Criação de Componentes

Veja neste artigo de Michael Benford, como criar componentes para o Delphi.

Componentes. Ah, o que seria de nós programadores sem eles. Poucos são os aplicativos que não fazem uso deles. Às vezes ele é um botão, uma caixa de opções. Outras, um engine de banco de dados, um timer. Enfim, há inúmeras funções para um componente. Neste artigo mostrarei como criá-los para tornar mais prático o processo de desenvolvimento de suas aplicações.

Por que criar um componente?

Sempre que você estiver necessitando reescrever algum código relativamente complexo que já tenha feito antes, ou no mesmo projeto ou em outro, é uma boa idéia parar e pensar se não seria mais prátic encapsular este código dentro de um componente.

Eu disse "relativamente complexo" porque se o código em questão é simplesmente uma função que retorna a imagem de um código de barras, por exemplo, então apenas ponha esta função em uma unit de propósito geral e faça referência a ela onde for necessário. Agora, se você quer que o usuário possa visualizar este código de barras em um TImage próprio, gerenciado pelo próprio aplicativo e independente da intervenção dele, o usuário, então seria mais lógico criar um componente.

Transformado em componente, seu código pode ser inserido em quantas aplicações você quiser. Da mesma forma, todas as alterações que você fizer no componente refletirão automaticamente em todos os projetos que o utilizarem.

Pré-requisitos

Para desenvolver componentes no Delphi, você deve conhecer os conceitos da Programação Orientada a Objetos, pois, como dito no tópico anterior, um componente é essencialmente um objeto. Você pode encontrar boas informações sobre POO (em inglês) no site delphi.about.com.

Como os componentes são criados

A primeira coisa a decidir sobre qual classe base você derivará o seu componente. Ao derivar de outra classe, você herda as funcionalidades (métodos, propriedades e eventos) da classe pai. Você deve, portanto, escolher qual classe base se adapta melhor ao tipo de componente que você deseja criar. Veja abaixo como escolher a classe base para seus componentes:

Quando você quer simplesmente adicionar novas funcionalidades a um componente existente, derive-o diretamente da classe desejada. Por exemplo, TMemo.

Às vezes é necessário não apenas adicionar funcionalidades, mas também remover algumas existentes. Assim sendo, derive de TCustomXXXX. Por exemplo, TCustomMemo. Essas classes, em geral, declaram seus métodos, propriedades e eventos na seção Protected. Assim você pode definir quais deles serão visíveis ao usuário da classe derivada.

Se o seu componente deve receber foco em uma janela, ele precisa então de um handle desta mesma janela. TWinControl é a classe onde este handle foi primeiramente introduzido. Se ele não precisar receber foco, derive-o de TGraphicControl.

Caso o seu componente não faça nada visualmente, mas funcione internamente no programa, você deve derivá-lo de TComponent. Todos os componentes derivados desta classe base são visíveis apenas em design time.

Criando nosso primeiro componente

O primeiro componente que iremos criar se chamará TChronometer (cronômetro, em inglês). Ele irá fazer – como o próprio nome diz – a função de um cronômetro. Ao longo do artigo, iremos aperfeiçoá-l com as técnicas que iremos aprender.

Nota: *Eu particularmente nomeio meus componentes utilizando nomes em inglês, assim como seus métodos, propriedades e eventos. Procedo assim porque o componente irá interagir com a IDE do Delphi, que está em inglês, e acho deselegante ver as propriedades misturando os dois idiomas. Mas nada impede, porém, que você desenvolva no idioma que desejar.*

Como o Delphi aberto, vá ao menu Components e selecione New component. Será exibida uma caixa de diálogo onde você deve informar, nesta ordem:

- A classe que o seu componente derivará;
- O nome da classe do novo componente;
- A paleta na guia de componentes do Delphi onde ele será exibido;
- Nome do arquivo da unit do componente.

Preencha o campo Ancestor Type com TComponent e em Class Name digite TChronometer. Em seguida clique em Install, para criar um pacote para nosso componente. Na próxima janela que aparecer, clique na segunda aba (Into new package), escolha um nome para o pacote, a pasta onde ele será salvo, informe uma descrição para ele e em seguida clique em OK. O Delphi automaticamente irá gerar o pacote e o código para o nosso novo componente, inclusive instalando-o na IDE. A partir daí, basta recompilar o pacote para que as alterações efetuadas sobre o componente sejam efetivadas.

Abra o pacote e dê uma olhada na unit criada para o componente e repare as seções Private, Protected, Public e Published:

```
type
   TChronometer = class(TComponent)
   private
     { Private declarations }
   protected
     { Protected declarations }
   public
     { Public declarations }
   published
     { Published declarations }
   end;
```

Estas quatro palavras-chave definem o nível de acessibilidade que os métodos, propriedades e eventos terão em nosso componente. Veja a tabela abaixo:

Private	Métodos, propriedades e eventos declarados nesta seção serão somente acessíveis para outros componentes declarados dentro da mesma unit.
Protected	Métodos, propriedades e eventos declarados aqui serão visíveis para todas as classes descendentes.
Public	Métodos, propriedades e eventos declarados aqui dentro são acessíveis de qualquer lugar.
Published	Esta seção permite que você declare propriedades e eventos que serão exibidas no Object Inspector do IDE do Delphi.

Continuando o desenvolvimento de nosso primeiro componente, digite na seção Private:

```
FStartTime, FElapsedTime: TTime;
```

Estas variáveis irão armazenar o tempo inicial e decorrido, respectivamente. Na seção Public, digite:

```
procedure Start; virtual;
 procedure Stop; virtual;
  property ElapsedTime: TTime read FElapsedTime;
```

Declaramos os métodos do componente responsáveis pelo início e fim da contagem do tempo, e a propriedade ElapsedTime, que dará acesso ao tempo decorrido. Observe que se trata de uma propriedade somente-leitura, pois não usamos a palavra-chave write, apenas read.

