

Aplicações MultiBanco

Adapte sua aplicação para funcionar com diferentes BDs, usando técnicas de desenvolvimento em camadas e P00



CESAR BLUMM

(cesarblumm@yahoo.com.br)

é tecnólogo em Processamento de Dados pela Universidade de Caxias do Sul (UCS) e pós-graduado em Banco de Dados pela Universidade Luterana do Brasil (ULBRA), Certificação MCP, trabalha em desenvolvimento há 19 anos, atualmente é DBA na Metadados Assessoria e Sistemas trabalhando com Oracle, MS SQL Server e Firebird e desenvolve projetos na plataforma Delphi.



MIGUEL RODRIGUES FORNARI

(fornari@ieee.org) é bacharel e mestre pela Universidade do Rio Grande do Sul (UFRGS) e doutorando na área de Sistemas de Informação Geográfica pela mesma universidade. Professor da Universidade Luterana do Brasil

(ULBRA) e Faculdades SENAC/RS. Também é consultor de diversas empresas na área de administração de SGBD.



FRANCISCO M. TRINDADE

(fmtrindade@m3tech.com.br)

é Engenheiro de Computação pela Universidade Federal do Rio Grande do Sul (UFRGS) e mestrando na área de Engenharia de Software pela mesma universidade. Atualmente trabalha com o desenvolvimento de soluções em Delphi e Firebird, pela empresa M3Tech Tecnologia para o Agronegócio.

Toda *software house*, ao desenvolver seus produtos, tem que viabilizar seus softwares para serem adquiridos pelo número máximo possível de clientes, independente do SGBD que o cliente utilizar. O objetivo deste artigo é mostrar como desenvolver software nessas condições, escrevendo o código-fonte uma única vez, e que possa funcionar para diferentes SGBDs, como Oracle, SQL Server, Firebird e PostgreSQL.

Para tanto, é necessário resolver problemas de compatibilidade entre os diferentes SGBDs, como tipos de dados e funções. Também deve-se criar uma arquitetura incluindo uma camada de persistência, capaz de gerenciar adequadamente o acesso a dados.

Criação de tabelas

Todos os fornecedores de BDs se propõem a armazenar os principais tipos representáveis de dados em tabelas, sendo que alguns têm mais opções do que os outros. Quem trabalha com mais de um fornecedor, provavelmente já investiu um tempo para pesquisar as ferramentas CASE ou para modelagem de diagramas E-R. A maioria delas possui o recurso de traduzir o modelo especificado para os diferentes fornecedores de SGBD, respeitando as suas diferenças.

Para criar uma ferramenta desse tipo para a sua aplicação, você terá que em um primeiro momento delimitar quais os bancos de dados que pretende trabalhar, e realizar um estudo dos tipos compatíveis entre eles. Na **Tabela 1** estão relacionados os tipos mais comuns em cada banco de dados e o equivalente nos outros SGBDs tratados neste artigo.

É importante considerar que além dos tipos de dados relacionados na **Tabela 1**, também existem tipos similares, que possuem funções específicas. Por exemplo: o tipo *nvarchar* no SQL Server também armazena valores alfa-

Tipo	Oracle	MS SQL Server	Firebird	PostgreSQL
Alfanumérico - tamanho variável	Varchar2	Varchar	Varchar	Varchar
Alfanumérico - tamanho fixo	Char	Char	Char	Char
Binário	Long Raw	Image	Blob	Bytea
Data/Hora	Date	DateTime	Date	Date
Inteiro	Int/Integer	Int	Integer	Int8
Númérico com Decimais	Number	Decimal	Decimal	Numeric
Texto	Long	Text	Varchar	Text

Tabela 1. Tipos de dados dos SGBDs

numéricos, porém usa a codificação *Unicode*, usada para representação de caracteres de línguas específicas como o japonês. Quanto à sintaxe da instrução *Create*, quando forem usadas as opções básicas da instrução, o comando é padrão para todos os bancos, conforme exemplo na **Listagem 1**.

Listagem 1. Instrução de criação de uma tabela (tipos do SQL Server)

```

Create Table Cliente
(ID          Int          Not Null,
Codigo      Int          Not Null,
Nome        Varchar(40)  Not Null,
Endereco    Varchar(40), Constraint PK_Cliente Primary Key(ID));

```

A manutenção das tabelas utiliza praticamente uma sintaxe padrão. Uma exceção são as versões anteriores do Oracle, que não permitem retirar um atributo da mesma, sendo necessário salvá-la em outra temporária, recriá-la sem o campo e restaurar as informações para a tabela original.

Triggers/Stored Procedures

A vantagem de se utilizar *Triggers* e *Stored Procedures* é que você consegue colocar as regras do negócio da aplicação no banco de dados, agilizando alguns procedimentos e, principalmente, protegendo a integridade dos dados independentemente do aplicativo que está acessando as informações. Entretanto a portabilidade da aplicação é inversamente proporcional ao uso dos mesmos.

As instruções SQL são padronizadas pelo padrão ANSI e por isso possuem uma boa similaridade entre os diversos fornecedores de SGBD, porém a linguagem empregada no desenvolvimento de *Triggers* e *Stored Procedure* não é controlada por nenhum padrão e cada fornecedor possui uma linguagem que geralmente é incompatível com os outros fornecedores.

