

**DESENVOLVENDO APLICAÇÕES  
ORIENTADAS A OBJETOS COM O  
BORLAND DELPHI**

## Introdução

O Delphi é uma ferramenta **RAD (Rapid Application Development – Desenvolvimento Rápido de Aplicações)** criada pela Borland. É uma ferramenta de propósito geral, permitindo o desenvolvimento de aplicações tanto científicas como comerciais com a mesma facilidade e alto desempenho. Integra-se facilmente com a API (Application Program Interface) do Windows, permitindo a criação de programas que explorem ao máximo os seus recursos, assim como os programas escritos em linguagem C/C++.

Possui um compilador extremamente rápido, que gera executáveis nativos (em código de máquina, não interpretado), obtendo assim melhor performance e total proteção do código fonte.

O Delphi é extensível, sua IDE (Integrated Development Environment – Ambiente de Desenvolvimento Integrado) pode ser ampliada e personalizada com a adição de componentes e ferramentas criadas utilizando-se o Object Pascal, a linguagem de programação do Delphi. Neste ambiente constroem-se as janelas das aplicações de maneira visual, ou seja, arrastando e soltando componentes que irão compor a interface com o usuário.

O Object Pascal é uma poderosa linguagem Orientada a Objeto, que além de possuir as características tradicionais das mesmas como classes e objetos, também possui interfaces (semelhantes às encontradas em COM e Java), tratamento de exceção, programação multithreaded e algumas características não encontradas nem mesmo em C++, como RTTI (Runtime Type Information). Assim como o C++, o Object Pascal é uma linguagem híbrida, pois além da orientação a objeto possui também uma parte da antiga linguagem estruturada (Pascal)

Devido ao projeto inicial da arquitetura interna do Delphi e da orientação a objeto, suas características básicas mantêm-se as mesmas desde o seu lançamento em 1995 (ainda para o Windows 3.1, pois o Windows 95 ainda não havia sido lançado), o que demonstra um profundo respeito com o desenvolvedor. Isto permite que uma aplicação seja facilmente portada de uma versão anterior para uma nova, simplesmente recompilando-se o código fonte.

Obs: Embora as características, teorias e exemplos abordadas aqui sejam sobre o Delphi 6, tudo pode ser aplicado em versões anteriores e posteriores do Delphi, **excetuando-se** o caso da utilização de componentes e ferramentas introduzidos apenas nesta versão.

# CAPÍTULO 1

## ***Princípios da Programação para Windows***

Antes de começar a trabalhar com o Delphi, é importante ter algumas noções do que está envolvido na programação Windows e no Delphi em particular. Algumas coisas tornam a tarefa de programação no Windows (e ambientes baseados em eventos e interface gráfica) bem diferente de outros ambientes e das técnicas de programação estruturada normalmente ensinadas nos cursos de lógica de programação:

***Independência do Hardware:*** No Windows, o acesso aos dispositivos de hardware é feito com intermédio de *drivers* fornecidos pelo fabricante do hardware, o que evita que o programador tenha que se preocupar com detalhes específicos do hardware. Como acontecia com a programação em DOS.

***Configuração Padrão:*** O Windows armazena centralmente as configurações de formato de números, moeda, datas e horas, além da configuração de cores, livrando o programador de se preocupar com esses detalhes específicos.

***Multitarefa:*** Antigamente, no DOS (não estamos falando do Prompt do MS-DOS), um programa geralmente tomava o controle da máquina só para si, e outros programas não rodavam até que o mesmo fosse fechado. Já no Windows vários programas são executados de maneira simultânea e não há como evitar isso.

***Controle da Tela:*** No DOS geralmente um programa ocupa todo o espaço da tela, e o usuário via e interagia apenas com aquele programa. Já no Windows, todas informações mostradas e todas entradas recebidas do usuário são feitas por meio de uma *janela*, uma área separada da tela que pode ser sobreposta por outras janelas do mesmo ou de outros programas.

***Padrões de Interface:*** No Windows, todos os elementos de interface aparecem para o usuário e interagem da mesma forma. Além disso, existem *padrões* definidos pela Microsoft que são recomendados para conseguir a consistência entre aplicativos. Falaremos de alguns deles no curso, mas a melhor forma de aprendê-los é analisar os aplicativos Windows mais usados do mercado.

***Eventos e a Cooperação com o Sistema:*** Num programa criado para DOS (como os programas escritos em Clipper) ele é responsável pelo fluxo de processamento, temos que definir claramente não só que instruções, mas também em que ordem devem ser executadas, ou seja a execução segue uma ordem preestabelecida pelo programador, e o programa só chama o sistema operacional quando precisa de alguma coisa dele. Em Windows não é bem assim. Nosso programa não controla o fluxo de processamento, ele responde e trata *eventos* que ocorrem no sistema. Existem muitos *eventos* que podem

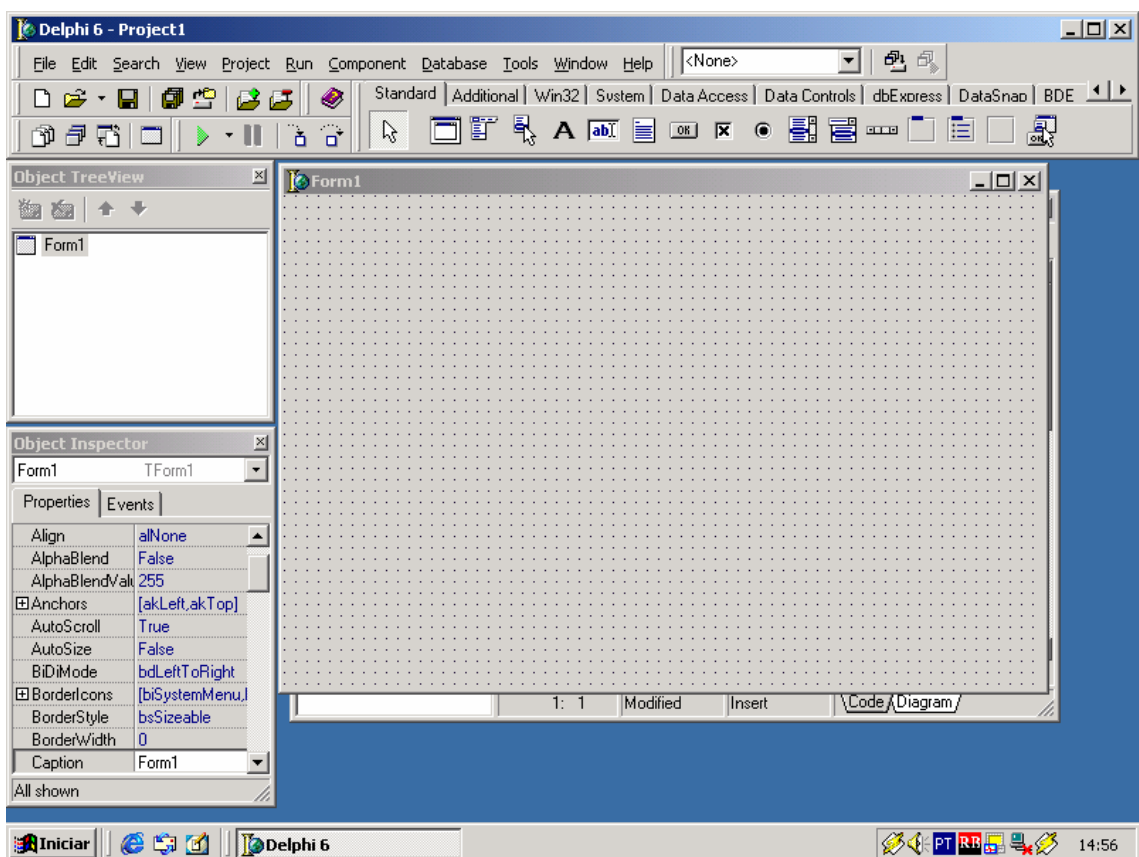
ocorrer, sendo que os principais são aqueles gerados pelo usuário através do mouse e do teclado. A coisa acontece mais ou menos assim: O usuário clica o mouse e o Windows verifica que aplicação estava debaixo do mouse no momento em que foi clicado. Em seguida ele manda uma mensagem para a aplicação informando que ocorreu um clique e as coordenadas do cursor do mouse na tela no momento do clique. A aplicação então responde à mensagem executando uma função de acordo com a posição do mouse na tela. É claro que o Delphi toma conta do serviço mais pesado e facilita muito as coisas para o programador. Detalhes como as coordenadas da tela em que ocorreu o clique, embora estejam disponíveis, dificilmente são necessários nos programas. Isso, como veremos, afeta radicalmente o estilo de programação e a forma de pensar no programa. A sequência de execução do programa depende da sequência de eventos.

## ***Conhecendo o Delphi 6***

Esta apostila se baseia na versão 6 do Borland Delphi, mas seus conceitos podem ser estendidos para qualquer versão do Delphi que possua recursos semelhantes.

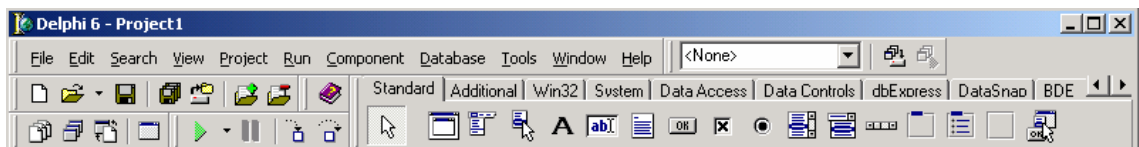
O Delphi oferece dois níveis de programação distintos. Existe o nível que o manual chama de *designer*, que utiliza os recursos de programação visual e aproveita os componentes prontos, e o nível do *component writer*, onde escrevemos os componentes para o *designer* utilizar nas aplicações. Podemos dizer que o *component writer* programa em um nível mais baixo e o *designer* em um nível mais alto. Para este curso, consideraremos apenas a programação no nível do *designer*.

Inicie o Delphi clicando no ícone Delphi 6 que normalmente se encontra no menu *Iniciar / Programas / Borland Delphi 6*.



Quando ativamos o Delphi, a tela inicial é parecida com a acima. Os itens que você está vendo formam o que chamamos de IDE, com um projeto novo aberto. Na janela superior, temos a barra do menu principal do Delphi, à esquerda a *SpeedBar*, com as opções mais comuns e à direita a paleta de componentes. Estes componentes formam a base da programação visual e é onde o *designer* vai buscar recursos para criar sua aplicação. A seguir, vamos analisar as ferramentas que compõe o ambiente de desenvolvimento e os arquivos que constituem um projeto.

## Janela Principal



A janela principal é o próprio Delphi, se a fecharmos estaremos fechando todo o Delphi. Esta janela é composta basicamente pelo menu e mais duas áreas distintas, o SpeedBar e a Component Palette - Paleta de Componentes.

## SpeedBar



SpeedBar (“Barra de Velocidade”) foi o nome dado pela Borland à barra de ferramentas com atalhos para os procedimentos comumente executados durante a fase de desenvolvimento de um projeto. São eles:

- **New.** Abre uma caixa de diálogo que permite selecionar o tipo de objeto a ser criado (novo aplicativo, formulário, DLL, relatórios, entre outros). Equivalente ao menu File | New | Other.
- **Open.** Abre uma Unit, Form, Projeto ou Package. Equivalente ao menu File | Open.
- **Save.** Salva a Unit/Form atual. Equivalente ao menu File | Save.
- **Save All.** Salva todas as Units/Forms abertos que sofreram alteração. Equivalente ao menu File | Save All ou as teclas Shift+Ctrl+S.
- **Open Project.** Abre um arquivo de projeto (.dpr – Delphi Project). Equivalente ao menu File | Open Project ou as teclas Ctrl+F11.
- **Add file to Project.** Acrescenta um arquivo já existente ao projeto atual. Equivalente ao menu Project | Add to Project ou as teclas Shift+F11.
- **Remove file to Project.** Remove um arquivo do projeto. O arquivo não será removido do disco, apenas deixará de fazer parte projeto. Equivalente ao menu Project | Remove from Project.
- **Help Contents.** Aciona o Help do Delphi. Equivalente ao menu Help | Delphi Help.
- **View Unit.** Permite escolher uma Unit do projeto para ser exibida. Equivalente ao menu View | Units ou as teclas Ctrl+F12.
- **View Form.** Permite escolher um Form do projeto para ser exibido. Equivalente ao menu View | Forms ou as teclas Shift+F12.
- **Toggle Form/Unit.** Permite alternar entre um formulário e seu respectivo código fonte. Equivalente ao menu View | Toggle Form/Unit ou a tecla de função F11.
- **New Form.** Adiciona um novo formulário ao projeto. Equivalente ao menu File | New Form.
- **Run.** Executa a aplicação, compilando-a se necessário. Equivalente ao menu Run | Run ou a tecla de função F9.
- **Pause.** Suspende a execução do programa. Equivalente ao menu Run | Pause Program.
- **Trace Into.** Executa o programa passo a passo, linha a linha, dentro da rotina que for invocado e dentro de todas as rotinas que forem

acessadas posteriormente. Equivalente ao menu Run | Trace Into ou a tecla de função F7.

**Step Over.** Semelhante ao Trace Into, porém a execução passo a passo ocorrerá somente dentro da rotina em que for invocado. Equivalente ao menu Run | Step Over ou a tecla de função F8.

## Component Palette (Paleta de Componentes)

Cada ícone na paleta refere-se a um componente que, quando colocado em um Form, executa determinada tarefa: por exemplo, um TLabel mostra um texto estático, e um TEdit é uma caixa de edição que permite mostrar e alterar dados, o TComboBox é uma caixa que permite selecionar um dentre os itens de uma lista, etc.

A paleta de componentes tem diversas guias, nas quais os componentes são agrupados por funcionalidade. Outras guias podem ser criadas com a instalação de componentes de terceiros.

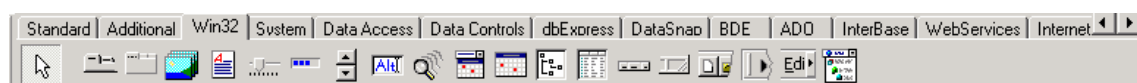
Segue abaixo a relação completa de todas as guias que compõe o Delphi 6 Enterprise:



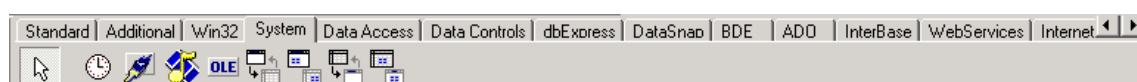
**Standard:** componentes padrão da interface do Windows, usados para barras de menu, exibição de texto, edição de texto, seleção de opções, iniciar ações de programa, exibir listas de itens etc. Geralmente são os mais usados.



**Additional:** componentes especializados que complementam os da página Standard. Contém botões com capacidades adicionais, componentes para exibição e edição de tabelas, exibição de imagens, gráficos etc.



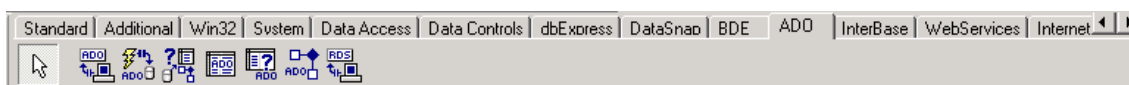
**Win32:** componentes comuns de interface que são fornecidos pelo Windows 95/NT para os programas. Contém componentes para dividir um formulário em páginas, edição de texto formatado, barras de progresso, exibição de animações, exibição de dados em árvore ou em forma de ícones, barras de status e de ferramentas etc.



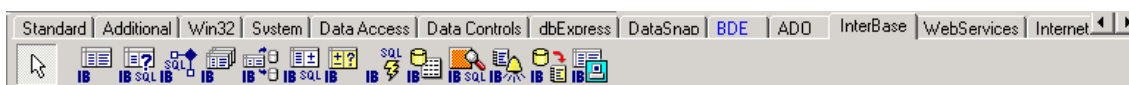




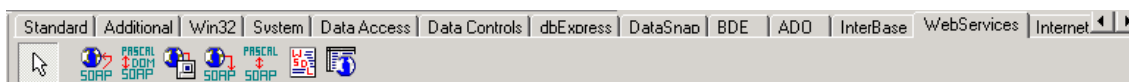




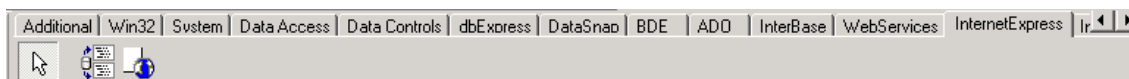
**ADO:** componentes de acesso a dados da interface dbGo (introduzida no Delphi 5 como o nome de ADO Express) através da tecnologia ADO (ActiveX Data Objects), da Microsoft. Tanto a ADO como o OLE DB (drivers de acesso) estão incluídos no MDAC (Microsoft Data Access Components) e permitem acesso a uma série de bancos de dados e à ODBC. Sua principal vantagem é estar incorporada as versões mais recentes do Windows (2000/XP e ME) não sendo necessário nenhuma instalação de engine de acesso. Também é escalável, permitindo acesso desde bases de dados desktop até aplicações multicamadas. A desvantagem é que não é portátil para outras plataformas, caso queira portar seu sistema para Linux, terá que trocar todos os componentes de acesso a dados.



**Interbase:** componentes para acesso nativo ao Interbase, através de sua API, constituindo o método de acesso mais rápido e eficiente para este banco de dados. Por não ser uma interface genérica permite utilizar todos os recursos que o Interbase disponibiliza. A desvantagem, no entanto é que ao utiliza-los perde-se a possibilidade de alterar o banco de dados sem mudar o programa, visto que os mesmos se destinam apenas ao Interbase.



**WebServices:** componentes que formam a BizSnap, uma plataforma RAD para desenvolvimento de Web services, que simplifica a integração B2B criando conexões e Web services baseados em XML/SOAP



**InternetExpress:** componentes para manipulação de XML e produção de páginas para internet.



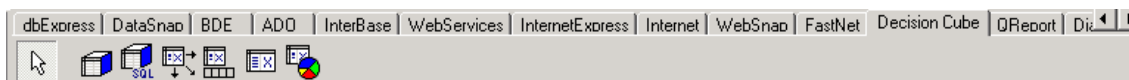
**Internet:** componentes para manipulação de Sockets, páginas e web browser.



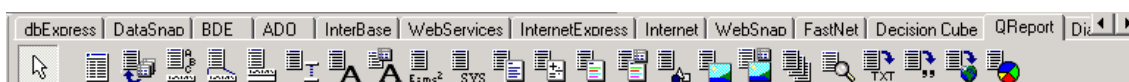
**WebSnap:** componentes para o desenvolvimento de aplicações Web que suporta os principais Servidores de Aplicações Web, inclusive Apache, Netscape e Microsoft Internet Information Services (IIS);.