Vamos agora implementar o código do nosso componente. Com o cursor dentro da declaração do componente, pressione Ctrl + Shift + C (ou clique com o botão direito do mouse e selecione Complete class at cursor), para que o Delphi gere automaticamente as declarações necessárias para nós. No método Start, informe o seguinte código:

```
procedure TChronometer.Start;
 begin
   FStartTime := Time;
   FElapsedTime := 0;
 end;
```

E, no método Stop:

```
procedure TChronometer.Stop;
 begin
   FElapsedTime := Time – FStartTime;
 end;
```

Pronto! Nosso primeiro componente já faz a função a qual se propõem. Vamos agora testá-lo. Para isso, compile novamente o pacote. Depois da compilação, inicie uma nova aplicação e insira o nosso componente no formulário. Coloque também dois TButton e um TLabel. Altere o Caption do primeiro botão para "Iniciar" e do segundo para "Terminar". Altere ainda a propriedade Name do TLabel para lblTempo. Agora, dê um duplo clique sobre o botão Iniciar e digite:

```
lblTempo.Caption := '00:00:00';
 Chronometer1.Start;
```

Repita o processo para o segundo botão, adicionando o seguinte código:

```
Chronometer1.Stop;
 lblTempo.Caption := TimeToStr(Chronometer1.ElapsedTime);
```

Execute a aplicação e veja nosso componente em ação. Clique em Iniciar, espere alguns segundos e depois clique em Parar. Você verá o tempo decorrido no label que adicionamos ao formulário.

Reaproveitamento de código

O leitor deve ter reparado na palavra-chave virtual, colocada após a declaração dos métodos do componente. Ela, juntamente com dynamic, abstract e override, servem para se reaproveitar o código do componente em classes que venham a derivar dele. Isto será mais bem entendido com o próximo exemplo. Crie um novo componente, derivando-o da nossa classe TChronometer. Chame-o de TChronometer2. O código gerado pelo Delphi deve se parecer como seguinte:

```
type
   TChronometer2 = class(TChronometer)
   private
     { Private declarations }
   protected
     { Protected declarations }
   public
     { Public declarations }
   published
     { Published declarations }
   end;
```

Agora, na seção Public, declare novamente os métodos da classe ancestral Start e Stop, mas, no lugar da palavra-chave virtual, coloque override. O código então ficaria assim:

```
procedure Start; override;
 procedure Stop; override;
```

Faça o Delphi implementar o código pressionando Ctrl + Shift + C e nas declarações geradas, digite:

```
procedure TChronometer2.Start;
 begin
   ShowMessage('O cronômetro será iniciado');
   inherited Start;
 end;

 procedure TChronometer2.Stop;
 begin
   ShowMessage('O cronômetro será finalizado');
   inherited Stop;
 end;
```

Instale este novo componente, inicie uma nova aplicação e proceda como no exemplo anterior. Execute-a e clique no botão Iniciar. Você verá nosso método modificado pelo novo componente, mas mantendo ainda sua função original. Portanto, a palavra-chave virtual define que um método poderá ser modificado em classes descendentes. E para sobrescrever, na classe derivada, algum método, usamos a palavra-chave override. Ao sobrescrever algum método virtual, ainda podemos executar o código original do mesmo método, bastando para isso fazer uso da palavra-chave inherited, seguida do nome do método.

Nota: *Se você não especificar a palavra-chave override, ou então criar um método como o mesmo nome de outro na classe ancestral que não seja virtual, o código original do componente continuará sendo executado, ao invés do novo código informado na classe derivada.*

Ainda falta explicar o uso de dynamic e abstract, então vamos lá. Dynamic tem a mesma função de virtual, a diferença é entre os dois é uma questão de tamanho versus velocidade. Métodos dinâmicos irão gerar instâncias do componente que exigem menos memória, enquanto métodos virtuais irão gera códigos mais rápidos, exigindo um pouco mais de memória do computador.

E abstract define um método que não foi implementado ainda, ou seja, não há código para ele. Sua funcionalidade deve ser implementada em classes derivadas.

Criando eventos

Quase todos os componentes possuem eventos. Quase, mas não todos. Um evento serve para notificar que alguma coisa foi executada pelo componente. O programador então pode escrever código que reaja a esta notificação.

Eventos são simplesmente procedimentos ou funções (raramente), que pertencem a uma classe. Para entender, volte ao código de nosso cronômetro e digite na seção type:

```
type
   TStateChronometer = (seStopped, seRunning);
   TChangeStateEvent = procedure (Sender: TObject; State: TStateChronometer) of object;
```

Aqui, criamos um tipo enumerado que irá identificar o estado de nosso cronômetro. Em seguida, definimos um evento que irá receber dois parâmetros: Sender e State. É sempre bom passarmos para nossos eventos o objeto (componente) que chamou o evento. Por isso usamos Sender: TObject. (Confesse, você já viu isso antes não...). Agora, modifique as seções Private e Published como mostrado a seguir:

```
private
   FOnStart, FOnStop: TNotifyEvent;
   FOnStateChange: TChangeStateEvent;
 (...)
 published
   property OnStart: TNotifyEvent read FOnStart write FOnStart;
   property OnStop: TNotifyEvent read FOnStop write FOnStop;
   property OnStateChange: TChangeStateEvent read FOnStateChange write FOnStateChange;
```

O leitor deve ter notado que criamos três eventos: um utilizando a definição que criamos previamente e dois fazendo uso do tipo pré-definido do Delphi TNotifyEvent. Esse tipo, declarado na unit Classes, é muito útil quando precisamos criar um evento que não precisa passar nenhum parâmetro especial ao usuário, pois ele repassa ao usuário (programador) apenas o objeto que executou o evento. De fato, se você der uma olhada na declaração dele na unit Classes, verá o código a seguir:

```
type
 ...
   TNotifyEvent = procedure (Sender: TObject) of object;
```

