

Kompendium wiedzy
na temat obiektów w PHP

▼ Jak wykorzystać techniki obiektowe w PHP?

▼ Jak obsługiwać wyjątkowe sytuacje?

▼ Jak zapewnić ciągłą integrację kodu?

PHP

Obiekty, wzorce, narzędzia

Wydanie III

Matt Zandstra



Apress®

» Idź do

- Spis treści
- Przykładowy rozdział
- Skorowidz

» Katalog książek

- Katalog online
- Zamów drukowany katalog

» Twój koszyk

- Dodaj do koszyka

» Cennik i informacje

- Zamów informacje o nowościach
- Zamów cennik

» Czytelnia

- Fragmenty książek online

» Kontakt

Helion SA
ul. Kościuszki 1c
44-100 Gliwice
tel. 32 230 98 63
e-mail: helion@helion.pl
© Helion 1991–2011

PHP. Obiekty, wzorce, narzędzia. Wydanie III

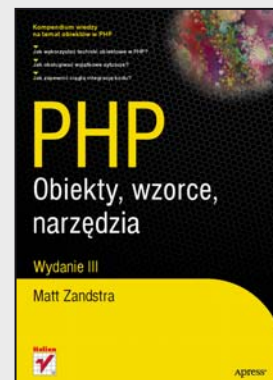
Autor: Matt Zandstra

Tłumaczenie: Przemysław Szeremiota

ISBN: 978-83-246-3026-4

Tytuł oryginału: [PHP Objects, Patterns and Practice, Third Edition](#)

Format: 168×237, stron: 496



Kompendium wiedzy na temat obiektów w PHP!

- Jak wykorzystać techniki obiektowe w PHP?
- Jaka obsługiwać wyjątkowe sytuacje?
- Jak zapewnić ciągłą integrację kodu?

PHP jest dowodem na to, że czas potrzebny na opanowanie języka programowania oraz uzyskanie pierwszych efektów wcale nie musi zmierzać do nieskończoności! Łatwa konfiguracja środowiska programistycznego, tanie i ogólnodostępne serwery do umieszczania własnych aplikacji oraz witryn opartych o PHP, a ponadto duża liczba publikacji i chętna do pomocy społeczność użytkowników sprawiły, że język PHP błyskawicznie zdobył uznanie. W ciągu ostatnich lat język ten przeszedł obiektywą rewolucję. Dostęp do zaawansowanych narzędzi, wzrost świadomości oraz zmiany w samym języku wystarczyły, by programiści coraz powszechniej zaczęli stosować techniki obiektowe w trakcie tworzenia rozwiązań w PHP.

W trakcie lektury tej książki zostaniesz wprowadzony w świat obiektów w PHP. Poznasz pojęcia ściśle związane z tym podejściem do programowania – klasa, obiekt, metoda, dziedziczenie czy widoczność zmiennych to słowa, które nabiorą dla Ciebie nowego znaczenia. Na kolejnych stronach przeczytasz o tym, jak obsługiwać wyjątkowe sytuacje, korzystać z interfejsów, domknięć i funkcji zwrotnych. Ponadto zdobędziesz wiedzę na temat projektowania obiektowego. Zasada hermetyzacji i diagramy UML staną się dla Ciebie całkowicie jasne. Autor bardzo dużo czasu poświęca wzorcom projektowym w PHP. Dzięki nim Twój kod stanie się przejrzysty, a nawet najtrudniejsze problemy będą zdecydowanie łatwiejsze do rozwiązania. Na sam koniec sprawdzisz, jak najlepiej dokumentować kod, korzystać z dodatkowych bibliotek oraz wykonywać testy jednostkowe. Książka ta stanowi kompendium wiedzy na temat obiektowego programowania w PHP, dlatego musi się znaleźć na półce każdej osoby choć trochę związanej z tym popularnym językiem programowania!

- Historia obiektowości w PHP
- Elementarz pojęć z programowania obiektowego
- Obsługa błędów
- Wykorzystanie interfejsów, klas abstrakcyjnych oraz metod statycznych
- Projektowanie obiektowe – diagramy UML, hermetyzacja
- Wzorce projektowe
- Wykorzystanie PEAR i Pyrus
- Generowanie dokumentacji za pomocą phpDocumentor
- Zarządzanie kodem za pomocą Subversion
- Przygotowywanie testów jednostkowych
- Automatyzacja instalacji
- Ciągła integracja kodu

Twórz lepszy, czytelniejszy i wydajniejszy kod w PHP!

Spis treści

O autorze	11
O recenzencie technicznym	13
Podziękowania	15
Przedmowa do trzeciego wydania	17
Część I 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 trzecim wydaniu	26
Podsumowanie	26
Część II Obiekty	27
Rozdział 2. PHP a obiekty	29
Nieoczekiwany sukces obiektów w PHP	29
PHP/FI — u zarania języka	29
PHP3 — składniowy lukier	30
Cicha rewolucja — PHP4	30
PHP5 — nieuchronne zmiany	31
W przyszłości	32
Debata obiektowa — za czy przeciw?	32
Podsumowanie	32
Rozdział 3. Obiektowy elementarz	33
Klasy i obiekty	33
Pierwsza klasa	33
Pierwszy obiekt (lub dwa)	34
Definiowanie składowych klasy	35

Metody	37
Metoda konstrukcji obiektu	38
Typy argumentów metod	39
Typy elementarne	40
Typy obiektowe	42
Dziedziczenie	44
Problemy związane z dziedziczeniem	44
Stosowanie dziedziczenia	48
Zarządzanie dostępem do klasy — słowa <code>public</code> , <code>private</code> i <code>protected</code>	52
Podsumowanie	56
Rozdział 4. Zaawansowana obsługa obiektów	57
Metody i składowe statyczne	57
Składowe stałe	60
Klasy abstrakcyjne	61
Interfejsy	62
Późne wiązanie statyczne: słowo <code>static</code>	64
Obsługa błędów	66
Wyjątki	68
Klasy i metody finalne	72
Przechwytywanie chybionych wywołań	73
Definiowanie destruktorów	77
Wykonywanie kopii obiektów	78
Reprezentacja obiektu w ciągach znaków	80
Wywołania zwrotne, funkcje anonimowe i domknięcia	81
Podsumowanie	85
Rozdział 5. Narzędzia obiektowe	87
PHP a pakiety	87
Pakiety i przestrzenie nazw w PHP	87
Ratunek — przestrzenie nazw	88
Symulowanie systemu pakietów na bazie systemu plików	92
Nazwy à la PEAR	93
Ścieżki przeszukiwania	93
Automatyczne wczytywanie kodu	95
Klasy i funkcje pomocnicze	96
Szukanie klasy	97
Badanie obiektów i klas	98
Badanie metod	99
Badanie składowych	100
Badanie relacji dziedziczenia	100
Badanie wywołań metod	101
Interfejs retrospekcji — Reflection API	102
Zaczynamy	102
Pora zakasać rękawy	103
Badanie klasy	104
Badanie metod	106
Badanie argumentów metod	107
Korzystanie z retrospekcji	108
Podsumowanie	111

Rozdział 6.	Obiekty a projektowanie obiektowe	113
	Czym jest projektowanie?	113
	Programowanie obiektowe i proceduralne	114
	Odpowiedzialność	117
	Spójność	117
	Sprzęganie	118
	Ortogonalność	118
	Zasięg klas	118
	Polimorfizm	119
	Hermetyzacja	121
	Nieważne jak	122
	Cztery drogowskazy	122
	Zwielokrotnianie kodu	123
	Przemądrzałe klasy	123
	Złota rączka	123
	Za dużo warunków	123
	Język UML	123
	Diagramy klas	124
	Diagramy sekwencji	129
	Podsumowanie	131
Część III	Wzorce	133
Rozdział 7.	Czym są wzorce projektowe? Do czego się przydają?	135
	Czym są wzorce projektowe?	135
	Wzorzec projektowy	137
	Nazwa	137
	Problem	138
	Rozwiązanie	138
	Konsekwencje	138
	Format wzorca według Bandy Czworga	138
	Po co nam wzorce projektowe?	139
	Wzorzec projektowy definiuje problem	139
	Wzorzec projektowy definiuje rozwiązanie	139
	Wzorce projektowe są niezależne od języka programowania	139
	Wzorce definiują słownictwo	140
	Wzorce są wypróbowane	140
	Wzorce mają współpracować	141
	Wzorce promują zasady projektowe	141
	Wzorce projektowe a PHP	141
	Podsumowanie	141
Rozdział 8.	Wybrane zasady wzorców	143
	Ośnienie wzorcami	143
	Kompozycja i dziedziczenie	144
	Problem	144
	Zastosowanie kompozycji	147
	Rozprzęganie	149
	Problem	149
	Osłabianie sprzężenia	150

Kod ma używać interfejsów, nie implementacji	151
Zmienne koncepcje	153
Nadmiar wzorców	153
Wzorce	154
Wzorce generowania obiektów	154
Wzorce organizacji obiektów i klas	154
Wzorce zadaniowe	154
Wzorce korporacyjne	154
Wzorce baz danych	154
Podsumowanie	154
Rozdział 9. Generowanie obiektów	157
Generowanie obiektów — problemy i rozwiązania	157
Wzorzec Singleton	161
Problem	161
Implementacja	162
Konsekwencje	163
Wzorzec Factory Method	164
Problem	164
Implementacja	166
Konsekwencje	167
Wzorzec Abstract Factory	168
Problem	168
Implementacja	169
Konsekwencje	171
Prototyp	172
Problem	173
Implementacja	173
Ależ to oszustwo!	175
Podsumowanie	177
Rozdział 10. Wzorce elastycznego programowania obiektowego	179
Strukturalizacja klas pod kątem elastyczności obiektów	179
Wzorzec Composite	179
Problem	180
Implementacja	182
Konsekwencje	185
Composite — podsumowanie	188
Wzorzec Decorator	188
Problem	188
Implementacja	190
Konsekwencje	193
Wzorzec Facade	193
Problem	193
Implementacja	195
Konsekwencje	195
Podsumowanie	196

Rozdział 11. Reprezentacja i realizacja zadań	197
Wzorzec Interpreter	197
Problem	197
Implementacja	198
Ciemne strony wzorca Interpreter	204
Wzorzec Strategy	205
Problem	205
Implementacja	206
Wzorzec Observer	210
Implementacja	211
Wzorzec Visitor	216
Problem	216
Implementacja	217
Wady wzorca Visitor	221
Wzorzec Command	222
Problem	222
Implementacja	222
Podsumowanie	225
Rozdział 12. Wzorce korporacyjne	227
Przegląd architektury	227
Wzorce	228
Aplikacje i warstwy	228
Małe oszustwo na samym początku	231
Wzorzec Registry	231
Implementacja	232
Warstwa prezentacji	240
Wzorzec Front Controller	240
Wzorzec Application Controller	249
Wzorzec Page Controller	259
Wzorce Template View i Helper View	263
Warstwa logiki biznesowej	266
Wzorzec Transaction Script	266
Wzorzec Domain Model	270
Podsumowanie	273
Rozdział 13. Wzorce bazodanowe	275
Warstwa danych	275
Wzorzec Data Mapper	276
Problem	276
Implementacja	276
Wzorzec Identity Map	288
Problem	288
Implementacja	288
Konsekwencje	291
Wzorzec Unit of Work	291
Problem	291
Implementacja	291
Konsekwencje	295

Wzorzec Lazy Load	295
Problem	295
Implementacja	296
Konsekwencje	297
Wzorzec Domain Object Factory	297
Problem	298
Implementacja	298
Konsekwencje	299
Wzorzec Identity Object	300
Problem	300
Implementacja	301
Konsekwencje	305
Wzorce Selection Factory i Update Factory	306
Problem	306
Implementacja	306
Konsekwencje	309
Co zostało z wzorca Data Mapper?	309
Podsumowanie	311
Część IV Narzędzia	313
Rozdział 14. Dobre (i złe) praktyki	315
Nie tylko kod	315
Pukanie do otwartych drzwi	316
Jak to zgrać?	317
Uskrzydlenie kodu	318
Dokumentacja	319
Testowanie	320
Ciągła integracja	320
Podsumowanie	321
Rozdział 15. PEAR i Pyrus	323
Czym jest PEAR?	323
Pyrus	324
Instalowanie pakietu	325
Kanały PEAR	327
Korzystanie z pakietu z PEAR	328
Obsługa błędów w pakietach PEAR	330
Tworzenie własnych pakietów PEAR	333
Plik package.xml	333
Składniki pakietu	333
Element contents	335
Zależności	337
Dookreślanie instalacji — phprelease	339
Przygotowanie pakietu do dystrybucji	340
Konfigurowanie własnego kanału PEAR	340
Podsumowanie	344

Rozdział 16. Generowanie dokumentacji — phpDocumentor	345
Po co nam dokumentacja?	345
Instalacja	346
Generowanie dokumentacji	347
Komentarze DocBlock	348
Dokumentowanie klas	349
Dokumentowanie plików	351
Dokumentowanie składowych	351
Dokumentowanie metod	352
Tworzenie odnośników w dokumentacji	354
Podsumowanie	356
Rozdział 17. Zarządzanie wersjami projektu z Subversion	357
Po co mi kontrola wersji?	357
Skąd wziąć Subversion?	358
Konfigurowanie repozytorium Subversion	359
Tworzenie repozytorium	359
Dostęp do repozytorium Subversion	360
Rozpoczynamy projekt	361
Aktualizacja i zatwierdzanie zmian	363
Dodawanie i usuwanie plików i katalogów	367
Dodawanie pliku	367
Usuwanie pliku	367
Dodawanie katalogu	367
Usuwanie katalogów	368
Etykietowanie i eksportowanie wydania	368
Etykietowanie projektu	368
Eksportowanie projektu	369
Rozgałęzianie projektu	369
Podsumowanie	373
Rozdział 18. Testy jednostkowe z PHPUnit	375
Testy funkcjonalne i testy jednostkowe	375
Testowanie ręczne	376
PHPUnit	378
Tworzenie przypadku testowego	378
Metody asercji	379
Testowanie wyjątków	380
Uruchamianie zestawów testów	381
Ograniczenia	382
Atrapy i imitacje	383
Dobry test to obłany test	386
Testy dla aplikacji WWW	389
Przygotowanie aplikacji WWW do testów	389
Proste testy aplikacji WWW	391
Selenium	393
Słowo ostrzeżenia	397
Podsumowanie	398

Rozdział 19. Automatyzacja instalacji z Phing	401
Czym jest Phing?	402
Pobieranie i instalacja pakietu Phing	402
Montowanie dokumentu kompilacji	403
Różnicowanie zadań kompilacji	404
Właściwości	406
Typy	410
Operacje	414
Podsumowanie	418
Rozdział 20. Ciągła integracja kodu	419
Czym jest ciągła integracja?	419
Przygotowanie projektu do ciągłej integracji	421
CruiseControl i phpUnderControl	427
Instalowanie CruiseControl	427
Instalowanie phpUnderControl	428
Instalowanie projektu do integracji ciągłej	430
Podsumowanie	439
Część V Konkluzje	441
Rozdział 21. Obiekty, wzorce, narzędzia	443
Obiekty	443
Wybór	444
Hermetyzacja i delegowanie	444
Osłabianie sprzężenia	444
Zdatność do wielokrotnego stosowania kodu	445
Estetyka	445
Wzorce	446
Co dają nam wzorce?	446
Wzorce a zasady projektowe	447
Narzędzia	448
Testowanie	449
Dokumentacja	449
Zarządzanie wersjami	449
Automatyczna kompilacja (instalacja)	450
System integracji ciągłej	450
Co pominęliśmy?	450
Podsumowanie	451
Część VI Dodatki	453
Dodatek A Bibliografia	455
Książki	455
Publikacje	456
Witryny WWW	456
Dodatek B Prosty analizator leksykalny	459
Skaner	459
Analizator leksykalny	466
Skorowidz	477

ROZDZIAŁ 6



Obiekty a projektowanie obiektowe

Znamy już dość szczegółowo mechanizmy obsługi obiektów w języku PHP, wypadaloby więc zostawić na boku szczegóły i zastanowić się nad najlepszymi możliwymi zastosowaniami poznanych narzędzi. W niniejszym rozdziale wprowadzę Cię w kwestie oddalone nieco od obiektów, a bliższe projektowaniu. Przyjrzymy się między innymi UML, czyli efektywnemu graficznemu językowi opisu systemów obiektowych.

Rozdział będzie traktował o:

- *Podstawach projektowania* — co rozumieć pod pojęciem projektowania i w czym projektowanie obiektowe różni się od proceduralnego.
- *Zasięgu klas* — jak decydować o zawartości i odpowiedzialności klas.
- *Hermetyzacji* — czyli ukrywaniu implementacji i danych za interfejsami.
- *Polimorfizmie* — czyli stosowaniu wspólnych typów bazowych dla uzyskania transparentnej podmiany specjalizowanych typów pochodnych.
- *Języku UML* — zastosowaniach diagramów w opisach architektur obiektowych.

Czym jest projektowanie?

Jedno ze znaczeń pojęcia „projektowanie kodu” to definiowanie systemu — określanie dla systemu wymagań i zakresu jego zadań. Co system powinien robić? Czego potrzebuje do realizacji swoich zadań? Jakże dane system generuje? Czy spełniają one wyrażone uprzednio wymagania? Na niższym poziomie projektowanie oznacza proces definiowania uczestników systemu i rozpoznawania zachodzących pomiędzy nimi relacji. W tym rozdziale zajmiemy się właśnie projektowaniem w tym drugim ujęciu, a więc klasami i obiektami oraz ich powiązaniem.

Jak rozpoznać elementy systemu? System obiektowy składa się z klas. Należy zdecydować o naturze poszczególnych klas uczestniczących w systemie. Klasy składają się po części z metod, więc definiując klasy, trzeba zdecydować o grupowaniu metod. Klasy często uczestniczą też w relacjach dziedziczenia, mających im zapewnić spełnianie wymogów wspólnych dla poszczególnych części systemu interfejsów. Pierwsze wyzwanie w projektowaniu systemu tkwi właśnie w rozpoznaniu i wytypowaniu tych interfejsów.

