

Nesta seção você encontra artigos sobre técnicas que poderão aumentar a qualidade do desenvolvimento de software

Padrões de Projeto – Parte 1

Entendendo os principais padrões de desenvolvimento de sistemas

Nesse artigo veremos

- O que são padrões de projeto e como estão organizados;
- Como implementá-los em Delphi;
- Como utilizá-los em nosso dia-a-dia através de exemplos práticos.

Qual a finalidade

Mostrar o uso adequado de padrões de projeto

em uma aplicação Delphi.

Quais situações utilizam esse recurso?

O desenvolvimento de aplicações utilizando-se de padrões pode ser feito em qualquer tipo de sistema, desde aplicações Win32 até complexos softwares de gerenciamento.



Paulo Quicoli

pauloquicoli@devmedia.com.br

Editor técnico da revista ClubeDelphi, formado em processamento de dados pela FATEC-TQ. Atua como analista e desenvolvedor da ControlM Informática (www.controlm.com.br). Utiliza Delphi desde de sua primeira versão, entusiasta do desenvolvimento orientado a objetos tem publicado vários artigos sobre o assunto. Blog: www.pauloquicoli.spaces.live.com.

Os padrões de projeto, ou *design patterns*, como são também conhecidos, estabelecem um conjunto de soluções para problemas que sempre aparecem ao se desenvolver um sistema, especialmente utilizando orientação a objetos. Esses problemas foram agrupados em três categorias: *criacional*, *estrutural* e *comportamental*. Cada categoria contém uma série de padrões envolvidos, vou listá-los aqui a caráter de informação.

- Padrões criacionais
- Abstract Factory;
- Builder;
- Factory Method;
- Prototype;
- Singleton.



Resumo do DevMan

Os padrões de projeto são soluções para problemas recorrentes que surgem no desenvolvimento de sistemas. Foram inicialmente catalogados pela GoF (Gang of Four). Quatro autores, Erich Gamma, Richard Helm, Ralph Johnson e John Vlissides, que lançaram o livro mais famoso sobre o assunto: *Design Patterns: Elements of Reusable Object-Oriented Software*. Padrões de projeto são considerados como boas práticas entre os desenvolvedores, por isso veremos como aplicar alguns desses padrões em Delphi, em situações reais que nos afligem no dia-a-dia.

- Padrões estruturais
 - Adapter;
 - Bridge;
 - Composite;
 - Decorator;
 - Façade;
 - Flyweight;
 - Proxy.
- Padrões comportamentais
 - Chain of Responsibility;
 - Command;
 - Interpreter;
 - Iterator;
 - Mediator;
 - Memento;
 - Observer;
 - State;
 - Strategy;
 - Template Method;
 - Visitor.

Nesta série de artigos quero abordar cada grupo de padrões e criar um exemplo prático que demonstre utilização de alguns deles.

Padrões criacionais

Nesta categoria de padrões encontramos soluções no que diz respeito à criação de objetos. Por exemplo, em como manter uma única instância de um objeto. Um dos padrões de desenvolvimento mais conhecido e que é bastante abordado no dia-a-dia é o *Singleton*. Vejamos como implementar soluções utilizando esse padrão de forma bastante prática.

Digamos que um determinado tipo de objeto de sua aplicação só pode ser instanciado uma única vez e então ser utilizado por toda a aplicação. Como resolvemos isso? Vamos aplicar o padrão *singleton* para garantir que esse objeto só exista uma única vez e que também tenhamos um ponto de acesso único a ele, ou seja, vamos acessar esse objeto apenas pelo *singleton*.

Inicie um novo projeto Win32 no *Delphi 7*, ou qualquer outra versão que preferir, usando o menu *File>New>Application* e salve-o como *PadraoSingleton*. Observe que um *Project Group* é criado por padrão. Vamos utilizá-lo para manter posteriormente os outros exemplos que virão. Desta forma teremos um catálogo para consulta. Vamos salvar o *Project Group* como *Padrões*. Para isso clique com o botão direito do mouse sobre ele escolha *Save Project Group*.

