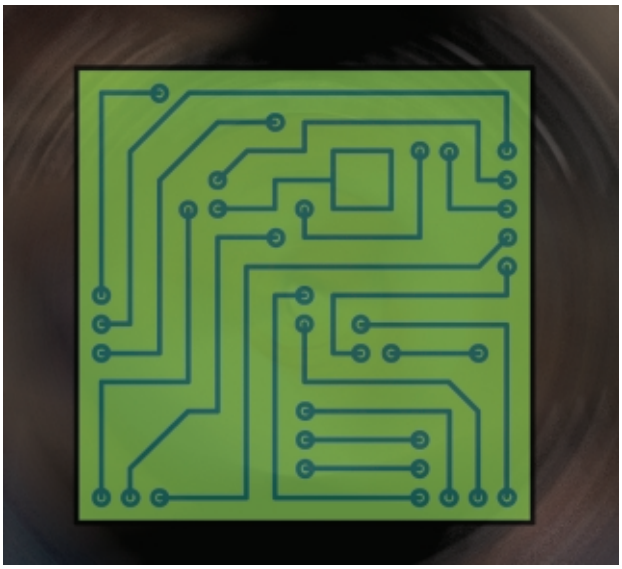
Ministério da Agricultura, Pecuária e Abastecimento

Comunicado Técnico 17

Dezembro, 2001
Campinas, SP



ISSN 1677-8464

Padronização da Codificação Programas em Borland Delphi: Experiência no Projeto SIGI

Marcos Cezar Visoli[1]
Moacir Pedroso Júnior[2]
João Francisco Gonçalves Antunes[3]
Gustavo R. Borges de Lima[4]

1. Introdução

A padronização de código fonte em um projeto de desenvolvimento de software, embora não seja fundamental para o seu sucesso, é uma prática que ajuda a fazer com que etapa de implementação seja executada de forma mais tranquila e harmoniosa.

Em *Extreme Programming* (Beck, 1999), um conjunto de práticas para desenvolvimento de software, e que são utilizados no projeto SIGI (Pedroso, 2001), a padronização de código se torna ainda mais importante, pois ajuda em muito a aplicação da prática de *pair programming*, programação em duplas, onde a leitura e intervenção de um código escrito por outro desenvolvedor tem alta frequência. Os desenvolvedores ficam mais à vontade tanto para ler como para escrever programas.

Pode-se adicionar outros pontos importantes na utilização de um padrão de codificação:

- desenvolvedores recém-incorporados em uma equipe podem obter alta produtividade rapidamente;
- pessoas com pouca prática na linguagem necessitam de um estilo pessoal e absorvendo o padrão, passam a defendê-lo e aprimorá-lo;
- certos padrões dificultam a inserção de erros de programação.

Este documento descreve o padrão, aqui muito mais como um conjunto de recomendações que devem ser seguidas na elaboração de programas em Borland Delphi, para o projeto SIGI.

Este documento não tem a intenção de definir um novo padrão ou discutir vantagens e desvantagens do padrão aqui colocado em relação a outros. O documento relata o padrão utilizado no SIGI, e que pode ser utilizado em outros projetos, e ressalta a importância de se adotar um no processo de elaboração de programas.

[1] *Bsc. em Ciência da Computação, Pesquisador da Embrapa Informática Agropecuária, Caixa Postal 6041, Barão Geraldo – 13083-970 – Campinas, SP. (E-mail: visoli@cnptia.embrapa.br)*

[2] *Ph.D. em Pesquisa Operacional, Pesquisador da Embrapa Informática Agropecuária. (E-mail: pedroso@cnptia.embrapa.br)*

[3] *Bsc. em Matemática Aplicada e Estatística, Pesquisador da Embrapa Informática Agropecuária. (E-mail: joaof@cnptia.embrapa.br)*

[4] *Consultor da Embrapa Informática Agropecuária na área de desenvolvimento do SIGI. (E-mail: gustavol@cnptia.embrapa.br)*

Embrapa

Embora seja dirigido para desenvolvedores em Borland Delphi, algumas sugestões podem auxiliar na confecção de recomendações para elaboração de programas em outras linguagens.

Também cabe ressaltar que não define, e nem é sua pretensão, o padrão para todas as situações e casos que possam ocorrer durante a etapa de escrita de código.

