



www.devmedia.com.br

[versão para impressão]

Link original: <http://www.devmedia.com.br/articles/viewcomp.asp?comp=27097>

Generics + MVC = Boas práticas e reusabilidade - Revista Clube Delphi Magazine 148

Neste artigo criaremos um aplicativo Win32 que utiliza o padrão MVC para mostrar como o recurso de Generics no Delphi pode trazer vantagens. Vamos além dos exemplos simples de listas tipadas, mostrando como o recursos pode ser útil no dia a dia.

Artigo do tipo **Tutorial**

Recursos especiais neste artigo:

Contém nota Quickupdate , Conteúdo sobre boas práticas .

Generics + MVC = Boas práticas e reusabilidade

Neste artigo criaremos um aplicativo Win32 que utiliza o padrão MVC para mostrar como o recurso de Generics no Delphi pode trazer vantagens. Vamos além dos exemplos simples de listas tipadas, mostrando como o recursos pode ser útil no dia a dia. Claro, explicaremos o que são esses tipos e como eram as coisas antes do seu surgimento, abordando os benefícios de seu uso em classes e interfaces, inclusive com uso de constraints. Veremos também como o próprio Delphi faz uso deste recurso em diversos elementos de sua arquitetura. Além disso, discutiremos um pouco sobre covariância e contravariância para explicitar limitações que o uso de generics ainda possui no Delphi.

Em que situação o tema é útil

Generics podem ser utilizados em quaisquer projetos que venhamos a desenvolver. Através deles podemos tornar nossas arquiteturas mais abstratas e reutilizáveis além de reduzir a quantidade de código de nossos projetos através da eliminação de códigos redundantes.

Ao longo deste artigo veremos como utilizar o recurso de generics, ou tipos genéricos, no Delphi. Os tipos genéricos foram introduzidos ao Delphi a partir de 2009. Com tipos genéricos nós podemos parametrizar determinados tipos, deixando os mesmos para serem definidos somente no momento de seu uso, isso nos traz certa flexibilidade, pois podemos criar classes mais abstratas para determinados tipos de objetos sem nos preocuparmos com qual tipo exato estaremos trabalhando além de reduzir consideravelmente a necessidade de type-casting.

Após passarmos pelos conceitos e exemplos isolados de Generics, criaremos um exemplo onde simularemos um cenário de um projeto MVC – Model View Controller – (ver **Nota do DevMan 1**) onde teremos um controller genérico que espera dois parâmetros, sendo um Form e o outro um objeto de nosso domínio. Para isso criaremos um pequeno cenário com dois cadastros básicos, um de cliente e um de fornecedor. Em nosso exemplo não chegaremos a persistir os dados, pois o objetivo do artigo é demonstrar o uso do generics e fazer o leitor perceber os benefícios do uso do mesmo.

Com este exemplo conseguiremos compreender o uso e os benefícios do Generics além de abordar o uso de constraints para nossos tipos genéricos.

Nota do DevMan 1

MVC, Model View Controller, é um padrão de projeto que prevê a separação da aplicação em três camadas lógicas, Model, View e Controller. A ideia é seguir o princípio da responsabilidade única e garantir que as classes da camada Model cuidem apenas dos aspectos relacionados ao modelo, ou seja, manutenção do estado, regras de negócio, comportamento, abstrações do mundo real etc... Enquanto que a camada View cuida apenas da apresentação para o usuário final, sem conhecer detalhes do sistema e sem possuir regras, a camada View deve possuir apenas elementos necessários para comunicação com o usuário

final. E por fim a camada controller seria a responsável pela comunicação entre a View e o Model. Este padrão é bastante conhecido e difundido e tem como um dos principais benefícios a separação de responsabilidades e com isso um maior nível de desacoplamento e reuso.

O que é Generics ?

O recurso de tipos genéricos permite a definição tardia de determinados tipos de objetos. Sendo assim, nós podemos parametrizar nossas classes, interfaces e métodos informando para elas com qual tipo irão trabalhar somente no momento do uso das mesmas.

Podemos retirar um belo exemplo (**Listagem 1**) disso utilizando um tipo do próprio Delphi, o TList da unit System.Generics.Collections. Antes do TList genérico no Delphi, precisávamos trabalhar com o TObjectList e normalmente necessitávamos realizar type casting (ver **Nota do DevMan 2**) para converter os objetos que eram recuperados do TObjectList, visto que ele trabalhava com instâncias de TObject. Agora com o TList genérico, no momento em que declaramos uma variável deste tipo informamos com qual tipo o TList irá trabalhar, dessa forma, todos os métodos de TList trabalharão com este tipo, inclusive os retornos dos métodos que usamos para recuperar elementos da lista, já retornarão o objeto no tipo correto.

