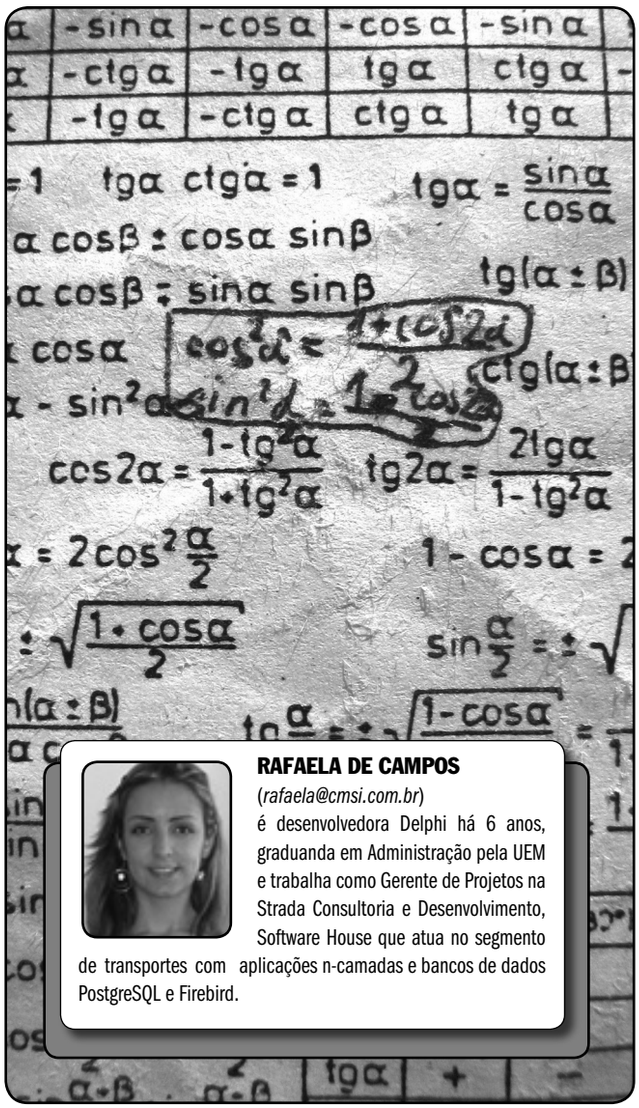
Implementando fórmulas na aplicação

Agregando valor ao seu sistema através da personalização de cálculos internos



RAFAELA DE CAMPOS
(*rafaela@cmsi.com.br*)
é desenvolvedora Delphi há 6 anos, graduanda em Administração pela UEM e trabalha como Gerente de Projetos na Strada Consultoria e Desenvolvimento, Software House que atua no segmento de transportes com aplicações n-camadas e bancos de dados PostgreSQL e Firebird.

Customização é uma das palavras da vez e a cada dia mais o mercado exige que os produtos além de ter qualidade, sejam flexíveis, personalizados e tenham preços competitivos. Agora, como fazer para oferecer um sistema de qualidade, com preço competitivo e que sirva para diversos clientes com regras de negócios diferentes?

A resposta é: parametrização. Uma das maneiras de parametrizar o sistema é possibilitar que os cálculos sejam definidos através de fórmulas criadas pelos próprios clientes. E para demonstrar como utilizar fórmulas dentro de um sistema, neste artigo criaremos uma aplicação simples de compra e venda onde os cálculos do valor total do movimento e do item serão obtidos através do resultado da fórmula parametrizada.

Será usado o *Evaluator* da biblioteca de componentes JEDI, que é um avaliador de expressões para realizar o cálculo das fórmulas do programa, ou seja, ela decodifica e substitui expressões por seus respectivos valores numéricos.

Além disso, teremos uma tela para cadastro da fórmula digitada herdada de um formulário básico de cadastro, demonstrando assim também como utilizar herança de formulários dentro do Delphi.

Obs.: O objetivo deste artigo é demonstrar como tornar uma aplicação flexível e personalizável, não se atendo a princípios básicos, como criar um banco de dados, fazer conexão com banco, criar formulários na aplicação ou relacionamentos Master/Detail.

A classe TEvaluator

A classe *TEvaluator* é definida na unit *JclExprEval* da biblioteca JEDI e é uma classe avaliadora de expressões. Ela, assim como as outras classes de avaliadores de expressão

ClubeDelphi 76.indb 14 27.07.06 15:38:47

da JCL, é descendente da classe *TEasyEvaluator*, também definida na mesma unit. A declaração das duas classes está descrita na **Listagem 1**.

