MVC (Model-View-Controller)
Aplique esse poderoso padrão em suas aplicações cliente/server

A grande maioria dos programadores Delphi desenvolve seus sistemas utilizando o modelo Cliente/Servidor, dividindo assim teoricamente seu sistema em duas camadas: interface e banco de dados.

É muito comum vermos no código de um formulário a construção de sentenças SQL e validação de regras de negócio, tudo muito RAD, já que é neste ponto que o Delphi se destaca. É possível construir rapidamente um cadastro e sua manutenção utilizando Delphi, porém um sistema completo não é composto apenas por cadastros. Muita lógica está envolvida.

Além disso existe outro fator que muitos não atentam: a vida do sistema. Por quanto tempo ele vai rodar. Se você quer desenvolver um sistema que tenha uma vida longa tem que pensar no tempo e custo das manutenções que ocorrerão. Nessa hora que surgirão os problemas em se utilizar o RAD sem pensar na arquitetura do sistema. É muito fácil construir uma aplicação arrastando componentes, gerar os eventos no formulário e dentro deles mesmo acessar banco de dados, SQL e etc. O difícil será reutilizar rotinas, regras de negócio e aplicação de testes, que juntos garantem uma facilidade na adequação dos sistemas para as necessidades dos usuários.

Agora você pode estar se perguntando, por onde começo? A primeira coisa a se fazer é pensar que seu software é composto por camadas e não apenas por formulários, relatórios e um banco de dados. Vamos explorar um pouco as camadas que um sistema pode ter.

Camadas

Em um sistema com boa arquitetura encontramos algumas divisões lógicas que visam facilitar a manutenção e adição de novos recursos. Essas camadas podem ser vistas na **Figura 1**.

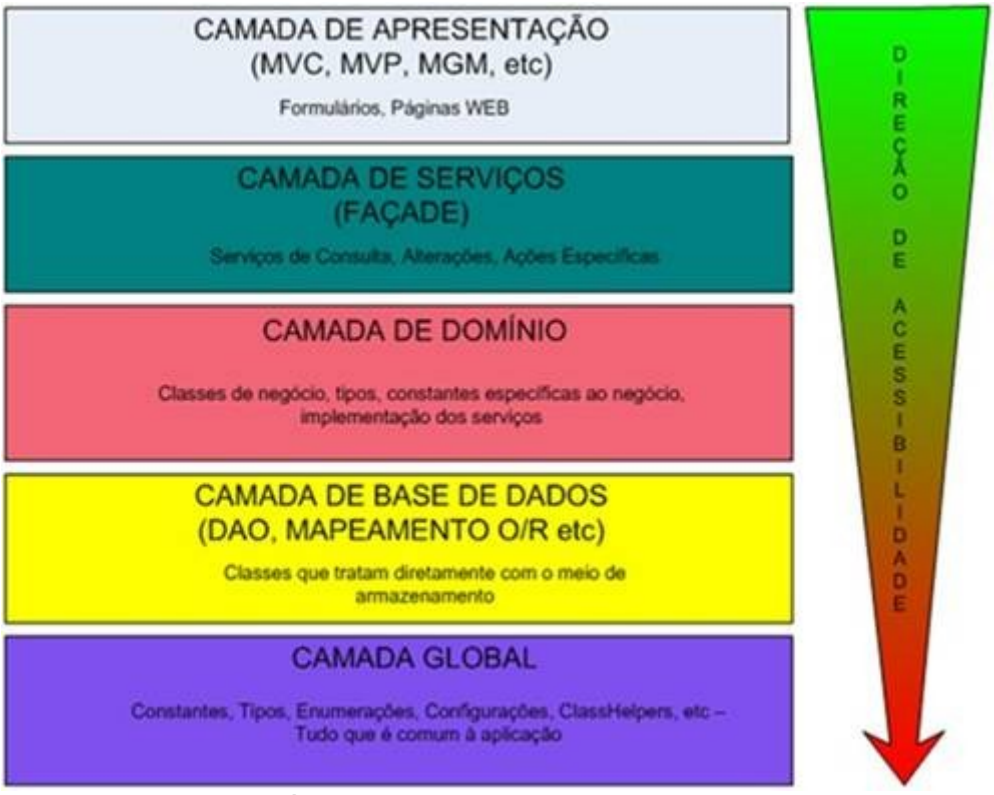

Figura 1. Camadas lógicas de um software

Para tornar a explicação sobre as camadas mais intuitiva quero que imagine seu sistema. Com toda certeza nele você possui um formulário de cadastro. O seu formulário representa a camada de apresentação do seu sistema, isto porque é nela que os dados são apresentados. Se seu sistema é cliente/servidor, a sua camada de domínio, de base dados e até mesmo serviços podem ser representadas pelos Data Modules existentes e seus componentes de acesso ao banco de dados. Por último temos a camada global que pode ser representada pelo seu conjunto de funções e procedimentos públicos.

Continuando com o paralelo a um sistema comum, desenvolvido por um programador Delphi, nos formulários nós fazemos acesso direto aos Data Modules, aos *ClientDataSets*, geramos SQL, tudo sem preocupação alguma, ou seja, nós misturamos o que seriam nossas camadas sendo que o correto é **separar** as camadas.

Quando essas camadas se misturam é dito que existe um acoplamento entre elas. No momento que você insere código SQL em evento *OnClick* de um botão o acoplamento foi realizado. Seu formulário que é sua camada de apresentação, está fazendo acesso direto aos recursos do que seria sua camada de dados. Isso impede a reutilização da lógica contida no formulário. Tudo isso dificulta a manutenção do seu sistema porque você terá SQL espalhado por todo código e não apenas centralizado em uma área.

Outra prática que é muito comum é de se criar os eventos e dentro deles escrever um código tão longo que para entender o que está sendo feito é quase como ler um livro. Consequentemente a rotina ali dentro poderia ser quebrada em classes ou funções.