2. Regras Gerais para Formatação de Código Fonte

2.1. Idioma dos nomes

O idioma dos nomes de variáveis, procedimentos, classes, métodos, etc. pode interferir na legibilidade de um código fonte, já que todas as palavras reservadas de comandos, por exemplo, estão na língua inglesa. É interessante que se defina um único idioma, facilitando assim a leitura do código. Exemplos específicos podem ser vistos nas próximas seções.

2.2. Indentação

A indentação definida é de três caracteres por nível. Não se deve utilizar caracteres de tabulação nos arquivos fonte. Para isto é necessário desabilitar as opções *Use tab character* e *Optimal fill* no diálogo *Editor Options*, do menu *Ferramentas*, no Borland Delphi.

2.3. Margens

As margens devem ser fixadas em 80 caracteres. O código não deve exceder esta margem, salvo exceções como, por exemplo, terminar uma palavra. Quando possível, os comandos que ultrapassarem a margem devem ser continuados na próxima linha. O ponto de quebra geralmente é após uma vírgula ou um operador. Neste caso também é necessário indentar a linha seguinte, para manter o nível de indentação.

2.4. Comentários

Comentários normalmente são delimitados por **{** e **}**. Para comentários de uma linha, deve-se usar **//**, deixando a notação **(* *)** para a remoção temporária de código fonte.

A inserção de comentários no código fonte sempre foi uma prática altamente recomendada para os desenvolvedores de qualquer projeto. No SIGI foram estabelecidas algumas regras para o formato de comentários de cabeçalhos de *units* e de rotinas, visando a utilização de algum extrator de documentação, para a produção automática de documentação de código fonte. Optou-se pelo extrator Time2Help (Digital LogiKK, 2001), que permite a produção de documentação de código fonte em html (*HyperText Markup Language*) num estilo como JavaDoc (Sun Microsystems, 2001), amplamente conhecido no meio de desenvolvimento de sistemas. A seguir são apresentados os formatos recomendado para cada tipo de comentário.

2.4.1. Cabeçalhos

Para o cabeçalho recomenda-se o seguinte padrão:

```
{: nome do arquivo
@desc descrição do módulo ou unit
<Br>
<Br><B>Informação do CVS</B>
<Br>$Source$
<Br>$Name$
<Br>$Revision$
<Br>$Author$
<Br>$Date$
<Br><B>Revisão</B>
<Br>nome, data, descrição

@author autor do módulo ou unit.
@cat categoria do módulo ou unit.
}
```

Os nomes entre os caracteres **$** são provenientes da ferramenta de controle de versão utilizada no projeto, o CVS (Fogel, 1999), e são substituídos durante a interação do Time2Help com o controlador de versão. Também foram utilizados alguns *tags* de HTML.

2.4.2. Rotinas

Para rotinas, deve-se inserir o comentário logo acima da declaração da mesma, como no exemplo:

```
{:
   Fecha todos os arquivos da aplicação
}
procedure CloseFiles();
```

Quando a rotina possui parâmetros, pode-se produzir tabelas que os documentem, como no exemplo:

```
{:
   Remove um item, por valor, de uma lista. Caso o item
   não exista, é emitida uma mensagem de advertência.
   @param item o item que deseja-se remover
   @param lista de onde deseja-se remover o item
}
procedure RemoveItem(item : string; lista : TLista);
```

2.5. Begin ... End

O uso recomendado do par begin...end é demonstrado nos exemplos a seguir:

```
for i:=0 to 10 do begin
   ...
   ...
end;

if condicao1 then begin
   ...
end
```

```
else if condicao2 then begin
   ...
else begin
   ...
end;

while condicao do begin
   ...
end;

with nome do begin
   ...
end;
```

Observe que o *begin* sempre está na mesma linha do comando, enquanto que o *end* sempre está no mesmo nível do comando. O par *begin..end* deve ser usado mesmo quando englobe apenas um comando, como no exemplo:

```
if condicao then begin
   comando;
end;
```

Isto evitará futuros problemas, caso alguma linha seja inserida na condição e se esqueça de colocar o *begin..end*.

