

Inclui Design Patterns

php

Programando com Orientação a Objetos



novatec

Pablo Dall'Oglio

PHP

Programando com Orientação a Objetos

PHP

Programando com Orientação a Objetos

Pablo Dall'Oglio

Copyright © 2007 da Novatec Editora Ltda.

Todos os direitos reservados e protegidos pela Lei 9.610 de 19/02/1998. É proibida a reprodução desta obra, mesmo parcial, por qualquer processo, sem prévia autorização, por escrito, do autor e da Editora.

Editor: Rubens Prates

Editoração eletrônica: Rodolpho Lopes

Revisão gramatical: Gabriela de Andrade Fazioni

Capa: Pablo Dall'Oglio e Rodolpho Lopes

ISBN: 978-85-7522-137-2

NOVATEC EDITORA LTDA.

Rua Luís Antônio dos Santos 110
02460-000 – São Paulo, SP – Brasil

Tel.: +55 11 6959-6529

Fax: +55 11 6950-8869

E-mail: novatec@novatec.com.br

Site: www.novatec.com.br

**Dados Internacionais de Catalogação na Publicação (CIP)
(Câmara Brasileira do Livro, SP, Brasil)**

Dall`Oglio, Pablo
PHP : programando com orientação a objetos /
Pablo Dall`Oglio. -- São Paulo : Novatec Editora,
2007.

ISBN 978-85-7522-137-2

1. PHP (Linguagem de programação para
computadores) I. Título.

07-7482

CDD-005.133

Índices para catálogo sistemático:

1. PHP : Linguagem de programação : Computadores :
Processamento de dados 005.133

*Um dia Martin Luther King disse: “O que me preocupa não é o grito dos maus.
É o silêncio dos bons”.*

Dedico este livro a você que não silencia.

Dedico este livro a você que é contra o preconceito, que é contra a violência.

*Dedico este livro a você que reage perante a injustiça quando poderia
simplesmente se omitir.*

*Dedico a você que ainda faz distinção entre o certo e o errado em uma sociedade em que
a ética e o caráter têm se tornado cada vez mais flexíveis.*

*Dedico este livro a você, que tem valores, princípios e consegue vencer na vida sem
passar por cima de ninguém.*

*Dedico este livro a você, que tem a grandeza de torcer e se emocionar com a vitória
alheia, de querer o bem do próximo.*

*Dedico este livro aos meus, minha família, meu amor, meus amigos.
Vocês são o que importa.*

Sumário

Sobre o autor	13
Agradecimentos.....	14
Nota do autor.....	16
Organização do livro.....	18
Capítulo 1 * Introdução ao PHP	20
1.1 O que é o PHP?	20
1.2 Um programa PHP.....	21
1.2.1 Extensão de arquivos.....	21
1.2.2 Delimitadores de código	22
1.2.3 Comentários	22
1.2.4 Comandos de saída (output)	22
1.3 Variáveis	24
1.3.1 Tipo booleano.....	26
1.3.2 Tipo numérico.....	27
1.3.3 Tipo string.....	27
1.3.4 Tipo array.....	28
1.3.5 Tipo objeto	28
1.3.6 Tipo recurso	28
1.3.7 Tipo misto	29
1.3.8 Tipo callback	29
1.3.9 Tipo NULL.....	29
1.4 Constantes	29
1.5 Operadores	30
1.5.1 Atribuição.....	30
1.5.2 Aritméticos.....	30
1.5.3 Relacionais	31
1.5.4 Lógicos.....	33
1.6 Estruturas de controle	34
1.6.1 IF	34
1.6.2 WHILE	37
1.6.3 FOR	38

1.6.4 SWITCH	39
1.6.5 FOREACH	42
1.6.6 CONTINUE.....	42
1.6.7 BREAK.....	42
1.7 Requisição de arquivos	43
1.8 Manipulação de funções	44
1.8.1 Criação.....	44
1.8.2 Variáveis globais	45
1.8.3 Variáveis estáticas	45
1.8.4 Passagem de parâmetros.....	46
1.8.5 Recursão	48
1.9 Manipulação de arquivos e diretórios	48
1.10 Manipulação de strings.....	58
1.10.1 Declaração.....	58
1.10.2 Concatenação	59
1.10.3 Caracteres de escape	59
1.10.4 Funções.....	60
1.11 Manipulação de arrays	64
1.11.1 Criando um array.....	64
1.11.2 Arrays associativos.....	65
1.11.3 Iterações	66
1.11.4 Acesso	66
1.11.5 Arrays multidimensionais	67
1.11.6 Funções	68
1.12 Manipulação de objetos	79
Capítulo 2 • Orientação a objetos.....	86
2.1 Introdução	86
2.1.1 Programação estruturada.....	86
2.1.2 Orientação a objetos	87
2.2 Classe.....	90
2.3 Objeto	93
2.4 Construtores e destrutores	95
2.5 Herança.....	98
2.6 Polimorfismo	101
2.7 Abstração	103
2.7.1 Classes abstratas	103
2.7.2 Classes finais	104
2.7.3 Métodos abstratos.....	105
2.7.4 Métodos finais	106

2.8 Encapsulamento	107
2.8.1 Private	109
2.8.2 Protected	111
2.8.3 Public	113
2.9 Membros da classe	114
2.9.1 Constantes	114
2.9.2 Propriedades estáticas	115
2.9.3 Métodos estáticos	116
2.10 Associação, agregação e composição	117
2.10.1 Associação	117
2.10.2 Agregação	118
2.10.3 Composição	122
2.11 Intercepções	124
2.11.1 Método __set()	124
2.11.2 Método __get()	126
2.11.3 Método __call()	127
2.11.4 Método __toString()	128
2.11.5 Método toXml()?	129
2.12 Interfaces	132
2.13 Método __clone()	133
2.14 Autoload	134
2.15 Objetos dinâmicos	135
2.16 Manipulação de XML	137
2.16.1 Exemplos	137
2.17 Tratamento de erros	145
2.17.1 A função die()	145
2.17.2 Retorno de flags	146
2.17.3 Lançando erros	148
2.17.4 Tratamento de exceções	150
Capítulo 3 :: Manipulação de dados	154
3.1 Acesso nativo	154
3.1.1 Introdução	154
3.1.2 Exemplos	155
3.2 PDO :: PHP Data Objects	159
3.2.1 Introdução	159
3.2.2 Exemplos	160
3.3 Uma API orientada a objetos	164
3.3.1 Introdução	164
3.3.2 Sintaxe SQL	165

3.3.3 Usando SQL no PHP.....	167
3.3.4 Design pattern	169
3.3.5 Query Object	170
3.3.6 Critérios de seleção.....	172
3.3.7 Instruções SQL.....	183
3.3.8 Insert	185
3.3.9 Update	189
3.3.10 Delete	192
3.3.11 Select.....	194
3.3.12 Conexão com banco de dados.....	199
3.3.13 Controle de transações.....	206
3.3.14 Registro de log.....	212
Capítulo 4 • Mapeamento Objeto-Relacional.....	221
4.1 Persistência	221
4.1.1 Introdução	221
4.2 Mapeamento objeto-relacional	222
4.2.1 Identity Field	223
4.2.2 Foreign Key Mapping	224
4.2.3 Association Table Mapping	226
4.2.4 Single Table Inheritance	228
4.2.5 Concrete Table Inheritance	230
4.2.6 Class Table Inheritance.....	230
4.2.7 Lazy Initialization.....	231
4.3 Modelo de negócios	235
4.3.1 Domain Model Pattern.....	236
4.3.2 Table Module	242
4.4 Gateways	245
4.4.1 Table Data Gateway	246
4.4.2 Row Data Gateway	253
4.4.3 Active Record.....	257
4.4.4 Data Mapper	261
4.5 Manipulando objetos.....	265
4.5.1 Introdução	265
4.5.2 Exemplos	274
4.5.3 Novo objeto	276
4.5.4 Obter objeto	278
4.5.5 Alterar objeto	280
4.5.6 Clonar objeto	283
4.5.7 Excluir objeto.....	285

4.6 Manipulando coleções	286
4.6.1 Repository	287
4.6.2 Obter coleção de objetos.....	292
4.6.3 Alterar coleção de objetos	295
4.6.4 Contar objetos.....	297
4.6.5 Excluir coleção de objetos.....	299
4.7 Aspectos avançados.....	301
4.7.1 Encapsulamento.....	301
4.7.2 Lazy Initialization	305
4.7.3 Criar métodos de negócio.....	309
Capítulo 5 * Apresentação e controle	312
5.1 Introdução	312
5.2 Componentes	313
5.2.1 Elementos HTML	314
5.2.2 Folhas de estilo.....	320
5.2.3 Imagens	325
5.2.4 Textos.....	328
5.3 Contêineres.....	331
5.3.1 Tabelas.....	331
5.3.2 Painéis	340
5.3.3 Janelas.....	344
5.4 Diálogos e controles	353
5.4.1 Page Controller	353
5.4.2 Ações	361
5.4.3 Diálogos de mensagem.....	369
5.4.4 Diálogos de questionamento.....	373
Capítulo 6 * Formulários e listagens	377
6.1 Formulários	377
6.1.1 Elementos de um formulário.....	378
6.1.2 Exemplo de formulário	380
6.1.3 Método POST	382
6.2 Um formulário orientado a objetos.....	384
6.2.1 Introdução.....	384
6.2.2 Elementos de um formulário	386
6.2.3 Disposição e layout	398
6.2.4 Outros componentes.....	404
6.2.5 Exemplos	421
6.3 Listagens	441
6.3.1 Exemplos de listagens.....	441

6.4 Listagens orientadas a objetos	445
6.4.1 Introdução.....	445
6.4.2 Elementos de uma DataGrid.....	445
6.4.3 Exemplos	457
Capítulo 7 * Criando uma aplicação.....	477
7.1 Organização da aplicação.....	477
7.1.1 Model View Controller.....	477
7.1.2 Pacotes	478
7.1.3 Internacionalização e Singleton Pattern	480
7.1.4 Seções e Registry Pattern.....	485
7.1.5 Front Controller.....	489
7.1.6 Template View	493
7.2 Uma aplicação-exemplo	497
7.2.1 Estrutura	497
7.2.2 Cadastro de cidades	498
7.2.3 Cadastro de fabricantes	506
7.2.4 Cadastro de clientes	511
7.2.5 Cadastro de produtos.....	523
7.2.6 Processo de venda	534
7.2.7 Emissão de relatórios.....	548
7.3 Web Services	555
7.3.1 Introdução	555
7.3.2 Arquitetura.....	556
7.3.3 Funcionamento.....	557
7.3.4 Remote Facade	561
7.4 Conclusão.....	570
Índice remissivo	571

Sobre o autor

PABLO DALL'OGLIO é graduado em Análise de Sistemas pela Unisinos e autor de softwares reconhecidos como o Agata Report e o Tulip. Possui grande experiência no desenvolvimento de sistemas e está constantemente envolvido com análise, projeto e implementação de softwares orientados a objetos, UML e design patterns. Criador da comunidade brasileira de PHP-GTK (www.php-gtk.com.br), é diretor de tecnologia da Adianti Solutions (www.adianti.com.br).

Agradecimentos

Quero agradecer ao Rubens Prates, o editor mais bacana do mundo, por acreditar nas minhas idéias malucas e publicar meus livros. Quero agradecer aos meus avós por terem acreditado em mim e por terem patrocinado meus primeiros cursos de informática na década de 1990, quando informática era tida como coisa para gênio, – hoje é coisa para louco. Agradeço aos meus pais por todo o empenho em me proporcionar educação superior – foram anos difíceis. Obrigado por terem lutado para que eu tivesse a oportunidade de estudar, obrigado pelos valores. Ter nascido humilde me faz valorizar muito cada pequena conquista da minha vida.

Trabalhei neste livro de dezembro de 2006 até hoje, 15 de agosto de 2007. Foi um verão escaldante e um inverno gelado aqui no Rio Grande do Sul. Este livro enfrentou temperaturas que variaram entre 0 e 40 graus durante todo este tempo. Ele literalmente foi escrito com muita transpiração.

Quero agradecer a minha avó pelas guloseimas que me abasteceram nos últimos meses de inverno, principalmente o chocolate caseiro, e ao meu pai por ter comprado um aquecedor que me impediu de congelar entre os capítulos cinco e seis. Para não ficar desinformado enquanto eu escrevia, meu avô me contava os resultados do nosso colorado durante o campeonato brasileiro. Também quero agradecer ao Abel Braga e ao Fernando Carvalho pelo título mundial sobre o Barcelona – foi uma verdadeira epopéia.

Quero deixar um abraço muito especial para minha irmã Daline, que sempre conseguiu o que eu precisava a preços módicos na loja de informática onde trabalha – e nesse tempo foi um suporte para o notebook e um teclado novo. Já meu cunhado William começou a programar em PHP-GTK e está se saindo um bom programador, pena que no xadrez ele não aprende; bom para mim, que levo a melhor! Um abraço aos grandes amigos Júlia e Fábio, parceiro de violão, pelo blues da quarta-feira – não tocamos nada, mas tudo sempre acaba em pizza! Como músicos somos bons programadores. Ao amigo Ranzi, que não perde a oportunidade de dizer a todos que tem um amigo escritor, hehehe. Tenho orgulho de dizer que já tomei chimarrão contigo, cara!

Ao Martim Fowler e ao Scott Ambler, pelos excelentes livros, pela didática e pelas idéias brilhantes. Aos meus clientes e parceiros de negócio, em especial o Carvalhaes, o Ceretta, o Daniel Alvão, o pessoal da DBSeller e da Univates. Aos mestres Mouriac Diemer, Sílvio Cazella, Leandro Pompermeyer, Sérgio Crespo, Cândido Fonseca e Carlo Bellini. Neste livro, há uma parcela de conhecimento de cada um de vocês. Aos artistas que fizeram parte da trilha sonora deste livro, que teve The Beatles, The Eagles, Emmerson Nogueira, Danni Carlos, Marisa Monte, Roupa Nova, Creedence, John Lennon, Cranberries, Mamoras, Beethoven e Vivaldi. Por último, mas não menos importante, à Fernanda, minha eterna namorada. Digo eterna por que já venho dizendo que vou casar com ela desde o primeiro livro que escrevi, em 2004. Acho que antes do próximo eu não escapo. Só você para compreender minha rotina de trabalho maluca. Te amo muito!

Nota do autor

Minha história com PHP inicia-se no ano de 2000 quando fiz parte de um grande projeto de sistema para gestão acadêmica, chamado SAGU. Na época éramos pioneiros no uso de PHP para o desenvolvimento de aplicações de negócio e isso nos rendeu inclusive uma visita do Rasmus, criador da linguagem, durante uma de suas passagens pelo Brasil. Desde aquela época, trabalhei em inúmeros projetos e passei a utilizar também o PHP-GTK para criar aplicações gráficas em PHP, culminando com o desenvolvimento do Agata Report em 2001 e o primeiro livro sobre PHP-GTK do mundo, publicado em 2004. Ao programar usando a extensão GTK, você precisa dominar a orientação a objetos, uma vez que seus componentes (janelas, botões, listas) são representados por classes. Foi uma transição para mim, uma vez que passei anos trabalhando com linguagens procedurais como o velho Clipper, minha primeira linguagem de trabalho, em 1994. Utilizando o GTK, os conceitos de orientação a objetos se solidificavam em mim que estava vislumbrado com a produtividade alcançada pela aplicação de conceitos como herança, encapsulamento e polimorfismo.

No início de 2004 fiquei muito impressionado ao conhecer as novas características do PHP5 em relação à orientação a objetos. Passei a estudar os novos conceitos implementados e tudo coincidiu com a época na qual estava estudando design patterns na disciplina de engenharia de software na universidade. Na época escrevi um artigo intitulado PHP5, Orientação a Objetos e design patterns, uma pequena contribuição minha para a comunidade de PHP que desejava se inteirar sobre o assunto, tendo em vista que a maioria das referências na época estavam em inglês. O artigo foi um sucesso e logo nas primeiras semanas alguns milhares de pessoas já tinham realizado o seu download.

Após anos trabalhando com PHP, fui desenvolvendo um conjunto de classes que implementavam funcionalidades básicas para qualquer projeto novo que eu iniciava, como conexão com bancos de dados, formulários e listagens. Em junho de 2006, resolvi reunir essas classes de forma consistente, sob a estrutura de um framework, para utilizar em meu trabalho de conclusão. Foi justamente quando eu estava desenvolvendo um projeto para uma universidade local, a Univates, cuja equipe gostou do meu framework e solicitou um treinamento para utilizá-lo em seus projetos. Ao mesmo tempo, tinha sido convidado para desenvolver a arquitetura de um sistema para

gerenciar as pesquisas em saúde realizadas no Brasil, pelo Observatório de Ciência e Tecnologia do Ministério da Saúde. Como a equipe do projeto era jovem e o tempo era escasso, resolvi pela utilização do framework para acelerar o desenvolvimento. Durante esses dois projetos, percebi que a utilização do framework tornava simples a tarefa de ensinar os conceitos básicos da orientação a objetos. Foi quando decidi escrever este livro.

Em dezembro de 2006 participei da primeira PHP Conference Brasil e lá pude conhecer de perto a comunidade PHP no Brasil e tomar ciência do crescente interesse em assuntos como a orientação a objetos e design patterns. Tudo que aconteceu colaborou, de certa forma, para que eu decidisse escrever este livro, que é minha parcela de contribuição para quem está começando agora.

Espero que você agregue conhecimento e espero que este livro desperte-lhe idéias. Se isto acontecer, valerá muito a pena ter escrito. Não deixe de me escrever sob quaisquer circunstâncias. Adoraria receber e-mails com sugestões. Como toda primeira edição de um livro, esta dificilmente sairá sem erros. Espero contar com você para evoluir as idéias e juntos construirmos novas edições ou mesmo novos livros.

Pablo Dall’Oglio
<pablo@dalloglio.net>

Organização do livro

Confira o que veremos em cada capítulo deste livro:

O Capítulo 1 consiste de uma introdução à linguagem PHP. Nele veremos os tipos de dados suportados, operadores lógicos e aritméticos, estruturas de controle, manipulação de funções, manipulação de arquivos, de strings e de arrays.

O Capítulo 2 aborda exclusivamente os princípios básicos da orientação a objetos por meio de vários exemplos práticos. Nele abordamos os mais diversos aspectos da orientação a objetos, como abstração, herança, polimorfismo, encapsulamento, agregação, composição, interfaces, métodos construtores, dentre outros. Ademais, abordaremos assuntos como tratamento de exceções e manipulação de arquivos XML.

O Capítulo 3 começa a abordar o acesso a bases de dados. Vemos primeiramente como se dá o acesso da forma tradicional, para então estudarmos a biblioteca PDO e propormos um conjunto de classes que possibilite manipular instruções SQL de forma orientada a objetos. Também criamos objetos para automatizar conexões, transações e registros de log.

O Capítulo 4 aborda a persistência de objetos em bases de dados, ou seja, o mapeamento objeto-relacional. Neste capítulo, estudamos padrões (design patterns) utilizados para converter o modelo de negócios de uma aplicação, representado por objetos associados em memória por meio de relações como heranças, agregações e composições para registros de uma base de dados relacional. Ao final, escolhemos um padrão para ser utilizado ao longo do livro.

O Capítulo 5 consiste da construção de uma série de classes que visam automatizar tarefas de visualização e controle. Nesse sentido, criaremos componentes para exibição de tabelas, janelas, painéis e também de diálogos de mensagem, além de abordarmos técnicas para o controle do fluxo de execução de uma aplicação.

O Capítulo 6 é voltado para a construção de formulários e listagens. Nele, partiremos para o desenvolvimento de um conjunto de classes para a construção de formulários e listagens por meio de diversos objetos que representam cada um dos elementos que os integram, como colunas, ações, combo-boxes, caixas de digitação e

radio buttons. Dessa forma, poderemos trabalhar com entrada, visualização, edição e exclusão de registros de forma orientada a objetos.

O Capítulo 7 apresenta o desenvolvimento de uma aplicação de negócios voltada à área de comércio. Esta aplicação será desenvolvida inteiramente por meio das classes criadas ao longo do livro, utilizando todos os conceitos vistos ao longo dos capítulos para solucionar problemas reais presentes no dia-a-dia do desenvolvimento de uma aplicação de negócios. Também iremos estudar soluções práticas voltadas a problemas pontuais como a distribuição da aplicação, internacionalização, manipulação de seções e web services.

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

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. Dentre 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.

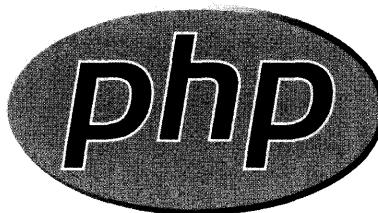


Figura 1.1 – Logo do PHP.

No inverno de 1998, após o lançamento do PHP 3, Zeev e Andi começaram a trabalhar em uma reescrita do núcleo do PHP, tendo em vista melhorar sua performance e modularidade em aplicações complexas. Para tanto, resolveram batizar este núcleo de *Zend Engine*, ou Mecanismo Zend (Zeev + Andi). O PHP 4, baseado neste mecanismo, foi lançado oficialmente em maio de 2000, trazendo muitas melhorias e recursos novos, como seções, suporte a diversos servidores web, além da abstração de sua API, permitindo inclusive ser utilizado como linguagem para shell script. Nesse momento, o PHP já estava presente em cerca de 20% dos domínios da internet, além de ser utilizado por milhares de desenvolvedores ao redor do mundo.

Apesar de todos os esforços, o PHP ainda necessitava maior suporte à orientação a objetos, tal qual existe em linguagens como C++ e Java. Tais recursos estão finalmente presentes no PHP 5, após um longo período de desenvolvimento que culminou com sua disponibilização oficial em julho de 2004. Ao longo do livro veremos esses recursos empregados em exemplos práticos.

Fonte: PHP Group

1.2 Um programa PHP

1.2.1 Extensão de arquivos

A forma mais comum de nomear programas em PHP é a seguinte:

Extensão	Significado
.php	Arquivo PHP contendo um programa.
.class.php	Arquivo PHP contendo uma classe.
.inc.php	Arquivo PHP a ser incluído, pode incluir constantes ou configurações.

Entretanto, outras extensões podem ser encontradas principalmente em programas antigos:

Extensão	Significado
.php3	Arquivo PHP contendo um programa PHP versão 3.
.php4	Arquivo PHP contendo um programa PHP versão 4.
.phtml	Arquivo PHP contendo um programa PHP e HTML na mesma página.

1.2.2 Delimitadores de código

O código de um programa escrito em PHP deve estar contido entre os seguintes delimitadores:

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

Observação: os comandos sempre são delimitados por ponto-e-vírgula (;).

1.2.3 Comentários

Para comentar uma única linha:

```
// echo "a";
# echo "a";
```

Para comentar muitas linhas:

```
/* echo "a";
echo "b"; */
```

1.2.4 Comandos de saída (output)

Estes são os comandos utilizados para gerar uma saída em tela (output). Se o programa PHP for executado na linha de comando (prompt do sistema), a saída será no próprio console. No entanto, se o programa for executado via servidor de páginas web (Apache ou IIS), a saída será exibida na própria página HTML gerada.

echo

É um comando que imprime uma ou mais variáveis no console. Exemplo:

```
echo 'a', 'b', 'c';
```

 **Resultado:**

```
abc
```

print

É uma função que imprime uma string no console. Exemplo:

```
print('abc');
```

 **Resultado:**

```
abc
```

var_dump

Imprime o conteúdo de uma variável de forma explanativa, muito comum para se realizar debug. Se o parâmetro for um objeto, ele imprimirá todos os seus atributos; se for um array de várias dimensões, imprimirá todas elas, com seus respectivos conteúdos e tipos de dados. Exemplo:

```
$vetor = array('Palio', 'Gol', 'Fiesta', 'Corsa');  
var_dump($vetor);
```

 **Resultado:**

```
array(4) {  
    [0]=>  
    string(5) "Palio"  
    [1]=>  
    string(3) "Gol"  
    [2]=>  
    string(6) "Fiesta"  
    [3]=>  
    string(5) "Corsa"  
}
```

print_r

Imprime o conteúdo de uma variável de forma explanativa, assim como a `var_dump()`, mas em um formato mais legível para o programador, com os conteúdos alinhados e suprimindo os tipos de dados. Exemplo:

```
$vetor = array('Palio', 'Gol', 'Fiesta', 'Corsa');  
print_r($vetor);
```

 **Resultado:**

```
Array
(
    [0] => Palio
    [1] => Gol
    [2] => Fiesta
    [3] => Corsa
)
```

1.3 Variáveis

Variáveis são identificadores utilizados para representar valores mutáveis e voláteis, que só existem durante a execução do programa. Elas são armazenadas na memória RAM e seu conteúdo é destruído após a execução do programa. Para criar uma variável em PHP, precisamos atribuir-lhe um nome de identificação, sempre precedido pelo caractere cifrão (\$). Veja os exemplos a seguir:

```
<?php
$nome = "João";
$sobrenome = "da Silva";
echo "$sobrenome, $nome";
?>
```

 **Resultado:**

da Silva, João

Algumas dicas:

- Nunca inicie a nomenclatura de variáveis com números.
- Nunca utilize espaços em branco no meio do identificador da variável.
- Nunca utilize caracteres especiais (! @ # % ^ & * / | [] { }) na nomenclatura das variáveis.
- Evite criar variáveis com mais de 15 caracteres em virtude da clareza do código-fonte.
- Nomes de variáveis devem ser significativos e transmitir a idéia de seu conteúdo dentro do contexto no qual a variável está inserida.
- Utilize preferencialmente palavras em minúsculo (separadas pelo caractere “_”) ou somente as primeiras letras em maiúsculo quando da ocorrência de mais palavras.

```
<?php  
$codigo_cliente // exemplo de variável  
$codigoCliente // exemplo de variável  
?>
```

O PHP é *case sensitive*, ou seja, é sensível a letras maiúsculas e minúsculas. Tome cuidado ao declarar variáveis e nomes de função. Por exemplo, a variável `$codigo` é tratada de forma totalmente diferente da variável `$Codigo`.

Em alguns casos, precisamos ter em nosso código-fonte nomes de variáveis que podem mudar de acordo com determinada situação. Neste caso, não só o conteúdo da variável é mutável, mas também seu nome. Para isso, o PHP implementa o conceito de variáveis variantes (*variable variables*). Sempre que utilizarmos dois sinais de cifrão (\$) precedendo o nome de uma variável, o PHP irá referenciar uma variável representada pelo conteúdo da primeira. Neste exemplo, utilizamos esse recurso quando declaramos a variável `$nome` (conteúdo de `$variavel`) contendo ‘maria’.

```
<?php  
// define o nome da variável  
$variavel = 'nome';  
  
// cria variável identificada pelo conteúdo de $variavel  
$$variavel = 'maria';  
  
// exibe variável $nome na tela  
echo $nome; // resultado = maria  
?>
```

Quando uma variável é atribuída a outra, sempre é criada uma nova área de armazenamento na memória. Veja neste exemplo que, apesar de `$b` receber o mesmo conteúdo de `$a`, após qualquer modificação em `$b`, `$a` continua com o mesmo valor.

```
<?php  
$a = 5;  
$b = $a;  
$b = 10;  
echo $a; // resultado = 5  
echo $b; // resultado = 10  
?>
```

Para criar referência entre variáveis, ou seja, duas variáveis apontando para a mesma região da memória, a atribuição deve ser precedida pelo operador `&`. Assim, qualquer alteração em qualquer uma das variáveis reflete na outra.

```
<?php  
$a = 5;  
$b = &$a;
```

```
$b = 10;
echo $a; // resultado = 10
echo $b; // resultado = 10
?>
```

1.3.1 Tipo booleano

Um booleano expressa um valor lógico que pode ser verdadeiro ou falso. Para especificar um valor booleano, utilize as palavras-chave `TRUE` ou `FALSE`. No exemplo a seguir, declaramos a variável booleana `$exibir_nome`, cujo conteúdo é `TRUE` (verdadeiro). Em seguida, testaremos o conteúdo desta variável para verificar se ela é realmente verdadeira. Caso positivo, será exibido na tela o nome “José da Silva”.

```
<?php
// declara variável com valor TRUE
$exibir_nome = TRUE;

// testa se $exibir_nome é TRUE
if ($exibir_nome)
{
    echo 'José da Silva';
}
?>
```

Resultado:

José da Silva

No programa que segue, criamos uma variável numérica contendo o valor 91. Em seguida, testamos se a variável é maior que 90. Tal comparação também retorna um valor booleano (`TRUE` ou `FALSE`). O conteúdo da variável `$vai_chover` é um boolean que será testado logo em seguida para a impressão da string “Está chovendo”.

```
<?php
// declara variável numérica
$umidade = 91;

// testa se é maior que 90. Retorna um boolean
$vai_chover = ($umidade > 90);

// testa se $vai_chover é verdadeiro
if ($vai_chover)
{
    echo 'Está chovendo';
}
?>
```

 **Resultado:**

Está chovendo

Também são considerados valores falsos em comparações booleanas:

- Inteiro 0
- Ponto flutuante 0.0
- Uma string vazia “” ou “0”
- Um array vazio
- Um objeto sem elementos
- Tipo NULL

1.3.2 Tipo numérico

Números podem ser especificados em notação decimal (base 10), hexadecimal (base 16) ou octal (base 8), opcionalmente precedido de sinal (- ou +).

```
<?php
// número decimal
$a = 1234;
// um número negativo
$a = -123;
// número octal (equivalente a 83 em decimal)
$a = 0123;
// número hexadecimal (equivalente a 26 em decimal)
$a = 0x1A;
// ponto flutuante
$a = 1.234;
// notação científica
$a = 4e23;
?>
```

1.3.3 Tipo string

Uma string é uma cadeia de caracteres alfanuméricos. Para declará-la podemos utilizar aspas simples '' ou aspas duplas ". Veja, com mais detalhes, como manipular strings na seção Manipulação de strings.

```
<?php
$variavel = 'Isto é um teste';
$variavel = "Isto é um teste";
?>
```

1.3.4 Tipo array

Array é uma lista de valores armazenados na memória, os quais podem ser de tipos diferentes (números, strings, objetos) e podem ser acessados a qualquer momento, pois cada valor é relacionado a uma chave. Um array também pode crescer dinamicamente com a adição de novos itens. Veja na seção Manipulação de arrays como manipular arrays.

```
<?php
$carros = array('Palio', 'Corsa', 'Gol');
echo $carros[1];      // resultado = 'Corsa'
?>
```

1.3.5 Tipo objeto

Um objeto é uma entidade com um determinado comportamento definido por seus métodos (ações) e propriedades (dados). Para criar um objeto deve-se utilizar o operador `new`. Veja o exemplo de um objeto do tipo `Computador` e aprenda, no Capítulo 2, como manipular classes e objetos.

```
<?php
class Computador
{
    var $cpu;
    function ligar()
    {
        echo "Ligando computador a {$this->cpu}...";
    }
}

$obj = new Computador;
$obj->cpu = "500Mhz";
$obj->ligar();
?>
```

Resultado:

Ligando computador a 500Mhz...

1.3.6 Tipo recurso

Recurso (`resource`) é uma variável especial que mantém uma referência de recurso externo. Recursos são criados e utilizados por funções especiais, como uma conexão ao banco de dados. Um exemplo é a função `mysql_connect()`, que, ao conectar-se ao banco de dados, retorna uma variável de referência do tipo recurso.

```
resource mysql_connect(...)
```

1.3.7 Tipo misto

O tipo misto (`mixed`) representa múltiplos (não necessariamente todos) tipos de dados em um mesmo parâmetro. Um parâmetro do tipo `mixed` indica que a função aceita diversos tipos de dados como parâmetro. Um exemplo é a função `gettype()`, a qual obtém o tipo da variável passada como parâmetro (que pode ser `integer`, `string`, `array`, `objeto`, entre outros).

```
string gettype (mixed var)
```

Veja alguns resultados possíveis:

```
"boolean"  
"integer"  
"double"  
"string"  
"array"  
"object"  
"resource"  
"NULL"
```

1.3.8 Tipo callback

Algumas funções como `call_user_func()` aceitam um parâmetro que significa uma função a ser executada. Este tipo de dado é chamado de `callback`. Um parâmetro `callback` pode ser o nome de uma função representada por uma `string` ou o método de um objeto a ser executado, representados por um `array`. Veja os exemplos na documentação da função `call_user_func()`.

1.3.9 Tipo NULL

A utilização do valor especial `NULL` significa que a variável não tem valor. `NULL` é o único valor possível do tipo `NULL`.

1.4 Constantes

Uma constante é um valor que não sofre modificações durante a execução do programa. Ela é representada por um identificador, assim como as variáveis, com a exceção de que só pode conter valores escalares (`boolean`, `inteiro`, `ponto flutuante` e `string`). Um valor escalar não pode ser composto de outros valores, como vetores ou objetos. As regras de nomenclatura de constantes seguem as mesmas regras das variáveis, com a exceção de que as constantes não são precedidas pelo sinal de cifrão (\$) e geralmente utilizam-se nomes em maiúsculo.

```
MAXIMO_CLIENTES // exemplo de constante
```

Você pode definir uma constante utilizando a função `define()`. Quando uma constante é definida, ela não pode mais ser modificada ou anulada. Exemplo:

```
<?php
define("MAXIMO_CLIENTES", 100);
echo MAXIMO_CLIENTES;
?>
```

Resultado:

100

1.5 Operadores

1.5.1 Atribuição

Um operador de atribuição é utilizado para definir uma variável atribuindo-lhe um valor. O operador básico de atribuição é `=`.

```
<?php
$var += 5;      // Soma 5 em $var;
$var -= 5;      // Subtrai 5 em $var;
$var *=5;       // Multiplica $var por 5;
$var /= 5;      // Divide $var por 5;
?>
```

Operadores	Descrição
<code>++\$a</code>	Pré-incremento. Incrementa <code>\$a</code> em um e, então, retorna <code>\$a</code> .
<code>\$a++</code>	Pós-incremento. Retorna <code>\$a</code> e, então, incrementa <code>\$a</code> em um.
<code>--\$a</code>	Pré-decremento. Decrementa <code>\$a</code> em um e, então, retorna <code>\$a</code> .
<code>\$a--</code>	Pós-decremento. Retorna <code>\$a</code> e, então, decrementa <code>\$a</code> em um.

1.5.2 Aritméticos

Operadores aritméticos são utilizados para realização de cálculos matemáticos.

Operadores	Descrição
<code>+</code>	Adição.
<code>-</code>	Subtração.
<code>*</code>	Multiplicação.
<code>/</code>	Divisão.
<code>%</code>	Módulo (resto da divisão).

Em cálculos complexos, procure utilizar parênteses, sempre observando as prioridades aritméticas. Por exemplo:

```
<?php
$a = 2;
$b = 4;
echo $a+3*4+5*$b;           // resultado = 34
echo ($a+3)*4+(5*$b);       // resultado = 40
?>
```

O PHP realiza automaticamente a conversão de tipos em operações:

```
<?php
// declaração de uma string contendo 10
$a = '10';
// soma + 5
echo $a + 5;
?>
```

Resultado:

15

1.5.3 Relacionais

Operadores relacionais são utilizados para realizar comparações entre valores ou expressões, resultando sempre um valor boolean (TRUE ou FALSE).

Comparadores	Descrição
==	Igual. Resulta verdadeiro (TRUE) se expressões forem iguais.
==	Idêntico. Resulta verdadeiro (TRUE) se as expressões forem iguais e do mesmo tipo de dados.
!= ou <>	Diferente. Resulta verdadeiro se as variáveis forem diferentes.
<	Menor.
>	Maior que.
<=	Menor ou igual.
>=	Maior ou igual.

Veja a seguir alguns testes lógicos e seus respectivos resultados. No primeiro caso, vemos a utilização errada do operador de atribuição “=” para realizar uma comparação: o operador sempre retorna o valor atribuído.

```
<?php
if ($a = 5)
{
    echo 'essa operação atribui 5 à variável $a e retorna verdadeiro';
}
?>
```

 **Resultado:**

essa operação atribui 5 à variável \$a e retorna verdadeiro

No exemplo que segue, declaramos duas variáveis, uma integer e outra string. Neste caso, vemos a utilização dos operadores de comparação == e !=.

```
<?php
$c = 1234;
$d = '1234';

if ($c == $d)
{
    echo '$c e $d são iguais';
}
else if ($c != $d)
{
    echo '$c e $d são diferentes';
}
?>
```

 **Resultado:**

\$c e \$d são iguais

No próximo caso, além da comparação entre as variáveis, comparamos também os tipos de dados das variáveis.

```
<?php
$c = 1234;
$d = '1234';

if ($c === $d)
{
    echo '$c e $d são iguais e do mesmo tipo';
}
if ($c !== $d)
{
    echo '$c e $d são de tipos diferentes';
}
?>
```

 **Resultado:**

\$c e \$d são de tipos diferentes

O PHP considera o valor zero como sendo falso em comparações lógicas. Para evitar erros semânticos em retorno de valores calculados por funções que podem

retornar tanto um valor booleano quanto um inteiro, podemos utilizar as seguintes comparações:

```
<?
$e = 0;

// zero sempre é igual à FALSE
if ($e == FALSE)
{
    echo '$e é falso';
}

// testa se $e é do tipo FALSE
if ($e === FALSE)
{
    echo '$e é do tipo false';
}

// testa se $e é igual a zero e do mesmo tipo que zero
if ($e === 0)
{
    echo '$e é zero mesmo';
}
?>
```

Resultado:

```
$e é falso
$e é zero mesmo
```

1.5.4 Lógicos

Operadores lógicos são utilizados para combinar expressões lógicas entre si, agrupando testes condicionais.

Operador	Descrição
(\$a and \$b)	E: Verdadeiro (TRUE) se tanto \$a quanto \$b forem verdadeiros.
(\$a or \$b)	OU: Verdadeiro (TRUE) se \$a ou \$b forem verdadeiros.
(\$a xor \$b)	XOR: Verdadeiro (TRUE) se \$a ou \$b forem verdadeiros, de forma exclusiva.
(! \$a)	NOT: Verdadeiro (TRUE) se \$a for FALSE.
(\$a && \$b)	E: Verdadeiro (TRUE) se tanto \$a quanto \$b forem verdadeiros.
(\$a \$b)	OU: Verdadeiro (TRUE) se \$a ou \$b forem verdadeiros.

Observação: or e and têm precedência maior que && ou ||.

No programa a seguir, se as variáveis \$vai_chover e \$esta_frio forem verdadeiras ao mesmo tempo, será impresso “Não vou sair de casa”.

```
<?php  
$vai_chover = TRUE;  
$esta_frio = TRUE;  
  
if ($vai_chover and $esta_frio)  
{  
    echo "Não vou sair de casa";  
}  
?>
```

Resultado:

Não vou sair de casa

Já neste outro programa, caso uma variável seja TRUE e a outra seja FALSE (OU exclusivo), será impresso no console “Vou pensar duas vezes antes de sair”.

```
<?php  
$vai_chover = FALSE;  
$esta_frio = TRUE;  
  
if ($vai_chover xor $esta_frio)  
{  
    echo "Vou pensar duas vezes antes de sair";  
}  
?>
```

Resultado:

Vou pensar duas vezes antes de sair

1.6 Estruturas de controle

1.6.1 IF

O IF é uma estrutura de controle que introduz um desvio condicional, ou seja, um desvio na execução natural do programa. Caso a condição dada pela expressão seja satisfeita, então serão executadas as instruções do bloco de comandos. Caso a condição não seja satisfeita, o bloco de comandos será simplesmente ignorado. O comando IF pode ser lido como “SE (*expressão*) ENTÃO { *comandos...* }”.

ELSE é utilizado para indicar um novo bloco de comandos delimitado por { }, caso a condição do IF não seja satisfeita. Pode ser lido como “caso contrário”. A utilização do ELSE é opcional.

Veja a seguir um fluxograma explicando o funcionamento do comando IF. Caso a avaliação da expressão seja verdadeira, o programa parte para execução de um bloco de comandos; caso seja falsa, parte para a execução do bloco de comandos dada pelo ELSE.

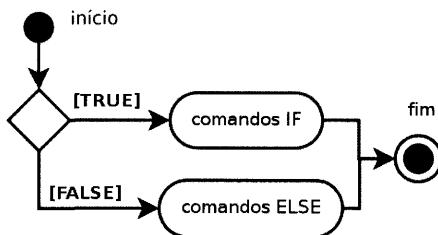


Figura 1.2 – Fluxo do comando IF.

```

if (expressão)
{
    comandos se expressão é verdadeira;
}
else
{
    comandos se expressão é falsa;
}
    
```

Exemplo:

```

<?php
$a = 1;
if ($a==5)
{
    echo "é igual";
}
else
{
    echo "não é igual";
}
?>
    
```

Resultado:

não é igual

Quando não explicitamos o operador lógico em testes por meio do IF, o comportamento-padrão do PHP é retornar TRUE sempre que a variável tiver conteúdo válido.

```
<?php
$a = 'conteúdo';
if ($a)
{
    echo '$a tem conteúdo';
}
if ($b)
{
    echo '$b tem conteúdo';
}
?>
```

Resultado:

\$a tem conteúdo

Para realizar testes encadeados, basta colocar um IF dentro do outro, ou mesmo utilizar o operador AND da seguinte forma:

```
<?php
$salario      = 1020;      // R$
$tempo_servico = 12;       // meses
$tem_reclamacoes = false;   // booleano
if ($salario > 1000)
{
    if ($tempo_servico >= 12)
    {
        if ($tem_reclamacoes != true)
        {
            echo 'parabéns, você foi promovido';
        }
    }
}

if (($salario > 1000) and ($tempo_servico >= 12) and ($tem_reclamacoes != true))
{
    echo 'parabéns, você foi promovido';
}
?>
```

Resultado:

parabéns, você foi promovido
parabéns, você foi promovido

O PHP nos oferece facilidades quando desejamos realizar tarefas simples como realizar uma atribuição condicional a uma variável. A seguir, você confere um código tradicional que verifica o estado de uma variável antes de atribuir o resultado.

```

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`).

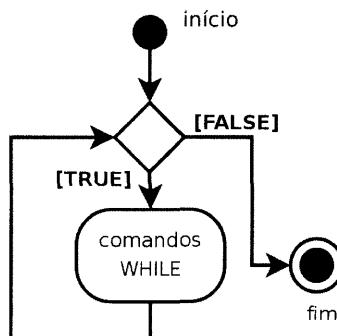


Figura 1.3 – Fluxo do comando `WHILE`.

```
while (expressão)
{
    comandos;
}
```

No exemplo a seguir, o comando `WHILE` está avaliando a expressão “se `$a` é menor que 5” como ponto de entrada do laço de repetições. Na primeira vez que é executada esta comparação, é retornado `TRUE`, visto que o valor de `$a` é 1. Logo o programa entra no laço de repetições executando os comandos entre `{ }`. Observe que, dentro do bloco de comandos, a variável `$a` é incrementada. Assim, esta execução perdurará por mais algumas iterações. Quando seu valor for igual a 5, a comparação retornará `FALSE` e não mais entrará no `WHILE`, deixando de executar o bloco de comandos.

```
<?php
$a = 1;
while ($a < 5)
{
    print $a;
    $a++;
}
?>
```

Resultado:

1234

1.6.3 FOR

O `FOR` é uma estrutura de controle que estabelece um laço de repetição baseado em um contador; é muito similar ao comando `WHILE`. O `FOR` é controlado por um bloco de três comandos que estabelecem uma contagem, ou seja, o bloco de comandos será executado um certo número de vezes.

```
for (expr1; expr2; expr3)
{
    comandos
}
```

Parâmetros	Descrição
<code>expr1</code>	Valor inicial da variável contadora.
<code>expr2</code>	Condição de execução. Enquanto for <code>TRUE</code> , o bloco de comandos será executado.
<code>expr3</code>	Valor a ser incrementado após cada execução.

Exemplo:

```
<?php  
for ($i = 1; $i <= 10; $i++)  
{  
    print $i;  
}  
?>
```

Resultado:

12345678910

Procure utilizar nomes sugestivos para variáveis, mas, em alguns casos específicos, como em contadores, permita-se utilizar variáveis de apenas uma letra, como no exemplo a seguir:

```
<?php  
for ( $i = 0; $i < 5; $i++ )  
{  
    for ( $j = 0; $j < 4; $j++ )  
    {  
        for ( $k = 0; $k < 3; $k++ )  
        {  
            // comandos...  
        }  
    }  
}
```

Evite laços de repetição com muitos níveis de iteração. Como o próprio Linus Torvalds já disse certa vez, se você está utilizando três níveis encadeados ou mais, considere a possibilidade de revisar a lógica de seu programa.

1.6.4 SWITCH

O comando `switch` é uma estrutura que simula uma bateria de testes sobre uma variável. É similar a uma série de comandos `IF` sobre a mesma expressão. Freqüentemente, é necessário comparar a mesma variável com valores diferentes e executar uma ação específica em cada um destes casos.

No fluxograma a seguir vemos que, para cada teste condicional (`CASE`), existe um bloco de comandos a ser executado caso a expressão avaliada retorne verdadeiro (`TRUE`). Caso a expressão retorne falso (`FALSE`), o `switch` parte para a próxima expressão a ser avaliada, até que não tenha mais expressões para avaliar. Caso todas as expressões sejam falsas, o `switch` executará o bloco de códigos representado pelo identificador `default`.

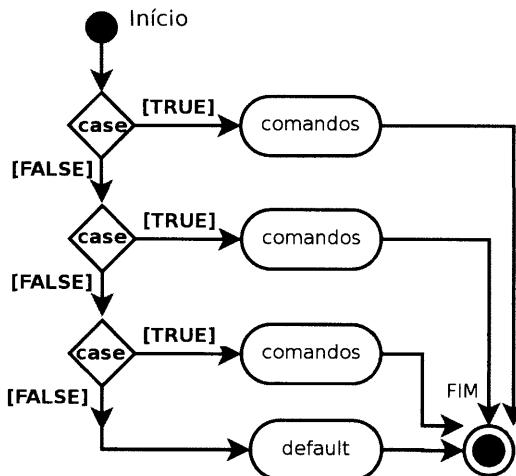


Figura 1.4 – Fluxo do comando SWITCH.

Sintaxe do comando:

```

switch ($expressão)
{
    case "valor 1":
        // comandos
        break;
    case "valor 2":
        // comandos
        break;
    case "valor n":
        // comandos
        break;
    default:
        // comandos
}
  
```

Os exemplos seguintes representam duas formas diferentes de se atingir o mesmo resultado. Primeiro, por meio de uma série de comandos IF e, logo em seguida, utilizando a estrutura switch.

Observação: se você tem um switch dentro de um loop e deseja continuar para a próxima iteração do laço de repetição, utilize o comando `continue 2`, que escapará dois níveis acima.

```

<?php
$i = 1;

if ($i == 0)
  
```

```
{  
    print "i é igual a 0";  
}  
elseif ($i == 1)  
{  
    print "i é igual a 1";  
}  
elseif ($i == 2)  
{  
    print "i é igual a 2";  
}  
else  
{  
    print "i não é igual a 0, 1 ou 2";  
}  
?>
```

 **Resultado:**

i é igual a 1

O `switch` executa linha por linha até encontrar a ocorrência de `break`. Por isso a importância do comando `break` para evitar que os blocos de comando seguintes sejam executados por engano. A cláusula `default` será executada caso nenhuma das expressões anteriores tenha sido verificada.

```
<?php  
$i = 1;  
switch ($i)  
{  
    case 0:  
        print "i é igual a 0";  
        break;  
    case 1:  
        print "i é igual a 1";  
        break;  
    case 2:  
        print "i é igual a 2";  
        break;  
    default:  
        print "i não é igual a 0, 1 ou 2";  
}  
?>
```

 **Resultado:**

i é igual a 1

1.6.5 FOREACH

O `foreach` é um laço de repetição para iterações em arrays ou matrizes. É um `FOR` simplificado que decompõe um vetor ou matriz em cada um de seus elementos por meio de sua cláusula `AS`.

```
foreach (expressão_array as $valor)
{
    instruções
}
```

Exemplo:

```
<?php
$fruta = array("maçã", "laranja", "pêra", "banana");
foreach ($fruta as $valor)
{
    print "$valor -";
}
?>
```

Resultado:

maçã - laranja - pêra - banana -

1.6.6 CONTINUE

A instrução `continue`, quando executada em um bloco de comandos `FOR/WHILE`, ignora as instruções restantes até o fechamento em “`}`”. Dessa forma, o programa segue para a próxima verificação da condição de entrada do laço de repetição.

1.6.7 BREAK

O comando `break` aborta a execução de blocos de comandos, como o `IF`, `WHILE`, `FOR`. Quando estamos em uma execução com muitos níveis de iteração e desejamos abortar n níveis, a sintaxe é a seguinte:

```
While...
For...
    break <quantidade de níveis>
```

1.7 Requisição de arquivos

Em linguagens de script como o PHP, freqüentemente precisamos incluir dentro de nossos programas outros arquivos com definições de funções, constantes, configurações, ou mesmo carregar um arquivo contendo a definição de uma classe. Para atingir este objetivo no PHP, podemos fazer uso de um dos seguintes comandos:

include <arquivo>

A instrução `include()` inclui e avalia o arquivo informado. Seu código (variáveis, objetos e arrays) entra no escopo do programa, tornando-se disponível a partir da linha em que a inclusão ocorre. Se o arquivo não existir, produzirá uma mensagem de advertência (*warning*).

Exemplo:

 **biblioteca.php**

```
<?php
/*
 * função quadrado
 * retorna o quadrado de um número
 */
function quadrado($numero)
{
    return $numero * $numero;
}
?>
```

 **teste.php**

```
<?php
// carrega arquivo com a função necessária
include 'biblioteca.php';

// imprime o quadrado do número 4
echo quadrado(4);
?>
```

 **Resultado:**

16

require <arquivo>

Idêntico ao `include`. Difere somente na manipulação de erros. Enquanto o `include` produz uma *warning*, o `require` produz uma mensagem de *Fatal Error* caso o arquivo não exista.

include_once <arquivo>

Funciona da mesma maneira que o comando `include`, a não ser que o arquivo informado já tenha sido incluído, não refazendo a operação (o arquivo é incluído apenas uma vez). Este comando é útil em casos em que o programa pode passar mais de uma vez pela mesma instrução. Assim, evitará sobreposições, redeclarações etc.

require_once <arquivo>

Funciona da mesma maneira que o comando `require`, a não ser que o arquivo informado já tenha sido incluído, não refazendo a operação (o arquivo é incluído apenas uma vez). Este comando é útil em casos em que o programa pode passar mais de uma vez pela mesma instrução. Assim, poderá-se evitar sobreposições, redeclarações etc.

1.8 Manipulação de funções

Uma função é um pedaço de código com um objetivo específico, encapsulado sob uma estrutura única que recebe um conjunto de parâmetros e retorna um dado. Uma função é declarada uma única vez, mas pode ser utilizada diversas vezes. É uma das estruturas mais básicas para prover reusabilidade.

1.8.1 Criação

Para declarar uma função em PHP, utiliza-se o operador `function` seguido do nome que desejamos lhe atribuir, sem espaços em branco e iniciando obrigatoriamente com uma letra. Na mesma linha, digitamos a lista de argumentos (parâmetros) que a função irá receber, separados por vírgula. Em seguida, encapsulado por chaves {}, vem o código da função. No final, utiliza-se a cláusula `return` para retornar o resultado da função (integer, string, array, objeto etc.).

```
<?php  
  
// exemplo de função  
function nome_da_funcao ($arg1, $arg2, $argN)  
{  
    $valor = $arg1 + $arg2 + $argN;  
    return $valor;  
}  
?>
```

No exemplo a seguir criamos uma função que calcula o índice de obesidade de alguém. A função recebe dois parâmetros (`$peso` e `$altura`) e retorna um valor definido por uma fórmula.

```
<?php
function calcula_obesidade($peso, $altura)
{
    return $peso / ($altura * $altura);
}
echo calcula_obesidade(70, 1.85);
?>
```

 **Resultado:**

20.45288531775

1.8.2 Variáveis globais

Todas as variáveis declaradas dentro do escopo de uma função são locais. Para acessar uma variável externa ao contexto de uma função sem passá-la como parâmetro, é necessário declará-la como `global`. Uma variável global é acessada a partir de qualquer ponto da aplicação. No exemplo a seguir, a função criada converte quilômetros para milhas, enquanto acumula a quantidade de quilômetros percorridos em uma variável global (`$total`).

```
<?php
$total = 0;
function km2mi($quilometros)
{
    global $total;
    $total += $quilometros;
    return $quilometros * 0.6;
}
echo 'percorreu ' . km2mi(100) . " milhas \n";
echo 'percorreu ' . km2mi(200) . " milhas \n";
echo 'percorreu no total ' . $total . " quilometros \n";
?>
```

 **Resultado:**

percorreu 60 milhas
percorreu 120 milhas
percorreu no total 300 quilometros

1.8.3 Variáveis estáticas

Dentro do escopo de uma função podemos armazenar variáveis de forma estática. Assim, elas mantêm o valor que lhes foi atribuído na última execução. Declaramos uma variável estática com o operador `static`.

```
<?php
function percorre($quilometros)
{
    static $total;
    $total += $quilometros;
    echo "percorreu mais $quilometros do total de $total\n";
}
percorre(100);
percorre(200);
percorre(50);
?>
```

 **Resultado:**

```
percorreu mais 100 do total de 100
percorreu mais 200 do total de 300
percorreu mais 50 do total de 350
```

1.8.4 Passagem de parâmetros

Existem dois tipos de passagem de parâmetros: por valor (*by value*) e por referência (*by reference*). Por padrão, os valores são passados *by value* para as funções. Assim, o parâmetro que a função recebe é tratado como variável local dentro do contexto da função, não alterando o seu valor externo. Os objetos, vistos no Capítulo 2, são a exceção.

```
<?php
function Incrementa($variavel, $valor)
{
    $variavel += $valor;
}
$a = 10;
Incrementa($a, 20);
echo $a;
?>
```

 **Resultado:**

10

Para efetuar a passagem de parâmetros *by reference* para as funções, basta utilizar o operador & na frente do parâmetro, fazendo com que as transformações realizadas pela função sobre a variável sejam válidas no contexto externo à função.

```
<?php
function Incrementa(&$variavel, $valor)
{
```

```
$variavel += $valor;  
}  
$a = 10;  
Incrementa($a, 20);  
echo $a;  
?>
```

 **Resultado:**

30

O PHP permite definir valores default para parâmetros. Reescreveremos a função anterior, declarando o default de `$valor` como sendo 40. Assim, se o programador executar a função sem especificar o parâmetro, será assumido o valor 40.

```
<?php  
function Incrementa(&$variavel, $valor = 40)  
{  
    $variavel += $valor;  
}  
$a = 10;  
Incrementa($a);  
echo $a;  
?>
```

 **Resultado:**

50

O PHP também permite definir uma função com o número de argumentos variáveis, ou seja, permite obtê-los de forma dinâmica, mesmo sem saber quais são ou quantos são. Para obter quais são, utilizamos a função `func_get_args()`; para obter a quantidade de argumentos, utilizamos a função `func_num_args()`.

```
<?php  
function Olá()  
{  
    $argumentos = func_get_args();  
    $quantidade = func_num_args();  
  
    for($n=0; $n<$quantidade; $n++)  
    {  
        echo 'Olá ' . $argumentos[$n] . "\n";  
    }  
}  
Olá('João', 'Maria', 'José', 'Pedro');  
?>
```

 **Resultado:**

Olá João
 Olá Maria
 Olá José
 Olá Pedro

1.8.5 Recursão

O PHP permite chamada de funções recursivamente. No caso a seguir criaremos uma função para calcular o fatorial de um número.

```
<?php
function Fatorial($numero)
{
    if ($numero == 1)
        return $numero;
    else
        return $numero * Fatorial($numero -1);
}
echo Fatorial(5) . "\n";
echo Fatorial(7) . "\n";
?>
```

 **Resultado:**

120
 5040

1.9 Manipulação de arquivos e diretórios

A seguir veremos uma série de funções utilizadas exclusivamente para manipulação de arquivos, como abertura, leitura, escrita e fechamento dos mesmos.

fopen

Abre um arquivo e retorna um identificador. Se o nome do arquivo está na forma “protocolo://...”, o PHP irá procurar por um manipulador de protocolo, também conhecido como wrapper, conforme o prefixo.

```
int fopen (string arquivo, string modo [,int usar_path [, resource contexto]])
```

Parâmetros	Descrição
<i>arquivo</i>	String identificando o nome do arquivo a ser aberto.
<i>modo</i>	Forma de abertura do arquivo (r=read, w=write, a=append).
<i>usar_path</i>	Se 1 ou TRUE, vasculha a path pelo arquivo a ser aberto.
<i>contexto</i>	Opções de contexto; variam de acordo com o protocolo do arquivo.

Exemplo:

```
<?php
$fp = fopen ("/home/pablo/file.txt", "r");
$fp = fopen ("/home/pablo/file.gif", "wb");
$fp = fopen ("http://www.example.com/", "r");
$fp = fopen ("ftp://user:password@example.com/", "w");
?>
```

feof

Testa se um determinado identificador de arquivo (criado pela função `fopen()`) está no fim de arquivo (End Of File). Retorna `TRUE` se o ponteiro estiver no fim do arquivo (EOF); do contrário, retorna `FALSE`.

```
intfeof (int identificador)
```

Parâmetro	Descrição
<i>identificador</i>	Parâmetro retornado pela <code>fopen()</code> .

fgets

Lê uma linha de um arquivo. Retorna uma string com até (tamanho – 1) bytes lidos do arquivo apontado pelo identificador de arquivo. Se nenhum tamanho for informado, o default é 1Kb ou 1024 bytes. Se um erro ocorrer, retorna `FALSE`.

```
stringfgetss (int identificador [, int tamanho])
```

Parâmetros	Descrição
<i>identificador</i>	Parâmetro retornado pela <code>fopen()</code> .
<i>tamanho</i>	Quantidade em bytes a retornar da leitura.

Exemplo:

```
<?php
$fd = fopen ("/etc/fstab", "r");
while (!feof ($fd))
{
    // lê uma linha do arquivo
    $buffer = fgets($fd, 4096);

    // imprime a linha.
    echo $buffer;
}
fclose ($fd);
?>
```

Resultado:

```
/dev/hda2      swap        swap      defaults      0  0
/dev/hda3      /           ext3      defaults      1  1
/dev/hda1      /windows    ntfs      defaults      1  0
/dev/cdrom    /mnt/cdrom  iso9660  noauto,owner,ro 0  0
/dev/fd0       /mnt/floppy auto      noauto,owner   0  0
devpts        /dev/pts    devpts    gid=5,mode=620  0  0
proc          /proc       proc      defaults      0  0
```

fwrite

Grava uma string (*conteúdo*) no arquivo apontado pelo *identificador* de arquivo. Se o argumento *comprimento* é dado, a gravação irá parar depois que *comprimento* bytes for escrito ou o fim da string *conteúdo* for alcançado, o que ocorrer primeiro. Retorna o número de bytes gravados, ou FALSE em caso de erro.

```
int fwrite (int identificador, string conteúdo [, int comprimento])
```

Parâmetros	Descrição
<i>identificador</i>	Parâmetro retornado pela fopen().
<i>conteúdo</i>	String a escrever no arquivo.
<i>comprimento</i>	Comprimento da string.

Exemplo:

```
<?php
// abre o arquivo
$fp = fopen ("/home/pablo/file.txt", "w");

// escreve no arquivo
fwrite ($fp, "linha 1\n");
fwrite ($fp, "linha 2\n");
fwrite ($fp, "linha 3\n");

// fecha o arquivo
fclose ($fp);
?>
```

fclose

Fecha o arquivo aberto apontado pelo identificador de arquivo. Retorna TRUE em caso de sucesso ou FALSE em caso de falha.

```
bool fclose (int identificador)
```

Parâmetro	Descrição
<i>identificador</i>	Parâmetro retornado pela fopen().

file_put_contents

Grava uma string em um arquivo. Retorna a quantidade de bytes gravados.

```
int file_put_contents (string nome_arquivo, mixed conteúdo)
```

Parâmetros	Descrição
<i>nome_arquivo</i>	Arquivo a ser aberto.
<i>conteúdo</i>	Novo conteúdo.

Exemplo:

```
<?php
echo file_put_contents('/tmp/teste.txt', "este \n é o conteúdo \n do arquivo");
?>
```

file_get_contents

Lê o conteúdo de um arquivo e retorna o conteúdo em forma de string.

```
string file_get_contents (string nome_arquivo, ...)
```

Exemplo:

```
<?php
echo file_get_contents('/etc/mtab');
?>
```

Resultado:

```
/dev/hda3 / ext3 rw 0 0
/dev/hda1 /windows ntfs rw 0 0
proc /proc proc rw 0 0
usbfs /proc/bus/usb usbfs rw 0 0
```

file

Lê um arquivo e retorna um array com todo o seu conteúdo, de modo que cada posição do array representa uma linha lida do arquivo. O nome do arquivo pode conter o protocolo, como no caso <http://www.servidor.com.br/pagina.html>. Assim, o arquivo remoto será lido para dentro do array.

```
array file (string nome_arquivo, [int usar_include_path])
```

Parâmetros	Descrição
<i>nome_arquivo</i>	Arquivo a ser lido.
<i>usar_include_path</i>	Se “1”, procura também nos diretórios da constante PHP_INCLUDE_PATH.

Exemplo:

```
<?php  
// lê o arquivo para o array $arquivo  
$arquivo = file ("/home/pablo/file.txt");  
// exibe o conteúdo  
echo $arquivo[0];  
echo $arquivo[1];  
echo $arquivo[2];  
?>
```

 **Resultado:**

```
linha 1  
linha 2  
linha 3
```

copy

Copia um arquivo para outro local/nome. Retorna TRUE caso tenha sucedido e FALSE em caso de falhas.

```
bool copy (string arquivo_origem, string arquivo_destino)
```

Parâmetros	Descrição
<i>arquivo_origem</i>	Arquivo a ser copiado.
<i>arquivo_destino</i>	Arquivo destino.

Exemplo:

```
<?php  
$origem = "/home/pablo/file.txt";  
$destino = "/home/pablo/file2.txt";  
if (copy($origem, $destino))  
{  
    echo "Cópia efetuada";  
}  
else  
{  
    echo "Cópia não efetuada";  
}  
?>
```

 **Resultado:**

```
Cópia efetuada
```

rename

Altera a nomenclatura de um arquivo ou diretório.

```
bool rename (string arquivo_origem, string arquivo_destino)
```

Parâmetros	Descrição
<i>arquivo_origem</i>	Arquivo a ser renomeado.
<i>arquivo_destino</i>	Arquivo destino.

Exemplo:

```
<?php
$origem = "/home/pablo/file2.txt";
$destino = "/tmp/file3.txt";
if (rename($origem, $destino))
{
    echo "Renomeação efetuada";
}
else
{
    echo "Renomeação não efetuada";
}
?>
```

**Resultado:**

Renomeação efetuada

unlink

Apaga um arquivo passado como parâmetro. Retorna TRUE em caso de sucesso e FALSE em caso de falhas.

```
bool unlink (string nome_arquivo)
```

Exemplo:

```
<?php
$arquivo = "/tmp/file3.txt";
if (unlink($arquivo))
{
    echo "Arquivo apagado";
}
else
{
    echo "Arquivo não apagado";
}
?>
```

 **Resultado:**

Arquivo apagado

file_exists

Verifica a existência de um arquivo ou de um diretório.

```
bool file_exists (string nome_arquivo)
```

Parâmetros	Descrição
<i>nome_arquivo</i>	Localização de um arquivo ou diretório.

Exemplo:

```
<?php
$arquivo = '/home/pablo/file2.txt';
if (file_exists($arquivo))
{
    echo "Arquivo existente";
}
else
{
    echo "Arquivo não existente";
}
?>
```

 **Resultado:**

Arquivo não existente

is_file

Verifica se a localização dada corresponde ou não a um arquivo.

```
bool is_file (string localização)
```

Parâmetros	Descrição
<i>localização</i>	Localização de um arquivo ou diretório.

Exemplo:

```
<?php
$arquivo = '/home/pablo/file.txt';
if (is_file($arquivo))
{
    echo "$arquivo é um arquivo";
}
else
{
```

```
echo "$arquivo não é um arquivo";
}
?>
```

Resultado:

/home/pablo/file.txt é um arquivo

mkdir

Cria um diretório.

```
bool mkdir (string localização, [int modo])
```

Parâmetros	Descrição
<i>localização</i>	Localização de um diretório.
<i>modo</i>	Permissão de acesso.

Exemplo:

```
<?php
$dir = '/tmp/diretorio';
if (mkdir($dir, 0777))
{
    echo "$dir criado com sucesso";
}
else
{
    echo "$dir não criado";
}
?>
```

Resultado:

/tmp/diretorio criado com sucesso

getcwd

Retorna o diretório corrente.

```
string getcwd ()
```

Exemplo:

```
<?php
echo 'o diretório atual é ' . getcwd();
?>
```

Resultado:

o diretório atual é /tmp

chdir

Altera o diretório corrente. Retorna TRUE em caso de sucesso e FALSE em caso de falhas.

```
bool chdir (string localização)
```

Parâmetros	Descrição
<i>localização</i>	Localização de um diretório.

Exemplo:

```
<?php
echo 'o diretório atual é ' . getcwd();
chdir('/home/pablo');
echo 'o diretório atual é ' . getcwd();
?>
```

 **Resultado:**

```
o diretório atual é /tmp
o diretório atual é /home/pablo
```

rmdir

Apaga um diretório.

```
bool rmdir (string localização)
```

Parâmetros	Descrição
<i>localização</i>	Localização de um diretório.

Exemplo:

```
<?php
$dir = '/tmp/diretorio';
if (rmdir($dir))
{
    echo "$dir apagado com sucesso";
}
else
{
    echo "$dir não apagado";
}
?>
```

 **Resultado:**

```
/tmp/diretorio apagado com sucesso
```

opendir

Abre um diretório e retorna um identificador.

```
resource opendir (string nome_diretorio)
```

Parâmetros	Descrição
<i>nome_arquivo</i>	String identificando o nome do diretório a ser aberto.

closedir

Libera um recurso alocado pela função `opendir()`.

```
void closedir (resource identificador)
```

Parâmetros	Descrição
<i>identificador</i>	Identificador retornado pela função <code>opendir()</code> .

readdir

Realiza a leitura do conteúdo de um diretório por meio do identificador criado pela função `opendir()`.

```
string readdir (resource identificador)
```

Parâmetros	Descrição
<i>identificador</i>	Identificador retornado pela função <code>opendir()</code> .

Exemplo:

```
<?php
// exibe as entradas do diretório raiz
$diretorio = '/';
// verifica se é diretório.
if (is_dir($diretorio))
{
    $ident = opendir($diretorio);
    // laço de repetição para leitura.
    while ($arquivo = readdir($ident))
    {
        echo $arquivo . "\n";
    }
    closedir($ident);
}
?>
```

 **Resultado:**

.

..

```
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.

1.10.2 Concatenação

Para concatenar strings, pode-se utilizar o operador “.” ou colocar múltiplas variáveis dentro de strings duplas “”, uma vez que seu conteúdo é interpretado.

```
<?php  
$fruta = 'maçã';  
  
// primeira forma  
echo $fruta . ' é a fruta de adão'; // resultado = maçã é a fruta de adão  
  
// segunda forma  
echo "{$fruta} é a fruta de adão"; // resultado = maçã é a fruta de adão  
?>
```

O PHP realiza automaticamente a conversão entre tipos, como neste exemplo de concatenação entre uma string e um número:

```
<?php  
$a = 1234;  
  
echo 'O salário é ' . $a . "\n";  
echo "O salário é $a \n";  
?>
```

Resultado:

```
0 salário é 1234  
0 salário é 1234
```

1.10.3 Caracteres de escape

Dentro de aspas duplas “” podemos utilizar controles especiais interpretados diferentemente pelo PHP, que são os caracteres de escape (\). Veja a seguir os mais utilizados:

Caractere	Descrição
\n	Nova linha, proporciona uma quebra de linha.
\r	Retorno de carro.
\t	Tabulação.
\\\	Barra invertida "\\" é o mesmo que \'.
\”	Aspas duplas.
\\$	Símbolo de \$.

Exemplo:

```
<?php
echo "seu nome é \"Paulo\".";// resultado: seu nome é "Paulo".
echo 'seu nome é "Paulo".';// resultado: seu nome é "Paulo".
echo 'seu salário é $650,00';// seu salário é $650,00
echo "seu salário é \$650,00"; // seu salário é $650,00
?>
```

Observação: utilize aspas duplas para declarar strings somente quando for necessário avaliar seu conteúdo, evitando, assim, tempo de processamento desnecessário.

1.10.4 Funções

As funções a seguir formam um grupo cuja característica comum é a manipulação de cadeias de caracteres (strings), como conversões, transformações, entre outras funcionalidades.

strtoupper

Transforma uma string (*conteúdo*) para maiúsculo. Retorna a string com todos os caracteres alfabéticos convertidos para maiúsculo.

`string strtoupper (string conteúdo)`

Exemplo:

```
<?php
echo strtoupper('Convertendo para maiúsculo');
?>
```

Resultado:

CONVERTENDO PARA MAIÚSCULO

strtolower

Transforma uma string (*conteúdo*) para minúsculo. Retorna a string com todos os caracteres alfabéticos convertidos para minúsculo.

`string strtolower (string conteúdo)`

Parâmetros	Descrição
<i>conteúdo</i>	String original a ser transformada.

Exemplo:

```
<?php  
echo strtolower('CONVERTENDO PARA MINÚSCULO');  
?>
```

 **Resultado:**

convertendo para minúsculo

substr

Retorna parte de uma string (conteúdo). Retorna uma porção de conteúdo, começando em *índice*, contendo *comprimento* caracteres. Se *comprimento* for negativo, conta *n* caracteres antes do final.

```
string substr (string conteúdo, int índice [, int comprimento])
```

Parâmetros	Descrição
<i>conteúdo</i>	String original a ser percorrida.
<i>índice</i>	Caractere inicial a ser lido.
<i>comprimento</i>	Comprimento da cadeia de caracteres a ser lida.

Exemplo:

```
<?php  
$rest = substr("América", 1);  
echo $rest . "\n";  
$rest = substr("América", 1, 3);  
echo $rest . "\n";  
$rest = substr("América", 0, -1);  
echo $rest . "\n";  
$rest = substr("América", -2);  
echo $rest . "\n";  
?>
```

 **Resultado:**

mérica
mér
Améric
ca

strpad

Preenche uma string com uma outra string, dentro de um tamanho específico.

```
string str_pad ( string entrada, int tamanho [, string complemento [, int tipo]])
```

Parâmetros	Descrição
<i>entrada</i>	String inicial a ser complementada.
<i>tamanho</i>	Comprimento da string a ser retornada.
<i>complemento</i>	String de preenchimento.
<i>tipo</i>	Tipo de preenchimento. Pode ser STR_PAD_RIGHT (preenche com caracteres à direita), STR_PAD_LEFT (preenche à esquerda) ou STR_PAD_BOTH (preenche em ambos os lados).

Exemplo:

```
<?php
$texto = "The Beatles";
print str_pad($texto, 20) . "\n";
print str_pad($texto, 20, "*", STR_PAD_LEFT) . "\n";
print str_pad($texto, 20, "*", STR_PAD_BOTH) . "\n";
print str_pad($texto, 20, "*") . "\n";
?>
```

Resultado:

```
The Beatles
*****The Beatles
***The Beatles***
The Beatles*****
```

str_repeat

Repete uma string uma certa quantidade de vezes.

```
string str_repeat ( string entrada, int quantidade)
```

Parâmetros	Descrição
<i>entrada</i>	String inicial a ser repetida.
<i>quantidade</i>	Quantidade de repetições.

Exemplo:

```
<?php
$txt = ".o000o.";
print str_repeat($txt, 5) . "\n";
?>
```

Resultado:

```
.o000o..o000o..o000o..o000o..o000o.
```

strlen

Retorna o comprimento de uma string.

```
int strlen ( string entrada )
```

Parâmetros	Descrição
<i>entrada</i>	String cujo comprimento será calculado.

Exemplo:

```
<?php
$txt = "O Rato roeu a roupa do rei de roma";
print 'O comprimento é: ' . strlen($txt) . "\n";
?>
```

Resultado:

O comprimento é: 34

str_replace

Substitui uma string por outra em um dado contexto.

```
mixed str_replace ( mixed procura, mixed substitui, mixed contexto )
```

Parâmetros	Descrição
<i>procura</i>	String a ser substituída.
<i>substitui</i>	String substituta.
<i>contexto</i>	String inicial a ser submetida à substituição.

Exemplo:

```
<?php
$txt = "O Rato roeu a roupa do rei de Roma";
print str_replace('Rato', 'Leão', $txt);
?>
```

Resultado:

O Leão roeu a roupa do rei de Roma

strpos

Encontra a primeira ocorrência de uma string dentro de outra.

```
int strpos (string principal, string procurada [, int offset])
```

Parâmetros	Descrição
<i>principal</i>	String qualquer.
<i>procurada</i>	String a ser procurada dentro da string principal.
<i>offset</i>	Quantidade de caracteres a ser ignorada.

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:

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

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

echo $nomes[0]; // resultado = maria
echo $nomes[1]; // resultado = joão
echo $nomes[2]; // resultado = carlos
echo $nomes[3]; // resultado = jósé
```

1.11.2 Arrays associativos

Os arrays no PHP são associativos pois contêm uma chave de acesso para cada posição. Para criar um array, pode-se utilizar a função `array([chave =>] valor, ...)`.

```
$cores = array('vermelho' => 'FF0000', 'azul' => '0000FF', 'verde' => '00FF00');
```

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

```
$pessoa['nome'] = 'Maria da Silva';
$pessoa['rua'] = 'São João';
$pessoa['bairro'] = 'Cidade Alta';
$pessoa['cidade'] = 'Porto Alegre';
```

De qualquer forma, para acessar o array, basta indicar a sua chave entre colchetes:

```
echo $cores['vermelho']; // resultado = FF0000
echo $cores['azul']; // resultado = 0000FF
echo $cores['verde']; // resultado = 00FF00

echo $pessoa['nome']; // resultado = Maria da Silva
echo $pessoa['rua']; // resultado = São João
echo $pessoa['bairro']; // resultado = Cidade Alta
echo $pessoa['cidade']; // resultado = Porto Alegre
```

Observação: a chave pode ser string ou integer não negativo; o valor pode ser de qualquer tipo.

1.11.3 Iterações

Os arrays podem ser iterados no PHP pelo operador FOREACH, percorrendo cada uma das posições do array. Exemplo:

```
$frutas['cor']      = 'vermelha';
$frutas['sabor']    = 'doce';
$frutas['formato'] = 'redonda';
$frutas['nome']     = 'maçã';
foreach ($frutas as $chave => $fruta)
{
    echo "$chave => $fruta \n";
}
```

 **Resultado:**

```
cor => vermelha
sabor => doce
formato => redonda
nome => maçã
```

1.11.4 Acesso

As posições de um array podem ser acessadas a qualquer momento, e sobre elas operações podem ser realizadas.

```
<?php
$minha_multa['carro'] = 'Pálio';
$minha_multa['valor'] = 178.00;

// alteração de valores
$minha_multa['carro'] .= ' ED 1.0';
$minha_multa['valor'] += 20;

// exibe o array
var_dump($minha_multa);

$comidas[] = 'Lazanha';
$comidas[] = 'Pizza';
$comidas[] = 'Macarrão';

// alteração de valores
$comidas[1] = 'Pizza Calabresa';

// exibe o array
var_dump($comidas);
?>
```

 **Resultado:**

```
array(2) {
    ["carro"]=>
        string(12) "Pálio ED 1.0"
    ["valor"]=>
        float(198)
}
array(3) {
    [0]=>
        string(7) "Lazanha"
    [1]=>
        string(15) "Pizza Calabresa"
    [2]=>
        string(8) "Macarrão"
}
```

1.11.5 Arrays multidimensionais

Arrays multidimensionais ou matrizes são arrays nos quais algumas de suas posições podem conter outros arrays de forma recursiva. Um array multidimensional pode ser criado pela função `array()`:

```
<?php
$carros = array('Palio' => array('cor'=>'azul',
                                    'potência'=>'1.0',
                                    'opcionais'=>'Ar Cond.'),
                'Corsa' => array('cor'=>'cinza',
                                    'potência'=>'1.3',
                                    'opcionais'=>'MP3'),
                'Gol'    => array('cor'=>'branco',
                                    'potência' => '1.0',
                                    'opcionais' => 'Metalica')
            );
echo $carros['Palio']['opcionais'];      // resultado = Ar Cond.
?>
```

Outra forma de criar um array multidimensional é simplesmente atribuindo-lhe valores:

```
<?php
$carros['Palio']['cor']      = 'azul';
$carros['Palio']['potência'] = '1.0';
$carros['Palio']['opcionais'] = 'Ar Cond.';
$carros['Corsa']['cor']       = 'cinza';
```

```
$carros['Corsa']['potência'] = '1.3';
$carros['Corsa']['opcionais'] = 'MP3';
$carros['Gol']['cor'] = 'branco';
$carros['Gol']['potência'] = '1.0';
$carros['Gol']['opcionais'] = 'Metalica';

echo $carros['Palio']['opcionais']; // resultado = Ar Cond.
?>
```

Para realizar iterações em um array multidimensional é preciso observar quantos níveis ele possui. No exemplo a seguir, realizamos uma iteração para o primeiro nível do array (veículos) e, para cada iteração, realizamos uma nova iteração, para imprimir suas características.

```
<?php
foreach ($carros as $modelo => $características)
{
    echo "> modelo $modelo\n";
    foreach ($características as $característica => $valor)
    {
        echo "característica $característica => $valor\n";
    }
}
?>
```

Resultado:

```
=> modelo Palio
característica cor => azul
característica potência => 1.0
característica opcionais => Ar Cond.

=> modelo Corsa
característica cor => cinza
característica potência => 1.3
característica opcionais => MP3

=> modelo Gol
característica cor => branco
característica potência => 1.0
característica opcionais => Metalica
```

1.11.6 Funções

A seguir veremos uma série de funções utilizadas exclusivamente para manipulação de arrays, funções de ordenação, intersecção, acesso, dentre outras.

array_push

Adiciona elementos ao final de um array. Tem o mesmo efeito de utilizar a sintaxe `$array[] = $valor.`

```
int array_push (array nome_array, mixed valor [, mixed ...])
```

Parâmetros	Descrição
<i>nome_array</i>	Array a ser acrescido do valor.
<i>valor</i>	Valor a ser adicionado.
...	Pode-se adicionar <i>n</i> valores.

Exemplo:

```
<?php
$a = array("verde", "azul", "vermelho");
array_push($a, "amarelo");
var_dump($a);
?>
```

Resultado:

```
array(4) {
[0]=> string(5) "verde"
[1]=> string(4) "azul"
[2]=> string(8) "vermelho"
[3]=> string(7) "amarelo"
}
```

array_pop

Remove um valor do final de um array.

```
mixed array_pop (array nome_array)
```

Exemplo:

```
<?php
$a = array("verde", "azul", "vermelho", "amarelo");
array_pop($a);
var_dump($a);
?>
```

Resultado:

```
array(3) {
[0]=> string(5) "verde"
[1]=> string(4) "azul"
[2]=> string(8) "vermelho"
}
```

array_shift

Remove um elemento do início de um array.

```
mixed array_shift (array nome_array)
```

Exemplo:

```
<?php
$a = array("verde", "azul", "vermelho", "amarelo");
array_shift($a);
var_dump($a);
?>
```

Resultado:

```
array(3) {
[0]=> string(4) "azul"
[1]=> string(8) "vermelho"
[2]=> string(7) "amarelo"
}
```

array_unshift

Adiciona um elemento no início de um array.

```
int array_unshift (array nome_array, mixed valor [, mixed ...])
```

Parâmetros	Descrição
<i>nome_array</i>	Array a ser acrescido do valor.
<i>valor</i>	Valor a ser adicionado.
...	Pode-se adicionar <i>n</i> valores.

Exemplo:

```
<?php
$a = array("verde", "azul", "vermelho");
array_unshift($a, "amarelo");
var_dump($a);
?>
```

Resultado:

```
array(4) {
[0]=> string(7) "amarelo"
[1]=> string(5) "verde"
[2]=> string(4) "azul"
[3]=> string(8) "vermelho"
}
```

array_pad

Preenche um array com um dado valor, determinada quantidade de posições.

```
array array_pad (array nome_array, int tamanho, mixed valor)
```

Parâmetros	Descrição
<i>nome_array</i>	Array a ser preenchido.
<i>tamanho</i>	Quantidade de posições.
<i>valor</i>	Valor a ser preenchido.

Exemplo:

```
<?php
$a = array("verde", "azul", "vermelho");
$a = array_pad($a, 6, "branco");
var_dump($a);
?>
```

 **Resultado:**

```
array(6) {
[0]=> string(5) "verde"
[1]=> string(4) "azul"
[2]=> string(8) "vermelho"
[3]=> string(6) "branco"
[4]=> string(6) "branco"
[5]=> string(6) "branco"
}
```

array_reverse

Recebe um array e retorna-o na ordem inversa.

```
array array_reverse (array nome_array, boolean preservar_chaves)
```

Parâmetros	Descrição
<i>nome_array</i>	Array a ser revertido.
<i>preservar_chaves</i>	Manter a associação de índices.

Exemplo:

```
<?php
$a[0] = 'green';
$a[1] = 'yellow';
$a[2] = 'red';
$a[3] = 'blue';
$b = array_reverse($a, true);
var_dump($b);
?>
```

Resultado:

```
array(4) {
    [3]=> string(4) "blue"
    [2]=> string(3) "red"
    [1]=> string(6) "yellow"
    [0]=> string(5) "green"
}
```

array_merge

Mescla dois ou mais arrays. Um array é adicionado ao final do outro. O resultado é um novo array. Se ambos arrays tiverem conteúdo indexado pela mesma chave, o segundo irá se sobrepor ao primeiro.

```
array array_merge (array nome_array1, array nome_array2 [, array ...])
```

Parâmetros	Descrição
<i>nome_array1</i>	Primeiro array a ser mesclado.
<i>nome_array2</i>	Segundo array a ser mesclado.
...	Pode-se mesclar <i>n</i> arrays.

Exemplo:

```
<?php
$a = array("verde", "azul");
$b = array("vermelho", "amarelo");
$c = array_merge($a, $b);
var_dump($c);
?>
```

Resultado:

```
array(4) {
    [0]=> string(5) "verde"
    [1]=> string(4) "azul"
    [2]=> string(8) "vermelho"
    [3]=> string(7) "amarelo"
}
```

array_keys

Retorna as chaves (índices) de um array. Se o segundo parâmetro for indicado, a função retornará apenas índices que apontam para um conteúdo igual ao parâmetro.

```
array array_keys (array nome_array [, mixed valor_procurado])
```

Parâmetros	Descrição
<i>nome_array</i>	Array cujos índices desejamos descobrir.
<i>valor_procurado</i>	Parâmetro opcional. Se preenchido, retornará apenas índices contendo este valor.

Exemplo:

```
<?php
$exemplo = array('cor' => 'vermelho', 'volume' => 5, 'animal'=>'cachorro');
$indices = array_keys($exemplo);
print_r($indices);
?>
```

Resultado:

```
Array
(
    [0] => cor
    [1] => volume
    [2] => animal
)
```

array_values

Retorna um array contendo os valores de outro array.

array array_values (array *nome_array*)

Parâmetros	Descrição
<i>nome_array</i>	Array cujos valores desejamos descobrir.

Exemplo:

```
<?php
$exemplo = array('cor' => 'vermelho', 'volume' => 5, 'animal'=>'cachorro');
$valores = array_values($exemplo);
print_r($valores);
?>
```

Resultado:

```
Array
(
    [0] => vermelho
    [1] => 5
    [2] => cachorro
)
```

array_slice

Extrai uma porção de um array.

```
array array_slice (array nome_array, int índice_início, int tamanho)
```

Parâmetros	Descrição
<i>nome_array</i>	Array cuja porção desejamos extrair.
<i>índice_início</i>	Primeira posição a ser extraída.
<i>tamanho</i>	Tamanho da porção extraída.

Exemplo:

```
<?php
$a[0] = 'green';
$a[1] = 'yellow';
$a[2] = 'red';
$a[3] = 'blue';
$a[4] = 'gray';
$a[5] = 'white';
$b = array_slice($a, 2, 3);
print_r($b);
?>
```

 **Resultado:**

```
Array
(
    [0] => red
    [1] => blue
    [2] => gray
)
```

count

Retorna a quantidade de elementos de um array.

```
int count (array nome_array)
```

Exemplo:

```
<?php
$a = array('refrigerante', 'cerveja', 'vodka', 'suco natural');
echo 'o array $a contém ' . count($a) . ' posições';
?>
```

 **Resultado:**

o array \$a contém 4 posições

array_in

Verifica se um array contém um determinado valor.

```
boolean array_in (mixed valor, array nome_array)
```

Parâmetro	Descrição
<i>valor</i>	Valor a ser procurado.
<i>nome_array</i>	Array a ser vasculhado.

Exemplo:

```
<?php
$a = array('refrigerante', 'cerveja', 'vodka', 'suco natural');
if (in_array('suco natural', $a))
{
    echo 'suco natural encontrado';
}
?>
```

 **Resultado:**

suco natural encontrado

sort

Ordena um array pelo seu valor, não mantendo a associação de índices.

```
boolean sort (array nome_array)
```

Exemplo:

```
<?php
$a = array('refrigerante', 'cerveja', 'vodka', 'suco natural');
sort($a);
print_r($a);
?>
```

 **Resultado:**

```
Array
(
    [0] => cerveja
    [1] => refrigerante
    [2] => suco natural
    [3] => vodka
)
```

rsort

Ordena um array em ordem reversa pelo seu valor, não mantendo a associação de índices.

```
boolean rsort (array nome_array)
```

Exemplo:

```
<?php
$a = array('refrigerante', 'cerveja', 'vodka', 'suco natural');
rsort($a);
print_r($a);
?>
```

 **Resultado:**

```
Array
(
    [0] => vodka
    [1] => suco natural
    [2] => refrigerante
    [3] => cerveja
)
```

asort

Ordena um array pelo seu valor, mantendo a associação de índices. Para ordenar de forma reversa, use **arsort()**.

```
void asort (array nome_array)
```

Exemplo:

```
<?php
$a[0] = 'green';
$a[1] = 'yellow';
$a[2] = 'red';
$a[3] = 'blue';
$a[4] = 'gray';
$a[5] = 'white';
asort($a);
print_r($a);
?>
```

 **Resultado:**

```
Array
(
```

```
[3] => blue  
[4] => gray  
[0] => green  
[2] => red  
[5] => white  
[1] => yellow  
)
```

ksort

Ordena um array pelos seus índices. Para ordem reversa, utilize `krsort()`.

```
boolean ksort (array nome_array)
```

Exemplo:

```
<?php  
$carro['potência'] = '1.0';  
$carro['cor'] = 'branco';  
$carro['modelo'] = 'celta';  
$carro['opcionais'] = 'ar quente';  
ksort($carro);  
print_r($carro);  
?>
```

Resultado:

```
Array  
(  
    [cor] => branco  
    [modelo] => celta  
    [opcionais] => ar quente  
    [potência] => 1.0  
)
```

explode

Converte uma string em um array, separando os elementos por meio de um separador.

```
array explode (string separador, string padrão)
```

Parâmetros	Descrição
<i>separador</i>	Caractere que será utilizado para desmembrar a string, convertendo-a em um array.
<i>padrão</i>	String que desejamos converter em um array.

Exemplo:

```
<?php
$string = "31/12/2004";
var_dump(explode("/", $string));
?>
```

Resultado:

```
array(3) {
    [0]=>
    string(2) "31"
    [1]=>
    string(2) "12"
    [2]=>
    string(4) "2004"
}
```

implode

Converte um array em uma string, separando os elementos do array por meio de um separador.

`array implode (string separador, array padrão)`

Parâmetros	Descrição
<i>separador</i>	Caractere que será utilizado para delimitar a nova string criada.
<i>padrão</i>	Array original que desejamos converter em string.

Exemplo:

```
<?php
$padrao = array('Maria', 'Paulo', 'José');
$resultado = implode(' + ', $padrao);

var_dump($resultado);
?>
```

Resultado:

```
string(20) "Maria + Paulo + José"
```

1.12 Manipulação de objetos

Nesta seção, veremos uma série de funções relacionadas à manipulação de objetos. Para maiores detalhes sobre orientação a objetos, veja o Capítulo 2.

get_class_methods

Retorna um vetor com os nomes dos métodos de uma determinada classe.

```
array get_class_methods (string nome_classe)
```

Exemplo:

```
<?php
class Funcionario
{
    function SetSalario()
    {
    }
    function GetSalario()
    {
    }
    function SetNome()
    {
    }
    function GetNome()
    {
}
}

print_r(get_class_methods('Funcionario'));
?>
```

Resultado:

```
Array
(
    [0] => SetSalario
    [1] => GetSalario
    [2] => SetNome
    [3] => GetNome
)
```

get_class_vars

Retorna um vetor com os nomes das propriedades e conteúdos de uma determinada classe. Note que são valores estáticos definidos na criação da classe.

```
array get_class_vars (string nome_classe)
```

Exemplo:

```
<?php
class Funcionario
{
    var $Codigo;
    var $Nome;
    var $Salario = 586;
    var $Departamento = 'Contabilidade';

    function SetSalario()
    {
    }

    function GetSalario()
    {
    }
}

print_r(get_class_vars('Funcionario'));
?>
```

Resultado:

```
Array
(
    [Codigo] =>
    [Nome] =>
    [Salario] => 586
    [Departamento] => Contabilidade
)
```

get_object_vars

Retorna um vetor com os nomes e conteúdos das propriedades de um objeto. São valores dinâmicos que se alteram de acordo com o ciclo de vida do objeto.

```
array get_object_vars (object nome_objeto)
```

Exemplo:

```
<?php
class Funcionario
{
    var $Codigo;
    var $Nome;
    var $Salario = 586;
    var $Departamento = 'Contabilidade';
    function SetSalario()
    {
    }
}
```

```
function GetSalario()
{
}
}

$jose = new Funcionario;
$jose->Codigo = 44;
$jose->Nome = 'José da Silva';
$jose->Salario += 100;
$jose->Departamento = 'Financeiro';

print_r(get_object_vars($jose));
?>
```

Resultado:

```
Array
(
    [Codigo] => 44
    [Nome] => José da Silva
    [Salario] => 686
    [Departamento] => Financeiro
)
```

get_class

Retorna o nome da classe a qual um objeto pertence.

```
string get_class (object nome_objeto)
```

Exemplo:

```
<?php
class Funcionario
{
    var $Codigo;
    var $Nome;

    function SetSalario()
    {
    }
    function GetSalario()
    {
    }
}

$jose = new Funcionario;
echo get_class($jose);
?>
```

 **Resultado:**

```
Funcionario
```

get_parent_class

Retorna o nome da classe ancestral (classe-pai). Se o parâmetro for um objeto, retorna o nome da classe ancestral da classe à qual o objeto pertence. Se o parâmetro for uma string, retorna o nome da classe ancestral da classe passada como parâmetro.

```
string get_parent_class (mixed objeto)
```

Parâmetros	Descrição
<i>objeto</i>	Objeto ou nome de uma classe.

Exemplo:

```
<?php
class Funcionario
{
    var $Codigo;
    var $Nome;
}

class Estagiario extends Funcionario
{
    var $Salario;
    var $Bolsa;
}

$jose = new Estagiario;

echo get_parent_class($jose);
echo "\n"; // quebra de linha
echo get_parent_class('estagiario');
?>
```

 **Resultado:**

```
Funcionario
Funcionario
```

is_subclass_of

Indica se um determinado objeto ou classe é derivado de uma determinada classe.

```
boolean is_subclass_of (mixed objeto, string classe)
```

Parâmetros	Descrição
<i>objeto</i>	Objeto ou nome de uma classe.
<i>classe</i>	Nome de uma classe ancestral.

Exemplo:

```
<?php
class Funcionario
{
    var $Codigo;
    var $Nome;
}

class Estagiario extends Funcionario
{
    var $Salario;
    var $Bolsa;
}

$jose = new Estagiario;

if (is_subclass_of($jose, 'Funcionario'))
{
    echo "Classe do objeto Jose é derivada de Funcionario";
}

echo "\n"; // quebra de linha

if (is_subclass_of('Estagiario', 'Funcionario'))
{
    echo "Classe Estagiario é derivada de Funcionario";
}
?>
```

Resultado:

```
Classe do objeto Jose é derivada de Funcionario
Classe Estagiario é derivada de Funcionario
```

method_exists

Verifica se um determinado objeto possui o método descrito. Podemos verificar a existência de um método antes de executar por engano um método inexistente.

```
boolean method_exists (object objeto, string método)
```

Parâmetros	Descrição
<i>objeto</i>	Objeto qualquer.
<i>método</i>	Nome de um método do objeto.

Exemplo:

```
<?php
class Funcionario
{
    var $Codigo;
    var $Nome;

    function GetSalario()
    {
    }

    function SetSalario()
    {
    }
}

$jose = new Funcionario;

if (method_exists($jose, 'SetNome'))
{
    echo 'Objeto Jose possui método SetNome()';
}
if (method_exists($jose, 'SetSalario'))
{
    echo 'Objeto Jose possui método SetSalario()';
}
?>
```

Resultado:

Objeto Jose possui método SetSalario()

call_user_func

Executa uma função ou um método de uma classe passado como parâmetro. Para executar uma função, basta passar seu nome como uma string, e, para executar um método de um objeto, basta passar o parâmetro como um array contendo na posição 0 o objeto e na posição 1 o método a ser executado. Para executar métodos estáticos, basta passar o nome da classe em vez do objeto na posição 0 do array.

mixed call_user_func (callback *função* [, mixed *parâmetro* [, mixed ...]])

Parâmetros	Descrição
<i>função</i>	Função a ser executada.
<i>parâmetro</i>	Parâmetro(s) da função.

Exemplo:

```
<?php  
// exemplo chamada simples  
function minhafuncao()  
{  
    echo "minha função! \n";  
}  
  
call_user_func('minhafuncao');  
  
// declaração de classe  
class MinhaClasse  
{  
    function MeuMetodo()  
    {  
        echo "Meu método! \n";  
    }  
}  
  
// chamada de método estático  
call_user_func(array('MinhaClasse', 'MeuMetodo'));  
  
// chamada de método  
$obj = new MinhaClasse();  
call_user_func(array($obj, 'MeuMetodo'));  
?>
```

 **Resultado:**

```
minha função!  
Meu método!  
Meu método!
```

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

bem específica e podem ser executadas várias vezes. As funções, por exemplo, podem receber parâmetros fazendo com que o resultado do seu processamento interno varie de acordo com os argumentos (parâmetros) de entrada, sendo possível executar esta função sob diferentes circunstâncias, o que caracteriza outro conceito importante da engenharia de software – o reuso.

Na programação estruturada, as unidades do código (funções) se interligam por meio de três mecanismos básicos: seqüência, decisão e iteração, como ilustrado na Figura 2.1.

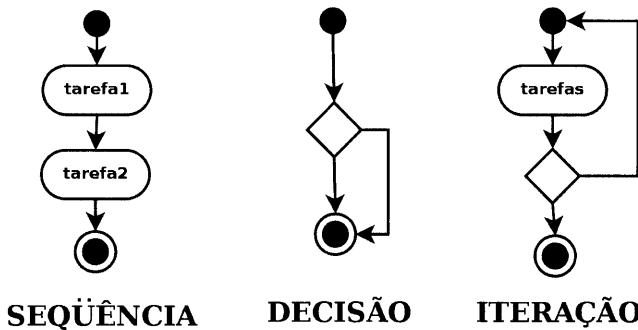


Figura 2.1 – Programação estruturada.

A seqüência representa os passos necessários para executar um programa em função de suas tarefas desempenhadas, por exemplo: 1) Leia os valores digitados; 2) Exiba a soma na tela.

A decisão permite selecionar um determinado fluxo de processamento baseado em determinadas expressões lógicas, por exemplo: SE o valor digitado for maior que 20, execute a função `alerta_usuario()` SENÃO execute a função `grava_dados()`.

A iteração permite a execução repetitiva de um determinado bloco de comandos do programa. Esta execução geralmente tem um ponto de entrada representado por uma expressão lógica. O bloco de comandos é executado repetitivamente enquanto a expressão for verdadeira.

2.1.2 Orientação a objetos

Ao trabalharmos com orientação a objetos é fundamental entender o conceito de classes e objetos. Uma classe é uma estrutura que define um tipo de dados, podendo conter atributos (variáveis) e também funções (métodos) para manipular esses atributos. No exemplo a seguir, declararemos a classe `Produto` com quatro propriedades, ou seja, quatro “variáveis” que existem dentro do contexto do objeto. Declaramos propriedades por meio da palavra-chave `var`, dentre formas vistas a seguir.

Observação: recomenda-se iniciar nomes de classes com a letra maiúscula e, de preferência, evitar a utilização do caractere “_”. Por exemplo, utilize `class CestaDeCompras` e não `class cesta_de_compras`.

Produto.class.php

```
<?php  
class Produto  
{  
    var $Codigo;  
    var $Descricao;  
    var $Preco;  
    var $Quantidade;  
}  
?>
```

Um objeto contém exatamente a mesma estrutura e as propriedades de uma classe, no entanto sua estrutura é dinâmica, seus atributos podem mudar de valor durante a execução do programa e podemos declarar diversos objetos oriundos de uma mesma classe. Na Figura 2.2, ilustramos exemplos de objetos do mundo real.



Figura 2.2 – Objetos do mundo real.

No exemplo a seguir, estamos fazendo uso da classe `Produto`, criando um objeto (`$produto`). Para instanciar um objeto é utilizado o operador `new`, seguido do nome da classe. Para acessar propriedades de um objeto em nosso programa, utilizamos o nome da propriedade precedido do nome do objeto.

Um objeto não foi feito para ser impresso diretamente, mas suas propriedades sim. Entretanto, se tentarmos imprimir um objeto diretamente (como na última linha do programa a seguir), o PHP retornará o identificador interno deste objeto na memória. Mesmo que criássemos vários objetos de uma mesma classe, cada um deles possuiria um identificador próprio e um comportamento único, diferenciando uns dos outros.

Observação: neste capítulo gravaremos todas as classes no diretório `classes` para facilitar a manutenção dos arquivos. Veja que introduzimos a classe `Produto` com o comando `include_once`.

 **objeto.php**

```
<?php  
// insere a classe  
include_once 'classes/Produto.class.php';  
  
// cria um objeto  
$produto = new Produto;  
  
// atribuir valores  
$produto->Codigo = 4001;  
$produto->Descricao = 'CD - The Best of Eric Clapton';  
  
echo $produto;  
?>
```

 **Resultado:**

```
Object id #1
```

Reescreveremos, então, a classe `Produto` para adicionar uma funcionalidade ou um método, que é uma função declarada dentro da estrutura da classe, agindo dentro desse escopo. O método criado, `ImprimeEtiqueta()`, trabalha com as propriedades do objeto. A fim de diferenciar as propriedades de um objeto de variáveis locais, utiliza-se a pseudovariável `$this` para representar o objeto atual e acessar suas propriedades.

A nomenclatura do método deve transmitir a ação desempenhada. O próprio nome deve transmitir o que o método realiza. Por exemplo: `ObtemDados()` ou `SalvaDados()`.

 **Produto.class.php**

```
<?php  
class Produto  
{  
    var $Codigo;  
    var $Descricao;  
    var $Preco;  
    var $Quantidade;  
  
    function ImprimeEtiqueta()  
    {  
        print 'Código: ' . $this->Codigo . "\n";  
        print 'Descrição: ' . $this->Descricao . "\n";  
    }  
}  
?>
```

Reescreveremos, então, o exemplo anterior para demonstrar a utilização do método `ImprimeEtiqueta()` e criaremos dois objetos para demonstrar tal característica.

 **objeto.php**

```
<?php  
// insere a classe  
include_once 'classes/Produto.class.php';  
  
// cria dois objetos  
$produto1 = new Produto;  
$produto2 = new Produto;  
  
// atribuir valores  
$produto1->Codigo = 4001;  
$produto1->Descricao = 'CD - The Best of Eric Clapton';  
  
$produto2->Codigo = 4002;  
$produto2->Descricao = 'CD - The Eagles Hotel California';  
  
// imprime informações de etiqueta  
$produto1->ImprimeEtiqueta();  
$produto2->ImprimeEtiqueta();  
?>
```

 **Resultado:**

Código: 4001
Descrição: CD - The Best of Eric Clapton
Código: 4002
Descrição: CD - The Eagles Hotel California

2.2 Classe

A classe é uma estrutura estática utilizada para descrever objetos mediante atributos (propriedades) e métodos (funcionalidades). A classe é um modelo ou template para criação desses objetos. Tem-se por propriedades características intrínsecas à classe em questão.

Podem ser classes: entidades do negócio da aplicação (pessoa, conta, cliente, fornecedor), entidades de interface (janela, botão, painel, frame, barra), dentre outras (conexão com banco de dados, um arquivo-texto, um arquivo XML, uma conexão SSH, uma conexão FTP, um Web Service).

No momento em que modelamos uma classe **Pessoa**, são suas propriedades: **Nome**, **Altura**, **Idade**, **Nascimento**, **Escolaridade** e **Salario**. Tem-se por métodos funcionalidades desempenhadas pela classe. No caso da classe **Pessoa**, poderiam ser métodos: **Crescer()**, **Formar()** e **Envelhecer()**.

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:

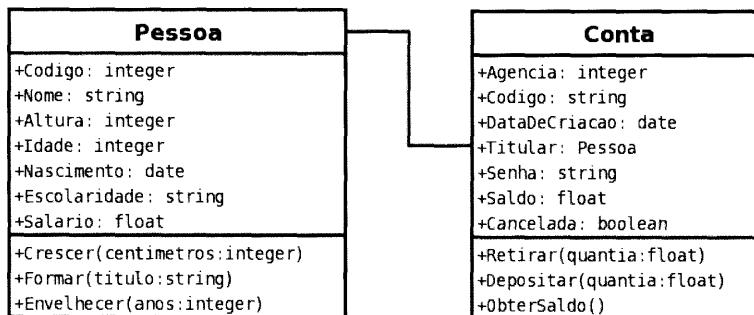


Figura 2.3 – Classes.

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
     */
}
  
```

```

function Crescer($centimetros)
{
    if ($centimetros > 0)
    {
        $this->Altura += $centimetros;
    }
}

/* método Formar
 * altera a Escolaridade para $titulacao
 */
function Formar($titulacao)
{
    $this->Escolaridade = $titulacao;
}

/* método Envelhecer
 * aumenta a Idade em $anos
 */
function Envelhecer($anos)
{
    if ($anos > 0)
    {
        $this->Idade += $anos;
    }
}
?>

```

Conta.class.php

```

<?php
class Conta
{
    var $Agencia;
    var $Codigo;
    var $DataDeCriacao;
    var $Titular;
    var $Senha;
    var $Saldo;
    var $Cancelada;

    /* método Retirar
     * diminui o Saldo em $quantia
     */
    function Retirar($quantia)
    {
        if ($quantia > 0)

```

```
{  
    $this->Saldo -= $quantia;  
}  
}  
  
/* método Depositar  
 * aumenta o Saldo em $quantia  
 */  
function Depositar($quantia)  
{  
    if ($quantia > 0)  
    {  
        $this->Saldo += $quantia;  
    }  
}  
  
/* método ObterSaldo  
 * retorna o Saldo Atual  
 */  
function ObterSaldo()  
{  
    return $this->Saldo;  
}  
}  
?>
```

Note que dentro de uma classe, para referenciar um de seus membros (propriedades), basta utilizar-se da expressão `$this`, que nada mais é do que o nome de uma pseudovariável interna que representa o próprio objeto que estará sendo manipulado.

2.3 Objeto

Um objeto é uma estrutura dinâmica originada com base em uma classe. Após a utilização de uma classe para criar diversas estruturas iguais a ela, que interagem no sistema e possuem dados nela armazenados, dizemos que estamos criando objetos, ou mesmo instanciando objetos de uma classe. Diz-se que o objeto é uma instância de uma classe, porque o objeto existe durante um dado instante de tempo – da sua criação até a sua destruição. São objetos da classe `Pessoa`: Carlos, João, Maria e José. Todos têm uma estrutura igual (como a da classe `Pessoa`), mas propriedades com valores diferentes, caracterizando cada um de forma distinta, dando-lhes unicidade no sistema. Na Figura 2.4, ilustramos exemplos de objetos relacionados.

Para instanciar um objeto de uma determinada classe, procedemos para a declaração de uma variável qualquer (nossa objeto em questão) e lhe atribuímos o operador `new` seguido do nome da classe que desejamos instanciar. Veja a seguir um exemplo da utilização de objetos.

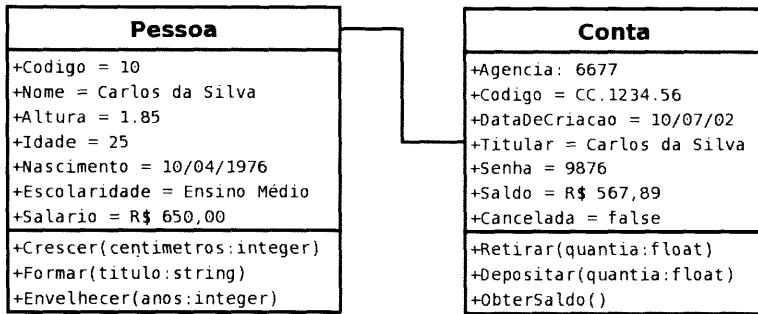


Figura 2.4 – Objetos.

objetos.php

```

<?php
# carrega as classes
include_once 'classes/Pessoa.class.php';
include_once 'classes/Conta.class.php';

# criação do objeto $carlos
$carlos = new Pessoa;
$carlos->Codigo = 10;
$carlos->Nome = "Carlos da Silva";
$carlos->Altura = 1.85;
$carlos->Idade = 25;
$carlos->Nascimento = '10/04/1976';
$carlos->Escolaridade = "Ensino Médio";

echo "Manipulando o objeto $carlos->Nome :\n";

echo "{$carlos->Nome} é formado em: {$carlos->Escolaridade} \n";
$carlos->Formar('Técnico em Eletricidade');
echo "{$carlos->Nome} é formado em: {$carlos->Escolaridade} \n";

echo "{$carlos->Nome} possui {$carlos->Idade} anos \n";
$carlos->Envelhecer(1);
echo "{$carlos->Nome} possui {$carlos->Idade} anos \n";

# criação do objeto $conta_carlos
$conta_carlos = new Conta;
$conta_carlos->Agencia = 6677;
$conta_carlos->Codigo = "CC.1234.56";
$conta_carlos->DataDeCriacao = "10/07/02";
$conta_carlos->Titular = $carlos;
$conta_carlos->Senha = 9876;
$conta_carlos->Saldo = 567.89;
$conta_carlos->Cancelada = false;

```

```
echo "\n";
echo "Manipulando a conta de: {$conta_carlos->Titular->Nome} \n";
echo "O saldo atual é R\$ {$conta_carlos->ObterSaldo()} \n";

$conta_carlos->Depositar(20);
echo "O saldo atual é R\$ {$conta_carlos->ObterSaldo()} \n";

$conta_carlos->Retirar(10);
echo "O saldo atual é R\$ {$conta_carlos->ObterSaldo()} \n";
?>
```

Resultado:

Manipulando o objeto Carlos da Silva :
Carlos da Silva é formado em: Ensino Médio
Carlos da Silva é formado em: Técnico em Eletricidade
Carlos da Silva possui 25 anos
Carlos da Silva possui 26 anos

Manipulando a conta de: Carlos da Silva
O saldo atual é R\$ 567.89
O saldo atual é R\$ 587.89
O saldo atual é R\$ 577.89

Observação: note que, para acessar propriedades e métodos de um objeto dentro de uma string dupla (que é interpretada), é necessário utilizar-se de chaves ao redor da expressão.

2.4 Construtores e destrutores

Um construtor é um método especial utilizado para definir o comportamento inicial de um objeto, ou seja, o comportamento no momento de sua criação. O método construtor é executado automaticamente no momento em que instanciamos um objeto por meio do operador `new`. Assim, não devemos retornar nenhum valor por meio do método construtor porque o mesmo retorna por definição o próprio objeto que está sendo instanciado.

