__>::= ε  |  , <ParamsList>
<BlockFunc>   ::= ;  |  <Block>  |  <StructDef> <Block>
//============================================================================
// Type Declaration
//============================================================================
<TypedefDecl_>::= <Decl_aux3_>  |  <Array> ;
<StructDef>   ::= #1 <VarDecl> #2 <StructDef_>
<StructDef_>  ::= ε  |  <StructDef>
//============================================================================
// Variable Declaration
//============================================================================
<VarDecl>     ::= <Mod> #9 <VarDecl_>  |  <Type> <Var> <VarList> ; #4
<VarDecl_>    ::= <Base> <Pointers> <Var> <VarList> ; #4
                |   Id <VarDecl__>
                |   & Id   <Array>  <Var_> <VarList> ; #4
<VarDecl__>   ::= <Pointers> <Var> <VarList> ; #4
                |   <Array> <Var_> <VarList> ; #4
<Var>         ::= & Id <Array> <Var_>  |  Id <Array> <Var_>
<Var_>        ::= ε  |  = <var__>
<var__>       ::= <OpIf>  |  { <Expr> }
<Array>       ::= ε  |  [ <Array_> ] <Array>
<Array_>      ::= ε  |  <Expr>
<VarList>     ::= ε  |  , <VarItem> <VarList>
<VarItem>     ::= <Pointers> <Var>
<Mod>         ::= extern | static | register | auto | volatile | const
//============================================================================
// Enumerations
//============================================================================
<EnumDef>     ::= <EnumVal> <EnumDef_>
<EnumDef_>    ::= ε  |  , <EnumDef>
<EnumVal>     ::= Id <EnumVal_>
<EnumVal_>    ::= ε  |  = <EnumVal__>
<EnumVal__>   ::= OctLiteral | HexLiteral | DecLiteral
```

```
//=============================================================================
// Types
//=============================================================================
<Type>       ::= <Base> <Pointers> | Id <Pointers>
<Base>       ::= <Sign> <Scalar> #13 | struct #8 <Base_> | union #15 <Base_>
             |   enum #3 Id
<Base_>      ::= Id #13  |  #13 { <StructDef> } #10
<Sign>       ::= ε  |  signed  |  unsigned
<Scalar>     ::= int | short <Scalar_> | long  <Scalar_> | float | double
             |   char | void | bool
<Scalar_>    ::= ε  |  int
<Pointers>   ::= ε  |  * <Pointers>
//=============================================================================
// Statements
//=============================================================================
<Stm>        ::= ;  |  <Block>
             |   if ( <Expr> ) <Stm> <Stm_>
             |   while ( <Expr> ) <Stm>
             |   for ( <Arg> ; <Arg> ; <Arg> ) <Stm>
             |   do <Stm> while ( <Expr> )
             |   switch ( <Expr> ) { <CaseStms> }
             |   <Expr> #16 ;  |  break ;  |  continue ;  | return <Expr> ;
<Stm_>       ::= ε  |  else <Stm>
<Arg>        ::= ε  |  <Expr>
<CaseStms>   ::= ε  |  case <Expr> : <StmList> <CaseStms>  |  default : <StmList>
<Block>      ::= { <StmList> }
<StmList>    ::=  ε  |  <Stm> <StmList>
//=============================================================================
// Here begins the C's 15 levels of operator precedence.
//=============================================================================
<Expr>       ::= <OpAssign> <Expr_>
<Expr_>      ::= ε  |  #17 , <OpAssign> <Expr_>
<OpAssign>   ::= <OpIf> <OpAssign_>
<OpAssign_>  ::= ε  |  <SignAssign> #7 <OpAssign>
<SignAssign> ::= = | += | -= | *= | /= | ^= | &= | |= | >>= |   <<= #7
<OpIf>       ::= <OpBooleanCompareMath> <OpIf_>
<OpIf_>      ::= ε  |  ? <OpIf> :  <OpIf>
<OpBooleanCompareMath>  ::= <OpUnary> <OpBooleanCompareMath_>
<OpBooleanCompareMath_> ::= ε
             |   <SignBooleanCompareMath> <OpUnary> <OpBooleanCompareMath_>
<SignBooleanCompareMath>::= "||" | && | "|" | ^ | & | == | != | < | > | <=
             |   >= | << | >> | + | - | * | / | %
<OpUnary>    ::= <SignUnary> <OpUnary>
             |   <Value> <OpPointer> <OpUnary_>
             |   sizeof  ( <sizeOf> )
<SignUnary>  ::= ! | ~ | - | * | & | ++ | --
<OpUnary_>   ::= ε  |  ++  |  --
<sizeOf>     ::=  <Base> <Pointers>  | Id <Pointers>
<OpPointer>  ::= ε  |  <SignPointer> <Value> <OpPointer> | [ <Expr> ] <OpPointer>
<SignPointer>::= ε  |  .  |  ->
<Value>      ::= <Element> | ( <Expr> )
<Element>    ::= OctLiteral | HexLiteral | DecLiteral | StringLiteralAps
             |   StringLiteralAsp | CharLiteral | FloatLiteral
             |   #18 #1 <Mod> #9 <VarDecl_> #2
             |   #18 #1 <Base> <Pointers> <Element_>
             |   #18 #1 Id #13 <Pointers> <Element_>
<Element___> ::= ε  |  <Var> <VarList> #4 #2
<Element_>   ::= ε #5
             |   ( <Element__> ) #5
             |   { <StructDef> } #4 #10 #2
             |   #13 <Pointers>  <Var> <Array> <VarList> #4 #2
             |   #5 sizeof  ( <sizeOf> )
```

Quadro 48 – Especificação adaptada do C ANSI

APÊNDICE B – Especificação da linguagem de *DocComments*

No quadro 49 é apresentada a especificação da linguagem de *DocComments*.