Listagem 1. Exemplo de lista genérica

```
procedure TFrmPrincipal.Teste;
    var MinhaLista:TList<String>;
        itemLista:String;
begin
    MinhaLista :=TList<String>.Create();
    MinhaLista.Add('Texto 1');
    MinhaLista.Add('Texto 2');
    MinhaLista.Add('Texto 3');
    MinhaLista.Add('Texto 4');
    itemLista := MinhaLista[0];
end;
```

No exemplo da **Listagem 1** declaramos um TList de strings e então os parâmetros dos métodos do nosso objeto MinhaLista só aceitarão String. Isto garante que nossa lista só trabalhará com objetos do tipo string e reduz a chance de obtermos um erro por armazenar um objeto de outro tipo na lista.

Outro benefício que podemos ver no exemplo acima é quando recuperamos um item da lista. Como já definimos o tipo da lista na declaração, não precisamos nos preocupar em fazer um Type Casting do item quando recuperamos o mesmo.

Nota do DevMan 2

Type casting é como chamamos o recurso de conversão de um determinado tipo para outro. Por exemplo, se temos uma variável "obj" declarada como TObject porém que recebeu uma instância de TCliente, se tentarmos acessar as propriedades de cliente em "obj" não conseguiremos, para isso precisaremos fazer um typecast para o tipo TCliente. Vale ressaltar que só é possível realizar o typecasting de elementos compatíveis com a instância da declaração. Ou seja, se neste caso tentarmos converter "obj" em um TProduto, vamos obter uma exception de "invalid type casting".

Utilizando tipos genéricos

Os tipos genéricos podem ser atribuídos em Classes, Interfaces, Records e Métodos. A definição mais simples de um tipo genérico é feita simplesmente inserindo o nome de referência para o tipo entre "<>". Por exemplo, como mostra o código a seguir.

```
function TestGenerics<T>(): T;
```

```
var variavelTeste:T;
```

```
begin
```

```
//FAZ ALGUMA COISA
```

```
result := variavelTeste;
```

```
end;
```

Temos o exemplo simples de um método com uso de um tipo genérico, onde T é o nome de seu tipo genérico. Veja que o próprio tipo que é passado como parâmetro é usado como retorno do método e como definição para a variável variavelTeste. O retorno deste método irá variar de acordo com o tipo passado como parâmetro para ele. Para executar o método, basta passarmos o tipo como parâmetro, como mostrado a seguir:

```
TestGenerics<String>();
```

Para a definição na classe, teríamos a mesma sintaxe, como no exemplo da

Listagem 2.

Listagem 2. Definição de tipo genérico em uma classe

```
type
  TTestGenerics<T> = class(TObject)
  private
    FAtributo: T;
    procedure SetAtributo(const Value: T);
  published
    property Atributo:T read FAtributo write SetAtributo;
  end;
implementation
  procedure TTestGenerics<T>.SetAtributo(const Value: T);
  begin
    FAtributo := Value;
  end;
```

Poderíamos utilizar esta classe de duas formas:

- Declarando uma variável do tipo da classe – Neste caso, teríamos que declarar a variável especificando o tipo que a classe trabalharia. Por Exemplo:

```
procedure TestUsandoGenerics();

var variavelTeste: TTestGenerics<String>;

begin

variavelTeste := TTestGenerics<String>.Create();

end;
```

- Herdando em uma classe mais específica – Neste caso, no momento da herança, teríamos que especificar qual o tipo utilizado pela classe. Por exemplo:

```
TTestGenericsClasseFilha = class(TTestGenerics<String>)

//Classe Filha

end;
```

Se não quisermos definir o tipo na definição da herança da classe filha e desejarmos postergar esta definição para o momento de uso da mesma, basta

definirmos um tipo genérico para ela e repassar o tipo para a classe pai, como no exemplo a seguir:

```
TTestGenericsClasseFilha<X> = class(TTestGenerics<X>)
```

```
//Classe Filha
```

```
end;
```

Nos exemplos mostrados até agora, qualquer tipo que for passado como parâmetro será aceito. Em alguns casos isso não é problema, porém em outros pode gerar certas dores de cabeça. O Delphi permite que sejam definidas **constraints** para os tipos genéricos. Estas constraints restringem os tipos possíveis de serem passados como parâmetros. Estas podem ser:

- Exigir que o tipo passado como parâmetro implemente uma ou várias interfaces.
- Exigir que o tipo passado como parâmetro herde de uma determinada classe
- Uma das seguintes palavras reservadas:

o **class** – Indica que o tipo passado como parâmetro precisa ser um tipo de referência, ou seja, uma Classe ou uma Interface. Isso indica que não poderá ser passado um tipo primitivo(inteiro, double etc...) como parâmetro para este tipo genérico.

o **constructor** – Quando usada esta constraint, o Delphi exigirá que o tipo passado como parâmetro tenha um construtor público padrão(Create), ou seja, sem parâmetros.

o **record** – Indica que o tipo passado como parâmetro não pode ser um tipo de referência, ou seja, precisa ser um Value Type, sendo um Record ou um tipo primitivo(inteiro, Double etc...).

