

TWÓJ PRZEWODNIK PO OBIEKTOWYM PHP!

Apress®

PHP

Obiekty, wzorce, narzędzia

WYDANIE IV

Matt Zandstra

Helion 

Tytuł oryginału: PHP Objects, Patterns, and Practice, Fourth Edition

Tłumaczenie: Przemysław Szeremiota

ISBN: 978-83-246-9178-4

Original edition copyright © 2013 by Matt Zandstra
All rights reserved.

Polish edition copyright © 2014 by HELION SA.
All rights reserved.

All rights reserved. No part of this book may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording or by any information storage retrieval system, without permission from the Publisher.

Wszelkie prawa zastrzeżone. Nieautoryzowane rozpowszechnianie całości lub fragmentu niniejszej publikacji w jakiegokolwiek postaci jest zabronione. Wykonywanie kopii metodą kserograficzną, fotograficzną, a także kopiowanie książki na nośniku filmowym, magnetycznym lub innym powoduje naruszenie praw autorskich niniejszej publikacji.

Wszystkie znaki występujące w tekście są zastrzeżonymi znakami firmowymi bądź towarowymi ich właścicieli.

Autor oraz Wydawnictwo HELION dołożyli wszelkich starań, by zawarte w tej książce informacje były kompletne i rzetelne. Nie bierze jednak żadnej odpowiedzialności ani za ich wykorzystanie, ani za związane z tym ewentualne naruszenie praw patentowych lub autorskich. Wydawnictwo HELION nie ponosi również żadnej odpowiedzialności za ewentualne szkody wynikłe z wykorzystania informacji zawartych w książce.

Wydawnictwo HELION
ul. Kościuszki 1c, 44-100 GLIWICE
tel. 32 231 22 19, 32 230 98 63
e-mail: helion@helion.pl
WWW: <http://helion.pl> (księgarnia internetowa, katalog książek)

Pliki z przykładami omawianymi w książce można znaleźć pod adresem:
<ftp://ftp.helion.pl/przyklady/phpob4.zip>

Drogi Czytelniku!
Jeżeli chcesz ocenić tę książkę, zajrzyj pod adres
<http://helion.pl/user/opinie/phpob4>
Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

Printed in Poland.

- [Kup książkę](#)
- [Poleć książkę](#)
- [Oceń książkę](#)

- [Księgarnia internetowa](#)
- [Lubię to! » Nasza społeczność](#)

Spis treści

	O autorze	13
	O recenzencie technicznym	15
	Podziękowania	17
	Wprowadzenie	19
Rozdział 1.	PHP — projektowanie i zarządzanie	21
	Problem	21
	PHP a inne języki programowania	22
	O książce	24
	Obiekty	24
	Wzorce	24
	Narzędzia	25
	Nowości w czwartym wydaniu	26
	Podsumowanie	26
Rozdział 2.	Obiekty	27
	Nieoczekiwany sukces obiektów w PHP	27
	PHP/FI — u zarania języka	27
	PHP3 — składniowy lukier	27
	Cicha rewolucja — PHP4	28
	PHP5 — nieuchronne zmiany	29
	Debata obiektowa — za czy przeciw?	30
	Podsumowanie	30
Rozdział 3.	Obiektowy elementarz	31
	Klasy i obiekty	31
	Pierwsza klasa	31
	Pierwszy obiekt (lub dwa)	32
	Definiowanie składowych klasy	33
	Metody	35
	Metoda konstrukcji obiektu	36

Typy argumentów metod	37
Typy elementarne	38
Typy obiektowe	40
Dziedziczenie	42
Problemy związane z dziedziczeniem	42
Stosowanie dziedziczenia	46
Zarządzanie dostępem do klasy — słowa public, private i protected	50
Podsumowanie	54
Rozdział 4. Zaawansowana obsługa obiektów	55
Metody i składowe statyczne	55
Składowe stałe	58
Klasy abstrakcyjne	59
Interfejsy	61
Cechy typowe	62
Zadanie dla cech typowych	62
Definiowanie i stosowanie cechy typowej	63
Stosowanie wielu cech typowych	64
Łączenie cech z interfejsami	64
Unikanie kolizji nazw metod za pomocą słowa insteadof	65
Aliasy metod cech typowych	67
Cechy typowe z metodami statycznymi	68
Dostęp do składowych klasy włączającej	68
Definiowanie metody abstrakcyjnej cechy typowej	69
Zmiana dostępności metod cech typowych	70
Późne wiązanie statyczne: słowo static	71
Obsługa błędów	73
Wyjątki	75
Klasy i metody finalne	80
Przechwytywanie chybionych wywołań	81
Definiowanie destruktorów	86
Wykonywanie kopii obiektów	87
Reprezentacja obiektu w ciągach znaków	90
Wywołania zwrotne, funkcje anonimowe i domknięcia	91
Podsumowanie	94
Rozdział 5. Narzędzia obiektowe	95
PHP a pakiety	95
Pakiety i przestrzenie nazw w PHP	95
Automatyczne wczytywanie kodu	103
Klasy i funkcje pomocnicze	105
Szukanie klasy	106
Badanie obiektów i klas	107
Pozyskiwanie ciągu pełnej nazwy klasy	108
Badanie metod	108
Badanie składowych	110
Badanie relacji dziedziczenia	110
Badanie wywołań metod	110

Interfejs retrospekcji — Reflection API	112
Zaczynamy	112
Pora zakasać rękawy	112
Badanie klasy	114
Badanie metod	116
Badanie argumentów metod	117
Korzystanie z retrospekcji	118
Podsumowanie	121
Rozdział 6. Obiekty a projektowanie obiektowe	123
Czym jest projektowanie?	123
Programowanie obiektowe i proceduralne	124
Odpowiedzialność	127
Spójność	127
Sprzęganie	127
Ortogonalność	128
Zasięg klas	128
Polimorfizm	129
Hermetyzacja	131
Nieważne jak	132
Cztery drogowaskazy	132
Zwielokrotnianie kodu	133
Przemądrzałe klasy	133
Złota rączka	133
Za dużo warunków	133
Język UML	133
Diagramy klas	134
Diagramy sekwencji	139
Podsumowanie	141
Rozdział 7. Czym są wzorce projektowe? Do czego się przydają?	143
Czym są wzorce projektowe?	143
Wzorzec projektowy	145
Nazwa	145
Problem	146
Rozwiązanie	146
Konsekwencje	146
Format wzorca według Bandy Czworga	146
Po co nam wzorce projektowe?	147
Wzorzec projektowy definiuje problem	147
Wzorzec projektowy definiuje rozwiązanie	147
Wzorce projektowe są niezależne od języka programowania	147
Wzorce definiują słownictwo	148
Wzorce są wypróbowane	148
Wzorce mają współpracować	149
Wzorce promują zasady projektowe	149
Wzorce są stosowane w popularnych frameworkach	149
Wzorce projektowe a PHP	149
Podsumowanie	150

Rozdział 8. Wybrane zasady wzorców	151
Olśnienie wzorcami	151
Kompozycja i dziedziczenie	152
Problem	152
Zastosowanie kompozycji	155
Rozprzęganie	157
Problem	157
Osłabianie sprzężenia	158
Kod ma używać interfejsów, nie implementacji	160
Zmienne koncepcje	161
Nadmiar wzorców	161
Wzorce	161
Wzorce generowania obiektów	162
Wzorce organizacji obiektów i klas	162
Wzorce zadaniowe	162
Wzorce korporacyjne	162
Wzorce baz danych	162
Podsumowanie	162
Rozdział 9. Generowanie obiektów	163
Generowanie obiektów — problemy i rozwiązania	163
Wzorzec Singleton	167
Problem	167
Implementacja	168
Konsekwencje	169
Wzorzec Factory Method	170
Problem	170
Implementacja	172
Konsekwencje	174
Wzorzec Abstract Factory	174
Problem	174
Implementacja	175
Konsekwencje	177
Prototyp	178
Problem	178
Implementacja	179
Ależ to oszustwo!	181
Podsumowanie	182
Rozdział 10. Wzorce elastycznego programowania obiektowego	183
Strukturalizacja klas pod kątem elastyczności obiektów	183
Wzorzec Composite	183
Problem	184
Implementacja	186
Konsekwencje	189
Composite — podsumowanie	191
Wzorzec Decorator	192
Problem	192
Implementacja	194
Konsekwencje	197

Wzorzec Facade	197
Problem	197
Implementacja	199
Konsekwencje	199
Podsumowanie	200
Rozdział 11. Reprezentacja i realizacja zadań	201
Wzorzec Interpreter	201
Problem	201
Implementacja	202
Ciemne strony wzorca Interpreter	209
Wzorzec Strategy	209
Problem	209
Implementacja	211
Wzorzec Observer	214
Implementacja	215
Wzorzec Visitor	220
Problem	220
Implementacja	221
Wady wzorca Visitor	225
Wzorzec Command	226
Problem	226
Implementacja	226
Podsumowanie	230
Rozdział 12. Wzorce korporacyjne	231
Przegląd architektury	231
Wzorce	232
Aplikacje i warstwy	232
Małe oszustwo na samym początku	235
Wzorzec Registry	235
Implementacja	236
Warstwa prezentacji	244
Wzorzec Front Controller	244
Wzorzec Application Controller	253
Wzorzec Page Controller	264
Wzorce Template View i View Helper	268
Warstwa logiki biznesowej	270
Wzorzec Transaction Script	270
Wzorzec Domain Model	274
Podsumowanie	277
Rozdział 13. Wzorce bazodanowe	279
Warstwa danych	279
Wzorzec Data Mapper	280
Problem	280
Implementacja	280
Wzorzec Identity Map	293
Problem	293
Implementacja	294
Konsekwencje	296

Wzorzec Unit of Work	297
Problem	297
Implementacja	297
Konsekwencje	301
Wzorzec Lazy Load	301
Problem	301
Implementacja	302
Konsekwencje	303
Wzorzec Domain Object Factory	303
Problem	303
Implementacja	304
Konsekwencje	305
Wzorzec Identity Object	306
Problem	306
Implementacja	307
Konsekwencje	311
Wzorce Selection Factory i Update Factory	312
Problem	312
Implementacja	312
Konsekwencje	315
Co zostało z wzorca Data Mapper?	316
Podsumowanie	318
Rozdział 14. Dobre (i złe) praktyki	319
Nie tylko kod	319
Pukanie do otwartych drzwi	320
Jak to zgrać?	321
Uskrzydlenie kodu	322
Dokumentacja	323
Testowanie	324
Ciągła integracja	325
Podsumowanie	325
Rozdział 15. PEAR i Pyrus	327
Czym jest PEAR?	327
Pyrus	328
Instalowanie pakietu	329
Kanały PEAR	331
Korzystanie z pakietu z PEAR	333
Obsługa błędów w pakietach PEAR	334
Tworzenie własnych pakietów PEAR	337
Plik package.xml	337
Składniki pakietu	338
Element contents	339
Zależności	342
Dookreślanie instalacji — phprelease	343
Przygotowanie pakietu do dystrybucji	344
Konfigurowanie własnego kanału PEAR	345
Podsumowanie	348

Rozdział 16. Generowanie dokumentacji — phpDocumentor	349
Po co nam dokumentacja?	349
Instalacja	350
Generowanie dokumentacji	350
Komentarze DocBlock	352
Dokumentowanie klas	354
Dokumentowanie plików	354
Dokumentowanie składowych	355
Dokumentowanie metod	357
Namespace support	357
Tworzenie odnośników w dokumentacji	359
Podsumowanie	361
Rozdział 17. Zarządzanie wersjami projektu z systemem Git	363
Po co mi kontrola wersji?	363
Skąd wziąć klienta Git?	364
Konfigurowanie serwera Git	365
Tworzenie repozytorium zdalnego	365
Rozpoczynamy projekt	367
Klonowanie repozytorium	369
Wprowadzanie i zatwierdzanie zmian	370
Dodawanie i usuwanie plików i katalogów	373
Dodawanie pliku	373
Usuwanie pliku	374
Dodawanie katalogu	374
Usuwanie katalogów	374
Etykietowanie wersji	375
Rozgałęzianie projektu	375
Podsumowanie	379
Rozdział 18. Testy jednostkowe z PHPUnit	381
Testy funkcjonalne i testy jednostkowe	381
Testowanie ręczne	382
PHPUnit	384
Tworzenie przypadku testowego	384
Metody asercji	385
Testowanie wyjątków	386
Uruchamianie zestawów testów	387
Ograniczenia	388
Atrapy i imitacje	389
Dobry test to obłany test	392
Testy dla aplikacji WWW	394
Przygotowanie aplikacji WWW do testów	395
Proste testy aplikacji WWW	396
Selenium	398
Słowo ostrzeżenia	402
Podsumowanie	404

Rozdział 19. Automatyzacja instalacji z Phing	405
Czym jest Phing?	406
Pobieranie i instalacja pakietu Phing	406
Montowanie dokumentu kompilacji	407
Różnicowanie zadań kompilacji	408
Właściwości	410
Typy	416
Operacje	420
Podsumowanie	424
Rozdział 20. Ciągła integracja kodu	425
Czym jest ciągła integracja?	425
Przygotowanie projektu do ciągłej integracji	427
Jenkins	436
Instalowanie Jenkinsa	436
Instalowanie rozszerzeń Jenkinsa	438
Konfigurowanie klucza publicznego serwera Git	439
Instalowanie projektu	439
Pierwsza kompilacja	441
Konfigurowanie raportów	441
Automatyzacja kompilacji	444
Podsumowanie	446
Rozdział 21. Obiekty, wzorce, narzędzia	447
Obiekty	447
Wybór	448
Hermetyzacja i delegowanie	448
Osłabianie sprzężenia	448
Zdatność do wielokrotnego stosowania kodu	449
Estetyka	449
Wzorce	450
Co dają nam wzorce?	450
Wzorce a zasady projektowe	451
Narzędzia	452
Testowanie	453
Dokumentacja	453
Zarządzanie wersjami	453
Automatyczna kompilacja (instalacja)	454
System integracji ciągłej	454
Co pominęliśmy?	454
Podsumowanie	455
Dodatek A Bibliografia	457
Książki	457
Publikacje	458
Witryny WWW	458
Dodatek B Prosty analizator leksykalny	461
Skaner	461
Analizator leksykalny	468
Skorowidz	481

ROZDZIAŁ 5



Narzędzia obiektowe

Poprzednie rozdziały zaznajamiały Czytelnika z programowaniem obiektowym, przybliżając mu podstawowe konstrukcje obiektowe, takie jak klasy i metody. Udogodnienia obiektowe nie kończą się na tych konstrukcjach i mechanizmach — sam język udostępnia też mechanizmy pomocnicze, ułatwiające pracę z obiektami.

Niniejszy rozdział poświęcony będzie prezentacji niektórych z tych narzędzi oraz technikom wykorzystywanym do organizowania, testowania i korzystania z klas i obiektów.

W rozdziale omawiam:

- *Pakiety* — czyli organizowanie kodu w kategorii logiczne.
- *Przestrzenie nazw* — od wersji 5.3 można osadzać elementy kodu w odrębnych przestrzeniach nazw.
- *Włączanie kodu* — z naciskiem na ustanowienie centralnie dostępnej lokalizacji kodu bibliotecznego.
- *Funkcje pomocnicze względem klas i obiektów* — służące do testowania obiektów, klas, składowych i metod.
- *Interfejs Reflection API* — bezprecedensowy zestaw wbudowanych klas pozwalających na retrospekcję: realizację dynamicznych odwołań do informacji o klasach.

PHP a pakiety

Pakiet to zbiór powiązanych ze sobą klas. Pakiety służą do wyodrębniania i rozdzielania poszczególnych części systemu. W niektórych językach programowania obsługa pakietów (modułów) jest sformalizowana — jak w Javie, gdzie pakiety dysponują własnymi przestrzeniami nazw. W PHP koncepcja pakietu jest cokolwiek obca, ale od wersji 5.3 wprowadzono przestrzenie nazw, o których napiszę więcej w następnym podrozdziale.

Skoro i tak przez jakiś czas będziemy musieli pracować również ze starym kodem, nie może tu zabraknąć klasycznego sposobu organizowania klasy w struktury pakietopodobne.

