Oráculo da ClubeDelphi

Guinther Pauli
(guinther@devmedia.com.br)
Atua no ramo de tecnologia e programação há mais de 17 anos, é autor de mais de 100 artigos publicados e 200 vídeo-aulas e do livro "Delphi – Programação para Banco de Dados e Web". É Bacharel em Sistemas de Informação, Microsoft Certified: MCP, MCAD, MCSD.NET, Borland Certified: Delphi 6, 7, 2005, 2006, Web e Kylix. Editor Geral das Revistas .net Magazine, ClubeDelphi, WebMobile (.NET) e Mr.Bool. Palestrante em vários eventos pelo Brasil, como TechDay SP,RJ,POA,BH, Web Days e todas as edições da Borland Conference.

Na edição 62 da Revista ClubeDelphi, publiquei uma coluna chamada "Oráculo da ClubeDelphi". A idéia seria apresentar situações reais que são enfrentadas no dia a dia de um desenvolvedor Delphi ou que não fossem tão triviais, não aquelas que você encontra em uma simples busca no Google ou nos demos do Delphi, mas problemas mais "bruxos".

O que apresento aqui, em mais uma versão da coluna, são quatro dúvidas para alguns problemas relacionados ao desenvolvimento com o Delphi. Antes de iniciar, lembrem-se que jamais se pergunta para um desenvolvedor Delphi se "dá para fazer determinada coisa", mas "para quando ele pode fazer", porque não há limites para quem tem um Delphi aberto na sua frente.

Missão Impossível?

Em certa ocasião, trabalhei para o CPD de uma universidade em Santa Maria no RS. Desenvolvíamos alguns sistemas internos, como o controle financeiro e vestibular. Porém, o sistema de biblioteca era terceirizado. Os alunos tinham um cartão pessoal (e intransferível) que utilizavam para retirar livros na biblioteca, através de uma leitura do cartão por código de barras.

Alguns alunos mais "espertinhos", e sem cadastro na instituição, utilizavam o cartão de outras pessoas para retirarem livros. A solução foi fazer toda a universidade tirar foto para ser examinada no momento da retirada do livro, já que o cartão não possuía foto.

Aí começaram os problemas. Foi requisitado ao CPD que construísse um sistema que exibisse a foto do usuário no terminal de empréstimo justamente no momento em que o cartão de código de barras fosse passado no sistema. A matrícula aparecia em um *TextBox* / *Edit* do sistema externo e precisaríamos interceptar essa matrícula, localizar a foto, e

exibir no canto da tela. Porém, tínhamos algumas pedras no caminho. Como iríamos fazer o sistema exibir a foto, se:

1. Não tinhamos os fontes do sistema de biblioteca;
2. Ele não foi feito em Delphi;
3. Não tínhamos como modificá-lo.

A API do Windows é repleta de surpresas. Apesar de usar tipos exotéricos e não ser OO, você pode fazer um sistema inteiro, totalmente funcional, só usando API. Para algumas situações, precisamos recorrer a esse artifício para solucionar problemas mais complexos ou não suportados diretamente pela VCL do Delphi.

Bom, como não existem limites para um desenvolvedor Delphi, topamos o desafio, e achamos essa solução na API do Windows. Vou simular aqui uma solução bastante semelhante ao que fiz na época.

No exemplo deste artigo construí primeiramente um formulário chamado "Server", com um "TextBox" para ser informado um texto (nesse caso um nome). Esse form representa o form de empréstimo onde o usuário fornecia a matrícula. Como precisava ser em outra linguagem, construí a aplicação em Windows Forms com C#. A **Figura 1** apresenta o layout do form, que não tem código.

Em seguida, construí o formulário Delphi, que no caso representaria o form que exibiria a foto. O layout está na **Figura 2**. Um *ClientDataSet* está ligado ao arquivo *biolife.xml* dos demos do Delphi, pela propriedade *FileName*. Um *DBText* e um *DBImage* ligam, via *DataSource*, aos campos *Common_Name* e *Graphic* da tabela *biolife*. O *Timer* é explicado a seguir. O objetivo é: quando um nome ou iniciais forem digitados no *TextBox* na aplicação "Server", vamos dar um *locate* no respectivo registro no form Delphi e exibir os dados (como se fosse a foto da biblioteca, o procedimento é o mesmo).

Figura 1. Formulário em C#

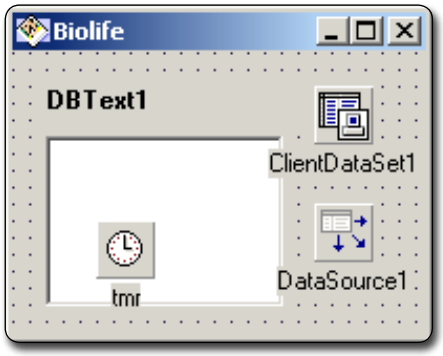


Figura 2. Formulário Delphi

A grande pergunta para achar a solução para o problema: "Como descobrir o que está digitado no *TextBox*?". Vamos começar pelo código do *Timer* (**Listagem 1**).

No Windows, toda janela tem uma *Handle*, que na verdade é um número. Exato, tipos no Windows são números! O primeiro passo foi localizar a janela *Server* pelo seu nome, usando a API *FindWindow*. O *Handle* da janela é armazenado na variável *Programa*. Pronto, já temos acesso à janela servidora. A seguir, chamamos *GetWindowRect* para obter as dimensões do form *Server*.