Klasy mogą jednak wchodzić również w inne relacje. Można bowiem tworzyć klasy składające się z innych klas i typów albo utrzymujące listy egzemplarzy innych typów. Klasy mogą korzystać z obiektów zewnętrznych. Klasy dysponują więc wbudowanym potencjałem do realizowania relacji kompozycji i użycia (na przykład za pośrednictwem narzucania typów obiektowych w sygnaturach metod), ale właściwe relacje zawiązują się dopiero w czasie wykonania, co zwiększa elastyczność projektu. W rozdziale zaprezentowane zostaną sposoby modelowania tego rodzaju zależności; będą one podstawą do omówienia zawartego w kolejnej części książki.

W ramach procesu projektowego należy również decydować, kiedy dana operacja należy do danego typu, a kiedy powinna należeć do innego typu, wykorzystywanego przez dany. Każdy etap projektowania oznacza nowe wybory i decyzje; jedne z nich prowadzą do elegancji i przejrzystości, inne mogą się na decyzjach zemścić.

Rozdział ten będzie w części poświęcony pewnym kwestiom, których rozpoznanie jest pomocne w podejmowaniu właściwych decyzji.

Programowanie obiektowe i proceduralne

Czym różni się kod obiektowy od tradycyjnego kodu proceduralnego? Najłatwiej powiedzieć, że główna różnica tkwi w obecności obiektów. Nie jest to jednak stwierdzenie ani odkrywcze, ani prawdziwe. Przecież w języku PHP obiekty mogą być z powodzeniem wykorzystywane w kodzie proceduralnym. Na porządku dziennym jest również definiowanie klas opartych na kodzie proceduralnym. Obecność klas i obiektów nie jest więc równoznaczna z obiektością — nawet w językach takich jak Java, gdzie większość elementów programu to obiekty.

Jedną z kluczowych różnic pomiędzy kodem obiektowym a proceduralnym odnajdujemy w podziale odpowiedzialności. Kod proceduralny przyjmuje postać sekwencji poleceń i wywołań metod. Do obsługi różnych stanów programu wydziela się kod kontrolujący. Taki model odpowiedzialności prowokuje powielanie kodu i uściślanie zależności pomiędzy elementami projektu. W kodzie obiektowym mamy zaś do czynienia z próbą minimalizacji owych zależności przez przekładanie odpowiedzialności za różne zadania na obiekty rezydujące w systemie.

W tym rozdziale przedstawię przykładowy, uproszczony problem i przeanalizuję jego proceduralne i obiektowe rozwiązania. Założmy, że zadanie polega na skonstruowaniu narzędzia odczytu i zapisu plików konfiguracyjnych. Ponieważ najbardziej interesuje nas ogólna struktura kodu, nie będziemy zagłębiać się w żadnym z przypadków w szczególności implementacyjne.

Zacznijmy od podejścia proceduralnego. Odczytywać i zapisywać będziemy dane tekstowe w formacie:

klucz:wartość

Wystarczy nam do tego dwie funkcje:

```
function readParams($sourceFile) {
    $params = array();
    // wczytaj parametry z pliku $sourceFile...
    return $params;
}

function writeParams($params, $destFile) {
    // zapisz parametry do pliku $destFile...
}
```

Funkcja `readParams()` wymaga przekazania jako argumentu wywołania nazwy pliku źródłowego. W jej ciele następuje próba otwarcia pliku, a potem odczyt kolejnych wierszy tekstu. Na podstawie wyszukiwanych w poszczególnych wierszach par kluczy i wartości konstruowana jest asocjacyjna tablica parametrów zwracana następnie do wywołującego. Funkcja `writeParams()` przyjmuje z kolei na wejście tablicę asocjacyjną i ścieżkę dostępu do pliku docelowego. W implementowanej w ciele funkcji pętli przegląda tablicę, zapisując wyodrębniane z niej pary klucz i wartość w pliku docelowym. Oto kod używający obu funkcji:

```
$file = "./param.txt";
$array['klucz1'] = "wartość1";
$array['klucz2'] = "wartość2";
$array['klucz3'] = "wartość3";
writeParams($array, $file); // zapis tablicy parametrów do pliku
$output = readParams($file); // odczyt tablicy parametrów z pliku
print_r($output);
```

Kod jest, jak widać, stosunkowo zwięzły i nie powinien sprawiać problemów konserwatorskich. Do utworzenia i zapisania pliku *param.txt* użyjemy wywołania `wriTeParams()`, którego zadaniem jest utrwalenie par klucz – wartość:

```
klucz1:wartość1
klucz2:wartość2
klucz3:wartość3
```

Niestety, dowiadujemy się właśnie, że narzędzie powinniśmy przystosować do obsługi prostych plików XML o następującym formacie:

```
<params>
  <param>
    <key>klucz</key>
    <val>wartość</val>
  </param>
</params>
```

Rozpoznanie formatu zapisu pliku powinno się odbywać na podstawie rozszerzenia pliku — dla plików z rozszerzeniem *.xml* należałoby wszcząć procedurę odczytu w formacie XML.

Choć i tym razem poradzimy sobie z zadaniem, zagraża nam komplikacja kodu i zwiększenie uciążliwości utrzymania (konserwacji). Mamy teraz dwie możliwości. Albo będziemy sprawdzać rozszerzenie pliku parametrów w kodzie zewnętrznym, albo wewnątrz funkcji odczytujących i zapisujących. Spróbujmy oprogramować drugą z opcji:

```
function readParams($source) {
    $params = array();
    if (preg_match( "\\.xml$/i", $source)) {
        // odczyt parametrów z pliku XML
    } else {
        // odczyt parametrów z pliku tekstowego
    }
    return $params;
}

function writeParams($params, $source) {
    if (preg_match( "\\.xml$/i", $source)) {
        // zapis parametrów do pliku XML
    } else {
        // zapis parametrów do pliku tekstowego
    }
}
```

■ **Uwaga** Kod przykładowy to zawsze sztuka kompromisu. Musi być dostatecznie czytelny, aby ilustrował konkretną koncepcję bądź problem, co często oznacza konieczność rezygnacji z kontroli błędów i elastyczności. Innymi słowy, prezentowany tu przykład jest jedynie ilustracją kwestii projektowania i powielania kodu, w żadnym razie nie będąc wzorcową implementacją parsowania i zapisywania danych w plikach. Z tego względu wszędzie tam, gdzie nie jest to konieczne do omówienia, implementacja fragmentów kodu została zwyczajnie pominięta.

Jak widać, w każdej z funkcji musieliśmy uwzględnić test rozszerzenia pliku parametrów. Tego rodzaju zwielokrotnienie kodu może być w przyszłości przyczyną problemów. Gdybyśmy bowiem stanęli w obliczu zadania obsługi kolejnego formatu pliku parametrów, musielibyśmy pamiętać o synchronizacji kodu sprawdzającego rozszerzenie w obu funkcjach.

Spróbujmy to samo zadanie zrealizować za pomocą prostych klas. Na początek zdefiniujemy abstrakcyjną klasę bazową wyznaczającą interfejs typu:

```

abstract class ParamHandler {
    protected $source;
    protected $params = array();

    function __construct($source) {
        $this->source = $source;
    }

    function addParam($key, $val) {
        $this->params[$key] = $val;
    }

    function getAllParams() {
        return $this->params;
    }

    static function getInstance($filename) {
        if ( preg_match("/\.xml$/i", $filename))
            return new XmlParamHandler($filename);
        }
        return new TextParamHandler($filename);
    }

    abstract function write();
    abstract function read();
}

```

W klasie tej definiujemy metodę `addParam()` służącą do uzupełniania tablicy parametrów i metodę `getAllParams()` dającą użytkownikom dostęp do kopii tablicy parametrów.

Tworzymy też statyczną metodę `getInstance()`, której zadaniem jest analiza rozszerzenia nazwy pliku parametrów i zwrócenie użytkownikowi klasy specjalizowanej do obsługi pliku odpowiedniego formatu. Wreszcie definiujemy dwie abstrakcyjne metody: `write()` i `read()`, wymuszając ich implementację w klasach pochodnych i tym samym narzucając im wspólny interfejs obsługi plików.

-
- **Uwaga** Użycie metody statycznej do generowania obiektów klas pochodnych w klasie nadrzędnej jest bardzo wygodne. Taka decyzja projektowa ma jednak również wady. Typ `ParamHandler` jest teraz zasadniczo w swoich głównych instrukcjach warunkowych ograniczony do pracy z konkretnymi klasami. A jeśli zechcemy obsłużyć inny format danych? Oczywiście właściciel klasy `ParamHandler` może zawsze uzupełnić metodę `getInstance()`. Ale już programista kodu klienckiego nie ma łatwej możliwości zmieniania klasy bibliotecznej (sama zmiana nie jest może specjalnie trudna, ale pojawia się problem ponownego aplikowania zmian w kolejnych wersjach bibliotek). Zagadnienia tworzenia obiektów omówię bardziej szczegółowo w rozdziale 9.
-

Zdefiniujemy teraz owe klasy specjalizowane (znów gwoli przejrzystości przykładu pomijając szczegóły implementacyjne):

```

class XmlParamHandler extends ParamHandler {
    function write() {
        // zapis tablicy parametrów $this->params w pliku XML
    }

    function read() {
        // odczyt pliku XML i wypełnienie tablicy parametrów $this->params
    }
}

class TextParamHandler extends ParamHandler {
    function write() {

```

```

    // zapis tablicy parametrów $this->params w pliku tekstowym
}

function read() {
    // odczyt pliku tekstowego i wypełnienie tablicy parametrów $this->params
}
}

```

Obie klasy ograniczają się do implementowania metod `wirte()` i `read()`. Każda z klas zapisuje i odczytuje parametry w odpowiednim dla siebie formacie.

Użytkownik takiego zestawu klas będzie mógł zapisywać i odczytywać pliki parametrów niezależnie od ich formatu, całkowicie ignorując (i nie mając nawet tego świadomości) znaczenie rozszerzenia nazwy pliku:

```

$test = ParamHandler::getInstance("./params.xml");
$test->addParam("klucz1", "wartość1");
$test->addParam("klucz2", "wartość2");
$test->addParam("klucz3", "wartość3");
$test->write(); // zapis w formacie XML

```

Równie łatwo można odczytywać parametry z pliku niezależnie od jego formatu:

```

$test = ParamHandler::getInstance("./params.txt");
$test->read(); // odczyt z pliku tekstowego

```

Spróbujmy podsumować naukę płynącą z ilustracji tych dwóch metod projektowych.

Odpowiedzialność

Odpowiedzialność za decyzję co do formatu pliku w podejściu proceduralnym bierze na siebie użytkownik (kod kontrolujący) i czyni to nie raz, a dwa razy. Co prawda kod sprawdzający rozszerzenie został przeniesiony do wnętrza funkcji, ale nie przesłania to faktycznego przepływu sterowania. Wywołanie funkcji `readParams()` musi zawsze występować w kontekście innym od kontekstu wywołania `writeParams()`, więc test rozszerzenia pliku musi być powtarzany w każdej z tych funkcji, niezależnie od historii ich wywołań.

W wersji obiektowej wybór formatu pliku dokonywany jest w ramach statycznej metody `getInstance()`, więc test rozszerzenia jest wykonywany tylko jednokrotnie, a jego wynik wpływa na wybór i konkretyzację odpowiedniej klasy pochodnej. Użytkownik nie bierze odpowiedzialności za implementację — korzysta po prostu z otrzymanego obiektu, nie wnikając w szczegóły implementacji klasy tego obiektu. Wie jedynie tyle, że korzysta z obiektu typu `ParamHandler` i że obiekt ten obsługuje operacje `wirte()` i `read()`. Kod proceduralny musiał wciąż zajmować się szczegółami swojej implementacji, a w kodzie obiektowym mamy przeniesienie zainteresowania z implementacji na interfejs. Przeniesienie odpowiedzialności za implementację z kodu użytkownika na kod klasy powoduje, że użytkownik nie cierpi w żaden sposób wskutek zmian decyzji i włączania do hierarchii klas obsługujących nowe formaty plików parametrów — dla niego te rozszerzenia są transparentne.

Spójność

Spójność to bliski stopień powiązania zależnych od siebie procedur. W idealnym przypadku mamy do czynienia z komponentami w jasny sposób dzielącymi odpowiedzialność. Jeśli kod rozprasza powiązane ze sobą procedury, jego konserwacja staje się utrudniona, ponieważ wprowadzanie zmian wiąże się z identyfikacją i wyszukiwaniem rozległych zależności.

Nasza klasa `ParamHandler` zbiera wszystkie procedury związane z obsługą plików parametrów we wspólnym kontekście. Metody operujące na plikach XML dzielą kontekst, w ramach którego mogą dzielić również dane i w ramach którego zmiany jednej metody (np. zmiana nazw elementów formatu XML) mogą być w razie konieczności łatwo odzwierciedlone w pozostałych metodach. Klasy hierarchii `ParamHandler` cechują się więc wysoką spójnością.

Z kolei podejście proceduralne rozdziela powiązane procedury. Kod obsługi XML jest rozproszony pomiędzy słabo powiązanymi funkcjami.

Sprzęganie

O ścisłym sprzęganiu mówimy wtedy, kiedy oddzielne części kodu systemu są ze sobą związane tak, że zmiany w jednej z nich wymuszają zmiany w pozostałych. Tego rodzaju sprzęganie jest charakterystyczne dla kodu proceduralnego z racji jego sekwencyjnej natury.

Sprzęganie to widać też dobrze w naszym przykładzie z podejściem proceduralnym. Funkcje `wri teParams()` i `readParams()` wykonują na nazwie pliku ten sam test mający na celu wykrycie formatu pliku parametrów i sposobu jego obsługi. Wszelkie zmiany w logice, jakie wprowadzilibyśmy w jednej z funkcji, musielibyśmy zaimplementować również w drugiej. Gdybyśmy, na przykład, zamierzali rozszerzyć obsługę plików parametrów o nowy format pliku, musielibyśmy zsynchronizować zmiany w obu funkcjach, tak aby obie w ten sam sposób realizowały test rozszerzenia pliku. Konieczność ta staje się jeszcze bardziej uciążliwa w miarę wzrostu liczby funkcji związanych z obsługą parametrów.

W przykładzie podejścia obiektowego rozdzieliliśmy od siebie poszczególne klasy pochodne, izolując je również od kodu użytkującego. W obliczu potrzeby uzupełnienia obsługi plików parametrów o nowy format pliku dodalibyśmy po prostu do hierarchii nową klasę pochodną, a jedyną tego reperkusją byłaby konieczność zmiany logiki testu w pojedynczej metodzie klasy bazowej — `getInstance()`.

Ortogonalność

Ortogonalność będziemy tu rozumieć (za Andrew Huntem i Davidem Thomasem i ich publikacją *The Pragmatic Programmer*, Addison-Wesley Professional, 1999) jako połączenie ściśle zdefiniowanej odpowiedzialności komponentów współzależnych z ich niezależnością od szerszej widzianego systemu.

Ortogonalność promuje możliwość ponownego wykorzystywania komponentów przez łatwość ich włączania do nowych systemów bez konieczności specjalnego przystosowywania ich w tym celu. Takie komponenty mają ściśle i w sposób niezależny od szerszego kontekstu zdefiniowane zbiory danych wejściowych i wyjściowych. Kod ortogonalny ułatwia wprowadzanie zmian, bo ogranicza oddźwięk zmian wprowadzanych w implementacji komponentów. Wreszcie kod ortogonalny jest bezpieczniejszy, ponieważ tak samo jak zakres oddźwięku zmian ograniczony jest zakres oddźwięku ewentualnych błędów. Dla porównania błąd w kodzie cechującym się wysoką współzależnością komponentów może obejmować swoimi negatywnymi efektami znaczne obszary systemu.

Nie istnieje coś takiego jak automatyzm wiążący osłabienie sprzęgania i wysoką spójność z zastosowaniem klasy. W końcu równie dobrze moglibyśmy w analizowanym przykładzie ująć całość proceduralnego kodu w pewnej klasie i nie zyskać na takim pseudoobiektywnym podejściu żadnej z typowych dla niego zalet. W jaki więc sposób osiągnąć pożądaną równowagę w kodzie? Osobiście starania rozpoczynam od analizy klas, które miałyby uczestniczyć w systemie.

Zasięg klas

Wyznaczanie granic odpowiedzialności i zakresu funkcji poszczególnych klas systemu okazuje się zaskakująco trudnym zadaniem, zwłaszcza w obliczu rozwoju systemu.

Zadanie to wydaje się proste, kiedy system ma modelować świat materialny. Obiektywne systemy często wykorzystują programowe reprezentacje obiektów świata materialnego — w postaci klas `Person` (osoba), `Invoice` (faktura) czy `Shop` (sklep). Sugeruje to, że wyznaczenie zasięgu klas sprowadza się do rozpoznania i wytypowania w systemie jego „elementów” oraz wyposażenia ich w możliwość wzajemnego oddziaływania za pośrednictwem metod. Nie jest to spostrzeżenie zupełnie nieprawdziwe i stanowi znakomity punkt wyjścia w projektowaniu systemu, nie wolno jednak przyjmować go bezkrytycznie. Jeśli bowiem klasy postrzegać jako rzeczowniki, podmioty dowolnej liczby czynności (czasowników), może się okazać, że w miarę rozwoju projektu i zmian wymagań natłok „czasowników” i zależności pomiędzy „rzeczownikami” jest nie do opanowania.

Wróćmy do pielęgnowanego w rozdziale 3. przykładu hierarchii ShopProduct. Nasz system ma prezentować klientom ofertę produktów, więc wydzielenie w nim klasy ShopProduct jest oczywiste, ale czy to jedyna decyzja, jaką należy podjąć? Klasę uzupełniliśmy o metody getTitle() i getPrice() udostępniające dane produktów. Poproszeni o mechanizm prezentacji zestawienia informacji o produkcie na potrzeby fakturowania i wysyłki moglibyśmy zdefiniować metodę write(). Gdyby okazało się, że zestawienia mają mieć różne formaty, moglibyśmy poza metodą write() wyposażyć naszą klasę również w metody writeXML() i writeXHTML() — albo uzupełnić write() o kod rozpoznający żądany format na podstawie dodatkowego znacznika i dostosowujący sposób prezentacji zestawienia.

