PHP

Programando com Orientação a Objetos

Pablo Dall'Oglio

Novatec

Editor: Rubens Prates
Assistente editorial: Priscila A. Yoshimatsu
Editoração eletrônica: Carolina Kuwabata
Revisão gramatical: Jussara Rodrigues Gomes
Capa: Pablo Dall'Oglio e Rodolpho Lopes

ISBN: 978-85-7522-465-6

Histórico de impressões:

Novembro/2015	Terceira edição
Abril/2009	Segunda edição (ISBN: 978-85-7522-200-3)
Setembro/2007	Primeira edição (ISBN: 978-85-7522-137-2)

Novatec Editora Ltda.
Rua Luís Antônio dos Santos 110
02460-000 – São Paulo, SP – Brasil
Tel.: +55 11 2959-6529
Email: novatec@novatec.com.br
Site: www.novatec.com.br
Twitter: twitter.com/novateceditora
Facebook: facebook.com/novatec
LinkedIn: linkedin.com/in/novatec

PHP

Programando com Orientação a Objetos

3ª edição

CAPÍTULO 1

Introdução ao PHP

A vida é uma peça de teatro que não permite ensaios...
Por isso, cante, ria, dance, chore e viva intensamente cada momento de sua vida,
antes que a cortina se feche e a peça termine sem aplausos...

Charles Chaplin

Ao longo deste livro utilizaremos diversas funções, comandos e estruturas de controle básicos da linguagem PHP, que apresentaremos neste capítulo. Conheceremos as estruturas básicas da linguagem, suas variáveis e seus operadores e também um conjunto de funções para manipulação de strings, arquivos, arrays, bancos de dados, entre outros.

1.1 O que é o PHP?

A linguagem de programação PHP, que no início significava Personal Home Page Tools, foi criada no outono de 1994 por Rasmus Lerdorf. Essa linguagem era formada por um conjunto de scripts escritos em linguagem C, voltados à criação de páginas dinâmicas que Rasmus utilizava para monitorar o acesso ao seu currículo na internet. Com o tempo, mais pessoas passaram a utilizá-la e Rasmus adicionou vários recursos, como a interação com bancos de dados. Em 1995, o código-fonte do PHP foi liberado, e com isso mais desenvolvedores puderam se juntar ao projeto. Naquela época, por um breve período de tempo, o PHP foi chamado de FI (Forms Interpreter).

O PHP passou por várias reescritas de código ao longo do tempo e nunca parou de conquistar novos adeptos. Uma segunda versão foi lançada em novembro de 1997, sob o nome PHP/FI 2.0. Naquele momento, aproximadamente 60 mil domínios, ou 1% da internet, já utilizavam PHP, que era mantido principalmente por Rasmus.

No mesmo ano Andi Gutmans e Zeev Suraski, dois estudantes que utilizavam essa linguagem em um projeto acadêmico de comércio eletrônico, resolveram cooperar com Rasmus para aprimorar o PHP. Para tanto, reescreveram todo o código-fonte, com base no PHP/FI 2, dando início assim ao PHP 3, disponibilizado oficialmente em junho de 1998. Entre as principais características do PHP 3 estavam a extensibilidade, a possibilidade de conexão com vários bancos de dados, novos protocolos, uma sintaxe mais consistente, suporte à orientação a objetos e uma nova API, que possibilitava a criação de novos módulos e que acabou por atrair vários desenvolvedores ao PHP. No final de 1998, o PHP já estava presente em cerca de 10% dos domínios da internet. Nessa época o significado da sigla PHP mudou para PHP: Hypertext Preprocessor, retratando, assim, a nova realidade de uma linguagem com propósitos mais amplos.

No inverno de 1998, Zeev e Andi começaram a trabalhar em uma reescrita do núcleo do PHP, tendo em vista melhorar seu desempenho e sua modularidade em aplicações complexas. O nome foi rebatizado para Zend Engine (Zeev + Andi). O PHP 4, já baseado nesse mecanismo, foi lançado em maio de 2000, trazendo melhorias como seções, suporte a diversos servidores web, além da abstração de sua API, permitindo inclusive ser utilizado como linguagem para shell script.

Apesar de todos os esforços, o PHP ainda necessitava de maior suporte à orientação a objetos. Esses recursos foram trazidos pelo PHP 5, após um longo período de desenvolvimento que culminou com sua disponibilização oficial em julho de 2004. Ao longo de mais de uma década, o PHP vem adicionando mais e mais recursos e se consolida ano após ano como uma das linguagens de programação orientadas a objetos que mais crescem no mundo. Estima-se que o PHP seja utilizado em mais de 80% dos servidores web existentes, tornando-a disparadamente a linguagem mais utilizada para desenvolvimento web. Ao longo do livro, veremos esses recursos empregados em exemplos práticos.

Fonte: PHP Group (http://php.net/history.php)

1.2 Um programa PHP

1.2.1 Estrutura do código-fonte

Normalmente um programa PHP tem a extensão *.php*. Entretanto não é incomum encontrarmos extensões como *.class.php* para armazenar classes ou *.inc.php,* em projetos mais antigos, para representar simplesmente arquivos a serem incluídos.

O código de um programa escrito em PHP deve iniciar com `<?php`, veja:

```
<?php
// código;
// código;
// código;
```

Observação: a finalização da maioria dos comandos se dá por ponto e vírgula (;).

1.2.2 Configuração

Ao iniciarmos com o PHP, é importante sabermos configurar o ambiente. Nesse sentido a função `phpinfo()` é muito importante, pois ela apresenta uma tabela com a configuração atual do PHP, como níveis de erro, extensões instaladas, entre outros.

```
<?php
phpinfo();
```

O arquivo de configuração do PHP é o *php.ini*, cuja localização varia conforme a instalação utilizada. Mas sua localização pode ser descoberta pela função `phpinfo()`, conforme visto na figura 1.1. A partir da localização do *php.ini* podemos realizar algumas configurações iniciais. No exemplo a seguir definimos o `timezone` (usado em funções com cálculo de tempo) habilitando o `display_errors` (para que todos os erros que ocorrerem sejam exibidos), o `log_errors` (para que os erros sejam registrados em um arquivo), definindo o arquivo de registro de erros com o `error_log`, bem como ligando simplesmente todos os níveis de erro, por meio do `error_reporting`. Essas definições são voltadas mais para um ambiente de desenvolvimento. Em produção, geralmente desligamos o `display_errors`.

```
date.timezone = America/Sao_Paulo
display_errors = On
log_errors = On
error_log = /tmp/php_errors.log
error_reporting = E_ALL
```

Resultado:

```
O Leão roeu a roupa do rei de Roma
```

1.11 Manipulação de arrays

A manipulação de arrays no PHP é, sem dúvida, um dos recursos mais poderosos dessa linguagem. O programador que assimilar bem esta parte terá muito mais produtividade no seu dia a dia. Isso porque os arrays no PHP servem como verdadeiros contêineres, armazenando números, strings, objetos, entre outros, de forma dinâmica. Além disso, o PHP nos oferece uma gama de funções para manipulá-los, as quais serão vistas a seguir.

1.11.1 Criando um array

Arrays são acessados mediante uma posição, como um índice numérico. Para criar um array, pode-se utilizar a função `array([chave =>] valor , ... )` ou a sintaxe resumida entre [].

```
$cores = array(0=>'vermelho', 1=>'azul', 2=>'verde', 3=>'amarelo');
```

ou

```
$cores = array('vermelho', 'azul', 'verde', 'amarelo');
```

ou

```
$cores = [ 'vermelho', 'azul', 'verde', 'amarelo' ];
```

Outra forma de criar um array é simplesmente adicionando-lhe valores com a seguinte sintaxe:

```
$nomes[] = 'maria';
$nomes[] = 'joão';
$nomes[] = 'carlos';
$nomes[] = 'josé';
```

De qualquer forma, para acessar o array indexado, basta indicar o seu índice entre colchetes:

```
echo $cores[0];  // resultado = vermelho
echo $cores[1];  // resultado = azul
echo $cores[2];  // resultado = verde
echo $cores[3];  // resultado = amarelo
```

1.12.2 Exemplos

Para exemplificar o acesso ao banco de dados, criaremos dois programas. O primeiro deles será responsável por inserir dados em um banco de dados PostgreSQL; o segundo irá listar os resultados inseridos pelo primeiro programa. Para criar o banco de dados utilizado nos exemplos, certifique-se de que você tem o PostgreSQL instalado em sua máquina e utilize os seguintes comandos:

```
# createdb livro
# psql livro
livro=# CREATE TABLE famosos (codigo integer, nome varchar(40));
CREATE TABLE
livro=# \q
```

Observação: é possível criar a base de dados utilizando o software PgAdmin (www.pgadmin.org), que apresenta uma interface totalmente gráfica para gerenciar o banco de dados.

Neste primeiro exemplo o programa se conectará ao banco de dados local `livro`, localizado em `localhost`, com o usuário `postgres`. Em seguida irá inserir dados de algumas pessoas famosas na base de dados. Por fim, ele fechará a conexão ao banco de dados.

pgsql_insere.php

```
<?php
// abre conexão com Postgres
$conn = pg_connect("host=localhost port=5432 dbname=livro user=postgres password=");

// insere vários registros
pg_query($conn, "INSERT INTO famosos (codigo, nome) VALUES (1, 'Érico Veríssimo')");
pg_query($conn, "INSERT INTO famosos (codigo, nome) VALUES (2, 'John Lennon')");
pg_query($conn, "INSERT INTO famosos (codigo, nome) VALUES (3, 'Mahatma Gandhi')");
pg_query($conn, "INSERT INTO famosos (codigo, nome) VALUES (4, 'Ayrton Senna')");
pg_query($conn, "INSERT INTO famosos (codigo, nome) VALUES (5, 'Charlie Chaplin')");
pg_query($conn, "INSERT INTO famosos (codigo, nome) VALUES (6, 'Anita Garibaldi')");
pg_query($conn, "INSERT INTO famosos (codigo, nome) VALUES (7, 'Mário Quintana')");

// fecha a conexão
pg_close($conn);
```

No próximo exemplo, o programa se conectará ao mesmo banco de dados do exemplo anterior, chamado `livro`, localizado em `localhost`. Em seguida ele irá selecionar código e nome de famosos existentes nesse banco de dados, exibindo-os em tela.

CAPÍTULO 2

Fundamentos de orientação a objetos

No que diz respeito ao desempenho, ao compromisso, ao esforço, à dedicação, não existe meio termo. Ou você faz uma coisa bem-feita ou não faz.

Ayrton Senna

A orientação a objetos é um paradigma que representa uma filosofia para construção de sistemas. Em vez de construir um sistema formado por um conjunto de procedimentos e variáveis nem sempre agrupadas de acordo com o contexto, como se fazia em linguagens estruturadas (Cobol, Clipper, Pascal), na orientação a objetos utilizamos uma ótica mais próxima do mundo real. Lidamos com objetos: estruturas que carregam dados e comportamento próprio, além de trocarem mensagens entre si com o objetivo de formar algo maior, um sistema.

Orientação a objetos é uma abordagem para concepção de sistemas, um paradigma de programação. Atualmente é a abordagem mais utilizada para concepção de sistemas. Entretanto nem sempre foi assim. Para entender o que é a orientação a objetos e como ela é utilizada, primeiro é importante compreender como era o desenvolvimento antes de a orientação a objetos se tornar o paradigma vigente.

2.1 Programação procedural

A abordagem mais usada durante muito tempo foi a programação procedural. Esta ocorre quando o programa é construído com base em um conjunto de procedimentos (*procedures*). Um procedimento (que no PHP pode ser construído com uma *function*), basicamente é uma unidade de código que pode ser reaproveitada em diversas situações. Como já vimos como declarar funções em PHP, você já compreende que uma função é uma unidade de código que recebe parâmetros de entrada, realiza determinado procedimento e devolve um retorno para quem a chamou.

Conceitualmente existe uma associação entre um produto e seu fabricante, em que um produto está relacionado a um fabricante e, por sua vez, um fabricante pode fabricar diferentes produtos; a figura 2.5 procura ilustrar esse relacionamento pelo diagrama de classes. No exemplo a seguir criaremos uma associação entre as classes `Produto` e `Fabricante`. A classe `Fabricante` terá como atributos: `nome`, `endereco` e `documento`. Em seu método construtor, ela receberá via parâmetro esses dados para já ser inicializada com conteúdo. Para poupar espaço aqui, criaremos somente o método *getter* `getNome()`.