Resumindo você terá que dar manutenção em tantas fontes quantos forem os bancos de dados onde a sua aplicação irá executar. Na **Listagem 2** são apresentadas *Triggers* que fazem exatamente a mesma operação, porém observe a sintaxe utilizada por cada fornecedor (neste exemplo citei o Firebird e SQL Server, no endereço para download você encontra também o código para Oracle e PostgreSQL). Todas as *Triggers* controlam a inclusão de um item da nota fiscal, limitando a um máximo de 30 itens.

Listagem 2. Comandos para criação de Triggers em diferentes banco de dados

Trigger para o SQL Server

```

-- Exclui a trigger caso ela já exista
If Object_id ('TR_IN_ITEM','TR') Is Not Null
Drop Trigger TR_IN_ITEM
Go
Create Trigger TR_IN_ITEM
On ITEM
After Insert
as
Declare
@LTOTALITENS INT,
@LNOTA INT
Begin
-- Seleciona o código da nota
que está sendo incluída.
Select @LNOTA = ID
From INSERTED;
-- Totaliza a quantidade de itens
já cadastrados na nota.
Select @LTOTALITENS = COUNT(*)
From ITEM
Where ID = @LNOTA;
-- Verifica se existem mais do que 30 itens na nota.
If @LTOTALITENS > 30
Begin
Raiserror('A nota %d já está cheia. '+
'Inclusão cancelada.', 16, 1, @LNOTA);
Rollback;
End
End
Go

```

Trigger para o Firebird

```

Create Exception Nota_Cheia
'A nota já está cheia. Inclusão cancelada.';

Create Trigger TR_IN_ITEM
For ITEM
After Insert
as
Declare LTOTALITENS INTEGER;
Declare LNOTA INTEGER;
Begin
/* Seleciona o código da nota
que está sendo incluída. */
LNOTA = New.ID;
/* Totaliza a quantidade de itens
já cadastrados na nota. */
Select COUNT(*)
From ITEM
Where ID = :LNOTA
Into :LTOTALITENS;
/* Verifica se existem mais do que
30 itens na nota. */
If (LTOTALITENS > 30) Then
Begin
Exception Nota_Cheia;
End
End

```

Comandos Insert/Update/Delete

Os comandos para inclusão, alteração e exclusão de dados no Delphi normalmente são geradas pelos próprios componentes utilizados para manipular as tabelas do banco de dados, como por exemplo o dbExpress. Porém em si-

tuações específicas, fornecer explicitamente os comandos *Insert*, *Update* e *Delete* pode ser bastante útil.

As sintaxes normalmente são bem similares. Uma das diferenças encontradas é que no SQL Server e no PostgreSQL não é permitido usar “apelidos” (aliases) no nome da tabela nas instruções *Update* e *Delete*. No código a seguir, vemos um exemplo da instrução no Oracle, com o uso do apelido *Cli* para a tabela, o que tornaria o código não-portável.

```
Update Cliente as Cli
Set Nome = 'Teste'
Where Exists
(Select 1 from NotaFiscal as Nota
Where Cli.ID = Nota.IDCliente)
```

Consultas SQL (Select)

As consultas SQL são essenciais aos nossos sistemas e, a princípio, todos SGBD atendem ao padrão ANSI. Porém, os fabricantes de SGBD, para ter um diferencial frente aos seus concorrentes, provêm seus produtos com extensões ao padrão. Essas extensões são muito úteis em algumas situações, porque podem reduzir a codificação necessária e facilitam a apresentação das informações nas aplicações. Entretanto, por elas não serem padronizadas, obrigam o desenvolvedor a escrever versões diferentes e não portáveis das consultas, sendo assim desaconselháveis. Ou seja, muitas vezes deixamos de usufruir de um poderoso recurso oferecido pelo BD, simplesmente para deixar nossa solução portátil.

O caso mais comum é o das funções. Afora as básicas, que estão previstas no padrão, como *Sum*, *Count*, *Min*, *Max* e *Avg*, os SGBDs oferecem várias opções, como pode ser visto na **Tabela 2**, onde estão o objetivo e as versões para cada um dos SGBDs examinados neste artigo.

A solução básica é não incluir a função na consulta SQL, deixando para processar a informação com as funções existentes no Delphi. Essa solução é prática para funções na cláusula *Select*, mas inviável para a cláusula *where*.

Existem outras alternativas para realizar operações semelhantes às apresentadas, aqui apresentamos a forma mais usual. Por exemplo, o formato usado no Firebird para extrair dia, mês e ano de uma data com o *Extract*, também existe no Oracle.

Uma observação importante para quem pretende portar a aplicação para o PostgreSQL é quanto aos “apelidos” nos nomes das colunas do *Select*. Nele, a palavra-chave “as” é obrigatória, enquanto nos demais é opcional. No código a seguir, temos um exemplo que irá funcionar nos quatro SGBDs:

```
Select ID as Cli, Nome as RazaoSocial
From Cliente
```

Em *subselects* no Firebird o uso de apelidos é obrigatório quando é usada a mesma tabela na consulta principal e no *subselect*. No Oracle e no SQL Server não há essa regra. Na **Listagem 3**, a referência à tabela *Cliente* na linha 5 é interpretada no Firebird como a tabela declarada na linha 4 e não como a tabela da linha 2.