Listagem 1. Declaração das classes TEasyEvaluator e TEvaluator

```
...
TEasyEvaluator = class(TObject)
  private
    FOwnContext: TExprHashContext;
    FExtContextSet: TExprSetContext;
    FInternalContextSet: TExprSetContext;
  protected
    property InternalContextSet: TExprSetContext
      read FInternalContextSet;
  public
    constructor Create;
    destructor Destroy; override;

    procedure AddVar(const AName: string;
      var AVar: TFloat32); overload;
    procedure AddVar(const AName: string;
      var AVar: TFloat64); overload;
    procedure AddVar(const AName: string;
      var AVar: TFloat80); overload;

    procedure AddConst(const AName: string;
      AConst: TFloat32); overload;
    procedure AddConst(const AName: string;
      AConst: TFloat64); overload;
    procedure AddConst(const AName: string;
      AConst: TFloat80); overload;

    procedure AddFunc(const AName: string;
      AFunc: TFloat32Func); overload;
    procedure AddFunc(const AName: string;
      AFunc: TFloat64Func); overload;
    procedure AddFunc(const AName: string;
      AFunc: TFloat80Func); overload;
    procedure AddFunc(const AName: string;
      AFunc: TUnary32Func); overload;
    procedure AddFunc(const AName: string;
      AFunc: TUnary64Func); overload;
    procedure AddFunc(const AName: string;
      AFunc: TUnary80Func); overload;
    procedure AddFunc(const AName: string;
      AFunc: TBinary32Func); overload;
    procedure AddFunc(const AName: string;
      AFunc: TBinary64Func); overload;
    procedure AddFunc(const AName: string;
      AFunc: TBinary80Func); overload;
    procedure AddFunc(const AName: string;
      AFunc: TTernary32Func); overload;
    procedure AddFunc(const AName: string;
      AFunc: TTernary64Func); overload;
    procedure AddFunc(const AName: string;
      AFunc: TTernary80Func); overload;
    procedure Remove(const AName: string);

    procedure Clear;
    property ExtContextSet: TExprSetContext
      read FExtContextSet;
  end;

  TEvaluator = class(TEasyEvaluator)
  private
    FLexer: TExprSimpleLexer;
    FParser: TExprEvalParser;
  public
    constructor Create;
    destructor Destroy; override;
    function Evaluate(const AExpr: string): TFloat;
  end;
...
```

Para que possamos entender o funcionamento da classe *TEvaluator* é importante estudarmos o significado de três conceitos dentro do contexto: Variáveis, Constantes e Funções.

- *Variáveis*: São entidades que representam valores e podem ter seu valor alterado;
- *Constantes*: Assim como as variáveis, representam valores, porém, não podem ser alteradas;
- *Funções*: Representam chamadas a funções internas da aplicação.

Com base nisso, veremos então os métodos principais da classe:

- *AddVar*: Adiciona ao interpretador de expressões uma variável cujo nome será passado como uma *string* e representará a variável interna passada como parâmetro;
- *AddConst*: Da mesma forma que a *AddVar*, passa uma constante para o interpretador, cujo nome será passado como primeiro parâmetro e a constante que representa, o segundo parâmetro;
- *AddFunc*: Adiciona ao interpretador chamada a funções do sistema, que poderão receber até 3 parâmetros, sendo:
  - *TFloatFunc* (sem parâmetros): function Exemplo: Double;
  - *TUnaryFunc* (1 parâmetro): function Exemplo(x: Double): Double;
  - *TBinaryFunc* (2 parâmetros): function Exemplo(x,y: Double): Double;
  - *TTernaryFunc* (3 parâmetros): function Exemplo (x,y,z: Double): Double.

E, por último, porém não menos importante, temos o método *Evaluate* responsável pela execução da fórmula propriamente dita.

