

VCL Revelada



Nossa ferramenta de desenvolvimento já tem mais de 10 anos de vida. Quantos programas foram feitos nesse período! Muitos deles, senão todos, foram feitos com o auxílio de componentes, controles, objetos etc., sejam para apenas exibir um botão na tela ou para fornecer acesso a um banco de dados.

E, independentemente da função dos componentes, a sua utilização (e criação) foi enormemente facilitada através de uma sigla que conhecemos bem: VCL. Mas, como a VCL funciona? Quais classes a sustentam? Entender o seu funcionamento interno é de suma importância para criar novos componentes corretamente, e saber como se comportam os que são nativos do Delphi. Isso tudo veremos neste artigo.

VCL: um pequeno histórico

Em inglês, VCL significa *Visual Component Library*. Em português, *Biblioteca de Componentes Visuais*. Embora o nome faça parecer que a VCL é composta apenas de controles visíveis ao usuário, também há classes que, mesmo não aparecendo no sentido literal do termo, servem como base ou auxiliam as que assim o são.

A VCL existe desde o Delphi 1, lançado em 1995, e assim como a ferramenta, vem sofrendo mudanças e melhorias ao longo do tempo. Sua principal função era abstrair do desenvolvedor o conhecimento de APIs nativas do sistema operacional para a criação de interfaces gráficas (*GUI* ou *Graphical User Interface*).

Esse papel, todos sabemos que a VCL cumpre muito bem, e isso foi um dos grandes atrativos para o Delphi, quando do seu lançamento. Afinal, alguém aqui já precisou escrever *CreateWindow* para criar um formulário?

Hierarquia da VCL

A hierarquia da VCL é absurdamente extensa. Nela es-

tão todos os componentes nativos do Delphi/C++ Builder. Nesse grupo de classes, podemos destacar um subconjunto que serve de base para todos os outros:

- TObject
 - TPersistent
 - ⊙ TComponent
- TControl
 - TGraphicControl
 - TWinControl
 - ⊙ TCustomControl

Veremos agora a função, alguns métodos e propriedades de cada classe.

TObject

Essa classe é a base para todos os objetos na Delphi Language. Todas as classes da VCL herdam dela, direta ou indiretamente. *TObject* provê as funções básicas de todo objeto, como criação, destruição, alocação e liberação de memória. Além disso, também publica informações de *run-time* (RTTI ou *Runtime Type Information*) sobre as propriedades declaradas em sua seção *published*.

Essa classe não possui propriedades nem eventos, apenas métodos, muitos estáticos (também chamados de métodos de classe). Algumas dessas rotinas são bem interessantes em determinadas situações. Vejamos então quais são elas:

AfterConstruction: chamado após o construtor da classe ter sido executado com sucesso. Na implementação original em *TObject* ele não faz nada. Você deve, portanto codificá-lo em uma classe derivada. *TCustomForm*, por exemplo, utiliza esse método para gerar o evento *OnCreate*. Como essa rotina é chamada automaticamente após o construtor da classe, você nunca deve chamá-la explicitamente;

BeforeDestruction: sua função é a oposta do método anterior. É chamado imediatamente antes do destrutor da classe ser executado. Novamente, não chame essa rotina diretamente, pois ela é disparada automaticamente em momento oportuno. É importante notar que o método só será chamado se o construtor da classe for bem sucedido. Isso quer dizer que se uma exceção ocorrer dentro do construtor, o destrutor será chamado para liberar a memória do objeto alocada até o momento do erro, mas *BeforeDestruction* não será executado;

ClassName: retorna o nome da classe do objeto. Por exemplo, um botão retornaria *TButton*, um *Label* retornaria *TLabel* etc. É muito útil quando precisamos diferenciar instâncias de objetos armazenadas em variáveis cujo tipo seja uma classe ancestral;

ClassNameIs: determina se o objeto é do tipo informado. Como o item anterior, é útil quando se faz criação dinâmica de objetos;

ClassParent: retorna a classe ancestral do objeto. Pode ser usado, por exemplo, para verificar a “hereditariedade” de

uma instância. Para uma aplicação prática do que vimos até aqui, inicie uma nova aplicação e coloque um *Button* e um *ListBox* no formulário. Em seguida implemente o evento *OnClick* do botão para com o código da **Listagem 1**.