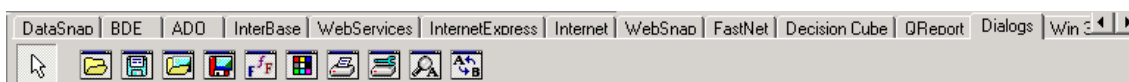
**FastNet:** componentes para manipulação de protocolos e serviços da internet como http, nntp, ftp, pop3 e smtp entre outros. (São mantidos por compatibilidade com o Delphi 5, nesta versão foram inseridos os componentes Indy com maior funcionalidade).



**Decision Cube:** componentes para tomada de decisão através da análise multidimensional de dados, com capacidades de tabulação cruzada, criação de tabelas e gráficos. Estes componentes permitem a obtenção de resultados como os obtidos por ferramentas OLAP (On-Line Analytical Processing – Processamento Analítico On-Line) utilizados para análise de Data Warehouses.



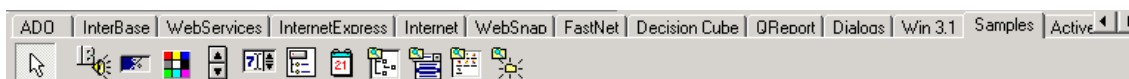
**Qreport:** QuickReport é um gerador de relatórios que acompanha o Delphi e se integra totalmente ao mesmo, sem a necessidade de run-time ou ferramentas externas como o Cristal Report, etc. Estes componentes permitem a criação de diversos tipos de relatórios de forma visual e também a criação de preview personalizado.



**Dialogs:** O Windows tem caixas de diálogo comuns, como veremos, que facilitam mostrar uma interface padrão dentro do seu programa para as tarefas comuns, como abrir e salvar arquivos, impressão, configuração de cores e fontes etc. Esta guia tem componentes que permitem utilizar essas caixas de diálogo comuns.

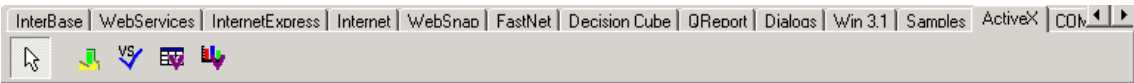


**Win 3.1:** esta guia contém controles considerados obsoletos, que estão disponíveis apenas para compatibilidade com programas antigos. Não crie programas novos que utilizem esses controles.

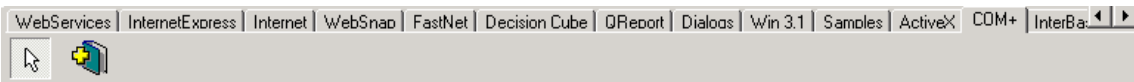


**Samples:** contém exemplos de componentes para que você possa estudá-los e aprender a criar seus próprios componentes. O código fonte desses

exemplos está no subdiretório SOURCE\SAMPLES do diretório de instalação do Delphi.



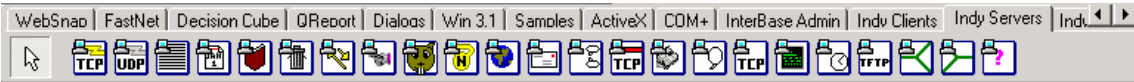
**ActiveX:** um componente ActiveX é um tipo de componente que pode ser criado em outra linguagem (como C++) e utilizado no Delphi. Esta página contém alguns exemplos de componentes ActiveX prontos para utilizar, que têm funções de gráficos, planilha, etc. O Delphi também pode criar componentes ActiveX, que podem ser utilizado em ambientes como Visual Basic e Visual FoxPro.



**COM+:** catálogo de objetos COM (Component Object Model), tecnologia desenvolvida pela Microsoft que possibilita a comunicação entre clientes e aplicações servidores. Uma interface COM é a maneira como um objeto expõe sua funcionalidade ao meio externo.



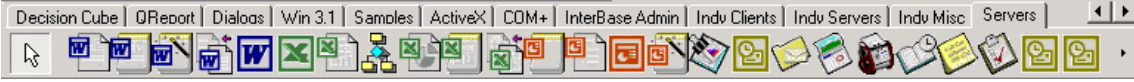
**Indy Clients:** componentes para criação de aplicativos clientes para protocolos HTTP, TCP, FTP, DNS Resolver, POP3, SMTP, TELNET, entre outros.



**Indy Servers:** componentes para criação de aplicativos servidores de HTTP, TCP, FTP, TELNET, GOPHER, IRC, entre outros.

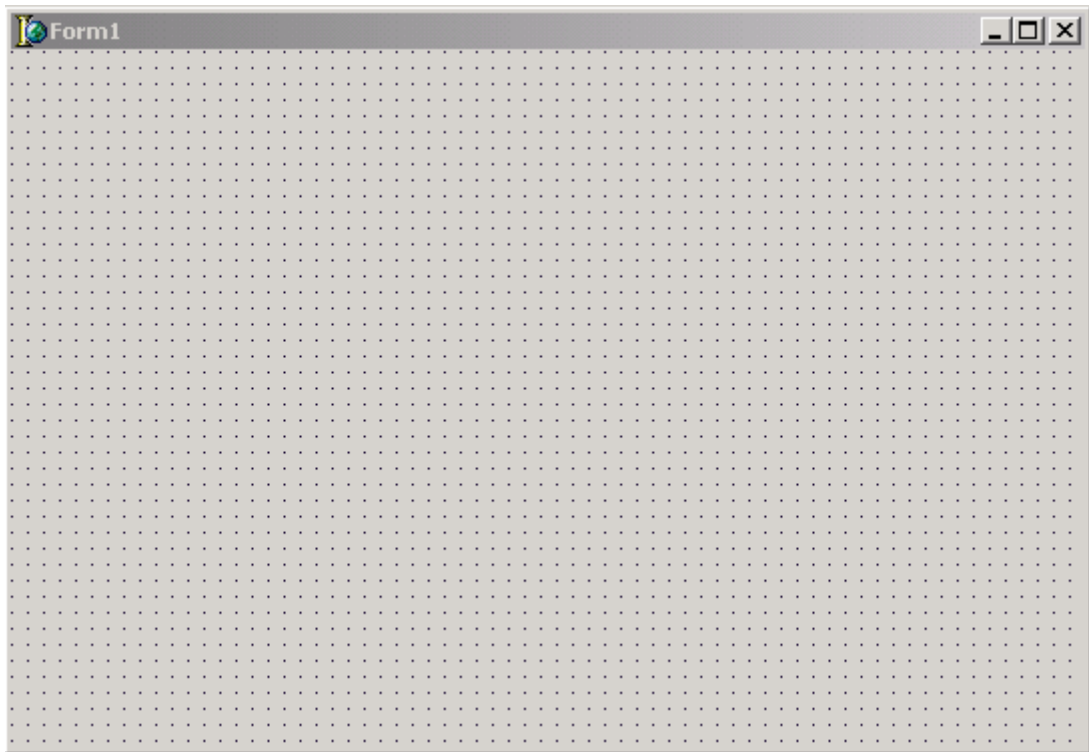


**Indy Misc:** componentes complementares aos das guias Indy Clients e Indy Servers, para criação de aplicativos clientes e servidores com acesso a internet, como clientes de ftp, irc e browsers.



**Servers:** componentes para automatização do Microsoft Office, permitindo o controle, impressão e criação de documentos destes aplicativos dentro do seu programa.

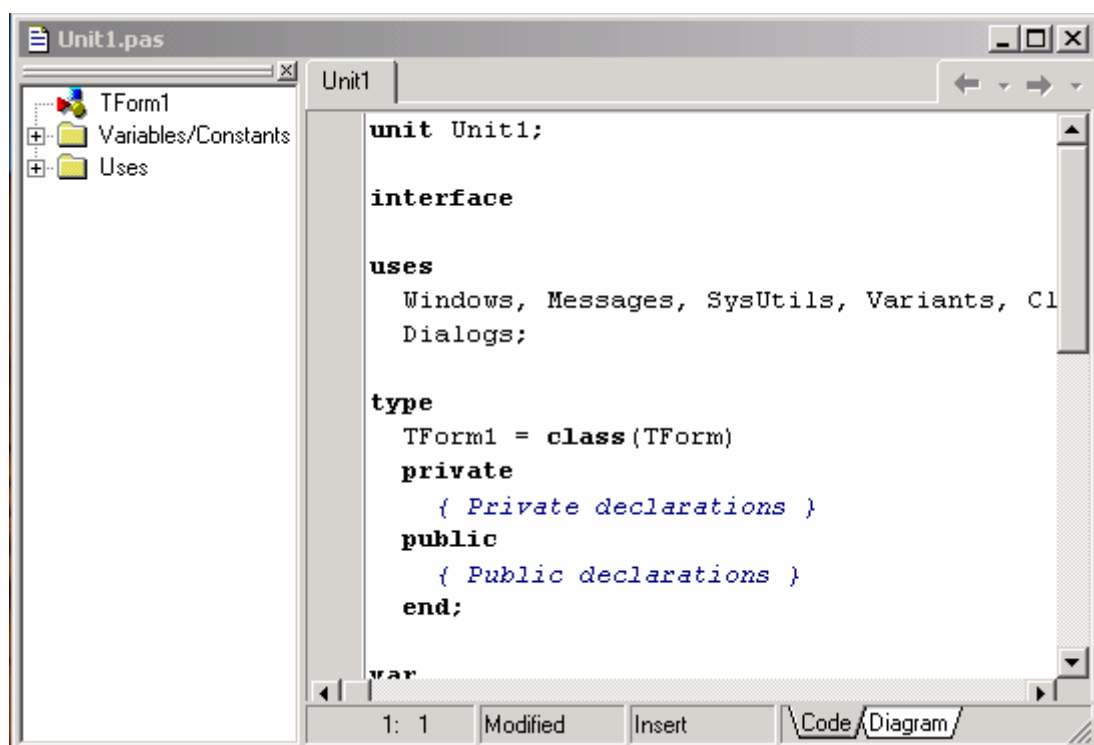
## Formulário para criação do Aplicativo



Cada aplicativo criado em Delphi é chamado de projeto e pode ser formado por um ou mais formulários (Janelas), ou ainda em casos especiais não possuírem janela alguma (Console Application).

É sobre estes formulários que serão colocados os componentes para a criação da interface do aplicativo.

Todo formulário possui um arquivo de programa-fonte correspondente, chamado **Unit**, que pode ser visualizado no editor de código (Code Editor). A janela do editor de código pode ser acessada clicando-se no botão Toggle Form/Unit da SpeedBar caso o Formulário esteja selecionado; você também pode clicar na borda que aparece logo abaixo do formulário, ou ainda pressionando-se a tecla F12 que permite alternar entre o editor de código e o formulário.

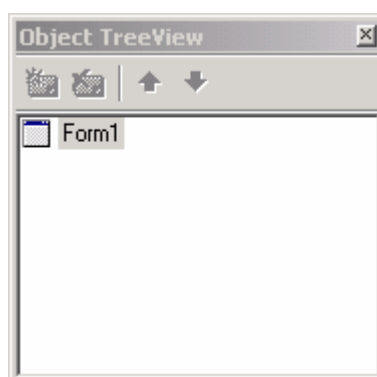


A Unit está intimamente ligada ao formulário, chamado **Form**: quando se adiciona um componente ao Form, o Delphi inclui na Unit deste Form o código referente à inclusão do mesmo, ou seja, uma mudança no lado visual resulta em uma alteração automática no código. A Borland, empresa que fez o Delphi, denominou isso de Two-Way-Tool (ferramenta de duas vias).

À esquerda do editor de código esta o Code Explorer, uma ferramenta que permite visualizar e acessar no código fonte as units, classes, variáveis, constantes e objetos (componentes) que fazem parte do Form.

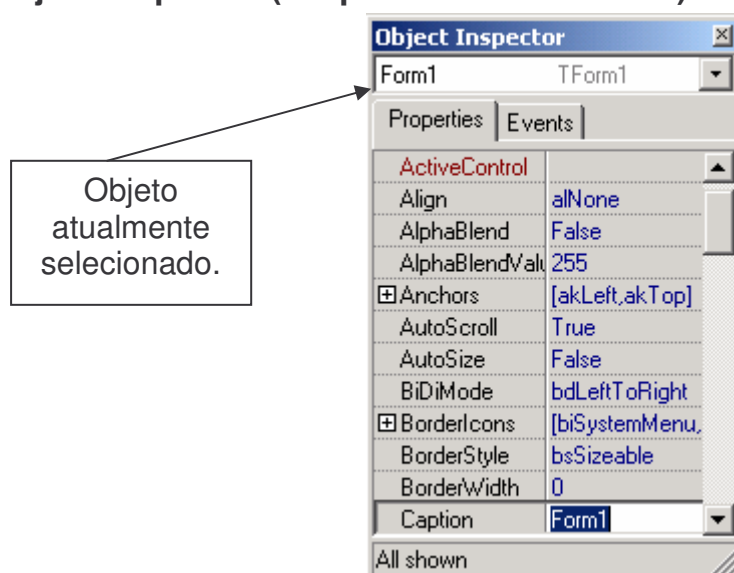
Na mesma janela do editor de código pode ser acessada a guia Diagram que permite documentar o relacionamento entre os componentes, além de modificar algumas características destas relações. Inicialmente ela está vazia, para incluir os componentes, você deve arrastá-los do Object TreeView e soltá-los sobre o diagrama.

## Object TreeView



O Delphi 6 introduziu uma nova janela, o Object TreeView, que mostra o relacionamento entre os componentes que são colocados no Form. Como veremos adiante, existem componentes que funcionam como “recipientes”, ou seja, podem conter outros componentes dentro de si. O Object TreeView permite a visualização destes relacionamentos e o acesso rápido a estes objetos.

### Object Inspector (Propriedades e Eventos)



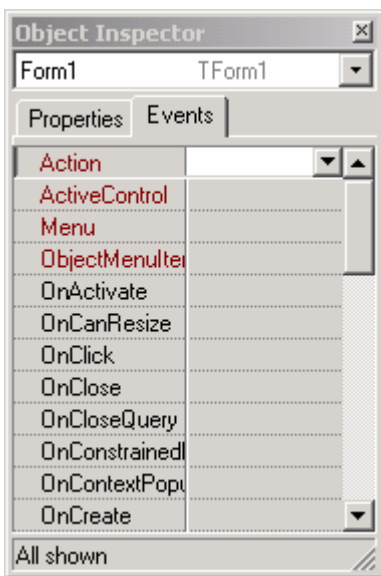
O Object Inspector é a ferramenta responsável por permitir a modificação das propriedades dos componentes/objetos de forma visual, durante o projeto (Design Time). Por enquanto, pense em propriedades como sendo as características dos componentes, tanto visuais quanto funcionais. O combobox na parte superior desta janela dá acesso a todos os objetos do Form atual e sempre exibe o nome do objeto/componente selecionado. Uma vez selecionado você pode inspecionar suas propriedades e alterá-las se desejar. A alteração das propriedades pode modificar a aparência de objetos visuais e também seu funcionamento padrão.

A alteração o valor de uma propriedade depende de seu tipo, para textos e números basta apenas digita-los no quadro correspondente (em frente ao nome da propriedade). Caso o mesmo seja formado por uma lista, será fornecido um combobox com os valores possíveis neste mesmo local.

Se existir um sinal de adição à esquerda do nome da propriedade, isto indica que a mesma possui subpropriedades, para acessá-las clique no sinal de adição que as mesmas serão exibidas. Algumas propriedades possuem um editor de propriedade, nestes casos é fornecido um botão com reticências. Clicando-se neste botão a tela de edição da propriedade deve ser exibida.

A ordem e as propriedades que devem ser exibidas no Object Inspector podem ser configuradas clicando-se com o botão direito do mouse sobre o mesmo para acionar seu menu popup. A opção *View* permite selecionar as categorias a serem exibidas, desativando uma categoria, todas as propriedades que pertencem a elas não serão mostradas. A opção *Arrange* do menu permite ordenar por categoria ou por nome.

Além da modificação das propriedades, os componentes sofrem a ação de eventos. Um evento ocorre quando o programa está sendo executado e o usuário pressiona o botão do mouse ou uma tecla, por exemplo. Você pode querer que seu programa execute uma determinada tarefa se uma dessas ações ocorrem.



Cada um dos eventos exibidos no Object Inspector para um determinado objeto é acionado quando é executada uma ação específica. Por exemplo, fechar a janela ativaria o evento OnClose da mesma. Se houver algum código associado a esse evento, ele será executado naquele momento. Assim, você pode observar que o programa no Windows é essencialmente passivo, isto é, ele espera até que algo aconteça e ele seja chamado. Se um evento é codificado, mas o mesmo não ocorre durante a execução do aplicativo seu código não é executado.

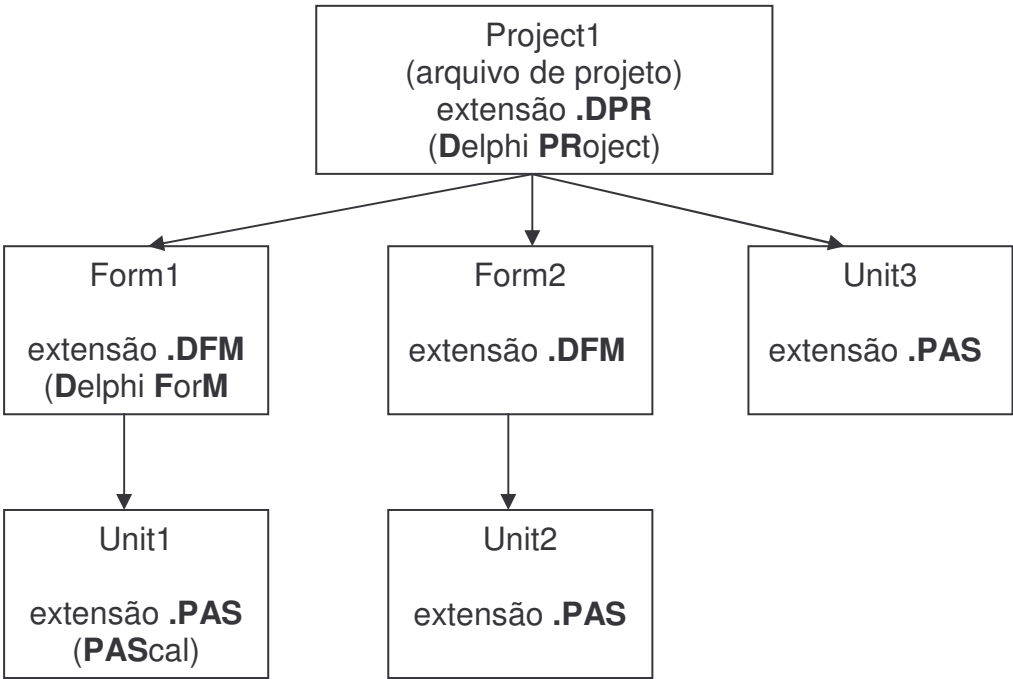


## Conhecendo a Estrutura de uma Aplicação

Uma aplicação feita em Delphi é composta de um arquivo de projeto, que gerencia quais Forms e Units compõem a aplicação. O nome dado ao arquivo do projeto, normalmente será o nome dado ao executável da aplicação quando a mesma for compilada.

