Padrões para Codificação (Coding Standards)

Firmino dos Santos Filho
Brasil
firminosantos@totumcontinens.com.br

Abstract

A eliminação de todos os erros está, pelo menos em ambientes industriais atuais, bem além da capacidade tecnológica de software disponível. Não obstante a experiência sugere que a densidade de erros de códigos liberados possam ser reduzidos pela metade através da utilização de processos internos de verificação apoiados por ferramentas analíticas adequadas. Isto não requer mudanças de paradigmas ou idiomas. É necessário a determinação na adoção de técnicas comprovadas. De todas as formas de controle de qualidade de software, a inspeção de código é sem dúvida a mais efetiva. Quando apoiado por boas ferramentas, o custo unitário típico de identificação de um erro estático é até duas vezes menor que o custo da identificação através de métodos dinâmicos. Para realizar o trabalho de inspeção é necessário um padrão de codificação (coding standard) definindo requisitos que o código a ser inspecionado deve satisfazer. Um padrão de codificação são regras que governam o uso de uma linguagem de programação. Complementa o padrão da linguagem definindo características de uso aceitável e o inaceitável. Características de uso inaceitável conduzem ao erro ou a má interpretação. Características de uso aceitável evitam situações dúbias ou problemáticas. Isto não garante que o código esteja livre de erros, porque sempre se pode alcançar uma implementação imaculada da coisa errada.
A definição de uma convenção formal para a implementação de código fonte simplifica e agiliza a manutenção, pois facilita a compreensão e evita ambigüidades. Deste modo a manutenção pode ser feito de modo seguro. Muitos erros podem ser introduzidos quando da manutenção de um código confuso.

Índice

1. Introdução

1.1. Propósito

O estilo padronizado de codificação guia os projetos de software de modo a atingir os seguintes objetivos:

- Qualidade
- Software reproduzível
- Implementação uniforme de projetos de software
- Componentes de software reutilizáveis.

Uma padronização de nomes é um pré-requisito para a implementação uniforme do projeto de software.

1.2. Escopo

Este Estilo Padronizado de codificação é válido para todos os projetos de software (incluindo “firmware”).

1.3. Definições e Abreviações

As seguintes convenção são utilizadas neste anexo:

Nota	este tipo de parágrafo fornece informações gerais.
Regra x.x.y.	este tipo de parágrafo descreve a regra; é mandatório.
Guia x.x.y	este tipo de parágrafo descreve uma sugestão; não é mandatório, porém é altamente recomendável.

Identificadores reservados (em função da linguagem de programação) são identificados em negrito.

2. Convenção de Nomes

Regra Geral: Utilize descrições significativas em português ou inglês.
Utilize terminologia correlata a aplicação.
Use uma letra maiúscula entre palavras para facilitar a leitura (exemplo: aUltimoNome)
Não utilize nomes diferentes somente por letras maiúsculas e minúsculas.
Não utilize nomes com acentos ou cedilha.

Regra 2.1 Utilize uma descrição significativa em português do objeto/valor sendo passado, prefixando o nome com a letra “**a**”.

Exemplo aVeiculo, aCarro.

Regra 2.2 Utilize uma descrição significativa em português do campo, prefixando o nome com a letra "**m**".

Exemplo mPrimeiraCarro, mUltimoSegmento, mSecao.

Regra 2.3 Utilize um prefixo "**set**" no nome de um método de definição ou ativação.

Exemplo setPrimeiraCarro( ) , setSecao( ), setSegmento( ), setPosicao( ).

Regra 2.4 Utilize um prefixo "**get**" no nome de um método de leitura de dado ou de obtenção de dado. Para um método booleano utilize "**is**" como prefixo.

Exemplo

```
int getCarro( );

String getPosicao( );

Char getSegmento( );

Boolean isPersistent( );

Boolean isCrcCorreto( );
```

Regra 2.5 Os nomes das classes devem ser nomes, com a primeira letra de cada palavra interna em letra maiúscula. Tente manter o nome das classes simples e descritivas. Utilize palavras inteiras e evite abreviações, a menos que a forma abreviada seja largamente utilizada.

Exemplo

```
class Carro;

class VeiculosBasicos;

class RedeAuxiliar;
```

Regra 2.6 Os nomes dos métodos devem ser verbos, com a primeira letra do nome em letra minúscula e com a primeira letra de cada palavra interna em letra maiúscula.

Exemplo

```
long calculaArea( );

string trataMensagem( );

string verificaCrc( );
```

Regra 2.7 Os nomes de variáveis não devem começar com um "underscore" "_" ou sinal de dólar ("$").

Regra 2.8 Os nomes de constantes devem ser nomes significativos com todas as letras em maiúsculo com as palavras separadas por "underscore" "_".

Exemplo static const int TAMANHO_MAX = 4;

static const int AREA_MIN = 4500;

static const int NUMERO_MAX_DE_VEICULOS = 8;

Regra 2.9 Utilize um prefixo “on” no nome de um evento seguido de uma descrição significativa em português do evento.

Exemplo onChange

3. Documentação

3.1. O que documentar

Regra 3.1 Todo arquivo de código fonte deve conter um comentário introdutório que forneça informações do nome do arquivo e seu conteúdo.

Justificativa Facilitar a identificação do arquivo, autor, projeto, e histórico de alterações (Veja item 3.7 referente ao formato do cabeçalho de cada arquivo).

Guia 3.1 Argumentos / Parâmetros: descrever qual sua finalidade, restrições ou pré-condições .

Guia 3.2 Campos: descrever qual sua finalidade ou restrições.

Guia 3.3 Propriedades: descrever qual sua finalidade, restrições e características.

Guia 3.4 Classes: descrever o propósito e características da classe; histórico de desenvolvimento / manutenção.

Guia 3.5 Unidades de compilação: cada classe definida deve conter sua descrição. O arquivo deve conter um cabeçalho conforme item X.8.

Guia 3.6 Variável Local: descrever seu uso e propósito.

Guia 3.7 Funções Membro: descrever qual sua finalidade; seus parâmetros e retorno; descrever também qualquer exceção emitida e como a função modifica o objeto ao qual está associado. Inclua um breve histórico sobre modificações e exemplos de utilização da função. Pré e pós condições se aplicável.

3.2. Formatação dos Comentários

Regra 3.2 Cada função ou método deve iniciar com um comentário que esclareça o valor de retorno e parâmetros.

Guia 3.8 Blocos de comentário são utilizados para prover descrições dos arquivos, métodos, estruturas de dados, classes e algoritmos. Blocos de comentários devem ser definidos no inicio do arquivo e antes de cada método. Bloco de comentário interno a função ou método deve ser indentado no mesmo nível do código que este descreve. O bloco de comentário deve ser precedido por uma linha em branco de modo a separá-lo do código restante.

```
/*
 * Bloco de comentário C/C++
 */

(*
 * Bloco de comentário Delphi
 *)

;/
; Bloco de comentário Assembler
;/
```

Guia 3.9 Comentários curtos podem aparecer numa única linha indentado no mesmo nível do código que o segue. Se não for possível escrever o comentário numa única linha, este deve seguir o formato do bloco de comentário.

```
if (condição)
{
   // linha de comentário, para C/C++
   ...
}
```

Guia 3.10 Comentários muito curtos podem aparecer na mesma linha do código que este descreve, mas deve ser deslocado para a direita o suficiente para ser destacado do código. Se varias linhas de código contiverem comentários muito curtos, todos os comentários devem estar indentados na mesma coluna.

```
  if (a == 2)
    {
        b = TRUE;             // caso especial
    }
    else
    {
        b = isPrime(a);       // valido somente para números
....
    }
```

3.3. Comentário do código gerado

Guia 3.11 São válidos os comentários inseridos por ferramentas de geração de código automático.

3.4. Formatação das declarações

Guia 3.12 Deve-se utilizar somente uma declaração por linha, visto que isto encoraja o comentário individual.

Exemplo

```
int mPrimeiraCarro;          // comentário...

int mUltimoSegmento;                  // comentário ...
```

Guia 3.13 Procure alinhar o identificar e os nomes das variáveis declaradas.

Exemplo

```
int       mPrimeiraCarro;          // comentário...

string    mNomeSegmento;           // comentário ...
```

Guia 3.14 Declare a variável o mais próximo possível de onde ela será utilizada (desde que permitido pela linguagem de programação). Declare variáveis de índice utilizadas internamente a um for-loop dentro do bloco.

Exemplo

```
for (int i = 0; i < NUMERO_MAX_DE_VEICULOS; i++)   // i declarado no bloco “for”
{
    ...
}
```

Guia 3.15 Evite uma declaração local que esconda uma declaração em nível superior. Por exemplo, não declare uma variável num bloco interno com mesmo nome de outra de nível superior.

Exemplo

```
metodoDeCalculo( )
{
   int count;

   if ( condição )
   {
      int count = 0;          //evite a declaração de variável com mesmo nome.

   }
   ...
}
```

3.5. Formatação das classes

Regra 3.3 Não insira espaços entre o nome do método e o parêntese "(" que inicia a lista de parâmetros.

Regra 3.4 O colchete de abertura "{" (C/C++) deve ser colocado na linha seguinte a declaração da classe.

Regra 3.5 O colchete de fechamento "}" (C/C++) deve estar numa linha independente alinhado (mesma coluna) com o correspondente colchete de abertura; exceto caso seja uma declaração nula, onde neste caso se deve utilizar "{}". No caso da linguagem Delphi o fechamento (end;) deve estar alinhado (mesma coluna) com a correspondente declaração da classe.

Exemplo

```
class MyClassB                                  // C++
{
public:
    MyClassB();
    ~MyClassB() {}
    int emptyMethod() {}
    Sample(int aInteger, int aNewInteger)
    {
        ivar1 = aInteger;
        ivar2 = aNewInteger;
    }
protected:
    int mValue;
};

type                                            // Delphi
   TDate = class
   public
      constructor Create; overload;
      constructor Create(aOldDate, aNewDate, aOldYear: integer);
overload;
      procedure setValue(aNewDate: TDateTime);
   private
      fDate: mDateTime;
      function getYear: integer;
end;
```

3.6. Formatação de Código

3.6.1. Linhas em Branco

Linhas em Branco facilita a leitura através da separação de segmentos do código.

Regra 3.6 Três (3) linhas em branco devem ser inseridas nas seguintes circunstancias: Entre seções do arquivo fonte;
Entre definições de classes.

Regra 3.7 Duas (2) linhas em branco devem ser inseridas nas seguintes circunstancias: Entre métodos;

Regra 3.8 Uma (1) linha em branco deve ser inserida nas seguintes circunstancias:
Entre variáveis locais definidas no método e a primeira instrução;
Antes de um bloco ou linha única de comentários;
Entre seções lógicas dentro de um método para facilitar a leitura.

3.6.2. Espaços em Branco

Regra 3.9 Quatro (4) espaços devem ser inseridos como passo de indentação. Não deve ser utilizado o caracter TAB. Utiliza a configuração do editor para forçar a conversão de tabs em espaços.

Regra 3.10 Quatro (4) espaços devem ser inseridos como passo de indentação. Não deve ser utilizado o caracter TAB. Utilize a configuração do editor para forçar a conversão de tabs em espaços.

Exemplo

```
if (test() == true)
{
1234if (a > b)
1234{
12341234a = b;
1234}
}
```

Regra 3.11 Uma palavra chave (“keyword”) seguida por parênteses deve ser separada por espaço.