Problem w tym, że w ten sposób obarczymy klasę ShopProduct nadmierną liczbą obowiązków — nagle okaże się, że klasa przeznaczona do przechowywania informacji o asortymencie będzie również odpowiadać za sposoby prezentacji tych informacji klientowi.

Jak więc *powinniśmy* podchodzić do zadania definiowania klasy? Najlepiej jest traktować klasy jako jednostki o ściśle ograniczonej odpowiedzialności, ograniczonej najlepiej do pojedynczego i dobrze zdefiniowanego zadania. Kiedy ograniczenie będzie odpowiednie? Kiedy uda się je zgrabnie wyrazić słowami. Przeznaczenie klasy powinno dać się opisać maksymalnie dwudziestoma pięcioma słowami, z rzadka przetykanymi spójnikami „i” czy „lub”. Jeśli opis się wydłuża albo zawiera zbyt wiele zdań podrzędnych, należałoby rozważyć wydzielenie dla części zadań odrębnych klas.

Klasy hierarchii ShopProduct są więc odpowiedzialne za przechowywanie i zarządzanie danymi produktów. Jeśli uzupełnimy je o metody prezentujące zestawieniowe informacje o produkcie w różnych formatach, obciążymy hierarchię nowym zadaniem — odpowiedzialnością za prezentowanie informacji klientom. W rozdziale 3. uniknęliśmy przeładowania klasy, wydzielając do zadania prezentacji osobny typ. Typ ShopProduct pozostał odpowiedzialny jedynie za zarządzanie danymi produktów, zaś do zadania wyświetlania informacji o nich powołałamiśmy klasę ShopProductWriter. Odpowiedzialność jest dalej zawężana w klasach pochodnych obu hierarchii.

-
- **Uwaga** Niewiele reguł projektowych cechuje się stanowczością. Niekiedy widuje się więc w klasie kod zapisujący dane obiektowe w innej, zupełnie niepowiązanej z nią klasie. Zdaje się to naruszać regułę zawężania odpowiedzialności, ale niekiedy najłatwiej o taką implementację, ponieważ metoda zapisująca musi dysponować pełnym dostępem do składowych egzemplarza. Stosowanie lokalnych względem klasy metod do utrwalania danych pozwala też na uniknięcie definiowania równoległych hierarchii klas utrwalających odzwierciedlających hierarchię klas danych — takie zrównoleglenie oznaczałoby przecież tak niepożądane powiązanie elementów systemu. Innym strategiem utrwalania danych obiektów przyjrzymy się w rozdziale 12. Na razie chciałbym po prostu przestrzec przed fanatycznym trzymaniem się reguł projektowych — żaden zbiór wytycznych nie zastąpi analizy konkretnego problemu. Warto więc wymowę reguły projektowej konfrontować z wymową własnych wniosków co do jej zasadności w danym miejscu projektu.
-

Polimorfizm

Polimorfizm, czyli przełączanie klas, to wspólna cecha systemów obiektowych. Zdążyliśmy już jej zresztą doświadczyć.

Polimorfizm polega na utrzymywaniu wielu implementacji wspólnego interfejsu. Brzmi to może zawile, ale w istocie polimorfizm stosowaliśmy już z powodzeniem w przykładach. Potrzeba polimorfizmu jest często sygnalizowana w kodzie nadmierną liczbą instrukcji warunkowych.

Tworząc w rozdziale 3. pierwsze wersje klasy ShopProduct, eksperymentowaliśmy z pojedynczą klasą, próbując pomieścić w niej funkcje pozwalające na zarządzanie nie tylko produktami pojmowanymi ogólnie, ale również całkiem konkretnym asortymentem — płytami CD i książkami. Doprowadziło to do naszpikowania kodu generującego zestawienie informacji o produkcie instrukcjami warunkowymi:

```
function getSummaryLine() {
    $base = "{$this->title} ({$this->producerMainName}, ";
    $base .= "{$this->producerFirstName}";
    if ($this->type == 'książka') {
        $base .= ": liczba stron - {$this->numPages}";
```

```

    } else if ($this->type == 'cd') {
        $base .= ": czas nagrania - {$this->playLength}";
    }
    return $base;
}

```

Instrukcje warunkowe sugerują możliwość wydzielenia dwóch klas pochodnych: `CdProduct` i `BookProduct`. Podobnie w analizowanym niedawno proceduralnym kodzie obsługi plików parametrów obecność instrukcji warunkowych stanowiła pierwszy sygnał struktury obiektowej, do której ostatecznie doszliśmy. Mieliśmy tam powtórzenie tych samych testów w dwóch miejscach kodu:

```

function readParams($source) {
    $params = array();
    if (preg_match("/\.xml$/i", $source)) {
        // odczyt parametrów z pliku XML
    } else {
        // odczyt parametrów z pliku tekstowego
    }
    return $params;
}

function writeParams($params, $source) {
    if (preg_match("/\.xml$/i", $source)) {
        // zapis parametrów do pliku XML
    } else {
        // zapis parametrów do pliku tekstowego
    }
}

```

Każda z instrukcji warunkowych sugerowała potrzebę zdefiniowania klas `XmlParamHandler` i `TextParamHandler` rozszerzających (czy raczej specjalizujących) klasę bazową `ParamHandler` i definiujących jej abstrakcyjne metody `read()` i `write()`:

```

// Może zwrócić obiekt klasy XmlParamHandler bądź TextParamHandler:
$test = ParamHandler::getInstance($file);

$test->read(); // XmlParamHandler::read() albo TextParamHandler::read()...
$test->addParam("klucz1", "wartość1");
$test->write(); // XmlParamHandler::write() albo TextParamHandler::write()...

```

Należy zauważyć, że polimorfizm nie delegalizuje instrukcji warunkowych. Wykorzystywane są one choćby w ramach metody `ParamHandler::getInstance()` celem wyboru odpowiedniej klasy obiektu. Chodzi o to, aby instrukcje decyzyjne były skupione w jednym miejscu kodu i nie musiały być powtarzane w różnych jego fragmentach.

Wiemy, że PHP wymusza definiowanie interfejsów wyznaczanych abstrakcyjnymi klasami bazowymi. To korzystne, bo mamy dzięki temu pewność, że wszystkie konkretne (nie abstrakcyjne) klasy pochodne będą obsługiwały metody o dokładnie takich sygnaturach, jak w abstrakcyjnej klasie nadrzędnej. Dotyczy to również sygnalizacji (wymuszania) typów obiektowych oraz ochrony dostępu. W kodzie klienckim można więc wymiennie stosować wszystkie pochodne wspólnej klasy nadrzędnej (dopóty, dopóki kod kliencki odwołuje się wyłącznie do funkcjonalności zdefiniowanej w klasie bazowej). Od tej reguły jest jeden istotny wyjątek: nie ma możliwości ograniczenia i wymuszenia typu zwracanego przez metodę klasy, niezależnie od definicji metody w klasie nadrzędnej.

■ **Uwaga** W czasie przygotowywania tej książki mówiło się o włączeniu wymuszania typów zwracanych do przyszłych wydań PHP, ale nie było w tym względzie ostatecznego postanowienia.

Niemożliwość określenia typów zwracanych oznacza, że wymiennosc typów pochodnych może ulec zaburzeniu przez zmienność typów wartości zwracanych z metod, w zależności od implementacji klasy. Warto więc narzucić sobie samemu dyscyplinę polegającą na ujednoceniu typów wartości zwracanych. Niektóre z metod mogą zresztą być tak definiowane, aby słabą kontrolę typów, charakterystyczną dla PHP, wykorzystać do zwracania różnych typów wartości w zależności od okoliczności wywołania. Reszta metod zawiera z użytkownikami hierarchii swego rodzaju kontrakt, obietnicę co do typu zwracanego. Jeśli kontrakt ten zostanie zawarty w abstrakcyjnej klasie bazowej, powinien być respektowany w implementacjach wszystkich jej konkretnych pochodnych, aby klienci byli pewni spójnego działania wywołań rozprowadzanych w hierarchii. Jeśli zgodzimy się na zwracanie obiektu pewnego typu, można będzie oczywiście zwrócić specjalizację tego typu w postaci obiektu jego klasy pochodnej. Choć więc interpreter nie może wymusić ujednoczenia typów wartości zwracanych z metod, nie powinno to być usprawiedliwieniem niekonsekwencji programisty. Typ wartości zwracanych z metod należałoby też określać w komentarzach dokumentujących kod.

Hermetyzacja

Hermetyzacja (ang. *encapsulation*) oznacza proste ukrywanie danych i funkcji przed użytkownikiem. To kolejne z kluczowych pojęć podejścia obiektowego.

Na najprostszym poziomie hermetyzacja danych polega na deklarowaniu składowych klas jako prywatnych bądź zabezpieczonych. Ukrywając składowe przed użytkownikami obiektów klas, wymuszamy na nich stosowanie pewnego interfejsu odwołań, zapobiegając tym samym przypadkowym naruszeniom spójności danych obiektów.

Inną formą hermetyzacji jest polimorfizm. Skrywając za wspólnym interfejsem rozmaite jego implementacje, ukrywamy strategię implementacji przed użytkownikami tego interfejsu. Dzięki temu wszelkie zmiany wprowadzane za osłoną interfejsu są dla jego użytkowników transparentne. Oznacza to możliwość dodawania i uzupełniania implementacji interfejsu bez wymuszania zmian w jego stosowaniu po stronie użytkownika. Użytkownik posługuje się wyłącznie interfejsem i nie interesują go skrywające się za nim mechanizmy. Im większa zaś niezależność tych mechanizmów, tym mniejsze ryzyko, że wprowadzane w nich zmiany czy poprawki odbiją się na pozostałych częściach projektu.

Hermetyzacja jest w pewnym sensie kluczem do programowania obiektowego. Naszym celem powinno być maksymalne uniezależnienie poszczególnych elementów systemu. Klasy i metody powinny otrzymywać tylko tyle informacji, ile im potrzeba do wykonywania ich — ściśle ograniczonych i odpowiednio zawężonych — zadań.

Wprowadzenie do języka PHP słów kluczowych `private`, `protected` i `public` znakomicie ułatwia hermetyzację. Hermetyzacja jest jednak również swego rodzaju stanem umysłu projektanta. W PHP4 nie mieliśmy do dyspozycji żadnych formalnych środków ukrywania danych. Prywatność była sygnalizowana jedynie w dokumentacji i konwencji nazewnictwa — symptomem zamierzonej prywatności składowej było na przykład rozpoczynanie jej nazwy od znaku podkreślenia:

```
var $_niedotkac;
```

Wymuszało to staranną kontrolę kodu, gdyż respektowanie tak sygnalizowanej prywatności nie było nijak egzekwowane przez interpreter języka. Co ciekawe, błędy były stosunkowo rzadkie, ponieważ już sama struktura i styl kodu jasno wskazywały na to, które ze składowych powinny być omijane w kodzie użytkującym klasy.

Również w PHP5 możemy złamać tę regułę i odkryć dokładny podtyp obiektu wykorzystywanego w kontekście przełączania klas — wystarczy użyć operatora `instanceof`.

```
function workWithProducts(ShopProduct $prod) {
    if ($prod instanceof CdProduct) {
        // operacje właściwe dla obiektów CdProduct...
    } else if ($prod instanceof BookProduct) {
        // operacje właściwe dla obiektów BookProduct...
    }
}
```

Być może istnieje niekiedy ważny powód do takiego postępowania, ale zazwyczaj nie jest ono dobrze widziane. Zapytując powyżej o konkretny podtyp, tworzymy zależność pomiędzy kodem implementacji interfejsu a kodem ów interfejs wykorzystującym. Jest to o tyle niebezpieczne, że hermetyzacja podtypów implementujących interfejs

ma na celu ich separację od użytkownika między innymi po to, aby dać twórcom implementacji owych podtypów swobodę zmian i poprawek — tutaj konkretnie chodziło zaś o możliwość modyfikowania hierarchii `ShopProduct` bez propagowania zmian do kodu użytkującego tę hierarchię. Powyższy kod eliminuje tę możliwość. Gdybyśmy bowiem z jakichś powodów zdecydowali o zmianie implementacji klas `CdProduct` i `BookProduct`, moglibyśmy zaburzyć zamierzone działanie funkcji `workWithProducts()`.

Z powyższego przykładu wyciągamy dwa wnioski. Po pierwsze, hermetyzacja pomaga w tworzeniu ortogonalnego kodu. Po drugie zaś, stopień, do jakiego hermetyzacja daje się wymusić, a od jakiego może być utrzymana jedynie przy odpowiedniej dyscyplinie projektowej, jest zupełnie bez znaczenia. Hermetyzacja jest bowiem techniką, która powinna znaleźć poszanowanie tak twórców klas, jak i ich użytkowników.

Nieważne jak

Jeśli Czytelnik myśli podobnie jak ja, to wzmianka o problemie wywołuje u niego intelektualny wyścig w poszukiwaniu mechanizmów dających rozwiązanie. Zaczyna się wybór funkcji przydatnych w implementacji rozwiązania, przypominanie sobie co sprytniejszych wyrażeń regularnych i poszukiwanie w repozytorium PEARL tudzież powroty do kodu napisanego wcześniej, a nadającego się do wykorzystania w rozwiązaniu postawionego zadania. Jednak wszystko to należy na etapie projektowania odłożyć na bok. Trzeba oczyścić umysł z mechanizmów i procedur.

Umysł powinien zaprzętać jedynie elementy uczestniczące w docelowym systemie: potrzebne w nim typy i ich interfejsy. Oczywiście wiedza odłożona na bok nie jest zupełnie ignorowana. Wiemy dzięki niej, że klasa otwierająca plik będzie potrzebować ścieżki dostępu, kod komunikujący się z bazą danych będzie musiał utrzymywać nazwy tabel oraz hasła i tak dalej. Główną rolę powinny jednak odgrywać struktury i zależności pomiędzy nimi. Łatwo się później przekonać, że implementacja elegancko wpasowuje się w wyznaczone interfejsy, a całość zyskuje elastyczność pozwalającą na łatwe wymienianie, ulepszanie i rozszerzanie implementacji bez zakłócania wzajemnych zależności komponentów systemu i zaburzenia go jako całości.

Gdy położy się nacisk na interfejs, należy myśleć kategoriami abstrakcyjnych klas bazowych, a nie ich konkretnych pochodnych. Przykład mamy w naszym kodzie odczytującym i zapisującym parametry — tutaj interfejs jest najważniejszym aspektem projektu. Potrzebujemy typu odczytującego i zapisującego pary klucz i wartość. I właśnie to jest podstawowym zadaniem owego typu, a nie faktycznie stosowany nośnik czy środki wykorzystywane w operacjach pozyskiwania i utrwalania danych. Projektujemy ten system na bazie abstrakcyjnej klasy `ParamHandler`, uzupełniając ją potem jedynie konkretnymi strategiami implementacji właściwych operacji odczytu i zapisu plików parametrów. W ten sposób uwzględniamy polimorfizm i hermetyzację od samego początku tworzenia systemu, zyskując ostatecznie możliwość przełączania klas implementacji.

To powiedziawszy, należy przyznać, że od początku wiadomo było, że zaistnieją implementacje klasy `ParamHandler` dla XML-a i plików tekstowych, i że bez wątplenia wpłynęło to na kształt interfejsu. I dobrze, bo nie sposób definiować interfejsów w całkowitym oderwaniu od wiadomych sobie aspektów systemu — doprowadziłoby to najprawdopodobniej do zbytnej ich generalizacji.

Sławna Banda Czworoga (autorzy klasycznej pozycji *Design Patterns*¹) podsumowała tę zasadę zdaniem: „Programuj pod kątem interfejsu, nie implementacji”. Warto je dodać do własnego zbioru złotych myśli.

Cztery drogowskazy

Mało kto nie myli się wcale na etapie projektowania. Większość z nas akceptuje fakt konieczności wprowadzania w przyszłości poprawek, nieuniknionych w miarę zdobywania lepszego rozeznania w rozwiązywanym problemie.

Jasny początkowo kurs poprawek łatwo zmienić w niekontrolowany dryf. Tu nowa metoda, tam dodatkowa klasa — i system chyli się ku upadkowi. Przekonaliśmy się już, że sugestie co do ulepszeń kodu widać często w nim samym. Owe tropy *mogą* wprost sugerować konkretne poprawki albo choćby skłaniać do weryfikacji projektu. W niniejszym podrozdziale spróbuję wyróżnić cztery oznaki mogące świadczyć o konieczności zmian projektowych.

¹ Wydanie polskie: *Wzorce projektowe. Elementy oprogramowania obiektowego wielokrotnego użytku*, Helion, 2010 — przyp. tłum.

Zwielokrotnianie kodu

Zwielokrotnianie kodu jest jednym z cięższych grzechów programowania. Uczucie *déjà vu* przy programowaniu procedury może sygnalizować problem projektowy.

Przyjrzyj się wtedy wystąpieniom powtórzonego kodu. Być może uda się je scalić. Zwielokrotnianie kodu oznacza zasadniczo ściśle powiązanie elementów projektu. Czy zmiana czegoś w jednej procedurze wymaga powtórzenia zmian w podobnych procedurach? Jeśli tak, to może wszystkie je należałoby ująć we wspólnej klasie.

Przemądrzałe klasy

Przekazywanie argumentów pomiędzy metodami może być uciążliwe. Dlaczego nie oszczędzić sobie kłopotu, wykorzystując zmienne globalne? Można wtedy zrezygnować z nużącego przekazywania...

Zmienne globalne mają swoje zastosowania, ale nie należy do ich wykorzystywania pochodzić bezkrytycznie. Przeciwnie, każda zmienna globalna powinna być traktowana wyjątkowo podejrzliwie. Stosując zmienne globalne albo ujmując w klasie wiedzę wykraczającą poza dziedzinę odpowiedzialności tej klasy, kotwiczymy klasę w kontekście tej wiedzy i tym samym zmniejszamy jej uniwersalność — klasa jest uzależniona od kodu pozostającego poza jej kontrolą. A przecież chodzi nam o rozluźnianie, a nie zacieśnianie współzależności pomiędzy klasami a procedurami. Wiedzę używaną w klasie należałoby ograniczać do kontekstu tejże klasy — strategie umożliwiające osiągnięcie tego celu poznasz w dalszej części książki.