Caso não seja definido um método construtor, automaticamente todas as propriedades do objeto criado são inicializadas com o valor `NULL`. Nos casos mostrados anteriormente (Pessoa e Conta), há a necessidade de um método construtor para definir os valores iniciais para as suas propriedades.

Observação: para definir um método construtor em uma determinada classe basta declarar o método `__construct()`.

Um destrutor ou finalizador é um método especial executado automaticamente quando o objeto é desalocado da memória, quando atribuímos o valor `NULL` ao objeto, quando utilizamos a função `unset()` sobre o mesmo ou, em última instância, quando o programa é finalizado. O método destrutor pode ser utilizado para finalizar conexões, apagar arquivos temporários criados durante o ciclo de vida do objeto, dentre outras circunstâncias.

Observação: para definir um método destrutor em uma determinada classe basta declarar o método `__destruct()`.

A seguir, complementaremos as classes `Pessoa` e `Conta` adicionando os métodos construtores e destrutores. Nesta etapa suprimiremos parte do código criada anteriormente.

Pessoa.class.php (complemento)

```
<?php
class Pessoa
{
    #
    # ...
    # conteúdo já escrito no exemplo anterior
    #

    /* método construtor
     * inicializa propriedades
     */
    function __construct($Codigo, $Nome, $Altura, $Idade, $Nascimento, $Escolaridade, $Salario)
    {
        $this->Codigo = $Codigo;
        $this->Nome = $Nome;
        $this->Altura = $Altura;
        $this->Idade = $Idade;
        $this->Nascimento = $Nascimento;
        $this->Escolaridade = $Escolaridade;
        $this->Salario = $Salario;
    }

    /* método destrutor
     * finaliza Objeto
     */
    function __destruct()
    {
        echo "Objeto {$this->Nome} finalizado...\n";
    }
}
?>
```

 Conta.class.php (complemento)

```
<?php
class Conta
{
    # ...
    # conteúdo já escrito no exemplo anterior
    # ...

    /* método construtor
     * inicializa propriedades
     */
    function __construct($Agencia, $Codigo, $DataDeCriacao, $Titular, $Senha, $Saldo)
    {
        $this->Agencia = $Agencia;
        $this->Codigo = $Codigo;
        $this->DataDeCriacao = $DataDeCriacao;
        $this->Titular = $Titular;
        $this->Senha = $Senha;

        // chamada a outro método da classe
        $this->Depositar($Saldo);
        $this->Cancelada = false;
    }

    /* método destrutor
     * finaliza objeto
     */
    function __destruct()
    {
        echo "Objeto Conta {$this->Codigo} de {$this->Titular->Nome} finalizada...\n";
    }
}
?>
```

 construtores.php

```
<?php
# carrega as classes
include_once 'classes/Pessoa.class.php';
include_once 'classes/Conta.class.php';

# criação do objeto $carlos
$carlos = new Pessoa(10, "Carlos da Silva", 1.85, 25, "10/04/1976", "Ensino Médio", 650.00);

echo "Manipulando o objeto {$carlos->Nome}: \n";
```

```

echo "{$carlos->Nome} é formado em: {$carlos->Escolaridade} \n";
$carlos->Formar('Técnico em Eletricidade');
echo "{$carlos->Nome} é formado em: {$carlos->Escolaridade} \n";

echo "{$carlos->Nome} possui {$carlos->Idade} anos \n";
$carlos->Envelhecer(1);
echo "{$carlos->Nome} possui {$carlos->Idade} anos \n";

# Criação do objeto $conta_carlos
$conta_carlos = new Conta(6677, "CC.1234.56", "10/07/02", $carlos, 9876, 567.89);

echo "\n";
echo "Manipulando a conta de: {$conta_carlos->Titular->Nome}: \n";

echo "O saldo atual é R\$ {$conta_carlos->ObterSaldo()} \n";
$conta_carlos->Depositar(20);
echo "O saldo atual é R\$ {$conta_carlos->ObterSaldo()} \n";
$conta_carlos->Retirar(10);
echo "O saldo atual é R\$ {$conta_carlos->ObterSaldo()} \n";
?>

```

Resultado:

Manipulando o objeto Carlos da Silva:
 Carlos da Silva é formado em: Ensino Médio
 Carlos da Silva é formado em: Técnico em Eletricidade
 Carlos da Silva possui 25 anos
 Carlos da Silva possui 26 anos

Manipulando a conta de: Carlos da Silva:
 O saldo atual é R\$ 567.89
 O saldo atual é R\$ 587.89
 O saldo atual é R\$ 577.89
 Objeto Carlos da Silva finalizado...
 Objeto Conta CC.1234.56 de Carlos da Silva finalizada...

2.5 Herança

A utilização da orientação a objetos e do encapsulamento do código em classes nos orienta em direção a uma maior organização, mas um dos maiores benefícios que encontramos na utilização desse paradigma é o reuso. A possibilidade de reutilizar partes de código já definidas é o que nos dá maior agilidade no dia-a-dia, além de eliminar a necessidade de eventuais duplicações ou reescritas de código.

Quando falamos em herança, a primeira imagem que nos aparece na memória é a de uma árvore genealógica com avós, pais, filhos e nas características que são

transmitidas geração após geração. O que devemos levar em consideração sobre herança em orientação a objetos é o compartilhamento de atributos e comportamentos entre as classes de uma mesma hierarquia (árvore). As classes inferiores da hierarquia automaticamente herdam todas as propriedades e os métodos das classes superiores, chamadas de superclasses.

Este recurso tem uma aplicabilidade muito grande, visto que é relativamente comum termos de criar novas funcionalidades em software. Utilizando a herança, em vez de criarmos uma estrutura totalmente nova (uma classe), podemos reaproveitar uma estrutura já existente que nos forneça uma base abstrata para o desenvolvimento, provendo recursos básicos e comuns.

Já criamos a classe genérica `Conta`; agora podemos aproveitar seu código-fonte para criar classes mais específicas como `ContaCorrente` e `ContaPoupanca`, como no diagrama a seguir:

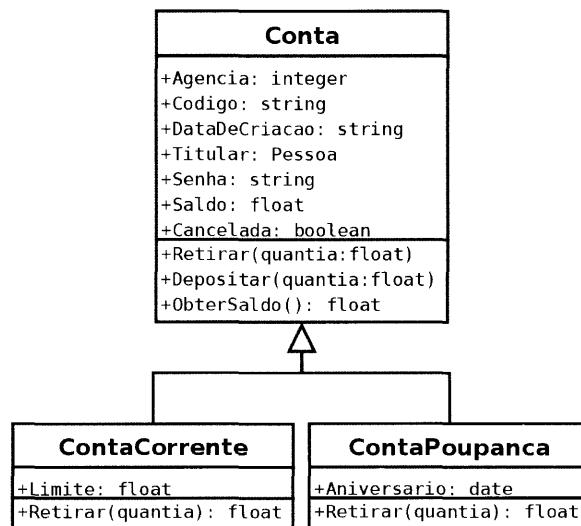


Figura 2.5 – Herança de classes.

Veja, a seguir, como fica o código das classes-filha utilizando o mecanismo de herança. Nestes exemplos, vemos a ocorrência do fenômeno chamado sobreescrita (*overriding*), que acontece quando modificamos o comportamento de um método da classe-pai (referida pelo operador `parent`) na classe-filha, adicionado a ela uma nova funcionalidade. Assim, na construção da classe `ContaPoupanca` adicionamos a inicialização da propriedade `Aniversario` e, na classe `ContaCorrente`, adicionamos a inicialização da propriedade `Limite`. O método `Retirar()` também foi sobreescrito. Na classe `ContaPoupanca` é verificado se há saldo para realizar a transação e na classe `ContaCorrente` é verificado se a retirada está dentro do limite da conta, além de debitar CPMF.

 ContaPoupanca.class.php

```
<?php
class ContaPoupanca extends Conta
{
    var $Aniversario;

    /* método construtor (sobrescrito)
     * agora, também inicializa a variável $Aniversario
     */
    function __construct($Agencia, $Codigo, $DataDeCriacao,
                        $Titular, $Senha, $Saldo, $Aniversario)
    {
        // chamada do método construtor da classe-pai.
        parent::__construct($Agencia, $Codigo, $DataDeCriacao, $Titular, $Senha, $Saldo);
        $this->Aniversario = $Aniversario;
    }

    /* método Retirar (sobrescrito)
     * verifica se há saldo para retirar tal $quantia.
     */
    function Retirar($quantia)
    {
        if ($this->Saldo >= $quantia)
        {
            // Executa método da classe-pai.
            parent::Retirar($quantia);
        }
        else
        {
            echo "Retirada não permitida...\n";
            return false;
        }

        // retirada permitida
        return true;
    }
}
?>
```

 ContaCorrente.class.php

```
<?php
class ContaCorrente extends Conta
{
    var $Limite;
```

```
/* método construtor (sobrescrito)
 * agora, também inicializa a variável $Limite
 */
function __construct($Agencia, $Codigo, $DataDeCriacao, $Titular, $Senha, $Saldo, $Limite)
{
    // chamada do método construtor da classe-pai.
    parent::__construct($Agencia, $Codigo, $DataDeCriacao, $Titular, $Senha, $Saldo);
    $this->Limite = $Limite;
}

/* método Retirar (sobrescrito)
 * verifica se a $quantia retirada está dentro do limite.
 */
function Retirar($quantia)
{
    // imposto sobre movimentação financeira
    $cpmf = 0.05;

    if ( ($this->Saldo + $this->Limite) >= $quantia )
    {
        // Executa método da classe-pai.
        parent::Retirar($quantia);

        // Debita o Imposto
        parent::Retirar($quantia * $cpmf);
    }
    else
    {
        echo "Retirada não permitida...\n";
        return false;
    }

    // retirada permitida
    return true;
}
?>
```

2.6 Polimorfismo

O significado da palavra polimorfismo nos remete a “muitas formas”. Polimorfismo em orientação a objetos é o princípio que permite que classes derivadas de uma mesma superclasse tenham métodos iguais (com a mesma nomenclatura e parâmetros), mas comportamentos diferentes, redefinidos em cada uma das classes-filha.

Veja que as classes `ContaPoupanca` e `ContaCorrente`, criadas anteriormente, possuem os mesmos métodos. Repare que o método `Retirar()` possui o mesmo nome, mas comportamento diferente em ambas. No caso da conta poupança, ele verifica somente se há saldo. Na conta corrente, verifica se a retirada está dentro do limite da operação, além de debitar o imposto correspondente (CPMF). Veja que o comportamento dessas operações é muito similar. Inicializamos duas contas com os mesmos valores e efetuamos os mesmos procedimentos sobre elas dentro de um laço de repetição. No exemplo a seguir, em vez de armazenar os objetos do tipo conta em variáveis, estamos formando um array de objetos: `$contas`. Veja os resultados obtidos.

poli.php

```
<?php
# carrega as classes
include_once 'classes/Pessoa.class.php';
include_once 'classes/Conta.class.php';
include_once 'classes/ContaPoupanca.class.php';
include_once 'classes/ContaCorrente.class.php';

# Criação do objeto $carlos
$carlos = new Pessoa(10, "Carlos da Silva", 1.85, 25, "10/04/1976", "Ensino Médio", 650.00);

echo "Manipulando o objeto {$carlos->Nome}: \n";

# Criação do objeto $conta_carlos
$contas[1] = new ContaCorrente(6677, "CC.1234.56", "10/07/02", $carlos, 9876, 500.00, 200.00);
$contas[2] = new ContaPoupanca(6678, "PP.1234.57", "10/07/02", $carlos, 9876, 500.00, '10/07');

// percorremos as contas
foreach ($contas as $key => $conta)
{
    echo "Manipulando a conta $key de: {$conta->Titular->Nome}: \n";
    echo "O saldo atual da conta $key é R\$ {$conta->ObterSaldo()} \n";
    $conta->Depositar(200);
    echo "O saldo atual da conta $key é R\$ {$conta->ObterSaldo()} \n";
    $conta->Retirar(100);
    echo "O saldo atual da conta $key é R\$ {$conta->ObterSaldo()} \n";
}
?>
```

Resultado:

```
Manipulando o objeto Carlos da Silva:
Manipulando a conta 1 de: Carlos da Silva:
O saldo atual da conta 1 é R$ 500
O saldo atual da conta 1 é R$ 700
O saldo atual da conta 1 é R$ 595
```

```
Manipulando a conta 2 de: Carlos da Silva:  
O saldo atual da conta 2 é R$ 500  
O saldo atual da conta 2 é R$ 700  
O saldo atual da conta 2 é R$ 600  
Objeto Carlos da Silva finalizado...  
Objeto Conta CC.1234.56 de Carlos da Silva finalizada...  
Objeto Conta PP.1234.57 de Carlos da Silva finalizada...
```

Observação: o PHP não suporta sobrecarga, ou métodos com o mesmo nome, mas assinaturas (parametrização) diferentes.

2.7 Abstração

No paradigma de orientação a objetos se prega o conceito da “abstração”. De acordo com o dicionário *Priberam*, “abstrair” é separar mentalmente, considerar isoladamente, simplificar, alhear-se. Para construir um sistema orientado a objetos, não devemos projetar o sistema como sendo uma grande peça monolítica; devemos separá-lo em partes, concentrando-nos nas peças mais importantes e ignorando os detalhes (em um primeiro momento), para podermos construir peças bem-definidas que possam ser reaproveitadas mais tarde, formando uma estrutura hierárquica.

2.7.1 Classes abstratas

Nesse contexto, encontraremos classes estruturais, ou seja, que estão na nossa hierarquia de classes para servirem de base para outras. São classes que nunca serão instanciadas na forma de objetos; somente suas filhas serão. Nestes casos, é interessante marcar essas classes como sendo classes abstratas, de modo que cada classe abstrata é tratada diferentemente pela linguagem de programação, a qual irá automaticamente impedir que se instanciem objetos a partir dela.

Seguindo os exemplos anteriores, uma pessoa pode ter uma `ContaCorrente` ou uma `ContaPoupança`, mas jamais poderá ter uma `Conta`. Isso porque `Conta` é uma estrutura abstrata, não definindo características próprias como taxas de retirada, limites, dentre outras especificidades que são escritas nas classes-filha. No exemplo a seguir, tentaremos instanciar um objeto da classe `Conta`, mesmo que ela seja abstrata, e obteremos a mensagem de erro dizendo que a classe abstrata `Conta` não pode ser instanciada.

Conta.class.php (complemento)

```
<?php  
abstract class Conta  
{  
    # ...
```

```
# conteúdo já escrito no exemplo anterior
#
}

?>
```

classe_abstrata.php

```
<?php
include_once 'classes/Conta.class.php';

$conta = new Conta;
?>
```

Resultado:

Fatal error: Cannot instantiate abstract class Conta in classe_abstrata.php on line 4

2.7.2 Classes finais

A classe final não pode ser uma superclasse, ou seja, não pode ser base em uma estrutura de herança. Se definirmos uma classe como final pelo operador FINAL, ela não poderá mais ser especializada. Em nosso exemplo, tornamos a classe `ContaPoupanca` uma classe final e, mesmo assim, tentamos especializá-la no que chamamos de `ContaPoupancaUniversitaria`. A mensagem de erro obtida diz que a classe `ContaPoupancaUniversitaria` não pode descender da classe final `ContaPoupanca`.

ContaPoupanca.class.php (complemento)

```
<?php
final class ContaPoupanca extends Conta
{
    #
    # Conteúdo já escrito no exemplo anterior
    #
}
```

classe_final.php

```
<?php
include_once 'classes/Conta.class.php';
include_once 'classes/ContaPoupanca.class.php';

class ContaPoupancaUniversitaria extends ContaPoupanca
{
    // ... sobrescrita de métodos
}
```

 **Resultado:**

```
Fatal error: Class ContaPoupancaUniversitaria may not inherit  
from final class (ContaPoupanca) in classe_final.php on line 5
```

2.7.3 Métodos abstratos

Um método abstrato consiste na definição de uma assinatura na classe abstrata. Este método deverá conter uma implementação na classe-filha, mas não deve possuir implementação na classe em que ele é definido. Em nosso exemplo, definiremos um método abstrato na classe `Conta`. Isso faz com que seja obrigatório a qualquer classe descendente da classe `Conta` (vide `ContaPoupanca` e `ContaCorrente`) ter em si a implementação deste método.

No exemplo que segue, definiremos o método `Transferir()` na classe `Conta` como sendo abstrato e tentaremos fazer uso das classes `Conta` e `ContaPoupanca`. Como esperado, o resultado que obtivemos foi um erro, indicando que a classe `ContaPoupanca` não possui a implementação do método `Transferir()`. Para sanar o erro, segue o complemento do código da classe `ContaPoupanca`. O complemento do código-fonte da classe `ContaCorrente` é apresentado no exemplo seguinte.

 **Conta.class.php (complemento)**

```
<?php  
abstract class Conta  
{  
    # ...  
    # conteúdo já escrito no exemplo anterior  
    # ...  
  
    abstract function Transferir($Conta, $Valor);  
}  
?>
```

 **abstrato.php**

```
<?php  
include_once 'classes/Pessoa.class.php';  
include_once 'classes/Conta.class.php';  
include_once 'classes/ContaPoupanca.class.php';  
  
$carlos = new Pessoa(10, "Carlos da Silva", 1.85, 25, 72, "Ensino Médio", 650.00);  
$conta = new ContaPoupanca(6677, "CC.1234.56", "10/07/02", $carlos, 9876, 500.00, '10/07');  
?>
```

 **Resultado:**

Fatal error: Class ContaPoupanca contains 1 abstract methods and must therefore be declared abstract (Conta::Transferir) in ContaPoupanca.class.php

 **ContaPoupanca.class.php (complemento)**

```
<?php
class ContaPoupanca extends Conta
{
    # ...
    # conteúdo já escrito no exemplo anterior
    # ...

    function Transferir($Conta, $Valor)
    {
        if ($this->Retirar($Valor))
        {
            $Conta->Depositar($Valor);
        }
    }
}
?>
```

2.7.4 Métodos finais

Um método final não pode ser sobreescrito, ou seja, não pode ser redefinido na classe-filha. Para marcar um método como final, basta utilizar o operador FINAL no início da sua declaração. No exemplo a seguir, declararemos o método `Transferir()` da classe `ContaCorrente` como sendo final e, ainda assim, tentaremos redefini-lo na classe-filha `ContaCorrenteEspecial`. O resultado é a mensagem de erro dizendo que não podemos sobreescriver o método `Transferir()` da classe `ContaCorrente`.

 **ContaCorrente.class.php (complemento)**

```
<?php
class ContaCorrente extends Conta
{
    var $TaxaTransferencia = 2.5;

    # ...
    # conteúdo já escrito no exemplo anterior
    # ...

    final function Transferir($Conta, $Valor)
    {
        if ($this->Retirar($Valor))
```

```
{  
    $Conta->Depositar($Valor);  
}  
  
if ($this->Titular != $Conta->Titular)  
{  
    $this->Retirar($this->TaxaTransferencia);  
}  
}  
}  
?>
```

metodo_final.php

```
<?php  
# carrega as classes  
include_once 'classes/Conta.class.php';  
include_once 'classes/ContaCorrente.class.php';  
  
class ContaCorrenteEspecial extends ContaCorrente  
{  
    function Depositar($Valor)  
    {  
        echo "sobrescrevendo método Depositar.\n";  
        parent::Depositar($Valor);  
    }  
  
    function Transferir($Conta, $Valor)  
    {  
        echo "sobrescrevendo método Transferir.\n";  
        parent::Transferir($Conta, $Valor);  
    }  
}  
?>
```

Resultado:

```
Fatal error: Cannot override final method ContaCorrente::Transferir()  
in metodo_final.php on line 5
```

2.8 Encapsulamento

Um dos recursos mais interessantes na orientação a objetos é o encapsulamento, um mecanismo que provê proteção de acesso aos membros internos de um objeto. Lembre-se que uma classe possui responsabilidade sobre os atributos que contém. Dessa forma, existem certas propriedades de uma classe que devem ser tratadas

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.

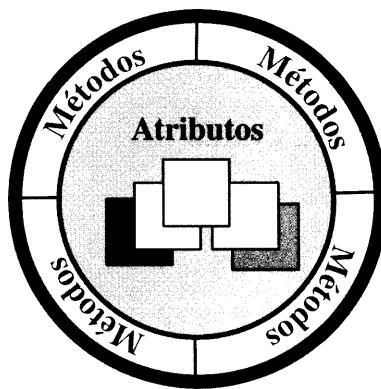


Figura 2.6 – Encapsulamento.

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
<code>private</code>	Membros declarados como <code>private</code> 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.
<code>protected</code>	Membros declarados como <code>protected</code> 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.
<code>public</code>	Membros declarados como <code>public</code> 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.

Até agora, nos exemplos anteriores, declaramos classes sem definir a visibilidade de propriedades e métodos. 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 ela será tratada como `public`.

2.8.1 Private

Para demonstrar a visibilidade `private`, criaremos a classe `Funcionario` e marcaremos a maioria das propriedades como `private`. Dessa forma, elas só poderão ser alteradas por métodos da mesma classe. A única propriedade que deixaremos para livre acesso será o nome, marcado como `public`.

Para demonstrar a necessidade de proteger o acesso a membros internos de uma classe, atribuiremos um valor inválido à propriedade `Salario`. Na Figura 2.7, temos a classe `Funcionario` no padrão UML.

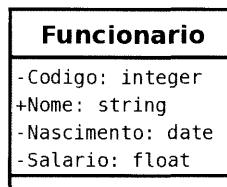


Figura 2.7 – Classe Funcionario.

Funcionario.class.php

```
<?php
class Funcionario
{
    private $Codigo;
    public $Nome;
    private $Nascimento;
    private $Salario;
}
```

```
?>
```

private.php

```
<?php
# carrega a classe
include_once 'classes/Funcionario.class.php';

$pedro = new Funcionario;
$pedro->Salario = 'Oitocentos e setenta e seis';
?>
```

 **Resultado:**

```
Fatal error: Cannot access private property Pessoa::$Salario in private.php on line 6
```

Veja que o PHP resulta em um erro de acesso à propriedade, como esperado, mas se não podemos alterar o valor da propriedade `Salario` dessa forma, como faremos? Simples! Criaremos métodos pertencentes à classe `Funcionario` para manipular essa propriedade. No exemplo a seguir criaremos o método `SetSalario()` para atribuir o valor da propriedade `Salario` e `GetSalario()` para retornar o valor. Aproveitamos para realizar uma validação no método `SetSalario()`, que somente atribuirá o valor de `Salario` caso este seja de um tipo numérico e positivo.

 **Funcionario.class.php (complemento)**

```
<?php
class Funcionario
{
    private $Codigo;
    public $Nome;
    private $Nascimento;
    private $Salario;

    /* método SetSalario
     * atribui o parâmetro $Salario à propriedade $Salario
     */
    function SetSalario($Salario)
    {
        // verifica se é numérico e positivo
        if (is_numeric($Salario) and ($Salario > 0))
        {
            $this->Salario = $Salario;
        }
    }

    /* método GetSalario
     * retorna o valor da propriedade $Salario
     */
    function GetSalario()
    {
        return $this->Salario;
    }
}
?>
```

private.php

```
<?php
# carrega a classe
include_once 'classes/Funcionario.class.php';

// instancia novo Funcionario
$pedro = new Funcionario;

// atribui novo Salário
$pedro->SetSalario(876);

// obtém o Salário
echo 'Salário : (R$) ' . $pedro->GetSalario();
?>
```

Resultado:

Salário : (R\$) 876

2.8.2 Protected

Para demonstrar a visibilidade `protected`, vamos especializar a classe `Funcionário`, criando a classe `Estagiario`. A única característica exclusiva de um estagiário é que o seu salário é acrescido de 12% de bônus. Para tanto, sobrescreveremos o método `GetSalario()` para que este retorne o salário já reajustado. Veja na Figura 2.8 o diagrama da classe `Estagiario`.

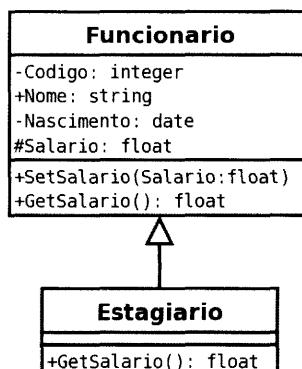


Figura 2.8 – Classe `Estagiario`.

Estagiario.class.php

```
<?php
class Estagiario extends Funcionario
```

```

{
    /* método GetSalario sobreescrito
     * retorna o $Salário com 12% de bônus.
     */
    function GetSalario()
    {
        return $this->Salario * 1.12;
    }
}
?>

```

protected.php

```

<?php
# carrega as classes
include 'classes/Funcionario.class.php';
include 'classes/Estagiario.class.php';

$pedrinho = new Estagiario;
$pedrinho->SetSalario(248);
echo 'O Salário do Pedrinho é R$: ' . $pedrinho->GetSalario() . "\n";
?>

```

Resultado:

O Salário do Pedrinho é R\$: 0

Como esperado, o programa não retornou o salário esperado. Isso ocorre por que a propriedade `Salario` é uma propriedade `private`, o que significa que ela somente pode ser acessada de dentro da classe em que ela foi declarada (`Funcionario`). Para permitir o acesso também nas classes-filha, alteraremos a classe `Funcionario` e marcaremos a propriedade `Salario` como `protected`.

Funcionario.class.php (complemento)

```

<?php
class Funcionario
{
    private $Codigo;
    public $Nome;
    private $Nascimento;
    protected $Salario;

    #
    # conteúdo já escrito no exemplo anterior
    #

}
?>

```

Agora, ao executarmos novamente nosso trecho de programa, veremos que o comportamento desejado é atingido, sendo a propriedade `Salario` passível de alterações tanto no escopo da classe `Funcionario` quanto no escopo da classe `Estagiario`.

 **protected.php**

```
<?php
# carrega as classes
include 'classes/Funcionario.class.php';
include 'classes/Estagiario.class.php';

$pedrinho = new Estagiario;
$pedrinho->SetSalario(248);
echo 'O Salário do Pedrinho é R$: ' . $pedrinho->GetSalario() . "\n";
?>
```

 **Resultado:**

O Salário do Pedrinho é R\$: 277.76

2.8.3 Public

Demonstrar a visibilidade `public` é uma tarefa simples, pois o comportamento-padrão do PHP é tratar uma propriedade como `public`, ou seja, se não especificarmos a visibilidade, automaticamente ela será pública. No exemplo que segue, estamos criando dois objetos e alterando suas propriedades à vontade, as quais poderiam ser alteradas por métodos internos e por classes descendentes também.

 **public.php**

```
<?php
# carrega as classes
include 'classes/Funcionario.class.php';
include 'classes/Estagiario.class.php';

// cria objeto Funcionario
$pedrinho = new Funcionario;
$pedrinho->nome = 'Pedrinho';

// cria objeto Estagiario
$mariana = new Estagiario;
$mariana->nome = 'Mariana';

// imprime propriedade nome
echo $pedrinho->nome;
echo $mariana->nome;
?>
```

 **Resultado:**

Pedrinho

Mariana

2.9 Membros da classe

Como visto anteriormente, a classe é uma estrutura-padrão para criação dos objetos. A classe permite que armazenemos valores nela de duas formas: constantes de classe e propriedades estáticas. Estes atributos são comuns a todos os objetos da mesma classe.

2.9.1 Constantes

No exemplo a seguir, veremos como se dá a declaração de uma constante pelo operador `const`, seu acesso de forma externa ao contexto da classe, pela sintaxe `NomeDaClasse::NomeDaConstante`, e dentro da classe, pela sintaxe `self::NomeDaConstante`. O operador `self` representa a própria classe.

 **constantes.php**

```
<?php
class Biblioteca
{
    const Nome = "GTK ";
}

class Aplicacao extends Biblioteca
{
    // declaração das constantes
    const Ambiente = "Gnome Desktop ";
    const Versao   = "3.8 ";

    /* método construtor
     * acessa as constantes internamente
     */
    function __construct($Nome)
    {
        echo parent::Nome . self::Ambiente . self::Versao . $Nome . "\n";
    }
}

// acessa as constantes externamente
echo Biblioteca::Nome . Aplicacao::Ambiente . Aplicacao::Versao . "\n";

new Aplicacao('Dia');
new Aplicacao('Gimp');
?>
```

 **Resultado:**

GTK Gnome Desktop 3.8
GTK Gnome Desktop 3.8 Dia
GTK Gnome Desktop 3.8 Gimp

2.9.2 Propriedades estáticas

Propriedades estáticas são atributos de uma classe; são dinâmicas como as propriedades de um objeto, mas estão relacionadas à classe. Como a classe é a estrutura comum a todos os objetos dela derivados, propriedades estáticas são compartilhadas entre todos os objetos de uma mesma classe.

 **proposta.php**

```
<?php
class Aplicacao
{
    static $Quantidade;

    /* método Construtor
     * incrementa a $Quantidade de Aplicações
     */
    function __construct($Nome)
    {
        // incrementa propriedade estática
        self::$Quantidade++;
        $i = self::$Quantidade ;
        echo "Nova Aplicação nr. $i: $Nome\n";
    }
}

# cria novos objetos
new Aplicacao('Dia');
new Aplicacao('Gimp');
new Aplicacao('Gnumeric');
new Aplicacao('Abiword');
new Aplicacao('Evolution');

echo 'Quantidade de Aplicações = ' . Aplicacao::$Quantidade . "\n";
?>
```

 **Resultado:**

Nova Aplicação nr. 1: Dia
Nova Aplicação nr. 2: Gimp
Nova Aplicação nr. 3: Gnumeric

Nova Aplicação nr. 4: Abiword
Nova Aplicação nr. 5: Evolution
Quantidade de Aplicações = 5

2.9.3 Métodos estáticos

Métodos estáticos podem ser invocados diretamente da classe, sem a necessidade de instanciar um objeto para isso. Eles não devem referenciar propriedades internas pelo operador `$this`, porque este operador é utilizado para referenciar instâncias da classe (objetos), mas não a própria classe; são limitados a chamarem outros métodos estáticos da classe ou utilizar apenas propriedades estáticas. Para executar um método estático, basta utilizar a sintaxe `NomeDaClasse::NomeDoMétodo()`.

No exemplo a seguir, temos um arquivo-texto `readme.txt` com informações sobre a aplicação. A classe `Aplicacao` possui o método `Sobre()`, o qual lê as informações desse arquivo e as exibe na tela. Como este procedimento não envolve nenhuma propriedade de objeto, o método pode ser declarado como estático por meio do operador `static`. Um método estático pode acessar ainda constantes de classe e propriedades estáticas da mesma classe (por meio do operador `self`) e da superclasse (por meio do operador `parent`).

readme.txt

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

metesta.php

```
<?php
class Aplicacao
{
    /* método Estático
     * lê o arquivo readme.txt
     */
    static function Sobre()
    {
        $fd = fopen('readme.txt', 'r');
        while ($linha = fgets($fd, 200))
        {
            echo $linha;
        }
    }
    echo "Informações sobre a aplicação:\n";
```

```
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.

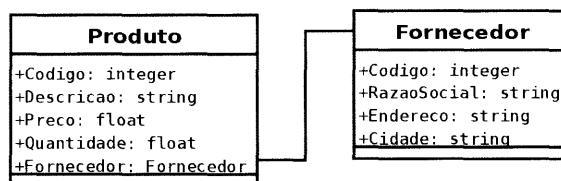


Figura 2.9 – Associação.

Fornecedor.class.php

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

 associacao.php

```
<?php
include_once 'classes/Fornecedor.class.php';
include_once 'classes/Produto.class.php';

// instancia Fornecedor
$fornecedor = new Fornecedor;
$fornecedor->Codigo      = 848;
$fornecedor->RazaoSocial= 'Bom Gosto Alimentos S.A.';
$fornecedor->Endereco    = 'Rua Ipiranga';
$fornecedor->Cidade      = 'Poços de Caldas';

// instancia Produto
$produto = new Produto;
$produto->Codigo      = 462;
$produto->Descricao   = 'Doce de Pêssego';
$produto->Preco       = 1.24;
$produto->Fornecedor = $fornecedor;

// imprime atributos
echo 'Código      : ' . $produto->Codigo . "\n";
echo 'Descrição   : ' . $produto->Descricao . "\n";
echo 'Código      : ' . $produto->Fornecedor->Codigo . "\n";
echo 'Razão Social: ' . $produto->Fornecedor->RazaoSocial . "\n";
?>
```

 **Resultado:**

```
Código      : 462
Descrição   : Doce de Pêssego
Código      : 848
Razão Social: Bom Gosto Alimentos S.A.
```

2.10.2 Agregação

Agregação é o tipo de relação entre objetos conhecida como todo/parte. Na agregação, um objeto agrupa outro objeto, ou seja, torna um objeto externo parte de si mesmo pela utilização de um dos seus métodos. Assim, o objeto-pai poderá utilizar funcionalidades do objeto agregado. Nesta relação, um objeto poderá agrregar uma ou muitas instâncias de um outro objeto. Para agregar muitas instâncias, a forma mais simples é utilizando arrays. Criamos um array como atributo da classe, sendo que o papel deste array é armazenar inúmeras instâncias de uma outra classe.

Para exemplificar esta relação utilizaremos o clássico exemplo de uma cesta de compras. A classe `Cesta` é o nosso todo e a classe `Produto` representa cada uma das partes. Uma `Cesta` é composta de inúmeros produtos (instâncias da classe `Produto`). Para agregar objetos do tipo `Produto` à `Cesta`, criamos o método `AdicionaItem()` na classe `Cesta`, que conta também com o método `ExibeLista()`, o qual, na verdade, chama o método `ImprimeEtiqueta()` de cada um dos produtos da `Cesta` e o método `CalculaTotal()`, que soma o preço de cada um dos produtos de uma `Cesta`. Na Figura 2.10, temos o relacionamento entre as classes.

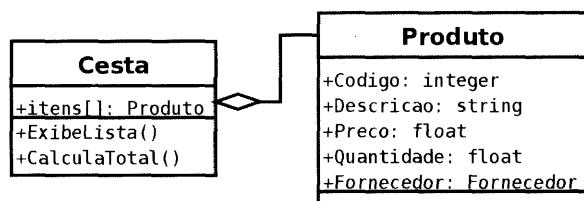


Figura 2.10 – Agregação.

Cesta.class.php

```

<?php
class Cesta
{
    private $itens;

    # adiciona Itens na cesta
    function AdicionaItem(Produto $item)
    {
        $this->itens[] = $item;
    }

    # exibe a lista de produtos
    function ExibeLista()
    {
        foreach ($this->itens as $item)
        {
            $item->ImprimeEtiqueta();
        }
    }

    # calcula o valor total da cesta
    function CalculaTotal()
    {
        foreach ($this->itens as $item)
        {
            $total += $item->Preco;
        }
    }
}
  
```

```
    }
    return 'R$ ' . $total;
}
?>
```

agregacao.php

```
<?php
include_once 'classes/Cesta.class.php';
include_once 'classes/Produto.class.php';

$produto1 = new Produto;
$produto2 = new Produto;
$produto3 = new Produto;
$produto4 = new Produto;

$produto1->Codigo      = 1;
$produto1->Descricao   = 'Ameixa';
$produto1->Preco       = 1.40;

$produto2->Codigo      = 2;
$produto2->Descricao   = 'Morango';
$produto2->Preco       = 2.24;

$produto3->Codigo      = 3;
$produto3->Descricao   = 'Abacaxi';
$produto3->Preco       = 2.86;

$produto4->Codigo      = 4;
$produto4->Descricao   = 'Laranja';
$produto4->Preco       = 1.14;

$cesta = new Cesta;
$cesta->AdicionaItem($produto1);
$cesta->AdicionaItem($produto2);
$cesta->AdicionaItem($produto3);
$cesta->AdicionaItem($produto4);

echo $cesta->CalculaTotal();
echo "\n"; // quebra de linha
echo $cesta->ExibeLista();
?>
```

 **Resultado:**

```
R$ 7,64
Código: 1
Descrição: Ameixa
Código: 2
Descrição: Morango
Código: 3
Descrição: Abacaxi
Código: 4
Descrição: Laranja
```

Talvez você já tenha percebido um possível problema neste tipo de abordagem. O que aconteceria se adicionássemos objetos que não são da classe `Produto` a uma `Cesta`? Se chamássemos métodos como `CalculaTotal()` e `ExibeLista()`, por exemplo, teríamos problemas, uma vez que eles confiam na existência do atributo `Preco` e do método `ImprimeEtiqueta()` da classe `produto`.

O PHP5 introduz o conceito de TypeHinting, ou seja, “sugestão de tipo”. Nesse exemplo da classe `cesta`, o método `AdicionaItem()` recebe um `produto` chamado `$item`. Veja que na frente do parâmetro indicamos a classe à qual ele deve pertencer. Caso o método `AdicionaItem()` seja executado passando um parâmetro que não é do tipo `Produto`, um erro fatal será lançado. Veja no exemplo a seguir:

```
function AdicionaItem(Produto $item)

 agregacao2.php

<?php
include_once 'classes/Cesta.class.php';
include_once 'classes/Fornecedor.class.php';
include_once 'classes/Produto.class.php';

$fornecedor = new Fornecedor;
$fornecedor->RazaoSocial = 'Produtos Bom Gosto S.A.';

$cesta = new Cesta;
$cesta->AdicionaItem($fornecedor);

$cesta->CalculaTotal();
?>
```

 **Resultado:**

Fatal error: Argument 1 must be an instance of `Produto` in `Cesta.class.php` on line 7

2.10.3 Composição

Composição também é uma relação que demonstra uma relação todo/parte. A diferença em relação à agregação é que, na composição, o objeto-pai ou “todo” é responsável pela criação e destruição de suas partes. O objeto-pai realmente “possui” a(s) instância(s) de suas partes. Diferentemente da agregação, na qual as instâncias do “todo” e das “partes” são independentes.

Na agregação, ao destruirmos o objeto “todo”, as “partes” permanecem na memória por terem sido criadas fora do escopo da classe “todo”. Já na composição, quando o objeto “todo” é destruído, suas “partes” também são, justamente por terem sido criadas pelo objeto “todo”. Veja no diagrama a seguir uma composição.

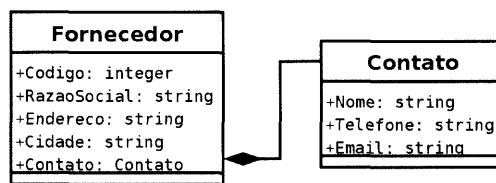


Figura 2.11 – Composição.

Contato.class.php

```

<?php
class Contato
{
    var $Nome;
    var $Telefone;
    var $Email;

    # grava informações de contato
    function SetContato($Nome, $Telefone, $Email)
    {
        $this->Nome = $Nome;
        $this->Telefone = $Telefone;
        $this->Email = $Email;
    }

    # obtém informações de contato
    function GetContato()
    {
        return "Nome: {$this->Nome}, Telefone: {$this->Telefone}, Email: {$this->Email}";
    }
}
?>
  
```

 Fornecedor.class.php

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

    # método construtor
    function __construct()
    {
        // instancia novo Contato
        $this->Contato = new Contato;
    }

    # grava contato
    function SetContato($Nome, $Telefone, $Email)
    {
        // delega chamada de método
        $this->Contato->SetContato($Nome, $Telefone, $Email);
    }

    # retorna contato
    function GetContato()
    {
        // delega chamada de método
        return $this->Contato->GetContato();
    }
}
?>
```

Neste exemplo, demonstramos a utilização de composição. Instanciamos um objeto da classe `Fornecedor`. A classe `Fornecedor` é composta de um contato, instanciado automaticamente no método construtor do `Fornecedor`. Veja que a classe `Fornecedor` é responsável por esta instância da classe `Contato`. Veja também que a chamada de métodos na classe principal é delegada à classe `Contato`.

 composicao.php

```
<?php
include_once 'classes/Fornecedor.class.php';
include_once 'classes/Contato.class.php';
```

```
# instancia novo fornecedor
$fornecedor = new Fornecedor;
$fornecedor->RazaoSocial = 'Produtos Bom Gosto S.A.';

# atribui informações de contato
$fornecedor->SetContato('Mauro', '51 1234-5678', 'mauro@bomgosto.com.br');

# imprime informações
echo $fornecedor->RazaoSocial . "\n";
echo "Informações de Contato\n";
echo $fornecedor->GetContato();
?>
```

Resultado:

```
Produtos Bom Gosto S.A.
Informações de Contato
Nome: Mauro, Telefone: 51 1234-5678, Email: mauro@bomgosto.com.br
```

2.11 Intercepções

O PHP5 introduziu o conceito de interceptação em operações realizadas por objetos por meio dos métodos `__set()`, `__get()` e `__call()`, vistos a seguir.

2.11.1 Método `__set()`

O método `__set()` intercepta a atribuição de valores a propriedades do objeto. Sempre que for atribuído um valor a uma propriedade do objeto, automaticamente esta atribuição passa pelo método `__set()`, o qual recebe o nome da propriedade e o valor a ser atribuído, podendo atribuí-lo ou não.

Cachorro.class.php

```
<?php
class Cachorro
{
    private $Nascimento;

    # método construtor
    function __construct($nome)
    {
        $this->nome = $nome;
    }
}
```

```
# intercepta atribuição
function __set($propriedade, $valor)
{
    if ($propriedade == 'Nascimento')
    {
        # verifica se valor é dividido em
        # 3 partes separadas por '/'
        if (count(explode('/', $valor))==3)
        {
            echo "Dado '$valor', atribuido à '$propriedade'\n";
            $this->$propriedade = $valor;
        }
        else
        {
            echo "Dado '$valor', não atribuído à '$propriedade'\n";
        }
    }
    else
    {
        $this->$propriedade = $valor;
    }
}
?>
```

No próximo exemplo, instanciaremos um objeto `$toto` da classe `Cachorro`. A data de `Nascimento` do objeto será atribuída duas vezes. Note que, nas duas vezes, o método `__set()` irá interceptar a atribuição, validando-a. Somente a segunda atribuição passará pelas regras de atribuição definidas dentro do método `__set()`. A segunda data é válida porque é formada por três partes separadas por “/”.

set.php

```
<?php
#inclui classe cachorro
include_once 'classes/Cachorro.class.php';

$toto = new Cachorro('Totó');
$toto->Nascimento = '3 de março'; // atribuição inválida
$toto->Nascimento = '10/04/2005'; // atribuição correta
?>
```

Resultado:

```
Dado '3 de março', não atribuído à 'Nascimento'
Dado '10/04/2005', atribuído à 'Nascimento'
```

2.11.2 Método __get()

O método `__get()` intercepta requisições de propriedades do objeto. Sempre que for requisitada uma propriedade, automaticamente essa requisição passará pelo método `__get()`, o qual recebe o nome da propriedade requisitada, podendo retorná-la ou não.

Produto.class.php

```
<?php
class Produto
{
    var $Codigo;
    var $Descricao;
    var $Quantidade;
    private $Preco;
    const MARGEM = 10;
    # método construtor de um Produto
    function __construct($Codigo, $Descricao, $Quantidade, $Preco)
    {
        $this->Codigo = $Codigo;
        $this->Descricao = $Descricao;
        $this->Quantidade= $Quantidade;
        $this->Preco = $Preco;
    }
    # intercepta a obtenção de propriedades
    function __get($propriedade)
    {
        echo "Obtendo o valor de '$propriedade' :\n";
        if ($propriedade == 'Preco')
        {
            return $this->$propriedade * (1 + (self::MARGEM / 100));
        }
    }
}
?>
```

O método `__get()` pode ser utilizado para retornar valores calculados. Neste caso, por exemplo, ao solicitar o preço de um produto, podemos retorná-lo já com seu valor ajustado (com a margem de lucro).

get.php

```
<?php
#inclui classe Produto
include_once 'classes/Produto.class.php';
#cria novo produto com o preço R$ 345.67
$produto = new Produto(1, 'Pendrive 512Mb', 1, 345.67);
```

```
#imprime o preço
echo $produto->Preco;
?>
```

 **Resultado:**

Obtendo o valor de 'Preco' :
380,237

2.11.3 Método __call()

O método `__call()` intercepta a chamada a métodos. Sempre que for executado um método que não existir no objeto, automaticamente a execução será direcionada para ele, que recebe dois parâmetros, o nome do método requisitado e o parâmetro recebido, podendo decidir o procedimento a realizar.

 **Produto.class.php**

```
<?php
class Produto
{
    var $Codigo;
    var $Descricao;
    var $Quantidade;
    private $Preco;
    const MARGEM = 10;

    # método construtor de um Produto
    function __construct($Codigo, $Descricao, $Quantidade, $Preco)
    {
        $this->Codigo      = $Codigo;
        $this->Descricao   = $Descricao;
        $this->Quantidade = $Quantidade;
        $this->Preco       = $Preco;
    }

    # intercepta a chamada à métodos
    function __call($metodo, $parametros)
    {
        echo "Você executou o método: {$metodo}\n";
        foreach ($parametros as $key => $parametro)
        {
            echo "\tParâmetro $key: $parametro\n";
        }
    }
?>
```

Neste exemplo, iremos instanciar um objeto da classe `Produto`. Tentaremos executar o método `Vender()`, sendo que este método não existe. Note que esta execução é interceptada pelo método `__call()`, que emite uma mensagem na tela, informando qual foi o método executado e quais parâmetros foram passados.

call.php

```
<?php  
# inclui classe Produto  
include_once 'classes/Produto.class.php';  
  
# criando novo produto com o preço R$ 345.67  
$produto = new Produto(1, 'Pendrive 512Mb', 1, 345.67);  
  
# executando método Vender, passando 10 unidades.  
echo $produto->Vender(10);  
?>
```

Resultado:

```
Você executou o método: Vender  
Parâmetro 0: 10
```

2.11.4 Método `__toString()`

Quando imprimimos objetos na tela, por meio de comandos como o `echo` e `print`, o PHP exibe no console o identificador interno do objeto, como nos exemplos a seguir:

```
Object id #1  
Object id #2
```

Para alterar esse comportamento, podemos definir o método `__toString()` para cada classe. Caso o método `__toString()` exista, no momento em que mandarmos exibir um objeto no console, o PHP irá imprimir o retorno dessa função.

No exemplo a seguir, o método `__toString()` retorna a propriedade `$nome`, que é definida no método construtor do objeto. Nesses casos, instanciaremos dois objetos: `$toto` e `$vava`. Note que no momento em que imprimimos esses objetos no console, será retornado o próprio nome dos mesmos.

tostring.php

```
<?php  
class Cachorro  
{  
    private $Nascimento;
```

```
# método construtor
function __construct($nome)
{
    $this->nome = $nome;
}

# tostring, executado sempre que o objeto for impresso
function __tostring()
{
    return $this->nome;
}
}

$toto = new Cachorro('Totó');
$vava = new Cachorro('Daba');
echo $toto;
echo "\n";
echo $vava;
echo "\n";
?>
```

Resultado:

```
Totó
Daba
```

2.11.5 Método `toXml()`?

O PHP não possui método para retornar o objeto em formato XML, mas isso não é problema, uma vez que é muito fácil desenvolver tal funcionalidade. No exemplo a seguir, a classe `Cachorro` possui um método `toXml()`, que retorna os dados do objeto em forma de uma string XML.

`toxml.php`

```
<?php
class Cachorro
{
    # método construtor
    function __construct($nome, $idade, $raca)
    {
        $this->nome = $nome;
        $this->idade = $idade;
        $this->raca = $raca;
    }
}
```

```
# toXml, retorna o objeto no formato XML
function toXml()
{
    return
<<<XML
<cachorro>
    <nome> {$this->nome} </nome>
    <idade> {$this->idade} </idade>
    <raca> {$this->raca} </raca>
</cachorro>

XML;
}
}

$toto = new Cachorro('Totó', 10, 'Fox Terrier');
$ava = new Cachorro('Daba', 8, 'Dálmata');
echo $toto->toXml();
echo $ava->toXml();
?>
```

Resultado:

```
<cachorro>
    <nome> Totó </nome>
    <idade> 10 </idade>
    <raca> Fox Terrier </raca>
</cachorro>
<cachorro>
    <nome> Daba </nome>
    <idade> 8 </idade>
    <raca> Dálmata </raca>
</cachorro>
```

Neste momento você deve estar imaginando que é muito trabalhoso escrever um método `toXml()` para cada classe do sistema. Para facilitar ainda mais, podemos criar uma classe genérica, que, neste exemplo, chamamos de `XMLBase`, a qual possui um método `toXml()` que a princípio gera um documento XML para qualquer classe que estender `XMLBase`. Isto porque esta classe se utiliza de funções como `get_class()` (retorna a classe de um objeto) e `get_object_vars()` (retorna as propriedades de um objeto), vistas na seção de funções para manipulação de objetos.

Veja que para a classe `Cachorro` possuir tal funcionalidade, basta estender a classe `XMLBase` pelo mecanismo de herança.

 XMLBase.class.php

```
<?php
class XMLBase
{
    function toXml()
    {
        $retorno = '<' . get_class($this) . '>' . "\n";
        $propriedades = get_object_vars($this);
        foreach ($propriedades as $propriedade => $valor)
        {
            $retorno .= "\t<$propriedade> $valor </$propriedade>\n";
        }
        $retorno .= '</' . get_class($this) . '>' . "\n";
        return $retorno;
    }
}
?>
```

 XMLBase.php

```
<?
# include classe XMLBase
include_once 'classes/XMLBase.class.php';
class Cachorro extends XMLBase
{
    # método construtor
    function __construct($nome, $idade, $raca)
    {
        $this->nome = $nome;
        $this->idade = $idade;
        $this->raca = $raca;
    }
}
$toto = new Cachorro('Totó', 10, 'Fox Terrier');
$vava = new Cachorro('Daba', 8, 'Dálmata');
echo $toto->toXml();
echo $vava->toXml();
?>
```

 Resultado:

```
<Cachorro>
    <nome> Totó </nome>
    <idade> 10 </idade>
    <raca> Fox Terrier </raca>
</Cachorro>
```

```
<Cachorro>
    <nome> Daba </nome>
    <idade> 8 </idade>
    <raca> Dálmatas </raca>
</Cachorro>
```

2.12 Interfaces

A programação orientada a objetos baseia-se fortemente na interação de classes e objetos. Um objeto deve conhecer quais são as funcionalidades que um outro objeto pode lhe fornecer (métodos). Na etapa de projeto do sistema, podemos definir conjuntos de métodos que determinadas classes do nosso sistema deverão implementar incondicionalmente. Tais conjuntos de métodos são as interfaces, as quais contêm a declaração de métodos de forma prototipada, sem qualquer implementação. Toda classe que implementar uma interface deverá obrigatoriamente possuir os métodos predefinidos na interface; caso contrário, resultará em erro.

No exemplo a seguir, criaremos a classe `Aluno`, mas o projeto do sistema indica que esta classe deve implementar um conjunto de métodos que devem estar disponíveis para os outros objetos do sistema. Para tanto, criamos a interface `IAluno` contendo os métodos `getNome()`, `setNome()` e `setResponsavel()`. Neste exemplo, a classe `Aluno` não está implementando o método `setResponsavel()` (que recebe um parâmetro da classe `Pessoa`), dessa forma, o programa irá retornar o erro indicado no final.

Observação: uma classe pode implementar diversas interfaces separadas por vírgula.

interface.php

```
<?php
# interface Aluno
interface IAluno
{
    function getNome();
    function setNome($nome);
    function setResponsavel(Pessoa $responsavel);
}

# Classe Aluno
class Aluno implements IAluno
{
    # atribui o nome do aluno
    function setNome($nome)
{
```

```
    $this->nome = $nome;  
}  
  
# retorna o nome do aluno  
function getName()  
{  
    return $this->nome;  
}  
}  
  
# instancia novo Aluno  
$joaninha = new Aluno;  
  
# chama métodos quaisquer  
$joaninha->setNome('Joana Maranhao');  
echo $joaninha->getName();  
?>
```

Resultado:

```
Fatal error: Class Aluno contains 1 abstract methods and must therefore be declared  
abstract (IAluno::setResponsavel) in interface.php on line 24
```

2.13 Método `__clone()`

O comportamento-padrão do PHP quando atribuímos um objeto ao outro é criar uma referência entre os objetos. Dessa forma, teremos duas variáveis apontando para a mesma região da memória. Este é o comportamento desejado na maioria das vezes, mas como proceder quando quisermos de fato duplicar um objeto na memória? Simples! Utilizaremos o operador `clone`. O objeto resultante será idêntico ao original, a menos que tenhamos declarado o método `__clone()`, responsável por definir o comportamento da ação de clonagem, atuando diretamente nas propriedades do objeto resultante dessa ação.

No exemplo, criaremos um objeto `$toto` da classe `Cachorro` e, em seguida, iremos cloná-lo. Neste exemplo, o método `__clone()` executado no momento da clonagem define que o código da coleira do `Cachorro` deverá ser incrementado, sua `idade` será zerada e seu `nome` será acrescido de 'Júnior' ao final.

clone.php

```
<?  
class Cachorro  
{  
    # método construtor
```

```
function __construct($coleira, $nome, $idade, $raca)
{
    $this->coleira = $coleira;
    $this->nome = $nome;
    $this->idade = $idade;
    $this->raca = $raca;
}

function __clone()
{
    $this->coleira = $this->coleira +1;
    $this->nome .= ' Junior';
    $this->idade = 0;
}
}

$toto = new Cachorro(100, 'Totó', 10, 'Fox Terrier');
$vava = clone $toto;

echo 'Código: ' . $toto->coleira . "\n";
echo 'Nome: ' . $toto->nome . "\n";
echo 'Idade: ' . $toto->idade . " anos \n";

echo "\n";
echo 'Código: ' . $vava->coleira . "\n";
echo 'Nome: ' . $vava->nome . "\n";
echo 'Idade: ' . $vava->idade . " anos \n";
?>
```

Resultado:

```
Código: 100
Nome: Totó
Idade: 10 anos
```

```
Código: 101
Nome: Totó Junior
Idade: 0 anos
```

2.14 Autoload

Sempre que se instancia um objeto em PHP, é necessário ter a declaração da classe na memória; caso contrário, a aplicação é finalizada com um erro. Para introduzir uma classe na memória podemos utilizar comandos como `include_once`. Podemos realizar tal operação no início da aplicação introduzindo todas as classes que poderão ser

necessárias durante a execução da aplicação. Outra forma seria introduzir a classe na linha imediatamente anterior à instância do objeto.

Para simplificar tal procedimento, o PHP introduz a função `__autoload()`, ou “carga automática”. Com ela, a carga da classe é realizada de forma dinâmica sempre que um objeto for instanciado. Essa função recebe o nome da classe que está para ser instanciada, mas, antes que isso aconteça, ela tem de introduzir sua classe na memória. No exemplo a seguir estamos lendo a classe no diretório `classes/` e adicionando `.class.php`, que é a extensão do arquivo. Perceba o correto funcionamento do programa.

autoload.php

```
<?php
# função de carga autoática
function __autoload($classe)
{
    # busca classe no diretório de classes...
    include_once "classes/{$classe}.class.php";
}

# instanciando novo Produto
$bolo = new Produto(500, 'Bolo de Fubá', 4, 4.12);
echo 'Código: ' . $bolo->Codigo . "\n";
echo 'Nome: ' . $bolo->Descricao . "\n";
?>
```

Resultado:

Código: 500
Nome: Bolo de Fubá

2.15 Objetos dinâmicos

O PHP nos oferece diversas facilidades para manipulação de objetos. Uma delas é a possibilidade de criar objetos dinamicamente, sem ter a classe previamente definida. Naturalmente, esses objetos irão armazenar somente dados, e não funções.

Neste primeiro exemplo, criaremos dois objetos, `$william` e `$daline`, com seus respectivos atributos. Note que esses objetos são da classe `stdClass` (Standard Class), classe vazia criada especialmente para esses propósitos.

dinamico.php

```
<?php
// cria objeto william
```

```

$william->nome      = 'William Scatola';
$william->idade     = 22;
$william->profissao = 'Controle de Estoque';

// cria objeto daline
$daline->nome      = 'Daline DallOglio';
$daline->idade     = 21;
$daline->profissao = 'Almoxarifado';

print_r($william);
print_r($daline);
?>

```

Resultado:

```

stdClass Object
(
    [nome] => William Scatola
    [idade] => 22
    [profissao] => Controle de Estoque
)
stdClass Object
(
    [nome] => Daline DallOglio
    [idade] => 21
    [profissao] => Almoxarifado
)

```

Outra possibilidade que o PHP nos oferece é utilizar variáveis variantes para declarar as propriedades de um objeto.

objarray.php

```

<?php
// cria array dados_william
$dados_william['nome']      = 'William Scatola';
$dados_william['idade']     = 22;
$dados_william['profissao'] = 'Controle de Estoque';

// cria array dados_daline
$dados_daline['nome']      = 'Daline DallOglio';
$dados_daline['idade']     = 21;
$dados_daline['profissao'] = 'Almoxarifado';

// cria objeto william
foreach ($dados_william as $chave => $valor)
{

```

```
// utiliza variáveis variantes
$william->$chave = $valor;
}

// cria objeto daline
foreach ($dados_daline as $chave => $valor)
{
    // utiliza variáveis variantes
    $daline->$chave = $valor;
}

echo "{$william ->nome} é {$william->profissao}\n";
echo "{$daline->nome} é {$daline->profissao}\n";
?>
```

Resultado:

William Scatola é Controle de Estoque
Daline Dall'Oglio é Almoxarifado

2.16 Manipulação de XML

Nesta seção abordaremos um conjunto de funções chamadas de SimpleXML, disponíveis a partir do PHP5. Seu objetivo, como diz seu nome, é tornar simples a tarefa de carregar, interpretar e alterar documentos XML. É de suma importância a manipulação de arquivos XML no desenvolvimento de aplicações, seja para o armazenamento de arquivos de configuração, de dados reais ou mesmo para o intercâmbio de informações entre diferentes aplicações. A seguir, veremos alguns exemplos práticos sobre carga e alteração de documentos XML utilizando a extensão SimpleXML.

2.16.1 Exemplos

Primeiramente vamos realizar uma simples leitura de um documento XML e analisar seu retorno. Neste primeiro exemplo, tomamos o arquivo XML a seguir como base. No pequeno programa que segue, iremos interpretar o documento XML com a função `simplexml_load_file()` e analisar o objeto resultante da sua criação com a função `var_dump()`. Vemos claramente cada tag do XML sendo transformada em propriedades do objeto resultante.

Observação: a função `simplexml_load_file()` realiza a leitura de um documento XML, criando um objeto do tipo `SimpleXMLElement` a partir dessa operação. Caso o documento seja mal formatado, ou mesmo não seja um documento XML, essa função retornará FALSE.

 paises.xml

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<pais>
    <nome> Brasil </nome>
    <idioma> portugues </idioma>
    <religiao> catolica </religiao>
    <moeda> Real (R$) </moeda>
    <populacao> 190 milhoes </populacao>
</pais>
```

 exemplo1.php

```
<?php
// interpreta o documento XML
$xml = simplexml_load_file('paises.xml');

// exibe as informações do objeto criado
var_dump($xml);
?>
```

 Resultado:

```
object(SimpleXMLElement)#1 (5) {
    ["nome"]=> string(8) " Brasil "
    ["idioma"]=> string(11) " portugues "
    ["religiao"]=> string(10) " catolica "
    ["moeda"]=> string(11) " Real (R$) "
    ["populacao"]=> string(13) " 190 milhoes "
}
```

Um objeto `SimpleXMLElement` representa um elemento de um documento XML, o qual, quando interpretado pelas funções `simplexml_load_file()` ou `simplexml_load_string()`, resulta em um objeto do tipo `SimpleXMLElement`, contendo seus atributos e valores, bem como outros objetos `SimpleXMLElement` internos, representando subelementos (nodos). A seguir, estão alguns dos métodos providos por um objeto do tipo `SimpleXMLElement`:

Método	Descrição
<code>asXML()</code>	Retorna uma string XML formatada representando o objeto, bem como seus subelementos.
<code>attributes()</code>	Lista os atributos definidos dentro da tag XML do objeto.
<code>children()</code>	Retorna os elementos-filho do objeto (subnodos), bem como seus valores.
<code>addChild(string nome, string valor, string namespace)</code>	Adiciona um elemento ao nodo especificado e retorna um objeto do tipo <code>SimpleXMLElement</code> .

Neste segundo exemplo, demonstraremos como acessar diretamente as propriedades do objeto `SimpleXMLElement` resultante da leitura do documento XML. Para tanto, utilizamos o mesmo arquivo XML do exemplo anterior, imprimindo na tela as propriedades do objeto resultante com seu respectivo valor.

exemplo2.php

```
<?php
// interpreta o documento XML
$xml = simplexml_load_file('paises.xml');

// imprime os atributos do objeto criado
echo 'Nome : ' . $xml->nome . "\n";
echo 'Idioma : ' . $xml->idioma . "\n";
echo 'Religiao : ' . $xml->religiao . "\n";
echo 'Moeda : ' . $xml->moeda . "\n";
echo 'População : ' . $xml->populacao . "\n";
?>
```

Resultado:

```
Nome : Brasil
Idioma : portugues
Religiao : catolica
Moeda : Real (R$)
População : 190 milhoes
```

Você já viu como acessar diretamente as propriedades do XML sabendo os seus nomes. Neste terceiro exemplo, estaremos percorrendo o mesmo documento XML e imprimindo na tela suas propriedades (tags), mesmo sem saber os seus nomes. Isso é possível por meio da utilização do método `children()`, que atua sobre um objeto `SimpleXMLElement` e retorna todos os seus elementos-filho na forma de um array contendo a chave e o valor, que pode ser iterado por um laço `FOREACH`.

exemplo3.php

```
<?php
// interpreta o documento XML
$xml = simplexml_load_file('paises.xml');

foreach ($xml->children() as $elemento => $valor)
{
    echo "$elemento -> $valor\n";
}
?>
```

Resultado:

```
nome -> Brasil
idioma -> portugues
religiao -> catolica
moeda -> Real (R$)
populacao -> 190 milhoes
```

paises2.xml

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<pais>
    <nome> Brasil </nome>
    <idioma> português </idioma>
    <capital> Brasília </capital>
    <religiao> católica </religiao>
    <moeda> Real (R$) </moeda>
    <populacao> 190 milhoes </populacao>
    <geografia>
        <clima> tropical </clima>
        <costa> 7367 km </costa>
        <pico> Neblina (3014 m) </pico>
    </geografia>
</pais>
```

Aperfeiçoamos um pouco nosso documento XML, adicionando uma seção geografia, agrupando informações como clima, costa e pico. Neste caso, o objeto resultante (`$xml`) irá conter a propriedade `geografia`, que é também um objeto do tipo `SimpleXMLElement` com suas respectivas propriedades, sendo passíveis da utilização dos mesmos métodos listados no início desta seção (`asXML()`, `children()`, `attributes()` etc.).

exemplo4.php

```
<?php
// interpreta o documento XML
$xml = simplexml_load_file('paises2.xml');

echo 'Nome : ' . $xml->nome . "\n";
echo 'Idioma : ' . $xml->idioma . "\n";

echo "\n";
echo "*** Informações Geográficas ***\n";
echo 'Clima : ' . $xml->geografia->clima . "\n";
echo 'Costa : ' . $xml->geografia->costa . "\n";
echo 'Pico : ' . $xml->geografia->pico . "\n";
?>
```

 **Resultado:**

Nome : Brasil
Idioma : português

*** Informações Geográficas ***

Clima : tropical
Costa : 7367 km
Pico : Neblina (3014 m)

Você já aprendeu a acessar o XML de diversas formas, agora veremos como alterar o seu conteúdo. Neste caso, após a carga do documento, atribuímos novos valores acessando diretamente cada uma das propriedades que se deseja alterar no objeto. Também adicionamos um novo nodo, chamado <presidente>, pelo método `addChild()`. Após as atribuições de novos valores, utilizamos o método `asXML()` para retornar o novo documento XML formatado e atualizado. Se quisermos sobrepor o arquivo original, basta utilizar a função `file_put_contents()`.

 **exemplo5.php**

```
<?php
// interpreta o documento XML
$xml = simplexml_load_file('paises2.xml');

// alteração de propriedades
$xml->populacao = '220 milhoes';
$xml->religiao = 'cristianismo';
$xml->geografia->clima = 'temperado';
// adiciona novo nodo
$xml->addChild('presidente', 'Chapolin Colorado');

// exibindo o novo XML
echo $xml->asXML();
// grava no arquivo paises2.xml
file_put_contents('paises2.xml', $xml->asXML());
?>
```

 **Resultado:**

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<pais>
    <nome> Brasil </nome>
    <idioma> português </idioma>
    <capital> Brasília </capital>
    <religiao>cristianismo</religiao>
    <moeda> Real (R$) </moeda>
```

```
<populacao>220 milhoes</populacao>
<geografia>
    <clima>temperado</clima>
    <costa> 7367 km </costa>
    <pico> Neblina (3014 m) </pico>
</geografia>
<presidente>Chapolin Colorado</presidente>
</pais>
```

países3.xml

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<pais>
    <nome> Brasil </nome>
    <idioma> português </idioma>
    <religiao> católica </religiao>
    <moeda> Real (R$) </moeda>
    <populacao> 190 milhoes </populacao>
    <geografia>
        <clima> tropical </clima>
        <costa> 7367 km </costa>
        <pico> Neblina (3014 m) </pico>
    </geografia>
    <estados>
        <nome> Rio Grande do Sul </nome>
        <nome> São Paulo </nome>
        <nome> Minas Gerais </nome>
        <nome> Rio de Janeiro </nome>
        <nome> Paraná </nome>
        <nome> Mato Grosso </nome>
    </estados>
</pais>
```

Agora que você já aprendeu a alterar o conteúdo do XML, vejamos como se dá o acesso a elementos repetitivos. Neste exemplo, adicionamos a seção `<estados>`, a qual contém uma listagem com nomes de estados. O código a seguir imprime na tela algumas informações do Documento XML, como nos exemplos anteriores, e também mostra como exibir esses elementos repetitivos pelo laço de repetições `foreach()`.

exemplo6.php

```
<?php
// interpreta o documento XML
$xml = simplexml_load_file('países3.xml');

echo 'Nome : ' . $xml->nome . "\n";
echo 'Idioma : ' . $xml->idioma . "\n";
```

```
echo "\n";
echo "*** Estados ***\n";

/* Você pode acessar um estado diretamente pelo seu índice
   echo $xml->estados->nome[0];
*/
foreach ($xml->estados->nome as $estado)
{
    echo 'Estado : ' . $estado . "\n";
}
?>
```

Resultado:

```
Nome : Brasil
Idioma : português

*** Estados ***
Estado : Rio Grande do Sul
Estado : São Paulo
Estado : Minas Gerais
Estado : Rio de Janeiro
Estado : Paraná
Estado : Mato Grosso
```

países4.xml

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<pais>
    <nome> Brasil </nome>
    <idioma> português </idioma>
    <religiao> católica </religiao>
    <moeda nome="Real" símbolo="R$"/>
    <populacao> 190 milhões </populacao>
    <geografia>
        <clima> tropical </clima>
        <costa> 7367 km </costa>
        <pico nome="Neblina" altitude="3014"/>
    </geografia>
    <estados>
        <estado nome="Rio Grande do Sul" capital="Porto Alegre"/>
        <estado nome="São Paulo" capital="São Paulo"/>
        <estado nome="Minas Gerais" capital="Belo Horizonte"/>
        <estado nome="Rio de Janeiro" capital="Rio de Janeiro"/>
        <estado nome="Paraná" capital="Curitiba"/>
        <estado nome="Mato Grosso" capital="Cuiabá"/>
    </estados>
</pais>
```

Agora vamos complicar um pouco mais nosso arquivo XML. O próximo exemplo ilustra como trabalhar com elementos que contêm atributos na definição da própria tag (`nome="Minas Gerais"` `capital="Belo Horizonte"`). Para tal, alteramos a lista de estados para conter o `nome` e a `capital`. Neste caso, as informações são retornadas em forma de array. Utilizando estruturas como o `foreach()` para percorrer a lista de estados, acessamos cada atributo do array de forma indexada, sendo o índice o próprio nome do atributo que desejamos acessar. Veja a seguir:

 **exemplo7.php**

```
<?php
// interpreta o documento XML
$xml = simplexml_load_file('paises4.xml');

echo "**** Estados ***\n";
// percorre a lista de estados
foreach ($xml->estados->estado as $estado)
{
    // imprime o estado e a capital
    echo str_pad('Estado : ' . $estado['nome'], 30) .
        'Capital: ' . $estado['capital'] . "\n";
}

?>
```

 **Resultado:**

```
*** Estados ***
Estado : Rio Grande do Sul      Capital: Porto Alegre
Estado : São Paulo              Capital: São Paulo
Estado : Minas Gerais           Capital: Belo Horizonte
Estado : Rio de Janeiro         Capital: Rio de Janeiro
Estado : Paraná                 Capital: Curitiba
Estado : Mato Grosso            Capital: Cuiabá
```

Veja que no exemplo anterior é necessário saber exatamente o nome dos atributos que desejamos acessar de forma indexada. No último exemplo, percorremos a lista de estados por meio de um `foreach()`, como no exemplo anterior, mas, dentro desse laço de repetição, podemos percorrer os atributos de cada `$estado` (que é na verdade um objeto `SimpleXMLElement`) pelo método `attributes()`, o qual retorna a chave e o valor de cada atributo de um elemento (nodo).

 exemplo8.php

```
<?php
// interpreta o documento XML
$xml = simplexml_load_file('paises4.xml');
echo "*** Estados ***\n";
// percorre os estados
foreach ($xml->estados->estado as $estado)
{
    // percorre os atributos de cada estado
    foreach ($estado->attributes() as $key => $value)
    {
        echo "$key=>$value\n";
    }
}
?>
```

 Resultado:

```
*** Estados ***
nome=>Rio Grande do Sul
capital=>Porto Alegre
nome=>São Paulo
capital=>São Paulo
nome=>Minas Gerais
capital=>Belo Horizonte
nome=>Rio de Janeiro
capital=>Rio de Janeiro
nome=>Paraná
capital=>Curitiba
nome=>Mato Grosso
capital=>Cuiabá
```

2.17 Tratamento de erros

Existem diversas formas de realizar tratamento de erros em PHP. Nesta seção veremos as formas mais comuns utilizadas, finalizando com o tratamento de exceções introduzido pelo PHP5.

2.17.1 A função die()

A forma de manipulação de erro mais simples é abortar a execução da aplicação. Claro que não é qualquer erro que deva causar esta interrupção. Esta é a forma mais simplista de tratar erros e, portanto, não deve ser utilizada amplamente.

 erro_die.php

```
<?php
function Abrir($file = null)
{
    if (!$file)
    {
        die('Falta o parâmetro com o nome do Arquivo');
    }
    if (!file_exists($file))
    {
        die('Arquivo não existente');
    }
    if (!$retorno = @file_get_contents($file))
    {
        die('Impossível ler o arquivo');
    }
    return $retorno;
}

// abrindo um arquivo
$arquivo = Abrir('/tmp/arquivo.dat');
echo $arquivo;
?>
```

 **Resultado:**

Arquivo não existente

No exemplo anterior vimos que, como o referido arquivo não existe, a aplicação seria abortada com a mensagem de erro. Caso faltasse o parâmetro indicando o nome do arquivo, a aplicação seria abortada com a mensagem “Falta o parâmetro com o nome do Arquivo” e, caso não fosse possível efetuar a leitura do mesmo por problemas como falta de direitos, a aplicação seria abortada com a mensagem “Impossível ler o arquivo”.

Utilizar a função `die()` para controlar erros é ruim porque ela simplesmente aborta a execução do programa, o que, na maioria dos casos, não é o comportamento desejado, visto que nem todos os tipos de erros são fatais para a execução da aplicação.

2.17.2 Retorno de flags

A segunda forma de manipulação de erros é o retorno de flags `TRUE` ou `FALSE`. Retornamos `TRUE` em caso de sucesso na operação e `FALSE` em caso de erros.

 erro_flag.php

```
<?php
function Abrir($file = null)
{
    if (!$file)
    {
        return false;
    }
    if (!file_exists($file))
    {
        return false;
    }
    if (!$retorno = @file_get_contents($file))
    {
        return false;
    }
    return $retorno;
}

$arquivo = Abrir('/tmp/arquivo.dat');

// verificando se abriu o arquivo.
if (!$arquivo)
{
    echo 'Falha ao abrir o arquivo';
}
else
{
    echo $arquivo;
}
?>
```

 **Resultado:**

Falhas ao abrir o arquivo

A vantagem desse tipo de abordagem em relação ao primeiro é que a aplicação segue a sua execução sem ser abortada (a não ser que a abortemos ao testar o retorno da função). A desvantagem é que não sabemos exatamente em qual ponto do programa a execução falhou, não tendo como exibir a mensagem de erro correta, apenas uma genérica.

2.17.3 Lançando erros

Uma forma mais elegante de realizar a manipulação de erros é por meio das funções `trigger_error()`, que lança um erro, e a função `set_error_handler()`, a qual define uma função que realizará a manipulação dos erros lançados.

`trigger_error()`

Gera um erro.

```
bool trigger_error (string erro, int tipo)
```

Parâmetros	Descrição
<i>erro</i>	Mensagem de erro gerada.
<i>tipo</i>	Tipo de erro. Veja tabela a seguir.
Tipo	Significado
<code>E_USER_ERROR</code>	Gera um erro fatal.
<code>E_USER_WARNING</code>	Gera uma warning.
<code>E_USER_NOTICE</code>	Gera um notice.

`set_error_handler()`

Define uma função a ser utilizada para manipular erros.

```
mixed set_error_handler (callback handler, int tipo)
```

Parâmetros	Descrição
<i>handler</i>	Função ou método de um objeto a ser invocado.
<i>tipo</i>	Tipo de erro a ser manipulado. Veja tabela anterior (<code>trigger_error</code>).

erro_trigger.php

```
<?php
function Abrir($file = null)
{
    if (!$file)
    {
        trigger_error('Falta o parâmetro com o nome do Arquivo', E_USER_NOTICE);
        return false;
    }
    if (!file_exists($file))
    {
        trigger_error('Arquivo não existente', E_USER_ERROR);
        return false;
    }
}
```

```
if (!$retorno = @file_get_contents($file))
{
    trigger_error('Impossível ler o arquivo', E_USER_WARNING);
    return false;
}
return $retorno;
}

// função para manipular o erro
function manipula_erro($numero, $mensagem, $arquivo, $linha)
{
    $mensagem = "Arquivo $arquivo : linha $linha # no. $numero : $mensagem\n";

    // escreve no log todo tipo de erro
    $log = fopen('erros.log', 'a');
    fwrite($log, $mensagem);
    fclose($log);

    // se for uma warning
    if ($numero == E_USER_WARNING)
    {
        echo $mensagem;
    }
    // se for um erro fatal
    else if ($numero == E_USER_ERROR)
    {
        echo $mensagem;
        die;
    }
}

// define a função manipula_erro como manipuladora dos erros ocorridos
set_error_handler('manipula_erro');

// abrindo um arquivo
$arquivo = Abrir('/tmp/arquivo.dat');
echo $arquivo;
?>
```

Resultado:

Arquivo erro_trigger.php : linha 11 # no. 256 : Arquivo não existente

A vantagem desse tipo de abordagem para manipulação de erros é a liberdade que temos para personalizar o tratamento de erros por meio da função `manipula_erro()`

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 concentrarmos 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
<code>getMessage()</code>	Retorna a mensagem de erro.
<code>getCode()</code>	Retorna o código de erro.
<code>getFile()</code>	Retorna o arquivo no qual ocorreu o erro.
<code>getLine()</code>	Retorna a linha na qual ocorreu o erro.
<code>GetTrace()</code>	Retorna um array com as ações até o erro.
<code>getTraceAsString()</code>	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.

 erro_exception.php

```
<?php
function Abrir($file = null)
{
    if (!$file)
    {
        throw new Exception('Falta o parâmetro com o nome do arquivo');
    }
    if (!file_exists($file))
    {
        throw new Exception('Arquivo não existente');
    }
    if (!$retorno = @file_get_contents($file))
    {
        throw new Exception('Impossível ler o arquivo');
    }
    return $retorno;
}

// abrindo um arquivo
// tratamento de exceções
try
{
    $arquivo = Abrir('/tmp/arquivo.dat');
    echo $arquivo;
}
// captura o erro
catch (Exception $excecao)
{
    echo $excecao->getFile() . ' : ' . $excecao->getLine() . '# ' . $excecao->getMessage();
}
?>
```

 **Resultado:**

```
erro_exception.php : 10 # Arquivo não existente
```

No exemplo anterior, capturamos o erro no bloco `try{}` e `catch{}`, mas ainda não temos uma boa separação para cada tipo de erro ocorrido. Para realizar tal separação, podemos lançar tipos customizados para cada tipo de erro. No exemplo a seguir, criaremos três tipos de erros a serem lançados `ParameterException`, `FileNotFoundException` e `FilePermissionException`. Para tanto, iremos declarar essas classes por meio do mecanismo de herança, a partir da superclasse `Exception`. Quando uma exceção do tipo `ParameterException` for lançada, não ocorrerá nada. Quando uma exceção do tipo `FileNotFoundException` for lançada, todas as informações a respeito do erro serão exibidas,

a aplicação será terminada e, quando uma exceção do tipo `FilePermissionException` for lançada, somente a mensagem de erro será exibida.

erro_subexception.php

```
<?php

function Abrir($file = null)
{
    if (!$file)
    {
        throw new ParameterException('Falta o parâmetro com o nome do arquivo');
    }
    if (!file_exists($file))
    {
        throw new FileNotFoundException('Arquivo não existente');
    }
    if (!$retorno = @file_get_contents($file))
    {
        throw new FilePermissionException('Impossível ler o arquivo');
    }
    return $retorno;
}

// definição das subclasses de erro
class ParameterException extends Exception{}
class FileNotFoundException extends Exception{}
class FilePermissionException extends Exception{}


// abrindo um arquivo
// tratamento de exceções
try
{
    $arquivo = Abrir('/tmp/arquivo.dat');
    echo $arquivo;
}
// captura o erro
catch (ParameterException $excecao)
{
    // não faz nada...
}
catch (FileNotFoundException $excecao)
{
    var_dump($excecao->getTrace());
    echo "finalizando aplicação...\n";
    die;
}
```

```
catch (FilePermissionException $excecao)
{
    echo $excecao->getFile() . ' : ' . $excecao->getLine() . '# ' . $excecao->getMessage();
}
?>
```

 **Resultado:**

```
array(1) {
[0]=>
array(4) {
["file"]=> string(70) "/home/pablo/book/exemplos/erro_subexception.php"
["line"]=> int(28)
["function"]=> string(5) "Abrir"
["args"]=> array(1) {
    [0]=> string(16) "/tmp/arquivo.dat" }
}
}
finalizando aplicação...
```

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

aos seus serviços. Neste meio destacamos o Oracle, o SQL Server, o PostgreSQL, o MySQL, o Interbase e o DB2, dentre outros.

O PHP conecta-se a uma enorme variedade de gerenciadores de bancos de dados disponíveis no mercado. Para cada tipo de banco de dados existe uma gama de funções associadas para realizar operações como conexão, consulta, retorno, desconexão, dentre outras. Nesta seção abordaremos de forma superficial as principais funções. Se você deseja desenvolver uma aplicação utilizando qualquer um dos bancos de dados listados a seguir, é imprescindível visitar o site do PHP para obter a lista completa de funções de acesso.

Banco	Site
Sybase	www.php.net/sybase
SQL Server	www.php.net/mssql
MySQL	www.php.net/mysql
MySQL>=4.1	www.php.net/mysql
Frontbase	www.php.net/fbsql
Interbase	www.php.net/ibase
ODBC	www.php.net/odbc
Informix	www.php.net/ifx
Oracle	www.php.net/oci8
PostgreSQL	www.php.net/pgsql
SQLite	www.php.net/sqlite

3.1.2 Exemplos

Para exemplificar o acesso ao banco de dados criaremos dois programas. O primeiro deles é 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
```

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.

 pgsql_lista.php

```
<?php  
// abre conexão com Postgres  
$conn = pg_connect("host=localhost port=5432 dbname=livro user=postgres password=");  
  
// define consulta que será realizada  
$query = 'SELECT codigo, nome FROM famosos';  
  
// envia consulta ao banco de dados  
$result = pg_query($conn, $query);  
  
if ($result)  
{  
    // percorre resultados da pesquisa  
    while ($row = pg_fetch_assoc($result))  
    {  
        echo $row['codigo'] . ' - ' . $row['nome'] . "<br>\n";  
    }  
}  
// fecha a conexão  
pg_close($conn);  
?>
```

 **Resultado:**

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

Nos exemplos anteriores, você deve ter notado quais funções precisamos utilizar para nos conectarmos a um banco de dados PostgreSQL e executar comandos SQL. Basicamente temos o `pg_connect()` para abrir uma conexão, `pg_query()` para executar uma instrução SQL e `pg_close()` para fechar uma conexão, dentre outros comandos. No PHP, as funções de acesso a banco de dados são específicas para cada sistema gerenciador de banco de dados (SGBD). Isto que dizer que as funções de acesso e consulta a um banco de dados MySQL são diferentes de um Oracle, um DB2, entre outros.

No exemplo seguinte, procuramos repetir o mesmo exemplo de inserção de dados em um banco, outrora em PostgreSQL, agora em MySQL. Veja que o exemplo é bastante similar, porém a nomenclatura das funções utilizadas e a ordem dos parâmetros são outras. Para o MySQL, utilizamos basicamente as funções `mysql_connect()` para conectar ao banco, `mysql_select_db()` para selecionar o banco de dados ativo, `mysql_query()` para enviar a instrução SQL para o banco e `mysql_close()` para fechar a conexão com o banco de dados.

Neste exemplo estamos conectando a um banco de dados localizado localmente (`localhost`), com o usuário `root` e a senha `mysql`. Primeiro, criaremos a base de dados `livro` e, posteriormente, criaremos a tabela de famosos.

```
# mysql  
mysql> create database livro;  
Query OK, 1 row affected (0.00 sec)  
  
mysql> use livro  
Database changed  
  
mysql> CREATE TABLE famosos (codigo integer, nome varchar(40));  
Query OK, 0 rows affected (0.01 sec)
```

 **mysql_insere.php**

```
<?php  
// abre conexão com o MySQL  
$conn = mysql_connect('localhost', 'root', 'mysql');
```

```
// seleciona o banco de dados ativo
mysql_select_db('livro', $conn);

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

// fecha a conexão
mysql_close($conn);
?>
```

Assim como reescrevemos o exemplo para inserção de dados de PostgreSQL para MySQL, iremos agora reescrever o programa de listagem de dados, responsável por obter os registros do banco de dados e exibi-los em tela.

mysql_lista.php

```
<?php
// abre conexão com o MySQL
$conn = mysql_connect('localhost', 'root', 'mysql');

// seleciona o banco de dados ativo
mysql_select_db('livro', $conn);

// define a consulta que será realizada
$query = 'SELECT codigo, nome FROM famosos';

// envia consulta ao banco de dados
$result = mysql_query($query, $conn);

if ($result)
{
    // percorre resultados da pesquisa
    while ($row = mysql_fetch_assoc($result))
    {
        echo $row['codigo'] . ' - ' . $row['nome'] . "<br>\n";
    }
}
// fecha a conexão
mysql_close($conn);
?>
```

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

Para podermos utilizar o PDO, deve-se ter habilitado no `php.ini` as bibliotecas (drivers) de acesso para o banco de dados de nossa aplicação. No Linux, habilitamos da seguinte forma:

```
extension=mysql.so
extension=pgsql.so
extension=sqlite.so
extension=pdo_mysql.so
extension=pdo_pgsql.so
extension=pdo_sqlite.so
```

No Windows, entretanto, habilitamos da seguinte forma:

```
extension=php_mysql.dll
extension=php_pgsql.dll
extension=php_sqlite.dll
extension=php_pdo_mysql.dll
extension=php_pdo_pgsql.dll
extension=php_pdo_sqlite.dll
```

3.2.2 Exemplos

Para facilitar a aprendizagem da PDO, veremos uma série de exemplos, cada qual abordando um aspecto diferente relacionado a bancos de dados. Por ora veremos como se dá uma inserção e, posteriormente, veremos uma listagem de dados.

3.2.2.1 Inserção, alteração e exclusão

Neste primeiro exemplo, assim como no exemplo de inserção de dados visto anteriormente, o programa irá se conectar ao banco de dados em `localhost`, chamado `livro`, com o usuário `postgres`. Em seguida, ele irá inserir dados relativos a clientes na base de dados. Por fim, ele fechará a conexão ao banco de dados. Caso ocorra algum erro durante a execução das instruções SQL, será gerada uma exceção, tratada adequadamente pelo bloco `try/catch`.

pdo_insere.php

```
<?php
try
{
    // instancia objeto PDO, conectando no postgresql
    $conn = new PDO('pgsql:dbname=livro;user=postgres;password=;host=localhost');
    // executa uma série de instruções SQL
    $conn->exec("INSERT INTO famosos (codigo, nome) VALUES (1, 'Érico Veríssimo')");
    $conn->exec("INSERT INTO famosos (codigo, nome) VALUES (2, 'John Lennon')");
    $conn->exec("INSERT INTO famosos (codigo, nome) VALUES (3, 'Mahatma Gandhi')");
}
```

```
$conn->exec("INSERT INTO famosos (codigo, nome) VALUES (4, 'Ayrton Senna')");
$conn->exec("INSERT INTO famosos (codigo, nome) VALUES (5, 'Charlie Chaplin')");
$conn->exec("INSERT INTO famosos (codigo, nome) VALUES (6, 'Anita Garibaldi')");
$conn->exec("INSERT INTO famosos (codigo, nome) VALUES (7, 'Mário Quintana')");
// fecha a conexão
$conn = null;
}
catch (PDOException $e)
{
    // caso ocorra uma exceção, exibe na tela
    print "Erro!: " . $e->getMessage() . "\n";
    die();
}
?>
```

3.2.2.2 Listagens

No próximo exemplo, assim como no programa de listagem de dados visto anteriormente, o programa se conectará ao banco de dados `livro`. Em seguida, ele irá selecionar código e nome dos famosos existentes nesse banco de dados e exibir na tela. Veja como os resultados são percorridos via laço de repetições (`FOREACH`) por meio de uma simples iteração. Cada linha (`resultset`) do resultado é retornada para dentro do array `$row`, indexado pelos nomes das colunas do `SELECT ("codigo", "nome")` e também por índices numéricos que representam as posições das colunas (0, 1, ...). Novamente o controle de erros é realizado por tratamento de exceções. Caso alguma exceção seja gerada, a sua mensagem será exibida na tela pelo bloco `catch`.

pdo_lista.php

```
<?php
try
{
    // instancia objeto PDO, conectando no Postgresql
    $conn = new PDO('pgsql:dbname=livro;user=postgres;password=;host=localhost');
    // executa uma instrução SQL de consulta
    $result = $conn->query("SELECT codigo, nome FROM famosos");
    if ($result)
    {
        // percorre os resultados via iteração
        foreach($result as $row)
        {
            // exibe os resultados
            echo    $row['codigo'] . ' - ' .
                    $row['nome'] . "<br>\n";
        }
    }
}
```

```
// fecha a conexão
$conn = null;
}
catch (PDOException $e)
{
    print "Erro!: " . $e->getMessage() . "<br/>";
    die();
}
?>
```

Resultado:

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

Podemos também retornar os dados de uma consulta por meio da função `fetch()`, a qual aceita como parâmetro o “estilo de fetch”. De acordo com o estilo, o retorno poderá ser um dos relacionados a seguir:

Parâmetros	Descrição
<code>PDO::FETCH_ASSOC</code>	Retorna um array indexado pelo nome da coluna.
<code>PDO::FETCH_NUM</code>	Retorna um array indexado pela posição numérica da coluna.
<code>PDO::FETCH_BOTH</code>	Retorna um array indexado pelo nome da coluna e pela posição numérica da mesma.
<code>PDO::FETCH_OBJ</code>	Retorna um objeto anônimo (<code>StdClass</code>), de modo que cada coluna é acessada como uma propriedade.

Neste exemplo a seguir, repetimos o mesmo programa que lista os códigos e os nomes dos famosos do banco de dados. A diferença é que agora utilizaremos a função `fetch()` para iterar os resultados. O estilo de fetch desejado será o `PDO::FETCH_OBJ`, que retorna os dados da consulta em forma de objeto. Veja que agora a variável `$row` é um objeto, e cada coluna retornada (`codigo, nome`) é acessada como uma propriedade deste objeto.

pdo_lista_obj.php

```
<?php
try
{
    // instancia objeto PDO, conectando no Postgresql
    $conn = new PDO('pgsql:dbname=livro;user=postgres;password=;host=localhost');
```

```
// executa uma instrução SQL de consulta
$result = $conn->query("SELECT codigo, nome FROM famosos");
if ($result)
{
    // percorre os resultados via fetch()
    while ($row = $result->fetch(PDO::FETCH_OBJ))
    {
        // exibe os dados na tela, acessando o objeto retornado
        echo $row->codigo . ' - ' .
              $row->nome . "<br>\n";
    }
}
// fecha a conexão
$conn = null;
}
catch (PDOException $e)
{
    print "Erro!: " . $e->getMessage() . "<br/>";
    die();
}
?>
```

Resultado:

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

Reescreveremos, então, o programa de inserção de dados, que anteriormente foi escrito para PostgreSQL, agora em MySQL. Veja que a única diferença entre um programa e outro é a linha em que instanciamos o objeto PDO (`new PDO`), de modo que os parâmetros são diferentes. Todo o restante do programa é simplesmente igual em razão da interface clara e unificada da biblioteca PDO. Veja a seguir como fica nosso programa que insere dados na tabela de famosos adaptado para o uso com MySQL.

pdo_insere_my.php

```
<?php
try
{
    // instancia objeto PDO, conectando no mysql
    $conn = new PDO('mysql:host=localhost;port=3307;dbname=livro', 'root', 'mysql');
```

```
// executa uma série de instruções SQL
$conn->exec("INSERT INTO famosos (codigo, nome) VALUES (1, 'Érico Veríssimo')");
$conn->exec("INSERT INTO famosos (codigo, nome) VALUES (2, 'John Lennon')");
$conn->exec("INSERT INTO famosos (codigo, nome) VALUES (3, 'Mahatma Gandhi')");
$conn->exec("INSERT INTO famosos (codigo, nome) VALUES (4, 'Ayrton Senna')");
$conn->exec("INSERT INTO famosos (codigo, nome) VALUES (5, 'Charlie Chaplin')");
$conn->exec("INSERT INTO famosos (codigo, nome) VALUES (6, 'Anita Garibaldi')");
$conn->exec("INSERT INTO famosos (codigo, nome) VALUES (7, 'Mário Quintana')");
// fecha a conexão
$conn = null;
}
catch (PDOException $e)
{
    // caso ocorra uma exceção, exibe na tela
    print "Erro!: " . $e->getMessage() . "\n";
    die();
}
?>
```

3.3 Uma API orientada a objetos

3.3.1 Introdução

A linguagem SQL (Structured Query Language) surgiu nos anos de 1970 durante o desenvolvimento de um sistema de banco de dados chamado de System R. Implementado por Chamberlin e Boyce, o System R foi baseado no modelo proposto por Edgar Codd em um artigo publicado em 1970 no periódico da ACM (Association for Computing Machinery).

No ano de 1974 foi desenvolvido o primeiro banco de dados relacional não SQL – o Ingres – criado por estudantes da universidade de Berkeley. O código-fonte do Ingres era distribuído em fitas por preços simbólicos na época. Alguns membros da equipe que participou do projeto vieram a formar mais tarde a Sybase, que foi, durante muito tempo (anos de 1980 e 1990), o segundo banco de dados mais utilizado, atrás da Oracle. O Sybase foi licenciado, mais tarde (1992), pela Microsoft, que o relançou sob o nome SQL Server.

Como o projeto System R provou ser um sucesso, a IBM continuou investindo e desenvolvendo produtos comerciais baseados nele, como o SQL/DS, de 1981, culminando com o lançamento do DB2 em 1983. Fundada em 1977, a Relational Software, que mais tarde (1983) veio a se chamar Oracle Corporation, criou sua própria versão de banco de dados SQL, baseado nas idéias de Chamberlin e Boyce, o que resultou no lançamento do Oracle V2 em 1979, o primeiro banco de dados SQL lançado comercialmente.

O projeto Ingres ainda veio a originar uma série de outros bancos de dados. O próprio líder da equipe original do Ingres, Stonebraker, ao retornar para Berkeley em 1985, após ter saído em 1982, veio a criar o projeto Postgres (post-ingres), para adicionar alguns recursos como a possibilidade de definir tipos e de descrever as relações do banco de dados. A versão 1 do Postgres foi lançada em 1989 para um pequeno número de usuários. Após concluir mais alguns recursos, o projeto terminou na versão 4, em 1993, mas com um grande número de usuários. Por utilizar uma licença “open source”, em 1994, dois estudantes de Berkeley (Yu e Chen) adicionaram a possibilidade de interpretar comandos SQL ao Postgres, que foi rebatizado de Postgres95 e lançado na web. Em 1996, alguns desenvolvedores (Momjian, Fournier e Mikheev) começaram a trabalhar colaborativamente, estabilizando o código-fonte herdado de Berkeley, que foi relançado com o nome PostgreSQL em janeiro de 1997.

Atualmente, a maioria das aplicações utiliza a linguagem SQL para manipular bancos de dados. Essa linguagem possui formas de definir a estrutura e a integridade dos dados, permite a inserção, alteração e exclusão de registros. A seguir veremos a sintaxe das instruções SQL utilizadas ao longo deste livro.

3.3.2 Sintaxe SQL

A seguir, resumiremos as principais instruções SQL para manipulação de dados, a DML (Data Manipulation Language). O objetivo aqui não é ensinar a linguagem SQL, para isto existem dezenas de livros excelentes no mercado. Faremos uma breve revisão para proporcionar embasamento técnico necessário para a continuidade dos nossos exemplos.

SELECT

A instrução SELECT é utilizada para retornar um conjunto de linhas a partir de uma ou mais tabelas do banco de dados. Ela permite definir quais colunas de quais tabelas desejamos obter, as fontes (tabelas) dos dados, os ordenamentos e agrupamentos das informações, entre outros. Veja a seguir a sintaxe do SELECT:

```
SELECT <coluna1>, <coluna2>
  FROM <tabela1>, <tabela2>
 WHERE <condições>
 GROUP BY <coluna1>, <coluna2>
 HAVING <condição>
 ORDER BY <colunaX>, <colunaY>
```

A instrução a seguir lista pessoas, juntamente com a sua cidade. Para tanto, é necessário relacionar as tabelas pessoas e cidades, o que produz um produto cartesiano (matriz), sendo necessário restringir o número de registros, explicitando como restrição a ligação entre as duas tabelas (por meio do código da cidade).

```
SELECT pessoas.codigo, pessoas.nome, cidades.codigo, cidades.nome
  FROM pessoas, cidades
 WHERE pessoas.cidade=cidades.codigo
 GROUP BY pessoas.nome
```

INSERT

A instrução **INSERT** é utilizada para inserir uma ou mais linhas em uma tabela do banco de dados. O **INSERT** permite definir quais colunas terão valores a serem preenchidos, bem como inserir um conjunto de registros a partir de um comando **SELECT**.

Veja a seguir a sintaxe do **INSERT**:

```
INSERT INTO <tabela> (<coluna1>, <coluna2>, <colunaX>)
VALUES (<valor1>, <valor2>, <valor3>);
```

Exemplo:

```
INSERT INTO pessoas (codigo, nome) VALUES (1, 'Maria');
INSERT INTO pessoas (codigo, nome) VALUES (2, 'Pedro');
```

UPDATE

A instrução **UPDATE** é utilizada para alterar os valores de colunas de uma ou mais linhas em uma tabela do banco de dados. O **UPDATE** permite alterar os valores de várias colunas ao mesmo tempo e também permite definir uma condição nos mesmos moldes do comando **SELECT**, envolvendo, inclusive, mais de uma tabela.

Veja a seguir a sintaxe do **UPDATE**:

```
UPDATE <tabela> SET <coluna1> = <valor1>, <coluna2> = <valor2>
 WHERE <condição>
```

No exemplo que segue estamos reajustando o valor de custo e o valor de venda de todos os produtos do fornecedor cujo código é “10”.

```
UPDATE produtos SET custo = custo * 1.2,
                    venda = venda * 1.25
 WHERE fornecedor = 10;
```

DELETE

A instrução **DELETE** é utilizada para remover uma ou mais linhas em uma tabela do banco de dados. O **DELETE** permite definir uma condição nos mesmos moldes do comando **SELECT**, envolvendo, inclusive, mais de uma tabela. Veja a seguir a sintaxe do **DELETE**:

```
DELETE FROM <tabela>
 WHERE <condição>
```

No exemplo que segue estamos excluindo todos os produtos cujo valor de venda é inferior a “0.5”.

```
DELETE FROM produtos
WHERE venda <= "0.5";
```

3.3.3 Usando SQL no PHP

Vimos anteriormente a sintaxe das instruções mais comuns do SQL. O uso dessas instruções no PHP é relativamente simples, como já visto anteriormente. Para inserir dados em uma tabela, por exemplo, basta concatenarmos a instrução SQL com nossas variáveis.

```
<?php
$codigo = 5;
$nome = 'maria';
$sql = "INSERT INTO pessoas (codigo, nome) VALUES ('$codigo', '$nome')";
echo $sql;
?>
```

 **Resultado:**

```
INSERT INTO pessoas (codigo, nome) VALUES ('5', 'maria')
```

Até aí tudo bem, mas e quando nos deparamos com uma situação em que a instrução SQL é dinâmica? Geralmente as aplicações trazem algum tipo de tela de consulta multicritério. Você encontra esse tipo de tela de consulta em sites de comércio eletrônico em que se pode filtrar por vários campos de pesquisa ao mesmo tempo. Nestes casos a instrução SQL deve ser construída por meio de desvios condicionais e concatenações, como no caso a seguir.

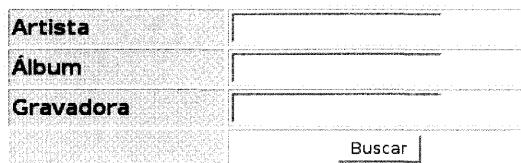


Figura 3.1 – Tela de consulta multicritério.

Veja que temos uma tela com três campos de entrada (`artista`, `álbum` e `gravadora`) para pesquisa em um catálogo de discos e, em seguida, o código-fonte do programa que irá construir a instrução SQL de pesquisa.

```
<?php
$artista = $_POST['artista'];
$album = $_POST['álbum'];
```

```

$gravadora = $_POST['gravadora'];

$sql = "SELECT codigo, nome FROM discos WHERE ";

if (isset($artista))
{
    $sql .= "artista ILIKE '%$artista%' AND ";
}

if (isset($album))
{
    $sql .= "album ILIKE '%$album%' AND ";
}

if (isset($gravadora))
{
    $sql .= "gravadora ILIKE '%$gravadora%' AND ";
}

echo substr($sql,0,-4) . "<br>\n";
?>

```

Caso o usuário tenha preenchido “beatles” no campo artista e “help” no campo álbum, o resultado da consulta SQL deve ser o seguinte:

Resultado:

```
SELECT codigo, nome FROM discos WHERE artista ILIKE '%beatles%' AND album ILIKE '%help%'
```

Você deve ter percebido que o código da aplicação se tornou complexo e sujeito a erros de digitação e concatenação. Não é muito raro encontrar programas com centenas de linhas com baterias de IF e concatenações de string de instruções SQL. Quando mais testes realizarmos e quanto mais campos tivermos de concatenar na instrução SQL, maior será a probabilidade de erros, uma vez que estaremos naturalmente adicionando maior complexidade ao sistema.

3.3.3.1 Proposta

O SQL é uma linguagem complexa e muito rica em funcionalidades. Apesar de ser utilizada pela grande maioria das aplicações, boa parte dos programadores ainda tem dificuldades em dominá-la, principalmente em virtude de sua sintaxe. Uma forma de facilitar o uso do SQL dentro das aplicações é por meio da criação de classes que escondam a linguagem atrás de uma interface, um conjunto de métodos que permitam ao desenvolvedor manipular a base de dados sem que o mesmo tenha profundo conhecimento da linguagem SQL. Existem diversos padrões de projeto (design patterns) já estabelecidos que tratam desse assunto. Mas o que é um design pattern?

3.3.4 Design pattern

Praticamente todo livro sobre design patterns explica este conceito por meio das palavras de Christopher Alexander. De acordo com ele, “Um pattern descreve um problema que ocorre com freqüência em nosso ambiente, e então explica a essência da solução para este problema, de forma que tal solução possa ser utilizada milhões de outras vezes, sem ao menos repeti-la uma única vez”.

Ao dizer isto, Christopher estava se referindo a padrões para construções, como prédios, pontes, dentre outros. Apesar disso, essas palavras podem ser perfeitamente utilizadas no contexto da engenharia de software.

O termo design pattern tem sido largamente utilizado no mundo da orientação a objetos para descrever formas de comunicação e relacionamento entre objetos e classes de maneira a solucionar determinados problemas de projeto. A utilização de um design pattern não é a mais simples nem a mais rápida maneira de solucionar um determinado problema, mas é, sem dúvidas, a solução forma que traz maior flexibilidade e capacidade de reuso da solução, trazendo benefícios a médio e a longo prazo na manutenibilidade do código-fonte.

Provavelmente você irá reconhecer em vários design patterns aspectos que já tenha vivenciado; eles não são criados ou inventados, são somente uma revelação de técnicas já existentes, descobertas, reunidas e documentadas sob um nome. Provavelmente você irá encontrar diversos design patterns em um sistema orientado a objetos. Um bom programador reconhece quando está enfrentando um determinado tipo de problema pela segunda vez. Neste caso, ele aplica o mesmo padrão de solução que adotou na primeira vez em vez de repensar e projetar novamente. Os design patterns propõem uma espécie de catálogo para tais soluções, provendo um nome, um contexto e uma solução genérica.

O nome do design pattern é utilizado para descrever o problema, as circunstâncias sob as quais ele ocorre, bem como sua solução. Escolher um bom nome é muito importante, pois nomes de patterns são constantemente utilizados em livros e outras documentações de referência quando é necessário descrever uma situação similar ou simplesmente fazer referência.

Um design pattern é independente de linguagem de programação. Em princípio, uma linguagem de programação com um bom suporte aos conceitos de orientação a objetos deve oferecer os recursos necessários para se implementar a maioria dos patterns existentes.

3.3.5 Query Object

Query Object é um design pattern implementado por uma estrutura de objetos que se transformam em uma instrução SQL, ou seja, ele é um objeto que representa uma instrução de insert, update, delete etc. As instruções SQL são criadas por meio da utilização desses objetos. Uma classe que implementa o pattern Query Object deve permitir ao programador expressar vários tipos de instruções (`Insert`, `Update`, `Delete`, `Select`) por meio de um conjunto de métodos e posteriormente transformar essa informação no formato SQL.

As facilidades não estão em somente facilitar o uso do SQL dentro de uma aplicação, mas também em tornar a forma como representamos as expressões mais independente de seu contexto, permitindo-nos reutilizar uma mesma instrução em tabelas e em bancos de dados diferentes.

Que tal criarmos classes para abstrair as instruções SQL? Frameworks como o Hibernate oferecem uma consistente API orientada a objetos que provê funcionalidades de manipulação de dados sem ser necessário a utilização explícita de código SQL. Podemos criar um conjunto mínimo de classes para manipulação de dados no PHP sem precisar usar comandos SQL diretamente no código-fonte. Para isso, precisamos projetar uma interface que nos forneça os métodos adequados. Um Query Object sempre deve ter relacionado a si um critério de seleção de dados, que chamamos de “Criteria”. Um critério nos permite definir o conjunto de dados sobre o qual o Query Object irá atuar. Veja na Figura 3.2 a representação de um Query Object.

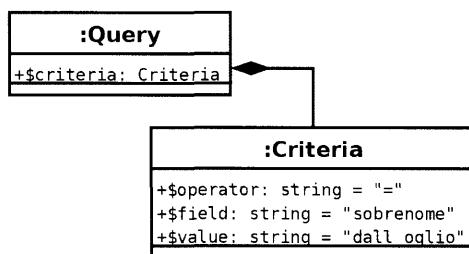


Figura 3.2 – Query Object.

Vamos chamar nosso Query Object de `TSqlInstruction`, por representar de forma abstrata uma instrução que enviamos ao banco de dados. Existem diversos tipos de instruções SQL (`Insert`, `Select`, `Update`, `Delete`) e, para cada uma delas, criaremos uma subclasse contendo os comandos específicos (`TSqlInsert`, `TSqlSelect`, `TSqlUpdate`, `TSqlDelete`).

As características comuns a todas as instruções SQL colocamos na classe-pai (`TSqlInstruction`). Todas as instruções SQL precisam manter a informação de qual

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()`.

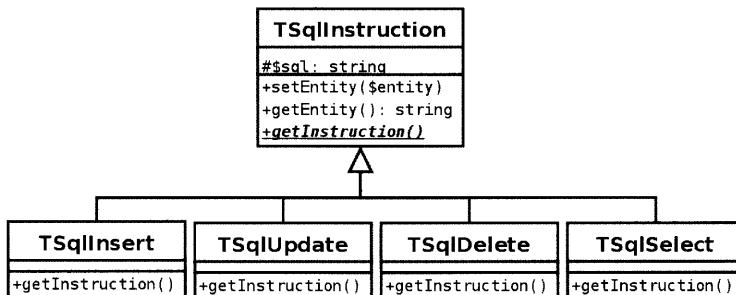


Figura 3.3 – Estrutura de classes SQL.

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.

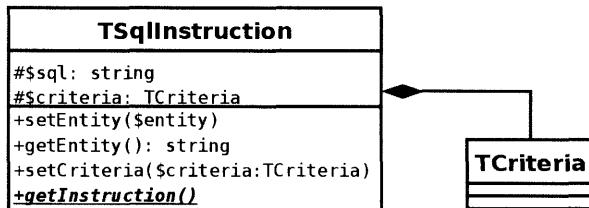
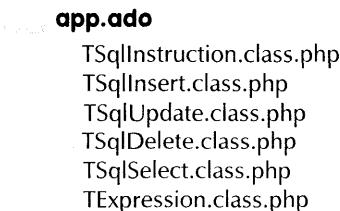


Figura 3.4 – TSqIInstruction e TCriteria.

Para melhor organizar nossa estrutura de diretórios, colocaremos, na pasta `app.ado`, todas as classes que se referem à manipulação de SQL. Escolhemos a sigla ADO para significar “Advanced Data Objects”, fazendo também uma analogia à tecnologia de mesmo nome largamente utilizada. Veja a seguir como ficou nossa estrutura de diretório com todas as classes criadas.

Figura 3.5 – Estrutura do diretório `app.ado`.

3.3.6 Critérios de seleção

3.3.6.1 Introdução

Como vimos anteriormente, as instruções `Update`, `Delete` e `Select` se utilizam de expressões lógicas para selecionar um conjunto de registros. Até o momento víhamos escrevendo manualmente tais expressões de seleção (`where coluna='valor'`). A partir de agora implementaremos a classe `TExpression`, que gerenciará tais expressões por meio de um mecanismo totalmente orientado a objetos. Ainda não sabemos quais nem quantos tipos de expressões nossa aplicação suportará, portanto estamos criando uma classe abstrata, a partir da qual iremos derivar os tipos de expressão. A partir dela, construiremos classes-filha que representam cada um dos tipos de expressão possíveis. Com ela, obrigamos que cada classe-filha tenha o método `dump()`, que será responsável por retornar a expressão em forma de string. Ainda nesta classe, aproveitamos para declarar duas constantes que definem os operadores `AND` e `OR`.

 **TExpression.class.php**

```
<?php
/*
 * classe TExpression
 * classe abstrata para gerenciar expressões
 */
abstract class TExpression
{
    // operadores lógicos
    const AND_OPERATOR = 'AND ';
    const OR_OPERATOR = 'OR ';
    // marca método dump como obrigatório
    abstract public function dump();
}
?>
```

Começaremos, então, a escrever os tipos de expressão que podem existir. O tipo mais simples que podemos imaginar é composto de três partes – uma variável, um operador e um valor – por exemplo: “`salario > 3000`”, de modo que `salario` é a variável, `>` é o operador, e `3000` é o valor a ser comparado. Esta expressão se constitui um filtro de seleção. Então o filtro é o primeiro tipo de expressão que podemos representar.

