

Nesta seção você encontra artigos sobre a linguagem PHP e a ferramenta Delphi for PHP

Orientação a Objetos no Delphi for PHP

Como aplicar conceitos de POO em aplicações PHP — Parte 3



Rodrigo Carreiro Mourão

(rodrigocarreiro@tdstecnologia.com.br)

é consultor da TDS Tecnologia – RJ atuando na área de desenvolvimento de projetos Orientados a Objetos, Design Patterns, MVC. BDS2006 Win32 Product Certified. Instrutor de treinamentos oficiais Delphi for Win32, Delphi for PHP, Delphi .Net. Palestrante do BorCon 2007. É mantenedor do site www.delphisophp.com desenvolvido totalmente em Delphi for PHP.

Olá, ainda nesta linha de OOP no Delphi for PHP neste terceiro e último artigo trataremos de um assunto pouco conhecido pela maioria dos programadores, mas que é fundamental para um bom desenvolvimento: os padrões de projeto. Como podemos observar, a orientação a objetos facilita e muito a nossa vida em relação aos códigos que escrevemos e a manutenção desses códigos, porém conforme vamos avançando em nossos projetos vamos nos deparando com situações delicadas que nos tomam um tempo precioso. Problemas recorrentes que se repetem em cada novo projeto que iniciamos. Se reparar bem todos os programadores têm aquela famosa *Unit* “Faz Tudo” onde temos rotinas prontas que nos ajudam a resolver estes problemas corriqueiros.

Mas em projetos grandes onde temos um a equipe envolvida esta *Unit* não ajudará muito. É preciso que haja

uma linguagem universal que todos entendam, ou melhor, é preciso que se desenvolva dentro de um padrão, mas o grande problema é que cada equipe pode ter o seu padrão, cada empresa sua linha de raciocínio e aquela velha máxima “Cada um na sua, mas com alguma coisa em comum”.

Enfim, um ponto final foi colocado neste dilema em 1995 com o surgimento dos Design Patterns ou Padrões de Projeto para Sistemas Orientados a Objetos.

Histórico

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 Cristopher 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. Neste 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*, Cristopher estabelece que um padrão deve 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.

Além destas características, Alexander definiu que um padrão deve ser descrito em 5 partes:

- Nome;
- Exemplo;
- Contexto;
- Problema;
- Solução;

Padrões de Projetos - Design Patterns

Anos mais tarde Erich Gamma, Richard Helm, Ralph Johnson e John Vlissides iniciaram suas pesquisas baseadas 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.

Padrões criacionais

- Abstract Factory;
- Builder;
- Factory Method;
- Prototype;
- Singleton;

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;

Nas **Figuras 1, 2, 3, 4 e 5** temos os diagramas UML de alguns Padrões de Projeto descritos acima.

Interfaces

Falar de Padrões de Projetos sem falar de *interfaces* é como cometer um crime. A maioria dos padrões utilizam *interfaces* para serem implementados. Mas o que vem a ser uma *interface*?

Em OO temos a herança como recurso para reaproveitar códigos e facilitar a manutenção, mas a herança