Criando o Banco de Dados

No IBExpert, criaremos o banco, chamado "Formulas.fdb", com suas respectivas tabelas, constantes na **Tabela 1.**

Tabela Formulas		
Campo	**Tipo**	**Tamanho**
CODFRM	VarChar	15
DESCRICAO	VarChar	50
TEXTO	VarChar	255
Tabela Movimentos		
Campo	**Tipo**	**Tamanho**
IDMOV	Integer	–
NOMECLIFOR	VarChar	50
DTEMISSAO	Date	–
VLROPCIONAL1	Numeric	15,2
VLROPCIONAL2	Numeric	15,2
VLRTOTALMOV	Numeric	15,2
Tabela ItensMovimentos		
Campo	**Tipo**	**Tamanho**
IDMOV	Integer	–
IDITM	Integer	–
QTDE	Numeric	15,2
VLRUNIT	Numeric	15,2
VLROPCIONAL1	Numeric	15,2
VLROPCIONAL2	Numeric	15,2
VLRICMS	Numeric	15,2
VLRIPI	Numeric	15,2
VLRTOTALITEM	Numeric	15,2

Tabela 1. Tabelas do banco de dados do exemplo

ClubeDelphi 76.indb 15 27.07.06 15:38:49

Serão mencionados somente os campos principais de cada uma das tabelas e não nos preocuparemos com chaves estrangeiras ou normalização do banco de dados. Nas tabelas *Movimentos* e *ItensMovimentos* temos dois valores opcionais que podem representar descontos, acréscimos, fretes ou quaisquer outros valores envolvidos nos cálculos dos totais.

Na tabela *ItensMovimentos*, além dos valores opcionais temos também os valores de impostos incidentes no item, onde o valor do ICMS não fará parte do total do item e o valor do IPI somente entrará no valor da nota.

Criando a aplicação

No Delphi, vamos criar uma nova aplicação e salvá-la como "P_Exemplo.dpr". No formulário principal adicionaremos um *MainMenu* que chamará os demais formulários da aplicação (informe o *Caption* do itens como "Fórmulas" e "Movimentos").

Renomeie o formulário criado por padrão, como "frmPrincipal". A chamada dos itens do menu está descrita na **Listagem 2** (lembre-se que esses formulários serão criados mais adiante no artigo).

Listagem 2. Chamadas dos itens de menus do frmPrincipal

```
procedure TfrmPrincipal.Formulas1Click(
  Sender: TObject);
begin
  frmCadFormulas.Show;
end;

procedure TfrmPrincipal.Movimentos1Click(
  Sender: TObject);
begin
  frmCadMovimentos.Show;
end;
```

Criaremos também um Data Module chamado "dmDados" (salve-o como "udmDados.pas"), para a conexão ao banco usaremos componentes da paleta *dbExpress* conforme mostrado na **Figura 1**.

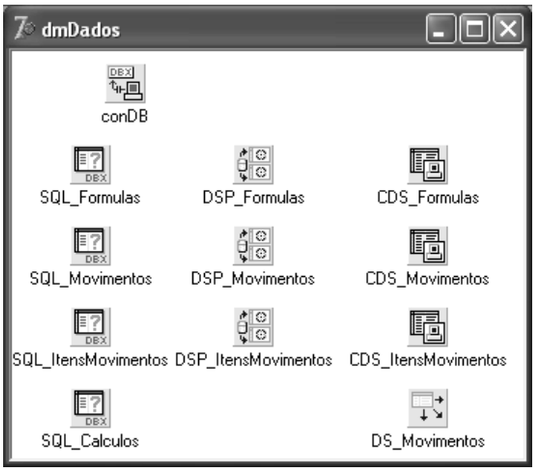


Figura 1. Componentes do dbExpress para o acesso ao banco

Nota: Não entrarei em detalhes sobre a conexão dos componentes dbExpress. Para saber como criar uma conexão, veja edições anteriores da ClubeDelphi ou o mini-curso disponível no Portal do Assinantes no site da revista.

Na **Listagem 3** temos o código de métodos e eventos de componentes do *dmDados*.

No código da listagem anterior, temos o método *CalculaFormula* (que será explicado em maiores detalhes nos próximos tópicos), e a codificação dos eventos *BeforePost* e *AfterPost* da *CDS_Movimentos* e *CDS_ItensMovimentos*. No *BeforePost* temos a chamada à fórmula fazendo com que os campos de totais recebam o resultado dela, e no *AfterPost* faremos a chamada ao método *ApplyUpdates*, para que os dados sejam gravados no banco.

Podemos observar em nosso código que no evento *BeforePost* tanto da *CDS_ItensMovimentos* quanto do *CDS_Movimentos*, só ocorrem chamadas à função *CalculaFormula* e seu resultado é passado como valor para o campo de total da tabela.

A diferença entre os dois eventos é que no *BeforePost* da *CDS_Movimentos* temos que recalcular os totais de todos os itens antes de calcular o total do movimento, pois assim nos certificamos que os itens encontram-se com seus valores atualizados e que o total do movimento estará correto.