Złota rączka

Czy nie każemy klasie wykonywać zbyt wielu zadań? Jeśli tak, spróbuj rozpisać listę tych zadań. Być może niektóre z nich dałoby się wyodrębnić do osobnej klasy.

Obecność przeładowanych zadaniami klas utrudnia wyprowadzanie klas pochodnych. Które z zadań powinny być w ramach pochodnej specjalizowane? A jeśli potrzebna będzie pochodna specjalizująca więcej niż jedno zadanie? Skończ się albo na nadmiernej liczbie pochodnych, albo na dużej liczbie instrukcji warunkowych w hierarchii.

Za dużo warunków

Stosowanie instrukcji `if` i `switch` w kodzie projektu to jeszcze nic złego. Niekiedy jednak obecność takich struktur warunkowych to niemy krzyk o polimorfizm.

Jeśli zorientujesz się, że w ramach jednej klasy wciąż trzeba testować jakiś warunek, a zwłaszcza jeśli test ten trzeba powtarzać w wielu metodach klasy, najprawdopodobniej powinienes rozdzielić klasę na dwie albo więcej klas. Sprawdź, czy struktura kodu warunkowego sugeruje rozróżnianie zadań i czy dałoby się nimi obarczyć osobne klasy. Owe klasy powinny implementować wspólną abstrakcyjną klasę bazową. Może też pojawić się wtedy kwestia przekazywania właściwej klasy do kodu użytkującego tak powstałą hierarchię. Można wtedy wykorzystać niektóre z wzorców projektowych z rozdziału 9., opisujących generowanie obiektów.

Język UML

Jak dotąd projekt wyrażaliśmy jedynie kodem, ilustrując nim koncepcje dziedziczenia czy polimorfizmu.

Miało to swoje zalety, ponieważ język PHP jest naszym — Czytelnika i moim — językiem wspólnym (musi tak być, skoro razem zabrnęliśmy aż tutaj). W miarę rozrastania się naszych przykładów pokazywanie kodu źródłowego przestanie wystarczać. Kilka wierszy kodu nie zawsze daje bowiem właściwy obraz koncepcji.

UML to skrót od *Unified Modeling Language* („ujednolicony język modelowania”). Według Martina Fowlera (autora książki *UML Distilled*, Addison-Wesley Professional, 1999), UML doczekał się rangi standardu dopiero po wieloletnich intelektualnych i biurokratycznych bataliach toczonych przez społeczność zwolenników projektowania obiektowego; stronami byli zwolennicy dobrego i zwolennicy lepszego.

Z pobojowiska wyłoniła się niezwykle przydatna składnia graficznego opisu systemów obiektowych. W tym rozdziale zaledwie muśniemy zagadnienie, wkrótce jednak Czytelnik przekona się, że odrobina języka UML jest w tej książce jak najbardziej na miejscu.

Przydatność UML-a przejawia się głównie w opisach struktur i wzorców, a to za sprawą diagramów klas. Uzyskiwana w tych diagramach przejrzystość intencji projektowych i podziału zadań rzadko daje się równie łatwo wyrazić w przykładowym kodzie.

Diagramy klas

Choć diagramy klas to tylko jeden z wielu elementów języka UML, to właśnie im język zawdzięcza powszechność stosowania. Diagramy te są nieocenione w opisach relacji zachodzących w systemach obiektowych. I właśnie z nich najczęściej będziemy korzystać w niniejszej książce.

Reprezentowanie klas

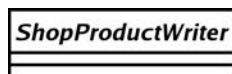
Łatwo się domyślić, że głównymi składnikami diagramów klas są same klasy. Klasa jest reprezentowana na diagramie prostokątem opatrzonym nazwą, jak na rysunku 6.1.



Rysunek 6.1. Klasa na diagramie klas UML

Prostokąt klasy podzielony jest na trzy części; pierwszą z nich zajmuje nazwa klasy. Jeśli piktogram klasy na diagramie nie powinien zawierać niczego poza nazwą, wyróżnianie pozostałych pól w piktogramie klasy nie jest obowiązkowe. Projektując diagram klas, szybko spostrzeżesz, że szczegółowość opisu klasy z rysunku 6.1 jest dla wielu klas wystarczająca. Język UML nie wymaga bowiem wymieniania wszystkich składowych czy metod poszczególnych klas — ba, diagram klas nie musi zawierać kompletu klas projektu!

Klasy abstrakcyjne są wyróżniane albo pochYLENIEM czcionki nazwy (jak na rysunku 6.2), albo umieszczonym poniżej nazwy oznaczeniem {abstrakcyjna} (jak na rysunku 6.3). Forma pierwsza jest popularniejsza, druga zaś lepiej nadaje się do odręcznych notatek.



Rysunek 6.2. Klasa abstrakcyjna na diagramie klas



Rysunek 6.3. Klasa abstrakcyjna na diagramie klas w notacji z ograniczeniem (metką)

-
- **Uwaga** Notacja {abstrakcyjna} jest przykładem notacji charakterystycznej dla „ograniczeń” (ang. *constraint*). Ograniczenia służą na diagramach klas do opisu sposobów, w jakie należy wykorzystywać konkretne elementy diagramu. Nie istnieje przy tym żadna wyróżniona składnia dla tekstu umieszczonego pomiędzy nawiasami klamrowymi — powinien on jedynie wyjaśniać warunki wymagane dla elementu.
-

Interfejsy obrazuje się na diagramie klas tak samo jak klasy, tyle że należy je uzupełnić o stereotyp (element zapewniający rozszerzalność języka UML), jak to zrobiono na rysunku 6.4.



Rysunek 6.4. Interfejs

Atrybuty

Ogólnie rzecz ujmując, atrybuty odwzorowują składowe klas.

Atrybuty klas wymieniane są w polu przylegającym bezpośrednio do pola nazwy klasy — patrz rysunek 6.5.



Rysunek 6.5. Atrybut

Przyjrzyjmy się bliżej określeniu atrybutu z rysunku 6.5. Poprzedzający właściwy atrybut symbol odzwierciedla poziom widoczności, czyli dostępności atrybutu spoza klasy. Można tu zastosować jeden z trzech symboli, których interpretację opisuje tabela 6.1.

Tabela 6.1. Symbole widoczności atrybutów

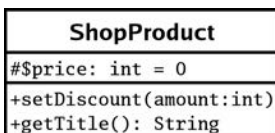
Symbol	Widoczność	Znaczenie
+	Publiczna	Atrybut dostępny ogólnie.
-	Prywatna	Atrybut dostępny wyłącznie w ramach bieżącej klasy.
#	Chroniona	Atrybut dostępny wyłącznie w ramach bieżącej klasy i jej pochodnych.

Za symbolem widoczności podaje się nazwę atrybutu. W naszym przypadku opis dotyczy składowej `ShopProduct::price`. Występujący za nią znak dwukropka oddziela nazwę atrybutu od jego typu (i opcjonalnie podawanej wartości domyślnej).

I znowu: na diagramie umieszczamy tylko to, co jest konieczne do czytelnego zilustrowania danej koncepcji.

Operacje

Operacje reprezentują metody, a mówiąc ściślej, opisują wywołania, jakie można inicjować na rzecz klasy. Na rysunku 6.6 widać element reprezentujący klasę `ShopProduct`, uzupełniony o dwie operacje:



Rysunek 6.6. Operacje

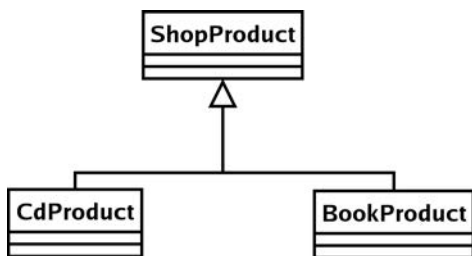
Jak widać, składnia operacji jest zbliżona do składni atrybutów. Nazwę metody poprzedza więc symbol jej widoczności. Operacje wyróżnia ujęta w nawiasy lista parametrów. Za listą parametrów, po znaku dwukropka, określany jest typ wartości zwracanej przez metodę (o ile jest zdefiniowany). Parametry na liście wymieniane są po przecinku, a ich składnia z grubsza odpowiada składni atrybutów — każdy parametr składa się z nazwy, dwukropka i typu.

Jak można się spodziewać, składnia ta jest dość elastyczna. Można na przykład pominąć symbol widoczności czy typ zwracany. Parametry z kolei często reprezentuje się wyłącznie typami (bez nazw) — w większości języków programowania nazwy argumentów przekazywanych w wywołaniu nie mają bowiem żadnego znaczenia.

Relacje dziedziczenia i implementacji

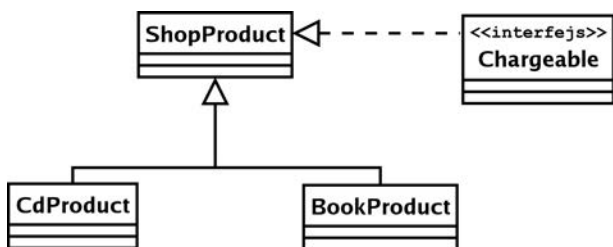
Język UML opisuje relację dziedziczenia jako relację uogólnienia, czyli „generalizacji” klas pochodnych w klasie bazowej². Relacja ta jest reprezentowana na diagramie przez linię wiodącą od klasy pochodnej do klasy bazowej. Linia kończy się zarysem (niewypełnionym) strzałki.

Relację dziedziczenia pomiędzy klasą ShopProduct a jej klasami pochodnymi ilustruje rysunek 6.7.



Rysunek 6.7. Opis relacji dziedziczenia

Relacja pomiędzy interfejsem a klasami implementującymi ten interfejs to w języku UML tzw. relacja „realizacji”. Gdyby więc klasa ShopProduct implementowała interfejs Chargeable, w języku UML wyrazilibyśmy to tak jak na rysunku 6.8.

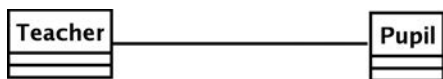


Rysunek 6.8. Opis relacji implementacji

Powiązania

Dziedziczenie to tylko jedna z wielu możliwych relacji, w jakie mogą wchodzić klasy w systemie obiektowym. Kolejną jest na przykład powiązanie (ang. *association*), zachodzące, kiedy składowa klasy przechowuje referencję egzemplarza (albo egzemplarzy) innej klasy.

Relację powiązania pomiędzy klasami Teacher (nauczyciel) i Pupil (uczeń) modelujemy na rysunku 6.9.

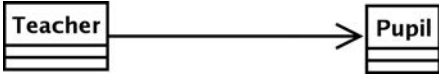


Rysunek 6.9. Powiązanie

² Zatem trochę na opak, bo programiści mówią zazwyczaj o specjalizacji klasy bazowej w klasach pochodnych — *przyp. tłum.*

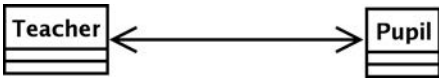
Taki opis nie informuje jednoznacznie o rodzaju powiązania. Wiadomo jedynie, że obiekt klasy `Teacher` będzie przechowywał referencję do jednego bądź wielu obiektów klasy `Pupil` albo odwrotnie — to obiekt klasy `Pupil` będzie przechowywał referencje obiektów klasy `Teacher`. Relacja powiązania może być również dwustronna.

Do określenia kierunku relacji powiązania służą strzałki. Gdyby to obiekt klasy `Teacher` miał przechowywać referencję obiektu klasy `Pupil` (ale nie odwrotnie), powinniśmy poprowadzić strzałkę od klasy `Teacher`, a w kierunku klasy `Pupil`. Takie powiązanie nosi nazwę jednokierunkowego (patrz rysunek 6.10).



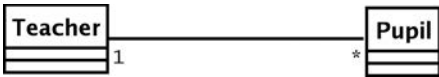
Rysunek 6.10. Powiązanie jednokierunkowe

Gdyby obiekty obu klas przechowywały referencję do obiektów drugiej klasy, relację taką, jako dwukierunkową, należałoby zasygnalizować na diagramie strzałkami w obu kierunkach, jak na rysunku 6.11.



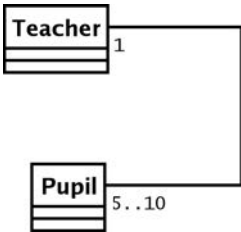
Rysunek 6.11. Powiązanie dwukierunkowe

Można też w relacji powiązania wyszczególnić liczbę egzemplarzy klasy, do której odwołuje się każdy obiekt klasy bieżącej. Czyni się to za pośrednictwem liczb albo zakresów umieszczanych przy prostokątach klas. Jeśli mowa o „dowolnej liczbie egzemplarzy”, należy w miejsce liczby czy zakresu zastosować znak gwiazdki (*). Wedle rysunku 6.12 jeden obiekt klasy `Teacher` przechowuje referencje do nieokreślonej z góry liczby obiektów klasy `Pupil`.



Rysunek 6.12. Definiowanie krotności relacji powiązania

Na rysunku 6.13 widać z kolei, że pojedynczy obiekt klasy `Teacher` będzie powiązany z minimalnie pięcioma, a maksymalnie dziesięcioma obiektami klasy `Pupil`.



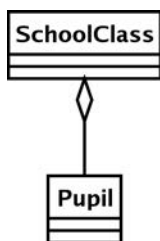
Rysunek 6.13. Definiowanie krotności relacji powiązania

Agregacja i kompozycja

Agregacja i kompozycja to relacje o charakterze zbliżonym do powiązania. Wszystkie one opisują bowiem sytuację, w której klasa przechowuje trwałą referencję do jednego albo wielu egzemplarzy innej klasy. Przy agregacji i kompozycji owe obiekty wchodzi jednak w skład obiektu bieżącej klasy.

W przypadku agregacji obiekt zawierający się w obiekcie danej klasy jest jego nieodłączną częścią, choć może równocześnie być zawierany w innych obiektach. Relacja agregacji jest obrazowana linią rozpoczynającą się symbolem pustego rombu.

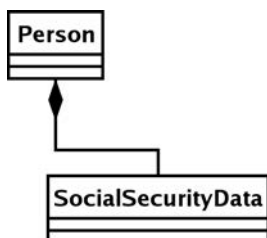
Rysunek 6.14 ilustruje dwie klasy: `SchoolClass` (grupa zajęciowa) i `Pupil` (uczeń). Klasa składa się tu z uczniów.



Rysunek 6.14. Agregacja

Uczniowie tworzą grupę zajęciową, równocześnie poszczególni uczniowie mogą należeć do więcej niż jednej grupy. Odwołanie zajęć grupy nie oznacza więc zwolnienia uczniów do domu — być może mają jeszcze zajęcia w innej grupie.

Kompozycja to zależność jeszcze silniejsza. W kompozycji do obiektu zawieranego może odwoływać się wyłącznie obiekt go zawierający. Relacja kompozycji ilustrowana jest tak samo jak relacja agregacji, jedynie romb jest wypełniany. Relację kompozycji w języku UML ilustruje rysunek 6.15.



Rysunek 6.15. Kompozycja

Klasa Person (osoba) zawiera referencję obiektu SocialSecurityData (dane ubezpieczenia społecznego). Nie może być osoby o więcej niż jednym numerze ubezpieczenia społecznego.

Relacja użycia

Relacja użycia jest w języku UML opisywana jako „zależność”. To najsłabsza z relacji omawianych w tym podrozdziale — nie opisuje bowiem żadnego stałego, a tylko przejściowe powiązanie pomiędzy klasami.

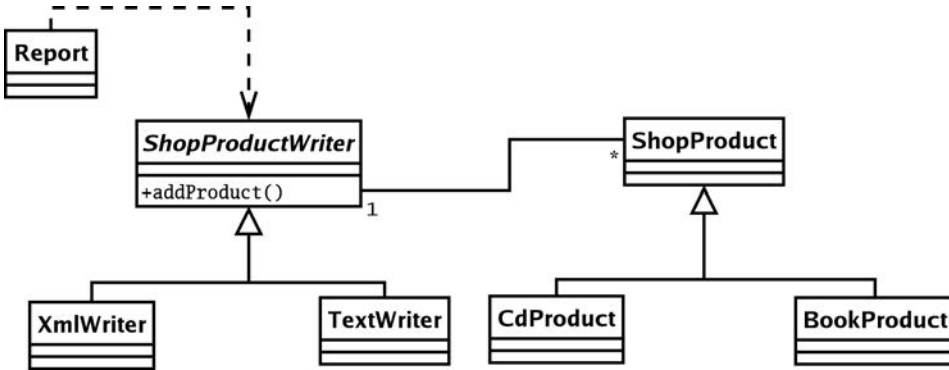
Obiekt klasy używanej może zostać przekazany do klasy używającej za pośrednictwem argumentu wywołania metody, może też zostać pozyskany jako wartość zwracana z wywołania metody.

Z rysunku 6.16 wynika, że klasa Report używa obiektu klasy ShopProductWriter. Relacja użycia jest reprezentowana przerywaną linią i otwartą strzałką łączącą dwie klasy. Nie oznacza to jednak, że klasa Report przechowuje referencję obiektu (czy obiektów) klasy ShopProductWriter; z kolei obiekt klasy ShopProductWriter przechowuje trwale tablicę obiektów klasy ShopProduct.

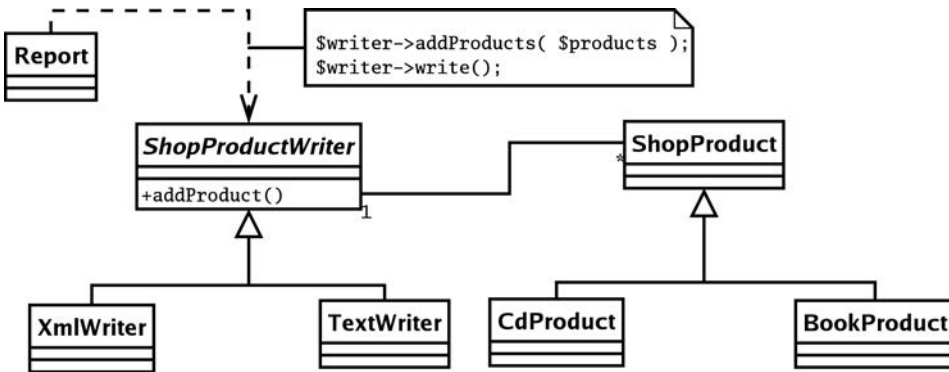
Notki

Diagramy klas mogą dobrze odzwierciedlać strukturę systemu obiektowego, nie dają jednak poglądu na proces odbywający się w systemie.

