PHP

Programando com Orientação a Objetos

Pablo Dall'Oglio

Novatec

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 arquivos, arrays, bancos de dados, entre outros.

1.1 O que é o PHP?

A linguagem de programação PHP, cujo logotipo vemos na Figura 1.1, foi criada no outono de 1994 por Rasmus Lerdorf. No início era formada por um conjunto de scripts voltados à criação de páginas dinâmicas que Rasmus utilizava para monitorar o acesso ao seu currículo na internet. À medida que essa ferramenta foi crescendo em funcionalidades, Rasmus teve de escrever uma implementação em C, a qual permitia às pessoas desenvolverem de forma muito simples suas aplicações para web. Rasmus nomeou essa versão de PHP/FI (Personal Home Pages/Forms Interpreter) e decidiu disponibilizar seu código na web, em 1995, para compartilhar com outras pessoas, bem como receber ajuda e correção de bugs.

Em novembro de 1997 foi lançada a segunda versão do PHP. Naquele momento, aproximadamente 50 mil domínios ou 1% da internet já utilizava PHP. No mesmo ano, Andi Gutmans e Zeev Suraski, dois estudantes que utilizavam PHP em um projeto acadêmico de comércio eletrônico, resolveram cooperar com Rasmus para

```
if ($valor_venda > 100)
{
    $resultado = 'muito caro';
}
else
{
    $resultado = 'pode comprar';
}
```

O mesmo código poderia ser escrito em uma única linha da seguinte forma:

```
$resultado = ($valor_venda > 100) ? 'muito caro' : 'pode comprar';
```

A primeira expressão é a condição a ser avaliada; a segunda é o valor atribuído caso ela seja verdadeira; e a terceira é o valor atribuído caso ela seja falsa.

1.6.2 WHILE

O `WHILE` é uma estrutura de controle similar ao `IF`. Da mesma forma, possui uma condição para executar um bloco de comandos. A diferença primordial é que o `WHILE` estabelece um laço de repetição, ou seja, o bloco de comandos será executado repetitivamente enquanto a condição de entrada dada pela expressão for verdadeira. Este comando pode ser interpretado como "ENQUANTO (expressão) FAÇA {comandos... }.".

A Figura 1.3 procura explicar o fluxo de funcionamento do comando `WHILE`. Quando a expressão é avaliada como `TRUE`, o programa parte para a execução de um bloco de comandos. Quando do fim da execução deste bloco de comandos, o programa retorna ao ponto inicial da avaliação e, se a expressão continuar verdadeira, o programa continua também com a execução do bloco de comandos, constituindo um laço de repetições, o qual só é interrompido quando a expressão avaliada retornar um valor falso (`FALSE`).

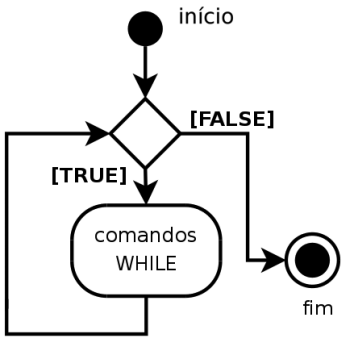

Figura 1.3 – Fluxo do comando `WHILE`.

```
var
dev
bin
etc
lib
usr
boot
home
```

1.10 Manipulação de strings

1.10.1 Declaração

Uma string é uma cadeia de caracteres alfanuméricos. Para declarar uma string podemos utilizar aspas simples ' ' ou aspas duplas " ".

```
$variavel = 'Isto é um teste';
$variavel = "Isto é um teste";
```

A diferença é que todo conteúdo contido dentro de aspas duplas é avaliado pelo PHP. Assim, se a string contém uma variável, esta variável será traduzida pelo seu valor.

```
<?php
$fruta = 'maçã';
print "como $fruta";      // resultado 'como maçã'
print 'como $fruta';      // resultado 'como $fruta'
?>
```

Também podemos declarar uma string literal com muitas linhas observando a sintaxe a seguir, na qual escolhemos uma palavra-chave (neste caso, escolhemos `CHAVE`) para delimitar o início e o fim da string.

```
<?php
$texto = <<<CHAVE
Aqui nesta área
você poderá escrever
textos com múltiplas linhas
CHAVE;
echo $texto;
?>
```

Resultado:

```
Aqui nesta área
você poderá escrever
textos com múltiplas linhas.
```

No exemplo a seguir, a função `strpos()` vasculha a variável `$minha_string` para encontrar em qualquer posição dentro dela a variável `$encontrar`:

```
<?php
$minha_string = 'O rato roeu a roupa do rei de Roma';
$encontrar = 'roupa';
$posicao = strpos($minha_string, $encontrar);
if ($posicao)
{
   echo "String encontrada na posição $posicao";
}
else
{
   echo "String não encontrada";
}
?>
```

Resultado:

```
String encontrada na posição 14
```

1.11 Manipulação de arrays

A manipulação de arrays no PHP é, sem dúvida, um dos recursos mais poderosos da linguagem. O programador que assimilar bem esta parte terá muito mais produtividade no seu dia-a-dia. Isto porque os arrays no PHP servem como verdadeiros contêineres, servindo para armazenar números, strings, objetos, dentre outros, de forma dinâmica. Além disso, o PHP nos oferece uma gama enorme 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 , ... )`.

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

ou

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

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

Capítulo 2

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 toda 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 já conhecemos do nosso dia-a-dia e sobre as quais possuímos maior compreensão.

2.1 Introdução

Neste capítulo abordaremos a implementação da orientação a objetos, seus diversos conceitos e técnicas das mais diversas formas, sempre com um exemplo ilustrado de código-fonte com aplicabilidade prática. Ao final do capítulo também abordaremos dois assuntos importantes: a manipulação de erros e a manipulação de arquivos XML. Antes de mais nada, no entanto, iremos rever os principais conceitos da programação estruturada.

2.1.1 Programação estruturada

A programação estruturada é um paradigma de programação que introduziu uma série de conceitos importantes na época em que foi criada e dominou a cena da engenharia de software durante algumas décadas. É baseada fortemente na modularização, cuja idéia é dividir o programa em unidades menores conhecidas por procedimentos ou funções. Essas unidades menores são construídas para desempenhar uma tarefa

Ao modelar uma classe `Conta` (bancária), são suas propriedades: `Agencia`, `Codigo`, `DataDeCriacao`, `Titular`, `Senha`, `Saldo` e se está `Cancelada`, bem como são seus métodos (funcionalidades): `Retirar()`, `Depositar()` e `ObterSaldo()`, dentre outros.

A seguir, veremos a representação gráfica de uma classe de acordo com o padrão UML:

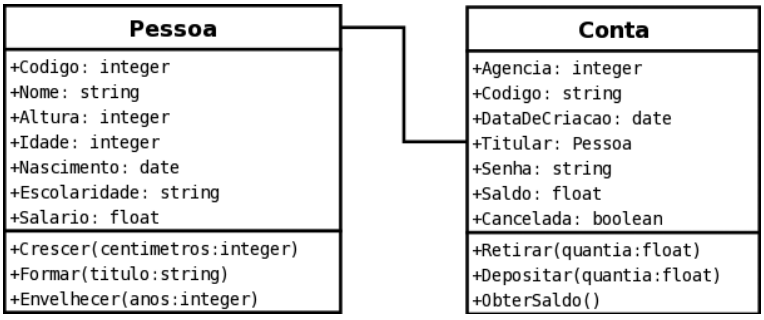

Figura 2.3 – Classes.

As classes são orientadas ao assunto, ou seja, cada classe é responsável por um assunto diferente e possui responsabilidade sobre o mesmo. Ela deve proteger o acesso ao seu conteúdo por meio de mecanismos como o de encapsulamento (visto a seguir). Dessa forma, criamos sistemas mais confiáveis e robustos. A classe `Pessoa` e a classe `Conta` são responsáveis pelas propriedades nelas contidas, evitando que essas propriedades contenham valores inconsistentes ou sejam manipuladas indevidamente.

Os membros de uma classe são declarados na ordem: primeiro as propriedades (mediante o operador `var`) e, em seguida, os métodos (pelo operador `function`, também utilizado para declarar funções). As classes `Pessoa` e `Conta` são apresentadas a seguir:

Pessoa.class.php

```
<?php
class Pessoa
{
    var $Codigo;
    var $Nome;
    var $Altura;
    var $Idade;
    var $Nascimento;
    var $Escolaridade;
    var $Salario;

    /* método Crescer
     * aumenta a altura em $centimetros
     */
```

exclusivamente por métodos dela mesma, que são implementações projetadas para manipular essas propriedades da forma correta. As propriedades nunca devem ser acessadas diretamente de fora do escopo de uma classe, pois dessa forma a classe não fornece mais garantias sobre os atributos que contém, perdendo, assim, a responsabilidade sobre eles.

