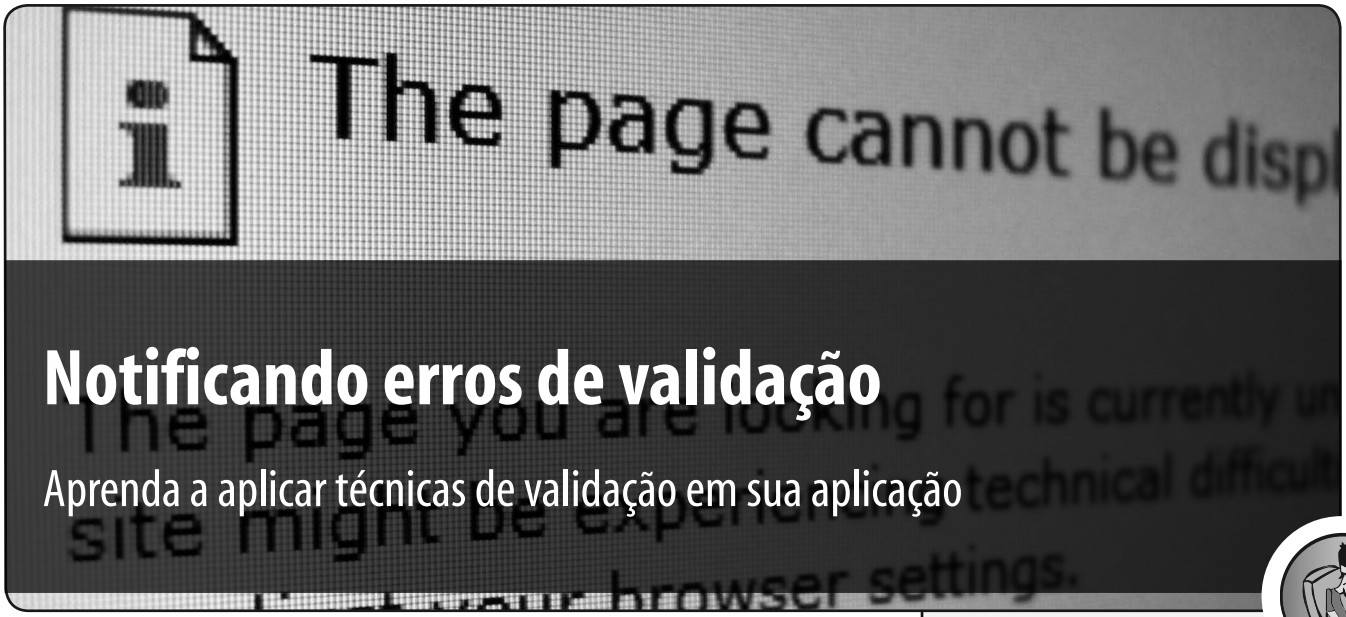
Nesta seção você encontra artigos sobre técnicas que poderão aumentar a qualidade do desenvolvimento de software



Notificando erros de validação

Aprenda a aplicar técnicas de validação em sua aplicação

Paulo Roberto Quicoli
(pauloquicoli@gmail.com)
é analista e programador da Control-M Informática. Trabalha com Delphi, desde sua primeira versão, e Firebird desenvolvendo aplicações cliente-servidor. Formado em Tecnologia de Processamento de dados pela FATEC, na cidade de Taquaritinga/SP. Blog: *http://pauloquicoli.spaces.live.com*

Ao observarmos a evolução dos aplicativos *Windows*, incluindo o próprio sistema operacional, vemos uma dedicação especial dos desenvolvedores para os recursos visuais. Por recursos visuais quero dizer a própria aparência em si e também a experiência do usuário no que diz respeito à usabilidade. Porém, nem só de visual vive uma aplicação. Seja *Win32* ou *Web*, uma aplicação precisa prever possíveis erros do usuário e notificá-lo, como um *CPF* ou *CNPJ* incorretos. Dessa forma teremos sempre um sistema visualmente atraente e ao mesmo tempo inteligente a ponto de prever erros e/ou falta de parâmetros em determinadas partes do sistema.