Listagem 1. ClassName, ClassType e ClassParent

```
procedure TForm1.Button1Click(Sender: TObject);
var
  ClassRef: TClass;
begin
  ListBox1.Clear;
  ClassRef := Sender.ClassType;
  while ClassRef <> nil do
  begin
    ListBox1.Items.Add(ClassRef.ClassName);
    ClassRef := ClassRef.ClassParent;
  end;
end;
```

Após executar a aplicação e clicar no botão, você receberá uma lista como a seguinte, que mostrará toda a hierarquia de classes a partir de *TButton*:

- TButton
- TButtonControl
- TWinControl
- TControl
- TComponent
- TPersistent
- TObject

Dica: Quando possível, evite usar *ClassParent*, dando preferência aos operadores *is* e *as* da Delphi Language.

ClassType: Retorna o tipo da classe do objeto. Geralmente é usado em conjunto com o método anterior. Na **Listagem 1** você pode ver essa função sendo utilizada. A mesma observação feita antes vale aqui: na maioria dos casos use *is* e *as*;

Free: conhecido por grande parte dos desenvolvedores Delphi. Sua função é liberar a memória alocada pelo objeto, se necessária. A vantagem de usar *Free* no lugar do destrutor da classe é o fato de ser realizada uma verificação de nulidade. Assim ele pode ser chamado mesmo que a instância do objeto seja *nil*, sem que um erro de violação de acesso seja disparado;

Dica: Consulte na edição 71 da revista ClubeDelphi o artigo Criação dinâmica de componentes, para ver em mais detalhes como usar corretamente o *Free*.

InheritsFrom: verifica se o objeto em questão descende ou não de uma determinada classe. Algo similar pode ser feito com o operador *is*, mas apenas em instâncias de objetos.

Note que, em sua maioria, esses métodos são estáticos, como *ClassParent* e *InheritsFrom*, e podem ser chamados independentemente de haver uma instância do objeto. É exatamente isso que não os deixa se tornarem obsoletos.

Como o leitor pode notar, *TObject* oferece métodos pouco úteis, com exceção de *Free*, para o nosso dia a dia. Tais funções se tornam interessantes e necessárias quando es-

tamos desenvolvendo aplicativos que dependam de informações dinâmicas, com RTTI.

TPersistent

Persistência é uma coisa que usamos frequentemente quando trabalhamos com componentes. Mas, o que significa o termo nesse contexto? Veja um exemplo: um botão é colocado em um formulário, e sua propriedade *Caption* é alterada para "Olá mundo". Em seguida o formulário é salvo e fechado. Ao carregá-lo novamente, qual será o valor da propriedade que modificamos anteriormente? A resposta é óbvia: "Olá mundo".

Você, leitor, deve estar pensando: "Mas o que isso tem a ver com *TPersistent*?". Tudo! Por padrão, o Delphi em si, não sabe como as propriedades de componentes devem ser salvas. Para isso, é necessário que cada classe diga ao IDE como o fazer.

Pois bem, a classe que introduz os métodos, que devem ser implementados, para permitir o salvamento e recuperação dessas propriedades é, como agora você já deve ter descoberto, *TPersistent*. De fato, todos os componentes herdam indiretamente dessa classe e, portanto, sabem como persistir seus atributos.

Nota: Essa persistência é feita gravando os dados da memória para o arquivo DFM (ou XFM, no caso de aplicações CLX) e vice-versa.

Nota: Nunca instancie diretamente *TPersistent*. Sempre crie classes que herdem dela para poder criar objetos.

Existem classes que também podem herdar de *TPersistent*, não sendo necessariamente componentes, mas que precisam salvar suas propriedades no arquivo do formulário. Tais objetos, em função disso, podem aparecer em *design time*, não de forma independente, mas sim implementando propriedades de outros componentes. É o caso das classes *TBrush*, *TBitmap*, *TFont*, *TStringList*, *TCollection* etc. Nessa situação são comumente conhecidas como subpropriedades de um componente.

Além de fornecer o mecanismo de persistência, *TPersistent* também dá suporte à atribuição de objetos para outros. Para entender melhor, veja o código da **Listagem 2**.

