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?

Rodrigo Carreiro Mourão
(r_c_mourao@hotmail.com)
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 CodeGear Delphi for Win32, Delphi for PHP, Delphi .Net. Palestrante do Borland Conference 2007.

Sem dúvida nenhuma POO é assunto para vários e vários capítulos de discussão. Nesse artigo veremos como aplicar e trabalhar com Orientação a Objetos em linguagem PHP com a nova ferramenta Delphi for PHP. Com toda certeza poderemos tirar bom proveito dos recursos do novo IDE e desenvolver aplicativos para *Web* cada vez mais robustos e flexíveis.

A POO é o alicerce da programação como um todo, e com ela é possível desenvolver os mais variados aplicativos com diversas vantagens. Essas vantagens podem ser chamadas de *Pilares*, que são ao todo quatro: *Polimorfismo, Herança, Abstração* e *Encapsulamento*.

Veremos os principais aspectos da Orientação a Objetos e como aplicá-los em um *website* escrito em PHP de maneira clara e objetiva. Para entendermos bem todo o conjunto de informações referentes a POO, farei breves comparações com o Delphi para que possamos nos situar e entender melhor como tudo funciona.

Os Pilares da Orientação a Objetos

Toda linguagem dita orientada a objetos deve estar firmada no que chamamos de *Pilares* (**Figura 1**), como dito anteriormente. Estes conceitos não estavam totalmente disponíveis até a versão 4 do PHP. Com o lançamento da versão 5, o modelo orientado a objetos da linguagem foi totalmente reescrito. Em relação a OO não seria exagero dizer que era uma outra linguagem e bastante difícil de lidar. Quem "tentou" programar OO no PHP 4 sabe do que estou falando.

Todos os pilares citados anteriormente estão presentes na versão 5 o que nos permite desenvolver aplicações *Web* em cima desta filosofia. Um modelo orientado a objetos gira em torno de classes e objetos. Como mencionado, OO é um assunto polêmico e muito extenso, portanto não serão abordados aqui todos os seus conceitos a fundo. É altamente recomendado um estudo aprofundado dos conceitos básicos da POO para que se possa entender perfei-

tamente toda a estrutura de programação nesses moldes. Duas boas referências para os iniciantes estão na revista ClubeDelphi 91 com dois excelentes artigos dos autores Adriano Santos e Maikel Sheid.

Contudo para que se possa situar, as classes seriam como a planta de um edifício com suas definições de altura, largura, quantidades de andares, elevadores etc., ou seja, o planejamento do edifício, algo ainda no papel, não tangível. Já o objeto seria o próprio edifício construído, concreto e palpável.

Todo objeto possui suas características que chamamos de *propriedades*. Possui também comportamentos que chamamos de *métodos*. Tudo isso definido na classe ao qual este objeto pertence. A vantagem da OO em relação a outras filosofias de programação é justamente a possibilidades de criar objetos passíveis de serem reaproveitados reduzindo assim o tempo de desenvolvimento. Outra vantagem é a de manter os dados (*propriedades*) e regras de negócios (*métodos*) unidos numa mesma estrutura (*objeto*).

Uma vez tendo estes conceitos em mente, basta aplicá-los a linguagem em que iremos trabalhar, no nosso caso PHP. Bem, até agora está tudo muito bom, mas na prática, como isso se aplica? Esta é a pergunta que não quer calar. Primeiro vamos à sintaxe OO no PHP.

Criando e entendendo as classes

Para criarmos uma classe utilizamos, como no Delphi, a palavra reservada *class*, porém a sintaxe é um pouco diferente, como podemos ver a seguir:

```
class Pessoa{}
```

Como podemos ver, não temos aqui a presença do operador *type* como no Delphi. As propriedades e métodos desta classe devem ser definidos entre as chaves que a definem. Tudo que for colocado neste escopo pertence a ela:

```
class Pessoa{
  var $nome;
  var $email;
}
```

Entendendo os objetos

Temos agora nossa classe que possui a definição do que será nosso objeto propriamente dito. Precisamos nesse momento criar esses objetos, alocá-los em memória e torná-los reais. Para instanciarmos uma classe no PHP utilizamos a palavra reservada *new*. Neste ponto não custa lembrar que o PHP não é fortemente tipado como é o Win32 e mais ainda ele é *Case Sensitive*, ou seja, difere maiúsculas de minúsculas, portanto muita atenção na hora que definir suas classes e as variáveis que irão receber seus objetos. Para criarmos um da classe *Pessoa*, definida anteriormente, bastaria atribuir a uma variável o resultado da função *new* como a seguir:

```
$joao = new Pessoa;
```

Neste momento temos em *$joao* um objeto com as definições da nossa classe, com isso é fato que *$joao* tem um nome e possui um e-mail, ambos aptos a receberem valores.