Listagem 3. Consulta com Subselect

```
1 Select ID, Nome
2 From Cliente
3 Where Exists
4 (Select 1 From Cliente Cli2
5 Where Cliente.ID = Cli2.ID
6 And Cli2.Endereco Is Null)
```

No Oracle e no SQL Server, o interpretador entende que *Cliente* diz respeito a tabela declarada na linha 2. Para resol-

Objetivo	Oracle	SQL Server	Firebird	PostgreSQL
Selecionar um pedaço de uma string	Substr(coluna, pos. Inicial, Tamanho)	Substring(coluna, pos.Inicial, Tamanho)	Substring(coluna From pos.Inicial For Tamanho)	Substring(coluna, pos.Inicial, Tamanho) Substr(coluna, pos.Inicial, Tamanho, Substring(coluna from pos.Inicial for Tamanho)
Separar o ano de uma data	To_Date(Coluna, 'YYYY')	Year(Coluna)	Extract(Year From Coluna)	Extract(Year From Coluna)
Separar o mês de uma data	To_Date(Coluna, 'MM')	Month(Coluna)	Extract(Month From Coluna)	Extract(Month From Coluna)
Separar o dia de uma data	To_Date(Coluna, 'DD')	Day(Coluna)	Extract(Day From Coluna)	Extract(Day From Coluna)
Resto da divisão	MOD(Coluna1, Coluna2)	Coluna1 % Coluna2 (Operador)		Coluna1 % Coluna2 (Operador)
Data corrente	Current_Date, Current_TimeStamp	Current_TimeStamp	Current_TimeStamp	Current_TimeStamp
Concatenação	String String	String + String	String + String	String String
Arredondamento	Round(Valor, Nro. Decimais)	Round(Valor, Nro. Decimais)	Não existe	Round(Valor, Nro.Decimais)

Tabela 2. Uso de funções nos SGBD

ver esse problema de portabilidade você precisa informar um apelido na tabela da linha 2 conforme a **Listagem 4**.

Listagem 4. Consulta com subselect portátil para os quatro SGBD

```
1 Select ID, Nome
2   From Cliente As Cli1
3 Where Exists
4   (Select 1 From Cliente Cli2
5    Where Cli1.ID = Cli2.ID
6    And Cli2.Endereco Is Null)
```

Caso você pretenda portar a aplicação para o Oracle versão 8, será necessário alterar a sintaxe das junções, porque não existem as opções *Inner Join* e *Outer Join*. No caso do *Inner Join* é mais simples, pois basta colocar a comparação do *join* na cláusula *where* e o nome da tabela no *From*, conforme exemplo da **Listagem 5**.

Listagem 5. Troca de sintaxe da consulta de Inner Join para Where

```
Select NotaFiscal.ID, Cliente.Nome
From NotaFiscal
Inner Join Cliente on
  NotaFiscal.IDCliente = Cliente.ID
```

Alterar para

```
Select NotaFiscal.ID, Cliente.Nome
From NotaFiscal, Cliente
Where NotaFiscal.IDCliente = Cliente.ID
```

No caso do *Outer Join*, a mudança é simples, porém a sintaxe não ajuda muito e o entendimento da cláusula não é tão fácil. Observe na **Listagem 6**, que além das alterações do *Inner Join* também foi acrescido o operador (+) à direita do campo *Departamento* da tabela *Departamentos* no *where*. Esse operador indica o *Outer Join*. O sinal (+) deve ficar ao lado da coluna onde a informação poderá faltar na tabela. Está sintaxe funciona somente para o Oracle.

Listagem 6. Troca de sintaxe da consulta de Outer Join

```
Select NotaFiscal.ID, Cliente.Nome
From NotaFiscal
Left Outer Join Cliente on NotaFiscal.IDCliente = Cliente.ID
```

Alterar para

```
Select NotaFiscal.ID, Cliente.Nome
From NotaFiscal, Cliente
Where NotaFiscal.IDCliente = Cliente.ID (+)
```

Tabela de Caracteres

Na criação do banco de dados existe a possibilidade de informar qual a tabela de caracteres será utilizada. Essa tabela de caracteres identifica quais os caracteres são válidos para serem armazenados, e está associada a língua utilizada por quem irá gerar informações para o sistema.

Uma vez definida qual será a tabela utilizada, existem várias restrições feitas pelos SGBD para trocar essa opção. Na melhor hipótese será possível apenas trocar para outra tabela similar a anterior, com risco do SGBD trocar acentos e coisas do gênero.

Se o banco de dados for instalado com as opções padrões dos SGBDs, existem situações que geram resultados diferentes entre os fornecedores que merecem a nossa atenção, por exemplo, quando classificamos uma consulta SQL com a cláusula *Order By* com um campo que possua conteúdo nulo em algumas linhas da tabela. No Oracle e no Firebird a ordem dessas linhas será depois de todas as letras e números, já no SQL Server vem antes.

