MVP: Model View Presenter - Revista .net Magazine 100

Este artigo trata sobre o pattern Model View Presenter, que apesar de não ser tão popular quanto o MVC (Model View Controller), pode ser implementado para prover uma separação da camada de visualização das outras camadas de sua aplicação.

De que se trata o artigo

Este artigo trata sobre o pattern Model View Presenter, que apesar de não ser tão popular quanto o MVC (Model View Controller), pode ser implementado para prover uma separação da camada de visualização das outras camadas de sua aplicação.

Em que situação o tema é útil

A implementação do MVP pode ser útil em situações onde você deseja ter uma clara separação das camadas lógicas do teu sistema, separando a visualização das outras camadas. Além disso, este padrão pode ajudar a aumentar o reuso do código, quando existe a necessidade de implementação de um mesmo recurso em tecnologias diferentes (Windows Forms, Web Forms etc..), sem contar que dá mais liberdade para as equipes de designer e desenvolvimento trabalharem de forma mais independente.

MVP - Model View Presenter

Neste artigo exploraremos mais a fundo os conceitos por traz do MVP, suas características e suas formas de implementação, através das abordagens Passive View e Supervising Presenter. Faremos ainda um exemplo de implementação do MVP, simulando uma tela de cadastro de usuários utilizando Windows Forms e Web Forms com o intuito de deixar claro alguns dos

benefícios do uso do MVP.

Diariamente no desenvolvimento de software nos deparamos com problemas corriqueiros em diversos projetos diferentes. Isso acontece nas equipes de desenvolvimento do mundo inteiro, e de todas as linguagens e tecnologias. Foi baseado na repetição destes problemas, nos mais variados cenários, que foram criados os Design Patterns. Os padrões de projeto são soluções comuns para problemas corriqueiros inerentes ao desenvolvimento de softwares orientados a objetos.

Os padrões de projeto são independentes de tecnologia e visam fornecer um caminho para solução de um problema, sem se prender à tecnologia ou linguagem que você está utilizando. Sendo assim, podemos implementar o mesmo padrão de projeto em .Net, Java, Delphi, em uma aplicação Web, desktop ou Mobile, respeitando apenas as características de cada linguagem e ambiente. Os padrões de projetos se dividem em três tipos:

• Estruturais – Padrões que visam descrever como classes e objetos devem ser combinados para formar estruturas maiores. Ex.: Composite.

• Criacionais – Padrões que visam resolver problemas inerentes à criação de objetos. Ex.: Singleton

• Comportamentais – Padrões que visam descrever como os objetos devem se comportar e relacionar uns com os outros. Ex.: Observer

Além dos padrões de projeto, nós temos ainda o conceito de Architectutral Patterns (Padrões de arquitetura). Este conceito é parecido com o de padrões de projeto, sendo mais amplo. Enquanto que um padrão de projeto é aplicado dentro de um contexto específico dentro de uma aplicação, os padrões de arquitetura se referem a como a aplicação em si irá se comportar e quais serão os padrões de comunicação da mesma, tanto entre suas camadas físicas e lógicas, quanto para com outras aplicações.

Existe certa controvérsia na comunidade com relação à classificação do MVP (Model View Presenter). Esta mesma controvérsia acontece com o MVC (Model View Controller). Alguns classificam como padrão de projeto e outros classificam como padrão de arquitetura. Acreditamos que é um padrão de arquitetura, que visa estabelecer a separação e uma forma de comunicação entre as camadas lógicas da sua aplicação,mas não seria de tudo errado em considerar como um padrão de projeto, se olharmos na ótica da descrição da solução de um problema.

MVP – Model View Presenter

O MVP é um padrão de arquitetura que visa a separação das camadas lógicas da aplicação em três elementos:

• Model – Camada de dados, com suas classes de domínio e regras de negócio;

• View – Camada de visualização, contendo todos os elementos de interface gráfica e toda a

interação com o usuário final;

• Presenter – Camada de apresentação de dados, responsável pela comunicação da view com os comportamentos e dados do model.

Com a adoção do MVP, conseguimos resolver dois problemas comuns na camada de visualização:

• **Testabilidade** – Automatizar testes em rotinas que dependem de operações da interface gráfica é muito mais complexo do que testar uma classe C#. Cada tecnologia de apresentação possui suas particularidades, os eventos foram preparados para serem disparados por ações do usuário e normalmente acaba-se por inserir complexidade demais na camada de visualização, tornando a automação dos testes uma tarefa mais complexa. Com o uso do MVP, nós conseguimos obter um nível maior de testabilidade no projeto, pois conseguimos abstrair a tecnologia de visualização dos testes. Com um simples mock (**Nota do DevMan 1**), nós conseguimos testar nosso presenter e suas operações, independentemente da tecnologia de visualização que estiver sendo usada.

• **Portabilidade** – Existem cenários em que precisamos portar determinadas telas e recursos em mais de uma tecnologia, por exemplo, Web e Desktop. Nestes casos, sem a adoção de um pattern como o MVP, normalmente pode-se ter repetição de código e um menor nível de reuso.

Nota do DevMan 1

Objetos mocks são objetos que simulam um determinado comportamento de um objeto, utilizados em testes unitários. No caso do MVP, poderíamos cria um mock da view, implementando a interface da mesma, para testarmos nosso presenter sem depender da implementação concreta da view (Windows forms, web forms etc...). Dessa forma, deixamos nosso teste mais independente da view, onde é sempre mais difícil se automatizar testes.

Teste unitário é uma prática de testes onde se testa cada uma das menores unidades da sua aplicação, ou seja, os métodos. Para cada método, teríamos um código para testar o mesmo, simulando as possíveis entradas e validando as saídas esperadas. A ideia do teste unitário é garantir que cada parte do seu código, faça o seu papel, evitando resultados inesperados em tempo de execução.