Rysunek 6.16 ilustruje klasy uczestniczące w naszym systemie. Widać na nim, że klasa Report używa obiektu klasy ShopProductWriter, nie wiadomo jednak, na czym owo użycie polega. Sens tego użycia możemy przybliżyć oglądającemu diagram, umieszczając na nim notki, jak na rysunku 6.17.



Rysunek 6.16. Relacja zależności



Rysunek 6.17. Notka wyjaśniająca charakter zależności użycia

Jak widać, notka to prostokąt z zagiętym narożnikiem. Często zawiera fragmenty pseudokodu.

Dzięki notce widać, że obiekt klasy Report używa obiektu klasy ShopProductWriter do wyprowadzania danych o produkcie. To żadne odkrycie, ale przecież relacje użycia nie zawsze są tak oczywiste, jak w tym przykładzie. Niekiedy nawet notka nie daje wystarczającej ilości informacji. Na szczęście poza modelowaniem samej struktury możemy w języku UML opisywać również interakcje zachodzące w systemie.

Diagramy sekwencji

Diagram sekwencji operuje raczej obiektami niż klasami. Służy do modelowania poszczególnych etapów procesu przebiegającego w systemie.

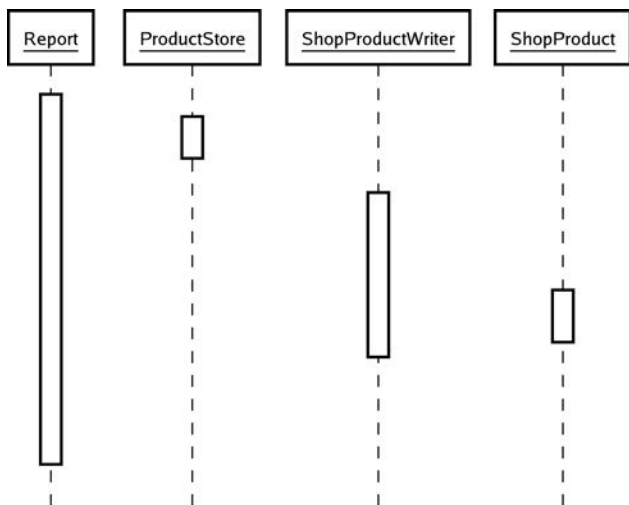
Spróbujmy skonstruować prosty diagram modelujący środki, za pomocą których obiekt klasy Report wypisuje dane o produktach. Diagram sekwencji wymienia w poziomie uczestników systemu, jak na rysunku 6.18.



Rysunek 6.18. Obiekty na diagramie sekwencji

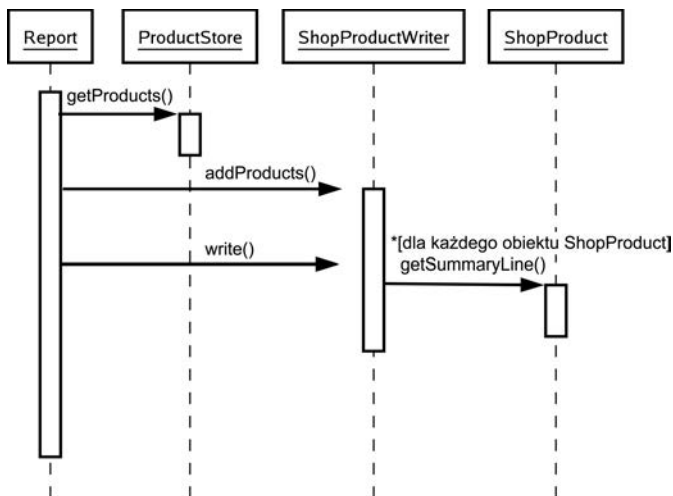
Obiekty oznaczyliśmy na diagramie nazwami ich klas. Gdyby w systemie działało niezależnie wiele egzemplarzy tej samej klasy, moglibyśmy umieścić na diagramie osobne bloki tych obiektów, stosując dla nich etykiety w formacie *obiekt::klasa* (np. `product1::ShopProduct`).

Czas życia obiektów w modelowanym systemie prezentuje się w pionie, jak na rysunku 6.19.



Rysunek 6.19. „Linie życia” obiektów na diagramie sekwencyjnym

Przerwane linie pionowe reprezentują „linie życia” obiektów w systemie. Umieszczone na nich prostokąty reprezentują zaś fakt uczestnictwa obiektów w poszczególnych fazach procesu odbywającego się w systemie. Gdy analizuje się rysunek 6.19 od góry do dołu, widać, jak sterowanie w procesie jest przenoszone pomiędzy obiektami. Przebieg sterowania jest jednak nieczytelny, jeśli diagram nie zawiera komunikatów przekazywanych pomiędzy obiektami. Diagram został więc na rysunku 6.20 uzupełniony stosownymi komunikatami.



Rysunek 6.20. Kompletny diagram sekwencji

Strzałki reprezentują kierunek przesyłania komunikatów pomiędzy obiektami. Wartości zwracane są często na diagramie sekwencji pomijane (choć można je reprezentować liniami przerywanymi prowadzącymi od wywołanego obiektu do inicjatora komunikatu). Każdy komunikat jest etykietowany wywołaniem metody. Etykiety można dobrać dość dowolnie, przyjęta jest jednak pewna prosta składnia. Otóż nawiasy prostokątne reprezentują warunki, więc:

```
[okToPrint]
write()
```

oznacza, że wywołanie metody `write()` jest uzależnione od spełnienia warunku `okToPrint`. Znak gwiazdki oznacza z kolei powtórzenie, którego charakter powinien zostać wyjaśniony w nawiasach prostokątnych:

```
*[dla każdego obiektu ShopProduct]
write()
```

Spróbujmy zatem dokonać analizy procesu implementowanego w systemie, analizując od góry do dołu diagram sekwencji. Na początku obiekt klasy `Report` pozyskuje listę obiektów `ShopProduct`, wywołując stosowną metodę obiektu klasy `ProductStore`. Listę tę przekazuje do obiektu `ShopProductWriter`, który najprawdopodobniej zachowuje referencję obiektów występujących na liście (choć trudno to wywnioskować z samego diagramu). Obiekt klasy `ShopProductWriter` wywołuje następnie dla każdego otrzymanego obiektu metodę `ShopProduct::getSummaryLine()`, prowokując wyprowadzenie na wyjście kompletu danych o wszystkich produktach.

Jak widać, przebieg procesu w systemie da się modelować diagramem sekwencji, dobrze odzwierciedlającym dynamiczne interakcje i prezentującym je w sposób nadspodziewanie czytelny.

-
- **Uwaga** Spójrz na rysunki 6.16 i 6.20. Zauważ, w jaki sposób diagram klas ilustruje polimorfizm, pokazując klasy pochodne klas `ShopProductWriter` i `ShopProduct`. Zwróć teraz uwagę, że szczegół ten stał się niewidoczny przy modelowaniu komunikacji pomiędzy obiektami. Tam, gdzie to możliwe, chcemy bowiem operować na obiektach najbardziej ogólnych z dostępnych typów, ukrywając tym samym szczegóły implementacji.
-

Podsumowanie

W rozdziale przeszliśmy od niskopoziomowych szczegółów programowania obiektowego do kwestii ogólniejszych — projektowych. Przedstawiono pojęcia hermetyzacji, spójności i osłabiania zależności mające zasadnicze znaczenie dla elastyczności systemów obiektowych i możliwości wielokrotnego wykorzystania ich komponentów. Udało się też omówić podstawy języka UML, kładąc tym samym fundament pod omówienie wzorców projektowych stanowiących temat kolejnej części niniejszej książki.

Skorowidz

`$_GET`, 245
`$_POST`, 245
`$_REQUEST`, 198, 245
`$_SESSION`, 237
`$this`, 38, 58, 65
`.htaccess`, 94
`/** */`, 348
`::`, 50, 58
`@author`, 350
`@copyright`, 350
`@license`, 351
`@link`, 354, 355
`@package`, 348, 349, 351
`@param`, 353
`@return`, 353
`@see`, 354, 355
`@uses`, 355
`@var`, 351, 352
`__autoload()`, 95, 96
`__call()`, 73, 76, 77, 102
`__clone()`, 78, 79, 80, 175, 443
`__construct()`, 38, 39, 51, 68, 443
`__destruct()`, 77
`__get()`, 73, 74
`__isset()`, 73, 74
`__NAMESPACE__`, 91
`__set()`, 73, 74, 75
`__sleep()`, 239
`__toString()`, 34, 68, 80, 443
`__unset()`, 73, 75
`__wakeup()`, 239
`==`, 78
`===`, 78
`->`, 35, 37

A

abstract, 61
Abstract Factory, 136, 140, 168, 175, 176, 306
 abstrakcyjne produkty, 169
 abstrakcyjny wytwórca, 169
 Factory Method, 171
 implementacja, 169
 konsekwencje, 171
 problem, 168
 Prototype, 172
abstrakcyjne klasy bazowe, 122
adres IP, 41
agregacja, 127
akcesory, 53, 74
aktualizacja pakietów PEAR, 346
aktualizacja zmian, 363
Alexander C., 136, 138
algorytmy, 207
aliasy nazw, 90
aliasy poleceń, 252
analiza leksykalna, 459
analyzer leksykalny, 459, 466
 AlternationParse, 470, 472
 analyzer agregatowy, 468
 analyzer kompozytowy, 468
 analyzer końcowy, 468
 BooleanAndHandler, 476
 BooleanOrHandler, 476
 CharacterParse, 468
 CollectionParse, 468, 469
 Context, 463
 dopasowania, 468
 dopasowywanie w analizatorze końcowym, 468
 drzewo obiektów, 466
 EBNF, 472
 elementy leksykalne, 459
 EqualsHandler, 476

- analizator leksykalny
 - getState(), 463
 - Handler, 475
 - klasy analizatorów, 472
 - konstruowanie drzewa obiektów, 466
 - MarkParse, 472
 - nextToken(), 463
 - odzworowanie reguły produkcyjnej
 - w drzewie analizatorów, 475
 - Parser, 466
 - Reader, 464
 - RepetitionParse, 469, 472
 - Scanner, 459
 - ScannerState, 463
 - SequenceParse, 472
 - setState(), 463
 - skaner, 459, 463
 - StringLiteralHandler, 476
 - StringLiteralParse, 471
 - VariableExpression, 475
 - wczytywanie kolejnych znaków, 465
 - WordParse, 471
 - wycofanie z prób dopasowań, 469
- anonimowe wywołania zwrotne, 183
- Ant, 402, 430
- any(), 385
- aplikacje korporacyjne, 227
 - Application Controller, 249
 - Data Mapper, 276
 - Domain Model, 270
 - Domain Object Factory, 297
 - Front Controller, 240
 - Helper View, 263
 - Identity Map, 288
 - Identity Object, 300
 - Lazy Load, 295
 - Page Controller, 259
 - przetwarzanie żądań, 229
 - Registry, 231
 - Selection Factory, 306
 - Template View, 263
 - Transaction Script, 266
 - Unit of Work, 291
 - Update Factory, 306
 - warstwa danych, 229, 275
 - warstwa logiki biznesowej, 229, 266
 - warstwa poleceń i kontroli, 229
 - warstwa prezentacji, 229, 240
 - warstwa widoku, 229
 - warstwy, 228
- aplikacje warstwowe, 227
- aplikacje WWW, 21
 - testowanie, 389
- AppController, 254
 - Application Controller, 228, 249
 - AppController, 254
 - Command, 257
 - FrontController, 250
 - implementacja, 250, 251
 - klasa bazowa hierarchii poleceń, 257
 - konkretna klasa polecenia, 258
 - konsekwencje, 259
 - odzworowywanie żądań do poleceń, 249
 - plik konfiguracyjny, 251
 - pozyskiwanie widoków i poleceń, 256, 257
 - problem, 249
 - utrwalanie danych konfiguracyjnych, 253
 - ApplicationHelper, 231, 242, 243
 - ApplicationRegistry, 243, 249, 268
 - archiwum JAR, 393
 - argumenty, 39, 41, 42, 107
 - array, 40
 - array_slice(), 105
 - as, 90
 - asemblerzy obiektów dziedziny, 309
 - asercje, 379, 382
 - assertAttributeSame(), 380
 - assertEquals(), 380
 - assertFalse(), 380
 - assertNotNull(), 380
 - assertNotSame(), 380
 - assertNull(), 380
 - assertRegExp(), 380
 - assertSame(), 380
 - assertThat(), 382
 - assertTrue(), 380, 382
 - assertType(), 380
 - association, 126
 - at(), 385
 - atLeastOnce(), 385
 - atrapy, 383, 384, 397
 - atrybuty klas, 103
 - automatyczna kompilacja, 450
 - automatyczne generowanie dokumentacji, 347
 - automatyczne wczytywanie kodu, 95
 - automatyzacja instalacji, 401
 - automatyzacja testów, 23
 - autorstwo kodu, 317

B

- badanie
 - argumenty metod, 107
 - klasy, 98, 104
 - metody, 99, 106
 - obiekty, 98
 - relacje dziedziczenia, 100
 - składowe, 100
 - wywołania metod, 101

Banda Czworga, 23, 137
 baza danych, 59, 149, 275
 aktualizacja danych, 291
 duplikaty obiektów, 288
 MySQL, 149
 redukcja masowych odwołań do bazy, 295
 relacje, 276
 tabele, 276
 wstawianie danych, 291
 wydajność, 287
 zapytania, 278
 Beck K., 23
 Bergmann S., 378
 bezwzględne nazwy przestrzeni nazw, 90
 biblioteki, 323
 błędy, 31, 66, 260, 262, 320, 388, 450
 boolean, 40
 branch, 361
 brudne obiekty, 292
 buforowanie danych, 248
 bug, 450
 Bugzilla, 450
 build.xml, 402, 403

C

cache, 291
 call_user_func(), 82, 101
 call_user_func_array(), 101, 102
 catch, 69, 71
 centralizacja konfiguracji systemu, 318
 channel.xml, 341
 checkout, 363
 CI, 420
 ciało klasy, 33
 ciąg połączeniowy, 278
 ciągi znaków, 40
 ciągła integracja, 320, 419, 450
 CruiseControl, 427
 dokumentacja, 423
 instalacja projektu, 430
 kompilacje, 426
 kontrola wersji, 421
 phpUnderControl, 422, 427
 pokrycie kodu testami jednostkowymi, 423
 przygotowanie projektu, 421
 standardy kodowania, 424
 Subversion, 421
 testy jednostkowe, 422
 zalety stosowania, 420
 class, 33
 class type hints, 43
 class_exists(), 97
 class_implements(), 100
 clone, 78, 172, 175

CodeSniffer, 424
 Collection, 283, 284
 Command, 222, 247, 248, 251, 257
 implementacja, 222
 inicjator, 222, 224
 odbiorca, 222
 problem, 222
 uczestnicy, 226
 CommandContext, 223
 CommandResolver, 244, 248
 Composite, 140, 179, 188
 diagram klas, 182
 drzewo dziedziczenia klas, 180
 drzewo obiektów, 180
 hierarchia dziedziczenia, 182
 implementacja, 182
 kaskadowy zasięg operacji, 184
 kompozyt, 182
 konsekwencje, 185
 końcówki, 182
 koszt operacji, 187
 liście, 182
 problem, 180
 trwałość, 187
 zalety wzorca, 184
 ConfException, 70
 Config, 328, 330
 connect(), 150
 const, 60
 contains(), 383
 Context, 231
 Continous Integration, 420
 ControllerMap, 253
 CREATE TABLE, 59
 create_function(), 82, 83, 183
 CruiseControl, 427
 artifact directory, 438
 błędy stylu kodowania, 434
 błędy testów, 435
 build.xml, 430, 432
 cc.pid, 429
 CodeSniffer, 433
 config.xml, 430, 431, 432
 definiowanie publikatorów, 432
 dodawanie projektu, 430
 dodawanie zadań kompilacji, 436
 dostęp do kodu źródłowego projektu, 431
 instalacja, 427
 instalacja projektu do integracji ciągłej, 430
 katalogi zestawień elementów, 438
 kompilacja projektów, 430
 konfiguracja ogólna, 432
 konfiguracja projektu, 430
 mechanizm publikacji przez pocztę elektroniczną, 435
 phpUnderControl, 433
 podsumowanie kompilacji, 433