Muito bem. Falta agora definir onde e quando os eventos serão disparados por nosso componente, sendo esta uma tarefa muito fácil neste exemplo. OnStart e OnStop devem ser disparados quando o cronômetro iniciar e parar, respectivamente. E OnStateChange? Nos dois eventos, onde a diferença será que o "estado" da execução será diferente para cada um dos dois. Resta-nos saber como chamar o evento, o que é também muito fácil. As variáveis FOnStart, FOnStop e FOnStateChange – campos no caso de componentes – armazenam o endereço da função na memória. Observe: apenas o endereço em memória, não o código da função. Assim, para executar a função armazenada no endereço salvo, basta chamar a própria variável como se ela fosse uma função, passando da mesma forma os parâmetros necessários. Veja a implementação no nosso componente para entender melhor:

```
procedure TChronometer.Start;
 begin
   if Assigned(FOnStart) then
     FOnStart(Self);
  (...)
 end;
```

Devemos testar se a variável de função não é nula (nil), através da função Assigned, que retorna true em caso positivo. Se não fizermos este teste, e chamarmos a função, será gerado um erro de violação de acesso caso a variável seja nil.

Nota: *Quando estamos trabalhamos com objetos, as variáveis que o pertencem passam a se chamar campos, e as funções/procedures, métodos. Lembre-se que componentes também são objetos! Daqui para frente iremos utilizar esta nomenclatura.*

Agora que já sabemos como invocar nossos eventos, basta repetir o mesmo processo para o método Stop. Neste exemplo não vamos nos preocupar com a "localização" do gatilho do evento, isto porque, se nosso evento existisse para ser executado explicitamente depois de o cronômetro iniciar, deveríamos colocá-lo no final do método, e, por elegância, modificar o nome do evento para algo como OnAfterStart. Assim, o programador que utilizasse o componente saberia com precisão o momento em que este evento seria executado.

Resta apenas o evento OnStateChange. O processo para ele é rigorosamente igual aos outros dois eventos, sendo a diferença localizada na passagem dos parâmetros:

```
procedure TChronometer.Start;
 begin
   { Evento OnStart }
   (...)
   { Evento OnStateChange }
   if Assigned(FOnStateChange) then
     FOnStateChange(Self, seRunning);
   (...)
 end;
```

O leitor deve estar pensando: "Bom, agora é só repetir o mesmo código para o método Stop", mas eu digo que esta não é a forma mais correta proceder, e vou explicar o porquê. Sempre que você se deparar com a necessidade de repetir algum código que já escreveu antes, eu sugiro criar uma procedure/função que execute a processo desejado. Assim, você encapsula o código em um único lugar, e todas as alterações refletirão automaticamente. Seguindo este pensamento, volte à declaração da classe TChronometer e adicione a seguinte procedure na seção Protected:

```
TChronometer = class(TComponent)
   private
   { Código da seção private }
   protected
     procedure DoChangeState(State: TStateChronometer); virtual;
   { Resto do código da classe }
 end;
```


Implemente este método como abaixo:

```
procedure TChronometer.DoChangeState(State: TStateChronometer);
 begin
   FOnStateChange(Self, State);
 end;
```

Agora, no método Start, modifique o código adicionado anteriormente por:

```
procedure TChronometer.Start;
 begin
   { Evento OnStart }
   (...)
   DoChangeState(seRunning);
   { Restante do código do método Start }
   (...)
 end;
```

E, finalmente, no método Stop:

```
procedure TChronometer.Stop;
 begin
   { Evento OnStop }
   (...)
   DoChangeState(seStopped);
   { Restante do código do método Stop }
```

```
  (...)
end;
```

Outros componentes como propriedades

Muitas vezes temos a necessidades de utilizar outro componente existente no formulário em nosso próprio componente. Um exemplo disto é o componente TLabel. Nele existe uma propriedade chamada FocusControl, que recebe qualquer outro componente derivado de TWinControl. Vamos ver neste tópico como interagir nosso componente com outros, que já existam no formulário onde ele for inserido.

Nota: *Não obrigatório que estes componentes com os quais queiramos interagir existam em design-time. Eles também podem ser criados e associados ao nosso componente em run-time.*

Modifique a seção Private para o seguinte:

```
private
  (...)
    FLabel: TLabel;
    FTimer: TTimer;
    FRefreshInterval: integer;
    procedure TimerEvent(Sender : TObject);
  (...)
 public
    constructor Create(AOwner: TComponent); override;
    destructor Destroy; override;
 published
    property Display: TLabel read FLabel write FLabel;
    property RefreshInterval: integer read FRefreshInterval write SetRefreshInterval default 1000;
  (...)
 end;
```


Eis uma breve explicação do que vamos fazer: ao invés de calcular o tempo decorrido apenas quando o método Stop for chamado, iremos utilizar um Timer interno para que a atualização seja automática. Também vamos permitir ao usuário especificar o intervalo de tempo desta atualização. Isto será controlado através da propriedade RefreshInterval. Note que utilizamos um termo novo: a palavra-chave default. Sempre que tivermos uma propriedade do tipo ordinal, e quisermos definir valores padrões para elas, devemos utilizar esta palavra-chave seguida do valor desejado, e, no construtor da classe, atribuir este valor explicitamente. Acompanhe o artigo para entender melhor.

