

# Mini-Curso de Delphi e UML – Parte III



**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

**D**ando continuidade ao curso, veremos neste artigo como gerar o código Delphi a partir do ModelMaker para construirmos nossa aplicação.

## Codificando e gerando classes

Inicie um novo projeto no Delphi e salve-o. Inclua uma nova unit e salve-a como "BusinessModel.pas". Acesse o menu *ModelMaker>Run ModelMaker Pascal Edition*. Agora no ModelMaker abra o projeto do curso e na guia *Modules* clique com o botão direito sobre *Classes not assigned to modules* e escolha *Add new Module*.

Agora é preciso escolher em qual unit as classes serão geradas e quais classes serão geradas nessa unit. É possível determinar que cada classe tenha sua própria unit ou gerar todas as classes em uma só. Veja na **Figura 1**, como deve ficar o módulo.

---

**Nota:** Eu recomendo a geração de uma unit por classe, isso facilita na hora da manutenção, para dividir as tarefas para a equipe. Como nosso projeto é simples, decidi criar as classes em uma única unit, a *BusinessModel*.

---

Vamos implementar os métodos *Get/Set* das propriedades das classes *TRevista* e *TArtigo*, e faremos isso dentro do próprio ModelMaker. Na guia *Classes*, clique sobre *TRevista*, observe que abaixo são exibidas as propriedades, métodos e campos da classe, de acordo com o filtro de visualização. Clique sobre o método *GetEmail*, acesse a guia *Implementation* e digite o seguinte código:

```
Result := _Email.Value;
```

Implemente também o método *SetEmail* com o seguinte código:

```
_Email.Value := Value;
```

Faça o mesmo para os outros *Gets/Sets* da classe *TRevista*. Ao terminar vamos gerar a classe *TRevista*. Primeiramente devemos habilitar a geração de código, que por padrão é bloqueada. Observe que temos na barra de atalhos dois botões que são representados por um cadeado aberto e outro fechado (**Figura 2**).

Clique sobre o cadeado aberto, para a geração de código ficar habilitada. Na guia *Modules* clique sobre o botão *Generate* (▶), caso seja solicitada alguma confirmação, por hora, confirme. Acesse o Delphi, sem fechar o ModelMaker e veja que ambas as classes *TRevista* e *TArtigo* estão implementadas.

Você vai observar que os métodos de *TArtigo* não estão completos. Para mostrar o nível de integração do Delphi e do ModelMaker vamos fazer o inverso, vamos implementar os métodos de *TArtigo* no Delphi e pedir para que sejam atualizados no ModelMaker.

Implemente os *Gets/Sets* como feito em *TRevista*, a diferença estará para o *Get/Set* da propriedade *TArtigo.Revista*, que deverá ser implementada conforme o seguinte código:

```
function TArtigo.GetRevista: TRevista;
begin
    Result := _Revista.Value as TRevista;
end;

procedure TArtigo.SetRevista(Value: TRevista);
begin
    _Revista.Value := Value;
end;
```

É preciso também adicionar à cláusula *uses* a unit *InstantPersistence* e criar a seção *Initialization* para registrar as classes, que é um requisito do *InstantObjects*, conforme o seguinte código:

```
initialization
    InstantRegisterClasses([TRevista, TArtigo]);
```

Feito isso, acesse o menu *ModelMaker>Refresh in Model*, vá para o ModelMaker e veja as propriedades de *TArtigo* implementadas!

**Nota:** sempre que fizer qualquer alteração no código do Delphi, utilize o menu anterior para atualizar o modelo do ModelMaker.

### Mapeamento das classes

Com as classes prontas, é preciso agora mapeá-las para um banco de dados relacional. Podemos fazer esse mapeamento através de *tags*, colocadas entre comentários, que o *InstantObjects* poderá interpretar. Veja na **Listagem 1** como fica o mapeamento para as classes.

Listagem 1. Mapeamento de *TRevista* e *TArtigo*

```
TRevista = class(TInstantObject)
    {IOMETADATA stored;
    Nome: String(50);
    Email: String(50);
    Link: string(50); }
private
    ...
end;

TArtigo = class(TInstantObject)
    {IOMETADATA stored;
    Nome: String(70);
    NumeroEdicao: Integer;
    PalavrasChave: String(70);
    Resumo: Memo;
    Revista: Reference(TRevista); }
private
    ...
end;
```

O *InstantObjects* oferece um assistente bem completo, que gera classes, *tags* de mapeamento e o banco de dados. Acesse o menu *View>InstantObjects Model Explorer*, para abrir a janela onde faremos o mapeamento.

