

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

Aplicando MVC em Delphi

Aprenda a aplicar este poderoso pattern em suas aplicações



Rodrigo Carreiro Mourão

(rodrigocarreiro@rmfactory.com.br)

apaixonado por tecnologia da informação. Atualmente é sócio diretor da RM Factory Soluções em TI, empresa especializada em consultoria na área de TI e desenvolvimento de softwares customizados, parceira da TDS Tecnologia - RJ. Possui os títulos de BDS2006 Win32 Product Certified e Borland Instructor Certified. Como instrutor de treinamentos oficiais já ministrou treinamentos para centenas de profissionais e servidores de órgãos como TJ-RJ, JF, BB, IBGE, MB, EB, etc. Palestrante da Borland Conference 2007. Matenedor da comunidade www.delphisphp.com, a primeira do Brasil 100% feita com Delphi for PHP. Autor de diversos artigos técnicos veiculados nos principais sites e revistas sobre Delphi do Brasil.

Nos últimos anos, temos visto um aquecimento em todas as áreas da economia e a área de TI tem um papel importantíssimo nesse crescimento, pois independente do setor todos estarão direta ou indiretamente ligados a ela. Com todo esse crescimento as necessidades também aumentam a demanda por softwares de qualidade da mesma forma e a única coisa que diminui é o prazo que já era curto e agora está ainda mais apertado.

Não há para onde escapar. As empresas de desenvolvimento de softwares têm contratado consultorias especializadas para poder atender a demanda de seus clientes. Seus softwares, que outrora atendiam, já não atendem mais. Com isso (isso digo por experiência própria), são apresentados a essas empresas e suas equipes técnicas, padrões e boas práticas de desenvolvimento que eram desconhecidas por eles. Digo isso para poder *justificar* o porquê de se ouvir

Resumo DevMan

O padrão de projeto MVC surgiu pela primeira vez em 1979 no projeto Smalltalk na Xerox por Trygve Reenskaug, e na minha opinião foi uma das maiores contribuições na área do desenvolvimento de softwares de todos os tempos pelo menos para a época em que surgiu. De lá pra cá diversas variações do MVC surgiram e a mais conhecida é a MVP (Model View Presenter) que devido a falta de material consistente sobre como o MVC tem que ser, causa grande confusão até em programadores experientes.

Nesse artigo veremos

- Padrões de Projeto;
- MVC;
- Boas Práticas;
- UML;

Qual a finalidade

- Modelar um cadastro de contato utilizando o padrão estrutural MVC como realmente ele deve ser, aplicando padrão de projeto Observer.

Quais situações utilizam esses recursos?

- Mais do que separar em camadas, o MVC deve ser utilizado quando se pretender aumentar o nível de abstração de uma aplicação, diminuir o forte acoplamento entre os componentes da mesma, porém mantendo a comunicação entre eles.

falar tanto em padrões de projeto, boas práticas, OO, etc. Como disse *Steve Jobs* certa vez:

Você não pode simplesmente perguntar ao usuário o que ele quer e tentar dar-lhe isso, quando você conseguir terminar o produto o usuário estará querendo outra coisa.

Eu vejo essa mudança de cenário com bons olhos, aliás com ótimos olhos, pois isso elevará o nível das empresas brasileiras, de sua equipe e conseqüentemente o nível da população de TI do Brasil.

Não podemos ficar fora deste cenário, por isso veremos neste artigo como introduzir nossas aplicações feitas em Delphi nesse novo cenário. Como aplicar em nossas aplicações essas técnicas e boas práticas e assim aumentar o nível de nossas aplicações? Óbvio que falar em boas práticas é falar de diversas soluções reutilizáveis para construção de softwares orientados a objetos eficientes e que em uma única edição não é possível abordar, assim o foco deste artigo será o *pattern arquitetural MVC* em conjunto com o padrão *Observer*. Assim vamos ao que interessa.

Padrões de Projeto

O conceito de padrão de projeto foi usado pela primeira vez na década de 70 pelo arquiteto e urbanista austríaco *Christopher Alexander*. Ele observou na época que as construções, embora fossem diferentes em vários aspectos, todas passavam pelos mesmos problemas na hora de se construir. Eram problemas que se repetiam em todas elas e na maioria das vezes numa mesma fase da construção.

Foi aí que *Christopher* resolveu documentar esses problemas e mais do que isso, passou também a documentar as soluções que eram aplicadas para resolução destes problemas. Entenda que até agora não há nada de programação ou informática e sim projetos, plantas, métricas, definições. Nesse momento surgiam os padrões de projetos para a engenharia civil, padrões esses que descreviam os problemas recorrentes em um projeto de engenharia e a solução reutilizável para este problema.

Em seus livros *Notes on the Synthesis of Form*, *The Timeless Way of Building* e *A Pattern Language* *Christopher* estabelece que um padrão deva ter as seguintes características:

- Encapsulamento;
- Generalidade;
- Equilíbrio;
- Abstração;
- Abertura;
- Combinatoriedade.

Ou seja, um padrão deve ser independente. Deve permitir a construção de outras realizações. Deve representar abstrações do conhecimento cotidiano. Deve permitir a sua extensão para níveis mais baixos de detalhe e deve ser relacionado hierarquicamente. E assim, foram definidos os padrões para projetos na engenharia civil.

Anos mais tarde *Erich Gamma*, *Richard Helm*, *Ralph Johnson* e *John Vlissides* iniciaram suas pesquisas baseados nos trabalhos de *Alexander*. Também conhecidos como *GOF* (*Gang of Four*) eles começaram a descrever e documentar os problemas e soluções para desenvolvimento de softwares orientados a objetos e em 1995 lançaram o livro que se tornou um fenômeno na área de análise e desenvolvimento de software: *Design Patterns: Elements of Reusable Object-Oriented Software*.

Neste momento iniciava-se uma nova fase no desenvolvimento de sistemas, pois agora havia um padrão a ser seguido, e cada padrão apresentando uma solução que poderia ser reutilizada várias vezes para solucionar aqueles problemas recorrentes no desenvolvimento de software. Os padrões *GOF*, como são conhecidos, se dividem em três grandes categorias: *criacionais*, *estruturais* e *comportamentais* somando 23 no total.