Na Figura 2.6, vemos a representação gráfica de um objeto. As propriedades são os pequenos blocos no núcleo do objeto; os métodos são os círculos maiores na parte externa do mesmo. Esta figura foi construída para demonstrar que os métodos devem ser projetados de forma a protegerem suas propriedades, fornecendo interfaces para a manipulação destas.

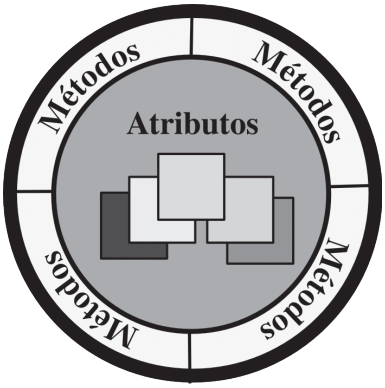

Figura 2.6 – Encapsulamento.

Para atingir o encapsulamento, uma das formas é definindo 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
`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 em si). Na UML, simbolizamos com um (-) em frente à 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 (#) em frente à propriedade.
`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 (+) em frente à propriedade.

```
echo "============================:\n";
Aplicacao::Sobre();
?>
```

Resultado:

```
Informações sobre a aplicação:
============================:
Esta aplicação está licenciada sob a GPL
Para maiores informações, www.fsf.org
Contate o autor através do e-mail autor@aplicacao.com.br
```

2.10 Associação, agregação e composição

2.10.1 Associação

Associação é a relação mais comum entre dois objetos, de modo que um possui uma referência à posição da memória onde o outro se encontra, podendo visualizar seus atributos ou mesmo acionar uma de suas funcionalidades (métodos). A forma mais comum de implementar uma associação é ter um objeto como atributo de outro. Veja o exemplo a seguir, no qual criamos um objeto do tipo `Produto` (já criado anteriormente) e outro do tipo `Fornecedor`. Um dos atributos do produto é o fornecedor. Observe, na Figura 2.9, como se dá a interação.

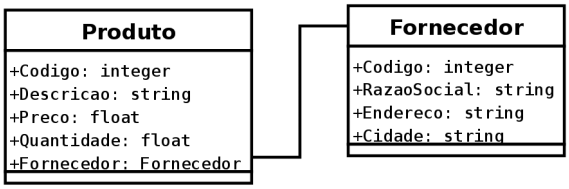

Figura 2.9 – Associação.

Fornecedor.class.php

```
<?php
class Fornecedor
{
    var $Codigo;
    var $RazaoSocial;
    var $Endereco;
    var $Cidade;
}
?>
```

definida por `set_error_handler()` como sendo a função a ser invocada quando algum erro ocorrer. Dentro desta função (que recebe o arquivo e a linha em que ocorreu o erro, além do número e a mensagem de erro ocorrido) podemos exibir ou suprimir a exibição do erro, gravá-lo em um banco de dados ou gravar em um arquivo de log, como fizemos no exemplo demonstrado. No exemplo, todas as mensagens (`ERROR`, `WARNING` e `NOTICE`) são armazenadas em um arquivo de log; somente as de `WARNING` e `ERROR` são exibidas na tela, e as de `ERROR` causam parada na execução da aplicação pelo comando `die`. A desvantagem deste tipo de abordagem é que concentramos todo tratamento de erro em uma única função genérica, quando muitas vezes precisamos analisar caso a caso para optar por uma determinada ação.

2.17.4 Tratamento de exceções

O PHP5 implementa o conceito de tratamento de exceções, da mesma forma que ele é implementado em linguagens como C++ ou Java. Uma exceção 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 estes 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.

O tratamento de erros ocorre em dois níveis. Quando executamos um conjunto de operações que pode resultar em algum tipo de erro, monitoramos essas operações, escrevendo o código dentro do bloco `try`. Dentro das operações críticas, quando ocorrer algum erro, devemos fazer uso do comando `throw` para "lançar" uma exceção (objeto `Exception`), isto é, para interromper a execução do bloco contido na cláusula `try`, a qual recebe esta exceção e repassa para outro bloco de comandos `catch`. Dentro do bloco de comandos `catch`, programamos o que deve ser realizado quando da ocorrência da exceção, podendo emitir uma mensagem ao usuário, interromper a execução da aplicação, escrever um arquivo de log no disco, dentre outros. O interessante desta abordagem é que a ação resultante do erro ocorrido fica totalmente isolada, externa ao contexto do código gerador da exceção. Esta modularidade permite mudarmos o comportamento de como é realizado este tratamento de erros, sem alterar o bloco código principal.

Capítulo 3

Manipulação de dados

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

Peter Drucker

Este capítulo abordará a manipulação de bancos de dados e suas nuances. Em um primeiro momento, iremos recapitular como se dá a manipulação de dados utilizando os comandos nativos que o PHP nos oferece para cada sistema gerenciador de banco de dados. Em seguida, estudaremos a utilização da biblioteca PDO, que nos permite abstrair o acesso a diferentes mecanismos de bancos de dados. Posteriormente, visando evitar a utilização direta de comandos SQL em nossas aplicações, construiremos um conjunto de classes que irão nos oferecer uma interface totalmente orientada a objetos para abstrair todos os aspectos relativos a manipulação e armazenamento de informações no banco de dados por meio da linguagem SQL, tornando isto uma tarefa simples, trivial e, sobretudo, transparente ao programador.

3.1 Acesso nativo

3.1.1 Introdução

No atual mundo das aplicações de negócio, o armazenamento de informações de uma organização é implementado por diferentes mecanismos de bancos de dados em decorrência, muitas vezes, da utilização de sistemas de diferentes fornecedores e tecnologias. Alguns dados relativos a recursos humanos podem estar armazenados em um banco de dados DB2, ao passo que os dados financeiros podem estar armazenados em um Oracle, as estatísticas de acesso ao site da empresa podem estar em um MySQL, e o Data Warehouse, que consiste de várias informações estratégicas, pode estar em um PostgreSQL. A diversidade reside nos mais diversos fornecedores de bancos de dados existentes, cada qual com métodos diferentes (interfaces) de acesso

Resultado:

```
1 - Érico Veríssimo
2 - John Lennon
3 - Mahatma Gandhi
4 - Ayrton Senna
5 - Charlie Chaplin
6 - Anita Garibaldi
7 - Mário Quintana
```

3.2 PDO :: PHP Data Objects

3.2.1 Introdução

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 à base de dados, cada qual com sua implementação particular, não havendo real consistência entre as interfaces das mesmas (Oracle, MySQL, PostgreSQL, SQL Server etc.).

Em razão da crescente adoção do PHP, surgiu a necessidade de unificar o acesso às diferentes extensões de bancos de dados presentes no PHP. Assim surgiu a PDO (PHP Data Objects), cujo objetivo é prover uma API limpa e consistente, unificando a maioria das características presentes nas extensões de acesso a banco de dados.

A PDO não é uma biblioteca completa para abstração do acesso à base de dados, uma vez que ela não faz a leitura e tradução das instruções SQL, adaptando-as aos mais diversos drivers de bancos de dados existentes. Ela simplesmente unifica a chamada de métodos, delegando-os para as suas extensões correspondentes e faz uso do que há de mais recente no que diz respeito à orientação a objetos presente no PHP5.

Para conectar em bancos de dados diferentes, a única mudança é na string de conexão. Veja a seguir alguns exemplos nos mais diversos bancos de dados.

Banco	String de conexão
SQLite	`new PDO('sqlite:teste.db');`
FireBird	`new PDO("firebird:dbname=C:\\base.GDB", "SYSDBA", "masterkey");`
MySQL	`new PDO('mysql:unix_socket=/tmp/mysql.sock;host=localhost;port=3307; dbname=livro', 'user', 'senha');`
Postgres	`new PDO('pgsql:dbname=example;user=user;password=senha;host=localhost');`

entidade (tabela) do banco de dados irá atuar, portanto será necessário termos um método para definir a entidade, o qual chamaremos de `setEntity()`, e um para retornar o nome desta entidade, que será chamado de `getEntity()`.