Neste artigo vamos desenvolver um pequeno *framework* de notificação de erros que tem como objetivo melhorar essa experiência do usuário, provando que um bom sistema precisa ir muito mais além do que o visual.

Resumo DevMan

Com cada vez mais aplicações no mercado, tornar nossa alternativa mais atraente e funcional passou a ser ainda mais importante. Alertar o usuário de possíveis erros é uma premissa importante no desenvolvimento de qualquer aplicação. Veremos nesse artigo como implementar algumas classes para a exibição de erros de forma funcional e visual.

Nesse artigo veremos

- Orientação a Objetos;
- Criação de Eventos;
- Criação de um notificador para nossas aplicações.

Qual a finalidade

- Desenvolver um pequeno framework que exiba de forma agradável ao usuário mensagens de validação.

Quais situações utilizam esses recursos?

- Formulários para entrada de dados.

Clube101.indb 42 10.11.08 15:56:11

Funcionamento e estrutura básica

Para desenvolver o exemplo, tomei idéias do validador do *.net* e também do *JVCL Validator*. O objetivo então é informar que um determinado controle visual – *TEdit, TDBEdit* – possui um valor não válido, seja mudando o controle de cor, movimentando o controle ou incluindo um ícone de alerta ao lado do mesmo. Também deve permitir obter, em forma de lista, todos os controles que não passaram nas validações, porém o código de validação em si não é de responsabilidade do *framework*, ou seja, é necessário apenas informar que um determinado controle está *ok* ou não. Na **Figura 1** vemos como nosso *ErrorNotifier* está dividido.

O *ErrorNotifier* é dividido em duas partes. O centro de notificação, que armazena os controles inválidos é um notificador, que é responsável por realizar a real notificação. Dessa forma, estamos aplicando o conceito de *responsabilidade de objetos*", onde não se deve acumular várias rotinas em uma mesma classe e sim dividi-las em classes, distribuindo a responsabilidade.

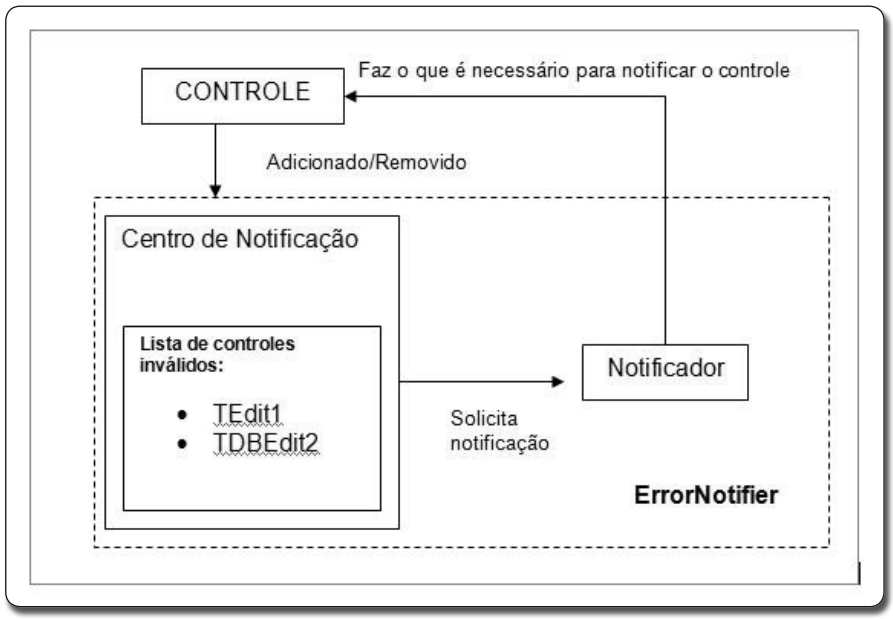


Figura 1. Funcionamento do ErrorNotifier

Listagem 1. Código da classe TInvalidControl

```
TInvalidControl = class
private
  FAlertHint: string;
  FOriginalHint: string;
  FControl: TWinControl;
  procedure SetAlertHint(const Value: string);
  procedure SetControl(const Value: TWinControl);
  procedure SetOriginalHint(const Value: string);
public
  constructor Create(pControl: TWinControl; sOriginalHint, sAlertHint: string);
  property Control: TWinControl read FControl write SetControl;
  property OriginalHint: string read FOriginalHint write SetOriginalHint;
  property AlertHint: string read FAlertHint write SetAlertHint;
end;
```