Exemplo

```
while (true)
{
    ...
}
```

Nota Este espaço em branco não deve ser utilizado entre o nome do método e seu respectivo parênteses de abertura. Isto auxilia distinguir entre "keywords" e chamadas de métodos.

Regra 3.12 Um espaço em branco deve ser inserido depois da virgula numa lista de argumentos.

Regra 3.13 Todos os operadores com exceção do ponto "**.**" devem estar separados do seus operandos por espaços. Não se deve separar operadores unários.

Exemplo

```
a += c + d;
a = (a + b) / (c * d);
printSize("size is " + foo + "\n");
```

Regra 3.14 As expressões numa instrução "for" deve ser separada por espaços em branco.

Exemplo

```
for (expr1; expr2; expr3)
```

3.6.3. Número de caracteres na linha de instrução

Regra 3.15 Linhas não devem conter mais do que 80 caracteres, visto que dificulta a leitura e compreensão.

3.6.4. Quebra de Linhas

Quando uma expressão não couber em uma única linha, quebre-a de acordo com as seguintes regras:

Regra 3.16 Quebra após uma virgula

Exemplo

```
someMethod(longExpression1, longExpression2, longExpression3,
           longExpression4, longExpression5);
var = someMethod1(longExpression1,
                  someMethod2(longExpression2,
                              longExpression3));
```

Regra 3.17 Prefira quebra nos níveis lógicos mais altos.

Exemplo

```
longName1 = longName2 * (longName3 + longName4 - longName5)
            + 4 * longname6; // Prefira

longName1 = longName2 * (longName3 + longName4
                         - longName5) + 4 * longname6; // Evite
```

Regra 3.18 Alinhe a nova linha com o inicio da expressão no mesmo nível da linha anterior.

Exemplo

```
someMethod(longExpression1, longExpression2, longExpression3,
               longExpression4, longExpression5);

    var = someMethod1(longExpression1,
                      someMethod2(longExpression2,
                                  longExpression3));
```

Regra 3.19 Indentação das declarações de métodos. Como primeira opção utilize as regras 3.15 à 3.18; como segunda opção, para se evitar longas indentações, use 8 espaços na indentação.

Exemplo

```
// Indentação convencional
    someMethod(int anArg, Object anotherArg, String aYetAnotherArg,
               Object andStillAnother)
    {
       ...
    }

    // Indentação de 8 espaços
    private static synchronized WorkingLongMethodName(int anArg,
            Object anotherArg, String aYetAnotherArg,
            Object andStillAnother)
    {
       ...
    }
```

Regra 3.20 A quebra de linha para expressões “if” devem seguir as regras anteriormente definidas.

Exemplo

```
// USE THIS INDENTATION
if ((condition1 && condition2) ||
    (condition3 && condition4) ||
    !(condition5 && condition6))
{
   doSomethingAboutIt();
}
```

3.7. Cabeçalho

3.7.1. Cabeçalho de Arquivo

Regra 3.21 Utilize o cabeçalho abaixo descrito para todos os arquivos de código fonte.

Exemplo

```
/* Copyright (c) 2003 FSF Company.
 *                   Todos os Direitos reservados.
 *
 * Autor(es):
 * Data de Criação:
 * Histórico de Modificações:
 * 01-Jan-2003 Autor, Índice da alteração, Descrição da alteração.
 * 01-Fev-2003 Autor, Índice da alteração, Descrição da alteração.
 */
```

onde: Índice da alteração poderá ser :

número da SA (solicitação de alteração) SA0453/03 ou caso ainda seja parte do projeto

o Dia da alteração (primeira alteração do dia): A010803

(Segunda alteração do dia): A010803B (C,D,...)

O índice será utilizado para referenciar as alterações, exemplo:

```
/*
 * Alteração A010803B, ..., comentários adicionais.
 */

    ... bloco de código adicionado

/*
 * Fim da alteração A010803B
 */
```

ou alteração pontual:

```
if (test() == true)
{
1234if (a > b)
1234{
12341234a = b;        //  A010803B,  ...,  comentários
adicionais
1234}
}
```

3.7.2. Cabeçalho de Método ou Função.

Regra 3.22 Utilize o cabeçalho abaixo descrito para todos os Métodos ou funções.

Exemplo

```
/* Nome: calculaArea
 *
 * Descrição: Descrição da finalidade do método, algoritmo utilizado e
 *            demais características pertinentes.
 * Parâmetros: aParametro1 -> descrição da finalidade do parâmetro
 *                  ...
 *             aParametron -> ...
 * Retorno: tipo -> descrição
 *
 * Objeto Associado: descrever o que e como altera o objeto associado.
 * Variáveis Locais: descrever seu uso e propósito.
 *
 * Campos / Propriedades: descrever qual sua finalidade, restrições e
 *                        características.
 *
 * Exceções emitidas: descrever.
 *
 * Pré e Pós-condições do método, se aplicável.
 *
 * Exemplos de utilização da função.
 *
 * Data de Criação:
 * Histórico de Modificações:
 * 01-Jan-2003 Autor, Índice da alteração, Descrição da alteração.
 * 01-Fev-2003 Autor, Índice da alteração, Descrição da alteração.
 */
        long calculaArea(int aParametro1, ..., long aParametron)
        {
            ...
        }
```

3.7.3. Cabeçalho de Modelo Visual

Regra 3.23 Utilize o cabeçalho abaixo descrito para todos os Modelos Visuais

```
        Copyright (c) 2003 FSF Company.
                           Todos os Direitos reservados.

Nome do Modelo
Descrição do Modelo
Autor(es):
Data de Criação:
Histórico de Modificações:
01-Jan-2003 Autor, Índice da alteração, Descrição da alteração.
01-Fev-2003 Autor, Índice da alteração, Descrição da alteração.
```

3.7.4. Cabeçalho de Classe

Regra 3.24 Utilize o cabeçalho abaixo descrito para Classes.

Exemplo

```
/* Nome: CodeTemplate
 *
 * Descrição: Descrição da finalidade da classe e
 *            características pertinentes.
 *
 * Campos / Propriedades: descrever qual sua finalidade, restrições e
 *                        características.
 *
 * Exceções emitidas: descrever.
 *
 * Pré e Pós-condições, se aplicável.
 *
 * Exemplos de utilização da classe.
 *
 * Data de Criação:
 * Histórico de Modificações:
 * 01-Jan-2003 Autor, Índice da alteração, Descrição da alteração.
 */
       public class CodeTemplate
       {
       public:
          /*
           * Descrição das variáveis publicas.
           */
          int mPublicVariable;

       protected:
           /**
           * Descrição das variáveis protegidas.
           */
          int mProtectedVariable;

       private:
          /**
           * Descrição das variáveis “private”.
           */
          int mPrivateVariable;

       public:
          /*
           * Descrição dos construtores
           */
          CodeTemplate(void);

          /*
           * Descrição dos destrutores.
           */
          virtual ~CodeTemplate(void);

          /*
           * Descrição dos métodos, conforme item 3.7.2.
           */
          void setMethod(int aParameter);

       }
```

4. Complexidade

Guia 4.1 Evite funções longas com mais de 100 linhas.

Justificativa Funções longas possuem as seguinte desvantagens:

- Dificuldade de compreensão. De modo geral, uma função deve ter menos que duas páginas de modo a facilitar sua compreensão.
- Funções complexas dificultam o teste. Por exemplo, caso uma função consista de 15 instruções "**if**" encadeadas então existem 2^15 (ou 32768) ramos diferentes a serem testados em uma única função.

Guia 4.2 Uma função deve ter no máximo 10 instruções ("statements") no seu corpo.

Justificativa Métodos com muitas instruções são de difícil compreensão.

Guia 4.3 Cada classe deve ter no máximo 20 métodos.

Justificativa Muitos métodos por classe dificulta sua compreensão.

Guia 4.4 Uma função deve ter no máximo 4 instruções encadeadas ("nested") do tipo **if-then-else, while, for, try** e **switch**.

Justificativa Muitos níveis de instruções dificultam a compreensão e o teste.

Guia 4.5 O código fonte deve ser suficientemente documentado. Por arquivo deve haver pelo menos uma linha de comentário para cada 2 linhas de código.

Justificativa Os comentários compõem uma significativa parte da documentação do projeto.

5. Padrão de codificação – C/C++

5.1. Introdução

Define regras e recomendações para a codificação em linguagem C/C++. Cada regra ou recomendação é seguida de uma justificativa. Os princípios deste padrão são segurança, confiabilidade, portabilidade e facilidade de compreensão. Isto envolve a definição de uma linguagem base e em seguida restringir seu uso a um subconjunto seguro, que não permite que características da linguagem sejam interpretadas diferentemente de um compilador para outro, ou características que permitam interpretação dúbia pelo programador. Código de qualidade é portável, facilmente compreendido, claro e não ambíguo.

5.2. Objetivo

O padrão de codificação para a linguagem C/C++ orienta os desenvolvedores de modo a obter os seguinte objetivos:

- Qualidade
- Implementação uniforme dos projetos de software
- Componentes de software reutilizáveis.

5.3. Escopo

Este padrão é valido para todos os desenvolvimentos de software de produto que utilizam a linguagem C++ e linguagem C onde for aplicável.

6. C/C++ Geral

Guia 6.1 Não declare uma variável antes que possa inicializa-la

Justificativa Uma declaração pode ocorrer em qualquer lugar que possa ser inserido um código. Para se manter o escopo o menor possível e chamar construtores o mais tarde possível (e como conseqüência somente onde ele é realmente necessário); variáveis podem ser declaradas no corpo da expressão: **for (int i; ...)**.

Regra 6.1 Palavras chaves **extern** e **goto** são proibidas.

Justificativa Isto quebra o principio do desenvolvimento de software orientado a objeto.

Regra 6.2 **malloc, realloc** e **free** são proibidas

Justificativa Foram substituídas por **new** e **delete**.

Regra 6.3 Aritmética utilizando **void *** pointer é proibido

Justificativa Somente seria necessário em blocos internos da implementação de uma classe ou função.

Regra 6.4 Argumento do tipo **void** pointer é proibido

Justificativa Somente seria necessário em blocos internos da implementação de uma classe ou função.

7. C/C++ Expressões

Expressões semânticas são versáteis em C++, entretanto muitas vezes ambíguas.

Regra 7.1 Expressões não devem conter múltiplos efeitos colaterais ocasionado pelo mesmo identificador sendo modificado mais de uma vez, ou devido ao mesmo identificador sendo modificado ou acessado.

Justificativa Algumas construções em C++, o padrão não define a seqüência de avaliação onde pode ocorrer os problemas.

```
x = i++ + a[i];              // Erro - 2 efeitos colaterais
                             // i modificado

x = i + a[i];                // correto
i++;
```

7.1. Operadores Unários – incremento e decremento (++, --)

Regra 7.2 Os operadores de incremento e decremento em forma de prefixo não devem ser atualizados conjuntamente com formas prefixadas de incremento e decremento na mesma expressão.

Justificativa Expressões que utilizem ambas as formas (pós e prefixada) de incremento (ou decremento) simultaneamente podem ser de difícil compreensão e podem levar a erros de implementação.

```
c = x+++++j;                 // Errado.

c = (++x) + (++j);           // correto ou
c = (x++) + (j++);           // correto ou
```

7.2. Operadores – multiplicativos (*, / %)

Regra 7.3 Tanto a operação de divisão como o resto da divisão devem ser protegidos por teste para que o operando seja sempre diferente de zero.