O MVP prevê a criação de uma interface entre o presenter e a view, abstraindo o presenter da tecnologia de visualização empregada no projeto. Sendo assim, todas as views concretas (que podem estar nas mais diversas tecnologias), tem que implementar uma determinada interface, que contém os comportamentos que o presenter precisa para se comunicar com as views.

Existem dois tipos de implementação do MVP, sendo eles:

• Passive View – Nesta implementação, o presenter é o responsável pelo binding dos dados do modelo na view e vice versa, deixando a view mais independente do modelo e com o código mais simples, como mostra a **Figura 1**.

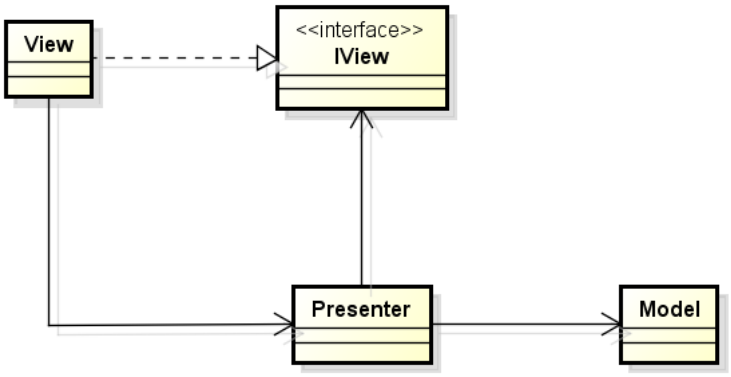

Figura 1. MVP – Passive View

• Supervising Presenter – Nesta implementação, a própria view é responsável pelo binding dos dados do modelo, fazendo com que ela tenha um pouco mais de código, como na **Figura 2**.

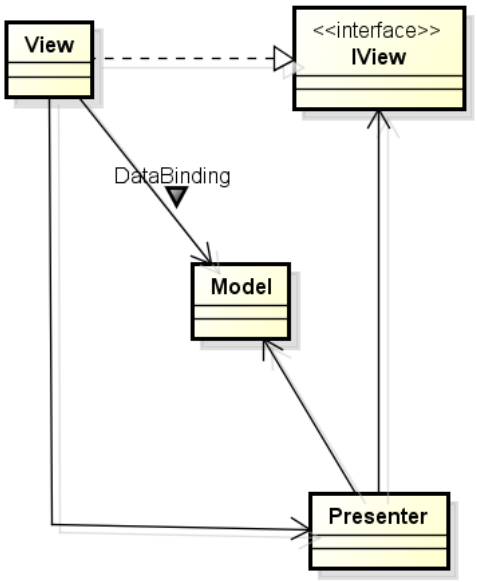

Figura 2. MVP – Supervising Presenter

Em ambos os casos, a view faz chamadas ao presenter para executar as operações desejadas. A diferença chave é no binding do modelo. No Passive View, a view disponibiliza métodos de acesso a cada um dos dados para que o presenter leia os mesmos e faça o databind para o modelo, enquanto que no Supervising Presenter, a view já faz este databinding e passa o modelo, já populado, para o presenter realizar suas tarefas.

Em ambos os casos temos um bom nível de testabilidade, pois não temos regras na view. Porém, o benefício da abordagem Passive View é que a view fica com menos responsabilidades, deixando o bind por conta do presenter, o que torna menor o risco de termos algum problema de testes na view, pois seria mais simples criarmos um objeto mock para testarmos nosso presenter, obtendo maior cobertura de testes. Por outro lado, Supervising Presenter se torna vantajoso quando se usa alguma tecnologia ou framework que possibilite o mapeamento view/model para realização do binding automaticamente, nos fazendo escrever menos código.

Com abstração da tecnologia de visualização utilizada pela aplicação, nós conseguimos também aumentar o reuso de nosso código e reduzir o impacto de uma possível mudança de tecnologia de visualização no futuro. Se precisarmos, por exemplo, implementar um cadastro tanto em Windows Forms quanto em Web, teremos apenas que ter duas Views, implementando a mesma interface e consumindo o mesmo presenter, como ilustrado na **Figura 3**.

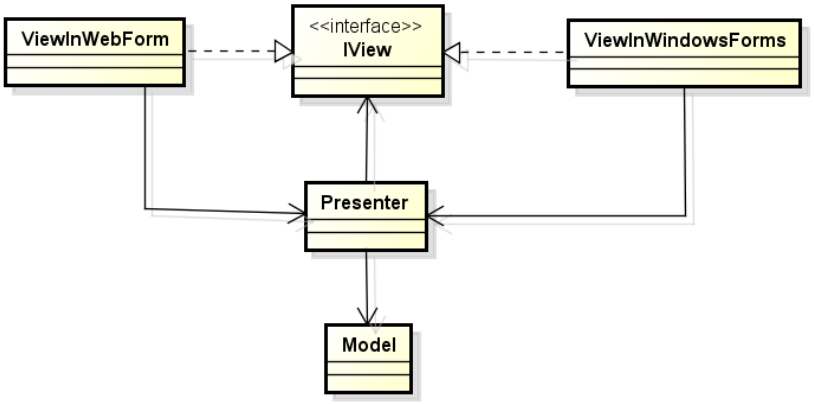

Figura 3. MVP – Duas views em tecnologias diferentes.

Outro benefício interessante na adoção do MVP é com relação à equipe. Com o uso do MVP nós conseguimos fazer com que o desenvolvedor e o designer trabalhem em paralelo de uma maneira mais independente um do outro, pois o desenvolvedor irá desenvolver o presenter baseado na interface que o designer irá implementar.

Criando os projetos

Em nosso exemplo, criaremos uma tela de cadastro de usuários, em Windows forms e depois iremos implementar a mesma tela em WebForm, ambos usando Visual Studio 2010, usando tanto a abordagem Passive View quando a abordagem Supervising Presenter do MVP.

Para nosso exemplo, criaremos quatro assemblies, sendo eles:

• DevMedia.Net.Exemplo.Model – Neste assembly teremos as classes de nosso domínio, que nosso caso se limitará a usuário.