Desenvolvimento

Aqui utilizei o Delphi 7, sinta-se à vontade para utilizar a versão que possuir. Inicie um novo projeto usando o menu *File>New>Application*, salve-o com o nome que preferir e em seguida crie uma nova *unit* usando o menu *File>New>Unit* salvando-a como *ErrorNotifier.pas*. Declare as seguintes classes:

```
TErrorNotifierCenter = class;
TInvalidControl = class;
THackWinControl = class(TWinControl);
```

Aqui temos nosso centro de notificação, representado pela classe *TErrorNotifierCenter*. Nele vamos adicionar os controles inválidos, que estão representados pela classe *TInvalidControl*. Essa classe irá conter algumas informações extras que serão utilizadas pelo *Notificador*, como a mensagem de erro a ser exibida, o controle que está inválido e o *hint* atual do controle. Para que possamos notificar vários tipos de controles, *TInvalidControl* não irá conter diretamente um controle do tipo *TEdit* ou *TDBEdit*, e sim um *TWinControl*, que está na hierarquia da maioria dos controles visuais de edição de dados. Definimos também a classe *THackWinControl* para que tenhamos acesso a alguns métodos protegidos de *TWinControl*. Implemente *TInvalidControl* como na **Listagem 1**.

Observe que temos apenas propriedades nessa classe, portando não vou listar aqui seu código, visto que são apenas *Get's* e *Set's*.

O Notificador

Para permitir flexibilidade vamos criar uma interface que deverá ser obedecida na implementação de qualquer *Notificador*, ou seja, ao final do *framework* qualquer desenvolvedor poderá criar uma nova forma de notificação, desde que siga a interface especificada. A seguir veja a declaração de nossa *interface*.

```
INotifier = interface
  ['{D76BA182-9E19-48E6-86CE-5EC69DCF7D8D}']
  procedure StartNotifying(pInvalidControl:
    TInvalidControl);
  procedure StopNotifying(pInvalidControl:
    TInvalidControl);
end;
```

Os dois métodos definidos recebem como parâmetro um objeto do tipo *TInvalidControl* e é através desse objeto que o controle será acessado e manipulado.

> **Nota:** Em *Win32* toda interface necessita de um identificador único. Esse identificador é o *GUID*, o código entre colchetes. Ele não pode ser qualquer número, deve ser gerado. No Delphi basta pressionar *CTRL+G* e o *GUID* será gerado.

Centro de notificação

Este é o núcleo do *framework*. É quem mantém a lista de controles inválidos e dispara a ordem de notificação. Na **Listagem 2** temos a estrutura da classe *TErrorNotifierCenter* e na seqüência os detalhes sobre seus métodos.

No *create* de nossa classe passamos um objeto que implementa *INotifier*, ou seja,

Clube101.indb 43 10.11.08 15:56:13

Listagem 2. Estrutura da classe TErrorNotifierCenter

```
TErrorNotifierCenter = class
private
  FNotifier: INotifier;
  FInvalidControlsList: TObjectList;
  FErrorList: TStringList;
  function ControlInList(pControl: TWinControl): integer;
  function GetFirstErroneousControl: TWinControl;
  function GetHasErros: boolean;
  function getErrorList: TStringList;
public
  constructor Create(Notifier: INotifier); reintroduce;
  destructor Destroy; override;
  procedure AddInvalidControl(pControl: TWinControl; AlertMessage: string);
  procedure RemoveInvalidControl(pControl: TWinControl);
  property ErrorList: TStringList read getErrorList;
  property HasErrors: boolean read GetHasErros;
  property FirstErroneousControl: TWinControl read GetFirstErroneousControl;
end;
```