Quando uma rotina faz mais do que foi projetada a fazer é dito que existe uma baixa coesão nela. A baixa coesão é outro problema para a boa manutenção do seu software. Então, como podemos separar as camadas sem, contudo uni-las? Pensando nessa solução alguns arquitetos de software criaram os **padrões de projeto**. Os padrões são de uma forma simplificada, como uma receita de bolo a ser seguida. Eles expõem o problema e mostram como solucioná-lo.

O problema apresentado neste artigo é como separar a interface do sistema do restante da aplicação. Como torná-la independente. A solução já existe, foi desenvolvida na forma de um padrão de projeto chamado **MVC – Model View Controller**, este é o foco do nosso artigo.

Utilizando o padrão em um exemplo

Nosso primeiro aplicativo de exemplo será simples, porém irá mostrar como iniciar o desenvolvimento utilizando o padrão MVC e fazer uso de interfaces.
Constará apenas de uma tela principal que irá chamar um formulário que permite ao usuário realizar multiplicações. Aqui utilizo o Delphi 2007, contudo, o exemplo pode ser reproduzido também no Delphi 5, 6, 7, 2005 e 2006 sem problema algum e para iniciarmos, inicie um novo projeto Win32.
Para separar a camada de apresentação o MVC cria três partes distintas, o Model, a View e o Controller. Podemos defini-los da seguinte forma:

- Model – É a representação da informação que a aplicação utiliza, que no nosso exemplo é o cálculo de multiplicação e os números que serão calculados.
- View – É a representação gráfica do Model. Por exemplo, um formulário expondo as informações de um cliente. Uma View está sempre ligada a um model e sabe como exibi-lo ao usuário. E para um determinado Model podem existir várias views. Além disso também pode ser composta por várias Views. Aplicando isso ao exemplo, a view é nosso formulário que irá permitir ao usuário digitar o valores e efetuar sua multiplicação.

Controller – É responsável em mapear as ações do usuário em ações do sistema, por exemplo se um botão ou item de menu é clicado é ele que deve definir como a aplicação responderá. No nosso exemplo é o controller que irá fazer com que ao clicar no botão "calcular" o cálculo seja feito.

Colocando a mão na massa
Vamos iniciar criando as partes básicas definidas pelo padrão, o Model, a View e o Controller. Adicione uma unit ao projeto renomeando-a para "MVCInterfaces.pas". Nela adicione as seguintes interfaces, conforme listagem **Listagem 1**.

Listagem 1. MVC
```
type
  TModelChangedEvent = procedure of object;

  IModel = interface
    ['{FDF18F94-0430-4C48-BE64-F4516C4C1011}']
    procedure Initialize;
    function GetOnModelChanged: TModelChangedEvent;
    procedure SetOnModelChanged(value:
      TModelChangedEvent);
    property OnModelChanged: TModelChangedEvent read
      GetOnModelChanged write SetOnModelChanged;
  end;

  IView = interface
     ['{A1F411E9-294D-444D-9D5B-1D6AC9988819}']
    procedure Initialize;
  end;

  IController = interface
    ['{E4BD8853-F6C8-4BD4-B19D-D70B156BD712}']
    procedure Initialize;
  end;
```

Como já foi dito, o Model é nossa regra de negócio, ou seja, nosso cálculo de multiplicação. Existem algumas formas de se implementar o MVC, variando de acordo com o entendimento do desenvolvedor. Em sua forma conceitual um Model possui uma referência à View. Neste exemplo não vejo a necessidade disso, já que o Model aqui é a regra de negócio em si.
Em sistemas mais complexos isso pode até fazer sentido por que o Model poderá encapsular várias regras e será um encapsulamento para elas. O que quero dizer é que a parte chamada Model do MVC nem sempre é diretamente uma classe da nossa camada de domínio. Isso significa que a classe *TAluno* pode conter rotinas e/ou outras classes que têm relação com *TAluno*, como por exemplo, *TCurso* e métodos para matrícula.
Adicione uma nova unit ao projeto, salvando-a como "Multiply.pas" e adicione ao uses da interface a *Unit MVCInterfaces*. Crie uma nova classe conforme **Listagem 2**.

Listagem 2. Model
```
type
  TModelMultiply = class(TInterfacedObject, IModel)
  private
    FValue1: integer;
    FValue2: integer;
    FOnModelChanged: TModelChangedEvent;
    FSolution: integer;
    procedure SetValue1(const Value: integer);
    procedure SetValue2(const Value: integer);
    function GetOnModelChanged: TModelChangedEvent;
    procedure SetOnModelChanged(value:
```

```
    TModelChangedEvent);
  procedure Initialize;
  procedure SetSolution(const Value: integer);
 public
  constructor Create; reintroduce;
  function Multiply: integer;
  property Value1: integer read FValue1 write
    SetValue1;
  property Value2: integer read FValue2 write
    SetValue2;
  property Solution: integer read FSolution write
    SetSolution;
  property OnModelChanged: TModelChangedEvent read
    GetOnModelChanged write SetOnModelChanged;
 end;

implementation

{ TModelMultiply }

constructor TModelMultiply.Create;
begin
 Initialize;
end;

function TModelMultiply.GetOnModelChanged:
 TModelChangedEvent;
begin
 result := FOnModelChanged;
end;

procedure TModelMultiply.Initialize;
begin
 Value1 := 0;
 Value2 := 0;
 Solution := 0;
end;

function TModelMultiply.Multiply: integer;
begin
 result := Value1 * Value2;
 Solution := result;
 if Assigned(OnModelChanged) then
   OnModelChanged;
end;

procedure TModelMultiply.SetOnModelChanged(value:
 TModelChangedEvent);
begin
 FOnModelChanged := value;
end;

procedure TModelMultiply.SetSolution(const Value:
 integer);
begin
 FSolution := Value;
end;
```

```
procedure TModelMultiply.SetValue1(const Value:
  integer);
begin
  FValue1 := Value;
end;

procedure TModelMultiply.SetValue2(const Value:
  integer);
begin
  FValue2 := Value;
end;
```

Veja que a classe *TModelMultiply* contém duas propriedades que armazenarão os números digitados pelo usuário e disponibiliza o método *Multiply* que realiza a multiplicação. Vamos agora criar nossa View correspondente.
A finalidade da View é saber como apresentar ao usuário os dados contidos no Model. Adicione um novo formulário ao projeto e dê a ele o nome de "View1". Salve sua unit como "ViewForm1.pas" e adicione ao uses da seção interface as *Units MVCInterfaces* e *Multiply*. Construa o formulário conforme **Figura 2**.