Pakiety i przestrzenie nazw w PHP

PHP nie posiada mechanizmów obsługi pakietów jako takich, ale programiści od zawsze radzili sobie z kategoryzacją kodu poprzez odrębne konwencje nazewnictwa i separację kodu w systemie plików. Niebawem zajmiemy się zalecanymi technikami organizowania kodu na bazie katalogów i plików, na początek jednak weźmiemy na warsztat konwencje nazewnictwa oraz nowy mechanizm przestrzeni nazw.

Aż do powstania wersji 5.3 programiści byli zmuszeni do dobierania nazw plików w kontekście globalnym. Innymi słowy, jeśli klasa nosiła miano ShoppingBasket, była pod tą nazwą dostępna w całym systemie. Prowadziło to do dwóch problemów. Przede wszystkim wprowadzało niemałe ryzyko kolizji nazw. Niby jest

to mało prawdopodobne, bo wystarczy zapamiętać wszystkie nazwy klas, prawda? Kłopot w tym, że każdy z programistów używa mnóstwa kodu bibliotecznego, zewnętrznego. To oczywiście pożądane, ale w kontekście kolizji nazw bardzo ryzykowne. Co jeśli nasz projekt robi tak:

```
// plik my.php
require_once "useful/Outputter1.php"
class Outputter {
    // wypisywanie danych
}
```

A plik włączany do projektu robi tak:

```
// plik useful/Outputter1.php
class Outputter {
    // ...
}
```

Chyba już wiemy, prawda? Oto co się stanie:

PHP Fatal error: Cannot redeclare class Outputter in ...Outputter1.php on line 2

Oczywiście, istniało konwencjonalne obejście tego problemu. Należało poprzedzać nazwy klas nazwami pakietów, co gwarantowało (w pewnym stopniu) unikatowość nazw klas:

```
// plik my.php
require_once "useful/Outputter2.php";
class my_Outputter {
    // wypisywanie danych
}
```

```
// plik useful/Outputter2.php
class useful_Outputter {
    // ...
}
```

Sęk w tym, że w miarę rozbudowywania projektów nazwy klas wydłużały się niemiłosiernie. Nie jest to może bardzo problematyczne, ale zmniejsza czytelność kodu i utrudnia zapamiętanie nazw klas przez programistów, a także przyczynia się do utraty wielu roboczogodzin potrzebnych na poprawianie pomyłek w coraz to dłuższych nazwach.

Jeszcze przez lata będziemy skazani na tę konwencję, bo każdy z nas korzysta z jakichś starszych bibliotek. Z tego względu do zagadnienia klasycznego sposobu zarządzania pakietami wrócimy jeszcze w dalszej części rozdziału.

Ratunek — przestrzenie nazw

W PHP 5.3 pojawiła się obsługa przestrzeni nazw. Zasadniczo przestrzeń nazw to pojemnik, w którym można umieszczać klasy, funkcje i zmienne. W obrębie przestrzeni nazw można się do tych elementów odwoływać bez kwalifikowania odwołań. Z zewnątrz należy albo zaimportować przestrzeń nazw, albo odwoływać się do jej elementów za pomocą nazw kwalifikowanych.

Skomplikowane? Przykład powinien rozjaśnić problem. Oto przykład kolidujących klas przepisany na przestrzenie nazw:

```
namespace my;
require_once "useful/Outputter3.php";

class Outputter {
    // wypisywanie danych
}
```

```
// plik useful/Outputter3.php
namespace useful;
class Outputter {
    //
}
```

Zauważmy słowo kluczowe `namespace`. Łatwo się domyślić, że ustanawia ono nową przestrzeń nazw. Użytkownicy tego mechanizmu powinni pamiętać, że deklaracja przestrzeni nazw musi być pierwszą instrukcją pliku. Powyżej utworzyliśmy dwie przestrzenie nazw: `my` oraz `useful`. Typowo jednak przestrzenie nazw tworzą głębszą hierarchię. Na samym szczycie definiuje się zazwyczaj przestrzeń z nazwą projektu albo organizacji. Następnie kwalifikuje się tę nazwę nazwą pakietu — PHP pozwala na deklarowanie zagnieżdżonych przestrzeni nazw. Poziomy w hierarchii przestrzeni nazw oddziela się znakami lewego ukośnika.

```
namespace com\getinstance\util;

class Debug {
    static function helloWorld() {
        print "hello from Debug\n";
    }
}
```

Gdybyśmy udostępniali repozytorium z kodem, w naturalny sposób moglibyśmy użyć członów nazwy domeny jako początkowych członów przestrzeni nazw. Sztuczkę tę stosują programiści Javy w nazwach pakietów: odwracają nazwy domen organizacji czy projektów, od członu najbardziej ogólnego do członu najbardziej szczegółowego. A po zidentyfikowaniu repozytorium można zacząć definiować pojedyncze pakiety — w tym przypadku pakiet `util`.

Jak wywołać metodę klasy z takiego pakietu? Zależy, skąd ta metoda ma być wywołana. Jeśli wywołanie odbywa się w obrębie przestrzeni nazw, w której metoda jest zadeklarowana, można ją wywołać wprost:

```
Debug::helloWorld();
```

Takie wywołanie nazwiemy niekwalifikowanym. W przestrzeni nazw `com\getinstance\util` nazwy klas i metod są dostępne bez żadnych członów poprzedzających. Ale spoza przestrzeni nazw należy używać nazwy klasy (metody) kwalifikowanej nazwą przestrzeni nazw:

```
com\getinstance\util\Debug::helloWorld();
```

Jaki będzie więc efekt wykonania poniższego kodu?

```
namespace main;
com\getinstance\util\Debug::helloWorld();
```

Pytanie było podchwytliwe. Oczywiście pojawi się błąd:

```
PHP Fatal error: Class 'main\com\getinstance\util\Debug' not found in ...
```

A to dlatego, że użyliśmy względnej przestrzeni nazw. PHP przy rozwiązywaniu nazw szuka przestrzeni nazw `com\getinstance\util` w obrębie przestrzeni nazw `main` i rzecz jasna — nie znajduje jej. Tak samo, jak można stosować bezwzględne ścieżki plików i URL-e, tak samo można konstruować bezwzględne nazwy przestrzeni nazw. Błąd poprzedniego programu można więc naprawić tak:

```
namespace main;
\com\getinstance\util\Debug::helloWorld();
```

Znak lewego ukośnika na początku identyfikatora przestrzeni nazw mówi, że poszukiwanie przestrzeni nazw należy zacząć od samego szczytu hierarchii, a nie od bieżącej przestrzeni nazw.

Ale czy przestrzenie nazw nie miały przypadkiem oszczędzić programistom długich nazw? Deklaracja klasy `Debug` jest co prawda krótsza, ale jej wywołania wcale się nie skróciły — są równie rozwlekłe jak w klasycznym modelu „pakietów” bez przestrzeni nazw. Do wyeliminowania tej rozwlekłości przewidziano osobne słowo kluczowe języka PHP: `use`. Pozwala ono na aliasowanie nazw innych przestrzeni nazw w bieżącej przestrzeni nazw. Oto przykład:

```
namespace main;
use com\getinstance\util;
util\Debug::helloWorld();
```

Przestrzeń nazw `com\getinstance\util` została tu skrócona do krótkiej nazwy `util`. Zauważmy, że nie rozpoczęto jej od znaku ukośnika: argument dla słowa kluczowego `use` jest rozpatrywany w globalnej, a nie w bieżącej przestrzeni nazw. Dalej, jeśli w ogóle chcemy się pozbyć kwalifikacji nazw, możemy zaimportować klasę `Debug` do bieżącej przestrzeni nazw:

```
namespace main;
use com\getinstance\util\Debug;
Debug::helloWorld();
```

A co się stanie, jeśli w bieżącej przestrzeni nazw (`main`) znajduje się już deklaracja klasy `Debug`? Łatwo zgadnąć. Oto stosowny kod i efekt jego wykonania:

```
namespace main;
use com\getinstance\util\Debug;

class Debug {
    static function helloWorld() {
        print "hello from main\Debug";
    }
}

Debug::helloWorld();
```

PHP Fatal error: Cannot declare class `main\Debug` because the name is already in use in ...

Zatoczyliśmy więc koło, wracając ponownie do kolizji nazw klas, nieprawdaż? Na szczęście nasz problem ma rozwiązanie w postaci jawnych aliasów dla używanych nazw:

```
namespace main;

use com\getinstance\util\Debug as uDebug;

class Debug {
    static function helloWorld() {
        print "hello from main\Debug";
    }
}

uDebug::helloWorld();
```

Użycie słowa `as` w klauzuli `use` pozwala na zmianę aliasu nazwy `Debug` na `uDebug`.

Kiedy programista pisze kod w jakiejś przestrzeni nazw i zamierza odwołać się do klasy z globalnej (nienazwanej) przestrzeni nazw, może po prostu poprzedzić nazwę klasy pojedynczym znakiem ukośnika. Oto deklaracja metody w globalnej przestrzeni nazw:

```
// plik global.php: bez przestrzeni nazw

class Lister {
    public static function helloWorld() {
        print "ahoj z modułu głównego\n";
    }
}
```

A oto kod zamknięty w przestrzeni nazw, odwołujący się do owej metody:

```
namespace com\getinstance\util;
require_once 'global.php';

class Lister {
    public static function helloWorld() {
        print "ahoj z modułu ".__NAMESPACE__."\n";
    }
}

Lister::helloWorld(); // odwołanie lokalne
\Lister::helloWorld(); // odwołanie globalne
```

Kod z przestrzeni nazw deklaruje własną wersję klasy Lister. Odwołanie z nazwą niekwalifikowaną to odwołanie do wersji lokalnej; odwołanie z nazwą kwalifikowaną pojedynczym znakiem ukośnika to odwołanie do klasy z globalnej przestrzeni nazw.

Oto efekt wykonania poprzedniego fragmentu kodu.

```
ahoj z modułu com\getinstance\util
ahoj z modułu głównego
```

Warto go pokazać, bo przy okazji ilustruje działanie stałej `__NAMESPACE__`. Otóż przechowuje ona nazwę bieżącej przestrzeni nazw i bardzo przydaje się w diagnostyce błędów.

W pojedynczym pliku można deklorować więcej niż jedną przestrzeń nazw — składnia pozostaje bez zmian. Można też stosować składnię alternatywną, z użyciem nawiasów klamrowych ujmujących ciało deklaracji przestrzeni nazw.

```
namespace com\getinstance\util {
    class Debug {
        static function helloWorld() {
            print "ahoj, tu Debug\n";
        }
    }
}

namespace main {
    \com\getinstance\util\Debug::helloWorld();
}
```

Jeśli zachodzi konieczność użycia wielu przestrzeni nazw w pojedynczym pliku, składnia z nawiasami klamrowymi jest wręcz zalecana. Ogólnie jednak zaleca się, aby przestrzenie nazw były definiowane w osobnych plikach.

Unikatową cechą składni z nawiasami klamrowymi jest możliwość przełączenia się do globalnej przestrzeni nazw wewnątrz pliku. Wcześniej do pozyskania kodu z globalnej przestrzeni nazw użyliśmy dyrektywy `require_once`. Mogliśmy jednak użyć alternatywnej składni przestrzeni nazw i zamknąć wszystko w jednym pliku.

```
namespace {
    class Lister {
        //...
    }
}

namespace com\getinstance\util {
    class Lister {
        //...
    }

    Lister::helloWorld(); // odwołanie lokalne
    \Lister::helloWorld(); // odwołanie globalne
}
```

Do globalnej przestrzeni nazw weszliśmy, otwierając blok przestrzeni nazw bez określenia nazwy.

- **Uwaga** Nie można mieszać składni wierszowej ze składnią klamrową w jednym pliku — w obrębie pliku trzeba wybrać jedną składnię i konsekwentnie się jej trzymać.

Symulowanie systemu pakietów na bazie systemu plików

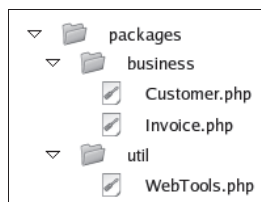
Niezależnie od wykorzystywanej wersji PHP możemy na własną rękę organizować klasy w pakiety, wykorzystując struktury charakterystyczne dla systemu plików. Możemy na przykład wydzielić dla dwóch grup klasy osobne katalogi (np. *util* i *business*) i włączać przechowywane w nich pliki implementujące klasy za pośrednictwem funkcji `require_once()`, jak poniżej:

```
require_once("business/Customer.php");
require_once("util/WebTools.php");
```

Z podobnym efektem można zastosować funkcję `include_once()`. Różnica pomiędzy instrukcjami `include()` i `require()` tkwi w obsłudze błędów. Otóż plik wywołany za pomocą `require()` w przypadku błędu zatrzyma przetwarzanie całego programu. Taki sam błąd w pliku włączanym instrukcją `include()` zaledwie spowoduje ostrzeżenie i przerwie wykonanie kodu z wciąganego pliku, ale nie przerwie wykonania całego programu. Dzięki temu `require()` i `require_once()` stanowią bezpieczniejsze sposoby włączania plików bibliotecznych, a `include()` i `include_once()` są bardziej przydatne przy szablonach.

- **Uwaga** `require()` i `require_once()` to w istocie instrukcje, a nie funkcje. Oznacza to, że można przy nich zrezygnować z nawiasów. Osobiście i tak stosuję nawiasy, ale zdarzają się pedanci zanudzający wyjaśnieniami różnicy pomiędzy funkcją a instrukcją.

Rysunek 5.1 prezentuje tak uzyskany podział kodu w przeglądarce plików Nautilus.



Rysunek 5.1. Organizacja pakietów PHP w konwencji systemu plików

- **Uwaga** Argumentem wywołania funkcji `require_once()` jest ścieżka do pliku; funkcja wstawia plik do bieżącego skryptu po jego uprzednim przetworzeniu. Nastąpi to jednak jedynie wtedy, kiedy plik określony przez argument wywołania nie został jeszcze włączony do procesu w innym miejscu. Tego rodzaju zabezpieczenie przed wielokrotnym włączaniem kodu jest użyteczne zwłaszcza w kodzie bibliotecznym, zapobiega bowiem przypadkowemu ponownemu definiowaniu klas i funkcji, do czego mogłoby dojść, gdyby plik kodu bibliotecznego był włączany do skryptu w kilku różnych miejscach za pośrednictwem funkcji `require()` czy `include()`.

Programista ma swobodę wyboru pomiędzy funkcjami `require()` i `require_once()` a podobnymi (ale nie identycznymi) w działaniu `include()` i `include_once()`, zalecałbym jednak korzystanie z tych pierwszych, a to dlatego, że błąd w pliku odczytywanym za pomocą funkcji `require()` przerywa wykonywanie skryptu. Taki sam błąd występujący w pliku włączanym do skryptu wywołaniem `include()` powoduje zaś jedynie wygenerowanie ostrzeżenia w skrypcie wywołującym, ale nie przerywa jego działania. W tym przypadku za bezpieczniejsze należy zaś uznać środki drastyczniejsze.

Z zastosowaniem `require_once()` w porównaniu z `require()` związany jest pewien narzut. Otóż tam, gdzie ważne są nawet milisekundy działania programu, warto rozważyć użycie `require()`.

Jeśli chodzi o PHP, struktura ta nie ma w sobie niczego szczególnego: różne skrypty biblioteczne umieszczamy po prostu w różnych katalogach. Wymusza to porządek w organizacji kodu i nie wyklucza używania przestrzeni nazw bądź klasycznych konwencji nazewniczych.

Nazwy à la PEAR

W praktyce nie zawsze można skorzystać z dobrodziejstw przestrzeni nazw; w zastanym oprogramowaniu modernizacja kodu może się okazać przedsięwzięciem nieopłacalnym — mimo że oprogramowanie wciąż jest używane i rozwijane. A nawet jeśli dany projekt w całości oparty jest na najnowszej wersji PHP, nie obędzie się pewnie bez wykorzystania jakiegoś starszego kodu. Jeśli możemy pozwolić sobie na przepisanie go na nazwy klas — świetnie. W większości przypadków będzie to jednak nieosiągalny luksus.

Jak więc poradzić sobie z ryzykiem kolizji nazw, jeśli nie można zdać się w całości na przestrzenie nazw? Jeden sposób już zaznaczyliśmy — mowa o wykorzystaniu konwencji nazewniczej typowej dla pakietów PEAR.