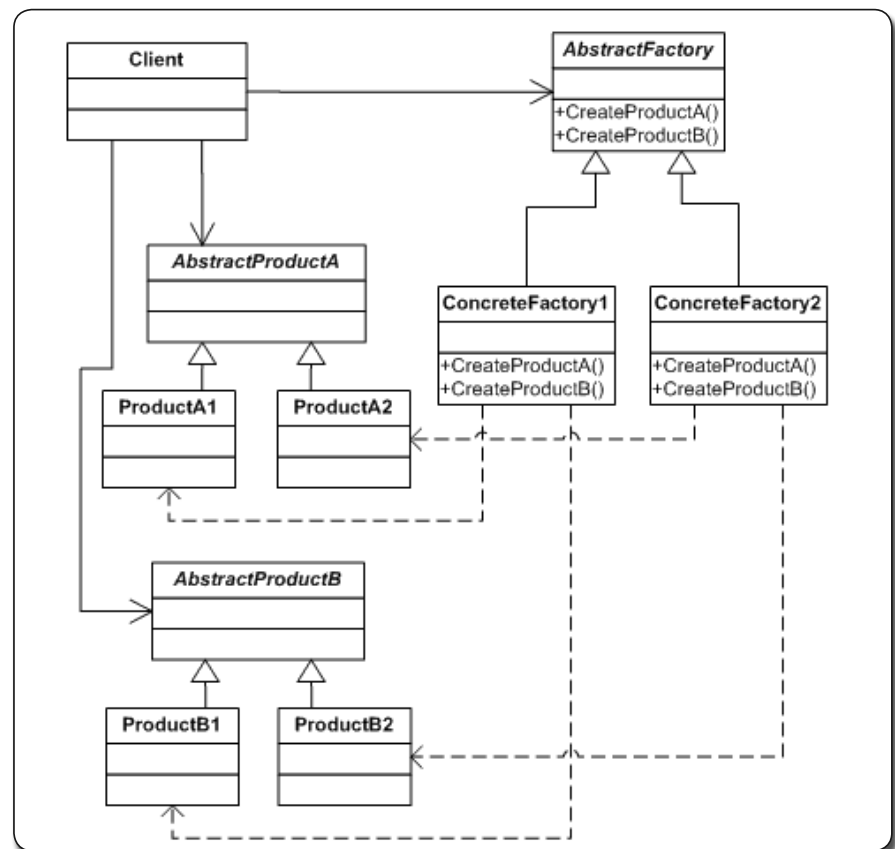


Figura 1. Diagrama UML para padrão Abstract Factory

Listagem 1. Utilizando Interfaces

```
<?php
interface IDAO{
    function Gravar();
}
class ClienteDAO implements IDAO{
    function Gravar(){
        echo "Gravando Cliente !!";
    }
}
class FornecedorDAO implements IDAO{
    function Gravar(){
        echo "Gravando Fornecedor !!";
    }
}
function GravarObjeto($Obj){
    if ($Obj instanceof IDAO){
        $Obj->Gravar();
    }else{
        echo "O objeto passado nao suporta a interface IDAO";
    }
}
?>
```

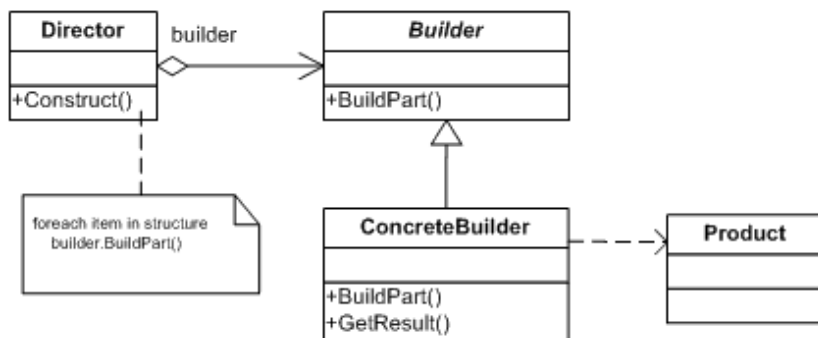


Figura 2. Diagrama UML para padrão Builder

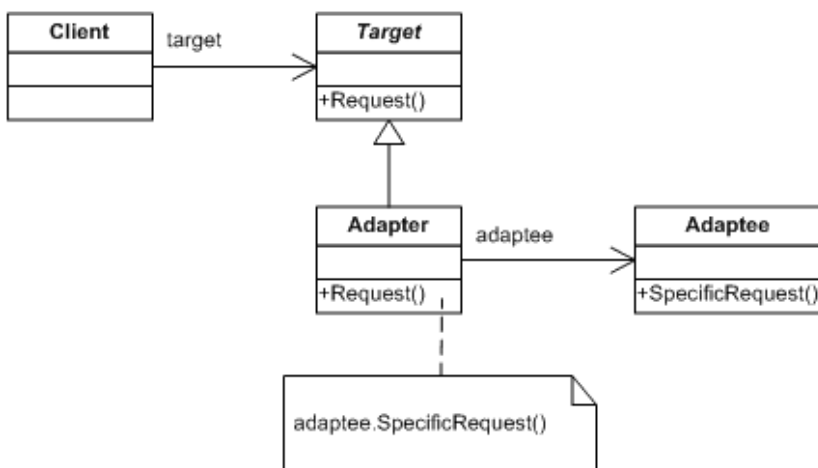


Figura 3. Diagrama UML para padrão Adapter

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 permite o uso de herança múltipla, porém os desenvolvedores do PHP 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 desta casa.