3. Linguagem *Object Pascal*

3.1. Parênteses

Não deve existir espaço em branco entre o abrir parêntese e o próximo caracter, assim como não deve existir espaço em branco entre um fechar parêntese e o caracter anterior. Por exemplo:

```
Procedimento( parametro ); // incorreto
Procedimento(parametro); // correto
```

Parênteses devem ser utilizados somente quando necessário, para obter uma melhor leitura do código fonte. Por exemplo:

```
if (condicao) then begin // incorreto - não é necessário
if (condicao1) or
   (condicao2) then begin // correto
```

3.2. Palavras reservadas e palavras-chave

As palavras reservadas e palavras-chave devem sempre ser escritas em letra minúscula, como por exemplo *for*, *if*, *while*, etc.

3.3. Rotinas - procedimentos e funções

3.3.1. Nome e formato

Para uma maior legibilidade do código fonte, os nomes das rotinas devem sempre começar com letra maiúscula. Quando compostas por mais de uma palavra, cada uma deve iniciar com letra maiúscula. Exemplo:

```
procedure badexampleofprocedurename; // incorreto
procedure GoodExampleOfProcedureName; // correto
```

Os nomes devem dar um sentido para o conteúdo das rotinas. Utilizar o verbo no infinitivo é um boa opção para isto. Por exemplo:

```
procedure PrintReport;
```

Para rotinas que atribuem valores os nomes devem ser iniciados com a palavra *Set*. Por exemplo:

```
procedure SetUserName;
```

Similarmente, os nomes de rotinas que recuperam valores devem ser iniciados com a palavra *Get*. Por exemplo:

```
function GetUserName : String;
```

Aqui há o uso de termos da língua inglesa (Get e Set), que pode entrar em conflito com a língua escolhida para a codificação. Porém, considerou-se que get e set são largamente utilizados, inclusive pelos ambientes de desenvolvimento que produzem automaticamente procedimentos e funções a partir da definição de classes.

3.3.2. Parâmetros

3.3.2.1. Formatação

Os parâmetros devem sempre ser acompanhados de seu tipo. Se houver necessidade de uma nova linha, a quebra deve ser realizada no nome do parâmetro e deve haver a indentação necessária para uma melhor leitura. Exemplos:

```
procedure Test(Param1 : Integer; Param2 : string);
procedure Test(Param1 : Integer; Param2 : Integer;
               Param3 : Integer; Param4 : string);
```

3.3.2.2. Nome

O nome de um parâmetro, como todo nome, deve ser significativo. Geralmente é baseado no nome do identificador que foi enviado para a rotina. Recomenda-se que o nome seja iniciado com letra maiúscula e que seja adicionado o prefixo p, para evitar uma possível ambigüidade do nome de parâmetros e de propriedades ou nome de um campo de uma classe. Por exemplo:

```
procedure Test(pNome: Integer;pIdade: Integer);
```

3.3.2.3. Parâmetros constantes

Quando parâmetros não são modificados pela rotina, é recomendado que se utilize a determinação de que o parâmetro é constante. Isto facilita a leitura da rotina, além de, para alguns tipos de variáveis, permitir um tratamento mais eficiente do código pelo compilador.

Colisão de nomes

Quando duas *units* possuem rotinas com o mesmo nome, a rotina da *unit* declarada por último na cláusula *uses* será a rotina invocada. Para evitar esta dependência da cláusula *uses*, deve-se sempre utilizar o prefixo, que é o nome da *unit*, antes da invocação da rotina. Por exemplo:

SysUtils.FindClose(SR);
ou
Windows.FindClose(Handle);

3.4. Variáveis

3.4.1. Nome e formatação

Devem iniciar com letra minúscula, e quando são compostas por mais de uma palavra, as demais palavras devem iniciar com letra maiúscula. Por exemplo:

```
var
   numberFiles : Integer;
   item : Integer;
```

Variáveis de controle de um *loop* geralmente são identificadas por um caracter como i, j ou k, mas pode-se utilizar nomes como, por exemplo, index Table. Variáveis booleanas devem ter preferencialmente um nome que signifique verdadeiro ou falso.

3.4.2. Declaração de variáveis

As variáveis devem ser declaradas uma por linha, juntamente com seu tipo. Embora pareça repetitivo quando existem várias delas do mesmo tipo, a inserção ou remoção de uma delas é feita sem afetar a declaração de outras. Como exemplo temos:

```
var
   i : Integer;
   j : Integer;
   ended : boolean;
```

3.4.2.1. Variáveis locais

Variáveis locais usadas dentro de rotinas seguem a mesma regra de nomenclatura citada anteriormente. Variáveis temporárias devem ser nomeadas apropriadamente, com o sufixo Temp.