Listagem 2. Atribuição de ponteiros ou propriedades?

```
var
  { TMeuObjeto herda de TPersistent }
  Objeto1, Objeto2: TMeuObjeto;
begin
  Objeto1 := TMeuObjeto.Create;
  Objeto2 := TMeuObjeto.Create;

  Objeto1.Propriedade1 := 'Teste';
  Objeto1.Propriedade2 := 10;

  Objeto2 := Objeto1;
end;
```

Como o leitor já sabe, o código da listagem anterior não copiará os valores das propriedades de *Objeto1* em *Objeto2*, mas sim faz ambos os objetos apontarem para a mesma instância em memória. A cópia desejada de valores pode ser obtida através do método *Assign*, disponibilizado virtualmente por *TPersistent*, desde que devidamente implementado pelas classes descendentes. Então, o código anterior poderia ser reescrito para:

```
Objeto2.Assign(Objeto1);
```

Como podemos perceber até aqui, *TPersistent* é uma das mais importantes classes da VCL. Mas ela não trabalha sozinha e veremos sua influência no próximo tópico.

TComponent

Essa classe é o coração de todo componente no Delphi. Eles herdam direta (componentes não-visuais) ou indiretamente (componentes visuais) dela. Na hierarquia da VCL, *TComponent* vem logo abaixo de *TPersistent*, e logo implementa métodos para a persistência de suas propriedades publicadas. O mecanismo de persistência é especial para essa classe: as propriedades declaradas na seção *published* são automaticamente gravadas e restauradas com o arquivo do formulário, como pudemos ver no exemplo no início do tópico anterior.

Além dessas características, *TComponent* também introduz o conceito de **proprietário** (*owner*) entre componentes. Um componente pode ser o dono de outros, e ao mesmo tempo ser de propriedade de mais outros. Essa relação é estabelecida através do parâmetro *AOwner* do construtor da classe:

```
constructor TComponent.Create(AOwner: TComponent);
```

O proprietário é responsável por duas coisas: carregar e salvar as propriedades publicadas dos seus "filhos"; e liberá-los da memória quando for destruído.

Nota: Se um componente tem como *Owner* outro que não seja um formulário ou um *DataModule*, suas propriedades não serão salvas e carregadas junto com ele. Nesse caso é necessário configurar o componente como um sub-componente. Isso é feito através do método *SetSubComponent*, visto adiante.

Existem duas propriedades oferecidas por *TComponent* para trabalhar com essa relação entre componentes:

- **Owner:** indica qual componente é o proprietário;
- **Components:** fornece a coleção de componentes "pos-suídos" por esse.

Essas propriedades são muito úteis quando estamos de-



envolvendo aplicações onde os componentes são criados em *runtime*.

A classe *TComponent* também oferece alguns métodos que podem ser bem úteis. Vejamos alguns deles:

FindComponent: retorna a instância de um objeto através do seu nome. Essencial em aplicações que utilizem criação dinâmica de componentes;

InsertComponent: troca o *owner* de um componente pelo chamador do método. Comumente é usado em conjunto com o método *RemoveComponent*. Veja na **Listagem 3** um exemplo de ambos.

Listagem 3. Exemplo de InsertComponent e RemoveComponent

```
var
  I: Integer;
  Temp: TComponent;
begin
  for I := ComponentCount - 1 downto 0 do
  begin
    Temp := Components[I];
    if not (Temp is TControl) then
    begin
      RemoveComponent(Temp);
      DataModule.InsertComponent(Temp);
    end;
  end;
end;
```

Nota: Esses métodos não modificam os arquivos dos formulários (ou *DataModules*) envolvidos. Ou seja, as mudanças não são persistidas.

IsImplementorOf: indica se o componente implementa a interface informada. É similar, até certo ponto, à função *Supports*, e em algumas situações pode tornar a leitura do código mais legível;

Loaded: inicializa o componente após sua leitura do arquivo do formulário. Você não deve chamar esse método manualmente, pois isso é feito pelo Delphi. Porém, pode sobrescrevê-lo para inserir código que deve ser executado imediatamente após o carregamento do componente para a memória. Muito útil quando em desenvolvimento de componentes;

PaletteCreated: chamado quando o componente é criado através da paleta de componentes do Delphi. Você, portanto, não deve chamá-lo explicitamente, mas sim sobrescrevê-lo, como com *Loaded*, para ser notificado da criação do componente pela *Component Palette* do IDE;

RemoveComponent: exclui o componente informado da lista de componentes do chamador. Como já vimos, pode ser usado em conjunto com *InsertComponent* (**Listagem 3**).