Listagem 1. Inclusão do construtor da classe

```
class Pessoa{
  var $nome;
  var $email;
  function __construct($nm, $em){
  }
}
```

Construtores

Até o PHP 4, para definirmos um construtor para nossa classe, bastava criar um método com o mesmo nome da classe que este seria invocado quando o objeto fosse construído. Isso ainda funciona com o PHP 5, porém agora temos um método construtor exclusivo para nossas classes e recomenda-se utilizá-lo uma vez que o conceito anterior só foi mantido por questões de compatibilidade.

No PHP 5 quando queremos determinar um construtor para nossa classe basta incluirmos na mesma o método *__construct()*. Este método será invocado sempre que um novo objeto desta classe for instanciado, com isso podemos definir regras para nossa classe como, por exemplo, só permitir que se crie uma nova pessoa se um nome e e-mail forem informados na hora de instanciar o objeto (**Listagem 1**).

Neste momento o código em que criamos o objeto *$joao* já não funciona mais, pois agora o construtor da classe ao qual *$joao* pertence pede dois parâmetros para ser invocado. O correto pode ser visto no código que segue:

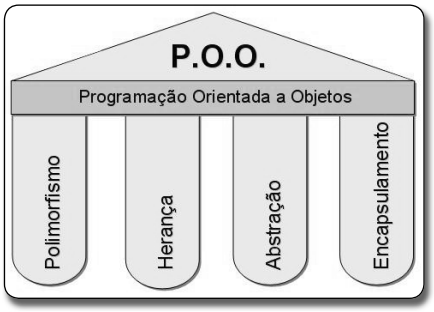

Figura 1. Pilares da programação orientada a objetos

```
$joao = new Pessoa('João das couves',
  'joao@joao.com.br');
```

Esta seria a forma correta de instanciarmos nosso objeto, pois agora seu construtor pede dois parâmetros e a palavra reservada *new* invoca o construtor, assim como no Delphi o *Classe.Create()* também invoca o método construtor.

Destrutores

Esta é uma novidade no PHP 5. Agora podemos definir métodos que são invocados quando um objeto é destruído: o destrutor. Assim como no caso dos construtores há um método exclusivo para ele. Basta inserirmos o método *__destruct* na definição de nossa classe.

Este método será invocado quando o objeto for destruído e isto acontece em duas situações no PHP. A primeira quando a referência ao objeto e destruída (*$joao = null*) e a outra é quando a requisição chega ao final. Como estamos desenvolvendo para *Web*, devemos lembrar que este ambiente é *stateless*, ou seja, não guarda por padrão valores de uma outra requisição, nós é que devemos nos valer de mecanismos para manter esses valores entre as requisições (*Session, Cookies. etc*).

Nota: Perceba que os métodos *constructor* e *destructor* são escritos com dois caracteres undercore no início da palavra. Isso é um padrão da linguagem.

A variável $this

Até agora está tudo muito bem, porém o nosso construtor não está nos valendo de nada, pois os parâmetros a ele passados não estão sendo utilizados, precisamos receber esses parâmetros e repassarmos para as propriedades *$nome* e *$email* e é ai que um conceito precisa ficar claro, pois isso difere muito do conceito que conhecemos no Delphi.

No Delphi, quando nos quereremos acessar uma propriedade de dentro da própria classe, basta digitar o nome desta ou mesmo chamar um método, pois o compilador sabe que aquela propriedade ou método pertence àquela classe, no PHP isso não acontece. Como PHP é um linguagem interpretada, não há necessidade de se declarar variáveis, muito menos declarar métodos e depois implementá-los. Um outro detalhe neste caso é que as variáveis criadas dentro de uma função têm como escopo a própria função, então observe o código da **Listagem 2**.