• DevMedia.Net.Exemplo.MVP – Neste assembly teremos as classes de nosso presenter e as interfaces de nossas views.

• DevMedia.Net.Exemplo.WinForms – Neste assembly teremos os formulários desktop que irão implementar as interfaces de nossas views .

• DevMedia.Net.Exemplo.WebForms – Neste assembly teremos os formulários web que também irão implementar as interfaces de nossas views .

Crie um novo projeto, do tipo class library no visual Studio, e nomeie o mesmo como DevMedia.Net.Exemplo.MVP.

Repita este procedimento para o model, nomeando o mesmo como DevMedia.Net.Exemplo.Model. Por enquanto ficaremos com estes dois projetos e mais a frente criaremos os projetos de nossas interfaces gráficas.

Remova os arquivos Class1.cs gerados automaticamente pelo visual studio, de forma que sua solution fique conforme mostrado na **Figura 4.**

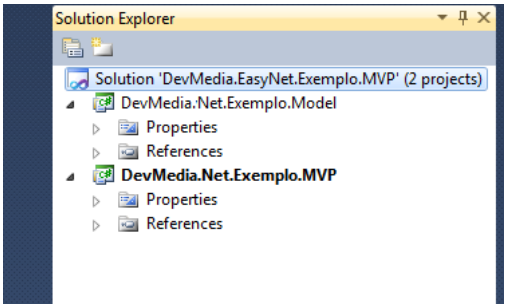

Figura 4. Estrutura inicial da solution

> **Nota:** Daqui para frente iremos nos referir aos projetos apenas pelo seu sufixo final, por exemplo: quando citar que vamos criar algo no projeto model, estaremos nos referindo ao projeto DevMedia.Net.Exemplo.Model. Dessa forma a leitura do artigo vai ficar menos poluída.

Implementando as classes do model

Antes de criarmos nossas classes do modelo, apenas por questão de organização, vamos criar uma pasta no projeto model, chamada Entities. O objetivo é puramente organizacional, para que não tenhamos todas as classes de nosso modelo diretamente na raiz do projeto, visto que no futuro podemos ter outros tipos de classe no projeto.

Agora adicione uma nova classe à pasta Entities e nomeie-a como “Usuario”, implementando o código da **Listagem 1.**

Listagem 1. Classe Usuario

```
01      using System;
02      using System.Collections.Generic;
03      using System.Linq;
04      using System.Text;

04      namespace DevMedia.Net.Exemplo.Model.Entities
05      {
60          public class Usuario
07          {
08            public long Id { get; set; }
09
10            public string Nome { get; set; }
11
12            public string Login { get; set; }
13
14             public string Senha { get; set; }
15          }
16      }
```

Observe na **Listagem 1** o namespace da classe DevMedia.Net.Exemplo.Model.Entities. O visual studio automaticamente cria a classe no namespace default do projeto concatenado com o diretório da classe. Por padrão, o namespace default tem o mesmo nome do projeto, porém, caso queira alterar isso, basta clicar com o botão direito do mouse sobre o projeto, opção properties, como mostrado na **Figura 5**.

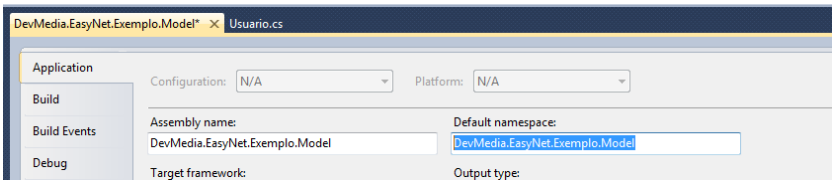

Figura 5. Properties do projeto

Implementando as classes do MVP

Seguindo a mesma premissa de organização, vamos criar dois diretórios no nosso projeto MVP:

• Presenters - Aqui ficarão os nossos presenters, responsáveis pela comunicação da View com o Model;

• Interfaces – Aqui ficarão as interfaces que nossas views (Windows Forms e WebForms) irão implementar.

Nossas classes do MVP precisarão conhecer nossa classe Cliente para fazer referência à mesma. Para isso, precisamos adicionar a referência ao projeto model no projeto MVP. Para isso, clique com o botão direito em references => add reference. Selecione a aba projects, marque o projeto model e clique em OK.

Agora nosso projeto MVP já poderá acessar as classes do projeto model. Vamos então criar nossas interfaces das views. Nosso cadastro de usuário terá uma grid com os usuários cadastrados e os campos para edição e inclusão de novos usuários. Então nosso formulário terá que estar apto a executar todo o CRUD de usuário.

Pensando neste cenário, nossas views terão que ser capazes de:

• Retornar os dados digitados pelo usuário na tela, para que o presenter possa persistir os mesmos.

• Renderizar uma lista de todos os usuários cadastrados em uma grid.

• Receber os valores de um determinado usuário para possibilitar a edição do mesmo.

• Limpar todos os controles da tela.

Sendo assim, nossas interfaces terão que possibilitar estas operações. Como o artigo tem apenas fins didáticos, criaremos duas interfaces para o cadastro de clientes, sendo uma com a abordagem Passive View e a outra com a abordagem Supervising Presenter. Em um projeto real nós teríamos apenas uma das duas abordagens.

Adicione uma nova interface na pasta Interfaces do projeto MVP e nomei-a como IUsuarioCadastroPassive, implemente-a de forma que fique como na **Listagem 2.**

Listagem 2. Interface IClienteCadastroPassive

```
01    using System;
02    using System.Collections.Generic;
03    using System.Linq;
04    using System.Text;
05    using DevMedia.Net.Exemplo.Model.Entities;
06
07    namespace DevMedia.Net.Exemplo.MVP.Interfaces
08    {
09        public interface IUsuarioCadastroPassive
10        {
11            long GetIdUsuarioSelecionado();
12
13            string GetNome();
14
15            string GetLogin();
16
17            string GetSenha();
18
19            void SetIdUsuarioSelecionado(long value);
20
21            void SetNome(string value);
22
23            void SetLogin(string value);
24
25            void SetSenha(string value);
26
27            void SetListaUsuariosCadastrados(IList<Usuario> usuarios);
28
29            void LimparControles();
30        }
31    }
```

Como podemos ver na **Listagem 2**, temos a passagem de todos os dados do usuário para a view, assim como a recuperação dos mesmos. Cada view que implementar esta interface terá que obrigatoriamente implementar estes métodos. Repita o procedimento de criação da interface, porém agora nomeie a mesma como IUsuarioCadastroSupervising e implemente-a de forma que fique como na **Listagem 3**.