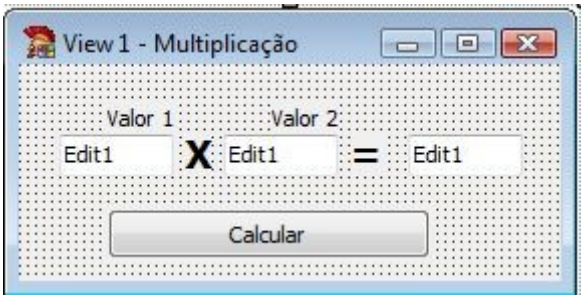

Figura 2. View

Nesse formulário vamos criar um evento que será utilizado pelo Controller. Esse evento representa o clique do botão calcular. Lembre-se que é o controller que transfere as ações do usuário para o Model, portanto não é correto fazer o formulário chamar diretamente nosso cálculo, que é o Model. Vamos apenas adicionar uma referência ao Model, veja a **Listagem 3.**

Listagem 3. Definição da View

```
type
  TCalcEvent = procedure of object;

  TView1 = class(TForm)
    Label1: TLabel;
    Edit1: TEdit;
    Label2: TLabel;
    Edit2: TEdit;
    Label3: TLabel;
    Label4: TLabel;
    Edit3: TEdit;
    btnCalc: TButton;
    procedure btnCalcClick(Sender: TObject);
  private
    FModel: TModelMultiply;
    FDoCalc: TCalcEvent;
    procedure SetModel(const Value: TModelMultiply);
```

```
    { Private declarations }
    procedure SetDoCalc(const Value: TCalcEvent);
  public
    { Public declarations }
    procedure Initialize;
    procedure ModelChanged;
    property Model: TModelMultiply read FModel write SetModel;
    property DoCalc: TCalcEvent read FDoCalc write SetDoCalc;
  end;
```

Observe na **Listagem 4** a implementação dos métodos *Initilialize* e *ModelChanged*. O *Initialize* é responsável por inicializar a View, trazendo para ela o estado inicial do Model. O *ModelChanged* por sua vez responde ao disparo do evento *OnModelChanged* realizado pelo Model e lê suas propriedades para atualizar sua representação. Veja também que no *OnClick* do botão disparamos o evento *DoCalc*.

Listagem 4. Atualizando a View

```
procedure TView1.btnCalcClick(Sender: TObject);
begin
  Model.Value1 := StrToInt(Edit1.Text);
  Model.Value2 := StrToInt(Edit2.Text);
  if Assigned(DoCalc) then
    DoCalc;
end;

procedure TView1.Initialize;
begin
  Edit1.Text := IntToStr(Model.Value1);
  Edit2.Text := IntToStr(Model.Value2);
  Edit3.Text := IntToStr(Model.Solution);
end;

procedure TView1.ModelChanged;
begin
  Initialize;
end;

procedure TView1.SetDoCalc(const Value: TCalcEvent);
begin
  FDoCalc := Value;
end;

procedure TView1.SetModel(const Value: TModelMultiply);
begin
  FModel := Value;
end;
```

O evento *DoCalc* será implementado pelo Controller. Adicione uma nova *Unit* e a salve-a como "FormController.pas", inclua no uses as units *Multiply*, *MVCInterfaces* e ViewForm1. Implemente o controle como na **Listagem 5**.

Listagem 5. Controller

```
type
  TFormController = class(TInterfacedObject, IController)
  private
    FView: TView1;
    FModel: TModelMultiply;
```

```
  public
    constructor Create; reintroduce;
    destructor Destroy; override;
    procedure Initialize;
    procedure Calc;
  end;

implementation

{ TFormController }

procedure TFormController.Calc;
begin
  FModel.Multiply;
end;

constructor TFormController.Create;
begin
  FModel := TModelMultiply.Create;
  FView := TView1.Create(nil);
end;

destructor TFormController.Destroy;
begin
  FView.Free;
  inherited;
end;

procedure TFormController.Initialize;
begin
  FView.Model := FModel;
  FModel.OnModelChanged := FView.ModelChanged;
  FView.DoCalc := Calc;
  FView.Initialize;
  FView.ShowModal;
end;
```

Vamos entender o que o controller está fazendo. De acordo com a forma conceitual o Controller precisa ter uma referência à View e ao Model. Temos isso representado pela variáveis FView e FModel. Temos também uma procedure *Calc* que executa o método *Multiply* do Model. Essa procedure é que irá responder ao evento *DoCalc* da View, assim ao clicar no botão *Calcular* da View, o controller responderá, criando assim o comportamento da View. Tudo isso é implementado no método *Initialize*. É nele que estamos atando cada parte que compõe a tríade MVC.

Executando

No formulário principal da aplicação adicione uma referência à unit *FormController*. Adicione também um botão, e no seu evento *OnClick* vamos chamar o controller. Veja **listagem 6** o código e na **figura 3** o resultado.