```
#RegularDefinitions
letra           : [a-zãâáíç]
Ascii_          : .
Ascii_semArroba : [^@\n\t\s\r]
Whitespace      : [\n\t\s\r]
```

```
#Tokens
Id : {Ascii_semArroba} {Ascii_}*

// Tags
tag : @{letra}+:

algoritmo     = tag : "@algoritmo:"
autor         = tag : "@autor:"
cabecalho     = tag : "@cabeçalho:"
data          = tag : "@data:"
descontinuado = tag : "@descontinuado:"
descricao     = tag : "@descrição:"
desde         = tag : "@desde:"
entrada       = tag : "@entrada:"
fonte         = tag : "@fonte:"
link          = tag : "@link:"
motivacao     = tag : "@motivação:"
orientação    = tag : "@orientação:"
parametro     = tag : "@parâmetro:"
retorno       = tag : "@retorno:"
saida         = tag : "@saída:"
versao        = tag : "@versão:"

// Delimitadores
inicio : /\*@{Whitespace}*
fim    : \*/
```

```
#Grammar
<DocComment> ::= inicio <Comments> fim
<Comments>   ::= ε  |  <Comment> <Comments>
<Comment>    ::= #1 <Tag> #2 <IdList>
<Tag>        ::= algoritmo #10
              |   autor #11
              |   cabecalho #12
              |   data #13
              |   descontinuado #14
              |   descricao #15
              |   desde #16
              |   entrada #17
              |   fonte #18
              |   link #19
              |   motivacao #20
              |   orientacao #21
              |   parametro #22
              |   retorno #23
              |   saida #24
              |   versao #25
<IdList>     ::= ε  |  Id #3 <IdList>
```

Quadro 49 – Especificação da linguagem de *DocComments*

APÊNDICE C – Especificação da linguagem de *templates*

No quadro 50 é apresenta a especificação da linguagem de *templates*.