Seguindo em frente com o desenvolvimento, implemente os novos métodos como a seguir, pressionando Ctrl+Shift+C. Note que o Delphi irá gerar um método adicional, que não definimos inicialmente.

```
constructor TChronometer.Create(AOwner : TComponent);
 begin
   inherited Create(AOwner);

   FTimer := TTimer.Create(Self);
   FRefreshInterval := 1000; // Necessário para que default funcione   corretamente
   FTimer.OnTimer := TimerEvent;
   FTimer.Interval := FRefreshInterval;
   FTimer.Enabled := false;
 end;

 destructor TChronometer.Destroy;
 begin
```

```
  FTimer.Free;
  inherited Destroy;
end;

procedure TChronometer.TimerEvent(Sender : TObject);
begin
  FElapsedTime := Time - FStartTime;
  if Assigned(FLabel) then
    FLabel.Caption := TimeToStr(FElapsedTime);
end;

{ Método gerado automaticamente pelo Delphi }
procedure TChronometer.SetRefreshInterval(const Value: integer);
begin
  { Adicione estas linhas }
  FRefreshInterval := Value;
  FTimer.Interval := FRefreshInterval;
end;
```

Compile e reinstale o pacote do componente. Volte a nossa aplicação de exemplo e veja as novas propriedades no Object Inspector. Atribua a Display o label existente no formulário, e retire do código do botão Parar a linha que atualiza a exibição do tempo decorrido. Agora, teste a aplicação.

Tudo certo, a princípio. Ainda falta um detalhe muito importante. O que aconteceria se excluíssemos o label da nossa aplicação? A referência a ele na propriedade Display exibida no Object Inspector de nosso componente irá desaparecer, mas o campo interno que usamos para guardar o valor desta propriedade FLabel, ainda apontará para o label excluído. Isto é ruim, pois quando executarmos novamente o método Start, a aplicação gerará um erro de violação de acesso, pois o label que deveria exibir o tempo decorrido não existe mais.

A solução para este problema é simples. Sempre que algum componente é excluído do formulário, ele notifica todos os demais sobre isto. Seria como ele dissesse: "Ei, outros componentes! Fui apagado pelo usuário!". O que precisamos fazer é receber esta mensagem, e para isso, o Delphi dispõe da procedure Notification. Este método, derivado de TComponent, é chamado sempre que uma notificação chega ao componente. Só precisamos sobrescrevê-la. Vejamos a seguir como fazer isso. Primeiramente, adicione o novo método na seção Protected:

```
protected
 procedure Notification(AComponent: TComponent; Operation: TOperation); override;
 (...)
```

Repare que esta procedure recebe como parâmetros o componente que gerou a notificação e a operação realizada, isto é, o que aconteceu com ele. Implemente-o agora pressionando Ctrl+Shift+C:

```
procedure TChronometer.Notification(AComponent: TComponent;
   Operation: TOperation);
 begin
   if (AComponent = FLabel) and (Operation = opRemove) then
     FLabel := nil;
   inherited;
 end;
```

Agora sim! Se o label for excluído do form, nosso componente será notificado e irá atribuir o valor nil para o campo responsável pelo armazenamento do "componente auxiliar". Porém, isto nos leva a outro problema: se o label estiver em outro formulário, diferente do que guarda nosso cronômetro, então não seremos notificados da sua exclusão, caso ela aconteça. Novamente há uma solução. TComponent possui um método chamado FreeNotification, que diz ao componente que ele deve lembrar de nós na hora em quer for excluído. Veja como implementar isso. Primeiro, na seção Published, altera a linha

```
property Display: TLabel read FLabel write FLabel;
```

para

```
property Display: TLabel read FLabel write SetDisplay;
```

Agora diga ao Delphi para implementar nosso novo método (Ctrl+Shift+C). Com ele gerado, digite o seguinte código:

```
procedure TChronometer.SetDisplay(const Value: TLabel);
 begin
   if Assigned(FLabel) then
     FLabel.RemoveFreeNotification(Self);
 // Removemos nosso componente da lista de FLabel, caso ele já exista
   FLabel := Value;
   if Assigned(FLabel) then
     FLabel.FreeNotification(Self);
 end;
```

Muito bem. Desta maneira seremos avisados quando o componente deixar de existir. Note que primeiro removemos a "FreeNotification", e depois, se necessário, a atribuímos novamente.

Conjuntos

O leitor já deve ter reparado na propriedade Style, do objeto TFont, no Object Inspector. Ela permite que sejam configurados os estilos de fonte (negrito, itálico, etc). Esta propriedade é um conjunto (set), e é sobre isto que falaremos neste tópico. Para não perdermos tempo, adicione o seguinte código na seção type da unit do nosso componente:

```
type
 (...)
   TViewTypes = (vtHours, vtMinutes, vtSeconds);
   TViewType = set of TViewType;
```

Inclua também o código a seguir na declaração da classe:

```
TChronometer = class (TComponent)
   private
   (...)
   FViewType : TViewType;
   (...)
   published
   (...)
   property ViewTypes : TViewType read FViewType write SetViewType;
```

```
  (...)
end;
```

Implemente o novo método com Ctrl+Shift+C. Agora, vá ao método construtor e adicione as seguintes linhas:

```
Include(FViewType, vtHours);
 Include(FViewType, vtMinutes);
 Include(FViewType, vtSeconds);
```

Estamos atribuindo os valores padrões para nossa propriedade-conjunto, através da função pré-definida Include. Basta informar o set e o elemento a ser adicionado.