Para implementar um filtro, vamos criar a classe `TFilter`, que será filha da classe `TExpression`. Um objeto `TFilter` já receberá em seu método construtor os três parâmetros (variável, operador e valor) e tratará de armazená-los internamente. No entanto, antes de armazenar o valor precisamos tratá-lo, afinal o PHP suporta diversos tipos de dados (`string`, `integer`, `array`, dentre outros), e precisamos converter esses tipos de dados em uma string plana. Esta será a função do método `transform()`, o qual fará vários testes para descobrir o tipo da variável `valor` e realizará as conversões necessárias. Isto é necessário porque alguns tipos de dados como o `array` são representados diferentemente no PHP e no banco de dados. Ao tipo `string`, por exemplo, devemos adicionar aspas antes de enviar para o banco.

 **TFilter.class.php**

```
<?php
/*
 * classe TFilter
 * Esta classe provê uma interface para definição de filtros de seleção
 */
class TFilter extends TExpression
{
    private $variable; // variável
    private $operator; // operador
    private $value; // valor
```

```
/*
 * método __construct()
 * instancia um novo filtro
 * @param $variable = variável
 * @param $operator = operador (>,<)
 * @param $value    = valor a ser comparado
 */
public function __construct($variable, $operator, $value)
{
    // armazena as propriedades
    $this->variable = $variable;
    $this->operator = $operator;
    // transforma o valor de acordo com certas regras
    // antes de atribuir à propriedade $this->value
    $this->value    = $this->transform($value);
}

/*
 * método transform()
 * recebe um valor e faz as modificações necessárias
 * para ele ser interpretado pelo banco de dados
 * podendo ser um integer/string/boolean ou array.
 * @param $value = valor a ser transformado
 */
private function transform($value)
{
    // caso seja um array
    if (is_array($value))
    {
        // percorre os valores
        foreach ($value as $x)
        {
            // se for um inteiro
            if (is_integer($x))
            {
                $foo[] = $x;
            }
            else if (is_string($x))
            {
                // se for string, adiciona aspas
                $foo[] = "'$x'";
            }
        }
        // converte o array em string separada por ","
        $result = '(' . implode(',', $foo) . ')';
    }
}
```

```
// caso seja uma string
else if (is_string($value))
{
    // adiciona aspas
    $result = "'$value'";
}

// caso seja valor nulo
else if (is_null($value))
{
    // armazena NULL
    $result = 'NULL';
}

// caso seja booleano
else if (is_bool($value))
{
    // armazena TRUE ou FALSE
    $result = $value ? 'TRUE' : 'FALSE';
}

else
{
    $result = $value;
}

// retorna o valor
return $result;
}

/*
 * método dump()
 * retorna o filtro em forma de expressão
 */
public function dump()
{
    // concatena a expressão
    return "{$this->variable} {$this->operator} {$this->value}";
}
?>
```

Agora que já criamos a classe `TFilter`, veremos no pequeno exemplo a seguir como se dá seu uso. Criaremos uma série de filtros utilizando os mais variados tipos de dados suportados pelo PHP. Primeiramente temos um filtro usando uma data (string), em seguida um número (3000), posteriormente um array de sexos, um valor nulo e por fim um valor booleano. Veja como a classe `TFilter` trabalha esses valores internamente, convertendo-os na expressão final.

 filter.php

```
<?php
// carrega as classes necessárias
include_once 'app.ado/TExpression.class.php';
include_once 'app.ado/TFilter.class.php';

// cria um filtro por data (string)
$filter1=new TFilter('data', '=', '2007-06-02');
echo $filter1->dump();
echo "<br>\n";

// cria um filtro por salário (integer)
$filter2=new TFilter('salario', '>', 3000);
echo $filter2->dump();
echo "<br>\n";

// cria um filtro por sexo (array)
$filter3=new TFilter('sexo','IN', array('M', 'F'));
echo $filter3->dump();
echo "<br>\n";

// cria um filtro por contrato (NULL)
$filter4=new TFilter('contrato', 'IS NOT', NULL);
echo $filter4->dump();
echo "<br>\n";

// cria um filtro por ativo (boolean)
$filter5=new TFilter('ativo', '=', TRUE);
echo $filter5->dump();
echo "<br>\n";
?>
```

 Resultado:

```
data = '2007-06-02'
salario > 3000
sexo IN ('M','F')
contrato IS NOT NULL
ativo = TRUE
```

Com a classe `TFilter` conseguimos suportar expressões simples, mas, freqüentemente, quando desejamos expressar filtros para seleção de dados, utilizamos operadores lógicos (`AND`, `OR`) para criar expressões compostas. Em se tratando de expressões lógicas, não há limite quanto à quantidade de níveis que podemos utilizar para nos expressar, por exemplo:

```
cidade="Porto Alegre" and (idade = 18 or idade = 20)
```

Para que possamos criar expressões como a destacada anteriormente, na qual podem ocorrer diversos níveis de filtros, precisamos de uma estrutura mais robusta que suporte esse tipo de representação utilizando a composição de objetos, o que veremos a seguir.

3.3.6.2 Composite Pattern

Uma das interações mais ricas da orientação a objetos é a composição, a qual nos permite que um objeto contenha outros objetos criando complexos relacionamentos. Um dos relacionamentos mais instigantes que a composição provê é a possibilidade de criar uma hierarquia de objetos, contendo relações todo-parte, de modo que você pode tratar exatamente da mesma forma objetos individuais, bem como objetos compostos. Esta forma de relacionamento é chamada de Composite Pattern. Existem diversas situações em que podemos aplicar esse pattern com grande sucesso.

O exemplo que demonstramos a seguir é a estrutura de um menu de opções. Sabemos que este (classe `Menu`) pode ser composto de opções (classe `MenuItem`) e também pode ser composto por outros submenus (classe `Menu`), neste caso temos uma composição recursiva, como exemplifica a Figura 3.6.

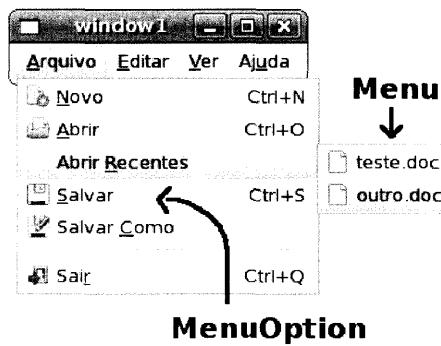


Figura 3.6 – Menu de opções.

No diagrama a seguir, note que existe a superclasse abstrata `MenuShell`, da qual as classes `Menu` e `MenuItem` são filhas. A classe `Menu` possui o método `add()`, utilizado para adicionar opções no menu (objetos `MenuItem`), mas como este método aguarda por um parâmetro do tipo `MenuShell`, ele aceitará tanto objetos do tipo `Menu` quanto `MenuItem`, criando uma representação recursiva, baseada na composição. Para que isso funcione, no entanto, é necessário que os objetos `Menu` e `MenuItem` se comportem de maneira similar. A chave de tudo isto é a direção da composição que aponta para a classe abstrata `MenuShell`. Quando estivermos adicionando opções no menu, não precisaremos fazer nenhum tipo de distinção, pois poderemos adicionar indiscriminadamente tanto objetos do tipo `Menu` quanto objetos do tipo `MenuItem`.

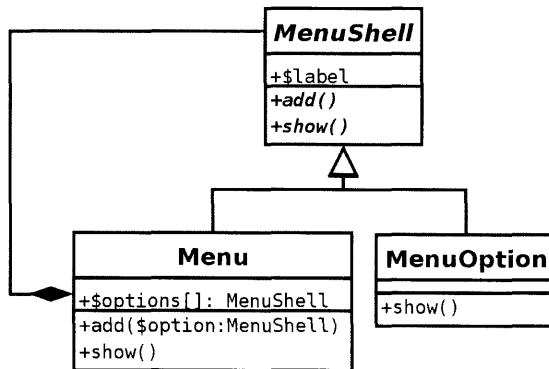


Figura 3.7 – Exemplo de Composite Pattern.

Depois de entendermos o funcionamento do Composite Pattern, iremos utilizá-lo para compor estruturas para representação de expressões. Para tanto, criaremos a classe `TCriteria`, que permitirá compor essas expressões. Essa classe basicamente irá nos oferecer o método `add()` que adicionará uma nova expressão a uma lista de expressões. A princípio, adicionaremos objetos do tipo `TFilter`, podendo formar uma expressão contendo diversos filtros. O segundo parâmetro do método `add()` é o tipo de operador lógico que será utilizado na junção das expressões. O operador default será o `AND` (`AND_OPERATOR`), mas poderemos utilizar também o `OR` (`OR_OPERATOR`). No diagrama mostrado na Figura 3.8, representamos essa composição de critérios.

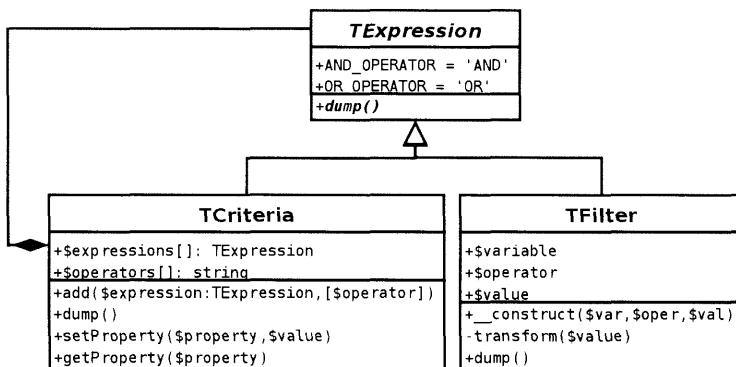


Figura 3.8 – Composite Pattern para expressões de critérios.

A grande vantagem do método `add()` está no fato de ele aceitar não somente objetos do tipo `TFilter`, mas também objetos `TCriteria`. Como um `TCriteria` pode ser formado por vários `TFilter`, temos aí um caminho para compor expressões recursivamente. Veja que o método aguarda por um parâmetro do tipo `TExpression`, ou seja, a superclasse-pai de `TCriteria` e de `TFilter`. Depois de receber o objeto, ele é armazenado em um vetor de objetos (`$expressions`), assim como o operador (`$operators`).

Depois de utilizarmos o método `add()` para adicionar filtros (`TFilter`) ou critérios (`TCriteria`) a um critério, fazemos uso do método `dump()` para converter essa cadeia de expressões no formato de string plana. O método `dump()` percorre a lista de expressões colhendo o retorno da execução do método `dump()` de cada expressão. Isto é possível porque todas as classes da hierarquia implementam o método `dump()`, uma vez que este é abstrato na classe-pai. Ao final, poderemos ter uma cadeia de expressões lógicas como a da Figura 3.9.

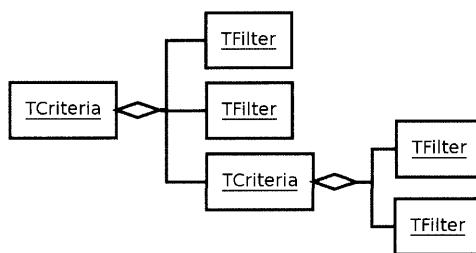


Figura 3.9 – Expressões compostas.

Quando um critério é utilizado em conjunto com algumas instruções como o `Select`, freqüentemente precisamos definir outras características como o ordenamento (`order by`) ou o intervalo da consulta (`offset` e `limit`). Para definir tais características, criaremos o método `setProperty()`, o qual receberá dois parâmetros: o nome da propriedade (`order, offset, limit`) e o seu respectivo valor. Para retornar o valor de uma propriedade, utilizaremos o método `getProperty()`. As propriedades serão armazenadas no array `$properties`.

TCriteria.class.php

```

<?php
/*
 * classe TCriteria
 * Esta classe provê uma interface utilizada para definição de critérios
 */
class TCriteria extends TExpression
{
    private $expressions; // armazena a lista de expressões
    private $operators; // armazena a lista de operadores
    private $properties; // propriedades do critério

    /*
     * método add()
     * adiciona uma expressão ao critério
     * @param $expression = expressão (objeto TExpression)
     * @param $operator = operador lógico de comparação
     */
}
  
```

```
public function add(TExpression $expression, $operator = self::AND_OPERATOR)
{
    // na primeira vez, não precisamos de operador lógico para concatenar
    if (empty($this->expressions))
    {
        unset($operator);
    }

    // agraga o resultado da expressão à lista de expressões
    $this->expressions[] = $expression;
    $this->operators[]   = $operator;
}

/*
 * método dump()
 * retorna a expressão final
 */
public function dump()
{
    // concatena a lista de expressões
    if (is_array($this->expressions))
    {
        foreach ($this->expressions as $i=> $expression)
        {
            $operator = $this->operators[$i];
            // concatena o operador com a respectiva expressão
            $result .= $operator. $expression->dump() . ' ';
        }
        $result = trim($result);
        return "({$result})";
    }
}

/*
 * método setProperty()
 * define o valor de uma propriedade
 * @param $property = propriedade
 * @param $value    = valor
 */
public function setProperty($property, $value)
{
    $this->properties[$property] = $value;
}

/*
 * método getProperty()
 * retorna o valor de uma propriedade
 * @param $property = propriedade
 */

```

```

public function getProperty($property)
{
    return $this->properties[$property];
}
?
?
```

3.3.6.3 Exemplo

Depois de criarmos essas classes para manipulação de critérios, criaremos um pequeno programa para exemplificar sua utilização. Neste ponto iremos utilizá-los de maneira isolada, o que não faz muito sentido inicialmente. Mais adiante, iremos utilizá-los em conjunto com as classes para manipulação de SQL (`Update`, `Delete` e `Select`), tornando mais clara sua função dentro do sistema.

Nos exemplos a seguir, temos critérios baseados nos operadores lógicos `OR` e `AND`. Lembre-se de que o operador default é o `AND`. Juntamente com esses operadores, podemos escrever operações contendo operadores como `>`, `<`, `>=`, `<=`, além de `IN`, `NOT IN`, entre outros. Como vimos na implementação da classe `TFilter`, seu método construtor já efetua a transformação do parâmetro `valor` (terceiro parâmetro) por meio do método privado `transform()`, o qual irá verificar o tipo de parâmetro (integer, string, array) e irá convertê-lo para fazer parte da instrução SQL. Veja que nos exemplos a seguir utilizamos arrays contendo números, strings, valores booleanos e também arrays. No final do script temos um exemplo contendo critérios compostos. Podemos compor diversos níveis aninhados de critérios, uma vez que o método `add()` aceita tanto um objeto do tipo `TFilter` quanto um objeto do tipo `TCriteria` como parâmetro.

A Figura 3.10 procura demonstrar a composição de objetos criada ao final do exemplo, no qual temos um critério composto de dois outros critérios. No nível superior, a ligação é realizada pelo operador `OR` e, em um nível abaixo, cada filtro é unido por um operador `AND`.

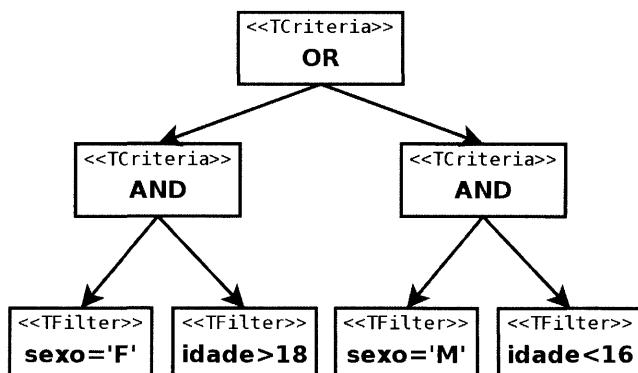


Figura 3.10 – Composição de critérios.

 criteria.php

```
<?php  
// carrega as classes necessárias  
include_once 'app.ado/TExpression.class.php';  
include_once 'app.ado/TCriteria.class.php';  
include_once 'app.ado/TFilter.class.php';  
  
// aqui vemos um exemplo de critério utilizando o operador lógico OR  
// a idade deve ser menor que 16 OU maior que 60  
$criteria = new TCriteria;  
$criteria->add(new TFilter('idade', '<', 16), TExpression::OR_OPERATOR);  
$criteria->add(new TFilter('idade', '>', 60), TExpression::OR_OPERATOR);  
echo $criteria->dump();  
echo " <br>\n";  
  
// aqui vemos um exemplo de critério utilizando o operador lógico AND  
// juntamente com os operadores de conjunto IN (dentro do conjunto) e NOT IN (fora do conjunto)  
// a idade deve estar dentro do conjunto (24,25,26) e deve estar fora do conjunto (10)  
$criteria = new TCriteria;  
$criteria->add(new TFilter('idade','IN', array(24,25,26)));  
$criteria->add(new TFilter('idade','NOT IN', array(10)));  
echo $criteria->dump();  
echo " <br>\n";  
  
// aqui vemos um exemplo de critério utilizando o operador de comparação LIKE  
// o nome deve iniciar por "pedro" ou deve iniciar por "maria"  
$criteria = new TCriteria;  
$criteria->add(new TFilter('nome', 'like', 'pedro%'), TExpression::OR_OPERATOR);  
$criteria->add(new TFilter('nome', 'like', 'maria%'), TExpression::OR_OPERATOR);  
echo $criteria->dump();  
echo " <br>\n";  
  
// aqui vemos um exemplo de critério utilizando os operadores "=" e IS NOT  
// neste caso, o telefone não pode conter valor nulo (IS NOT NULL)  
// e o sexo deve ser feminino (sexo='F')  
$criteria = new TCriteria;  
$criteria->add(new TFilter('telefone', 'IS NOT', NULL));  
$criteria->add(new TFilter('sexo', '=' , 'F'));  
echo $criteria->dump();  
echo " <br>\n";  
  
// aqui vemos o uso dos operadores de comparação IN e NOT IN juntamente com  
// conjuntos de strings. Neste caso, a UF deve estar entre (RS, SC e PR) E  
// não deve estar entre (AC e PI).  
$criteria = new TCriteria;  
$criteria->add(new TFilter('UF', 'IN', array('RS', 'SC', 'PR')));
```

```
$criteria->add(new TFilter('UF', 'NOT IN', array('AC', 'PI')));
echo $criteria->dump();
echo "<br>\n";

// neste caso temos o uso de um critério composto
// o primeiro critério aponta para sexo='F'
// (sexo feminino) e idade > 18 (maior de idade)
$criteria1 = new TCriteria;
$criteria1->add(new TFilter('sexo', '=', 'F'));
$criteria1->add(new TFilter('idade', '>', '18'));

// o segundo critério aponta para sexo=M (masculino)
// e idade < 16 (menor de idade)
$criteria2 = new TCriteria;
$criteria2->add(new TFilter('sexo', '=', 'M'));
$criteria2->add(new TFilter('idade', '<', '16'));

// agora juntamos os dois critérios utilizando o operador lógico OR (ou). O resultado
// deve conter "mulheres maiores de 18" OU "homens menores de 16"
$criteria = new TCriteria;
$criteria->add($criteria1, TExpression::OR_OPERATOR);
$criteria->add($criteria2, TExpression::OR_OPERATOR);
echo $criteria->dump();
echo "<br>\n";
?>
```

Resultado:

```
(idade < 16 OR idade > 60)
(idade IN (24,25,26) AND idade NOT IN (10))
(nome like 'pedro%' OR nome like 'maria%')
(telefone IS NOT NULL AND sexo = 'F')
(UF IN ('RS','SC','PR') AND UF NOT IN ('AC','PI'))
((sexo = 'F' AND idade > '18') OR (sexo = 'M' AND idade < '16'))
```

3.3.7 Instruções SQL

3.3.7.1 Introdução

Como já vimos anteriormente, as instruções SQL compartilham um comportamento em comum entre elas. Tal comportamento será implementado por uma superclasse que será herdada por todas as instruções SQL. Para iniciar, essa classe terá o método `setEntity()` para definir qual é a tabela do banco de dados (entidade) sobre a qual a instrução SQL agirá. Ademais, temos o método `getEntity()` que irá justamente retornar o nome da tabela. Marcaremos esses métodos como `final`, o que impedirá que os

mesmos possam ser reescritos nas subclasses. Desta forma, a implementação será definitiva para todas as subclasses.

Como cada instrução SQL é composta por um critério de seleção de dados, precisamos construir o método `setCriteria()` para atribuir este critério (objeto `TCriteria`) à instrução. O critério será armazenado na propriedade `$criteria`, que é `protected`, ou seja, visível em todas as classes dessa hierarquia. Por último, definimos o método `getInstruction()`. Veja que este método não possui um corpo contendo um código. Esta definição de método é abstrata, o que força todas as classes-filha a terem o método `getInstruction()`. Isto é um claro sinal de polimorfismo, tendo em vista que cada classe-filha terá um método `getInstruction()` implementado de forma diferente. No polimorfismo, vemos a ocorrência de várias implementações em diferentes classes de uma mesma hierarquia, sob uma interface comum.

A classe `TSqlInstruction` será marcada como `abstract` porque não é possível existir instâncias dessa classe dentro da aplicação. Tal classe existe somente por propósitos estruturais de hierarquia (para servir de base para as demais). Os objetos que irão existir na aplicação são as instâncias de suas subclasses, as quais serão criadas logo a seguir.

`TSqlInstruction.class.php`

```
<?php
/*
 * classe TSqlInstruction
 * Esta classe provê os métodos em comum entre todas instruções
 * SQL (SELECT, INSERT, DELETE e UPDATE)
 */
abstract class TSqlInstruction
{
    protected $sql;        // armazena a instrução SQL
    protected $criteria;   // armazena o objeto critério
    /*
     * método setEntity()
     * define o nome da entidade (tabela) manipulada pela instrução SQL
     * @param $entity = tabela
     */
    final public function setEntity($entity)
    {
        $this->entity = $entity;
    }
    /*
     * método getEntity()
     * retorna o nome da entidade (tabela)
     */
}
```

```
final public function getEntity()
{
    return $this->entity;
}

/*
 * método setCriteria()
 * Define um critério de seleção dos dados através da composição de um objeto
 * do tipo TCriteria, que oferece uma interface para definição de critérios
 * @param $criteria = objeto do tipo TCriteria
 */
public function setCriteria(TCriteria $criteria)
{
    $this->criteria = $criteria;
}

/*
 * método getInstruction()
 * declarando-o como <abstract> obrigamos sua declaração nas classes filhas,
 * uma vez que seu comportamento será distinto em cada uma delas, configurando polimorfismo.
 */
abstract function getInstruction();
}

?>
```

3.3.8 Insert

3.3.8.1 Introdução

Criaremos agora a primeira subclasse de `TSqlInstruction`. Para inserir dados no banco de dados criaremos a classe `TSqlInsert`, a qual será uma classe `final`, ou seja, ela não poderá servir de base (superclasse) para a construção de uma nova classe. Ela irá automaticamente herdar todos os métodos definidos em `TSqlInstruction` e, além disso, irá implementar mais um método: o `setRowData()`, o qual define um array associativo contendo os dados de uma linha a ser inserida na tabela do banco de dados. Ele pode ser invocado diversas vezes, de modo que a cada vez ele irá atribuir um valor (`$value`) a uma determinada coluna da tabela (`$column`). O método `setRowData()` também verifica o tipo de dado passado como parâmetro, adicionando aspas, quando necessário (strings), e, ainda, verifica quando se trata de um valor `NULL` ou `boolean`.

Como o comando de `INSERT` é o único que não precisa de um critério de seleção de dados (que irá se converter em uma cláusula `WHERE`), temos de gerar uma exceção sempre que o método `setCriteria()` for executado na classe `TSqlInsert`, isto porque ele está definido em sua superclasse. Para finalizar, iremos finalmente definir o compor-

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`.

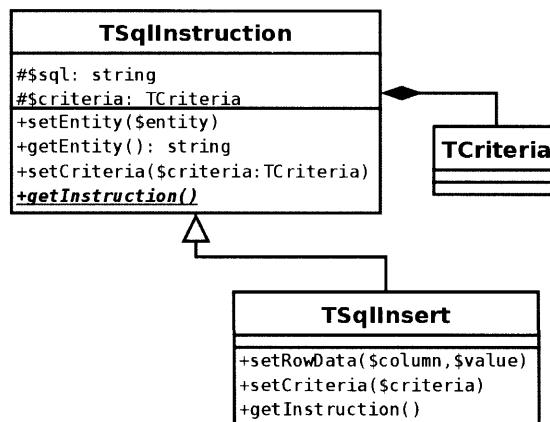


Figura 3.11 – Classe `TSqIInsert`.

`TSqIInsert.class.php`

```

<?php
/*
 * classe TSqIInsert
 * Esta classe provê meios para manipulação de uma instrução de INSERT no banco de dados
 */
final class TSqIInsert extends TSqIInstruction
{
    /*
     * 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'";
        }
    }
}
  
```

```
else if (is_bool($value))
{
    // caso seja um boolean
    $this->columnValues[$column] = $value ? 'TRUE': 'FALSE';
}
else if (isset($value))
{
    // caso seja outro tipo de dado
    $this->columnValues[$column] = $value;
}
else
{
    // caso seja NULL
    $this->columnValues[$column] = "NULL";
}

/*
 * método setCriteria()
 * não existe no contexto desta classe, logo, irá lançar um erro se for executado
 */
public function setCriteria($criteria)
{
    // lança o erro
    throw new Exception("Cannot call setCriteria from " . __CLASS__);
}

/*
 * método getInstruction()
 * retorna a instrução de INSERT em forma de string.
 */
public function getInstruction()
{
    $this->sql = "INSERT INTO {$this->entity} (";
    // monta uma string contendo os nomes de colunas
    $columns = implode(', ', array_keys($this->columnValues));
    // monta uma string contendo os valores
    $values = implode(', ', array_values($this->columnValues));
    $this->sql .= $columns . ')';
    $this->sql .= " values ({$values})";

    return $this->sql;
}
?>
```

3.3.8.2 Exemplo

No primeiro exemplo, utilizaremos a classe `TSqlInsert` para executar a instrução `INSERT`. Iremos inserir `id`, `nome`, `fone`, `nascimento`, `sexo`, `serie` e `mensalidade` na tabela `aluno`. Em seguida, definimos a tabela a ser manipulada pelo método `setEntity()` e correlacionamos os dados que serão gravados em cada coluna pelo método `setRowData()`. Ao final, exibimos a instrução resultante pelo método `getInstruction()`.

Em cada exemplo criado, será necessário carregar as classes, sempre que elas forem necessárias. Para tanto, faremos uso da função `__autoload()`, que é executada sempre que um objeto é instanciado pela primeira vez.

insert.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";
    }
}
// define o LOCALE do sistema, para permitir ponto decimal.
// PS: no Windows, usar "english"
setlocale(LC_NUMERIC, 'POSIX');

// cria uma instrução de INSERT
$sql = new TSqlInsert;
// define o nome da entidade
$sql->setEntity('aluno');
// atribui o valor de cada coluna
$sql->setRowData('id',            3);
$sql->setRowData('nome',          'Pedro Cardoso');
$sql->setRowData('fone',          '(88) 4444-7777');
$sql->setRowData('nascimento',    '1985-04-12');
$sql->setRowData('sexo',          'M');
$sql->setRowData('serie',         '4');
$sql->setRowData('mensalidade',   280.40);

// processa a instrução SQL
echo $sql->getInstruction();
echo "<br>\n";
?>
```

 **Resultado:**

```
INSERT INTO aluno (id, nome, fone, nascimento, sexo, serie, mensalidade)
values (3, 'Pedro Cardoso', '(88) 4444-7777', '1985-04-12', 'M', '4', 280.4)
```

3.3.9 Update

3.3.9.1 Introdução

Nesta seção, trataremos de implementar a classe para manipulação de instruções UPDATE, responsável pela alteração de dados já existentes no banco de dados. Esta classe, que se chamará `TSqlUpdate`, irá oferecer o método `setRowData()`, da forma idêntica ao da classe `TSqlInsert`, definindo um array associativo contendo os dados que serão armazenados no banco de dados. Por fim, a classe `TSqlUpdate` define o método `getInstruction()`. O método `getInstruction()` constrói a string contendo a instrução SQL, para tanto utiliza-se das informações definidas pela função `setRowData()`, para montar os pares de atribuição de valores (`coluna1=valor1, coluna2=valor2`), e também do nome da entidade, definido pelo método `setEntity()` da superclasse. Para completar, a parte mais importante, a cláusula `WHERE`, é retornada pelo objeto agregado (`$this->criteria`). Este objeto é atribuído pela função `setCriteria()` da superclasse, deve ser um objeto do tipo `TCriteria` e irá retornar a cláusula `WHERE` pelo seu método `dump()`. Veja a seguir o diagrama que representa a classe `TSqlUpdate`.

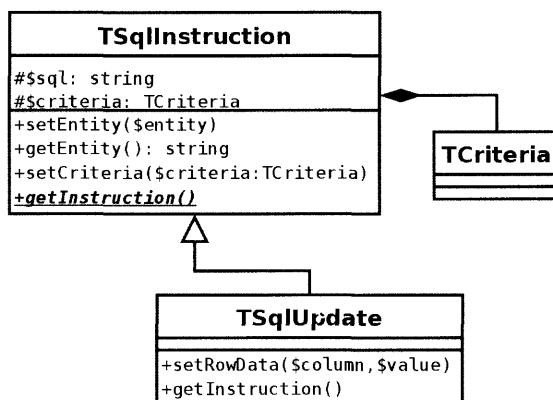


Figura 3.12 – Classe `TSqlUpdate`.

 **TSqlUpdate.class.php**

```
<?php
/*
 * classe TSqlUpdate
 * Esta classe provê meios para manipulação de uma instrução de UPDATE no banco de dados
 */
```

```
final class TSqlUpdate extends TSqlInstruction
{
    /*
     * método setRowData()
     * Atribui valores à determinadas colunas no banco de dados que serão modificadas
     * @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] = "'$value'";
        }
        else if (is_bool($value))
        {
            // caso seja um boolean
            $this->columnValues[$column] = $value ? 'TRUE': 'FALSE';
        }
        else if (isset($value))
        {
            // caso seja outro tipo de dado
            $this->columnValues[$column] = $value;
        }
        else
        {
            // caso seja NULL
            $this->columnValues[$column] = "NULL";
        }
    }

    /*
     * método getInstruction()
     * retorna a instrução de UPDATE em forma de string.
     */
    public function getInstruction()
    {
        // monsta a string de UPDATE
        $this->sql = "UPDATE {$this->entity}";
    }
}
```

```
// monta os pares: coluna=valor, ...
if ($this->columnValues)
{
    foreach ($this->columnValues as $column => $value)
    {
        $set[] = "{$column} = {$value}";
    }
}
$this->sql .= ' SET ' . implode(', ', $set);

// retorna a cláusula WHERE do objeto $this->criteria
if ($this->criteria)
{
    $this->sql .= ' WHERE ' . $this->criteria->dump();
}
return $this->sql;
}
}
?>
```

3.3.9.2 Exemplo

No próximo exemplo, alteraremos os dados recém-inseridos pela classe `TSqlUpdate`. Neste caso, precisamos criar um critério de seleção; do contrário iremos alterar todos os registros da tabela. Neste critério de seleção, estamos filtrando somente o `id=3`. Novamente utilizamos o método `setRowData()` para definir quais informações serão alteradas na tabela. O método `setCriteria()` relaciona o critério recém-criado ao `UPDATE` e, no final, utilizamos o método `getInstruction()` para exibir a instrução SQL na tela.

update.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 critério de seleção de dados
$criteria = new TCriteria;
$criteria->add(new TFilter('id', '=', '3'));

// cria instrução de UPDATE
$sql = new TSqlUpdate;
// define a entidade
$sql->setEntity('aluno');
// atribui o valor de cada coluna
$sql->setRowData('nome',      'Pedro Cardoso da Silva');
$sql->setRowData('rua',        'Machado de Assis');
$sql->setRowData('fone',       '(88) 5555');
// define o critério de seleção de dados
$sql->setCriteria($criteria);

// processa a instrução SQL
echo $sql->getInstruction();
echo "<br>\n";
?>
```

Resultado:

```
UPDATE aluno SET
    nome = 'Pedro Cardoso da Silva',
    rua = 'Machado de Assis',
    fone = '(88) 5555'
WHERE id= '3'
```

3.3.10 Delete

3.3.10.1 Introdução

Para manipular instruções `DELETE`, criaremos a classe `TSqlDelete`, a qual é extremamente simples, pois a maioria dos recursos necessários para seu funcionamento já estão implementados pela superclasse `TSqlInstruction`, como é o caso do método `setEntity()`, ou mesmo pela classe agregada `TCriteria`, que define os critérios de seleção (`WHERE`) para os registros a serem excluídos. Dessa forma, resta a classe definir o método `getInstruction()`, a qual construirá a instrução SQL, contendo a instrução `DELETE`. Para isso, ela utiliza o nome da entidade (`$this->entity`) e o objeto criteria (`$this->criteria`), que retorna toda a cláusula `WHERE` por seu método `dump()`. Veja a seguir o diagrama que representa a classe `TSqlDelete`.

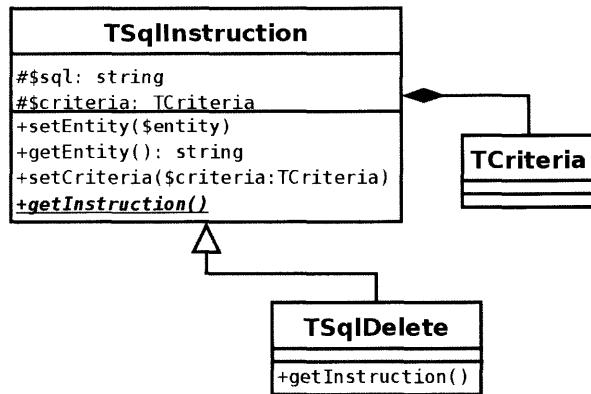


Figura 3.13 – Classe TSqIDelete.

TSqIDelete.class.php

```

<?php
/*
 * classe TSqIDelete
 * Esta classe provê meios para manipulação de uma instrução de DELETE no banco de dados
 */
final class TSqIDelete extends TSqIInstruction
{
    /*
     * método getInstruction()
     * retorna a instrução de DELETE em forma de string.
     */
    public function getInstruction()
    {
        // monta a string de DELETE
        $this->sql = "DELETE FROM {$this->entity}";

        // retorna a cláusula WHERE do objeto $this->criteria
        if ($this->criteria)
        {
            $expression = $this->criteria->dump();
            if ($expression)
            {
                $this->sql .= ' WHERE ' . $expression;
            }
        }
        return $this->sql;
    }
}
?>
  
```

3.3.10.2 Exemplo

Neste exemplo, estamos excluindo o registro `id=3` por meio da classe `TSqlDelete`. Para isso, estamos novamente criando, com o uso da classe `TCriteria`, um critério de seleção de registros. Veja o resultado a seguir:

delete.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 critério de seleção de dados
$criteria = new TCriteria;
$criteria->add(new TFilter('id', '=', '3'));

// cria instrução de DELETE
$sql = new TSqlDelete;
// define a entidade
$sql->setEntity('aluno');
// define o critério de seleção de dados
$sql->setCriteria($criteria);

// processa a instrução SQL
echo $sql->getInstruction();
echo "<br>\n";
?>
```

Resultado:

```
DELETE FROM aluno WHERE (id = '3')
```

3.3.11 Select

3.3.11.1 Introdução

A instrução de `SELECT` é, sem dúvida, a mais complexa de todas, por envolver uma maior riqueza de detalhes em suas cláusulas de seleção. Para começar, implementaremos

o método `addColumn()` responsável por adicionar uma coluna a ser retornada pela instrução de `SELECT`, por meio da adição do parâmetro recebido em um array de colunas. Esse método poderá ser executado quantas vezes forem necessárias (quantas colunas desejarmos retornar). O método `getInstruction()`, por sua vez, irá se basear primeiramente nas colunas adicionadas pelo método `addColumn()` para construir a instrução de `SELECT`; em segundo lugar, irá obter a entidade definida pelo método `setEntity()` para montar a cláusula `FROM`. O objeto `TCriteria`, acessado via (`$this->criteria`), será utilizado para retornar toda cláusula `WHERE`. O critério ainda irá retornar informações de ordenamento (`Order by`) e do intervalo da consulta (`Limit` e `Offset`) pelo método `getProperty()`. Veja a seguir o diagrama que representa a classe `TSqlSelect`:

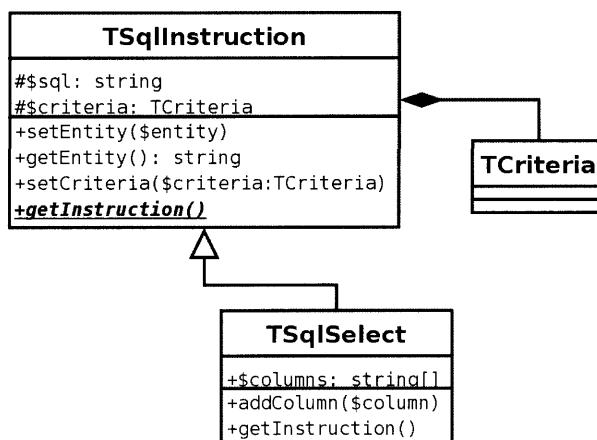


Figura 3.14 – Classe `TSqlSelect`.

`TSqlSelect.class.php`

```

<?php
/*
 * classe TSqlSelect
 * Esta classe provê meios para manipulação de uma instrução de SELECT no banco de dados
 */
final class TSqlSelect extends TSqlInstruction
{
    private $columns; // array de colunas a serem retornadas.

    /*
     * método addColumn
     * adiciona uma coluna a ser retornada pelo SELECT
     * @param $column = coluna da tabela
     */
    public function addColumn($column)
    {

```

```
// adiciona a coluna no array
$this->columns[] = $column;
}

/*
 * método getInstruction()
 * retorna a instrução de SELECT em forma de string.
 */
public function getInstruction()
{
    // monta a instrução de SELECT
    $this->sql = 'SELECT ';
    // monta string com os nomes de colunas
    $this->sql .= implode(', ', $this->columns);
    // adiciona na cláusula FROM o nome da tabela
    $this->sql .= ' FROM ' . $this->entity;

    // obtém a cláusula WHERE do objeto criteria.
    if ($this->criteria)
    {
        $expression = $this->criteria->dump();
        if ($expression)
        {
            $this->sql .= ' WHERE ' . $expression;
        }
        // obtém as propriedades do critério
        $order = $this->criteria->getProperty('order');
        $limit = $this->criteria->getProperty('limit');
        $offset= $this->criteria->getProperty('offset');

        // obtém a ordenação do SELECT
        if ($order)
        {
            $this->sql .= ' ORDER BY ' . $order;
        }
        if ($limit)
        {
            $this->sql .= ' LIMIT ' . $limit;
        }
        if ($offset)
        {
            $this->sql .= ' OFFSET ' . $offset;
        }
    }
    return $this->sql;
}
?>
```

3.3.11.2 Exemplo

Criaremos agora um exemplo para recuperar registros do banco de dados. Primeiramente utilizaremos a classe `TCriteria` para declarar um critério de seleção de registros. Note que, além de utilizarmos o método `add()` da classe `TCriteria` para concatenar expressões lógicas `AND` e `OR`, utilizaremos o método `setProperty()` para definir um intervalo de consulta (`OFFSET` e `LIMIT`) e também o ordenamento desta. No exemplo a seguir, estamos selecionando `nome` e `fone` da tabela `aluno` sempre que o nome iniciar por “maria” e a cidade iniciar por “Porto”.

select.php

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

// cria critério de seleção de dados
$criteria = new TCriteria;
$criteria->add(new TFilter('nome', 'like', 'maria%'));
$criteria->add(new TFilter('cidade','like ', 'Porto%'));

// define o intervalo de consulta
$criteria->setProperty('offset', 0);
$criteria->setProperty('limit', 10);
// define o ordenamento da consulta
$criteria->setProperty('order', 'nome');

// cria instrução de SELECT
$sql = new TSqlSelect;
// define o nome da entidade
$sql->setEntity('aluno');
// acrescenta colunas à consulta
$sql->addColumn('nome');
$sql->addColumn('fone');
// define o critério de seleção de dados
$sql->setCriteria($criteria);
```

```
// processa a instrução SQL
echo $sql->getInstruction();
echo "<br>\n";
?>
```

Resultado:

```
SELECT nome,fone
  FROM aluno
 WHERE (nome like 'maria%' AND cidade like  'Porto%')
 ORDER BY nome
 LIMIT 10
```

Até o momento utilizamos critérios de seleção simples, sem expressões lógicas aninhadas. No próximo exemplo, criaremos um critério de seleção múltiplo, ligado por uma expressão OR, no qual cada expressão lógica é formada por uma outra expressão. Para se atingir tal resultado é necessário utilizar três objetos de critério: um para cada expressão lógica, e um terceiro para agregar as duas primeiras. No exemplo estamos selecionando todas as alunas da terceira série OU todos os alunos da quarta série.

select2.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 critério de seleção de dados
$criteria1 = new TCriteria;
// seleciona todas as meninas (F)
// que estão na terceira (3) série
$criteria1->add(new TFilter('sexo', '=', 'F'));
$criteria1->add(new TFilter('serie', '=', '3'));

// seleciona todos os meninos (M)
// que estão na quarta (4) série
$criteria2 = new TCriteria;
```

```
$criteria2->add(new TFilter('sexo', '=', 'M'));
$criteria2->add(new TFilter('serie', '=', '4'));

// agora juntamos os dois critérios utilizando
// o operador lógico OR (ou). O resultado deve conter
// "meninas da 3a serie OU meninos da 4a série"
$criteria = new TCriteria;
$criteria->add($criteria1, TExpression::OR_OPERATOR);
$criteria->add($criteria2, TExpression::OR_OPERATOR);
// define o ordenamento
$criteria->setProperty('order', 'nome');

// cria instrução de SELECT
$sql = new TSqlSelect;
// define o nome da entidade
$sql->setEntity('aluno');
// acrescenta todas colunas à consulta
$sql->addColumn('*');
// define o critério de seleção de dados
$sql->setCriteria($criteria);

// processa a instrução SQL
echo $sql->getInstruction();
echo "<br>\n";
?>
```

 **Resultado:**

```
SELECT * FROM aluno
WHERE ((sexo = 'F' AND serie = '3')
       OR (sexo = 'M' AND serie = '4'))
ORDER BY nome
```

3.3.12 Conexão com banco de dados

Vimos anteriormente que a utilização da biblioteca PDO traz inúmeras vantagens no desenvolvimento de aplicações, uma vez que deixamos de utilizar comandos específicos de um determinado sistema gerenciador de bancos de dados (SGBD) para utilizar uma interface orientada a objetos unificada, na qual podemos a qualquer momento alterar a tecnologia de banco de dados, sem acarretar modificações no código-fonte da aplicação.

Ainda assim temos a linha de conexão ao banco de dados, na qual instanciamos o objeto PDO (`new PDO`). Essa linha contém as informações específicas de cada banco de dados, como o tipo, o seu nome, o usuário e a senha. Não é recomendável ter

espalhados ao longo do código-fonte de uma aplicação esses detalhes de conexão. Imagine uma situação em que você desenvolveu toda a aplicação para MySQL, mas um importante cliente tem como política utilizar somente PostgreSQL, ou você precisa implantar o sistema em uma empresa que só possui DBAs Oracle para dar suporte ao banco de dados. O que deve ser feito nessa ocasião? Imagine que você possui esses detalhes de conexão com banco de dados em cada um dos 240 programas que fazem parte de seu sistema. Sempre que você precisou conectar-se ao banco de dados para inserir, excluir ou para listar alguma informação, lá estava você a copiar e a colar a linha de conexão, passando informações como usuário e senha novamente. Isso é muito ruim, porque se o programa for escrito dessa forma, para se adaptar a outro sistema de banco de dados você terá de rever cada um desses arquivos, para alterar as strings de conexão.

Uma das soluções para esse problema é criar um ponto central de conexão com o banco de dados, tornando mais simples a tarefa de gerenciar esse tipo de informação. Para tanto, implementaremos um design pattern conhecido como “Factory”.

3.3.12.1 Factory Pattern

Factory pode ser uma classe ou um método responsável pela geração de objetos (criação das instâncias). Implementaremos um Simple Factory, que geralmente possui um bloco de comandos `SWITCH` ou `IF` para tomar a decisão sobre qual classe instanciar.

O padrão Factory existe para esconder os detalhes da criação de um grupo de objetos com características semelhantes. Factory centraliza a geração de objetos, fornecendo uma interface única para criação das instâncias. É uma técnica importante pois evita que tenhamos essas instâncias espalhadas por diversos programas diferentes ao fornecer um ponto central, facilitando a manutenção do código.

Para implementar o padrão Factory, criaremos a classe `TConnection`, cujo papel será instanciar um objeto PDO de acordo com as informações de conexão passadas (driver, usuário, senha). Além disso, faremos algo mais sofisticado, gravando essas informações em arquivos de configuração INI no disco rígido. Dessa forma, nossa aplicação não tomará conhecimento de qual tecnologia de banco de dados estará rodando e poderemos a qualquer momento alterar o sistema gerenciador de banco de dados (PostgreSQL, Oracle, MySQL, DB2), simplesmente alterando os arquivos de configuração INI. A classe `TConnection` terá o método `open()`, o qual irá receber o nome de um dos arquivos de configuração, irá ler as informações nele contida (usuário, senha, driver etc.) e instanciar o objeto PDO adequado. Caso o arquivo de configuração não seja encontrado, uma exceção será lançada.

Agora, criaremos os arquivos de configuração para cada um dos dois bancos de dados criados. Primeiro o arquivo para o banco de dados PostgreSQL e posteriormente

o arquivo com os dados de conexão para o banco em MySQL. Nesses arquivos, que serão armazenados na pasta `app.config`, teremos variáveis para indicar localização do banco de dados (`host`), o nome do usuário (`name`) com permissões de acessar o banco, juntamente com sua senha (`pass`) e o tipo de banco de dados utilizado (`type`).

pg_livro.ini

```
host = localhost
name = livro
user = postgres
pass = nonono
type = pgsql
```

my_livro.ini

```
host = localhost
name = livro
user = root
pass = mysql
type = mysql
```

Agora, criaremos a classe `TConnection`. Note que os objetos são instanciados pelo método estático `open()`. Não faz sentido criar instâncias da classe `TConnection`, uma vez que ela só existe para prover objetos PDO por meio da chamada de seu método estático `open()`. Como não existirão objetos do tipo `TConnection`, preferimos marcar o método construtor da mesma como `private`, o que significa que este método só poderia ser chamado dentro do escopo da própria classe, evitando que algum programador execute por engano `new TConnection`.

Para que os objetos PDO reportem exceções (`PDOException`) quando ocorrerem erros de SQL, é necessário chamar o método `setAttribute()`, com os parâmetros apropriados. Como esta operação é comum a qualquer objeto do tipo PDO, podemos colocar juntamente com o código que irá instanciar os objetos. Esta classe foi suficientemente testada com os bancos de dados PostgreSQL, MySQL e Sqlite; para os demais, não é garantido o funcionamento. Veja na Figura 3.15 como fica nossa estrutura de diretório.

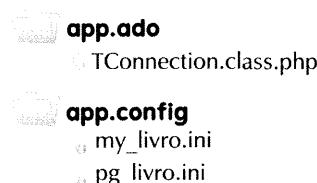


Figura 3.15 – Estrutura do diretório `app.ado` e `app.config`.

 TConnection.class.php

```
<?php
/*
 * classe TConnection
 * gerencia conexões com bancos de dados através de arquivos de configuração.
 */
final class TConnection
{
    /*
     * método __construct()
     * não existirão instâncias de TConnection, por isto estamos marcando-o como private
     */
    private function __construct() {}

    /*
     * método open()
     * recebe o nome do banco de dados e instancia o objeto PDO correspondente
     */
    public static function open($name)
    {
        // verifica se existe arquivo de configuração para este banco de dados
        if (file_exists("app.config/{$name}.ini"))
        {
            // lê o INI e retorna um array
            $db = parse_ini_file("app.config/{$name}.ini");
        }
        else
        {
            // se não existir, lança um erro
            throw new Exception("Arquivo '$name' não encontrado");
        }

        // lê as informações contidas no arquivo
        $user  = $db['user'];
        $pass  = $db['pass'];
        $name  = $db['name'];
        $host  = $db['host'];
        $type  = $db['type'];

        // descobre qual o tipo (driver) de banco de dados a ser utilizado
        switch ($type)
        {
            case 'pgsql':
                $conn = new PDO("pgsql:dbname={$name};user={$user}; password={$pass};host={$host}");
                break;
        }
    }
}
```

```
case 'mysql':
    $conn = new PDO("mysql:host={$host};port=3307;dbname={$name}", $user, $pass);
    break;
case 'sqlite':
    $conn = new PDO("sqlite:{$name}");
    break;
case 'ibase':
    $conn = new PDO("firebird:dbname={$name}", $user, $pass);
    break;
case 'oci8':
    $conn = new PDO("oci:dbname={$name}", $user, $pass);
    break;
case 'mssql':
    $conn = new PDO("mssql:host={$host},1433;dbname={$name}", $user, $pass);
    break;
}

// define para que o PDO lance exceções na ocorrência de erros
$conn->setAttribute(PDO::ATTR_ERRMODE, PDO::ERRMODE_EXCEPTION);

// retorna o objeto instanciado.
return $conn;
}
?>
```

A função `parse_ini_file()` lê um arquivo no formato INI e retorna um array contendo as suas variáveis. O método `open()` descobre primeiro o valor da variável `$type` (`pgsql`, `mysql`, `sqlite`, `ibase`, `oci8`, `mssql`) para posteriormente instanciar o objeto PDO de acordo com cada driver.

Escreveremos agora um exemplo de conexão ao banco de dados, utilizando a classe criada em vez de instanciar explicitamente os objetos PDO no código da aplicação. Veja que no mesmo programa conectamos em dois sistemas de bancos de dados diferentes (MySQL e PostgreSQL), mudando somente o nome do arquivo INI (nossa DataSource) na chamada do método estático `TConnection::open()`.

Na Figura 3.16 temos um diagrama que explica a interação existente entre o programa e as classes `TConnection` e `PDO`. Note que o programa executa o método `open()` da classe `TConnection` para obter uma conexão, que é um objeto `PDO`. Em seguida, o programa pode executar diversos métodos dessa classe até o momento de encerrar a conexão com o banco de dados, atribuindo o valor `NULL` ao objeto `PDO`.

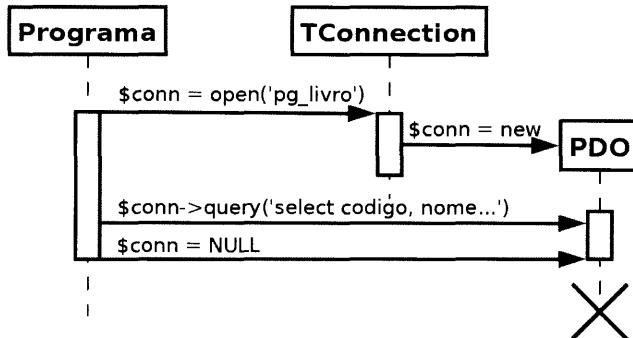


Figura 3.16 – Interação entre as classes TConnection e PDO.

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 instanciada 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');

```

```
// cria critério de seleção de dados
$criteria = new TCriteria;
// obtém a pessoa de código "1"
$criteria->add(new TFilter('codigo', '=', '1'));
// atribui o critério de seleção de dados
$sql->setCriteria($criteria);

try
{
    // abre conexão com a base my_livro (mysql)
    $conn = TConnection::open('my_livro');

    // executa a instrução SQL
    $result = $conn->query($sql->getInstruction());
    if ($result)
    {
        $row = $result->fetch(PDO::FETCH_ASSOC);
        // exibe os dados resultantes
        echo $row['codigo'] . ' - ' . $row['nome'] . "\n";
    }
    // fecha a conexão
    $conn = null;
}
catch (PDOException $e)
{
    // exibe a mensagem de erro
    print "Erro!:" . $e->getMessage() . "<br/>";
    die();
}

try
{
    // abre conexão com a base pg_livro (postgres)
    $conn = TConnection::open('pg_livro');

    // executa a instrução SQL
    $result = $conn->query($sql->getInstruction());
    if ($result)
    {
        $row = $result->fetch(PDO::FETCH_ASSOC);
        // exibe os dados resultantes
        echo $row['codigo'] . ' - ' . $row['nome'] . "\n";
    }
    // fecha a conexão
    $conn = null;
}
```

```
catch (Exception $e)
{
    // exibe a mensagem de erro
    print "Erro!: " . $e->getMessage() . "<br/>";
    die();
}
?>
```

Resultado:

```
1 - Érico Veríssimo
1 - Érico Veríssimo
```

Se ocorrer algum erro durante a conexão com o banco de dados MySQL, o primeiro a ser utilizado neste exemplo, será exibido o seguinte erro:

```
Erro!: SQLSTATE[HY000] [2002] Can't connect to local MySQL server through socket '/tmp/mysql.sock' (2)
```

Se ocorrer algum erro durante a conexão com o PostgreSQL, que é o segundo banco de dados a ser conectado neste exemplo, será exibido:

```
Erro!: SQLSTATE[08006] [7] could not connect to server: Conexão recusada
      Is the server running on host "localhost" and accepting
      TCP/IP connections on port 5432?
```

3.3.13 Controle de transações

Uma transação representa uma interação entre a aplicação e o sistema de banco de dados tratada de forma única e independente. Pense em uma transação financeira de transferência de fundos. Este tipo de transação necessita de um grande cuidado pois, do contrário, uma conta pode ser debitada e a outra não ser creditada, ou vice-versa. Uma transação é uma seqüência de trabalho que tem um ponto de início e de fim bem definidos. Uma transação tem algumas propriedades, dentre as quais podemos destacar:

- **Atomicidade** – é a propriedade que garante que todas as tarefas da transação sejam cumpridas, ou a mesma seja cancelada como um todo.
- **Consistência** – é a propriedade que garante que o banco de dados esteja em um estado íntegro depois de a transação ser realizada, tal qual estava antes de a mesma iniciar, sem quebrar nenhuma integridade referencial, por exemplo.
- **Isolamento** – é a propriedade que garante que o resultado de uma transação só seja visível para outras transações no momento em que ela é finalizada com sucesso, ou seja, a transação é isolada de outras operações.

- **Durabilidade** – é a propriedade que garante que a transação seja persistida assim que finalizada, ou seja, não será desfeita ou perdida mesmo na ocorrência de falhas do sistema.

No desenvolvimento de aplicações de negócios é muito importante a utilização de transações para garantir as propriedades anteriormente descritas quando processamos e armazenamos dados interdependentes no banco de dados após sucessivas interações com o mesmo. Implementaremos o conceito de transações de forma transparente e que nos permita realizar inúmeras operações no banco de dados, podendo aproveitar a mesma conexão durante todo o processo.

Quando aberta, uma transação deve disponibilizar uma conexão com o banco de dados à aplicação. Tal conexão deve ser visível por diferentes partes do sistema (métodos, funções etc.), para que as diferentes partes possam enviar comandos ao banco de dados e os comandos sejam corretamente encapsulados pela transação atual.

Para que um objeto seja visível ao longo de diferentes partes do sistema uma das formas utilizadas é tornar esse objeto global. Utilizar uma variável global é uma técnica condenável sob o paradigma da orientação a objetos pois ela pode a qualquer momento ser excluída ou sobreposta, uma vez que seu acesso não é protegido, colocando por terra o encapsulamento.

Uma forma mais consistente de se disponibilizar um objeto para acesso global é por meio de uma propriedade estática. Uma propriedade estática de uma classe também é global, mas podemos utilizar a estrutura da própria classe para proteger tal objeto, fazendo com que a própria classe garanta que não irão se construir outras instâncias, impedindo que se execute seu método construtor, e também impedindo que se sobreponha tal objeto.

Existem alguns tipos de classes para as quais não faz sentido ter mais de uma instância na aplicação – uma classe para gerenciar as preferências da aplicação, por exemplo. A existência de várias instâncias dessa classe poderia gerar uma inconsistência, pois, em um determinado momento, cada objeto poderia conter valores diferentes, sendo que, em se tratando de configuração do sistema, temos somente um conjunto de valores. Um outro exemplo é a conexão com o banco de dados, uma vez que podemos ter um único objeto para realizar a conexão durante todo o ciclo de vida da aplicação e retornar este mesmo objeto sempre que uma nova conexão for solicitada.

Para tornar a transação visível durante toda aplicação e também para impedir que se sobreponha o objeto, criaremos uma classe para manipular transações (`Transaction`), cujo método construtor é marcado como privado (`private`), para que não existam várias instâncias da mesma transação. O método `open()` é responsável por abrir uma conexão com o banco de dados, armazená-la em uma variável estática e iniciar uma

transação, a qual só será iniciada se não existir outra em andamento. O método `get()` é responsável por tornar a conexão visível durante todo o processo, retornando-a sempre que for necessário interagir com o banco de dados. O método `rollback()` irá desfazer todas operações realizadas desde o início da transação, e o método `close()` irá fechar a conexão com o banco de dados, aplicando todas as operações realizadas ao longo da transação.

Só poderemos abrir uma transação pelo método `open()` caso não exista nenhuma transação aberta pendente. Isto é feito verificando se a variável estática (`self::$conn`) está vazia. Sempre que uma transação é encerrada, esta variável é reinicializada.

TTransaction.class.php

```
<?php
/*
 * classe TTransaction
 * esta classe provê os métodos necessários manipular transações
 */
final class TTransaction
{
    private static $conn;      // conexão ativa

    /*
     * método __construct()
     * Está declarado como private para impedir que se crie instâncias de TTransaction
     */
    private function __construct(){}
    /*
     * método open()
     * Abre uma transação e uma conexão ao BD
     * @param $database = nome do banco de dados
     */
    public static function open($database)
    {
        // abre uma conexão e armazena na propriedade estática $conn
        if (empty(self::$conn))
        {
            self::$conn = TConnection::open($database);
            // inicia a transação
            self::$conn->beginTransaction();
        }
    }
    /*
     * método get()
     * retorna a conexão ativa da transação
     */
}
```

```
public static function get()
{
    // retorna a conexão ativa
    return self::$conn;
}

/*
 * método rollback()
 * desfaz todas operações realizadas na transação
 */
public static function rollback()
{
    if (self::$conn)
    {
        // desfaz as operações realizadas durante a transação
        self::$conn->rollback();
        self::$conn = NULL;
    }
}

/*
 * método close()
 * Aplica todas operações realizadas e fecha a transação
 */
public static function close()
{
    if (self::$conn)
    {
        // aplica as operações realizadas
        // durante a transação
        self::$conn->commit();
        self::$conn = NULL;
    }
}
?>
```

Após criada a classe `TTransaction`, iremos agora criar um programa para utilizá-la. Para retratar fielmente a transação de uma aplicação de negócios, acabaríamos nos estendendo demais em nosso exemplo, pois precisaríamos contextualizar um problema e inserir dados relacionados em tabelas diferentes, nas quais uma operação depende do sucesso das anteriores. Para simplificar nosso exemplo, utilizaremos apenas duas instruções SQL. No primeiro caso estamos inserindo a pessoa de código “8” e nome “Galileu” na base de dados. Posteriormente estamos alterando o nome deste mesmo registro de código “8” para “Galileu Galilei”. Veja que, neste caso, não faria sentido executar a segunda instrução caso a primeira não tenha sido executada com sucesso,

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.

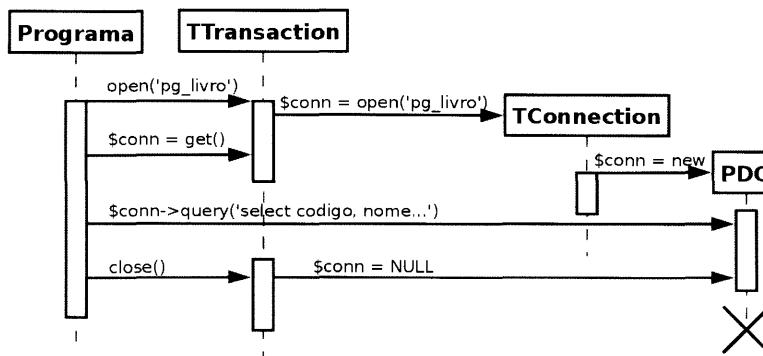


Figura 3.17 – Interação entre as classes durante a transação.

transaction.php

```

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

```

```
try
{
    // abre uma transação
    TTransaction::open('pg_livro');

    // cria uma instrução de INSERT
    $sql = new TSqlInsert;
    // define o nome da entidade
    $sql->setEntity('famosos');
    // atribui o valor de cada coluna
    $sql->setRowData('codigo', 8);
    $sql->setRowData('nome', 'Galileu');

    // obtém a conexão ativa
    $conn = TTransaction::get();
    // executa a instrução SQL
    $result = $conn->Query($sql->getInstruction());

    // cria uma instrução de UPDATE
    $sql = new TSqlUpdate;
    // define o nome da entidade
    $sql->setEntity('famosos');
    // atribui o valor de cada coluna
    $sql->setRowData('nome', 'Galileu Galilei');

    // cria critério de seleção de dados
    $criteria = new TCriteria;
    // obtém a pessoa de código "8"
    $criteria->add(new TFilter('codigo', '=', '8'));

    // atribui o critério de seleção de dados
    $sql->setCriteria($criteria);

    // obtém a conexão ativa
    $conn = TTransaction::get();
    // executa a instrução SQL
    $result = $conn->Query($sql->getInstruction());

    // fecha a transação, aplicando todas operações
    TTransaction::close();
}

catch (Exception $e)
{
    // exibe a mensagem de erro
    echo $e->getMessage();
    // desfaz operações realizadas durante a transação
    TTransaction::rollback();
}
?>
```

Um bom teste para verificar se as transações estão efetivamente funcionando seria forçar um erro em um dos comandos. Para isso, iremos alterar a seguinte linha:

```
$sql->setRowData('nome', 'Galileu Galilei');
```

para:

```
$sql->setRowData('nome_', 'Galileu Galilei');
```

Essa alteração na instrução de `UPDATE` fará com que um erro seja lançado, visto que não existe uma coluna chamada `nome_` na tabela de famosos. Assim, será exibido na tela:

```
SQLSTATE[42703]: Undefined column: 7 ERROR: column "nome_" of relation "famosos" does not exist
```

Além do erro gerado, você irá perceber que toda a transação foi abortada e nenhuma das instruções SQL envolvidas na mesma foi efetivamente aplicada.

3.3.14 Registro de log

No exemplo anterior, criamos uma classe para realizar transações com o banco de dados. É muito importante podermos registrar as operações ocorridas durante uma transação, registrando-as em um arquivo de log. No entanto, existem diversos tipos de log: em formato XML, em arquivo HTML, em formato TXT, em bancos de dados, dentre outros.

A forma mais simples de implementar o registro de log seria criar um método na classe `TTransaction` responsável por registrar os comandos SQL executados em um arquivo qualquer. Se quiséssemos utilizar tipos diferentes de log, teríamos de usar um bloco de comandos `IF/SWITCH` para tomar a decisão sobre qual algoritmo utilizar baseado nas preferências do programador. Essa forma de implementação acabaria inflando este método, o qual teria de implementar todos os algoritmos possíveis, e a nossa classe ficaria grande demais.

Outra forma de implementar o registro de log seria pela criação de diversas classes-filha de `TTransaction`, uma para cada tipo de log. Neste caso, teríamos de estendê-la cada vez que surgisse um tipo novo, de modo que teríamos a separação de cada tipo de log em uma classe diferente. Essa forma de implementação não é a mais indicada, pois logs não são tipos de transação, ou seja, a herança não faz sentido. Além disso, se surgissem derivações futuras da classe `TTransaction`, teríamos de criar diversas outras subclasses intermediárias, tornando a estrutura inflexível.

A terceira opção é separar totalmente os algoritmos de log da classe `TTransaction`, criando um outro conjunto de classes que irão exclusivamente representar essas várias estratégias diferentes de log. Dessa forma, faremos com que a transação mantenha

uma referência para o objeto que implemente o algoritmo de log. Neste caso, estamos falando do Strategy Pattern.

3.3.14.1 Strategy Pattern

Um Strategy Pattern é representado por uma família de algoritmos encapsulados em uma hierarquia de objetos, de forma que possamos facilmente trocar de algoritmo.

Para implementar o Strategy Pattern, iremos declarar uma classe abstrata contendo os métodos que este algoritmo deve prover e um conjunto de classes concretas filhas desta, as quais implementam cada versão diferente desse algoritmo. A decisão de qual algoritmo tomar fica por conta do programador, que só terá de escolher uma determinada classe a instanciar. Independente da classe escolhida, todas as classes-filha implementarão a mesma interface.

Cada formato de log necessita de um algoritmo diferente para implementação. Pensando nisto, utilizaremos o Strategy Pattern para estruturar esses três algoritmos sob a classe abstrata `TLogger`. Esta será uma classe abstrata que só indicará quais são os métodos necessários a serem implementados nas classes concretas. Em seu método construtor ela irá receber o nome do arquivo de log. Criamos também o método `write()`, que será marcado como abstract para que todas as classes-filha sejam obrigadas a implementá-lo. Sua função será justamente escrever o registro de log no arquivo. Veja no diagrama a seguir o relacionamento entre as classes.

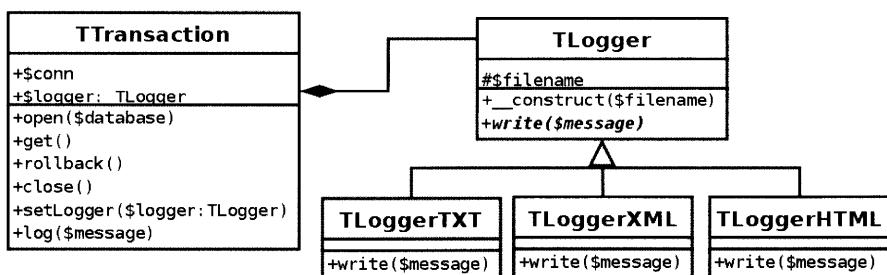


Figura 3.18 – Relacionamento entre a transação e a classe `TLogger`.

TLogger.class.php

```

<?php
/*
 * classe TLogger
 * Esta classe provê uma interface abstrata para definição de algoritmos de LOG
 */
abstract class TLogger
{
    protected $filename;      // local do arquivo de LOG
  
```

```

/*
 * método __construct()
 * instancia um logger
 * @param $filename = local do arquivo de LOG
 */
public function __construct($filename)
{
    $this->filename = $filename;
    // reseta o conteúdo do arquivo
    file_put_contents($filename, '');
}

// define o método write como obrigatório
abstract function write($message);
}
?>

```

A partir daí teremos três subclasses: `TLoggerTXT`, `TLoggerXML` e `TLoggerHTML`. Cada classe concreta terá de implementar o método `write()`, que será responsável por armazenar a mensagem no arquivo de log. A primeira classe será a `TLoggerXML`. Esta classe possui o método `write()`, o qual receberá a mensagem e irá armazená-la em um arquivo no formato XML, observando algumas tags. Também armazenaremos a hora do log `<time>`.

TLoggerXML.class.php

```

<?php
/*
 * classe TLoggerXML
 * implementa o algoritmo de LOG em XML
 */
class TLoggerXML extends TLogger
{
    /*
     * método write()
     * escreve uma mensagem no arquivo de LOG
     * @param $message = mensagem a ser escrita
     */
    public function write($message)
    {
        $time = date("Y-m-d H:i:s");
        // monta a string
        $text = "<log>\n";
        $text.= "  <time>$time</time>\n";
        $text.= "  <message>$message</message>\n";
        $text.= "</log>\n";
    }
}

```

```
// adiciona ao final do arquivo
$handler = fopen($this->filename, 'a');
fwrite($handler, $text);
fclose($handler);
}
}
?>
```

O segundo algoritmo será representado pela classe `TLoggerHTML`. Esta classe também terá o método `write()`, que receberá a mensagem e irá armazená-la em um arquivo no formato HTML, com algumas tags como `<p>` de parágrafo, `` de negrito, ou `<i>` de itálico.

TLoggerHTML.class.php

```
<?php
/*
 * classe TLoggerHTML
 * implementa o algoritmo de LOG em HTML
 */
class TLoggerHTML extends TLogger
{
    /*
     * método write()
     * escreve uma mensagem no arquivo de LOG
     * @param $message = mensagem a ser escrita
     */
    public function write($message)
    {
        $time = date("Y-m-d H:i:s");
        // monta a string
        $text = "<p>\n";
        $text.= "  <b>$time</b> : \n";
        $text.= "  <i>$message</i> <br>\n";
        $text.= "</p>\n";
        // adiciona ao final do arquivo
        $handler = fopen($this->filename, 'a');
        fwrite($handler, $text);
        fclose($handler);
    }
}
?>
```

A terceira e última classe irá registrar as mensagens de log em um arquivo-texto. Mais simples que as anteriores, esta armazenará a mensagem recebida em uma linha contendo a data e a hora.

 **TLoggerTXT.class.php**

```
<?php
/*
 * classe TLoggerTXT
 * implementa o algoritmo de LOG em TXT
 */
class TLoggerTXT extends TLogger
{
    /*
     * método write()
     * escreve uma mensagem no arquivo de LOG
     * @param $message = mensagem a ser escrita
     */
    public function write($message)
    {
        $time = date("Y-m-d H:i:s");
        // monta a string
        $text = "$time :: $message\n";
        // adiciona ao final do arquivo
        $handler = fopen($this->filename, 'a');
        fwrite($handler, $text);
        fclose($handler);
    }
}
?>
```

Para que a classe `TTransaction` utilize um dos algoritmos de log é necessário que ela tenha uma referência para um objeto do tipo `TLoggerXML`, `TLoggerHTML` ou `TLoggerTXT`. Para isso, criaremos o método `setLogger()` na transação, que irá receber uma instância de alguma classe concreta de `TLogger` e irá compor esta instância, ou seja, irá armazená-la na propriedade `$logger`, para que seja utilizada posteriormente no momento do log. O registro da mensagem de log será realizado pelo método `log()`. Como o objeto `TTransaction` possuirá uma referência para um objeto concreto do tipo `TLogger`, a responsabilidade pela execução do registro do log será delegada a ele por meio da execução do método `write()`.

 **TTransaction.class.php (continuação)**

```
<?php
/*
 * classe TTransaction
 * esta classe provê os métodos necessários manipular transações
 */
final class TTransaction
{
```

```
private static $conn;      // conexão ativa
private static $logger;    // objeto de LOG

private function __construct(){}

public static function open($database)
{
    if (empty(self::$conn))
    {
        // ...
        // desliga o log de SQL
        self::$logger = NULL;
    }
}

public static function get() {...}
public static function rollback() {...}
public static function close() {...}

/*
 * método setLogger()
 * define qual estratégia (algoritmo de LOG será usado)
 */
public static function setLogger(TLogger $logger)
{
    self::$logger = $logger;
}

/*
 * método log()
 * armazena uma mensagem no arquivo de LOG
 * baseada na estratégia ($logger) atual
 */
public static function log($message)
{
    // verifica existe um logger
    if (self::$logger)
    {
        self::$logger->write($message);
    }
}

?>
```

Como a classe `TTransaction` não sabe exatamente que tipo de objeto `TLogger` será passado para ela, a estratégia de log (algoritmo) fica totalmente a cargo do programador, o qual poderá escolher entre as estratégias `TLoggerXML`, `TLoggerHTML` ou `TLoggerTXT`. Veja

agora um exemplo de utilização das estratégias de log. Escreveremos um programa para inserir dois novos registros na tabela de famosos e registraremos no arquivo de log os comandos SQL utilizados, além de algumas mensagens de contexto para nos localizarmos dentro do arquivo de log. Na primeira parte do programa registraremos o log utilizando a estratégia `TLoggerHTML` e armazenando as mensagens no arquivo `/tmp/arquivo.html`. Em seguida, mudamos a estratégia e passamos a registrar o log utilizando a classe `TLoggerXML` e armazenando as mensagens no arquivo `/tmp/arquivo.xml`. Veja a seguir o resultado.

log.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";
    }
}

try
{
    // abre uma transação
    TTransaction::open('pg_livro');
    // define a estratégia de LOG
    TTransaction::setLogger(new TLoggerHTML('/tmp/arquivo.html'));
    // escreve mensagem de LOG
    TTransaction::log("Inserindo registro William Wallace");

    // cria uma instrução de INSERT
    $sql = new TSqlInsert;
    // define o nome da entidade
    $sql->setEntity('famosos');
    // atribui o valor de cada coluna
    $sql->setRowData('codigo', 9);
    $sql->setRowData('nome', 'William Wallace');

    // obtém a conexão ativa
    $conn = TTransaction::get();
    // executa a instrução SQL
    $result = $conn->Query($sql->getInstruction());
    // escreve mensagem de LOG
```

```
    TTransaction::log($sql->getInstruction());

    // define a estratégia de LOG
    TTransaction::setLogger(new TLoggerXML('/tmp/arquivo.xml'));

    // escreve mensagem de LOG
    TTransaction::log("Inserindo registro Albert Einstein");

    // cria uma instrução de INSERT
    $sql = new TSqlInsert;
    // define o nome da entidade
    $sql->setEntity('famosos');
    // atribui o valor de cada coluna
    $sql->setRowData('codigo', 10);
    $sql->setRowData('nome', 'Albert Einstein');

    // obtém a conexão ativa
    $conn = TTransaction::get();
    // executa a instrução SQL
    $result = $conn->Query($sql->getInstruction());
    // escreve mensagem de LOG
    TTransaction::log($sql->getInstruction());

    // fecha a transação, aplicando todas as operações
    TTransaction::close();
}

catch (Exception $e)
{
    // exibe a mensagem de erro
    echo $e->getMessage();
    // desfaz operações realizadas durante a transação
    TTransaction::rollback();
}
?>
```

Poderemos ver a seguir o resultado dos registros de log utilizando as estratégias TLoggerHTML e TLoggerXML.

 /tmp/arquivo.html

```
<p>
<b>2007-06-05 14:42:56</b> :
<i>Inserindo registro William Wallace</i> <br>
</p>
<p>
<b>2007-06-05 14:42:56</b> :
<i>INSERT INTO famosos (codigo, nome) values (9, 'William Wallace')</i> <br>
</p>
```

✉ /tmp/arquivo.xml

```
<log>
  <time>2007-06-05 14:42:56</time>
  <message>Inserindo registro Albert Einstein</message>
</log>
<log>
  <time>2007-06-05 14:42:56</time>
  <message>INSERT INTO famosos (codigo, nome) values (10, 'Albert Einstein')</message>
</log>
```

Capítulo 4

Mapeamento Objeto-Relacional

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

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

dados oferecem vários recursos que permitem armazenar e manipular, por meio da linguagem SQL, os dados neles contidos.

Os bancos de dados relacionais surgiram na década de 1970 quando o padrão no desenvolvimento de aplicações era o estruturado. Nas últimas décadas, com a crescente adoção da orientação a objetos no desenvolvimento de aplicações, surgiram os bancos de dados orientados a objetos. Apesar disso, sua adoção ainda é muito pequena, em parte, por causa do desempenho inferior se comparado aos bancos de dados relacionais, que dominam amplamente o mercado e são utilizados há muito mais tempo, fornecendo ferramentas mais testadas, aceitas e maduras. Sendo assim, os bancos de dados relacionais são amplamente utilizados para armazenar objetos.

Entretanto, existem diversos conceitos na orientação a objetos para os quais o modelo relacional simplesmente não oferece suporte, então temos uma diferença de conceito. Os bancos de dados foram construídos para armazenar dados, ao passo que, no paradigma orientado a objetos, além de dados (atributos), temos comportamento (métodos) e outras estruturas complexas. Dessa forma, é necessário utilizar alguma ferramenta de compatibilidade entre os dois modelos, que geralmente implementam alguma técnica de mapeamento objeto-relacional. Tais técnicas visam à persistência, ou seja, ao armazenamento de objetos em bancos de dados relacionais.

Com o passar do tempo, alguns bancos de dados relacionais foram adicionando alguns recursos relacionados a objetos, como a herança em banco de dados. Ainda assim sua maior limitação reside em sua própria estrutura, representada pela relação entre tabelas, nas quais os dados residem em linhas e colunas, estrutura incapaz de retratar com precisão a natureza do mundo orientado a objetos em que temos, além de atributos, métodos, heranças e relações mais complexas entre os objetos, como agregações e composições.

Para realizar este trabalho de armazenar um objeto de negócios em um banco de dados, também conhecido como mapeamento objeto-relacional, existem diversas técnicas (*design patterns*) e também existem alguns frameworks que desempenham tal papel de forma automatizada.

4.2 Mapeamento objeto-relacional

Um dos grandes motivos para o sucesso dos bancos de dados relacionais é a utilização do SQL, uma linguagem padronizada para acesso e manipulação de dados.

Apesar da grande utilização da linguagem SQL no meio corporativo, muitos programadores ainda fazem uso errado da linguagem por não a entenderem totalmente ou por fazerem uso incorreto de seus recursos, o que leva a problemas na definição das consultas e consequentes problemas de performance e manutenibilidade.

Por essas e outras razões é recomendável separar acesso e manipulação dos dados via linguagem SQL do domínio de negócios da aplicação, distribuindo esta parte em diferentes classes da aplicação e constituindo uma camada de acesso aos dados.

Uma das maiores diferenças entre objetos e bancos de dados relacionais está na forma de representação dos relacionamentos. Os objetos trabalham com ponteiros (links) para outros objetos. Esses ponteiros são criados em tempo de execução, ao passo que os bancos de dados relacionam suas estruturas por meio de chaves (primárias e estrangeiras) presentes nas tabelas, estabelecendo vínculos.

A seguir, estudaremos algumas das técnicas (patterns) utilizadas para equalizar as diferenças entre os bancos de dados relacionais e o modelo de objetos durante a etapa de mapeamento.

4.2.1 Identity Field

A forma que os bancos de dados relacionais encontram para dar unicidade aos seus registros é a criação de chaves primárias nas tabelas, nas quais criamos uma coluna cujo valor-chave é não-repetitivo e não-nulo.

Os objetos de um sistema não precisam de tal chave para se tornarem únicos. A unicidade dos objetos em memória é controlada por mecanismos da própria linguagem de programação, a qual geralmente oferece meios de se referenciar um objeto, ou mesmo de clonar um objeto já existente, controlando sua identidade por meio de um OID (Object Identifier). De qualquer forma, dois objetos distintos são armazenados em regiões diferentes da memória e referenciados por suas variáveis.

Como compatibilizar então o modelo conceitual e o modelo de dados da aplicação? A resposta é simples e consiste da propagação da chave de identificação do registro também para os objetos de negócio da aplicação. A adoção dessa chave é de suma importância, uma vez que lemos as informações de um registro do banco de dados, instanciamos um objeto de negócio com tais informações, modificamos esse objeto e, em um dado momento, desejamos armazenar este objeto novamente no banco de dados, substituindo suas informações originais. Isto é possível somente pela sua chave primária.

Para realizar o mapeamento objeto-relacional é necessário perder alguns vícios. O mais importante é a utilização de atributos com significado no modelo conceitual em chaves primárias, como RG, CPF, CNPJ, CEP, dentre outros. É fundamental que não se utilize um atributo para tal identificação, pois atributos significativos, relativos ao domínio dos negócios, estão sujeitos a mudanças, como qualquer outra informação. Pode-se, no entanto, utilizar recursos do próprio banco de dados, como campos “auto-increment” como estratégia para gerar automaticamente esses identificadores únicos dos objetos.

A utilização de chaves compostas em tabelas também torna mais complexo o mapeamento objeto-relacional. Podemos automatizar a tarefa de leitura e gravação de objetos no banco de dados quando uma tabela utiliza um único campo como chave primária, mapeando o objeto entre a aplicação e o banco de dados por meio desse campo. Entretanto, quando uma tabela utiliza chaves compostas, a complexidade das comparações, e por consequência da camada de mapeamento, aumenta. Portanto é fundamental a utilização de um único campo como chave primária. Campos numéricos tendem a ser mais rápidos, além de facilitarem a criação de seqüências (obter o próximo ID). Na Figura 4.1 temos uma classe com seu Identity Field.

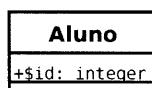


Figura 4.1 – Identity Field.

4.2.2 Foreign Key Mapping

Como já sabemos, os objetos podem ter diversos tipos de relacionamento. As associações são um tipo de relacionamento em que um objeto tem uma propriedade que referencia um outro objeto. Como vimos anteriormente, os objetos em memória têm sua unicidade garantida pelos mecanismos da linguagem de programação, mas para armazená-los no banco de dados é preciso mais do que isso, como vimos no pattern Identity Field.

Para mapearmos um objeto para o banco de dados, precisamos de um campo de identificação (Identity Field), ou seja, a chave primária da tabela também presente no objeto, pois, para mapearmos relacionamentos entre objetos, utilizamos a mesma técnica, implementando o mapeamento pelo uso de chaves estrangeiras (Foreign Keys).

Na Figura 4.2 temos duas classes. A classe `Pessoa` possui uma associação com a classe `Cidade`, que por sua vez é representada pelo atributo `Cidade` da classe `Pessoa`. Os objetos ainda possuem alguns métodos que não são relevantes para este exemplo.

A forma que dispomos para representar tal estrutura no banco de dados é criar uma chave estrangeira na tabela `Pessoa`, apontando para a chave primária da tabela `Cidade`. Para essa técnica funcionar bem, é fundamental que cada objeto tenha o seu Identity Field, como demonstrado na Figura 4.3.

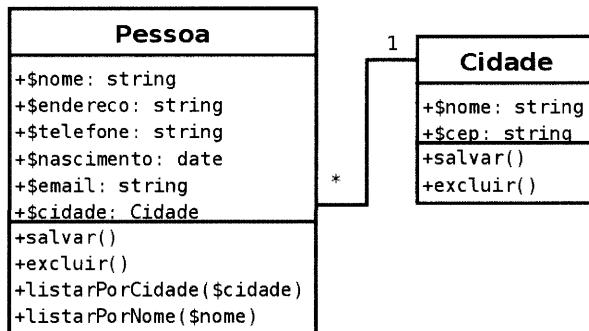


Figura 4.2 – Associação.

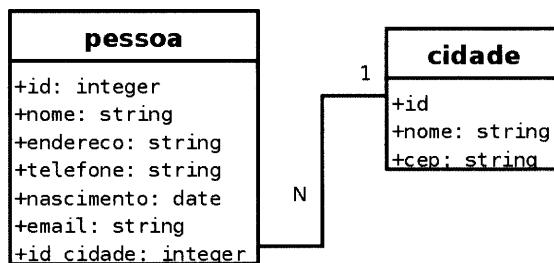


Figura 4.3 – Mapeando uma associação.

Um pouco mais complexo é o relacionamento de composição, no qual um objeto pode conter de uma a várias instâncias de um outro objeto, sendo responsável pela sua criação e destruição. Esse relacionamento é visto na Figura 4.4, no qual temos um objeto **Pessoa** e um objeto **Telefone**. Uma **Pessoa** poderá ter vários telefones que são armazenados na forma de um atributo no objeto **Pessoa**, chamado **\$telefones**. Tal atributo, na verdade, é um array de objetos do tipo **Telefone**. Esse relacionamento é representado na forma de composição porque os objetos **Telefone** são parte intrínseca do objeto **Pessoa**, ou seja, um objeto **Telefone** (parte) fará parte somente de um único objeto **Pessoa** (todo). Quando o objeto **Pessoa** for destruído, seus objetos **Telefone** também serão. Os atributos do objeto **Telefone** são o tipo (celular, residencial, profissional) e o número em si.

A forma mais simples de se realizar o mapeamento desse tipo de relacionamento para o banco de dados é por meio da criação de uma chave estrangeira (foreign key) em cada um dos objetos compostos (parte), apontando para o objeto principal.

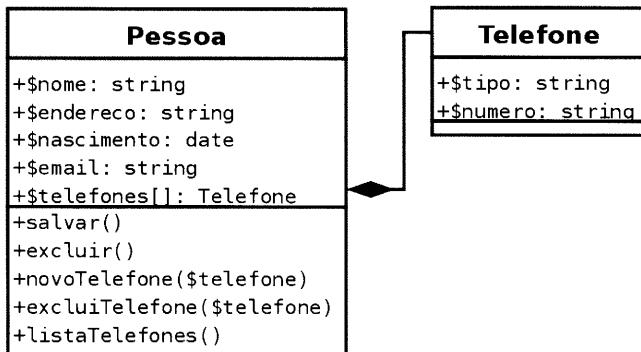


Figura 4.4 – Composição.

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.

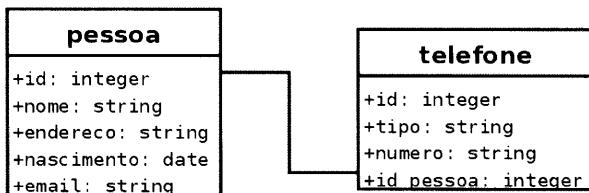


Figura 4.5 – Mapeando uma composição.

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

de compras, o `removeProduto()`, o qual irá remover um produto da cesta, e também `calculaTotal()`, que irá calcular o valor total da cesta. Os métodos `insereProduto()` e `removeProduto()` armazenam estes objetos do tipo `Produto` na propriedade `$produtos`, vetor que contém objetos do tipo `Produto`.

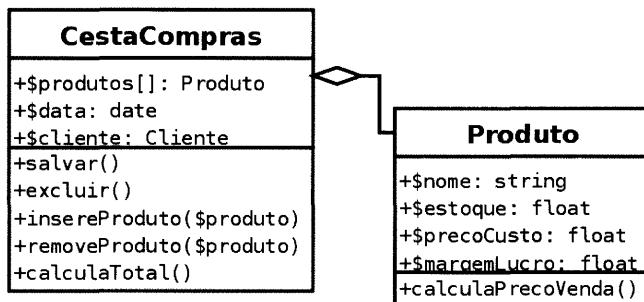


Figura 4.6 – Agregação.

Até aí não temos muita diferença para a composição. A grande diferença é que um mesmo produto pode fazer parte de cestas de compras diferentes e os objetos são criados no contexto externo à classe `CestaCompras` para depois serem agregados pelo método `insereProduto()`, diferentemente da composição, na qual os objetos compostos são criados dentro da própria classe que representa o todo. Então não podemos criar na tabela de `Produto` uma chave estrangeira (Foreign Key Mapping) apontando para a chave primária da cesta de compras, pois isso limitaria um produto a fazer parte somente de uma cesta. Para resolver esse problema, é necessário criar uma tabela intermediária no banco de dados (`itens_cesta`) contendo as chaves primárias das duas tabelas (`cesta` e `produtos`), permitindo, então, que, nesta tabela, possamos representar diversos produtos em uma cesta e também que um mesmo produto possa estar em cestas de compras diferentes, como ilustrado pela Figura 4.7.

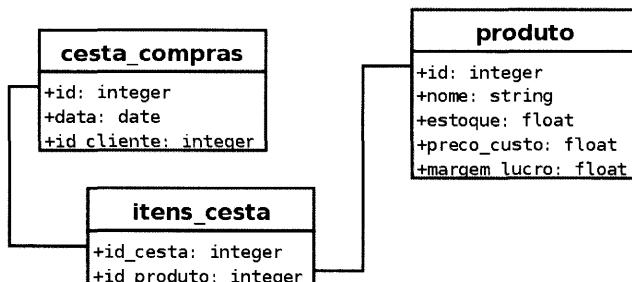


Figura 4.7 – Mapeando a agregação.

É importante dizer que a tabela `itens_cesta` será totalmente manipulada pela classe `CestaCompras`, ou seja, será totalmente transparente ao programador. Em nível de

aplicação teremos duas classes (`CestaCompras` e `Produto`) que ao serem armazenadas no banco de dados terão seus dados distribuídos em três tabelas diferentes (`cesta_compras`, `itens_cesta` e `produto`).

Em alguns casos, o relacionamento entre duas tabelas extrapola o que pode ser representado com uma simples agregação. Um exemplo clássico é o relacionamento entre um aluno e um livro em um sistema de biblioteca. O aluno realiza um empréstimo de um livro, sendo que ele pode retirar vários livros e um livro pode ser retirado por alunos diferentes. Até aí temos tudo para acreditar que se trata de uma relação de agregação. No entanto, existem atributos importantes nesse relacionamento que não conseguiríamos representar somente com uma agregação, como a data do empréstimo, a data da devolução, o operador que efetuou o empréstimo, dentre outros. Esses atributos importantes ao negócio justificam promover o relacionamento de empréstimo a uma classe no nível da aplicação. Dessa forma, teríamos o seguinte diagrama de classes.



Figura 4.8 – Promovendo um relacionamento.

4.2.4 Single Table Inheritance

Já vimos como mapear relacionamentos de associação, herança e agregação ao banco de dados. Vamos estudar, então, como se dá o mapeamento de heranças. Existem basicamente três formas de mapear um relacionamento de herança. Para exemplificar, tomaremos por base o diagrama de classes (Figura 4.9), no qual temos uma herança da classe `Material` que pode ser tanto do tipo `Livro` quanto do tipo `DVD`. Cada classe tem seus atributos próprios, suas peculiaridades, mas ambas têm algumas características em comum, como os atributos `titulo` e `genero` e os métodos `salvar()`, `excluir()` e `listar()`.

A primeira forma de mapear as classes para uma base de dados relacional é a criação de uma tabela contendo os campos de toda a estrutura da herança, ou seja, a soma de todos os atributos. Na Figura 4.10 temos essa tabela.

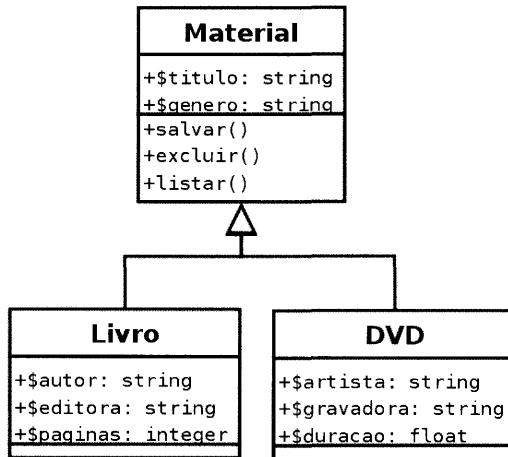


Figura 4.9 – Herança entre classes.

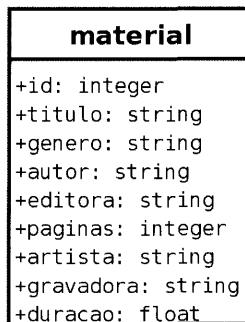


Figura 4.10 – Single Table Inheritance.

Você deve estar se perguntando como distinguir se um registro é um livro ou é um DVD. Quando tivermos de instanciar um objeto `Livro` ou `DVD` a partir do banco de dados precisaremos dessa distinção. Nesse caso, torna-se obrigatório criarmos um campo indicando o tipo do objeto na tabela. Este campo pode ser um `char(1)` contendo “L” para `Livro` e “D” para `DVD`, por exemplo. Também fica claro que nunca teremos todos os campos da tabela preenchidos. Se o material for um `Livro`, `artista`, `gravadora` e `duracao` ficarão vazios. Se o material for um `DVD`, `autor`, `editora` e `paginas` ficarão vazios, o que não é um problema tão grave, pois os gerenciadores modernos de banco de dados otimizam bastante o armazenamento de informações, diferentemente de bancos de dados antigos como DBF ou Cobol.

Esta abordagem é a mais simples e fácil de implementar, pois dispensa o uso de joins entre tabelas, e, com um simples `SELECT`, conseguimos trazer quaisquer informações na memória, bem como instanciar objetos `Livro` e `DVD`. Entretanto, o programador pode se confundir com o fato de ter alguns campos preenchidos em alguns casos e outros não, como se a tabela tivesse uma “crise de identidade” por ser ampla demais.

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.

livro	dvd
<code>+id: integer</code>	<code>+id: integer</code>
<code>+titulo: string</code>	<code>+titulo: string</code>
<code>+genero: string</code>	<code>+genero: string</code>
<code>+autor: string</code>	<code>+artista: string</code>
<code>+editora: string</code>	<code>+gravadora: string</code>
<code>+paginas: integer</code>	<code>+duracao: float</code>

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.

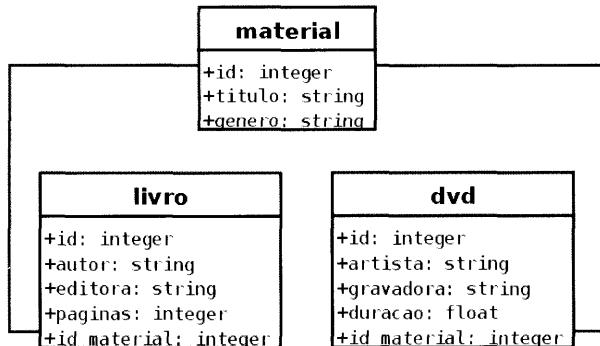


Figura 4.12 – Class Table Inheritance.

Esta abordagem é a que nos leva a uma maior normalização do banco de dados frente aos dois outros patterns abordados até o momento. Cada classe e cada atributo do modelo de objetos encontra exatamente um correspondente no modelo do banco de dados, o que leva a um melhor entendimento.

Quanto mais normalizada fica a estrutura do banco de dados, mais complexa é a operação de carregar posteriormente esses objetos na memória. Se quisermos construir uma listagem com objetos de todos os tipos de materiais e suas respectivas informações, possivelmente teremos de fazer um *join* entre várias tabelas da hierarquia, e se quisermos listar somente livros, teremos de fazer um *join* entre esta tabela e a tabela de materiais. Dependendo do tamanho e estrutura do banco de dados, a operação poderá prejudicar a performance.

Outra questão que deve ser analisada com cuidado é que, às vezes, um objeto transita dentro da hierarquia de classes. Um *livro* dificilmente virá a se transformar em um *DVD*, mas entre as classes *Pessoa*, *Funcionario* e *Estagiario*, sendo que *Funcionario* e *Estagiario* são classes-filha de *Pessoa*, freqüentemente um *Estagiario* pode vir a ser efetivado como um *Funcionario*, e, neste caso, troca de tipo. Nesta abordagem, teríamos de fazer um registro transitar de uma tabela para outra, ao passo que em “Single Table Inheritance” teríamos apenas que alterar a flag da tabela indicando o tipo de objeto. Cada pattern de mapeamento de herança tem suas vantagens e desvantagens, cabe ao projetista descobrir qual pattern é mais indicado em cada caso.

4.2.7 Lazy Initialization

Em um modelo conceitual temos muitos objetos conectados por meio de vários tipos de relacionamentos como associações e agregações. Freqüentemente precisamos navegar por entre os relacionamentos dos objetos, o que torna necessário que esses objetos relacionados estejam disponíveis (instanciados). Poderíamos carregar automaticamente todos os relacionamentos de um objeto quando o recuperamos

(obtemos) da base de dados, mas, dependendo do nosso objeto de ponto de partida, poderíamos estar carregando quase todo banco de dados nesta operação, consumindo uma quantidade enorme de memória.

Para evitar esse tipo de situação, o ideal seria que os objetos relacionados fossem instanciados somente quando eles fossem necessários para a aplicação. Esta é exatamente a proposta do pattern Lazy Initialization, também conhecido por Inicialização Tardia. Utiliza-se o termo tardia, pois os relacionamentos somente são disponibilizados para aplicação quando são necessários, evitando uma carga inicial desnecessária.

Uma forma simples de fazer isso é criar um método encarregado de instanciar o(s) objeto(s) relacionado sempre que estes se tornam necessários, mas então teríamos de lembrar de executar este método sempre antes de acessar um objeto relacionado.

Felizmente o PHP nos oferece um recurso que facilita essa tarefa. Estamos falando dos métodos interceptadores, os quais vimos anteriormente no capítulo que trata de orientação a objetos. Os métodos interceptadores permitem que tenhamos um grande controle sobre as interações que acontecem com um objeto. O método `__get()`, por exemplo, é executado automaticamente sempre que uma propriedade não existente é requisitada. Este método cumpre exatamente a função de que precisamos para implementar o pattern Lazy Initialization. Imagine uma marcação no objeto que represente um relacionamento. Sempre que precisarmos acessar tal marcação, o objeto relacionado será instanciado. Essa marcação pode ser uma propriedade inexistente do objeto. Sempre que esta propriedade for requisitada, o método `__get()` será executado, tratando de instanciar automaticamente o objeto associado.

No programa a seguir estamos demonstrando este conceito por meio das interações existentes entre duas classes: `Pessoa` e `Cidade`. A classe `Pessoa` possui como propriedades: `$nome` e `$cidadeID`, sendo este último o código da sua cidade. Quando instanciamos um objeto da classe `Pessoa`, passamos como parâmetro o seu nome e o código da cidade. Já a classe `Cidade` possui como propriedades `$id` e `$nome`. No método construtor da classe `Cidade` passamos o seu `$id` como parâmetro e automaticamente seu nome será definido com base em um array de cidades, que simula o banco de dados. A classe `Cidade` ainda possui os métodos `setNome()` e `getNome()` para manipular a propriedade `$nome`.

Veja que, ao instanciarmos um objeto `Pessoa`, nenhum outro objeto relacionado é instanciado automaticamente, como poderia ser a classe `Cidade`. Mesmo assim, você pode ver na seqüência do programa que podemos acessar as propriedades de uma pessoa livremente (`$pessoa->cidade`). Isto é possível por causa do método `__get()`, que possibilita investigarmos qual propriedade de um objeto está sendo requerida. Caso a propriedade requerida se chame “`cidade`”, automaticamente é instanciado um objeto `Cidade`, baseado no código da cidade, e este objeto é retornado ao usuário. Portanto,

estamos carregando o objeto relacionado na memória somente quando o mesmo se torna necessário e é efetivamente acessado, neste caso pela marcação chamada “cidade”, que é na realidade uma propriedade inexistente, como demonstrado pelo exemplo a seguir.

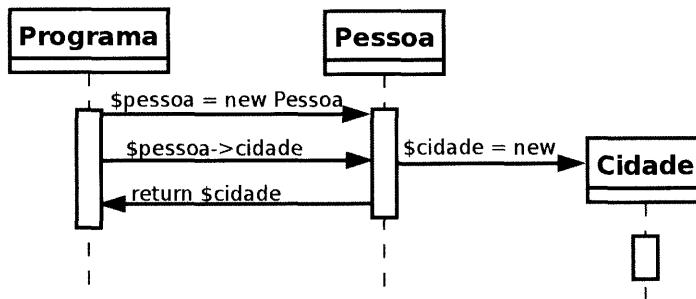


Figura 4.13 – Lazy Initialization.

Observação: para não nos estendermos demais na listagem do código-fonte, substituiremos o acesso ao banco de dados por arrays estáticos.

💻 lazy_init.php

```

<?php
/*
 * classe Pessoa
 */
class Pessoa
{
    private $nome;      // nome da pessoa
    private $cidadeID; // ID da cidade

    /*
     * método construtor
     * instancia o objeto, define alguns atributos
     * @param $nome      = nome da pessoa
     * @param $cidadeID = código da cidade
     */
    function __construct($nome, $cidadeID)
    {
        $this->nome      = $nome;
        $this->cidadeID = $cidadeID;
    }

    /*
     * método __get
     * intercepta a obtenção de propriedades
  
```

```
* @param $propriedade = nome da propriedade
*/
function __get($propriedade)
{
    if ($propriedade == 'cidade');
    {
        return new Cidade($this->cidadeID);
    }
}

/*
* classe Cidade
*/
class Cidade
{
    private $id;
    private $nome; // nome da cidade

    /*
     * método construtor
     * instancia o objeto
     * @param $id = ID da cidade
     */
    function __construct($id)
    {
        $data[1] = 'Porto Alegre';
        $data[2] = 'São Paulo';
        $data[3] = 'Rio de Janeiro';
        $data[4] = 'Belo Horizonte';

        // atribui o id
        $this->id = $id;

        // define seu nome
        $this->setNome($data[$id]);
    }

    /*
     * método setNome
     * define o nome da cidade
     * @param $nome = nome da cidade
     */
    function setNome($nome)
    {
        $this->nome = $nome;
    }
}
```

```
/*
 * método getName
 * retorna o nome da cidade
 */
function getName()
{
    return $this->nome;
}

// instancia dois objetos Pessoa
$maria = new Pessoa('Maria da Silva', 1);
$pedro = new Pessoa('Perdo Cardoso', 2);

// exibe o nome da cidade de cada Pessoa
echo $maria->cidade->getName() . "<br>\n";
echo $pedro->cidade->getName() . "<br>\n";

// exibe o atributo cidade
print_r($maria->cidade);
?>
```

Resultado:

```
Porto Alegre
São Paulo
Cidade Object
(
    [id:private] => 1
    [nome:private] => Porto Alegre
)
```

4.3 Modelo de negócios

Por modelo de negócios entendemos toda a camada da aplicação responsável por implementar os conceitos da lógica de negócios. Essa camada envolve a representação de conceitos próximos ao mundo real modelados na forma de objetos, os quais são responsáveis por cálculos, processamentos e, em alguns casos, pela sua própria persistência em bases de dados. Para estruturar tal camada da aplicação, existem alguns padrões que estudaremos a seguir.

4.3.1 Domain Model Pattern

O pattern Domain Model é utilizado para capturar as idéias e expressar os conceitos envolvidos na aplicação, bem como seus relacionamentos por meio de um conjunto de objetos relacionados. Estes objetos (pessoas, notas fiscais, produtos, fornecedores) representarão fielmente o modelo de negócios usando os relacionamentos disponíveis no modelo de objetos.

No pattern Domain Model, os objetos são responsáveis por manipular os dados e conter regras de negócio. Um modelo de negócios geralmente tem semelhanças com o modelo de banco de dados. Mesmo assim, algumas diferenças são evidentes. O modelo de negócios trabalha com dados e métodos, ao passo que o modelo de dados armazena somente dados. Além disso, este pattern permite a utilização de relacionamentos complexos como a herança.

O Domain Model pode ser simples a ponto de ter um objeto para cada tabela do Modelo de Dados ou pode ter complexas representações com heranças, agregações, composições etc. Quanto mais complexo for o Domain Model, mais difícil será a tarefa de mapeá-lo para o Modelo de Dados.

No exemplo a seguir estamos representando um modelo com duas classes: `Produto` e `Venda`. Um produto representa uma mercadoria a ser vendida e possui como propriedades sua descrição, quantidade em estoque e preço de custo. O preço de venda desse produto será calculado automaticamente baseado em uma margem de lucro de 30% sobre o preço de custo por meio do método `calculaPrecoVenda()`. Esta classe inicializará suas propriedades no método construtor e ainda terá o método `registraCompra()` acionado sempre que forem adquiridas mais quantidades de um produto. Este método é responsável por definir o novo preço de custo e aumentar o estoque. Também teremos o método `registraVenda()`, responsável por diminuir o estoque quando da ocorrência de uma venda e o método `getEstoque()` que retorna o estoque atual.

A classe `Venda` representa uma venda realizada e será responsável por agrupar um grupo de produtos vendidos e suas respectivas quantidades. Ela possui o método `addItem()` que recebe a quantidade vendida e um objeto do tipo `Produto` e armazena-os em um array. Também oferece o método `getItems()`, o qual retorna o array de produtos vendidos.

No exemplo a seguir estamos instanciando uma cesta e agregando a ela alguns objetos do tipo `Produto`. Ao final estamos finalizando a cesta, totalizando o seu valor e diminuindo o estoque dos produtos vendidos.

Uma das grandes dificuldades na orientação a objetos está em perceber o correto tipo de relacionamento entre os objetos do nosso modelo conceitual. Veja que, no exemplo a seguir, `Produto` e `Venda` se relacionam por meio de uma relação de agregação.

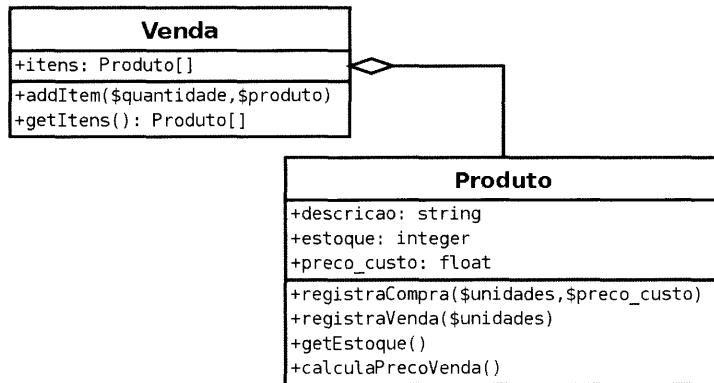


Figura 4.14 – Modelo da aplicação.

domain_model.php

```

<?php
/*
 * class Produto
 * representa um Produto a ser vendido
 */
final class Produto
{
    private $descricao;      // descricao do produto
    private $estoque;        // estoque atual
    private $preco_custo;    // preco de custo

    /*
     * método construtor
     * define alguns valores iniciais
     * @param $descricao = descrição do produto
     * @param $estoque = estoque atual
     * @param $preco_custo= preco de custo
     */
    public function __construct($descricao, $estoque, $preco_custo)
    {
        $this->descricao = $descricao;
        $this->estoque = $estoque;
        $this->preco_custo= $preco_custo;
    }
    /*
     * método registraCompra
     * registra uma compra, atualiza custo e incrementa o estoque atual
     * @param $unidades = unidades adquiridas
     * @param $preco_custo= novo preco de custo
     */

```

```
public function registraCompra($unidades, $preco_custo)
{
    $this->preco_custo = $preco_custo;
    $this->estoque += $unidades;
}

/*
 * método registraVenda
 * registra uma venda e decrementa o estoque
 * @param $unidades = unidades vendidas
 */
public function registraVenda($unidades)
{
    $this->estoque -= $unidades;
}

/*
 * método getEstoque
 * retorna a quantidade em estoque
 */
public function getEstoque()
{
    return $this->estqoue;
}

/*
 * método calculaPrecoVenda
 * retorna o preco de venda, baseado em uma margem de 30% sobre o custo
 */
public function calculaPrecoVenda()
{
    return $this->preco_custo * 1.3;
}

/*
 * classe Venda
 * representa uma Venda de Produtos
 */
final class Venda
{
    private $itens; // itens da venda

    /*
     * método addItem
     * adiciona um item na venda
     * @param $quantidade = quantidade vendida
    */
```

```
* @param $produto    = objeto produto
*/
public function addItem($quantidade, Produto $produto)
{
    $this->itens[] = array($quantidade, $produto);
}

/*
* método getItems
* retorna o array de itens da venda
*/
public function getItens()
{
    return $this->itens;
}

// instancia objeto Venda
$venda= new Venda;
// adiciona alguns produtos
$venda->addItem(3, new Produto('Vinho', 10, 15)); // 58.5
$venda->addItem(2, new Produto('Salame', 20, 20)); // 52
$venda->addItem(1, new Produto('Queijo', 30, 10)); // 13

/*
* rotina para calcular o total
* de uma venda e diminuir o estoque
*/
$total=0;
foreach ($venda->getItens() as $item)
{
    $quantidade = $item[0];
    $produto    = $item[1];

    // soma o total
    $total+= $produto->calculaPrecoVenda() * $quantidade;
    // diminui estoque
    $produto->registraVenda($quantidade);
}
echo $total;
?>
```

■ Resultado:

123,5

Existem algumas tarefas relacionadas aos objetos de negócio que se repetem ao longo do código da aplicação em diferentes partes do sistema. Estas tarefas podem ser promovidas a métodos de um objeto de negócio, o que diminui a duplicação de código no sistema, uma vez que passaremos a fazer chamadas a métodos deste objeto, em vez de copiar e colar este bloco de código ao longo de diferentes partes do sistema nas quais ele se faz necessário.

É necessário tomar cuidado para não promover quaisquer tarefas a métodos de nosso objeto de negócio. É preciso que exista uma generalidade no seu uso, que permita a este método ser utilizado em diferentes locais do sistema. Tarefas muito específicas devem ser mantidas fora do escopo dos objetos de negócio, sob a pena de inflarmos essas estruturas, jogando por terra todo o benefício de estarmos utilizando uma estrutura orientada a objetos.

Neste exemplo em que demonstramos a interação entre `Produto` e `Venda`, temos um bloco de código ao final da execução que finaliza a venda. Caso esse bloco de código se repita em outras partes do sistema, é conveniente promovê-lo a método. Como ele diz respeito à `Venda`, o lugar mais conveniente para movê-lo é a própria classe `Venda`. Vejamos como ficaria nosso código após essa mudança.

domain_model2.php