- pokrycie kodu testami, 434
- powiadomienia o błędach, 435
- testy, 432
- testy jednostkowe, 432
- uruchamianie phpUnderControl, 433
- wskaźniki projektu, 434
- zadania kompilacji, 436
- zestawianie błędów kompilacji projektów, 435

current(), 280

CVS, 449

czytelność kodu, 319

D

- dane, 229, 275
- Data Access Object, 276
- Data Mapper, 276, 309, 311
 - ciąg połączeniowy, 278
 - Collection, 284
 - DomainObject, 284
 - HelperFactory, 284
 - implementacja, 276
 - Iterator, 280
 - klasa odwzorowania, 276
 - konsekwencje, 287
 - Mapper, 276, 286, 287
 - obsługa wielu wierszy, 280
 - pozyskiwanie kolekcji, 286
 - pozyskiwanie narzędzi utrwalania danych, 284
 - problem, 276
 - wytwórnia, 284
 - zarządzanie kolekcjami wielowierszowymi, 283
- Data Source Name, 150, 161
- Data Transfer Object, 300
- Decorator, 188, 191
 - implementacja, 190
 - konsekwencje, 193
 - konstruowanie ciągów dekoracji, 192
 - problem, 188
 - wbudowywanie modyfikacji cech w drzewo dziedziczenia, 189
- DeferredEventCollection, 296
- definiowanie
 - destruktor, 77
 - kanały PEAR, 341
 - klasy pochodne, 206
 - składowe, 35
 - testy WWW, 393
- deklaracja
 - interfejsy, 62
 - klasy, 33
 - klasy abstrakcyjne, 61
 - klasy finalne, 72
 - metody, 37
 - metody abstrakcyjne, 61
 - metody statyczne, 57
 - składowe, 35
 - składowe stałe, 60
 - składowe statyczne, 57
 - zmiennne, 40
- dekorator, 188
- delegowanie wywołań, 76
- destruktor, 77
 - __destruct(), 77
- diagnostyka, 69
- diagramy klas, 124
 - #, 125
 - {abstrakcyjna}, 124
 - +, 125
 - agregacja, 127
 - atributy, 125
 - definiowanie krotności relacji powiązania, 127
 - implementacja, 126
 - interfejsy, 125
 - kierunek relacji, 127
 - klasy, 124
 - klasy abstrakcyjne, 124
 - kompozycja, 127, 128
 - metody, 125
 - notki, 128, 129
 - ograniczenia, 124
 - operacje, 125
 - powiązania, 126
 - powiązanie jednokierunkowe, 127
 - relacja implementacji, 126
 - relacja „realizacji”, 126
 - relacja użycia, 128
 - relacja zależności, 129
 - relacje dziedziczenia, 126
 - składowe, 125
 - widoczność atrybutów, 125
- diagramy sekwencji, 129, 130
 - czas życia obiektów, 130
 - kierunek przesyłania komunikatów, 130
 - linie życia obiektów, 130
 - obiekty, 129
- die(), 62
- DocBlock, 348
- dokumentacja, 319, 345, 449, 451
 - brak dokumentacji, 346
 - ciąga integracja, 423
 - generowanie, 347
 - klasy, 348, 349
 - komentarze DocBlock, 348
 - metody, 352
 - nota o sposobie licencjonowania, 351
 - odnośniki, 354
 - pliki, 351
 - składowe, 351
- dokumentowanie kodu, 346
- dokumenty XML, 40, 62, 67
- DokuWiki, 451

Domain Model, 228, 270
 czynności, 270
 implementacja, 271
 klasy, 271
 konsekwencje, 273
 podmioty, 270
 problem, 270
 schemat bazy danych, 271
 Domain Object Assembler, 312
 domain object assemblers, 309
 Domain Object Factory, 297, 312
 implementacja, 298
 konsekwencje, 299
 problem, 298
 Domain Specific Language, 198
 DomainObject, 284
 DomainObjectAssembler, 311
 DomainObjectFactory, 310
 domknięcie, 24, 81
 dopełnienie, 84
 doStatement(), 268, 269
 dostęp do repozytorium Subversion, 360
 dostęp do składowych, 35
 składowe statyczne, 58
 dostosowanie typu argumentu wywołania, 42
 double, 40
 DSL, 198
 DSN, 150, 161, 268, 278
 dynamiczna kompozycja obiektów, 188
 dynamiczne uzupełnianie składowych, 36
 dziedziczenie, 44, 144, 166, 188
 badanie relacji dziedziczenia, 100
 diagramy klas, 126
 hierarchia klas, 145, 146
 klasy abstrakcyjne, 61
 klasy bazowe, 44
 klasy finalne, 72
 klasy nadrzędne, 44
 klasy pochodne, 44, 49
 klasy potomne, 44
 konstruktory, 49, 50
 metody finalne, 72
 problemy, 44
 static, 65
 stosowanie, 48
 UML, 126, 144
 wywołanie metod klasy bazowej, 50
 wywołanie metod przesłoniętych, 52

E

EBNF, 198, 472
 echo, 414
 egzemplarz klasy, 34
 eksportowanie projektu, 369
 elastyczność obiektów, 179

elementy leksykalne, 459
 encapsulation, 121, 149
 equalTo(), 383
 error_reporting, 330
 estetyka kodu, 445
 etykietowanie projektu, 368
 EventCollection, 296
 exactly(), 385
 Exception, 68, 70
 execute(), 108, 268
 exit(), 260
 export(), 102, 104
 Extended Backus-Naur Form, 198
 extends, 49, 64
 eXtreme Programming, 23, 32, 153, 378
 ezcGraph, 429

F

fabryka, 60, 160
 Facade, 193, 266
 implementacja, 195
 konsekwencje, 195
 problem, 193
 punkt dostępu do warstwy czy podsystemu, 195
 Factory Method, 164, 171
 implementacja, 166, 167
 klasy produktów, 166
 klasy wytwórców, 166
 konsekwencje, 167
 powielanie kodu, 167
 problem, 164
 fail(), 380
 false, 40, 41
 feedback form, 153
 FileNotFoundException, 70
 FileSet, 410
 FilterChain, 412
 final, 72
 Finder, 284
 fixture, 379
 fluent interface, 302
 fopen(), 94
 foreach, 280
 Form Interpreter, 29
 format INI, 330
 format wzorca według Bandy Czworoga, 138
 formaty zapisu konfiguracji, 330
 formularze, 29
 formularze zwrotne, 153
 Foswiki, 451
 Fowler M., 136, 137, 230, 232
 Front Controller, 141, 228, 240, 389
 ApplicationHelper, 242
 Command, 247
 CommandResolver, 244

Front Controller

- hierarchia klas poleceń, 241
 - implementacja, 241
 - klasa kontrolera, 241
 - konsekwencje, 248
 - podjmowanie decyzji o sposobie obsługi
 - żądania HTTP, 244
 - polecenia, 247
 - problem, 240
 - strategia wyboru logicznego, 244
 - żądania, 245
- FrontController, 250
- function, 37, 83
- funkcje
- __autoload(), 95, 96
 - array_slice(), 105
 - call_user_func(), 101
 - call_user_func_array(), 101, 102
 - class_exists(), 97
 - class_implements(), 100
 - create_function(), 82, 83
 - die(), 62
 - dostosowanie typu argumentu wywołania, 42
 - exit(), 260
 - fopen(), 94
 - get_class(), 98
 - get_class_methods(), 99
 - get_class_vars(), 100, 102
 - get_declared_classes(), 97
 - get_include_path(), 95
 - get_parent_class(), 100
 - is_a(), 98
 - is_array(), 40, 388
 - is_bool(), 40
 - is_callable(), 82, 83, 99
 - is_double(), 40
 - is_int(), 44
 - is_integer(), 40
 - is_null(), 40
 - is_object(), 40
 - is_resource(), 40
 - is_string(), 40
 - is_subclass_of(), 100
 - method_exists(), 99, 100
 - mysql_connect(), 149
 - mysql_query(), 149
 - print_r(), 99, 279
 - set_include_path(), 95
 - simplexml_load_file(), 41, 67
 - var_dump(), 34, 104
- funkcje anonimowe, 81
- śledzenie zmiennych z zewnętrznego zasięgu, 84
 - tworzenie, 83
 - use, 84
- funkcje pomocnicze, 96
- funkcje składowe, *Patrz* metody

G

- Gamma E., 23
- garbage collection, 77
- generalizacja, 126
- generowanie
- dokumentacja, 345, 347
 - obiekty klas pochodnych, 116
 - zapytania, 306
- generowanie obiektów, 17
- Abstract Factory, 168
 - Factory Method, 164
 - problemy, 157
 - rozwiązania, 157
 - Singleton, 161
- get_class(), 98
- get_class_methods(), 99
- get_class_vars(), 100, 102
- get_declared_classes(), 97
- get_include_path(), 95
- get_parent_class(), 100
- getBacktrace(), 330
- getCode(), 68, 330
- getEndLine(), 107
- getErrorClass(), 331
- getErrorData(), 331
- getFile(), 68
- getFileName(), 107
- getInstance(), 116
- getLine(), 68
- getMessage(), 68, 330
- getMethods(), 110
- getMock(), 384
- getParameters(), 108
- getPrevious(), 68
- getStartLine(), 107
- getTrace(), 68
- getTraceAsString(), 68
- Git, 358, 449
- globalna przestrzeń nazw, 92
- gramatyka języka, 198
- greaterThan(), 383
- greaterThanOrEqual(), 383
- Gutmans A., 30

H

- Helm R., 23
- Helper View, 263, 264
- implementacja, 264
 - konsekwencje, 265
 - problem, 264
- HelperFactory, 284
- hermetyzacja, 121, 149, 444, 445
- hierarchia klas, 48, 54, 145, 146, 182

hint, 43
 historia języka PHP, 29
 Hunt D., 232

I

identicalTo(), 383
 Identity Map, 64, 288, 289, 312
 identyfikacja obiektów, 289
 implementacja, 288
 konsekwencje, 291
 Mapper, 290
 problem, 288
 przechowywanie informacji o obiektach, 288
 Identity Object, 300, 312
 implementacja, 301
 konsekwencje, 305
 problem, 300
 zarządzanie kryteriami zapytań, 301
 if, 123
 Image_GraphViz, 328
 imitacje, 383, 397
 implementacja, 151
 implementacja interfejsu, 63
 implementacja metody abstrakcyjnej, 61
 implements, 63, 64
 include(), 92, 93, 260
 include_once(), 92
 include_path, 94
 infrastruktura testów, 320
 INI, 330
 inline tags, 354
 instalacja, 401, 450
 instalacja CruiseControl, 427
 instalacja pakietów PEAR, 325
 instalacja projektu, 318
 integracja ciągła, 430
 instalatory, 318
 instanceof, 48, 98, 100, 121, 187
 instrukcje warunkowe, 152, 445
 integer, 40
 integracja, 419
 integracja ciągła, 450
 interceptor methods, 73
 interface, 62, 63
 interfejs kaskadowy, 302, 305, 384
 interfejs wiersza poleceń, 197
 interfejs WWW, 197
 interfejsy, 62, 73, 120, 151
 deklaracja, 62
 implementacja, 63
 Iterator, 280
 Reflection, 287
 Interpreter, 197
 implementacja, 198
 problem, 197
 wady, 204

invoke(), 111
 inżynieria oprogramowania, 22
 is_a(), 98
 is_array(), 40, 388
 is_bool(), 40
 is_callable(), 82, 83, 99
 is_double(), 40
 is_int(), 44
 is_integer(), 40
 is_null(), 40
 is_object(), 40
 is_resource(), 40
 is_string(), 40
 is_subclass_of(), 100
 isError(), 330
 isPassedByReference(), 108
 isSubclassOf(), 110
 Iterator, 280, 287
 iterators, 280

J

JAR, 324, 393
 Java, 22, 393, 427
 Java Archive, 324
 JAVA_HOME, 427
 jednostka pracy, 291, 312
 język DSL, 198
 język Java, 22, 393
 język MarkLogic, 198
 język o osłabionej kontroli typów, 40
 język Perl, 29
 język PHP3, 30
 język PHP4, 21, 30
 język PHP5, 21, 31
 język programowania, 22, 197, 198
 język Selenese, 393
 język UML, 123
 Johnson R., 23
 JUnit, 23, 378

K

kanały PEAR, 327
 channel.xml, 341
 definiowanie kanału, 341
 domyślna strona interfejsu kanału, 343
 konfiguracja, 340
 PEAR2_SimpleChannelFrontend, 341
 PEAR2_SimpleChannelServer, 341
 zarządzanie kanałem, 341
 zarządzanie pakietem w kanale, 342
 kaskady operacji wczytywania, 287
 kategoryzacja kodu, 87
 key(), 280

- klasy, 33, 39, 113
 - akcesory, 53
 - badanie, 98, 104
 - badanie relacji dziedziczenia, 100
 - ciało, 33
 - definiowanie składowych, 35
 - deklaracja, 33
 - destruktory, 77
 - diagramy klas, 124
 - dokumentacja, 348, 349
 - dziedziczenie, 44, 144
 - egzemplarze, 34
 - Exception, 68, 70
 - hermetyzacja, 121
 - hierarchia klas, 48, 54
 - implementacja interfejsu, 63
 - jednostki o ściśle ograniczonej odpowiedzialności, 119
 - klasy bazowe, 44, 144
 - klasy nadrzędne, 44
 - klasy o wysokim stopniu wzajemnego powiązania, 149
 - klasy pochodne, 44, 49, 61, 144, 206
 - klasy pomocnicze, 96
 - klasy potomne, 44
 - konstruktory, 38
 - MDB2, 150
 - metoda konstrukcji obiektu, 38
 - metody, 37
 - metody statyczne, 57, 163
 - modyfikatory dostępu, 35
 - nazwy, 88
 - pakiety, 92
 - PDO, 268
 - PEAR_Error, 68, 330
 - PHPUnit_Framework_TestCase, 378
 - PHPUnit2_Framework_TestCase, 380
 - podział na podklasy, 205
 - powiązanie, 126
 - projektowanie, 113
 - Reflection, 102, 103
 - ReflectionClass, 102, 103, 104
 - ReflectionException, 102
 - ReflectionExtension, 102
 - ReflectionFunction, 102
 - ReflectionMethod, 102, 106, 110
 - ReflectionParameter, 102, 107
 - ReflectionProperty, 102
 - ReflectionUtil, 105
 - rozprzęganie, 149
 - składowe, 34
 - składowe stałe, 60
 - składowe statyczne, 57
 - strukturalizacja pod kątem elastyczności obiektów, 179
 - ukrywanie składowych, 53
 - widoczność składowych, 52
 - właściwości, 34, 35
 - zarządzanie dostępem, 52
 - zasięg klas, 118
 - zmiennie koncepcje, 153
- klasy abstrakcyjne, 61, 115, 169
 - deklaracja, 61
 - metody abstrakcyjne, 61
 - PHP4, 62
 - rozszerzanie, 61
- klasy finalne, 72
 - deklaracja, 72
- kod bajtowy, 30
- kod HTML, 22
- kod obiektowy, 114, 445
- kod proceduralny, 114
- kod wielokrotnego wykorzystania, 323
- kolekcje, 287
- kolizje nazw, 88, 90, 161
- komentarze DocBlock, 348, 350
 - @author, 350
 - @copyright, 350
 - @license, 351
 - @link, 354
 - @package, 348, 349, 351
 - @param, 353
 - @return, 353
 - @see, 354
 - @uses, 355
 - @var, 351
 - znaczniki, 348
- kompilacja projektów PHP, 402
- kompilacje, 426
- komponenty programowe, 448
- kompozycja, 127, 128, 144, 147, 149, 179, 188, 207, 447
- kompozyt, 179, 182
- komunikacja z bazą danych, 149
- konfiguracja kanału PEAR, 340
- konfiguracja repozytorium Subversion, 359
- konfiguracja testu, 379
- konkretyzacja obiektów, 34, 159
- konstruktory, 38, 162
 - dziedziczenie, 49, 50
 - PHP4, 39, 51
- konstruowanie drzewa obiektów, 466
- kontrola jakości kodu, 419
- kontrola typów, 40, 43
- kontrola wersji, 317, 357, 421
- kontroler aplikacji, 250
- kontroler fasady, 141, 240
- kontroler strony, 259
- konwencja nazewnictwa, 319
- kopiowanie obiektów, 78
 - __clone(), 78, 80
 - plytka kopia, 79
 - powierzchowna kopia, 79
- korzystanie z pakietu PEAR, 328
- koszty rozwoju projektu, 397

L

Layer Supertype, 268, 273
 Lazy Load, 295, 312
 implementacja, 296
 konsekwencje, 297
 problem, 295
 Lerdorf R., 29
 lessThan(), 383
 lessThanOrEqual(), 383
 libxml_get_last_error(), 71
 liczby, 39, 40
 linie życia obiektów, 130
 listy dystrybucyjne, 450
 Log, 326
 logicalAnd(), 383
 logicalNot(), 383
 logicalOr(), 383
 logika biznesowa, 229, 264, 266
 lokalizowanie obiektów dziedziny, 300

Ł

łańcuchy znaków, 39

M

magiczne metody, 77
 Mailman, 450
 make, 402
 makefile, 402
 mapa tożsamości, 64, 288, 289, 312
 Mapper, 276, 284, 286, 287, 290, 294, 297
 mapy ścieżek projektu, 345
 MarkLogic, 198, 459
 Expression, 200
 gramatyka, 199
 nawiasy, 199
 OperatorExpression, 202
 VariableExpression, 201
 wyrażenia, 199
 matchesRegularExpression(), 383
 MDB2, 150
 MDB2_Driver_Common, 150
 MDB2_Driver_mysql, 150
 MDB2_Driver_sqlite, 150
 mechanizm ciągłej integracji, 420
 mechanizm obserwacji, 214
 mechanizm późnego wiązania składowych statycznych, 65
 mechanizm sygnalizowania argumentów tablicowych, 44
 mechanizm sygnalizowania oczekiwanego typu, 43
 mechanizm wywołań zwrotnych, 82
 Memcached, 291
 Mercurial, 358
 method_exists(), 99, 100

methodData(), 106
 metoda szablonowa, 202
 metoda wytwórcza, 164
 metody, 37
 __clone(), 78, 79, 80, 175, 443
 __construct(), 38, 39, 51, 443
 __destruct(), 77
 __sleep(), 239
 __toString(), 34, 80, 443
 __wakeup(), 239
 akcesory, 53
 argumenty, 39
 asercje, 379
 badanie, 99, 101, 106
 badanie argumentów, 107
 deklaracja, 37
 destrukторы, 77
 diagramy klas, 125
 dokumentacja, 352
 dostosowanie typu argumentu wywołania, 42
 konstruktory, 38
 metoda konstrukcji obiektu, 38
 metody finalne, 72
 metody magiczne, 77
 metody testujące, 379
 metody z ostrzeżeniami, 62
 odwołanie do egzemplarza klasy, 38
 parametry, 43
 PHP4, 37
 polimorfizm, 120
 typ wartości zwracanej, 68
 widoczność, 37
 wskazówki parametrów, 43
 wywołanie, 37
 metody abstrakcyjne, 61
 implementacja, 61
 metody przechwytyjące, 73
 __call(), 73, 76
 __get(), 73, 74
 __isset(), 73, 74
 __set(), 73, 74, 75
 __unset(), 73, 75
 metody statyczne, 57, 58, 64, 163
 deklaracja, 57
 generowanie obiektów klas pochodnych, 116
 wywołanie, 58
 minijęzyk, 198
 model dziedziny, 270
 model obiektowy, 31
 Module, 108, 110
 modyfikatory dostępu, 35
 montowanie dokumentu kompilacji, 403
 MSSQL, 150
 MySQL, 149, 150
 mysql_connect(), 149
 mysql_query(), 149