Não importa qual construtora irá construí-la, 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. Observe na **Listagem 1** como utilizamos *interfaces* na sintaxe do PHP.

O simples fato de implementar a *interface* IDAO em nossas classes nos obriga a declarar o método *Gravar()* em todas as classes. Veja que foi declarado um “contrato” (interface), que possui uma única “cláusula” (método), que é *Gravar()*. Agora toda classe que implementar esta *interface* deve respeitar esta cláusula, ou seja, deve implementar este método para fazer conforme sua necessidade. Feito isto criamos duas classes que implementam esta *interface* e cada uma destas classes chamadas concretas implementa o método a sua maneira.

O segredo do código está no método *GravarObjeto*. Observe que ele verifica o parâmetro passado para certificar-se de que ele implementa a *interface* IDAO, ou

seja, ele verifica se a classe passada como parâmetro respeita o contrato. Se a classe segue o contrato então ela terá o método *Gravar()*, pois esta é a única “cláusula” deste contrato. Assim invocamos o método *Gravar()* sem nem mesmo saber de qual classe se trata no momento.

Com *interfaces* resolvemos um problema comum do uso da herança: o acoplamento. Desta maneira programamos para uma *interface* e não para uma implementação. Diminuímos o acoplamento e aumentamos a abstração.

Padrões no PHP

Antes de abordarmos os padrões no PHP é preciso deixar claro que *Design Patterns* não é uma nova linguagem e sim boas práticas no desenvolvimento de software. Estas práticas não são proprietárias desta ou daquela linguagem de desenvolvimento. Já ouvi de algumas pessoas que padrões de projetos eram restritos a linguagem Java, outros dizendo ser impossível aplicar *Design Patterns* em projetos desenvolvidos em Delphi, o que não é verdade. Padrões de projetos podem e devem ser utilizados em toda linguagem que seja Orientada a Objetos. Não será possível aqui abordar todos os padrões descritos pelo *GOF*, porém abordaremos um dos padrões que é frequentemente utilizado: o *Singleton*.

Singleton

Singleton é um padrão *criacional* que garante a existência de apenas uma instância de uma classe, mantendo um ponto global de acesso ao seu objeto (Figura 6).

Há situações no desenvolvimento de software que queremos garantir que haverá uma e apenas uma instância de determinada classe em nosso projeto. Um exemplo que ilustra bem este cenário seria a classe usuário de uma aplicação. Quantos objetos *Usuário* terão ou precisamos ter instanciados por vez na aplicação? Ou quantos objetos de conexão com o banco de dados precisamos ter? Apenas um. Para estas situações o padrão *Singleton* se aplica perfeitamente.

Abra o Delphi for PHP e crie uma nova aplicação usando *File|New>Application*. Salve-a numa pasta de sua preferência com

Listagem 2. Classe usuario com padrão Singleton

```
<?php
class Usuario{
    private static $instance;
    public $nome = null;
    private function __construct(){
    }
    public static function GetInstance(){
        if (!isset(self::$instance)){
            self::$instance = new Usuario();
        }
        return self::$instance;
    }
    public function Login($user, $pwd){
        if (($user == 'admin') && ($pwd == 'admin')){
            $this->nome = $user;
            return(true);
        }else{
        }
    }
}
?>

<script language="javascript">
    alert('Usuario ou Senha Invalidos');
</script>
<?php
}
}
?>
```