Justificativa Técnicas de programação defensiva como esta, reduzem o efeito de comportamentos indefinidos ou definidos pela implementação.

7.3. Operadores – Deslocamento (<<, >>)

Regra 7.4 Em uma expressão constante, o operando a direita deve ser não negativo nem deve implicar em deslocamentos extensos.

Justificativa Estas operações não são definidas na linguagem padrão.

```
int array[ 32<<-3];        //  indefinido
int array[ 0] = 32<<66;    //  indefinido
```

Regra 7.5 O operando a esquerda de um operador do tipo deslocamento a direita não deve ser com sinal.

Justificativa Um operando com sinal a esquerda de uma operação de deslocamento a direita irá produzir um deslocamento em algumas plataformas e deslocamento lógico em outras.

```
int             j;
unsigned int    x;
unsigned int    u;

x = j >> 8;                 // errado

x = u >> 8;                 // correto
```

7.4. Operadores Relacionais (<, >, <=, >=)

Regra 7.6 Os operandos de um operador relacional devem ambos serem utilizados com parênteses, a menos que sejam valores simples, identificadores ou chamadas de funções.

Justificativa A precedência dos operadores em C/C++ não é intuitiva. Parênteses extras ajudam ao leitor compreender o agrupamento dos operandos.

```
if (i <= 10)                // ok. - comparação de valores
simples.

if (i-k > 10 && k-i < 10)   // errado: a precedencia não é
obvia.

if (i < j < k)              // errado: significado não é
claro.
```

7.5. Operador Condicional (... ? ... : ...)

Guia 7.1 Evite operador condicional.

Justificativa A conveniência do operador condicional não compensa a dificuldade de compreensão do código.

```
 (a > b) ? i+=(a*b-a): j++;  // ruim
x = ((a > b) ? a : b);       // permitido
```

8. C/C++ Tipos e Dados

8.1. Declarações e Definições

Guia 8.1 Defina objetos no menor escopo possível. Exceção: prevenir operações de uso extensivo do “stack” e com propostos de depuração.

Justificativa É uma boa prática manter o escopo de objetos o menor possível por questões de manutenção de modo a evitar a alocação de espaço para objetos não desejados e chamadas desnecessárias a construtores e destrutores.

```
int function(int aParam)
{
   int returnValue = 0;

   if(aParam < 0)
   {
      returnValue = -1;      // Não é necessario criar e
destruir uma instância       // da classe SomeLarge
   }
   else
   {
      SomeLargeClass myObject(aParam);
...
      returnValue = myObject.getInteger();
   }
   return returnValue;
}
```

Regra 8.1 É proibido a utilização de dados globais.

Justificativa Dados globais quebram o principio de encapsulamento e leva a poluição do espaço de variáveis globais e como conseqüência aumenta o risco e esforço da fase de manutenção.

Regra 8.2 Variáveis e objetos definidos num bloco interno de uma função devem ter nomes únicos dentro da função.

Justificativa Nem todos os compiladores tratam o escopo de variáveis corretamente, por exemplo, numa instrução “for”. Isto também dificulta a compreensão e manutenção.

```
int someFunction(void)
{
   int i = 0;                // primeira ocorrência de i

   for(int i = 0; i < 10; i++)
   {                         // de acordo com a ISO C++ este i
é
                             // valido somente dentro deste
                             " loop" . Nova
                             // variavel i que dificulta a
                             compreenção.
   }
                             // fora o i anterior ainda
persiste.
}
```

8.2. Tipos “Float”

Regra 8.3 Variáveis do tipo “float” não devem ser comparadas por igualdade (==) ou por não igual (!=).

Justificativa Raramente a comparação direta leva ao resultado esperado; exemplo: **sqrt(3) * sqrt(3)** usualmente não resulta igual a 3, por causa do erro de arredondamento.

```
float f1, f2;
...
if (f1 == f2)                // não leva ao resultado esperado
```

8.3. Conversão de tipos

Regra 8.4 Se for necessário conversão de tipo, use conversão de tipo explicita, use os operadores do tipo “cast” do C++.

Justificativa Os operadores do tipo “cast” tem um propósito especializado que é avaliado pelo compilador e é muito mais seguro que o estilo antigo do “C” (que ainda é uma construção valida em C++). Os novos operadores aumentam a facilidade de compreensão e são mais fáceis de se manter.

8.4. Constantes

Regra 8.5 Use **const** sempre que possível (sempre que tiver significado).

Justificativa Declarar um item como **const** permite especificar uma restrição reforçada pelo compilador. Como conseqüência se um objeto não deve ser modificado ele deve ser declarado **const.**

```
// exemplos diferentes para ilustrar a versatilidade do const
char* some_pointer;

// variavel ponteiro de um string constante
const char* ptr_to_const_obj = " xyz" ;

ptr_to_const_obj[1] = 'A';               // erro do compilador
ptr_to_const_obj = some_pointer;         // o.k.

// constant pointer to variable string
char* const const_ptr_to_obj = " xyz" ;

const_ptr_to_obj[1] = 'A';               // o.k.
const_ptr_to_obj = some_pointer;         // erro do compilador

void find (const ULong ulBigNumber, UShort& rusSmallNumber);
AtmVclIndex getAtmVclIndex (const ULong ulId) const;
```

Regra 8.6 Constantes devem ser definidas utilizando-se **const.** Não use **#define.**

Justificativa Objetos **const** tem tipo, escopo e referencia tais como variáveis que estão sendo verificadas pelo compilador. Constantes criadas com

#define não possuem estas características desde que são mera substituição de texto, o que é muito importante também para a fase de depuração.

```
// use const e enum ao contrário de #define
const int MAX_SIZE = 1000;

void function(void)
{
   int someArray[MAX_SIZE];
   const int lowerLimit = 0;
   const int upperLimit = MAX_SIZE;

   for (int i = lowerLimit; i < upperLimit; i++)
   {
...
   }
```

Regra 8.7 **consts** e **enums** devem ser declarados dentro da definição da classe.

Justificativa Isto previne a poluição do espaço global de nomes e especifica explicitamente a origem do dado.

8.4.1. Constantes “Integer” e “Float”

Regra 8.8 O código não deve conter valores constantes explícitos – chamados de números mágicos.

Justificativa A facilidade de leitura requer a definição de constantes, e dando a estes valores um nome com significado. Isto também facilita a manutenção.

```
// ruim
for (int i = 1; i < 256; ++i)              //porque 256?
{
   if (ucVal[i] != 0xff)                   //porque != de 0xff
   {
      ulSum += ucVal[i];
   }
}

// bom
const UChar VAL_INVALID = 0xff;
const int   ARRAY_MIN   = 1;
const int   ARRAY_MAX   = 256;

for (int i = ARRAY_MIN; i < ARRAY_MAX; ++i)
{
  if (ucVal[i] != VAL_INVALID)
   {
      ulSum += ucVal[i];
   }
}
```

8.5. Ponteiros

Regra 8.9 As seguintes operações com ponteiros são proibidas: **!, &&, ||,** além da adição e subtração de dois ponteiros.

Justificativa Estas operações não definem a semântica do código. Exceções: subtração pode ser útil em aplicações de tempo crítico com arrays. Código especifico de hardware pode também ser necessário utilizar estas construções.

```
if(MyPtr > RefPtr)          // proibido
{
}

if(MyPtr != NULL)           // OK, existe uma expressão lógica
{                           //
}
```

Regra 8.10 Ponteiros para funções / métodos são proibidos.

Justificativa Isto não ocorre naturalmente em C++. Embora seja uma construção valida, é um legado do C que não é mais necessário em C++. São também de difícil compreensão e pode dificultar a depuração. A programação orientada a objetos prove a derivação de funções da classe base virtual de modo a se obter o mesmo resultado mas de modo mais transparente.

Guia 8.2 Evite ponteiros para ponteiros.

Justificativa São difíceis de se entender e tendem a provocar erros durante a manutenção.

8.6. Arrays

Guia 8.3 Classes contêineres devem ser usadas ao invés de arrays “built-in” (com exceção de arrays de tipos integrais).

Justificativa Arrays e polimorfismo não devem se misturar. Desde que não há verificação de limites (bound check), pode-se ter problemas quando se armazena um objeto derivado em arrays da classe base.

8.7. Strings

Guia 8.4 Não utilize strings com estilo da linguagem “C”.

Justificativa Os strings nativos do C são apenas arrays de caracteres. O uso de quase todas classes String disponíveis libera o desenvolvedor de uma serie de dificuldades e são a prova de erros de manipulação de memória e são mais confortáveis no uso que os strings nativos do C.

9. C/C++ Instruções

9.1. Instruções com “Label”

Guia 9.1 Instruções não devem ser endereçáveis através de labels (tarjas).

Justificativa A única razão para ser utilizar um label em uma instrução é para acessá-la através de “goto”. Estes pulos são proibidos.

9.2. Instruções de controle

Regra 9.1 Uma expressão de controle não deve ser uma designação.

Justificativa A separação da designação do teste em expressões de controle, evita efeitos colaterais e facilita a compreensão. Ações implícitas (da linguagem C++) devem ser evitadas em prol da clareza da

implementação. As versões atuais dos compiladores geram um código otimizado dispensando qualquer argumento de eficiência.

```
int i;

if (i = 1)                    // errado – designação com teste
implicito

if (i)                        // errado – comparação deve ser
explicita

if (i == 1)                   // CORRETO
```

Regra 9.2 As instruções **if, else, for, while** devem ser seguidas de um bloco {..}, mesmo quando houver apenas uma instrução (mesmo que nula (;)).

Justificativa Erros freqüentes ocorrem quando somente a indentação é utilizada para indicar que uma instrução pertence ao **if,** e na manutenção uma nova instrução é acrescentada *sem a adição dos marcadores de bloco {...}* para indicação que ambas as instruções pertencem ao **if**. Esta pratica também facilita a compreensão.

```
while ( ... );                // errado – sem {}
while ( ... )                 // correto
{
   // vazio
}

if ( ... ) i++;               // errado – sem {}
if ( ... )                    // correto
{
   i++;
}
```

9.2.1. Instruções “If”

Regra 9.3 Múltiplas construções utilizando **if**…**else if**... devem ter uma clausula **else** de modo a capturar todos os casos contrários.

Justificativa A programação defensiva requer a presença do else em construções de múltipla escolha, de modo a evitar estados não desejados. Use else vazios para tratamento de erros.

```
if (temp < 0)
{
...
}
else if (temp > 0)
{
...
}
else                              // correto – else preventivo
{
   error_message( .... );
}
```

Guia 9.2 Evite negações em expressões lógicas sempre que possível.

Justificativa Negações são difíceis de se avaliar e são facilmente incompreendidas.

```
if (activate(*pNvRamParam) != NO_SUCCESS ) // evite
{
   printf (" \n\rEverything successful!" );
}

if (activate(*pNvRamParam) == SUCCESS ) // melhor
{
   printf (" \n\rEverything successful!" );
}
```

9.2.2. Instruções “switch”

Regra 9.4 Todas as instruções **switch** devem conter uma clausula **default**, mesmo que esta clausula seja vazia.

Justificativa Aplica-se a mesma justificativa do item 9.2.1.