N

nadmiar wzorców, 153
 nadużywanie wzorca Singleton, 447
 namespace, 89
 narzędzia, 448
 narzędzia obiektowe, 87
 nazwy, 32, 88, 319

- klasy, 88
- nazwy kwalifikowane, 89
- PEAR, 93
- wzorce projektowe, 136, 137

 never(), 385
 new, 34
 newInstance(), 110, 111
 next(), 280
 niedopasowanie obiektowo-relacyjne, 276
 Nock C., 297, 309
 notacja EBNF, 198
 notifyPasswordFailure(), 385
 notki, 128
 nowi programiści, 345
 null, 40

O

obiekt tożsamości, 300, 312
 obiektowość, 29, 32
 obiekty, 29, 30, 32, 33, 34, 179, 443

- akcesory, 53
- badanie, 98
- destruktory, 77
- diagramy sekwencji, 129
- dostęp do składowych, 35
- dynamiczne uzupełnianie składowych, 36
- generowanie obiektów, 157
- konkretyzacja, 34
- kopiowanie, 78
- metoda konstrukcji obiektu, 38
- metody, 37
- obsługa obiektów, 443
- reprezentacja w ciągach znaków, 80
- składowe, 34
- składowe dynamiczne, 36
- tworzenie, 34
- właściwości, 34, 35
- wypisywanie zawartości, 34

 obiekty danych, 59
 obiekty PDO, 59, 268, 278
 obiekty-atrapy, 229
 object, 40
 Object Mothers, 398
 ObjectWatcher, 292
 Observer, 210, 331

- implementacja, 211, 215
- komunikacja między klasą obserwatora a podmiotem, 215
- SPL, 214

obsługa błędów, 66

- pakiety PEAR, 330

 obsługa danych, 229
 obsługa obiektów, 31, 443
 obsługa PEAR, 324
 obsługa sesji, 234
 obsługa typów argumentów, 41
 obsługa żądań, 240
 odnośniki w dokumentacji, 354
 odpowiedzialność, 117
 odwołanie do plików bibliotecznych, 94
 odwołanie do przesłoniętej wersji metody, 58
 odwzorowanie danych, 276, 311
 odwzorowanie żądań do poleceń, 249
 once(), 385
 operatory

- ::, 50, 58
- ==, 78
- ===, 78
- >, 35
- dostęp do składowej, 35
- instanceof, 48, 98, 121
- new, 34

 opis gramatyki języka, 198
 opóźnione ładowanie, 295, 296
 organizacja obiektów i klas, 179
 organizacja pakietów w konwencji systemu plików, 92
 ortogonalność projektu, 118, 210
 oryginalność kodu, 317
 osadzanie logiki biznesowej w warstwie prezentacji, 264
 osłabianie sprzężenia, 150, 444

P

package.xml, 327, 333, 426

- active, 334
- channel, 334, 338
- contents, 335, 336
- date, 334
- definiowanie plików i katalogów, 335
- dependencies, 338
- description, 334
- dir, 335
- dookreślanie instalacji, 339
- elementy, 334
- email, 334
- file, 335
- identyfikator URI, 334
- informacje o twórcach pakietu, 334
- installconditions, 339
- lead, 334
- license, 335
- name, 334
- nazwa pakietu, 334
- notes, 335

- package, 333, 338
- pearinstall, 338
- php, 338
- phprelease, 339
- release, 335
- required, 338
- role plików pakietu, 335
- składniki pakietu, 333
- stability, 334, 335
- summary, 334
- szczegóły o pakiecie, 334
- task:replace, 337
- time, 334
- typy zależności, 338
- uczestnicy projektu, 334
- uri, 334
- user, 334
- version, 334, 335
- zależności, 337
- zależności opcjonalne, 338
- zależności wymagane, 338
- Page Controller, 228, 259
 - hierarchia klas, 263
 - implementacja, 260
 - konsekwencje, 262
 - PageController, 261
 - powielanie kodu, 263
 - problem, 259
 - widok, 260
 - włączanie widoków, 263
- pakiety, 87, 316
 - symulowanie systemu pakietów na bazie systemu plików, 92
 - wywołanie metod, 89
- pakiety PEAR, 93, 316, 401
 - aktualizacja, 346
 - Config, 328
 - definiowanie plików i katalogów, 335
 - Image_GraphViz, 328
 - informacje o pakiecie, 333
 - instalacja, 325
 - korzystanie, 328
 - Log, 326
 - obsługa błędów, 330
 - package.xml, 333
 - przygotowanie pakietu do dystrybucji, 340
 - role plików, 335
 - składniki pakietu, 333
 - tworzenie, 333
 - wersje PHP, 330
 - zależności opcjonalne, 338
 - zależności pakietów, 326, 328, 337
- pamięć współdzielona, 238
- parametry metody, 43
- parametry żądania, 265
- parent, 50, 52, 58
 - parent::__construct(), 50, 51
 - parsowanie, 459
 - PatternSet, 411
 - PDO, 59, 60, 149, 268, 278
 - prepare(), 269
 - PDOStatement, 286
 - execute(), 269, 279
 - pear, 324
 - PEAR, 31, 93, 141, 316, 323
 - definiowanie kanału, 341
 - instalacja pakietów, 325
 - kanały, 327
 - konfiguracja kanału, 340
 - miejsce instalowania pakietów, 324
 - obsługa błędów, 330
 - package.xml, 327, 333
 - pakiety rdzenia, 324
 - parametry konfiguracyjne, 324
 - PHP Foundation Classes, 324
 - przygotowanie pakietu do dystrybucji, 340
 - repozytorium, 323
 - składniki pakietu, 333
 - tworzenie pakietów, 333
 - wersje PHP, 330
 - zależności pakietów, 326
 - zarządzanie kanałem, 341
 - zarządzanie pakietem w kanale, 342
 - pear channel-discover, 327
 - pear channel-info, 327
 - pear config-get, 324
 - pear config-get php_dir, 324
 - pear config-show, 324
 - pear install, 325, 327, 328, 346
 - pear package, 340
 - pear upgrade, 346
 - PEAR::getCause(), 331
 - PEAR::isError(), 330
 - PEAR::MDB2, 149
 - PEAR_Config, 328
 - PEAR_Error, 68, 330
 - getBacktrace(), 331
 - PEAR_Exception, 331, 332
 - getCause(), 333
 - PEAR2_SimpleChannelFrontend, 341
 - PEAR2_SimpleChannelServer, 341
 - Perl, 29
 - PersistenceFactory, 310
 - Personal Homepage Tools, 29
 - PHAR, 324
 - phar.readonly, 343
 - phing, 403
 - Phing, 401, 402
 - atrybuty elementu copy, 416
 - atrybuty elementu fileset, 411
 - atrybuty elementu input, 417
 - atrybuty elementu patternsset, 412

- Phing
 - atrybuty elementu project, 404
 - atrybuty elementu target, 408
 - build.xml, 403
 - copy, 415, 416
 - delete, 417
 - dokument kompilacji, 403
 - echo, 406, 414
 - env, 409
 - fileset, 410
 - filterchain, 412
 - filtry, 413
 - if, 407
 - input, 416
 - instalacja, 402
 - katalog domowy użytkownika, 409
 - katalog projektu, 409
 - kopiowanie, 415
 - lista zadań, 405
 - montowanie dokumentu kompilacji, 403
 - nazwa projektu, 409
 - operacje, 414
 - override, 408
 - patternset, 411
 - phing.project.name, 409
 - pliki kompilacji, 403
 - pobieranie danych, 416
 - pobieranie pakietu instalacyjnego, 402
 - project, 403
 - project.basedir, 409
 - projecthelp, 405
 - property, 406, 409
 - przekształcanie zawartości plików tekstowych, 412
 - ReplaceTokens, 413
 - różnicowanie zadań kompilacji, 404
 - StripLineBreak, 413
 - TabToSpaces, 413
 - target, 403, 404, 405, 407, 408
 - typy danych, 410
 - unless, 407
 - user.home, 409
 - ustawianie właściwości, 409
 - usuwanie, 417
 - usuwanie znaków nowego wiersza, 413
 - właściwości kompilacji, 406
 - właściwości wbudowane, 409
 - XSLT, 413
 - XsltFilter, 413
 - zadanie kompilacji, 403, 404
 - zastępowanie znaków tabulacji znakami spacji, 413
 - zmienne środowiskowe, 409
- PHP, 22
- PHP Data Object, 149
- PHP Extension and Application Repository, 93, 316
- PHP Foundation Classes, 324
- PHP SPL, 100
- PHP/FI, 29
- PHP/FI 2.0, 29
- PHP_CodeBrowser, 425
- PHP_CodeSniffer, 424
- PHP3, 22, 30, 443
- PHP4, 21, 22, 23, 30, 443
- PHP5, 21, 24, 29, 31, 443
- PHP6, 32
- phpdoc, 347
- phpDocumentor, 319, 345
 - @author, 350
 - @copyright, 350
 - @license, 351
 - @link, 354
 - @package, 349, 351
 - @param, 353
 - @return, 353
 - @see, 354
 - @uses, 355
 - @var, 351
 - dokumentowanie klas, 349
 - dokumentowanie metod, 352
 - dokumentowanie plików, 351
 - dokumentowanie składowych, 351
 - generowanie dokumentacji, 347
 - inline tags, 354
 - instalacja, 346
 - komentarze DocBlock, 348
 - menu dokumentacji, 347
 - nazwy klas, 348
 - tworzenie odnośników w dokumentacji, 354
 - wiersz poleceń, 347
 - znaczniki odnośników osadzanych w tekście, 354
- PHPP Archive, 324
- phpuc, 429
 - graph, 432
 - project, 430
- phpUnderControl, 422, 427, 428, 430, 433
 - instalacja, 428
- PHPUnit, 327, 375, 378, 422
 - asercje, 379, 382
 - atrapy, 383, 384, 385
 - falszywe obiekty, 384
 - imitacje, 383
 - instalacja, 378
 - klasy testujące, 378
 - konfiguracja testu, 379
 - Matcher Methods, 385
 - metody do tworzenia ograniczeń, 383
 - metody testowe, 379
 - metody wytwórcze obiektów dopasowań, 385
 - ograniczenia, 382
 - PHPUnit_Framework_Constraint, 382
 - PHPUnit_Framework_TestCase, 382

- TestCase, 383
- testowanie, 386
- testowanie wyjątków, 380
- testy, 378
- tworzenie przypadku testowego, 378
- uruchamianie zestawów testów, 381
- zarządzanie środowiskiem testu, 384
- PHPUnit_Framework_Constraint, 382
- PHPUnit_Framework_MockObject_Builder_
 - ↳InvocationMocker, 385
- PHPUnit_Framework_MockObject_Matcher_Invocation, 385
- PHPUnit_Framework_MockObject_Stub_Return, 386
- PHPUnit_Framework_TestCase, 378, 382
 - assertThat(), 382
 - getMock(), 384
- PHPUnit_Framework_TestSuite, 422
- PHPUnit2, 378, 449
- PHPUnit2_Framework_TestCase, 380
- PhpWiki, 451
- pinezki, 32
- plik konfiguracji, 251
 - przetwarzanie, 252
 - utrwalanie danych konfiguracyjnych, 253
- pliki
 - .htaccess, 94
 - makefile, 402
 - XML, 67
- plytka kopia obiektu, 79
- podejmowanie decyzji, 222
- podział klasy na podklasy, 205
- pokrycie kodu testami, 423, 434
- polimorfizm, 119, 152, 158, 166, 445
- pomocnik widoku, 264
- Portland Pattern Repository, 138
- powiązanie, 126
 - powiązanie dwukierunkowe, 127
 - powiązanie jednokierunkowe, 127
- powielanie kodu, 123, 167, 323, 444
- powierzchnowa kopia obiektu, 79
- późne wiązanie statyczne, 64
- praca w zespole programistycznym, 315
- prawidła projektowania obiektowego, 141
- Preferences, 161
- prepare(), 269
- prepareStatement(), 268, 269
- prezentacja, 229, 240
- primitive types, 39
- print_r(), 99, 279
- private, 35, 37, 52, 121
- productions, 198
- programowanie, 315
 - programowanie ekstremalne, 23
 - programowanie obiektowe, 32, 114
 - programowanie pod kątem interfejsu, 448
 - programowanie proceduralne, 32, 114
- projekt, 21, 361
 - aktualizacja zmian, 363
 - ciągła integracja, 421
 - dodawanie katalogu, 367
 - dodawanie pliku, 367
 - eksportowanie, 369
 - etykietowanie, 368
 - listy dystrybucyjne, 450
 - przygotowanie do ciągłej integracji, 421
 - rozgałęzianie, 369
 - tworzenie, 361
 - usuwanie katalogów, 368
 - usuwanie pliku, 367
 - zarządzanie wersjami, 357
 - zatwierdzanie zmian, 363
- projektowanie, 23, 122, 315, 447
 - projektowanie aplikacji warstwowych, 227
 - projektowanie kodu, 113
 - projektowanie obiektowe, 113, 149
- properties, 34
 - protected, 35, 37, 52, 121
- Prototype, 172, 175
 - implementacja, 173
 - problem, 173
- przechwytywanie chybionych wywołań, 73
- przechwytywanie wyjątków, 69, 71
- przeglądarka kodu, 425
- przekazywanie argumentów, 123
 - przekazywanie przez referencję, 30
 - przekazywanie przez wartość, 30
- przekazywanie obiektów, 78
 - przekazywanie obiektów pomiędzy metodami, 163
- przemądralne klasy, 123
- przenoszenie projektu do innego środowiska, 318
- przestrzenie nazw, 24, 32, 87, 88
 - __NAMESPACE__, 91
 - aliasy nazwy, 90
 - globalna przestrzeń nazw, 92
 - kolizje nazw, 90
 - poziomy hierarchii, 89
 - składnia z nawiasami klamrowymi, 91
 - tworzenie, 89
 - use, 90
 - zagnieżdżanie przestrzeni nazw, 89
- przetwarzanie pliku konfiguracji, 252
- przygotowanie pakietu PEAR do dystrybucji, 340
- przygotowanie projektu do ciągłej integracji, 421
- przypadki testowe, 376, 378
- przypisanie obiektów, 30, 78
- przypisanie przez referencję, 30
- public, 35, 37, 52, 121
- Pyrus, 323, 324
 - config-show, 325
 - install, 325
 - konfiguracja, 325

R

- RapidSVN, 358
- reagowanie na żądania użytkowników, 222
- realizacja zadań, 197
- redukcja masowych odwołań do bazy danych, 295
- refaktoryzacja, 23
- ReferenceClass::getMethods(), 110
- referencje, 30, 78
- Reflection, 102, 103, 287
 - export(), 103, 104
- Reflection API, 97, 102, 244, 379
 - atrybuty klas, 103
 - badanie argumentów metod, 107
 - badanie klasy, 104
 - badanie metod, 106
 - dostęp do źródła klasy, 105
 - Reflection, 102, 103
 - ReflectionClass, 102, 103, 104, 110
 - ReflectionException, 102
 - ReflectionExtension, 102
 - ReflectionFunction, 102
 - ReflectionMethod, 102, 106, 110
 - ReflectionParameter, 102, 107
 - ReflectionProperty, 102
 - ReflectionUtil, 105
 - stosowanie, 108
- ReflectionClass, 102, 103, 104, 110
 - getEndLine(), 105
 - getFileName(), 105
 - getMethod(), 106, 108
 - getMethods(), 106
 - getName(), 105
 - getStartLine(), 105
 - isAbstract(), 105
 - isInstantiable(), 105
 - isInternal(), 105
 - isSubclassOf(), 110
 - isUserDefined(), 105
 - newInstance(), 110, 111
- ReflectionException, 102
- ReflectionExtension, 102
- ReflectionFunction, 102
- ReflectionMethod, 102, 106, 110
 - getParameters(), 107
 - invoke(), 111
 - methodData(), 106
 - returnsReference(), 107
- ReflectionParameter, 102, 107, 108
 - getClass(), 111
 - getName(), 108
- ReflectionProperty, 102
- ReflectionUtil, 105
 - getClassSource(), 105
 - getMethodSource(), 107
- registerCallback(), 82, 83
- Registry, 228, 231, 232, 389
 - ApplicationRegistry, 237
 - implementacja, 232
 - konsekwencje, 239
 - MemApplicationRegistry, 238
 - obiekty rejestru, 233
 - problem, 231
 - RequestRegistry, 236
 - SessionRegistry, 236
 - testowanie, 234
 - wytwórnie obiektów, 233
 - zasięg, 234, 235
 - zasięg aplikacji, 237
 - zasięg sesji, 236
 - zasięg żądania, 235
- reguły produkcyjne, 198
- rejestr, 231, 232
- rejestrowanie wywołań zwrotnych, 331
- relacja dziedziczenia, 100
- relacja użycia, 128
- relacje, 276
- relacyjne bazy danych, 276
- ReplaceTokens, 413
- repozytorium PEAR, 31, 93, 141, 316, 317, 318, 323
- repozytorium Subversion, 318, 359
- reprezentacja obiektu w ciągach znaków, 80
- reprezentacja zadań, 197
- Request, 264, 389
- RequestHelper, 223
- require(), 92, 93, 94
- require_once(), 91, 92, 93, 94
- resource, 40
- RESTful API, 229
- retrospekcja, 102, 108
- rewind(), 280
- rewizja kodu, 23
- role plików pakietu PEAR, 335
- rozgałęzianie projektu, 369
- rozprzężenie, 149, 171, 444
- rozszerzanie klasy bazowej, 44
- rozszerzona notacja Backusa-Naura, 198
- równoległe instrukcje warunkowe, 152
- RPC, 93
- rzutowanie obiektu na ciąg znaków, 34

S

- Scrum, 378
- SELECT, 279
- Selection Factory, 306, 312
 - implementacja, 306
 - konsekwencje, 309
 - problem, 306
- Selenese, 393

