

UNIVERSIDADE REGIONAL DE BLUMENAU
CENTRO DE CIÊNCIAS EXATAS E NATURAIS
CURSO DE CIÊNCIAS DA COMPUTAÇÃO
(Bacharelado)

**FERRAMENTA DE APOIO A REESTRUTURAÇÃO DE
CÓDIGO FONTE EM LINGUAGEM C++ BASEADO EM
PADRÕES DE LEGIBILIDADE**

TRABALHO DE CONCLUSÃO DE CURSO SUBMETIDO À UNIVERSIDADE
REGIONAL DE BLUMENAU PARA A OBTENÇÃO DOS CRÉDITOS NA
DISCIPLINA COM NOME EQUIVALENTE NO CURSO DE CIÊNCIAS DA
COMPUTAÇÃO — BACHARELADO

DENIS ALBERTO DALMOLIN

BLUMENAU, NOVEMBRO/2000

2000/2-18

FERRAMENTA DE APOIO A REESTRUTURAÇÃO DE CÓDIGO FONTE EM LINGUAGEM C++ BASEADO EM PADRÕES DE LEGIBILIDADE

DENIS ALBERTO DALMOLIN

ESTE TRABALHO DE CONCLUSÃO DE CURSO, FOI JULGADO ADEQUADO
PARA OBTENÇÃO DOS CRÉDITOS NA DISCIPLINA DE TRABALHO DE
CONCLUSÃO DE CURSO OBRIGATÓRIA PARA OBTENÇÃO DO TÍTULO DE:

BACHAREL EM CIÊNCIAS DA COMPUTAÇÃO

Prof. Everaldo Artur Grahl — Orientador na FURB

Prof. José Roque Voltolini da Silva — Coordenador do TCC

BANCA EXAMINADORA

Prof. Everaldo Artur Grahl

Prof. Roberto Heinzle

Prof. Paulo César Rodacki Gomes

SUMÁRIO

SUMÁRIO.....	iii
RESUMO	v
ABSTRACT	vi
LISTA DE FIGURAS	vii
1 INTRODUÇÃO.....	1
1.1 ORIGEM	1
1.2 OBJETIVOS.....	2
1.3 ORGANIZAÇÃO.....	2
2 LINGUAGEM C	3
2.1 HISTÓRICO.....	3
2.2 LINGUAGEM C++.....	4
2.3 ESTRUTURA BÁSICA DE UM PROGRAMA EM C.....	5
2.4 VARIÁVEIS EM C.....	5
2.5 OPERADORES.....	6
2.6 ESTRUTURAS DE REPETIÇÃO.....	7
2.7 ESTRUTURAS DE DECISÃO.....	8
2.8 PROGRAMAÇÃO ORIENTADA A OBJETO.....	10
2.8.1 OBJETOS	10
2.8.2 POLIMORFISMO	11
2.8.3 HERANÇA.....	12
3 QUALIDADE DE SOFTWARE.....	13
3.1 CONSIDERAÇÕES INICIAIS	13

3.2	PADRONIZAÇÃO	15
3.3	LEGIBILIDADE	15
3.4	REESTRUTURAÇÃO	16
3.5	PADRÕES	17
3.6	PADRÃO DE COMPOSIÇÃO DE MÓDULOS	17
3.7	PADRÃO GERAL	18
3.8	PADRÃO DE ESTILO	18
3.9	PADRÃO PARA ESCOLHA DE NOMES	18
4	DESENVOLVIMENTO DO SOFTWARE	20
4.1	ANÁLISE DO PROGRAMA FONTE.....	20
4.2	DEFINIÇÕES DO SOFTWARE	22
4.3	ESPECIFICAÇÕES DO SOFTWARE	24
4.4	ESTRUTURA DO SOFTWARE	28
4.5	FUNCIONAMENTO DO SOFTWARE.....	31
5	CONCLUSÃO.....	36
5.1	CONSIDERAÇÕES FINAIS	36
5.2	LIMITAÇÕES E SUGESTÕES.....	37
	ANEXOS.....	38
	ANEXO 1 - PADRÃO DE COMPOSIÇÃO DE MÓDULOS.....	38
	ANEXO 2 - PADRÃO GERAL	42
	ANEXO 3 - PADRÃO DE ESTILO	51
	ANEXO 4 - PADRÃO PARA A ESCOLHA DE NOMES	60
	REFERÊNCIAS BIBLIOGRÁFICAS	68

RESUMO

Este trabalho consiste no desenvolvimento de uma ferramenta que analisa e faz a reestruturação de código fonte em C++, utilizando padrões de legibilidade, obtidos em pesquisa bibliográfica. A ferramenta permite melhorar a legibilidade e conseqüentemente aumenta a qualidade dos fontes escritos em linguagem de programação C++, através do padrão geral e o padrão de estilo.

ABSTRACT

This study consists a development tool that analyzes and restructuring C++ sources codes, using legibility patterns, obtained in library research. The tool permit of improving the legibility and consequently increasing sources quality writings in language of programming C++, through in general and style patterns.

LISTA DE FIGURAS

1	ESTRUTURA DE UM PROGRAMA EM C	5
2	DIAGRAMA DE CASOS DE USO.....	25
3	DIAGRAMA DE CLASSES.....	26
4	DIAGRAMA DE SEQUÊNCIA	28
5	FLUXOGRAMA DO FUNCIONAMENTO DO SOFTWARE	29
6	TELA PRINCIPAL DO SOFTWARE.....	31
7	TELA DE ABERTURA DE ARQUIVOS PARA REESTRUTURAÇÃO	31
8	TELA PRINCIPAL COM UM ARQUIVO FONTE EM C++ ABERTO	32
9	TELA PARA INFORMAR O TAMANHO DE INDENTAÇÃO DO CÓDIGO FONTE. .	32
10	TELA DE MENSAGENS SOBRE A REESTRUTURAÇÃO DO CÓDIGO FONTE	33
11	TELA DE MENSAGENS POSICIONANDO NA LINHA COM O PROBLEMA	34
12	TELA DO SOFTWARE COM EXEMPLO DE CÓDIGO FONTE ANTES E DEPOIS DA REESTRUTURAÇÃO.	35

1 INTRODUÇÃO

Este capítulo trata das considerações iniciais sobre o trabalho, sua importância e objetivos a serem alcançados e organização do trabalho.

1.1 ORIGEM

Hoje em dia busca-se a qualidade no software a um custo mais baixo. Para obter um produto final com qualidade, a empresa deve estabelecer mecanismos de garantia da qualidade desde o começo de um projeto. Estes mecanismos são difíceis de serem implantados em certas empresas devido a utilização de métodos e ambientes tradicionais pelas equipes de projeto e desenvolvimento de software ([PRE1995]).

As empresas estão reconhecendo a importância econômica do software para a competitividade empresarial e começam a exigir dos seus fornecedores produtos com maior qualidade. As empresas de software por sua vez também estão sentindo que a concorrência está aumentando e obrigam-se a melhorar a qualidade dos seus produtos e a revisar seus métodos e ambiente de desenvolvimento, procurando por padrões de desenvolvimento do código fonte para agilizar e reduzir custos com a manutenção do software ([FER1995]).

Uma das linguagens de programação mais populares atualmente no mercado é o C++. Com a linguagem C++ pode-se construir programas organizados e concisos, ocupando pouco espaço de memória com alta velocidade de execução. Infelizmente, dada toda a flexibilidade da linguagem, também pode-se escrever programas desorganizados e difíceis de serem compreendidos. Esta prática, não tão incomum nas empresas é justamente o foco deste trabalho.

1.2 OBJETIVO

Este trabalho tem como objetivo principal especificar e implementar uma ferramenta que apoie a reestruturação de código fonte escrito em linguagem C++ utilizando padrões de legibilidade. Estes padrões foram obtidos a partir de pesquisa bibliográfica.

1.3 ORGANIZAÇÃO

O trabalho é composto por seis capítulos. Neste primeiro capítulo foram apresentados a origem do trabalho, objetivo e organização.

No segundo capítulo são apresentadas as principais características da linguagem de programação C e C++, além de uma apresentação sobre a programação orientada a objeto, seus conceitos e características.

No terceiro capítulo são apresentados conceitos e características da qualidade de software e os padrões utilizados para a elaboração do software.

No quarto capítulo é descrito a especificação do protótipo, assim como detalhes sobre sua implementação.

No quinto capítulo são apresentadas as conclusões e sugestões sobre o trabalho desenvolvido.

2 LINGUAGEM C

Este capítulo trata da linguagem de programação C e sua evolução, demonstrando seu histórico, principais construções, vantagens na utilização do C. Descreve também a programação orientada a objeto.

2.1 HISTÓRICO

Nos anos 70, a linguagem C foi inventada e implementada pela primeira vez por Dennis Ritchie em um equipamento DEC PDP-11, usando o sistema operacional UNIX. A linguagem C é o resultado do processo de desenvolvimento iniciado com outra linguagem, chamada BCPL, desenvolvida por Martin Richards. Esta linguagem influenciou a linguagem inventada por Ken Thompson, chamada B, a qual levou ao desenvolvimento da linguagem C ([SCH1992]).

A linguagem C é freqüentemente referenciada como uma linguagem de nível médio, posicionando-se entre o assembler (baixo nível) e o Pascal (alto nível). Uma das razões da invenção da linguagem C foi dar ao programador uma linguagem de alto nível que poderia ser utilizada como uma substituta para a linguagem assembler. Suas características segundo [MIZ1994] são:

- a) portabilidade entre máquinas e sistemas operacionais.
- b) dados compostos em forma estruturada.
- c) programas Estruturados.
- d) total interação com o Sistema Operacional.

A linguagem C quando comparada com outras linguagens de complexidade análoga se mostra mais compacta e rápida.

A linguagem C tornou-se uma das linguagens de programação mais usadas (e desejadas). Flexível, ainda que poderosa, a linguagem C tem sido utilizada na criação de

alguns dos mais importantes produtos de software dos últimos anos. Entretanto, a linguagem encontra seus limites quando o tamanho do projeto ultrapassa um certo ponto. Ainda que esse limite possa variar de projeto para projeto, quando o tamanho de um programa se encontra entre 25.000 e 100.000 linhas, torna-se problemático o seu gerenciamento, tendo em vista que é difícil compreendê-lo como um todo. Para resolver este problema, em 1980, enquanto trabalhava nos laboratórios da Bell, em Murray Hill, New Jersey, Bjarne Stroustrup acrescentou várias extensões à linguagem C e chamou inicialmente esta nova linguagem de “C com Classes”. Entretanto, em 1983, o nome foi mudado para C++ ([SCH1992]).

2.2 LINGUAGEM C++

Quando o C++ foi inventado ([SCH1992]), Bjarne Stroustrup sabia que era importante manter o espírito original da linguagem C, incluindo a eficiência, a natureza de nível médio e a filosofia de que o programador, não a linguagem, está com as responsabilidades, enquanto, ao mesmo tempo, acrescenta o suporte à programação orientada ao objeto. Assim, o C++ proporciona ao programador a liberdade e o controle da linguagem C junto com o poder dos objetos. As características da orientação ao objeto em C++, usando as palavras de Stroustrup, “permite aos programas serem estruturados quanto a clareza e extensibilidade, tornando fácil a manutenção sem perda de eficiência”.

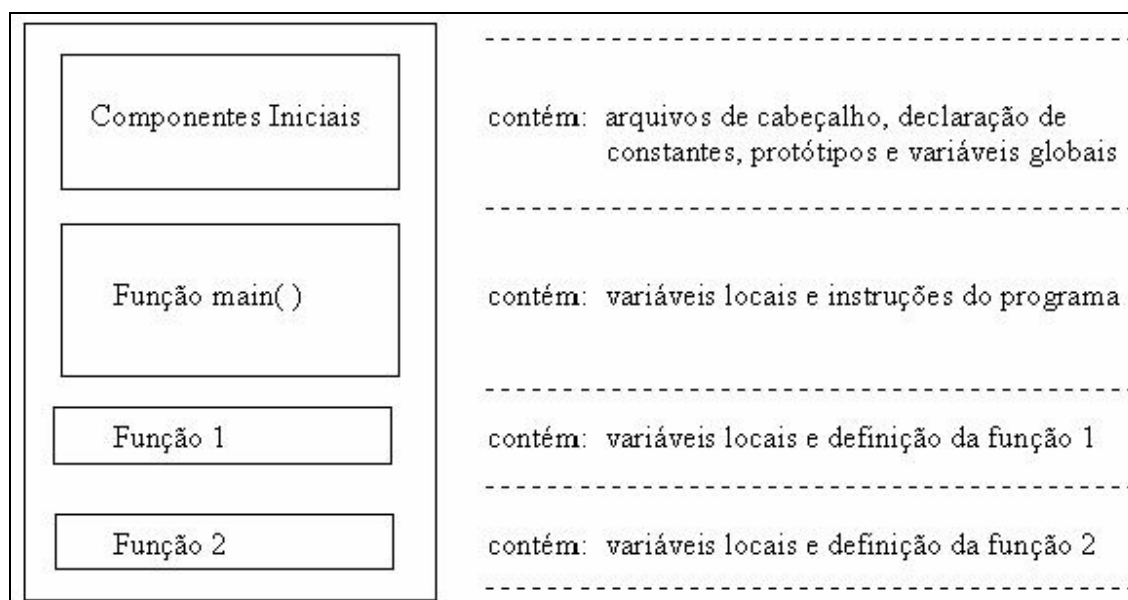
Segundo [HOL1994], o que tornou o C++ famoso foi a capacidade de trabalhar com objetos. A linguagem C++ serve exatamente para isto: dividir programas grandes em objetos gerenciáveis – e independentes. Esses objetos poderão cuidar de todas as operações que eles precisam para si mesmos – tais como gravar a si próprios num arquivo em disco. Dessa forma, pode-se pensar em objetos em termos de seu uso geral, sem ter de lembrar todos os detalhes de seu manuseio interno. Na verdade, um objeto é muito parecido com um novo tipo de estrutura – exceto que ele pode conter tanto dados quanto funções.

2.3 ESTRUTURA BÁSICA DE UM PROGRAMA C

Os conceitos e comandos sobre a linguagem de programação C encontrados à partir deste capítulo, foram obtidos através de manuais e tutoriais encontrados em [PIR2000] e [AID2000].

A estrutura de um programa em C consiste em uma coleção de funções, onde a função `main ()` é a primeira função a ser executada, `{` é o início da função e `}` é o fim da função. A função `main ()` tem que existir em algum lugar, ela marca o início da execução, como demonstrado na figura 1.

FIGURA 1 – ESTRUTURA DE UM PROGRAMA EM C



2.4 VARIÁVEIS EM C

Os nomes de variáveis podem conter letras, números e caracter de sublinhado. Porém:

a) o primeiro caracter não pode ser número:

ex.: `br_01`, `_br01`
`01_br` (NÃO é permitido)

b) letras maiúsculas são diferentes de letras minúsculas (**convenção : minúsculas**)

ex.: `A1` é diferente de `a1`

- c) não podemos usar palavras reservadas
ex.: int, float, if, else, etc...

Os tipos básicos em na linguagem C são:

- char** - apenas 1 caracter alfanumérico (geralmente ocupa 1 byte)
int - n°s inteiros ex.: 7 (geralmente ocupa 2 bytes)
float - n°s fracionários com precisão simples ex.: 7.5 (geralmente ocupa 4 bytes)
double - n°s fracionários com precisão dupla (geralmente ocupa 8 bytes)
void - indica que não retorna nada.