Apesar de estarmos utilizando uma estrutura orientada a objetos para definir as consultas enviadas ao banco de dados, em um determinado momento precisaremos obter esse comando no formato string para poder realizar a consulta, e é este o objetivo do método `getInstruction()`, que na superclasse é um método abstrato, portanto sem implementação, o que obriga a cada classe-filha implementá-lo. Cada classe-filha possui uma forma de realizar a consulta ao banco de dados (formato) diferente, portanto convém que cada uma implemente o seu `getInstruction()` de maneira peculiar. Veja na Figura 3.3 como será nossa estrutura de classes. A classe `TSqlInstruction` irá fornecer os métodos comuns e cada uma das classes-filha oferecerá sua variação do método `getInstruction()`.

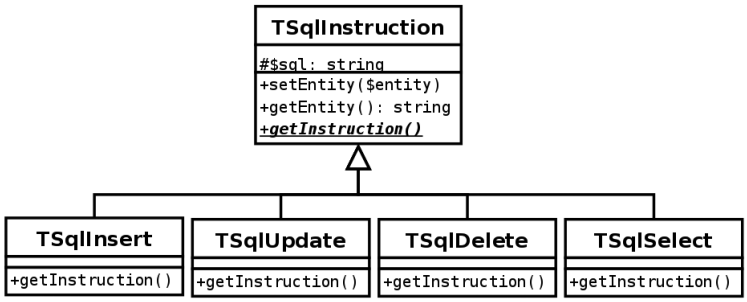

Figura 3.3 – Estrutura de classes SQL.

Durante o projeto destas classes percebemos alguns fatores interessantes: a maioria das instruções SQL (com exceção do `INSERT`) possui um critério de seleção de dados que se traduz em uma cláusula `WHERE`. É assim com o `Update`, o `Delete` e o `Select`. Tal expressão (filtro) pode ser uma instrução complexa, composta de operadores lógicos (`AND`, `OR`), operadores de comparação (<,>,=,<>), dentre outros. Poderíamos escrever métodos para definir esse critério de seleção em cada uma das classes que necessitam deste recurso, mas estaríamos duplicando o código-fonte sem necessidade. A solução que adotaremos é a criação de mais uma classe no sistema, a qual irá se chamar `TCriteria` e será responsável pela criação dessas expressões, utilizando operadores lógicos. Para relacionar um critério (`TCriteria`) a uma instrução SQL (`TSqlInstruction`), utilizamos um relacionamento do tipo composição, de modo que a instrução SQL terá uma referência ao objeto que contém o critério de seleção, como demonstrado na Figura 3.4. Não entraremos em detalhes sobre a estrutura da classe `TCriteria` neste momento.

tamento do método `getInstruction()` que retornará a instrução SQL montada e pronta para ser executada sobre o banco de dados. Para tanto, ela se utilizará das definições geradas pelo método `setRowData()` e pelo método `setEntity()` da superclasse. Veja a seguir o diagrama que representa a classe `TSqlInsert`.

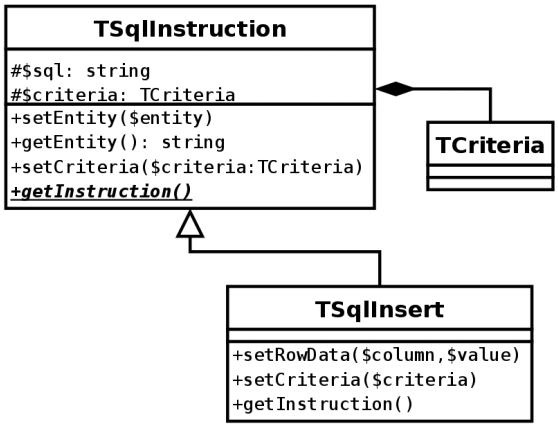

Figura 3.11 – Classe `TSqlInsert`.

TSqlInsert.class.php

```
<?php
/*
 * classe TSqlInsert
 * Esta classe provê meios para manipulação de uma instrução de INSERT no banco de dados
 */
final class TSqlInsert extends TSqlInstruction
{
    /*
     * método setRowData()
     * Atribui valores à determinadas colunas no banco de dados que serão inseridas
     * @param $column = coluna da tabela
     * @param $value = valor a ser armazenado
     */
    public function setRowData($column, $value)
    {
        // monta um array indexado pelo nome da coluna
        if (is_string($value))
        {
            // adiciona \ em aspas
            $value = addslashes($value);
            // caso seja uma string
            $this->columnValues[$column] = "'$value'";
        }
```

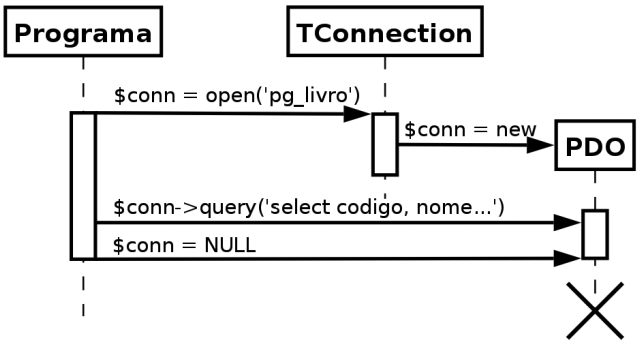

Figura 3.16 – Interação entre as classes `TConnection` e `PDO`.

No programa a seguir, estamos selecionando o registro de código "1" da tabela de famosos e exibindo-o na tela. No primeiro bloco `try/catch`, estamos realizando esta operação com o banco de dados MySQL, configurado no arquivo `my_livro.ini`; no segundo bloco `try/catch`, estamos realizando essa operação com o banco de dados PostgreSQL, configurado no arquivo `pg_livro.ini`. Antes de tudo, estamos instanciando um objeto da classe `TSqlSelect`, para definir a tabela sobre a qual atuaremos e quais colunas retornaremos, além de instanciar um objeto da classe `TCriteria`, para definir o critério de seleção dos dados.

connection.php

```
<?php
/*
 * função __autoload()
 * Carrega uma classe quando ela é necessária,
 * ou seja, quando ela é instancia pela primeira vez.
 */
function __autoload($classe)
{
    if (file_exists("app.ado/{$classe}.class.php"))
    {
        include_once "app.ado/{$classe}.class.php";
    }
}

// cria instrução de SELECT
$sql = new TSqlSelect;
// define o nome da entidade
$sql->setEntity('famosos');
// acrescenta colunas à consulta
$sql->addColumn('codigo');
$sql->addColumn('nome');
```

e é exatamente por isso que estamos utilizando o controle de transações. Caso ocorra algum erro na primeira execução, automaticamente uma exceção é lançada e a aplicação é direcionada para o bloco `catch`, ignorando as demais instruções.

Na Figura 3.17 procuramos demonstrar como é a interação entre as classes participantes de uma transação. Veja que o programa abre uma transação executando o método `open()` da classe `TTransaction`. Este método, por sua vez, executa o método `open` da classe `TConnection`, obtendo um objeto de conexão PDO, e armazena-o na propriedade estática `$conn`. Sempre que o programa quiser essa variável de conexão, ele irá executar o método estático `get()`, que retornará a conexão da transação ativa. De posse da variável de conexão (objeto `$conn`), o programa poderá enviar diversas consultas ao banco de dados por meio do método `query()`. Ao final, quando desejarmos finalizar a transação, executaremos o método `close()` da classe `TTransaction`, que, por sua vez, eliminará o objeto PDO, atribuindo `NULL` à variável de conexão.