- Selenium, 393
 - akcesory, 394
 - akcje, 394
 - asercje, 394
 - instalacja, 393
 - polecenie testu, 394
 - Selenese, 393
 - tworzenie testu, 393
 - zmiana formatu zapisanego testu jednostkowego, 395
- Selenium IDE, 393
- Selenium RC, 393
- self, 58, 65
- separacja modelu dziedziny od warstwy danych, 273
- serializacja, 238, 239
- Service Layer, 266
- serwer integracji ciąglej, 427
- sesje, 234
 - inicjowanie, 237
- Session Facade, 266
- session_start(), 237
- set_include_path(), 95
- setExpectedException(), 381
- SHM, 238
- silne sprzężenie, 444, 447
- SimpleXml, 67
- SimpleXML API, 41
- simplexml_load_file(), 41, 67, 71
- Singleton, 161, 175, 176, 447
 - ilustracja graficzna, 163
 - implementacja, 162
 - konsekwencje, 163
 - problem, 161
 - stosowanie, 163
- skaner, 459, 463
- składnia wywołania statycznego, 58
- składowe, 34, 35
 - badanie, 100
 - definiowanie, 35
 - diagramy klas, 125
 - dokumentacja, 351
 - dostęp do składowych, 35
 - PHP4, 35
 - składowe dynamiczne, 36
 - składowe stałe, 60
 - składowe statyczne, 57
 - widoczność, 35
- skrypt transakcji, 266
- słowa kluczowe
 - abstract, 61
 - as, 90
 - catch, 69
 - class, 33
 - clone, 78, 172, 175
 - const, 60
 - extends, 49, 64
 - final, 72
 - foreach, 280
 - function, 37, 83
 - implements, 63
 - interface, 62, 63
 - namespace, 89
 - new, 34
 - parent, 50, 52, 58
 - private, 35, 52, 121
 - protected, 35, 52, 121
 - public, 35, 52, 121
 - self, 58, 65
 - static, 57, 64, 65
 - throw, 69
 - try, 69
 - use, 90
 - var, 35, 52
- słownictwo, 140
- SOAP, 229
- specjalizacja klas abstrakcyjnych, 61
- specjalizacja klas wyjątku, 70
- spikes, 32
- SPL, 214, 215
- SplObjectStorage, 214, 215
- SplObserver, 214, 215
- SplSubject, 214, 215
- spójność, 117
- sprzężenie, 118, 149, 444
 - osłabianie sprzężenia, 150
- SQLite, 59, 150
- SSH, 360
- ssh-keygen, 360
- stałe składowe, 60
- Standard PHP Library, 100, 214
- standardy kodowania, 424
 - PHP_CodeSniffer, 424
 - Zend, 424
- static, 57, 64, 65
- strategia, 147
- Strategy, 147, 205
 - implementacja, 206
 - problem, 205
- string, 40
- stringContains(), 383
- struktura dziedziczenia, 145
- strukturalizacja klas pod kątem elastyczności obiektów, 179
- Subversion, 317, 357, 421, 449
 - aktualizacja zmian, 363
 - branch, 361
 - branches, 361, 368
 - checkout, 363
 - dodawanie katalogu, 367
 - dodawanie pliku, 367
 - dodawanie projektu, 361
 - dostęp do repozytorium, 360
 - eksportowanie projektu, 369
 - etykietowanie projektu, 368

Subversion

- gałąź główna, 361, 368
- gałęzie, 361, 368
- importowanie, 361
- instalacja, 358
- komunikaty importu, 362
- konfiguracja repozytorium, 359
- konflikty zmian, 365
- migawki, 361
- organizacja wersjonowania, 361
- projekt, 361
- protokoły komunikacji, 360
- przemieszczanie katalogów, 361
- repozytorium, 318, 359
- rozgałęzianie projektu, 369
- rozwiązywanie konfliktów, 366
- scalanie, 372
- SSH, 360
- struktura katalogów projektu, 361
- tag, 361
- tags, 361, 368
- trunk, 361, 368, 369
- tworzenie repozytorium, 359
- URL, 360
- usuwanie katalogu, 368
- usuwanie pliku, 367
- wyciąganie kopii roboczej projektu, 363
- wyciąganie nowej gałęzi kodu, 370
- wysyłanie nowej wersji pliku do repozytorium, 365
- wysyłanie zmienionego pliku do repozytorium, 364
- zarządzanie wieloma wersjami projektu, 361
- zатwierdzanie zmian, 363

SUnit, 378

Suraski Z., 30

svn, 358, 360

- add, 367, 368
- checkout, 363, 370
- commit, 364, 366, 367, 371
- copy, 368, 370
- export, 369
- import, 361, 421
- list, 369
- merge, 372
- remove, 367, 368
- resolve, 366
- status, 364
- update, 364, 365, 366

svnadmin create, 359

switch, 123

sygnalizowanie argumentów tablicowych, 44

sygnalizowanie błędów, 69

sygnalizowanie oczekiwanego typu, 43

symbole końcowe, 198

symulowanie systemu pakietów na bazie systemu plików, 92

system integracji ciągłej, 450

system kontroli wersji, 317, 357, 358, 421

system obiektowy, 113

System V Shared Memory, 238

szablon widoku, 263

szukanie klasy, 97

Ś

ścieżki przeszukiwania, 93

ściśle sprzężanie, 118, 149

środowisko programistyczne, 425

T

tabele, 276

- kolumny, 276
- wiersze, 276

tablice, 40

Template Method, 202, 276

Template View, 228, 263

- implementacja, 264
- konsekwencje, 265
- problem, 264

terminals, 198

test case, 376

TestCase, 383, 386

- onConsecutiveCalls(), 386
- returnValue(), 386

testowanie, 23, 320, 375, 397, 449

- asercje, 379
- atrapy, 383, 397
- imitacje, 383, 397
- koszt rozwoju projektu, 397
- PHPUnit, 378
- przypadek testowy, 376
- testowanie ręczne, 376
- testowanie systemu, 229
- wyjātki, 380
- zarządzanie środowiskiem testu, 384
- zestaw testów, 381

testowanie aplikacji WWW, 229, 389

- przygotowanie do testów, 389
- przypadki testowe, 391

testy, 320, 419, 420

- testy funkcjonalne, 375
- testy regresyjne, 388
- testy warunków, 445

testy jednostkowe, 375, 376

- atrapy, 383
- ciągła integracja, 422
- imitacje, 383
- ograniczenia, 382
- pokrycie kodu, 423

Thomas D., 232

throw, 69

tokens, 459

Transaction Script, 228, 266
 implementacja, 267
 konsekwencje, 270
 problem, 266
 true, 40, 41
 trunk, 361, 368
 try, 69
 tunele SSH, 360
 tworzenie
 asercje, 382
 dokumentacja, 319
 funkcje anonimowe, 83
 kanały PEAR, 341
 klasy, 33
 klasy pochodne, 49
 minijęzyk, 198
 obiekty, 30, 34, 157
 odnośniki w dokumentacji, 354
 pakiety PEAR, 333
 projekt, 361
 przestrzenie nazw, 89
 przypadki testowe, 378
 repozytorium Subversion, 359
 testy Selenium, 393
 zapytania SQL, 269
 typ bazowy warstwy, 268
 type hinting, 31
 typy abstrakcyjne, 158
 typy argumentów metod, 39
 typy danych, 39, 40
 array, 40
 boolean, 40
 double, 40
 integer, 40
 null, 40
 object, 40
 Phing, 410
 resource, 40
 string, 40
 typy elementarne, 39, 40
 typy obiektowe, 42

U

uchwyty, 40
 udostępnianie obiektów danych, 231
 ukrywanie danych, 121
 ukrywanie implementacji, 149
 ukrywanie składowych, 53
 UML, 123, 124
 diagramy klas, 124
 diagramy sekwencji, 129
 dziedziczenie, 126, 144
 Unified Modeling Language, 124
 unikatowość nazw klas, 88

Unit of Work, 291, 312
 aktualizacja danych, 291
 brudne obiekty, 292
 DomainObject, 293
 implementacja, 291
 konsekwencje, 295
 Mapper, 294
 ObjectWatcher, 292
 problem, 291
 wstawianie danych, 291
 uogólnienie, 126
 Update Factory, 306, 312
 implementacja, 306
 konsekwencje, 309
 problem, 306
 URL, 369
 uruchamianie phpUnderControl, 433
 uruchamianie testów, 420
 zestaw testów, 381
 use, 84, 90
 as, 90
 utrwalanie danych, 275
 utrwalanie danych konfiguracyjnych, 253

V

valid(), 280
 var, 35, 52
 var_dump(), 34, 104
 View Helper, 264
 Visitor, 140, 216
 implementacja, 217
 problem, 216
 wady wzorca, 221
 Vlissides J., 23

W

warstwa danych, 229, 275
 warstwa logiki biznesowej, 229, 266
 warstwa poleceń i kontroli, 229
 warstwa prezentacji, 229, 240
 warstwa trwałości, 311
 warstwa widoku, 229
 warstwy systemu korporacyjnego, 228
 wartości logiczne, 39, 41
 wartość pusta, 40
 wdrażanie, 318
 WebDav, 360
 WHERE, 302
 wiązanie statyczne, 64
 widoczność, 52
 widoczność metod, 37
 widoczność składowych, 35
 widok, 229, 240

- wielokrotne wykorzystanie kodu, 149, 323, 445
- Wiki, 451
- właściwości, 34, 35
- wskazówki parametrów, 43
- współbieżne odwołania do obiektu, 291
- współzależności komponentów projektu, 444
- wyjątki, 68, 380
 - blok kodu chronionego, 69
 - catch, 69, 71
 - ConfigurationException, 70
 - Exception, 68, 70
 - FileException, 70
 - kolejność klauzul catch, 71
 - PEAR_Exception, 331
 - przechwytywanie wyjątków, 69, 71
 - specjalizowanie klasy wyjątku, 70
 - testowanie, 380
 - try, 69
 - XmlException, 70
 - zrzucanie wyjątku, 69
- wykonywanie kopii obiektów, 78
- wykrywanie błędów, 450
- wykrywanie stopnia pokrycia kodu, 423
- wymuszanie typów argumentów, 31, 43
- wyodrębnianie algorytmów, 207
- wypisywanie zawartości obiektu, 34, 80
- wyrażenia regularne, 459
- wyszukiwanie obiektów w bazie danych, 300
- wytwórnia, 60, 64, 160, 284
- wytwórnia obiektów, 140
- wytwórnia obiektów dziedziny, 298, 299
- wytwórnia zapytań, 306
- wywołanie metody, 37
 - metody klasy bazowej, 50
 - metody przesłonięte, 52
 - metody statyczne, 58
 - przesłonięta wersja metody, 58
- wywołanie niekwalifikowane, 89
- wywołanie zwrotne, 81, 82, 83, 331
- wzorce projektowe, 22, 135, 154, 446
 - Abstract Factory, 136, 140, 168
 - Application Controller, 249
 - Banda Czworoga, 137
 - Command, 222
 - Composite, 140, 179
 - Data Access Object, 276
 - Data Mapper, 276, 309, 311
 - Data Transfer Object, 300
 - Decorator, 188
 - Domain Model, 270
 - Domain Object Assembler, 312
 - Domain Object Factory, 297, 312
 - Facade, 193
 - Factory Method, 164
 - format portlandzki, 138
 - format wzorca według Bandy Czworoga, 138
 - Front Controller, 141, 240
 - Helper View, 263
 - Identity Map, 64, 288, 312
 - Identity Object, 300, 312
 - implementacja, 139
 - interakcje, 139
 - Interpreter, 197
 - konsekwencje, 138
 - Layer Supertype, 268
 - Lazy Load, 295, 312
 - nadmiar wzorców, 153
 - nazwy wzorców, 136, 137
 - niewłaściwe stosowanie wzorców, 153
 - niezależność od języka programowania, 139
 - Object Mothers, 398
 - Observer, 210, 331
 - opis problemu, 138
 - Page Controller, 259
 - PHP, 141
 - praktyki projektowe i programistyczne, 140
 - prawidła projektowania obiektowego, 141
 - problem, 138, 139
 - Prototype, 172
 - próbki kodu, 139
 - Registry, 231
 - rozwiązanie, 138, 139
 - Selection Factory, 306, 312
 - Service Layer, 266
 - Singleton, 161, 176
 - słownictwo, 140
 - stosowanie, 139
 - Strategy, 147, 205
 - struktura, 139
 - Template Method, 202, 276
 - Template View, 263
 - Transaction Script, 266
 - Unit of Work, 291, 312
 - Update Factory, 306
 - Visitor, 140, 216
 - wzorce bazodanowe, 154, 275
 - wzorce elastycznego programowania obiektowego, 179
 - wzorce generowania obiektów, 154, 157
 - wzorce korporacyjne, 154, 227
 - wzorce organizacji obiektów i klas, 154, 179
 - wzorce pokrewne, 139
 - wzorce zadaniowe, 154, 197
 - zakres zastosowań, 139
 - zamysł, 138
 - zasady projektowe, 447
 - znane wdrożenia, 139

X

- Xdebug, 423
- XML, 40, 67
- XML_Feed_Parser, 331, 332

XML_RPC_Server, 93
 XmlException, 70
 XP, 23, 153
 XSLT, 413
 XsltFilter, 413
 xUnit, 378

Z

zadanie kompilacji, 403, 404
 zagnieżdżanie przestrzeni nazw, 89
 zależności pakietów PEAR, 337
 zależność systemu od platformy zewnętrznej, 149
 zapytania SQL, 264, 268, 279
 zarządzanie dostępem do klasy, 52
 zarządzanie grupami obiektów, 180
 zarządzanie kanałem PEAR, 341
 zarządzanie kolekcjami wielowierszowymi, 283
 zarządzanie relacjami zachodzącymi pomiędzy żądaniami,
 logiką dziedziny a prezentacją, 259
 zarządzanie serializacją, 239
 zarządzanie środowiskiem testu, 384
 zarządzanie wersjami, 357, 449
 zarządzanie wieloma wersjami projektu, 361
 zasady projektowe, 447
 zasięg, 32, 35, 234
 zasięg aplikacji, 234
 zasięg klas, 118

zasięg sesji, 234, 236
 zasięg standardowy, 234
 zatwierdzanie zmian, 363
 zautomatyzowane testy, 23
 zawężanie odpowiedzialności klas, 149
 zbieranie nieużytków, 77
 zdatność do wielokrotnego stosowania kodu, 445
 Zend, 424
 Zend Engine, 30
 Zend Engine 2, 24, 444
 Zend Engine 3, 32
 zespoły programistyczne, 315
 zestaw testów, 320, 381
 zintegrowane środowiska programistyczne, 425
 zmiany, 317
 zmienne, 35, 40
 deklaracja, 40
 zmienne globalne, 35, 123, 161, 163, 444
 zmienne lokalne, 35
 zmienne o zasięgu aplikacji, 234
 zmienne koncepcje, 153
 rzucanie wyjątku, 69
 zwielokrotnienie kodu, 115, 123
 zwracanie obiektów przez referencję, 31

Ż

żądania HTTP, 245, 389

PHP. Obiekty, wzorce, narzędzia

PHP jest dowodem na to, że czas potrzebny na opanowanie języka programowania oraz uzyskanie pierwszych efektów wcale nie musi zierać do nieskończoności! Łatwa konfiguracja środowiska programistycznego, tania i ogólnodostępne serwery do umieszczania własnych aplikacji oraz witryn opartych na PHP, a ponadto duża liczba publikacji i chętna do pomocy społeczność użytkowników sprawiły, że język PHP błyskawicznie zdobył uznanie. W ciągu ostatnich lat język ten przeszedł obiektywą rewolucję. Dostęp do zaawansowanych narzędzi, wzrost świadomości oraz zmiany w samym języku wystarczyły, by programiści coraz powszechniej zaczęli stosować techniki obiektywne w tworzeniu rozwiązań PHP.

W trakcie lektury tej książki zostaniesz wprowadzony w świat obiektów w PHP. Poznasz pojęcia ściśle związane z tym podejściem do programowania – klasa, obiekt, metoda, dziedziczenie czy widoczność zmiennych to słowa, które nabiorą dla Ciebie nowego znaczenia. Na kolejnych stronach przeczytasz o tym, jak obsługiwać wyjątkowe sytuacje, korzystać z interfejsów, domknąć i funkcji zwrotnych. Ponadto zdobędziesz wiedzę na temat projektowania obiektywego. Zasada hermetyzacji i diagramy UML staną się dla Ciebie całkowicie jasne. Autor bardzo dużo czasu poświęca wzorcom projektowym w PHP. Dzięki nim Twój kod stanie się przejrzysty, a nawet najtrudniejsze problemy będą zdecydowanie łatwiejsze do rozwiązania. Na sam koniec sprawdzisz, jak najlepiej dokumentować kod, korzystać z dodatkowych bibliotek oraz wykonywać testy jednostkowe. Książka ta stanowi kompendium wiedzy na temat obiektywego programowania w PHP, dlatego musi się znaleźć na półce każdej osoby choć trochę związanej z tym popularnym językiem programowania!

▼ Historia obiektowości w PHP

▼ Elementarz pojęć z programowania obiektowego

▼ Obsługa błędów

▼ Wykorzystanie interfejsów, klas abstrakcyjnych oraz metod statycznych

▼ Projektowanie obiektowe – diagramy UML, hermetyzacja

▼ Wzorce projektowe

▼ Wykorzystanie PEAR i Pylus

▼ Generowanie dokumentacji za pomocą phpDocumentor

▼ Zarządzanie kodem za pomocą Subversion

▼ Przygotowywanie testów jednostkowych

▼ Automatyzacja instalacji

▼ Ciągła integracja kodu

Twórz lepszy, czytelniejszy i wydajniejszy kod w PHP!



Helion

Nr katalogowy: **6208**



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



Zamówienia telefoniczne:
0 801 339900
0 601 339900

Sprawdź najnowsze promocje:

🔴 <http://helion.pl/promocje>

📖 Książki najchętniej czytane:

🔴 <http://helion.pl/bestsellery>

Zamów informacje o nowościach:

🔴 <http://helion.pl/nowosci>

Helion SA

ul. Kościuszki 1c, 44-100 Gliwice

tel.: 32 230 98 63

e-mail: helion@helion.pl

<http://helion.pl>

helion.pl
księgarnia
internetowa

Cena 79,00 zł

ISBN 978-83-246-3026-4



9 788324 630264

Informatyka w najlepszym wydaniu