Continuando, a ligação entre as tabelas de *Movimentos* e *Itens_Movimentos* (para criar um Master/Detail) é mostrada na **Figura 2.**

O *SQL_Calculos* será usado para cálculo da soma dos totais dos itens de movimentos e sua propriedade *SQL* deverá ter o código da **Listagem 4.**

Figura 2. Ligação entre CDS_ItensMovimentos e CDS_Movimentos

ClubeDelphi 76.indb 16 27.07.06 15:38:50

Listagem 3. Código do dmDados

```
...
 public
    function CalculaFormula(CodFormula: string): Double;
  end;
...
uses U_FuncoesFormulas, JclExprEval, Dialogs;
...
function TdmDados.CalculaFormula(CodFormula: string): Double;
var
  { Variáveis do movimento }
  MvtVlrOpcional1, MvtVlrOpcional2,
  MvtVlrTotal: Double;
  { Variáveis do item de movimento }
  ItmVlrUnit, ItmQtde, ItmVlrTotal, ItmVlrICMS,
  ItmVlrIPI, ItmVlrOpcional1, ItmVlrOpcional2: Double;
  { Variáveis do cálculo }
  Evaluator: TEvaluator;
  Formula: string;
begin
  evaluator := TEvaluator.Create;
  try
    { Variáveis do movimento }
    evaluator.AddVar('MvtVlrOpcional1', MvtVlrOpcional1);
    evaluator.AddVar('MvtVlrOpcional2', MvtVlrOpcional2);
    evaluator.AddVar('MvtVlrTotal', MvtVlrTotal);
    { Funções do movimento }
    evaluator.AddFunc('SItmVlrTotal', SomaItmVlrTotal);
    evaluator.AddFunc('SItmVlrOpcional1', SomaItmVlrOpcional1);
    evaluator.AddFunc('SItmVlrOpcional2', SomaItmVlrOpcional2);
    evaluator.AddFunc('SItmVlrICMS', SomaItmVlrICMS);
    evaluator.AddFunc('SItmVlrIPI', SomaItmVlrIPI);
    { Variáveis do item de movimento }
    evaluator.AddVar('ItmVlrTotal', ItmVlrTotal);
    evaluator.AddVar('ItmVlrOpcional1', ItmVlrOpcional1);
    evaluator.AddVar('ItmVlrOpcional2', ItmVlrOpcional2);
    evaluator.AddVar('ItmVlrIPI', ItmVlrIPI);
    evaluator.AddVar('ItmVlrICMS', ItmVlrICMS);
    evaluator.AddVar('ItmVlrUnit', ItmVlrUnit);
    evaluator.AddVar('ItmQtde', ItmQtde);

    { Localizar na tabela de fórmulas o texto do
      código da fórmula passada como parâmetro }
    if not CDS_Formulas.Active then
      CDS_Formulas.Open;

    CDS_Formulas.Locate('CODFRM',CodFormula,[]);
    Formula := CDS_FormulasTEXTO.AsString;

    { Passagem de Valores para as variáveis }
    if pos('MvtVlrOpcional1', Formula) > 0 then
      MvtVlrOpcional1 := CDS_MovimentosVLROPCIONAL1.AsFloat;
    if pos('MvtVlrOpcional2', Formula) > 0 then
      MvtVlrOpcional2 := CDS_MovimentosVLROPCIONAL2.AsFloat;
    if pos('MvtVlrTotal', Formula) > 0 then
      MvtVlrTotal := CDS_MovimentosVLRTOTALMOV.AsFloat;

    if pos('ItmVlrTotal', Formula) > 0 then
      ItmVlrTotal := CDS_ItensMovimentosVLRTOTALITEM.AsFloat;
    if pos('ItmVlrOpcional1', Formula) > 0 then
      ItmVlrOpcional1 := CDS_ItensMovimentosVLROPCIONAL1.AsFloat;
    if pos('ItmVlrOpcional2', Formula) > 0 then
      ItmVlrOpcional2 := CDS_ItensMovimentosVLROPCIONAL2.AsFloat;
    if pos('ItmVlrIPI', Formula) > 0 then
      ItmVlrIPI := CDS_ItensMovimentosVLRIPI.AsFloat;
    if pos('ItmVlrICMS', Formula) > 0 then
      ItmVlrICMS := CDS_ItensMovimentosVLRICMS.AsFloat;
    if pos('ItmVlrUnit', Formula) > 0 then
      ItmVlrUnit :=  CDS_ItensMovimentosVLRUNIT.AsFloat;
    if pos('ItmQtde', Formula) > 0 then
      ItmQtde := CDS_ItensMovimentosQTDE.AsFloat;
    try
      Result := evaluator.Evaluate(Formula);
    except
      Showmessage('Erro ao executar fórmula. '+
        'Verifique a sintaxe digitada.');
      Result := 0;
    end;
  finally
    evaluator.Free;
  end;
end;

procedure TdmDados.CDS_ItensMovimentosBeforePost(
  DataSet: TDataSet);
begin
  CDS_ItensMovimentosVLRTOTALITEM.AsFloat := CalculaFormula('002');
end;

procedure TdmDados.CDS_MovimentosBeforePost(
  DataSet: TDataSet);
begin
  with CDS_ItensMovimentos do
  begin
    DisableControls;
    First;
    while not Eof do
    begin
      { Somente chamar o Edit e Post para que ele
        passe no BeforePost e refaça os cálculos }
      Edit;
      Post;
      Next;
    end;
    EnableControls;
  end;
  CDS_MovimentosVLRTOTALMOV.AsFloat := CalculaFormula('001');
end;

procedure TdmDados.CDS_ItensMovimentosAfterPost(DataSet: TDataSet);
begin
  CDS_ItensMovimentos.ApplyUpdates(-1);
end;

procedure TdmDados.CDS_MovimentosAfterPost(DataSet: TDataSet);
begin
  CDS_Movimentos.ApplyUpdates(-1);
end;
end.
```