Na declaração de variáveis utiliza-se a seguinte sintaxe:

tipo nome_variável ;

ex.: int x, y ;
float f ;

Para a inicialização de variáveis utiliza-se a sintaxe:

nome_variável = valor ;

ex.: x = y = 10 ;
f = 3.5 ;

2.5 OPERADORES

Os operadores de atribuição possui a sintaxe:

nome_variável = expressão ;

ex.: y = 2 ; /* atribui o valor 2 a y */
x = 4 * y + 3 ; /* atribui o valor da expressão a x */

Os operadores aritméticos são divididos em operadores unários:

a) Operadores Unários – que atuam sobre apenas um operando

- (**menos unário**) multiplica o operando por (-1)
- ++ (**incremento**) incrementa o operando em uma unidade
- (**decremento**) decrementa o operando em uma unidade

ex.: x = 2 ; e y = 4*x + 3 ;

b) Operadores Binários – atuam sobre dois operandos

- + (adição)
- (subtração)
- * (multiplicação)
- / (divisão)

% (mod) - fornece o resto da divisão de 2 n^os inteiros

ex.: 10 % 2 = **0**, 11 % 2 = **1**

Os operadores de Atribuição Compostos possuem a seguinte sintaxe:

expressão_1 operador = expressão_2 é equivalente a
expressão_1 = expressão_1 operador expressão_2

ex.: a = a + 1 **a += 1** ou a ++
y = y - 1 **y -= 1** ou --y

Os operadores Relacionais são usados para comparar expressões. Resultam em falso ou verdadeiro.

= (igual – comparação) - compara se 2 valores são iguais
> (maior que)
< (menor que)
>= (maior ou igual)
<= (menor ou igual)
!= (diferente)

ex.: 4 == 3 /* resulta em falso */
3 > 2 /* resulta em verdadeiro */

Os operadores Lógicos permitem relacionar duas ou mais expressões.

&& (e) - resulta em verdadeiro se ambas expressões forem verdadeiras
|| (ou) - resulta em verdadeiro se pelo menos uma expressão for verdadeira
! (não) - resulta em verdadeiro se a expressão for falsa

ex.: (5 > 2) && (3 != 2) /* resulta em verdadeiro – ambos verdadeiros */
(3 >= 2) || (4 == 2) /* resulta em verdadeiro – pelo menos 1 verdadeiro */
!(4 == 2) /* resulta em verdadeiro – pois a expressão é falsa */

2.6 ESTRUTURAS DE REPETIÇÃO

As principais estruturas de repetição são: **for**, **while**, **do-while**.

Sintaxe do **while**:

while (expressão de teste)
Instrução;

Ex:

```
main ( )
{
    int num = 0;
    while (num < 3)
        printf (" %d", num++);
}
```

Sintaxe do **for**:

```
for ( inicialização; teste; incremento )
```

Ex:

```
main ( )
{
    int num;
    for (num = 1; num <= 1000; num++)
        printf ( " % d", num);
}
```

Sintaxe do **do-while**:

```
do
{
    instrução;
} while (expressão de teste);
```

Ex:

```
main ( )
{
    char ch;
    do
    {
        printf ("digite uma letra");
        printf ("continua? (s / n):");
    } while (getche( ) == 's');
}
```

2.7 ESTRUTURAS DE DECISÃO

As estruturas de decisão devem permitir testes para decidir ações alternativas. Os principais comandos são: if, if - else, switch e Operador Condicional (?:)

O COMANDO if

Forma Geral: if (condição)
instrução;

```
main ( )
{
    char ch;
    ch = getche ( );
    if (ch == 'p')
        printf ("você pressionou a tecla p");
}
```

O COMANDO if - else

O comando if só executa a instrução caso a condição de teste seja verdadeira, nada fazendo se a expressão for falsa, o comando else executará um conjunto de instruções se a expressão de teste for falsa.

Forma Geral: if (condição)
 instrução
 else
 instrução

```
main ( )
{
    if (getche ( ) == 'p')
        printf (" você digitou p");
    else
        printf (" você não digitou p");
}
```

O COMANDO switch

O comando switch é uma forma de substituir o comando if - else ao se executar vários testes. Similar ao if - else com maior flexibilidade e formato limpo.

Forma Geral: switch (expressão)
 {
 case constante 1:
 instruções; /* opcional */
 break; /* opcional */
 case constante 2:
 instruções
 break;
 default:
 instruções
 }

```
main ( )
{
    float num1;
    printf (" digite um n.º");
    scanf ("%f", &num1);
    switch (num1)
    {
        case '1':
            printf ("digitou 1");
            break;
        case '2':
            printf ("digitou 2");
            break;
        default:
            printf ("digitou outro");
    }
}
```


O OPERADOR CONDICIONAL TERNÁRIO ?:

É uma forma compacta de expressar uma instrução if – else.

Forma Geral: (Condição) ? expressão 1 : expressão 2

Max = (num1 > num2) ? num1 : num2;

Exemplo:

ABS = (num < 0) ? - num : num;

2.8 PROGRAMAÇÃO ORIENTADA A OBJETO

A programação orientada a objeto aproveitou as melhores idéias da programação estruturada e combinou-as com novos conceitos. A programação orientada a objeto permite que um problema seja mais facilmente decomposto em subgrupos relacionados ([SCH1992]).

Todas as linguagens de programação orientadas a objeto possuem três coisas em comum: objetos, polimorfismo e herança ([SCH1992]).

2.8.1 OBJETOS

A característica mais importante de uma linguagem orientada a objeto é o objeto. De maneira simples, um *objeto* é uma entidade lógica que contém dados e código para manipular esses dados. Dentro de um objeto, alguns códigos e/ou dados podem ser privados ao objeto e inacessíveis diretamente para qualquer elemento fora dele. Dessa maneira, um objeto evita significativamente que algumas outras partes não relacionadas de programa modifiquem ou usem incorretamente as partes privadas do objeto. Essa ligação dos códigos e dos dados é freqüentemente referenciada como encapsulamento ([SCH1992]).

Exemplo de um objeto em C++:

```
class CToken
{
public:
    CToken();
    virtual ~CToken();
```

```
protected:
    char  TokenName;
    int   TokenType;
};
```

2.8.2 POLIMORFISMO

Linguagens de programação orientadas a objeto suportam *polimorfismo*, o que significa essencialmente que um nome pode ser usado para muitos propósitos relacionados, mas ligeiramente diferentes. A intenção do polimorfismo é permitir a um nome ser usado para especificar uma classe geral de ações. Entretanto, dependendo do tipo de dado que está sendo tratado, uma instância específica de um caso geral é executada. Por exemplo, pode-se ter um programa que define três tipos diferentes de pilha. Uma pilha é usada para valores inteiros, uma para valores de ponto flutuante e uma, para inteiros longos. Por causa do polimorfismo, pode-se criar três conjuntos de funções para essas pilhas, chamadas **põe()** e **tira()**, e o compilador selecionará a rotina correta, dependendo com qual tipo a função é chamada. Nesse exemplo, o conceito geral é pôr e tirar dados de e para a pilha. As funções definem a maneira específica como isso é feito para cada tipo de dado ([SCH1992]).

Exemplo de polimorfismo em C++:

```
class CInteiro
{
public:
    void SetaValor (int var);

protected:
    int  Variavel;
};

class CLongo
{
public:
    void SetaValor (long var);
protected:
    long  Variavel;
};
```

Observe que no exemplo anterior quando se chama a função `SetaValor`, com um parâmetro `long`, ela vai ser tratada na classe `CLongo`, e quando chamar com o parâmetro `int`, ela vai ser tratada na classe `CInteiro`.

2.8.3 HERANÇA

Herança é o processo em que um objeto pode adquirir as propriedades de outro objeto. Sendo importante, pois suporta o conceito de classificação. Considerando-o por classificações hierárquicas, mais conhecimento torna-se gerenciável. Por exemplo, uma deliciosa maçã vermelha é parte da classificação *maçã*, que, por sua vez, faz parte da classificação *frutas*, que está na grande classe *comida*. Sem o uso de classificações, cada objeto precisaria definir explicitamente todas as suas características. Portanto, usando-se classificações, um objeto precisa definir somente aquelas qualidades que o tornam único dentro da classe. Ele pode herdar as qualidades que compartilha com a classe mais geral. É o mecanismo de herança que torna possível a um objeto ser uma instância específica de uma classe mais geral.

Exemplo de herança em C++:

```
class Frutas
{
public:
    Frutas();
    void Comer();
    ...
}

class Banana : public Frutas
{
public:
    Banana();
    void Amassar();
    ...
};
```

Observe no exemplo anterior que a classe *Banana* é derivada da classe *Frutas*, a classe *Banana* está herdando propriedades e métodos da outra classe mais geral chamada *Frutas*.

3 QUALIDADE DE SOFTWARE

Este capítulo trata sobre a qualidade de software e sua importância no desenvolvimento e manutenção de programas, a influência da legibilidade e da reestruturação de software na melhoria da qualidade de software e padrões de legibilidade pesquisados para programação em linguagem C++.

3.1 CONSIDERAÇÕES INICIAIS

Ao longo da década de 1980, segundo [PRE1995] os principais objetivos da área de computação estavam voltados para o avanço do hardware. Houve uma crescente evolução no hardware ocasionando em um aumento da capacidade de processamento e armazenamento de informação a um custo cada vez mais baixo. Hoje se procura resolver outro problema que é a busca da qualidade do software a um custo mais baixo.

Durante os primeiros anos de sua existência, o software era projetado sob medida e produzido por funcionários da própria empresa que o utilizava. O projeto só existia na mente de alguém. A manutenção era feita pela mesma pessoa que desenvolvia o software e a documentação era considerada desnecessária ([PRE1995]).

Com o crescimento do tamanho e da complexidade dos programas, bem como com o desenvolvimento de programas para fins econômicos, tornou-se evidente a necessidade de utilizar um instrumental mais poderoso do que simplesmente arte e intuição. Assim sendo, apenas recentemente é que o estudo aprofundado do processo de programação tornou-se interessante. Como consequência deste estudo, existem hoje diversas propostas de métodos para o desenvolvimento de programas com elevado nível de qualidade ([STA1983]).

Para obter um produto final com qualidade, a empresa deve estabelecer mecanismos de garantia da qualidade desde o começo de um projeto. Estes mecanismos são difíceis de serem implantados em certas empresas devido a utilização de métodos e ambientes tradicionais pelas equipes de projeto e desenvolvimento de software. Dados estatísticos indicam que entre 50%

e 70% de todo o esforço gasto num programa acontece depois que ele é entregue ao cliente ([PRE1995]).

De uma forma genérica pode-se dizer que um software de boa qualidade produz resultados úteis e confiáveis na oportunidade certa; é ameno ao uso; é mensurável e auditável; é corrigível, modificável, e volutível; opera em máquinas e ambientes reais; foi desenvolvido de forma econômica e no prazo estipulado; e opera com economia de recursos. Qualidade de software é, pois, um conceito muito mais amplo do que software correto e bem documentado, requerendo, para ser conseguida, métodos e técnicas de desenvolvimento específicas ([STA1983]).

Atualmente, as empresas estão reconhecendo a importância econômica do software para a competitividade empresarial e começam a exigir dos seus fornecedores, produtos com maior qualidade. As empresas de software por sua vez também estão sentindo que a concorrência está aumentando e se obrigam a melhorar a qualidade dos seus produtos e a revisar seus métodos e ambiente de desenvolvimento ([FER1995]).

É possível desenvolver software de boa qualidade mesmo trabalhando de forma pouco sistemática ([STA1983]). Entretanto, estes são casos raros e, usualmente, são conseguidos por profissionais muito talentosos. Interessa-nos elevar o nível médio de qualidade dos softwares desenvolvidos, aumentar a produtividade do desenvolvimento, sem, no entanto, requerer profissionais difíceis de serem encontrados. Necessita-se pois, instrumentos que assegurem uma melhor qualidade quando aplicados de modo sistemático.

Um software de boa qualidade tenderá a ter uma maior rentabilidade, do que um software funcionalmente semelhante porém de baixa qualidade. Porém, se for exagerado nos níveis de qualidade, a rentabilidade poderá vir a ficar comprometida. Isto reforça a necessidade de se projetar a qualidade desejável, de modo que não se venha a ter prejuízos decorrentes, tanto da falta como do excesso de qualidade ([STA1983]).

A falta de boas especificações tem levado a insatisfações e frustrações, a programas que não fazem o que se deseja, a programas resistentes a alterações (manutenção), a programas remendados e de baixa qualidade e a um considerável volume de esforço perdido.

Em suma, é na falta de especificações adequadas que se localiza uma das principais causas da existência de programas de baixa qualidade e com custos de desenvolvimento e operação elevados ([STA1983]).

O aumento da concorrência e crescentes exigências do mercado global em solicitar produtos com maior qualidade leva as organizações a procurarem padrões de desenvolvimento do código fonte para agilizar e reduzir custos com a manutenção do software.

3.2 PADRONIZAÇÃO

Infelizmente os engenheiros de software parecem ter uma certa repulsa sobre a palavra padronização. Um padrão é “algo estabelecido por uma autoridade, cliente, ou pelo consenso geral, para um modelo ou exemplo”. Entretanto com o melhoramento da qualidade, os problemas são resolvidos por pessoas que conhecem tanto o trabalho quanto o problema. Um padrão é normalmente o menor denominador comum pelo qual todos podem concordar em usar. A frase “tente agradar a todos e você não agradará a ninguém”, vem imediatamente à nossa mente. O que se quer realmente é consistência, mas não necessariamente uma forma idêntica de aplicação dos melhoramentos por toda a organização do sistema ([ART1994]).

3.3 LEGIBILIDADE

Legibilidade diz respeito às características lexicais das informações apresentadas que possam dificultar ou facilitar a leitura desta informação (brilho do caracter, contraste letra/fundo, tamanho da fonte, espaçamento entre palavras, espaçamento entre linhas, espaçamento de parágrafos, comprimento da linha, etc.) ([LAB2000]).

A performance melhora quando a apresentação da informação leva em conta as características cognitivas e perceptivas dos usuários. Uma boa legibilidade facilita a leitura da informação apresentada.

3.4 REESTRUTURAÇÃO

Segundo [FUR1994], reestruturação é uma área da Reengenharia da Informação que trata do processo de padronização de nome de dados e estruturação de programas.

Existem basicamente dois tipos de reestruturação:

- a) reestruturação de código fonte: análise dos fluxos de controle e lógica de programação com geração de uma versão estruturada do código-fonte original sem alterações de sua funcionalidade;
- b) reestruturação de dados: visa eliminar redundâncias de nomes para a mesma lógica de dados, adotando-se um nome padrão para os elementos em nível de análise de área de negócio.

A reestruturação do código-fonte contribui para a melhoria da produtividade na manutenção de sistemas, mas em alguns casos, poderá ter efeito contrário. Quando alguém conhece um programa desestruturado, normalmente não mais irá reconhecê-lo após uma reestruturação. Já no caso de não conhecer o programa, certamente encontrará mais facilidade em compreendê-lo após tal reestruturação. Um questão fundamental na reestruturação é que preservar o conhecimento é mais importante do que melhorar sua reestruturação.

Para se reestruturar programas ou módulos, primeiramente deve-se analisar, revisar e corrigir e, em seguida, gerar automaticamente o programa de forma estruturada. O processo automatizado de reestruturação de programas cria por sua vez alguns problemas colaterais do tipo: os programas tendem a se tornar maiores, as sentenças deixam de ser familiares.

Segundo [FUR1994], as ferramentas para reestruturação tem sido empregadas com sucesso para a melhoria da produtividade no processo de manutenção, redução de defeitos, realocação de pessoal técnico e posicionamento do código para análises subsequentes. Tais ferramentas utilizadas por profissionais adequadamente capacitados, podem desempenhar um papel-chave na preparação dos sistemas para eventual recuperação do desenho ou reutilização.

3.5 PADRÕES

Serão apresentados os tópicos de regras e recomendações aplicáveis às diversas etapas do desenvolvimento de programas, para visualizar as regras de cada padrão você pode consultar o Anexo 1, 2, 3 e 4 deste trabalho. Estes padrões estudados foram obtidos do livro de [STA2000].