Regra 9.5 A expressão **switch** não deve conter nenhuma expressão lógica (um ou mais dos seguintes operadores: ‘>‘, ‘>=‘, ‘<‘, ‘<=‘, ‘==‘, ‘!=‘, ‘&&’, ‘||’ ou ‘!’).

Justificativa O uso de operadores relacionais ou lógicos numa expressão switch muito provavelmente leva ao erro, e é no minimo confuso.

Regra 9.6 A clausula **default** deve ser a última instrução do um bloco **switch**.

Justificativa Aumenta a clareza.

Regra 9.7 Em todo segmento de código deve haver uma instrução **break** no final.

Justificativa Aumenta a clareza do código.

```
switch (i)
{
    case '1':                // correto – case vazio
    case '2':
    {
       k = i
       break;
    }
    case '3':
    {
       k = i + 2;            // errado – sem break
    }
    default:                 // correto - default disponível
    {
       assert( … );
       break;
    }
}
```

9.2.3. Instruções “for”

Regra 9.8 Uma variável de controle não deve ser alterado dentro do corpo de uma instrução **for**.

Justificativa Modificar uma variável de controle é uma pratica perigosa e confusa – erros podem ser facilmente introduzidos quando ocorre uma alteração posterior a implementação.

```
for (int i = 0; i < NUMERO_MAX_DE_VEICULOS; i++ )
{
   i--;                      // ruim: altera o comportamento
usual
}
```

9.2.4. Instruções “return”

Regra 9.9 Funções devem ter exatamente uma entrada e uma saída.

Justificativa Com o propósito de clareza, as funções devem ser estruturadas. Funções que retornam **void** não devem ter nenhuma instrução **return**; todas as outras funções devem preferencialmente ter uma instrução **return**. Funções podem ter mais de uma instrução **return** se aumentar substancialmente a clareza.

10. C/C++ Funções e Métodos

10.1. Argumentos e Parâmetros das Funções

Regra 10.1 Não é permitido definir funções com um número não especificado de argumentos (notação com elipses).

Justificativa A função mais conhecida que utiliza um número de argumentos não especificados é a função **printf()**. O uso de tais funções não é recomendado visto que o recurso de verificação de tipo do C++ fica desabilitado neste caso. O uso deste tipo de argumento deve ficar restrito as funções padrões de biblioteca, o que considera-se como exceção a regra.

Guia 10.1 Prefira referencia como argumento das funções. Somente se realmente for necessário trabalhar com ponteiros de um objeto permita a utilização de argumentos do tipo “pointer”.

Justificativa Utilizando referencia no lugar de ponteiros como argumentos de função, o código pode ser mais facilmente compreendido, especialmente internamente a função. Outra vantagem é que não existe referencia nula, o que assegura que exista uma instancia do objeto para ser trabalhado.

```
// o uso de ponteiros é complexo
void addOneComplicated(int* integerPointer)
{
   *integerPointer += 1;
}

addOneComplicated(&j);       // call

// melhor
void addOneEasy(int& integerReference)
{
   integerReference += 1;
}

addOneEasy(i);               // call
```

Regra 10.2 Utilize referencias constantes (const &) no lugar de chamada por valor (call-by-value) a menos que seja utilizado um tipo de dado pré-definido.

Justificativa Uma das diferenças entre ponteiros e referencia é que não existe referencia nula em C++, enquanto existe ponteiro nulo (null-pointer). Isto significa que um objeto deve ser alocado antes de ser passado como argumento da função. A vantagem é não ser necessário testar a existência do objeto internamente a função.

Nota C++ em chamadas de função por valor (call-by-value) os argumentos são copiados no stack utilizando construtores de copia, que para objetos grandes, reduz o desempenho. Adicionalmente, o destrutor será chamado quando do encerramento da função. Argumento **Const &** significa que somente a referencia do objeto em questão é colocada no stack (call-by-reference) e o estado do objeto não pode ser alterado.

```
// ineficiente: uma copia do argumento é criada no stack
```

```
void foo1( String s );
String a;
foo1( a );     // call-by-value

// ineficaz: pode conter um ponteiro nulo
void foo4( const String* s );
String d;
foo4( &d );    // call-by-constant-pointer

// o argumento real pode ser modificado pela função
void foo2( String& s );
String b;
foo2( b );    // call-by-reference

// o argumento real não pode ser modificado pela função
void foo3( const String& s );
String c;
foo3( c );    // call-by-constant-reference
```

10.2. “Overloading” de Função

“Overloading” de funções pode ser uma ferramenta poderosa para a criação de uma família de funções que diferem apenas pelo tipo do argumento, entretanto, muito cuidado de ser tomado pois este recurso pode causar considerável confusão.

Guia 10.2 Quando se utilizar sobrecarga de função, todas as variações devem Ter a mesma semântica (devem ser utilizadas para o mesmo propósito).

Justificativa Se não for utilizado corretamente (tal como utilizar funções com mesmo nome para diferentes propósitos) pode-se chegar ao uso incorreto destas funções e gerar erros difíceis de serem localizados.

```
// uso adequado de overloading de função
class String
{
public:                                 // Use deste modo:
   //                                   // String x=” abc123” ;
   bool contains( const char c );       // bool i=x.contains(‘b’);
   bool contains( const char* cs);      // bool
j=x.contains(” bc1” );
   bool contains( const String& s);     // bool k=x.contains( x );
   // ...
};

// Este é o preço pago pela comodidade:
if ( my_str.contains(5) ){   // O compilador tem muitas
possibilidade
                             // de achar uma conversão :
                             // int -> char, int -> char*, int
-> String
                             // estas ambiguidades causam
                             // erros do compilador.
```

10.3. Parâmetros formais.

Regra 10.3 Os nomes dos argumentos formais da função tem que ser definidos e devem ser os mesmos na declaração da função e na definição da função.

Justificativa Prover nomes significativos para os argumentos da função faz parte da documentação da função. O nome do argumento pode clarear como o argumento é utilizado, reduzindo a necessidade de inclusão de comentários. Também é mais fácil se referir a um argumento na documentação de uma classe se este tem um nome.

```
int setPoint(int, int);      // ruim
int setPoint(int PontoA, int PontoB);     // ok.

int setPoint(int PontoA, int PontoB)
{
    // ...
}
```

10.4. Tipos e valores retornados pelas funções

Regra 10.4 Uma função não deve retornar uma referencia ou um ponteiro para uma variável local não estática (variável alocada no stack por exemplo).

Justificativa Se uma função retorna uma referencia ou ponteiro para uma variável local, a posição de memória que ela se refere já foi dealocada, logo estará acessando a variável incorreta.

```
char* strangeFunction(void)
{
   char localBuff[Max];

   return localBuff;         // ruim: retorna ponteiro para o
stack!
}

a = strangeFunction();       // Pode até funcionar para alguns
testes,
                             // mas cedo ou tarde produzirá
                             // exceção de memória!
```

10.5. Funções “Inline”

Funções Inline possuem a vantagem de serem de execução mais rápida do que funções comuns. A desvantagem é que expõe sua implementação, visto que por definição a função inline deve adicionada a definição da classe que a utiliza. Use a extensão “.inl” para um arquivo de implementação Inline. O compilador não é obrigado a criar a função como Inline. O critério de decisão difere de um compilador para outro. Freqüentemente é possível definir um flag no compilador que o obriga a emitir uma mensagem de aviso toda vez que uma função não for implementada como Inline ao contrário da declaração.

Guia 10.3 Evite longas funções **Inline** (mais do que 3 instruções).

Justificativa Como mencionado acima, a implementação é dependente do compilador, quebra a modularidade e dificulta a manutenção.

11. C/C++ Classes

As Classes introduzem o paradigma da orientação a objeto na linguagem C++. São objetos ou dados do tipo abstrato. Elas não só contém dados mas também código procedural. Elas encapsulam grupos de dados e os métodos para trabalhar com os dados. Pela derivação ou herança certos aspectos da classe base, o chamado polimorfismo é introduzido. Trabalhar com estas características complexas da linguagem não é intuitivo e as regras a seguir serão um caminho mais seguro.

11.1. Direitos de Acesso

Guia 11.1 Os dados da classe devem ser do tipo "**private**", sempre que possível.

Justificativa Do contrário as seguintes desvantagens podem ocorrer:

1. Dados do tipo **"public**" representam uma quebra da orientação a objeto visto que o principio do encapsulamento é violado. Este tipo de dado pode ser manipulado fora da classe e como conseqüência a integridade do objeto depende de eventos externos levando a uma implementação complexa. Dados do tipo **private** somente podem ser manipulados por métodos pertencentes a própria classe e como conseqüência mantém um encapsulamento consistente.
2. A depuração de classes com membros de dados **public** é muito difícil porque o provável local do erro pode ser em qualquer lugar da aplicação e não é restrito a sua própria classe.
3. Evitando o uso de dados do tipo **public** a implementação interna da classe é totalmente oculta dos usuários. A implementação da classe pode eventualmente ser totalmente modificada sem afetar o código dos usuários desta classe.
4. Dados do tipo **protected** devem ser evitados porque eles podem ser manipulados por classes derivadas, causando problemas análogos aos problemas do uso de dados do tipo **private**, porém com menores implicações.

Regra 11.1 Siga a regra de definição de Classe (vide Anexo B). Defina consts, enums, variáveis e métodos e para cada seção use primeiro **public**, então **protected** e por último **private**.

Justificativa Padrão de documentação.

```
class Priority
{
protected:
   int mPriority;
private:
   int mOldPriority

public:
   int getPriority(void);
private:
   void setOldPriority(void);
};
```

Guia 11.2 O uso de estruturas (**structs**) deve ser evitado**.**

Justificativa Estruturas por definição contém somente dados públicos e não seguem os princípios básicos da orientação a objeto. Embora se possa declarar também seções do tipo **protected** e **private**.

```
struct Priority
{
protected:
   int mPriority;
private:
   int mOldPriority

public:
   int getPriority(void);
private:
   void setOldPriority(void);
};
```

11.2. Retorno de funções membro

Regra 11.2 Não se deve retornar um ponteiro ou referencia não constante (não const) para fora da classe a partir de uma função membro publica, exceto nos casos de compartilhamentos de dados com outros objetos.

Justificativa Esta é um outro caminho que pode quebrar o encapsulamento. Se um objeto possui dados que podem ser modificados externamente, inconsistências podem resultar facilmente.

```
// Nunca retorne referencias não constante a partir de uma
função membro publica
class Account
{
public:
   Account(int aMoney) : mMoneyAmount(aMoney) {};
   const int& getSafeMoney() const { return mMoneyAmount; }
   int& getRiskyMoney() const { return mMoneyAmount; } // Não!

private:
   int mMoneyAmount;
};

Account myAcc(10);

// Erro! Não se pode alterar constantes.
myAcc.getSafeMoney() += 1000000;

// myAcc::moneyAmount = 1000010 !
myAcc.getRiskyMoney() += 1000000; // quebra do encapsulamento.
```

11.3. Funções Amigas

Operações sobre um objeto são providas por uma coleção de classes e funções. A função amiga (**friend**) é uma função não membro da classe, que tem acesso a membros não públicos da classe. Um **friend** oferece um modo ordenado de contornar o encapsulamento dos dados da classe, sem quebrar totalmente este encapsulamento. A utilização de uma classe amiga pode ser vantajoso pois prove funções que requerem dados que normalmente não estariam disponíveis.