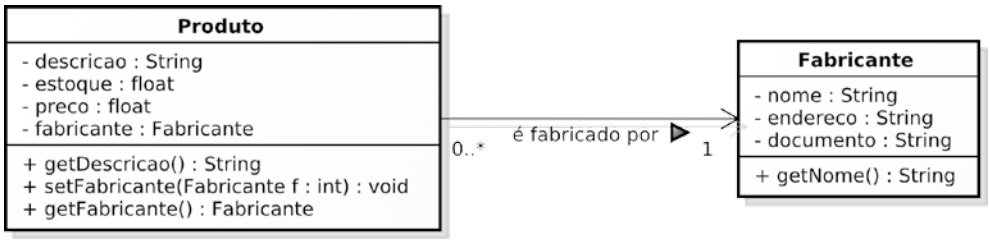

Figura 2.5 – Relacionamento de associação.

classes/Fabricante.php

```
<?php
class Fabricante {
    private $nome;
    private $endereco;
    private $documento;

    public function __construct( $nome, $endereco, $documento ) {
        $this->nome      = $nome;
        $this->endereco  = $endereco;
        $this->documento = $documento;
    }

    public function getNome() {
        return $this->nome;
    }
}
```

A associação entre as classes `Produto` e `Fabricante` se estabelece a partir da classe `Produto`. A classe `Produto` terá como atributos: `descricao`, `estoque`, `preco` e `fabricante`. Enquanto os primeiros são atributos escalares (variáveis escalares são as que contêm `integer`, `float`, `string` ou `boolean`), o atributo `fabricante` é, na verdade, um objeto, ou seja, ele apontará para um objeto da classe `Fabricante`. Para estabelecer o vínculo (associação) entre as duas classes, criaremos os métodos `setFabricante()`

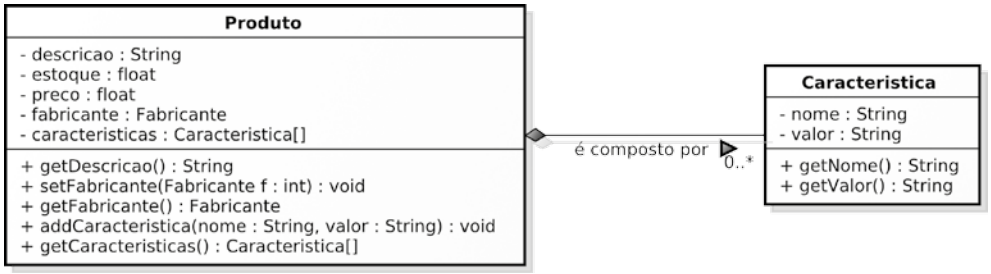

Figura 2.6 – Relacionamento de composição.

Inicialmente será criada a classe `Caracteristica`. Uma característica terá apenas dois atributos: nome e valor. São exemplos de nomes: "cor", "peso", "tamanho" e "potência". São exemplos de valores: "branco", "20 kg", "200 watts" e assim por diante.

classes/Caracteristica.php

```
<?php
class Caracteristica {
    private $nome;
    private $valor;

    public function __construct( $nome, $valor ) {
        $this->nome   = $nome;
        $this->valor  = $valor;
    }

    public function getNome() {
        return $this->nome;
    }

    public function getValor() {
        return $this->valor;
    }
}
```

Em seguida, alteraremos a classe `Produto` para que ela tenha os métodos para compor características. Inicialmente precisaremos declarar um atributo `$caracteristicas`, que será um vetor interno que armazenará uma ou mais instâncias contendo objetos da classe `Caracteristica`. Escreveremos o método `addCaracteristica()`, que receberá os parâmetros `$nome` e `$valor` e criará internamente um objeto da classe `Caracteristica` para armazenar esses dois dados. O objeto criado será adicionado ao vetor (`$this->caracteristicas`). Já o método `getCaracteristicas()` simplesmente retornará o vetor de características.

ou seja, somente poderá receber objetos da classe `Produto`. A classe `Cesta` contará também com o método `getItens()`, que por sua vez retornará os itens da cesta.

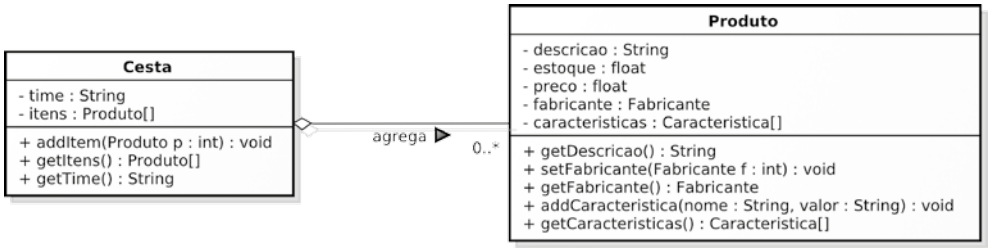

Figura 2.7 – Relacionamento de agregação.

classes/Cesta.php

```
<?php
class Cesta {
    private $time;
    private $itens;

    public function __construct( ) {
        $this->time = date('Y-m-d H:i:s');
        $this->itens = array();
    }

    public function addItem( Produto $p ) {
        $this->itens[] = $p;
    }

    public function getItens() {
        return $this->itens;
    }

    public function getTime() {
        return $this->time;
    }
}
```

Agora iremos demonstrar o relacionamento de agregação na prática. Para tal, inicialmente vamos importar as classes necessárias (`require_once`). Em seguida instanciaremos um objeto da classe `Cesta`. Depois disso iremos agregar por três vezes os objetos da classe `Produto` dentro da `Cesta` (`$c1`) por meio do método `addItem()`. Veja que, diferentemente da composição, na agregação o método que cria as "partes" já recebe as instâncias prontas, externas ao escopo da classe. Assim no escopo principal do programa temos quatro objetos em memória (`$c1, $p1, $p2, $p3`) e,

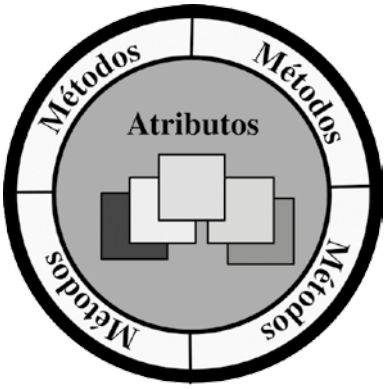

Figura 2.9 – Encapsulamento.

Para atingir o encapsulamento, uma das formas é definir a visibilidade das propriedades e dos métodos de um objeto. A visibilidade define a forma como essas propriedades devem ser acessadas. Existem três formas de acesso:

Visibilidade	Descrição
`public`	Membros declarados como `public` poderão ser acessados livremente a partir da própria classe em que foram declarados, a partir de classes descendentes e a partir do programa que faz uso dessa classe (manipulando o objeto em si). Na UML, simbolizamos com um + na frente da propriedade.
`private`	Membros declarados como `private` somente podem ser acessados dentro da própria classe em que foram declarados. Não poderão ser acessados a partir de classes descendentes nem a partir do programa que faz uso dessa classe (manipulando o objeto). Na UML, simbolizamos com um - na frente da propriedade.
`protected`	Membros declarados como `protected` somente podem ser acessados dentro da própria classe em que foram declarados e a partir de classes descendentes, mas não poderão ser acessados a partir do programa que faz uso dessa classe (manipulando o objeto em si). Na UML, simbolizamos com um caractere # na frente da propriedade.

Observação: a visibilidade foi introduzida pelo PHP5 e, para manter compatibilidade com versões anteriores, quando a visibilidade de uma propriedade ou de um método não for definida, automaticamente será tratada como `public`.

2.11.1 Public

Demonstrar a visibilidade `public` é uma tarefa simples, pois um atributo declarado como `public` pode ser alterado de qualquer parte, ou seja, tanto de dentro da classe

biblioteca e encontrou a maravilhosa PHPMailer. Entretanto você percebeu que os métodos da PHPMailer (`IsSMTP`, `MsgHTML`, `AddAttachment`) eram diferentes dos métodos da biblioteca que você utilizava (`setUseSmtp`, `setHtmlBody`, `addAttach`) e percebeu também que iria ter trabalho para adaptar todas as chamadas, visto que existiam diversos pontos do seu sistema que já enviavam email.

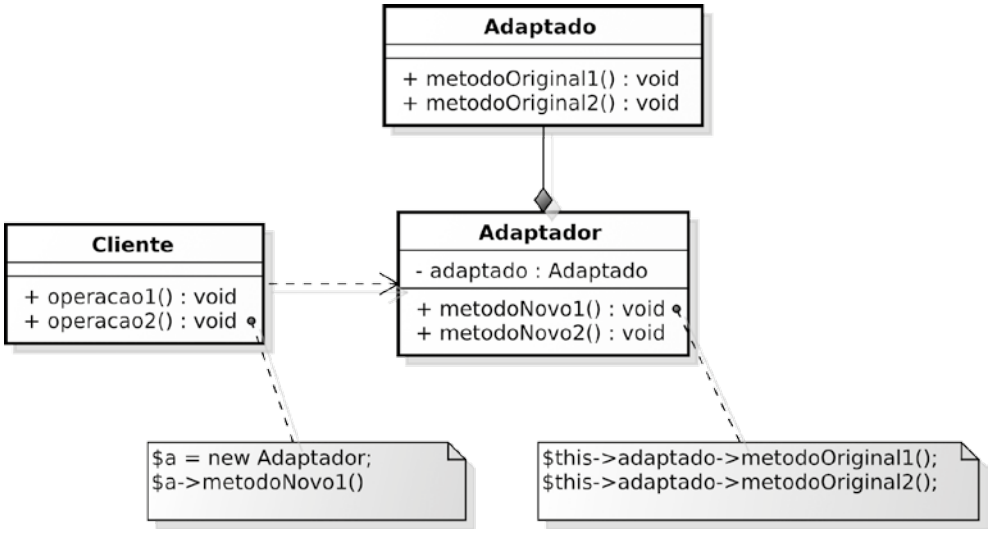

Figura 2.14 – Estrutura de um Adapter.

Para resolver esse problema, podemos usar o Design Pattern Adapter para converter os métodos da PHPMailer para o formato de chamada de métodos que você espera em sua aplicação. Vamos construir a classe `PHPMailerAdapter`, que se trata de uma camada fina ao redor da PHPMailer para converter o formato de chamada de métodos. Assim `IsSMTP()` se transformará em `setUseSmtp()`, `MsgHTML()` se transformará em `setHtmlBody()` e `AddAttachment()` se transformará em `addAttach()`, entre outros. A seguir temos a classe `PHPMailerAdapter`. Trata-se de uma classe funcional, que só precisa da PHPMailer para funcionar.

classes/PHPMailerAdapter.php

```
<?php
class PHPMailerAdapter {
    private $pm;

    public function __construct() {
        $this->pm = new PHPMailer;
        $this->pm-> CharSet = 'utf-8';
    }
```

CAPÍTULO 3

Tópicos especiais em orientação a objetos

A melhor maneira de prever o futuro é criá-lo.

Peter Drucker

A orientação a objetos é um assunto bastante extenso para ser tratado em apenas um capítulo. Por isso, no capítulo anterior estudamos como implementar no PHP os principais conceitos da orientação a objetos: encapsulamento, herança, associação, agregação, composição, polimorfismo, abstração e interfaces. Neste capítulo vamos abordar uma série de tópicos especiais que podem ser vistos de maneira isolada: métodos mágicos, exceções, SPL, Reflection, Traits, Namespaces e PDO. Muitos desses tópicos diferenciam mais o PHP de outras linguagens orientadas a objetos por oferecer recursos específicos não encontrados em outras linguagens da mesma forma ou com a mesma sintaxe.

3.1 Tratamento de erros

Existem diversas formas de realizar tratamento de erros em PHP. Nesta seção veremos as formas mais comuns utilizadas, desde a simplória função `die()` até o refinado tratamento de exceções.

3.1.1 O cenário proposto

Antes de estudarmos algumas formas de tratamento de erros no PHP, vamos propor um cenário. Esse cenário consiste em demonstrar uma operação que pode resultar em falha em algum ponto do processo, e essa falha precisa ser tratada. Dessa forma vamos criar uma classe que efetua a leitura de arquivos no formato

Observação: poderíamos fazer o método em questão retornar diferentes valores para ter um controle mais apurado (ex.: 1, 2, 3). Porém, isso não será necessário visto que o PHP tem o tratamento de exceções.

3.1.4 Tratamento de exceções

Vimos que a função `die()` oferece uma abordagem bastante simplória ao abortar a execução do programa de maneira abrupta. Vimos também que o retorno de flags permite que o programa prossiga a execução, porém oferece um controle ainda limitado quanto às mensagens de erro. Poderíamos melhorar o retorno de flags para retornar códigos de erro. Porém essa estratégia não é indicada, pois em alguns casos nosso método terá de retornar valores válidos que podem conflitar com os códigos de erro.

Para oferecer um tratamento de erros mais refinado, o PHP implementa o conceito de tratamento de exceções de maneira bastante similar a outras linguagens como C++ ou Java. Tratamento de exceções é um processo dividido em duas etapas: emitir e tratar uma exceção. Por um lado, sempre que uma rotina passar por um ponto de falha, deverá emitir uma exceção. Por outro lado, a rotina que está chamando e aguardando a resposta deverá tratar essa exceção. Uma exceção é uma ocorrência de programação que se sobressai à execução normal do programa, parando a execução no momento em que ela é emitida e continuando a execução diretamente para o ponto em que ela é tratada.