O objetivo é fornecer aos projetistas e aos programadores padrões básicos visando construir programas de boa qualidade. Estes padrões foram desenvolvidos sem visar uma organização (empresa) específica, portanto devem ser adaptados às características específicas de cada organização ([STA200]).

Os principais padrões de legibilidade abordados são:

- a) padrão de composição de módulos;
- b) padrão geral;
- c) padrão de estilo;
- d) padrão para a escolha de nomes.

3.6 PADRÃO DE COMPOSIÇÃO DE MÓDULOS

Este padrão trata de regras e recomendações de como criar e declarar módulos de definição e de implementação em C e C++.

O padrão de composição de módulos tem como principais objetivos:

- a) assegurar a existência de definições de interfaces entre módulos;
- b) assegurar a consistência dos módulos que compartilhem uma mesma interface.

3.7 PADRÃO GERAL

Este padrão trata de regras e recomendações de como utilizar a linguagem padrão , declaração de protótipos e de cabeçalhos de funções, fazer declarações, o uso de expressões e comandos e sobre sequência de execução de comandos em C++, e tem por objetivo reduzir a frequência de erros e defeitos mais corriqueiros em programação.

3.8 PADRÃO DE ESTILO

Este padrão trata de regras e recomendações sobre o estilo de programação como o espaçamento da margem esquerda, estilo de declarações de variáveis, estilo de expressões e atribuições, estilo de blocos e estruturas de controle.

O padrão de estilo tem como principais objetivos:

- a) uniformizar o estilo de programação da organização;
- b) reduzir a necessidade de treinamento ao alocar um novo programador;
- c) facilitar a compreensão, manuseio e modificação de programas escritos por outros.

3.9 PADRÃO PARA A ESCOLHA DE NOMES

Este padrão trata sobre a estrutura genérica de um nome, sobre prefixos de componente e de módulo, prefixos de tipo, temas, abreviações, sufixo de tipo, variáveis de rascunho, constantes enumeradas e possui ainda uma tabela de identificadores de tipos primários, tabela de abreviações e palavras padronizadas, tabela de verbos identificadores de funções e métodos, e de sufixos de tipo padronizado.

O padrão para escolha de nomes e tem como principais objetivos:

- a) uniformizar a forma de escolher nomes ao desenvolver ou manter programas redigidos em C ou C++;

- b) tornar a criação de novos nomes independente do redator do programa;
- c) facilitar a lembrança e o entendimento dos nomes dos elementos, reduzindo a necessidade de consulta a documentação complementar;
- d) facilitar a verificação do correto uso dos elementos do programa;
- e) facilitar a localização do documento e a correspondente seção, contendo a especificação e a declaração completa do elemento denominado.

4 DESENVOLVIMENTO DO SOFTWARE

Este capítulo trata sobre o software desenvolvido, suas definições, especificações e implementação e descreve também sobre o funcionamento passo a passo do software.

4.1 ANÁLISE DO PROGRAMA FONTE

O estudo do padrão de composição de módulos não gerou nenhuma implementação no software aqui proposto, pois ele relata como deve ser nomeado arquivos escritos em C e C++, como deve-se declarar módulos de definição e implementação, e como organizar as declarações e implementações, tornando difícil a reestruturação do código.

O estudo do padrão geral gerou a implementação de mensagens de avisos, de recomendações e de erro no software implementado. Como por exemplo a utilização de comandos não recomendados e não permitidos, que para não alterar o fluxo e a integridade do programa a ser reestruturado, estão gerando mensagens para o usuário.

Muitas das regras contidas no padrão de estilo foram utilizadas para a implementação do software aqui proposto, devido o estilo de programação do programador ser mais previsível do que regras e recomendações de outros padrões aqui estabelecidos.

O estudo do padrão para escolha de nomes não gerou nenhuma implementação para o software aqui proposto, pois não se sabe sobre o escopo do programa ou o que uma função ou procedimento deverá fazer, tornando a reestruturação de nomes muito complexa ou mesmo impossível.

Para o desenvolvimento do software, seguiu-se a mesma linha da análise de código fonte feito por compiladores. No processo de compilação, a análise compõe-se de três fases ([ZSC1999]):

- a) Análise linear, na qual o programa fonte é percorrido e o analisador extrai os elementos básicos da linguagem chamados tokens. Os tokens são seqüências de caracteres com sentido coletivo.
- b) Análise hierárquica, na qual os tokens são agrupados hierarquicamente em coleções aninhadas.
- c) Análise semântica, na qual certos testes são efetuados para se certificar que os componentes do programa estão agrupados de uma forma correta.

O analisador linear ou léxico é comumente chamado *scanner* e o processo de análise léxica, portanto, é chamado *scanning*. Na análise léxica, o programa fonte é percorrido e o analisador extrai os elementos básicos da linguagem chamados tokens. Os tokens são as palavras que formam a linguagem. Por exemplo, na linguagem C++ alguns tokens seriam `if`, `while`, `{`, `}`, `=`, identificadores, constantes, etc. O analisador léxico é chamado por demanda pelo analisador sintático, como será visto mais adiante. A cada vez que o léxico for chamado, devolve um token válido da linguagem. Ocasionalmente, quando detectar um símbolo não reconhecido, deve ser capaz de retornar o erro apropriado ([ZSC1999]).

O analisador léxico lê as linhas e procura os tokens definidos. No exemplo abaixo o token que esta sendo procurado é o `{`, ao ser encontrado na linha ele é inserido da lista de tokens:

```
int Pos = Linha.Find("{");
if (Pos != -1)
{
    CToken token;
    token.PosicaoInicial = Pos + 2;
    token.PosicaoFinal   = token.PosicaoInicial + 1;
    token.TokenType     = inicioT;
    token.NumLinha      = NumLinha;
    token.Acumulo        = 0;
    token.TokenName     = "{";
    LexicalAnalyser.AddToken(token);
}
```

Segundo [ZSC1999] o analisador hierárquico ou sintático é comumente chamado de *parser* e, portanto, o processo de análise sintática é chamado de *parsing*. Na análise sintática, os tokens são agrupados em uma estrutura de árvore chamada árvore sintática que expressa a

ordem em que esses elementos devem ser avaliados, e essa árvore deve expressar a precedência (ordem em que operações devem ser feitas, por exemplo, '*' antes de '+') e a associatividade (ordem em que operações devem ser feitas, considerando a mesma precedência) dos operadores de uma expressão da linguagem. Se um token não pode ser encaixado nessa árvore sintática, então ocorre um erro sintático, significando que o programa fonte está incorreto, pois não corresponde à definição da linguagem que o compilador deve reconhecer. Portanto, deve-se entender o analisador sintático como um reconhecedor de linguagens.

O analisador hierárquico percorre os tokens encontrados e conforme o tipo faz o tratamento adequado. No exemplo abaixo o token que está sendo analisado é o if, onde é verificado o próximo caracter, se for um abre parênteses coloca um espaço:

```
for (int i=0; i<QtdTokens; i++)
{
    if (Token->TokenType == ifT)
    {
        //verifica se proximo caracter = (
        if (Fonte.GetAt(Token->PosicaoFinal) == '(')
        {
            Fonte.Insert(Token->PosicaoFinal, " ");
        }
    }
}
```

4.2 DEFINIÇÕES DO SOFTWARE

O software desenvolvido efetua uma análise sobre os tokens coletados e gera as seguintes análises no código fonte em C++:

- 1) Análise que gera avisos: neste tipo de análise o software irá avisar ao usuário sobre comandos não permitidos, e construções que não estão de acordo com algum padrão estabelecido;
- 2) Análise que gera modificações: neste tipo de análise o software irá copiar o texto do código fonte em C++ e irá modifica-lo de acordo com os padrões estabelecidos.

O software irá fazer as consistências com relação à Análise que gera avisos, e irá gerar as seguintes mensagens:

- 1) Verificar a existência do comando ***goto*** e recomendar não utilizar;
- 2) Verificar a existência do comando ***?*** e recomendar não utilizar;
- 3) Verificar a existência do comando ***continue*** e recomendar não utilizar;
- 4) Verificar a existência do comando ***friend*** e recomendar não utilizar;
- 5) Verificar a existência do comando ***malloc*** e recomendar utilizar o comando ***new***;
- 6) Verificar a existência do comando ***realloc*** e recomenda utilizar o comando ***new***;
- 7) Verificar a existência do comando ***free*** e recomenda utilizar o comando ***delete***;
- 8) Verificar a inexistência do comando ***break*** no final do bloco do comando ***case*** e recomenda colocar o comando;
- 9) Verificar a inexistência do comando ***default*** no final do bloco do comando ***switch*** e recomenda colocar o comando;
- 10) Verificar a inexistência do comando ***else*** final em um bloco dos comandos ***if*** e ***else if*** e recomenda colocar o comando;
- 11) Verificar no bloco do comando ***case*** se o tamanho do bloco for maior que 5 linhas, e então recomendar a criação de função;
- 12) Verificar o tamanho da linha é maior que 100 caracteres e recomendar dividir a linha.
- 13) Verificar em expressões a existência de atribuições (ex.: `vTC[i++] = A`), e recomendar não utilizar.

O software irá fazer as consistências com relação à Análise que gera modificações, e irá gerar as seguintes alterações:

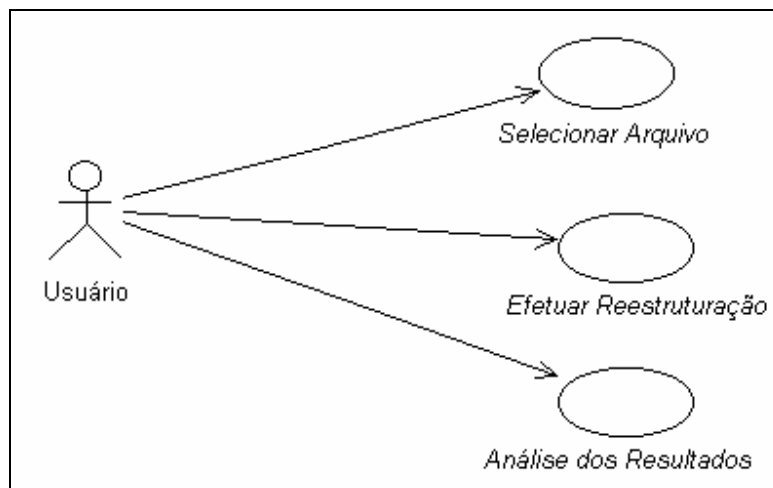
- 1) Fazer o alinhamento do texto do código fonte em C++ de acordo com o valor o tamanho da indentação informado antes da reestruturação;
- 2) Verificar se cada variável está declarada em um linha individual, senão colocar cada variável em linha individual;
- 3) Na declaração de variáveis, separar o comando *, indicador de tipo ponteiro, do nome da variável e do tipo da variável;
- 4) Nas estruturas de controle (if, for, while, switch, ...), verificar se antes do abre parênteses da condição existe um espaço, caso não existe separar colocando um espaço.
- 5) Nas expressões de condição, quando haver operadores lógicos (||, &&), inserir uma nova linha na condição após os operadores;
- 6) Alinhar os delimitadores de inicio e fim de bloco (*{, }*) com o texto do código fonte em C++;
- 7) Todos as estruturas de controle devem ser seguidos de um início de bloco e posteriormente de final de bloco;
- 8) Os delimitadores de início e fim de bloco (*{, }*) devem estar em uma linha só para eles.

4.3 ESPECIFICAÇÕES DO SOFTWARE

O software foi especificado utilizando a análise orientada à objeto e mais especificamente usou-se a UML (*Unified Modeling Language*), com suporte da ferramenta CASE Rational Rose.

Para o software foi elaborado o diagrama de Casos de Uso apresentado na figura 2. Além disso, foi elaborado também o diagrama de Classes da figura 3:

FIGURA 2 - DIAGRAMA DE CASOS DE USO

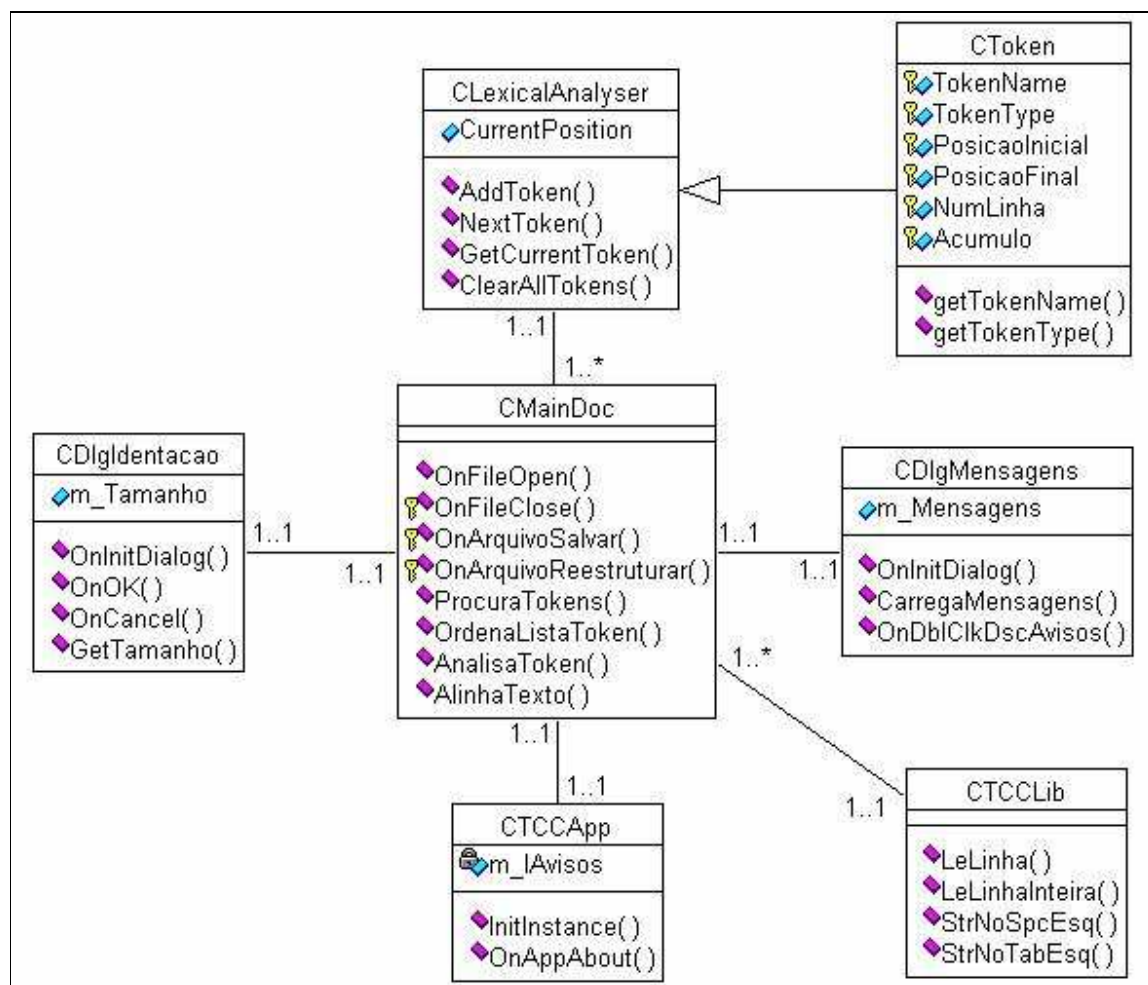


Uma breve descrição dos Casos de Uso pode ser vista a seguir:

- a) Selecionar Arquivo - o usuário seleciona o arquivo de código fonte em C++ para fazer a análise;
- b) Efetuar Reestruturação – o usuário inicia o processo de reestruturação do código fonte em C++, informando o tamanho da indentação do documento;
- c) Análise dos Resultados – ou usuário analisa e efetua as devidas alterações sugeridas pelas mensagens geradas pelo software.