Nota: *O Delphi fornece também a função Exclude, que faz o processo inverso de Include.*

Prosseguindo, no método SetViewType, digite o código seguinte:

```
var
   CaptionLabel: string;
 begin
   FViewType := Value;
   if (csDesigning in ComponentState) and Assigned(FLabel) then
     begin
       CaptionLabel := '';
       if vtHours in FViewType then
         CaptionLabel := '00:';
       if vtMinutes in FViewType then
         CaptionLabel := CaptionLabel + '00:';
       if vtSeconds in FViewType then
         CaptionLabel := CaptionLabel + '00';
       if vtMSeconds in FViewType then
         CaptionLabel := CaptionLabel + ':000';
       FLabel.Caption := CaptionLabel;
     end;
 end;
```


Eis o que fizemos: quando a propriedade ViewType for alterada no Object Inspector, iremos atualizar o label para que ele reflita, ainda em design-time, as novas configurações. Temos dois conceitos novos aqui, csDesigning e ComponentState. ComponentState é uma propriedade derivada de TComponent que informa o estado atual do componente. Esta propriedade é também um set, e, portanto pode armazenar múltiplas opções. No nosso exemplo, verificamos se csDesigning está definido, indicando que o componente está em design-time, ou seja, na IDE do Delphi.

Podemos, desta forma, escrever código que não seja executado em run-time, apenas em design-time. Em seguida, verificamos se existe algum label relacionado ao nosso componente e checamos também as opções de nossa propriedade-conjunto, uma a uma.

Para finalizar, no método TimerEvent, modifique o código como a seguir:

```
procedure TChronometer.TimerEvent(Sender: TObject);
 var
   Format: string;
 begin
   FElapsedTime := Time - FStartTime;
   Format := '';
```

```
  if vtHours in FViewType then
    Format := 'hh:';
  if vtMinutes in FViewType then
    Format := Format + 'nn:';
  if vtSeconds in FViewType then
    Format := Format + 'ss';
  if vtMSeconds in FViewType then
    Format := Format + ':zzz';
  if Assigned(FLabel) then
     FLabel.Caption := FormatDateTime(Format, FElapsedTime);
end;
```

Utilizando a mesma checagem usada no método SetViewType, formatamos uma string que será usada na função FormatDateTime. Esta permite que especifiquemos como a saída do tempo será formatada. O padrão, usado na função TimeToStr é hh:nn:ss, que indica horas:minutos:segundos. Com FormatDateTime, podemos omitir os parâmetros que quisermos, ou adicionar outros (como 'zzz', que exibe também os milisegundos do tempo) e modificar esta saída. Procure na ajuda do Delphi por esta função para obter mais detalhes sobre estes formatos.

Para finalizar este tópico, vá ao método SetDisplay e adicione o código a seguir:

```
if Assigned(FLabel) then
 begin
  FLabel.FreeNotification(Self);
  SetViewType(FViewType); // Adicione esta linha
 end;
```


Isto fará com que o Caption do label seja atualizado também quando for "anexado" ao nosso componente.

Agora salve a unit, recompile e reinstale o componente e teste-o novamente em nossa aplicação exemplo.

Sub-propriedades

Durante o processo de desenvolvimento de componentes, nós programadores muitas vezes sentimos a necessidade de agrupar propriedades e, mais raramente, métodos, de forma que estes fiquem mais organizados, sejam no Object Inspector dentro da IDE do Delphi ou mesmo no nível da programação em si. Não apenas por necessidade, mas também pela "etiqueta de criação de componentes", devemos agrupar propriedades que estejam relacionadas, para tornar a manipulação das propriedades mais prática e menos confusa. Um exemplo disto é a propriedade Font, presente em muitos componentes da VCL.

Se pararmos um pouco para pensar, talvez a maioria chegue à conclusão de que o ideal aqui seria criar um outro objeto, contendo as propriedades a serem agrupadas e declarar este objeto como uma propriedade do componente. E esta é mesmo a conclusão correta. O importante é derivar este "objeto-propriedade" da classe correta, que neste caso é a classe TPersistent, e tomar alguns cuidados especiais para este tipo de técnica. Eis o que discutiremos neste tópico: a criação destes "objetos-propriedades", ou simplesmente, sub-propriedades.

Não iremos mais aproveitar o código de nosso cronômetro aqui, por uma simples razão: não nos determos apenas em criar componentes derivados de uma única classe, no caso TComponent. Iremos criar um novo componente para ilustrar nossos exemplos daqui em diante. Então vamos lá! Crie um novo componente, da mesma forma que criamos o primeiro, e derive-o de TStringGrid. Dê à classe o nome de TExtendedStringGrid, salve a unit dentro de um novo pacote e instale-o na IDE. Dentro do package criado, abra a unit do nosso novo componente. A declaração da classe deve estar assim:

```
type
   TExtendedStringGrid = class(TStringGrid)
   private
     { Private declarations }
   protected
     { Protected declarations }
   public
     { Public declarations }
   published
     { Published declarations }
   end;
```

Vamos agora iniciar a criação da nossa sub-propriedade. A sua função será a seguinte: permitir ao usuário alterar a cor e a fonte da célula selecionada na StringGrid. Para isso teremos, portanto, duas propriedades, que chamaremos de BackColor e Font, respectivamente. Acima da linha que declara a classe TExtendedStringGrid, adicione o código a seguir:

```
type
   TSelectedCell = class;
   TExtendedStringGrid = class(TStringGrid)
   (...)
```


Agora, modifique as seções Private e Published para o seguinte:

```
private
   FSelectedCell : TSelectedCell;
   (...)
 published
   property SelectedCell: TSelectedCell read FSelectedCell write SetSelectedCell;
 end;
```

Vamos entender o que fizemos: criamos a classe TSelectedCell "crua" assim para podermos declarar os campos necessários na classe do nosso componente. Usamos este recurso porque iremos declarar TSelectedCell na íntegra após a declaração de TExtendedStringGrid, e o este último precisa saber que o primeiro existe. Poderíamos simplesmente declarar a classe completa antes da declaração de nosso componente, mas aproveitei a "deixa" para explicar como podemos usar objetos em outros declarados antes de eles existirem de fato.