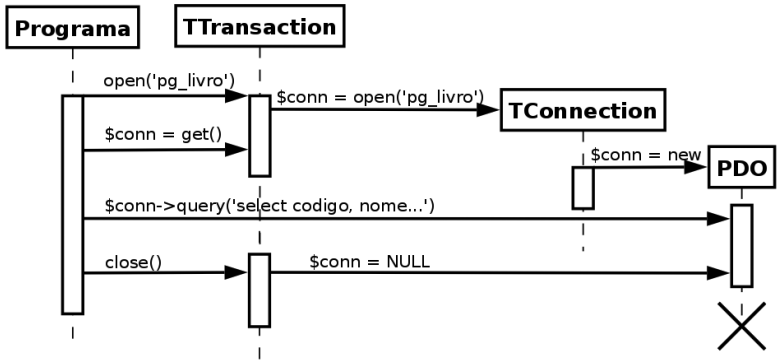

Figura 3.17 – Interação entre as classes durante a transação.

transaction.php

```
<?php
/*
 * função __autoload()
 * carrega uma classe quando ela é necessária,
 * ou seja, quando ela é instancia pela primeira vez.
 */
function __autoload($classe)
{
    if (file_exists("app.ado/{$classe}.class.php"))
    {
        include_once "app.ado/{$classe}.class.php";
    }
}
```

Capítulo 4

Mapeamento Objeto-Relacional

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

Blaise Pascal

No capítulo anterior revemos como se dá a manipulação de dados em bases relacionais com PHP por meio da utilização da linguagem SQL. Também criamos uma API orientada a objetos para manipulação de dados. Agora podemos subir um nível de complexidade e pensar em nossa aplicação como um conjunto de objetos que é o nosso modelo conceitual. Quando trabalhamos com um conjunto de objetos, precisamos persistir estes objetos na base de dados, ou seja, armazená-los e permitir posterior recuperação. 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 Persistência

4.1.1 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.), a 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

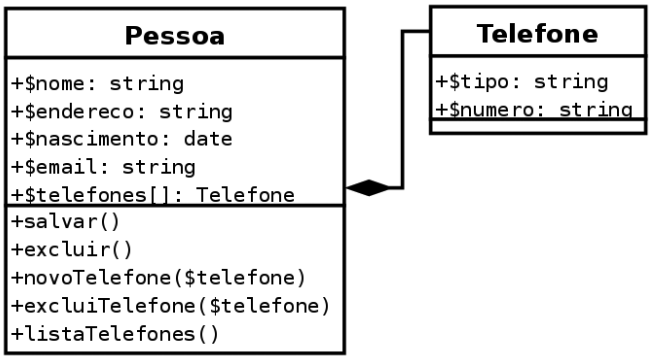

Figura 4.4 – Composição.

No nosso modelo de dados, precisamos criar as chaves de cada tabela (Identity Field). Em seguida, adicionamos uma chave estrangeira na tabela `Telefone` (`id_pessoa`), apontando para a chave primária da tabela `Pessoa`, como demonstrado na Figura 4.5.

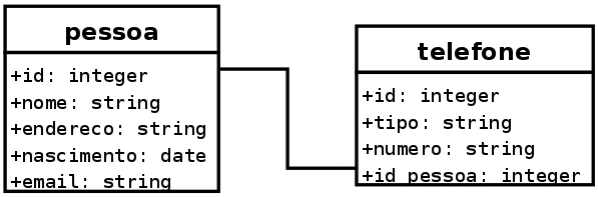

Figura 4.5 – Mapeando uma composição.

4.2.3 Association Table Mapping

Até o momento vimos como se dá o mapeamento de associações e de composições no banco de dados por meio do design pattern conhecido como "Foreign Key Mapping". Um outro relacionamento importante é a agregação, no qual também temos uma relação todo/parte como na composição, mas, desta vez, o objeto parte não está intrinsecamente ligado ao todo, ou seja, quando destruímos o objeto todo, as partes ainda continuam a existir, isto porque uma parte pode pertencer a diferentes objetos. Vamos exemplificar.

No caso a seguir, temos uma agregação em que um objeto `CestaCompras` representa uma cesta de compras em um sistema de comércio eletrônico. Uma cesta de compras simboliza os vários produtos que um cliente resolve comprar. No objeto `CestaCompras` temos atributos como a data em que a compra foi realizada (`$data`) e o cliente para o qual aquela cesta foi constituída (`$cliente`). Na cesta de compras temos métodos como o `insereProduto()`, o qual irá receber um objeto do tipo `Produto` e inserir na cesta

4.2.5 Concrete Table Inheritance

A segunda forma de mapearmos um relacionamento de herança para o banco de dados é termos uma tabela para cada classe "folha", ou para cada classe concreta da hierarquia. Dessa forma, teríamos em nosso exemplo duas tabelas: uma para armazenar os objetos `Livro` e outra para `DVD`. Cada tabela deve ter, além dos atributos da própria classe que ela está mapeando, todos os atributos da classe-pai. Assim, a tabela `livro` iria conter os atributos da classe `Livro` mais os atributos da classe `Material`, bem como a tabela `dvd` iria conter os atributos da classe `DVD` mais os atributos da classe `Material`, como demonstrado na Figura 4.11.

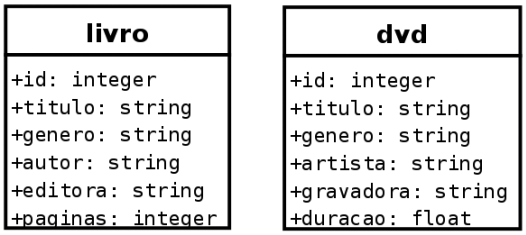

Figura 4.11 – Concrete Table Inheritance.

A vantagem deste pattern sobre o anterior reside no fato de que cada tabela contém somente os campos relativos ao seu contexto, diferentemente do "Single Table Inheritance", no qual temos campos irrelevantes que podem ficar vazios.

A dificuldade em implementar este pattern reside no fato de que o objeto é único na memória, independente se ele é um `Livro` ou `DVD`, ele faz parte da mesma hierarquia de classes e os objetos devendo ser tratados com unicidade também no banco de dados. Dessa forma, temos de ter cuidado para que o `ID` não fique duplicado nas tabelas, ou seja, para que não exista na tabela `livro` um registro com um `ID` que exista na tabela `dvd`, e vice-versa. A unicidade no `ID` deve ser estendida para o universo das duas tabelas e não somente para uma. Imagine uma tela em que o usuário pode selecionar qualquer tipo de material, seja `Livro` ou `DVD`. Adotando essa estratégia, teríamos de realizar duas consultas ao banco de dados (uma para cada tabela) ou usar um join, agrupando os resultados em uma única consulta. Neste caso, fica mais claro ainda a necessidade de não termos `IDs` que se repetem entre as tabelas.

4.2.6 Class Table Inheritance

A terceira forma de mapearmos uma herança para o banco de dados é criar uma tabela para cada classe da estrutura de herança e relacionar estas tabelas por meio de pares de chaves primárias e chaves estrangeiras, como visto na Figura 4.12.

```
// recupera um objeto e realiza alteração
$objeto= new ProdutoGateway;
$objeto->getObject(2);
$objeto->estoque   = $objeto->estoque*2;
$objeto->descricao = 'Salaminho Italiano';
$objeto->update();

// exclui o produto vinho da tabela
$vinho->delete();
?>
```

Resultado:

```
INSERT INTO Produtos (id, descricao, estoque, preco_custo) VALUES ('1', 'Vinho Cabernet', '10', '10')
INSERT INTO Produtos (id, descricao, estoque, preco_custo) VALUES ('2', 'Salame', '20', '20')
SELECT * FROM produtos where id='2'
UPDATE produtos set descricao = 'Salaminho Italiano', estoque = '40', preco_custo = '20'
   WHERE id = '2'
DELETE FROM produtos where id='1'
```

4.4.3 Active Record

O pattern Active Record é muito parecido com o Row Data Gateway. A diferença primordial é que o pattern Row Data gateway não possui nenhum método pertencente ao modelo de negócios, somente métodos de acesso à base de dados. Quando adicionamos lógica de negócio, ou seja, métodos que tratam de implementar características do modelo de negócios a um Row Data Gateway, temos um Active Record. Este pattern pode ser utilizado com sucesso onde o modelo de negócios se parece bastante com o modelo de dados.

Os patterns vistos anteriormente (Table Data Gateway e Row Data Gateway) proviam uma camada de acesso ao banco de dados para a camada superior (modelo conceitual). Com o pattern Active Record, temos uma única camada, na qual temos lógica de negócios (modelo conceitual) e métodos de persistência do objeto na base de dados (gateway). Um Active Record deve prover o mesmo que um Row Data Gateway, além de implementar métodos do modelo conceitual (lógica de negócios).

Um Active Record, assim como o Row Data Gateway, pode possuir métodos *getters* e *setters* para realizar algumas conversões de tipo entre uma propriedade do objeto do modelo conceitual e uma coluna de uma tabela do banco de dados. Exemplo disso é a conversão de arrays ou mesmo de um objeto relacionado. Um método `__get()` pode instanciar automaticamente um objeto relacionado baseado em seu `ID`.