Na figura 3 segue o diagrama de Classes:

FIGURA 3 – DIAGRAMA DE CLASSES



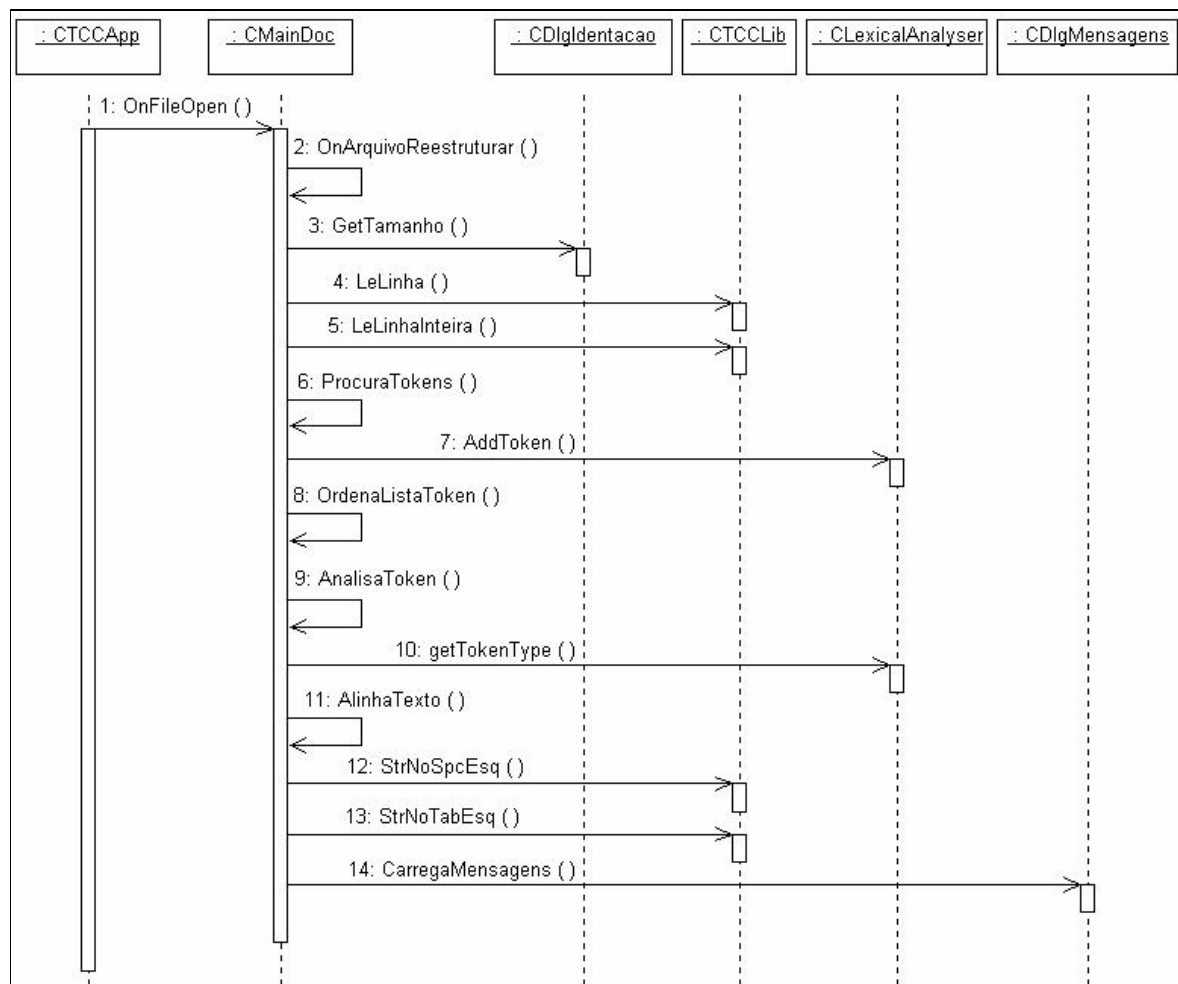
A seguir é apresentada na tabela a descrição das funções das classes do software:

Classe	Função	Descrição
CMainDoc	OnFileOpen	Abre o arquivo de código fonte em C++
CMainDoc	OnFileClose	Fecha o arquivo de código fonte em C++ aberto
CMainDoc	OnArquivoSalvar	Salva o arquivo reestruturado
CMainDoc	OnArquivoReestruturar	Efetua a reestruturação do arquivo de código fonte em C++ aberto
CMainDoc	ProcuraTokens	Procura por tokens no arquivo de código fonte aberto e armazena em uma lista
CMainDoc	OrdenaListaToken	Ordena a lista de tokens armazenados

CMainDoc	AnalisaToken	Faz a análise e alteração no código fonte em C++, conforme padrões estabelecidos
CMainDoc	AlinhaTexto	Faz o alinhamento do código fonte em C++
CLexicalAnalyser	AddToken	Adiciona um token na lista de tokens
CLexicalAnalyser	NextToken	Retorna o índice do próximo token
CLexicalAnalyser	GetCurrentToken	Retorna o índice do token corrente
CLexicalAnalyser	ClearAllTokens	Limpa a lista de tokens
CToken	getTokenName	Retorna o nome do token corrente
CToken	getTokenType	Retorna o tipo do token corrente
CTCCLib	LeLinha	Lê a linha sem comentários do arquivo aberto para a reestruturação
CTCCLib	LeLinhaInteira	Lê a linha inteira do arquivo aberto para a reestruturação
CTCCLib	StrNoSpcEsq	Tira os espaços a esquerda da linha lida
CTCCLib	StrNoTabEsq	Tira as tabulações a esquerda da linha lida
CDlgMensagens	OnInitDialog	Função de inicialização da classe
CDlgMensagens	CarregaMenagens	Joga as mensagens geradas para a tela
CDlgMensagens	OnDbkClkDscAvisos	Função que trata o clique duplo na tela de mensagens
CdlgIdentação	GetTamanho	Função que retorna o tamanho da indentação informado
CdlgIdentação	OnOk	Função que finaliza a classe retornando sucesso de execução
CdlgIdentação	OnCancel	Função que finaliza a classe retornando não sucesso de execução
CdlgIdentação	OnInitDialog	Função de inicialização da classe

Para facilitar o entendimento do Caso de Uso: Efetuar Reestruturação, foi elaborado o diagrama de sequência apresentado na figura 4:

FIGURA 4 – DIAGRAMA DE SEQUÊNCIA



4.4 ESTRUTURA DO SOFTWARE

O software foi implementado na linguagem de programação C++ utilizando a ferramenta Microsoft Visual C++ versão 6.0.

O software foi desenvolvido utilizando basicamente três classes, onde o funcionamento básico de cada uma delas são:

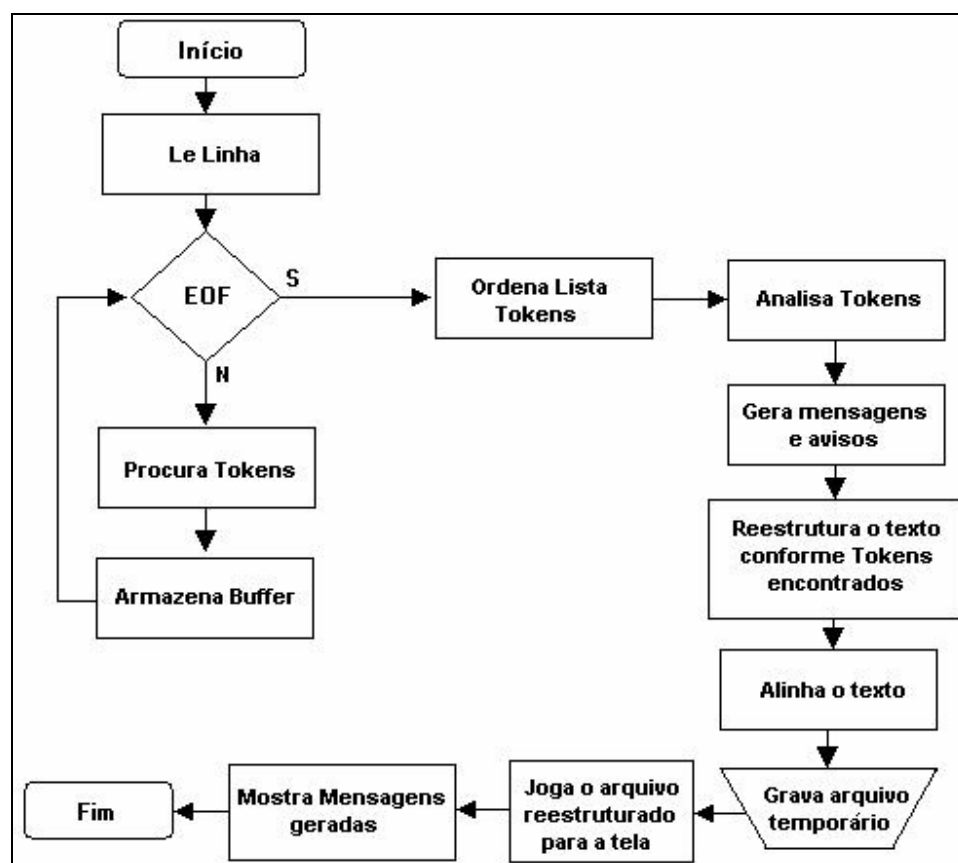
- a) classe CToken – classe da estrutura onde são armazenados os tokens encontrados no arquivo de código fonte em C++;

- b) classe CLexicalAnalyser – classe que gerencia e armazena os tokens em uma estrutura de lista;
- c) classe CMainDoc – classe que efetua todas as operações referentes ao arquivo que está sendo aberto e o arquivo gerado após a reestruturação, além das operações de procurar, ordenar e analisar o tokens coletados, e também alinhar o texto do código fonte informado;

As outras classes restantes são apenas classes auxiliares.

Na figura 5 segue o fluxograma do software.

FIGURA 5 - FLUXOGRAMA DO FUNCIONAMENTO DO SOFTWARE



O funcionamento do software atende os seguintes passos. Primeiro é aberto um arquivo de código fonte em C++ através do menu Arquivo – Abrir. Depois é feita a reestruturação do código através das opções do menu Efetuar – Reestruturação, onde é aberta

uma tela para informar o tamanho de alinhamento do texto do código fonte. Após informar o tamanho, aparecerá na tela o código fonte reestruturado proposto.

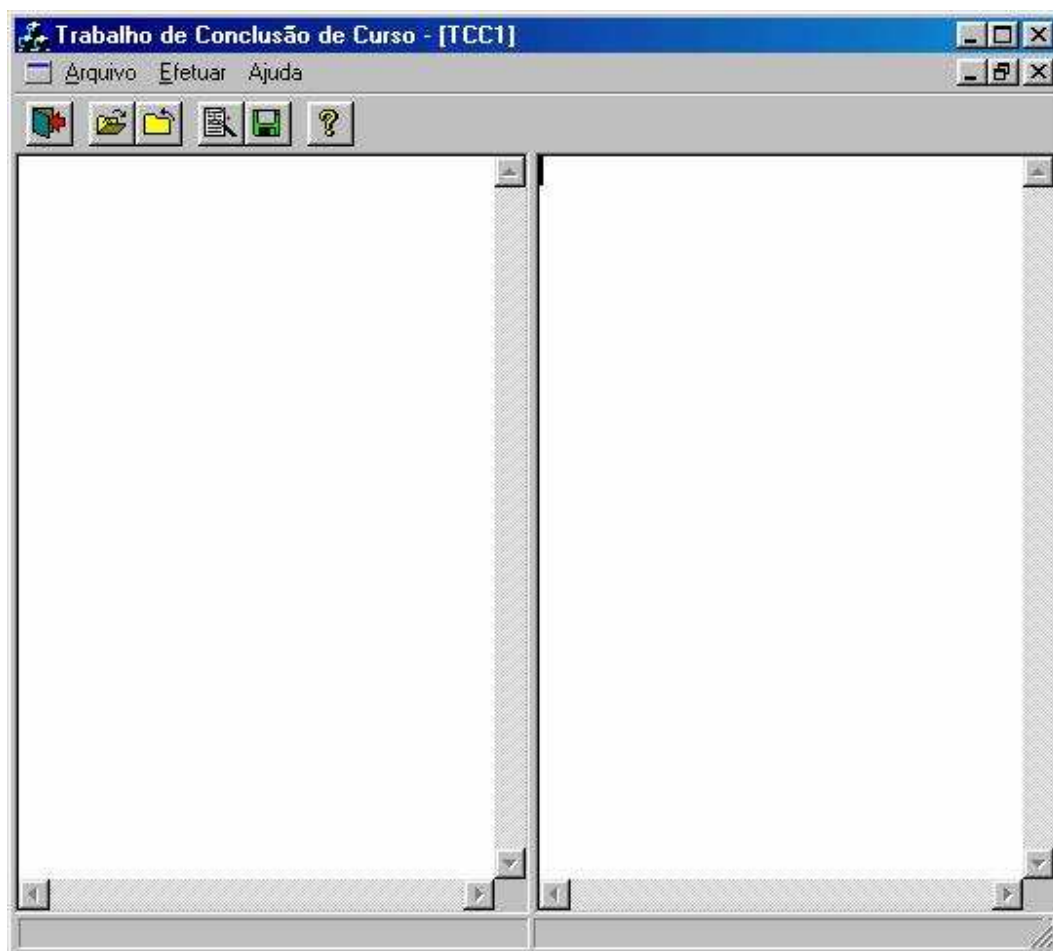
A função de reestruturação do arquivo fonte realiza as seguintes tarefas:

- a) leitura do arquivo origem – onde o arquivo é lido linha a linha e armazenado em um buffer;
- b) procura, separação e armazenamento dos tokens – onde o tokens são identificados e armazenados na estrutura de lista;
- c) ordenação dos tokens – após o armazenamento de todos os tokens, a lista com os tokens e ordenada pela posição do token no arquivo;
- d) analisa, gera mensagens e reestrutura o arquivo aberto – verifica os tokens, armazena as mensagens e avisos em uma lista e altera ou reestrutura o buffer do arquivo aberto conforme padrões e recomendações estabelecidos para os tokens;
- e) alinha o texto do arquivo aberto armazenado no buffer – alinha o texto conforme valor de indentação informado antes da reestruturação;
- f) grava o arquivo de destino – gera e grava o arquivo de saída;
- g) mostra o arquivo reestruturado e as mensagens geradas – coloca o arquivo reestruturado para a tela, e mostra as mensagens e avisos gerados pela reestruturação.