-
- **Uwaga** PEAR to skrót od *PHP Extension and Application Repository* (repozytorium rozszerzeń i aplikacji PHP). To oficjalne archiwum pakietów i narzędzi rozszerzających możliwości i zakres zastosowań języka PHP. Podstawowe pakiety z tego repozytorium wchodzi w skład dystrybucji PHP, inne mogą być do niej dodawane za pośrednictwem prostego narzędzia wywoływanego z wiersza polecenia. Pod adresem <http://pear.php.net> dostępna jest przeglądarka pakietów repozytorium. Do aspektów korzystania z PEAR wrócimy w rozdziale 15.
-

W PEAR stosuje się strukturę pakietów bazującą właśnie na systemie plików. Nazwa każdej z klas jest więc odzwierciedleniem ścieżki dostępu — nazwy poszczególnych podkatalogów są w nazwie klasy rozdzielane znakiem podkreślenia.

Repozytorium PEAR obejmuje na przykład pakiet o nazwie XML, zawierający pakiet RPC. Pakiet RPC zawiera z kolei plik o nazwie *Server.php*. Klasa definiowana wewnątrz tego pliku nie nosi bynajmniej prostej nazwy *Server*. Prędzej czy później stosowanie tak oczywistej nazwy doprowadziłoby bowiem do kolizji z kodem użytkującym pakiet RPC. Rzeczona klasa nosi więc nazwę *XML_RPC_Server*. Nie czyni to nazwy klasy atrakcyjniejszą, zwiększa jednak łatwość czytania kodu, bo nazwa klasy zawsze opisuje swój własny kontekst.

Ścieżki przeszukiwania

Przy organizowaniu komponentów warto pamiętać o dwóch perspektywach. Pierwszą mamy omówioną: chodzi o położenie plików i katalogów w systemie plików. Trzeba też jednak uwzględnić sposób realizacji odwołań pomiędzy komponentami. Jak dotąd zignorowałem niemal całkowicie tematykę ścieżek dostępu występujących w wywołaniach funkcji włączających kod do skryptu. Tymczasem, włączając plik kodu, możemy określać ów plik za pośrednictwem ścieżki względnej, odnoszącej się do bieżącego katalogu roboczego, albo ścieżki bezwzględnej, zakorzenionej w katalogu głównym systemu plików.

W prezentowanych dotychczas przykładach stosowaliśmy wyłącznie ścieżki względne:

```
require_once("business/User.php");
```

Ale to oznacza konieczność obecności w bieżącym katalogu roboczym podkatalogu *business*, a prędzej czy później taki wymóg stanie się niepraktyczny. Jeśli już chce się stosować w wywołaniach włączających kod biblioteczny ścieżki względne, to lepiej, aby miały one postać:

```
require_once("../projectlib/business/User.php");
```

Można by też stosować ścieżki bezwzględne:

```
require_once("/home/john/projectlib/business/User.php");
```

Żadne rozwiązanie nie jest jednak idealne, bo określając ścieżkę zbyt szczegółowo, zamrażamy niejako położenie pliku bibliotecznego.

Przy stosowaniu ścieżek bezwzględnych wiążemy połączenie bibliotek z konkretnym systemem plików. Instalacja projektu na nowym serwerze wymaga wtedy aktualizacji wszystkich wywołań funkcji włączających pliki biblioteczne.

Stosując ścieżki względne, ustalamy położenie plików bibliotecznych względem bieżącego katalogu roboczego, przez co utrudniamy przenoszenie plików bibliotecznych. W ten sposób utrudnia się przeniesienie biblioteki w systemie plików bez koniecznej zmiany instrukcji `require()`, co sprawia, że w projektach innych niż macierzysty biblioteki nie da się łatwo używać. W obu zaś przypadkach tracimy perspektywę pakietu na rzecz perspektywy systemu plików — nie bardzo wiadomo bowiem, czy mamy pakiet `business`, czy może `projectlib/business`.

Aby ułatwić sobie odwoływanie się do plików bibliotecznych, musimy więc oddzielić kod wywołujący od konkretnego położenia plików bibliotecznych, tak aby ścieżkę:

```
business/User.php
```

można było wykorzystać w dowolnym miejscu systemu i aby w każdym z nich odnosiła się ona do tego samego pakietu. Można to osiągnąć, umieszczając pakiet w jednym z katalogów, do których odwołuje się parametr `include_path`. Parametr ten jest zwykle ustawiany w pliku `php.ini` — centralnym pliku konfiguracji PHP. Definiuje on listę ścieżek dostępu wymienianych po dwukropkach (w systemach uniksowych) albo średnikach (w systemach z rodziny Windows).

```
include_path = ".:usr/local/lib/php-libraries"
```

Użytkownicy serwera Apache mogą też ustawić dyrektywę `include_path` w pliku konfiguracyjnym serwera (zazwyczaj jest to plik `httpd.conf`) albo w plikach konfiguracji poszczególnych katalogów (zazwyczaj pod nazwą `.htaccess`). Odbywa się to za pomocą składni:

```
php_wartość include_path wartość .:usr/local/lib/php-libraries
```

■ **Uwaga** Pliki `.htaccess` są przydatne zwłaszcza w przestrzeni WWW udostępnianej przez firmy hostingowe, w których do środowiska konfiguracji samych serwerów mamy bardzo ograniczony dostęp.

W wywołaniach funkcji systemowych, jak `fopen()` czy `require()`, z względnymi ścieżkami dostępu, których nie uda się dopasować w kontekście bieżącego katalogu roboczego, inicjowane jest przeszukiwanie ścieżek wymienionych w ramach parametru `include_path` w kolejności zgodnej z kolejnością ich definiowania w ramach parametru (w przypadku funkcji `fopen()` włączenie automatycznego przeszukiwania ścieżek wymaga przekazania za pomocą argumentów odpowiedniego znacznika). Wyszukiwanie kończy się po odnalezieniużądanego pliku w którymś z kolejnych katalogów wymienionych w ramach parametru `include_path`.

Gdy umieścimy katalog pakietów w katalogach ścieżek przeszukiwania, możemy w wywołaniach włączających kod zrezygnować z samodzielnego określenia ścieżek dostępu.

W takim układzie listę ścieżek przeszukiwania należałoby uzupełnić o wyróżniony katalog przeznaczony wyłącznie na pliki biblioteczne. Wymaga to edycji pliku konfiguracji `php.ini` (oczywiście wprowadzone w nim zmiany zostaną uwzględnione przez moduł PHP serwera dopiero po przeładowaniu tego serwera).

W przypadku nieposiadania uprawnień niezbędnych do modyfikowania pliku `php.ini` można uciec się do modyfikacji parametru `include_path` z poziomu samego skryptu. Służy do tego funkcja `set_include_path()`. Funkcja ta przyjmuje w wywołaniu ciąg reprezentujący ścieżkę przeszukiwania i ustawia ową ścieżkę, ale wyłącznie dla bieżącego procesu. Zazwyczaj parametr `include_path` zdefiniowany w pliku `php.ini` zawiera już przydatne ścieżki przeszukiwania, więc zamiast go zamazywać, można go uprzednio odczytać i jedynie dopisać swoje ścieżki do bieżącej wartości parametru. Odczyt parametru `include_path` możliwy jest za pośrednictwem funkcji `get_include_path()`. Opisywane uzupełnienie ścieżek przeszukiwania może wyglądać następująco:

```
set_include_path(get_include_path() . PATH_SEPARATOR . "/home/john/phplib/");
```

Stała `PATH_SEPARATOR` będzie w systemach Unix zamieniana na znak dwukropka, a w systemach Windows na znak średnika; jej stosowanie przybliży nas więc do pożądanej wysokiej przenośności kodu aplikacji.

Automatyczne wczytywanie kodu

W pewnych okolicznościach pożądana jest taka organizacja klas, aby każda z nich była trzymana w osobnym pliku. Taki model ma swoje wady (włączanie dużej liczby małych plików może zwiększać ogólny koszt włączania), ale jest użyteczny, zwłaszcza kiedy system ma być rozbudowywany i ma korzystać z nowych klas w czasie wykonania (zobacz wzorzec *Command* w rozdziałach 11. i 12.). W takich przypadkach nazwy plików poszczególnych klas mogą mieć powiązania z nazwami klas zapisanych w tych plikach: klasa *ShopProduct* wylądowuje więc w pliku *ShopProduct.php*. Można pójść o krok dalej i użyć konwencji nazw pakietowych PEAR. W takim przypadku, jeśli zechcemy zdefiniować klasę *ShopProduct* w pakiecie o nazwie *business*, powinniśmy przy nazwie *ShopProduct.php* umieścić w katalogu o nazwie *business*. Samą klasę należy natomiast zdefiniować przez nazwę pakietową, a mianowicie *business_ShopProduct*. Alternatywnie, jeśli można sobie pozwolić na stosowanie przestrzeni nazw, można stosować konwencję PEAR odnośnie do rozmieszczenia plików (*business/ShopProduct.php*), ale pakietowe adresowanie klas przenieść z nazwy klasy do nazwy przestrzeni nazw.

W PHP5 zaimplementowano mechanizmy ładowania klas pomocne w automatyzacji włączania plików klas. Domyślne działanie tych mechanizmów jest dość ograniczone, niemniej jednak już przydatne. Można z niego skorzystać, wywołując funkcję o nazwie `spl_autoload_register()` (bez argumentów). Po aktywowaniu w ten sposób mechanizmu automatycznego ładowania klas za każdym razem, kiedy zechcemy utworzyć egzemplarz nieznaną jeszcze klasy, dojdzie do wywołania specjalnej funkcji o nazwie `spl_autoload()`. Funkcja `spl_autoload()` otrzyma w wywołaniu nazwę klasy i spróbuje użyć przekazanej nazwy (po konwersji na małe litery) uzupełnionej o rozszerzenie (domyślnie *.php* lub *.inc*) do znalezienia pliku klasy w systemie plików.

Oto prosty przykład:

```
spl_autoload_register();
$writer = new Writer();
```

Przy założeniu, że nie włączyliśmy jeszcze do aplikacji pliku zawierającego klasę *Writer*, powyższa próba utworzenia obiektu nie może się udać. Ale skoro wcześniej uruchomiliśmy mechanizm automatycznego ładowania klas, PHP spróbuje znaleźć i włączyć do aplikacji plik *writer.php* lub *writer.inc*, i ponownie przeprowadzić konkretyzację obiektu klasy *Writer*. Jeśli któryś z tych plików istnieje i zawiera klasę *Writer*, druga próba zakończy się sukcesem.

Domyślny mechanizm ładowania klas obsługuje przestrzenie nazw, odwzorowując kolejne nazwy pakietowe na nazwy katalogów. Poniższy kod:

```
spl_autoload_register();
$writer = new util\Writer();
```

sprowołuje wyszukanie pliku o nazwie *writer.php* (pamiętajmy o zamianie wielkości liter w nazwie klasy) w katalogu o nazwie *util*.

A jeśli pliki z klasami będą miały nazwy zawierające wielkie litery? Jeśli klasa *Writer* zostanie umieszczona w pliku *Writer.php*, to domyślny mechanizm ładowania klas nie poradzi sobie z odszukaniem pliku klasy.

Na szczęście możemy rejestrować własne funkcje obsługi ładowania klas, w których można implementować dowolne konwencje odwzorowania nazwy klasy na plik. Aby skorzystać z tego udogodnienia, należy do wywołania `spl_autoload_register()` przekazać referencję do własnej (może być anonimowa) funkcji ładującej. Funkcja ładująca powinna przyjmować pojedynczy argument. Jeśli wtedy PHP napotka próbę utworzenia egzemplarza niezaladowanej jeszcze klasy, zainicjuje wywołanie naszej funkcji z pojedynczym argumentem zawierającym nazwę klasy. Funkcja ładująca może zupełnie arbitralnie definiować strategię odwzorowania i włączania brakujących plików klas. Po zakończeniu wykonywania funkcji ładującej PHP ponownie spróbuje utworzyć egzemplarz klasy.

Oto prosty przykład własnej funkcji ładującej:

```
function straightIncludeWithCase($classname) {
    $file = "{$classname}.php";
    if (file_exists($file)) {
        require_once($file);
    }
}
spl_autoload_register('straightIncludeWithCase');
$product = new ShopProduct('The Darkening', 'Harry', 'Hunter', 12.99);
```

Po nieudanej pierwszej próbie utworzenia obiektu klasy `ShopProduct` PHP uwzględni funkcję ładującą zarejestrowaną wywołaniem `spl_register_function()` i przekaże do niej ciąg znaków `"ShopProduct"`. Nasza implementacja tej funkcji ogranicza się jedynie do próby włączenia pliku o nazwie skonstruowanej na bazie przekazanego ciągu. Poprawne włączenie pliku jest uwarunkowane jego obecnością w bieżącym katalogu roboczym albo w jednym z katalogów wymienionych w ramach parametru `include_path` (czy to zgodnie ze starą konwencją nazewniczą PEAR, czy to zgodnie z konwencją przestrzeni nazw). Bardzo łatwo zaimplementować też obsługę nazw pakietowych PEAR:

```
function replaceUnderscores($classname) {
    $path = str_replace('_', DIRECTORY_SEPARATOR, $classname);
    if (file_exists("${path}.php")) {
        require_once("${path}.php");
    }
}
spl_autoload_register('replaceUnderscores');

$x = new ShopProduct();
$y = new business_ShopProduct();
```

W powyższej implementacji w ciele funkcji `replaceUnderscores()` następuje dopasowanie znaków podkreślenia występujących w argumencie wywołania `$classname` i ich zamiana na znaki separatora katalogów (w systemach uniksowych rolę tę pełnią znaki ukośnika `— /`). Ostatecznie więc do skryptu włączany jest plik `business/ShopProduct.php`. Jeśli taki plik istnieje, a zawarta w nim klasa ma odpowiednią nazwę, uda się skutecznie skonkretyzować obiekt klasy. To znaczne ułatwienie, pod warunkiem że programiści dostosują się i będą się konsekwentnie trzymać raz przyjętej konwencji nazewniczej klas i konwencji rozmieszczania plików definicji (i unikać stosowania znaków podkreśleń, jeśli nie reprezentują one katalogowego rozmieszczenia pakietów).

A co z przestrzeniami nazw? Wiemy, że domyślny mechanizm ładowania klas obsługuje przestrzeń nazw, odwzorowując je na podkatalogi. Ale jeśli przesłaniamy mechanizm ładowania własną funkcją, musimy ten przypadek również obsłużyć samodzielnie. Jest to zresztą jedynie kwestia dopasowania i zastąpienia znaków lewego ukośnika:

```
function myNamespaceAutoload($path) {
    if (preg_match('/\\\\\\\\/', $path)) {
        $path = str_replace('\\\\', DIRECTORY_SEPARATOR, $path);
    }
    if (file_exists("${path}.php")) {
        require_once("${path}.php");
    }
}
```

Wartość przekazywana do funkcji ładującej jest zawsze znormalizowana do postaci pełnej kwalifikowanej nazwy klasy, z pominięciem pierwszego ukośnika, nie ma więc potrzeby martwienia się o rozpoznanie przestrzeni nazw czy uwzględnianie aliasów klas.

A jak zrealizować rozpoznawanie nazw klas w konwencji PEAR i nazw klas używających przestrzeni nazw? Cóż, wystarczy połączyć dwie pokazywane implementacje funkcji ładujących i przekształcić je w jedną funkcję uniwersalną. Można też zarejestrować więcej niż jedną funkcję ładującą, bo funkcje rejestrowane przez `spl_register_autoload()` są zachowywane w kolejności rejestracji:

```
spl_autoload_register('replaceUnderscores');
spl_autoload_register('myNamespaceAutoload');

$x = new ShopProduct();
$y = new business_ShopProduct();
$z = new business\ShopProduct2();
$a = new \business\ShopProduct3();
```

Kiedy PHP napotka nieznaną klasę, będzie wywoływać kolejno funkcje `replaceUnderscores()` i `myNamespaceAutoLoad()` do momentu, kiedy kolejna próba utworzenia obiektu zakończy się powodzeniem albo wyczerpie się lista funkcji ładujących.