No programa a seguir temos basicamente a repetição do que vimos no exemplo anterior do pattern Row Data Gateway. A grande diferença está agora na presença

Um Repository utiliza outros patterns já vistos neste livro, como o Query Object (classes `TSqlInstruction/TCriteria`). O seu funcionamento inicia pela definição de algum critério de seleção de objetos (`$criteria->add(new TFilter('idade', '>', 20))`) por parte do programador (Programa). Este, então, passa o critério para o Repository por meio de algum método (obter coleção, remover coleção, contar elementos etc.), sendo que esses métodos atuam sobre uma coleção de objetos que satisfazem os critérios definidos, alterando-os ou retornando-os. Na Figura 4.23 vemos o funcionamento de um Repository Pattern.

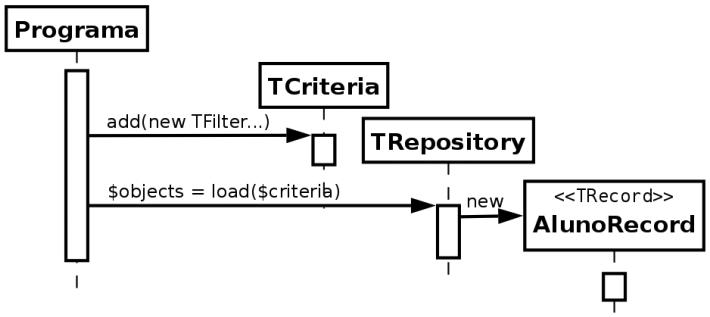

Figura 4.23 – Repository Pattern.

O programador não precisa saber quais instruções SQL estão sendo executadas dentro de cada um desses métodos, ele só precisa conhecer o método necessário para desempenhar determinada função e passar o critério de seleção desejado.

Para implementar um Repository, criaremos a classe `TRepository`, a qual implementará os métodos necessários para a manipulação de coleções de objetos. O método `load()` será responsável por carregar uma coleção de objetos, ao passo que o método `delete()` irá excluir uma coleção de objetos, e o método `count()` irá contar quantos objetos satisfazem um determinado critério. Em seu método construtor, a `TRepository` recebe o nome da classe que irá manipular, ou seja, a classe à qual pertence a coleção de objetos manipulada.

TRepository.php

```
<?php
/*
 * classe TRepository
 *  esta classe provê os métodos necessários para manipular coleções de objetos.
 */
final class TRepository
{
   private $class; // nome da classe manipulada pelo repositório
```

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 esta camada da aplicação, precisamos nos preocupar com a sua interface com o usuário. Neste capítulo, desenvolveremos uma série de classes com o objetivo de construir o visual da aplicação de forma orientada a objetos. Para isso, na primeira parte deste capítulo construiremos classes para exibição de textos, imagens e tabelas, dentre outros elementos gráficos. Na segunda, desenvolveremos classes com o objetivo de coordenar o fluxo de execução e controle da aplicação por meio de parâmetros.

5.1 Introdução

Se você programa em alguma linguagem para web há algum tempo, já deve ter percebido o quanto a exibição de tags HTML deixa o código confuso e pouco legível quando temos de concatenar expressões que mesclam código HTML e variáveis da aplicação PHP.

Ao longo deste capítulo criaremos alguns componentes de apresentação, como imagens e textos, e alguns contêineres para exibir tabelas, painéis e janelas, além de alguns componentes de controle, como diálogos de mensagem e controladores de página de forma orientada a objetos.

O objetivo das classes que desenvolveremos é justamente o de substituir a utilização de tags presentes no código HTML pela utilização dessas classes e seus métodos,

5.2.1 Elementos HTML

Como descreveremos, ainda neste capítulo, classes que precisam exibir diretamente código HTML em tela, consideramos melhor criar uma classe para manipular as tags HTML. Assim, podemos exibir qualquer elemento (tag) por uma estrutura orientada a objetos, sem a necessidade de utilizar comandos como o `echo` ou `print` diretamente no código da aplicação.

Para atingir esse objetivo, iremos criar a classe `TElement`. Esta classe representa um elemento HTML (`<p>`, `<img>`, `<table>`) e recebe em seu método construtor o nome do elemento (tag), exibindo este elemento na tela do usuário. É importante notar que uma tag possui propriedades e, neste caso, optamos por atribuir tais propriedades diretamente ao objeto. Lembre-se que ao atribuirmos uma propriedade que não existe ao objeto, automaticamente o método `__set()` interceptará esta atribuição. Assim sendo, faremos com que as propriedades da tag sejam armazenadas no array `$properties`, o qual será lido no momento da exibição da tag de abertura pelo método `open()`. Na Figura 5.3, temos a classe `TElement`.

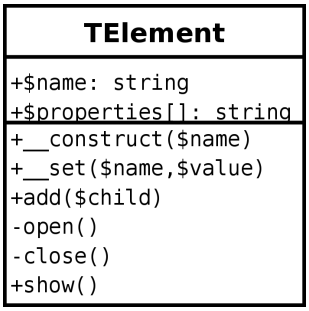

Figura 5.3 – Classe `TElement`.

TElement.class.php

```
<?php
/**
 * classe TElement
 * classe para abstração de tags HTML
 */
class TElement
{
    private $name;          // nome da TAG
    private $properties;    // propriedades da TAG
    /**
     * método construtor
     * instancia uma tag html
     * @param  $name    = nome da tag
     */
```

TImage.class.php

```
<?php
/**
 * classe TImage
 * classe para exibição de imagens
 */
class TImage extends TElement
{
    private $source;      // localização da imagem
    /**
     * método construtor
     * instancia objeto TImage
     * @param  $source = localização da imagem
     */
    public function __construct($source)
    {
        parent::__construct('img');
        // atribui a localização da imagem
        $this->src = $source;
        $this->border = 0;
    }
}
?>
```

5.2.3.1 Exemplo

No exemplo a seguir, exibiremos, por meio da utilização da classe recém-criada, duas imagens na tela. Mostraremos primeiro o logotipo do gnome (`gnome.png`) e depois o logotipo do gimp (`gimp.png`), resultando na página exibida pela Figura 5.9.

Figura 5.9 – Exibindo imagens com a classe `TImage`.

```
    $total += $pessoa[3];
    $i++;
}

// instancia uma linha para o totalizador
$linha = $tabela->addRow();

// adiciona células
$celula= $linha->addCell('Total');
$celula->colspan = 3;

$celula = $linha->addCell($total);
$celula->bgcolor = "#a0a0a0";
$celula->align   = "right";

// exibe a tabela
$tabela->show();
?>
```

Neste novo exemplo, demonstramos o objeto tabela contendo objetos de tipos diferentes, como um objeto do tipo `TImage` (imagem) e `TParagraph` (parágrafo). O objetivo principal é demonstrar o objeto `TTable` como contêiner, capaz de conter, agrupar e organizar outros objetos que ofereçam em sua interface o método `show()`. Veja na Figura 5.14 o resultado.

Figura 5.14 – Tabela orientada a objetos.

painel.php

```
<?php
// inclui as classes necessárias
include_once 'app.widgets/TStyle.class.php';
include_once 'app.widgets/TElement.class.php';
include_once 'app.widgets/TPanel.class.php';
include_once 'app.widgets/TImage.class.php';
include_once 'app.widgets/TParagraph.class.php';

// instancia novo painel
$painel = new TPanel(400,300);

// coloca objeto parágrafo na posição 10,10
$texto = new TParagraph('isso é um teste, x:10,y:10');
$painel->put($texto, 10,10);

// coloca objeto parágrafo na posição 200,200
$texto = new TParagraph('outro teste, x:200,y:200');
$painel->put($texto, 200,200);

// coloca objeto imagem na posição 10,180
$texto = new TImage('app.images/gnome.png');
$painel->put($texto, 10,180);

// coloca objeto imagem na posição 240,10
$texto = new TImage('app.images/gimp.png');
$painel->put($texto, 240,10);
$painel->show();
?>
```

5.3.3 Janelas

Uma janela pop-up é uma janela que se abre sobre a janela principal da aplicação, geralmente sem muitas opções e de tamanho reduzido, para exibir algumas informações como o resultado de uma operação, uma mensagem ao usuário, um formulário de dados, uma propaganda, dentre outros. Entretanto, abrir janelas é uma prática não muito bem aceita no ambiente web, e a maioria dos navegadores de última geração já conta com ferramentas de bloqueio para abertura de janelas pop-up.