Listagem 3. Interface IClienteCadastroSupervising

```
01    using System;
02    using System.Collections.Generic;
03    using System.Linq;
04    using System.Text;
05    using DevMedia.Net.Exemplo.Model.Entities;
06
07    namespace DevMedia.Net.Exemplo.MVP.Interfaces
08    {
09        public interface IUsuarioCadastroSupervising
10        {
11            Usuario GetUsuario();
12
13            void SetUsuario(Usuario usuario);
14
15            void SetListaUsuariosCadastrados(IList<Usuario> usuarios);
16
17            void LimparControles();
```

```
18          }
19      }
```

Observe que na **Listagem 2 e 3** nós precisamos adicionar a referência ao namespace using DevMedia.EasyNet.Exemplo.Model.Entities para acessarmos a classe Usuario. Comparando o código dessas duas listagens conseguimos visualizar a diferença entre as duas abordagens. Na primeira (Passive View), o presenter terá que passar/recuperar o valor de cada atributo de usuário para a view, enquanto que na segunda (Supervising Presenter), ele irá passar/recuperar apenas um objeto usuário, sendo a view responsável por popular este objeto.

Uma vez implementadas as interfaces de nossas views, teríamos agora que implementar nossos presenters. Basicamente, nosso presenter terá que ter os métodos a serem acionados pelos eventos da nossa view, realizando a persistência dos mesmos. Por outro lado, ainda não temos nenhuma classe para persistir nossos dados. Como o foco do artigo não é este, vamos implementar uma persistência simbólica, em memória, através de um repository, apenas para que possamos fechar o ciclo da aplicação de exemplo.

Para isso, vá ao nosso projeto model, adicione a pasta Repository e crie uma classe nesta pasta, chamada UsuarioRepository. Implemente esta classe como mostrado na **Listagem 4**.

Listagem 4. UsuarioRepository

```
01      using System;
02      using System.Collections.Generic;
03      using System.Linq;
04      using System.Text;
05      using DevMedia.Net.Exemplo.Model.Entities;
06
07      namespace DevMedia.Net.Exemplo.Model.Repository
08      {
09          public static class UsuarioRepository
10          {
11              private static long ultimoId = 0;
12              private static List<Usuario> usuariosCadastrados = new List<Usuario>();
13
14              public static void Inserir(Usuario usuario)
15              {
16                  ultimoId++;
17
18                  usuario.Id = ultimoId;
19                  usuariosCadastrados.Add(usuario);
20              }
21
22              public static void Atualizar(Usuario usuario)
23              {
24                  Deletar(usuario.Id);
25                  usuariosCadastrados.Add(usuario);
26              }
27
28              public static void Deletar(long IdUsuario)
29              {
30                  int idxUsuarioLocalizado = usuariosCadastrados.FindIndex(x => 01
31
32                  usuariosCadastrados.RemoveAt(idxUsuarioLocalizado);
33              }
34
35              public static List<Usuario> GetUsuariosCadastrados()
36              {
```

```
37                  return usuariosCadastrados;
38              }
39
40              public static Usuario GetUsuarioById(long IdUsuario)
41              {
42                  return usuariosCadastrados.Find(x => x.Id.Equals(IdUsuario));
43              }
44          }
45      }
```

Na **Listagem 4** temos umaclasseque simula um repositório de dados. Temos um List privado (linha 12) de usuários em memória e as operações com o CRUD básico que vamos necessitar em nosso exemplo.

O pattern repository prevê uma classe que faz a mediação entre o domínio e as classes de mapeamento e acesso a dados. No nosso caso, para fins didáticos, estamos apenas simulando uma persistência, mas em um caso real, teríamos no nosso repositório o acesso à camada de persistência do sistema.

Observe que tanto a classe quanto todos os métodos da **Listagem 4** são estáticos. A palavra reservada static na classe indica que a classe não pode ser instanciada e que todos os seus métodos precisam ser estáticos. Métodos e properties estáticas são membros da classe e não da instância do objeto.

Por exemplo, em uma implementação comum, suponhamos uma classe Calculo com o método Somar. Para usar o método somar, precisamos criar um objeto do tipo Calculo e então invocar o método somar deste objeto, como por exemplo, new Calculo().somar();

Caso o método somar fosse estático, nós não conseguiríamos acionar o mesmo a partir do objeto, mas somente a partir da classe, sem necessidade de instanciar o objeto Calculo. Executaríamos apenas Calculo.somar();

Por outro lado, a definição de properties estáticas na classe faz com que a mesma tenha comportamento semelhante, porém, com um detalhe que é importante observar: a definição do valor desta variável será compartilhado por toda a aplicação. Portanto, é preciso tomar cuidado quando for usar variáveis e métodos estáticos.

No nosso caso utilizamos o repository estático apenas para fins didáticos, para termos um local centralizado com todos os usuários cadastrados sem precisar fazer rotinas de persistência em disco.

Agora que já temos nossa simulação de persistência implementada, vamos voltar aos nossos presenters. Vamos implementar dois presenters, um para a abordagem Passive View e outra para a Supervising Presenter.