```
#RegularDefinitions
Ascii_              : .
letra               : [a-zâãáíóõç\-]
whitespace          : [\n\t\s\r]
#Tokens
identificador       : {Ascii_}
propriedade         : @{letra}+
componente          : @{letra}+"."{letra}+
instrucao           : #{letra}+

// Símbolos especiais
(  )  =
#Grammar
<template>          ::= <elemento> <lista>
<lista>             ::= ε #18 |  <elemento> <lista>
<elemento>          ::= <simbolo>  |  propriedade |  componente  |  <comando>
<simbolo>           ::= identificador  |  instrucao  |  =  |  (  |  )
<comando>           ::= #18 <defina>
                    |   #18 <se>
                    |   #18 <para_cada>
                    |   #18 <componente_com_propriedades>
<defina>            ::= #defina ( @comentário = <componente_com_comentario> ) #5
<se>                ::= #se #1 ( <expressao> )  #3 <lista> #fecha #2 #5
                    |   #se #1 ( @projeto #16 ) #3 <lista> #fecha #2 #5
<expressao>         ::= @arquivo.possui #10 ( <propriedade_do_arquivo> )
                    |   @função.possui  #11 ( <propriedade_da_funcao> )
                    |   @struct.possui  #12 ( <propriedade_do_struct> )
                    |   @tipo.possui    #13 ( <propriedade_do_tipo> )
                    |   @união.possui   #14 ( <propriedade_da_uniao> )
                    |   @comentário.possui #15 ( <propriedade_do_comentario> )
<para_cada>         ::= #para_cada #8 ( <componente> ) #9
                              <lista> #19
                         #fecha #17 #5
//-------------------------------------------------------------------------------
<componente>        ::= @tipo       #100
                    |   @enumerador #101
                    |   @struct     #102
                    |   @include    #103
                    |   @união      #104
                    |   @função     #105
                    |   @diretiva   #106
                    |   @variável   #107
                    |   @arquivo    #108

<propriedade_do_arquivo>::= @diretivas #20
                    |   @diretório    #21
                    |   @enumeradores #22
                    |   @extensão     #45
                    |   @funções      #23
                    |   @includes     #24
                    |   @nome         #25
                    |   @structs      #26
                    |   @tipos        #27
                    |   @uniões       #28
                    |   @variáveis    #29

<propriedade_do_comentario>::= @algoritmo #30
                    |   @autor         #31
                    |   @data          #32
                    |   @descontinuado #33
```

```
                    |   @descrição     #34
                    |   @desde         #35
                    |   @entrada       #36
                    |   @fonte         #37
                    |   @link          #38
                    |   @motivação     #39
                    |   @orientação    #40
                    |   @parâmetro     #41
                    |   @retorno       #42
                    |   @saída         #43
                    |   @versão        #44

<propriedade_da_funcao>::= @variáveis #29  |  @structs #26  |  @uniões #28

<propriedade_da_struct>::= @uniões #28  |  @variáveis #29

<propriedade_do_tipo>::= @variáveis #29

<propriedade_da_uniao>::= @structs #26  |  @variáveis #29

<componente_com_propriedades>::= @arquivo.diretório #50 #5
                    |   @arquivo.nome             #51 #5
                    |   @arquivo.extensão         #82 #5
                    |   @comentário.algoritmo     #52 #5
                    |   @comentário.autor         #53 #5
                    |   @comentário.data          #54 #5
                    |   @comentário.descontinuado #55 #5
                    |   @comentário.descrição     #56 #5
                    |   @comentário.desde         #57 #5
                    |   @comentário.entrada       #58 #5
                    |   @comentário.fonte         #59 #5
                    |   @comentário.link          #60 #5
                    |   @comentário.motivação     #61 #5
                    |   @comentário.orientação    #62 #5
                    |   @comentário.parâmetro     #63 #5
                    |   @comentário.retorno       #64 #5
                    |   @comentário.saída         #65 #5
                    |   @comentário.versão        #66 #5
                    |   @diretiva.declaração      #67 #5
                    |   @diretiva.tipo            #68 #5
                    |   @enumerador.declaração    #69 #5
                    |   @enumerador.tipo          #70 #5
                    |   @função.declaração        #71 #5
                    |   @função.tipo              #72 #5
                    |   @include.arquivo          #73 #5
                    |   @include.extensão         #83 #5
                    |   @struct.declaração        #74 #5
                    |   @struct.tipo              #75 #5
                    |   @tipo.declaração          #76 #5
                    |   @tipo.tipo                #77 #5
                    |   @união.declaração         #78 #5
                    |   @união.tipo               #79 #5
                    |   @variável.declaração      #80 #5
                    |   @variável.tipo            #81 #5