EnumChildWindows é uma função interessante. Ela funciona assim, passamos para ela a *Handle* do form desejado e um ponteiro para uma função de *CallBack*. Ou seja, para cada controle filho que existir no form encontrado, a função de *CallBack* será chamada passando os devidos parâmetros. É o que o nome da função diz, "enumere as janelas (ou controles) filhos". Em uma dessas iterações, será localizado o nosso tão desejado *TextBox*. *ListaFilhos* é nossa função de *callback* e é exibida na **Listagem 2.**

Lembrando que como essa função é chamada para cada controle que está dentro do form *Server*, precisamos saber se o controle varrido é o *TextBox* desejado. A técnica usada consiste em testar a posição do controle. O primeiro passo é capturar a posição e tamanho do controle filho. Comparamos então esses valores com as

Listagem 1. Código do evento OnTimer

```
implementation
var
  MainForm: TMainForm;
  Texto: string;
  PRect: TRect;
....
procedure TMainForm.tmrTimer(Sender: TObject);
var
  Programa: THandle;
begin
   Programa := FindWindow(nil,pchar('Server'));
   if programa <> 0 Then
   begin
      GetWindowRect(Programa,Prect);
      EnumChildWindows(Programa,@ListaFilhos,0);
      if Texto <> '' then
        ClientDataSet1.Locate('Common_Name',Texto, [loCaseInsensitive,loPartialKey]);
   end;
end;
```

Listagem 2. Função de CallBack

```
function ListaFilhos(Win: THandle;
    lp: LPARAM): Boolean; stdcall;
var
 pc : pchar;
 loop : integer;
 str : string;
 Pos : TRect;
begin
  result:=true;
  getwindowrect(win,Pos);
  if PRect.left - pos.left = -65 then
    if Prect.top - pos.Top = -35 then
    begin
      (* Aloca buffer *)
        str:='';
        for loop := 1 to 300 do
            str:=str + ' ';
        (* Obtém um ponteiro para o buffer *)
        pc:=strnew(pchar(str));
        (* Manda uma mensagem para a janela *)
        sendmessage(Win,WM_GETTEXT,300,cardinal(pc));
        (* Texto é o desreferencimanto do ponteiro *)
        Texto := string(pc);
        strdispose(pc); (* Libera buffer *)
    end;
end;
```

dimensões do controle pai, e se as diferenças relativas forem as informadas (valor do deslocamento X e Y), então localizamos o *TextBox*, representado pela variável *Win*. Observe que o valor informado para o deslocamento precisa ser "hard-coded".

A seguir, alocamos um *buffer* de dados com *StrNew* e enviamos uma mensagem ao controle *TextBox* (com *sendMessage*) solicitando que ele coloque o seu texto em nossa variável que será enviada de volta à aplicação Delphi. O código da mensagem é WM_GETTEXT.

Em outras palavras, é como se o programa falasse: "Form Server, já descobri que tu tens um TextBox que contém algo que eu preciso, estou te enviando um pacote para você colocar nele o conteúdo que necessito, depois mande-me de volta, please". Pronto! Conseguimos recuperar o texto do *TextBox*.

E isso finaliza nosso exemplo. O *Locate* no Delphi que já codificamos no *Timer* se encarrega de exibir o "peixinho" correto de acordo com suas iniciais. Veja o resultado na **Figura 3**.

Processamento paralelo "sem" TThread

Quase todo mundo sabe que para processar uma tarefa em segundo plano em Delphi é preciso usar *Threads*. Criamos uma classe descendente de *TThread*, sobrescrevemos o método virtual abstrato *Execute* e codificamos nossa tarefa. Mas sinceramente, isso não é tão produtivo como colocar um componente no form e configurar um evento, certo? Muitas vezes, deixando um pouco o purismo da OO de lado, é bem mais fácil aderir ao EDP (*Event Driven Programming*) para ganhar velocidade e produtividade no desenvolvimento.