Colocando em prática, criando a camada model

Para nosso exemplo foi criado um novo projeto VCL. Vamos primeiramente criar uma classe base para nossos modelos, para isso, adicionamos uma nova unit ao projeto, nomeando como UBaseModel e nela criamos a classe TBaseModel. Declare uma property Id a esta classe, conforme demonstrado na **Listagem 3**.

Listagem 3. Modelo base

```

unit UBaseModel;

interface

uses System.Generics.Collections;

type
    TBaseModel = class(TObject)
    private
        FId: Integer;
        procedure SetId(const Value: Integer);
    published
        property Id: Integer read FId write SetId;
    end;
implementation
{ TBaseModel }
procedure TBaseModel.SetId(const Value: Integer);
begin
    FId := Value;
end;

end.

```

A ideia é que todos os nossos modelos tenham um identificador artificial. Em todo domínio, existem atributos que são candidatos a chaves naturais para nossos modelos, por exemplo, CPF, CNPJ, Cod. Produto, etc. Porém é uma boa prática termos um identificador artificial, que servirá apenas para identificar o objeto e manter o relacionamento do mesmo com outras entidades no banco de dados. Este identificador uma vez atribuído a uma entidade, jamais será alterado, diferentemente de uma chave natural que poderá sofrer mudanças.

Agora, vamos criar duas classes de domínio, TCliente e TFornecedor, que herdam de TBaseModel. Elas possuem campos básicos, pois o objetivo do exemplo é garantir o mínimo de entendimento sobre Generics. Então adicione duas novas units ao projeto e nomeie as mesmas como UCliente e UFornecedor, respectivamente, adicione o código mostrado nas **Listagens 4** e 5.

Listagem 4. Classe TCliente

```

unit UCliente;

interface

uses UBaseModel;

type
    TCliente = class(TBaseModel)

```

```

private
    FAtivo: Boolean;
    FIIdade: Integer;
    FNome: String;
    procedure SetAtivo(const Value: Boolean);
    procedure SetIdade(const Value: Integer);
    procedure SetNome(const Value: String);
published
    property Nome:String read FNome write SetNome;
    property Idade:Integer read FIIdade write SetIdade;
    property Ativo:Boolean read FAtivo write SetAtivo;
end;

implementation

//o código da implementação pode ser obtido no download do projeto

end.

```

Listagem 5. Classe TForneecedor

```

unit UForneecedor;

interface

uses UBaseModel;

type
    TForneecedor = class(TBaseModel)
    private
        FCNPJ: String;
        FRazaoSocial: String;
        FNomeFantasia: String;
        procedure SetCNPJ(const Value: String);
        procedure SetNomeFantasia(const Value: String);
        procedure SetRazaoSocial(const Value: String);
    published
        property NomeFantasia:String read FNomeFantasia write SetNomeFantasia;
        property RazaoSocial:String read FRazaoSocial write SetRazaoSocial;
        property CNPJ:String read FCNPJ write SetCNPJ;
    end;

implementation

//o código da implementação pode ser obtido no download do projeto

end.

```


Todas as classes de domínio de nosso projeto irão herdar de TBaseModel. Esta é uma maneira interessante de manter certos padrões, como por exemplo, o próprio Id do objeto. Além disso, facilitaria futuras implementações de auditoria (usuário/dia/hora de criação e alteração), pois poderíamos implementar isto apenas na classe base. Sem contar que possibilita restringirmos o uso de determinados métodos apenas às classes de domínio reduzindo as chances de serem passados objetos inesperados como parâmetros para as classes de nossa arquitetura.

Criando a camada view

Vamos criar agora as telas de nosso projeto. Teremos apenas quatro formulários para nossos exemplos.

- BaseForm – Formulário base para os cadastros que terá um botão de salvar e um contador de registros “salvos”, visto que não teremos persistência em nosso exemplo.
- FrmPrincipal – Formulário principal do projeto que terá as opções para acesso aos cadastros de cliente e fornecedor.
- FrmCadastroCliente – Formulário de cadastro de clientes, que deverá herdar de BaseForm e será controlado pelo ClienteController.
- FrmCadastroFornecedor – Formulário de cadastro de fornecedores, que deverá herdar de BaseForm e será controlado pelo FornecedorController.

Primeiramente vamos criar nosso formulário base. Para isso, clique com o botão direito sobre o projeto, opção Add New VCL Form, nomeie a unit como UBaseForm. Feito isso, renomeie o formulário para FrmBaseForm e adicione um label e um botão no mesmo, de forma que fique com a aparência semelhante à **Figura 1**.