Adicione uma nova classe na pasta Presenters do projeto MVP, e nomeia como UsuarioCadastroPresenterPassiveView. Sendo assim, implemente o presenter de forma que fique como na **Listagem 5.**

Listagem 5. Presenter UsuarioCadastroPresenterPassiveView

```
01  using System;
02  using System.Collections.Generic;
03  using System.Linq;
04  using System.Text;
05  using DevMedia.Net.Exemplo.MVP.Interfaces;
06  using DevMedia.Net.Exemplo.Model.Entities;
07  using DevMedia.Net.Exemplo.Model.Repository;
08
09  namespace DevMedia.Net.Exemplo.MVP.Presenters
10  {
11      public class UsuarioCadastroPresenterPasssiveView
12      {
13          private IUsuarioCadastroPassive _view = null;
14
15          private UsuarioCadastroPresenterPasssiveView() {
16
17          }
18
19          public UsuarioCadastroPresenterPasssiveView(IUsuarioCadastroPassive view)
20          {
21              this._view = view;
22          }
23
24          public void ExibirUsuarioSelecionado()
25          {
26              Usuario usuario = UsuarioRepository.GetUsuarioById(_view.GetIdUsuarioS
27              _view.SetLogin(usuario.Login);
28              _view.SetNome(usuario.Nome);
29              _view.SetSenha(usuario.Senha);
30          }
31
32          public void Salvar() {
33              Usuario usuario = new Usuario();
34              usuario.Login = _view.GetLogin();
35              usuario.Nome = _view.GetNome();
36              usuario.Senha = _view.GetSenha();
37
38              UsuarioRepository.Inserir(usuario);
39              LoadAllUsuarios();
40              _view.LimparControles();
41          }
42
43          public void Atualizar()
44          {
45              Usuario usuario = new Usuario();
46              usuario.Id = _view.GetIdUsuarioSelecionado();
47              usuario.Login = _view.GetLogin();
48              usuario.Nome = _view.GetNome();
49              usuario.Senha = _view.GetSenha();
50
51              UsuarioRepository.Atualizar(usuario);
52              LoadAllUsuarios();
53              _view.LimparControles();
54          }
55
56          public void Deletar()
57          {
58              UsuarioRepository.Deletar(_view.GetIdUsuarioSelecionado());
59              LoadAllUsuarios();
60              _view.LimparControles();
61          }
62
63          public void LoadAllUsuarios()
64          {
65              IList<Usuario> usuarios = UsuarioRepository.GetUsuariosCadastrados();
66              _view.SetListaUsuariosCadastrados(usuarios);
67          }
68      }
69  }
```

Observe na linha15 da **Listagem 5** nós definimos o construtor padrão do presenter como privado, para anular o mesmo e obrigar que o desenvolvedor informe a view quando criar o presenter, usando o construtor sobrecarregado(linha 19) que recebe uma instância de IUsuarioCadastroPassive, que será as nossas views concretas.

Ainda analisando a **Listagem 5,** começa a ficar mais fácil compreendermos o aspecto da testabilidade e do paralelismo da equipe de desenvolvimento. No aspecto da testabilidade, percebemos que temos um presenter pronto para ser testado, independentemente da implementação concreta view. Já no aspecto do paralelismo, podemos notar que todo o código das camadas foram gerados independentemente da implementação da view.

Vamos agora implementar nosso presenter com a abordagem Supervising Presenter. Crie uma nova classe na pasta Presenters do projeto MVP e adicione uma classe chamada UsuarioCadastroPresenterSupervising e implemente a mesma de forma que fique como na **Listagem 6**.

Listagem 6. Presenter UsuarioCadastroPresenterSupervising

```
01     using System;
02     using System.Collections.Generic;
03     using System.Linq;
04     using System.Text;
05     using DevMedia.Net.Exemplo.MVP.Interfaces;
06     using DevMedia.Net.Exemplo.Model.Entities;
07     using DevMedia.Net.Exemplo.Model.Repository;
08
09     namespace DevMedia.Net.Exemplo.MVP.Presenters
10     {
11         public class UsuarioCadastroPresenterSupervising
12         {
13             private IUsuarioCadastroSupervising _view = null;
14
15             private UsuarioCadastroPresenterSupervising() {
16
17             }
18
19             public UsuarioCadastroPresenterSupervising(IUsuarioCadastroSupervising vie
20             {
21                 this._view = view;
22             }
23
24             public void ExibirUsuarioSelecionado()
25             {
26                 Usuario usuario = UsuarioRepository.GetUsuarioById(_view.GetUsuario().
27                 _view.SetUsuario(usuario);
28             }
29
30             public void Salvar() {
31                 Usuario usuario = _view.GetUsuario();
32                 UsuarioRepository.Inserir(usuario);
33                 LoadAllUsuarios();
34                 _view.LimparControles();
35             }
36
37             public void Atualizar()
38             {
39                 Usuario usuario = _view.GetUsuario();
40                 UsuarioRepository.Atualizar(usuario);
41                 LoadAllUsuarios();
42                 _view.LimparControles();
```

```
43              }
44
45              public void Deletar()
46              {
47                  UsuarioRepository.Deletar(_view.GetUsuario().Id);
48                  LoadAllUsuarios();
49                  _view.LimparControles();
50              }
51
52              public void LoadAllUsuarios()
53              {
54                  IList<Usuario> usuarios = UsuarioRepository.GetUsuariosCadastrados();
55                  _view.SetListaUsuariosCadastrados(usuarios);
56              }
57          }
58      }
```

O presenter com abordagem supervising da **Listagem 6** segue o mesmo padrão do presenter passive. A diferença está no tráfego dos dados para a view, onde podemos perceber que o passive possui a responsabilidade do binding dos dados na tela e vice-versa, enquanto que no caso do supervising, o presenter já recupera da view, uma instância pronta do modelo usuário. Essa é a diferença chave entre as duas abordagens.

Implementando nossas views

Agora que já implementamos nosso modelo, nosso repositório e nossos presenters, podemos começar a fazer nossas telas. Observe um detalhe importante: mesmo sem ter as telas, nós já conseguiríamos realizar o teste unitário e automatizar o mesmo criando objetos mocks que implementassem as interfaces da view e passando-os para nossos presenters. Esse é um dos grandes benefícios da separação da view usando o MVP, pois o código de nossas telas tende a ficar muito mais enxuto e simples, reduzindo a complexidade das telas e transferindo-as para nossos presenters, que são mais fáceis de realizar testes automatizados.