Obviamente que o intuito deste projeto não é focar em Padrões de Projeto, isso poderá ser feito em artigos posteriores onde poderemos demonstrar como aplicá-los em Delphi, porém gostaria de rapidamente falar sobre interfaces que são os pilares dos padrões de projetos e que por sinal faremos dela, ao programarmos, nosso exemplo MVC.



Nota do DevMan

Os 23 padrões do GOF são:

Padrões Criacionais

Abstract Factory
Builder
Factory Method
Prototype
Singleton

Padrões Estruturais

Adapter
Bridge
Composite
Decorator
Facade
Flyweight
Proxy

Padrões Comportamentais

Chain of Responsibility
Command
Interpreter
Iterator
Mediator
Memento
Observer
State
Strategy
Template Method
Visitor

Em OO temos a herança como recurso para reaproveitar códigos e facilitar a manutenção, mas a herança em algumas linguagens como o PHP restringem-se a uma única classe. Não podemos herdar de duas ou mais classes ao mesmo tempo. Há linguagens como o C que permitem o uso de herança múltipla, porém os desenvolvedores do Delphi (pascal) optaram pelo uso de interfaces.

Interfaces nada mais são do que uma espécie de regra que a classe que a implementa deverá seguir ou ter em sua estrutura. Voltando àquela analogia básica que sempre usamos em OO, uma classe é como a planta de uma casa, algo ainda intangível, no papel e o objeto é a casa construída, algo concreto. Assim podemos aqui colocar a interface como um contrato de licitação onde estarão todas as cláusulas que ditam as regras e normas para a construção desse objeto.

Não importa qual construtora irá

construir a casa, desde que esta construtora se enquadre nas regras do contrato. Assim podemos crer que a casa será construída como foi predeterminada. Na prática podemos criar uma interface em nosso modelo que terá os métodos que julgamos necessários para determinada ação e desta maneira toda a classe que implementar esta interface terá estes métodos e os implementará de acordo com a sua necessidade.

A vantagem é que uma classe pode implementar mais de uma interface e com isso manter um relacionamento *é um* com mais de uma estrutura

MVC – Model View Controller

Separar as aplicações em camadas não é algo novo no meio de desenvolvimento de softwares. A busca por aplicações de fácil manutenção inspirou grandes mentes a desenvolver técnicas e modelos que auxiliam nesta tarefa. Porém de início quero deixar claro um fato importantíssimo: separar uma aplicação em camadas pura e simplesmente não significa que você está aplicando MVC.

A arquitetura em camadas é utilizada para separar responsabilidades em uma aplicação. Este tipo de aplicação se popularizou muito no início da década de 90 com o *boom* da internet, porém muitos desenvolvedores ainda não conhecem a técnica a fundo, devido a escassez de documentação eficiente sobre este padrão arquitetural. Junto com a disseminação

da arquitetura em camadas, veio o ressurgimento do modelo MVC criado em *Smalltalk* e que traz simplicidade e coerência à *View*.

Tanto o modelo em camadas quanto o MVC são padrões arquiteturais muito similares e que passaram a ser continuamente confundidos. MVC e desenvolvimento em camadas são abordagens diferentes que podem ser ou não aplicados em conjunto.

A abordagem do desenvolvimento em camadas visa organizar e manter os componentes separados baseados em algum critério que na maioria das vezes é a função que este componente desempenha. Separando os componentes em grupo, conseguimos diminuir o acoplamento entre eles isolando assim a mudanças que são realizadas em uma camada para que não afetem as outras. Uma aplicação Win32 desenvolvida 100% orientada a objetos poderia sem problema algum ser separada em camadas para facilitar o desenvolvimento e manutenção. Observe o diagrama da **Figura 1** e tente identificar as camadas.

Nota: O emprego correto do termo componente é utilizado para descrever um artefato de software que pode ser uma classe, objeto, camada, etc. Nós, que estamos familiarizados com a VCL do Delphi, tendemos a associar o termo componente com os controles utilizados na VCL. Estes últimos são

identificados pelo termo *controle*. Assim, estarei aqui me referindo a artefatos de software como componente.

A separação em camadas não se dá pela separação *física* das classes, mas sim pela separação *lógica* de acordo com um critério definido pelo *analista/arquiteto* de sistemas. No caso exposto na **Figura 1** as camadas (lógicas) estão divididas segundo suas responsabilidades, na **Figura 2** isso fica mais claro. Observe:

Bem se separar em camadas não é MVC então o que é? Digamos que temos uma aplicação onde aplicamos a separação em camadas de acordo com a responsabilidade de cada componente. Se estes componentes estão separados, ou melhor, desacoplados então temos que de alguma forma fazer com que se comuniquem, mas que continuem independentes em sua essência. Isso realmente seria excelente, é o famoso *cada um na sua, mas com alguma coisa em comum*. Este tipo de abordagem torna o modelo muito eficiente. Mas como manter as interfaces gráficas atualizadas refletindo o atual estado do modelo com quem ela interage?

Deu para perceber com o questionamento que entre outras coisas o modelo MVC tem a ver com sincronização entre os componentes? As interfaces gráficas geralmente exibem o estado dos objetos de uma aplicação, e isso deve ser feito em tempo real. Qualquer alteração no objeto