Como, neste caso, o interpretador irá entender que você não está criando uma variável e sim acessando a propriedade definida na classe? Não vai! O que ele vai fazer aqui e exatamente criar duas variáveis cujo escopo é a função e então atribuir a elas os valores passados como parâmetro, nada será feito com as propriedades do objeto, que ficarão intactas.

Para resolver este problema, o PHP traz em sua sintaxe OO a variável *$this*. Ela é usada sempre que você quiser acessar métodos e propriedades de dentro da própria classe, um conceito bem diferente da programação Win32 no Delphi. Seria redundante você escrever o código adiante para trocar o *Caption* de um *Label* em um formulário:

```
Self.Label1.Caption := 'Rodrigo Mourão';
```

Já no PHP isso não é redundante e sim necessário, porém no lugar do *Self* utilizamos o *$this*, mas isso não significa que o PHP não tenha o seu *Self*. O código correto para nosso construtor podemos ver na **Listagem 3**.

Agora sim o interpretador tem ciência de que você não esta criando duas variáveis novas e sim acessando propriedades do objeto onde o construtor está sendo chamado.

Listagem 2. Declaração das propriedades da classe

```
class Pessoa{
  var $nome;
  var $email;
  function __construct($nm, $em){
   $nome = $nm;
    $email = $em;
  }
}
```

Listagem 3. Uso do $this na atribuição dos valores das propriedades

```
class Pessoa{
  var $nome;
  var $email;
  function __construct($nm, $em){
    $this->nome = $nm;
    $this->email = $em;
  }
}
```

Listagem 4. Uso de private e public em propriedades e funções

```
class Pessoa{
  private $nome;
  private $email;
  public function __construct($nm, $em){
    $this->nome = $nm;
    $this->email = $em;
  }
}
```

Listagem 5. Aplicação de regras de negócio na classe

```
class Pessoa{
  private $nome;
  private $email;
  function __construct($nm, $em){
  $this->nome = strtoupper($nm);
  $this->email = strtolower($em);
}
```

Escopo de Visibilidade

Você deve ter notado a maneira pela qual criei as propriedades da nossa classe:

```
var $nome;
```

Esta era a maneira utilizada até a versão 4, todas as propriedades e métodos eram públicos por padrão e com isso visíveis de qualquer outra classe externa. A partir da versão 5 foram introduzidos operadores para definir escopo de visibilidade, e como no Delphi, se nenhum escopo for definido o padrão será público.

Agora suas classes podem ter métodos e propriedades privadas("private"), protegidas ("protected") e públicas ("public").

• *Private:* Métodos e propriedades visíveis apenas dentro da própria classe, não sendo passíveis de acessá-los externamente. Nesse caso esses valores são manipulados pela própria classe por métodos que são públicos;

• *Protected:* Métodos e propriedades que são visíveis dentro da própria classe e também das classes que descendem da mesma. No PHP utilizamos este escopo na maioria das vezes para declaramos métodos que serão sobrescritos numa classe descendente, porém não serão acessíveis de fora desta hierarquia;

• *Public:* Visíveis dentro da própria classe, classes descendentes e de qualquer outra classe externa a ela. Geralmente em OO encapsulamos o que varia e disponibilizamos métodos públicos para acessar estas propriedades;

Podemos nos valer desses escopos de visibilidade para adicionar mais uma regra a nossa classe *Pessoa* não permitindo que se acesse suas propriedades externamente e com isso garantir que o *nome* e *e-mail* sejam repassados somente pelo construtor e assim verificarmos se esses valores atendem as nossas especificações. Veja como podemos fazer para que nossas propriedades não sejam visíveis ao mundo externo visualizando o código da **Listagem 4**.

Nota do DevMan

O operador public seria desnecessário aqui visto que quando não informado o escopo padrão é público.

Desta maneira conseguimos validar a entrada de dados nas propriedades. Poderíamos aqui verificar se o e-mail passado é um endereço válido através de uma expressão regular ou mesmo colocar o nome passado pelo parâmetro em letras maiúsculas.