Listagem 6. Executando o controller

```
procedure TMainF.Button1Click(Sender: TObject);
var
  lController: TFormController;
begin
  lController := TFormController.Create;
  lController.Initialize;
end;
```

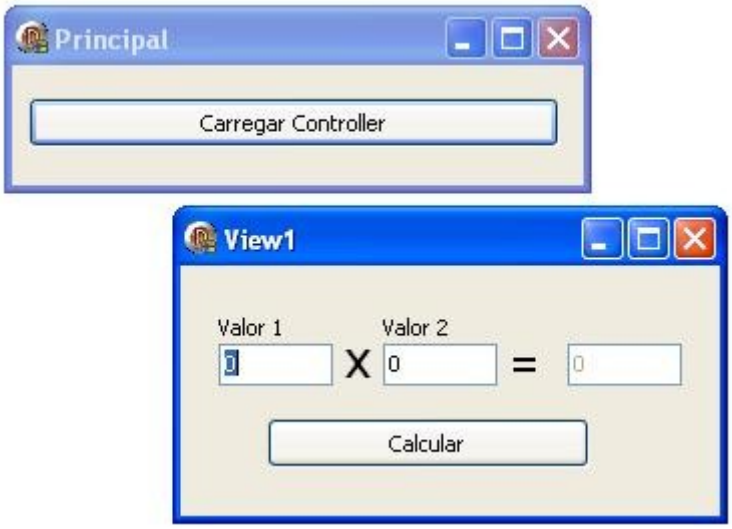

Figura 3. MVC em ação

Complicou?
A primeira vista pode parecer complicado, mas não é. O segredo é ter em mente o que cada parte do MVC deve fazer e como se dá a comunicação entre elas. Na **Figura 4** temos uma ilustração dessa comunicação.

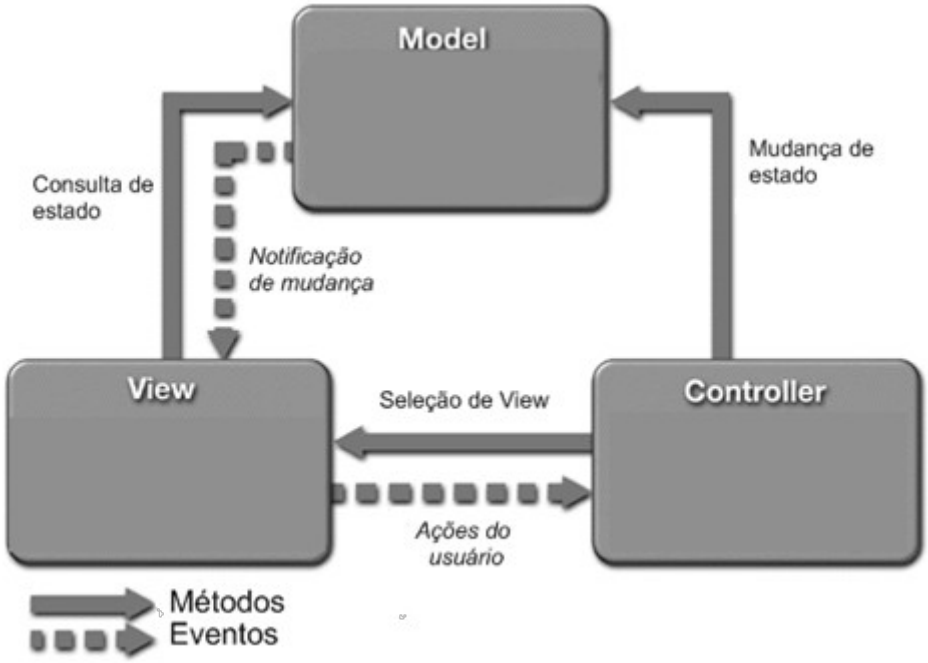

Figura 4. A comunicação dentro do MVC

Basicamente o Model aponta para a View o que permite enviar a ela notificações de mudança, no nosso exemplo isso está implementado pela definição do evento *OnModelChanged*. O Model não deve saber nada sobre a View, portanto esta referência é feita através de uma superclasse da View que expõe meios de ser notificada. Veja que nossa View (formulário) implementa o evento anteriormente definido pelo Model e o utiliza para receber a notificação de atualização do mesmo.

Por outro lado, a View está diretamente ligada a um Model e sabe exatamente os detalhes do Model, de tal forma que seus atributos possam ser acessados, isso pode ser visto no acesso que a View faz às propriedades *Value1* e *Value2* do Model. Além disso possui também uma referência ao Controller que deve estar ligada a uma classe base do Controller. Isso, para permitir que um controller possa ser trocado. Este por sua vez possui referências diretas ao Model e à View porque é ele que define o comportamento geral. Observe a existência de dois campos privados, respectivamente dos tipos View e Model.

Um exemplo mais real
Ainda ficou complicado não é? Programador Delphi não se dá por satisfeito até ver sua comum pergunta respondida: "Mas o que fazer com minhas queries e tabelas?". Para provar que não é tão complicado retirar do seu formulário todo SQL que está lá vamos transformar um aplicativo Cliente/Servidor comum, em um aplicativo Cliente/Servidor com a interface separada das regras de negócio e do banco de dados.
Neste exemplo não seremos puristas quanto ao uso da orientação a objetos, mas vamos utilizar o que há de melhor no Delphi, RAD e a orientação a objetos, para que você, programador, possa ir se acostumando e ver que vale a pena aplicar tudo isso em seus projetos.

O tradicional
Vamos criar uma agenda de telefone. Primeiramente vamos criá-la de forma tradicional e posteriormente vamos aproveitar o que já tem e adaptar um MVC. Crie um banco de dados Firebird contendo a seguinte tabela, vista na **Figura 5**.

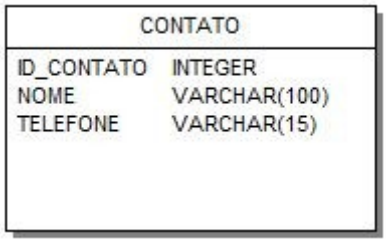

Figura 5. Tabela CONTATO

Inicie agora uma nova aplicação Win32, adicione um Data Module e crie uma conexão DBX com o banco de dados criado. Adicione ao Data Module um *TSQLQuery* que acesse a tabela CONTATO e ligue a esse *TSQLQuery* um *ClientDataSet*. Vamos utilizar esse *ClientDataSet* em nossa aplicação. Configure o formulário principal conforme **Figura 6**. Adicione um segundo formulário e o deixe como na **Figura 7**.