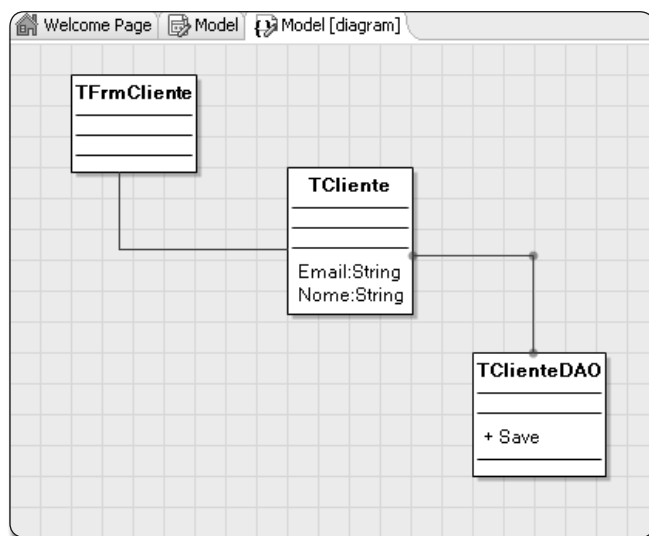


Figura 1. Modelo UML

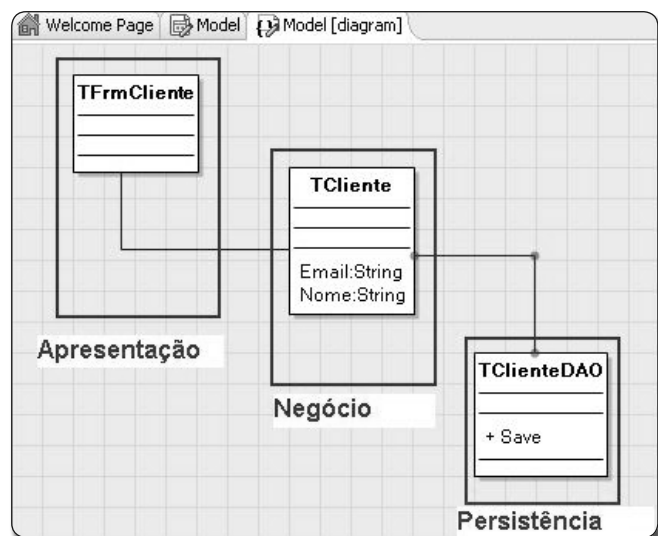


Figura 2. Modelo Separado em Camadas

deverá ser refletida na hora, na *View*.

Outro ponto é que o usuário da sua aplicação deverá interagir com o seu modelo de negócio e ele o vai fazer através da *View*. Daí, vemos importância do MVC aplicado a softwares desenvolvidos em Delphi, pois por ser RAD o Delphi nos trás uma produtividade muito grande, porém isso tem um lado ruim. Se cada controle em um *form* for responsável por invocar métodos nos objetos o código tende a ficar repetitivo e difícil de manter. Diversas regras e rotinas poderão estar contidas dentro dos botões e demais controles no formulário. Isso torna o código poluído, de difícil manutenção e altamente acoplado, pois suas regras de negócio estão dispersas entre o modelo e a *View*.

É neste cenário que entra o MVC, para além de termos as camadas com suas responsabilidades, temos os componentes da aplicação interagindo entre si em tempo real onde o modelo passa a ser ativo notificando as *views* inscritas nele. Este modelo consiste no bom uso integrado de alguns *Design Patterns* (Padrões de Projeto) clássicos, como *Observer* e *Strategy*.

Se buscarmos na internet ou em livros o significado de MVC, vamos encontrar algo como:

Model-view-controller (MVC) é um padrão de arquitetura de software. Com o aumento da complexidade das aplicações desenvolvidas torna-se fundamental a separação entre

os dados (Model) e o layout (View). Desta forma, alterações feitas no layout não afetam a manipulação de dados, e estes poderão ser reorganizados sem alterar o layout.

O model-view-controller resolve este problema através da separação das tarefas de acesso aos dados e lógica de negócio, lógica de apresentação e de interação com o utilizador, introduzindo um componente entre os dois: o Controller. MVC é usado em padrões de projeto de software, mas MVC abrange mais da arquitetura de uma aplicação do que é típico para um padrão de projeto.

No modelo em camadas é comum dividir a aplicação em: apresentação (interface), domínio (*Model*) e acesso a dados. Podemos dizer que a, grosso modo, em MVC a camada de apresentação também é separada em *View* e *Controller*.

Geralmente o fluxo de iteração entre os componentes no MVC se dá na maioria das vezes da seguinte maneira: O usuário interage com a interface de alguma forma, clicando em um botão da *View* (no nosso caso *Form*). O *Controller* é quem recebe o estímulo provocado na *View*, acessando o *Model*, e invocando um método ou atualizando de alguma forma. O *Model* por sua vez notifica a(s) *View*'s inscritas neles para receber as atualizações. A *View* recebe a notificação e se encarrega de atualizar seus controles conforme a necessidade. Os diagramas das **Figuras 3 e 4** expressam a sequência dos acontecimentos.

O estímulo vindo do usuário (ou de

outro componente se você está usando MVC fora de uma interface gráfica) vai para o *Controller* que é o componente com inteligência o suficiente para saber qual operação invocar no *Model*. A operação invocada pode efetuar a mudança de estado no *Model*. Como a *View* observa este, assim que a mudança de estado for realizada ela é atualizada.

Assim vemos que o diferencial do MVC para o modelo separado em camadas é que o primeiro foca em mais do que separar em camadas, mais cuida da iteração entre os componentes que fazem parte do modelo.

MVC no Delphi

Vamos então aplicar o conceito MVC em um pequeno exemplo feito em Delphi. Nosso exemplo focará em criar o modelo MVC. No próximo artigo daremos sequência ao exemplo adicionando a camada de persistência e concluindo assim nosso exemplo.



Nota do DevMan

Em MVC geralmente as camadas são descritas assim:
Model: classes de domínio no seu modelo de negócio. Representa o estado do sistema.

View: parte exposta, renderiza o Model em uma forma específica para interação (WebForm, Form, HTML, etc.)

Controller: Processa e responde a eventos, geralmente ações do usuário, invocando alterações no Model.