Um manipulador de classes do tipo container são exemplos do uso de **friends**. Um manipulador prove acesso aos dados da classe do tipo container sem ser membro do container nem do objeto armazenado, mas um amigo da classe container. Isto não quebra o encapsulamento desde que todos os dados membros da classe container sejam do tipo **private**, e ainda haja acesso aos dados dentro do container pelo manipulador **friend**.

Guia 11.3 Utilize friend exparsamente e proveja uma documentação detalhada com for utilizar.

Justificativa **friend**s ocasiona um furo no encapsulamento que pode ser útil. Entretanto, adiciona complexidade extra que deve ser documentada em detalhe.

Guia 11.4 Classes **friends** só devem ser utilizadas para prover funcionalidades adicionais que é melhor manter fora da classe original.

Regra 11.3 Um objeto não deve ser **friend** de mais de uma classe.

Justificativa Mesmo que **friends** sejam uma forma ordenada de contornar o encapsulamento, ele contradiz este principio elementar. O uso inadequado desta característica leva a confusão e dependência que uma vez implementada dificilmente será resolvida.

11.4. Funções Membros const

Regra 11.4 Uma função membro que não altera o estado de um objeto (as instancias das variáveis) deve ser declarada como **const**.

Justificativa Funções membros declaradas como **const** não podem alterar dados membros e são as únicas funções que podem ser invocadas sobre um objeto constante (**const**). (Tal objeto é intrinsecamente inútil sem métodos constantes (const)). Uma declaração **const** é uma forma segura que objetos não serão modificados quando não devem. Uma grande vantagem provida pelo C++ é sua habilidade de “overload” de funções com respeito a sua característica **const**.

```
//função declarada como acesso const para dados internos a
classe
class SpecialAccount : public Account
{
public:
   void setAmountOfMoney(int aMoney);
   //int getAmountOfMoney(void); // Não, deve-se utilizar uma
função const
                                  // para se acessar um dado do
                                  // objeto sem alteração
   int getAmountOfMoney(void) const;      // Correto.
   // ...
private:
   int mMoneyAmount;
};
```

Regra 11.5 Se o comportamento de um objeto é dependente de um dado externo ao objeto, este dado não deve ser modificado por funções membro **const**.

Justificativa Nos casos excepcionais em que ocorre esta situação, este dado deve ser tratado como se este pertencesse a classe e como conseqüência não deve ser modificado por funções membro **const**.

11.5. Construtores e Destrutores

Guia 11.5 Dados membros de uma classe devem ser inicializados utilizando-se uma lista de inicialização que irá inicializar os dados na ordem que estes são declarados.

Justificativa A lista de inicialização é usualmente o modo mais eficaz de se inicializar dados. Manter uma ordem de declaração explicita documenta a ordem que o compilador utilizará, o que previne desenvolvedores inexperientes de tentar ou introduzir dependências entre os dados.

```
Class MyClass
{
public:
   // constructor
   MyClass(int aStatusInfo);

private:
   int mStatus;
};

// initialisation list
MyClass::MyClass(int aStatusInfo) : mStatus(aStatusInfo)
{
};
```

Regra 11.6 Todas as classes que tenham funções virtuais devem definir um destrutor virtual.

Justificativa Se uma classe tem uma função virtual mas sem um destrutor virtual é utilizada como classe base, pode ocorrer uma surpresa se um ponteiro para esta classe for usado. Se tal ponteiro é designado como instancia da classe derivada e se “delete” é utilizado neste ponteiro, somente o destrutor da classe base será invocado. Se o programa depende do destrutor da classe derivada ser invocado, o programa falhará.

```
// Problemas, se o destrutor da classe polimorfica é não
virtual
class Base
{
public:
   Base();                    // construtor default
   ~Base();                   // sem destrutor virtual
   // virtual ~Base();        // <- esta seria o correto.
   virtual void foo() { cout << ” Base::foo”  << endl; }
};

// Classe derivado “ override”  foo()
class Derived : public Base
{
public:
```

```
   Derived();                // construtor para os dados
privados
   //foo " overridden"
   virtual void foo() { cout << " Derived::foo"  << endl; }
private:
   bool IsUpToDate;          // own data
};

void func( Base* pb )
{
   delete pb;                // delete explicito:
}                            // destrutor ~Base() é chamado

Derived d;
func(&d);                    // ~Base() é chamado, mas
~Derived()
                             // seria necessário pois existem
=>
                             // os dados proprios da classe
" Derived" ,
                             // que não são destruidos!
```

Regra 11.7 Uma classe que utilize "**new**" para criar instancias gerenciadas pela classe, deve definir um construtor do tipo "**copy**" e designar um operador. A classe que não necessite prover esta semântica deve declara-la como "**private**" de modo a prevenir seu uso.

Justificativa Um construtor "**copy**" é recomendado para se evitar surpresas quando um objeto é inicializado utilizando um objeto do mesmo tipo. Se um objeto gerencia a alocação e desalocação do objeto no "**heap**" (o gerenciador do objeto tem um ponteiro do objeto a ser criado pelo construtor da classe), somente o ponteiro será copiado. Isto leva a duas chamadas do destrutor para o mesmo objeto (no "**heap**"), provavelmente resultando num "**run-time error**". As operações de designação não são herdadas como outros operadores, causando um situação muito similar. O compilador irá automaticamente definir uma designação chamando uma designação para cada dado membro o que leva a ponteiros do mesmo endereço físico. Isto novamente leva a mais de uma desalocação se objetos são destruídos ou dereferenciado de memória desalocada e como conseqüência causa um "**runtime errors**". Se uma copia ou designação de tal classe não faz sentido (por exemplo existe sempre somente uma instancia deste objeto em toda a aplicação) é razoável declarar estes operadores como "**private**". Isto previne que alguém posteriormente tente chamar um construtor "**copy**" gerado pelo compilador ou operador de designação provocando um erro de compilação.

```
//Definição de uma classe " perigosa"  que não tem um
construtor " copy"
#include <string.h>

class String
{
public:
   String(const char* cp = "");           // Construtor
   ~String();                             // Destrutor
private:
   char* sp;
};

// Constructor
String::String(const char* cp) : sp( new char[strlen(cp)] )
{
   strcpy(sp,cp);
}

String::~String()    // Destrutor
{
   delete sp;
}

// Classe String " Perigosa"
void main(void)
{
   String w1;
   String w2 = w1;
   // atenção: na copia bit a bit de w1::sp,
   // o destrutor de W1::SP será chamado duas vezes:
   // primeiro quando w1 é destruido; e novamente quando w2 é
destruido.
}

// Classe " segura"  tendo um construtor " copy"  default
#include <string.h>
class String
{
public:
   String(const char* cp = "");           // Construtor
   String(const String& sp);              // Construtor copy
   ~String();                             // Destrutor
private:
   char* sp;
};

// Construtor
String::String(const char* cp) : sp( new char[strlen(cp)])
{
   strcpy(sp, cp);
}

String::String(const String& aString) :
   sp(new char[strlen(aString.sp)])
{
   strcpy(sp, aString.sp);
}

String::~String()                         // Destructor
{
```

```
   delete sp;
}

// " Safe"  String class
void main(void)
{
   String w1;
   // Copia segura : String::String(const String&) Chamada.
   String w2 = w1;
}
```

Guia 11.6 Evite utilizar objetos globais em construtores e destrutores.

Justificativa No processo de inicialização de objetos estáticos, não existe garantia da ordem de inicialização, que podem estar definidos em varias unidades de compilação.

```
// O seguinte exemplo mostra porque chamadas a
// objetos estáticos dentro de um construtor pode falhar.
class MyClassA
{
public:
    MyClassA (const int aValue = 0);
    ~MyClassA () {}
public:
    inline int getValue(void) const { return mValue; }
protected:
    int mValue;
};

class MyClassB
{
public:
    MyClassB ();
    ~MyClassB () {}
public:
    inline int getValue(void) const { return mValue; }
protected:
    int mValue;
};

// Não está claro, qual rotina " rootValue"  ou " virtualValue"
// será inicializada primeira.
static const MyClassA rootValue(100);
static MyClassB virtualValue;

MyClassA::MyClassA (const int aValue) : mValue(aValue)
{
}

MyClassB::MyClassB ()
{
    // Situação perigosa, pois " rootValue"  pode não Ter sido
instanciada ainda!
    mValue = rootValue.getValue() + 1000;
```

```
}
```

Guia 11.7 Construtores que recebem um único argumento devem ser adicionados da palavra chave “**explicit**”.

Justificativa Este tipo de construtor será usado pelo compilador para executar conversão implícita embora elas sejam raramente escritas para este propósito.

```
Objetos da seguinte classe podem ser designado valores que
combinem com o tipo de construtor ou do tipo da classe:


class X
{
public:
   X(int);
   X(const char*, int = 0);
};


Então, a seguinte designação é correta:


void f(X arg)
{
   X a = 1;
   X B = "Jessie";
   a = 2;
}


Entretanto, objetos da seguinte classe podem ser designados
valores que combinem somente com o tipo de classe:


class X
{
public:
   explicit X(int);
   explicit X(const char*, int = 0);
};


O construtor " explicit" requer que os valores no designação
das instruções a seguir sejam convertido para o tipo da classe
para o qual está sendo designado.


void f(X arg)
{
   X a = X(1);
   X b = X("Jessie",0);
   a = X(2);
}
```

11.6. Operador de designação (=)

Regra 11.8 Um operador de designação que execute uma ação destrutiva deve estar protegido contra executar esta ação no objeto em que está operando.

Justificativa Um erro comum é designar um objeto com ele mesmo (a=a). Normalmente, os destrutores das instancias alocadas no "**heap**" são invocados antes que a designação ocorra. Se um objeto é designado a si mesmo, os valores da variável de instancia será perdido antes da designação. Isto pode levar a estranhos "**run-time errors**". Se a = a é detectado, o objeto designado não deve ser alterado.

```
// o operador de designação verifica se não é uma designação
para si mesmo.
const MyObj& MyObj::operator=(const MyObj& o)             //
operador de designação
{
  if( this != &o)                          // verifica
  {
    delete....                             // ação destrutiva
permitida agora.
    ...
  }
  return *this;
}
```

Regra 11.9 Um operador de designação deve retornar uma referencia "**const"** para o objeto designado.

Justificativa Se um operador de designação retorna "void", então não é possível escrever a = b = c. Pode ser tentador definir que este operador retorne uma referencia para o objeto designado. Infelizmente este tipo de projeto pode ser de difícil compreensão. A instrução (a = b) = c pode significar que **a** ou **b** é designado o valor de **c** antes ou depois de que **a** seja designado o valor de **b**. Este tipo de código pode ser evitado definindo o retorno do operador de designação como sendo uma referencia **const** para o objeto designado ou o objeto que originou a designação. Desde que o retorno do objeto não pode ser colocado no lado esquerdo da designação, não faz diferença qual das soluções é retornada.

```
//Retorno incorreto e correto de um operador de designação void
void MySpecialClass::operator=
   ( const MySpecialClass& msp ); // ?

MySpecialClass& MySpecialClass::operator=
   ( const MySpecialClass& msp ); // Incorreto

const MySpecialClass& MySpecialClass::operator=
   ( const MySpecialClass& msp ); // Recomendado.

//Definição de uma classe com um operador de designação
" overloaded"
class DangerousBlob
{
public:
   const DangerousBlob& operator=( const DangerousBlob& dbr );
   // ...
private:
   char* cp;
};

// Definição do operador de designação
```

```
const DangerousBlob&
DangerousBlob::operator=( const DangerousBlob& dbr )
{
   // proteção contra designação a si proprio
   if ( this != &dbr )
   {
      delete cp;
   }
   return *this;
}
```

11.7. “Overloading” de Operador

As mesmas regras de “overloading” de função se aplicam a “overloading” de operador (veja seção 10.2).