Apesar disso, é interessante termos disponível esse recurso sempre que quisermos dar um destaque maior para um evento que ocorre. Para contornar esta situação, uma das formas de continuar exibindo uma janela para o usuário é simular a abertura de uma janela utilizando uma camada `<div>` com um botão de fechar (para esconder a

Figura 5.20 – Janelas construídas com a classe `TWindow`.

window.php

```
<?php
/*
 * função __autoload()
 * carrega as classes necessárias sob demanda
 */
function __autoload($class)
{
    include_once("app.widgets/{$class}.class.php");
}

// instancia um objeto TWindow nas coordenadas 20,20 contendo um texto
$janela1 = new TWindow('janela1');
$janela1->setPosition(20,20);
$janela1->setSize(200,200);
$janela1->add(new TParagraph('conteúdo da janela 1'));
$janela1->show();

// instancia um objeto TWindow nas coordenadas 300,20 contendo uma imagem
$janela2 = new TWindow('janela2');
$janela2->setPosition(300,20);
$janela2->setSize(200,200);
$janela2->add(new TImage('app.images/gimp.png'));
$janela2->show();
```

```
    if (file_exists("app.widgets/{$classe}.class.php"))
    {
        include_once "app.widgets/{$classe}.class.php";
    }
}

// exibe uma mensagem de informação
new TMessage('info', 'Esta ação é inofensiva, isto é só um lembrete');
?>
```

No exemplo a seguir, estamos fazendo uso da classe `TMessage` para exibir a mensagem de erro ao usuário "Agora eu estou falando sério. Este erro é fatal!" , como visto na Figura 5.31.

Figura 5.31 – Mensagem de erro.

message_error.php

```
<?php
function __autoload($classe)
{
    if (file_exists("app.widgets/{$classe}.class.php"))
    {
        include_once "app.widgets/{$classe}.class.php";
    }
}

// exibe uma mensagem de erro
new TMessage('error', 'Agora eu estou falando sério. Este erro é fatal!');
?>
```

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, seja para a inserção de novos registros, para definição de preferências do sistema ou para definir parâmetros para filtragem de um relatório, dentre outros. Utilizamos listagens para exibir os dados da aplicação, seja para simples conferência, em relatórios ou ainda possibilitando a edição e exclusão de registros. Neste capítulo criaremos componentes que visam facilitar a implementação de formulários e listagens de forma orientada a objetos.

6.1 Formulários

Um formulário é um conjunto de campos dispostos ao usuário de forma agrupada para que este os preencha com informações requisitadas pela aplicação. Um formulário é composto por campos de diversos tipos como caixas de digitação, radio buttons e combo-boxes, além de possuir botões de ação, os quais definem o programa que processará os dados. Em uma aplicação temos interfaces de entrada de dados, rotinas de processamento e interfaces para apresentação de dados. Podemos dizer que os formulários dominam amplamente as interfaces de entradas de dados em aplicações, portanto é imprescindível dominarmos o seu uso e também simplificarmos ao máximo para ganharmos maior produtividade no desenvolvimento.

Classe	Descrição
`TEntry`	Classe que representará campos de entrada de dados `<input type="text">`.
`TPassword`	Classe que representará campos de senha `<input type="password">`.
`TFile`	Classe que representará um botão de seleção de arquivo `<input type="file">`.
`THidden`	Classe que representará um campo escondido `<input type="hidden">`.
`TCombo`	Classe que representará uma lista de seleção `<select>`.
`TText`	Classe que representará uma área de texto `<textarea>`.
`TCheckButton`	Classe que representará campos de checagem `<input type="checkbox">`.
`TCheckGroup`	Classe que representará um grupo de `TCheckButton`.
`TRadioButton`	Classe que representará botões de rádio `<input type="radio">`.
`TRadioGroup`	Classe que representará um grupo de `TRadioButton`.

Acima de todas estas classes listadas anteriormente, teremos a classe `TField`. Tal classe irá prover a infra-estrutura básica e comum a todo campo de um formulário, ou seja, aquelas operações básicas que todo elemento de um formulário deverá oferecer, como métodos para alterar seu nome, seu valor e seu tamanho. Veja a seguir um exemplo de formulário contendo as classes que serão criadas.

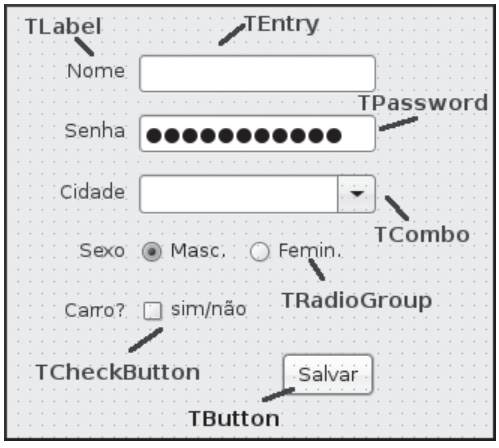

Figura 6.2 – Exemplo de formulário.

A classe responsável por montar o formulário irá encapsular os elementos anteriormente listados. Para isso, criaremos a classe `TForm`, que poderá conter, por meio de uma estrutura de agregação, qualquer elemento derivado da classe `TField`. Veja a seguir o diagrama de classes resumido.

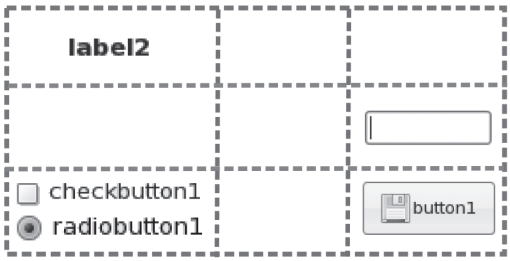

Figura 6.7 – Visual de tabela.

O formulário que construiremos com o programa a seguir é formado por três tabelas. Existe uma grande tabela com duas linhas: a primeira linha contém o título do formulário; a segunda contém duas células. Como o componente `TTable` que criamos aceita outros objetos em suas células, nestas duas células da segunda linha colocamos uma nova tabela dentro de cada uma. Nestas tabelas internas temos, por sua vez, novas linhas e colunas para dispor e organizar seus elementos (rótulos de texto e campos de digitação de dados). Utilizamos esta tática para organizar o visual do formulário em duas colunas, cada uma contendo uma nova tabela. Note que podemos fazer diversas combinações, adicionando tabelas dentro de tabelas ou ainda outros componentes para organizar o layout de nosso formulário. Veja a seguir o resultado final.

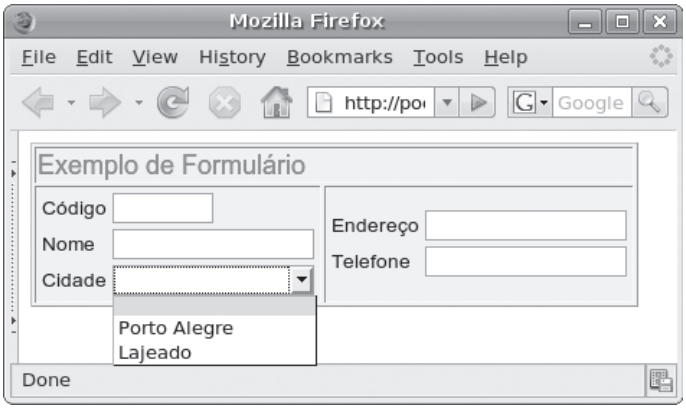

Figura 6.8 – Formulário com tabela.

formA.php

```
<?php
/*
 * função __autoload()
 *  carrega uma classe quando ela é necessária,
 *  ou seja, quando ela é instancia pela primeira vez.
 */
```

Figura 6.19 – Usando a TForm de maneira estruturada e estática.

form1.php

```
<?php
/*
 * função __autoload()
 *  carrega uma classe quando ela é necessária,
 *  ou seja, quando ela é instancia pela primeira vez.
 */
function __autoload($classe)
{
    if (file_exists("app.widgets/{$classe}.class.php"))
    {
        include_once "app.widgets/{$classe}.class.php";
    }
}
// instancia um formulário
$form = new TForm('form_email');
// instancia uma tabela
$table = new TTable;

// adiciona a tabela ao formulário
$form->add($table);

// cria os campos do formulário
$nome     = new TEntry('nome');
$email    = new TEntry('email');
$titulo   = new TEntry('titulo');
$mensagem = new TText('mensagem');
```

6.4 Listagens orientadas a objetos

6.4.1 Introdução

Nos exemplos anteriores, vimos como se dá a visualização de listagens estáticas e também com informações do banco de dados. Construiremos, agora, uma classe que possibilite criar listagens utilizando uma estrutura totalmente orientada a objetos, denominada `TDataGrid`. Esta classe irá conter alguns métodos que possibilitarão ao programador adicionar novas colunas e também ações na listagem. As colunas irão conter os dados da listagem; as ações serão exibidas na frente dos dados e permitirão manipulá-los (deletar, editar, visualizar).