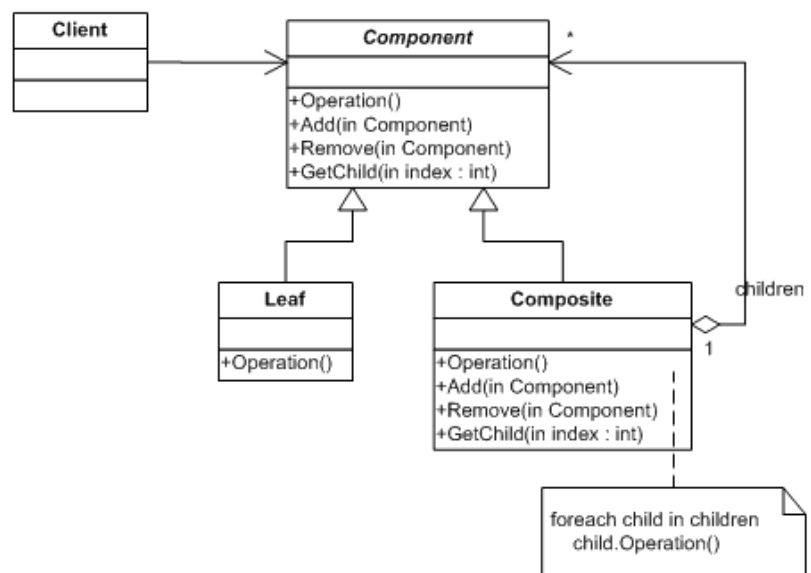


Figura 4. Composite

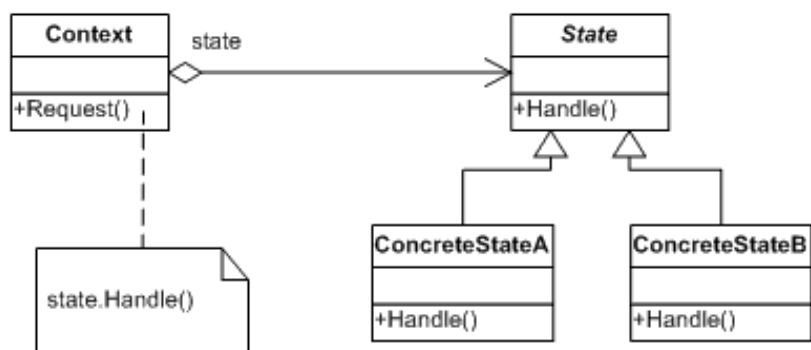


Figura 5. Diagrama UML para padrão State

o nome "Singleton.php", e o formulário principal como "index.php". A idéia é criar um sistema de *Login* onde teremos uma classe usuário implementando o *Pattern Singleton*. No formulário principal da aplicação crie uma tela de *Login* conforme a **Figura 7**. Para isso inclua um componente *Panel* da paleta *Standard* e limpe sua propriedade *Caption*. Modifique sua propriedade *BorderColor* usando o valor hexadecimal "#00C6C6" para mudar a cor e o valor 2 na *BorderWidth* para que tenhamos uma borda mais grossa. Em seguida inclua um

Label dentro do *Panel* modificando as propriedades *Caption* igual "Acesso Restrito", *Color* igual a "#00C6C6", em *Height* coloque 17. Modifique também *Font.Case* para *CaCapitaliz*, *Font.Color* para "#FFFFFF", *Font.Size* igual a "15px" e *Font.Weight* com o valor *bold*. Por fim alinhe manualmente esse *Label* de forma que fique "grudado" com a parte superior do *Panel*. Agora complemente o *Panel* inserindo componentes *Label*, *Edit* e um *Button* assim como na **Figura 7**.

Altere os nomes dos *Edit*'s para "EdtUsuario" e "EdtSenha". Adicione ao projeto uma nova *Unit* onde iremos criar nossa classe *Usuário*. Para adicionar essa *Unit* use o menu *File|New>Unit*. Salve-a como "usuario.php". Feito isso nela adicione o código da **Listagem 2**.

A definição deste padrão diz que temos que garantir que apenas um objeto seja instanciado em nosso modelo. Este é o motivo pelo qual se define neste padrão o construtor da classe como privado como segue:

```
private function __construct(){}

```

De outra maneira poderíamos instanciar quantos objetos quiséssemos. Se tentarmos instanciar um objeto através do método *new* uma exceção seria levantada semelhante a mensagem a seguir:

```
Fatal error: Call to private Usuário::__construct() from context 'Unit2' in <caminho>\index.php on line 21.