Também conforme a configuração da tabela de caracteres, as letras maiúsculas e minúsculas terão ou não distinção. Por exemplo, se você criar duas tabelas onde o campo chave é alfanumérico e a tabela detalhe possuir uma chave estrangeira para a tabela mestre, mesmo assim será possível ocorrer o seguinte tipo de situação:

```
Chave da Tabela Mestre: 'CD01'
Chave da Tabela Detalhe: 'cd01'
```

Observe que na tabela mestre o código está em maiúsculo e no detalhe o código está em minúsculo, o banco gravará dessa forma nas respectivas tabelas sem problema nenhum. Quando for montada uma instrução SQL selecionando essa coluna tanto faz se o argumento for "CD01", "cd01", "Cd01" ou "cD01" a linha será encontrada. Porém se na sua aplicação você carregar um cursor com várias linhas da tabela e for interpretar a seleção dentro do código Delphi, para o Delphi maiúsculo sempre será diferente de minúsculo, claro.

Outra situação importante nesse caso é quando for necessário converter um banco ou uma tabela com uma configuração onde exista distinção de maiúsculo e minúsculo para outro onde não exista essa distinção, e nela já existirem duas linhas uma em um formato e outra em outro formato, por exemplo: "CD01" e "cd01", neste caso quando for incluir a segunda linha nesse banco de dados, ocorrerá um erro de chave primária duplicada.

Essa configuração é padrão no SQL Server, nos outros bancos todos são instalados fazendo distinção de maiúsculas e minúsculas.



Camada de persistência

Uma possível solução para a utilização de SGBDs de diferentes fornecedores é a criação de uma camada de persistência independente, que seja responsável pela criação dos comandos SQL específicos de cada fornecedor, deixando a camada de negócio da aplicação isolada da solução de banco de dados. Será demonstrado neste artigo um exemplo de uma camada de persistência simples, que pode ser utilizada com esse objetivo.

Para a implementação da camada de persistência, será utilizado o padrão de projeto *Abstract Factory*, que estabelece uma interface abstrata para uma classe, que será instanciada através de classes descendentes. O diagrama de classes do padrão *Abstract Factory* é mostrado na **Figura 1**.

No modelo que implementaremos será criada uma classe abstrata de persistência, que terá em sua interface as funções de *Insert*, *Delete*, *Update* e *Select*, de forma que a aplicação possa, através dessa interface, realizar as operações necessárias com o banco de dados.

As funções declaradas na interface dessa classe de persistência genérica serão implementadas em classes de persistência descendentes, uma para cada tipo de banco de dados que necessitar ser utilizado. O diagrama de classes da camada de persistência proposta é mostrado na **Figura 2**.

Exemplo

De forma a exemplificar a criação e utilização da camada de persistência proposta, foi criado um pequeno aplicativo, que possui os conceitos de *Item*, *Cliente* e *Nota Fiscal*, sendo que a *Nota Fiscal* possui ligação com um *Cliente* e diversas instâncias de *Item*. A **Figura 3** mostra o modelo conceitual do exemplo implementado.

Para a implementação do banco de dados do aplicativo, serão necessárias três tabelas, cada uma representando um dos conceitos propostos anteriormente. A estrutura de cada tabela do banco de dados é mostrada na **Tabela 3**.

Tabela	Campos
Cliente	Id, Codigo, Nome e Endereco
Item	Id, Codigo, Preco, Quantidade e IdNotaFiscal
Nota Fiscal	Id, Codigo e IdCliente

Tabela 3. Estrutura das tabelas do aplicativo-exemplo

Seguindo o modelo proposto, para cada tabela especificada, será criada uma classe abstrata de persistência, responsável por determinar a interface a ser utilizada pelas classes descendentes.

Dessa forma, foram criadas as classes *TClienteBD*, *TItemBD* e *TNotaFiscalBD*, que são responsáveis pela persistência das classes de negócio *TCliente*, *TItem* e *TNotaFiscal*, respectivamente. O código das classes *TCliente* e *TClienteBD* é mostrado na **Listagem 7**.

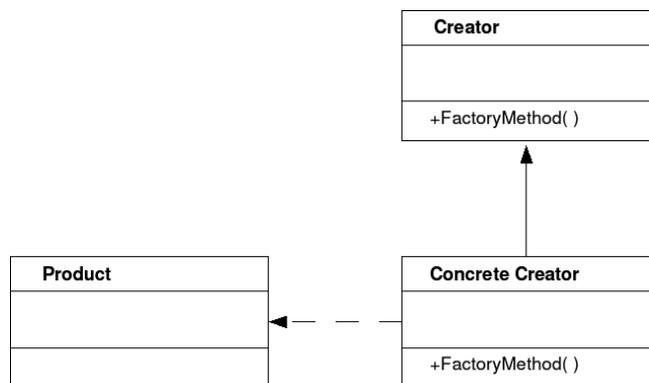


Figura 1. Diagrama de classes Abstract Factory

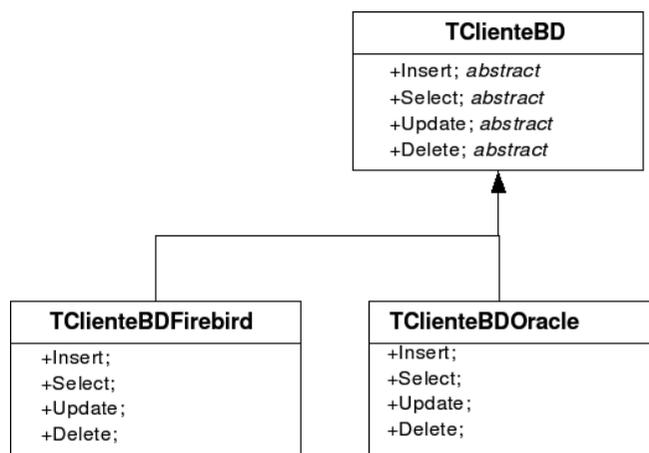


Figura 2. Diagrama de classes da camada de persistência independente de banco de dados