Listagem 4. Código SQL do SQL_Calculos

```
SELECT
  SUM(I.VLROPCIONAL1) SVLROPCIONAL1,
  SUM(I.VLROPCIONAL2) SVLROPCIONAL2,
  SUM(I.VLRICMS) SVLRICMS,
  SUM(I.VLRIPI) SVLRIPI,
  SUM(I.VLRTOTALITEM) SVLRTOTALITEM
FROM ITENSMOVIMENTOS I
WHERE IDMOV=:PIDMOV
```

Importante: Nos *DataSetProviders*, propriedade *Options* a opção *AllowCommandText* deverá ter o valor *True*, para que possamos colocar as instruções SQL nas *ClientDataSets* (adicione para cada componente o respectivo código SQL das tabelas do banco, ex: "select * from FORMULAS" no *CDS_Formulas*). Adicione também os *TFields* em todos os *ClientDataSets*.

Para os formulários da aplicação, criaremos um formulário básico de cadastro, que chamaremos de "frmCadBase" (salve-o

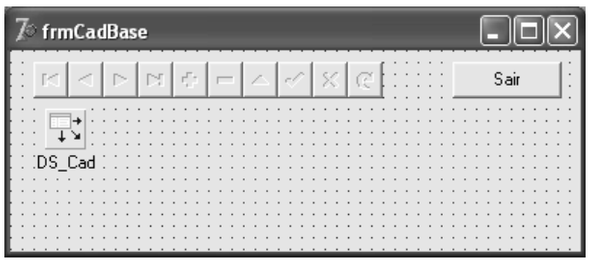


Figura 3. Formulário básico de cadastro contendo o código comum a todos os cadastros

ClubeDelphi 76.indb 17 27.07.06 15:38:51

como "ufrmCadBase.pas", do qual os outros serão descendentes). Para isso, vamos criar um novo formulário e o deixaremos parecido com o mostrado na **Figura 3** (o *DBNavigator* deverá ter sua propriedade *DataSource* setada para *DS_Cad*).

O código, referente ao evento *OnClick* do botão e *OnShow* do formulário, está na **Listagem 5.**

Listagem 5. Eventos do botão e formulário

```
procedure TfrmCadBase.FormShow(Sender: TObject);
begin
  if not DS_Cad.DataSet.Active then
    DS_Cad.DataSet.Open;
end;
procedure TfrmCadBase.btnSairClick(Sender: TObject);
begin
  Close;
end;
end.
```

Através da técnica de herança visual de formulários, criaremos os demais formulários da nossa aplicação, para isso, acesse o menu *File>New>Other*, escolha na aba *P_Exemplo* o item *frmCadBase* e clique em OK.