Nessa janela, clique sobre o botão *Select Units* (📁), e localize a unit que contém as classes a serem mapeadas. Selecione então a unit *BusinessModel*, envie para a *Model Units* e confirme. O *Model Explorer* então exibirá as classes contidas na unit (**Figura 3**).



Figura 2. Habilitando a geração de código

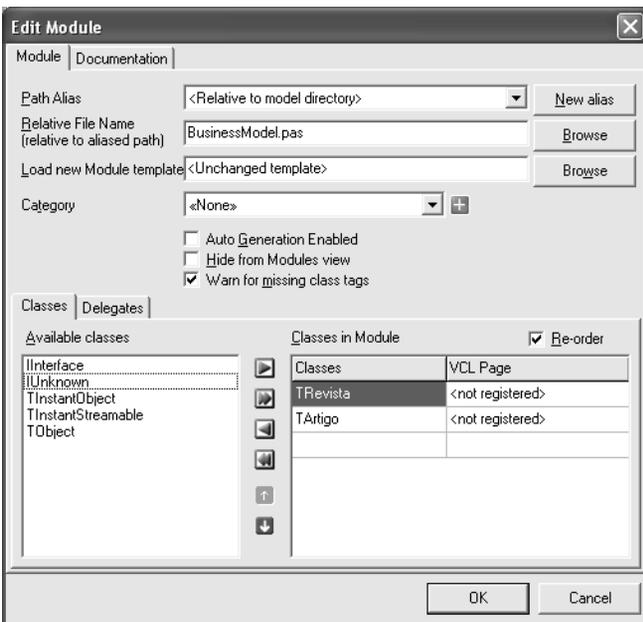


Figura 1. Escolhendo a unit onde serão geradas as classes

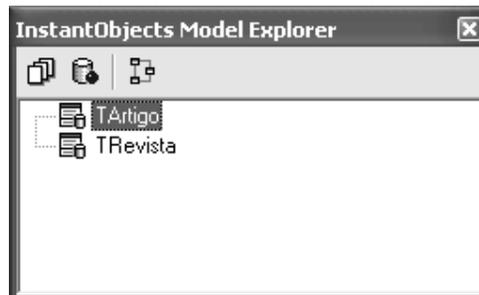


Figura 3. Model Explorer do InstantObjects

Clicando com o botão direito do mouse sobre as classes é aberto um menu por onde podemos editar nossa classe e configurar como suas propriedades serão exibidas na tela.

## Criando o banco de dados

No Model Explorer clique no botão *Build Database* ( ). Na janela *Database Builder* clique com o botão direito do mouse, escolha *New>IBX Connection* e mude seu nome para “conexao”. É preciso configurar a nova conexão, clicando com o botão direito do mouse sobre ela e escolhendo *Edit*. Configure a conexão conforme a **Figura 4**, escolhendo um local para o seu banco de dados.

Clique em OK e no botão *Build*. Na janela de criação clique sobre *Show Build Sequence*, que mostrará a geração do banco. Finalmente, clique sobre *Build Database*, o InstantObjects notifica que se uma base de dados já existir ela perderá seus dados, e caso não seja essa a necessidade, utilize a opção *Evolue*. Como estamos criando um banco vazio, apenas confirme e o banco será criado.

**Nota:** Se você estiver usando o Firebird e receber uma mensagem do tipo: *unavailable database*, normalmente basta adicionar o IP ou apenas *localhost* antes do caminho do banco na janela de configurações.

Adicione um DataModule ao projeto. Adicione um *IBDatabase* (paleta *InterBase*) e faça uma conexão com o banco que criamos anteriormente. Insira um *InstantIBXConnector* (*InstantObjects*) e ligue sua propriedade *Connection* ao *IBDatabase1*. Altere suas propriedades *IDDataType* para *dtString*, *IdSize* para “10”, *IsDefault* para *True* e *LoginPrompt* para *False*.

