Delphi

Capítulo destinado aos padrões como nomenclatura, indentação entre outros, usados em Delphi.

Nomenclatura Geral

Nomenclatura

Componentes

Os componentes possuem seus prefixos utilizando letras que remetem ao nome completo da classe do componente. Componentes herdados dos componentes listados e sem conexão ao banco de dados devem possuir o mesmo prefixo do componente principal, com exceção dos componentes visuais com conexão ao banco de dados, que utilizam o prefixo “DB“ seguido do padrão identificado logo abaixo. Exemplo: DBED_Nome é um componente da classe TDBEdit, responsável pela representação de um nome.

Seguem alguns exemplos de prefixos padrões utilizados:

ED_: TEdit;	TS_: TTabSheet;
PN_: TPanel;	RB_: TRadioButton;
GB_: TGroupBox;	OD_: TOpenDialog;
LB_: TLabel;	SD_: TSaveDialog;
MM_: TMemo;	Q_: TQuery, TFDQuery;
BT_: TButton (Botões num geral);	DS_: TDataSource;
CK_: TCheckBox;	CDS_: TClientDataSet;
CB_: TComboBox;	DSP_: TDataSetProvider;
LT_: TListBox;	T_: TTable;
RG_: TRadioGroup;	FR_: TFrxReport;
PC_: TPageControl;	FDS_: TFrxDBDataSet;
TB_: TToolBar;	RV_: TRvProject;
GD_: TGrid;	DSC_: TRvDataSetConnection;

MI_: TMenuItem;	SH_: TShape
IM_: TImage	TM_: TTimer

Exemplo de componentes visuais com conexão ao banco de dados:

DBED_: TDBEdit;	DBGD_: TDBGrid;
DBCK_: TDBCheckBox;	DBRG_: TDRadioGroup;

Variáveis

O nome da variável deve descrever de maneira clara e sem abreviações a função da variável. Utilizar a notação PascalCase para o nome da variável.
As variáveis podem ter ainda um prefixo para adicionar mais informações sobre sua origem. Para **variáveis locais** utiliza-se o prefixo **‘L’**, para **argumentos (Parâmetros) de métodos** utiliza-se o **'A’**. Para **atributos (Field)** de classes utiliza-se o **‘F’**.

```
type
  TPessoa = class
    protected
      FNome: String; // Campo nome da classe Pessoa. Prefixo F
    public
      constructor Create(ANome: String); // Argumento do método. Prefixo A.
  end;

implementation

constructor TPessoa.Create(ANome: String);
var
  LNome: String; // Variável local. Prefixo L
begin

end;
```

Constantes

Para nomes de constantes, o padrão utilizado é:

```
const
  NOME_DA_CONSTANTE_1 = 0;
  NOME_DA_CONSTANTE_2 = False;
```

```
  NOME_DA_CONSTANTE_3 = ‘Teste’;
```

O tipo da constante é deduzido através do valor dessa constante no momento da declaração. Porém, é possível declarar explicitamente um tipo para as constantes, e em alguns casos, como o de arrays, é necessário que esse tipo seja declarado.

```
const
  NOME_DA_CONSTANTE_4: Double = 10;
  NOME_DA_CONSTANTE_5: array [1..2] of String = (‘a’, ‘b’);
```

Métodos

Assim como o nome das variáveis, o nome do método deve seguir o mesmo modelo, com nomes que descrevem a função do método, utilizando também a notação PascalCase.
Ao editar um método, é importante manter sua função original, caso sua funcionalidade seja alterada, deve-se refatorar o seu nome para que permaneça fiel ao seu comportamento.

Exemplo 1: Criar um método que faça a confirmação de um rotina de gravação. Sem retorno e sem argumentos.

```
procedure ConfirmarGravacao();
```

Exemplo 2: Criar um método que faça o cálculo da área de um retângulo. Seu retorno será um inteiro, esse método terá ainda dois argumentos, altura e largura.

```
function CalcularArea(AAltura, ALargura: Integer): Integer;
```

Units

As units padrões utilizadas anteriormente para as entidades eram nomeadas acrescentando 1, 2, 3 ou 4. Para facilitar o entendimento na leitura do nome da unit, elas passam a ser utilizadas no singular, seguido do nome de sua funcionalidade. Por exemplo, a entidade “Empresa” passa a ser utilizada como:
● Empresas1: Empresa.Manutencao;
● Empresas2: Empresa.Cadastro;
● Empresas3: Empresa.Pesquisa;
● Empresas4: Empresa.DataModule;
Para units que tenham outra função que não se enquadrem nessas descritas anteriormente, o padrão é utilizar o nome da entidade seguido da descrição de sua função.
Exemplo: Empresa.Copiar.