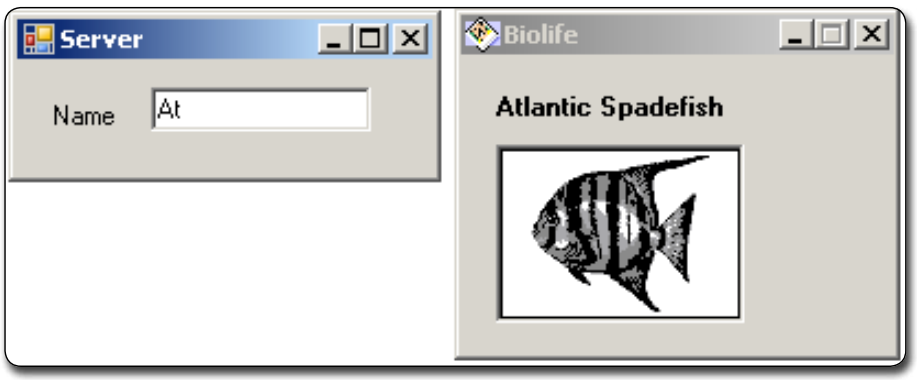


Figura 3. Pesquisando em um form Delphi a partir de um form C#

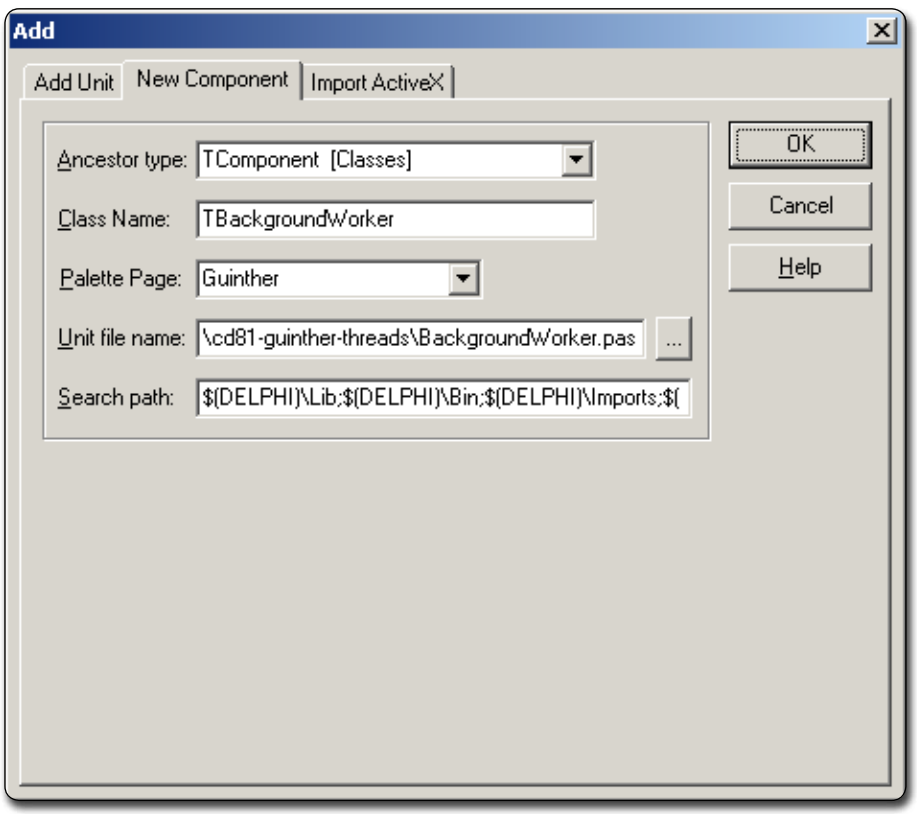


Figura 4. Criando o componente TBackgroundWorker

Quando comecei a estudar o .NET Framework 2.0, conheci um interessante controle chamado *BackgroundWorker*. Ele funciona de forma muito simples, você o coloca no form, no seu evento *DoWork* codifica o que gostaria de executar em segundo plano e depois simplesmente chama seu método *RunWorkerAsync* para executar o código de forma assíncrona (processamento paralelo). Bem mais RAD que a solução da VCL.

Mas o que não dá para fazer com Delphi? Talvez chover (ainda), então resolvi criar meu próprio *BackgroundWorker* para VCL Win32. O desenvolvimento ficou muito mais RAD e simples, e orientado a eventos, nada de criar classes descendentes de *TThread* (não que isso seja ruim, é apenas uma outra forma de ver as coisas). Vamos lá!

Primeiro criamos um *Package*, acessando o respectivo item no *Object Repository* (*File>New>Other>Package*). Salve o pacote como *pkBackgroundWorker.dpk*. No editor que aparece, clique em *Add* para adicionar um novo componente ao pacote, preenchendo as informações conforme a **Figura 4**. Observe que demos o apropriado nome de *TBackgroundWorker* à nova classe, que descende de *TComponent*. Clique em OK. Codifique a classe como mostrado na **Listagem 3.**

Se o leitor observar atentamente o código, vai perceber a "jogada". O que fizemos foi encapsular a classe *TThread* dentro de um componente, para que você literalmente nunca mais precise vê-la. Esse é um padrão de projeto bastante conhecido, chamado *Wrapper*.

A nossa classe *TBackgroundWorker* tem uma propriedade do tipo evento chamado *DoWork* (exatamente como no .NET), e seu respectivo campo privado *FDoWork*. Veja que adicionamos eventos a um componente como propriedades do tipo

TNotifyEvent (para o tipo mais básico).

O método *RunWorkerAsync* nada mais faz do que verificar se o evento *DoWork* está apontando para alguma coisa e o chama (esse seria o código que o usuário do componente gostaria de executar em paralelo). Porém, ao invés de chamá-lo diretamente (síncrono), passa um ponteiro para o evento para ser chamado a partir de nossa *Thread* interna, que chama o evento de forma assíncrona. A *Thread* recebe esse evento na variável *FMethod* e o chama no método virtual *Execute*.

Voltando ao editor do pacote, clique no botão *Install* para instalar o componente no IDE. Crie uma nova aplicação e vamos fazer nosso primeiro exemplo com o novo *TBackgroundWorker*. Coloque no form um botão e um *Edit*. No botão digite:

```
BackgroundWorker1.RunWorkerAsync;
```

Coloque o controle *TBackgroundWorker* no form e no seu evento *DoWork* digite o código da **Listagem 4.** O código tem um loop infinito, que atribui um número randômico ao *Edit*. Em uma situação normal, isso travaria a aplicação, aqui ela continua funcionando normalmente. Você pode mover o form, redimensioná-lo etc. Ou seja, a aplicação não "trava"! E não foi preciso criar *TThread*, você fez tudo isso usando o melhor do RAD do Delphi.