Herança

Um dos motivos que levam muitas pessoas a programarem OO é justamente a possibilidade de se reaproveitar códigos e regras através da herança. Claro que seu uso indiscriminado pode vir a prejudicar o projeto, pois quando se fala em herança fala-se também em acoplamento.

Herdar de uma classe significa possuir tudo aquilo que a classe *Pai* possui e poder acrescentar algo mais que seja pertinente a ela. Ainda no contexto da

nossa classe *Pessoa*, vamos fazer novas modificações no construtor para tratar os valores que serão armazenados em suas propriedades. A regra será nome em letras maiúsculas e e-mail em letras minúsculas (**Listagem 5**).

Não importa como os dados serão enviados ao construtor. Uma vez recebidos pela classe, ela se encarrega de efetuar os tratamentos antes de repassá-los às propriedades. Esse conceito é chamado de encapsulamento. E com isso criamos nossa regra.

Pensando na classe *Pessoa*, podemos criar duas novas classe *PessoaFisica* e *PessoaJuridica,* ambas com *nome* e *e-mail* e cada uma com uma propriedade exclusiva *CPF* e *CNPJ* respectivamente.

Não criaremos duas novas classes com todos os atributos e depois as mesmas regras de negócio para elas. É aí que entra a herança, uma vez que a classe *Pessoa* já detém algum dos atributos e as regras.

O que faremos aqui é simplesmente fazer com que as classes *PessoaFisica* e *PessoaJuridica* herdem de *Pessoa* e, conseqüentemente, possua as mesmas propriedades, métodos e regras de negócios. Feito isso acrescentaremos as propriedades pertinentes a cada uma exclusivamente. Essa é uma das vantagens de se trabalhar com herança, o que justifica seu uso em um modelo OO é justamente a possibilidade de se implementar o polimorfismo, uma vez que com herança temos acoplamento e nem sempre é isso que queremos em nossos códigos.

Entendido o que será feito basta agora fazê-lo em PHP. O PHP utiliza a palavra reservada *extends* para indicar o uso de herança no código. Na **Listagem 6** podemos ver a definição das classes *Pessoa*, *PessoaFisica* e *PessoaJuridica*.

Nota: O PHP 5 ainda não possui suporte a herança múltipla, nesse caso é necessário fazer uso de *Interfaces*, o que está fora do escopo deste artigo.

Aparentemente nossas classes estão prontas para uso, mas há um detalhe a se entender. Da maneira que está se o usuário que quiser instanciar umas das classes (*PessoaFisica)* ou (*PessoaJuridica),* deverá informar agora três parâmetros ao construtor, até aí tudo bem. Porém se reparar bem, somente o *CPF* ou *CNPJ* está sendo repassado para a classe, *nome* e *e-mail* não. O problema é que não podemos utilizar a variável *$this* para elas, pois essas propriedades foram declaradas como *private* ("privadas"), ou seja, mesmo pertencendo ao mesmo *Objeto* só serão acessíveis pela classe onde foram declaradas inicialmente, neste caso *Pessoa*.

Neste momento você deve estar pensando: "Simples vamos alterar o escopo para *Protected*, assim elas serão visíveis também nas classes descendentes". Em parte, se você pensou nesta solução, não está de todo errado, porém se fizermos isso estaremos burlando a regras que nós estabelecemos anteriormente. Isso significa que as regras criadas para colocar maiúsculas e minúsculas em nome e e-mail não se aplicarão a *PessoaFisica* e *PessoaJuridica*, pois foram implementadas no construtor da classe *Pessoa*, sendo assim temos que de alguma maneira invocar o construtor de *Pessoa* quando o construtor de *PessoaFisica* ou *PessoaJuridica* for chamado. Lembra *inherited* do Delphi? Pois bem.

Vale lembrar que, no PHP, se nenhum método construtor for declarado na classe *Filha*, o construtor do *Pai* é automaticamente chamado. Se sobrescrevermos o construtor ou qualquer outro método nos *Filhos,* o método na classe *Pai* deve ser invocado explicitamente.