Em alguns casos podemos ter uma Unit sem Form, um exemplo seria uma Unit com funções para serem utilizadas por toda a aplicação (em vários Forms), mas todo Form obrigatoriamente deve possuir sua Unit correspondente.

Vejamos um esquema gráfico de como é estruturado um projeto em Delphi:



## Estrutura da Unit de um Form

As Units do Delphi possuem uma estrutura que deve ser obedecida. Quando um Form é criado também é criada uma Unit associada ao mesmo.

A estrutura básica de uma Unit pode ser visualizada observando-se o código fonte da mesma no editor de código. Será semelhante ao exibido a seguir:

*unit Unit1;*

*interface*

*uses*

*Windows, Messages, SysUtils, Variants, Classes, Graphics, Controls, Forms,*

*Dialogs;*

```
type
  TForm1 = class(TForm)
  private
    { Private declarations }
  public
    { Public declarations }
  end;
```

```
var
  Form1: TForm1;
```

*implementation*

```
{ $R *.dfm }
```

*end.*

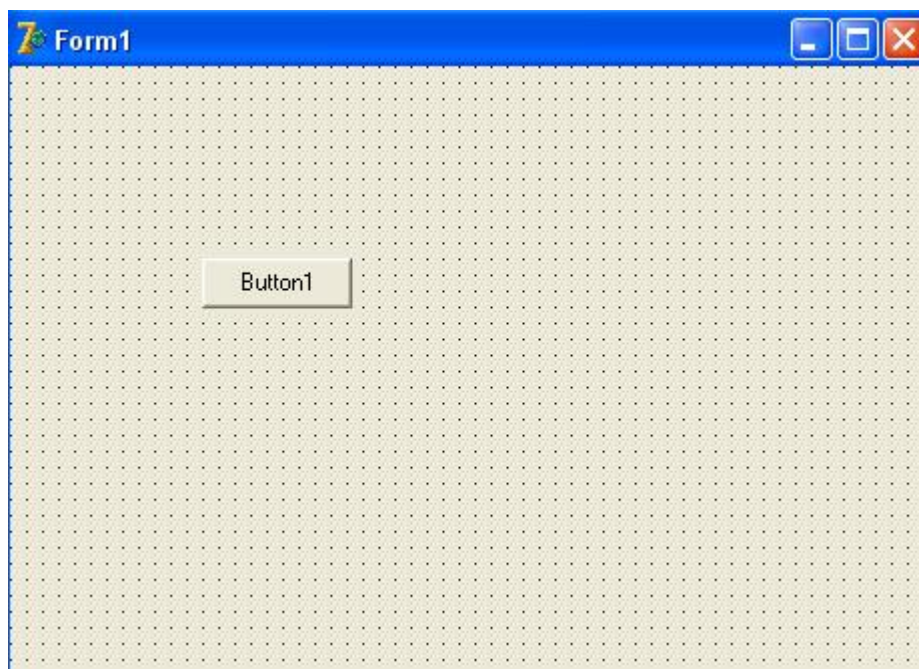
Vamos analisar o código acima:

- Na primeira linha, o nome em frente à palavra **unit**, no caso Unit1, indica o nome dado ao arquivo com a programação do formulário. Se o formulário fosse salvo com este nome ele geraria um arquivo externo com o nome de Unit1.pas e outro com o nome de Unit1.dfm. (Quando for salvar seus formulários você deve dar nomes mais significativos).
- Na linha seguinte, a palavra **interface** delimita a seção de interface na qual serão colocadas as definições de funções, procedimentos, tipos e variáveis que poderão ser vistos por outras units da aplicação.
- A cláusula **Uses** dentro da seção interface indica quais units deverão ser ligadas para poder complementar a nossa . Ao criar um Form as Units definidas no código acima são inseridas automaticamente, pois fornecem o suporte para criação do mesmo. Ao inserir componentes num Form, outras Units podem ser adicionadas a esta lista.
- A seguir temos a definição dos tipos do programa, identificada pela palavra **type**. Neste ponto temos a definição de uma classe TForm1 que é derivada da classe base TForm. Ao se acrescentar componentes no Form também será gerado no código definição correspondente aos mesmos. (O conceito de classes e objetos será explicado no Capítulo 2)
- O próximo item é a definição de variáveis e constantes globais, através da palavra reservada **var**. Neste ponto é criada uma variável com visibilidade global (pode ser vista em outras units nas quais a mesma seja incluída na cláusula uses)
- A palavra chave **implementation** delimita a segunda seção da unit, onde serão colocadas as funções e variáveis que serão acessadas apenas por ela mesma (não são visíveis em outras units).

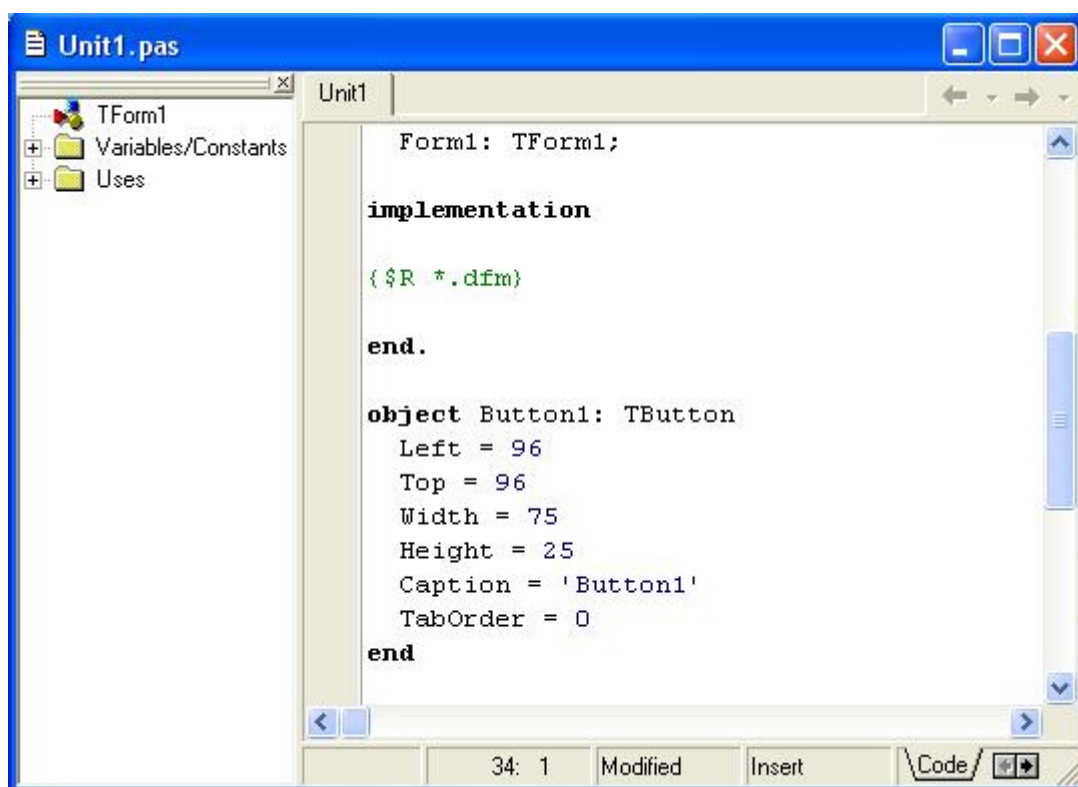
- O símbolo ***{ \$R \*.dfm }*** faz a associação da unit com seu respectivo form e não deve ser modificado. Uma unit de funções não possui esta diretiva.
- **Ao Final** da Unit, temos uma linha com **end**. Ele é o marcador de final de arquivo. Qualquer coisa colocada após esta linha será ignorada.
- Opcionalmente, uma unit pode ter ainda duas seções: **initialization** e **finalization**, com comandos que são executados quando a aplicação é iniciada ou finalizada.

## Um exemplo não muito comum mas interessante

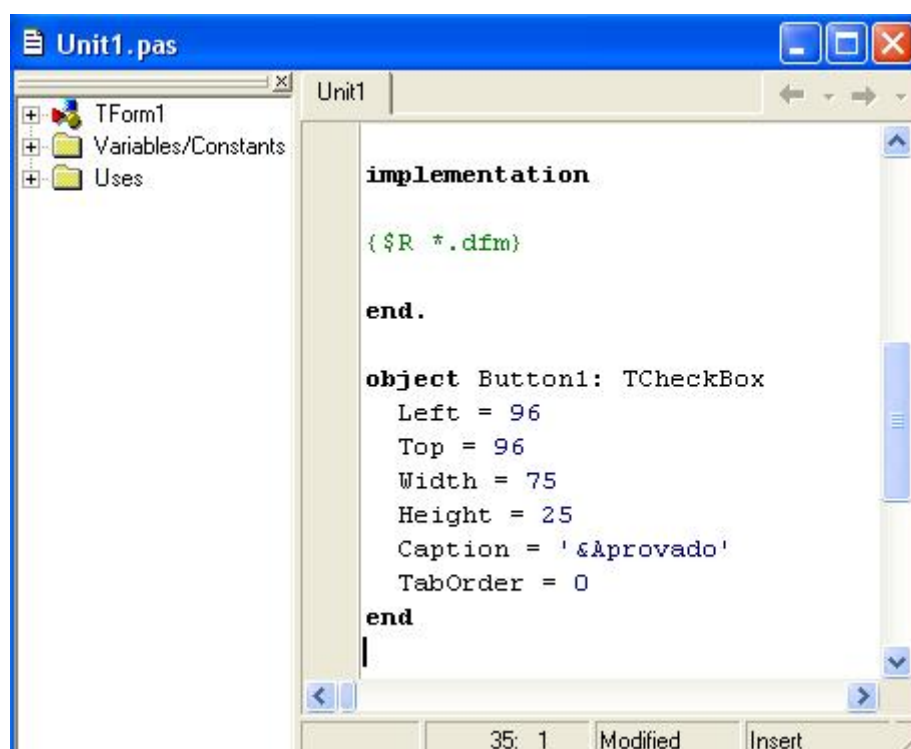
Este exemplo não é algo que você faz regularmente quando está desenvolvendo suas aplicações, mas ele serve para mostrar a independência entre o **Code Editor** e o **Designer**. Inicie o Delphi e clique em **File | New | Application**. Coloque um button do **Tool Palette** no formulário como abaixo:



Agora recorte o button do formulário usando o menu **Edit** ou as teclas de atalho **Ctrl-X**. Depois, pressione F12 para abrir o editor de código e posicione o cursor no final do arquivo. Escolha **Paste** (ou **Ctrl-V**) o button será copiado no **Code Editor**.



O código que você vê acima é igual ao código que aparece nos arquivos de formulário \*.dfm. Você pode modificar o código acima. Que tal mudarmos, via código, o **button** para um **checkbox**? Para isso temos que mudar o tipo do objeto para a classe **TCheckBox**. Você também poderá mudar outras propriedades, mude a propriedade **Caption** para '&Aprovado'. As alterações são mostradas abaixo:



Agora, recorte o código do **Editor**, usando **Edit | Cut**. Retorne ao **Form Designer** e cole diretamente no form usando (**Edit | Paste**) e veja o resultado.

## ***Criando seu Primeiro Programa em Delphi***

É comum que todo programador ao iniciar numa linguagem de programação crie um programa chamado “Alô Mundo!” (Hello World!). Não quebraremos esta tradição aqui.

Agora iremos adicionar um controle do tipo Button no form. Mudaremos algumas propriedades do nosso form. Clique no formulário para “setar” o focus. Usando o **Object Inspector**, configure a propriedade Caption do formulário para “Hello World Demo”.

Selecione o Button no form. Altere a propriedade Caption para “Click here”.

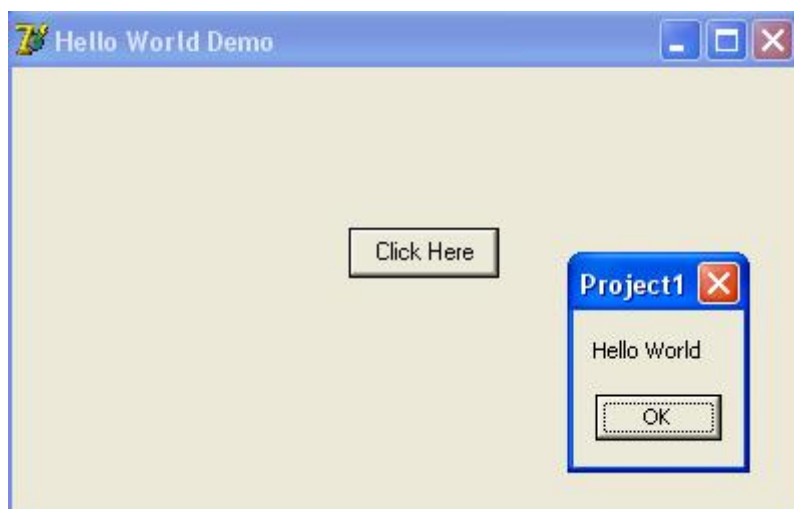
De um duplo clique no Button para abrir o evento Click. O Delphi já cria e registra para você um evento chamado `ButtonClick` event handler. Você poderá começar escrever seu código que será executado quando o Button for clicado.



Adicione o código para o Evento Button1Click:

```
procedure TForm1.Button1Click(Sender: TObject);  
begin  
    ShowMessage('Hello World');  
end;
```

Pressione **F9** para executar a aplicação. A aplicação iniciará em uma janela do tipo Windows irá aparecer, como na figura abaixo. Você pode clicar no **Close** button no topo superior direito para fechar a aplicação e voltar ao ambiente de desenvolvimento.



Agora vamos aprender a salvar nosso projeto. Para isso vá ao menu *File* e escolha a opção *Save All*, ou ainda clique no botão *Save All* do SpeedBar. Você também pode utilizar o atalho de teclado *Shift + Ctrl + S*.

Será apresentada uma caixa de diálogo com o título “Save Unit1 As”, o “Unit” do título indica que estamos salvando o Form (a Unit do mesmo). Observe no item *Salvar em* qual o local onde será salvo o projeto e modifique se necessário ou então crie uma pasta para o mesmo. Em *Nome do arquivo* digite **Principal.pas** e pressione Enter (ou clique no botão *Salvar*).

Uma nova caixa de diálogo será apresentada, desta vez com o título “Save Project1 As”. No item *Nome do arquivo* digite **AloMundo.dpr**.

Com estes procedimentos acabamos de salvar nosso projeto e o formulário do mesmo. Se você observar o diretório em que os salvou, notará a existência dos seguintes arquivos:

- AloMundo.dpr: o arquivo de projeto
- Principal.pas: arquivo com a Unit do Form
- Principal.dfm: arquivo do Form, que recebe automaticamente o mesmo nome dado a sua Unit.
- AloMundo.res: arquivo de recursos do projeto, normalmente contém apenas o ícone que será usado no executável, mas pode receber outros recursos.
- AloMundo.dof e AloMundo.cfg: arquivos de configuração de compilação e ambiente.

Os arquivos principais são os **.dpr**, **.pas** e **.dfm**, os demais caso não sejam encontrados serão recriados com as opções padrão.

Vá ao editor de código e observe que na primeira linha onde estava escrito **unit Unit1**; agora se encontra **unit Principal**; o nome que atribuímos ao formulário quando o salvamos.

Executando novamente a aplicação você observará novamente a pasta onde os arquivos foram salvos verá que dois novos arquivos foram criados:

- **Principal.dcu:** a Unit e o Form compilados, dcu significa Delphi Compiled Unit. A união de todos os .dcu de um projeto juntamente com outras Units internas do Delphi irão formar o arquivo executável .exe.
- **AloMundo.exe:** o aplicativo independente (executável que roda sem o Delphi), é criado com o mesmo nome dado ao projeto.

Caso se deseje fazer uma cópia de segurança de um projeto, os arquivos .dcu e .exe podem ser apagados, pois podem ser recriados através de uma nova compilação do projeto. Após a compilação, apenas o executável é necessário. É ele que será enviado ao usuário final.

## Manipulando Componentes

Usando o **Form Designer**, você pode mudar ou gerenciar os componentes, individualmente ou em grupo que são colocados no formulário.

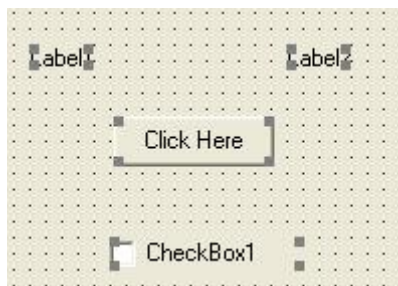
### *Movendo Componentes*

Após colocar os componentes no formulário (buttons, labels, etc..) é muito fácil move-los para novas posições. Componentes visíveis e não-visíveis poderão ser movidos apenas clicando e arrastando-os para a nova posição desejada.

Na maior parte do tempo haverá sempre um componente com o foco. As propriedades deste componente então poderá ser alterada via **Object Inspector**. Quando um componente tem o foco, bordas de redimensionamento aparecem:



É possível também mover e adicionar mais que um componente por vez. Há várias maneiras de fazer isso uma delas é mantendo o botão esquerdo do mouse pressionado criar uma área de seleção (retângulo) para ter vários componentes selecionados:



Outra maneira fácil de fazer múltipla seleção de componentes é pressionar a tecla Shift enquanto você vai clicando nos componentes que você quer selecionar.



Depois de selecionar múltiplos componentes, eles todos podem ser movidos juntos para uma nova posição no formulário.

### *Redimensionando Componentes*

Você pode também mover/redimensionar componentes via teclado diretamente.

- Ctrl+<teclas de setas> moverá o controle ou grupo de controles para pixel por pixel.
- Shift+<teclas de setas> redimensionará o controle ou grupo de controles pixel por pixel.

## Janelas MDI

Os exercícios anteriores foram construirmos apenas aplicativos com SDI (Single document Interface – interface de um único documento). Tais programas (incluindo o Bloco de notas e o Paint) suportam apenas uma janela ou documento aberto por vez. Os aplicativos SDI normalmente tem capacidades limitadas - O paint e o Bloco de notas, por exemplo, tem recursos de edição de imagem e texto limitados. Para editar vários documentos, o usuário precisa criar mais instâncias do aplicativo SDI.