Nota: Tenha cuidado ao usar esse método. Antes de excluir um componente do seu *owner*, guarde uma referência para ele, para ser possível o seu posterior acesso.

SetSubComponent: define o componente com um sub-componente, que é um componente cujo *owner* não é nem um formulário nem um *DataModule*, permitindo que suas propriedades publicadas possam ser salvas automaticamente no

arquivo do formulário ou *DataModule* em questão.

Por fim, não podemos esquecer que é em *TComponent* que a propriedade *Tag* aparece inicialmente. Essa propriedade como muitos sabem, não tem significado específico, podendo ser utilizada livremente pelo desenvolvedor para armazenar valores juntamente com o componente. Porém, a maioria a utiliza apenas para guardar valores inteiros, mas é possível guardar qualquer estrutura dentro dela, como mostrado na **Listagem 4**.

Listagem 4. Utilizando Tag para armazenar um record

```
type
  TCliente = record
    Nome: string;
    Idade: Integer;
    Adimplente: Boolean;
  end;

var
  Cliente: TCliente;
  ...
procedure SetTag;
begin
  Cliente.Nome := 'Michael Benford';
  Cliente.Idade := 21;
  Cliente.Adimplente := True;

  Button.Tag := Integer(@Cliente);
end;

procedure GetTag;
var
  PCliente: ^TCliente;
begin
  PCliente := Pointer(Button.Tag);
  with PCliente^ do
  ShowMessage(Nome + sLineBreak + IntToStr(Idade) +
    sLineBreak + BoolToStr(Adimplente));
end;
```

Como todo ponteiro pode ser representado por um inteiro, é fácil guardar qualquer tipo de dado em uma *Tag*. Apenas tome o cuidado de não liberar a memória representada pelo ponteiro armazenado sem atualizar a propriedade, pois caso contrário ocorrerá um erro de violação de acesso no momento em que se tentar acessar a área da memória que não existe mais.

TControl

Na VCL, os componentes são divididos em dois grandes grupos: visuais e não-visuais. Entende-se por componente visual aquele que será visível em *runtime* e irá interagir com o usuário de alguma forma. No jargão da VCL, chama-se de **controle** o componente que pertencer a esse conjunto, e apenas de **componente** o que for englobado pelo outro grupo.

Nota: De agora em diante vamos usar esses termos para referenciar corretamente os componentes. Como vimos no tópico anterior, os componentes são derivados de *TComponent*. Já os controles, como você já deve supor, derivam de *TControl*.

Nessa classe são primeiramente introduzidos métodos, propriedades e eventos que gerenciam a aparência e o comportamento do controle. É o caso das propriedades *Caption*, *Left*, *Top*, *Height* e *Width*, dos métodos *Click*, *Show*

e *Hide*; e dos eventos *OnClick*, *OnMouseMove* e *OnResize*, e muitos outros. Embora herdados por todos os controles visuais, tais atributos são publicados de acordo com a função de cada classe. Por exemplo, o *Image* não publica as propriedades *Caption* e *Font*, nem o evento *OnResize*.

Outro aspecto importante que *TControl* introduz é o conceito de container visual, obtido através da propriedade *Parent*. Esse atributo indica qual controle será o “pai” desse, isso é, qual controle que irá contê-lo visualmente.

Algo essencial que deve ser falado sobre controles, derivados diretamente de *TControl*, é o fato de não receberem a entrada do teclado. Dessa forma, não há o conceito de foco, e a única interação possível com o usuário é através do mouse.

A maioria dos métodos, eventos e propriedades dessa classe já são largamente conhecidos pelos desenvolvedores Delphi. Porém, vale a pena destacar alguns não tão “populares”:

ControlState: informa o estado atual do controle. Seu uso é maior entre criadores de componentes, mas há alguns estados oferecidos que podem ser utilizados também em aplicativos. Como exemplo, inicie uma nova aplicação VCL e coloque um *Button* sobre o formulário. Em seguida implemente o seu evento *OnMouseMove* como mostrado na **Listagem 5**, que permite arrastar um controle com auxílio de *ControlState*.

Listagem 5. Arrastando o controle com o auxílio de *ControlState*

```
procedure TForm1.Button1MouseMove(Sender: TObject;
  Shift: TShiftState; X, Y: Integer);
begin
  with Sender as TControl do
    if csLButtonDown in ControlState then
      begin
        Left := ClientToParent(
          Point(X, Y)).X - (Width div 2);
        Top := ClientToParent(
          Point(X, Y)).Y - (Height div 2);
      end;
    end;
end;
```

Nota: Foi feito o *typecast* de *Sender* para *TControl* apenas para frisar que as propriedades utilizadas no código aparecem inicialmente neste classe.