```
<?php
final class Produto
{
    ...
}
/*
 * classe Venda
 * representa uma Venda de Produtos
 */
final class Venda
{
    private $itens; // itens da cesta

    /*
     * método addItem
     * adiciona um item na cesta
     * @param $quantidade = quantidade vendida
     * @param $produto    = objeto produto
     */
    public function addItem($quantidade, Produto $produto)
    {
        $this->itens[] = array($quantidade, $produto);
    }
}
```

```
/*
 * método getItems
 * retorna o array de itens da cesta
 */
public function getItens()
{
    return $this->itens;
}

/*
 * método finaliza
 * calcula o total de uma cesta e diminuir o estoque
 */
public function finalizaVenda()
{
    foreach ($this->itens as $item)
    {
        $quantidade = $item[0];
        $produto    = $item[1];

        // soma o total
        $total+= $produto->calculaPrecoVenda() * $quantidade;
        // diminui estoque
        $produto->registraVenda($quantidade);
    }
    return $total;
}

// instancia objeto Venda
$venda= new Venda;
// adiciona alguns produtos
$venda->addItem(3, new Produto('Vinho', 10, 15));
$venda->addItem(2, new Produto('Salame', 20, 20));
$venda->addItem(1, new Produto('Queijo', 30, 10));

// finaliza a venda
echo $venda->finalizaVenda();
?>
```

Observação: nesse exemplo, é importante notar que existe uma instância de objeto para cada produto e para cada venda que o sistema irá manipular, e cada um desses objetos contém dados (propriedades) e comportamento (métodos) que atuam sobre os mesmos.

4.3.2 Table Module

Uma das principais idéias da orientação a objetos, senão a principal, é a possibilidade de mesclarmos dados e comportamento em uma única estrutura de dados – o objeto. O pattern Domain Module, visto anteriormente, reflete bem esse comportamento. Se temos um jogador na tabela de jogadores, temos também uma instância da classe `Jogador` para representar este jogador na aplicação. Entretanto, há uma dificuldade inerente no mapeamento de objetos que utilizam o pattern Domain Module para uma estrutura de banco de dados relacional.

Existe uma forma diferente de estruturar os objetos de negócio – utilize o pattern Table Module. Este pattern organiza a aplicação de tal forma que uma única instância de uma classe controla a lógica de negócios de todos os registros de uma tabela, diferentemente do Domain Module, no qual cada registro é representado por uma instância diferente. Nesse pattern também temos o comportamento e os dados em uma mesma estrutura de classe. Embora seja ele mais permissivo, permitindo a manipulação direta de estruturas de dados que passamos por meio de seus métodos como `datasets`, temos a desvantagem de subutilizar alguns relacionamentos da orientação a objetos como a agregação, a composição e a herança. Podemos dizer que o pattern Table Module pode ser utilizado quando a aplicação é baseada em uma estrutura orientada a tabelas, e não temos necessidade de complexos relacionamentos para representar o modelo conceitual de nossa aplicação.

Estes `datasets` podem ser retornos de consultas provenientes do banco de dados. Sempre que precisarmos de um conjunto de dados, estaremos falando de um `dataset`. Sempre que precisarmos atuar sobre um registro específico, precisaremos passar como parâmetro seu identificador no banco de dados, tendo em vista que neste pattern não trabalhamos vários objetos individuais como no Domain Model.

Geralmente temos um banco de dados relacional sendo manipulado por instruções SQL. Para facilitar a explicação do exemplo, demonstraremos a utilização dos datasets por meio de um array estático. Uma classe que implementa o pattern Table Module pode ser utilizada na forma estática ou também podemos ter uma instância sua para inicializar o `dataset`, como neste exemplo.

No programa a seguir, temos uma classe representando um `Produto`. Note que, neste pattern, temos uma única instância da classe `Produto` manipulando vários registros de produtos. O armazenamento de informações, que geralmente é realizado por um gateway de conexão com banco de dados (visto na seção que trata sobre Gateways), foi substituído pelo array estático `$recordset`, o qual nos permitirá manter o foco no conceito aqui demonstrado. Note também a necessidade de identificar, por meio de seu `ID`, o produto que desejamos manipular em cada método, o que é necessário por causa da utilização de uma única instância para manipulação de todos os objetos de dados.

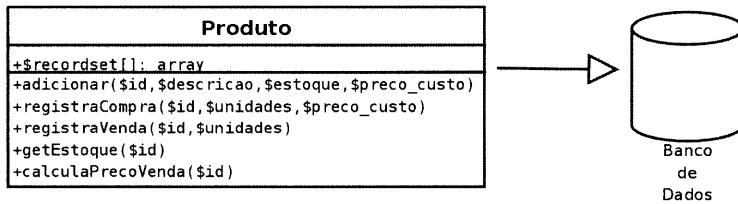


Figura 4.15 – Table Module.

table_module.php

```

<?php
/*
 * class Produto
 * representa um Produto a ser vendido
 */
final class Produto
{
    static $recordset = array(); // representa nossa estrutura de dados

    /*
     * método adicionar
     * adiciona um produto ao registro
     * @param $descricao = descrição do produto
     * @param $estoque = estoque atual
     * @param $preco_custo= preço de custo
     */
    public function adicionar($id, $descricao, $estoque, $preco_custo)
    {
        self::$recordset[$id]['descricao'] = $descricao;
        self::$recordset[$id]['estoque'] = $estoque;
        self::$recordset[$id]['preco_custo'] = $preco_custo;
    }

    /*
     * método registraCompra
     * registra uma compra, atualiza custo e incrementa o estoque atual do produto
     * @param $unidades = unidades adquiridas
     * @param $preco_custo= novo preço de custo
     */
    public function registraCompra($id, $unidades, $preco_custo)
    {
        self::$recordset[$id]['preco_custo'] = $preco_custo;
        self::$recordset[$id]['estoque'] += $unidades;
    }
}

```

```
/*
 * método registraVenda
 * registra uma venda e decrementa o estoque
 * @param $unidades = unidades vendidas
 */
public function registraVenda($id, $unidades)
{
    self::$recordset[$id]['estoque'] -= $unidades;
}

/*
 * método getEstoque
 * retorna a quantidade em estoque
 */
public function getEstoque($id)
{
    return self::$recordset[$id]['estoque'];
}

/*
 * método calculaPrecoVenda
 * retorna o preço de venda, baseado em uma margem de 30% sobre o custo
 */
public function calculaPrecoVenda($id)
{
    return self::$recordset[$id]['preco_custo'] * 1.3;
}

// instancia objeto Produto
$produto = new Produto;

// adiciona alguns Produtos
$produto->adicionar(1, 'Vinho', 10, 15);
$produto->adicionar(2, 'Salame', 20, 20);

// exibe os estoques atuais
echo "estoque: <br>\n";
echo $produto->getEstoque(1) . "<br>\n";
echo $produto->getEstoque(2) . "<br>\n";

// exibe os preços de venda
echo "preços de venda : <br>\n";
echo $produto->calculaPrecoVenda(1) . "<br>\n";
echo $produto->calculaPrecoVenda(2) . "<br>\n";
```

```
// vende algumas unidades  
$produto->registraVenda(1, 5);  
$produto->registraVenda(2, 10);  
  
// repõe o estoque  
$produto->registraCompra(1, 5, 16);  
$produto->registraCompra(2, 10, 22);  
  
// exibe os preços de venda atuais  
echo "preços de venda : <br>\n";  
echo $produto->calculaPrecoVenda(1) . "<br>\n";  
echo $produto->calculaPrecoVenda(2) . "<br>\n";  
?>
```

 **Resultado:**

```
estoques:  
10  
20  
preços de venda :  
19,5  
26  
preços de venda :  
20,8  
28,6
```

4.4 Gateways

Até o momento vimos patterns para organização do modelo de negócios da aplicação. Em algum momento, precisaremos armazenar as informações do modelo de negócios em algum meio externo à aplicação, como um banco de dados relacional. Para isso, precisaremos construir uma interface que mantenha o acesso mais transparente possível a esse recurso externo.

Existem diversas formas de se organizar classes para acesso aos dados. A forma mais simples é ter uma classe para manipular o acesso a cada tabela do banco de dados; é ter o que chamamos de Gateway. Um Gateway é uma interface que se comunica com um recurso externo escondendo seus detalhes. Assim, a aplicação só precisará conhecer esta interface para manipular as informações. Todo o acesso aos dados via linguagem SQL, por exemplo, fica contido nessa interface. Dessa forma, evitamos ter a parte do sistema relativa à 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 Gateway Patterns que implementam o acesso às estruturas de dados, como demonstrado na Figura 4.16. A seguir estudaremos alguns. Os mais importantes são Row Data Gateway e Table Data Gateway. Também temos o Active Record, variação do Row Data Gateway.

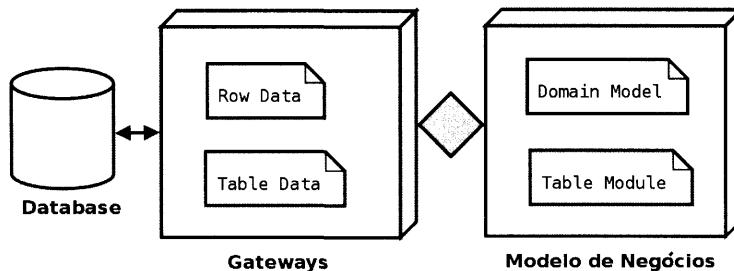


Figura 4.16 – Gateways.

4.4.1 Table Data Gateway

O objetivo do 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. Esta 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 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 negócio e o banco de dados.

Uma das formas mais utilizadas de implementação do pattern Table Module é em conjunto com o pattern Table Data Gateway. Na Figura 4.17 vemos a representação da classe criada neste exemplo.

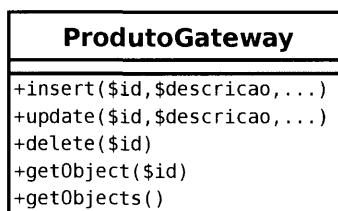


Figura 4.17 – Table Data Gateway.

No programa a seguir estamos criando a classe `ProdutoGateway`. Esta classe, que implementa o pattern Table Data Gateway, possui métodos para inserção, alteração, exclusão e recuperação de registros em base de dados. Note que, a cada método executado, é necessário fazer a identificação do registro sobre o qual o método irá operar. Em algumas vezes tal identificação é realizada pelo ID dos registros, como na operação de exclusão (`delete`); em outras, é necessário informar o registro completo, como na operação de inserção (`insert`). Neste exemplo que segue, estamos inserindo alguns registros. Em seguida, alguns dados serão alterados e outros excluídos. Observe que o pattern Table Data Gateway não implementa lógica de negócios, somente acesso aos dados.

Observação: nestes exemplos não utilizamos a API orientada a objetos criada no capítulo anterior para tornar a demonstração do conceito mais enxuta, embora iremos retomar sua utilização após o pattern Active Record.

`table_gateway.php`

```
<?php
/*
 * classe ProdutoGateway
 * implementa Table Data Gateway
 */
class ProdutoGateway
{
    /*
     * método insert
     * insere dados na tabela de Produtos
     * @param $id      = ID do produto
     * @param $descricao = descrição do produto
     * @param $estoque   = estoque atual
     * @param $preco_custo= preço de custo
    */
    function insert($id, $descricao, $estoque, $preco_custo)
    {
        // cria instrução SQL de insert
        $sql = "INSERT INTO Produtos (id, descricao, estoque, preco_custo)" .
               " VALUES ('$id', '$descricao', '$estoque', '$preco_custo')";

        // instancia objeto PDO
        $conn = new PDO('sqlite:produtos.db');
        $conn->setAttribute(PDO::ATTR_ERRMODE, PDO::ERRMODE_WARNING);
        // executa instrução SQL
        $conn->exec($sql);
        unset($conn);
    }
}
```

```
/*
 * método update
 * altera os dados na tabela de Produtos
 * @param $id      = ID do produto
 * @param $descricao = descrição do produto
 * @param $estoque   = estoque atual
 * @param $preco_custo= preço de custo
 */
function update($id, $descricao, $estoque, $preco_custo)
{
    // cria instrução SQL de UPDATE
    $sql = "UPDATE produtos set descricao = '$descricao', ".
        "    estoque = '$estoque', preco_custo = '$preco_custo' ".
        "    WHERE id = '$id'";

    // instancia objeto PDO
    $conn = new PDO('sqlite:produtos.db');
    $conn->setAttribute(PDO::ATTR_ERRMODE, PDO::ERRMODE_WARNING);
    // executa instrução SQL
    $conn->exec($sql);
    unset($conn);
}

/*
 * método delete
 * deleta um registro na tabela de Produtos
 * @param $id = ID do produto
 */
function delete($id)
{
    // cria instrução SQL de DELETE
    $sql = "DELETE FROM produtos where id='$id'";
    // instancia objeto PDO
    $conn = new PDO('sqlite:produtos.db');
    $conn->setAttribute(PDO::ATTR_ERRMODE, PDO::ERRMODE_WARNING);
    // executa instrução SQL
    $conn->exec($sql);
    unset($conn);
}

/*
 * método getObject
 * busca um registro da tabela de produtos
 * @param $id = ID do produto
 */
function getObject($id)
{
    // cria instrução SQL de SELECT
    $sql = "SELECT * FROM produtos where id='$id'";
```

```
// instancia objeto PDO
$conn = new PDO('sqlite:produtos.db');
$conn->setAttribute(PDO::ATTR_ERRMODE, PDO::ERRMODE_WARNING);
// executa a consulta SQL
$result = $conn->query($sql);
$data = $result->fetch(PDO::FETCH_ASSOC);
unset($conn);
return $data;
}

/*
 * método getObjects
 * lista todos registros da tabela de produtos
*/
function getObjects()
{
    // cria instrução SQL de SELECT
    $sql = "SELECT * FROM produtos";

    // instancia objeto PDO
    $conn = new PDO('sqlite:produtos.db');
    $conn->setAttribute(PDO::ATTR_ERRMODE, PDO::ERRMODE_WARNING);
    // executa a consulta SQL
    $result = $conn->query($sql);
    $data = $result->fetchAll(PDO::FETCH_ASSOC);
    unset($conn);
    return $data;
}
}

// instancia objeto ProdutoGateway
$gateway = new ProdutoGateway;

// insere alguns registros na tabela
$gateway->insert(1, 'Vinho', 10, 10);
$gateway->insert(2, 'Salame', 20, 20);
$gateway->insert(3, 'Queijo', 30, 30);

// efetua algumas alterações
$gateway->update(1, 'Vinho', 20, 20);
$gateway->update(2, 'Salame', 40, 40);
// exclui o produto 3
$gateway->delete(3);
// exibe novamente os registros
echo "Lista de Produtos<br>\n";
print_r($gateway->getObjects());
?>
```

 **Resultado:**

```
Lista de Produtos
Array
[0] => Array
    [id] => 1
    [descricao] => Vinho
    [estoque] => 20
    [preco_custo] => 20
[1] => Array
    [id] => 2
    [descricao] => Salame
    [estoque] => 40
    [preco_custo] => 40
```

O pattern Table Data Gateway também é, muitas vezes, utilizado em conjunto com Data Transfer Object. Data Transfer Object é um objeto simples, sem relacionamentos como associações, agregações ou heranças, utilizado apenas como estrutura para o transporte de dados entre uma chamada de método e outra.

Geralmente esses objetos são pequenos e permitem transportar vários dados por meio de suas propriedades. Como geralmente não persistimos todo objeto do modelo conceitual, mas apenas algumas de suas propriedades, os Data Transfer Objects se constituem uma boa escolha quando se implementa o pattern Table Data Gateway. Você perceberá a simplificação que ocorre nas chamadas de métodos.

No exemplo a seguir estamos reescrevendo o programa anterior utilizando o auxílio de Data Transfer Objects. Note como simplificamos a chamada dos métodos, reduzindo o número de parâmetros necessários para a execução das chamadas. Substituímos os inúmeros parâmetros por um objeto, derivados da classe `Produto`. Para alcançar um efeito similar, poderíamos utilizar um array para transportar os dados. No entanto, um objeto tem algumas vantagens: ele permite que se declare um conjunto específico de propriedades, as quais podem ter valores-padrão. Um objeto também possui um tipo, o que nos permite verificar o tipo do objeto passado como parâmetro, para somente aceitar objetos de uma determinada classe, como no exemplo a seguir.

 **data_transfer.php**

```
<?php
/*
 * classe ProdutoGateway
 * implementa Table Data Gateway com Data Transfer Object
 */
class ProdutoGateway
{
```

```
/*
 * método insert
 * insere dados na tabela de Produtos
 * @param $object = objeto a ser inserido
 */
function insert(Produto $object)
{
    // cria instrução SQL de insert
    $sql = "INSERT INTO Produtos (id, descricao, estoque, preco_custo) .
        " VALUES ('$object->id', '$object->descricao', ".
        "'$object->estoque', '$object->preco_custo')";

    // instancia objeto PDO
    $conn = new PDO('sqlite:produtos.db');
    $conn->setAttribute(PDO::ATTR_ERRMODE, PDO::ERRMODE_WARNING);
    // executa instrução SQL
    $conn->exec($sql);
    unset($conn);
}

/*
 * método update
 * altera os dados na tabela de Produtos
 * @param $object = objeto a ser alterado
 */
function update(Produto $object)
{
    // cria instrução SQL de UPDATE
    $sql = "UPDATE produtos set ".
        "descricao = '$object->descricao', ".
        "estoque = '$object->estoque', ".
        "preco_custo = '$object->preco_custo' ".
        "WHERE id = '$object->id'";

    // instancia objeto PDO
    $conn = new PDO('sqlite:produtos.db');
    $conn->setAttribute(PDO::ATTR_ERRMODE, PDO::ERRMODE_WARNING);
    // executa instrução SQL
    $conn->exec($sql);
    unset($conn);
}

/*
 * método getObject
 * busca um registro da tabela de produtos
 * @param $id = ID do produto
 */
```

```
function getObject($id)
{
    // cria instrução SQL de SELECT
    $sql = "SELECT * FROM produtos where id='$id'";
    // instancia objeto PDO
    $conn = new PDO('sqlite:produtos.db');
    $conn->setAttribute(PDO::ATTR_ERRMODE, PDO::ERRMODE_WARNING);
    // executa consulta SQL
    $result = $conn->query($sql);
    $data = $result->fetch(PDO::FETCH_ASSOC);
    unset($conn);
    return $data;
}

class Produto
{
    public $id;
    public $descricao;
    public $estoque;
    public $preco_custo;
}

// instancia objeto ProdutoGateway
$gateway = new ProdutoGateway;

$vinho = new Produto;
$vinho->id      = 1;
$vinho->descricao = 'Vinho';
$vinho->estoque   = 10;
$vinho->preco_custo = 15;

// insere o objeto no banco de dados
$gateway->insert($vinho);

// exibe o objeto de código 1
print_r($gateway->getObject(1));

$vinho->descricao = 'Vinho Cabernet';
// atualiza o objeto no banco de dados
$gateway->update($vinho);

// exibe o objeto de código 1
print_r($gateway->getObject(1));
?>
```

Resultado:

```
Array
[id] => 1
[descricao] => Vinho
[estoque] => 10
[preco_custo] => 15

Array
[id] => 1
[descricao] => Vinho Cabernet
[estoque] => 10
[preco_custo] => 15
```

4.4.2 Row Data Gateway

O pattern Row Data Gateway provê uma estrutura de classe para persistir um objeto do modelo conceitual no banco de dados, assim como o pattern Table Data Gateway. Contudo, diferentemente deste, no Row Data Gateway temos uma classe em que cada instância sua (objeto) representa um registro diferente do banco de dados.

Este objeto se comporta exatamente como um registro, com suas propriedades representando as colunas do banco de dados, e ainda provê métodos para armazená-lo no banco de dados. Uma das principais vantagens é que o pattern Row Data Gateway é StateFull, ou seja, mantém os valores de suas propriedades ao longo do seu ciclo de vida, não sendo necessário passar todos os valores novamente via parâmetros, como no caso do Table Data Gateway, devido à sua natureza StateLess. Uma desvantagem é que aumentamos o consumo de memória, pois instanciamos naturalmente mais objetos no nosso sistema, tendo em vista que cada objeto agora representará um único registro da tabela, e não a tabela como um todo (como no Table Data Gateway). A diferença de performance reside no fato de que os objetos carregam consigo comportamento, e não somente dados. Apesar dessa desvantagem, este pattern retrata com mais fidelidade o modelo de orientação a objetos, sendo aceitável esta pequena perda de performance para atingirmos maior clareza e entendimento do código-fonte.

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. Na Figura 4.18 vemos a representação da classe criada neste exemplo.



Figura 4.18 – Row Data Gateway.

No programa a seguir estamos demonstrando a implementação do pattern Row Data Gateway. Nele temos a classe `ProdutoGateway`, responsável pela manipulação dos dados de um produto na base de dados. Veja que internamente os métodos da classe referenciam os dados como propriedades do objeto corrente (`$this->id`, `$this->estoque`). Isto é possível porque instanciamos um objeto de `ProdutoGateway` para cada registro que desejamos manipular do banco de dados. Assim, os dados do registro serão representados como propriedades desse objeto. Veja também que, depois de instanciarmos um objeto `ProdutoGateway` e definirmos algumas de suas propriedades, podemos inseri-lo na base de dados somente por meio da chamada do método `insert()`, sem a necessidade de identificar quaisquer parâmetros, uma vez que o objeto já possui conhecimento das propriedades relativas ao registro (como descrição, estoque etc.) necessárias para a inserção.

Observação: Neste exemplo, todas as propriedades serão armazenadas em um array chamado `$data`. Isto por que estamos utilizando os métodos `__get()` e `__set()` para interceptar os acessos de retorno e atribuição à propriedades do objeto.

row_gateway.php

```

<?php
/*
 * classe ProdutoGateway
 * implementa Row Data Gateway
 */
class ProdutoGateway
{
    private $data;

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

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

/*
* método insert
* armazena o objeto na tabela de produtos
*/
function insert()
{
    // cria instrução SQL de insert
    $sql = "INSERT INTO Produtos (id, descricao, estoque, preco_custo) ".
        " VALUES ('{$this->id}',      '{$this->descricao}', ".
        "         '{$this->estoque}', '{$this->preco_custo}')";
    echo $sql . "<br>\n";
    // instancia objeto PDO
    $conn = new PDO('sqlite:produtos.db');
    $conn->setAttribute(PDO::ATTR_ERRMODE, PDO::ERRMODE_WARNING);
    // executa instrução SQL
    $conn->exec($sql);
    unset($conn);
}
/*
* método update
* altera os dados do objeto na tabela de Produtos
*/
function update()
{
    // cria instrução SQL de UPDATE
    $sql = "UPDATE Produtos SET ".
        "     descricao = '{$this->descricao}', ".
        "     estoque   = '{$this->estoque}', ".
        "     preco_custo = '{$this->preco_custo}' ".
        " WHERE id    = '{$this->id}'";
    echo $sql . "<br>\n";
    // instancia objeto PDO
    $conn = new PDO('sqlite:produtos.db');
    $conn->setAttribute(PDO::ATTR_ERRMODE, PDO::ERRMODE_WARNING);
    // executa a instrução SQL
    $conn->exec($sql);
    unset($conn);
}
/*
* método delete
* deleta o objeto da tabela de Produtos
*/
```

```
function delete()
{
    // cria instrução SQL de DELETE
    $sql = "DELETE FROM produtos where id='{$this->id}'";
    echo $sql . "<br>\n";
    // instancia objeto PDO
    $conn = new PDO('sqlite:produtos.db');
    $conn->setAttribute(PDO::ATTR_ERRMODE, PDO::ERRMODE_WARNING);
    // executa instrução SQL
    $conn->exec($sql);
    unset($conn);
}

/*
 * método getObject
 * carrega um objeto a partir da tabela de produtos
 */
function getObject($id)
{
    // cria instrução SQL de SELECT
    $sql = "SELECT * FROM produtos where id='{$id}'";
    echo $sql . "<br>\n";

    // instancia objeto PDO
    $conn = new PDO('sqlite:produtos.db');
    $conn->setAttribute(PDO::ATTR_ERRMODE, PDO::ERRMODE_WARNING);
    // executa consulta SQL
    $result = $conn->query($sql);
    $this->data = $result->fetch(PDO::FETCH_ASSOC);
    unset($conn);
}
}

// insere produtos na base de dados
$vinho= new ProdutoGateway;
$vinho->id      = 1;
$vinho->descricao = 'Vinho Cabernet';
$vinho->estoque   = 10;
$vinho->preco_custo = 10;
$vinho->insert();

$salame= new ProdutoGateway;
$salame->id      = 2;
$salame->descricao = 'Salame';
$salame->estoque   = 20;
$salame->preco_custo = 20;
$salame->insert();
```

```
// 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

de métodos relativos ao modelo de negócios, como os métodos `registraCompra()`, `registraVenda()` e `calculaPrecoVenda()`. A presença destes métodos torna essa classe em um Active Record, ou seja, um objeto que se comporta exatamente como um registro do banco de dados e ainda oferece métodos relativos à lógica de negócios da aplicação. No exemplo a seguir estamos inserindo um produto `Vinho` na tabela de produtos, por meio do método `insert()`. Em seguida, estamos registrando uma venda pelo método `registraVenda()` e repondo o estoque pelo método `registraCompra()`. Finalmente estamos atualizando o registro na base de dados por meio do método `update()`. Na Figura 4.19 temos a representação da classe criada neste exemplo.

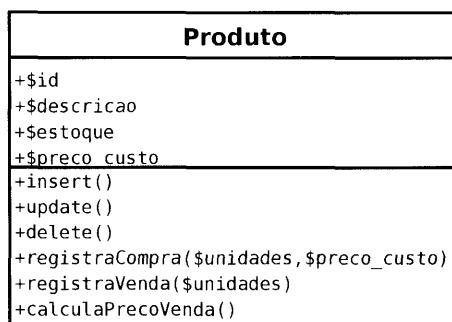


Figura 4.19 – Active Record.

active_record.php

```
<?php
/*
 * classe Produto
 * implementa Active Record
 */
class Produto
{
    private $data;

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

    function __set($prop, $value)
    {
        $this->data[$prop] = $value;
    }
    /*
     * método insert
     * armazena o objeto na tabela de produtos
    */
}
```

```
function insert()
{
    // cria instrução SQL de insert
    $sql = "INSERT INTO Produtos (id, descricao, estoque, preco_custo) .
            " VALUES ('{$this->id}', '{$this->descricao}', ".
            "         '{$this->estoque}', '{$this->preco_custo}')";

    // instancia objeto PDO
    $conn = new PDO('sqlite:produtos.db');
    $conn->setAttribute(PDO::ATTR_ERRMODE, PDO::ERRMODE_WARNING);
    // executa instrução SQL
    $conn->exec($sql);
    unset($conn);
}

/*
 * método update
 * altera os dados do objeto na tabela de Produtos
 */
function update()
{
    // cria instrução SQL de UPDATE
    $sql = "UPDATE produtos set ".
        "descricao = '{$this->descricao}', ".
        "estoque = '{$this->estoque}', ".
        "preco_custo = '{$this->preco_custo}' ".
        "WHERE id = '{$this->id}'";

    // instancia objeto PDO
    $conn = new PDO('sqlite:produtos.db');
    $conn->setAttribute(PDO::ATTR_ERRMODE, PDO::ERRMODE_WARNING);
    // executa instrução SQL
    $conn->exec($sql);
    unset($conn);
}

/*
 * método delete
 * deleta o objeto da tabela de Produtos
 */
function delete()
{
    // cria instrução SQL de DELETE
    $sql = "DELETE FROM produtos where id='{$this->id}'";
    // instancia objeto PDO
    $conn = new PDO('sqlite:produtos.db');
    $conn->setAttribute(PDO::ATTR_ERRMODE, PDO::ERRMODE_WARNING);
    // executa instrução SQL
    $conn->exec($sql);
    unset($conn);
}
```

```
/*
 * método registraCompra
 * registra uma compra, atualiza custo e incrementa o estoque atual
 * @param $unidades = unidades adquiridas
 * @param $preco_custo= novo preco de custo
 */
public function registraCompra($unidades, $preco_custo)
{
    $this->preco_custo = $preco_custo;
    $this->estoque += $unidades;
}

/*
 * método registraVenda
 * registra uma venda e decremente o estoque
 * @param $unidades = unidades vendidas
 */
public function registraVenda($unidades)
{
    $this->estoque -= $unidades;
}

/*
 * método calculaPrecoVenda
 * retorna o preco de venda, baseado em uma margem de 30% sobre o custo
 */
public function calculaPrecoVenda()
{
    return $this->preco_custo * 1.3;
}

// instancia objeto Produto
$vinho= new Produto;
$vinho->id = 1;
$vinho->descricao = 'Vinho Cabernet';
$vinho->estoque = 10;
$vinho->preco_custo = 10;
$vinho->insert();

$vinho->registraVenda(5);
echo 'estoque: ' . $vinho->estoque . "<br>\n";
echo 'preco_custo: ' . $vinho->preco_custo . "<br>\n";
echo 'preco_venda: ' . $vinho->calculaPrecoVenda() . "<br>\n";

$vinho->registraCompra(10, 20);
$vinho->update();
echo 'estoque: ' . $vinho->estoque . "<br>\n";
```

```
echo 'preco_custo: ' . $vinho->preco_custo . "<br>\n";
echo 'preco_venda: ' . $vinho->calculaPrecoVenda() . "<br>\n";
?>
```

Resultado:

```
estoque:      5
preco_custo: 10
preco_venda: 13
estoque:     15
preco_custo: 20
preco_venda: 26
```

4.4.4 Data Mapper

Como já vimos anteriormente, a orientação a objetos nos proporciona uma gama de relacionamentos entre os objetos que enriquecem a representação de idéias do modelo conceitual. Tais relacionamentos passam por associações, agregações e heranças, cuja utilização facilita a compreensão do domínio da aplicação.

Enquanto esses relacionamentos nos auxiliam a representar complexas estruturas em nossas aplicações, distanciamo-nos cada vez mais do modelo de dados (tabular) de nossa aplicação, pois nele não conseguimos representar com tamanha riqueza de detalhes os relacionamentos existentes, o que dificulta a tarefa de mapear os objetos para o banco de dados.

Até agora vimos três tipos de gateway (Table Data, Row Data e Active Record) que funcionam muito bem quando a representação do modelo conceitual da aplicação é próximo ao do modelo de dados. No entanto, quando precisamos utilizar vários tipos de relacionamento entre os objetos conceituais, temos de utilizar uma forma mais flexível de mapear tais objetos para o modelo de dados.

O pattern Data Mapper se constitui em uma camada da aplicação que separa os objetos conceituais do banco de dados, permitindo transferir dados entre uma camada e outra de forma transparente, sem que os objetos do modelo conceitual precisem implementar métodos de persistência.

Para que o Data Mapper possa fazer isso, é necessário que o objeto conceitual (camada de negócio) ofereça métodos que permitam o Data Mapper inspecionar o valor de suas propriedades, uma vez que torná-las `public` não é a melhor solução.

A classe que implementará o Data Mapper deverá prover métodos de persistência e recuperação (obtenção) dos objetos de negócio, além de métodos para pesquisar objetos com base em diferentes padrões (busca por nome, por idade no caso de uma pessoa).

No programa a seguir temos a classe `Produto` e a classe `Venda`. A primeira recebe algumas informações em seu método construtor (descrição, estoque e preço de custo) e oferece o método `getDescricao()` para retornar sua descrição. A segunda, `Venda`, permite agregar itens (neste caso, produtos) pelo método `addItem()`, que recebe a quantidade vendida e um objeto representando o produto. O método `getItens()` retorna os itens agregados de uma venda.

Para armazenar esse complexo relacionamento de agregação na base de dados poderíamos utilizar um dos gateways vistos anteriormente. Apesar disso, o mapeamento dos dados para a base de dados não seria transparente, uma vez que teríamos de lidar com os relacionamentos manualmente. Assim, estamos criando a classe `VendaMapper`, responsável pelo mapeamento do objeto `Venda`, juntamente com seus produtos, para a base de dados. Esta classe provê o método `insert()`, que recebe um objeto `Venda` e trata de inspecioná-lo, obtendo as informações necessárias para que seja realizado o mapeamento dos objetos para o banco de dados, como ilustrado pelo seguinte diagrama:

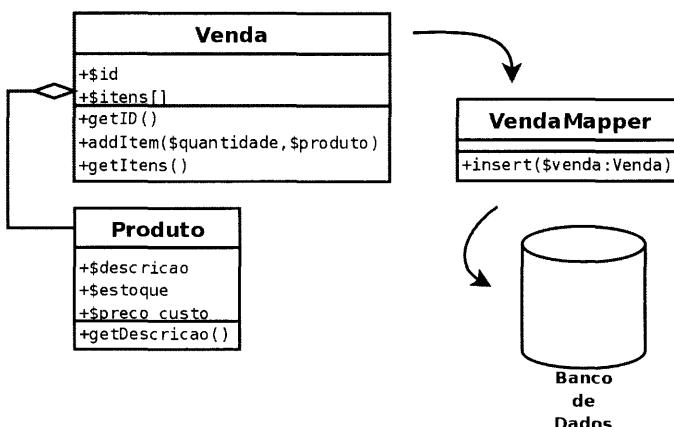


Figura 4.20 – Data Mapper.

data_mapper.php

```

<?php
/*
 * class Produto
 * representa um Produto a ser vendido
 */
final class Produto
{
    private $descricao;      // descricao do produto
    private $estoque;        // estoque atual
    private $preco_custo;    // preco de custo
  
```

```
/*
 * método construtor
 * define alguns valores iniciais
 * @param $descricao = descrição do produto
 * @param $estoque   = estoque atual
 * @param $preco_custo= preço de custo
 */
public function __construct($descricao, $estoque, $preco_custo)
{
    $this->descricao = $descricao;
    $this->estoque   = $estoque;
    $this->preco_custo= $preco_custo;
}

/*
 * método getDescricao
 * retorna a descrição do produto
 */
public function getDescricao()
{
    return $this->descricao;
}

/*
 * classe Venda
 * representa uma Venda de Produtos
 */
final class Venda
{
    private $id;
    private $itens; // itens da cesta

    /*
     * método construtor
     * instancia uma nova venda
     * @param $id = Identificador
     */
    function __construct($id)
    {
        $this->id = $id;
    }

    /*
     * método getID
     * retorna o identificador
     */
}
```

```
function getID()
{
    return $this->id;
}

/*
 * método addItem
 * adiciona um item na cesta
 * @param $quantidade = quantidade vendida
 * @param $produto    = objeto produto
 */
public function addItem($quantidade, Produto $produto)
{
    $this->itens[] = array($quantidade, $produto);
}

/*
 * método getItens
 * retorna o array de itens da cesta
 */
public function getItens()
{
    return $this->itens;
}

/*
 * class Venda Mapper
 * Implementa Data Mapper para a classe Venda
 */
final class VendaMapper
{
    function insert(Venda $venda)
    {
        $id = $venda->getID();
        $date = date("Y-m-d");

        // insere a venda no banco de dados
        $sql = "INSERT INTO venda (id, data) values ('$id', '$date')";
        echo $sql . "<br>\n";

        // percorre os itens vendidos
        foreach ($venda->getItens() as $item)
        {
            $quantidade = $item[0];
            $produto    = $item[1];
            $descricao  = $produto->getDescricao();
```

```
// insere os itens da venda no banco de dados
$sql = "INSERT INTO venda_items (ref_venda, produto, quantidade)".
        " values ('$id', '$descricao', '$quantidade')";
echo $sql . "<br>\n";
}

}

}

// instancia objeto Venda
$venda= new Venda(1000);

// adiciona alguns produtos
$venda->addItem(3, new Produto('Vinho', 10, 15));
$venda->addItem(2, new Produto('Salame', 20, 20));
$venda->addItem(1, new Produto('Queijo', 30, 10));

// Data Mapper persiste a venda
VendaMapper::insert($venda);
?>
```

Resultado:

```
INSERT INTO venda (id, data) values ('1000', '2007-04-09')
INSERT INTO venda_items (ref_venda, produto, quantidade) values ('1000', 'Vinho', '3')
INSERT INTO venda_items (ref_venda, produto, quantidade) values ('1000', 'Salame', '2')
INSERT INTO venda_items (ref_venda, produto, quantidade) values ('1000', 'Queijo', '1')
```

4.5 Manipulando objetos

4.5.1 Introdução

Até o momento estudamos diversos patterns de persistência de objetos em bases de dados. Não existe um pattern que seja melhor que outro. Existem casos em que a utilização de um é mais apropriada que de outro. Precisaremos escolher um desses patterns para utilizar ao longo do livro nos exemplos subsequentes. Em decorrência da facilidade de uso e da simplicidade dos exemplos que construiremos, nos quais não existirão relações complexas entre as tabelas (o que poderia exigir um Data Mapper), decidimos optar pelo pattern Active Record.

Um Active Record contém métodos de persistência do objeto na base de dados e também métodos relativos ao modelo de negócios. Tais métodos de persistência, que devem ser comuns a todos os objetos de nossa aplicação, podem ser automatizados se criarmos algumas convenções. Esta automatização dos métodos de um Active

Record poderia estar localizada em uma superclasse, herdada por todos os nossos objetos. Quando criamos uma superclasse que reúne funcionalidades em comum para toda uma camada de objetos, estamos utilizando um pattern conhecido como Layer Supertype. Neste caso movemos aquelas funcionalidades que são comuns a todos os objetos para essa superclasse.

Em um Active Record, temos algumas operações que são necessárias para todo e qualquer objeto. Dentre elas podemos destacar a operação de armazenar um objeto na base de dados (que vamos chamar de `store`), a operação de remover um objeto da base de dados (que vamos chamar de `delete`), a operação de ler um registro da base de dados, instanciando o objeto correspondente para a aplicação (que vamos chamar de `load`). Essas operações, que são comuns a todo Active Record podem ser movidas a um nível superior, uma superclasse (um Layer Supertype). Em nossa aplicação, chamaremos esta classe de `TRecord` por implementar o padrão Active Record.

Na Figura 4.21 retratamos a classe `TRecord` em um possível cenário de modelo de negócios. Por ser uma Layer Supertype, ela fornece todos os recursos necessários para persistência de um objeto em bases de dados (`store`, `load`, `delete` etc.). Dessa forma, a maioria das classes do modelo de negócios só precisará estendê-la e, em alguns casos específicos, poderá inclusive reescrever algum de seus métodos.

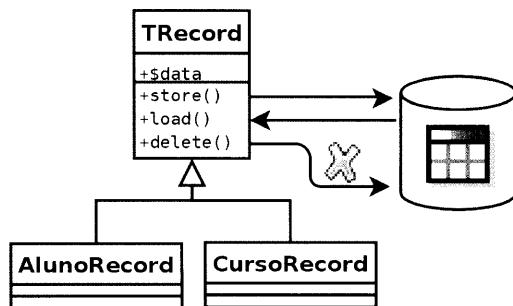


Figura 4.21 – Classe `TRecord` e o modelo de negócio.

A seguir temos o código da classe `TRecord`. Em seu método construtor, podemos passar opcionalmente o ID de um objeto. Neste caso, o objeto correspondente será carregado automaticamente por meio do método `load()`.

`TRecord.class.php`

```

<?
/*
 * classe TRecord
 * Esta classe provê os métodos necessários para persistir e
 * recuperar objetos da base de dados (Active Record)
 */
  
```

```

abstract class TRecord
{
    protected $data; // array contendo os dados do objeto

    /* método __construct()
     * instancia um Active Record. Se passado o $id, já carrega o objeto
     * @param [$id] = ID 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. Nestes casos em que um Active Record é clonado, ele deve manter todas as suas propriedades, com exceção de seu ID, por isso estamos eliminando 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.

```

/*
 * método __clone()
 * executado quando o objeto for clonado.
 * limpa o ID para que seja gerado um novo ID para o clone.
 */
public function __clone()
{
    unset($this->id);
}

```

Sempre que um valor for atribuído a uma propriedade do objeto, o método `__set()` será executado, interceptando esta atribuição. O valor a ser atribuído será armazenado no array `$data`, indexado pelo nome da propriedade. Antes disto, porém, será verificada a existência de um método nomeado por `set-<propriedade>` definido pelo usuário (por meio da função `method_exists`). Caso esse método exista, ele terá prioridade de execução. Veremos, na seção Encapsulamento, como o usuário poderá definir métodos para proteger a atribuição de valores.

```
/*
 * método __set()
 * executado sempre que uma propriedade for atribuída.
 */
private function __set($prop, $value)
{
    // verifica se existe método set_<propriedade>
    if (method_exists($this, 'set_'.$prop))
    {
        // executa o método set_<propriedade>
        call_user_func(array($this, 'set_'.$prop), $value);
    }
    else
    {
        // atribui o valor da propriedade
        $this->data[$prop] = $value;
    }
}
```

Sempre que uma propriedade for requisitada, o método `__get()` será executado. O valor da propriedade será lido a partir do array `$data`, mas, antes disso, será verificada a existência de um método nomeado por `get_<propriedade>` definido pelo usuário. Caso esse método exista, ele será executado no lugar. Mais adiante, na seção Lazy initialization, veremos um caso de uso dessa funcionalidade, a qual permite que o programador influencie no retorno de propriedades do objeto.

```
/*
 * método __get()
 * executado sempre que uma propriedade for requerida
 */
private function __get($prop)
{
    // verifica se existe método get_<propriedade>
    if (method_exists($this, 'get_'.$prop))
    {
        // executa o método get_<propriedade>
        return call_user_func(array($this, 'get_'.$prop));
    }
    else
    {
        // retorna o valor da propriedade
        return $this->data[$prop];
    }
}
```

Seguindo o código de nossa classe `TRecord`, temos o método `getEntity()`. Este método é responsável por retornar o nome da classe atual subtraindo os últimos seis caracteres. Se a nossa classe se chamar `AlunoRecord`, este método irá retornar “Aluno”. Se a nossa classe se chamar `InscricaoRecord`, este método irá retornar “Inscricao”. O objetivo aqui é adotar esse nome para operações realizadas no banco de dados, pois, dessa forma, o nome da classe será considerado como o nome da tabela.

```
/*
 * método getEntity()
 * retorna o nome da entidade (tabela)
 */
private function getEntity()
{
    // obtém o nome da classe
    $classe = strtolower(get_class($this));
    // retorna o nome da classe - "Record"
    return substr($classe, 0, -6);
}
```

O método `fromArray()` será utilizado para preencher os atributos de um Active Record com os dados de um array, de modo que os índices deste array são os atributos do objeto. O método `toArray()` será utilizado para retornar todos os atributos de um objeto (armazenados na propriedade `$data`) em forma de array.

```
/*
 * método fromArray
 * preenche os dados do objeto com um array
 */
public function fromArray($data)
{
    $this->data = $data;
}

/*
 * método toArray
 * retorna os dados do objeto como array
 */
public function toArray()
{
    return $this->data;
}
```

O método `store()` é responsável por persistir um objeto no banco de dados. Este método, por meio da função `empty()`, faz uma verificação para descobrir se o objeto possui o atributo `ID` com algum valor e também verifica, por meio do método `load()`, se o objeto já existe no banco de dados.

Caso o objeto tenha seu ID vazio ou não esteja armazenado no banco de dados, devemos utilizar a classe `TSqlInsert` para persisti-lo pela primeira vez, realizando uma operação de `INSERT`. Caso contrário, devemos utilizar a classe `TSqlUpdate` e alterar suas propriedades no banco de dados.

O fato de o objeto que estamos persistindo não possuir seu campo de ID com valor significa que é um objeto que está para ser persistido pela primeira vez e, dessa forma, precisamos gerar um ID para ele. Neste caso existem diversas estratégias: podemos utilizar recursos do próprio banco de dados como auto-increment ou seqüências. Entretanto, nem todos os bancos de dados oferecem este recurso, e os que oferecem trabalham de maneiras diferentes.

A estratégia que iremos utilizar para geração dos IDs de novos objetos será utilizar a função `MAX()`. Pelo método `getLast()`, descobriremos o maior ID já inserido na tabela e, então, iremos incrementar este valor. Sabemos que esta é uma solução simplista, mas atende aos propósitos.

Para executar a instrução na base de dados, o método `store()` obtém a transação ativa por meio do método `TTransaction::get()`. Para isto, é necessário existir uma transação aberta; caso contrário uma exceção será lançada. Em seguida, o método `store()` registra a instrução SQL no arquivo de log e a executa por meio do método `exec()` da conexão.

```
/*
 * método store()
 * armazena o objeto na base de dados e retorna
 * o número de linhas afetadas pela instrução SQL (zero ou um)
 */
public function store()
{
    // verifica se tem ID ou se existe na base de dados
    if (empty($this->data['id']) or (!$this->load($this->id)))
    {
        // incrementa o ID
        $this->id = $this->getLast() +1;
        // cria uma instrução de insert
        $sql = new TSqlInsert;
        $sql->setEntity($this->getEntity());
        // percorre os dados do objeto
        foreach ($this->data as $key => $value)
        {
            // passa os dados do objeto para o SQL
            $sql->setRowData($key, $this->$key);
        }
    }
}
```

```
else
{
    // instancia instrução de update
    $sql = new TSq1Update;
    $sql->setEntity($this->getEntity());
    // cria um critério de seleção baseado no ID
    $criteria = new TCriteria;
    $criteria->add(new TFilter('id', '=', $this->id));
    $sql->setCriteria($criteria);
    // percorre os dados do objeto
    foreach ($this->data as $key => $value)
    {
        if ($key !== 'id') // o ID não precisa ir no UPDATE
        {
            // passa os dados do objeto para o SQL
            $sql->setRowData($key, $this->$key);
        }
    }
}
// obtém transação ativa
if ($conn = TTransaction::get())
{
    // faz o log e executa o SQL
    TTransaction::log($sql->getInstruction());
    $result = $conn->exec($sql->getInstruction());
    // retorna o resultado
    return $result;
}
else
{
    // se não tiver transação, retorna uma exceção
    throw new Exception('Não há transação ativa!!');
}
```

O método `load()` será responsável por ler um registro do banco de dados e preencher as propriedades do objeto (atributo `$data`) com esses dados. Para isso, utiliza-se da classe `TSq1Select` para realizar a consulta no banco de dados. O método `load()` sempre irá selecionar todas as colunas da tabela (*). O critério de seleção de dados (`TCriteria`) será baseado no atributo `ID`. Após isto, o método detecta a transação ativa pelo método `TTransaction::get()`, registra a operação no arquivo de log e executa a consulta SQL, retornando os dados para a aplicação. Note que os dados são retornados na forma de objeto (`fetchObject`) e a classe deste objeto será a mesma que a classe atual (`get_class($this)`).

```

/*
 * método load()
 * recupera (retorna) um objeto da base de dados
 * através de seu ID e instancia ele na memória
 * @param $id = ID do objeto
 */
public function load($id)
{
    // instancia instrução de SELECT
    $sql = new TSqlSelect;
    $sql->setEntity($this->getEntity());
    $sql->addColumn('*');

    // cria critério de seleção baseado no ID
    $criteria = new TCriteria;
    $criteria->add(new TFilter('id', '=', $id));
    // define o critério de seleção de dados
    $sql->setCriteria($criteria);
    // obtém transação ativa
    if ($conn = TTransaction::get())
    {
        // cria mensagem de log e executa a consulta
        TTransaction::log($sql->getInstruction());
        $result= $conn->Query($sql->getInstruction());
        // se retornou algum dado
        if ($result)
        {
            // retorna os dados em forma de objeto
            $object = $result->fetchObject(get_class($this));
        }
        return $object;
    }
    else
    {
        // se não tiver transação, retorna uma exceção
        throw new Exception('Não há transação ativa!!');
    }
}

```

O método `delete()` será responsável por excluir o objeto atual da base de dados e poderá receber opcionalmente um `$id` como parâmetro. Neste caso, assumirá este `$id` a ser excluído, mas o comportamento-padrão é excluir o objeto atual (`$this->id`). O método `delete()` instancia um objeto da classe `TSqlDelete` e cria um critério baseado no ID do objeto. Posteriormente, detecta, pelo método `TTransaction::get()`, se existe uma transação já aberta com o banco de dados, registra a instrução SQL no arquivo de log e executa o comando.

```
/*
 * método delete()
 * exclui um objeto da base de dados através de seu ID.
 * @param $id = ID do objeto
 */
public function delete($id = NULL)
{
    // o ID é o parâmetro ou a propriedade ID
    $id = $id ? $id : $this->id;
    // instancia uma instrução de DELETE
    $sql = new TSqlDelete;
    $sql->setEntity($this->getEntity());

    // cria critério de seleção de dados
    $criteria = new TCriteria;
    $criteria->add(new TFilter('id', '=', $id));
    // define o critério de seleção baseado no ID
    $sql->setCriteria($criteria);

    // obtém transação ativa
    if ($conn = TTransaction::get())
    {
        // faz o log e executa o SQL
        TTransaction::log($sql->getInstruction());
        $result = $conn->exec($sql->getInstruction());
        // retorna o resultado
        return $result;
    }
    else
    {
        // se não tiver transação, retorna uma exceção
        throw new Exception('Não há transação ativa!!');
    }
}
```

O método `getLast()` é utilizado apenas internamente. O seu objetivo é retornar o último ID armazenado na tabela, para que se calcule automaticamente o próximo ID a ser inserido naquela tabela, no caso de objetos que estão sendo persistidos pela primeira vez pelo método `store()`.

```
/*
 * método getLast()
 * retorna o último ID
 */
private function getLast()
{
```

```

// inicia transação
if ($conn = TTransaction::get())
{
    // instancia instrução de SELECT
    $sql = new TSq1Select;
    $sql->addColumn('max(ID) as ID');
    $sql->setEntity($this->getEntity());
    // cria log e executa instrução SQL
    TTransaction::log($sql->getInstruction());
    $result= $conn->Query($sql->getInstruction());
    // retorna os dados do banco
    $row = $result->fetch();
    return $row[0];
}
else
{
    // se não tiver transação, retorna uma exceção
    throw new Exception('Não há transação ativa!!');
}
}

?>

```

4.5.2 Exemplos

Para demonstrar a utilização da classe criada, criaremos uma série de exemplos procurando ilustrar os mais diversos aspectos envolvidos no momento de trabalharmos com objetos. Nesses exemplos, veremos aplicações para inserir objetos na base de dados, alterar atributos de objetos, retornar objetos, excluir, dentre outras. Para isso, precisaremos criar uma estrutura mínima de tabelas para contextualizar nossos programas. Assim, utilizaremos a seguinte estrutura de tabelas nos nossos programas:

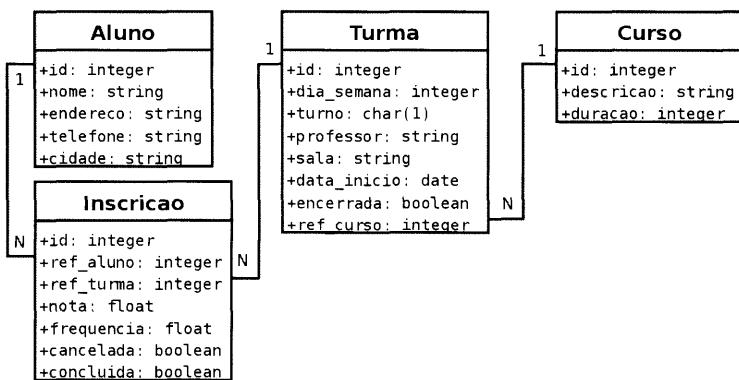


Figura 4.22 – Estrutura de tabelas.

Essa estrutura de tabelas procura demonstrar uma aplicação para controlar uma pequena escola de cursos técnicos. Temos uma tabela para armazenar alunos e uma tabela de inscrições, a qual relaciona os alunos com determinadas turmas. Uma turma é a ocorrência de um curso, sendo que cada turma terá um professor, uma sala, uma data de início, se está encerrada, o turno (manhã, tarde, noite), dentre outros. As informações de nota e freqüência serão armazenadas na própria tabela de inscrições, e a tabela de cursos terá apenas código, descrição e duração (em horas). A seguir, temos as instruções SQL relativas à criação desse banco de dados.

 **banco.sql**

```
CREATE TABLE aluno (id integer, nome varchar(40), endereco varchar(40),
                    telefone varchar(40), cidade varchar(40));
CREATE TABLE inscricao (id serial, ref_aluno integer, ref_turma integer,
                       nota float, frequencia float, cancelada boolean,
                       concluida boolean);
CREATE TABLE turma (id integer, dia_semana integer, turno char(1),
                    professor varchar(40), sala varchar(20), data_inicio date,
                    encerrada boolean, ref_curso integer);
CREATE TABLE curso (id integer, descricao varchar(40), duracao integer);
```

A seguir, apresentaremos uma série de exemplos sobre persistência de objetos em base de dados utilizando as classes criadas. No início de cada programa, deverá ser introduzida a função `__autoload()` responsável por carregar as classes automaticamente no momento em que estas são criadas. Observe que as classes deverão estar dentro da pasta `app.ado`, que é a pasta lida por esta função.

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

4.5.3 Novo objeto

Neste primeiro programa que utiliza a superclasse `TRecord`, iremos demonstrar como se dá a persistência (gravação) de novos objetos na base de dados. Para isto, estamos criando duas classes: `AlunoRecord` (que irá persistir objetos na tabela `Aluno`) e `CursoRecord` (que irá persistir objetos na tabela `Curso`). Por enquanto não iremos adicionar um comportamento a essas classes, por isso elas estão vazias. Faremos isso posteriormente na seção Aspectos avançados. As classes `AlunoRecord` e `CursoRecord`, por serem filhas de `TRecord`, possuem todas as funcionalidades desta.

Nosso código que irá manipular os objetos e consequentemente irá realizar interações com o banco de dados está todo contido dentro de um bloco `try/catch` para tratamento de exceções. Sempre que qualquer erro acontecer durante o `try{}`, o programa será abortado e a execução seguirá para o bloco `catch{}`, exibindo a mensagem de erro na tela e desfazendo a transação.

Esse programa inicia abrindo uma transação com a base `pg_livro` e, então, definimos o arquivo de log para o SQL e demais mensagens de debug. Em seguida, instanciamos dois objetos da classe `AlunoRecord`, definimos algumas de suas propriedades e executamos o método `store()`, responsável por persistir tais objetos na base de dados. Ao mesmo tempo em que estamos persistindo tais objetos, estamos também escrevendo algumas mensagens no arquivo de log para podermos entendê-lo mais facilmente após a execução do programa, contextualizando as operações.

Depois de persistirmos alguns objetos `AlunoRecord`, estamos instanciando dois objetos da classe `CursoRecord` e armazenando-os no banco de dados, novamente pelo método `store()`. Por último finalizamos a transação, que irá executar um comando `commit`, aplicando todas as operações que realizamos.

model_novo.php

```
<?php
// inserir função __autoload() para carregar as classes...

/*
 * classe AlunoRecord, filha de TRecord
 * persiste um Aluno no banco de dados
 */
class AlunoRecord extends TRecord { }

/*
 * classe CursoRecord, filha de TRecord
 * persiste um Curso no banco de dados
 */
```

```
class CursoRecord extends TRecord { }

// insere novos objetos no banco de dados
try
{
    // inicia transação com o banco 'pg_livro'
    TTransaction::open('pg_livro');
    // define o arquivo para LOG
    TTransaction::setLogger(new TLoggerTXT('/tmp/log1.txt'));

    // armazena esta frase no arquivo de LOG
    TTransaction::log("** inserindo alunos");

    // instancia um novo objeto Aluno
    $daline= new AlunoRecord;
    $daline->nome      = 'Daline Dall Oggio';
    $daline->endereco = 'Rua da Conceição';
    $daline->telefone = '(51) 1111-2222';
    $daline->cidade   = 'Cruzeiro do Sul';
    $daline->store(); // armazena o objeto

    // instancia um novo objeto Aluno
    $william= new AlunoRecord;
    $william->nome      = 'William Scatolla';
    $william->endereco = 'Rua de Fátima';
    $william->telefone = '(51) 1111-4444';
    $william->cidade   = 'Encantado';
    $william->store(); // armazena o objeto

    // armazena esta frase no arquivo de LOG
    TTransaction::log("** inserindo cursos");
    // instancia um novo objeto Curso
    $curso = new CursoRecord;
    $curso->descricao = 'Orientação a Objetos com PHP';
    $curso->duracao  = 24;
    $curso->store(); // armazena o objeto

    // instancia um novo objeto Curso
    $curso = new CursoRecord;
    $curso->descricao = 'Desenvolvendo em PHP-GTK';
    $curso->duracao  = 32;
    $curso->store(); // armazena o objeto

    // finaliza a transação
    TTransaction::close();
    echo "Registros inseridos com Sucesso<br>\n";
}
```

```

catch (Exception $e) // em caso de exceção
{
    // exibe a mensagem gerada pela exceção
    echo '<b>Erro</b>' . $e->getMessage();
    // desfaz todas alterações no banco de dados
    TTransaction::rollback();
}
?>

```

/tmp/log1.txt

```

2007-06-05 15:05:56 :: ** inserindo alunos
2007-06-05 15:05:56 :: SELECT max(ID) as ID FROM aluno
2007-06-05 15:05:56 :: INSERT INTO aluno (nome, endereco, telefone, cidade, id)
                         values ('Daline Dall Oglia', 'Rua da Conceição', '(51) 1111-2222',
                         'Cruzeiro do Sul', 1)
2007-06-05 15:05:56 :: SELECT max(ID) as ID FROM aluno
2007-06-05 15:05:56 :: INSERT INTO aluno (nome, endereco, telefone, cidade, id)
                         values ('William Scatolla', 'Rua de Fátima', '(51) 1111-4444', 'Encantado', 2)
2007-06-05 15:05:56 :: ** inserindo cursos
2007-06-05 15:05:56 :: SELECT max(ID) as ID FROM curso
2007-06-05 15:05:56 :: INSERT INTO curso (descricao, duracao, id)
                         values ('Orientação a Objetos com PHP', 24, 1)
2007-06-05 15:05:56 :: SELECT max(ID) as ID FROM curso
2007-06-05 15:05:56 :: INSERT INTO curso (descricao, duracao, id)
                         values ('Desenvolvendo em PHP-GTK', 32, 2)

```

4.5.4 Obter objeto

No programa que segue procuramos demonstrar a forma pela qual iremos obter (retornar) os objetos a partir da base de dados. Para tanto, novamente declaramos nossas classes `AlunoRecord` (que irá manipular a tabela de `Aluno`) e `CursoRecord` (que irá manipular a tabela de `Curso`). O código está novamente contido em um bloco `try/catch` para controlar a ocorrência de exceções. No início do programa, abrimos conexão com o banco de dados `pg_livro` e definimos o arquivo onde serão armazenadas as mensagens de log.

Depois disso, instanciamos objetos `AlunoRecord`, passando o `ID` do objeto que devemos retornar como parâmetro. Desta forma, o objeto será instanciado já com os dados daquele registro, que será imediatamente carregado na memória. Para ilustrar o carregamento do objeto, estamos exibindo algumas de suas propriedades (`nome` e `endereco`) na tela.

Depois de obtermos alguns objetos `AlunoRecord`, estamos instanciando um objeto `CursoRecord` e carregando primeiro o registro de `ID` 1 e depois o registro de `ID` 2 na

memória e exibindo uma de suas propriedades (descrição) na tela. Ao final, estamos fechando a transação.

model_get.php

```
<?php  
// inserir função __autoload() para carregar as classes...  
  
/*  
 * classe AlunoRecord, filha de TRecord  
 * persiste um Aluno no banco de dados  
 */  
class AlunoRecord extends TRecord { }  
/*  
 * classe CursoRecord, filha de TRecord  
 * persiste um Curso no banco de dados  
 */  
class CursoRecord extends TRecord { }  
  
// obtém objetos do banco de dados  
try  
{  
    // inicia transação com o banco 'pg_livro'  
    TTransaction::open('pg_livro');  
    // define o arquivo para LOG  
    TTransaction::setLogger(new TLoggerTXT('/tmp/log2.txt'));  
  
    // exibe algumas mensagens na tela  
    echo "obtendo alunos<br>\n";  
    echo "=====<br>\n";  
  
    // obtém o aluno de ID 1  
    $aluno= new AlunoRecord(1);  
    echo 'Nome : ' . $aluno->nome . "<br>\n";  
    echo 'Endereço : ' . $aluno->endereco . "<br>\n";  
  
    // obtém o aluno de ID 2  
    $aluno= new AlunoRecord(2);  
    echo 'Nome : ' . $aluno->nome . "<br>\n";  
    echo 'Endereço : ' . $aluno->endereco . "<br>\n";  
  
    // obtém alguns cursos  
    echo "<br>\n";  
    echo "obtendo cursos<br>\n";  
    echo "=====<br>\n";
```

```

// obtém o curso de ID 1
$curso= new CursoRecord(1);
echo 'Curso : ' . $curso->descricao . "<br>\n";

// obtém o curso de ID 2
$curso= new CursoRecord(2);
echo 'Curso : ' . $curso->descricao . "<br>\n";

// finaliza a transação
TTransaction::close();
}

catch (Exception $e) // em caso de exceção
{
    // exibe a mensagem gerada pela exceção
    echo '<b>Erro</b>' . $e->getMessage();
    // desfaz todas alterações no banco de dados
    TTransaction::rollback();
}
?>

```

Resultado:

obtendo alunos

```
=====
Nome      : Daline Dall Oglia
Endereço : Rua da Conceição
Nome      : William Scatolla
Endereço : Rua de Fátima
```

obtendo cursos

```
=====
Curso : Orientação a Objetos com PHP
Curso : Desenvolvendo em PHP-GTK
```

/tmp/log2.txt

```

2007-06-05 15:07:20 :: SELECT * FROM aluno WHERE (id = 1)
2007-06-05 15:07:20 :: SELECT * FROM aluno WHERE (id = 2)
2007-06-05 15:07:20 :: SELECT * FROM curso WHERE (id = 1)
2007-06-05 15:07:20 :: SELECT * FROM curso WHERE (id = 2)
```

4.5.5 Alterar objeto

Neste programa, iremos alterar as propriedades de alguns objetos já existentes na base de dados. Para isso, abrimos uma transação com a base pg_livro e definimos o arquivo de log em /tmp/log3.txt.

O primeiro passo é instanciar um objeto `AlunoRecord`. Por meio do seu método `load()` carregaremos os seus dados em memória. Caso o objeto realmente exista, alteraremos o seu telefone e executaremos novamente o método `store()` para armazená-lo no banco de dados.

Depois disso, repetimos a mesma operação, agora para cursos. Para tanto, instanciamos um objeto `CursoRecord`, executamos o `load()` para retornar o curso de ID 1, alteramos sua duração e novamente mandamos persisti-lo na base de dados pelo método `store()`.

Por fim, finalizamos a transação e exibimos a mensagem de sucesso na tela.

model_update.php

```
<?php
// inserir função __autoload() para carregar as classes...

/*
 * classe AlunoRecord, filha de TRecord
 * persiste um Aluno no banco de dados
 */
class AlunoRecord extends TRecord { }

/*
 * classe CursoRecord, filha de TRecord
 * persiste um Curso no banco de dados
 */
class CursoRecord extends TRecord { }

// altera objetos no banco de dados
try
{
    // inicia transação com o banco 'pg_livro'
    TTransaction::open('pg_livro');
    // define o arquivo para LOG
    TTransaction::setLogger(new TLoggerTXT('/tmp/log3.txt'));

    TTransaction::log("** obtendo o aluno 1");
    // instancia registro de Aluno
    $record = new AlunoRecord;
    // obtém o Aluno de ID 1
    $aluno = $record->load(1);
    if ($aluno) // verifica se ele existe
    {
        // altera o telefone
        $aluno->telefone = '(51) 1111-3333';
        TTransaction::log("** persistindo o aluno 1");
    }
}
```

```

    // armazena o objeto
    $aluno->store();
}

TTransaction::log("** obtendo o curso 1");
// instancia registro de Curso
$record = new CursoRecord;
// obtém o Curso de ID 1
$curso= $record->load(1);
if ($curso) // verifica se ele existe
{
    // altera a duração
    $curso->duracao = 28;
    TTransaction::log("** persistindo o curso 1");
    // armazena o objeto
    $curso->store();
}

// finaliza a transação
TTransaction::close();
// exibe mensagem de sucesso
echo "Registros alterados com sucesso<br>\n";
}
catch (Exception $e) // em caso de exceção
{
    // exibe a mensagem gerada pela exceção
    echo '<b>Erro</b>' . $e->getMessage();
    // desfaz todas alterações no banco de dados
    TTransaction::rollback();
}
?>

```

Resultado:

Registros alterados com sucesso

/tmp/log3.txt

```

2007-06-05 15:10:13 :: ** obtendo o aluno 1
2007-06-05 15:10:13 :: SELECT * FROM aluno WHERE (id = 1)
2007-06-05 15:10:13 :: ** persistindo o aluno 1
2007-06-05 15:10:13 :: SELECT * FROM aluno WHERE (id = 1)
2007-06-05 15:10:13 :: UPDATE aluno SET nome = 'Daline Dall Oglia',
                           endereco = 'Rua da Conceição', telefone = '(51) 1111-3333',
                           cidade = 'Cruzeiro do Sul' WHERE (id = 1)
2007-06-05 15:10:13 :: ** obtendo o curso 1
2007-06-05 15:10:13 :: SELECT * FROM curso WHERE (id = 1)

```

```
2007-06-05 15:10:13 :: ** persistindo o curso 1
2007-06-05 15:10:13 :: SELECT * FROM curso WHERE (id = 1)
2007-06-05 15:10:13 :: UPDATE curso SET descricao = 'Orientação a Objetos com PHP',
                           duracao = 28 WHERE (id = 1)
```

4.5.6 Clonar objeto

Neste programa demonstraremos como criar, por meio da clonagem, novos objetos baseados em outros já existentes. Para isso, primeiro instanciamos um objeto `AlunoRecord` (`$fabio`), definindo algumas de suas propriedades. Em seguida, executamos a operação `clone` sobre este objeto, o que resultará em um novo objeto com as mesmas características.

Em seguida, iniciamos o tratamento de exceções e a transação com o banco de dados para persistir esses objetos. A persistência é realizada pela chamada simples do método `store()`. Por fim, finalizamos a transação.

model_clone.php

```
<?php
// inserir função __autoload() para carregar as classes...

/*
 * classe AlunoRecord, filha de TRecord
 * persiste um Aluno no banco de dados
 */
class AlunoRecord extends TRecord { }
/*
 * classe CursoRecord, filha de TRecord
 * persiste um Curso no banco de dados
 */
class CursoRecord extends TRecord { }

// instancia objeto Aluno
$fabio = new AlunoRecord;
// define algumas propriedades
$fabio->nome      = 'Fábio Locatelli';
$fabio->endereco = 'Rua Merlin';
$fabio->telefone = '(51) 2222-1111';
$fabio->cidade   = 'Lajeado';

// clona o objeto $fabio
$julia = clone $fabio;
// altera algumas propriedades
$julia->nome      = 'Júlia Haubert';
$julia->telefone = '(51) 2222-2222';
```

```

try
{
    // inicia transação com o banco 'pg_livro'
    TTransaction::open('pg_livro');
    // define o arquivo para LOG
    TTransaction::setLogger(new TLoggerTXT('/tmp/log4.txt'));

    // armazena o objeto $fabio
    TTransaction::log("** persistindo o curso \$fabio");
    $fabio->store();
    // armazena o objeto $julia
    TTransaction::log("** persistindo o curso \$julia");
    $julia->store();

    // finaliza a transação
    TTransaction::close();

    echo "clonagem realizada com sucesso <br>\n";
}
catch (Exception $e) // em caso de exceção
{
    // exibe a mensagem gerada pela exceção
    echo '<b>Erro</b>' . $e->getMessage();
    // desfaz todas alterações no banco de dados
    TTransaction::rollback();
}
?>

```

Resultado:

clonagem realizada com sucesso

/tmp/log4.txt

```

2007-06-05 15:13:29 :: ** persistindo o curso $fabio
2007-06-05 15:13:29 :: SELECT max(ID) as ID FROM aluno
2007-06-05 15:13:29 :: INSERT INTO aluno (nome, endereco, telefone, cidade, id)
                         values ('Fábio Locatelli', 'Rua Merlin', '(51) 2222-1111', 'Lajeado', 3)
2007-06-05 15:13:29 :: ** persistindo o curso $julia
2007-06-05 15:13:29 :: SELECT max(ID) as ID FROM aluno
2007-06-05 15:13:29 :: INSERT INTO aluno (nome, endereco, telefone, cidade, id)
                         values ('Júlia Haubert', 'Rua Merlin', '(51) 2222-2222', 'Lajeado', 4)

```

4.5.7 Excluir objeto

Neste programa, iremos excluir alguns objetos da base de dados. Assim como nos programas anteriores, todas as operações estão contidas dentro de uma transação, que, por sua vez, é controlada por um try/catch (tratamento de exceções).

Existem duas formas possíveis de se excluir um objeto: na primeira, já instanciamos o objeto passando o seu ID, ou seja, ele é carregado na memória, e, em seguida, executamos o método `delete()`, que irá excluir este objeto; na segunda, instanciamos um objeto `AlunoRecord` vazio para posteriormente executar o método `delete()` passando o ID do objeto como parâmetro. Neste segundo caso é executada somente a instrução de `DELETE` na base de dados, ao passo que, na primeira forma, antes de excluído, o objeto é carregado na memória (`SELECT`).

Durante as operações, escrevemos algumas mensagens no arquivo de log para contextualizarmos melhor a ordem de execução dos comandos.

model_delete.php

```
<?php
// inserir função __autoload() para carregar as classes...

/*
 * classe AlunoRecord, filha de TRecord
 * persiste um Aluno no banco de dados
 */
class AlunoRecord extends TRecord { }
/*
 * classe CursoRecord, filha de TRecord
 * persiste um Curso no banco de dados
 */
class CursoRecord extends TRecord { }

// exclui objetos da base de dados
try
{
    // inicia transação com o banco 'pg_livro'
    TTransaction::open('pg_livro');
    // define o arquivo para LOG
    TTransaction::setLogger(new TLoggerTXT('/tmp/log5.txt'));

    // armazena esta frase no arquivo de LOG
    TTransaction::log("## Apagando da primeira forma");

    // carrega o objeto
    $aluno = new AlunoRecord(1);
```

```
// delete o objeto
$aluno->delete();

// armazena esta frase no arquivo de LOG
TTransaction::log("** Apagando da segunda forma");
// instancia o modelo
$modelo= new AlunoRecord;
// delete o objeto
$modelo->delete(2);

// finaliza a transação
TTransaction::close();

echo "Exclusão realizada com sucesso <br>\n";
}

catch (Exception $e)      // em caso de exceção
{
    // exibe a mensagem gerada pela exceção
    echo '<b>Erro</b>' . $e->getMessage();
    // desfaz todas alterações no banco de dados
    TTransaction::rollback();
}
?>
```

Resultado:

Exclusão realizada com sucesso

/tmp/log5.txt

```
2007-06-05 15:15:31 :: ** Apagando da primeira forma
2007-06-05 15:15:31 :: SELECT * FROM aluno WHERE (id = 1)
2007-06-05 15:15:31 :: DELETE FROM aluno WHERE (id = 1)
2007-06-05 15:15:31 :: ** Apagando da segunda forma
2007-06-05 15:15:31 :: DELETE FROM aluno WHERE (id = 2)
```

4.6 Manipulando coleções

Uma coleção representa um conjunto de objetos. Cada objeto possui identidade própria e comportamento, representado por suas propriedades e por seus métodos. No PHP, a melhor forma de representarmos um conjunto de objetos é por meio de arrays, estruturas flexíveis capazes de comportar vários tipos de dados. Arrays são mapas que procuram relacionar determinados valores a chaves. Como chaves de acesso, os arrays aceitam variáveis do tipo integer ou string. Como valor de cada posição, os arrays permitem uma grande variedade de campos como integer, float, string, boolean e até mesmo um outro array, construindo uma estrutura de matriz. A grande vantagem da

utilização de arrays está em sua flexibilidade, que permite adicionarmos e removermos elementos de qualquer posição do array de forma dinâmica em qualquer momento. Como os arrays podem conter também objetos, iremos utilizá-los, para representar coleções ou conjuntos de objetos.

Observação: para os próximos exemplos, iremos inserir um maior volume de dados em nossa base. Para isso, juntamente com os exemplos deste capítulo, você terá o arquivo `criabanco.sql`, responsável por inserir os registros de exemplo.

4.6.1 Repository

Sistemas com um modelo de negócios complexo geralmente requerem a elaboração de consultas SQL complexas em um dado momento para manipular seus objetos. Nestes casos, torna-se necessário, na maioria das vezes, escrever métodos responsáveis por manipular objetos baseados em diferentes critérios de seleção de dados. A maneira mais simples de se atingir este objetivo, geralmente, é por meio da escrita de um método para cada critério de seleção, como demonstrado no esboço a seguir:

```
<?php  
function listarPessoasPorNome($nome)  
{ ... }  
  
function listarPessoasPorCidade($cidade)  
{ ... }  
  
function listarPessoasPorIdade($idade)  
{ ... }  
?>
```

Quando adotamos essa abordagem, temos de escrever nossas consultas SQL à mão e, pior do que isto, temos de criar um novo método sempre que desejamos retornar objetos sob um diferente critério de seleção, o que acaba por dificultar a manutenção do código. Sendo assim, torna-se importante adicionar uma camada na aplicação que permita manipular coleções de objetos de forma flexível. Esta é justamente o contexto descrito pelo pattern conhecido por Repository.

Um Repository, ou repositório, é uma camada na aplicação que trata de mediar a comunicação entre os objetos de negócio e o banco de dados, atuando como um gerenciador de coleções de objetos. A classe que implementar um Repository deve aceitar especificações como as estudadas no capítulo anterior (critérios), que permitam selecionar um determinado grupo de objetos de forma flexível. Os objetos devem ser selecionados, excluídos e retornados a partir de uma classe Repository, por meio das especificações de critérios. Dessa forma, passamos um pouco da responsabilidade que antes seria da classe para o código que fará uso do Repository.

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.

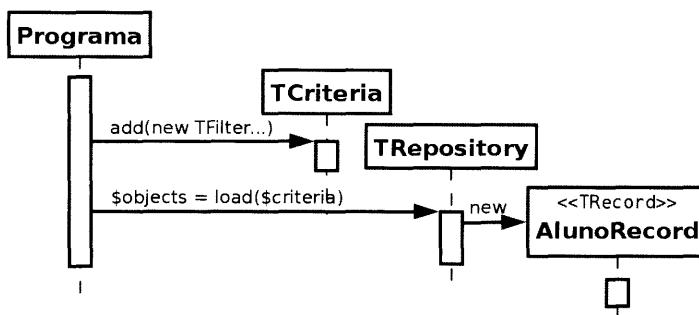


Figura 4.23 – Repository Pattern.

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
  
```

```
/* método __construct()
 * instancia um Repositório de objetos
 * @param $class = Classe dos Objetos
 */
function __construct($class)
{
    $this->class = $class;
}
```

O método `load()` será responsável por carregar na memória uma coleção de objetos. Ele receberá um critério de seleção e o utilizará em conjunto com um `TSqlSelect`. Em seguida, esse método irá detectar se existe alguma transação aberta pelo método `ITransaction::get()`. Caso exista transação aberta, ele registrará a instrução SQL no log e enviará a consulta para o banco de dados por meio do método `Query()`. Caso a consulta gere resultados, estes serão percorridos pelo método `fetchObject()` que irá instanciar objetos baseados na classe passada como parâmetro. Depois disso, o resultado (um array de objetos) é retornado.

```
/*
 * método load()
 * Recuperar um conjunto de objetos (collection) da base de dados
 * através de um critério de seleção, e instanciá-los em memória
 * @param $criteria = objeto do tipo TCriteria
 */
function load(TCriteria $criteria)
{
    // instancia a instrução de SELECT
    $sql = new TSqlSelect;
    $sql->addColumn('*');
    $sql->setEntity($this->class);
    // atribui o critério passado como parâmetro
    $sql->setCriteria($criteria);

    // obtém transação ativa
    if ($conn = ITransaction::get())
    {
        // registra mensagem de log
        ITransaction::log($sql->getInstruction());

        // executa a consulta no banco de dados
        $result = $conn->Query($sql->getInstruction());

        if ($result)
        {
            // percorre os resultados da consulta, retornando um objeto
            while ($row = $result->fetchObject($this->class . 'Record'))
            {

```

```

        // armazena no array $results;
        $results[] = $row;
    }
}
return $results;
}
else
{
    // se não tiver transação, retorna uma exceção
    throw new Exception('Não há transação ativa!!');
}
}

```

O método `delete()`, da mesma forma que o `load()`, irá receber um critério como parâmetro. Com base nesse critério, o programador poderá especificar quais registros deseja excluir da base de dados. A exclusão é definida pela classe `TSqlDelete`, que formará internamente a instrução de `DELETE` e que posteriormente será executada pelo método `exec()` da conexão ativa.

```

/*
 * método delete()
 * Excluir um conjunto de objetos (collection) da base de dados
 * através de um critério de seleção.
 * @param $criteria = objeto do tipo TCriteria
 */
function delete(TCriteria $criteria)
{
    // instancia instrução de DELETE
    $sql = new TSqlDelete;
    $sql->setEntity($this->class);
    // atribui o critério passado como parâmetro
    $sql->setCriteria($criteria);

    // obtém transação ativa
    if ($conn = TTransaction::get())
    {
        // registra mensagem de log
        TTransaction::log($sql->getInstruction());
        // executa instrução de DELETE
        $result = $conn->exec($sql->getInstruction());
        return $result;
    }
    else
    {
        // se não tiver transação, retorna uma exceção
        throw new Exception('Não há transação ativa!!');
    }
}

```

O método `count()` irá contar quantos objetos satisfazem um dado critério. Para isso, o critério é recebido como parâmetro do método. Para realizar a contagem, utilizamos a classe `TSqlSelect`, que irá retornar uma coluna `count(*)`. Em seguida, detectamos a transação ativa, realizamos a consulta ao banco de dados e retornamos o número encontrado de registros.

```
/*
 * método count()
 * Retorna a quantidade de objetos da base de dados
 * que satisfazem um determinado critério de seleção.
 * @param $criteria = objeto do tipo TCriteria
 */
function count(TCriteria $criteria)
{
    // instancia instrução de SELECT
    $sql = new TSqlSelect;
    $sql->addColumn('count(*)');
    $sql->setEntity($this->class);
    // atribui o critério passado como parâmetro
    $sql->setCriteria($criteria);

    // obtém transação ativa
    if ($conn = TTransaction::get())
    {
        // registra mensagem de log
        TTransaction::log($sql->getInstruction());
        // executa instrução de SELECT
        $result= $conn->Query($sql->getInstruction());
        if ($result)
        {
            $row = $result->fetch();
        }
        // retorna o resultado
        return $row[0];
    }
    else
    {
        // se não tiver transação, retorna uma exceção
        throw new Exception('Não há transação ativa!!');
    }
}
```

?>

Nos próximos exemplos demonstraremos a utilização de métodos da classe `TRepository`, como `load()` e `delete()`, os quais lidam com critérios de seleção de dados

e manipulam conjuntos de objetos, e não objetos individuais. Também estudaremos a utilização do método `count()`, que retorna a quantidade de objetos que satisfazem um dado critério de seleção.

Observação: para os próximos exemplos, precisaremos inserir mais dados nas nossas tabelas. Para isso, utilizaremos o script `criabanco.sql`, que acompanha os exemplos deste livro, responsável por inserir dados de alunos, inscrições, turmas e cursos.

4.6.2 Obter coleção de objetos

Neste primeiro exemplo de programa que manipula coleções de objetos, demonstraremos como obter um conjunto de objetos do banco de dados. Para isso, faremos uso do método `load()` da classe `TRepository`, que, por sua vez, retorna todos os objetos que satisfazem um determinado critério de seleção de dados. Portanto, precisamos utilizar a classe `TCriteria`, criada no capítulo anterior. Utilizaremos esta classe para expressar os filtros de seleção de dados. Note que o programa, assim como os anteriores, possui tratamento de exceções e irá gravar as instruções SQL processadas no arquivo `/tmp/log6.txt`.

Na primeira parte do programa, estamos obtendo todos os objetos armazenados na tabela `turma` em andamento (ou seja, `encerrado = FALSE`), do turno da tarde (`turno='T'`). O método `load()` retornará um array de objetos caso exista algum objeto na base de dados que satisfaça este critério. Em seguida, estamos percorrendo este array de objetos por meio de um `foreach`, exibindo as suas propriedades (`id`, `dia_semana`, `sala`, `turno` e `professor`) na tela.

Na segunda parte do programa, obtemos todos os alunos aprovados da turma “1”, ou seja, todos que alcançaram nota superior a “7” e freqüência superior a “75” e cuja inscrição não fora cancelada. Caso alguma inscrição seja retornada, estaremos percorrendo-as por meio de um `foreach` e exibindo algumas de suas propriedades (`id` e `ref_aluno`). Note que não temos as informações do aluno nesta tabela; para isso, instanciamos um objeto `AlunoRecord`, passando o código do aluno (`$inscricao->ref_aluno`), que é chave estrangeira na tabela de inscrições. Fazendo isto, temos acesso a todas as propriedades do objeto `aluno`, como `nome` e `endereco`, podendo exibi-las na tela.

collection_get.php

```
<?php  
// inserir função __autoload() para carregar as classes...  
  
/* cria as classes Active Record  
para manipular os registros das tabelas correspondentes */
```

```
class AlunoRecord      extends TRecord {}
class TurmaRecord      extends TRecord {}
class InscricaoRecord extends TRecord {}

// obtém objetos do banco de dados
try
{
    // inicia transação com o banco 'pg_livro'
    TTransaction::open('pg_livro');
    // define o arquivo para LOG
    TTransaction::setLogger(new TLoggerTXT('/tmp/log6.txt'));

#####
# primeiro exemplo, lista todas turmas em andamento no turno Tarde #
#####

// cria um critério de seleção
$criteria = new TCriteria;
// filtra por turno e encerrada
$criteria->add(new TFilter('turno', '=', 'T'));
$criteria->add(new TFilter('encerrada', '=', FALSE));

// instancia um repositório para Turma
$repository = new TRepository('Turma');
// retorna todos objetos que satisfazem o critério
$turmas = $repository->load($criteria);
// verifica se retornou alguma turma
if ($turmas)
{
    echo "Turmas retornadas <br>\n";
    echo "===== <br>\n";
    // percorre todas turmas retornadas
    foreach ($turmas as $turma)
    {
        echo ' ID  : ' . $turma->id;
        echo ' Dia : ' . $turma->dia_semana;
        echo ' Sala : ' . $turma->sala;
        echo ' Turno: ' . $turma->turno;
        echo ' Professor: ' . $turma->professor;
        echo "<br>\n";
    }
}

#####
# segundo exemplo, lista todos aprovados da turma "1" #
#####
```

```

// instancia um critério de seleção
$criteria = new TCriteria;
$criteria->add(new TFilter('nota',      '>=' ,    7));
$criteria->add(new TFilter('frequencia','>=' , 75));
$criteria->add(new TFilter('ref_turma', '=' ,    1));
$criteria->add(new TFilter('cancelada', '=' , FALSE));

// instancia um repositório para Inscricao
$repository = new TRepository('Inscricao');
// retorna todos objetos que satisfazem o critério
$inscricoes = $repository->load($criteria);
// verifica se retornou alguma inscrição
if ($inscricoes)
{
    echo "Inscrições retornadas <br>\n";
    echo "===== <br>\n";
    // percorre todas inscrições retornadas
    foreach ($inscricoes as $inscricao)
    {
        echo ' ID     : ' . $inscricao->id;
        echo ' Aluno : ' . $inscricao->ref_aluno;

        // obtém o aluno relacionado à inscrição
        $aluno = new AlunoRecord($inscricao->ref_aluno);
        echo ' Nome : ' . $aluno->nome;
        echo ' Rua   : ' . $aluno->endereco;
        echo "<br>\n";
    }
}

// finaliza a transação
TTransaction::close();
}

catch (Exception $e)      // em caso de exceção
{
    // exibe a mensagem gerada pela exceção
    echo '<b>Erro</b>' . $e->getMessage();
    // desfaz todas alterações no banco de dados
    TTransaction::rollback();
}

```



```
2007-06-05 15:20:34 :: SELECT * FROM aluno WHERE (id = 5)
2007-06-05 15:20:34 :: SELECT * FROM aluno WHERE (id = 7)
2007-06-05 15:20:34 :: SELECT * FROM aluno WHERE (id = 8)
```

Resultado:

Turmas retornadas

```
=====
ID : 2 Dia : 2 Sala : 100 Turno: T Professor:Pablo Dall Oglia
ID : 4 Dia : 4 Sala : 100 Turno: T Professor:Pablo Dall Oglia
ID : 6 Dia : 2 Sala : 200 Turno: T Professor:Fabio Locatelli
ID : 8 Dia : 4 Sala : 200 Turno: T Professor:Fabio Locatelli
```

Inscrições retornadas

```
=====
ID : 5 Aluno : 5 Nome : HENRIQUE BARBOSA Rua : Rua Érico Veríssimo
ID : 7 Aluno : 7 Nome : LUIZ BENEDITO Rua : Rua Érico Veríssimo
ID : 8 Aluno : 8 Nome : MARIA BENJAMIM Rua : Rua Érico Veríssimo
```

4.6.3 Alterar coleção de objetos

No próximo exemplo, demonstraremos como alterar propriedades de uma coleção de objetos e como persisti-los no banco de dados. Neste programa, definiremos a nota para “8” e a frequencia para “75” de todos os alunos da turma “2”. Para isso, instanciaremos um critério de seleção de dados para retornar todas as inscrições cuja propriedade ref_turma é igual a “2” e a propriedade cancelada é FALSE. Em seguida, buscaremos esses objetos no banco de dados por meio do método `load()` da classe `TRepository`. Caso alguma inscrição seja retornada, iremos percorrê-la por meio de um `foreach`, alterando suas propriedades `nota` e `frequencia` e definindo novos valores. Ao final, persistimos a inscrição alterada no banco de dados pelo método `store()`.

collection_update.php

```
<?php
// inserir função __autoload() para carregar as classes...

/* cria as classes Active Record
   para manipular os registros das tabelas correspondentes */
class InscricaoRecord extends TRecord { }

// obtém objetos do banco de dados
try
{
    // inicia transação com o banco 'pg_livro'
```

```
TTransaction::open('pg_livro');
// define o arquivo para LOG
TTransaction::setLogger(new TLoggerTXT('/tmp/log7.txt'));
TTransaction::log("** seleciona inscrições da turma 2");

// instancia critério de seleção de dados
// seleciona todas inscrições da turma "2"
$criteria = new TCriteria;
$criteria->add(new TFilter('ref_turma', '=', 2));
$criteria->add(new TFilter('cancelada', '=', FALSE));

// instancia repositório de Inscrição
$repository = new TRepository('Inscricao');
// retorna todos objetos que satisfazem o critério
$inscricoes = $repository->load($criteria);

// verifica se retornou alguma inscrição
if ($inscricoes)
{
    TTransaction::log("** altera as inscrições");
    // percorre todas inscrições retornadas
    foreach ($inscricoes as $inscricao)
    {
        // altera algumas propriedades
        $inscricao->nota      = 8;
        $inscricao->frequencia = 75;

        // armazena o objeto no banco de dados
        $inscricao->store();
    }
}
// finaliza a transação
TTransaction::close();
}

catch (Exception $e) // em caso de exceção
{
    // exibe a mensagem gerada pela exceção
    echo '<b>Erro</b>' . $e->getMessage();
    // desfaz todas alterações no banco de dados
    TTransaction::rollback();
}
?>
```

 /tmp/log7.txt

```
2007-06-05 15:24:11 :: ** seleciona inscrições da turma 2
2007-06-05 15:24:11 :: SELECT * FROM Inscricao WHERE (ref_turma = 2 AND cancelada = FALSE)
2007-06-05 15:24:11 :: ** altera as inscrições
2007-06-05 15:24:11 :: SELECT * FROM inscricao WHERE (id = 11)
2007-06-05 15:24:11 :: UPDATE inscricao SET ref_aluno = 11, ref_turma = 2,
                                nota = 8, frequencia = 75, cancelada = FALSE,
                                concluida = FALSE WHERE (id = 11)
2007-06-05 15:24:11 :: SELECT * FROM inscricao WHERE (id = 12)
2007-06-05 15:24:11 :: UPDATE inscricao SET ref_aluno = 12, ref_turma = 2,
                                nota = 8, frequencia = 75, cancelada = FALSE,
                                concluida = FALSE WHERE (id = 12)
....
```

4.6.4 Contar objetos

No próximo exemplo, demonstraremos como contar objetos. No programa a seguir, realizamos duas contagens. Na primeira delas, estamos contando todos os alunos da cidade “Porto Alegre”; para isso, criamos um critério de seleção no qual “cidade=Porto Alegre”. Ao final exibimos o total de alunos na tela.

Na segunda contagem, estamos contando todas as turmas com aula na sala “100” no turno da tarde OU com aula na sala “200” no turno da manhã. Como este critério de seleção é mais complexo, precisamos decompô-lo em dois outros critérios de seleção menores. O primeiro irá selecionar as turmas com aula na sala “100” no turno da tarde; o segundo irá selecionar as turmas com aula na sala “200” no turno da manhã, e o critério de seleção principal irá juntar esses dois por meio do operador “ou”. Ao final, exibimos a quantidade de turmas.

 collection_count.php

```
<?php
// inserir função __autoload() para carregar as classes...

/* cria as classes Active Record
   para manipular os registros das tabelas correspondentes */
class AlunoRecord extends TRecord { }
class TurmaRecord extends TRecord { }

// conta objetos do banco de dados
try
{
    // inicia transação com o banco 'pg_livro'
    TTransaction::open('pg_livro');
```

```
// define o arquivo para LOG
TTransaction::setLogger(new TLoggerTXT('/tmp/log8.txt'));

#####
# primeiro exemplo, conta todos alunos de Porto Alegre #
#####
TTransaction::log("** Conta Alunos de Porto Alegre");

// instancia um critério de seleção
$criteria = new TCriteria;
$criteria->add(new TFilter('cidade', '=', 'Porto Alegre'));

// instancia um reposotório de Alunos
$repository = new TRepository('Aluno');
// obtém o total de alunos que satisfazem o critério
$count = $repository->count($criteria);

// exibe o total na tela
echo "Total de alunos de Porto Alegre: {$count} <br>\n";

#####
# segundo exemplo, Contar todas as turmas com aula na sala #
# "100" no turno da Tarde OU na "200" pelo turno da manha. #
#####
TTransaction::log("** Conta Turmas");

// instancia um critério de seleção
// sala "100" e turno "T" (tarde)
$criteria1 = new TCriteria;
$criteria1->add(new TFilter('sala', '=', '100'));
$criteria1->add(new TFilter('turno', '=', 'T'));

// instancia um critério de seleção
// sala "200" e turno "M" (manha)
$criteria2 = new TCriteria;
$criteria2->add(new TFilter('sala', '=', '200'));
$criteria2->add(new TFilter('turno', '=', 'M'));

// instancia um critério de seleção
// com OU para juntar os critérios anteriores
$criteria = new TCriteria;
$criteria->add($criteria1, TExpression::OR_OPERATOR);
$criteria->add($criteria2, TExpression::OR_OPERATOR);

// instancia um repositório de Turmas
$repository = new TRepository('Turma');
```

```
// retorna quantos objetos satisfazem o critério
$count = $repository->count($criteria);
echo "Total de turmas: {$count} <br>\n";

// finaliza a transação
TTransaction::close();
}

catch (Exception $e)      // em caso de exceção
{
    // exibe a mensagem gerada pela exceção
    echo '<b>Erro</b>' . $e->getMessage();
    // desfaz todas alterações no banco de dados
    TTransaction::rollback();
}
?>
```

/tmp/log8.txt

```
2007-06-05 15:28:43 :: ** Conta Alunos de Porto Alegre
2007-06-05 15:28:43 :: SELECT count(*) FROM Aluno WHERE (cidade = 'Porto Alegre')
2007-06-05 15:28:43 :: ** Conta Turmas
2007-06-05 15:28:43 :: SELECT count(*) FROM Turma WHERE ((sala = '100' AND turno = 'T')
OR (sala = '200' AND turno = 'M'))
```

Resultado:

```
Total de alunos de Porto Alegre: 10
Total de turmas: 4
```

4.6.5 Excluir coleção de objetos

Neste exemplo, demonstraremos como excluir uma coleção de objetos. Para isto existem basicamente duas formas, demonstradas no programa a seguir. No primeiro exemplo, excluiremos todas as turmas com aula no turno da tarde. Para tanto, criamos um critério de seleção “turno = ‘T’” e obtemos todos os objetos do modelo de turmas que satisfazem este critério. Percorremos este resultado e, para cada objeto retornado, executamos o método `delete()`. Esta forma de deletar um conjunto de objetos é onerosa porque executamos uma instrução `DELETE` para cada objeto da coleção. No exemplo seguinte, excluímos todas as inscrições existentes para o aluno “1”. Em vez de obter a coleção de objetos antes de excluí-la, estamos executando diretamente o método `delete()`, passando como parâmetro o critério utilizado para selecionar os registros que serão excluídos. Desta segunda forma, executamos a operação `DELETE` uma única vez no banco de dados, ganhando em velocidade.

 collection_delete.php

```
<?php
// inserir função __autoload() para carregar as classes...

/* cria as classes Active Record
   para manipular os registros das tabelas correspondentes */
class TurmaRecord extends TRecord { }
class InscricaoRecord extends TRecord { }

// deleta objetos do banco de dados
try
{
    // inicia transação com o banco 'pg_livro'
    TTransaction::open('pg_livro');
    // define o arquivo para LOG
    TTransaction::setLogger(new TLoggerTXT('/tmp/log9.txt'));

    #####
    # primeiro exemplo, exclui todas turmas da tarde #
    #####
    TTransaction::log("** exclui as turmas da tarde");

    // instancia um critério de seleção turno = 'T'
    $criteria = new TCriteria;
    $criteria->add(new TFilter('turno', '=', 'T'));

    // instancia repositório de Turmas
    $repository = new TRepository('Turma');
    // retorna todos objetos que satisfazem o critério
    $turmas = $repository->load($criteria);
    // verifica se retornou alguma turma
    if ($turmas)
    {
        // percorre todas turmas retornadas
        foreach ($turmas as $turma)
        {
            // exclui a turma
            $turma->delete();
        }
    }

    #####
    # segundo exemplo, exclui as inscrições do aluno "1" #
    #####
    TTransaction::log("** exclui as inscrições do aluno '1'");
}
```

```
// instancia critério de seleção de dados ref_aluno ='1'  
$criteria = new TCriteria;  
$criteria->add(new TFilter('ref_aluno', '=', 1));  
  
// instancia repositório de Inscrição  
$repository = new TRRepository('Inscricao');  
// exclui todos objetos que satisfaçam este critério de seleção  
$repository->delete($criteria);  
  
echo "registros excluídos com sucesso <br>\n";  
// finaliza a transação  
TTransaction::close();  
}  
catch (Exception $e) // em caso de exceção  
{  
    // exibe a mensagem gerada pela exceção  
    echo '<b>Erro</b>' . $e->getMessage();  
    // desfaz todas alterações no banco de dados  
    TTransaction::rollback();  
}  
?>
```

/tmp/log9.txt

```
2007-06-05 15:32:09 :: ** exclui as turmas da tarde  
2007-06-05 15:32:09 :: SELECT * FROM Turma WHERE (turno = 'T')  
2007-06-05 15:32:09 :: DELETE FROM turma WHERE (id = 2)  
2007-06-05 15:32:09 :: DELETE FROM turma WHERE (id = 4)  
2007-06-05 15:32:09 :: DELETE FROM turma WHERE (id = 6)  
2007-06-05 15:32:09 :: DELETE FROM turma WHERE (id = 8)  
2007-06-05 15:32:09 :: ** exclui as inscrições do aluno '1'  
2007-06-05 15:32:09 :: DELETE FROM Inscricao WHERE (ref_aluno = 1)
```

Resultado:

registros excluídos com sucesso

4.7 Aspectos avançados

4.7.1 Encapsulamento

Como vimos no Capítulo 2, o encapsulamento trata de proteger o acesso às propriedades internas de um objeto. Até o momento não nos preocupamos com isso, mas essa questão é de fundamental importância, principalmente em um ambiente em equipe onde uma pessoa é responsável por criar uma classe e outras por utilizarem-na. Se a

classe prover encapsulamento, ela protegerá as suas propriedades contra o mal uso de terceiros. Assim teremos um sistema mais confiável e íntegro.

No exemplo a seguir, demonstraremos uma forma de prover encapsulamento quando trabalhamos com um Active Record, como é a nossa classe `TRecord`. Sempre que tentarmos atribuir um valor a uma propriedade de um objeto derivado dessa classe, o método `__set()` é executado, interceptando esta atribuição. Caso exista algum método na classe nomeado por `set_<propriedade>`, ele será executado no lugar da simples atribuição. Dessa forma, se estivermos atribuindo um valor para a propriedade `nome`, o método `__set()` irá verificar se existe algum método nomeado por `set_nome()` na classe e irá executá-lo. Caso contrário, o valor será atribuído à propriedade diretamente.

Com o exemplo a seguir, procuramos demonstrar como podemos proteger algumas propriedades internas de um objeto. Para isso, criamos a classe `TurmaRecord`, como vimos fazendo até o momento. Além disso, criamos dois novos métodos: `set_dia_semana()`, que será executado automaticamente quando for atribuído um valor para a propriedade `dia_semana`, e o método `set_turno()`, que será executado automaticamente quando tentarmos atribuir algum valor para a propriedade `turno`.

Esses dois métodos fazem alguns testes de consistência para evitar que possamos atribuir um valor inválido para as propriedades. O método `set_dia_semana()` verificará se o valor é numérico e está entre 1 e 7. O método `set_turno()` verificará se o valor está compreendido entre “M”, “T” ou “N”. Caso o valor passe pelos testes, é atribuído ao array de dados do objeto. Para verificar seu funcionamento, criaremos duas turmas: a primeira com os dados consistentes, e a segunda com `dia_semana` e `turno` inválidos. No segundo caso, essas propriedades não serão atribuídas ao objeto e, consequentemente, não serão persistidas no banco de dados. Além disso, poderíamos lançar uma exceção (`throw`), dentre outros.

encapsulamento.php

```
<?php
// inserir função __autoload() para carregar as classes...
/*
 * classe TurmaRecord, filha de TRecord
 * persiste um Aluno no banco de dados
 */
class TurmaRecord extends TRecord
{
    /*
     * método set_dia_semana()
     * executado sempre que há uma atribuição para "dia_semana"
     * @param $valor = valor atribuído
    */
}
```

```
function set_dia_semana($valor)
{
    // verifica se o dia da semana está entre 1 e 7 e é número
    if (is_int($valor) and ($valor>=1) and ($valor <=7))
    {
        // atribui o valor à propriedade
        $this->data['dia_semana'] = $valor;
    }
    else
    {
        // exibe mensagem de erro
        echo "Tentou atribuir '{$valor}' em dia_semana <br>\n";
    }
}
/*
 * método set_turno()
 * executado sempre que há uma atribuição para "turno"
 * @param $valor = valor atribuído
 */
function set_turno($valor)
{
    // verifica se o valor é 'M', 'T' ou 'N'
    if (($valor=='M') or ($valor == 'T') or ($valor == 'N'))
    {
        // atribui o valor à propriedade
        $this->data['turno'] = $valor;
    }
    else
    {
        // exibe mensagem de erro
        echo "Tentou atribuir '{$valor}' em turno <br>\n";
    }
}
// insere novos objetos no banco de dados
try
{
    // inicia transação com o banco 'pg_livro'
    TTransaction::open('pg_livro');
    // define o arquivo para LOG
    TTransaction::setLogger(new TLoggerTXT('/tmp/log10.txt'));

    // armazena esta frase no arquivo de LOG
    TTransaction::log("## inserindo turma 1");

    // instancia um novo objeto Turma
    $turma = new TurmaRecord;
```

```

$turma->dia_semana = 1;
$turma->turno      = 'M';
$turma->professor  = 'Carlo Bellini';
$turma->sala        = '100';
$turma->data_inicio = '2002-09-01';
$turma->encerrada   = FALSE;
$turma->ref_curso  = 2;

$turma->store(); // armazena o objeto

// armazena esta frase no arquivo de LOG
TTransaction::log("** inserindo turma 1");
$turma= new TurmaRecord;
$turma->dia_semana = 'Segunda';
$turma->turno      = 'Manhã';
$turma->professor  = 'Sérgio Crespo';
$turma->sala        = '200';
$turma->data_inicio = '2004-09-01';
$turma->encerrada   = FALSE;
$turma->ref_curso  = 3;
$turma->store(); // armazena o objeto
// finaliza a transação
TTransaction::close();

echo "Registros inseridos com Sucesso<br>\n";
}

catch (Exception $e) // em caso de exceção
{
    // exibe a mensagem gerada pela exceção
    echo '<b>Erro</b>' . $e->getMessage();
    // desfaz todas alterações no banco de dados
    TTransaction::rollback();
}
?>
```

/tmp/log10.txt

```

2007-06-05 15:35:32 :: ** inserindo turma 1
2007-06-05 15:35:32 :: SELECT max(ID) as ID FROM turma
2007-06-05 15:35:32 :: INSERT INTO turma (dia_semana, turno, professor, sala,
                                         data_inicio, encerrada, ref_curso, id)
                                         values (1, 'M', 'Carlo Bellini', '100', '2002-09-01', FALSE, 2, 8)
2007-06-05 15:35:32 :: ** inserindo turma 2
2007-06-05 15:35:32 :: SELECT max(ID) as ID FROM turma
2007-06-05 15:35:32 :: INSERT INTO turma (professor, sala, data_inicio, encerrada,
                                         ref_curso, id)
                                         values ('Sérgio Crespo', '200', '2004-09-01', FALSE, 3, 9)
```

 **Resultado:**

```
Tentou atribuir 'Segunda' em dia_semana
Tentou atribuir 'Manh  ' em turno
Registros inseridos com Sucesso
```

4.7.2 Lazy Initialization

Como vimos anteriormente, precisamos, com freq  u  cia, navegar por entre os relacionamentos dos objetos, o que faz necess  rio que esses objetos relacionados estejam dispon  veis (instanciados). O ideal seria que os objetos relacionados fossem instanciados somente quando necess  rios para a aplic  o  . Vimos que este comportamento  poss  vel gra  as ao m  todo `__get()` que intercepta a obten  o de propriedades de um objeto, permitindo que, durante essa oper  o  , poss  mos instanciar objetos.

Quando declaramos um m  todo em uma superclasse, ele  v  lido para todas as suas classes-filha. Dessa forma, a classe `TRecord` possui o m  todo `__get()` que ser   executado sempre que tentarmos obter uma de suas propriedades. Tal m  todo, que recebe o nome da propriedade que est   sendo requerida, est   programado para verificar a exist  ncia de um m  todo nomeado por `get_<propriedade>` na classe atual. Se tentarmos obter a propriedade `cidade`, automaticamente o m  todo `__get()` ir   procurar pela exist  ncia da fun  o `get_cidade()` na mesma classe. Caso esta fun  o exista, ela ser   executada; caso contr  rio, simplesmente ser   retornado o valor da propriedade.

Este recurso pode ser executado em v  rios casos em que desejamos retornar objetos relacionados (associa  es, composi  es e agreg  es). No exemplo a seguir, a partir de uma inscri  o, retornaremos o objeto `AlunoRecord` relacionado, simplesmente acessando a propriedade `aluno` da `Inscricao`. Esta propriedade n  o existe na realidade; podemos cham  -la de pseudopropriedade, mas o m  todo `__get()` da classe `TRecord` tentar   executar o m  todo `get_aluno()` antes de retornar o valor da propriedade. Assim, escreveremos nesse m  todo o c  digo necess  rio para buscar o objeto `aluno` relacionado com a inscri  o.

No segundo exemplo, estamos retornando uma cole  o de objetos. Um aluno tem n inscri  es relacionadas a si. Esse relacionamento est   presente pela chave estrangeira `ref_aluno` na tabela `Inscricao`. O que faremos  criar a pseudopropriedade `inscricoes`. Tal propriedade n  o existe, mas o m  todo `__get()` tentar   executar a fun  o `get_inscricoes()`; caso essa fun  o exista, ela ser   executada, e  justamente dentro desta fun  o que colocaremos o c  digo necess  rio para retornar o conjunto (collection) de objetos do tipo `Inscricao` de um mesmo aluno. A cria  o das pseudopropriedades implementa o conceito de “lazy initialization”, recurso que permite trabalhar com objetos relacionados de forma transparente  aplic  o  .

No diagrama a seguir procuramos demonstrar as duas relações existentes entre as classes `AlunoRecord` e `InscricaoRecord`.

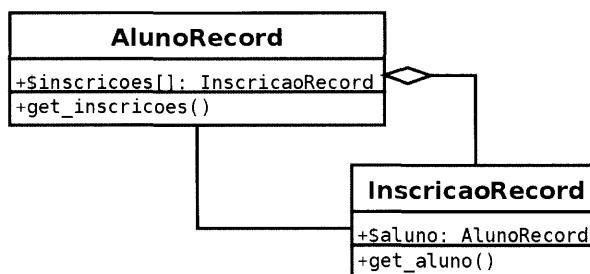


Figura 4.24 – Classes `AlunoRecord` e `InscricaoRecord`.

lazy.php

```

<?php
// inserir função __autoload() para carregar as classes...
/*
 * classe InscricaoRecord, filha de TRecord
 * persiste uma Inscricao no banco de dados
 */
class InscricaoRecord extends TRecord
{
    /*
     * método get_aluno()
     * executado sempre se for acessada a propriedade "aluno"
     */
    function get_aluno()
    {
        // instancia AlunoRecord, carrega
        // na memória o aluno de código $this->ref_aluno
        $aluno = new AlunoRecord($this->ref_aluno);

        // retorna o objeto instanciado
        return $aluno;
    }
}
/*
 * classe AlunoRecord, filha de TRecord
 * persiste um Aluno no banco de dados
 */
class AlunoRecord extends TRecord
{
    /*
     * método get_inscricoes()
     */
}
  
```

```
* executado sempre se for acessada a propriedade "inscricoes"
*/
function get_inscricoes()
{
    // cria um critério de seleção
    $criteria = new TCriteria;
    // filtra por código do aluno
    $criteria->add(new TFilter('ref_aluno', '=', $this->id));

    // instancia repositório de Inscrições
    $repository = new TRepository('Inscricao');
    // retorna todas inscrições que satisfazem o critério
    return $repository->load($criteria);
}

try
{
    // inicia transação com o banco 'pg_livro'
    TTransaction::open('pg_livro');
    // define o arquivo para LOG
    TTransaction::setLogger(new TLoggerTXT('/tmp/log11.txt'));

    // armazena esta frase no arquivo de LOG
    TTransaction::log("** obtendo o aluno de uma inscrição");

    // instancia a Inscrição cujo ID é "2"
    $inscricao = new InscricaoRecord(2);
    // exibe os dados relacionados de aluno (associação)
    echo "Dados da inscrição<br>\n";
    echo "======<br>\n";
    echo 'Nome : ' . $inscricao->aluno->nome . "<br>\n";
    echo 'Endereço : ' . $inscricao->aluno->endereco . "<br>\n";
    echo 'Cidade : ' . $inscricao->aluno->cidade . "<br>\n";

    // armazena esta frase no arquivo de LOG
    TTransaction::log("** obtendo as inscrições de um aluno");

    // instancia o Aluno cujo ID é "2"
    $aluno = new AlunoRecord(2);

    echo "<br>\n";
    echo "Inscrições do Aluno<br>\n";
    echo "======<br>\n";
    // exibe os dados relacionados de inscrições (agregação)
    foreach ($aluno->inscricoes as $inscricao)
    {
```

```

        echo ' ID    : ' . $inscricao->id;
        echo ' Turma : ' . $inscricao->ref_turma;
        echo ' Nota  : ' . $inscricao->nota;
        echo ' Freq.  : ' . $inscricao->frequencia;
        echo "<br>\n";
    }

    // finaliza a transação
    TTransaction::close();
}
catch (Exception $e) // em caso de exceção
{
    // exibe a mensagem gerada pela exceção
    echo '<b>Erro</b>' . $e->getMessage();
    // desfaz todas alterações no banco de dados
    TTransaction::rollback();
}
?>

```

/tmp/log11.txt

```

2007-06-05 15:41:05 :: ** obtendo o aluno de uma inscrição
2007-06-05 15:41:05 :: SELECT * FROM inscricao WHERE (id = 2)
2007-06-05 15:41:05 :: SELECT * FROM aluno WHERE (id = 2)
2007-06-05 15:41:05 :: SELECT * FROM aluno WHERE (id = 2)
2007-06-05 15:41:05 :: SELECT * FROM aluno WHERE (id = 2)
2007-06-05 15:41:05 :: ** obtendo as inscrições de um aluno
2007-06-05 15:41:05 :: SELECT * FROM aluno WHERE (id = 2)
2007-06-05 15:41:05 :: SELECT * FROM Inscricao WHERE (ref_aluno = 2)

```

Resultado:

Dados da inscrição

=====

Nome : ELCIO ANCHIETA
 Endereço : Rua Érico Veríssimo
 Cidade : Porto Alegre

Inscrições do Aluno

=====

ID : 2 Turma : 1 Nota : 0 Freq. : 80

Observação: cada vez que acessamos a propriedade `aluno` de uma `Inscricao`, é realizada uma consulta na tabela de `aluno` para obter o registro relacionado. Essas consultas poderiam ser racionalizadas caso guardássemos este objeto como uma propriedade.