```

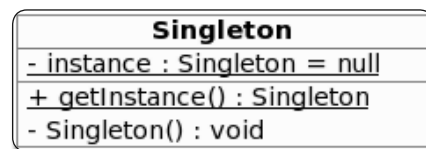


Figura 6. Padrão Singleton

Figura 7. Tela exemplo de login

A definição ainda diz que neste padrão devemos ter um ponto de acesso global. Este ponto é o método estático *GetInstance*. Ele é estático, pois assim não precisamos instanciar a classe para acessá-lo e é nele que fazemos a verificação para saber se já existe uma instância deste objeto em memória. Isto é feito testando-se a variável também estática *\$instance*.

Se seu conteúdo for *null* instanciamos um objeto do tipo *Usuario* e guardamos sua referência na variável *\$instance*. Do contrário apenas retornamos o ponteiro para o objeto que já existe.

Para verificar o funcionamento do nosso código criamos um método *Login* que recebe dois parâmetros: o *usuário* e a *senha* de quem pretende se logar. Claro que aqui temos um *Login* hipotético, mas nada impede que neste momento seja feito um *Select* ao banco de dados para verificar a existência ou não de um usuário com estes parâmetros. Nesse caso, fixamos o usuário e a senha como sendo *admin*, assim como podemos ver a seguir:

```
...
if (($user == 'admin') && ($pwd == 'admin')){
...

```

Após se logar com sucesso, nós guardamos o nome do usuário logado na propriedade *\$nome* e retornamos *true* no método. Caso contrário enviamos uma mensagem ao usuário utilizando *JavaScript*.

Vamos agora à implementação efetiva do nosso exemplo. Ainda na tela de *Login* dê um duplo clique no botão e digite o código a seguir:

```
if (Usuario::GetInstance()->Login($this->EdtUsuario->Text,$this->EdtSenha->Text)){
    $this->Label3->Caption = "Bem Vindo " . Usuario::GetInstance()->nome;
}

```

Observe que não houve a necessidade de instanciar o objeto através do operador *new* e não poderíamos. Como o método *GetInstance()* foi declarado como estático, podemos invocá-lo atrás da classe. Este método irá nos retornar uma instância do objeto usuário e mais do que isto, cada chamada a este método sempre retornará o mesmo usuário.

O que estamos fazendo é bem simples, pedimos uma instância de usuário e invocamos o método *Login* passando o conteúdo dos *Edit*'s. Uma vez auten-



Nota do DevMan

Se consultarmos o verbete JavaScript no wikipedia.org veremos a seguinte definição sobre JavaScript:

JavaScript é uma linguagem de programação criada pela Netscape em 1995, que a princípio se chamava LiveScript, para atender, principalmente, às seguintes necessidades:

- * Validação de formulários no lado cliente (programa navegador);
- * Interação com a página. Assim, foi feita como uma linguagem de script. Javascript tem sintaxe semelhante à do Java, mas é totalmente diferente no conceito e no uso.

1. Oferece tipagem dinâmica - tipos de variáveis não são definidos;
2. É interpretada, ao invés de compilada;
3. Possui ótimas ferramentas padrão para listagens (como as linguagens de script, de modo geral);
4. Oferece bom suporte a expressões regulares (característica também comum a linguagens de script).

Sua união com o CSS é conhecida como DHTML. Usando o JavaScript, é possível modificar dinamicamente os estilos dos elementos da página em HTML.

Dada sua enorme versatilidade e utilidade ao lidar com ambientes em árvore (como um documento HTML), foi criado a partir desta linguagem um padrão ECMA, o ECMA-262, também conhecido como ECMAScript. Este padrão é seguido, por exemplo, pela linguagem ActionScript da Adobe (antiga Macromedia).