Crie dois formulários, chamando-os de: "frmCadFormulas" e "frmCadMovimentos" (salvando-os respectivamente como "ufrmCadFormulas" e "ufrmCadMovimentos"). O formulário de cadastro de Fórmulas deve ficar semelhante ao mostrado na **Figura 4.**

Os itens mostrados nas fórmulas disponíveis, conforme descrições mostradas na **Figura 5**, dividem-se em três categorias:

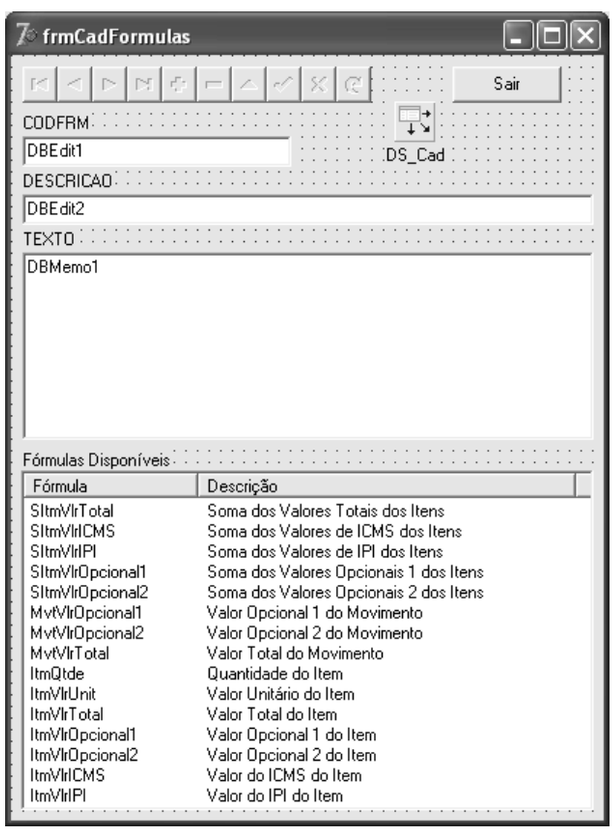


Figura 4. Formulário de cadastro das fórmulas

- Campos que trazem a soma de um determinado campo dos *Itens Movimentos*:
  - *SItmVlrTotal*, *SItmVlrICM*, *SItmVlrIPI*, *SItmVlrOpcional1* e *SItmVlrOpcional2*
- Campos que trazem o valor de um determinado campo dos *Movimentos*:
  - *MvtVlrOpcional1*, *MvtVlrOpcional2* e *MvtVlrTotal*
- Campos que trazem o valor de um determinado campo dos Itens Movimentos:
  - *ItmQtde*, *ItmVlrUnit*, *ItmVlrTotal*, *ItmVlrOpcional1*, *ItmVlrOpcional2*, *ItmVlrICMS* e *ItmVlrIPI*

O formulário de Movimentos ficará semelhante a **Figura 5.**

No formulário de Movimentos, no botão *Atualizar Total* chamaremos simplesmente os métodos *Edit* e *Post* da *CDS_Movimentos*, ou *DS_Cad.DataSet*, visto que o mesmo está ligado a esse *DataSource*.

Essa chamada é feita somente para que o evento *BeforePost* da *CDS_Movimentos* seja executado, pois ele é o responsável direto pela atualização dos valores dos itens e do movimento, como podemos observar na **Listagem 3**, com os métodos e eventos criados no *dmDados*.

A **Listagem 6**, mostra o código a ser adicionado no *Atualizar Total* e evento *OnShow* do formulário.

Listagem 6. Código do frmCadMovimentos

```
...
uses udmDados;
...
procedure TfrmCadMovimentos.FormShow(Sender: TObject);
begin
  inherited;
  { Abrir tabela-filha }
  if not DS_Itens.DataSet.Active then
    DS_Itens.DataSet.Open;
end;
procedure TfrmCadMovimentos.Button1Click(
  Sender: TObject);
begin
  inherited;
  { Somente chamar Edit e Post para que ele refaça
    os cálculos do total do movimento }
  DS_Cad.DataSet.Edit;
  DS_Cad.DataSet.Post;
end;
end.
```