4.7.3 Criar métodos de negócio

Como vimos anteriormente, um Active Record possui também métodos que implementam a lógica de negócios da aplicação, não somente o acesso aos dados da mesma. Pensando nisso, procuramos ilustrar esse conceito. É provável que, após algum tempo, você consiga identificar partes do código da aplicação que se repetem em vários locais diferentes. Nesse momento, você deve parar e refletir se este código pode ser encapsulado em alguma das classes do sistema. Para isso, é necessário ter a clara noção de responsabilidade, ou seja, qual classe é a responsável pelos dados sobre os quais estamos falando. No programa a seguir, criamos a classe `AlunoRecord` com um método `Inscrever()`, o qual ilustra um pedaço de código que poderia estar solto na aplicação. A ação de `Inscrever()` é relativa a `AlunoRecord`, uma vez que, quando fazemos uma inscrição, fazemos a inscrição de um aluno. Portanto, a classe `AlunoRecord` é o local mais adequado para colocar esse método.

Desse ponto em diante, sempre que precisarmos inscrever um aluno em uma turma, será necessário apenas chamar o método `Inscrever()` do `Aluno`, passando a `$turma` como parâmetro. O fato de termos um ponto central para executar tal operação de inscrição nos dá maior modularidade e consistência, visto que daremos preferência a executar este método sobre copiar e colar um bloco de código de um programa para outro.

business.php

```
<?php
// inserir função __autoload() para carregar as classes...
/*
 * classe AlunoRecord, filha de TRecord
 * persiste um Aluno no banco de dados
 */
class AlunoRecord extends TRecord
{
    /*
     * método Inscrever
     * cria uma inscrição para este aluno
     * @param $turma = número da turma
     */
    function Inscrever($turma)
    {
        // instancia uma inscrição
        $inscricao = new InscricaoRecord;
        // define algumas propriedades
        $inscricao->ref_aluno = $this->id;
        $inscricao->ref_turma = $turma;
```

```
// persiste a inscrição
$inscricao->store();
}

}

/*
 * classe InscricaoRecord, filha de TRecord
 * persiste um Inscricao no banco de dados
 */
class InscricaoRecord extends TRecord {}

// insere novos objetos no banco de dados
try
{
    // inicia transação com o banco 'pg_livro'
    TTransaction::open('pg_livro');
    // define o arquivo para LOG
    TTransaction::setLogger(new TLoggerTXT('/tmp/log12.txt'));

    // armazena esta frase no arquivo de LOG
    TTransaction::log("** inserindo o aluno '$carlos');

    // instancia um aluno novo
    $carlos= new AlunoRecord;
    $carlos->nome      = "Carlos Ranzi";
    $carlos->endereco = "Rua Francisco Oscar";
    $carlos->telefone = "(51) 1234-5678";
    $carlos->cidade   = "Lajeado";
    // persiste o objeto aluno
    $carlos->store();

    // armazena esta frase no arquivo de LOG
    TTransaction::log("** inscrevendo o aluno nas turmas");

    // executa o método Inscrever (na turma 1 e 2)
    $carlos->Inscrever(1);
    $carlos->Inscrever(2);

    // finaliza a transação
    TTransaction::close();
}

catch (Exception $e) // em caso de exceção
{
    // exibe a mensagem gerada pela exceção
    echo '<b>Erro</b>' . $e->getMessage();
    // desfaz todas alterações no banco de dados
    TTransaction::rollback();
}

?>
```

 /tmp/log12.txt:

```
2007-06-05 15:43:22 :: ** inserindo o aluno $carlos
2007-06-05 15:43:22 :: SELECT max(ID) as ID FROM aluno
2007-06-05 15:43:22 :: INSERT INTO aluno (nome, endereco, telefone, cidade, id)
                         values ('Carlos Ranzi', 'Rua Francisco Oscar', '(51) 1234-5678', 'Lajeado', 41)
2007-06-05 15:43:22 :: ** inscrevendo o aluno nas turmas
2007-06-05 15:43:22 :: SELECT max(ID) as ID FROM inscricao
2007-06-05 15:43:22 :: INSERT INTO inscricao (ref_aluno, ref_turma, id) values (41, 1, 41)
2007-06-05 15:43:22 :: SELECT max(ID) as ID FROM inscricao
2007-06-05 15:43:22 :: INSERT INTO inscricao (ref_aluno, ref_turma, id) values (41, 2, 42)
```

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,

podendo construir uma página HTML por meio de objetos, de modo que cada objeto representará um elemento gráfico (texto, imagem, tabela), como na Figura 5.1, na qual temos uma pequena página contendo alguns elementos, como um parágrafo, uma imagem, uma tabela e um campo de entrada de dados.



Figura 5.1 – Elementos visuais.

Para melhor organização do nosso sistema, as classes aqui desenvolvidas serão armazenadas na pasta `app.widgets`, como ilustrado na Figura 5.2. Widgets é um termo do inglês resultado da junção de Window + Gadgets, cujo significado é algo parecido com dispositivo ou apetrechos de janelas, ou seja, itens relacionados ao ambiente de janelas.

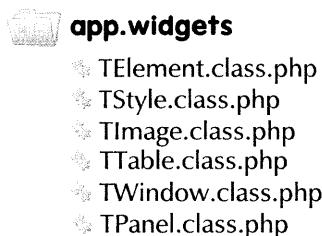


Figura 5.2 – Estrutura do diretório `app.widgets`.

5.2 Componentes

Nesta primeira seção construiremos alguns componentes que serão utilizados ao longo do livro. Inicialmente iremos construir algumas classes para abstração de HTML e folhas de estilo css para posteriormente criar classes que trabalham em um nível superior.

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>`, ``, `<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`.

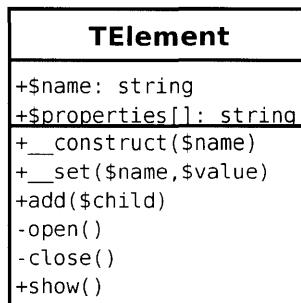


Figura 5.3 – Classe `TElement`.

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
     */
}
```

```
public function __construct($name)
{
    // define o nome do elemento
    $this->name = $name;
}

/**
 * método __set()
 * intercepta as atribuições à propriedades do objeto
 * @param $name    = nome da propriedade
 * @param $value   = valor
 */
public function __set($name, $value)
{
    // armazena os valores atribuídos
    // ao array properties
    $this->properties[$name] = $value;
}
```

O método `add()` nos permitirá adicionar conteúdo(s) à tag. Neste caso, a variável `$child` poderá ser uma string qualquer, mas também poderá ser qualquer outro objeto. Esses “filhos” serão armazenados no vetor `$children`, que será percorrido no momento da exibição da tag na tela por meio do método `show()`.

```
/**
 * método add()
 * adiciona um elemento filho
 * @param $child = objeto filho
 */
public function add($child)
{
    $this->children[] = $child;
}

/**
 * método open()
 * exibe a tag de abertura na tela
 */
private function open()
{
    // exibe a tag de abertura
    echo "<{$this->name}>";
    if ($this->properties)
    {
        // percorre as propriedades
        foreach ($this->properties as $name=>$value)
        {

```

```

        echo "{$name}='{$value}'";
    }
}
echo '>';
}

```

Para exibir a tag na tela, basta executarmos o método `show()`. Este método exibirá, por meio do método `open()`, a tag de abertura e percorrerá todos os objetos-filho por meio de um laço de repetição `foreach` sobre a propriedade `$children`. Além disso, tal método exibirá a tag de fechamento por meio do método `close()`. Veja que, dentro do laço de repetição, testamos cada um dos filhos, se forem objetos (`is_object`); caso contrário, são tratados como strings.

```

/**
 * método show()
 * exibe a tag na tela, juntamente com seu conteúdo
 */
public function show()
{
    // abre a tag
    $this->open();
    echo "\n";
    // se possui conteúdo
    if ($this->children)
    {
        // percorre todos objetos filhos
        foreach ($this->children as $child)
        {
            // se for objeto
            if (is_object($child))
            {
                $child->show();
            }
            else if ((is_string($child)) or (is_numeric($child)))
            {
                // se for texto
                echo $child;
            }
        }
        // fecha a tag
        $this->close();
    }
}
/**
 * método close()
 * Fecha uma tag HTML
*/

```

```
private function close()
{
    echo "</{$this->name}>\n";
}
?>
```

5.2.1.1 Exemplo

No exemplo a seguir, incluiremos, por meio do comando `include_once`, a classe recém-criada e demonstraremos a criação de algumas tags pelo uso da classe `TElement`. O primeiro elemento que criamos é o `html`. Em seguida, adicionamos um elemento `head`, para então adicionarmos neste um elemento `title`, contendo o título da página. Na seção `body`, adicionamos um elemento `center`, que centralizará todos os elementos por ele contidos. Posteriormente, adicionamos uma série de elementos, começando com um parágrafo, escrito “Sport Club Internacional”, a imagem `inter.png`, um separador horizontal (`hr`), e um link para o site `www.internacional.com.br`, todos centralizados. Ao final, criamos um botão (elemento do tipo `input`), que, ao ser clicado, exibe uma mensagem de alerta com o conteúdo “Clube do povo do Rio Grande do Sul!”.

Você deve estar se questionando sobre o por quê de se criar uma classe para exibir tags HTML na tela, uma vez que a utilização simples do HTML implica em menos linhas de código. Neste momento você terá certa dificuldade em assimilar isso, mas fique atento aos próximos exemplos, não somente neste capítulo. Sempre que a classe `TElement` for utilizada, imagine como ficaria o código caso tivéssemos que exibir o conteúdo em HTML diretamente na tela por meio de comandos como o `echo`. Neste ponto lembre-se da palavra “legibilidade”, ou seja, o código-fonte fica mais legível e mais claro de ser interpretado quando se utiliza puramente uma única linguagem – estamos falando do PHP e não mais de pedaços de código HTML inseridos dentro de nossos programas. Veja o programa que segue e confira na Figura 5.4 o resultado do exemplo.

tags.php

```
<?php
include_once 'app/widgets/TElement.class.php';

// inicia o documento html
$html = new TElement('html');

// instancia seção head
$head= new TElement('head');
$html->add($head); // adiciona ao html
```

```
// define o título da página
$title= new TElement('title');
$title->add('Título da página');
$head->add($title); // adiciona ao head

// inicia o body do html
$body = new TElement('body');
$body->bgcolor='#ffffdd';
$html->add($body); // adiciona ao html

$center = new TElement('center');
$body->add($center);

// instancia um parágrafo
$p= new TElement('p');
$p->align = 'center';
$p->add('Sport Club Internacional');
$center->add($p); // adiciona ao body

// instancia uma imagem
$img= new TElement('img');
$img->src  = 'app.images/inter.png';
$img->width = '120';
$img->height= '120';
$center->add($img); // adiciona ao body

// instancia um separador horizontal
$hr= new TElement('hr');
$hr->width = '150';
$hr->align = 'center';
$center->add($hr); // adiciona ao body

// instancia um link
$a= new TElement('a');
$a->href = 'http://www.internacional.com.br';
$a->add('Visite o Site Oficial');
$center->add($a); // adiciona ao body

// instancia uma quebra de linhas
$br= new TElement('br');
$center->add($br); // adiciona ao body

// instancia um botão
$input= new TElement('input');
$input->type    = 'button';
$input->value   = 'clique aqui para saber...';
```

```
$input->onclick = "alert('Clube do Povo do Rio Grande do Sul!');  
$center->add($input); // adiciona ao body  
  
// exibe o html com todos seus elementos filhos  
$html->show();  
?>
```

Resultado:

```
<html>  
<head>  
<title>  
Título da página</title>  
</head>  
<body bgcolor="#ffffdd">  
<center>  
<p align="center">  
Sport Club Internacional</p>  
  
<hr width="150" align="center">  
<a href="http://www.internacional.com.br">  
Visite o Site Oficial</a>  
<br>  
<input type="button" value="clique aqui para saber..."  
      onclick="alert('Clube do Povo do Rio Grande do Sul!')">  
</center>  
</body>  
</html>
```



Figura 5.4 – Resultado do programa.

5.2.2 Folhas de estilo

Assim como as tags HTML, temos os estilos CSS (Cascading Style Sheets). CSS é uma linguagem utilizada para definir a apresentação e estilo de um documento HTML. Seu objetivo é prover a separação entre o formato (apresentação) dos dados e seu conteúdo.

Com abstrairmos as tags HTML para utilizá-las de maneira orientada a objetos, também construiremos uma classe para manipular estilos css por meio de uma interface orientada a objetos. Para isso, iremos construir a classe `TStyle`. Seu comportamento é bastante similar ao da classe `TElement`. Em seu método construtor ela receberá o nome do estilo. Assim como na classe `TElement`, definiremos os atributos do estilo por meio da atribuição de valores às propriedades do objeto. Como tais propriedades não existem, a classe `TStyle` armazenará esses valores no array interno `$properties` que será lido (iterado) posteriormente para apresentação deste estilo em tela. Veja na Figura 5.5 a classe `TStyle`.

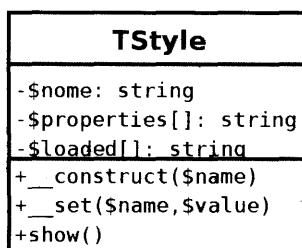


Figura 5.5 – Classe `TStyle`.

`TStyle.php`

```

<?php
/**
 * classe TStyle
 * classe para abstração Estilos CSS
 */
class TStyle
{
    private $name;           // nome do estilo
    private $properties;     // atributos
    static private $loaded;   // array de estilos carregados

    /**
     * método construtor
     * instancia uma tag html
     * @param $name = nome da tag
     */
  
```

```
public function __construct($name)
{
    // atribui o nome do estilo
    $this->name = $name;
}
```

Nas folhas de estilo utilizamos o hífen “-” para separar palavras compostas, como em `font-style`. Como `font-style` seria um nome inválido de propriedade do PHP, optamos por utilizar o sinal de sublinhado “_” para definir as propriedades e realizar esta troca por “-” no momento de atribuição do valor a uma propriedade do estilo.

```
/**
 * método __set()
 * intercepta as atribuições à propriedades do objeto
 * @param $name    = nome da propriedade
 * @param $value   = valor
 */
public function __set($name, $value)
{
    // substitui o "_" por "-" no nome da propriedade
    $name = str_replace('_', '-', $name);
    // armazena os valores atribuídos ao array properties
    $this->properties[$name] = $value;
}
```

O método `show()` irá inserir a folha de estilos no documento. A classe `TStyle` terá um mecanismo de proteção para evitar que um estilo seja inserido no documento duas vezes. Tal mecanismo será provido por meio da utilização da propriedade estática `$loaded`. Como as propriedades estáticas são relativas às classes, sempre que tentarmos exibir um estilo que já fora exibido, a segunda tentativa será ignorada. Lembre-se que uma propriedade estática mantém seu valor ao longo de várias instâncias de uma mesma classe (objetos), de modo que uma propriedade simples só é visível dentro do objeto ao qual ela pertence. A propriedade `$loaded` será um vetor indexado pelo nome do estilo que irá conter um valor `TRUE` (carregada) ou não estará definida.

```
/**
 * método show()
 * exibe a tag na tela
 */
public function show()
{
    // verifica se este estilo já foi carregado
    if (!self::$loaded[$this->name])
    {
        echo "<style type='text/css' media='screen'>\n";
        // exibe a abertura do estilo
```

```

echo ".{$this->name}\n";
echo "{\n";
if ($this->properties)
{
    // percorre as propriedades
    foreach ($this->properties as $name=>$value)
    {
        echo "\t {$name}: {$value};\n";
    }
}
echo "}\n";
echo "</style>\n";
// marca o estilo como já carregado
self::$loaded[$this->name] = TRUE;
}
}
?>

```

5.2.2.1 Exemplo

No exemplo a seguir estamos fazendo uso da classe criada. Neste caso estamos criando dois estilos, um nomeado `texto` e outro nomeado `celula`. O estilo `texto` define algumas características de fonte, cor e tamanho, ao passo que o estilo `celula` define algumas características de espaçamento e margem. Veja que, por engano, os estilos estão sendo exibidos duas vezes, o que poderia fazer com que ficassem duplicados no documento. No entanto, em decorrência do nosso mecanismo de controle (propriedade estática `$loaded`), os estilos são exibidos na primeira vez, mas a segunda chamada do método `show()` será ignorada por causa desse controle. Confira a seguir o exemplo de utilização da classe `TStyle`, bem como o resultado de sua execução.

estilos.php

```

<?php
// inclui classe TStyle
include_once 'app.widgets/TStyle.class.php';

// instancia objeto TStyle
// define características de um estilo chamado texto
$estilo1 = new TStyle('texto');
$estilo1->font_family      = 'arial,verdana,sans-serif';
$estilo1->font_style       = 'normal';
$estilo1->font_weight      = 'bold';
$estilo1->color            = 'white';
$estilo1->text_decoration = 'none';
$estilo1->font_size         = '10pt';

```

```
// instancia objeto TStyle  
// define características de um estilo chamado célula  
$estilo2 = new TStyle('celula');  
$estilo2->background_color = 'white';  
$estilo2->padding_top      = '10px';  
$estilo2->padding_bottom   = '10px';  
$estilo2->padding_left     = '10px';  
$estilo2->padding_right    = '10px';  
$estilo2->margin_left      = '5px';  
$estilo2->width            = '142px';  
$estilo2->height           = '154px';  
  
// exibe estilos na tela  
$estilo1->show();  
$estilo2->show();  
  
// exibe estilos na tela  
$estilo1->show();  
$estilo2->show();  
?>
```

 **Resultado:**

```
<style type='text/css' media='screen'>  
.texto  
{  
    font-family: arial,verdana,sans-serif;  
    font-style: normal;  
    font-weight: bold;  
    color: white;  
    text-decoration: none;  
    font-size: 10pt;  
}  
</style>  
<style type='text/css' media='screen'>  
.celula  
{  
    background-color: white;  
    padding-top: 10px;  
    padding-bottom: 10px;  
    padding-left: 10px;  
    padding-right: 10px;  
    margin-left: 5px;  
    width: 142px;  
    height: 154px;  
}  
</style>
```

Demonstraremos agora a utilização em conjunto das classes TElement e TStyle. No exemplo a seguir, estamos primeiramente criando um estilo nomeado `estilo_texto`, definindo algumas características de exibição de textos, como cor e fonte. Em seguida, estamos criando um elemento do tipo parágrafo (`<p>`). Neste parágrafo, adicionaremos o texto “Sport Club Internacional” e, posteriormente, definiremos a classe do elemento (propriedade classe) como sendo `estilo_texto`. Desta forma, estamos escrevendo um parágrafo na tela utilizando um estilo css. Veja na Figura 5.6 o resultado.

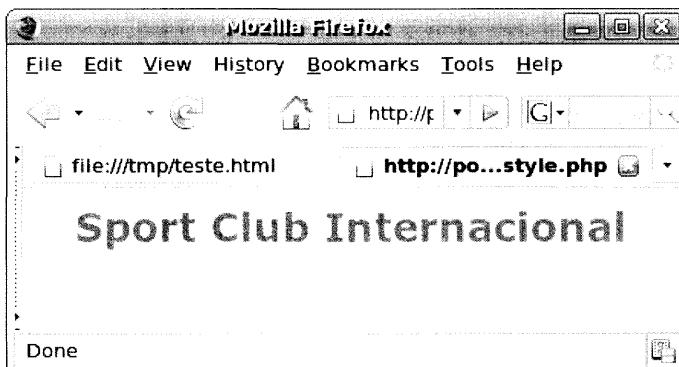


Figura 5.6 – Utilização de TElement com TStyle.

tagstyle.php

```

<?php
// inclui as classes
include_once 'app.widgets/TElement.class.php';
include_once 'app.widgets/TStyle.class.php';

// cria um estilo
$style = new TStyle('estilo_texto');
$style->color      = '#FF0000';
$style->font_family = 'Verdana';
$style->font_size   = '20pt';
$style->font_weight= 'bold';
$style->show();

// instancia um parágrafo
$texto= new TElement('p');
$texto->align = 'center';
$texto->add('Sport Club Internacional');

// define o estilo do parágrafo
$texto->class = 'estilo_texto';
$texto->show();
?>

```

Veja a seguir o código-fonte da página HTML gerada:

```
<style type='text/css' media='screen'>
.estilo_texto
{
    color: #FF0000;
    font-family: Verdana;
    font-size: 20pt;
    font-weight: bold;
}
</style>
<p align="center" class="estilo_texto">
Sport Club Internacional</p>
```

5.2.3 Imagens

Como já temos uma classe para formação dos elementos HTML (`TElement`), podemos começar a compor outros elementos baseados nela. Para demonstrar novamente o uso da classe `TElement`, criaremos uma pequena classe para exibição de imagens. A primeira forma que temos de reaproveitar a classe `TElement` é acessá-la via mecanismo de composição, de modo que a classe `TImage` irá manter uma referência para um objeto do tipo `TElement` por meio de uma de suas propriedades. Basicamente essa classe recebe a localização da imagem em seu método construtor, instancia um objeto `TElement` e, em seu método `show()`, trata da exibição da mesma, como os detalhes da utilização da tag `` escondidos pela interface de `TElement`. Veja a seguir o código da classe `TImage` e o diagrama demonstrando a composição.

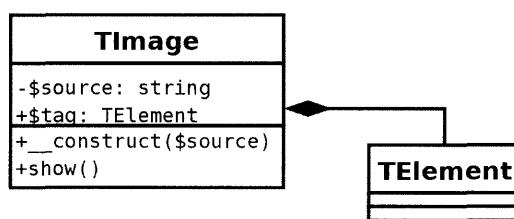


Figura 5.7 – Classe `TImage` compondo `TElement`.

`TImage.class.php`

```
<?php
/**
 * classe TImage
 * classe para exibição de imagens
 */
class TImage
{
```

```

private $source;      // localização da imagem
private $tag;        // objeto TElement
/**
 * método construtor
 * instancia objeto TImage
 * @param $source = localização da imagem
 */
public function __construct($source)
{
    // atribui a localização da imagem
    $this->source = $source;
    // instancia objeto TElement
    $this->tag    = new TElement('img');
}
/**
 * método show()
 * exibe imagem na tela
 */
public function show()
{
    // define algumas propriedades da tag
    $this->tag->src = $this->source;
    $this->tag->border=0;

    // exibe tag na tela
    $this->tag->show();
}
}
?>

```

A segunda forma que temos de reutilizar a classe `TElement` é por meio de uma herança. Neste caso, a classe `TImage` irá herdar todas as funcionalidades de `TElement`. O código ficará mais simples, pois o método `show()` da classe `TElement` já trata da exibição do elemento, bastando para nós definirmos algumas de suas propriedades, como a localização da imagem (`src`). Veja no diagrama a seguir a herança entre as classes.

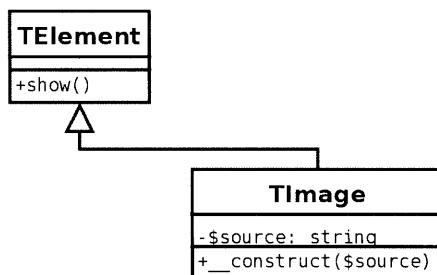


Figura 5.8 – Classe `TImage` herdando `TElement`.

 **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.

image.php

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

// instancia objeto imagem
$gnome = new TImage('app.images/gnome.png');
// exibe objeto imagem
$gnome->show();

// instancia objeto imagem
$gimp= new TImage('app.images/gimp.png');
// exibe objeto imagem
$gimp->show();
?>
```

Resultado:

```


```

5.2.4 Textos

Assim como no exemplo anterior, no qual criamos uma classe para exibir imagens baseada na classe `TElement`, criaremos uma classe para exibir textos. Esta classe também utilizará o mecanismo de herança para adquirir todo o comportamento da classe `TElement`. Esta classe receberá, no seu método construtor, o texto a ser exibido e terá o método `setAlign()` para definir seu alinhamento. Os demais estilos de formatação, como negrito (``), sublinhado (`<u>`) ou itálico (`<i>`), podem ser definidos diretamente dentro do texto a ser exibido. A classe se chamará `TParagraph` em razão do fato de utilizarmos a tag `<p>` de abertura de novo parágrafo para conter o texto. Vejamos a seguir o código da classe e também o diagrama com sua representação.

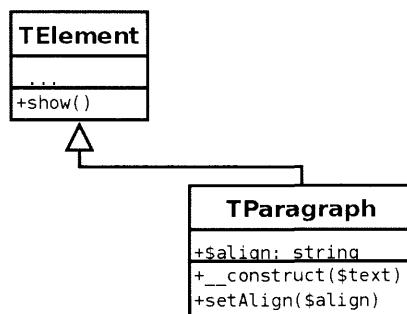


Figura 5.10 – Classe `TParagraph`.

 TParagraph.class.php

```
<?php
/**
 * classe TParagraph
 * classe para exibição de parágrafos
 */
class TParagraph extends TElement
{
    /**
     * método construtor
     * instancia objeto TParagraph
     * @param $texto = texto a ser exibido
     */
    public function __construct($text)
    {
        parent::__construct('p');
        // atribui o conteúdo do texto
        parent::add($text);
    }

    /**
     * método setAlign()
     * define o alinhamento do texto
     * @param $align = alinhamento do texto
     */
    function setAlign($align)
    {
        $this->align = $align;
    }
}
?>
```

5.2.4.1 Exemplo

No exemplo a seguir faremos uso da classe recém-criada para exibição de dois parágrafos, um alinhado à esquerda e o outro à direita. Confira na Figura 5.11, o resultado da execução do programa.

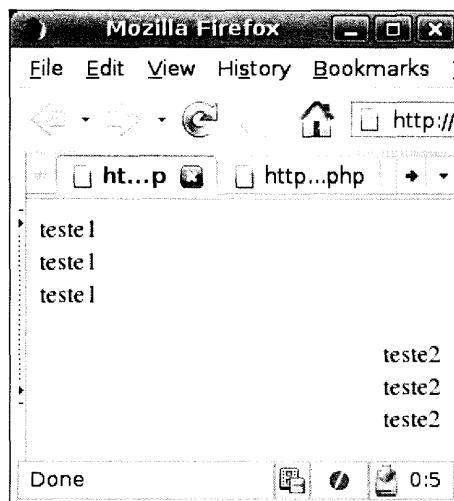


Figura 5.11 – Exibindo textos com a classe TParagraph.

paragraph.php

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

// instancia objeto parágrafo
$texto1= new TParagraph('teste1<br>teste1<br>teste1');
$texto1->setAlign('left');

// exibe objeto
$texto1->show();

// instancia objeto parágrafo
$texto2= new TParagraph('teste2<br>teste2<br>teste2');
$texto2->setAlign('right');

// exibe objeto
$texto2->show();
?>
```

Resultado:

```
<p align="left">teste1<br>teste1<br>teste1</p>
<p align="right">teste2<br>teste2<br>teste2</p>
```

5.3 Contêineres

Nesta seção criaremos alguns objetos contêineres, os quais podem conter outros objetos em seu interior, seguindo certa disposição visual.

5.3.1 Tabelas

Um dos elementos mais utilizados em páginas web são as tabelas, as quais são, muitas vezes, utilizadas em listagens de dados e outras vezes imperceptíveis ao usuário (sem bordas), servindo para organizar os elementos de uma página (sua estrutura visual). O certo é que as tabelas são um tipo de contêiner muito utilizado, pois oferecem certa facilidade de organizar seu conteúdo sob a forma de linhas e colunas, possibilitando a mescla de células, cores diferentes para cada célula, espessuras de borda, dentre outros.

Para demonstrar o funcionamento de uma tabela, construiremos um exemplo em que o objetivo é apresentar na tela uma matriz de dados. Esta matriz contém informações como código, nome e site de algumas pessoas, juntamente com o salário que deve ser totalizado na última coluna. Primeiramente construiremos um exemplo simples, envolvendo PHP e HTML, para depois propor uma estrutura totalmente orientada a objeto para atingir o mesmo resultado.

5.3.1.1 Implementação simples

Como podemos ver no exemplo a seguir, a implementação simples é recheada de código HTML. Você deve se perguntar “Qual é o problema?”. Para começar, o código HTML definirá a estrutura visual da aplicação. Imagine vários desses arquivos dentro do sistema. Se algum dia resolvemos trocar a estrutura de tabelas para uma estrutura *tableless*, teremos de editar todos os arquivos à mão, justamente para realizar as substituições necessárias das partes que utilizam código HTML. Veja na Figura 5.12 a tabela criada.

The screenshot shows a Mozilla Firefox window with the title bar "Mozilla Firefox". The address bar displays the URL "http://poo/cap5/tabelas_html". The main content area shows a simple HTML table with the following data:

Código	Nome	Site	Salário
1	Maria do Rosário	http://www.maria.com.br	1200
2	Pedro Cardoso	http://www.pedro.com.br	800
3	João de Barro	http://www.joao.com.br	1500
3	Joana Pereira	http://www.joana.com.br	700
3	Erasmo Carlos	http://www.erasmo.com.br	2500
Total			6700

Figura 5.12 – Tabela em HTML.

 tabelas_html.php

```
<?php  
// constrói matriz com os dados  
$dados[] = array(1, 'Maria do Rosário', 'http://www.maría.com.br', 1200);  
$dados[] = array(2, 'Pedro Cardoso', 'http://www.pedro.com.br', 800);  
$dados[] = array(3, 'João de Barro', 'http://www.joão.com.br', 1500);  
$dados[] = array(3, 'Joana Pereira', 'http://www.joana.com.br', 700);  
$dados[] = array(3, 'Erasmo Carlos', 'http://www.erasmo.com.br', 2500);  
  
// abre tabela  
echo '<table border=1 width=600>';  
  
// exibe linha com o cabeçalho  
echo '<tr bgcolor="#a0a0a0">';  
echo '<td> Código </td>';  
echo '<td> Nome </td>';  
echo '<td> Site </td>';  
echo '<td> Salário </td>';  
echo '</tr>';  
  
$i = 0;  
// percorre os dados  
foreach ($dados as $pessoa)  
{  
    // verifica qual cor utilizar para o fundo  
    $bgcolor = ($i % 2) == 0 ? '#d0d0d0' : '#ffffff';  
  
    // imprime a linha  
    echo "<tr bgcolor='$bgcolor'>";  
    // exibe o código  
    echo "<td>{$pessoa[0]}</td>";  
    // exibe o nome  
    echo "<td>{$pessoa[1]}</td>";  
    // exibe o site  
    echo "<td>{$pessoa[2]}</td>";  
    // exibe o salário  
    echo "<td align='right'>{$pessoa[3]}</td>";  
    echo '</tr>';  
    $total += $pessoa[3]; // soma o salário  
    $i++;  
}  
// exibe células vazias mescladas  
echo '<tr>';  
echo '<td colspan=3>Total</td>';  
  
// exibe linha com totalizador  
echo '<td align="right" bgcolor="#a0a0a0">';
```

```

echo $total;
echo '</td>';
echo '</tr>';

// finaliza a tabela
echo '</table>';
?>

```

5.3.1.2 Proposta

Nossa proposta é transformar a tabela HTML em uma estrutura orientada a objetos. Para isso, identificamos três objetos que compõem uma tabela: a tabela (`TTable`), a linha (`TTableRow`) e a célula (`TTableCell`). Veja no diagrama a seguir o relacionamento entre as três classes propostas. Uma tabela é composta de linhas e uma linha é composta de células.

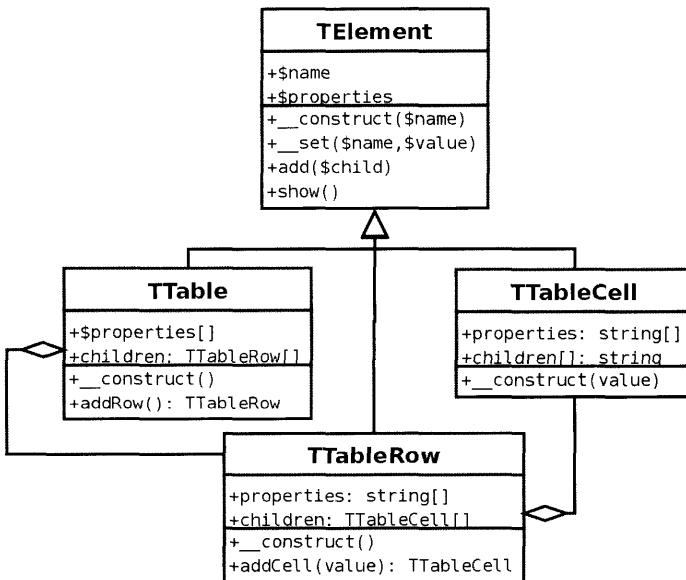


Figura 5.13 – Objetos Tabela, Linha e Célula.

Todas as classes serão filhas de `TElement`. Dessa forma, já teremos alguns métodos básicos implementados, como o `add()` e o `show()`. A tabela (`TTable`) é nossa classe principal. Em seu método construtor definimos qual será o nome da tag que ela irá representar (“table”). A tabela contém o método `addRow()`, responsável por adicionar uma linha na tabela. O método `addRow()` já instancia e retorna um objeto do tipo `TTableRow` para que o programador possa manipulá-lo. Observe que o objeto tabela (`TTable`) é “composto” por vários objetos do tipo linha (`TTableRow`), em uma estrutura de agregação. O método `add()` da classe `TElement` adiciona todos os filhos na propriedade `$children`. O método `show()` da classe `TElement` não será reescrito; sua função é exibir

a tag de abertura (`<table>`) e percorrer cada um dos seus elementos-filho (`$children`) recursivamente, executando o método `show()` um a um.

 **TTable.class.php**

```
<?php
/**
 * classe TTable
 * responsável pela exibição de tabelas
 */
class TTable extends TElement
{
    /**
     * método construtor
     * instancia uma nova tabela
     */
    public function __construct()
    {
        parent::__construct('table');
    }

    /**
     * método addRow
     * agrupa um novo objeto linha (TTableRow) na tabela
     */
    public function addRow()
    {
        // instancia objeto linha
        $row = new TTableRow;
        // armazena no array de linhas
        parent::__add($row);
        return $row;
    }
}
?>
```

Vimos então que os objetos linha (`TTableRow`) são criados (instanciados) pelo método `addRow()` da tabela. Agora que já temos linhas, precisamos adicionar células dentro de uma linha. Para isso, a classe linha (`TTableRow`) provê o método `addCell()`, responsável por instanciar um objeto do tipo `TTableCell`. O método `addCell()` recebe como informação o conteúdo da célula. Note que a linha (`TTableRow`) é composta por objetos do tipo célula (`TTableCell`), também em uma estrutura de agregação, visto que todos os objetos adicionados pelo método `add()` da classe `TElement` fazem parte do vetor `$children`, sendo que, no momento em que exibimos uma linha (`TTableRow`), também exibimos suas células agregadas (objetos `TTableCell`) uma a uma, as quais, por sua vez, também implementam o método `show()`.

 **TTableRow.class.php**

```
<?php
/**
 * classe TTableRow
 * responsável pela exibição de uma linha de uma tabela
 */
class TTableRow extends TElement
{
    /**
     * método construtor
     * instancia uma nova linha
     */
    public function __construct()
    {
        parent::__construct('tr');
    }

    /**
     * método addCell
     * agrupa um novo objeto célula (TTableCell) à linha
     * @param $value = conteúdo da célula
     */
    public function addCell($value)
    {
        // instancia objeto célula
        $cell = new TTableCell($value);
        parent::add($cell);
        // retorna o objeto instanciado
        return $cell;
    }
}
?>
```

Por último temos a célula (`TTableCell`). O objeto célula é automaticamente instanciado pelo método `addCell()` da classe linha (`TTableRow`). Depois de receber essas informações, a célula será exibida em tela pelo método `show()` da classe-pai (`TElement`), responsável por exibir as tags que contêm propriedades e valores das células. Aqui chamamos atenção para um detalhe: o método `addCell()` executa o método `add()` da classe `TElement`, que, por sua vez, aceita tanto uma string quanto um objeto como parâmetro. Já escrevemos isto pensando que a tabela poderá usada como contêiner para outros objetos que serão adicionados a ela. Neste caso, se adicionarmos um objeto pelo método `addCell()`, seu respectivo método `show()` será executado.

 TTableCell.class.php

```
<?php
/**
 * classe TTableCell
 * responsável pela exibição de uma célula de uma tabela
 */
class TTableCell extends TElement
{
    /**
     * método construtor
     * instancia uma nova célula
     * @param $value = conteúdo da célula
     */
    public function __construct($value)
    {
        parent::__construct('td');
        parent::add($value);
    }
}
?>
```

5.3.1.3 Implementação orientada a objetos

No próximo exemplo iremos reescrever a listagem em HTML que fizemos no início desta seção. Para isso, utilizaremos as três classes criadas (TTable, TTableRow e TTableCell). Observe como se efetua a chamada de métodos desses objetos e compare com o primeiro exemplo. Inicialmente há uma evolução quanto à clareza do código-fonte, que se torna mais simples de se entender por abstrair o código HTML. Dessa forma, concentramo-nos no que o programa deve realmente fazer, não em detalhes de exibição em tela. Outro fator importante é que todo o trabalho de exibição e montagem das tags HTML ficou contido dentro das classes criadas. Isto quer dizer que a qualquer momento podemos alterar a forma como os dados são apresentados em tela. Para trocar todas as tabelas do sistema para uma estrutura *tableless*, por exemplo, bastaria alterar o código das três classes (TTable, TTableRow e TTableCell).

Como as classes TTable, TTableRow e TTableCell são filhas de TElement, a qualquer momento podemos definir seus atributos simplesmente definindo valores para suas propriedades como fizemos para definir a largura da tabela (`$table->width = 600`) ou para definir o alinhamento do salário (`$x->align = 'right'`).

 tabelas_oo.php

```
<?php
// inclui classes necessárias
include_once 'app.widgets/TElement.class.php';
```

```
include_once 'app.widgets/TTable.class.php';
include_once 'app.widgets/TTableRow.class.php';
include_once 'app.widgets/TTableCell.class.php';

// constrói matriz com os dados
$dados[] = array(1, 'Maria do Rosário', 'http://www.maria.com.br', 1200);
$dados[] = array(2, 'Pedro Cardoso', 'http://www.pedro.com.br', 800);
$dados[] = array(3, 'João de Barro', 'http://www.joao.com.br', 1500);
$dados[] = array(3, 'Joana Pereira', 'http://www.joana.com.br', 700);
$dados[] = array(3, 'Erasmo Carlos', 'http://www.erasmo.com.br', 2500);

// instancia objeto tabela
$tabela = new TTable;

// define algumas propriedades
$tabela->width = 600;
$tabela->border= 1;

// instancia uma linha para o cabeçalho
$cabecalho = $tabela->addRow();
// define a cor de fundo
$cabecalho->bgcolor = '#a0a0a0'; // cor de fundo

// adiciona células
$cabecalho->addCell('Código');
$cabecalho->addCell('Nome');
$cabecalho->addCell('Site');
$cabecalho->addCell('Salário');

$i = 0;
// percorre os dados
foreach ($dados as $pessoa)
{
    // verifica qual cor utilizar para o fundo
    $bgcolor = ($i % 2) == 0 ? '#d0d0d0' : '#ffffff';

    // adiciona uma linha para os dados
    $linha = $tabela->addRow();
    $linha->bgcolor = $bgcolor;

    // adiciona as células
    $linha->addCell($pessoa[0]);
    $linha->addCell($pessoa[1]);
    $linha->addCell($pessoa[2]);
    $x = $linha->addCell($pessoa[3]);
    $x->align = 'right';
```

```

$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.

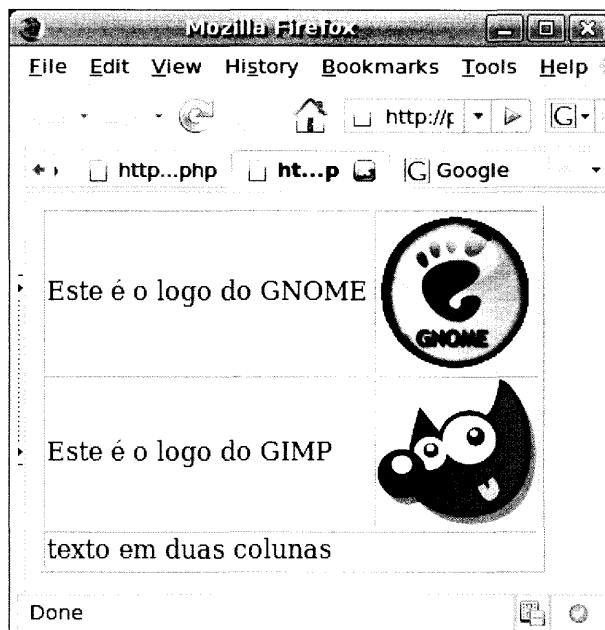


Figura 5.14 – Tabela orientada a objetos.

 tabelas_oo2.php

```
<?php  
// inclui as classes necessárias  
include_once 'app.widgets/TElement.class.php';  
include_once 'app.widgets/TImage.class.php';  
include_once 'app.widgets/TTable.class.php';  
include_once 'app.widgets/TTableRow.class.php';  
include_once 'app.widgets/TTableCell.class.php';  
include_once 'app.widgets/TParagraph.class.php';  
  
// instancia objeto tabela com borda de 1 pixel  
$tabela = new TTable;  
$tabela->border = 1;  
// acrescenta uma linha na tabela  
$linha1 = $tabela->addRow();  
// cria um objeto parágrafo  
$paragrafo= new TParagraph('Este é o logo do GNOME');  
$paragrafo->setAlign('left');  
// adiciona célula contendo o objeto  
$linha1->addCell($paragrafo);  
  
// cria um objeto imagem  
$imagem= new Timage('app.images/gnome.png');  
$linha1->addCell($imagem);  
  
// acrescenta uma linha na tabela  
$linha2 = $tabela->addRow();  
  
// cria um objeto parágrafo  
$paragrafo= new TParagraph('Este é o logo do GIMP');  
$paragrafo->setAlign('left');  
// adiciona célula contendo o objeto  
$linha2->addCell($paragrafo);  
  
// cria um objeto imagem  
$imagem= new Timage('app.images/gimp.png');  
// adiciona célula contendo o objeto  
$linha2->addCell($imagem);  
  
// acrescenta uma linha na tabela  
$linha3 = $tabela->addRow();  
// acrescenta um célula que ocupará o espaço de duas  
$celula = $linha3->addCell(new TParagraph('texto em duas colunas'));  
$celula->colspan = 2;  
  
// exibe a tabela  
$tabela->show();  
?>
```

5.3.2 Painéis

Ao construirmos uma tabela, vimos que o seu layout é um pouco limitado pela existência de linhas e colunas. Em alguns casos precisamos de maior liberdade para dispor os elementos na tela, sem limitações de tamanho ou ordem de inserção. O HTML oferece recursos para o posicionamento em coordenadas absolutas da tela para elementos contidos dentro de um elemento `<div>`. Pensando nisso, criaremos a classe `TPanel` que nos permitirá fixar objetos em determinadas posições do painel, definidas em termos de linha e coluna em pixels. Esta classe facilitará a criação visual de formulários, vista no Capítulo 6, dando total liberdade para posicionar os objetos dentro de sua área.

5.3.2.1 Implementação simples

Primeiramente vamos ver como poderíamos criar um painel com posições fixas em HTML utilizando camadas `<div>`. O princípio é bastante simples. Primeiramente criamos uma camada mestre responsável por encapsular todas as demais que estarão nela contidas. Cada subcamada terá uma posição absoluta dentro da camada mestre (`position:absolute`) e uma posição fixa definida em pixels (`left` e `top`). A partir disso, basta colocarmos o conteúdo que quisermos dentro das subcamadas para dispor nossos conteúdos nas posições determinadas. Para ficar mais evidente que os objetos estão ancorados em determinadas posições do painel, iremos alterar seu estilo e colocar uma imagem de fundo que representa uma grade pontilhada. Na Figura 5.15 temos um painel.

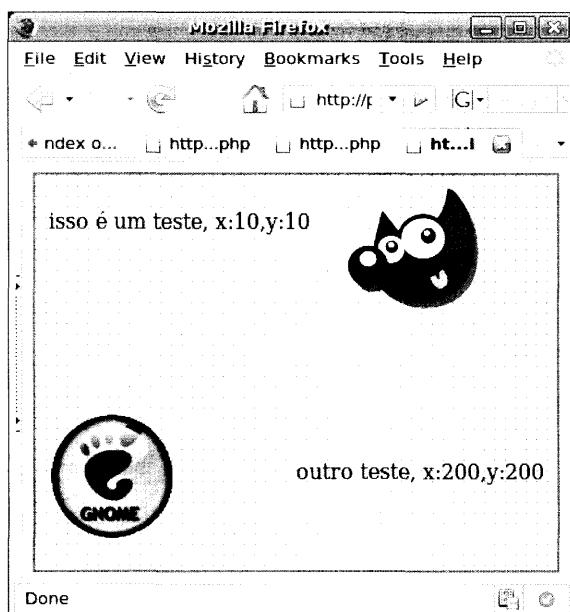


Figura 5.15 – Painel com posições fixas.

Nas coordenadas (10,10) colocaremos o texto “isso é um teste”; já nas coordenadas (200,200), colocaremos o texto “outro teste”. Nas coordenadas (10,180) colocaremos a imagem `gnome.png` e nas coordenadas (240,10) colocaremos a imagem `gimp.png`.

painel.html

```
<div style="position:relative; width:400px;
background-image: url(app.images/background.png);
height:300px; border:2px solid; border-color:grey">

<div style="position:absolute; left:10px; top:10px;">
<p align="left">isso é um teste, x:10,y:10</p>
</div>

<div style="position:absolute; left:200px; top:200px;">
<p align="left">outro teste, x:200,y:200</p>
</div>

<div style="position:absolute; left:10px; top:180px;">

</div>

<div style="position:absolute; left:240px; top:10px;">

</div>
</div>
```

5.3.2.2 Proposta

A seguir temos a implementação da classe proposta. A classe `TPanel` receberá em seu método construtor a largura e a altura do painel. Seu método construtor também criará o estilo `tpanel`. Este estilo será utilizado para definir as características (altura, largura, borda, cor de fundo) do painel que irá conter os elementos. Na Figura 5.16 temos a classe `TPanel`.

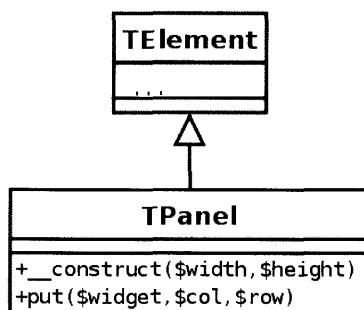


Figura 5.16 – Classe `TPanel`.

 TPanel.class.php

```
<?php
/**
 * classe TPanel
 * painel de posições fixas
 */
class TPanel extends TElement
{
    /**
     * método __construct()
     * instancia objeto TPanel.
     * @param $width    = largura do painel
     * @param $height   = altura do painel
     */
    public function __construct($width, $height)
    {
        // instancia objeto TStyle
        // para definir as características do painel
        $painel_style = new TStyle('tpanel');
        $painel_style->position      = 'relative';
        $painel_style->width         = $width;
        $painel_style->height        = $height;
        $painel_style->border        = '2px solid';
        $painel_style->border_color  = 'grey';
        $painel_style->background_color = '#f0f0f0';

        // exibe o estilo na tela
        $painel_style->show();

        parent::__construct('div');
        $this->class = 'tpanel';
    }
}
```

Esta classe terá o método `put()`, responsável por definir quais objetos estarão em quais posições do painel. O método `put()` recebe como primeiro parâmetro um objeto (widget), também recebe a coluna e a linha nas quais este objeto será posicionado. Esse método simplesmente chama o método `add()` da classe-pai (`TElement`) para adicionar o widget no painel, mas note que, antes disto, ele adiciona o widget em um objeto (`$camada`), como se este objeto fosse um envelope para o widget. Dizemos isto, pois o objeto `$camada` envolverá o widget antes de o mesmo ser adicionado no painel. Fizemos isto, pois é a única forma de definirmos as coordenadas do widget antes de ele ser adicionado ao painel principal.

```
/**
 * método put()
 * posiciona um objeto no painel
```

```
* @param $widget = objeto a ser inserido no painel
* @param $col    = coluna em pixels.
* @param $row    = linha em pixels.
*/
public function put($widget, $col, $row)
{
    // cria uma camada para o widget
    $camada = new TElement('div');
    // define a posição da camada
    $camada->style = "position:absolute; left:{$col}px; top:{$row}px;";
    // adiciona o objeto (widget) à camada recém-criada
    $camada->add($widget);

    // adiciona widget no array de elementos
    parent::add($camada);
}
}

?>
```

5.3.2.3 Implementação orientada a objetos

Reescreveremos em PHP o exemplo inicial, antes escrito em HTML. Primeiramente iremos incluir as classes necessárias no código-fonte e, em seguida, instanciar alguns objetos de texto e imagem, ancorando-os em determinadas posições dentro do painel. Iremos basicamente colocar os mesmos objetos do no nosso primeiro exemplo. Na Figura 5.17 temos o resultado deste programa.

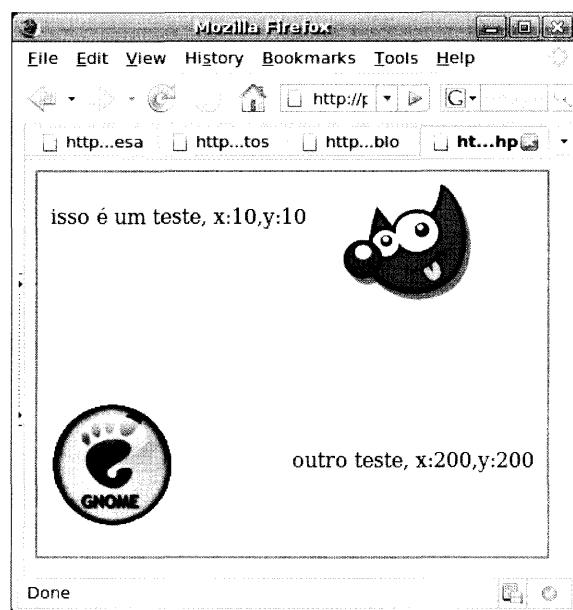


Figura 5.17 – Painel criado pela classe TPanel.

 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

camada). Esta camada pode ter qualquer conteúdo. Primeiramente demonstraremos como criar essa camada utilizando apenas HTML+CSS. Em seguida, escreveremos uma classe que se chamará `TWindow`.

5.3.3.1 Implementação simples

Para implementar nossas janelas utilizaremos uma camada `<div>` para cada janela, que irá conter ainda uma tabela dentro de si. Na primeira linha da tabela teremos o título e o botão fechar; na segunda, teremos o conteúdo da janela.

No código a seguir criaremos alguns estilos. O primeiro estilo (`table_style`) será utilizado para definir as características da tabela contida dentro da janela. Os demais (`panel_style1` e `panel_style2`) irão definir as características de posicionamento e dimensão de cada uma das camadas (nossas janelas pop-up).

Após a definição dos estilos, você verá algumas ocorrências de `<div id="panel..">`, em que cada `<div>` corresponde a uma janela. Dentro da camada, temos uma tabela na qual sua primeira linha corresponde ao título da janela e possui um botão fechar, cuja tarefa é esconder a camada em que a janela está contida, alterando sua propriedade `display` para `none`. Na segunda linha desta tabela temos o conteúdo da janela propriamente dito. Veja na Figura 5.18 o resultado deste programa.

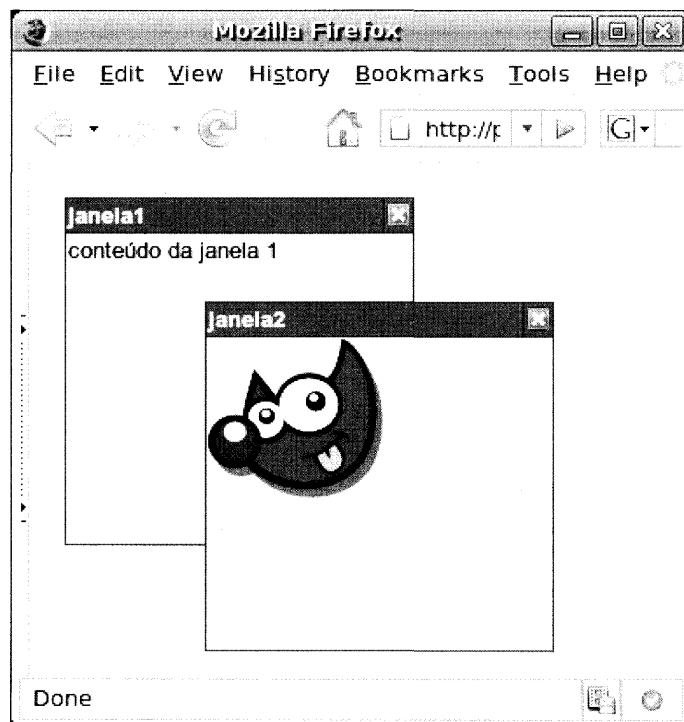


Figura 5.18 – Janela em camadas.

 window.html

```
<style type='text/css' media='screen'>
/*
 * estilo utilizado para a tabela contida pelo painel
 */
.table_style
{
    font-family: Arial, Verdana, Sans-serif;
    font-size: 10pt;
    border-collapse: collapse;
}
/*
 * estilo utilizado para o primeiro painel
 */
.panel_style1
{
    position: absolute;
    left: 20px;
    top: 20px;
    width: 200px;
    height: 200px;
    z-index: 10000;
}
/*
 * estilo utilizado para o segundo painel
 */
.panel_style2
{
    position: absolute;
    left: 100px;
    top: 80px;
    width: 200px;
    height: 200px;
    z-index: 10000;
}
</style>
<div id="panel1" class="panel_style1">
    <table bgcolor="#e0e0e0" width="100%" border="1" height="100%" class="table_style">
        <tr bgcolor="#707070" height="20px">
            <td width="100%"><font color=white><b>janela1</b></font></td>
            <td align="center" valign="middle">
                <a href="#" onclick="document.getElementById('panel1').style.display='none!'">
                    
                </a>
            </td>
        </tr>
    </table>
</div>
```

```
<tr valign="top">
    <td colspan="2"><p align="left">conteúdo da janela 1</p></td>
</tr>
</table>
</div>

<div id="pane12" class="panel_style2">
    <table bgcolor="#e0e0e0" width="100%" border="1" height="100%" class="table_style">
        <tr bgcolor="#707070" height="20px">
            <td width="100%"><font color=white><b>janela2</b></font></td>
            <td align="center" valign="middle">
                <a href="#" onclick="document.getElementById('pane12').style.display='none'">
                    
                </a>
            </td>
        </tr>
        <tr valign="top">
            <td colspan="2"></td>
        </tr>
    </table>
</div>
```

5.3.3.2 Proposta

Agora que já vimos como construir um código HTML que exiba camadas em posições fixas simulando janelas, podemos construir uma classe que faça isto de forma organizada e orientada a objetos. Tal classe se chamará `TWindow` e irá prover um método construtor que recebe o título da janela como parâmetro. Também haverá o método `setPosition()` que recebe as coordenadas X,Y do canto superior esquerdo da janela (onde ela será posicionada na tela). O método `setSize()` definirá suas dimensões (largura e altura), e o método `add()` adicionará conteúdo à janela, armazenando-o no atributo `$this->content`. Este atributo deve necessariamente ser um objeto, pois sobre ele será posteriormente executado o método `show()` para exibi-lo na tela.

Como poderemos criar várias janelas na mesma aplicação é necessário que cada uma tenha um identificador próprio. Precisaremos de uma variável contadora que seja visível dentre todos os objetos `TWindow` criados. Para isso, criaremos a propriedade estática `$counter`. Esta propriedade estática será incrementada sempre que instanciarmos uma nova janela pelo método construtor. Veja na Figura 5.19 a classe `TWindow`.

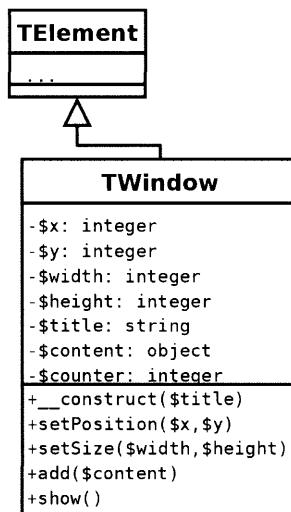


Figura 5.19 – Classe TWindow.

TWindow.class.php

```

<?php
/**
 * classe TWindow
 * TWindow é um container que exibe seu conteúdo em uma camada simulando uma janela
 */
class TWindow
{
    private $x;          // coluna
    private $y;          // linha
    private $width;       // largura
    private $height;      // altura
    private $title;       // título da janela
    private $content;     // conteúdo da janela
    static private $counter; // contador

    /**
     * método construtor
     * incrementa o contador de janelas
     */
    public function __construct($title)
    {
        // incrementa o contador de janelas
        // para exibir cada um com um ID diferente
        self::$counter++;
        $this->title = $title;
    }
}
  
```

```
/*
 * método setPosition()
 * define a coluna e linha (x,y) que a janela será exibido na tela
 * @param $x = coluna (em pixels)
 * @param $y = linha (em pixels)
 */
public function setPosition($x, $y)
{
    // atribui os pontos cardinais do canto superior esquerdo da janela
    $this->x = $x;
    $this->y = $y;
}
/*
 * método setSize()
 * define a largura e altura da janela em tela
 * @param $width = largura (em pixels)
 * @param $height = altura (em pixels)
 */
public function setSize($width, $height)
{
    // atribui as dimensões
    $this->width = $width;
    $this->height = $height;
}
/*
 * método add()
 * adiciona um conteúdo à janela
 * @param $content = conteúdo a ser adicionado
 */
public function add($content)
{
    $this->content = $content;
}
```

O método `show()` exibirá a janela na tela. Cada janela será contida por uma camada `<div>` e, para declararmos as características de exibição desta camada, precisaremos criar um estilo. O objeto `$style` terá como nome `TWindow` mais a propriedade estática `$counter`, que é incrementada a cada nova instância de janela. Assim, cada estilo será único. O estilo definirá o posicionamento da janela (`left`, `top`) e também suas dimensões (`width` e `height`).

Em seguida, criaremos o objeto `$painei`, que será a camada principal da janela. Dentro do `$painei` colocamos uma tabela, a qual terá duas linhas e duas colunas. A primeira linha é a barra de título e nela teremos o título da janela (na primeira célula) e o botão de fechar, ou seja, esconder a camada (na segunda célula). A segunda linha terá o conteúdo da janela.

```
/*
 * método show()
 * exibe a janela na tela
 */
public function show()
{
    $window_id = 'TWindow'.self::$counter;
    // instancia objeto TStyle para definir as características
    // de posicionamento e dimensão da camada criada
    $style= new TStyle($window_id);
    $style->position      = 'absolute';
    $style->left         = $this->x;
    $style->top          = $this->y;
    $style->width        = $this->width;
    $style->height       = $this->height;
    $style->background   = '#e0e0e0';
    $style->border       = '1px solid #000000';
    $style->z_index     = "10000";

    // exibe o estilo em tela
    $style->show();

    // cria tag <div> para a camada que representará a janela
    $painel = new TElement('div');
    $painel->id     = $window_id;      // define o ID
    $painel->class  = $window_id;      // define a classe CSS

    // instancia objeto TTable
    $table = new TTable;
    // define as propriedades da tabela
    $table->width  = '100%';
    $table->height = '100%';
    $table->style  = 'border-collapse:collapse';

    // adiciona uma linha para o título
    $row1 = $table->addRow();
    $row1->bgcolor = '#707070';
    $row1->height  = '20px';

    // adiciona uma célula para o título
    $titulo = $row1->addCell("<font face=Arial size=2 color=white><b>{$this->title}</b></font>");
    $titulo->width = '100%';
```

```
// cria um link com ação para esconder o <div>
$link = new TElement('a');
$link->add(new TImage("app.images/ico_close.png"));
$link->onclick = "document.getElementById('$window_id').style.display='none'";

// adiciona uma célula com o link de fechar
$cell = $row1->addCell($link);

// cria uma linha para o conteúdo
$row2 = $table->addRow();
$row2->valign = 'top';

// adiciona o conteúdo ocupando duas colunas (colspan)
$cell = $row2->addCell($this->content);
$cell->colspan = 2;

// adiciona a tabela ao painel
$painel->add($table);
// exibe o painel
$painel->show();
}

}

?>
```

5.3.3.3 Implementação orientada a objetos

Neste momento, utilizaremos a classe `TWindow` para recriar o exemplo anterior construído em HTML. Veja que não estamos mais incluindo todas as classes no início do código-fonte; estamos utilizando a função `__autoload()` que faz a carga das classes automaticamente baseada no seu nome, a partir do diretório `app.widgets` que é onde estamos gravando nossos componentes.

Neste exemplo, estamos criando três janelas em posições diferentes da tela, contendo textos (instâncias da classe `TParagraph`), imagens (instâncias de `TImage`) ou painéis (instâncias de `TPanel`) que podem conter em seu interior diversos outros objetos. Dentro de uma janela `TWindow` poderemos inserir qualquer tipo de objeto e, para provar isto, na última janela estamos inserindo um objeto da classe `TPanel`, que possibilita alocarmos objetos em posições fixas. Estamos colocando um texto (`texto1`) e uma imagem (`gnome.png`) em coordenadas fixas dentro deste painel. Desta forma percebemos como é possível construir nossa interface por meio dos vários componentes criados, combinando criativamente uns com os outros. Veja na Figura 5.20 o resultado desse programa.

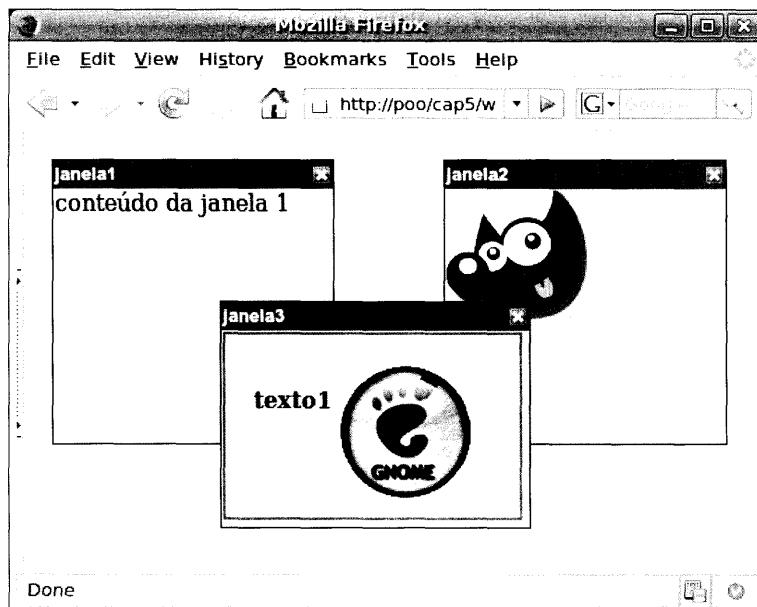


Figura 5.20 – Janelas construídas com a classe TWindow.

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();
```

```
// instancia um objeto painel  
// coloca dentro do painel um texto e uma imagem  
$painel = new TPanel(210,130);  
$painel->put(new TParagraph('<b>texto1</b>'), 20,20);  
$painel->put(new TImage('app.images/gnome.png'), 80, 20);  
  
// instancia um objeto TWindow nas coordenadas 140,120 contendo um painel  
$janela3 = new TWindow('janela3');  
$janela3->setPosition(140,120);  
$janela3->setSize(220,160);  
$janela3->add($painel);  
$janela3->show();  
?>
```

5.4 Diálogos e controles

5.4.1 Page Controller

5.4.1.1 Introdução

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 você 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 de clientes, por exemplo.

Programa	Objetivo
/clientes/listar.php	Listar registros.
/clientes/gravar.php	Inserir registros.
/clientes/editar.php	Editar registros.
/clientes/excluir.php	Excluir registros.

Não é difícil encontrar programas estruturados dessa forma, afinal, esse tipo de pensamento é natural quando evoluímos a partir de páginas estáticas em HTML. Entretanto, um servidor web em conjunto com uma linguagem de programação dinâmica, como é o PHP, pode fazer muito mais em termos de estrutura do fluxo de execução de um sistema ou de um site. Uma página em PHP é dinâmica e pode decidir sobre o seu fluxo de execução baseada em determinados parâmetros, os quais podem ser passados via URL, como no programa a seguir:

Programa	Objetivo
page.php?option=listar	Listar registros.
page.php?option=incluir	Incluir registros.
page.php?option=alterar	Alterar registros.
page.php?option=excluir	Excluir registros.

page.php

```
<?php
$option = $_GET['option'];

switch ($option)
{
    case 'listar':
        echo 'listando registro';
        break;
    case 'incluir':
        echo 'incluindo registro';
        break;
    case 'alterar':
        echo 'alterando registro';
        break;
    case 'excluir':
        echo 'excluindo registro';
        break;
}
?>
```

No programa anterior percebemos claramente uma decisão sendo tomada pelo comando `switch`, que determinará o fluxo de execução do programa baseado em um parâmetro recebido via URL (método GET). Este parâmetro se chama `$option`.

Esta decisão que tomamos no programa anterior é uma versão bastante simplificada de um controlador de página, que é um pattern também conhecido como Page Controller. Trata-se de um objeto que interpreta uma requisição para uma determinada ação dentro do sistema e decide o fluxo de execução a tomar, qual método a ser executado. A idéia principal é que cada página tenha seu próprio controlador de fluxo de execução.

5.4.1.2 Proposta

Para implementar um Page Controller criaremos a classe `TPage`. Esta classe terá como responsabilidade encapsular todo e qualquer componente visual para formar uma página web, seja ele uma tabela (`TTable`), um painel (`TPanel`), um parágrafo (`TParagraph`), uma imagem (`TImage`), ou todos eles utilizados de forma combinada. Para isso, esta

classe será filha de `TElement` e assim irá prover o método `add()`, permitindo adicionar widgets ao seu conteúdo.

A principal característica de um Page Controller é decidir sobre o fluxo de execução a ser seguido baseado em algum parâmetro. Para isso, iremos analisar a URL e extrair esta informação para então descobrir qual ação executar. Para isso, no entanto, teremos de criar um padrão de nomenclatura de métodos para utilizar em nosso sistema. Veja a seguir o padrão que iremos utilizar:

Programa	Objetivo
<code>page.php?method=listar</code>	Executa a função listar presente neste arquivo.
<code>page.php?method=incluir</code>	Executa a função incluir presente neste arquivo.

Esse padrão funciona muito bem se nosso sistema for composto de funções, mas, como estamos pensando em adotar uma estrutura totalmente orientada a objetos, teremos de criar um padrão que permita identificarmos não somente o nome de uma função, mas também o nome de um método de uma determinada classe. Para tanto, utilizaremos o seguinte padrão:

Programa	Objetivo
<code>page.php?class=Clientes&method=listar</code>	Executa o método listar da classe Clientes.
<code>page.php?class=Clientes&method=incluir</code>	Executa o método incluir da classe Clientes.

Dessa forma, cada URL identificará a classe e o método correspondente que serão executados. Esta classe não precisará necessariamente estar presente neste arquivo, uma vez que ela pode ser carregada na memória por meio de recursos como o comando `include`.

A seguir temos o código da classe `TPage`. Como vimos, ela será filha da classe `TElement` e irá representar o elemento `<html>`. Como filha de `TElement`, poderemos utilizar o método `add()` da classe-pai para acrescentar widgets em seu interior. Veja na Figura 5.21 a classe `TPage`.

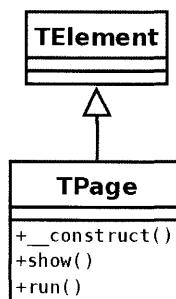


Figura 5.21 – Classe `TPage`.

 TPage.class.php

```
<?php
/**
 * classe TPage
 * classe para controle do fluxo de execução
 */
class TPage extends TElement
{
    /**
     * método __construct()
     */
    public function __construct()
    {
        // define o elemento que irá representar
        parent::__construct('html');
    }

    /**
     * método show()
     * exibe o conteúdo da página
     */
    public function show()
    {
        $this->run();
        parent::show();
    }
}
```

Aqui temos o método mais importante da classe `TPage` – o método `run()` – que é executado automaticamente dentro do método `show()` da classe. Esse método irá ler as informações vindas via URL (mais precisamente as variáveis `$class` e `$method`). A partir dessas informações, o método `run()` verificará primeiro pela existência da classe e se o referido método realmente existe (`method_exists`) para, então, verificar a existência de uma função com o mesmo nome (`function_exists`). A função ou o método a ser executado sempre receberá como parâmetro o vetor `$_GET`, que contém todos os parâmetros recebidos via URL, podendo analisar e utilizar esses parâmetros. Ainda estamos verificando se o método existe na classe atual (`get_class`), para que, caso venhamos a estender a classe `TPage`, possamos executar tal método diretamente, sem a necessidade de instanciar um novo objeto. Veja na Figura 5.22 o fluxo de interação entre o usuário e a página.

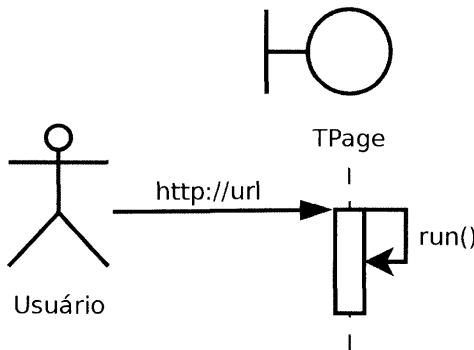


Figura 5.22 – Fluxo de interação do usuário com a página.

```

/**
 * método run()
 * executa determinado método de acordo com os parâmetros recebidos
 */
public function run()
{
    if ($_GET)
    {
        $class  = $_GET['class'];
        $method = $_GET['method'];

        if ($class)
        {
            $object = $class == get_class($this) ? $this : new $class;
            if (method_exists($object, $method))
            {
                call_user_func(array($object, $method), $_GET);
            }
        }
        else if (function_exists($method))
        {
            call_user_func($method, $_GET);
        }
    }
}
?>
  
```

5.4.1.3 Exemplos

Neste primeiro exemplo criamos a função `ola_mundo()`. Em seguida, instanciamos um objeto `TPage` e executamos o método `show()`. Este método executará internamente o

método `run()`, que irá analisar a URL e executar os métodos correspondentes. Para que a função `ola_mundo()` seja executada, será necessário que o usuário acione o endereço `page1.php?method=ola_mundo&nome=pablo`. Assim, a função `ola_mundo()` será executada e irá receber todas as variáveis da URL como parâmetro, sendo que a variável `$nome` será exibida na tela como “Olá meu amigo pablo”. Veja na Figura 5.23 o resultado desse programa.

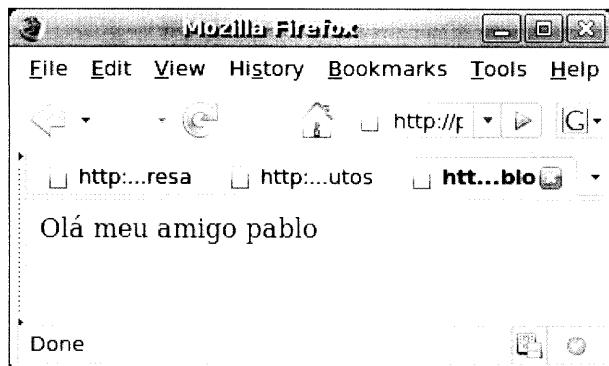


Figura 5.23 – Utilizando a classe `TPage` de forma estruturada.

page1.php

```
<?php
include_once 'app.widgets/TElement.class.php';
include_once 'app.widgets/TPage.class.php';

function ola_mundo($param)
{
    echo 'Olá meu amigo ' . $param['nome'];
}

$pagina = new TPage;
$pagina->show();
?>
```

Neste outro exemplo, em vez de acionar uma função, estaremos acionando um método de um objeto. Para isso, estamos estendendo a classe `TPage` e criando a classe-filha `Mundo`. Esta classe possui exatamente a mesma estrutura da classe `TPage`. Dentro da classe `Mundo`, temos o método `ola()`, o qual receberá o vetor `$_GET` (`$param`). Para acionar o método `ola()` da classe `Mundo`, passando “pablo” como parâmetro, seria necessário digitar o seguinte: `page2.php?class=Mundo&method=ola&nome=pablo`. Desta forma, seria impresso na página “Olá meu amigo pablo”. Veja também na Figura 5.24 o fluxo de funcionamento da página. O usuário aciona o script que executa o método `show()` da classe `Mundo`. Internamente o método `show()` aciona o método `run()` que avalia

a URL e descobre que se está querendo executar o método `ola()` da própria classe, sendo assim, simplesmente é realizada a chamada, passando o conteúdo da variável `$_GET`. Caso o nome da classe passada na URL fosse diferente da classe em execução, o programa instanciaria um objeto desta outra classe. Veja na Figura 5.24 o fluxo de execução de um método.

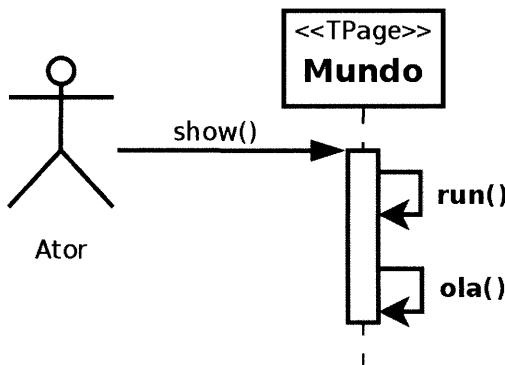


Figura 5.24 – Fluxo de execução de um método.

page2.php

```

<?php
include_once 'app.widgets/TElement.class.php';
include_once 'app.widgets/TPage.class.php';

class Mundo extends TPage
{
    function ola($param)
    {
        echo 'Olá meu amigo ' . $param['nome'];
    }
}

$pagina = new Mundo;
$pagina->show();
?>
  
```

No exemplo a seguir, estamos construindo uma página bem simples, com apenas três links (**Produtos**, **Contato** e **Empresa**). Quando o usuário clicar no link **Produtos** será executada a função `onProdutos()`; quando clicar no link **Contato** será executada a função `onContato()`; e quando clicar sobre o link **Empresa** será executada a função `onEmpresa()`. O papel de cada função será simplesmente exibir um texto diferente na tela. Observe a parte do programa responsável por exibir os links na tela, passando os nomes dos métodos a serem executados. Veja na Figura 5.25 o resultado desse programa.

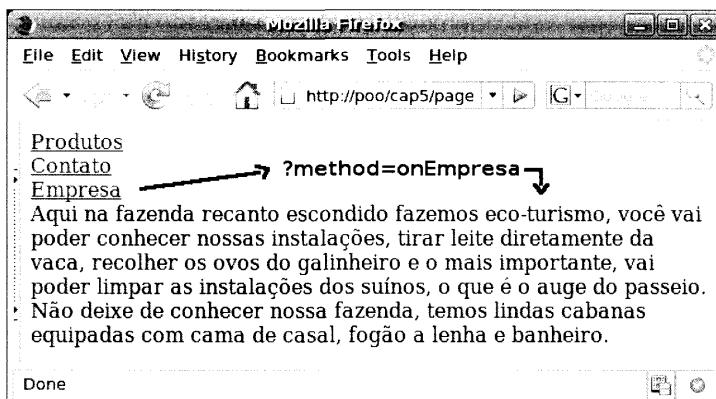


Figura 5.25 – Uma página simples construída com a classe TPage.

page3.php

```
<?php
function __autoload($classe)
{
    if (file_exists("app.widgets/{$classe}.class.php"))
    {
        include_once "app.widgets/{$classe}.class.php";
    }
}
/***
 * método onProdutos()
 * executado quando o usuário clicar no link "Produtos"
 * @param $get = variável $_GET
 */
function onProdutos($get)
{
    echo "Nesta seção você vai conhecer os produtos da nossa empresa
          Temos desde pintos, frangos, porcos, bois, vacas e todo tipo de animal
          que você pode imaginar na nossa fazenda.";
}
/***
 * método onContato()
 * executado quando o usuário clicar no link "Contato"
 * @param $get = variável $_GET
 */
function onContato($get)
{
    echo "Para entrar em contato com nossa empresa,
          ligue para 0800-1234-5678 ou mande uma carta endereçada para
          Linha alto coqueiro baixo, fazenda recanto escondido.";
}
```

```
/**  
 * método onEmpresa()  
 * executado quando o usuário clicar no link "Empresa"  
 * @param $get = variável $_GET  
 */  
function onEmpresa($get)  
{  
    echo "Aqui na fazenda recanto escondido fazemos eco-turismo,  
        você vai poder conhecer nossas instalações,  
        tirar leite diretamente da vaca, recolher os ovos do galinheiro e o mais  
        importante, vai poder limpar as instalações dos suínos, o que é o auge  
        do passeio. Não deixe de conhecer nossa fazenda, temos lindas cabanas  
        equipadas com cama de casal, fogão a lenha e banheiro.";  
}  
  
// exibe alguns links  
echo "<a href='?method=onProdutos'>Produtos</a><br>";  
echo "<a href='?method=onContato'>Contato</a><br>";  
echo "<a href='?method=onEmpresa'>Empresa</a><br>";  
  
// interpreta a página  
$pagina = new TPage;  
$pagina->show();  
?>
```

5.4.2 Ações

Quando o usuário clica no botão para editar um registro, clica na opção de um menu ou clica no botão para salvar os dados de um formulário, uma determinada ação é executada. Geralmente mapeamos a ocorrência desses eventos diretamente à execução de uma função ou um método de um objeto. No exemplo anterior, o programa construído tinha três ações (Produtos, Contato e Empresa). Veja na Figura 5.26 alguns exemplos de ações.

Em alguns ambientes, principalmente o ambiente web, que é stateless, ou seja, não armazena o estado atual de seus objetos durante a transição de páginas (a não ser que utilizemos seções), torna-se difícil mapear tais eventos diretamente aos objetos que irão receber (ou executar) a ação. Nestes casos é fundamental termos a possibilidade de transportar a ação de forma independente do objeto emissor e do objeto receptor do evento.

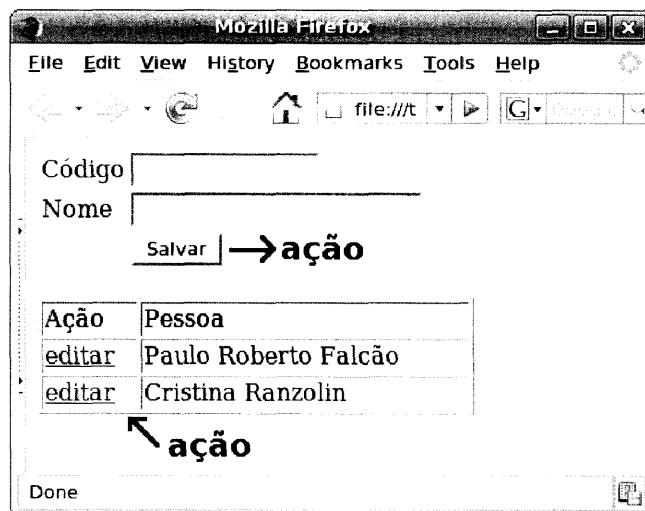
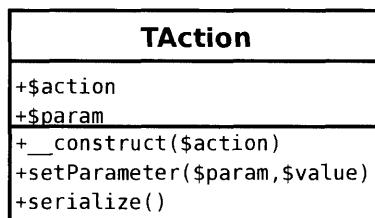


Figura 5.26 – Exemplos de ações.

Para transportar uma ação de uma página para outra estamos utilizando a própria URL, mas é necessário converter o nome de uma função ou o método de um objeto para o padrão de nomenclatura que convencionamos (`?method=função`) ou (`?class=classe&method=método`). Para isso, criaremos a classe `TAction`. Esta classe receberá em seu método construtor um parâmetro do tipo `callback`, o qual poderá ser tanto o nome de uma função existente (string) quanto um par representado por objeto e nome de método (array). Essa classe também terá o método `setParameter()` que possibilitará adicionarmos parâmetros a esta chamada de função. Veja na Figura 5.27 a classe `TAction`.

Figura 5.27 – Classe `TAction`.

`TAction.php`

```
<?php
/**
 * classe TAction
 * encapsula uma ação
 */
class TAction
{
```

```
private $action;
private $param;

/**
 * método __construct()
 * instancia uma nova ação
 * @param $action = método a ser executado
 */
public function __construct($action)
{
    $this->action = $action;
}

/**
 * método setParameter()
 * acrescenta um parâmetro ao método a ser executado
 * @param $param = nome do parâmetro
 * @param $value = valor do parâmetro
 */
public function setParameter($param, $value)
{
    $this->param[$param] = $value;
}
```

O método `serialize()` tratará da conversão da ação (propriedade `$action`) em uma string que possa ser passada via URL. Para isso, este método primeiro verifica se a ação (`$action`) se trata de um par (objeto e método) por meio da função `is_array()`. Neste caso, a URL será formada pelo nome da classe e o nome do método. Caso a ação seja uma string (`is_string`), a URL será formada somente pelo nome da função. A função `http_build_query()` do PHP trata de construir a URL de acordo com o vetor `$url`, no qual fomos montando os parâmetros do endereço.

```
/**
 * método serialize()
 * transforma a ação em uma string do tipo URL
 */
public function serialize()
{
    // verifica se a ação é um método
    if (is_array($this->action))
    {
        // obtém o nome da classe
        $url['class'] = get_class($this->action[0]);
        // obtém o nome do método
        $url['method'] = $this->action[1];
    }
}
```

```

else if (is_string($this->action)) // é uma string
{
    // obtém o nome da função
    $url['method'] = $this->action;
}

// verifica se há parâmetros
if ($this->param)
{
    $url = array_merge($url, $this->param);
}

// monta a URL
return '?' . http_build_query($url);
}
?>

```

5.4.2.1 Exemplos

No exemplo a seguir demonstramos o uso da classe `TAction`. Primeiro criamos uma ação que corresponde ao método `acao()` da classe `Receptor` e depois exibimos, por meio do método `serialize()`, esta ação devidamente convertida ao formato de URL. Em seguida, criamos uma ação que corresponde à execução da função `strtoup()` e a convertemos também ao formato de URL.

A classe `TAction` será utilizada principalmente na formação de links de listagens e também em botões de ação de formulários.

action.php

```

<?php
include_once 'app\widgets\TAction.class.php';

class Receptor
{
    function acao($parameter)
    {
        echo "Ação executada com sucesso\n<br>";
    }
}

$receptor = new Receptor;
$action1 = new TAction(array($receptor, 'acao'));
$action1->setParameter('nome', 'pablo');
echo $action1->serialize();

```

```

echo "<br>\n";
$action2 = new TAction('strtoup');
$action2->setParameter('nome', 'pablo');
echo $action2->serialize();
?>

```

 **Resultado:**

```
?class=Receptor&method=acao&nome=pablo
?method=strtoup&nome=pablo
```

No exemplo a seguir, reescreveremos uma página web contendo alguns links (**Produtos**, **Contato**, **Empresa**). No entanto, desta vez iremos utilizar a classe **TPage** da maneira que ela fora projetada, ou seja, como contêiner de um outro objeto. Para isso, iremos estendê-la, criando a classe **Pagina**. Em vez de executarmos o método **run()** da página, utilizaremos seu método **show()**, que, além de executar internamente o método **run()**, interpretando os parâmetros que vêm via URL, exibe o objeto filho, adicionado por meio do método **add()** da classe **TPage**. Veja na Figura 5.28 a estrutura de classes deste exemplo.

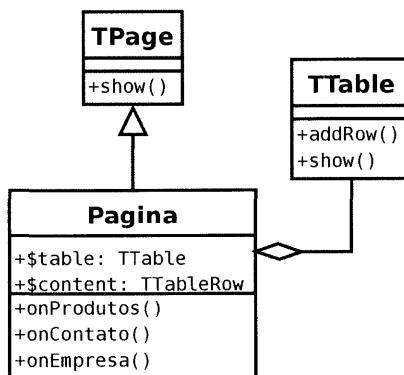


Figura 5.28 – Classe **TPage** sendo estendida.

Também aproveitaremos este novo exemplo para utilizar a classe **TAction** e criar os links desta página. Esta classe em seu método construtor cria uma tabela (objeto **TTable**) e cria uma barra de links (**Produtos**, **Contato**, **Empresa**). Cada link está vinculado com uma ação: **onProdutos()**, **onContato()** ou **onEmpresa()**. Ainda no método construtor, adicionamos uma linha chamada **\$this->content** à tabela, a qual só será preenchida com conteúdo quando um desses métodos for executado. O papel de cada método é justamente adicionar uma célula nesta tabela com o texto correspondente a cada opção. Veja na Figura 5.29 o resultado do programa.

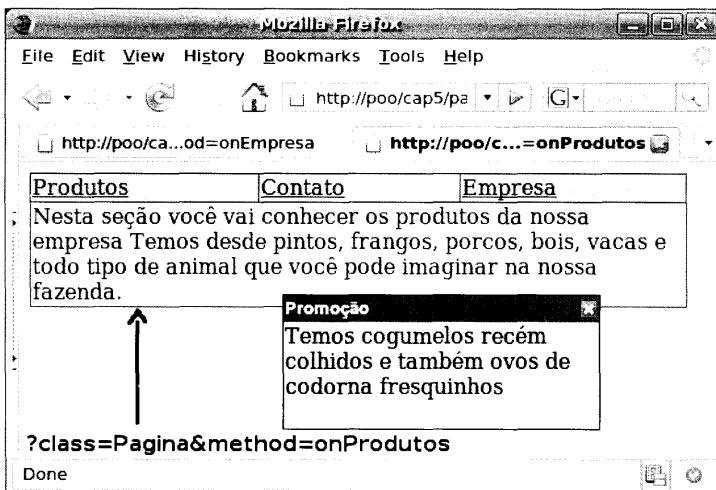


Figura 5.29 – Resultado da execução do programa a seguir.

page4.php

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

class Pagina extends TPage
{
    private $table;
    private $content;

    /**
     * método __construct()
     * instancia uma nova página
     */
    function __construct()
    {
        parent::__construct();
        // cria uma tabela
        $this->table = new TTable;

        // define algumas propriedades para tabela
        $this->table->border = 1;
```

```
$this->table->width = 500;
$this->table->style = 'border-collapse:collapse';

// adiciona uma linha na tabela
$row = $this->table->addRow();
$row->bgcolor = '#d0d0d0';

// cria três ações
$action1 = new TAction(array($this, 'onProdutos'));
$action2 = new TAction(array($this, 'onContato'));
$action3 = new TAction(array($this, 'onEmpresa'));

// cria três links
$link1 = new TElement('a');
$link2 = new TElement('a');
$link3 = new TElement('a');

// define a ação dos links
$link1->href = $action1->serialize();
$link2->href = $action2->serialize();
$link3->href = $action3->serialize();

// define o rótulo de texto dos links
$link1->add('Produtos');
$link2->add('Contato');
$link3->add('Empresa');

// adiciona os links na linha
$row->addCell($link1);
$row->addCell($link2);
$row->addCell($link3);

// cria uma linha para conteúdo
$this->content = $this->table->addRow();

// adiciona a tabela na página
parent::add($this->table);
}
```

O método `onProdutos()` adicionará uma célula contendo um texto que descreve os produtos da empresa. Além disso, ele irá instanciar um objeto `TWindow` e exibir uma janela pop-up contendo um texto promocional.

```
/**
 * método onProdutos()
 * executado quando o usuário clicar no link "Produtos"
 * @param $get = variável $_GET
 */
```

```

function onProdutos($get)
{
    $texto = "Nesta seção você vai conhecer os produtos da nossa empresa
    Temos desde pintos, frangos, porcos, bois, vacas e todo tipo de animal
    que você pode imaginar na nossa fazenda.";

    // adiciona o texto na linha de conteúdo da tabela
    $celula = $this->content->addCell($texto);
    $celula->colspan = 3;

    // cria uma janela
    $win = new TWindow('Promoção');
    // define posição e tamanho
    $win->setPosition(200,100);
    $win->setSize(240,100);

    // adiciona texto na janela
    $win->add('Temos cogumelos recém colhidos e também ovos de codorna fresquinhos');

    // exibe a janela
    $win->show();
}

```

O método `onContato()` será executado quando o usuário clicar no link **Contato** e adicionará à tabela um texto descrevendo as formas de entrar em contato com a empresa.

```

/**
 * método onContato()
 * executado quando o usuário clicar no link "Contato"
 * @param $get = variável $_GET
 */
function onContato($get)
{
    $texto = "Para entrar em contato com nossa empresa,
        ligue para 0800-1234-5678 ou mande uma carta endereçada para
        Linha alto coqueiro baixo, fazenda recanto escondido.';

    // adiciona o texto na linha de conteúdo da tabela
    $celula = $this->content->addCell($texto);
    $celula->colspan = 3;
}

```

O método `onEmpresa()` será executado quando o usuário clicar na opção **Empresa** e adicionará na tabela uma célula contendo a descrição da empresa.

```

/**
 * método onEmpresa()
 * executado quando o usuário clicar no link "Empresa"

```

```
* @param $get = variável $_GET
*/
function onEmpresa($get)
{
    $texto = "Aqui na fazenda recanto escondido fazemos eco-turismo,
    você vai poder conhecer nossas instalações,
    tirar leite diretamente da vaca, recolher os ovos do galinheiro e o mais
    importante, vai poder limpar as instalações dos suínos, o que é o auge
    do passeio. Não deixe de conhecer nossa fazenda, temos lindas cabanas
    equipadas com cama de casal, fogão a lenha e banheiro";

    // adiciona o texto na linha de conteúdo da tabela
    $celula = $this->content->addCell($texto);
    $celula->colspan = 3;
}
// instancia nova página
$pagina = new Pagina;

// exibe a página juntamente com seu conteúdo e interpreta a URL
$pagina->show();
?>
```

5.4.3 Diálogos de mensagem

Seguidamente precisamos emitir mensagens ao usuário, seja quando uma operação concluiu com sucesso, seja quando algum erro ocorreu. Pensando nisso, iremos construir a classe `TMessage`. O objetivo desta classe é exibir a mensagem que desejamos transmitir ao usuário, juntamente com uma pequena imagem ilustrativa do que está ocorrendo (informação ou erro) e um botão para fechar o diálogo. Tudo isso será exibido dentro da camada (`div`), a qual permite que a mensagem seja exibida sobre o conteúdo da página. Além disso, é possível escondê-la, alterando sua propriedade `display`, como se o usuário estivesse fechando uma janela de verdade.

Para definirmos as características da camada (`div`) que será utilizada para comportar a mensagem ao usuário, criaremos um estilo nomeado `tmassage`, o qual irá definir algumas características de posicionamento (`left, top`), tamanho (`width, height`), cor (`background`) e borda (`border`).

Tudo começa com a instância `$painei`. Este objeto (`div`) irá conter uma tabela (objeto `$table`), a qual irá conter na primeira linha uma imagem (que será `info.png` ou `error.png`), de acordo com o parâmetro `$type` passado pelo usuário, e a mensagem propriamente dita (parâmetro `$message`). Já na segunda linha, a tabela irá conter um botão (objeto `$button`) que será responsável por fechar o diálogo quando for clicado

(`style.display='none'`). Note que o parâmetro `$type` será exatamente o nome do arquivo de imagem que será exibido (exceto pela extensão).

TMessage.class.php

```
<?
/***
 * classe TMessage
 * exibe mensagens ao usuário
 */
class TMessage
{
    /**
     * método construtor
     * instancia objeto TMessage
     * @param $type    = tipo de mensagem (info, error)
     * @param $message = mensagem ao usuário
     */
    public function __construct($type, $message)
    {
        $style = new TStyle('tmessage');
        $style->position      = 'absolute';
        $style->left         = '30%';
        $style->top          = '30%';
        $style->width         = '300';
        $style->height        = '150';
        $style->color         = 'black';
        $style->background     = '#DDDDDD';
        $style->border        = '4px solid #000000';
        $style->z_index       = '1000000000000000';

        // exibe o estilo na tela
        $style->show();

        // instancia o painel para exibir o diálogo
        $painel = new TElement('div');
        $painel->class = "tmessage";
        $painel->id   = "tmessage";

        // cria um botão que vai fechar o diálogo
        $button = new TElement('input');
        $button->type = 'button';
        $button->value = 'Fechar';
        $button->onclick="document.getElementById('tmessage').style.display='none'";

        // cria um tabela para organizar o layout
        $table = new TTable;
```

```
$table->align = 'center';
// cria uma linha para o ícone e a mensagem
$row=$table->addRow();
$row->addCell(new TImage("app.images/{$type}.png"));
$row->addCell($message);

// cria uma linha para o botão
$row=$table->addRow();
$row->addCell('');
$row->addCell($button);

// adiciona a tabela ao painel
$painel->add($table);
// exibe o painel
$painel->show();
}

}

?>
```

No exemplo a seguir, estamos fazendo uso da classe `TMessage` para exibir a mensagem de informação ao usuário “Esta ação é inofensiva, isto é só um lembrete”, como visto na Figura 5.30.

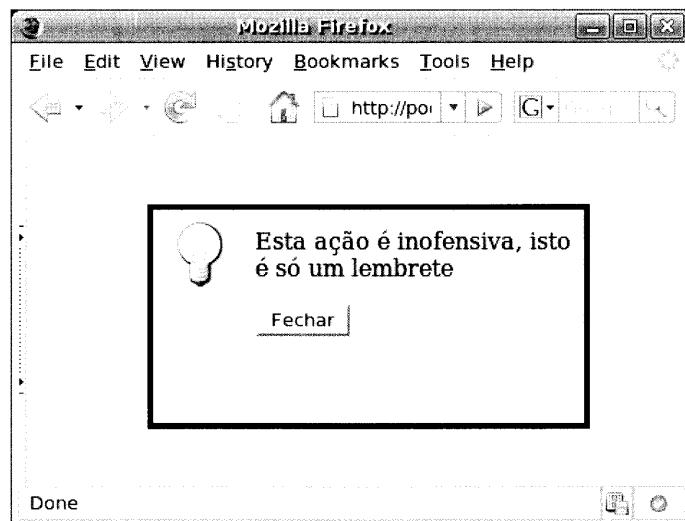


Figura 5.30 – Mensagem de informação.

✉ message_info.php

```
<?php
function __autoload($classe)
{
```

```

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.

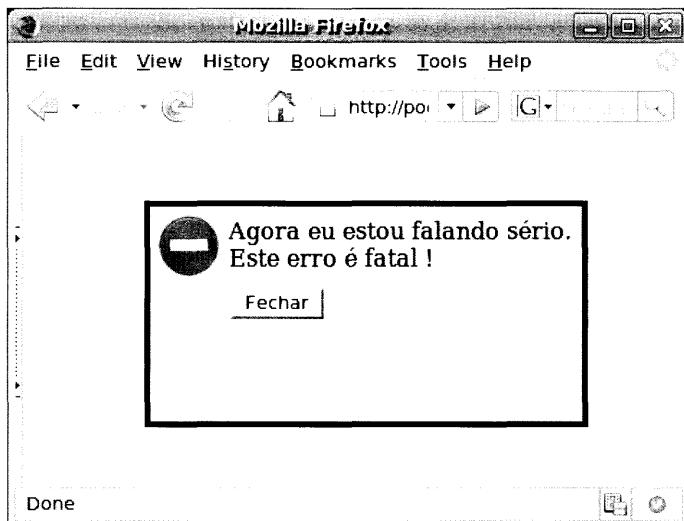


Figura 5.31 – Mensagem de erro.

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!');
?>

```

5.4.4 Diálogos de questionamento

Assim como mensagens de erro e mensagens de informação, seguidamente precisamos emitir questionamentos ao usuário, como: deseja excluir este registro? Deseja confirmar o processamento?, dentre outros. Um diálogo de questionamento é caracterizado por ter uma pergunta e ações correspondentes às respostas positiva e negativa.

O diálogo de questionamento que criaremos possuirá sempre dois botões de ação: **SIM** e **NÃO**. Para cada botão de ação, definiremos uma ação correspondente que será executada. As ações serão representadas por objetos do tipo `TAction`, os quais irão conter o método a ser executado. As ações serão traduzidas no formato de URL para que possam ser acionadas por meio de Javascript (`location=$url`).

Assim como na classe `TMessage`, a classe `TQuestion` também criará uma camada (`div`) para exibir a pergunta ao usuário. Criaremos, então, um estilo nomeado `tquestion` para definir as características dessa camada. Assim como na classe `TMessage`, teremos uma tabela, a qual irá conter na sua primeira linha um ícone para representar uma pergunta (`question.png`) e a pergunta a ser realizada. Na segunda linha teremos dois botões (elementos `input`). Cada botão terá como ação trocar a localização da página quando for clicado (`location=$url`).

TQuestion.class.php

```
<?
/**
 * classe TQuestion
 * Exibe perguntas ao usuário
 */
class TQuestion
{
    /**
     * método construtor
     * instancia objeto TQuestion
     * @param $message = pergunta ao usuário
     * @param $action_yes = ação para resposta positiva
     * @param $action_no = ação para resposta negativa
     */
    function __construct($message, TAction $action_yes, TAction $action_no)
    {
        $style = new TStyle('tquestion');
        $style->position      = 'absolute';
        $style->left         = '30%';
        $style->top          = '30%';
        $style->width         = '300';
        $style->height        = '150';
        $style->border_width  = '1px';
```

```
$style->color      = 'black';
$style->background   = '#DDDDDD';
$style->border      = '4px solid #000000';
$style->z_index     = '1000000000000000';

// converte os nomes de métodos em URL's
$url_yes = $action_yes->serialize();
$url_no  = $action_no->serialize();

// exibe o estilo na tela
$style->show();

// instancia o painel para exibir o diálogo
$painel = new TElement('div');
$painel->class = "tquestion";

// cria um botão para a resposta positiva
$button1 = new TElement('input');
$button1->type = 'button';
$button1->value = 'Sim';
$button1->onclick="javascript:location='".$url_yes"'";

// cria um botão para a resposta negativa
$button2 = new TElement('input');
$button2->type = 'button';
$button2->value = 'Não';
$button2->onclick="javascript:location='".$url_no"'";

// cria uma tabela para organizar o layout
$table = new TTable;
$table->align = 'center';
$table->cellspacing = 10;
// cria uma linha para o ícone e a mensagem
$row=$table->addRow();
$row->addCell(new TImage('app.images/question.png'));
$row->addCell($message);

// cria uma linha para os botões
$row=$table->addRow();
$row->addCell($button1);
$row->addCell($button2);

// adiciona a tabela ao painel
$painel->add($table);
// exibe o painel
$painel->show();
}

?>
```

Para demonstrar a utilização de um diálogo de questionamento, criaremos uma página contendo um painel (400 x 200). Dentro deste painel colocaremos o texto “Responda a questão”. Em seguida, faremos a pergunta “Deseja realmente excluir o registro?”. Caso o usuário responda **SIM**, o método `onYes()` será executado, exibindo “Você escolheu a opção sim” nas coordenadas 10 x 40. Caso contrário, o método `onNo()` será executado, exibindo o texto “Você escolheu a opção não”, nas mesmas coordenadas. Na Figura 5.32 temos o resultado deste programa.

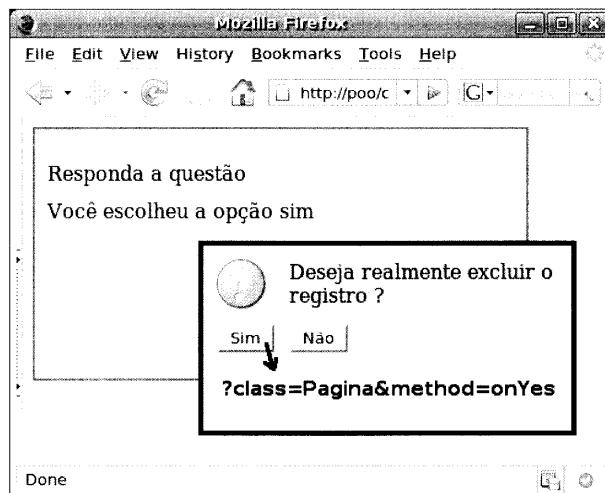


Figura 5.32 – Mensagem de questionamento.

question.php

```
<?php
function __autoload($classe)
{
    if (file_exists("app.widgets/{$classe}.class.php"))
    {
        include_once "app.widgets/{$classe}.class.php";
    }
}
/*
 * classe Pagina
 * demonstra o funcionamento de um diálogo de questão
 */
class Pagina extends TPage
{
    private $panel;
    /**
     * método construtor
     * instancia uma nova página
     */
}
```

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

    // cria um novo painel
    $this->panel = new TPanel(400,200);
    // coloca um novo texto na coluna:10 linha:10
    $this->panel->put(new TParagraph('Responda a questão'), 10, 10);

    // cria duas ações
    $action1 = new TAction(array($this, 'onYes'));
    $action2 = new TAction(array($this, 'onNo'));

    // exibe a pergunta ao usuário
    new TQuestion('Deseja realmente excluir o registro?', $action1, $action2);

    // adiciona o painel à janela
    parent::add($this->panel);
}

/**
 * método onYes
 * executado caso o usuário responda SIM para pergunta
 */
function onYes()
{
    $this->panel->put(new TParagraph('Você escolheu a opção sim'), 10, 40);
}

/**
 * método onYes
 * executado caso o usuário responda NÃO para pergunta
 */
function onNo()
{
    $this->panel->put(new TParagraph('Você escolheu a opção não'), 10, 40);
}

// instancia a página
$pagina = new Pagina;
// exibe a página
$pagina->show();
?>
```

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.

6.1.1 Elementos de um formulário

Para criarmos um formulário, utilizamos a tag `<form>`. Dentro dela podemos dispor diversos elementos, que chamamos também de widgets em linguagens visuais como o Delphi ou o Gtk. Cada campo é descrito por uma tag. A seguir, explicaremos brevemente cada um dos elementos que podem compor um formulário, tendo em vista que o objetivo principal deste livro não é explicar a linguagem HTML, para a qual já existem excelentes livros.

6.1.1.1 Formulário

Todo o formulário está contido entre as tags `<form>` e `</form>`. Destacamos a propriedade `name`, que indicará o nome do formulário, a propriedade `method`, que indicará se os dados serão submetidos por meio de um “`post`” ou um “`get`”, vistos a seguir, a propriedade `action`, que indica o nome do script que será acionado, recebendo os dados do formulário e também o tipo de codificação utilizada para o transporte dos dados (`enctype`).

```
<form name=myform method=post action="form_gravar.php" enctype='multipart/form-data'>
... elementos ...
</form>
```

A seguir, veremos quais elementos podem fazer parte de um formulário, ou seja, quais elementos poderão estar contidos entre as tags `<form>` e `</form>`.

6.1.1.2 Componentes

O input do tipo `text` é o elemento mais utilizado. Trata-se de uma caixa de digitação de texto simples, composta de uma única linha. É utilizado para digitação de números e strings.

```
<input name='nome' value='teste' type='text' size='40' maxlength=40>
```

O input do tipo `password` é utilizado para a digitação de senhas. Seu comportamento é muito parecido com o input tipo `text`. A diferença está no momento da digitação, quando são exibidos “`*`” no lugar dos caracteres inseridos pelo usuário.

```
<input name='senha' type='password' size='40'>
```

O input do tipo `checkbox` é utilizado para exibir caixas de verificação. Uma caixa de verificação possui dois estados: clicada ou não. É útil para realizar perguntas “booleanas” ao usuário.

```
<input name='tem_filhos' value='1' type='checkbox' size='40'>
<input name='tem_renda' value='1' type='checkbox' size='40' checked>
```

O input do tipo `radio` é utilizado para exibir botões de seleção exclusiva, que permitem a seleção de apenas um item dentre vários exibidos. É utilizado para permitir ao usuário selecionar uma opção a partir de um conjunto de itens.

```
<input name='sexo' value='F' type='radio'>
<input name='sexo' value='M' type='radio' checked>
```

O input do tipo `file` é utilizado para que o usuário possa selecionar um arquivo e realizar “upload” do mesmo, enviando-o para processamento do programa.

```
<input name='foto' type='file' size='40'>
```

O input do tipo `hidden` é utilizado para armazenar um campo escondido dentro do formulário. Um campo escondido não é visível ao usuário e é utilizado, na maioria das vezes, para enviar informações que serão processadas sem que o usuário tome conhecimento.

```
<input name='id' type='hidden' size='40'>
```

O input do tipo `button` é utilizado para exibir um botão na tela. Podemos definir ações para este botão usando funções javascript, por exemplo.

```
<input type='button' size='40' value='botão' onclick="alert('teste123')">
```

O input do tipo `submit` é utilizado para exibir o botão de submissão do formulário. Este botão irá submeter os dados para processamento no servidor, passando todas as informações preenchidas pelo usuário para o “script” identificado pela tag `<form>`

```
<input type='submit' value='Gravar'>
```

O input do tipo `reset` é utilizado para exibir um botão de limpar o formulário. Botões de reset trazem o formulário de volta ao seu estado inicial, ou seja, aos valores iniciais que o mesmo continha quando a página foi carregada pelo navegador.

```
<input type='reset' value='Limpar'>
```

O input do tipo `select` é utilizado para exibir listas de seleção ao usuário. Uma lista de seleção pode permitir ao usuário a seleção de apenas um item (se comportando como uma combo-box) ou de vários itens ao mesmo tempo (como um campo de múltipla seleção).

```
<select name='cor' size=4 multiple>
  <option value='red'>Vermelho</option>
  <option value='green'>Verde</option>
  <option value='blue' selected>Azul</option></select>
</select>
```

O input do tipo `textarea` disponibiliza ao usuário uma área de digitação de texto com múltiplas linhas e colunas, dentro de uma área delimitada que apresentará barras de rolagem verticais e horizontais quando necessário.