Para entendermos como as exceções funcionam, primeiro precisamos compreender que uma exceção em PHP é um objeto especial, derivado da classe `Exception`, que contém alguns métodos para informar ao programador um relato do que aconteceu. A seguir você confere esses métodos:

Método	Descrição
`getMessage()`	Retorna a mensagem de erro.
`getCode()`	Retorna o código de erro.
`getFile()`	Retorna o arquivo no qual ocorreu o erro.
`getLine()`	Retorna a linha na qual ocorreu o erro.
`getTrace()`	Retorna um array com as ações até o erro.
`getTraceAsString()`	Retorna as ações em forma de string.

Utilizar o tratamento de exceções não é muito complicado. Em um primeiro momento reescrevemos nosso método `parse()`, que é o local em que uma falha pode ocorrer, e substituímos o tratamento de erros existente por lançamentos

```
                [line] => 10
                [function] => parse
                [class] => CSVParser
                [type] => ->
                [args] => Array ( )
            )
    )
Arquivo não encontrado
```

3.2 Métodos mágicos

Alguns fatores que tornam o PHP uma linguagem tão atraente são a sua flexibilidade e a sua dinamicidade. Já vimos até o momento o quão fácil é trabalhar com arrays e também objetos. Mas as facilidades não param por aí. No que diz respeito à orientação a objetos, o PHP implementa uma série de métodos que realizam interceptação de operações, também conhecidos por "métodos mágicos". Esse termo não é à toa. Existem diversos métodos mágicos no PHP e todos eles iniciam com __, como: `__construct()`, `__destruct()`, `__call()`, `__get()`, `__set()`, `__isset()`, `__unset()`, entre outros. Enquanto o método `__construct()` é executado automaticamente durante a construção do objeto e o método `__destruct()` é executado automaticamente durante a destruição do objeto, outros métodos são executados automaticamente em algumas circunstâncias. Veremos a seguir exemplos de uso para cada um dos principais métodos mágicos.

3.2.1 Introdução aos métodos mágicos

Os métodos `__set()`, `__get()`, `__isset()` e `__unset()` são utilizados para definir um comportamento para o objeto sempre que houver uma tentativa de acesso a uma propriedade não acessível (`private`, não existente etc.). Para entender melhor os "comportamentos mágicos", vamos começar com uma classe `Titulo` na qual todos os seus atributos são `private`. Como ela não oferece nenhum método de acesso, logo esses atributos são inacessíveis. Se houver uma tentativa de leitura do atributo valor, um `Fatal Error` ocorrerá como pode ser percebido a seguir.

magic_intro.php

```
<?php
class Titulo {
    private $dt_vencimento;
```

```
    nome=>Mato Grosso
    capital=>Cuiabá
```

3.4 Manipulação de XML com DOM

Ao utilizarmos a SimpleXML vimos que ela oferece uma API simples para acessar documentos XML simples e percorrer estruturas. Porém ela tem algumas limitações quanto à manipulação de documentos. A SimpleXML contém funcionalidades limitadas para percorrer os elementos. Por outro lado, o PHP oferece a DOM, que é uma implementação em conformidade com os padrões do W3C e portanto apresenta consistência entre diferentes linguagens de programação, além de uma quantidade muito maior de funcionalidades que permitem diferentes meios de acesso aos nodos e formas de rearranjá-los conforme a necessidade. Apesar de a DOM ser mais poderosa, a SimpleXML atende bem a maioria dos casos em que a demanda por leitura e manipulação de XML é pequena.

3.4.1 Leitura de conteúdo

Inicialmente vamos ler um arquivo XML usando a implementação DOM. Para tal, vamos declarar o arquivo a seguir que contém uma lista de bases de dados. Cada base de dados tem um `ID`, que é um atributo da tag, e tags filhas, como `name`, `host`, `type` e `user`. Precisaríamos de mais informações para conectar em uma base de dados, mas aqui o foco não é esse.

bases.xml

```
<bases>
    <base id="1">
        <name>teste</name>
        <host>192.168.0.1</host>
        <type>mysql</type>
        <user>mary</user>
    </base>
    <base id="2">
        <name>producao</name>
        <host>192.168.0.2</host>
        <type>pgsql</type>
        <user>admin</user>
    </base>
</bases>
```

```
$base->appendChild($dom->createElement('name', 'teste'));
$base->appendChild($dom->createElement('host', '192.168.0.1'));
$base->appendChild($dom->createElement('type', 'mysql'));
$base->appendChild($dom->createElement('user', 'mary'));
echo $dom->saveXML($bases);
```

Resultado:

```
<bases>
  <base id="1">
    <name>teste</name>
    <host>192.168.0.1</host>
    <type>mysql</type>
    <user>mary</user>
  </base>
</bases>
```

Observação: caso esteja rodando o PHP no browser, veja o XML pelo código-fonte da página.

3.5 SPL

SPL é um conjunto de classes e interfaces que fornece ao desenvolvedor uma API padronizada para resolver problemas comuns e também criar classes com maior potencial de interoperabilidade.

O PHP sempre ofereceu muitas funções para manipulação de arquivos (`file`, `basename`, `fopen`, `file_get_contents`, `file_put_contents`, `copy`, `unlink`, `rename`, `filesize`, `fwrite`, `pathinfo`), manipulação de vetores (`array_diff`, `array_merge`, `array_shift`, `array_pop`, `array_unshift`, `array_push`, `array_keys`. `array_values`), entre outros. Antes do PHP 5, quando a SPL ainda não existia, mesmo tendo esse arsenal de funções à disposição, sempre existiram programadores que preferiam fazer tudo de maneira orientada a objetos. Assim muitos acabaram criando classes que "agrupavam" funcionalidades em comum para fornecer um meio orientado a objetos de resolver problemas. O que a SPL oferece é um conjunto de classes com funcionalidades comuns, como manipulação de arquivos, vetores, pilhas e filas de maneira padronizada.

```
Item: linguado
Item: salmão
Item: tilápia
x:i:0;a:10:{i:10;s:4:"atum";i:9;s:8:"bacalhau";i:3;s:6:"badejo";i:8;s:5:"bagre";
i:7;s:6:"cavala";i:6;s:7:"corvina";i:5;s:7:"dourado";i:2;s:8:"linguado";
i:0;s:7:"salmão";i:1;s:8:"tilápia";};m:a:0:{}
```

3.6 Reflection

Reflection (Reflexão) é uma API formada por um conjunto de classes que permite que um programa possa obter informações sobre sua própria estrutura, suas classes, interfaces, seus métodos, suas funções, entre outros. Entre essas informações podemos descobrir as constantes, os atributos e métodos de uma classe, descobrir se um método é abstrato, privado ou final, descobrir informações sobre os parâmetros de um método, entre outros.

3.6.1 ReflectionClass

A API Reflection pode ser aplicada sobre código-fonte construído pelo desenvolvedor, mas também sobre classes nativas do PHP, como as classes da SPL. Para demonstrar algumas funcionalidades, vamos declarar uma pequena classe somente para propósito didático. No código a seguir teremos as classes `Veiculo` e `Automovel` com alguns atributos e métodos definidos. O próximo passo será investigar essa classe.

veiculo.php

```
<?php
class Veiculo {
    protected $ano;
    protected $cor;
    protected $marca;
    protected $parts;

    public function getParts() {}
    public function setMarca($marca) {}
}

class Automovel extends Veiculo {
    private $placa;
    const RODAS=4;
```

- Uma parte inicial do Namespace corresponderá ao diretório-base de onde as classes serão carregadas. Subníveis do Namespace corresponderão a subdiretórios.
- Para que a classe seja carregada apropriadamente, o arquivo em que ela é salva deve ter exatamente o mesmo nome que a classe, acrescido do sufixo ".php". Ex.: classe `\Livro\Widgets\Form\Field` => *Lib/Livro/Widgets/Form/Field.php*.
- Os arquivos devem ser armazenados no formato UTF-8.
- As classes devem ser representadas no formato StudlyCaps (ex.: `FormField`).
- Os métodos devem ser representados no formato camelCase (ex.: `getData()`).
- Deve existir pelo menos uma linha em branco antes da declaração do Namespace e também antes da declaração de uso (use).
- Namespaces e classes devem seguir um autoloader-padrão.

Ainda existem diversas diretrizes que definem padrão de abertura de chaves, de declaração de visibilidade de métodos, tamanho de linhas, indentação, entre outros. Mas o objetivo aqui não é transcrever toda a PSR.

As classes do livro seguem os padrões da PSR em sua grande maioria. Da mesma maneira, o carregamento das classes é realizado por um mecanismo de carga (Autoloader) que segue as diretrizes da PSR-4 que, no momento da escrita desta edição, era a PSR mais atual sobre esse assunto.

3.10 Namespaces

Quando o PHP surgiu não existiam classes. Depois de algumas versões foram criadas as classes e os programadores começaram a criar códigos orientados a objetos em PHP. No PHP3 já era possível declarar classes e criar objetos a partir delas. Mas naquela época o suporte à orientação a objetos no PHP não era robusto e os objetos eram tratados internamente como vetores associativos. Apesar de ter suporte à orientação a objetos, a maioria ainda programava de maneira estruturada, declarando funções e usando os comandos `include`/`require` para incluir arquivos e reaproveitar código.

O PHP4 melhorou algumas coisas, mas foi o PHP5 que causou uma revolução. No PHP5 foram introduzidas grandes funcionalidades como passagem de objetos por referência, métodos mágicos, interfaces, classes e métodos abstratos, visibilidade, `__construct()` e `__destruct()`, autoload, static, exceções, SimpleXML, SPL, entre outras.

Por fim temos uma variação do exemplo anterior, na qual solicitaremos que determinada classe registre ela mesma o seu método de carregamento de classes. Neste caso instanciaremos a classe de carga `ApplicationLoader` e solicitaremos o método `register()`. O método `register()` por sua vez irá executar a função `spl_autoload_register()` para registrar o método `$this->loadClass()` como autoloader. Essa abordagem é interessante, pois permite que a classe de carregamento altere não somente o algoritmo de carga, mas também o próprio nome do método de carga (`loadClass`) sem interferir na chamada externa (`register`).

spl_autoload3.php

```
<?php
$al = new ApplicationLoader;
$al->register();

class ApplicationLoader {
    public function register() {
        spl_autoload_register(array($this, 'loadClass'));
    }
    public function loadClass($class) {
        if (file_exists("App/{$class}.php")) {
            require_once "App/{$class}.php";
            return TRUE;
        }
    }
}
```

Observação: por motivos de simplificação colocamos a declaração da classe (`ApplicationLoader`), bem como sua chamada, no mesmo arquivo, porém essa não é uma boa prática. Na realidade a classe deverá estar localizada em um arquivo próprio, que contenha somente a sua definição, enquanto sua instanciação e consequente chamada (`register`) normalmente estará no principal arquivo da aplicação (*index.php*), já que as requisições para as demais classes do projeto partem dele.

3.12 PDO – PHP Data Objects

O PHP é, em sua maioria, um projeto voluntário cujos colaboradores estão distribuídos geograficamente ao redor de todo o planeta. Como resultado, o PHP evoluiu baseado em necessidades individuais para resolver problemas pontuais, movidos por razões diversas. Por um lado essas colaborações fizeram o PHP crescer rapidamente, por outro geraram uma fragmentação das extensões de acesso

CAPÍTULO 4

Persistência

O homem está sempre disposto a negar tudo aquilo que não compreende.

Blaise Pascal

Ao programarmos de maneira orientada a objetos, nossa aplicação deve ser formada exclusivamente por um conjunto de objetos relacionados. Quando trabalhamos com um conjunto de objetos em memória, em algum momento precisamos persistir esses objetos na base de dados, ou seja, armazená-los e permitir que sejam posteriormente carregados a partir do banco de dados para a memória. Pensando nisso, estudaremos as técnicas mais utilizadas para persistência de objetos em bases de dados relacionais, assim como criaremos uma API orientada a objetos que irá permitir que façamos tudo isso de forma transparente, sem nos preocuparmos com os detalhes internos de implementação.

4.1 Introdução

De modo geral, persistência significa continuar a existir, perseverar, durar longo tempo ou permanecer. No contexto de uma aplicação de negócios, na qual temos objetos representando as mais diversas entidades a serem manipuladas (pessoas, mercadorias, livros, clientes, arquivos etc.), persistência significa a possibilidade de esses objetos existirem em um meio externo à aplicação que os criou, de modo que esse meio deve permitir que o objeto perdure, ou seja, não deve ser um meio volátil. Os bancos de dados relacionais são o meio mais utilizado para isso (embora não seja o único). Com o auxílio de mecanismos sofisticados específicos de cada fornecedor, esses bancos de dados oferecem vários recursos que permitem armazenar e manipular, por meio da linguagem SQL, os dados neles contidos.

conhecer essa interface para manipular as informações. O acesso aos dados via linguagem SQL, por exemplo, fica contido nessa camada. Dessa forma evitamos ter a manipulação de dados espalhada pelo código-fonte da aplicação, podendo inclusive estruturar a equipe de desenvolvimento para que os mais talentosos em SQL sejam responsáveis pelas classes de acesso aos dados.