Voltando ao desenvolvimento, após a classe TExtendedStringGrid adicione o código a seguir:

```
TSelectedCell = class(TPersistent)
   private
     FOwner: TPersistent;
     FBackColor: TColor;
     FFont: TFont;
     procedure SetBackColor(const Value: TColor);
   public
```

```
    constructor Create(AOwner: TComponent);
    destructor Destroy; override;
    procedure Assign(Source: TPersistent); override;
    function GetOwner: TPersistent; override;
  published
    property BackColor: TColor read FBackColor write SetBackColor default clHighlight;
    property Font : TFont read FFont write FFont;
  end;
```

Eis a declaração de nossa sub-propriedade. Os cuidados especiais mencionados anteriormente são estes: sobrescrever o método Assign e, opcionalmente, o método GetOwner. O leitor vai entender o porquê ao acompanhar a implementação destes métodos. As demais propriedades são auto-explicativas.

Prosseguindo, adicione a unit Graphics na cláusula uses (necessário para TColor e TFont), e pressione Ctrl+Shift+C para implementar todos os métodos necessários. Vá ao construtor da classe e digite:

```
constructor TSelectedCell.Create(AOwner: TComponent);
 begin
   inherited Create;
   FOwner := AOwner;
   FBackColor := clHighlight;
   FFont := TFont.Create;
 end;
```

Armazenamos o proprietário da nossa classe e inicializamos os campos necessários. No destrutor da classe, adicione o código abaixo:

```
destructor TSelectedCell.Destroy;
 begin
   FFont.Free;
   inherited Destroy;
 end;
```


Vamos ver agora os métodos herdados e sobrescritos de TPersistent: Assign e GetOwner. O primeiro, caso o leitor não saiba, serve para atribuirmos todas as propriedades de um objeto em outro, ambos da mesma classe. Como estamos criando novas propriedades, este método não tem como saber da existência destes novos campos. Devemos, portanto nos encarregar de escrever sua função corretamente. Já o segundo (GetOwner), deve retornar o proprietário da classe, que é o componente armazenado no campo FOwner, inicializado no construtor da classe. Confira o código completo a seguir:

```
procedure TSelectedCell.Assign(Source: TPersistent);
 begin
   if Source is TSelectedCell then
     with TSelectedCell(Source) do
     begin
       Self.BackColor := BackColor;
       Self.Font.Assigned(Font);
     end
     else
     inherited; // Gera uma exceção caso Source não seja do tipo TSelectedCell
 end;

 function TSelectedCell.GetOwner: TPersistent;
 begin
```

```
  Result := FOwner;
end;
```

Muito bem. Agora vamos implementar a funcionalidade de nosso componente. Volte à declaração da classe TExtendedStringGrid e sobrescreva os construtores e destrutores da classe, deixando a seção Public assim:

```
public
  constructor Create(AOwner : TComponent); override;
  destructor Destroy; override;
  (...)
```

Na seção Protected, adicione o código a seguir:

```
protected
  procedure DrawCell(ACol, ARow: Longint; ARect: TRect; AState: TGridDrawState); override;
```

Nota 1: *Você deve adicionar a unit Types na cláusula uses para que o tipo TRect seja reconhecido.*

Nota 2: *A função DrawCell será explicada mais adiante.*

Vamos por a mão na massa agora. Implemente tudo (Ctrl+Shift+C) e adicione o código abaixo (segue a listagem completa):

```
constructor TExtendedStringGrid.Create(AOwner: TComponent);
begin
  inherited Create(AOwner);
  FSelectedCell := TSelectedCell.Create(Self);
end;

destructor TExtendedStringGrid.Destroy;
begin
  FSelectedCell.Free;
  inherited;
end;

procedure TExtendedStringGrid.DrawCell(ACol, ARow: Integer; ARect: TRect; AState: TGridDrawState);
begin
  with Canvas do
  begin
    if (ACol = Col) and (ARow = Row) then
    begin
      Brush.Color := FSelectedCell.BackColor;
      Font.Assign(FSelectedCell.Font);
    end;
    FillRect(ARect);
    TextRect(ARect, ARect.Left+2, ARect.Top+2, Cells[ACol, ARow]);
  end;
  if Assigned(OnDrawCell) then
    inherited DrawCell(ACol, ARow, ARect, AState);
end;

procedure TExtendedStringGrid.SetSelectedCell(const Value: TSelectedCell);
begin
  FSelectedCell.Assign(Value);
end;
```

Apenas o método sobrescrito DrawCell merece uma breve explicação: ele é chamado toda vez que cada célula da StringGrid precisa ser redesenhada. Portanto é através dele que vamos alterar o estilo da célula selecionada. Observe como acessamos os campos da sub-propriedade. E no final do método verificamos se o usuário escreveu algum código especial para o desenha da Grid, e em caso positivo, executamos este código.

Por fim, no método SetBackColor de TSelectedCell, adicione a linha comentada abaixo:

```
procedure TSelectedCell.SetBackColor(const Value: TColor);
 begin
   FBackColor := Value;
   TWinControl(GetOwner).Invalidate; // Adicione esta linha, que irá redesenhar o componente no formulário
 end;
```

Isto fará com que as alterações sejam vistas ainda em design-time. Salve tudo, recompile o pacote e faça uma aplicação de testes para visualizar nosso componente em ação. Não vou mencionar como fazer isso aqui, mas a aplicação pronta está disponível para download, no final deste artigo.

Manipulando mensagens do Windows

Esta seção será de fácil entendimento e não tomará muito nosso tempo. Como o leitor deve saber, o Windows se comunica com as aplicações que estão sendo executadas através de mensagens, que carregam uma infinidade de informações sobre muitos procedimentos executados pelo sistema operacional. Às vezes torna-se necessário fazer uso destas mensagens para implementar certas funcionalidades a um programa, ou porque é mais fácil utilizar estas mensagens ou simplesmente porque não há outra maneira de se fazer tais coisas.