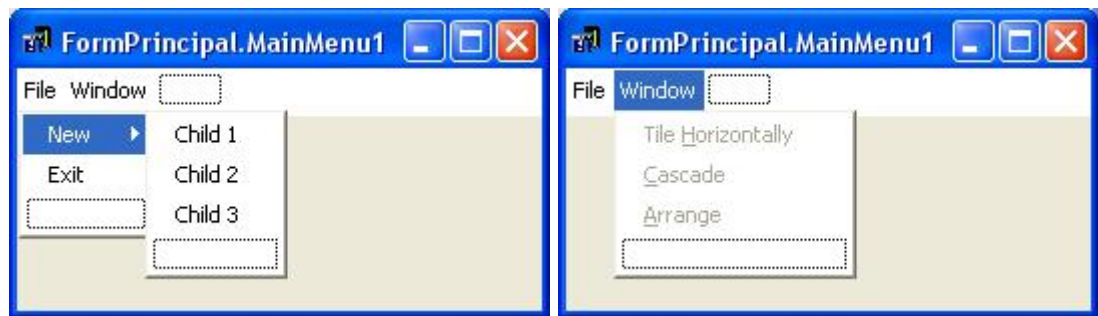
Os programas MDI (Multiple Document Interface – interface de múltiplos documentos), como o PaintShop Pro e o Adobe Photoshop, permitem aos usuários editar vários documentos simultaneamente. Os programas MDI também tendem a ser mais complexos – O PaintShop Pro e o Photoshop tem um número maior de recursos de edição de imagem do que o Paint.

A janela de aplicativo de um programa MDI é chamada de *janela progenitora*, e cada janela dentro do aplicativo é referida como *janela filha*. Embora um aplicativo MDI possa ter muitas janelas filhas, cada um tem apenas uma janela progenitora. Além disso, um máximo de janela filha pode estar ativa por vez. As janelas filhas não podem ser progenitoras e não podem ser movidas para fora de suas progenitoras. Fora isso, uma janela filha se comporta como qualquer outra janela (com relação ao fechamento, à minimização, ao redimensionamento etc). A funcionalidade de uma janela filha pode ser diferente da funcionalidade de outras janelas filhas da progenitora. Por exemplo, uma janela filha poderia editar imagens, outra poderia editar texto, e uma terceira poderia exibir o tráfego da rede graficamente, mas todas poderiam pertencer à mesma progenitora MDI. Veja o exemplo abaixo:



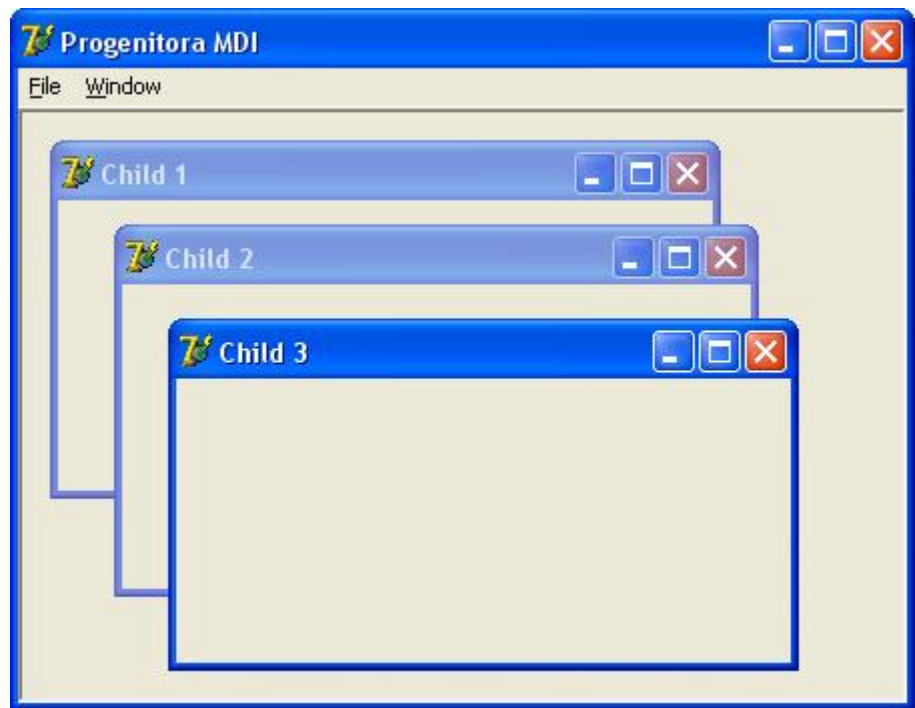
Para criar um formulário MDI, crie um novo projeto e mude a propriedade do **Form** `FormStyle` para `fsMDIForm`. Coloque nomes mais significativos para as propriedades `Name` (`FormPrincipal`) e `Caption` (`Progenitora`

MDI) do **Form**. Adicione também um componente do **TMainMenu** e deixe-o configurado como abaixo:

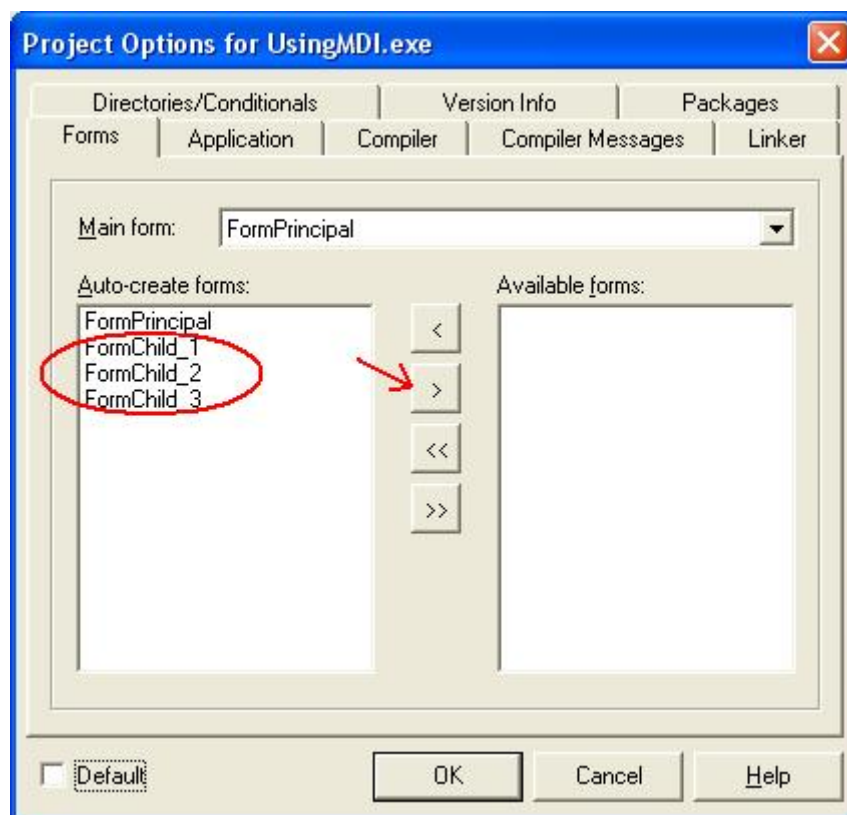


Já temos definida a nossa janela progenitora do projeto. Para criar uma janela filha devemos criar uma classe de formulário filho para ser adicionado ao progenitor. Para isso vá em **File | New | Form**. Mude a propriedade do **Form** **FormStyle** para **fsMDIChild**. Dê nomes significativos para as propriedades **Name** (**FormChild1\_1**) e **Caption** (**Child 1**) do **Form**. Repita o mesmo procedimento para dois novos Forms.

Se você seguiu corretamente os passos até aqui, o seu projeto, ao ser executado, deve estar parecido com este:



Ao executar o aplicativo, nota-se que o progenitor já abre automaticamente os três **Form** filhos definidos anteriormente. Para que isso não aconteça devemos tirar os formulários filhos do Auto-Create no Delphi. Em tempo de projeto vá no menu **Project | Options**:



Deixe no campo **Auto-Create forms** somente o FormPrincipal (formulário progenitor), os demais devem ser transferidos para o campo **Available forms**. Isso impedirá que a aplicação crie automaticamente esses formulários filhos no momento da execução.

Nota: Sempre faça esse procedimento quando criar novos **Forms** filhos.

Iremos agora criar os formulários filhos a partir do **Menu** criado no formulário progenitor. As seleções do menu (como **File**, seguido da opção de submenu **Child 1**, **Child 2** e **Child 3** seguido da opção de menu **Window**) são métodos comuns utilizados para criar novas janelas filhas.

Para criar o formulário a partir do menu adicione o seguinte código no evento **OnClick** do submenu **Child 1**:

```
//Cria o formulário filho 1
Procedure TFormPrincipal.Child1Click(Sender: TObject);
var Child: TFormChild_1;
begin
    Child := TFormChild_1.Create(Application);
end;
```

Faça o mesmo para as opções **Child 2** e **Child 3** do menu.

No próximo capítulo introduziremos conceitos sobre OOP (Object Oriented Programming – Programação Orientada a Objeto) e analisaremos em maior profundidade as propriedades e os eventos do Form e de alguns componentes.

## Capítulo 2

### **Programação Orientada a Objeto (POO)**

De maneira prática, podemos pensar no objeto sendo uma estrutura que agrupa dados e funções para manipular estes dados. Como as funções são sempre “intimamente ligadas” aos dados, o sistema todo funciona de maneira mais segura e confiável. Além disso, a POO utiliza conceitos como **encapsulamento e herança** que facilitam muito a programação e manutenção dos programas.

Os dados de um objeto costumam ser chamados de **variáveis de instância** e as funções de **métodos**. As **variáveis de instância** definem as **propriedades** (também chamadas de **atributos**) do objeto e os **métodos** definem o seu comportamento.

Para programar no nível do *designer* não é necessário um conhecimento profundo de POO. Mas é preciso conhecer pelo menos a sintaxe. Todas as aplicações para Windows precisam de pelo menos uma janela, que no Delphi é chamada de *Form*. Cada *form* (assim como todos os objetos visuais) tem um objeto associado a ele e sua representação visual, que vemos na tela. Todos os componentes que incluímos no form passam a fazer parte do objeto que define o form. Exemplo: Se colocarmos um botão no form, a classe deste form será modificada para incluir este botão. Os eventos e métodos deste form, também estão na classe. Assim, supondo que o form se chame Form1 (nome default), por exemplo, para desativar o botão que incluímos (de nome Button1) escreveríamos:

```
Form1.Button1.Enabled := false;
```

Note como o Button1 faz parte da estrutura que define o form. Mas se quisermos ativar **método RePaint** (Repintar) do form faríamos:

```
Form1.Repaint;
```

Veja que Repaint, não é uma variável, tecnicamente **é uma procedure (método)**, mas fazemos referência a ela como parte da estrutura Form1. Isso pode parecer confuso no início, mas facilita muito a programação.

### **Conhecendo propriedades, métodos e eventos básicos**

Para que possamos prosseguir veremos agora algumas **propriedades, métodos e eventos** comuns a maior parte dos componentes do Delphi:

#### **Propriedades Básicas**

**Name:** É comum a todos os componentes da paleta. O Delphi nomeia automaticamente todos os componentes que são incluídos no form (inclusive o próprio form). Usa o nome da classe do componente **mais** um número sequencial. O nome atribuído pelo Delphi pode ser mantido, mas é aconselhável renomear os componentes que serão referidos no programa. No nosso primeiro exemplo, o programa AloMundo, o Form e o Label que inserimos tinham os nomes de Form1 e Label1, mas nós os alteramos para FormPrincipal e LabelMensagem. Como não fizemos nenhuma programação isso seria até desnecessário, mas você deve criar o hábito de renomear todos os componentes que utiliza, caso seja feita referência a eles no código fonte, é muito mais claro imaginar que um botão chamado ButtonFechar seja para fechar seu form do que se o nome fosse Button1 ou Button13. Quando você renomeia um componente, o Delphi atualiza automaticamente todo o código gerado por ele, o que inclui o cabeçalho da Unit, os eventos do componente e as propriedades de outros componentes que fazem referência ao componente renomeado, **mas não atualiza o código gerado por você**. Exemplo: se renomearmos um botão de Button1 para ButtonIncluir, o Delphi atualizará o cabeçalho da unit, o nome dos eventos de Button1, mas você terá que atualizar todas as referências que você fez ao Button1 em seu código com o novo nome. Aliás, esta é uma regra geral no Delphi: ele nunca modifica automaticamente o código gerado pelo programador, mesmo que esteja em comentário. E por segurança, você deve fazer o mesmo, não modifique o código gerado por ele, a não ser que saiba exatamente o que está fazendo, poderá inutilizar seu form, em alguns casos tendo problemas até para salva-lo.

**Caption:** Todos os componentes que podem apresentar um rótulo tem esta propriedade. Armazena a string que será mostrada quando o componente for desenhado. No Form ele corresponde ao texto exibido na barra de título.

**Left e Top:** Esquerda e Topo. Armazenam a posição do componente em relação ao form ou painel que o contém. Movendo o componente, estas propriedades se atualizam automaticamente, e alterando estas propriedades, o componente é movido.

**Height e Width:** Altura e comprimento do componente, alterando estas propriedades, o componente terá seu tamanho alterado.

**Font:** Permite selecionar a fonte, o tamanho, o estilo e a cor da fonte que será usada para escrever o texto no componente.

**Color:** Cor do componente. Existe uma lista de cores padrão usadas pelo Windows e pelo Delphi, mas é possível definir qualquer cor através de seus componentes RGB.

**TabOrder:** Ordem do componente no Form ou painel. Quando há vários componentes selecionáveis no Form ou painel, a tecla Tab permite navegar entre os componentes. Esta propriedade define a ordem em que os componentes são selecionados quando o usuário tecla Tab.



**Hint:** (Dica) Este é um recurso muito útil e fácil de usar. Permite que apareça um texto de ajuda quando o usuário posiciona o cursor do mouse sobre um componente. Todos os componentes podem ter Hint. Na propriedade Hint, você deve digitar a frase que deve aparecer. Veja a propriedade abaixo.

**ShowHint:** Ativa o hint para o componente. Se estiver desligado, o hint não é mostrado.

## Eventos básicos

**OnClick:** É gerado cada vez que o botão esquerdo do mouse é pressionado e solto. O evento só ocorre quando o usuário libera o botão. O Delphi já direciona o evento para o componente que está debaixo do cursor do mouse.

**OnDbClick:** Gerado quando é feito um duplo clique no botão esquerdo.

**OnKeyPress:** Gerado quando o usuário pressiona (e libera) uma tecla no teclado.

**OnEnter:** Gerado quando o foco de atenção do Windows “cai” sobre o componente. Exemplo: Suponha uma tela de entrada de dados, onde vários campos (caixas de texto) devem ser digitados. Quando a tela é apresentada, o foco, ou o cursor de texto, está sobre o primeiro campo. Depois de digitar o campo, o usuário pressiona Tab para *passar* para o campo seguinte. Veja que o que *passa* para o campo seguinte é a atenção da aplicação e do Windows. Essa atenção é chamada de foco, ou **focus** em inglês. Este evento é gerado assim que o componente recebe o foco, antes de executar qualquer código do componente.

**OnExit:** Gerado imediatamente antes de o foco deixar o componente.

**OnResize:** Gerado quando o tamanho do componente é alterado. É usado principalmente em forms e painéis.

**OnChange:** Gerado quando o valor do componente muda. Só é aplicado em componentes que permitem edição de seu conteúdo.

## Métodos Básicos

**Show:** Desenha o componente. Se for uma janela (form) ela é desenhada e ativada.

**Close:** Fechar. Aplicado geralmente em forms e componentes de acesso a dados. Quando utilizado no form principal, encerra a aplicação.

**Repaint:** Repintar. Redesenha o componente ou form.

**Refresh:** Tem o mesmo efeito que o **Repaint**, mas antes de desenhar, apaga o componente. Quando aplicado em componentes de acesso a dados, faz com que o buffer do arquivo seja recarregado.

**Create:** Aloca memória para criar um componente ou form, dinamicamente.

**Free:** Libera a memória alocada com o **Create**.

## Implementando Orientação à Objetos em Delphi

Iniciaremos a implementação de OO em Delphi criando uma classe que simule um contador de cédulas. Os objetivos dessa classe é emitir a quantidade de notas necessárias para compor um determinado valor. A fim de simplificar a classe, não aceitaremos centavos e a saída da distribuição de nota será na realidade uma string descritiva.

Analizando o problema, chegamos à representação da classe exibida abaixo:

ContagemCedulas
+valorContagem : integer -qtdNotas100:integer -qtdNotas50:integer -qtdNotas10:integer -qtdNotas5:integer -qtdNotas2:integer -qtdNotas1:integer
-distribuição() +ObterDistribuicao():string

Além da classe propriamente dita, devemos pensar na interface que se comunicará com essa classe. Essa interface deve solicitar ao usuário um valor para contagem de cédulas. Mostrarei a implementação da interface e da classe.

Nossa primeira tarefa será criar a classe ContagemCedulas. Para isso, vamos aos passos a seguir:

- Inicie o Delphi. Você deve reparar que, ao abrir o programa, ele automaticamente lhe traz uma nova aplicação, com um formulário vazio. Esse formulário nos servirá para a interface com o usuário. Mas nesse momento, vamos deixá-lo um pouco de lado.
- Vamos criar a unit que conterá a classe. No menu do Delphi, escolha as opções **File – New – Unit**. Ao ser aberta, salve a unit com o nome **CLContagemCedulas**. Digite o código abaixo que corresponde à definição da referida classe. A seguir, será realizada uma explicação do código da Unit.

```

unit CLContagemCedulas;

interface

uses Classes, Dialogs, SysUtils;

type
  TContagemCedulas = class
  protected
    FValorContagem : integer;
    FQtdNotas100   : integer;
    FQtdNotas50    : integer;
    FQtdNotas10    : integer;
    FQtdNotas5     : integer;
    FQtdNotas2     : integer;
    FQtdNotas1     : integer;

    procedure Distribuicao;
    procedure DefineValorContagem(valor: integer);
  public
    function ObterDistribuicao: TStringList;
    property ValorContagem: integer read FValorContagem
                                     write DefineValorContagem;
  end;

implementation

uses Math;

{ TContagemCedulas }

procedure TContagemCedulas.Distribuicao;
var iAux: integer;
begin
  iAux := FValorContagem;

  FQtdNotas100 := iAux div 100;
  iAux := iAux mod 100;

  if iAux > 0 then
  begin
    FQtdNotas50 := iAux div 50;
    iAux := iAux mod 50;

    if iAux > 0 then
    begin
      FQtdNotas10 := iAux div 10;
      iAux := iAux mod 10;

      if iAux > 0 then
      begin
        FQtdNotas5 := iAux div 5;
        iAux := iAux mod 5;

        if iAux > 0 then
        begin
          FQtdNotas2 := iAux div 2;
          iAux := iAux mod 2;

          FQtdNotas1 := iAux;
        end;
      end;
    end;
  end;
end;

```

```

procedure TContagemCedulas.DefineValorContagem(valor: integer);
begin
    if valor < 0 then
        FValorContagem := 0
    else
        begin
            FValorContagem := valor;
            Distribuicao();
        end;
    end;
end;

function TContagemCedulas.ObterDistribuicao: TStringList;
var lista : TStringList;
begin
    lista := TStringList.Create;

    lista.Add(IntToStr(FQtdNotas100) + ' nota(s) de 100');
    lista.Add(IntToStr(FQtdNotas50) + ' nota(s) de 50');
    lista.Add(IntToStr(FQtdNotas10) + ' nota(s) de 10');
    lista.Add(IntToStr(FQtdNotas5) + ' nota(s) de 5');
    lista.Add(IntToStr(FQtdNotas2) + ' nota(s) de 2');
    lista.Add(IntToStr(FQtdNotas1) + ' nota(s) de 1');

    result := lista;
end;

end.