Existem alguns Gateways que implementam o acesso às estruturas de dados. A seguir estudaremos alguns dos mais utilizados, que são Table Data Gateway, Row Data Gateway, Active Record e Data Mapper.

4.2.1 Table Data Gateway

O objetivo do Design Pattern Table Data Gateway é oferecer uma interface de comunicação com o banco de dados que permita operações de inserção, alteração, exclusão e busca de registros. Essa interface pode ser implementada por uma classe responsável por persistir e retornar dados do banco de dados. Para isso existem métodos específicos que traduzem sua função em instruções SQL.

No Design Pattern Table Data Gateway, existe uma classe para manipulação de cada tabela do banco de dados, e apenas uma instância dessa classe irá manipular todos os registros da tabela, por isso é necessário sempre identificar o registro sobre o qual o método estará operando. Uma classe Table Data Gateway é por natureza *stateless*, ou seja, não mantém o estado de suas propriedades; atua simplesmente como ponte entre o objeto de negócios e o banco de dados. Na figura 4.1 vemos a representação de uma classe Table Data Gateway.

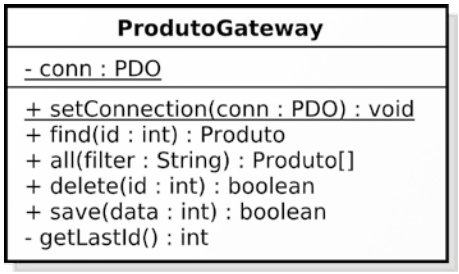

Figura 4.1 – Table Data Gateway.

No programa a seguir estamos criando a classe `ProdutoGateway`. Essa classe, que implementa o Design Pattern Table Data Gateway, contém métodos para gravação (`save`), exclusão (`delete`) e busca (`find`, `all`) de registros em base de dados. Antes de mais nada, é preciso criar um método para receber a conexão ativa (`setConnection`). Esse método implementa uma injeção de dependência, pois os demais métodos

os valores novamente via parâmetros em métodos como `delete()` ou `save()`, como era necessário para o Table Data Gateway em virtude da sua natureza *stateless*. Uma desvantagem é que em alguns cenários o consumo de memória poderá ser superior, pois instanciamos naturalmente mais objetos no nosso sistema, tendo em vista que cada objeto agora representará um único registro da tabela. Apesar dessa desvantagem, esse Design Pattern retrata com mais fidelidade o modelo de orientação a objetos, sendo aceitável essa pequena perda de desempenho para atingirmos maior clareza e entendimento do código-fonte.

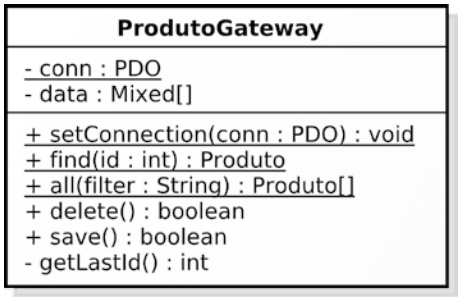

Figura 4.3 – Row Data Gateway.

Um Row Data Gateway deve prover métodos que permitam construir um objeto e posteriormente armazená-lo no banco de dados, além de métodos estáticos que permitam carregar um objeto ou um conjunto de objetos do banco de dados.

A classe `ProdutoGateway` a seguir demonstra a implementação de um Row Data Gateway. A classe utiliza os métodos mágicos `__set()` e `__get()` para armazenar atributos no vetor `$this->data`. O método estático `setConnection()` é utilizado para receber a conexão `PDO` com a base de dados. Os métodos estáticos `find()` e `all()` são utilizados para buscar respectivamente um registro e um conjunto de registros. Os métodos `find()` e `all()` sempre retornarão objetos da classe `ProdutoGateway`, o que é alcançado pelo uso da constante `__CLASS__` nos métodos `fetch()` e `fetchAll()`. A constante `__CLASS__` representa a própria classe em que é utilizada. Como o Row Data Gateway armazena seus atributos internamente no vetor `$this->data`, esses atributos estarão disponíveis no momento de realizar operações como `delete()` e `save()`. Assim, essas operações não precisarão receber parâmetros para indicar o registro sobre o qual as operações serão realizadas, bastando coletar esses dados do vetor `$this->data`.

Observação: neste exemplo as propriedades serão armazenadas em um array chamado `$data`. Isso porque estamos utilizando os métodos `__get()` e `__set()` para interceptar os acessos aos atributos do objeto.

classes/rdg/ProdutoGateway.php

```
<?php
class ProdutoGateway {
    private static $conn;
    private $data;

    function __get($prop) {
        return $this->data[$prop];
    }

    function __set($prop, $value) {
        $this->data[$prop] = $value;
    }

    public static function setConnection( PDO $conn ) {
        self::$conn = $conn;
    }

    public static function find($id) {
        $sql = "SELECT * FROM produto where id = '$id' ";
        print "$sql <br>\n";
        $result = self::$conn->query($sql);
        return $result->fetchObject(__CLASS__);
    }

    public static function all($filter = '') {
        $sql = "SELECT * FROM produto  ";
        if ($filter) {
            $sql .= "where $filter";
        }
        print "$sql <br>\n";
        $result = self::$conn->query($sql);
        return $result->fetchAll(PDO::FETCH_CLASS, __CLASS__);
    }

    public function delete() {
        $sql = "DELETE FROM produto where id = '{$this->id}' ";
        print "$sql <br>\n";
        return self::$conn->query($sql);
    }

    public function save() {
        if (empty($this->data['id'])) {
            $id = $this->getLastId() +1;
            $sql = "INSERT INTO produto (id, descricao, estoque, preco_custo, ".
```

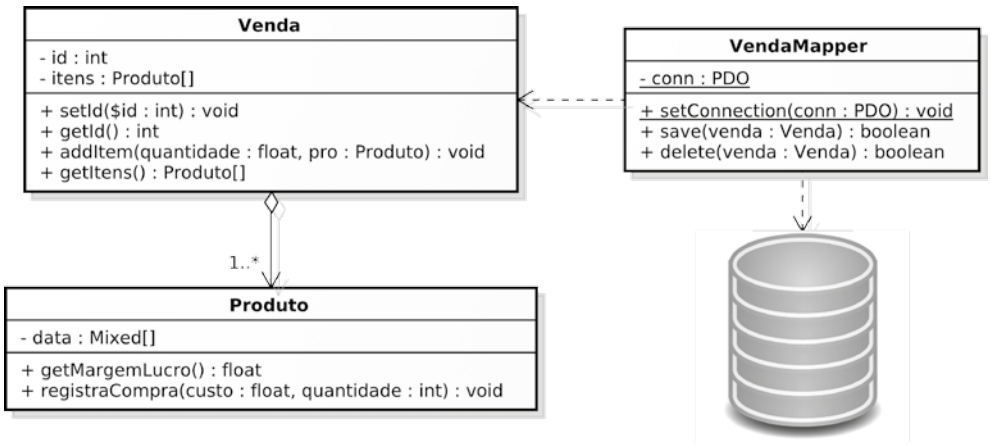

Figura 4.5 – Data Mapper.

Para demonstrar esse Design Pattern, vamos criar algumas classes bastante simples. Inicialmente precisamos de uma classe para representar um produto. Fica claro aqui que a classe proposta é uma classe incompleta, pois carece de métodos de negócio.

classes/dm/Produto.php

```
<?php
class Produto {
    private $data;

    function __get($prop) {
        return $this->data[$prop];
    }

    function __set($prop, $value) {
        $this->data[$prop] = $value;
    }
}
```

Agora vamos criar uma classe para representar uma venda. Essa classe terá métodos para atribur e retornar um `ID`, bem como adicionar (`addItem`) e retornar (`getItens`) itens (produtos).

classes/dm/Venda.php

```
<?php
class Venda {
    private $id;
```

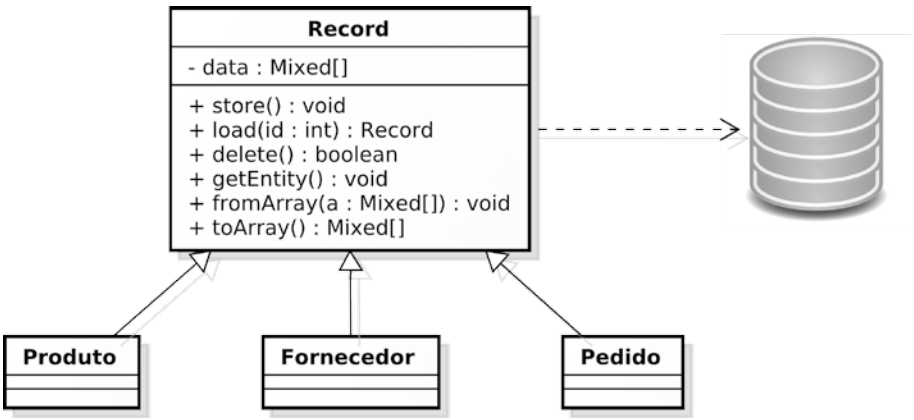

Figura 4.9 – Classe Record e a camada de domínio.

classes/api/Record.php

```
<?php
abstract class Record {
    protected $data; // array contendo os dados do objeto

    public function __construct($id = NULL) {
        if ($id) { // se o ID for informado
            // carrega o objeto correspondente
            $object = $this->load($id);
            if ($object)
            {
                $this->fromArray($object->toArray());
            }
        }
    }
```

O método `__clone()` será executado sempre que um objeto for clonado. Nesses casos em que um Active Record é clonado, ele deve manter todas as suas propriedades, com exceção de seu `ID`, por isso estamos limpando o `ID` do clone. Caso mantivéssemos o mesmo `ID`, teríamos dois objetos Active Record com o mesmo `ID` no sistema, o que causaria erros na persistência do objeto. Limpar o `ID` fará com que um novo `ID` seja gerado para o clone no momento em que ele for persistido na base de dados.

```
    public function __clone() {
        unset($this->data['id']);
    }
```

CAPÍTULO 5

Apresentação e controle

Aquele que conhece os outros é sábio; mas quem conhece a si mesmo é iluminado! Aquele que vence os outros é forte; mas aquele que vence a si mesmo é poderoso! Seja humilde e permanecerás íntegro.

Lao-Tsé

Nos capítulos anteriores vimos fundamentos de orientação a objetos, acesso à base de dados e persistência de objetos. Agora que já concluímos essa camada da aplicação, precisamos nos preocupar com outros aspectos, como a sua interface com o usuário, a interpretação e a execução de ações (fluxo de controle). Neste capítulo, desenvolveremos uma série de classes com o objetivo de construir o visual da aplicação e também de interpretar as ações solicitadas pelo usuário, coordenando o fluxo de execução da aplicação. Como vamos criar uma grande quantidade de classes, precisaremos antes organizá-las sob uma estrutura de diretórios e Namespaces. Para começar, vamos abordar o padrão MVC.

5.1 Padrão MVC

Model View Controller (MVC) é um Design Pattern que está entre os mais conhecidos. Seus conceitos remontam à plataforma Smaltalk na década de 1970. Basicamente uma aplicação que segue o Design Pattern Model View Controller tem as suas classes separadas em três grandes grupos de responsabilidades. A intenção principal ao utilizarmos o Design Pattern MVC é não misturarmos em uma mesma classe responsabilidades diferentes. A seguir veremos quais são essas responsabilidades.

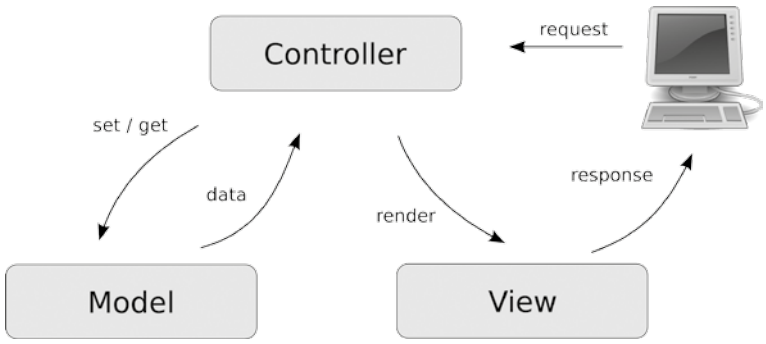

Figura 5.1 – Modelo MVC.

Ao utilizarmos o padrão MVC, alguns cuidados devem ser tomados, por exemplo:

- Uma classe de modelo não deve emitir mensagens ao usuário por meio de comandos como `print`, muito menos gerar mensagens contendo marcações como HTML. Exibir informações ao usuário por meio de uma linguagem de marcação é uma tarefa de uma classe de visualização.
- Uma classe de controle não deve executar diretamente comandos de acesso a dados como SQL. Buscar e atualizar dados relativos ao modelo de domínio são tarefas de uma classe de modelo.
- Uma classe de visualização não deve conter regras de negócio, nem decidir o que deve ser executado em determinado momento. Regras de negócio são de responsabilidade de uma classe de modelo, e decidir o que deve ser executado é de responsabilidade de uma classe de controle.