3.4.2.2. Uso de variáveis globais

Variáveis globais devem ser utilizadas o mínimo possível. Quando for o caso devem estar declaradas na seção de implementação de uma *unit*. Quando uma variável for utilizada por mais de uma *unit*, ela deve ser movida para uma *unit* específica de variáveis globais.

Para diferenciar variáveis globais de variáveis locais recomenda-se adicionar o prefixo g ao nome da variável. Por exemplo:

```
var
   gNumItens : Integer;
   gNome : Integer;
```

Uma outra alternativa é construir um registro com todas as variáveis globais, como em:

```
var
   global : record
      numItens : Integer;
      nome : string;
   end;
```

A utilização sempre será com o nome do registro (global, neste caso) o que identifica claramente que a variável que é utilizada é global.

3.5. Tipos

3.5.1. Nome

O nomes de tipos que são palavras reservadas deveriam ser escritas em minúsculo. Tipos do Win32 geralmente são todos em maiúsculo. Tipos de outras API devem seguir a mesma nomenclatura especificada na *unit* ou documentação correspondente. Para os outros tipos, deve-se iniciar com letra maiúscula, e quando são compostas por mais de uma palavra, cada uma deve iniciar com letra maiúscula.

Exemplos:

```
var
   texto : string; // palavra reservada
   Handle : HWND;  // tipo Win32 API
   i : Integer;    // outro tipo
```

3.5.2. Tipos ponto flutuante

Real não deveria ser utilizado, pois existe somente para prover compatibilidade com antigos código pascal.

Double deve ser sempre utilizado, inclusive por ser um padrão IEEE (Institute of Electrical and Electronics Engineers).

Extended somente deve ser utilizado quando *Double* não suporta a faixa de valores necessária. *Extended* é um tipo específico da Intel.

Single somente deve ser utilizado se o tamanho físico da variável é importante, como por exemplo, quando utiliza-se uma ou mais DLLs (Dynamic Link Library) escritas em outras linguagens.

3.5.3. Tipos enumerados

Nomes de tipos enumerados deve ser significativos para o propósito de enumeração. O nome do tipo deve ser prefixado com a letra T (*type*). A lista de identificadores deve ser prefixada com até três letras minúsculas das iniciais do tipo. Por exemplo:

```
type
   TMusicalNote = (nmDo, nmRe, nmMi, nmFa,
   nmSol, nmLa, nmSi)
```

A variável de um tipo enumerado pode conter o nome do tipo sem a letra T. Por exemplo:

```
var
   MusicalNote : TMusicalNote;
```

3.5.4. Tipos estruturados

3.5.4.1. Tipos *Array*

Os nomes para tipos *array* devem ser significativos para o seu propósito. O nome do tipo deve ser prefi-

xado com a letra T (*type*). Caso um ponteiro para o *array* seja declarado, o seu nome deve ser prefixado ainda com a letra P (*pointer*). Por exemplo:

```
type
  PAnimalArray = ^TAnimalArray;
  TAnimalArray = array [1..100] of Integer;
```

3.5.4.2. Tipos *Record*

Os nomes para tipos *record* também devem ser significativos. De forma similar ao tipos *array*, a declaração deve ser prefixada com a letra T e se um ponteiro para o tipo registro é declarado, ele deve ser prefixado com a letra P. Os elementos que compõem o *record* devem ser indentados. Por exemplo:

```
type
   PCustomer = ^TCustomer;
   TCustomer = record
      Name : string;
      Address : string;
   end;
```

3.6. Comandos

3.6.1. if

Sempre que possível colocar a maior quantidade de casos a serem tratados na seção da cláusula *then*, deixando o mínimo possível na seção da cláusula *else*.

Não utilizar muitos aninhamentos de *if*. Um número de até 5 aninhamentos é razoável. Para evitar um encadeamento de *if...else if*, quando possível utilize o comando *case*.

Quando o comando *if* possuir múltiplas condições, é recomendável ordená-las da esquerda para a direita, da mais simples para a mais complexa. Isto possibilita ao compilador utilizar a vantagem de avaliação *short-circuit Boolean* e tornar a execução mais rápida. Também é recomendável colocar cada condição em uma linha, respeitando a indentação. Por exemplo:

```
if Condicao1 or
   (Condicao2 and
       Condicao3) then begin
   ...
   ...
end;
```

3.6.2. case

Os casos num comando *case* deveriam ser ordenados pela constante, numérica ou alfabética, referenciada no comando, mas pode-se ordenar pela importância ou freqüência da ocorrência do caso.