Clicando em *View>Debug Windows>Threads* (*Ctrl+Alt+T*) podemos comprovar que realmente existem duas linhas de execução na aplicação (*Threads*). Uma é a principal, outra é a que está processando nosso algoritmo de randomização (**Figura 5**).

Para um exemplo mais real, que tal abrir aquele relatório ou query enorme e pesada em background? Como exemplo, criei uma tabela "Foo" no Firebird e gerei 100 mil registros aleatórios, e coloquei um *SimpleDataSet* do dbExpress associado a essa tabela. Colocamos um segundo *TBackgroundWorker* e no seu evento *DoWork* digitamos simplesmente:

```
procedure TForm1.BackgroundWorker2DoWork(
    Sender: TObject);
begin
    SimpleDataSet1.Open;
end;
```

Coloque um botão para chamar o método *RunWorkerAsync* do *BackgroundWorker2*. Rode a aplicação e faça o teste. Veja que a aplicação não trava mesmo abrindo uma query tão pesada. E me-

Listagem 3. Código do TBackgroundWorker

```
unit BackgroundWorker;

interface

uses
  SysUtils, Classes;

type
  TMyThread = class(TThread)
  private
    FMethod: TNotifyEvent;
  protected
     procedure Execute(); override;

  public
     constructor Create(CreateSuspended: boolean; AMethod: TNotifyEvent);
  end;

  TBackgroundWorker = class(TComponent)
  private
    FDoWork: TNotifyEvent;
  protected

  public
    procedure RunWorkerAsync;
  published
     property DoWork: TNotifyEvent read FDoWork write FDoWork;
  end;

procedure Register;

implementation

procedure TBackgroundWorker.RunWorkerAsync;
begin
   if Assigned(DoWork) then
     TMyThread.Create(false,DoWork);
end;

procedure Register;
begin
  RegisterComponents('Guinther', [TBackgroundWorker]);
end;

{ TMyThread }

constructor TMyThread.Create(CreateSuspended: boolean;
  AMethod: TNotifyEvent);
begin
  inherited Create(CreateSuspended);
  FMethod := AMethod;
end;

procedure TMyThread.Execute;
begin
  FreeOnTerminate := True;
  FMethod(nil);
end;

end.
```

Listagem 4. Evento DoWork

```
procedure TForm1.BackgroundWorker1DoWork(Sender: TObject);
var
 i: integer;
begin
  randomize;
  while true do
    Edit1.Text := IntToStr(Random(10000));
end;
```

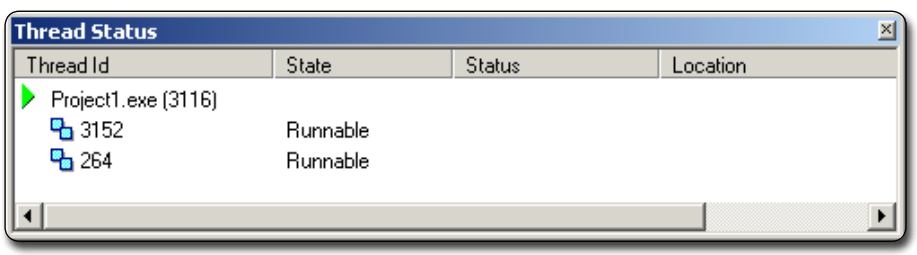


Figura 5. Threads rodando simultaneamente

lhor, aproveite para clicar também no botão *Random* e comprove que podemos ter múltiplas tarefas sendo executadas simultaneamente (**Figura 6**).

> **Nota:** Para mais informações sobre Threads, consulte meu artigo na edição 20 ou o utilitário *ThreadManager* na minha página de projetos no CodeCentral da CodeGear, no link *http://cc.codegear.com/Author/222668*

Adicionando uma propriedade em um TForm no Object Inspector

O uso de Propriedades é um dos principais fundamentos da Programação Orientada a Objetos. E não pense que você precisa criar sistemas enormes e complexos para desfrutar das maravilhas da OO. Mesmo em situações muito simples, uma técnica OO bem empregada pode tornar seu código muito mais simples, elegante, modular e reaproveitável.

Lembro-me do meu primeiro exemplo com propriedades. Construir um form que tinha uma *StatusBar*. Então, a cada ação do usuário ou mudança de estado do sistema, eu mudava a descrição no controle para indicar o que estava acontecendo. Ex.:

```
StatusBar1.SimpleText := 'Inserindo registro';
...
StatusBar1.SimpleText := 'Dados gravados com
  sucesso';
...
```

Mas e se um dia eu resolvesse trocar a *StatusBar* por outro controle, por exemplo um *Panel*? Ou exibir as mensagens em caixas de mensagem? Precisaria obviamente trocar as referências em vários lugares.

Resolvi o problema encapsulando o acesso à *StatusBar* em um único ponto central do código, criando uma propriedade somente escrita, como mostra a **Listagem 5.**

Agora, para atribuir uma mensagem à *StatusBar*, simplesmente escrevemos:

```
Mensagem := 'Dados gravados com sucesso';
```

Se uma modificação de controle fosse feita, um único local do código precisaria ser modificado.