Observação: um sistema MVC clássico terá uma classe Controller para cada View existente, mas essa abordagem não é a única. Alguns frameworks frequentemente mesclam View e Controller na mesma camada, deixando-as diretamente vinculadas.

5.2 Organização de Namespaces e diretórios

Estamos criando uma grande quantidade de classes. No capítulo anterior, classes relativas à persistência e log foram criadas. Neste capítulo criaremos classes para apresentação de informações e para controle de execução de ações. Assim, é fundamental organizarmos as classes em uma estrutura de diretórios que reflita a responsabilidade de cada grupo de classes. Então, vamos separar para melhor organizar.

Os arquivos de configuração, que compreendem, entre outras coisas, as definições de acesso às bases de dados, estarão localizados na pasta *App/Config*. No arquivo a seguir temos como exemplo a definição de acesso a uma base de dados chamada `livro`, que por sua vez representa um banco de dados SQLite.

App/Config/livro.ini

```
host = localhost
name = App/Database/livro.db
user =
pass =
type = sqlite
```

5.3 SPL Autoloaders

Não basta definirmos um conjunto de classes, é preciso carregá-las apropriadamente no momento de sua utilização. Antigamente, quando o recurso de Namespaces não existia no PHP, um simples `require` era suficiente. Agora, com uma estrutura organizada em torno de Namespaces, é preciso um algoritmo mais robusto para carregamento de classes. Assim, seguindo as recomendações da PSR, vamos utilizar um carregador de classes utilizando a SPL, como demonstrado no capítulo 3.

Para realizar o carregamento das classes, criaremos dois algoritmos distintos. O primeiro fará o carregamento das classes do framework (a partir da pasta */Lib*) e o outro fará o carregamento das classes da aplicação (a partir da pasta */App*). Esses dois carregadores (*loaders*) serão muito úteis e serão posteriormente executados a partir do `index` da aplicação, pois todas as requisições passarão pelo `index`, o que será demonstrado na próxima seção.

5.3.1 Library Loader

Sempre que precisarmos utilizar uma classe de nosso pequeno framework (a partir da pasta */Lib*), utilizaremos a classe `ClassLoader`. Essa classe executará um algoritmo de carregamento de classes. No capítulo 3 abordamos a SPL e, dentro de suas características, vimos a função `spl_autoload_register()`, que registra um método de carregamento de classes em uma pilha de execução. Podemos executar essa função diversas vezes, e a cada vez "registraremos" uma função para carregamento.

A classe `ClassLoader` terá o método `register()` que irá registrar como método de carregamento de classes o método `loadClass()`, que por sua vez receberá via parâmetro

exemplo_loader.php

```
<?php
require_once 'Lib/Livro/Core/ClassLoader.php';
$al= new Livro\Core\ClassLoader;
$al->addNamespace('Livro', 'Lib/Livro');
$al->register();

use Livro\Database\Connection;
$obj1 = Connection::open('livro');
var_dump($obj1);
```

Resultado:

```
object(PDO)#2 (0) { }
```

5.4 Padrões de controle

Na seção anterior, vimos como realizar o carregamento das classes de nossa aplicação. Agora que organizamos as classes em estruturas de diretórios e Namespaces, o próximo passo será implementarmos a interação do usuário com a aplicação, o que é feito pela interpretação de ações que normalmente são requisitadas a partir de uma URL no protocolo HTTP. Inicialmente vamos estudar os principais padrões utilizados para organizar o fluxo de controle de uma página: Page Controller e Front Controller.

5.4.1 Page Controller

Para Martin Fowler, um Page Controller é "um objeto que controla uma requisição para uma página ou ação específica". Esse objeto pode ser representado por uma classe que representa uma página e decide qual ação (método) executar para cada requisição HTTP realizada.

Quando começamos a estudar o mundo da web, suas páginas HTML e posteriormente os programas em PHP, temos tendência a resumir tudo em termos de scripts, de modo que um script representa um programa ou mesmo uma pequena funcionalidade de um programa. Para executar uma página, você passa para o servidor a localização do script, que é processado, e então obtém o retorno desse processamento. Se exagerarmos, podemos ter vários scripts para o mesmo programa, como na listagem a seguir, na qual cada script pode representar uma funcionalidade de um cadastro, por exemplo.

que solicita algo ao serviço. Um "Client" pode ser desenvolvido em qualquer linguagem de programação que tenha suporte a Web Services e pode rodar em um ambiente desktop, servidor, dispositivos móveis, e outros. Além disso, temos o servidor de Web Service que fornece o serviço e se comunica com o "Client" por meio de um pacote descrito no protocolo SOAP sobre HTTP. A figura 5.6 procura ilustrar essa comunicação.

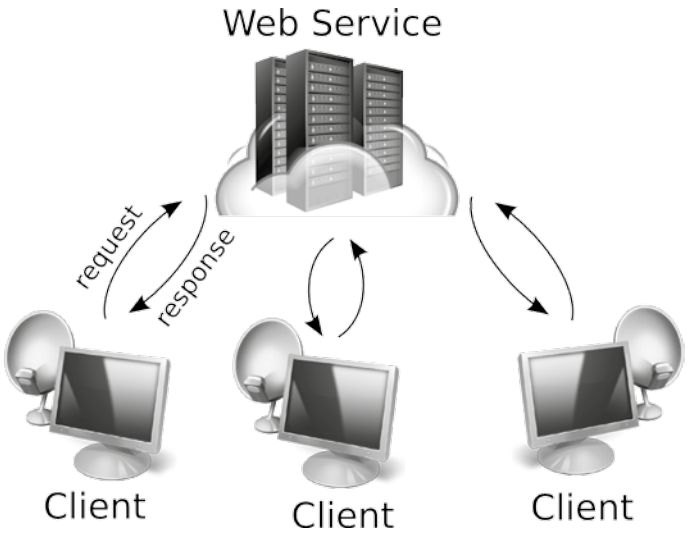

Figura 5.6 – Visão geral de Web Services.

5.4.3.2 Remote Facade

Como exposto no início desta seção, precisamos disponibilizar algumas funcionalidades de nossa aplicação para aplicações externas. Para tal, precisamos construir uma interface que concentre a responsabilidade no lado da aplicação servidora e seja responsável internamente por diversas chamadas a métodos internos da camada de modelo. Essa alternativa permite que a aplicação cliente faça reduzidas chamadas à aplicação servidora. Em vez de invocarmos vários métodos para atingir determinado objetivo, concentramos a lógica na aplicação servidora em uma camada a qual damos o nome de Fachada (Facade). Como estamos em um cenário com execuções remotas, esse Design Pattern é chamado de Remote Facade.

Remote Facades são ideais para utilizar em um ambiente distribuído, em que temos uma aplicação cliente e uma aplicação servidora. Em aplicações de negócio, as aplicações cliente precisam abrir muitas transações com a base de dados para inserir, alterar, excluir e listar registros. Colocar o código transacional no lado da aplicação cliente diminui a eficiência da aplicação, além de aumentar o tráfego

- **Not found** – A URL do serviço não foi localizada. Verifique a localização (parâmetro `location`).
- **Permission denied** – Exceção gerada pelo servidor ao indicar que a classe solicitada pela URL (`?class`) não está na lista de classes permitidas.
- **Método "xyz" não encontrado** – O método que foi executado sobre o objeto `SoapClient` não foi localizado no lado só servidor dentro da classe de serviço solicitada (`?class`).

Observação: ao trabalhar com Web Services, mais do que nunca é muito importante observar os logs de erros gerados pelo PHP. Portanto, verifique se no *php.ini* a cláusula de log está ligada (`log_errors = On`) e se o arquivo de log está indicado (*error_log = /tmp/php_errors.log*). Se o programa não funcionar, ligue e monitore os logs, pois eles indicarão onde está o problema.

5.5 Padrões de apresentação

Antes de ler este livro, provavelmente você já escreveu em algum momento um código-fonte que misturava diferentes tecnologias, como PHP, HTML, SQL, JavaScript, em um mesmo arquivo. Com o tempo, você vai percebendo que não é muito produtivo trabalhar dessa maneira e que, apesar de inicialmente funcionar, ao longo do tempo acabam criando-se códigos de difícil manutenção e interpretação, justamente por misturar diferentes aspectos de desenvolvimento (apresentação, lógica, regras de negócio).

Ao criarmos códigos que misturam diferentes aspectos, ficamos de certa maneira "presos" a determinadas escolhas. O código a seguir mostra justamente como não devemos implementar. O código tem a configuração de acesso ao banco de dados explícita, e ela deveria estar isolada da implementação. Temos também a consulta SQL, que deveria estar concentrada em uma classe, espalhada no código. Temos também o uso de funções específicas (`pg_*`) do banco PostgreSQL, o que nos deixa presos a essa tecnologia. Por fim, usamos diretamente HTML para exibir os resultados em uma tabela.

O cenário que hoje funciona bem pode se tornar desastroso em questão de alguns meses. Ao decidirmos realocar o banco de dados, precisaremos editar muitos arquivos para realizar essa substituição. Se precisarmos trocar a tecnologia de banco de dados, teremos um trabalho enorme ao editar muitos arquivos. Por fim, se resolvermos trocar as tabelas (`<table>`) por outras *tags* utilizando alguma biblioteca visual como a Bootstrap, por exemplo, teremos novamente muitos

CAPÍTULO 6

Formulários e listagens

Uma sociedade só será democrática quando ninguém for tão rico que possa comprar alguém e ninguém for tão pobre que tenha de se vender a alguém.

Jacques Rousseau

Neste capítulo iremos nos concentrar em alguns dos componentes que estão entre os mais utilizados na maioria das aplicações: formulários e listagens. Utilizamos formulários para as mais diversas formas de entrada de dados na aplicação, como para a inserção de novos registros, definição de preferências do sistema ou de parâmetros para filtragem de um relatório, entre outras. Utilizamos listagens para exibir os dados da aplicação para simples conferência, em relatórios, ou ainda para editar e excluir registros. Neste capítulo criaremos componentes que visam facilitar a implementação de formulários e listagens de forma orientada a objetos.

6.1 Formulários

No final do capítulo 1 criamos alguns exemplos simples de utilização de formulários em PHP. Naqueles exemplos, construímos os formulários utilizando simplesmente HTML. No capítulo anterior abordamos os benefícios que a utilização de componentes e templates nos proporciona para a criação de interfaces, sendo o principal deles o maior isolamento entre a apresentação e a lógica da aplicação. Nesta seção utilizaremos os conhecimentos adquiridos no capítulo anterior para desenvolver um conjunto de classes que permitirão a criação e manipulação de formulários de maneira totalmente orientada a objetos.

Aplicações de negócio frequentemente utilizam diversas telas para entrada de dados por meio de formulários. Ao projetar um sistema, temos de pensar no reaproveitamento de código. A abordagem tradicional, vista no capítulo 1, minimiza

oferecer. Entre essas operações, podemos citar: definir o nome do campo, definir um rótulo (label) para o campo, definir um valor para o campo, definir se o campo será editável, entre outros.

Para montar um formulário, utilizaremos um relacionamento de agregação entre a classe `Form` e a classe `Field`. Dessa forma, será possível adicionar ao formulário quaisquer objetos filhos de `Field`. Para tal, será criado na classe `Form`, um método que permitirá adicionar campos (objetos `Field`) em sua estrutura. Veja na figura 6.1 o diagrama de classes resumido.

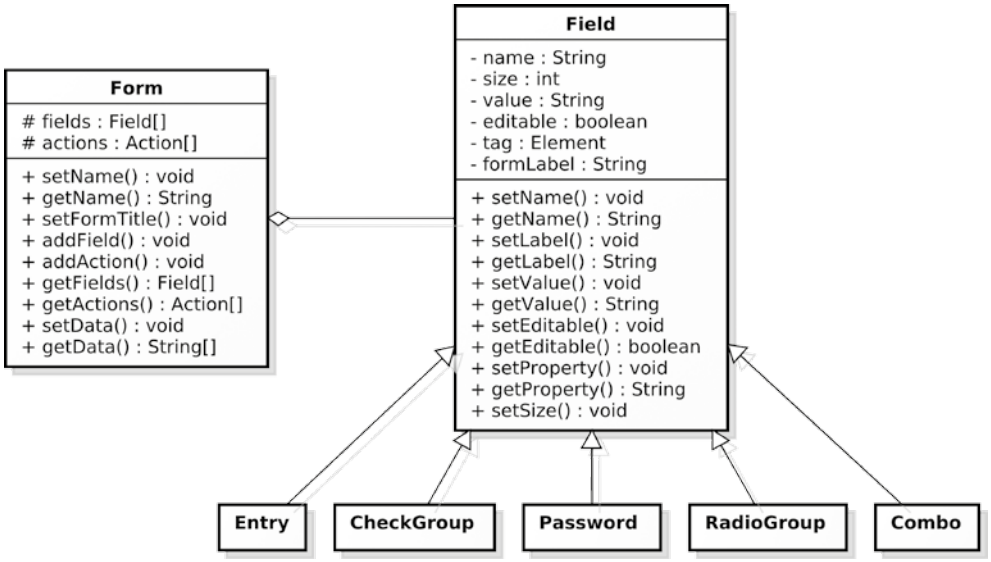

Figura 6.1 – Estrutura de classes para formulários.