<componente_com_comentario>::= @arquivo.comentário #90
                    |   @diretiva.comentário       #91
                    |   @enumerador.comentário     #92
                    |   @função.comentário         #93
                    |   @include.comentário        #94
                    |   @struct.comentário         #95
                    |   @tipo.comentário           #96
                    |   @união.comentário          #97
                    |   @variável.comentário       #98
```

Quadro 50 – Especificação da linguagem de *templates*

APÊNDICE D – Atributos das classes do pacote `Código fonte C e DocComments`

Nos quadros 40 a 47 são apresentados os atributos das classes do pacote `Código fonte C e DocComments` especificadas para o GDC.

ATRIBUTO	DESCRIÇÃO
`algoritmo`	armazena conteúdo associado ao *tag* GDC `@algoritmo:`
`autor`	armazena conteúdo associado ao *tag* GDC `@autor:`
`data`	armazena conteúdo associado ao *tag* GDC `@data:`
`descontinuado`	armazena conteúdo associado ao *tag* GDC `@descontinuado:`
`descricao`	armazena conteúdo associado ao *tag* GDC `@descrição:`
`desde`	armazena conteúdo associado ao *tag* GDC `@desde:`
`entrada`	armazena conteúdo associado ao *tag* GDC `@entrada:`
`fonte`	armazena conteúdo associado ao *tag* GDC `@fonte:`
`link`	armazena conteúdo associado ao *tag* GDC `@link:`
`motivacao`	armazena conteúdo associado ao *tag* GDC `@motivação:`
`orientacao`	armazena conteúdo associado ao *tag* GDC `@orientação:`
`parametro`	armazena conteúdo associado ao *tag* GDC `@parâmetro:`
`retorno`	armazena conteúdo associado ao *tag* GDC `@retorno:`
`saida`	armazena conteúdo associado ao *tag* GDC `@saída:`
`versao`	armazena conteúdo associado ao *tag* GDC `@versão:`
`ehCabecalho`	indica se o comentário é um cabeçalho ou se está associado a uma diretiva ou a uma declaração

Quadro 40 – Atribuitos da classe `TComment`

ATRIBUTO	DESCRIÇÃO
`tipoDefinicao`	armazena o tipo (`int`, `float`, etc.) associado à definição de tipos
`declaracaoDefinicao`	armazena a declaração da definição de tipos
`tabelaVariaveis`	agrupa objetos da classe `TVariavel` que, por sua vez, contêm informações sobre as variáveis declaradas na definição de tipos
`docComment`	armazena um objeto da classe `TComment`, contendo o comentário associado à definição de tipos

Quadro 41 – Atributos da classe `TDefinicaoTipo`

ATRIBUTO	DESCRIÇÃO
`tipoDiretiva`	armazena o tipo da diretiva
`codigo`	armazena o código associado à diretiva
`docComment`	armazena um objeto da classe `TComment`, contendo o comentário associado à diretiva

Quadro 42 – Atributos da classe `TDiretiva`

ATRIBUTO	DESCRIÇÃO
`tipoEnumerador`	armazena informação de tipo
`declaracaoEnumerador`	armazena a declaração do enumerador
`docComment`	armazena um objeto da classe `TComment`, contendo o comentário associado ao enumerador

Quadro 43 – Atributos da classe `TEnumerador`

ATRIBUTO	DESCRIÇÃO
`tipoFuncao`	armazena o tipo da função (`int`, `void`, etc.)
`declaracaoFuncao`	armazena a declaração da função, exceto o tipo da mesma
`tabelaVariaveis`	agrupa objetos da classe `TVariavel` contendo informações das variáveis declaradas na função
`tabelaStructs`	agrupa objetos da classe `TUniaoStruct` contendo informações de *structs* declarados na função
`tabelaUnioes`	agrupa objetos da classe `TUniaoStruct` contendo informações de uniões declaradas na função
`commentSetado`	indica se o comentário referente à função, foi ou não setado. Essa indicação se faz necessária visto que a função pode ser declarada na parte inicial do código e implementada em qualquer outra parte. Apenas o primeiro *DocComment* associado à mesma é considerado pelo GDC
`docComment`	armazena um objeto da classe `TComment`, contendo o comentário associado à função

Quadro 44 – Atributos da classe `TFuncao`

ATRIBUTO	DESCRIÇÃO
`nomeArquivo`	armazena o nome do arquivo associado ao `include`
`extensaoArquivo`	armazena a extensão do arquivo associado ao `include`
`docComment`	armazena um objeto da classe `TComment`, contendo o comentário associado ao `include`

Quadro 45 – Atributos da classe `TInclude`

ATRIBUTO	DESCRIÇÃO
`tipoUniaoStruct`	armazena o tipo da estrutura (`union` ou `struct`)
`declaracaoUniaoStruct`	armazena a declaração da estrutura (`union` ou `struct`), exceto o tipo da mesma
`tabelaVariaveis`	agrupa objetos da classe `TVariavel`, contendo informações das variáveis declaradas na estrutura (`union` ou `struct`)
`tabelaUnioesStructs`	agrupa objetos da própria classe `TUniaoStruct`, contendo informações de declarações de estruturas (`union` ou `struct`) na estrutura (`union` ou `struct`)
`docComment`	armazena um objeto da classe `TComment`, contendo o comentário associado à declaração da estrutura (`union` ou `struct`)

Quadro 46 – Atributos da classe `TUniaoStruct`

ATRIBUTO	DESCRIÇÃO
`tipoVariavel`	armazena o tipo da variável
`declaracaoVariavel`	armazena a declaração da variável, exceto o tipo da mesma
`docComment`	armazena um objeto da classe `TComment`, contendo o comentário associado à variável

Quadro 47 – Atributos da classe `TVariavel`

ANEXO A – Gramática do C ANSI

No quadro 51 é apresentada a gramática do C ANSI.