```

Nos tópicos a seguir, serão feitos comentários sobre os diversos pontos dessa classe. Vamos retornar à análise do código.

### ***Herança***

Na definição da **classe TcontagemCedulas**, a **classe-pai** foi omitida, por ser automaticamente a **classe TObject**.

TcontagemCedulas = class

Se desejássemos herdar essa classe de outra, bastaria definir da seguinte forma:

TcontagemCedulas = class(**superclasse**)

Supondo, por exemplo, uma **classe AlunoUniversitário** que herde da **superclasse Aluno**, sua definição seria:

Type TalunoUniversitario = class(**Taluno**)

### ***Visibilidade de Atributos e Operações***

Para a **classe ContagemCedulas**, estaremos definindo seus **elementos** privados, protegidos e públicos, por meio das declarações **Private**, **Protected** e **Public**.

- Tudo que for definido como **private** só poderá ser acessado dentro da própria classe, e no caso de uma herança, não será enxergado.
- Já as definições abaixo da declaração **protected** serão acessadas dentro da própria classe e pelas classes descendentes.
- As definições feitas na declaração **public** serão acessadas dentro e fora (objetos e outras classes) da classe.

## Encapsulamento

O encapsulamento determina que os atributos não possam ser acessados diretamente, mas somente pela interface da classe. De maneira geral, esse encapsulamento é representado pelo uso de operações *Set* e *Get* que realizam, respectivamente, a escrita e a leitura dos atributos. Entretanto, o Delphi permite que sejam definidas **properties** para representar esse encapsulamento.

Por exemplo: vamos supor uma classe **Funcionario** com os atributos Nome e Salario. A definição inicial dessa classe (permitindo a herança dos atributos) seria:

```
Type Tfuncionario = class
Protected
    Fnome: string;
    Fsalario: real;
    ...
```

Repare que os atributos, sendo protegidos, não poderão ser acessados pelas instâncias dessas classes. Nesse caso, precisamos criar operações que realizem esse acesso:

```
Type Tfuncionario = class
Protected
    Fnome: string;
    Fsalario: real;
Public
    Procedure DefinirNome(sNome:string);
    Procedure DefinirSalario(rSalario:real);
    Procedure ObterNome:string;
    Procedure ObterSalario:real;
End;
```

Na definição dessas operações estamos determinando que quando quisermos atribuir valores para os atributos Nome e Salario devemos chamar as operações *DefinirNome* e *DefinirSalario*, respectivamente, passando os valores por parâmetro. Da mesma forma, quando quisermos ler os valores de Nome e Salário, devemos chamar as operações *ObterNome* e *ObterSalario*, que são do tipo funções e retornam os valores desejados. Por exemplo:

```

Var
  oFuncionario: TFuncionario;
  sNome: string;
  rSalario: real;
begin
  ...
  oFuncionario.DefinirNome('Maria');
  oFuncionario.DefinirSalario(2000);
  ...
  sNome := oFuncionario.ObterNome;
  rSalario := oFuncionario.ObterSalario;
  ...

```

Uma **property** também **define um atributo**, todavia podendo associar a ações relacionadas **à leitura ou modificação** de seus dados. Assim, ao acessarmos uma property não estaríamos acessando diretamente o conteúdo de um atributo, e sim, **um procedimento** que seria um intermediário desse acesso. Esse procedimento **permite a escrita** de códigos de validação e **formatação** dos atributos.

Para o mesmo exemplo anterior, veja como ficaria a definição da classe, utilizando a **property**.

```

Type Tfuncionario = class
Protected
  Fnome: string;
  Fsalario: real;
  Procedure DefinirNome(snome: string);
  Procedure DefinirSalario(rSalario: real);
Public
  Property Nome: string read Fnome      write DefinirNome;
  Property Salario: real read Fsalario    write DefinirSalario;
End;

```

Utilizando o exemplo anterior, vejamos, para essa classe, como ficaria o código de acesso aos atributos:

```

Var
  oFuncionario: TFuncionario;
  sNome: string;
  rSalario: real;
begin
  ...
  oFuncionario.Nome := 'Maria';
  oFuncionario.Salario := 2000;
  ...
  sNome := oFuncionario.Nome;
  rSalario := oFuncionario.Salario;
  ...

```

## Seções Interface e Implementation

A estrutura de uma unit **determina** que as definições das procedures, funções ou dos métodos da classe fiquem na **seção interface**, enquanto que o código, propriamente dito, fique na seção **implementation**. Assim na seção **implementation** as operações “criam vida”. **A definição do método** na seção implementation deve ser feito da seguinte forma:

**Escopo Tipo-do-método Nome-da-Classe.Assinatura-da-Operação**

Escopo corresponde ao escopo da operação (**de instância ou de classe**). Se nada for colocado, é considerado como de instância. Para definir um escopo de classe deve-se colocar a palavra **class**.

Tipo-do-método corresponde à **procedure** ou **function**.

Nome-da-Classe **é o mesmo** nome definido na declaração **type**.

Assinatura-da operação é a mesma assinatura definida na seção interface.

Por exemplo:

**Class function Tfuncionario.GerarFolhaPagamento: boolean**

Desta forma, continuando a analisar o código da classe ContagemCedulas, verificamos que foram criados os métodos para as operações Distribuicao, DefineValorContagem e ObterDistribuicao.

## Métodos Construtor e Destrutor

Toda classe possui seus **métodos construtores de destrutores**. Um método construtor **cria uma área** em memória para o objeto, **colocando nesta área a estrutura da classe**, ou seja, **instanciando um objeto** a partir da classe. No Delphi o método construtor de uma classe é definido da seguinte forma:

**constructor** Nome-do-Método;

Por exemplo:

**constructor** Create

A **classe Tobject** já possui um método *constructor*, o que facilita a criação de novas classes. Entretanto, se numa subclasse for necessário realizar alguma ação especial na criação de uma instância, basta **redefinir** o método *constructor* na subclasse. No corpo dos métodos, **reescreve-se** a assinatura da operação e na seção de implementação, **define-se** novamente o métodos. Veja o exemplo:



```

Type Taluno = class
  Private
  ...
  Public
  ...
  Constructor Create;
  ...
End;

Implementation

...

Constructor Taluno.Create;
Begin
  Inherited Create;
  //novas ações...
End;

```

Da mesma forma, o **método destrutor** serve para **eliminar o objeto da memória**, liberando o espaço ocupado. A redefinição também é possível para este método. No Delphi o método *destructor* de uma classe é definido da seguinte forma:

**Destructor** Nome-do-Método; override;

Por exemplo:

**Destructor** Destroy; **override;**

Retornando ao formulário que deixamos de lado. Nele construiremos a interface que fará a ponte entre o usuário e nossa classe. Para isso, **monte** o formulário como abaixo:

**Mude** a propriedade **name** dos seguintes componentes:

- (Edit) : edtValorContagem
- (Button) : btnProcessar
- (Memo) : mmoCedulas
- (Formulário) : frmDLContagemCedulas

Salve o formulário com o nome de **DLContagemCedulas**.

Agora, precisamos fazer com que o formulário se relacione com a Classe ContagemCedulas, passando o valor para contagem e recebendo a quantidade de cédulas. Para isso, vamos acessar a Unit do Formulário. No código, procure seção implementation e logo abaixo coloque a cláusula uses acompanhada do nome da unit CLContagemCedulas, conforme descrito a seguir:

```
Implementation
{$R *.dfm}

Uses
    CLContagemCedulas;
```

Essas duas linhas dizem ao Formulário que ele passará a enxergar a interface da unit CLContagemCedulas, que é nossa classe ContagemCedulas. Em seguida, volte ao Formulário, dê um duplo clique no botão “Contar Cédulas” a fim de acionar o evento default – o **OnClick**. O Delphi automaticamente cria a estrutura deste evento. Basta que você digite o código que irá se comunicar com a classe. Assim, digite o código marcado em negrito que está abaixo. Note que, para melhor entendimento, o exemplo apresenta toda a unit do formulário DLContagemCedulas.

```
unit UPrincipal;

interface

uses
    Windows, Messages, SysUtils, Variants, Classes, Graphics, Controls,
    Forms,
    Dialogs, StdCtrls, Buttons;

type
    TfrmDLContagemCedulas = class(TForm)
        Label1: TLabel;
        Label2: TLabel;
        edtValorContagem: TEdit;
        btnProcessar: TBitBtn;
        mmoCedulas: TMemo;
        procedure btnProcessarClick(Sender: TObject);
    private
        { Private declarations }
    public
        { Public declarations }
    end;
```

```

var
    frmDLContagemCedulas: TfrmDLContagemCedulas;

implementation

uses CLContagemCedulas;

{$R *.dfm}

procedure TfrmDLContagemCedulas.btnProcessarClick(Sender: TObject);
var oContagem : TContagemCedulas;
begin
    oContagem := TContagemCedulas.Create;

    try
        oContagem.ValorContagem := StrToInt(edtValorContagem.Text);
        mmoCedulas.Clear;
        mmoCedulas.Lines.AddStrings(oContagem.ObterDistribuicao);
    except
        ShowMessage('Valo de contagem inválido');
    end;

    oContagem.Free;

end;

end.

```

**Inicialmente** foi criada uma variável **oContagem** que é do tipo Classe TcontagemCedulas. Essa definição da variável não cria efetivamente o objeto na memória. Para isso precisamos da linha:

oContagem := TcontagemCedulas.Create.

que, efetivamente, **instancia** um objeto da Classe TcontagemCedulas e o identifica como oContagem.

**A seguir** preenchemos o valor do atributo ValorContagem, com o valor informado pelo usuário no campo edtValorContagem. Ao definir esse valor, a classe, automaticamente, se encarrega de calcular a quantidade notas necessárias para representar o valor conhecido. **Logo a seguir**, o **campo Memo** é limpo e preenchido com o resultado do método ObeterDistribuicao, que retorna uma string contendo a distribuição das notas. Essas operações estão voltadas num **comando try.. except**, que captura qualquer erro gerado no acesso à classe ou nas informações recebidas do usuário. Por não precisar mais do objeto, **no final do código** chamamos o **método que o destrói (Free)**.

Aparentemente o código da classe é grande em relação ao código para seu uso. Mas na realidade o código da classe está tão grande quanto o código que escreveríamos numa programação não orientada a objetos. Todavia, nesse caso, temos um **código encapsulado**, que nos dará grandes benefícios no momento de uma manutenção. Além disso, podemos distribuir essas classes para quem delas precisar, sabendo que o esforço gerado para usá-las será mínimo. Pense em você do outro lado – como um usuário da classe!

Ao executar este programa e informar o valor 1215, obtemos no campo Memo o resultado demonstrado abaixo:

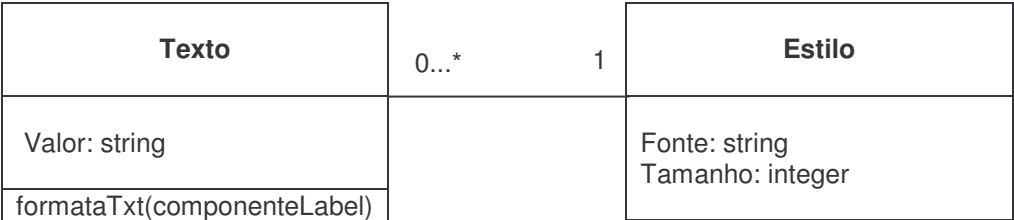
12 nota(s) de 100
0 nota(s) de 50
1 nota(s) de 10
1 nota(s) de 5
0 nota(s) de 2
0 nota(s) de 1

Implementando Associação

Antes de exemplificar o relacionamento de associação, é preciso falar um pouco sobre a modelagem de classes na fase de projeto. Os relacionamentos, na realidade, são implementados no formato de atributos ou coleções de objetos.

Na fase de projeto, diante de relacionamentos entre classes do tipo um para um, podemos gerar atributos simples em ambas as classes, que serão responsáveis pela ligação entre as mesmas. Para relacionamentos entre classes do tipo um para muitos, colocamos um atributo simples na classe que tem a multiplicidade muitos e/ou um atributo que representa uma coleção de objetos na classe que tem a multiplicidade um. Para relacionamentos entre classes do tipo muitos para muitos, podemos colocar atributos representando coleção de objetos em ambas as classes.

Para exemplificarmos o relacionamento de associação, vamos considerar a seguinte situação: necessitamos de uma classe que permita a formatação de um label com um determinado estilo. Esse estilo está associado ao texto, contendo a fonte e seu tamanho:



Note que a classe Estilo se relaciona com a classe Texto. No momento de implantarmos esse relacionamento, a classe texto passará a ter um atributo estilo do tipo classe Estilo.

Implementando o Código

Começaremos pela definição das classes. **Crie** uma nova aplicação. **Deixe** de lado, temporariamente o formulário criado automaticamente pelo Delphi e vamos criar nossa primeira classe (**menu File – new – Unit**). **Salve-a** como o nome de **CLEstilo**. **Digite**, então o código abaixo:

```
unit CLEstilo;

interface

uses Classes;

type
  TEstilo = class
  protected
    FFonte : string;
    Ftamanho : integer;
    procedure SetTamanho(valor : integer);
  public
    constructor Create;
    property Fonte : string      read FFonte write FFonte;
    property Tamanho : integer  read Ftamanho write SetTamanho;
  end;

implementation

{ TEstilo }

constructor TEstilo.Create;
begin
  Fonte := 'MS Sans Serif';
  Tamanho := 10;
end;

procedure TEstilo.SetTamanho(valor: integer);
begin
  if valor > 4 then
    Ftamanho := valor
  else
    Ftamanho := 4;
end;

end.
```

Da mesma forma, crie uma **nova unit**, salve com o nome de **CLTexto** e digite o código abaixo:

```
unit CLTexto;

interface

uses Classes, StdCtrls, CLEstilo;

type
  TTexto = class
  protected
    FValor : string;
    FEstilo : TEstilo;
```

```

public
    constructor Create;
    destructor Destroy; override;

    procedure formataTxt(componente : TLabel);

    property Valor : string      read FValor write FValor;
    property Estilo: TEstilo     read FEstilo write FEstilo;
end;

implementation

{ TTexto }

constructor TTexto.Create;
begin
    FValor := '';
    FEstilo := TEstilo.Create;
end;

destructor TTexto.Destroy;
begin
    FEstilo.Free;
    inherited;
end;

procedure TTexto.formataTxt(componente: TLabel);
begin
    componente.Caption := Valor;
    componente.Font.Name := FEstilo.Fonte;
    componente.Font.Size := FEstilo.Tamanho;
end;

end.

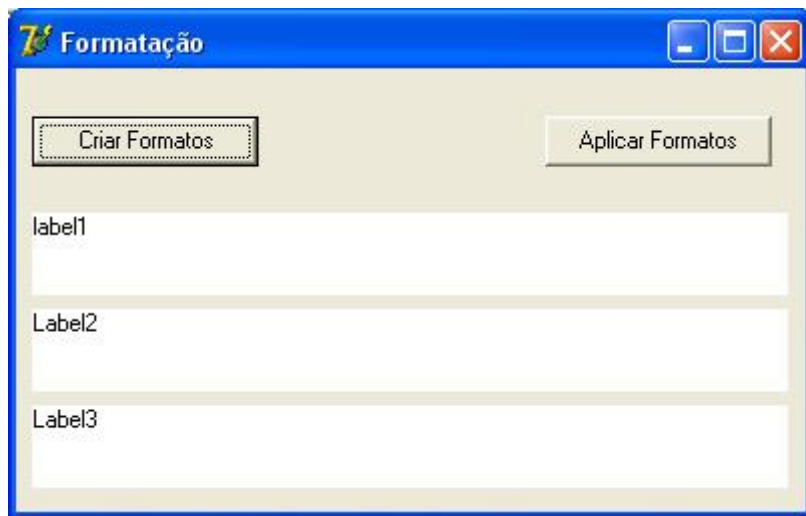
```

### Criação dos **métodos Constructor e Destructor**

Note que nessa classe temos os métodos **Create** e **Destroy** criados, explicitamente. Isto foi necessário pois possuímos um atributo na **Classe Texto**, que é a instância de **outra classe**, ou seja, um objeto. Ao criarmos a classe Texto, se nada for dito à classe Estilo, esta não será criada. Na hora de nos **referenciarmos** à Estilo da Classe Texto, uma exceção será gerada.

Assim, aproveitamos o método Constructor para instanciar o **atributo Festilo**. Também **foi inicializado** a classe com um valor *default*, que permite que a chamada da operação formataTxt nunca gere erro (lembrando que o Create da classe Estilo também inicializa o objeto com valores *default*).

Vamos criar, agora, a interface que usará essa classe. **Retorne** ao formulário do projeto e **coloque** dois botões e três componentes label, conforme figura abaixo, e **salve** o formulário com o nome **DLTexto**.