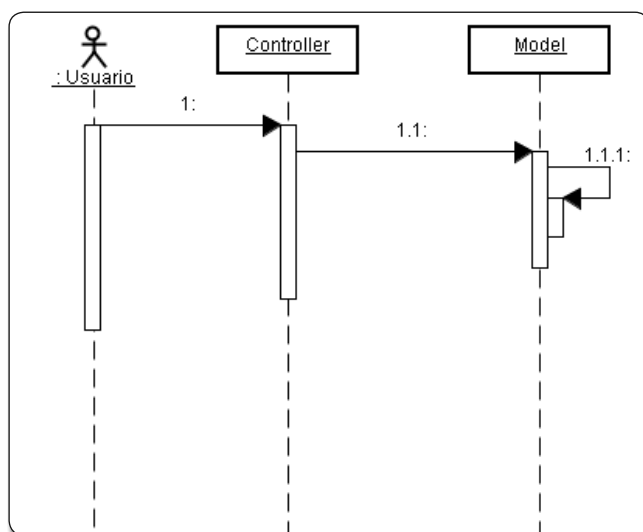


Figura 3. Usuário interage com a view e o controller repassa ao modelo

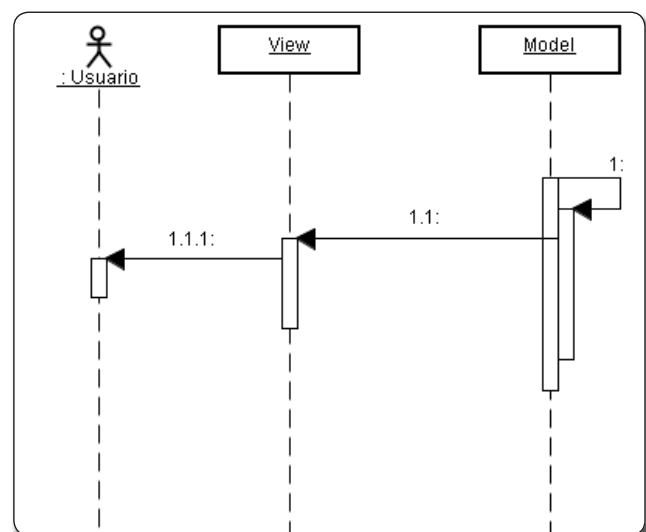


Figura 4. O modelo notifica e View que é atualizada

Como visto anteriormente, um dos desafios da arquitetura em camadas é justamente manter a camada de apresentação sincronizada com a camada de negócio. No MVC isso não é um desafio e sim parte do modelo e isso é feito aplicando o padrão de projeto *Observer*. O padrão *Observer* define uma dependência um-para-muitos entre objetos de forma que se um mudar seu estado, todos os objetos dependentes serão notificados e atualizados automaticamente. [GoF]. Este padrão é muito freqüente em projetos orientados a objetos. No Delphi este padrão está presente na notificação entre os componentes no *form designer*, quando um componente é removido ou inserido.

Em nosso exemplo em Delphi faremos uso do padrão *Observer*, então vamos a ele.

Nota: O exemplo será desenvolvido utilizando a versão 2006 do Delphi, porém com exceção da modelagem UML feita no *Together* o exemplo poderá ser construído com qualquer versão do Delphi a partir da versão 3.

No Delphi 2006 crie uma nova aplicação Win32 e salve-a em uma pasta padrão. Renomeie o formulário para *FrmMenuPrincipal.dpr*. Ainda neste projeto adicione uma nova *Unit* onde iremos criar nosso modelo padrão, ou seja, nossa

classe básica assim como as interfaces necessárias ao padrão *Observer*. Salve-a como *BaseModel.pas*. A **Figura 6** demonstra nosso modelo de classe.

As interfaces *IObserver* e *IObservable* são requeridas para o padrão *Observer*. *TBaseObject* será o objeto base para todas as outras classes presente em nossa aplicação. A interface *IController* é apenas uma sugestão que eu costumo utilizar para criar um repositório de *Controller*.

É evidente que utilizando o *Together* após fazermos a modelagem UML não há a necessidade de gerarmos o código, pois o mesmo é gerado automaticamente, porém é possível codificar manualmente para aqueles que possuem outras versões do Delphi. Veja a **Listagem 1** que contém todo o código do diagrama gerado na **Figura 6**. Assim na *Unit BaseModel.pas* digite o código da **Listagem 1**.

No código da **Listagem 1** temos a declaração das interfaces requerida pelo padrão *Observer*. A interface *IObserver* possui apenas um método (*Update*), que deverá ser implementado por toda a classe que quiser ser um observador em nosso modelo. Isso por que o *observado* quando for notificar os observadores irá invocar o método *Update* daí o porquê deste método estar presente aí. O parâmetro do tipo *IObservable* serve para que o observador possa saber por quem ele foi notificado. Observe que na linha um temos apenas um *hint* da

interface *IObservable*. Isso é necessário, pois o compilador Delphi é *Top-Down* e a interface *IObservable* é declarada abaixo da interface *IObserver* e como o método *Update* possui um parâmetro do tipo *IObservable* temos que lançar mão da técnica de *forward*. O mesmo ocorre na declaração de *TBaseObject*.

Para incluí-las basta pressionar *Ctrl + Shift + G* dentro da declaração de sua interface. A interface *IObservable* possui 3 métodos, isso porque qualquer objeto que possa ser observado em um modelo deverá ter um canal para que os observadores possam se inscrever para serem notificados (*Attach*), da mesma forma os observadores devem possuir uma maneira de se retirar da lista de notificações (*Detach*) e por fim um método para notificar os observadores que algo aconteceu. A interface *IController* como citei anteriormente é apenas uma sugestão para que possamos criar uma coleção de *Controllers* para futura manipulação.

Bem interfaces são apenas contratos, regras, declarações que precisarão ser implementados por uma classe concreta. Como no MVC nossos *Model's* são *Observadores* então nossa *BaseObject* é quem vai implementar a interface *IObservable* e nossas *views* a interface *IObserver*. Assim vamos ao código da **Listagem 2** que descreve a nossa *BaseObject* ainda na *Unit BaseModel*.