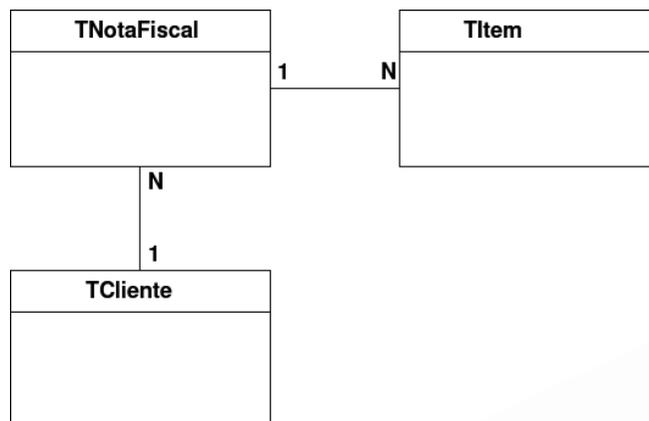


Figura 3. Modelo Conceitual do aplicativo

Listagem 7. Código de implementação das classes TCliente e TClienteBD

```

unit uClasses;

interface

uses
  Classes, ContNrs;

type
  TCliente = class
  private
    FId: Integer;
    FCodigo: Integer;
    FEndereco: string;
    FNome: string;

  public
    property Id: Integer read FId write FId;
    property Codigo: Integer read FCodigo
      write FCodigo;
    property Nome: string read FNome write FNome;
    property Endereco: string read FEndereco
      write FEndereco;
  end;

  TClienteBD = class
  public
    function Insert(ACodigo: Integer; ANome,
      AEndereco: string): TCliente; virtual; abstract;
    procedure Delete(var ACliente: TCliente);
      virtual; abstract;
    procedure Update(ACliente: TCliente);
      virtual; abstract;
    function Select(CondicoesSQL: string):
      TObjectList; virtual; abstract;
  end;

```

O exemplo pode ser criado em qualquer versão do Delphi. A classe *TCliente* possui somente os campos *Codigo*, *Endereco* e *Nome*, que definem o cliente, e o campo *Id*, que é utilizado como chave primária no banco de dados.

A classe *TClienteBD* é uma classe abstrata, portanto contém somente a declaração dos métodos necessários para acesso ao banco de dados, que são declarados com as diretivas *virtual* e *abstract*, deixando a implementação para ser realizada pelas classes descendentes.

O primeiro tipo de banco de dados escolhido para o aplicativo foi o Firebird. Assim, foi criada uma implementação concreta das classes de persistência do sistema em implementação, chamadas de *TClienteBDFirebird*, *TItemBDFirebird* e *TNotaFiscalBDFirebird*.

Nota: O código completo das classes *TItem* e *TNotaFiscal*, assim como as implementações das classes descendentes, está disponível para download. Mas sua concepção é bastante semelhante ao já apresentado neste artigo, diferenciando apenas pela quantidade e nomenclatura dos campos de cada tabela.

Essas classes não só implementam a interface da classe ascendente, como também possuem funções de conexão e desconexão com o banco de dados, além de outras rotinas necessárias para a sua execução. **A Listagem 8** mostra a classe *TClienteBDFirebird*.

Listagem 8. Classe TClienteBDFirebird

```

uses SqlExpr, Provider;
...

TClienteBDFirebird = class(TClienteBD)
  private
    BdLib: string;
    Bd: string;
    Password: string;
    Username: string;
    AppPath: string;

    { Variáveis de conexão }
    dbExpConnection: TSQLConnection;
    dbExpQuery: TSQLQuery;
    DbExpDataSetProvider: TDataSetProvider;
    GeneratorQuery: TSQLQuery;

  procedure Conectar;
  procedure Desconectar;
  procedure ExecutarQuerySQL(stringSQL: string);
  function SelectQuerySQL(
    stringSQL: string): Integer;
  function GetValorGenerator() : Integer;
  public
    constructor Create;
    destructor Destroy; override;

    function Insert(ACodigo: Integer; ANome,
      AEndereco: string): TCliente; override;
    procedure Delete(var ACliente: TCliente); override;
    procedure Update(ACliente: TCliente); override;
    function Select(CondicoesSQL: string):
      TObjectList; override;
  end;

```

A **Listagem 9** mostra a implementação dos métodos *Conectar* e *Desconectar* da classe *TClienteBDFirebird*, que são invocados na *Create* e *Destroy* da classe, respectivamente.

Na **Listagem 10** temos a implementação dos outros métodos da classe *TClienteBDFirebird*.

A classe deve ser definida na unit *uCliente*, criada anteriormente. A classe criada deve implementar os métodos abstratos declarados na classe *TClienteBD*, usando-se a diretiva *override* nas funções abstratas que serão sobrescritas.

Essa função recebe os parâmetros do objeto a ser criado, monta a instrução SQL para execução no banco e realiza a inserção. Após a operação, um objeto do tipo *TCliente* é criado com os parâmetros utilizados. O método *GetValorGenerator* é responsável por verificar o valor da chave primária, executando um comando *select* no *Generator* criado no banco (nesse caso, *GEN_CLIENTE_ID*).