Adicione também um *InstantSelector* e altere seu nome para “ArtigoSelector”. O *InstantSelector* é responsável por realizar a seleção de objetos de uma determinada classe, essa que deve ser indicada na propriedade *ObjectClassName*, portanto configure-a para *TArtigo*.

Adicione um *DataSource* e ligue-o ao *ArtigoSelector*. Adicione mais um *InstantSelector*, dessa vez para a classe *TRevista*. Altere seu nome para “RevistaSelector” e *ObjectClassName* para *TRevista*. Adicione e ligue um *DataSource* ao *RevistaSelector*. Nesse momento, seu DataModule deve estar semelhante ao da **Figura 5**.

## Um pouco de Arquitetura

Muito se fala em “aplicações em camadas” e quando ouvimos isso logo pensamos em sistemas 3 camadas. Uma outra forma de desenvolvimento em camadas é organizar nossas classes logicamente, de acordo com o papel que exercem (**Figura 6**).

Os nomes e finalidades de cada camada podem variar de acordo com o entendimento, porém a idéia principal deve ser mantida: organizar as classes de acordo com sua função e sempre que possível manter a ordem de acessibilidade.

As camadas superiores fazem referência às camadas



Figura 4. Configurando a conexão com o banco de dados

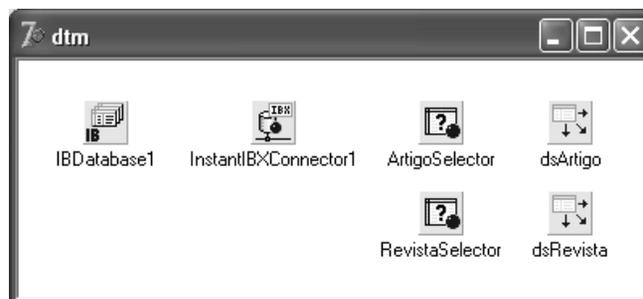


Figura 5. DataModule da aplicação

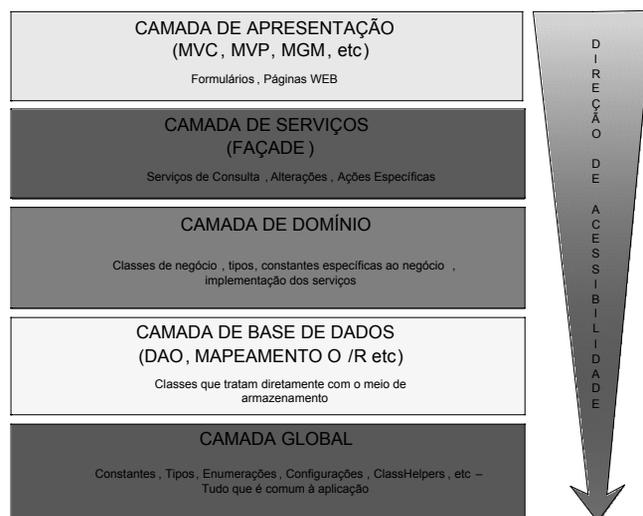


Figura 6. Arquitetura de camadas de sistemas

inferiores, levando sempre em conta que cada uma deve fazer o que foi projetada para fazer. No nosso exemplo, a camada de base de dados já foi implementada pelo *InstantObjects*, que é responsável em saber como acessar e lidar com os dados que serão mantidos em armazenamento físico.

Na camada de domínio, temos as classes *TRevista* e *TArtigo* que representam o negócio, porém estão muito pobres. Segundo Martin Fowler o design pobre em um domínio está no fato dele apenas conter as entidades que devem ser persistidas.

Por outro lado, um design rico de uma camada de negócios conduz a implementações de outras classes que auxiliam nas regras de negócio, como validações, consultas, listagem que posteriormente podem ser utilizadas pela camada de serviços. Vamos enriquecer então nosso domínio desenvolvendo uma classe de consulta, conforme solicitado no caso.

### Classe de consulta

No ModelMaker vamos adicionar um novo diagrama de classes e adicionar uma classe abstrata, chamada “TConsulta” e nela adicionar um método abstrato, chamado “Consultar”, que recebe os seguintes parâmetros: “TipoConsulta: TTipo, Texto: string, Filtro: TRevista = nil”. Você pode se perguntar agora, por que consulta é uma classe abstrata?