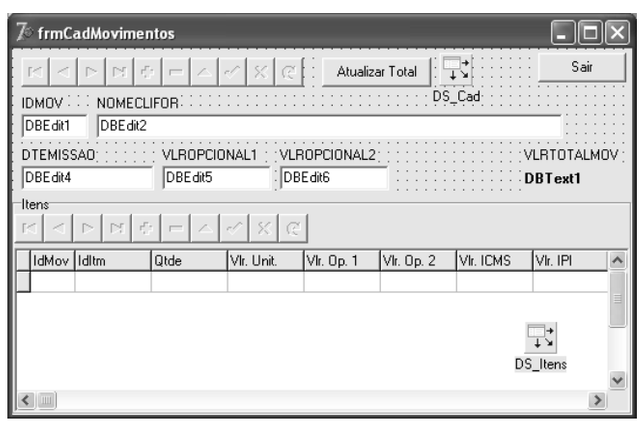


Figura 5. Tela de Movimentação

ClubeDelphi 76.indb 18 27.07.06 15:38:51

Codificando a aplicação

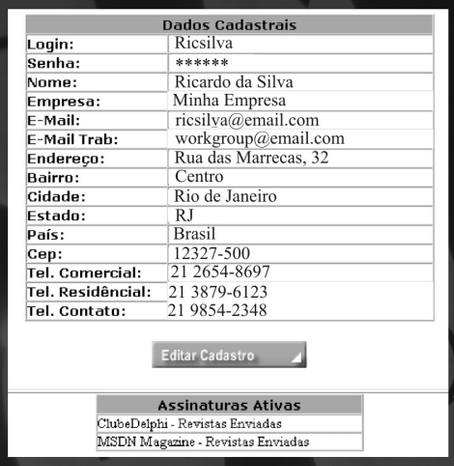
Agora que as telas de nossa aplicação estão criadas e conhecemos a classe *TEvaluator*, vamos codificar nosso projeto para que os valores totais sejam calculados. Criaremos uma unit chamada "U_FuncoesFormulas.pas" (*File>New>Unit*) e digitaremos o código conforme a **Listagem 7**, que conterá as funções chamadas na fórmula.

Listagem 7. Unit U_FuncoesFormulas

```
U_FuncoesFormulas;

interface

uses Dialogs, SysUtils;

  { Exemplo de TFloatFunc }
  function SomaItmVlrTotal: Double;
  function SomaItmVlrICMS: Double;
  function SomaItmVlrIPI: Double;
  { Exemplo de TUnaryFunc }
  function SomaItmVlrOpcional1(Movimento: Double): Double;
  function SomaItmVlrOpcional2(Movimento: Double): Double;

implementation

uses udmDados;
function SomaItmVlrTotal: Double;
begin
  with dmDados.SQL_Calculos do
  begin
    if Active then
      Close;
    ParamByName('PIDMOV').AsInteger :=
      dmDados.CDS_MovimentosIDMOV.AsInteger;
    Open;
    Result := FieldByName('SVLRTOTALITEM').AsFloat;
  end;
end;
function SomaItmVlrICMS: Double;
begin
  with dmDados.SQL_Calculos do
  begin
    if Active then
      Close;
    ParamByName('PIDMOV').AsInteger :=
      dmDados.CDS_MovimentosIDMOV.AsInteger;
    Open;
    Result := FieldByName('SVLRICMS').AsFloat;
  end;
end;
function SomaItmVlrIPI: Double;
begin
  with dmDados.SQL_Calculos do
  begin
    if Active then
      Close;
   ParamByName('PIDMOV').AsInteger := dmDados.CDS_MovimentosIDMOV.AsInteger;
    Open;
    Result := FieldByName('SVLRIPI').AsFloat;
  end;
end;

function SomaItmVlrOpcional1(
  Movimento: Double): Double;
begin
  with dmDados.SQL_Calculos do
  begin
    if Active then
      Close;
    ParamByName('PIDMOV').AsFloat := Movimento;
    Open;
    Result := FieldByName('SVLROPCIONAL1').AsFloat;
  end;
end;
function SomaItmVlrOpcional2(Movimento: Double): Double;
begin
  with dmDados.SQL_Calculos do
  begin
    if Active then
      Close;
    ParamByName('PIDMOV').AsFloat := Movimento;
    Open;
    Result := FieldByName('SVLROPCIONAL2').AsFloat;
  end;
end;
end.
```


Fale conosco!

Conheça a Central de Atendimento on-line da DevMedia. Agora ficou muito mais fácil de acompanhar sua assinatura, alterar seus dados cadastrais e retirar qualquer dúvida sobre nossos serviços e produtos. Para isso basta ter em mãos seu login e senha.