Veja na Figura 6.27 o resultado de uma listagem construída com a classe `TDataGrid`. No lado esquerdo da tabela temos os ícones contendo as ações e, em seguida, as colunas contendo os dados. As ações serão representadas por objetos do tipo `TDataGridAction`, e as colunas serão representadas por objetos do tipo `TDataGridColumn`. O objeto do tipo `TDataGrid` irá conter objetos do tipo `TDataGridColumn` e `TDataGridAction` em uma estrutura de agregação, como veremos a seguir. Poderemos agregar diversas colunas (`TDataGridColumn`) e ações (`TDataGridAction`) na listagem (`TDataGrid`).

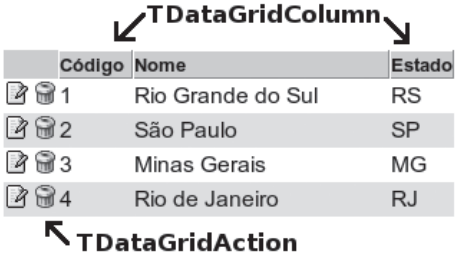

Figura 6.27 – Exemplo de listagem construída com TDataGrid.

6.4.2 Elementos de uma DataGrid

6.4.2.1 TDataGrid

A classe `TDataGrid` será responsável pela exibição de listagens. Uma listagem basicamente é um tipo de tabela. Assim, iremos aproveitar toda estrutura já desenvolvida pela classe `TTable` e iremos utilizá-la como superclasse de `TDataGrid`. Em seu método construtor iremos criar os estilos que serão utilizados para renderizar a tabela e seu cabeçalho. O estilo `tdatagrid_table` será aplicado na tabela para indicar algumas características como a fonte e o espaçamento entre células. O estilo `tdatagrid_col` será utilizado para as colunas do cabeçalho. Este estilo irá definir a espessura, a cor das bordas e do fundo de cada célula do cabeçalho. O estilo `tdatagrid_col_over` será

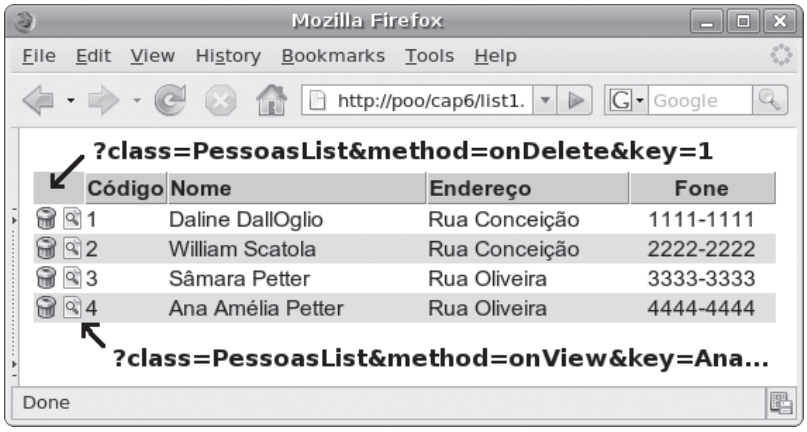

Figura 6.33 – Listagem orientada a objetos e estática.

list2.php

```
<?php
/*
 * função __autoload()
 *  carrega uma classe quando ela é necessária,
 *  ou seja, quando ela é instancia pela primeira vez.
 */
function __autoload($classe)
{
    if (file_exists("app.widgets/{$classe}.class.php"))
    {
        include_once "app.widgets/{$classe}.class.php";
    }
}

class PessoasList extends TPage
{
    private $datagrid;

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

        // instancia objeto DataGrid
        $this->datagrid = new TDataGrid;

        // instancia as colunas da DataGrid
        $codigo   = new TDataGridColumn('codigo',   'Código',   'left',    50);
        $nome     = new TDataGridColumn('nome',     'Nome',     'left',   180);
        $endereco = new TDataGridColumn('endereco', 'Endereço', 'left',   140);
        $telefone = new TDataGridColumn('fone',     'Fone',     'center', 100);
```

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 formulários, listagens, tabelas e ações, dentre outros. Ao longo de cada capítulo procuramos dar exemplos da utilização de cada classe e agora chegou o momento de utilizar este conhecimento para construir algo maior, uma aplicação completa. Para isso, estudaremos formas de organizar e estruturar esta aplicação, como o padrão MVC e Web Services.

7.1 Organização da aplicação

7.1.1 Model View Controller

Model View Controller (MVC) é o design pattern mais conhecido de todos. Seus conceitos remontam à plataforma Smaltalk na década de 1970. Basicamente uma aplicação que segue o pattern Model View Controller é dividida em três camadas. As letras que compõem o nome deste pattern representam cada um desses aspectos.

Model significa modelo. Um modelo é um objeto que representa as informações do domínio de negócios da aplicação. A camada de Modelo pode ser representada por um Domain Model ou um Active Record, dentre outros. View significa Visualização. Nesta camada, teremos a definição da interface com o usuário, como os campos serão organizados e distribuídos na tela. Por exemplo: se temos um cadastro de pessoas, em algum lugar precisamos definir como será este formulário, sua estrutura. Esta

não exista, então poderá ser cadastrado, como acontece na tabela de cidades. Veja na Figura 7.7 nosso Modelo Entidade Relacionamento.

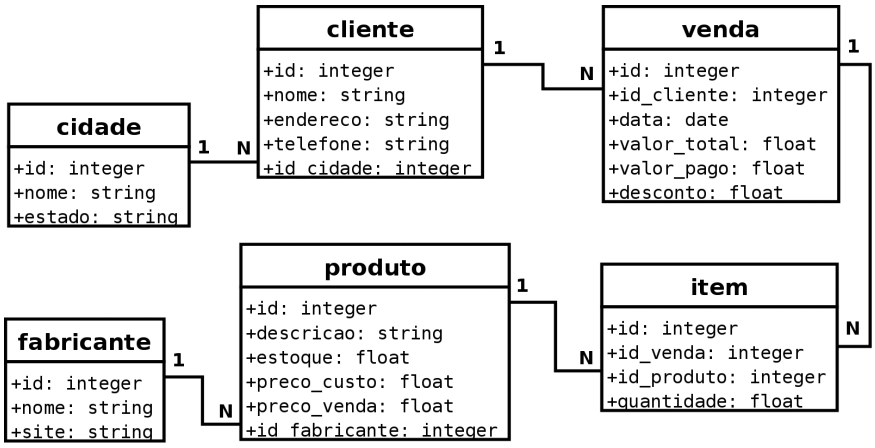

Figura 7.7 – Modelo Entidade Relacionamento.

A venda é um caso à parte. A empresa pode realizar inúmeras vendas, para clientes diferentes e em datas diferentes. Portanto, precisaremos armazenar para qual cliente a venda foi realizada e em qual data. Também poderemos armazenar o valor total da venda e também o valor pago, caso tenha sido concedido algum desconto. Uma venda é composta de produtos. Em uma venda, podem ser vendidas quantidades diferentes de produtos. Portanto faz-se necessária a criação de uma tabela para armazenar quais foram os produtos que participaram de uma determinada venda. Para isso, criaremos a tabela `item`, na qual cada item corresponde a um produto que fez parte de uma venda. Nessa tabela ainda teremos a quantidade vendida de cada item, além, é claro, das chaves estrangeiras para cada tabela.

7.2.2 Cadastro de cidades

O cadastro de cidades será um cadastro básico. Chamamos de básico porque é utilizado de forma a complementar um registro mais importante, que carrega um maior número de informação e que, neste caso, é o cliente. O nosso cadastro de cidades terá somente o nome da cidade e o estado, sendo que o estado será representado por apenas dois caracteres (`UF`) e virá a partir de uma combo-box.