Oczywiście stosowanie kaskady funkcji ładujących oznacza pewien narzut czasowy wykonania, po co więc wprowadzono taką możliwość? Otóż w praktyce łączy się odwzorowanie klas według różnych konwencji w jednej funkcji ładującej. Ale w większych systemach, z dużym udziałem komponentów zewnętrznych, konieczność rejestrowania dodatkowych funkcji ładujących może się okazać nieunikniona — wiele bibliotek stosuje własne, unikatowe mechanizmy ładowania. Kaskada funkcji ładujących pozwala na realizowanie niezależnych metod ładowania klas w poszczególnych komponentach. Zresztą kiedy porządnie napisana biblioteka nie potrzebuje już stosować własnego mechanizmu ładowania, może swoją funkcję ładującą wyrejestrować wywołaniem metody `spl_unregister_function()`!

-
- **Uwaga** PHP obsługuje funkcję `__autoload()`, realizującą znacznie mniej zaawansowany mechanizm zarządzania automatyzacją włączania plików; jeśli zaimplementujemy taką funkcję, PHP prześle do niej kompetencję ładowania klas w przypadku nieudanej próby utworzenia obiektu. Jest to jednak podejście mniej uniwersalne, bo funkcja `__autoload()` może mieć tylko jedną implementację; jest też wielce prawdopodobne, że w przyszłych wersjach języka PHP zostanie ona wycofana z użycia.
-

Klasy i funkcje pomocnicze

Programista języka PHP ma do swojej dyspozycji szereg klas i funkcji służących do analizowania obiektów. Jaka jest ich przydatność? W końcu większość klas wykorzystywanych w projekcie konstruujemy sami i mamy pełną wiedzę o strukturze ich obiektów.

Często jednak nie posiadamy wystarczających informacji o obiektach wykorzystywanych w czasie wykonywania skryptu, niejednokrotnie bowiem własne projekty opieramy na transparentnym delegowaniu zadań do klas autorstwa osób trzecich. W takich przypadkach obiekt konkretyzuje się często jedynie na podstawie dynamicznie konstruowanej nazwy klasy. PHP pozwala na dynamiczne odwołania do klas za pośrednictwem ciągów znaków, jak tutaj:

```
// plik Task.php
namespace tasks;
class Task {
    function doSpeak() {
        print "Ahoj\n";
    }
}

// Plik TaskRunner.php
$classname = "Task";

require_once( "tasks/{$classname}.php" );
$classname = "tasks\\$classname";
$myObj = new $classname();
$myObj->doSpeak();
```

Ciąg przypisywany powyżej do zmiennej `$classname` typowo odczytywany jest z pliku konfiguracyjnego albo określany na podstawie odebranego żądania z zawartością katalogu. Ciąg taki można wykorzystać do wczytania pliku definicji klasy i konkretyzacji jej obiektu. Zauważmy, że w tym fragmencie skonstruowaliśmy *de facto* kwalifikację przestrzeni nazw.

Operacje tego rodzaju wykorzystywane są w takich systemach, które mają zapewniać możliwość uruchamiania dodatków i rozszerzeń definiowanych zewnętrznie. Zanim dopuścimy tego rodzaju rozwiązanie w prawdziwym (a nie tylko przykładowym) skrypcie, powinniśmy jeszcze upewnić się, że żądana klasa istnieje, a także sprawdzić, czy udostępniła oczekiwane metody itd.

- **Uwaga** Nawet w obliczu zabezpieczeń należy zachować szczególną ostrożność przy dynamicznym instalowaniu zewnętrznego kodu. Nie powinno się pod żadnym pozorem automatycznie łączyć kodu dostarczanego przez użytkowników zewnętrznych: każdy tak zainstalowany dodatek może zazwyczaj wykonywać się z uprawnieniami właściwymi dla całej aplikacji, więc złośliwy kod może spowodować niemałe zamieszanie w systemie.

Nie oznacza to, że dynamiczne ładowanie kodu nie jest w ogóle przydatne; możliwość rozszerzania podstawowej funkcjonalności systemu przez programistów trzecich może zaowocować znacznym zwiększeniem elastyczności systemu. Aby przy tym zachować jego bezpieczeństwo, można na przykład rozważyć zabezpieczenie w postaci wydzielenia katalogu kodu ładowanego dynamicznie, z uprawnieniami ograniczonymi do grona administratorów albo programistów zarejestrowanych i posiadających konto w specjalnie wydzielonym repozytorium; w takim układzie administrator systemu mógłby osobiście rewidować kod z repozytorium i wybiórczo instalować rozszerzenia. W ten sposób działa na przykład popularna platforma WordPress.

Niektóre z funkcji analizy klas zostały w PHP5 zdublowane w ramach znacznie rozbudowanego interfejsu Reflection API, któremu przyjrzymy się w dalszej części rozdziału. Jednak ich prostota i łatwość użycia czynią je bardzo wygodnymi narzędziami.

Szukanie klasy

Funkcja `class_exists()` przyjmuje w wywołaniu ciąg reprezentujący klasę do zbadania i zwraca wartość logiczną: `true`, jeśli klasa istnieje, i `false`, jeśli nie istnieje (nie napotkano dotąd definicji klasy).

Za pomocą tej funkcji możemy uczynić poprzedni fragment kodu odrobinę bezpieczniejszym:

```
// plik TaskRunner.php
$classname = "Task";

$path = "tasks/{$classname}.php";
if (!file_exists($path)) {
    throw new Exception("Brak pliku {$path}");
}

require_once($path);
$qclassname = "tasks\\$classname";
if (!class_exists($qclassname)) {
    throw new Exception("Brak klasy $qclassname");
}

$myObj = new $qclassname();
$myObj->doSpeak();
```

Nie daje nam to jeszcze pewności co do wymaganych argumentów wywołania konstruktora. Aby mieć taką pewność i uczynić konkretyzację obiektu jeszcze bardziej niezawodną, musimy uciec się do interfejsu Reflection API opisywanego w dalszej części rozdziału. Tak czy inaczej wywołanie `class_exists()` pozwala na sprawdzenie obecności klasy przed próbą jej użycia.

- **Uwaga** Pamiętajmy, że zawsze należy ostrożnie korzystać z danych pobieranych ze źródeł zewnętrznych. Każdorazowo trzeba je weryfikować przed właściwym użyciem. W przypadku ścieżki dostępu do pliku należy usunąć albo oznaczyć kropki oraz znaki separatora katalogów — w ten sposób zabezpiecza się kod przed niepożądaną zmianą katalogu i włączeniem do programu nieoczekiwanych plików. Natomiast w przypadku budowania rozszerzalnych systemów techniki te dotyczą generalnie właściciela systemu (posiadającego uprawnienia do zapisu plików w katalogach), a nie użytkowników zewnętrznych.

Programista może również uzyskać tablicę wszystkich zdefiniowanych dotąd klas — wystarczy, że wywoła funkcję `get_declared_classes()`:

```
print_r(get_declared_classes());
```

Po wykonaniu tej instrukcji na wyjściu skryptu pojawi się lista klas, obejmująca zarówno klasy definiowane przez użytkownika, jak i te wbudowane. Warto pamiętać, że lista obejmuje jedynie te klasy, których deklaracje zostały przetworzone przed momentem wywołania funkcji. Lista klas używanych w toku wykonania skryptu może przecież być później uzupełniana, choćby za pomocą wywołań `require()` czy `require_once()`.

Badanie obiektów i klas

Jak już Czytelnikowi wiadomo, obiektowe typy argumentów wywołania metod mogą być narzucane przez klasy. Mimo to nie zawsze możemy jednak mieć pewność co do konkretnego typu obiektu przetwarzanego w ramach klasy — w czasie przygotowywania tej publikacji język PHP nie pozwalał bowiem choćby na wymuszenie konkretnego typu obiektu zwracanego przez metody — taki mechanizm jest zapowiadany w następnych wydaniach PHP.

Typ obiektu można sprawdzać za pośrednictwem rozmaitych narzędzi. Przede wszystkim programista może sprawdzać klasę obiektu — służy do tego funkcja `get_class()`. Funkcja ta przyjmuje w wywołaniu obiekt dowolnej klasy i zwraca ciąg znaków reprezentujący nazwę klasy:

```
$product = getProduct();
if (get_class($product) === 'CdProduct') {
    print "\$product to obiekt klasy CdProduct\n";
}
```

W powyższym przykładzie pobieramy *coś* z funkcji `getProduct()`. Aby zyskać pewność, że zwrócona wartość jest oczekiwanym obiektem klasy `CdProduct`, korzystamy z wywołania funkcji `get_class()`.

■ **Uwaga** Klasy `CdProduct` i `BookProduct` były prezentowane w rozdziale 3.

Oto kod funkcji `getProduct()`:

```
function getProduct() {
    return new CdProduct("Exile on Coldharbour Lane", "The", "Alabama 3", 25.99,
        60.33);
}
```

Jak widać, funkcja `getProduct()` po prostu konkretyzuje obiekt klasy `CdProduct`. Przyda się nam ona w tym rozdziale jeszcze wielokrotnie.

Funkcja `get_class()` jest narzędziem bardzo szczególnym, często potrzebujemy zaś bardziej ogólnego potwierdzenia typu klasy. Możemy na przykład próbować określić przynależność obiektu do hierarchii `ShopProduct`, ale bez rozróżniania pomiędzy poszczególnymi klasami tej hierarchii — nie interesuje nas bowiem, czy obiekt jest klasy `BookProduct`, czy `CdProduct`; ważne, że reprezentuje jakiś asortyment. Aby to stwierdzić, należy posłużyć się operatorem `instanceof`.

■ **Uwaga** W PHP4 brakowało operatora `instanceof`. Zamiast niego dostępna była funkcja `is_a()`, która jednak w wersji 5.0 została oznaczona jako zarzucona. Ponownie przywrócono ją w PHP 5.3.

Operator `instanceof` działa na dwóch operandach: lewym jest obiekt podlegający badaniu pod kątem przynależności do hierarchii klas, a prawy to nazwa klasy albo interfejsu. Jeśli obiekt jest egzemplarzem danej klasy (interfejsu), operator zwraca wartość logiczną `true`.

```
$product = getProduct();
if ($product instanceof ShopProduct) {
    print "\$product jest obiektem klasy ShopProduct\n";
}
```


Pozyskiwanie ciągu pełnej nazwy klasy

Przestrzenie nazw umożliwiły wyeliminowanie wielu niedogodności obiektowej implementacji PHP. Nie musimy już tolerować niedorzecznie rozbudowanych nazw klas ani ryzykować kolizji nazw (to dotyczy już tylko zastanego, niezmodyfikowanego kodu). Z drugiej strony, względne odwołania do przestrzeni nazw i aliasy utrudniają niekiedy określenie pełnej nazwy klasy, jak w poniższych przypadkach:

```
namespace mypackage;

use util as u;
use util\db\Querier as q;

class Local {}

// Zagadki:

// Przestrzeń nazw określana przez alias
// u\Writer;

// Klasa określana przez alias
// q;

// Klasa wymieniana w kontekście lokalnym
// Local
```

Określenie właściwej nazwy klasy nie wydaje się bardzo trudne, ale implementacja kodu, który zadziała poprawnie we wszystkich możliwych kombinacjach, jest już kłopotliwa. Weźmy na przykład `u\Writer`. W przypadku takiej nazwy automat musiałby „wiedzieć”, że `u` jest aliasem przestrzeni nazw `util`, a nie właściwą nazwą przestrzeni nazw. Na szczęście w PHP 5.5 wprowadzono składnię odwołania `NazwaKlasy::class`. Innymi słowy, dowolną posiadaną referencję do klasy możemy uzupełnić o operator zasięgu i słowo kluczowe `class` w celu pozyskania pełnej kwalifikowanej nazwy klasy. Tak więc poniższy kod:

```
print u\Writer::class."\n";
print q::class."\n";
print Local::class."\n";
wypisze na wyjściu:

util\Writer
util\db\Querier
mypackage\Local
```

Badanie metod

Za pośrednictwem funkcji `get_class_methods()` możemy pozyskać listę wszystkich metod udostępnianych przez klasę. Funkcja ta wymaga przekazania w wywołaniu nazwy klasy, a zwraca tablicę z nazwami wszystkich metod tejże klasy:

```
print_r(get_class_methods('CdProduct'));
```

Jeśli założymy dostępność klasy `CdProduct`, na wyjściu powinno pojawić się coś takiego:

```
Array
(
    [0] => __construct
    [1] => getPlayLength
    [2] => getSummaryLine
    [3] => getProducerFirstName
```



```

[4] => getProducerMainName
[5] => setDiscount
[6] => getDiscount
[7] => getTitle
[8] => getPrice
[9] => getProducer
)

```

W tym przykładzie przekazujemy w wywołaniu funkcji `get_class_methods()` nazwę klasy zdefiniowanej w poprzednich rozdziałach i wynik wywołania przekazujemy natychmiast do funkcji `print_r()`, wypisującej go na wyjście skryptu. Identyczny efekt osiągnęlibyśmy, przekazując w wywołaniu `get_class_methods()` nie nazwę klasy, a jej *obiekt*.

Użytkownicy najwcześniejszych wersji PHP5 zobaczą na wykazie komplet metod — w nieco późniejszych wersjach wykaz introspekcji klasy obejmuje jedynie metody publiczne.

Nazwy metod są reprezentowane jako ciągi znaków, co daje możliwość dynamicznego konstruowania ich wywoływań na rzecz obiektu, jak tutaj:

```

$product = getProduct(); // pozyskanie obiektu...
$method = "getTitle";    // konstrukcja nazwy metody...
print $product->$method(); // wywołanie metody...

```

Takie konstrukcje mogą oczywiście być groźne. Co będzie w przypadku nieobecności metody w klasie? Oczywiście skrypt zostanie przerwany z powodu krytycznego błędu wykonania. Znamy już jeden sposób testowania klasy na obecność metody:

```

if (in_array($method, get_class_methods($product))) {
    print $product->$method(); // wywołanie metody...
}

```

Upewniamy się w ten sposób co do istnienia w klasie obiektu konkretnej metody. Jednak ten sam test możemy wykonać za pośrednictwem bardziej specjalizowanego narzędzia dostępnego w PHP. Nazwy metod możemy konfrontować z definicją klasy za pośrednictwem dwóch funkcji: `is_callable()` i `method_exists()`. Pierwsza z tych dwóch funkcji jest bardziej specjalizowana; przyjmuje ciąg znaków reprezentujący nazwę funkcji i zwraca `true`, jeśli funkcja istnieje i może zostać wywołana. W przypadku metod argument wywołania powinien mieć postać tablicy, której pierwszy element zawiera obiekt albo nazwę klasy, a drugi — nazwę metody do sprawdzenia. W tej wersji wywołania wartość `true` zwracana z funkcji oznacza obecność metody w klasie:

```

if (is_callable(array($product, $method))) {
    print $product->$method(); // wywołanie metody...
}

```

Funkcja `is_callable()` opcjonalnie przyjmuje drugi argument wywołania, którym powinna być zmienna logiczna. Jeśli ustawimy ją na `true`, funkcja będzie sprawdzać jedynie składnię danej nazwy, a nie faktyczną obecność metody czy funkcji o takiej nazwie.

Funkcja `method_exists()` wymaga przekazania obiektu (albo nazwy klasy) oraz nazwy metody i zwraca `true`, jeśli w klasie danego obiektu występuje wskazana metoda:

```

if (method_exists($product, $method)) {
    print $product->$method(); // wywołanie metody...
}

```

■ **Ostrzeżenie** Warto pamiętać, że obecność metody w klasie nie oznacza jeszcze możliwości jej wywołania w danym kontekście. Funkcja `method_exists()` zwraca bowiem `true` również dla metod oznaczonych jako prywatne i zabezpieczone, których nie da się wywołać spoza klasy obiektu.

Badanie składowych

Tak jak można wykrywać w klasie obecność metod, można też badać ją pod kątem obecności konkretnych składowych. Pełny wykaz składowych zwraca funkcja `get_class_vars()` przyjmująca w wywołaniu nazwę klasy. Zwracana przez nią tablica asocjacyjna zawiera nazwy składowych w roli kluczy i wartości składowych w roli wartości. Spróbujmy przetestować za jej pomocą zawartość składowych w klasie `CdProduct`. Dla lepszej ilustracji działania funkcji uzupełnimy tę klasę o publiczną składową `CdProduct::$coverUr1`:

```
print_r(get_class_vars('CdProduct'));
```

Jako wynik ujawni się tylko publiczna składowa:

```
Array
(
    [coverUr1] =>
)
```

Badanie relacji dziedziczenia

Funkcje badania klas pozwalają również na rozpoznawanie relacji dziedziczenia. Możemy więc dla danej klasy znaleźć jej klasę bazową — służy do tego funkcja `get_parent_class()`. Funkcja ta wymaga przekazania albo obiektu, albo nazwy klasy, a zwraca nazwę klasy nadrzędnej (bazowej), jeśli taka istnieje. W przeciwnym przypadku — czyli kiedy badana klasa nie posiada klasy bazowej — funkcja zwraca wartość `false`:

```
print get_parent_class('CdProduct');
```

Łatwo się domyślić, że w wyniku otrzymamy nazwę klasy nadrzędnej: `ShopProduct`.