Os comandos para cada caso devem ser simples e não ultrapassar um número de 4 ou 5 linhas. Se for necessário, pode-se colocar o código numa rotina em separado, de preferência local.

Como exemplo temos:

```
case Condicao of
   condicao1:
      begin
         ...
      end;
   condicao2:
      begin
         ...
      end;
   else
      begin
         ...
      end;
end;
```

3.6.3. while

Todo e qualquer trecho de código de preparação para o comando *while* deve ocorrer imediatamente antes do comando, sendo que outros comandos não relacionados não devem estar neste trecho.

É completamente indesejável que se utilize o procedimento *Break* para encerrar o *while*. Sempre que possível, utilize somente a condição do próprio *while*.

```
i := 0;
while (i<100) do begin
   ...
   i := i + 1;
end;
```

3.6.4. for

Comandos *for* devem ser utilizados quando um trecho de código deve ser executado por um número conhecido de iterações.

```
for i:=0 to 100 do begin
   ...
end;
```

3.6.5. repeat

Comandos *repeat* seguem de forma similar as mesmas recomendações para o comando *while*.

```
i := 0;
repeat
   ...
   i := i + 1;
until i = 100;
```

3.6.6. with

O comando *with* deve ser utilizado com muito cuidado. É recomendável não utilizar múltiplos objetos ou outro elemento no comando *with*, como no exemplo:

```
with Record1, Record2 do begin
   ...
   ...
end;
```

Isto pode confundir um desenvolvedor e levar a bugs difíceis de encontrar.

4. Tratamento de Exceções Estruturado

O tratamento de exceções deve ser utilizado com freqüência. Nos casos em que recursos são alocados é necessário utilizar um **try...finally** para assegurar a liberação adequada dos recursos.

Por exemplo:

```
Class1 := TClass1.Create;
Class2 := TClass2.Create;
try
   { trecho de código }
finally
   Class1.Free;
   Class2.Free;
end;
```

5. Classes

5.1. Nomenclatura

Os nomes para classes, como os nomes para outros elementos, também devem ser significativos para o seu propósito. O nome, em minúsculo, porém com a primeira letra em maiúsculo, deve ser prefixado com a letra T (*type*), como no exemplo:

```
type
   Tcustomer = class (TObject);
```

O nome de uma instância da classe é o próprio nome da classe, sem o prefixo T. Observe que neste caso, embora a classe seja uma variável, permaneceu com a letra inicial em maiúsculo.

```
var
   Customer : TCustomer;
```

5.2. Atributos

Nomes de atributos das classes seguem a mesma convenção para nomes de variáveis, exceto que adiciona-se o prefixo F. Todos os atributos devem ser privados (*private*). O acesso a atributos fora do escopo da classe deve ser realizado através de uma *property*.

5.3. Métodos

5.3.1. Nomenclatura

Os nomes para os métodos seguem as mesmas convenções descritas para nomes de procedimentos e funções.

5.3.2. Métodos estáticos (*static*)

Métodos estáticos devem ser utilizados quando é necessário que o método não seja redefinido por uma classe descendente.

5.3.3. Métodos virtuais/métodos dinâmicos

Métodos virtuais devem ser utilizados quando o método pode ser redefinido pelas classes descendentes. Recomenda-se que métodos dinâmicos sejam utilizados em classes que tem um grande número de classes descendentes que irão redefinir o método.

Em princípio, deve-se sempre utilizar métodos virtuais, e somente sob excepcional circunstâncias deve-se utilizar métodos dinâmicos.

5.3.4. Uso de métodos abstratos (*abstract*)

Não deve-se utilizar métodos abstratos em classes cujas instâncias serão criadas.

5.3.5. Métodos de acesso a propriedades (*properties*)

Todos os métodos de acesso devem aparecer, na declaração de uma classe, nas seções privada (*private*) ou protegida (*protected*).