Nossas primeiras views serão em WindowsForms. Vamos criar uma com a abordagem Passive View e outra com a abordagem Supervising Presenter. Adicione um novo projeto WindowsForms à solution de nosso exemplo e nomeie o mesmo como DevMedia.Net.Exemplo.WinForms.

Com o projeto criado, a primeira coisa a fazermos é adicionar as referências aos demais projetos (MVP e Model) para que possamos acessar as classes dos mesmos.

Uma vez adicionadas às referências, vamos criar nossas telas. Por padrão, o Visual Studio criou um formulário, chamado Form1, junto com o projeto. Altere o nome deste formulário para FrmCadastroUsuarioPassiveView.

Na opção de designer do formulário, inclua um DataGridView (aba Data), três labels e três textBox, como mostrado na **Figura 6**.

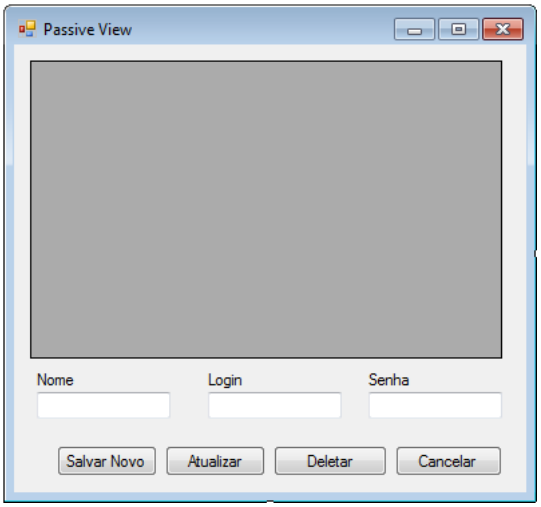

Figura 6. Formulário de Cadastro de Usuários em Windows Forms

Altere a propriedade SelectionMode do DataGridView para FullRowSelect para que possamos recuperar o Id da linha selecionada. Além disso, altere o nome dos controles para que fiquem da seguinte forma:

• dgvUsuarios

• txtNome

• txtLogin

• txtSenha

• btnSalvar

• btnAtualizar

• btnDeletar

• btnCancelar

Na **Listagem 7** temos o código c# de nossa view em Windows forms. Observe na linha 14 que nossa view implementa a interface IUsuarioCadastroPassive, o que a obriga a implementar todos os seus métodos. Na linha 18 nós temos o presenter privado que é instanciado ao criarmos o formulário e é usado em todas as iterações da view. Os métodos getters e setters basicamente recuperam e atribuem os valores passados como parâmetro aos controles da interface.

Listagem 7. View FrmCadastroUsuarioPassiveView

```
01      using System;
02      using System.Collections.Generic;
03      using System.ComponentModel;
04      using System.Data;
05      using System.Drawing;
06      using System.Linq;
07      using System.Text;
08      using System.Windows.Forms;
```

```
09  using DevMedia.Net.Exemplo.MVP.Presenters;
10  using DevMedia.Net.Exemplo.MVP.Interfaces;
11
12  namespace DevMedia.Net.Exemplo.WinForms
13  {
14      public partial class FrmCadastroUsuarioPassiveView : Form, IUsuarioCadastroPas
15      {
16
17          private long IdUsuarioSelecionado = 0;
18          private UsuarioCadastroPresenterPasssiveView presenter = null;
19
20          public FrmCadastroUsuarioPassiveView()
21          {
22              InitializeComponent();
23              presenter = new UsuarioCadastroPresenterPasssiveView(this);
24          }
25
26          private void btnSalvar_Click(object sender, EventArgs e)
27          {
28              presenter.Salvar();
29          }
30
31          private void btnAtualizar_Click(object sender, EventArgs e)
32          {
33              presenter.Atualizar();
34          }
35
36          private void btnDeletar_Click(object sender, EventArgs e)
37          {
38              presenter.Deletar();
39          }
40
41          private void btnCancelar_Click(object sender, EventArgs e)
42          {
43              LimparControles();
44
45              IdUsuarioSelecionado = 0;
46          }
47
48          public long GetIdUsuarioSelecionado()
49          {
50              return IdUsuarioSelecionado;
51          }
52
53          public string GetNome()
54          {
55              return txtNome.Text;
56          }
57
58          public string GetLogin()
59          {
60              return txtLogin.Text;
61          }
62
63          public string GetSenha()
64          {
65              return txtSenha.Text;
66          }
67
68          public void SetIdUsuarioSelecionado(long value)
69          {
70              IdUsuarioSelecionado = value;
71          }
72
73          public void SetNome(string value)
74          {
75              txtNome.Text = value;
76          }
77
```

```
78              public void SetLogin(string value)
79              {
80                  txtLogin.Text = value;
81              }
82
83              public void SetSenha(string value)
84              {
85                  txtSenha.Text = value;
86              }
87
88              public void SetListaUsuariosCadastrados(IList<Model.Entities.Usuario> usua
89              {
90                  dgvUsuarios.DataSource = null;
91                  dgvUsuarios.DataSource = usuarios;
92                  dgvUsuarios.Refresh();
93              }
94
95              private void dgvUsuarios_CellClick(object sender, DataGridViewCellEventArg
96              {
97                  IdUsuarioSelecionado = 0       long.Parse(dgvUsuarios.SelectedRows[0].
98                  presenter.ExibirUsuarioSelecionado();
99              }
100
101             public void LimparControles()
102             {
103                 txtNome.Clear();
104                 txtLogin.Clear();
105                 txtSenha.Clear();
106             }
107         }
108     }
```

A view com a abordagem Supervising Presenter terá os mesmos controles e layout da Passive View e o código também será parecido. As únicas diferenças são:

• A interface implementada agora é a IUsuarioCadastroSupervising

• O presenter utilizado é o UsuarioCadastroPresenterSupervising

• Os getters e setters de cada propriedade foram substituídos pelo GetUsuario e SetUsuario, como mostrado na **Listagem 8**.