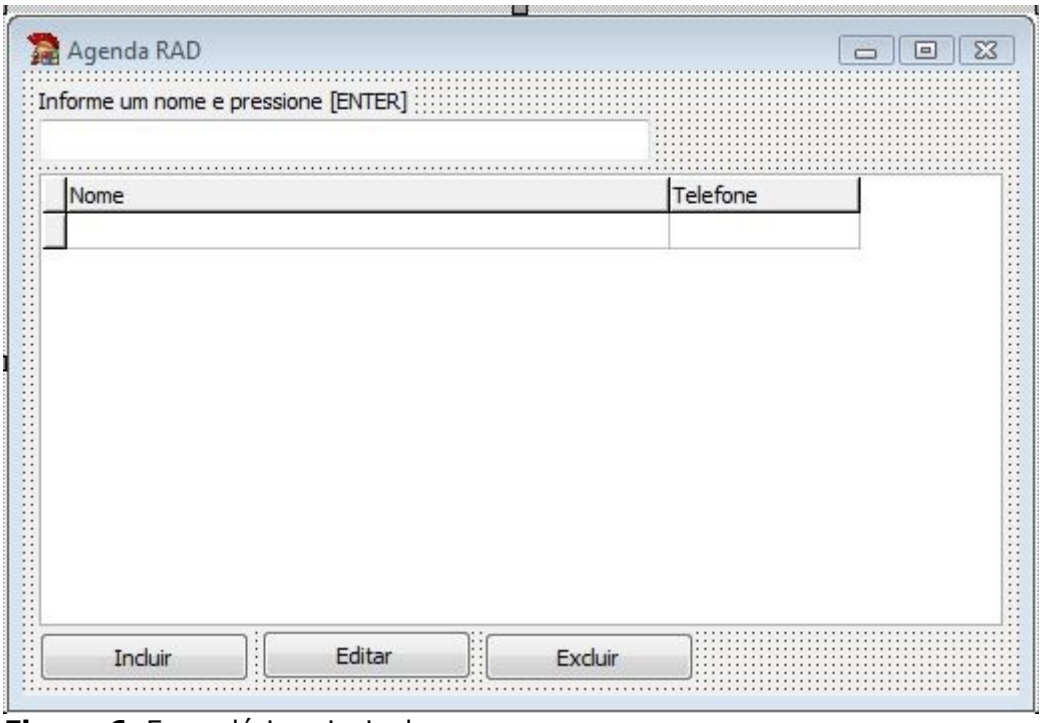

Figura 6. Formulário principal

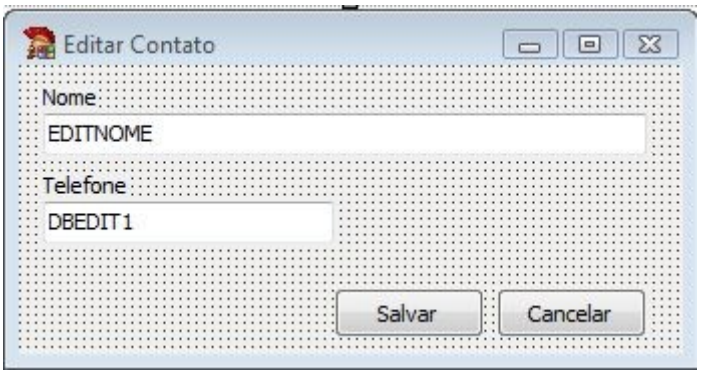

Figura 7. Formulário de edição

Não vou entrar em detalhes de como ligar os controles *Dataware* ao *ClientDataset*, porque como disse, esta primeira parte do exemplo você usa no seu dia-a-dia. Observe a **Listagem 7,** ela contém a lógica do sistema, SQL e validação. Não existiria nada de errado com elas desde que não estivessem dentro dos formulários. Essas listagens mostram o que a grande maioria dos desenvolvedores Delphi faz de forma natural.

Listagem 7. Evento onKeyUp do Edit de busca

```
procedure TPrincF.editBuscaKeyUp(Sender: TObject;
  var Key: Word; Shift: TShiftState);
begin
  if (Key = VK_RETURN) then
  begin
    if (length(Trim(EditBusca.Text)) >= 3) then
    begin
      dm.cdsContatos.Close;
      dm.qryContatos.SQL.Clear;
      dm.qryContatos.SQL.Add('SELECT * FROM CONTATO');
      dm.qryContatos.SQL.Add(
```

```
        'WHERE NOME STARTING WITH ' +
        QuotedStr(editBusca.Text));
      dm.qryContatos.SQL.Add('ORDER BY NOME');
      dm.cdsContatos.Open;
    end
    else
    begin
      if Trim(EditBusca.Text) = '' then
      begin
        dm.cdsContatos.Close;
        dm.qryContatos.SQL.Clear;
        dm.qryContatos.SQL.Add(
          'SELECT * FROM CONTATO');
        dm.qryContatos.SQL.Add('ORDER BY NOME');
        dm.cdsContatos.Open;
      end
      else
      begin
        ShowMessage('Para realizar a busca você deve
          informar no mínimo 3 caracteres');
        editBusca.SetFocus;
      end;
    end;
  end;
end;

procedure TPrincF.btIncluirClick(Sender: TObject);
begin
  dm.cdsContatos.Insert;
  EditarContatoF := TEditarContatoF.Create(nil);
  EditarContatoF.ShowModal;
  FreeAndNil(EditarContatoF);
end;

procedure TPrincF.btEditarClick(Sender: TObject);
begin
  dm.cdsContatos.Edit;
  EditarContatoF := TEditarContatoF.Create(nil);
  EditarContatoF.ShowModal;
  FreeAndNil(EditarContatoF);
end;

procedure TPrincF.btExcluirClick(Sender: TObject);
begin
  if Application.MessageBox('Confirma exclusão de
    contato ?','Atenção',MB_YESNO + MB_ICONQUESTION +
    MB_DEFBUTTON2)=ID_YES then
  begin
    dm.cdsContatos.Delete;
    dm.cdsContatos.ApplyUpdates(-1);
  end;
end;

procedure TEditarContatoF.btSalvarClick(Sender:
  TObject);
begin
  if trim(DBEdit1.Text) = '' then
```

```
  begin
    ShowMessage('Nome de contato inválido');
    exit;
  end;
  dm.cdsContatos.Post;
  dm.cdsContatos.ApplyUpdates(-1);
  Close;
end;

procedure TEditarContatoF.btCancelarClick(Sender:
  TObject);
begin
  dm.cdsContatos.CancelUpdates;
  Close;
end;
```