4.5 FUNCIONAMENTO DO SOFTWARE

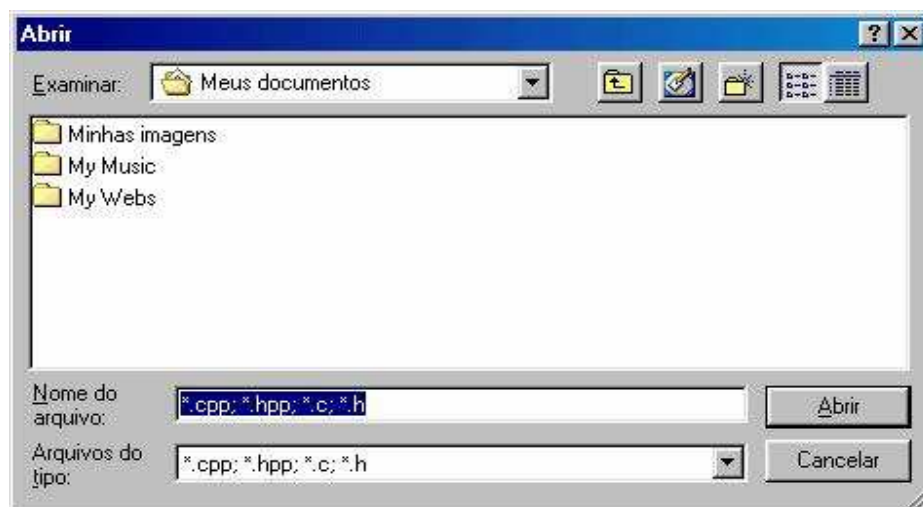
Ao iniciar o programa, será apresentado a tela principal do programa ao usuário, como demonstra a figura 6:

FIGURA 6 – TELA PRINCIPAL DO SOFTWARE



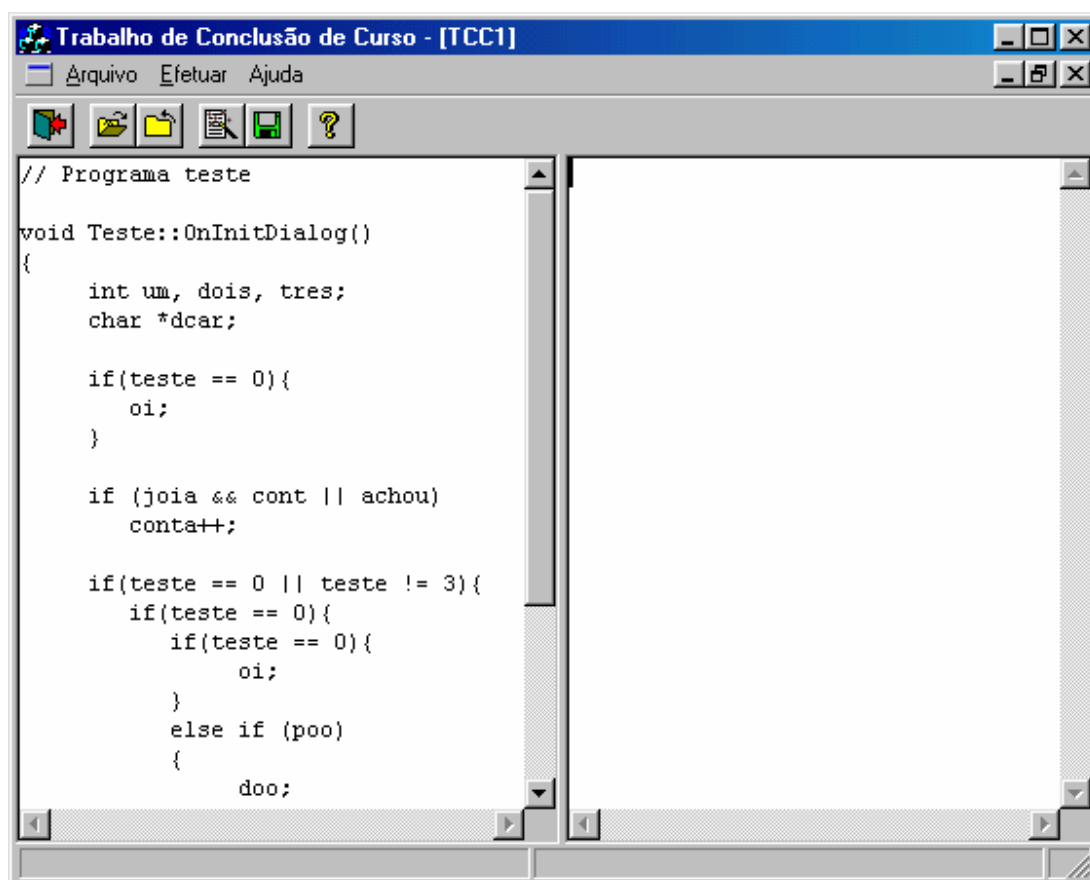
Para iniciar o usuário deverá abrir um arquivo que contenha um programa fonte em C++, para isso ele deverá selecionar a opção Arquivo e posteriormente Abrir, o que irá abrir uma janela conforme figura 7:

FIGURA 7 – TELA DE ABERTURA DE ARQUIVOS PARA REESTRUTURAÇÃO



Após selecionar e abrir um arquivo, ele aparecerá na tela principal, conforme figura 8:

FIGURA 8 – TELA PRINCIPAL COM UM ARQUIVO FONTE EM C++ ABERTO



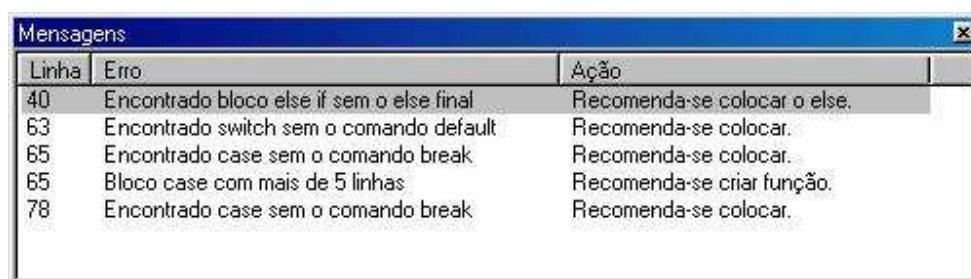
Após ter o arquivo aberto, o usuário poderá fazer a reestruturação do mesmo, clicando no menu em Efetuar e posteriormente Reestruturação, o que abrirá a uma tela pedindo para informar um tamanho de indentação do código fonte conforme Figura 9:

FIGURA 9 – TELA PARA INFORMAR O TAMANHO DE INDENTAÇÃO DO CÓDIGO FONTE



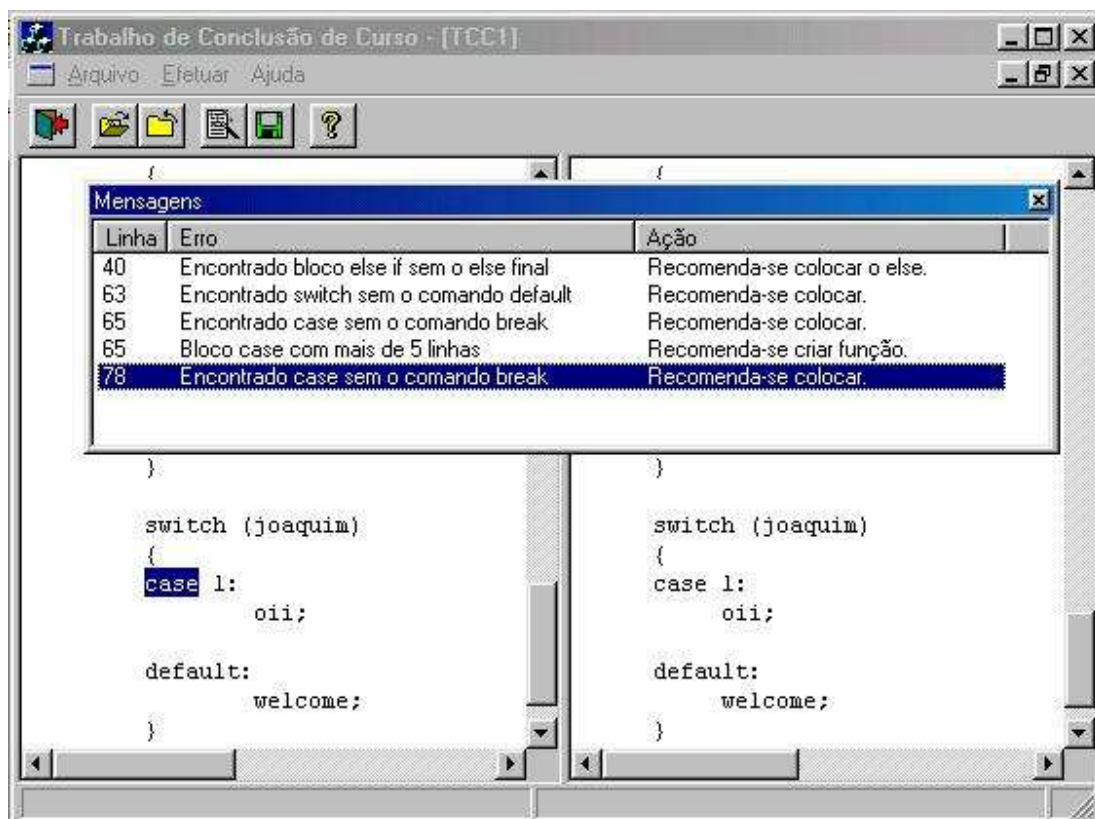
Após informar o tamanho da indentação e clicar em OK o programa irá reestruturar o código fonte aberto. Se no código fonte houver comandos não permitidos ou outra condição que gere mensagens, irá abrir a tela de mensagens. Clicando duas vezes sobre a mensagem, o software irá posicionar o cursor na linha em que ocorreu o erro ou advertência. A tela de mensagens segue conforme a figuras 10 e 11.

FIGURA 10 – TELA DE MENSAGENS SOBRE A REESTRUTURAÇÃO DO CÓDIGO FONTE

A imagem mostra uma janela de mensagens com o título "Mensagens". Ela contém uma tabela com três colunas: "Linha", "Erro" e "Ação".

Linha	Erro	Ação
40	Encontrado bloco else if sem o else final	Recomenda-se colocar o else.
63	Encontrado switch sem o comando default	Recomenda-se colocar.
65	Encontrado case sem o comando break	Recomenda-se colocar.
65	Bloco case com mais de 5 linhas	Recomenda-se criar função.
78	Encontrado case sem o comando break	Recomenda-se colocar.

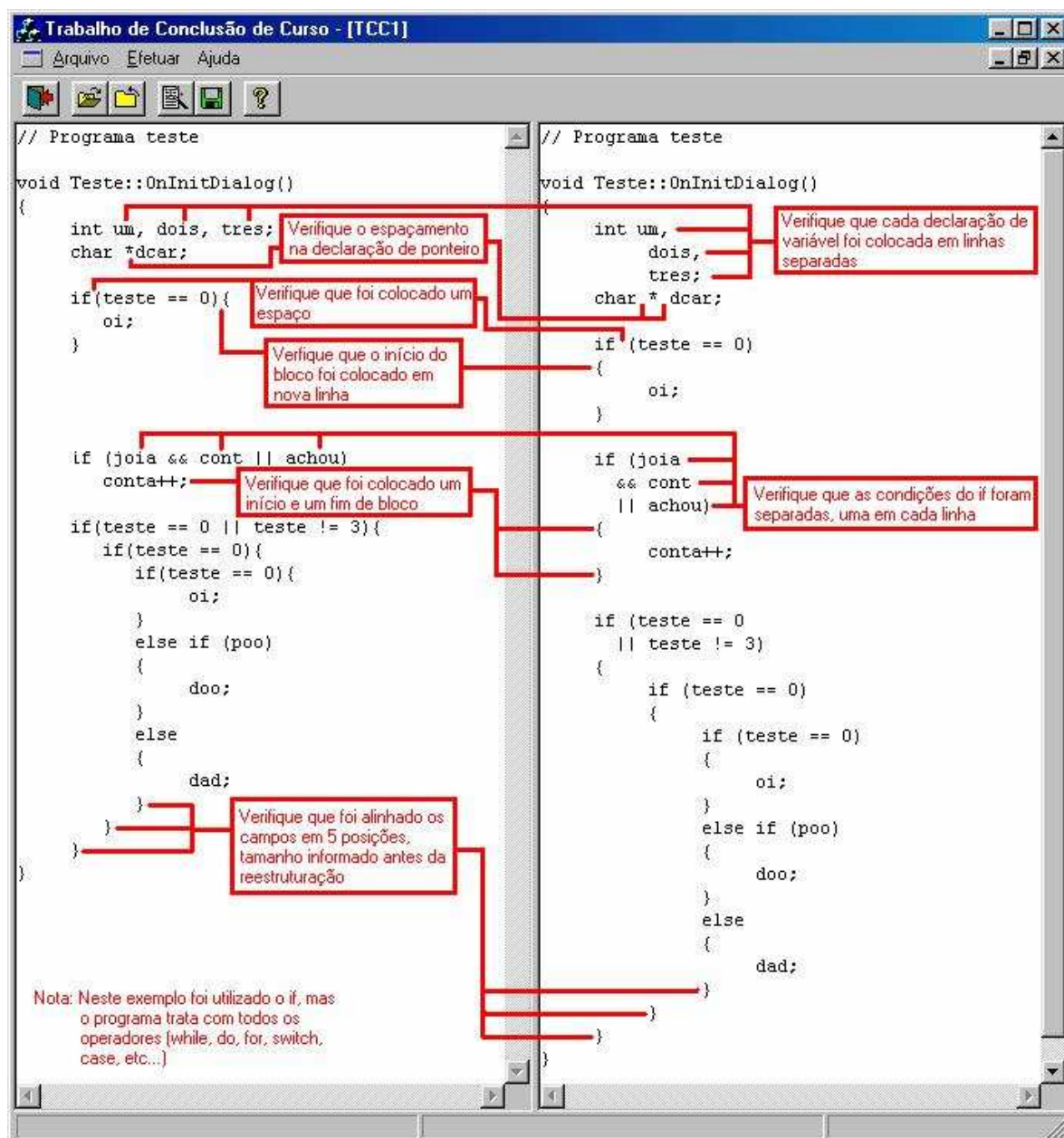
FIGURA 11 – TELA DE MENSAGENS POSICIONANDO NA LINHA COM O PROBLEMA



A reestruturação apresentará um código fonte reestruturado na janela ao lado do original aberto, com todas as alterações e ajustes por ele estabelecidos.

Veja que a figura 12 mostra as principais reestruturações efetuadas pelo software, mostrando o código original, o código reestruturado e comentários sobre a mudança .

FIGURA 12 – TELA DO SOFTWARE COM EXEMPLO DE CÓDIGO FONTE ANTES E DEPOIS DA REESTRUTURAÇÃO



5 CONCLUSÃO

Este capítulo apresenta as conclusões do trabalho. As principais limitações e sugestões para futuros trabalhos também são citadas neste capítulo.

5.1 CONSIDERAÇÕES FINAIS

Os padrões utilizados para efetuar a reestruturação são de grande valia, pois tornam o código fonte mais legível, aumentando a qualidade do software e conseqüentemente diminuindo os futuros custos com a manutenção dos mesmos.

O padrão geral e o padrão de estilo são importantes e por isso foram explorados na implementação do software, gerando alterações no código fonte em linguagem C++ e gerando mensagens de avisos, de recomendações e de erro no software implementado.

O padrão de composição de módulos e o padrão para escolha de nomes não geraram nenhuma implementação no software. Pois as regras do padrão de composição de módulos tratam de como os arquivos de código-fonte em C++ devem ser nomeado e o que eles devem conter, e as regras do padrão para escolha de nomes tratam sobre o formato para a elaboração de nomes. Como as duas regras possuem muitas variáveis, podendo existir inúmeros estilos para uma mesma regra, a empresa ou o programador deve adotar um estilo que mais lhe agrada, o que dificulta a análise e reestruturação dos mesmos.

No padrão geral foram utilizadas as regras e recomendações para o tratamento de expressões e de comandos, com o intuito de reduzir a frequência de erros e defeitos mais corriqueiros em programação.

O padrão de estilo foram utilizadas as regras e recomendações sobre margem esquerda, estilo de declarações, de expressões, de atribuições, de blocos e de estruturas de controle, com o intuito de uniformizar o estilo de programação, facilitando a compreensão, manuseio e modificação de programas escritos por outros.

Uma das constatações feitas durante o trabalho é que a utilização de tokens para a construção de um analisador léxico nesta ferramenta tornou o programa mais veloz. O código fonte fica mais legível e conseqüentemente de fácil manutenção.

Os objetivos do trabalho foram alcançados visto que a ferramenta permite analisar e reestruturar código-fonte de programas feitos na linguagem C++, a partir de padrões de legibilidade estudados na bibliografia, como o padrão geral e o padrão de estilo que foram melhor atendidos pela ferramenta desenvolvida.