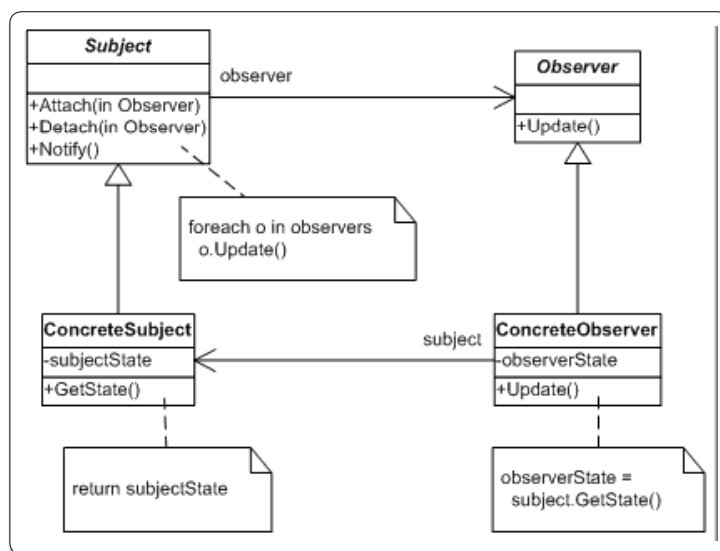


Figura 5. Modelo do padrão *Observer*

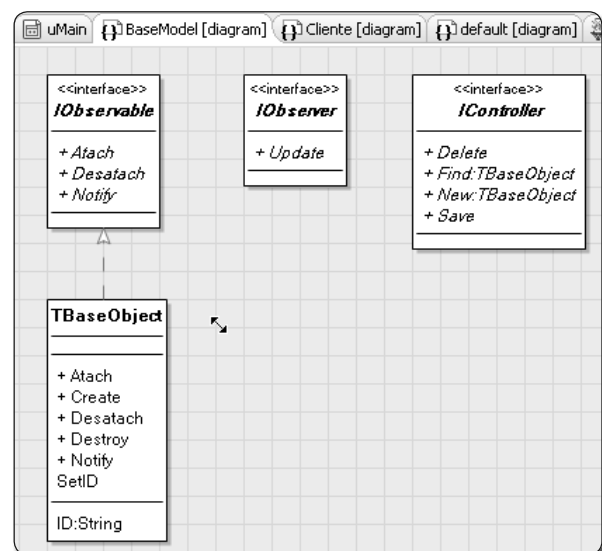


Figura 6. Modelo de classe

Na **Listagem 2** temos a declaração da classe que servirá de base para todas as demais em nosso modelo. A primeira consideração a fazer é em relação ao fato de herdarmos de *TInterfacedObject*. Isso é uma condição para todas as classes que irão implementar uma ou mais interfaces no Delphi, isso se deve ao fato desta classe já implementar 3 métodos essenciais para se trabalhar com interfaces. São eles: *QueryInterface*, *_AddRef* e *_Release*. Existe ainda outras classes que implementam estes três métodos e, portanto poderiam ser usadas aqui também, como *TContainedObject*, *TInterfaceList* e *TInterfacedPersistent*, porém a classe *TInterfacedObject* é a mais básica de todas elas.

Seguindo adiante temos na linha 3 um atributo privado *FObservers* que servirá como contêiner para todos os observadores que se inscreverem em um de nossos modelos. O campo *FID* e o método *setId* são resultado da implementação da propriedade ID. Na seção pública da nossa classe temos além do construtor e destrutor a declaração, para posterior implementação, dos métodos presentes na interface *IObservable* (*Atach*, *Desatach* e *Notify*). Vale ressaltar aqui que somente poderá ser passado como parâmetros para os dois primeiros métodos objetos produtos de uma classe que implemente a interface *IObserver*, ou seja, estamos garantindo que somente observadores se inscrevam na lista para serem notificados e se são observadores logo possuem sem dúvida nenhuma o método *update* que será invocado quando a notificação for disparada. Na **Listagem 3** podemos conferir a implementação dos 5 métodos presentes em nossa *TBaseObject*.

Na **Listagem 3** temos a implementação dos métodos outrora definidos na interface *IObservable* além do destrutor e construtor. O método *Attach* recebe como parâmetro um objeto do tipo *IObserver*, ou seja, todo objeto que implemente a interface *IObserver* poderá ser passado como parâmetro e isso é excelente, pois se uma classe que hoje não é um observador amanhã poderá ser, bastando para isso implementar a interface *IObserver*. E como uma classe pode implementar uma ou mais interfaces não corremos o

Listagem 1. Declaração das Interfaces

```
IObservable = interface;

IObserver = interface
['{65032CAD-7441-4754-A860-BFECE58D50EF}']
procedure Update(Observable: IObservable);
end;

IObservable = interface
['{637D22E1-45BD-479E-96D3-39F6DD94234B}']
procedure Attach(Observer: IObserver);
procedure Detatch(Observer: IObserver);
procedure Notify;
end;

TBaseObject = class;

IController = interface
['{2C5753F9-473F-4D22-9FEA-28E2B84C629E}']
procedure Save;
function Find(ID: String): TBaseObject;
function New: TBaseObject;
end;
```

Listagem 2. Declaração da Classe BaseObject

```
TBaseObject = class(TInterfacedObject, IObservable)
private
  FObservers: TInterfaceList;
  FID: String;
  procedure SetID(const Value: String);
protected

public
  constructor Create;
  destructor Destroy; override;
  procedure Attach(Observer: IObserver);
  procedure Detatch(Observer: IObserver);
  procedure Notify;
published
  property ID: String read FID write SetID;
end;
```

Listagem 3. Implementação dos Métodos

```
procedure TBaseObject.Attach(Observer: IObserver);
begin
  FObservers.Add(Observer);
end;

constructor TBaseObject.Create;
var
  GUID: TGUID;
begin
  FObservers := TInterfaceList.Create;
  if CoCreateGuid(GUID) = S_OK then
    FID := GUIDToString(GUID);
end;

procedure TBaseObject.Detatch(Observer: IObserver);
begin
  FObservers.Remove(Observer);
end;

destructor TBaseObject.Destroy;
begin
  FreeAndNil(FObservers);
  inherited;
end;

procedure TBaseObject.Notify;
var
  Obs: IInterface;
begin
  for Obs in FObservers do
    IObserver(Obs).Update(Self);
  end;
```

Listagem 4. Declaração da Classe Cliente

```
uses BaseModel;  
  
type  
  TCliente = class(TBaseObject)  
  private  
  
    protected  
  
    public  
      procedure Salvar;  
      function Buscar(Codigo: String): TCliente;  
  published  
    property Nome: String;  
    property Email: String;  
    property Telefone: String;  
  end;  
end;
```