Aplicando o MVC sem perder o RAD

Como já mencionei anteriormente não serei um "purista", quero aproveitar o que o Delphi oferece, contudo, aplicando boas prática OO. Para relembrar, o MVC é "Model-View-Controller". Aproveitando a estrutura existente a View são os nossos formulários, o Model são as rotinas de Inclusão, Edição, Exclusão e Localização que podem ser encapsuladas em uma classe que as represente, por exemplo *ManterContato*. Os dados dos contatos continuarão sendo acessados pela estrutura *SQLQuery, DatasetProvider* e *ClientDataSet*. O Controller se encarregará de chamar os métodos do Model de acordo com as ações do usuário em cima da View.

Semelhantemente ao primeiro exemplo construído vamos adicionar uma *Unit* ao projeto e chamá-la de "MVCInterfaces.pas". Adicione a ela o código da **Listagem 8**.

Listagem 8. Interfaces básicas do MVC

```
unit MVCInterfaces;

interface

type
  IModel = interface
    ['{FDF18F94-0430-4C48-BE64-F4516C4C1011}']
    procedure Initialize;
  end;
  IView = interface
    ['{A1F411E9-294D-444D-9D5B-1D6AC9988819}']
    procedure Initialize;
  end;
  IController = interface
    ['{E4BD8853-F6C8-4BD4-B19D-D70B156BD712}']
    procedure Initialize;
  end;
implementation
end.
```

Agora vamos utilizar essas interfaces para implementar a lógica do nosso sistema de Agenda. Para começar vamos criar o Model. Adicione uma nova *Unit*, salve-a como "ModelAgenda.pas". A ela adicione uma referência na seção uses à unit *MVCInterface* e ao Data Module da aplicação, lembre-se que vamos utilizar a estrutura já criada. Crie a classe *TManterContato* conforme visto na **Listagem 9**.

Listagem 9. Manter contatos

```
uses MVCInterfaces, dmU, SysUtils;

type
   IManterContato = interface(IModel)
   ['{01AFE56E-1787-4B79-9B48-A018B2859B0C}']
   procedure LocalizarPorNome(Nome: string);
   procedure Inserir;
   procedure Excluir;
   procedure Editar;
   procedure Salvar;
   procedure Cancelar;
  end;

  TManterContato = class(TInterfacedObject,
    IManterContato)
  public
    procedure LocalizarPorNome(Nome: string);
    procedure Inserir;
    procedure Excluir;
    procedure Editar;
    procedure Salvar;
    procedure Cancelar;
    procedure Initialize;
  end;

  TQuantidadeCaracteresInvalidaEx = class(Exception);
  TDadosInvalidosEx = class(Exception);

implementation

{ TManterContato }

procedure TManterContato.Cancelar;
begin
  dm.cdsContatos.CancelUpdates;
end;

procedure TManterContato.Editar;
begin
  dm.cdsContatos.Edit;
end;

procedure TManterContato.Excluir;
begin
  dm.cdsContatos.Delete;
  dm.cdsContatos.ApplyUpdates(-1);
end;

procedure TManterContato.Initialize;
begin
  dm.cdsContatos.Close;
  dm.qryContatos.SQL.Clear;
  dm.qryContatos.SQL.Add(
    'SELECT * FROM CONTATO ORDER BY NOME');
  dm.cdsContatos.Open;
end;
```

```
procedure TManterContato.Inserir;
begin
  dm.cdsContatos.Insert;
end;

procedure TManterContato.LocalizarPorNome(Nome:
  string);
begin
  if (length(Trim(Nome)) >= 3) then
  begin
    dm.cdsContatos.Close;
    dm.qryContatos.SQL.Clear;
    dm.qryContatos.SQL.Add('SELECT * FROM CONTATO');
    dm.qryContatos.SQL.Add('WHERE NOME STARTING WITH '
      + QuotedStr(Nome));
    dm.qryContatos.SQL.Add('ORDER BY NOME');
    dm.cdsContatos.Open;
  end
  else
  begin
    if Trim(Nome) = '' then
    begin
      Self.Initialize;
    end
    else
    begin
      raise TQuantidadeCaracteresInvalidaEx.Create(
        'Para realizar a busca você deve informar no
        mínimo 3 caracteres');
    end;
  end;
end;

procedure TManterContato.Salvar;
begin
  if Trim(dm.cdsContatosNOME.AsString) = '' then
  begin
    raise TDadosInvalidosEx.Create('Nome de
      contato inválido');
  end;
  dm.cdsContatos.Post;
  dm.cdsContatos.ApplyUpdates(-1);
end;
```

Veja que nossa classe *TManterContato*, que é nosso Model, centraliza em si a manipulação de SQL, o CRUD e sua validação. Também criamos duas exceções que são utilizadas caso algo de errado aconteça. Então você pode se perguntar: Porque não exibimos uma mensagem ao usuário ao invés de levantar a exceção? A razão é bem simples, a função de interação com o usuário é da View e não do Model. O Model nem sabe como seus dados serão exibidos, por isso ele apenas notifica que algo está errado. Isso garante a reutilização desse Model.