Guia 11.8 Use “overloading” de operador exparsamente e de maneira uniforme.

Justificativa “overloading” de operador tem vantagens e desvantagens. Uma vantagem é que o código que usa classe com operadores “overloaded” podem ser escritos mais compactamente (facilita a compreensão). Outra vantagem é que a semântica pode ser simples e natural. Uma desvantagem é que fácil ter uma compreensão errada do operador “overloaded”. Num caso extremo poderíamos ter o operador (+) sendo redefinido como (-) e o operador de menos (-) sendo redefinido como de mais (+).

Regra 11.10 Quando existem dois operadores opostos (tais como == e !=) , deve-se definir ambos os operadores.

Justificativa Se o operador != foi definido para uma classe o usuário pode se surpreender se o operador == não foi definido.

11.8. Herança

Guia 11.9 Para as classes derivadas forneça acesso aos dados membro da classe declarando funções de acesso “**protected**”.

Justificativa Uma classe derivada freqüentemente requer acesso a dados membro da classe base com a finalidade de criar funções. A vantagem em se utilizar funções “**protected**” é que os nomes dos dados da classe base não são visíveis nas classes derivadas, protegendo-os contra mudanças acidentais. Tais funções de acesso devem somente retornar valores dos dados membro (**read-only access**). Isto é feito invocando funções **const** para os dados membro.

Nota Assume-se que o desenvolvedor que usa herança, sabe o suficiente a respeito da classe base de modo a estar apto a utilizar os dados “**private**” corretamente.

Guia 11.10 Evite herança múltipla. Herança com mistura de classes (interfaces) com somente métodos virtuais puros pode ser utilizado.

Justificativa Herança múltipla cria dependências de classes que causam problemas no projeto. Somente em casos excepcionais a herança múltipla pode ser

uma solução viável, mas considerando-se as dificuldade de implementação e compreensão, deve-se evitar.

Note: A derivação de uma classe a partir de mais de uma classe base é denominada herança múltipla.

12. C/C++ "Exceptions"

Regra 12.1 Construtores devem ser analisados em função do aspecto segurança quando da introdução das exceções.

Justificativa Alocações de memória não planejadas em construtores podem levar a "memory leaks" em caso de ocorrência de exceções.

```
// classe insegura
class Y
{
   int* p;
   void init();
public:
   Y(int s){ p = new int[s]; init(); }
   ~Y { delete [] p; }
...
}
// Se durante a chamada ao construtor, a rotina " init()"
gerar uma " exception"
// a memoria alocada para 'p' não será liberada,
// visto que o construtor não foi interamente executado.

// Classe segura, variante da descrita acima:
class Z
{
   vector<int> p;
   void init();
public:
   Z(int s) : p(s) { init(); }
...
}
// A memoria utilizada para 'p' agora é manipulada por vetor.
// Se 'init()' gera uma 'exception', a memoria adquirida será
liberada
// quando o destrutor de 'p' for invocado (implicitamente).
```

13. C/C++ Memória Dinâmica

O conceito de gerenciamento de memória dinâmica em C++ difere do conceito em C. Objetos gerenciam o armazenamento de seus dados pela alocação e desalocação de memória com os operadores 'new' e 'delete', que estes invocam nos seus construtores e destrutores. Este encapsulamento libera o usuário da classe das preocupações com alocação de memória, visto que esta (alocação) já foi feita na classe (ou pelo menos deveria estar). Este capitulo menciona algumas armadilhas deste contexto.

Regra 13.1 Proibido o uso de **malloc**, **realloc** ou **free**. Utilize **new** e **delete**

Justificativa Em C malloc, realloc e free são utilizados para alocar memória dinâmica no "heap". Isto pode levar a conflitos com o uso dos operadores **new** e **delete** do C++.

Utilizações inseguras:

1. Invocar **delete** para um ponteiro obtido via malloc/realloc,
2. Invocar **malloc/realloc** para objetos com construtores,
3. Invocar **free** para qualquer objeto alocado utilizando **new**.

Portanto, evite o uso de malloc, realloc e free.

Regra 13.2 Sempre coloca colchetes (“[]”) para **delete** quando desalocando arrays.

Justificativa Se um array **a** do tipo T é alocado, é importante invocar **delete** no modo correto. Escrever somente **delete a;** pode resultar que o destrutor seja invocado somente para o primeiro objeto do tipo T ("resource leak"). Escrevendo **delete [m] a;** onde **m** é um inteiro maior do que o número de objetos alocados anteriormente, o destrutor para T será invocado para uma memória que não representa os objetos do tipo T. O modo mais fácil é utilizar **delete [] a;** visto que o destrutor será invocado somente para os objetos alocados anteriormente.

```
//  Modo certo e o errado para invocar delete para arrays com
destrutores
int n = 7;

// T é um tipo com construtores e destrutores definidos
T* myT = new T[n];

// ...
delete myT;                    // Não! Destrutor chamado somente
           para o
                               // primeiro objeto do array

// Não! Destrutor chamado para memória fora do " range"  do
array.
delete [10] myT;

delete [] myT;                 // OK, e é sempre seguro!
```

Guia 13.1 Quem quer que aloque memória é responsável por sua liberação.

Justificativa Uma grande vantagem do C++ sobre o C é a possibilidade de encapsular o gerenciamento de memória no objeto. Isto significa que a alocação deve ocorrer no construtor da classe e a desalocação seja feita no destrutor. Deste modo os recursos (memória) serão liberados depois que o tempo de vida do objeto expire.

```
// " resource leak"  resultante de um mau gerenciamento de
memória
String myFunc(const char* aArgument)
{
   String* temp = new String(aArgument);
   return *temp;               // temp nunca é desalocado.
                               // Um usuário de " myFunc"  não
pode liberar
                               // a memória utilizada porque
somente obtem uma
                               // copia. Não utilize esta forma
}
```

Regra 13.3 Sempre designe um valor “NULL” para um ponteiro que aponte para memória liberada.

Justificativa Evita-se que se acesse memória liberada. Isto pode criar um problema de solução complexa quando existem diversos ponteiros apontando para a mesma área de memória, visto que C++ não tem um coletor de lixo (“garbage collector”).

14. C/C++ Pré-processamento

O pré-processamento manipula as substituições de texto. É o mesmo pré-processamento conhecido do C. Entre as tarefas típicas estão suprimir comentários, substituição de constantes definidas com a diretiva #define e seleção de texto pela avaliação das diretivas **#if**, **#else, #endif**. Não existe verificação de tipo. A implementação de construções complexas no pré-processamento leva a erros de difícil compreensão.

14.1. Diretivas

Guia 14.1 Não é recomendado o uso de diretivas de pré-processamento, exceto para prevenir múltipla inclusões (**#include**s).

Justificativa O uso de diretivas de pré-processamento faz o processo de geração do código complexo, dificulta a compreensão e manutenção.

14.2. Macros

Regra 14.1 Macros não devem ser utilizados para substituição de construções da linguagem.

Justificativa São difíceis para depurar, e são inseguros pela falta de tipo (não tipificados) e podem levar a efeitos colaterais.

```
// proibido
#define OK        0          // utilize membro const
#define GOOD      (-1)       // utilize membro const
#define SQUARE(a) ((a)*(a))  // utilize função (inline)
```

14.3. Inclusão de arquivos

Regra 14.2 A diretiva **#include** somente pode ser seguida pelos seguintes caracteres **<...>** ou "**...**" contendo o nome do arquivo apropriado.

Justificativa Qualquer coisa diferente não está definido.

Regra 14.3 Os seguintes caracteres não podem estar presente na diretiva **#include**: **‘ ” \\ /***

Justificativa O significado destes caracteres não estão padronizados.

Regra 14.4 Os arquivos de “headers” devem ser sintaticamente independente.

Justificativa Independente significa que estes devem conter todas as diretivas **#include** necessárias, de modo a que possam ser compilados isoladamente. O encapsulamento requer que o usuário da classe necessite somente estar familiarizado com a classe que ele deseje utilizar sem ter que se preocupar com a necessidade de definição de outras classes (interdependentes) nos arquivos de “header”.

Regra 14.5 Cada arquivo de “header” de prover um mecanismo de guarda contra múltiplas inclusões.

Justificativa Este mecanismo pode ser implementado encapsulando o código dentro de um par de diretivas **#ifndef** / **#define** no começo do arquivo e um **#endif** no fim. Este mecanismo evita “overhead” de processamento do compilador reduzindo significativamente o tempo de processamento.

Regra 14.6 Não utilize caminhos (paths) absolutos ou relativos nas diretivas de **#include**.

Justificativa Um código portável deve ser independente do ambiente em que é escrito. Paths explícitos levam também a problemas de manutenção nos arquivos de “makefiles” e no processo de construção com um todo. Caminhos de procura (search paths) para os arquivos de “header” podem ser especificados na linha de comando do compilador de modo a não ser necessário alterar informações no código.

```
// Não utilize caminhos absolutos ou relativos.
#include <../include/fnutt.h>

// Ruim!
#include <c:\msc7\include\sys\socket.h>
```

14.4. Compilação condicional.

Regra 14.7 Utilize compilação condicional somente para diferentes soluções de hardware.

Justificativa Facilita a manutenção e a compreensão do código.

```
#if VEICULO == _CCT_EDT
    taskPrio = 12;
#endif // CCT_EDT
```

Regra 14.8 A diretiva #line não deve ser utilizada.

Justificativa Isto leva a distorcer a informação de número de linha nos depuradores e da macro __LINE__.

```
#line 1234 ” file.c”  // ruim!
```

Guia 14.2 A diretiva **#pragma** somente deve ser usada quando absolutamente necessária.

Justificativa Diretiva não portável, não clara e freqüentemente redundante.

14.5. Nomes pré-definidos

Regra 14.9 Não modifique uma macro pré-definida (**__LINE__, __FILE__**, **__DATE__**, **__TIME__**, **__STDC__**).

Justificativa Isto leva a um comportamento indefinido.

```
#define  __STDC__  xxx                  //
                                        //
#undef   __STDC__                       // indefinido
```

15. Padrão de codificação – DELPHI

15.1. Introdução

Este documento define regras e recomendações para a codificação em linguagem Delphi. Cada regra ou recomendação é seguida de uma justificativa. Os princípios deste padrão são segurança, confiabilidade, portabilidade e facilidade de compreensão. Isto envolve a definição de uma linguagem base e em seguida restringir seu uso a um subconjunto seguro, que não permite que características da linguagem sejam interpretadas diferentemente de um compilador para outro, ou características que permitam interpretação dúbia pelo programador. Código de qualidade é portável, facilmente compreendido, claro e não ambíguo.

15.2. Objetivo

O padrão de codificação para a linguagem Delphi orienta os desenvolvedores de modo a obter os seguinte objetivos:

- Qualidade
- Implementação uniforme dos projetos de software
- Componentes de software reutilizáveis.