Existe uma boa prática, um princípio na programação OO dizendo que devemos programar para interfaces para obter modelos mais flexíveis. Vamos criar a classe de consulta que implementará o método *Consultar* de forma específica ao *InstantObjects*. Adicione uma classe que herda de *TConsulta* e sobreponha o método *Consultar* original (Figura 7).

Adicione as classes de consulta ao módulo *BusinessModel.pas* e gere o código, conforme fizemos anteriormente. Pelo Delphi vamos criar na unit *BusinessModel* um *Enum*, o *TTipo* que será utilizado na *Consulta*, conforme o seguinte código:

```
TTipo = (NomeArtigo, PalavraChave, Resumo);
```

Implemente o método *Consulta* conforme a Listagem 2.

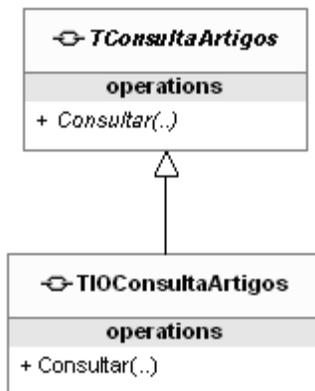


Figura 7. Diagrama de classes de Consultas

### Listagem 2. Método para consulta de artigos

```

{ Adicione no uses a unit do DataModule }
procedure TIOConsultaArtigos.Consultar(
  TipoConsulta: TTipo; Texto: string;
  Filtro: TRevista = nil);
begin
  if Trim(Texto) = EmptyStr then
  begin
    raise Exception.Create(
      'Texto a ser buscado está vazio');
  end;
  case TipoConsulta of
    NomeArtigo:
      begin
        with dtm.ArtigoSelector do
          begin
            Close;
            Command.Clear;
            Command.Add('SELECT * FROM ANY TArtigo ');
            Command.Add('WHERE Nome LIKE ` +
              QuotedStr('%' + Texto) +
              Command.Add('ORDER BY Nome');
          end;
        end;
      PalavraChave:
        begin
          with dtm.ArtigoSelector do
            begin
              Close;
              Command.Clear;
              Command.Add('SELECT * FROM ANY TArtigo ');
              Command.Add('WHERE PalavrasChave LIKE ` +
                QuotedStr('%' + Texto + '%' +
                Command.Add('ORDER BY Nome');
            end;
          end;
        end;
        Resumo:
          begin
            with dtm.ArtigoSelector do
              begin
                Close;
                Command.Clear;
                Command.Add('SELECT * FROM ANY TArtigo ');
                Command.Add('WHERE Resumo LIKE ` +
                  QuotedStr('%' + Texto + '%' +
                  Command.Add('ORDER BY Nome');
                end;
              end;
            end;
          if Filtro <> nil then
            dtm.ArtigoSelector.Command.Add(
              'AND Revista.Nome = ` + QuotedStr(
                Filtro.Nome));
            dtm.ArtigoSelector.Open;
            if dtm.ArtigoSelector.RecordCount = 0 then
              raise Exception.Create('Busca sem resultados');
            end;
          end;
        end;
      end;
  end;
end;

```

A primeira observação na implementação do método é como foi efetuada a busca. Veja que passamos para o *ArtigoSelector* um código que parece em muito o SQL padrão. Mas olhe bem, veja que não mencionamos nas sentenças nomes de tabela ou campos, mas nome de classes e suas propriedades.

Se dermos agora uma olhada na estrutura como foi montado o método, podemos ver que segue uma das implementações do padrão *Strategy*. No formulário principal da aplicação para utilizar nossa classe de consulta, adicione alguns componentes para que fique semelhante à Figura 8.

O *dsArtigo* deve estar ligado ao *dtm.ArtigoSelector* e o *DBGrid* ao *dsArtigo*. O *dsRevista* é ligado ao *RevistaSelector* (se preferir coloque o componente no *DataModule*), que possui as seguintes propriedades alteradas: *AutoOpen* para *True*, *ObjectClassName* para *TRevista*, *Connector* para *dtm.InstantIBXConnector1* e *Command* para “SELECT \* FROM TRevista”.