5.2 LIMITAÇÕES E SUGESTÕES

Uma das limitações encontradas foi que o padrão para escolha de nomes e o padrão de composição de módulos possuem regras e/ou recomendações que não puderam ser implementadas, devido à complexidade das estruturas que as envolviam, ou mesmo a grande diversidade de definições existentes para o comando. Como exemplo os padrões para escolha de nome, que tratam de uma estrutura para o nome, que contém componente ou módulo, prefixo do tipo, tema e sufixo do tipo, e que estes componentes ainda são opcionais, dificultando ainda mais a verificação e validação do código. E os padrões para composição de módulos que estabelecem regras para o descrição da extensão de arquivos que contém código-fonte em C++, e também de uma seqüência de declarações e definições dentro de um arquivo de código-fonte em C++ que devem ser seguidos.

Uma sugestão para futuros trabalhos seria a implementação de novas regras, padrões e recomendações para enriquecer a padronização e reestruturação de código fonte em C++. Ou ainda fazer a análise e reestruturação de código fonte de outras linguagens bastante utilizadas no mercado.

Outra sugestão seria tornar este software mais flexível, permitindo habilitar e desabilitar determinados padrões de acordo com a necessidade do usuário

ANEXOS

ANEXO 1 – PADRÃO DE COMPOSIÇÃO DE MÓDULOS

ARQUIVOS EM GERAL

- 1) Utilize os seguintes nomes de extensão de arquivos:
 - a) C para módulos de implementação escritos em C.
 - b) H para módulos de definição escritos em C.
 - c) CPP para módulos de implementação escritos em C++.
 - d) HPP para módulos de definição escritos em C++.
- 2) Cada módulo será composto pelo menos por dois arquivos, um contendo o módulo de definição e outro contendo o correspondente módulo de implementação.

Módulos definição de arquivos devem conter controles para evitar a inclusão duplicada ao compilar um módulo. Deve ser utilizado o esquema de código a seguir:

Comentário cabeçalho do arquivo

```
#if !defined( Nome-arquivo_MOD)
#define Nome-arquivo_MOD
    Corpo do arquivo
#endif
```

Comentário fim de arquivo

Comentário cabeçalho do arquivo – é o comentário padrão identificando o arquivo. O formato deste comentário deve ser especificado pela organização.

Nome-arquivo – é o nome do arquivo (sem a extensão) contendo o módulo de definição ou a tabela de constantes. O nome do arquivo será seguido da seqüência de caracteres `_MOD`, assegurando a unicidade do nome. Este nome será definido quando da primeira inclusão e nunca será excluído.

Corpo do arquivo – é o código do módulo de definição ou do arquivo de definição.

Comentário fim do arquivo – é o comentário padrão de final de arquivo especificado pela organização.

MÓDULOS DE DEFINIÇÃO

Inicie o código do corpo do módulo de definição com o seguinte esquema de código:

```
//Controle de escopo do arquivo de definição
#if defined (Nome-arquivo_OWN)
    #define Nome-arquivo_EXT
#else
    #define Nome-arquivo_EXT extern
#endif
```

Declare cada variável global externa da seguinte forma:

```
Nome-arquivo_EXT variável global externa
```

Declare a inicialização de variáveis externas, utilizando código com a organização a seguir. Atente para o fato que o código a seguir inclui o caractere ";" do final da declaração.

```
Declaração-da-variável
#if defined (Nome-arquivo_EXT)
    = inicialização;
#else
    ;
#endif
```

1) Ao final do código do módulo de definição coloque o código:

```
#undef Nome-arquivo_EXT
```

2) Organize os módulos de implementação e de definição da seguinte forma:

- a) Cabeçalho do módulo;
- b) Se necessário, inclua o arquivo contendo as declarações não padronizadas segundo o padrão ANSI requeridas pelo ambiente de desenvolvimento. Este arquivo deve viabilizar a compilação do programa utilizando a chave de controle de código ANSI ligada;
- c) Inclusões requeridas pelas bibliotecas do compilador;
- d) Inclusão do módulo de definição próprio;
- e) Inclusões dos módulos de definição dos módulos servidores requeridos;

- f) Inclusões dos arquivos de definição requeridas;
 - g) Corpo do modulo de implementação ou de definição;
 - h) Comentário de final do módulo.
- 3) Inclua somente os arquivos que forem efetivamente necessários.
 - 4) Módulos de definição devem conter exclusivamente declarações.
 - 5) Módulos de definição devem conter também o código executável de funções *inline* e de macros, mas somente se necessários para que módulos cliente possam ser compilados. Funções *inline* e macros cujo escopo se restrinja ao modulo sendo compilado, devem estar no respectivo modulo de implementação.

Exemplo de módulo de definição

```

/*****
/* Modulo de definição: Modulo exemplo
/* Nome do arquivo:      EXEMP.H
*****/
#if !defined (EXEMP_MOD)
#define EXEMP_MOD

// Controle de escopo do arquivo de definição
#if defined (EXEMP_OWN)
    #define EXEMP_EXT
#else
    #define EXEMP_EXT extern
#endif

/***** Tipo de dados exportado pelo módulo *****/
typedef struct
{
    int UmInt;
    int OutroInt;
} EX_tpMeuTipo; /* note que a declaração de tipos não
                e afetada pelas regras estabelecidas */

/***** Estruturas de dados exportada pelo modulo *****/
/* Estrutura de dados : Vetor inicializado */

EXEMP_EXT int EX_vtNum[ ]
    #if defined (EXEMP_OWN)
        = { 1, 2, 3, 4, 5 };
    #else
        ;
    #endif

#undef EXEMP_EXT
#endif
/***** Fim da definição: modulo *****/

```

MÓDULO DE IMPLEMENTAÇÃO

- 1) No módulo de implementação redija o código de inclusão do respectivo modulo de definição na forma a seguir:

```
#define Nome-Arquivo_OWN
#include "Nome-Arquivo.H"
#undef Nome-Arquivo_OWN
```

- 2) Todas as variáveis globais e funções encapsuladas devem ser declaradas nos respectivos módulos de implementação e devem estar precedidas do declarador *static*.

Exemplo de módulo de implementação

```

/*****
/* Módulo de implementação: Módulo exemplo
/* Nome do arquivo: EXEMP.C
*****/

/* Inclusões do compilador */
#include <stdio.h>

/* Inclusão do respectivo modulo de definição */
#define EXEMP_OWN
#include "EXEMP.H"
#undef EXEMP_OWN

/*Inclusão de módulos de definição de servidores e tabelas de definição */
#include "Modulo1.H"
#include "Modulo2.H"
#include "Tabela.INC"

/* Código do modulo de implementação */
...
/***** Fim da implementação: módulo *****/

```


ANEXO 2 – PADRÃO GERAL

USO DA LINGUAGEM PADRÃO

- 1) Ponha legibilidade e compreensibilidade antes das demonstrações de domínio de nuances da linguagem.
- 2) Somente otimize código por meio de artifícios de programação se for estritamente necessário e, ainda assim, somente depois de ter medido o desempenho dos elementos do programa
- 3) Ao programar em C utilize somente construções válidas no padrão ISO [C1990].
- 4) Ao programar em C++ utilize somente construções válidas no padrão ISSO [C++ 1998, Stroustrup 1997].
- 5) Todas as palavras não padronizadas devem ser agregadas em um único arquivo de definições.
- 6) Utilize a chave de controle de mensagens de advertência mais restritiva disponível no compilador e corrija o código até que não sejam mais geradas advertências ao compilar.
- 7) Caso a advertência seja decorrente de um uso justificado de uma construção problemática, ou decorrente de otimização aceitável, ou gerada por falha conhecida do compilador, inclua um comentário no código fonte, no local indicado pela mensagem de advertência, informando que a mensagem está justificada e a razão da justificativa.
- 8) Ao desenvolver um programa utilizando um Ambiente Integrado de Programação (IDE – Integrated Development Environment, por exemplo, Turbo C/C++, Visual C/C++. Visual Café), realize o conjunto completo de testes com relação ao programa executável gerado no final. Os testes de aceitação devem ser realizados fora do controle do ambiente de desenvolvimento.
- 9) Mantenha a última listagem de mensagens de erro e advertências junto com os respectivos módulos de implementação e de definição.

DECLARAÇÃO DE PROTÓTIPOS E DE CABEÇALHOS DE FUNÇÕES

- 1) Declare a lista de parâmetros de um protótipo de função contendo os mesmos nomes que os contidos na lista de parâmetros do correspondente cabeçalho.
- 2) Ao programar em C use *void* como tipo de função que retorne nada, e quando a lista de parâmetros formais for vazia.
- 3) Ao programar em C++ utilize *void* para denotar um valor retornado de tipo indefinido.

DECLARAÇÕES

- 1) Utilize blocos aninhados para declarar variáveis locais de modo que tenham o menor escopo possível.
- 2) Evite obliterar com nomes de variáveis locais os nomes de elementos globais ou de membros de classes.
- 3) Sempre que for possível, declare e inicialize as variáveis em um mesmo comando.
- 4) Não é necessário inicializar variáveis globais se o valor inicial for 0.
- 5) Caso uma variável deva receber um valor calculado em uma função antes de ser utilizada, inicialize-a com um valor constante não permitido no conjunto de valores legais e que denote *variável não inicializada*.

Exemplo

```
#define VALOR_ILEGAL  0xFFFF
...
tpAlgumTipo AlgumaArea = VALOR_LEGAL;
```

- 6) Inclua imediatamente antes do primeiro uso de uma variável inicializada com a constante valor ilegal um dos fragmentos de código:

```

ASSERT Variavel != VALOR_ILEGAL;

ou

if ( Variavel == VALOR_ILEGAL )
{
    ativar exceção de erro de uso da variável
} // if

```

- 6) Use tipos *unsigned* somente para declarar variáveis que jamais poderão ter valores negativos, mesmo durante a avaliação de uma expressão envolvendo valores deste tipo.

Exemplo

```

// Evite:
unsigned Conta;
for ( Conta = 10; Conta >= 0; Conta-- ) //nunca termina

// Redija assim:
int Conta;
for ( Conta = 10; Conta >= 0; Conta-- )

```

DADOS GLOBAIS

- 1) Evite o uso de dados globais externos.
- 2) Sendo necessário declarar dados globais, agregue-os em um tipo estrutura e declare uma única variável com este tipo.

Exemplo

```

/* Declaração do tipo estrutura de dados da interface */
typedef struct
{
    int Var1;
    int Var2;
} MD_tpInterface;

/* Declaração da interface */
MD_tpInterface MD; /* O acesso aos elementos terá a
                    forma MD.Var1 */

/* Declaração do tipo estrutura de dados encapsulada
typedef struct
{
    int Var1;
    int Var2;
} st_tpCapsula;

```

```

/* Declaração da cápsula */
st_tpCapsula st; /* O acesso aos elementos terá a forma
                  st.Var1 */

```

SEQÜÊNCIA DE EXECUÇÃO

- 1) Agregue e ordene seqüências de comandos ou pseudo-instruções de acordo com o seu significado.

Exemplo

```

// Seqüência a evitar
Gravar BufferA em Saída
Gravar BufferB em Saída
Ler registro de ArquivoB para BufferB
Ler registro de ArquivoA para BufferA

// Prefira a seguinte seqüência
// Transferir A para Saída
Gravar BufferA em Saída
Ler registro de ArquivoA para BufferA
// Transferir B para Saída
Gravar BufferB em Saída
Ler registro de ArquivoB para BufferB

```

A possibilidade de identificar novas pseudo-instruções é particularmente interessante pois facilita encontrar agregados que possam ser reutilizados. No exemplo, acima, pode-se criar uma função que grava o *buffer* e lê um novo valor. Após esta transformação, o texto fica assim:

```

// Transferir A e B para Saída
Tranferir (ArquivoA, BufferA, ArquivoSaida);
Tranferir (ArquivoB, BufferB, ArquivoSaida);

// A função Transferir seria semelhante a:
void Transferir ( FILE * ArquivoEntra,
                 char * BufferEntra,
                 FILE * ArquivoSai      )
{
    Gravar BufferEntra em ArquivoSai
    Ler registro de ArquivoEntra para BufferEntra
}

```

- 2) Sequências de comandos que são pares de operações onde uma desfaz o efeito da outra, por exemplo abrir e fechar arquivos, alocar e desalocar espaço de dados, construir e destruir classes, devem ser redigidos obedecendo ao correspondente aninhamento.

Exemplo

```
// Ordenação de código a evitar
Abrir arquivo A
Abrir arquivo B
Abrir arquivo C
...
Fechar arquivo A
Fechar arquivo B
Fechar arquivo C

// Ordenação de código segundo a regra
Abrir arquivo A
Abrir arquivo B
Abrir arquivo C
...
Fechar arquivo C
Fechar arquivo B
Fechar arquivo A
```

EXPRESSÕES

- 1) Evite o uso do operador ternário “?” quando pelo menos uma das expressões contiver mais de um operador, ao invés utilize o comando *if*.
- 2) Sempre use *sizeof(NomeVariavel)* para determinar o tamanho ocupado por uma variável.
- 3) Sempre use *sizeof(NomeDoTipo)* para determinar o tamanho a ser ocupado por um elemento a alocar (ex.: *malloc*), ou para saber o tamanho do espaço de dados apontado por um ponteiro ou referência.
- 4) Evite o uso de imposição de tipos (*type cast*). Use-o somente para determinar o tipo específico ao alocar um espaço de dados, ou ao copiar um valor contido em estrutura de persistência (por exemplo, arquivo) ou em estrutura genérica (por exemplo, lista).
- 5) Sempre verifique o retorno de uma função ou método que possa gerar uma exceção ou retornar uma condição de retorno.

Exemplo

```
//Evite:
pStr = (char *) malloc(strlen (Str)+ 1);
strcpy(pStr, Str);
pArq = fopen(NomeArq, "w");
fputs(pStr, pArq);
```

```

//Redija:
pStr = (char *) malloc(strlen (Str)+ 1);
if (pStr == NULL)
{
    printf("\nFaltou memória");
    exit(4);          /* ou tratamento de erro */
} /* if */
strcpy(pStr, Str);

pArq = fopen(NomeArq, "w");
if (pArq == NULL)
{
    printf("\nNão abriu: %", NomeArq);
    exit(4);          /* ou tratamento de erro */
} /* if */
fputs(pStr, pArq);

// Evite:
CFile Arq(NomeArq, CFile::modeCreate | CFile::modeWrite));
Arq.Write(Buffer, sizeof(Buffer));

// Redija:
CFile Arq;
if (!Arq.Open(NomeArq, CFile::modeCreate | CFile::modeWrite))
{
    out < "\nNão abriu " < NomeArq < " em virtude de "
    < expArq.m_cause;    tratamento de erro
} // if
Arq.Write(Buffer, sizeof(Buffer));

```

COMANDOS

- 1) É proibido o uso de `goto`.

switch e case

- 2) Mantenha curto o código associado a cada `case` (em torno de 5 linhas). Se o código ficar longo, converta-o em uma chamada de função.
- 3) Sempre termine o bloco de comandos que segue um `case` com um comando `break`.

Exemplo

```

// Evite, pois torna o código sensível à arrumação
case 1:
    algum código sem break no final
case 2:
    mais código sem break no final

//Redija:case 1:
    algum código

```

```

        break;
    case 2:
        mais código
        break;

// Outro exemplo, redija assim
case 'd':
case 't':
    printf("Consoantes palatais");
    break;

```

4) Sempre inclua uma opção default nas estruturas de switch.

5) O default do switch deve capturar somente as condições não previstas.