15.3. Escopo

Este padrão é valido para todos os desenvolvimentos de software de produto que utilizam a linguagem Delphi.

16. DELPHI - Geral

Regra 16.1 Palavra chave **goto** é proibida.

Justificativa Isto quebra o principio do desenvolvimento de software orientado a objeto.

17. DELPHI - Expressões

Expressões semânticas são versáteis em Delphi, entretanto muitas vezes ambíguas.

Regra 17.1 Expressões não devem conter múltiplos efeitos colaterais ocasionado pelo mesmo identificador sendo modificado mais de uma vez, ou devido ao mesmo identificador sendo modificado ou acessado.

Justificativa Algumas construções em Delphi o padrão não define a seqüência de avaliação onde pode ocorrer os problemas.

```
a: array[1..3] of integer = (1, 2, 3);
x := 1;
x := Pred(x) + a[x];          // Erro - 2 efeitos colaterais
                              // Predecessor de "x" é zero
                              // assim, a[x] é um valor fora do
array

Dec(x);                       // correto
if x >= Low(a) then           // "x" está dentro da faixa do
array
begin
```

```
    x = x + a[x];
end;
```

17.1. Operadores – multiplicativos (*, / %)

Regra 17.2 Tanto a operação de divisão como o resto da divisão devem ser protegidos por teste para que o operando seja sempre diferente de zero.

Justificativa Técnicas de programação defensiva como esta, reduzem o efeito de comportamentos indefinidos ou definidos pela implementação.

17.2. Operadores – Deslocamento (shl, shr)

Regra 17.3 Em uma expressão constante, o operando a direita deve ser não negativo nem deve implicar em deslocamentos extensos.

Justificativa Estas operações podem superar o tamanho da variável (integer = 32 bits).

```
a: array[1..3 shl 32] of integer;                    //
indefinido
a: array[1..3] of integer = (1,2, (3 shl 32));       //
indefinido
```

Regra 17.4 O operando a esquerda de um operador do tipo deslocamento a direita não deve ser com sinal.

Justificativa Um operando com sinal a esquerda de uma operação de deslocamento a direita irá produzir um deslocamento em algumas plataformas e deslocamento lógico em outras.

```
i: integer;
j: integer;
u: WORD;

i := -1;

u = i shl 1;                 // errado, u = 65534


j = i shl 1;                 // correto, j = -2
```

17.3. Operadores Relacionais (<, >, <=, >=)

Regra 17.5 Os operandos de um operador relacional devem ambos serem utilizados com parênteses, a menos que sejam valores simples, identificadores ou chamadas de funções.

Justificativa A precedência dos operadores não é intuitiva. Parênteses extras ajudam ao leitor compreender o agrupamento dos operandos.

```
if (i <= 10)                 // ok. – comparação de valores
simples.
```

```
if (i - j > j + 10) and (u / 10 + i < j)  // errado: a
precedencia não é obvia.
```

18. DELPHI - Tipos e Dados

18.1. Declarações e Definições

Guia 18.1 Defina objetos no menor escopo possível. Exceção: prevenir operações de uso extensivo do “stack” e com propostos de depuração.

Justificativa É uma boa prática manter o escopo de objetos o menor possível por questões de manutenção de modo a evitar a alocação de espaço para objetos não desejados e chamadas desnecessárias a construtores e destrutores.

Regra 18.1 É proibido a utilização de dados globais.

Justificativa Dados globais quebram o princípio de encapsulamento e leva a poluição do espaço de variáveis globais e como conseqüência aumenta o risco e esforço da fase de manutenção.

Regra 18.2 Variáveis e objetos definidos num bloco interno de uma função devem ter nomes únicos dentro da função.

Justificativa Nem todos os compiladores tratam o escopo de variáveis. Isto também dificulta a compreensão e manutenção.

```
procedure TForm.ButtonClick(Sender: TObject);
    function Incrementa(i: integer): integer;
    begin
        Result := i + 1;     // Este "i" é local desta function
    end;
var
    i, j: integer;
begin
    i := 1;                                  // Este "i" é local
desta procedure
    j := inttostr(Incrementa(i));            // Nova variavel i
                                             que dificulta a
                                             // compreenção.
end;
```

18.2. Tipos “Real”

Regra 18.3 Variáveis do tipo "Real" não devem ser comparadas por igualdade (=) ou por não igual (<>).

Justificativa Raramente a comparação direta leva ao resultado esperado por causa do erro de arredondamento.

```
var
    i, j: extended;
begin
    i := pi;                                                    // =
3.1415926536
    j := 3.1415926536;
    if  i = j then                                              //
comparacao
    begin
        edit1.text := 'certo'
    end;
    else
    begin
        edit1.text := 'erro';   // resultado!
    end;

    i := pi;
    j := 3.1415926536;
    if  (i - j) <= 0 then       // comparacao
    begin
        edit2.text := 'certo'   // resultado OK
    end;
    else
    begin
        edit2.text := 'erro';
    end;
```

18.3. Conversão de tipos

Regra 18.4 Se for necessário conversão de tipo, use conversão de tipo explícita, use os operadores do tipo “cast” do Delphi.

Justificativa Os operadores do tipo “cast” tem um propósito especializado que é avaliado pelo compilador e é muito mais seguro, facilitando a compreensão e manutenção.

18.4. Constantes

Regra 18.5 Use **const** sempre que possível (sempre que tiver significado).

Justificativa Declarar um item como **const** permite especificar uma restrição reforçada pelo compilador. Como conseqüência se um objeto não deve ser modificado ele deve ser declarado **const.**

Regra 18.6 **consts** e **enums** devem ser declarados dentro da definição da classe.

Justificativa Isto previne a poluição do espaço global de nomes e especifica explicitamente a origem do dado.

18.4.1. Constantes “Integer” e “Float”

Regra 18.7 O código não deve conter valores constantes explícitos – chamados de números mágicos.

Justificativa A facilidade de leitura requer a definição de constantes, e dando a estes valores um nome com significado. Isto também facilita a manutenção.

```
// ruim
for i := 1 to 256 do                        // porque 256?
begin
    if Val[i] <> #$ff then                  // porque <> de #$ff
    begin
        //...
    end;
end;

// bom
const VAL_INVALID = #$ff;
const ARRAY_MIN   = 1;
const ARRAY_MAX   = 256;

for i := ARRAY_MIN to ARRAY_MAX do
begin
    if Val[i] <> VAL_INVALID then
end;
```

18.5. Ponteiros

Regra 18.8 As seguintes operações com ponteiros são proibidas: Not, And, Or, além da adição e subtração de dois ponteiros.

Justificativa Estas operações não definem a semântica do código. Exceções: subtração pode ser útil em aplicações de tempo crítico com arrays. Código especifico de hardware pode também ser necessário utilizar estas construções.

```
s: array[1..2] of string;
len: integer;
s[1] := '123';
len := integer(@(s[2]))-integer(@(s[1]))-1;              //
Proibido

if RADIO <> nil then                                     // OK
```

Regra 18.9 Ponteiros para funções / métodos são proibidos.

Justificativa Isto não ocorre naturalmente em Delphi, embora seja uma construção. São também de difícil compreensão e pode dificultar a depuração. A programação orientada a objetos prove a derivação de funções da classe base virtual de modo a se obter o mesmo resultado mas de modo mais transparente.

Guia 18.2 Evite ponteiros para ponteiros.

Justificativa São difíceis de se entender e tendem a provocar erros durante a manutenção.

18.6. Arrays

Guia 18.3 Classes contêineres devem ser usadas ao invés de arrays “built-in” (com exceção de arrays de tipos integrais).

Justificativa Arrays e polimorfismo não devem se misturar. Desde que não há verificação de limites (bound check), pode-se ter problemas quando se armazena um objeto derivado em arrays da classe base.

19. DELPHI - Instruções

19.1. Instruções com "Label"

Guia 19.1 Instruções não devem ser endereçáveis através de labels (tarjas).

Justificativa A única razão para ser utilizar um label em uma instrução é para acessá-la através de "goto". Estes pulos são proibidos.

19.2. Instruções de controle

Regra 19.1 Uma expressão de controle não deve ser uma designação.

Justificativa A separação da designação do teste em expressões de controle, evita efeitos colaterais e facilita a compreensão. Ações implícitas devem ser evitadas em prol da clareza da implementação. As versões atuais dos compiladores geram um código otimizado dispensando qualquer argumento de eficiência.

```
int i;

if i                          // errado – comparação deve ser
explícita

if i > 1                      // CORRETO
```

Regra 19.2 As instruções **if, else, for, while** devem ser seguidas de um bloco, mesmo quando houver apenas uma instrução (mesmo que nula (;)).

Justificativa Erros freqüentes ocorrem quando somente a indentação é utilizada para indicar que uma instrução pertence ao **if,** e na manutenção uma nova instrução é acrescentada *sem a adição dos marcadores de bloco begin...end* para indicação que ambas as instruções pertencem ao **if**. Esta prática também facilita a compreensão.

```
while i < j do inc(i);        // errado – sem begin end
while i < j do                // correto
begin
    inc(i);
end;

if i < j then inc(i);         // errado – sem begin end
if i < j then                 // correto
begin
    inc(i);
end;
```

19.2.1. Instruções "If"

Regra 19.3 Múltiplas construções utilizando **if**…**else if**... devem ter uma clausula **else** de modo a capturar todos os casos contrários.

Justificativa A programação defensiva requer a presença do else em construções de múltipla escolha, de modo a evitar estados não desejados. Use else vazios para tratamento de erros.

```
temp := 0;

if temp < 0 then
begin
    edit1.text := 'Menor';
end;
else if temp > 0 then
begin
    edit1.text := 'Maior';
end;
else                          // correto - else preventivo
begin
    MessageDlg('Zero', mtError, [mbOK], 0);
end;
```

Guia 19.2 Evite negações em expressões lógicas sempre que possível.

Justificativa Negações são difíceis de se avaliar e são facilmente incompreendidas.

```
if Result <> NO_SUCCESS then              // evite
begin
    MessageDlg('SUCCESS', mtOk, [mbOK], 0);
end;

if Result = SUCCESS then                                    //
melhor
begin
    MessageDlg('SUCCESS', mtOk, [mbOK], 0);
end;
```

19.2.2. Instruções “case”

Regra 19.4 Todas as instruções **case** devem conter uma clausula **else**, mesmo que esta clausula seja vazia.

Justificativa Aplica-se a mesma justificativa do item 9.2.1.

```
case TipoDado of
    X:
    begin
        Tratamento_X;
    end;
    Y:
    begin
        Tratamento_Y;
    end;
else
begin
        MessageDlg('Invalido', mtError, [mbOK], 0);
end;
```

end;

Regra 19.5 A clausula **else** deve ser a última instrução de um bloco **case**.

Justificativa Aumenta a clareza.

19.2.3. Instruções “exit”

Regra 19.6 Funções devem ter exatamente uma entrada e uma saída.

Justificativa Com o propósito de clareza, todas as funções não devem usar instruções **exit** como saída. Funções podem ter mais de uma instrução **exit** se aumentar substancialmente a clareza.

20. DELPHI - Funções e Métodos

20.1. Argumentos e Parâmetros das Funções

Regra 20.1 Não é permitido definir funções com um número não especificado de argumentos (notação com elipses).