Nota

É importante lembrar que os padrões aqui demonstrados não estão associados ou ligados a determinada versão do Delphi, ou seja, esse artigo foi criado com o auxílio do Delphi 7, mas todos os conceitos e estruturas de código podem ser totalmente utilizados em qualquer outra versão da ferramenta.

Nosso *singleton* irá manter a instância de uma classe responsável pela leitura e gravação no registro do *Windows*. Adicione uma nova *unit* ao projeto utilizando o menu *File>New>Unit* e salve-a como *SingletonU.pas*. Nela vamos adicionar as seguintes classes vistas na **Listagem 1**.

Listagem 1. Estrutura das classes

```
IRegistro = interface
  ['{E89DC81B-6004-4DB1-B5ED-3227B67A8915}']
  function LeRegistro(Nome: string): string;
  procedure GravaRegistro(Nome: string; Valor: string);
end;

TRegistro = class(TInterfacedObject, IRegistro)
public
  function LeRegistro(Nome: string): string;
  procedure GravaRegistro(Nome: string; Valor: string);
end;

TRegistroSingleton = class
public
  class function Instance : IRegistro;
end;

const
  Chave = 'Software\Padroes';

var
  RegistroGlobal: IRegistro;
```

Nota

A leitura e gravação do registro do *Windows* não será comentada aqui por não fazer parte do escopo do artigo.

A implementação do padrão é feita na classe *TRegistroSingleton*. Veja que ela possui um método chamado *Instance*. Esse método é responsável por instanciar um objeto que implemente a interface *IRegistro* e manter apenas uma única instância existente. No código a seguir podemos ver como isso é feito. Repare que fazemos uma checagem na variável *RegistroGlobal* para verificar se ela está instanciada. Caso não esteja, seu conteúdo será *nil* indicando que não há nenhuma instância da variável em memória.

```
class function TRegistroSingleton.Instance:
  IRegistro;
begin
  if RegistroGlobal = nil then
    RegistroGlobal := TRegistro.Create;
  Result := RegistroGlobal;
end;
```



Nota do DevMan

As interfaces são estruturas de linguagens orientadas a objetos que definem uma espécie de contrato que deve ser obedecido por quem vai utilizá-las ou implementá-las. Nelas não construímos os métodos, apenas os definimos. Quem implementa uma determinada interface é outra classe que tem por obrigação implementar todos os métodos que são expostos. Vimos isso com clareza no código anterior onde temos na interface *IRegistro* a definição de dois métodos e em sua implementação, feita na classe *TRegistro* vemos ambos construídos.

ClubeDelphi PLUS www.devmedia.com.br/clubedelphi/portal.asp

Acesse agora o mesmo o portal do assinante ClubeDelphi e veja uma vídeo-aula de Renato Matos que mostra como trabalhar com interfaces.

<http://www.devmedia.com.br/articles/viewcomp.asp?comp=10096>

A variável *RegistroGlobal* é uma variável pública que é utilizada pelo padrão para que se retorne sempre uma única instância. Vamos ver como utilizamos agora nosso *singleton*.

Adicione ao formulário principal três botões, alterando sua propriedade *Caption* para 1. Ler valores padrão, 2. Alterar valores padrão e 3. Ler valores alterados, respectivamente. Na **Listagem 2** vemos o *onClick* do botão 1.

Listagem 2. Utilizando o padrão

```
procedure TForm1.Button1Click(Sender: TObject);
var
  Registro: IRegistro;
begin
  Registro := TRegistroSingleton.Instance;
  ShowMessage(Registro.LeRegistro('Valor1'));
  ShowMessage(Registro.LeRegistro('Valor2'));
end;
```

Observe que não instanciamos diretamente um objeto que implemente *IRegistro*, solicitamos à classe *TRegistroSingleton* que nos retorne uma instância dele. Ao *debugarmos* o método *Instance* vemos que na primeira vez que ele é chamado a variável *RegistroGlobal* é igual à *nil* e então é criada, porém na segunda vez ela é diferente dessa condição, portando não é criada novamente, sendo apenas retornada. Isso é visto nas **Figuras 1 e 2**.