DevMedia GROUP

Na Central de Atendimento você:

- Consulta as perguntas mais frequentes

- Renova por boleto ou cartão de crédito

- Consulta e altera seus dados cadastrais

- Consulta a data de envio de cada exemplar

Para entrar em contato com a DevMedia, basta postar sua dúvida na própria Central de Atendimento através de um sistema web, prático, seguro e fácil de usar. Com essa nova ferramenta você tem a garantia de receber suas respostas em menor tempo e com alto padrão de qualidade.

www.devmedia.com.br/central

ClubeDelphi 76.indb 19 27.07.06 15:38:52

O código da listagem anterior utilizará o *SQL_Calculos*, pois retornará individualmente os valores trazidos pela SQL configurado nesse componente. Por exemplo: a função *SomaItmVlrOpcional2* passa como parâmetro para o *SQL_Calculos* a variável *Movimento*, retornando o campo *SVLROPCIONAL2*.

Para uma melhor compreensão do uso da classe *TEvaluator* foram criadas cinco funções que retornam a soma de valores dos itens, sendo duas do tipo *TFloatFunc* e três *TUnaryFunc*. Além das funções foram criadas variáveis que recebem o valor de campos da tabela de Movimentos ou Itens no registro corrente.

Agora, a parte principal de nosso código está concentrada na função *CalculaFormula*, que apresenta uma sintaxe básica para o uso da classe *TEvaluator* que é:

1) Criação do objeto *Evaluator*;
2) Adição das variáveis e funções ao avaliador;
3) Passagem dos valores às variáveis adicionadas ao avaliador;
4) Execução da fórmula;
5) Tratamento de erros na execução e
6) Destruição do objeto *Evaluator*.

Podemos observar que somente as variáveis devem ter seus valores passados ao objeto avaliador de expressões, pois as funções que adicionamos a ele vão ser chamadas automaticamente ao executarmos a fórmula, não necessitando serem chamadas dentro da função.

Para tratarmos possíveis erros de execução usamos o bloco *try..except..end* fazendo com que, caso ocorra exceção, a fórmula simplesmente retorne valor nulo (Result := 0) e avise sobre a ocorrência do erro numa mensagem personalizada.

E como temos um objeto criado em *runtime*, usamos o bloco *try..finally..end* pois queremos, tendo sido executada a função com sucesso ou não, o objeto seja destruído, para evitarmos erros de memória.

Testando a aplicação

Para o teste de nossa aplicação vamos primeiramente cadastrar nossas duas fórmulas envolvidas nos cálculos (na ordem os campos: CODFRM, DESCRICAO e TEXTO):

001 - Valor Total do Movimento – (SItmVlrTotal + SItmVlrIPI) – MvtVlrOpcional1 e

002 - Valor Total do Item - (ItmQtde * ItmVlrUnit) + ItmVlrOpcional1

E, logo após, vamos cadastrar um movimento, para isso, no formulário de Movimentos, pelo *DBNavigator* crie um novo registro, preencha os campos do movimento, salve e só então crie seus itens pelo *DBNavigator* do *GroupBox* preenchendo seus campos.

Não esqueça de salvar novamente o movimento após a inserção dos itens para calcular seu total. Para certificar do funcionamento do sistema, basta alterar uma das fórmulas e salvar o movimento para que os totais sejam recalculados.

Classes de avaliadores de expressões

Na unit *JclExprEval*, temos outro avaliador de expressões, o *TCompiledEvaluator*. Fora da biblioteca do JEDI podemos citar também alguns componentes disponíveis com a mesma funcionalidade, entre eles: *TCioinaEval*, *TCioinaComplexEval*, *StExpr*, *CalcExpress*, *Parser*, *ExpressionParser* e *RPNParser*.

Temos também um artigo publicado no site da ClubeDelphi que mostra como construir um componente analisador de expressões. E, além disso, para encontrar mais analisadores de expressão, podemos recorrer ao nosso bom e velho Google.

Conclusão

Pudemos ver neste exemplo, através de soluções fáceis de serem implementadas, como podemos agregar valor ao nosso software de modo que ele se torne muito mais atrativo aos olhos de nossos clientes e nos possibilitem um diferencial competitivo.

Com base nisso poderíamos inclusive, possibilitar ao usuário escolher quais fórmulas utilizar através de uma parametrização nos tipos de movimento, ou seja, orçamentos, pedidos etc. flexibilizando assim mais nosso aplicativo. ■

Links

Site de pesquisa de código-fonte de projetos, bibliotecas e outros

www.koders.com

Site do projeto JEDI

sourceforge.net/projects/jcl/

Site do TCioinaComplexEval

www.geocities.com/alexei_cioina/cioinaeval.html

Site do RPNParser 0.2b

www.joaomorais.com.br/pascal/download.php?cat=101

Site da Borland Developer Network com download de avaliadores de expressão

cc.borland.com/Item.aspx?id=22972