O *DBLookupComboBox1* mostrará o nome da revista utilizando o *dsRevista*. Codifique os eventos *OnShow* e *OnClose* do formulário com o seguinte código, respectivamente (não esqueça de adicionar a unit do *DataModule*):

```

OnShow
dtm.ArtigoSelector.Connector.Connect;

OnClose
dtm.ArtigoSelector.Connector.Disconnect;

```

Ao iniciar a aplicação conectamos no banco, ao fecharmos, desconectamos.

## Adicionando artigos e revistas

Vamos agora criar uma classe que pode ser identificada como uma classe de serviço, que é responsável por cadastrar ou editar um artigo. Antes, vamos definir os formulários que serão utilizados por essa classe. Teremos uma hierarquia simples de formulários, um formulário base de cadastro e mais dois especializados para *TArtigo* e *TRevista*. Crie um novo formulário e o defina conforme a **Figura 9**.

Crie um novo formulário, herdando-o de *TCadBaseF* e adicione um *InstantExposer*, da guia *InstantObjects*. Em *Name* digite "ArtigoExposer", em *ObjectClassName* escolha *TArtigo* e por fim em *FieldOptions.foObjects* escolha *True*. Adicione um *DataSource* ao formulário, chamando-o de "dsArtigo" e faça a ligação ao *ArtigoExposer*.

Dando um duplo clique sobre o *ArtigoExposer*, adicione as propriedades da classe e arraste-as para o formulário, onde automaticamente serão criados os campos e seus rótulos. Não adicione a propriedade *Revista.Nome*, pois nela vamos criar um campo *lookup*.

Insira um *InstantSelector* ajustando suas propriedades conforme a **Tabela 1**, ligando-o a um *DataSource* ("dsSelector").

Propriedade	Valor
Name	RevistaSelector
AutoOpen	True
Command	SELECT * FROM TRevista
FieldOptions.foObjects	True
Connector	dtm.InstantIBXConnector1
ObjectClassName	TRevista

**Tabela 1.** Propriedades de dsSelector

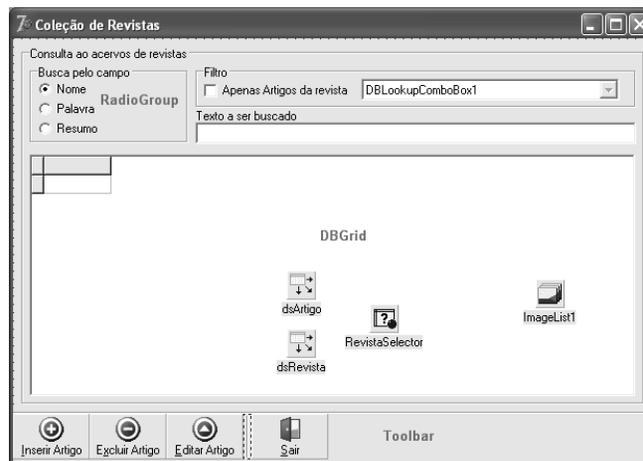
Para o *lookup* utilize um *DBLookupComboBox*. Veja como configurá-lo na **Tabela 2**.

Propriedade	Valor
DataSource	dsArtigo
DataField	Revista
ListSource	dsSelector
KeyField	Self
ListField	Nome

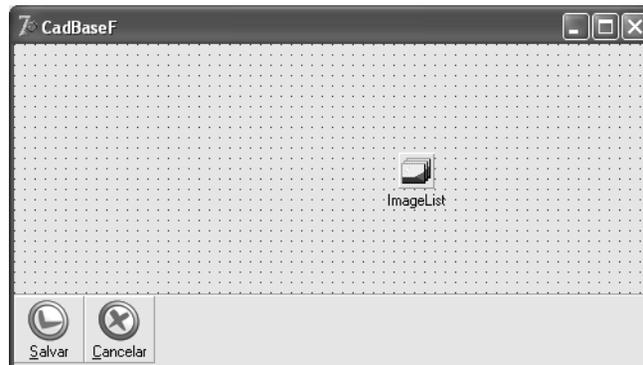
**Tabela 2.** Propriedades do LookupComboBox

Organize a tela de cadastro de artigos conforme a **Figura 10**.

No evento *OnClick* dos botões *Salvar* e *Cancelar* digite o código da **Listagem 3**, ele respectivamente salva os dados inseridos e cancela a edição.