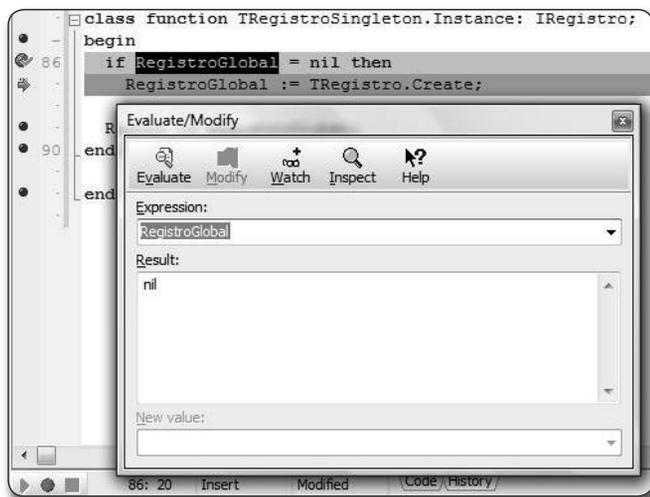


Figura 1. Variável *RegistroGlobal* é nil na primeira execução

Como pode ser visto, a implementação de um *singleton* em *Delphi* nada mais é de que gerenciar a criação de variáveis públicas. Você pode se perguntar agora porque não utilizar apenas a variável pública. Um dos fatores que levam ao uso do padrão *singleton* é economia de memória. Veja que sempre retornamos a mesma instância da variável pública, não a deixamos ser criada mais de uma vez dentro do código.

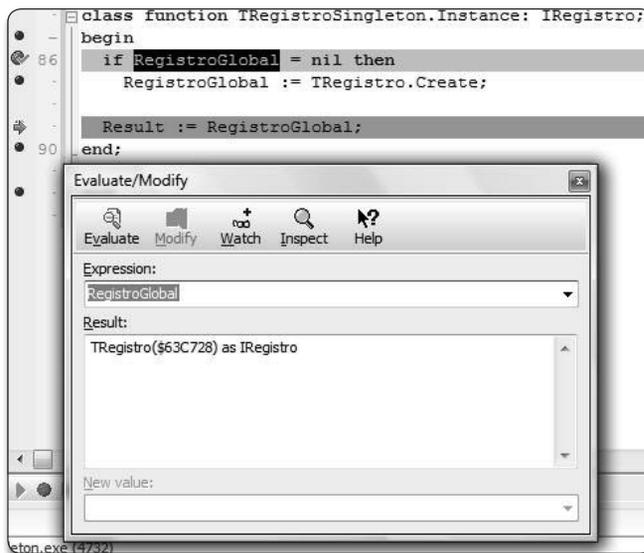


Figura 2. Variável *RegistroGlobal* não é mais nil nas execuções subsequentes

Nota do DevMan

Todo objeto durante a execução de um aplicativo recebe um endereçamento de memória. Porém esse endereçamento só é atribuído no momento em que instanciamos esse objeto. Enquanto não for instanciado ele possui um endereçamento nulo que em cada linguagem possui uma constante diferente. Em *Delphi* esse valor nulo é representado pela constante *nil*, em *C#* por exemplo, é *null* em *Visual Basic* é *nothing*, e assim por diante. Portanto, quando um objeto é igual à *nil* significa que ele ainda não foi instanciado.

Padrão Builder

O *Builder* possui a tarefa de encapsular a construção complexa de um objeto. Por complexa podemos dizer por exemplo a necessidade de se relacionar vários objetos dependentes. Para que essa situação fique mais clara, vamos criar o segundo exemplo do nosso projeto. Adicione ao *Project Group* um novo projeto. Para isso clique com o botão direito no nome do *Project Group* e escolha *Add New Project>VCL Forms Application*. Uma nova aplicação *Win32* será criada. Vamos salvá-la com o nome de *Builder*. A esse projeto *Builder* vamos adicionar agora um *DataModule* e criar uma conexão *IBX* com o banco de exemplo que vem com o *Firebird*, o *Employee.fdb*. O que quero criar aqui é uma situação rotineira, onde criamos um objeto *TIBQuery* dinamicamente para realizar alguma forma de consulta. O porém é que sempre que criamos um *TIBQuery* precisamos associar a ele um objeto do tipo *TIBDatabase* e *TIBTransaction*. Como já foi mencionado, o objetivo do padrão *Builder* é esconder essa complexidade que está definida pela necessidade de se ligar dois objetos no nosso caso, do restante do sistema, ou seja, quem saberá como instanciar um *TIBQuery* é o *Builder*. Dessa forma centralizamos a criação do objeto em um único lugar,

assim, caso algo mude nessa criação vamos alterar apenas no *Builder* e não o sistema todo. Vamos a prática:

Com nosso *DataModule* já conectado ao banco de dados vamos adicionar uma nova *unit* ao projeto e salvá-la como *PadraoBuilder.pas*. Adicione a ela o código da **Listagem 3**.