Os nomes do métodos seguem as mesmas regras para os procedimentos e funções. Os métodos de leitura (*read*) deve ser prefixado com a palavra *Get*. O método de escrita (*write*) deve ser prefixado com a palavra *Set*. Por exemplo:

```
TClasse = class(TObject)
private
   FAttrib : Integer;
protected
   function GetAttrib : Integer;
   procedure SetAttrib(pValue : Integer);
public
   property campo: Integer read GetAttrib write SetAttrib;
end;
```

5.4. Propriedades (*properties*)

Propriedades que servem para acessar atributos privados terão o mesmo nome do atributo, mas sem o prefixo F. Os nomes de propriedades devem ser substantivos, pois representam dados, enquanto verbos representam os métodos ou ações.

6. Arquivos

Em geral os nomes de arquivos devem ser descritivos e por si só representar o conteúdo do arquivo. Os nomes devem iniciar com a letra minúscula e podem ser compostos por mais de uma palavra, sendo que a separação entre elas se dá através de um caracter em maiúsculo.

6.1. Arquivos de projeto (*project*)

Os nomes de arquivos de projetos possuem extensão .dpr. Exemplos: queries.dpr, tstInsert.dpr.

6.2. Arquivos de formulários (*forms*)

Os nomes de arquivos de formulários possuem extensão .dfm. Recomenda-se adicionar ao nome o prefixo ufrm para indicar que se trata de um formulário. Exemplos: ufrmHelp.dfm, ufrmInfo.dfm. No caso de formulários de módulo de dados (*data modules*) deve-se utilizar o prefixo udm, como por exemplo, udmCustomers.

6.3. Arquivos de *units*

6.3.1. Nomenclatura

A nomenclatura dos arquivos de *units* segue o que já foi descrito neste documento em termos de clareza e significado.

6.3.2. Cláusula *Uses*

A cláusula *uses* na seção de interface (*interface section*) deve conter somente os nomes das *units* que são necessárias para o código da seção de interface. Qualquer outro nome de *unit* que não é necessário, inclusive aqueles que o Delphi insere automaticamente, devem ser removidos.

A cláusula *uses* da seção de implementação (*implementation section*) contém somente as *units* que são utilizadas nesta seção. Qualquer outro nome de *unit* que não é utilizado deve ser removido.

6.3.3. Seção de interface (*interface section*)

A seção de interface deve conter somente as declarações de tipos, variáveis e procedimentos, funções que são necessárias exportar para outras *units*. Caso contrário, as declarações devem ser movidas para a seção de implementação.

6.3.4. Seção de implementação (*implementation section*)

A seção de implementação deve conter quaisquer declarações de tipos, variáveis e procedimentos e funções particulares da *unit*.

6.4. Units de formulários

Um arquivo de *unit* de um formulário deve ter o mesmo nome do arquivo de formulário correspondente. Por exemplo, para o formulário ufrmCustomer.dfm, o arquivo da *unit* chama-se ufrmCustomer.pas.

6.5. Units dos módulo de dados (*data module*)

Uma *unit* de módulo de dados tem o mesmo nome do arquivo de formulário correspondente, mas com a extensão .pas. Por exemplo: udmCustomer.pas, udmSupplier.pas.

6.6. *Units* de propósito geral

Geralmente são *units* utilizadas por outras *units*, com definições de informações globais, rotinas utilitárias, etc. como por exemplo customerGlobals.pas, fileUtilities.pas.

7. Formulários (*Forms*) e Módulo de Dados (*Data Modules)*

7.1. Formulários

7.1.1. Nomenclatura

Os nomes dos tipos de formulários devem ser prefixados com a letra T (*type*). Após o nome é adicionada a palavra *Form*. Este nome é produzido automaticamente quando um formulário é criado no Borland Delphi. Como exemplo temos:

```
TCustomerForm = class (TForm)
TMainForm = class (TForm)
```

Os nomes das instâncias são idênticos aos do tipo, mas sem o prefixo T. Por exemplo:

```
CustomerForm : TCustomerForm;
MainForm : TMainForm;
```

7.1.2. Criação automática de formulários

A criação automática é realizada somente com o formulário principal, a menos que seja necessário. Todos os outros formulários devem ser removidos da lista de criação automática no diálogo *Project Options*.

A criação e destruição de formulários pode ser realizada através de uma rotina exportada por uma *unit* que contém o formulário. Esta rotina garante a criação, apresentação e destruição do formulário.