```
<Decls>    ::= <Decl> <Decls>  |  ε
<Decl>     ::= <Func Decl>
           |   <Func Proto>
           |   <Struct Decl>
           |   <Union Decl>
           |   <Enum Decl>
           |   <Var Decl>
           |   <Typedef Decl>
! ============================================================================
! Function  Declaration
! ============================================================================
<Func Proto>::= <Func ID> ( <Types> ); | <Func ID> ( <Params> ); | <Func ID> ( );
<Func Decl>::= <Func ID>  ( <Params> ) <Block>
           |   <Func ID>  ( <Id List> ) <Struct Def> <Block>
           |   <Func ID>  ( ) <Block>
<Params>   ::= <Param> , <Params>  |  <Param>
<Param>    ::= const <Type> Id     |  <Type> Id
<Types>    ::= <Type>  , <Types>   |  <Type>
<Id List>  ::= Id , <Id List>      |  Id
<Func ID>  ::= <Type> Id           |  Id
! ============================================================================
! Type Declaration
! ============================================================================
<Typedef Decl> ::= typedef <Type> Id ;
<Struct Decl>  ::= struct Id { <Struct Def> }  ;
<Union Decl>   ::= union  Id { <Struct Def> }  ;
<Struct Def>   ::= <Var Decl> <Struct Def>  |  <Var Decl>
! ============================================================================
! Variable Declaration
! ============================================================================
<Var Decl> ::= <Mod> <Type> <Var> <Var List> ;
           |         <Type> <Var> <Var List> ;
           |   <Mod>        <Var> <Var List> ;
<Var>      ::= Id <Array>  |  Id <Array> = <Op If>
<Array>    ::= [ <Expr> ]  |  [ ]  |  ε
<Var List> ::=  , <Var Item> <Var List>  |  ε
<Var Item> ::= <Pointers> <Var>
<Mod>      ::= extern  |  static  |  register  |  auto  |  volatile  |  const
! ============================================================================
! Enumerations
! ============================================================================
<Enum Decl>::= enum Id { <Enum Def> }  ;
<Enum Def> ::= <Enum Val> , <Enum Def>  |  <Enum Val>
<Enum Val> ::= Id  |  Id = OctLiteral   |  Id = HexLiteral  |  Id = DecLiteral
! ============================================================================
! Types
! ============================================================================
<Type>     ::= <Base> <Pointers>
<Base>     ::= <Sign> <Scalar>  |  struct Id  |  struct { <Struct Def> }
           |   enum Id  |  union Id  |  union { <Struct Def> }
<Sign>     ::= signed   |  unsigned   |  ε
<Scalar>   ::= char | int | short | long | short int | long int | float | double
           |   void
<Pointers> ::= * <Pointers>  |  ε
! ============================================================================
! Statements
! ============================================================================
<Stm>      ::= <Var Decl>
           |   Id :
           |   if ( <Expr> ) <Stm>
```