A solução para o impasse anterior é justamente utilizar o operador *Parent*. Sua função é exatamente invocar um método na classe imediatamente acima na estrutura hierárquica de suas classes e com isso garantir que as regras e códigos sejam executados. Observe agora como ficaram nossas classes com esta implementação. (**Listagem 7**)

Agora sim podemos ter certeza que nossa regra de negócio será implementada em ambas as classes descendentes. Repare que invocamos o construtor de *Pessoa* e repassamos os parâmetros que o mesmo necessita, *nome* e *e-mail*. O terceiro parâmetro é atribuído a própria

Listagem 6. Declaração das classes Pessoa, PessoaFisica e PessoaJuridica

```
class Pessoa{
  private $nome;
  private $email;
  function __construct($nm, $em){
    $this->nome = strtoupper($nm);
    $this->email = strtolower($em);
  }
class PessoaFisica extends Pessoa{private $cpf;
  function __construct($nm, $em, $cpf){
   $this->cpf = $cpf;
  }
}
class PessoaJuridica extends Pessoa{private $cnpj;
  function __construct($nm, $em, $cnpj){
   $this->cnpj = $cnpj;
  }
}
```

Listagem 7. Implentação do Parent

```
class Pessoa{
  private $nome;
  private $email;
  function __construct($nm, $em){
    $this->nome = strtoupper($nm);
    $this->email = strtolower($em);
  }
}
class PessoaFisica extends Pessoa{private $cpf;
   function __construct($nm, $em, $cpf){
    parent::__construct($nm, $em);
    $this->cpf = $cpf;
   }
}
class PessoaJuridica extends Pessoa{
  private $cnpj;
    function __construct($nm, $em, $cnpj){
    parent::__construct($nm, $em);
  $this->cnpj = $cnpj;
  }
}
```

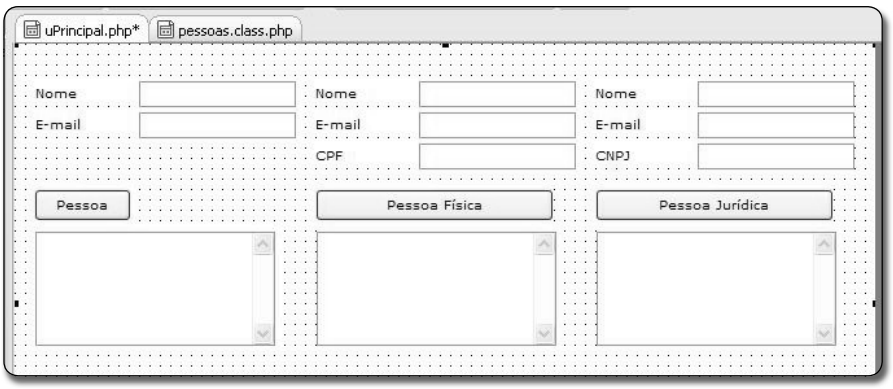

Figura 2. Exemplo de formulário para testes de classes

Listagem 8. Declaração de um formulário no Delphi for PHP

```
<?php
  //Includes
  require_once("vcl/vcl.inc.php");
  use_unit("forms.inc.php");
  use_unit("extctrls.inc.php");
  use_unit("stdctrls.inc.php");

  //Class definition
  class Unit3 extends Page{
    public $Button1 = null;
    public $Label1 = null;
    public $Edit1 = null;
  }
  global $application;
  global $Unit3;

  //Creates the form
  $Unit3 = new Unit3($application);

  //Read from resource file
  $Unit3->loadResource(__FILE__);

  //Shows the form
  $Unit3->show();
?>
```

Listagem 9. Código de classes do arquivo pessoas.class.php

```
<?php
  class Pessoa{
    private $nome;
    private $email;
    function __construct($nm, $em){
      $this->nome = strtoupper($nm);
      $this->email = strtolower($em);
    }
    function getnome(){
      return $this->nome;
    }
    function getemail(){
      return $this->email;
    }
  }
  class PessoaFisica extends Pessoa{
    private $cpf;
    function __construct($nm, $em, $cpf){
      parent::__construct($nm, $em);
      $this->cpf = $cpf;
    }
    function getcpf(){
      return $this->cpf;
    }
  }
  class PessoaJuridica extends Pessoa{
    private $cnpj;
    function __construct($nm, $em, $cnpj){
      parent::__construct($nm, $em);
      $this->cnpj = $cnpj;
    }
    function getcnpj(){
      return $this->cnpj;
    }
  }
?>
```

classe que o detém, mas lembre-se, o objeto criado em memória é um só, uma pessoa física com nome, e-mail e CPF ou uma pessoa jurídica com nome, e-mail e CNPJ, porém compartilhando regras em comum que foram implementadas em uma classe ascendente.