Nota: A implementação das classes dos outros tipos de banco de dados, assemelha-se em muito com a classes criadas anteriormente; a modificação refere-se apenas as características de cada banco de dados (como por exemplo, o método *Conectar*).

Apresentação dos dados do banco

Com as camadas de persistência e negócio prontas, foi criado um formulário para apresentação do aplicativo desenvolvido. Esse formulário possui somente ligação com as classes de persistência abstratas, isolando-se da implementação específica da persistência do programa.

No momento de ativação do formulário, ele então instancia a camada de persistência específica que será utili-

zada, através de um tipo definido na aplicação, sendo esse a única parte da camada superior do programa que teria que ser modificada no caso de mudança no tipo de banco de dados utilizado.

Listagem 9. Métodos para conexão e desconexão com o banco de dados

```

procedure TClienteBDSQLServer.Conectar;
begin
  try
    BdLib := '<caminho>\fbclient.dll';
    Bd := '<caminho>\ARTIGO.FDB';
    Password := 'masterkey';
    Username := 'SYSDBA';

    dbExpConnection := TSQLConnection.Create(
      dbExpConnection);
    dbExpQuery := TSQLQuery.Create(dbExpQuery);
    DbExpDataSetProvider := TDataSetProvider.Create(
      DbExpDataSetProvider);

    with dbExpConnection do
      begin
        DriverName := 'INTERBASE';
        VendorLib := BdLib;
        ConnectionName := 'IBConnection';
        LibraryName := 'dbexpint.dll';
        GetDriverFunc := 'getSQLDriverINTERBASE';

        Params.Add('SQLDialect=3');
        Params.Add('Database=' + Bd);
        Params.Add('user_name=' + Username);
        Params.Add('password=' + Password);
        LoginPrompt := False;
        Connected := True;
      end;

    dbExpQuery.SQLConnection := dbExpConnection;

    { Query para recuperar o Id do objeto criado }
    GeneratorQuery := TSQLQuery.Create(GeneratorQuery);
    GeneratorQuery.SQLConnection := dbExpConnection;

    with DbExpDataSetProvider do
      begin
        DataSet := dbExpQuery;
        Name := 'DbExpProvider';
      end;

    except
      on E: Exception do
        raise EAbort.Create(
          'TClienteBDFirebird.Conectar: Erro conectando');
    end;
  end;

procedure TClienteBDFirebird.Desconectar;
begin
  try
    if dbExpConnection <> nil then
      begin
        dbExpConnection.Connected := False;
        FreeAndNil(dbExpQuery);
        FreeAndNil(dbExpConnection);
        FreeAndNil(DbExpDataSetProvider);
      end
    else
      raise EAbort.Create(
        'TClienteBDFirebird.Desconectar: ' +
        'Já desconectado');
    except
      on EAbort do
        begin
          raise;
        end;
    end;
  end;

```

Listagem 10. Outras implementações da classe TClienteBDFirebird

```

function TClienteBDFirebird.Insert(ACodigo: Integer;
  ANome, AEndereco: string): TCliente;
var
  SQLString: string;
  VCliente: TCliente;
begin
  SQLString := 'INSERT INTO CLIENTE VALUES (-1, ' +
  SQLString := SQLString + IntToStr(ACodigo) + ', ' +
  SQLString := SQLString + QuotedStr(ANome) + ', ' +
  SQLString := SQLString + QuotedStr(AEndereco) + ')';

```

```

  ExecutarQuerySQL(SQLString);

  VCliente := TCliente.Create;
  VCliente.Id := Self.GetValorGenerator;
  VCliente.Codigo := ACodigo;
  VCliente.Nome := ANome;
  VCliente.Endereco := AEndereco;
  Result := VCliente;
end;

procedure TClienteBDFirebird.Delete(
  var ACliente: TCliente);
var
  SQLString: string;
begin
  SQLString := 'DELETE FROM CLIENTE WHERE ID = ' +
  IntToStr(ACliente.Id) + ' ';
  ExecutarQuerySQL(SQLString);
  FreeAndNil(ACliente);
end;

procedure TClienteBDFirebird.Update(
  ACliente: TCliente);
var
  SQLString: string;
begin
  SQLString := 'UPDATE CLIENTE SET ' +
  'CODIGO = ' + IntToStr(ACliente.Codigo) + ', ' +
  'NOME = ' + QuotedStr(ACliente.Nome) + ', ' +
  'ENDERECO = ' + QuotedStr(ACliente.Endereco) +
  ' WHERE ID = ' + IntToStr(ACliente.FID);
  ExecutarQuerySQL(SQLString);
end;

function TClienteBDFirebird.Select(
  CondicoesSQL: string): TObjectList;
var
  SQLString: string;
  VClientes: TObjectList;
  VClienteTemp: TCliente;
  VCount: Integer;
begin
  SQLString := 'SELECT * FROM CLIENTE WHERE ' +
  CondicoesSQL + ' ';
  SelectQuerySQL(SQLString);
  VClientes := TObjectList.Create(True);
  for VCount := 0 to
    Self.dbExpQuery.RecordCount - 1 do
    begin
      VClienteTemp := TCliente.Create;
      VClienteTemp.FID := Self.dbExpQuery.FieldByName(
        'ID').Value;
      VClienteTemp.FCodigo :=
        Self.dbExpQuery.FieldByName('CODIGO').Value;
      VClienteTemp.FNome := Self.dbExpQuery.FieldByName(
        'NOME').Value;
      VClienteTemp.FEndereco :=
        Self.dbExpQuery.FieldByName('ENDERECO').Value;
      VClientes.Add(VClienteTemp);
      dbExpQuery.Next;
    end;
  Result := VClientes;
end;

```

Para isso, devemos criar um formulário que conterá a interface de nossa aplicação exemplo. Adicione também alguns componentes visuais, como *ListBox*, *TextBox* e *Buttons* para iniciar o exemplo (**Figura 4**).