Justificativa A função mais conhecida que utiliza um número de argumentos não especificados é a função **format**.O uso deste tipo de argumento deve ficar restrito as funções padrões de biblioteca, o que considera-se como exceção a regra.

Guia 20.1 Prefira referencia como argumento das funções. Somente se realmente for necessário trabalhar com ponteiros de um objeto permita a utilização de argumentos do tipo “pointer”.

Justificativa Utilizando referencia no lugar de ponteiros como argumentos de função, o código pode ser mais facilmente compreendido, especialmente internamente a função. Outra vantagem é que não existe referencia nula, o que assegura que exista uma instancia do objeto para ser trabalhado.

```
// O uso de ponteiros é complexo
procedure TForm1.AddOneNotEasy(aI: pointer);
var
    p: ^integer;
begin
    p := aI;
    p^ := p^ + 1;
end;

AddOneNotEasy(@(i));          // Chamada

// Melhor
procedure TForm1.AddOneEasy(var aI: integer);
begin
        aI := aI + 1;
end;

AddOneEasy(i);                // Chamada
```

20.2. “Overloading” de Função

“Overloading” de funções pode ser uma ferramenta poderosa para a criação de uma família de funções que diferem apenas pelo tipo do argumento, entretanto, muito cuidado de ser tomado pois este recurso pode causar considerável confusão.

Guia 20.2 Quando se utilizar sobrecarga de função, todas as variações devem ter a mesma semântica (devem ser utilizadas para o mesmo propósito).

Justificativa Se não for utilizado corretamente (tal como utilizar funções com mesmo nome para diferentes propósitos) pode-se chegar ao uso incorreto destas funções e gerar erros difíceis de serem localizados.

```
// uso adequado de overloading de função
function Divide(aX, aY: Real): Real; overload;
begin
    Result := aX / aY;
end;
```

```
function Divide(aX, aY: Integer): Integer; overload;
begin
    Result := aX div aY;
end;

if Divide(1.0/2)                          // Caso nao previsto
```

20.3. Parâmetros formais.

Regra 20.2 Os nomes dos argumentos formais da função tem que ser definidos e devem ser os mesmos na declaração da função e na definição da função.

Justificativa Prover nomes significativos para os argumentos da função faz parte da documentação da função. O nome do argumento pode clarear como o argumento é utilizado, reduzindo a necessidade de inclusão de comentários. Também é mais fácil se referir a um argumento na documentação de uma classe se este tem um nome.

```
Function Teste(X: Integer);  // Ruim

Function TestaTrasnmissao(aPortaNumero: Integer);         // OK

Function TestaTrasnmissao(aPortaNumero: Integer);         // OK
begin
    //
end;
```

21. DELPHI - Classes

As Classes introduzem o paradigma da orientação a objeto. São objetos ou dados do tipo abstrato. Elas não só contém dados mas também código procedural. Elas encapsulam grupos de dados e os métodos para trabalhar com os dados. Pela derivação ou herança certos aspectos da classe base, o chamado polimorfismo é introduzido. Trabalhar com estas características complexas da linguagem não é intuitivo e as regras a seguir serão um caminho mais seguro.

21.1. Direitos de Acesso

Guia 21.1 Os dados da classe devem ser do tipo “**private**”, sempre que possível.

Justificativa Do contrário as seguintes desvantagens podem ocorrer:

5. Dados do tipo **“public**” representam uma quebra da orientação a objeto visto que o principio do encapsulamento é violado. Este tipo de dado pode ser manipulado fora da classe e como conseqüência a integridade do objeto depende de eventos externos levando a uma implementação complexa. Dados do tipo **private** somente podem ser manipulados por métodos pertencentes a própria classe e como conseqüência mantém um encapsulamento consistente.
6. A depuração de classes com membros de dados **public** é muito difícil porque o provável local do erro pode ser em qualquer lugar da aplicação e não é restrito a sua própria classe.
7. Evitando o uso de dados do tipo **public** a implementação interna da classe é totalmente oculta dos usuários. A implementação da classe pode eventualmente ser totalmente modificada sem afetar o código dos usuários desta classe.

8. Dados do tipo **protected** devem ser evitados porque eles podem ser manipulados por classes derivadas, causando problemas análogos aos problemas do uso de dados do tipo **private**, porém com menores implicações.

Regra 21.1 Siga a regra de definição de Classe (vide Anexo B). Defina consts, enums, variáveis e métodos e para cada seção use primeiro **public**, então **protected** e por último **private**.

Justificativa Padrão de documentação.

```
type
    Priority  = class
    public
       function getPriority: integer;
    protected
       mPriority: integer;
    private
       mOldPriority :integer;
       procedure setOldPriority;
    end;
```

Guia 21.2 O uso de estruturas (**record**) deve ser evitado**.**

Justificativa Estruturas por definição contém somente dados públicos e não seguem os princípios básicos da orientação a objeto.

```
TVarMetsman = record
        Velocity,
        Direction,
        Temperature,
        Pressure: real;
end;
```

21.2. Funções Amigas

Operações sobre um objeto são providas por uma coleção de classes e funções. A função amiga (**friend**) é uma função não membro da classe, que tem acesso a membros não públicos da classe. Um **friend** oferece um modo ordenado de contornar o encapsulamento dos dados da classe, sem quebrar totalmente este encapsulamento. A utilização de uma classe amiga pode ser vantajoso pois prove funções que requerem dados que normalmente não estariam disponíveis.
Em Delphi uma função amiga é aquela definida no mesmo arquivo.

Guia 21.3 Definir cada classe em arquivos separados.

21.3. Construtores e Destrutores

Guia 21.4 Dados membros de uma classe devem ser inicializados utilizando-se uma lista de inicialização que irá inicializar os dados na ordem que estes são declarados.

Justificativa A lista de inicialização é usualmente o modo mais eficaz de se inicializar dados. Manter uma ordem de declaração explicita documenta a ordem que o compilador utilizará, o que previne desenvolvedores inexperientes de tentar ou introduzir dependências entre os dados.

Regra 21.2 Uma classe que utilize "**new**" para criar instancias gerenciadas pela classe, deve definir um construtor do tipo "**copy**" e designar um operador. A classe que não necessite prover esta semântica deve declara-la como "**private**" de modo a prevenir seu uso.

Justificativa Um construtor "**copy**" é recomendado para se evitar surpresas quando um objeto é inicializado utilizando um objeto do mesmo tipo. Se um objeto gerencia a alocação e desalocação do objeto no "**heap**" (o gerenciador do objeto tem um ponteiro do objeto a ser criado pelo construtor da classe), somente o ponteiro será copiado. Isto leva a duas chamadas do destrutor para o mesmo objeto (no "**heap**"), provavelmente resultando num "**run-time error**". As operações de designação não são herdadas como outros operadores, causando um situação muito similar. O compilador irá automaticamente definir uma designação chamando uma designação para cada dado membro o que leva a ponteiros do mesmo endereço físico. Isto novamente leva a mais de uma desalocação se objetos são destruídos ou dereferenciado de memória desalocada e como conseqüência causa um "**runtime errors**". Se uma copia ou designação de tal classe não faz sentido (por exemplo existe sempre somente uma instancia deste objeto em toda a aplicação) é razoável declarar estes operadores como "**private**". Isto previne que alguém posteriormente tente chamar um construtor "**copy**" gerado pelo compilador ou operador de designação provocando um erro de compilação.

Guia 21.5 Evite utilizar objetos globais em construtores e destrutores.

Justificativa No processo de inicialização de objetos estáticos, não existe garantia da ordem de inicialização, que podem estar definidos em varias unidades de compilação.

21.4. Herança

Guia 21.6 Para que as classes derivadas forneça acesso aos dados membro da classe declarando funções de acesso "**protected**".

Justificativa Uma classe derivada freqüentemente requer acesso a dados membro da classe base com a finalidade de criar funções. A vantagem em se utilizar funções "**protected**" é que os nomes dos dados da classe base não são visíveis nas classes derivadas, protegendo-os contra mudanças acidentais.

Nota Assume-se que o desenvolvedor que usa herança, sabe o suficiente a respeito da classe base de modo a estar apto a utilizar os dados "**private**" corretamente.

Guia 21.7 Evite herança múltipla. Herança com mistura de classes (interfaces) com somente métodos virtuais puros pode ser utilizado.

Justificativa Herança múltipla cria dependências de classes que causam problemas no projeto. Somente em casos excepcionais a herança múltipla pode ser uma solução viável, mas considerando-se as dificuldade de implementação e compreensão, deve-se evitar.

Note: A derivação de uma classe a partir de mais de uma classe base é denominada herança múltipla.

22. DELPHI - "Exceptions"

Regra 22.1 Construtores devem ser analisados em função do aspecto segurança quando da introdução das exceções.

Justificativa Alocações de memória não planejadas em construtores podem levar a "memory leaks" em caso de ocorrência de exceções.

23. DELPHI - Memória Dinâmica

O conceito de gerenciamento de memória dinâmica em Delphi difere do conceito em C. Objetos gerenciam o armazenamento de seus dados pela alocação e desalocação de memória com os operadores 'Create' e 'FreeAndNil', que estes invocam nos seus construtores e destrutores. Este encapsulamento libera o usuário da classe das preocupações com alocação de memória, visto que esta (alocação) já foi feita na classe (ou pelo menos deveria estar). Este capitulo menciona algumas armadilhas deste contexto.

Guia 23.1 Quem quer que aloque memória é responsável por sua liberação.

Justificativa Uma grande vantagem do Delphi é a possibilidade de encapsular o gerenciamento de memória no objeto. Isto significa que a alocação deve ocorrer no construtor da classe e a desalocação seja feita no destrutor. Deste modo os recursos (memória) serão liberados depois que o tempo de vida do objeto expire.

Regra 23.2 Sempre designe um valor "nil" para um ponteiro que aponte para memória liberada, use FreeAndNil ao invés de Free.

Justificativa Evita-se que se acesse memória liberada. Isto pode criar um problema de solução complexa quando existem diversos ponteiros apontando para a mesma área de memória.

24. DELPHI - Pré-processamento

O pré-processamento manipula as substituições de texto. É o mesmo pré-processamento conhecido do C. Entre as tarefas típicas estão suprimir comentários, substituição de constantes definidas com a diretiva #define e seleção de texto pela avaliação das diretivas **#if**, **#else, #endif**. Não existe verificação de tipo. A implementação de construções complexas no pré-processamento leva a erros de difícil compreensão.

24.1. Diretivas

Guia 24.1 Não é recomendado o uso de diretivas de pré-processamento.

Justificativa O uso de diretivas de pré-processamento faz o processo de geração do código complexo, dificulta a compreensão e manutenção.

24.2. Inclusão de arquivos

Regra 24.3 Não utilize caminhos (paths) absolutos ou relativos.

Justificativa Um código portável deve ser independente do ambiente em que é escrito. Paths explícitos levam também a problemas de manutenção e no processo de construção com um todo.

```
// Não utilize caminhos absolutos ou relativos.
OpenFile('../include/sys/socket.dat');

// Ruim!
OpenFile('c:\msc7\include\sys\socket.dat');
```

24.3. Compilação condicional

Regra 24.4 Utilize compilação condicional somente para diferentes soluções de hardware.

Justificativa Facilita a manutenção e a compreensão do código.

```
{$DEFINE VCC}

{$IFDEF VCC}
        taskPrio
{$ENDIF}
```