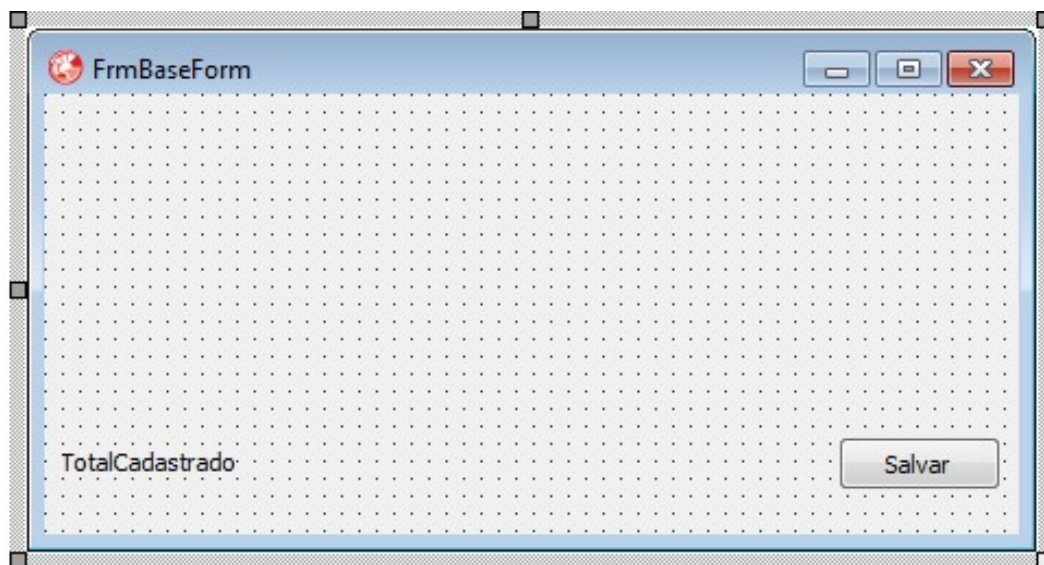


Figura 1. Formulário base

Nomeie o label como `lblTotalCadastrado` e o Button como `btnSalvar`. Isto basta para nosso formulário base.

Vamos criar agora nossos formulários de Cadastro de Cliente e Fornecedores. Esses formulários terão um processo de criação diferente do `BaseForm`, pois os mesmos serão herdados de `BaseForm` usando o recurso de herança visual do Delphi. O recurso de herança visual nos permite facilmente padronizar nossas interfaces com o usuário.

Para criar um formulário herdado, clique com o botão direito sobre o projeto, opção `Add New - Other`. Uma janela se abrirá, selecione a opção `Inheritable Items`, dentro de `Delphi Projects` na árvore do lado esquerdo e selecione `FrmBaseForm` na área que aparecerá no lado direito da tela.

Nomeie este formulário como `FrmCadastroCliente` e repita o procedimento para o cadastro de Fornecedores, nomeando o mesmo como `FrmCadastroFornecedor`.

Inclua controles nestes formulários de forma a atender os modelos de Cliente e Fornecedor que criamos anteriormente. Os formulários terão que ter aparência semelhante a estas das **Figuras 2 e 3**.