A partir de então, podemos começar a construir as classes que farão parte desse ecossistema de objetos inter-relacionados que é o formulário.

6.1.1 Classe para formulários

A primeira classe que criaremos representará um formulário e será também a mais importante, tendo em vista que ela centralizará as chamadas para as demais. Assim, para agruparmos todos os tipos de campo que vimos anteriormente (input texto, combo, radio, check etc.), será necessário criar uma estrutura que represente um formulário. Essa estrutura deverá criar a tag `<form>` do HTML e agrupar vários elementos (campos). Para isso, criaremos a classe `Form`. Essa classe será filha de `Element`, classe que criamos para elaborar a estrutura de um elemento HTML.

ao comportamento, como se fosse uma nova camada.

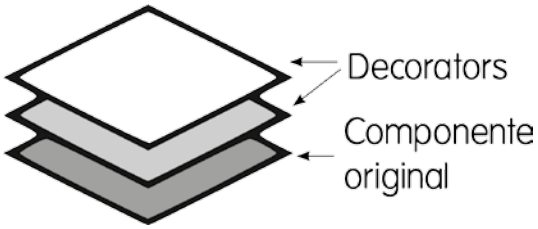

Figura 6.20 – Camadas de decoração.

6.1.4.1 Form Wrappers com Decorator Pattern

Vamos utilizar a ideia do Design Pattern Decorator e escrever uma nova classe para montagem de formulários chamada `FormWrapper`, que irá "transformar" a classe já existente. A nova classe irá atuar sobre um objeto da classe `Form` já instanciado e irá percorrer os objetos que fazem parte do formulário, gerando uma nova saída de HTML, dessa vez no formato esperado pela biblioteca Bootstrap.

> **Observação:** para a demonstração do Design Pattern Decorator, foi escolhida a biblioteca Bootstrap por ser muito difundida no momento da escrita dessa edição. Mas a partir do momento que você entender o conceito poderá aplicar o mesmo Design Pattern para realizar outros tipos de transformação.

Para iniciar, a classe `FormWrapper` irá receber em seu método construtor uma instância já existente da classe `Form`, sobre a qual irá realizar as transformações necessárias para adequar a estrutura do formulário existente e convertê-la na estrutura que a biblioteca Bootstrap espera. A instância já existente será armazenada no atributo `$decorated`.

Como a ideia do Design Pattern é adicionar comportamento a um objeto já existente, não podemos "anular" os comportamentos (métodos) já existentes. Também não vamos reescrevê-los aqui. Métodos da classe `Form`, como `setName()`, `addField()`, `addAction()`, entre outros, devem continuar funcionando a partir da classe nova. Para que eles continuem a funcionar, sempre que o desenvolvedor executar um método não encontrado na classe `FormWrapper`, automaticamente a execução será redirecionada para o objeto "decorado", representado pelo atributo `$decorated`. Esse redirecionamento de chamadas é obtido pelo método `__call()`, automaticamente executado sempre que um método não encontrado na classe atual for executado. Nesses casos a execução é redirecionada por meio da função `call_user_func_array()` para o objeto decorado.

seguida, devemos alterar a declaração do formulário (`Form`), declarando um objeto da nova classe (`FormWrapper`) no lugar deste, mas passando o formulário existente como parâmetro do construtor da nova classe conforme a sintaxe: `new FormWrapper(new Form('form_contato'))`. Dessa forma, passaremos a utilizar a nova classe (`FormWrapper`), mas as chamadas aos métodos não encontrados serão automaticamente direcionadas ao objeto "decorado" que foi recebido como parâmetro. É importante lembrar que esse redirecionamento é feito pelo método `__call()`.

App/Control/ContatoFormWrapper.php

```
<?php
use Livro\Control\Page;
use Livro\Control\Action;
use Livro\Widgets\Form\Form;
use Livro\Widgets\Form\Entry;
use Livro\Widgets\Form\Combo;
use Livro\Widgets\Form\Text;
use Livro\Widgets\Container\Panel;
use Livro\Widgets\Wrapper\FormWrapper;

class ContatoFormWrapper extends Page {
    private $form;

    function __construct() {
        parent::__construct();

        // instancia um formulário
        $this->form = new FormWrapper(new Form('form_contato'));
        $this->form->style = 'display:block; margin:20px;';

        // cria os campos do formulário
        $nome     = new Entry('nome');
        $email    = new Entry('email');
        $assunto  = new Combo('assunto');
        $mensagem = new Text('mensagem');

        $this->form->addField('Nome', $nome, 300);
        $this->form->addField('E-mail', $email, 300);
        $this->form->addField('Assunto', $assunto, 300);
        $this->form->addField('Mensagem', $mensagem, 300);

        // define alguns atributos
        $assunto->addItems( array('1' => 'Sugestão',
                                  '2' => 'Reclamação',
                                  '3' => 'Suporte técnico',
```

```
                                '4' => 'Cobrança',
                                '5' => 'Outro') );
        $mensagem->setSize(300, 80);

        $this->form->addAction('Enviar', new Action(array($this, 'onSend')));

        $panel = new Panel('Formulário de contato');
        $panel->add($this->form);

        // adiciona o painel à página
        parent::add($panel);
    }

    public function onSend() {
        // ...
    }
}
```

A figura 6.21 demonstra a utilização do programa recém-criado. Para acioná-lo, basta acessarmos pela URL *index.php?class=ContatoFormWrapper*, onde quer que esteja rodando nosso servidor de páginas. É possível perceber imediatamente a transformação realizada, desde que a biblioteca Bootstrap esteja carregada, o que é garantido no arquivo *index.php*.

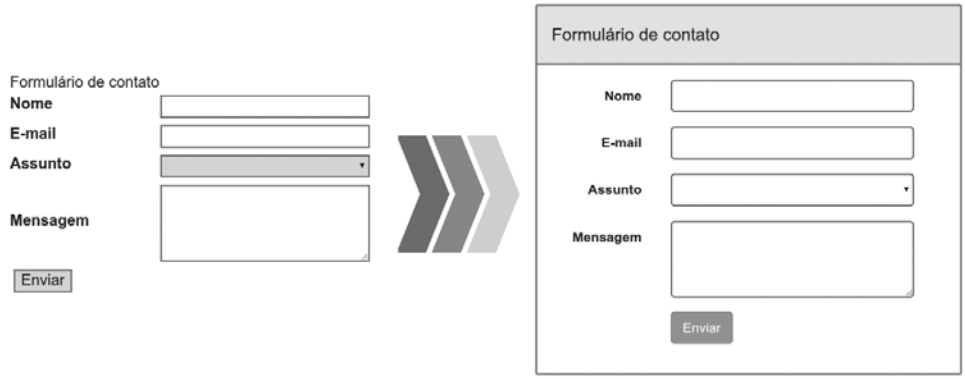

Figura 6.21 – Transformação de formulário.

6.2 Listagens

Até o momento, vimos como implementar formulários de uma maneira orientada a objetos. Usamos formulários para cadastrar novos registros, bem como editar registros existentes. Porém, antes de chegarmos a uma tela contendo um formulário,

```
        // incrementa o contador de linhas
        $this->rowcount ++;
    }
}
```

6.2.2 Colunas da Datagrid

A classe `DatagridColumn` será utilizada para representar as características que fazem parte de uma coluna em uma listagem. Para tal, essa classe receberá em seu método construtor o nome do campo do banco de dados que a coluna exibirá (`$name`), o rótulo de texto que será exibido no título da coluna (`$label`), o alinhamento da coluna (`$align`) e a largura da coluna (`$width`). Os métodos `getName()`, `getLabel()`, `getAlign()` e `getWidth()` serão utilizados para retornar essas propriedades definidas pelo método construtor. Veja na figura 6.24 a classe `DatagridColumn`.

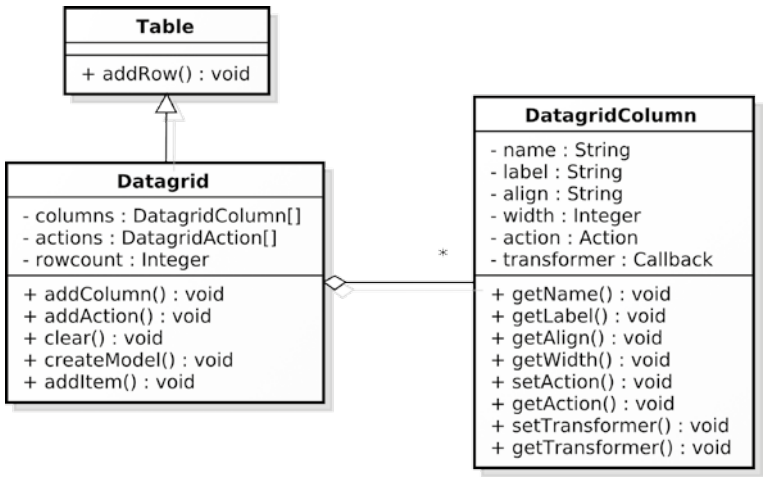

Figura 6.24 – Classe `DatagridColumn`.

Lib/Livro/Widgets/Datagrid/DatagridColumn.php

```
<?php
namespace Livro\Widgets\Datagrid;

use Livro\Control\Action;

class DatagridColumn {
    private $name;
    private $label;
    private $align;
```

CAPÍTULO 7

Criando uma aplicação

Quando morremos, nada pode ser levado conosco, com a exceção das sementes lançadas por nosso trabalho e do nosso conhecimento.

Dalai Lama

Ao longo desta obra, criamos uma série de classes para automatizar desde a conexão com banco de dados, transações, persistência de objetos e manipulação de coleções de objetos, até a criação de componentes para interface como diálogos, formulários, listagens, e outros. Ao longo de cada capítulo procuramos dar exemplos da utilização de cada classe criada e agora chegou o momento de utilizar esse conhecimento para construir algo maior, uma aplicação completa.

7.1 Aplicação

O objetivo deste capítulo é a construção de uma aplicação para controle de vendas. Essa aplicação contará com cadastros básicos como clientes, cidades, produtos, fabricantes, processo de registro de vendas, e também relatórios de vendas e de contas geradas. Com isso, pretendemos mostrar como construir diferentes tipos de interface e interação entre o usuário e a aplicação. A aplicação criada será de pequeno porte, e não terá como objetivo o uso em um ambiente real, afinal é somente um protótipo voltado para o aprendizado. Ao compreender como o protótipo foi construído, você poderá construir aplicações maiores.

A aplicação proposta contará com as seguintes funcionalidades:

- **Cadastro de cidades** – Oferecer um cadastro de cidades, com informações como: nome da cidade e estado.

- **Cadastro de fabricantes** – Oferecer um cadastro de fabricantes, com informações como: nome e site.
- **Cadastro de produtos** – Oferecer um cadastro de produtos, com informações como: descrição, estoque, preço de custo, preço de venda, fabricante, tipo (máquina, acessório) e unidade de medida.
- **Cadastro de pessoas** – Oferecer um cadastro de pessoas, com informações como: nome, endereço, bairro, telefone, email, cidade e grupo (cliente, fornecedor, revendedor, colaborador).
- **Registro de vendas** – Oferecer uma tela para registro das vendas ocorridas, podendo informar uma série de itens (produtos) vendidos com suas respectivas quantidades, e permitir finalizar a venda informando os dados do cliente, descontos, acréscimos, observação e parcelamento financeiro.
- **Relatório de vendas** – Oferecer um relatório de vendas, permitindo filtrar as vendas ocorridas por datas, e listar cada venda ocorrida agrupada pelo cliente, totalizando o valor da venda.
- **Relatório de contas** – Oferecer um relatório de contas a receber, permitindo filtrar as contas por datas, e listar cada uma das contas com informações como: emissão, vencimento, cliente, valor etc.

7.1.1 Index

No capítulo 5 abordamos o Design Pattern Front Controller pela primeira vez. É importante lembrar que para Martin Fowler um Front Controller é "um controlador que manipula todas as requisições do sistema". Para implementar um Front Controller, utilizamos o próprio *index.php*. A partir daquele instante, convencionamos que, para executar alguma classe controladora específica, seria necessário acessar *index.php?class=ClienteControl*. Porém, para executar uma ação específica, seria necessário acessar *index.php?class=ClienteControl&method=listar*, por exemplo.