Mas veja que essa propriedade não aparece no *Object Inspector*. Esse ponto é um pouco mais complexo. Vamos então criar um segundo exemplo que vai adicionar uma propriedade a um formulário, em tem-

Listagem 5. Propriedade em um form

```
TForm1 = class(TForm)
  StatusBar1: TStatusBar;
private
  procedure SetMensagem(const Value: string);
public
  property Mensagem: string
     write SetMensagem;
end;

implementation
...
procedure TForm1.SetMensagem(const Value: string);
begin
   StatusBar1.SimpleText := Value;
end;
```

Listagem 6. Usando SetPropValue

```
uses TypInfo;
...
procedure TForm1.Button1Click(Sender: TObject);
begin
SetPropValue(
      FindComponent(Edit1.Text), //componente
      Edit2.Text, //propriedade
      Edit3.Text); //novo valor
end;
```

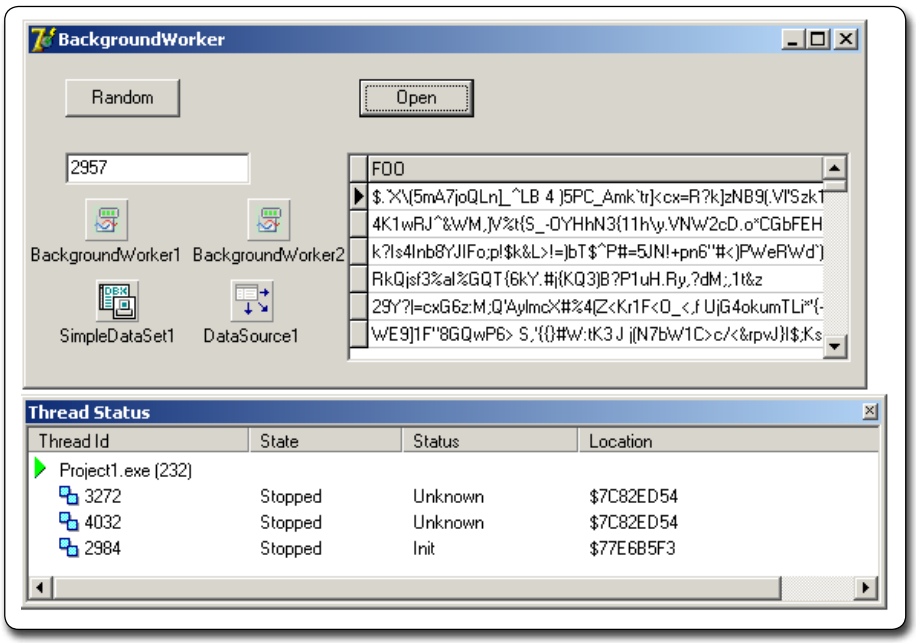


Figura 6. Múltiplas tarefas sendo executadas em paralelo

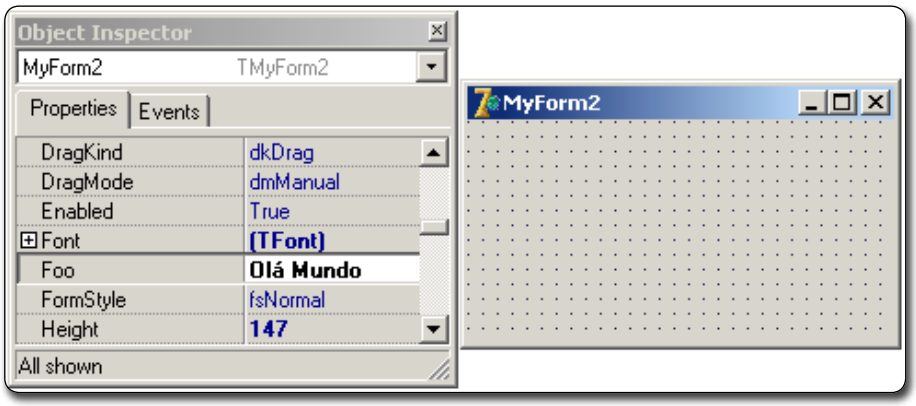


Figura 7. Adicionando uma propriedade a um formulário

po de design. Siga os seguintes passos.

1. Crie um pacote e adicione nele um formulário. Chame-o de "MyForm" e dê o nome da unit de "uMyForm.pas". Para ver como criar pacotes, veja o item 1 deste artigo. Chame o pacote de "pkMyForm.dpk";
2. Na unit do form, declare o seguinte:

```
// aqui é o final da declaração da classe
procedure Register;
...
uses
  DesignIntf, DesignEditors;
...

procedure Register;
begin
  RegisterCustomModule(TMyForm, TCustomModule);
end;
```

3. Declare uma propriedade de exemplo no form, e após pressione *Shift+Ctrl+C* (repare que ela tem que ser *published*):

```
published
   property Foo: string
        read FFoo write SetFoo;
end;
```

4. No editor do pacote, na seção *Requires*, adicione o arquivo *designide.dcp* que se encontra no diretório *Lib* do Delphi;
5. Instale o pacote no IDE clicando em *Install*;
6. Abra o form, clique de direita sobre ele e acione a opção para adicioná-lo ao repositório (*Add to Repository*). Preencha as informações do caixa de diálogo e clique em Ok.