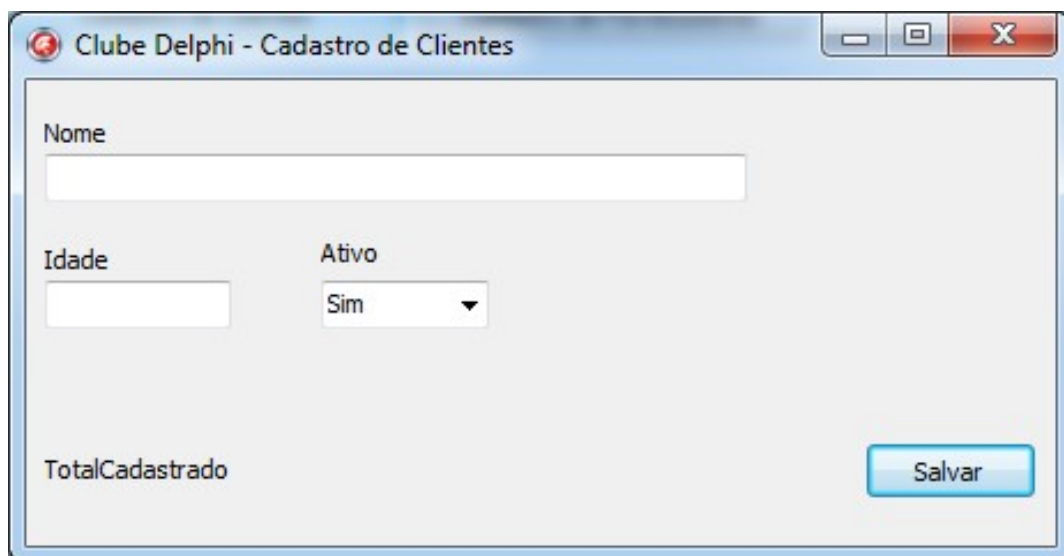


Figura 2. Cadastro de Clientes

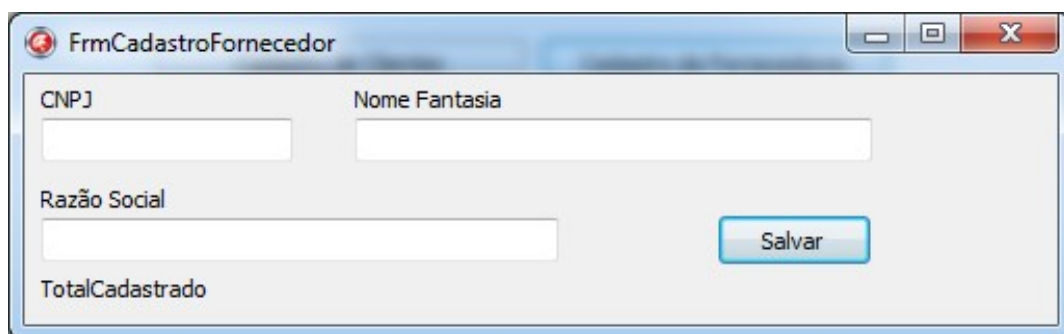


Figura 3. Cadastro de Fornecedores

Altere o nome dos controles de forma que fiquem assim:

- Cadastro de Clientes

- o edtNome

- o edtIdade

- o cbbAtivo

- Cadastro de Fornecedores

- o edtCNPJ

- o edtFantasia

- o edtRazaoSocial

Vale ressaltar a importância de utilizarmos as convenções para nomenclatura dos componentes, como visto, TEdit com prefixo edt, TComboBox com prefixo cbb, TButton com prefixo btn, dessa forma a manutenção fica facilitada e toda vez que estes componentes forem usados no código, o programador já saberá qual o tipo do mesmo.

Criando a camada controller

Agora que já criamos nossos modelos e nossas views, vamos implementar nossa camada controller, que é justamente onde aplicaremos o Generics. Temos um controller base, que receberá dois tipos genéricos, um para indicar o formulário e o outro para indicar o modelo nos quais ele terá que controlar.

Adicione uma nova unit ao projeto, chamada UBaseController. Defina a interface da classe de forma que ela fique como na **Listagem 6**.

Listagem 6. Interface da classe TBaseController

```
01 unit UBaseController;
02
03 interface
04
05 uses UGenericDAO, System.Generics.Collections,
    Vcl.Forms, UBaseForm, UBaseModel;
06
07 type
08     TBaseController<TFormulario: TFrmBaseForm;
        TModelo:TBaseModel > = 09 class(TObject)
09
10     private
11         FFormulario:TFormulario;
12         FModelo: TModelo;
13         FModelosCadastrados: TList<TModelo>;
14     public
15         procedure ShowFormModal(modelo: TModelo);
16         procedure ClearControls();virtual;abstract;
17         procedure BindInView();virtual;abstract;
18         procedure BindInModel();virtual;abstract;
19         procedure Save();
20         function GetTotalCadastrados():Integer;
21         constructor Create();
22         destructor Destroy();override;
23         procedure SalvarClick(sender : TObject);
24         procedure CloseForm(Sender: TObject;
            var Action: TCloseAction);
25     end;
```

Observe que na definição da classe `TBaseController`, indicamos quais os tipos parametrizados que teremos:

```
TBaseController<TFormulario:TFrmBaseForm;TModelo:TBaseModel>
```

Definimos a classe `TBaseController` esperando dois tipos genéricos, `TFormulario` e `TModelo`. Observe que ambos possuem uma constraint. O tipo genérico `TFormulario` possui uma constraint indicando que o mesmo deverá obrigatoriamente herdar de `TFrmBaseForm` enquanto que o tipo genérico deverá obrigatoriamente herdar de `TModelo`. Com isso garantimos que nossos controllers só serão usados com classes que herdaram destes tipos, evitando erros em tempo de execução.

Temos também na seção `Private` (linha 10) três variáveis definidas:

- `FFormulario` – Esta variável armazenará a instância do formulário gerenciado pelo Controller.
- `FModelo` – Esta variável armazenará a instância do modelo gerenciado pelo controller.
- `FModelosCadastrados` – Lista com os modelos “cadsatrados”. Lembrando que não teremos persistência em nosso exemplo.

Na seção `Public` (linha 14) temos os métodos para controle do formulário e do modelo. Observe os que são abstratos e virtuais (linha 16, 17 e 18). Estes métodos dependem do conhecimento dos modelos e das views concretas, com isto não podemos implementar os mesmos em nosso `BaseController`, deixando que esta implementação seja feita pelos controllers concretos. O objetivo destes métodos é o seguinte:

- `ClearControls` – Limpar os controles do formulário gerenciado pelo controller
- `BindInView` – Ler as propriedades do modelo e renderizar nos controles da view.
- `BindInModel` – Ler os controles da view e inserir os dados nas propriedades do modelo.