Mude a propriedade **name** dos seguintes componentes:

- (botão Criar Formatos) – btnFormatos
- (botão Aplicar Formatos) - btnAplicar
- (1º Label) – label1
- (2º Label) – label2
- (3º Label) – label3
- (formulário) – frmFormatcao

Mude o valor das propriedades dos labels **label1**, **label2** e **label3** para:

- AutoSize – False
- Height – 41
- Width – 377
- Color – clWhite

Vamos passar para a **unit do Formulário**. Para que o objeto usado no primeiro botão (Criar Formatos) seja usado pelo segundo botão (Aplicar Formatos), é necessário que o objeto seja criado público na classe. Assim, localize a **seção implementation** do formulário e digite as linhas abaixo para respectivamente, ligar o formulário à classe Texto e definir objetos de formatação públicos.

```
implementation  
  
uses CLTexto;  
  
var  
    oFormato1, oFormato2 : TTexto;
```

Como os objetos **oFormato1** e **oFormato2** são públicos à unit, é uma boa opção criá-los no momento da criação do Formulário e destruí-los no momento da destruição do formulário. Assim, volte ao formulário e acione os eventos **OnCreate** e **OnDestroy** do Formulário. Neles digite o código abaixo:

```

procedure TfrmFormatacao.FormCreate(Sender: TObject);
begin
    oFormato1 := TTexto.Create;
    oFormato2 := TTexto.Create;
end;

procedure TfrmFormatacao.FormDestroy(Sender: TObject);
begin
    oFormato1.Free;
    oFormato2.Free;
end;

```

Vamos, então criar o código do evento **OnClick** dos botões **Criar Formatos** e **Aplicar Formatos**. **Copie** o código abaixo:

```

procedure TfrmFormatacao.btnFormatosClick(Sender: TObject);
begin
    oFormato1.Valor := 'Linha em Arial - 20';
    oFormato1.Estilo.Fonte := 'Arial';
    oFormato1.Estilo.Tamanho := 20;

    oFormato2.Valor := 'Linha em Times - 26';
    oFormato2.Estilo.Fonte := 'Times New Roman';
    oFormato2.Estilo.Tamanho := 26;

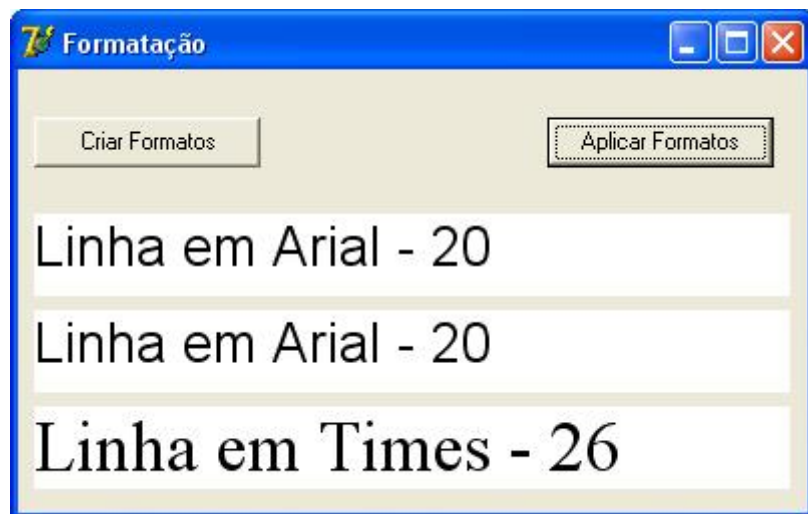
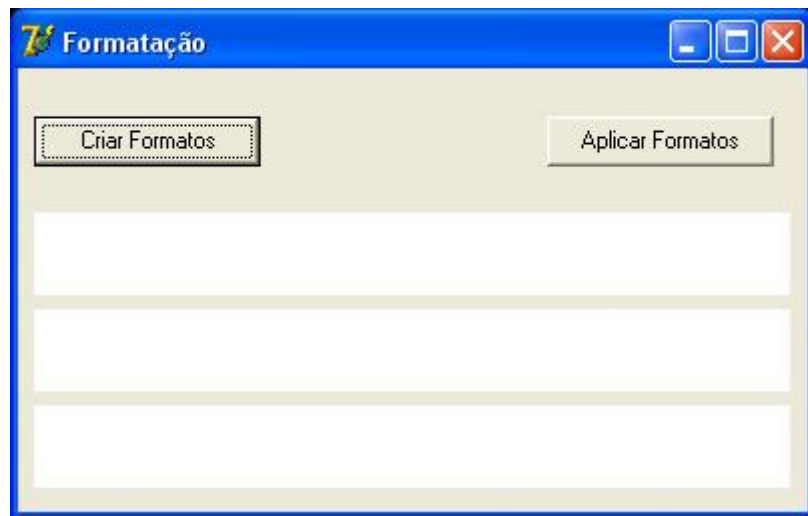
    ShowMessage('Formatos criados corretamente');
end;

procedure TfrmFormatacao.btnAplicarClick(Sender: TObject);
begin
    oFormato1.formataTxt(label1);
    oFormato1.formataTxt(Label2);
    oFormato2.formataTxt(Label3);
end;

```

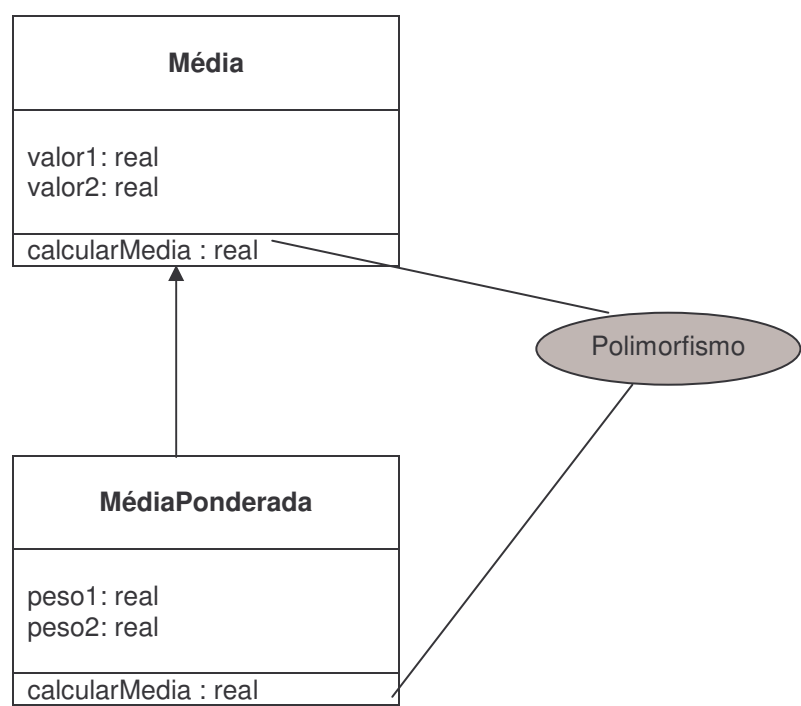
Ao executar a aplicação, clicando primeiramente no botão “Aplicar Formatos”, obteremos o resultado da primeira figura abaixo. Isto ocorrer porque o botão “Criar formatos” não foi acionado primeiro. Então, a classe devolveu seu **valor default** (sem texto, fonte MS Sans Serif e tamanho 10). Se clicarmos em seguida no botão “Criar Formatos” e logo depois em “Aplicar Formatos”, obteremos o resultado da segunda figura.





### Implementando Herança e Polimorfismo

Demonstraremos agora com implementamos herança e polimorfismo. Para tanto, vamos considerar uma superclasse Media que possua os atributos valor1 e valor2. Esta classe possui, ainda uma operação CalcularMedia que calcula a média aritmética destes valores. Se criarmos uma subclasse MediaPonderada que herde da classe Media, teremos um problema na operação CalcularMedia, pois na superclasse o cálculo é feito sobre a Média Aritmética e no segundo caso sobre a média ponderada. Assim, manteremos a herança, porém fazendo com que a operação CalcularMedia seja polimórfica. Chegamos, então ao diagrama de classes da figura abaixo:



### Implementando o código

Vamos começar pela definição das classes. Cria uma nova aplicação. Deixe de lado, temporariamente, o formulário criado automaticamente pelo Delphi e vamos criar nossa primeira classe (menu **File – new Unit**). Salve-a com o nome de **CLMedia**. Digite, então o código abaixo:

```
unit CLMedia;

interface
uses Classes;

type
    TMedia = class
```

```

protected
    FValor1 : real;
    FValor2 : real;
    procedure SetVal1(valor : real);
    procedure SetVal2(valor : real);
public
    constructor Create;
    function CalcularMedia : real; virtual;

    property Valor1 : real    read FValor1 write SetVal1;
    property Valor2 : real    read FValor2 write SetVal2;
end;

implementation

{ TMedia }

function TMedia.CalcularMedia: real;
begin
    Result := (Valor1 + Valor2) / 2;
end;

constructor TMedia.Create;
begin
    FValor1 := 0;
    FValor2 := 0;
end;

procedure TMedia.SetVal1(valor: real);
begin
    if valor > 0 then
        FValor1 := valor
    else
        FValor1 := 0;
end;

procedure TMedia.SetVal2(valor: real);
begin
    if valor > 0 then
        FValor2 := valor
    else
        FValor2 := 0;
end;

end.

```

Para que na interface possamos criar um único objeto que trabalhe, dependendo dos valores dos pesos, com a média aritmética ou ponderada, **precisaremos** de uma declaração especial nas classes.

Quando definimos uma operação na classe, esta é definida como estática, ou seja, seu **método (código)** é determinado no momento da compilação. Para que **possamos definir** o método apenas na execução, **devemos** defini-lo como **virtual**. Para tanto, **deve-se** colocar a palavra-chave **virtual** (acompanhada de um ponto-e-vírgula), logo após a declaração na superclasse.

Nas subclasses, quando essa operação for redefinida, **coloca-se** a palavra-chave **override** (acompanhada de um ponto-e-vírgula), logo após a redeclaração da operação.

Da mesma forma, **crie** uma nova unit, salve com o nome de **CLMediaPonderada** e digite o código abaixo:

```
unit CLMediaPonderada;

interface

uses Classes, CLMedia;

type
  TMediaPonderada = class(TMedia)
  protected
    FPeso1 : integer;
    FPeso2 : integer;
    procedure SetPeso1(valor : integer);
    procedure SetPeso2(valor : integer);
  public
    constructor Create;
    function CalcularMedia : real; override;

    property Peso1 : integer read FPeso1 write SetPeso1;
    property Peso2 : integer read FPeso2 write SetPeso2;
  end;

implementation

{ TMediaPonderada }

function TMediaPonderada.CalcularMedia: real;
var rMedia : real;
    iSomaPesos : integer;
begin
  rMedia := ( Valor1 * Peso1 ) + ( Valor2 * Peso2 );
  iSomaPesos := Peso1 + Peso2;

  rMedia := rMedia / iSomaPesos;
  Result := rMedia;
end;

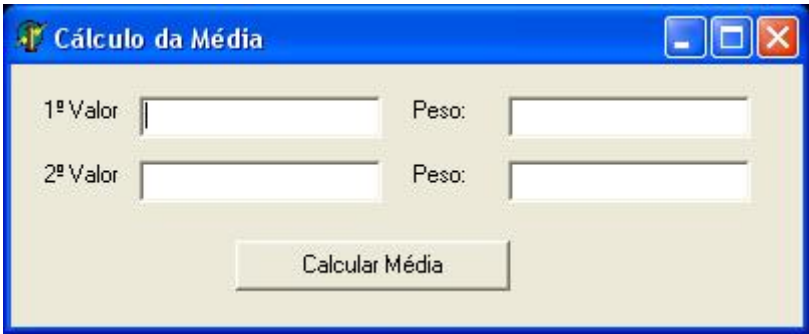
constructor TMediaPonderada.Create;
begin
  inherited;
  FPeso1 := 1;
  FPeso2 := 1;
end;

procedure TMediaPonderada.SetPeso1(valor: integer);
begin
  if valor > 1 then
    FPeso1 := valor
  else
    FPeso1 := 1;
end;

procedure TMediaPonderada.SetPeso2(valor: integer);
begin
  if valor > 1 then
    FPeso2 := valor
  else
    FPeso2 := 1;
end;

end.
```

Vamos criar, agora a interface que usará essa classe. Retorne ao formulário do projeto e coloque os componentes conforme a figura abaixo. Salve o formulário com o nome de **DLMedia**.



Mude a propriedade **name** dos seguintes componentes:

- (edit 1º valor) – edtValor1
- (edit 2º valor) – edtValor2
- (edit 1º Peso) – edtPeso1
- (edit 2º Peso) – edtPeso2
- (botão) – btnMedia
- (formulário) – frmMedia

Vamos passar para a unit do formulário. Vamos ligar essa Unit com as Classes Media e MediaPonderada, conforme código a seguir:

```
implementation

uses CLMedia, CLMediaPonderada;

{$R *.dfm}
```

Volte ao formulário e acione o evento **OnClick** do botão. Nele digite o código abaixo:

```
procedure TfrmMedia.btnMediaClick(Sender: TObject);
var oMedia : TMedia;
    rMedia : real;
begin
    // verifica se o cálculo tem peso ou nao
    try
        if (edtPeso1.Text = '') and (edtPeso2.Text = '') then
            begin
                oMedia := TMedia.Create;
                oMedia.Valor1 := StrToFloat(edtValor1.Text);
                oMedia.Valor2 := StrToFloat(edtValor2.Text);
            end
        else
            begin
                oMedia := TMediaPonderada.Create;
```

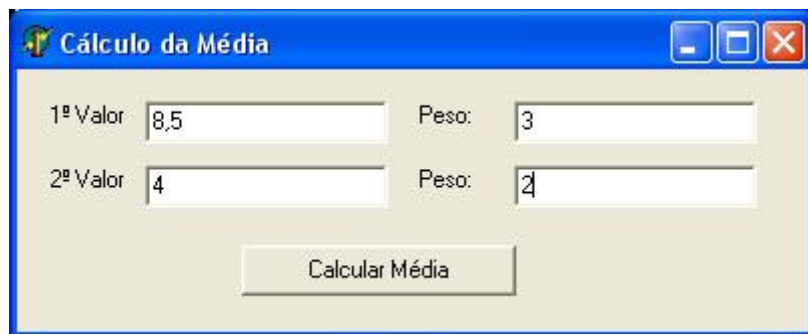
```

with (oMedia as TMediaPonderada) do
begin
    Valor1 := StrToFloat(edtValor1.Text);
    Valor2 := StrToFloat(edtValor2.Text);
    Peso1 := StrToInt(edtPeso1.Text);
    Peso2 := StrToInt(edtPeso2.Text);
end;
end;

rMedia := oMedia.CalcularMedia;
ShowMessage('A média é ' + FloatToStr(rMedia));
except
    ShowMessage('Problemas na associação dos valores');
end;
end;

```

Ao executar a aplicação, se digitarmos os valores conforme mostrado na figura abaixo, receberemos a mensagem “A média é 6,7”. Se limpamos os valores de Peso 1 e 2, e clicarmos em Calcular Média, o programa mostrará a mensagem “A média é 6,25”;



Label	Value
1º Valor	8,5
Peso:	3
2º Valor	4
Peso:	2

Calcular Média

## Capítulo 3

### *Implementando OO em Camadas*

Para melhor entendimento sobre a implementação de um sistema em camadas, vamos tomar como base um programa para **Controle de Vendas de uma loja de R\$ 1,99**. Nesse caso não controlamos os produtos vendidos.

A partir da análise dos requisitos (que aqui não estão explicitamente descritos), chegamos à seguinte lista (simplificada) de **casos de uso**:

- Cadastrar vendedores
- Lançar Vendas
- Consultar vendedores
- Consultar comissões
- Emitir faturamento
- Emitir folha de Pagamento de Comissões

Com isso já é possível imaginar a seguinte **estrutura de menu**:

#### **Cadastro**

Vendedor  
Vendas

#### **Consulta**

Vendedores  
Comissões

#### **Relatório**

Emitir Faturamento  
Emitir Folha de Pagamento de Comissões

Cada item de menu estará chamando uma tela, que pode pertencer à própria aplicação ou a classe. Se a tela pertencer à própria aplicação, ela será uma **classe visual** que se encarregará de coletar as informações do usuário e transmitir às classes, por meio de troca de mensagens.

### *Modelagem em Bancos Relacionais*

No capítulo anterior foi apresentado uma pequena amostra do mapeamento de modelos conceituais em códigos orientados a objetos. E é lógico que a próxima preocupação seja: **como persistir esses objetos?**

A primeira resposta seria: por meio dos bancos de objetos. Esses bancos permitem armazenar mais do que somente textos, mas também imagens, sons, vídeo, etc. Todavia, os bancos de objetos ainda estão buscando “um lugar ao sol” no mercado. Mais é certo que muito se evoluiu nos últimos anos. E já

encontramos depoimentos de produtos estáveis, oferecendo performance e segurança. São alguns dos bancos de objeto de mercado: **Cachê** (InterSystems [www.intersystems.com.br](http://www.intersystems.com.br)) , **Jasmine** da Computer Associates ([www.ca.com](http://www.ca.com)) e **Gemstone** da Gemstone ([www.gemstone.com](http://www.gemstone.com)). Esperamos ansiosamente que eles completem a travessia (já percorria há muito tempo pelos bancos relacionais) e conquistem a confiança dos usuários.

Além disso, lutamos com a resistência do mercado em migrar para novas tecnologias. Certo é que não podemos esperar de nossos usuários a mudança de todo o seu ambiente. A aceitação da mudança de paradigma é bem fácil se, nesse primeiro momento, não tocarmos em suas bases de dados. A performance e segurança fornecida pelos bancos de dados relacionais, nos levam a implementar nossas aplicações (desenvolvidas sobre a ótica da orientação a objetos) nesses bancos de dados. Essa decisão, é a solução mais utilizada, visto que se não tivéssemos seguido esse caminho, em que estágio estaria hoje a orientação a objetos?

Não existem mecanismos nativos nos bancos relacionais que permitam a implementação de conceitos próprios da orientação a objetos, como: **herança e polimorfismo**. Acostumamos-nos a enxergar **a ligação** entre tabelas dos modelos relacionais **por meio de chaves primárias e secundárias**. Já **a ligação** entre objetos é **feita por referência**, visto eles possuírem uma identidade única. Para conseguirmos, então, que os objetos sejam persistidos em bancos relacionais fazer com que tabelas representem os objetos.

Para se conseguir mapear modelos de objetos em tabelas relacionais, precisamos atender algumas necessidades ou seguir algumas orientações, descritas a seguir:

- Cada classe é mapeada diretamente em uma ou mais tabelas do banco de dados. Para cada tabela derivada devemos ter uma chave primária. Alguns atributos podem se tornar chaves primárias desde que cumpram os pré-requisitos necessários para serem uma chave primária. Caso contrário, devemos criar um identificador para exercer tal papel.
- Quanto aos relacionamentos de associação, podemos ter ou não o mapeamento de uma tabela. A exceção é a associação que possua multiplicidade de muitos para muitos. Esse caso sempre gerará uma tabela. O mapeamento da agregação segue regras da associação.