```
            |   if ( <Expr> ) <Then Stm> else <Stm>
            |   while ( <Expr> ) <Stm>
            |   for ( <Arg> ; <Arg> ; <Arg> ) <Stm>
            |   <Normal Stm>
<Then Stm> ::= if ( <Expr> ) <Then Stm> else <Then Stm>
            |   while ( <Expr> ) <Then Stm>
            |   for ( <Arg> ; <Arg> ; <Arg> ) <Then Stm>
            |   <Normal Stm>
<Normal Stm>::= do <Stm> while ( <Expr> )
            |   switch ( <Expr> ) { <Case Stms> }
            |   <Block>
            |   <Expr>   ;
            |   goto Id  ;
            |   break    ;
            |   continue ;
            |   return <Expr> ;
            |   ;
<Arg>      ::= <Expr>  |  ε
<Case Stms>::= case <Value> : <Stm List> <Case Stms> | default : <Stm List> | ε
<Block>    ::= { <Stm List> }
<Stm List> ::= <Stm> <Stm List>  |  ε
! ========================================================================
! Here begins the C's 15 levels of operator precedence.
! ========================================================================
<Expr>     ::= <Expr> , <Op Assign>           |  <Op Assign>
<Op Assign>::= <Op If> <OP1> <Op Assign>      |  <Op If>
<Op If>    ::= <Op Or> ? <Op If> : <Op If>    |  <Op Or>
<Op Or>    ::= <Op Or> || <Op And>            |  <Op And>
<Op And>   ::= <Op And> && <Op BinOR>         |  <Op BinOR>
<Op BinOR> ::= <Op BinOr> | <Op BinXOR>       |  <Op BinXOR>
<Op BinXOR>::= <Op BinXOR> ^ <Op BinAND>      |  <Op BinAND>
<Op BinAND>::= <Op BinAND> & <Op Equate>      |  <Op Equate>
<Op Equate>::= <Op Equate> <OP2> <Op Compare> |  <Op Compare>
<Op Compare>::=<Op Compare> <OP3> <Op Shift>  |  <Op Shift>
<Op Shift> ::= <Op Shift> <OP4> <Op Add>      |  <Op Add>
<Op Add>   ::= <Op Add> <OP5> <Op Mult>       | <Op Mult>
<Op Mult>  ::= <Op Mult> <OP6> <Op Unary>     | <Op Unary>
<Op Unary> ::= <OP7> <Op Unary>
            |   <Op Pointer> ++
            |   <Op Pointer> --
            |   ( <Type> ) <Op Unary>
            |   sizeof ( <Type> )
            |   sizeof ( Id <Pointers> )
            |   <Op Pointer>
<Op Pointer>::= <Op Pointer> . <Value>
            | <Op Pointer> -> <Value>
            | <Op Pointer> [ <Expr> ]
            | <Value>
<Value>    ::= OctLiteral  |  HexLiteral  |  DecLiteral  |  FloatLiteral
            |   StringLiteral  |  CharLiteral
            | Id ( <Expr> )  |  Id ( )  |  Id  |  ( <Expr> )
<OP1>      ::= =  |  +=  |  -=  |  *=  |  /=  |  ^=  |  &=  |  |=  |  >>=  |  <<=
<OP2>      ::= == |  !=
<OP3>      ::= <  |  <=  |  >  |  >=
<OP4>      ::= << |  >>
<OP5>      ::= +  |  -
<OP6>      ::= *  |  /  |  %
<OP7>      ::= !  |  ~  |  -  |  *  |  & |  ++  |  --
```

Fonte: adaptado de Grammar (2006).

Quadro 51 – Gramática do C ANSI