Na **Listagem 7** vemos a implementação do `Create` e do `Destroy`. No `Create` nós instanciamos nossa lista de modelos cadastrados e em seguida criamos o formulário, de acordo com o tipo do formulário definido no tipo genérico. Após isso atribuímos o evento click do botão salvar ao nosso método `SalvarClick` e o `onClose` do formulário ao nosso método `CloseForm`.

Listagem 7. Implementação do construtor e destrutor do BaseController

```
constructor TBaseController<TFormulario, TModelo>.Create();
begin
    FModelosCadastrados := TList<TModelo>.Create();
    Application.CreateForm(TFormulario, FFormulario);
    FFormulario.BtnSalvar.OnClick := SalvarClick;
    FFormulario.OnClose := CloseForm;
end;

destructor TBaseController<TFormulario, TModelo>.Destroy;
begin
    FFormulario.Release();
    FModelo.Destroy();
    inherited;
end;
```

No destrutor nós liberamos os objetos criados da memória. Um detalhe importante a se destacar é o uso da palavra reservada "override" na declaração do destrutor. Este é um detalhe muito importante, pois o método Destroy da classe TObject(Base de todas as classes do Delphi), é virtual. Um erro muito comum dos desenvolvedores e que causa muita dor de cabeça, é a não definição do destroy como override. Isso faz com que, caso a variável de referência seja do tipo TObject, porém a instancia seja do seu tipo concreto(que possui um destroy sem override), ao chamar variável.destroy, será executado o método Destroy de TObject e não de sua classe concreta, ocasionando um memory leak (objeto instanciado mas não liberado) em seu projeto.

Na **Listagem 8** temos a implementação de outros métodos do BaseController, neste caso é importante destacarmos CloseForm, que seta a action como caHide, libera a referência do modelo e limpa os controles do formulário. Em seguida temos o método Save que "salvaria" o objeto no banco de dados.

Em nosso exemplo ele simplesmente faz um bind do que foi digitado para o objeto, verificamos se é um objeto novo, através da verificação se o Id é igual a 0. Caso seja novo, incluímos em nossa lista de modelos cadastrados e atualizamos a label de total no formulário.

É interessante observar que mesmo sem ter conhecimento de qual modelo e com qual formulário iremos trabalhar, conseguimos implementar um algoritmo básico, deixando que as partes concretas sejam definidas pelos controllers concretos, isto caracteriza o uso do design pattern chamado template method (ver **Nota do**

DevMan 3 e 4).**Listagem 8.** Implementação dos métodos adicionais BaseController

```
function TBaseController<TFormulario,  
TModelo>.GetTotalCadastrados: Integer;  
begin  
    result := FModelosCadastrados.Count;  
end;  
  
procedure TBaseController<TFormulario, TModelo>.CloseForm  
(Sender: TObject; var Action: TCloseAction);  
begin  
    Action := TCloseAction.caHide;  
    FModelo := nil;  
    ClearControls();  
    FFormulario.Hide();  
end;  
  
procedure TBaseController<TFormulario,  
TModelo>.SalvarClick(sender: TObject);  
begin  
    Save();  
end;  
  
procedure TBaseController<TFormulario,  
TModelo>.ShowFormModal(modelo: TModelo);  
begin  
    FModelo := modelo;  
    FFormulario.ShowModal();  
end;  
  
procedure TBaseController<TFormulario, TModelo>.Save;  
begin  
    BindInModel();  
    if(FModelo.Id = 0)then  
    begin  
        FModelo.Id := GetTotalCadastrados() + 1;  
        FModelosCadastrados.Add(FModelo);  
        FFormulario.lblTotalCadastrado.Caption := 'Total de Cadastros:  
    ' + IntToStr(GetTotalCadastrados());  
    end;  
end;  
  
end.
```

Nota do DevMan 3

Design patterns são padrões de projeto que preveem a definição de soluções comuns para problemas recorrentes. Cada design pattern provê a solução de um problema específico. Os design patterns são independentes de tecnologia e podem ser implementados em qualquer linguagem que suporte orientação a objetos. Existem diversos padrões para os mais variados problemas, sendo que os mais famosos estão catalogados no livro conhecido como GOF(Gang of Four) e se dividem em três grandes grupos:

- Patterns Estruturais – São aqueles que cuidam do relacionamento entre as entidades do projeto, fazendo com que as mesmas tenham suas estruturas mais claras e desacopladas, visando a solução de um determinado problema. Ex.: Proxy, Facade, Adapter etc...
- Patterns Criacionais – São aqueles que cuidam de estratégias de criação de objetos, fornecendo uma linguagem única e um local centralizado para criação de determinados tipos de objetos. Ex.: Singleton, Builder, Abstract Factory etc...
- Patterns Comportamentais – São aqueles que cuidam da colaboração entre as entidades do seu projeto, deixando esta comunicação mais fluída e desacoplada entre as mesmas. Ex.: Strategy, Observer, Command etc....