Para você interceptar as mensagens que o Windows passa ao seu aplicativo basta saber duas coisas: o nome da mensagem em si e eventualmente os nomes dos parâmetros que ela carrega. Com estes dois dados em mãos, o resto é "mamão com açúcar". ;-). Vamos lá então.

Crie dois novos eventos, do tipo TNotifyEvent, e adicione-os ao nosso componente. Chame-os de OnMouseEnter, OnMouseLeave. Na seção Protected, crie duas procedures (métodos), como mostrado a seguir:

```
procedure CMMouseEnter(var Message: TMessage); message CM_MOUSEENTER;
 procedure CMMouseLeave(var Message: TMessage); message CM_MOUSELEAVE;
```

Implemente estes métodos e adicione o código a seguir:

```
procedure TExtendedStringGrid.CMMouseEnter(var Message: TMessage);
 begin
   { Supondo que você nomeou o campo do evento como FOnMouseEnter }
   if Assigned(FOnMouseEnter) then
     FOnMouseEnter(Self);
 end;


 procedure TExtendedStringGrid.CMMouseLeave(var Message: TMessage);
 begin
   { Supondo que você nomeou o campo do evento como FOnMouseLeave }
```

```
  if Assigned(FOnMouseLeave) then
    FOnMouseLeave(Self);
end;
```

Adicione Messages na cláusula uses. Isto é tudo. Procure no help do Delphi e na unit Messages para verificar as mensagens existentes. Lembre-se que mensagens iniciadas com CM_ indicam uma mensagem sobre um componente e WM_ sobre o Windows.

Nota: *Mensagens não são suportadas pelo Linux.*

Coleções

O último tópico abordado neste artigo, e também sem dúvida o de maior complexidade, trata do uso de coleções. O leitor, ao ler "coleção", talvez pense que não saiba o que este termo representa, mas eu apostaria que você já viu uma coleção antes. Se você já utilizou o componente DBGrid, então conhece a propriedade Columns, que nada mais é do que uma coleção. Como você deve imaginar, este tipo de propriedade é um objeto, derivado diretamente de TPersistent, que acabamos de ver, e sua função é armazenar outros objetos (itens), de quantidade indefinida no momento da criação do componente e cada um com suas próprias características (propriedades). Explicado o novo termo, vamos agora ver como ele funciona.

As coleções são objetos do tipo TCollection e seus itens objetos do tipo TCollectionItem. Você deve criar classes derivadas de ambos para que tudo funcione corretamente. Também é necessário33 que alguns métodos da classe ancestral sejam sobrescritos nas classes derivadas. Vamos utilizar um exemplo para entendimento destes conceitos.

Eis o que faremos: criaremos uma propriedade Columns, semelhante à existente no componente DBGrid, que permitirá o controle de algumas características das colunas do componente. A primeira coisa a fazer é criar nossa classe derivada de TCollectionItem, que armazenará as propriedades relativa a cada item da coleção, ou, em termos práticos, a cada coluna da nossa StringGrid modificada. Antes d iniciarmos a escrita do código, volte à seção Type da unit e adicione as linhas a seguir:

```
type
   TSelectedCell = class;
   TSGColumn = class; // Adicione esta linha
   TSGColumnItem = class; // e esta também
   (...)
```

Como explicado anteriormente, isto nos permitirá desenvolver o código após a declaração do componente. Vejamos então o código da nova classe TSGColumnItem:

```
TSGColumnItem = class(TCollectionItem)
   private
     FColor: TColor;
     FHeader: string;
     FHeaderFont: TFont;
     FFont: TFont;
     FTag: integer;
   protected
     function GetDisplayName: string; override;
   public
     constructor Create(Collection: TCollection); override;
```

```
    destructor Destroy; override;
    procedure Assign(Source: TPersistent); override;
  published
    property Color: TColor read FColor write SetColor default clWhite;
    property Font: TFont read FFont write SetFont;
    property Header: string read FHeader write SetHeader;
    property HeaderFont: TFont read FHeaderFont write SetHeaderFont;
    property Tag: integer read FTag write FTag;
end;
```

Há apenas duas coisas novas aqui: o construtor da classe, que, diferente do usual, recebe a coleção que gerenciará o item; e a função herdada da classe ancestral GetDisplayName. Esta deve retornar uma string que irá representar o editor padrão para coleções na IDE do Delphi (aquela janela pop-up que aparece quando editamos uma propriedade derivada de TCollection). O método sobrescrito Assign apenas deve configurar corretamente todas as propriedades da classe, exatamente como fizemos no tópico Sub-propriedades.

Vejamos então o código para o construtor e para GetDisplayName:

```
constructor TSGColumnItem.Create(Collection: TCollection);
begin
  inherited Create(Collection);
  FFont := TFont.Create;
  FHeaderFont := TFont.Create;
  FColor := clWhite;
end;

function TSGColumnItem.GetDisplayName: string;
begin
  Result := Format('Column %d', [Index]);
end;
```


O destrutor, não mostrado aqui, faz o processo inverso: libera os campos e chama o método ancestral da classe. Repare na propriedade Index, usada na função GetDisplayName. Ela armazena o índice do item na coleção.

Implemente os outros métodos da classe pressionando Ctrl+Shift+C e escreva o código como nos exemplos anteriores. Lembre-se que TFont deriva de TPersistent e que é necessário usar o método Assign para atribuir as propriedades corretamente.