**Figura 8.** Tela principal do sistema de exemplo



**Figura 9.** Formulário base para cadastros



**Figura 10.** Tela para cadastro de artigos

**Listagem 3. Evento OnClick dos botões Salvar e Cancelar**

```

procedure TCadArtigoF.ToolButton1Click(
  Sender: TObject);
begin
  inherited;
  ArtigoExposer.PostChanges;
  ModalResult := mrOk;
end;

procedure TCadArtigoF.ToolButton2Click(
  Sender: TObject);
begin
  inherited;
  ModalResult := mrCancel;
end;

```

É preciso criar o cadastro das Revistas, assim crie mais um formulário que descenda de *TCadBase*. Adicione um *InstantExposer*, chame-o de “RevistaExposer”, configurando-o como fizemos no cadastro de artigos, e o evento *OnClick* dos botões é similar também ao cadastro de artigos, com a diferença que referenciamos o *RevistaExposer*. Formate o formulário como na **Figura 11**.

Chegou a hora de implementar a classe de serviço. Adicione uma nova unit, salve-a como “*Servicos.pas*” e em sua cláusula *uses* faça referência à *BusinessModel.pas*. No ModelMaker, crie um novo diagrama de classes e dê o nome de “*Classes – Serviço*”. Nesse diagrama insira uma classe chamada “*TEdicao*”, com dois métodos estáticos:

```

EditRevista(Revista: TRevista): boolean
EditArtigo(Artigo: TArtigo): boolean

```

Adicione ao modelo a unit e mande o ModelMaker gerar o código da classe nessa unit e implemente os métodos como na **Listagem 4**.

**Listagem 4. Classe de serviço da aplicação**

```

implementation
...

{ Adicione no uses a unit BusinessModel e a unit de
  cada formulário de cadastro }
class function TEdicao.EditArtigo(
  Artigo: TArtigo): Boolean;
var
  Form: TCadArtigoF;
begin
  Form := TCadArtigoF.Create(nil);
  try
    Form.ArtigoExposer.Subject := Artigo;
    Result := Form.ShowModal = mrOk;
  finally
    Form.Free;
  end;
end;

class function TEdicao.EditRevista(
  Revista: TRevista): Boolean;
var
  Form: TCadRevistaF;
begin
  Form := TCadRevistaF.Create(nil);
  try
    Form.RevistaExposer.Subject := Revista;
    Result := Form.ShowModal = mrOk;
  finally
    Form.Free;
  end;
end;

```

**Fazendo “rodar”**

Para fazer a consulta criamos a classe *TIOConsultaArtigos*, portanto vamos usá-la no evento *OnKeyUp* do campo



**Figura 11.** Tela de cadastro de revistas

de texto a ser buscado (no formulário principal), conforme a **Listagem 5** (atente para os nomes dos componentes).

**Listagem 5. Realizando a busca**

```

procedure TPrincF.EditTextoKeyUp(Sender: TObject;
  var Key: Word; Shift: TShiftState);
var
  lConsulta: TIOConsultaArtigos;
  lRevista: TRevista;
begin
  lRevista := nil;
  if ckFiltro.Checked then
    lRevista :=
      dtm.RevistaSelector.CurrentObject as TRevista;
  if Key = VK_Return then
    begin
      lConsulta := TIOConsultaArtigos.Create;
      case RadioGroup1.ItemIndex of
        0: lConsulta.Consultar(NomeArtigo,
          EditTexto.Text, lRevista);
        1: lConsulta.Consultar(PalavraChave,
          EditTexto.Text, lRevista);
        2: lConsulta.Consultar(Resumo, EditTexto.Text,
          lRevista);
      end;
      lConsulta.Free;
    end;
end;

```

Para cadastrar um artigo, no botão *Inserir Artigo* temos o código da **Listagem 6**.