Listagem 3. Adicionando um controle inválido

```
procedure TErrorNotifierCenter.AddInvalidControl(pControl: TWinControl;
  AlertMessage: string);
var
 index: integer;
begin
  if ControlInList(pControl) < 0 then
  begin
    Index := FInvalidControlsList.Add(TInvalidControl.Create(
      pControl,pControl.Hint, AlertMessage));
    FNotifier.StartNotifying(FInvalidControlsList[index] as TInvalidControl);
  end;
end;
```

Listagem 4. Procedimento RemoveInvalidControl

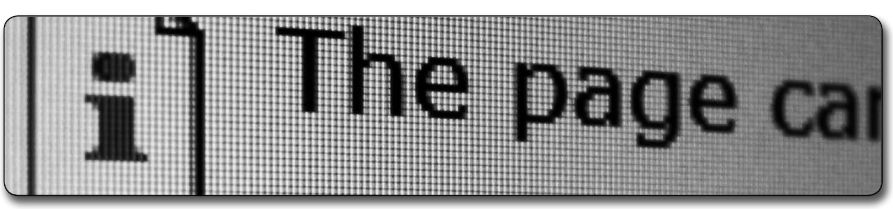
```
procedure TErrorNotifierCenter.RemoveInvalidControl(pControl: TWinControl);
var
 Index: integer;
begin
  Index := ControlInList(pControl);
  if Index >= 0 then
  begin
    FNotifier.StopNotifying(FInvalidControlsList[index] as TInvalidControl);
    FInvalidControlsList.Delete(index);
  end;
end;
```

Listagem 5. TdefaultNotifier

```
TDefaultNotifier = class(TInterfacedObject, INotifier)
private
  FDefaultColor: TColor;
  FErrorColor: TColor;
public
  constructor Create(DefaultColor, ErrorColor: TColor);
  procedure StartNotifying(pInvalidControl: TInvalidControl); virtual;
  procedure StopNotifying(pInvalidControl: TInvalidControl); virtual;
end;
```

Listagem 6. Implementação das procedure StartNotifying e StopNotifying

```
procedure TDefaultNotifier.StartNotifying(pInvalidControl: TInvalidControl);
begin
  THackWinControl(pInvalidControl.Control).Color := FErrorColor;
  pInvalidControl.Control.Hint := pInvalidControl.AlertHint;
end;
procedure TDefaultNotifier.StopNotifying(pInvalidControl: TInvalidControl);
begin
  THackWinControl(pInvalidControl.Control).Color := FDefaultColor;
  pInvalidControl.Control.Hint := pInvalidControl.OriginalHint;
end;
```



TErrorNotifierCenter conhecerá apenas os métodos definidos pela interface *INotifier*, dessa maneira podemos criar o *Notificador* que desejarmos, sem contudo, alterar uma linha de código no *ErrorNotifierCenter*. Neste construtor inicializamos os campos *FNotifier, FInvalidControlList* (*Lista de controles inválidos*) e *FErrorList* (*Lista com os erros existentes*).

Em seguida, declaramos uma *procedure* chamada *AddInvalidControl* como podemos ver a seguir:

```
procedure AddInvalidControl(pControl:
  TWinControl; AlertMessage: string);
```

Este é o método que será usado em nossos programas para *dizer* ao *framework* que um determinado controle está inválido. Passamos como parâmetro o controle e a mensagem de erro. Na **Listagem 3** temos sua implementação.

Aqui um objeto do tipo *TInvalidControl* é instanciado e armazenado na lista de controles inválidos *FInvalidControlsList* e logo após é solicitado ao *Notificador*, aquele passado como parâmetro do *Create*, que inicia a notificação do controle adicionado. Quero ressaltar aqui algo muito importante: veja que não criamos dependência entre quem mantém a lista dos controles e quem realiza a notificação. *TErrorNotifierCenter* não sabe, e nem precisa saber, como a notificação será realizada, ele apenas ordena: *Notificador, notifique o controle agora!*. Para que não corramos o risco de adicionar o mesmo controle duas vezes o método *ControlInList* é chamado.