A **Listagem 11** mostra trechos do código de implementação do formulário principal da aplicação.

A criação dos componentes de interface, assim como declaração das funções de tratamento de eventos não será explicada aqui, por estar fora do escopo deste artigo. De forma a acessar as informações contidas no banco de dados, o formulário deverá possuir as classes de persistência criadas anteriormente, *TClienteBD*, *TItemBD* e *TNotaFiscalBD*.



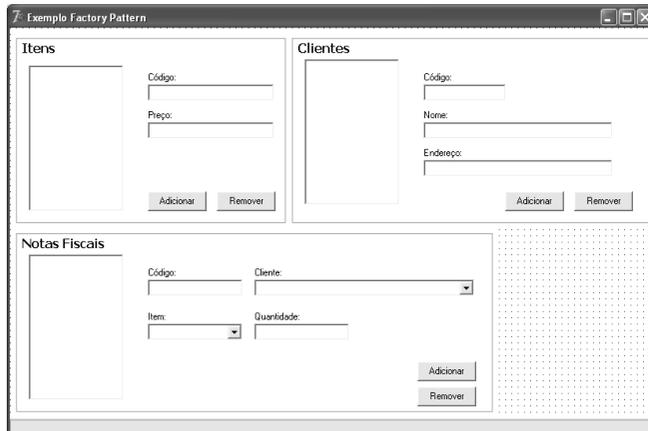


Figura 4. Sugestão de layout da aplicação

Listagem 11. Implementação do formulário principal da aplicação

```
uses uClasses; { unit das classes criadas }

type
{ Tipo enumerado para definição da
camada de persistência }
TTipoBancoDeDados = (sgbdFirebird, sgbdSQLServer);

TMainForm = class (TForm)
...
{ Campos independentes do tipo de SGBD }
FClienteBD: TClienteBD;
...

procedure TMainForm.InicializarCamadaPersistencia(
ATipo: TTipoBancoDeDados);
begin
{ Instância da camada de persistência de
acordo com parâmetro }
case ATipo of
sgbdFirebird: FClienteBD :=
TClienteBDFirebird.Create;
end;
end;

procedure TMainForm.FormActivate(Sender: TObject);
begin
{ Classes específicas de persistência }
Self.InicializarCamadaPersistencia(sgbdFirebird);
{ Inicializar os itens da NotaFiscal ou qualquer
outra necessidade }
end;

procedure TMainForm.FormClose(
Sender: TObject; var Action: TCloseAction);
begin
Self.FClienteBD.Free;
end;
```

Deve-se observar que as classes declaradas no formulário são abstratas, sendo que na criação do objeto deveremos especificar o tipo de banco de dados que utilizaremos. Isso é feito no nosso código através da função *InicializarCamadaPersistencia*, que recebe como parâmetro o tipo de banco de dados desejado.

Trocando de SGBD

Para exemplificar a independência da camada de persistência proposta, vamos simular a troca do sistema de gerencia-

mento de banco de dados, do Firebird, que apresentamos anteriormente, para SQL Server.

No desenvolvimento da camada de persistência para o novo SGBD, iremos implementar novas classes específicas de persistência, chamadas de *TClienteBDSQLServer*, *TItemBDSQLServer* e *TNotaFiscalBDSQLServer*.

Essas classes suportarão as mesmas interfaces propostas pelas classes genéricas de persistência, adequando os parâmetros de conexão, assim como diferenças no SQL existentes no novo tipo de banco de dados. Como exemplo, a **Listagem 12** mostra a implementação do *Conectar* e *Insert* da classe *TClienteBDSQLServer*.