Vejamos quais seriam as **tabelas geradas** a partir das classes para o nosso programa de vendas:

Vendedor	Venda
IDVendedor	IDVenda
Matricula	IDVendedor
Nome	Numero
SalarioBruto	Data
PercentualComissao	Valor



### Modelando em Camadas

A arquitetura de sistemas em camadas, aplicada às aplicações desenvolvidas sobre a orientação a objetos, **determina a distribuição** das classes por três camadas.

A **primeira camada** identificada como **apresentação** abrange as classes que são responsáveis pela **interação dos usuários** com o próprio sistema. Por exemplo: No Delphi, quando projetamos um formulário para receber os dados do usuário e transmitir às nossas classes por meio de solicitação de serviços, estamos na **camada de apresentação**. O formulário é uma **classe visual**. (View)

A **segunda camada** identificada como **camada de negócios controla as regras de negócios** de nossa aplicação. São as nossas classes de negócios, aquelas que armazenam nossos requisitos, as que determinam **como calcular, como processar, o que persistir e o que recuperar**. Podemos dizer que é alma da nossa aplicação. (Controller)

A **terceira camada** é identificada como **camada de persistência controla a persistência** de nossos dados. Devemos lembrar que nossos dados vivem temporariamente nas instâncias das classes da segunda camada. Todavia, necessitamos que esses dados sejam mantidos ao término da aplicação. Para isso, precisamos armazená-lo em um meio físico permanente. Assim, nessa camada tratamos como os dados são persistidos em simples arquivos ou em banco de dados (situação mais freqüente). (Model)

Assim já temos em mãos conhecimentos para montar a arquitetura de nossa aplicação em três camadas.

### Implementando a Aplicação em 3 Camadas

Vamos montar a tela principal de nossa aplicação para poder a partir dela, chamar rotinas que caracterizarão nossa “primeira camada”.

Inicie uma nova aplicação, que corresponderá ao nosso Programa de Vendas. No **formulário principal**, defina um menu com a seguinte estrutura:

<b>Cadastro</b>
Vendedor
Vendas
<b>Consulta</b>
Vendedores
Comissões
<b>Relatório</b>
Emitir Faturamento
Emitir Folha de Pagamento de Comissões

Para os **eventos OnClick** dos itens “Cadastro de Vendedor”, “Cadastro de Vendas” e “Consulta de Comissões”, **digite** o código abaixo. Não esqueça de **colocar** no **início da seção Implementation**, a cláusula uses apontando para as referidas units.

```
unit MenuPrincipal;

interface

uses
  Windows, Messages, SysUtils, Variants, Classes, Graphics, Controls,
  Forms,
  Dialogs, Menus;

type
  TfrmMenu = class(TForm)
    MainMenu1: TMainMenu;
    Cadastro1: TMenuItem;
    Vendedor1: TMenuItem;
    Vendas1: TMenuItem;
    Consulta1: TMenuItem;
    Vendedores1: TMenuItem;
    Comisses1: TMenuItem;
    Relatrio1: TMenuItem;
    Emitir1: TMenuItem;
    EmitirFolhadePagamentodeComisses1: TMenuItem;
    Sair1: TMenuItem;
    procedure Vendedor1Click(Sender: TObject);
    procedure Vendas1Click(Sender: TObject);
    procedure Sair1Click(Sender: TObject);
    procedure Comisses1Click(Sender: TObject);
  private
    { Private declarations }
  public
    { Public declarations }
  end;

var
  frmMenu: TfrmMenu;

implementation

{$R *.dfm}
uses
  CDVendedores, CDVendas, CNComissoes;

procedure TfrmMenu.Vendedor1Click(Sender: TObject);
var
  formCadastro : TfrmCadastroVendedores;
begin
  formCadastro : TfrmCadastroVendedores.Create(Application);
end;

procedure TfrmMenu.Vendas1Click(Sender: TObject);
var
  formCadastro : TfrmCadastroVendas;
begin
  formCadastro : TfrmCadastroVendas.Create(Application);
end;
```

```

procedure TfrmMenu.Sair1Click(Sender: TObject);
begin
    Close;
end;

procedure TfrmMenu.Comisses1Click(Sender: TObject);
var
    formConsulta : TfrmConsultaComissoes;
begin
    formConsulta := TfrmConsultaComissoes.Create(Application);
end;

end.

```

### Implementando a Primeira **Camada (Interface)**

Vamos montar agora as telas de “Cadastrar Vendedores”, “Cadastrar Vendas” e “Consultar Comissões”. Em seguida, vamos escrever o código de cada rotina.

#### **Cadastrar Vendedores**

Va em **File – New – Form** e **desenhe** a tela descrita na figura abaixo que será o nosso cadastro de vendedores.

Agora estamos em condição de criar o código por trás da tela exibida na figura acima. **Digite** o código abaixo para a tela acima:

```

unit CDVendedores;

interface

uses
    Windows, Messages, SysUtils, Variants, Classes, Graphics, Controls,
    Forms,
    Dialogs, StdCtrls, IDGlobal;

type
    TfrmVendedores = class(TForm)

```

```

    Label1: TLabel;
    Label2: TLabel;
    Label3: TLabel;
    Label4: TLabel;
    edtMatricula: TEdit;
    edtNome: TEdit;
    edtSalBruto: TEdit;
    edtPercentualComissao: TEdit;
    btnGravar: TButton;
    btnCancelar: TButton;
    procedure btnCancelarClick(Sender: TObject);
    procedure edtMatriculaChange(Sender: TObject);
    procedure FormCreate(Sender: TObject);
    procedure FormDestroy(Sender: TObject);
    procedure btnGravarClick(Sender: TObject);
    procedure edtSalBrutoKeyPress(Sender: TObject; var Key: Char);
    procedure edtPercentualComissaoKeyPress(Sender: TObject;
      var Key: Char);
    procedure edtSalBrutoExit(Sender: TObject);
    procedure edtPercentualComissaoExit(Sender: TObject);
    procedure FormClose(Sender: TObject; var Action: TCloseAction);
  private
    { Private declarations }
  public
    { Public declarations }
end;

var
  frmVendedores: TfrmVendedores;

implementation

{$R *.dfm}

uses ClVendedor;

var oVendedor : TVendedor;

procedure TfrmVendedores.btnCancelarClick(Sender: TObject);
begin
  close;
end;

procedure TfrmVendedores.edtMatriculaChange(Sender: TObject);
begin
  if Length(edtMatricula.Text) <> 4 then
  begin
    edtNome.Clear;
    edtSalBruto.Clear;
    edtPercentualComissao.Clear;
    oVendedor.Clear;
  end
  else
  begin
    if oVendedor.Busca(edtMatricula.Text) then
    begin
      edtNome.Text := oVendedor.Nome;
      edtSalBruto.Text := FloatToStr(oVendedor.SalarioBruto);
      edtPercentualComissao.Text :=
        FloatToStr(oVendedor.PercentualComissao);
    end;
  end;
end;

```

```

procedure TfrmVendedores.FormCreate(Sender: TObject);
begin
    oVendedor := TVendedor.Create;
end;

procedure TfrmVendedores.FormDestroy(Sender: TObject);
begin
    oVendedor.Free;
end;

procedure TfrmVendedores.btnGravarClick(Sender: TObject);
begin
    if (edtMatricula.Text <> '') and
        (edtNome.Text <> '') and
        (edtSalBruto.Text <> '') and
        (edtPercentualComissao.Text <> '') then
    begin
        oVendedor.Matricula := edtMatricula.Text;
        oVendedor.Nome := edtNome.Text;
        oVendedor.SalarioBruto := StrToFloat(edtSalBruto.Text);
        oVendedor.PercentualComissao :=
StrToFloat(edtPercentualComissao.Text);

        if oVendedor.Grava then
        begin
            edtMatricula.Clear;
            edtNome.Clear;
            edtSalBruto.Clear;
            edtPercentualComissao.Clear;
            edtMatricula.SetFocus;
            oVendedor.Clear;
        end
        else
            edtNome.SetFocus;
        end
    else
        ShowMessage('Campos inválidos');
    end;

procedure TfrmVendedores.edtSalBrutoKeyPress(Sender: TObject;
    var Key: Char);
begin
    if not (IsNumeric(Key) or (Key = TAB) or
        (Key = ',') or (Key = BACKSPACE)) THEN
        Key := #0;
end;

procedure TfrmVendedores.edtPercentualComissaoKeyPress(Sender: TObject;
    var Key: Char);
begin
    if not (IsNumeric(Key) or (Key = TAB) or
        (Key = ',') or (Key = BACKSPACE)) THEN
        Key := #0;
end;

procedure TfrmVendedores.edtSalBrutoExit(Sender: TObject);
begin
    if edtSalBruto.Text <> '' then
    begin
        try
            StrToFloat(edtSalBruto.Text);
        except
            begin

```

```

        ShowMessage('Valor inválido');
        edtSalBruto.SetFocus;
    end;
end
end;
end;

procedure TfrmVendedores.edtPercentualComissaoExit(Sender: TObject);
begin
    if edtPercentualComissao.Text <> '' then
    begin
        try
            StrToFloat(edtPercentualComissao.Text);
        except
            begin
                ShowMessage('Valor inválido');
                edtPercentualComissao.SetFocus;
            end;
        end
    end;
end;

end;

procedure TfrmVendedores.FormClose(Sender: TObject;
    var Action: TCloseAction);
begin
    action := caFree;
end;

end.

```

## ***Cadastrar Vendas***

Vá em **File – New – Form** e **desenhe** a tela descrita na figura abaixo, que será o nosso cadastro de vendas.

Vejamos o código escrito por trás da tela de Cadastro de Vendas:

```

unit CDVendas;

interface

uses
  Windows, Messages, SysUtils, Variants, Classes, Graphics, Controls,
  Forms,
  Dialogs, ComCtrls, StdCtrls, IDGlobal;

type
  TfrmVendas = class(TForm)
    Label1: TLabel;
    Label2: TLabel;
    Label3: TLabel;
    Label4: TLabel;
    edtValor: TEdit;
    edtNumero: TEdit;
    cmbVendedor: TComboBox;
    edtData: TDateTimePicker;
    btnGravar: TButton;
    btnCancelar: TButton;
    procedure btnCancelarClick(Sender: TObject);
    procedure edtNumeroExit(Sender: TObject);
    procedure FormCreate(Sender: TObject);
    procedure FormDestroy(Sender: TObject);
    procedure btnGravarClick(Sender: TObject);
    procedure edtValorKeyPress(Sender: TObject; var Key: Char);
    procedure edtNumeroKeyPress(Sender: TObject; var Key: Char);
    procedure FormClose(Sender: TObject; var Action: TCloseAction);
  private
    { Private declarations }
  public
    { Public declarations }
  end;

var
  frmVendas: TfrmVendas;

implementation

{$R *.dfm}

uses CLVenda, CLVendedor;

var oVenda : TVenda;

procedure TfrmVendas.btnCancelarClick(Sender: TObject);
begin
  Close;
end;

procedure TfrmVendas.edtNumeroExit(Sender: TObject);
begin
  if edtNumero.Text = '' then
  begin
    edtData.Date := Date;
    edtValor.Clear;
    cmbVendedor.Text := '';
  end
  else
  begin
    if oVenda.Busca(StrToInt(edtNumero.Text)) then
    begin

```

```

        edtData.Date := oVenda.DataVenda;
        edtValor.Text := FloatToStr(oVenda.Valor);
        cmbVendedor.Text := oVenda.Vendedor.Nome;
    end;
end;
end;

procedure TfrmVendas.FormCreate(Sender: TObject);
var
    lstVendedores : TList;
    oVendedor : TVendedor;
    iAux : integer;
begin
    oVenda := TVenda.Create;
    lstVendedores := TVendedor.obterListaVendedoresAtivos;

    for iAux := 0 to lstVendedores.Count - 1 do
    begin
        oVendedor := lstVendedores[iAux];
        cmbVendedor.AddItem(oVendedor.Nome, oVendedor);
    end;
end;

procedure TfrmVendas.FormDestroy(Sender: TObject);
begin
    OVenda.Free;
end;

procedure TfrmVendas.btnGravarClick(Sender: TObject);
begin
    if (edtNumero.Text <> '') and
        (edtData.Date <> 0) and
        (edtValor.Text <> '') and
        (cmbVendedor.Text <> '') then
    begin
        oVenda.Numero := StrToInt(edtNumero.Text);
        oVenda.DataVenda := edtData.Date;
        oVenda.Valor := StrToFloat(edtValor.Text);

        oVenda.Vendedor :=
            (cmbVendedor.Items.Objects[cmbVendedor.ItemIndex] as TVendedor);

        if oVenda.Grava then
        begin
            edtNumero.Clear;
            edtData.Date := Date;
            edtValor.Clear;
            cmbVendedor.Text := '';
            edtNumero.SetFocus;
            oVenda.Clear;
        end
        else
            edtData.SetFocus;
    end
    else
    begin
        ShowMessage('Campos inválidos');
    end;
end;
end;

```



```

procedure TfrmVendas.edtValorKeyPress(Sender: TObject; var Key: Char);
begin
    if not (IsNumeric(Key) or (Key = TAB) or
        (Key = ',') or (Key = BACKSPACE)) THEN
        Key := #0;
end;

procedure TfrmVendas.edtNumeroKeyPress(Sender: TObject; var Key: Char);
begin
    if not (IsNumeric(Key) or (Key = TAB) or
        (Key = ',') or (Key = BACKSPACE)) THEN
        Key := #0;
end;

procedure TfrmVendas.FormClose(Sender: TObject; var Action:
TCloseAction);
begin
    action := caFree;
end;

end.

```

### Consultar Comissões

Vá em **File – New – Form** e **desenhe** a tela descrita na figura abaixo, que será a tela de consulta de comissões:

Vejamos o código escrito por trás da tela de Consulta de Comissões:

```

unit CNComissoes;

interface

uses
  Windows, Messages, SysUtils, Variants, Classes, Graphics, Controls,
  Forms,
  Dialogs, ComCtrls, StdCtrls;

type
  TFrmConsultaComissoes = class(TForm)
    Label1: TLabel;
    cmbVendedor: TComboBox;
    Label2: TLabel;
    edtData1: TDateTimePicker;
    Label3: TLabel;
    edtData2: TDateTimePicker;
    btnVerificar: TButton;
    Label4: TLabel;
    Label5: TLabel;
    edtSalBruto: TEdit;
    edtSalLiquido: TEdit;
    Label6: TLabel;
    edtComissao: TEdit;
    procedure FormClose(Sender: TObject; var Action: TCloseAction);
    procedure FormCreate(Sender: TObject);
    procedure btnVerificarClick(Sender: TObject);
  private
    { Private declarations }
  public
    { Public declarations }
  end;

var
  FrmConsultaComissoes: TFrmConsultaComissoes;

implementation

{$R *.dfm}

uses CLVendedor;

procedure TFrmConsultaComissoes.FormClose(Sender: TObject; var Action:
TCloseAction);
begin
  action := caFree;
end;

procedure TFrmConsultaComissoes.FormCreate(Sender: TObject);
var
  lstVendedores : TList;
  oVendedor : TVendedor;
  iAux : integer;
begin
  lstVendedores := TVendedor.obterListaVendedoresAtivos;

  for iAux := 0 to lstVendedores.Count - 1 do
    begin
      oVendedor := lstVendedores[iAux];
      cmbVendedor.AddItem(oVendedor.Nome, oVendedor);
    end;
  end;
end;

```

```

procedure TFrmConsultaComissoes.btnVerificarClick(Sender: TObject);
var
  oVendedor : TVendedor;
  rSalLiq, rSalBruto, rComissao: real;
begin
  if (cmbVendedor.Text <> '') and (edtData1.Date <> 0) and
    (edtData2.Date <> 0) then
    begin
      oVendedor := (cmbVendedor.Items.Objects[cmbVendedor.ItemIndex] as
TVendedor);

      rSalBruto := oVendedor.SalarioBruto;
      rSalLiq := oVendedor.CalcularSalarioLiquido(edtData1.DateTime,
edtData2.DateTime);

      rComissao := rSalLiq - rSalBruto;

      edtSalBruto.Text := FloatToStr(rSalBruto);
      edtComissao.Text := FloatToStr(rComissao);
      edtSalLiquido.Text := FloatToStr(rSalLiq);

    end;
end;
end.

```

## Implementação da Segunda Camada (Negócios)

Vejam como fica a implementação de nossas classes. Abra uma nova unit e **salve** com o nome de CLVendedor. Veja como fica o código na figura abaixo:

```

unit CLVendedor;

interface

uses
  Classes, DateUtils;

type
  TVendedor = class
  protected
    FId : integer;
    FMatricula : string;
    FNome : string;
    FSalarioBruto : real;
    FSalarioLiquido : real;
    FPercentualComissao : real;
    procedure SetSalarioBruto(rSalario : real);
    procedure SetPercentualComissao(rPercentual : real);
    function obterTotalVendas(dataRef1, dataRef2 : TDateTime) : real;
  public
    constructor Create;
    function CalcularSalarioLiquido(dataRef1, dataRef2 : TDateTime) :
real;
    function Busca(sMatricula : string) : boolean;
    function BuscaMatricula(iID : integer):boolean;

```

```

class function obterListaVendedoresAtivos : TList;
function Grava : boolean;
procedure Clear;

property ID : Integer          read FId;
property Matricula : string    read FMatricula      write
FMatricula;
property Nome : string         read FNome           write
FNome;
property SalarioBruto : real    read FSalarioBruto   write
SetSalarioBruto;
property PercentualComissao:real read FPercentualComissao
write SetPercentualComissao;

end;

implementation

{ TVendedor }

uses CLVenda, DMVendedor, DB;

function TVendedor.Busca(sMatricula: string): boolean;
var iID:integer;
begin
    if dtmVendedor.Busca(sMatricula, self, iID) then
    begin
        result := true;
        FId := iID;
    end
    else
        Result := false;
    end;
end;

function TVendedor.BuscaMatricula(iID: integer): boolean;
begin
    if dtmVendedor.BuscaMatricula(iId, Self) then
    begin
        Result := true;
        FId := iID;
    end
    else
        Result := false;
    end;
end;

function TVendedor.CalcularSalarioLiquido(dataRef1,      dataRef2:
TDateTime): real;
var rTotalVendas : real;
begin
    rTotalVendas := Self.obterTotalVendas(dataRef1, dataRef2);

    FSalarioLiquido := FSalarioBruto + (rTotalVendas *
FPercentualComissao);
    Result := FSalarioLiquido;
end;

procedure TVendedor.Clear;
begin
    FId := 0;
    FMatricula := '';
    FNome := '';
    FSalarioBruto := 0;
    FPercentualComissao := 0;
end;

```

```

constructor TVendedor.Create;
begin
    inherited;
    FSalarioBruto := 0;
    FSalarioLiquido := 0;
    FPercentualComissao := 0;
end;

function TVendedor.Grava: boolean;
begin
    dtmVendedor.Grava(Self);
end;

class function TVendedor.obterListaVendedoresAtivos: TList;
var lstVendedores : TList;
    oVendedor: TVendedor;
begin
    lstVendedores := TList.Create;

    dtmVendedor.obterVendedoresAtivos;

    with dtmVendedor.gryAcesso do
    begin
        if RecordCount > 0 then
        begin
            first;
            while not eof do
            begin
                oVendedor := TVendedor.Create;
                oVendedor.FId := FieldByName('IdVendedor').AsInteger;
                oVendedor.FMatricula := FieldByName('Matricula').AsString;
                oVendedor.FNome := FieldByName('Nome').AsString;
                oVendedor.FSalarioBruto :=
                    FieldByName('SalarioBruto').AsFloat;
                oVendedor.FPercentualComissao :=
                    FieldByName('PercentualComissao').AsFloat;

                lstVendedores.Add(oVendedor);
                Next;
            end;
        end;
    end;

    Result := lstVendedores;

end;

function TVendedor.obterTotalVendas(dataRef1, dataRef2: TDateTime):
real;
var
    iAux : integer;
    rTotal : real;
    lstVendas : TList;
    oVenda : TVenda;
begin
    lstVendas := TList.Create;
    lstVendas := TVenda.ObterTotalVendas(dataRef2, dataRef2, self.ID);
    rTotal := 0;

    for iAux := 0 to lstVendas.Count - 1 do
    begin
        oVenda := lstVendas[iAux];
        rTotal := rTotal + oVenda.Valor;
        oVenda.Free;
    end;
end;

```

```

    lstVendas.Free;

    Result := rTotal;

end;

procedure TVendedor.SetPercentualComissao(rPercentual: real);
begin
    if rPercentual > 0 then
        if rPercentual > 1 then
            FPercentualComissao := rPercentual / 100
        else
            FPercentualComissao := rPercentual
        else
            FPercentualComissao := 0;
    end;
end;

procedure TVendedor.SetSalarioBruto(rSalario: real);
begin
    if rSalario >= 0 then
        FSalarioBruto := rSalario
    else
        FSalarioBruto := 0;
end;

end.