ControlState é um *set* e possui um elemento chamado *csLButtonDown*, que informa se o botão esquerdo do mouse está pressionado sobre o controle. Se estiver, então movemos o botão pelo formulário com o ponteiro do mouse. Alguns, em situações semelhante, podem ter usado um *flag* auxiliar para saber se o botão do mouse estava pressionado. Observe como a implementação fica bem mais simples como mostrado.

ControlStyle: são definidas várias características do estilo do controle. É possível, por exemplo, impedir que o controle receba os cliques do mouse, ou fixar sua altura e largura, ou ainda saber se existe uma ação associada a ele. Não deve ser modificada em *runtime*, exceto dentro de um construtor de classe. Ou seja, seu uso é restrito ao código de criação de um novo controle;

ClientHeight e *ClientWidth*: respectivamente, a altura e a largura útil do controle. Como implementado em *TControl*, possuem os mesmos valores de *Height* e *Width*, nessa ordem. Em outras classes, o seu significado pode mudar, como em *TForm*, onde *ClientHeight* é a medida da altura do formulário menos a altura da barra de título da janela;

BringToFront e *SendToBack*: enviam o controle para frente e para trás, respectivamente, dos demais controles dentro do mesmo container, mudando a chamada **ordem Z** (*z order*). É importante notar que controles derivados de *TGraphicControl* e *TWinControl*, vistos mais adiante, sempre ficarão dispostos à frente de controles descendentes de *TControl*, independentemente de se chamar o método *BringToFront*;

ClientToParent: translada as coordenadas do ponto informado para as relativas ao controle-pai. No exemplo da **Listagem 5** ele é usado para mover corretamente o botão pelo formulário, já que os valores dos parâmetros *X* e *Y* são relativos ao controle em si, mas as propriedades *Top* e *Left* são representadas no plano do formulário-pai;

ClientToScreen: Como no método anterior, translada o ponto passado para o sistema de coordenadas da tela;

ParentToClient e *ScreenToClient*: fazem o oposto dos dois últimos métodos apresentados, respectivamente;

Invalidate: força o redesenho total do controle. Se mais de uma região dentro do controle precisar ser redesenhada, forçará toda a janela a se desenhar novamente, evitando que o efeito de *flickering* ocorra;

Update: processa quaisquer mensagens de desenho do controle, forçando seu redesenho imediatamente. Deve ser usado quando queremos atualizar algum controle visual, durante um processamento demorado, por exemplo;

Repaint: chama *Invalidate*, seguido de *Update* para redesenhar completamente o controle de forma imediata;

Refresh: como implementado em *TControl*, simplesmente chama *Repaint*;

Perform: serve para simular o envio de uma mensagem do Windows para o controle, geralmente para se obter uma funcionalidade não publicada.

TWinControl

Essa classe engloba todos os controles que são *wrappers* (camadas entre o código nativo e as aplicações que precisam utilizá-lo) dos objetos do Windows, como *Edits*, *CheckBoxes*, *RadioButtons*, *Memos*, *Menus* etc.

Os controles descendentes de *TWinControl* podem processar as entradas do teclado e, logo, receber foco. Além disso, podem conter outros controles, fechando o conceito de container visual, iniciado por *TControl*. Assim, controles derivados daquela classe podem estar contidos em controles derivados de *TWinControl*, mas não vice-versa.



Diferentemente da classe apresentada no tópico anterior, onde as rotinas de desenho dos controles precisa ser implementada, os controles descendentes de *TWinControl* já fazem isso de forma automática, pois é o próprio Windows que cuida de desenhá-los. Por serem *wrappers* de objetos, a camada inferior do código quase nunca é vista, e essa é uma das razões que tornou a VCL tão popular.

Toda a comunicação com a API do Windows, tanto para renderizar quanto para processar as mensagens dos controles é feita de forma transparente pela VCL. Se for necessário “pular” o *wrapper*, *TWinControl* oferece a propriedade *Handle*, que é um atalho para o objeto nativo do sistema operacional.