Listagem 3. Builder de TIBQuery

```
unit PadraoBuilder;

interface

uses DataModule1U, IBQuery;

type
  TIBQueryBuilder = class
    class function CreateInstance: TIBQuery;
  end;

implementation

{ TIBQueryBuilder }

class function TIBQueryBuilder.CreateInstance: TIBQuery;
var
  lquery: TIBQuery;
begin
  lquery := TIBQuery.Create(nil);
  lquery.Database := DataModule1.IBDatabase1;
  lquery.Transaction := DataModule1.IBTransaction1;
  result := lquery;
end;

end.
```

Observe que na seção *uses* adicionamos uma referência ao nosso *DataModule* e também à *unit IBQuery*, que é onde o tipo *TIBQuery* está definido. Posteriormente criamos a classe *TIBQueryBuilder* e adicionamos a ela um método *CreateInstance*. Veja agora o código desse método, ele apenas cria um objeto do tipo *TIBQuery*, faz a associação necessária entre ele e os objetos *IBDatabase1* e *IBTransaction1* que estão no *DataModule1* para que finalmente seja retornado. Essa é a estrutura de um *Builder*. Para vermos em ação vamos adicionar um botão ao formulário do projeto e nele vamos criar uma consulta dinâmica que retorne a quantidade de registros da tabela *Employee* do banco de dados. Na **Listagem 4** vemos como isso é feito.

Observe que nesse momento não adicionamos referência ao *DataModule*, visto que quem o utiliza é o nosso *Builder* que está implementado na *unit PadraoBuilder*. O momento exato em que o padrão é utilizado é quando instanciamos a variável *lquery*.

```
Listagem 4. Builder em execução
uses PadraoBuilder, IBQuery;

({$R *.dfm})

procedure TBuilderF.Button1Click(Sender: TObject);
var
  lquery: TIBQuery;
begin
  lquery := TIBQueryBuilder.CreateInstance;
  lquery.SQL.Add('SELECT COUNT(*) AS TOTAL FROM EMPLOYEE');
  lquery.Open;
  ShowMessage('A quantidade registros na tabela EMPLOYEE é de: '+
    lquery.FieldByName('TOTAL').AsString);
  lquery.Close;
  FreeAndNil(lquery);
end;
```

```
lquery := TIBQueryBuilder.CreateInstance;
```

Nesse momento não temos mais que nos preocupar com o que é necessário para que um *TIBQuery* funcione, isso agora é responsabilidade da classe *TIBQueryBuilder*.

Conclusão

O uso dos padrões de projeto acaba por organizar melhor nosso código porque aplicamos um conceito muito almejado no desenvolvimento orientado a objetos, o *SoC* (*Separation of Concerns*). Uma tradução livre para ele é separação de responsabilidades. Cada procedimento ou função que compõem uma rotina tem sua responsabilidade bem definida e só faz o que tem que fazer. Um exemplo disso é visto no exemplo do *Builder*. Veja que ao aplicá-lo retiramos da *procedure Button1Click* a necessidade dela ter que ligar o *IBDatabase1* e o *IBTransaction1* ao objeto *lquery*.

Nos próximos artigos continuaremos a explorar as outras categorias de padrões e como aplicá-los, mesmo que nossos sistemas não sejam totalmente orientados a objeto. ●

Dê seu feedback sobre esta edição!

A Clubedelphi tem que ser feita ao seu gosto. Para isso, precisamos saber o que você, leitor, acha da revista!

Dê seu voto sobre este artigo, através do link:

www.devmedia.com.br/clubedelphi/feedback