Em um sistema grande, existem tarefas comuns a todas as páginas do sistema, como: verificação de permissão, carregamento das classes (Autoloader), carregamento do template do sistema, entre outras. Essas tarefas comuns podem ser realizadas justamente no *index.php*. O *index.php* a seguir, que será utilizado para a nova aplicação, define os Autoloaders, responsáveis pelo carregamento dos componentes construídos no diretório *Lib* (`ClassLoader`) e também para as classes da aplicação construídas no diretório *App* (`AppLoader`). Em seguida, verifica se há alguma requisição (`$_GET`). Se há uma requisição e se existe um parâmetro chamado

`class`, ele instanciará a classe correspondente identificada pela URL e executará seu método `show()` sobre essa classe. O método `show()` deverá decidir qual método deve ser executado internamente na classe.

Para desenvolver a nova aplicação proposta neste capítulo, vamos realizar uma melhoria no *index.php* construído no capítulo 5 para "encaixar" o conteúdo gerado por uma página dentro de um "template", ou seja, um layout predefinido em HTML, com alguns conteúdos como cabeçalho da página, menus de opções, entre outros. A figura 7.1 mostra o template escolhido, que também é baseado na biblioteca Bootstrap. Veja que na parte central do template será exibido o conteúdo da página (classe de controle) que por sua vez é identificada na URL em cada requisição.

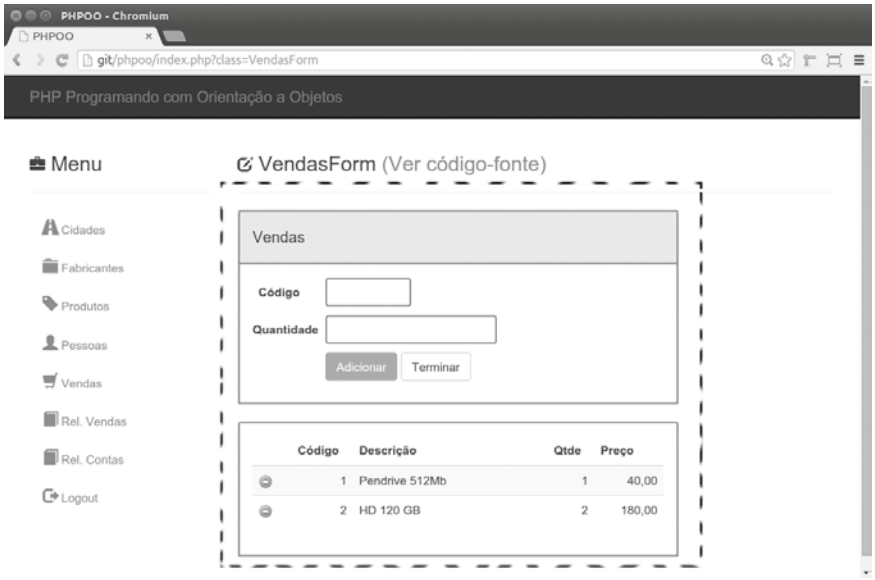

Figura 7.1 – Template.

Para "encaixar" o conteúdo do programa dentro do template é relativamente simples. Veja no programa a seguir que é realizada a leitura do conteúdo do arquivo de template que está localizado em *App/Templates/template.html*. O conteúdo desse arquivo é armazenado na variável `$template`. Logo em seguida, é verificado se há alguma requisição (`$_GET`) informando uma classe (`$_GET['class']`). Caso exista alguma requisição desse tipo, o conteúdo da página é exibido no momento em que executamos o seu método `show()`. Todo esse processo está "protegido" por um controle de exceções `try/catch`.

Para "capturar" o conteúdo gerado por uma página, usamos o controle de output do PHP, com as funções `ob_start()`, `ob_get_contents()` e `ob_end_clean()`, que "capturam" o conteúdo gerado por comandos como `echo` ou `print`. Dessa forma, a execução do

(classes de modelo, arquivos de configuração etc.) localizados principalmente na pasta *App*. A segunda forma é baixar uma estrutura pronta que preparamos para a criação de novos projetos que contém somente as classes criadas da pasta *Lib*, os arquivos do diretório principal e a pasta *App* somente com os diretórios vazios. Para isso, basta clonar o seguinte repositório público e iniciar o projeto a partir da estrutura criada:

```
git clone https://github.com/pablodalloglio/phpoo.git
```

7.2 Modelo

Para desenvolver nossa aplicação de vendas proposta neste capítulo, precisamos criar um modelo coerente e compreensível, que facilite o posterior desenvolvimento do mesmo. Vamos então criar um modelo de classes, que demonstrará o relacionamento entre os objetos da aplicação, e em seguida convertê-lo em um modelo relacional, com os relacionamentos entre as tabelas de um banco de dados.

7.2.1 Modelo de classes

A figura 7.2 apresenta o modelo de classes da aplicação proposta.

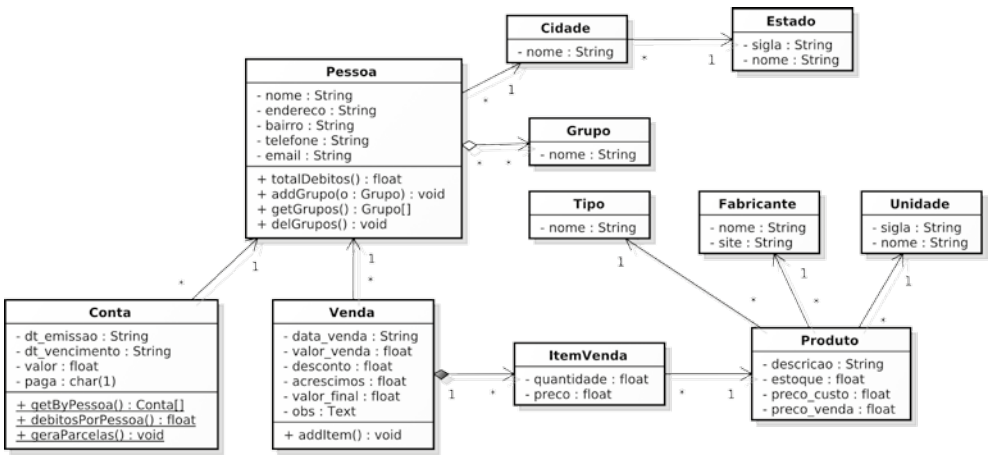

Figura 7.2 – Modelo de classes.

Temos a classe `Pessoa`, que armazenará os clientes do sistema. `Pessoa` tem associação com `Cidade` e esta com `Estado`. Uma `Pessoa` poderá ter vários grupos, por isso a agregação com a classe `Grupo`. Objetos da classe `Conta` estarão associados com a `Pessoa` para qual aquela `Conta` deve ser paga. Objetos da classe `Venda` estarão associados

7.2.2 Modelo relacional

Para armazenar os dados da aplicação de vendas, é fundamental modelarmos a estrutura de dados. Para tal, foi definido o modelo relacional apresentado pela figura 7.3.

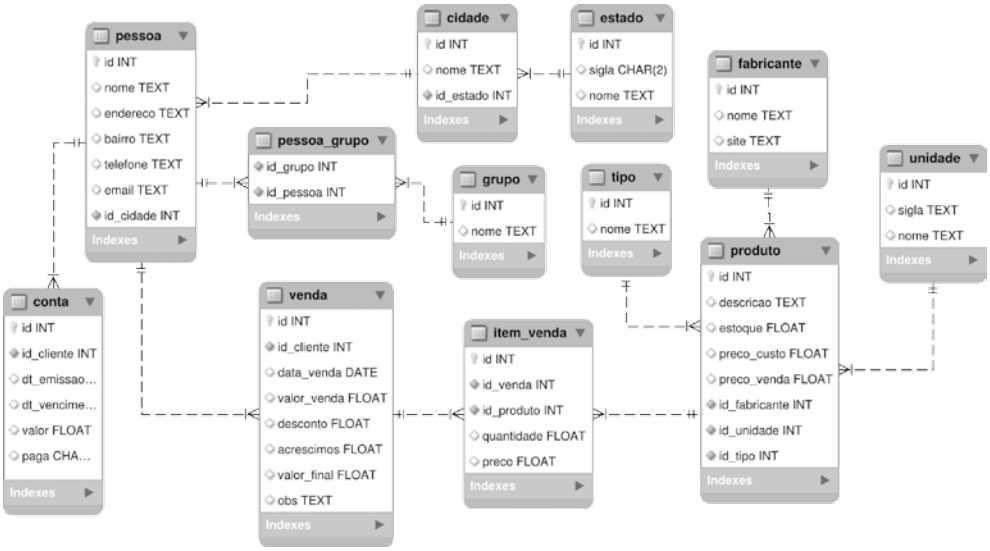

Figura 7.3 – Modelo relacional.

Para converter o modelo de classes no modelo relacional, é importante utilizar técnicas de mapeamento objeto relacional. Neste cenário proposto, essas técnicas se resumem em aplicar os seguintes padrões:

- **Chaves primárias** – Cada classe, ao ser representada por uma tabela, recebe uma chave primária. Assim, é possível visualizar a presença do campo `id` em cada uma das tabelas geradas.
- **Associações** – Relacionamentos de associação, presente entre várias classes como: `Pessoa` e `Cidade`, `Cidade` e `Estado`, `Produto` e `Tipo`, e várias outras, são convertidos automaticamente em chaves estrangeiras. Neste modelo usamos o padrão `id_<tabela>`. A direção da chave estrangeira segue a direção da associação, ou seja, parte sempre do objeto que contém a referência para o outro.
- **Composições** – Como a presente entre as classes `Venda` e `VendaItem`, são convertidas automaticamente em chaves estrangeiras, em que a direção da chave estrangeira parte sempre do objeto que representa a "parte" (`VendaItem`) para o objeto que representa o "todo" (`Venda`).

formato esperado pelo componente `CheckGroup`. Em seguida, o objeto é utilizado para preencher o formulário por meio do método `setData()`.

```
public function onEdit($param) {
    try {
        if (isset($param['key'])) {
            $id = $param['id']; // obtém a chave
            Transaction::open('livro'); // inicia transação com o BD
            $pessoa = Pessoa::find($id);
            $pessoa->ids_grupos = $pessoa->getIdsGrupos();
            $this->form->setData($pessoa); // lança os dados da pessoa no formulário
            Transaction::close(); // finaliza a transação
        }
    }
    catch (Exception $e) {
        // exibe a mensagem gerada pela exceção
        new Message('error', '<b>Erro</b>' . $e->getMessage());
        // desfaz todas alterações no banco de dados
        Transaction::rollback();
    }
}
```

Na figura 7.4 é apresentado o resultado final do formulário construído. O formulário em questão está no modo de edição.

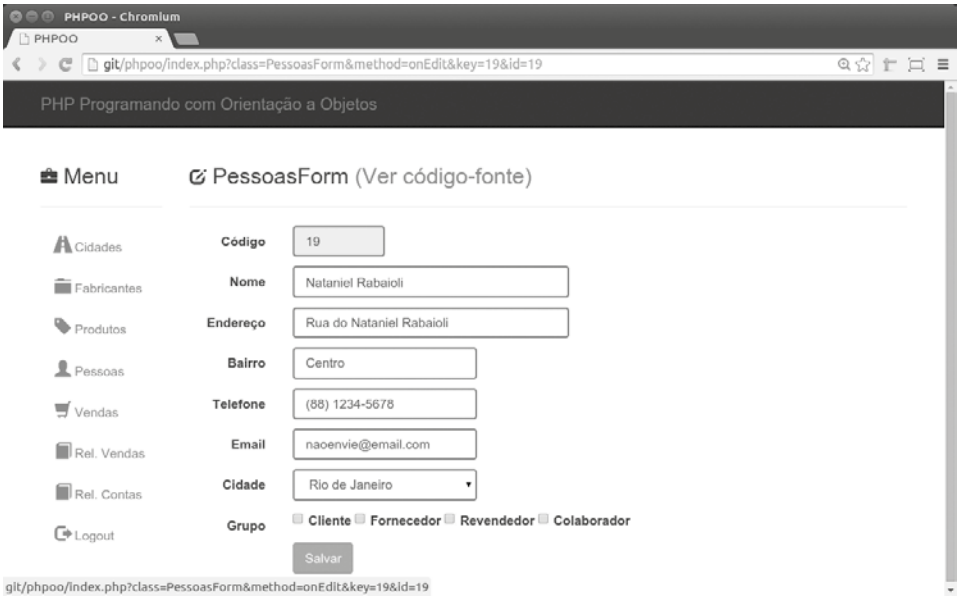

Figura 7.4 – Formulário de pessoas.