Mais uma função é declarada em seguida, a *ControlInList*, como mencionado anteriormente. Veja sua declaração:

```
function ControlInList(pControl:
  TWinControl): integer;
```

Ela verifica se o controle passado como parâmetro já foi adicionado. Caso já tenha sido adicionado sua posição na lista é retornada. Essa verificação é feita utilizando um laço *for*, varrendo a lista e comparando a propriedade *name* dos controles.

Outro procedimento importante é o *RemoveInvalidControl:*

```
procedure RemoveInvalidControl(pControl:
  TWinControl);
```


Clube101.indb 44 10.11.08 15:56:16

Este é o método que será usado em nossos programas para *dizer* ao *framework* que um determinado controle está válido e deverá ser removido da lista. Na **Listagem 4** temos sua implementação.

Para também evitar que o usuário, por engano, tente retirar da lista um controle que não está lá, chamamos o método *ControlInList*.

```
function TErrorNotifierCenter.getErrorList:
  TStringList;
```

Essa função retorna uma lista com as mensagens de erro de cada controle existente na lista de controles inválidos. Essa lista pode posteriormente ser utilizada em um *ListBox* como uma espécie de log dos erros encontrados.

Criando um notificador

Como mencionado anteriormente um *notificador* deve implementar a interface *INotifier*. Nosso *notificador* irá deixar o controle em vermelho caso esteja inválido. Adicione uma nova classe na *Unit ErrorNotifier* e a defina como na **Listagem 5**.

No *Create* passamos duas cores que representam o estado do controle. *DefaultColor* quando o controle é válido e *ErrorColor* quando está inválido. Para saber como isso é feito, temos que observar os métodos *StartNotifying* e *StopNotifying* na **Listagem 6**.

Para acessarmos a propriedade *Color*, que é uma propriedade protegida, usamos nossa classe de *Hack* e então passamos a cor apropriada.

Colocando pra funcionar

Nesse momento, já temos nossas classes todas criadas e funcionando. Vamos ver agora como criar um aplicação e utilizá-las. Altere o formulário principal do projeto para que fique com a aparência semelhante à **Figura 2** e adicione a *Unit ErrorNotifier* na seção *Uses*. Se preferir utilizar o atalho *Alt + F11* e escolha a *unit ErrorNotifier*.

Na seção *private* do formulário adicione um campo chamado *FCentralErros* do tipo *TErrorNotifierCenter*. Este campo será nossa referência ao *TErrorNotifierCenter*. No *create* do formulário vamos então instanciá-lo chamando o método *Create* da classe *TErrorNotifierCenter*, basta fazer o seguinte:

```
FCentralErros := TErrorNotifierCenter.Create(
  TDefaultNotifier.Create(clWhite, clRed));
```

Veja que no próprio *Create* do *TErrorNotifierCenter* estamos instanciando um *INotifier*. Podemos fazer isso sem nos preocupar com *memory leaks* porque *TErrorNotifierCenter* nesse momento espera uma implementação da interface *INotifier*, veja **Listagem 3**.

Agora é preciso informar ao *framework* que algum controle está inválido. Então no *onChange* do primeiro *Edit* vamos verificar se o conteúdo é vazio, se for, o controle está inválido. Veja como fazemos isso a seguir. Execute a aplicação e veja!

```
if Trim(editNaoVazio.Text) = '' then
  FCentralErros.AddInvalidControl(
    editNaoVazio,'Campo não pode ser vazio')
else
  FCentralErros.RemoveInvalidControl(
    editNaoVazio);
```

Aprimorando o TErrorNotifierCenter

Imagine que é necessário exibir a lista de erros que existem, como um resumo, como podemos fazer isso? O *framework* expõe o método *getErrorList* que retorna essa lista, porém fica a cargo do usuário chamar o método sempre que necessário. Porém podemos melhorar isso através do uso de eventos. Sempre que um controle for removido ou incluído disparamos dois eventos, um avisando que a lista foi atualizada e outro informando detalhes do controle. Vamos implementar a primeira situação.