Continuando, temos o Controller e é ele que irá chamar os métodos do Model. Na **Listagem 10** temos a implementação completa do Controller, adicione um nova *Unit* e adicione o código.

Listagem 10. Controller

```
interface
```

```
uses MVCInterfaces, ModelAgenda, SysUtils;
type
  IAgendaController = interface(IController)
    ['{E52C60DD-181B-4272-A3DF-72610E9BA2F7}']

    procedure LocalizarPorNome(Nome: string);
    procedure Inserir;
    procedure Excluir;
    procedure Editar;
    procedure Salvar;
    procedure Cancelar;
    procedure Initialize;
    procedure SetModel(const Value: IManterContato);
    procedure SetView(const Value: IView);
    function GetView: IView;
    function GetModel: IManterContato;
    property Model:IManterContato read GetModel
      write SetModel;
    property View: IView read GetView  write SetView;
  end;
  TAgendaController =  class(TInterfacedObject,
    IAgendaController)
  private
    FModel: IManterContato;
    FView: IView;
    procedure SetModel(const Value: IManterContato);
    procedure SetView(const Value: IView);
    function GetView: IView;
    function GetModel: IManterContato;
  public
    procedure LocalizarPorNome(Nome: string);
    procedure Inserir;
    procedure Excluir;
    procedure Editar;
    procedure Salvar;
    procedure Cancelar;
    procedure Initialize;
    property Model:IManterContato read GetModel
      write SetModel;
    property View: IView read GetView write SetView;
  end;

implementation

{ TAgendaController }

procedure TAgendaController.Cancelar;
begin
  FModel.Cancelar;
end;

procedure TAgendaController.Editar;
begin
  FModel.Editar;
end;

procedure TAgendaController.Excluir;
```

```
begin
  FModel.Excluir;
end;

function TAgendaController.GetModel: IManterContato;
begin
  result:= FModel;
end;

function TAgendaController.GetView: IView;
begin
  result := FView;
end;

procedure TAgendaController.Initialize;
begin
  FModel.Initialize;
end;

procedure TAgendaController.Inserir;
begin
  FModel.Inserir;
end;

procedure TAgendaController.LocalizarPorNome(Nome:
  string);
begin
  FModel.LocalizarPorNome(Nome);
end;

procedure TAgendaController.Salvar;
begin
  FModel.Salvar;
end;

procedure TAgendaController.SetModel(const Value:
  IManterContato);
begin
  FModel := Value;
end;

procedure TAgendaController.SetView(const Value:
  IView);
begin
  FView := Value;
end;
```

Vamos agora ajustar nosso formulário principal para agir como uma View. Adicione na seção *Uses* da seção *Interface* as units *MVCInterfaces*, *ControllerAgenda*, e *ModelAgenda*. Na declaração do formulário indique que ele implementa *IView*, conforme abaixo:

```
TPrincF = class(TForm, IView)
```

E por implementar *IView*, ele precisa ter referências ao Model, ao Controller e deve implementar o método *Initialize* de *IView*. Veja isso na **Listagem 11**.

Listagem 11. Implementando a View

```
type
  TPrincF = class(TForm, IView)
  private
    FController: IAgendaController;
    { Private declarations }
    procedure Initialize;
    procedure SetController(const Value:
      IAgendaController);
  public
    { Public declarations }
    property Controller: IAgendaController read
      FController write SetController;
  end;

procedure TPrincF.Initialize;
var
 lModel: TManterContato;
begin
  Controller := TAgendaController.Create;
  lModel:= TManterContato.Create;
  Controller.Model := lModel;
  Controller.View := Self;
  Controller.Initialize;
end;
```

Com isso temos nossa View pronta. Agora vamos remover as chamadas aos componentes de acesso e adicionar as chamadas ao controller. A primeira a ser removida é a que consta no evento *OnShow* do formulário principal. Substitua o código a seguir:

```
Self.Initialize;
```

Veja que o *Initialize* da View é chamado e este por sua vez realiza a ligação entre os componentes do MVC e inicia o Controller. Depois, vamos substituir o evento *OnKeyUp* do *Edit* de busca pelo código da **Listagem 12**.

Listagem 12. Utilizando o controller para realizar a busca

```
procedure TPrincF.editBuscaKeyUp(Sender: TObject;
  var Key: Word; Shift: TShiftState);
begin
  if (Key = VK_RETURN) then
  begin
    try
      Controller.LocalizarPorNome(editBusca.Text);
    except
    on e: TQuantidadeCaracteresInvalidaEx do
      begin
        ShowMessage(e.Message);
        editBusca.SetFocus;
      end;
    end;
  end;
end;
```

Observe o código. Veja que a chamada ao controller está dentro de um bloco *try..except*. Lembre-se que o Controller chama o método *LocalizarPorNome* de

TManterContato, que é nosso Model, portanto a regra para validar se a consulta pode ser feita ou não está contida no Model e não no formulário. Vamos substituir agora o código do botão excluir para o código da **Listagem 13**.

Listagem 13. Utilizando o controller para realizar uma exclusão

```
procedure TPrincF.btExcluirClick(Sender: TObject);
begin
  if Application.MessageBox(
    'Confirma exclusão de contato ?','Atenção',
    MB_YESNO + MB_ICONQUESTION +
    MB_DEFBUTTON2)= ID_YES then
  begin
    Controller.Excluir;
  end;
end;
```

Para utilizar agora o Controller para fazer a inserção e edição de um contato temos que fazer uma alteração também no formulário de edição (**Figura 7**). A primeira coisa a se fazer é adicionar no *uses* da seção *Interface* uma referência às *Units ControllerAgenda* e *ModelAgenda*, depois vamos criar uma propriedade publica do tipo *IAgendaController*, conforme a seguir**:**

```
...
public
  { Public declarations }
  property Controller: IAgendaController
    read FController write SetController;
...
```

Aí então podemos substituir os códigos do botão *Salvar* e *Cancelar* pelos da **Listagem 14**.