Relację dziedziczenia możemy też analizować za pośrednictwem funkcji `is_subclass_of()`. Wymaga ona przekazania obiektu klasy pochodnej i nazwy klasy bazowej. Jeśli relacja dziedziczenia faktycznie zachodzi, tzn. jeśli klasa przekazanego obiektu faktycznie jest pochodną klasy określonej za pomocą drugiego argumentu domniemanej klasy bazowej, funkcja zwraca `true`:

```
$product = getProduct(); // pozyskanie obiektu
if (is_subclass_of($product, 'ShopProduct')) {
    print "CdProduct to klasa pochodna klasy ShopProduct\n";
}
```

Funkcja `is_subclass_of()` informuje jedynie o relacjach w obrębie drzewa dziedziczenia klas. Nie zwraca natomiast informacji o tym, że dana klasa implementuje interfejs. Do tego celu należy użyć operatora `instanceof`, ewentualnie funkcji wchodzącej w skład standardowej biblioteki języka PHP SPL (*Standard PHP Library*), a mianowicie funkcji `class_implements()`, która przyjmuje nazwę klasy bądź referencję obiektu i zwraca tablicę interfejsów implementowanych przez daną klasę (obiekt).

```
if (in_array('jakisInterfejs', class_implements($product))) {
    print "CdProduct jest interfejsem jakisInterfejs\n";
}
```

Badanie wywołań metod

Prezentowałem już przykład próby wywołania metody, której nazwa była określona poprzez dynamicznie konstruowany ciąg znaków:

```
$product = getProduct(); // pozyskanie obiektu...
$method = "getTitle"; // konstrukcja nazwy metody...
print $product->$method(); // wywołanie metody...
```

Programista PHP może podobny efekt uzyskać za pośrednictwem wywołania funkcji `call_user_func()`. Funkcja ta nadaje się tak do wywoływania zwykłych funkcji, jak i metod klas. Wywołanie funkcji wymaga przekazania pojedynczego argumentu — ciągu znaków zawierającego nazwę funkcji:

```
$returnValue = call_user_func("myFunction");
```

Wywołanie metody wymaga już przekazania tablicy. Pierwszym jej elementem powinien być obiekt, drugim zaś metoda, którą funkcja ma na rzecz owego obiektu wywołać:

```
$returnValue = call_user_func(array($myObj, "methodName"));
```

Argumenty wywołania docelowej funkcji czy metody, realizowanego za pośrednictwem funkcji `call_user_func()`, należy przekazywać za pośrednictwem kolejnych (to jest za pośrednictwem trzeciego i następnych) argumentów wywołania `call_user_func()`:

```
$product = getProduct(); // pozyskanie obiektu
call_user_func(array($product, 'setDiscount'), 20);
```

Powyższe dynamicznie skonstruowane wywołanie jest rzecz jasna równoznaczne poniższemu wywołaniu statycznemu:

```
$product->setDiscount(20);
```

Przydatność funkcji `call_user_func()` jest o tyle ograniczona, że dynamiczne wywołanie metody możemy skonstruować również samodzielnie:

```
$method = "setDiscount";
$product->$method(20);
```

Znacznie większe wrażenie robi już funkcja `call_user_func_array()`. Działa ona podobnie jak `call_user_func()`, przynajmniej jeśli chodzi o sposób określania docelowej funkcji czy metody wywołania. Tyle że wszelkie argumenty przekazywane do owego wywołania przyjmuje za pośrednictwem tablicy.

Cóż w tym niezwykłego? Otóż niekiedy otrzymujemy zestaw argumentów właśnie w postaci tablicy. Jeśli nie znamy z góry liczby jej elementów, przekazanie argumentów w wywołaniu może się skomplikować. Przykład mieliśmy choćby w rozdziale 4., przy okazji implementowania klas delegujących chybione wywołania do innych klas. Oto uproszczony przykład takiej metody przechwytyjącej:

```
function __call($method, $args) {
    if (method_exists($this->thirdpartyShop, $method)) {
        return $this->thirdpartyShop->$method();
    }
}
```

Powyższa metoda jest wywoływana w obliczu próby wywołania na rzecz obiektu klasy niezdefiniowanej w tej klasie metody. W tym przykładzie owo chybione wywołanie delegujemy do obiektu przechowywanego za pośrednictwem składowej `$thirdpartyShop`. Jeśli w owym obiekcie wykryjemy obecność metody pasującej do argumentu `$method`, wywołujemy ją na rzecz obiektu `$thirdpartyShop`. Zakładamy przy tym, że docelowa metoda nie przyjmuje żadnych argumentów — założenie takie może zaś okazać się chybione. Pisząc kod metody `__call()`, nie mamy przecież możliwości określenia z góry rozmiaru tablicy argumentów `$args`. Gdybyśmy zaś po prostu przekazali tablicę `$args` wprost do metody docelowej, naruszylibyśmy być może składnię jej wywołania — wynikiem może być wiele, a nie tylko jeden (choćby i tablicowy) argument. Problem rozwiązuje właśnie funkcja `call_user_func_array()`:

```
function __call($method, $args) {
    if (method_exists($this->thirdpartyShop, $method)) {
        return call_user_func_array(
            array($this->thirdpartyShop, $method),
            $args);
    }
}
```

Interfejs retrospekcji — Reflection API

Interfejs retrospekcji Reflection API jest dla PHP tym, czym dla Javy jest pakiet `java.lang.reflect`. Interfejs Reflection API składa się z wbudowanych klas umożliwiających badanie metod, składowych i klas. W pewnych aspektach dubluje dostępne już wcześniej funkcje, jak choćby `get_class_vars()`, jest jednak nieporównywalnie bardziej elastyczny i szczegółowy. Do tego uwzględnia najnowsze obiektywne elementy PHP, jak kontrolę widoczności i dostępu do składowych, interfejsy i ich implementacje czy klasy abstrakcyjne — próżno tych udogodnień szukać w starszych funkcjach opisujących cechy klas.

Zaczynamy

Interfejs retrospekcji nie służy wyłącznie do analizy klas. Na przykład klasa `ReflectionFunction` wykorzystywana jest do pozyskiwania informacji o zwykłych funkcjach, a klasa `ReflectionExtension` określa szczegóły rozszerzeń kompilowanych do języka. Wybrane klasy tego rozbudowanego interfejsu wymienia tabela 5.1.

Tabela 5.1. Wybrane klasy interfejsu Reflection API

Klasa	Opis
<code>Reflection</code>	Udostępnia statyczną metodę <code>export()</code> generującą zestawienia informacji o klasach.
<code>ReflectionClass</code>	Informacje i narzędzia badania klas.
<code>ReflectionMethod</code>	Informacje i narzędzia badania metod.
<code>ReflectionParameter</code>	Informacje i narzędzia badania argumentów metod.
<code>ReflectionProperty</code>	Informacje i narzędzia badania składowych.
<code>ReflectionFunction</code>	Informacje i narzędzia badania funkcji.
<code>ReflectionExtension</code>	Informacje o rozszerzeniach PHP.
<code>ReflectionException</code>	Klasa wyjątku.
<code>ReflectionZendExtension</code>	Informacje o rozszerzeniach PHP Zend.

Klasy interfejsu Reflection API dają bezprecedensową możliwość dynamicznego odwoływania się do informacji o obiektach, funkcjach i wyjątkach przetwarzanych w skrypcie.

Z racji możliwości i zakresu zastosowań owego interfejsu należy go preferować wobec realizujących podobne zadania funkcji. Wkrótce Czytelnik przekona się, że interfejs ten jest wprost nieocenionym narzędziem badania klas. Można za jego pomocą generować diagramy na potrzeby dokumentacji albo utrwać informacje o obiektach w bazach danych, czy też wreszcie analizować metody akcesory dostępne w obiekcie celem ustalenia nazw jego składowych. Jeszcze jednym zastosowaniem interfejsu Reflection jest konstruowanie szkieletu wywołań metod w klasach wedle pewnej konwencji nazewnictwa.

Pora zakasać rękawy

Wiemy już, że atrybuty klas można analizować za pośrednictwem zestawu specjalnych funkcji. Wiele z tych funkcji nie spełnia jednak wszystkich naszych wymagań, zwłaszcza w odniesieniu do rozszerzeń obiektowych wprowadzonych w PHP5. Pora więc przyjrzeć się narzędziu, które takich wad nie posiada. Klasa `ReflectionClass` pozwala na pozyskanie informacji o każdym dosłownie aspekcie danej klasy — a działa równie skutecznie wobec klas definiowanych przez użytkownika, jak i wobec klas wbudowanych. Jedynym argumentem wywołania konstruktora klasy `ReflectionClass` jest nazwa klasy wyznaczonej do analizy:

```
$prod_class = new ReflectionClass('CdProduct');
Reflection::export($prod_class);
```

Po utworzeniu obiektu klasy `ReflectionClass` można za pośrednictwem klasy narzędziowej `Reflection` wypisać informacje o klasie `CdProduct` na wyjście skryptu. Klasa `Reflection` udostępnia statyczną metodę `export()`, która formatuje i wypisuje informacje zebrane w obiekcie retrospekcji `Reflection` (ściśle mówiąc, w dowolnym obiekcie dowolnego obiektu implementującym interfejs `Reflection`). Oto fragment wydruku generowanego przez metodę `Reflection::export()`:

```
Class [ <user> class CdProduct extends ShopProduct ] {
  @@ fullshop.php 53-73

  - Constants [0] {
  }

  - Static properties [0] {
  }

  - Static methods [0] {
  }

  - Properties [2] {
    Property [ <default> private $playLength ]
    Property [ <default> protected $price ]
  }

  - Methods [10] {
    Method [ <user, overwrites ShopProduct, ctor> public method __construct ] {
      @@ fullshop.php 56 - 61

      - Parameters [5] {
        Parameter #0 [ <required> $title ]
        Parameter #1 [ <required> $firstName ]
        Parameter #2 [ <required> $mainName ]
        Parameter #3 [ <required> $price ]
        Parameter #4 [ <required> $playLength ]
      }
    }

    Method [ <user> public method getPlayLength ] {
      @@ fullshop.php 63 - 65
    }

    Method [ <user, overwrites ShopProduct, prototype ShopProduct> public method getSummaryLine ] {
      @@ fullshop.php 67 - 71
    }
  }
}
```

Jak widać, metoda `Reflection::export()` daje dostęp do znacznej ilości informacji o klasie. `Reflection::export()` w generowanym zestawieniu uwzględnia każdy niemal aspekt klasy `CdProduct`, w tym informacje o widoczności i dostępie do metod i składowych, o argumentach poszczególnych metod i położeniu kodu każdej metody w pliku kodu definiującego klasę. Szczegółowość informacji jest zdecydowanie wyższa niż w tradycyjnie wykorzystywanej w podobnych zastosowaniach funkcji diagnostycznej `var_dump()`. Wprawdzie funkcja ta wymaga konkretyzacji obiektu, dla którego ma wygenerować zestawienie diagnostyczne, ale mimo to nie dorównuje szczegółowością diagnozy metodzie `Reflection::export()`:

```
$cd = new CdProduct("cd1", "bob", "bobbleson", 4, 50 );
var_dump( $cd );
```

Jako wynik tego programu zobaczymy:

```
object(CdProduct)#1 (6) {
  ["playLength:private"]=>
  int(50)
  ["title:private"]=>
  string(3) "cd1"
  ["producerMainName:private"]=>
  string(9) "bobbleson"
  ["producerFirstName:private"]=>
  string(3) "bob"
  ["price:protected"]=>
  int(4)
  ["discount:private"]=>
  int(0)
}
```

Funkcja `var_dump()` i spokrewniona z nią `print_r()` są niezwykle wygodne, jeśli celem jest ekspozycja danych w skryptach. Jednak w przypadku klas i funkcji interfejs `Reflection API` przynosi diagnostykę i analizę na nowy poziom.

Badanie klasy

Metoda `Reflection::export()` jest znakomitym źródłem informacji diagnostycznych, ale interfejs `Reflection` da się też wykorzystywać w sposób bardziej specjalizowany — za pośrednictwem jego specjalizowanych klas.

Wiemy już, jak konkretyzować obiekt klasy `ReflectionClass`:

```
$prod_class = new ReflectionClass('CdProduct');
```

Możemy teraz spróbować wykorzystać powołany do życia obiekt klasy `ReflectionClass` do dynamicznej analizy klasy `CdProduct`. Jakiego rodzaju jest klasą? Czy da się utworzyć jej egzemplarz? Na pytania te odpowie następująca funkcja:

```
function classData(ReflectionClass $class) {
    $details = "";
    $name = $class->getName();
    if ($class->isUserDefined()) {
        $details .= "$name to klasa definiowana przez użytkownika\n";
    }
    if ($class->isInternal()) {
        $details .= "$name to klasa wbudowana\n";
    }
    if ($class->isInterface()) {
        $details .= "$name definiuje interfejs\n";
    }
    if ($class->isAbstract()) {
        $details .= "$name to klasa abstrakcyjna\n";
    }
    if ($class->isFinal()) {
        $details .= "$name to klasa finalna\n";
    }
    if ($class->isInstantiable()) {
        $details .= "Można tworzyć obiekty klasy $name\n";
    } else {
        $details .= "Nie można tworzyć obiektów klasy $name\n";
    }
    if ( $class->isCloneable() ) {
```

```

        $details .= "Można klonować obiekty klasy $name\n";
    } else {
        $details .= "Nie można klonować obiektów klasy $name\n";
    }

    return $details;
}
$prod_class = new ReflectionClass('CdProduct');
print classData($prod_class);

```

Tworzymy tu obiekt klasy `ReflectionClass` kojarzony z klasą `CdProduct` (której nazwa przekazywana jest w wywołaniu konstruktora klasy `ReflectionClass`). Następnie tak powołany do życia obiekt przekazujemy do funkcji `classData()`, która ilustruje sposób pozyskiwania niektórych informacji o klasie.

Wywoływane w jej wnętrzu metody klasy `ReflectionClass` nie wymagają chyba komentarza — ograniczę się więc do króciutkiego opisu każdej z nich:

- `ReflectionClass::getName()` zwraca nazwę badanej klasy.
- `ReflectionClass::isUserDefined()` zwraca `true`, jeśli badana klasa jest klasą definiowaną przez użytkownika w kodzie skryptu PHP; analogicznie metoda `ReflectionClass::isInternal()` zwraca `true`, jeśli badana klasa jest klasą wbudowaną.
- `ReflectionClass::isAbstract()` sprawdza, czy badana klasa jest klasą abstrakcyjną; bytność klasy jako interfejsu można zaś sprawdzić wywołaniem metody `ReflectionClass::isInterface()`.
- Metoda `ReflectionClass::isInstantiable()` informuje, czy klasa nadaje się do konkretyzacji, czyli czy można stworzyć jej egzemplarze.

Wreszcie metoda `ReflectionClass::isCloneable()` pozwala na określenie, czy obiekty klasy implementują mechanizm klonowania.

Diagnostyka może sięgać nawet do kodu źródłowego klas definiowanych przez użytkownika. Obiekt klasy `ReflectionClass` daje bowiem dostęp do informacji o nazwie pliku definicji klasy, podaje też początkowy i końcowy wiersz kodu źródłowego definicji klasy w tym pliku.