Nota do DevMan 4

Template Method é um design pattern comportamental que prevê a definição de um esboço do algoritmo de determinada rotina, possibilitando que pedaços deste algoritmo sejam sobrescritos em classes filhas, garantindo que o algoritmo principal será executado. O template method parte do princípio de que a classe base conhece o que precisa ser feito, porém não tem conhecimento sobre o como fazer, delegando tal responsabilidade, em determinados passos do algoritmo, para suas classes filhas.

Agora que temos nosso BaseController implementado, precisamos criar nossos controllers concretos, com a definição dos tipos utilizados e com a implementação dos métodos abstratos. Para isso, adicione duas novas units e as nomeie como ClienteController e FornecedorController. As mesmas terão a implementação conforme as **Listagens 9 e 10**, respectivamente. Os controllers criados têm a

responsabilidade de passar para os controles visuais os dados dos objetos e vice-versa.

Listagem 9. Implementação do ClienteController

```
unit UClienteController;

interface

uses UBaseController, UFrmCadastroCliente, UCliente;

type
  TClienteController = class(TBaseController<TFrmCadastroCliente,TCliente>)
    procedure BindInView();override;
    procedure BindInModel();override;
    procedure ClearControls();override;
  end;

implementation

uses
  System.SysUtils;

{ TClienteController }

procedure TClienteController.BindInModel;
begin
  inherited;
  FModelo.Nome := FFormulario.edtNome.Text;
  FModelo.Idade := StrToInt(FFormulario.edtIdade.Text);
  FModelo.Ativo := FFormulario.cbbAtivo.ItemIndex = 0;
end;

procedure TClienteController.BindInView;
begin
  inherited;
  FFormulario.edtNome.Text := FModelo.Nome;
  FFormulario.edtIdade.Text := IntToStr(FModelo.Idade);

  if(FModelo.Ativo) then
    FFormulario.cbbAtivo.ItemIndex := 0
  else
    FFormulario.cbbAtivo.ItemIndex := 1;
  end;

procedure TClienteController.ClearControls;
begin
```

```

    inherited;
    FFormulario.edtNome.Text := '';
    FFormulario.edtIdade.Text := '';
    FFormulario.cbbAtivo.ItemIndex := 0;
end;

end.

```

Na **Listagem 9** temos o controller do cadastro de cliente, onde temos o método `BindInModel`, responsável por carregar os dados da View para o Modelo. Além disso, temos também o método `BindInView`, responsável pela operação inversa, ou seja, carregar os dados do modelo para a View, exibindo os mesmos para o usuário. Por fim temos o método `ClearControls` responsável por limpar os valores da tela.

Listagem 10. Implementação do FornecedorController

```

unit UFornecedorController;

interface

uses UBaseController, UFrmCadastroFornecedor, UFornecedor;

type
  TFornecedorController = class(TBaseController<
    UFrmCadastroFornecedor, TFornecedor>)
    procedure BindInView(); override;
    procedure BindInModel(); override;
    procedure ClearControls(); override;
  end;

implementation

{ TFornecedorController }

procedure TFornecedorController.BindInModel;
begin
    inherited;
    FModelo.CNPJ := FFormulario.edtCNPJ.Text;
    FModelo.RazaoSocial := FFormulario.edtRazaoSocial.Text;
    FModelo.NomeFantasia := FFormulario.edtFantasia.Text;
end;

procedure TFornecedorController.BindInView;
begin

```

```

        inherited;
        FFormulario.edtCNPJ.Text := FModelo.CNPJ;
        FFormulario.edtRazaoSocial.Text := FModelo.RazaoSocial;
        FFormulario.edtFantasia.Text := FModelo.NomeFantasia;
    end;

    procedure TForneecedorController.ClearControls;
    begin
        inherited;
        FFormulario.edtCNPJ.Text := '';
        FFormulario.edtFantasia.Text := '';
        FFormulario.edtRazaoSocial.Text := '';
    end;

end.

```

A **Listagem 10** segue o mesmo padrão da **Listagem 9**, com o controller responsável pela comunicação entre a View e o Model, realizando o carregamento dos dados entre os mesmos com os métodos `BindInView` e `BindInModel`, porém desta vez para o cadastro de Fornecedor.

Criando o formulário principal e juntando tudo

Após criados os modelos, as views e os controllers, só falta agora testarmos nossa aplicação criando um formulário principal, que utilizará os controllers concretos para abriremos nossos formulários.