Quando o evento a ser criado apenas retorna o *Sender*, por exemplo o clique de um botão, é preciso criar um tipo que herde de *TNotifyEvent*. *TNotifyEvent* é o evento básico do Delphi. Para ser caracterizado como evento, um tipo deve ser uma *procedure (Sender: TObject) of object*, e *TNotityEvent* representa esse tipo básico. Adicione então o seguinte tipo na seção *type* da *Unit ErrorNotifier*:

```
TErrorListUpdatedEvent = TNotifyEvent;
```

Agora, como adicionamos esse evento em nossa classe? A resposta é muito simples. Crie uma propriedade para ela:

```
property OnErrorListUpdated:
  TErrorListUpdatedEvent read
    FOnErrorListUpdated write
    FOnErrorListUpdated;
```

Já foi dito que um evento é uma *procedure*. Portanto para dispará-lo basta chamá-lo como uma *procedure* comum. Veja na **Listagem 7** como ficaram os métodos *RemoveInvalidControl* e *AddInvalidControl*.

Quando você efetua um clique duplo em um botão, por exemplo, em seu código fonte é criada uma *procedure*, como a seguir:

```
procedure TForm1.Button1Click(sender: TObject)
```

Então você codifica ali o que é necessário. Agora você sabe como o Delphi sabe que precisa, no momento do clique duplo no botão, chamar o seu código? Muito bem, um botão também possui uma propriedade do tipo do evento, e o IDE do Delphi se encarrega de gerar, no arquivo *.dfm*, a ligação da propriedade que representa o evento com o seu código. Essa ligação fica em um campo privado. Porém, e se não existe código associado a um evento, como não dispará-lo? É preciso então verificar se o campo que armazena a referência do evento está nulo, isso é feito através da função *Assigned*. Veja que antes de chamar o evento, nós verificamos sua situação.

Nota do DevMan

Quando utilizada de forma correta, as *Interfaces* realizam uma operação de *Garbage Collection*, nos livrando assim da necessidade de se dar um *Free* na referência. Por isso não criamos uma variável do tipo *TDefaulNotifier*. Veja mais detalhes sobre o uso correto de interfaces em: *dn.codegear.com/article/30125*

Nota do DevMan

Sempre que você for criar um evento personalizado, crie o tipo adicionando a palavra *Event* a ele. E na propriedade que dará o acesso ao tipo, adicione o prefixo *On*. Dessa forma mantemos o padrão do próprio Delphi, por exemplo o evento *OnKeyUp* de um *TEdit* é um evento do tipo *TKeyEvent*.

Clube101.indb 45 10.11.08 15:56:20

Implementando o evento

Em nosso formulário principal podemos agora implementar o evento criando um método que possua a *mesma assinatura* do evento e então associá-lo à propriedade da classe. Veja como fazemos isso na **Listagem 8**.

Execute a aplicação e você verá que a lista de erros agora é atualizada automaticamente.

Criando seu próprio notificador

Como já foi mencionado, basta criar uma nova classe que implemente a interface *INotifier*. Essa nova classe necessariamente não precisa estar na *Unit ErrorNotifier*, você pode criar uma nova *Unit* e adicionar ao *Uses* a referência à *ErrorNotifier*. Junto aos arquivos deste projeto existe também uma *Unit ImageNotifier.pas*. Ela é uma implementação de notificador que exibe um ícone de erro ao lado de cada controle. Na **Figura 3** o vemos em execução.

Conclusão

De forma simples podemos melhorar o *relacionamento* de nossos clientes com nossos sistemas. São pequenas coisas, que ao fim das contas, fazem diferença. Acredito que o usuário quer algo que o atenda, não apenas funcionalmente – que é o principal – mas também visualmente. Acredito que o visual de um sistema pode ser sim uma questão de desempate, já que ele é o que o usuário vê diariamente. Com um pouco de criatividade, e com o auxílio da orientação a objetos desenvolvemos um controle de notificação de erros reutilizável e *customizável*.