7.2. Módulo de dados

Os nomes dos tipos de módulo de dados devem ser prefixados com a letra T (*type*) mais dm (*Data Module*). Este nome também produzido automaticamente quando um formulário é criado no Delphi. Como exemplo temos:

```
TdmCustomer = class (TDataModule)
```

Os nomes das instâncias devem ser idênticos aos do tipo, mas sem o prefixo T. Por exemplo:

```
dmCustomer : TdmCustomer;
dmOrder : TdmOrder;
```

8. Conclusão

A utilização do padrão de codificação na elaboração de programas tem-se apresentado como um fator importante para a compreensão dos mesmos. O desconforto e uma possível reação de cunho negativo provocado quando um desenvolvedor lê o código elaborado por outro profissional é muito baixo e isto facilita em muito a aplicação de algumas práticas de *Extreme Programming*, como *pair programming*.

Algumas reações dos desenvolvedores podem surgir quando se aplica quaisquer padrões, e aqui caso de um padrão de codificação não é diferente:

- o padrão normalmente está errado porque foi feito por alguém que não conhece a linguagem;
- o padrão está errado porque não segue o que eu sei fazer;
- o padrão reduz a criatividade;
- o padrão não é necessário quando as pessoas têm experiência e são consistentes;
- normalmente as pessoas ignoram os padrões.

Assim algumas ações são importantes na implantação de um padrão de codificação para reduzir a resistência dos desenvolvedores e deixar que a tarefa de codificação seja prazerosa e eficiente:

- discutir e tomar as decisões sobre a padronização com a equipe de desenvolvimento;
- não impor um estilo, mas controlar o ego e convencer a equipe de que o padrão é o mais adequado;
- ser flexível às possíveis mudanças sugeridas pela equipe;
- sempre lembrar que um projeto é o esforço de uma equipe, e não de apenas um indivíduo.

Em relação a aplicação no projeto SIGI das recomendações para codificação em Borland Delphi, algumas observações podem ser tomadas:

- dos desenvolvedores que contribuíram para definir as recomendações, houve uma adoção tranquila do seu uso, desde o início.
- dos desenvolvedores que começaram a participar do projeto durante seu curso, houve uma certa resistência inicial, porém dado o devido tempo, também adotaram as recomendações.

Em todo o processo de desenvolvimento foram sugeridos melhorias, inclusive a de se adotar ferramentas para formatação do código fonte. Algumas ferramentas, devidadamente configuradas, auxiliam a formatar o código fonte seguindo muitas das recomendações definidas neste documento, e podem ser utilizadas para apresentação final. Porém, o objetivo de se definir o padrão de codificação, é o de que se utilize durante a elaboração, principalmente quando esta é feita em pares.

9. Referências Bibliográficas

BECK, K. *Extreme Programming Explained*: embrace change. Boston: Addison-Wesley , 2000. 190p. (The XP series).

DIGITAL LOGIKK. *Time2Help*. Disponível em: <http://www.time2help.com>. Acesso em: ago. 2001.

FOGEL, K. *Open Source Development with CVS*. Disponível em: < http://www.gnu.org>. Acesso em: dez. 2001.

HOFFMEISTER, S. *Delphi 4 Developer's Guide Coding Standards Document*. Disponível em: < http://www.econos.de/delphi/cs.html>. Acesso em: 11 fev. 2002.

PEDROSO, M. JR. *SIGI*: Sistema de Informação Gerencial de Projetos de Pesquisa Agropecuária para o Instituto Nacional de Investigaciones Agrícolas da Venezuela. Campinas: Embrapa Informática Agropecuária, 2001. (Sistema Embrapa de Planejamento, Projeto 12.2001.950). Projeto em andamento.

SUN MICROSYSTEMS. *How to Write Doc Comments for the Javadoc Tool*. Disponível em: <http://java.sun.com/j2se/javadoc/writingdoccomments/index.html#sourcefiles>. Acesso em: 5 fev. 2002.

Comunicado Técnico, 17

MINISTÉRIO DA AGRICULTURA, PECUÁRIA E ABASTECIMENTO

GOVERNO FEDERAL
Trabalhando em todo o Brasil

Embrapa Informática Agropecuária
Área de Comunicação e Negócios
Av. Dr. André Tosello s/nº
Cidade Universitária - "Zeferino Vaz"
Barão Geraldo - Caixa Postal 6041
13083-970 - Campinas, SP
Telefone/Fax: (19) 3789-5743
E-mail: sac@cnptia.embrapa.br

1ª edição