Listagem 8. View FrmCadastroSupervisingPresenter

```
01      public Model.Entities.Usuario GetUsuario()
02      {
03            Usuario usuario = new Usuario();
04            usuario.Id = IdUsuarioSelecionado;
05            usuario.Nome = txtNome.Text;
06            usuario.Login = txtLogin.Text;
07            usuario.Senha = txtSenha.Text;
08
09            return usuario;
10      }
11
12      public void SetUsuario(Model.Entities.Usuario usuario)
13      {
14            IdUsuarioSelecionado = usuario.Id;
15            txtNome.Text = usuario.Nome;
16            txtLogin.Text = usuario.Login;
17            txtSenha.Text = usuario.Senha;
18      }
```

Nota: Para evitar redundância e poluição na leitura do artigo, o código completo do FrmCadastroSupervisingPresenter foi suprimido e foi exibido apenas os métodos adicionais na **Listagem 8.**

Como podemos observar na **Listagem 8**, com a abordagem Supervising Presenter, a view fica encarregada de realizar o binding do objeto nos controles.

Nota: Neste caso, como o objetivo é apresentar o MVP, usamos uma maneira simples para atualizar o modelo com os dados alterados em tela e vice-versa. Porém, é comum a implementação do pattern observer junto ao MVP, fazendo com que as views sejam observadoras do modelo e a cada alteração no modelo as views são automaticamente notificadas e atualizam a renderização na tela.

Para finalizar nosso exemplo, vamos implementar nossa view na web, consumindo os mesmos presenters. No exemplo anterior, em WindowsForms, nós já falamos sobre como seria a implementação do PassiveView e do SupervisingPresenter e mostramos as diferenças das duas abordagens. Por este motivo, no exemplo em web, faremos apenas a implementação de uma das formas, a Passive View.

Adicione à solution um novo projeto, do tipo Web Application, chamado DevMedia.Net.Exemplo.WebForms. Por padrão o Visual Studio já cria alguns arquivos e estilos padrões. Vamos manter os mesmos e trabalhar em cima do próprio Default.aspx. Antes de prosseguir, adicione a este projeto as referências aos projetos MVP e Model.

Nossa página aspx terá o layout parecido com o que criamos no Windows forms. A única diferença vai ser na edição e na deleção, onde teremos uma opção diretamente na grid para realizar estas operações, como mostrado na **Figura 7**.

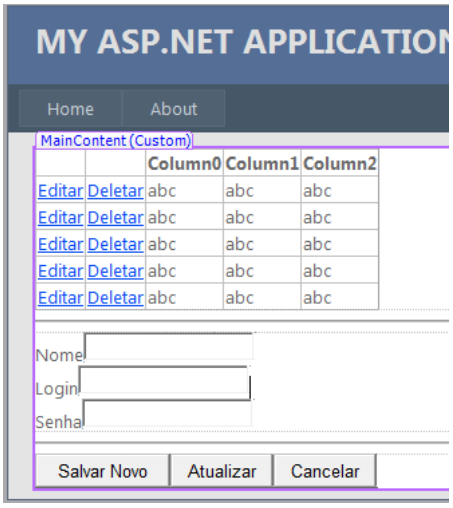

Figura 7. Formulário de Cadastro de Usuários na Web

Na **Listagem 9** temos o aspx de nossa página. Observe na linha 07 que temos o evento

RowCommand da grid. Nele iremos verificar qual o command disparado(editar / deletar) e então chamaremos a respectiva rotina do presenter.

Listagem 9. Código ASPX do cadastro de usuários na web

```
01    <%@ Page Title="Home Page" Language="C#" MasterPageFile="~/Site.master" AutoEventW
02
03    <asp:Content ID="HeaderContent" runat="server" ContentPlaceHolderID="HeadContent">
04    </asp:Content>
05    <asp:Content ID="BodyContent" runat="server" ContentPlaceHolderID="MainContent">
06    <asp:GridView runat="server" ID="grvUsuarios" DataKeyNames="Id"
07       onrowcommand="grvUsuarios_RowCommand" >
08       <Columns>
09           <asp:ButtonField CommandName="editar" Text="Editar" />
10           <asp:ButtonField CommandName="deletar" Text="Deletar" />
11       </Columns>
12   </asp:GridView>
13   <hr />
14   <asp:Label ID="Label1" runat="server">Nome</asp:Label>
15   <asp:TextBox ID="txtNome" runat="server"></asp:TextBox><br />
16
17   <asp:Label ID="Label2" runat="server">Login</asp:Label>
18   <asp:TextBox ID="txtLogin" runat="server"></asp:TextBox><br />
19
20   <asp:Label ID="Label3" runat="server">Senha</asp:Label>
21   <asp:TextBox ID="txtSenha" runat="server"></asp:TextBox><br />
22
23   <hr />
24   <asp:Button runat="server" ID="btnSalvar" OnClick="btnSalvar_click" Text="Salvar No
25   <asp:Button runat="server" ID="btnAtualizar" OnClick="btnAtualizar_click"
      Text="Atualizar"/>
26       <asp:Button runat="server" ID="btnCancelar" OnClick="btnCancelar_click" Text="
27    </asp:Content>
```

Na **Listagem 10** temos o code behind de nossa view. Observe que seguimos o mesmo padrão usado no Windows Forms, fazendo com que nosso objeto(neste caso Page), implementasse a interface IUsuarioCadastroPassive(Linha 12).