Oto szybki sposób użycia klasy `ReflectionClass` do uzyskania dostępu do źródła klasy:

```

class ReflectionUtil {
    static function getClassSource(ReflectionClass $class) {
        $path = $class->getFileName();
        $lines = @file($path);
        $from = $class->getStartLine();
        $to = $class->getEndLine();
        $len = $to - $from + 1;
        return implode(array_slice($lines, $from - 1, $len));
    }
}

print ReflectionUtil::getClassSource(
    new ReflectionClass('CdProduct'));

```

`ReflectionUtil` to prosta klasa definiująca zaledwie jedną metodę statyczną — `ReflectionUtil::getClassSource()`. Jedynym argumentem jej wywołania jest obiekt klasy `ReflectionClass`, metoda zwraca zaś kod źródłowy wskazanej klasy. Nazwę pliku definicji klasy udostępnia wywołanie metody `ReflectionClass::getFileName()`; zwrócona nazwa jest ścieżką bezwzględną, więc można od razu otworzyć plik kodu. Listę wierszy kodu źródłowego z tego pliku pozyskuje się przez wywołanie funkcji `file()`. Numer pierwszego wiersza definicji klasy określa wywołanie `ReflectionClass::getStartLine()`, numer wiersza końcowego — wywołanie `ReflectionClass::getEndLine()`. Po uzyskaniu tych danych pozostaje już jedynie wyciąć z tablicy interesujące nas wiersze, wywołując funkcję `array_slice()`.

Dla uproszczenia i gwoli zwięzłości w powyższym kodzie pominięto wszelką obsługę błędów. W prawdziwych aplikacjach należałoby oczywiście uzupełnić kod o stosowną kontrolę argumentów i wartości zwracanych.

Badanie metod

Tak jak klasa `ReflectionClass` pośredniczy w analizie klas, tak obiekt klasy `ReflectionMethod` pozwala na pozyskiwanie informacji o metodach klas.

Obiekt klasy `ReflectionMethod` pozyskuje się na dwa sposoby: można bowiem albo pozyskać tablicę obiektów `ReflectionMethod` zwracaną przez wywołanie `ReflectionClass::getMethods()`, albo — jeśli interesuje nas pojedyncza metoda — skorzystać z wywołania `ReflectionClass::getMethod()` przyjmującego nazwę metody i zwracającego opisujący ją obiekt `ReflectionMethod`.

Poniżej prezentowany jest sposób pierwszy:

```
$prod_class = new ReflectionClass('CdProduct');
$methods = $prod_class->getMethods();

foreach($methods as $method) {
    print methodData($method);
    print "\n---\n";
}

function methodData(ReflectionMethod $method) {
    $details = "";
    $name = $method->getName();
    if ($method->isUserDefined()) {
        $details .= "$name to metoda definiowana przez użytkownika\n";
    }
    if ($method->isInternal()) {
        $details .= "$name to metoda wbudowana\n";
    }
    if ($method->isAbstract()) {
        $details .= "$name to metoda abstrakcyjna\n";
    }
    if ($method->isPublic()) {
        $details .= "$name jest metodą publiczną\n";
    }
    if ($method->isProtected()) {
        $details .= "$name jest metodą zabezpieczoną\n";
    }
    if ($method->isPrivate()) {
        $details .= "$name jest metodą prywatną\n";
    }
    if ($method->isStatic()) {
        $details .= "$name to metoda statyczna\n";
    }
    if ($method->isFinal()) {
        $details .= "$name to metoda finalna\n";
    }
    if ($method->isConstructor()) {
        $details .= "$name to konstruktor\n";
    }
    if ($method->isReturnsReference()) {
        $details .= "$name zwraca referencję (nie wartość)\n";
    }
    return $details;
}
```

Powyższy kod za pośrednictwem wywołania `ReflectionClass::getMethods()` pozyskuje tablicę obiektów opisujących metody klasy `CdProduct`, a następnie dokonuje przeglądu zawartości tablicy, wywołując dla każdego zawartego w niej obiektu `ReflectionMethod` funkcję `methodData()`.

Poszczególne wywołania w ciele funkcji `methodData()` nie wymagają raczej komentarza — funkcja sprawdza, czy bieżąca metoda jest definiowana przez użytkownika, czy może jest metodą wbudowaną, czy jest abstrakcyjna, czy jest publiczna, chroniona czy prywatna, czy jest statyczna, a może finalna. Dodatkowo funkcja sprawdza, czy metoda nie jest przypadkiem konstruktorem i czy zwraca wartości, czy referencje.

Słowo komentarza: metoda `ReflectionMethod::returnsReference()` nie zwraca `true`, jeśli badana metoda zwraca obiekty, mimo że obiekty są w PHP5 przekazywane przez referencje, a nie wartości. Wywołanie `ReflectionMethod::returnsReference()` zwraca `true` jedynie wtedy, kiedy dana metoda została jawnie zadeklarowana jako zwracająca referencje (deklaracja taka polega na poprzedzeniu nazwy metody znakiem `&`).

Jak można się spodziewać, i tym razem możemy spróbować odwołać się do kodu źródłowego metody, stosując zresztą technikę bardzo przypominającą tę stosowaną dla całych klas:

```
class ReflectionUtil {
    static function getMethodSource(ReflectionMethod $method) {
        $path = $method->getFileName();
        $lines = @file($path);
        $from = $method->getStartLine();
        $to = $method->getEndLine();
        $len = $to - $from + 1;
        return implode(array_slice($lines, $from - 1, $len));
    }
}
$class = new ReflectionClass('CdProduct');
$method = $class->getMethod('getSummaryLine');
print ReflectionUtil::getMethodSource($method);
```

Wyodrębnienie kodu źródłowego jest bardzo proste, ponieważ klasa `ReflectionMethod` udostępnia komplet potrzebnych do tego informacji za pośrednictwem metod `getFileName()`, `getStartLine()` i `getEndLine()`.

Badanie argumentów metod

W PHP5 sygnatury metod mogą ograniczać typy argumentów obiektowych, przydatna więc byłaby możliwość analizowania tych deklaracji. Interfejs `Reflection API` udostępnia do tego celu klasę `ReflectionParameter`. Aby pozyskać obiekt tej klasy, należy odwołać się do obiektu `ReflectionMethod`, wywołując jego metodę `ReflectionMethod::getParameters()` — zwraca ona tablicę obiektów klasy `ReflectionParameter`.

Obiekt klasy `ReflectionParameter` może dawać wywołującemu informacje o nazwie argumentu, o tym, czy argument jest przekazywany przez referencję (czyli czy został zadeklarowany w sygnaturze metody ze znakiem `&`), jak również o wymuszanej deklaracji klasie argumentu i o akceptacji w jego miejsce wartości pustej.

Oto jedno z zastosowań metod klasy `ReflectionParameter`:

```
$prod_class = new ReflectionClass(CdProduct);
$method = $prod_class->getMethod("__construct");
$params = $method->getParameters();

foreach ($params as $param) {
    print argData($param)."\n";
}

function argData(ReflectionParameter $arg) {
    $details = "";
    $declaringClass = $arg->getDeclaringClass();
    $name = $arg->getName();
    $class = $arg->getClass();
    $position = $arg->getPosition();
    $details .= "$$name na pozycji $position\n";
    if (!empty($class)) {
        $classname = $class->getName();
        $details .= "$$name musi być obiektem klasy $classname\n";
    }
}
```

```

    }

    if ($arg->isPassedByReference()) {
        $details .= "\$$name jest przekazywany przez referencję\n";
    }

    if ( $arg->isDefaultValueAvailable() ) {
        $def = $arg->getDefaultValue();
        $details .= "\$$name has default: $def\n";
    }
    return $details;
}

```

W powyższym kodzie metoda `ReflectionClass::getMethod()` służy nam do pozyskania obiektu klasy `ReflectionMethod` opisującego wybraną metodę. Następnie za pośrednictwem zainicjowanego na rzecz tego obiektu wywołania metody `getParameters()` pobierana jest tablica obiektów `ReflectionParameter`. Są one kolejno wyodrębniane z tablicy i przekazywane do funkcji `argData()`.

Ta z kolei w pierwszej kolejności sprawdza przez wywołanie `ReflectionParameter::getName()` nazwę parametru. Wywoływana później metoda `getClass()` zwraca obiekt klasy `ReflectionClass` opisujący wymuszaną w sygnaturze metody klasę argumentu. Wreszcie kod sprawdza (za pomocą `isPassedByReference()`), czy argument jest dany referencją i czy posiada wartość domyślną, którą ewentualnie dopisuje do ciągu zwracanego.

Korzystanie z retrospekcji

Uzbrojeni w umiejętność korzystania (przynajmniej w podstawowym zakresie) z interfejsu Reflection API możemy zaprząć go do pracy.

Załóżmy, że tworzymy klasę, która w sposób dynamiczny wywołuje obiekty klasy `Module`. Chodzi o to, aby kod mógł akceptować rozszerzenia i wtyczki autorstwa osób trzecich, możliwe do wywoływania z aplikacji bez potrzeby ciągłego zmieniania jej kodu. W tym celu można by zdefiniować w interfejsie albo w klasie bazowej `Module` metodę `execute()`, zmuszając wszystkie klasy pochodne `Module` do implementacji tej metody. Zakładamy też, że użytkownicy systemu będą mieć możliwość prowadzenia listy dostępnych modułów w zewnętrznym pliku konfiguracyjnym zapisanym w formacie XML. System powinien na podstawie tej listy zgromadzić odpowiednią liczbę obiektów `Module` i wywołać na rzecz każdego z nich metodę `execute()`.

Jak jednak obsłużyć sytuację, w której każdy z modułów (obektów `Module`) wymaga do wykonania swoich zadań *odmiennego* zestawu informacji? W takim przypadku stosowne klucze i wartości składowych powinny zostać zapisane w pliku XML, a twórca każdego obiektu `Module` powinien udostępnić zestaw stosownych akcesorów. Na takim fundamencie musimy sami już zapewnić prawidłowe wywołania akcesorów dla odpowiednich składowych.

Oto pierwszy zarys interfejsu `Module` i kilku implementujących go klas:

```

class Person {
    public $name;
    function __construct($name) {
        $this->name = $name;
    }
}

interface Module {
    function execute();
}

class FtpModule implements Module {
    function setHost($host) {
        print "FtpModule::setHost(): $host\n";
    }

    function setUser($user) {

```

```

    print "FtpModule::setUser(): $user\n";
}

function execute() {
    // właściwe operacje obiektu
}

}

class PersonModule implements Module {
    function setPerson(Person $person) {
        print "PersonModule::setPerson(): {$person->name}\n";
    }

    function execute() {
        // właściwe operacje obiektu
    }
}

```

Prezentowane tu klasy `FtpModule` i `PersonModule` udostępniają (na razie puste) implementacje metody `execute()`. Ponadto każda klasa implementuje pewne metody akcesory, których działanie ogranicza się chwilowo do sygnalizowania faktu wywołania. W naszym systemie przyjęliśmy konwencję, że wszystkie akcesory ustawiające przyjmują dokładnie jeden argument, którym jest albo ciąg znaków, albo obiekt dający się konkretyzować na podstawie pojedynczego ciągu znaków. Metoda `PersonModule::setPerson()` oczekuje przekazania obiektu klasy `Person`, więc uzupełniliśmy przykład o definicję tej klasy.

Aby zacząć pracę z wykorzystaniem obiektów klas `PersonModule` i `FtpModule`, musimy jeszcze utworzyć klasę, która będzie te obiekty wywoływała. Nazwiemy ją `ModuleRunner`. Informacje odczytane z pliku konfiguracyjnego XML będą w niej reprezentowane wielowymiarową tablicą indeksowaną nazwą modułu. Oto kod klasy:

```

class ModuleRunner {
    private $configData
        = array(
            "PersonModule" => array('person'=>'bob'),
            "FtpModule"    => array('host' => 'przyklad.com',
                                   'user' => 'anon')
        );
    private $modules = array();
    //...
}

```

Składowa `ModuleRunner::$configData` przechowuje odwołania do dwóch klas implementujących interfejs `Module`. Każde takie odwołanie reprezentowane jest podtablicą gromadzącą zestaw składowych. Za tworzenie obiektów `Module` odpowiedzialna jest metoda `init()` klasy `ModuleRunner` zdefiniowana jak poniżej:

```

class ModuleRunner {
    // ...

    function init() {
        $interface = new ReflectionClass('Module');
        foreach($this->configData as $modulename => $params) {
            $module_class = new ReflectionClass($modulename);
            if (!$module_class->isSubclassOf($interface)) {
                throw new Exception("nieznany typ modułu: $modulename");
            }
            $module = $module_class->newInstance();
            foreach ($module_class->getMethods() as $method) {
                $this->handleMethod($module, $method, $params);
                // metoda handleMethod() prezentowana na następnym listingu...
            }
            array_push($this->modules, $module);
        }
    }
}

```

```

    }
    //...
}

$test = new ModuleRunner();
$test->init();

```

Metoda `init()` przegląda tablicę `ModuleRunner::$configData` i dla każdego jej elementu opisującego moduł podejmuje próbę utworzenia obiektu klasy `ReflectionClass`. Jeśli konstruktor tej klasy zostanie wywołany z nazwą klasy nieistniejącej, generowany jest wyjątek — w praktycznych zastosowaniach trzeba by uzupełnić kod o obsługę tegoż wyjątku. Dalej za pośrednictwem wywołania `ReflectionClass::isSubclassOf()` sprawdzana jest przynależność klasy modułu do typu `Module`.

Przed próbą wywołania metody `execute()` każdego z modułów należy najpierw skonkretyzować ich obiekty. To zadanie składamy na barki metody `ReflectionClass::newInstance()`. Metoda ta przyjmuje dowolną liczbę argumentów, które przekazuje do konstruktora odpowiedniej klasy (dla której skonkretyzowano uprzednio obiekt klasy `ReflectionClass`). Jeśli wszystko się powiedzie, wywołanie zwróci referencję nowego obiektu (w kodzie produkcyjnym należałoby zadbać o większą zachowawczość — wypadałoby choćby sprawdzić przed konkretyzacją obiektów `Module`, czy ich konstruktory faktycznie obchodzą się bez argumentów).

Inicjowane potem wywołanie `ReflectionClass::getMethods()` zwraca tablicę obiektów `ReflectionMethod` reprezentujących wszystkie dostępne w danej klasie metody. Dla każdego elementu tej tablicy kod ten wywołuje metodę `ModuleRunner::handleMethod()`, przekazuje do niej egzemplarz obiektu `Module`, obiekt `ReflectionMethod` oraz tablicę składowych skojarzonych z obiektem `Module`. Metoda `handleMethod()` weryfikuje dostępność i ostatecznie wywołuje odpowiednie metody akcesory obiektu `Module`.

```

class ModuleRunner {
    //...
    function handleMethod(Module $module, ReflectionMethod $method, $params) {
        $name = $method->getName();
        $args = $method->getParameters();

        if (count($args) != 1 ||
            substr($name, 0, 3) != "set") {
            return false;
        }

        $property = strtolower(substr($name, 3));
        if (!isset($params[$property])) {
            return false;
        }

        $arg_class = $args[0]->getClass();
        if (empty($arg_class)) {
            $method->invoke($module, $params[$property]);
        } else {
            $method->invoke($module,
                $arg_class->newInstance($params[$property]));
        }
    }
}

```

Metoda `handleMethod()` sprawdza najpierw, czy wytypowana do wywołania metoda jest aby odpowiednim akcesorem ustawiającym. Akcesory takie rozpoznawane są tu na podstawie wzorca nazwy, która musi rozpoczynać się od ciągu `set` i zawierać nazwę składowej; poza tym musi deklarować dokładnie jeden argument.

Jeśli argument się zgadza, kod wyodrębnia z nazwy metody nazwę składowej, usuwając z nazwy przedrostek `set` i konwertując resztę na ciąg zawierający wyłącznie małe litery. Wynik konwersji jest następnie wykorzystywany w analizie tablicy argumentów `$params`. Tablica ta zawiera przekazane przez użytkownika składowe skojarzone z obiektem `Module`. Jeśli tablica `$params` nie zawiera szukanej składowej, kod zwraca `false`.

Jeśli wyodrębniona z nazwy akcesora nazwa składowej pasuje do elementu tablicy `$params`, możemy pójść dalej i wywołać właściwy akcesor. Wcześniej jednak należy sprawdzić typ pierwszego (i jedyne) argumentu wywołania akcesora ustawiającego. Informację tę zwraca metoda `ReflectionParameter::getClass()`. Jeśli wywołanie zwróci wartość pustą, akcesor oczekuje przekazania wartości elementarnej — w przeciwnym razie wymaga przekazania obiektu.