Para testar, inicie uma nova aplicação, abra o *Object Repository* e localize nosso form adicionado. Atenção, certifique-se que a opção *inherit* esteja marcada ao invés de *copy*. A propriedade deverá então aparecer no form em design time, totalmente funcional (**Figura 7).**

RTTI

A RTTI (*RunTime Type Information*) é um importante fundamento do Delphi. É amplamente utilizado pelo IDE, desde o momento que você abre o Delphi. Por exemplo, o Delphi usa RTTI para ler e salvar arquivos DFM e configurar propriedades de seus controles. Com RTTI, podemos obter informações de tipo em tempo de execução, e resolver uma série de problemas que muitas vezes não são supridos diretamente pela linguagem.

Você pode não saber, mas já usou RTTI com Delphi. Se já usou os operadores *is* e *as*, então fez uso da técnica. Mas os recursos desse poderoso artifício não param por aí. A unit *typinfo* traz uma série de rotinas que podem ajudar e muito o desenvolvedor a resolver problemas do dia a dia, de maneira simples e direta (sugiro o leitor abrir os fontes da unit e examiná-la).

Neste último tópico, vou propor dois exemplos que fazem uso avançado de RTTI. O primeiro será um form que o usuário poderá personalizar controles, e o segundo será um "mini-IDE", onde o usuário poderá desenhar forms. Quem sabe você não usa a idéia para que seu próprio usuário desenhe a tela do sistema da maneira como ele deseja, e você só importa para o Delphi? Vamos lá!

Em uma nova aplicação, coloque no formulário principal três *Edits* e um *Button* dentro de um *Panel*, e logo abaixo deles, alguns controles a sua escolha, como *Labels, Edits, ComboBoxes, ListBoxes, Buttons* etc. Seu formulário deve estar semelhante à **Figura 8.** No *OnClick* do botão *Setar* digite o código da **Listagem 6.**

SetPropValue é uma importante rotina da unit *typinfo* que permite configurar o valor de qualquer propriedade de qualquer controle baseado no **nome** do componente, sem depender do seu tipo. É uma rotina muito flexível e bastante utilizada para resolver uma série de problemas em

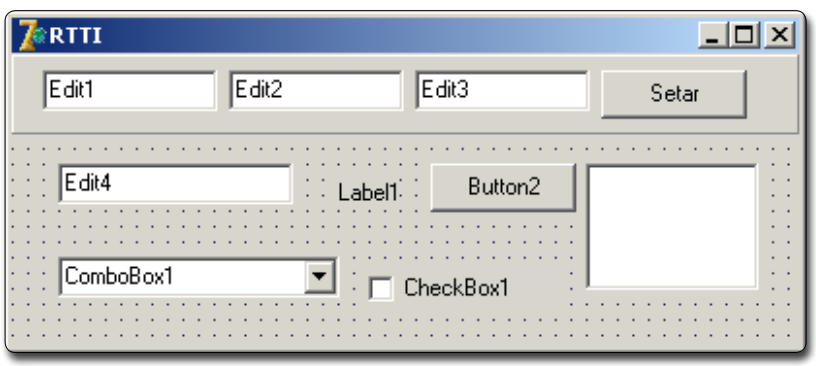


Figura 8. Formulário para teste de RTTI

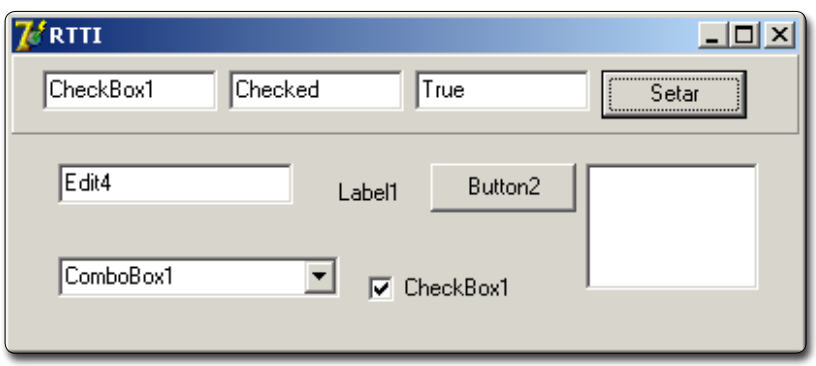


Figura 9. Usando RTTI

.net PLUS www.devmedia.com.br/msdn/portal.asp
Acesse agora o mesmo o portal do assinante ClubeDelphi e assista a uma vídeo-aula de Guinther Pauli que mostra como usar TThreads.
www.devmedia.com.br/articles/viewcomp.asp?comp=1733