Listagem 14. Utilizando Controller para Salvar e Cancelar

```
procedure TEditarContatoF.btCancelarClick(Sender: TObject);
begin
  Controller.Cancelar;
  Close;
end;

procedure TEditarContatoF.btSalvarClick(
  Sender: TObject);
begin
  try
    Controller.Salvar;
    Close;
  except
  on e: TDadosInvalidosEx do
    begin
      ShowMessage('Nome de contato inválido');
      exit;
    end;
  end;
end;
```

Com o nosso formulário de edição pronto, podemos agora terminar a alteração no formulário principal. Vamos remover os códigos dos botões *Inserir* e *Editar* pelo código da **Listagem 15**.

Listagem 15. Passando o controller para o formulário de edição

```
procedure TPrincF.btIncluirClick(Sender: TObject);
begin
  Controller.Inserir;
  EditarContatoF := TEditarContatoF.Create(nil);
  EditarContatoF.Controller := Self.Controller;
  EditarContatoF.ShowModal;
  FreeAndNil(EditarContatoF);
end;

procedure TPrincF.btEditarClick(Sender: TObject);
begin
  Controller.Editar;
  EditarContatoF := TEditarContatoF.Create(nil);
  EditarContatoF.Controller := Self.Controller;
  EditarContatoF.ShowModal;
  FreeAndNil(EditarContatoF);
end;
```

Antes de executar a aplicação, dê uma olhada no código do formulário principal por completo. Veja que não temos no código nenhuma chamada a *ClientDataSet* ou instrução SQL. Execute agora e você verá tudo funcionando. Insira um *breakpoint* por exemplo na chamada ao *Controller*.*LocalizarPorNome* e vá debugando. Você irá perceber como se dá a passagem de chamadas de um item a outro do MVC. Na **Figura 8** temos uma simplificação desse processo.

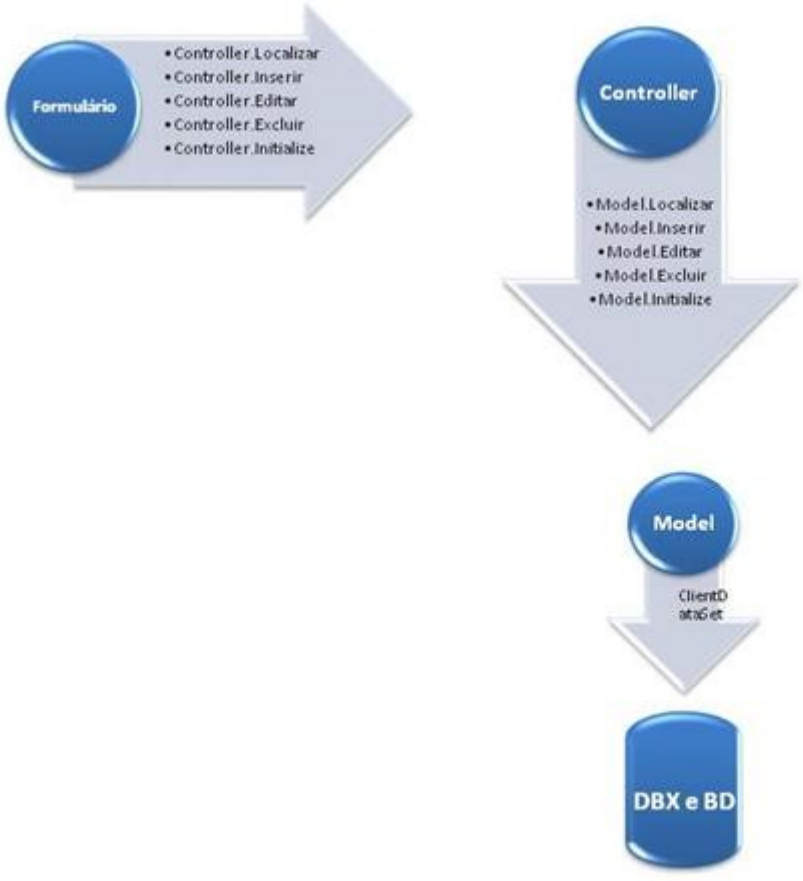

Figura 8. Comunicação do MVC

Conclusão

Neste artigo vimos que podemos utilizar o RAD e utilizar ao mesmo tempo boas práticas, que neste artigo é a separação da interface (formulários) com o restante do sistema. Utilizamos uma adaptação do modelo MVC para a realidade dos programadores Delphi, conseguimos retirar dos formulários as chamadas SQL e chamadas diretas a *ClientDataSet* passando isso ao Controller que por sua vez passa ao Model.

Dessa forma obtivemos um formulário que não possui regra de negócio e com isso, pode ser substituído por outro formulário qualquer, desde que implemente a interface *IView*. As regras do negócio ficaram encapsuladas em classes e podem ser reutilizadas em outras variações da agenda, por exemplo uma agenda pra web.

Ao utilizar os *ClientDataSets* como camada de dados ganhamos em agilidade na montagem do formulário e no acesso ao banco de dados, isto porque nos prendemos em melhorar um sistema já pronto. Com a separação do formulário podemos agora aplicar testes unitários ao controller e model, coisa que é impossível de se fazer programando do "jeitão" comum.

Sem muito esforço acrescentamos qualidade ao nosso software e ganhamos conhecimento, lembrando que aplicar essa separação de formulários é apenas um passo para se conseguir uma separação ideal das camadas de um sistema, porém já temos um começo utilizando o RAD e OO.