Listagem 10. Code behind cadastro de usuários na web

```
01    using System;
02    using System.Collections.Generic;
03    using System.Linq;
04    using System.Web;
05    using System.Web.UI;
06    using System.Web.UI.WebControls;
07    using DevMedia.Net.Exemplo.MVP.Interfaces;
08    using DevMedia.Net.Exemplo.MVP.Presenters;
09
10    namespace DevMedia.Net.Exemplo.WebForms
11    {
12        public partial class _Default : System.Web.UI.Page, IUsuarioCadastroPassive
13        {
14
15            private const string VIEW_STATE_ID_USUARIO = "ID_USUARIO_SELECIONADO";
16            private UsuarioCadastroPresenterPasssiveView _presenter = null;
17
18            protected void Page_Load(object sender, EventArgs e)
19            {
20                _presenter = new UsuarioCadastroPresenterPasssiveView(this);
21            }
22
23            protected void grvUsuarios_RowCommand(object sender, GridViewCommandEventA
```

```
24              {
25                  int idxUsuarioSelecionado =       Convert.ToInt32(grvUsuarios.Rows[Con
26                  SetIdUsuarioSelecionado(idxUsuarioSelecionado);
27
28                  if (e.CommandName.Equals("editar"))
29                      _presenter.ExibirUsuarioSelecionado();
30                  else
31                  if (e.CommandName.Equals("deletar"))
32                      _presenter.Deletar();
33              }
34
35              protected void btnSalvar_click(object sender, EventArgs e)
36              {
37                   _presenter.Salvar();
38             }
39
40          protected void btnAtualizar_click(object sender, EventArgs e)
41          {
42                 _presenter.Atualizar();
43          }
44
45          protected void btnCancelar_click(object sender, EventArgs e)
46          {
47                 SetIdUsuarioSelecionado(0);
48                 LimparControles();
49          }
50
51          public long GetIdUsuarioSelecionado()
52          {
53             if (ViewState[VIEW_STATE_ID_USUARIO] != null)
54                return Convert.ToInt32(ViewState[VIEW_STATE_ID_USUARIO].ToString());
55             else
56                 return 0;
57          }
58
59          public string GetNome()
60          {
61             return txtNome.Text;
62          }
63
64          public string GetLogin()
65          {
66             return txtLogin.Text;
67          }
68
69          public string GetSenha()
70          {
71             return txtSenha.Text;
72          }
73
74          public void SetIdUsuarioSelecionado(long value)
75          {
76             ViewState[VIEW_STATE_ID_USUARIO] = value;
77          }
78
79          public void SetNome(string value)
80          {
81             txtNome.Text = value;
82          }
83
84          public void SetLogin(string value)
85          {
86             txtLogin.Text = value;
87          }
88
89           public void SetSenha(string value)
90           {
91              txtSenha.Text = value;
92           }
```

```
93
94            public void SetListaUsuariosCadastrados(IList<Model.Entities.Usuario> usuar
95            {
96               grvUsuarios.DataSource = null;
97               grvUsuarios.DataSource = usuarios;
98               grvUsuarios.DataBind();
99            }
100
101            public void LimparControles()
102            {
103               txtNome.Text = "";
104               txtLogin.Text = "";
105               txtSenha.Text = "";
106            }
107       }
108    }
```

A principal particularidade desta implementação na Web é com relação ao armazenamento do id do usuário selecionado. No Windows forms nós armazenamos ele em uma variável privada da classe. Isso foi possível, pois uma aplicação desktop é Statefull, ou seja, possui estado gerenciado. Uma vez instanciado o objeto, ele permanece “vivo” em memória, permitindo armazenarmos objetos e variáveis nele pelo período em que ele existir.

No caso da **Listagem 10** temos o armazenamento do id do usuário selecionado na ViewState. Isto se faz necessário, pois as aplicações Web são aplicações StateLess, ou seja, sem estado. Isso significa que ao acessar uma página web, uma requisição é disparada ao servidor, que processa a mesma (neste momento entramos no ciclo de vida da página asp.net), após o processamento o servidor retorna um HTML em texto para o browser que renderiza o mesmo. Isso quer dizer que nosso objeto Page só existiu durante o processamento do servidor, ou seja, ele foi criado, mantido e destruído após o processamento, com isso perderíamos o valor do id selecionado.

Usando ViewState, nós serializamos este valor no HTML da página, fazendo com que o ASP.NET consiga recuperar o mesmo ao longo das requisições.

espalhadas pelo código. Definimos em uma constante o nome da chave que vamos usar no ViewState e então usamos ela sempre que precisamos acessar o valor armazenado nesta chave do ViewState.

Conclusão

Como podemos ver, o MVP nos fornece uma solução elegante para o desacoplamento de nossa camada de visualização. O fato do mesmo facilitar a testabilidade e também a troca de tecnologias de visualização faz com que tenhamos mais uma boa opção em mãos. O MVP é um dos padrões existentes para isolar a camada de apresentação, porém existem outros, como por exemplo, o MVC (Model View Controller) e o MVVM(Model View View Model), que também merecem sua atenção antes de decidir qual padrão usar para seu projeto.

Além disso, o MVP pode ser combinado com outros padrões para que se consiga aumentar ainda mais o reuso de código e o desacoplamento, como por exemplo, o caso do Observer para comunicação da view com o model. Outra ideia que pode ser interessante é criar um presenter genérico para seus cadastros básicos, com o CRUD padrão, combinando com Generics e com algum pattern criacional, como o Single Factory por exemplo. Enfim, existem algumas possibilidades bem interessantes do uso do MVP, procure não se limitar a um padrão, mas a um mix de padrões, de acordo com cada problema que você encontra no seu dia a dia.

Publicado no Canal .net

por Ricardo da Silva Coelho
Guru .net e tecnologias MS

· Publicado em 2012

Ficou com alguma dúvida?

Post aqui sua dúvida ou comentário que nossa equipe responderá o mais rápido possível.

Mais posts

Artigo
Introdução ao WinJS

Artigo
Conheça o Microsoft Azure IoT

Artigo
Utilizando o Bootstrap com ASP.NET

Artigo
Utilizando o Microsoft Azure Storage

Revista
Revista .net Magazine 129

Listar mais conteúdo