Adicione um novo formulário ao projeto e nomeie o mesmo como `FrmPrincipal`. Insira dois botões neste formulário, um com caption "Cadastro de Clientes" e nome `btnCadastroClientes` e outro com "Cadastro de Fornecedores" e nome `btnCadastroFornecedores`.

Declare duas variáveis no escopo **public** do formulário, conforme mostrado a seguir:

```
ClienteController:TClienteController;
```

```
FornecedorController:TForneecedorController;
```

Agora basta implementarmos o evento click dos botões para instanciar os controllers e então abrir os formulários, como mostrado na **Listagem 11**.

Listagem 11. Click dos botões do formulário principal

```
procedure TFrmPrincipal.BtnCadastroClientesClick(Sender: TObject);
begin
    if(ClienteController = nil)then
        ClienteController := TClienteController.Create();

    ClienteController.ShowFormModal(TCliente.Create());
end;

procedure TFrmPrincipal.BtnCadastroFornecedoresClick(Sender: TObject);
begin
    if(FornecedorController = nil)then
        FornecedorController := TFornecedorController.Create();

    FornecedorController.ShowFormModal(TFornecedor.Create());
end;
```

Generics na API do Delphi

Agora que já vimos como implementar nossas próprias classes genéricas, é só colocar a mão na massa para aumentar a abstração e reuso de nossos projetos. Por outro lado, antes de colocar a mão na massa, é interessante conhecer o que o próprio Delphi já nos fornece para não correremos o risco de reinventar a roda. A seguir na **Tabela 1** segue uma lista dos principais recursos do Delphi que utilizam Generics para que você pode utilizar em seus projetos.

Classes / Interfaces	Descrição
IComparer<T>	Fornece uma interface para comparação dos tipos "menor que", "maior que" or "igual a"
IEqualityComparer<T>	Fornece uma interface para comparação de igualdade, "igual a" ou "diferente"
TComparer<T>	Classe abstrata para implementação da interface IComparer
TEqualityComparer<T>	Classe abstrata para implementação da interface IEqualityComparer<T>
TDelegatedEqualityComparer<T>	Uma classe de comparação que utiliza delegates para determinar igualdade.
TDelegatedComparer<T>	Uma classe de comparação que usa delegates para fazer a comparação.
TStringComparer	Classe usada para comparação de strings.
TArray	Esta classe não contém dados, mas fornece métodos para trabalhar com arrays como por exemplo os métodos Sort e BinarySearch.
TEnumerator<T>	Implementa o design pattern Iterator para seus dados.
TList<T>	Classe genérica que permite o armazenamento de uma coleção genérica de qualquer tipo de dado, sejam objetos ou tipos primitivos.
TQueue<T>	Classe genérica que fornece uma fila de objetos, com algoritmo Filo(first in / last out)
TStack<T>	Classe genérica que fornece uma pilha de objetos com algoritmo FiFo(first in / first out).
TPair<TKey, TValue>	Estrutura simples de par chave/valor.
TDictionary<TKey, TValue>	Coleção de estruturas chave/valor.
TObjectList<T: class>	Coleção genérica para armazenamento de objetos.
TObjectQueue<T: class>	Fila genérica para objetos.
TObjectStack<T: class>	Pilha genérica de objetos.

[abrir imagem em nova janela](#)

Tabela 1. APIs do Delphi que utilizam Generics

Conclusão

Neste artigo vimos o que são tipos genéricos no Delphi, como utilizar os mesmos e as restrições que podemos aplicar a eles. Construímos um exemplo básico com um MVC, com a criação de um controller genérico e outros controllers concretos.

Buscamos demonstrar como criar seus tipos genéricos e também quais as classes nativas do Delphi que implementam este recurso.

Escolhi o exemplo do MVC, pois assim conseguimos acrescentar um pouco mais ao leitor, além de mostrar que Generics podem e devem ser explorados juntamente com padrões e boas práticas de desenvolvimento, ao invés de se limitar a classes de serviços específicos.

Os tipos genéricos nos fornecem grande flexibilidade no desenvolvimento. Com eles podemos desacoplar mais nosso código e torná-lo mais elegante. Com o uso de tipos genéricos, conseguimos criar classes mais coesas que dispensam a necessidade de typecasting. Aliando generics a design patterns e boas práticas de desenvolvimento, podemos obter maior reuso e facilitar a manutenção de nosso código.

Use sua imaginação para criar arquiteturas mais flexíveis em seus projetos no dia a dia. A combinação de Generics com Design pattern é explosiva, onde podemos conseguir um aumento de produtividade altamente relevante através do maior reuso de código.



Ricardo Da Silva Coelho

Graduado em Análise de Sistemas, Pós-graduando em Engenharia de software, mais de 7 anos de experiência em desenvolvimento de software, onde atuou em projetos de diversos segmentos como varejo, telecomunicações, ERPs etc e nas m [...]