```
<textarea name='resumo' rows='7' cols='40'>digite o resumo aqui</textarea>
```

6.1.2 Exemplo de formulário

No exemplo a seguir, temos uma demonstração de praticamente todos os elementos que compõem um formulário HTML. Uma das formas mais comuns de se construir um formulário em tela é utilizando algum contêiner, o qual é responsável por distribuir os elementos de um formulário sob um certo layout.

O contêiner mais utilizado para formulários é a tabela. Utilizando uma estrutura de tabela, podemos alinhar os rótulos de texto e os campos do formulário de forma que fiquem bem dispostos visualmente.

No exemplo a seguir, temos um formulário para digitação de dados de pessoas. Nele, temos um campo de digitação de texto para o nome, um campo de senha (`password`), um grupo de radio buttons para seleção do sexo, uma combo-box para a seleção da religião, um grupo de check buttons para seleção de idiomas, uma caixa de múltipla seleção de estilo, um campo do tipo arquivo para upload da foto e um campo do tipo `textarea` para digitação do currículo, além de botões de ação para gravar (`submit`) e limpar (`reset`) e um botão personalizado que executa uma função javascript. Na Figura 6.1, temos um formulário HTML.

The screenshot shows a Mozilla Firefox browser window with a complex HTML form. The form consists of several input fields and buttons:

- Nome:** Pablo Dall Oglie
- Senha:** ****
- Sexo:** Masculino Feminino
- Religião:** Católica
- Idiomas:**
 - Inglês
 - Espanhol
 - Italiano
 - Francês
- Estilo:** Classico
Contemporâneo
Alternativo
Elegante
- Fotografia:** /home/pablo/foto.png
- Curriculo:**
digite o currículo aqui

At the bottom of the form are three buttons: **gravar**, **limpar**, and **botao**.

Figura 6.1 – Formulário HTML.

 form.html

```
<html>
<body>
<form name=myform method=post action="form_gravar.php" enctype="multipart/form-data">
<table border="1" width="400px" bgcolor="#f0f0f0" style="border-collapse:collapse">
<tr>
    <td valign=top width="200px"><b><font face="Arial">Nome</font></b></td>
    <td><input maxlength="40" name="nome" value="Pablo Dall Oglia" type="text" size="14"></td>
</tr>
<tr>
    <td valign=top width="140px"><b><font face="Arial">Senha</font></b></td>
    <td><input name="senha" value="1234" type="password" size="14"></td>
</tr>
<tr>
    <td valign=top width="140px"><b><font face="Arial">Sexo</font></b></td>
    <td>
        <input name="sexo" value="M" type="radio" checked="1">Masculino<br>
        <input name="sexo" value="F" type="radio">Feminino<br>
    </td>
</tr>
<tr>
    <td valign=top width="140px"><b><font face="Arial">Religião</font></b></td>
    <td>
        <select name="religiao">
            <option value="0">Clique Aqui</option>
            <option value="C">Católica</option>
            <option value="E">Evangélica</option>
            <option value="U">Igreja Universal</option>
        </select>
    </td>
</tr>
<tr>
    <td valign=top width="140px"><b><font face="Arial">Idiomas</font></b></td>
    <td>
        <input name="idiomas[]" value="E" type="checkbox">Inglês<br>
        <input name="idiomas[]" value="S" type="checkbox">Espanhol<br>
        <input name="idiomas[]" value="I" type="checkbox">Italiano<br>
        <input name="idiomas[]" value="F" type="checkbox">Francês<br>
    </td>
</tr>
<tr>
    <td valign=top width="140px"><b><font face="Arial">Estilo</font></b></td>
    <td>
        <select name="estilo[]" multiple>
            <option value="C">Clássico</option>
```

```
<option value="T">Contemporâneo</option>
<option value="A">Alternativo</option>
<option value="E">Elegante</option>
</select>
</td>
</tr>
<tr>
    <td valign=top width="140px"><b><font face="Arial">Fotografia</font></b></td>
    <td><input name="fotografia" type="file"></td>
</tr>
<tr>
    <td valign=top width="140px"><b><font face="Arial">Curriculo</font></b></td>
    <td><textarea name="curriculo" rows="4" cols="40">digite o currículo aqui</textarea></td>
</tr>
</table>
<input type="submit" value="gravar">
<input type="reset" value="limpar">
<input type="button" value="botao" onclick="alert('teste123')">
</form>
</body>
</html>
```

6.1.3 Método POST

Formulários podem ser submetidos para processamento por meio dos métodos **POST** ou **GET**. Tecnicamente a diferença básica entre os dois métodos é a forma como o transporte dos dados é efetuado. O método **GET** transfere os dados do formulário via URL, ao passo que o método **POST** encapsula os dados dentro do corpo da mensagem a ser transmitida, não sendo visível ao usuário.

O método **GET** pode ser utilizado sempre que o processamento do formulário for “idempotente”, ou seja, o formulário poderá ser submetido diversas vezes sendo que o mesmo resultado é obtido, ou seja, não causa efeitos colaterais. Um exemplo disto é um formulário de consultas. Se utilizarmos o método **GET**, os campos que utilizamos como filtro da consulta serão expostos na URL e poderemos inclusive enviar a URL por e-mail para um amigo e este ao acessá-la visualizaria os mesmos resultados da busca.

Em se tratando de gravar dados em bancos de dados, no qual um formulário poderá inserir ou mesmo alterar o estado de um registro do banco de dados cada vez que é acionado, causando efeitos colaterais, é recomendável utilizarmos o método **POST**, inclusive por questões de segurança, uma vez que um usuário mal-intencionado poderia passar dados inconsistentes via URL para causar algum mal-funcionamento do programa (se utilizarmos **GET**).

Como ação do formulário anterior, definimos o script `form_gravar.php`, cujo código é apresentado a seguir. O programa que receberá a ação do formulário já recebe por padrão o array `$_POST` contendo os dados do formulário. Para exibi-lo, utilizaremos a função `print_r()`. Você perceberá que os índices do array são exatamente os nomes dos campos.

No caso de arquivos enviados pelo campo `<input type="file">`, o PHP disponibiliza o array `$_FILES`. Estes arquivos são enviados para o servidor e lá recebem um nome e uma localização temporária (veja `[tmp_name]`). Demais informações como o tipo de arquivo e seu nome original também são disponibilizadas por esse array.

form_gravar.php

```
<?php
echo "<pre>";
print_r($_POST);
print_r($_FILES);
echo "</pre>";
?>

 Resultado:
Array
(
    [nome] => Pablo Dall Oglia
    [senha] => 1234
    [sexo] => M
    [religiao] => C
    [idiomas] => Array
        (
            [0] => E
            [1] => I
        )

    [estilo] => Array
        (
            [0] => T
            [1] => E
        )

    [curriculo] => digite o currículo aqui
)

Array
(
    [fotografia] => Array
        (
```

```
[name] => foto.png  
[type] => image/jpeg  
[tmp_name] => /tmp/phptlin9t  
[error] => 0  
[size] => 1722  
)  
)
```

6.2 Um formulário orientado a objetos

6.2.1 Introdução

Com o exemplo anterior, recapitulamos como se dá a montagem e o processamento de formulários da forma tradicional. Aplicações de negócio freqüentemente possuem diversas telas utilizadas para entrada de dados por meio de formulários. Ao projetar um sistema, temos de pensar no reaproveitamento de código. A abordagem tradicional vista anteriormente torna mínimo o reaproveitamento de código. Sempre que quisermos criar um novo formulário baseado em um que já existe teremos de copiar e colar seu conteúdo.

Para maximizar o reaproveitamento de código na construção de formulários, todos os aspectos, ou seja, todos os elementos que fazem parte de um formulário serão transformados em objetos. O primeiro objetivo ao fazer isto é eliminar a necessidade de escrever código HTML diretamente na aplicação, que irá trabalhar em um nível mais alto somente utilizando os componentes que iremos criar. Cada objeto terá uma interface bem delineada e utilizaremos somente esses métodos para construir o formulário.

A primeira classe que criaremos irá representar justamente o todo, ou seja, o formulário – agrupamento de elementos (campos) que possui uma ação e um nome. Então esta classe deve oferecer métodos que nos permitam definir essas características.

A partir do formulário, precisamos pensar em cada um dos elementos que ele contém. Criaremos uma classe para cada elemento que um formulário trabalha (`text`, `radio`, `checkbox`, `password`, `select` etc.). Cada elemento possui um nome, um valor e um tamanho. Aliás, todos os elementos possuem algumas características em comum, o que pode ser resumido em uma superclasse abstrata, da qual todo elemento será filho.

Os nomes dos elementos dados pelo HTML não são os mais adequados para uma aplicação orientada a objetos. Então, inspirados em outras tecnologias como Delphi, Windows Forms (.NET), Swing e AWT (Java) e GTK, criaremos uma nomenclatura diferente:

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.

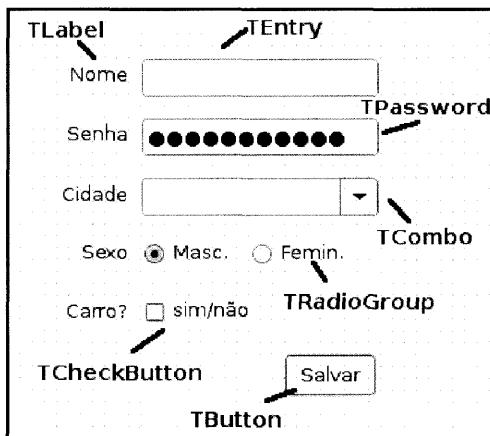


Figura 6.2 – Exemplo de formulário.

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.

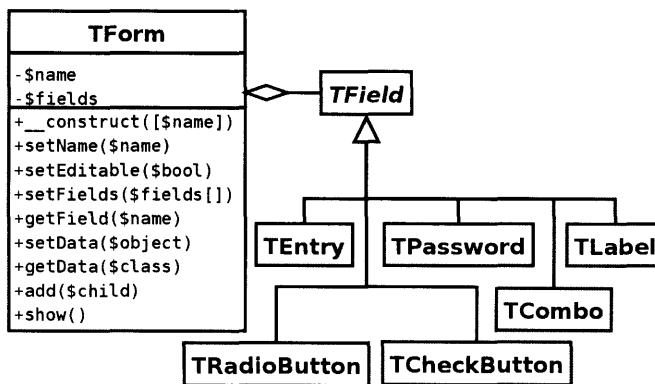


Figura 6.3 – Diagrama de classes para formulário.

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

6.2.2 Elementos de um formulário

6.2.2.1 Formulário

Para agruparmos todos os tipos de campos que vimos anteriormente (entrada, combo, label, text, radio, check, arquivo etc.), é necessário utilizar uma estrutura para encapsular o formulário, que é a própria tag `<form>` do HTML. Para isso, criaremos a classe `TForm`, cuja função será agrupar logicamente os elementos do formulário.

A nossa classe `TForm` começa em seu método construtor recebendo o nome que utilizaremos para o formulário a ser construído. Caso não especifiquemos um nome, automaticamente o nome `my_form` será utilizado. Também teremos o método `setName()` que será utilizado para alterar esse nome de formulário. Poderemos alterar o nome quantas vezes quisermos antes de executar o método `show()`.

TForm.class.php

```

<?php
/**
 * classe TForm
 * classe para construção de formulários
 */
class TForm
{
    protected $fields;      // array de objetos contidos pelo form
    private  $name;         // nome do formulário
    /**
     * método construtor
  
```

```
* instancia o formulário
* @param $name = nome do formulário
*/
public function __construct($name = 'my_form')
{
    $this->setName($name);
}
/**
 * método setName()
 * define o nome do formulário
 * @param $name      = nome do formulário
 */
public function setName($name)
{
    $this->name = $name;
}
```

O método `setEditable()` será utilizado para indicar se o formulário pode ou não ser editado. O parâmetro que utilizarmos aqui será propagado para cada um dos campos do formulário, ou seja, iremos iterar cada um dos campos (vetor `$fields`) e, para cada campo do formulário, chamaremos seu método `setEditable()` correspondente, passando o parâmetro `$bool`. Dessa forma, fica claro que todos os elementos derivados da classe `TField` deverão prover este método.

```
/**
 * método setEditable()
 * define se o formulário poderá ser editado
 * @param $bool = TRUE ou FALSE
 */
public function setEditable($bool)
{
    if ($this->fields)
    {
        foreach ($this->fields as $object)
        {
            $object->setEditable($bool);
        }
    }
}
```

O método `setFields()` será talvez o mais importante da classe `TForm` e será utilizado para atribuir os campos de um formulário. O parâmetro desta classe será representado por um vetor contendo uma série de objetos do tipo `TField`, ou seja, campos para o formulário. Estes objetos serão armazenados na propriedade `$fields` do objeto formulário. O(s) objeto(s) somente é agregado ao formulário se for descendente da classe `TField`. Ainda, se for uma instância de botão, executamos seu método `setFormName()`,

identificando o nome do formulário atual. Na classe `TButton` veremos que o botão precisa saber o nome do formulário no qual está inserido.

```
/**
 * método setFields()
 * define quais são os campos do formulário
 * @param $fields = array de objetos TField
 */
public function setFields($fields)
{
    foreach ($fields as $field)
    {
        if ($field instanceof TField)
        {
            $name = $field->getName();
            $this->fields[$name] = $field;

            if ($field instanceof TButton)
            {
                $field->setFormName($this->name);
            }
        }
    }
}

/**
 * método getField()
 * retorna um campo do formulário por seu nome
 * @param $name      = nome do campo
 */
public function getField($name)
{
    return $this->fields[$name];
}
```

O método `setData()` será utilizado para atribuir dados ao formulário. Ele deve sempre ser utilizado antes do método `show()`. Seu objetivo é passar como parâmetro um objeto qualquer, cujas propriedades são utilizadas para preencher cada um dos campos do formulário. Para isso, é necessário que os nomes das propriedades coincidam com os nomes dos campos do mesmo. Por exemplo, se passarmos um objeto `$pessoa` cuja propriedade `$nome` contenha “Maria” e a propriedade `$endereco` contenha “Rua Conceicao”, esse método preencherá o campo `nome` do formulário justamente com o valor “Maria” e o campo `endereco` com o valor “Rua da Conceicao”.

```
/**
 * método setData()
 * atribui dados aos campos do formulário
```

```
* @param $object = objeto com dados
*/
public function setData($object)
{
    foreach ($this->fields as $name => $field)
    {
        if ($name) // labels não possuem nome
        {
            $field->setValue($object->$name);
        }
    }
}
```

O método `getData()`, por sua vez, é utilizado para retornar os dados de um formulário depois de o mesmo ser processado. Quando um formulário é processado, seus dados são enviados pelo método `POST` e disponibilizados ao programador pelo vetor `$_POST`, ao passo que os arquivos são disponibilizados pelo vetor `$_FILES`.

O que o método `getData()` faz é percorrer esse vetor e disponibilizar ao usuário um objeto contendo em cada uma de suas propriedades exatamente os valores que foram enviados pelo formulário. Dessa forma, o nome das propriedades (atributos) deverá coincidir com os nomes dos campos do formulário. Se o usuário preencheu “Rua General Netto” no campo `rua`, este objeto retornado irá conter a propriedade `$rua`, exatamente com este valor. A princípio, o objeto retornado será da classe `StdClass`, mas o programador poderá indicar uma outra classe por meio do parâmetro. O objetivo disso é que, posteriormente, possamos retornar os dados do formulário diretamente em instâncias da classe `TRecord`, ou seja, objetos Active Record que poderão ser persistidos.

```
/**
 * método getData()
 * retorna os dados do formulário em forma de objeto
 */
public function getData($class = 'StdClass')
{
    $object = new $class;
    foreach ($_POST as $key=>$val)
    {
        if (get_class($this->fields[$key]) == 'TCombo')
        {
            if ($val !== '0')
            {
                $object->$key = $val;
            }
        }
    }
    else
```

```

    {
        $object->$key = $val;
    }
}

// percorre os arquivos de upload
foreach ($_FILES as $key => $content)
{
    $object->$key = $content['tmp_name'];
}
return $object;
}

```

O método `add()` será utilizado para adicionar um objeto contêiner ao formulário, o qual aceitará somente um objeto desse tipo. Geralmente adicionaremos ao formulário uma tabela (`TTable`) ou um painel (`TPanel`), como estudaremos em um dos exemplos a seguir. O objetivo desses objetos é o de organizar e dispor os elementos do formulário (`$fields`). Assim, deveremos adicionar os elementos do formulário em tabelas ou painéis e então adicionar estes ao formulário. Dentro deste objeto, os campos serão dispostos seguindo um determinado padrão de layout. O método `show()` será responsável por exibir o formulário na tela (tag `<form>`) juntamente com seu conteúdo (`$child`).

```

/**
 * método add()
 * adiciona um objeto no formulário
 * @param $object = objeto a ser adicionado
 */
public function add($object)
{
    $this->child = $object;
}

/**
 * método show()
 * Exibe o formulário na tela
 */
public function show()
{
    // instancia TAG de formulário
    $tag = new TElement('form');
    $tag->name = $this->name; // nome do formulário
    $tag->method = 'post'; // método de transferência
    // adiciona o objeto filho ao formulário
    $tag->add($this->child);
    // exibe o formulário
    $tag->show();
}
?>

```

6.2.2.2 TField

Existem algumas características que estão presentes em todos os campos de entrada de dados em um formulário. Para não escrevermos tais características em cada uma das classes, iremos primeiramente construir a classe-base `TField`, a qual irá prover esta estrutura, ou seja, irá prover comportamento a todos os outros tipos de campos. Podemos dizer que todo e qualquer campo de um formulário de entrada de dados possuirá um nome, um tamanho, um valor, assim como poderá ser editado ou não. Dessa forma, a classe-base irá prover métodos como `setName()` e `getName()` para manipular a propriedade `name`; `setSize()` para definir o tamanho; `setValue()` e `getValue()` para manipular a propriedade `value`; e `setEditable()` e `getEditable()` para indicar se o campo poderá ou não ser editado.

A classe `TField` terá um objeto agregado – um objeto `TElement` (`$tag`). Este objeto será utilizado para exibir o campo em tela. Como a maioria dos campos de um formulário são campos do tipo `<input>`, assim será o objeto `TElement` instanciado pelo método construtor. O método `setProperty()` será utilizado para definir outras características da tag (elemento) do campo, como por exemplo um atributo `onBlur`.

Os campos de um formulário seguem um determinado padrão de exibição definido pelo navegador. Para melhorar, criaremos dois estilos: o primeiro, `tfield`, será utilizado para definir as características de exibição de um campo e definirá alguns atributos como cor e largura da borda. O estilo `tfield_disabled`, por sua vez, será utilizado para definir as características de exibição de um campo quando estiver no modo não-editável, definido pelo método `setEditable(FALSE)`. Neste caso o campo ficará em tons de cinza. Na Figura 6.4 temos a classe `TField`.

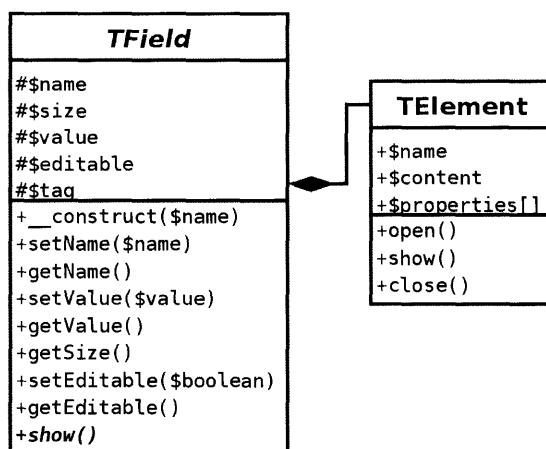


Figura 6.4 – Classe `TField`.

 **TField.class.php**

```
<?php
/**
 * classe TField
 * classe base para construção dos widgets para formulários
 */
abstract class TField
{
    protected $name;
    protected $size;
    protected $value;
    protected $editable;
    protected $tag;

    /**
     * método construtor
     * instancia um campo do formulário
     * @param $name = nome do campo
     */
    public function __construct($name)
    {
        // define algumas características iniciais
        self::setEditable(true);
        self::setName($name);
        self::setSize(200);

        // Instancia um estilo CSS chamado tfield
        // que será utilizado pelos campos do formulário
        $style1 = new TStyle('tfeld');
        $style1->border      = 'solid';
        $style1-> border_color = '#a0a0a0';
        $style1-> border_width = '1px';
        $style1->z_index     = '1';

        $style2 = new TStyle('tfeld_disabled');
        $style2-> border      = 'solid';
        $style2-> border_color = '#a0a0a0';
        $style2-> border_width = '1px';
        $style2-> background_color= '#e0e0e0';
        $style2-> color        = '#a0a0a0';

        $style1->show();
        $style2->show();

        // cria uma tag HTML do tipo <input>
        $this->tag = new TElement('input');
```

```
    $this->tag->class = 'tfield'; // classe CSS
}

/***
 * método setName()
 * define o nome do widget
 * @param $name      = nome do widget
 */
public function setName($name)
{
    $this->name = $name;
}

/***
 * método getName()
 * retorna o nome do widget
 */
public function getName()
{
    return $this->name;
}

/***
 * método setValue()
 * define o valor de um campo
 * @param $value     = valor do campo
 */
public function setValue($value)
{
    $this->value = $value;
}

/***
 * método getValue()
 * retorna o valor de um campo
 */
public function getValue()
{
    return $this->value;
}

/***
 * método setEditable()
 * define se o campo poderá ser editado
 * @param $editable = TRUE ou FALSE
 */

```

```
public function setEditable($editable)
{
    $this->editable= $editable;
}

/**
 * método getEditable()
 * retorna o valor da propriedade $editable
 */
public function getEditable()
{
    return $this->editable;
}

/**
 * método setProperty()
 * define uma propriedade para o campo
 * @param $name = nome da propriedade
 * @param $valor = valor da propriedade
 */
public function setProperty($name, $value)
{
    // define uma propriedade de $this->tag
    $this->tag->$name = $value;
}

/**
 * método setSize()
 * define a largura do widget
 * @param $size = tamanho em pixels
 */
public function setSize($size)
{
    $this->size = $size;
}
?>
```

6.2.2.3 TLabel

A classe `TLabel` será responsável por exibir um texto na tela. Em seu método construtor receberá o conteúdo deste label. No método construtor também definirá algumas características iniciais como tamanho da fonte (14), família (Arial) e cor (black). A classe `TLabel` irá prover o método `setFontSize()` que receberá um número, representando o tamanho da fonte, o método `setFontFace()` que receberá o nome da fonte e o método `setFontColor()` que irá receber o nome da cor, ou a mesma no formato RGB. Tais méto-

dos juntos definem as características da fonte que será utilizada para decorar o texto do label quando o método `show()` for executado, exibindo o texto na tela por meio do objeto `$tag` agregado de `TField`. Na Figura 6.5 temos a classe `TLabel`.

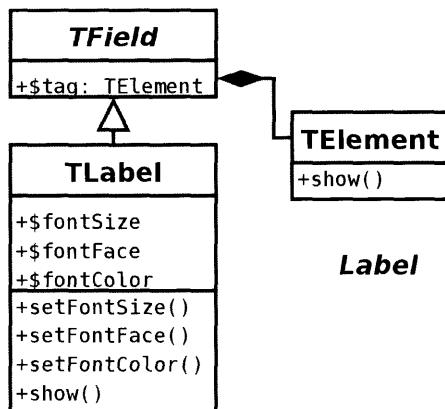


Figura 6.5 – Classe `TLabel`.

`TLabel.class.php`

```

<?php
/**
 * classe TLabel
 * classe para construção de rótulos de texto
 */
class TLabel extends TField
{
    private $fontSize; // tamanho da fonte
    private $fontFace; // nome da fonte
    private $fontColor; // cor da fonte

    /**
     * método construtor
     * instancia o label, cria um objeto <font>
     * @param $value = Texto do Label
     */
    public function __construct($value)
    {
        // atribui o conteúdo do label
        $this->setValue($value);

        // instancia um elemento <font>
        $this->tag = new TElement('font');
    }
}
  
```

```
// define valores iniciais às propriedades
$this->fontSize = '14';
$this->fontFace = 'Arial';
$this->fontColor = 'black';
}

/***
 * método setSize
 * define o tamanho da fonte
 * @param $size = tamanho da fonte
 */
public function setSize($size)
{
    $this->fontSize = $size;
}

/***
 * método setFontFace
 * define a família da fonte
 * @param $font = nome da fonte
 */
public function setFontFace($font)
{
    $this->fontFace = $font;
}

/***
 * método setFontColor
 * define a cor da fonte
 * @param $color = cor da fonte
 */
public function setFontColor($color)
{
    $this->fontColor = $color;
}

/***
 * método show()
 * exibe o widget na tela
 */
public function show()
{
    // define o estilo da tag
    $this->tag->style = "font-family:{$this->fontFace}; ";
    "color:{$this->fontColor}; ";
    "font-size:{$this->fontSize}";
```

```

    // adiciona o conteúdo à tag
    $this->tag->add($this->value);
    // exibe a tag
    $this->tag->show();
}
}

?>

```

6.2.2.4 TEntry

A classe `TEntry` será responsável por exibir em tela um campo de entrada de dados para digitação de dados. Essa classe possui todos os métodos de sua superclasse `TField`. O método `show()` é responsável por exibir o campo na tela. Para isso, ele se utiliza da classe `TElement`, criada no Capítulo 5, para construção da tag `<input>`. Em razão da facilidade da utilização da classe `TElement`, só precisamos definir seus atributos antes de utilizar seu método `show()`. O objeto `$this->tag` é criado pelo método construtor da classe-pai `TField`, logo ele sempre estará disponível. Na Figura 6.6 temos a classe `TEntry`.

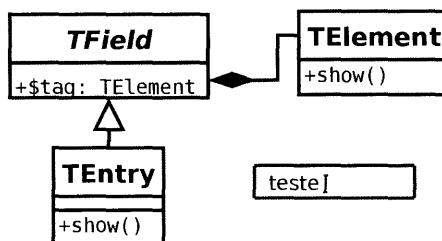


Figura 6.6 – Classe `TEntry`.

Veja que definimos o nome do campo e seu valor baseados nas propriedades `$name` e `$value` definidas respectivamente pelos métodos `setName()` e `setValue()` da classe-pai. Definimos também o tipo do input (`text`) e seu estilo (`style`). A seguir, por meio de um IF, verificamos o valor da propriedade `$editable`. Caso o campo não possa ser editado, ele será definido como somente leitura (`readonly="1"`) e também alteraremos sua classe CSS para `tfied_disabled`, a qual exibirá o elemento em tons de cinza.

TEntry.class.php

```

<?php
/**
 * classe TEntry
 * classe para construção de caixas de texto
 */
class TEntry extends TFiel
{

```

```

/*
 * método show()
 * exibe o widget na tela
 */
public function show()
{
    // atribui as propriedades da TAG
    $this->tag->name = $this->name;      // nome da TAG
    $this->tag->value = $this->value;     // valor da TAG
    $this->tag->type = 'text';           // tipo de input
    $this->tag->style = "width:{$this->size}"; // tamanho em pixels

    // se o campo não é editável
    if (!parent::getEditable())
    {
        $this->tag->readonly = "1";
        $this->tag->class = 'tfield_disabled'; // classe CSS
    }

    // exibe a tag
    $this->tag->show();
}
?>

```

6.2.3 Disposição e layout

Como vimos anteriormente, um formulário em si não possui meios de organizar os elementos nele contidos, ou seja, não possui estrutura própria para dispor os componentes na tela. O formulário é um agrupamento lógico de elementos. Quando o formulário for submetido, aqueles elementos terão seus dados enviados para o programa de destino. Como o formulário não possui meios de organizar visualmente seus elementos, precisamos confiar esta tarefa a outro elemento.

6.2.3.1 Visual de tabela

A estrutura mais utilizada para organizar os campos e os rótulos de texto de um formulário é a tabela, que permite dispor estes elementos em linhas e colunas. Construímos a classe `TForm` com o método `add()`. Por meio deste método `add()` podemos adicionar qualquer contêiner ao formulário. Como já temos a classe `TTable`, instanciaremos um objeto dela e adicionaremos este objeto ao formulário. Os elementos do formulário (`TEntry`, `TCombo` etc.) serão colocados dentro desta `TTable`. Veja a seguir um gráfico que procura demonstrar como fica a distribuição dos elementos em um visual de tabela. Note que eles são contidos ao longo de suas linhas e colunas.

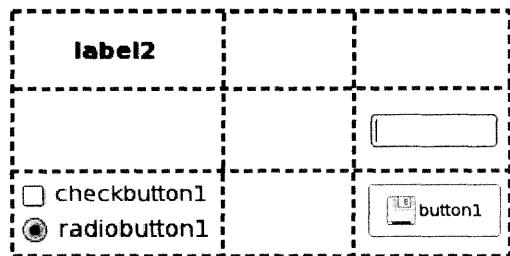


Figura 6.7 – Visual de tabela.

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.

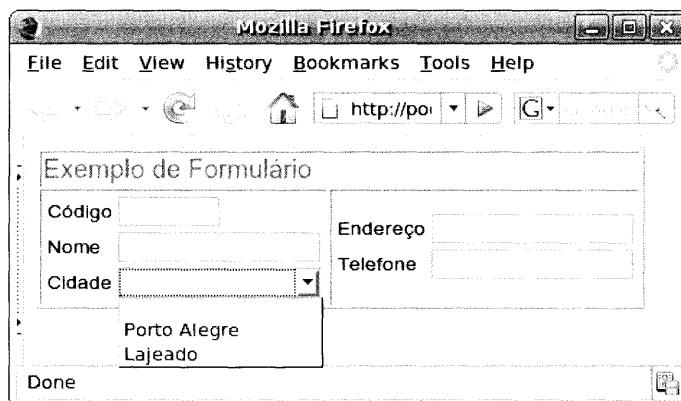


Figura 6.8 – Formulário com tabela.

formA.php

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

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

// cria o formulário
$form = new TForm('form_pessoas');
// cria a tabela para organizar o layout
$table = new TTable;
$table->border = 1;
$table->bgcolor ='#f2f2f2';

// adiciona a tabela no formulário
$form->add($table);

// cria um rótulo de texto para o título
$titulo = new TLabel('Exemplo de Formulário');
$titulo->setFontFace('Arial');
$titulo->setFontColor('red');
$titulo->setFontSize(18);
// adiciona uma linha à tabela
$row=$table->addRow();
$titulo = $row->addCell($titulo);
$titulo->colspan = 2;

// cria duas outras tabelas
$table1 = new TTable;
$table2 = new TTable;

// cria uma série de campos de entrada de dados
$codigo      = new TEntry('codigo');
$nome        = new TEntry('nome');
$endereco    = new TEntry('endereco');
$telefone    = new TEntry('telefone');
$cidade      = new TCombo('cidade');
$itemss= array();
$itemss['1'] = 'Porto Alegre';
$itemss['2'] = 'Lajeado';
$cidade->addItems($itemss);

// ajusta o tamanho destes campos
$codigo->setSize(70);
$nome->setSize(140);
$endereco->setSize(140);
$telefone->setSize(140);
$cidade->setSize(140);
```

```
// cria uma série de rótulos de texto
$label1=new TLabel('Código');
$label2=new TLabel('Nome');
$label3=new TLabel('Cidade');
$label4=new TLabel('Endereço');
$label5=new TLabel('Telefone');

// adiciona linha na tabela para o código
$row=$table1->addRow();
$row->addCell($label1);
$row->addCell($codigo);

// adiciona linha na tabela para o nome
$row=$table1->addRow();
$row->addCell($label2);
$row->addCell($nome);

// adiciona linha na tabela para a cidade
$row=$table1->addRow();
$row->addCell($label3);
$row->addCell($cidade);

// adiciona linha na tabela para o endereço
$row=$table2->addRow();
$row->addCell($label4);
$row->addCell($endereco);

// adiciona linha na tabela para o telefone
$row=$table2->addRow();
$row->addCell($label5);
$row->addCell($telefone);

// adiciona as tabelas lado-a-lado da tabela principal
$row=$table->addRow();
$row->addCell($table1);
$row->addCell($table2);

// exibe o formulário
$form->show();
?>
```

6.2.3.2 Coordenadas absolutas

No exemplo anterior, vimos como organizar o visual de um formulário pela utilização do componente `TTable`, que dispõe os elementos em linhas e colunas. Às vezes precisamos de uma flexibilidade maior para construção do formulário, uma liberdade de posicionamento de elementos que a tabela não nos dá, justamente porque

trabalha com linhas e colunas. Pensando nisto, construímos o componente `TPanel` no Capítulo 5, que permite dispor os elementos em coordenadas absolutas (linha e coluna) definidas em pixels. Uma vantagem do componente `TPanel` é justamente a liberdade de posicionamento dos componentes. No programa que será desenvolvido a seguir, faremos uso do componente `TPanel` dentro de um formulário para organizar os componentes do mesmo. Veja a seguir um gráfico que representa a idéia central da utilização de coordenadas absolutas na organização de objetos.

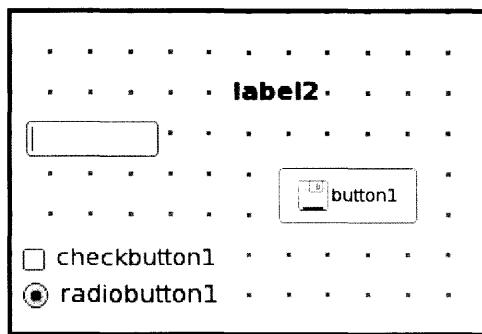


Figura 6.9 – Visual de grade.

Veja a seguir o resultado deste programa. Nele procuramos utilizar os mesmos objetos do programa anterior e organizá-los de maneira similar. Em vez de colocar os objetos dentro de células da tabela, iremos colocá-los no painel por meio de seu método `put()`, especificando as coordenadas de coluna e linha em pixels.

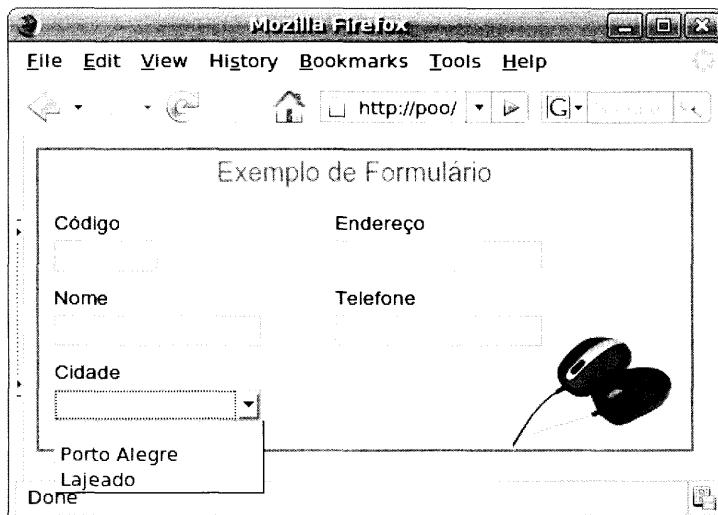


Figura 6.10 – Formulário com posições fixas.

 formB.php

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

// cria um painel
$panel = new TPanel(440,200);

// adiciona o painel ao formulário
$form->add($panel);

// cria um rótulo para o título
$titulo = new TLabel('Exemplo de Formulário');
$titulo->setFontFace('Arial');
$titulo->setFontColor('red');
$titulo->setFontSize(18);

// posiciona o título no painel
$panel->put($titulo, 120, 4);

$imagem = new TImage('app.images/mouse.png');
// posiciona a imagem no painel
$panel->put($imagem, 320, 120);

// cria os campos do formulário
$codigo      = new TEntry('codigo');
$nome        = new TEntry('nome');
$endereco   = new TEntry('endereco');
$telefone   = new TEntry('telefone');
$cidade     = new TCombo('cidade');

$itemss= array();
$itemss['1'] = 'Porto Alegre';
$itemss['2'] = 'Lajeado';
```

```
// adiciona as opções na combo
$cidade->addItems($items);

// ajusta o tamanho destes campos
$codigo->setSize(70);
$nome->setSize(140);
$endereco->setSize(140);
$telefone->setSize(140);
$cidade->setSize(140);

// cria os rótulos de texto
$label1=new TLabel('Código');
$label2=new TLabel('Nome');
$label3=new TLabel('Cidade');
$label4=new TLabel('Endereço');
$label5=new TLabel('Telefone');

// posiciona os campos e os rótulos dentro do painel
$panel->put($label1, 10, 40);
$panel->put($codigo, 10, 60);
$panel->put($label2, 10, 90);
$panel->put($nome, 10, 110);
$panel->put($label3, 10, 140);
$panel->put($cidade, 10, 160);
$panel->put($label4, 200, 40);
$panel->put($endereco, 200, 60);
$panel->put($label5, 200, 90);
$panel->put($telefone, 200, 110);

// exibe o formulário
$form->show();
?>
```

6.2.4 Outros componentes

6.2.4.1 TPassword

A classe `TPassword` será responsável por exibir em tela um campo de entrada de dados para digitação de senhas. A classe `TPassword` é praticamente igual a classe `TEntry`. A única diferença entre as duas é o tipo do `<input>` que neste caso é `password`. Um campo de senha é tratado da mesma forma que um campo de entrada de dados (`TEntry`) no processamento do formulário, a única diferença será visual. No momento em que o usuário digitar valores dentro de um campo senha, serão exibidos asteriscos no lugar dos caracteres reais. Veja na Figura 6.11 a classe `TPassword`.

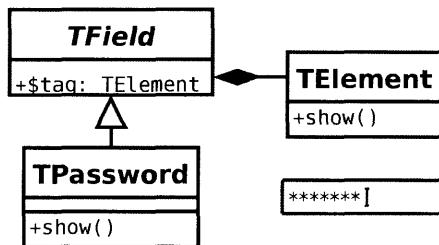


Figura 6.11 – Classe TPassword.

A classe `TPassword` também possui todos os métodos de sua superclasse `TField`. O método `show()` é responsável por exibir o campo na tela. Para isso, ele também se utiliza da classe `TElement` para construção da tag `<input>` e posterior exibição por meio de seu método `show()`. Assim como na classe `TEEntry`, verificamos se o campo pode ser editado pelo método `getEditable()`.

TPassword.class.php

```

<?php
/**
 * classe TPassword
 * classe para construção de campos de digitação de senhas
 */
class TPassword extends TField
{
    /**
     * método show()
     * exibe o widget na tela
     */
    public function show()
    {
        // atribui as propriedades da TAG
        $this->tag->name = $this->name; // nome da TAG
        $this->tag->value = $this->value; // valor da TAG
        $this->tag->type = 'password'; // tipo do input
        $this->tag->style = "width:{$this->size}"; // tamanho em pixels

        // se o campo não é editável
        if (!parent::getEditable())
        {
            $this->tag->readonly = "1";
            $this->tag->class = 'tfield_disabled'; // classe CSS
        }
        // exibe a tag
        $this->tag->show();
    }
}
?>

```

6.2.4.2 *TFile*

A classe *TFile* será responsável por exibir na tela um componente para envio de arquivos. Este componente permitirá ao usuário selecionar um arquivo por meio de um botão que abre um diálogo de seleção de arquivos. Quando o formulário for processado, este arquivo será enviado ao servidor para processamento e ficará disponível ao programador em alguma pasta temporária, provavelmente em */tmp*. Neste momento, o programador geralmente move o arquivo para alguma pasta definitiva. Veja na Figura 6.12 a classe *TFile*.

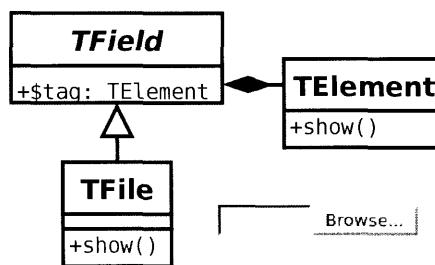


Figura 6.12 – Classe *TFile*.

Note que uma das poucas diferenças entre a classe *TFile* e as anteriores, *TEntry* e *TPassword*, é novamente o tipo de campo (*type*), que neste caso é *file*. Como a classe *TFile* também é filha de *TField*, ela possui todos os métodos desta (construtor, *setName()*, *getName()*, *setEditable()*, dentre outros).

TFile.class.php

```

<?php
/**
 * classe TFile
 * classe para construção de botões de seleção de arquivos
 */
class TFile extends TField
{
    /**
     * método show()
     * exibe o widget na tela
     */
    public function show()
    {
        // atribui as propriedades da TAG
        $this->tag->name = $this->name;      // nome da TAG
        $this->tag->value = $this->value;      // valor da TAG
        $this->tag->type = 'file';           // tipo de input
    }
}
  
```

```
// se o campo não é editável
if (!parent::getEditable())
{
    // desabilita a TAG input
    $this->tag->readonly = "1";
    $this->tag->class = 'tfield_disabled'; // classe CSS
}

// exibe a tag
$this->tag->show();
}

?>
```

6.2.4.3 THidden

A classe `THidden` será responsável pela construção de campos escondidos, invisíveis ao usuário. Apesar disto, eles fazem parte do formulário, podem conter valores e são processados assim como os demais campos do formulário, tendo seus valores enviados ao servidor quando da submissão do formulário.

Campos escondidos são utilizados para armazenar informações de status e flags a serem enviadas para processamento quando não é necessário que o usuário tenha ciência. Um exemplo de utilização é o armazenamento do ID do registro. Um usuário não precisa ver na tela o ID do registro sendo editado do banco de dados, embora seja imprescindível ter este campo ID no formulário com o real valor para que o script, ao processar estes valores, saiba se o formulário a ser editado contém um registro já existente no banco de dados que deve ser alterado ou contém um registro totalmente novo, que deve ser incluído.

Veja que a única diferença desta classe para as anteriores é também o tipo (`type`) do input, que neste caso é `hidden`.

THidden.class.php

```
<?php
/**
 * classe THidden
 * classe para construção de campos escondidos
 */
class THidden extends TField
{
    /**
     * método show()
     * exibe o widget na tela
     */
}
```

```

public function show()
{
    // atribui as propriedades da TAG
    $this->tag->name = $this->name;      // nome da TAG
    $this->tag->value = $this->value;     // valor da TAG
    $this->tag->type = 'hidden';         // tipo de input
    $this->tag->style = "width:{$this->size}"; // tamanho em pixels

    // exibe a tag
    $this->tag->show();
}

?>

```

6.2.4.4 TText

A classe `TText` será responsável pela construção de um objeto do tipo `textarea`. Um objeto `text area` disponibiliza uma área de digitação de texto, na qual o usuário poderá entrar com várias linhas de texto, diferente do componente `TEntry`, no qual o mesmo poderá entrar com somente uma linha de dados.

Veja que nesta classe estamos redefinindo o método construtor da classe `TField`, isto porque naquela estamos construindo um objeto do tipo `<input>`, e o objeto `TText` utiliza de um objeto do tipo `<textarea>`. A classe `TText` terá também o método `setSize()` que permitirá definir a largura e a altura de um campo em pixels. Veja na Figura 6.13 a classe `TText`.

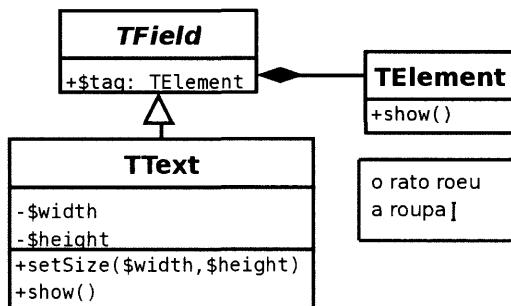


Figura 6.13 – Classe `TText`.

No método `show()` a tag `<textarea>` será exibida em tela. Note que seu funcionamento continua muito similar ao das classes anteriormente criadas. Antes de exibir a tag na tela, utilizamos o método `add()` para adicionar o texto (seu conteúdo). O conteúdo da tag passa pela função `htmlspecialchars()` para que caracteres especiais como “`<`” e “`>`” sejam corretamente convertidos para seus respectivos códigos HTML `<` e `>`;

 TText.class.php

```
<?php
/**
 * classe TText
 * classe para construção de caixas de texto
 */
class TText extends TField
{
    private $width;
    private $height;

    /**
     * método construtor
     * instancia um novo objeto
     * @param $name = nome do campo
     */
    public function __construct($name)
    {
        // executa o método construtor da classe-pai.
        parent::__construct($name);

        // cria uma tag HTML do tipo <textarea >
        $this->tag = new TElement('textarea');
        $this->tag->class = 'tfield';           // classe CSS
        // define a altura padrão da caixa de texto
        $this->height= 100;
    }
    /**
     * método setSize()
     * define o tamanho de um campo de texto
     * @param $width  = largura
     * @param $height = altura
     */
    public function setSize($width, $height)
    {
        $this->size  = $width;
        $this->height = $height;
    }
    /* método show()
     * exibe o widget na tela
     */
    public function show()
    {
        $this->tag->name = $this->name;      // nome da TAG
        $this->tag->style = "width:{$this->size};height:{$this->height}"; // tamanho em pixels
    }
}
```

```

// se o campo não é editável
if (!parent::getEditable())
{
    // desabilita a TAG input
    $this->tag->readonly = "1";
    $this->tag->class = 'tfield_disabled'; // classe CSS
}

// adiciona conteúdo ao textarea
$this->tag->add(htmlspecialchars($this->value));
// exibe a tag
$this->tag->show();
}
?>

```

6.2.4.5 TCombo

Uma combo é uma lista de opções que permite a seleção de um único elemento, sendo construída em HTML pelo elemento `<select>` em conjunto com vários elementos-filho `<option>`.

A classe `TCombo` será responsável pela construção de combo-boxes. O método construtor foi reescrito, pois na classe `TField` estamos instanciando um elemento do tipo `<input>`, quando, para construir uma combo, precisamos de um elemento do tipo `<select>`.

Uma combo possui diversos elementos (itens). Para definirmos as opções da combo, construímos o método `addItem()`, o qual irá receber um array e irá armazená-lo na propriedade `$items`. Os índices deste array serão utilizados para representar os valores das opções, e os conteúdos do array serão exibidos ao usuário. Veja na Figura 6.14 a classe `TCombo`.

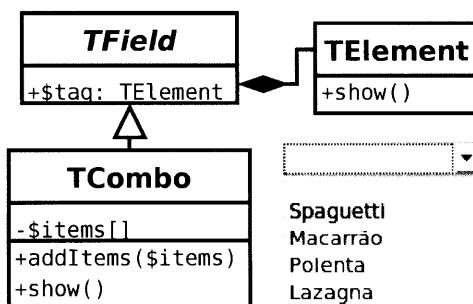


Figura 6.14 – Classe `TCombo`.

Em seu método `show()`, além de exibirmos a tag `<select>` na tela, precisamos percorrer os itens adicionados à combo pelo método `addItems()` por meio de um `foreach` e, para cada opção, instanciar um elemento `TElement` do tipo `<option>`, adicionando-o ao elemento principal. Quando formos exibir a tag pelo seu método `show()`, suas opções serão também exibidas.

TCombo.class.php

```
<?php
/**
 * classe TCombo
 * classe para construção de combo boxes
 */
class TCombo extends TField
{
    private $items; // array contendo os itens da combo

    /**
     * método construtor
     * instancia a combo box
     * @param $name = nome do campo
     */
    public function __construct($name)
    {
        // executa o método construtor da classe-pai.
        parent::__construct($name);
        // cria uma tag HTML do tipo <select>
        $this->tag = new TElement('select');
        $this->tag->class = 'tfield'; // classe CSS
    }

    /**
     * método addItems()
     * adiciona items à combo box
     * @param $items = array de itens
     */
    public function addItems($items)
    {
        $this->items = $items;
    }

    /**
     * método show()
     * exibe o widget na tela
     */
```

```
public function show()
{
    // atribui as propriedades da TAG
    $this->tag->name = $this->name;      // nome da TAG
    $this->tag->style = "width:{$this->size}"; // tamanho em pixels

    // cria uma TAG <option> com um valor padrão
    $option = new TElement('option');
    $option->add('');
    $option->value = '0';    // valor da TAG
    // adiciona a opção à combo
    $this->tag->add($option);

    if ($this->items)
    {
        // percorre os itens adicionados
        foreach ($this->items as $chave => $item)
        {
            // cria uma TAG <option> para o item
            $option = new TElement('option');
            $option->value = $chave; // define o índice da opção
            $option->add($item);   // adiciona o texto da opção

            // caso seja a opção selecionada
            if ($chave == $this->value)
            {
                // seleciona o item da combo
                $option->selected = 1;
            }
            // adiciona a opção à combo
            $this->tag->add($option);
        }
    }

    // verifica se o campo é editável
    if (!parent::getEditable())
    {
        // desabilita a TAG input
        $this->tag->readonly = "1";
        $this->tag->class = 'tfield_disabled'; // classe CSS
    }
    // exibe a combo
    $this->tag->show();
}

?>
```

6.2.4.6 TCheckButton

A classe `TCheckButton` será responsável pela exibição de botões de verificação, ou caixas de verificação. Um check button é representado por uma pequena caixa que pode ser marcada e desmarcada. Um conjunto de check buttons permite ao usuário fazer uma seleção não-exclusiva, ou seja, selecionar vários itens ao mesmo tempo. Veja no método `show()` que o tipo de input utilizado para um campo check button é `checkbox`. Veja na Figura 6.15 a classe `TCheckButton`.

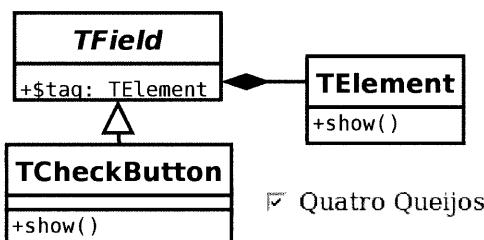


Figura 6.15 – Classe `TCheckButton`.

`TCheckButton.class.php`

```

<?php
/**
 * classe TCheckButton
 * classe para construção de botões de verificação
 */
class TCheckButton extends TField
{
    /**
     * método show()
     * exibe o widget na tela
     */
    public function show()
    {
        // atribui as propriedades da TAG
        $this->tag->name = $this->name;      // nome da TAG
        $this->tag->value = $this->value;      // valor
        $this->tag->type = 'checkbox';        // tipo do input

        // se o campo não é editável
        if (!parent::getEditable())
        {
            // desabilita a TAG input
            $this->tag->readonly = "1";
            $this->tag->class = 'tfield_disabled'; // classe CSS
        }
    }
}
  
```

```
// exibe a tag
$this->tag->show();
}

?>
```

6.2.4.7 TCheckGroup

Um check button raramente é utilizado de forma isolada. Geralmente utilizamos check buttons em grupos, permitindo ao usuário selecionar dentre várias opções. Como a classe que criamos `TCheckButton` permite a exibição somente de um check button, criaremos uma classe responsável pela exibição de um conjunto de check buttons na tela. Chamaremos esta classe de `TCheckGroup`, a qual terá três métodos: `setLayout()`, `addItems()` e `show()`.

O método `setLayout()` indicará se os botões estarão um ao lado do outro (`horizontal`) ou um abaixo do outro (`vertical`). A única diferença é que no layout vertical é dada uma quebra de linha ao final de cada opção.

O método `addItems()` recebe um conjunto de opções e armazena internamente na propriedade `$items`. As opções serão representadas por um array indexado. O índice do array será utilizado como valor-chave para cada check button, ao passo que o valor de cada opção será exibido ao usuário como rótulo. Veja na Figura 6.16 a classe `TCheckGroup`.

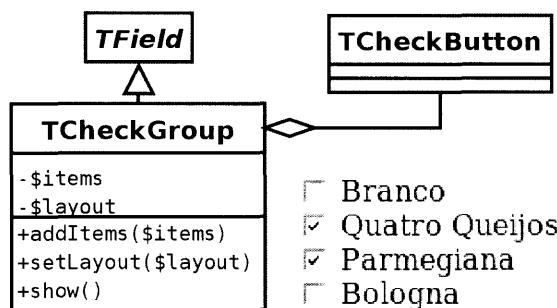


Figura 6.16 – Classe `TCheckGroup`.

O método `show()` será responsável por exibir o conjunto de check buttons na tela. Note que os itens são percorridos por meio de um `foreach` e, para cada item, é instanciado um novo check button. Para representar que se trata de um array de campos, ao final do check button adicionamos colchetes “`[]`”.

 TCheckGroup.class.php

```
<?php
/**
 * classe TCheckGroup
 * classe para exibição um grupo de CheckButtons
 */
class TCheckGroup extends TField
{
    private $layout = 'vertical';
    private $items;

    /**
     * método setLayout()
     * define a direção das opções (vertical ou horizontal)
     */
    public function setLayout($dir)
    {
        $this->layout = $dir;
    }

    /* método addItems($items)
     * adiciona itens ao check group
     * @param $items = um vetor indexado de itens
     */
    public function addItems($items)
    {
        $this->items = $items;
    }

    /**
     * método show()
     * exibe o widget na tela
     */
    public function show()
    {
        if ($this->items)
        {
            // percorre cada uma das opções do rádio
            foreach ($this->items as $index => $label)
            {
                $button = new TCheckButton("{$this->name}[]");
                $button->setValue($index);
                // verifica se deve ser marcado
                if (@in_array($index, $this->value))
                {

```

```

        $button->setProperty('checked', '1');
    }
    $button->show();
    $obj = new TLabel($label);
    $obj->show();

    if ($this->layout == 'vertical')
    {
        // exibe uma tag de quebra de linha
        $br = new TElement('br');
        $br->show();
        echo "\n";
    }
}
}
}
?

```

6.2.4.8 TRadioButton

Um RadioButton ou botão de rádio é um componente representado por um círculo que ao ser clicado é preenchido por um ponto, indicando que está selecionado naquele momento. Um radio button é criado por meio da tag <input> do HTML, sendo que a propriedade type é igual à radio. Veja na Figura 6.17 a classe TRadioButton.

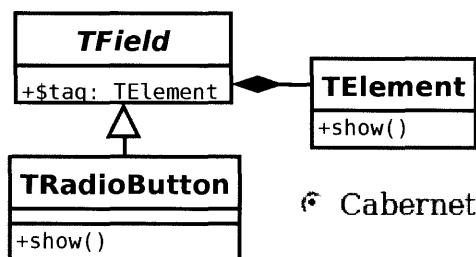


Figura 6.17 – Classe TRadioButton.

A classe para construção de um radio button é muito similar às anteriores, uma vez que a única diferença é o tipo do <input> (propriedade type). Não faz muito sentido utilizar um radio button de forma isolada, tendo em vista que ele existe para que possamos selecionar um elemento dentre vários, portanto vamos criar em seguida a classe TRadioGroup.

 **TRadioButton.class.php**

```
<?php
/*
 * classe TRadioButton
 * classe para construção de rádio
 */
class TRadioButton extends TField
{

    /**
     * método show()
     * exibe o widget na tela
     */
    public function show()
    {
        // atribui as propriedades da TAG
        $this->tag->name = $this->name;
        $this->tag->value = $this->value;
        $this->tag->type = 'radio';

        // se o campo não é editável
        if (!parent::getEditable())
        {
            // desabilita a TAG input
            $this->tag->readonly = "1";
            $this->tag->class = 'tfield_disabled';      // classe CSS
        }

        // exibe a tag
        $this->tag->show();
    }
}
?>
```

6.2.4.9 TRadioGroup

Como vimos anteriormente, não faz sentido utilizar um radio button isoladamente. Pensando nisto, criaremos uma classe para agrupar um conjunto de radio buttons. A classe **TRadioGroup** terá o método **addItems()** que recebe um array. Este array será utilizado posteriormente pelo método **show()** que irá percorrê-los por meio de um **foreach** e instanciará um objeto **TCheckButton** para cada item do array de itens. Os índices do array serão utilizados como valores de cada check button (veja **setValue()**). O método **show()** ainda marcará a propriedade **checked** para aquele check button cujo índice (**\$indice**) for igual ao valor do **TRadioGroup** (**\$this->value**). Veja na Figura 6.18 a classe **TRadioGroup**.

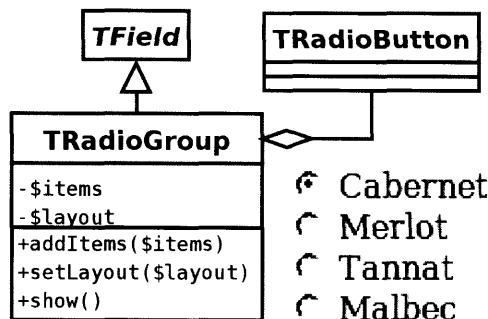


Figura 6.18 – Classe TRadioGroup.

O método `setLayout()`, assim como na classe `TCheckGroup`, definirá se os botões serão dispostos um ao lado do outro ou um abaixo do outro (com uma quebra de linha).

TRadioGroup.class.php

```

<?php
/**
 * classe TRadioGroup
 * classe para exibição de um grupo de Radio Buttons
 */
class TRadioGroup extends TField
{
    private $layout = 'vertical';
    private $items;

    /**
     * método setLayout()
     * define a direção das opções (vertical ou horizontal)
     */
    public function setLayout($dir)
    {
        $this->layout = $dir;
    }

    /**
     * método addItems($items)
     * adiciona itens (botões de rádio)
     * @param $items = array indexado contendo os itens
     */
    public function addItems($items)
    {
        $this->items = $items;
    }
}
  
```

```
/**  
 * método show()  
 * exibe o widget na tela  
 */  
public function show()  
{  
    if ($this->items)  
    {  
        // percorre cada uma das opções do rádio  
        foreach ($this->items as $index => $label)  
        {  
            $button = new TRadioButton($this->name);  
            $button->setValue($index);  
            // se possui qualquer valor  
            if ($this->value == $index)  
            {  
                // marca o radio button  
                $button->setProperty('checked', '1');  
            }  
            $button->show();  
            $obj = new TLabel($label);  
            $obj->show();  
            if ($this->layout == 'vertical')  
            {  
                // exibe uma tag de quebra de linha  
                $br = new TElement('br');  
                $br->show();  
            }  
            echo "\n";  
        }  
    }  
}  
?>
```

6.2.4.10 TButton

A classe **TButton** representará um botão de ação em um formulário. Um botão deve submeter o formulário para processamento no servidor. Para isso, é necessário que o botão saiba o nome do formulário do qual faz parte. Isto é possível por meio do método **setFormName()**, que passa o nome do formulário como parâmetro para o **TButton**.

A ação do botão é definida pelo método **setAction()**, no qual passamos como parâmetro um objeto do tipo **TAction** e um rótulo de texto (label) para ser utilizado como rótulo do botão.

A principal função da classe `TButton` é, em seu método `show()`, exibir a tag de tipo `<input>` com a ação do evento `onclick` disparando o submit do formulário, mas antes disto, no mesmo momento, a ação do formulário (`action`) é definida pela variável `$url`, que é a própria ação que passamos ao botão por meio do método `setAction()` traduzida na forma de uma URL.

 **TButton.class.php**

```
<?php
/* classe TButton
 * responsável por exibir um botão
 */
class TButton extends TField
{
    private $action;
    private $label;
    private $formName;

    /**
     * método setAction
     * define a ação do botão (função a ser executada)
     * @param $action = ação do botão
     * @param $label = rótulo do botão
     */
    public function setAction($action, $label)
    {
        $this->action = $action;
        $this->label = $label;
    }

    /**
     * método setFormName
     * define o nome do formulário para a ação botão
     * @param $name = nome do formulário
     */
    public function setFormName($name)
    {
        $this->formName = $name;
    }

    /**
     * método show()
     * exibe o botão
     */
```

```
public function show()
{
    $url = $this->action->serialize();

    // define as propriedades do botão
    $this->tag->name    = $this->name;      // nome da TAG
    $this->tag->type     = 'button';        // tipo de input
    $this->tag->value    = $this->label;       // rótulo do botão

    // se o campo não é editável
    if (!parent::getEditable())
    {
        $this->tag->disabled = "1";
        $this->tag->class   = 'tfield_disabled'; // classe CSS
    }

    // define a ação do botão
    $this->tag->onclick = "document.{$_this->formName}.action='{$url}'; ".
                           "document.{$_this->formName}.submit()";
    // exibe o botão
    $this->tag->show();
}

?>
```

6.2.5 Exemplos

6.2.5.1 Formulário estruturado e estático

Neste exemplo, além de utilizarmos a classe `TForm`, criaremos botões de ação (`TButton`), demonstrando a utilização de callbacks, que são funções definidas pelo programador que reagem a determinadas ações. Neste caso são representadas pelo clique do usuário nos botões. Neste programa, utilizaremos a classe `TTable` para organizar os elementos do formulário em linhas e colunas. Note que estamos adicionando a tabela ao formulário por meio do método `$form->add($table)`.

Este programa se constitui em um formulário para envio de e-mails contendo uma tabela com duas colunas: na primeira adicionamos sempre os rótulos dos campos, representados por objetos do tipo `TLabel`, como nome, email, título e mensagem; na segunda, adicionamos os objetos editáveis pelo usuário, como `TEntry` e `TText`, dentre outros. Veja na Figura 6.19 o resultado deste exemplo.

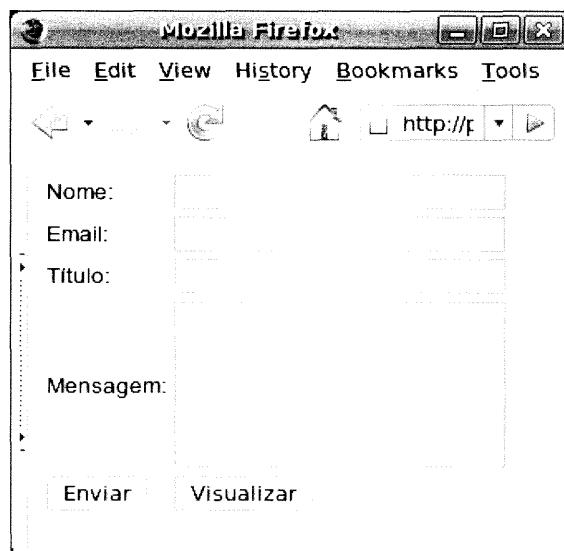


Figura 6.19 – Usando a TForm de maneira estruturada e estática.

form1.php

```

<?php
/*
 * função __autoload()
 * carrega uma classe quando ela é necessária,
 * ou seja, quando ela é instanciada 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');
$título    = new TEntry('título');
$mensagem  = new TText('mensagem');

```

```
// adiciona uma linha para o campo nome
$row=$table->addRow();
$row->addCell(new TLabel('Nome:'));
$row->addCell($nome);

// adiciona uma linha para o campo email
$row=$table->addRow();
$row->addCell(new TLabel('Email:'));
$row->addCell($email);

// adiciona uma linha para o campo título
$row=$table->addRow();
$row->addCell(new TLabel('Título:'));
$row->addCell($titulo);

// adiciona uma linha para o campo mensagem
$row=$table->addRow();
$row->addCell(new TLabel('Mensagem:'));
$row->addCell($mensagem);

// cria dois botões de ação para o formulário
$action1=new TButton('action1');
$action2=new TButton('action2');
// define as ações dos botões
$action1->setAction(new TAction('onSend'), 'Enviar');
$action2->setAction(new TAction('onView'), 'Visualizar');

// adiciona uma linha para as ações do formulário
$row=$table->addRow();
$row->addCell($action1);
$row->addCell($action2);

// define quais são os campos do formulário
$form->setFields(array($nome, $email, $titulo, $mensagem, $action1, $action2));
```

Este programa também terá dois botões de ação: **Enviar** e **Visualizar**. Para o botão **Enviar**, definimos a ação `onSend()` por meio do método `setAction()`. Isso significa que sempre que o botão **Enviar** for clicado, a função `onSend()` será executada, assim como sempre que o botão **Visualizar** for clicado, a função `onView()` será executada. Note que o clique do usuário no botão leva o navegador a recarregar a página, passando alguns parâmetros adicionais; neste caso “`?method=onSend`” ou “`?method=onEnviar`”, que são interpretados e executados pelo método `show()` da classe `TPage`.

A função `onView()` coleta os dados postados (enviados) pelo formulário por meio do método `getData()` da classe `TForm`, e, em seguida, cria uma janela nova (`TWindow`). Dentro desta janela colocamos um componente de edição de texto (`TText`) e, dentro deste,

colocamos uma string contendo os dados digitados pelo usuário no formulário. Veja na Figura 6.20 o resultado da ação ao clicar no botão **Visualizar**.

The screenshot shows a web form on the left and its output in a separate window on the right.

Form Data:

Nome:	Marcelo Ferretti
Email:	marcelo@rslibris.com.br
Título:	Olá
Mensagem:	Estou fazendo contato...

Buttons:

- Enviar
- Visualizar

Dados do Form (Output Window):

```

Nome: Marcelo Ferretti
Email: marcelo@rslibris.com.br
Título: Olá
Mensagem:
Olá
Estou fazendo contato...

```

Figura 6.20 – Resultado ao clicar no botão **Visualizar**.

```

/*
 * função onView
 * visualiza os dados do formulário
 */
function onView()
{
    global $form;
    // obtém os dados do formulário
    $data = $form->getData();
    // atribui os dados de volta ao formulário
    $form->setData($data);

    // cria uma janela
    $window = new TWindow('Dados do Form');
    // define posição e tamanho em pixels
    $window->setPosition(300, 70);
    $window->setSize(300,150);

    // monta o texto a ser exibido
    $output = "Nome: {$data->nome}\n";
    $output.= "Email: {$data->email}\n";
    $output.= "Título: {$data->título}\n";
    $output.= "Mensagem: \n{$data->mensagem}";

    // cria um objeto de texto
    $text = new TText('texto', 300);
    $text->setSize(290,120);
    $text->setValue($output);
    // adiciona o objeto à janela
    $window->add($text);
    $window->show();
}

```

A função `onSend()` é executada sempre que o botão **Enviar** for clicado. Esta função simplesmente exibe uma mensagem ao usuário contendo o texto “Enviando dados...” enquanto torna o formulário não-editável por meio do método `setEditable(FALSE)`. Note que é necessário utilizar o método `setData()`, passando os dados de volta ao formulário, caso desejemos que os dados permaneçam mesmo depois de o método ser executado. Veja na Figura 6.21 o resultado da ação ao clicar no botão **Enviar**.



Figura 6.21 – Resultado ao clicar no botão **Enviar**.

```
/*
 * função onSend
 * exibe mensagem "Enviando dados..."
 */
function onSend()
{
    global $form;
    // obtém os dados do formulário
    $data = $form->getData();
    // atribui os dados de volta ao formulário
    $form->setData($data);
    // torna o formulário não-editável
    $form->setEditable(FALSE);
    // exibe mensagem ao usuário
    new TMessage('info', 'Enviando dados...');

}
```

Ao final da execução instanciamos o objeto `TPage`, adicionamos o formulário na página e então exibimos a mesma na tela. No momento em que executamos `$page->show()`, uma série de verificações são feitas no sentido de descobrir quais são os métodos que devem ser executados, baseando-se nos parâmetros passados via URL (`$class` e `$method`).

```
$page = new TPage;
$page->add($form);
$page->show();
?>
```

6.2.5.2 Formulário orientado a objetos e estático

Até o momento, construímos nossos formulários programando de maneira estruturada. No exemplo a seguir, trabalharemos exclusivamente com uma estrutura de classes, inclusive para a construção da página, que passa a ser uma classe-filha de TPage, criada por um mecanismo de herança (*extends*). Chamaremos esta classe de EmailForm, já que se trata de um formulário para envio de e-mails.

No método construtor da página (*__construct*) definimos o nosso formulário da mesma forma que no exemplo anterior. Observe que a variável que armazena o formulário é agora uma propriedade desta classe (*\$this->form*), podendo ser acessada a partir de qualquer método sem que seja necessário declará-la como global. Note também que os métodos da classe TPage, quando necessários, são executados via *parent::metodo()*, uma vez que a classe TPage é a classe-pai de EmailForm. Veja na Figura 6.22 o resultado deste exemplo.

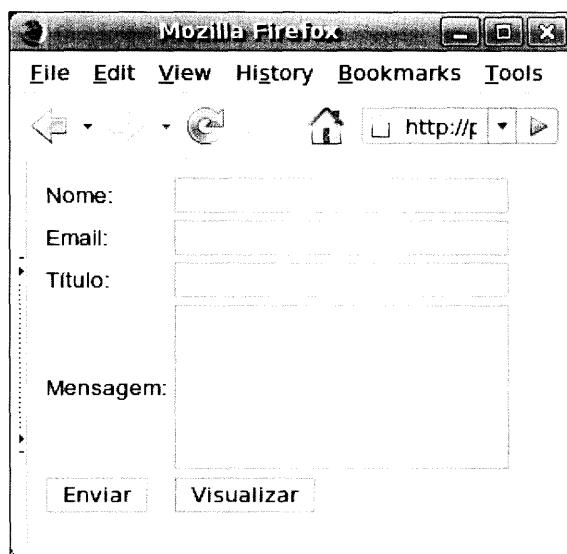


Figura 6.22 – Usando a TForm de maneira orientada a objetos e estática.

Este programa é exatamente igual ao anterior. Note que ao final dele, instanciamos um objeto da classe EmailForm e executamos o seu método *show()*, exibindo a página juntamente com todo o seu conteúdo na tela.

form2.php

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

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

class EmailForm extends TPage
{
    private $form; // objeto formulário

    function __construct()
    {
        parent::__construct();
        // instancia um formulário
        $this->form = new TForm('form_email');
        // instancia uma tabela
        $table = new TTable;

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

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

        // adiciona uma linha para o campo nome
        $row=$table->addRow();
        $row->addCell(new TLabel('Nome:'));
        $row->addCell($nome);

        // adiciona uma linha para o campo email
        $row=$table->addRow();
        $row->addCell(new TLabel('Email:'));
        $row->addCell($email);

        // adiciona uma linha para o campo título
        $row=$table->addRow();
        $row->addCell(new TLabel('Título:'));
        $row->addCell($titulo);

        // adiciona uma linha para o campo mensagem
        $row=$table->addRow();
        $row->addCell(new TLabel('Mensagem:'));
        $row->addCell($mensagem);
```

```

// cria dois botões de ação para o formulário
$action1=new TButton('action1');
$action2=new TButton('action2');
// define as ações dos botões
$action1->setAction(new TAction(array($this, 'onSend')), 'Enviar');
$action2->setAction(new TAction(array($this, 'onView')), 'Visualizar');

// adiciona uma linha para as ações do formulário
$row=$table->addRow();
$row->addCell($action1);
$row->addCell($action2);

// define quais são os campos do formulário
$this->form->setFields(array($nome, $email, $titulo, $mensagem, $action1, $action2));

parent::add($this->form);
}

```

Outra diferença muito importante é que antes estávamos representando o nome das ações a serem desempenhadas por uma string, como em “onSend” ou “onView”. Ao programarmos com orientação a objetos, não mais trabalharemos com funções, mas com métodos de objetos. Neste caso, precisamos de uma forma para representar métodos de objetos, e a forma mais comum para isto é utilizar um array de duas posições, de modo que a primeira posição contém o objeto, e a segunda contém o nome do método a ser executado. Veja no método setAction() do código anterior como fizemos isto.

```

/*
 * função onView
 * visualiza os dados do formulário
 */
function onView()
{
    // obtém os dados do formulário
    $data = $this->form->getData();
    // atribui os dados de volta ao formulário
    $this->form->setData($data);

    // cria uma janela
    $window = new TWindow('Dados do Form');
    // define posição e tamanho em pixels
    $window->setPosition(300, 70);
    $window->setSize(300,150);

    // monta o texto a ser exibido
    $output = "Nome:    {$data->nome}\n";
    $output.= "Email:   {$data->email}\n";
    $output.= "Título:  {$data->titulo}\n";
}

```

```
$output.= "Mensagem: \n{$data->mensagem}";  
  
// cria um objeto de texto  
$text = new TText('texto', 300);  
$text->setSize(290,120);  
$text->setValue($output);  
// adiciona o objeto à janela  
$window->add($text);  
$window->show();  
}  
  
/*  
 * função onSend  
 * exibe mensagem "Enviando dados..."  
 */  
function onSend()  
{  
    // obtém os dados do formulário  
    $data = $this->form->getData();  
    // atribui os dados de volta ao formulário  
    $this->form->setData($data);  
    // torna o formulário não-editável  
    $this->form->setEditable(FALSE);  
    // exibe mensagem ao usuário  
    new TMessage('info', 'Enviando dados...');  
}  
}
```

Ao final do programa, instanciamos um objeto dessa classe e executamos seu método `show()`, exibindo assim a página e seu conteúdo em tela. Lembre que o conteúdo é adicionado pelo método `add()` da classe `TPage`. Neste caso o conteúdo foi adicionado no método construtor pelo método `parent::add()`, colocando o formulário dentro da página.

```
$page = new EmailForm;  
$page->show();  
?>
```

6.2.5.3 Formulário estruturado, com banco de dados

Neste programa, demonstraremos alguns componentes ainda não vistos até o momento, como `TRadioGroup`, para construir um grupo de botões de rádio, ou `TCheckGroup`, para construir um grupo de botões de verificação. O objetivo deste programa é construir um formulário para cadastro de pessoas e, para isso, teremos campos para informar o nome, o endereço, o sexo, as línguas que a pessoa fala e as suas qualificações. Para criar a tabela de pessoas no banco de dados utilizaremos o seguinte comando:

```
CREATE TABLE pessoa ( id integer, nome varchar(40), endereco varchar(40), dataasc date,
    sexo char(1), linguas varchar(40), qualifica text);
```

O campo código (`id`) será um campo `TEntry` não-editável. Fizemos isso porque o código da pessoa será dado automaticamente após a sua gravação no banco de dados, quando este campo será preenchido. Ademais, diminuimos seu tamanho para 100 pixels. O campo nome e o campo endereço também serão objetos `TEntry` de livre digitação. O campo sexo será um grupo de botões de rádio (radio buttons), com duas opções `M=>Masculino` e `F=>Feminino`. Estas opções são representadas por um array (`$items`), adicionado ao objeto `TRadioGroup` por seu método `addItems()`.

O campo língua, diferentemente do campo sexo, não é um campo de seleção exclusiva, portanto utilizaremos um objeto `TCheckGroup`, que permite a múltipla seleção de elementos. As opções também são representadas por um array (`$items`), onde `E=>Inglês`, `S=>Espanhol`, `I=>Italiano` e `F=>Francês`. Assim como no grupo de radio do sexo, deixamos a opção `M=>Masculino` marcada por default por meio do método `setValue()`. No grupo de check buttons deixamos marcados por default as opções `E=>Inglês` e `I=>Italiano`. Neste caso, precisamos passá-las por um vetor, indicando os índices das opções que serão marcadas. O campo para digitação das qualificações será um campo `TText`, no qual o usuário poderá digitar várias linhas de texto; ele terá um texto inicial `<digite suas qualificações aqui>` definido pelo método `setValue()` e um tamanho de 240x100 definido pelo método `setSize()`. Veja na Figura 6.23 o resultado deste exemplo.

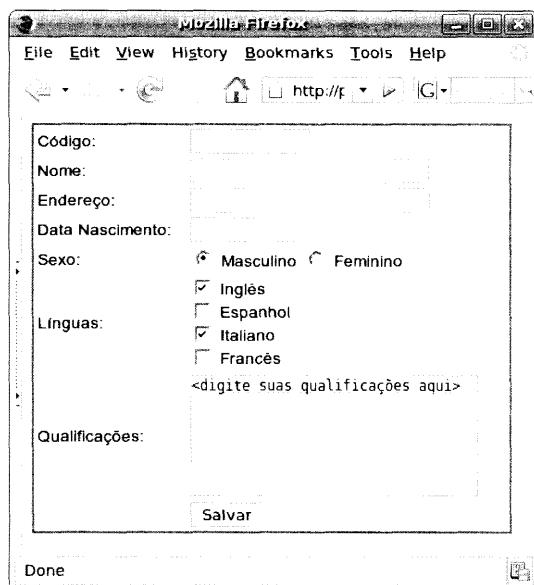


Figura 6.23 – Usando a `TForm`, de maneira estruturada, com banco de dados.

Após definidos os campos, estes são adicionados em um objeto `TTable` que irá organizá-los em um visual de tabela, na qual as primeiras colunas irão conter objetos

do tipo TLabel (rótulos de texto), ao passo que na segunda colunas teremos os objetos anteriormente descritos.

 form3.php

```
<?php
/*
 * função __autoload()
 * carrega uma classe quando ela é necessária,
 * ou seja, quando ela é instanciada pela primeira vez.
 */
function __autoload($classe)
{
    $pastas = array('app.widgets', 'app.ado');
    foreach ($pastas as $pasta)
    {
        if (file_exists("${$pasta}/{$classe}.class.php"))
        {
            include_once "${$pasta}/{$classe}.class.php";
        }
    }
}
// cria classe para manipulação dos registros de pessoas
class PessoaRecord extends TRecord {}

// instancia um formulário
$form = new TForm('form_pessoas');

// instancia uma tabela
$table = new TTable;

// define algumas propriedades da tabela
$table->bgcolor = '#f0f0f0';
$table->style   = 'border:2px solid grey';
$table->width   = 400;

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

// cria os campos do formulário
$codigo  = new TEntry('id');
$nome    = new TEntry('nome');
$endereco = new TEntry('endereco');
$datanasc = new TEntry('datanasc');
$sexo     = new TRadioGroup('sexo');
$linguas  = new TCheckGroup('linguas');
$qualifica= new TText('qualifica');
```

```
// define tamanho para campo código
$codigo->setSize(100);
// define como somente leitura
$codigo->setEditable(FALSE);

// cria um vetor com as opções de sexo
$items = array();
$items['M'] = 'Masculino';
$items['F'] = 'Feminino';

// define tamanho para campo data de nascimento
$datanasc->setSize(100);

// adiciona as opções ao radio button
$sexo->addItems($items);
// define a opção ativa
$sexo->setValue('M');
// define a posição dos elementos
$sexo->setLayout('horizontal');

// cria um vetor com as opções de idiomas
$items= array();
$items['E'] = 'Inglês';
$items['S'] = 'Espanhol';
$items['I'] = 'Italiano';
$items['F'] = 'Francês';
// adiciona as opções ao check button
$linguas->addItems($items);
// define as opções ativas
$linguas->setValue(array('E','I'));

// define um valor padrão para o campo
$qualifica->setValue('<digite suas qualificações aqui>');
$qualifica->setSize(240,100);

// adiciona uma linha para o campo código na tabela
$row=$table->addRow();
$row->addCell(new TLabel('Código:'));
$row->addCell($codigo);

// adiciona uma linha para o campo nome na tabela
$row=$table->addRow();
$row->addCell(new TLabel('Nome:'));
$row->addCell($nome);

// adiciona uma linha para o campo endereço na tabela
$row=$table->addRow();
$row->addCell(new TLabel('Endereço:'));
$row->addCell($endereco);
```

```
// adiciona uma linha para o campo data na tabela
$row=$table->addRow();
$row->addCell(new TLabel('Data Nascimento:'));
$row->addCell($datanasc);

// adiciona uma linha para o campo sexo na tabela
$row=$table->addRow();
$row->addCell(new TLabel('Sexo:'));
$row->addCell($sexo);

// adiciona uma linha para o campo línguas na tabela
$row=$table->addRow();
$row->addCell(new TLabel('Línguas:'));
$row->addCell($linguas);

// adiciona uma linha para o campo qualificações na tabela
$row=$table->addRow();
$row->addCell(new TLabel('Qualificações:'));
$row->addCell($qualifica);

// adiciona um botão de ação ao formulário
// ele irá executar a função onSave
$submit=new TButton('action1');
$submit->setAction(new TAction('onSave'), 'Salvar');

$row=$table->addRow();
$row->addCell(new TLabel(''));
$row->addCell($submit);

// define quais são os campos do formulário
$form->setFields(array($codigo, $nome, $endereço, $datanasc, $sexo, $linguas, $qualifica, $submit));

// instancia uma nova página
$page = new TPage;
// adiciona o formulário na página
$page->add($form);
// exibe a página e seu conteúdo
$page->show();
```

Este programa terá duas funções: `onSave()` e `onEdit()`. A função `onSave()` será executada sempre que o usuário clicar no botão **Salvar**. Esta função irá coletar os dados do formulário por meio do método `getData()` e irá instanciar um objeto Active Record da classe `PessoaRecord`. A partir de então, dentro de uma transação com a base de dados `pg_livro`, transformamos o array contendo as línguas (`$pessoa->linguas`) em uma string separada por espaços em branco por meio da função `implode()`.

Também convertemos a data de nascimento digitada no formato brasileiro (dd/mm/aaaa) para o formato americano por meio da função `conv_data_to_us()` e, então, executamos o método `store()` para armazenar esse registro na base de dados. Em seguida, jogamos os dados de volta ao formulário, quando já deveremos ter o campo código preenchido com o próximo ID disponível na base de dados, atribuído durante a gravação do registro na base de dados. Também exibimos ao usuário a mensagem “Dados armazenados com sucesso”.

```

/*
 * função onSave
 * obtém os dados do formulário e salva na base de dados
 */
function onSave()
{
    global $form;
    $pessoa = $form->getData('PessoaRecord');

    try
    {
        // inicia transação com o banco 'pg_livro'
        TTransaction::open('pg_livro');
        $pessoa->linguas = implode(' ', $pessoa->linguas);
        $pessoa->datanasc = conv_data_to_us($pessoa->datanasc);
        $pessoa->store();

        // finaliza a transação
        TTransaction::close();
        new TMessage('info', 'Dados armazenados com sucesso');
    }
    catch (Exception $e)      // em caso de exceção
    {
        // exibe a mensagem gerada pela exceção
        new TMessage('error', '<b>Erro</b>' . $e->getMessage());
        // desfaz todas alterações no banco de dados
        TTransaction::rollback();
    }
}

```

A segunda função deste programa é `onEdit()`. Note que não existe nenhum botão disponível no formulário para a execução do método `onEdit()`, isto porque o método `onEdit` será executado diretamente pela URL do sistema sempre que quisermos editar um registro da seguinte forma:

`form3.php?method=onEdit&id=3`

Dessa forma, estaremos dizendo à classe TPage (que controla qual método será executado) que queremos executar a função `onEdit()`, passando o parâmetro “3”, que é o código da pessoa que queremos editar. Esse método se encarregará de buscar as informações da pessoa no banco de dados e já exibir o formulário preenchido com tais dados para que o usuário possa editá-los.

Todo método recebe por padrão a variável `$_GET`, que contém todos os parâmetros passados via URL da página. Neste caso, o método `onEdit()` abre uma transação com a base de dados `pg_livro`, instancia um Active Record `PessoaRecord`, passando o parâmetro `ID` recebido via URL. Em seguida, convertemos a propriedade línguas, que é uma string, em um array, por meio da função `explode()`, e também convertemos a data, armazenada no formato americano `yyyy-mm-dd`, para o formato brasileiro. Posteriormente, jogamos o registro no formulário e fechamos a transação. Assim como no método `onSave()`, toda a operação é realizada dentro de um bloco `try/catch` para controle de transações.

```
/***
 * função onEdit
 * carrega os dados do registro no formulário
 * @param $param = parâmetros passados via URL ($_GET)
 */
function onEdit($param)
{
    global $form;

    try
    {
        // inicia transação com o banco 'pg_livro'
        TTransaction::open('pg_livro');

        // obtém a pessoa a partir do parâmetro ID
        $pessoa= new PessoaRecord($param['id']);
        $pessoa->linguas = explode(' ', $pessoa->linguas);
        $pessoa->datanasc = conv_data_to_br($pessoa->datanasc);
        $form->setData($pessoa);
        // finaliza a transação
        TTransaction::close();
    }
    catch (Exception $e) // em caso de exceção
    {
        // exibe a mensagem gerada pela exceção
        new TMessage('error', '<b>Erro</b>' . $e->getMessage());
        // desfaz todas alterações no banco de dados
        TTransaction::rollback();
    }
}
```

```

/**
 * função conv_data_to_us
 * converte uma data do formato brasileiro para o americano
 * @param $data = data (dd/mm/aaaa) a ser convertida
 */
function conv_data_to_us($data)
{
    $dia = substr($data,0,2);
    $mes = substr($data,3,2);
    $ano = substr($data,6,4);

    return "{$ano}-{$mes}-{$dia}";
}

/**
 * função conv_data_to_br
 * converte uma data do formato americano para o brasileiro
 * @param $data = data (yyyy-mm-dd) a ser convertida
 */
function conv_data_to_br($data)
{
    $ano = substr($data,0,4);
    $mes = substr($data,5,2);
    $dia = substr($data,8,4);

    return "{$dia}/{$mes}/{$ano}";
}
?>

```

6.2.5.4 Formulário orientado a objetos, com banco de dados

No próximo programa, diferentemente do anterior, iremos novamente utilizar uma estrutura de classe para a construção do formulário. O objetivo deste programa é construir um formulário para cadastro de livros. Neste formulário, teremos um campo com o ID do registro, que será preenchido automaticamente no momento em que clicarmos no botão **salvar**, assim como no exemplo anterior. Teremos campos do tipo TEntry para a digitação do título e do autor do livro. Teremos uma combo-box (objeto TCombo) para seleção do tema do livro (Administração, Informática, Economia, Matemática), dois campos TEntry para digitação da Editora e do ano de edição e também um campo de digitação de texto (TText) para digitação do resumo da obra. Os itens da combo-box serão adicionados por meio do método `addItems()`, que recebe um array contendo os itens da combo. O funcionamento é idêntico ao da TCheckGroup e TRadioGroup. Para criar a tabela de livros na base de dados utilizaremos o seguinte comando:

```
CREATE TABLE livro (
    id integer,
    titulo varchar(40),
    autor varchar(40),
    tema char(1),
    editora varchar(40),
    ano varchar(4),
    resumo text);
```

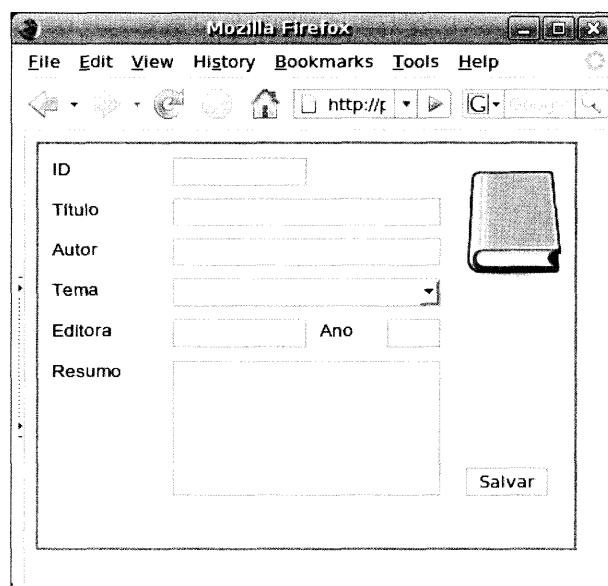


Figura 6.24 – Usando a TForm de maneira orientada a objetos, com banco de dados.

Neste formulário, visto na Figura 6.24, adotaremos uma abordagem diferente para distribuição dos objetos. Até o momento, vinhamos utilizando a classe TTable para distribuir os objetos em um visual de linhas e colunas. Neste exemplo, utilizaremos a classe TPanel, que permite definir coordenadas absolutas para fixarmos os objetos em seu interior. Todos os objetos serão posicionados dentro do painel por meio do método put() da classe TPanel, indicando as coordenadas (coluna, linha). Para demonstrar a flexibilidade de posicionamento de objetos pelo TPanel, posicionaremos uma imagem de um livro (objeto TImage) dentro do formulário ao lado dos campos título e autor.

form4.php

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

```
function __autoload($classe)
{
    $pastas = array('app.widgets', 'app.ado');
    foreach ($pastas as $pasta)
    {
        if (file_exists("${$pasta}/${$classe}.class.php"))
        {
            include_once "${$pasta}/${$classe}.class.php";
        }
    }
}

// Active Record para tabela Livro
class LivroRecord extends TRecord {}

class LivrosForm extends TPage
{
    private $form;

    function __construct()
    {
        parent::__construct();
        // instancia o formulário
        $this->form = new TForm;
        $this->form->setName('form_livros');

        // instancia o painel
        $panel= new TPanel(400,300);
        $this->form->add($panel);

        // coloca o campo id no formulário
        $panel->put(new TLabel('ID'), 10, 10);
        $panel->put($id = new TEntry('id'), 100,10);
        $id->setSize(100);
        $id->setEditable(FALSE);

        // coloca a imagem de um livro
        $panel->put(new TImage('book.png'), 320, 20);

        // coloca o campo titulo no formulário
        $panel->put(new TLabel('Título'), 10, 40);
        $panel->put($titulo = new TEntry('titulo'), 100,40);

        // coloca o campo autor no formulário
        $panel->put(new TLabel('Autor'), 10, 70);
        $panel->put($autor = new TEntry('autor'), 100,70);
        // coloca o campo tema no formulário
        $panel->put(new TLabel('Tema'), 10, 100);
```

```
$panel->put($tema= new TCombo('tema'), 100,100);

// cria um vetor com as opções da combo tema
$itemss= array();
$itemss['1'] = 'Administração';
$itemss['2'] = 'Informática';
$itemss['3'] = 'Economia';
$itemss['4'] = 'Matemática';
// adiciona os itens na combo
$tema->addItems($itemss);

// coloca o campo editora no formulário
$editora = new TEntry('editora');
$panel->put(new TLabel('Editora'), 10,130);
$panel->put($editora, 100, 130);
// coloca o campo ano no formulário
$panel->put(new TLabel('Ano'), 210, 130);
$panel->put($ano = new TEntry('ano'), 260, 130);

$editora->setSize(100);
$ano->setSize(40);
// coloca o campo resumo no formulário
$panel->put(new TLabel('Resumo'), 10, 160);
$panel->put($resumo = new TText('resumo'), 100, 160);

// cria uma ação
$panel->put($acao = new TButton('action'), 320, 240);
$acao->setAction(new TAction(array($this, 'onSave')));

// define quais são os campos do formulário
$this->form->setFields(array($id, $titulo, $autor, $tema, $editora,
                                $ano, $resumo, $acao));

parent::add($this->form);
}
```

O método `onSave()` é acionado pelo botão **salvar**. Ele abre uma transação com a base de dados, obtém os dados digitados no formulário em um objeto do tipo `LivroRecord` por meio do método `getData()`, armazena o registro (`store`), joga os dados de volta ao formulário (`setData`), marca o formulário como não-editável (`setEditable`), fecha a transação e exibe a mensagem de sucesso. Em caso de falhas, abortamos a transação (`rollback`) e exibimos uma mensagem de erro.

```
/*
 * método onSave
 * obtém os dados do formulário e salva na base de dados
 */
```

```

function onSave()
{
    try
    {
        // inicia transação com o banco 'pg_livro'
        TTransaction::open('pg_livro');
        // obtém dados
        $livro = $this->form->getData('LivroRecord');
        // armazena registro
        $livro->store();
        // joga os dados de volta ao formulário
        $this->form->setData($livro);
        // define o formulário como não-editável
        $this->form->setEditable(FALSE);
        // finaliza a transação
        TTransaction::close();
        new TMessage('info', 'Dados armazenados com sucesso');
    }
    catch (Exception $e)      // em caso de exceção
    {
        // exibe a mensagem gerada pela exceção
        new TMessage('error', '<b>Erro</b>' . $e->getMessage());
        // desfaz todas alterações no banco de dados
        TTransaction::rollback();
    }
}

```

O método `onEdit()` é acionado sempre que o usuário entrar via URL do sistema com o seguinte conteúdo: “`form4.php?class=LivrosForm&method=onEdit&id=4`”. Neste caso, estaremos solicitando para que seja executado o método `onEdit()` da classe `LivrosForm`, passando o `id=4`. Este método carrega o registro identificado por este `id` no momento de instanciar o objeto `LivroRecord` e joga os dados no formulário por meio do método `setData()` dentro de uma transação.

```

/**
 * método onEdit
 * carrega os dados do registro no formulário
 * @param $param = parâmetros passados via URL ($_GET)
 */
function onEdit($param)
{
    try
    {
        // inicia transação com o banco 'pg_livro'
        TTransaction::open('pg_livro');
        // obtém o livro pelo ID
        $livro= new LivroRecord($param['id']);
    }
}

```

```
// joga os dados no formulário
$this->form->setData($livro);

// finaliza a transação
TTransaction::close();
}

catch (Exception $e) // em caso de exceção
{
    // exibe a mensagem gerada pela exceção
    new TMessage('error', '<b>Erro</b>' . $e->getMessage());
    // desfaz todas alterações no banco de dados
    TTransaction::rollback();
}
}

// instancia e exibe a página
$page = new LivrosForm;
$page->show();
?>
```

6.3 Listagens

Uma listagem geralmente é utilizada para apresentar um conjunto de registros de um banco de dados, apesar de poder listar informações de praticamente qualquer fonte de informações, como arquivos XML, por exemplo. A apresentação de uma listagem segue uma forma tabular e geralmente utiliza-se uma tabela para organizar visualmente a listagem. Uma listagem é composta por linhas, colunas e ações. Nas colunas, distribuímos as informações que desejamos apresentar (Código, Nome, Telefone); nas linhas distribuímos cada um dos itens a exibir. Uma listagem geralmente apresenta ações ao usuário (editar, visualizar, excluir, imprimir), sendo que estas ações são executadas sobre cada um dos itens apresentados na listagem.

6.3.1 Exemplos de listagens

6.3.1.1 Listagem estática

Neste primeiro exemplo, iremos demonstrar uma listagem estática no formato HTML. Veja que o formato tabular da listagem é definido pela utilização do elemento `<table>`. O cabeçalho da listagem é formado por uma linha contendo os rótulos de cada coluna. Nesta linha temos uma coluna vazia (será a coluna de ação) e também colunas para código, nome, endereço e telefone. As demais linhas irão conter os dados da listagem. Observe que a primeira coluna de cada linha terá sempre um link para a

ação editar, juntamente com o ícone ico_edit.png. Quando o usuário clicar nesta ação, será direcionado para o respectivo programa (edit.php), passando como parâmetro o código do registro (chave primária), que neste caso é o id. Veja na Figura 6.25 o resultado deste exemplo.

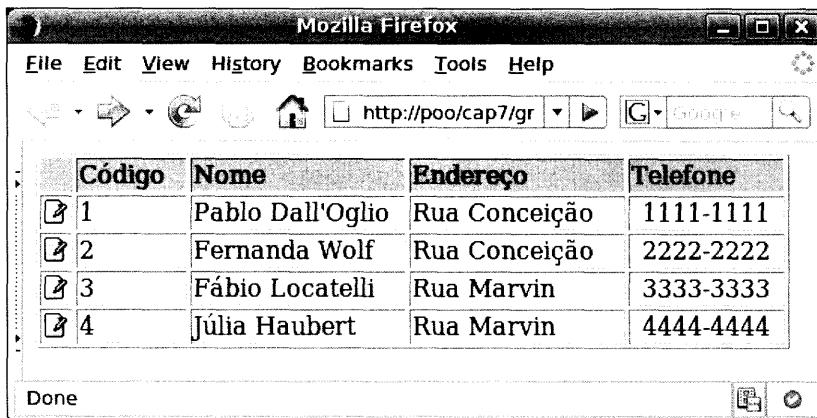


Figura 6.25 – Listagem de dados estática.

list.html

```
<html>
<body>
    <table border=1>
        <tr bgcolor=#c0c0c0>
            <td></td>
            <td width="70">Código</td>
            <td width="140">Nome</td>
            <td width="140">Endereço</td>
            <td width="100">Telefone</td>
        </tr>

        <tr>
            <td><a href="edit.php?id=1"></a></td>
            <td align="left">1</td>
            <td align="left">Pablo Dall'Oglio</td>
            <td align="left">Rua Conceição</td>
            <td align="center">1111-1111</td>
        </tr>

        <tr>
            <td><a href="edit.php?id=2"></a></td>
            <td align="left">2</td>
            <td align="left">Fernanda Wolf</td>
            <td align="left">Rua Conceição</td>
```

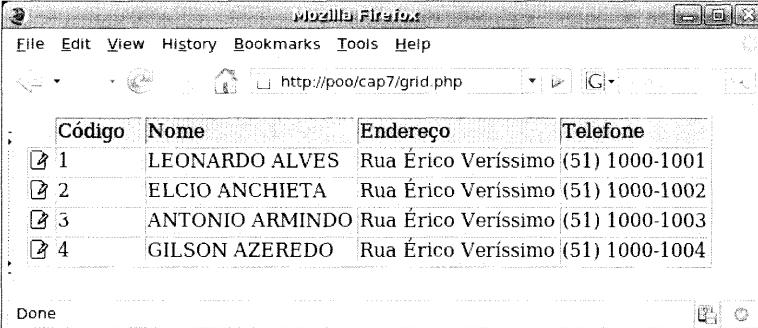
```
<td align="center">2222-2222</td>
</tr>

<tr>
    <td><a href="edit.php?id=3"></a></td>
    <td align="left">3</td>
    <td align="left">Fábio Locatelli</td>
    <td align="left">Rua Marvin</td>
    <td align="center">3333-3333</td>
</tr>

<tr>
    <td><a href="edit.php?id=4"></a></td>
    <td align="left">4</td>
    <td align="left">Júlia Haubert</td>
    <td align="left">Rua Marvin</td>
    <td align="center">4444-4444</td>
</tr>
</table>
</body>
</html>
```

6.3.1.2 Listagem dinâmica

Utilizamos o exemplo anterior para demonstrar como se dá a estrutura de uma listagem, embora poucas vezes utilizemos um programa desta forma. Geralmente buscamos as informações no banco de dados e as exibimos de forma dinâmica, como em um laço de repetições. No programa a seguir, mesclaremos algumas informações estáticas, como o cabeçalho da listagem com informações provindas do banco de dados. Para isso, iremos abrir uma conexão com o banco de dados e, em seguida, percorrer o seu retorno, exibindo uma linha de tabela `<tr>` para cada registro retornado, inserindo as variáveis dentro de cada string. Veja na Figura 6.26 o resultado deste exemplo.



The screenshot shows a Mozilla Firefox browser window with the title bar "Mozilla Firefox". The menu bar includes "File", "Edit", "View", "History", "Bookmarks", "Tools", and "Help". The address bar displays the URL "http://poo/cap7/grid.php". The main content area shows a table with four rows of data. The table has four columns with headers: "Código", "Nome", "Endereço", and "Telefone". The data rows are as follows:

Código	Nome	Endereço	Telefone
1	LEONARDO ALVES	Rua Érico Veríssimo	(51) 1000-1001
2	ELCIO ANCHIETA	Rua Érico Veríssimo	(51) 1000-1002
3	ANTONIO ARMINDO	Rua Érico Veríssimo	(51) 1000-1003
4	GILSON AZEREDO	Rua Érico Veríssimo	(51) 1000-1004

Figura 6.26 – Listagem de dados dinâmica.

 list.php

```
<?php  
// exibe início da tabela  
echo '<table border=1>  
    <tr bgcolor=#c0c0c0>  
        <td></td>  
        <td width="70">Código</td>  
        <td width="140">Nome</td>  
        <td width="140">Endereço</td>  
        <td width="100">Telefone</td>  
    </tr>';  
  
// abre conexão com Postgres  
$conn = pg_connect("host=localhost port=5432 dbname=livro user=postgres");  
// define consulta que será realizada  
$query = 'select id, nome, endereco, telefone from aluno limit 4';  
// envia consulta ao banco de dados  
$result = pg_query($conn, $query);  
  
if ($result)  
{  
    // percorre resultados da pesquisa  
    while ($row = pg_fetch_assoc($result))  
    {  
        $id      = $row['id'];  
        $nome    = $row['nome'];  
        $endereco = $row['endereco'];  
        $telefone = $row['telefone'];  
        // exibe uma linha de resultados  
        echo "<tr>  
            <td><a href='edit.php?id={$id}'><img border=0 src='app/images/ico_edit.png'></a></td>  
            <td align='left'>{$id}</td>  
            <td align='left'>{$nome}</td>  
            <td align='left'>{$endereco}</td>  
            <td align='center'>{$telefone}</td>  
        </tr>";  
    }  
}  
// fecha a conexão  
pg_close($conn);  
  
// imprime fechamento da tabela  
echo '</table>';  
?>
```

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 agrregar diversas colunas (`TDataGridColumn`) e ações (`TDataGridAction`) na listagem (`TDataGrid`).

Código	Nome	Estado
1	Rio Grande do Sul	RS
2	São Paulo	SP
3	Minas Gerais	MG
4	Rio de Janeiro	RJ

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

utilizado para renderizar uma coluna do cabeçalho quando o usuário estiver com o mouse sobre ela. Neste caso, uma borda laranja será exibida sobre a célula e o cursor será alterado para uma mão (pointer). Veja na Figura 6.28 a classe TDataGrid.

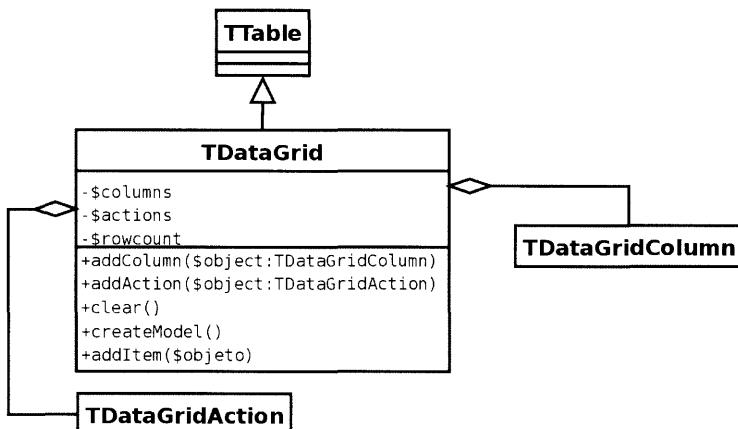


Figura 6.28 – Classe TDataGrid.

TDataGrid.class.php

```

<?
/** 
 * classe TDataGrid
 * classe para construção de Listagens
 */
class TDataGrid extends TTable
{
    private $columns;
    private $actions;
    private $rowcount;
    /**
     * método __construct()
     * instancia uma nova DataGrid
     */
    public function __construct()
    {
        parent::__construct();
        $this->class = 'tdatagrid_table';

        // instancia objeto TStyle
        // este estilo será utilizado para a tabela dadatagrid
        $style1 = new TStyle('tdatagrid_table');
        $style1->borderCollapse = 'separate';
        $style1->fontFamily      = 'arial,verdana,sans-serif';
    }
}
  
```

```
$style1->font_size      = '10pt';
$style1->border_spacing = '0pt';

// instancia objeto TStyle
// Este estilo será utilizado para os cabeçalhos dadatagrid
$style2 = new TStyle('tdatagrid_col');
$style2->font_size      = '10pt';
$style2->font_weight    = 'bold';
$style2->border_left    = '1px solid white';
$style2->border_top     = '1px solid white';
$style2->border_right   = '1px solid gray';
$style2->border_bottom  = '1px solid gray';
$style2->padding_top   = '1px';
$style2->background_color= '#CCCCCC';

// instancia objeto TStyle
// Este estilo será utilizado quando
// o mouse estiver sobre um cabeçalho dadatagrid
$style3 = new TStyle('tdatagrid_col_over');
$style3->font_size      = '10pt';
$style3->font_weight    = 'bold';
$style3->border_left    = '1px solid white';
$style3->border_top     = '2px solid orange';
$style3->border_right   = '1px solid gray';
$style3->border_bottom  = '1px solid gray';
$style3->padding_top   = '0px';
$style3->cursor         = 'pointer';
$style3->background_color= '#dcdcdc';

// exibe estilos na tela
$style1->show();
$style2->show();
$style3->show();
}
```

O método `addColumn()` será utilizado para adicionar uma coluna à listagem. Cada coluna será representada por um objeto do tipo `TGridColumn`, o qual criaremos em seguida. Um objeto `TGridColumn` irá conter informações como alinhamento, largura, rótulo e ação, dentre outros. Para adicionar uma coluna na DataGrid, iremos simplesmente adicionar, por meio de uma operação que constitui uma agregação, o objeto `TGridColumn` ao vetor `$columns`, propriedade da classe `TDataGrid`.

```
/**
 * método addColumn()
 * adiciona uma coluna à listagem
 * @param $object = objeto do tipo TGridColumn
 */
```

```
public function addColumn(TDataGridColumn $object)
{
    $this->columns[] = $object;
}
```

Da mesma forma que adicionamos colunas à listagem pela agregação, adicionaremos ações. Estas serão representadas por objetos do tipo `TDataGridAction` e irão conter informações como o ícone da ação e a função que será executada quando o usuário clicar sobre o link da ação. As ações serão armazenadas no vetor `$actions`.

```
/**
 * método addAction()
 * adiciona uma ação à listagem
 * @param $object = objeto do tipo TDataGridAction
 */
public function addAction(TDataGridAction $object)
{
    $this->actions[] = $object;
}
```

Em algumas operações, como, por exemplo, a exclusão de elementos, precisaremos recarregar a listagem. Nesses casos, poderemos utilizar o método `clear()`, cuja função será eliminar todas as linhas da listagem. Como a listagem é uma classe derivada de `TTable`, que, por sua vez, é derivada de `TElement`, as linhas da tabela estão armazenadas no vetor `$children`. Então, basta eliminarmos este vetor. Como a primeira linha representa o cabeçalho (`$children[0]`), preservaremos esta linha. Além disso, reiniciaremos a variável contadora de linhas (`$rowcount`).

```
/**
 * método clear()
 * elimina todas linhas de dados da DataGrid
 */
function clear()
{
    // faz uma cópia do cabeçalho
    $copy = $this->children[0];
    // inicializa o vetor de linhas
    $this->children = array();
    // acrescenta novamente o cabeçalho
    $this->children[] = $copy;
    // zera a contagem de linhas
    $this->rowcount = 0;
}
```

Quando já tivermos acrescentado as colunas e ações necessárias, a classe `TDataGrid` já poderá criar internamente a linha responsável por exibir o cabeçalho da listagem, porque já saberá quantas colunas ao total nossa listagem irá possuir. Nesse mo-

mento, o programador poderá executar o método `createModel()`, responsável por criar o cabeçalho das colunas e definir a estrutura da listagem. Para tanto, esse método acrescentará uma linha na tabela e primeiramente adicionará células para cada uma das ações. Em seguida, percorreremos cada uma das colunas da listagem (`$columns`) descobrindo as propriedades de cada uma por meio de seus métodos como `getName()`, `getLabel()` e `getAlign()`, os quais criaremos posteriormente, na classe `TGridColumn`. Para cada coluna, adicionaremos uma célula no cabeçalho. Cada coluna ainda poderá ter uma ação relacionada a si, o que significa que quando o usuário clicar exatamente sobre a coluna, uma determinada ação será executada, como, por exemplo, ordenar os resultados da listagem pela coluna. Nesses casos, quando a coluna possuir uma ação relacionada, alteraremos seu estilo quando o usuário passar o cursor sobre ela, destacando-a com uma linha laranja na parte superior (estilo `tdatagrid_col_over`).

```
/**  
 * método createModel()  
 * cria a estrutura da Grid, com seu cabeçalho  
 */  
public function createModel()  
{  
    // adiciona uma linha à tabela  
    $row = parent::addRow();  
  
    // adiciona células para as ações  
    if ($this->actions)  
    {  
        foreach ($this->actions as $action)  
        {  
            $celula = $row->addCell('');  
            $celula->class = 'tdatagrid_col';  
        }  
    }  
    // adiciona as células para os dados  
    if ($this->columns)  
    {  
        // percorre as colunas da listagem  
        foreach ($this->columns as $column)  
        {  
            // obtém as propriedades da coluna  
            $name  = $column->getName();  
            $label = $column->getLabel();  
            $align = $column->getAlign();  
            $width = $column->getWidth();  
  
            // adiciona a célula com a coluna  
            $celula = $row->addCell($label);  
        }  
    }  
}
```

```

$celula->class = 'tdatagrid_col';
$celula->align = $align;
$celula->width = $width;
// verifica se a coluna tem uma ação
if ($column->getAction())
{
    $url = $column->getAction();
    $celula->onmouseover = "this.className='tdatagrid_col_over';";
    $celula->onmouseout = "this.className='tdatagrid_col'";
    $celula->onclick = "document.location='$url'";
}
}
}
}

```

O método `addItem()` será utilizado para adicionar um objeto à DataGrid. Este objeto irá conter os dados que serão distribuídos em cada uma das colunas da listagem. O objeto poderá ser tão simples quanto um objeto `StdClass`, contendo somente propriedades, como também poderá ser um objeto Active Record, derivado da classe `TRecord`.

Para cada objeto adicionado na listagem teremos de adicionar as células necessárias. Para isso, primeiramente percorreremos o vetor de ações (`$actions`), adicionando uma célula para cada ação. Dentro desta célula, acrescentaremos um link (`TElement`), o qual poderá conter uma imagem (ícone) ou um texto (label). O link apontará ainda para o campo (`$field`) selecionado pelo usuário (`id`, `nome` etc.) por meio do método `setField()` da classe `TDataGridAction`.