Exemplo:

```

//Evite: switch(menuItem)
{
    case ITEM_COPY:
        Copiar();
        break;
    case ITEM_CUT:
        Cortar();
        break;
    default:           // Captura tudo: colar e todos os erros.
        Colar();
        break;
} // switch

// Redija assim:
switch(menuItem)
{
    case ITEM_COPY:
        Copiar();
        break;
    case ITEM_CUT:
        Cortar();
        break;
    case ITEM_PASTE:
        Colar();
        break;
    default:
        ASSERT(FALSE);           // Sempre corresponde a um erro
        break;
} // switch

// Exemplo de default que efetivamente corresponde a "demais casos"
switch(NivelAcesso)
{
    case 0:
        InserirAdministrador();
        break;
    case 1:
        InserirConsultor();
        break;
    case 4:

```

```

        InserirInstalador();
        break;
    default:           // compreende 2, 3, 5, 6, ..., 100
        InserirUsuarioNormal();
        break;
    } // switch

```

if e else

- 1) Em seleções múltiplas heterogêneas criadas com sucessivos comandos `if ... else if`, assegure que o último `else` exista.
- 2) O `else` final de uma seleção heterogênea deve capturar somente as condições não previstas.
- 3) Assegure que o conjunto das condições de uma seleção heterogênea, exceto o `else` final, cobre a totalidade de condições possíveis.

Exemplo

```

if (cond1)
{
    // fragmento de código selecionado se cond1 for verdadeira
} else if (cond2)
{
    // fragmento de código selecionado se cond1 for falsa
    // e cond2 for verdadeira
} else if (cond3)
{
    ...
} else if (condn)
{
    // fragmento de código selecionado se cond1 até condn-1 forem
    // todas falsas e condn for verdadeira
} else
{
    // fragmento de código selecionado se cond1 até condn forem
    // todas falsas, condição default
} // if

```

Repetições

- 1) Não crie variáveis temporárias apenas para controle de término de um ciclo, use `break` ou `return` para sair de repetições antes de processar todos os elementos.
- 2) Evite o uso de `continue`.
- 3) Antes de ativar o corpo da repetição, o estado corrente deve estar completamente definido.

- 4) Ao retornar do corpo para o controle de repetição, o estado corrente deve corresponder ao próximo estado a ser processado.
- 5) Durante a execução da repetição cada iteração deve corresponder a um estado e deve ser diferente dos demais.
- 6) O número de estados de uma repetição deve ser finito.

Assegurar o número de estados de uma repetição seja finito pode requerer truques de programação. Exemplo: caminhamento em grafo.

```
void VisitarFilhos( tpQQ * pNo )
{
    tpQQ* pNoCorr;
    if ( pNo == NULL )
    {
        return;
    } /* if */
    pNoCorr = pNo->pOrgFil;
    while ( pNoCorr ) // entra em loop se existirem ciclos
                    // no grafo
    {
        VisitarFilhos( pNoCorr );
        pNoCorr = pNoCorr->pProxIrmao;
    } /* while */
} /* VisitarFilhos */
```

Código modificado

```
void VisitarFilhos( tpQQ * pNo )
{
    tpQQ* pNoCorr;
    if ( pNo == NULL )
    {
        return;
    } /* if */
    if ( FoiVisitado( pNo ))
    {
        return;
    } /* if */
    MarcarVisitado( pNo );
    pNoCorr = pNo->pOrgFil;
    while ( pNoCorr )
    {
        VisitarFilhos( pNoCorr );
        pNoCorr = pNoCorr->pProxIrmao;
    } /* while */
} /* VisitarFilhos */
```

ANEXO 3 – PADRÃO DE ESTILO

MARGEM ESQUERDA

- 1) Selecione um espaço padrão de indentação. Recomenda-se um valor entre 3 e 5.
- 2) Evidencie o aninhamento da estrutura de código (aninhamento de pseudo-instruções e de estruturas de controle) por meio de indentação. Ao indentar para a direita, avance sempre 3 caracteres. Ao destacar uma linha de comentário, recue sempre 3 caracteres para a esquerda.
- 3) Utilize caracteres *espaço em branco* e não tabulações para alinhar a margem esquerda.
- 4) Comentários relativos a uma ou mais linhas de código devem estar indentados 3 caracteres para a *esquerda* com relação à primeira destas linhas.
- 5) Separe por uma linha em branco a primeira linha de um bloco de comentários da última linha do bloco de código que a antecede.
- 6) Estabeleça um limite para a margem direita e assegure que nenhuma linha de código ultrapasse este limite.
- 7) Linhas de continuação devem estar 10 caracteres para a direita da linha inicial do comando continuado.
- 8) Linhas de continuação utilizadas em expressões lógicas e em listas de parâmetros tem formatação própria.
- 9) Parâmetros de função redigidos em linhas sucessivas, devem alinhar-se na margem esquerda do primeiro parâmetro da função.
- 10) Quando o espaço ficar excessivamente pequeno, recomece a indentação a partir da margem 20.
- 11) Procure sempre iniciar a linha de continuação com um operador ou com um compositor de nome.

As linhas de continuação precisam ser claramente entendidas como tal, portanto a indentação deve ser significativamente diferente da indentação de estrutura.

Neste exemplo observe também o posicionamento dos comentários e dos parâmetros de funções.

```

/* Abrir arquivos */
if (nArg == 5)
{
/* Abrir arquivo de entrada A */
pArqA = fopen(vtArg[1], "rb");
if (pArqA == NULL)
{
    ErroDados = 1;
    printf("\nNÃO abriu arquivo de entrada A: %s", vtArg[1]);
} /* if */

/* Abrir arquivo de entrada B */
pArqB = fopen(vtArg[2], "rb");

```

Exemplos de linha longa

```

VariavelMuitoLonga = OutraVariavelLonga->CampoLongo[
    IndiceLongo].OutroCampoLongo;

Tam = (short int)(MV_DIM_PAG - ((GL_tpNoLista *) GL.pNo)
    ->vetElem[ ((GL_tpNoLista *) GL.pNo)
        ->Cabeca.NumChaves - 1].OffVal);

```

No primeiro exemplo a linha de continuação começa com um operando uma vez que o abre colchete é um qualificador do nome que o antecede. Abre colchetes e abre parênteses devem sempre ficar juntos com o nome que qualificam. No segundo exemplo a terceira linha é uma continuação de um elemento incompletamente declarado na segunda linha. A indentação adicional enfatiza isto.

Exemplo: Primeiro parâmetro na mesma linha que o nome da função

```

FuncaoLongaComTresParametros( EsteEhOPrimeiroParametro      ,
                               EsteEhOSegundoParametro      ,
                               EsteEhOTerceiroParametro
                               ->CampoDeNomeMuitoLongo[IndiceLongo]
                               .OutroCampoLongo              );

```

Observe o alinhamento das vírgulas, do fecha parênteses e a reorganização da margem esquerda do terceiro parâmetro.

Exemplo: Primeiro parâmetro em linha de continuação

```

FuncaoLongaComTresParametros(
    EsteEhOPrimeiroParametro      ,
    EsteEhOSegundoParametro      ,
    EsteEhOTerceiroParametro->CampoDeNomeMuitoLongo[
        IndiceLongo ].OutroCampoLongo      );

```

ESTILO DE DECLARAÇÕES

- 1) Procure dar um aspecto tabular ao código de declaração.
- 2) Declare cada variável em linha individual.
- 3) Declare cada variável ponteiro ou referência independentemente.
- 4) Alinhe o nome de variáveis declaradas em comandos sucessivos na mesma margem esquerda.
- 5) Alinhe inicializações de modo que os nome de variáveis declaradas permaneçam destacados no texto.
- 6) Redija o caractere "*" indicador de tipo ponteiro um espaço em branco após ao nome do tipo e mantenha-o separado do nome da variável declarada.

Tanto em C com em C++ a propriedade de ser um ponteiro ou uma referência é uma propriedade do tipo e não da variável.

Exemplos

```
// Não declare assim:
int i, j, k;

// Declare assim:
int i,
    j,
    k;

// Não declare assim:
int *pA,
    *pB,
    *pC;

// Declare assim:
int *pA;
int *pB;
int *pC;

// Não declare assim:
char *NomeUsuario = 0;
int NumLivros=42;
int & IntRef = NumLivros;

// Declare assim - observe o alinhamento dos nomes e do
// operador "=":
char * NomeUsuario = 0;
int    NumLivros    = 42;
int & IntRef        = NumLivros;
```

- 7) Declare na mesma linha o tipo retornado e o nome da função.

- 8) Declare parâmetros formais de uma função em linhas sucessivas, alinhadas com o primeiro parâmetro.
- 9) Alinhe as vírgulas separadoras de parâmetros formais na mesma coluna, e coloque o fecho parênteses da função um caractere para a direita da coluna das vírgulas.

Exemplos

```
// Não declare funções assim:
char*
  Cobj::sString( );
  Outro( int Parm, char* Str);
  MaisUm( int );
  main( );

// Declare assim:
char* Objeto::sString( void );
void  Outro( int  Parm,
           char* Str  );
void  MaisUm( int Nome );
void  main( void );
```

ESTILO DE EXPRESSÕES E ATRIBUIÇÕES

- 1) Separe todos os operandos, operadores e sinais de pontuação do elemento precedente por pelo menos um espaço em branco. No entanto, não separe:
 - a) Os operadores ‘.’ e ‘->’ dos seus operandos antecessores e sucessores.
 - b) Vírgulas dos elementos que as antecedem.
 - c) Operadores unários dos respectivos operandos que os sucedem.
 - d) Abre parênteses do nome de funções que os antecedem.
 - e) Abre colchetes do nome do vetor.
- 2) Separe as palavras reservadas de estruturas de controle (por exemplo, `if`, `for`, `while`, `switch`) do abre parênteses da correspondente expressão por pelo menos um espaço em branco.
- 3) Os caracteres abre parênteses “(“ início de lista de parâmetros, e abre colchetes “[“ início de expressão de indexação, devem ficar sempre na mesma linha que o correspondente nome de função ou de vetor.
- 4) Procure dar um aspecto tabular ao texto do código, alinhando operadores de atribuição em uma mesma coluna.
- 5) Sempre que possível agrupe comandos de atribuição sucessivos segundo o seu significado.

- 6) Redija atribuições de um mesmo valor a diversas variáveis em forma de uma lista de linhas, cada qual contendo uma atribuição.

Procure sempre destacar os elementos de uma expressão de modo a tornar mais legível o programa. Um forte aliado para o aumento da legibilidade é a diagramação do texto de código.

Exemplos

```

Var0 = Var1 + Var2 * (( Var3 + Var4 ) * 5 );
Funcao( -Vet[ Var6->Var7.Campo, &OutroParm ] );
Valor = OutraFunc( Parm, OutroParm + 1,
                  scanf( " %s", TerceiroParm ) );

// Evite
SegSlv = MV.IdSeg;
ReprSlv = ET_Parm.ReprCorr;
memcpy(( char * ) &GLSlv, ( char * ) &GL, sizeof( GLSlv ));
MV.IdSeg = BS.IdSegBS;
ListaExtrac = PAG_NIL;
TipoSel = SELEC_NIL;
ElemSel = ElemInic;
FrmSel = SELEC_NIL;
Navegou = 0;
DE.IdF = 0;
DE.ModaNavega = 0;

// Redija assim, código diagramado como uma tabela:
memcpy(( char * ) &GLSlv, ( char * ) &GL, sizeof( GLSlv ));
SegSlv      = MV.IdSeg;
ReprSlv     = ET_Parm.ReprCorr;
MV.IdSeg    = BS.IdSegBS;
ListaExtrac = PAG_NIL;
TipoSel     = SELEC_NIL;
ElemSel     = ElemInic;
FrmSel      = SELEC_NIL;
Navegou    = 0;
DE.IdF      = 0;
DE.ModaNavega = 0;

// Evite
Var1 = 0;
Var3 = 'x';
vtVar2[ Var1 ].Campo2 = 1;

// Redija assim, texto reorganizado e diagramado.
// A variável Var3 é independente.
Var1      = 0;
vtVar2[ Var1 ].Campo2 = 1;
Var3     = 'x';

// Evite
Var1 = Var2 = Var3 = 0;

```

```
// Redija assim, cada atribuição em uma linha.
Var1 =
Var2 =
Var3 = 0;

// Evite:
if( A != 0 )
for( i = 0; i < n; i++ )

// Redija assim (if, while, etc. não são funções!):
if ( A != 0 )
for ( i = 0; i < n; i++ )
```

- 7) Expressões lógicas envolvendo operadores lógicos || e && devem ser quebradas em linhas sucessivas a cada operador lógico. As linhas sucessivas iniciam com o operador lógico.
- 8) Evite expressões lógicas complexas, particione-as em vários ifs aninhados.
- 9) A indentação deve assegurar que os abres parênteses de cada subexpressão estejam na mesma coluna conforme o nível de aninhamento.
- 10) Cada subexpressão lógica deve estar contida entre parêntesis, exceto quando contiver um único termo. Os caracteres abre e fecha parêntesis de um mesmo nível de operação devem estar alinhados na mesma coluna.

Exemplo

```
if ( ( ( DeltaTam > 0 )
      && ( EspacoDisponivel < MV_DIM_PAG / 2 ) )
    || ( ( DeltaTam < 0 )
        && ( EspacoDisponivel >= 0 ) ) )
```

ESTILO DE BLOCOS E ESTRUTURAS DE CONTROLE

- 1) Cada comando deve iniciar em uma nova linha.
- 2) Expressões não devem conter atribuições.

Da mesma forma como cada variável deve ter um somente significado, cada instrução deve servir a um só propósito. Cabe salientar que expressões tais como `a++` contém uma atribuição implícita, portanto devem sempre estar sozinhas em uma linha de código.

Exemplos

```
// Evite
vtAbc[ i++ ] = 0;

// Redija
vtAbc[ i ] = 0;
i++;
```

```
// Evite
vtAbc[ ++i ] = 0;

// Redija
i++;
vtAbc[ i ] = 0;

// Evite
if ( i -= X )

// Redija
i -= X;
if ( i != 0 )
```

- 3) Os caracteres '{' e '}' delimitadores de um bloco devem ser colocados na mesma coluna, alinhados na margem esquerda do correspondente comando de controle.
Corolário: os dois delimitadores nunca estarão em uma mesma linha.
- 4) Quando o corpo de um método for vazio, redija '{}'. Observe o espaço entre os caracteres.
- 5) Todos os comandos de controle devem ser seguidos de um bloco, mesmo que o código deste bloco seja vazio ou tenha uma única linha.
- 6) Exceto no caso de `else`, todos os blocos vazios devem receber um comentário indicando que estão propositalmente vazios.
- 7) Seleções heterogêneas formadas por uma sucessão de `ifs` devem ser redigidos como uma lista formada por `else if` seguindo o esquema:

```
if ( cond1 )
{
    código1
} else if ( cond2 )
{
    código2
} else
{
    códigofinal
} // if
```

- 8) Evite um aninhamento de mais de três `ifs`.

Exemplo:

```
// Evite
while ( cond ) {
    código do bloco
} // while

// Redija
while ( cond )
{
    código do bloco
} // while
```



```
// Evite
while ( cond )
{
    código do bloco
} // while

// Redija assim
while ( cond )
{
    código do bloco
} // while

// Não escreva assim
while ( cond );

// Escreva assim
while ( cond )
{
    // corpo vazio
} // while

// Não escreva assim
if ( cond ) return;

// Nem assim
if ( cond )
    return;

// Escreva assim
if ( cond )
{
    return;
} // if

// Evite
if ( cond1 )
    if ( cond2 )
        return;
    else // Você tem certeza a que if pertence este else?
        exit(1);

// Escreva assim
if ( cond1 )
{
    if ( cond2 )
    {
        return;
    } else
    {
        exit(1);
    } // if
} // if
```

- 9) Organize as seqüências de seleção em ordem de probabilidade presumida de ocorrência.

O código que você normalmente espera que seja executado deve seguir o `if`, O código menos provável de ser executado deve seguir o `else`.