Vejamos agora a classe da coleção que irá gerenciar estes itens, observando o código a seguir:

```
TSGColumn = class(TCollection)
  private
    FOwner: TComponent;
  protected
    function GetOwner: TPersistent; override;
  public
    constructor Create(AOwner: TComponent);
    function Add: TSGColumnItem;
    function Insert(Index: integer): TSGColumnItem;
    property Items[Index: integer]: TSGColumnItem read GetColumnItem write SetColumnItem;
  end;
```

GetOwner é um método virtual que deve retornar o proprietário da classe. Isto é necessário para que o Object Inspector funcione corretamente. Este proprietário ficará armazenado no campo FOwner, inicializado no construtor da classe. Repare que este último não é um método virtual, e, portanto não podemos sobrescrevê-lo usando override. Veja o código de GetOwner e Create abaixo:

```
constructor TSGColumn.Create(AOwner: TComponent);
 begin
   inherited Create(TSGColumnItem);
   FOwner := AOwner;
 end;

 function TSGColumn.GetOwner: TPersistent;
 begin
   Result := FOwner;
 end;
```

Add deve adicionar um novo item, do tipo TSGColumnItem, ao final da lista e Insert na posição desejada, como mostrado a seguir:

```
function TSGColumn.Add: TSGColumnItem;
 begin
   Result := TSGColumnItem(inherited Add);
 end;

 function TSGColumn.Insert(Index: integer): TSGColumnItem;
 begin
   Result := TSGColumnItem(inherited Insert(Index));
 end;
```


Por fim a propriedade Items[Index: integer], declarada desta forma, permite que utilizemos uma estrutura parecida como uma array, onde existirão vários itens ao mesmo tempo. Esta propriedade dev utilizar métodos explícitos para a leitura/escrita, neste caso, GetColumnItem e SetColumnItem (Na verdade esta propriedade já existe na classe ancestral; nós apenas a reescrevemos para nosso próprio tipo de item). Instrua o Delphi a criá-los com Ctrl+Shift+C e digite o código a seguir:

```
function TSGColumn.GetColumnItem(Index: integer): TSGColumnItem;
 begin
   Result := TSGColumnItem(inherited Items[Index]);
 end;

 procedure TSGColumn.SetColumnItem(Index: integer; const Value: TSGColumnItem);
 begin
   Items[Index].Assign(Value);
 end;
```

Muito bem. Agora, na declaração da classe de nosso componente, apenas adicione um campo para TSGColumn, declare uma propriedade apontando para ele e pronto. Lembre-se de usar um método do tipo Set para a gravação da propriedade, e nele use Assign para atribuir o valor ao campo da coleção. Seria mais ou menos isso:

```
procedure TExtendedStringGrid.SetColumns(const Value: TSGColumn);
 begin
   FColumns.Assign(Value);
 end;
```

Salve tudo, recompile o pacote, inicie uma nova aplicação e adicione um TExtendedStringGrid ao formulário. Vá à propriedade Columns e edite-a. Veja o editor pop-up do Delphi e insira alguns itens para testar.

Tudo funciona bem, mas não há nenhuma funcionalidade implementada que use nossa coleção. Deixo isso a cargo do leitor, como forma de exercitar os conhecimentos adquiridos neste artigo. Porém, você encontrará no pacote para download no final deste artigo o código já implementado.

Aprendendo a criar um bitmap para representar o componente na Component Pallete

Neste tópico "bônus" vamos ver como criar um bitmap para representar do nosso componente na Component Palette na IDE do Delphi.

O segredo está em um tipo de arquivo chamado Component Resource File (Arquivo de recurso de componente), que pode ser criado através do Image Editor, que acompanha o Delphi. Este arquivo pode armazenar muitos recursos relativos a um componente.

Abra, então, o Image Editor. Em seguida, clique no menu File, New, Component Resource File (.dcr). Na janela de edição que irá aparecer, clique com o botão direito do mouse sobre Contents, escolha New e em seguida Bitmap. Na caixa Bitmap properties, informe em Size o tamanho da imagem, que deve ser do tipo 24x24, escolha, em Colors, VGA (16 colors) (deve ser este o esquema de cores), e clique em OK. Aparecerá em Contents um novo item, inicialmente chamado de Bitmap1. Dê um duplo-clique sobre ele para editar a imagem do componente.

Depois de concluído o desenho da imagem, feche o editor (da imagem), e de volta a grade Contents, selecione Bitmap1, clique com o botão direito sobre ele e escolha Rename. Digite então a classe do componente que será representado pelo bitmap na IDE do Delphi. Atenção agora, pois este nome deve estar todo em maiúsculas. Por exemplo, para nosso componente TChronometer, digite TCHRONOMETER. Terminado, clique no menu File, Save. Atenção novamente aqui: o nome do arquivo deve ser o mesmo da unit onde está declarada a classe do componente. Tomando como exemplo novamente nosso cronômetro, o arquivo DCR deveria se chamar Chronometer.dcr. Não se esqueça de salvar este arquivo na mesma pasta da unit do component.

Isto é o necessário para criar um bitmap que representará seu componente na Component Palette. Se você criou o arquivo DCR depois do desenvolvimento do componente, faça isto então: remova a unit do componente do pacote e adicione-a novamente. Repare que o arquivo DCR irá constar na relação de arquivos do package. Depois que o pacote for reconstruído e instalado, o novo bitmap irá estar na Component Palette representando seu componente.

Aqui termina este artigo sobre criação de componentes, onde tentei e espero ter conseguido ensinar um pouco sobre este fascinante mundo dentro do universo da programação. Desejo que com o conhecimento adquirido através destas linhas sirvam para minimizar e agilizar o processo de desenvolvimento de você, leitor. Sugiro que continue no estudo deste segmento da programação, que este artigo passou longe de elucidar complemente. Ainda existem muitas outras técnicas que podem melhorar ainda mais o uso e a funcionalidade de componentes, como editores de propriedades e editores de componentes. Infelizmente, se fosse abordar tudo isso aqui, então já não seria mais um artigo de programação, estando mais para um livro! Mantenham-se firmes no estudo e lembrem-se que conhecimento nunca é demais.

Um forte abraço a todos!

por Michael Benford
Delphi na veia (!)