Listagem 5. Classe ClienteController

```
TClienteController = class(TInterfacedObject, IController)  
private  
  FModel: TCliente;  
  class var FInstance: TClienteController;  
  constructor PrivateCreate;  
protected  
  
public  
  class function GetInstance: TClienteController;  
  constructor Create;  
  destructor Destroy; override;  
  procedure Save;  
  function Find(ID: String): TBaseObject;  
  function New: TBaseObject;  
published  
  property Model: TCliente read FModel;  
end;  
  
implementation  
  
uses  
  SysUtils;  
  
constructor TClienteController.PrivateCreate;  
begin  
  inherited Create;  
end;  
  
constructor TClienteController.Create;  
begin  
  raise Exception.Create('Para Obter um ClienteController Invoque o Metodo  
  GetInstance');  
end;  
  
destructor TClienteController.Destroy;  
begin  
  inherited;  
end;  
  
function TClienteController.Find(ID:String): TBaseObject;  
begin  
  Result := FModel.Buscar(ID);  
end;  
  
class function TClienteController.GetInstance: TClienteController;  
begin  
  if not Assigned(FInstance) then  
    FInstance := TClienteController.PrivateCreate;  
  Result := FInstance;  
end;  
  
function TClienteController.New: TBaseObject;  
begin  
  FModel := TCliente.Create;  
  Result := FModel;  
end;  
  
procedure TClienteController.Save;  
begin  
  FModel.Salvar;  
end;  
  
end.
```

risco dela já estar implementando uma interface como ocorre no caso da herança. Veja na terceira linha que apenas adicionamos o objeto passado como na lista de observadores.

Na sequência temos a implementação do construtor da nossa classe. Nele instanciamos a classe *TInterfaceList* que irá funcionar como repositório de *observadores* e em seguida geramos uma nova *GUID* e associamos a propriedade *ID* do nosso objeto, assim garantimos um identificador único para cada objeto que for criado em nosso modelo de dados.

O código do método *Desattach* apenas remove o objeto referido como parâmetro da lista de notificações, assim este objeto não será mais notificado. No *destructor* apenas destruimos o objeto *TInterfaceList* para, por fim, chegarmos ao tão aguardado método *Notify*. Este será o método chamado toda vez que o estado no objeto em questão for alterado, ou seja, ao menor sinal de iteração com o objeto o método *Notify* deverá ser invocado e tem um motivo óbvio para isso.

Observe na *Listagem 3* que nosso código começa fazendo um loop com o método *for in* em nossa lista de observadores e para cada item da nossa lista invocamos o método *update* e passamos o próprio objeto como parâmetro para que o observador saiba por quem ele foi notificado. Assim cabe agora ao observador, que no MVC será a *View*, tomar a providência necessária para atualizar os dados do modelo na *View*. E é isso que veremos mais a frente.

Com nossa classe base pronta vamos a criação da nossa classe de negócio que será a classe *TCliente* que herdará de nossa *BaseObject* já “nascendo” com a possibilidade de ser observada por um observador. Assim ainda em nosso pro-



Nota do DevMan

TGUID é um record no Delphi utilizado para representar um GUID (Global Unique Identifier). A declaração de uma interface pode conter um GUID que é declarado como uma cadeia de caracteres entre colchetes da seguinte forma: `{'xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxx'}` onde cada x é um dígito hexadecimal 0 – F. Você pode gerar suas GUID's no Delphi pressionando Ctrl + Shift + G no Code Editor.

jeto no Delphi adicione mais uma *Unit*, salve-a como *uCliente.pas* e nela declare o código da **Listagem 4**.

O código da **Listagem 4** mostra a nossa classe cliente ainda sem a implementação dos métodos e *property*. Para gerar a implementação basta pressionar *Ctrl + Shift + C*. O código gerado nada mais é do que a implementação das *property* o que dispensa comentário e mais a implementação dos métodos *Salvar*, *Deletar* e *Buscar*. Esses métodos são apenas ilustrativos e estão aí para demonstrar a mudança de estado do objeto, e como estamos falando sobre MVC a cada alteração no estado do objeto os observadores, nesse caso serão as *views*, deverão ser notificadas.

O método salvar, quando chamado, deverá persistir o objeto em questão no banco de dados. Não é escopo deste artigo tratar de persistência, sendo assim além de invocar o método *notify* na implementação do método salvar você invocaria sua classe de persistência e solicitaria a ela que persistisse seu objeto. Já o método *Buscar* servirá para carregar na classe informações de um objeto específico. Como em OO temos o conceito de *Object ID*, ou seja, cada objeto em seu modelo deverá possuir um identificador único, passamos apenas o ID do objeto para que possa ser carregado no mesmo os dados referente ao objeto possuidor do ID. Como dito acima, não teremos aqui a implementação do método como um todo, pois isso também seria função da classe de persistência, porém o método *notify* será invocado para notificar os observadores que os dados do modelo foram alterados. Sendo assim na implementação deste dois métodos apenas invoque o método *notify*.

Com isso concluímos nosso *Model*. O mesmo já pode ser “persistido” e notificado aos observadores caso seu estado seja alterado. Passemos então a criação da nossa classe de controle.

Controller

Ainda neste mesmo projeto, adicione uma nova *Unit* e salve-a como *ClienteController.pas*. É nesta *Unit* que iremos criar a classe de controle para iterar com nosso modelo. Assim, esta classe de controle esta apresentada na **Listagem 5**.

Se você atentou a definição do padrão MVC e a imagem da **Figura 3** irá perceber que a função do *Controller* é exatamente mapear as ações do usuário na *View* e ter inteligência suficiente para saber qual método invocar no modelo. Por isso é muito comum que você encontre no *controller* os métodos de *CRUD*, para que o usuário através da *View* possa invocar métodos para manipular um determinado *Model*. Assim começamos a declaração do nosso *Controller* implementando a interface *IController* e desta maneira temos em *Controller* os métodos presentes na interface *IController*. Já podemos então permitir a *View*, através do *Controller*, criar um novo Objeto, Buscar um Objeto e salvar alterações no modelo. Porém vamos atentar para o fato de, se o *Controller* tem a função que tem, não justifica ter num projeto dois *ClienteController* se o usuário só poderá manipular um cliente por vez (pelo menos essa é a minha abordagem).