Depois de adicionar as células para as ações, adicionaremos as células que irão conter os dados do objeto. Para tanto, obtemos algumas de suas propriedades por meio de métodos como `getName()`, `getAlign()` e `getWidth()`. Uma coluna poderá ter uma função “modificadora”, ou seja, uma função que modifica seu conteúdo. Obtemos esta função pelo método `getTransformer()`. Caso a coluna tenha alguma função, seu conteúdo será transformado pela mesma. Uma função é executada por meio de `call_user_func()`. Após tudo isso, a célula é adicionada pelo método `addCell()` à linha que contém a informação (`$data`).

```
/**  
 * método addItem()  
 * adiciona um objeto na grid  
 * @param $object = Objeto que contém os dados  
 */  
public function addItem($object)  
{  
    // cria um estilo com cor variável  
    $bcolor = ($this->rowcount % 2) == 0 ? '#ffffff' : '#e0e0e0';
```

```
// adiciona uma linha na DataGridView
$row = parent::addRow();
$row->bgcolor = $bgcolor;

// verifica se a listagem possui ações
if ($this->actions)
{
    // percorre as ações
    foreach ($this->actions as $action)
    {
        // obtém as propriedades da ação
        $url    = $action->serialize();
        $label  = $action->getLabel();
        $image  = $action->getImage();
        $field  = $action->getField();
        // obtém o campo do objeto que será passado adiante
        $key    = $object->$field;

        // cria um link
        $link = new TElement('a');
        $link->href ="{$url}&key={$key}";

        // verifica se o link será com imagem ou com texto
        if ($image)
        {
            // adiciona a imagem ao link
            $image=new TImage("app.images/$image");
            $link->add($image);
        }
        else
        {
            // adiciona o rótulo de texto ao link
            $link->add($label);
        }
        // adiciona a célula à linha
        $row->addCell($link);
    }
}
if ($this->columns)
{
    // percorre as colunas da DataGridView
    foreach ($this->columns as $column)
    {
        // obtém as propriedades da coluna
        $name    = $column->getName();
        $align   = $column->getAlign();
        $width   = $column->getWidth();
```

```

$function = $column->getTransformer();
$data     = $object->$name;
// verifica se há função para transformar os dados
if ($function)
{
    // aplica a função sobre os dados
    $data = call_user_func($function, $data);
}
// adiciona a célula na linha
$celula = $row->addCell($data);
$celula->align = $align;
$celula->width = $width;
}

}

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

```

6.4.2.2 TDataGridColumn

A classe `TDataGridColumn` será utilizada para representar as características que fazem parte de uma coluna em uma listagem. Para tanto, esta 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 as propriedades definidas pelo método construtor. Veja na Figura 6.29 a classe `TDataGridColumn`.

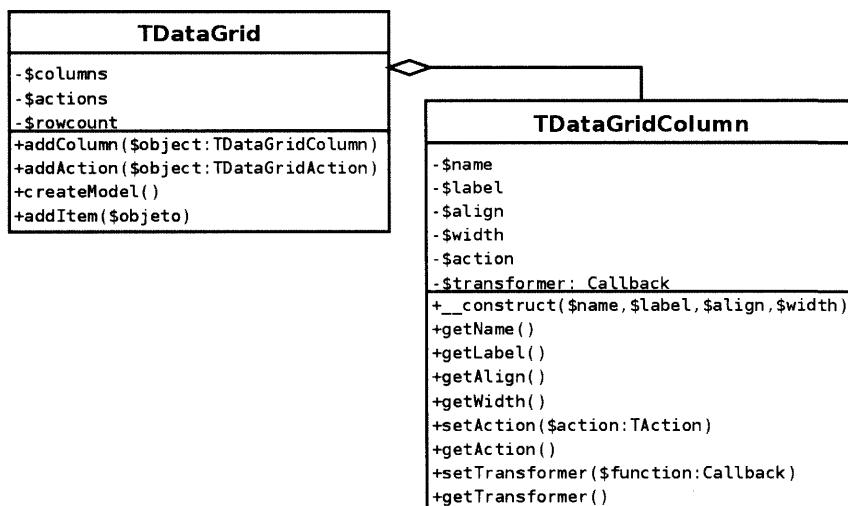


Figura 6.29 – Classe `TDataGridColumn`.

 TDataGridColumn.class.php

```
<?php
/**
 * class TDataGridColumn
 * representa uma coluna de uma listagem
 */
class TDataGridColumn
{
    private $name;
    private $label;
    private $align;
    private $width;
    private $action;
    private $transformer;

    /**
     * método __construct()
     * instancia uma coluna nova
     * @param $name = nome da coluna no banco de dados
     * @param $label = rótulo de texto que será exibido
     * @param $align = alinhamento da coluna (left, center, right)
     * @param $width = largura da coluna (em pixels)
     */
    public function __construct($name, $label, $align, $width)
    {
        // atribui os parâmetros às propriedades do objeto
        $this->name = $name;
        $this->label = $label;
        $this->align = $align;
        $this->width = $width;
    }

    /**
     * método getName()
     * retorna o nome da coluna no banco de dados
     */
    public function getName()
    {
        return $this->name;
    }

    /**
     * método getLabel()
     * retorna o nome do rótulo de texto da coluna
     */
}
```

```
public function getLabel()
{
    return $this->label;
}

/**
 * método getAlign()
 * retorna o alinhamento da coluna (left, center, right)
 */
public function getAlign()
{
    return $this->align;
}

/**
 * método getWidth()
 * retorna a largura da coluna (em pixels)
 */
public function getWidth()
{
    return $this->width;
}
```

O método `setAction()` será utilizado para definir uma ação que será executada sempre que o usuário clicar sobre o título da coluna. Esta ação será representada por um objeto do tipo `TAction`, criado no Capítulo 5. O método `getAction()` será utilizado para retornar esta ação na forma de URL pelo método `serialize()` da classe `TAction`. Assim, se indicarmos como ação da coluna o método `ordenar` da classe `PessoasList`, o método `getAction()` retornará “`?class=PessoasList&method=ordenar`”.

```
/**
 * método setAction()
 * define uma ação a ser executada quando o usuário
 * clica sobre o título da coluna
 * @param $action = objeto TAction contendo a ação
 */
public function setAction(TAction $action)
{
    $this->action = $action;
}

/**
 * método getAction()
 * retorna a ação vinculada à coluna
 */
public function getAction()
{
```

```
// verifica se a coluna possui ação
if ($this->action)
{
    return $this->action->serialize();
}
```

O método `setTransformer()` será utilizado para definir o nome de uma função ou de um método do PHP, ou definido pelo usuário, que será aplicado sobre cada um dos elementos listados naquela coluna, modificando-os de acordo com a necessidade (aplicando uma máscara, convertendo para maiúsculo etc.). O método `getTransformer()` retornará esta função.

```
/**
 * método setTransformer()
 * define uma função (callback) a ser aplicada sobre
 * todo dado contido nesta coluna
 * @param $callback = função do PHP ou do usuário
 */
public function setTransformer($callback)
{
    $this->transformer = $callback;
}

/**
 * método getTransformer()
 * retorna a função (callback) aplicada à coluna
 */
public function getTransformer()
{
    return $this->transformer;
}
?>
```

6.4.2.3 *TDataGridAction*

A classe `TDataGridAction` representará uma ação que pode ser executada pelo usuário sobre cada item da listagem. Esta classe será derivada da classe `TAction` e herdará todos os métodos já existentes nesta. A classe `TDataGridAction` apresentará ainda o método `setImage()` para definir a localização de uma imagem que será utilizada como ícone da ação; o método `setLabel()` definirá um rótulo de texto que será utilizado para a ação; o método `setField()` definirá o nome de uma propriedade do objeto (coluna do banco de dados) que desejamos passar como parâmetro quando a ação for executada, no formato `&key=<propriedade>`, no qual `<propriedade>` pode ser o ID, o nome ou uma outra propriedade qualquer. Veja na Figura 6.30 a classe `TDataGridAction`.

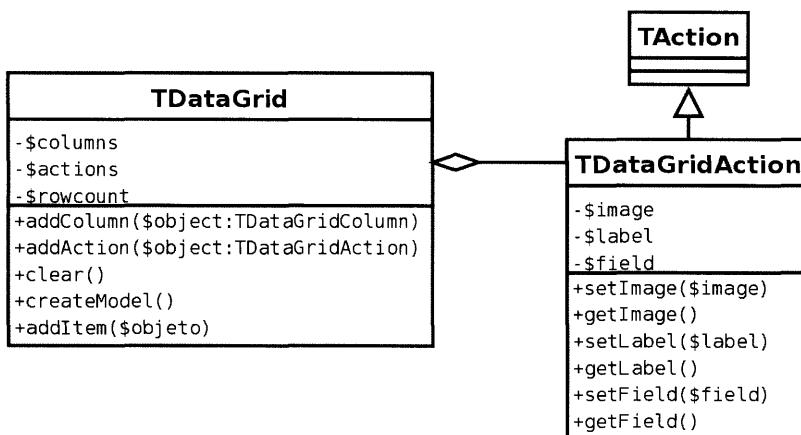


Figura 6.30 – Classe TDataGridAction.

TDataGridAction.class.php

```

<?php
/**
 * class TDataGridAction
 * representa uma ação de uma listagem
 */
class TDataGridAction extends TAction
{
    private $image;
    private $label;
    private $field;

    /**
     * método setImage()
     * atribui uma imagem à ação
     * @param $image = local do arquivo de imagem
     */
    public function setImage($image)
    {
        $this->image = $image;
    }
    /**
     * método getImage()
     * retorna a imagem da ação
     */
    public function getImage()
    {
        return $this->image;
    }
}
  
```

```
/***
 * método setLabel()
 * define o rótulo de texto da ação
 * @param $label = rótulo de texto da ação
 */
public function setLabel($label)
{
    $this->label = $label;
}
/***
 * método getLabel()
 * retorna o rótulo de texto da ação
 */
public function getLabel()
{
    return $this->label;
}
/***
 * método setField()
 * define o nome do campo do banco de dados que será passado juntamente com a ação
 * @param $field = nome do campo do banco de dados
 */
public function setField($field)
{
    $this->field = $field;
}
/***
 * método getField()
 * retorna o nome do campo de dados definido pelo método setField()
 */
public function getField()
{
    return $this->field;
}
?>
```

6.4.3 Exemplos

6.4.3.1 Listagem estruturada e estática

O programa a seguir é o nosso primeiro exemplo de utilização de uma DataGrid. Primeiramente, garantimos, por meio do método `__autoload()`, que os componentes sejam carregados assim que forem solicitados pela primeira vez (`new`). Em seguida, instanciamos um objeto da classe `TDataGrid` e partimos para a criação das colunas.

Esta listagem será composta de quatro colunas (Código, Nome, Endereço e Telefone), que são objetos `TGridColumn`. Tais colunas terão diferentes alinhamentos (terceiro parâmetro) e tamanhos (quarto parâmetro).

Ao instanciarmos um objeto `TGridColumn`, note que o primeiro parâmetro se refere ao nome das colunas no modelo de dados e o segundo parâmetro se refere ao rótulo de texto que será exibido no cabeçalho da listagem. Depois de criadas as colunas, partimos para a criação das ações da listagem, a qual terá duas ações – chamaremos a primeira de “Deletar” e a segunda de “Visualizar”. Instanciamos objetos `TDataGridAction`, indicando o método que será executado, o rótulo da ação, por meio do método `setLabel()`, o ícone, por meio do método `setImage()` e o parâmetro dos dados que será passado para a ação pelo método `setField()`. Depois de adicionarmos as ações na listagem, utilizamos o método `createModel()` para criar a estrutura básica da listagem (seu cabeçalho, suas colunas). Veja na Figura 6.31 o resultado deste exemplo.

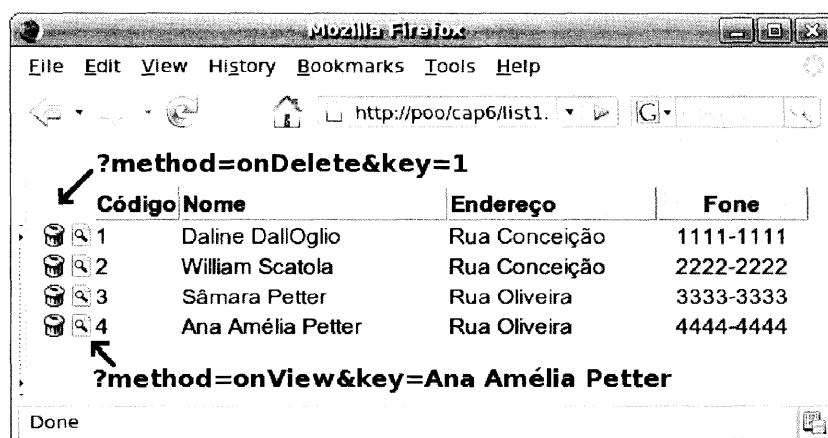


Figura 6.31 – Listagem estruturada e estática.

```
list1.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 objeto DataGrid
$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);

// adiciona as colunas à DataGrid
$datagrid->addColumn($codigo);
$datagrid->addColumn($nome);
$datagrid->addColumn($endereco);
$datagrid->addColumn($telefone);

// instancia duas ações da DataGrid
$action1 = new TDataGridAction('onDelete');
$action1->setLabel('Deletar');
$action1->setImage('ico_delete.png');
$action1->setField('codigo');

$action2 = new TDataGridAction('onView');
$action2->setLabel('Visualizar');
$action2->setImage('ico_view.png');
$action2->setField('nome');

// adiciona as ações à DataGrid
$datagrid->addAction($action1);
$datagrid->addAction($action2);

// cria o modelo da DataGrid, montando sua estrutura
$datagrid->createModel();
```

Neste exemplo ainda não utilizaremos banco de dados. Em seu lugar, adicionaremos vários objetos-padrão, derivados da classe `StdClass` do PHP. Estes objetos irão conter dados estáticos, declarados no próprio código do programa. Para adicionar estes objetos na listagem, utilizamos o método `addItem()` da `TDataGrid`. Note que o nome das propriedades dos objetos deve coincidir com o nome do campo indicado quando instanciamos os objetos `TDataGridColumn`, para que a classe saiba onde distribuir cada coluna do banco de dados. Em seguida, adicionamos a listagem em um objeto `TPage`, que controlará o fluxo de execução da página, ou seja, a execução dos métodos.

```
// adiciona um objeto padrão à DataGrid
$item = new StdClass;
$item->codigo  = '1';
$item->nome    = 'Daline Dall'oglio';
```

```
$item->endereco = 'Rua Conceição';
$item->fone     = '1111-1111';
$datagrid->addItem($item);

// adiciona um objeto padrão à DataGrid
$item = new StdClass;
$item->codigo  = '2';
$item->nome    = 'William Scatola';
$item->endereco = 'Rua Conceição';
$item->fone    = '2222-2222';
$datagrid->addItem($item);

// adiciona um objeto padrão à DataGrid
$item = new StdClass;
$item->codigo  = '3';
$item->nome    = 'Sâmara Petter';
$item->endereco = 'Rua Oliveira';
$item->fone    = '3333-3333';
$datagrid->addItem($item);

// adiciona um objeto padrão à DataGrid
$item = new StdClass;
$item->codigo  = '4';
$item->nome    = 'Ana Amélia Petter';
$item->endereco = 'Rua Oliveira';
$item->fone    = '4444-4444';
$datagrid->addItem($item);

// instancia uma página TPage
$page = new TPage;
// adiciona a DataGrid à página
$page->add($datagrid);
// exibe a página
$page->show();
```

Ao final, temos os métodos `onDelete()` e `onView()`, disparados pelas ações da listagem. O método `onDelete()` recebe, dentre seus parâmetros (que vêm da variável `$_GET`), o código da pessoa da listagem, definido pelo método `setField()` da classe `TDataGridAction`, e exibe uma mensagem de erro dizendo que aquele registro não poderá ser excluído. O método `onView()` receberá como parâmetro o nome da pessoa, também definido pelo método `setField()`, e exibirá uma mensagem de informação contendo o nome da pessoa da linha que teve a ação executada. A seguir temos o código-fonte e um exemplo. Quando o usuário clicar na ação **Deletar** sobre o registro da primeira linha, a seguinte mensagem será exibida em tela:

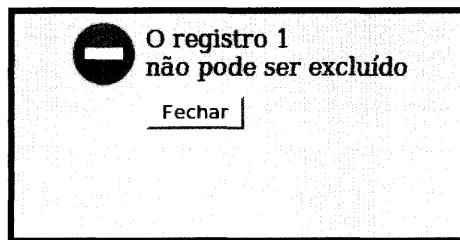


Figura 6.32 – Resultado da ação de clicar em **Deletar**.

```
/*
 * função onDelete()
 * executada quando o usuário clicar no botão excluir
 */
function onDelete($param)
{
    // obtém o parâmetro e exibe mensagem
    $key=$param['key'];
    new TMessage('error', "O registro $key <br> não pode ser excluído");
}

/*
 * função onView()
 * Executada quando o usuário clicar no botão visualizar
 */
function onView($param)
{
    // obtém o parâmetro e exibe mensagem
    $key=$param['key'];
    new TMessage('info', "O nome é: <br> $key");
}
?>
```

6.4.3.2 Listagem orientada a objetos e estática

Neste segundo exemplo da utilização da classe `TDataGrid` reescreveremos o código construído no exemplo anterior de forma orientada a objetos. Agora a página em si não será mais somente um objeto `$page`, como no exemplo anterior, mas será uma classe completa, a qual chamaremos de `PessoasList`. Esta classe terá uma propriedade chamada `$datagrid` que será um objeto do tipo `TDataGrid`. Toda estrutura da listagem é montada no método construtor. Veja que no momento de criar as ações (objetos `TDataGridAction`) não mais passamos somente o nome da função a ser executada, como no exemplo anterior. Por estarmos utilizando uma estrutura orientada a objetos, precisamos indicar o nome do método e o objeto ao qual ele pertence, formando um array de duas posições, como em `array($this, 'onDelete')`, o qual indica que será executado o método `onDelete()` do objeto atual. Veja na Figura 6.33 o resultado deste exemplo.

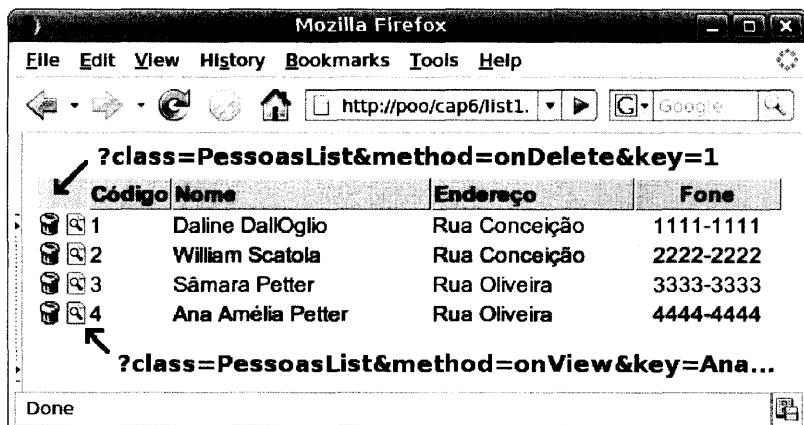


Figura 6.33 – Listagem orientada a objetos e estática.

list2.php

```
<?php
/*
 * função __autoload()
 * carrega uma classe quando ela é necessária,
 * ou seja, quando ela é instanciada 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);
    }
}
```

```
// adiciona as colunas à DataGrid
$this->datagrid->addColumn($codigo);
$this->datagrid->addColumn($nome);
$this->datagrid->addColumn($endereco);
$this->datagrid->addColumn($telefone);

// instancia duas ações da DataGrid
$action1 = new TDataGridAction(array($this, 'onDelete'));
$action1->setLabel('Deletar');
$action1->setImage('ico_delete.png');
$action1->setField('codigo');

$action2 = new TDataGridAction(array($this, 'onView'));
$action2->setLabel('Visualizar');
$action2->setImage('ico_view.png');
$action2->setField('nome');

// adiciona as ações à DataGrid
$this->datagrid->addAction($action1);
$this->datagrid->addAction($action2);

// cria o modelo da DataGrid, montando sua estrutura
$this->datagrid->createModel();

// adiciona a DataGrid à página
parent::add($this->datagrid);
}
```

Diferentemente do exemplo anterior, no qual adicionamos os dados juntamente à construção da listagem, neste exemplo deixamos para inserir os dados no momento da exibição da lista em tela, ou seja, quando da execução do seu método `show()`. Para não alterar o comportamento já existente do método `show()` da classe `TPage`, que é a classe da qual nossa classe `PessoalList` descende, executaremos, ao final deste método, o método `show()` da classe-pai, constituindo uma sobrescrita.

```
function show()
{
    $this->datagrid->clear();

    // adiciona um objeto padrão à DataGrid
    $item = new StdClass;
    $item->codigo = '1';
    $item->nome = 'Daline DallOglio';
    $item->endereco = 'Rua Conceição';
    $item->fone = '1111-1111';
    $this->datagrid->addItem($item);
```

```

// adiciona um objeto padrão à DataGridView
$item = new StdClass;
$item->codigo = '2';
$item->nome = 'William Scatola';
$item->endereco = 'Rua Conceição';
$item->fone = '2222-2222';
$this->datagrid->addItem($item);

// adiciona um objeto padrão à DataGridView
$item = new StdClass;
$item->codigo = '3';
$item->nome = 'Sâmara Petter';
$item->endereco = 'Rua Oliveira';
$item->fone = '3333-3333';
$this->datagrid->addItem($item);

// adiciona um objeto padrão à DataGridView
$item = new StdClass;
$item->codigo = '4';
$item->nome = 'Ana Amélia Petter';
$item->endereco = 'Rua Oliveira';
$item->fone = '4444-4444';
$this->datagrid->addItem($item);

parent::show();
}

```

As funções `onDelete()` e `onView()` passam a ser métodos da classe `PessoalList`, mas não têm seu comportamento alterado. Ao final do programa instanciamos um objeto da classe `PessoalList` e o exibimos em tela. Na imagem a seguir, conferimos o código-fonte e a mensagem que é exibida ao usuário quando este clicar na ação **Visualizar** sobre o terceiro registro.

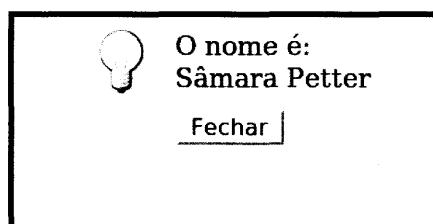


Figura 6.34 – Resultado da ação de clicar em **Visualizar**.

```

/*
 * função onDelete()
 * executada quando o usuário clicar no botão excluir
 */

```

```
function onDelete($param)
{
    // obtém o parâmetro e exibe mensagem
    $key=$param['key'];
    new TMessage('error', "O registro $key <br> não pode ser excluído");
}

/*
 * função onView()
 * executada quando o usuário clicar no botão visualizar
 */
function onView($param)
{
    // obtém o parâmetro e exibe mensagem
    $key=$param['key'];
    new TMessage('info', "O nome é: <br> $key");
}
}

$page = new PessoasList;
$page->show();
?>
```

6.4.3.3 Listagem estruturada com modificadores de exibição

Nos dois exemplos anteriores de DataGrid, vinhemos utilizando dados estáticos, criando objetos StdClass para posteriormente adicionar na listagem. No exemplo a seguir, criaremos uma DataGrid cujo conteúdo será lido do banco de dados por meio de nossa classe **TRepository**, criada no Capítulo 4.

Quando trabalhamos com exibição de informações provindas do banco de dados, freqüentemente precisamos alterar o seu formato de exibição. Isto ocorre porque nem sempre armazenamos a informação no banco de dados da mesma forma que ela deverá ser exibida ao usuário. Datas são freqüentemente armazenadas no formato americano (aaaa-mm-dd), ao passo que são exibidas no formato brasileiro. Em outros casos armazenamos somente o índice da opção selecionada, como em radio buttons ou em combo-boxes, sendo que exibimos normalmente uma palavra no lugar daquele índice.

Para tratar esses casos, a classe **TGridColumn** oferece o método **set_transformer()**, o qual recebe o nome de uma função que é aplicada a todos os elementos de uma coluna. Esta função pode ser utilizada para transformar o conteúdo do elemento, alterando o seu formato, sua máscara, convertendo para minúsculo, maiúsculo, dentre outros. Veja na Figura 6.35 o resultado deste exemplo.

Código	Nome	Endereço	Data Nasc	Sexo
1	PABLO DALL'OGLIO	Conceicao	24/09/1980	Masculino
2	FABIO LOCATELLI	Rua Marvin	20/03/1976	Masculino
3	CARLOS RANZI	Rua Francisco	29/06/1980	Masculino
4	WILLIAM SCATOLA	Rua Fontana	15/04/1985	Masculino

Figura 6.35 – Listagem estruturada com modificadores de exibição.

Neste exemplo criaremos uma DataGrid com cinco colunas (Código, Nome, Endereço, Data de Nascimento e Sexo). Note que estamos definindo a `strtoupper()`, função interna do PHP responsável por converter uma string para maiúsculo, como modificadora (transformer) para a coluna `$nome`. A função `conv_data_to_br()` será aplicada sobre a coluna `$datanasc`. Esta função será responsável por converter uma data que está armazenada no formato `aaaa-mm-dd` no banco de dados para o formato `dd/mm/aaaa`. Já na coluna `$sexo`, aplicaremos a função `get_sexo()`, a qual atuará sobre o valor que é lido do banco de dados (`M,F`) e retornará a string por extenso (Masculino, Feminino).

list3.php

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

/*
 * função get_sexo()
 * converte um caractere (M,F) para extenso
 * @param $sexo = M ou F (Masculino/Feminino)
 */
function get_sexo($sexo)
{
    switch ($sexo)
    {
        case 'M':
            return 'Masculino';
            break;
        case 'F':
            return 'Feminino';
            break;
    }
}

// declara a classe PessoaRecord
class PessoaRecord extends TRecord {}

// instancia objeto DataGrid
$datagrid = new TDataGrid;

// instancia as colunas da DataGrid
$codigo = new TDataGridColumn('id',      'Código',   'right', 50);
$nome   = new TDataGridColumn('nome',    'Nome',     'left', 160);
$endereco = new TDataGridColumn('endereco', 'Endereço', 'left', 140);
$datanasc = new TDataGridColumn('datanasc', 'Data Nasc', 'left', 100);
$sexo    = new TDataGridColumn('sexo',     'Sexo',     'center', 100);

// aplica as funções para transformar as colunas
$nome->setTransformer('strtoupper');
$datanasc->setTransformer('conv_data_to_br');
$sexo->setTransformer('get_sexo');

// adiciona as colunas à DataGrid
$datagrid->addColumn($codigo);
$datagrid->addColumn($nome);
$datagrid->addColumn($endereco);
```

```
$datagrid->addColumn($datanasc);
$datagrid->addColumn($sexo);

// cria o modelo da DataGrid, montando sua estrutura
$datagrid->createModel();
```

Neste exemplo estamos listando os registros da tabela pessoa, alimentada pelo formulário de pessoas, criado no capítulo anterior. Para tanto, estamos abrindo uma transação com o banco de dados pg_livro e lendo todos os objetos Pessoa por meio do método load() da classe TRepository. A partir do array de objetos retornados (\$pessoas) adicionaremos cada um dos objetos na DataGrid por meio do método addItem(). Note que estamos utilizando um critério para definir a ordenação dos registros (por id).

```
// obtém objetos do banco de dados
try
{
    // inicia transação com o banco 'pg_livro'
    TTransaction::open('pg_livro');

    // instancia um repositório para Pessoa
    $repository = new TRepository('Pessoa');
    // cria um critério, definindo a ordenação
    $criteria = new TCriteria;
    $criteria->setProperty('order', 'id');
    // carrega os objetos $pessoas
    $pessoas = $repository->load($criteria);
    foreach ($pessoas as $pessoa)
    {
        // adiciona o objeto na DataGrid
        $datagrid->addItem($pessoa);
    }
    // finaliza a transação
    TTransaction::close();
}

catch (Exception $e)      // em caso de exceção
{
    // exibe a mensagem gerada pela exceção
    new TMessage('error', $e->getMessage());
    // desfaz todas alterações no banco de dados
    TTransaction::rollback();
}

// instancia uma página TPage
$page = new TPage;
// adiciona a DataGrid à página
$page->add($datagrid);
// exibe a página
$page->show();

?>
```

6.4.3.4 Listagem orientada a objetos com banco de dados

Neste exemplo, além de trabalharmos com elementos do banco de dados, criaremos uma listagem com ações sobre esses elementos. Nossa listagem terá uma opção para visualizar um registro e outra para excluir. Já tínhamos criado opções similares nos primeiros exemplos, porém com dados estáticos. Neste caso, iremos interagir dinamicamente com a lista.

Para isso, criaremos duas ações: uma chamada “Deletar”, a qual executará o método `onDelete()` e terá como ícone a imagem `ico_delete.png`; e outra chamada “Visualizar”, a qual executará o método `onView()` e terá a imagem `ico_view.png` como ícone. Na Figura 6.36 você confere a listagem, bem como a mensagem exibida quando o usuário clica sobre cada uma das ações.

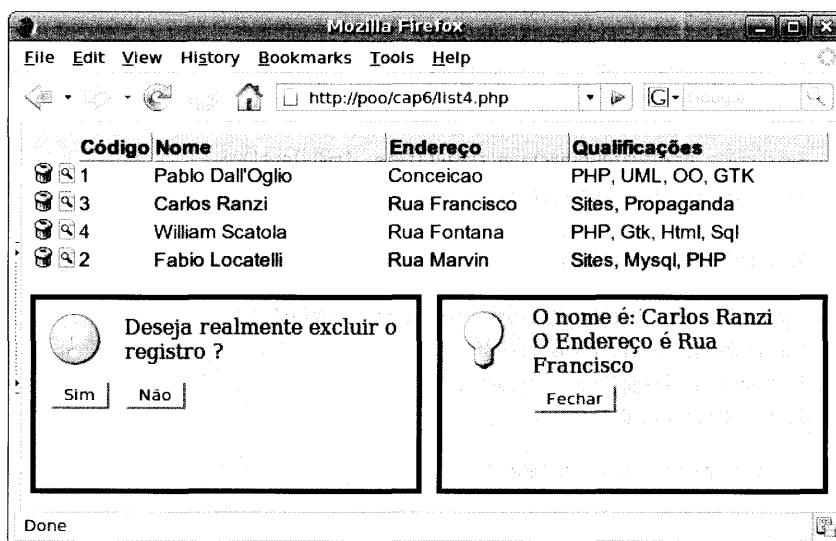


Figura 6.36 – Listagem orientada a objetos com banco de dados.

`list4.php`

```
<?php
// método __autoload()...

// declara a classe PessoaRecord
class PessoaRecord extends TRecord {}

/*
 * classe PessoasList
 */
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('id',           'Código',      'left',  50);
    $nome    = new TDataGridColumn('nome',         'Nome',       'left', 180);
    $endereco = new TDataGridColumn('endereco',   'Endereço',   'left', 140);
    $qualifica= new TDataGridColumn('qualifica','Qualificações','left', 200);

    // adiciona as colunas à DataGrid
    $this->datagrid->addColumn($codigo);
    $this->datagrid->addColumn($nome);
    $this->datagrid->addColumn($endereco);
    $this->datagrid->addColumn($qualifica);

    // instancia duas ações da DataGrid
    $action1 = new TDataGridAction(array($this, 'onDelete'));
    $action1->setLabel('Deletar');
    $action1->setImage('ico_delete.png');
    $action1->setField('id');

    $action2 = new TDataGridAction(array($this, 'onView'));
    $action2->setLabel('Visualizar');
    $action2->setImage('ico_view.png');
    $action2->setField('id');

    // adiciona as ações à DataGrid
    $this->datagrid->addAction($action1);
    $this->datagrid->addAction($action2);

    // cria o modelo da DataGrid, montando sua estrutura
    $this->datagrid->createModel();

    // adiciona a DataGrid à página
    parent::add($this->datagrid);
}
```

Como nossa listagem terá várias ações, ao final de cada ação executada precisaremos recarregar a listagem. Para isso, criaremos o método `onReload()`. Este será sempre executado no método `show()` da página, mas também será executado sempre após uma ação `onDelete()` ou `onView()`. O método `onReload()` abre uma transação com o banco de dados, faz a leitura dos objetos a partir da tabela `Pessoa` e adiciona estes objetos na `DataGrid`.

```
/*
 * função onReload()
 * carrega a DataGrid com os objetos do banco de dados
 */
function onReload()
{
    // inicia transação com o banco 'pg_livro'
    TTransaction::open('pg_livro');

    // instancia um repositório para Pessoa
    $repository = new TRepository('Pessoa');
    // cria um critério, definindo a ordenação
    $criteria = new TCriteria;
    $criteria->setProperty('order', 'id');
    // carrega os objetos $pessoas
    $pessoas = $repository->load($criteria);
    $this->datagrid->clear();
    if ($pessoas)
    {
        foreach ($pessoas as $pessoa)
        {
            // adiciona o objeto na DataGrid
            $this->datagrid->addItem($pessoa);
        }
    }
    // finaliza a transação
    TTransaction::close();
    $this->loaded = true;
}
```

O método `onDelete()` será executado sempre que o usuário clicar no ícone “deletar” sobre um registro. Este método abrirá um diálogo de questionamento ao usuário, perguntando se o mesmo tem certeza que deseja excluir o registro. A ação para a resposta “sim” será a execução do método `Delete()`, passando como parâmetro a chave (`key`) do registro.

```
/*
 * função onDelete()
 * executada quando o usuário clicar no botão excluir
 */
function onDelete($param)
{
    // obtém o parâmetro e exibe mensagem
    $key=$param['key'];

    $action1 = new TAction(array($this, 'Delete'));
    $action2 = new TAction(array($this, 'teste'));
```

```

$action1->setParameter('key', $key);
$action2->setParameter('key', $key);

new TQuestion('Deseja realmente excluir o registro?', $action1, $action2);
}

```

Caso o usuário responda afirmativamente a pergunta, o método `Delete()` será executado; ele abrirá uma transação com o banco de dados, carregará o registro cujo ID é identificado pela variável `$key` e o eliminará do banco de dados pelo método `delete()`. Em seguida, a DataGrid será recarregada pelo método `onReload()`.

```

/*
 * função Delete()
 * exclui um registro
 */
function Delete($param)
{
    // obtém o parâmetro e exibe mensagem
    $key=$param['key'];
    // inicia transação com o banco 'pg_livro'
    TTransaction::open('pg_livro');

    $pessoa = new PessoaRecord($key);
    $pessoa->delete();
    // finaliza a transação
    TTransaction::close();
    new TMessage('info', "Registro excluído com sucesso");
    $this->onReload();
}

```

Ainda temos o método `onView()`, executado sempre que o usuário clicar no botão **Visualizar**. Este método abre uma conexão com o banco de dados, lê o registro identificado pelo parâmetro `$key` e abre uma mensagem ao usuário por meio da classe `TMessage`, exibindo informações como o nome e o endereço da pessoa.

```

/*
 * função onView()
 * Executada quando o usuário clicar no botão visualizar
 */
function onView($param)
{
    // obtém o parâmetro e exibe mensagem
    $key=$param['key'];
    // inicia transação com o banco 'pg_livro'
    TTransaction::open('pg_livro');

    $pessoa = new PessoaRecord($key);

```

```
// finaliza a transação
TTransaction::close();
$mensagem = "O nome é: $pessoa->nome<br>";
$mensagem.= "O Endereço é $pessoa->endereco";
new TMessage('info', $mensagem);
$this->onReload();
}
```

Por fim, temos o método `show()` da página. Caso o método `onReload()` ainda não tenha sido executado, ele será nesse momento.

```
/*
 * função show()
 * executada quando o usuário clicar no botão excluir
 */
function show()
{
    if (!$this->loaded)
    {
        $this->onReload();
    }
    parent::show();
}

$page = new PessoasList;
$page->show();
?>
```

6.4.3.5 Listagem orientada a objetos com ordenação

Uma listagem somente já é bastante útil, mas o usuário da aplicação geralmente vai querer visualizar o conteúdo sob diferentes ordenações, ora ordenada por código, ora por nome, ora por algum outro campo. Para permitir ordenações na listagem, criamos o método `setAction()` da classe `TGridColumn`. Cada coluna da listagem poderá ter uma ação, o que significa que, ao posicionar o cursor sobre a coluna e dar um clique, o usuário estará provocando esta ação.

A listagem demonstrada na Figura 6.37 terá quatro colunas (Código, Nome, Endereço e Qualificações), sendo que as colunas Código e Nome são ordenáveis. Para isso, criamos duas ações (`$action1` e `$action2`). Estas ações executarão o método `onReload()`, passando o parâmetro `order`, contendo o valor `id` para a coluna Código e `nome` para a coluna Nome.

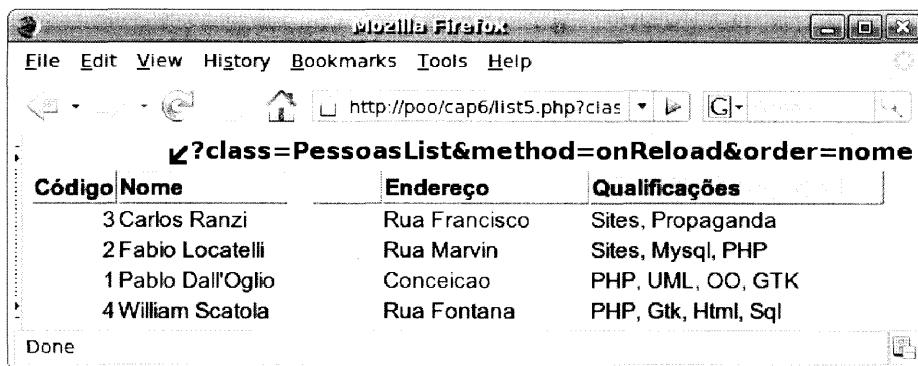


Figura 6.37 – Listagem orientada a objetos com ordenação.

list5.php

```
<?php
// __autoload...

// declara a classe PessoaRecord
class PessoaRecord extends TRecord {}

/*
 * classe PessoasList
 */
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 TGridColumn('id', 'Código', 'right', 50);
        $nome = new TGridColumn('nome', 'Nome', 'left', 180);
        $endereco = new TGridColumn('endereco', 'Endereço', 'left', 140);
        $telefone = new TGridColumn('qualifica', 'Qualificações', 'left', 200);

        $action1 = new TAction(array($this, 'onReload'));
        $action1->setParameter('order', 'id');

        $action2 = new TAction(array($this, 'onReload'));
        $action2->setParameter('order', 'nome');
        $codigo->setAction($action1);
        $nome->setAction($action2);
    }
}
```

```
// adiciona as colunas à DataGrid
$this->datagrid->addColumn($codigo);
$this->datagrid->addColumn($nome);
$this->datagrid->addColumn($endereco);
$this->datagrid->addColumn($telefone);

// cria o modelo da DataGrid, montando sua estrutura
$this->datagrid->createModel();

// adiciona a DataGrid à página
parent::add($this->datagrid);
}
```

O método `onReload()`, como no exemplo anterior, será responsável por abrir uma transação com o banco de dados, carregar os objetos da tabela `Pessoa` e adicioná-los na `DataGrid`. A diferença básica é que, desta vez, estamos identificando a ordenação dos resultados por meio do método `setProperty()` do objeto `$criteria`, passando o parâmetro `$order`, recebido via `$_GET` sempre que o usuário clicar sobre uma coluna, disparando este método.

```
/*
 * função onReload()
 * carrega a DataGrid com os objetos do banco de dados
 */
function onReload($param = NULL)
{
    $order = $param['order'];

    // inicia transação com o banco 'pg_livro'
    TTransaction::open('pg_livro');
    // instancia um repositório para Pessoa
    $repository = new TRepository('Pessoa');
    // retorna todos objetos que satisfazem o critério
    $criteria = new TCriteria;
    $criteria->setProperty('order', $order);
    $pessoas = $repository->load($criteria);

    if ($pessoas)
    {
        $this->datagrid->clear();
        foreach ($pessoas as $pessoa)
        {
            // adiciona o objeto na DataGrid
            $this->datagrid->addItem($pessoa);
        }
    }
}
```

```
// finaliza a transação
TTransaction::close();
$this->loaded = true;
}

/*
* função show()
* Executada quando o usuário clicar no botão excluir
*/
function show()
{
    if (!$this->loaded)
    {
        $this->onReload();
    }
    parent::show();
}
}

$page = new PessoasList;
$page->show();
?>
```

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 Smalltalk 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

representação pode ser uma página HTML, por exemplo. A camada de visualização (View) deve ser responsável somente pela visualização, não exercendo qualquer tipo de controle no fluxo de execução da aplicação, tampouco deverá conter lógica de negócios. Controller significa controle. Nesta camada teremos a manipulação dos inputs do usuário, sua interpretação e a execução das tarefas correspondentes. Esta camada é formada por um conjunto de objetos que recebem informações da View e tratam de atualizar o modelo de dados (Model) de acordo com a ação do usuário.

A separação da aplicação nesses três aspectos traz uma série de vantagens ao desenvolvedor. A separação entre o modelo de dados (Model) e a visualização (View), por exemplo, permite ao desenvolvedor reutilizar um mesmo objeto de modelo em diversas visualizações diferentes. Para ficar mais claro, imagine uma listagem de clientes e uma listagem de compras de um cliente, ou uma listagem de clientes por cidade. Todas são visualizações diferentes, mas tratam de um objeto do modelo de negócios em comum: o cliente. A camada de visualização (View) deve se preocupar com a disposição de objetos, com a organização visual, ao passo que o modelo deve se preocupar com regras de negócios e interação com o banco de dados.

As três camadas são distintas, porém interagem umas com as outras da maneira demonstrada pela Figura 7.1. A seta indica uma relação de dependência. As camadas View e Controller dependem do Modelo, porém este é totalmente independente das demais, no sentido de que o Modelo não faz referência a objetos das duas outras camadas. View e Controller dependem uma da outra: a camada de controle depende dos dados provindos da View e das ações disparadas pela mesma; a View aciona a camada Controller e aguarda o retorno necessário.

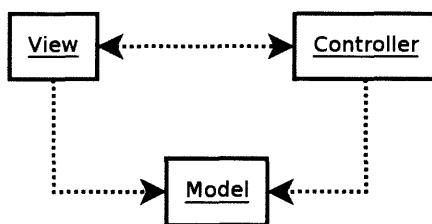


Figura 7.1 – Modelo MVC.

Um sistema MVC clássico terá uma classe Controller para cada View existente, mas esta abordagem não é a única existente. Alguns frameworks freqüentemente mesclam View e Controller na mesma camada, deixando-as diretamente vinculadas.

7.1.2 Pacotes

O PHP ainda não suporta o conceito de pacotes como em outras linguagens como Java. A implementação de Name Spaces estava no planejamento da versão 5, mas acabou

sendo removida, o que não impede, entretanto, que organizemos nossas classes sob certos aspectos. Você deve ter notado que nos capítulos anteriores criamos um certo conjunto de classes com objetivos diferentes e armazenamos essas classes dentro de pastas do sistema de arquivos. As pastas são uma forma primitiva mas funcional para organização de classes. Na Figura 7.2, vemos um resumo das pastas criadas e de algumas classes que nelas se encontram. Algumas pastas porém, como `app.control` e `app.model`, serão criadas somente no decorrer deste capítulo.

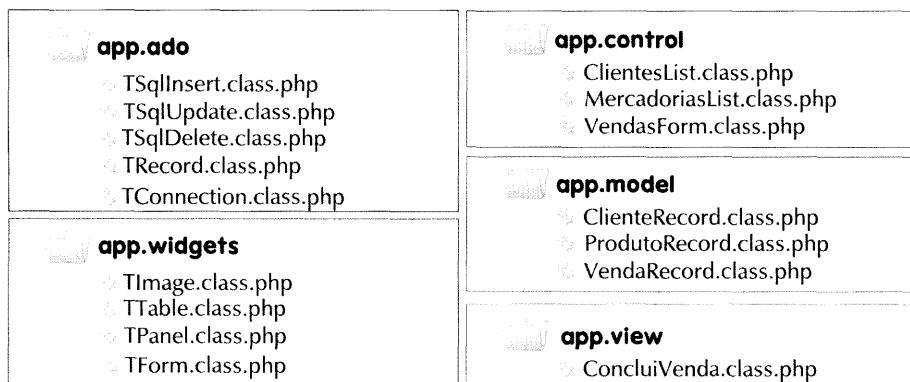


Figura 7.2 – Estrutura de pacotes.

Você também deve ter notado que utilizamos muito a função `__autoload()` do PHP, executada sempre que uma classe é instanciada pelo operador `new`. Nesse momento, tal função vasculhava as pastas para tentar localizar a classe requerida, para então carregá-la na memória. Quando desenvolvemos a aplicação, é relativamente interessante termos as classes distribuídas ao longo do sistema de arquivos. Isto facilita a localização e a manutenção destes.

No entanto, quando distribuímos uma aplicação, uma quantidade grande de arquivos pode se tornar difícil de gerenciar. Nestes casos, o PHP nos permite empacotar nossas classes. Isto se dá facilmente porque a linguagem possui certos *wrappers* ou empacotadores, que são utilizados como protocolos no momento de acessarmos recursos externos, principalmente arquivos. Um dos *wrappers* mais conhecidos é o `zlib`. Este empacotador nos permite acessar arquivos compactados por utilitários como o comando `gzip` do Linux. No exemplo a seguir, estamos entrando na pasta `app.widgets` e juntando todas as classes no arquivo `app.widgets.php`. Posteriormente, vamos compactar este arquivo com o utilitário `gzip`, gerando o arquivo `app.widgets.php.gz`.

```
cd app.widgets
cat *.php > app.widgets.php
gzip app.widgets.php
```

A partir desse momento, o arquivo `app.widgets.php.gz` é um arquivo compactado que contém todas as classes da pasta `app.widgets`. Neste caso, para incluir de uma vez

só todas as classes nele contidas dentro de nossa aplicação (`TTable`, `TForm`, `TDataGridView` etc.), bastaria digitar o seguinte comando no início do programa. Veja que o wrapper utilizado se chama `compress.zlib`. Este automaticamente descompacta o arquivo e disponibiliza as classes para a aplicação.

```
<?php  
include "compress.zlib://app.widgets.php.gz";  
?>
```

Dessa forma, podemos vir a compactar um conjunto de arquivos, geralmente separados por pasta, em um único arquivo com a extensão `.gz`, facilitando assim a distribuição da aplicação. Em vez de vasculharmos o sistema de arquivos classe por classe, poderíamos carregar o arquivo empacotado contendo todas as classes de uma única vez. Claro que essa abordagem possui alguns pontos negativos, principalmente relativos ao consumo de memória, tendo em vista que estaremos carregando na memória todas as classes daquele pacote. Não podemos, no entanto, generalizar; temos de analisar cada caso individualmente. Em muitos casos, nossas classes não passam de alguns poucos milhares de linhas, o que, em termos de memória, não representa muito.

No nosso caso em específico, o arquivo com todas as classes da pasta `app.widgets` ficou em torno de 12kb, ou seja, muito pouco em se tratando de memória. Outro fato que devemos analisar também é o acesso ao disco, que representa também um tempo grande de acesso se comparado ao acesso à memória RAM. Ademais, neste caso, em que carregamos todas as classes empacotadas de uma única vez, pouparamos vários acessos a disco (`include`) e disponibilizamos todas as classes para a aplicação.

7.1.3 Internacionalização e Singleton Pattern

Um dos recursos presentes em quase toda aplicação de grande porte é a internacionalização, técnica que permite que nossa aplicação adapte facilmente sua interface a um certo idioma, desde que sigamos uma determinada convenção no momento de apresentar as strings em tela. O PHP oferece suporte à biblioteca `get_text`, a qual é especializada nesse assunto e possui diversas ferramentas que podem ser utilizadas em conjunto. Os termos traduzidos pela biblioteca `get_text` ficam, porém, localizados externamente à aplicação no sistema de arquivos. Em nossa aplicação, implementaremos a internacionalização de maneira própria, utilizando uma classe. Esta classe irá conter uma lista (array) de todos os termos utilizados pela nossa aplicação em todos os idiomas por ela suportados. Este array será criado no método construtor da classe, que se chamará `TTranslation`. Esta ainda apresentará um método para definir a linguagem a ser utilizada para traduções – `setLanguage()` – e um método que recebe uma palavra em inglês e a traduz para o idioma de destino – `Translate()`.

Não faz sentido instanciarmos vários objetos da classe `Translation` ao longo da aplicação sempre que quisermos traduzir um termo. Deveríamos ter disponível somente uma única instância desta classe para toda a aplicação e faríamos uso dela sempre que necessário. No entanto, dificilmente conseguíramos visualizar este objeto ao longo de diferentes partes da aplicação, como classes, métodos e funções sem tornar tal instância global.

No desenvolvimento de aplicações, muitas vezes é necessário compartilhar algumas informações no domínio da aplicação, tornando-as visíveis dentro de diferentes contextos, em diferentes classes no sistema. Nessas ocasiões, somos tentados a utilizar um recurso chamado variáveis globais, que são fáceis de utilizar e disponibilizam seu conteúdo de forma indiscriminada a toda aplicação. Isto significa que qualquer código de qualquer classe pode alterar uma variável global, armazenando valores inconsistentes ou mesmo excluindo todo o seu conteúdo, sendo que, às vezes, outras partes da aplicação esperam que ela tenha determinada informação que não existe mais.

7.1.3.1 Singleton Pattern

Os motivos anteriormente descritos são apenas alguns dos quais tornam o uso de variáveis globais condenável dentro do paradigma orientado a objetos, por violar alguns conceitos fundamentais como o encapsulamento. Para esses casos, existe uma solução já estudada e catalogada – o pattern Singleton. Este pattern permite que um objeto fique disponível para toda a aplicação e que o mesmo tenha uma interface de encapsulamento.

A estrutura do Singleton garante que existirá uma única instância desse objeto e que esta instância possa ser obtida a qualquer momento. Esta solução, que parece mágica à primeira vista, é relativamente simples. O objeto é armazenado em uma propriedade estática da classe (neste caso chamada `$instance`), que também é privada. Para que só exista uma instância deste objeto disponível na aplicação, seu método construtor é marcado como `private`. Desta forma, o único local de onde poderá se instanciar um objeto Singleton será dentro da própria classe. No lugar do método construtor, cria-se um método estático responsável por instanciar o objeto, ao mesmo tempo em que este método verifica se o objeto já foi instanciado alguma vez, impedindo a sobreposição e a perda de informações. Nesta classe criada, o método `getInstance()` será responsável por retornar a instância única da classe.

Veja que no método construtor da classe, estamos criando um grande array contendo os termos utilizados pela aplicação em todos os idiomas possíveis (inglês, português e italiano).

 **TTranslation.class.php**

```
<?php
/**
 * classe TTranslation
 * classe utilitária para tradução de textos
 */
class TTranslation
{
    private static $instance;      // instância de TTranslation
    private $lang;                 // linguagem destino

    /**
     * método __construct()
     * instancia um objeto TTranslation
     */
    private function __construct()
    {
        $this->messages['en'][] = 'Function';
        $this->messages['en'][] = 'Table';
        $this->messages['en'][] = 'Tool';

        $this->messages['pt'][] = 'Função';
        $this->messages['pt'][] = 'Tabela';
        $this->messages['pt'][] = 'Ferramenta';

        $this->messages['it'][] = 'Funzione';
        $this->messages['it'][] = 'Tabelle';
        $this->messages['it'][] = 'Strumento';
    }
}
```

Veja que o método `getInstance()` verifica se realmente a variável `$instance` está vazia antes de instanciar um objeto. Desta forma, somente da primeira vez que o método for executado a instância será criada. Nas vezes posteriores, a variável estática `$instance` somente será retornada.

```
/**
 * método getInstance()
 * retorna a única instância de TTranslation
 */
public static function getInstance()
{
    // se não existe instância ainda
    if (empty(self::$instance))
    {
        // instancia um objeto
        self::$instance = new TTranslation;
```

```
    }
    // retorna a instância
    return self::$instance;
}
```

O método `setLanguage()` será responsável por definir o idioma de destino a ser utilizado para as traduções. Note que este método atua sobre a instância da classe, retornada pelo método `getInstance()`, e então define o valor do atributo `$lang`.

```
/**
 * método setLanguage()
 * define a linguagem a ser utilizada
 * @param $lang = linguagem (en,pt,it)
 */
public static function setLanguage($lang)
{
    $instance = self::getInstance();
    $instance->lang = $lang;
}
```

O método `getLanguage()`, assim como o método `setLanguage()`, atua sobre a instância única retornada pelo método `getInstance()`. Este método retorna a linguagem atual utilizada nas traduções.

```
/**
 * método getLanguage()
 * retorna a linguagem atual
 */
public static function getLanguage()
{
    $instance = self::getInstance();
    return $instance->lang;
}
```

O método `Translate()` recebe uma palavra e a traduz para o idioma de destino. Para isso, ele obtém a instância atual do objeto `Translation`, busca no array de palavras (`$messages`) pela palavra em inglês por meio da função `array_search()`, descobre seu índice (`$key`) e, a partir de então, retorna a ocorrência dessa palavra no mesmo array de palavras (`$messages`), mas desta vez indexado pelo idioma de destino (`$language`).

```
/**
 * método Translate()
 * traduz uma palavra para a linguagem definida
 * @param $word = Palavra a ser traduzida
 */
public function Translate($word)
{
```

```

// obtém a instância atual
$instance = self::getInstance();
// busca o índice numérico da palavra dentro do vetor
$key = array_search($word, $instance->messages['en']);

// obtém a linguagem para tradução
$language = self::getLanguage();
// retorna a palavra traduzida
// vetor indexado pela linguagem e pela chave
return $instance->messages[$language][$key];
}

} // fim da classe TTranslation

```

Para facilitar a tradução das palavras, criamos a função `_t()`. Sem ele, o programador teria de digitar `TTranslation::Translate('palavra')` sempre que quisesse obter o termo traduzido. Como geralmente uma aplicação possui muitas strings, criamos uma função para a qual o programador precisa digitar o mínimo possível para obter a palavra traduzida.

```

/**
 * método _t()
 * fachada para o método Translate da classe Translation
 * @param $word = Palavra a ser traduzida
 */
function _t($word)
{
    return TTranslation::Translate($word);
}
?>

```

Para exemplificar a classe recém-criada, criamos o programa a seguir, cujo objetivo é traduzir algumas palavras para os idiomas português e italiano.

trans.php

```

<?php
// inclui a classe TTranslation
include_once 'app.widgets/TTranslation.class.php';

// define a linguagem como português
TTranslation::setLanguage('pt');
echo "Em Português:<br>\n";

// imprime palavras traduzidas
echo _t('Function') . "<br>\n";
echo _t('Table') . "<br>\n";
echo _t('Tool') . "<br>\n";

```

```
// define a linguagem como italiano
TTranslation::setLanguage('it');
echo "Em Italiano:<br>\n";

// imprime palavras traduzidas
echo _t('Function') . "<br>\n";
echo _t('Table') . "<br>\n";
echo _t('Tool') . "<br>\n";
?>
```

 **Resultado:**

Em Português:

Função

Tabela

Ferramenta

Em Italiano:

Funzione

Tabelle

Strumento

7.1.4 Seções e Registry Pattern

Uma requisição de páginas é a forma mais simples de solicitar a execução de um determinado programa. Uma página pode ser requisitada via método POST ou GET. Durante a requisição de uma página, o trabalho é realizado no servidor, enquanto o cliente aguarda pela resposta.

No entanto, as requisições de páginas são interações curtas nas quais as variáveis existem somente durante aquela requisição. Uma forma de se construir uma interação de longa duração entre o cliente e o servidor é por meio do uso de seções. Uma seção é representada por meio de um array no qual podemos armazenar valores, e que mantém seu estado mesmo através de sucessivas requisições de páginas, estabelecendo um vínculo entre o cliente e o servidor. Geralmente utilizam-se seções para controle de login e permissões, de modo que podemos armazenar na seção quem está logado, para verificar página por página se este usuário tem permissões para tal. Este é apenas um exemplo simples.

Seções são comumente relacionadas aos tópicos stateless ou statefull Objects. Utilizamos o termo stateless quando não é necessário manter o estado dos nossos objetos durante a interação do usuário com o sistema, ou seja, quando consultamos um site de compras no qual somente visualizamos produtos, sendo que cada visualização consiste em uma requisição de página. Utilizamos o termo statefull quando precisamos manter o estado de objetos mesmo durante várias requisições de pági-

nas. Um exemplo é quando o usuário interage com os produtos, adicionando-os em uma cesta de compras. Neste caso, é necessário que o objeto que representará a cesta de compras mantenha o seu estado mesmo durante várias requisições de páginas. Para isso utilizamos seções, as quais garantem que o nosso objeto mantenha o seu estado.

7.1.4.1 Registry Pattern

Em diversas situações no desenvolvimento de aplicações de negócio precisamos compartilhar algumas informações por meio de várias etapas de um mesmo processo dentro do sistema. Para isso, precisamos de um local visível globalmente que sirva para o armazenamento compartilhado de informações. Poderíamos utilizar variáveis globais, mas esta técnica contraria uma série de princípios da orientação a objetos, tais como o encapsulamento. Então, para implementarmos isso, muitas vezes utilizamos um objeto, o qual atua como um registrador de informações, não tendo relações diretas com outros objetos de negócio como associações, agregações ou heranças. Tais objetos registradores existem apenas para armazenar e retornar determinados valores. Pense no registro do Windows ou em uma memória com determinadas posições que podem ser utilizadas para alocar valores.

Um objeto que implementa o Registry Pattern geralmente utiliza métodos estáticos para atribuir e retornar a informação que desejamos armazenar em seus registradores. Não faz sentido termos várias instâncias de objetos registry no sistema, uma vez que estaremos falando de um local compartilhado (comunitário) de armazenamento de informações. Em nossa aplicação, utilizaremos um objeto registry para lidar com seções. Antes, porém, estudaremos um pouco mais sobre seções.

O protocolo HTTP é por natureza stateless, ou seja, ele não mantém o seu estado no decorrer das várias interações de um usuário com o sistema, o que torna difícil identificarmos um usuário com unicidade ou mesmo passarmos valores de variáveis de uma página a outra durante a navegação do usuário no sistema. Podemos passar pela URL (método `GET`), assim como via submissão de formulários (método `POST`). No entanto, torna-se excessivamente trabalhoso lembrar sempre de passar uma variável adiante no sistema. Uma seção torna este trabalho muito mais simples.

Uma seção pode ser definida como o tempo decorrido durante uma interação do usuário com o sistema. A seção para o PHP representa um espaço físico localizado no servidor onde podemos armazenar variáveis que se mantêm persistentes mesmo durante sucessivos acessos ao site/sistema. Durante esta interação do usuário com o sistema podemos armazenar valores de variáveis neste espaço chamado seção. Cada visitante do site/sistema recebe um código de seção único, chamado de `SESSION_ID`, que geralmente é armazenado em um cookie.

Uma seção no PHP é representada por um array chamado `$_SESSION`. Este array é visível por toda a aplicação e os dados armazenados nele persistem durante toda interação do usuário, mesmo passando por diferentes páginas. O que faremos é criar uma interface orientada a objetos (um Registry Pattern) que trate de registrar e obter valores a partir deste array. Esta interface será representada pela classe `TSession` que em seu método construtor inicializará a seção. Tal inicialização deverá ser realizada no início de cada página que desejar fazer uso dos dados da seção. A classe `TSession` também terá o método `setValue()`, que registra um valor em uma determinada posição da seção (identificado pelo primeiro parâmetro). Já a função `getValue()` retorna o valor atribuído a uma determinada posição da seção por meio do método `setValue()`. Terminamos com o método `freeSession()` que destrói todos os dados contidos em uma seção.

A Figura 7.3 foi construída no sentido de demonstrar que a classe `TSession` poderá ser utilizada por diversas outras classes do sistema em momentos diferentes, sem que tenha com elas qualquer tipo de relacionamento como associação, agregação ou herança.

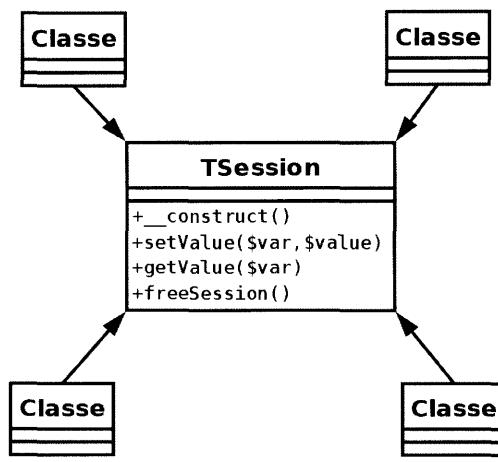


Figura 7.3 – Classe `TSession`.

TXT `TSession.class.php`

```
<?php
/**
 * classe TSession
 * gerencia uma seção com o usuário
 */
class TSession
{
```

```
/*
 * método construtor
 * inicializa uma seção
 */
function __construct()
{
    session_start();
}

/**
 * método setValue()
 * armazena uma variável na seção
 * @param $var = Nome da variável
 * @param $value = Valor
 */
function setValue($var, $value)
{
    $_SESSION[$var] = $value;
}

/**
 * método getValue()
 * retorna uma variável da seção
 * @param $var = Nome da variável
 */
function getValue($var)
{
    return $_SESSION[$var];
}

/**
 * método freeSession()
 * destrói os dados de uma seção
 */
function freeSession()
{
    $_SESSION = array();
    session_destroy();
}
?>
```

No exemplo a seguir estamos criando um pequeno controle para registrar a visita de um usuário no site. Como um usuário pode visitar repetidamente várias páginas de um site e até mesmo recarregar diversas vezes a página, não queremos registrar erroneamente a mesma visita diversas vezes no banco de dados. Para isso, utilizare-

mos uma seção, já que sua informação persiste durante um período de consequentes interações do usuário com o site. Para realizar este controle estamos criando dentro da seção a variável `counted`. Esta variável estará vazia sempre que for a primeira interação, durante a qual o programa irá entrar no primeiro IF e registrar a visita no banco de dados, ao mesmo tempo em que registra o valor TRUE nesta posição da seção. Em todas as visitas subsequentes, o programa cairá no `else`, exibindo na tela a mensagem “visita já registrada”.

session.php

```
<?php
include 'app.widgets/TSession.class.php';

new TSession;

if (!TSession::getValue('counted'))
{
    echo 'registrando visita';
    TSession::setValue('counted', true);
}
else
{
    echo 'visita já registrada';
}
?>
```

7.1.5 Front Controller

Em um sistema grande, existem tarefas que se repetem por todas as suas rotinas. Podemos citar algumas: verificações de segurança, log de operações, autenticação, internacionalização e padronização de interface com o usuário. Uma aplicação pode ser composta por um conjunto de programas totalmente independentes uns dos outros, como também pode ter um ponto central de acesso, um ponto em comum que coordena qual programa será executado ou qual página será exibida. Chamamos este ponto de entrada da aplicação de Front Controller.

O Front Controller é um tipo de script centralizador, também conhecido por “One script serves all” ou “Um script serve a todos”. Para implementar um script centralizador geralmente utiliza-se o próprio `index.php` para isto. Pode-se passar um parâmetro identificando qual programa queremos executar, e o próprio “`index`” toma as medidas necessárias para realizar a sua inclusão no código-fonte. Neste exemplo temos um script centralizador estático. Para executar o programa de clientes, seria necessário acessar “`index.php?program=clientes`”. Note que o fato de ser estático faz com que tenhamos de alterar o script centralizador sempre que criamos um novo programa.

 index.php

```
<?php
switch ($_GET['program'])
{
    case 'clientes':
        include 'clientes.php';
        break;
    case 'produtos':
        include 'produtos.php';
        break;
    case 'cidades':
        include 'cidades.php';
        break;
    case 'fabricantes':
        include 'fabricantes.php';
        break;
}
?>
```

Essa abordagem é interessante porque com ela podemos centralizar o layout da aplicação no arquivo `index.php`, deixando para os arquivos a serem incluídos a tarefa de implementar cada uma das funcionalidades desejadas. Tarefas como a inicialização de seções precisariam ser realizadas uma única vez no arquivo principal. Entretanto, ficamos, de certo modo, “engessados” em relação ao layout da aplicação, uma vez que pode ser necessário ter diferentes layouts para determinadas funcionalidades a serem acessadas.

Uma outra abordagem que podemos adotar é criar um script para cada página, também conhecido por “One script per page”, ou seja, criar o script `clientes.php`, outro `mercadorias.php`. Dessa forma, em vez de todos passarem pelo ponto centralizador `index.php` e serem incluídos no código, cada um teria autonomia de incluir classes e funções e de inicializar seções, tarefa que na abordagem anterior é mais simples. Uma vantagem é a de podermos ter diferentes layouts para diferentes tarefas do sistema. O programa de inclusão de clientes poderia ter um visual diferente do programa de inclusão de mercadorias, por exemplo, uma vez que internamente cada um poderá montar o layout separadamente. No exemplo a seguir, temos o arquivo `menu.html` contendo todas as opções do sistema, sendo que cada programa individual, como o `clientes.php`, inclui este menu e lida com aspectos como seções separadamente.

 menu.html

```
<a href="clientes.php">clientes</a> <br>
<a href="mercadorias.php">mercadorias</a> <br>
```

 clientes.php

```
<?php
session_start();
include 'menu.html';
/*
código ...
*/
?>
```

Um Front Controller geralmente é representado por um objeto que recebe todas as requisições de um site. A tarefa deste objeto é analisar alguns parâmetros, como a URL, e decidir qual comando executar, qual objeto instanciar. A seguir temos nossa primeira versão de um Front Controller. O arquivo `index.php` será nosso script centralizador, e a classe `TApplication` será o nosso Front Controller. Todas as requisições de página no sistema passarão por ela. Observe que fazemos a chamada do método estático `run()`. Este método analisa a presença de parâmetros na URL (variável `$_GET`), verificando a existência do parâmetro `class`. Caso essa classe exista no sistema, o que é verificado por meio da função `class_exists()`, instanciamos um objeto desta classe e a exibimos na tela pelo método `show()`.

Dessa forma, estes seriam alguns exemplos de chamadas ao arquivo `index.php`:

```
index.php?class=ClientesList
index.php?class=ProdutosForm
```

O nome da classe em questão (`ClientesList` ou `ProdutosForm`) será o objeto a ser exibido na tela. Note que, no caso de não passarmos o nome da classe que queremos instanciar via URL, o Front Controller não tem ação nenhuma e visualizamos somente uma página em branco. Como este é um Front Controller dinâmico que analisa a URL e, baseado nela, toma a decisão de qual objeto instanciar, não precisamos modificá-lo quando da inclusão de novos programas. Veja na Figura 7.4 o funcionamento de um Front Controller.

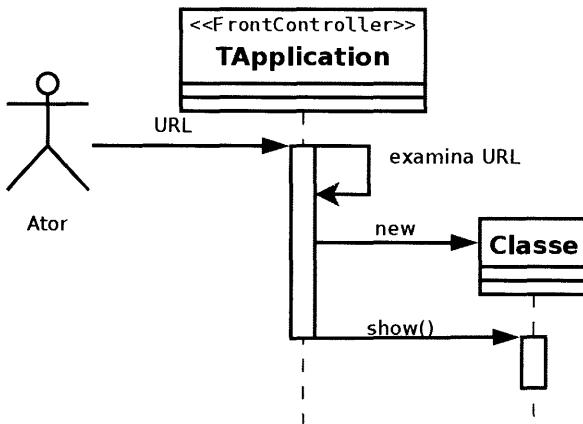


Figura 7.4 – Funcionamento de um FrontController.

index.php

```

<?php
class TApplication
{
    static public function run()
    {
        if ($_GET)
        {
            $class = $_GET['class'];
            if (class_exists($class))
            {
                $pagina = new $class;
                $pagina->show();
            }
        }
    }
}

TApplication::run();
?>
    
```

Como essas classes realmente não existem dentro do `index.php`, elas nunca seriam instanciadas, uma vez que o programa nunca passaria pela verificação `class_exists()`. Então temos de carregar tais classes no início do `index.php`. Como não sabemos se elas serão necessárias nem quando o serão, utilizaremos a função `__autoload()` do PHP, a qual carregará estas classes somente quando forem instanciadas pela primeira vez. Como nosso sistema utilizará diversas classes distribuídas ao longo das pastas do sistema, como `app.widgets` e `app.ado`, verificaremos pela presença de classes em qualquer uma dessas pastas.

index.php

```
<?php
function __autoload($classe)
{
    $pastas = array('app.widgets', 'app.ado', 'app.model', 'app.control');
    foreach ($pastas as $pasta)
    {
        if (file_exists("${$pasta}/{$classe}.class.php"))
        {
            include_once "${$pasta}/{$classe}.class.php";
        }
    }
}

...
...
?>
```

Você deve ter notado que, além das pastas em que temos armazenadas as classes que criamos até o momento, surgiram duas pastas novas: `app.control` e `app.model`. Na pasta `app.control`, armazenaremos as páginas da aplicação que iremos desenvolver (camada de controle), e na pasta `app.model` armazenaremos as classes contendo a definição dos Active Records (camada de Modelo). Veja na Figura 7.5 a nossa estrutura de diretórios.

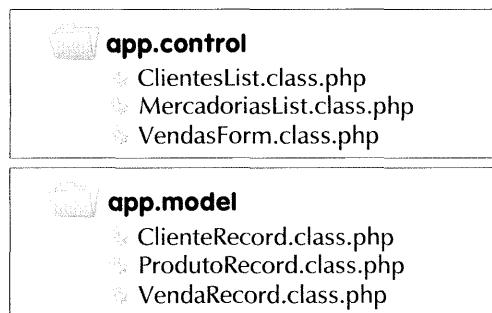


Figura 7.5 – Estrutura de diretórios.

7.1.6 Template View

Até o momento, desenvolvemos uma série de classes para renderização de componentes como formulários e listagens e também de controle do fluxo de execução, mas em nenhum momento nos preocupamos com a aparência da aplicação. Geralmente definimos a aparência de uma aplicação web em uma ferramenta WYSIWYG que permite arrastar, soltar, colorir e formatar textos, frames e tabelas. Os componentes

que desenvolvemos sofrem de uma rigidez estética grave, ou seja, a aparência é fixa e definida pelos mesmos. Quando construímos uma aplicação maior para fazer uso desses diversos formulários que criamos, torna-se necessário encaixar ou dispor tais objetos dentro de um visual mais elaborado. A melhor forma de se criar um layout para um site é deixar um designer especializado fazer para que posteriormente vinhemos a integrar este layout na aplicação.

Quando recebemos um layout para integrar nossa aplicação dentro do mesmo, obtemos um código HTML com imagens, camadas, tabelas, dentre outros, e precisamos fazer nossa aplicação se ajustar a esse layout. Uma forma de se atingir este objetivo é criar marcas dentro do arquivo de layout. Quando um programa é executado, ele lê o arquivo de layout e substitui essas marcas pelo seu resultado. A criação de marcas dentro do arquivo de layout não compromete a estrutura do mesmo, que continua podendo ser editado em qualquer editor WYSIWYG ao mesmo tempo em que fica livre de qualquer lógica de programação, visto que a tradução das marcas pelo respectivo conteúdo ficará a cargo do programa que utilizará o Template View.

O próprio PHP já possui diversas ferramentas para trabalhar com templates. O projeto Smarty talvez seja o mais conhecido de todos. Neste nosso sistema, criaremos um mecanismo de template realmente bastante simples, somente para suprir nossas necessidades que são mínimas. Para isso, criaremos um layout HTML com um banner ao topo, uma área de menu com as opções do sistema à esquerda e um marcador ao centro. Dentro do código HTML, este marcador será representado por #CONTENT#. Quando qualquer classe for requerida pelo sistema, esse conteúdo será substituído pelo output gerado pela página. Para capturarmos este output, é necessário utilizarmos as funções de buffer do php, como `ob_start()`, `ob_get_contents()` e `ob_end_clean()`. Dessa forma, nossa classe Front Controller (`TApplication`) irá capturar o resultado do método `show()` de cada página e utilizá-lo no lugar da marcação `#CONTENT#`, dentro do layout. Veja na Figura 7.6 o layout de nosso Template View.

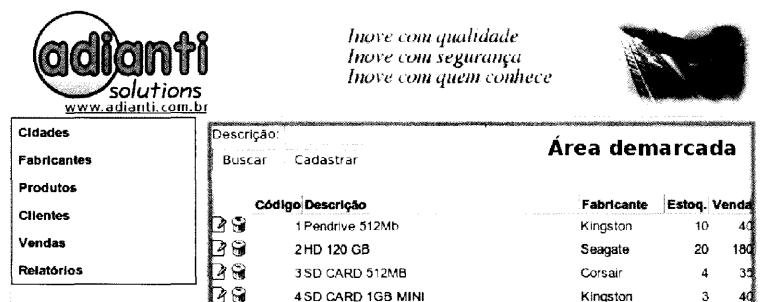


Figura 7.6 – Layout do TemplateView.

 index.php

```
<?php
function __autoload() {...}

class TApplication
{
    static public function run()
    {
        $template = file_get_contents('template.html');
        if ($_GET)
        {
            $class = $_GET['class'];
            if (class_exists($class))
            {
                $pagina = new $class;
                ob_start();
                $pagina->show();
                $content = ob_get_contents();
                ob_end_clean();
            }
            else if (function_exists($method))
            {
                call_user_func($method, $_GET);
            }
        }
        echo str_replace('#CONTENT#', $content, $template);
    }
}
TApplication::run();
?>
```

A seguir, você confere o arquivo de layout que iremos utilizar. Verifique a presença da marca #CONTENT# no meio do arquivo.

 template.html

```
<style>
estilos...
estilos...
</style>
<body>
<center>
<div id="pageheader">
    <img border=0 src='app.images/pageheader.png'>
</div>
```

```
<div id="pagebody">
    <div id="pageleft">
        <div id='inativo'
            OnMouseOver="this.id='ativo'" OnMouseOut="this.id='inativo'"
            OnClick="document.location='?class=CidadesList'">
            Cidades
        </div>
        <div id='inativo'
            OnMouseOver="this.id='ativo'" OnMouseOut="this.id='inativo'"
            OnClick="document.location='?class=FabricantesList'">
            Fabricantes
        </div>
        <div id='inativo'
            OnMouseOver="this.id='ativo'" OnMouseOut="this.id='inativo'"
            OnClick="document.location='?class=ProdutosList'">
            Produtos
        </div>
        <div id='inativo'
            OnMouseOver="this.id='ativo'" OnMouseOut="this.id='inativo'"
            OnClick="document.location='?class=ClientesList'">
            Clientes
        </div>
        <div id='inativo'
            OnMouseOver="this.id='ativo'" OnMouseOut="this.id='inativo'"
            OnClick="document.location='?class=VendasForm'">
            Vendas
        </div>
        <div id='inativo'
            OnMouseOver="this.id='ativo'" OnMouseOut="this.id='inativo'"
            OnClick="document.location='?class=RelatorioForm'">
            Relatórios
        </div>
    </div>

    <div id="pagecontent">
        #CONTENT#
    </div>

    <div id="pageboth"></div>
</div>
</center>
```

7.2 Uma aplicação-exemplo

Como falamos no início deste capítulo, chegou o momento de construirmos uma aplicação completa, com formulários de cadastros, listagens, processamentos e relatórios. Como tema da aplicação, escolhemos o comércio, pois praticamente todos entendem o vocabulário utilizado neste meio.

O objetivo de nossa aplicação é muito simples: controlar cadastros de clientes e produtos, permitir categorizar os clientes por sua cidade e os produtos pelo seu fabricante, permitir registrar as vendas ocorridas e também permitir a emissão de relatório em que irão constar as vendas realizadas em um determinado período.

Sempre que você ler classes de modelo (`CidadeRecord`, `ClienteRecord`, `FabricanteRecord`), elas serão armazenadas na pasta `app.model`, e sempre que você ler classes de controle (`CidadesList`, `ClientesList`, `VendasForm`, `RelatorioForm`), elas serão armazenadas na pasta `app.control`. Como toda aplicação passará pelo Front Controller `TApplication` (`index.php`), as classes estarão disponíveis para a aplicação sempre que forem requisitadas (via `__autoload`), bastando ao usuário entrar com sua localização (ou acessar o menu), como da forma a seguir:

```
index.php?class=CidadesList  
index.php?class=ClientesList
```

7.2.1 Estrutura

A idéia central do sistema não é muito complexa. Para estruturá-lo de acordo com as necessidades é necessário ter uma tabela para cadastrarmos os clientes com as suas respectivas informações, como nome, endereço e telefone. Como precisamos categorizar os clientes por cidade, é necessário criar uma tabela auxiliar, ligando os clientes com suas respectivas cidades.

A criação dessa tabela auxiliar nos permitirá inserir novas cidades a qualquer momento. Precisamos também controlar produtos; neste caso, cadastraremos informações como: descrição, quantidade em estoque, preço de custo e preço de venda. Precisamos também saber qual é o fabricante de cada produto. Como podem surgir novos fabricantes a qualquer momento, preferimos criar uma tabela auxiliar que contenha o nome e o site do fabricante para podermos categorizar os produtos por fabricante. Caso o fabricante fosse armazenado em um campo de livre digitação (`string`) na tabela de produtos, não teríamos uma uniformidade no cadastro. Em um momento, um usuário poderia digitar “samsung” e em outro momento “sansumg”, o que tornaria impossível emitirmos um relatório de produtos por fabricante com a informação exata. Tornando o fabricante uma informação ligada por uma tabela auxiliar, obrigamos o usuário a buscar um registro nesta tabela, e, caso o registro

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

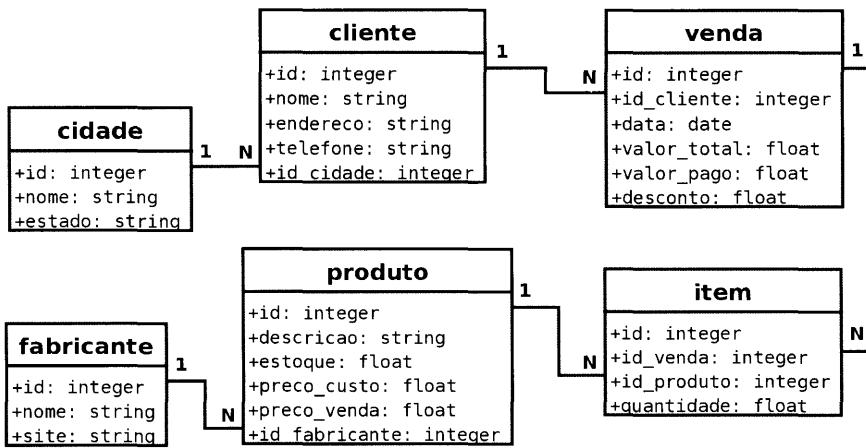


Figura 7.7 – Modelo Entidade Relacionamento.

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`.

CidadeRecord.class.php

```
<?php
/*
 * classe CidadeRecord
 * Active Record para tabela Cidade
 */
class CidadeRecord extends TRecord
{
}
?>
```

7.2.2.2 Aplicação

Como o cadastro de cidades é composto de poucos campos, poderemos fazer com que o formulário de cadastro e a listagem de cidades sejam feitos no mesmo programa. Para isso, teremos na parte superior da tela o formulário de cadastros e, na parte inferior, a listagem. Cada vez que o usuário clicar no botão de salvar, automaticamente a tela será recarregada e a listagem já exibirá a cidade cadastrada. Quando o usuário clicar no botão de editar cidade (na listagem), o registro sobre o qual ele clicou será carregado no formulário acima da listagem, permitindo que o mesmo altere as informações da cidade e a grave novamente no banco de dados. Quando o usuário clicar no botão de exclusão, será perguntado se o mesmo tem certeza de que deseja excluir o registro antes que este seja deletado da base de dados. Na Figura 7.8 você confere os métodos que serão executados para cada ação existente na página.

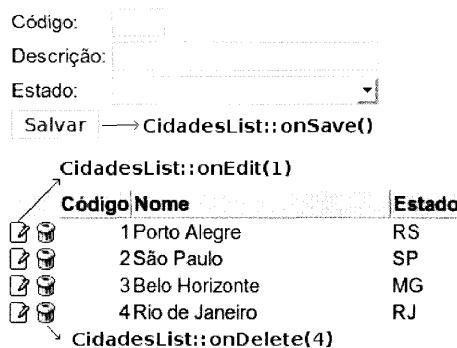


Figura 7.8 – Listagem de cidades.

A seguir temos a classe `CidadesList`, a qual será executada sempre que o usuário clicar sobre a opção **cidades** no menu de opções. Seu método construtor será responsável por instanciar os componentes que farão parte da tela de cadastro. Primeiro instanciamos um formulário com três campos: `id`, `nome` e `estado`, sendo que o estado é uma combo-box (`TCombo`) e os demais são campos de digitação (`TEntry`). Depois de

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 formulário 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);
    }
}
```

```
// define os tamanhos dos campos
$codigo->setSize(40);
$estado->setSize(200);

// adiciona uma linha para o campo código
$row=$table->addRow();
$row->addCell(new TLabel('Código:'));
$row->addCell($codigo);

// adiciona uma linha para o campo descrição
$row=$table->addRow();
$row->addCell(new TLabel('Descrição:'));
$row->addCell($descricao);

// adiciona uma linha para o campo estado
$row=$table->addRow();
$row->addCell(new TLabel('Estado:'));
$row->addCell($estado);

// cria um botão de ação (salvar)
$save_button=new TButton('save');
// define a ação do botão
$save_button->setAction(new TAction(array($this, 'onSave')), 'Salvar');

// adiciona uma linha para a ação do formulário
$row=$table->addRow();
$row->addCell($save_button);

// define quais são os campos do formulário
$this->form->setFields(array($codigo, $descricao, $estado, $save_button));
```

Depois de criarmos o formulário de cadastro da cidade, criamos a listagem de cidades, que será representada por um objeto `TDataGrid`. A listagem de cidades terá três colunas: id, nome e estado. Além disso, terá duas ações: editar e deletar. A ação editar, ao ser clicada, executará o método `onEdit()`, representado pela imagem `ico_edit.png`, e passará como parâmetro para a função executada o valor do campo `id`. A ação deletar, por sua vez, executará o método `onDelete()`, será representada pela imagem `ico_delete.png` e passará o valor do campo `id` para a função.

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

// instancia as colunas da DataGrid
$codigo  = new TGridColumn('id',      'Código', 'right', 50);
$nome    = new TGridColumn('nome',   'Nome',   'left', 200);
$estado  = new TGridColumn('estado', 'Estado', 'left', 40);
```

```

// adiciona as colunas à DataGrid
$this->datagrid->addColumn($codigo);
$this->datagrid->addColumn($nome);
$this->datagrid->addColumn($estado);

// instancia duas ações da DataGrid
$action1 = new TDataGridAction(array($this, 'onEdit'));
$action1->setLabel('Editar');
$action1->setImage('ico_edit.png');
$action1->setField('id');

$action2 = new TDataGridAction(array($this, 'onDelete'));
$action2->setLabel('Deletar');
$action2->setImage('ico_delete.png');
$action2->setField('id');

// adiciona as ações à DataGrid
$this->datagrid->addAction($action1);
$this->datagrid->addAction($action2);

// cria o modelo da DataGrid, montando sua estrutura
$this->datagrid->createModel();

```

Depois de criarmos o formulário de cadastro e a listagem, precisamos adicionar estes componentes à página por meio do método `add()` da classe `TPage`. Para isso, criamos um objeto tabela para organizar o visual da página. Nesta tabela, adicionaremos duas linhas: a primeira linha irá conter o formulário, e a segunda linha irá conter a listagem de cidades.

```

// monta a página através de uma tabela
$table = new TTable;
// cria uma linha para o formulário
$row = $table->addRow();
$row->addCell($this->form);
// cria uma linha para a datagrid
$row = $table->addRow();
$row->addCell($this->datagrid);
// adiciona a tabela à página
parent::add($table);
}

```

Precisamos criar um método para carregar os registros na listagem de cidades – esta é a função do método `onReload()`, o qual será executado sempre que a página for exibida na tela, por meio do método `show()`, mas também poderá ser executado a partir de qualquer outro método, como, por exemplo, o método `onSave()`.

O método `onReload()` abre uma conexão com a base de dados `pg_livro` e instancia um repositório para o registro de `Cidade`. Criamos um critério para definir a ordenação dos registros por seu id. A partir disso, utilizamos o método `load()` para carregar todos os objetos da tabela de `Cidade`, percorremos estes objetos por meio de um laço de repetições `foreach` e os adicionamos na listagem (`$datagrid`).

```
/*
 * método onReload()
 * carrega a DataGrid com os objetos do banco de dados
 */
function onReload()
{
    // inicia transação com o banco 'pg_livro'
    TTransaction::open('pg_livro');

    // instancia um repositório para Cidade
    $repository = new TRepository('Cidade');

    // cria um critério de seleção, ordenado pelo id
    $criteria = new TCriteria;
    $criteria->setProperty('order', 'id');
    // carrega os objetos de acordo com o criterio
    $cidades = $repository->load($criteria);
    $this->datagrid->clear();
    if ($cidades)
    {
        // percorre os objetos retornados
        foreach ($cidades as $cidade)
        {
            // adiciona o objeto na DataGrid
            $this->datagrid->addItem($cidade);
        }
    }
    // finaliza a transação
    TTransaction::close();
    $this->loaded = true;
}
```

O método `onSave()` será executado sempre que o usuário clicar no botão **Salvar** do formulário. Quando isto acontecer, a página será recarregada e este método será executado, abrindo uma transação com a base de dados e obtendo os dados do formulário pelo método `getData()`, que irá retorná-los na forma de uma instância da classe `CidadeRecord`. Só teremos o trabalho de executar o método `store()`, armazenando este registro no banco de dados. Ao final, executamos o método `onReload()`, recarregando a listagem de registros em tela.

```

/*
 * método onSave()
 * executada quando o usuário clicar no botão salvar do formulário
 */
function onSave()
{
    // inicia transação com o banco 'pg_livro'
    TTransaction::open('pg_livro');
    // obtém os dados no formulário em um objeto CidadeRecord
    $cidade = $this->form->getData('CidadeRecord');
    // armazena o objeto
    $cidade->store();

    // finaliza a transação
    TTransaction::close();
    // exibe mensagem de sucesso
    new TMessage('info', 'Dados armazenados com sucesso');
    // recarrega listagem
    $this->onReload();
}

```

O método `onDelete()` será executado sempre que o usuário clicar no botão **deletar** na listagem. Quando isto acontecer, esse método receberá automaticamente todos os parâmetros passados via URL, ou seja, o vetor `$_GET` é passado como parâmetro ao método `onDelete()`, o qual irá obter o valor da variável `$key`, que deve conter o `id` do registro a ser deletado. O método `onDelete()` abre um diálogo de pergunta ao usuário (`TQuestion`). Caso o usuário responda afirmativamente, será executado o método `Delete()`, o qual receberá novamente o parâmetro `$key` contendo a chave do registro a ser deletado.

```

/*
 * método onDelete()
 * executada quando o usuário clicar no botão excluir dadatagrid
 * pergunta ao usuário se deseja realmente excluir um registro
 */
function onDelete($param)
{
    // obtém o parâmetro $key
    $key=$param['key'];

    // define duas ações
    $action1 = new TAction(array($this, 'Delete'));
    $action2 = new TAction(array($this, 'teste'));

    // define os parâmetros de cada ação
    $action1->setParameter('key', $key);
    $action2->setParameter('key', $key);
}

```

```
// exibe um diálogo ao usuário
new TQuestion('Deseja realmente excluir o registro?', $action1, $action2);
}
```

Como vimos anteriormente, o método `onDelete()` pergunta ao usuário se o registro deve ser realmente excluído. Em caso afirmativo, o método `Delete()` é executado, abrindo uma transação com a base de dados `pg_livro` e carregando o registro da tabela de cidades identificado pela variável `$key`. Em seguida, executamos o método `delete()` para registrar a exclusão do mesmo na base de dados e o método `onReload()`, que irá recarregar a listagem na tela. Também exibimos uma mensagem de informação com a frase “Registro excluído com sucesso”.

```
/*
 * método Delete()
 * exclui um registro
 */
function Delete($param)
{
    // obtém o parâmetro $key
    $key=$param['key'];
    // inicia transação com o banco 'pg_livro'
    TTransaction::open('pg_livro');

    // instancia objeto CidadeRecord
    $cidade = new CidadeRecord($key);
    // deleta objeto do banco de dados
    $cidade->delete();

    // finaliza a transação
    TTransaction::close();

    // recarrega adatagrid
    $this->onReload();
    // exibe mensagem de sucesso
    new TMessage('info', "Registro excluído com sucesso");
}
```

Sempre que o usuário clicar no botão **editar** da listagem, o método `onEdit()` será executado. Esse método recebe todos os parâmetros da URL (vetor `$_GET`) e deve carregar os dados correspondentes àquele registro, no formulário de cadastro de cidades, localizado logo acima da listagem. Para isso, esse método abre uma transação com a base de dados, instancia um objeto do tipo `CidadeRecord`, já passando a chave do registro (`$key`), e logo em seguida executa o método `setData()` do formulário, passando o objeto lido a partir da base de dados. Quando a página for exibida em tela pelo método `show()`, o formulário já apresentará os dados do registro na tela.

```

/*
 * método onEdit()
 * executada quando o usuário clicar no botão editar dadatagrid
 */
function onEdit($param)
{
    // obtém o parâmetro $key
    $key=$param['key'];
    // inicia transação com o banco 'pg_livro'
    TTransaction::open('pg_livro');
    // instancia objeto CidadeRecord
    $cidade = new CidadeRecord($key);
    // lança os dados da cidade no formulário
    $this->form->setData($cidade);

    // finaliza a transação
    TTransaction::close();
    $this->onReload();
}

```

Por último temos o método `show()` que sempre é executado. Esse método é, na verdade, responsável por identificar na URL do sistema qual método deve ser executado e, logo em seguida, apresentar o programa na tela. Caso o método `onReload()` ainda não tenha sido executado, será executado aqui.

```

/*
 * método show()
 * exibe a página
 */
function show()
{
    // se a listagem ainda não foi carregada
    if (!$this->loaded)
    {
        $this->onReload();
    }
    parent::show();
}
?>

```

7.2.3 Cadastro de fabricantes

Assim como temos o cadastro de cidades, que é um cadastro auxiliar ao cadastro de clientes (ainda não construído), temos o cadastro de fabricantes. Este cadastro será utilizado para segmentarmos os produtos por fabricante. Para tanto, criamos a tabela

de fabricantes contendo o nome e o site do fabricante. Faremos o cadastro de fabricantes igual ao cadastro de cidades, ou seja, com um formulário na parte superior da tela e uma listagem na parte inferior. Como esses dois programas tendem a ser muito parecidos, listaremos somente algumas pequenas partes nas quais os dois diferem.

7.2.3.1 Modelo

O registro do fabricante no banco de dados será bastante simples. Não precisaremos criar nenhum método diferente dos já existentes na classe `TRecord`, como `store()`, `delete()` e `load()`. Lembre-se de que esta classe será armazenada na pasta `app.model`.

FabricanteRecord.class.php

```
<?php
/*
 * classe FabricanteRecord
 * Active Record para tabela Fabricante
 */
class FabricanteRecord extends TRecord
{
}
?>
```

7.2.3.2 Aplicação

Como vimos anteriormente, o cadastro de fabricantes tende a ser muito parecido com o cadastro de cidades. Uma das diferenças encontra-se na criação dos campos do formulário de cadastro, na primeira parte do método construtor, no qual instanciamos três campos de digitação (`TEntry`), um para o código, um para o nome do fabricante e outro para o site. Marcamos com reticências as partes que permanecem inalteradas em relação ao programa de cadastro de Cidades. Na Figura 7.9 você confere os métodos que serão executados para cada ação que o usuário poderá tomar na página.

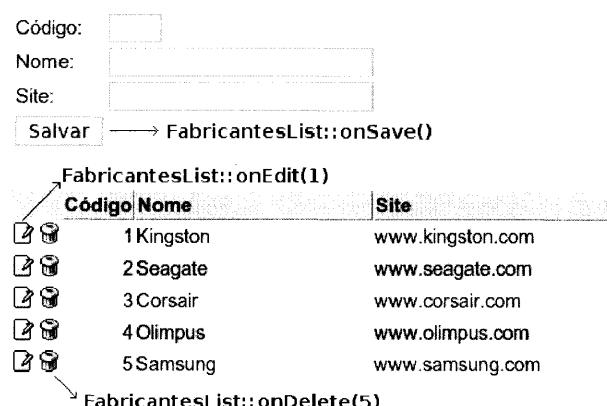


Figura 7.9 – Listagem de fabricantes.

O formulário de cadastro de fabricante também terá uma ação de salvar que executará o método `onSave()` quando for clicada. Esta parte permanece inalterada. A classe `FabricanteList` será armazenada na pasta `app.control`.

FabricanteList.class.php

```
<?php
/*
 * classe FabricantesList
 * cadastro de fabricantes
 * Contém o formulário e a listagem
 */
class FabricantesList 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_fabricantes');
        ...

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

        // adiciona uma linha para o campo código
        $row=$table->addRow();
        $row->addCell(new TLabel('Código:'));
        $row->addCell($codigo);

        // adiciona uma linha para o campo nome
        $row=$table->addRow();
        $row->addCell(new TLabel('Nome:'));
        $row->addCell($nome);

        // adiciona uma linha para o campo site
        $row=$table->addRow();
        $row->addCell(new TLabel('Site:'));
        $row->addCell($site);
```

A listagem deve refletir os campos do formulário de cadastro. Portanto, na listagem de fabricantes, listaremos os campos id, nome e site de cada fabricante. A listagem de fabricantes também terá duas ações: Editar e Excluir. Assim como no cadastro de cidades, a ação “Editar” executará o método `onEdit()`, e a ação “Excluir” executará o método `onDelete()`, sendo que ambos receberão os parâmetros da URL do sistema, que contém a chave (`$key`) do registro sobre o qual a ação foi aplicada. Suprimimos a criação das ações por ser idêntica ao cadastro de cidades.

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

// instancia as colunas da DataGrid
$codigo = new TDataGridColumn('id', 'Código', 'right', 50);
$nome = new TDataGridColumn('nome', 'Nome', 'left', 180);
$site = new TDataGridColumn('site', 'Site', 'left', 180);

// adiciona as colunas à DataGrid
$this->datagrid->addColumn($codigo);
$this->datagrid->addColumn($nome);
$this->datagrid->addColumn($site);
...
}
```

O método `onReload()`, responsável por carregar os objetos na listagem, é praticamente igual ao método `onReload()` do cadastro de cidades (`CidadesList`). A diferença básica ocorre ao instanciarmos o repositório de objetos. Antes trabalhávamos com “`Cidade`” e agora, com “`Fabricante`”. Lembre que este parâmetro representa o nome do objeto com que estamos trabalhando – `CidadeRecord` ou `FabricanteRecord`.

```
/*
 * função onReload()
 * carrega a DataGrid com os objetos do banco de dados
 */
function onReload()
{
    ...
    // instancia um repositório para Fabricante
    $repository = new TRepository('Fabricante');
    ...
}
```

O método `onSave()` também sofre pequenas modificações a partir do cadastro de cidades. No momento em que obtemos os dados do formulário pelo método `getData()`, em vez de passarmos o nome do objeto a ser instanciado como `CidadeRecord`, passamos `FabricanteRecord`; automaticamente um objeto deste tipo será instanciado.

```
/*
 * função onSave()
 * executada quando o usuário clicar no botão salvar do formulário
 */
function onSave()
{
    ...
    // obtém os dados no formulário em um objeto FabricanteRecord
    $fabricante = $this->form->getData('FabricanteRecord');
    // armazena o objeto
    $fabricante->store();
    ...
}
```

O método `onDelete()`, responsável por perguntar ao usuário se o mesmo tem certeza de que deseja excluir o registro, permanece inalterado.

```
/*
 * método onDelete()
 */
function onDelete($param)
{
    ...
}
```

O método `Delete()`, responsável efetivamente por excluir o registro da base de dados, é alterado em relação ao objeto que estamos excluindo. Antes estávamos instanciando um objeto `CidadeRecord`; agora estamos instanciando um objeto `FabricanteRecord`, para depois executar o seu método `delete()`.

```
/*
 * método Delete()
 * exclui um registro
 */
function Delete($param)
{
    ...
    // instancia objeto FabricanteRecord
    $fabricante = new FabricanteRecord($key);
    // deleta objeto do banco de dados
    $fabricante->delete();
    ...
}
```

Da mesma forma que no método `Delete()`, no método `onEdit()`, responsável por carregar um objeto do banco de dados e preencher o formulário com os atributos do mesmo, substituímos o objeto `CidadeRecord` por um `FabricanteRecord`.

```
/*
 * método onEdit()
 * executada quando o usuário clicar no botão visualizar
 */
function onEdit($param)
{
    ...
    // instancia objeto FabricanteRecord
    $fabricante = new FabricanteRecord($key);
    // lança os dados do fabricante no formulário
    $this->form->setData($fabricante);
    ...
}
```

O método `show()`, responsável por exibir a página na tela e executar o método `onReload()`, permanece igual ao cadastro de cidades.

```
/*
 * método show()
 * Executada quando o usuário clicar no botão excluir
 */
function show()
{
    ...
}
?>
```

7.2.4 Cadastro de clientes

Como os cadastros de cidade e de fabricantes são relativamente pequenos, pudemos concentrá-los em um só programa, o qual continha, na parte superior da tela, o formulário de cadastro e, na parte inferior, a listagem contendo os seus registros.

Para os cadastros de clientes e produtos vamos adotar uma abordagem um pouco diferente. Construiremos uma tela somente com a listagem dos registros, mas contendo um formulário de buscas na parte superior. Este formulário de buscas será utilizado para filtrar os dados da listagem. Quando o usuário digitar um termo no formulário de buscas e mandar “Buscar”, apenas os registros que satisfazem o termo que foi digitado serão exibidos. Quando o usuário clicar no botão de editar da listagem, será redirecionado para um outro programa que representa somente o formulário de edição de registros (`ClientesForm`). Em programas que manipulam uma quantidade maior de informações fica inviável colocarmos na mesma tela o formulário de cadastro juntamente com a listagem dos dados; é melhor separar em dois programas distintos.

7.2.4.1 Modelo

Todo cliente será manipulado pela classe `ClienteRecord`, uma extensão da classe `TRecord`. Com ela, utilizaremos os métodos `store()`, `load()`, `delete()`, além do método `get_nome_cidade()`, cujo objetivo é retornar o nome da cidade de um determinado cliente, baseado no código da cidade (`$id_cidade`), que é um atributo do cliente.

Como a classe `TRecord` verifica a existência de métodos iniciando com `get_`, que são considerados métodos *getters*, não será necessário executar o método `get_nome_cidade()` para descobrir o nome da cidade de um determinado cliente, bastando que se acesse sua propriedade `nome_cidade`. Quando isto é feito, automaticamente a classe `TRecord` verifica a existência de um método *getter* para esta propriedade. Veja na Figura 7.10 o relacionamento entre as classes `ClienteRecord` e `CidadeRecord`. A classe `ClienteRecord` será armazenada na pasta `app.model`.

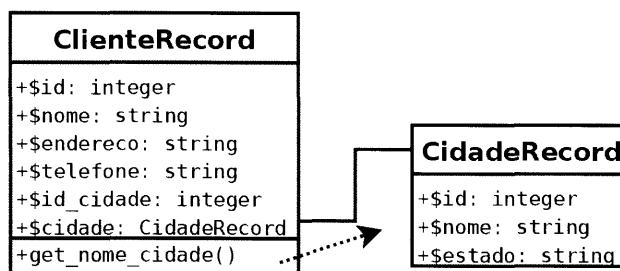


Figura 7.10 – Relacionamento entre o cliente e a cidade.

ClienteRecord.class.php

```

<?php
/*
 * classe ClienteRecord
 * Active Record para tabela Cliente
 */
class ClienteRecord extends TRecord
{
    private $cidade;
    /*
     * método get_nome_cidade()
     * executado sempre se for acessada a propriedade "nome_cidade"
     */
    function get_nome_cidade()
    {
        // instancia CidadeRecord, carrega
        // na memória a cidade de código $this->id_cidade
        if (empty($this->cidade))
            $this->cidade = new CidadeRecord($this->id_cidade);
    }
}

```

```

    // retorna o objeto instanciado
    return $this->cidade->nome;
}
?>

```

7.2.4.2 Aplicação

Nosso cadastro de clientes será desdobrado em dois programas: o primeiro para listagem de clientes (`ClientesList`), permitindo edição e exclusão, e o segundo será somente o formulário para cadastros e edições (`ClientesForm`). A listagem de clientes (`ClientesList`) terá, além da listagem, um formulário de buscas localizado na parte superior da listagem para que possamos filtrar por registros na base de dados e trazer para a listagem somente os registros que contêm o termo digitado. Veja na Figura 7.11 o formulário de buscas e a listagem.

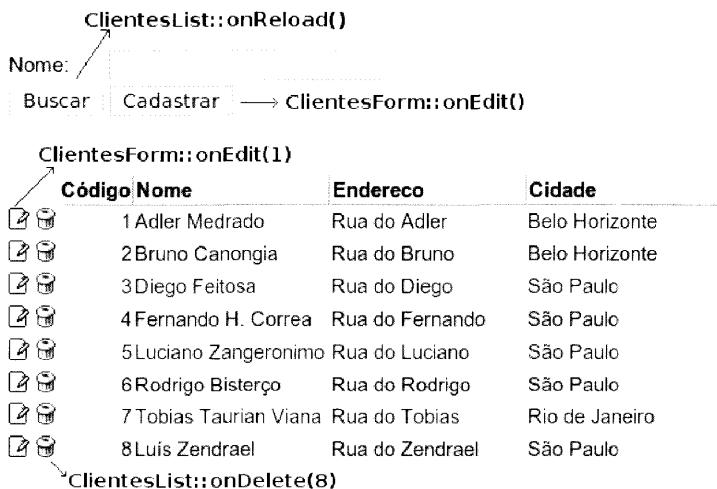


Figura 7.11 – Listagem de clientes.

Na Figura 7.11, temos um panorama da tela de listagem de clientes. Quando o usuário clicar no botão **excluir**, será executado o método `onDelete()` da própria classe `ClientesList`. O mesmo acontece quando ele clicar no botão **Buscar** do formulário, quando será executado o método `onReload()`. Entretanto, quando o usuário clicar no botão **Cadastrar** ou no botão **Editar**, será executado o método `onEdit()` da classe `ClientesForm`, a qual contém somente o formulário de cadastro de clientes e os métodos para ler (`onEdit`) e armazená-los (`onSave`) na base de dados.

No método construtor da classe `ClientesList`, construímos a interface, o formulário de buscas e a listagem de clientes. O formulário de buscas terá somente um único campo nome para que possamos filtrar clientes pelo nome. Este formulário de cadastro terá dois botões: o primeiro, chamado de **Buscar**, executará o método `onReload()` que

irá justamente ler o conteúdo digitado no formulário de buscas e filtrar os resultados da listagem. Já o segundo botão, chamado de **Cadastrar**, executará o método `onEdit()` da classe `ClientesForm`, que será criada a seguir. A classe `ClientesForm` será responsável somente pelo formulário de cadastro/alteração de registros. A classe `ClientesList` será armazenada na pasta `app.control`.

ClientesList.class.php

```
<?php
/*
 * classe ClientesList
 * listagem de Clientes
 */
class ClientesList extends TPage
{
    private $form;      // formulário de buscas
    private $datagrid; // listagem
    /*
     * método construtor
     * cria a página, o formulário de buscas e a listagem
     */
    public function __construct()
    {
        parent::__construct();

        // instancia um formulário
        $this->form = new TForm('form_busca_clientes');
        // instancia uma tabela
        $table = new TTable;

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

        // cria os campos do formulário
        $nome = new TEntry('nome');

        // adiciona uma linha para o campo nome
        $row=$table->addRow();
        $row->addCell(new TLabel('Nome:'));
        $row->addCell($nome);

        // cria dois botões de ação para o formulário
        $find_button = new TButton('busca');
        $new_button = new TButton('cadastra');
        // define as ações dos botões
        $find_button->setAction(new TAction(array($this, 'onReload')), 'Buscar');
```

```
$obj = new ClientesForm;  
$new_button->setAction(new TAction(array($obj, 'onEdit')), 'Cadastrar');  
  
// adiciona uma linha para as ações do formulário  
$row=$table->addRow();  
$row->addCell($find_button);  
$row->addCell($new_button);  
  
// define quais são os campos do formulário  
$this->form->setFields(array($nome, $find_button, $new_button));
```

A listagem de clientes terá quatro colunas: id, nome, endereço e nome da cidade. Observe que não existe a coluna `nome_cidade` no cadastro de clientes, mas existe o método `getter get_nome_cidade()` na classe `ClienteRecord`. Dessa forma, sempre que aces-sarmos a propriedade `nome_cidade`, este método será executado, consultando o nome da cidade na tabela de cidades baseado no campo `id_cidade` e retornando o nome correspondente.

A listagem de clientes possuirá duas ações. A primeira ação, “Editar”, representa-dada pelo ícone `ico_edit.png`, quando acionada, executará o método `onEdit()` da classe `ClientesForm`, passando o `id` do cliente como parâmetro. O método `onEdit()` da classe `ClientesForm` será visto mais adiante, mas podemos adiantar que ele carrega o objeto cliente baseado no seu `id` e preenche o formulário de clientes. A segunda ação, “De-letar”, quando acionada, executa o método `onDelete()` da classe `ClientesList`, passando o “`id`” do cliente e perguntando ao usuário se ele tem certeza de que deseja excluir o registro.

```
// instancia objeto DataGrid  
$this->datagrid = new TDataGrid;  
  
// instancia as colunas da DataGrid  
$codigo = new TGridColumn('id', 'Código', 'right', 50);  
$nome = new TGridColumn('nome', 'Nome', 'left', 140);  
$endereco = new TGridColumn('endereco', 'Endereço', 'left', 140);  
$cidade = new TGridColumn('nome_cidade', 'Cidade', 'left', 140);  
  
// adiciona as colunas à DataGrid  
$this->datagrid->addColumn($codigo);  
$this->datagrid->addColumn($nome);  
$this->datagrid->addColumn($endereco);  
$this->datagrid->addColumn($cidade);  
  
// instancia duas ações da DataGrid  
//$obj = new ClientesForm;  
$action1 = new TDataGridAction(array($obj, 'onEdit'));  
$action1->setLabel('Editar');
```

```

$action1->setImage('ico_edit.png');
$action1->setField('id');

$action2 = new TDataGridAction(array($this, 'onDelete'));
$action2->setLabel('Deletar');
$action2->setImage('ico_delete.png');
$action2->setField('id');

// adiciona as ações à DataGrid
$this->datagrid->addAction($action1);
$this->datagrid->addAction($action2);

// cria o modelo da DataGrid, montando sua estrutura
$this->datagrid->createModel();

// monta a página através de uma tabela
$table = new TTable;
$table->width='100%';
// cria uma linha para o formulário
$row = $table->addRow();
$row->addCell($this->form);
// cria uma linha para adatagrid
$row = $table->addRow();
$row->addCell($this->datagrid);
// adiciona a tabela à página
parent::add($table);
}

```

O método `onReload()` é executado sempre que a página é exibida ou antes disto por algum dos outros métodos presentes na página, como o `onDelete()`. Sua função é ler todos os registros da tabela de clientes, retornando-os na forma de objetos e adicionando-os na listagem de clientes.