Classes

O padrão Delphi para nome de classes aconselha que ela inicie com o prefixo T (*Type*). É aconselhável que o nome dessas classes sejam descritivos quanto à sua entidade, função e/ou

tipo. Por exemplo, uma classe que define a entidade produtos poderia ter o nome *TProdutoModelo*, enquanto o formulário para importação desses produtos poderia ser *TProdutoImportacaoFormulario.*

Espaçamento

Regras para Formatação Horizontal

Tabulação: Utilizar 2 caracteres (espaços) ao invés da tabulação.

Para manter a estrutura do código de modo que sua leitura seja fácil, o número máximo de caracteres por linha é **130**. Caso a expressão ultrapasse esse comprimento, a linha deve ser quebrada segundo essas regras:

1. A linha tem comparações? `('a = b', 'a <> b', 'a <= b', 'a >= b', 'a < b' ou 'a > b')`
    1. Envolver cada comparação com parênteses.
2. A linha contem negação? `('not')`
    1. Envolver cada negação com um parênteses.
3. A linha contém operadores lógicos? `('and' e 'or')`
    1. Envolver cada sequência de `'and'` com um parênteses;
    2. Envolver cada sequência de `'or'` com um parênteses.
4. A linha é uma atribuição? `(':=')`
    1. Adicionar uma quebra de linha após o sinal de atribuição `(':=')`, alinhar a nova linha com a linha original e indentar em um nível (2 espaços);
    2. Caso o código resultante não tenha ultrapassado a linha das 130 colunas, processo pode ser interrompido;
    3. Caso a atribuição seja uma String, ela pode ser quebrada em uma ou mais concatenações. Essas podem ser quebradas e alinhadas com a linha original, adicionando um nível de indentação;
    4. Caso a atribuição seja uma comparação (Não considerar comparações dentro de métodos aninhados), adicionar uma quebra de linha após o sinal de comparação `('=', '<>', '<=', '>=', '<' ou '>')`, e alinhar a nova linha com o primeiro operando.
5. A linha tem um 'if'?
    1. Caso a condição do if não tenha um parênteses envolvendo a condição inteira, envolvê-la com um;
    2. Adicionar uma quebra de linha após cada operador `'and'` ou `'or'`;
    3. Após cada quebra de linha, alinhar a nova linha com o parênteses onde ela está contida e adicionar um espaço;
    4. Caso o código resultante não tenha ultrapassado a linha das 1 colunas, processo pode ser interrompido;
    5. Caso alguma das condições do if tenha uma comparação (Não considerar comparações dentro de métodos aninhados), adicionar uma quebra de linha após o sinal de comparação `('=', '<>', '<=', '>=', '<'` ou `'>')`, e alinhar a nova linha com o primeiro operando.
    6. Caso o código resultante não tenha ultrapassado a linha das 130 colunas, processo pode ser interrompido.

6. A linha contém métodos aninhados?
    1. Aplicar essa regra do métodos mais externo para o mais interno;
    2. Adicionar uma quebra de linha exatamente antes do início do método aninhado;
    3. Alinhar a nova linha com o método onde ela está contida (Caso o método tenha uma negação (not), alinhar com o início dessa negação), adicionar ainda um nível de indentação (2 espaços);
    4. Caso o código resultante não tenha ultrapassado a linha das 130 colunas, processo pode ser interrompido.
7. A linha contém parâmetros?
    1. Ao realizar a quebra de linha de um parâmetro, ele deve estar indentado um nível (2 espaços) em relação ao nível do método proprietário do parâmetro (caso o método proprietário tenha uma negação (not), alinhar com o início dessa negação);
    2. Caso o código resultante não tenha ultrapassado a linha das 130 colunas, processo pode ser interrompido.
8. Após todas essas etapas a linha ainda ultrapassa a marca de 130 colunas?
    1. Caso ainda assim, a linha ultrapasse a marca de 130 colunas então deve-se verificar a existência de pontos na expressão (“.”), sendo que, ao realizar esse tipo de quebra de linha, o ponto deve ficar com a expressão na linha inferior, indentado em um nível em relação ao início da expressão.

Observação: Algumas vezes, quando uma linha ultrapassa as 130 colunas, é um sinal de que esse trecho pode ser refatorado. O programador pode então refatorar, se julgar que é necessário ou melhor.