Isso garante que funcionalidades não implementadas pela VCL possam ser realizadas, como acessar propriedades e métodos não publicados dos objetos do Windows. Dentre os atributos de *TWinControl*, podemos destacar:

Focused: verifica se o controle tem ou não o foco da aplicação;

GetTabOrderList: retorna a lista dos controles filhos desse, ordenados por seus índices de tabulações;

SelectFirst: foca o primeiro controle filho definido pela ordem de tabulação;

SelectNext: foca o próximo controle filho. É possível dizer se a busca será para frente ou para trás, e se será respeitado o valor da propriedade *TabStop*. Como exemplo, inicie uma nova aplicação e coloque diversos *Edits* no formulário. Implemente então o evento *OnKeyPress* de algum deles para o código da **Listagem 6**.

Listagem 6. Movendo o foco com Enter

```
procedure TForm1.Edit1KeyPress(Sender: TObject;
var Key: Char);
begin
if Key = Chr(VK_RETURN) then
begin
SelectNext(Sender as TWinControl, True, True);
Key := #0;
end;
end;
```

Agora atribua aos demais *Edits* o mesmo manipulador de evento. Rode a aplicação e veja que o foco das caixas de texto será movido de acordo com o pressionamento da tecla ENTER;

SetBounds: permite informar a posição e o tamanho do controle de uma única vez, isso é, *Left*, *Top*, *Width* e *Height*, respectivamente.

TGraphicControl

Essa classe serve de base para controles conhecidos como *lightweight* (leves). Uma vez que eles não englobam objetos visuais do Windows, eles são mais rápidos e consomem menos recursos. *TGraphicControl* não aceita entradas do teclado e não pode conter outros controles.

Em contrapartida, ela fornece a propriedade *Canvas*, que dá acesso à “superfície” do controle, permitindo desenhar diretamente nela. Esse desenho é realizado pelo método *Paint*, que deve ser implementado nas classes descendentes

da forma desejada. Os controles *Image* e *PaintBox* são alguns exemplos que herdam de *TGraphicControl*.

TCustomControl

E se eu quisesse herdar de *TWinControl*, para me beneficiar de suas funcionalidades, e precisasse implementar meu próprio algoritmo de desenho? A resposta a essa pergunta é o título do tópico. *TCustomControl* fornece todas as características de *TWinControl*, e também a propriedade *Canvas*, que, como em *TGraphicControl*, provê acesso à área de desenho do controle. Além da propriedade, também é oferecido o método *Paint*, que tem a mesma função da classe citada no tópico anterior.

Conclusão

Vimos neste artigo as principais classes que compõem a estrutura da VCL. Todos os demais componentes são baseados nelas, direta ou indiretamente, cada um implementando determinadas características, estendendo outras etc. Dominar o funcionamento dessas classes pode ajudar a resolver problemas triviais, como qual método chamar para realizar uma determinada tarefa ou evitar escrever uma propriedade que já existe em uma classe ascendente, por exemplo.

Como leitura complementar, eu sugiro uma longa olhada no Help do Delphi. Pesquise por cada classe apresentada aqui e estude seus métodos, propriedades e eventos. Se você tiver inclinação, abra o código-fonte da VCL, disponível com o Delphi, e tente entender como ele funciona.

Com certeza isso aumentará consideravelmente seu conhecimento sobre a linguagem. Um forte abraço a todos e até a próxima! ■

CLUBEDELPHI PLUS!

Acesse agora o mesmo o portal do assinante ClubeDelphi (www.clubedelphi.net/portal) e assista a uma vídeo-aula de Luciano Pimenta que mostra como pintar ícones em componentes *ListBox*, *Statusbar*, *ComboBox*, usando *Canvas*, no endereço <http://www.devmedia.com.br/articles/viewcomp.asp?comp=2083>.

Em uma vídeo aula de Guinther Pauli, no endereço <http://www.devmedia.com.br/articles/viewcomp.asp?comp=570>, veja como criar um controle visual personalizado, descendente de *TGraphicControl*.

CLUBEDELPHI PLUS!

Acesse agora o mesmo o portal do assinante ClubeDelphi (www.clubedelphi.net/portal) e assista a uma vídeo-aula de Guinther Pauli, que mostra como criar um componente que automaticamente atualiza um *ClientDataSet* a cada *n* segundos, usando um timer interno, publicado através do uso de *SetSubComponent*: <http://www.devmedia.com.br/articles/viewcomp.asp?comp=986>