Até a próxima, e sejam criativos! ●

Listagem 7. Métodos chamando o evento

```
procedure TErrorNotifierCenter.AddInvalidControl(pControl: TWinControl;
  AlertMessage: string);
var
 index: integer;
begin
  if ControlInList(pControl) < 0 then
  begin
    index := FInvalidControlsList.Add(TInvalidControl.Create(
      pControl,pControl.Hint, AlertMessage));
    FNotifier.StartNotifying(FInvalidControlsList[index] as
      TInvalidControl);
    if Assigned(FOnErrorListUpdated) then
      OnErrorListUpdated(Self);
  end;
end;
procedure TErrorNotifierCenter.RemoveInvalidControl(pControl: TWinControl);
var
 index: integer;
begin
  index := ControlInList(pControl);
  if index >= 0 then
  begin
    FNotifier.StopNotifying(FInvalidControlsList[index] as
      TInvalidControl);
    FInvalidControlsList.Delete(index);
    if Assigned(FOnErrorListUpdated) then
      OnErrorListUpdated(Self);
  end;
end;
```

Listagem 8. Implementando um evento

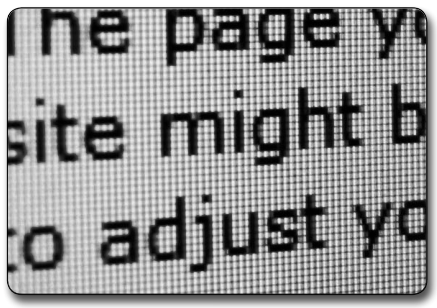
```
procedure TForm1.AtualizaLista(Sender: TObject);
begin
  ListBoxErros.Clear;
  ListBoxErros.Items.Assign(FCentralErros.ErrorList);
  btConfirmar.Enabled := not FCentralErros.HasErrors;
end;
procedure TForm1.FormCreate(Sender: TObject);
begin
  FCentralErros := TErrorNotifierCenter.Create(TDefaultNotifier.Create(
    clWhite, clRed));
  FCentralErros.OnErrorListUpdated := self.AtualizaLista;
end;
```

Dê seu feedback sobre esta edição!

A Clubedelphi tem que ser feita ao seu gosto. Para isso, precisamos saber o que você, leitor, acha da revista!

Dê seu voto sobre este artigo, através do link:

www.devmedia.com.br/clubedelphi/feedback



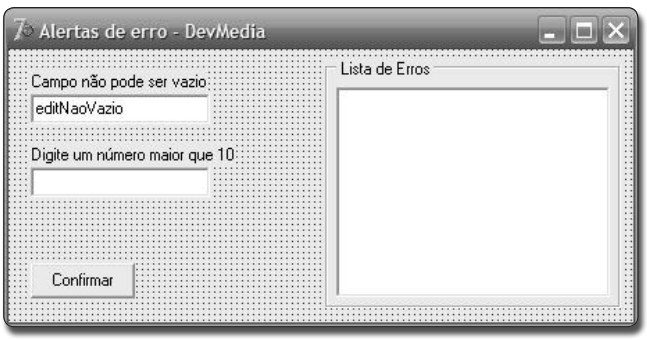


Figura 2. Exemplo de formulário para testes

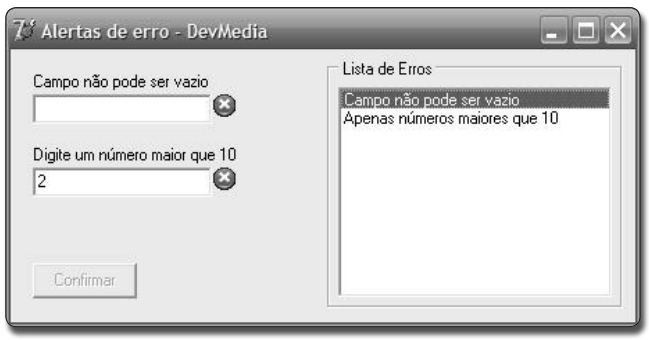


Figura 3. TimageNotifier em execução

Clube101.indb 46 10.11.08 15:56:31