**Listagem 6. Inserindo um novo artigo**

```

{ Declare no uses a unit da classe TEdicao }
procedure TPrincF.btInserirClick(Sender: TObject);
var
  lArtigo: TArtigo;
begin
  lArtigo := TArtigo.Create;
  try
    if TEdicao.EditArtigo(lArtigo) then
      dtm.ArtigoSelector.AddObject(lArtigo);
  finally
    lArtigo.Free;
  end;
end;

```

Para excluir, chamamos o método *Delete* do *DataSet*:

```

dtm.dsArtigo.DataSet.Delete;

```

Para editar, precisamos obter qual objeto *TArtigo* está selecionado e após sua edição atualizamos sua exibição:

```

if TEdicao.EditArtigo(
  dtm.ArtigoSelector.CurrentObject as TArtigo) then
  dtm.ArtigoSelector.Refresh;

```

Veja que nos códigos anteriores fazemos uso da classe de serviço. No formulário principal, não fazemos menção alguma aos formulários de cadastro, apenas a classe *TEdicao* sabe como utilizá-los. Isso chama-se *encapsulamento*.

Não criamos um acesso direto ao cadastro de Revistas, sendo feito através do cadastro de Artigos. Veja que ao lado do *DBLookupComboBox* temos um *SpeedButton*, que é encarregado de chamar o cadastro de Revistas. Veja sua implementação na **Listagem 7**.

**Listagem 7. Abrindo o formulário de cadastro de revistas**

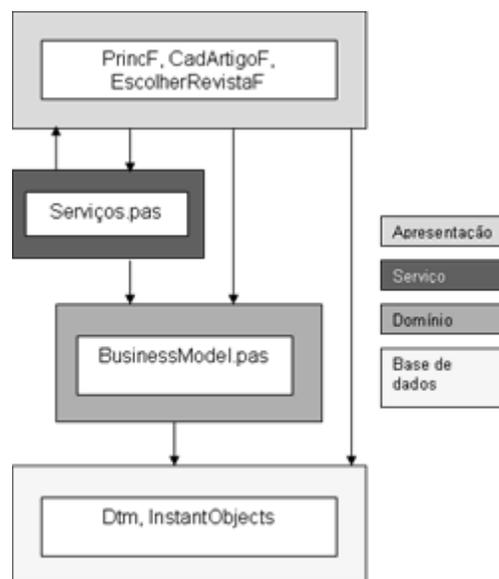
```
{ Adicione no uses BusinessModel e Servicos }
procedure TCadArtigoF.btAdicionaRevistaClick(
  Sender: TObject);
var
  lRevista: TRevista;
begin
  inherited;
  lRevista := TRevista.Create;
  if TEdicao.EditRevista(lRevista) then
    RevistaSelector.AddObject(lRevista);
  lRevista.Free;
  ComboRevistas.Refresh;
end;
```

Após tudo isso, temos o aplicativo pronto, mesmo que simples, em camadas (**Figura 12**). Na **Figura 13** temos a aplicação em execução.

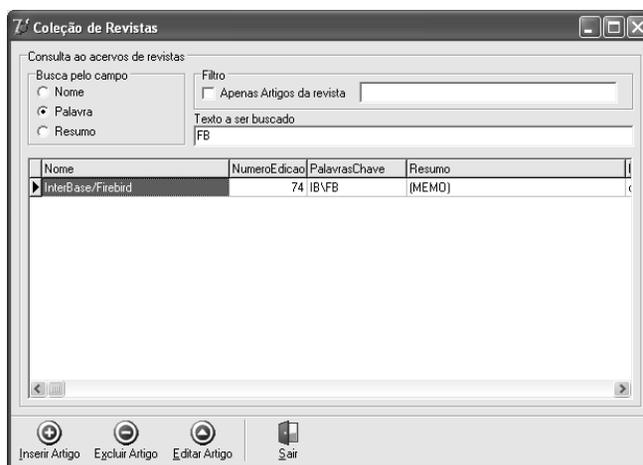
**Conclusão**

Ao longo desses artigos, vimos que o desenvolvimento Orientado a Objetos não é moroso como diz a lenda. Em minha opinião tal afirmação é utilizada por aqueles que não conhecem OO e têm medo de mudanças, mesmo que positivas. Utilizando a documentação de Caso de Uso, o desenvolvimento torna-se mais dirigido a atender o que o usuário realmente precisa e solicita.

Utilizando ferramentas eficientes como Delphi, Model-Maker e o framework InstantObjects ganhamos tempo porque mantemos nosso foco nas regras de negócio e nos problemas do nosso cliente (e não de infra-estrutura). Um abraço a todos e até a próxima! ■



**Figura 12.** Camadas do sistema de exemplo



**Figura 13.** Aplicação sendo executada