Além disso, o método `onReload()` também é executado em resposta ao botão **Buscar** do formulário de buscas. Ele, então, verifica se o campo `nome` deste formulário contém informação. Em caso afirmativo, adicionamos um filtro no critério de seleção dos dados “`nome like...`” antes de carregar os objetos da tabela de clientes que satisfazem este critério por meio do método `load()`.

```

/*
 * método onReload()
 * carrega a DataGrid com os objetos do banco de dados
 */
function onReload()
{
    // inicia transação com o banco 'pg_livro'
    TTransaction::open('pg_livro');
}

```

```
// instancia um repositório para Cliente
$repository = new TRepository('Cliente');

// cria um critério de seleção de dados
$criteria = new TCriteria;
// ordena pelo campo id
$criteria->setProperty('order', 'id');

// obtém os dados do formulário de buscas
$dados = $this->form->getData();
// verifica se o usuário preencheu o formulário
if ($dados->nome)
{
    // filtra pelo nome do cliente
    $criteria->add(new TFilter('nome', 'like', "%{$dados->nome}%"));
}

// carrega os produtos que satisfazem o critério
$clientes = $repository->load($criteria);
$this->datagrid->clear();
if ($clientes)
{
    foreach ($clientes as $cliente)
    {
        // adiciona o objeto na DataGrid
        $this->datagrid->addItem($cliente);
    }
}
// finaliza a transação
TTransaction::close();
$this->loaded = true;
}
```

O método `onDelete()`, assim como vimos no programa de cadastro de cidades, pergunta ao usuário se ele tem certeza de que deseja excluir determinado registro. Em caso afirmativo, é executado o método `Delete()` passando como parâmetro a chave do registro (`$key`), que é lido a partir da própria URL.

```
/*
 * método onDelete()
 * executada quando o usuário clicar no botão excluir dadatagrid
 * pergunta ao usuário se deseja realmente excluir um registro
 */
function onDelete($param)
{
    // obtém o parâmetro $key
    $key=$param['key'];
```

```

// define duas ações
$action1 = new TAction(array($this, 'Delete'));
$action2 = new TAction(array($this, 'teste'));

// define os parâmetros de cada ação
$action1->setParameter('key', $key);
$action2->setParameter('key', $key);

// exibe um diálogo ao usuário
new TQuestion('Deseja realmente excluir o registro?', $action1, $action2);
}

```

Caso o usuário responda afirmativamente à pergunta sobre o desejo de excluir o registro, efetuada no método `onDelete()`, o método `Delete()` é executado. Nele, abrimos transação com o banco de dados `pg_livro`, instanciamos um objeto `ClienteRecord`, passando o “`id`” do cliente (variável `$key`) como parâmetro, e, logo em seguida, executamos o seu método `delete()`, excluindo o registro da base de dados.

```

/*
 * método Delete()
 * exclui um registro
 */
function Delete($param)
{
    // obtém o parâmetro $key
    $key=$param['key'];

    // inicia transação com o banco 'pg_livro'
    TTransaction::open('pg_livro');

    // instancia objeto ClienteRecord
    $cliente = new ClienteRecord($key);
    // deleta objeto do banco de dados
    $cliente->delete();

    // finaliza a transação
    TTransaction::close();

    // recarrega adatagrid
    $this->onReload();
    // exibe mensagem de sucesso
    new TMessage('info', "Registro excluído com sucesso");
}

```

O método `show()` simplesmente executa o método `show()` da classe-pai (`TPage`), mas antes executa o método `onReload()` caso ele ainda não tenha sido executado.

```
/*
 * método show()
 * executada quando o usuário clicar no botão excluir
 */
function show()
{
    // se a listagem ainda não foi carregada
    if (!$this->loaded)
    {
        $this->onReload();
    }
    parent::show();
}
?>
```

ClientesForm.class.php

A classe `ClientesForm` será executada a partir da `ClientesList`. Dentro da listagem de clientes, quando o usuário clicar no botão **Cadastrar** ou no botão **Editar** sobre algum registro, executaremos o método `onEdit()` da classe `ClientesForm`.

A classe `ClientesForm` basicamente representa o formulário de cadastro de um cliente. Ela apresenta os campos `id`, `nome`, `endereco`, `telefone` e `id_cidade` (código da cidade) para serem preenchidos, sendo que o campo código da cidade é alimentado por uma combo-box, que é automaticamente preenchida a partir de um repositório (`TRepository`) de cidades. Veja que percorremos uma coleção de objetos por um laço de repetições `foreach` para posteriormente adicionarmos os itens da combo-box por meio de seu método `addItem()`. O formulário de clientes terá um único botão de ação, chamado “Salvar”, que, ao ser clicado, executa o método `onSave()` da própria classe. Veja na Figura 7.12 o formulário de clientes.

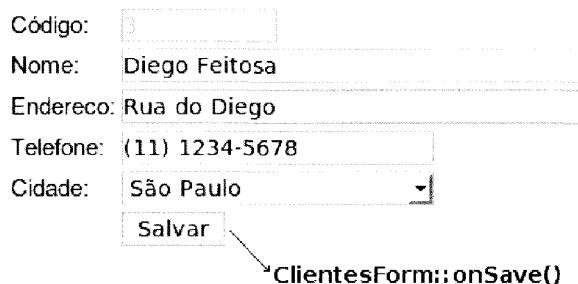


Figura 7.12 – Formulário de clientes.

```
<?php
/*
 * classe ClientesForm
 * formulário de cadastro de Clientes
 */
class ClientesForm extends TPage
{
    private $form; // formulário
    /*
     * método construtor
     * cria a página e o formulário de cadastro
     */
    function __construct()
    {
        parent::__construct();
        // instancia um formulário
        $this->form = new TForm('form_clientes');
        // instancia uma tabela
        $table = new TTable;

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

        // cria os campos do formulário
        $codigo = new TEntry('id');
        $nome = new TEntry('nome');
        $endereco = new TEntry('endereco');
        $telefone = new TEntry('telefone');
        $cidade = new TCombo('id_cidade');

        // define alguns atributos para os campos do formulário
        $codigo->setEditable(FALSE);
        $codigo->setSize(100);
        $nome->setSize(300);
        $endereco->setSize(300);

        // carrega as cidades do banco de dados
        TTransaction::open('pg_livro');
        // instancia um repositório de Cidade
        $repository = new TRepository('Cidade');
        // carrega todos os objetos
        $collection = $repository->load(new TCriteria);
        // adiciona objetos na combo
        foreach ($collection as $object)
        {
            $items[$object->id] = $object->nome;
        }
    }
}
```

```
$cidade->addItems($items);
TTransaction::close();

// adiciona uma linha para o campo código
$row=$table->addRow();
$row->addCell(new TLabel('Código:'));
$row->addCell($codigo);

// adiciona uma linha para o campo nome
$row=$table->addRow();
$row->addCell(new TLabel('Nome:'));
$row->addCell($nome);

// adiciona uma linha para o campo endereço
$row=$table->addRow();
$row->addCell(new TLabel('Endereço:'));
$row->addCell($endereco);

// adiciona uma linha para o campo telefone
$row=$table->addRow();
$row->addCell(new TLabel('Telefone:'));
$row->addCell($telefone);

// adiciona uma linha para o campo cidade
$row=$table->addRow();
$row->addCell(new TLabel('Cidade:'));
$row->addCell($cidade);

// cria um botão de ação para o formulário
$button1=new TButton('action1');
// define a ação do botão
$button1->setAction(new TAction(array($this, 'onSave')), 'Salvar');

// adiciona uma linha para a ação do formulário
$row=$table->addRow();
$row->addCell('');
$row->addCell($button1);

// define quais são os campos do formulário
$this->form->setFields(array($codigo, $nome, $endereco, $telefone, $cidade, $button1));

// adiciona o formulário na página
parent::add($this->form);
}
```

O método `onEdit()` é executado somente a partir da listagem de clientes (`ClientesList`). Sua função é ler um registro identificado pelo seu código (`$key`), carregar este objeto na memória e preencher o formulário com essas informações. Neste exemplo estamos instanciando um objeto `ClienteRecord` e utilizando-o para preencher o formulário pelo

método `setData()` do formulário. Note que neste método estamos utilizando controle de exceções `try/catch`. Na verdade, deveríamos utilizar o controle de exceções em todos os exemplos que envolvam operações que possam lançar algum tipo de erro, principalmente em interações com o banco de dados. Em alguns momentos, porém, suprimimos esse tipo de controle para que os exemplos não se estendam muito.

```
/*
 * método onEdit
 * edita os dados de um registro
 */
function onEdit($param)
{
    try
    {
        // inicia transação com o banco 'pg_livro'
        TTransaction::open('pg_livro');

        // obtém o Cliente de acordo com o parâmetro
        $cliente= new ClienteRecord($param['key']);
        // lança os dados do cliente no formulário
        $this->form->setData($cliente);

        // finaliza a transação
        TTransaction::close();
    }
    catch (Exception $e)      // em caso de exceção
    {
        // exibe a mensagem gerada pela exceção
        new TMessage('error', '<b>Erro</b>' . $e->getMessage());
        // desfaz todas alterações no banco de dados
        TTransaction::rollback();
    }
}
```

O método `onSave()` é executado sempre que o usuário clica no botão **Salvar**. O formulário, então, é submetido, e a classe `TPage` o executa. Nesse momento, obtemos os dados do formulário pelo método `getData()`, que já instancia um objeto `ClienteRecord` em memória, para logo em seguida executar seu método `store()`, que irá persisti-lo definitivamente na base de dados. Novamente estamos utilizando controle de exceções neste exemplo.

```
/*
 * método onSave
 * executado quando o usuário clicar no botão salvar
 */
```

```
function onSave()
{
    try
    {
        // inicia transação com o banco 'pg_livro'
        TTransaction::open('pg_livro');

        // lê os dados do formulário e instancia um objeto ClienteRecord
        $cliente = $this->form->getData('ClienteRecord');
        // armazena o objeto no banco de dados
        $cliente->store();

        // finaliza a transação
        TTransaction::close();
        // exibe mensagem de sucesso
        new TMessage('info', 'Dados armazenados com sucesso');
    }
    catch (Exception $e)      // em caso de exceção
    {
        // exibe a mensagem gerada pela exceção
        new TMessage('error', '<b>Erro</b>' . $e->getMessage());
        // desfaz todas alterações no banco de dados
        TTransaction::rollback();
    }
}
?>
```

7.2.5 Cadastro de produtos

Assim como fizemos com o cadastro de clientes, dividindo em dois programas diferentes, um para listagem de registros e outro para o formulário de cadastro, faremos com o cadastro de produtos. Este terá informações como a descrição do produto, quantidade em estoque, preço de custo, preço de venda e código do fabricante, além, é claro, do próprio “id”.

7.2.5.1 Modelo

Para manipular os registros na tabela de produtos, utilizaremos a classe `ProdutoRecord`. Além dos métodos básicos fornecidos pela classe-pai, como `store()`, `load()` e `delete()`, iremos também implementar um método *getter* chamado `get_nome_fabricante()`. Dessa forma, sempre que tentarmos acessar a propriedade `nome_fabricante`, esse método será executado. O método `get_nome_fabricante()` instancia um registro de `FabricanteRecord`, passando o código do fabricante do produto (`$id_fabricante`). Depois de obter o objeto `$fabricante`, retornamos a sua propriedade `nome`, que será posteriormente utilizada na

listagem de produtos. Veja na Figura 7.13 o relacionamento entre as classes ProdutoRecord e FabricanteRecord. A classe ProdutoRecord será armazenada na pasta app.model.

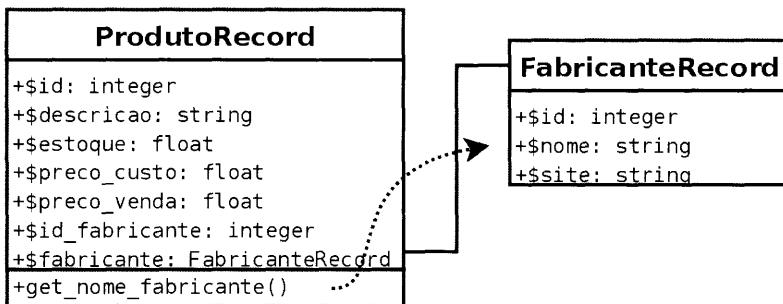


Figura 7.13 – Relacionamento entre produto e fabricante.

ProdutoRecord.class.php

```

<?php
/*
 * classe ProdutoRecord
 * Active Record para tabela Produto
 */
class ProdutoRecord extends TRecord
{
    private $fabricante;

    /*
     * método get_nome_fabricante()
     * retorna o nome do fabricante do produto
     */
    function get_nome_fabricante()
    {
        // instancia FabricanteRecord, carrega
        // na memória a fabricante de código $this->id_fabricante
        if (empty($fabricante))
            $this->fabricante = new FabricanteRecord($this->id_fabricante);
        // retorna o nome do fabricante
        return $this->fabricante->nome;
    }
}
?>

```

7.2.5.2 Aplicação

A listagem de produtos será implementada pela classe ProdutosList, que se assemelha em muito à listagem de clientes, com a diferença das informações nela manipuladas.

O método construtor da classe `ProdutosList` construirá a interface, criando um formulário de buscas de produtos e logo abaixo a listagem de produtos. No formulário, teremos somente um campo descrição. Assim como no cadastro de clientes, este formulário terá dois botões de ação: **Buscar** e **Cadastrar**, sendo que o botão **Buscar** recarrega a página executando o método `onReload()`, o qual, por sua vez, irá carregar na listagem os registros que forem compatíveis com o termo digitado pelo usuário. Já o botão **Cadastrar** irá recarregar a página, executando o método `onEdit()` da classe `ProdutosForm`. Esta contém somente o formulário de cadastro de produtos, que permite tanto o cadastramento de novos produtos quanto a edição de registros já existentes. Veja na Figura 7.14 a listagem de produtos.

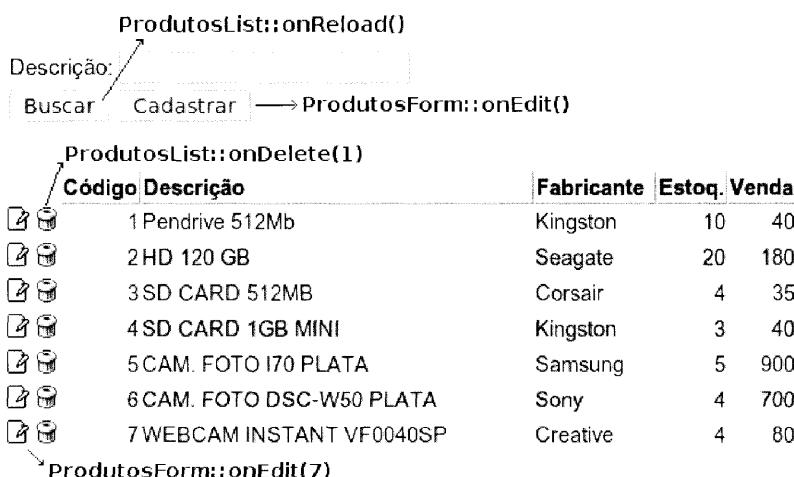


Figura 7.14 – Listagem de produtos.

■ ProdutosList.class.php

```
<?php
/*
 * classe ProdutosList
 * listagem de produtos
 */
class ProdutosList extends TPage
{
    private $form;      // formulário de buscas
    private $datagrid; // listagem

    /*
     * método construtor
     * Cria a página, o formulário de buscas e a listagem
     */
}
```

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

    // instancia um formulário
    $this->form = new TForm('form_busca_produtos');
    // instancia uma tabela
    $table = new TTable;

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

    // cria os campos do formulário
    $descricao= new TEntry('descricao');

    // adiciona uma linha para o campo descrição
    $row=$table->addRow();
    $row->addCell(new TLabel('Descrição:'));
    $row->addCell($descricao);

    // cria dois botões de ação para o formulário
    $find_button = new TButton('busca');
    $new_button = new TButton('cadastrar');
    // define as ações dos botões
    $find_button->setAction(new TAction(array($this, 'onReload')), 'Buscar');

    $obj = new ProdutosForm;
    $new_button->setAction(new TAction(array($obj, 'onEdit')), 'Cadastrar');

    // adiciona uma linha para as ações do formulário
    $row=$table->addRow();
    $row->addCell($find_button);
    $row->addCell($new_button);

    // define quais são os campos do formulário
    $this->form->setFields(array($descricao, $find_button, $new_button));
```

Logo abaixo do formulário de busca de produtos, temos uma listagem (objeto `TDataGrid`) que contém colunas para exibir os campos id, descrição, nome do fabricante (que é descoberto via método `getter`), estoque e preço de venda. Essa listagem, bem como no cadastro de clientes, terá duas ações: `onEdit()` e `onDelete()`. Ambos os métodos receberão como parâmetro o campo `id` do produto. Novamente suprimimos as partes do programa que coincidem com o cadastro de clientes. No final do método construtor, construímos uma tabela para adicionar à página o formulário de pesquisa de produtos e a listagem em si.

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

// instancia as colunas da DataGrid
$codigo = new TDataGridColumn('id', 'Código', 'right', 50);
$descricao= new TDataGridColumn('descricao', 'Descrição', 'left', 270);
$fabrica = new TDataGridColumn('nome_fabricante','Fabricante','left', 80);
$estoque = new TDataGridColumn('estoque', 'Estoq.', 'right', 40);
$preco = new TDataGridColumn('preco_venda', 'Venda', 'right', 40);

// adiciona as colunas à DataGrid
$this->datagrid->addColumn($codigo);
$this->datagrid->addColumn($descricao);
$this->datagrid->addColumn($fabrica);
$this->datagrid->addColumn($estoque);
$this->datagrid->addColumn($preco);

// instancia duas ações da DataGrid
$obj = new ProdutosForm;
$action1 = new TDataGridAction(array($obj, 'onEdit'));
$action1->setLabel('Editar');
$action1->setImage('ico_edit.png');
$action1->setField('id');

$action2 = new TDataGridAction(array($this, 'onDelete'));
$action2->setLabel('Deletar');
$action2->setImage('ico_delete.png');
$action2->setField('id');

// adiciona as ações à DataGrid
$this->datagrid->addAction($action1);
$this->datagrid->addAction($action2);

// cria o modelo da DataGrid, montando sua estrutura
$this->datagrid->createModel();
...
}
```

O método `onReload()` será executado quando a página for exibida na tela pelo método `show()`, e sua função é ler todos os produtos do banco de dados, ordenados pelo `id`, e adicioná-los na listagem de produtos (objeto `$this->datagrid`). Como o método `onReload()` também é executado em resposta ao botão **Buscar** do formulário, ele faz uma verificação (if `$dados->descricao`) para descobrir se o usuário digitou a descrição de algum produto para efetuar a busca. Note que a variável `$dados` só terá valores quando esse formulário for enviado (clique no botão **Buscar**). Quando há dados no formulário de buscas, adicionamos um filtro (`descricao like...`) ao critério de seleção de dados.

```
/*
 * método onReload()
 * carrega a DataGrid com os objetos do banco de dados
 */
function onReload()
{
    // inicia transação com o banco 'pg_livro'
    TTransaction::open('pg_livro');

    // instancia um repositório para Produto
    $repository = new TRepository('Produto');

    // cria um critério de seleção de dados
    $criteria = new TCriteria;
    // ordena pelo campo id
    $criteria->setProperty('order', 'id');

    // obtém os dados do formulário de buscas
    $dados = $this->form->getData();
    // verifica se o usuário preencheu o formulário
    if ($dados->descricao)
    {
        // filtra pela descrição do produto
        $criteria->add(new TFilter('descricao', 'like', "%{$dados->descricao}%"));
    }

    // carrega os produtos que satisfazem o critério
    $produtos = $repository->load($criteria);
    $this->datagrid->clear();
    if ($produtos)
    {
        foreach ($produtos as $produto)
        {
            // adiciona o objeto na DataGrid
            $this->datagrid->addItem($produto);
        }
    }
    // finaliza a transação
    TTransaction::close();
    $this->loaded = true;
}
```

O método `onDelete()` é exatamente igual ao do cadastro de clientes, ou seja, ele exibirá um diálogo ao usuário, perguntando se este tem certeza de que deseja excluir o registro. Em caso de resposta afirmativa, o método `Delete()` será executado.

```
/*
 * método onDelete()
 * executada quando o usuário clicar no botão excluir dadatagrid
 * pergunta ao usuário se deseja realmente excluir um registro
 */
function onDelete($param)
{
    ...
}
```

O método `Delete()` é responsável por excluir efetivamente o registro do banco de dados. Ele será praticamente igual ao método `Delete()` do cadastro de clientes, com exceção da classe que instanciamos: em vez de `ClienteRecord` será `ProdutoRecord`. Em seguida, executamos seu método `delete()`, excluindo efetivamente o registro do banco de dados.

```
/*
 * método Delete()
 * exclui um registro
 */
function Delete($param)
{
    ...
    // instancia objeto ProdutoRecord
    $produto = new ProdutoRecord($key);
    // deleta objeto do banco de dados
    $produto->delete();
    ...
}
```

O método `show()` também permanece inalterado em relação aos exemplos anteriores.

```
/*
 * método show()
 * exibe a página
 */
function show()
{
    ...
}
?>
```

Assim como construímos o formulário de clientes, construiremos o formulário de produtos. Este formulário (`ProdutosForm`) será executado exclusivamente a partir da listagem de produtos pelo botão **Cadastrar** ou pelo clique no botão **Editar** de algum item.

O formulário de cadastro de produtos terá campos de entrada de dados (`TEntry`) para os campos id, descrição, estoque e preço de venda. Terá ainda uma combo-box para o campo código do fabricante (`id_fabricante`), sendo que esta é alimentada a partir do banco de dados. Para isso, abrimos uma transação com o banco de dados por meio do método `open()` da classe `TTransaction`, instanciamos um repositório (`TRepository`) para objetos do tipo `FabricanteRecord`, carregamos todos os objetos dessa tabela por meio do método `load()` da classe `TRepository`, para posteriormente adicioná-los à combo-box `$fabricante`, pelo seu método `addItems()`. Veja na Figura 7.15 o formulário de produtos.

Código:

Descrição:	Pendrive 512Mb
Estoque:	10
Preço Custo:	20
Preço Venda:	40
Fabricante:	Kingston
Salvar —→ ProdutosForm::onSave()	

Figura 7.15 – Formulário de produtos.

Como o cadastro de produtos tende a ser em vários aspectos repetitivo em relação ao cadastro de clientes, cortaremos algumas partes do código, substituindo-as por reticências (...). No método construtor, cortaremos a parte em que definimos o tamanho de alguns campos e também a adição dos campos à tabela, em que deixaremos somente o primeiro (código); para os demais, o processo é o mesmo. O formulário de cadastro de produtos, assim como o de clientes, terá um botão de ação **Salvar** que irá disparar o método `onSave()`.

ProdutosForm.class.php

```
<?php
/*
 * classe ProdutosForm
 * formulário de cadastro de produtos
 */
class ProdutosForm extends TPage
{
    private $form; // formulário

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

```
function __construct()
{
    parent::__construct();
    // instancia um formulário
    $this->form = new TForm('form_produtos');
    // 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('descricao');
    $estoque     = new TEntry('estoque');
    $preco_custo = new TEntry('preco_custo');
    $preco_venda = new TEntry('preco_venda');
    $fabricante  = new TCombo('id_fabricante');

    // carrega os fabricantes do banco de dados
    TTransaction::open('pg_livro');
    // instancia um repositório de Fabricante
    $repository = new TRepository('Fabricante');
    // carrega todos objetos
    $collection = $repository->load(new TCriteria);
    // adiciona objetos na combo
    foreach ($collection as $object)
    {
        $items[$object->id] = $object->nome;
    }
    $fabricante->addItems($items);
    TTransaction::close();

    ...

    // adiciona uma linha para o campo código
    $row=$table->addRow();
    $row->addCell(new TLabel('Código:'));
    $row->addCell($codigo);

    ...

    // cria um botão de ação para o formulário
    $button1=new TButton('action1');
    // define a ação dos botão
    $button1->setAction(new TAction(array($this, 'onSave')), 'Salvar');
```

```
...  
  
// define quais são os campos do formulário  
$this->form->setFields(array($codigo, $descricao, $estoque, $preco_custo,  
    $preco_venda, $fabricante, $button1));  
  
// adiciona o formulário na página  
parent::add($this->form);  
}
```

O método `onEdit()` será executado sempre que o usuário clicar sobre algum item na listagem de clientes. Lembre-se que, nestes casos, o sistema receberá via URL os seguintes parâmetros:

```
?class=ProdutosForm&method=onEdit&key=5
```

A classe `TPage`, então, irá verificar qual classe deverá instanciar, qual método deverá executar e, para este método, ela passará todos os parâmetros da URL.

O método `onEdit()` abre uma transação com o banco de dados, instancia um objeto `ProdutoRecord`, carregando o registro do banco de dados identificado por `$key` e preenchendo o formulário de cadastro com esses dados. Caso ocorra alguma exceção, abortamos a transação e exibimos uma mensagem de erro.

```
/*  
 * método onEdit  
 * edita os dados de um registro  
 */  
function onEdit($param)  
{  
    try  
    {  
        // inicia transação com o banco 'pg_livro'  
        TTransaction::open('pg_livro');  
  
        // obtém o Produto de acordo com o parâmetro  
        $produto = new ProdutoRecord($param['key']);  
        // lança os dados do produto no formulário  
        $this->form->setData($produto);  
  
        // finaliza a transação  
        TTransaction::close();  
    }  
    catch (Exception $e)      // em caso de exceção  
    {  
        // exibe a mensagem gerada pela exceção  
        new TMessage('error', '<b>Erro</b>' . $e->getMessage());  
    }  
}
```

```
// desfaz todas alterações no banco de dados  
TTransaction::rollback();  
}  
}
```

O método `onSave()` será executado sempre que clicarmos no botão **Salvar** do formulário, o qual fará com que os dados do formulário sejam postados para o mesmo programa, que identificará o método a ser executado na URL como o método `onSave()`:

```
?class=ProdutosForm&method=onSave
```

O método `onSave()` irá abrir uma conexão com o banco de dados e obter os dados do formulário a partir do método `getData()`, que resultará em um objeto do tipo `ProdutoRecord`, o qual será persistido no banco de dados pelo método `store()`. Caso ocorra alguma exceção, abortamos a transação e exibimos uma mensagem de erro.

```
/*  
 * método onSave  
 * executado quando o usuário clicar no botão salvar  
 */  
function onSave()  
{  
    try  
    {  
        // inicia transação com o banco 'pg_livro'  
        TTransaction::open('pg_livro');  
  
        // lê os dados do formulário e instancia um objeto ProdutoRecord  
        $produto = $this->form->getData('ProdutoRecord');  
        // armazena o objeto no banco de dados  
        $produto->store();  
  
        // finaliza a transação  
        TTransaction::close();  
        // exibe mensagem de sucesso  
        new TMessage('info', 'Dados armazenados com sucesso');  
    }  
    catch (Exception $e)      // em caso de exceção  
    {  
        // exibe a mensagem gerada pela exceção  
        new TMessage('error', '<b>Erro</b>' . $e->getMessage());  
        // desfaz todas as alterações no banco de dados  
        TTransaction::rollback();  
    }  
}  
?>
```

7.2.6 Processo de venda

Como todo sistema comercial, o nosso também terá uma tela de vendas, mas esta será muito simples. Não teremos nada de formas de pagamento, opção para cartão de débito ou crédito, cheque, nada disto. Nosso processo de vendas será implementado por uma tela composta por um formulário e uma listagem. O usuário vai adicionando itens à listagem, como se ela fosse nossa cesta de compras. Para adicionar itens na cesta de compras (venda), basta digitarmos o código do produto e sua respectiva quantidade. Quando terminarmos de adicionar produtos, o usuário finaliza a venda. Na finalização da venda perguntamos qual o código do cliente para o qual a venda foi realizada, se foi dado algum desconto e qual o valor efetivamente pago.

Enquanto o usuário estiver adicionando produtos na cesta (venda), ele poderá excluir um item ou mesmo cancelar todo o processo. Isto significa que a adição de produtos à cesta de compras não deve implicar em gravação de nada no banco de dados, somente quando finalizamos a venda. Mas onde armazenaremos os dados dos produtos já adicionados à cesta? A resposta é: na seção. A seção é um espaço onde podemos armazenar variáveis que persistem mesmo após sucessivas trocas de páginas ou recarregamentos da própria página. Além disso, a seção existe somente para o contexto daquele usuário que está interagindo com o sistema. Outro usuário que estiver adicionando produtos à cesta terá uma seção totalmente diferente. Então a seção é o meio de armazenamento ideal, apesar de volátil, para ser utilizado durante esse processo enquanto não finalizamos a venda.

7.2.6.1 Modelo

Todo o processo de venda será persistido no banco de dados por duas classes: `VendaRecord` e `ItemRecord`. A venda representa o todo, e os itens são as partes deste todo. A venda armazena o cliente, a data e o valor da venda, ao passo que os itens armazenam cada um dos produtos que fizerem parte daquela venda e seus respectivos valores. Como aqui temos uma agregação, a classe `VendaRecord` não deverá apenas ser responsável por armazenar a venda em si no banco de dados, mas também cada um dos itens que fazem parte dela. Veja na Figura 7.16 o relacionamento entre as classes `VendaRecord` e `ItemRecord`.

Para agregar objetos do tipo `ItemRecord` à venda, criamos o método `addItem()`, o qual aceita somente objetos `ItemRecord` como parâmetro e os acrescenta ao array privado `$itens`. Observe que o método `addItem()` não armazena os itens no banco de dados, apenas os agrupa ao objeto `VendaRecord` que mantém referência a eles em memória.

Já o método `store()`, que normalmente armazena somente um registro no banco de dados (o próprio objeto em questão), teve de ser alterado para, além de armazenar a própria venda por meio do método `parent::store()`, armazenar cada um dos itens

da venda (objetos `ItemRecord` agregados). Para isso, percorremos esses objetos por um `foreach` e para cada um executamos o seu método `store()` correspondente. Antes de armazenar estes objetos filhos, entretanto, temos de atribuir a eles o atributo `id_venda`, que é a chave de referência (chave estrangeira) que fará a relação entre registro-pai e registro-filho na base de dados.

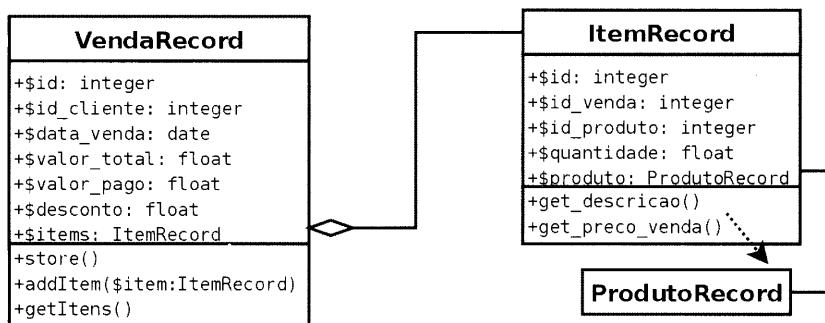


Figura 7.16 – Relacionamento entre a venda e o item.

VendaRecord.class.php

```

<?php
/*
 * classe VendaRecord
 * Active Record para tabela Venda
 */
class VendaRecord extends TRecord
{
    private $itens; // array de objetos do tipo ItemRecord

    /*
     * função addItem()
     * adiciona um item (produto) à venda
     */
    public function addItem(ItemRecord $item)
    {
        $this->itens[] = $item;
    }

    /*
     * função store()
     * armazena uma venda e seus itens no banco de dados
     */
    public function store()
    {
        // armazena a venda
        parent::store();
    }
}
  
```

```

// percorre os itens da venda
foreach ($this->itens as $item)
{
    $item->id_venda = $this->id;
    // armazena o item
    $item->store();
}
}
?>

```

Um item representa um produto dentro de uma venda, possui informações como o `id_produto`, quantidade e valor, mas em diversos momentos precisaremos saber informações relativas ao produto, que é o objeto associado ao item. Na tela de vendas exibiremos itens que fazem parte daquela venda em uma listagem logo abaixo ao formulário no qual adicionamos itens à venda. Nesse formulário, exibimos descrição do produto, quantidade e preço de venda. Algumas dessas informações não existem no `Item`, somente no `Produto`, portanto teremos de implementar alguns métodos no `ItemRecord` para buscar informações no `ProdutoRecord`. Esses métodos serão `get_descricao()`, para retornar a descrição do produto baseado no atributo `id_produto`, e `get_preco_venda()`, para retornar o preço de venda do produto.

ItemRecord.class.php

```

<?php
/*
 * classe ItemRecord
 * Active Record para tabela Item
 */
class ItemRecord extends TRecord
{
    private $produto;
    /*
     * método get_descricao()
     * retorna a descrição do produto
     */
    function get_descricao()
    {
        // instancia ProdutoRecord, carrega
        // na memória o produto de código $this->id_produto
        if (empty($this->produto))
            $this->produto = new ProdutoRecord($this->id_produto);
        // retorna a descrição do produto instanciado
        return $this->produto->descricao;
    }
}

```

```

/*
 * método get_preco_venda()
 * retorna o preço de venda do produto
 */
function get_preco_venda()
{
    // instancia ProdutoRecord, carrega
    // na memória o produto de código $this->id_produto
    if (empty($this->produto))
        $this->produto = new ProdutoRecord($this->id_produto);

    // retorna o preço de venda do produto instanciado
    return $this->produto->preco_venda;
}
?

```

7.2.6.2 Aplicação

Nosso programa de vendas será formado por um formulário em que o usuário preencherá o código do produto e a quantidade a ser vendida. Quando ele fizer isso, clicará no botão **Adicionar**, o qual, por sua vez, tem como ação resultante a execução do método `onAdiciona()`. Este irá coletar dois dados do formulário (`id_produto` e `quantidade`) e, com eles, irá instanciar um objeto do tipo `ItemRecord`, para, então, armazená-lo em um array em memória chamado `$list`. Este array será armazenado na seção, portanto será uma informação que persistirá mesmo se o usuário navegar em outras opções do sistema (páginas) e retornar para continuar a venda. Esse formulário terá ainda um botão chamado **Finalizar**, o qual, ao ser clicado, executa o método `onFinal()` que abre uma janela para perguntar algumas informações antes de finalizar a venda, como o código do cliente. Veja na Figura 7.17 o formulário de vendas.

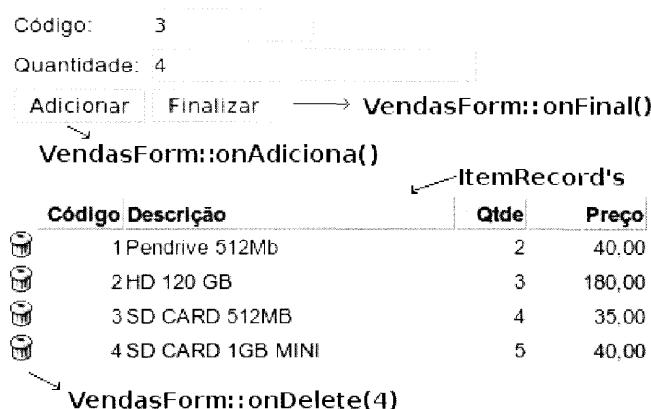


Figura 7.17 – Formulário de vendas.

 VendasForm.class.php

```
<?php
/*
 * função formata_string
 * exibe um valor com as casas decimais
 */
function formata_money($valor)
{
    return number_format($valor, 2, ',', '.');
}

/*
 * classe VendasForm
 * formulário de vendas
 */
class VendasForm extends TPage
{
    private $form;      // formulário de novo item
    private $datagrid; // listagem de itens

    /*
     * método construtor
     * cria a página e o formulário de cadastro
     */
    public function __construct()
    {
        parent::__construct();
        // instancia nova seção
        new TSession;

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

        // instancia uma tabela
        $table = new TTable;

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

        // cria os campos do formulário
        $codigo      = new TEntry('id_produto');
        $quantidade = new TEntry('quantidade');

        // define os tamanhos
        $codigo->setSize(100);
```

```
// adiciona uma linha para o campo código
$row=$table->addRow();
$row->addCell(new TLabel('Código:'));
$row->addCell($codigo);

// adiciona uma linha para o campo quantidade
$row=$table->addRow();
$row->addCell(new TLabel('Quantidade:'));
$row->addCell($quantidade);

// cria dois botões de ação para o formulário
$save_button = new TButton('save');
$fim_button = new TButton('fim');
// define as ações dos botões

$save_button->setAction(new TAction(array($this, 'onAdiciona')), 'Adicionar');
$fim_button->setAction(new TAction(array($this, 'onFinal')), 'Finalizar');

// adiciona uma linha para as ações do formulário
$row=$table->addRow();
$row->addCell($save_button);
$row->addCell($fim_button);

// define quais são os campos do formulário
$this->form->setFields(array($codigo, $quantidade, $save_button, $fim_button));
```

Abaixo do formulário do processo de venda, teremos uma listagem contendo algumas informações como o código do produto, a sua descrição, a quantidade e o preço de venda. Essa DataGrid irá conter um objeto do tipo `ItemRecord`, o qual implementamos anteriormente. A coluna `preco_venda` terá a função transformadora `formatar_money()`, a qual irá receber o conteúdo da coluna preço de venda da forma bruta que vem do banco de dados e irá formatar com os separadores e casas decimais corretos.

A DataGrid terá uma ação chamada “Deletar”, representada pelo ícone `ico_delete.png`. Esta ação excluirá o item da venda. Para isso, será executado o método `onDelete()`, que receberá o código do produto (`id_produto`). Como a lista de produtos vendidos estará na memória, armazenada em uma seção, o método `onDelete()` deverá excluir esse produto da seção. Note que as colunas descrição e preço de venda serão obtidas automaticamente pelos métodos getters da classe `ItemRecord`, instanciando o `Produto` correspondente baseado no `id_produto`, para então buscar tais propriedades.

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

// instancia as colunas da DataGrid
$codigo = new TGridColumn('id_produto', 'Código', 'right', 50);
```

```
$descricao = new TDataGridColumn('descricao', 'Descrição', 'left', 200);
$quantidade= new TDataGridColumn('quantidade', 'Qtde', 'right', 40);
$preco      = new TDataGridColumn('preco_venda', 'Preço', 'right', 70);

// define um transformador para a coluna preço
$preco->setTransformer('formata_money');

// adiciona as colunas à DataGrid
$this->datagrid->addColumn($codigo);
$this->datagrid->addColumn($descricao);
$this->datagrid->addColumn($quantidade);
$this->datagrid->addColumn($preco);

// cria uma ação para adatagrid
$action = new TDataGridAction(array($this, 'onDelete'));
$action->setLabel('Deletar');
$action->setImage('ico_delete.png');
$action->setField('id_produto');

// adiciona a ação à DataGrid
$this->datagrid->addAction($action);

// cria o modelo da DataGrid, montando sua estrutura
$this->datagrid->createModel();

// monta a página através de uma tabela
$table = new TTable;
// cria uma linha para o formulário
$row = $table->addRow();
$row->addCell($this->form);
// cria uma linha para adatagrid
$row = $table->addRow();
$row->addCell($this->datagrid);
// adiciona a tabela à página
parent::add($table);
}
```

O método `onAdiciona()` acrescentará os dados do item recém-preenchidos no formulário na seção. Para isso, recebemos os dados do formulário pelo método `getData()` e já instanciamos um objeto `ItemRecord` contendo esses dados. Em seguida, obtemos a variável `$list`, armazenada na seção para, posteriormente, acrescentar o `$item` digitado a ela. Dessa forma, a variável `$list` da seção irá conter um array de objetos do tipo `ItemRecord`. Depois de acrescentada à lista, armazenamos essa variável de volta à seção pelo método `setValue()` da classe `TSession`.

```
/*
 * método onAdiciona()
 * executada quando o usuário clicar no botão salvar do formulário
 */
function onAdiciona()
{
    // obtém os dados do formulário
    $item = $this->form->getData('ItemRecord');

    // lê variável $list da seção
    $list = TSession::getValue('list');

    // acrescenta produto na variável $list
    $list[$item->id_produto] = $item;

    // grava variável $list de volta à seção
    TSession::setValue('list', $list);

    // recarrega a listagem
    $this->onReload();
}
```

O método `onDelete()` é executado sempre que o usuário clicar na ação “Deletar” na lista, no botão de excluir presente em cada linha de itens adicionados à venda. Esse método receberá como parâmetro toda a URL, a qual, na posição `$key`, contém o código do produto sobre o qual o usuário clicou. O funcionamento desse método é bastante simples: primeiro ele obtém a variável `$list` da seção, em seguida, elimina a posição dessa lista indexada pela chave recebida (código do produto), para finalmente armazená-la outra vez na seção por meio do método `setValue()`.

```
/*
 * método onDelete()
 * executada quando o usuário clicar no botão excluir dadatagrid
 */
function onDelete($param)
{
    // lê variável $list da seção
    $list = TSession::getValue('list');
    // exclui a posição que armazena o produto de código $key
    unset($list[$param['key']]);
    // grava variável $list de volta à seção
    TSession::setValue('list', $list);

    // recarrega a listagem
    $this->onReload();
}
```

O método `onReload()` será executado sempre pelo método `show()` da página, caso ainda não tenha sido executado. Sua função é preencher a listagem (`TDataGrid`) com uma série de objetos. Neste caso específico do processo de venda, os objetos não estão armazenados no banco de dados, como nos exemplos anteriores, mas na seção. Dessa forma, obteremos tais objetos pelo método `getValue()` da classe `TSession`, para posteriormente percorrê-los por meio de um `foreach`, adicionando um a um na listagem por meio de seu método `addItem()`.

Neste método, estamos abrindo uma transação com o banco de dados, pois estamos exibindo as colunas `descricao` e `preco_venda`, que não existem em um objeto do tipo `ItemRecord`. Como elas não existem, serão executados automaticamente os métodos `get_descricao()` e `get_preco_venda()`, obtendo tais informações a partir de um objeto `ProdutoRecord`. Para isso, precisamos estar conectados ao banco de dados.

```
/*
 * método onReload()
 * carrega a DataGrid com os objetos
 */
function onReload()
{
    // obtém a variável de seção $list
    $list = TSession::getValue('list');
    // limpa a datagrid
    $this->datagrid->clear();
    if ($list)
    {
        // inicia transação com o banco 'pg_livro'
        TTransaction::open('pg_livro');
        // percorre o array $list
        foreach ($list as $item)
        {
            // adiciona cada objeto $item nadatagrid
            $this->datagrid->addItem($item);
        }
        // fecha transação
        TTransaction::close();
    }
    $this->loaded = true;
}
```

O grande momento do sistema é quando a venda é finalizada. Nesse instante, abrimos uma janela na tela, instanciando um objeto do tipo `Twindow`. Esta janela irá conter um formulário de finalização da venda, no qual preencheremos o código do cliente, o valor do desconto (caso exista) e também o valor pago. Antes disso, porém, totalizaremos a venda. Para tanto, iremos obter o array `$list` da seção, percorrer

cada um de seus objetos (que são do tipo `ItemRecord`) e multiplicar seu “`preco_venda`” por “`quantidade`”. Como a propriedade `preco_venda` não existe, será buscada a partir do método `get_preco_venda()` existente na classe `ItemRecord`, por isso estamos abrindo uma transação com a base de dados. O valor total da venda será utilizado para preenchermos automaticamente o campo “Valor Total” do formulário de conclusão da venda, por meio do método `setData()` do formulário `ConcluiVendaForm`.

Depois de totalizar a venda, precisamos abrir o formulário com os campos a serem preenchidos. Para não tornarmos esse programa muito extenso, declarando o formulário dentro desse método, optamos por criar a classe `ConcluiVendaForm`, que irá conter os campos necessários nesta fase de conclusão da venda. Veremos como é a classe `ConcluiVendaForm` a seguir. Ela terá a propriedade `$button`, cujo conteúdo é um botão **Salvar**. Ao ser clicado, este botão irá gravar a venda na base de dados e preparar a tela para uma nova venda. Veja a seguir que este botão está sendo vinculado ao método `onGravaVenda()` por meio do método `setAction()`. Veja na Figura 7.18 o formulário de conclusão da venda.

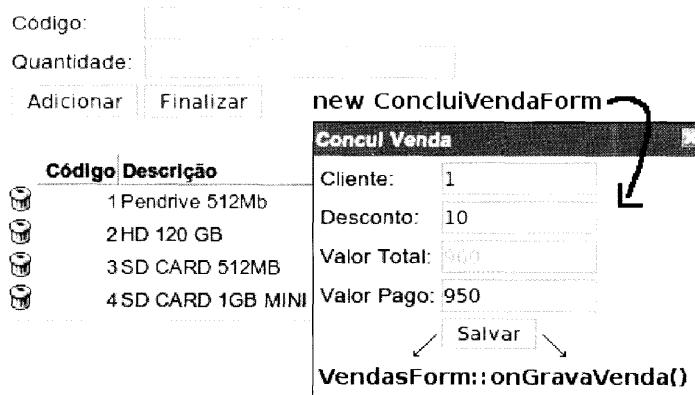


Figura 7.18 – Formulário de conclusão da venda.

```
/*
 * método onFinal()
 * executada quando o usuário finalizar a Venda
 */
function onFinal()
{
    // instancia uma nova janela
    $janela = new TWindow('Concui Venda');
    $janela->setPosition(520,200);
    $janela->setSize(250,180);

    // lê a variável $list da seção
    $list = TSession::getValue('list');
```

```

// inicia transação com o banco 'pg_livro'
TTransaction::open('pg_livro');
foreach ($list as $item)
{
    // soma o total de produtos vendidos
    $total += $item->preco_venda * $item->quantidade;
}
// fecha a transação
TTransaction::close();

// instancia formulário de conclusão de venda
$form = new ConcluiVendaForm;
// define a ação do botão deste formulário
$form->button->setAction(new TAction(array($this, 'onGravaVenda')), 'Salvar');
// preenche o formulário com o valor_total
$dados->valor_total = $total;
$form->setData($dados);

// adiciona o formulário à janela
$janela->add($form);
$janela->show();
}

```

Depois de clicar no botão **Salvar**, o programa é redirecionado para o método `onGravaVenda()`, o qual recebe os dados do formulário de conclusão da venda (`ConcluiVendaForm`) pelo método `getData()`, presente na classe `TForm`. Em seguida, abrimos uma transação com o banco de dados e instanciamos um objeto `VendaRecord`. Antes de armazenar a venda, definimos os valores de seus atributos. O código do cliente (`id_cliente`), o desconto, o valor total e o valor pago virão do formulário `ConcluiVendaForm`; a data da venda, por sua vez, é gerada a partir da função `date()` do PHP. Depois de definirmos os atributos de uma venda, obtemos a variável `$list` a partir da seção, por meio do método `getValue()` e percorremos cada uma de suas posições por meio de um `foreach`. Cada `$item` resultante (objeto `ItemRecord`) é adicionado à venda pelo seu método `addItem()`.

Depois de adicionarmos os itens à venda, executamos o método `store()` do objeto `VendaRecord`. Lembre-se que esse método armazenará não somente o objeto venda na base de dados, mas também os seus objetos agregados (`ItemRecords`).

```

/*
* método onGravaVenda()
* executada quando o usuário Finalizar a venda
*/

```

```
function onGravaVenda()
{
    // obtém os dados do formulário de conclusão de venda
    $form = new ConcluiVendaForm;
    $dados = $form->getData();

    // inicia transação com o banco 'pg_livro'
    TTransaction::open('pg_livro');

    // instancia novo objeto VendaRecord
    $venda = new VendaRecord;

    // define os atributos a serem gravados
    $venda->id_cliente = $dados->id_cliente;
    $venda->data_venda = date('Y-m-d');
    $venda->desconto = $dados->desconto;
    $venda->valor_total = $dados->valor_total;
    $venda->valor_pago = $dados->valor_pago;
    // lê a variável $list da seção
    $itens = TSession::getValue('list');
    if ($itens)
    {
        // percorre os itens
        foreach ($itens as $item)
        {
            // adiciona o item na venda
            $venda->addItem($item);
        }
    }
    // armazena venda no banco de dados
    $venda->store();

    // finaliza a transação
    TTransaction::close();

    // limpa lista de itens da seção
    TSession::setValue('list', array());

    // exibe mensagem de sucesso
    new TMessage('info', 'Venda registrada com sucesso');

    // recarrega lista de itens
    $this->onReload();
}
```

O método `show()` exibe a página na tela, mas antes verifica se o método `onReload()` já fora executado, carregando os itens na listagem.

```

/*
 * função show()
 * executada quando o usuário clicar no botão excluir
 */
function show()
{
    if (!$this->loaded)
    {
        $this->onReload();
    }
    parent::show();
}
?>

```

A classe `ConcluiVendaForm` representa o formulário de conclusão da venda, exibido pelo método `onFinal()` da classe anterior. Esta classe basicamente estende a classe `TForm` e adiciona alguns campos no formulário, como o código do cliente, o desconto, o valor total e o valor pago. Estes campos são dispostos na tela por uma tabela (`TTable`). O formulário terá um botão de ação (`$this->button`), cuja ação não vai ser definida neste formulário; a ação do botão será definida pela classe que fizer uso deste formulário. Veja que a sua ação fora definida dentro do método `onFinal()` da classe anterior. Para permitir isso, tivemos de declarar este botão como propriedade pública desta classe, sendo assim acessível de fora do escopo dela.

ConcluiVendaForm.class.php

```

<?php
/*
 * classe ConcluiVenda
 * formulário de conclusão de venda
 */
class ConcluiVendaForm extends TForm
{
    public $button; // botão de ação do formulário

    /*
     * método construtor
     * Cria a página e o formulário de cadastro
     */
    function __construct()
    {
        parent::__construct('form_conclui_venda');
        // instancia uma tabela
        $table = new TTable;

```

```
// adiciona a tabela ao formulário
parent::add($table);

// cria os campos do formulário
$cliente      = new TEntry('id_cliente');
$desconto     = new TEntry('desconto');
$valor_total  = new TEntry('valor_total');
$valor_pago   = new TEntry('valor_pago');

// define alguns atributos para os campos do formulário
$valor_total->setEditable(FALSE);
$cliente->setSize(100);
$desconto->setSize(100);
$valor_total->setSize(100);
$valor_pago->setSize(100);

// adiciona uma linha para o campo cliente
$row=$table->addRow();
$row->addCell(new TLabel('Cliente:'));
$row->addCell($cliente);

// adiciona uma linha para o campo desconto
$row=$table->addRow();
$row->addCell(new TLabel('Desconto:'));
$row->addCell($desconto);

// adiciona uma linha para o campo valor total
$row=$table->addRow();
$row->addCell(new TLabel('Valor Total:'));
$row->addCell($valor_total);

// adiciona uma linha para o campo valor pago
$row=$table->addRow();
$row->addCell(new TLabel('Valor Pago:'));
$row->addCell($valor_pago);

// cria um botão de ação para o formulário
$this->button=new TButton('action1');

// adiciona uma linha para as ações do formulário
$row=$table->addRow();
$row->addCell('');
$row->addCell($this->button);

// define quais são os campos do formulário
parent::setFields(array($cliente, $desconto, $valor_total, $valor_pago, $this->button));
}

?>
```

7.2.7 Emissão de relatórios

Com os exemplos anteriores, estudamos alguns recursos necessários a toda aplicação de negócios, como formulários de cadastro, listagem, edição e exclusão de registros, dentre outros. Como toda a aplicação, a nossa também terá de apresentar alguns relatórios ao usuário, os quais geralmente demandam maior flexibilidade na coleta de informações no banco de dados. Na maioria das vezes, precisamos buscar informações em diversas tabelas diferentes até mostrar o que o usuário realmente precisa em tela.

Como aqui não temos espaço suficiente para construção de vários relatórios, criaremos somente um que contém diversas informações relacionadas. Nosso relatório exibirá em tela uma tabela contendo as vendas de um período, dentro do qual serão exibidas em tela a data da venda, o cliente para o qual a venda fora realizada e todos os itens que compuseram aquela venda: descrição do produto, quantidade, preço e um subtotal.

7.2.7.1 Modelo

Para emitir o relatório de vendas, precisaremos incrementar nossa classe `VendaRecord` com alguns recursos adicionais. Em primeiro lugar, no procedimento de vendas, alteramos o método `store()` para que, além de armazenar a venda em si, armazene também os itens da venda. Agora precisamos de um método que faça o caminho contrário, ou seja, que carregue esses itens da venda em memória para que possamos percorrê-los e exibi-los no relatório. Para isso, iremos criar o método `get_itens()`, que instancia um repositório para `ItemRecord` e carrega todos os itens que sejam relacionados com determinada venda (campo `id_venda`). Depois de carregar esses objetos em memória pelo método `load()`, eles são retornados para o usuário. Lembre-se que não será necessário executarmos diretamente o método `get_itens()`, pois a classe-pai (`TRecord`) já verifica a existência de métodos getters (iniciados por `get_`) sempre que uma propriedade não existir. Assim, só precisaremos acessar a propriedade `$venda->itens` para que tal método seja executado.

Também precisamos descobrir as informações do cliente relacionado à venda, tais como `nome` e possivelmente `endereco` e `telefone`. Não sabendo exatamente de quais informações realmente precisaremos, optamos por, neste momento, retornar o objeto inteiro pelo método `get_cliente()`, que instancia um objeto `ClienteRecord` e o retorna ao usuário. Lembre-se que, assim como no método `get_itens()`, não será necessário executarmos diretamente o método `get_cliente()`; basta acessarmos a sua propriedade `$venda->cliente`, que este método será automaticamente executado.

 VendaRecord.class.php (continuação...)

```
<?php
/*
 * classe VendaRecord
 * Active Record para tabela Venda
 */
class VendaRecord extends TRecord
{
    private $itens; // array de objetos do tipo ItemRecord

    ...
    ...

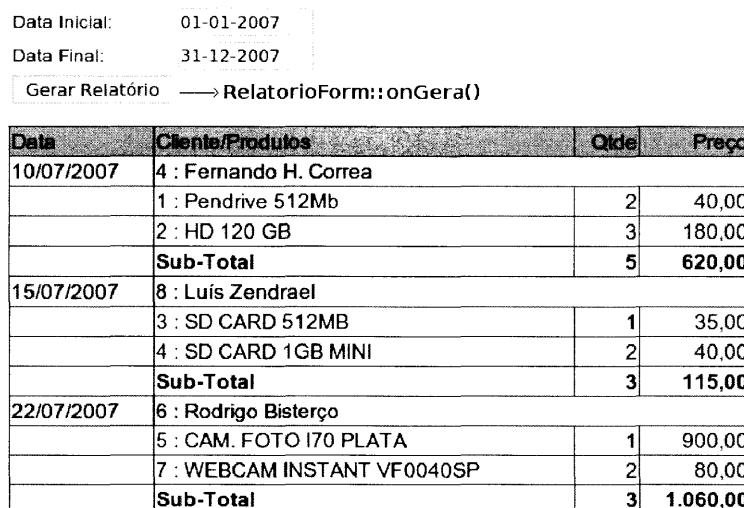
    /*
     * função get_itens()
     * retorna os itens da venda
     */
    public function get_itens()
    {
        // instancia um repositório de Item
        $repositorio = new TRepository('Item');
        // define o critério de seleção
        $criterio = new TCriteria;
        $criterio->add(new TFilter('id_venda', '=', $this->id));
        // carrega a coleção de itens
        $this->itens = $repositorio->load($criterio);
        // retorna os itens
        return $this->itens;
    }

    /*
     * método get_cliente()
     * retorna o objeto cliente vinculado à venda
     */
    function get_cliente()
    {
        // instancia ClienteRecord, carrega
        // na memória o cliente de código $this->id_cliente
        $cliente = new ClienteRecord($this->id_cliente);

        // retorna o objeto instanciado
        return $cliente;
    }
}
```

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.



The screenshot shows a web-based application interface. At the top, there are two input fields: 'Data Inicial' with the value '01-01-2007' and 'Data Final' with the value '31-12-2007'. Below these is a button labeled 'Gerar Relatório' with the action 'RelatorioForm::onGera()'. A horizontal line separates this from a table below. The table has four columns: 'Data', 'Cliente/Produtos', 'Qtd', and 'Preço'. It contains three groups of data, each representing a different customer's purchases:

Data	Cliente/Produtos	Qtd	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
     */
}
```

```
public function __construct()
{
    parent::__construct();
    // instancia um formulário
    $this->form = new TForm('form_relat_vendas');

    // instancia uma tabela
    $table = new TTable;

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

    // cria os campos do formulário
    $data_ini = new TEntry('data_ini');
    $data_fim = new TEntry('data_fim');
    // define os tamanhos
    $data_ini->setSize(100);
    $data_fim->setSize(100);

    // adiciona uma linha para o campo data inicial
    $row=$table->addRow();
    $row->addCell(new TLabel('Data Inicial:'));
    $row->addCell($data_ini);

    // adiciona uma linha para o campo data final
    $row=$table->addRow();
    $row->addCell(new TLabel('Data Final:'));
    $row->addCell($data_fim);

    // cria um botão de ação
    $gera_button=new TButton('gera');
    // define a ação do boão
    $gera_button->setAction(new TAction(array($this, 'onGera')), 'Gerar Relatório');

    // adiciona uma linha para a ação do formulário
    $row=$table->addRow();
    $row->addCell($gera_button);

    // define quais são os campos do formulário
    $this->form->setFields(array($data_ini, $data_fim, $gera_button));

    // adiciona o formulário à página
    parent::add($this->form);
}
```

O método `onGera()` será executado sempre que o usuário clicar no botão **Gerar Relatório**. Esse método irá capturar os dados do formulário por meio do método `getData()` e então irá converter as datas para o formato americano pelo método `conv_data_to_us()`, visto que o banco de dados está com este padrão.

A partir daí, instanciamos uma tabela e adicionamos uma linha para o cabeçalho, contendo informações como a data da venda, o cliente, seus produtos, a quantidade e o preço pago. Dentro de um grande controle de exceções `try/catch`, abrimos uma transação com a base de dados para carregar todas as vendas cuja data (`data_venda`) está entre os períodos digitados pelo usuário. Carregamos os objetos por um repositório (`TRepository`) e então percorremos venda a venda por um `foreach`.

Para cada venda carregada do banco de dados, adicionamos uma linha contendo a data da venda (`$venda->data_venda`), o código (`$venda->id_cliente`) e nome do cliente (`$venda->cliente->nome`). Em seguida, verificamos se a venda possui itens. Em caso afirmativo, percorremos os itens da venda por um `foreach` sobre a propriedade `$venda->itens`. Lembre-se de que essa propriedade será o resultado do método `get_itens()`. Para cada item, adicionamos uma linha com suas informações: `id_produto` (código), `descricao`, `quantidade` e `preco`, que é formatado devidamente com as casas decimais pela função `number_format()`. Ao final, adicionamos uma linha contendo os totais da coluna `quantidade` e da coluna `preco`.

```
/*
 * método onGera
 * gera o relatório, baseado nos parâmetros do formulário
 */
function onGera()
{
    // obtém os dados do formulário
    $dados = $this->form->getData();
    // joga os dados de volta ao formulário
    $this->form->setData($dados);

    // lê os campos do formulário, converte para o padrão americano
    $data_ini = $this->conv_data_to_us($dados->data_ini);
    $data_fim = $this->conv_data_to_us($dados->data_fim);

    // instancia uma nova tabela
    $table = new TTable;
    $table->border = 1;
    $table->width  = '100%';
    $table->style = 'border-collapse:collapse';

    // adiciona uma linha para o cabeçalho do relatório
    $row = $table->addRow();
    $row->bgcolor = '#a0a0a0';
    // adiciona as células ao cabeçalho
    $cell = $row->addCell('Data');
    $cell = $row->addCell('Cliente/Produtos');
    $cell = $row->addCell('Qtde');
```

```
$cell->align = 'right';
$cell = $row->addCell('Preço');
$cell->align = 'right';

try
{
    // inicia transação com o banco 'pg_livro'
    TTransaction::open('pg_livro');

    // instancia um repositório da classe VendaRecord
    $repositorio = new TRepository('Venda');
    // cria um critério de seleção por intervalo de datas
    $criterio = new TCriteria;
    $criterio->add(new TFilter('data_venda', '>=', $data_ini));
    $criterio->add(new TFilter('data_venda', '<=', $data_fim));
    $criterio->setProperty('order', 'data_venda');

    // lê todas vendas que satisfazem ao critério
    $vendas = $repositorio->load($criterio);
    // verifica se retornou algum objeto
    if ($vendas)
    {
        // percorre as vendas
        foreach ($vendas as $venda)
        {
            // adiciona uma linha à tabela e define suas propriedades
            $row = $table->addRow();
            $row->bgcolor = "#e0e0e0";
            // adiciona células para data da venda e dados do cliente
            $cell = $row->addCell($this->conv_data_to_br($venda->data_venda));
            $cell = $row->addCell($venda->id_cliente . ' : ' . $venda->cliente->nome);
            $cell->colspan=3;

            // verifica se a venda possui itens
            if ($venda->itens)
            {
                $sub_total =0;
                $total_qtde=0;
                // percorre os itens da venda
                foreach ($venda->itens as $item)
                {
                    // adiciona uma linha para cada item da venda
                    $row = $table->addRow();
                    // adiciona as células com os dados do item
                    $cell = $row->addCell('');
                    $cell = $row->addCell($item->id_produto . ' : ' .
                                         $item->descricao);
                }
            }
        }
    }
}
```

```

        $cell = $row->addCell($item->quantidade);
        $cell->align = 'right';
        $cell = $row->addCell(number_format($item->preco_venda,2,',','.'));
        $cell->align = 'right';
        // acumula totais de valor e quantidade
        $sub_total += $item->quantidade * $item->preco_venda;
        $total_qtde += $item->quantidade;
    }
    // adiciona uma linha para os totais da venda
    $row = $table->addRow();
    $cell = $row->addCell('');
    $cell = $row->addCell('<b>Sub-Total</b>');
    $cell = $row->addCell('<b>' . $total_qtde . '</b>');
    $cell->align = 'right';
    $cell = $row->addCell('<b>' . number_format($sub_total,2,',','.') . '</b>');
    $cell->align = 'right';
}
}
}
// finaliza a transação
TTransaction::close();
}
catch (Exception $e) // em caso de exceção
{
    // exibe a mensagem gerada pela exceção
    new TMessage('error', $e->getMessage());
    // desfaz todas alterações no banco de dados
    TTransaction::rollback();
}
// adiciona a tabela à página
parent::add($table);
}

/*
* método conv_data_to_us()
* Converte uma data para o formato yyyy-mm-dd
* @param $data = data no formato dd/mm/yyyy
*/
function conv_data_to_us($data)
{
    $dia = substr($data,0,2);
    $mes = substr($data,3,2);
    $ano = substr($data,6,4);
    return "{$ano}-{$mes}-{$dia}";
}

```

```
/*
 * 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 açãoem serviços e se comuniquem umas com as outras, como ilustrado a seguir.

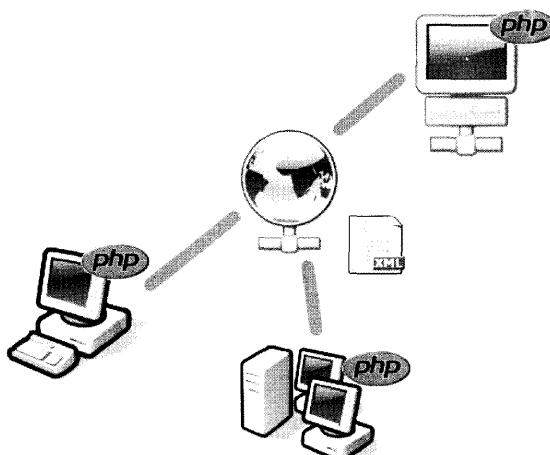


Figura 7.20 – Web Services.

A internet, que começou como um esforço de pesquisa militar e educacional, tornou-se a base do comércio do século XXI. Nesse contexto, os Web Services oferecem um novo modo de comunicação entre aplicações e novas formas de negócio pela web. Web Services podem ser utilizados dentro das empresas (intranet), comunicando aplicações já existentes, mas sua maior vantagem fica visível quando esses são expostos na internet combinando serviços entre organizações.

7.3.2 Arquitetura

Na Figura 7.21 temos um panorama geral do funcionamento de Web Services, por meio de um conjunto de tecnologias de padrão aberto, interagindo sob uma plataforma de internet. Temos o papel do requerente, que é a aplicação que interage com o serviço, podendo ser desde um desktop ou servidor até um celular ou palmtop. Temos o servidor de aplicação, que provê o serviço e se comunica com o requerente por meio de um pacote SOAP, e temos o registro do Web Service codificado no formato WSDL, como se fosse um guia telefônico, descrevendo os serviços que a aplicação fornece.

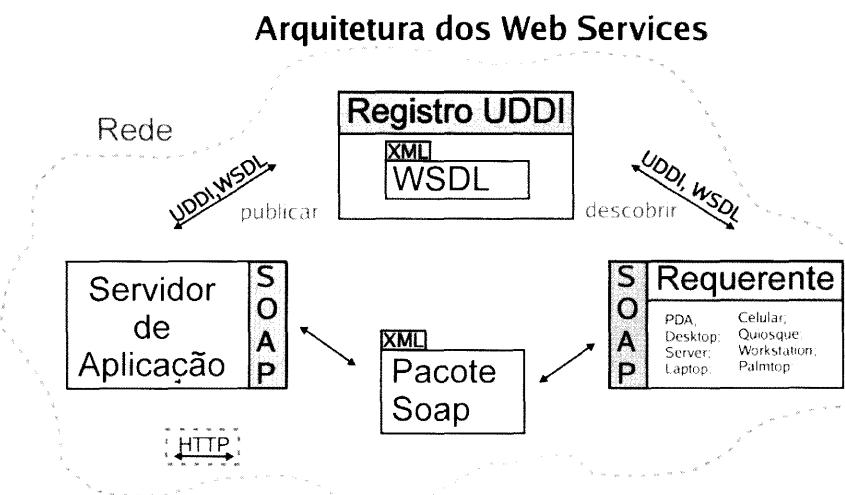


Figura 7.21 – Arquitetura dos Web Services.

A aplicação cliente envia para aplicação servidora um pacote XML por meio do protocolo SOAP. A aplicação servidora processa a requisição de acordo com suas regras internas de negócio e retorna ao cliente a resposta também por meio de um pacote XML pelo protocolo SOAP.

Tendo em vista esse panorama, fica claro que a utilização de padrões abertos (SOAP é mantido pelo W3C) e amplamente conhecidos para implementação/distribuição de Web Services são fatores preponderantes que acabam por levar ao sucesso de sua adoção.

HTTP (Hypertext Transfer Protocol) é o protocolo-padrão usado sobre a porta 80, responsável pela requisição e transmissão de dados sobre a internet. XML (Extensible Markup Language) é a linguagem de marcação utilizada para descrever a informação. Ambos são padrões utilizados mundialmente.

No meio de todo o processo, provendo a comunicação entre as aplicações, está o protocolo SOAP (Simple Object Access Protocol). SOAP é um protocolo herdeiro do padrão XML que encapsula um conjunto de regras para descrição de dados e processos por meio de um mecanismo simples para definir a semântica de uma aplicação por um modelo de empacotamento e um mecanismo de codificação. É projetado para a troca de informações em um ambiente descentralizado pelo protocolo de comunicação HTTP e pelo formato XML. Dessa forma, para uma aplicação trabalhar com WS, basta a compatibilidade com SOAP, tanto no lado do cliente (criando o documento XML com a informação necessária para invocar o serviço) quanto no lado do servidor (responsável por executar a mensagem como um interpretador).

SOAP é peça central de um Web Service porque provê um mecanismo leve para troca de informação estruturada entre pares, em um ambiente descentralizado e distribuído, usando XML. SOAP em si não define o modelo de programação. Define um mecanismo simples para expressar a semântica da aplicação por meio de um modelo de pacotes.

Para descrever os serviços é utilizada a linguagem WSDL (Web Services Description Language), a qual se baseia no formato XML que descreve um WS pela definição das interfaces e os mecanismos de interação, contendo informações como o protocolo, formato de dados, segurança, dentre outras. O WSDL descreve um conjunto de mensagens SOAP e fornece informações necessárias para os clientes interagirem com ele. Ademais, ele especifica a localização do Web Service, as operações que estão disponíveis, os tipos de dados intercambiados e os protocolos de comunicação que serão utilizados.

Utilizando estes padrões abertos, os desenvolvedores podem criar componentes abertos, que podem ser acessados de qualquer plataforma ou linguagem de programação capaz de se comunicar com os conhecidos protocolos da internet.

7.3.3 Funcionamento

Para demonstrar o funcionamento de WS em PHP, utilizaremos as classes nativas para manipulação do protocolo SOAP, desenvolvido por Dmitry Stogov. Através da utilização de SOAP em PHP, poderíamos escrever um código em PHP que envia uma pesquisa (encapsulada em XML pelo SOAP) para uma aplicação de banco de dados em C++ localizada em outro continente para obter o preço de um livro, por exemplo.

O funcionamento é simples, a transação inicia quando a aplicação cliente realiza a chamada remota de uma função enviando um pacote SOAP contendo a descrição do método a ser invocado via HTTP. O servidor recebe o pacote SOAP, interpreta-o, executa a função correspondente e retorna a resposta da execução também encapsulada via SOAP. Para demonstração, construiremos um exemplo de Web Service em PHP para simular a requisição de dados de um cliente. Os dados estão armazenados em um banco de dados PostgreSQL ao lado do servidor, e a aplicação cliente poderá ser executada de qualquer parte do mundo, abrindo uma conexão com o servidor e executando seu método `getNome()`, passando o código do cliente e obtendo seus dados.

Para construir este Web Service em PHP, o primeiro passo é criar o arquivo WSDL (Web Services Description Language), que descreverá o funcionamento do Web Service. A seção `message` descreve processos de requisição (`request`) e resposta (`response`) de uma determinada funcionalidade do Web Service, que neste caso é a função `getNome`. Cada processo possui seus parâmetros, bem como os tipos de dados. A seção `portType` define uma funcionalidade do Web Service e indica quais processos de requisição e resposta serão utilizados para tal. Neste caso, `getNomeRequest` para `input` e `getNomeResponse` para `output`. A seção `binding` define como as mensagens devem ser transmitidas e codificadas. Neste caso, via RPC usando SOAP sobre HTTP. E por fim, a seção `service` define a URL na qual o serviço está rodando. Neste caso, no servidor Apache local.

exemplo.wsdl

```
<?xml version ='1.0' encoding ='ISO-8859-1' ?>
<definitions name='Exemplo'
    targetNamespace='urn:Exemplo'
    xmlns:tns='http://example.org/Exemplo'
    xmlns:soap='http://schemas.xmlsoap.org/wsdl/soap/'
    xmlns:xsd='http://www.w3.org/2001/XMLSchema'
    xmlns:soapenc='http://schemas.xmlsoap.org/soap/encoding/'
    xmlns:wsdl='http://schemas.xmlsoap.org/wsdl/'
    xmlns='http://schemas.xmlsoap.org/wsdl/'>

<message name='getNomeRequest'>
    <part name='codigo' type='xsd:string'/>
</message>
<message name='getNomeResponse'>
    <part name='resultado' type='xsd:string[]' />
</message>

<portType name='ExemploPortType'>
    <operation name='getNome'>
        <input message='tns:getNomeRequest' />
        <output message='tns:getNomeResponse' />
    </operation>
</portType>
```

```
<binding name='ExemploBinding' type='tns:ExemploPortType'>
    <soap:binding style='rpc' transport='http://schemas.xmlsoap.org/soap/http'/>
    <operation name='getNome'>
        <soap:operation soapAction='exemplo#getNome'/>
        <input>
            <soap:body use='encoded' namespace='exemplo'
                encodingStyle='http://schemas.xmlsoap.org/soap/encoding/' />
        </input>
        <output>
            <soap:body use='encoded' namespace='exemplo'
                encodingStyle='http://schemas.xmlsoap.org/soap/encoding/' />
        </output>
    </operation>
</binding>

<service name='ExemploService'>
    <port name='ExemploPort' binding='ExemploBinding'>
        <soap:address location='http://127.0.0.1/servidor.php' />
    </port>
</service>
</definitions>
```

O programa servidor basicamente é construído com a classe `SoapServer`. O servidor deve disponibilizar serviços e, para tanto, criamos funções e adicionamos a ele pelo método `addFunction()`, identificando o nome da função que desejamos disponibilizar externamente. Neste exemplo, estamos adicionando a função `getNome()`, a qual recebe o código de um cliente, consulta seus dados no banco de dados e retorna todas as suas informações na forma de um vetor. Caso não consiga obter as informações, retorna os devidos erros, comentados mais adiante. O servidor começa a “aguardar” por requisições a partir do método `handle()`.

servidor.php

```
<?php
function getNome($codigo)
{
    // verifica a passagem do parâmetro
    if (!$codigo)
    {
        throw new SoapFault('Client', 'Parâmetro não preenchido');
    }

    // conecta ao banco de dados
    $id = @pg_connect("dbname=livro user=postgres");

    if (!$id)
        throw new SoapFault("Server", "Conexão não estabelecida");
```

```

// realiza consulta ao banco de dados
$result = pg_query($id, "select * from pessoa where id = '$codigo'");
$matriz = pg_fetch_all($result);
if ($matriz == null)
    throw new SoapFault("Server", "Cliente não encontrado");
// retorna os dados
return $matriz[0];
}
// instancia servidor SOAP
$server = new SoapServer("exemplo.wsdl", array('encoding'=>'ISO-8859-1'));
$server->addFunction("getNome");
$server->handle();
?>

```

A seguir, temos o código da aplicação cliente, que estabelece conexão com o servidor da aplicação. Para construir a aplicação cliente, precisamos somente instanciar um objeto do tipo `SoapClient`, informando alguns parâmetros como o descritor WSDL. A partir daí, podemos executar os métodos disponibilizados pelo servidor como se fossem métodos do objeto `SoapClient`. Veja como executamos o método `getNome()`. Depois de executar a função `getNome()`, exibimos os dados de retorno em formato HTML (no Browser).

cliente.php

```

<?php
// instancia cliente SOAP
$client = new SoapClient("exemplo.wsdl", array('encoding'=>'ISO-8859-1'));
try
{
    // realiza chamada remota de método
    $retorno = $client->getNome(3);

    // imprime os dados de retorno
    echo '<table border=1 width=300>';
    echo '<tr bgcolor="#c0c0c0"><td>Coluna </td> <td> Conteúdo </td></tr>';
    echo '<tr><td>Código </td> <td>' . $retorno['id'] . '</td></tr>';
    echo '<tr><td>Nome </td> <td>' . $retorno['nome'] . '</td></tr>';
    echo '<tr><td>Endereço </td> <td>' . $retorno['endereco'] . '</td></tr>';
    echo '<tr><td>Data Nas.</td> <td>' . $retorno['datanasc'] . '</td></tr>';
    echo '</table>';
}
catch (SoapFault $excecao) // ocorrência de erro
{
    echo "Erro: ";
    echo "<b> {$excecao->faultstring} </b>";
}
?>

```

Caso não ocorram erros, serão exibidos no browser os dados requisitados pela aplicação cliente, como na imagem a seguir.

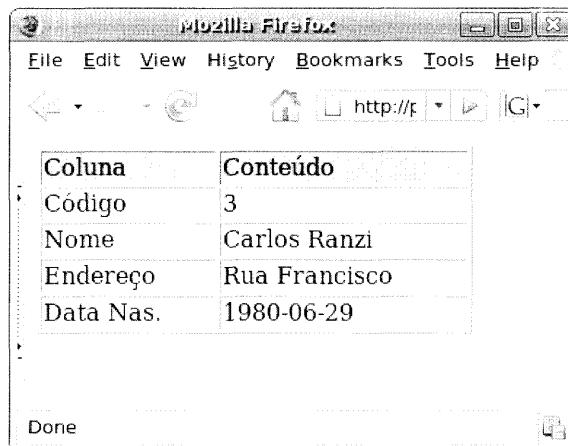


Figura 7.22 – Resultado da execução do Web Service.

Caso o parâmetro do método remoto não seja informado, a mensagem de erro retornada será “Parâmetro não preenchido”, e, caso o banco de dados não seja conectado devidamente, a mensagem de erro retornada será “Conexão não estabelecida”.

7.3.4 Remote Facade

Como vimos no Capítulo 2, cada objeto tem uma responsabilidade, ou um papel bem definido dentro do sistema, o que nos leva invariavelmente a ter uma quantidade relativamente grande de objetos no sistema. Lembre-se de que quanto mais classes houver, melhor, uma vez que diminuímos a responsabilidade de uma única classe e passamos a ter a responsabilidade do sistema distribuída ao longo das classes. Essa distribuição nos leva a um menor nível de acoplamento e a uma maior possibilidade de reuso dos objetos.

A orientação a objetos não nos leva somente a distribuir a responsabilidade do sistema ao longo dos objetos, mas também faz com que cada objeto forneça uma interface (conjunto de métodos) que nos possibilite acessar comportamentos bem específicos. É preferível ter vários métodos em uma classe, cada um com um papel bem definido, apesar de conter poucas linhas de código, do que um supermétodo que recebe 20 parâmetros e realiza 30 coisas diferentes, cada uma delas que você só saberá se ler e interpretar o próprio código-fonte. Ter uma quantidade grande de métodos com comportamentos bem definidos também aumenta a possibilidade de reuso de funcionalidades já existentes, além de diminuir o acoplamento. Quanto maior a implementação de um método, mais difícil será sua manutenção.

Vimos que a orientação a objetos nos leva naturalmente a uma certa quantidade de classes/objetos e métodos com comportamentos bem definidos no sistema. Vimos que isso é uma característica positiva dentro da aplicação. Em alguns contextos, porém, essa característica deixa de ser positiva para se tornar um fator preocupante. Talvez o maior exemplo de situação em que isso acontece é na comunicação entre aplicações. Quando temos uma aplicação conversando com outra, independente do meio e protocolo no qual isso acontece, existe um grande intercâmbio de informações que é realizado pela invocação de métodos (interface da aplicação). Dessa forma, uma aplicação disponibiliza uma interface de comunicação (vamos chamar de aplicação servidora) para uma aplicação cliente. A aplicação cliente instancia objetos ou invoca métodos que fazem parte da interface da aplicação servidora. Dessa forma, as duas se comunicam.

Até aí tudo bem, mas imagine a seguinte situação: uma aplicação cliente resolve armazenar um objeto na aplicação servidora, ou a aplicação cliente resolve obter uma coleção de objetos a partir da aplicação servidora. Nas duas situações expostas, a quantidade de objetos e métodos necessários para que tal ação se concretize é grande. Assim, a única forma de isso acontecer seria a aplicação servidora disponibilizar os vários objetos e métodos para que a aplicação cliente possa invocá-los. Isso, no entanto, também não é muito desejável porque em determinado momento estaremos disponibilizando uma quantidade de métodos que nem sempre são necessários, e sua exposição pode comprometer inclusive a segurança da aplicação. Contudo, o fator mais preocupante talvez seja o tráfego de rede. Na maioria das vezes em que uma aplicação conversa com outra, isso se dá pelo protocolo HTTP, utilizando uma tecnologia como Web Services. Neste caso, quanto mais objetos e métodos disponibilizados da aplicação servidora para a aplicação cliente, mais chamadas ocorrem e, consequentemente, mais tráfego de rede acontece. O tráfego de rede é um fator complicador quando temos várias aplicações cliente invocando métodos da aplicação servidora.

Para resolver a situação exposta precisamos construir uma interface enxuta que concentre maior responsabilidade no lado da aplicação servidora e seja responsável internamente por diversas chamadas a objetos/métodos que desta vez estarão localizados no mesmo lado (aplicação servidora). Esta alternativa permite que a aplicação cliente faça reduzidas chamadas à aplicação servidora. Em vez de invocarmos vários objetos e métodos para se atingir determinado objetivo, concentraremos esta lógica na aplicação servidora em uma camada que damos o nome de Fachada (Facade). Como estamos em um ambiente de rede com invocações remotas chamamos este pattern de Fachada Remota (Remote Facade).

Na Figura 7.23 procuramos demonstrar isso. Temos uma aplicação servidora formada por um conjunto de objetos que são disponibilizados externamente por uma Fachada (RemoteFacade), a qual provê alguns métodos que internamente instanciam

os objetos necessários, realizam as chamadas de métodos e retornam a informação para as possíveis aplicações cliente (ClientApp) que requisitam informações.

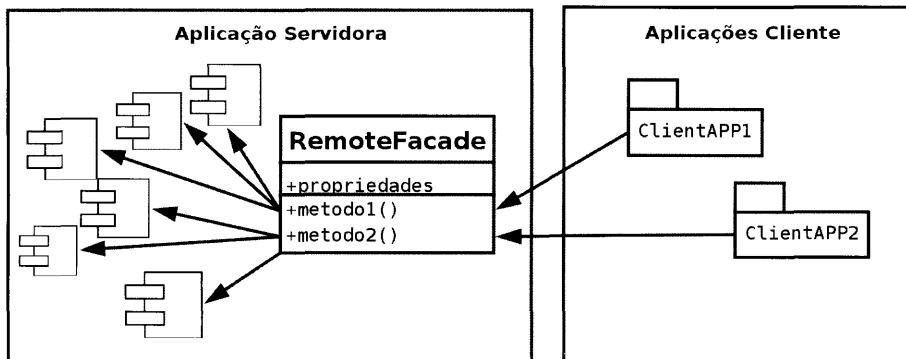


Figura 7.23 – Estrutura de um Remote Facade.

Remote Facades são ideais para se utilizar em um ambiente distribuído como colocamos na figura anterior, em que temos uma aplicação cliente e uma aplicação servidora. Em aplicações de negócios, 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 de dados. Ao colocar esse código na Remote Facade, transfere-se a responsabilidade de abrir a transação, registrar as alterações, finalizar a transação e controlar exceções no lado da aplicação servidora (Remote Facade).

7.3.4.1 Exemplo

Para criarmos um exemplo bem real de implementação do pattern Remote Facade, optamos por criar uma interface cliente para nossa aplicação PHP. O próprio PHP possibilita criarmos aplicações cliente com interfaces gráficas por meio da biblioteca GTK. Nossa exemplo envolverá a criação de um servidor web em PHP e de uma interface cliente em PHP-GTK. Esses dois programas poderão ser executados distantes geograficamente como qualquer outro programa cliente-servidor. Caso você queira saber mais sobre a biblioteca PHP-GTK, acesse o site www.php-gtk.com.br ou o site da Novatec Editora, em www.novatec.com.br, para obter mais informações sobre o livro *PHP-GTK – Criando Aplicações Gráficas com PHP*.

Nossa aplicação consistirá em uma janela gráfica para cadastro de clientes. Este programa em PHP-GTK invocará via Web Services nosso programa servidor, que será um Remote Facade encapsulado sob a classe `ClienteFacade` (Cliente aqui não é de aplicação cliente, mas de cliente comprador). Esta classe oferecerá o método `salvar()` que irá encapsular internamente as chamadas aos objetos e métodos necessários para persistir um cliente na base de dados, tudo por meio da aplicação servidora. A classe `ClienteFacade` é a implementação do pattern Remote Facade, pois disponibiliza

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.

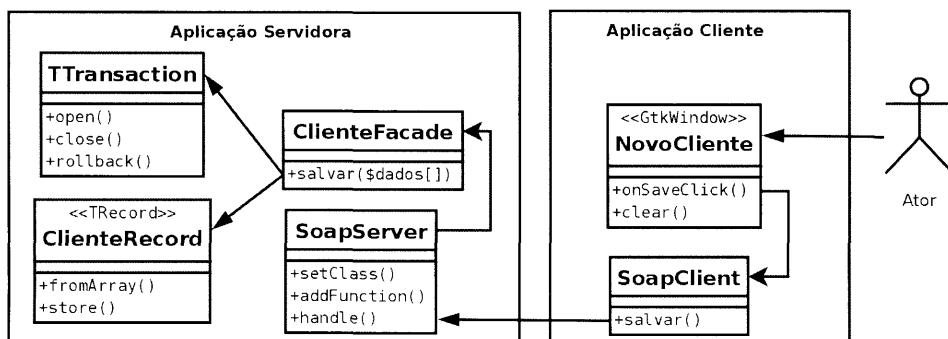


Figura 7.24 – Estrutura da aplicação proposta.

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.

 server.php

```
<?php
/*
 * função __autoload()
 * carrega uma classe quando ela é necessária,
 * ou seja, quando ela é instanciada pela primeira vez.
 */
function __autoload($classe)
{
    $pastas = array('app.ado', 'app.model');
    foreach ($pastas as $pasta)
    {
        if (file_exists("${$pasta}/{$classe}.class.php"))
        {
            include_once "${$pasta}/{$classe}.class.php";
        }
    }
}
/*
 * classe ClienteFacade
 * Remote Facade para cadastro de Clientes
 */
class ClienteFacade
{
    /*
     * método salvar()
     * recebe um array com dados de cliente e armazena no banco de dados
     */
    function salvar($dados)
    {
        try
        {
            // inicia transação com o banco 'pg_livro'
            TTransaction::open('pg_livro');
            // define um arquivo de log
            TTransaction::setLogger(new TLoggerTXT('/tmp/log.txt'));
            // instancia um Active Record para cliente
            $cliente = new ClienteRecord;
            // alimenta o registro com dados do array
            $cliente->fromArray($dados);
            $cliente->store();    // armazena o objeto
            // fecha transação
            TTransaction::close();
        }
        catch (Exception $e)
        {
```

```

    // caso ocorra erros, volta a transação
    TTransaction::rollback();
    // retorna o erro na forma de um objeto SoapFault
    return new SoapFault("Server", $e->getMessage());
}
}

// instancia servidor SOAP
$server = new SoapServer(NULL, array('encoding' => 'ISO-8859-1', 'uri' => 'http://test-uri/'));
// define a classe que irá responder às chamadas remotas
$server->setClass('ClienteFacade');
// prepara-se para receber as chamadas remotas
$server->handle();
?>

```

A nossa aplicação cliente será uma aplicação GTK, como visto na Figura 7.25. Nossa classe será uma janela, portanto a classe `NovoCliente` irá estender a classe `GtkWindow`. No método construtor da aplicação, criamos a janela e declaramos uma série de rótulos de texto (objetos `GtkLabel`) e também campos de entrada de dados (objetos `GtkEntry`). No método construtor também dispomos tais objetos na tela e criamos dois botões de ação: o botão **Salvar**, que executará o método `onSaveClick()` quando clicado, e o botão **Fechar**, que abortará a aplicação quando clicado.

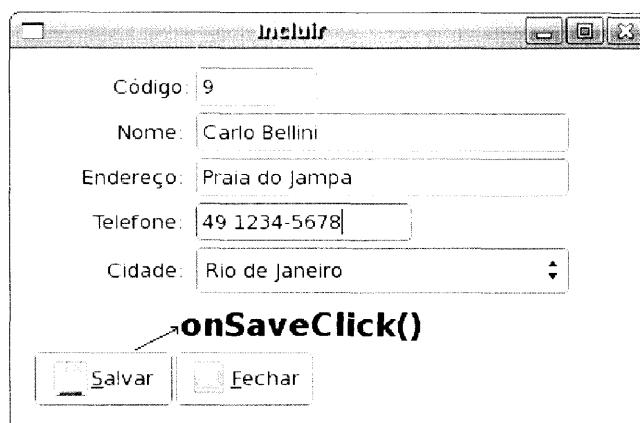


Figura 7.25 – Interface cliente da aplicação.

NovoCliente.php

```

<?php
/*
 * classe NovoCliente
 * formulário de cadastro de Clientes GTK
 */

```

```
class NovoCliente extends GtkWindow
{
    private $window;
    private $campos;
    private $labels;

    /*
     * método construtor
     * instancia a janela e constrói os campos
     */
    public function __construct()
    {
        parent::__construct();
        parent::set_title('Incluir');
        parent::connect_simple('destroy', array('Gtk', 'main_quit'));
        parent::set_default_size(400, 240);
        parent::set_border_width(10);
        parent::set_position(GTK::WIN_POS_CENTER);

        $vbox = new GtkVBox();

        $this->labels['id'] = new GtkLabel('Código:');
        $this->campos['id'] = new GtkEntry;
        $this->campos['id']->set_size_request(80,-1);

        $this->labels['nome'] = new GtkLabel('Nome: ');
        $this->campos['nome'] = new GtkEntry;
        $this->campos['nome']->set_size_request(240,-1);

        $this->labels['endereco'] = new GtkLabel('Endereço: ');
        $this->campos['endereco'] = new GtkEntry;
        $this->campos['endereco']->set_size_request(240,-1);

        $this->labels['telefone'] = new GtkLabel('Telefone: ');
        $this->campos['telefone'] = new GtkEntry;
        $this->campos['telefone']->set_size_request(140,-1);

        $this->labels['id_cidade'] = new GtkLabel('Cidade: ');
        $this->campos['id_cidade'] = GtkComboBox::new_text();
        $this->campos['id_cidade']->set_size_request(240,-1);

        $this->campos['id_cidade']->insert_text(0, 'Porto Alegre');
        $this->campos['id_cidade']->insert_text(1, 'São Paulo');
        $this->campos['id_cidade']->insert_text(2, 'Rio de Janeiro');
        $this->campos['id_cidade']->insert_text(3, 'Belo Horizonte');
```

```

/*
...
adiciona os labels e campos ao formulário...
adiciona alinhamento e posições...
...
*/
$vbox->pack_start(new GtkHSeparator, true, true);

// cria uma caixa de botões
$buttonbox= new GtkHButtonBox;
$buttonbox->set_layout(Gtk::BUTTONBOX_START);

// cria um botão de salvar
$botao = GtkButton::new_from_stock(Gtk::STOCK_SAVE);
// conecta o botão ao método onSaveClick()
$botao->connect_simple('clicked', array($this, 'onSaveClick'));
$buttonbox->pack_start($botao, false, false);

// cria um botão de fechar a aplicação
$botao = GtkButton::new_from_stock(Gtk::STOCK_CLOSE);
$botao->connect_simple('clicked', array('Gtk', 'main_quit'));
$buttonbox->pack_start($botao, false, false);

$vbox->pack_start($buttonbox, false, false);

parent::add($vbox);
// exibe a janela
parent::show_all();
}

```

O método `onSaveClick()` será executado quando o usuário clicar no botão **Salvar**. Sua tarefa será coletar os dados digitados nos objetos de entrada de dados e, em seguida, instanciar um objeto `SoapClient` para abrir conexão com o servidor Web Service, identificando alguns parâmetros como a forma de codificação (`ISO`), a possibilidade de a chamada gerar exceções (`exceptions`) e o local do servidor (`server`). Para realizarmos a chamada do método remoto, simplesmente o executamos sobre o objeto `SoapClient`. Após isso, estamos criando uma mensagem de diálogo de sucesso (`GtkMessageDialog`). Caso alguma exceção seja retornada pelo servidor, o bloco de `try/catch` irá capturar e exibir uma mensagem de erro.

```

/*
 * método onSaveClick()
 * executado quando usuário clica no botão salvar
*/

```

```
public function onSaveClick()
{
    // obtém os valores dos campos
    $dados['id']      = $this->campos['id']->get_text();
    $dados['nome']     = $this->campos['nome']->get_text();
    $dados['endereco'] = $this->campos['endereco']->get_text();
    $dados['telefone'] = $this->campos['telefone']->get_text();
    $dados['id_cidade']= $this->campos['id_cidade']->get_active();

    try
    {
        // instancia cliente SOAP
        $client = new SoapClient(NULL, array('encoding' =>'ISO-8859-1',
                                              'exceptions' => TRUE,
                                              'location'   => "http://127.0.0.1/server.php",
                                              'uri'         => "http://test-uri/"));

        // realiza chamada remota de método
        $retorno = $client->salvar($dados);
        // exibe diálogo de mensagem
        $dialog = new GtkMessageDialog(null, Gtk::DIALOG_MODAL, Gtk::MESSAGE_INFO,
                                       Gtk::BUTTONS_OK, 'Registro inserido com sucesso!');
        $dialog->run();
        $dialog->destroy();
    }
    catch (SoapFault $excecao) // ocorrência de erro
    {
        // exibe diálogo de erro
        $error = new GtkMessageDialog(null, Gtk::DIALOG_MODAL, Gtk::MESSAGE_ERROR,
                                      Gtk::BUTTONS_OK, $excecao->getMessage());
        $error->run();
        $error->destroy();
    }
}

// instancia janela NovoCliente
new NovoCliente;
Gtk::Main();
?>
```

7.4 Conclusão

A orientação a objetos é um tópico muito grande. Seria loucura pensar em escrever um livro que abordasse todos os seus aspectos com completeza. O objetivo deste livro foi o de mostrar a você, leitor, os aspectos de um sistema orientado a objetos focado em aplicações de negócio. Por isso a ênfase em patterns para persistência de objetos em bases de dados e construção de classes visuais como formulários e listagens, as quais vimos em praticamente toda aplicação de negócios. Ao mesmo tempo, procuramos abordar temas introdutórios à linguagem PHP em si (no Capítulo 1) e em relação à orientação a objetos (no Capítulo 2), tornando a leitura deste livro um processo evolutivo que poderia ser acompanhado por qualquer programador, mesmo sem o prévio conhecimento de orientação a objetos.

Símbolos

`__construct()`, 95
`__destruct()`, 96

A

Abstração, 103
 Ações, 361
 Active Record, 257
 Agregação, 118
 array(), 67
 array_in, 75
 array_keys, 72
 array_merge, 72
 array_pad, 71
 array_pop, 69
 array_push, 69
 array_reverse, 71
 array_shift, 70
 array_slice, 74
 array_unshift, 70
 array_values, 73
 Arrays
 associativos, 65
 multidimensionais, 67
 asort, 76
 Associação, 117
 Association Table Mapping, 226
 Autoload, 134

B

BREAK, 42
 By
 reference, 46
 value, 46

C

Cadastro
 de cidades, 498
 de clientes, 511
 de fabricantes, 506
 de produtos, 523
 call_user_func, 84
 Caracteres de escape, 59
 chdir, 56
 CidadeRecord, 499
 CidadesList, 500
 Classes, 90
 abstratas, 103
 finais, 104
 Class Table Inheritance, 230
 ClienteRecord, 512
 ClientesForm, 519
 ClientesList, 514
 closedir, 57

Comentários, 22
 Componentes, 313
 Composição, 122
 Composite Pattern, 177
 Concatenação, 59
 ConcluiVendaForm, 546
 Concrete Table Inheritance, 230
 Constantes, 29, 114
 Construtores, 95
 Contêineres, 331
 CONTINUE, 42
 Controle, 478
 de transações, 206
 Coordenadas absolutas, 401
 copy, 52
 count, 74
 Criando um array, 64
 Critérios de seleção, 172

D

Data
 Mapper, 261
 Transfer Object, 250
 DELETE, 166
 Delimitadores de código, 22
 Design pattern, 169
 Destruidores, 95
 Diálogos
 de mensagem, 369
 de questionamento, 373
 Domain Model Pattern, 236

E

echo, 23
 ELSE, 35
 Emissão de relatórios, 548
 Encapsulamento, 107
 Exception, 150
 explode, 77
 Extensão de arquivos, 21

F

FabricanteList, 508
 FabricanteRecord, 507
 Factory Pattern, 200
 fclose, 50
 feof, 49
 fgets, 49
 file, 51
 file_exists, 54
 file_get_contents, 51
 file_put_contents, 51
 final, 104
 Folhas de estilo, 320

fopen, 48
 FOR, 38
 FOREACH, 42
 Foreign Key Mapping, 224
 Formulários, 377
 Front Controller, 489
 func_get_args, 47
 func_num_args, 47
 Função die(), 145
 Funções, 68
 fwrite, 50

G

Gateways, 245
 get_class, 81
 get_class_methods, 79
 get_class_vars, 79
 get_object_vars, 80
 get_parent_class, 82
 getcwd, 55

H

Herança, 98

I

Identity Field, 223
 IF, 34
 Imagens, 325
 implode, 78
 include, 43
 include_once, 44, 88
 INSERT, 166
 Interfaces, 132
 is_file, 54
 is_subclass_of, 82
 ItemRecord, 536
 Iterações, 66

J

Janelas, 344

K

ksort, 77

L

Lançando erros, 148
 Lazy Initialization, 231
 Listagens, 441
 dinâmica, 443
 estática, 441
 orientadas a objetos, 445

M

Manipulação
de arrays, 64
de dados, 154
de funções, 44
de strings, 58
de XML, 137
Manipulando
coleções, 286
objetos, 265
Mapeamento objeto-relacional, 222
Membros da classe, 114
method_exists, 83
Método
__call, 127
__clone, 133
__get, 126
__set, 124
__toString, 128
POST, 382
toXml(), 129
Métodos
abstratos, 105
estáticos, 116
finais, 106
mkdir, 55
Modelo, 477
de negócios, 235
Model View Controller, 477
MVC, 478
my_livro.ini, 201
mysql_close, 157
mysql_connect, 157
mysql_query, 157

N

NovoCliente, 566

O

Objeto, 93
opendir, 57
Operadores
aritméticos, 30
de atribuição, 30
lógicos, 33
relacionais, 31
Orientação a objetos, 86
overriding, 99

P

Pacotes, 478
Page Controller, 353
Painéis, 340

paises.xml, 138
paises2.xml, 140
paises3.xml, 142
paises4.xml, 143
parent, 99
parse_ini_file, 203
Passagem de parâmetros, 46
PDO, 159
Persistência, 221
pg_close, 157
pg_connect, 157
pg_livro.ini, 201
pg_query, 157
PHP, 20
Polimorfismo, 101
PostgreSQL, 155
print, 23
print_r, 23
Private, 109
Processo de venda, 534
ProdutoRecord, 524
ProdutosList, 525
Programa
abstrato.php, 105
action.php, 364
active_record.php, 258
agregacao.php, 120
agregacao2.php, 121
associacao.php, 118
autoload.php, 135
banco.sql, 275
business.php, 309
Cachorro.class.php, 124
call.php, 128
Cesta.class.php, 119
CidadeRecord.class.php, 499
CidadesList.class.php, 500
classe_abstrata.php, 104
classe_final.php, 104
cliente.php, 560
ClienteRecord.class.php, 512
clientes.php, 491
ClientesForm.class.php, 519
ClientesList.class.php, 514
clone.php, 133
collection_count.php, 297
collection_delete.php, 300
collection_get.php, 292
collection_update.php, 295
composicao.php, 123
ConcluiVendaForm.class.
php, 546
connection.php, 204
constantes.php, 114
construtores.php, 97
Conta.class.php, 92, 97, 103, 105
ContaCorrente.class.php,
100, 106
ContaPoupanca.class.php,
100, 104, 106
Contato.class.php, 122
criteria.php, 182
data_mapper.php, 262
data_transfer.php, 250
delete.php, 194
dinamico.php, 135
domain_model.php, 237
domain_model2.php, 240
encapsulamento.php, 302
erro_die.php, 146
erro_exception.php, 151
erro_flag.php, 147
erro_subexception.php, 152
erro_trigger.php, 148
Estagiario.class.php, 111
estilos.php, 322
exemplo.wsdl, 558
exemplo1.php, 138
exemplo2.php, 139
exemplo3.php, 139
exemplo4.php, 140
exemplo5.php, 141
exemplo6.php, 142
exemplo7.php, 144
exemplo8.php, 145
FabricanteList.class.php, 508
FabricanteRecord.class.php, 507
filter.php, 176
form.html, 381
form_gravar.php, 383
form1.php, 422
form2.php, 426
form3.php, 431
form4.php, 437
formA.php, 399
formB.php, 403
Fornecedor.class.php, 117, 123
Funcionario.class.php,
110, 112
get.php, 126
image.php, 328
index.php, 490, 492, 493, 495
insert.php, 188
interface.php, 132
ItemRecord.class.php, 536
lazy.php, 306
lazy_init.php, 233
list.html, 442
list.php, 444
list1.php, 458
list2.php, 462
list3.php, 466

- list4.php, 469
list5.php, 474
log.php, 218
menu.html, 491
message_error.php, 372
message_info.php, 371
metesta.php, 116
metodo_final.php, 107
model_clone.php, 283
model_delete.php, 285
model_get.php, 279
model_novo.php, 276
model_update.php, 281
mysql_insere.php, 157
mysql_lista.php, 158
NovoCliente.php, 566
objarray.php, 136
objeto.php, 89, 90
objetos.php, 94
page.php, 354
page1.php, 358
page2.php, 359
page3.php, 360
page4.php, 366
painel.html, 341
painel.php, 344
paragraph.php, 330
pdo_insere.php, 160
pdo_insere_my.php, 163
pdo_lista.php, 161
pdo_lista_obj.php, 162
Pessoa.class.php, 91, 96
pgsql_insere.php, 156
pgsql_lista.php, 156
poli.php, 102
private.php, 109, 111
Produto.class.php, 88, 89,
 126, 127
ProdutoRecord.class.php, 524
ProdutosForm.class.php, 530
ProdutosList.class.php, 525
propesta.php, 115
protected.php, 112, 113
public.php, 113
question.php, 375
RelatorioForm.class.php, 550
row_gateway.php, 254
select.php, 197
select2.php, 198
server.php, 565
servidor.php, 559
session.php, 489
set.php, 125
tabelas_html.php, 332
tabelas_oo.php, 336
tabelas_oo2.php, 339
table_gateway.php, 247
table_module.php, 243
TAction.php, 362
tags.php, 317
tagsstyle.php, 324
TButton.class.php, 420
TCheckButton.class.php, 413
TCheckGroup.class.php, 415
TCombo.class.php, 411
TConnection.class.php, 202
TCriteria.class.php, 179
TDataGrid.class.php, 446
TDataGridAction.class.php, 456
TDataGridColumn.class,
 php, 453
TElement.class.php, 314
template.html, 495
TEntry.class.php, 397
TExpression.class.php, 173
TField.class.php, 392
TFile.class.php, 406
TFilter.class.php, 173
TForm.class.php, 386
THidden.class.php, 407
TImage.class.php, 325, 327
 TLabel.class.php, 395
TMessage.class.php, 370
tostring.php, 128
toxml.php, 129
TPage.class.php, 356
TPanel.class.php, 342
TParagraph.class.php, 329
TPassword.class.php, 405
TQuestion.class.php, 373
TRadioButton.class.php, 417
TRadioGroup.class.php, 418
trans.php, 484
transaction.php, 210
TRecord.class.php, 266
TRepository.php, 288
TSession.class.php, 487
TSqlDelete.class.php, 193
TSqlInsert.class.php, 186
TSqlInstruction.class.php, 184
TSqlSelect.class.php, 195
TSqlUpdate.class.php, 189
TStyle.php, 320
TTable.class.php, 334
TTableCell.class.php, 336
TTableRow.class.php, 335
TText.class.php, 409
TTransaction.class.php, 216
TTranslation.class.php, 482
TWindow.class.php, 348
update.php, 191
VendaRecord.class.php, 535
VendaRecord.class.php (con-
tinuação...), 549
VendasForm.class.php, 538
window.html, 346
window.php, 352
XMLBase.class.php, 131
XMLBase.php, 131
Programação estruturada, 86
Propriedades estáticas, 115
Protected, 111
Public, 113
- Q**
- Query Object, 170
- R**
- readdir, 57
readme.txt, 116
Recursão, 48
Registro de log, 212
Registry Pattern, 486
RelatorioForm, 550
Remote Facade, 561
rename, 53
Repository, 287
require, 43
require_once, 44
rmdir, 56
rollback, 208
Row Data Gateway, 253
rsort, 76
- S**
- Seções, 485
SELECT, 165
set_error_handler, 148
SimpleXML, 137
Single Table Inheritance, 228
Singleton Pattern, 481
Sintaxe SQL, 165
SOAP, 555
sort, 75
str_repeat, 62
str_replace, 63
Strategy Pattern, 213
strlen, 62
strpad, 61
strpos, 63
strtolower, 60
strtoupper, 60
substr, 61
SWITCH, 39

T

Tabelas, 331
Table Data Gateway, 246
Table Module, 242
TAction, 362
TApplication, 492
TButton, 419
TCheckButton, 413
TCheckGroup, 414
TCombo, 410
TConnection, 200
TCriteria, 178
TDataGrid, 445
TDataGridColumn, 452
TElement, 314
Template View, 493
TEntry, 397
TExpression, 172
Textos, 328
TField, 391
TFile, 406
TFilter, 173
TForm, 386
THidden, 407
TImage, 325
Tipo
 array, 28
 booleano, 26
 callback, 29
 misto, 29
 NULL, 29
 numérico, 27
 objeto, 28
 recurso, 28
 string, 27
 TLabel, 394
 TLogger, 213
 TLoggerHTML, 215
 TLoggerTXT, 216
 TLoggerXML, 214
 TMessage, 370
 TPage, 356
 TPanel, 342
 TParagraph, 329
 TPassword, 404
 TQuestion, 373
 TRadioButton, 416
 TRadioGroup, 417
 Tratamento
 de erros, 145
 de exceções, 150
 TRecord, 266
 TRepository, 288
 trigger_error, 148
 TSession, 487
 TSqDelete, 192
 TSqInsert, 186
 TSqInstruction, 184
 TSqSelect, 195
 TSqUpdate, 189
 TStyle, 320
 TTable, 334
 TTableCell, 336
 TTableRow, 335
 TText, 408
 TTransaction, 208
 TTranslation, 482
 TWindow, 348

U

unlink, 53
UPDATE, 166

V

var_dump, 23
Variáveis, 24
 estáticas, 45
 globais, 45
VendaRecord, 535
VendasForm, 538
Visual de tabela, 398
Visualização, 477

W

Web Services, 555
WHILE, 37
Widgets, 313
WSDL, 557

Z

Zend, 21

NOVATEC EDITORA – A CASA DO PHP

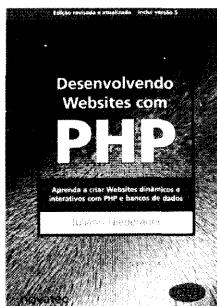
LIVROS



PHP-GTK – 2^a Edição Criando Aplicações Gráficas com PHP

Autor: Pablo Dall'Oglio
445 pág.
ISBN: 978-85-7522-110-5

O PHP tem crescido muito nos últimos anos e se tornou uma linguagem de propósitos gerais. Este livro é prova disto. Por intermédio do PHP-GTK, você pode utilizar o PHP para desenvolver aplicações com visual atraente e nativo para várias plataformas, como Linux, Windows e Mac, utilizando a mesma linguagem que utiliza para criar aplicações para a Web.



Desenvolvendo Web sites com PHP

Autor: Juliano Niederauer
272 pág.
ISBN: 85-7522-050-0

O livro abrange desde noções básicas de programação até a criação e manutenção de bancos de dados, mostrando como são feitas inclusões, exclusões, alterações e consultas a tabelas de uma base de dados. O autor apresenta diversos exemplos de programas para facilitar a compreensão da linguagem.



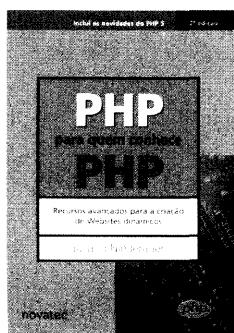
Web Interativa com Ajax e PHP

Juliano Niederauer

ISBN: 978-85-7522-126-6

288 págs.

Ajax (acrônimo de Asynchronous JavaScript and XML) é uma técnica de desenvolvimento web que combina tecnologias conhecidas, como JavaScript, XML, entre outras, para tornar as páginas web mais dinâmicas e interativas.



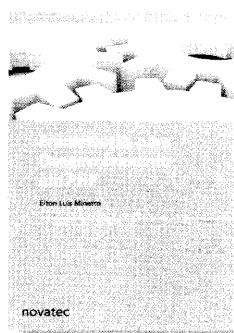
PHP para quem conhece PHP

Juliano Niederauer

ISBN: 85-7522-044-6

480 págs.

Aborda diversos assuntos úteis ao desenvolvedor, como cookies e sessões, upload de arquivos, geração de imagens e gráficos, arquivos PDF, templates, abstração de bancos de dados, entre outros.



Frameworks para Desenvolvimento em PHP

Elton Luís Minetto

ISBN: 978-85-7522-124-2

Páginas: 192

Os frameworks facilitam o desenvolvimento de software, permitindo que os programadores se ocupem mais com os requerimentos do software do que com os detalhes tediosos, de baixo nível do sistema.



GUIA DE CONSULTA RÁPIDA

Juliano Niederauer

novatec



GUIA DE CONSULTA RÁPIDA

Juliano Niederauer

novatec



GUIA DE CONSULTA RÁPIDA

Juliano Niederauer

novatec

O PHP é uma das linguagens mais utilizadas no mundo. Sua popularidade se deve à facilidade em criar aplicações dinâmicas com suporte à maioria dos bancos de dados existentes e ao conjunto de funções que, por meio de uma estrutura flexível de programação, permitem desde a criação de simples portais até complexas aplicações de negócios.

O uso da orientação a objetos juntamente com o emprego de boas práticas de programação nos possibilita manter um ritmo sustentável no desenvolvimento de aplicações. O foco deste livro é demonstrar como se dá a construção de uma aplicação totalmente orientada a objetos. Para isso, implementaremos alguns padrões de projeto (design patterns) e algumas técnicas de mapeamento objeto-relacional, além de criarmos vários componentes para que você possa criar complexas aplicações de negócios com PHP.

Principais tópicos abordados no livro:

- Introdução ao PHP, arrays, strings e arquivos
- Orientação a objetos, conceitos e implementações
- XML, Web Services, tratamento de exceções
- Técnicas de mapeamento objeto-relacional
- Criação de classes para apresentação de HTML
- Criação de classes para formulários e listagens
- Criação de classes para manipulação de SQL
- Criação de uma aplicação orientada a objetos



Pablo Dall'Oglio é graduado em Análise de Sistemas pela Unisinos e autor de softwares reconhecidos como o Agata Report e o Tulip. Possui grande experiência no desenvolvimento de sistemas e está constantemente envolvido com análise, projeto e implementação de softwares orientados a objetos, UML e design patterns. Criador da comunidade brasileira de PHP GTK (www.php-gtk.com.br), é diretor de tecnologia da Adianti Solutions (www.adianti.com.br).