Listagem 12. Implementação da função TClienteBDSQLServer.Insert

```
procedure TClienteBDSQLServer.Conectar;
begin
try
Password := '$root';
Username := 'root';

dbExpConnection := TSQLConnection.Create(
dbExpConnection);
dbExpQuery := TSQLQuery.Create(dbExpQuery);
DbExpDataSetProvider := TDataSetProvider.Create(
DbExpDataSetProvider);

with dbExpConnection do
begin
DriverName := 'MSSQL';
VendorLib := 'oledb.dll';
ConnectionName := 'MSSQLConnection';
LibraryName := 'dbexpmss.dll';
GetDriverFunc := 'getSQLDriverMSSQL';

Params.Add('Database=tempdb');
Params.Add('HostName=HOST\SQLEXPRESS');
Params.Add('user_name=' + Username);
Params.Add('password=' + Password);
Params.Add('OS Authentication=True');
LoginPrompt := False;
Connected := True;
end;

dbExpQuery.SQLConnection := dbExpConnection;

{ Query para recuperar o Id do objeto criado }
GeneratorQuery := TSQLQuery.Create(GeneratorQuery);
GeneratorQuery.SQLConnection := dbExpConnection;

with DbExpDataSetProvider do
begin
DataSet := dbExpQuery;
Name := 'DbExpProvider';
end;

except
on E : Exception do
raise EAbort.Create(
'TClienteBDSQLServer.Conectar: Erro conectando');
end;
end;

function TClienteBDSQLServer.Insert(
ACodigo: Integer; ANome,
AEndereco: string): TCliente;
var
SQLString: string;
VCliente: TCliente;
begin
SQLString := 'INSERT INTO [tempdb].[dbo].[CLIENTE] VALUES (';
SQLString := SQLString + IntToStr(ACodigo) + ',';
SQLString := SQLString + QuotedStr(ANome) + ',';
SQLString := SQLString + QuotedStr(AEndereco) + ')';

ExecutarQuerySQL(SQLString);

VCliente := TCliente.Create;
VCliente.Id := Self.GetValorId;
VCliente.Codigo := ACodigo;
VCliente.Nome := ANome;
VCliente.Endereco := AEndereco;

Result := VCliente;
end;
```

Pode-se observar que o comando SQL utilizado para a execução da inserção é diferente do utilizado anteriormente, pois o valor para ID, que anteriormente era -1 na implementação para o Firebird, aqui não é colocado. Como também o método *Conectar* que possui parâmetros diferentes dos utilizados para conectar ao Firebird.

Nota: Uma outra característica da camada de persistência está na estrutura da tecnologia dbExpress, onde facilmente, somente alterando parâmetros, podemos conectar a outra base de dados.

Com a camada de persistência para o banco de dados SQL Server implementada, basta modificar o tipo de objeto de persistência que é instanciado no início do aplicativo, de forma rápida e fácil, conforme mostrado pela **Listagem 13**.

Listagem 13. Função de inicialização do aplicativo instanciando a camada de persistência para o BD SQLServer

```

procedure TMainForm.FormActivate(Sender: TObject);
begin
  { Self.InicializarCamadaPersistencia(sgbdFirebird); }
  Self.InicializarCamadaPersistencia(sgbdSQLServer);
  Self.InicializarGrpClientes;
end;

procedure TMainForm.InicializarCamadaPersistencia(
  ATipo: TTipoBancoDeDados);
begin
  { Instância camada de persistência de
  acordo com parâmetro }
  case ATipo of
    sgbdFirebird: FClienteBD := TClienteBDFirebird.Create;
    sgbdSQLServer: FClienteBD := TClienteBDSQLServer.Create;
  end;
end;

```

A variável *Atipo* pode ser um parâmetro de configuração do sistema, que indica o SGBD que está sendo utilizado. Após essa pequena modificação, o aplicativo estará pronto para ser executado com o banco de dados SQL Server. Veja na **Figura 5** o exemplo em execução.

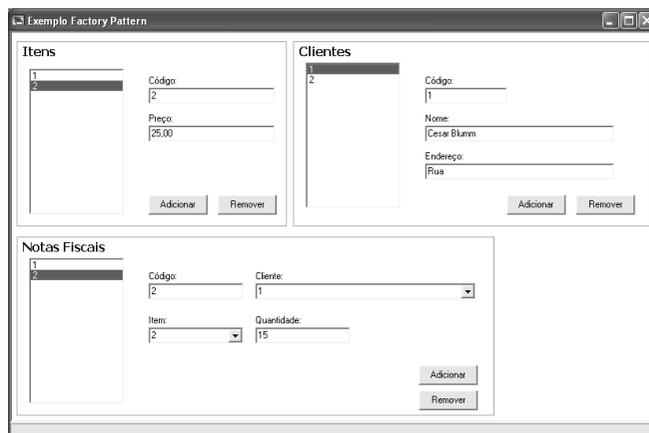


Figura 5. Projeto em execução, utilizando o Firebird

Conclusão

Apesar dos padrões existentes que regem a sintaxe do SQL, é um tanto questionável se esse é um padrão de fato, pois existem muitas diferenças entre os fornecedores de SGBD. Quando alguma empresa pretender portar a sua aplicação para mais do que um SGBD, provavelmente começarão os problemas com o sistema.

É muito importante que em tempo de projeto estejam previstos quais os bancos de dados alvos que serão utilizados pela aplicação, para que essas diferenças sejam tratadas adequadamente. Em tempo de desenvolvimento, sempre que possível, é interessante ir testando a aplicação em todos os bancos de dados onde ela executará, assim outros detalhes que existam entre os SGBD, já vão sendo tratados antes do término do projeto.

Além disso, foi apresentada a implementação de uma camada de persistência independente do sistema de banco de dados, utilizando-se do padrão de projeto *Abstract Factory*. Essa abordagem pode ser muito útil na implementação de aplicações para diferentes tipos de SGBD, possuindo a vantagem de manter a camada de negócios da aplicação isolada da implementação das rotinas de persistência, permitindo assim o reuso de código de aplicação. ■

Links

Site do Firebird

www.ibphoenix.org

Site do Microsoft SQL Server

www.microsoft.com/sql

Site da Oracle

otn.oracle.com

Site do PostgreSQL

www.postgresql.org