Assim vamos aplicar a nossa classe *ClienteController* o padrão de projeto *Singleton*. Temos os métodos necessários para poder implementar o *Singleton*. Uma variável de classe para guardar a instância quando criada, um método de classe (*GetInstance*) para retornar uma instância de *TClienteController* e um construtor privado para construir nosso objeto.

Esse construtor é privado, pois o construtor público foi invalidado, observe na linha 26. Se alguém tentar instanciar um objeto do tipo *TClienteController* irá receber um erro. Desta forma garantimos que só teremos um único objeto instanciado em nossa aplicação, e como eu posso garantir isso? Simples. Como a única maneira de instanciar um Objeto *TClienteController* é através do método *GetInstance*, então é nele que fazemos a verificação para saber se já não há um objeto instanciado. Caso não haja um objeto instanciado então criamos um e o armazenamos na variável *FInstance*. Da próxima vez que o método *GetInstance* for chamado não será criado um novo objeto apenas retornado o objeto que já existe.

Há apenas uma ressalva aqui. Como disse no início do artigo que este exem-

plo poderia ser feito com versões do Delphi posteriores a 3, então aqueles que estiverem com versões anteriores a 2006 deverão substituir a *class var* por uma variável global, pois versões anteriores a 2006 não suportam *class var*.

Continuando no código da **Listagem 5** temos os métodos que mapearão a ação do usuário na *View* para o *Model*. Assim fica simples entender o que será feito na implementação de cada método. Mais abaixo é mostrado a implementação do método *Save*, ele invoca o método salvar no modelo, que se encarrega que “persistir” os dados (neste artigo não abordamos isso) e notificar a *View*. Assim quando o usuário clicar no botão salvar o modelo mudará de estado e a *View* receberá a notificação e se atualizará. Isso será feito adiante.

Em seguida faz a mesma coisa porém com o método *Buscar*. Observe que sempre fazemos a chamada de *FModel* que é a propriedade do próprio *Controller*. porém uma propriedade somente leitura. Isso para garantir que ninguém ira passar um *Model* para o *Controller* garantindo assim que a única maneira de um *Model* ser associado ao *Controller* e através do método *New*, e é isto que mostra nas seguintes.

Cliente View

Vamos adicionar ao nosso projeto mais um formulário. Salve-o como *uFrmCliente.pas* e renomeie-o para *FrmCliente*. Lembrando, a função da *View* é permitir que o usuário interaja com o modelo com suas ações sendo mapeadas pelo *Controller*. Sendo assim temos que ter em nossa *View* controles visuais para podermos valorar nosso *Model*.

O principal será feito agora. Como no padrão MVC a *View* observa o modelo



Nota do DevMan

O padrão de projeto *Singleton* assegura que haja somente uma instância de uma classe, e fornece um ponto de acesso global para acessá-la [GoF]. Utilizado em situações onde queremos garantir que só haverá um e somente um objeto deste tipo na aplicação. Isto é feito invalidando o construtor da classe e criando um método de classe que retorne uma instancia deste objeto.

Listagem 6. Implementação da interface *IObserver*

```
TFrmCliente = class(TForm, IObserver)
private
    FController: TClienteController;
public
    constructor Create(AOwner: TComponent; Controller: TClienteController); reintroduce;
    procedure Update(Observable: IObservable);
end;
```

Listagem 7. Implementação dos Métodos da *View*

```
constructor TFrmCliente.Create(AOwner: TComponent;
    Controller: TClienteController);
begin
    inherited Create(AOwner);
    FController := Controller;
end;
procedure TFrmCliente.Update(Observable: IObservable);
begin
    EdtID.Text      := FController.Model.ID;
    EdtNome.Text    := FController.Model.Nome;
    EdtEmail.Text   := FController.Model.Email;
    EdtTelefone.Text := FController.Model.Telefone;
end;
```

Listagem 8. Implementação dos métodos

```
procedure TFrmCliente.BtnNovoClick(Sender: TObject);
begin
    FController.New;
    FController.Model.Atach(Self);
end;
procedure TFrmCliente.BtnPesquisarClick(Sender: TObject);
begin
    FController.Model.Buscar('');
end;
procedure TFrmCliente.BtnSalvarClick(Sender: TObject);
begin
    FController.Model.Nome := EdtNome.Text;
    FController.Model.Email := EdtEmail.Text;
    FController.Model.Telefone := EdtTelefone.Text;
    FController.Save;
end;
```

Listagem 9. Chamando a *View* do Cliente

```
procedure TForm1.BtnClienteClick(Sender: TObject);
var
    C: TClienteController;
begin
    C := TClienteController.GetInstance;
    FrmCliente := TFrmCliente.Create(Self, C);
    FrmCliente.Show;
end;
```

então temos que transformar nossa *View* num *observador* e faremos isso implementando a interface *IObserver* na nossa classe *TFrmCliente*. Desta forma o formulário de cliente é obrigado a implementar o método *Update*, que é extremamente importante, pois é este método que será responsável por atualizar a *View*. Observe a **Listagem 6**, ela mostra a como fica a declaração da classe cliente com a interface implementada e os demais métodos requeridos.

Em primeiro lugar é notório a presença do método *update*, isso porque implementamos a interface *IObserver*. Com isso nosso *form*, ou melhor, *View* tem que poder ser atualizado com o método *update*. Lembre-se que em nosso *Model* a cada mudança de estado invocamos o método

Notify e este por sua vez faz um loop na lista de observadores e chama para cada um o método *update*. Ainda no código da **Listagem 6** temos uma atributo do tipo *TClienteController*, isso para podermos guardar a referência do *Controller* que a *View* está se comunicando. Toda *View* tem suas ações, ou melhor, ações do usuário mapeadas para um *Controller* daí o motivo de ainda na **Listagem 6** nos reescrevermos o construtor do nosso formulário. Observe que o construtor agora pede dois parâmetros: o *owner* que já é padrão para todo componente e agora também pede um *Controller*, assim garantiremos que ao criar uma *View* já teremos um *Controller* associado a ela. E para que isto?