7.2.2.1 Modelo

O nosso registro de cidade será apenas uma extensão da classe `TRecord`, não oferecendo nenhum novo método. Só utilizaremos os métodos básicos para armazenar o registro no banco de dados (`store`), excluir (`delete`) e carregar (`load`). A classe de modelo `CidadeRecord` será armazenada na pasta `app.model`.

instanciar os objetos, adicionamos os mesmos em uma tabela. Para este formulário, criaremos um botão de ação (`save_button`) que, ao ser clicado irá recarregar a página e executar o método `onSave()`, de modo que serão tomadas as medidas para que o registro seja armazenado no banco de dados.

CidadesList.class.php

```
<?php
/*
 * classe CidadesList
 * cadastro de cidades: contém o formuláro e a listagem
 */
class CidadesList extends TPage
{
    private $form;        // formulário de cadastro
    private $datagrid;    // listagem
    /*
     * método construtor
     * Cria a página, o formulário e a listagem
     */
    public function __construct()
    {
        parent::__construct();

        // instancia um formulário
        $this->form = new TForm('form_cidades');

        // instancia uma tabela
        $table = new TTable;

        // adiciona a tabela ao formulário
        $this->form->add($table);

        // cria os campos do formulário
        $codigo     = new TEntry('id');
        $descricao  = new TEntry('nome');
        $estado     = new TCombo('estado');

        // cria um vetor com as opções da combo
        $items= array();
        $items['RS'] = 'Rio Grande do Sul';
        $items['SP'] = 'São Paulo';
        $items['MG'] = 'Minas Gerais';
        $items['PR'] = 'Paraná';
        // adiciona as opções na combo
        $estado->addItems($items);
```

7.2.7.2 Aplicação

Nosso programa de emissão do relatório de vendas será muito simples. Será composto basicamente de um formulário, no qual o usuário irá preencher duas datas: a data inicial e a data final. O programa deverá pesquisar na base de dados todas as vendas realizadas entre esses dois períodos e montar o relatório em tela. Veja na Figura 7.19 o relatório de vendas.

Data Inicial: 01-01-2007
Data Final: 31-12-2007
Gerar Relatório ⟶ RelatorioForm::onGera()

Data	Cliente/Produtos	Qtde	Preço
10/07/2007	4 : Fernando H. Correa		
	1 : Pendrive 512Mb	2	40,00
	2 : HD 120 GB	3	180,00
	Sub-Total	**5**	**620,00**
15/07/2007	8 : Luís Zendrael		
	3 : SD CARD 512MB	1	35,00
	4 : SD CARD 1GB MINI	2	40,00
	Sub-Total	**3**	**115,00**
22/07/2007	6 : Rodrigo Bisterço		
	5 : CAM. FOTO I70 PLATA	1	900,00
	7 : WEBCAM INSTANT VF0040SP	2	80,00
	Sub-Total	**3**	**1.060,00**

Figura 7.19 – Relatório de vendas.

Nossa página terá então um formulário com dois campos do tipo `TEntry` (`data_ini` e `data_fim`). O formulário terá também o botão de ação **Gerar Relatório** que está vinculado à execução do método `onGera()`, o qual tem como função ler o intervalo de datas e montar o relatório em tela.

RelatorioForm.class.php

```
<?php
/*
 * classe RelatorioForm
 * relatório de vendas por período
 */
class RelatorioForm extends TPage
{
    private $form;     // formulário de entrada

    /*
     * método construtor
     * cria a página e o formulário de parâmetros
     */
```

```
    /*
     * método  conv_data_to_br()
     * Converte uma data para o formato dd/mm/yyyy
     * @param $data = data no formato yyyy-mm-dd
     */
    function conv_data_to_br($data)
    {
        // captura as partes da data
        $ano = substr($data,0,4);
        $mes = substr($data,5,2);
        $dia = substr($data,8,4);
        // retorna a data resultante
        return "{$dia}/{$mes}/{$ano}";
    }
}
?>
```

7.3 Web Services

7.3.1 Introdução

Web Services são serviços disponibilizados pela internet por meio de um conjunto de tecnologias independentes de plataforma que permitem interoperabilidade entre aplicações por meio da entrega de serviços e a comunicação entre aplicações por meio de padrões abertos e conhecidos como XML e SOAP. A interoperabilidade provida pelos Web Services torna possível agregar e publicar dinamicamente aplicações. Web Services permitem que tecnologias de diferentes plataformas acionem serviços e se comuniquem umas com as outras, como ilustrado a seguir.

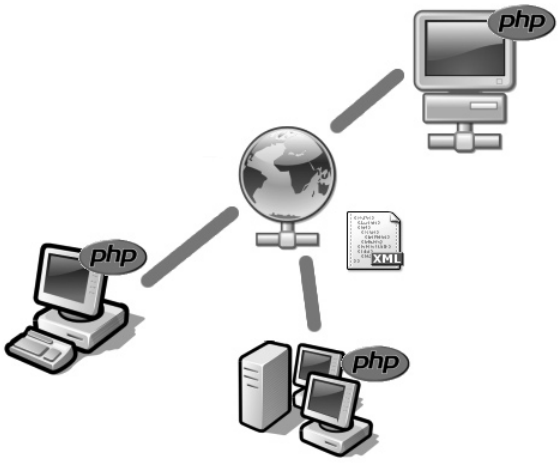

Figura 7.20 – Web Services.

uma interface enxuta, que encapsula uma série de chamadas a métodos e objetos da aplicação servidora, por um meio remoto.

Do lado cliente da aplicação, teremos a classe `NovoCliente`, que na verdade é uma janela GTK (`GtkWindow`). Esta janela disponibiliza alguns campos para preenchimento do usuário. Quando este terminar de preencher esses campos e clicar no botão **Salvar**, o método `onSaveClick()` será executado, instanciando um objeto da classe `SoapClient`, que realiza a conexão com o servidor `SoapServer`, de modo que será executado o método `handle()` que irá despachar a chamada ao método correto no lado do servidor, que neste caso é o método `salvar()`. Veja na Figura 7.24 a estrutura da aplicação.

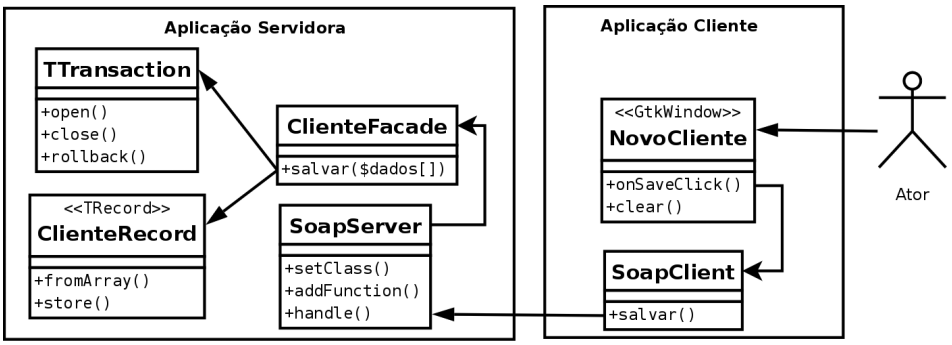

Figura 7.24 – Estrutura da aplicação proposta.

A primeira parte da aplicação será o servidor. Para implementá-lo, criaremos a classe `ClienteFacade`, que estará dentro do arquivo `server.php`. Esta classe oferecerá o método `salvar()`, que recebe um array com dados de cliente. O método `salvar()` instancia um objeto `ClienteRecord`, alimenta-o com os dados vindos do array e armazena-o na base de dados pelo método `store()`.

Observe que não temos as classes necessárias declaradas neste arquivo. Assim, escrevemos a função `__autoload()`, que se encarregará de automaticamente carregar as classes a partir das pastas `app.ado` e `app.model` quando necessário.

As exceções lançadas da forma "`throw`" no código da aplicação servidora não são capturadas pelo bloco de `try/catch` da aplicação cliente do Web Service. Para que isso funcione, precisamos retornar o objeto de exceção por um "`return`" e não lançá-lo. Neste programa, estamos capturando qualquer exceção lançada e retornando-a na forma de um objeto `SoapFault`.

Nesta aplicação, mostraremos uma forma diferente de utilizar Web Services: sem o arquivo WSDL. Dependendo dos parâmetros que passamos no momento de instanciarmos os objetos `SoapServer` e `SoapClient`, podemos identificar que estamos no modo não-wsdl, indicando o primeiro parâmetro como sendo `NULL`. Isso, no entanto, obriga-nos a identificar outros parâmetros como `uri` e `location`. Esta forma de utilizar Web Services é mais dinâmica.