Além de uso em navegadores processando páginas HTML dinâmicas, o JavaScript é hoje usado também na construção do navegador Mozilla, o qual oferece para a criação de sistemas GUI todo um conjunto de ferramentas (em sua versão normal como navegador, sem a necessidade de nenhum software adicional), que incluem (e não apenas) um interpretador de Javascript, um comunicador Javascript <-> C++ e um interpretador de XUL, linguagem criada para definir a interface gráfica de aplicações.

O uso de JavaScript em páginas XHTML, pelo padrão W3C, deve ser informado ao navegador da seguinte forma:

```
<script type="text/javascript">
/* aqui fica o script */
</script>

```

Em outras palavras, o JavaScript é uma linguagem de script que nos auxilia no desenvolvimento de aplicações para Web, sejam elas desenvolvidas em PHP, ASP.NET, HTML, ou qualquer outra linguagem destinada a estes fins.

ticado, apenas exibimos no *Label* uma mensagem de boas-vindas ao usuário. Mas como saber se realmente só há uma instância do objeto? Simples. Repare que para recuperar o nome do usuário também utilizamos o *GetInstance()*. Se

um novo objeto fosse retornado, o conteúdo da propriedade nome seria vazio, pois atribuímos o valor a esta propriedade no método *Login*. Assim, caso uma nova instância fosse criada ao chamar o método *GetInstance()* pela segunda vez

a mensagem exibida seria apenas “Bem-Vindo”. Execute a aplicação, faça o *Login* e observe o resultado na **Figura 8**.

Conclusão

Concluimos assim esta pequena introdução sobre o mundo OO no Delphi for PHP. Muito ainda há o que se falar, principalmente sobre padrões de projeto. Lembrando que orientação a objetos gera muita discussão por ser um tema polêmico e altamente detalhado. Recomendo a consulta de livros e comunidades especializadas no assunto, pois quanto maior for o aprofundamento das técnicas de OO, melhor êxito terá no desenvolvimento de suas aplicações futuras.

Um abraço a todos e até a próxima. Eu sou Rodrigo Carreiro e pela sua atenção, muito obrigado! ●

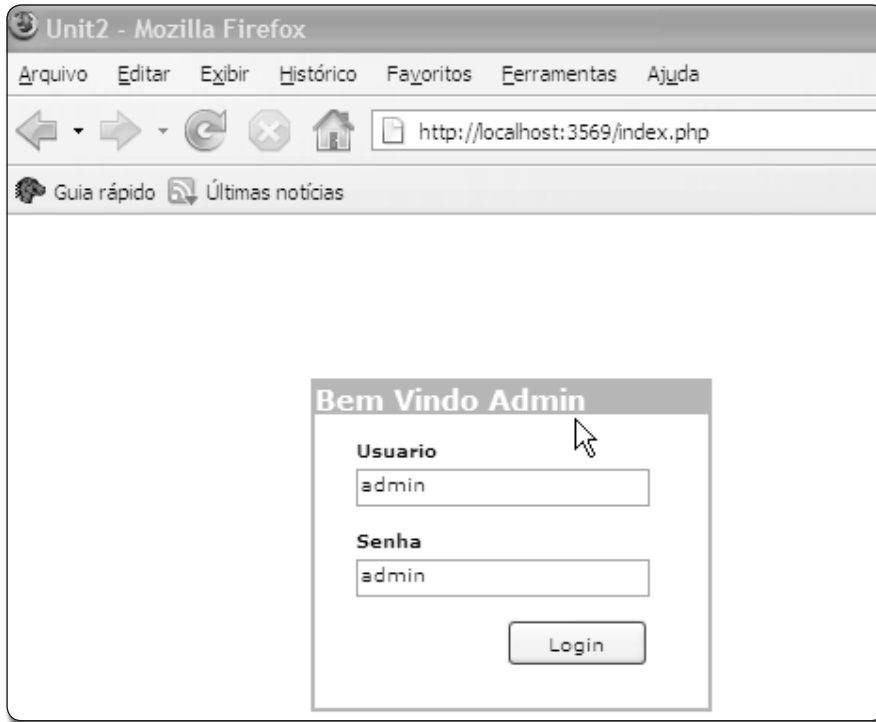


Figura 8. Aplicação em Execução

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