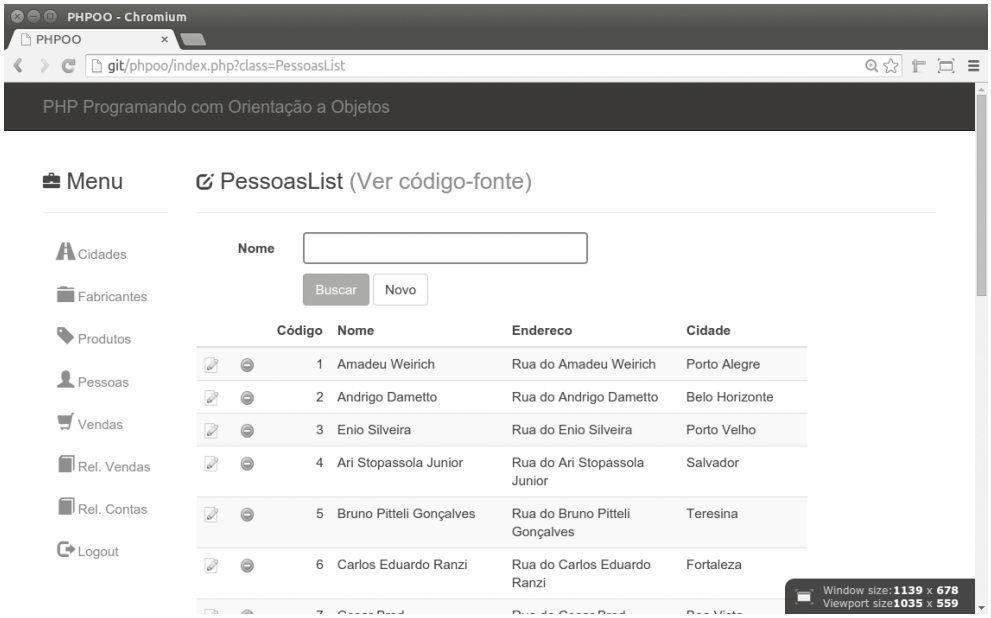

Figura 7.5 – Listagem de pessoas.

7.4.2 Criando Traits para ações comuns

Ao criarmos programas para cadastrar e listar registros do banco de dados, como foi o caso dos programas `PessoasForm` e `PessoasList`, criamos uma série de métodos para realizar ações comuns. No formulário de cadastro criamos ações para salvar (`onSave`) e editar (`onEdit`) registros. Na listagem criamos métodos para carregar (`onReload`) e excluir (`onDelete`, `Delete`) registros. Ao analisar os códigos desenvolvidos podemos concluir que é possível reaproveitar grandes porções de código-fonte desses métodos criados para criar novas classes. Podemos concluir que bastaria apenas "copiar e colar" (perdão pelo uso da expressão) essas classes alterando algumas pequenas definições, como é o caso do nome da conexão (`livro`), que pode ser diferente em outro caso, bem como o nome da classe de modelo (`Pessoa`), que certamente será diferente em outros casos. Porém "copiar e colar" certamente não é uma boa estratégia para ganhar produtividade, pois mesmo ganhando velocidade inicial, à medida que precisarmos realizar uma manutenção, teremos de lembrar de realizá-la em inúmeros locais. Uma estratégia melhor para reaproveitar trechos são os Traits.

Como vimos no capítulo 3, para atender à necessidade de compartilhar pequenos comportamentos entre diferentes classes, independentemente da hierarquia (superclasses), é implementado no PHP o conceito de Traits (traços). Um Trait é formado

```
            return $_SESSION[$var];
        }
    }

    public static function freeSession() {
        $_SESSION = array();
        session_destroy();
    }
}
```

7.4.6 Registro de vendas

O próximo passo no desenvolvimento de nossa aplicação é criar uma interface para o registro de vendas. O processo de vendas será dividido em duas etapas. A primeira etapa terá uma interface que permitirá ao usuário adicionar itens (produtos) à uma lista de objetos armazenada na sessão. A figura 7.9 demonstra essa interface na qual o usuário informa o código do produto, a quantidade e adiciona-os em uma lista armazenada em sessão. Essa classe será chamada de `VendasForm`.

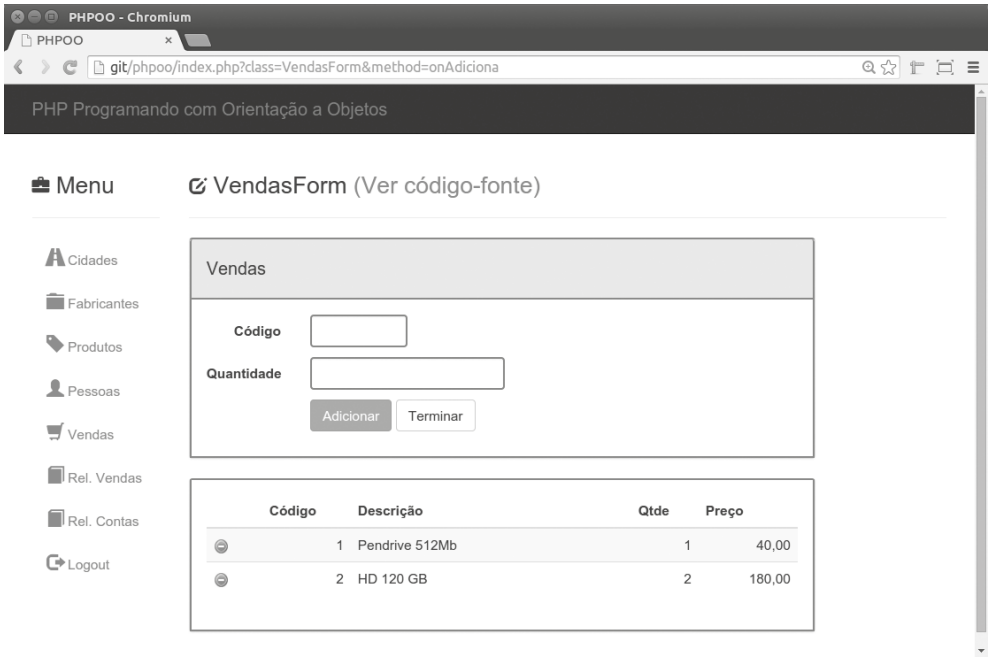

Figura 7.9 – Formulário de registro de venda.

A partir do momento em que o usuário clicar no botão "Terminar", será direcionado à outra interface, que por sua vez permitirá que o mesmo informe detalhes

da transação para que a mesma seja armazenada no banco de dados. Nesta etapa, ele informará: cliente, descontos, acréscimos, parcelamento e observação. A figura 7.10 demonstra essa tela de finalização, que irá registrar a venda na base de dados, bem como gerar as parcelas financeiras. Esta classe será chamada de `ConcluiVendaForm`.

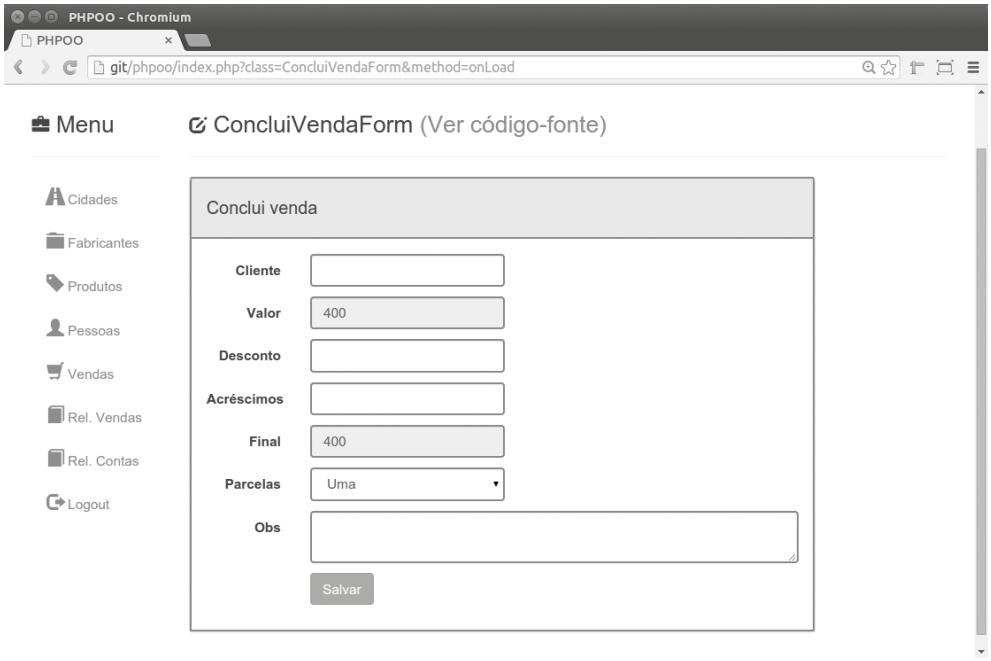

Figura 7.10 – Formulário de conclusão de venda.

7.4.6.1 Registro de itens da venda

Iniciaremos o desenvolvimento pela primeira tela, na qual o usuário informará os produtos e suas respectivas quantidades, que serão armazenados na sessão e exibidos em uma Datagrid. Essa classe inicia com a definição das classes necessárias, bem como de seus Namespaces. No método construtor, começamos iniciando a sessão ao instanciarmos a classe `Session`. Em seguida, criamos um formulário que será utilizado para informar os produtos que serão vendidos. Esse formulário terá dois campos: `id_produto` e `quantidade`, e duas ações: "Adicionar", que está vinculada ao método `onAdiciona()`, que receberá os dados do formulário e os adicionará à sessão, e "Terminar", que está vinculada ao método `onLoad()` da classe `ConcluiVendaForm`. Como esse método pertence à outra classe, provocará uma troca de tela no momento do clique, ou seja, o usuário será transportado para a classe `ConcluiVendaForm`.

7.4.7 Relatório de contas

Nas últimas seções, construímos programas que realizam cadastros, como o cadastro de produtos (`ProdutosForm`), e que também registram atividades de um processo, como o registro de vendas (`VendasForm`, `ConcluiVendaForm`). Agora chegou o momento de prepararmos a "saída" de informações para o usuário por meio de relatórios. Para tal, construiremos a classe `ContasReport`, que terá como responsabilidade gerar um relatório de contas com base em um intervalo de datas informadas pelo usuário.

> **Observação:** neste livro abordamos de maneira superficial a geração de relatórios. Caso queira ver em maior grau de profundidade, procure pelo livro *Criando relatórios com PHP* também publicado pela Novatec Editora. Ele aborda bibliotecas para geração de relatórios HTML, PDF, RTF e gráficos, relatórios tabulares, com filtros, seleção de colunas e ordenação, relatórios hierárquicos (quebras) e matriciais (cross-tab reports), gráficos gerenciais reais e documentos (notas fiscais e cartas), entre outros.

Para gerar um relatório de contas no formato HTML, utilizaremos a biblioteca de templates Twig, já abordada no capítulo 5. A interface do programa será como demonstrado na figura 7.12.

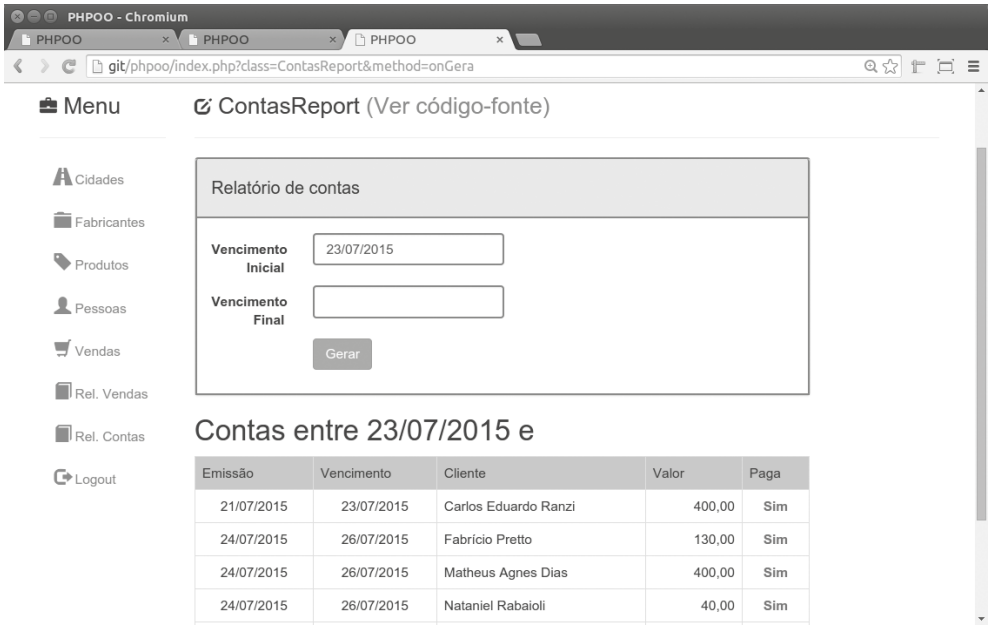

Figura 7.12 – Relatório de contas.

Na parte superior da tela teremos um formulário perguntando duas datas (inicial e final de vencimento). Com base nas datas informadas (que são opcionais) o programa irá gerar uma listagem no formato HTML sobre as contas com