Tenha sempre em mente que as ações

do usuário mapeadas pela *View* devem ser enviadas ao *Model* através do *Controller*, por isso esta condição de só criarmos uma *View* com um *Controller* previamente criado. A implementação dos métodos explica-se por si só. Observe a **Listagem 7**.

Veja que no construtor da nossa *View* nós invocamos o construtor de *TForm* e passamos para ele o *Owner* passado como parâmetro. O segundo parâmetro, o *Controller*, nós guardamos no *Field FController*, com isso garantimos que esta *View* o será criada se um *Controller* for passado como parâmetro. Na sequência vemos a implementação do método *Update*. Observe que a *View* ao ser notificada atualiza os controles da *View* repassando para os mesmos os novos dados do modelo. Note bem que os valores são obtido via *Controller*, porém também poderia ser feito fazendo um *typecasting* de *IObservable* para *TCliente*.

Para concluir nossa *View* vamos a implementação dos eventos clique dos botões novo, pesquisar e salvar. A **Listagem 8** se encarrega de exibir os métodos.

Como dito anteriormente o *Controller* mapeia as ações executadas na *View* para o *Model*. E a implementação dos métodos na **Listagem 8** ilustra bem isto. O clique do botão novo invoca o método *new* do *Controller* e este por sua vez cria um novo objeto *Cliente* e o armazena internamente na propriedade *Model*. Feito isso o próximo passo é inscrever esta *View* onde estamos neste novo modelo para que a mesma possa receber as notificações necessárias ao longo do ciclo de vida o objeto. O botão pesquisar apenas invoca o método *Buscar* do modelo que (se tivesse implementação com DAO) configuraria as propriedades do Objeto *Cliente* com os dados oriundos do banco de dados.

Por fim temos o código do clique do botão salvar. Este por sua vez pega os valores dos *edits*, atribui as propriedades do *Model* que esta sendo controlado pelo *Controller* no momento e invoca o método *Save*. Este método é mapeado para o *Model* que se encarrega de persistir o objeto e notificar os Observadores.

Com isto temos nosso MVC implemen-

tado e pronto para uso. Vamos, em caráter de teste, invocar nossa *View* e realizar os testes. Assim no *form* principal da aplicação coloque um botão para invocar a *View Cliente* e no evento *OnClick* insira o código da **Listagem 9**.

No clique do botão que invocamos a *View* do cliente nós declaramos uma variável do tipo *TClienteController* e *Pedimos* um *Controller* para ela, lembrando que o *ClienteController* é um *Singleton* e como tal só termos um instanciado por vez em nossa aplicação. Em seguida criamos nossa *View* do cliente e repare que ao chamar o construtor ele nos pede além do *owner* um *Controller*, e é aí que passamos a variável *C* com o *Controller* criado acima. Feito isso exibimos a *View* com o já conhecido método *Show*. Rode a aplicação, clique no botão novo e em seguida no botão salvar (**Figura 7**).

Veja que nosso MVC já está em pleno funcionamento. Ao clicar em novo o *Controller* criou um novo objeto e neste momento já temos um *GUID* associado a ele, porém como não houve notificação a *View* não está sabendo. Quando chamamos o método salvar a notificação é feita a todos os observadores e como nossa *View* está inscrita na lista a mesma é notificada e os dados atualizados na *View*. É óbvio que uma aplicação *Stand Alone* como esta pode não mostrar todo o potencial deste padrão arquitetural, mas experimente chamar outro formulário e clique em novo. Depois altere os dados no primeiro *Form* e salve. Você irá perceber que o segundo *Form* também será atualizado com os dados do primeiro. A **Figura 8** mostra o exemplo citado em funcionamento.

Conclusão

O padrão MVC é muito mais comum na WEB, isso porque fica fácil criar um repositório de *Controllers*, que ficaria na sessão e assim conseguiria notificar todas as *Views*. Porém nada nos impede de aplicarmos em aplicações Win32. Porém há de se ficar claro que não justifica aplicar o padrão MVC em uma aplicação *Stand Alone* como fizemos aqui. Mas imagine este padrão aplicado numa Aplicação que tenha um servidor de objetos, COM+ por exemplo. Poderíamos ali criar

nosso repositório de objetos e um *Controller Singleton* para mapear as ações dos usuários. O importante é deixar claro que este é um padrão arquitetural que pode ser, e deve ser aplicado junto com outros padrões de projeto, pois como os outros se trata também de uma boa prática de programação.

Eu fico por aqui, espero que tenha contribuído com mais este artigo para o nosso aprendizado. Até a persistência. ●

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



Figura 7. Novo cliente criado e salvo

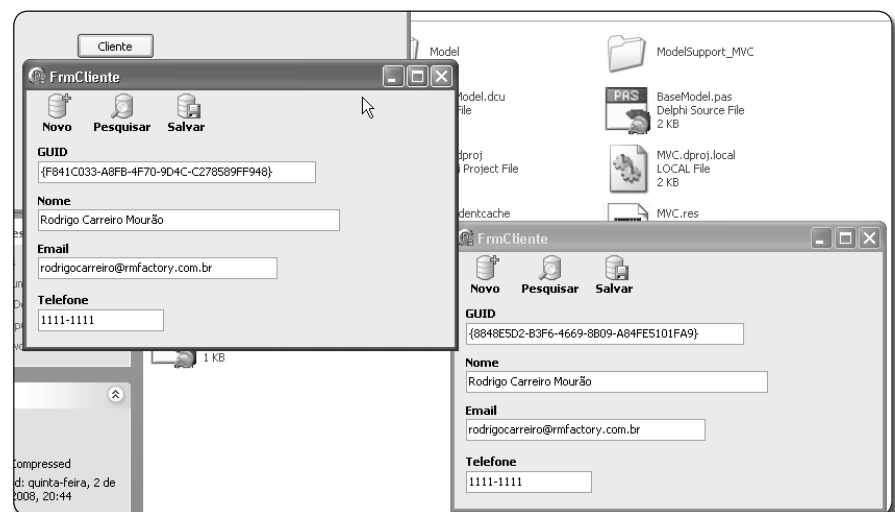


Figura 9. Duas View sendo atualizadas ao mesmo tempo