Exemplo

```
// Evite
Status = LeArquivo();
if ( Status != SEM_ERRO )
    Retorno = ERRO_DE_LEITURA;
else
{
    Status = EscreveArquivo();
    if ( Status != SEM_ERRO )
    {
        Retorno = ERRO_DE_ESCRITA;
    } else
    {
        FechaArquivo();
        MostraMensagem( MSG_ARQUIVO_SALVO );
    }
}

// Organize assim
Status = LeArquivo();
if ( Status == SEM_ERRO )
{
    Status = EscreveArquivo();
    if ( Status == SEM_ERRO )
    {
        FechaArquivo();
        MostraMensagem( MSG_ARQUIVO_SALVO );
    } else
    {
        Retorno = ERRO_DE_ESCRITA;
    }
} else
{
    Retorno = ERRO_DE_LEITURA;
}
```

ANEXO 4 – PADRÃO PARA A ESCOLHA DE NOMES

ESTRUTURA GENÉRICA DE UM NOME

- 1) Selecione um idioma (por exemplo, português, inglês) no qual será redigido o módulo. Todos os nomes e comentários contidos neste módulo deverão ser redigidos neste idioma.
- 2) Todos os nomes, independentemente da categoria do elemento que denominam, possuem a mesma estrutura. A estrutura padrão dos nomes de elementos é a seguinte:

$\{ \langle \text{Componente} \rangle \mid \langle \text{Módulo} \rangle \}_ \langle \text{Prefixo tipo} \rangle \langle \text{Tema} \rangle \langle \text{Sufixo Tipo} \rangle$

<i>Componente</i>	opcional, identifica o componente que torna público o elemento.
<i>Módulo</i>	opcional, identifica o módulo onde o elemento está declarado, ou o fato do elemento ser global e encapsulado no módulo (static).
<i>Prefixo tipo</i>	opcional, identifica o <i>tipo computacional</i> do elemento.
<i>Tema</i>	obrigatório, estabelece o significado do elemento designado pelo nome. O tema deve refletir precisamente o significado do objeto ou da ação designada pelo nome.
<i>Sufixo tipo</i>	opcional, complementa o <i>tipo computacional</i> do elemento.

- 3) Quando o nome designa um elemento cujo nome é difundido na literatura, utilize este nome publicado (por exemplo, A, B, e C como nomes dos parâmetros dos termos de uma equação do segundo grau).
- 4) Exceto para elementos rascunho, nomes devem ter entre 6 e 20 caracteres.

PREFIXO DE COMPONENTE, MÓDULO, GLOBAL ENCAPSULADO

- 1) Possuem prefixo todos os elementos globais públicos (externos) ou encapsulados, declarados em contexto de módulo. Não possuem prefixo os elementos declarados em contexto de classe, função ou bloco. Os prefixos de componente, módulo ou global encapsulado são mutuamente exclusivos.
- 2) Possui *prefixo de componente* qualquer elemento público definido por um componente formado por vários módulos (por exemplo, API). O elemento estará declarado no módulo de definição do componente identificado.
- 3) Possui *prefixo de módulo* qualquer elemento público definido por um módulo. O elemento estará declarado no módulo de definição do módulo identificado.

- 4) Possui *prefixo de elemento global encapsulado* qualquer elemento global encapsulado (static) definido em um módulo. O elemento estará declarado no módulo de implementação do módulo. O valor do prefixo será sempre: “ST”.
- 5) Ao programas em C, funções encapsuladas não conterão o prefixo de elemento global encapsulado.
- 6) O prefixo de componente ou de módulo deve ser estabelecido pela gerência de desenvolvimento e divulgado através de um glossário explorável via WWW.
- 7) O prefixo de componente ou módulo é composto por uma a três letras seguido do caracter sublinhado (“_”), é diferente de ST_ e é mutuamente diferente para todos os componentes e módulos conhecidos na organização.

Exemplos

BT_pObterElemLista identifica uma função que retorna um ponteiro para um elemento de lista e está definida no módulo BT.

ST_NUM_PORT é uma constante simbólica declarada por #define no módulo de implementação a que pertence o texto corrente.

ExibirJanela é um método de uma classe. Descobre-se isto pelo fato do nome principiar por verbo e não possuir prefixo de componente, nem de módulo, nem de global encapsulado.

BT_RaizFloresta identifica uma variável global pública declarada no módulo de definição do módulo BT.

RaizArvore em C++ identifica ou uma variável local, ou uma variável membro de classe. Em C identifica uma variável local.

- 8) A gerência de desenvolvimento deve manter um documento – *Catálogo de Prefixos de Produtos e Domínios* – identificando todos os prefixos produto e de domínio conhecidos na empresa. O documento deve registrar, para cada produto, subproduto e módulo, o prefixo e o nome completo do produto ou do módulo.

PREFIXO DE TIPO

- 1) Todos o elementos devem identificar o seu tipo computacional através do prefixo de tipo.
- 2) Não terão prefixo de tipo:
 - a) métodos e funções,
 - b) constantes simbólicas inteiras, inclusive constantes enumeradas,
 - c) nomes cujo tema implica, por convenção, o tipo computacional,
 - d) elementos locais de rascunho, definidos mais adiante neste padrão,

- e) referências a objetos, entretanto ponteiros para objetos terão prefixo de tipo.
- 3) Todos os caracteres do prefixo de tipo devem ser redigidos em minúsculas sem separação do restante do nome.
 - 4) Quando o nome designar uma classe C++, o caractere inicial do prefixo de tipo será “C” maiúsculo.
 - 5) Quando o nome denominar uma constante simbólica declarada em #define, todos os caracteres do prefixo de tipo deverão ser maiúsculas e o prefixo deverá ser separado do restante do nome por um caractere sublinhado.
 - 6) Em C++ o prefixo de tipo de declarações struct, union, enum e typedef deve iniciar com as letras tp.
 - 7) Em C o prefixo de tipo de declarações struct, union e enum deve iniciar com as letras tg (tag).
 - 8) Em C o prefixo de tipos declarados por intermédio de typedef deve iniciar com as letras tp.
 - 9) A gerência de desenvolvimento deve manter um *Catálogo de Prefixos de Tipo*.

Para facilitar o entendimento de um programa e para aumentar a capacidade do programador ou revisor determinar se o programa está correto, é recomendado que o tipo computacional do elemento faça parte integrante do nome. Foi adotado neste padrão a convenção húngara desenvolvida por Charles Simony.

Exemplos de tipos primários

c	char
c	nome de classe
h	handle, referência a um recurso alocado (ex.: janela)
i	inteiro declarado com int
id	identificador de um elemento, usualmente um valor unsigned
l	long
p	ponteiro
s	string
sz	string terminado com o caractere zero
tp	declaração de tipo
tg	declaração de tag ao declarar struct, union ou enum em C.

Exemplos de tipos declarados

mnu	menu
msg	mensagem
wnd	descritor de janela

Exemplos de declarações

hwndCorrente	<i>handle</i> da janela corrente. Note que o identificador de prefixo <i>wnd</i> é suficiente para que se saiba que se trata de uma janela, não sendo necessário repetir isto no tema. Ou seja, evite declarações redundantes: tais com <i>hwndJanCorrente</i> .
pszNomeAluno	ponteiro para um string terminado em zero contendo o nome de um aluno.
UN_Caluno	classe Aluno declarada no módulo UN.
tpDiaSemana	tipo enumeração de dias da semana.
DiaSemanaSegunda	valor constante enumerada do tipo dia da semana, denotando segunda-feira.

TEMA

- 1) Cada tema é composto por uma ou mais palavras. A leitura das palavras de esquerda para a direita deve refletir precisamente o significado do elemento denotado pelo tema.
- 2) Cada palavra componente de um tema deve iniciar com uma letra maiúscula, sendo minúsculas as demais letras. Não deve existir separador entre as palavras, nem entre o tema e o restante do nome.
- 3) No caso de constantes declaradas em `#define`, todas as palavras do tema devem ser redigidas com letras maiúsculas e devem ser separadas entre si e dos demais componentes do nome por um caractere sublinhado.
- 4) A palavra inicial de um tema que denomina uma função ou método deve ser um verbo no infinitivo impessoal
- 5) Métodos ou funções que processam eventos devem iniciar com a palavra `Ao` (On em inglês) seguida de um verbo no infinitivo (por exemplo, `AoTerminarExecucao()`).

- 6) Funções correspondentes a predicados (verificam se uma condição é ou não verdadeira) devem começar com Eh (Is em inglês) seguida do tema designativo da condição (por exemplo, EhPilhaVazia()).
- 7) Métodos correspondentes a predicados (verificam se uma condição é ou não verdadeira) devem começar com Esta ou Existe seguida do tema designativo da condição (por exemplo, Pilha.EstaVazia()).
- 8) Utilize somente palavras comuns. Evite palavras “difíceis”, ambíguas ou de pouco uso.
- 9) Assegure que cada palavra corresponda a um conceito concreto.
- 10) Os verbos escolhidos para designar função ou método devem corresponder a ações cujo efeito seja observável.
- 11) Utilize sempre a mesma palavra ou locução (seqüência de palavras) para denotar um dado conceito. Não use sinônimos ou pronomes.
- 12) Associe sempre o mesmo conceito a uma dada palavra ou locução. Evite o uso de palavras ou locuções com múltiplos significados.

Exemplos

<i>tpItemEstoque</i>	designa um tipo declarado, tipicamente um struct, que contém os dados de um item de estoque. Note que se o item de estoque correspondesse a uma classe, o nome seria CitemEstoque. Note ainda que o tema é ItemEstoque.
<i>EhidItemEstoqueValido</i>	designa um predicado que verifica se o identificador de um item de estoque é válido.
<i>ObterItemEstoque</i>	designa uma função que, dada uma identificação de um item de estoque, torna disponíveis os dados do correspondente item de estoque.
<i>szNomeItemEstoque</i>	string do nome do item de estoque, onde o string é terminado com zero.
<i>idItemEstoque</i>	identificador do item do estoque. É subentendido ser um inteiro ou unsigned.
<i>sidItemEstoque</i>	identificador do item de estoque, neste caso um string.
<i>DispItemEstoque</i>	disponibilidade do item no estoque.

NumItemEstoque contagem de diferentes itens contidos no estoque. Note a potencial ambigüidade do prefixo Num. Se utilizado de forma não uniforme, em um determinado contexto poderia designar a disponibilidade de um certo item e, em outro contexto, a contagem de diferentes itens. Adotamos aqui o significado padrão “contagem”. Os dois exemplos, números e disponibilidade, ilustram o uso do tema para denotar o tipo semântico.

ID_ITEM_ESTOQUE_NIL constante designando uma identificação de item de estoque inexistente.

Exemplo de uma linha de código:

```
PItemEstoque = ObterItemEstoque( idItemEstoque );
```

ABREVIACÕES

- 1) A gerência de desenvolvimento deve manter um Catálogo de Palavras, Locuções e Abreviações Padronizadas acessível via WWW. Este catálogo, além de registrar todas as locuções, as palavras e abreviações, deve fornecer, ainda, o respectivo significado padrão.
- 2) Ao criar nomes de elementos, utilize as abreviações contidas no catálogo.
- 3) Para manter o tamanho de nomes dentro do limite de 6 a 20 caracteres, as palavras que formam o correspondente tema podem ser abreviadas. No entanto, deve-se dar preferência à clareza e não ao tamanho do nome.
- 4) Nunca abrevie o verbo que denota uma função ou método.
- 5) Não abrevie palavras caso não economize 2 ou mais caracteres.
- 6) Elimine artigos, advérbios, pronomes e preposições do conjunto de palavras que formam o tema. Elimine também as palavras que não contribuam para definir o significado exato do tema.
- 7) Ao abreviar, evite trocadilhos e homofonias (por exemplo, Bin2-Hex).
- 8) Utilize uma das formas de abreviar a seguir, procurando a que resulte em um nome que melhor reflita o significado do elemento sendo batizado:

- a) Se a primeira sílaba da palavra for suficiente para denotar o significado, utilize a primeira sílaba. Caso não seja suficiente, observe os itens a seguir.
- b) Se a palavra inicia com uma consoante, elimine todas as vogais da palavra.
- c) Se a palavra inicia com uma vogal, deixe a vogal inicial e elimine todas as demais.

9) Evite distinguir temas por meio de um numeral.

10) Procure dar nomes curtos a temas de classes e de tipos declarados.

Exemplos de abreviações:

Nome Original	Nome Abreviado	Explicação
JanelaCorrente	JanCorr	Jan e Corr são abreviações padronizadas.
ItemDeEstoque	ItemEstoque	eliminou-se a preposição de
NumeroDeLinhasNoTexto	NumLinTexto	utilizaram-se abreviações padronizadas.
CalcularTamanhoDoSalto\ DeRolagemDaPagina	CalcularTamSaltoPagel	eliminaram-se palavras pouco significativas e abreviaram-se outras.

SUFIXO DE TIPO

- 1) Somente poderão ser utilizados sufixos de tipo contidos no *Catálogo de Sufixos de Tipo*.

Exemplos

ID_ITEM_ESTOQUE_NIL O termo NIL é um sufixo tipo que denota chave ou identificador nulo. Utilize NULL somente para referências ou ponteiros nulos.

VARIÁVEIS DE RASCUNHO

- 1) Variáveis de rascunho podem ser redigidas de forma simplificada.
- 2) Uma variável é de *rascunho* se satisfaz a todas as condições a seguir:
 - a) é local a uma função ou bloco;
 - b) não é um parâmetro da função;
 - c) possui pequenos domínios de definição.

Exemplos

<i>i, j, k</i>	contadores de ciclos
<i>Buffer</i>	armazenamento temporário de dados de entrada
<i>Temp</i>	valor temporário

CONSTANTES ENUMERADAS

- 1) O início do tema de cada uma das constantes enumeradas deve ser igual ao nome da enumeração, excluindo-se o prefixo de tipo.

Exemplo

```
enum JAN_tpCor
{
    JAN_CorAzul,
    JAN_CorVermelho,
    JAN_CorVerde,
    JAN_CorAmarelo
}
```

REFERÊNCIAS BIBLIOGRÁFICAS

- [AID2000] AIDO HomePage. **Aido HomePage**, 2000 – Endereço Eletrônico: <http://aidohomepage.cjb.net>. Data da consulta: 26/03/2000.
- [ART1994] ARTHUR, Lowell Jay. **Melhorando a qualidade do software: um guia para o TQM**. Tradução de Flávio Eduardo Frony Morgado. Rio de Janeiro : Infobook, 1994.
- [FER1995] FERNANDES, A. A. **Referência de software através de métricas**. São Paulo : Atlas, 1995.
- [FUR1994] FURLAN, José Davi. **Reengenharia da informação**. São Paulo : Makron Books, 1994.
- [HOL1994] HOLZNER, Steven. **Borland C++ programação for Windows**. São Paulo : Makron Books, 1994.
- [LAB2000] LABIUTIL – Laboratório de Utilizabilidade. **Legibilidade**, 2000 - Endereço Eletrônico: <http://www.labiutil.inf.ufsc.br/legib41.html>. Data da consulta: 28/08/2000.
- [MIZ1994] MIZRAHI, Victorine Viviane. **Treinamento em linguagem C++**. São Paulo : Makron Books, 1994.
- [PIR2000] PIRES, Hudson. **Tutorial On-Line Home Page**, 2000 – Endereço Eletrônico: <http://Tutorial.VirtualAve.net>. Data da consulta: 15/08/2000.
- [PRE1995] PRESSMAN, Roger S.. **Engenharia de software**. São Paulo : Makron Books, 1995.
- [SCH1992] SCHILDT, Herbert. **Turbo C++: guia do usuário**; São Paulo : Makron Books, 1992.

- [STA1983] STAA, Arndt von. **Engenharia de programas**. Rio de Janeiro : LTC – Livros Técnicos e Científicos Editora S.A., 1983.
- [STA2000] STAA, Arndt von. **Programação modular**. Rio de Janeiro : Ed. Campus, 2000.
- [ZSC1999] ZSCHORNACK, Fábio. **HoMe-PaGe Fábio Zschornack**, 1999 – Endereço Eletrônico: <http://minerva.ufpel.tche.br/~fabio/baguall/introd.htm>. Data da consulta: 16/10/2000.