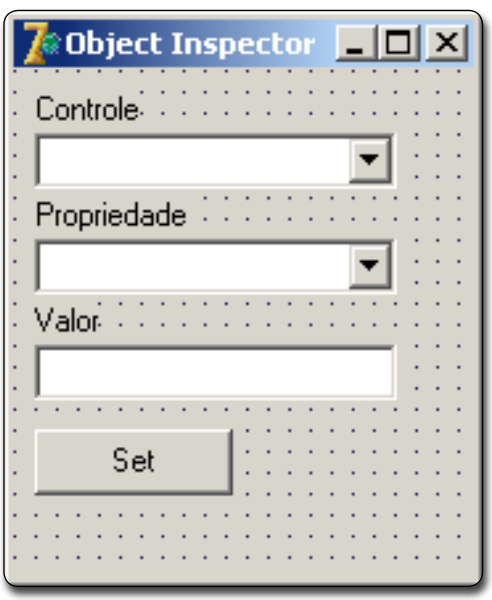


Figura 10. Object Inspector

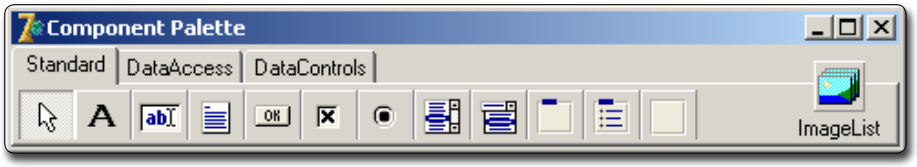
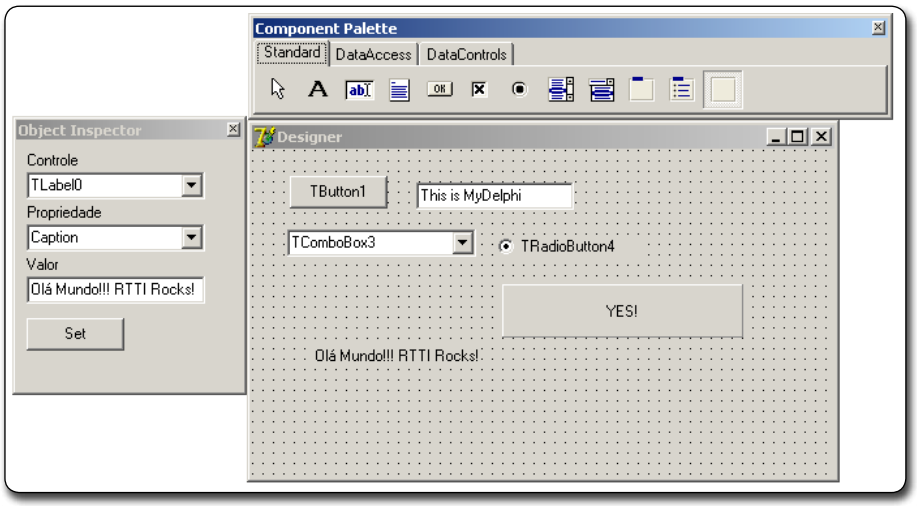


Figura 11. Paleta de componentes



Listagem 7. Código da unit UFormOI.pas

```
...
type
  TFormOI = class(TForm)
    ...
  public
     procedure GetComponetProperties(AComponent: TComponent);
     procedure GetComponentList;
...
implementation
...
uses TypInfo, uFormDesigner;
procedure TFormOI.GetComponentList;
var
  i: integer;
begin
   ComboBox1.Items.Clear;
   for i := 0 to FormDesigner.ControlCount - 1 do
     ComboBox1.Items.Add(FormDesigner.Controls[i].Name)
end;
procedure TFormOI.GetComponetProperties(
  AComponent: TComponent);
var
  PropCount,i : Integer;
  PropList : PPropList;
  PropName : string;
begin
  ComboBox2.Items.Clear;
  PropCount := GetPropList(AComponent.ClassInfo, PropList) - 1;
  for i := 0 to PropCount do
  begin
    PropName := PropList[i].Name;
    ComboBox2.Items.Add(PropName)
  end;
end;
procedure TFormOI.ComboBox1Change(Sender: TObject);
var
  obj: TComponent;
begin
  Obj := FormDesigner.FindComponent(ComboBox1.Text);
  GetComponetProperties(Obj);
end;
procedure TFormOI.Button1Click(Sender: TObject);
begin
   SetPropValue(FormDesigner.FindComponent(
     ComboBox1.Text),ComboBox2.Text,Edit1.Text);
end;
procedure TFormOI.FormShow(Sender: TObject);
begin
  GetComponentList;
end;
```

estruturas de programas que precisam depender de vários tipos de dados.

Execute a aplicação e faça um teste, digite no primeiro *Edit* o nome de um componente que esteja no form, no segundo o nome de uma de suas propriedades e no terceiro um valor. Ao clicar no botão, o valor é alterado, tudo pelo nome e em tempo de execução! No exemplo da **Figura 9**, modifiquei o *Checked* do *CheckBox* para *True*. Faça outros testes, mude *Labels*, posições, tamanhos etc.

Seguindo essa mesma idéia, vamos um pouco mais longe, vamos criar um "Delphizinho". Será uma aplicação semelhante a anterior, só que porém teremos uma paleta de componentes e um mini-*object inspector*. O usuário será capaz de mover os controles pela tela, e para finalizar, o formulário desenhado é persistido no disco.

Primeiro, crie uma aplicação com três formulários, com os nomes de "FormDesigner", "FormOI" e "FrmPalette". Certifique-se que todos estejam com *Visible = True*. O *FormOI* pode ser visto na **Figura 10,** nele temos dois *ComboBoxes*, três *Labels* e um *Button*. O *FormDesigner* ficará em branco por enquanto.

O *FrmPalette* (**Figura 11**) tem a aparência da paleta de componentes do Delphi. Para construí-la, usamos um *PageControl*, dentro de cada aba existe uma *ToolBar*, onde cada componente é representado por um botão (atenção, o nome do botão deve ser o nome da classe que representa, exemplo *TLabel*, *TButton*). Para este exemplo, coloque somente os controles básicos do Delphi.