Wywołanie metody akcesora wymaga pośrednictwa nieomawianej jeszcze metody klasy `ReflectionMethod::invoke()`, wymagającej przekazania obiektu i dowolnej liczby argumentów, które przekazywane są dalej do metody docelowej. Kiedy przekazany obiekt nie pasuje do metody, wywołanie `ReflectionMethod::invoke()` zgłasza wyjątek. Metoda `invoke()` wywoływana jest na dwa sposoby: jeśli akcesor nie wymaga przekazania obiektu konkretnego typu, wywołanie `invoke()` jest inicjowane z podsuniętym przez użytkownika ciągiem znaków. Jeśli metoda wymaga argumentu obiektowego, ów ciąg jest wykorzystywany do konkretyzacji obiektu odpowiedniego typu, który jest następnie przekazywany do `invoke()`.

Przykład ten bazuje na założeniu, że wymagany obiekt da się konkretyzować wywołaniem konstruktora z pojedynczym argumentem w postaci ciągu znaków. Najlepiej oczywiście byłoby jednak sprawdzić wymagania konstruktora jeszcze przed wywołaniem `ReflectionClass::newInstance()`.

W miarę postępu wykonywania metody `ModuleRunner::init()` obiekt klasy `ModuleRunner` wypełnia się obiektami `Module` zawierającymi stosowne dane. Klasa mogłaby zostać teraz uzupełniona o metodę przeglądającą owe obiekty i inicjującą na ich rzecz wywołanie metody `execute()`.

Podsumowanie

W rozdziale zajmowaliśmy się narzędziami i technikami pomocnymi w zarządzaniu bibliotekami i klasami. Czytelnik mógł poznać nowy w PHP mechanizm przestrzeni nazw. Wyjaśniono, jak organizować kod, uciekając się do odpowiednich ustawień ścieżek wyszukiwania, odpowiedniej (tu zaczerpniętej z repozytorium PEAR) konwencji nazewnictwa i cech systemu plików. Przyjrzelśmy się ponadto funkcjom dynamicznej analizy klas i obiektów, a następnie realizującym podobne zadania elementom interfejsu retrospekcji `Reflection API`. Na koniec zaś na bazie klas hierarchii `Reflection` skonstruowaliśmy prosty przykład ilustrujący potencjał tkwiący w interfejsie dynamicznej analizy klas.

Skorowidz

A

- abstrakcja Expression, 204
- abstrakcyjna
 - klasa bazowa, 160
 - klasa dekoratora, 196
- abstrakcyjny
 - produkt, 175
 - typ, 164
 - wytwórca, 175
- agregacja, 137
- akcesor, 52, 118–120
- alias, 98
- aliasy metod cech typowych, 67
- analizator leksykalny, 461, 468
- aplikacje, 232
- archiwum JAR, 398
- argumenty metod, 117
- asemblery obiektów dziedziny, 316
- asercje, 388
- atrapy, 389, 390
- atributy, 135
 - elementu copy, 422
 - elementu input, 423
 - elementu patternset, 418
 - elementu project, 408
 - elementu target, 412
- automatyczna kompilacja, 454
- automatyczne
 - scalanie wersji, 364
 - wczytywanie kodu, 103
- automatyzacja
 - instalacji, 405
 - kompilacji, 444
 - konfiguracji obiektów, 31

B

- badanie
 - argumentów metod, 117
 - klas, 107, 114
 - metod, 108, 116
 - obiektów, 107
 - relacji dziedziczenia, 110
 - składowych, 110
 - wywołań metod, 110
- baza danych MySQL, 157
- biblioteka PHP SPL, 110
- bieżąca przestrzeń nazw, 98
- blok
 - finally, 80
 - try-catch, 79
- błąd, 73, 90
 - krytyczny, 81
 - wykonania, 66, 70, 81
- błędy testów, 444
- budowanie pakietu, 434
- buforowanie, 247

C

- cecha typowa
 - IdentityTrait, 64
 - PriceUtilities, 64
- cechy typowe, traits, 62
 - z metodami statycznymi, 68
- CI, Continuous Integration, 325, 426, 454
 - deklaracji przestrzeni nazw, 99
 - metody, 35
- ciąg znaków, 90
- ciągła integracja, CI, 325, 426, 454
 - budowanie pakietu, 434
 - dokumentacja, 430
 - kontrola wersji, 427

- Phing, 428
- przygotowanie projektu, 427
- serwer Jenkins, 436
- standardy kodowania, 433
- testy jednostkowe, 429

D

- definiowanie
 - cechy typowej, 63
 - destruktorów, 86
 - kanału, 345
 - metody abstrakcyjnej, 69
 - składowych, 33
- deklaracja use, 66
- dekorator, 192
- delegowanie, 448
- destruktor, 86
- diagram
 - klas, 134, 152, 186, 193, 208
 - sekwencji, 139, 252
- dobrze praktyki, 319
- dodawanie
 - katalogu, 374
 - pakietu do kanału, 346
 - pliku, 373
- dokumentacja, 323, 349, 430, 453
- dokumentacja domyślna, 352
- dokumentowanie
 - klas, 354
 - metod, 357, 358
 - plików, 354
 - przestrzeni nazw, 358
 - składowych, 355
- dołączanie funkcji, 224
- domknięcia, 91
- dopełnienia, closures, 93

dostęp do
 bazy MySQL, 157
 klasy, 50
 metody, 63
 powłoki zdalnej, 366
 składowej, 33, 43, 68

dostępność
 metod cech typowych, 70
 obiektu PDO, 58

drzewo dziedziczenia, 193, 210

duplikacja kodu, 222

dynamiczne ładowanie kodu, 106

dynamicznie konstruowany ciąg znaków, 110

dyrektywa `include_path`, 102

dystrybucja pakietu, 344

dziedziczenie, 42, 48, 110, 152, 194, 451

E

EBNF, 203

elastyczne zapytania, 279

elastyczność
 obiektów, 183
 systemu, 164

element
 channel, 343
 contains, 343
 contents, 339
 copy, 421, 422
 delete, 423
 dir, 339
 echo, 410, 420
 exec, 435
 file, 339
 fileset, 416
 filterchain, 418
 input, 423
 install, 344
 installconditions, 344
 lead, 339
 patternset, 417
 phprelease, 343
 project, 408, 410
 property, 410
 QuickAddVenue, 256
 required, 342
 stability, 339
 status, 257
 target, 409, 412
 token, 419
 uri, 338
 user, 339
 view, 256

elementy leksykalne, tokens, 461

estetyka kodu, 449

etykietowanie wersji, 375

F

falszywe obiekty, 390

fasada, 197

finalizacja obsługi wyjątków, 79

format
 BloggsCal, 175, 178
 MegaCal, 173, 177
 PEAR, 434
 pliku, 127
 portlandzki, 146
 wzorca, 146

framework, 19

PHPUnit, 384

SUnit, 384

funkcja
 call_user_func_array(), 111
 class_exists(), 106
 fopen(), 102
 get_class(), 107
 get_class_methods(), 108, 109
 get_include_path(), 102
 get_parent_class(), 110
 getProduct(), 107
 include_once(), 100
 is_callable(), 109
 is_int(), 42
 is_subclass_of(), 110
 method_exists(), 109
 print_r(), 283
 require(), 100
 require_once(), 100
 spl_autoload(), 103
 spl_autoload_register(), 103
 var_dump(), 113

funkcje
 anonimowe, 91
 diagnostyczne, 329
 kontroli typów, 38
 ładujące, 103
 pomocnicze, 105

G

generator, 288

generowanie
 dokumentacji, 349, 350
 obiektów, 163
 Abstract Factory, 174
 Factory Method, 170, 172
 przez klonowanie, 178
 Singleton, 167

Git, 363

globalna przestrzeń nazw, 100

gra Civilisation, 184

gramatyka języka MarkLogic, 202

H

hermetyzacja, encapsulation, 123, 131, 157, 448

hermetyzacja algorytmów, 211

hierarchia
 Command, 228, 245, 255, 261
 dziedziczenia, 154
 Expression, 203
 klas, 129, 153
 klas kontrolera strony, 267
 klas poleceń, 245
 Marker, 212
 polimorficzna, 164
 przestrzeni nazw, 357
 Question, 211
 Registry, 239
 Unit, 189

Hunt Wes, 15

I

identyfikator URI, 338

identyfikowanie algorytmów, 201

imitacje, 389

implementacja, 160
 Abstract Factory, 177
 interfejsu, 62
 komunikacji, 173
 metody abstrakcyjnej, 60
 wzorca Observer, 219

informacje
 o klasie, 113
 o trendach, 443

inspekcja kodu, 325

instalacja pakietu Phing, 406

instalator
 PEAR, 342, 348, 405
 Pyrus, 328

instalowanie
 automatyczne, 405
 Jenkinsa, 436
 pakietu, 329
 Phinga, 428
 programu phpDocumentor, 350
 projektu, 439
 rozszerzeń Jenkinsa, 438

instrukcja
 require(), 100
 require_once(), 100

interfejs, 61, 160
 Chargeable, 61
 do kodu proceduralnego, 199
 Iterator, 284, 288
 kaskadow, fluent interface, 308
 Observable, 216, 217, 220
 Reflection, 293, 385
 retrospekcji, 112
 warstwy danych, 279
 WWW kanału, 347
 interpretatory plików kompilacji,
 25
 iterator, 288
 izolacja od warstwy danych, 275
 izolowanie implementacji, 158

J

jednostka pracy, 297, 318
 język
 MarkLogic, 202, 203
 UML, 133–141, 152

K

kanał PEAR, 331, 345
 katalog
 commands, 230
 główny projektu, 341
 klasa, 31
 AddressManager, 39
 AddVenue, 257, 262
 AddVenueController, 266
 AppConfig, 182
 AppController, 255, 261
 ApplicationHelper, 235, 246, 257
 ApplicationRegistry, 243, 253
 Archer, 187
 ArmyVisitor, 223
 BloggsCommsManager, 176
 Collection, 287, 305
 Command, 226, 255, 262
 CommandFactory, 228
 CommsManager, 170–177
 CompositeUnit, 190
 Conf, 74
 ConfException, 77
 Controller, 245
 ControllerMap, 257
 CostStrategy, 156
 Debug, 98
 DeferredEventCollection, 302
 DomainObject, 71, 72, 73, 289
 EqualsExpression, 207
 Exception, 75

Expression, 204
 FileException, 77
 FrontController, 255
 IdentityObject, 309
 kontrolera aplikacji, 258
 Lesson, 153, 155
 Lister, 99
 LiteralExpression, 205
 Login, 214
 LoginObserver, 217
 Mapper, 280, 292, 296, 300
 Marker, 212
 MDB2, 158
 ModuleRunner, 119
 NastyBoss, 164
 Notifier, 159
 ObjectWatcher, 295, 297
 Observable, 217
 OperatorExpression, 206
 PageController, 265, 267
 ParamHandler, 125
 Parser, 468
 PDO, 57
 PEAR_Error, 334
 PEAR_Exception, 336
 PersistenceFactory, 306
 PHPUnit_Framework_TestCase,
 385
 Plains, 192
 Preferences, 167
 ProcessRequest, 193, 195
 Question, 209
 ReflectionClass, 112–116
 RegistrationMgr, 159
 Registry, 236
 Request, 250
 SelectionFactory, 314
 ShopProduct, 36, 40–44, 57, 166
 ShopProductWriter, 52, 59
 Space, 277, 300
 SpaceMapper, 301
 TaxCollectorVisitor, 225
 TestCase, 389
 Tile, 192
 Transaction Script, 272
 Unit, 184, 189, 221
 UserStore, 392
 Validate, 444
 Venue, 263, 277, 291
 VenueCollection, 286
 VenueManager, 273
 VenueUpdateFactory, 314
 XML_Feed_Parser, 336
 XML_RPC_Server, 101

klasy
 abstrakcyjne, 59, 60, 152, 239
 analizatorów, 474
 bazowe, 152
 dekoracji, 195
 finalne, 80
 hierarchii, 52
 interfejsu Reflection API, 112
 modelu dziedziny, 275
 odwzorowania, 281
 pochodne, 152
 pomocnicze, 105
 specjalizowane, 126
 symboli końcowych, 205
 warstwy trwałości, 317
 włączające, 68
 wytwórców i produktów, 172
 wzorca Domain Object
 Factory, 304
 wzorca Interpreter, 203
 wzorca Observer, 218

klauzula
 catch, 76, 79
 extends, 62
 finally, 26, 80
 implements, 62
 try, 76
 use, 93
 warunkowa, 416
 WHERE, 308

client Git, 364
 klonowanie
 obiektów, 55, 181
 repozytorium, 369
 klucz publiczny, 439

kod
 ortogonalny, 128
 proceduralny, 197, 199
 produkcyjny, 241
 zewnętrzny, 106
 kolekcja
 Collection, 305
 SpaceCollection, 290, 292
 kolekcje obiektów, 279
 kolizja nazw, 68
 komentarze DocBlock, 352, 359
 kompilacja, 406–408
 operacje, 420
 typy, 416
 właściwości, 410
 kompilacje automatyczne, 444
 komponent Selenium Server, 398
 kompozycja, 137, 151, 157, 451

konfigurowanie
 kanału PEAR, 345
 klucza publicznego, 439
 raportów, 441–443
 repozytorium kontroli wersji, 440
 serwera Git, 365
 zadań Phing, 441
 konflikt procesów, 243
 konsola Output, 442
 konstruktor, 36, 48
 konstruowanie
 interpreterów minijęzyków, 201
 struktur, 220
 kontrola
 tożsamości, 295
 typu, 42
 wersji, 363, 373, 427
 kontroler
 aplikacji, 254, 258
 fasady, 245, 248, 252
 strony, 264, 265
 konwerter formatów, 175
 kopia powierzchniowa, 89
 kopie obiektów, 87

Ł

ładowanie klas, 103
 łączenie cech z interfejsami, 64

M

mapa tożsamości, 294, 318
 mechanizm MarkLogic, 213
 menu dokumentacji, 351
 metoda, 35
 __call(), 84–87
 __clone(), 88, 181
 __construct(), 37, 74, 82
 __destruct(), 86, 87
 __get(), 82–86
 __isset(), 83
 __set(), 83, 85
 __sleep(), 244
 __toString(), 90
 __unset(), 84
 __wakeup(), 244
 accept(), 221, 225
 addChargeableItem, 62
 addDirty(), 298
 addParam(), 126
 addUnit(), 185, 187
 attach(), 216
 bombardStrength(), 185
 buildStatement(), 313

calculateTax(), 62, 63
 create(), 71
 detach(), 216
 die(), 80
 doCreateObject(), 302
 doExecute(), 262, 263
 doInterpret(), 206
 execute(), 118, 227, 252
 exit(), 80
 find(), 282, 284
 findByVenue(), 292
 finder(), 300
 generateId(), 64
 get(), 74, 242
 getApptEncoder(), 173
 getCommand(), 261
 getComposite(), 189
 getContactEncoder(), 177
 getErrorData(), 336
 getFromMap(), 296
 getHeaderText(), 171
 getInstance(), 126, 166
 getNotifier(), 159
 getOptions(), 247
 getPrice(), 51
 getProducer(), 45
 getProperty(), 168, 269
 getResource(), 260
 getSummaryLine(), 45, 48
 handleLogin(), 214
 handleMethod(), 120
 init(), 119, 120, 246
 insert(), 282
 instance(), 238
 interpret(), 206
 make(), 178
 Mapper::findAll(), 291
 mark(), 213
 notify(), 216
 outputAddresses(), 38
 prepareStatement(), 273
 process(), 196, 229
 read()., 127
 recruit(), 166
 Reflection::export(), 113
 removeUnit(), 187
 sale(), 91
 scan(), 469
 set(), 242
 setDiscount(), 51
 setProperty(), 168
 statuses(), 262
 targetClass(), 296
 visit(), 222
 write(), 41, 45, 77, 127

metody
 abstrakcyjne, 59, 126
 abstrakcyjne cechy typowej, 69
 asercji, 385
 cech typowych, 70
 chronione, 50
 destrukcji obiektów, 55
 finalne, 80
 klasy WebDriverBy, 401
 konstrukcji, 31
 konstrukcji obiektu, 36
 prywatne, 50
 przechwytyjące, 55, 81, 85
 przesłonięte, 50
 publiczne, 50
 publiczne klasy wyjątku, 75
 statyczne, 55, 68, 71, 239
 wytwórcze, 170, 178, 401
 wytwórcze obiektów
 dopasowań, 391
 minijęzyki, 201
 model dziedziny, 277
 montowanie dokumentu
 kompilacji, 407