Acredito que agora o código da **Listagem 8** faça mais sentido para você. Sabemos agora que quando adicionamos um novo formulário em um projeto no Delphi for PHP estamos na verdade adicionando uma classe *Unit1, Unit2, Unit3* etc que herda da classe *Page* e que possui tantas propriedades quantos componentes adicionarmos no formulário. Se verificar no código em seguida, veremos que *Unit3* está sendo instanciada e repassada para a variável *$Unit3*, ou seja, criando um novo objeto.

```
$Unit3 = new Unit3($application);
```

O construtor do formulário, como no Delphi, pede um parâmetro que na verdade é o *Owner* ("proprietário"). Aqui está sendo passado a variável *global $application*. Agora se entende o porquê de se colocar o *$this* sempre que queremos manipular um componente via código no Delphi for PHP. Todos os componentes são propriedades da classe por isso usamos o *$this*.

Desenvolvendo uma aplicação teste

Bem, nos resta testar o nosso modelo para ver se tudo está funcionando conforme o planejado. Abra o Delphi for PHP, vá em *File|New>Application* e salve a *Unit* principal como "uPrincipal.php". Em seguida salve o projeto como "poo.phprj". Em seguida crie uma nova *Unit* em *File|New>Unit* onde faremos a codificação de nossas classes. Salve-a como "pessoas.class.php" no mesmo diretório da aplicação e digite o código da **Listagem 9**.

Tudo que vimos até agora foi acrescentado a nossa *Unit pessoa.class.php*, com exceção dos métodos *getnome(), getemail(), getcpf()* e *getcnpj*. Isso porque nossas propriedades são privadas, então precisamos de um método público para auxiliar no retorno do conteúdo das mesmas para podermos nos certificar que nossas regras estão funcionando bem.

Agora no formulário principal vamos colocar alguns controles para que possamos interagir com nossas classes. Desenhe um formulário como no exemplo mostrado da **Figura 2**.

Insira oito componentes *Edit* da paleta *Standard*. Modifique o *Name* de cada um para "EdtNomeP", "EdtEmailP", "EdtNomePF", "EdtEmailPF", "EdtCPF", "EdtNomePJ", "EdtEmailPJ" e "EdtCNPJ". Coloque também três *Button's* com os *Captions* conforme a **Figura 2** e *Name's* "BtnPessoa", "BtnPessoaPF" e "BtnPessoaPJ". Abaixo dos botões vão três componentes *Memo* com os *Name's* "mmP", "mmPF" e "mmPJ", da esquerda para a direita.

O que temos são *Edit's* para inserirmos os valores para as propriedades de cada classe e um *Memo* para publicar os valores. Como nossas classes foram definidas em outra *Unit*, precisamos fazer referência a ela em nosso formulário. Para isso use a opção *File>Use Unit* ou pressione *Alt + F12* e selecione *pessoas.class.php*, assim como no Delphi.

O último passo é codificar cada botão do exemplo. Basta clicar duas vezes no botão para que sejamos enviados ao *Code Editor* do Delphi for PHP. Digite o código da **Listagem 10** no botão *BtnPessoa*. Repita esses passos para *BtnPessoaPF* (**Listagem 11**) e *BtnPessoaPJ* (**Listagem 12**).

A rotina é praticamente a mesma para os três casos. Primeiro criamos uma variável para receber a instância da nossa classe. Em seguida no construtor passamos os parâmetros que cada uma delas requer e neste caso passamos o conteúdo dos respectivos *Edit's*. Neste ponto nossa regra já foi aplicada, pois é no construtor que executados nosso código e é ai que entra em cena as funções *getnome(), getemail(), getcpf()* e *getcnpj()*.