Declaração de variáveis

Correto

```
var
  LTeste: Integer;
```

Incorreto

```
var
  LTeste : Integer;
  LTeste :Integer;
```

Atribuições

Correto

```
LTeste := 15;
```

Incorreto

```
LTeste:=15;
LTeste:= 15;
LTeste :=15;
```

Métodos

Correto

```
procedure Ex(AParametro: Integer);
procedure Ex(AParametro1, AParametro2: Integer);
procedure Ex(AParametro: Integer; AParametro2: String);
```

Incorreto

```
procedure Ex (AParametro1: Integer);
procedure Ex( AParametro: Integer);
procedure Ex(AParametro: Integer );
procedure Ex(AParametro1,AParametro2: Integer);
procedure Ex(AParametro1 , AParametro2: Integer);
procedure Ex(AParametro: Integer;AParametro2: String);
procedure Ex(AParametro: Integer ; AParametro2: String);
```

Matrizes

Correto

```
LTeste := LMatriz[0];
```

Incorreto

```
LTeste := LMatriz[ 0];
LTeste := LMatriz[0 ];
LTeste := LMatriz[ 0 ];
```

Operadores binários

Correto

```
LTeste := 1 + 1;
```

Incorreto

```
LTeste := 1+1;
LTeste := 1 +1;
LTeste := 1+ 1;
```

Operadores unários

Correto

```
LTeste := -1;
```

Incorreto

```
LTeste := - 1;
LTeste :=-1;
```

Subrotinas

Correto

```
function MeuMetodo: String;

    procedure SubMetodo;
    begin
      //Código SubMetodo
    end;

begin
  //Código MeuMetodo
end;
```

Incorreto

```
function MeuMetodo: String;

procedure SubMetodo;
begin
  //Código SubMetodo
end;
begin
  //Código MeuMetodo
end;
```

```
function MeuMetodo: String;

  procedure SubMetodo;
  begin
    //Código SubMetodo
  end;

begin
  //Código MeuMetodo
end;
```

Uses

> **Dica:** Para identificar se uma unit deve ser declarada na uses superior ou inferior, realize o seguinte teste: Copie a unit para a uses inferior e compile o programa, caso o processo de compilação não apresente erros, a unit deve permanecer na uses inferior. Caso contrário, mova a unit para a uses superior.

No primeiro “uses“ (logo abaixo de “interface”) declarar as units no .dfm e nas assinaturas dos métodos.

```
interface

uses
  Winapi.Windows,
  Winapi.Messages,
  System.SysUtils,
  System.Classes,
```

```
  Vcl.Graphics,
  Vcl.Controls,
  Vcl.Forms,
  Vcl.Dialogs;
```

No segundo “uses“ (logo abaixo de “implementation”) declarar as units usadas na implementação do código.

```
implementation

uses
  uExisteTabela,
  uProcuraRegistro,
  uGravaSistema;
```

A fim de evitar conflito em Merge devido a alterações em mesma linha, cada Unit deve ser declarada em linha diferente. Esse erro costuma acontecer principalmente quando há refatoração ou exclusão de algum componente visual.

> Correto

```
implementation

uses
  uExisteTabela,
  uProcuraRegistro,
  uGravaSistema;
```

> Incorreto

```
implementation

uses
  uExisteTabela, uProcuraRegistro, uGravaSistema;
```

Arrays

Ao indentar elementos de um array, eles devem seguir a lógica de ordenação de parênteses. Por não estarem em níveis diferentes de indentação, devem ser indentados da seguinte forma

>

Correto

```
TClasse.Metodo(
  Parametro1,
  Parametro2,
  [Item1, Item2, Item3
   Item4, Item5, Item6]);
```

Incorreto

```
TClasse.Metodo(
  Parametro1,
  Parametro2,
  [Item1, Item2, Item3
    Item4, Item5, Item6]);
```

```
TClasse.Metodo(
  Parametro1,
  Parametro2,
  [Item1, Item2, Item3
  Item4, Item5, Item6]);
```

IN em tratamento de condições

Correto

```
Result :=
  (Self in
    [cdfIncidenciadecisaojudicial,
     cdfIncidenciaDecisaoJudicial13Sal,
     cdfIncidenciaDecisaoJudicialAvisoPrevioIndenizado]);
```

Incorreto

```
Result :=
  Self in [cdfIncidenciadecisaojudicial,
           cdfIncidenciaDecisaoJudicial13Sal,
```

```
        cdfIncidenciaDecisaoJudicialAvisoPrevioIndenizado];
```

```
Result := (Self in [cdfIncidenciadecisaojudicial,
                    cdfIncidenciaDecisaoJudicial13Sal,
                    cdfIncidenciaDecisaoJudicialAvisoPrevioIndenizado]);
```

Palavras reservadas

As palavras reservadas da linguagem Delphi devem ser escritas com todas as letras minúsculas, a única exceção é a palavra String, pois segue o padrão dos tipos, os quais iniciam com letra maiúscula. As palavras reservadas do Delphi são:

dispinterface	nil	threadvar
div	not	to
do	object	try
downto	of	type
dynamic	operator	unit
else	or	until
end	out	uses
except	overload	var
export	override	virtual
exports	package	while
external	pascal	write
far	private	writeonly
file	procedure	xor
finally	program	
for	property	