N

narzędzia, 25, 452–455
 narzędzia obiektowe, 95
 narzędzie
 Ant, 25, 406
 Phing, 25, 406
 phpDocumentor, 349
 PHPUnit, 381
 Pirum, 345
 nawiasy klamrowe, 35, 99
 nazwa
 akcesora, 84
 klasy, 108
 konstruktora, 50
 nazwy
 metod, 65
 wzorców, 145
 nieznaną klasą, 105
 notacja EBNF, 474
 notki, 139

O

obiekt, 24, 27, 32, 123, 447
 PDO, 272, 282
 Request, 237, 239
 obiekt-kompozyt, 186

- obiekty
 - brudne, 298
 - danych, 57
 - poleceń, 226
 - rejestr, 237
 - tożsamości, 306
 - weryfikujące, 383
 - obserwator, 215
 - obserwowanie
 - interfejsu, 215
 - obiektów, 279
 - obsługa
 - błędów, 73, 78, 255
 - błędów w PEAR, 334
 - obiektów, 55
 - PEAR, 328
 - sesji, 238
 - wierszy, 284
 - wyjątków, 79
 - żądania HTTP, 248
 - odczyt danych, 318
 - w formacie XML, 125
 - z pliku, 124
 - odnośniki w dokumentacji, 359
 - odpowiedzialność, 127, 129
 - odwzorowanie danych, 280, 318
 - ograniczenia, 134
 - we wzorcu Composite, 191
 - opcje instalacji, 330
 - operacja, 135
 - copy, 420
 - delete, 423
 - echo, 420
 - input, 422
 - operator
 - ::, 48
 - ==, 88
 - ===, 88
 - as, 67, 68
 - dostępu do składowej, 33, 35
 - instanceof, 107, 191
 - insteadof, 66, 68
 - new, 37
 - opis klasy, 355
 - opóźnione ładowanie, 301
 - oprogramowanie kontroli wersji, 373
 - ortogonalność, 128
 - ortogonalność projektu, 214
 - osłabianie sprzężenia, 158, 448
- P**
- pakiet, 95
 - Auth, 321
 - Benchmark, 320
 - Cache_Lite, 320
 - Config, 321, 334
 - File_HtAccess, 320
 - Log, 330
 - Mail_Mime, 321
 - MDB2, 157, 320
 - Phing, 406
 - reflect, 112
 - RPC, 101
 - util, 97
 - XML_RSS, 320
 - pakiety
 - JAR, 329
 - PEAR, 74, 101, 320, 334, 406
 - własne, 337
 - para klucz – wartość, 125
 - parametr
 - cmd, 256
 - include_path, 102
 - parsowanie, 125
 - PDO, PHP Data Object, 157
 - PEAR, 25, 101, 327
 - pętla foreach, 288
 - PHP/FI, 27
 - PHP3, 27
 - PHP4, 28
 - PHP5, 29, 32
 - PHPUnit, 381, 384, 432
 - pierwsza kompilacja, 441
 - pisanie testu, 400
 - plik
 - build.xml, 407
 - Config.php, 334
 - Dialekt.php, 342
 - FeedbackCommand.php, 230
 - httpd.conf, 102
 - konfiguracji kompilacji, 429
 - konfiguracyjny, 38, 255
 - main.php, 252
 - makefile, 406
 - package.xml, 337, 344
 - php.ini, 102
 - README, 424
 - Server.php, 101
 - User.php, 370, 371
 - venues.php, 267
 - pliki
 - .htaccess, 102
 - .inc, 103
 - .php, 103
 - biblioteczne, 102
 - projekt, 368
 - podłączanie obserwatorów, 218
 - pola filtrowania, 311
 - polecenia, 251
 - polecenie
 - channel-discover, 347
 - channel-info, 331, 332
 - commit, 368
 - config-show, 328, 329
 - get, 329
 - git add, 371
 - git branch, 368
 - git checkout, 377
 - git clone, 369
 - git merge, 378
 - git status, 370
 - phing, 409, 411
 - set, 340
 - polimorfizm, 123, 129, 160
 - połączenie z serwerem Selenium, 399
 - pomocnik widoku, 268, 269
 - pouczenie, hint, 41
 - powiadomienia, 215
 - powiązania, 136
 - powiązanie procedur, 127
 - powielanie kodu, 188
 - powłoka git-shell, 366
 - pozyskiwanie
 - kolekcji, 290, 292
 - obiektu polecenia, 261
 - obiektu widoku, 261
 - odwzorowań, 290
 - późne wiązanie statyczne, 71
 - produkt, 170, 176
 - program
 - Phing, 405
 - phpDocumentor, 349
 - programowanie
 - obiektowe, 21, 124, 183
 - proceduralne, 124
 - projekt php-webdriver, 399
 - projektowanie, 21, 123
 - projektowanie obiektowe, 123
 - prototyp, 178
 - przechwytywanie chybionych wywołań, 81
 - przegląd architektury, 231
 - przekazywanie obiektu, 61
 - przemądrzałe klasy, 133
 - przesłanie
 - mechanizmu ładowania, 104
 - metod, 221
 - przestrzenie nazw, 95–97

przetwarzanie
 pliku konfiguracji, 257
 żądania, 245
 pseudozmienna \$this, 36
 Pyrus, 328

R

raporty, 442
 realizacja zadań, 201
 refaktoryzacja, 23
 Reflection API, 112
 reguły projektowe, 129
 rejestr, 236, 238
 rejestracja, 215
 relacje
 dziedziczenia, 110, 136
 implementacji, 136
 użycia, 138
 repozytorium
 dla użytkownika lokalnego, 365
 Git, 369
 kontroli wersji, 440
 PEAR, 25, 101, 320–327, 333
 zdalne, 365
 reprezentacja obiektu, 90
 reprezentowanie klas, 134
 retrospekcja, 112, 118
 rezygnacja, 215
 rodziny produktów, 175
 role plików pakietu, 340
 rozgałęzianie projektu, 375
 rozprzęganie, 157
 rozszerzalność języka UML, 135
 rozszerzanie
 klasy ProcessRequest, 193
 klasy Unit, 190
 rozszerzenia Jenkinsa, 438
 rozszerzenie
 .tgz, 338
 apc, 242
 PDO, 157
 SimpleXml, 74
 Xdebug, 431
 rozwój projektu, 21
 równoległe rodziny produktów, 175

S

schemat bazy danych, 271
 segmenty pamięci
 współdzielonej, 242
 Selenium, 398
 separacja modelu dziedziny, 277
 serializacja, 242, 244

serwer
 Apache, 102
 ciągłej integracji, 436
 Git, 365, 439
 Jenkins, 436
 Selenium, 398
 skaner, 461
 składnia
 klamrowa, 100
 wierszowa, 100
 składniki pakietu, 338
 składowanie danych, 318
 składowe
 chronione, 50
 klasy, 33
 prywatne, 50
 publiczne, 50
 stałe, 58
 statyczne, 55
 skrypt transakcji, 270, 274, 276
 słowo kluczowe
 abstract, 59
 as, 98
 catch, 76
 class, 31, 108
 clone, 88, 178
 const, 58
 extends, 47, 62
 final, 80, 81
 finally, 79
 function, 35, 92
 implements, 62
 instanceof, 65, 66
 interface, 61
 namespace, 97
 parent, 48, 72
 private, 33, 35, 50, 70
 protected, 33, 35, 50
 public, 33, 35, 50
 self, 56, 71
 static, 55, 71
 throw, 75
 trait, 64
 try, 76
 use, 98
 yield, 288
 specjalizowanie klasy wyjątku, 76
 SPL, 219
 spójność, 127
 sprzęganie, 127, 157
 sprzężenie, 158
 stała
`__NAMESPACE__`, 99
`__PATH_SEPARATOR`, 102
 standardy kodowania, 433

stosowanie
 cech typowych, 63
 dziedziczenia, 46
 wzorców, 146
 strategia kreacji obiektów, 165
 strona WWW kanału, 346, 347
 struktura
 dziedziczenia, 153
 plików pakietu, 340
 wzorca, 146
 wzorca Composite, 188
 strukturalizacja klas, 183
 symbole widoczności atrybutów,
 135
 symulowanie systemu pakietów,
 100
 system
 integracji ciągłej, 454
 pakietów, 100
 plików, 100
 szablon widoku, 268, 269
 szukanie klasy, 106

Ś

ścieżka dostępu do plików, 248
 ścieżki przeszukiwania, 101

T

tabela, 57
 tablica getBacktrace(), 335
 testowanie, 324, 453
 aplikacji WWW, 394
 ręczne, 382
 systemu WOO, 396
 wyjątków, 386
 testy
 funkcjonalne, 381
 jednostkowe, 381, 429, 431
 tryb
 BLOGGS, 171
 MEGA, 171
 tworzenie
 kanału, 345
 kopii obiektu, 88
 obiektów, 163
 obserwatorów, 218
 odnośników, 359
 pakietów PEAR, 337, 405
 przypadku testowego, 384
 repozytorium zdalnego, 365
 szkieletu testu, 399
 typ ParamHandler, 126

- typy
argumentów metod, 37
elementarne, 37, 38
obiektywne, 40
zależności, 343
- U**
- udostępnianie repozytorium, 366
ukrywanie szczegółów implementacji, 158
UML, Unified Modeling Language, 133–141
unikanie
kolizji nazw, 65
silnego sprzężenia, 451
usuwanie
katalogu, 374
obiektów, 219
pakietów, 329
pliku, 374
utrwalanie danych
konfiguracyjnych, 257
uzupełnianie ścieżek, 102
użycie klas dekoracji, 195
- W**
- warstwa
danych, 233, 277, 279
kontrolni żądań, 226
logiczna aplikacji, 226
logiki biznesowej, 233, 270
poleceń i kontroli, 233
prezentacji, 244, 279
widoku, 233
warstwy systemu korporacyjnego, 232
warunkowe zadanie, 416
wczytywanie kodu, 103
wdrażanie Abstract Factory, 179
widok, 244
wielokrotne stosowanie kodu, 449
wizytator, 224, 225
właściwości
kompilacji, 410
warunkowe, 415
właściwość
\$xml, 74
dbpass, 413
włączanie pakietu, 333
współużytkowanie składowych, 90
wyjątek, 75, 78, 336, 386
FileException, 77
MyPearException, 336
wykonywanie zadań, 201
wykrywanie błędów, 454
wymuszanie typu, 41
wyodrębnianie
algorytmów, 155, 211
przepływu sterowania, 256
wyprowadzanie pochodnych klasy, 161
wyrażenia regularne, 202, 213
wyróżnienie parametru żądania, 228
wytwórca, 170, 176
wytwórnia
abstrakcji, 174
aktualizacji, 312
obiektów dziedziny, 304
selekcji, 312, 314
TerrainFactory, 180, 181
wywołania chybione, 81
wywołania metod
przesłoniętych, 50
wywołanie
metody, 111
zwrotne, 55, 91, 94
wzorce, 24, 450
wzorce projektowe, 24, 143–151, 161, 450
Abstract Factory, 174, 177, 181, 306
Application Controller, 232, 253, 254
baz danych, 162, 279
Command, 103, 226, 230
Composite, 183, 188, 191
Data Mapper, 280, 316, 318
Decorator, 192, 195, 197
Domain Model, 232, 274
Domain Object Assembler, 318
Domain Object Factory, 303, 304, 318
Facade, 197, 199
Factory Method, 170, 172
Front Controller, 232, 244
generowania obiektów, 162
Identity Map, 293, 296, 318
Identity Object, 306, 318
Intercepting Filter, 196
Interpreter, 201, 209
korporacyjne, 162, 231
Lazy Load, 301, 318
Observer, 214, 215
organizacji obiektów i klas, 162
Page Controller, 232, 264
Prototype, 178, 180
Registry, 232, 235
Selection Factory, 312, 318
Singleton, 167, 169, 181
Strategy, 209, 211
Template View, 232, 268
Transaction Script, 232, 270, 274, 275, 276
Unit of Work, 297, 318
Update Factory, 312, 318
View Helper, 268
Visitor, 220, 225
zadaniowe, 162
- X**
- XP, eXtreme Programming, 23
- Z**
- zaczepy, 201
zadania Ping, 441
zadanie, target, 408
exec, 435
warunkowe condition, 415
zależności, 342, 343
Zandstra Matt, 13
zapis do pliku, 124
zapytania SQL, 273
zarządzanie, 21
grupami obiektów, 184
kolekcjami wielowierszowymi, 287
komponentami bazodanowymi, 279
kryteriami zapytań, 307
serializacją, 244
wersjami, 363, 453
żądaniem i widokami, 264
zasady projektowe, 451
zasięg
aplikacji, 238, 241
klasy, 123, 128
zmiennej, 238
zastosowanie kompozycji, 155
zatwierdzanie zmian, 370
zbieranie nieużytków, garbage collection, 86
zbiór plików, 416
zestaw testów, 387
złota rączka, 133
zmiana
aliasu, 98
dostępności metod, 70
zmienna
globalna, 169, 241
środowiskowa DBPASS, 414
znacznik, 182

@link, 360, 361
@package, 353
@see, 360
@uses, 361
contents, 341
dependencies, 342

znak ukośnika, 98
zrównoleganie funkcjonalności,
175
zrzucanie wyjątku, 75, 80
zwielokrotnianie kodu, 133

Ż

żądania użytkowników, 226
żądanie, 250
Request, 250
śladu polecenia, 261

PROGRAM PARTNERSKI

GRUPY WYDAWNICZEJ HELION



1. ZAREJESTRUJ SIĘ
2. PREZENTUJ KSIĄŻKI
3. ZBIERAJ PROWIZJĘ

Zmień swoją stronę WWW
w działający bankomat!

Dowiedz się więcej i dołącz już dzisiaj!

<http://program-partnerski.helion.pl>

GRUPA WYDAWNICZA

 **Helion SA**

PHP

Obiekty, wzorce, narzędzia

Język PHP przeżył długą drogę od swoich początków do obecnego poziomu rozwoju. Dziś jest pełnoprawnym, obiektowym językiem programowania, wciąż zdobywającym zaufanie i używanym w coraz większych projektach. Jeżeli znasz ten język od dawna, lecz nie jesteś przekonany, że nadaje się do zaawansowanych zastosowań, albo dopiero zaczynasz karierę i potrzebujesz szybkiego wprowadzenia w świat obiektów czy wzorców projektowych w języku PHP, to jest właśnie książka dla Ciebie!

Sięgnij po nią i przekonaj się na własnej skórze, że PHP to dojrzały język, który nie ustępuje konkurentom. W trakcie lektury poznasz podstawowe zagadnienia związane z programowaniem obiektowym, a następnie przejdziesz do nauki zaawansowanej obsługi obiektów w języku PHP. Kolejne rozdziały zostały poświęcone między innymi wzorcom projektowym, dobrym i złym praktykom, zastosowaniu PEAR i Pyrus oraz sposobom automatycznego generowania dokumentacji i tworzenia kodu wysokiej jakości dzięki testom automatycznym. Książka ta jest doskonałą lekturą dla każdego programisty PHP chcącego podnieść swoje kwalifikacje.

Dzięki tej książce:

- poznasz podstawowe zagadnienia związane z programowaniem obiektowym,
- nauczysz się operować na obiektach w PHP,
- poznasz przydatne wzorce projektowe,
- unikniesz typowych problemów,
- przygotujesz testy jednostkowe.

Apress®

Nr katalogowy: 24885

Księgarnia internetowa:
<http://helion.pl>

Zamówienia telefoniczne:
0 801 339900
0 601 339900

helion.pl
księgarnia
internetowa

Sprawdź najnowsze promocje:
• <http://helion.pl/promocje>
Książki najchętniej czytane:
• <http://helion.pl/bestsellery>
Zamów informacje o nowościach:
• <http://helion.pl/nowosci>

Helion

Helion SA
ul. Kościuszki 1c, 44-100 Gliwice
tel.: 32 230 98 63
e-mail: helion@helion.pl
<http://helion.pl>

sięgnij po WIECEJ



KOD KORZYŚCI

cena: 79,00 zł

ISBN 978-83-246-9178-4



9 788324 691784