São através delas que recuperamos os valores devidamente formatados e publicamos nos controles *Memos* para que possamos verificar o conteúdo das propriedades. Execute a aplicação, preencha os *Edit's* e veja o resultado (**Figura 3**).

Observe que a regra estabelecida em *Pessoa* foi aplicada também em *PessoaFisica* e *PessoaJuridica*. Vale lembrar que qualquer alteração feita na mesma também ira refletir em ambas, por exemplo, a validação do e-mail. Você pode incluir uma rotina para validar o endereço antes de atribuí-lo e fará isso na classe *Pessoa*, esta regra automaticamente testará disponível nas demais.

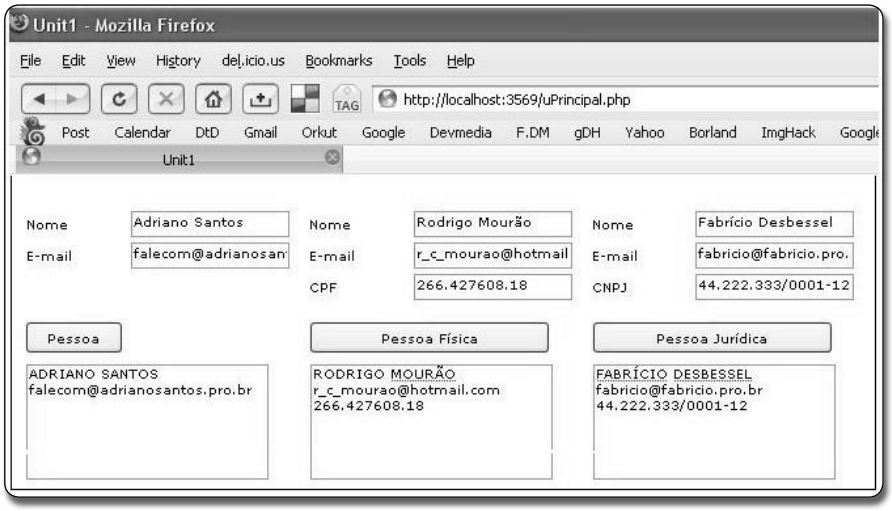

Figura 3. Aplicação em execução, já com os testes das classes

Listagem 10. Código do BtnPessoa

```
function BtnPessoaClick($sender, $params)
{
  $p = new Pessoa($this->EdtNomeP->Text,
    $this->EdtEmailP->Text);
  $this->mmP->Add($p->getnome());
  $this->mmP->Add($p->getemail());
}
```

Listagem 11. Código do BtnPessoaPF

```
function BtnPessoaPFClick($sender, $params)
{
  $f = new PessoaFisica($this->EdtNomePF->Text,
    $this->EdtEmailPF->Text, $this->EdtCPF->Text);
  $this->mmPF->Add($f->getnome());
  $this->mmPF->Add($f->getemail());
  $this->mmPF->Add($f->getcpf());
}
```

Listagem 12. Código do BtnPessoaPJ

```
function BtnPessoaPJClick($sender, $params)
{
  $j = new PessoaJuridica($this->EdtNomePJ->Text,
    $this->EdtEmailPJ->Text, $this->EdtCNPJ->Text);
  $this->mmPJ->Add($j->getnome());
  $this->mmPJ->Add($j->getemail());
  $this->mmPJ->Add($j->getcnpj());
}
```

Conclusão

Muito ainda há o que se falar em relação a OO no Delphi for PHP, mas isso fica para os próximos artigos onde abordaremos tópicos avançados como *polimorfismo*, *classes abstratas*, *métodos* e *propriedades estáticas*, *interfaces* e *Design Patterns*. Espero que esta pequena introdução já lhe ajude nas tarefas simples do seu dia-a-dia de desenvolvedor. OO é mais do que uma "maneira" de se programar, é uma filosofia de desenvolvimento onde temos total controle sobre aquilo que estamos criando.

Olhe a sua volta, enumere cinco linguagens de programação que conhece, agora das cinco quais são Orientadas a Objetos? Veja a importância de tal metodologia. Não se trata de Delphi, PHP, Java, etc. Estamos falando de Orientação a Objetos, algo que é comum a todas essas.

Eu sou Rodrigo Carreiro e pela sua atenção muito obrigado! ●