```

Abra uma nova unit e salve com o nome de CLVendas. Veja o código na Figura abaixo:

```

unit CLVenda;

interface

uses
    Classes, CLVendedor;

type
    TVenda = class
    protected
        FID      : integer;
        FNumero  : integer;
        FData    : TDateTime;
        FValor   : real;
        FVendedor : TVendedor;

        procedure SetNumero(iNumero : integer);
        procedure SetValor(rValor : real);
    public
        constructor Create;
        destructor Destroy; override;
        function Busca(iNumero : integer) : boolean;
        function Grava : boolean;
        procedure Clear;

        class function ObterTotalVendas(dataRef1, dataRef2 : TDateTime;
                                         idVendedor : integer) : TList;
    end;

```

```

        property ID : integer           read FID;
        property Numero : integer       read FNumero write SetNumero;
        property DataVenda : TDateTime read FData   write FData;
        property Valor : real           read FValor  write SetValor;
        property Vendedor : TVendedor  read FVendedor write FVendedor;
    end;

implementation

{ TVenda }

uses DMVenda, DB;

function TVenda.Busca(iNumero: integer): boolean;
var iId : integer;
begin
    if dtmVenda.Busca(iNumero, Self, iId) then
    begin
        result := true;
        FID := iId;
    end
    else
        result := false;
    end;

end;

procedure TVenda.Clear;
begin
    FID := 0;
    FNumero := 0;
    FData := 0;
    FValor := 0;
    FVendedor.Clear;
end;

constructor TVenda.Create;
begin
    FVendedor := TVendedor.Create;
end;

destructor TVenda.Destroy;
begin
    FVendedor.Free;
    inherited;
end;

function TVenda.Grava: boolean;
begin
    result := dtmVenda.Grava(Self);
end;

class function TVenda.ObterTotalVendas(dataRef1, dataRef2: TDateTime;
    idVendedor: integer): TList;

var lstVendas : TList;
    oVenda : TVenda;
begin
    lstVendas := TList.Create;

    dtmVenda.ObterTotalVendas(dataRef1, dataRef2, idVendedor);

    with dtmVenda.qryAcesso do
    begin
        if RecordCount <> 0 then
        begin

```

```

First;
while not eof do
begin
    oVenda := TVenda.Create;
    oVenda.Numero := FieldByName('Numero').AsInteger;
    oVenda.DataVenda := FieldByName('Data').AsDateTime;
    oVenda.Valor := FieldByName('Valor').AsFloat;

    oVenda.Vendedor.BuscaMatricula(FieldByName('IDVendedor').AsInteger);
    oVenda.FID := FieldByName('idvenda').AsInteger;

    lstVendas.Add(oVenda);
    Next;

end;
end;
end;
Result := lstVendas;
end;

procedure TVenda.SetNumero(iNumero: integer);
begin
    if iNumero >= 0 then
        FNumero := iNumero
    else
        FNumero := 0;
end;

procedure TVenda.SetValor(rValor: real);
begin
    if rValor >= 0 then
        FValor := rValor
    else
        FValor := 0;
end;

end.

```

### Implementação da Terceira Camada (**Persistência**)

A terceira camada cuida da gravação dos dados em nosso banco de dados. Para simplificar, usaremos o banco Firebird e conexão via IBX.

Crie um DataModule, **salve-o** como DMVendedor e **renomeie** o Data Module para dtmVendedor. **Coloque** um componente TIBDatabase, TIBTransaction e um TIBQuery.

Para cada método da classe que necessite acessar ou atualizar os dados no banco de dados, é feito um repasse para procedures ou funções do Data Module. Por exemplo: o **método Busca** da classe chama a **função busca** do Data Module. O Data Module se encarrega de acessar o banco de dados e preencher o objeto a ser devolvido.





Veja o código do **Data Module DMVendedor** abaixo, e em seguida, o código do **Data Module DMVendas**.

```
unit DMVendedor;

interface

uses
  SysUtils, Classes, IBDatabase, DB, IBCustomDataSet, IBQuery,
  CLVendedor, Dialogs;

type
  TdtmVendedor = class(TDataModule)
    IBConexao: TIBDatabase;
    qryAcesso: TIBQuery;
    IBTransaction: TIBTransaction;
  private
    { Private declarations }
  public
    function Grava(oVendedor : TVendedor) : boolean;
    function Busca(sMatricula : string; oVendedor : TVendedor;
      var Id: integer) : boolean;
    function BuscaMatricula(iId : integer; oVendedor : TVendedor) :
      Boolean;
    procedure obterVendedoresAtivos;
    { Public declarations }
  end;

var
  dtmVendedor: TdtmVendedor;

implementation

{$R *.dfm}

{ TdtmVendedor }

function TdtmVendedor.Busca(sMatricula: string; oVendedor: TVendedor;
  var Id: integer): boolean;
begin
  Result := false;
  try
    if not IBConexao.Connected then
    begin
      IBConexao.Open;
      IBTransaction.StartTransaction;
    end;

    if qryAcesso.Active then
      qryAcesso.Close;

    qryAcesso.SQL.Clear;
    qryAcesso.SQL.Add('select ven."IDVendedor", ven."Matricula",
ven."Nome", ven."SalarioBruto", ven."PercentualComissao"');
    qryAcesso.SQL.Add('from vendedor ven');
    qryAcesso.SQL.Add('where ven."Matricula" = :prmMat');

    qryAcesso.Params.ParamByName('prmMat').Value := sMatricula;

    qryAcesso.Open;

    if qryAcesso.RecordCount <> 0 then
      begin
```

```

        Result := true;
        oVendedor.Matricula :=
qryAcesso.FieldByName('Matricula').AsString;
        oVendedor.Nome := qryAcesso.FieldByName('Nome').AsString;
        oVendedor.SalarioBruto :=
qryAcesso.FieldByName('SalarioBruto').AsFloat;
        oVendedor.PercentualComissao :=
qryAcesso.FieldByName('PercentualComissao').AsFloat;

        Id := qryAcesso.FieldByName('IDVendedor').AsInteger;
    end;
except on e:exception do
    ShowMessage('Problemas na conexão com a base de dados ' + #13 +
e.Message);
end
end;

function TdtmVendedor.BuscaMatricula(iId: integer;
oVendedor: TVendedor): Boolean;
begin
    result := false;
    try
        if not IBConexao.Connected then
            begin
                IBConexao.Open;
                IBTransaction.StartTransaction;
            end;

            if qryAcesso.Active then
                qryAcesso.Close;

                qryAcesso.SQL.Clear;
                qryAcesso.SQL.Add('select ven."IDVendedor", ven."Matricula",
ven."Nome", ven."SalarioBruto", ven."PercentualComissao"');
                qryAcesso.SQL.Add('from vendedor ven');
                qryAcesso.SQL.Add('where ven."IDVendedor" = :prmId');

                qryAcesso.Params.ParamByName('prmId').Value := iId;

                qryAcesso.Open;

                if qryAcesso.RecordCount <> 0 then
                    begin
                        Result := true;
                        oVendedor.Matricula :=
qryAcesso.FieldByName('Matricula').AsString;
                        oVendedor.Nome := qryAcesso.FieldByName('Nome').AsString;
                        oVendedor.SalarioBruto :=
qryAcesso.FieldByName('SalarioBruto').AsFloat;
                        oVendedor.PercentualComissao :=
qryAcesso.FieldByName('PercentualComissao').AsFloat;
                    end;
                except on e:exception do
                    ShowMessage('Problemas na conexão com a base de dados ' + #13 +
e.Message);
                end
            end;
        end;

function TdtmVendedor.Grava(oVendedor: TVendedor): boolean;
begin
    result := false;
    try
        if not IBConexao.Connected then
            begin

```

```

        IBConexao.Open;
        IBTransaction.StartTransaction;
    end;

    if qryAcesso.Active then
        qryAcesso.Close;

    qryAcesso.SQL.Clear;

    if oVendedor.ID <> 0 then
    begin
        qryAcesso.SQL.Add('update vendedor ven');
        qryAcesso.SQL.Add('set ven."Matricula" = :prmMat,');
        qryAcesso.SQL.Add('    ven."Nome" = :prmNome,');
        qryAcesso.SQL.Add('    ven."SalarioBruto" = :prmSal,');
        qryAcesso.SQL.Add('    ven."PercentualComissao" = :prmPerc');
        qryAcesso.SQL.Add('where ven."IDVendedor" = :prmId');
        qryAcesso.Params.ParamByName('prmId').Value := oVendedor.ID;
        qryAcesso.Params.ParamByName('prmMat').Value :=
oVendedor.Matricula;
        qryAcesso.Params.ParamByName('prmNome').Value :=
oVendedor.Nome;
        qryAcesso.Params.ParamByName('prmSal').Value :=
oVendedor.SalarioBruto;
        qryAcesso.Params.ParamByName('prmPerc').Value :=
oVendedor.PercentualComissao;
    end
    else
    begin
        qryAcesso.SQL.Add('insert into vendedor');
        qryAcesso.SQL.Add('("Matricula", "Nome", "PercentualComissao",
"SalarioBruto")');
        qryAcesso.SQL.Add('values');
        qryAcesso.SQL.Add('(:prmMat, :prmNome, :prmPerc, :prmSal)');
        qryAcesso.Params.ParamByName('prmMat').Value :=
oVendedor.Matricula;
        qryAcesso.Params.ParamByName('prmNome').Value :=
oVendedor.Nome;
        qryAcesso.Params.ParamByName('prmPerc').Value :=
oVendedor.PercentualComissao;
        qryAcesso.Params.ParamByName('prmSal').Value :=
oVendedor.SalarioBruto;
    end;

    qryAcesso.ExecSQL;
    IBTransaction.Commit;
    result := true;
except
    on e:exception do
        ShowMessage('Problemas no acesso ao banco de dados ' + #13 +
e.Message);
    end;
end;

end;

procedure TdtmVendedor.obterVendedoresAtivos;
begin
    try
        if not IBConexao.Connected then
        begin
            IBConexao.Open;
            IBTransaction.StartTransaction;
        end;
        if qryAcesso.Active then
            qryAcesso.Close;

```

```

        qryAcesso.SQL.Clear;
        qryAcesso.SQL.Add('select ven."IDVendedor", ven."Matricula",
ven."Nome", ven."SalarioBruto", ven."PercentualComissao"');
        qryAcesso.SQL.Add('from vendedor ven');

        qryAcesso.Open;

        except on e:exception do
            ShowMessage('Problemas na conexão com a base de dados ' + #13 +
e.Message);
        end

    end;

end.

```

Veja o código do **DMVenda**:

```

unit DMVenda;

interface

uses
    SysUtils, Classes, CLVenda, IBDatabase, DB, IBCustomDataSet, IBQuery,
    Dialogs;

type
    TdtmVenda = class(TDataModule)
        IBConexao: TIBDatabase;
        qryAcesso: TIBQuery;
        IBTransaction: TIBTransaction;
    private
        { Private declarations }
    public
        function Grava(oVenda : TVenda) : Boolean;
        function Busca(iNum : integer; oVenda : TVenda;
            var Id : integer) : boolean;
        procedure ObterTotalVendas(dataRef1, dataRef2:TDateTime; IdVendedor
: integer);
    end;

var
    dtmVenda: TdtmVenda;

implementation

uses CLVendedor;

{$R *.dfm}

{ TdtmVenda }

function TdtmVenda.Busca(iNum: integer; oVenda: TVenda;
    var Id: integer): boolean;
begin
    result := false;

    try
        if not IBConexao.Connected then
            begin
                IBConexao.Open;
                IBTransaction.StartTransaction;

```

```

end;

if qryAcesso.Active then
    qryAcesso.Close;

    qryAcesso.SQL.Clear;
    qryAcesso.SQL.Add('select ven."IDVenda", ven."IDVendedor",
ven."Numero", ven."Data", ven."Valor"');
    qryAcesso.SQL.Add('from venda ven');
    qryAcesso.SQL.Add('where ven."Numero" = :num');

    qryAcesso.Params.ParamByName('num').Value := iNum;

    qryAcesso.Open;

    if qryAcesso.RecordCount <> 0 then
        begin
            result := True;
            oVenda.Numero := iNum;
            oVenda.DataVenda := qryAcesso.FieldByName('Data').AsDateTime;
            oVenda.Valor := qryAcesso.FieldByName('Valor').AsFloat;

oVenda.Vendedor.BuscaMatricula(qryAcesso.FieldByName('IDVendedor').AsInt
eger);
            Id := qryAcesso.FieldByName('IDVenda').AsInteger;
        end;
    except
        on e: Exception do
            ShowMessage('Problemas no acesso ao banco de dados ' + #13 +
E.Message);
        end;
    end;

function IdtmVenda.Grava(oVenda: TVenda): Boolean;
begin
    Result := false;
    try
        if not IBConexao.Connected then
            begin
                IBConexao.Open;
                IBTransaction.StartTransaction;
            end;

        if qryAcesso.Active then
            qryAcesso.Close;

        qryAcesso.SQL.Clear;

        if oVenda.ID <> 0 then
            begin
                qryAcesso.SQL.Add('update venda ven');
                qryAcesso.SQL.Add('set ven."IDVendedor" = :prmVend,');
                qryAcesso.SQL.Add('    ven."Numero" = :prmNum,');
                qryAcesso.SQL.Add('    ven."Data" = :prmData,');
                qryAcesso.SQL.Add('    ven."Valor" = :prmValor');
                qryAcesso.SQL.Add('where (ven."IDVenda" = :prmId)');

                qryAcesso.Params.ParamByName('prmId').Value := oVenda.ID;
                qryAcesso.Params.ParamByName('prmVend').Value :=
oVenda.Vendedor.ID;
                qryAcesso.Params.ParamByName('prmNum').Value := oVenda.Numero;
                qryAcesso.Params.ParamByName('prmData').Value :=
oVenda.DataVenda;
                qryAcesso.Params.ParamByName('prmValor').Value := oVenda.Valor;
            end

```

```

else
begin
    qryAcesso.SQL.Add('insert into venda');
    qryAcesso.SQL.Add('('IDVendedor", "Numero", "Data", "Valor"');
    qryAcesso.SQL.Add('values');
    qryAcesso.SQL.Add('(:prmVend, :prmNum, :prmData, :prmValor)');

    qryAcesso.Params.ParamByName('prmVend').Value :=
oVenda.Vendedor.ID;
    qryAcesso.Params.ParamByName('prmNum').Value := oVenda.Numero;
    qryAcesso.Params.ParamByName('prmData').Value :=
oVenda.DataVenda;
    qryAcesso.Params.ParamByName('prmValor').Value := oVenda.Valor;

    end;

    qryAcesso.ExecSQL;
    IBTransaction.Commit;
    Result := True;

    except on e: exception do
        ShowMessage('Problemas no acesso ao banco de dados ' + #13 +
e.Message);
        end;
    End;

procedure TdtmVenda.ObterTotalVendas(dataRef1, dataRef2: TDateTime;
IdVendedor: integer);
begin
    try
        if not IBConexao.Connected then
            begin
                IBConexao.Open;
                IBTransaction.StartTransaction;
            end;

            if qryAcesso.Active then
                qryAcesso.Close;

            qryAcesso.SQL.Clear;
            qryAcesso.SQL.Add('select ven."IDVenda", ven."IDVendedor",
ven."Numero", ven."Data", ven."Valor"');
            qryAcesso.SQL.Add('from venda ven');
            qryAcesso.SQL.Add('where ("IDVendedor" = :prmNum and ven."Data"
between :prmData1 and :prmData2)');

            qryAcesso.Params.ParamByName('prmNum').Value := IdVendedor;
            qryAcesso.Params.ParamByName('prmData1').Value := dataRef1;
            qryAcesso.Params.ParamByName('prmData2').Value := dataRef2;

            qryAcesso.Open;

        except
            on e: exception do
                ShowMessage('Problemas no acesso ao banco de dados ' + #13 +
e.Message);
                end;
            end;
        end.

```

## **BIBLIOGRAFIA**

MELO, Ana Cristina, Desenvolvendo Aplicações com UML 2.0: do conceitual à implementação / Ana Cristina Melo. – 2. ed. – Rio de Janeiro: Brasport, 2004.

BORLAND, Treinamento Oficial, Delphi 2005 – Application Development with Delphi for Win32 – 2005.

DEITEL, H. M. Deitel, C# - Como Programar – São Paulo: Pearson Education do Brasil LTDA, 2003.