No *FrmPalette*, declare a seguinte variável pública:

```
public
    SelectedClass : TClass;
```

Dando um clique em qualquer botão da *ToolBar*, digite o código a seguir. Depois, faça com que todos os demais botões apontem para o mesmo evento.

```
procedure TFrmPalette.PaletteClick(
  Sender: TObject);
begin
  SelectedClass := GetClass((Sender as
    TComponent).Name);
end;
```

Esse é todo o código do form. Basicamente o que estamos fazendo é usando RTTI para obter uma referência para uma

classe na qual o usuário clicou, pois quando ele clicar no *FormDesigner*, devemos criar um objeto desse tipo.

A **Listagem 7** mostra o código do *FormOI*, o nosso *Object Inspector*. O método *GetComponentList* varre o *FormDesigner* procurando por componentes, e vai preenchendo o primeiro *ComboBox* com os seus nomes. Quando o usuário seleciona um componente nessa lista, devemos gerar a lista de suas propriedades, novamente usando RTTI através da função *GetPropList*. Colocamos essas propriedades no segundo *ComboBox*. No botão, configuramos o valor da propriedade escolhida com base no valor digitado no *Edit*.

A **Listagem 8** mostra o código do *FormDesigner*. O método *NewControl*, como o nome sugere, criar um novo controle baseado no botão selecionado na paleta de componentes, usando RTTI. Isso acontece quando o usuário clica com o mouse sobre o form, veja o método *FormMouseDown*.

O *ControlMouseDown* é interessante, é nele que permitimos que o usuário arraste controles pela tela em runtime, semelhante ao designer do Delphi. Para isso, usamos a mensagem *WM_SysCommand* da API do Windows. O *Create* e o *Close* do form se encarregam de salvar e carregar o formulário, ou seja, persisti-lo. E por fim, no *initialization* da unit, precisamos registrar as classes que vamos trabalhar na aplicação. A **Figura 12** mostra nosso "Delphizinho" em execução.

Conclusão

Neste artigo vimos como a API pode resolver problemas que vão além dos limites da VCL. Nosso *BackgroundWorker* vai ajudá-lo agora a criar processamento paralelo em suas aplicações de forma extremamente RAD. Formulários agora têm propriedades no Object Inspector, tornando-os muito mais flexíveis. E finalmente, vimos que a RTTI é realmente uma técnica poderosa para explorar alguns recursos mais "obscuros" do Delphi.

Amigos leitores, não deixem de enviar suas dúvidas por e-mail para que possamos trazer sempre mais e mais dicas interessantes aqui nesta coluna. Espero que tenham aproveitado o conhecimento adquirido e lembrem-se, não existem limites para quem tem um Delphi. Um grande abraço e até a próxima! ●

Listagem 8. Código da unit UFormDesigner.pas

```
...
type
  TFormDesigner = class(TForm)
    procedure FormMouseDown(Sender: TObject; Button: TMouseButton;
       Shift: TShiftState; X, Y: Integer);
    procedure FormClose(Sender: TObject;
      var Action: TCloseAction);
  private
    procedure NewControl(ClassRef: TClass; x, y: integer);
    procedure ControlMouseDown(Sender: TObject; Button: TMouseButton;
      Shift: TShiftState; X, Y: Integer);
  public
     constructor create (AOwner: TComponent); override;
  end;
...
implementation

uses uFrmPalette, uFormOI;

{$R *.dfm}

type
 TGetProtected = class (TControl)
 end;

procedure TFormDesigner.NewControl(ClassRef: TClass; x, y: integer);
var
 NewObj : TControl;
begin
 if ClassRef<>nil then
 begin
   NewObj:=TControlClass(
     FrmPalette.SelectedClass).Create(self);
   NewObj.Parent:=self;
   NewObj.top:=y;
   NewObj.left:=x;
   TGetProtected(NewObj).OnMouseDown:=ControlMouseDown;
   NewObj.Name:=NewObj.ClassName+IntToStr(NewObj.ComponentIndex);
   FrmPalette.SelectedClass:=nil;
   formOi.GetComponentList;
 end;
end;

procedure TFormDesigner.FormMouseDown(Sender: TObject; Button: TMouseButton;
  Shift: TShiftState; X, Y: Integer);
begin
  NewControl(FrmPalette.SelectedClass,x,y);
end;

procedure TFormDesigner.ControlMouseDown(Sender: TObject; Button: TMouseButton;
   Shift: TShiftState; X, Y: Integer);
begin
  ReleaseCapture;
  TControl(Sender).Perform(WM_SysCommand, $F012, 0);
end;

constructor TFormDesigner.create(AOwner: TComponent);
var
  i: integer;
begin
  if FileExists('Designer.dfm') then
  begin
    inherited CreateNew(AOwner) ;
    ReadComponentResFile('Designer.dfm', self) ;
    for i := 0 to controlCount - 1 do
      TGetProtected(Controls[i]).OnMouseDown:=ControlMouseDown;
  end
  else
    inherited Create(AOwner);
end;

procedure TFormDesigner.FormClose(Sender: TObject; var Action: TCloseAction);
begin
 WriteComponentResFile('Designer.dfm', self) ;
end;

initialization
  RegisterClass(TButton);
  RegisterClass(TEdit);
  RegisterClass(TLabel);
  RegisterClass(TMemo);
  RegisterClass(TCheckBox);
  RegisterClass(TRadioButton);
  RegisterClass(TListBox);
  RegisterClass(TComboBox);
  RegisterClass(TGroupBox);
  RegisterClass(TRadioGroup);
  RegisterClass(TPanel);
  RegisterClass(TDBGrid);
  ...
//adicione no uses as units respectivas a esses tipos
```